第 6 章 机器人系统仿真

对于 ROS 新手而言，可能会有疑问:学习机器人操作系统，实体机器人是必须的吗？答案是否定的，机器人一般价格不菲，为了降低机器人学习、调试成本，在 ROS 中提供了系统的机器人仿真实现，通过仿真，可以实现大部分需求，本章主要就是围绕“仿真”展开的，比如，本章会介绍:

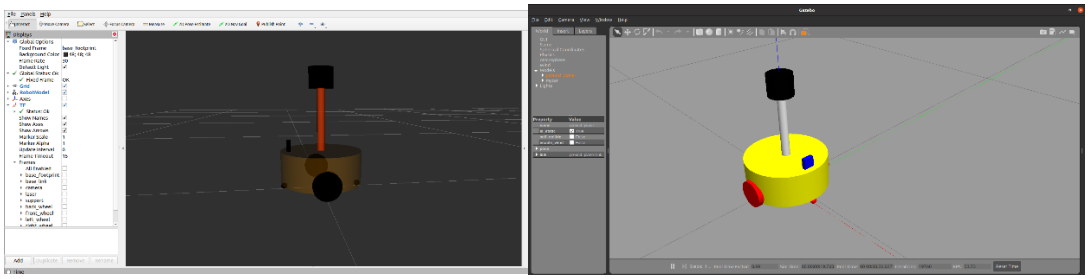
- 如何创建并显示机器人模型；
- 如何搭建仿真环境；
- 如何实现机器人模型与仿真环境的交互。

本章预期的学习目标如下:

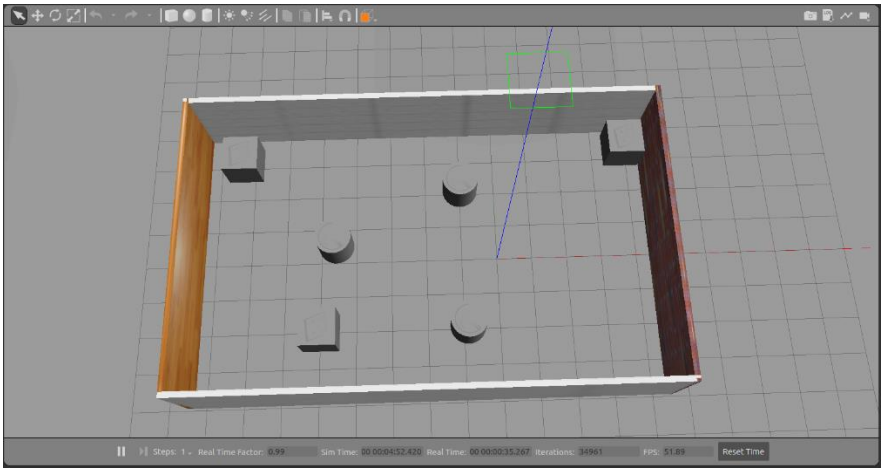
- 能够独立使用 URDF 创建机器人模型，并在 Rviz 和 Gazebo 中分别显示；
- 能够使用 Gazebo 搭建仿真环境；
- 能够使用机器人模型中的传感器(雷达、摄像头、编码器...)获取仿真环境数据。

案例演示:

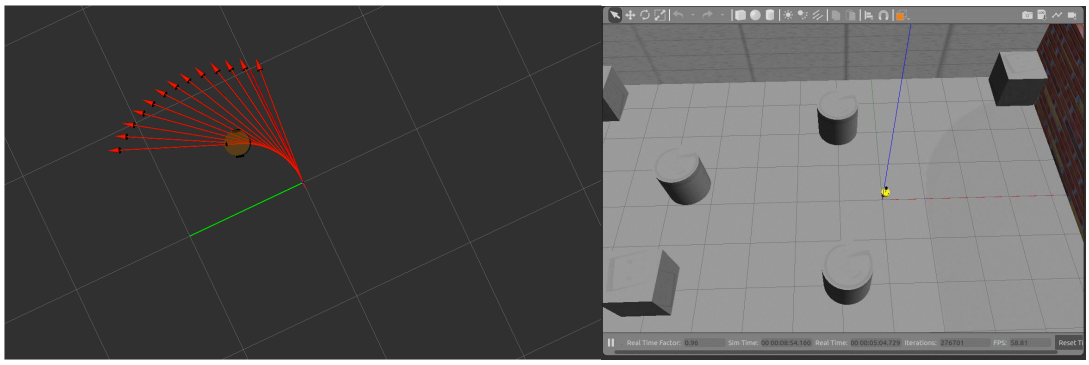
1.创建并显示机器人模型



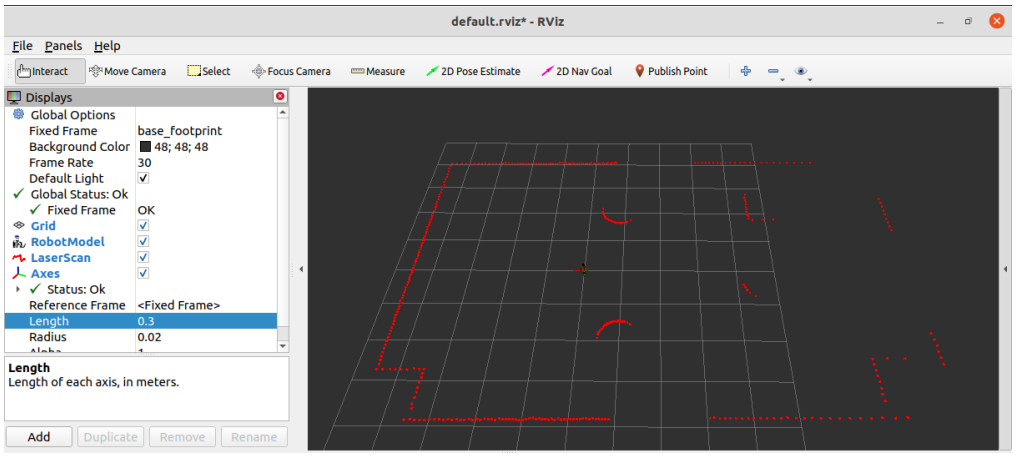
2.仿真环境搭建



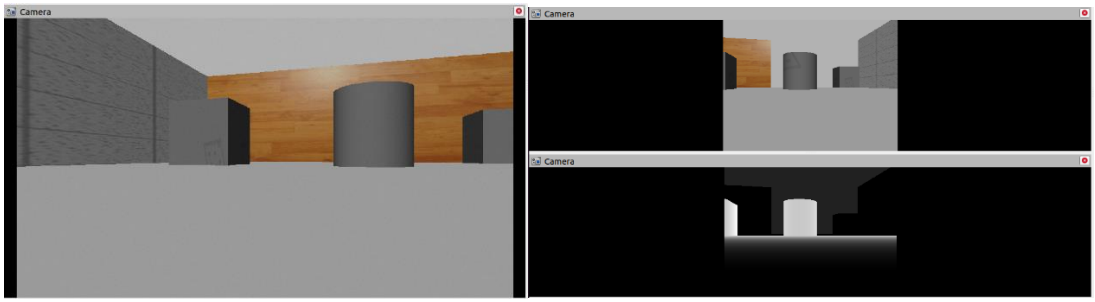
3.控制机器人运动



4.雷达仿真



5.摄像头仿真



6.1 概述

机器人操作系统学习、开发与测试过程中，会遇到诸多问题，比如:

场景 1:机器人一般价格不菲，学习 ROS 要购买一台机器人吗？

场景 2:机器人与之交互的外界环境具有多样性，如何实现复杂的环境设计？

场景 3:测试时，直接将未经验证的程序部署到实体机器人运行，安全吗？

...

在诸如此类的场景中，ROS 中的**仿真**就显得尤为重要了。

1.概念

机器人系统仿真：是通过计算机对实体机器人系统进行模拟的技术，在 ROS 中，仿真实现涉及的内容主要有三:对机器人建模(URDF)、创建仿真环境(Gazebo)以及感知环境(Rviz)等系统性实现。

2.作用

2.1 仿真优势:

仿真在机器人系统研发过程中占有举足轻重的地位，在研发与测试中较之于实体机器人实现，仿真有如下几点的显著优势:

1.**低成本:**当前机器人成本居高不下，动辄几十万，仿真可以大大降低成本，减小风险

2.**高效:**搭建的环境更为多样且灵活，可以提高测试效率以及测试覆盖率

3.**高安全性:**仿真环境下，无需考虑耗损问题

2.2 仿真缺陷:

机器人在仿真环境与实际环境下的表现差异较大，换言之，仿真并不能完全做到模拟真实的物理世界，存在一些"失真"的情况，原因:

1.仿真器所使用的物理引擎目前还不能够完全精确模拟真实世界的物理情况

2.仿真器构建的是关节驱动器（电机&齿轮箱）、传感器与信号通信的绝对理想情况，目前不支持模拟实际硬件缺陷或者一些临界状态等情形

3.相关组件

3.1URDF

URDF 是 Unified Robot Description Format 的首字母缩写，直译为**统一(标准化)机器人描述格式**，可以以一种 XML 的方式描述机器人的部分结构，比如底盘、摄像头、激光雷达、机械臂以及不同关节的自由度.....,该文件可以被 C++ 内置的解释器转换成可视化的机器人模型，是 ROS 中实现机器人仿真的重要组件

3.2rviz

RViz 是 ROS Visualization Tool 的首字母缩写，直译为 **ROS 的三维可视化工具**。它的主要目的是以三维方式显示 ROS 消息，可以将 数据进行可视化表达。例如:可以显示机器人模型，可以无需编程就能表达激光测距仪（LRF）传感器中的传感 器到障碍物的距离，RealSense、Kinect 或 Xtion 等三维距离传感器的点云数据（PCD， Point Cloud Data），从相机获取的图像值等

以“ros- [ROS_DISTRO] -desktop-full”命令安装 ROS 时，RViz 会默认被安装。

运行使用命令 `rviz` 或 `rosrun rviz rviz`
如果 rviz 没有安装，请调用如下命令自行安装:

```
sudo apt install ros-[ROS_DISTRO]-rviz
Copy
```

3.3gazebo

Gazebo 是一款 3D 动态模拟器，用于显示机器人模型并创建仿真环境,能够在复杂的室内和室外环境中准确有效地模拟机器人。与游戏引擎提供高保真度的视觉模拟类似，Gazebo 提供高保真度的物理模拟，其提供一整套传感器模型，以及对用户和程序非常友好的交互方式。

以“ros- [ROS_DISTRO] -desktop-full”命令安装 ROS 时，gzebo 会默认被安装。

运行使用命令 `gazebo` 或 `rosrun gazebo_ros gazebo`
注意 1:*在 Ubuntu20.04 与 ROS Noetic 环境下，gazebo 启动异常以及解决*

- **问题 1:**VMware: vmw_ioctl_command error Invalid argument(无效的参数)

  解决:

  ```
  echo "export SVGA_VGPU10=0" >> ~/.bashrc
  source .bashrc
  ```

- **问题 2:**[Err] [REST.cc:205] Error in REST request

  解决:`sudo gedit ~/.ignition/fuel/config.yaml`
  然后将 `url : https://api.ignitionfuel.org` 使用 # 注释
  再添加 `url: https://api.ignitionrobotics.org`

- **问题 3:**启动时抛出异常:`[gazebo-2] process has died [pid xxx, exit code 255, cmd.....`
  解决:`killall gzserver` 和 `killall gzclient`
  注意 2:*如果 gazebo 没有安装，请自行安装:*

  1.添加源:

  ```
  sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` main"
  >
   /etc/apt/sources.list.d/gazebo-stable.list'
  Copy
  wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
  Copy
  ```

  2.安装:

  ```
  sudo apt update
  Copy
  sudo apt install gazebo11
  sudo apt install libgazebo11-dev
  Copy
  ```

另请参考:

- https://wiki.ros.org/urdf
- http://wiki.ros.org/rviz
- http://gazebosim.org/tutorials?tut=ros_overview

课程说明:

机器人的系统仿真是一种集成实现，主要包含三部分:

- URDF 用于创建机器人模型
- Gzebo 用于搭建仿真环境
- Rviz 图形化的显示机器人各种传感器感知到的环境信息

三者应用中，只是创建 URDF 意义不大，一般需要结合 Gazebo 或 Rviz 使用，在 Gazebo 或 Rviz 中可以将 URDF 文件解析为图形化的机器人模型，一般的使用组合为:

- 如果非仿真环境，那么使用 URDF 结合 Rviz 直接显示感知的真实环境信息
- 如果是仿真环境，那么需要使用 URDF 结合 Gazebo 搭建仿真环境，并结合 Rviz 显示感知的虚拟环境信息

后续课程安排:

- 先介绍 URDF 与 Rviz 集成使用，在 Rviz 中只是显示机器人模型，主要用于学习 URDF 语法
- 再介绍 URDF 与 Gazebo 集成，主要学习 URDF 仿真相关语法以及仿真环境搭建
- 最后集成 URDF 与 Gazebo 与 Rviz，实现综合应用

素材链接:

- https://github.com/zx595306686/sim_demo.git

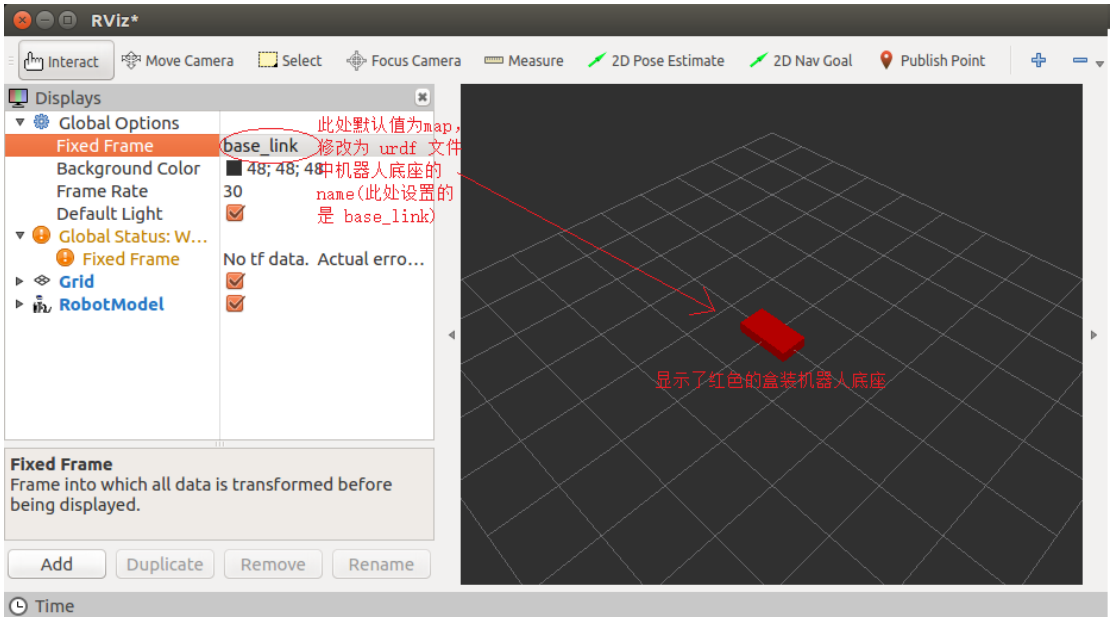
6.2 URDF 集成 Rviz 基本流程

前面介绍过，URDF 不能单独使用，需要结合 Rviz 或 Gazebo，URDF 只是一个文件，需要在 Rviz 或 Gazebo 中渲染成图形化的机器人模型，当前，首先演示 URDF 与 Rviz 的集成使用，因为 URDF 与 Rviz 的集成较之于 URDF 与 Gazebo 的集成更为简单，后期，基于 Rviz 的集成实现，我们再进一步介绍 URDF 语法。

需求描述:

在 Rviz 中显示一个盒状机器人

结果演示:



实现流程:

1. 准备:新建功能包，导入依赖
2. 核心:编写 urdf 文件
3. 核心:在 launch 文件集成 URDF 与 Rviz
4. 在 Rviz 中显示机器人模型

1.创建功能包，导入依赖

创建一个新的功能包，名称自定义，导入依赖包:`urdf` 与 `xacro`
在当前功能包下，再新建几个目录:

`urdf`: 存储 urdf 文件的目录
`meshes`:机器人模型渲染文件(暂不使用)
`config`: 配置文件
`launch`: 存储 launch 启动文件

2.编写 URDF 文件

新建一个子级文件夹:`urdf`(可选)，文件夹中添加一个`.urdf`文件,复制如下内容:

```
<robot name="mycar">
    <link name="base_link">
        <visual>
            <geometry>
```

```
            <box size="0.5 0.2 0.1" />
        </geometry>
    </visual>
</link>
</robot>
Copy
```

