***roslaunch*** is the command used to launch a ROS program. Its structure goes as follows:

In [ ]:

```
roslaunch <package_name> <launch_file>
```

As you can see, that command has two parameters: the first one is **the name of the package** that contains the launch file, and the second one is **the name of the launch file** itself (which is stored inside the package).

ROS 使用 package 来组织它的程序。你可以把 package 认为是一个具体的 ROS 程序包含的所有文件：所有 cpp 文件、python 文件、配置文件、编译文件、launch 文件和参数文件。

上述的所有文件按照以下形式进行组织：

- launch 文件夹：包含了 launch 文件。
- src 文件夹：包含了 cpp 和 Python 的源文件。
- CMakeLists.txt：编译时 cmake 的规则集合。
- package.xml：包信息和依赖。

为了前往 ROS 的 package 中，ROS 提供了一个命令叫做 roscd：

Roscd <package_name>

```
roscd turtlebot_teleop
cd launch
cat keyboard_teleop.launch
```

keyboard_teleop.launch

```
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key.py" name="turtlebot_teleop_keyboard" output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/> <!-- cmd_vel_mux/input/teleop"/-->
  </node>
</launch>
```

在这个启动文件中，你有一些额外的 tag 来设置参数和 remap。所有启动文件包含一个<launch>标签，在这个标签中，你可以发现<node>标签，其中指定了如下的一些参数:

1. pkg=”package_name” #包含了所要执行的 ROS 包的名字
2. type=”python_file_name.py” #包含了我们希望执行的程序
3. name=”node_name” #包含了我们希望把我们的 python 脚本执行在哪

个 ROS 节点上

4. output=”type_of_output” #通过哪个 channel 来 print 我们的 python 文件

当我们希望自己创建 package 时，我们需要在所谓的 catkin workspace 中工作。Catkin workspace 是我们的硬盘上自己的 ROS 包所必须在的位置。只有在之中的 ros package 才能被使用。通常这个 catkin workspace 目录叫做 catkin_ws。

我们通过 roscd 命令可以直接到达 catkin_ws/devel 目录下。

我们可以使用 catkin_create_pkg my_package rospy 来创建在 catkin_ws/src 目录下创建我们自己的包。这个命令的格式如下：

catkin_create_pkg <package_name> <package_dependencies>，其中 package_name 是我们希望创建的 package 的名字，而 package_dependencies 是我们创建的 package 需要依赖的其他 ROS 包。

创建完成后，我们可以使用以下命令来过滤查找我们创建的 package

```
user:~/catkin_ws/src$ rospack list | grep my_package
my_package /home/user/catkin_ws/src/my_package
```

在 package 中，我们创建一个 launch 目录，其中存放新建的 my_package_launch_file.launch 文件。

my_package_launch_file.launch

```
<launch>
    <!-- My Package launch file -->
    <node pkg="my_package" type="simple.py" name="ObiWan"    output="screen">
    </node>
</launch>
```

同时在 src 目录下，我们可以新建 simple.py 文件。

simple.py

```
#! /usr/bin/env python
# This line will ensure the interpreter used is the first one on your environment's $PATH. Every Python file needs
# to start with this line at the top.

import rospy # Import the rospy, which is a Python library for ROS.

rospy.init_node('ObiWan') # Initiate a node called ObiWan

print("Help me Obi-Wan Kenobi, you're my only hope") # A simple Python print
```

注意第一行是必须要加上的，否则就会报出如下错误：

```
NODES
  /
    ObiWan (my_package/simple.py)

ROS_MASTER_URI=http://3_simulation:11311

RLException: Unable to launch [ObiWan-1].
If it is a script, you may be missing a '#!' declaration at the top.
The traceback for the exception was written to the log file
```

#!/usr/bin/python 是告诉操作系统执行这个脚本的时候，调用/usr/bin 下的 python 解释器；
#!/usr/bin/env python 这种用法是为了防止操作系统用户没有将 python 装在默认的/usr/bin 路径里。当系统看到这一行的时候，首先会到 env 设置里查找 python 的安装路径，再调用对应路径下的解释器程序完成操作。
#!/usr/bin/python 相当于写死了 python 路径;
#!/usr/bin/env python 会去环境设置寻找 python 目录,推荐这种写法

配置完成后，我们就可以使用 roslaunch my_package my_package_launch_file.launch 来运行脚本。

rosnode list 可以显示出当前所有正在运行的 node 节点。

```
user:~$ rosnode list
/ObiWan
/gazebo
/mobile_base_nodelet_manager
/robot_state_publisher
/rosout
user:~$ rosnode info /ObiWan
--------------------------------------------------------------------------------
Node [/ObiWan]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]

Services:
 * /ObiWan/get_loggers
 * /ObiWan/set_logger_level


contacting node http://3_xterm:41185/ ...
Pid: 3900
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (45173 - 192.168.16.7:49920) [10]
    * transport: TCPROS
 * topic: /clock
    * to: /gazebo (http://3_simulation:39079/)
    * direction: inbound
    * transport: TCPROS
```

我们可以通过 rosnode info 来得到具体 node 的 info。

当我们创建一个 package 的时候，我们通常会去编译它。ROS 开发者最常用的编译它们的 package 的方法是 catkin_make。这个命令会编译我们的 src 目录，它需要在 catkin_ws 目录下启动，并且这是强制要求的。我们在别的目录下是没有办法使用它编译的。

在编译以后我们需要使用命令 source devel/setup.bash 来确保 ROS 始终能获取到我们 workspace 中最新的更新。

我们也可以使用命令 catkin_make --only-pkg-with-deps <package_name>来只编译其中指定的 package。

我们也可以使用 catkin build 来编译我们 catkin_ws 中的 src 目录下的所有文件。同样的，这个命令也需要在 catkin_ws 根目录下执行。

一个 parameter server 是一个目录，ROS 使用它来存储参数。这些参数可以被 node 在运

行时使用并且可以作为 static data 使用，比如 configuration parameter。

查看 parameter server 中的所有参数名字：rosparam list

获取指定参数：rosparam get <parameter_name>

设置指定参数：rosparam set <parameter_name> <value>

为了让之前提到的所有东西运行起来，我们需要运行一个 roscore。roscore 是掌管 ROS 系统一切东西的主进程。始终需要一个 roscore 在运行才可以使用 ROS。启动 roscore 的命令就是 roscore，示例图如下：

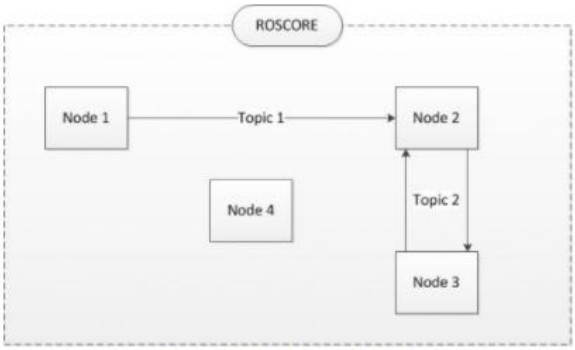


Fig.1.2 - ROS Core Diagram

ROS 使用一系列的 Linux 系统环境变量来正常工作。我们可以使用命令 export | grep ROS 来获取它们。

```
user:~/catkin_ws$ export | grep ROS
declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-lisp:/home/simulations/pu
blic_sim_ws/devel/share/common-lisp"
declare -x ROS_DISTRO="noetic"
declare -x ROS_ETC_DIR="/opt/ros/noetic/etc/ros"
declare -x ROS_HOSTNAME="3_xterm"
declare -x ROS_IP=""
declare -x ROS_MASTER_URI="http://3_simulation:11311"
declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src/my_package:/home/simulations/public_sim_ws/src:/opt
/ros/noetic/share"
declare -x ROS_PYTHON_VERSION="3"
```

其中最重要的变量是 ROS_MASTER_URI 和 ROS_PACKAGE_PATH。

ROS_MASTER_URI -> Contains the url where the ROS Core is being executed. Usually, your own computer (localhost).

ROS_PACKAGE_PATH -> Contains the paths in your Hard Drive where ROS has packages in it.

