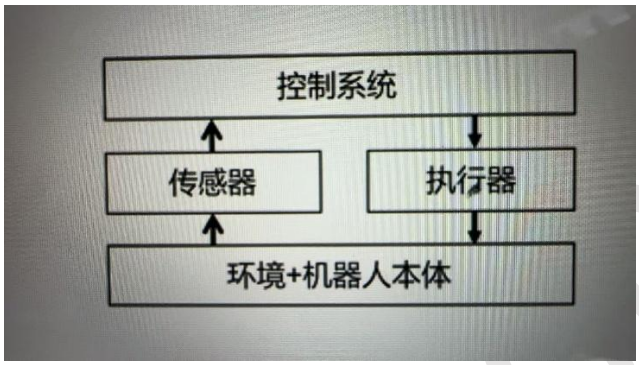
注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

ROS2 机器人仿真与建模

概述：在实际开发过程中，将算法和程序在部署到真正机器人上之前会先把程序算法部署到仿真机器人上以规避可能出现的问题（磕碰损坏……），在进行仿真前要先对机器人进行建模。本章节先从机器人建模开始，创建一个机器人然后对机器人硬件进行仿真，最后将讨论 ROS2_Control 驱动机器人及便捷性。

一． 机器人建模

6.1 移动机器人的结构介绍



执行器 （执行动作）： eg：轮子，刷子

传感器（感知环境）：相机识别地毯，距离传感器识别障碍

控制系统：根据传感器传输的数据控制执行器执行动作

*本书将使用 Gazebo 作为仿真平台

6.2 使用 URDF 创建机器人

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

📘 什么是 URDF（Unified Robot Description Format）

URDF 是 ROS 中用于描述机器人结构的 XML 格式，它不包含行为逻辑，而是定义机器人各部件的几何形状、连接方式、惯性属性等，供仿真、运动规划、可视化使用。URDF 文件定义了一棵**树形结构**，由 link（部件）和 joint（连接）构成，形成完整的机器人模型。

🔧 关键组件概览

组件名	描述	常用子标签	示例
`<link>`	表示机器人的一个刚体部分，如底盘、轮子、手臂	`<visual>`（可视模型）、`<collision>`（碰撞模型）、`<inertial>`（惯性）	`<link name="base_link">...</link>`
`<joint>`	连接两个 link，定义其相对运动方式	`type="revolute"/"fixed"/"prismatic"`，`<origin>`（位置）、`<axis>`（方向）、`<limit>`（限制）	`<joint name="wheel_joint" type="continuous">...</joint>`
`<origin>`	坐标变换：定义 joint/link 相对于父节点的位置和方向（xyz+rpy）	所有 link/joint 都用它来定义变换	`<origin xyz="0 0 0.1" rpy="0 0 0"/>`
`<axis>`	定义关节运动的方向（单位向量）	常见于旋转或线性关节中	`<axis xyz="0 0 1"/>`
`<limit>`	设置关节的旋转角度/移动范围	只对非固定关节有效	`<limit lower="-1.57" upper="1.57" effort="10" velocity="2.0"/>`

编写代码：

打开 vsCode 正常创建目录 （注意这里不需要添加依赖）

Ros2 pkg create firstbot_description --build-type ament_cmake --license Apache-2.0

在 firstbot_description 目录下创建 urdf 目录

键入以下代码 （代码看不懂问 ai； 添加注释快捷键 Ctrl + /）

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```xml
<?xml version="1.0"?>
<robot name='First_robot'>
    <!-- 机器人的身体部分 -->
    <link name="base_link">
        <!-- 部件的外观描述 -->
        <visual>
            <!-- 沿着自己几何中心的偏移和旋转量 -->
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <!-- 几何形状 -->
            <geometry>
            <!-- 圆柱体 半径0.1 高度0.12 -->
                <cylinder radius="0.10" length="0.12"/>
            </geometry>
            <!-- 材质颜色的描述 -->
            <material name = "white">
                <color rgba="1.0 1.0 1.0 0.5"/>
            </material>
        </visual>
    </link>
    <!-- 机器人IMU部件，惯性测量传感器 -->
    <link name="imu_link">
        <!-- 部件的外观描述 -->
        <visual>
            <!-- 沿着自己几何中心的偏移和旋转量 -->
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <!-- 几何形状 -->
            <geometry>
                <!-- 正方体 （单位是米） -->
                <box size="0.02 0.02 0.02"/>
            </geometry>
            <!-- 材质颜色的描述 -->
            <material name = "black">
                <color rgba="0.0 0.0 0.0 0.5"/>
            </material>
        </visual>
    </link>

    <!-- 机器人的关节 用于组合机器人的部件 -->
    <joint name="imu_joint" type="fixed">
        <parent link="base_link"/>
        <child link="imu_link"/>
        <!-- 关节的偏移和旋转量 -->
        <origin xyz="0.0 0.0 0.03" rpy="0.0 0.0 0.0"/>
    </joint>

</robot>
```

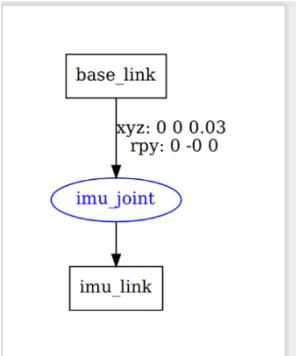
在 urdf 处打开集成终端键入:

urdf_to_graphviz firstbot.urdf

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

最终结果显示：



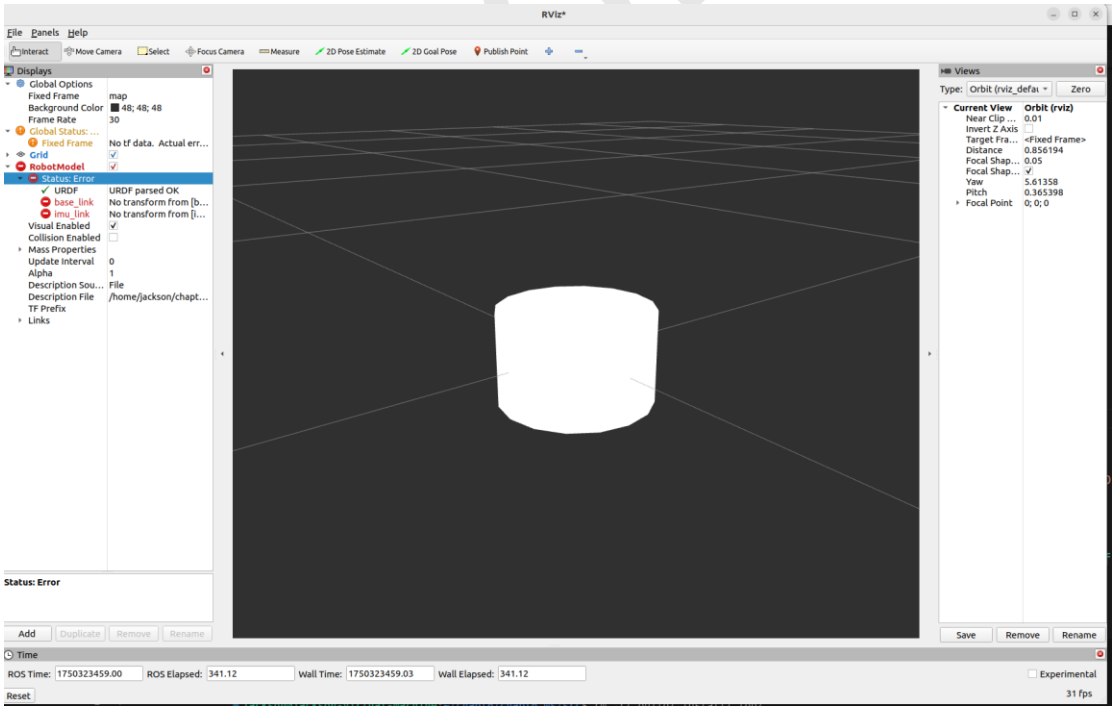
你可以理解为：“在机器人底盘 base_link 上，固定安装了一个 IMU 模块 imu_link，它通过 imu_joint 稳定连接，安装在上方 3cm 处，没有旋转偏移。”

6.3 在 Rviz 中显示机器人

终端输入 rviz2 打开 Rviz

点击下方的 Add 后选择 RobotModel

然后选择 description source 为 File 选择刚才创建编写的 urdf 文件导入 会看到这样 下面我们重点讲解那两个报错的修复！



- **URDF parsed OK** ☑（说明 URDF 文件没语法错误）
- **No transform from [base_link] to [map]** ❌
- **No transform from [imu_link] to [base_link]** ❌

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

根本原因：RViz 没有 TF（坐标变换）发布！模型挂在那儿但没有“位置”概念！

*这里 map 就是 rviz 基本坐标 我们把 fixed frame (最上面)的 map 改成 base_link 就可以解决 base_link to map 问题，现在就是 base_link 和 imu_link 问题，没有发现 joint （下面这段代码 所以不能确定 base 和 imu 位置关系）。

```
<!-- 机器人的关节 用于组合机器人的部件 -->
<joint name="imu_joint" type="fixed">
    <parent link="base_link"/>
    <child link="imu_link"/>
    <!-- 关节的偏移和旋转量 -->
    <origin xyz="0.0 0.0 0.03" rpy="0.0 0.0 0.0"/>
</joint>
```

要让 URDF 模型在 RViz 中正确显示、连上坐标系，你至少要运行两个 ROS 节点：

节点	功能	为什么需要
`joint_state_publisher`	发布所有关节的角度值	URDF 里定义了 joint，没有这个节点你就无法把 joint "激活"
`robot_state_publisher`	根据 joint 状态 + URDF 计算出 TF	这个节点才会让 `base_link` 等坐标出现在 TF 树中，RViz 才能定位

注意：这里视频给的清华源代码可能正常下载不了，用官方的下载（不会问 ai）

因为我们要同时运行两个节点，所以这里创建一个 launch 文件

在 firstbot_description 文件夹里创建 launch 文件夹创建 display_robot.launch.py

键入以下代码

此文件用来加载 URDF 模型并启动 robot_state_publisher、joint_state_publisher 和 RViz 节点。从 URDF 文件中读取机器人模型，通过 robot_state_publisher 广播 TF，再启动 RViz 显示整个模型，提供一个“可动的、带坐标”的机器人骨架可视化环境。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```python
import launch
import launch_ros # 用于启动ROS2节点
from ament_index_python.packages import get_package_share_directory # 用于获取功能包的路径
import os
import launch_ros.parameter_descriptions

def generate_launch_description():
    # 获取默认的URDF文件路径
    urdf_file = get_package_share_directory('firstbot_description') # 功能包的路径
    # 拼接URDF文件的完整路径
    urdf_path = os.path.join(urdf_file, 'urdf', 'firstbot.urdf')

    # 声明一个urdf参数 方便修改路径
    action_declare_arg_model_path = launch.actions.DeclareLaunchArgument(
        name = 'model_path',
        default_value = urdf_path,
        description = 'Path to the robot model file'
    )

    # 通过文件路径获取内容： cat命令  注意cat 这里一定要有空格！！
    robot_description = launch.substitutions.Command(['cat ',launch.substitutions.LaunchConfiguration('model_path')])
    # 并转换成参数值对象，以供传入robot_state_publisher节点
    robot_description = launch_ros.parameter_descriptions.ParameterValue(
        robot_description, # 被转换成参数值的内容
        value_type=str    # 转换成的参数值类型
    )

    # 启动robot_state_publisher节点，注意他需要的是文件内容而不是文件路径
    action_robot_state_publisher = launch_ros.actions.Node(
        package='robot_state_publisher', # 功能包名称
        executable='robot_state_publisher', # 可执行文件名称
        name='robot_state_publisher', # 节点名称
        parameters=[{'robot_description': robot_description}],   # 传入的参数
    )

    action_joint_state_publisher = launch_ros.actions.Node(
        package='joint_state_publisher', # 功能包名称
        executable='joint_state_publisher', # 可执行文件名称
        name='joint_state_publisher', # 节点名称
    )

    action_rviz_node = launch_ros.actions.Node(
        package='rviz2', # 功能包名称
        executable='rviz2', # 可执行文件名称
        name='rviz2', # 节点名称
    )

    return launch.LaunchDescription([
        action_declare_arg_model_path,
        action_robot_state_publisher,
        action_joint_state_publisher,
        action_rviz_node,
    ])
```

