# 实验三：机器人仿真

## 实验目的

* 机器人定义与组成
* ROS坐标变换
* 机器人系统建模
* 机器人建模与仿真

## 实验原理

## 1 TF简介

### 1.1 基本概念

机器人系统一般都包含很多坐标系，如世界坐标系，机体坐标系，机械手坐标系，头坐标系等。tf是一个让使用者可以跟踪所有这些坐标系，它采用树型数据结构，根据时间缓冲并维护各个参考系的坐标变换关系，可以在任意时间完成2个参考系的坐标变换。常用操作：

* 监听坐标变换
* 广播坐标变换

### 1.2 tf demo

详细介绍这个例子（两个turtle）,首先在终端输入下面的命令，安装相应的包。

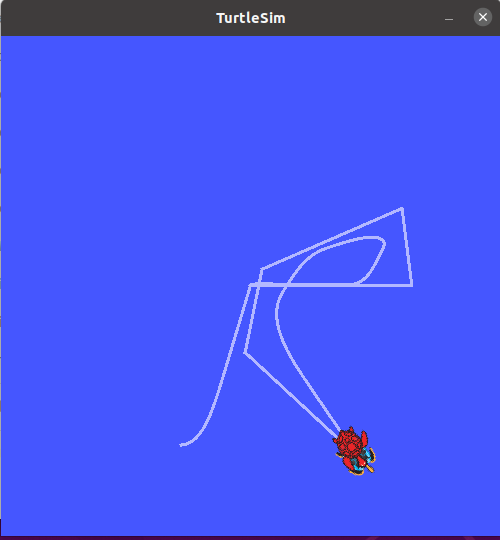
sudo apt-get install ros-noetic-ros-tutorials ros-noetic-geometry-tutorials

ros-noetic-rviz ros-noetic-rosbash ros-noetic-rqt-tf-tree

然后运行下面的命令

roslaunch turtle\_tf turtle\_tf\_demo.launch

移动键盘的方向键，会发现后一个乌龟跟着前一个乌龟运动。



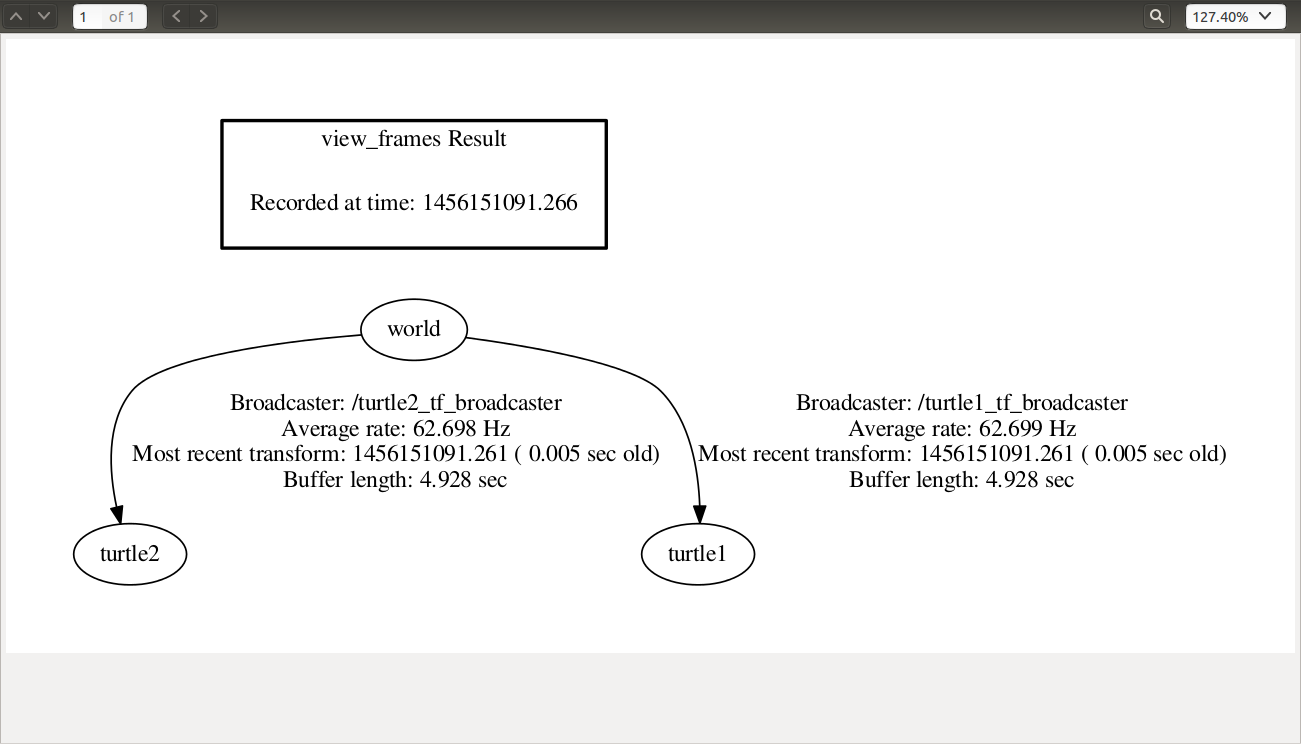
这个例子应用tf库建立了三个坐标系，world坐标系,turtle1坐标系,turtle2坐标系. 接下来介绍下tf的常用工具：

**view\_frames**

rosrun tf view\_frames

view\_frames创建一个在ROS上被tf广播的图表。它会监听在ROS上广播的坐标系，然后画一个这些坐标系怎么连接的树形图。

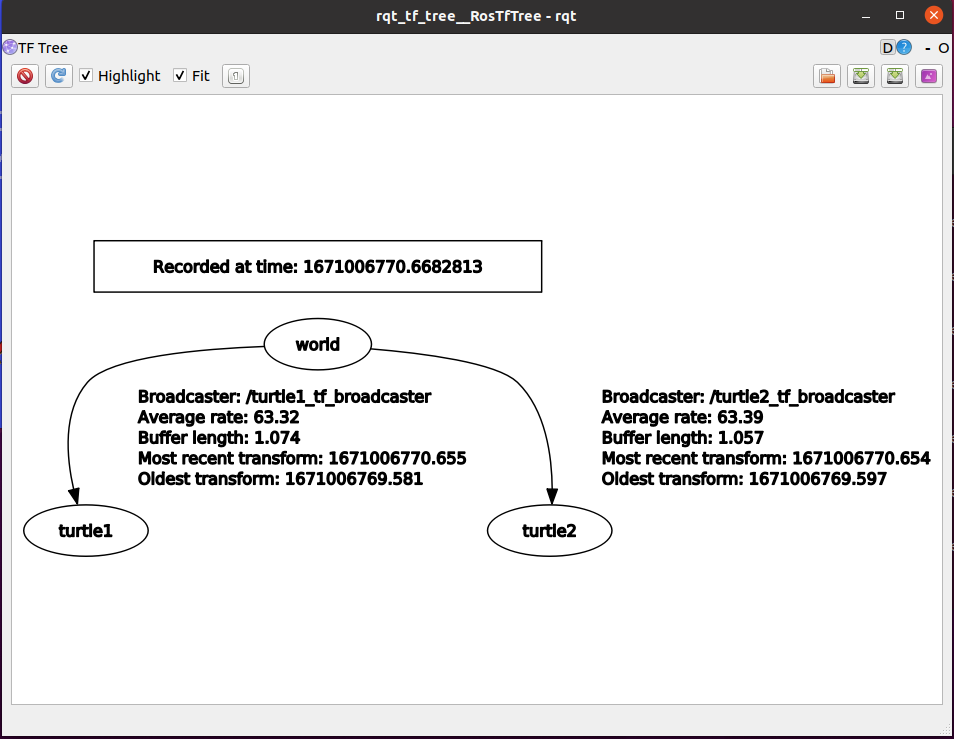
evince frames.pdf



**rqt\_tf\_tree**

查看在ROS上广播的坐标轴的树形图的图形化工具，可以手动刷新。

rosrun rqt\_tf\_tree rqt\_tf\_tree

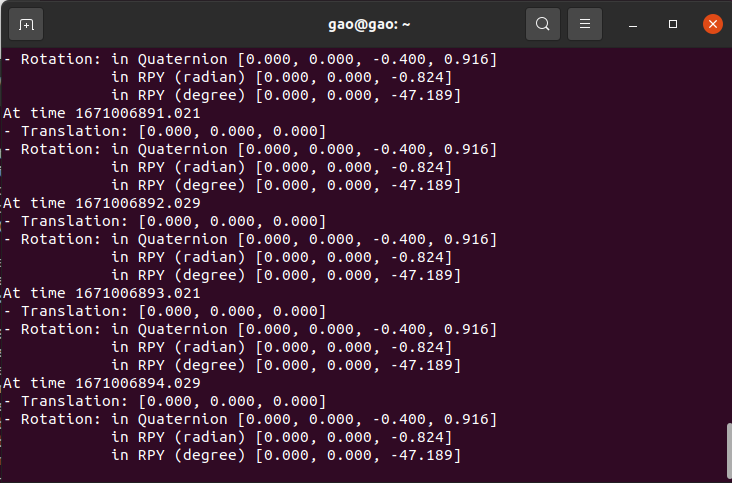


**tf\_echo**

该命令显示出两个坐标轴的转换关系。

rosrun tf tf\_echo [reference\_frame] [target\_frame]

rosrun tf tf\_echo turtle1 turtle2

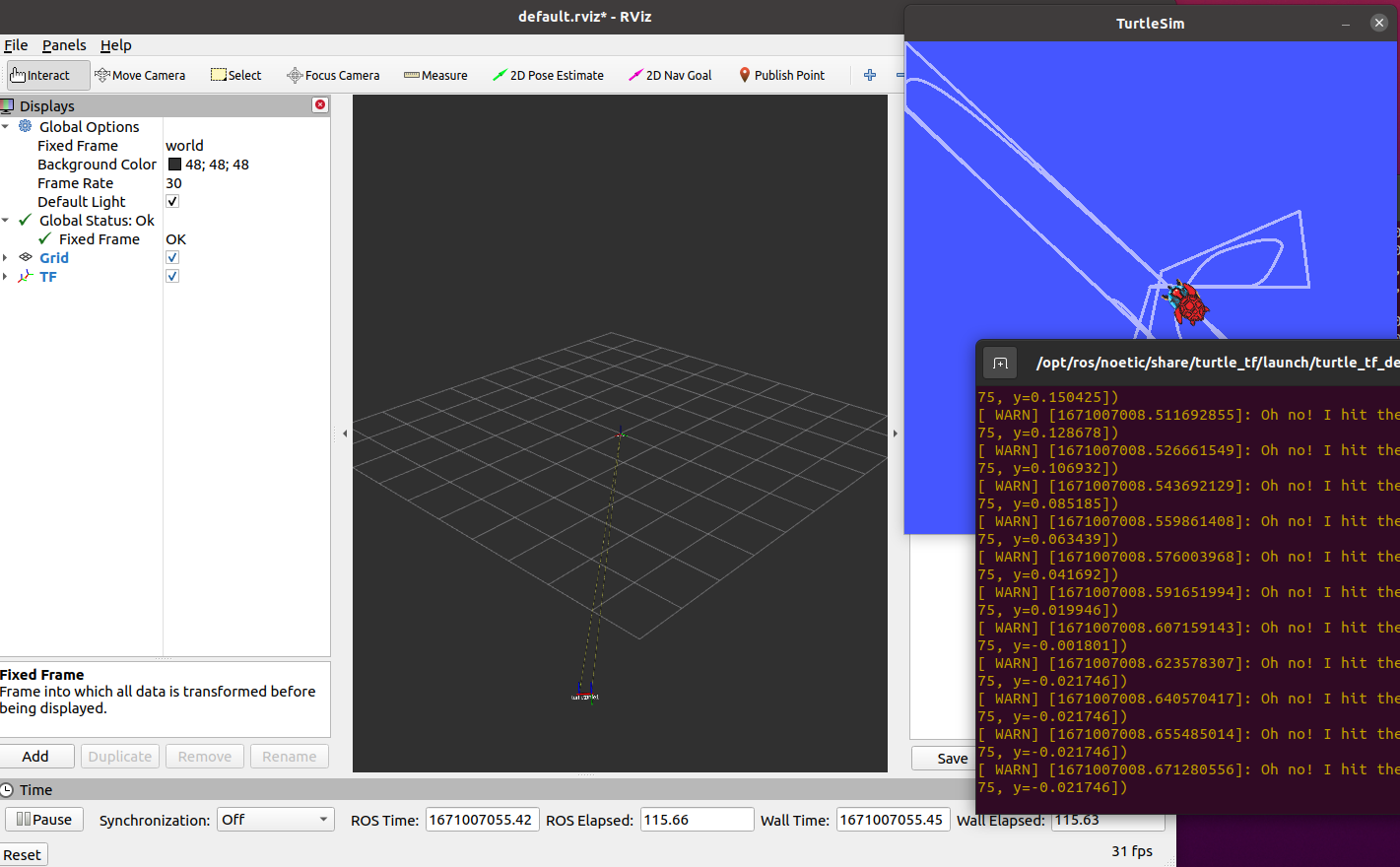


**rviz和tf**

roslaunch turtle\_tf turtle\_tf\_demo.launch

rosrun rviz rviz -d 'rospack find turtle\_tf' /rviz/turtle\_rviz.rviz

Fixed frame选择turtle2,添加TF显示类型，就可以看到



## 2 TF编程基础

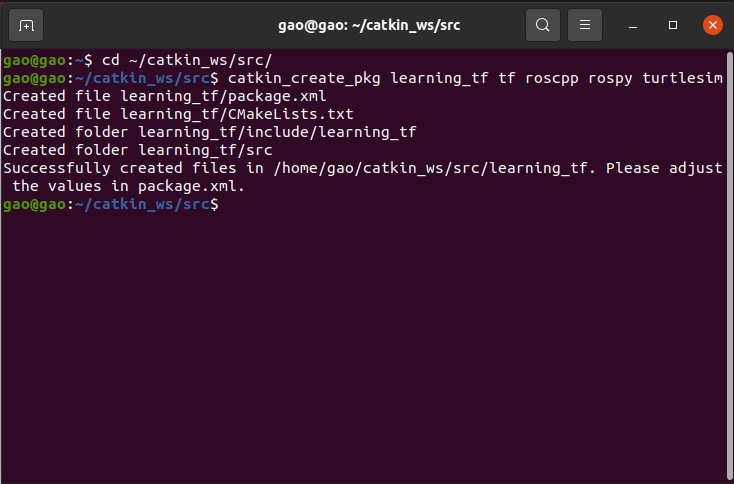
### 2.1 写一个广播tf的程序(C++)

输出下面命令:

cd ~/catkin\_ws/src

catkin\_create\_pkg learning\_tf geometry\_msgs tf roscpp rospy turtlesim

cd learning\_tf



在src文件夹家建立如下cpp文件，src/turtle\_tf\_broadcaster.cpp.