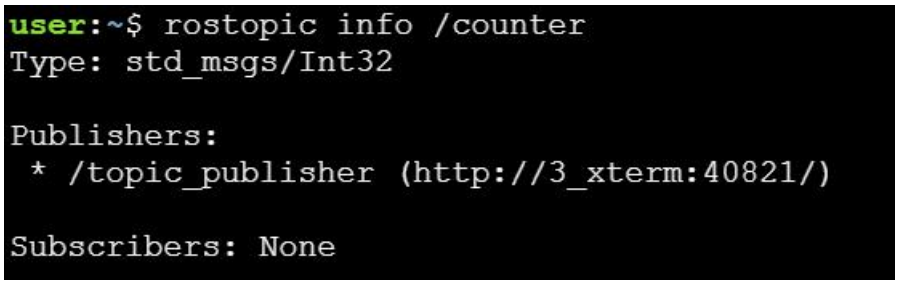
在如下的例子中，我们创建一个 my_examples_pkg

simple_topic_publisher.py

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

rospy.init_node('topic_publisher')
pub = rospy.Publisher('/counter', Int32, queue_size=1)
rate = rospy.Rate(2)
count = Int32()
count.data = 0
```

```
while not rospy.is_shutdown():
   pub.publish(count)
   count.data += 1
   rate.sleep()
```

运行这个程序之后，我们创建了一个 topic 叫做/counter，并且我们无限 publish 一个不断增长的整数。一个 topic 就像一个管道。node 节点使用 topic 来 publish 信息给其他 node。从而实现腾讯，我们可以通过命令：rostopic list 来看到在当前时间的所有 topic。我们可以再使用 rostopic info 来查看具体 topic 的信息。




上图中的信息指定了传递的信息的类型是 std_msgs/Int32，正在输出信息的 node 是/topic_publisher，并且这个消息没有订阅者。

我们可以使用命令：rostopic echo /counter 来实时查看这个 topic 的输出。

我们进一步解释之前的 simple_topic_publisher.py 代码，其实它就是实例化了一个 node，并且创建了一个 publisher，持续地向其中输出一系列连续的整数。所以 publisher 是持续向 topic 中输入消息的 node。而 topic 是一个 channel，其他 ROS 节点可以从中输入或者读取信息。

我们可以使用 rostopic echo <topic_name>来读取被发布到 topic 中的信息。这会输出所有被发布的消息，有时候因为数量太大或者每个消息有一个很大的结构会导致非常庞大。这时候我们就可以使用：rostopic echo <topic_name> -n1，它只会输出最近的一条 message 消息。

Unit 4

订阅者（subscriber）是从 topic 中读取信息的节点（node）。

simple_topic_subscriber.py

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

def callback(msg):
   print (msg.data)

rospy.init_node('topic_subscriber')
sub = rospy.Subscriber('/counter', Int32, callback)
rospy.spin()
```

上述程序就是创建了一个对/counter 话题的 subscriber，并且设置了回调函数。

我们可以使用命令 rostopic pub /counter std_msgs/Int32 5，向着/counter 话题主动推送一个 5，这样我们的程序就可以阅读到 5 并且输出到 screen 上。

```
user:~/catkin_ws$ rostopic echo /counter
WARNING: no messages received and simulated time is active.
Is /clock being published?
data: 5
---
```

习题 4.2 Modify the code in order to print the odometry of the robot.

提示：

1.The odometry of the robot is published by the robot into the /odom topic.

2.You will need to figure out what message uses the /odom topic, and how the structure of this message is.

首先我们可以通过 rostopic 知道，/odom 话题提供的是 nav_msgs/Odometry 数据结构

```
user:~$ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://3_simulation:38929/)

Subscribers:
 * /ObiWan1 (http://3_xterm:44921/)
```

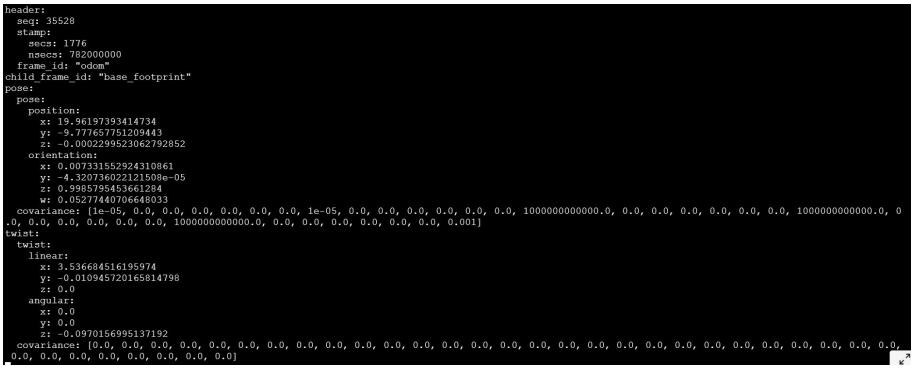
然后我们在 package.xml 引入 nav_msgs，这样我们就不用重新创建一个 package 了

```
<buildtool_depend>catkin</buildtool_depend
<build_depend>geometry_msgs</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>nav_msgs</build_depend>
<build_export_depend>geometry_msgs</build_
<build_export_depend>rospy</build_export_d
<build_export_depend>std_msgs</build_expor
<build_export_depend>nav_msgs</build_expor
<exec_depend>geometry_msgs</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
<exec_depend>nav_msgs</exec_depend>
```

```
#! /usr/bin/env python

import rospy
from nav_msgs.msg import Odometry

def callback(msg):
    print (msg)
```

```
rospy.init_node('topic_subscriber')
sub = rospy.Subscriber('/odom', Odometry, callback)
rospy.spin()
```

写上对应代码，就可以不停输出这个数据结构了。




- Exercise 4.3 -

1. Add to {Exercice 4.2}, a Python file that creates a publisher that indicates the age of the robot.
2. For that, you'll need to create a new message called **Age.msg**. To see how you can do that, have a look at the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

- End of Exercise 4.3 -

练习 4.3

我们知道了如何发布一些已有的类型，如 Int32。但是我们如果想自定义一个 message 类型该怎么办呢？

我们需要做到以下几步：

1. 创建 msg 目录
2. 创建一个文件叫做 Name_of_your_message.msg
3. 修改 CMakeLists.txt 文件
4. 修改 package.xml 文件
5. 编译
6. 在代码中使用这个 message

比如说，我们希望在 package 中添加一个 Age.msg，其中包括 float32 的年月日。那么我们就可以在 Age.msg 中添加

```
float32 years
float32 months
float32 days
```

第二步是修改 CMakeLists.txt 文件，我们需要修改其中的 4 个函数

1. find_package()

这个函数是所有 package 需要用来编译 topic, service 和 action 的 message 的地方。

```
find_package(catkin REQUIRED COMPONENTS
          rospy
          std_msgs
          message_generation      # Add message_generation here, after the other packages
)
```

2. add_message_files()

这个函数包含了这个包中编译中需要的所有的 message(在 msg 目录下)，如下所示：

```
add_message_files(
        FILES
        Age.msg
      ) # Dont Forget to UNCOMENT the parenthesis and add_message_files TOO
```

3. generate_messages()

这里是在编译 message 中需要 import 的 dependency

```
generate_messages(
        DEPENDENCIES
        std_msgs
) # Dont Forget to uncoment here TOO
```

4. catkin_package()

这里的 package 是所有被需要来执行某些行为。所有这里声明的 package 都需要在 package.xml 写为 exec_depend。

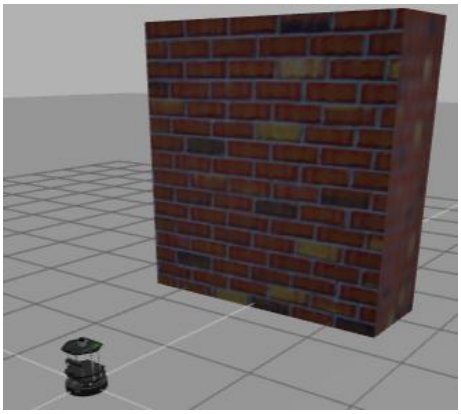
```
catkin_package(
        CATKIN_DEPENDS rospy message_runtime      # This will NOT be the only thing here
)
```

在 package.xml 中，我们也需要添加依赖:

```
<build_depend>message_generation</build_depend>

<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

小 quiz：

1. 创建一个发布者，向着/cmd_vel 中写对 robot 的控制命令
2. 创建一个订阅者，从/kobuki/laser/scan 中读取激光雷达的数据。
3. 根据读到的数据来动态地计算需要向/cmd_vel 中发送的控制命令，从而避开机器人远端的墙。



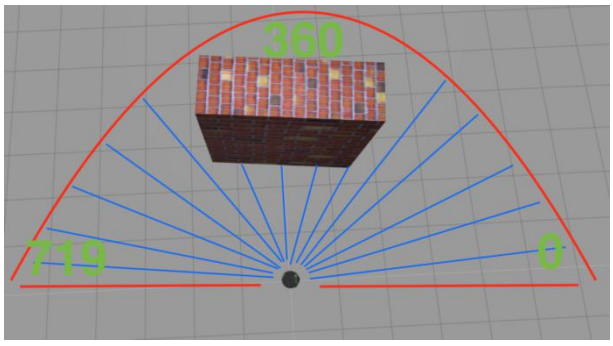
程序的逻辑如下：

1. 如果测得的距离大于 1m，我们可以认为 robot 面前没有障碍物，robot 会往前走
2. 如果小于 1m，说明有障碍物，robot 会左转
3. 如果 robot 的右侧距离小于 1m，robot 会左转
4. 如果 robot 的左侧距离小于 1m，robot 会右转

首先，我们可以使用 rosmsg show sensor_msgs/LaserScan 来查看 LaserScan 数据结构的格式：

```
std_msgs/Header header
   uint32 seq
   time stamp
   string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges <-- Use only this one
float32[] intensities
```

我们只需要用到 ranges，ranges 有 720 个数据点，如下图所示：




可以通过 0 来读 robot 右侧的距离，719 来读 robot 左侧的距离，360 是 robot 前方的距离。

subscriber.py