在 CMakesLists 里新键入以下代码

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

ament_package()
```

三件套：colcon build （在 chapt6_ws 下）

Source install/setup.bash

ros2 launch firstbot_description display_robot.launch.py

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

最后可以发现 rviz2 被打开，但是这里打开是全新的，我们只需要按照前面说的配置 rviz2 即可 把 map 改成 base_link 发现没有报错了 因为 Launch 文件同时启动了 robot_state_publisher 和 joint_state_publisher 了，TF 和角度位置信息 joint 都被成功加载。在 add 里选择 RobotModel 选择 description topic 中的 robot_description 即可发现模型已经创建成功（包括 imu 黑色小模块）

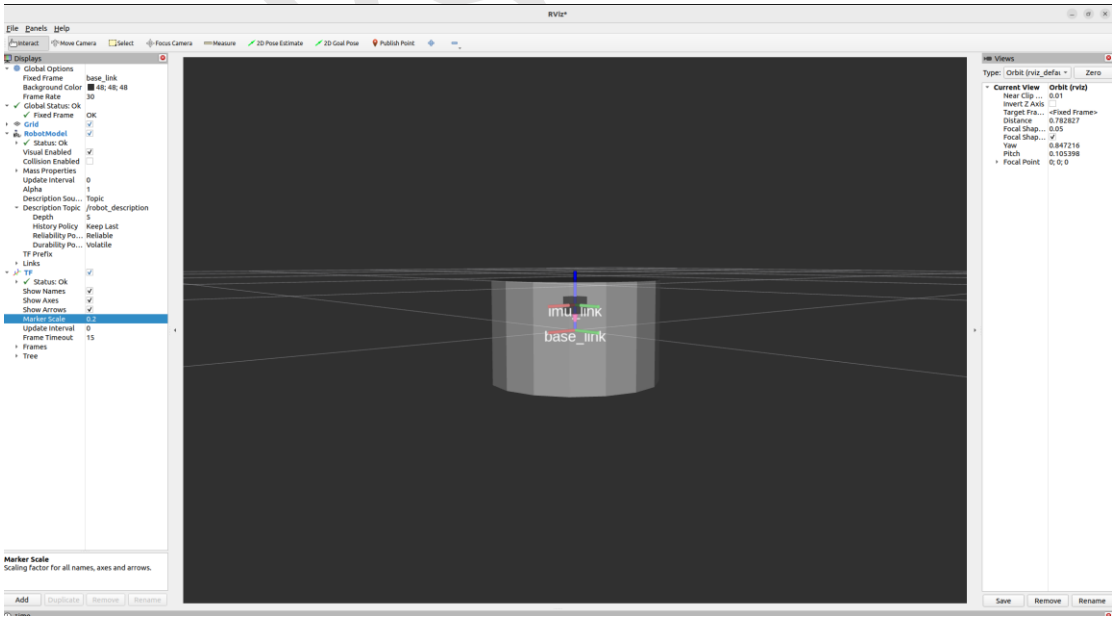
注：

Description Topic 是 RobotModel 插件从 ROS 系统中“获取机器人 URDF 描述字符串”的通道。

它必须匹配你的 robot_state_publisher 中发布/注册的参数名，通常是 robot_description

RViz 中的 RobotModel 插件并不会自己解析 URDF 文件，而是通过订阅某个 Topic 来获取机器人模型的 XML 描述字符串（robot_description），这个 topic 名字就是 launch 文件中的参数 robot_description.

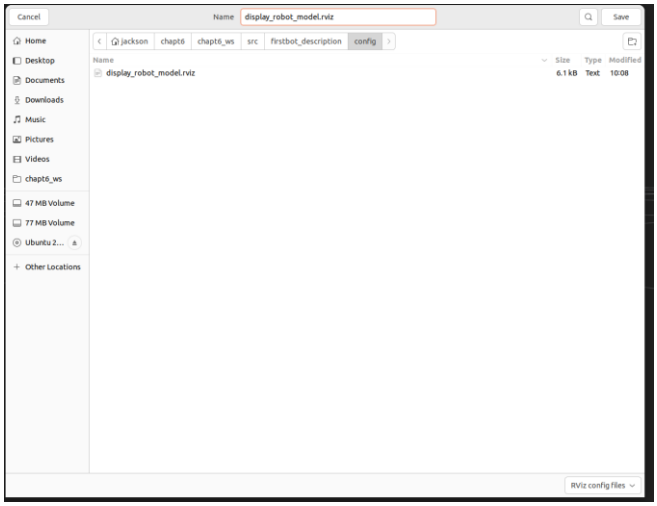
我们现在在 add 中选择 TF，勾选择 show_names，再调整 Marker Scales 调整为 0.2 即可完成配置了。



点击 File， Save File as Config，然后在下面目录下（没有就创建文件夹）保存即可。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可



回到 vsCode 那个 launch 文件下，我们想一启动一运行 launch 文件就自动显示刚才配置的文件，所以要在里面加两段代码：

```
def generate_launch_description():
    # 获取默认的URDF文件路径
    urdf_file = get_package_share_directory('firstbot_description') # 功能包的路径
    # 拼接URDF文件的完整路径
    urdf_path = os.path.join(urdf_file, 'urdf', 'firstbot.urdf')
    default_rviz_config_path = os.path.join(urdf_file, 'config', 'display_robot_model.rviz')
```

```
action_rviz_node = launch_ros.actions.Node(
    package='rviz2', # 功能包名称
    executable='rviz2', # 可执行文件名称
    name='rviz2', # 节点名称
    arguments=['-d', default_rviz_config_path],# parameter和 arguments的区别：parameter是传入参数，arguments是传入命令
)
```

重新构建后打开即可发现不需要我们配置参数，已经加载成功了！

6.4 使用 Xacro 简化 URDF

Xacro （XML Macro）是基于 XML 的宏语言，用于简化 URDF 文件的创建与维护。使用它可以将部件定义为宏，在需要时调用即可。

Xacro（XML Macro） 是 ROS 中专门用来简化和增强 URDF 写法的一个工具。
URDF 是纯 XML，不支持变量、循环、宏调用等 ➜ 写大型机器人模型时非常冗长重复。

> **Xacro = URDF + 变量 + 宏 + 条件语句 ➜ 更灵活更易维护**

（基本上和函数很像，传参那些可以避免写死代码，灵活性高、在很多部件上很有用）

在 urdf 目录下（即 firstbot.urdf 同级目录下创建 firstbot.xacro）键入以下代码

代码分析：直接复制 firstbot.urdf 中代码过来调整即可 可以看到几乎不用改太多 和函数的参数声明传参很像

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

first_robot.xacro × display_robot.launch.py

src > fishbot_description > urdf > first_robot.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="first_robot">
    <xacro:macro name="base_link" params="length radius">
        <!-- 机器人身体部分 -->
        <link name="base_link">
            <!-- 部件外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状-->
                <geometry>
                <!-- 圆柱体 半径 高度 -->
                    <cylinder radius="${radius}" length="${length}"/>
                </geometry>
                <material name='white'>
                    <!-- 材质颜色 -->
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>
    </xacro:macro>
    <xacro:macro name="imu_link" params="imu_name xyz">
        <!-- 机器人的IMU部件 惯性测量传感器 -->
        <link name="${imu_name}_link">
            <!-- 部件外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状-->
                <geometry>
                    <box size="0.02 0.02 0.02"/>
                </geometry>
                <material name='white'>
                    <!-- 材质颜色 -->
                    <color rgba="0.0 0.0 0.0 0.5"/>
                </material>
            </visual>
        </link>

        <!-- 机器人关节 用于组合机器人部件 -->
        <joint name="${imu_name}_joint" type="fixed">
            <parent link="base_link"/>
            <child link="${imu_name}_link"/>
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
        </joint>

    </xacro:macro>

    <xacro:base_link length="0.12" radius="0.1"/>
    <xacro:imu_link imu_name="imu_up" xyz="0.0 0.0 0.03"/>
    <xacro:imu_link imu_name="imu_down" xyz="0.0 0.0 -0.03"/>
</robot>
```

但是 xacro 并不能直接传入 rviz 中，我们要把它转化成 urdf 才行。

在 launch 文件里把原来 cat 那里改成 xacro，另外如果现在运行发现 rviz 模型还是原来的，我们要把 model_path 换成 xacro 文件的绝对路径在命令行终端参数传入即可

下载 xacro 命令：sudo apt install ros-$ROS_DISTRO-xacro

运行 xacro 文件：xacro + 绝对路径

运行大致结果图（部分：）

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```
<?xml version="1.0" ?>
<!-- =================================================================================== -->
<!-- |    This document was autogenerated by xacro from /home/jackson/chapt6/chapt6_ws/src/firstbot_description/urdf/firstbot.xacro | -->
<!-- |    EDITING THIS FILE BY HAND IS NOT RECOMMENDED                                   | -->
<!-- =================================================================================== -->
<robot name="First_robot">
  <!-- 机器人的身体部分 -->
  <link name="base_link">
    <!-- 部件的外观描述 -->
    <visual>
      <!-- 沿着自己几何中心的偏移和旋转量 -->
      <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.0"/>
      <!-- 几何形状 -->
      <geometry>
        <!-- 圆柱体 半径0.1 高度0.12 -->
        <cylinder length="0.12" radius="0.1"/>
      </geometry>
      <!-- 材质颜色的描述 -->
      <material name="white">
        <color rgba="1.0 1.0 1.0 0.5"/>
      </material>
    </visual>
  </link>
  <!-- 机器人IMU部件， 惯性测量传感器 -->
  <link name="imu_up_link">
    <!-- 部件的外观描述 -->
    <visual>
      <!-- 沿着自己几何中心的偏移和旋转量 -->
      <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.0"/>
      <!-- 几何形状 -->
      <geometry>
        <!-- 正方体 （单位是米） -->
        <box size="0.02 0.02 0.02"/>
      </geometry>
      <!-- 材质颜色的描述 -->
      <material name="black">
        <color rgba="0.0 0.0 0.0 0.5"/>
      </material>
    </visual>
  </link>
  <!-- 机器人的关节 用于组合机器人的部件 -->
  <joint name="imu_up_joint" type="fixed">
    <parent link="base_link"/>
    <child link="imu_up_link"/>
    <!-- 关节的偏移和旋转量 -->
    <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.03"/>
  </joint>
```

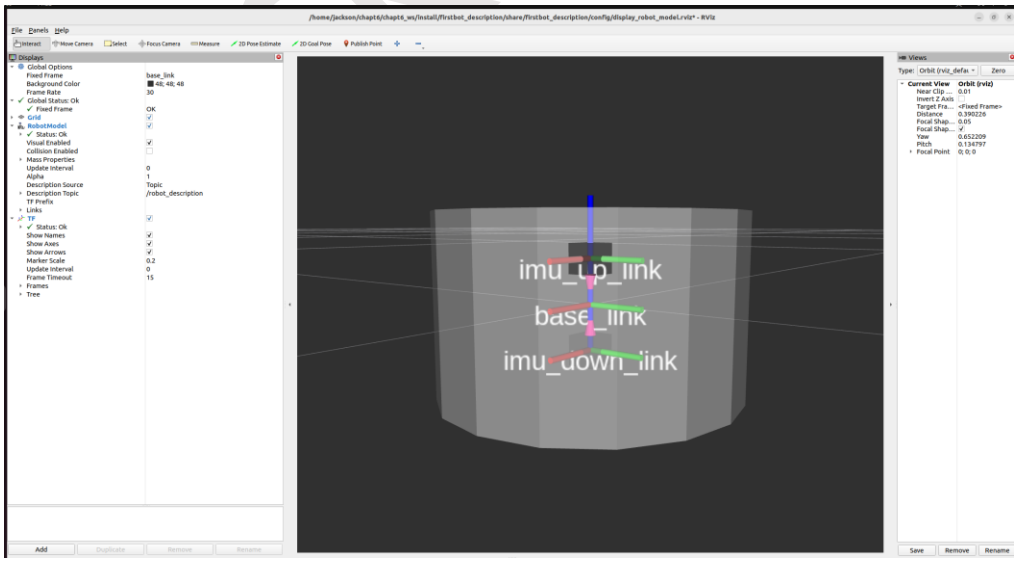
三件套： colcon build

Source install/setup.bash

Ros2 launch firstbot_description display_firstbot.launch.py model_path:= xacro 绝对路径

(注：这里 model_path 是参数名 我和视频里的参数名不一样)

最终结果：



这里还看不出好处简化了的可以试试在下面多传几个参数 imu_middle 这样 10 多个就会发现了，或者我们可以继续往后学习，往仿真扫地机器人里添加更多的元件部件！

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

6.5 创建机器人及传感器部件

本节课我们创建一个机器人模型身体、传感器（imu、相机、雷达）——Jacksonbot

1） 创建身体部件——base

在 urdf 目录下新建目录文件夹 jacksonbot 用于存放这个机器人所有部件

在 jacksonbot 文件夹下创建 base.urdf.xacro

可以选择复制前面的代码 我们保留 base 部分 代码如下

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "base_xacro" params="radius length">
        <!-- 机器人的身体部分 -->
        <link name="base_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                <!-- 圆柱体 半径0.1 高度0.12 -->
                    <cylinder radius="${radius}" length="${length}"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>
    </xacro:macro>
</robot>
```

