

ROS提供一些标准操作系统服务，例如**硬件抽象**、**底层设备控制**、**常用功能实现**、**进程间消息**以及**数据包管理**。ROS是基于一种图状架构，从而不同节点的进程能接受、发布、聚合各种信息（如传感、控制、状态、规划等）。

目前ROS仅适用于Apollo 3.0之前的版本，最新代码及功能还请参照Apollo 3.5及5.0版本。

以下，ENJOY



ROS的历史

History of ROS

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
- Since 2013 managed by OSRF
- Today used by many robots, universities and companies
- De facto standard for robot programming



ROS是2007年在斯坦福大学里面的一个实验室学生开发出来的一套机器人通用的一个框架，2013年被纳入OSRF机构统一管理，如今被很多公司和大学的研究所广泛地使用到一些科研项目中。



ROS的特征

ROS有5个比较明显的特征如下：

1. 点对点：两个Node之间进行消息通讯是一个点对点的行为。
2. 它支持分布式：在部署多机之间的消息通讯时，ROS提供了一个天然的支持。
3. 它是跨语言，它并不关注每个节点之间是用什么语言来写的。你只需要按照ROS提供的一些接口完成消息的订阅和分发即可以完成一个消息之间的通信。
4. 它是一个轻量级的ROS程序，用户只需要关注自己核心模块的算法逻辑，不需要关注底层是如何通信、如何断开通信、如何进行Service 和Param之间的一些交互的。

5. 它是一个开源的框架，大家都可以往ROS里面贡献自己的一些想法和代码。



ROS的几个核心概念

/// 松耦合

ROS是一个松耦合的框架，松耦合就是各个节点之间的通信是一个解耦合的关系。

/// 节点

一个算法模块，比如自动驾驶系统里面的**感知模块**、**定位模块**、**决策模块**或者**控制模块**，这些模块就是一个简单的算法集合，在ROS里面被称为一个节点。

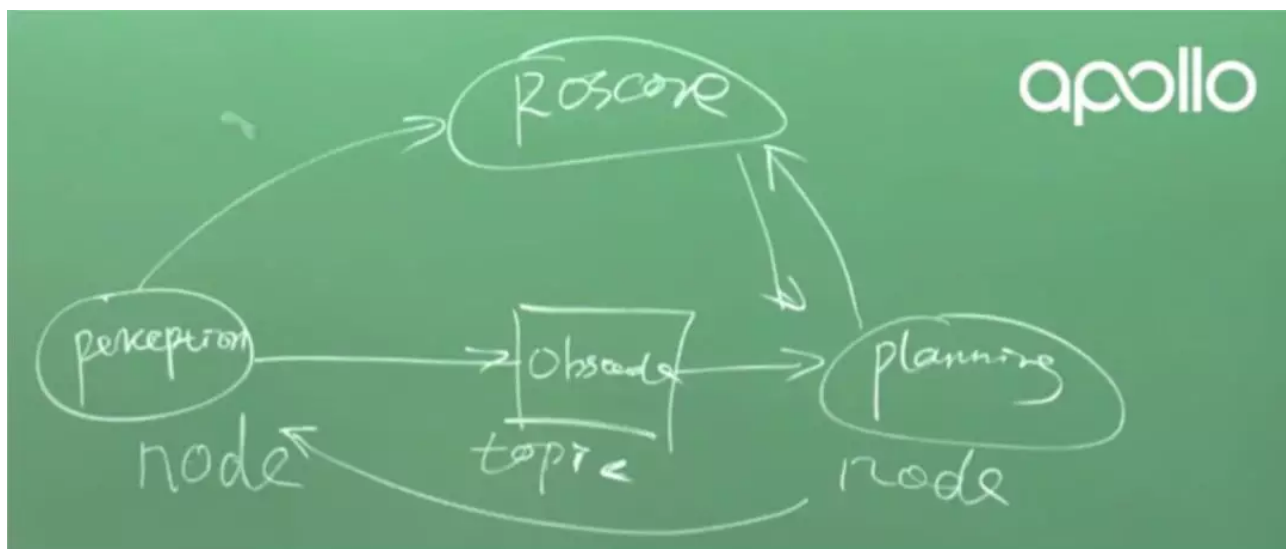
/// 节点管理器

在ROS里面被定义为Master，用来集中式管理各个独立的、松耦合、无序节点之间的逻辑关系，它是轻量级的介入，当各个节点启动完成以后，他们在通信连接完成之前起到中转也就是类似于交换机的作用。

