Week8

6. 패키지 설계

실행

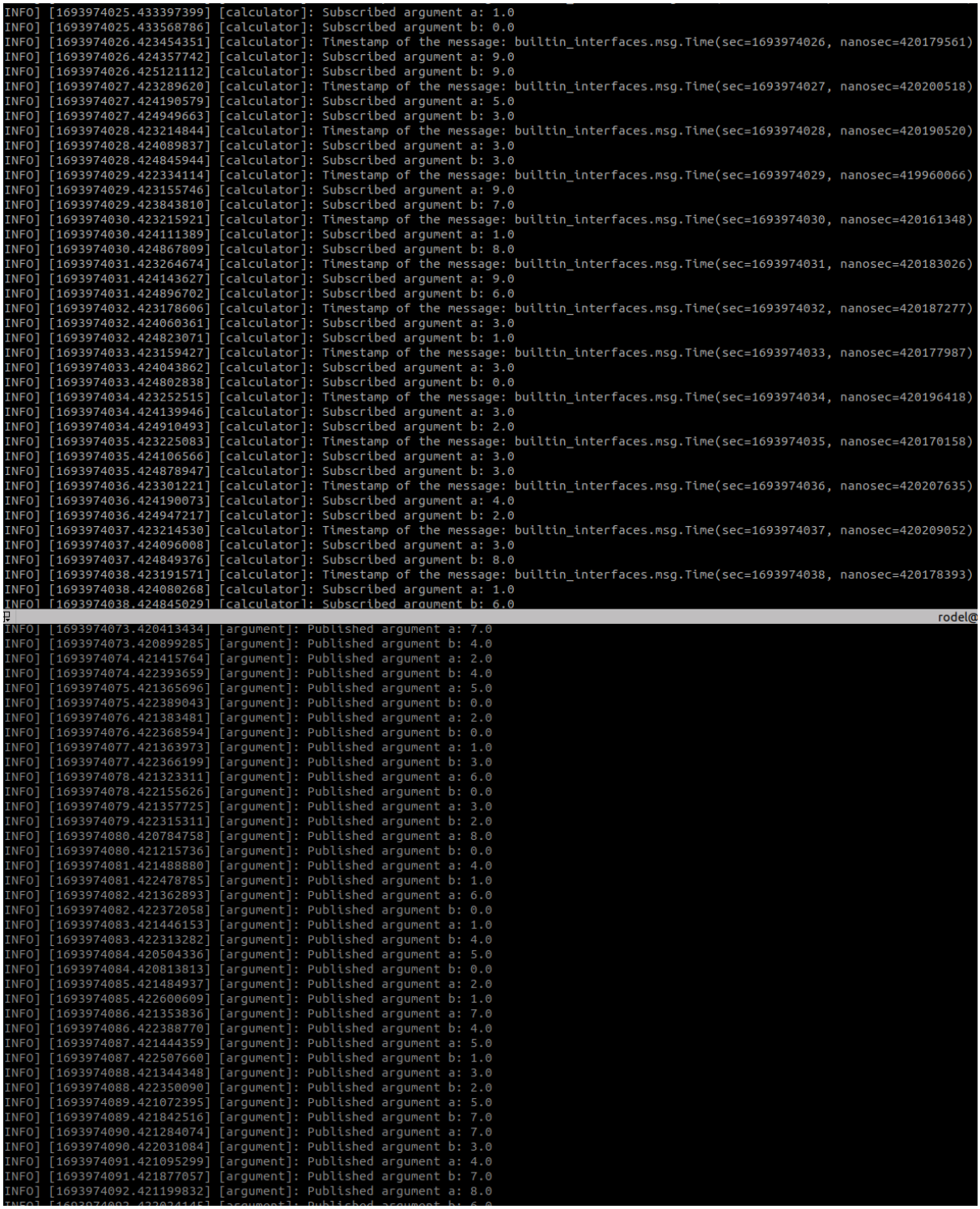
calculator 노드를 실행한다

본 노드는 토픽 subscriber, 서비스 server, 액션 server 역할을 수행한다

```
$ ~/robot_ws$ ros2 run topic_service_action_rclpy_example calculator-
```

