```
#
#
#
#
#
#
#
#
#
#
#
#
#                                ````                                `::-
#    MMMMMMMMo mMMMMNNds-     -yNMMMMNd-      sMMs                  -MMh                                        `ddh
#    MMM/----. mMMo--+mMMy  oMMN+.`./h:       sMMy      -+osso/.  -MMh-+so:     :osso/`  .++:`/+`-+osso/. `+MMN++/  -+sso/.  `++/`:+/++-   :++.
#    MMMyssss` mMM/    `mMM/.MMM-             sMMy      oysosmMM/ -MMNhydMMy -mMNsodMN/ /MMNMNN.oysosmMM/.yMMMyyo`dMMyohMMo .MMNNNN+NMN. -MMd
#    MMMhyyyy` mMM/     dMMo-MMM`            sMMy      .+syhmMMs -MMh   NMM.hMM/  `MMM`/MMh    .+syhmMMs `MMN   sMMo   mMM-.MMm`   :MMh dMN.
#    MMM.      mMM/  `oMMm` mMMy`    /.      sMMy     :MMd:-oMMs -MMh  `NMM`hMM/  `MMN`/MMy   :MMd:-oMMs `MMN   oMMs   mMM-.MMm     oMMdMM:
#    MMM.      mMMmdmMMNs`  `yMMNdhdNM:      sMMNmmmmd-MMNssmMMs -MMNyyNMN/ .mMNysmMN/ /MMy   -MMNssmMMs  mMMhss`hMMysdMMo .MMm      hMMMs
#    ///`      //////:.       `:+oo+:.       -//////// .+o+:.//- `//::++/`    -+oo+:` .//-    .+o+:.//-   :+o+:  ./ooo/`  `///      +MMd
#                                                                                                                                  .NMN.
#                                                                                                                                  `..`
#
#                                      T H E   M O T I O N   T E C H N O L O G Y   I N N O V A T I O N S
#
```

```
#                `++++++/-`    `+++++++-   -+ooo+:  .++.     ./oooo+-   +++`     :+/   +++++++-  -++++++++++++//:.
#                .MMmyyhmMMh-  `MMmyyyy: -mMmsosyy  /MM:   +mMmyssydh  MMMN:    hMm   NMmyyyy/  oMMMMMMMMMMMMMMMMMMmy+.
#                .MMo    -mMN. `MMo     sMM/       /MM:  hMN/          MMhNMo   hMm   NMy       oMMMMMMMMMMMMMMMMMMMMMMh:
#                .MMo     :MMo `MMmyyyy` `yMMmy+.   /MM: :MMo  `+++++. MMs.dMh` hMm   NMmyyyy.  oMMMMM+      `.:odMMMMMMMy
#                .MMo     /MM+ `MMh++++`   `:sdMMy  /MM: :MMo  `yydMM: MMs  sMm-hMm   NMd++++`  oMMMMM+          .yMMMMMMy
#                .MMo   `+NMh  `MMo      .     sMM: /MM:  dMN/    /MM: MMs   /NMmMm   NMy       oMMMMM+            +MMMMMM:
#                .MMNddmMMh/   `MMNddddds sNdyydMMy  /MM:  `omMNdhhNMM- MMs    .dMMm   NMNddddy  oMMMMM+             hMMMMMy
#                 `-----_       `-------_ `-////-`   `--`      .///.-`  --       --_   `-------_  oMMMMM+              +MMMMMm
```

```
#                                                                                                    oMMMMM+             +MMMMMd
#                                                                                .-`                 oMMMMM+             hMMMMMs
#                                                                                dM:                 oMMMMM+            /MMMMMM. `mmm.    /hmmm
#                                                                                dM:-//- `//`  `/:   oMMMMM+          `oMMMMMM/  hMoMd   -MM.
#                                                                                dMdo+hMy sMs  yM/   oMMMMM+       `:sNMMMMMm:  oMs yMo   sNNh+
#                                                                                dM:  `MM` dM::Ms    oMMMMMmhhhhddNMMMMMMMNo`  :MMssyMM:    -om
#                                                                                dMy..sMd  `mmNd     oMMMMMMMMMMMMMMMMMMmy:    `NM+:::+MN`-+-.-s
#                                                                                os/shy/    :MN.     /hhhhhhhhhhhyso+:`        /s+     +s/.syhys
#                                                                                         /+mN-
#                                                                                        /+:`
```

ROS kinetic Beginning

[2편 초급응용과 MAVROS 기초편]

[3편 Gazebo와 시뮬레이션편]

다선이가 책보고 따라하고 검색 하면서 만나는 문제점들을 해결하며 의식의 흐름대로 만든 속성 요점정리

출처 :

https://m.blog.naver.com/PostView.nhn?blogId=opusk&logNo=220984001237&proxyReferer=https%3A%2F%2Fwww.google.com%2F

http://wiki.ros.org/ko/ROS/Tutorials

https://github.com/EtainClub/etainclub/wiki/ROS

Programming Robots with ROS (Morgan Quigley) (박쥐책)

ROS 로봇 프로그래밍(표윤석)

외 기타등등 웹 검색

예제코드 일부 : https://github.com/Jpub/ROS

더블클릭 또는 Enter 키를 눌러 수정

initial installation

https://m.blog.naver.com/PostView.nhn?blogId=opusk&logNo=220984001237&proxyReferer=https%3A%2F%2Fwww.google.com%2F

---

tx2 페이지에서 처럼 설치 라고 하는데, 그냥 데스크탑에서 깔때처럼 깔면 된다.

For TX2

https://www.jetsonhacks.com/2017/03/27/robot-operating-system-ros-nvidia-jetson-tx2/

내 환경

Ubuntu 16.04

ROS kinetic

코딩은 VScode로 하는게 괜찮은것 같다 괜?괜?

작업공간 만들기

bashrc 과정 안거쳤을경우 아래 실행
$ source /opt/ros/kinetic/setup.bash

catkin 작업공간 만들기

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

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

```
$ catkin_init_workspace
```

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

catkin_ws에서 터미널 실행

```
!!!!!  $ source devel/setup.bash !!!!!!!
 이건 뭐 실행 시킬라면 터미널 열때마다 계속 실행시켜줘야된다.
```

그러기 싫으면 home 폴더의 숨겨진파일 .bashrc 를 열어

맨 아래에

source ~/catkin_ws/devel/setup.bash

를 추가하자 위 링크에서 설치했을때 이미 입력된거같으면 추가하면 안된다!

