**中国大学慕课**

**实训文档**

**ROS机器人开发技术**

**实训6.3 Xacro来整理URDF文件及搭建自己的URDF可视化模型**

|  |  |
| --- | --- |
| **姓名:** | **组号:** |
| **实训负责人:** | **日期:** |

# 背景知识

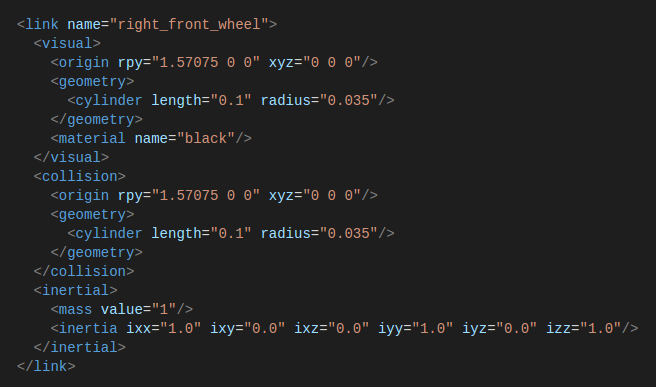
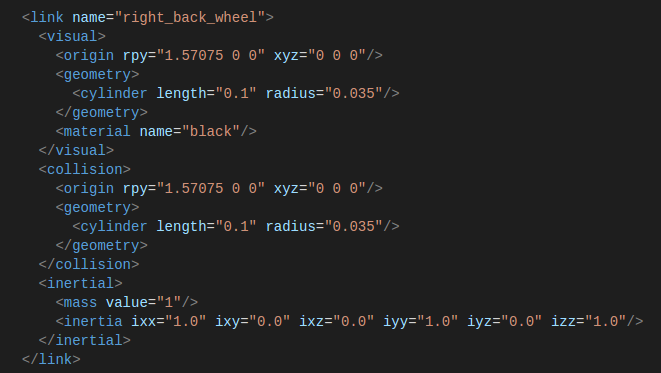
在机器人各种各样的模型里，我们需要构建自己的造型，我们在之前构造的机器人URDF文件的过程中，发现代码有很多重复的地方，因此，我们希望能够找到一个统一简便的方式来简化代码，通过使用宏命令来构建更精悍短小但又具有更高可读性的XML文件，这里，我们就需要了解使用Xacro来整理URDF文件。

# 子任务1 使用Xacro来整理URDF文件 (15min)

**问题：**我们为什么要使用Xacro来整理URDF文件？

答：Xacro（XML Macros）Xacro是一种XML宏语言。 使用xacro，可以通过使用宏命令构建更精悍短小但又具有更高可读性的XML文件，这种宏命令可以扩展达到更大的XML表达范围。它在处理大型XML文档（如机器人说明）时最为有用。

比如，我们之前右前轮和右后轮的urdf代码差不多，如下图：

****

使用xacro使URDF文件更为简洁：

**1、properties作为常量简化**

首先我们查看一下下图r2d2中的base\_link代码。Xacro允许我们指定properties作为常量，这样我们可以将下面左图的代码升级一下，将重复使用的变量定义为常量，这样不仅方便我们代码的维护且增强了代码的可读性。

****

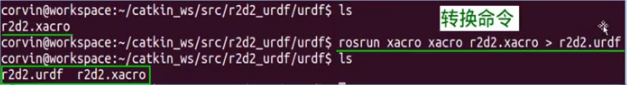
以后如果需要修改cylinder的length和radius属性的话，我们直接在最开头的部分将对于的width和bodylen修改其value值即可，在取值时需要使用${}，这样所有引用了width和bodylen的值都会相应的修改。使用xacro后最终生成的代码与urdf原始定义是相同的，只不过完成了简单的字符串替换。

