6 Продвинутая разработка ROS

Service. Пример сервера

Эта часть занятий будет сосредоточена в большей степени на практике. Прежде чем мы напишем первый сервис ROS, стоит еще раз посмотреть вводную часть о том, что такое Service в формулировках ROS.

В практической части мы даже сами запускали сервис сложения двух чисел и выполняли тестовые сложения при помощи утилиты `rosservice`.

Давайте напишем такой же сервис самостоятельно.

Service Server

> **Используемые типы сообщений**
>
> Для того, чтобы упростить нашу работу, мы будем использовать уже созданные сообщения из пакета с примерами `rospy_tutorials`. Создавать собсвенные сообщения для сервисов мы будем в разделе Администрование ROS.

Сервер - это программа, которая выполняет непосредственно полезную работу. В нашем случае складывает два числа.

Код программы (`service_server.py`)

```
import rospy
from rospy_tutorials.srv import AddTwoInts,AddTwoIntsResponse

def add_two_ints(req):
    sum = req.a + req.b
    print("Returning [%s + %s = %s]" % (req.a, req.b, sum))
    return AddTwoIntsResponse(sum)

```

```
9
10 rospy.init_node("add_server_node")
11 s = rospy.Service('add_two_ints', AddTwoInts, add_two_ints)
12 rospy.spin()
```

Начало вполне стандартное - импортируем `rospy` .

Дальше, по аналогии с топиками, которые требуют импорта используемой в них структуры данных, сервисы тоже требуют импорта структуры данных используемых сообщений.

```
1 import rospy
2 from rospy_tutorials.srv import AddTwoInts,AddTwoIntsResponse
```

В данном случае это две структуры: `AddTwoInts` для определения используемых типов запросов выполнения сервиса и `AddTwoIntsResponse` для объекта ответа с результатом работы.

Давайте быстро посмотрим, из чего они состоят: используем команду `rossrv` с параметром `show` , чтобы посмотреть состав структуры. Напишем путь до нашей структуры

```
1 rossrv show rospy_tutorials/AddTwoInts
2 ====
3 int64 a
4 int64 b
5 ---
6 int64 sum
```

Сейчас мы не будем останавливаться подробно на том, почему и как именно записываются структуры данных, просто примем к сведению, что вот эта часть int64 a и int64 b, это запрос к сервису, а вот эта часть, после трех тире - int 64 sum ответ сервиса.

Далее мы реализуем собственно саму "рабочую" функцию сложения. Из листинга выше мы понимаем, что в запросе будут находиться две переменные `a` и `b`, которые мы должны сложить.

```
def add_two_ints(req):
    sum = req.a + req.b
    print("Returning [%s + %s = %s]" % (req.a, req.b, sum))
    return AddTwoIntsResponse(sum)
```

Метод сервиса должен вернуть объект, который "понятен" ROS, поэтому мы возвращаем объект `AddTwoIntsResponse,` а не просто число.

Похожий код мы уже использовали, когда разбирали работу с топиками.

```
rospy.init_node("add_server_node")
s = rospy.Service('add_two_ints', AddTwoInts, add_two_ints)
rospy.spin()
```

Мы инициализируем ноду, создаем сервис с параметрами

- Имя созданного сервиса `add_two_ints`
- Тип используемых для работы сообщений `AddTwoInts`
- Функция для обработки полученных запросов `add_two_ints`

И запускаем "вечный" цикл, для того чтобы программа не выходила.

Сохраним и запустим файл

```
python3 service_server.py
```

Убедимся, что сервис появился в списке сервисов

```
rosservice list

/add_two_ints/get_loggers
/add_two_ints/set_logger_level
/add_two_ints
/rosout/get_loggers
/rosout/set_logger_level
```

Вызовем сервис через терминал и убедимся, что результат суммирования верный.

```
rosservice call /add_two_ints "a: 2
b: 3"
sum: 5
```

Можем считать, что мы смогли написать и проверить наш первый сервис, далее мы научимся вызывать этот сервис из программы клиента.

Service. Пример клиента

На уроке ранее мы написали Сервер `add_two_ints` Давайте напишем клиент для него.