#include <ros/ros.h>

#include <tf/transform\_broadcaster.h>

#include <turtlesim/Pose.h>

std::string turtle\_name;

void poseCallback(const turtlesim::PoseConstPtr& msg){

static tf::TransformBroadcaster br;

tf::Transform transform;

transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );

tf::Quaternion q;

q.setRPY(0, 0, msg->theta);

transform.setRotation(q);

br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", turtle\_name));

}

int main(int argc, char\*\* argv){

ros::init(argc, argv, "my\_tf\_broadcaster");

if (argc != 2){ROS\_ERROR("need turtle name as argument"); return -1;};

turtle\_name = argv[1];

ros::NodeHandle node;

ros::Subscriber sub = node.subscribe(turtle\_name+"/pose", 10, &poseCallback);

ros::spin();

return 0;

}

**代码解释**

程序订阅了turtle\_name/pose话题，turtle\_name是程序运行的输入参数，有了前面的基础这块就不细说了，主要介绍回调函数的代码。

static tf::TransformBroadcaster br;

tf是一个命名空间，详细的文档介绍在如下网址： [TF命名空间](http://docs.ros.org/indigo/api/tf/html/c++/namespacetf.html)

TransformBroadcaster是tf命名空间下的一个类，这个类提供一个简单的方法去发布坐标转换信息，处理说有的发布和填充消息。 常用成员函数：

void sendTransform (const StampedTransform &transform)

StampedTransform类：tf的采样转换的数据类型。 常用成员函数：

StampedTransform (const tf::Transform &input, const ros::Time &timestamp,\

const std::string &frame\_id, const std::string &child\_frame\_id)

Quaternion类，常规的四元数类，表示刚体的旋转。

Transform类，支持刚体的平移和旋转变换。

tf::Transform transform;

transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );

tf::Quaternion q;

q.setRPY(0, 0, msg->theta);

transform.setRotation(q);

建立了一个tf::Transform实例，然后把turtle2D的pose信息转换成3D的信息。 > br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", turtle\_name));

第一个参数是transform实例，第二个是时间戳，第三个是父坐标系名，第四个是子坐标系名。

打开该package的CMakeLists.txt文件，添加下面命令代码

add\_executable(turtle\_tf\_broadcaster src/turtle\_tf\_broadcaster.cpp)

target\_link\_libraries(turtle\_tf\_broadcaster ${catkin\_LIBRARIES})

然后编译工作区

catkin\_make

编译成功后，我们在编写如下的launch文件

<launch>

<!-- Turtlesim Node-->

<node pkg="turtlesim" type="turtlesim\_node" name="sim"/>

<node pkg="turtlesim" type="turtle\_teleop\_key" name="teleop" output="screen"/>

<!-- Axes -->

<param name="scale\_linear" value="2" type="double"/>

<param name="scale\_angular" value="2" type="double"/>

<node pkg="learning\_tf" type="turtle\_tf\_broadcaster"

args="/turtle1" name="turtle1\_tf\_broadcaster" />

<node pkg="learning\_tf" type="turtle\_tf\_broadcaster"

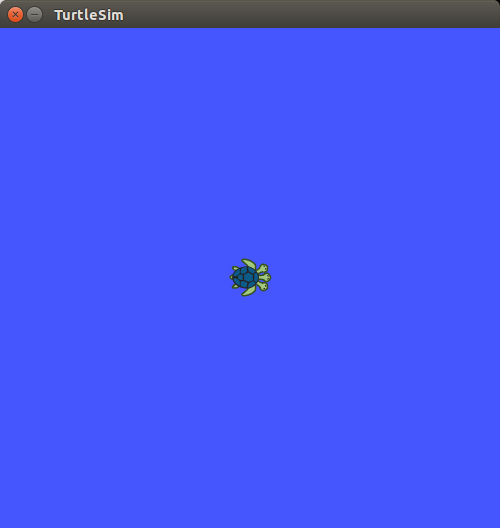
args="/turtle2" name="turtle2\_tf\_broadcaster" />

</launch>

然后运行该launch文件

roslaunch learning\_tf start\_demo.launch

如下图所示:



检验下结果

rosrun tf tf\_echo /world /turtle1

### 2.2 写一个tf 订阅者(C++)

切换到learning\_tf目录下

roscd learning\_tf

在src文件夹下建立src/turtle\_tf\_listener.cpp文件

#include <ros/ros.h>

#include <tf/transform\_listener.h>

#include <geometry\_msgs/Twist.h>

#include <turtlesim/Spawn.h>

int main(int argc, char\*\* argv){

ros::init(argc, argv, "my\_tf\_listener");

ros::NodeHandle node;

ros::service::waitForService("spawn");

ros::ServiceClient add\_turtle =

node.serviceClient<turtlesim::Spawn>("spawn");

turtlesim::Spawn srv;

add\_turtle.call(srv);

ros::Publisher turtle\_vel =

node.advertise<geometry\_msgs::Twist>("turtle2/cmd\_vel", 10);

tf::TransformListener listener;

ros::Rate rate(10.0);

while (node.ok()){

tf::StampedTransform transform;

try{

listener.lookupTransform("/turtle2", "/turtle1",

ros::Time(0), transform);

}

catch (tf::TransformException &ex) {

ROS\_ERROR("%s",ex.what());

ros::Duration(1.0).sleep();

continue;

}

geometry\_msgs::Twist vel\_msg;

vel\_msg.angular.z = 4.0 \* atan2(transform.getOrigin().y(),

transform.getOrigin().x());

vel\_msg.linear.x = 0.5 \* sqrt(pow(transform.getOrigin().x(), 2) +

pow(transform.getOrigin().y(), 2));

turtle\_vel.publish(vel\_msg);

rate.sleep();

}

return 0;

}

**代码解释**

tf::Transformer类，提供一个系统内的任意两个坐标系的转换。 tf::TransformListener类，继承tf::Transformer类，自动订阅ROS的transform消息。 常用成员函数：

void Transformer::lookupTransform(const std::string & target\_frame,const std::string & source\_frame,\

const ros::Time & time,StampedTransform & transform ) const

通过frameID得到两个坐标系的转换关系。

target\_frame 目标坐标系：需要将数据变换到的坐标系

source\_frame 起始坐标系：数据的起始坐标系

time 时间：变换所需的时间（0表示时间待定）

transform 变换：添加变换参考

tf::TransformListener listener;

try{

listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0), transform);

}

建立一个TransformListener 实例，一旦实例创建，它就自动接收tf的转换消息，并缓存10秒。然后查询一个指定的转换，就是查询从/turtle2到/turtle1的转换， ros::Time(0) 表示我们想要的是最新的一次变换，然后把变换存到transform变量。

geometry\_msgs::Twist vel\_msg;

vel\_msg.angular.z = 4.0 \*atan2(transform.getOrigin().y(),transform.getOrigin().x());

vel\_msg.linear.x = 0.5 \* sqrt\

(pow(transform.getOrigin().x(), 2) +pow(transform.getOrigin().y(), 2));

turtle\_vel.publish(vel\_msg);

根据上面检查的变换矩阵，把位置变成线速度，把角度变成角度度，然后发布turtle2/cmd\_vel话题，驱动turtle2走向turtle1.

然后在CMakeLists.txt中加入如下代码

add\_executable(turtle\_tf\_listener src/turtle\_tf\_listener.cpp)

target\_link\_libraries(turtle\_tf\_listener ${catkin\_LIBRARIES})

然后进行编译 catkin\_make 然后编写launch文件，代码如下

<launch>

<!-- Turtlesim Node-->

<node pkg="turtlesim" type="turtlesim\_node" name="sim"/>

<node pkg="turtlesim" type="turtle\_teleop\_key" name="teleop" output="screen"/>

<!-- Axes -->

<param name="scale\_linear" value="2" type="double"/>

<param name="scale\_angular" value="2" type="double"/>

<node pkg="learning\_tf" type="turtle\_tf\_broadcaster"

args="/turtle1" name="turtle1\_tf\_broadcaster" />

<node pkg="learning\_tf" type="turtle\_tf\_broadcaster"

args="/turtle2" name="turtle2\_tf\_broadcaster" />

<node pkg="learning\_tf" type="turtle\_tf\_listener"

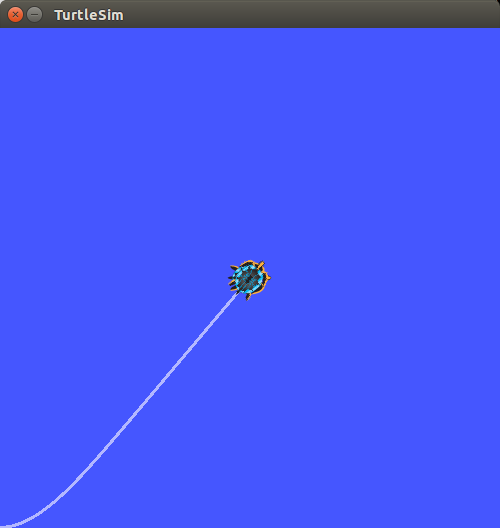
name="listener" />

</launch>

然后运行代码

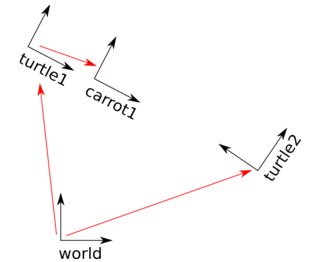
roslaunch learning\_tf start\_demo.launch

效果如下图所示:



### 2.3 参考坐标系(C++)

有些场景，局部坐标系会更利于我们的理解和调试，例如激光雷达的中心，TF允许我们为每个sensor，link建立一个局部坐标系，而且它会处理所有被引进的坐标系。 TF采用了树型结构，一个坐标系只能有一个父坐标系，但是它可以有很多子坐标系。 之前的TF树包含三个坐标系，world，turtle1，turtle2.world是turtle\*的父坐标系，下面我们将添加一个坐标系。



在src/frame\_tf\_broadcaster.cpp中添加如下代码:

#include <ros/ros.h>

#include <tf/transform\_broadcaster.h>

int main(int argc, char\*\* argv){

ros::init(argc, argv, "my\_tf\_broadcaster");

ros::NodeHandle node;

tf::TransformBroadcaster br;

tf::Transform transform;

ros::Rate rate(10.0);

while (node.ok()){

transform.setOrigin( tf::Vector3(0.0, 2.0, 0.0) );

transform.setRotation( tf::Quaternion(0, 0, 0, 1) );

br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "turtle1", "carrot1"));

rate.sleep();

}

return 0;

};

**代码解释**

transform.setOrigin( tf::Vector3(0.0, 2.0, 0.0) );

transform.setRotation( tf::Quaternion(0, 0, 0, 1) );

br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "turtle1", "carrot1"));

我们建立一个Transform对象，表示从父坐标系turtle1到子坐标系carrot1的变换，子坐标系carrot1在父坐标系的左边2m

**代码运行**

在CMakeLists.txt中添加如下代码 >

add\_executable(frame\_tf\_broadcaster src/frame\_tf\_broadcaster.cpp)

target\_link\_libraries(frame\_tf\_broadcaster ${catkin\_LIBRARIES})

在之前的turtle\_tf\_listener.cpp文件上做如下更改：

listener.lookupTransform("/turtle2", "/carrot1", ros::Time(0), transform);