토픽 퍼블리셔

argument 노드 실행

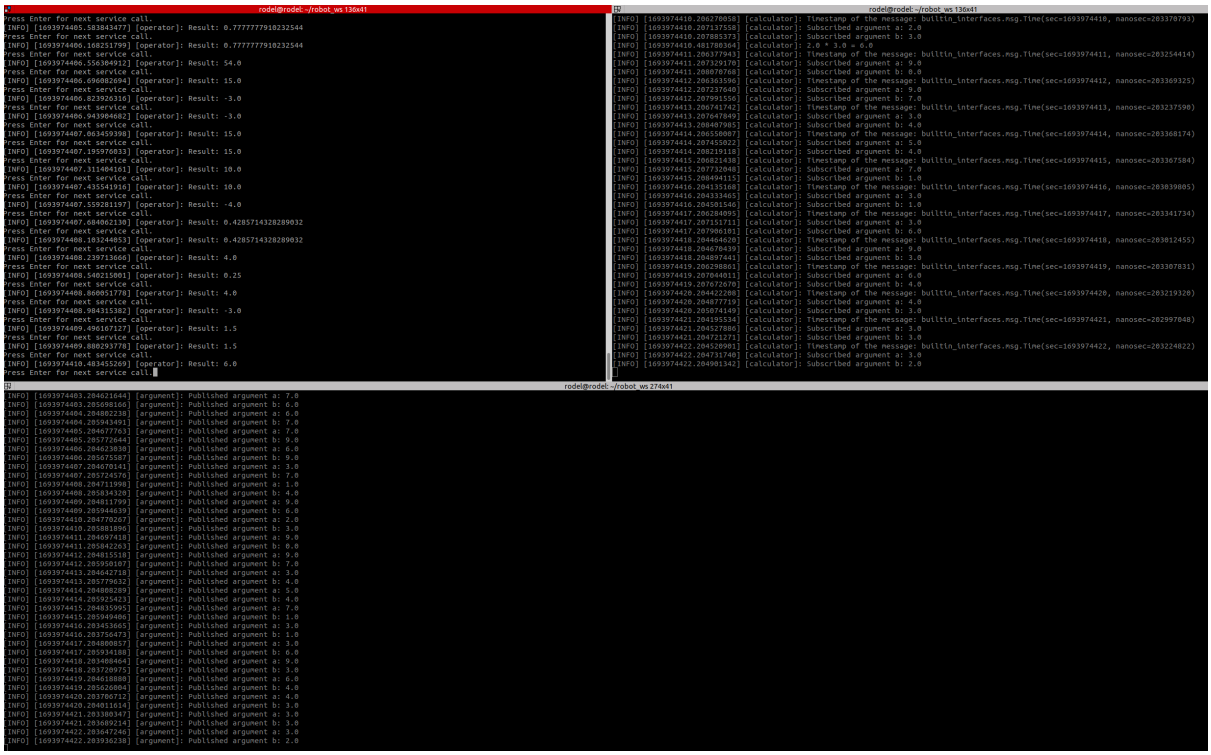


서비스 클라이언트 실행

operator 노드 : calculator 노드에게 랜덤으로 선택한 연산자(+,-,*,/)를 서비스 요청값으로 보내고 연산된 결괏값을 받아 터미널 창에 표시한다

```
$ ~/robot_ws$ ros2 run topic_service_action_rclpy_example operator
```

결과



액션 클라이언트 실행

checker 노드 : 연산 결괏값의 누적 한계치를 액션 목푯값으로 전달한다.

calculator 노드는 액션 목푯값을 전달받은 이후부터 연산 결괏값을 누적하고 액션 피드백으로 연산식을 보낸다. 누적된 연산 결괏값이 전달받은 액션 목푯값보다 커지면 액션 결괏값으로 누적된 연산 결괏값과 진행된 연산식 모두를 보낸다.

```
$ ~/robot_ws$ ros2 run topic_service_action_rclpy_example checker
```

런치 파일 실행

argument 노드와 calculator 노드를 한 번에 실행시키려면 런치파일(arithmetic.launch.py)을 실행한다.

```
$ ros2 launch topic_service_action_rclpy_example arithmetic.launch.py
```

7. 토픽 프로그래밍(파이썬)

토픽

토픽은 비동기식 단방향 메시지 송수신 방식으로 Publisher 와 Subscriber 간의 통신이다.