2) 创建传感器部件——imu

在 jacksonbot 文件夹下新建文件夹 sensor 创建 imu.urdf.xacro

位置与 base 的关联 xyz 设置为参数 将 imu_link 与 base_link 固定连接

键入以下代码:

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "imu_xacro" params="xyz">
        <!-- 机器人IMU部件，惯性测量传感器 -->
        <link name="imu_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 正方体 （单位是米） -->
                    <box size="0.02 0.02 0.02"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "black">
                    <color rgba="0.0 0.0 0.0 0.5"/>
                </material>
            </visual>
        </link>

        <!-- 机器人的关节 用于组合机器人的部件 -->
        <joint name="imu_joint" type="fixed">
            <parent link="base_link"/>
            <child link="imu_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

3） 创建传感器部件——camera

相机部位通过 xyz 参数调整 与 base 固定

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "camera_xacro" params="xyz">
        <!-- 机器人相机部件 -->
        <link name="camera_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 正方体 （单位是米） -->
                    <box size="0.02 0.10 0.02"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "black">
                    <color rgba="0.0 0.0 0.0 0.5"/>
                </material>
            </visual>
        </link>

        <!-- 机器人的关节 用于组合机器人的部件 -->
        <joint name="camera_joint" type="fixed">
            <parent link="base_link"/>
            <child link="camera_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

4） 创建传感器部件——laser

这部分需要两个 link 部件：雷达支撑杆和雷达 雷达放在雷达支撑杆最上端

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "laser_xacro" params="xyz">
        <!-- 雷达支撑杆 -->
        <link name="laser_cylinder_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 正方体 （单位是米） -->
                    <cylinder radius='0.01' length='0.10'/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "black">
                    <color rgba="0.0 0.0 0.0 1.0"/>
                </material>
            </visual>
        </link>

        <!-- 雷达 -->
        <link name="laser_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 正方体 （单位是米） -->
                    <cylinder radius='0.02' length='0.02'/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "green">
                    <color rgba="0.0 1.0 0.0 0.8"/>
                </material>
            </visual>
        </link>

        <!-- 机器人的关节 用于组合机器人的部件 -->
        <joint name="laser_joint" type="fixed">
            <parent link="laser_cylinder_link"/>
            <child link="laser_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="0.0 0.0 0.05" rpy="0.0 0.0 0.0"/>
        </joint>

        <joint name="laser_cylinder_joint" type="fixed">
            <parent link="base_link"/>
            <child link="laser_cylinder_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

(注：这里我按照视频和书分别不同调整颜色参数，所以可能看起来和标准的不一样)

这段代码有点绕，通过观察 joint 可以发现，雷达支撑杆的 xyz 是参数传入的，

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

而雷达的 joint 是绑定雷达支撑杆和雷达的，不是参数，是写死的。

所以马上我们组装后传入的参数实际上是雷达支撑杆相对 base 的位置

接下来我们就需要把他组装在一起　在 jacksonbot 下创建 jacksonbot.urdf.xacro

键入以下代码即可：

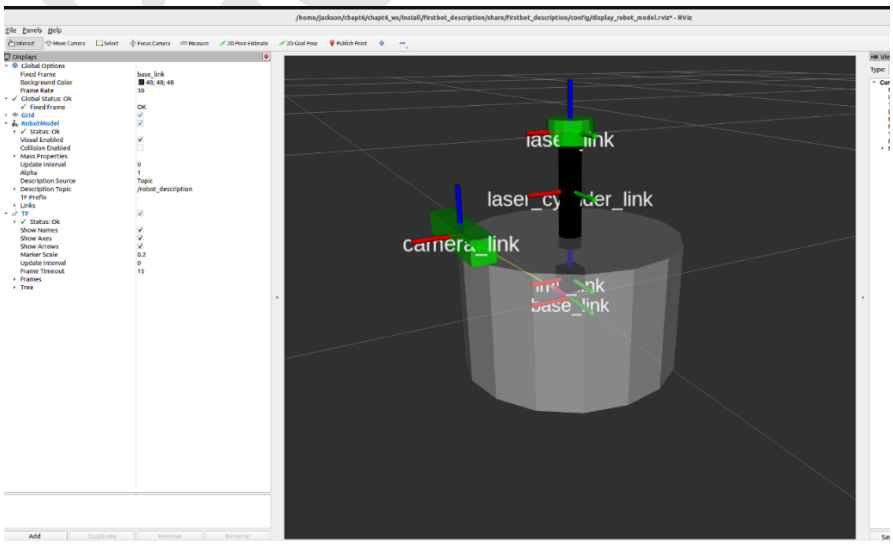
```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name='jacksonbot'>

<!-- 基础部分 -->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/base.urdf.xacro"/>
<!-- 传感器部分 -->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/laser.urdf.xacro"/>
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/camera.urdf.xacro"/>
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/imu.urdf.xacro"/>

<xacro:base_xacro radius="0.1" length="0.12"/>
<xacro:laser_xacro xyz="0.0 0.0 0.10"/>
<xacro:camera_xacro xyz="0.10 0.0 0.075"/>
<xacro:imu_xacro xyz="0.0 0.0 0.02"/>
</robot>
```

完成后三件套：Colcon build

Source install/setup.bash

Ros2 launch firstbot_description display_firstbot.launch.py model_path:= xacro 绝对路径

最终结果：



只有传感器没有执行器，机器人无法运动 接下来----→ 添加执行器

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

6.6 完善机器人执行器部件

在开始前了解两个概念：二轮差速模型 与 万向轮模型

🚗 一、两轮差速模型（Differential Drive）

📌 基本结构：

- **左右两个主动轮**（通常是同规格的马达驱动）
- 可能配有一个或两个**从动轮**或**万向轮**（用于支撑平衡）
- 类似轮椅、扫地机器人结构

mathematica 复制 编辑

```
    ↑前进方向
   ┌─────────┐
   |         |
   |         |
o  |         | o   ← 两个主动轮（左轮 / 右轮）
   |_________|
        o        ← 万向小轮支撑
```

⚙️ 控制原理：

- 机器人通过控制**左右轮子的速度差**来实现转向和前进
  - **左轮 = 右轮** → 直线
  - **左轮 > 右轮** → 向右转
  - **左轮反向 = 右轮正向** → 原地转圈

💡 特点：

优点	缺点
控制简单，模型清晰	无法侧移（不能横着走）
易于建模（支持简化 kinematics）	转向半径受限，不灵活
硬件成本低、易维护	对地面摩擦力较依赖（旋转时易打滑）

🔁 二、万向轮模型（Omni Drive）

> 常见类型包括：**三轮全向轮、四轮麦克纳姆轮（Mecanum Wheel）**

📌 基本结构：

- 每个轮子可以单独驱动，带有斜向滚轮或特殊结构实现**侧向力**
- 可以实现**全向移动（X + Y + θ 三自由度）**

📐 麦克纳姆轮示意（四个斜向小轮）：

css 复制 编辑

```
   ↖     ↗
 [FL]   [FR]    ← 前轮
   ↙     ↘
 [RL]   [RR]    ← 后轮
```

⚙️ 控制原理：

通过 4 个轮子各自的旋转方向和速度组合，可以实现：

动作	轮子运动策略
向前	全部向前转
向右	FL/RL向后，FR/RR向前
原地旋转	左两轮向后，右两轮向前

控制需要进行**逆运动学解算**，并映射到每个轮子的速度。

💡 特点：

优点	缺点
可以原地转动、侧移、斜向移动	控制复杂，需解运动学模型
灵活性高，适合狭小空间导航	对地面要求高（摩擦一致、平整）
更高自由度（XY + 角度）	机械结构复杂、成本高

(RM 车)

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

两个轮子无法保持直立

万向轮 (球体)：支撑　　驱动轮（圆柱）：二轮差速 实现移动

1）创建执行器部件——wheel

在 fishbot 目录下新建 actuator 子目录，在子目录下创建 wheel.urdf.xacro

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "wheel_xacro" params="wheel_name xyz">
        <!-- 机器人IMU部件，惯性测量传感器 -->
        <link name="${wheel_name}_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="1.57079 0.0 0.0"/> <!-- 90度 -->
                <!-- 几何形状 -->
                <geometry>
                    <!-- 正方体 （单位是米） -->
                    <cylinder radius='0.032' length='0.04'/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>

        <!-- 机器人的关节 用于组合机器人的部件 -->
        <joint name="${wheel_name}_joint" type="continuous">
            <parent link="base_link"/>
            <child link="${wheel_name}_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
            <!-- 关节的轴向：这里是绕y轴转动 -->
            <axis xyz="0.0 1.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

参数：轮子名称（因为分为左轮右轮）和位置

轮子默认是躺平状态，所以是绕 x 轴转 90 度（1.57079 rad）让它立起来

在 joint 连接里，调整 type 为 continuous，表示可以绕着某个轴进行无限制旋转 绕 y 轴所以把 y 轴设置为 1.0

2）创建执行器部件—— caster 万向轮

在子目录下创建 caster.urdf.xacro

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "caster_xacro" params="caster_name xyz">
        <!-- 机器人IMU部件，惯性测量传感器 -->
        <link name="${caster_name}_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 球体 （单位是米） -->
                    <sphere radius="0.016"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>

        <!-- 机器人的关节 用于组合机器人的部件 -->
        <joint name="${caster_name}_joint" type="fixed">
            <parent link="base_link"/>
            <child link="${caster_name}_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="${xyz}" rpy="0.0 0.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

