7 Администрирование ROS

Пакеты в ROS. Установка и сборка. Утилита Catkin.

Пакет (Package)

Пакет является одной из основных единицей ROS. Любое приложение в ROS оформляется в пакет, в котором определяются: конфигурация, необходимые программы, зависимости от других пакетов ROS, системные зависимости.

Работа с пакетами ROS очень похожа на работу с пакетами linux. Пакет ROS можно поставить готовым из репозитория пакетов, так и скачать и скомпилировать из исходных кодов.

В данный момент написано и доступно для использования более 6000 пакетов. Около 3000 пакетов доступны для установки через пакетный менеджер apt

Поиск пакетов ROS лучше начинать на сайте http://wiki.ros.org/

Структура пакета

Пакет ROS содержат множество различных файлов. Для того, чтобы было проще ориентироваться с файлами любого пакета, сообщество разработчиков рекомендует использовать единообразную файловую структуру пакета:

- `bin/` Директория, в которой хранятся скомпилированные программы
- `include/` Директория содержит файлы с заголовками (headers) библиотек
- `launch/` Директория для хранения файлов конфигурации запуска `.launch`
- `msg/` Директория для сообщений (для топиков)
- `src/` Директория для хранения исходников программ (в том числе и скриптов)
- `srv/` Директория для хранения сообщений для использования Сервисами (Services)
- `CMakeLists.txt:` Файл форматы `Cmake` с инструкциями для установки пакете
- `package.xml` Файл "манифест" для описание пакета

С точки зрения "системы пакетов" самый важный файл в структуре – это файл `package.xml` . Именно в нем описано описание пакета, что делает "обычную" директорию с файлами, именно пакетом.

Также в этом файле описаны все необходимые "внешние" зависимости, необходимые для работы. Если пакет установлен "верно", это гарантирует, что нам не нужно самим разбираться с дополнительными "установками".

Минимально в файле должно находиться описание полей

```
<name> - Название пакета
<version> - Версия пакета
<description> - Описание пакета
<maintainer> - Авторы пакета
<license> - Описание лицензии
```

Самый простой файл может выглядеть так

```
<package format="2">
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
  This package provides foo capability.
  </description>
  <maintainer email="ivana@osrf.org">Ivana Bildbotz</maintainer>
  <license>BSD</license>
</package>
```

Для описания зависимостей в файл могут быть добавлены параметры (xml теги)

```
buildtool_depend - Зависимости от системы "сборки"
depend - Зависимости  необходимые для всех случаев использования (сборка,
build_depend - Зависимости необходимые при "сборке" пакета
exec_depend - Зависимости необходимые при запуске пакета
test_depend - Зависимости необходимые при запуске тестов
```

Не слишком сложный файл `package.xml`

```xml
<package format="2">
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>
  <url>http://ros.org/wiki/foo_core</url>
  <author>Ivana Bildbotz</author>
  <buildtool_depend>catkin</buildtool_depend>
  <depend>roscpp</depend>
  <depend>std_msgs</depend>
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
  <exec_depend>rospy</exec_depend>
  <test_depend>python-mock</test_depend>
  <doc_depend>doxygen</doc_depend>
</package>
```

Более подробно можно посмотреть на странице Manifest

Установка пакета ROS из репозитория

Если мы используем дистрибутив линукс семейства Ubuntu, то сообщество разработчиков уже подготовила за нас основные пакеты, и мы можем установить их просто утилитой `apt`. Именно таким способом мы и устанавливали "весь" ROS ранее.

Посмотреть уже установленные пакеты ROS можно командой

```
apt list --installed | grep ros-noetic
```

Поиск пакета ROS

```
apt seach ros-noetic-Имя или просто apt seach ros-noetic-Имя
```

Если вы нашли пакет `pkg_name` на wiki, то скорее всего в системе apt будет назваться `ros-noetic-pkg_name`

Установка нового пакета происходит стандартным способом

```
sudo apt install ros-noetic-pkg_name
```

Установка пакетов и исходного кода

Если пакет ROS доступен только в виде исходного кода, то мы тоже можем его установить (для этого чаще используют формулировку "собрать") на свой компьютер и начать его использовать.