在使用xacro的property时需要知道，name所对应的value值会直接替换到${}取值的地方，和C语言宏定义效果相同。

**问题：**xacro文件转换为urdf文件可以装换吗？

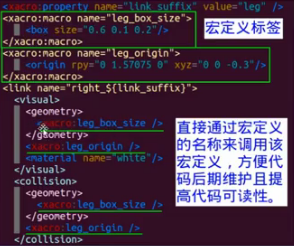
答：我们直接使用xacro工具来把xacro文件转换为urdf文件，在使用时,，均先会调用

rosrun xacro xacro.py xxx.urdf.xacro > xxx.urdf, 将其解析成对应的urdf文件，然后再使用，如图：

****

**2、通过macro来简化**

我们可以将雷同的代码通过macro来简化，即</xacro:macro>，如下图所示：

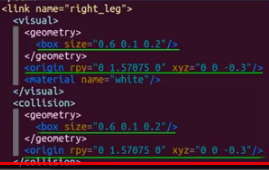
****

当我们使用<xacro:leg\_box\_size />来调用宏定义时，就是把对应的<box size=“0.6 0.1 0.2”/>直接替换过来。

我们可以通过命令行来直接转换，如下图：

**ijhk**

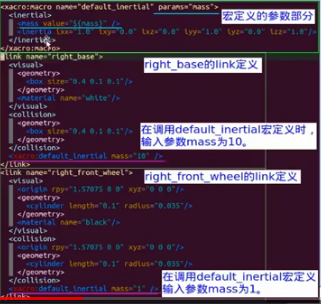
打开r2d2.urdf来查看宏定义替换结果：

****

我们发现转换后的urdf文件和我们预想的一致。

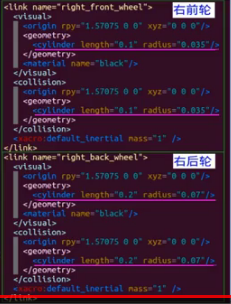
**3、通过Parameterized Macro来简化**

接下来介绍Parameterized Macro带有参数的宏定义，他比简单的宏定义只是完成简单的替换更高级一点，可以在替换时，输入参数使替换更为高级一点，如下图所示：

** **

**4、通过块整体作为参数来简化**

除了可以使用上面提到的简单输入参数，我们还可以使用整个块整体作为参数，这样的话，大大减少我们的代码量，首先我们需要查看原始代码中雷同的link定义，假设右前轮和右后轮只有cylinder不同，优化的代码如下图：

** **

**注意事项：**

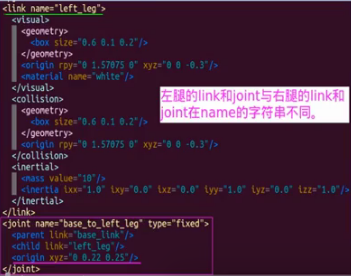
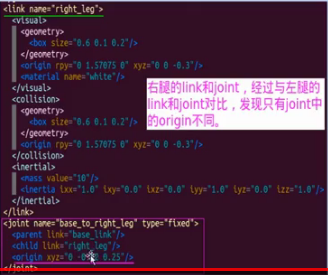
1、在指定块参数时，需要在参数名前加上\*

2、在调用块参数时，需要在xacro后加上insert\_block

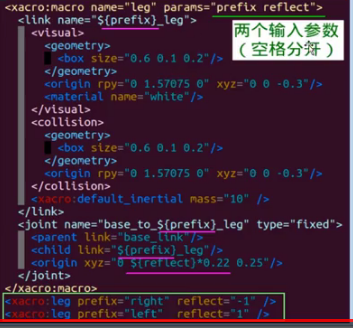
3、对于块参数在宏定义中可以多次调用

**实例：**

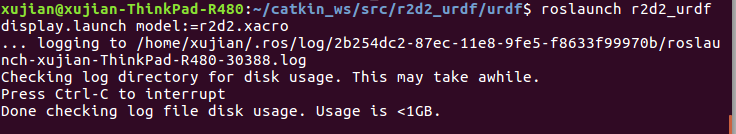
我们在创建link时经常会发现有许多的link模块比较雷同，只不过有些origin方向略微不同，这时我们就可以使用xacro的宏定义来将这种大块的代码直接定义为宏定义，这样就可以大幅度的减少代码量，我们来看r2d2的左右腿的link和joint定义，发现只有命令时的字符串不同和joint字段的origin不同：