Рассмотрим код (`service_client.py`)

```
import rospy
import random
from rospy_tutorials.srv import AddTwoInts, AddTwoIntsRequest

def add_two_ints_client(a, b):
    rospy.wait_for_service('add_two_ints')
    service_client = rospy.ServiceProxy('add_two_ints', AddTwoInts)

    resp = service_client.call(AddTwoIntsRequest(a, b))
    return resp.sum

a = random.randint(-100 , 100)
b = random.randint(-100 , 100)

sum = add_two_ints_client(a,b)
print("%s + %s = %s"%(a, b, sum))
```

Подключаем необходимые внешние библиотеки

```
import rospy
import random
from rospy_tutorials.srv import AddTwoInts, AddTwoIntsRequest
```

Библиотека random нам понадобится, чтобы мы могли генерировать случайные числа для запросов. Также мы получили два класса с сообщениями `AddTwoInts` и `AddTwoIntsRequest` . Первый определяет тип используемого сообщения, второй помогает создавать правильный объект при формировании запроса на сервер.

Далее мы рассмотрим саму функцию выполнения запроса на сервер

```
def add_two_ints_client(a, b):
    rospy.wait_for_service('add_two_ints')
    service_client = rospy.ServiceProxy('add_two_ints', AddTwoInts)

    resp = service_client.call(AddTwoIntsRequest(a, b))
    return resp.sum
```

Первая строчка - это ожидание, когда "появится" сервис. Данный код "заморозит" работу программы до тех пор, пока не будет доступен сервер (например, когда мы забыли запустить его сами). Это точно позволит обработать запрос, а не завершить программу ошибкой из-за вызова несуществующего сервиса.

```
rospy.wait_for_service('add_two_ints')
```

Далее мы создаем объект самого клиента.

```
service_client = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

Параметрами мы указывавшем имя сервиса и тип сообщений.

Непосредственно вызов сервиса.

```
resp = service_client.call(AddTwoIntsRequest(a, b))
```

При вызове мы создаем правильный для нас тип запроса `AddTwoIntsRequest`

И после выполнения запроса мы вернем результат

```
return resp.sum
```

Остальной код генерит два числа и вызывает нашу функцию сложения, которая потом вызывает уже сам сервис.

```
a = random.randint(-100 , 100)
b = random.randint(-100 , 100)

sum = add_two_ints_client(a,b)
print("%s + %s = %s"%(a, b, sum))
```

Запустим код

Прежде чем запустить клиента, нужно запустить сначала сам сервер, иначе клиент не сможет работать.

```
python3 service_server.py
```

После этого уже запустим клиент

```
python3 service_client.py

-59 + -37 = -96
```

Мы увидим результат работы клиента.

При этом в терминале сервера мы тоже увидим, что сервер обработал запрос

```
Returning [-59 + -37 = -96]
```

Дополнительные материалы

Основной пакет http://wiki.ros.org/Services

Примеров использования `rospy` http://wiki.ros.org/rospy/Overview/Services

Action. Пример сервера

Прежде всего, начнем с повторения уже пройденного материала из главы Введение: что такое Actions. Также мы самостоятельно в практической части уже запускали сервисы и отправляли команды для выполнения заданий.

В стандартной установке POC есть пример, который рассчитывает последовательность Фибоначчи до введенного клиентом порядкового номера.

С расчетом числа Фибоначчи мы уже сталкивались когда изучали основы python. Но кратко напомним, что такое ряд Фибоначчи. Это такая последовательность чисел, при которой каждое следующее число является суммой двух предыдущих. При этом вся последовательность начинается с нуля и единицы. Т.е. третий член последовательности: ноль плюс один - один. Четвертый: один плюс один - два. Следующий: два плюс один - три, потом два плюс три - пять, потом три плюс пять - восемь и т.д.

Формула вычисления числа

```
f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2)
```

Вот такой вот сервер, считающий ряд Фибоначчи до заданного порядкового номера мы и рассмотрим. При этом мы позаимствуем уже созданные сообщений из примеров, чтобы упростить нашу задачу. Более детально о создании сообщений для Actions мы поговорим в разделе Администрирование ROS.

Action Server

Рассмотрим код Action сервера

```python
import rospy
import actionlib
from actionlib_tutorials.msg import FibonacciAction, FibonacciFeedback, Fi

class FibonacciAction(object):