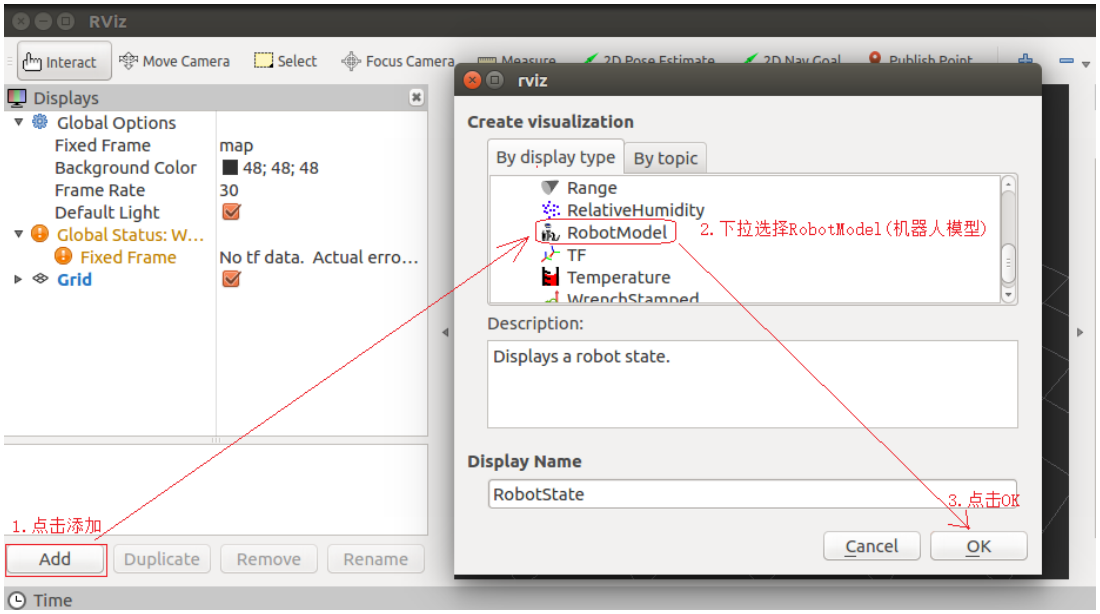
3.在 launch 文件中集成 URDF 与 Rviz

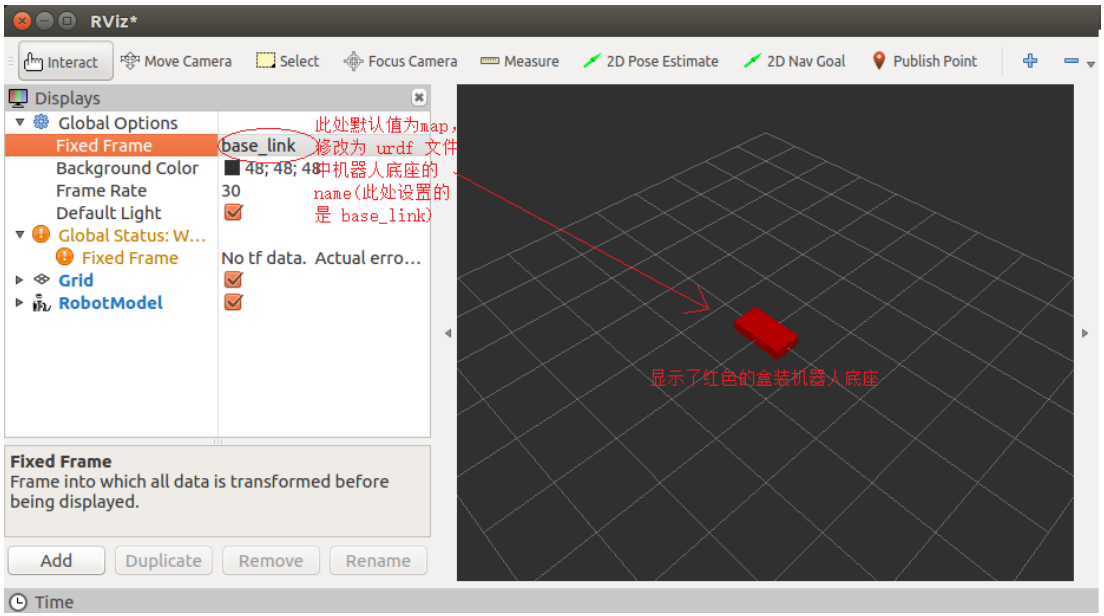
在`launch`目录下，新建一个 launch 文件，该 launch 文件需要启动 Rviz，并导入 urdf 文件，Rviz 启动后可以自动载入解析`urdf`文件，并显示机器人模型，核心问题:如何导入 urdf 文件? 在 ROS 中，可以将 urdf 文件的路径设置到参数服务器，使用的参数名是:`robot_description`,示例代码如下:

```
<launch>

    <!-- 设置参数 -->
    <param name="robot_description" textfile="$(find 包
名)/urdf/urdf/urdf01_HelloWorld.urdf" />

    <!-- 启动 rviz -->
    <node pkg="rviz" type="rviz" name="rviz" />

</launch>
Copy
```

4.在 Rviz 中显示机器人模型

rviz 启动后，会发现并没有盒装的机器人模型，这是因为默认情况下没有添加机器人显示组件，需要手动添加，添加方式如下:



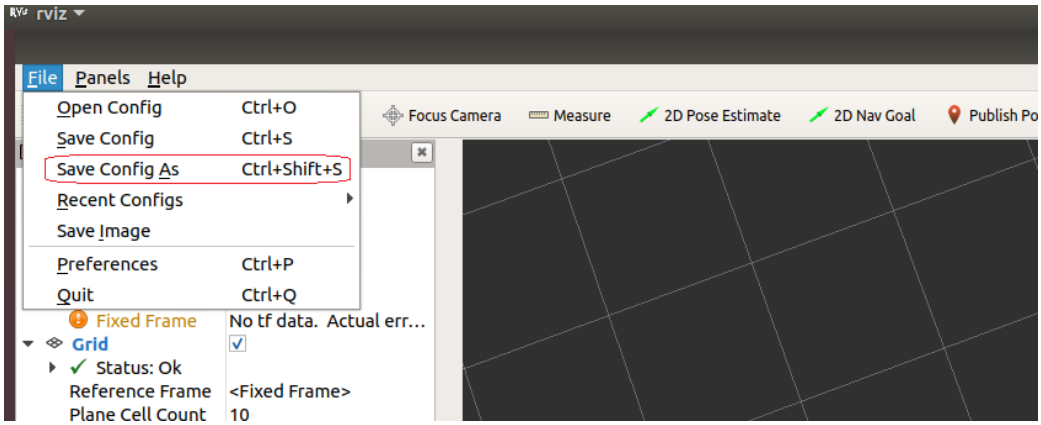


设置完毕后，可以正常显示了

5.优化 rviz 启动

重复启动 `launch` 文件时，Rviz 之前的组件配置信息不会自动保存，需要重复执行步骤 4 的操作，为了方便使用，可以使用如下方式优化:

首先，将当前配置保存进 `config` 目录



然后，`launch` 文件中 Rviz 的启动配置添加参数:`args`,值设置为`-d 配置文件路径`

```
<launch>
    <param name="robot_description" textfile="$(find 包名)/urdf/urdf/urdf01_HelloWorld.urdf" />
    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find 报名)/config/rviz/show_mycar.rviz" />
</launch>
```

Copy

再启动时，就可以包含之前的组件配置了，使用更方便快捷。

6.3 URDF 语法详解

URDF 文件是一个标准的 XML 文件，在 ROS 中预定义了一系列的标签用于描述机器人模型，机器人模型可能较为复杂，但是 ROS 的 URDF 中机器人的组成却是较为简单，可以主要简化为两部分:连杆(link 标签) 与 关节(joint 标签)，接下来我们就通过案例了解一下 URDF 中的不同标签:

- robot 根标签，类似于 launch 文件中的 launch 标签
- link 连杆标签
- joint 关节标签
- gazebo 集成 gazebo 需要使用的标签

关于 gazebo 标签，后期在使用 gazebo 仿真时，才需要使用到，用于配置仿真环境所需参数，比如: 机器人材料属性、gazebo 插件等，但是该标签不是机器人模型必须的，只有在仿真时才需设置

另请参考:

- https://wiki.ros.org/urdf/XML

6.3.1 URDF 语法详解 01_robot

robot

urdf 中为了保证 xml 语法的完整性，使用了 `robot` 标签作为根标签，所有的 link 和 joint 以及其他标签都必须包含在 robot 标签内,在该标签内可以通过 name 属性设置机器人模型的名称

1.属性

name: 指定机器人模型的名称

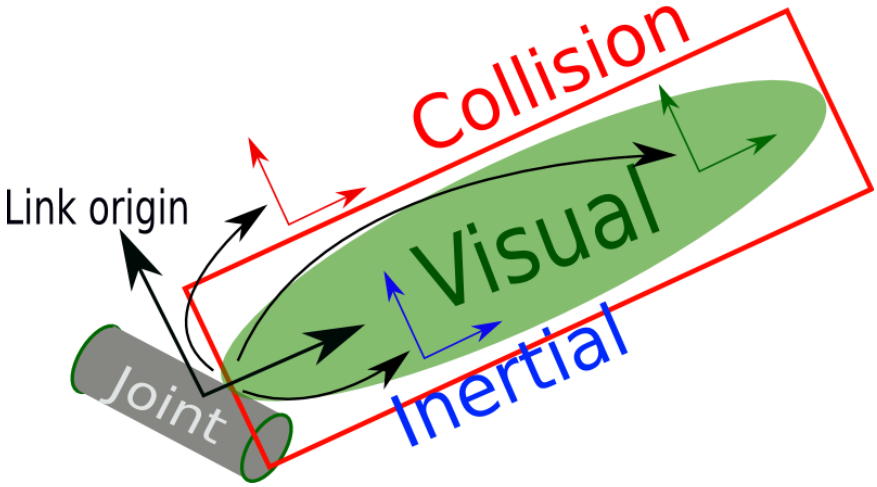
2.子标签

其他标签都是子级标签

6.3.2 URDF 语法详解 02_link

link

urdf 中的 link 标签用于描述机器人某个部件(也即刚体部分)的外观和物理属性，比如: 机器人底座、轮子、激光雷达、摄像头...每一个部件都对应一个 link, 在 link 标签内，可以设计该部件的形状、尺寸、颜色、惯性矩阵、碰撞参数等一系列属性



1.属性

- name ---> 为连杆命名

2.子标签

- visual ---> 描述外观(对应的数据是可视的)
  - geometry 设置连杆的形状
    - 标签 1: box(盒状)
    - 属性:size=长(x) 宽(y) 高(z)
    - 标签 2: cylinder(圆柱)
    - 属性:radius=半径 length=高度
    - 标签 3: sphere(球体)
    - 属性:radius=半径
    - 标签 4: mesh(为连杆添加皮肤)
    - 属性: filename=资源路径(格式:**package://<packagename>/<path>/文件**)
  - origin 设置偏移量与倾斜弧度
    - 属性 1: xyz=x 偏移 y 便宜 z 偏移
    - 属性 2: rpy=x 翻滚 y 俯仰 z 偏航 (单位是弧度)
  - metrial 设置材料属性(颜色)
    - 属性: name
    - 标签: color
    - 属性: rgba=红绿蓝权重值与透明度 (每个权重值以及透明度取值[0,1])
- collision ---> 连杆的碰撞属性
- Inertial ---> 连杆的惯性矩阵

在此，只演示 `visual` 使用。

3.案例

需求:分别生成长方体、圆柱与球体的机器人部件

```
<link name="base_link">
    <visual>
```

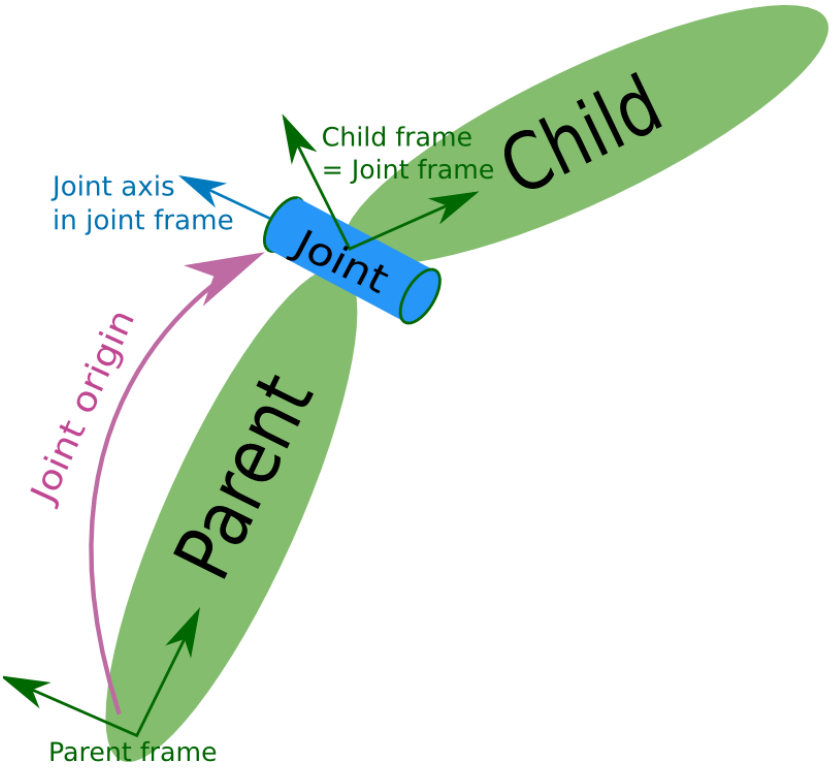
```
            <!-- 形状 -->
            <geometry>
                <!-- 长方体的长宽高 -->
                <!-- <box size="0.5 0.3 0.1" /> -->
                <!-- 圆柱，半径和长度 -->
                <!-- <cylinder radius="0.5" length="0.1" /> -->
                <!-- 球体，半径-->
                <!-- <sphere radius="0.3" /> -->

            </geometry>
            <!-- xyz 坐标 rpy 翻滚俯仰与偏航角度(3.14=180 度 1.57=90 度) -->
            <origin xyz="0 0 0" rpy="0 0 0" />
            <!-- 颜色: r=red g=green b=blue a=alpha -->
            <material name="black">
                <color rgba="0.7 0.5 0 0.5" />
            </material>
        </visual>
    </link>
Copy
```

6.3.3 URDF 语法详解 03_joint

joint

urdf 中的 joint 标签用于描述机器人关节的运动学和动力学属性，还可以指定关节运动的安全极限，机器人的两个部件(分别称之为 parent link 与 child link)以"关节"的形式相连接，不同的关节有不同的运动形式: 旋转、滑动、固定、旋转速度、旋转角度限制....,比如:安装在底座上的轮子可以 360 度旋转，而摄像头则可能是完全固定在底座上。

joint 标签对应的数据在模型中是不可见的



1.属性

- name ---> 为关节命名
- type ---> 关节运动形式
  - continuous: 旋转关节，可以绕单轴无限旋转
  - revolute: 旋转关节，类似于 continues,但是有旋转角度限制
  - prismatic: 滑动关节，沿某一轴线移动的关节，有位置极限
  - planer: 平面关节，允许在平面正交方向上平移或旋转
  - floating: 浮动关节，允许进行平移、旋转运动
  - fixed: 固定关节，不允许运动的特殊关节

2.子标签

- parent(必需的)

  parent link 的名字是一个强制的属性：

  - link:父级连杆的名字，是这个 link 在机器人结构树中的名字。
- child(必需的)

child link 的名字是一个强制的属性：

- link:子级连杆的名字，是这个 link 在机器人结构树中的名字。
  - origin
- 属性: xyz=各轴线上的偏移量 rpy=各轴线上的偏移弧度。
  - axis
- 属性: xyz 用于设置围绕哪个关节轴运动。

3.案例

需求:创建机器人模型，底盘为长方体，在长方体的前面添加一摄像头，摄像头可以沿着 Z 轴 360 度旋转。

URDF 文件示例如下:

```
<!--
    需求: 创建机器人模型，底盘为长方体，
         在长方体的前面添加一摄像头，
         摄像头可以沿着 Z 轴 360 度旋转

 -->
<robot name="mycar">
    <!-- 底盘 -->
    <link name="base_link">
        <visual>
            <geometry>
                <box size="0.5 0.2 0.1" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="blue">
                <color rgba="0 0 1.0 0.5" />
            </material>
        </visual>
    </link>

    <!-- 摄像头 -->
    <link name="camera">
        <visual>
            <geometry>
                <box size="0.02 0.05 0.05" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="red">
```

```
                <color rgba="1 0 0 0.5" />
            </material>
        </visual>
    </link>

    <!-- 关节 -->
    <joint name="camera2baselink" type="continuous">
        <parent link="base_link"/>
        <child link="camera" />
        <!-- 需要计算两个 link 的物理中心之间的偏移量 -->
        <origin xyz="0.2 0 0.075" rpy="0 0 0" />
        <axis xyz="0 0 1" />
    </joint>

</robot>
```
Copy

launch 文件示例如下:

```
<launch>

    <param name="robot_description" textfile="$(find
urdf_rviz_demo)/urdf/urdf/urdf03_joint.urdf" />
    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
urdf_rviz_demo)/config/helloworld.rviz" />

    <!-- 添加关节状态发布节点 -->
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" />
    <!-- 添加机器人状态发布节点 -->
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" />
    <!-- 可选:用于控制关节运动的节点 -->
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui" />

</launch>
```
Copy

PS:

1.状态发布节点在此是必须的:

```
    <!-- 添加关节状态发布节点 -->
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" />
    <!-- 添加机器人状态发布节点 -->
```

```
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" />
Copy
```

2.关节运动控制节点(可选)，会生成关节控制的 UI，用于测试关节运动是否正常。

```
    <!-- 可选:用于控制关节运动的节点 -->
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui" />
Copy
```