接下来我们就需要把他组装在一起在jacksonbot.urdf.xacro 键入以下代码即可：

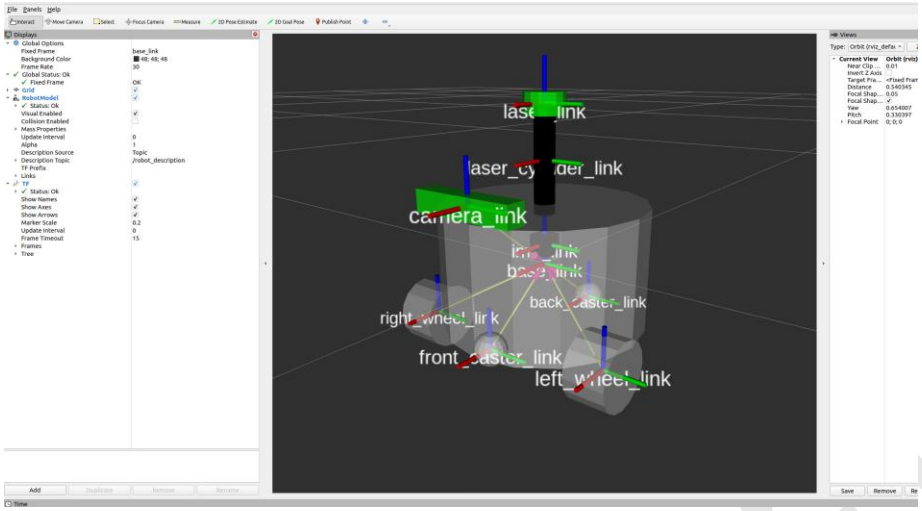
```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name='jacksonbot'>

<!-- 基础部分 -->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/base.urdf.xacro"/>
<!-- 传感器部分 -->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/laser.urdf.xacro"/>
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/camera.urdf.xacro"/>
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/sensor/imu.urdf.xacro"/>

<!-- 执行器部分 -->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/actuator/wheel.urdf.xacro"/>
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/actuator/caster.urdf.xacro"/>

<xacro:base_xacro radius="0.1" length="0.12"/>
<xacro:laser_xacro xyz="0.0 0.0 0.10"/>
<xacro:camera_xacro xyz="0.10 0.0 0.075"/>
<xacro:imu_xacro xyz="0.0 0.0 0.02"/>

<xacro:wheel_xacro wheel_name="left_wheel" xyz="0.05 0.10 -0.06"/>
<xacro:wheel_xacro wheel_name="right_wheel" xyz="0.05 -0.10 -0.06"/>

<xacro:caster_xacro caster_name="front_caster" xyz="0.08 0.0 -0.06"/>
<xacro:caster_xacro caster_name="back_caster" xyz="-0.08 0.0 -0.06"/>
</robot>
```

都保存后三件套（后续改参数的时候不需要 source 但需要 colcon）

最终结果：

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可



整个机器人的所有部件建模到此结束 但我们设置轮子的时候怎么保证他们不会跑到负半轴：添加虚拟部件

6.7 贴合地面 添加虚拟部件

虚拟部件：有 link / joint 但是没有实体、看不见的部件

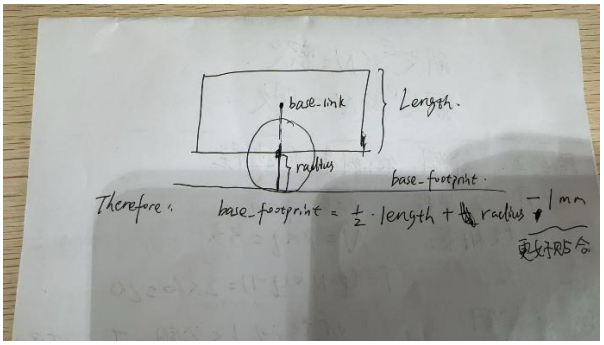
在 base 里添加以下代码

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "base_xacro" params="radius length">
        <!-- 虚拟部件 -->
        <link name="base_footprint"/>

        <!-- 机器人的身体部分 -->
        <link name="base_link">
            <!-- 部件的外观描述 -->
            <visual>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                <!-- 圆柱体 半径0.1 高度0.12 -->
                    <cylinder radius="${radius}" length="${length}"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>

        <joint name="base_joint" type="fixed">
            <parent link="base_footprint"/>
            <child link="base_link"/>
            <!-- 关节的偏移和旋转量 -->
            <origin xyz="0.0 0.0 ${length/2.0+0.032-0.001}" rpy="0.0 0.0 0.0"/>
        </joint>
    </xacro:macro>
</robot>
```

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

这里首先声明 base_footprint 空部件 里面不需要放任何东西 在 joint 里怎么把 base_link 固定在这个上面，高度参数设置可以画图表示一下：

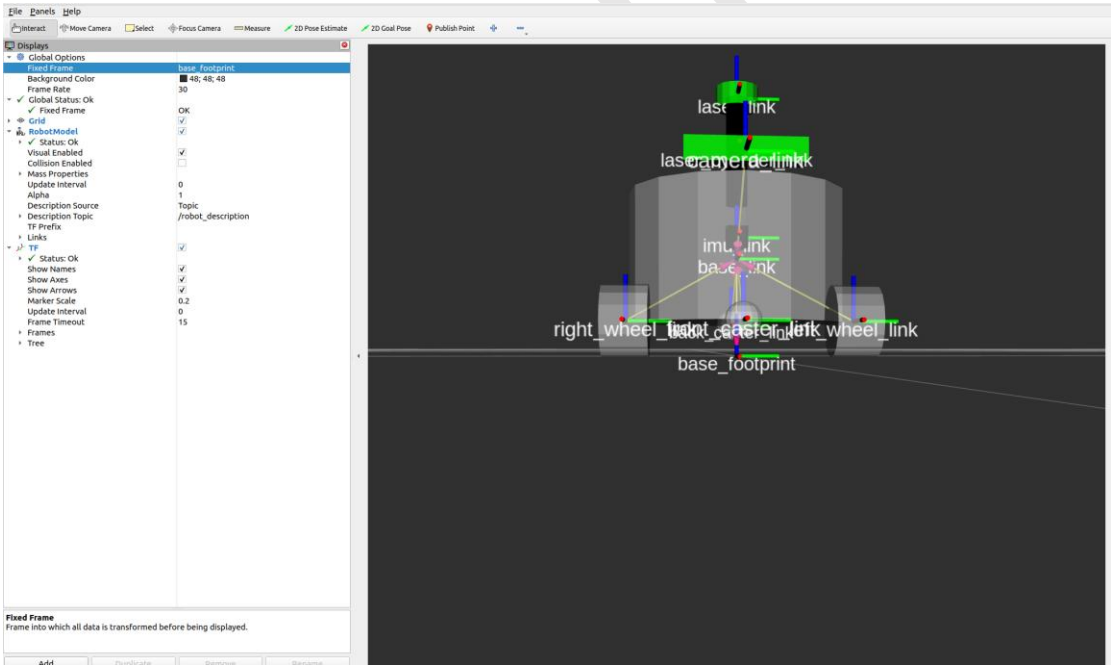


保存后重新三件套

打开 rviz 后注意调整 Fixed Frame 为 base_footprint

（也可以 尝试看看以不同的部件作为基准面的区别）

最终结果：可以看到已经完全紧贴地面了



至此我们完成了对 Jacksonbot 的传感器及执行器的基本建模

在正式开始仿真前我们还需要给机器人添加物理属性

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

二． 机器人的物理属性

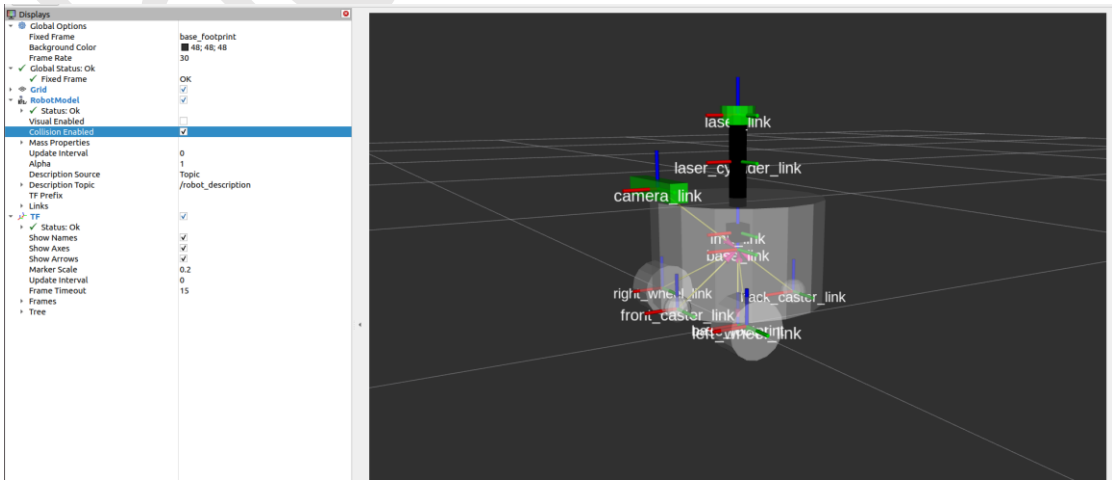
真正机器人各部件一定是有惯性和重量的 关节也是有力矩的 下面我们来为机器人添加这些物理属性

6.8 为机器人添加碰撞属性

在 URDF 中，可以直接在 Link 标签下与 visual 同级标签添加 collision 标签 内容可以与 visual 里的内容一样 然后再构建运行 launch 文件即可 最终效果因为我们设置的是与 visual 一样 所以最终结果应该与原来一模一样

（这里只粘贴 base 的 其他文件一样 如果不会可以用 Copilot 直接生成）

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name = "base_xacro" params="radius length">
        <link name="base_link">
            <visual>
                <!-- 圆柱体 半径0.1 高度0.12 -->
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>

            <!-- 碰撞属性 -->
            <collision>
                <!-- 沿着自己几何中心的偏移和旋转量 -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- 几何形状 -->
                <geometry>
                    <!-- 圆柱体 半径0.1 高度0.12 -->
                    <cylinder radius="${radius}" length="${length}"/>
                </geometry>
                <!-- 材质颜色的描述 -->
                <material name = "white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </collision>
        </link>

        <joint name="base_joint" type="fixed">
            <parent link="base_footprint"/>
```



注意这里我左侧选择的标签 Collision Enabled & Visual Disabled

接下来我们给机器人添加质量与惯性

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

6.9 为机器人添加质量与惯性

在开始前了解两个概念：惯性矩阵

惯性矩阵（Inertia Matrix）是刚体物理学中的一个非常重要的概念，通常用于描述刚体相对于某一坐标系的转动惯量。它告诉我们物体在旋转时如何分布质量，并影响其旋转动态。

📌 一、惯性矩阵的定义

惯性矩阵是描述刚体 **转动惯量** 的一个矩阵，通常是一个 **3x3 的对称矩阵**，用于表示物体在三维空间中对任意旋转轴的惯性特性。

一般形式：

对于一个刚体的惯性矩阵 `I`，它可以表示为：

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

其中：

- I_{xx}, I_{yy}, I_{zz} 是沿各坐标轴的 **转动惯量**。
- I_{xy}, I_{xz}, I_{yz} 是 **惯性产品（product of inertia）**，表示物体在不同轴之间的相互影响。

各元素的含义：

- **主惯性矩**（I_{xx}, I_{yy}, I_{zz}）：是物体绕着各自坐标轴旋转时的惯性。
  - I_{xx} 表示物体绕 x 轴旋转的惯性，I_{yy} 是绕 y 轴的惯性，I_{zz} 是绕 z 轴的惯性。
- **惯性积（cross inertia products）**（I_{xy}, I_{xz}, I_{yz}）：描述物体质量分布在不同坐标轴之间的耦合影响。例如，I_{xy} 是描述物体绕 x 轴和 y 轴同时旋转时的惯性。

📐 二、物理意义

1. 转动惯量（Rotational Inertia）：

转动惯量表示物体在旋转时对旋转加速度的阻力程度。它与质量分布密切相关：

- 转动惯量越大，意味着物体需要更多的力矩来改变其旋转速度。
- 转动惯量和物体的质量分布密切相关，离旋转轴越远的质量对转动惯量的贡献越大。

2. 惯性矩阵的几何意义：

惯性矩阵描述的是物体的质量是如何分布在不同的旋转轴上。通过惯性矩阵，你可以了解物体在各个方向上的“抗旋转”能力，以及它在不同旋转轴上的耦合行为。

- **对称矩阵**：惯性矩阵是对称的，因为转动惯量是标量（不依赖于旋转方向），并且惯性积是互相对称的。
- **物体质量分布**：比如，对于一个长条形物体，绕长度方向的转动惯量可能较大，而绕横向的转动惯量较小。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