    _feedback = FibonacciFeedback()
    _result = FibonacciResult()

    def __init__(self, name):
        self._action_name = name
        self._as = actionlib.SimpleActionServer(self._action_name, Fibonac
        self._as.start()

    def execute_cb(self, goal):

        r = rospy.Rate(1)
        success = True

        # append the seeds for the fibonacci sequence
        self._feedback.sequence = []
        self._feedback.sequence.append(0)
        self._feedback.sequence.append(1)

        # publish info to the console for the user
        rospy.loginfo('%s: Executing, creating fibonacci sequence of order

        # start executing the action
        for i in range(1, goal.order):
            # check that preempt has not been requested by the client
            if self._as.is_preempt_requested():
                rospy.loginfo('%s: Preempted' % self._action_name)
                self._as.set_preempted()
                success = False
                break
            self._feedback.sequence.append(self._feedback.sequence[i] + se
            # publish the feedback
            self._as.publish_feedback(self._feedback)
            # this step is not necessary, the sequence is computed at 1 Hz
            r.sleep()

        if success:
            self._result.sequence = self._feedback.sequence
            rospy.loginfo('%s: Succeeded' % self._action_name)
            self._as.set_succeeded(self._result)

if __name__ == '__main__':
    rospy.init_node('fibonacci')
    server = FibonacciAction(rospy.get_name())
```

```
50      rospy.spin()
```

Начало, как всегда, состоит из подключения различных модулей

```
1  import rospy
2  import actionlib
3  from actionlib_tutorials.msg import FibonacciAction, FibonacciFeedback, Fi
```

Так как модуль работы с Action не является часть rospy, нам необходимо подключить его отдельно. Также по аналогии с топиками, которые требуют импорта используемой в них структуры данных, экшн-серверы тоже требуют импорта структуры данных используемых сообщений.

В данном случае `FibonacciAction` - это класс, описывающий используемые сообщения для работы, а также `FibonacciFeedback` и `FibonacciResult`, объекты для использования feedback (промежуточного результата) и result (результата выполнения задания).

Давайте посмотрим, из чего они состоят

```
1  roscd actionlib_tutorials/msg/
2  cat FibonacciAction.msg
3  ======
4  FibonacciActionGoal action_goal
5  FibonacciActionResult action_result
6  FibonacciActionFeedback action_feedback
```

Видно, что структура экшна состоит из трех других структур: запроса (goal), результата (result) и промежуточного результата (feedback). Мы не будем останавливаться подробно на том, как именно она собирается, а просто примем к сведению, что она создается автоматически при сборке пакета.

Давайте также посмотрим на структуры запроса, результата и фидбэка:

```
cat FibonacciGoal.msg
# ====== DO NOT MODIFY! AUTOGENERATED FROM AN ACTION DEFINITION ======
#goal definition
int32 order
```

Переменная в структуре всего одна `int32 order`, это и есть порядковый номер указанного члена, до которого нам надо провести расчет ряда. Обратите внимание, мы не импортируем эту структуру в нашу программу, хотя и неявно будем пользоваться ей. Это произойдет потому, что FibonacciGoal уже есть в составе структуры FibonacciAction.

Давайте также посмотрим на FibonacciFeedback и FibonacciResult

```
cat FibonacciFeedback.msg
====
int32[] sequence
```

```
cat FibonacciResult.msg

int32[] sequence
```

Это очень "похожие" объекты, оба содержат массивы `int32`. Но один ответ необходимо использовать как `Feedback`, другой как `Result`. И понятно, что в общем случае эти сообщения могут быть разными, разной структуры и состава, но в нашем случае они одинаковые.

С сообщениями мы разобрались, можем перейти и к самой программе.

Создаем наш основной класс. И инициализируем переменные _feedback и _result

```
class FibonacciAction(object):
    # create messages that are used to publish feedback/result
    _feedback = FibonacciFeedback()
```

```
4       _result = FibonacciResult()
```

Конструктор класса

```
1       def __init__(self, name):
2           self._action_name = name
3           self._as = actionlib.SimpleActionServer(self._action_name, Fibonac
4           self._as.start()
```

Переменная `self._as` - это переменная с объектом нашего ActionServer. При его инициализации мы передаем параметрами:

- Имя Action сервера
- Тип используемого сообщения (`FibonacciAction`)
- параметр execute_cb название функции обратного вызова для запуска после получения `goal`

Далее мы запускаем сервис командой `self._as.start()`

Мы обозначили при инициализации функцию обратного вызова execute_cb, рассмотрим ее.

Инициализация вспомогательных переменных

```
1   r = rospy.Rate(1)
2   success = True
```

rospy.Rate(1) Задает частоту "генерации" последовательности. success флаг завершения расчета последовательности.