숨겨진파일은 ctrl+h를 눌러 확인할 수 있다.

패키지 만들기

catkin_ws/src 경로로 이동해서 터미널을 열건 cd로 경로로 들어간다

그리고 얘를 친다

```
~/catkin_ws/src$ catkin_create_pkg my_awesom_code rospy
```

```
my_awesome_code라는 패키지를 만드는 명령어이다
```

ROSRUN

rosrun을 수행하기 위해 먼저 roscore를 실행하여야함

별도의 터미널을 열어

```
$ roscore
```

```
이건 ros의 말그대로 core로 뭘 할라면 항상 켜놔야 한다
```

```
마지막줄에 started core service [/rosout]이라고 써져있으면 잘 되는거다
```

그런다음 위와 다른 터미널창에서 talker를 rosrun에서 실행

```
$ rosrun rospy_tutorials talker
```

1초에 10번씩 타임스탬프와 함께 hello world가 출력된다.

listener 노드 또한 실행해본다

```
$ rosrun rospy_tutorials listener
```

ROS에서 노드들이 연결된 구조를 graph라고 부르는거 같은데 그 그래프는 다음과 같이 볼 수 있다.

```
$ rqt_graph
```

이 상태에서 보면 talker와 listener가 연결된것을 볼 수 있다.

중복된 노드 처리

ROS에서 노드(가장 작은 처리모듈? 같은거) 와 메시지 스트림(토픽이라고 하며 노드간의 신호전달책으로 보임)와 매개변수는 모두 유일한 이름을 가져야함

하지만 같은 역할을 하는 노드들을 여러개 연결하고 싶으면 이름이 충돌남

이를 위해서 ROS 에서는 namespace와 remapping으로 이를 다룬다.

아까 위에처럼 한번 해본다.

```
$ roscore                ,얘는 안껏으면 안쳐도 되고
```

첫번째 talker 만들기(__ 이거는 _ 두번친거

```
$ rosrun rospy_tutorials talker __name:=talker1
```

두번째 talker 만들기

```
$ rosrun rospy_tutorials talker __name:=talker2
```

```
같은방식으로 좀 여러개 만들고
```

listener 하나 만들기

```
$ rosrun rospy_tutorials listener __name:=listener1
```

그리고 그래프 열어보기

```
$ rqt_graph
```

보면 listener 하나에 두개 talker가 말하고 있는게 보이고 listener에 뭐가 엄청많이 뜨는게 보인다.

roslaunch

근데 위와같은 문제를 해결하기위해 roslaunch는 이걸 다 알아서 해준단다.

```
$ roslaunch rospy_tutorials talker_listener.launch
```

이 다음에 새로운 터미널을 열어서 이름 지정 이런거 없이

```
$ rosrun rospy_tutorials talker
```

```
또 새로운 터미널을 열어서
```

```
    $ rosrun rospy_tutorials talker
```

몇개 쳐주면 roslaunch 했던 테이블에 talker 여러개가 엄청많이 말하는걸 볼 수 있다.

TAB키

```
$ rosrun rospy_tutorials ta
```

까지만 치고 탭키 한번 누르면

```
$ rosrun rospy_tutorials talker
```

가 자동완성되는것을 볼 수 있다.

토픽 TOPIC

topic은 노드사이의 비동기통신

publish/subscribe 통신기법을 구현한다.

topic을 통해 데이터를 전송하려면 노드에서는 토픽 이름과 전송된 메시지 자료형을 advertise(광고!) 해야한다.

그 다음에서야 노드는 이제 실제 데이터를 해당 토픽으로서 publish할 수 있다.

다른노드가 특정 토픽을 subscribe 하려면 그 다른노드는 roscore에 요청을 하게되고 subscribe 할 수 있다.

근데 ROS에서는 광고하는거며 요청하는거며 복잡한 내부 통신을 쉽게 처리해 준다.

토픽발행

연습용 패키지 만들기

```
$ cd catkin_ws/src          를 쳐서 src경로로 들어간다
```

basics라는 패키지를 하나 만든다

```
~$ catkin_create_pkg basics
```

```
만약 basics 경로에 src 폴더가 안생겼으면 만들어준다.
```

연습용 퍼블리셔 만들기

아래와 같은 코드를 만들고 파일을 해당 catkin_ws/src/basics/src 내에 topic_publisher.py 라고 저장한다.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy

from std_msgs.msg import Int32 # 토픽을통해 전송할 메시지 정의를 import

rospy.init_node('topic_publisher')

#package어디에 뭐 추가해서 publisher를 통하여 topic을 광고한다는데 무슨소린지...
pub = rospy.Publisher('counter', Int32) # topic에 counter라는 이름을 부여, 이때 publishr가 roscore에 연결을 설정하고 관련 정보를 보낸다.

```

```
'''

#이제부터 메시지를 실제로 publish 할 수 있게 된다.

rate = rospy.Rate(2) #publish 속도를 Hz단위로 설정한다.

count = 0
while not rospy.is_shutdown(): #rospy가 안꺼졌으면 돌려
  pub.publish(count)
  count += 1
  rate.sleep() # 대충 2Hz로 while문 실행하게끔 딜레이

  # 맨 윗줄은 interpreter로 python을 쓴다는소리고 (#!은 shebang이라고 부르며 인터프리터 지시어이다)
  # 그 아랫줄은 저게 없으면 한글주석 달았을때 오류나서 썼다.
```

위에껀 일단 저장만 해놓고,

광고를 아직 보지 않은 ros가 어떤 topic을 내줄 수 있을지 알아보자

물론 roscore가 실행중이어야 한다. 꺼졌다면 다시 실행시켜주자.

```
$ rostopic list
```

위와같이 하면 ros가 보내줄수있는 topic list가 나온다.

rostopic명령에서 사용할 수 있는 인자를 보려면

```
$ rostopic -h
```

를 치면 된다고 한다.

그리고 해당파일에 실행권한을 주기위해 python파일이 설치된 경로로 가서

```
~$ chmod u+x topic_publisher.py
```

를 쳐준다.

연습용 퍼블리시 노드 실행

그럼 이제 실행해보자

```
$ rosrun basics topic_publisher.py
```

안되면 catkin_ws폴더로 들어가서

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

를 치고 다시해보자 bashrc 수정하면 이거 안쳐도 된다그랬는데 난 자꾸 왜 안먹히지?

했는데 bashrc 맨밑 source에 명령이 중복되면 안먹히는거같다

이제

```
$ rostopic list
```

를 보면 /counter가 추가되어있는것을 볼 수 있다.