4.base_footprint 优化 urdf

前面实现的机器人模型是半沉到地下的，因为默认情况下: 底盘的中心点位于地图原点上，所以会导致这种情况产生，可以使用的优化策略，将初始 link 设置为一个尺寸极小的 link(比如半径为 0.001m 的球体，或边长为 0.001m 的立方体)，然后再在初始 link 上添加底盘等刚体，这样实现，虽然仍然存在初始 link 半沉的现象，但是基本可以忽略了。这个初始 link 一般称之为 base_footprint

```
<!--

    使用 base_footprint 优化

-->
<robot name="mycar">
    <!-- 设置一个原点(机器人中心点的投影) -->
    <link name="base_footprint">
        <visual>
            <geometry>
                <sphere radius="0.001" />
            </geometry>
        </visual>
    </link>

    <!-- 添加底盘 -->
    <link name="base_link">
        <visual>
            <geometry>
                <box size="0.5 0.2 0.1" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="blue">
                <color rgba="0 0 1.0 0.5" />
```

```
            </material>
        </visual>
    </link>

    <!-- 底盘与原点连接的关节 -->
    <joint name="base_link2base_footprint" type="fixed">
        <parent link="base_footprint" />
        <child link="base_link" />
        <origin xyz="0 0 0.05" />
    </joint>

    <!-- 添加摄像头 -->
    <link name="camera">
        <visual>
            <geometry>
                <box size="0.02 0.05 0.05" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="red">
                <color rgba="1 0 0 0.5" />
            </material>
        </visual>
    </link>
    <!-- 关节 -->
    <joint name="camera2baselink" type="continuous">
        <parent link="base_link"/>
        <child link="camera" />
        <origin xyz="0.2 0 0.075" rpy="0 0 0" />
        <axis xyz="0 0 1" />
    </joint>

</robot>
```
Copy

launch 文件内容不变。

5.遇到问题以及解决

问题 1:

命令行输出如下错误提示

```
UnicodeEncodeError: 'ascii' codec can't encode characters in position
463-464: ordinal not in range(128)
```

```
[joint_state_publisher-3] process has died [pid 4443, exit code 1, cmd
/opt/ros/melodic/lib/joint_state_publisher/joint_state_publisher
__name:=joint_state_publisher __log:=/home/rosmelodic/.ros/log/b38967c0-
0acb-11eb-aee3-0800278ee10c/joint_state_publisher-3.log].
log file: /home/rosmelodic/.ros/log/b38967c0-0acb-11eb-aee3-
0800278ee10c/joint_state_publisher-3*.log
Copy
```

rviz 中提示坐标变换异常，导致机器人部件显示结构异常

原因:编码问题导致的

解决:去除 URDF 中的中文注释

问题 2:[ERROR] [1584370263.037038]: Could not find the GUI, install the 'joint_state_publisher_gui' package

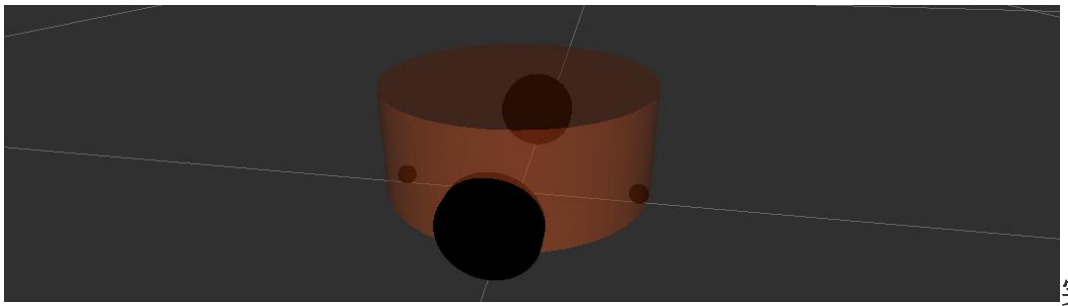
解决:`sudo apt install ros-noetic-joint-state-publisher-gui`

6.3.4 URDF 练习

需求描述:

创建一个四轮圆柱状机器人模型，机器人参数如下,底盘为圆柱状，半径 10cm，高 8cm，四轮由两个驱动轮和两个万向支撑轮组成，两个驱动轮半径为 3.25cm,轮胎宽度 1.5cm，两个万向轮为球状，半径 0.75cm，底盘离地间距为 1.5cm(与万向轮直径一致)

结果演示:



实现流程:

创建机器人模型可以分步骤实现

1. 新建 urdf 文件，并与 launch 文件集成
2. 搭建底盘
3. 在底盘上添加两个驱动轮
4. 在底盘上添加两个万向轮

1.新建 urdf 以及 launch 文件

urdf 文件:基本实现

```
<robot name="mycar">
    <!-- 设置 base_footprint  -->
    <link name="base_footprint">
        <visual>
            <geometry>
                <sphere radius="0.001" />
            </geometry>
        </visual>
    </link>

    <!-- 添加底盘 -->


    <!-- 添加驱动轮 -->


    <!-- 添加万向轮(支撑轮) -->

</robot>
```

Copy

launch 文件:

```
<launch>
    <!-- 将 urdf 文件内容设置进参数服务器 -->
    <param name="robot_description" textfile="$(find
demo01_urdf_helloworld)/urdf/urdf/test.urdf" />

    <!-- 启动 rivz -->
    <node pkg="rviz" type="rviz" name="rviz_test" args="-d $(find
demo01_urdf_helloworld)/config/helloworld.rviz" />

    <!-- 启动机器人状态和关节状态发布节点 -->
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" />
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" />

    <!-- 启动图形化的控制关节运动节点 -->
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui" />
```

```
</launch>
```
Copy

2.底盘搭建

```
<!--
        参数
            形状:圆柱
            半径:10     cm
            高度:8      cm
            离地:1.5    cm

    -->
    <link name="base_link">
        <visual>
            <geometry>
                <cylinder radius="0.1" length="0.08" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="yellow">
                <color rgba="0.8 0.3 0.1 0.5" />
            </material>
        </visual>
    </link>

    <joint name="base_link2base_footprint" type="fixed">
        <parent link="base_footprint" />
        <child link="base_link"/>
        <origin xyz="0 0 0.055" />
    </joint>
```
Copy

3.添加驱动轮

```
<!-- 添加驱动轮 -->
    <!--
        驱动轮是侧翻的圆柱
        参数
            半径: 3.25 cm
            宽度: 1.5  cm
            颜色: 黑色
        关节设置:
```

```
            x = 0
            y = 底盘的半径 + 轮胎宽度 / 2
            z = 离地间距 + 底盘长度 / 2 - 轮胎半径 = 1.5 + 4 - 3.25 =
2.25(cm)
            axis = 0 1 0
    -->
    <link name="left_wheel">
        <visual>
            <geometry>
                <cylinder radius="0.0325" length="0.015" />
            </geometry>
            <origin xyz="0 0 0" rpy="1.5705 0 0" />
            <material name="black">
                <color rgba="0.0 0.0 0.0 1.0" />
            </material>
        </visual>

    </link>

    <joint name="left_wheel2base_link" type="continuous">
        <parent link="base_link" />
        <child link="left_wheel" />
        <origin xyz="0 0.1 -0.0225" />
        <axis xyz="0 1 0" />
    </joint>


    <link name="right_wheel">
        <visual>
            <geometry>
                <cylinder radius="0.0325" length="0.015" />
            </geometry>
            <origin xyz="0 0 0" rpy="1.5705 0 0" />
            <material name="black">
                <color rgba="0.0 0.0 0.0 1.0" />
            </material>
        </visual>

    </link>

    <joint name="right_wheel2base_link" type="continuous">
        <parent link="base_link" />
        <child link="right_wheel" />
        <origin xyz="0 -0.1 -0.0225" />
```

```
        <axis xyz="0 1 0" />
    </joint>
Copy
```

4.添加万向轮

```
<!-- 添加万向轮(支撑轮) -->
    <!--
        参数
            形状: 球体
            半径: 0.75 cm
            颜色: 黑色

        关节设置:
            x = 自定义(底盘半径 - 万向轮半径) = 0.1 - 0.0075 = 0.0925(cm)
            y = 0
            z = 底盘长度 / 2 + 离地间距 / 2 = 0.08 / 2 + 0.015 / 2 = 0.0475
            axis= 1 1 1

    -->
    <link name="front_wheel">
        <visual>
            <geometry>
                <sphere radius="0.0075" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="black">
                <color rgba="0.0 0.0 0.0 1.0" />
            </material>
        </visual>
    </link>

    <joint name="front_wheel2base_link" type="continuous">
        <parent link="base_link" />
        <child link="front_wheel" />
        <origin xyz="0.0925 0 -0.0475" />
        <axis xyz="1 1 1" />
    </joint>

    <link name="back_wheel">
        <visual>
            <geometry>
                <sphere radius="0.0075" />
            </geometry>
```

```
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="black">
                <color rgba="0.0 0.0 0.0 1.0" />
            </material>
        </visual>
    </link>

    <joint name="back_wheel2base_link" type="continuous">
        <parent link="base_link" />
        <child link="back_wheel" />
        <origin xyz="-0.0925 0 -0.0475" />
        <axis xyz="1 1 1" />
    </joint>
Copy
```

思考:

- 上述代码实现存在什么问题吗？比如复用性！

6.3.5 URDF 工具

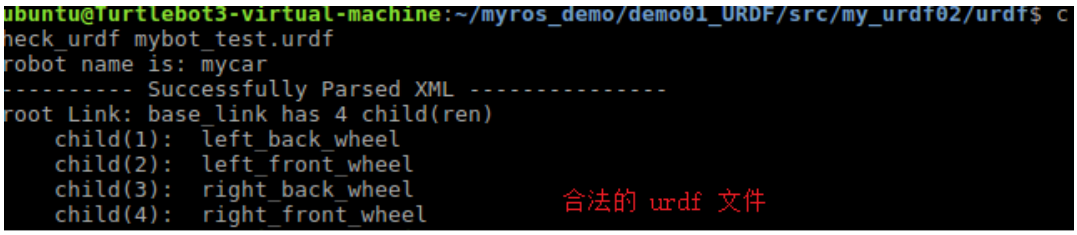
在 ROS 中，提供了一些工具来方便 URDF 文件的编写，比如:

- `check_urdf` 命令可以检查复杂的 urdf 文件是否存在语法问题
- `urdf_to_graphiz` 命令可以查看 urdf 模型结构，显示不同 link 的层级关系

当然，要使用工具之前，首先需要安装，安装命令:`sudo apt install liburdfdom-tools`

1.check_urdf 语法检查

进入 urdf 文件所属目录，调用:`check_urdf urdf 文件`，如果不抛出异常，说明文件合法,否则非法

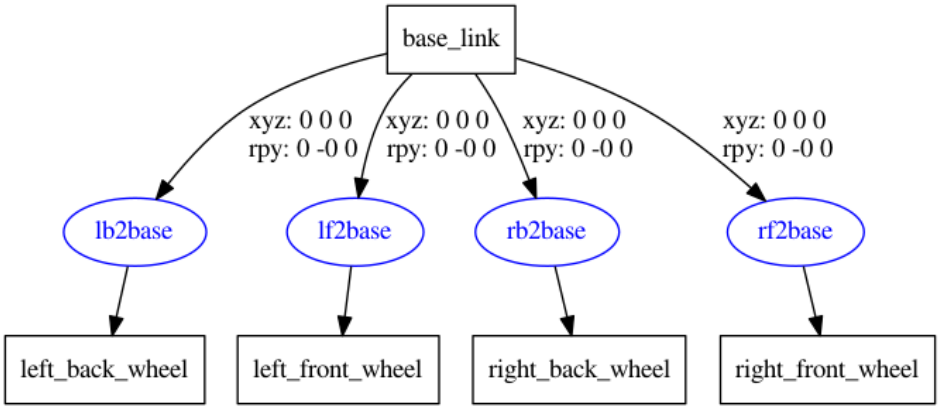


```
ubuntu@turtlebot3-virtual-machine:~/myros_demo/demo01_URDF/src/my_urdf02/urdf$ c
heck_urdf mybot_test.urdf
Error:   joint [lf2base] has no type, check to see if it's a reference.
         at line 441 in /build/urdfdom-UJ3kd6/urdfdom-0.4.1/urdf_parser/src/join
t.cpp
Error:   joint xml is not initialized correctly
         at line 207 in /build/urdfdom-UJ3kd6/urdfdom-0.4.1/urdf_parser/src/mode
l.cpp
ERROR: Model Parsing the xml failed
```

非法的 URDF 文件

2.urdf_to_graphiz 结构查看

进入 urdf 文件所属目录，调用:`urdf_to_graphiz urdf 文件`，当前目录下会生成 pdf 文件



6.4 URDF 优化_xacro

前面 URDF 文件构建机器人模型的过程中，存在若干问题。

问题 1:在设计关节的位置时，需要按照一定的公式计算，公式是固定的，但是在 URDF 中依赖于人工计算，存在不便，容易计算失误，且当某些参数发生改变时，还需要重新计算。

问题 2:URDF 中的部分内容是高度重复的，驱动轮与支撑轮的设计实现，不同轮子只是部分参数不同，形状、颜色、翻转量都是一致的，在实际应用中，构建复杂的机器人模型时，更是易于出现高度重复的设计，按照一般的编程涉及到重复代码应该考虑封装。

.....

如果在编程语言中，可以通过变量结合函数直接解决上述问题，在 ROS 中，已经给出了类似编程的优化方案，称之为:**Xacro**

概念

Xacro 是 XML Macros 的缩写，Xacro 是一种 XML 宏语言，是可编程的 XML。

原理

Xacro 可以声明变量，可以通过数学运算求解，使用流程控制控制执行顺序，还可以通过类似函数的实现，封装固定的逻辑，将逻辑中需要的可变的数据以参数的方式暴露出去，从而提高代码复用率以及程序的安全性。

作用

较之于纯粹的 URDF 实现，可以编写更安全、精简、易读性更强的机器人模型文件，且可以提高编写效率。

另请参考:

- http://wiki.ros.org/xacro

6.4.1 Xacro_快速体验

目的:简单了解 xacro 的基本语法。

需求描述:

使用 xacro 优化上一节案例中驱动轮实现，需要使用变量封装底盘的半径、高度，使用数学公式动态计算底盘的关节点坐标，使用 Xacro 宏封装轮子重复的代码并调用宏创建两个轮子(注意: 在此，演示 Xacro 的基本使用，不必要生成合法的 URDF)。

准备:

创建功能包，导入 urdf 与 xacro。

1.Xacro 文件编写

编写 Xacro 文件，以变量的方式封装属性(常量半径、高度、车轮半径...)，以函数的方式封装重复实现(车轮的添加)。

```
<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">
    <!-- 属性封装 -->
    <xacro:property name="wheel_radius" value="0.0325" />
    <xacro:property name="wheel_length" value="0.0015" />
    <xacro:property name="PI" value="3.1415927" />
    <xacro:property name="base_link_length" value="0.08" />
    <xacro:property name="lidi_space" value="0.015" />

    <!-- 宏 -->
    <xacro:macro name="wheel_func" params="wheel_name flag" >
        <link name="${wheel_name}_wheel">
            <visual>
                <geometry>
                    <cylinder radius="${wheel_radius}"
length="${wheel_length}" />
                </geometry>

                <origin xyz="0 0 0" rpy="${PI / 2} 0 0" />

                <material name="wheel_color">
                    <color rgba="0 0 0 0.3" />
                </material>
            </visual>
        </link>

        <!-- 3-2.joint -->
        <joint name="${wheel_name}2link" type="continuous">
            <parent link="base_link"  />
            <child link="${wheel_name}_wheel" />
            <!--
                x 无偏移
                y 车体半径
                z z= 车体高度 / 2 + 离地间距 - 车轮半径

            -->
            <origin xyz="0 ${0.1 * flag} ${(base_link_length / 2 +
lidi_space - wheel_radius) * -1}" rpy="0 0 0" />
            <axis xyz="0 1 0" />
        </joint>

    </xacro:macro>
    <xacro:wheel_func wheel_name="left" flag="1" />
    <xacro:wheel_func wheel_name="right" flag="-1" />
</robot>
```

Copy

2.Xacro 文件转换成 urdf 文件

命令行进入 xacro 文件 所属目录，执行:`rosrun xacro xacro xxx.xacro > xxx.urdf`，会将 xacro 文件解析为 urdf 文件，内容如下:

```
<?xml version="1.0" ?>
<!--
==========================================================================
========== -->
<!-- |    This document was autogenerated by xacro from test.xacro
| -->
<!-- |    EDITING THIS FILE BY HAND IS NOT RECOMMENDED
| -->
<!--
==========================================================================
========== -->
<robot name="mycar">
  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder length="0.0015" radius="0.0325"/>
      </geometry>
      <origin rpy="1.57079635 0 0" xyz="0 0 0"/>
      <material name="wheel_color">
        <color rgba="0 0 0 0.3"/>
      </material>
    </visual>
  </link>
  <!-- 3-2.joint -->
  <joint name="left2link" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <!--
                x 无偏移
                y 车体半径
                z z= 车体高度 / 2 + 离地间距 - 车轮半径

            -->
    <origin rpy="0 0 0" xyz="0 0.1 -0.0225"/>
    <axis xyz="0 1 0"/>
  </joint>
  <link name="right_wheel">
    <visual>
```

```
      <geometry>
        <cylinder length="0.0015" radius="0.0325"/>
      </geometry>
      <origin rpy="1.57079635 0 0" xyz="0 0 0"/>
      <material name="wheel_color">
        <color rgba="0 0 0 0.3"/>
      </material>
    </visual>
  </link>
  <!-- 3-2.joint -->
  <joint name="right2link" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <!--
                x 无偏移
                y 车体半径
                z z= 车体高度 / 2 + 离地间距 - 车轮半径

            -->
    <origin rpy="0 0 0" xyz="0 -0.1 -0.0225"/>
    <axis xyz="0 1 0"/>
  </joint>
</robot>
```
Copy

注意: 该案例编写生成的是非法的 URDF 文件，目的在于演示 Xacro 的极简使用以及优点。

6.4.2 Xacro_语法详解

xacro 提供了可编程接口，类似于计算机语言，包括变量声明调用、函数声明与调用等语法实现。在使用 xacro 生成 urdf 时，根标签 `robot` 中必须包含命名空间声明:`xmlns:xacro="http://wiki.ros.org/xacro"`

1.属性与算数运算

用于封装 URDF 中的一些字段，比如: PAI 值，小车的尺寸，轮子半径 ....