/// Topic

两个节点之间的通信主题。Topic内部使用的数据格式是Message。Message是一系统简单的数据类型或者是一些自定义的复杂数据类型，所组装成的一个描述文件。

以上几个概念之间的相互关系，如下图所示：



感知模块**Perception**，感知车辆周围的一些障碍物信息，用**CNN**或者**RNN**算法将障碍物信息提取出来，即Obstacle。再将这些信息输出给下游Planning节点。这两个节点之间的通信连接就需要Roscore，即节点管理器。

Perception、Planning在启动的时候没有先后关系，这是松耦合的一个具体体现。Perception先启动并向Roscore发送一个注册信息，同时会订阅名为Obstacle的Topic；Planning节点启动后也向Roscore发送一个注册信息，同时会订阅名为Obstacle的Topic；在这种情况下，Roscore会发送一个通知信息给Planning，在它发送注册信息之前已经有一个节点启动了。此时Planning会向Perception发送消息请求通信连接，Planning收到消息之后会在Planning和Perception两个节点中间建立一个实时通信链路。当通信链路建立之后，Roscore的功能就暂时完成了。

所以，松耦合在此有两种体现：

1. Perception和Planning两者之间的启动没有先后关系。
2. 当通信链路建立之后，Roscore的功能就暂时完成了。

这些概念在ROS系统都有一整套的命令工具支持如下：

1. **Roscore**：启动一个节点管理器。

2. 节点常用命令：

Rosnode list：可以列出当前系统里面所存在的节点。

Rosnode info：查看某一节点的具体的一些信息。

3. Topic常用命令：

Rostopic list：可以查看所存在Topic的一些列表。

Rostopic info：可以查看到发送这个Topic的发送方，订阅这个Topic的订阅方。

Rostopic type：查看Topic内部所使用的MSG的数据结构。

Rostopic pub：调试计算节点模块的一些基本功能。

04

ROS的实践

下面是一个简单的实践：

第一部分：启动Roscore

Example

Console Tab Nr. 1 – Starting a *roscore*

Start a roscore with

```
> roscore
```

```

student@ubuntu:~/catkin_ws$ roscore
... logging to /home/student/.ros/log/6c1852aa-e961-11e6-8543-000c297bd368/ros
launch-ubuntu-6696.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:34089/
ros_comm version 1.11.20

SUMMARY
=====
PARAMETERS
 * /rostdistro: indigo
 * /rosversion: 1.11.20

NODES
auto-starting new master
process[master]: started with pid [6708]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to 6c1852aa-e961-11e6-8543-000c297bd368
process[roscout-1]: started with pid [6721]
started core service [/roscout]

```



启动Roscore，通过一个简单的命令行Roscore就可以启动一个节点管理器。不需要进行任何参数的传入，也不需要进行任何配置。

如果有很多个节点启动的时候，会使用Roslaunch。[Roslaunch](#)就是把所有启动节点的行为放到统一的描述文件里，在启动的时候会在描述文件里找到定义的各个节点的位置,然后启动节点。

第二部分：启动一个简单的Talker程序

Example

Console Tab Nr. 2 – Starting a *talker* node

Run a talker demo node with

```
> rosrunc roscpp_tutorials talker
```

```


student@ubuntu:~/catkin_ws$ rosrunc roscpp_tutorials talker
[ INFO] [1486051708.424661519]: hello world 0
[ INFO] [1486051708.525227845]: hello world 1
[ INFO] [1486051708.624747612]: hello world 2
[ INFO] [1486051708.724826782]: hello world 3
[ INFO] [1486051708.825928577]: hello world 4
[ INFO] [1486051708.925379775]: hello world 5
[ INFO] [1486051709.024971132]: hello world 6
[ INFO] [1486051709.125450960]: hello world 7
[ INFO] [1486051709.225272747]: hello world 8
[ INFO] [1486051709.325389210]: hello world 9

```



Talker就是一个发送节点的程序。单独启动一个节点，除了Roscore启动之外，其它的节点启动ROS提供了一个Rosrun命令。Rosrun，前面是Package包，后面是实际的可执行文件。通过这样一个命令可以直接启动一个Talker。