1:1이 기본이지만 1:N과 N:N 도 가능하다

Publisher 인 argument 노드의 소스코드는 다음과 같다

```
# Copyright 2021 OROCA
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import random

from msg_srv_action_interface_example.msg import ArithmeticArgument
from rcl_interfaces.msg import SetParametersResult
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rclpy.qos import QoSDurabilityPolicy
from rclpy.qos import QoSHistoryPolicy
from rclpy.qos import QoSProfile
from rclpy.qos import QoSReliabilityPolicy


class Argument(Node):

    def __init__(self):
        super().__init__('argument')
        self.declare_parameter('qos_depth', 10)
        qos_depth = self.get_parameter('qos_depth').value
        self.declare_parameter('min_random_num', 0)
        self.min_random_num = self.get_parameter('min_random_num').value
        self.declare_parameter('max_random_num', 9)
        self.max_random_num = self.get_parameter('max_random_num').value
        self.add_on_set_parameters_callback(self.update_parameter)

        QOS_RKL10V = QoSProfile(
            reliability=QoSReliabilityPolicy.RELIABLE,
            history=QoSHistoryPolicy.KEEP_LAST,
            depth=qos_depth,
            durability=QoSDurabilityPolicy.VOLATILE)

        self.arithmetic_argument_publisher = self.create_publisher(
            ArithmeticArgument,
            'arithmetic_argument',
            QOS_RKL10V)

        self.timer = self.create_timer(1.0, self.publish_random_arithmetic_arguments)

    def publish_random_arithmetic_arguments(self):
```

```
        msg = ArithmeticArgument()
        msg.stamp = self.get_clock().now().to_msg()
        msg.argument_a = float(random.randint(self.min_random_num, self.max_random_num))
        msg.argument_b = float(random.randint(self.min_random_num, self.max_random_num))
        self.arithmetic_argument_publisher.publish(msg)
        self.get_logger().info('Published argument a: {0}'.format(msg.argument_a))
        self.get_logger().info('Published argument b: {0}'.format(msg.argument_b))

    def update_parameter(self, params):
        for param in params:
            if param.name == 'min_random_num' and param.type_ == Parameter.Type.INTEGER:
                self.min_random_num = param.value
            elif param.name == 'max_random_num' and param.type_ == Parameter.Type.INTEGER:
                self.max_random_num = param.value
        return SetParametersResult(successful=True)


def main(args=None):
    rclpy.init(args=args)
    try:
        argument = Argument()
        try:
            rclpy.spin(argument)
        except KeyboardInterrupt:
            argument.get_logger().info('Keyboard Interrupt (SIGINT)')
        finally:
            argument.destroy_node()
    finally:
        rclpy.shutdown()


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

토픽 퍼블리셔

중요 포인트는 arithmetic_argument_publisher 선언으로 create_publisher 함수로 토픽타입, 토픽이름, Qos를 설정하며 선언한다.

그리고 create_timer 함수로 1초마다 publish_random_arithmetic_arguments 함수를 실행시킨다.

```
self.arithmetic_argument_publisher = self.create_publisher(
            ArithmeticArgument,
            'arithmetic_argument',
            QOS_RKL10V)

        self.timer = self.create_timer(1.0, self.publish_random_arithmetic_arguments)
```

publish_random_arithmetic_arguments 함수를 보자

우선 msg 변수를 msg 인터페이스의 ArithmeticArgument() 클래스로 생성한다. msg.stamp 변수는 **get_clock().now().to_msg()** 함수를 통해 **토픽이 생성된 시간을 기록**하고, 랜덤 함수를

통해 0~9까지의 정수를 float 타입으로 변환하여 msg.argument_a와 msg.argument_b에 저장한다.

get_logger().info()는 디버깅에 사용하는 함수다

```
def publish_random_arithmetic_arguments(self):
        msg = ArithmeticArgument()
        msg.stamp = self.get_clock().now().to_msg()
        msg.argument_a = float(random.randint(self.min_random_num, self.max_random_num))
        msg.argument_b = float(random.randint(self.min_random_num, self.max_random_num))
        self.arithmetic_argument_publisher.publish(msg)
        self.get_logger().info('Published argument a: {0}'.format(msg.argument_a))
        self.get_logger().info('Published argument b: {0}'.format(msg.argument_b))
```

토픽 서브스크라이버

Subscriber 역할인 calculator 노드는 subscriber, 서비스 서버, 액션 서버를 모두 포함한다.

우선 Calculator 클래스로, rclpy.node 모듈의 Node 클래스를 상속해 생성자(**__init__**)를 통해 노드 이름을 calculator로 초기화했다.