Такая операция сложнее, чем использование установка через `apt`, но позволяет устанавливать любые пакеты, а не только те, которые уже были "собраны" заранее.

Убедитесь, что вы "правильно" установили ROS Установка и запуск ROS, выполнили часть "Настройка рабочего окружения"
http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

Если все выполнено правильно, то у вас в домашней директории пользователя должна быть директория `catkin_ws`

Зайдем в директорию, где должны находиться исходные коды пакетов

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

Для демонстрации мы будем устанавливать наш тестовый пакет https://github.com/voltbro/ros_book_samples

Получим исходный код пакета

```
git clone https://github.com/voltbro/ros_book_samples
```

У нас должны получиться директория `ros_book_samples` с исходниками пакета.

Мы может "собрать" конкретно наш пакет командой

```
catkin_make --pkg=ros_book_samples
```

После этого мы увидим процесс сборки и установки пакета в систему. При сборке демонстрационного пакета будут добавлены новые типы сообщений, которые добавились с пакетом.

```
[  0%] Built target std_msgs_generate_messages_cpp
[  0%] Built target _ros_book_samples_generate_messages_check_deps_Baromet
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target actionlib_msgs_generate_messages_cpp
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target _ros_book_samples_generate_messages_check_deps_AddTwoI
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target _ros_book_samples_generate_messages_check_deps_DoDishe
[  0%] Built target std_msgs_generate_messages_nodejs
[  0%] Built target actionlib_msgs_generate_messages_nodejs
[  0%] Built target std_msgs_generate_messages_lisp
[  0%] Built target actionlib_msgs_generate_messages_lisp
[  0%] Built target std_msgs_generate_messages_eus
[  0%] Built target actionlib_msgs_generate_messages_eus
[  0%] Built target std_msgs_generate_messages_py
[ 20%] Built target ros_book_samples_generate_messages_cpp
[ 20%] Built target actionlib_msgs_generate_messages_py
```

```
[ 37%] Built target ros_book_samples_generate_messages_nodejs
[ 57%] Built target ros_book_samples_generate_messages_eus
[ 77%] Built target ros_book_samples_generate_messages_lisp
[100%] Built target ros_book_samples_generate_messages_py
[100%] Built target ros_book_samples_generate_messages
```

Также мы можем пересобрать сразу все пакеты нашей рабочей директории командой

```
catkin_make
```

После сборки пакета проверим правильность его установки, например, командой

```
roscd ros_book_samples
```

Команда должны выполнится без ошибок, а мы должны перейти в директорию `~/catkin_ws/src/ros_book_samples`

Установка пакета ROS из исходников завершена. Обычно для установки более сложный примеров разработчики формируют полные инструкции о том, как установить их пакет.

Утилита catkin

`catkin` это модуль, который служит для "сборки" пакетов. "Сборкой" называют ряд определенный действий по установке пакета в систему. Это могут быть операции копирования файлов, компиляция программ (например для языка C), генерации сообщений и тд.

Для описания необходимый операция по установке, используется стандарт CMake (Cross Platform Make), а сама последовательность операций описана в файле `CMakeLists.txt` .

Стандарт CMake был изменен в ROS для создания системы сборки, более специфичной для нужд ROS, так появился `catkin`, который стал стандартом для ROS, начиная с версии Groovy

---

Навигация по файлам пакетов ROS

В ROS есть несколько консольных утилит, которые могу упростить навигацию по файлам разных пакетов.

Наиболее часто используемые утилиты `rospack`, `rospcd`, `rosls`

Например если мы хотим найти директорию с установленным пакетом `turtlesim`

```
rospack find turtlesim
```

То мы получим директорию установки пакета `/opt/ros/noetic/share/turtlesim`

Получить список файлов пакета `rosls turtlesim`

```
rosls turtlesim
===
cmake  images  msg  package.xml  srv
```

Перейти в директорию пакета

```
roscd turtlesim
pwd
```

Мы увидим, что находимся в директории пакета `/opt/ros/noetic/share/turtlesim`

Создание пакета

Создание простого пакета

На данный момент мы познакомились с файловой структурой пакетов и научились их устанавливать.

