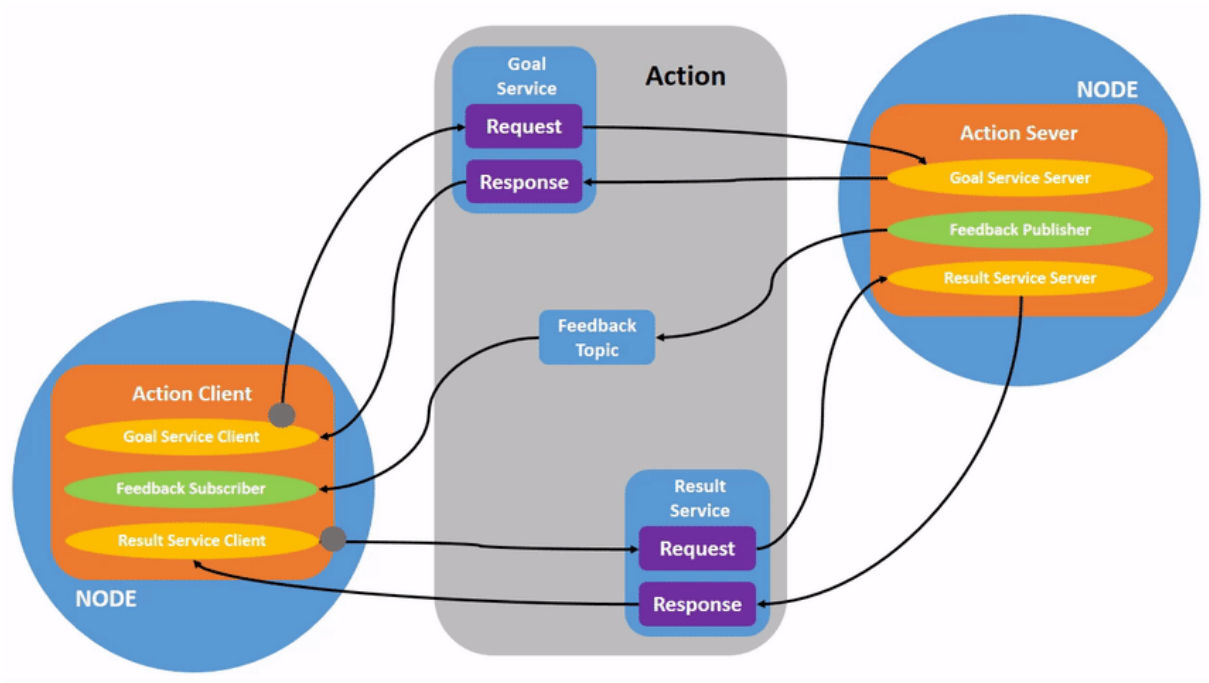
Action 프로그래밍 - Python

해당 실습 자료는 한양대학교 Road Balance - ROS 2 for G Camp와 ROS 2 Documentation: Foxy, 표윤석, 임태훈 <ROS 2로 시작하는 로봇 프로그래밍> 루피페이퍼(2022) 를 참고하여 작성하였습니다.

> 이번 장에서는 `Server Node`와 `Client Node` 간의 메세지 통신 `Action`을 구현해볼 예정입니다.



- Client
- Server

<1. Action 인터페이스 패키지 만들기>

Creating custom msg and srv files — ROS 2 Documentation: Foxy documentation
https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html#create-a-new-package

인터페이스

- 노드 간의 데이터를 주고받을 때 사용되는 데이터의 형태를 **인터페이스(interface)**라하며, 유저가 원하는 형태로 구성된 인터페이스를 생성할 수 있습니다.
  (
  단순 자료형을 기본으로하며 메시지를 포함하는 간단한 데이터 구조 및 메시지들이 나열된 배열 구조로 사용할 수 있습니다.)
  - 메시지를 포함하는 간단한 데이터 구조 및 메시지들이 나열된 배열 구조는 단순 자료형을 기반으로하며 정의시 아래와 같이 기술합니다.
    - fieldtype은 메시지 자료형, fieldname은 메시지 이름에 해당합니다.