```
class Calculator(Node):

    def __init__(self):
        super().__init__('calculator')
        self.argument_a = 0.0
        self.argument_b = 0.0
        self.argument_operator = 0
        self.argument_result = 0.0
        self.argument_formula = ''
        self.operator = ['+', '-', '*', '/']
        self.callback_group = ReentrantCallbackGroup()
```

중요 포인트는 Subscriber 선언이다.

토픽 타입과 이름은 Publisher와 동일하게 각각 ArithmeticArgument 와 arithmetic_argument로 선언함

Publisher와 다른점은 get_arithmetic_argument 콜백 함수로 메시지를 subscribe할 때마다 실행하는 함수를 지정한 것이다.

또한 **callback_group을 지정함으로써 콜백함수를 병렬로 실행**할 수 있다. 만약 지정하지 않으면 한 번에 하나의 콜백 함수만 실행하도록 한다.

```
self.arithmetic_argument_subscriber = self.create_subscription(
            ArithmeticArgument,
            'arithmetic_argument',
            self.get_arithmetic_argument,
            QOS_RKL10V,
            callback_group=self.callback_group)
```

노드 실행 코드

```
$ ros2 run topic_service_action_rclpy_example calculator # 퍼블리셔
$ ros2 run topic_service_action_rclpy_example argument   # 서브스크라이버
```

ros2 run으로 이 노드들을 실행하려면 setup.py의 **entry_points**에 추가해줘야 한다.

```
entry_points={
        'console_scripts': [
            'argument = topic_service_action_rclpy_example.arithmetic.argument:main',
            'operator = topic_service_action_rclpy_example.arithmetic.operator:main',
            'calculator = topic_service_action_rclpy_example.calculator.main:main',
            'checker = topic_service_action_rclpy_example.checker.main:main',
        ],
},
```

8. 서비스 프로그래밍(파이썬)

서비스(Service)는 동기식 양방향 메시지 송수신 방식으로 서비스의 요청을 하는 쪽을 Client, 요청 받은 서비스를 수행 후 응답하는 쪽을 Server라고 한다. 결국 서비스는 특정 요청을 하는 클라이언트 단과 요청받은 일을 수행한 후 결괏값을 전달하는 서버 단과의 통신이다

서비스 서버 코드

Calculator 노드는 서비스 Server역할도 한다. Subscribe해서 저장하고 있는 멤버 변수 a,b와 operator 노드로부터 ‘요청값’으로 받은 연산자를 이용해 연산하고 결괏값을 ‘응답값’으로 보낸다

arithmetic_service_server는 Node 클래스의 create_service함수로 서비스 서버로 선언된다.

서비스 타입, 이름을 지정하고 서비스 클라이언트로부터 요청받으면 실행하는 **콜백함수는 get_arithmetic_operator로 지정**하고 병렬 콜백 함수 실행을 위해 callback_group설정했다.

```
self.arithmetic_service_server = self.create_service(
            ArithmeticOperator,
            'arithmetic_operator',
            self.get_arithmetic_operator,
            callback_group=self.callback_group)
```

콜백함수는 ArithmeticOperator 클래스로 생성된 인터페이스로 서비스 요청(request)과 응답(response)로 구분된다.

서비스 요청 시 요청값으로 받은 연산자(PLUS, MINUS, MULTIPLY, DIVISION)와 이전에 토픽 subscriber가 토픽값으로 전달받아 저장해둔 a,bㄹㄹ 연산한 결과값을 응답값으로 반환함

```
def get_arithmetic_operator(self, request, response):
        self.argument_operator = request.arithmetic_operator
        self.argument_result = self.calculate_given_formula(
            self.argument_a,
            self.argument_b,
            self.argument_operator)
        response.arithmetic_result = self.argument_result
        self.argument_formula = '{0} {1} {2} = {3}'.format(
                self.argument_a,
                self.operator[self.argument_operator-1],
                self.argument_b,
                self.argument_result)
        self.get_logger().info(self.argument_formula)
        return response
```

```
def calculate_given_formula(self, a, b, operator):
        if operator == ArithmeticOperator.Request.PLUS:
            self.argument_result = a + b
        elif operator == ArithmeticOperator.Request.MINUS:
            self.argument_result = a - b
        elif operator == ArithmeticOperator.Request.MULTIPLY:
            self.argument_result = a * b
        elif operator == ArithmeticOperator.Request.DIVISION:
            try:
                self.argument_result = a / b
            except ZeroDivisionError:
                self.get_logger().error('ZeroDivisionError!')
                self.argument_result = 0.0
                return self.argument_result
        else:
            self.get_logger().error(
                'Please make sure arithmetic operator(plus, minus, multiply, division).')
            self.argument_result = 0.0
        return self.argument_result
```

서비스 클라이언트 코드

Operator 노드는 서비스 클라이언트 역할을 한다.

서비스 타입과 이름은 서비스 클라이언트와 동일하게 선언한다.

wait_for_service 함수는 서비스 요청을 할 수 있는 상태인지 알아보기 위해 **서비스 서버가 실행되어 있는지 확인하는 함수**로 0.1초 간격으로 서비스 서버가 실행되어 있는지 확인한다.