В этой главе мы научимся самостоятельно создавать собственные ROS пакеты.

В принципе, создать пакет можно и "руками" правильно написав файлы `CMakeLists.txt` и `package.xml`. Но проще это все делать при помощи уже созданной утилиты `catkin_create_pkg`

Все операции с пакетами мы будет производить в нашей рабочей директории `catkin_ws`

Перейдем в директорию, где расположены исходники пакетов

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

И теперь создадим наш пакет, где `my_first_package` - это название нашего первого пакета

```
catkin_create_pkg my_first_package
===
Created file my_first_package/package.xml
Created file my_first_package/CMakeLists.txt
Successfully created files in /home/cola/catkin_ws/src/my_first_package. P
```

Мы увидим, что созданы все минимально необходимые файлы.

```
ls my_first_package
```

```
===
CMakeLists.txt  package.xml
```

Далее мы можем его установить в нашу систему, выполнив `catkin_make`

```
cd ../
catkin_make --pkg=my_first_package
```

Проверим, что ROS правильно видит наш новый пакет

```
rospack find my_first_package
===
/home/cola/catkin_ws/src/my_first_package
```

Мы видим, что ROS правильно смог найти новый пакет, значит он установлен в системе верно.

Работа с зависимостями

Обычно при создании пакета мы уже знаем, какие пакеты ROS нам понадобятся. Для того, чтобы нам в ручном режиме не редактировать файлы описания настроек пакета `package.xml` и `CMakeLists.txt`, нам проще сразу создать пакет, указав все зависимости.

Синтаксис создания пакета с указанием зависимостей

```
catkin_create_pkg [package_name] [depend1] [depend2] [depend3]
```

Создадим пакет `my_package` указав зависимости `std_msgs` , `rospy` , `actionlib_msgs` , `message_generation` , `message_runtime`

```
cd ~/catkin_ws/src
catkin_create_pkg my_package std_msgs rospy actionlib_msgs message_generat
```

При создании пакета мы указали зависимости, которые мы будем использовать в работе в следующих главах, поэтому сейчас не так важно, что они все делают.

Мы можем увидеть, что в новом пакете в файле `package.xml` появились новые строчки описания зависимостей

```
<build_depend>actionlib_msgs</build_depend>
<build_depend>message_generation</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>actionlib_msgs</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>actionlib_msgs</exec_depend>
<exec_depend>message_runtime</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Также в файле `CMakeLists.txt` появились новые команды сборки

```
find_package(catkin REQUIRED COMPONENTS
   actionlib_msgs
   message_generation
   message_runtime
   rospy
   std_msgs
 )
```

Соберем наш новый пакет

```
cd ~/catkin_ws
catkin_make --pkg=my_package
```

Проверим список зависимостей пакета

```
rospack depends my_package
===
catkin
genmsg
gencpp
geneus
gennodejs
genlisp
genpy
message_generation
cpp_common
rostime
roscpp_traits
roscpp_serialization
message_runtime
std_msgs
actionlib_msgs
rosbuild
rosconsole
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
ros_environment
rospack
roslib
rospy
```

Мы видим явно больше зависимостей, чем мы указали при создании пакета. Но на самом деле мы увидели **полный** список всех необходимых пакетов. Включая зависимости к тем пакетам, которые мы выбрали при создании.

Создание собственных типов сообщений

Сообщения - это одна из базовых сущностей ROS. Ранее мы всегда использовали уже созданные типы сообщений. В этой главе мы научимся создавать собственные сообщения.

Работать с файлами необходимо в директории пакета `my_package`, который мы создали в прошлой главе.

Создание сообщений для Топиков (Topic)

Файлы с описанием сообщений необходимо сохранить в директории `./msg` нашего пакета. При этом имя файла определяет название типа созданного сообщения.

Структура файла сообщений для Топиков очень проста, мы в строчку перечисляем переменные и тип этих переменных.

Например, мы хотим использовать данные с датчика давления и температуры `BMP180`. Мы предполагаем, что в одном сообщении нам необходимо передавать два значения: давление и температура (стандартные параметры для барометрических датчиков).

Создадим файл `./msg/Barometer.msg`