然后

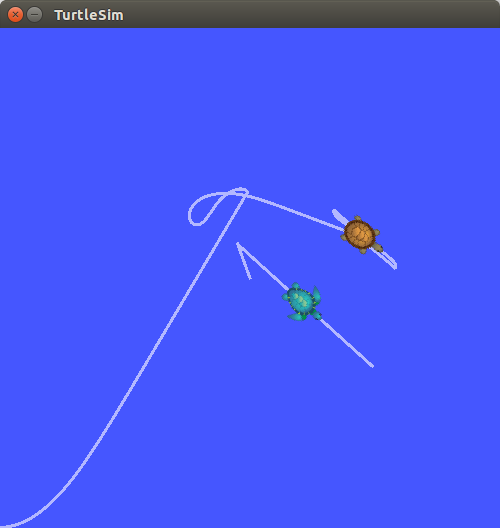
catkin\_make

在start\_demo.launch中 和中添加如下代码：

<node pkg="learning\_tf" type="frame\_tf\_broadcaster" name="broadcaster\_frame" />

然后运行

roslaunch learning\_tf start\_demo.launch



**广播一个移动坐标**

在源代码上做如下修改

transform.setOrigin( tf::Vector3(2.0\*sin(ros::Time::now().toSec()), 2.0\*cos(ros::Time::now().toSec()), 0.0) );

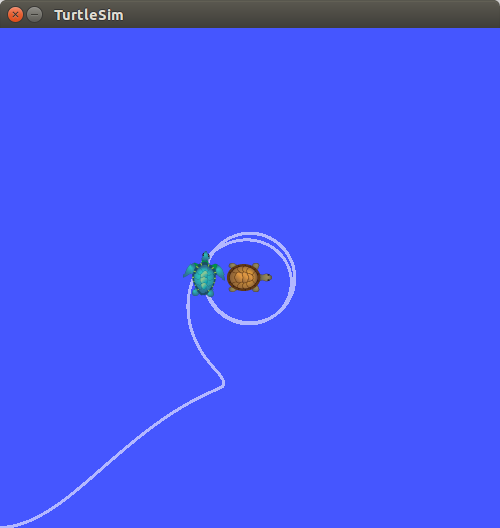
transform.setRotation( tf::Quaternion(0, 0, 0, 1) );

Carrot1的原点在以turtle1为圆心，半径2米的圆上随时间变化。 然后

catkin\_make

roslaunch learning\_tf start\_demo.launch

可以看到如下效果。



### 2.4 tf和时间(C++)

之前我们获得的都是最新的一次转换，但是不知道这个转换时什么时候记录的，现在我们将要了解如何得到一个指定时间的转换。代码如下

try{ ros::Time now = ros::Time::now();

listener.waitForTransform("/turtle2", "/turtle1", now, ros::Duration(3.0));

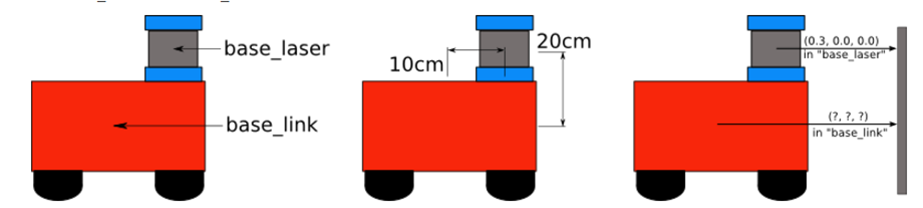
listener.lookupTransform("/turtle2", "/turtle1", now, transform);

waitForTransform()包含四个参数： 1.等待的target坐标系。2.源坐标系。3.指定时间。4.超时时间。 这个函数会阻塞，知道指定的转换到来或者是因为等待时间超时。 实验现象同上几乎一致。

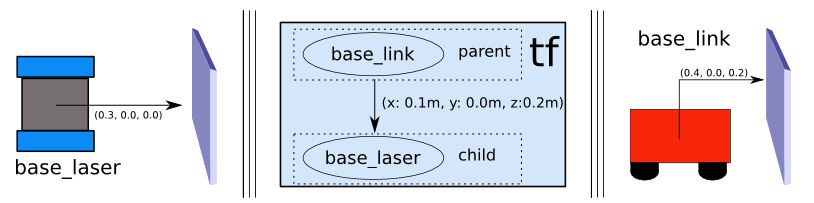
## 3 机器人tf实践

### 3.1 用tf配置机器人

很多ros包都要求用tf软件库发布的机器人的变换树。抽象层面，一个变换树定义了在不同坐标系间的平移和旋转。举例来说，一个机器人有一个移动底盘和一个安装在它上面的激光雷达。然后我们可以定义两个坐标系，一个在机器人底部的中心（base\_link)，一个在激光雷达的中心(base\_laser）。对于导航包来说，base\_link就是机器人的旋转中心。



现在我们有一些来自laser的距离数据，是在base\_laser坐标系下的，现在我们要把这个数据转换到base\_link下，帮助机器人避障。 假设我们知道激光在机器人底部中心的前方10cm上方20cm，没有旋转，如果我们在base\_laser下测得的距离是(0.3,0.0,0.0),那么在base\_link下距离为(0.4,0.0,0.2).



接下来我们将在ros下广播一个base\_laser->base\_link的变换，在learning\_tf功能包里，粘贴如下代码。

**广播一个变换**

#include <ros/ros.h>

#include <tf/transform\_broadcaster.h>

int main(int argc, char\*\* argv)

{ ros::init(argc, argv, "robot\_tf\_publisher");

ros::NodeHandle n;

ros::Rate r(100);

tf::TransformBroadcaster broadcaster;

while(n.ok())

{

broadcaster.sendTransform( tf::StampedTransform(

tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.2)),

ros::Time::now(),"base\_link", "base\_laser"));

r.sleep();

}

}

**编写代码：利用变换**

#include <ros/ros.h>

#include <geometry\_msgs/PointStamped.h>

#include <tf/transform\_listener.h>

void transformPoint(const tf::TransformListener& listener)

{

//we'll create a point in the base\_laser frame that we'd like

//to transform to the base\_link frame

geometry\_msgs::PointStamped laser\_point;

laser\_point.header.frame\_id = "base\_laser";

//we'll just use the most recent transform available for our simple example

laser\_point.header.stamp = ros::Time();

//just an arbitrary point in space

laser\_point.point.x = 1.0;

laser\_point.point.y = 0.2;

laser\_point.point.z = 0.0;

try

{

geometry\_msgs::PointStamped base\_point;

listener.transformPoint("base\_link", laser\_point, base\_point);

ROS\_INFO("base\_laser: (%.2f, %.2f. %.2f) -----> base\_link: (%.2f, %.2f, %.2f)

at time %.2f", laser\_point.point.x, laser\_point.point.y, laser\_point.point.z,

base\_point.point.x, base\_point.point.y, base\_point.point.z,

base\_point.header.stamp.toSec());

}

catch(tf::TransformException& ex){

ROS\_ERROR("Received an exception trying to transform

a point from \"base\_laser\" to \"base\_link\": %s", ex.what());

}

}

int main(int argc, char\*\* argv)

{

ros::init(argc, argv, "robot\_tf\_listener");

ros::NodeHandle n;

tf::TransformListener listener(ros::Duration(10));

//we'll transform a point once every second

ros::Timer timer = n.createTimer(ros::Duration(1.0),

boost::bind(&transformPoint, boost::ref(listener))); ros::spin();

}

**代码解释**

//we'll create a point in the base\_laser frame that we'd like to

//transform to the base\_link frame geometry\_msgs::PointStamped laser\_point;

laser\_point.header.frame\_id = "base\_laser";

//we'll just use the most recent transform available for our simple example

laser\_point.header.stamp = ros::Time();

//just an arbitrary point in space

laser\_point.point.x = 1.0;

laser\_point.point.y = 0.2;

laser\_point.point.z = 0.0;

我们手动生成了一个点的激光数据，(1.0,0.2,0.0)

try{

geometry\_msgs::PointStamped base\_point;

listener.transformPoint("base\_link", laser\_point, base\_point);

ROS\_INFO("base\_laser: (%.2f, %.2f. %.2f) -----> base\_link: (%.2f, %.2f, %.2f)

at time %.2f", laser\_point.point.x, laser\_point.point.y, laser\_point.point.z,

base\_point.point.x, base\_point.point.y, base\_point.point.z,

base\_point.header.stamp.toSec());

}

把base\_laser的数据转到base\_link下 在CMakeLists.txt文件中输入

add\_executable(tf\_broadcaster src/tf\_broadcaster.cpp)

add\_executable(tf\_listener src/tf\_listener.cpp)

target\_link\_libraries(tf\_broadcaster ${catkin\_LIBRARIES})

target\_link\_libraries(tf\_listener ${catkin\_LIBRARIES})

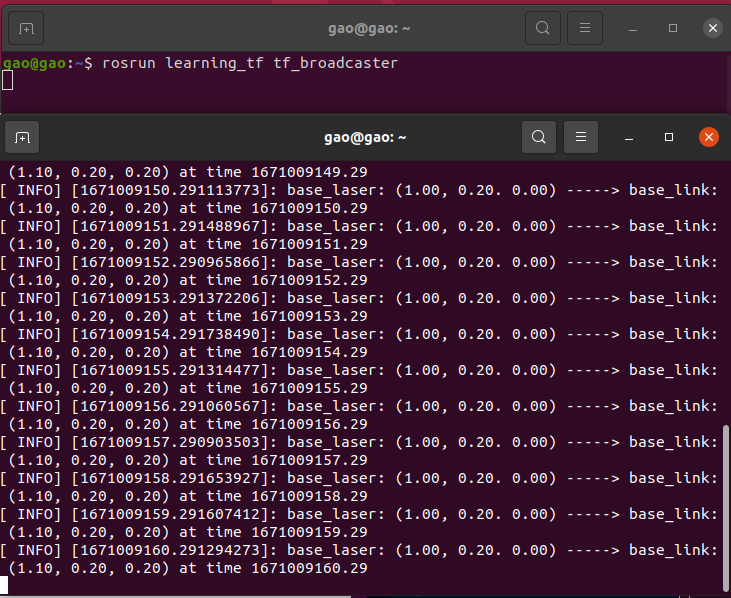
编译这个包

catkin\_make

**运行代码**

rosrun learning\_tf tf\_broadcaster

rosrun learning\_tf tf\_listener



### 3.2 发布机器人状态

当你的机器人有很多相关坐标系时，把他们都发布到tf会比较繁琐，但是机器人状态发布器（robot state publisher）可以帮我们完成这个工作。机器人状态发布器内部有一个机器人的运动学模型，因此给定机器人的关节位置，它能计算出连杆的3D姿势(pose)我们可以把它当做一个节点(推荐方法）也可以当做一个库。当做一个节点运行必备：

* 一个参数服务器上的urdf xml机器人描述
* 一个发布sensor\_msgs/JointState关节位置的源

小结：通过该节课的学习，我们掌握了TF的基本概念，掌握了tf的常用工具，通过c++编写了tf的常用操作的程序，如广播和监听等，更通过实例加深了对tf的理解。

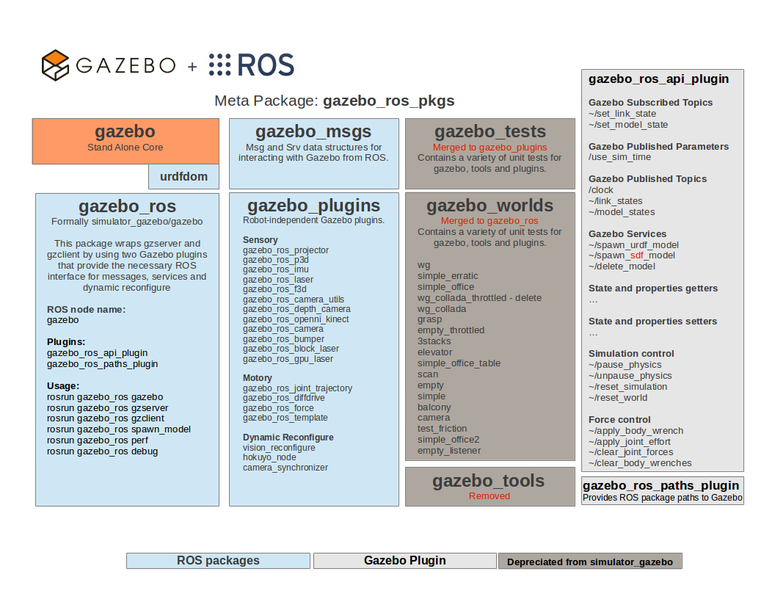
## 4 Gazebo

### 4.1 了解Gazebo

Gazebo是一个三维动态仿真器，能准确高效地仿真在复杂的室内外环境下机器人群体。与游戏引擎类似，Gazego能对一整套传感器进行高度逼真的物理仿真、为程序和用户提供交互接口，其典型应用场景包括：