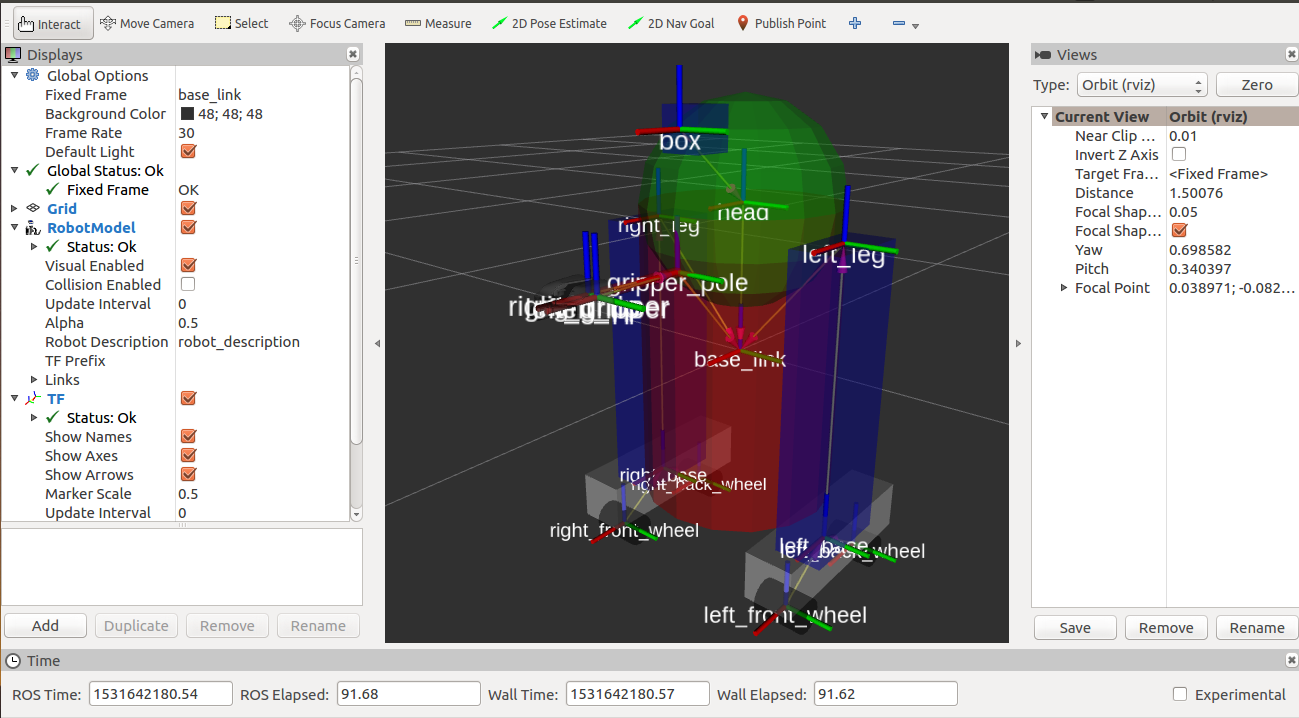
****

我们可以将上述的左右腿差不多的代码通过宏定义方式来简化代码结构并增加可维护性：

****

最后，我们在launch文件中加载xacro文件，可视化我们之前的那个机器人模型：





# 背景知识

在机器人模型中，我们需要构建自己的造型，往往需要从简单的模型开始，本次实训课，我们要尝试自己去构建一些基本造型，再去构建小车模型，最后在仿真环境中模拟小车模型，从而实现让我们的机器人模型在实验室环境中呈现。

