5 Разработка в ROS

Python для ROS

Ранее мы изучили базовые методы разработки на `python` и познакомились с основными сущностями ROS, смогли практически их использовать при помощи консольных утилит.

В этой части мы разберемся, как самостоятельно из своих программ работать с данными, используя паттерны Издателя и Подписчика. Напишем первые программы ROS.

Библиотека rospy

`rospy` - это клиентская библиотека Python для ROS. Библиотека позволяет разработчикам быстро взаимодействовать с Топиками и Сервисами.

Архитектура rospy отдает предпочтение скорости реализации (то есть времени, затраченному разработчиком) по сравнению с производительностью во время выполнения. Библиотека отлично продходит для быстрого прототипирования и тестирования в ROS. Многие инструменты ROS написаны на `rospy`, например, уже известные нам утилиты `rostopic` и `rosservice`.

Библиотека `ropsy` устанавливается при установке ROS, поэтому отдельно устанавливать нам ничего не нужно.

Полезное

Основная страница библиотеки http://wiki.ros.org/rospy

Программа Издатель

Начнем с классического примера "Hello robot". Это программа, которая передает в топик сообщение с приветствием. Или если перефразировать более точно, то публикует сообщение "Hello robot" в топик `/welcome_topic`

Полный код программы (`pub.py`) дан ниже, скопируйте его себе, и дальше мы разберем, что происходит в каждой строчке программы.

```
import rospy
from std_msgs.msg import String

rospy.init_node("welcome_node")

pub = rospy.Publisher("/welcome_topic", String, queue_size=10)

s = String()
s.data = "Hello robot"

while not rospy.is_shutdown():
    pub.publish(s)
    rospy.sleep(1)
```

Самое начало программы - это блок подключения внешних файлов. Мы подключаем библиотеку `rospy` и класс для работы со стандартным сообщением типа `String` . Нам для передачи данных необходим обьект ROS "строка", если мы хотим передать "Hello robot"

```
import rospy
from std_msgs.msg import String
```

Инициализируем ноду рос. В этот момент происходит "магия" подключения ноды к мастер-ноде, поэтому стоит проверить, что вы уже запустили `roscore`

```
rospy.init_node("welcome_node")
```

Инициализация Издателя. В библиотеке rospy есть созданный класс, который реализует возможность работы с топикапи для публикации данных. Это `rospy.Publisher.` Для его инициализации необходимо указать обязательные атрибуты:

- Имя топика, в который мы должны пубиковать данные
- Тип сообщения, который должен быть в топике
- И не очень явный, но обязательный параметрт queue_size (это размер очереди на отправку сообщений)

В программном виде инициализация объекта Издатель будет выглядеть вот так:

```
pub = rospy.Publisher("/welcome_topic", String, queue_size=10)
```

Далее мы инициализируем само сообщение. Напомним, что "строчка" - это некоторый объект с точки зрения ROS. Поэтому мы сначала инициализируем пустой объект, а потом заполняем его атрибуты.

```
s = String()
s.data = "Hello robot"
```

На самом деле тут можно было сократить код до одной строчки, но важно понять, что такое объект `String`

```
s = String("Hello robot")
```

Сейчас у нас есть объект `pub`, который умеет отправлять сообщения и объект `s` собственно с сообщением. Запустим "вечный" цикл, который будет публиковать данные

```
while not rospy.is_shutdown():
    pub.publish(s)
    rospy.sleep(1)
```

`rospy.is_shutdown()` - это метод который проверяет, что есть подключение к мастер ноде. Нам его удобно использовать для того, чтобы программа или сама завершалась при отключении мастер ноды, и работал выход из программы по нажатию комбинации прерывания работы программы `Ctrl-C`.

Но в целом мы могли написать просто `while True`

`pub.publish(s)` - вызов публикации данных. Объект `pub` отправляет объект `s`, который содержит нашу строчку "Hello Robot"

`rospy.sleep(1)` - это функция "паузы" отправки данных в топик, отправлять данные с максимальной скоростью нам абсолютно незачем. Мы отправляем данные раз в секунду.

Запуск программы

```
python3 pub.py
```

В нашей программе нет ни одного сообщения для отладки, поэтому при запуске программы мы увидим пустой экран и нам будет казаться, что ничего не происходит. Но это не так, программа запустилась без ошибок и работает!

Проверим, что на самом деле все работает.

Найдем нашу ноду

```
rosnode list
➜
/rosout
/welcome_node
```

Нода `/welcome_node` появилась

Проверим, что у нас появился "наш" топик `welcome_topic`

```
rostopic list

/rosout
/rosout_agg
/welcome_topic
```

И финальное - посмотри данные в топике

```
rostopic echo /welcome_topic

data: "Hello robot"
---

data: "Hello robot"

---
data: "Hello robot"
```

Мы видим данные из нашей первой программы.

Наша первая программа ROS работает!

