目录

[1 ROS简介 2](#_Toc26606731)

[1.1 ROS资源 3](#_Toc26606732)

[1.2 ROS的目的 4](#_Toc26606733)

[1.3 ROS的生态系统 5](#_Toc26606734)

[1.4 版本周期 5](#_Toc26606735)

[2. ROS开发环境安装 6](#_Toc26606736)

[2.1 安装ROS 6](#_Toc26606737)

[3. ROS重要概念 7](#_Toc26606738)

[3.1 ROS: 7](#_Toc26606739)

[主节点(Ros Master)： 7](#_Toc26606740)

[节点(Node)： 8](#_Toc26606741)

[话题(Topic) 9](#_Toc26606742)

[发布者(Publisher) 10](#_Toc26606743)

[订阅者(Subscriber) 10](#_Toc26606744)

[服务(service) 11](#_Toc26606745)

[动作(Action) 12](#_Toc26606746)

[参数(Parameter) 13](#_Toc26606747)

[Computer Graphs 13](#_Toc26606748)

[4.ROS命令 14](#_Toc26606749)

[1: 配置环境 14](#_Toc26606750)

[2. Starting the Master process 14](#_Toc26606751)

[3.running Turtlesim Nodes 15](#_Toc26606752)

[4.Listing all Active Nodes 15](#_Toc26606753)

[5. Listing All Topics 15](#_Toc26606754)

[6. Get Information About a Specific Topic 16](#_Toc26606755)

[7.Show Message Information 16](#_Toc26606756)

[8.Echo Messages on a Topic 17](#_Toc26606757)

[Lesson5. Packages & Catkin Workspaces 17](#_Toc26606758)

[1.Create a Catkin Workspace 17](#_Toc26606759)

[2. Add a package 18](#_Toc26606760)

[Step1: Cloning the simple\_arm Package 18](#_Toc26606761)

[Step2: Building the simple\_arm package 18](#_Toc26606762)

[step3: launch multiple nodes 18](#_Toc26606763)

[3.rosdep 自动检查依赖库 19](#_Toc26606764)

[6 Writing ROS Nodes 21](#_Toc26606765)

[6.0 add a script 21](#_Toc26606766)

[6.1 Add a Publisher 21](#_Toc26606767)

[6.2 ROS Services 24](#_Toc26606768)

[6.2.1 Defining services 24](#_Toc26606769)

[6.2.2 Using Services 24](#_Toc26606770)

[6.3: add a service 25](#_Toc26606771)

[6.4 add Subscribers 34](#_Toc26606772)

[Step1: Creating the empty look\_away node scrip 35](#_Toc26606773)

[step2: Updating the launch file 36](#_Toc26606774)

[7. log信息 40](#_Toc26606775)

[7.1 Logging levels and outputs 40](#_Toc26606776)

[7.2 Filtering and saving log messages from /rosout 41](#_Toc26606777)

[7.3 Modifying message level sent to /rosout 41](#_Toc26606778)

[7.5 Modifying display of messages sent to stdout and stderr 41](#_Toc26606779)

# 1 ROS简介

ROS (Robot Operating System)

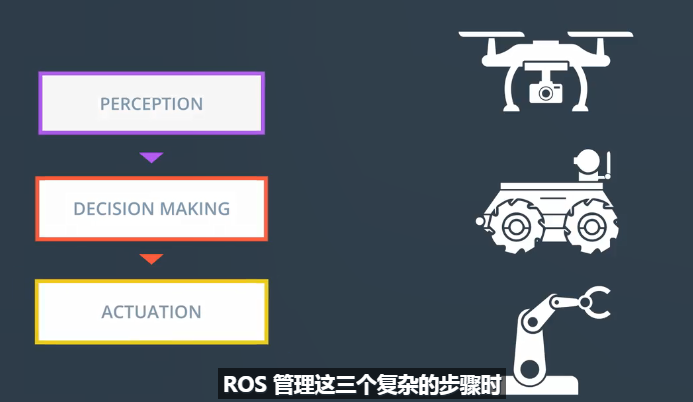
provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

来自<https://wiki.ros.org/>

ROS是一个机器人操作系统，提供一系列库和工具，帮助软件开发者创建机器人应用。它提供硬件抽象，设备驱动，函数库，可视化工具，消息传递和包管理等诸多功能。

更确切的说，ROS是一个元操作系统，不同于传统的Windows，Linux，iOS，Android。它基于现有的操作系统，比如官方维护的ROS安装在Ubuntu之上，以使用进程管理，文件系统，用户界面等。以库的形式提供了机器人应用程序所需的多数不同类型的硬件之间的数据传输，调度和错误处理等功能。这个该你啊你亦可称为中间件(Middleware)或软件框架(Software framework)。

ROS开发、管理和提供基于元操作系统的各种用途的应用功能包，并拥有一个负责分享用户所开发的功能包的生态系统(Ecosystem)。ROS在使用现有传统操作系统同时，通过使用硬件抽象概念来控制机器人应用所必须的机器人和传感器，同时也是开发用户的机器人应用的支持系统。

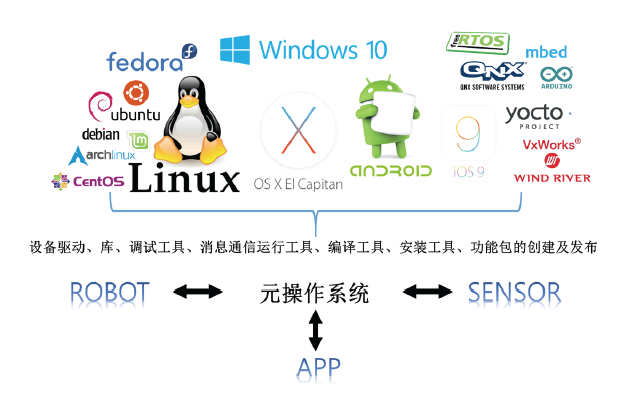


## 1.1 ROS资源

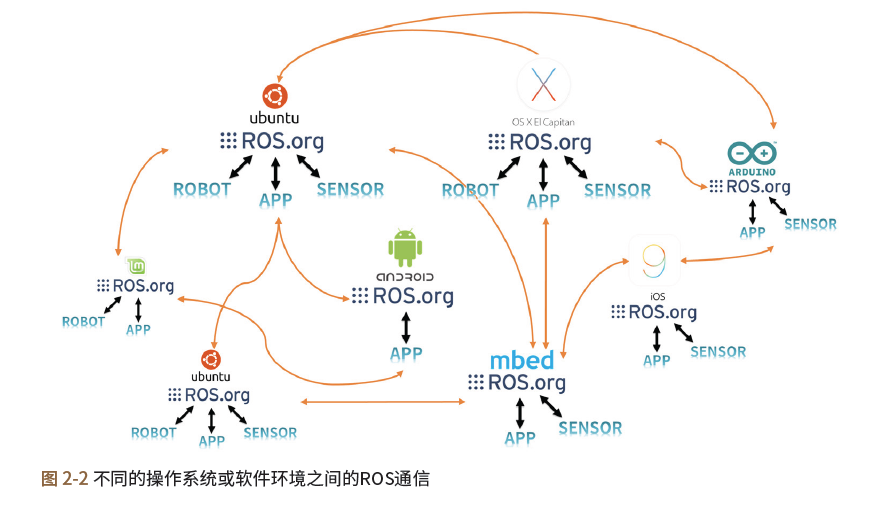
官网：<https://www.ros.org/>

ROS Wiki是所有ROS软件包的官方文档来源；<https://wiki.ros.org/>

ROS Answers是解答疑惑的第二个最好选择，包含数以万计的问题解答。



PS:ROS数据通信可以在一个OS中进行，但也适用于使用多种硬件开发的机器人，因为可以在不同的操作系统、硬件和程序之间交换数据。



## 1.2 ROS的目的

建立一个在全球范围内协作开发机器人软件的环境！

ROS致力于将机器人研究和开发中的代码重用做到最大化。为了支持这个，ROS具有以下特征。

分布式进程：它以可执行进程的最小单位（节点，Node）的形式进行编程，每个进程独立运行，并有机地收发数据。

功能包单位管理：由于它以功能包的形式管理着多个具有相同目的的进程，所以开发和使用起来很容易，并且很容易共享、修改和重新发布。

公共存储库：每个功能包都将其功能包公开给开发人员首选的公共存储库（例如GitHub），并标识许可证。

API类型：使用ROS开发程序时，ROS被设计为可以简单地通过调用API将其加载到其使用的代码中。在源代码中，您会发现ROS编程与C++和Python程序没有区别。