关于惯性矩阵的计算，不需要了解太多 一般几何体都会有与之对应的计算公式直接用即可：

1. 长方体（宽 w，高 h，深 d，质量 m）

对于长方体，惯性矩阵的公式假设它的质量分布是均匀的，且物体坐标原点位于其几何中心。

$$I = \frac{1}{12} m \begin{bmatrix} h^2 + d^2 & 0 & 0 \\ 0 & w^2 + d^2 & 0 \\ 0 & 0 & w^2 + h^2 \end{bmatrix}$$

3. 球体（半径 r，质量 m）

对于球体，惯性矩假设质量均匀分布，且坐标原点位于球体的几何中心。

$$I = \frac{2}{5} m \begin{bmatrix} r^2 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \end{bmatrix}$$

圆柱体（质量 m, 半径 r, 高度 h）

$$I = \begin{bmatrix} \frac{1}{12} m \left(3r^2 + h^2\right) & 0 & 0 \\ 0 & \frac{1}{12} m \left(3r^2 + h^2\right) & 0 \\ 0 & 0 & \frac{1}{2} m r^2 \end{bmatrix}$$

下面编写一个专门用于惯性和质量的宏定义 在 urdf / jacksonbot 目录下新建 Common_inertia.xacro 文件 键入以下代码：

```
<!-- 常用惯性矩阵计算—球体、圆柱体、长方体 -->

<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name="box_inertia" params="m w h d">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${m/12*(h*h+d*d)}" ixy="0.0" ixz="0.0" iyy="${m/12*(w*w+d*d)}" iyz="0.0"
                izz="${m/12*(w*w+h*h)}"/>
        </inertial>
    </xacro:macro>

    <xacro:macro name="cylinder_inertia" params="m r h">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${m/12*(3*r*r+h*h)}" ixy="0.0" ixz="0.0" iyy="${m/12*(3*r*r+h*h)}" iyz="0.0"
                izz="${m/2*r*r}"/>
        </inertial>
    </xacro:macro>

    <xacro:macro name="sphere_inertia" params="m r">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${2*m/5*r*r}" ixy="0.0" ixz="0.0" iyy="${2*m/5*r*r}" iyz="0.0"
                izz="${2*m/5*r*r}"/>
        </inertial>
    </xacro:macro>

</robot>
```

Inertial 标签描述机器人惯量 mass 子标签描述质量 inertia 子标签描述惯量

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

编写完成后 我们即可在其他 Xacro 文件导入并使用该宏 逐个修改

（这里以 base 为例）

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/common_inertia.xacro"/>
    <xacro:macro name = "base_xacro" params="radius length">
```

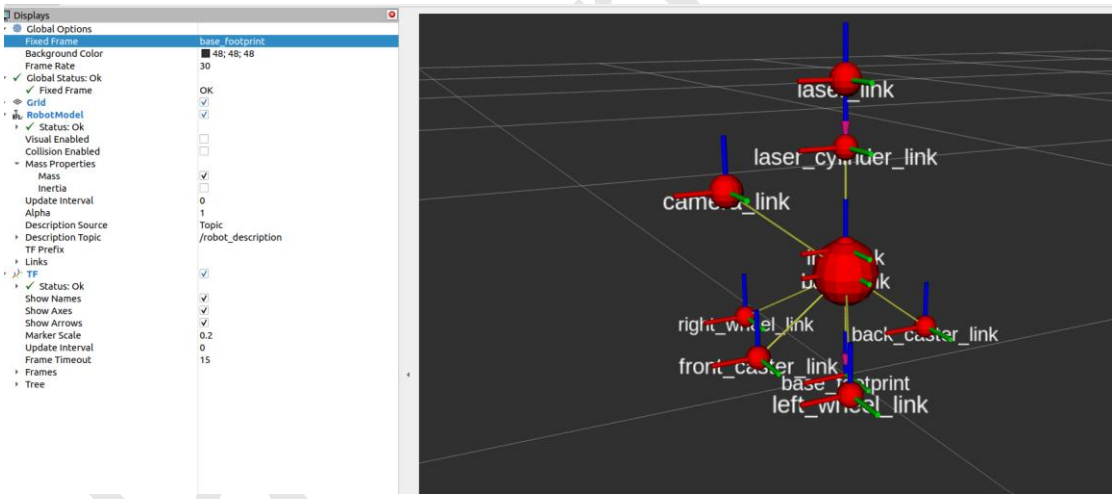
引入中间那个 include

```
    <!-- 惯性属性  注意：参数名必须一致-->
    <xacro:cylinder_inertia m="1.0" r="${radius}" h="${length}"/>
</link>
```

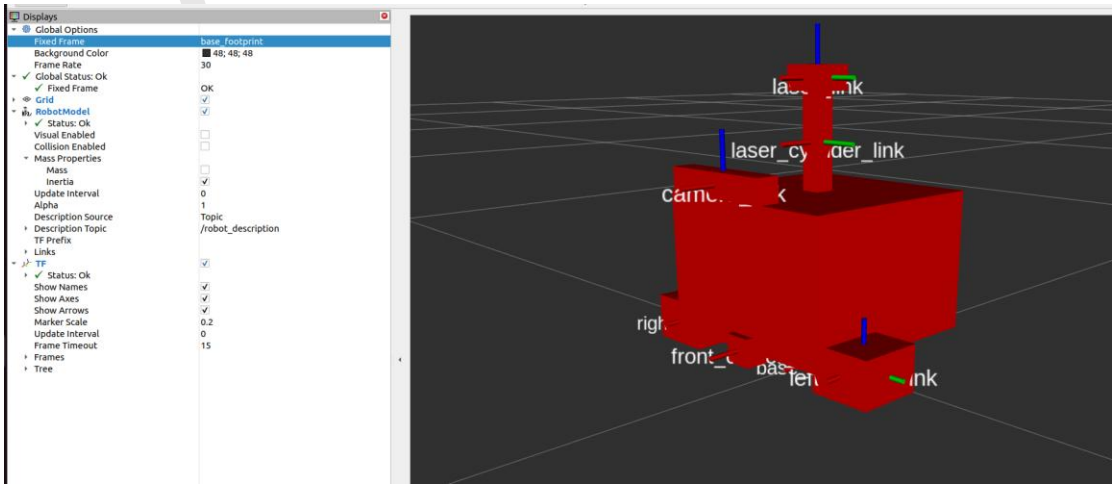
在 link 里传入参数 注意参数名应与 common_inertia 一致

其他函数也按照同样的方法写即可 如果有写死不传参的则直接赋值传过去即可

三件套可以看到下图：质量配置图 在 Mass Properties 里调整那两个框即可



惯性图：



提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

三． 机器人仿真（Gazebo）

6.10 Gazebo 安装与构建世界

安装 Gazebo: sudo apt install gazebo

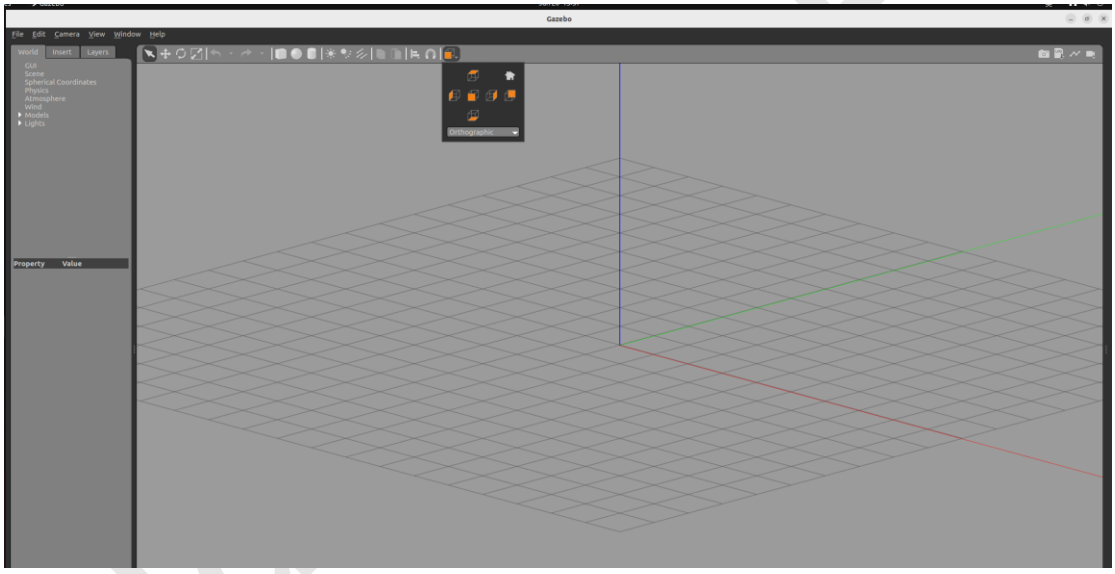
下载模型: mkdir -p ~/.gazebo

cd ~/ .gazebo

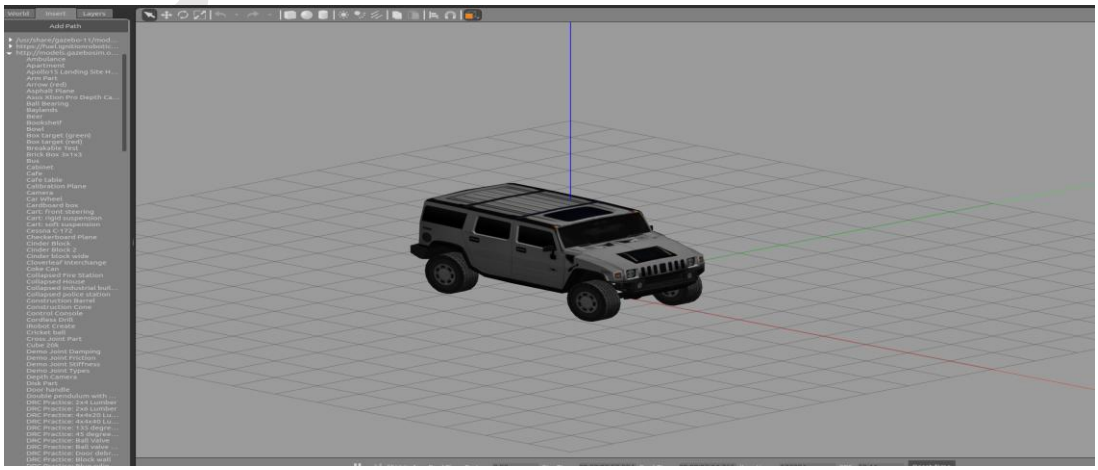
git clone https://gitee.com/ohhuo/gazebo_models.git ~/ .gazebo/models

rm -rf ~/gazebo/models/ .git

终端输入 gazebo 启动后看到一堆灰 调正如图



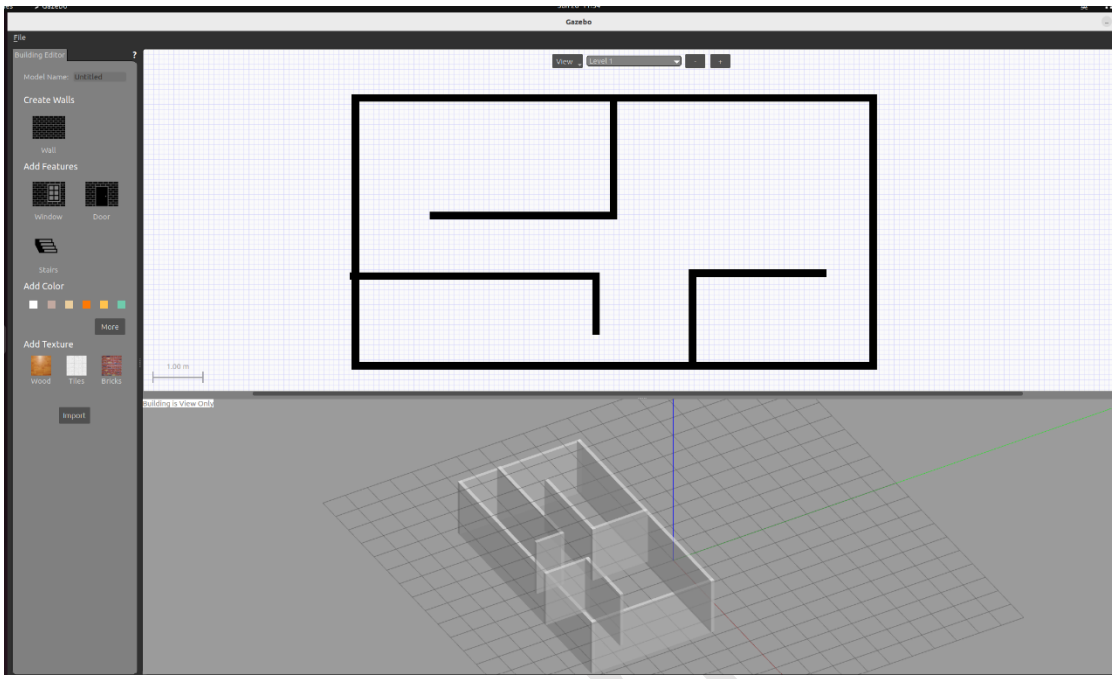
随后打开 insert 可以看到右侧加载了一堆 model (选择 SUV 即可看到下图)



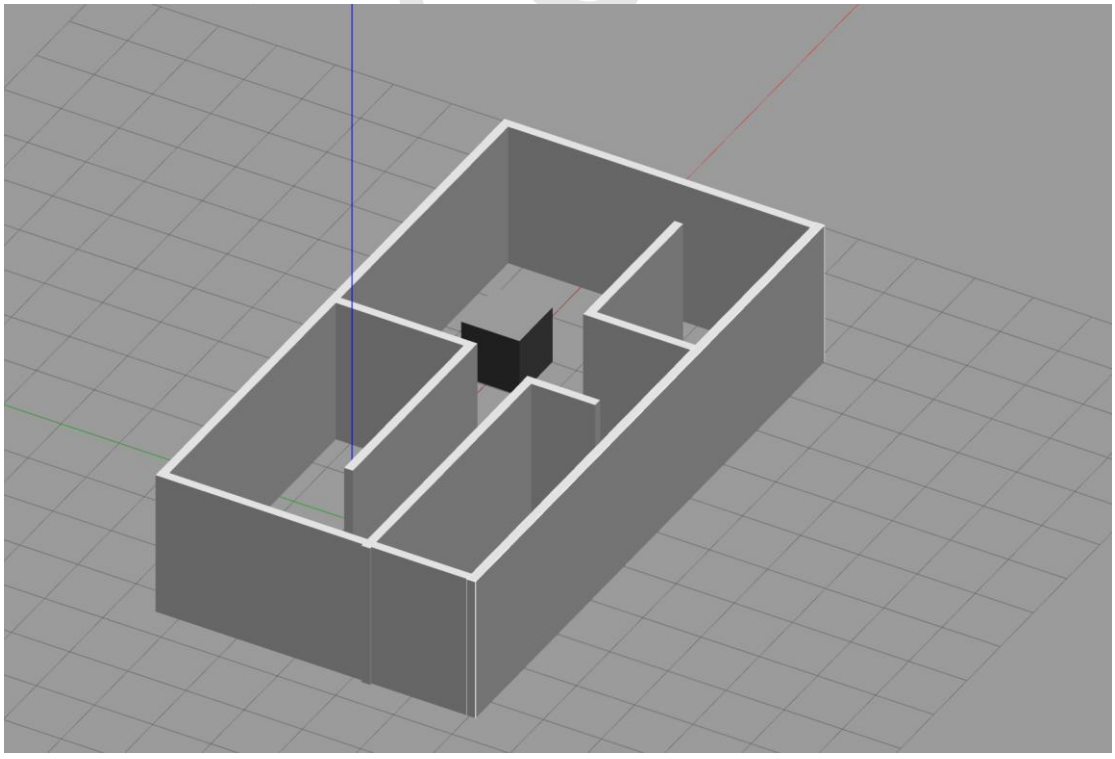
提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

为了后续的导航学习，下面我们来画一个房间，点击 Edit 中的 Building Editor 打开 Create Wall 画线即可



画完后点击 Exit Building Editor 点击 save and exit 存到 chapt6_ws 下 firstbot_description 目录下的新建文件夹 world 后命名为 room 存即可



在屋子中放入正方体后点击 Save world as 放到 wold 同级目录下 上方命名为

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

custom_room.world 放到 world 点击 save 即可。

之前的是保存各模块，现在的是保存各个模块的汇总为一个世界

Gazebo 使用模型文件格式为 sdf 格式 机器人建模使用的是 URDF 所以要转化

6.11 在 Gazebo 中加载机器人模型

安装 gazebo-ros-pkgs 插件帮助实现 URDF -> sdf

sudo apt install ros-$ROS_DISTRO-gazebo-ros-pkgs

在 launch 目录下新建 gazebo_sim.launch.py

把前面展示机器人 URDF 发布的代码拿过来 略作改动