第三部分：通过命令查看Talker node



Example

Console Tab Nr. 3 – Analyze *talker* node

See the list of active nodes

```
> rosnode list
```

Show information about the *talker* node

```
> rosnode info /talker
```

See information about the *chatter* topic

```
> rostopic info /chatter
```

```
student@ubuntu:~/catkin_ws$ rosnode list
/rosout
/talker
```

```
student@ubuntu:~/catkin_ws$ rosnode info /talker
-----
Node [/talker]
Publications:
 * /chatter [std_msgs/String]
 * /rosout [roscpp_msgs/Log]


Subscriptions: None

Services:
 * /talker/get_loggers
 * /talker/set_logger_level
```

```
student@ubuntu:~/catkin_ws$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://ubuntu:39173/)

Subscribers: None
```

Peter Fackhauser | 16.02.2018 | 17

当启动这个节点之后，用Rosnode list，见上图Talker的Node文件，还有一个Rosout的程序节点。Roscore默认启动的时候启动了一个隐藏节点，它是一个记录日志相关的节点，所有节点发生的Log都会被Roscore启动的Rosout所订阅，订阅完之后会根据一些特定的规则把这些Log分级，然后分模块、分文件打印到对应的模块日志里。

Rosnode info查看Talker相关的一些节点，Talker发送的Topic以及它发送的Service。它有两个Service：**Setlogger**、**Getlogger**。这两个是每一个节点都会默认启动的两个Service，这两个Service的作用是设置这一个节点里面的日志层级，如果日志层级是INFO，那么它打印的Debug信息就不会记录在Roscore的Rosout节点里面。

Rostopic info，通过这个命令我们能看到Topic的发送方和接收方。

Example

Console Tab Nr. 3 – Analyze *chatter* topic

Check the type of the *chatter* topic

```
> rostopic type /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic type /chatter
std_msgs/String
```

Show the message contents of the topic

```
> rostopic echo /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic echo /chatter
data: hello world 11874
---
data: hello world 11875
---
data: hello world 11876
```

Analyze the frequency

```
> rostopic hz /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic hz /chatter
subscribed to [/chatter]
average rate: 9.991
  min: 0.099s max: 0.101s std dev: 0.00076s window: 10
average rate: 9.996
  min: 0.099s max: 0.101s std dev: 0.00069s window: 20
```



Rostopic type是查看Topic的一个Message的消息类型。

Rostopic echo是相当于起了一个Listener节点，去展示Talker发的Topic包含的具体信息。

Rostopic还提供了**HZ**和**BW**功能，HZ是统计Talker节点发送Obstacle topic的频率，根据此频率能简单的探测系统是否按照我们所预期的方向来执行，例如自动驾驶整个车系统里面每一个传感器有一定的频率，激光雷达是十赫兹，即一秒钟转十圈，会发十帧点云图像，我们可以通过Rostopic HZ去检测Topic是不是一秒钟发送十赫兹，如果低于十赫兹，说明当前系统肯定是有异常，要么是激光雷达扫描的过程受到影响，要么是顶层的Driver节点在处理激光雷达顶层信息的时候中间出现了一些故障，此时我们就需要具体探测问题出现在哪个地方。

/// 第四部分：启动一个Listener节点

Example

Console Tab Nr. 4 – Starting a *listener* node

Run a listener demo node with

```
> rosrn roscpp_tutorials listener
```

```
student@ubuntu:~/catkin_ws$ rosrn roscpp_tutorials listener
[INFO] [1486053802.204104598]: I heard: [hello world 19548]
[INFO] [1486053802.304538027]: I heard: [hello world 19549]
[INFO] [1486053802.403853395]: I heard: [hello world 19550]
[INFO] [1486053802.504438133]: I heard: [hello world 19551]
[INFO] [1486053802.604297608]: I heard: [hello world 19552]
```



现在启动一个Listener节点，启动Listener节点之后整个拓扑会有一个比较明显的变化，Listener启动向Roscore发送一个注册信息，同时会订阅Topic，Roscore会发送一个通知信息给Listener：在它发送注册信息之前已经有一个节点启动了。此时，Listener会向Talker发送消息请求通信连接，Listener收到消息之后会在Listener和Talker两个节点中间建立一个实时通信链路。这个通信链路是基于TCP的，TCP建立起来之后Talker就持续不断的发送信息，Listener接到信息之后去做回调处理供实际的决策和执行。

第五部分：再次通过命令查看Node

Example

Console Tab Nr. 3 – Analyze

See the new *listener* node with

```
> rosnodet list
```

```
student@ubuntu:~/catkin_ws$ rosnodet list
/listener
/rosout
/talker
```

Show the connection of the nodes over the chatter topic with

```
> rostopic info /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://ubuntu:39173/)

Subscribers:
 * /listener (http://ubuntu:34664/)
```

