

ROS 基本操作和项目代码修改

PIX 实训基地

目录

ROS 简介

1.1 简介

理解工作空间和程序包

2.1 什么是工作空间

2.2 什么是程序包

理解 ROS 节点和话题

3.1 什么是 ROS 节点

3.2 什么是 ROS 话题

3.3 创建一个自定义消息

3.4 编写一个 ROS 消息发布节点

3.5 编写一个 ROS 消息接收节点

3.6 测试发布和接收节点

修改项目源码

4.1 项目代码简介

4.2 添加数据模型

4.3 代码移植

4.4 编译与测试

修改 Autoware 源码

5.1 Autoware 代码简介

5.2 修改指定部分代码

5.2 编译并测试

帮助

ROS 简介

1.1 简介

>ROS 简介： ROS 是一个适用于机器人的开源的元操作系统。它提供了操作系统应有的服务，包括硬件抽象，底层设备控制，常用函数的实现，进程间消息传递，以及包管理。它也提供用于获取、编译、编写、和跨计算机运行代码所需的工具和库函数。在某些方面 ROS 相当于一种“机器人框架（robot frameworks）”。

> 项目简介： 为了让 Udacity 所学项目能够顺利的与真车相结合并通过测试，我们必须依靠 ROS 的帮助来完成这些工作，所以我们必须学会如何使用 ROS 这样的高效工具来完成移植的事情，了解 ROS 同样也对自己的知识面等待方向有着不可估计的帮助。

整个教程将会分成两个部分来讲解，前半部分是对 ROS 的基础介绍和使用教程，后半部分是对项目移植的详细讲解，如果你对 ROS 已经熟悉，则可以不用学习前半部分内容。



理解工作空间和程序包

2.1 什么是工作空间

catkin 工作空间是一个用于存放 ROS 程序包的地方，一个工作空间可以放很多个程序包，工作空间中的各个节点可以相互通讯而不会产生任何消息隔离，就好比是一个汽车生产公司的厂房，而厂房里面包含了很多用来生产汽车的设备，这个厂房将生产汽车的设备和生产飞机的设备分隔开来。其实它便是一个有 ROS 属性的文件夹而已。

在安装好 ROS 之后，我们可以通过以下方式来创建一个工作空间：

> 在 Home 下创建一个新的文件夹：

```
$ mkdir -p ~/catkin_ws/src
```

```
$ cd ~/catkin_ws/src
```

> cd 到该文件夹下面，并编译这个工作空间

```
$ cd ~/catkin_ws/
```

```
$ catkin_make
```

> 接下来首先 source 一下新生成的 setup.*sh 文件：

```
$ source devel/setup.bash
```

source 命令目的是为了刷新工作空间环境，要想保证工作空间已配置正确需确保 ROS_PACKAGE_PATH 环境变量包含你的工作空间目录，采用以下命令查看：

```
$ echo $ROS_PACKAGE_PATH
```

至此，一个工作空间便创建起来了。

```
up@up: ~/catkin_ws
up@up:~$ mkdir -p ~/catkin_ws/src
up@up:~$ cd ~/catkin_ws
up@up:~/catkin_ws$ ls
src
up@up:~/catkin_ws$
```

```
up@up: ~/catkin_ws
nd
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found gtest sources under '/usr/src/gmock': gtest
ts will be built
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.7.14
-- BUILD_SHARED_LIBS is on
-- Configuring done
-- Generating done
-- Build files have been written to: /home/up/catki
n_ws/build
####
#### Running command: "make -j8 -l8" in "/home/up/c
atkin_ws/build"
####
up@up:~/catkin_ws$
```

```
up@up:~/catkin_ws$ echo $ROS_PACKAGE_PATH
/home/up/Autoware/ros/src:/opt/ros/kinetic/share
up@up:~/catkin_ws$
```

理解工作空间和程序包

2.2 什么是程序包

catkin 程序包也可以称作功能包，程序包是 ROS 组织程序的主要方式，程序包相当于一个 project，一个程序包一般包含程序文件（src 文件夹中的 .cpp 和 .py 文件）、编译描述文件（package.xml）和配置文件（CMakeList.txt）等。

> 创建 catkin 程序包：

```
$ cd ~/catkin_ws/src
```

> 在这个工作空间下创建一个名叫 beginner_tutorials 的程序包：

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

这将会创建一个名为 beginner_tutorials 的文件夹，这个文件夹里面包含一个 package.xml 文件和一个 CMakeLists.txt 文件。

创建一个程序包的方式大概是这样：

```
# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

> 编译这个程序包：

```
$ cd ~/catkin_ws/
```

```
$ ls src
```

此时你会发现刚刚创建的程序包。

```
$ catkin_make
```

你会发现右边所示的输出信息。如果没有报错则表示该程序包已经编译成功了。但是这只是一个空的程序包。

```
up@up:~/catkin_ws$ cd src
up@up:~/catkin_ws/src$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
Created file beginner_tutorials/CMakeLists.txt
Created file beginner_tutorials/package.xml
Created folder beginner_tutorials/include/beginner_tutorials
Created folder beginner_tutorials/src
Successfully created files in /home/up/catkin_ws/src/beginner_tutorials. Please adjust the values in package.xml.
up@up:~/catkin_ws/src$
```

```
up@up:~$ cd catkin_ws/
up@up:~/catkin_ws$ ls src
beginner_tutorials  CMakeLists.txt
up@up:~/catkin_ws$
```

```
up@up: ~/catkin_ws
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.7.14
-- BUILD_SHARED_LIBS is on

-- traversing 1 packages in topological order:
--   - beginner_tutorials

-- +++ processing catkin package: 'beginner_tutorials'
-- ==> add_subdirectory(beginner_tutorials)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/up/catkin_ws/build
####
#### Running command: "make -j8 -ls" in "/home/up/catkin_ws/build"
####
up@up:~/catkin_ws$
```

理解 ROS 节点和话题

3.1 什么是 ROS 节点

ROS 节点是程序包中的可执行文件，由于 ROS 是分布式的操作系统，所以 ROS 下真正具体工作的实体部分都是由节点来完成的，有他们来具体的指令操作和处理数据，每一个节点都是一个独立的可执行体，同时可以通过 ROS 实现节点与节点之间的相互通讯。节点可以发布或接收一个话题。节点也可以提供或使用某种服务。

节点可以简单的理解为一个遥控器和一个机器人，遥控器是一个可以发布消息的节点，机器人是一个可以接收话题的节点，遥控器发送的消息通过特定频段的电磁波传送，这个波段只有这个遥控器节点和机器人节点才能是别的到，所以电磁波可以理解为一个消息。当遥控器节点发布一个话题的时候，同时收听这个波段的机器人节点就会接收到来自遥控器节点的消息，从而根据接收到的消息来运动。

> 客户端库：ROS 客户端库允许使用不同编程语言编写的节点之间互相通信。

rospy = python 客户端库

roscpp = c++ 客户端库

> roscore: roscore 是你在运行所有 ROS 程序前首先要运行的命令。

> 查看当前活跃的话题：

\$ rostopic list

```
roscore http://up:11311/
up@up:~/catkin_ws$ roscore
... logging to /home/up/.ros/log/292b6f92-e9a8-11e8-9413-1c1b0daf78ce/roslaunch-up-20395.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://up:44373/
ros_comm version 1.12.14

* /rosversion: 1.12.14

NODES
auto-starting new master
process[master]: started with pid [20406]
ROS_MASTER_URI=http://up:11311/

setting /run_id to 292b6f92-e9a8-11e8-9413-1c1b0daf78ce
process[rostop-1]: started with pid [20430]
started core service [/rostop]
```

```
up@up: ~
up@up:~$ rostopic list
/rostop
up@up:~$
```

理解 ROS 节点和话题

> 使用 rosrund 运行一个节点：

在运行节点之前确保自己已经安装了用于演示的 ros-kinetic-ros-tutorials 程序包。