```
$ rostopic echo counter -n 10
```

을 치면 10번의 counter 토픽에 대한 값을 받아올 수 있다.

또 지정한 속도대로 잘 되고 있는지를 알아보기위해 다음과 같이 확인할 수 있다.

```
$ rostopic hz counter
```

그럼 rate로 나타나는 publishing frequency 값을 알 수 있다.

또한

```
$ rostopic info counter
```

를 이용하여 광고되는 토픽의 정보를 알 수 있다.

```
$ rostopic find std_msgs/Int32
```

를 이용하여 패키지 이름과 특정 자료형으로 publish되는 토픽을 모두 검색할 수 있다.

토픽 구독

일단 다음과같은 코드를 만들어 basics 패키지에 src에 저장한다.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy

from std_msgs.msg import Int32

def callback(msg):  #이렇게 짜놓으면 노드가 토픽을 구독하면 메시지가 도착할 때마다 그 메시지를 매개변수로 하여 과련 콜백 함수를 호출한다.
```

```
    print msg.data

rospy.init_node('topic_subscriber')

sub = rospy.Subscriber('counter', Int32, callback) # 이와같이 counter 토픽을 구독한다. (토픽명,자료형,콜백함수명)

rospy.spin()
# 이 부분은 ROS에게 제어를 넘긴다는 뜻이라고 한다.
# 이 함수는 노드가 종료될 준비가 될 경우에만 제어를 반환하게 한다.
# 이것은 최상위 수준의 while 반복문 정의를 회피할 수 있는 유용한 방법이라고 한다
```

그리 항상 파일을 만들고 실행시킬때는 실행권한을 주기위해서 파일이 설치된 경로에서 아래와같이 해야한다.

```
~$ chmod u+x topic_subscriber.py
```

그리고

```
$ rostopic list
```

를 통해 아직도 /counter 토픽이 광고되고 있는지 살펴보고 만약 안되고 있다면

```
$ rosrun basics topic_publisher.py
```

를 실행시켜 다시 켜자

그다음에

```
$ rosrun basics topic_subscriber.py
```

를 실행시키고 0.5초마다 숫자가 하나씩 올라가는지 살펴보자.

그런데 만약 안된다면

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

```
이걸 다시 실행시켜보거나
```

```
$ roscore
```

```
를 껏을수도 있으므로 실행시켜서 해보자
```

추가로

```
$ rostopic pub counter std_msgs/Int32 1000000
```

을 사용해서 임의의 publish를 강제할수도 있다.

```
$ rostopic info counter
```

를 통해서 새로운 퍼블리셔가 등장한것을 볼 수 있다.

래치된 토픽, Latched topic

latched topic은 데이터를 가끔 발행하는 노드설계에 사용된다.

ros 메시지는 순식간에 사라지고 메시지가 publish 될때 바로 subscribe하지 않으면 다음메시지로 넘어가 버린다.

예를들어 map_server node는 map 토픽으로 지도정보(nav_msgs/OccupancyGrid 자료형)를 advertise 한다.

이같은경우 지도는 변하지 않고 map_server가 저장장치로부터 실릴때 한번만 publish 된다.

이때 일반 토픽의 경우 이후에 다른노드가 지도정보를 subscribe 하고싶을때, 이 지도는 날아가서 지도정보를 얻을수 없을것이다.

물론 작은 frequency로 드문드문 보낼수도 있는데 이 빈도를 설정하는것이 쉽지 않을것이다.

latched topic은 이럴때 쓰면 좋은데. 토픽이 광고될때 latch 된다고 설정하면 subscriber는 토픽을 구독할때 제일 마지막에 전송된 메시지를 자동적으로 얻을 수 있게 된다. 이는 위에서 말했던 map과 같은 정보를 publish할때 한번만 publish하면 된다는것을 의미한다.

latched topic은 다음과같이 퍼블리셔 인자를 통해 설정할 수 있다.

```
pub = rospy.Publisher('map', nav_msgs/OccupancyGrid, latched=True)  //under python
```

자료형

파이썬과 C++ 를 각각 사용한 노드를 혼합하여 ros를 구동할때 자료형에서 문제가 생길 수 있다.

c++의 uint8[] 은 파이썬에서 문자열로 취급되고 ROS는 유니코드 문자열을 지원하지 않는다. UTF-8을 사용한다.

한글 주석 달때 파이썬에서 맨 위에 *#-- coding: utf-8 --* 를 넣어주자

새 메시지 (자료형) 정의

패키지의 msg 디렉터리의 특수한 메시지 정의 파일을 사용해서 ROS 메시지를 정의한다.

이경우 이 파일은 코드에서 사용할수 있는 언어별 구현으로 컴파일되는데

이말은 즉 파이썬같은 인터프리터 언어라도 사용자 메시지를 정의하려면

```
~$ catkin_make
```

를 실행해야 함을 뜻한다.

그리고 파일을 수정하고서도 $ catkin_make를 다시 실행해주어야 하며 그렇지 않으면 옛날코드 고대로 동작한다.

예제는 이전에 했던것처럼 정수를 publish 하는것이 아닌 임의의 복소수를 publish 하도록 하는것으로 진행한다.

basics 폴더에 msg 폴더를 만들어 (사실 예제코드에 다 있긴함, 지금알았네)

Complex.msg 파일을 만들고 다음처럼 입력한다.

```
1 float32 real
2 float32 imaginary
```

메시지를 정의하고 사용할 수 있도록 언어별 코드를 생성하는 catkin_make를 실행할 필요가 있다.

이 과정은 자료형 정의와 토픽전송을 위해 자료형을 마샬링하고 언마샬링하는 코드를 포함한다.

%마샬링 : ROS 메시지 자료형을 네트워크 전송에 적합한 자료형으로 변환하는 과정을 뜻함

%언마샬링: 은 그반대

그래서 다른언어로 작성된 노드들 사이에서 토픽을 구동할 수 있고 다른아키텍처를 가지는 컴퓨터간에도

매그럽게 통신할 수 있도록 메시지를 사용할 수 있게한다

ROS가 언어별 메시지 코드를 생성할 수 있도록 새로운 메시지 정의를 빌드 시스템에 알려줘야 한다.