```
uint16  pressure
float32 temperature
```

Мы описали, что тип сообщения `Barometer` состоит из двух переменных, значение давление (pressure) типа `uint16` и температуры (temperature) типа `float32`

Ранее мы уже создали пакет с необходимыми зависимостями, поэтом сейчас нам достаточно в процесс сборки, добавить задание на генерацию нашего нового

сообщения.

Файл `CMakeLists.txt` должен содержать директивы генерации.

```
add_message_files(FILES Barometer.msg)
```

И на случай, если у нас не было "стандартных" сообщений, их нужно тоже создать

```
generate_messages(DEPENDENCIES std_msgs)
```

Далее запустим сборку пакета

```
cd ~/catkin_ws
catkin_make --pkg=my_package
```

Мы увидим сообщения о генерации нашего сообщения для разных языков

```
....
[ 20%] Generating Lisp code from my_package/Barometer.msg
....
[ 60%] Generating C++ code from my_package/Barometer.msg
...
[ 80%] Generating Python from MSG my_package/Barometer
```

Проверим, что сообщение видно в системе (получим список всех сообщений и сделаем фильтр с названием нашего пакета)

```
rosmsg list | grep my_package
===

```

```
my_package/Barometer
```

Выведем информацию о сообщении

```
rosmsg show my_package/Barometer
===

uint16 pressure
float32 temperature
```

Мы можем подключить созданный тип сообщения в наш python скрипт

```
from my_package.msg import Barometer
```

Создание сообщений для Сервисов (Services)

Для создания нового сообщения сервисов нам необходимо определить специальное парное сообщение, состоящие из двух частей. Первая часть - это Запрос (Service Request), вторая часть - это Ответ (Service Response).

В файле описания сервиса первая часть (до разделителя ---) - это описание сообщения Запроса, далее описание сообщения Ответа.

Например, файл для сервиса сложения `srv/AddTwoInts.srv`

```
uint32 a
uint32 b
---
uint32 sum
```

Файлы с описанием сервисов хранятся в директории `srv` и имеют расширение `` `.srv `` Подробное описание файла доступно на странице wiki http://wiki.ros.org/rosbuild/srv

При этом, имя файла `AddTwoInts.srv` соответствует имени типа `AddTwoInts` .

Для подключения генерации новый сообщений сервисов, нам необходимо в файле `CMakeLists.txt` добавить файл .srv для обработки

```
add_service_files(FILES
    AddTwoInts.srv
)
```

Запустим заново сборку пакета

```
cd ~/catkin_ws
catkin_make --pkg=my_package
```

Мы увидим сообщения о генерации нашего сообщения для разных языков

```
....
[  0%] Generating Python code from SRV my_package/AddTwoInts
```

Проверим наличии нового типа сообщений

```
rossrv list | grep my_
===
my_package/AddTwoInts
```

Сообщение найдено.

Для работы в python мы можем импортировать сообщение следующим образом

```
from my_package.srv import AddTwoInts, AddTwoIntsResponse, AddTwoIntsRequest
```

Создание сообщений для Экшенов (Actions)

Описание формата сообщений

Файлы описания действия (Action) находятся в директории `./action` пакета имеют расширение `.action` и выглядят приблизительно так:

Пример файла с сообщением `action/DoDishes.action`

```
# Определение цели (goal)
uint32 dishes  # Сколько мыть тарелок
---
# Определение результата (result)
uint32 total_dishes_cleaned # Сколько всего было вымыто
---
# Определение обратной связи (feedback)
uint32 dishes_cleaned # Сколько вымыто посуды сейчас
```

На основе этого файла `.action` создаются 6 вспомогательных сообщений, чтобы клиент и сервер могли общаться. Создание всех этих сообщений происходит автоматически во время процесса сборки пакета.

Для файла `DoDishes.action` будут созданы файлы