支持多种编程语言：ROS程序提供客户端库（Client Library）2以支持各种语言。它可以用于如JAVA、C#、Lua和Ruby等语言，也可以用于机器人中常用的编程语言，如Python、C++和Lisp。换句话说，您可以使用熟悉的语言开发ROS程序。

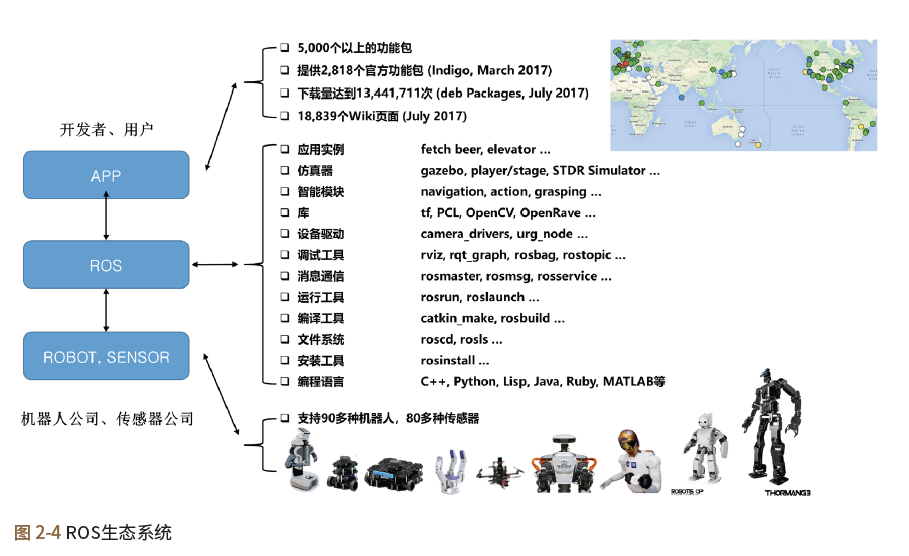
这种支持使得在全球范围内开发机器人软件的合作成为可能，并且ROS的终极目的-机

器人研究和开发过程中的代码重用-变得越来越普遍。

## 1.3 ROS的生态系统

希望机器人硬件领域的开发者、ROS开发运营团队、应用软件开发者以及用户也能像机

器人公司和传感器公司一样从中受益。



## 1.4 版本周期

ROS的版本周期和ROS正式支持的操作系统Ubuntu的新版本发布周期一样。

是每年发布两次（4月和10月），但2013年很多用户由于频繁的升级，提议新的版本周期。

因此在2013年接纳了用户们的意见，决定从Hydro Medusa版本开始，每年发布一次正式版

本，时间点则在新的Ubuntu xx.04版本发布一个月之后的5月份。而恰好每年的5月23日

是世界乌龟日（World Turtle Day）19，所以现在每年的这个有象征意义的一天会发布新

的ROS版本。

ROS的支持期间根据版本而不同，一般是发布后提供两年的支持。而针对每两年发

布的Ubuntu长期支持版（Long Term Support）20 发布的ROS版本提供与LTS一样的5

年的支持期间。

比如，支持2016年的Ubuntu 16.04 LTS的ROS Kinetic Kame具有直到

2021年5月的5年的支持期间。非LTS版本保持最新版本的Linux核，仅仅是进行了次要的

升级以及基本的维护。

在2018年发布的新的Linux版本和ROS LTS版本达到稳定的2019年之前我想推荐的组

合如下：

■■ 操作系统：Ubuntu 16.04 Xenial Xerus23 (LTS) 或 Linux Mint 18.x

■■ ROS版本：ROS Kinetic Kame24

# 2. ROS开发环境安装

## 2.1 安装ROS

为省时间，先用udacity的虚拟机环境；

# 3. ROS重要概念

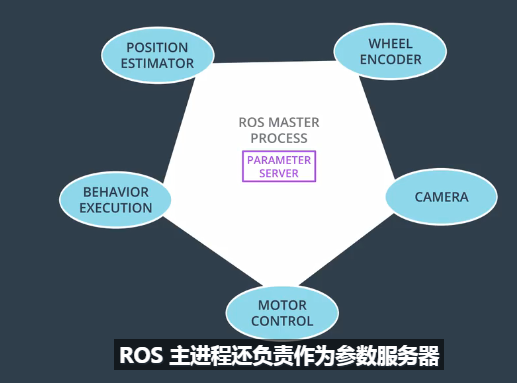
## 3.1 ROS:

表示整个机器人开发生态，提供共享库，消息解析，功能包管理，丰富开发和调试工具。

## 主节点(Ros Master)：

负责节点到节点的链接和消息通信，类似于名称服务器(Name Server)。Rescore就是运行它，当运行主节点时，可以注册每个节点的名字，并根据需要获取信息。

没有主节点，就不能在节点之间建立访问和消息通信(如Topic和server)



当启动ROS时，主节点将获取用户设置的ROS\_MASTER\_URI变量中列出的URI地址和端口。除非另外设置，默认情况下，URI地址使用当前的本地IP，端口使用11311。

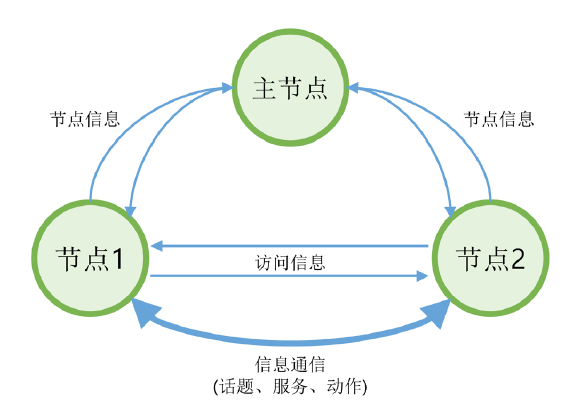


## 节点(Node)：

Node是指ROS中运行的最小处理单元。在ROS中，建议为一个目的创建一个节点，设计时注重可重用性。

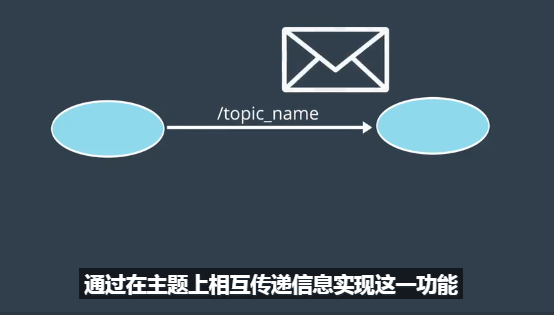
节点运行的同时，向主节点注册节点名称，并且还注册发布者(publisher)/订阅者(subscriber)，服务服务器(service server)、服务客户端(service client),且注册消息形式，URI地址和端口。