이를위해 package.xml 파일에

```
<build_depend>message_generation</build_depend>

<run_depend>message_runtime</run_depend>  (나중에 exec 으로 하라고 오류가 날수도 있다. 그럼 그때가서 바꾸자)
```

```
책은 run_depend인데 오류나서 exec_depend로 바꿨는데 잘 되더라
```

를 추가한다. 어디다가?

```
<!--   <test_depend>gtest</test_depend> --> 얘 바로 밑에다가

인덴트는 위아래랑 비슷하게 맞춰서 해준다.

참고로 -->이건 화살표가 아니라 저렇게 써져있음
```

다음에는 CMakeList.txt 에 수정을 좀 해줘야 한다.

1. '#' 안달려있는 find_package를 찾는다.
2. find_package(catkin REQUIRED COMPONENTS

```
std_msgs</br>

message_generation
)
```

   이렇게 써준다

   이는 catkin이 message_generation패키지를 찾는것을 알 수 있도록 하기 위함이다.
3. '#' 안달려있는 catkin_package를 찾는다.
4. catkin_package(

```
CATKIN_DEPENDS message_runtime
```

)

이렇게 CATKIN_DEPENDS message_runtime 을 추가해 준다.

5. add_message_files를 찾는다.
6. '#' 없애버린다.
7. Complex.msg를 항목으로 추가한다.
8. generate_messages 를 찾는다.
9. DEPENDENCIES와 std_msgs를 추가한다.

!!!!!기억해야될게 닫는 괄호의 주석처리도 지워줘야함

새로 생성된 메시지 정의는 checkSum을 포함한다. ROS는 이게 메시지의 올바른 버전을 사용하고 있는지 확인하는데 사용한다.

메시지 정의를 수정하고 catkin_make를 실행해도 이 메시지를 사용하는 다른 코드를 수정하고서 또한 catkin_make를 실행해야한다.

새로운 메시지 사용법

src 폴더에 message_publisher.py 라는 새 파일을 만들어주고 다음과같이 코딩한다.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy                                # import rospy
from basics.msg import Complex              # basic의 msg 항목에서 Complex를 불러와
from random import random                   # 랜덤함수 쓸수잇게 불러와

rospy.init_node('message_publisher')        # initialization of message_publisher node
pub = rospy.Publisher('complex', Complex)   # (토픽이름 complex, 자료형 Complex)

```

```
rate = rospy.Rate(2)                        # 송신빈도 2Hz

while not rospy.is_shutdown():              # rospy 꺼지지 않았으면 아래 계속 실행
    msg = Complex()                         # 메시지는 complex
    msg.real = random()                     # 실수부 랜덤값
    msg.imaginary =random()                 # 허수부 랜덤값
    pub.publish(msg)                        # msg를 퍼블리시
    rate.sleep()                            # 다음 송신주기까지 휴식
```

또 message_subscriber.py 라는 새 파일을 만들어주고 다음과같이 코딩한다

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy
from basic.msg import Complex

def callback(msg):
    print 'Real: ', msg.real
    print 'Imaginary: ',msg.imaginary
    print

rospy.init_node('message_subscriber')               # subscriber 노드 초기화

sub = rospy.Subscriber('complex', Complex, callback)# (받는토픽, 자료형, 호출할 함수)

rospy.spin()                                          # ROS에 통제권한 넘김
```

항상 코드실행전에는 권한 설정 해줘야되는거 맨날까먹네이씨

For you to remind

```
~$ chmod u+x message_publisher.py
```

```
~$ chmod u+x message_subscriber.py
```

로 활성화 먼저 해주고

```
$ rosrun basics message_publisher.py
```

로 퍼블리시 실행

```
$ rostopic echo complex -n 5
```

명령으로 잘 나오고 있나 볼 수 있음

```
$ rostopic hz complex
```

명령으로 주파수를 어떻게 갖는지 볼 수 있음

```
$ rqt_graph
```

로 그래프 상황 볼 수 있음

그럼 이제 여기서 subscribe를 실행시키자.

```
$ rosrun basics message_subscriber.py
```

값이 잘 나오나 확인해 보자.

```
$ rosmsg show Complex
```

를 이용하면 Complex 자료형의 내용을 볼 수 있다.

```
$ rosmsg show PointStamped
```

는 뭔지 모르겠다.

```
$ rosmsg list
```

는 ros에서 사용할 수 있는 모든 메시지를 보여준다.

```
$ rosmsg packages
```

는 메시지를 정의하는 모든 패키지를 보여준다.

```
$ rosmsg package basics
```

는 basics 패키지에서 정의한 메시지 목록을 보여준다.

자료형은 언제 만들어야되나?

반드시 만들어야 할때만 만들어야한단다.
웬만하면 기본 제공되는 메시지 자료형을 사용하시기를.

Publisher와 Subscriber의 퓨전

충격!

사실은 하나의 노드가 publisher와 subscriber 둘다 동시에 될 수 있을 뿐만 아니라

여러개를 publish 할 수도 있고 여러개를 subscribe 할 수도 있다!

박쥐 책에서는 예를들어 노드는 카메라 이미지가 있는 토픽을 subscribe하고 이미지에 있는 얼굴을 식별하고

다른 토픽으로서 그 얼굴의 위치를 publish 할 수도 있다 라고 했다.

실험을 위해 src 폴더에 doubler.py 파일을 만들고 다음과 같이 코딩한다.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy                                    # 이건 뭐 그냥 있는거고
from std_msgs.msg import Int32                  # standard message 팩에서 Int32 자료형 취하는거고
from random import random
from basics.msg import Complex

rospy.init_node('doubler')                      # 노드명 doubler

def callback(msg):                              # 세번 말하면 잔소리
  doubled = Int32()                             # doubled 는 int32
  doubled.data = msg.data * 2                   # 콜백을 호출한 subscriber가 받은 data가 msg.data에 들어가는 모양 이걸 2로 곱함
  print 'dat : ', doubled
  outout = Complex()
  outout.real = random()
  outout.imaginary = random()
  pub.publish(outout)                          # 뱉어내기

sub = rospy.Subscriber('counter', Int32, callback)
pub = rospy.Publisher('complex', Complex)

rospy.spin()
```

파이썬 파일을 만들고서 제일먼저 실행권한 설정해줘야 한다. (4번째)

```
$ chmod u+x doubler.py
```

그리고 doubler 노드먼저 실행시켜준다.

```
$ rosrun basics doubler.py
```

```
$ rqt_graph를 보면 노드는 하나로 표시된것을 볼 수 있다.
```

그리고 옛날에 만들었었던 카운터 토픽으로 데이터 나오는 topic_publisher를 추가 실행시켜준다.