```
$ sudo apt-get install ros-kinetic-ros-tutorials
```

启动一个小乌龟节点：

```
$ rosrund turtlesim turtlesim_node
```

启动小乌龟节点之后你变会发现活跃节点里面多了一个新的 /turtlesim 节点，这边是小乌龟节点。

> 修改要运行的节点名称：

```
$ rosrund turtlesim turtlesim_node __name:=my_turtle
```

运行这个指令之后，你会发现一个新的小乌龟被启动，同时它的名字已经被改变了。



The image displays three terminal windows and one graphical window. The top terminal window shows the command `roslaunch turtlesim turtlesim_node` being executed, with output messages indicating the start of the turtlesim node and the spawning of a turtle named 'turtle1' at coordinates (5.544445, 5.544445) with a theta of 0.000000. The middle window is the TurtleSim graphical interface, which has a blue background and a small green turtle icon at the bottom center. The bottom terminal window shows the command `roslaunch turtlesim turtlesim_node __name:=my_turtle` being executed, with output messages indicating the start of the turtlesim node and the spawning of a turtle named 'turtle1' at coordinates (5.544445, 5.544445) with a theta of 0.000000. The bottom-most terminal window shows the command `rostopic list` being executed, with output messages indicating the list of topics: /my_turtle, /rosout, and /turtlesim.

```
up@up: ~  
up@up:~$ roslaunch turtlesim turtlesim_node  
[ INFO] [1542377407.376378284]: Starting turtlesim with node name /turtlesim  
[ INFO] [1542377407.380131242]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]  
  
TurtleSim  
  
up@up: ~  
up@up:~$ roslaunch turtlesim turtlesim_node __name:=my_turtle  
[ INFO] [1542377684.640200836]: Starting turtlesim with node name /my_turtle  
[ INFO] [1542377684.643939075]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]  
  
up@up: ~  
up@up:~$ rostopic list  
/my_turtle  
/rosout  
/turtlesim
```


理解 ROS 节点和话题

3.2 什么是 ROS 话题

ROS 话题可以理解为一个特定频段地无线电，当一个遥控器向这个频段发送一个消息之后，如果有一个或者几个机器人都用了这个频段来接收消息，那么他们就会收到来自遥控器发过来的信号，但是其他没有接收这个频段的机器人就不会听到那个遥控器发出来的消息。

下面通过例子来具体了解话题是如何起作用的：

> 确保你已经运行了 roscore

```
$roscore
```

如果你没有退出在上一篇教程中运行的 roscore，那么你可能会看到下面的错误信息：

```
roscore cannot run as another roscore/master is already running.
```

```
Please kill other roscore/master processes before relaunching
```

这是正常的，因为只需要有一个 roscore 在运行就够了。

> 启动小乌龟节点，如果在上一个教程中没有推出则可以不用运行。

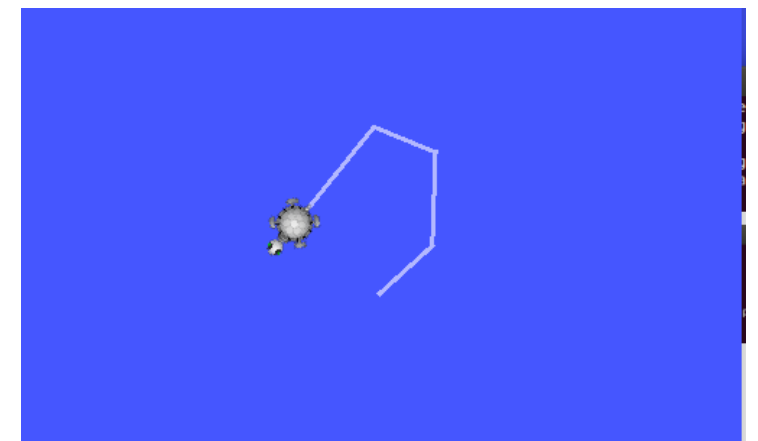
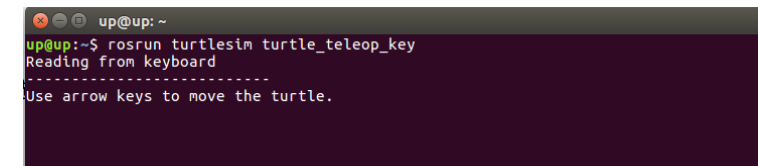
```
$ rosrun turtlesim turtlesim_node
```

> 运行一个新的键盘遥控器节点，便可以通过它来控制小乌龟移动。

```
$ rosrun turtlesim turtle_teleop_key
```

此时，你可以使用键盘上的方向键来控制 turtle 运动了。如果不能控制，请选中 turtle_teleop_key 所在的终端窗口以确保你的按键输入能够被捕获。

可以看到，这个时候小乌龟会随着你的按键移动，这是因为刚刚运行的遥控器节点读取了你的指令并通过话题发布给了小乌龟这个节点。



理解 ROS 节点和话题

> 通过 rqt_graph 查看节点之间的联系：

```
$ roslaunch rqt_graph rqt_graph
```

从节点图中可以看到，一个名叫 /teleop_turtle 的节点向 turtle1/cmd_vel 话题发布了数据，而另外一个节点 /turtlesim 从这个话题中接收了数据。这两个节点分别是键盘遥控器个小乌龟。其中椭圆形为节点，方框为话题，箭头指向表示数据流方向。

> 通过 topic echo /topic 查看相应话题发布的内容：

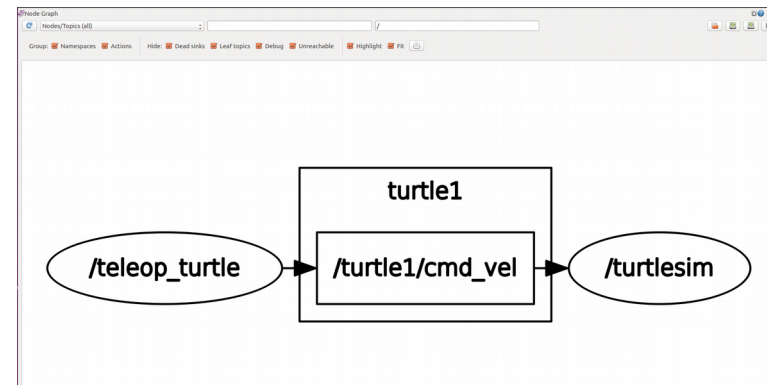
```
$ rostopic echo /turtle1/cmd_vel
```

刚刚打开这个命令窗口的时候可能没有数据发出来，当你找到之前那个运行遥控器的终端，在他激活的情况下按下方向键，此时再调取查看话题的终端时变回看到相应的数据被显示出来。这些数据便是遥控器节点发布出来的数据。

> 通过 topic info /topic 查看相应话题详细参数：

```
$ rostopic info /turtle1/cmd_vel
```

可以看到，在输入该命令之后你会发现终端中打印出来了关于这个话题的详细信息，包括它的发布者，订阅者和话题类型。话题类型就是如果要往这个话题内发布数据应该用的数据类型，如整型，浮点型或者数据结构以及其他自定义的数据类型等。



```
up@up:~$ rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

```
up@up:~$ rostopic info /turtle1/cmd_vel
Type: geometry_msgs/Twist

Publishers:
* /teleop_turtle (http://up:40155/)

Subscribers:
* /turtlesim (http://up:33639/)
* /rostopic_5982_1542596459465 (http://up:34799/)

up@up:~$
```