```
1   self._feedback.sequence = []
2   self._feedback.sequence.append(0)
3   self._feedback.sequence.append(1)
```

Инициализация пустой последовательности для заполнения и "генерация" первый двух элементов.

```
    rospy.loginfo('%s: Executing, creating fibonacci sequence of order %i with s
```

Вывод отладочного сообщения

Ну и сам цикл расчета числа

```
# start executing the action
for i in range(1, goal.order):
    # check that preempt has not been requested by the client
    if self._as.is_preempt_requested():
        rospy.loginfo('%s: Preempted' % self._action_name)
        self._as.set_preempted()
        success = False
        break
    self._feedback.sequence.append(self._feedback.sequence[i] + self._feed
    # publish the feedback
    self._as.publish_feedback(self._feedback)
    # this step is not necessary, the sequence is computed at 1 Hz for dem
    r.sleep()
```

По общей логике этого блока мы "крутим" цикл от 1 до заданной цели `goal.order`

Если клиент запросил "отмену" операции `if self._as.is_preempt_requested():` то мы меняем статус сервера на "preempted", убирая флаг о успешном завершении, прерываем цикл `for`

Если с программой ничего не случилось, то продолжаем "работать". Заполняем последовательность новым элементом
`self._feedback.sequence[i] + self._feedback.sequence[i-1]`

Публикуем последовательность в feedback.

Метод `r.sleep()` создает паузу выполнения (`r = rospy.Rate(1)`)

```
if success:
    self._result.sequence = self._feedback.sequence
    rospy.loginfo('%s: Succeeded' % self._action_name)
    self._as.set_succeeded(self._result)
```

Если цикл `for` с расчётом последовательности закончился, и мы все еще в статусе "все хорошо", то стоит ответить заполненным `` `result ``

На этом работа по расчету последовательности закончена, как и функция `execute_cb`

Далее нам необходимо только правильно запустить сервер

```
if __name__ == '__main__':
    rospy.init_node('fibonacci')
    server = FibonacciAction(rospy.get_name())
    rospy.spin()
```

Уже привычный для нас код, инициализируем ROS ноду, инициализируем сервер и переходим в бесконечное ожидание функцией `rospy.spin()`

Запуск и проверка

Ранее мы уже запускали аналогичный Action Server, поэтому тут для нас все знакомо.

Запустим сервер

```
python3 action_server.py
```

Проверим, что появилась наша нода

```
rosnode list

/fibonacci
/rosout
```

Проверим, что появились необходимые топики

```
rostopic list
/fibonacci/cancel
/fibonacci/feedback
/fibonacci/goal
/fibonacci/result
/fibonacci/status
```

Запустим просмотр топиков result и feedback в разных консолях

```
rostopic echo /fibonacci/feedback
```

```
rostopic echo /fibonacci/result
```

Запустим расчет последовательности для числа 5

```
rostopic pub /fibonacci/goal actionlib_tutorials/FibonacciActionGoal "head
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
```

```
    secs: 0
    nsecs: 0
  id: ''
goal:
  order: 5"
```

В топике `feedback` мы увидим как раз в секунду заполняется последовательность

```
feedback:
  sequence: [0, 1, 1]
===
feedback:
  sequence: [0, 1, 1, 2]
===
feedback:
  sequence: [0, 1, 1, 2, 3]
===
feedback:
  sequence: [0, 1, 1, 2, 3, 5]
```

А в топике result увидим только одну запись

```
result:
  sequence: [0, 1, 1, 2, 3, 5]
```

Мы убедились, что наш `ActionServer` работает.

При получении `Goal` начинается расчет, а по мере расчета публикуются промежуточные данные `Feedback` и при завершении расчета публикуется результат `Result`

Action. Пример клиента

На прошлом уроке мы рассмотрели серверную часть экшн-сервера ROS, а в этом напишем клиента для нашего экшна.

Как правило, клиент экшн-сервера - это часть какого-то большого пакета, который реализует взаимодействие с пользователем. Например, в `rviz` передача цели, до который роботу надо ехать при помощи системы навигации, происходит именно через экшн-клиент рвиза.

Как мы помним, наш экшн-сервер считал ряд Фибоначчи до заданного члена и возвращал нам всю последовательность. Клиент, которого мы будем писать, будет передавать задание серверу на расчет последовательности и выводить результат после окончания расчета.