```
$ rosrun basics topic_publisher.py
```

doubler를 실행시킨 터미널에서 2의 배수들이 나오고 있는것을 볼 수 있다.

나아가 얼마전에 했던 message_subscriber도 추가 실행시켜준다.

```
$ rosrun basics message_subscriber.py
```

그리고 그래프를 보자

```
$ rqt_graph
```

doubler 노드는 publisher도 할 뿐만 아니라 subscriber도 하고있음을 볼 수 있다.

토픽은 여기까지

토픽에 대해서 더 자세히 알고싶다면

http://wiki.ros.org/Topics?distro=kinetic

를 방문하자!

서비스 Service

서비스는 ROS에서 노드간 데이터를 전송하는 또다른 방법

서비스는 동기화된 원격 프로시져 호출을 뜻함

즉 한 노드가 다른 노드에서 실행되는 함수 호출을 가능하게 한다는 뜻

새 메시지 자료형을 정의하는 방법과 비슷하게 이 함수의 입력과 출력을 정의한다.

서비스를 제공하는 서버는 서비스 요청을 다루는 콜백을 정의하고 서비스를 광고(advertise)한다.

서비스 호출은 가끔 해야 할 필요가 있고 수행하는 데 제한된 시간이 필요한 일에 매우 적합하다..

다른 컴퓨터로 배포하기 원하는 공통적인 계산이 그런 예이다.

센서를 동작시키거나 카메라로 고해상도 사진을 촬영하는 것처럼 로봇이 해야하는 discret한 액션도

서비스 호출 구현을 위한 좋은 후보라고 한다.

여기서는 문자열에 단어수를 세는 서비스 구현 방법을 다룬다.

서비스 정의

서비스 구축의 첫 단계는 서비스 호출의 입출력 정의이다.

이는 메시지 정의파일 구조랑 유사한 service-definition file에서 할 수 있다.

basics 폴더에 srv 폴더를 만들고 WordCount.srv 파일을 생성하고 아래와 같이 코딩한다.

```
1 string words
```

```
2 ---
3 uint32 count
```

--- 는 구분기호 이고 이 위로는 입력 정의 아래는 출력 정의이다.

또 재미없는 CMakeList.txt 편집을 시작해야한다.

CMakeList 파일을 열자.

1. find_package 를 찾자

2. message_generation 을 포함하고 있는것을 확인하자.

3. package.xml 파일에서

4. <!-- <test_depend>gtest</test_depend> --> 밑에

5. <build_depend>rospy</build_depend>
   <exec_depend>rospy</exec_depend>

   <build_depend>message_generation</build_depend>
   <exec_depend>message_runtime</exec_depend>가 존재하는지 확인하자.
   물론 없으면 추가

   책은 indigo 버전이라그런지 exec이 아닌 run을 쓰라는데 그러면 오류나더라

6. 다시 Cmakelist로 돌아와서 add_service_files()를 사용하여
   어떤 서비스 정의파일을 컴파일 하기 원하는지 catkin에게 알려주자

7. add_service_files 의 주석을 해제, FILES와 WordCount.srv를 추가 (이후에 서비스 추가할라면 이것만 하면 대겟네)
   거듭 말하지만 닫는 괄호도 주석 해제 해줘야한다 ㅠㅠ

```
8. 마지막으로 generate_messages 에서 DEPENDENCIES가 선언되었는지 확인해야한다.
```

catkin_ws 에서

```
~$ catkin_make
```

```
해준다.
```

이 단계에서 WordCount, WordCountRequest, WordCountResponse 세개의 클래스가 생성된다고 한다.
얘네들은 WordCount.srv 파일에서 비롯되는 이름인것같다.

```
$ rossrv show WordCount
```

를 보면 service 호출 정의가 우리가 만들었던 대로인지 검증할 수 있다.

서비스 구현

서비스에 대한 입출력 정의하고, cmakelist 정리하고 package 정리했고 catkin_make 했으면
이제 서비스 구현 코드를 작성할 수 있게 된다.
토픽과 마찬가지로 서비스도 콜백 기반 메커니즘을 사용한다고 한다.
서비스 제공자는 서비스가 호출될 때 실행되는 콜백을 정의하고, 요청이 오기만을 손꼽아 기다린다.

서버 만들기

src 폴더에 다음 코드를 넣어 service_server.py라는 이름으로 파일을 만들자

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import rospy                                                    #항상 넣는거고

from basics.srv import WordCount, WordCountResponse             #WordCount와 WordCountRespose를 둘다 불러와야된다.
                                                                #얘네는 파이썬 모듈 basic.srv 에 생성된 것이다.

def count_words(request):                                       # WordCountRequest 자료형의 인자하나를 받아서
    return WordCountResponse(len(request.words.split()))        # 단어갯수를 세서 WordCountResponse 자료형을 반환한다.

rospy.init_node('service_server')                               # service_server 이름을 가진 노드 초기화

service = rospy.Service('word_count', WordCount, count_words)   # (서비스이름, 자료형, 서비스구현 콜백)

rospy.spin()                                                    # 노드제어를 ROS에 넘겨주며 노드가 종료될 준비가 됐을때 빠져나온다.
                                                                # 하지만 콜백이 자신의 스레드에서 동작하므로 얘를 호출해서 실제로 제어를 넘겨줄
                                                                # 다른 일을 할 필요가 있으면 노드 종료를 확인하면서 자신만의 반복문 설정이 가능
                                                                # 그러나 rospy.spin()을 사용하는게, 종료될 준비가 되기 전까지 노드가 살아있도록
```

파이썬 파일을 만들고서 제일먼저 그 파일 경로에서 아래와같이 실행권한 설정해줘야 한다.(5번째로 까먹음)

```
$ chmod u+x service_server.py
```

결과확인

서비스를 실행시켜 보자

```
$ rosrun basics service_server.py
```

서비스가 실행되고 있는지 확인해보자

```
$ rosservice list
```

서비스에 관한 정보를 얻어보자

```
$ rosservice info word_count
```

```
$ rosservice type word_count
```

```
$ rosservice args word_count
```

서비스에서 값 반환하는 다른방법

위에서는 WordCountResponse 객체를 명시적으로 생성해서 서비스 콜백에서 반환했다.

서비스 콜백으로부터 값을 반환하는 방법에는 여러가지 다른 방법들이 있다.