```
#! /usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

def callback(msg):
    print (len(msg.ranges))
    distance_to_wall = msg.ranges[360]
    print (msg.ranges[360], "m")
    twister = Twist()
    if distance_to_wall > 1:
        twister.linear.x = 1
    else:
        twister.linear.x = 0
        twister.angular.z = 1
    pub.publish(twister)



rospy.init_node('topics_quiz_node')
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
sub = rospy.Subscriber('/kobuki/laser/scan', LaserScan, callback)
rospy.spin()
```

使用 topic 我们可以做到任何我们需要做到的事情，甚至是操控一个航天机器人。许多的 ros package 只使用了 topic 来完美实现其功能。那么我们为什么还需要学习 service 呢？因为在一些情况下，topics 是不充分的或者使用起来太过笨重。

为了理解什么是 service 并且什么时候使用 service，我们需要把 services 和 topics 还有 actions 比较。

想象一下我们有一个自己的 BB-8 机器人，它有激光雷达、人脸识别系统和导航系统。激光雷达使用 topic 来以 20hz 的频率发布所有激光雷达获取的数据。我们之所以发布在 topic 上是因为我们希望其他的 ROS 系统都可以通过订阅这个 topic 的方式来获取到激光雷达实时发布的数据，比如我们的导航系统。

人脸识别系统会作为一个 service 来提供，我们的 ros 程序会调用这个服务并且等待它返回一个人脸识别的结果。

而导航系统作为一个 action 来提供。我们的 ROS 程序会调用这个 action 来移动我们的机器人，并且当这个 action 在执行的时候，我们的程序可以执行其他的任务。并且 action 在移动的过程中会返回给你 feedback（eg.距离目的地的距离）

那么 service 和 action 的区别到底是什么呢？Service 是同步的，当我们的 ROS 程序调用了 service 以后，就必须等 service 执行完返回。Action 是异步的，它就像启动一个新的线程一样。当 ROS 程序调用一个 action 以后，ROS 程序可以继续执行其他任务，因为 action 是执行在另一个线程中。结论：当我们的 ROS 程序没有结果就没有办法执行后续逻辑的时候，

就使用 service 来阻塞住其他行为等待 service 执行完毕。

同样地，我们可以通过 rosservice list 来列出所有的 service。也可以通过 rosservice into <service_name>来获取一个 service 的具体信息。

```
user:~$ rosservice info /execute_trajectory
Node: /iri_wam_reproduce_trajectory
URI: rosrpc://2_xterm:37731
Type: iri_wam_reproduce_trajectory/ExecTraj
Args: file
```

Node 信息：这个属性声明了哪个节点提供的这个服务。

Type：它代表了这个服务需要使用的 message。它的组成如下：提供 service message 的包/定义 service message 的文件。在上述的例子中，就是 iri_wam_reproduce_trajectory 这个包中的 ExecTraj 文件定义的 message。

Args：这个是调用 service 时需要传入的参数，在上例中需要传入 file 变量中的 trajectory file path。

Start_demo.launch

```
<launch>

  <include file="$(find iri_wam_reproduce_trajectory)/launch/start_service.launch"/>

  <node pkg ="iri_wam_aff_demo"
        type="iri_wam_aff_demo_node"
        name="iri_wam_aff_demo"
        output="screen">
  </node>

</launch>
```

上例的 launch 文件中的第一部分是调用了另一个叫做 start_service.launch 的 launch 文件。这个文件会提供出/execute_trajectory 服务。记住它使用一个特别的 ROS launch 文件来找到这个 package 的路径。第二部分定义了一个 node，但是是一个 cpp 编译的 node。我们在 ROS 系统中可以提供 cpp 和 Python 写我们的 ROS 程序。

我们可以手动在 console 中调用 service，使用 rosservice call /the_service_name TAB-TAB（其中 TAB-TAB 代表着快速按下 tab 键两次，它会自动为我们补全结构）

Eg：rosservice call /trajectory_by_name "traj_name: 'get_food'"

以下例子教我们怎么在 ros 脚本中调用 service 服务：

simple_service_client.py

```
#! /usr/bin/env python

import rospy
# 加载 service /trajectory_by_name 需要用到的 service message 及其对应的 Request
from trajectory_by_name_srv.srv import TrajByName, TrajByNameRequest
```

```
import sys

# 使用名字’service_client’来初始化一个 ROS node
rospy.init_node('service_client')
# 等待 service client /trajectory_by_name 开始运行
rospy.wait_for_service('/trajectory_by_name')
# 创建向着服务的连接，参数是服务名字
traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name', TrajByName)
# 创建一个 TrajByNameRequest 的请求对象
traj_by_name_object = TrajByNameRequest()
# 在请求对象中请求的参数
traj_by_name_object.traj_name = "release_food"
# 向 service 中传递参数
result = traj_by_name_service(traj_by_name_object)
# 输出 service 的结果
print(result)
```

在上例中，我们需要分清楚三个概念：service 的 package 名字，service 名字，service 使用的 service message。

1. service 的 package 名字：/trajectory_by_name_src
2. service 名字：/trajectory_by_name
3. service message 名字：/TrajByName

我们可以用过 rosservice info /name_of_the_service 来获取到 service 使用的 message。我们可以通过 rossrv show 来继续研究 TrajByName 的用法。

```
user:~/catkin_ws/src/my_examples_pkg/scripts$ rosservice info /trajectory_by_name
Node: /traj_by_name_node
URI: rosrpc://2_xterm:34753
Type: trajectory_by_name_srv/TrajByName
Args: traj_name
user:~/catkin_ws/src/my_examples_pkg/scripts$ rossrv show trajectory_by_name_srv/TrajByName
string traj_name
---
bool success
string status_message
```

上述和 topic 的对应操作似乎是很类似的。但是也有一些不同：

1. service message 文件是有扩展名.srv 形式存在的。而在 topic 中是以扩展名.msg 存在。
2. Service message 文件是在 srv 文件夹中定义的，而 topic message 是在 msg 目录中。
3. Service message 包含两个部分

```
**REQUEST**
---
**RESPONSE**
```

在 TrajByName 服务中，request 包含一个叫做 traj_name 的字符串，而 response 包含一个叫做 success 的布尔值，以及一个叫做 status_message 的字符串。

Service message 中元素的数量可以根据需求变化。不需要的话也可以设置为 None。重点是这个三个横线---，这标志着这个文件是一个 service message 文件。

**REQUEST**定义了我们如何调用这个 service，也就是 service server 用来完成 task 所需要使用的变量。

**RESPONSE**定义了 service 执行完成后该如何响应结果，比如说它可以返回一个字符

串来返回一些信息。

我们该怎么样在代码中使用 service message 呢？只要 service message 编译成功以后，3 个 message 对象就会被创建。打个比方来说，我们有一个 service message 叫做 MyServiceMessage，编译成功以后我们就会有：

1. MyServiceMessage:这是 service message 本身。它用来创建一个和 service server 的连接。Eg：traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name', TrajByName)
2. MyServiceMessageRequest:这是用来创建一个发到服务器请求的对象。和游览器使用 HTTPRequest 对象类似。

```
traj_by_name_object = TrajByNameRequest()
# Fill the variable traj_name of this object with the desired value
traj_by_name_object.traj_name = "release_food"
# Send through the connection the name of the trajectory to be executed by the robot
result = traj_by_name_service(traj_by_name_object)
```

3. MyServiceMessageResponse:这个结构是服务器返回给客户端的数据。

练习 5.1

创建一个 unit_5_service，然后调用/execute_trajectory 服务。参数是这个服务下的 config 文件夹的 txt 文件。

exercise_5_1.py

```
#! /usr/bin/env python
import rospkg
from iri_wam_reproduce_trajectory.srv import ExecTraj, ExecTrajRequest
import sys
import rospy

rospack = rospkg.RosPack()
# This rospack.get_path() works in the same way as $(find name_of_package) in the launch files.
traj = rospack.get_path('iri_wam_reproduce_trajectory') + "/config/get_food.txt"

print("traj = ", traj)

# Initialise a ROS node with the name service_client
rospy.init_node('service_client')
# Wait for the service client /trajectory_by_name to be running
rospy.wait_for_service('/execute_trajectory')
# Create the connection to the service
traj_by_name_service = rospy.ServiceProxy('/execute_trajectory', ExecTraj)
# Create an object of type TrajByNameRequest
traj_by_name_object = ExecTrajRequest()
# Fill the variable traj_name of this object with the desired value
traj_by_name_object.file = traj
# Send through the connection the name of the trajectory to be executed by the robot
```

```
result = traj_by_name_service(traj_by_name_object)
# Print the result given by the service called
print(result)
```

接下来，我们将学习如何创建一个新的 service。

simple_service_server.py

```
#! /usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse # you import the service message python classes generated from Empty.srv.



def my_callback(request):
    print("My_callback has been called")
    return EmptyResponse() # the service Response class, in this case EmptyResponse
    #return MyServiceResponse(len(request.words.split()))

rospy.init_node('service_server')
my_service = rospy.Service('/my_service', Empty , my_callback) # create the Service called my_service with the defined callback
rospy.spin() # maintain the service open.
```

然后我们可以使用命令 rosrun my_examples_pkg simple_service_server.py 在系统中启动这个 service server。此时这个 service 就可以被其他节点调用了。我们可以通过 rosservice list 找到/my_service 这个服务。

我们可以手动调用这个服务： rosservice call /my_service [tab]+[tab] ,然后在原先启动 simple_service_server.py 的 cmd。

bb8_move_in_circle_service_client.py

```
#! /usr/bin/env python

import rospkg
from std_srvs.srv import Empty, EmptyRequest, EmptyResponse # you import the service message python classes generated from Empty.srv.
import sys
import rospy



# Initialise a ROS node with the name service_client
rospy.init_node('service_client')
```

bb8_move_in_circle_service_server.py

```
#! /usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse # you import the service message python classes generated from Empty.srv.
from geometry_msgs.msg import Twist

def my_callback(request):
    print("My_callback has been called")
    twister = Twist()
    twister.linear.x = 1
    twister.angular.z = 1
```