每个节点可以使用话题和服务与其他节点交换消息。



## 话题(Topic)

Publisher向主节点注册topic之后，它以消息的形式发布该topic的广播，subscriber节点获得在主节点中以这个topic注册的，那个发布者节点的信息。



## 发布者(Publisher)



发布（publish）是指以与话题的内容对应的消息的形式发送数据。

为了执行发布，发布者（publisher）节点在主节点上注册自己的话题等多种信息，并向希望订阅的订阅者节点发送消息。发布者在节点中声明自己是执行发布的个体。单个节点可以成为多个发布者。

## 订阅者(Subscriber)

订阅是指以与话题内容对应的消息的形式接收数据。

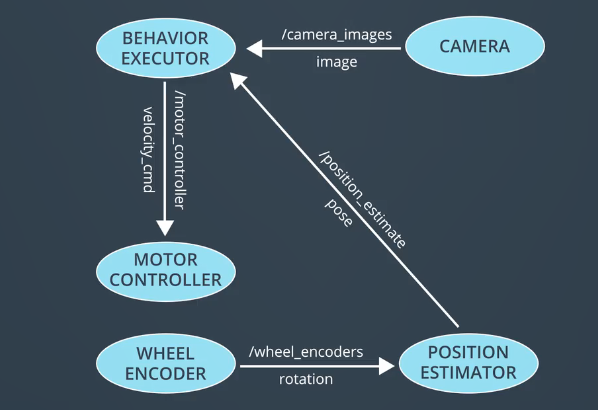
为了执行订阅，订阅者节点在主节点上注册自己的话题等多种信息，并从主节点接收那些发布此节点要订阅的话题的发布者节点的信息。基于这个信息，订阅者节点直接联系发布者节点来接收消息。

单个节点可以成为多个订阅者。

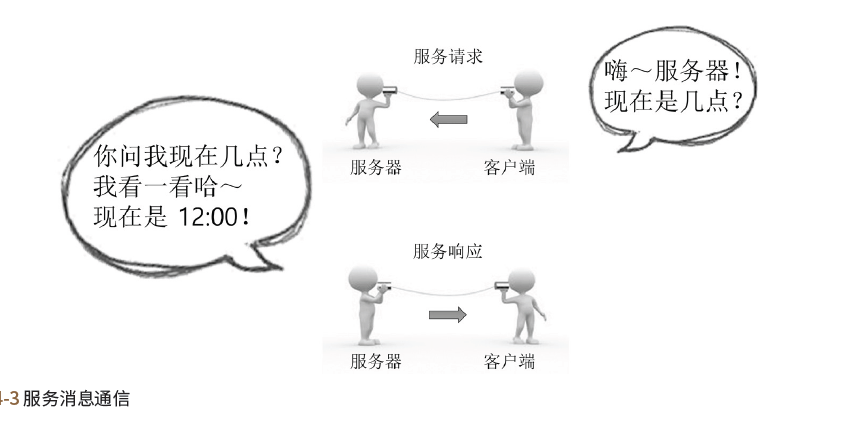
发布和订阅概念中的话题是异步的，这是一种根据需要发送和接收数据的好方法。另外，由于它通过一次的连接，发送和接收连续的消息，所以它经常被用于必须连续发送消息的传感器数据。

然而，在某些情况下，需要一种共同使用请求和响应的同步消息交换方

案。因此，ROS提供叫做服务（service）的消息同步方法。服务分为响应请求的服务服务器和请求后接收响应的服务客户端。与话题不同，服务是一次性的消息通信。当服务的请求和响应完成时，两个节点的连接被断开。



## 服务(service)



服务（service）消息通信是服务客户端（service client）与服务服务器（service server）之间的同步双向消息通信。其中服务客户端请求对应于特定目的任务的服务，而服务服务器则负责服务响应。

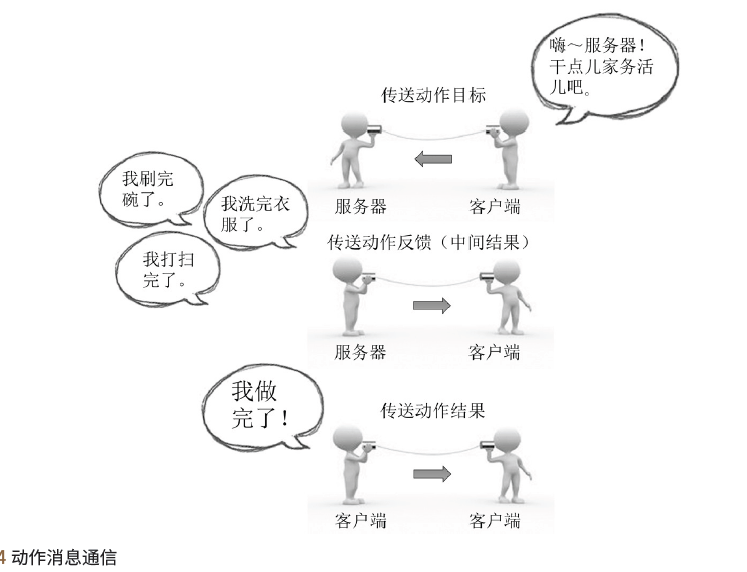
服务服务器

服务服务器（service server）是以请求作为输入，以响应作为输出的服务消息通信的服务器。请求和响应都是消息，服务器收到服务请求后，执行指定的服务，并将结果下发给服务客户端。服务服务器用于执行指定命令的节点。

服务客户端

服务客户端（service client）是以请求作为输出并以响应作为输入的服务消息通信的客户端。请求和响应都是消息，并发送服务请求到服务服务器后接收其结果。服务客户端用于传达给定命令并接收结果值的节点。

## 动作(Action)



动作（action）是在需要像服务那样的双向请求的情况下使用的消息通信方式，不同点是在处理请求之后需要很长的响应，并且需要中途反馈值。动作文件也非常类似于服务，目标（goal）和结果（result）对应于请求和响应。此外，还添加了对应于中途的反馈（feedback）。它由一个设置动作目标（goal）的动作客户端（action client）和一个动作服务器（action server），动作服务器根据目标执行动作，并发送反馈和结果。

动作客户端和动作服务器之间进行异步双向消息通信。

动作服务器

动作服务器（action server）以从动作客户端接收的目标作为输入并且以结果和反馈值作为输出的消息通信的服务器。

在接收到来自客户端的目标值后，负责执行实际的动作。

动作客户端

动作客户端（action client）是以目标作为输出并以从动作服务器接收待结果和反馈值作为输入的消息通信的客户端。

它将目标交付给动作服务器，收到结果和反馈，并给出下一个指示或取消目标。

## 参数(Parameter)

ROS中的参数（parameter）是指节点中使用的参数。可以把它想象成一个

Windows程序中的\*.ini配置文件。这些参数是默认（default）设置的，可以根据需要从外部读取或写入。尤其是，它可以通过使用外部的写入功能实时更改设置值，因此非常有用。