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3


##################################
## example
##################################
float64 x
float32[] ranges
string datas
##################################
```

▼ [참고]기본 자료형과 언어별 자료형 매칭

Type name	Python	C++	DDS type
bool	builtins.bool	bool	boolean
byte	builtins.bytes*	uint8_t	octet
char	builtins.str*	char	char
float32	builtins.float*	float	float
float64	builtins.float*	double	double
int8	builtins.int*	int8_t	octet
uint8	builtins.int*	uint8_t	octet
int16	builtins.int*	int16_t	short
uint16	builtins.int*	uint16_t	unsigned short
int32	builtins.int*	int32_t	long
uint32	builtins.int*	uint32_t	unsigned long
int64	builtins.int*	int64_t	long long
uint64	builtins.int*	uint64_t	unsigned long long
string	builtins.str	std::string	string
wstring	builtins.str	std::u16string	wstring
static array	builtins.list*	std::array<T, N>	T[N]
unbounded dynamic array	builtins.list	std::vector	sequence
bounded dynamic array	builtins.list*	custom_class<T, N>	sequence<T, N>
bounded string	builtins.str*	std::string	string

[출처] 016 ROS 2 인터페이스 (interface) (오픈소스 소프트웨어 & 하드웨어: 로봇 기술 공유 카페 (오로카)) | 작성자 표윤석

- 인터페이스만으로 구성된 별도의 패키지를 만드는 것이 의존성을 관리하기 편합니다.

인터페이스

	msg 인터페이스	srv 인터페이스	action 인터페이스
확장자	*.msg	*.srv	*.action
데이터	토픽 데이터 (data)	서비스 요청 (request) --- 서비스 응답 (response)	액션 목표 (goal) --- 액션 결과 (result) --- 액션 피드백 (feedback)
형식	fieldtype1 fieldname1 fieldtype2 fieldname2 fieldtype3 fieldname3	fieldtype1 fieldname1 fieldtype2 fieldname2 --- fieldtype3 fieldname3 fieldtype4 fieldname4	fieldtype1 fieldname1 fieldtype2 fieldname2 --- fieldtype3 fieldname3 fieldtype4 fieldname4 --- fieldtype5 fieldname5 fieldtype6 fieldname6

▼ 👉방법1: 직접 인터페이스 패키지 만들기

1. 패키지 생성 및 디렉토리 생성

- 인터페이스는 c++을 이용하여 생성합니다.(패키지 이름: custom_action_interface)

```
$ cd ~/ros2_ws/src
$ ros2 pkg create custom_action_interface --build-type ament_cmake
```

```
$ cd custom_action_interface
$ mkdir action
```

▼ 생성된 디렉토리 구조

```
.
├── action
├── include
│   └── my_first_ros_rclcpp_pkg
├── src
├── CMakeLists.txt
└── package.xml

4 directories, 2 files
```

2. Fibonacci.action 생성

- 경로: `~/ros2_ws/src/custom_action_interface/action`

```
$ cd ~/ros2_ws/src/custom_action_interface/action

## action 예제를 위한 Fibonacci.action 파일 생성
$ gedit Fibonacci.action
```

- **`Fibonacci.action` 파일 내용**

```
# Goal
int32 order
---
# Result
int32[] sequence
---
# Feedback
int32[] partial_sequence
```

3. 관련 설정 파일 수정하기

- 경로: `~/ros2_ws/src/custom_action_interface`

1. package.xml 수정하기
   - 파일 수정하기 위해 이동하기

```
$ cd ~/ros2_ws/src/custom_action_interface
$ gedit package.xml
```

▼ **package.xml 내용**

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd'
<package format="3">
  <name>custom_action_interface</name>
  <version>0.0.0</version>
  <description> ROS2 example for action interface </description>
  <maintainer email="jetson@todo.todo">jetson</maintainer>
  <license>TODO: License declaration</license>
  <buildtool_depend>ament_cmake</buildtool_depend>

  <buildtool_depend>rosidl_default_generators</buildtool_depend> ## 
  <exec_depend>builtin_interfaces</exec_depend> ## 실행시 사용되는 패키지
  <exec_depend>rosidl_default_runtime</exec_depend> ## 실행시 사용되는 

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <member_of_group>rosidl_interface_packages</member_of_group>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

2. CmakeLists.txt 수정하기 위해 이동