# 子任务2：利用urdf中构建长方体和小车的模型， 并在rviz中可视化。(20min)。

**构建长方体模型：**

第一步：自己在工作空间下面创建一个新的package，命名为myurdf

第二步：在myurdf文件夹下面建立两个文件夹：一个是urdf文件夹，一个是launch文件夹

第三步：参考之前的一些实例代码写urdf文件里面的代码和launch文件里面的代码。urdf文件夹下的urdf文件命名为：myfirstrobot.urdf，dispaly文件夹下面的文件命名为display.launch。具体代码如下：

**myfirstrobot.urdf**

<?xml version="1.0"?>

<robot name="myfirstrobot">

<!-- Base Link -->

<link name="base\_link">

<visual>

<geometry>

<box size="0.1 0.1 2"/>

</geometry>

</visual>

</link>

</robot>

**display.launch**

<launch>

<param

name="robot\_description"

textfile="$(find myurdf)/robot/myfirstrobot.urdf" />

<!-- send fake joint values -->

<node

name="joint\_state\_publisher"

pkg="joint\_state\_publisher"

type="joint\_state\_publisher">

<param name="use\_gui" value="TRUE"/>

</node>

<!-- Combine joint values -->

<node

name="robot\_state\_publisher"

pkg="robot\_state\_publisher"

type="state\_publisher" />

<node

name="rviz"

pkg="rviz"

type="rviz"

respawn="false"

output="screen"