理解 ROS 节点和话题

> 向话题中发布数据：

> 使用 `rostopic pub` 向某个话题中发布数据：

格式：`rostopic pub [topic] [msg_type] [args]`

`$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'`

以上命令会发送一条消息给 `turtlesim`，告诉它以 2.0 大小的线速度和 1.8 大小的角速度开始移动。

> 命令解析：

`rostopic pub`：这条命令将会发布消息到某个给定的话题。

`-1`：（单个破折号）这个参数选项使 `rostopic` 发布一条消息后马上退出。

`/turtle1/command_velocity`：这是消息所发布到的话题名称。

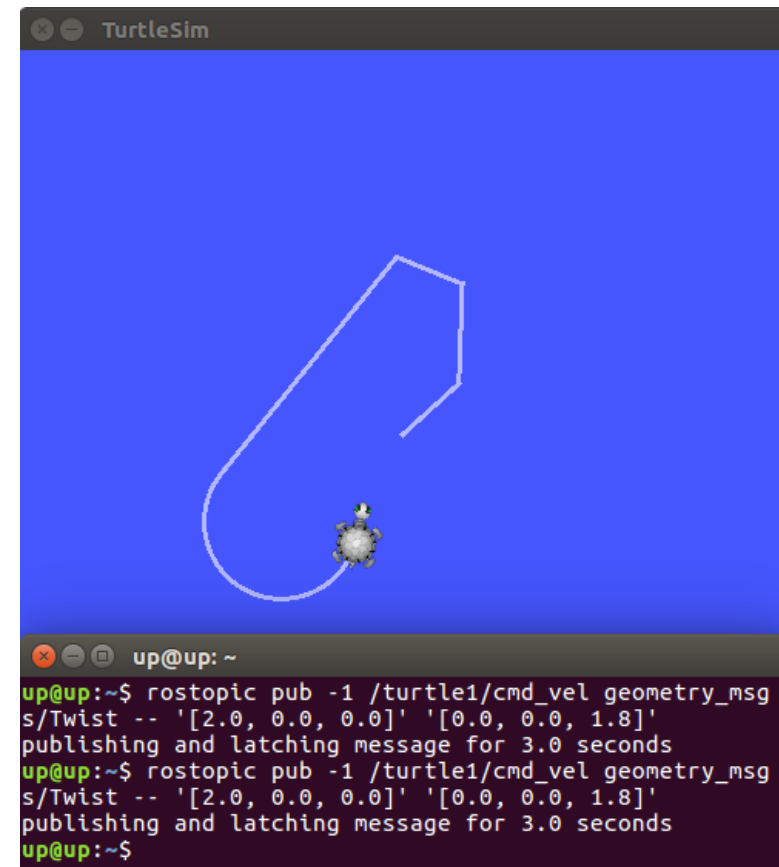
`turtlesim/Velocity`：这是所发布消息的类型。

`--`：（双破折号）这会告诉命令选项解析器接下来的参数部分都不是命令选项。这在参数里面包含有破折号 -（比如负号）时是必须要添加的。

`2.0 1.8`：正如之前提到的，在一个 `turtlesim/Velocity` 消息里面包含有两个浮点型元素：`linear` 和 `angular`。在本例中，2.0 是 `linear` 的值，1.8 是 `angular` 的值。这些参数其实是按照 YAML 语法格式编写的，这在 YAML 文档中有更多的描述。

> 使用 `rostopic hz` 查看话题数据发布的频率：

`$ rostopic hz /turtle1/pose`



理解 ROS 节点和话题

- > 使用 `rostopic help` 来获得更多帮助：
 `$ rostopic help`
 会打印出很多可以查看话题更多信息的命令和解释。
- > 使用 `rostopic type /topic` 查看相应话题的消息类型：
 `$ rostopic type /turtle1/cmd_vel`
 通过这个话题可以查看相应话题的数据类型。
- > 使用 `rosmmsg` 查看消息的详细数据：
 `$ rosmmsg help`
 输入命令系统将会打印出 `rosmmsg` 提供的命令和注释
- > 使用 `rosmmsg show` 查看指定消息详细情况：
 `$ rosmmsg show geometry_msgs/Twist`
 可以看到，之前话题上发布出来的话题类型和该消息类型格式保持一致。

```
up@up: ~  
up@up:~$ rostopic help  
rostopic is a command-line tool for printing information about ROS Topics.  
  
Commands:  
  rostopic bw      display bandwidth used by topic  
  rostopic delay   display delay of topic from timestamp in header  
  rostopic echo    print messages to screen  
  rostopic find    find topics by type  
  rostopic hz      display publishing rate of topic  
  rostopic info    print information about active topic  
  rostopic list    list active topics  
  rostopic pub     publish data to topic  
  rostopic type    print topic or field type  
  
Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'  
up@up:~$
```

```
up@up: ~  
up@up:~$ rostopic type /turtle1/cmd_vel  
geometry_msgs/Twist  
up@up:~$
```

```
up@up:~$ rosmmsg show geometry_msgs/Twist  
geometry_msgs/Vector3 linear  
  float64 x  
  float64 y  
  float64 z  
geometry_msgs/Vector3 angular  
  float64 x  
  float64 y  
  float64 z  
up@up:~$
```

```
up@up:~$ rosmmsg -h  
rosmmsg is a command-line tool for displaying information about ROS Message types  
.  
  
Commands:  
  rosmmsg show    Show message description  
  rosmmsg info    Alias for rosmmsg show  
  rosmmsg list    List all messages  
  rosmmsg md5     Display message md5sum  
  rosmmsg package List messages in a package  
  rosmmsg packages List packages that contain messages  
  
Type rosmmsg <command> -h for more detailed usage
```

理解 ROS 节点和话题

3.3 创建一个自定义消息

消息是可以自己创建的，下面，我们将在之前创建的 package 里定义新的消息。

```
$ cd ~/catkin_ws/src/beginner_tutorials
```

```
$ mkdir msg
```

```
$ echo "int64 num" > msg/Num.msg
```

命令分别表示：cd 到之前创建的那个程序包中，重新创建一个名叫 :msg 的文件夹用来装消息文件，创建一个名叫 Num.msg 的文件并写入 int64 num 这句话。上面是最简单的例子——在 .msg 文件中只有一行数据。当然你还可以往这个 msg 文件中添加跟多的类型。

接下来，还有关键的一步：我们要确保 msg 文件被转换成为 C++，Python 和其他语言的源代码，查看 package.xml，确保它包含了右边所示语句。

如果没有，添加进去。注意，在构建的时候，我们只需要 "message_generation"。然而，在运行的时候，我们只需要 "message_runtime"。

```
up@up: ~/catkin_ws/src/beginner_tutorials
up@up:~$ cd ~/catkin_ws/src/beginner_tutorials/
up@up:~/catkin_ws/src/beginner_tutorials$ mkdir msg
up@up:~/catkin_ws/src/beginner_tutorials$ ls
CMakeLists.txt  include  msg  package.xml  src
up@up:~/catkin_ws/src/beginner_tutorials$ echo "int64 num">msg/Num.msg
up@up:~/catkin_ws/src/beginner_tutorials$
```

```
string first_name
string last_name
uint8 age
uint32 score
```

```
up@up: ~/catkin_ws/src/beginner_tutorials
<!-- <depend>roscpp</depend> -->
<!-- Note that this is equivalent to the following: -->
<!-- <build_depend>roscpp</build_depend> -->
<!-- <exec_depend>roscpp</exec_depend> -->
<!-- Use build_depend for packages you need at compile time: -->
<build_depend>message_generation</build_depend>
<!-- Use build_export_depend for packages you need in order to build against t
his package: -->
<!-- <build_export_depend>message_generation</build_export_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!-- <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use exec_depend for packages you need at runtime: -->
<exec_depend>message_runtime</exec_depend>
<!-- Use test_depend for packages you need only for testing: -->
<!-- <test_depend>gtest</test_depend> -->
<!-- Use doc_depend for packages you need only for building documentation: -->
<!-- <doc_depend>doxygen</doc_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
```