```
# Wait for the service client /trajectory_by_name to be running
rospy.wait_for_service('/move_bb8_in_circle')
# Create the connection to the service
traj_by_name_service = rospy.ServiceProxy('/move_bb8_in_circle', Empty)
# Create an object of type TrajByNameRequest
traj_by_name_object = EmptyRequest()
# Fill the variable traj_name of this object with the desired value
# Send through the connection the name of the trajectory to be executed by the robot
result = traj_by_name_service(traj_by_name_object)
# Print the result given by the service called
print(result)
```

```
    print("Twist", twister)
    pub.publish(twister)
    return EmptyResponse() # the service Response class, in this case EmptyResponse

rospy.init_node('service_server')
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
my_service = rospy.Service('/move_bb8_in_circle', Empty , my_callback) # create the Service called my_service with the defined callback
rospy.spin() # maintain the service open.
```

我们创建两个 launch 文件，分别启动。我们就可以看见我们的 bb8 机器人在绕圈圈了。

那么我们如果想创建自定义的 service message 应该这么办呢？

1. 在 srv 目录下，添加 MyCustomServiceMessage.srv

其内容为：

```
int32 duration       # The time (in seconds) during which BB-8 will keep moving in circles
---
bool success         # Did it achieve it?
```

2. 然后我们需要修改 CMakeLists.txt

I. 在 find_package 中我们需要添加 std_msgs 和 message_generation

II. 在 add_service_files 中，我们需要添加 DEPENDENCIES std_msgs

III. 在 generate_messages 中，我们需要添加 DEPENDENCIES std_msgs

IV. 在 catkin_package 中，我们需要添加 CATKIN_DEPENDS rospy

3. 然后我们需要修改 package.xml，添加

```
<build_depend>message_generation</build_depend>
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

find_package()：所有编译 topic,service,action 的 message 过程中需要的包都定义在这里。这里写的包也需要在 package.xml 中定义在 build_depend 中。

add_service_files()：这个函数包括了一系列这个包中定义的 service message（在 srv 目录下定义）。

generate_messages()：这里是这个 service message 编译过程中需要导入的包。

catkin_package()：这里声明了包中某些地方需要使用到的一些包，需要同时在 package.xml 的<exec_depend>中声明。

custom_service_server.py

```
#! /usr/bin/env python
import rospy
from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage, MyCustomServiceMessageResponse

def my_callback(request):
    print("Request Data==> duration="+str(request.duration))
    my_response = MyCustomServiceMessageResponse()
    if request.duration > 5.0:
        my_response.success = True
    else:
        my_response.success = False
    return my_response

rospy.init_node('service_service')
my_service = rospy.Service('/my_service', MyCustomServiceMessage , my_callback)
rospy.spin()
```

call_bb8_move_custom_service_server.py

```
#! /usr/bin/env python
import rospkg
from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage, MyCustomServiceMessageRequest, MyCustomServiceMessageResponse
import sys
import rospy

rospy.init_node('service_client_call')
rospy.wait_for_service('/my_service')
my_service = rospy.ServiceProxy('/my_service', MyCustomServiceMessage)
my_service_request = MyCustomServiceMessageRequest()
my_service_request.duration = 6
result = my_service(my_service_request)
print(result)
```

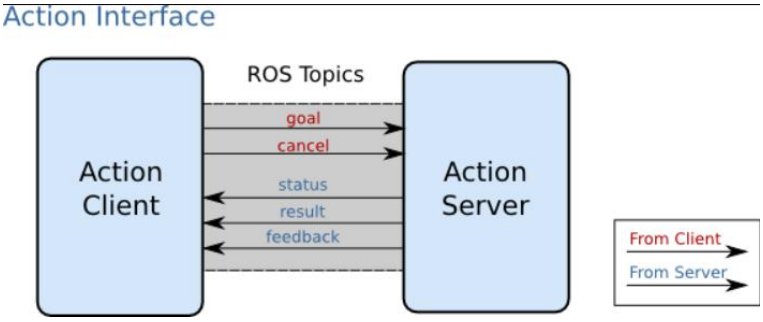
MyCustomServiceMessage.srv

```
int32 duration # the time (in seconds)
---
bool success # did it achieve it?
```

action 其实就是异步的 service。当我们调用一个 action 的时候，我们就在调用另一个节点提供的功能，和 service 一样。唯一的差别就是当我们当前 node 调用了一个 service 的时候，这个 node 必须等待 service 结束。而当前 node 调用一个 action 的时候，它不需要等待 action 执行结束。

提供功能的 node 必须包含一个 action server，这个 action server 允许其他 node 来调用它。

调用 action 的 node 必须包含一个 action client，这个 action client 能够连接到另一个 node 的 action server。




我们如果想要查看 topic，那么是 rostopic list

我们如果想要查看 service，那么可以使用 rosservice list

那么如果我们想要查看 action，是 rosaction list 吗？不是的。当我们启动 action server 后，我们使用 rostopic list 可以看到：

```
user:~$ rostopic list
/ardrone_action_server/cancel
/ardrone_action_server/feedback
/ardrone_action_server/goal
/ardrone_action_server/result
/ardrone_action_server/status
/camera_info
/clock
/cmd_vel
/drone/down_camera/image_raw
/drone/down_camera/image_raw/compressed
```

上述的这五个其实就是我们 action service 所提供的。当一个 robot 提供 action 的时候，我们可以在 topic list 中看到五个 topic，也就是 cancel, feedback, goal, result 和 status。这五个就是和 action server 通讯的五个 message。

这个例子中的 ardrone_action_server 是一个 action server，一旦我们调用它，它就会开始从无人机的前摄像头中每一秒拍一张照，持续我们指定的时间。

调用一个 action server 意味着给它发送你 message，这个是和 topics 还有 services 一致的。

topic 的 message 只包含 topic 需要提供的信息。

Service 的 message 包含 request 和 response。

action server 的 message 分成三个部分，goal，result 和 feedback。

所有的 action message 是被定义在 action 目录下.action 文件。

在本例中，我们可以看到 ardrone_as/action/Ardrone.action 如下所示：

| Ardrone.action |
| --- |
| #goal for the drone<br>int32 nseconds  # the number of seconds the drone will be taking pictures<br>---<br>#result<br>sensor_msgs/CompressedImage[] allPictures # an array containing all the pictures taken along the nseconds<br>---<br>#feedback<br>sensor_msgs/CompressedImage lastImage  # the last image taken |

我们可以看到这个 action message 被分成了三部分，goal 是一个 int32 的 ROS 标准数据

结构，可以在 std_megs 中找到。因为它是 ROS 标准包中的结构，所以不需要指定 int32 在哪个 package 中。

result 包含一个叫做 allPicture 的数组。这个 CompressedImage 在 sensor_msgs 中定义。feedback 中的 lastImage 同理。

因为调用一个 action server 并不会影响我们的线程，action server 提供了一个叫做 feedback 的 message。这个信息会被 action server 每次想要展示它想要做什么的时候创建。（告诉 caller 当前 action 的状态是什么）

要调用一个 action server，我们需要创建一个 action_client 脚本。

ardrone_action_client.py

```
#! /usr/bin/env python
import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

nImage = 1

# 定义 feedback 回调函数。这个函数会在从 action server 获取到 feedback 时被调用。
# 我们目前让它单纯地 print 出收到了 message
def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

rospy.init_node('drone_action_client')

# 创建向着 action server 的连接
client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)
# 等待 action server 被启动
client.wait_for_server()

# 创建 goal 结构
goal = ArdroneGoal()
goal.nseconds = 10
# 把 goal 发送给 action server 并且绑定 feedback 的回调函数
client.send_goal(goal, feedback_cb=feedback_callback)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()  # would cancel the goal 3 seconds after starting

# 等待获取到 result 结构
```

```
# you can do other stuff here instead of waiting
# and check for status from time to time
# status = client.get_state()
# check the client API link below for more info

client.wait_for_result()

print('[Result] State: %d'%(client.get_state()))
```

我们现在已经学会了调用 action 以及等待 result 结构。但是这个 service 一样了，我们该怎么样执行 action 的过程中还去做别的事情呢？SimpleActionClient 结构提供了两个函数来帮助我们：

1. wait_for_result()：这个函数很简单，一旦被调用，它就阻塞住等待 action 返回 result 结果。
2. get_state()：这个函数一旦被调用，它会返回一个整数，代表着 action 目前的状态是什么。

```
0 ==> PENDING
1 ==> ACTIVE
2 ==> DONE
3 ==> WARN
4 ==> ERROR
```

这允许我们创建一个 while 循环来不断 check get_state()的返回值是否大于等于 2，如果小于 2，那就说明 action 还在做。

wait_for_result_test.py

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

nImage = 1

def feedback_callback(feedback):
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1



rospy.init_node('example_with_waitforresult_action_client_node')
action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
```

```
rospy.loginfo('Action Server Found...'+action_server_name)

goal = ArdroneGoal()
goal.nseconds = 10
client.send_goal(goal, feedback_cb=feedback_callback)
# 初始化一个 rate 对象，可以设置循环的频率为 1hz
rate = rospy.Rate(1)

rospy.loginfo("Lets Start The Wait for the Action To finish Loop...")
while not client.wait_for_result():
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    rate.sleep()

rospy.loginfo("Example with WaitForResult Finished.")
```

我们会发现上述脚本根本没有进循环。Action server 返回 result 后就直接跳出了循环。