```
$ cd ~/ros2_ws/src/custom_action_interface
$ gedit CMakeLists.txt
```

▼ **CMakeLists.txt 내용**

```
cmake_minimum_required(VERSION 3.5)
project(custom_action_interface)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

```
##============================================================
##  Find and load build settings from external packages
##============================================================
find_package(ament_cmake REQUIRED)
find_package(builtin_interfaces REQUIRED) ## 추가된 부분
find_package(rosidl_default_generators REQUIRED)## 추가된 부분
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)
##============================================================



##============================================================
##  Declare ROS messages, Fibonacci action
##============================================================
set(action_files
 "action/Fibonacci.action" ## 정의한 action 파일 경로
)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${action_files}
  DEPENDENCIES builtin_interfaces
)
##============================================================



##============================================================
## Macro for ament package
##============================================================
ament_export_dependencies(rosidl_default_runtime)
ament_package()
##============================================================
```

▼ 방법2: 깃클론해서 패키지 빌드하기

```
$ cd ~/ros2_ws/src
$ git clone https://github.com/2seung0708/ros2_example.git
$ mv ./ros2_example/src/custom_action_interface ~/ros2_ws/src/
```

4. Build하기

- 경로: `~/ros2_ws`

```
$ source /opt/ros/foxy/setup.bash
$ cd ~/ros2_ws/
```

```
$ colcon build --symlink-install --packages-select custom_action_interface
```

- 확인하기

```
$ source ./install/setup.bash

$ ros2 interface show custom_action_interface/action/Fibonacci
##################### 출력 결과 ==================
# Goal
int32 order
---
# Result
int32[] sequence
---
# Feedback
int32[] partial_sequence
##################################################
```

<2. Action 프로그래밍>

Action 예제 작성

1. 패키지 생성

- 작성한 워크스페이스인 `ros2_ws/src` 디렉토리에 이동하신 다음 새로운 패키지를 생성합니다.

```
$ cd ~/ros2_ws/src/
$ ros2 pkg create fibonacci_action --build-type ament_python --dependencies r
```

- 새로운 패키지 이름은 `fibonacci_action` 으로 동명의 디렉토리에 패키지 기본 구성이 생성된 것을 확인 할 수 있을 겁니다.
- 추가로 `--dependencies` 인수를 통해 패키지 환경 설정 파일 `package.xml` 에 필요한 종속성 패키지인 `rclpy` , `custom_action_interface`가 추가됩니다.

Action Server Node 작성

- Action 기능: `Goal Response`, `Feedback`, `Result Response`를 구현
- Action Server Node의 파이썬 스크립트는 `` `~/ros2_ws/src/fibonacci_action/fibonacci_action /` `` 폴더에 `` `fibonacci_action_server.py` `` 라는 이름으로 소스 코드 파일을 저장하시면 됩니다.

```
$ cd ~/ros2_ws/src/fibonacci_action/fibonacci_action
$ gedit fibonacci_action_server.py
```