属性定义

```
<xacro:property name="xxxx" value="yyyy" />
```
Copy

属性调用

```
${属性名称}
Copy
```

算数运算

```
${数学表达式}
Copy
```

2.宏

类似于函数实现，提高代码复用率，优化代码结构，提高安全性

宏定义

```
<xacro:macro name="宏名称" params="参数列表(多参数之间使用空格分隔)">

    .....

    参数调用格式: ${参数名}

</xacro:macro>
Copy
```

宏调用

```
<xacro:宏名称 参数1=xxx 参数2=xxx/>
Copy
```

3.文件包含

机器人由多部件组成，不同部件可能封装为单独的 xacro 文件，最后再将不同的文件集成，组合为完整机器人，可以使用文件包含实现

文件包含

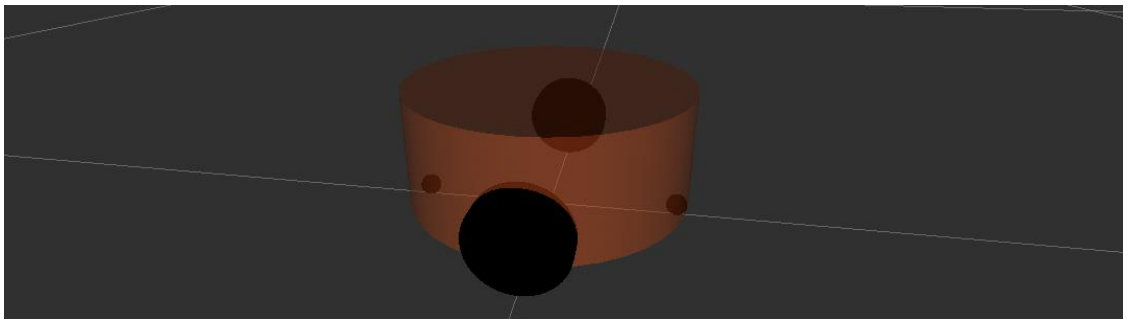
```
<robot name="xxx" xmlns:xacro="http://wiki.ros.org/xacro">
     <xacro:include filename="my_base.xacro" />
     <xacro:include filename="my_camera.xacro" />
     <xacro:include filename="my_laser.xacro" />
     ....
</robot>
```

6.4.3 Xacro_完整使用流程示例

需求描述:

使用 Xacro 优化 URDF 版的小车底盘模型实现

结果演示:



1.编写 Xacro 文件

```
<!--
    使用 xacro 优化 URDF 版的小车底盘实现:

    实现思路:
    1.将一些常量、变量封装为 xacro:property
      比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度 ....
    2.使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动轮与支撑轮

-->
<!-- 根标签，必须声明 xmlns:xacro -->
<robot name="my_base" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <!-- 封装变量、常量 -->
    <xacro:property name="PI" value="3.141"/>
    <!-- 宏:黑色设置 -->
    <material name="black">
        <color rgba="0.0 0.0 0.0 1.0" />
    </material>
    <!-- 底盘属性 -->
    <xacro:property name="base_footprint_radius" value="0.001" /> <!--
base_footprint 半径  -->
    <xacro:property name="base_link_radius" value="0.1" /> <!-- base_link
半径 -->
    <xacro:property name="base_link_length" value="0.08" /> <!--
base_link 长 -->
    <xacro:property name="earth_space" value="0.015" /> <!-- 离地间距 -->

    <!-- 底盘 -->
    <link name="base_footprint">
      <visual>
        <geometry>
          <sphere radius="${base_footprint_radius}" />
```

```
      </geometry>
    </visual>
  </link>

  <link name="base_link">
    <visual>
      <geometry>
        <cylinder radius="${base_link_radius}"
length="${base_link_length}" />
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <material name="yellow">
        <color rgba="0.5 0.3 0.0 0.5" />
      </material>
    </visual>
  </link>

  <joint name="base_link2base_footprint" type="fixed">
    <parent link="base_footprint" />
    <child link="base_link" />
    <origin xyz="0 0 ${earth_space + base_link_length / 2 }" />
  </joint>

  <!-- 驱动轮 -->
  <!-- 驱动轮属性 -->
  <xacro:property name="wheel_radius" value="0.0325" /><!-- 半径 -->
  <xacro:property name="wheel_length" value="0.015" /><!-- 宽度 -->
  <!-- 驱动轮宏实现 -->
  <xacro:macro name="add_wheels" params="name flag">
    <link name="${name}_wheel">
      <visual>
        <geometry>
          <cylinder radius="${wheel_radius}" length="${wheel_length}" />
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />
        <material name="black" />
      </visual>
    </link>

    <joint name="${name}_wheel2base_link" type="continuous">
      <parent link="base_link" />
      <child link="${name}_wheel" />
      <origin xyz="0 ${flag * base_link_radius} ${-(earth_space +
base_link_length / 2 - wheel_radius) }" />
```

```
        <axis xyz="0 1 0" />
      </joint>
    </xacro:macro>
    <xacro:add_wheels name="left" flag="1" />
    <xacro:add_wheels name="right" flag="-1" />
    <!-- 支撑轮 -->
    <!-- 支撑轮属性 -->
    <xacro:property name="support_wheel_radius" value="0.0075" /> <!-- 支撑轮半径 -->

    <!-- 支撑轮宏 -->
    <xacro:macro name="add_support_wheel" params="name flag" >
      <link name="${name}_wheel">
        <visual>
            <geometry>
                <sphere radius="${support_wheel_radius}" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="black" />
        </visual>
      </link>

      <joint name="${name}_wheel2base_link" type="continuous">
          <parent link="base_link" />
          <child link="${name}_wheel" />
          <origin xyz="${flag * (base_link_radius - support_wheel_radius)} 0 ${-(base_link_length / 2 + earth_space / 2)}" />
          <axis xyz="1 1 1" />
      </joint>
    </xacro:macro>

    <xacro:add_support_wheel name="front" flag="1" />
    <xacro:add_support_wheel name="back" flag="-1" />

</robot>
Copy
```

2.集成 launch 文件

方式 1:先将 xacro 文件转换出 urdf 文件，然后集成

先将 xacro 文件解析成 urdf 文件:`rosrun xacro xacro xxx.xacro > xxx.urdf` 然后再按照之前的集成方式直接整合 launch 文件,内容示例:

```
<launch>
```

```
    <param name="robot_description" textfile="$(find
demo01_urdf_helloworld)/urdf/xacro/my_base.urdf" />

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
demo01_urdf_helloworld)/config/helloworld.rviz" />
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" output="screen" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" />
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui"
output="screen" />

</launch>
```
Copy

方式 2:在 launch 文件中直接加载 xacro(**建议使用**)

launch 内容示例:

```
<launch>
    <param name="robot_description" command="$(find xacro)/xacro $(find
demo01_urdf_helloworld)/urdf/xacro/my_base.urdf.xacro" />

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
demo01_urdf_helloworld)/config/helloworld.rviz" />
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" output="screen" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" />
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui"
output="screen" />

</launch>
```
Copy

核心代码:

```
<param name="robot_description" command="$(find xacro)/xacro $(find
demo01_urdf_helloworld)/urdf/xacro/my_base.urdf.xacro" />
```
Copy

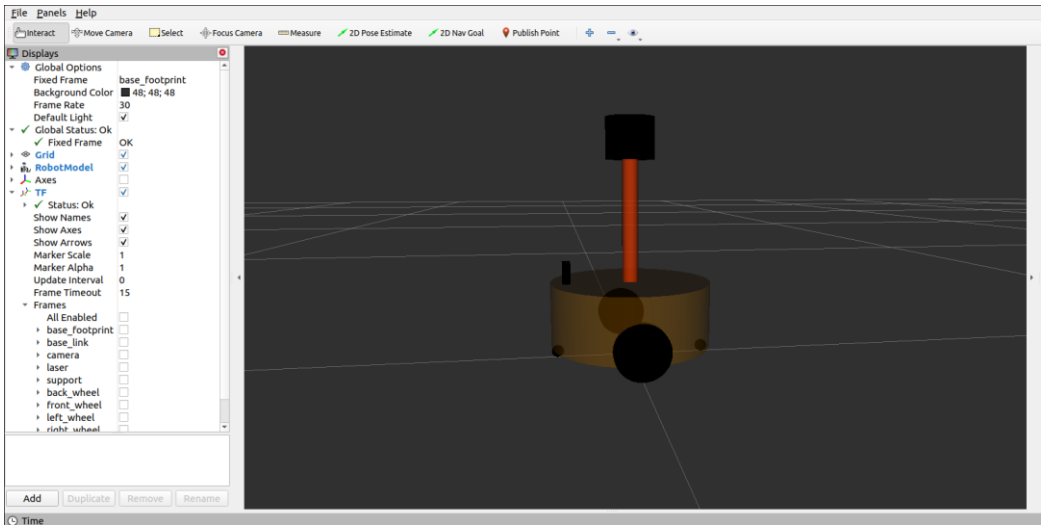
加载 `robot_description` 时使用 `command` 属性，属性值就是调用 xacro 功能包的 xacro 程序直接解析 xacro 文件。

6.4.4 Xacro_实操

需求描述:

在前面小车底盘基础之上，添加摄像头和雷达传感器。

结果演示:



实现分析:

机器人模型由多部件组成，可以将不同组件设置进单独文件，最终通过文件包含实现组件的拼装。

实现流程:

1. 首先编写摄像头和雷达的 xacro 文件
2. 然后再编写一个组合文件，组合底盘、摄像头与雷达
3. 最后，通过 launch 文件启动 Rviz 并显示模型

1.摄像头和雷达 Xacro 文件实现

摄像头 xacro 文件:

```
<!-- 摄像头相关的 xacro 文件 -->
<robot name="my_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <!-- 摄像头属性 -->
    <xacro:property name="camera_length" value="0.01" /> <!-- 摄像头长度
(x) -->
    <xacro:property name="camera_width" value="0.025" /> <!-- 摄像头宽度
(y) -->
```

```
    <xacro:property name="camera_height" value="0.025" /> <!-- 摄像头高度
(z) -->
    <xacro:property name="camera_x" value="0.08" /> <!-- 摄像头安装的 x 坐标
-->
    <xacro:property name="camera_y" value="0.0" /> <!-- 摄像头安装的 y 坐标
-->
    <xacro:property name="camera_z" value="${base_link_length / 2 +
camera_height / 2}" /> <!-- 摄像头安装的 z 坐标:底盘高度 / 2 + 摄像头高度 / 2
-->

    <!-- 摄像头关节以及 link -->
    <link name="camera">
        <visual>
            <geometry>
                <box size="${camera_length} ${camera_width}
${camera_height}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="black" />
        </visual>
    </link>

    <joint name="camera2base_link" type="fixed">
        <parent link="base_link" />
        <child link="camera" />
        <origin xyz="${camera_x} ${camera_y} ${camera_z}" />
    </joint>
</robot>
```
Copy

雷达 xacro 文件:

```
<!--
    小车底盘添加雷达
-->
<robot name="my_laser" xmlns:xacro="http://wiki.ros.org/xacro">

    <!-- 雷达支架 -->
    <xacro:property name="support_length" value="0.15" /> <!-- 支架长度 -
->
    <xacro:property name="support_radius" value="0.01" /> <!-- 支架半径 -
->
    <xacro:property name="support_x" value="0.0" /> <!-- 支架安装的 x 坐标 -
->
    <xacro:property name="support_y" value="0.0" /> <!-- 支架安装的 y 坐标 -
->
```

```
    <xacro:property name="support_z" value="${base_link_length / 2 +
support_length / 2}" /> <!-- 支架安装的 z 坐标:底盘高度 / 2 + 支架高度 / 2  -
->

    <link name="support">
        <visual>
            <geometry>
                <cylinder radius="${support_radius}"
length="${support_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="red">
                <color rgba="0.8 0.2 0.0 0.8" />
            </material>
        </visual>
    </link>

    <joint name="support2base_link" type="fixed">
        <parent link="base_link" />
        <child link="support" />
        <origin xyz="${support_x} ${support_y} ${support_z}" />
    </joint>


    <!-- 雷达属性 -->
    <xacro:property name="laser_length" value="0.05" /> <!-- 雷达长度 -->
    <xacro:property name="laser_radius" value="0.03" /> <!-- 雷达半径 -->
    <xacro:property name="laser_x" value="0.0" /> <!-- 雷达安装的 x 坐标 -->
    <xacro:property name="laser_y" value="0.0" /> <!-- 雷达安装的 y 坐标 -->
    <xacro:property name="laser_z" value="${support_length / 2 +
laser_length / 2}" /> <!-- 雷达安装的 z 坐标:支架高度 / 2 + 雷达高度 / 2  -->

    <!-- 雷达关节以及 link -->
    <link name="laser">
        <visual>
            <geometry>
                <cylinder radius="${laser_radius}"
length="${laser_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="black" />
        </visual>
    </link>
```

```
    <joint name="laser2support" type="fixed">
        <parent link="support" />
        <child link="laser" />
        <origin xyz="${laser_x} ${laser_y} ${laser_z}" />
    </joint>
</robot>
Copy
```

2.组合底盘摄像头与雷达的 xacro 文件

```
<!-- 组合小车底盘与摄像头与雷达 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
</robot>
Copy
```

3.launch 文件

```
<launch>
    <param name="robot_description" command="$(find xacro)/xacro $(find
demo01_urdf_helloworld)/urdf/xacro/my_base_camera_laser.urdf.xacro" />

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
demo01_urdf_helloworld)/config/helloworld.rviz" />
    <node pkg="joint_state_publisher" type="joint_state_publisher"
name="joint_state_publisher" output="screen" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" />
    <node pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" name="joint_state_publisher_gui"
output="screen" />

</launch>
```

6.5 Rviz 中控制机器人模型运动

通过 URDF 结合 rviz 可以创建并显示机器人模型，不过，当前实现的只是静态模型，如何控制模型的运动呢？在此，可以调用 Arbotix 实现此功能。

简介

Arbotix: Arbotix 是一款控制电机、舵机的控制板，**并提供相应的 ros 功能包**，这个功能包的功能不仅可以驱动真实的 Arbotix 控制板，它还提供一个差速控制器，通过接受速度控制指令更新机器人的 joint 状态，从而帮助我们实现机器人在 rviz 中的运动。

这个差速控制器在 arbotix_python 程序包中，完整的 arbotix 程序包还包括多种控制器，分别对应 dynamixel 电机、多关节机械臂以及不同形状的夹持器。

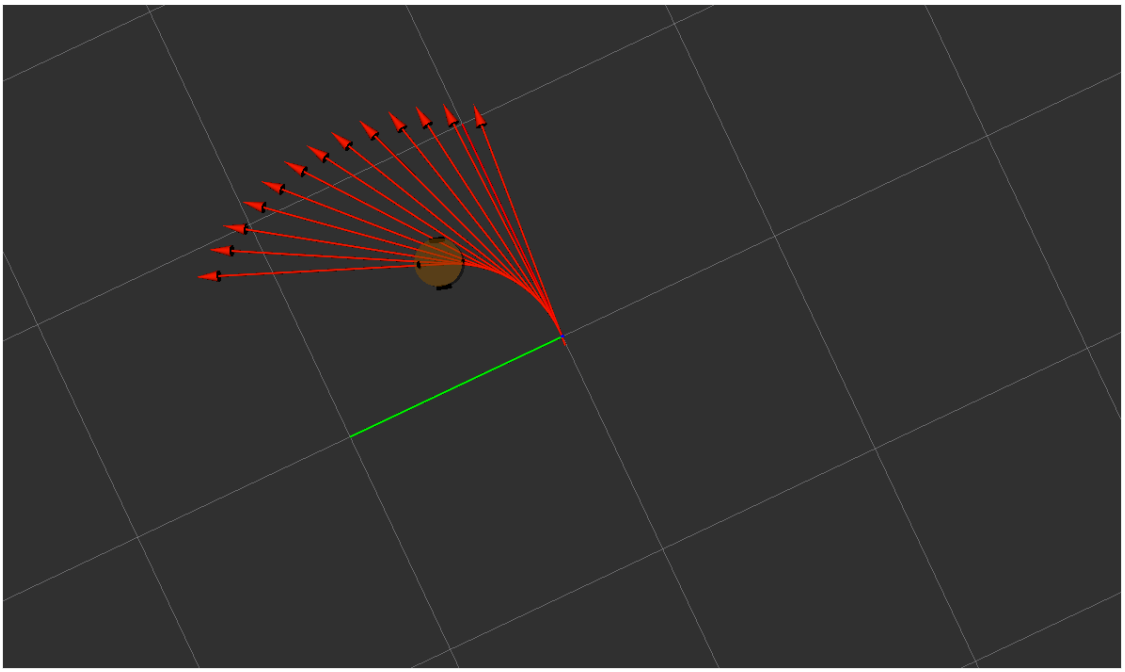
6.5.1 Arbotix 使用流程

接下来，通过一个案例演示 arbotix 的使用。

需求描述:

控制机器人模型在 rviz 中做圆周运动

结果演示:



实现流程:

1. 安装 Arbotix
2. 创建新功能包，准备机器人 urdf、xacro 文件

3. 添加 Arbotix 配置文件

4. 编写 launch 文件配置 Arbotix

5. 启动 launch 文件并控制机器人模型运动

1.安装 Arbotix

方式 1:命令行调用

```
sudo apt-get install ros-<<VersionName()>>-arbotix
Copy
```

将 <<VsersionName()>> 替换成当前 ROS 版本名称，如果提示功能包无法定位，请采用方式 2。

方式 2:源码安装

先从 github 下载源码，然后调用 catkin_make 编译

```
git clone https://github.com/vanadiumlabs/arbotix_ros.git
Copy
```

2.创建新功能包，准备机器人 urdf、xacro

urdf 和 xacro 调用上一讲实现即可

3.添加 arbotix 所需的配置文件

添加 arbotix 所需配置文件

```
# 该文件是控制器配置,一个机器人模型可能有多个控制器，比如：底盘、机械臂、夹持器(机械手)....
# 因此，根 name 是 controller
controllers: {
   # 单控制器设置
   base_controller: {
          #类型：差速控制器
       type: diff_controller,
       #参考坐标
       base_frame_id: base_footprint,
       #两个轮子之间的间距
       base_width: 0.2,
       #控制频率
       ticks_meter: 2000,
       #PID 控制参数，使机器人车轮快速达到预期速度
```

```
        Kp: 12,
        Kd: 12,
        Ki: 0,
        Ko: 50,
        #加速限制
        accel_limit: 1.0
    }
}
Copy
```

另请参考: http://wiki.ros.org/arbotix_python/diff_controller

4.launch 文件中配置 arbotix 节点

launch 示例代码

```
<node name="arbotix" pkg="arbotix_python" type="arbotix_driver"
output="screen">
    <rosparam file="$(find my_urdf05_rviz)/config/hello.yaml"
command="load" />
    <param name="sim" value="true" />
</node>
Copy
```

代码解释:

<node> 调用了 arbotix_python 功能包下的 arbotix_driver 节点

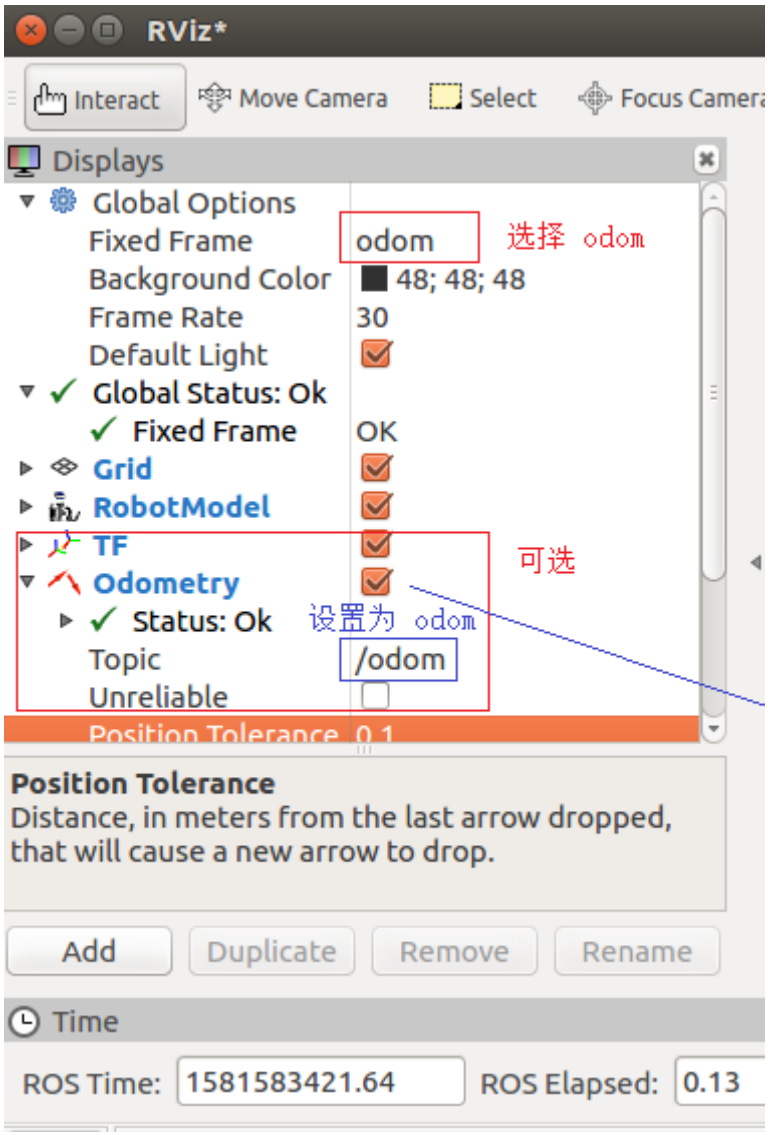
<rosparam> arbotix 驱动机器人运行时，需要获取机器人信息，可以通过 file 加载配置文件

<param> 在仿真环境下，需要配置 sim 为 true

5.启动 launch 文件并控制机器人模型运动

启动 **launch:**roslaunch xxxx ....launch

配置 rviz:



控制小车运动:

此时调用 rostopic list 会发现一个熟悉的话题: /cmd_vel

```
ubuntu@Turtlebot3-virtual-machine: ~
ubuntu@Turtlebot3-virtual-machine:~$ rostopic list
/clicked_point
/cmd_vel
/diagnostics
/initialpose
/joint_states
/move_base_simple/goal
/odom
/rosout
/rosout_agg
/tf
/tf_static
```

也就说我们可以发布 cmd_vel 话题消息控制小陈运动了，该实现策略有多种，可以另行编写节点，或者更简单些可以直接通过如下命令发布消息:

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
Copy
```

现在，小车就可以运动起来了。

另请参考:

- http://wiki.ros.org/arbotix

6.6 URDF 集成 Gazebo

URDF 需要集成进 Rviz 或 Gazebo 才能显示可视化的机器人模型，前面已经介绍了 URDF 与 Rviz 的集成，本节主要介绍:

- URDF 与 Gazebo 的基本集成流程；
- 如果要在 Gazebo 中显示机器人模型，URDF 需要做的一些额外配置；
- 关于 Gazebo 仿真环境的搭建。

6.6.1 URDF 与 Gazebo 基本集成流程

URDF 与 Gazebo 集成流程与 Rviz 实现类似，主要步骤如下:

1. 创建功能包，导入依赖项
2. 编写 URDF 或 Xacro 文件
3. 启动 Gazebo 并显示机器人模型

1.创建功能包

创建新功能包，导入依赖包: urdf、xacro、gazebo_ros、gazebo_ros_control、gazebo_plugins

2.编写 URDF 文件

```
<!--
    创建一个机器人模型(盒状即可)，显示在 Gazebo 中
-->

<robot name="mycar">
    <link name="base_link">
```

```
        <visual>
            <geometry>
                <box size="0.5 0.2 0.1" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="yellow">
                <color rgba="0.5 0.3 0.0 1" />
            </material>
        </visual>
        <collision>
            <geometry>
                <box size="0.5 0.2 0.1" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
        </collision>
        <inertial>
            <origin xyz="0 0 0" />
            <mass value="6" />
            <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1" />
        </inertial>
    </link>
    <gazebo reference="base_link">
        <material>Gazebo/Black</material>
    </gazebo>

</robot>
```
Copy

注意， 当 URDF 需要与 Gazebo 集成时，和 Rviz 有明显区别:

1.必须使用 collision 标签，因为既然是仿真环境，那么必然涉及到碰撞检测，collision 提供碰撞检测的依据。

2.必须使用 inertial 标签，此标签标注了当前机器人某个刚体部分的惯性矩阵，用于一些力学相关的仿真计算。

3.颜色设置，也需要重新使用 gazebo 标签标注，因为之前的颜色设置为了方便调试包含透明度，仿真环境下没有此选项。

3.启动 Gazebo 并显示模型

launch 文件实现:

```
<launch>

    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
```

```
    <param name="robot_description" textfile="$(find
demo02_urdf_gazebo)/urdf/urdf01_helloworld.urdf" />

    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <!-- 在 gazebo 中显示机器人模型 -->
    <node pkg="gazebo_ros" type="spawn_model" name="model" args="-urdf -
model mycar -param robot_description"  />
</launch>
```
Copy

代码解释:

```
<include file="$(find gazebo_ros)/launch/empty_world.launch" />
<!-- 启动 Gazebo 的仿真环境，当前环境为空环境 -->
```
Copy
```
<node pkg="gazebo_ros" type="spawn_model" name="model" args="-urdf -model
mycar -param robot_description"  />

<!--
    在 Gazebo 中加载一个机器人模型，该功能由 gazebo_ros 下的 spawn_model 提
供:
    -urdf 加载的是 urdf 文件
    -model mycar 模型名称是 mycar
    -param robot_description 从参数 robot_description 中载入模型
    -x 模型载入的 x 坐标
    -y 模型载入的 y 坐标
    -z 模型载入的 z 坐标
-->
```

6.6.2 URDF 集成 Gazebo 相关设置

较之于 rviz，gazebo 在集成 URDF 时，需要做些许修改，比如:必须添加 collision 碰撞属性相关参数、必须添加 inertial 惯性矩阵相关参数，另外，如果直接移植 Rviz 中机器人的颜色设置是没有显示的，颜色设置也必须做相应的变更。

1.collision

如果机器人 link 是标准的几何体形状，和 link 的 visual 属性设置一致即可。

2.inertial

惯性矩阵的设置需要结合 link 的质量与外形参数动态生成，标准的球体、圆柱与立方体的惯性矩阵公式如下(已经封装为 xacro 实现):

球体惯性矩阵

```
<!-- Macro for inertia matrix -->
    <xacro:macro name="sphere_inertial_matrix" params="m r">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${2*m*r*r/5}" ixy="0" ixz="0"
                iyy="${2*m*r*r/5}" iyz="0"
                izz="${2*m*r*r/5}" />
        </inertial>
    </xacro:macro>
Copy
```

圆柱惯性矩阵

```
<xacro:macro name="cylinder_inertial_matrix" params="m r h">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
                iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
                izz="${m*r*r/2}" />
        </inertial>
    </xacro:macro>
Copy
```

立方体惯性矩阵

```
 <xacro:macro name="Box_inertial_matrix" params="m l w h">
      <inertial>
              <mass value="${m}" />
              <inertia ixx="${m*(h*h + l*l)/12}" ixy = "0" ixz = "0"
                  iyy="${m*(w*w + l*l)/12}" iyz= "0"
                  izz="${m*(w*w + h*h)/12}" />
      </inertial>
   </xacro:macro>
Copy
```

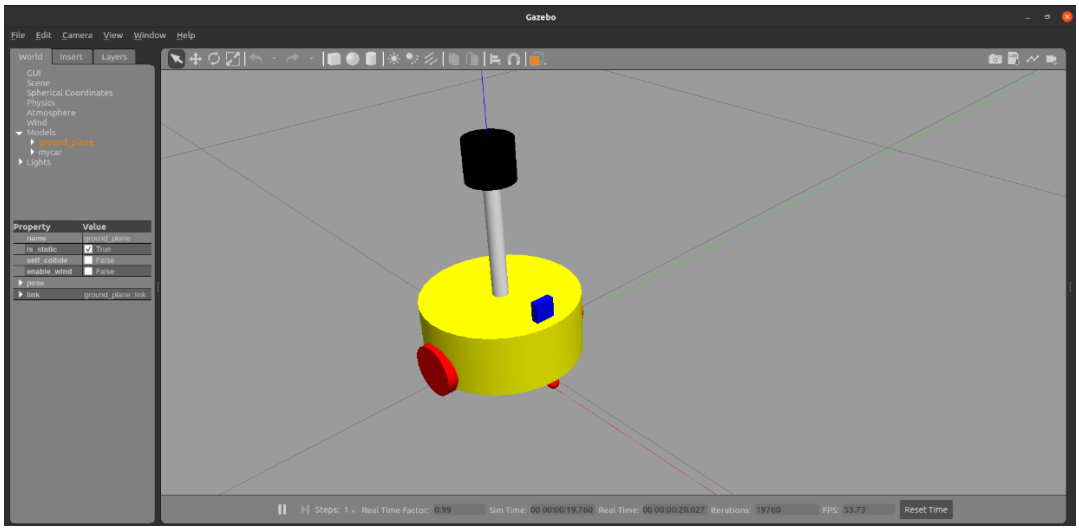
需要注意的是，原则上，除了 base_footprint 外，机器人的每个刚体部分都需要设置惯性矩阵，且惯性矩阵必须经计算得出，如果随意定义刚体部分的惯性矩阵，那么可能会导致机器人在 Gazebo 中出现抖动，移动等现象。

6.6.3 URDF 集成 Gazebo 实操

需求描述:

将之前的机器人模型(xacro 版)显示在 gazebo 中

结果演示:



实现流程:

1. 需要编写封装惯性矩阵算法的 xacro 文件
2. 为机器人模型中的每一个 link 添加 collision 和 inertial 标签，并且重置颜色属性
3. 在 launch 文件中启动 gazebo 并添加机器人模型

1.编写封装惯性矩阵算法的 xacro 文件

```
<robot name="base" xmlns:xacro="http://wiki.ros.org/xacro">
    <!-- Macro for inertia matrix -->
    <xacro:macro name="sphere_inertial_matrix" params="m r">
        <inertial>
            <mass value="${m}" />
            <inertia ixx="${2*m*r*r/5}" ixy="0" ixz="0"
                iyy="${2*m*r*r/5}" iyz="0"
                izz="${2*m*r*r/5}" />
        </inertial>
    </xacro:macro>

    <xacro:macro name="cylinder_inertial_matrix" params="m r h">
        <inertial>
```

```
            <mass value="${m}" />
            <inertia ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
                iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
                izz="${m*r*r/2}" />
        </inertial>
    </xacro:macro>

    <xacro:macro name="Box_inertial_matrix" params="m l w h">
       <inertial>
              <mass value="${m}" />
              <inertia ixx="${m*(h*h + l*l)/12}" ixy = "0" ixz = "0"
                  iyy="${m*(w*w + l*l)/12}" iyz= "0"
                  izz="${m*(w*w + h*h)/12}" />
       </inertial>
   </xacro:macro>
</robot>
Copy
```

2.复制相关 xacro 文件，并设置 collision inertial 以及 color 等参数

A.底盘 Xacro 文件

```
<!--
    使用 xacro 优化 URDF 版的小车底盘实现：

    实现思路:
    1.将一些常量、变量封装为 xacro:property
      比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度 ....
    2.使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动轮与支撑轮

-->
<!-- 根标签，必须声明 xmlns:xacro -->
<robot name="my_base" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <!-- 封装变量、常量 -->
    <!-- PI 值设置精度需要高一些，否则后续车轮翻转量计算时，可能会出现肉眼不能察觉的车轮倾斜，从而导致模型抖动 -->
    <xacro:property name="PI" value="3.1415926"/>
    <!-- 宏:黑色设置 -->
    <material name="black">
        <color rgba="0.0 0.0 0.0 1.0" />
    </material>
```

```
    <!-- 底盘属性 -->
    <xacro:property name="base_footprint_radius" value="0.001" /> <!--
base_footprint 半径  -->
    <xacro:property name="base_link_radius" value="0.1" /> <!-- base_link
半径 -->
    <xacro:property name="base_link_length" value="0.08" /> <!--
base_link 长 -->
    <xacro:property name="earth_space" value="0.015" /> <!-- 离地间距 -->
    <xacro:property name="base_link_m" value="0.5" /> <!-- 质量  -->

    <!-- 底盘 -->
    <link name="base_footprint">
      <visual>
        <geometry>
          <sphere radius="${base_footprint_radius}" />
        </geometry>
      </visual>
    </link>

    <link name="base_link">
      <visual>
        <geometry>
          <cylinder radius="${base_link_radius}"
length="${base_link_length}" />
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <material name="yellow">
          <color rgba="0.5 0.3 0.0 0.5" />
        </material>
      </visual>
      <collision>
        <geometry>
          <cylinder radius="${base_link_radius}"
length="${base_link_length}" />
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
      </collision>
      <xacro:cylinder_inertial_matrix m="${base_link_m}"
r="${base_link_radius}" h="${base_link_length}" />

    </link>


    <joint name="base_link2base_footprint" type="fixed">
```

```
      <parent link="base_footprint" />
      <child link="base_link" />
      <origin xyz="0 0 ${earth_space + base_link_length / 2 }" />
    </joint>
    <gazebo reference="base_link">
        <material>Gazebo/Yellow</material>
    </gazebo>

    <!-- 驱动轮 -->
    <!-- 驱动轮属性 -->
    <xacro:property name="wheel_radius" value="0.0325" /><!-- 半径 -->
    <xacro:property name="wheel_length" value="0.015" /><!-- 宽度 -->
    <xacro:property name="wheel_m" value="0.05" /> <!-- 质量  -->