```
#! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult, ArdroneFeedback

PENDING = 0
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

def feedback_callback(feedback):
    """
    Error that might jump
    self._feedback.lastImage =    AttributeError: 'ArdroneAS' obj
    """

    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

rospy.init_node('example_no_waitforresult_action_client_node')
action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name, ArdroneAction)
```

```
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)



# You can access the SimpleAction Variable "simple_state", that will be 1 if active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2 if NO ERROR, 3 If Any Warning, and 3 if ERROR
state_result = client.get_state()
rate = rospy.Rate(1)
rospy.loginfo("state_result: "+str(state_result))

while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result))

rospy.loginfo("[Result] State: "+str(state_result))
if state_result == ERROR:
    rospy.logerr("Something went wrong in the Server Side")
if state_result == WARN:
    rospy.logwarn("There is a warning in the Server Side")

#rospy.loginfo("[Result] State: "+str(client.get_result()))
```

上述代码我们可以看到，在 action_server 还在执行逻辑的时候，我们可以在 while 中执行自己线程的逻辑。

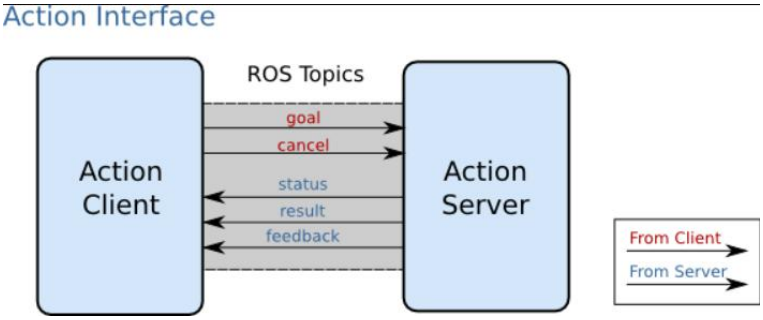
在一个 goal 的执行过程中，我们可以取消掉它，也就是 preempting a goal。我们需要使用代码 client.cancel_goal()

```
while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a result....")
    counter += 1
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result)+", counter ="+str(counter))
```

```
    if counter == 2:
        rospy.logwarn("Canceling Goal...")
        client.cancel_goal()
        rospy.logwarn("Goal Canceled")
        state_result = client.get_state()
        rospy.loginfo("Update state_result after Cancel : "+str(state_result)+", counter ="+str(counter))
```

上述代码中在 counter== 2 的时候，我们调用了 cancel_goal 函数。

```
[WARN] [1632538492.917013, 2540.425000]: Canceling Goal...
[Feedback] image n.3 received
[WARN] [1632538492.920371, 2540.428000]: Goal Canceled
[INFO] [1632538492.922596, 2540.429000]: Update state_result after Cancel : 1, counter =2
[INFO] [1632538492.923984, 2540.431000]: Doing Stuff while waiting for the Server to give a result..
..
[INFO] [1632538494.136153, 2541.425000]: state_result: 1, counter =3
[INFO] [1632538494.138501, 2541.426000]: Doing Stuff while waiting for the Server to give a result..
..
[INFO] [1632538495.298177, 2542.425000]: state_result: 2, counter =4
```



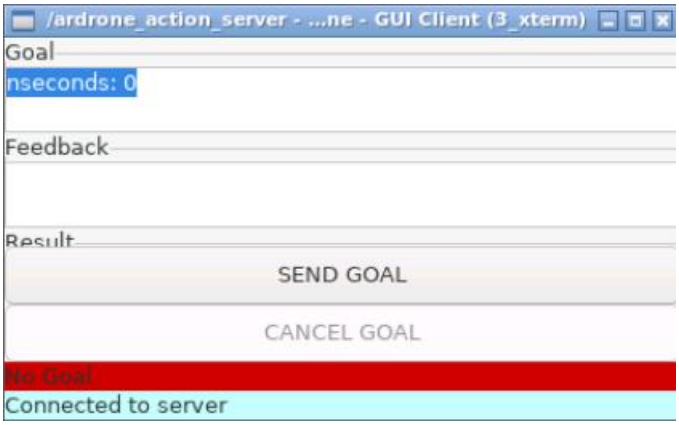

此时我们回顾上图就清晰了，action_client 调用 action_server 的时候是通过传递一个 goal 到/ardrone_action_server/goal 话题中。当 action_server 开始执行 goal 的时候，它会通过/ardrone_action_server/feedback 话题来传递 feedback 数据。当 action_server 结束 goal 的时候它会通过/ardrone_action_server/result 来传递 result 对象。

goal,feedback,result 都有它们自己种类的 message。这个 message 是 ROS 自动帮我们从.action 文件中构建的。比如说，在上例中，我们有一个 Ardrone.action，在编译的时候，ROS 会自动帮我们创建三个数据结构：

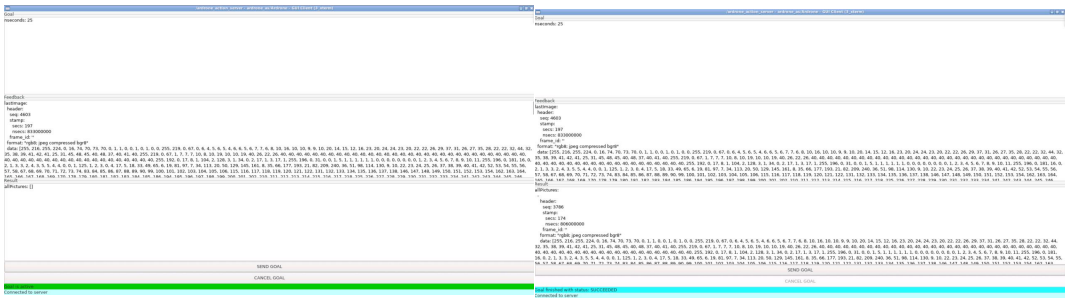
ArdroneActionGoal,ArdroneActionFeedback,ArdroneActionResult

我们也可以直接在命令行中调用 action，通过命令：rostopic pub /[name_of_action_server]/goal /[type_of_the_message_used_by_the_topic] [TAB][TAB]。其中 tab-tab 会帮我们自动补全格式

到目前为止，我们学习了如何通过 python 代码和命令行来调用 action server。其实还有一种方法来把 goal 传递给 action server。也就是使用 axclient。它是一个 actionlib 中提供的一个 GUI 工具，它提供了一个和 action server 交互的便捷、可视化的方式。我们可以使用命令 rosrun actionlib_tools axclient.py /<name_of_action_server>来启动




我们可以看到这样的一个界面。



我们可以传递不同的 goal 参数进去，它上面会实时显示 feedback 数据和最终的 reuslt 数据。

fibonacci_action_server.py

```
#! /usr/bin/env python
import rospy
import actionlib
from actionlib_tutorials.msg import FibonacciFeedback, FibonacciResult, FibonacciAction

class FibonacciClass(object):
  # 创建 Feedback message 和 Result message
  _feedback = FibonacciFeedback()
  _result   = FibonacciResult()

  def __init__(self):
    # 创建 action server
    self._as = actionlib.SimpleActionServer("fibonacci_as", FibonacciAction, self.goal_callback, False)
    self._as.start()

  def goal_callback(self, goal):
    # 一旦 action server 被调用，就会调用这个回调函数，这个函数计算斐波那契数列并且返回给调用 action server 的 node
    r = rospy.Rate(1)
    success = True
```

```
        # 准备斐波那契数列
        self._feedback.sequence = []
        self._feedback.sequence.append(0)
        self._feedback.sequence.append(1)

        # 把信息输出到 console 里
        rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence of order %i with seeds %i, %i' % ( goal.order, self._feedback.sequence[0], self._feedback.sequence[1]))

        # 开始计算斐波那契数列
        fibonacciOrder = goal.order
        for i in range(1, fibonacciOrder):

            # 检查 action client 有没有调用 cancel_action
            if self._as.is_preempt_requested():
                rospy.loginfo('The goal has been cancelled/preempted')
                # 取消 goal
                self._as.set_preempted()
                success = False
                # 结束计算斐波那契数列
                break

            # 添加进 feedback.sequence 中
            self._feedback.sequence.append(self._feedback.sequence[i] + self._feedback.sequence[i-1])
            # 发布 feedback
            self._as.publish_feedback(self._feedback)
            # 每秒计算一次
            r.sleep()
        # 到这里以后，要么达到了 goal(SUCCESS)，要么提前取消了(SUCCESS==FALSE)
        # 如果是 success，我们就发布 result
        # 如果提前终止了，我们就不发布 result
        if success:
            self._result.sequence = self._feedback.sequence
            rospy.loginfo('Succeeded calculating the Fibonacci of order %i' % fibonacciOrder )
            self._as.set_succeeded(self._result)

if __name__ == '__main__':
    rospy.init_node('fibonacci')
    FibonacciClass()
    rospy.spin()
```

以上定义了一个 action server，可以通过调用 action “fibonacci_as”来进行，它会使用 FibonacciAction，其定义如下：

```
user:/opt/ros/noetic/share/actionlib_tutorials/action$ cat Fibonacci.action
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence
```

我们在一个窗口启动 action server： rosrun my_examples_pkg fibonacci_action_server.py

并且在另一个窗口启动监听 rostopic echo /fibonacci_as/feedback

然后在第三个窗口手动向着/fibonacci_as/goal 发布一个 goal，注意下图的绿框是我们 tab+tab 自动补全的，而 order=999 是我们自己加的，也就是要求这个 action server 计算前 999 项 fibonacci 数列的值。