35,1

73%

理解 ROS 节点和话题

> 打开 CMakeLists.txt 文件，并修改之：

在 CMakeLists.txt 文件中，利用 find_package 函数，增加对 message_generation 的依赖，这样就可以生成消息了。你可以直接在 COMPONENTS 的列表里增加 message_generation，就像右边一样。

有时候你会发现，即使你没有调用 find_package，你也可以编译通过。这是因为 catkin 把你所有的 package 都整合在一起，因此，如果其他的 package 调用了 find_package，你的 package 的依赖就会是同样的配置。但是，在你单独编译时，忘记调用 find_package 会很容易出错。

同样，你需要确保你设置了运行依赖，如右边所示：

找到如下代码块：

```
# add_message_files(  
#   FILES  
#   Message1.msg  
#   Message2.msg  
# )
```

掉注释符号 #，用你的 .msg 文件替代 Message*.msg，就像下边这样：

```
add_message_files(  
  FILES  
  Num.msg  
)
```

```
up@up: ~/catkin_ws/src/beginner_tutorials  
  
## Find catkin macros and libraries  
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS  
## is used, also find other catkin packages  
find_package(catkin REQUIRED COMPONENTS  
  roscpp  
  rospy  
  std_msgs  
  message_generation  
)
```

```
up@up: ~/catkin_ws/src/beginner_tutorials  
  
## CATKIN_DEPENDS: catkin_packages dependent projects also  
## DEPENDS: system dependencies of this project that depend  
d  
catkin_package(  
#   INCLUDE_DIRS include  
#   LIBRARIES beginner_tutorials  
#   CATKIN_DEPENDS message_runtime roscpp rospy std_msgs  
#   DEPENDS system_lib  
)  
  
#####
```

```
up@up: ~/catkin_ws/src/beginner_tutorials  
  
## Generate messages in the 'msg' folder  
# add_message_files(  
#   FILES  
#   Message1.msg  
#   Message2.msg  
# )
```

```
up@up: ~/catkin_ws/src/beginner_tutorials  
  
## Generate messages in the 'msg' folder  
add_message_files(  
  FILES  
  Num.msg  
)
```

理解 ROS 节点和话题

手动添加 .msg 文件后，我们要确保 CMake 知道在什么时候重新配置我们的 project。确保 CMake 文件中添加了如下代码：

```
generate_messages( )
```

添加完成之后重新编译程序包就可以使用自定义消息了。

> 使用 rosmmsg 命令查看是否创建成功：

以上就是你创建消息的所有步骤。下面通过 rosmmsg show 命令，检查 ROS 是否能够识消息。

使用方式：\$ rosmmsg show [message type]

打开终端输入如下命令：

\$ rosmmsg show beginner_tutorials/Num

你会看到如是消息被打印出来。在上边的样例中，消息类型包含两部分：

beginner_tutorials -- 消息所在的 package

Num -- 消息名 Num.

如果你忘记了消息所在的 package，你也可以省略掉 package 名。输入

:

\$ rosmmsg show Num

```
up@up: ~/catkin_ws/src/beginner_tutorials
# )
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
    std_msgs
)
```

```
up@up:~/catkin_ws$ rosmmsg show beginner_tutorials/Num
int64 num
up@up:~/catkin_ws$
```

```
up@up:~/catkin_ws$ rosmmsg show Num
[beginner_tutorials/Num]:
int64 num
up@up:~/catkin_ws$
```


理解 ROS 节点和话题

3.4 编写一个 ROS 消息发布者节点 (python)

发布者节点主要的作用就是发布消息的，相当于之前比喻的遥控器，当然 ROS 强大的生态中有很多这样的发布者可以提供给我们使用，同样我们也可以编写自己的节点。

> 进入之前创建的 /beginner_tutorials 程序包下面，这里可以使用 roscd 直接索引到该文件夹下面，如果你没有吧该工作空间的 bash 文件加载到系统中，需要手动 source 这个 bash 文件。

```
$ roscd beginner_tutorials
```

> 创建新的用于存放节点程序的文件夹

```
$ mkdir scripts
```

```
$ cd scripts
```

> 创建新的程序，这里为了介绍，你可以直接下载源代码：

```
$ wget
```

https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/talker.py

```
$ chmod +x talker.py # 赋予权限
```

> 查看该文件内容：

```
$ roscd beginner_tutorials talker.py
```

> 具体代码说明请查阅 ROS 官方 wiki：

<http://wiki.ros.org/cn/ROS/Tutorials/WritingPublisherSubscriber%28python%29>

```
up@up:~/catkin_ws$ source devel/setup.bash
up@up:~/catkin_ws$ roscd beginner_tutorials/
up@up:~/catkin_ws/src/beginner_tutorials$
```

```
Toggle line numbers
1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)
8     rospy.init_node('talker', anonymous=True)
9     rate = rospy.Rate(10) # 10hz
10    while not rospy.is_shutdown():
11        hello_str = "hello world %s" % rospy.get_time()
12        rospy.loginfo(hello_str)
13        pub.publish(hello_str)
14        rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```


理解 ROS 节点和话题

3.5 编写一个 ROS 消息订阅器节点 (python)

> 重新 cd 到新创建的程序包中:

```
$ roscd beginner_tutorials/scripts/
```

> 下载订阅器节点的源码:

```
$ wget
```

https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/listener.py

> 赋值权限:

```
$ chmod +x listener.py
```

> 打开之后如下图所示, 具体代码解释参见 ROS 官方 wiki 解说。

> 编译程序包:

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

如果编译成功说明发布器和接收器已经创建成功。

Toggle line numbers

```
1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)
8     rospy.init_node('talker', anonymous=True)
9     rate = rospy.Rate(10) # 10hz
10    while not rospy.is_shutdown():
11        hello_str = "hello world %s" % rospy.get_time()
12        rospy.loginfo(hello_str)
13        pub.publish(hello_str)
14        rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

```
up@up: ~/catkin_ws
Source space: /home/up/catkin_ws/src
Build space: /home/up/catkin_ws/build
Devel space: /home/up/catkin_ws/devel
Install space: /home/up/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/up/catkin_ws/build"
####
####
#### Running command: "make -j8 -l8" in "/home/up/catkin_ws/build"
####
[ 0%] Built target std_msgs_generate_messages_nodejs
[ 0%] Built target std_msgs_generate_messages_cpp
[ 0%] Built target std_msgs_generate_messages_py
[ 0%] Built target std_msgs_generate_messages_lisp
[ 0%] Built target std_msgs_generate_messages_eus
[ 0%] Built target _beginner_tutorials_generate_messages_check_deps_Num
[ 28%] Built target beginner_tutorials_generate_messages_eus
[ 42%] Built target beginner_tutorials_generate_messages_lisp
[ 71%] Built target beginner_tutorials_generate_messages_py
[ 85%] Built target beginner_tutorials_generate_messages_nodejs
[100%] Built target beginner_tutorials_generate_messages_cpp
[100%] Built target beginner_tutorials_generate_messages
up@up:~/catkin_ws$
```

理解 ROS 节点和话题

3.6 测试消息发布器和订阅器

在创建好发布器和订阅器之后，我们就可以来运行他们了。

> 打开 roscore;

```
$ roscore
```

> 运行发布器节点:

```
$ cd ~/catkin_ws
```

```
$ source devel/setup.bash
```

```
$ rosrun beginner_tutorials talker.py
```