    <!-- 驱动轮宏实现 -->
    <xacro:macro name="add_wheels" params="name flag">
      <link name="${name}_wheel">
        <visual>
          <geometry>
            <cylinder radius="${wheel_radius}" length="${wheel_length}" />
          </geometry>
          <origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />
          <material name="black" />
        </visual>
        <collision>
          <geometry>
            <cylinder radius="${wheel_radius}" length="${wheel_length}" />
          </geometry>
          <origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />
        </collision>
        <xacro:cylinder_inertial_matrix m="${wheel_m}" r="${wheel_radius}"
h="${wheel_length}" />

      </link>

      <joint name="${name}_wheel2base_link" type="continuous">
        <parent link="base_link" />
        <child link="${name}_wheel" />
        <origin xyz="0 ${flag * base_link_radius} ${-(earth_space +
base_link_length / 2 - wheel_radius) }" />
        <axis xyz="0 1 0" />
      </joint>

      <gazebo reference="${name}_wheel">
```

```
        <material>Gazebo/Red</material>
      </gazebo>

    </xacro:macro>
    <xacro:add_wheels name="left" flag="1" />
    <xacro:add_wheels name="right" flag="-1" />
    <!-- 支撑轮 -->
    <!-- 支撑轮属性 -->
    <xacro:property name="support_wheel_radius" value="0.0075" /> <!-- 支撑轮半径 -->
    <xacro:property name="support_wheel_m" value="0.03" /> <!-- 质量  -->

    <!-- 支撑轮宏 -->
    <xacro:macro name="add_support_wheel" params="name flag" >
      <link name="${name}_wheel">
        <visual>
            <geometry>
                <sphere radius="${support_wheel_radius}" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <material name="black" />
        </visual>
        <collision>
            <geometry>
                <sphere radius="${support_wheel_radius}" />
            </geometry>
            <origin xyz="0 0 0" rpy="0 0 0" />
        </collision>
        <xacro:sphere_inertial_matrix m="${support_wheel_m}" r="${support_wheel_radius}" />
      </link>

      <joint name="${name}_wheel2base_link" type="continuous">
          <parent link="base_link" />
          <child link="${name}_wheel" />
          <origin xyz="${flag * (base_link_radius - support_wheel_radius)} 0 ${-(base_link_length / 2 + earth_space / 2)}" />
          <axis xyz="1 1 1" />
      </joint>
      <gazebo reference="${name}_wheel">
        <material>Gazebo/Red</material>
      </gazebo>
    </xacro:macro>
```

```
    <xacro:add_support_wheel name="front" flag="1" />
    <xacro:add_support_wheel name="back" flag="-1" />


</robot>
```
Copy

注意: 如果机器人模型在 Gazebo 中产生了抖动，滑动，缓慢位移 .... 诸如此类情况，请查看

1. 惯性矩阵是否设置了，且设置是否正确合理
2. 车轮翻转需要依赖于 PI 值，如果 PI 值精度偏低，也可能导致上述情况产生

B.摄像头 Xacro 文件

```
<!-- 摄像头相关的 xacro 文件 -->
<robot name="my_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <!-- 摄像头属性 -->
    <xacro:property name="camera_length" value="0.01" /> <!-- 摄像头长度
(x) -->
    <xacro:property name="camera_width" value="0.025" /> <!-- 摄像头宽度
(y) -->
    <xacro:property name="camera_height" value="0.025" /> <!-- 摄像头高度
(z) -->
    <xacro:property name="camera_x" value="0.08" /> <!-- 摄像头安装的 x 坐标
-->
    <xacro:property name="camera_y" value="0.0" /> <!-- 摄像头安装的 y 坐标
-->
    <xacro:property name="camera_z" value="${base_link_length / 2 +
camera_height / 2}" /> <!-- 摄像头安装的 z 坐标:底盘高度 / 2 + 摄像头高度 / 2
-->

    <xacro:property name="camera_m" value="0.01" /> <!-- 摄像头质量 -->

    <!-- 摄像头关节以及 link -->
    <link name="camera">
        <visual>
            <geometry>
                <box size="${camera_length} ${camera_width}
${camera_height}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="black" />
        </visual>
```

```
        <collision>
            <geometry>
                <box size="${camera_length} ${camera_width}
${camera_height}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
        </collision>
        <xacro:Box_inertial_matrix m="${camera_m}" l="${camera_length}"
w="${camera_width}" h="${camera_height}" />
    </link>

    <joint name="camera2base_link" type="fixed">
        <parent link="base_link" />
        <child link="camera" />
        <origin xyz="${camera_x} ${camera_y} ${camera_z}" />
    </joint>
    <gazebo reference="camera">
        <material>Gazebo/Blue</material>
    </gazebo>
</robot>
```
Copy

C.雷达 Xacro 文件

```
<!--
    小车底盘添加雷达
-->
<robot name="my_laser" xmlns:xacro="http://wiki.ros.org/xacro">

    <!-- 雷达支架 -->
    <xacro:property name="support_length" value="0.15" /> <!-- 支架长度 -
->
    <xacro:property name="support_radius" value="0.01" /> <!-- 支架半径 -
->
    <xacro:property name="support_x" value="0.0" /> <!-- 支架安装的 x 坐标 -
->
    <xacro:property name="support_y" value="0.0" /> <!-- 支架安装的 y 坐标 -
->
    <xacro:property name="support_z" value="${base_link_length / 2 +
support_length / 2}" /> <!-- 支架安装的 z 坐标:底盘高度 / 2 + 支架高度 / 2  -
->

    <xacro:property name="support_m" value="0.02" /> <!-- 支架质量 -->
```

```
    <link name="support">
        <visual>
            <geometry>
                <cylinder radius="${support_radius}"
length="${support_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="red">
                <color rgba="0.8 0.2 0.0 0.8" />
            </material>
        </visual>

        <collision>
            <geometry>
                <cylinder radius="${support_radius}"
length="${support_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
        </collision>

        <xacro:cylinder_inertial_matrix m="${support_m}"
r="${support_radius}" h="${support_length}" />

    </link>

    <joint name="support2base_link" type="fixed">
        <parent link="base_link" />
        <child link="support" />
        <origin xyz="${support_x} ${support_y} ${support_z}" />
    </joint>

    <gazebo reference="support">
        <material>Gazebo/White</material>
    </gazebo>

    <!-- 雷达属性 -->
    <xacro:property name="laser_length" value="0.05" /> <!-- 雷达长度 -->
    <xacro:property name="laser_radius" value="0.03" /> <!-- 雷达半径 -->
    <xacro:property name="laser_x" value="0.0" /> <!-- 雷达安装的 x 坐标 -->
    <xacro:property name="laser_y" value="0.0" /> <!-- 雷达安装的 y 坐标 -->
    <xacro:property name="laser_z" value="${support_length / 2 +
laser_length / 2}" /> <!-- 雷达安装的 z 坐标:支架高度 / 2 + 雷达高度 / 2  -->

    <xacro:property name="laser_m" value="0.1" /> <!-- 雷达质量 -->
```

```
    <!-- 雷达关节以及 link -->
    <link name="laser">
        <visual>
            <geometry>
                <cylinder radius="${laser_radius}"
length="${laser_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
            <material name="black" />
        </visual>
        <collision>
            <geometry>
                <cylinder radius="${laser_radius}"
length="${laser_length}" />
            </geometry>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
        </collision>
        <xacro:cylinder_inertial_matrix m="${laser_m}" r="${laser_radius}"
h="${laser_length}" />
    </link>

    <joint name="laser2support" type="fixed">
        <parent link="support" />
        <child link="laser" />
        <origin xyz="${laser_x} ${laser_y} ${laser_z}" />
    </joint>
    <gazebo reference="laser">
        <material>Gazebo/Black</material>
    </gazebo>
</robot>
Copy
```

D.组合底盘、摄像头与雷达的 Xacro 文件

```
<!-- 组合小车底盘与摄像头 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_head.urdf.xacro" />
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
</robot>
Copy
```

3.在 gazebo 中执行

launch 文件:

```
<launch>
    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
    <param name="robot_description" command="$(find xacro)/xacro $(find demo02_urdf_gazebo)/urdf/xacro/my_base_camera_laser.urdf.xacro" />
    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <!-- 在 gazebo 中显示机器人模型 -->
    <node pkg="gazebo_ros" type="spawn_model" name="model" args="-urdf -model mycar -param robot_description"  />
</launch>
```

6.6.4 Gazebo 仿真环境搭建

到目前为止，我们已经可以将机器人模型显示在 Gazebo 之中了，但是当前默认情况下，在 Gazebo 中机器人模型是在 empty world 中，并没有类似于房间、家具、道路、树木... 之类的仿真物，如何在 Gazebo 中创建仿真环境呢？

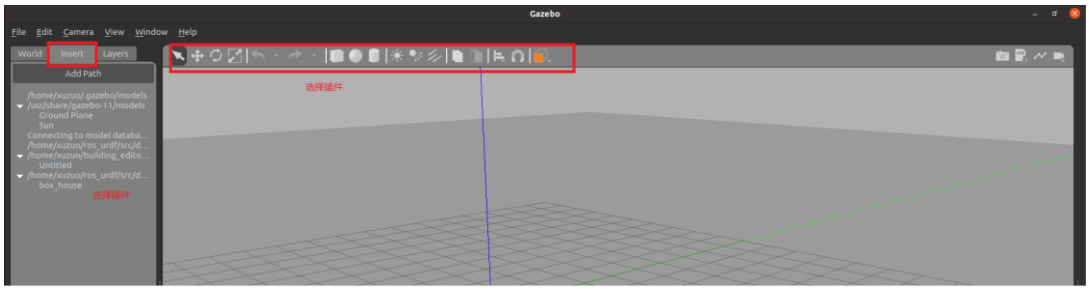
Gazebo 中创建仿真实现方式有两种:

- 方式 1: 直接添加内置组件创建仿真环境
- 方式 2: 手动绘制仿真环境(更为灵活)

也还可以直接下载使用官方或第三方提高的仿真环境插件。

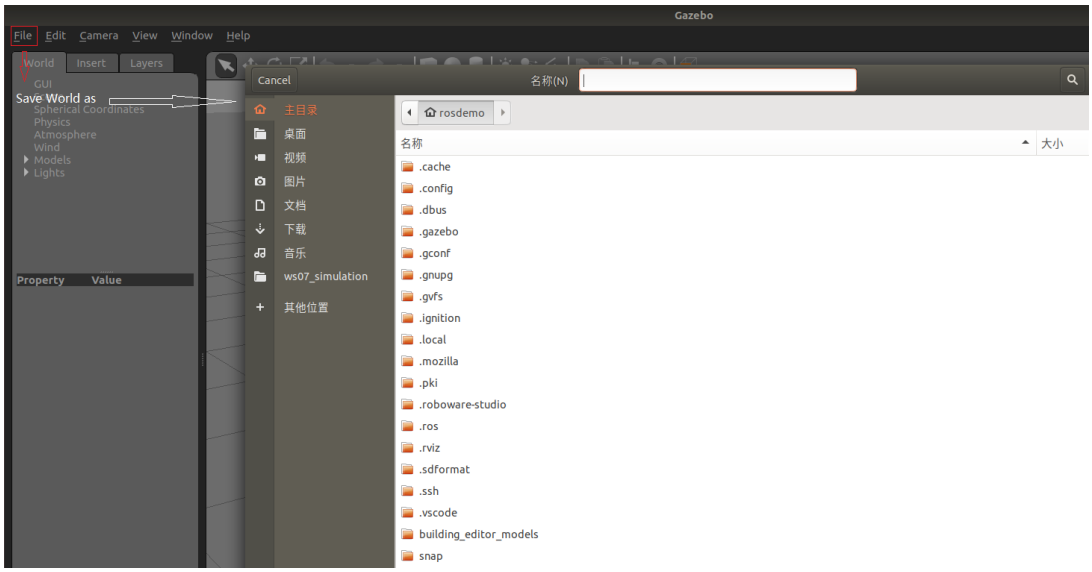
1.添加内置组件创建仿真环境

1.1 启动 Gazebo 并添加组件



1.2 保存仿真环境

添加完毕后，选择 file ---> Save World as 选择保存路径(功能包下: worlds 目录)，文件名自定义，后缀名设置为 .world



1.3 启动

```
<launch>

    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
    <param name="robot_description" command="$(find xacro)/xacro $(find demo02_urdf_gazebo)/urdf/xacro/my_base_camera_laser.urdf.xacro" />
    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find demo02_urdf_gazebo)/worlds/hello.world" />
```

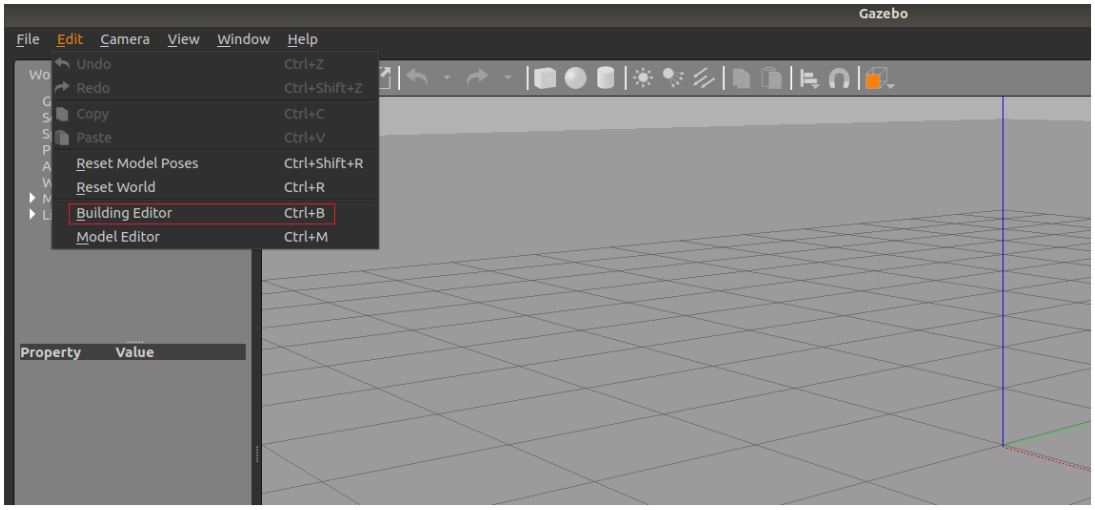
```
    </include>

    <!-- 在 gazebo 中显示机器人模型 -->
    <node pkg="gazebo_ros" type="spawn_model" name="model" args="-urdf -
model mycar -param robot_description"  />
</launch>
```
Copy

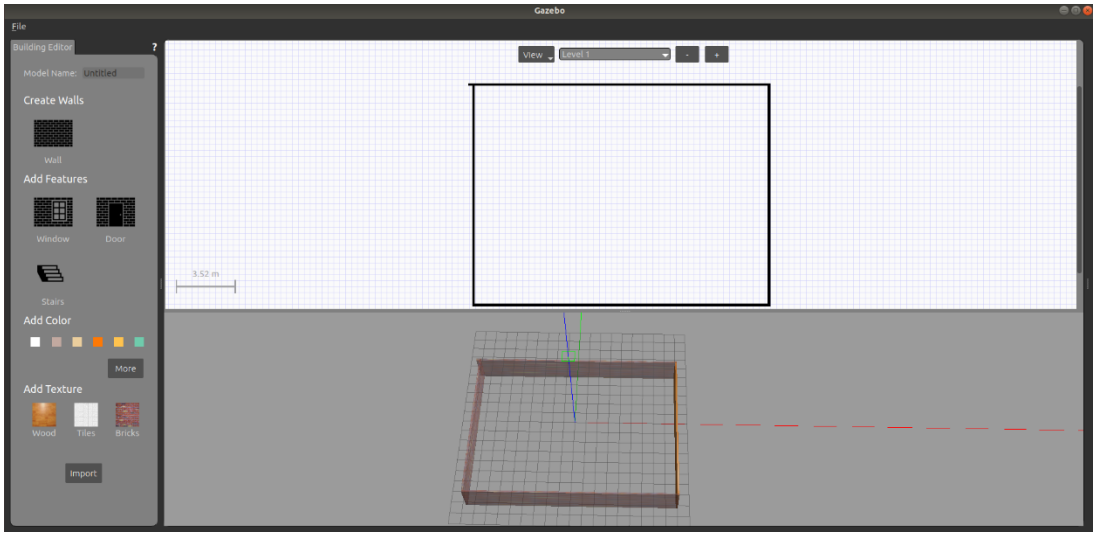
核心代码：启动 empty_world 后，再根据 `arg` 加载自定义的仿真环境

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
demo02_urdf_gazebo)/worlds/hello.world" />
</include>
```
Copy