- **fibonacci_action_server.py** 코드

```
import rclpy # Python ROS2 프로그래밍을 위한 rclpy
from rclpy.node import Node# rclpy 의 Node 클래스
from rclpy.action import ActionServer, GoalResponse # ActionServer와 GoalR

from custom_action_interface.action import Fibonacci # 사전에 정의한 인터페이스

import time

class FibonacciActionServer(Node):# Node 클래스를 상속
    def __init__(self):
        super().__init__('fibonacci_action_server') # 부모 클래스(Node)의 생성

        ##==================== Action Server 정의 ==========================
        self.action_server = ActionServer(
            self, #  실행 노드
            Fibonacci, # 메시지 타입
            'fibonacci',# 액션 이름(client도 동일하게 받아야함)
            ##### 콜백 함수#####
                        # Goal Response가 오면, 우선 goal_callback을 실행시킨 뒤
                        # execute_callback으로 넘어가게 됩니다.
                        ####################
            self.execute_callback,
            goal_callback=self.goal_callback)
        ##==================================================================

        self.get_logger().info("=== Fibonacci Action Server Started ====")

    ##================ Goal Request 이후의 콜백함수 ===============
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')

        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]

        for i in range(1, goal_handle.request.order):
            feedback_msg.partial_sequence.append(
                feedback_msg.partial_sequence[i] + feedback_msg.partial_se
            )

            print(f"Feedback: {feedback_msg.partial_sequence}")
            goal_handle.publish_feedback(feedback_msg) # 피드백 publish
            time.sleep(1)
```

```
        goal_handle.succeed() # 액션 client에 현재 액션 상태(성공) 알림
        self.get_logger().warn("==== Succeed ====")

        result = Fibonacci.Result()# Result로 선언
        result.sequence = feedback_msg.partial_sequence # feedback에 있던 값

        return result # 결과값을 반환
    ##============================================================

    ##=============== Goal Request시  콜백함수 ===============
    def goal_callback(self, goal_request):
        """Accept or reject a client request to begin an action."""
        self.get_logger().info('Received goal request')

        return GoalResponse.ACCEPT
    ##============================================================


def main(args=None):
    rclpy.init(args=args) # 초기화
    node = FibonacciActionServer() # FibonacciActionServer를 node라는 이름으로
    try:
        rclpy.spin(node) # rclpy에게 이 Node를 반복해서 실행 (=spin) 하라고 전달
    except KeyboardInterrupt: # `Ctrl + c`가 동작했을 때
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()  # 노드 소멸
        rclpy.shutdown() # rclpy.shutdown 함수로 노드 종료


if __name__ == '__main__':
    main()
```

- class 내부

  ▼ `def __init__(self)`

```
class FibonacciActionServer(Node):# Node 클래스를 상속
    def __init__(self):
        super().__init__('fibonacci_action_server') # 부모 클래스(Node)의

        ##==================== Action Server 정의 ======================
        self.action_server = ActionServer(
            self, #  실행 노드
            Fibonacci, # 메시지 타입
            'fibonacci',# 액션 이름(client도 동일하게 받아야함)
```

```
                    ##----- 콜백 함수 -----
                    # Goal Response가 오면, 우선 goal_callback을 실행시
                    # execute_callback으로 넘어가게 됩니다.
                    ##---------------------
            self.execute_callback,
            goal_callback=self.goal_callback)
        ##=============================================================

        self.get_logger().info("=== Fibonacci Action Server Started ===
```

▼ `def execute_callback(self, goal_handle)`

```
    ##==================== goal_callback 이후의 콜백함수 ================
    ##  =>  Feedback과 Result를 처리
    ## goal_handle: rclpy.action 모듈의 ServerGoalHandle 클래스로 생성된 액션
    ##              execute, succeed, canceled 등 액션 상태에 따른 관련 함수
    ##              & 피드백을 publish가능
    ##=============================================================
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')# 터미널 창에 출력하며 로

                # Feedback action을 준비
        feedback_msg = Fibonacci.Feedback() # Feedback으로 선언
        feedback_msg.partial_sequence = [0, 1]

                # Request 숫자만큼의 피보나치 수열을 계산
        for i in range(1, goal_handle.request.order):

                    # 피보나치 로직
            feedback_msg.partial_sequence.append( # 연산 결과를 피드백에 추
                feedback_msg.partial_sequence[i] + feedback_msg.partial
            )

                    # feedback publish가 이루어지는 부분!!
            print(f"Feedback: {feedback_msg.partial_sequence}")
            goal_handle.publish_feedback(feedback_msg) # 피드백 publish
            time.sleep(1)

        goal_handle.succeed() # 액션 client에 현재 액션 상태(성공) 알림
        self.get_logger().warn("==== Succeed ====")

                # 모든 계산을 마치고, result를 되돌려주는 부분
        result = Fibonacci.Result()# Result로 선언
        result.sequence = feedback_msg.partial_sequence # feedback에 있던
```

```
        return result # 결과값을 반환
    ##=====================================================================
```