* 测试机器人算法
* 机器人设计
* 现实情景下的回溯测试

在ROS系统中，Gazebo主要用来测试机器人算法。 Gazebo本是独立发布运行的程序，将其用于ROS系统两者须整合。在ROS系统的hydro版本之前，gazebo以simulator\_gazebo功能包集形式移植到ROS系统上；之后的gazebo不在依赖ROS系统而独立安装于Ubuntu系统，通过gazebo\_ros\_pkgs元功能包封装Gazebo接口。此功能包提供了使用ROS message、service和dynamic reconfigure在Gazebo中进行机器人仿真的必要接口。Gazebo\_ros\_pkgs功能包的接口概况如下：



gazebo\_ros主要用于gazebo接口封装、gazebo服务端和客户端的启动、URDF模型生成等；gazebo\_msgs是gazebo的Msg和Srv数据结构；gazebo\_plugins用于gazebo的通用传感器插件。此元功能包对Gazebo的封装通过gazebo\_ros\_api\_plugin和gazebo\_ros\_path\_plugin这两个Gazebo的插件实现，两个插件的功能见图中的淡灰色部分。

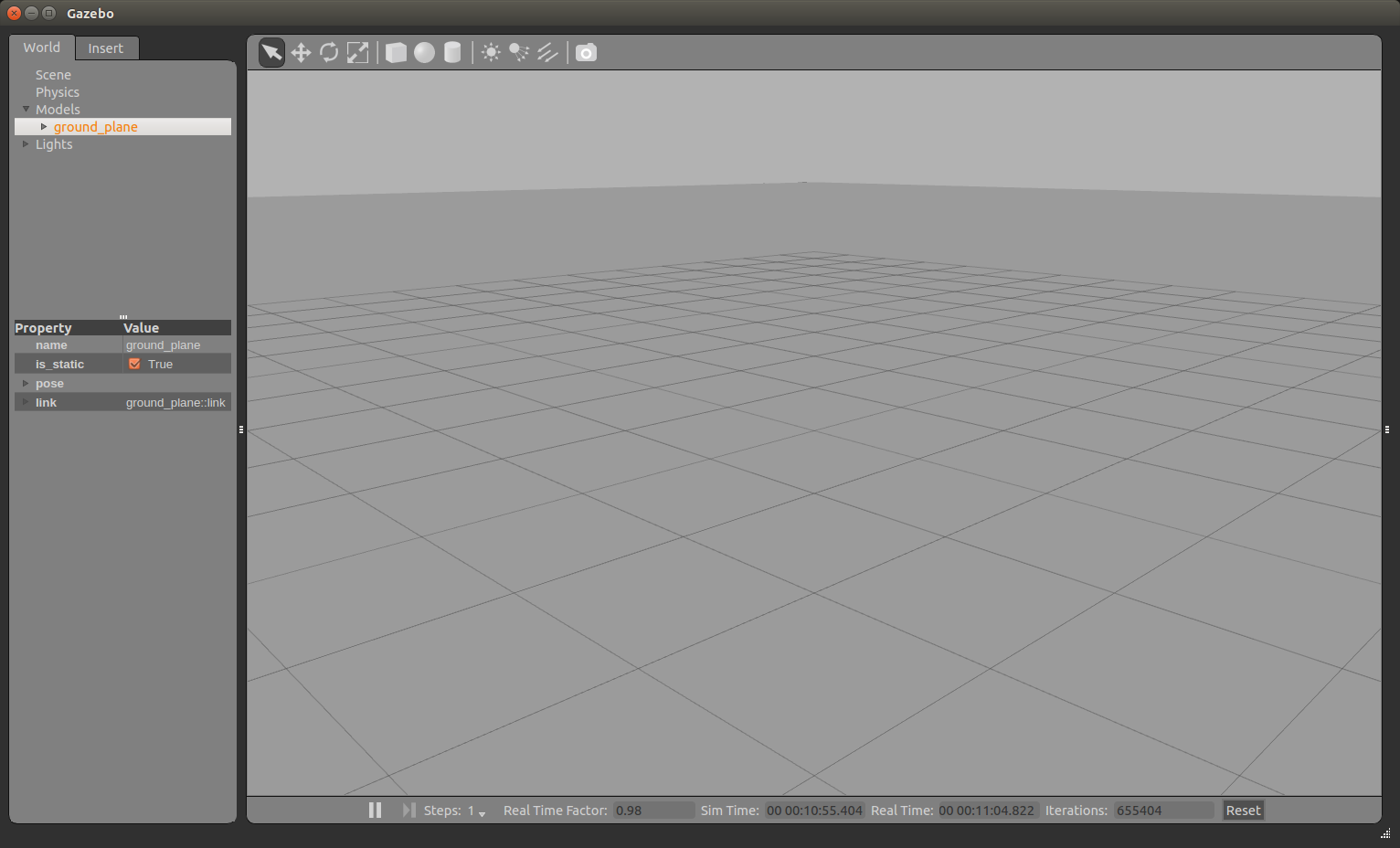
### 4.2 使用Gazebo

启动Gazebo并在仿真环境下打开世界模型和生成机器人模型有多种方式，下面我们讨论Gazebo在ROS系统中的启动方式。

#### 4.2.1 启动Gazebo

上面已经提到Gazebo\_ros\_pkgs封装了Gazebo，在ROS系统中启动Gazebo通过gazebo\_ros功能包完成。用roslaunch工具启动Gazebo并加载一个空的世界模型，命令如下：

roslaunch gazebo\_ros empty\_world.launch

 此功能包可设置的映射参数有：

* paused 以暂停状态启动Gazebo，默认为*false*。
* usesimtime ROS的node是否使用Gazebo通过/clock topic发布的仿真时间，默认为*true*。
* gui 启用Gazebo的用户交互接口来控制视图，默认为*true*。
* headless 禁用仿真渲染组件的功能，当*gui:=true*时，不能使能此参数，默认为*false*。 -debug 在gdb中启动gzserver(Gazebo Server)以用于调试，默认为*false*。

如：

roslaunch gazebo\_ros empty\_world.launch paused:=true use\_sim\_time:=false gui:=true headless:=false debug:=true

当然如果我们使用roslaunch工具而非rosrun启动Gazebo，一般映射参数可以通过roslaunch文件指定。 功能包gazebo\_ros还提供了其它的示例，使用以下命令执行：

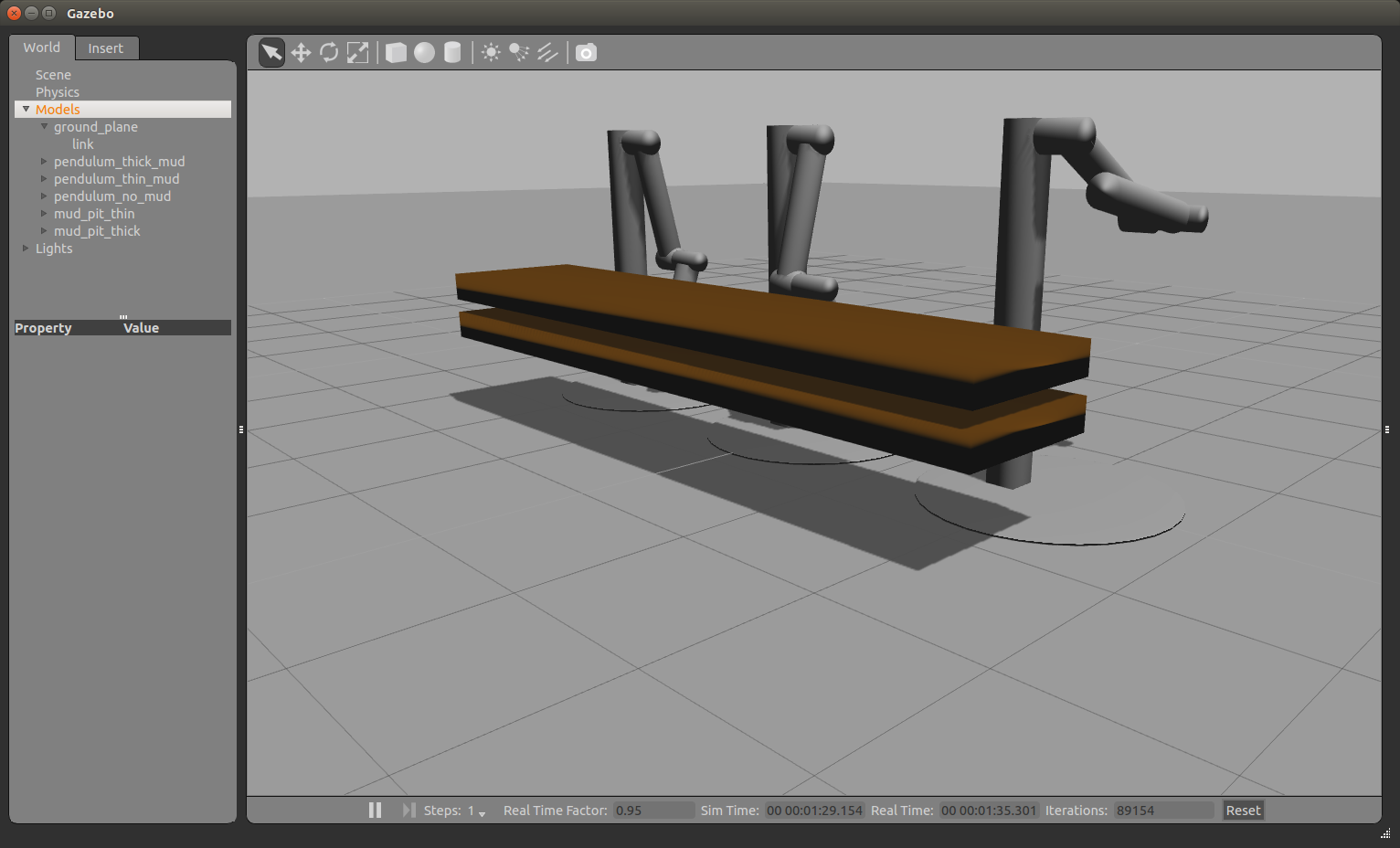
roslaunch gazebo\_ros willowgarage\_world.launch

roslaunch gazebo\_ros mud\_world.launch

roslaunch gazebo\_ros shapes\_world.launch

roslaunch gazebo\_ros rubble\_world.launch

如下图为roslaunch gazebo\_ros mud\_world.launch的执行效果（首次运行需要等待一段时间自动下载模型）：



下面就以roslaunch gazebo\_ros mud\_world.launch为例分析Gazebo的运行过程。先看文件mud\_world.launch文件：

<launch>

<!-- We resume the logic in empty\_world.launch,

changing only the name of the world to be launched -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="world\_name" value="worlds/mud.world"/>

<!-- Note: the world\_name is with respect to

GAZEBO\_RESOURCE\_PATH environmental variable -->

<arg name="paused" value="false"/>

<arg name="use\_sim\_time" value="true"/>

<arg name="gui" value="true"/>

<arg name="headless" value="false"/>

<arg name="debug" value="false"/>

</include>

</launch>

此文件内容包含了empty\_world.launch文件，其它均为映射参数的赋值，其中的*world\_name*以相对路径指定了world模型的定义文件，绝对路径由GAZEBO\_RESOURCE\_PATH这个环境变量决定。默认为/usr/share/gazebo-x.x，对kinetic版本来说为/usr/share/gazebo-7.0.0。

再看看empty\_world.launch：

<launch>

<!-- these are the arguments you can pass this launch file,

for example paused:=true -->

<arg name="paused" default="false"/>

<arg name="use\_sim\_time" default="true"/>

<arg name="extra\_gazebo\_args" default=""/>

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

<arg name="headless" default="false"/>

<arg name="debug" default="false"/>

<arg name="physics" default="ode"/>

<arg name="verbose" default="false"/>

<arg name="world\_name" default="worlds/empty.world"/>

<!-- Note: the world\_name is with respect

to GAZEBO\_RESOURCE\_PATH environmental variable -->

<!-- set use\_sim\_time flag -->

<group if="$(arg use\_sim\_time)">

<param name="/use\_sim\_time" value="true" />

</group>

<!-- set command arguments -->

<arg unless="$(arg paused)" name="command\_arg1" value=""/>

<arg if="$(arg paused)" name="command\_arg1" value="-u"/>

<arg unless="$(arg headless)" name="command\_arg2" value=""/>

<arg if="$(arg headless)" name="command\_arg2" value="-r"/>

<arg unless="$(arg verbose)" name="command\_arg3" value=""/>

<arg if="$(arg verbose)" name="command\_arg3" value="--verbose"/>

<arg unless="$(arg debug)" name="script\_type" value="gzserver"/>

<arg if="$(arg debug)" name="script\_type" value="debug"/>

<!-- start gazebo server-->

<node name="gazebo" pkg="gazebo\_ros" type="$(arg script\_type)" respawn="false"

output="screen" args="$(arg command\_arg1) $(arg command\_arg2) $(arg command\_arg3)

-e $(arg physics) $(arg extra\_gazebo\_args) $(arg world\_name)" />

<!-- start gazebo client -->

<group if="$(arg gui)">

<node name="gazebo\_gui" pkg="gazebo\_ros" type="gzclient"

respawn="false" output="screen"/>

</group>

</launch>

此文件通过映射参数设置Gazebo运行的相关参数，最后通过*gzserver*（*debug*）和*gzclient*两个脚本分别运行Gazebo的*gzserver*和*gzclient*,实现Gazebo的正常工作。 我们再看看world文件，了解世界模型是如何定义的，以mud.world为例：

<sdf version="1.4">

<world name="default">

<include>

<uri>model://sun</uri>

</include>

<include>

<uri>model://ground\_plane</uri>

</include>

<include>

<uri>model://double\_pendulum\_with\_base</uri>

<name>pendulum\_thick\_mud</name>

<pose>-2.0 0 0 0 0 0</pose>

</include>

...