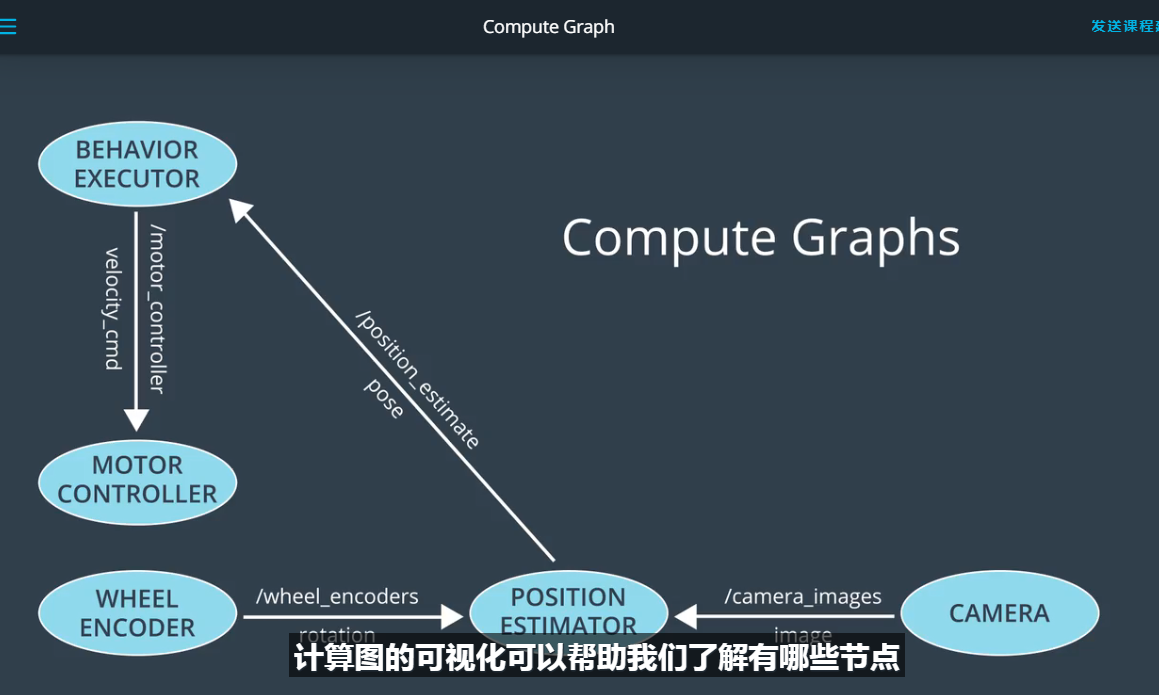
例如，您可以指定与外部设备连接的PC的USB端口、相机校准值、电机速度或命令的最大值和最小值等设置值。

参数服务器

参数服务器（parameter server）是指在功能包中使用参数时，注册各参数的服务器。参数服务器也是主节点的一个功能。

## 计算图(Computer Graph)

所有节点和图表，以及它们的连接方式，称为计算图；



计算图的可视化可以帮助我们了解有哪些节点，以及它们如何沟通，ROS提供了一个工具叫做rqt\_graph可以显示系统计算图；

一个中等机器人可以包含几十个节点，以及更多的topic；利用rqt\_graph的放大和平移功能，可以清晰的了解系统内部节点连接关系。

# 4.ROS命令

## 1: 配置环境

Source /opt/ros/kinetic/setup.bash

Env

ROS\_ROOT=/opt/ros/kinetic/share/ros The path where core ros packages are stored

PATH=/opt/ros/kinetic/bin:... The path to the ROS binaries, which we will be using throughout this lesson.

ROS\_DISTRO=kinetic Which distribution of ROS is being used. In this case, we are using kinetic

PYTHONPATH=/opt/ros/kinetic/lib/python2.7/dist-packages The path to the ROS python packages, which we will be using next lesson.

More information about the environment variables mentioned above, the ones not mentioned here, and others can be found [**here**](http://wiki.ros.org/ROS/EnvironmentVariables).

<http://wiki.ros.org/ROS/EnvironmentVariables>

自动配置

Echo “Source /opt/ros/kinetic/setup.bash ” >> ~/.bashrc

2. Starting the Master process

Before any ROS nodes can be run, the Master process must be started.

The Master process is responsible for the following (and more)

* Providing naming and registration services to other running nodes
* Tracking all publishers and subscribers
* Aggregating log messages generated by the nodes
* Facilitating connections between nodes

$ roscore

## 3.running Turtlesim Nodes

Now that the ROS master is running, we can run our first two ROS nodes.

First we will start the turtlesim\_node, in the turtlesim package using the following command.

$ rosrun turtlesim turtlesim\_node

Next, we will start the turtle\_teleop\_key node, also from the turtlesim package.

$ rosrun turtlesim turtle\_teleop\_key

## 4.Listing all Active Nodes

To get a list of all nodes that are active and have been registered with the ROS Master, we can use the command

rosnode list

We can see that there are three active nodes that have been registered with the ROS Master, /rosout, /teleop\_turtle, and /turtlesim.

* /rosout This node is launched by roscore. It subscribes to the standard /rosout topic, the topic to which all nodes send log messages.
* /teleop\_turtle This is our keyboard teleop node. Notice that it’s not named turtle\_teleop\_key. There’s no requirement that a node’s broadcasted name is the same as the name of it’s associated executable.
* /turtlesim The node name associated with the turtlebot\_sim node

## 5. Listing All Topics

In a similar fashion, we are able to query the ROS Master for a list of all topics. To do so, we use the command

rostopic list.

* /rosout\_agg Aggregated feed of messages published to /rosout.
* /turtle1/cmd\_vel Topic upon which velocity commands are sent/received. Publishing a velocity message to this topic will command turtle1 to move.
* /turtle1/color\_sensor Each turtle in turtlesim is equipped with a color sensor, and readings from the sensor are published to this topic.
* /turtle1/pose The position and orientation of turtle1 are published to this topic.

## 6. Get Information About a Specific Topic

If we wish to get information about a specific topic, who is publishing to it, subscribed to it, or the type of message associated with it, we can use the command

 $rostopic info.

Let’s check into the /turtle1/cmd\_vel topic:

$rostopic info /turtle1/cmd\_vel

As would be expected, there are two nodes registered on this topic. Our publisher, the teleop\_turtlenode, and our subscriber, the turtlesim node. Additionally, we can see that the type of message used on this topic is geometry\_msgs/Twist.

## 7.Show Message Information

Let’s get some more information about the geometry\_msgs/Twist message on the /turtle1/cmd\_veltopic, to do so, we will use the rosmsg info command.

We can see that a Twist message consists nothing more than two Vector3 messages. One for linear velocity, and another for angular velocities, with each velocity component being represented by a float64.

**Note**: Sometimes, the message definition won’t provide an ample amount of detail about a message type. For example, in the example above, how can we be sure that linear and angular vectors above refer to velocities, and not positions? One way to get more detail would be to look at the comments in the message’s definition file. To do so, we can issue the following command:

rosed geometry\_msgs Twist.msg.

**Note 2:** More information about rosed, including how to select which editor is used by default can be found [**here**](http://wiki.ros.org/ROS/Tutorials/UsingRosEd).

<http://wiki.ros.org/ROS/Tutorials/UsingRosEd>

## 8.Echo Messages on a Topic

Sometimes it may be useful to look at a topic’s published messages in real time.

To do so, we can use the command rostopic echo. Let’s take a look at the /turtle1/cmd\_vel topic.

$ rostopic echo /turtle1/cmd\_vel

If we then command the turtle to move from the turtle\_teleop\_key window, we will be able to see the output message in real-time.