▼ `def goal_callback(self, goal_request)`

```
    ##==================== goal_Request시 사용되는 콜백함수 ===============
    ##=====================================================================
        # Goal Request 시 가장 처음 진입하게 되는 callback입니다.
    def goal_callback(self, goal_request):
        """Accept or reject a client request to begin an action."""
        self.get_logger().info('Received goal request')

                # 도저히 불가능한 Request가 왔다면, 여기에서 판단하여 REJECT합니
                # GoalResponse의 ACCEPT=1, REJECT=2로 상태는 숫자로 바꿔 사용
                # 아래 ACCEPT => REJECT로 바꾼 뒤, 다시 실행시켜보세요!!
        return GoalResponse.ACCEPT
    ##=====================================================================
```

- main 부분

```
def main(args=None):
    rclpy.init(args=args) # 초기화
    node = FibonacciActionServer() # FibonacciActionServer를 node라는 이름으로
    try:
        rclpy.spin(node) # rclpy에게 이 Node를 반복해서 실행 (=spin) 하라고 전달
    except KeyboardInterrupt: # `Ctrl + c`가 동작했을 때
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()  # 노드 소멸
        rclpy.shutdown() # rclpy.shutdown 함수로 노드 종료


if __name__ == '__main__':
    main()
```

▼ 전체 코드 (with 주석)

```
import rclpy # Python ROS2 프로그래밍을 위한 rclpy
from rclpy.node import Node# rclpy 의 Node 클래스
from rclpy.action import ActionServer, GoalResponse # ActionServer와 GoalRe

from custom_action_interface.action import Fibonacci # 사전에 정의한 인터페이스

class FibonacciActionServer(Node):# Node 클래스를 상속
    def __init__(self):
        super().__init__('fibonacci_action_server') # 부모 클래스(Node)의 생성
```

```
        qos_profile = QoSProfile(depth=10) # 통신상태가 원활하지 못할 경우 퍼블리

        ##==================== Action Server 정의 ========================
        self._action_server = ActionServer(
            self, #  실행 노드
            Fibonacci, # 메시지 타입
            'fibonacci',# 액션 이름(client도 동일하게 받아야함)

                        ##### 콜백 함수#####
                        # Goal Response가 오면, 우선 goal_callback을 실행시킨
                        # execute_callback으로 넘어가게 됩니다.
                        ####################
            self.execute_callback,
            goal_callback=self.goal_callback)
        ##================================================================

        self.get_logger().info("=== Fibonacci Action Server Started ====")


    ##==================== goal_callback 이후의 콜백함수 ====================
    ##  =>  Feedback과 Result를 처리
    ## goal_handle: rclpy.action 모듈의 ServerGoalHandle 클래스로 생성된 액션 상
    ##              execute, succeed, canceled 등 액션 상태에 따른 관련 함수 호출
    ##              & 피드백을 publish가능
    ##================================================================
    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')# 터미널 창에 출력하며 로그

                # Feedback action을 준비
        feedback_msg = Fibonacci.Feedback() # Feedback으로 선언
        feedback_msg.partial_sequence = [0, 1]

                # Request 숫자만큼의 피보나치 수열을 계산
        for i in range(1, goal_handle.request.order):

                        # 피보나치 로직
            feedback_msg.partial_sequence.append( # 연산 결과를 피드백에 추가
                feedback_msg.partial_sequence[i] + feedback_msg.partial_se
            )

                        # feedback publish가 이루어지는 부분!!
            print(f"Feedback: {feedback_msg.partial_sequence}")
            goal_handle.publish_feedback(feedback_msg) # 피드백 publish
            time.sleep(1)

        goal_handle.succeed() # 액션 client에 현재 액션 상태(성공) 알림
        self.get_logger().warn("==== Succeed ====")
```

```
                # 모든 계산을 마치고, result를 되돌려주는 부분
        result = Fibonacci.Result()# Result로 선언
        result.sequence = feedback_msg.partial_sequence # feedback에 있던 값

        return result # 결과값을 반환
    ##=========================================================================



    ##==================== goal_Request시 사용되는 콜백함수 ==================
    ##  =>  Feedback과 Result를 처리
    ## goal_handle: rclpy.action 모듈의 ServerGoalHandle 클래스로 생성된 액션 상
    ##              execute, succeed, canceled 등 액션 상태에 따른 관련 함수 호출
    ##              & 피드백을 publish가능
    ##=========================================================================
        # Goal Request 시 가장 처음 진입하게 되는 callback입니다.
    def goal_callback(self, goal_request):
        """Accept or reject a client request to begin an action."""
        self.get_logger().info('Received goal request')

                # 도저히 불가능한 Request가 왔다면, 여기에서 판단하여 REJECT합니다.
                # 아래 ACCEPT => REJECT로 바꾼 뒤, 다시 실행시켜보세요!!
        return GoalResponse.ACCEPT
    ##=========================================================================


def main(args=None):
    rclpy.init(args=args) # 초기화
    node = FibonacciActionServer() # FibonacciActionServer를 node라는 이름으로
    try:
        rclpy.spin(node) # rclpy에게 이 Node를 반복해서 실행 (=spin) 하라고 전달
    except KeyboardInterrupt: # `Ctrl + c`가 동작했을 때
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()  # 노드 소멸
        rclpy.shutdown() # rclpy.shutdown 함수로 노드 종료


if __name__ == '__main__':
    main()
```