```
user:~$ rostopic pub /fibonacci_as/goal actionlib_tutorials/FibonacciActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
  order: 999"
```

我们在监听 feedback 的命令行中可以看到，sequence 在不断地增长，刷新这个 feedback topic 中的值。

```
---
header:
  seq: 31
  stamp:
    secs: 1121
    nsecs: 731000000
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1091
      nsecs: 731000000
    id: "/fibonacci-1-1091.731000000"
  status: 1
  text: "This goal has been accepted by the simple action server"
feedback:
  sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 1
21393, 196418, 317811, 514229, 832040, 1346269, 2178309]
---
```

我们必须记住 message(Fibonacci.action) 在 python 中的类叫做 FibonacciGoal, FibonacciResult, FibonacciFeedback。而在 topic 中这些 message 叫做 FibonacciActionGoal, FibonacciActionResult 和 FibonacciActionFeedback

通常建议使用 ROS 自带的 action message，但是我们如果想要定义自己的怎么办呢？

1. 在 package 下创建 action 目录
2. 创建 Name.action 文件。注意这个 action message file 的名字会影响我们在 action server 中使用的 class 的名字。ROS 的规范是使用驼峰法。

| Name.action |
| --- |
| #goal<br>package_where_message_is/message_type goal_var_name<br>---<br>#result |

```
package_where_message_is/message_type result_var_name
---
#feedback
package_where_message_is/message_type feedback_var_name
```

如果我们的 action 不需要提供 feedback，那么对应位置也可以留空，但是短横线是不能省略的。

3. 修改 CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_action_msg_pkg)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
   std_msgs
   actionlib_msgs
)

## Generate actions in the 'action' folder
add_action_files(
     FILES
     Name.action
  )

## Generate added messages and services with any dependencies listed here
generate_messages(
     DEPENDENCIES
     std_msgs actionlib_msgs
  )

catkin_package(
  CATKIN_DEPENDS rospy
)

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
   ${catkin_INCLUDE_DIRS}
)
```

4. 修改 package.xml

```
<?xml version="1.0"?>
```

```
<package format="2">
  <name>my_custom_action_msg_pkg</name>
  <version>0.0.0</version>
  <description>The my_custom_action_msg_pkg package</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>actionlib</build_export_depend>
  <build_export_depend>actionlib_msgs</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>actionlib</exec_depend>
  <exec_depend>actionlib_msgs</exec_depend>
  <exec_depend>rospy</exec_depend>

  <export>
  </export>
</package>
```

一旦我们都设置完毕了，我们可以通过 rosmsg list | grep Name 看到如下情况：

```
my_custom_action_msg_pkg/NameAction
my_custom_action_msg_pkg/NameActionFeedback
my_custom_action_msg_pkg/NameActionGoal
my_custom_action_msg_pkg/NameActionResult
my_custom_action_msg_pkg/NameFeedback
my_custom_action_msg_pkg/NameGoal
my_custom_action_msg_pkg/NameResult
```

无人机起飞 quiz

my_script.py

```
#! /usr/bin/env python
import rospy
import actionlib
from std_msgs.msg import Empty
from actions_quiz.msg import CustomActionMsgResult, CustomActionMsgFeedback, CustomActionMsgAction

#创建一个 ardrone action server
```

```
class Ardrone_Action_Server(object):
    _feedback = CustomActionMsgFeedback()
    def __init__(self):
        # creates the action server
        self._as = actionlib.SimpleActionServer("/action_custom_msg_as", CustomActionMsgAction,
self.goal_callback, False)
        self._as.start()



    def goal_callback(self, goal):
        print("goal = ", goal)
        action_string = goal.goal

        # 根据 goal 的不同值来决定是起飞还是降落
        if action_string == "TAKEOFF":
            pub = rospy.Publisher('/drone/takeoff', Empty, queue_size=1)
            pub.publish(Empty())
        if action_string == "LAND":
            pub = rospy.Publisher('/drone/land', Empty, queue_size=1)
            pub.publish(Empty())

        r = rospy.Rate(1)
        success = True
        self._feedback.feedback = action_string
        self._as.set_succeeded(True)

if __name__ == '__main__':
    rospy.init_node('actions_quiz')
    Ardrone_Action_Server()
    rospy.spin()
```

CustomActionMsg.action

```
string goal
---
---
string feedback
```

机器人中最难的一部分就是这么把我们的 idea 转化为真正的项目。通常会和理论上表现的不一致。现实会更加复杂，我们需要工具来知道发生了什么以及找到对应的问题。这就是为什么 debug 工具和可视化工具在机器人学中非常重要，尤其我们的机器人和图像、激光雷达、点云、动力学数据一起工作的时候。

我们在这门课中已经学会使用 print()来查看我们的程序工作的怎么样。我们接下来学习 logs，它允许我们在 screen 中打印出来，也允许我们存储在 ROS 框架中。在 log 系统中，通常存在 log 等级。有五个 log 等级，每个等级包含了更深的等级。如果我们使用 error，那么会显示所有 error 和 fatal 等级的 log。如果我们的等级是 warning，那么会显示所有 warning，error 和 fatal 等级的日志。

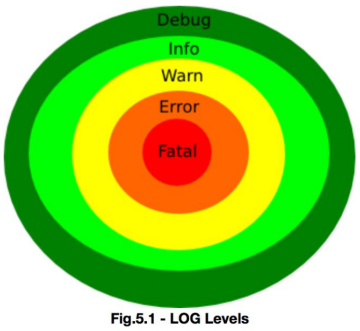


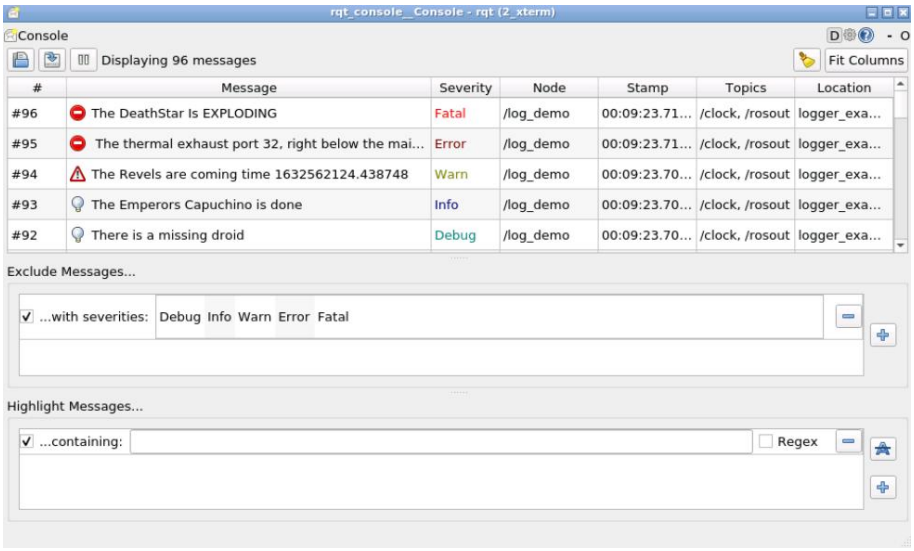
Fig.5.1 - LOG Levels

**Use the Python module *rospy* to access the logging functionality in Python.**
**DEBUG ==> rospy.logdebug(msg, *args)**
***INFO* ==> *rospy.loginfo(msg,* args)**
**WARNING ==> rospy.logwarn(msg, *args)**
***ERROR* ==> *rospy.logerr(msg,* args)**
**FATAL ==> rospy.logfatal(msg, *args)**

logger_example.py

```
#! /usr/bin/env python

import rospy
import random
import time

# Options: DEBUG, INFO, WARN, ERROR, FATAL
rospy.init_node('log_demo', log_level=rospy.DEBUG)
rate = rospy.Rate(0.5)

#rospy.loginfo_throttle(120, "DeathStars Minute info: "+str(time.time()))

while not rospy.is_shutdown():
    rospy.logdebug("There is a missing droid")
    rospy.loginfo("The Emperors Capuchino is done")
    rospy.logwarn("The Revels are coming time "+str(time.time()))
    exhaust_number = random.randint(1,100)
    port_number = random.randint(1,100)
    rospy.logerr(" The thermal exhaust port %s, right below the main port %s", exhaust_number,
port_number)
    rospy.logfatal("The DeathStar Is EXPLODING")
    rate.sleep()
    rospy.logfatal("END")
```

我们可以通过 topic 中的/rosout 来阅读我们所有记录的日志。但是我们会发现，当节点越来越多的时候，会有大量的日志出现，导致我们来不及阅读。这时候就出现了 rqt_console。




The rqt_console window is divided into three subpanels.