```
import rospy
import actionlib

from actionlib_tutorials.msg import FibonacciAction, FibonacciGoal

def fibonacci_client():

    client = actionlib.SimpleActionClient('fibonacci', FibonacciAction)
    client.wait_for_server()
    goal = FibonacciGoal(order=5)
    client.send_goal(goal)

    client.wait_for_result()

    return client.get_result()


rospy.init_node('fibonacci_client')
result = fibonacci_client()

print("Have result:")
print(result)
```

Подключаем все необходимые библиотеки и сообщения

```
import rospy
import actionlib

from actionlib_tutorials.msg import FibonacciAction, FibonacciGoal
```

Инициализируем клиента для сервера. В параметрах укажем имя Экшен Сервера (`fibonacci`) и тип используемых сообщений (`FibonacciAction`)

```
client = actionlib.SimpleActionClient('fibonacci', FibonacciAction)
```

Ожидаем подключение к серверу (необходимо запустить сервер, чтобы программы клиент смогла работать далее)

```
client.wait_for_server()
```

Создадим объект `` `Goal `` , обозначим, что хотим расчет до 5. И запустим выполнение задания.

```
goal = FibonacciGoal(order=5)
client.send_goal(goal)
```

Далее мы ожидаем выполнение задания. После получения результата мы его возвращаем и выходим функции.

```
client.wait_for_result()
return client.get_result()
```

Запустим и проверим

Перед запуском клиента проверим, что у нас запущен Action Server

```
python3 action_client.py
====
sequence: [0, 1, 1, 2, 3, 5]
```

Наш Action Client работает.

Server parameters

На этом уроке мы столкнемся с новой для нас темой - Сервер Параметров (Parameter Server).

Сервер параметров проще всего описать как некоторое сетевое хранилище, где можно хранить переменные, которые мы хотим использовать в наших программах.

Например, если в программе есть какая-либо настройка (например максимальная скорость движения), то нам удобно хранить эти настройки в одном месте. Таким образом можем довольно просто разобраться, какие настройки есть у каких программ и какие у этих настроек текущие значения. Также для изменения настроек нам не нужно менять код программы, а достаточно изменить параметры на сервере.

Сервер параметров может хранить только простые переменные, числа, строки и тп. Он не создан для хранения сложных типов данных, например сообщений ROS.

Для работы с параметрами есть удобная консольная утилита `rosparam`

```
rosparam list: Получить список параметров
rosparam get parameter: Прочитать значение параметра
rosparam set parameter value: Установить значение параметра
rosparam delete parameter: Удалить значение параметра
rosparam dump file: Сохранить все переменные в файл
rosparam load file: Восстановить переменные из файла
```

Использование параметров в программах

Библиотека `rospy` имеет все необходимые методы для работы с Сервером параметров.

Рассмотрим простой пример

```python
import rospy

rospy.init_node('params_demo')
default_param = "some_value"

try:
    my_param = rospy.get_param("~my_param")
except KeyError as e:
    my_param = default_param
    rospy.set_param("~my_param", my_param)

print(my_param)
```

Знак "~" в названии параметра означает, что параметр "приватный" и принадлежит конкретной ноде.

В этой программе мы "пытаемся" получить значение параметра `my_params`, если у нас не получилось (параметра пока нет на сервере), то мы сохраняем параметр на сервере со значением по умолчанию.

При запуске программы мы увидим вывод строчки `some_value`, что является значением по умолчанию.

Получим список параметров

```
rosparam list

/params_demo/my_param
/rosdistro
/roslaunch/uris/host_ubuntu__33259
/rosversion
/run_id
```

Мы видим, что параметр `my_param` появился в списке. При этом полный адрес параметра начинается с `/params_demo`, что является названием ноды.

Прочитаем его и убедимся, что параметр содержит установленное значение.