</world>

</sdf>

以上为截取的引用三个模型的内容，gzserver运行时会在本地模型数据库中查找这个三模块，默认本地模型数据库路径为~/.gazebo/models；如果找不到则会连接Gazebo在线模型库查找并下载到本地。因为众所周知的原因，Gazebo网站连接并不稳定，所以时常出现Gazebo仿真场景显示慢、不显示甚至程序出错的问题，建议如果用到Gazebo官网提供的模型可提前下载到本地。

### 4.3 机器人模型生成

Gazebo使用SDF（Simulation Description Format）仿真描述格式定义仿真环境和模型。而ROS使用URDF（Universal Robotic Description Format）通用机器人描述格式来定义机器人模型，且此格式定义的文件不能直接用于Gazebo，使用中需加以转换，添加用于描述仿真定义的标签。

在Gazebo中引入以URDF定义的机器人模型有两种方式：

**ROS服务调用方式：**这种方法将模型定义文件做成ROS功能包存放，通过python脚本调用ROS service生成Gazebo识别的机器人模型文件，使得功能包更易在其它机器上使用。

**模型库方式：**这种方法通过创建模型库，然后通过向修改环境变量将自建模型库添加到Gazebo模型库中，以更直接将机器人模型定义文件包含到world文件中。

一般我们使用第一种方式，这也是官网推荐的方式。下面详细说明下这两种方式。

#### 4.3.1 ROS服务调用方式

这种方式的具体过程就是用gazebo\_ros功能包下名为\*spawn\_model\*的python脚本通过封装接口向Gazebo发起服务调用，请求加载指定的URDF文件。示例如下：

rosrun gazebo\_ros spawn\_model -file \`rospack find MYROBOT\_description\`/urdf/MYROBOT.urdf -urdf -x 0 -y 0 -z 1 -model MYROBOT

此node的更多参数可通过以下方式查看：

>rosrun gazebo\_ros spawn\_model -h

![spawn\_model\_help](./spawn\_model\_help.png)

下面以baxter功能包为例，了解此方法的过程：

新环境可能需要安装一些依赖库

sudo apt-get install git-core python-argparse python-wstool python-vcstools python-rosdep ros-noetic-control-msgs ros-noetic-joystick-drivers

cd ~/catkin\_ws/src

wstool init .

wstool merge https://raw.githubusercontent.com/RethinkRobotics/baxter/master/baxter\_sdk.rosinstall

wstool update

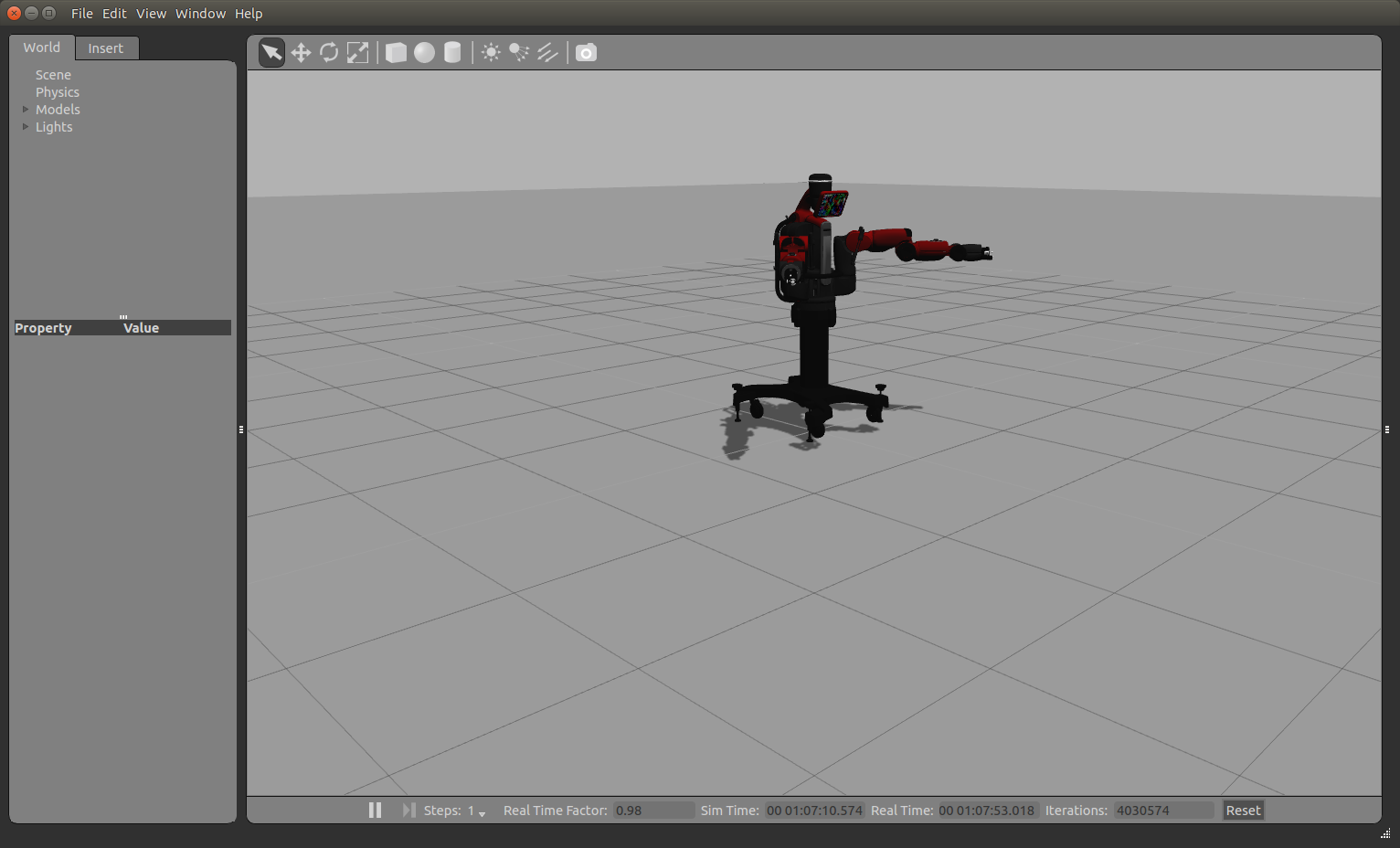
cd ..  
catkin\_make  
source devel/setup.bash

roslaunch gazebo\_ros empty\_world.launch

新开一个终端：

cd ~/catkin\_ws/src   
source devel/setup.bash   
rosrun gazebo\_ros spawn\_model -file `rospack find baxter\_description`/urdf/baxter.urdf -urdf -z 1 -model baxter

结果如下图所示：



#### 4.3.2 模型库方式

这种方式通过将机器人模型添加到Gazebo模型库，通过在world文件中引用模型的方式实现机器人模型的生成加载。因为目前Gazebo是不依赖ROS系统的，Gazebo无法感知ROS包，不能直接将URDF包的路径用于world文件，所以通过更改环境变量的形式添加自建模型库进Gazebo模型库。

这种方法的具体实现过程是先建立只包含单个机器人的模型数据库，然后编写此模型的信息配置文件\*model.config\*，最后通过将模型库的路径添加到环境变量GAZEBO\_MODEL\_PATH中，使得Gazebo搜索模型能包含此目录。

例如，我们建立用于描述机器人信息的功能包\*MYROBOT\_description\*，其典型结构如下：

../catkin\_ws/src /MYROBOT\_description package.xml CMakeLists.txt model.config /urdf MYROBOT.urdf /meshes mesh1.dae mesh2.dae ... /materials /plugins /cad ``` 图中的目录层次结构是符合Gazebo模型数据库样式的，各文件目录作用如下：

* ~/catkin\_ws/src/ 模型库目录
* MYROBOT\_description/ 模型目录
* model.config 模型配置文件，用于Gazebo加载模型
* MYROBOT.urdf 机器人模型描述文件
* meshes/ - URDF使用的纹理文件

文件model.config的内容一般形如：

<?xml version="1.0"?>

<model>

<name>MYROBOT</name>

<version>1.0</version>

<sdf>urdf/MYROBOT.urdf</sdf>

<author>

<name>My name</name>

<email>name@email.address</email>

</author>

<description>

A description of the model

</description>

</model>

使用时，在world文件使用*include*引用模型，如：

<?xml version="1.0" ?>

<sdf version="1.4">

<world name="default">

<include>

<uri>model://ground\_plane</uri>

</include>

<include>

<uri>model://sun</uri>

</include>

<include>

<uri>model://gas\_station</uri>

<name>gas\_station</name>

<pose>-2.0 7.0 0 0 0 0</pose>

</include>

<include>

<uri>model://MYROBOT</uri>

</include>

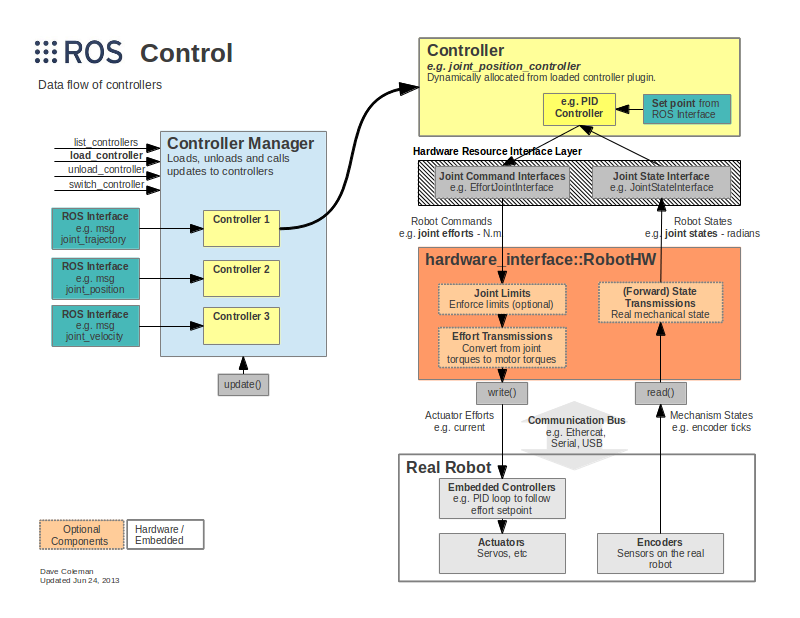
</world>

</sdf>

这样，机器人模型已经能在Gazebo中显示，但要实现物理特性的仿真，还需要添加<gazebo>标签进行相关定义。

### 4.4 仿真控制

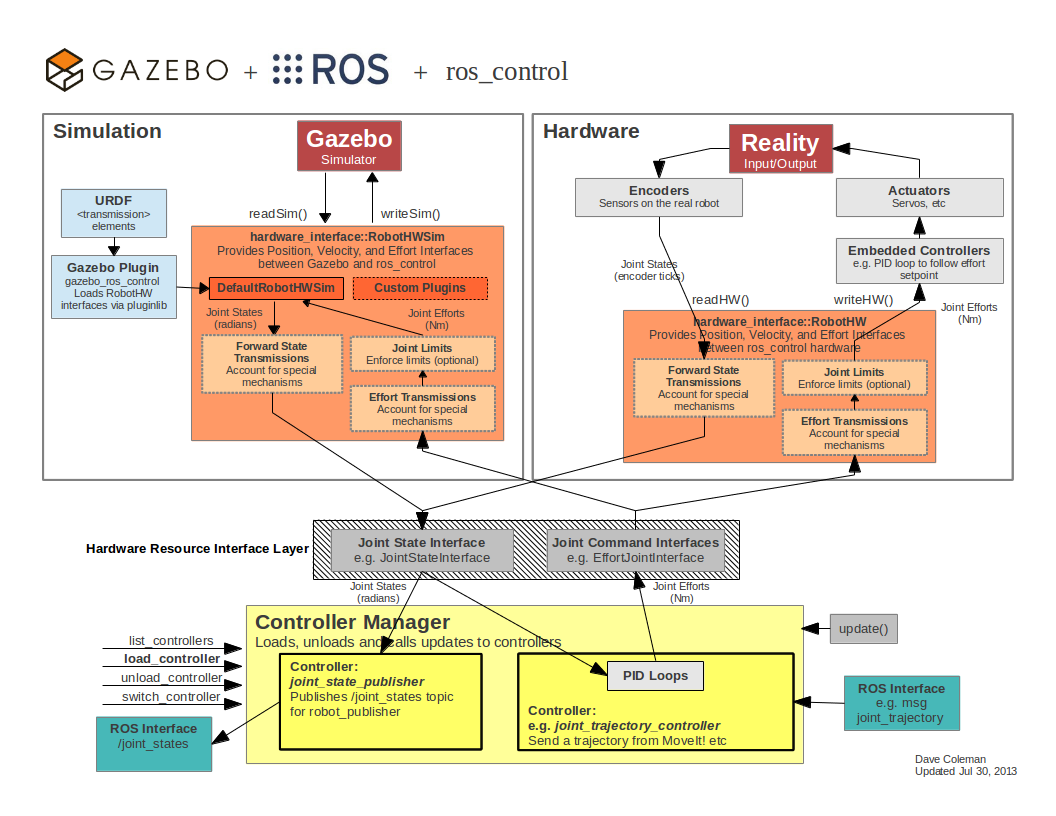
上面我们已经让机器人模型正确加入到Gazebo仿真环境中，下面讲讲ROS如何控制仿真环境中的机器人。在ROS系统中，对仿真环境中或现实环境中的机器人控制都是通过**ros\_control**元功能包实现。此元功能包包括了硬件接口、控制器接口、控制器管理、Transmissions及控制工具箱等功能的功能包。其实质上是一组硬件抽象及控制算法的封装库，负责统一管理硬件驱动与传感器底层细节，处理异常，分配资源，向上提供统一接口；是重写**pr2\_mechanism**元功能包（栈）所得，以适用于所有机器人。各功能包及控制硬件之间数据交互情况如下：