서비스에 대한 한개의 Return 값이 있을 때는 간단하게 그냥 그 값을 반환하면 된다.

```
1 def count_words(request):
2   return len(request.words.split())
```

Return이 여러개면 튜플이나 리스트를 반환하면 된다.

```
1 def count_words(request):
2   return [len(request.words.split())]
```

dictionary 형태로도 Return 가능하다 (Wow)

```
def count_words(request):
  return {'count' : len(request.words.split())}
```

서비스 사용해보기

서비스는 가장 간한하게는 터미널에서 rosservice 명령으로 가능하다.

```
$ rosservice call word_count 'Flight Dynamics and Control'
```

하지만 이런 방법은 다른노드에서 정보를 받아오기에 적절치 않다.

클라이언트 만들기

다음과 같이 src 폴더에 service_client.py라는 이름의 파일을 만들고 코딩하자

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy

from basics.srv import WordCount

import sys

rospy.init_node('service_client')                    # 노드 실행

rospy.wait_for_service('word_count')                 # 서비스가 광고될때까지 대기탄다.
```

```
12
13 word_counter = rospy.ServiceProxy('word_count', WordCount)  # 서비스가 광고되고나면 지역 프록시를 설정할 수 있다.
14                                                            # (서비스이름, 자료형)
15
16 words = ' '.join(sys.argv[1:])                             # $ rosrun basics service_client.py {} 에서 {}부분 받아오기
17
18 word_count = word_counter(words)                           # word를 서버에 보내서 받아와, word_count에 count요소로 값이 들어가게됨(WordCount.s
19
20 print words, '->', word_count.count
```

이제는 익숙해 졌겠지만 항상 코드만들고

```
$ cd catkin_ws/src/basics/src
```

한다음에

```
$ chmod u+x service_client.py
```

해줘서 실행권한 줘야한다.

TestTest

```
$ rosrun basics service_client.py Flight Dynamics and Control Laboratory
```

를 친다. (rosrun은 항상 기본경로 home 경로에서 실행)

다른 서비스 호출방법

위의 예제는 하나의 인자를 받고 하나의 인자를 뱉는 서비스를 활용했다

반면에

```
string first_word
string second_word
string third_word
---
string short_ver
int16 how_many
```

이런 정의를 갖고있는 서비스가 있다면 프록시 함수는 세개의 인자를 받아서 두개의 값을 반환할 것이다.

연습을 위해 srv 폴더에 MakeShort.srv 파일을 만들어 위의 정의를 입력해주자.

그리고 CMakeList.txt 에 접근해서

add_service_files 에, MakeShort.srv 파일을 추가하자.

그리고 당연히 catkin_make 돌려줘야된다.

서버파일의 코드는 다음과 같다.
makeshort_server.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import rospy

from basics.srv import MakeShort,MakeShortResponse


def make_short(request):
    squeezed = request.first_word[0]₩
            + request.second_word[0]₩
```

```
            + request.third_word[0]
    return [squeezed, len(squeezed)]

rospy.init_node('short_service_server')

service = rospy.Service('make_them_short', MakeShort, make_short)

rospy.spin()

```

클라이언트파일의 코드는 다음과 같다.

makeshort_client.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import rospy

from basics.srv import MakeShort

import sys

rospy.init_node('short_service_client')

rospy.wait_for_service('make_them_short')

short_maker = rospy.ServiceProxy('make_them_short', MakeShort)


words = ' '.join(sys.argv[1:])

sliced_words = words.split()

shorten = short_maker(sliced_words[0], sliced_words[1], sliced_words[2])

print words, '->', shorten.short_ver, '->', shorten.how_many
```

예제를 급조해서, 굉장히 투박하고 아무 의미없는 service 이지만 이해를 도울 가능성이 조금 있다.

그럼 실행해 보자

```
$ rosrun basics makeshort_client.py Flight Dynamics Control
```

Recap

토픽은 publisher 가 계속해서 흘려 보내는 데이터를 subscriber가 주섬주섬 줏어담는 느낌적인 느낌이고 UART같은 느낌
서비스는 client가 받고싶은 데이터를 그때그때 server에 요청해서 받는 느낌적인 느낌이다. SPI I2C 같은 느낌

액션 Action

약간 먼 위치로 이동하도록 로봇에게 명령하는 goto_position 함수를 가정해보자.
앞에 뭐가 있을지도 몰라서 정확한 시간을 미리 알수는 없다.
로봇은 이를 수행하는데 상당한 시간이 필요할 것이다.
만약 서비스 인터페이스로 로봇에게 무슨일 있냐고 물어보면
클아이언트는 그 답을 막연히 기다리기만 한다.
그동안 프로그램은 정지하게되고 목적지로 가는 로봇의 진행에 관한 정보를 얻을 수 없다.

이런것을 위해 ROS 에서는 Action을 제공한다.

ROS Action은 긴시간이 소요되는 목표 지향적인 행위의 인터페이스를 구현하는 데 가장 좋은 방법이다.

서비스가 동기적인데 비해 액션은 비동기적이다.

서비스의 요청, 응답과 마찬가지로 액션은 행위를 시작하기위한 goal을 사용하며

행위가 완료되면 result를 보낸다.

나아가 action은 목표에 대한 행위의 진행 정보 갱신을 보내주기 위해 feedback을 사용하고 목표 취소 또한 허용한다.

액션 자체는 사실 Topic을 사용하여 구현된다.

액션은 본질적으로 토픽의 집합(목표, 결과, 피드백)을 짝지어 사용하는 방법을 구사하는 높은수준의 프로토콜이다.

액션은 서비스보다 정의와 사용에 좀더 많은 노력이 필요하지만

보다 많은 권한과 유연성을 제공한다.

액션 정의

액션 정의파일은 보통 .action 접미사를 가진다

그 모습은 서비스 정의와 비슷하지만 추가적인 필드가 있다.

첫번째 예시는 타이머 이다 더 유용한 예제는 나중에 다룬다고 한다.

이 예시는 지정된 시간이 경과했을 때 알려주는 초 읽기 타이머 이고,

시간이 지나면서 주기적으로시간이 얼마나 남았는지 알려 주어야 한다.

타이머가 종료되면 실제 경과 시간을 알려 주어야 한다.

사실 근데 나중엔 rospy.sleep()과 같은 ros 클라이언트 라이브러리에 내장된 시간 지원을 사용하면 된단다.

먼저 action 폴더에 Timer.action 파일로 정의를 해보도록 하자