Программа Подписчик

Уроком ранее мы написали первую программу, которая публикует данные. Сейчас мы рассмотрим программу на `python`, которая будет принимать эти данные и выводить их на экран.

Полный код программы (sub.py) дан ниже, скопируйте его себе и сохраните в файл. Далее мы рассмотрим его более подробно.

```
import rospy
from std_msgs.msg import String

rospy.init_node("welcome_sub_node")

def callback(msg):
    print(msg)

rospy.Subscriber("welcome_topic", String, callback)
rospy.spin()
```

Как обычно программа начинается с подключения других модулей. Мы подключили основную библиотеку `rospy` и тип сообщений `String` из прошлой программы; мы помним, что используем именно этот тип сообщения.

```
import rospy
from std_msgs.msg import String
```

Инициализируем ноду рос, укажем новое имя.

```
rospy.init_node("welcome_sub_node")
```

Создадим функцию обработки полученных данных.

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

Тут стоит немного поговорить про эту функцию. В программировании такая функция называется `CallBack` (функция обратного вызова). Это некоторый подход к разработке, который позволяет запускать обработку данных только в момент, когда данные есть, в остальное время программа "занимается" другими своими делами. Поэтому мы говорим, что это callback функция, которая запускается по вызову.

Сама по себе функция ничего особого не делает, только печатает переданные ей данные. Также мы можем ее назвать как нам удобно, это не так важно. Важно, что мы понимаем, что это та функция, которая должна запуститься, когда получены данные.

Далее мы создаем наш основной объект "подписчик". Который, собственно, и делает то что нужно.

```
rospy.Subscriber("/welcome_topic", String, callback)
```

При инициализации объекта мы передаем параметры

- Имя топика для подписки
- Тип сообщения
- Имя функции CallBack (в нашей программе это `callback`)

По сути, мы написали, что если в топике `/welcome_topic` появится сообщение типа `String`, то надо запустить функцию `callback` При этом как параметр функции будет передано само сообщение для дальнейшей ее обработки.

По логике нашй программы, она должна работать "вечно", а не сразу выходить. Для этого мы в конец программы добавим функцию, которая ничего не делает, но и не дает программе закрыться. Это `rospy.spin()`

```
rospy.spin()
```

Если мы запустим одновременно и Издателя (pub.py), и нашего подписчика (sub.py), то в выводе Подписчика мы увидим

```
python sub.py
data: "Hello robot"

data: "Hello robot"
```

Ради интереса мы можем поменять программу pub.py, изменив текст сообщения, и мы увидим, что текст вывода в sub.py поменяется.

Пример работы Издателя и Подписчика

На уроках ранее мы рассмотрели примеры простых программ, которые могли или отправлять данные, или только получать данные.

Но в реальных условиях таких программ очень мало. Обычно программы одновременно и принимают множество различных данных, анализируют их и отправляют их далее.

В текущий момент у вас нет доступа к реальному роботу, поэтому задачи будем рассматривать исключительно теоретические.

Давайте создадим программу, которая будет принимать данные из одного топика и передавать в другой. Например, мы будем получать в топик `/name` данные типа `String` с именем человека и возвращать приветствие в виде `Hello, Имя` в топик `/greeting`

Также из нового для нас – мы будем использовать подходы ООП для решения задачи. Мы напишем класс `GreetingWorker`, который будет содержать в себе всю логику работы.

Рассмотрим реализацию такой программы `pubsub.py`

```
import rospy
from std_msgs.msg import String

class GreetingWorker(object):

    def __init__(self):

        rospy.loginfo("Start Greeting Node")
        self.pub = rospy.Publisher('/greeting', String, queue_size=10)
        rospy.Subscriber("/name", String, self.sayHello)

    def sayHello(self, income_msg):
        self.pub.publish('Hello, {}'.format(income_msg.data))


rospy.init_node('greeting_node')
greeter = GreetingWorker()
```

```
18
19  rospy.spin()
```

Рассмотрим класс `GreetingWorker` . В конструкторе класса мы для отладки выводим информационное сообщение:

```
rospy.loginfo("Start Greeting Node")
```

Инициализируем издатель на топик `/greeeting` и присваиваем его атрибуту класса `self.pub`

```
self.pub = rospy.Publisher('/greeting', String, queue_size=10)
```

Инициализируем подписчик на топик `/name` , указав функцией обратного вызова метод текущего класса `sayHello`

```
rospy.Subscriber("/name", String, self.sayHello)
```

Функция CallBack публикует форматированные данные

```
1  def sayHello(self, income_msg):
2      self.pub.publish('Hello, {}'.format(income_msg.data))
```

Класс мы завершили, нам осталось только запустить его.

```
1  rospy.init_node('greeting_node')
2  greeter = GreetingWorker()
3
```

```
4  rospy.spin()
```

Мы инициализируем ноду рос, создаем объект `GreetingWorker` и запускаем rospy.spin() для того, чтобы программа не завершала свою работу.