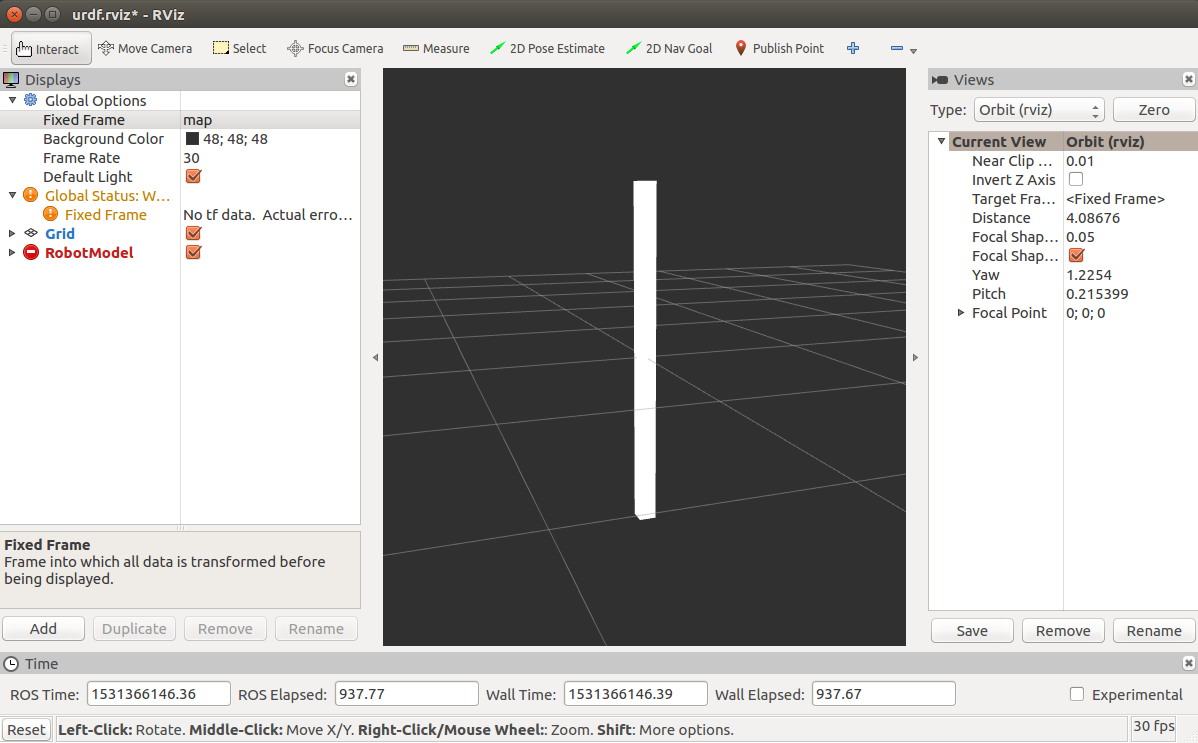
args="-d $(find myurdf)/urdf.rviz" />

</launch>

在终端运行命令：

roslaunch myurdf display.launch model:=urdf/myfirstrobot.urdf

在rviz里面添加robot即可，就可以看到下面的效果图：



**构建小车模型：**

第一步：自己在工作空间下面创建一个新的package，命名为sam\_load\_urdf\_into\_rviz

第二步：在sam\_load\_urdf\_into\_rviz文件夹下面建立两个文件夹：一个是urdf文件夹，一个是launch文件夹

第三步：参考之前的一些实例的代码写自己的urdf文件里面的代码和launch文件里面的代码。Urdf文件夹下的urdf文件命名为：robot1.urdf，dispaly文件夹下面的文件命名为display.launch。具体代码如下：

**Robot1.urdf**

<?xml version="1.0"?>

<robot name="Robot1">

<link name="base\_link">

<visual>

<geometry>

<box size="0.2 .3 .1"/>

</geometry>

<origin rpy="0 0 0" xyz="0 0 0.05"/>

<material name="white">

<color rgba="1 1 1 1"/>

</material>

</visual>

</link>

<link name="wheel\_1">

<visual>

<geometry>

<cylinder length="0.05" radius="0.05"/>

</geometry>

<origin rpy="0 1.5 0" xyz="0.1 0.1 0"/>

<material name="black">

<color rgba="0 0 0 1"/>

</material>

</visual>

</link>

<link name="wheel\_2">

<visual>

<geometry>

<cylinder length="0.05" radius="0.05"/>

</geometry>

<origin rpy="0 1.5 0" xyz="-0.1 0.1 0"/>

<material name="black"/>

</visual>

</link>

<link name="wheel\_3">

<visual>

<geometry>

<cylinder length="0.05" radius="0.05"/>

</geometry>

<origin rpy="0 1.5 0" xyz="0.1 -0.1 0"/>

<material name="black"/>

</visual>

</link>

<link name="wheel\_4">

<visual>

<geometry>

<cylinder length="0.05" radius="0.05"/>

</geometry>

<origin rpy="0 1.5 0" xyz="-0.1 -0.1 0"/>

<material name="black"/>

</visual>

</link>

<joint name="base\_to\_wheel1" type="fixed">

<parent link="base\_link"/>

<child link="wheel\_1"/>

<origin xyz="0 0 0"/>

</joint>

<joint name="base\_to\_wheel2" type="fixed">

<parent link="base\_link"/>

<child link="wheel\_2"/>

<origin xyz="0 0 0"/>

</joint>

<joint name="base\_to\_wheel3" type="fixed">

<parent link="base\_link"/>

<child link="wheel\_3"/>

<origin xyz="0 0 0"/>

</joint>

<joint name="base\_to\_wheel4" type="fixed">

<parent link="base\_link"/>

<child link="wheel\_4"/>

<origin xyz="0 0 0"/>

</joint>

</robot>

**Display.launch**

<launch>

<arg name="model" />

<arg name="gui" default="False" />

<param name="robot\_description" textfile="$(find sam\_load\_urdf\_into\_rviz)/urdf/robot1.urdf" />