此时你会看到这个节点会不断的发不出信息。

> 运行订阅器节点:

```
$ rosrun beginner_tutorials listener.py
```

在刚刚发布器节点没有退出来的情况下，你会看到右边所示消息，当你打开 rqt_graph 的时候，你会看到这两个节点通过话题连接在一起了。

```
roscore http://up:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://up:42925/
ros_comm version 1.12.14

SUMMARY
=====
PARAMETERS
* /roscore: kinetic
* /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [29624]
ROS_MASTER_URI=http://up:11311/

setting /run_id to 14cc6692-ebf0-11e8-97fb-1c1b0daf78ce
process[roscout-1]: started with pid [29648]
started core service [/roscout]
```

```
up@up:~/catkin_ws$ rosrun beginner_tutorials talker.py
[INFO] [1542627718.574007]: hello world 1542627718.57
[INFO] [1542627718.674427]: hello world 1542627718.67
[INFO] [1542627718.774403]: hello world 1542627718.77
[INFO] [1542627718.874406]: hello world 1542627718.87
[INFO] [1542627718.974419]: hello world 1542627718.97
[INFO] [1542627719.074194]: hello world 1542627719.07
[INFO] [1542627719.174198]: hello world 1542627719.17
[INFO] [1542627719.274204]: hello world 1542627719.27
[INFO] [1542627719.374427]: hello world 1542627719.37
[INFO] [1542627719.474203]: hello world 1542627719.47
[INFO] [1542627719.574186]: hello world 1542627719.57
[INFO] [1542627719.674197]: hello world 1542627719.67
[INFO] [1542627719.774423]: hello world 1542627719.77
[INFO] [1542627719.874413]: hello world 1542627719.87
[INFO] [1542627719.974400]: hello world 1542627719.97
[INFO] [1542627720.074502]: hello world 1542627720.07
```

```
up@up:~/catkin_ws$ rosrun beginner_tutorials listener.py
[INFO] [1542628275.328890]: /listener_31257_1542628275032I heard hello world 154
2628275.33
[INFO] [1542628275.428628]: /listener_31257_1542628275032I heard hello world 154
2628275.43
[INFO] [1542628275.529044]: /listener_31257_1542628275032I heard hello world 154
2628275.53
[INFO] [1542628275.628901]: /listener_31257_1542628275032I heard hello world 154
2628275.63
[INFO] [1542628275.729051]: /listener_31257_1542628275032I heard hello world 154
2628275.73
[INFO] [1542628275.828834]: /listener_31257_1542628275032I heard hello world 154
2628275.83
[INFO] [1542628275.928872]: /listener_31257_1542628275032I heard hello world 154
2628275.93
[INFO] [1542628276.028771]: /listener_31257_1542628275032I heard hello world 154
2628276.03
[INFO] [1542628276.129049]: /listener_31257_1542628275032I heard hello world 154
2628276.13
[INFO] [1542628276.228559]: /listener_31257_1542628275032I heard hello world 154
```

修改项目源码

4.1 项目代码介绍

本次实训中，你会用到之前你在 Udacity 中学到的一个项目里面的部分内容，或许你可能还没有完成最后一期自动驾驶的课程，但是这里需要用到这部分代码来实现部分功能，不过不用担心，这里我们会教你怎么一点点的去移植这部分程序到 Autoware 中，并且让他在你训练的模型中正常的运行起来。

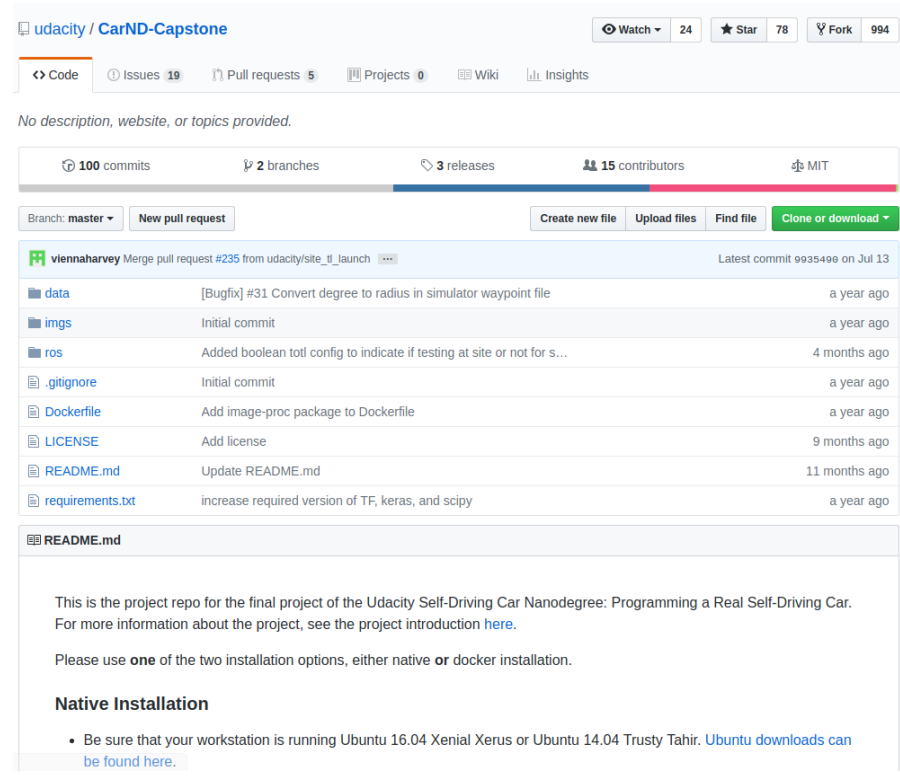
这部分代码是 Udacity 无人驾驶 课程第三期的最后一个毕业项目取出来的代码，

链接：<https://github.com/udacity/CarND-Capstone>

这个项目主要是利用模拟器将之前学习的机器学习等课程内容运用在自动驾驶汽车红绿灯识别中，我们截取的这部分代码正式这个项目的核心部分 :tl_detector 部分，也就是检测部分。由于之前这个部分是结合模拟器来跑的，所以要把它分离出来，并把它加入到 Autoware 中。

很高兴的是从原生项目分离的这部分工作已经由我们工作人员解决了，现在我们需要做的是：

1. 如何将这部分代码加入 Autoware 中
2. 重新加入我们自己训练好的模型
3. 确保这个程序能在新环境下面很好的结合
4. 不断的优化它。

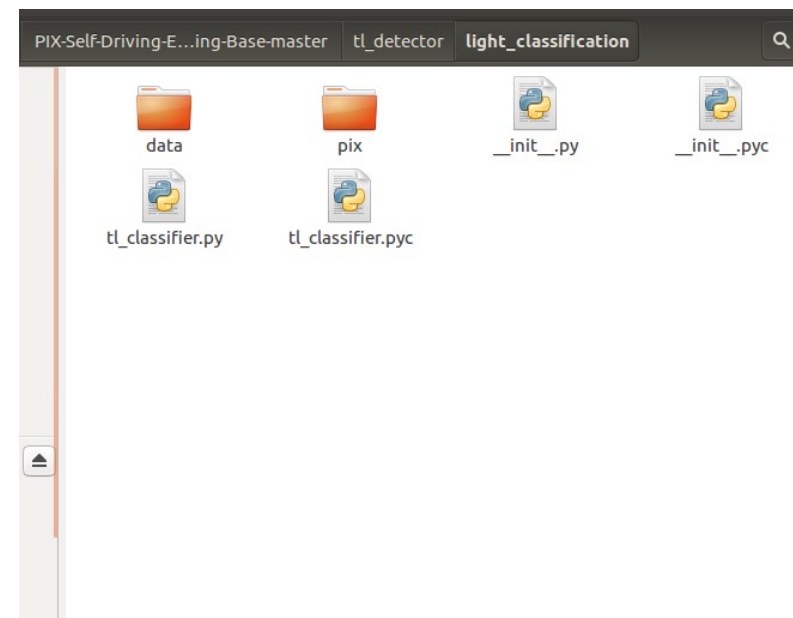
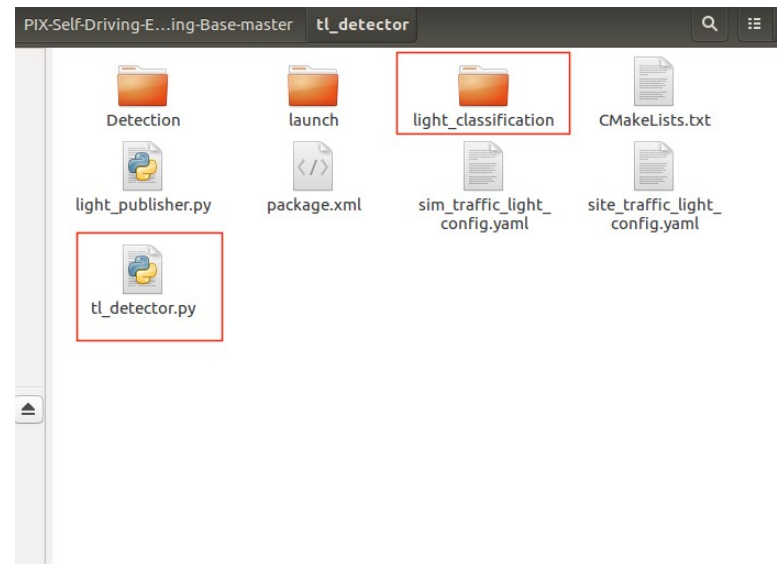