```
DoDishesAction.msg
DoDishesActionGoal.msg
DoDishesActionResult.msg
DoDishesActionFeedback.msg
```

```
DoDishesGoal.msg
DoDishesResult.msg
DoDishesFeadback.msg
```

Дополнительная информация о работе с Действиями (Action) доступна на станице ROS wiki

Генерация сообщений

Как мы уже делали ранее, в файле `CMakeLists.txt` необходимо внести новые директивы для обработки `.action` файлов `add_action_files`

```
add_action_files(
  FILES
  DoDishes.action
)
```

Также нам необходимо убедиться, что есть все базовые типы для сообщений, поэтому в блоке `generate_messages` должны быть добавлены сообщения `actionlib_msgs`

```
generate_messages(
   DEPENDENCIES
   actionlib_msgs
   std_msgs
)
```

Запустим заново сборку пакета

```
cd ~/catkin_ws
catkin_make --pkg=my_package
```

При сборке пакета мы видим, что генерируются сообщения типа `DoDishes`

```
...
[  0%] Generating C++ code from my_package/DoDishesAction.msg
[  0%] Generating Python from MSG my_package/DoDishesAction
[  3%] Generating Python from MSG my_package/DoDishesResult
[  7%] Generating C++ code from my_package/DoDishesResult.msg
...
```

Проверим, что установка выполнена и сообщения доступны для использования

```
rosmsg list | grep DoDi
===
my_package/DoDishesAction
my_package/DoDishesActionFeedback
my_package/DoDishesActionGoal
my_package/DoDishesActionResult
my_package/DoDishesFeedback
my_package/DoDishesGoal
my_package/DoDishesResult
```

Все сообщения доступны для использования.

Для использования в python мы можем подключить наши сообщения следующим образом

```
from my_package.msg import DoDishesAction, DoDishesGoal
```

roslaunch, rosrun управление запуском

rosrun

`rosrun` - это команда, которая может запустить только одну ноду в выбранном пакете. Ранее мы уже запускали ноды рос, например

```
rosrun turtlesim turtlesim_node
```

Это команды найдет пакет `turtlesim` и запустит в нем программу `turtlesim_node`

Подготовка python скриптов для запуска

В параметрах программы `rosrun` необходимо указать "программу", а не имя python файла. Поэтому мы должны превратить python скрипты в \"программу\". Мы рассматривали такую задачу в части введения в Python.

Если кратко, то мы должны убедиться, что python скрипт начинается со строчки

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

И для скрипта добавлена возможность запуска

```
chmod +x pub.py
```

Если мы сохраним программу издателя `pub.py` в директорию `/src` нашего пакета и выполним "подготовку" к запуску, то мы сможем запустить процесс публикации данных командой

```
rosrun my_package pub.py
```

Важное замечание

Важно учесть, что в нашем рабочем окружении (папка catkin_ws) место расположения исходников проекта и место расположения пакета после выполнения сборки одно и тоже.

По этой причине конкретно в нашем случае нам не нужно добавлять дополнительных команд в сборку для копирования файлов пакета. Но в реальных проектах очень часто эти директории разные. Поэтому для всех python файлов необходимо в файл `CMakeLists.txt` добавить задание копирования из папки с исходниками в папку для запуска.

```
install(PROGRAMS
  src/pub.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

А ресурсные файли и тому подобное необходимо устанавливать (переносить в рабочую папку), например, вот так

```
install(DIRECTORY
  launch
  DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
)
```

Где `launch` - это директория с `.launch` файлами

---

roslunch , управление запуском множества нод

Раньше мы использовали для запуска утилиту `rosrun`, которая запускает конкретный исполняемый файл.

В реальных системах одновременно должны работать множество программ. Для их запуска и конфигурации служит утилита `roslaunch`

Используя `roslaunch`, возможно дополнительно настраивать исполняемые файлы в момент их запуска (передавать параметры, изменять имена и тп)

`roslaunch` использует файлы с расширением `.launch`, которые представляет собой обычный XML файл.