Action Client Node작성

- Action Client Node의 파이썬 스크립트는 `~/ros2_ws/src/fibonacci_action/fibonacci_action /`폴더에 `fibonacci_action_client.py.py`라는 이름으로 소스 코드 파일을 저장하시면 됩니다.

```
$ cd ~/ros2_ws/src/fibonacci_action/fibonacci_action
```

```
$ gedit fibonacci_action_client.py
```

- **fibonacci_action_client.py** 코드

```
import rclpy # Python ROS2 프로그래밍을 위한 rclpy
from rclpy.node import Node# rclpy 의 Node 클래스
from rclpy.action import ActionClient, GoalResponse # ActionServer와 GoalR

from custom_action_interface.action import Fibonacci # 사전에 정의한 인터페이스

class FibonacciActionClient(Node):# Node 클래스를 상속
    def __init__(self):
        super().__init__('fibonacci_action_client') # 부모 클래스(Node)의 생성

        ##==================== Action Server 정의 ========================
        self.action_client= ActionClient(
            self, #  실행 노드
            Fibonacci, # 메시지 타입
            'fibonacci')# 액션 이름(action server에서 정한 이름과 동일해야함!)
        ##================================================================

        self.get_logger().info("=== Fibonacci Action Client Started ====")

    ##================ send Gaol ===============
    def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order

        # 10초간 server를 기다리다가 응답이 없으면 에러를 출력
        if self.action_client.wait_for_server(10) is False:
          self.get_logger().error("Server Not exists")

        # goal request가 제대로 보내졌는지 알기 위해 future가 사용됩니다.
        # 더불어, feedback_callback을 묶어 feedback 발생 시 해당 함수로 이동합니다
        self._send_goal_future = self.action_client.send_goal_async(
            goal_msg, feedback_callback=self.feedback_callback
        )


        # server가 존재한다면, Goal Request의 성공 유무,
        # 최종 Result에 대한 callback도 필요
        self._send_goal_future.add_done_callback(self.goal_response_callba
    ##============================================================

    ##================ feedback을 받아오는 함수 ===============
    def feedback_callback(self, feedback_msg):#send_goal에 사용됨
        feedback = feedback_msg.feedback
        print(f"Received feedback: {feedback.partial_sequence}") # 출력
```

```
        ##=========================================================


        ##=============== Goal Request에 대한 응답으로 실행되는 callback ==========
        def goal_response_callback(self, future):
            goal_handle = future.result()

            # Goal type에 따라 성공 유무를 판단합니다.
            if not goal_handle.accepted:
                self.get_logger().info("Goal rejected")
                return

            self.get_logger().info("Goal accepted")

            # 최종 Result 데이터를 다룰 callback을 연동합니다.
            self._get_result_future = goal_handle.get_result_async()
            self._get_result_future.add_done_callback(self.get_result_callback
        ##=========================================================

        ##===============  Result callback 함수 ===============
        def get_result_callback(self, future):
            result = future.result().result
            self.get_logger().warn(f"Action Done !! Result: {result.sequence}"
            rclpy.shutdown()
        ##=========================================================

    def main(args=None):
        rclpy.init(args=args)

        fibonacci_action_client = FibonacciActionClient()

        # Client Node 생성 이후 직접 send_goal을 해줍니다. (Service와 유사)
            # Goal Request에 대한 future를 반환받음
        future = fibonacci_action_client.send_goal(5)

        rclpy.spin(fibonacci_action_client)


    if __name__ == '__main__':
        main()
```