# Lesson5. Packages & Catkin Workspaces

Catkin工作空间本质上是一个目录，可以在这里编写、修改并安装catkin包，通常当你开发基于ROS的机器人或项目时，你需要在单独的工作空间里完成。这个单独的空间需要保存大量不同的catkin包，所有ROS软件组件都以catkin包的形式组织并发布。

Catkin包可能包含节点的源代码，有用的脚本、配置文件等等。

## 1.Create a Catkin Workspace

Step 1: mkdir -p ~/catkin\_ws/src

$ mkdir -p ~/catkin\_ws/src

Step 2:

$ cd ~/catkin\_ws/src

Step3:

$ catkin\_init\_workspace

Step4:

$ cd ~/catkin\_ws

Step5:

$ catkin\_make

More in

<http://wiki.ros.org/catkin/conceptual_overview>

You now have two new directories: build and devel. The aptly named build directory is the build space for C++ packages and, for the most part, you will not interact with it. The devel directory does contain something of interest, a file named setup.bash. This setup.bash script must be sourced before using the catkin workspace.

## 2. Add a package

### Step1: Cloning the simple\_arm Package

One of the biggest benefits of using ROS is that it has a really large community of users and developers, so there is a lot of code that you can use.

$ cd ~/catkin\_ws/src

$ git clone https://github.com/udacity/simple\_arm\_01.git simple\_arm

### Step2: Building the simple\_arm package

After the repo has finished cloning, you can change directory to the top-level of the ros workspace and build the new package.

$ cd ~/catkin\_ws

$ catkin\_make

### step3: launch multiple nodes

roslaunch allows you to do the following

* Launch ROS Master and multiple nodes with one simple command
* Set default parameters on the parameter server
* Automatically re-spawn processes that have died

Once the workspace has been built, you can source it’s setup script:

$ source devel/setup.bash

With your workspace sourced you can now launch simple\_arm:

$ roslaunch simple\_arm robot\_spawn.launch

## 3.rosdep 自动检查依赖库

After the last exercise, you might have noticed the following warning line:

The controller spawner couldn’t find the expected controller\_manager ROS interface.

ROS packages have two different types of dependencies: build dependencies, and run dependencies. This error message was due to a missing runtime dependency.

The rosdep tool will check for a package's missing dependencies, download them, and install them.

To check for missing dependencies in the simple\_arm package:

$ rosdep check simple\_arm

To have rosdep install packages, invoke the following command from the root of the catkin workspace

$ rosdep install -i simple\_arm

Issues with this command may arise when using a VM. If this is the case, please try:

sudo apt-get install ros-kinetic-gazebo-ros-control

Dive Deeper into Packages

$ cd ~/catkin\_ws/src

The syntax for creating a catkin package is simply,

$ catkin\_create\_pkg <your\_package\_name> [dependency1 dependency2 …]

$ catkin\_create\_pkg first\_package

I mentioned earlier that ROS packages have a conventional directory structure. Let’s take a look at a more typical package.

* scripts (python executables)
* src (C++ source files)
* msg (for custom message definitions)
* srv (for service message definitions)
* include -> headers/libraries that are needed as dependencies
* config -> configuration files
* launch -> provide a more automated way of starting nodes

Other folders may include

* urdf (Universal Robot Description Files)
* meshes (CAD files in .dae (Collada) or .stl (STereoLithography) format)
* worlds (XML like files that are used for Gazebo simulation environments)

There are many packages that you can install. To see a list of available packages for the Kinetic distribution, take some time to explore [here](http://www.ros.org/browse/list.php).

<https://www.ros.org/browse/list.php>

# 6 Writing ROS Nodes

## 6.0 add a script

step1. Adding the scripts directory

In order to create a new node in python, you must first create the scripts directory within the simple\_arm package, as it does not yet exist.

$ cd ~/catkin\_ws/src/simple\_arm/

$ mkdir scripts

step2. Creating a new script

$ cd scripts

$ echo '#!/bin/bash' >> hello

$ echo 'echo Hello World' >> hello

Step3:run the script

After setting the appropriate execution permissions on the file, rebuilding the workspace, and sourcing the newly created environment, you will be able to run the script.

$ chmod u+x hello

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ rosrun simple\_arm hello

## 6.1 Add a Publisher

Step1:Creating the empty simple\_mover node script

$ cd ~/catkin\_ws/src/simple\_arm

$ cd scripts

$ touch simple\_mover

$ chmod u+x simple\_mover

Step2: Edit simple\_mover file

Publishers allow a node to send messages to a topic

pub1 = rospy.Publisher("/topic\_name", message\_type, queue\_size=size)

*#!/usr/bin/env python*

**import** math

**import** rospy

**from** std\_msgs.msg **import** Float64

**def** **mover**():

pub\_j1 = rospy.Publisher('/simple\_arm/joint\_1\_position\_controller/command',

Float64, queue\_size=10)

pub\_j2 = rospy.Publisher('/simple\_arm/joint\_2\_position\_controller/command',

Float64, queue\_size=10)

‘’’

Initializes a client node and registers it with the master. Here “arm\_mover” is the name of the node. init\_node() must be called before any other rospy package functions are called. The argument anonymous=True makes sure that you always have a unique name for your node

’’’

rospy.init\_node('arm\_mover')

rate = rospy.Rate(10)

start\_time = 0

**while** **not** start\_time:

start\_time = rospy.Time.now().to\_sec()

**while** **not** rospy.is\_shutdown():

elapsed = rospy.Time.now().to\_sec() - start\_time

pub\_j1.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

pub\_j2.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

rate.sleep()

**if** \_\_name\_\_ == '\_\_main\_\_':

**try**:

mover()

**except** rospy.ROSInterruptException:

**pass**

Step3 Running simple\_mover

Assuming that your workspace has recently been built, and it’s setup.bash has been sourced, you can launch simple\_arm as follows:

$ cd ~/catkin\_ws

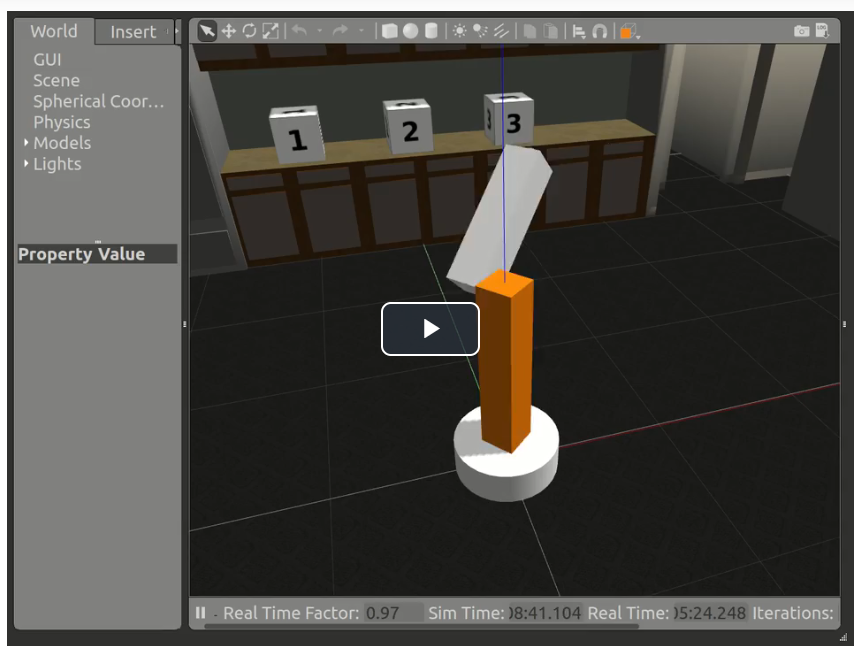
$ roslaunch simple\_arm robot\_spawn.launch

Once ROS Master, Gazebo, and all of our relevant nodes are up and running, we can finally launch simple\_mover. To do so, open a new terminal and type the following commands:

$ cd ~/catkin\_ws

$ source devel/setup.bash

$ rosrun simple\_arm simple\_mover



## 6.2 ROS Services

### 6.2.1 Defining services

service = rospy.Service('service\_name', serviceClassName, handler)

each service has a definition provided in an .srv file; this is a text file that provides the proper message type for both requests and responses.