在启动Talker节点之后，通过Rosnode list看到增加的一个节点就是Listener，它包含了一个完整的拓扑：包含发送节点和接收节点。



ROS的Catkin编译系统

ROS是基于Cmake编写的Catkin编译系统。建立一个工程包，在ROS里面写一个节点，通过Catkin create 可以简单创建一个文件夹，这个文件夹里面会预先设置一些文件目录，通过Catkin build编译建立软件包的过程。Catkin build执行之后，里面会多两个文件夹：DEVEL、BUILD。BUILD是编译中间过程产生的文件。编译完成之后，通过Source devel下面的Setup bash就可以把自己编写的节点程序给Source到ROS的环境里面，然后去执行我们节点里面的一些基本功能。

catkin Build System

The catkin workspace contains the following spaces

Work here



SRC

The *source space* contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

Don't touch



build

The *build space* is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don't touch



devel

The *development (devel)* space is where built targets are placed (prior to being installed).

If necessary, clean the entire build and devel space with

```
> catkin clean
```

More info

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



以上是三个比较重要的文件夹，第一个是SRC用来放源文件的一些目录；第二个是BUILD，第三个是DEVEL，这两个是在Catkin build的过程当中产生的临时文件夹。想重编译的话可以直接Catkin build，如果环境里面有一些冲突，可以通过Catkin clean 简单的去把编译产生的临时文件和之前的一些产出文件直接清除掉。

Catkin config 指定了命令行编译的一些方式，这些方式可以在Cmakelists里面进行编写。Cmakelists里面指定了这个文件编译过程当中所依赖的一些库、产出的一些可执行文件和这些可执行文件链接了一些什么库，Cmakelists里面都有一些很清晰的定义。

Example

Go to your catkin workspace

```
> cd ~/catkin_ws
```

Build the package with

```
> catkin build ros package template
```

Re-source your workspace setup

```
> source devel/setup.bash
```

Launch the node with

```
> roslaunch ros_package_template
  ros_package_template.launch
```

```

Note: forcing cmake to run for each package.
[build] Found '1' packages in 0.0 seconds.
[build] Updating package table.
Starting >>> catkin_tools_prebuild
Finished <<< catkin_tools_prebuild [ 1.0 seconds ]
Starting >>> ros_package_template
Finished <<< ros_package_template [ 4.1 seconds ]
[build] Summary: All 2 packages succeeded!
[build] Ignored: None.
[build] Warnings: None.
[build] Abandoned: None.
[build] Failed: None.
[build] Runtime: 5.2 seconds total.
[build] Note: Workspace packages have changed, please re-source setup files to use them.
student@ubuntu:~/catkin_ws$

```

```
* /ros/package_template/subscriber_topic /temperature
* /rosversion: 1.11.20

NODES
/
  ros_package_template (ros_package_template/ros_package_template)

auto-starting new master
process[master]: started with pid [27185]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to e43f937a-ed52-11e6-9789-000c297bd368
process[roscout-1]: started with pid [27198]
started core service [/roscout]
process[ros_package_template-2]: started with pid [27201]
[ INFO] [1486485895.843512614]: Successfully launched node.
```



在启动节点的时候使用了Roslaunch，Roslaunch是一个Shell脚本文件，Shell脚本文件里面根据语言定义的一些Xml格式去找到运行的一系列节点所在的位置然后执行它。它的执行格式是前面加上Package Name，后面加上实际的Launch文件。



ROS的仿真功能Gazebo

ETH zürich

Gazebo Simulator

- Simulate 3d rigid-body dynamics
- Simulate a variety of sensors including noise
- 3d visualization and user interaction
- Includes a database of many robots and environments (*Gazebo worlds*)
- Provides a ROS interface
- Extensible with plugins

Run Gazebo with

```
> rosrn gazebo_ros gazebo
```

Object tree Toolbar (to navigate and new objects)

Properties Start and pause simulation

More info
<http://gazebosim.org/>
<http://gazebosim.org/tutorials>

Peter Fankhauser | 16.02.2018 | 31

这个是Gazebo的Simulator仿真工具。我们在实际进行开发，不管是机器人还是自动驾驶相关的一些具体功能的时候，我们不可能就是开发一个功能然后到实体的机器人或者是自动驾驶的汽车上去进行模拟实验。ROS提供了仿真功能Gazebo，我们定义的节点在里面是实体的存在，通过控制一些参数和变量去模拟他们之间的一些交互，去验证算法在实际的运行中是否按预期进行表现。