- class 내부

  ▼ `def __init__(self)`

  ```
  class FibonacciActionClient(Node):# Node 클래스를 상속
      def __init__(self):
          super().__init__('fibonacci_action_client') # 부모 클래스(Node)의
  ```

```
        ##==================== Action Server 정의 ======================
        self.action_client= ActionClient(
            self, #  실행 노드
            Fibonacci, # 메시지 타입
            'fibonacci')# 액션 이름(action server에서 정한 이름과 동일해야함!)
        ##============================================================

        self.get_logger().info("=== Fibonacci Action Client Started ===
```

▼ `def send_goal(self, order)`

```
    ##=============== send Gaol ===============
    def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order

        # 10초간 server를 기다리다가 응답이 없으면 에러를 출력
        if self.action_client.wait_for_server(10) is False:
          self.get_logger().error("Server Not exists")

        # goal request가 제대로 보내졌는지 알기 위해 future가 사용됩니다.
        # 더불어, feedback_callback을 묶어 feedback 발생 시 해당 함수로 이동합
        self._send_goal_future = self.action_client.send_goal_async(
            goal_msg, feedback_callback=self.feedback_callback
        )


        # server가 존재한다면, Goal Request의 성공 유무,
        # 최종 Result에 대한 callback도 필요
        self._send_goal_future.add_done_callback(self.goal_response_cal
    ##============================================================
```

▼ `def feedback_callback (self, feedback_msg)`

```
    ##=============== feedback을 받아오는 함수 ===============
    def feedback_callback(self, feedback_msg):#send_goal에 사용됨
        feedback = feedback_msg.feedback
        print(f"Received feedback: {feedback.partial_sequence}") # 출력
    ##============================================================
```

▼ `def goal_response_callback(self, future)`

```
    ##=============== Goal Request에 대한 응답으로 실행되는 callback =======
    def goal_response_callback(self, future):
        goal_handle = future.result()

        # Goal type에 따라 성공 유무를 판단합니다.
        if not goal_handle.accepted:
```

```
            self.get_logger().info("Goal rejected")
            return

        self.get_logger().info("Goal accepted")

        # 최종 Result 데이터를 다룰 callback을 연동합니다.
        self._get_result_future = goal_handle.get_result_async()
        self._get_result_future.add_done_callback(self.get_result_callb
    ##=============================================================
```

▼ `def get_result_callback(self, future)`

```
    ##===============  Result callback 함수 ===============
    def get_result_callback(self, future):
        result = future.result().result
        self.get_logger().warn(f"Action Done !! Result: {result.sequenc
        rclpy.shutdown()
    ##=============================================================
```

📌 함수에 대한 실행 시점

- `send_goal` : goal send 시점에 `feedback_callback`이 묶이며 send_goal 이 완료되는 시점에 `goal_response_callback` 으로 이동합니다.

```
self._send_goal_future = self.action_client.send_goal_async(
    goal_msg, feedback_callback=self.feedback_callback
)

self._send_goal_future.add_done_callback(self.goal_response_callback)
```

- `goal_response_callback` : `get_result_async` 이 완료되는 시점에 `get_result_callback` 으로 이동합니다.

```
self._get_result_future = goal_handle.get_result_async()
self._get_result_future.add_done_callback(self.get_result_callback)
```

- `feedback_callback`: 지속적으로 feedback을 출력합니다.
- `get_result_callback` : 최종 마지막에 실행되는 함수로 Result를 출력합니다.