The handler is the name of the function or method that handles the incoming service message. This function is called each time the service is called, and the message from the service call is passed to the handler as an argument. The handler should return an appropriate service response message.

### 6.2.2 Using Services

Services can be called directly from the command line, and you will see an example of this in the upcoming arm\_mover classroom concepts.

On the other hand, to use a ROS service from within another node, you will define a ServiceProxy, which provides the interface for sending messages to the service:

service\_proxy = rospy.ServiceProxy('service\_name', serviceClassName)

One way the ServiceProxy can then be used to send requests is as follows:

msg = serviceClassNameRequest()

*#update msg attributes here to have correct data*

response = service\_proxy(msg)

<http://wiki.ros.org/rospy/Overview/Services>

## 6.3: add a service

Namely, we still need to cover:

* Custom message generation
* Services
* Parameters
* Launch Files
* Subscribers
* Logging

step1. Creating a new service definition

As you learned earlier, an interaction with a service consists of two messages being passed. A request passed to the service, and a response received from the service. The definitions of the request and response message type are contained within .srv files living in the srv directory under the package’s root.

$ cd ~/catkin\_ws/src/simple\_arm/

$ mkdir srv

$ cd srv

$ touch GoToPosition.srv

“GoToPosition” 是服务类型名字

You should now edit GoToPosition.srv, so it contains the following:

float64 joint\_1

float64 joint\_2

---

duration time\_elapsed

Service definitions always contain two sections, separated by a ‘---’ line. The first section is the definition of the request message. Here, a request consists of two float64 fields, one for each of simple\_arm’s joints. The second section contains is the service response. The response contains only a single field, time\_elapsed. The time\_elapsed field is of type duration, and is responsible for indicating how long it took the arm to perform the movement.

Step2: Modifying CMakeLists.txt

In order for catkin to generate the python modules or C++ libraries which allow you to utilize messages in your code you must first modify simple\_arm’s CMakeLists.txt (~/catkin\_ws/src/simple\_arm/CMakeLists.txt).

**First,** ensure that the find\_package() macro lists std\_msgs and message\_generation as required packages. The find\_package() macro should look as follows:

find\_package(catkin REQUIRED COMPONENTS

std\_msgs

message\_generation

)

As the names might imply, the std\_msgs package contains all of the basic message types, and message\_generation is required to generate message libraries for all the supported languages (cpp, lisp, python, javascript).

**Note**: In your CMakeLists.txt, you may also see controller\_manager listed as a required package. In actuality this package is not required. It was simply added as a means to demonstrate a build failure in the previous lesson. You may remove it from the list of REQUIRED COMPONENTS if you choose.

**Next,** uncomment the commented-out add\_service\_files() macro so it looks like this:

## Generate services in the 'srv' folder

add\_service\_files(

FILES

GoToPosition.srv

)

This tells catkin which files to generate code for.

**Lastly,** make sure that the generate\_messages() macro is uncommented, as follows:

generate\_messages(

DEPENDENCIES

std\_msgs # Or other packages containing msgs

)

Step3: Modifying package.xml

package.xml is responsible for defining many of the package’s properties, such as the name of the package, version numbers, authors, maintainers, and dependencies.

When rosdep is searching for these dependencies, it’s the package.xml file that is being parsed. Let’s add the message\_generation and message\_runtimedependencies.

<buildtool\_depend>catkin</buildtool\_depend>

<build\_depend>message\_generation</build\_depend>

<run\_depend>controller\_manager</run\_depend>

<run\_depend>effort\_controllers</run\_depend>

<run\_depend>gazebo\_plugins</run\_depend>

<run\_depend>gazebo\_ros</run\_depend>

<run\_depend>gazebo\_ros\_control</run\_depend>

<run\_depend>joint\_state\_controller</run\_depend>

<run\_depend>joint\_state\_publisher</run\_depend>

<run\_depend>robot\_state\_publisher</run\_depend>

<run\_depend>message\_runtime</run\_depend>

<run\_depend>xacro</run\_depend>

You are now ready to build the package! For more information about package.xml, check out the [ROS Wiki](http://wiki.ros.org/catkin/package.xml).

<http://wiki.ros.org/catkin/package.xml>

step4: Building the package

If you build the workspace successfully, you should now find that a python package containing a module for the new service GoToPosition has been created deep down in the devel directory.

$ cd ~/catkin\_ws

$ catkin\_make

$ cd devel/lib/python2.7/dist-packages

$ ls

$ env | grep PYTHONPATH

Step5: Creating the empty arm\_mover node script

The steps you take to create the arm\_mover node are exactly the same as the steps you took to create the simple\_mover script, excepting the actual name of the script itself.

$ cd ~/catkin\_ws

$ cd src/simple\_arm/scripts

$ touch arm\_mover

$ chmod u+x arm\_mover

You can now edit the empty arm\_mover script with your favorite text editor.

arm\_mover

*#!/usr/bin/env python*

**import** math

**import** rospy

**from** std\_msgs.msg **import** Float64

**from** sensor\_msgs.msg **import** JointState

**from** simple\_arm.srv **import** \*

**def** **at\_goal**(pos\_j1, goal\_j1, pos\_j2, goal\_j2):

tolerance = .05

result = abs(pos\_j1 - goal\_j1) <= abs(tolerance)

result = result **and** abs(pos\_j2 - goal\_j2) <= abs(tolerance)

**return** result

**def** **clamp\_at\_boundaries**(requested\_j1, requested\_j2):

clamped\_j1 = requested\_j1

clamped\_j2 = requested\_j2

min\_j1 = rospy.get\_param('~min\_joint\_1\_angle', 0)

max\_j1 = rospy.get\_param('~max\_joint\_1\_angle', 2\*math.pi)

min\_j2 = rospy.get\_param('~min\_joint\_2\_angle', 0)

max\_j2 = rospy.get\_param('~max\_joint\_2\_angle', 2\*math.pi)

**if** **not** min\_j1 <= requested\_j1 <= max\_j1:

clamped\_j1 = min(max(requested\_j1, min\_j1), max\_j1)

rospy.logwarn('j1 is out of bounds, valid range (%s,%s), clamping to: %s',

min\_j1, max\_j1, clamped\_j1)

**if** **not** min\_j2 <= requested\_j2 <= max\_j2:

clamped\_j2 = min(max(requested\_j2, min\_j2), max\_j2)

rospy.logwarn('j2 is out of bounds, valid range (%s,%s), clamping to: %s',

min\_j2, max\_j2, clamped\_j2)

**return** clamped\_j1, clamped\_j2

**def** **move\_arm**(pos\_j1, pos\_j2):

time\_elapsed = rospy.Time.now()

j1\_publisher.publish(pos\_j1)

j2\_publisher.publish(pos\_j2)

**while** **True**:

joint\_state = rospy.wait\_for\_message('/simple\_arm/joint\_states', JointState)

**if** at\_goal(joint\_state.position[0], pos\_j1, joint\_state.position[1], pos\_j2):

time\_elapsed = joint\_state.header.stamp - time\_elapsed

**break**

**return** time\_elapsed

**def** **handle\_safe\_move\_request**(req):

rospy.loginfo('GoToPositionRequest Received - j1:%s, j2:%s',

req.joint\_1, req.joint\_2)

clamp\_j1, clamp\_j2 = clamp\_at\_boundaries(req.joint\_1, req.joint\_2)

time\_elapsed = move\_arm(clamp\_j1, clamp\_j2)

**return** GoToPositionResponse(time\_elapsed)

**def** **mover\_service**():

rospy.init\_node('arm\_mover')

service = rospy.Service('~safe\_move', GoToPosition, handle\_safe\_move\_request)

rospy.spin()

**if** \_\_name\_\_ == '\_\_main\_\_':

j1\_publisher = rospy.Publisher('/simple\_arm/joint\_1\_position\_controller/command',

Float64, queue\_size=10)

j2\_publisher = rospy.Publisher('/simple\_arm/joint\_2\_position\_controller/command',

Float64, queue\_size=10)

**try**:

mover\_service()

**except** rospy.ROSInterruptException:

**pass**

step6: Launch and Interact

**First**, Launching the project with the new service

To get the arm\_mover node, and accompanying safe\_move service to launch along with all of the other nodes, you will modify robot\_spawn.launch.