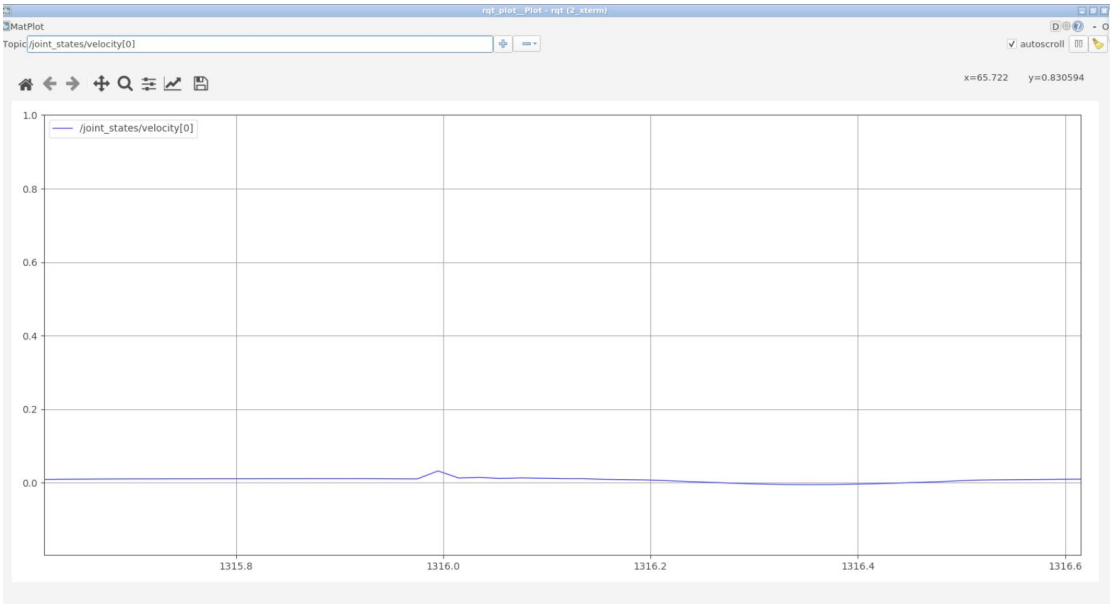
rqt_console 窗口分成三个子窗口。第一个子窗口输出日志，它有日志信息、日志等级、记录日志的节点和其他信息，我们可以从这里提取日志信息。

第二个子窗口允许我们过滤第一个子窗口的日志信息，可以通过 node、日志等级和包含字来进行过滤。为了添加一个过滤，只需要添加右边的+号即可。

第三个子窗口允许我们高亮置顶的 message，我们可以改变展示 message 的数量。

在 robotics 中，我们需要知道比如我们的速度是正确的，关节的扭矩是正常的，或者我们的激光雷达读取到的数据的正常的。对于这些数据，我们需要绘图工具来 plot 出来。我们可以使用命令 roslaunch iri_wam_aff_demo start_demo.launch 来运行我们的机械臂，然后我们就可以通过话题 rostopic echo /joint_states -n1 知道我们正在运动的机械臂的关节属性。

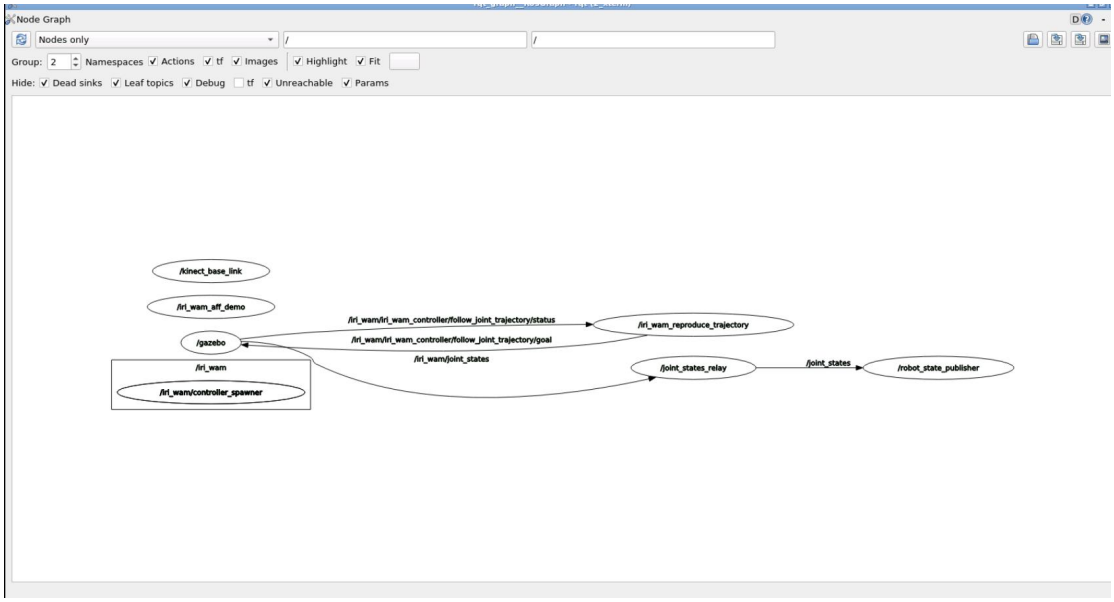
我们可以使用命令 rqt_plot 打开绘图窗口。




在左上角输入 Topic: /joint_states/velocity[0]即可绘制出实时的速度曲线。

我们为了查看我们的 nodes 是否以正确的方式连接，以及定位为什么没有从某个 topic

收到消息。我们可以使用 rqt_graph 来定位这些问题。它展示一个节点连接的图结构以及对应的 connection 连接。




在上图中，我们可以看到当前正在运行的 node，以及被 topic 的通讯关系所联系。我们可以使用刷新按钮来刷新 nodes，也可以通过 filtering 选型来过滤 node 和 topics。

在 robotics 中很常见的场景就是，我们有一个很贵的现实中的机器人，比如说是 R2-D2 在一个很难过去的地方。我们希望复现出相同的情况来复现出 R2-D2 开门的算法。我们既没有那个地方的条件，我们手头也没有 R2-D2 机器人。我们该怎么样获取到完全一致的传感器数据呢？我们可以记录下来。这就是 rosbag 帮助我们做的事情。它记录所有在 ROS 话题系统中传递的数据，并且允许我们重放它们。

使用 rosbag 播放的命令是

1. 记录我们希望的 topic 的数据： rosbag record -O name_bag_file.bag name_topic_to_record1 …… name_topic_to_recordN

2. 提取我们记录的数据的信息：rosbag info name_bag_file.bag

3. 重放我们记录的数据：rosbag play name_bag_file.bag

重放数据的时候，robag 会向相同的 topic 输出完全一样的数据在相同的时间。

RVIZ 是一个工具，它允许我们可视化图像、点云、激光雷达、动力学矩阵、机器人模型。你甚至可以定义自己的标记。RVIZ 不是一个仿真环境。RVIZ 展示了发布到 topic 的数据。RVIZ 是一个非常复杂的工具，它需要你花一节课的时间去学习。

1- Type in WebShell #2 the following command:

RVIZ is NOT a simulation. I repeat: It's NOT a simulation.

RVIZ is a representation of what is being published in the topics, by the simulation or the real robot.

RVIZ is a really complex tool and it would take you a whole course just to master it. Here, you will get a glimpse of what it can give you.

Remember that you should have unpaused the simulations and stopped the rosbag as described

in the rosbag section.

1- Type in WebShell #2 the following command:

Execute in Shell #2

In [ ]:

rosrun rviz rviz

2- Then go to the graphical interface to see the RVIZ GUI:

You will be greeted by a window like {Fig-10.9}:

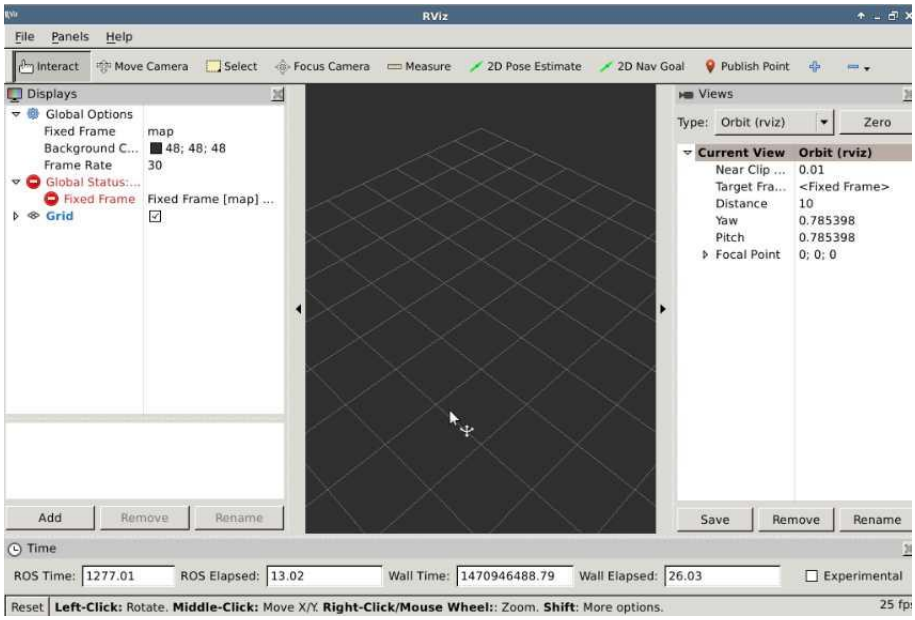



Fig-10.9 - RVIZ Starting Window

NOTE: In case you don't see the lower part of Rviz (the Add button, etc.), double-click at the top of the window to maximize it. Then you'll see it properly.

You need only to be concerned about a few elements to start enjoying RVIZ.

Central Panel: Here is where all the magic happens. Here is where the data will be shown. It's a 3D space that you can rotate (LEFT-CLICK PRESSED), translate (CENTER MOUSE BUTTON PRESSED) and zoom in/out (RIGHT-CLICK PRESSED).

Left Displays Panel: Here is where you manage/configure all the elements that you wish to visualize in the central panel. You only need to use two elements:

In Global Options, you have to select the Fixed Frame that suits you for the visualization of the data. It is the reference frame from which all the data will be referred to.