- main 부분

```
def main(args=None):
    rclpy.init(args=args)

    fibonacci_action_client = FibonacciActionClient()

    # Client Node 생성 이후 직접 send_goal을 해줍니다. (Service와 유사)
```

```
        # Goal Request에 대한 future를 반환받음
    future = fibonacci_action_client.send_goal(5)

    rclpy.spin(fibonacci_action_client)


if __name__ == '__main__':
    main()
```

Add an entry point

- `ros2 run` 커맨드를 통해 작성한 service node 실행시키기 위해서는 `setup.py` 속의 `entry_points` 구역에 아래의 내용을 추가해야합니다.
- 경로로 이동 및 수정
-

```
$ cd ~/ros2_ws/src/fibonacci_action
$ gedit setup.py
```

```
entry_points={
    'console_scripts': [
        'fibonacci_action_server= fibonacci_action.fibonacci_action_server:ma
        'fibonacci_action_client= fibonacci_action.fibonacci_action_client:ma
    ],
},
```

▼ 방법2: 깃클론해서 패키지 빌드하기

- 앞서 git clone해온 예제 파일들 중 fibonacci에 해당하는 폴더 `~/ros2_ws/src/` 디렉토리로 옮기기

```
$ cd ~/ros2_ws/src
$ mv ./ros2_example/src/fibonacci_action ~/ros2_ws/src/
```

Build and run

- 이제 패키지를 build하고 실행해보도록 하겠습니다
- 실행 과정(`ros2 run` 실행 전에 수행해야 하는 코드)
  1. 먼저 실행을 위한 경로로 이동하여 ROS2 실행 환경을 실행합니다.

```
$ cd ~/ros2_ws
$ source /opt/ros/foxy/setup.bash
```

2. 그 다음에 빌드를 수행합니다.

```
$ colcon build --symlink-install --packages-select fibonacci_action
Starting >>> fibonacci_action
Finished <<< fibonacci_action [0.94s]

Summary: 1 package finished [1.12s]
```

3. 마지막으로 로컬에 위치한 패키지의 환경 변수를 설정하기 위해서 setup file을 source 합니다!

```
$ source install/local_setup.bash
```

> 💡 install 디렉토리에 위치한 `local_setup` 과 `setup` 은 뭐가 다른 걸까요?
>
> - `local_setup` 은 내가 설치한 패키지의 환경 변수를 source 하기 위한 파일!
> - `setup` 은 `/opt/ros/foxy` 와 같이 글로벌하게 사용되는 환경 변수도 source 합니다.
>   즉,
>   `source /opt/ros/foxy/setup.bash & source install/setup.bash` 과 동일합니다.

실행

```
# 터미널1

$ ros2 run fibonacci_action fibonacci_action_server
[INFO] [1683486665.920066930] [fibonacci_action_server]: === Fibonacci Action
[INFO] [1683486673.420139016] [fibonacci_action_server]: Received goal reques
[INFO] [1683486673.421320117] [fibonacci_action_server]: Executing goal...
Feedback: array('i', [0, 1, 1])
Feedback: array('i', [0, 1, 1, 2])
Feedback: array('i', [0, 1, 1, 2, 3])
Feedback: array('i', [0, 1, 1, 2, 3, 5])
[WARN] [1683486677.428286037] [fibonacci_action_server]: ==== Succeed ====
```

```
# 터미널 2

$ ros2 run fibonacci_action fibonacci_action_client
[INFO] [1683486673.418898780] [fibonacci_action_client]: === Fibonacci Action
future?:<rclpy.task.Future object at 0x7ff002c90b20>
[INFO] [1683486673.421179634] [fibonacci_action_client]: Goal accepted
Received feedback: array('i', [0, 1, 1])
```

```
Received feedback: array('i', [0, 1, 1, 2])
Received feedback: array('i', [0, 1, 1, 2, 3])
Received feedback: array('i', [0, 1, 1, 2, 3, 5])
[WARN] [1683486677.431033816] [fibonacci_action_client]: Action Done !! Resul
```

> 이번 장에서는 Action을 이용하여 'Fibonacci'를 작성해보았습니다. 다음 장에서는 단발적인 통신에 활용되는 동기식 양방향 메세지 통신 `Service` 에 대해 다뤄볼 예정입니다.

Writing an action server and client (Python) — ROS 2 Documentation: Foxy documentation

2 https://docs.ros.org/en/foxy/Tutorials/Intermediate/Writing-an-Action-Server-Client/Py.html