Launch files, when they exist, are located within the launch directory in the root of a catkin package. simple\_arm’s launch file is located in ~/catkin\_ws/src/simple\_arm/launch

To get the arm\_mover node to launch, simply add the following:

*<!-- The arm mover node -->*

<node name="arm\_mover" type="arm\_mover" pkg="simple\_arm">

<rosparam>

min\_joint\_1\_angle: 0

max\_joint\_1\_angle: 1.57

min\_joint\_2\_angle: 0

max\_joint\_2\_angle: 1.0

</rosparam>

</node>

More information on the format of the launch file can be found

<http://wiki.ros.org/roslaunch/XML>

step7 Testing the new service

Now that you've modified the launch file, you are ready to test everything out.

First. To do so, launch the simple\_arm, verify that the arm\_mover node is running, and that the safe\_move service is listed:

**Note:** You will need to make sure that you've exited out of your previous roslaunch session before re-launching.

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

Then, in a new terminal, verify that the node and service have indeed launched.

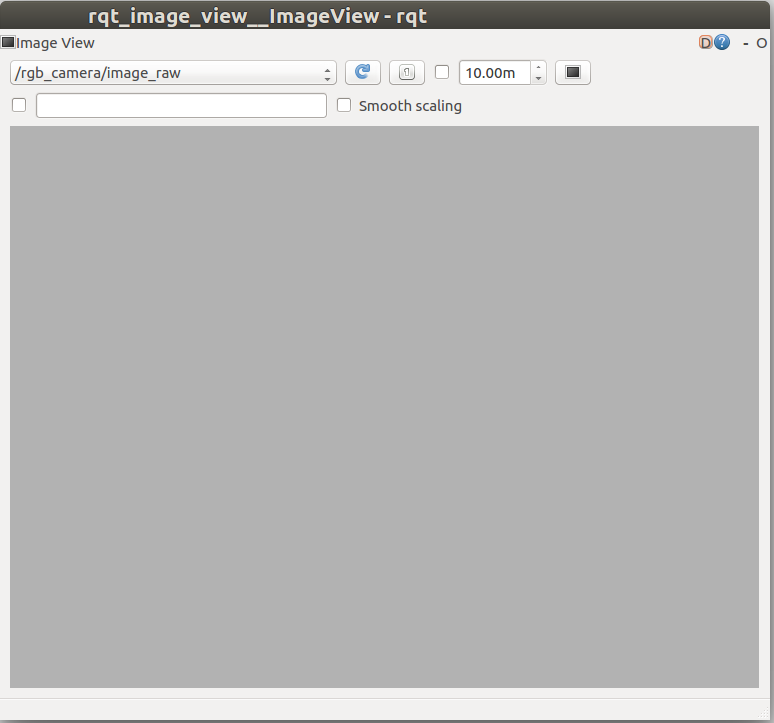
$ rosnode list

$ rosservice list

To view the camera image stream, you can use the command rqt\_image\_view (you can learn more about rqt and the associated tools [here](http://wiki.ros.org/rqt)):

<http://wiki.ros.org/rqt>

$ rqt\_image\_view /rgb\_camera/image\_raw



Adjusting the view

The camera is displaying a gray image. This is as to be expected, given that it is straight up, towards the gray sky of our gazebo world.

To point the camera towards the numbered blocks on the counter top, we would need to rotate both joint 1 and joint 2 by approximately pi/2 radians. Let’s give it a try:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ rosservice call /arm\_mover/safe\_move "joint\_1: 1.57

joint\_2: 1.57"

Note: rosservice call can tab-complete the request message, so that you don’t have to worry about writing it out by hand. Also, be sure to include a line break between the two joint parameters.

What was not expected is the resulting position of the arm. Looking at the roscore console, we can very clearly see what the problem was. The requested angle for joint 2 was out of the safe bounds. We requested 1.57 radians, but the maximum joint angle was set to 1.0 radians.

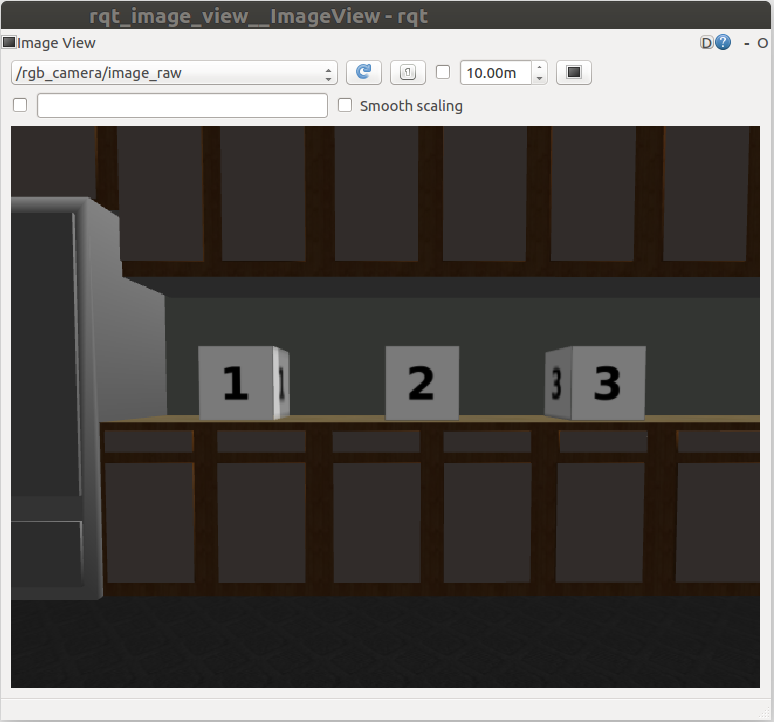
By setting the max\_joint\_2\_angle on the parameter server, we should be able to bring the blocks into view the next time a service call is made. To increase joint 2’s maximum angle, you can use the command rosparam

$ rosparam set /arm\_mover/max\_joint\_2\_angle 1.57

Now we should be able to move the arm such that all of the blocks are within the field of view of the camera:

rosservice call /arm\_mover/safe\_move "joint\_1: 1.57

joint\_2: 1.57"



And there you have it. All of the blocks are within the field of view!

## 6.4 add Subscribers

添加一个订阅者节点，根据从主题获取的数据，做相应处理，当出现机械臂指向天空时，发送一个service消息，把机械臂移动到指定位置。

也包括一个发送服务消息

A Subscriber enables your node to read messages from a topic, allowing useful data to be streamed into the node.

sub1 = rospy.Subscriber("/topic\_name", message\_type, callback\_function)

The "/topic\_name" indicates which topic the Subscriber should listen to.