```python
import launch
import launch_ros # 用于启动ROS2节点
from ament_index_python.packages import get_package_share_directory # 用于获取功能包的路径
import os
import launch_ros.parameter_descriptions

def generate_launch_description():
    # 先获取jacksonbot urdf文件
    # 获取功能包share路径
    urdf_file = get_package_share_directory('firstbot_description') # 功能包的路径
    # 拼接URDF文件的完整路径
    urdf_path = os.path.join(urdf_file, 'urdf', 'jacksonbot/jacksonbot.urdf.xacro')
    default_gazebo_world_path = os.path.join(urdf_file, 'world', 'custom_room.world')
    # 声明一个urdf参数 方便修改路径
    action_declare_arg_model_path = launch.actions.DeclareLaunchArgument(
        name = 'model_path',
        default_value = urdf_path,
        description = 'Path to the robot model file'
    )

    # 通过文件路径获取内容： cat命令 注意cat 这里一定要有空格！！
    robot_description = launch.substitutions.Command(['xacro ',launch.substitutions.LaunchConfiguration('model_path'
    # 并转换成参数值对象，以供传入robot_state_publisher节点
    robot_description = launch_ros.parameter_descriptions.ParameterValue(
        robot_description, # 被转换成参数值的内容
        value_type=str    # 转换成的参数值类型
    )

    # 启动robot_state_publisher节点，注意他需要的是文件内容而不是文件路径
    action_robot_state_publisher = launch_ros.actions.Node(
        package='robot_state_publisher', # 功能包名称
        executable='robot_state_publisher', # 可执行文件名称
        name='robot_state_publisher', # 节点名称
        parameters=[{'robot_description': robot_description}],   # 传入的参数
    )

    action_launch_gazebo = launch.actions.IncludeLaunchDescription(
        launch.launch_description_sources.PythonLaunchDescriptionSource(
            [get_package_share_directory('gazebo_ros'), '/launch', '/gazebo.launch.py']
        ),
        launch_arguments=[
            ('world',default_gazebo_world_path), ('verbose','true') # 指定Gazebo世界文件
        ],
    )

    action_spawn_entity = launch_ros.actions.Node(
        package='gazebo_ros', # 功能包名称
        executable='spawn_entity.py', # 可执行文件名字
        arguments=['-topic', '/robot_description', '-entity', 'jacksonbot'], # 通过robot_description参数来生成机器人实体
    )

    return launch.LaunchDescription([
        action_declare_arg_model_path,
        action_robot_state_publisher,
        action_launch_gazebo,
        action_spawn_entity,
    ])
```

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

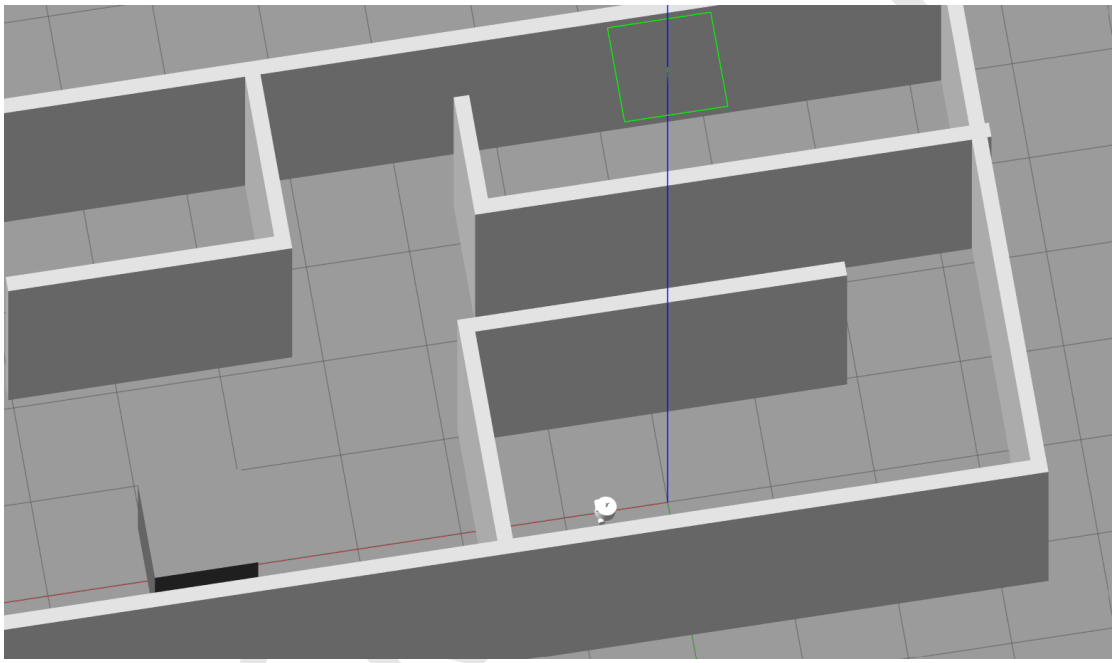
在这里我们不需要看机器人建模和使用 rviz 了，删掉 joint，rviz

需要加上 gazebo 打开以及转换 转换这里使用 gazebo——ros 下的节点

spwan_entity.py 来做转换 我们只需要请求加载即可 指定从 /robot_description 接受机器人模型

该节点启动后先从 topic 中获取 URDF, 再把 URDF 转成 sdf

将 world 写入 CMakeLists.txt 即可



三件套后可以看到机器人的颜色变成白色 是因为 URDF 再转化到 SDF 时一些标签没被处理 我们看下一节 在 URDF 中添加 gazebo 标签。

6.12 使用 Gazebo 标签扩展 URDF

我们可以利用 Gazebo 标签在 URDF 里改动包括但不限于：

颜色 (以 camera 为例) 摩擦力(以 wheel / caster 为例)

```
        <origin xyz="${xyz}" rpy="0.0 0.0 0
    </joint>

    <!-- gazebo 相机颜色设置 -->
    <gazebo reference="camera_link">
        <material>Gazebo/Blue</material>
    </gazebo>
</xacro:macro>
</robot>
```

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

```
<!-- gazebo caster摩擦力设置 -->
<gazebo reference="${caster_name}_link">
    <mu1 value = "0.0"/>
    <mu2 value = "0.0"/>
    <kp value= "1000000000.0"/>
    <kd value= "1.0"/>
</gazebo>
```

```
<!-- gazebo 轮子摩擦力设置 -->
<gazebo reference="${wheel_name}_link">
    <mu1 value = "20.0"/>
    <mu2 value = "20.0"/>
    <kp value= "1000000000.0"/>
    <kd value= "1.0"/>
</gazebo>
```

更多颜色参考以下网址: [simulator_gazebo/Turtorials/ListOfMaterials - ROS Wiki]

摩擦力参数介绍:

Mu1: 切向摩擦系数

Mu2: 法向摩擦系数

Kp: 接触刚度系数

kd: 阻尼系数

Gazebo 中控制物体摩擦行为的四个核心参数分别是:

参数名	含义	说明
`mu1`	切向摩擦系数	与滑动方向一致的摩擦系数（主摩擦方向）
`mu2`	法向摩擦系数	与滑动方向垂直的摩擦系数（副摩擦方向）
`kp`	接触刚度系数	用于模拟接触弹簧的刚度，越大表示越“硬”的接触
`kd`	阻尼系数	模拟接触过程中的能量耗散（例如摩擦产生的阻尼）

1 caster 万向轮设置（摩擦极低）：

xml 复制 编辑