修改项目源码

> 代码结构简介：

通过我们提供的 github 网页你可以很容易的下载到 `tl_detector`，这个部分代码便是和 Udacity 课程直接联系的一部分，其中主要包含了几个部分：

- `tl_detector.py`：这个文件是我们测试中较为核心的一部分，因为这里改动地方很多，但是你也可以不需要进行改动，因为我们已经提前做好了工作，如果你需要优化你的代码，那么修改这部分代码是可行的。
- `light_classification`：该文件夹下面包好了两个部分的内容，一个部分是 `tl_detector.py` 需要调用的分类器的部分代码，在 `tl_classifier` 可以找到对应源码，另一部分是需要将你按照规定格式训练好的分类模型放到 `pix` 文件夹下面，这样才能让这个分类器进行正常的工作。
- 其他部分的代码可以自行优化，因为考虑到可操作性我们是在原项目的基础之上进行的简单改动，优化还行你自行完成，同时也保存了之前遗留下来的部分代码，优化过程中可以去除来提高分类器的性能。



修改项目源码

>tl_detector 代码解析：

tl_detector 其实是 ROS 下的一个程序包，他包含了 tl_detector 这个主要的节点，该节点会订阅相关话题，并通过训练好的模型来进行红绿灯分类，但是我们只提供了一个最原始的源码，如果你还需要让它变得更强大，或者更多的功能，比如怎讲红绿灯距离检测，停止线等等功能，则需要你自行优化，同时我们保留了大量的原生代码提供参考，下面介绍改动部分和相关部分代码。

- 话题订阅：初始 tl_detector 节点会订阅两个话题，一个是相机节点，用于接收图像信息，一个是当前位置节点，这个节点可以从 Autoware 中活动车辆当前位置信息。

```
sub1 = rospy.Subscriber('/current_pose', PoseStamped, self.pose_cb)
#sub2 = rospy.Subscriber('/base_waypoints', lane, self.waypoints_cb)
#sub3 = rospy.Subscriber('/vehicle/traffic_lights', TrafficLightArray, self.traffic_cb)
sub6 = rospy.Subscriber('/camera0/image_raw', Image, self.image_cb)
```

- image_cb 函数：这是一个回调函数，当 /camera0/image_raw 话题下有数据的时候便会调用这个话题进行相关处理。

```
def image_cb(self, msg):
```

- get_light_state 函数：该函数为获取红绿灯状态的函数，这里会调用分类器进行红绿灯识别，最后返回一个红绿灯状态，同时会利用新创建的一个话题中发布出这个红绿灯的状态。

```
def get_light_state(self, light=None):
    self.traffic_light_state.publish(state)
    return state
```

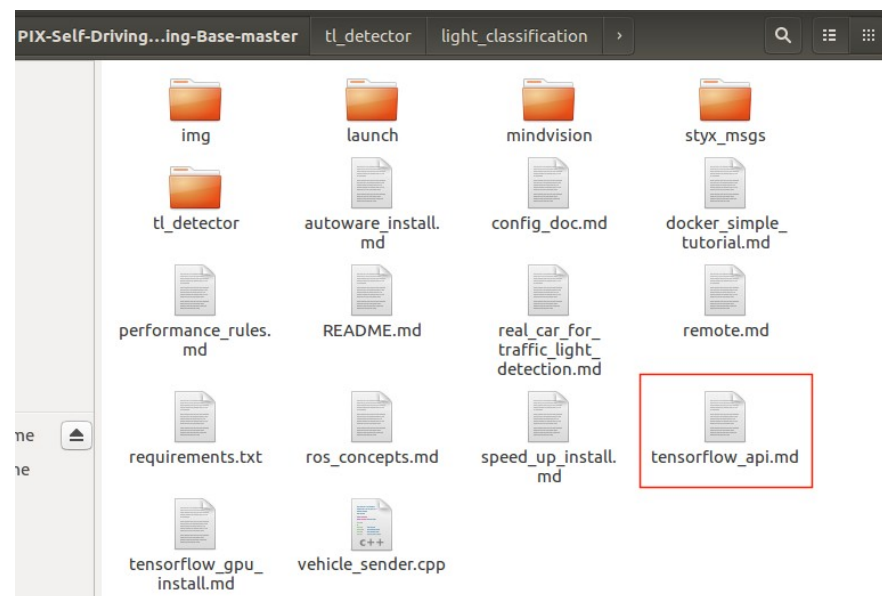
- process_traffic_lights 函数为该节点的主要程序，它会在 image_cb 回调函数中被调用，同时它会调用 get_light_state 函数进行红绿灯识别操作。

```
def process_traffic_lights(self):
```

修改项目源码

4.2 添加数据模型

按照提供教程完成采集数据，打标签，训练目标模型之后，将训练好的模型放入指定的位置，为真车红绿灯识别提供数据支撑。详细的模型创建教程和目标位置存放请查看 [github](#) 教程下的 `tensorflow_api.md` 教程。



修改项目源码

4.3 代码移植

> 添加 tl_detector 和 styx_msgs 文件夹：

通过之前的工作，我们已经基本完成了对 tl_detector 的修改和学习，现在我们需要将这个文件夹移植到 Autoware 的工作空间之中。

将下载下来的 tl_detector 文件夹和 styx_msgs 文件夹保存到路径为：~/Autoware/ros/src 文件夹下面，简单介绍：Autoware/ros 为 Autoware 的 ROS 工作空间，文件夹 src 为该工作空间下保存程序包的地方。