 ROS控制功能包读取机器人编码器数据和*set point*,利用通用的控制回路反馈机制如PID控制器控制向执行器的输出。 其主要架构如下：

* 机器人的硬件通讯库。如：串口，usb， ethercat等。
* hardware\_interface:RobotHW。工作在硬件资源接口层与与实际硬件之间。其中的Transmissions通过映射公式负责硬件与控制器之间输入输出数据的传送转换，以消除偏差；Joint limits用来管理各个joint的effort、position、velocity的极限。
* 硬件资源接口层。通过硬件资源管理器提供对执行器、力矩传感器、惯性传感器等硬件的控制接口。
* 控制器。通过控制回路对effort、position、velocity等状态量进行控制。
* 控制管理器。负责对各控制器进行加载、卸载、更新等管理。

关于ros\_control的具体细节请查看[ROS官网](http://wiki.ros.org/ros_control)。

Gazebo通过ros\_control和适配器插件实现控制器仿真，从而实现控制仿真实现相似的控制过程，其中的数据交互如下：



ROS对仿真环境机器人和实体机器人的控制主要区别在于hardware\_interface:RobotHW部分：Gazebo通过插件加载默认RobotHW的接口或替换默认的RobotHW。所以要在Gazebo中使用ros\_control，必须做两项工作：

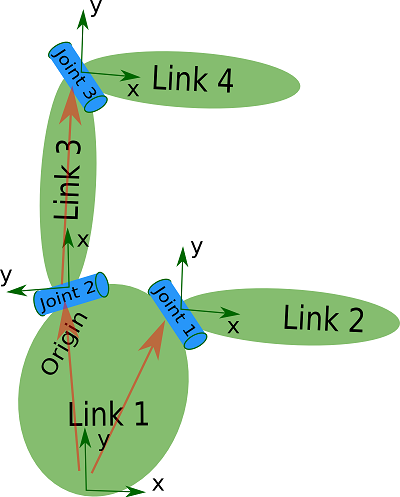
* 向URDF文件中添加<transmission>标签。下面讲解URDF规则时说明。
* 向URDF文件中添加gazebo\_ros\_control插件。示例如下：

xml <gazebo> <plugin name="gazebo\_ros\_control" filename="libgazebo\_ros\_control.so"> <robotNamespace>/MYROBOT</robotNamespace> </plugin> </gazebo>

## 5. URDF与机器人模型

机器人模型的实现主要通过URDF定义的文件和纹理文件实现。URDF是XML类用来描述机器人模型的格式定义，可用来描述机器人的如下特性： - 机器人的运动学和动力学描述 - 机器人视觉呈现 - 机器人碰撞模型

机器人基本模型定义主要由一组**link**元素及一组连接*link*的**joint**元素组成。



其大致定义形如：

<robot name="pr2">

<link> ... </link>

<link> ... </link>

<link> ... </link>

<joint> .... </joint>

<joint> .... </joint>

<joint> .... </joint>

</robot>

下面我们从创建机器人可视模型、添加物理特性两方面进行URDF的学习。

### 5.1 创建机器人可视模型

这一部分以构建形如R2D2的机器人可视模型以例，讲解URDF在定义机器人可视模型中的使用。此示例来自ROS官网的URDF教程，涉及的机器人模型等文件可从功能包[urdf\_tutorial](http://wiki.ros.org/urdf_tutorial)获取，以下所有内容均以此为基础。

#### 5.1.1 创建单一形状结构

上面提到过模型形状结构由*<link>*标签定义，编写urdf/01-myfirst.urdf文件，内容如下：

<?xml version="1.0"?>

<robot name="myfirst">

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</visual>

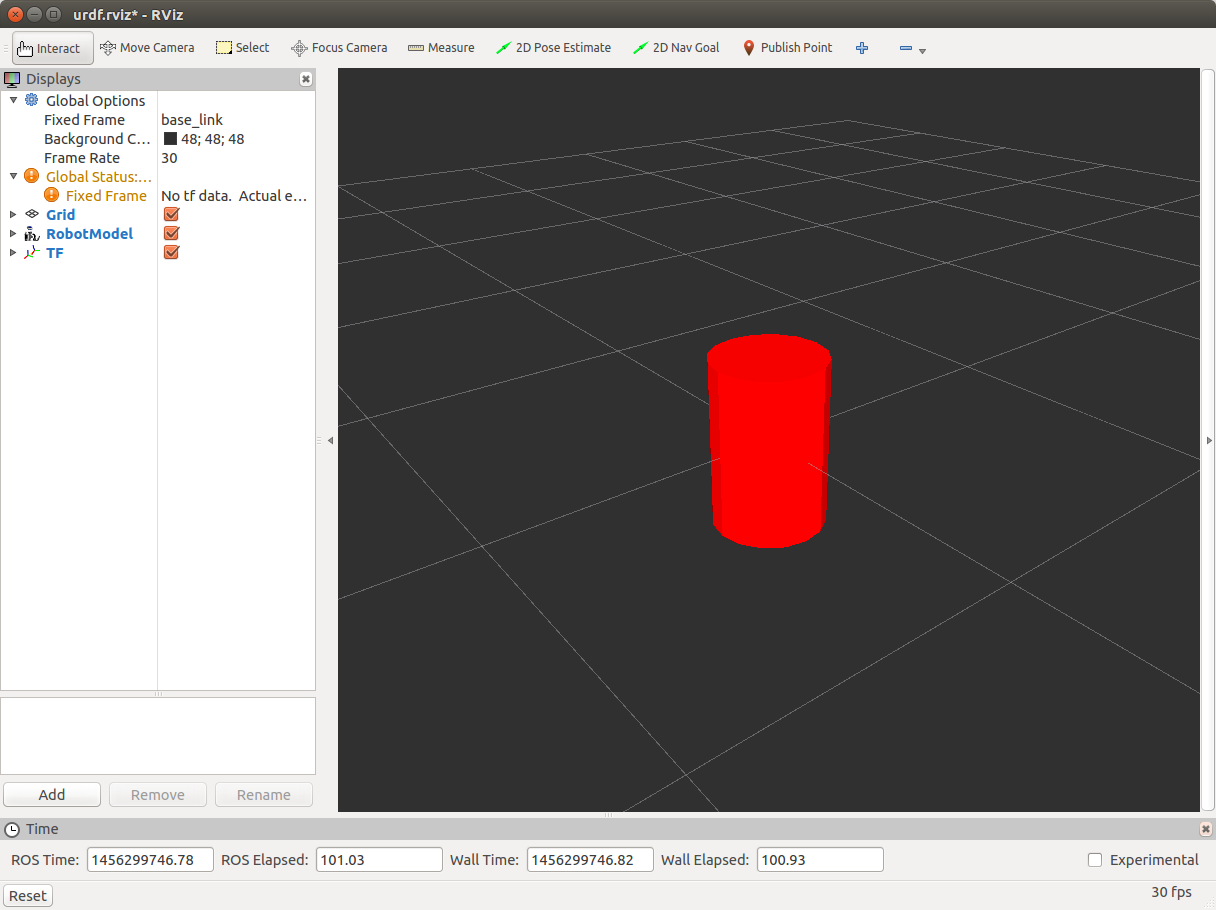
</link>

</robot>

此文件定义了名为*myfirst*的机器人模型，只包含一个长0.6米、半径为0.2米的圆柱体。运行以下命令查看效果：

> roslaunch urdf\_tutorial display.launch model:='$(find urdf\_tutorial)/urdf/01-myfirst.urdf'

如图：



以上命令启动了display.launch文件，并映射了*model*参数。来看看display.launch文件，位于urdf\_tutorial功能包内：

<launch>

<arg name="model" />

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

<param name="robot\_description" textfile="$(arg model)" />

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

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

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

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find urdf\_tutorial)/urdf.rviz"/>

</launch>

此文件主要做了三件工作：将模型定义加载至参数服务器运行相关node，发布sensor\_msgs/JointState and transforms消息 - 以urdf.rviz配置文件启动rviz，显视可视化模型

提醒一下，此处命令通过'$(find urdf\_tutorial)/urdf/01-myfirst.urdf'将model参数值传递进roslaunch文件，必须要加引号，否则$(find urdf\_tutorial)将在参数传递前执行shell命令，一般会出错。也可以通过`rospack find urdf\_tutorial`/urdf/01-myfirst.urdf进行参数传递，将示例参数传递进roslaunch文件与此效果是一样的，一个是在shell执行命令，一个是通过roslaunch参数替换。

#### 5.1.2 创建多形状结构

为了能添加更多*link*元素，就必须使用**<joint>**标签。joint用于连接父*link*和子*link*，有固定joint和活动joint之分。我们先创建固定的joint。见文件urdf/02-multipleshapes.urdf:

<?xml version="1.0"?>

<robot name="multipleshapes">

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</visual>

</link>

<link name="right\_leg">

<visual>

<geometry>

<box size="0.6 .2 .1"/>

</geometry>

</visual>

</link>

<joint name="base\_to\_right\_leg" type="fixed">

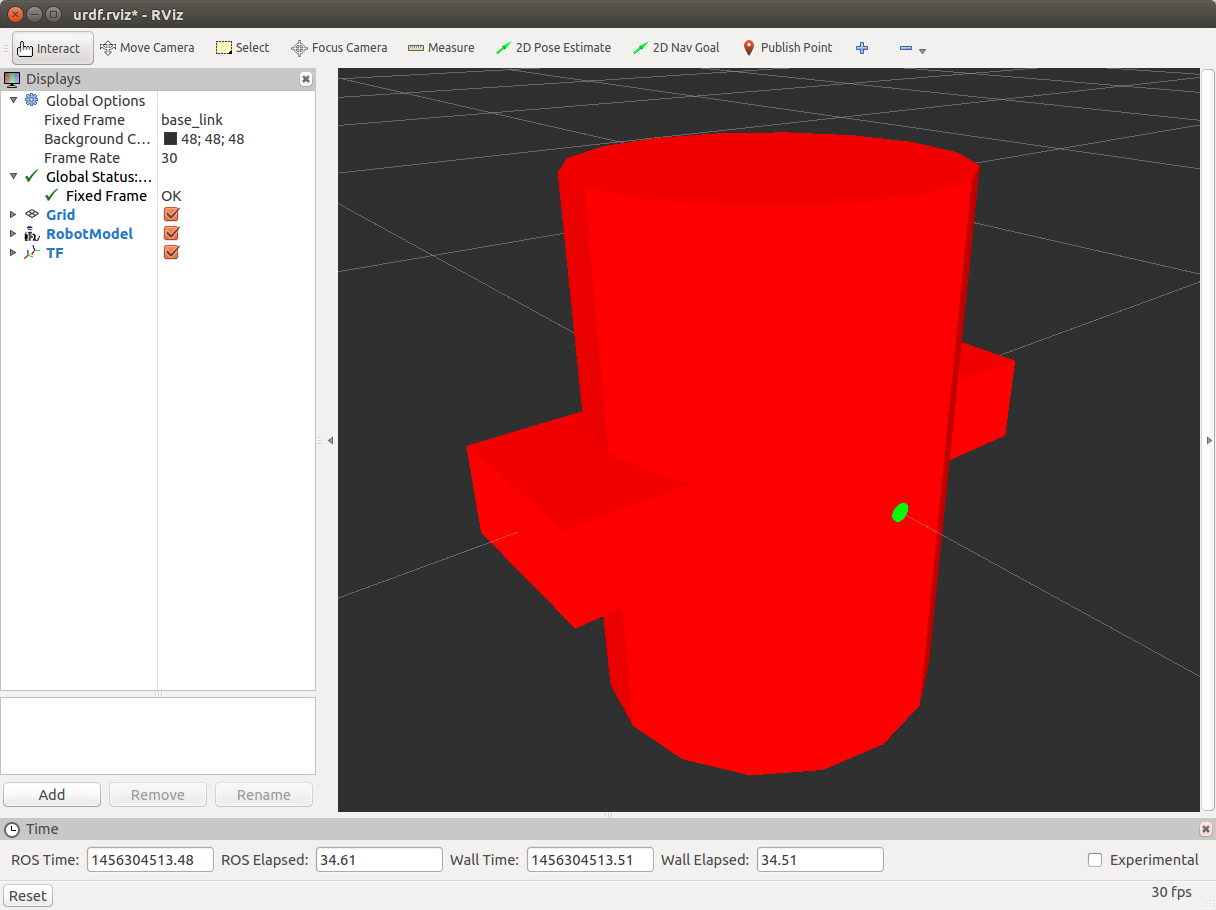
<parent link="base\_link"/>

<child link="right\_leg"/>

</joint>

</robot>

文件增加了一个长0.6米、宽0.2米、厚0.1的长方体，并定义了连接两个*link*的固定*joint*。URDF文件最终是以最顶*link*为根的树形结构，子l*ink*的位置取决于父*link*。运行以下命令通过rviz看看现在的模型： > roslaunch urdf\_tutorial display.launch model:='$(find urdf\_tutorial)/urdf/02-multipleshapes.urdf'