```xml
<!-- gazebo caster摩擦力设置 -->
<gazebo reference="${caster_name}_link">
  <mu1 value="0.0"/>
  <mu2 value="0.0"/>
  <kp value="1000000000.0"/>
  <kd value="1.0"/>
</gazebo>
```

解释：

- `mu1 = mu2 = 0.0` 表示这个轮子几乎没有摩擦，适合模拟支撑轮，不提供驱动力。
- `kp = 1e9`：非常高的刚度，避免穿透。
- `kd = 1.0`：适当阻尼，防止抖动。

2 主动轮设置（摩擦较大）：

xml 复制 编辑

```xml
<!-- gazebo 轮子摩擦力设置 -->
<gazebo reference="${wheel_name}_link">
  <mu1 value="20.0"/>
  <mu2 value="20.0"/>
  <kp value="1000000000.0"/>
  <kd value="1.0"/>
</gazebo>
```

解释：

- `mu1 = mu2 = 20.0`：非常大的摩擦系数，确保轮子在地面上有足够抓地力，能推动机器人。
- 其余同上：高刚度和适当阻尼确保稳定接触。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

6.13 使用二轮差速插件控制机器人

总目标：让机器人能在仿真 Gazebo 平台上实现由键盘控制的运动

需要节点插件：diff_drive

图 6-31　/diff_drive 节点订阅发布话题

总流程如下：现在看不懂等学完这节再回来看

---

1. 用户操作与键盘控制

- **发布节点**: `teleop_twist_keyboard`（用户通过键盘控制）
- **发布话题**: `/cmd_vel`
- **接收者**: `diff_drive`（订阅 `/cmd_vel`）
- **机器人变化**: 用户通过键盘输入控制机器人运动，机器人接收到的命令通常包括线速度和角速度。

2. `diff_drive` 节点接收并执行命令

- **发布节点**: `diff_drive`（机器人驱动控制节点）
- **发布话题**:
  - `/odom`（里程计信息，描述机器人的位置和姿态）
  - `/tf`（坐标变换信息，描述机器人各坐标系之间的位置关系）
- **接收者**:
  - `/odom` 和 `/tf` 会被其他节点（如 `robot_state_publisher`，`rviz`，`gazebo` 等）订阅。
- **机器人变化**: `diff_drive` 根据接收到的 `/cmd_vel` 命令，控制机器人的运动（前进、后退、转向）。此时，机器人在物理世界或仿真环境中发生运动。

3. `robot_state_publisher` 节点处理与发布

- **发布节点**: `robot_state_publisher`（机器人状态发布节点）
- **发布话题**: `/tf`
- **接收者**: `rviz`，`gazebo`（这两个节点订阅 `/tf` 话题）
- **机器人变化**: `robot_state_publisher` 根据 `diff_drive` 发布的 `/odom` 和机器人配置文件（`robot_description`）中的模型，计算并发布机器人各个关节之间的坐标变换。此时，机器人的关节和坐标系发生变化，仿真和可视化界面可以更新机器人的状态。

4. 仿真环境（Gazebo）

- **发布节点**: `gazebo`（仿真环境）
- **接收者**: `diff_drive`（订阅 `/cmd_vel` 和 `/odom`）
- **机器人变化**: Gazebo 中的虚拟机器人根据 `diff_drive` 发布的运动命令在仿真环境中进行移动，仿真过程中也会更新机器人位置。

5. 可视化（RViz）

- **发布节点**: `rviz`（可视化工具）
- **接收者**: `rviz` 会订阅 `/tf` 和 `/odom` 话题
- **机器人变化**: `rviz` 通过订阅 `/tf` 和 `/odom` 更新机器人在可视化界面上的位置、姿态以及运动过程。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

在 urdf/jacksonbot 目录下创建 plugins 文件夹并在目录下新建 gazebo_control_plugin.xacro 键入以下代码：

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="gazebo_control_plugin">
    <gazebo>
      <plugin name="diff_drive" filename="libgazebo_ros_diff_drive.so">
        <ros>
          <namespace>/</namespace> <!-- ROS命名空间 -->
          <remapping>cmd_vel:=/cmd_vel</remapping>
          <remapping>odom:=/odom</remapping>
        </ros>
        <!--发布速率 单位HZ-->
        <update_rate>30.0</update_rate>

        <!--wheels-->
        <left_joint>left_wheel_joint</left_joint> <!--左轮子关节名称-->
        <right_joint>right_wheel_joint</right_joint> <!--右轮子关节名称-->

        <!--kinematics-->
        <wheel_separation>0.2</wheel_separation>  <!--轮子间距-->
        <wheel_diameter>0.064</wheel_diameter> <!--轮子直径-->

        <!--limits-->
        <max_wheel_torque>20</max_wheel_torque> <!--最大轮子扭矩-->
        <max_wheel_acceleration>1.0</max_wheel_acceleration> <!--最大轮子加速度-->

        <!--output-->
        <publish_odom>true</publish_odom> <!--是否发布里程计信息-->
        <publish_odom_tf>true</publish_odom_tf> <!--是否发布里程计TF信息 base_footprint到里程计-->
        <publish_wheel_tf>true</publish_wheel_tf> <!--是否发布轮子TF信息 看到轮子转动-->

        <odometry_frame>odom</odometry_frame> <!--里程计坐标系-->
        <robot_base_frame>base_footprint</robot_base_frame> <!--机器人基座坐标系-->
      </plugin>
    </gazebo>
  </xacro:macro>
</robot>
```

表 6-1 diff_drive 标签的含义

标签	含义
<update_rate>	发布信息的更新频率（以 Hz 为单位）
<left_joint>	左轮关节的名称
<right_joint>	右轮关节的名称
<wheel_separation>	轮子之间的距离
<wheel_diameter>	轮子的直径
<max_wheel_torque>	可应用于轮子的最大扭矩
<max_wheel_acceleration>	轮子的最大加速度
<publish_odom>	是否发布里程计信息
<publish_odom_tf>	是否将里程计信息以 TF 变换的形式发布
<publish_wheel_tf>	是否将轮子的变换以 TF 形式发布
<odometry_frame>	里程计计算所使用的坐标系名称
<robot_base_frame>	机器人基座坐标系的名称

新建宏文件了所以要在 jacksonbot.urdf.xacro 文件中引用调用宏

```xml
<!--插件-->
<xacro:include filename="$(find firstbot_description)/urdf/jacksonbot/plugins/gazebo_control_plugin.xacro"/>
```

```xml
<!-- gazebo 插件 -->
<xacro:gazebo_control_plugin/>
```

此时我们已经完成了基本配置 通过三件套可以打开 gazebo

再新建终端输入 ros2 run teleop_twist_keyboard teleop_twist_keyboard

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

（用键盘控制机器人运动 这里 moving around 的字母相对位置代表方向）

一定要在终端打开并鼠标点击终端输入字母控制才有用 效果如下视频

testing_robot.mp4

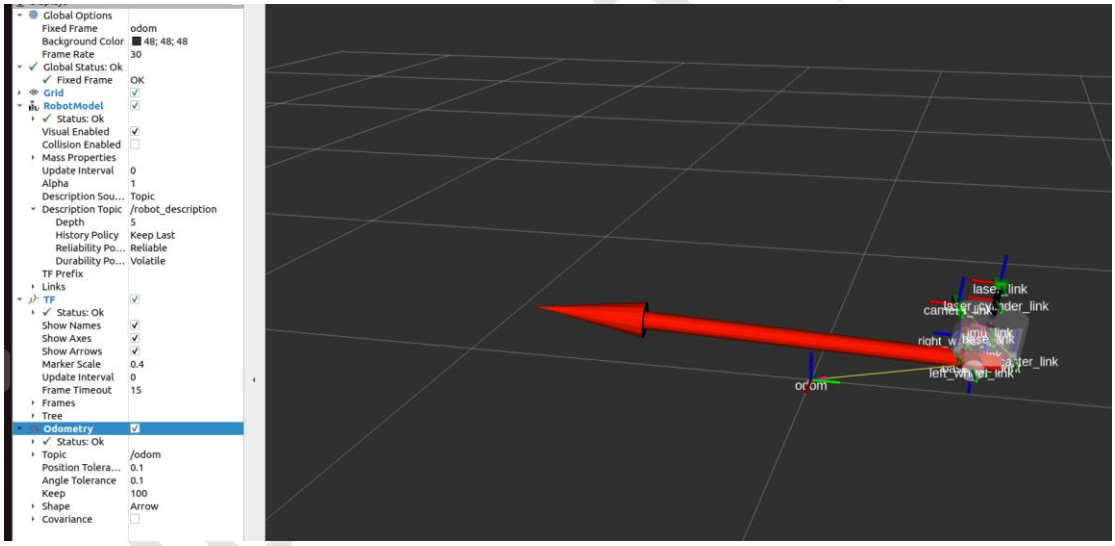
保持 gazebo 运行的状态下，下面介绍里程计

里程计：记录机器人行走位置的传感器数据 可以通过 rviz 显示其位置。

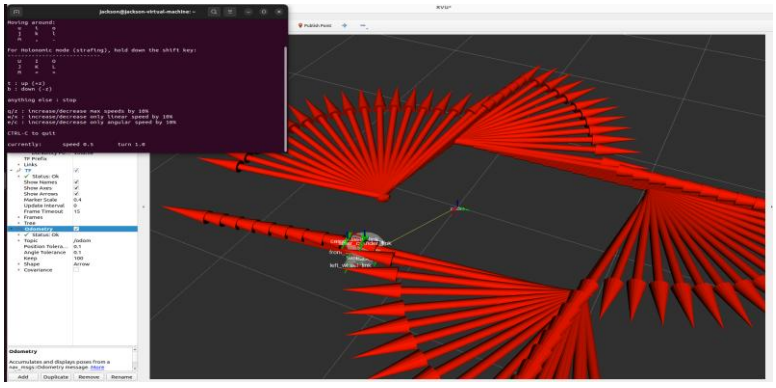
打开 rviz 修改 Fixed Frame 为 odom （基准坐标系为 odom）