styx_msgs 文件夹，为原生项目中用到的自定义消息，同样 tl_detector 也使用了，所以需要一起包含进来。

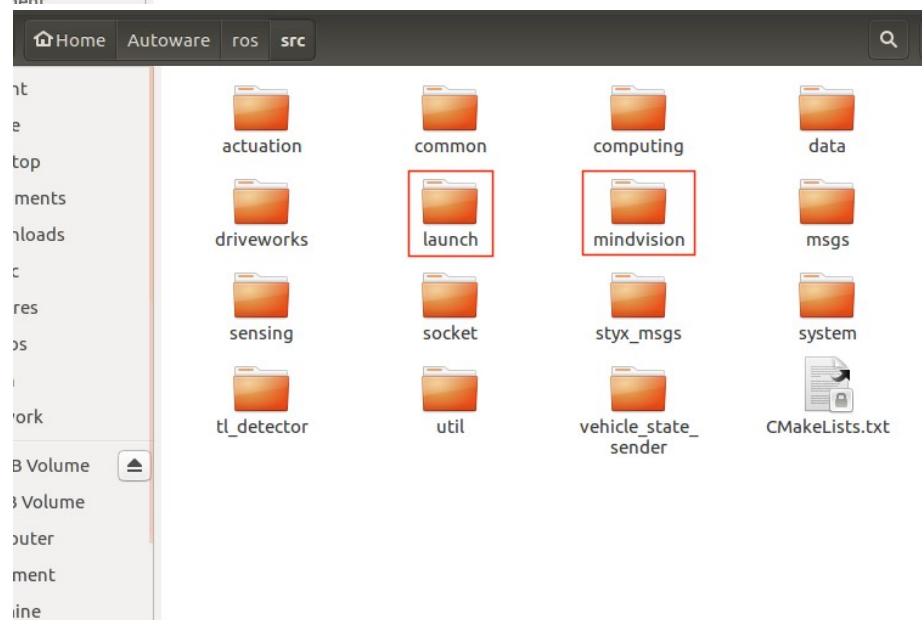
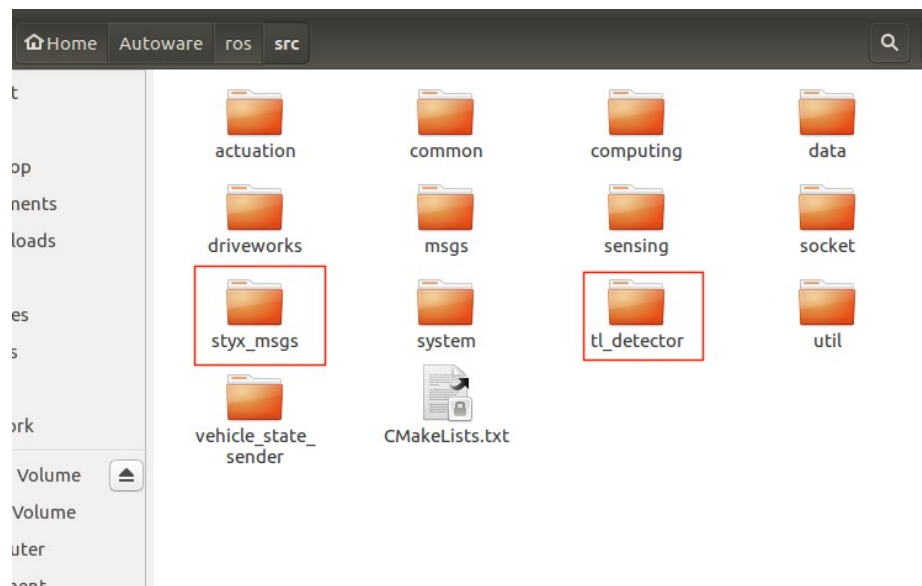
> 添加 mindvision 和 launch：

由于实训这款相机是我们自己的相机，在 Autoware 中没有相应的驱动程序提供，所以我们需要添加一个名叫：mindvision 驱动节点到 Autoware 中以驱动相机硬件工作。同时将该文件夹 mindvision/3rdparty/lib 下的 libMVSDK.so 文件加载到根菜单的 lib 文件夹下面（终端使用 cp 命令，获得权限并复制）。

launch 文件夹里面包含了一个 tl_detector launch 启动文件，用来启动 tl_detector 节点和配置相关环境。

从 github 教程中下载这两个文件夹并移动到路径为：

~/Autoware/ros/src 文件夹下面，这个程序包主要的作用是驱动相机，然后将图像数据发布到 /camera0/image_raw 上面，为所需节点提供图像数据。



修改项目源码

4.3 编译与测试

在添加好相应文件之后，下一步便是编译我们新组建好的 Autoware 啦。

> cd 到 ~/Autoware/ros 下

> 使用 ./catkin_make_release 脚本进行编译

> 等待编译成功

> 在完成编译之后，运行相机和检测器节点，测试是否可以正常工作：

> 启动相机节点：

\$ cd ~/Autoware/ros

\$ sudo su

输入密码获得超级权限

\$ source devel/setup.bash

\$ rosrun mindvision mindvision_node(保证 roscore 驱

动)

\$ rostopic echo /camera0/image_raw

查看该话题时候正确输出数据。

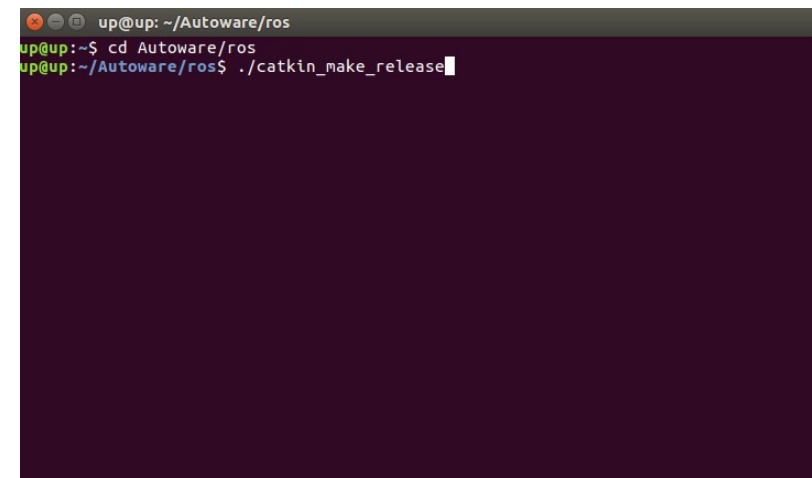
> 启动 tl_detector 节点：

\$ cd ~/Autoware/ros

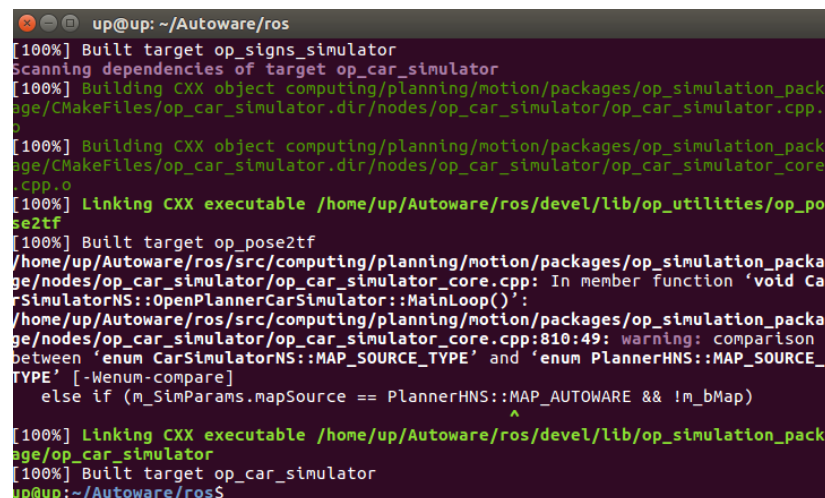
\$ source devel/setup.bash

\$ roslaunch src/launch/run_detector.launch

如果没有运行起来或者有错误提示，确保之前的环境配置成功



```
up@up: ~/Autoware/ros
up@up:~$ cd Autoware/ros
up@up:~/Autoware/ros$ ./catkin_make_release
```



```
up@up: ~/Autoware/ros
[100%] Built target op_signs_simulator
Scanning dependencies of target op_car_simulator
[100%] Building CXX object computing/planning/motion/packages/op_simulation_package/CMakeFiles/op_car_simulator.dir/nodes/op_car_simulator/op_car_simulator.cpp.o
[100%] Building CXX object computing/planning/motion/packages/op_simulation_package/CMakeFiles/op_car_simulator.dir/nodes/op_car_simulator/op_car_simulator_core.cpp.o
[100%] Linking CXX executable /home/up/Autoware/ros/devel/lib/op_utilities/op_pose2tf
[100%] Built target op_pose2tf
/home/up/Autoware/ros/src/computing/planning/motion/packages/op_simulation_package/nodes/op_car_simulator/op_car_simulator_core.cpp: In member function 'void CarSimulatorNS::OpenPlannerCarSimulator::MainLoop()':
/home/up/Autoware/ros/src/computing/planning/motion/packages/op_simulation_package/nodes/op_car_simulator/op_car_simulator_core.cpp:810:49: warning: comparison between 'enum CarSimulatorNS::MAP_SOURCE_TYPE' and 'enum PlannerHNS::MAP_SOURCE_TYPE' [-Wenum-compare]
    else if (m_SinParams.mapSource == PlannerHNS::MAP_AUTOWARE && !m_bMap)
                                         ^
[100%] Linking CXX executable /home/up/Autoware/ros/devel/lib/op_simulation_package/op_car_simulator
[100%] Built target op_car_simulator
up@up:~/Autoware/ros$
```