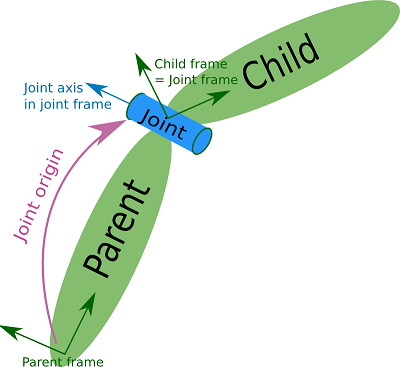
 从图中看出，两个形状结构已经重叠在一起，这是因为形状结构的默认原点为其几何中心，两者原点重合在一起。*link*和*joint*都有其原点，为使形状结构不重叠，必须重新定义其原点。

#### 5.1.3 定义原点

原点的定义都是通过<origin>标签定义，有两个属性：

* xyz - 表示位移偏移量
* rpy- 表示旋转量 (横滚，俯仰，偏航)

对于*link*的原点定义是以其原点为参考的中心偏移量，xyz默认为0向量，rpy默认为模型本身状态；对于*joint*的来说则定义从父*link*到子*link*的变换量，*joint*始终位于子*link*的原点处，其属性xyz默认为0向量，rpy默认为0向量。如图所示：



所以原点定义应如文件urdf/03-origins.urdf所示：

<?xml version="1.0"?>

<robot name="origins">

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</visual>

</link>

<link name="right\_leg">

<visual>

<geometry>

<box size="0.6 .2 .1"/>

</geometry>

<origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>

</visual>

</link>

<joint name="base\_to\_right\_leg" type="fixed">

<parent link="base\_link"/>

<child link="right\_leg"/>

<origin xyz="0.22 0 .25"/>

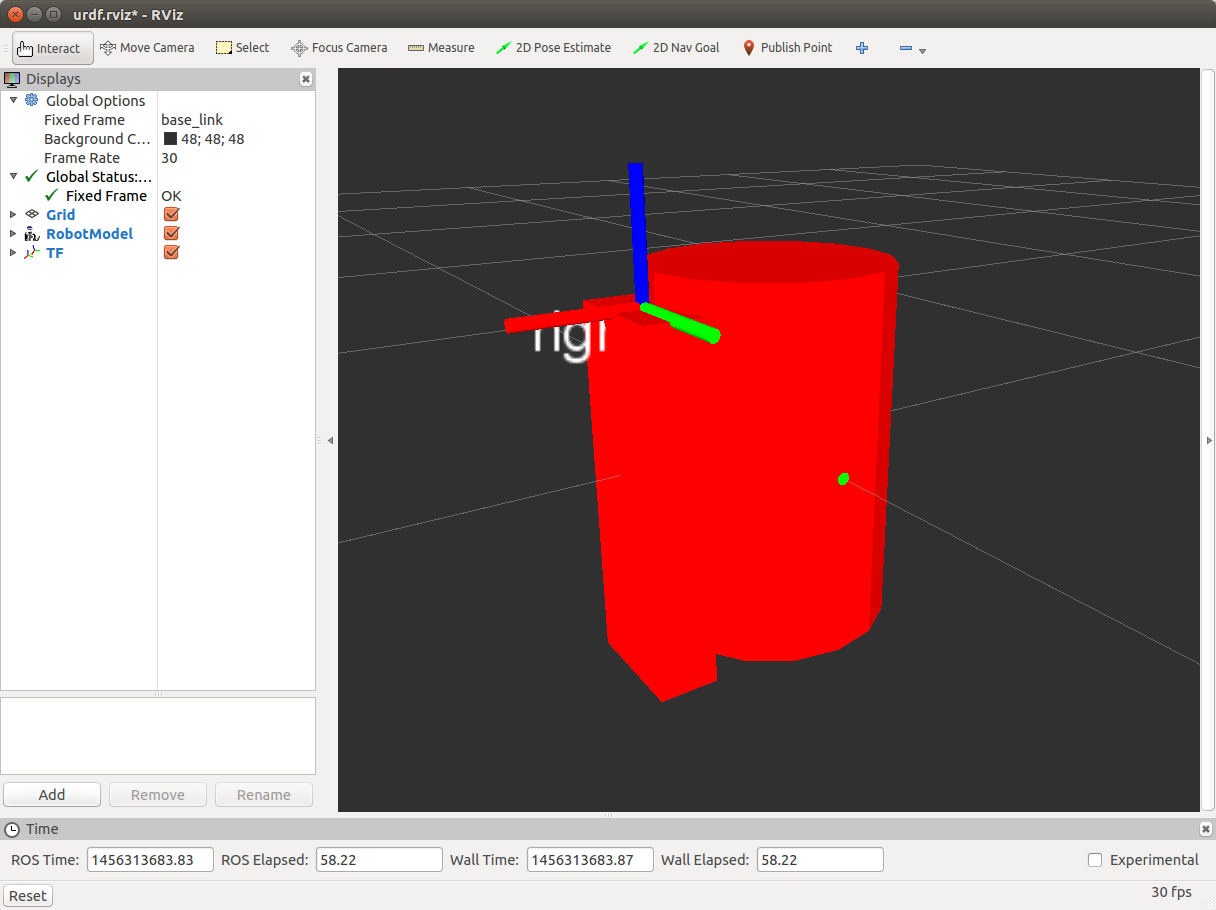
</joint>

</robot>

让我们来分析下以上内容关于原点的定义：

* 对于*joint*原点：参照父*link*坐标系，向X方向偏移0.22，向Z方向偏移0.25。也就是说子*link*的原点将向上向左移动，同时保此朝向不变
* 对于*right\_leg*原点：绕Y轴旋转PI/2，向Z方向偏移-0.3。也就是形状结构的中心向下移动0.3，其结构绕Y轴旋转。

其结果如下：



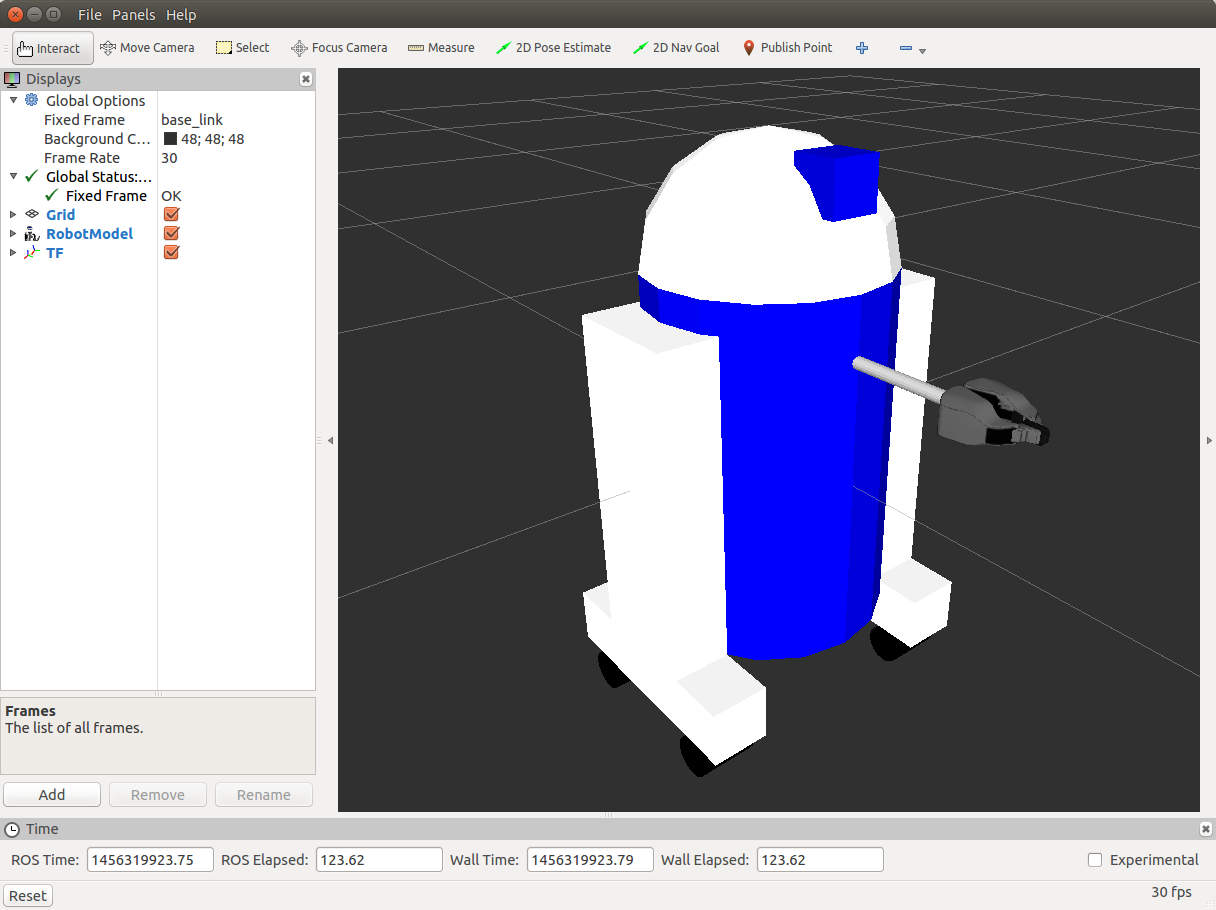
按同样方法，我们可以添加*left\_leg*等其它形状结构。

#### 5.1.4 材质与纹理

以上模型都是单一外观，要使其更美观，需为为其指定材质和纹理。

* + - 添加材质 材质的添加通过<material>标签，其属性有：
* name – 材质的名字，可引用已定义的材质。
* <color> -可选。以**rgba**形式，用4个数字表示材质的颜色，数字分别代表红、绿、蓝、透明度，取值范围在[0,1]内，如：.1,0,0,.5。
* <texture>- 可选。以**filename**指定文本文件。
  + - 添加纹理 纹理文件能使模型更贴近现实。目前支持的纹理文件格式有：.dae 和.stl。通过<mesh>标签指定*filename*实现加载纹理文件。

示例使用*pr2\_description*中的纹理文件，最终效果为：



#### 5.1.5 活动类joint

上面我们使用的*joint*都是固定的，模型在rviz中不能操作活动的。可以通过更改*joint*类型让模型动起来。其类型主要有：

* revolute 沿轴旋转的铰链型joint，具有限定范围。
* continuous 一个连续的铰链型joint，绕轴旋转，没有上下限。
* prismatic 沿轴滑动的滑动型joint，具有限定范围。
* fixed 固定的joint。
* floating 有6个自由度的joint。
* planar 垂直于轴做平面运动的joint。

### 5.2 定义物理特性和碰撞属性

如果让机器人模型用于Gazebo仿真环境，还要对其添加物理特性和碰撞属性。

#### 5.2.1 碰撞属性

元素*link*的碰撞属性有可能不同于其视觉属性，如常用简化的碰撞模型来减少计算时间。同一个*link*可存在多个碰撞属性，使用其定义的几何组合表示该*link*的碰撞属性。 碰撞属性使用<collision>标签定义，其组成有：

* + - name (可选) 指定*link*该几何部分名字，便于引用。
* <origin> (可选) 指定该元素的参考坐标系，以*link*的参考坐标系为参照。
* xyz (可选，默认为0向量) 表示*x,y,z*方向的位移。
* rpy (可选) 表示固定坐标轴的旋转。
  + - <geometry> 定义几何形状。