点击 add 引入机器人模型 选择里面的订阅话题为 robot_description 导入模型

导入 tf 调整里面的参数 点击 add 后点击 By topic 中的 Odometry 并取消 Convariance 红色的箭头就是里程计的位置和方向



此时键盘控制运动可看到下面效果图 这里会保存 100 条里程计数据 可以看到 Rviz 中里程计保存了机器人走过的路径



提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

6.14 激光雷达传感器仿真

只有里程计传感器还不够 要完成复杂交互必须还需要各种传感器 我们下面来添加激光雷达传感器，它可以提供机器人周围环境的距离信息

在 plugins 下新建 gazebo_sensor_plugin.xacro 键入以下代码：

*注意那个映射是~/out 不是-/out

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name="gazebo_sensor_plugin">
        <gazebo reference="laser_link"> <!-- 写雷达所在的关节名称 -->
            <sensor name ="laserscan" type="ray"> <!-- 传感器名称和类型（射线激光） -->
                <plugin name="laserscan" filename="libgazebo_ros_ray_sensor.so">
                    <ros>
                        <namespace>/</namespace> <!-- ROS命名空间 -->
                        <remapping>~/out:=scan</remapping>  <!-- ROS话题重映射 -->
                    </ros>
                    <output_type>sensor_msgs/LaserScan</output_type> <!-- 输出类型 -->
                    <frame_name>laser_link</frame_name> <!-- 传感器链接名称 -->
                </plugin>
                <always_on>true</always_on> <!-- 是否始终开启传感器 -->
                <visualize>true</visualize> <!-- 是否可视化激光光线 -->
                <update_rate>5.0</update_rate> <!--发布速率 单位HZ-->
                <pose>0 0 0 0 0 0</pose> <!-- 传感器位置和姿态 000000没有任何偏移旋转 -->
                <!--激光传感器配置-->
                <ray>
                    <scan>
                        <horizontal>
                            <samples>360</samples> <!-- 扫描样本数 （转一圈360个点） -->
                            <resolution>1.000000</resolution> <!-- 分辨率 -->
                            <min_angle>0.000000</min_angle> <!-- 最小角度 -->
                            <max_angle>6.280000</max_angle> <!-- 最大角度 （3.14*2） -->
                        </horizontal>
                    </scan>
                    <!-- 设置扫描范围 -->
                    <range>
                        <min>0.120000</min> <!-- 最小范围 （米） -->
                        <max>8.0</max> <!-- 最大范围 （米） -->
                        <resolution>0.015000</resolution> <!-- 分辨率 -->
                    </range>
                    <!-- 设置噪声 -->
                    <noise>
                        <type>gaussian</type> <!-- 噪声类型 -->
                        <mean>0.0</mean> <!-- 均值 -->
                        <stddev>0.01</stddev> <!-- 标准差 -->
                    </noise>
                </ray>
            </sensor>
        </gazebo>
    </xacro:macro>
</robot>
```

别忘了到 jacksonbot.urdf.xacro 中添加插件引入并引用

发布话题 / 关系 / 订阅节点 / 如何查看

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

3. 激光雷达与 Gazebo 的通信

在 Gazebo 中，激光雷达插件（如 `gazebo_ros_laser`）模拟激光传感器，生成激光束并进行环境扫描。当激光雷达与 Gazebo 环境中的物体发生碰撞时，它会生成距离数据，并通过 ROS 的话题（如 `/scan`）发布出去。

- **发布话题**: `/scan`
- **发布节点**: `gazebo_ros_laser` 插件
- **话题类型**: `sensor_msgs/LaserScan`

4. 激光雷达与 ROS 的通信

激光雷达生成的数据（距离和强度信息）通过 ROS 发布到 `/scan` 话题，其他 ROS 节点（如 `rviz` 或机器人控制节点）可以订阅这个话题来获取环境感知数据。

订阅者:

- `rviz`：通过订阅 `/scan` 话题，`rviz` 可以实时显示激光雷达扫描数据（例如，在 2D 或 3D 环境中显示激光束或障碍物）。
- **其他ROS节点**：例如，路径规划、避障模块可以订阅这个话题来进行环境建模和决策。

5. 激光雷达与 RViz 的通信

- **发布话题**: `/scan`
- **订阅话题**: `rviz`
- **消息类型**: `sensor_msgs/LaserScan`
- **显示方式**: 激光雷达数据可以在 `RViz` 中以激光扫描的形式显示，通常是通过 `LaserScan` 显示类型来展示扫描的结果。

如何在 RViz 中可视化

在 RViz 中显示激光雷达数据时，需要添加一个 `LaserScan` 显示类型并选择 `/scan` 话题进行订阅。

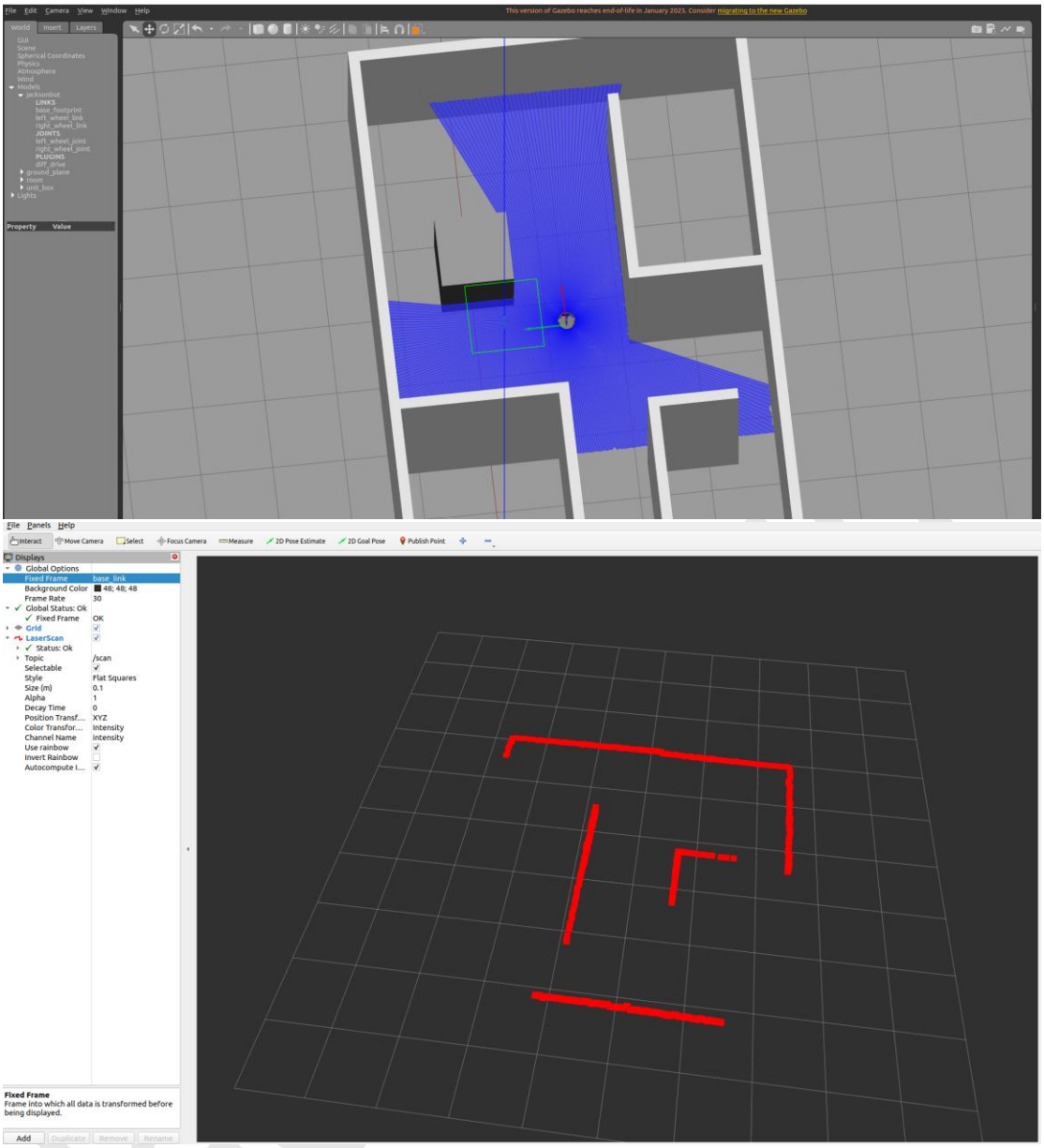
操作步骤:

1. 在 `RViz` 中，点击左侧的 "Add" 按钮。
2. 选择 "LaserScan" 类型。
3. 在 "Topic" 字段中输入 `/scan`（确保它与发布的激光雷达话题一致）。
4. 调整显示参数（例如颜色、大小等），以便更好地可视化激光数据。

最终 Rviz & Gazebo 结果显示如下两图：

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可



（雷达点云信息）

6.15 惯性测量传感器仿真

我们最后再来看惯性测量传感器作用及相关介绍 结合代码运行结果可以更直观理解

在 gazebo_sensor_plugin 中继续添加代码

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可

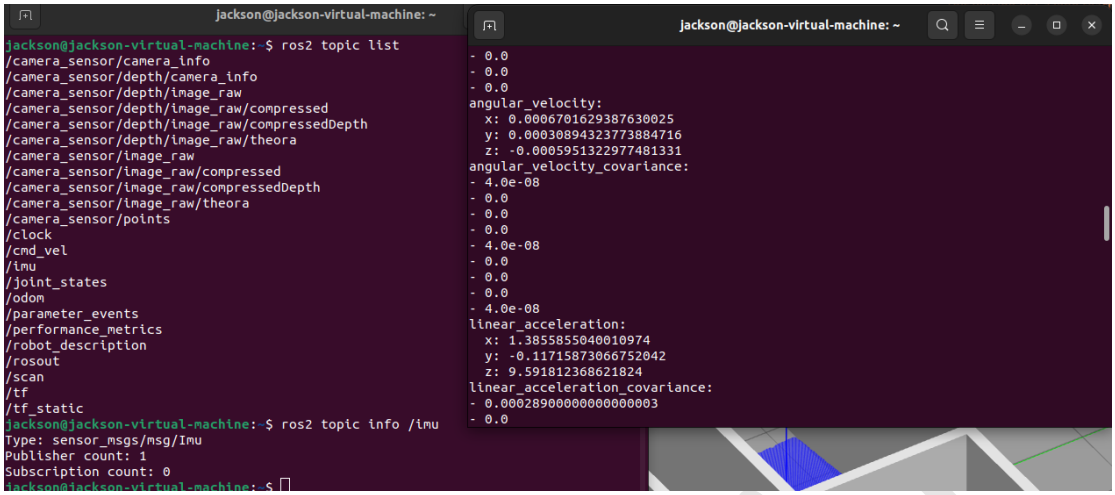
```
        <!--IMU传感器插件-->
        <gazebo reference="imu_link">
            <sensor name="imu_sensor" type="imu">
            <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
                <ros>
                    <namespace>/</namespace>
                    <remapping>~/out:=imu</remapping>
                </ros>
                <initial_orientation_as_reference>false</initial_orientation_as_reference> <!--是否使用初始方向作为参考
                </plugin>
                <always_on>true</always_on>
                <update_rate>100</update_rate>
                <visualize>true</visualize>
                <!-- 六轴噪声设置 -->
                <imu>
                <angular_velocity>
                    <x>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>2e-4</stddev>
                        <bias_mean>0.0000075</bias_mean>
                        <bias_stddev>0.0000008</bias_stddev>
                    </noise>
                    </x>
                    <y>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>2e-4</stddev>
                        <bias_mean>0.0000075</bias_mean>
                        <bias_stddev>0.0000008</bias_stddev>
                    </noise>
                    </y>
                    <z>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>2e-4</stddev>
                        <bias_mean>0.0000075</bias_mean>
                        <bias_stddev>0.0000008</bias_stddev>
                    </noise>
                    </z>
                </angular_velocity>
                <linear_acceleration>
                    <x>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>1.7e-2</stddev>
                        <bias_mean>0.1</bias_mean>
                        <bias_stddev>0.001</bias_stddev>
                    </noise>
                    </x>
                    <y>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>1.7e-2</stddev>
                        <bias_mean>0.1</bias_mean>
                        <bias_stddev>0.001</bias_stddev>
                    </noise>
                    </y>
                    <z>
                    <noise type="gaussian">
                        <mean>0.0</mean>
                        <stddev>1.7e-2</stddev>
                        <bias_mean>0.1</bias_mean>
                        <bias_stddev>0.001</bias_stddev>
                    </noise>
                    </z>
                </linear_acceleration>
                </imu>
            </sensor>
            </gazebo>
```

保存后构建启动仿真 输入以下命令：

Ros2 topic echo /imu 即可看到 IMU 数据

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!

注意：本章节有部分代码图片中的注释存在形状体描述错误 但是不影响整体运行 发现问题后改正即可



Recall：

Ros2 topic list 查看所有的发布的话题

Ros2 topic info /imu 查看 imu 的发布类型及发布节点 / 订阅节点

右侧终端我们可以看到订阅了 imu 话题后会得到

1) 协方差矩阵
2) 线速度 / 线加速度
3) 角速度 / 角加速度

IMU 固定在机器人上 当机器人发生位姿变化时 IMU 数据就会变化

Eg: 当机器人发生打滑的时候，IMU 数据不会变化但是里程计却会发生变化 我们可以通过这样来判断机器人是否发生打滑。

提示：一定要注意各种配置文件代码中的下划线有几个 / 是{}还是[]!