```
class Operator(Node):

    def __init__(self):
        super().__init__('operator')
```

```
        self.arithmetic_service_client = self.create_client(
            ArithmeticOperator,
            'arithmetic_operator')

        while not self.arithmetic_service_client.wait_for_service(timeout_sec=0.1):
            self.get_logger().warning('The arithmetic_operator service not available.')
```

서비스 클라이언트의 목적은 서버에게 연산에 필요한 연산자를 보내는 것이다.

send_request 함수가 실질적인 클라이언트 실행코드로 서버에게 요청값을 보내고 이에 해당하는 응답값을 받는다.

```
def send_request(self):
        service_request = ArithmeticOperator.Request()
        service_request.arithmetic_operator = random.randint(1, 4)
        futures = self.arithmetic_service_client.call_async(service_request)
        return futures
```

main 함수를 보자

서비스 클라이언트는 토픽 퍼블리셔와 같은 지속적 수행을 하지 않고 한번만 수행하도록 했다.

서비스 요청 전달 이후 다시 요청하려면 Operator 노드를 실행시킨 터미널 창에서 엔터키를 누르면된다.

```
def main(args=None):
    rclpy.init(args=args)
    operator = Operator()
    future = operator.send_request()
    user_trigger = True
    try:
        while rclpy.ok():
            if user_trigger is True:
                rclpy.spin_once(operator)
                if future.done():
                    try:
                        service_response = future.result()
                    except Exception as e:  # noqa: B902
                        operator.get_logger().warn('Service call failed: {}'.format(str(e)))
                    else:
                        operator.get_logger().info(
                            'Result: {}'.format(service_response.arithmetic_result))
                        user_trigger = False
            else:
                input('Press Enter for next service call.')
                future = operator.send_request()
                user_trigger = True

    except KeyboardInterrupt:
        operator.get_logger().info('Keyboard Interrupt (SIGINT)')

    operator.destroy_node()
    rclpy.shutdown()
```

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

9. 액션 프로그래밍(파이썬)

액션 서버

액션은 비동기식+동기식 양방향 메시지 송수신방식으로 액션 목표(Goal)을 지정하는 클라이언트(Client)와 목표를 받아 특정 태스크를 수행하며 중간 결괏값을 전송하는 피드백(Feedback) 그리고 최종 결괏값에 해당하는 액션 결과(result)를 전송하는 액션 서버(Server)간의 통신이다

Calculator 노드는 액션 서버(Server)역할을 한다.

```
self.arithmetic_action_server = ActionServer(
            self,
            ArithmeticChecker,
            'arithmetic_checker',
            self.execute_checker,
            callback_group=self.callback_group)
```

실제 액션 목표를 받은 후 실행되는 콜백 함수는 **execute_checker** 함수다

```
def execute_checker(self, goal_handle):
        self.get_logger().info('Execute arithmetic_checker action!')
        feedback_msg = ArithmeticChecker.Feedback()
        feedback_msg.formula = []
        total_sum = 0.0
        goal_sum = goal_handle.request.goal_sum
        while total_sum < goal_sum:
            total_sum += self.argument_result
            feedback_msg.formula.append(self.argument_formula)
            self.get_logger().info('Feedback: {0}'.format(feedback_msg.formula))
            goal_handle.publish_feedback(feedback_msg)
            time.sleep(1)
        goal_handle.succeed()
        result = ArithmeticChecker.Result()
        result.all_formula = feedback_msg.formula
        result.total_sum = total_sum
        return result
```

액션 클라이언트

Checker 노드는 액션 클라이언트 역할을 한다

액션의 타입과 이름은 액션 서버와 동일하게 ArithmeticChecker, arithmetic_checker로 선언한다.

클라이언트의 액션 목표는 주기적으로 퍼블리시하는 토픽과 달리 필요 시 비주기적으로 수행한다.