示例如下：

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

<material name="blue">

<color rgba="0 0 .8 1"/>

</material>

</visual>

<collision>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</collision>

</link>

#### 5.2.2 物理特性

为了使机器人模型能正解地仿真，需要定义<link>的物理特性。一般需定义的物理特性有：惯性、接触系数、*joint*的动力学。

* 惯性 惯性通过<inertial>标签定义，其组成有：
* <origin> (可选) 指定惯性参考坐标系，以*link*参考坐标系为参照。坐标系原点必须位于重心。- xyz (可选，默认为0向量)表示*x,y,z*方向的位移。- rpy (可选)表示固定坐标轴的旋转。
* <mass> 该*link*的质量。
* <inertia> 用于惯性坐标系统的3X3旋转惯性矩阵，由于转动惯性矩阵是对称的，所以在指定6个元素：ixx, ixy, ixz, iyy, iyz, izz。
* 接触系数 接触系数主要指*link*之间接触时特性，使用<contact\_coefficients>标签定义，其属性有：
* mu - 摩擦系数
* kp - 刚度系数
* kd - 阻尼系数
* 动力学 *link*如何运行由<dynamics>定义，其属性有：
* friction - 物理静摩擦力。对滑动*joint*，单位是N；对于旋转*joint*，单位是N\*m。
* damping - 物理阻尼值。对于滑动*joint*，单位是Ns/m；对于旋转*joint*，单位是Nsm/rad。

### 5.3 在Gazebo中使用URDF

虽然前面我们已经在Gazebo中加载了URDF定义的可视化模型，URDF只能孤立地定义单一机器人的运动学及动力学特性，而不能在世界模型中定义自身的姿态，也不能定义环境类模型，为使其更好地用于Gazebo，需要如下的转换步骤：

* 必须步骤
* 在每一个<link>内添加<inertia>标签。这在定义物理特性时提到过。
* 可选步骤
* 在每一个<link>内添加<gazebo>标签。将外观颜色转换成Gazebo格式；将stl文件转换成dae文件，获得更好的文字效果；添加传感器插件。
* 在每一个<joint>内添加<gazebo>标签。设置适当的阻尼动力系数；添加执行器控制插件
* 在<robot>标签内添加<gazebo>
* 如果机器人要附加到world/base\_link上，需添加<link name="world"/>

细节请查看[Gazebo官网](http://gazebosim.org/tutorials/?tut=ros_urdf)。

### 5.4 补充说明

ROS官方提供了一个名为**xacro**的宏脚本语言来定义机器人描述文件，以使其更简单易读，并提供了工具转换成URDF。

通常较复杂的机器人模型都是用三维建模软件建立，solidworks提供了导出URDF文件的插件。

## 6 Gazebo与ROS通信接口

Gazebo提供了一系列的ROS接口，以便于修改，获取仿真环境的各方面信息。

### 6.1 接口插件

在本课开始提及ROS通过插件*gazebo\_ros\_api\_plugin*和*gazebo\_ros\_paths\_plugin*封装了Gazebo的接口：

* *gazebo\_ros\_api\_plugin* 位于与gazebo\_ros包，用于初始化名为*gazebo*的node。其集成了ROS回调调度器（通过message传递），通过Gazebo内部调度器提供ROS接口。ROS通过这些接口可管理仿真环境。此插件只由*gzserver*加载。
* *gazebo\_ros\_paths\_plugin* 同样位于与gazebo\_ros包，用于Gazebo查找ROS资源，如解析ROS功能包路径。此插件由*gzserver*和*gzclient*加载。

### 6.2 Gazebo发布的参数

Gazebo只向ROS参数管理器发布一个参数： /use\_sim\_time : Bool - 表征ROS使用Gazebo发布的/clock topic作为ROS的时间。 ROS系统为了和仿真时间同步，可设为true。一般可用于rviz，默认为true。

### 6.3 Gazebo订阅的话题

Gazebo订阅的话题有：

* ~/set\_link\_state : gazebo\_msgs/LinkState - 设置*link*的状态 (pose/twist)。
* ~/set\_model\_state : gazebo\_msgs/ModelState -设置模型的状态 (pose/twist)。

### 6.4 Gazebo发布的话题

Gazebo发布的话题有：

* /clock : rosgraph\_msgs/Clock - 发布仿真环境时间，和/use\_sim\_time参数配合使用。
* ~/link\_states : gazebo\_msgs/LinkStates - 发布各*link*在仿真环境中的状态
* ~/model\_states : gazebo\_msgs/ModelStates - 发布各模型在仿真环境中的状态

### 6.5 Gazebo提供的服务

Gazebo提供了众多服务用于仿真环境中的模型管理，主要分为模型创建与销毁服务、仿真环境及对象状态属性设置服务、仿真环境及对象状态属性获取服务、力控制服务、仿真器控制服务五类。细节查阅官网-[Tutorial: ROS Communication](http://gazebosim.org/tutorials/?tut=ros_comm)。

小结：本课介绍了Gazebo仿真工具，示例了URDF定义机器人模型，简述了Gazebo和ROS的通信接口。要进一步了解Gazebo中ROS中的使用问题请查阅ROS及Gazebo官网。

## 实验内容

### URDF模型创建

创建一个URDF移动机器人平台

<?xml version="1.0" ?>

<robot name="irobot">

<!-- 设置 base\_footprint -->

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="0.001" />

</geometry>

</visual>

</link>

<!-- 添加底盘 -->

<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\_link\_joint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link"/>

<origin xyz="0 0 0.055" />

</joint>

<!-- 添加驱动轮 -->

<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\_wheel\_joint" 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\_wheel\_joint" type="continuous">

<parent link="base\_link" />

<child link="right\_wheel" />

<origin xyz="0 -0.1 -0.0225" />

<axis xyz="0 1 0" />

</joint>

<!-- 添加万向轮(支撑轮) -->

<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\_wheel\_joint" 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\_wheel\_joint" type="continuous">

<parent link="base\_link" />

<child link="back\_wheel" />

<origin xyz="-0.0925 0 -0.0475" />

<axis xyz="1 1 1" />

</joint>

</robot>

launch文件启动URDF模型

<?xml version="1.0" ?>

<launch>

<arg name="model" default="$(find xacro)/xacro --inorder '$(find irobot\_description)/urdf/irobot\_base.urdf'" />

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

<param name="robot\_description" command="$(arg model)" />

<!-- 设置GUI参数，显示关节控制插件 -->

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

<!-- 运行joint\_state\_publisher节点，发布机器人的关节状态 -->

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

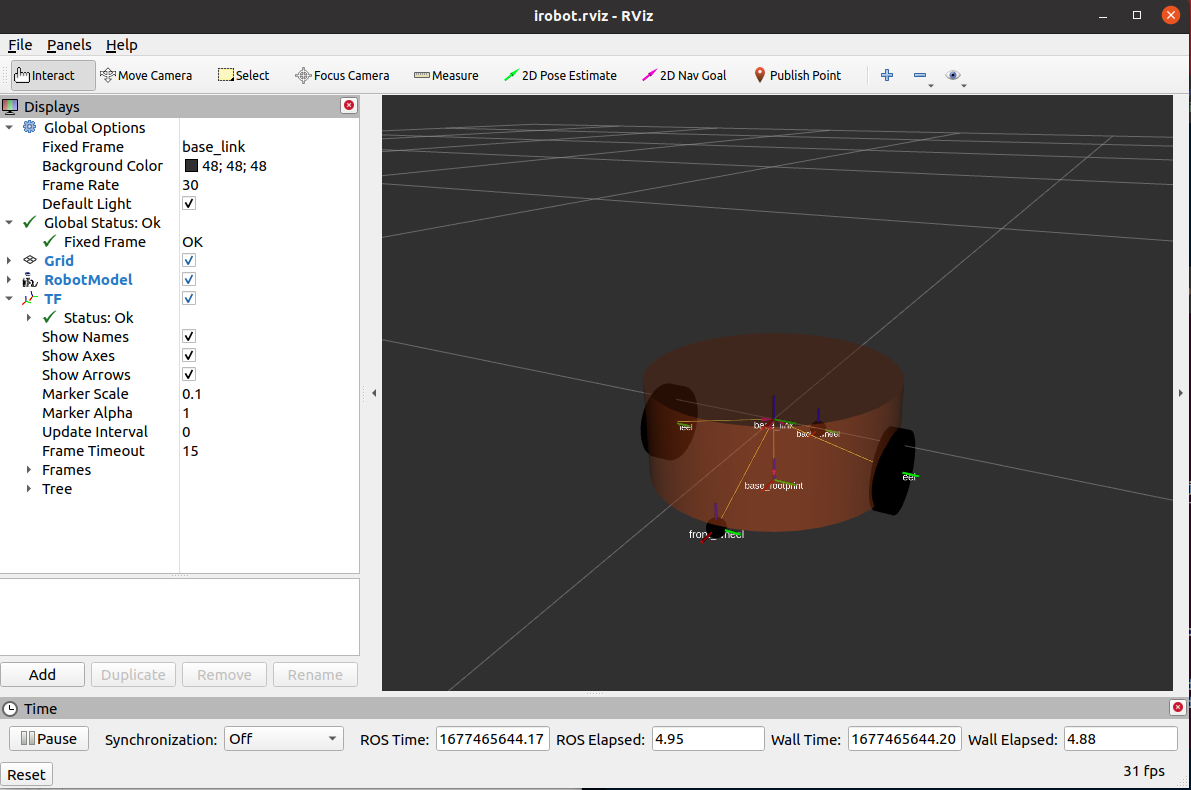
<!-- 运行robot\_state\_publisher节点，发布tf -->

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

<!-- 运行rviz可视化界面 -->

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find irobot\_description)/config/irobot.rviz" required="true" />

</launch>



### 2、XACRO模型建立

将URDF格式移动机器人改写成XACRO文件

<?xml version="1.0"?>

<!-- 根标签，必须声明 xmlns:xacro -->

<robot name="irobot\_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}" 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="base\_l\_wheel\_joint" flag="1" />

<xacro:add\_wheels name="base\_r\_wheel\_joint" 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>

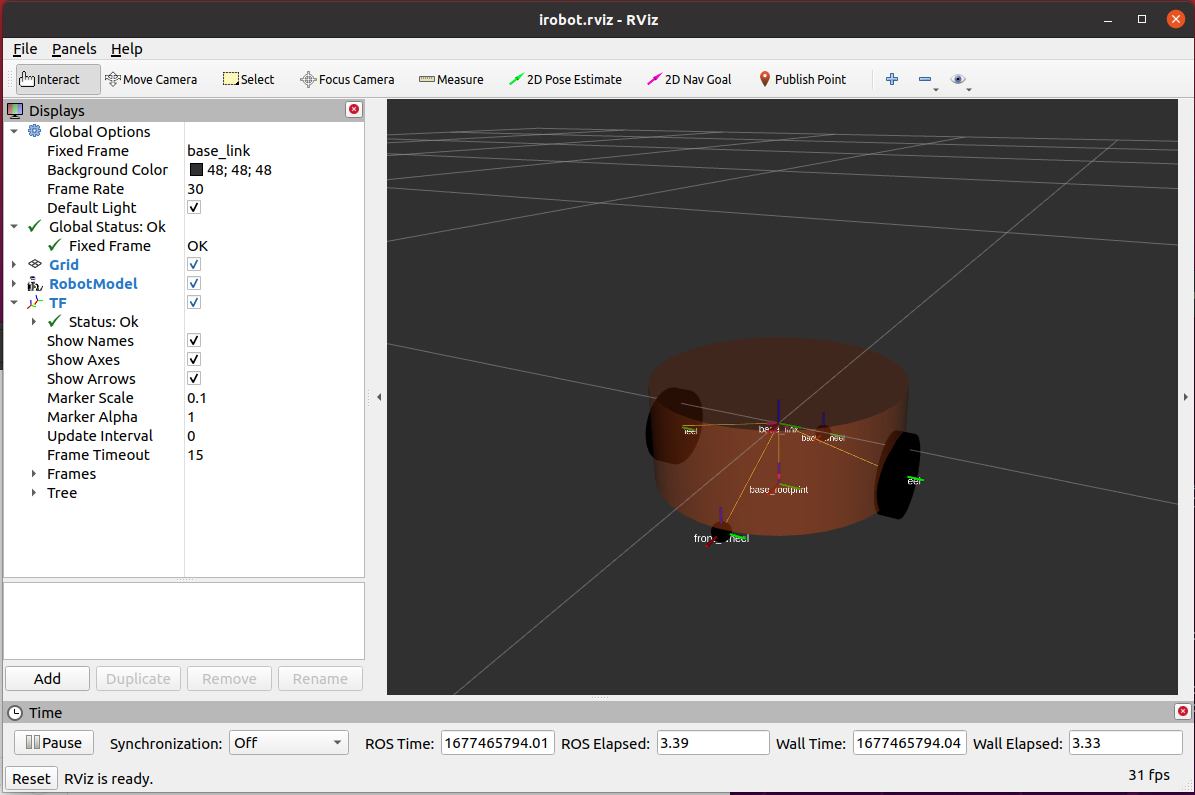
</xacro:macro>

<xacro:add\_support\_wheel name="front" flag="1" />

<xacro:add\_support\_wheel name="back" flag="-1" />

</robot>

使用launch文件启动XACRO模型



## 参考文献

1. <http://wiki.ros.org/>
2. https://wiki.ros.org/ROS/Tutorials