```
# --- 기호도 세개의 영역으로 구분되며 각 부분은 각각 목표, 결과, 피드백 세개 부분이다.
# 이부분은 목표이며 클라이언트에서 보낸다.
duration time_to_wait   # 대기하기 원하는 시간
---
# 이부분은 결과이며 종료시 서버가 보낸다.
duration time_elapsed   # 대기 시간
uint32 updates_sent     # 진행되는 동안 제공한 업데이트 횟수
---
# 이부분은 피드백이고 실행되는동안 서버가 주기적으로 보낸다.
duration time_elapsed   # 시작부터 경과된 시간
duration time_remaining # 종료될때까지 남은시간
```

또 재미없는 CMakeList.txt 편집을 해보자.

1. find_package 를 찾는다.

2. actionlib_msgs 를 추가한다.

3. add_action_file 을 찾아내서 주석 해제한다. (끝내기 괄호 주석 해제를 잊지 말아야 한다)

4. 그 안에 DIRECTORY action 과 FILES Timer.action 두개를 추가한다.

5. 액션에 대한 Dependancy 설정이 있어야 한다. generate_messages를 찾아

   DEPENDENCIES 밑에 actionlib_msgs 를 추가하자

6. 마지막으로 catkin_package 를 찾아 캣킨에 대한 의존성으로 추가해야한다

   CATKIN_DEPENDS 아래에 actionlib_msgs 를 추가하자.

7. 이제 뭘해야 하는지 곰곰히 생각해보자

   그리고 뭘 해야할지 3초안에 떠오르지 않는다면 첫 페이지로 되돌아가서 처음부터 다시 시작하자!

정답 보기.

그리고 이게 맞는건지 모르겠지만 이거 안하면 자꾸 오류가 나서 해본건데

package.xml 에 들어가서 test_depend 아래쪽에

```
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

요거 넣어주자

그러면 빌드 하더라

이 과정에서 많은 것들이 생성되는데

```
TimerAction.msg
TimerActionFeedback.msg
TimerActionGoal.msg
TimerActionResult.msg
TimerFeedback.msg
TimerGoal.msg
TimerResult.msg
```

이렇게 생성된다. 그거참 신통방통하다.

이 메시지들은 ROS 토픽상에 만들어져 액션 클라이언트/서버 프로토콜을 구현하는데 사용된다.

생성된 메시지 정의가 차례대로 메시지 생성기에 의해 처리되서 해당하는 클래스 정의 파일을 생성한다.

하지만 많은경우 이들 클래스중 몇가지만 사용한다.

기본적인 액션 서버를 구현해 보자 가장 쉬운방법은 actionlib 패키지에 있는 simpleActionServer 클래스를 사용하느 것이다.

액션 클라이언트가 새로운 목표를 보냈을 때 호출되는 콜백만을 정의 함으로써 시작한다.

콜백에서 타이머 작업을 하고 난 후 끝났을 때 결과를 반환한다.

src 폴더에 simple_action_server.py 파일을 만들어 다음과 같이 입력하자.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import rospy
import time                                                                 # 파이썬 표준 time 패키지
import actionlib                                                            # simpleactionserver 클래스등을 제공하는 패키지

from basics.msg import TimerAction, TimerGoal, TimerResult                  # cmake할때 자동 생성된 몇몇 메시지 클래스

def do_timer(goal):                                                         # 콜백 함수 정의 goal의 자료형은 TimerGoal
                                                                            #       Timer.action의 목표영역에 대응한다
    start_time = time.time()                                                # 현재 시각 저장
    time.sleep(goal.time_to_wait.to_sec())                                  # time_to_wait 필드를 초단위로 변경, 목표에서 요청 시간동안 일시정지
    result = TimerResult()                                                  # result는 TimerResult 자료형
                                                                            #       이는 Timer.action의 결과 영역에 대응한다.
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time) # 현재시각에서 시작시간을 차감해 ROS duration 자료형으로 변환 후 필
    result.updates_sent = 0                                                 # 이 과정 동안 어떤 갱신 정보도 보내지 않았으므로 0으로 초기화
    server.set_succeeded(result)                                            # set_succeeded를 호출하여 결과를 전송하여 목표를 달성했음을
                                                                            #       SimpleActionServer에 알리는 것이다. 얘는 뭐 맨날 성공하겠지

rospy.init_node('timer_action_server')                                      # 노드 생성
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)# (광고될 서버이름, 액션의 자료형, 목표 콜백, 자동시작 여부)
server.start()                                                              # 서버 시 작
rospy.spin()
```

주의할께, 액션 서버 자동시작을 활성화 해 버리면 race condition이라는 골치아픈 문제를 만든다고 한다.

항상 False로 해놓고 이후에 server.start()로 실행하는 방식으로 짜자.

이제 액션 서버를 실행시켜보자

```
$ rosrun basics simple_action_server.py
```

혹시 실행권한 설정을 안했거나 roscore를 실행시키지 않았거나 basics를 basic이라고 쳐서 에러가 났다면
첫페이지로 돌아가서 다시 시작하자. 난 벌써 세번째 다시 시작하고 오는 중이다.

실행이 되고 있다면 다른 터미널을 열어

```
$ rostopic list
```

를 실행시켜
timer /cancer /feedback /goal /result /status
가 존재하는지 확인하자.

```
$ rosrun basics simple_action_server.py
```

를 실행시켜서 자세히 살펴볼 수 있다.

```
$ rosmsg show TimerActionGoal
```

를 보면 goal.time_to_wait 필드에서 처럼 목표가 정의된 것을 찾을 수 있는데, 명시하지 않은 다른 필드들도 들어 있음을 알 수 있다.
얘네들은 액션 서버와 클라이언트 코드에서 사용하는것으로
어떤일이 벌어지고 있는지 추적할 수 있게 한다.
다행히도 서버코드가 목표 메시지를 읽기전에 자동으로 제거된다.

결국 목표 실행에서 볼수 있는것은 .action에서 정의한 그대로 TimerGoal 메시지이다.

```
$ rosmag show TimerGoal
```

액션 사용하기 Using an Action

다음과 같이 src 폴더에 simple_action_client.py 파일을 만들어준다.