```
class Checker(Node):

    def __init__(self):
        super().__init__('checker')
        self.arithmetic_action_client = ActionClient(
          self,
          ArithmeticChecker,
          'arithmetic_checker')
```

send_goal_total_sum 함수는 액션 목표를 서버에게 전송하고, 피드백 및 결과값을 받기 위한 콜백 함수를 지정하는 함수다

```
def send_goal_total_sum(self, goal_sum):
        wait_count = 1
        while not self.arithmetic_action_client.wait_for_server(timeout_sec=0.1):
            if wait_count > 3:
                self.get_logger().warning('Arithmetic action server is not available.')
                return False
            wait_count += 1
        goal_msg = ArithmeticChecker.Goal()
        goal_msg.goal_sum = (float)(goal_sum)
        self.send_goal_future = self.arithmetic_action_client.send_goal_async(
            goal_msg,
            feedback_callback=self.get_arithmetic_action_feedback)
        self.send_goal_future.add_done_callback(self.get_arithmetic_action_goal)
        return True
```

액션 처리 과정을 정리하자면

1. 액션 클라이언트 선언
2. 액션 목푯값 전달 함수 선언
3. 액션 피드백값 콜백 함수 선언
4. 액션 상태 콜백 함수 선언
5. 액션 결괏값 콜백 함수 선언

액션 피드백값 콜백 함수는 피드백을 서버로부터 전달받으면 호출되는데, 피드백인 feedback_msg.feedback.formula 값을 저장한다.

```
def get_arithmetic_action_feedback(self, feedback_msg):
        action_feedback = feedback_msg.feedback.formula
        self.get_logger().info('Action feedback: {0}'.format(action_feedback))
```

액션 상태 콜백함수는 비동기 future task 로 선언된 send_goal_future의 add_done_callback 함수를 통해 콜백 함수로 선언된 함수다. **액션 목푯값**이 문제없이 전달되면 **액션 결과값 콜백 함수**를 선언하게 된다.

```
def get_arithmetic_action_result(self, future):
        action_status = future.result().status
        action_result = future.result().result
        if action_status == GoalStatus.STATUS_SUCCEEDED:
            self.get_logger().info('Action succeeded!')
            self.get_logger().info(
                'Action result(all formula): {0}'.format(action_result.all_formula))
            self.get_logger().info(
                'Action result(total sum): {0}'.format(action_result.total_sum))
        else:
            self.get_logger().warning(
                'Action failed with status: {0}'.format(action_status))
```

10. 파라미터 프로그래밍(파이썬)

Ros2의 파라미터는 Ros1의 parameter server와 dynamic_reconfigure 패키지의 기능을 모두 가지고 있어 노드가 동작하는 동안 특정값에 대한 저장, 변경, 회수가 가능하다.

Ros2의 모든 노드는 파라미터 서버를 가지고 있어 파라미터 Client와 서비스 통신을 통해 파라미터에 접근할 수 있다.

서비스가 특정 태스크 수행을 위한 요청과 응답이라는 목적이라면 파라미터는 특정 매개변수를 노드 내부 또는 외부에서 쉽게 저장(set)하거나 변경할 수 있고, 회수(Get)하여 사용할 수 있게 하는 목적이다

파라미터 설정

argument 노드에서 파라미터를 선언하고 변경되는 함수를 보자

파라미터를 사용하려면 3가지 함수가 필요하다

1. **declare_parameter 함수** → 파라미터 고유 이름 지정, 초깃값 설정
2. **get_parameter 함수** →파라미터 값 불러온다. 주로 런치파일에서 호출된 *.yaml 확장자를 가지는 파라미터 파일에 저장된 값을 불러오는데 사용한다.

3. **add_on_set_parameters_callback 함수** → 서비스 형태로 파라미터 변경 요청이 있을 때 사용되는 함수로 지정된 콜백 함수를 호출하게 된다.

아래 코드를 보자

declare_parameter 함수로 max_random_num 파라미터를 선언하면 get_parameter 함수를 통해 호출된 파라미터 파일에 저장된 파라미터 초깃값을 가져와 설정한다.

그 뒤 파라미터 변경 요청이 있을 땐 **add_on_set_parameters_callback 함수**를 통해 지정된 callback 함수인 **update_parameter 함수**를 실행한다. 이 콜백 함수에서는 변경하려는 파라미터의 이름과 파라미터 타입이 같을 때 해당 파라미터 값을 변경한다.