修改 Autoware 源码

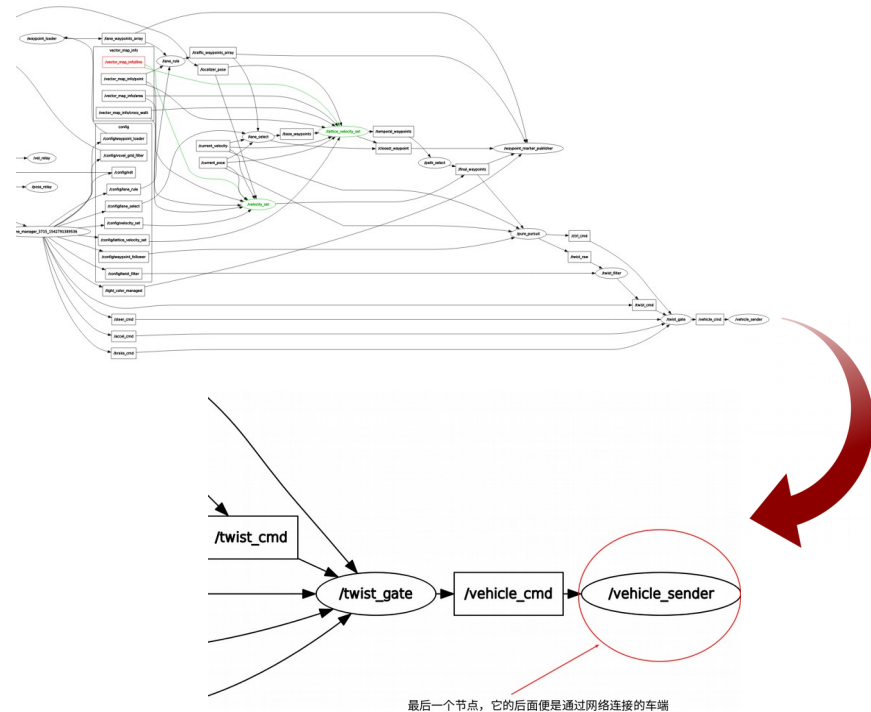
5.1 Autoware 代码简介

Autoware 其实可以理解为一个 ROS 功能包的大集合，里面包含了很多如设备驱动节点，路径规划节点，预测和跟随等功能包，我们在运行 Autoware 之后便可通过可视化界面来选择这些节点并启动。

如此之多的功能包我们该如何修改呢，其实要通过红绿灯状态来修改发送给车辆的速度转向等信息可以从很多地方下手，如果你不知道从何入手，你可以在运行 Autoware 并让汽车实现无人驾驶的情况下，打开 rqt_graph，你便可以掌握节点之间的联系，从而知道他们是怎么一步一步来控制汽车的了。

这里我们提供一个方案，便是修改路径为：/home/[User]/Autoware/ros/src/socket/packages/vehicle_socket/nodes/vehicle_sender 的一个节点，从节点图中我们可以看到它是最后一个节点。

该节点主要作用是接收来自话题 /vehicle_cmd 的数据，然后下发给车辆。



修改 Autoware 源码

5.2 修改指定部分的代码

通过上一节的介绍，现在我们准备对 vehicle_sender 节点进行改造，让它能接收到来自 tl_detector 的红绿灯消息。

- > 从 github 实训教程中下载 vehicle_sender.cpp 文件。并替换掉原来路径为：/home/up/Autoware/ros/src/socket/packages/vehicle_socket/nodes/vehicle_sender 的相同文件，这个文件已经是我们提前修改过的文件，你可以在此基础上修改。
- > 用你喜欢的编辑器打开该 cpp 文件。

修改后的代码简介：

在修改后的文件中，我们添加了一个订阅红绿灯状态话题的回调函数，并添加了简单的通过红绿灯状态值的框架，同时具备在你没有任何改动的情况下便可以实现识别红绿灯并做出反应的能力。但是只是一个框架所以准确率和容错率非常的，你必须在此基础上不断修改和优化这部分代码。

- > 添加用于存放红绿灯状态值的宏变量：

```
//-----  
int32_t tls = 2, tls_cont=0; //init green light  
//-----
```

- > 修改发送指令函数，接收红绿灯状态值并通过该值改变车速。

```
static void *sendCommand(void *arg)  
{  
    //-----  
    extern int32_t tls;  
    //tls = 0 --> red light  
    if(tls==0){  
        command_data.linear_x = 0 ;  
        oss << command_data.linear_x << " ";  
        oss << command_data.angular_z << " ";  
    }  
}
```

修改 Autoware 源码

> 添加用于监听发布红绿灯状态话题的回调函数，并做简单的滤出误判处理：

```
//added for dosen use get traffic static
void chattercallback(const std_msgs::Int32::ConstPtr& msg)
{
    extern int32_t tls,tls_cont;

    tls_cont=msg->data;
    if (msg->data!=0){
        tls_cont++;
        if(tls_cont>30){
            tls =msg->data;
            tls_cont = 0;
        }
    }else
        tls = msg->data;
}
```

> 添加一个声明新创建的回调函数的指令：

```
//-----
ros::Subscriber subb = n.subscribe("/chatter",5,chattercallback);
//-----
```

代码修改优化完成后便可进行最后一步操作：编译和测试你的代码。

修改 Autoware 源码

5.3 编译并测试

在两部分的代码都已经修改好之后，你便需要编译整个项目。

>cd 到 ~/Autoware/ros 下

```
$ cd ~/Autoware/ros
```

> 使用 ./catkin_make_release 编译源文件：

```
$ ./catkin_make_release
```

> 测试你代码的模型的准确率：

在编译完成之后，你便需要上车去测试你的代码和模型准确率了，记得将相机，检测和 Autoware 都配置完成。并且通过 ROS 话题，节点工具来检测你的红绿灯识别状态。

> 启动相机节点：

```
$ cd ~/Autoware/ros
```

```
$ sudo su
```

输入密码获得超级权限

```
$ source devel/setup.bash
```

```
$ rosrun mindvision mindvision_node( 保证 roscore 驱动 )
```

```
$ rostopic echo /camera0/image_raw
```

修改 Autoware 源码

> 启动分类器:

```
$ cd ~/Autoeare/ros
```

```
$ source devel/setup.bash
```

```
$ roslaunch src/launch/run_detector.launch
```

> 启动 Autoware:

```
$ cd ~/Autoware/ros
```

```
$ ./run
```

> 配置你的 Autoware 实现自动驾驶，并确保红绿灯检测奏效。

帮助

- > 开启新终端并运行： `$ rosrun rqt_graph rqt_graph` 可以让你看到节点，话题之间的详细联系。
- > 开启新终端并运行： `$ rosrun rqt_image_view rqt_image_view` 在弹出的视框之中配置好相机对应话题，可以让你看到相机实时拍摄下来的图像信息。（同样 Rviz 中也提供了类似功能）
- > 如果出现类似节点运行但是不起反应的时候，通过 `rostopic echo /topic` 来查看订阅的话题中是否有数据。
- > 备份你优化之前并测试成功的代码