The Add button. Clicking here you get all of the types of elements that can be represented in RVIZ.

Go to RVIZ in the graphical interface and add a TF element. For that, click "Add" and select the element TF in the list of elements provided, as shown in {Fig-10.10}.

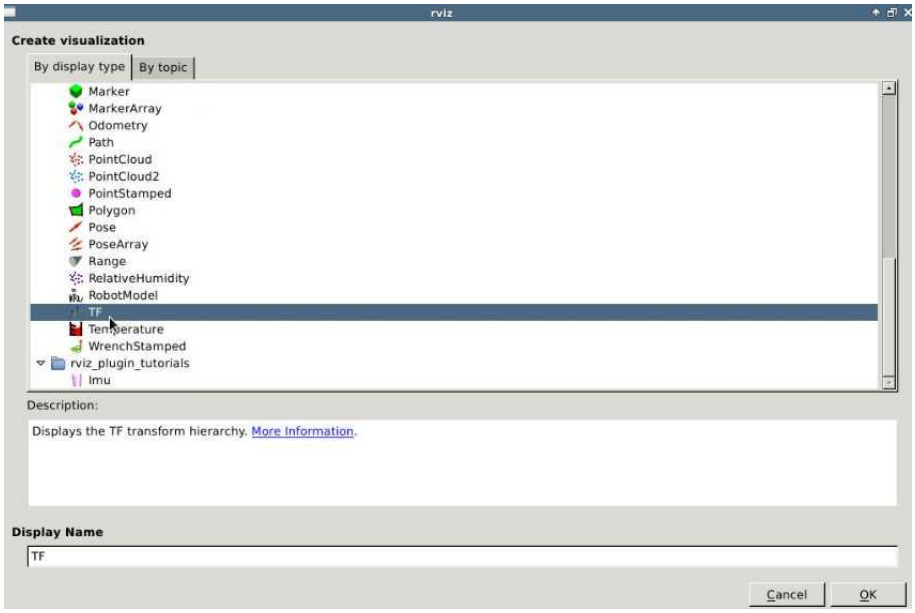



Fig-10.10 - RVIZ Add element

Go to the RVIZ Left panel, select as Fixed Frame the iri_wam_link_footprint and make sure that the TF element checkbox is checked. In a few moments, you should see all of the Robots Elements Axis represented in the CENTRAL Panel.

Now, go to a WebShell #1 and enter the command to move the robot:

roslaunch iri_wam_aff_demo start_demo.launch
You should see something like this:

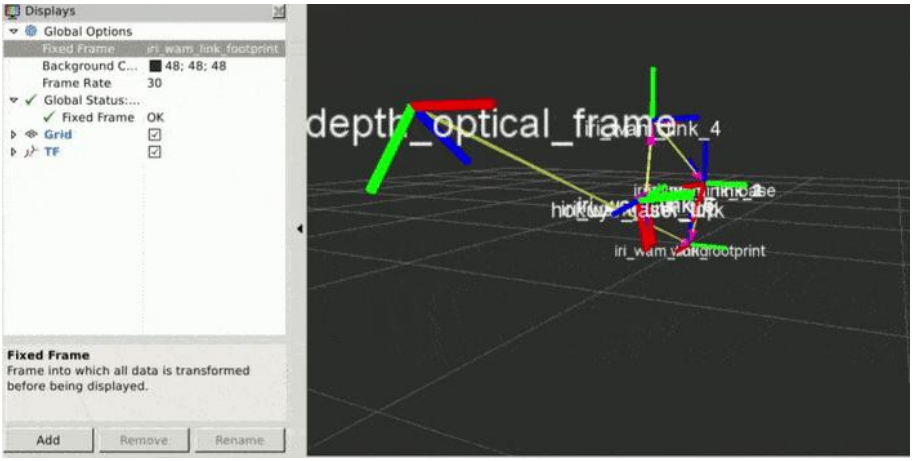



Fig-10.11 - RVIZ TF

In {Fig-10.11}, you are seeing all of the transformations elements of the IRI Wam Simulation in real-time. This allows you to see exactly what joint transformations are sent to the robot arm to check if it's working properly.

Now press "Add" and select RobotModel, as shown in {Fig-10.12}

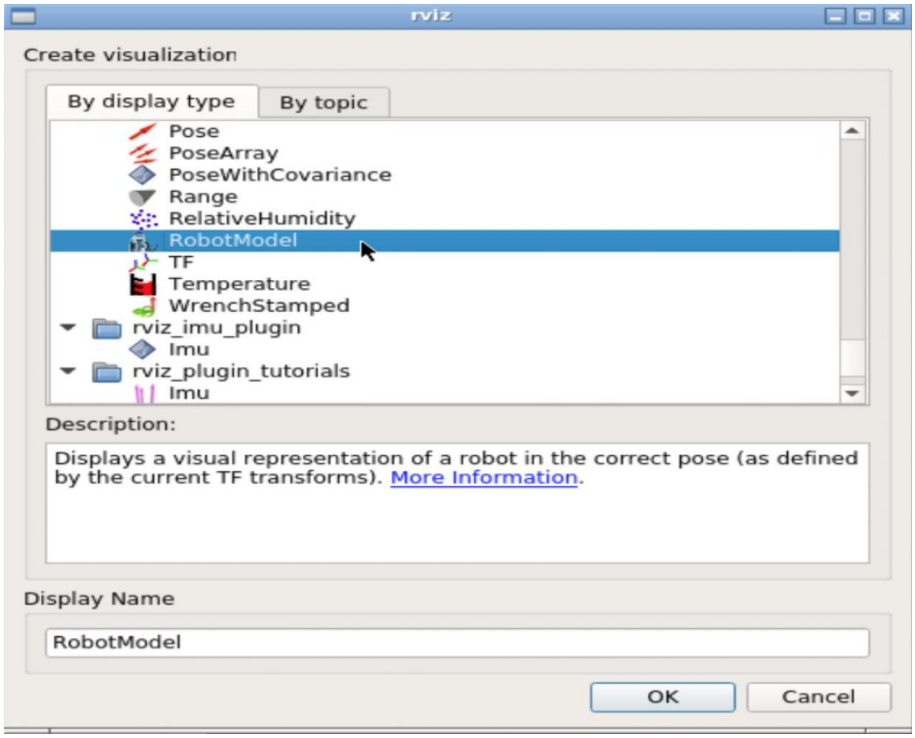



Fig-10.12 - RVIZ Add Robot Model

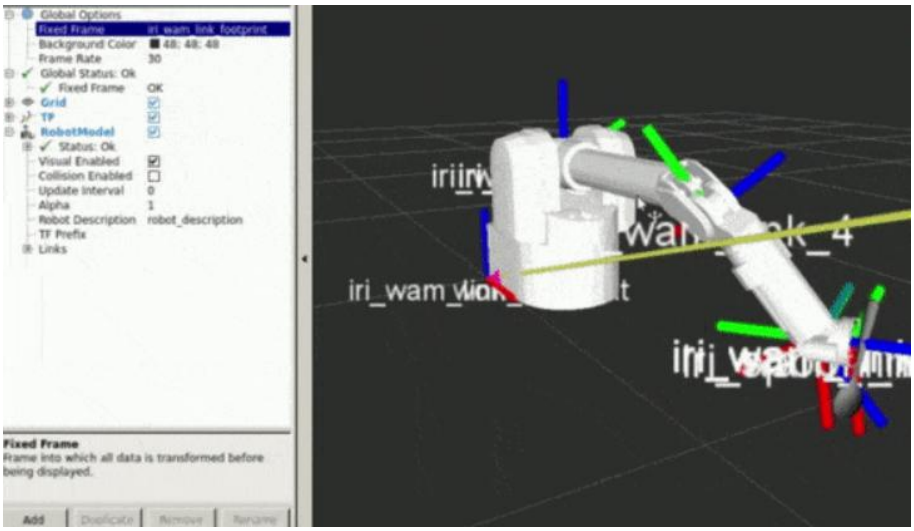
You should see now the 3D model of the robot, as shown in {Fig-10.13}:

Fig-10.13 - RVIZ Robot Model + TF

Why can't you see the table? Or the bowl? Is there something wrong? Not at all!

Remember: RVIZ is NOT a simulation, it represents what the TOPICS are publishing. In this case the models that are represented are the ones that the RobotStatePublisher node is publishing in some ROS topics. There is NO node publishing about the bowl or the table.

Then how can you see the object around? Just like the robot does, through cameras, lasers, and other topic data.



Remember: RVIZ shows what your robot is perceiving, nothing else.

- Exercise 10.4 -

Add to RVIZ the visualization of the following elements:

What the RGB camera from the Kinect is seeing. TIP: The topic it has to read is /camera/rgb/image_raw. It might take a while to load the images, so just be patient.

What the Laser mounted at the end effector of the robot arm is registering. TIP: You can adjust the appearance of the laser points through the element in the LEFT PANEL.

What the PointCloud Camera / Kinect mounted in front of the robot arm is registering. TIP: You can adjust the appearance of the pointcloud points through the element in the LEFT PANEL. You should select points for better performance.

TIP: You should have a similar result as the one depicted beneath:

Notice that activating the pointcloud has a huge impact on the system performance. This is due to the huge quantity of data being represented. It's highly recommended to only use it with a high-end graphics card.

Play around with the type of representations of the laser, size, and so on, as well as with the pointcloud configuration.