```
class Argument(Node):

    def __init__(self):
        super().__init__('argument')
        self.declare_parameter('qos_depth', 10)
        qos_depth = self.get_parameter('qos_depth').value
        self.declare_parameter('min_random_num', 0)
        self.min_random_num = self.get_parameter('min_random_num').value
        self.declare_parameter('max_random_num', 9)
        self.max_random_num = self.get_parameter('max_random_num').value
        self.add_on_set_parameters_callback(self.update_parameter)
...

  def update_parameter(self, params):
        for param in params:
            if param.name == 'min_random_num' and param.type_ == Parameter.Type.INTEGER:
                self.min_random_num = param.value
            elif param.name == 'max_random_num' and param.type_ == Parameter.Type.INTEGER:
                self.max_random_num = param.value
        return SetParametersResult(successful=True)
```

argument 노드는 랜덤으로 생성되는 변수 a,b의 생성 범위를 파라미터를 통해 설정할 수 있도록 했는데 다음 코드와 같이 파라미터 값을 담고 있는 min, max_random_num변수를 이용한다

```
def publish_random_arithmetic_arguments(self):
        msg = ArithmeticArgument()
        msg.stamp = self.get_clock().now().to_msg()
        msg.argument_a = float(random.randint(self.min_random_num, self.max_random_num))
        msg.argument_b = float(random.randint(self.min_random_num, self.max_random_num))
        self.arithmetic_argument_publisher.publish(msg)
        self.get_logger().info('Published argument a: {0}'.format(msg.argument_a))
        self.get_logger().info('Published argument b: {0}'.format(msg.argument_b))
```

파라미터 사용방법(CLI)

파라미터 관련 설정 후 ros2 param CLI를 이용해 파라미터를 조회하고 변경하고 읽어보자

```
rodel@rodel:~/robot_ws$ ros2 run topic_service_action_rclpy_example argument
[INFO] [1694018990.121317461] [argument]: Published argument a: 4.0
[INFO] [1694018990.121553947] [argument]: Published argument b: 5.0
[INFO] [1694018991.113970551] [argument]: Published argument a: 0.0
[INFO] [1694018991.114781410] [argument]: Published argument b: 7.0
[INFO] [1694018992.113939174] [argument]: Published argument a: 3.0
[INFO] [1694018992.114735202] [argument]: Published argument b: 3.0
[INFO] [1694018993.113481973] [argument]: Published argument a: 5.0
[INFO] [1694018993.114162644] [argument]: Published argument b: 4.0
[INFO] [1694018994.113885416] [argument]: Published argument a: 6.0
[INFO] [1694018994.114610429] [argument]: Published argument b: 4.0
[INFO] [1694018995.113957194] [argument]: Published argument a: 7.0
[INFO] [1694018995.114764923] [argument]: Published argument b: 4.0
[INFO] [1694018996.113958107] [argument]: Published argument a: 1.0
[INFO] [1694018996.114758143] [argument]: Published argument b: 3.0
[INFO] [1694018997.112918579] [argument]: Published argument a: 6.0
[INFO] [1694018997.113133095] [argument]: Published argument b: 3.0
[INFO] [1694018998.112924444] [argument]: Published argument a: 2.0
[INFO] [1694018998.113131277] [argument]: Published argument b: 4.0
[INFO] [1694018999.112911264] [argument]: Published argument a: 1.0
[INFO] [1694018999.113117670] [argument]: Published argument b: 8.0
[INFO] [1694019000.112923905] [argument]: Published argument a: 0.0
```

노드 실행 중 ros2 param list 명령어로 파라미터를 조회한다

```
rodel@rodel:~/robot_ws$ ros2 param list
/argument:
  max_random_num
  min_random_num
  qos_depth
  use_sim_time
```

파라미터 값을 확인하고 변경해보자

```
rodel@rodel:~/robot_ws$ ros2 param get /argument max_random_num
Integer value is: 9
```

```
rodel@rodel:~/robot_ws$ ros2 param set /argument max_random_num 100
Set parameter successful
rodel@rodel:~/robot_ws$ ros2 param get /argument max_random_num
Integer value is: 100
```

이러면 출력 범위가 달라진다.

기본 파라미터 설정 방법

한 두개 파라미터 경우엔 declare, get, set은 쉽다. 그러나 하나의 노드에 수 십개의 파라미터, 그리고 수십개의 노드가 실행된다면 관리하고 매번 설정하기 어렵다

복수 개의 파라미터를 저장하고 노드 실행 시 지정된 파라미터로 초깃값을 주고 싶을 때

파라미터 파일(*yaml)을 작성하고 이를 런치 파일에서 불러오는 방법을 사용한다

파라미터 파일을 yaml로 지정하고 패키지의 param폴더에 저장

```yaml
/**: # namespace and node name
  ros__parameters:
    qos_depth: 30
    min_random_num: 0
    max_random_num: 9
```

런치 파일에서 LaunchConfiguration 함수의 인자로 파라미터 파일의 경로를 지정하고 LaunchDescription 함수의 인자로 Node 함수를 이용해 parameters 인자에 앞서 지정한 파라미터 파일의 경로를 지정한다.