<param name="use\_gui" value="$(arg gui)"/>

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher" ></node>

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="state\_publisher" />

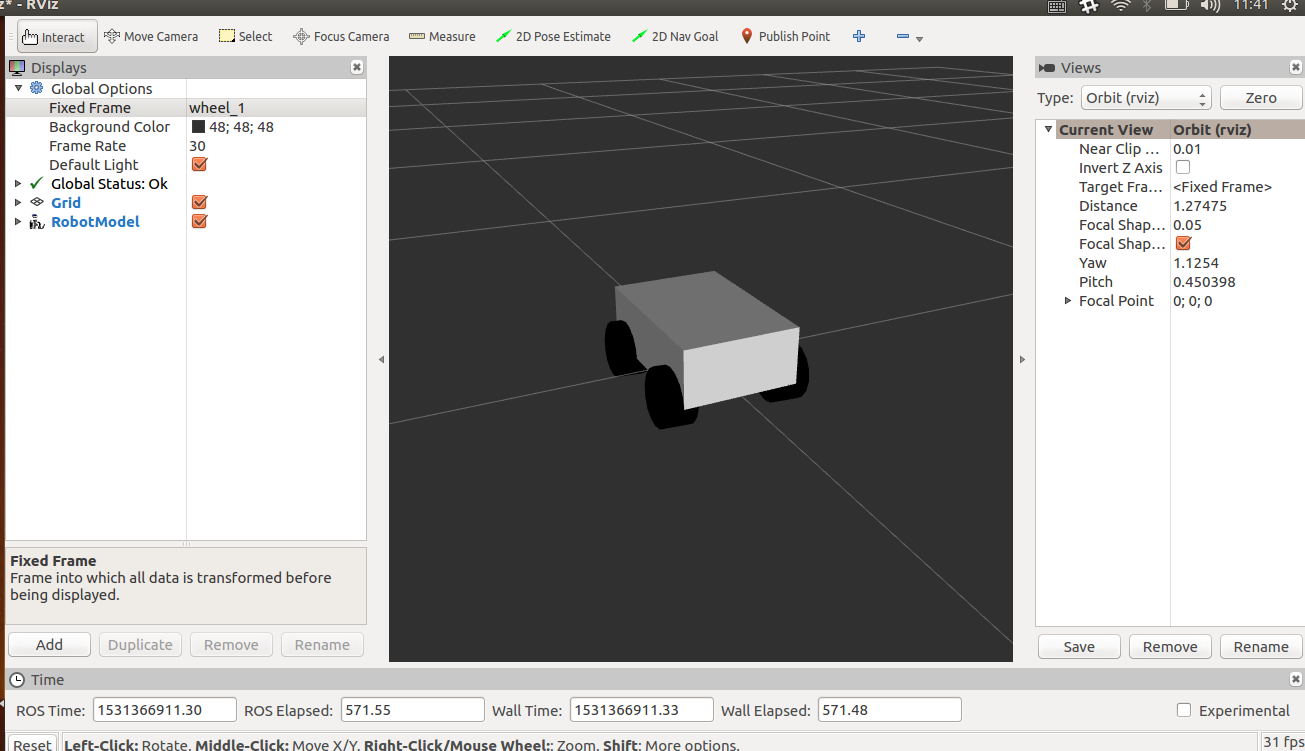
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find sam\_load\_urdf\_into\_rviz)/urdf.rviz" />

</launch>

输入：

roslaunch sam\_load\_urdf\_into\_rviz display.launch model:=urdf/robot1.urdf

可以得到下图：



# 背景知识

在生活中，我们的机器人往往是在一定场景下工作的，我们需要在实际的场景下仿真它的三维模型，这样就能够考察模型的运动属性和物理属性，因此，我们此次任务，是在rviz和gazebo中可视化我们的新的小车模型。