2.自定义仿真环境

2.1 启动 gazebo 打开构建面板，绘制仿真环境





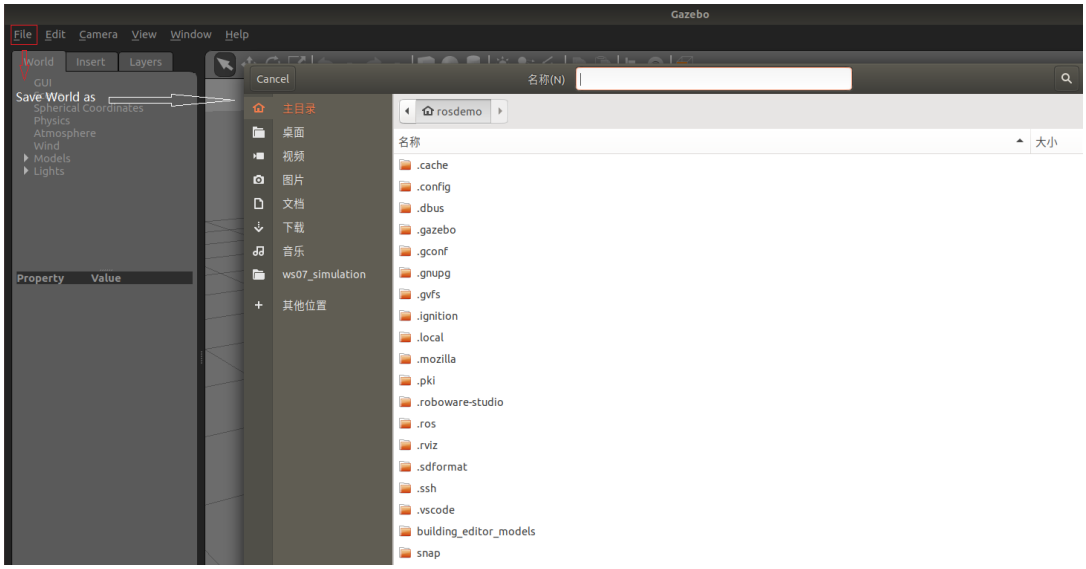
2.2 保存构建的环境

点击: 左上角 file ---> Save (保存路径功能包下的: models)

然后 file ---> Exit Building Editor

2.3 保存为 world 文件

可以像方式 1 一样再添加一些插件，然后保存为 world 文件(保存路径功能包下的: worlds)



2.4 启动

同方式 1

3.使用官方提供的插件

当前 Gazebo 提供的仿真道具有限，还可以下载官方支持，可以提供更为丰富的仿真实现，具体实现如下:

3.1 下载官方模型库

`git clone https://github.com/osrf/gazebo_models`
之前是:`hg clone https://bitbucket.org/osrf/gazebo_models` 但是已经不可用
注意: 此过程可能比较耗时

3.2 将模型库复制进 gazebo

将得到的 gazebo_models 文件夹内容复制到 /usr/share/gazebo-*/models

3.3 应用

重启 Gazebo，选择左侧菜单栏的 insert 可以选择并插入相关道具了

6.7 URDF、Gazebo 与 Rviz 综合应用

关于 URDF(Xacro)、Rviz 和 Gazebo 三者的关系，前面已有阐述: URDF 用于创建机器人模型、Rviz 可以显示机器人感知到的环境信息，Gazebo 用

于仿真，可以模拟外界环境，以及机器人的一些传感器，如何在 Gazebo 中运行这些传感器，并显示这些传感器的数据(机器人的视角)呢？本节主要介绍的重点就是将三者结合:通过 Gazebo 模拟机器人的传感器，然后在 Rviz 中显示这些传感器感知到的数据。主要内容包括:

- 运动控制以及里程计信息显示
- 雷达信息仿真以及显示
- 摄像头信息仿真以及显示
- kinect 信息仿真以及显示

另请参考:

- http://gazebosim.org/tutorials?tut=ros_gzplugins

6.7.1 机器人运动控制以及里程计信息显示

gazebo 中已经可以正常显示机器人模型了，那么如何像在 rviz 中一样控制机器人运动呢？在此，需要涉及到 ros 中的组件: ros_control。

1.ros_control 简介

场景:同一套 ROS 程序，如何部署在不同的机器人系统上，比如：开发阶段为了提高效率是在仿真平台上测试的，部署时又有不同的实体机器人平台，不同平台的实现是有差异的，如何保证 ROS 程序的可移植性？ROS 内置的解决方式是 ros_control。

ros_control:是一组软件包，它包含了控制器接口，控制器管理器，传输和硬件接口。ros_control 是一套机器人控制的中间件，是一套规范，不同的机器人平台只要按照这套规范实现，那么就可以保证 与 ROS 程序兼容，通过这套规范，实现了一种可插拔的架构设计，大大提高了程序设计的效率与灵活性。

gazebo 已经实现了 ros_control 的相关接口，如果需要在 gazebo 中控制机器人运动，直接调用相关接口即可

2.运动控制实现流程(Gazebo)

承上，运动控制基本流程:

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加传动装置以及控制器

2. 将此文件集成进 xacro 文件
3. 启动 Gazebo 并发布 /cmd_vel 消息控制机器人运动

2.1 为 joint 添加传动装置以及控制器

两轮差速配置

```
<robot name="my_car_move" xmlns:xacro="http://wiki.ros.org/xacro">

    <!-- 传动实现:用于连接控制器与关节 -->
    <xacro:macro name="joint_trans" params="joint_name">
        <!-- Transmission is important to link the joints and the
controller -->
        <transmission name="${joint_name}_trans">
            <type>transmission_interface/SimpleTransmission</type>
            <joint name="${joint_name}">

<hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInt
erface>
            </joint>
            <actuator name="${joint_name}_motor">

<hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInt
erface>
                <mechanicalReduction>1</mechanicalReduction>
            </actuator>
        </transmission>
    </xacro:macro>

    <!-- 每一个驱动轮都需要配置传动装置 -->
    <xacro:joint_trans joint_name="left_wheel2base_link" />
    <xacro:joint_trans joint_name="right_wheel2base_link" />

    <!-- 控制器 -->
    <gazebo>
        <plugin name="differential_drive_controller"
filename="libgazebo_ros_diff_drive.so">
            <rosDebugLevel>Debug</rosDebugLevel>
            <publishWheelTF>true</publishWheelTF>
            <robotNamespace>/</robotNamespace>
            <publishTf>1</publishTf>
            <publishWheelJointState>true</publishWheelJointState>
            <alwaysOn>true</alwaysOn>
            <updateRate>100.0</updateRate>
```

```
            <legacyMode>true</legacyMode>
            <leftJoint>left_wheel2base_link</leftJoint> <!-- 左轮 -->
            <rightJoint>right_wheel2base_link</rightJoint> <!-- 右轮 -->
            <wheelSeparation>${base_link_radius * 2}</wheelSeparation> <!-
- 车轮间距 -->
            <wheelDiameter>${wheel_radius * 2}</wheelDiameter> <!-- 车轮直
径 -->
            <broadcastTF>1</broadcastTF>
            <wheelTorque>30</wheelTorque>
            <wheelAcceleration>1.8</wheelAcceleration>
            <commandTopic>cmd_vel</commandTopic> <!-- 运动控制话题 -->
            <odometryFrame>odom</odometryFrame>
            <odometryTopic>odom</odometryTopic> <!-- 里程计话题 -->
            <robotBaseFrame>base_footprint</robotBaseFrame> <!-- 根坐标系 -
->
        </plugin>
    </gazebo>

</robot>
Copy
```

2.2 xacro 文件集成

最后还需要将上述 xacro 文件集成进总的机器人模型文件，代码示例如下:

```
<!-- 组合小车底盘与摄像头 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_head.urdf.xacro" />
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
    <xacro:include filename="move.urdf.xacro" />
</robot>
Copy
```

当前核心: 包含 控制器以及传动配置的 xacro 文件

```
<xacro:include filename="move.urdf.xacro" />
Copy
```

2.3 启动 gazebo 并控制机器人运动

launch 文件:

```
<launch>
```

```
    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
    <param name="robot_description" command="$(find xacro)/xacro $(find
demo02_urdf_gazebo)/urdf/xacro/my_base_camera_laser.urdf.xacro" />
    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find
demo02_urdf_gazebo)/worlds/hello.world" />
    </include>

    <!-- 在 gazebo 中显示机器人模型 -->
    <node pkg="gazebo_ros" type="spawn_model" name="model" args="-urdf -
model mycar -param robot_description"  />
</launch>
```

Copy

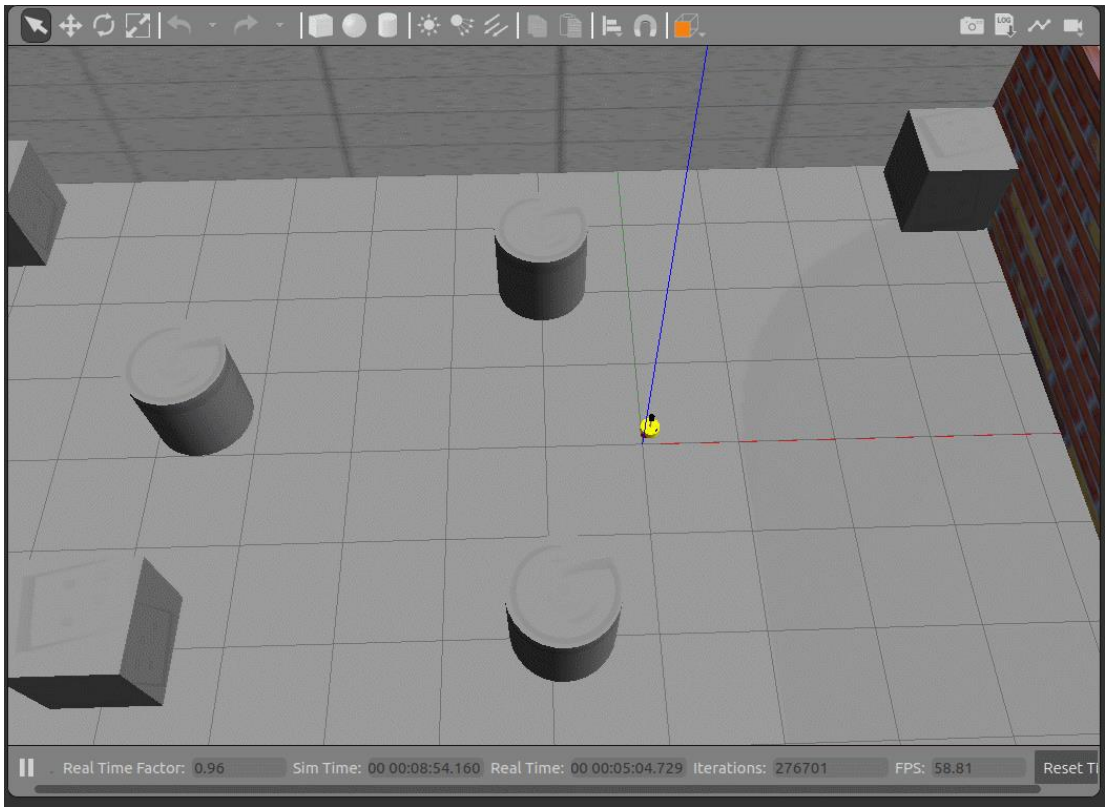
启动 launch 文件，使用 topic list 查看话题列表，会发现多了 /cmd_vel 然后发布 vmd_vel 消息控制即可

使用命令控制(或者可以编写单独的节点控制)

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0,
z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
```

Copy

接下来我们会发现：小车在 Gazebo 中已经正常运行起来了



3.Rviz 查看里程计信息

在 Gazebo 的仿真环境中，机器人的里程计信息以及运动朝向等信息是无法获取的，可以通过 Rviz 显示机器人的里程计信息以及运动朝向

里程计: 机器人相对出发点坐标系的位姿状态(X 坐标 Y 坐标 Z 坐标以及朝向)。

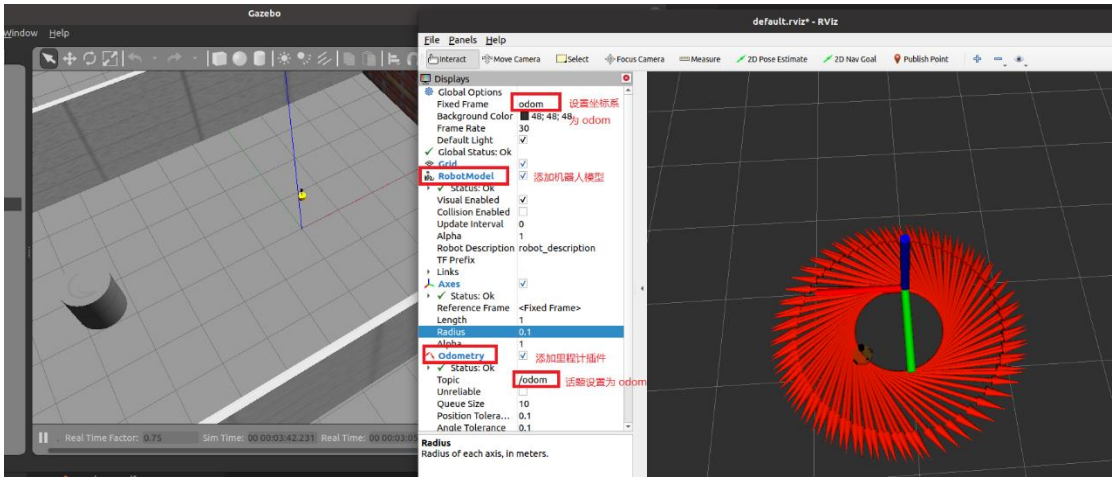
3.1 启动 Rviz

launch 文件

```
<launch>
    <!-- 启动 rviz -->
    <node pkg="rviz" type="rviz" name="rviz" />

    <!-- 关节以及机器人状态发布节点 -->
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
    <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />

</launch>
Copy
```

3.2 添加组件

执行 launch 文件后，在 Rviz 中添加图示组件:



6.7.2 雷达信息仿真以及显示

通过 Gazebo 模拟激光雷达传感器，并在 Rviz 中显示激光数据。

实现流程:

雷达仿真基本流程:

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加雷达配置；
2. 将此文件集成进 xacro 文件；
3. 启动 Gazebo，使用 Rviz 显示雷达信息。

1.Gazebo 仿真雷达

1.1 新建 Xacro 文件，配置雷达传感器信息

```
<robot name="my_sensors" xmlns:xacro="http://wiki.ros.org/xacro">

  <!-- 雷达 -->
  <gazebo reference="laser">
    <sensor type="ray" name="rplidar">
      <pose>0 0 0 0 0 0</pose>
      <visualize>true</visualize>
      <update_rate>5.5</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>360</samples>
            <resolution>1</resolution>
            <min_angle>-3</min_angle>
            <max_angle>3</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.10</min>
          <max>30.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="gazebo_rplidar" filename="libgazebo_ros_laser.so">
        <topicName>/scan</topicName>
```

```
        <frameName>laser</frameName>
      </plugin>
    </sensor>
  </gazebo>

</robot>
Copy
```

1.2 xacro 文件集成

将步骤 1 的 Xacro 文件集成进总的机器人模型文件，代码示例如下:

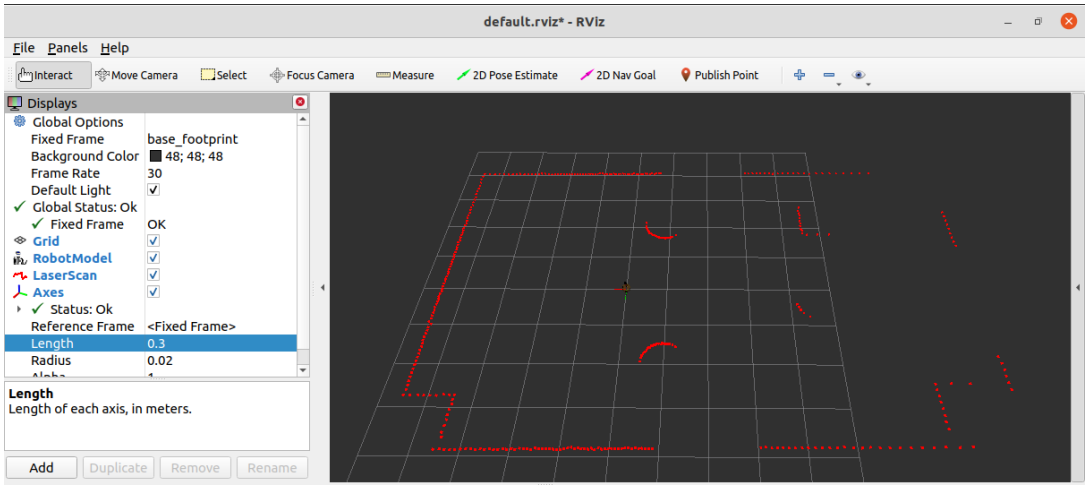
```
<!-- 组合小车底盘与传感器 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_head.urdf.xacro" />
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
    <xacro:include filename="move.urdf.xacro" />
    <!-- 雷达仿真的 xacro 文件 -->
    <xacro:include filename="my_sensors_laser.urdf.xacro" />
</robot>
Copy
```

1.3 启动仿真环境

编写 launch 文件，启动 gazebo，此处略...

2.Rviz 显示雷达数据

先启动 rviz,添加雷达信息显示插件



6.7.3 摄像头信息仿真以及显示

通过 Gazebo 模拟摄像头传感器，并在 Rviz 中显示摄像头数据。

实现流程:

摄像头仿真基本流程:

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加摄像头配置；
2. 将此文件集成进 xacro 文件；
3. 启动 Gazebo，使用 Rviz 显示摄像头信息。

1.Gazebo 仿真摄像头

1.1 新建 Xacro 文件，配置摄像头传感器信息

```
<robot name="my_sensors" xmlns:xacro="http://wiki.ros.org/xacro">
  <!-- 被引用的 link -->
  <gazebo reference="camera">
    <!-- 类型设置为 camara -->
    <sensor type="camera" name="camera_node">
      <update_rate>30.0</update_rate> <!-- 更新频率 -->
      <!-- 摄像头基本信息设置 -->
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>1280</width>
          <height>720</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
```