The message\_type is the type of message being published on "/topic\_name".

The callback\_function is the name of the function that should be called with each incoming message. Each time a message is received, it is passed as an argument to callback\_function. Typically, this function is defined in your node to perform a useful action with the incoming data. Note that unlike service handler functions, the callback\_function is not required to return anything.

More in <http://docs.ros.org/api/rospy/html/rospy.topics.Subscriber-class.html>

创建一个look\_away节点， look\_away 节点订阅/rgb\_camera/image\_raw 主题，该主题可以从robotic arm获取摄像头数据。

## Step1: Creating the empty look\_away node scrip

$ cd ~/catkin\_ws

$ cd src/simple\_arm/scripts

$ touch look\_away

$ chmod u+x look\_away

Note: If look\_away starts before the system has fully initialized, then look\_away hangs in the call to safe\_move.

解决办法：*wait\_for\_message*

**def** **\_\_init\_\_**(self):

rospy.init\_node('look\_away')

self.last\_position = **None**

self.arm\_moving = **False**

rospy.wait\_for\_message('/simple\_arm/joint\_states', JointState)

rospy.wait\_for\_message('/rgb\_camera/image\_raw', Image)

self.sub1 = rospy.Subscriber('/simple\_arm/joint\_states',

JointState, self.joint\_states\_callback)

self.sub2 = rospy.Subscriber('/rgb\_camera/image\_raw',

Image, self.look\_away\_callback)

self.safe\_move = rospy.ServiceProxy('/arm\_mover/safe\_move',

GoToPosition)

rospy.spin()

## step2: Updating the launch file

*<!-- The look away node -->*

<node name="look\_away" type="look\_away" pkg="simple\_arm"/>

修改默认值

*<!-- The arm mover node -->*

<node name="arm\_mover" type="arm\_mover" pkg="simple\_arm">

<rosparam>

min\_joint\_1\_angle: 0

max\_joint\_1\_angle: 1.57

min\_joint\_2\_angle: 0

max\_joint\_2\_angle: 1.57

</rosparam>

</node>

Look\_up

*#!/usr/bin/env python*

**import** math

**import** rospy

**from** sensor\_msgs.msg **import** Image, JointState

**from** simple\_arm.srv **import** \*

**class** **LookAway**(object):

**def** **\_\_init\_\_**(self):

rospy.init\_node('look\_away')

self.sub1 = rospy.Subscriber('/simple\_arm/joint\_states',

JointState, self.joint\_states\_callback)

self.sub2 = rospy.Subscriber("rgb\_camera/image\_raw",

Image, self.look\_away\_callback)

self.safe\_move = rospy.ServiceProxy('/arm\_mover/safe\_move',

GoToPosition)

self.last\_position = **None**

self.arm\_moving = **False**

rospy.spin()

**def** **uniform\_image**(self, image):

**return** all(value == image[0] **for** value **in** image)

**def** **coord\_equal**(self, coord\_1, coord\_2):

**if** coord\_1 **is** **None** **or** coord\_2 **is** **None**:

**return** **False**

tolerance = .0005

result = abs(coord\_1[0] - coord\_2[0]) <= abs(tolerance)

result = result **and** abs(coord\_1[1] - coord\_2[1]) <= abs(tolerance)

**return** result

**def** **joint\_states\_callback**(self, data):

**if** self.coord\_equal(data.position, self.last\_position):

self.arm\_moving = **False**

**else**:

self.last\_position = data.position

self.arm\_moving = **True**

**def** **look\_away\_callback**(self, data):

**if** **not** self.arm\_moving **and** self.uniform\_image(data.data):

**try**:

rospy.wait\_for\_service('/arm\_mover/safe\_move')

msg = GoToPositionRequest()

msg.joint\_1 = 1.57

msg.joint\_2 = 1.57

response = self.safe\_move(msg)

rospy.logwarn("Camera detecting uniform image. \

Elapsed time to look at something nicer:\n%s",

response)

**except** rospy.ServiceException, e:

rospy.logwarn("Service call failed: %s", e)

**if** \_\_name\_\_ == '\_\_main\_\_':

**try**:

LookAway()

**except** rospy.ROSInterruptException:

**pass**

step4: Launch and Interact

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

After launching, the arm should move away from the grey sky and look towards the blocks. To view the camera image stream, you can use the same command as before:

观察图片数据

$ rqt\_image\_view /rgb\_camera/image\_raw

发送消息操作simple\_arm

rosservice call /arm\_mover/safe\_move "joint\_1: 0

joint\_2: 0"

Note：

1.对safe\_move发送一个service消息，移动simple\_arm，移动的相关信息会发布在主题“joint\_states”和“rgb\_camera/image\_raw”；

2.look\_away节点作为订阅者节点，从这两个主题中获取位置和图像信息，当发现摄像头朝向天空时，发送一个service消息，将simple\_arm移动到指定位置。

## 7. log信息

By default all logging messages for a node are written to the node's log file which can be found in ~/.ros/log or ROS\_ROOT/log .

If roscore is running, you can use roscd to find log file directory by opening a new terminal window and typing:

roscd log

In this directory, you should see directories from runs of your ROS code, along with a latest directory with log files from the most recent run.

### 7.1 Logging levels and outputs

rospy.logdebug(...)

rospy.loginfo(...)

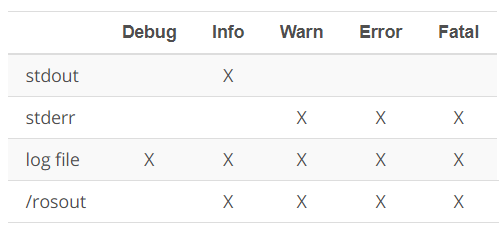
rospy.logwarn(...)

rospy.logerr(...)

rospy.logfatal(...)

All levels of logging messages are recorded in ROS log files, but some message levels may also be sent to Python stdout, Python stderr, or the ROS topic /rosout.

The loginfo messages are written to Python's stdout, while logwarn, logerr, and logfatal are written to Python's stderr by default. Additionally, loginfo, logwarn, logerr, and logfatal are written to /rosout.



More in <http://wiki.ros.org/rospy/Overview/Logging>

## 7.2 Filtering and saving log messages from /rosout

Note that for messages written to /rosout, you can see the messages in real time as your program is running by echoing:

rostopic echo /rosout

一般可以用grep来过滤有用信息

rostopic echo /rosout | grep insert\_search\_expression\_here

rostopic echo /rosout | grep insert\_search\_expression\_here > path\_to\_output/output.txt

### 7.3 Modifying message level sent to /rosout

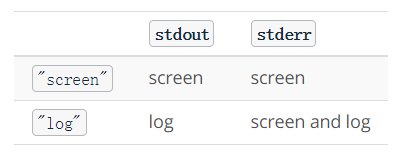
To do this you must set the log\_level attribute within the rospy.init\_node code.

For example, if you'd like to allow lodebug messages to be written to /rosout, that can be done as follows:

rospy.init\_node('my\_node', log\_level=rospy.DEBUG)

### 7.5 Modifying display of messages sent to stdout and stderr

It is also possible to change how messages to stdout and stderr are displayed or logged. Within a package's .launch file, the output attribute for a node tag can be set to "screen" or "log". The following table summarizes how the different output options change the display of the node's stdoutand stderr messages:



For example, setting output="screen" for the look\_away node in robot\_spawn.launch will display both stdout and stderr messages in the screen:

*<!-- The look away node -->*

<node name="look\_away" type="look\_away" pkg="simple\_arm" output="screen"/>

If the output attribute is left empty, the default is "log".