```
import os

from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node


def generate_launch_description():
    param_dir = LaunchConfiguration(
        'param_dir',
        default=os.path.join(
            get_package_share_directory('topic_service_action_rclpy_example'),
            'param',
            'arithmetic_config.yaml'))

    return LaunchDescription([
        DeclareLaunchArgument(
            'param_dir',
            default_value=param_dir,
            description='Full path of parameter file'),

        Node(
            package='topic_service_action_rclpy_example',
            executable='argument',
            name='argument',
            parameters=[param_dir],
            output='screen'),

        Node(
            package='topic_service_action_rclpy_example',
            executable='calculator',
            name='calculator',
            parameters=[param_dir],
            output='screen'),
    ])
```

새롭게 지정된 *.yaml 파일 및 *.launch.py 파일을 ROS 파일 시스템에 맞추어 설치하려면 다음과 같이 setup.py에 옵션을 추가해야한다.

```
(share_dir + '/launch', glob.glob(os.path.join('launch', '*.launch.py'))),
        (share_dir + '/param', glob.glob(os.path.join('param', '*.yaml'))),
    ],
```

11. 실행 인자 프로그래밍(파이썬)

프로그램 실행 시 옵션으로 인수를 추가하여 실행하는 경우가 있다. 이 때 사용되는 인자를 '실행 인자'라고 하고 main 함수에서 매개변수를 통해 접근할 수 있다.

참고로 파라미터는 매개변수, argument는 실행 인자로 풀이한다. 파라미터는 함수 선언 시 사용되고 argument는 함수 호출 시의 인자이다.

Ros2에서 실행 인자 처리

파이썬에서 인수들을 무시할 땐 args를 None으로 설정한 후 rclpy 모듈의 init 함수에 바로 넘기게 된다.

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

만약 실행 인자를 사용하고자 하면 실행명 및 실행 경로를 뺀 나머지를 argv(Argument vector)에 매개변수에 저장하고, 이를 rclpy모듈의 init 함수의 args 매개변수로 넘기게 된다. 그리고 argparse 모듈을 이용하여 실행 인자를 위한 구문 해석 프로그램을 작성해야 한다

```
def main(argsv =sys.argv[1:]):
(argparse 구문 추가)
    rclpy.init(args=args)
```

실행 인자의 구문 해석

```
import argparse
import sys

import rclpy

from topic_service_action_rclpy_example.checker.checker import Checker
```

```
def main(argv=sys.argv[1:]):
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument(
        '-g',
        '--goal_total_sum',
        type=int,
        default=50,
        help='Target goal value of total sum')
    parser.add_argument(
        'argv', nargs=argparse.REMAINDER,
        help='Pass arbitrary arguments to the executable')
    args = parser.parse_args()

    rclpy.init(args=args.argv)
    try:
        checker = Checker()
        checker.send_goal_total_sum(args.goal_total_sum)
        try:
            rclpy.spin(checker)
        except KeyboardInterrupt:
            checker.get_logger().info('Keyboard Interrupt (SIGINT)')
        finally:
            checker.arithmetic_action_client.destroy()
            checker.destroy_node()
    finally:
        rclpy.shutdown()


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

실행 인자의 구문해석 프로그램은 파이썬의 argparse 모듈을 이용하여 파서를 선언한 후 사용할 실행 인자를 지정하게 된다.

1. 파서 만들기(parser = argparse.ArgumentParser)
2. 인자 추가하기(parser.add_argument)
3. 인자 파싱하기(args = parser.parse_args())
4. 인자 사용하기(args.xxx)

파서 만들기

```
  def main(argsv =sys.argv[1:]):
    parser = argparse.ArgumentParser(formatter_class = argparse.ArgumentDefaultHelpFormatter)
```

인자 추가하기

```
parser.add_argument(
  '-g',
```

```
'--goal_total_sum',
type=int,
default =50,
help = 'Target goal value of total sum')
```

인자 파싱하기

```
args = parser.parse_args()
```

인자 사용하기

```
checker.send_goal_total_sum(args.goal_total_sum)
```