```
      </camera>
      <!-- 核心插件 -->
      <plugin name="gazebo_camera" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>/camera</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera</frameName>
        <hackBaseline>0.07</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
      </plugin>
    </sensor>
  </gazebo>
</robot>
Copy
```

1.2 xacro 文件集成

将步骤 1 的 Xacro 文件集成进总的机器人模型文件，代码示例如下:

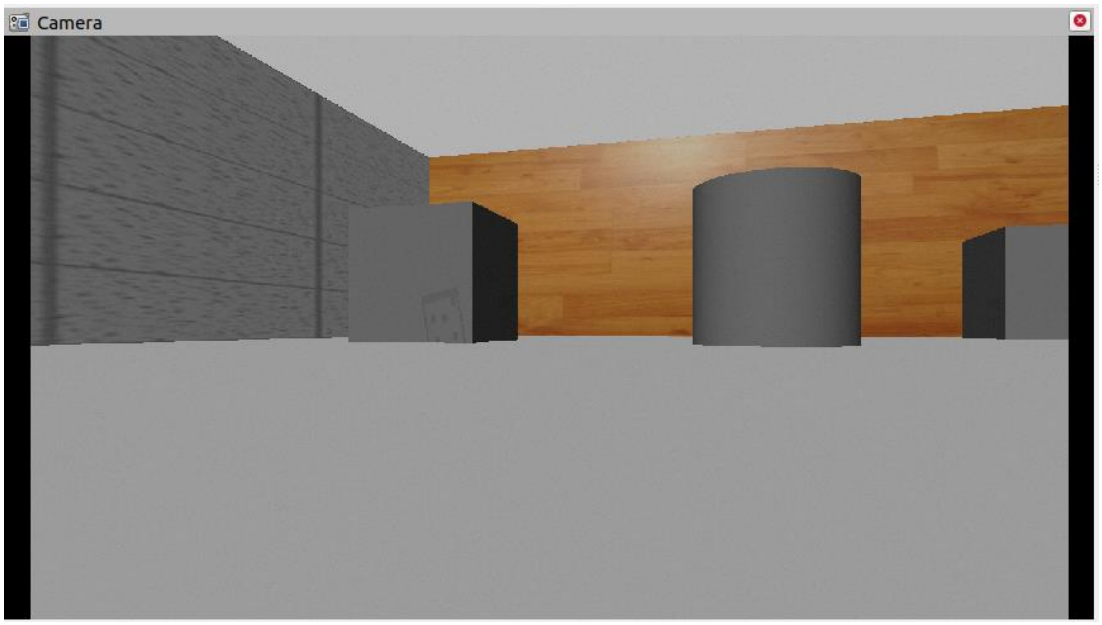
```
<!-- 组合小车底盘与传感器 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_head.urdf.xacro" />
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
    <xacro:include filename="move.urdf.xacro" />
    <!-- 摄像头仿真的 xacro 文件 -->
    <xacro:include filename="my_sensors_camara.urdf.xacro" />
</robot>
Copy
```

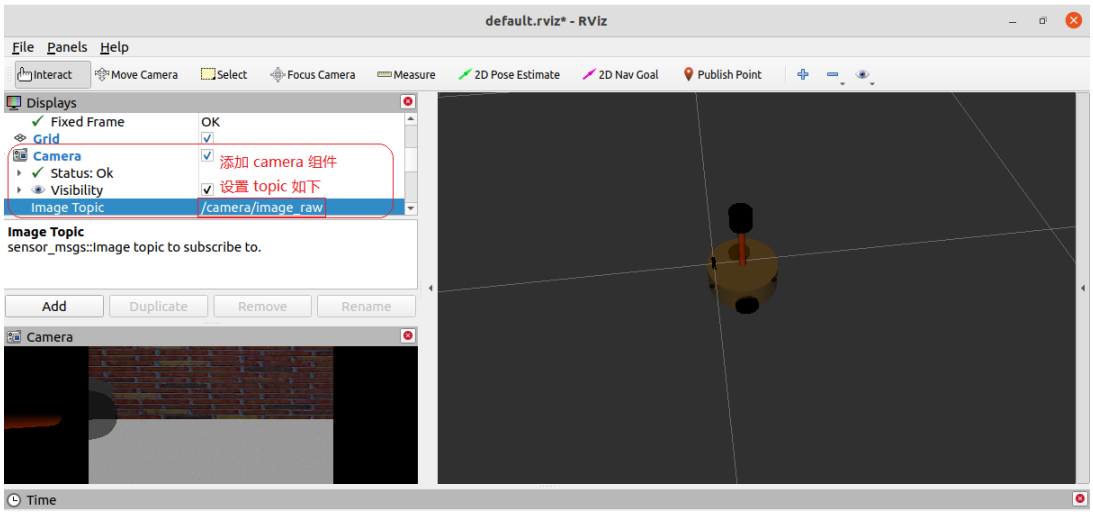
1.3 启动仿真环境

编写 launch 文件，启动 gazebo，此处略...

2.Rviz 显示摄像头数据

执行 gazebo 并启动 Rviz,在 Rviz 中添加摄像头组件。





6.7.4 kinect 信息仿真以及显示

通过 Gazebo 模拟 kinect 摄像头，并在 Rviz 中显示 kinect 摄像头数据。

实现流程:

kinect 摄像头仿真基本流程:

1. 已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加 kinect 摄像头配置；
2. 将此文件集成进 xacro 文件；

3. 启动 Gazebo，使用 Rviz 显示 kinect 摄像头信息。

1.Gazebo 仿真 Kinect

1.1 新建 Xacro 文件，配置 kinetic 传感器信息

```
<robot name="my_sensors" xmlns:xacro="http://wiki.ros.org/xacro">
    <gazebo reference="kinect link 名称">
      <sensor type="depth" name="camera">
        <always_on>true</always_on>
        <update_rate>20.0</update_rate>
        <camera>
          <horizontal_fov>${60.0*PI/180.0}</horizontal_fov>
          <image>
            <format>R8G8B8</format>
            <width>640</width>
            <height>480</height>
          </image>
          <clip>
            <near>0.05</near>
            <far>8.0</far>
          </clip>
        </camera>
        <plugin name="kinect_camera_controller"
filename="libgazebo_ros_openni_kinect.so">
          <cameraName>camera</cameraName>
          <alwaysOn>true</alwaysOn>
          <updateRate>10</updateRate>
          <imageTopicName>rgb/image_raw</imageTopicName>
          <depthImageTopicName>depth/image_raw</depthImageTopicName>
          <pointCloudTopicName>depth/points</pointCloudTopicName>
          <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>

<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTop
icName>
          <frameName>kinect link 名称</frameName>
          <baseline>0.1</baseline>
          <distortion_k1>0.0</distortion_k1>
          <distortion_k2>0.0</distortion_k2>
          <distortion_k3>0.0</distortion_k3>
          <distortion_t1>0.0</distortion_t1>
          <distortion_t2>0.0</distortion_t2>
          <pointCloudCutoff>0.4</pointCloudCutoff>
```

```
        </plugin>
      </sensor>
    </gazebo>

</robot>
Copy
```

1.2 xacro 文件集成

将步骤 1 的 Xacro 文件集成进总的机器人模型文件，代码示例如下:

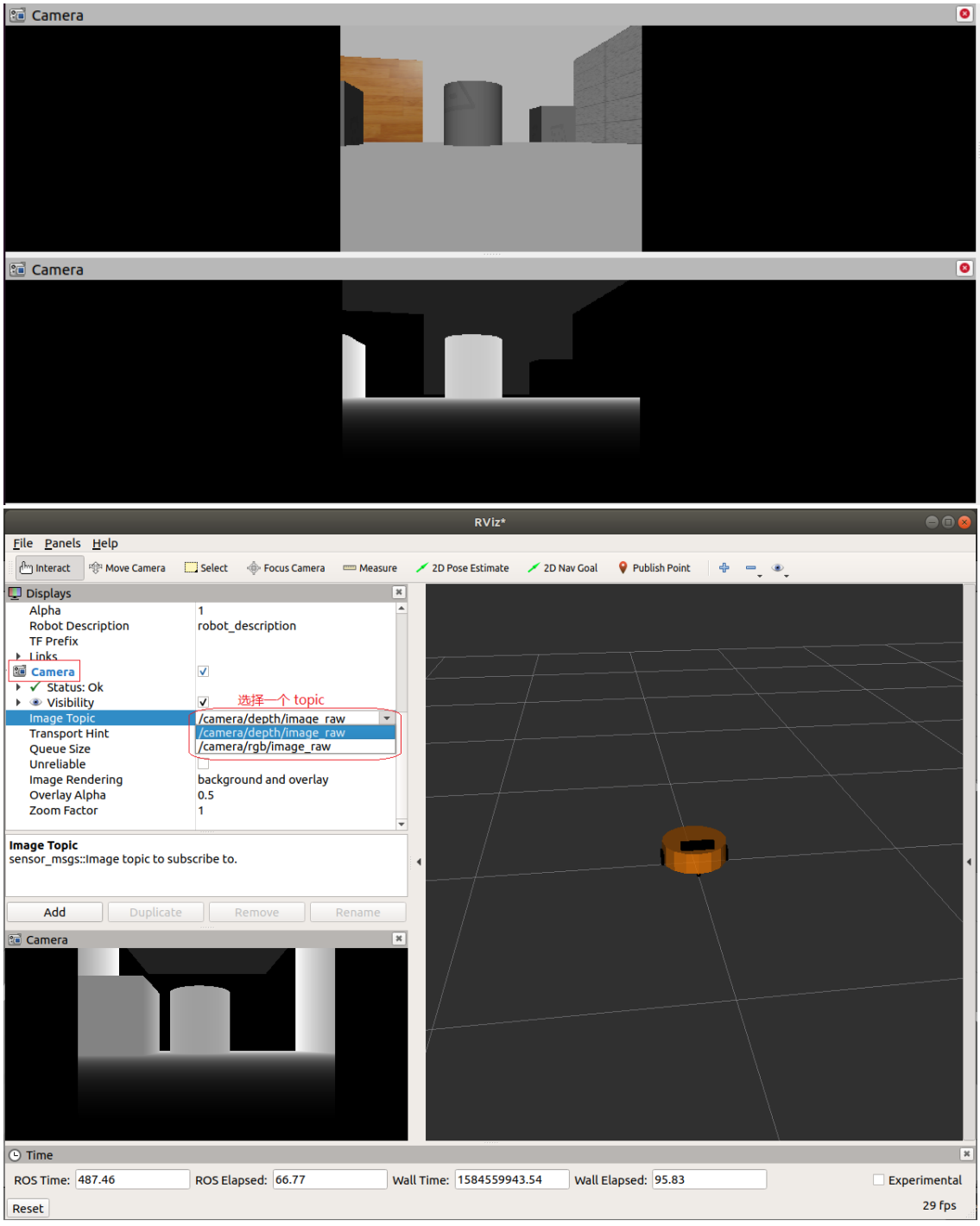
```
<!-- 组合小车底盘与传感器 -->
<robot name="my_car_camera" xmlns:xacro="http://wiki.ros.org/xacro">
    <xacro:include filename="my_head.urdf.xacro" />
    <xacro:include filename="my_base.urdf.xacro" />
    <xacro:include filename="my_camera.urdf.xacro" />
    <xacro:include filename="my_laser.urdf.xacro" />
    <xacro:include filename="move.urdf.xacro" />
    <!-- kinect 仿真的 xacro 文件 -->
    <xacro:include filename="my_sensors_kinect.urdf.xacro" />
</robot>
Copy
```

1.3 启动仿真环境

编写 launch 文件，启动 gazebo，此处略...

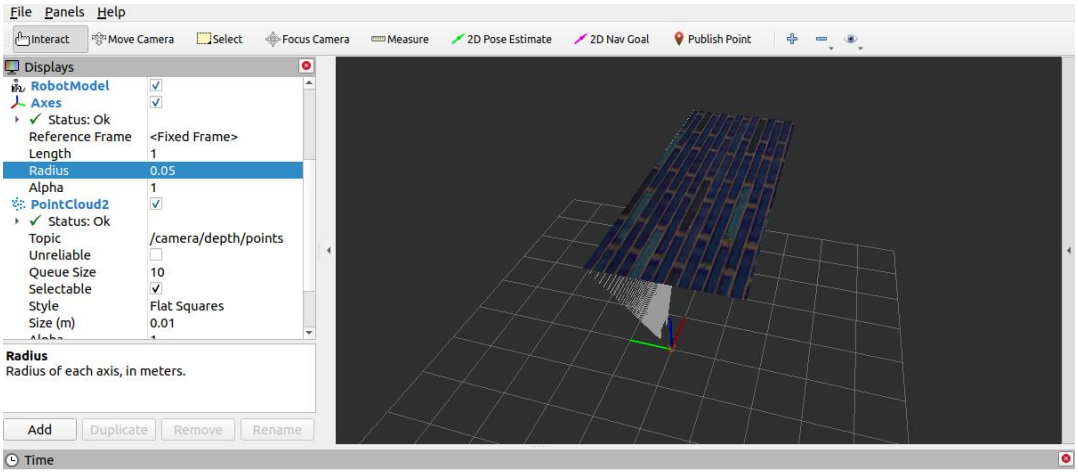
2 Rviz 显示 Kinect 数据

启动 rviz,添加摄像头组件查看数据



补充:kinect 点云数据显示

在 kinect 中也可以以点云的方式显示感知周围环境，在 rviz 中操作如下:



问题:在 rviz 中显示时错位。

原因:在 kinect 中图像数据与点云数据使用了两套坐标系统，且两套坐标系统位姿并不一致。

解决:

1.在插件中为 kinect 设置坐标系，修改配置文件的`<frameName>`标签内容：

```
<frameName>support_depth</frameName>
```

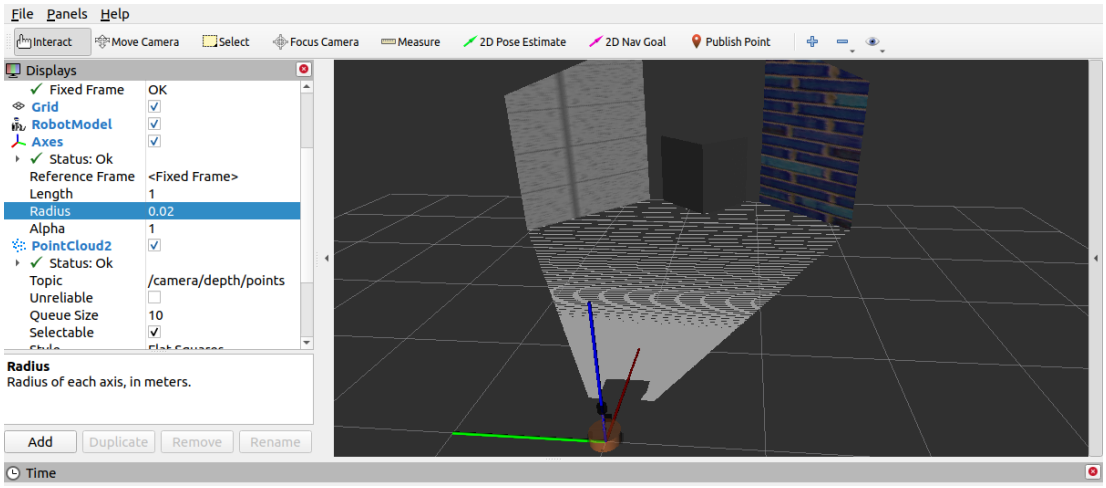
Copy

2.发布新设置的坐标系到 kinect 连杆的坐标变换关系，在启动 rviz 的 launch 中，添加:

```
<node pkg="tf2_ros" type="static_transform_publisher"
name="static_transform_publisher" args="0 0 0 -1.57 0 -1.57 /support
/support_depth" />
```

Copy

3.启动 rviz，重新显示。



6.8 本章小结

本章主要介绍了 ROS 中仿真实现涉及的三大知识点:

- URDF(Xacro)
- Rviz
- Gazebo

URDF 是用于描述机器人模型的 xml 文件，可以使用不同的标签具代表不同含义，URDF 编写机器人模型代码冗余，xacro 可以优化 URDF 实现，代码实现更为精简、高效、易读。容易混淆的是 Rviz 与 Gazebo，在此我们着重比较以下二者的区别:

rviz 是**三维可视化工具**，强调把已有的数据可视化显示；

gazebo 是**三维物理仿真平台**，强调的是创建一个虚拟的仿真环境。

rviz 需要**已有数据**。

rviz 提供了很多插件，这些插件可以显示图像、模型、路径等信息，但是前提都是这些数据已经以话题、参数的形式发布，rviz 做的事情就是订阅这些数据，并完成可视化的渲染，让开发者更容易理解数据的意义。

gazebo 不是显示工具，强调的是仿真，**它不需要数据，而是创造数据**。

我们可以在 gazebo 中免费创建一个机器人世界，不仅可以仿真机器人的运动功能，还可以仿真机器人的传感器数据。而这些数据就可以放到 rviz 中显示，所以使用 gazebo 的时候，经常也会和 rviz 配合使用。当我们手上没有机器人硬件或实验环境难以搭建时，仿真往往是非常有用的利器。

综上，如果你手上已经有机器人硬件平台，并且在上边可以完成需要的功能，用 rviz 应该就可以满足开发需求。

如果你手上没有机器人硬件，或者想在仿真环境中做一些算法、应用的测试，gazebo+rviz 应该是你需要的。

另外，rviz 配合其他功能包也可以建立一个简单的仿真环境，比如 rviz+ArbotiX。