Запустим и проверим

```
python pubsub.py
```

Откроем вывод топика `/greeting`

```
rostopic echo /greeting
```

Отправим тестовые данные в топик `/name` (параметр -r10 означает, что данные публикуются с частотой 10 герц)

```
rostopic pub /name std_msgs/String "Robot" -r10
```

В консоли вывода топика `/greeting` мы увидим, что данные начали выводиться на экран.

```
1  Hello, Robot
2
3  Hello, Robot
4
5  Hello, Robot
```

Пример работы Издателя и Подписчика, часть2

На предыдущем уроке мы создали пример, в котором **Подписчик** и **Издатель** работают совместно. А именно, Подписчик вызывает Издателя. На этом уроке мы попрактикуемся создавать примеры, в которых Подписчик и Издатель будут действовать независимо.

Давайте разберемся, в чем принципиальная разница зависимой и независимой работы. В примере предыдущего урока Подписчик вызвал метод `greeter.sayHello` и передавал ей данные, которые он получал из топика `/name.` Функция `greeter.sayHello` в свою очередь форматировала полученные от Подписчика данные и передавала их в Издатель для публикации в топик `/greetings` . Как вы видите, вся логика программы увязана с получением данных в топике `/name` . Т.е. если данные в этот топик не поступают, то и функция `greeter.sayHello` не вызывается и Издатель не публикует ничего в топик `/greetings` .

Теперь давайте создадим программу, которая будет принимать данные из одного топика и сохранять эти данные в некую переменную, чтобы другие методы нашей программы могли этими данными пользоваться когда они им понадобятся.

Слегка модифицируем пример из прошлого урока. Мы также будем получать в топик `/name` данные типа `String` с именем человека и возвращать приветствие в виде `Hello, Имя` в топик `/greeting` . Однако сейчас сохранять данные "Имя" мы будем в свойства класса, а публиковать приветствие будем, используя это "Имя", но в другом цикле.

Файл `pubsub2.py`

```
1  import rospy
2
3  from std_msgs.msg import String
4
5  class GreetingWorker(object):
6
7      def __init__(self):
```

```
        rospy.loginfo('Start Greeting Node')
        self.pub = rospy.Publisher('/greeting', String, queue_size=10)
        self.sub = rospy.Subscriber('/name', String, self.cbName)
        self.name = ""

    def cbName(self, income_msg):
        self.name = income_msg.data

    def pubGreet(self):
        self.pub.publish('Hello, {}'.format(self.name))

    def run(self):
        while not rospy.is_shutdown():
            self.pubGreet()
            rospy.sleep(1)


rospy.init_node('greeting_indep_node')
greeter = GreetingWorker()
greeter.run()
```

В этом примере мы создали класс `GreetingWorker`, который выполняет обработку получения данных в топик `/name` с последующей публикацией в топик `/greeting`.

Обратите внимание, что мы изменили callback функцию этого Подписчика. `cbName` получает данные из Подписчика и только сохраняет их в переменную `name` экземпляра класса. В прошлом примере мы сразу отправляли данные в топик.

```
    def cbName(self, income_msg):
        self.name = income_msg.data
```

Также у нас появился новый метод `run()`

```
def run(self):
    while not rospy.is_shutdown():
        self.pubGreet()
        rospy.sleep(1)
```

В этом методе мы запускаем "бесконечный" цикл на публикацию данных с интервалом в одну секунду. При этом "получение" данных не останавливается, и метод `cbName` всегда вызывается как только получены данные в топике.

Запустим скрипт.

```
python pubsub2.py
```

Откроем вывод топика `/greeting`

```
rostopic echo /greeting
```

Отправим тестовые данные в топик `/name`

```
rostopic pub /name std_msgs/String "Robot" -r10
```

В консоли вывода топика `/greeting` мы увидим, что данные начали публиковаться

```
Hello, Robot

Hello, Robot

Hello, Robot
```

Обратите внимание, что мы публикуем данные из консоли в топик `/name` с частотой 10 герц, а выводятся данные с частотой один герц. Это наглядно видно, если посмотреть, с какой частотой идут публикации в топик `/name` и `/greetings` открыв `rostopic echo name` и `rostopic echo greetings` в соседних терминалах.

Также у нас получилось, что если остановим публикацию имени, то публикация приветствия продолжит работать, так как это независимые части программы.

Практическое задание

1. Необходимо написать программу Издатель, которая в топик `/numbers` отправляет случайные числа от 1 до 99 с частотой 5 грц. (`num_pub.py`)
2. Написать программу (`base_test.py`), которая:
    1. Подпишется на топик `/numbers`
    2. Для 5 последовательно полученных цифр рассчитает среднее арифметическое.
        1. После расчета среднего арифметического для 5 цифр программа должна начать рассчитывать последовательность заново.
    3. Опубликует результат вычисления в топик `/result`