```
#! /usr/bin/env python
#-*- coding: utf-8 -*-
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')

client = actionlib.SimpleActionClient('timer', TimerAction)            # (action 서버 이름, 액션의 type) 둘다 서버와 일치해야함
client.wait_for_server()                                               # 서버가 실행되기를 손꼽아 기다리기

goal = TimerGoal()                                                     # TimerGoal 자료형의 goal 생성
goal.time_to_wait = rospy.Duration.from_sec(5.0)                       # 골 설정 같은 느낌?
client.send_goal(goal)                                                 # goal 메시지를 서버에 보냄

client.wait_for_result()                                               # 결과를 받을때까지 대기탄다.
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))  # 결과가 들어오면 클라이언트 오브젝트의 get_result()로 받아올수 있다.
                                                                       # result 중에서도 소요시간을 초로 받아와서 출력한다.


```

제대로 되는지 보자

확인사항!

1. roscore
2. chmod u+x
3. 서버 실행 선행

```
$ rosrun basics simple_action_client.py
```

요청한 대로 약 5초정도의 elapsed time을 뱉어내는것을 볼 수 있다.

좀더 복잡한 액션 서버 구현

지금까지는 왠지 동기화 된 느낌의 구현으로 서비스와 매우 똑같다는 느낌을 지우기 어려웠을 것이다.

이제 서비스와 구별되는 액션의 비 동기적인 부분을 알아보자

서버 측에서 시작하여 약간의 수정을 통해 목표를 중단시키는 방법

목표 선점 요청을 처리하는 방법

목표를 추적하면서 피드백 제공 방법을 보여준다.

좀더 멋있는 액션 서버코드는 다음과 같이 src 폴더에 fancy_action_server.py 라는 이름으로 만들어보자

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

```
#-*- coding: utf-8 -*-

import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback          # feedback 제공하기위해 새로운자료형 추가

def do_timer(goal):                                                                # 뻔한 콜백함수
    start_time = time.time()
    update_count = 0                                                               # feedback이 몇번 발생하는지 알수있는 변수 만들고, 초기화
    # 오류 확인코드/60초 이상으로 설정됐으면 set_aborted() 호출하여 중지. set_succeeded()처럼 결과를 포함한다.
    if goal.time_to_wait.to_sec() > 60.0:
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Timer aborted due to too-long wait")           # 너무 긴 시간 요청으로 종료
        return                                                                     # 이 목표를 종료했으므로 콜백으로부터 돌아간단 얘기

    while (time.time() - start_time) < goal.time_to_wait.to_sec():                 # 이제 요청시간동안 조금식 일시정지 하면서 while 반복문 돌기

        if server.is_preempt_requested():                                          # 선점 여부를 확인한다. 이함수는 클라이언트가 목표추적을 멈
                                                                                   # 새로운 목표를 요청하면 True를 반환한다.
            result = TimerResult()
            result.time_elapsed = W
                rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")                        # 만약 선점되었다면 set_preempted()를 호출하여 결과를 채우
            return

        feedback = TimerFeedback()                                                 # TimerFeedback()자료형의 feedback을 만든다.
                                                                                   # 이는 Timer.action의 피드백 영역에 해당한다.
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)  # 적당한 정보들을
        feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed        # 적당히 채워넣어서
        server.publish_feedback(feedback)                                          # feedback을 퍼블리시 하고
        update_count += 1                                                          # 업데이트 횟수를 하나 더한다.

        time.sleep(1.0)                                                            # 잠시 일시정지
```

```
        time.sleep(1.0)                                                 # 잠시 일시정지.
                                                                        # 요청한 시간보다 더 길게 일시정지할 수 있어 이것은 타이머를

    result = TimerResult()                                              # 반복문을 빠져나왔단것은 요청한 기간동안 성공적으로 일시정
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)  # 적한한 값들
    result.updates_sent = update_count                                  # 적당히 채워서
    server.set_succeeded(result, "Timer completed successfully")        # succeeded 메세지 보낸다.

rospy.init_node('timer_action_server')                                  # Simple 때랑 똑같음
server = actionlib.SimpleActionServer('timer',TimerAction, do_timer, False)
server.start()
rospy.spin()
```

까먹기전에 권한설정 해주자

이렇게 서버를 만들었으면 클라이언트도 만들어야겟지?

src 폴더에 fancy_action_client.py 파일을 아래와같이 만들어보자.

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import rospy
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_cb(feedback):                                              # 콜백
    print('[Feedback] Time elapsed : %f'%(feedback.time_elapsed.to_sec()))  # 피드백 받은거 출력
    print('[feedback] Time remaining : %f'%(feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client')                                  # 클라이언트 초기화
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()                                                # 서버 기다리기

goal = TimerGoal()                                                      # goal 은 TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)                        # 5초 타이머 요청 대입
```

```
#goal.time_to_wait = rospy.Duration.from_sec(500.0)                    # 중단테스트 중단중단
client.send_goal(goal, feedback_cb = feedback_cb)                     # 요청 보내

#time.sleep(3.0)                                                      # 선점테스트 선점선점
#client.cancel_goal()                                                 # 선점테스트 선점선점

client.wait_for_result()                                              # Result 기다리기
print('[Result] State: %d'%(client.get_state()))                      # 2: 선점 3: 성공 4: 거부됨 등등 10개의 메시지 있음
print('[Result] Status: %s'%(client.get_goal_status_text()))          # 서버에서 보낸 상태 텍스트 출력
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

결과 확인

실행 권한 설정은 진즉에 했었어야 됐고

이제 예상 결과를 확인해보자

```
$ roscore
```

코어 실행해 주고.

```
$ rosrun basics fancy_action_server.py
```

실행시켜주자

주의사항 : 노드명은 아까 예제의 simple 액션 서버로 돌아간다. 따라서 중복실행이 안될꺼다.

이제 클라이언트를 실행시켜보자

```
$ rosrun basics fancy_action_client.py
```

잘 되는것을 확인하자.

또 코드에 주석처리되있는 여러가지 예외도 주석 해제하여 확인해 보자!

Summary

토픽, 서비스, 액션 비교

자료형	사용 시 최선 분야
토픽	단방향 통신, 특히 다수의 노드가 청취하고 있을때(예, 센서 데이터 스트림
서비스	노드의 현재 상태 질의와 같은 간단한 요청/응답 상호작용
액션	특히, 요청 처리가 순간적이지 않은(예, 목표 위치까지 주행) 대부분의 요청/응답 상호작용

액션은 로봇 프로그래밍의 많은 측면에서 적합한 점을 가지고 있다. 액션은 사용하기에 조금 더 많은 코드를 요구하지만 ,

반대로 서비스보다 훨씬 더 많이 강력하고 확장성이 있다. 앞으로 여러개의 복잡한 행위에 대해 풍부하지만 사용하기 쉬운

인터페이스를 제공하는 액션의 많은 사례를 볼 것이다.

이 장에서도 전체 API를 다루지 않으셨다고 한다. 더 알고싶다면 자세한 사항은

여기 를 참고하자.

첫번째 정리 끝~

다음 정리 페이지로 가기

1