# 子任务3：在rviz和gazebo中呈现自己的模型。(10min)。

第一步：自己在工作空间下面创建一个新的package，命名为urdf\_demo

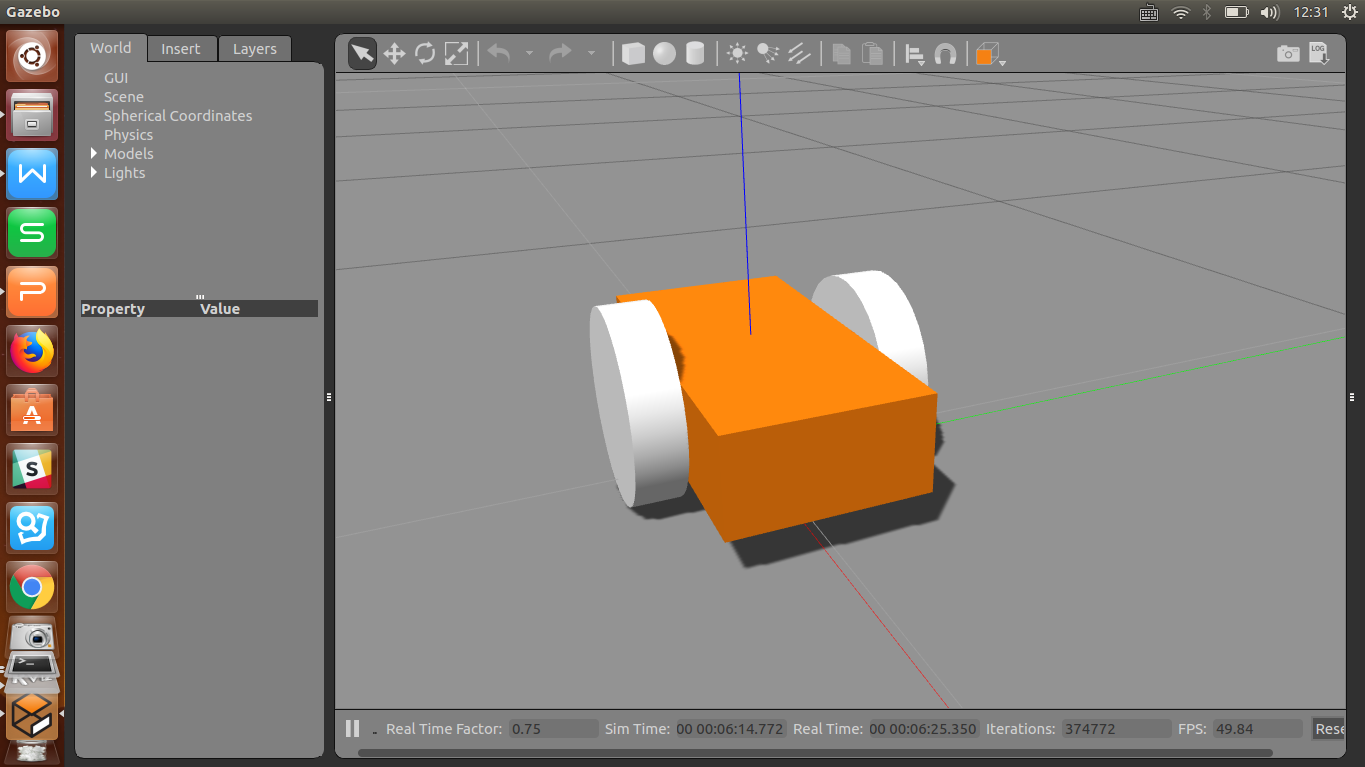
第二步：拷贝代码素材下的urdf\_demo里面的所有文件到该文件夹下面

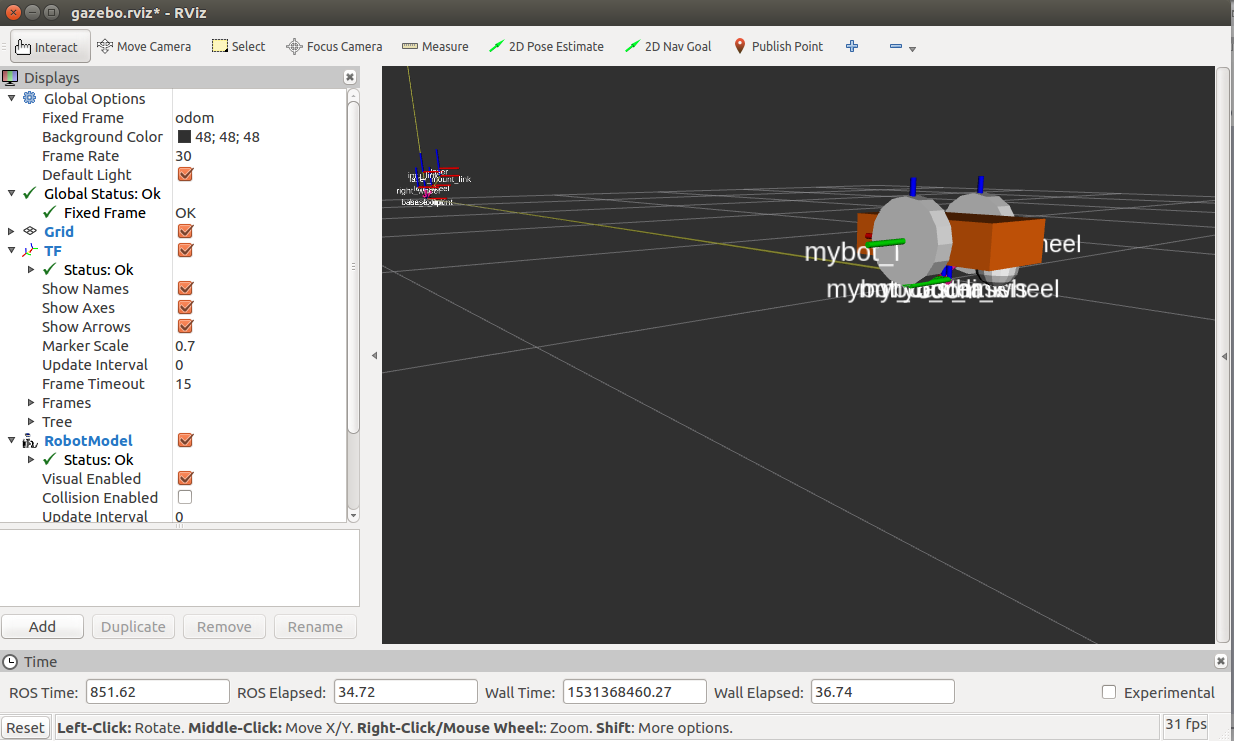
第三步：打开urdf\_demo下面的launch文件夹，这些launch文件都可以在仿真环境中运行。

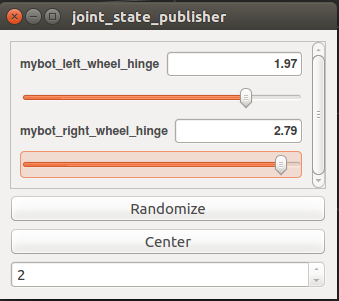
运行命令如下：

roslaunch urdf\_demo display\_gazebo\_rviz.launch

就可以看到模型呈现在实验室的场景中：







作业：思考display\_gazebo\_rviz.launch这个launch文件为什么能够在gazebo中可视化，和之前的launch文件有什么不同？

Launch文件相关代码注释：

宣告launch档，格式如：<launch> … </launch>

引数的语法会像这样：<arg name="…" value="…">

其中name是参数的名称。Value 是参数的值。有时候也用default=”…”来设定预设值。

用$(find <pkg>)这种语法来直接找包裹下的路径，所以不管这个包裹的路径被更改，程式照样能找得到目标。

<param>，用来定义一个设置在parameter server的参数，可以添加到中。参数用法如下：

Name=”namespace/name”参数名字。

Value=”value”（可选）定义参数的值，如果省略这个参数，则应该指定一个文件（binfile/textfile）或命令：command=” (find pkg-name)/exe’ $(find pkg-name)/arg.txt’” ，其中，exe是可执行文件，arg.txt参数文件。

呼叫节点：<node pkg="…" type="…" name="…" respawn=true ns="…" args=”….”/>

pkg：表示该节点所在的包。type：表示这个节点实际的名称，也就是开发的时候取的名字。name：虽然也是指该节点的名称，不过你可以再另外帮这个节点取名字，那么该节点便会把原名给覆盖掉，以这个名称表示。

respawn/required：是当该节点由于不明原因停止执行的时候，会自动重新启动。而required比较霸道一点，当该节点停止执行的时候，会让整个launch

档都停止执行、关闭。