Давайте создадим файл `./launch/demo.launch`

```
<launch>
  <node pkg="my_package" type="pub.py" name="topic_publisher1"/>
  <node pkg="my_package" type="sub.py" name="topic_subscriber1"/>
  <node pkg="my_package" type="pub.py" name="topic_publisher2"/>
  <node pkg="my_package" type="sub.py" name="topic_subscriber2"/>
</launch>
```

Файлы `pub.py` и `sub.py` Можно взять из примеров работы с топиками, и их нужно сохранить в директорию `src`

Теги, необходимые для запуска узла с помощью команды `roslaunch`, описаны в теге `launch`. Тег `node` описывает ноды, который которые необходимо запускать с помощью `roslaunch`. Параметры включают «pkg», «type» и «name».

Параметр	Описание
pkg	Имя пакета
type	Название ноды, которая будет выполняться

name	Имя (исполняемое имя), используемое при выполнении ноды, соответствующего выше параметру type. Обычно это имя совпадает, но при его запуске можно использовать другое имя.

После сохранения файла мы можем его запустить, выполнив

```
$ roslaunch my_package demo.launch --screen

NODES
  /
    topic_publisher1 (my_package/topic_publisher.py)
    topic_publisher2 (my_package/topic_publisher.py)
    topic_subscriber1 (my_package/topic_subscriber)
    topic_subscriber2 (my_package/topic_subscriber)

ROS_MASTER_URI=http://localhost:11311

process[topic_publisher1-1]: started with pid [24963]
process[topic_subscriber1-2]: started with pid [24964]
process[topic_publisher2-3]: started with pid [24965]
process[topic_subscriber2-4]: started with pid [24978]
```

Если добавить параметр `--screen`, информация о работе запущенных программ будет отображена на экране текущего терминала.

Или для тестов мы можем указывать просто путь к .launch файлу

```
$ roslaunch /home/user/catkin_ws/src/my_package/launch/demo.launch --screen
```

Как мы видим, запущенно 4 процесса с разными `pid.` Также мы можем увидеть список запущенных нод

```
$ rosnode list

/rosout
/topic_publisher1
```

```
/topic_publisher2
/topic_subscriber1
/topic_subscriber2
```

Объединение процессов в группы

Если мы посмотрим на список топиков

```
$ rostopic list
===
/welcome_topic
```

То поймем, что оба Издателя (Publisher) публикуют данные в один топик `/welcome_topic` (других топиков не создана) и каждый из подписчиков получает сообщения сразу от двух Издателей. Скорее всего, такой режим работы нам не интересен.

Если мы хотим, чтобы один конкретный Издатель и Подписчик были изолированны от аналогичных процессов, мы можем объединить их в группы.

Создадим новый файл запуска `./launch/demo2.launch`

```
<launch>
  <group ns="ns1">
    <node pkg="my_package" type="pub.py" name="topic_publisher"/>
    <node pkg="my_package" type="sub.py" name="topic_subscriber"/>
  </group>
  <group ns="ns2">
    <node pkg="my_package" type="pub.py" name="topic_publisher"/>
    <node pkg="my_package" type="sub.py" name="topic_subscriber"/>
  </group>
</launch>
```

Запустим новый файл

```
$ roslaunch my_package demo2.launch
```

```
$ rostopic list

/ns1/welcome_topic
/ns2/welcome_topic
```

Мы видим, что создано два отдельных топика, с которым работает один Издатель и один Подписчик.

```
$ rostopic info /ns1/welcome_topic


Type: std_msgs/String

Publishers:
 * /ns1/topic_publisher (http://cola:44225/)


Subscribers:
 * /ns1/topic_subscriber (http://cola:44043/)



$ rostopic info /ns2/welcome_topic

Type: std_msgs/String


Publishers:
 * /ns2/topic_publisher (http://cola:45259/)


Subscribers:
 * /ns2/topic_subscriber (http://cola:35017/)
```

Установка переменных окружения при запуске

Ранее мы уже разобрали, как работает Сервер Параметров. Но его использование все еще было не таким удобным, если бы мы решили изменить какой либо параметр по умолчанию, то нам бы пришлось это делать руками через утилиту `rosparams` при каждом запуске.

Чтобы решить данную проблему, мы можем использовать возможность передавать параметры в запускаемые программы через `.launch` файлы

Рассмотрим пример (для файла `params.py` который мы создали когда работали с параметрами)

```
<launch>
    <node pkg="my_package" type="params.py" name="params_demo" output="scr
        <param name="my_param" value="launch_file_param"/>
    </node>
</launch>
```

Если файл запустить, то мы увидим, что значение по умолчанию изменено на то, которое мы задали в `.launch` файле

```
process[params_demo-1]: started with pid [6825]
launch_file_param
```

В ветке `node` добавились элементы `param` с настройками. Открывая такой файл, сразу видно, какие параметры возможно конфигурировать, а также их значения по умолчанию.

Если запустить этот файл, то мы увидим что программа получила параметр из нашего `.launch` файла.

Вторым параметром в функции `get_param` указывается значение по умолчанию, если параметр не определен в `.launch` файле.

Параметры возможно передавать при запуске через `rosrun`

```
rosrun my_package params.py _my_param:=run_file_param
```

Подключение других .launch файлов <include>

В реальных проектах запускаются десятки нод. Конфигурировать каждую из них в одном файле не всегда удобно. К тому же обычно сторонние пакеты уже содержат подходящие `.launch` файлы. Поэтому существуют механизм `include`, который позволяет подключать другие файлы запуска.

Приведем пример

```
<launch>
 <include file="$(find my_package)/launch/test_params.launch">
     <arg name="device" value="/dev/ttyS1"/>
  </include>
  <include file="$(find navibro)/camera/camerav1_640x480.launch"/>
  <include file="$(find navibro)/launch/aruco_detect.launch"/>
  <include file="$(find navibro)/launch/fiducial_slam.launch"/>
</launch>
```

В этом примере мы подключаем файл `test_params.launch`, который находиться в нашем пакете, и настраиваем его на работу через устройство `/dev/ttyS1.`. А также подключаем три других `.launch` файла из другого пакета.

Использование условий if и unless

При написании сложный .launch файлов, очень помогают атрибуты `if` и `unless`, которые позволяют формировать простые алгоритмы ветвления при работе с `roslaunch`

Приведем несколько примеров

```
<launch>
   <arg name="have_serial" value="true"/>

   <group if="$(arg have_serial)">
     <!-- Блок выполнится только если have_serial установлено в true -->
     <node pkg="my_package" type="test_params.py" name="my_package" output
  </group>
  <!-- Также if можно использовать для одного тега-->
  <include if="$(arg have_serial)" file="$(find my_package)/launch/test_pa
    <arg name="device" value="/dev/ttyS1"/>
  </include>
</launch>
```

Атрибут `unless` работает противоположно атрибуту `if` . Если значение `0` то блок выполняется.

Значение атрибутов для `if` и `unless` должно быть булевым типом данных(`0,1,true,false`)

Практическое задание

Для выполнения задания вам необходимо

- Создать новый пакет `home_work`
  - Изменить описание пакета, указав авторство и описание
- Переместить ранее разработанные скрипты (pub.py, sub.py, service_client.py, service_server.py, action_client.py, action_server.py)
  - Добиться того, что скрипты можно было запускать через `rosrun home_work script_name.py`
- Создать для пакета собственные типы сообщений
  - Сообщение для топиков `Welcome.msg` (по аналогии с сообщением `std_msgs/String`)
  - Сообщение для сервисов `AddTwoInts.srv` (по аналогии `rospy_tutorials/AddTwoInts`)
  - Экшен `FibonacciAction.action` на основе `actionlib_tutorials/FibonacciAction`
  - Модифицировать все исходники для работы с собственными типами сообщениями
- Создать `.launch` файл для запуска одновременно Издателя и Подписчика
  - Через конфигурацию в `.launch` файле изменить текст сообщения и название топика
- Создать `.launch` файл для запуска сервера `AddTwoInts`
- Создать `.launch` файл для запуска Action сервера `FibonacciAction`
- Создать общий `.launch` файл для запуска трех `.launch` файлов созданных ранее