```
rosparam get /params_demo/my_param

some_value
```

Поменяем значение

```
rosparam set /params_demo/my_param other_value
```

Запустим наш скрипт и убедимся, что в `python` мы получили новое значение, а не значение по умолчанию.

```
python params.py
other_value
```

Также можно использовать конструкцию получения параметров со значением по умолчанию

```
default_param = rospy.get_param('default_param', 'default_value')
```

Тогда если параметр будет найден на сервере, то мы получим значение с сервера, а если нет, значение по умолчанию. Но у такого способа есть небольшой минус: параметр не сохранится на сервере, поэтому мы не сможем найти его командой `rosparam list`. Но если мы знаем имя, мы можем использовать `rosparam set/get` и провести его настройку.

ROS Bags. Работа с данными

На этом занятии мы столкнемся с новой для нас темой - это `bag` файлы.

Идея их использования довольно проста и немного похожа на машину времени. Мы можем включить "запись" и сохранять все сообщения, которые происходят в нашей системе, результат этой записи сохраняется в `.bag` файлах.

Также, имея `.bag` файл, мы можем сколь угодно раз проигрывать записанный отрезок времени. Такой подход упрощает отладку алгоритмов, когда нам для проверки не нужно моделировать объекты реального мира, а достаточно один раз записать данные.

Для работы с `.bag` служит консольная утилита `rosbag`, перечислим основные ее аргументы

```
rosbag record [TOPIC_NAME] Начать запись .bag
rosbag play [FILE_NAME] Проиграть .bag файл
rosbag compress [FILE_NAME] Архивировать файл
rosbag decompress [FILE_NAME] Разархивировать файл
rosbag record -a Записать все сообщения
```

Применение

Запустим скрипт, который мы написали когда разбирались с топиками `pub.py`. Мы помним, что он публикует данные в топик `/welcome_topic`

```
python3 pub.py
```

Проверим, что в топике действительно есть данные

```
rostopic echo /welcome_topic

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

Начнем запись топика. Лучше подождать секунд 15, прежде чем остановить запись.

```
rosbag record /welcome_topic
[ INFO] [1625835886.569004918]: Subscribing to /welcome_topic
[ INFO] [1625835886.574587140]: Recording to 2021-07-09-16-04-46.bag.
```

Мы видим, что запись пошла в файл `2021-07-09-16-04-46.bag`

Остановим тестовый скрипт и "проиграем" файл

```
rosbag play 2021-07-09-16-04-46.bag
[ INFO] [1625836223.604592718]: Opening 2021-07-09-16-04-46.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1625835902.209460   Duration: 14.730354 / 23.037502
```

В топике `welcome_topic` появились данные.

```
rostopic echo /welcome_topic
data: "Hello robot"
---
data: "Hello robot"
---
data: "Hello robot"
---
```

Практическое задание

Разработка ActionClient

В данный момент клиент работает "молча" пока идет расчет всей последовательности. Но сервер отправляет промежуточные данные (feedback). Необходимо модифицировать программу клиент, чтобы он начал выводить на экран полученные промежуточные данные.

Пример вывода работы программы

```
python action_client_fb.py
sequence: [0, 1, 1]
sequence: [0, 1, 1, 2]
sequence: [0, 1, 1, 2, 3]
sequence: [0, 1, 1, 2, 3, 5]
Have final result:
sequence: [0, 1, 1, 2, 3, 5]
```

Разработка Service

Написать `Service Server`, который имитирует управление обогревателем. Для управления необходимо использовать тип сообщений `std_srvs/SetBool`, где `True` - включение обогревателя, `False` - выключение.

Для обозначения изменения статуса работы обогревателя необходимо в консоль выводить его статус "`Heater:On` или `Heater:Off`" при изменении режима работы.

Написать `Service Client`, который, используя данные из топика `/temp`, будет управлять, используя сервис обогревателя.

В топике `/temp` находится температура окружающей среды.

При понижении температуры ниже `18` градусов необходимо включить обогреватель (вызвать сервис). При повышении температуры выше `23` градусов необходимо выключить обогреватель.

Данные с топиком температуры находятся в .bag файле temp.bag

Использование параметров

Для программы управления обогревателем необходимо добавить два системных параметра `/low_temp`, `/hight_temp`, определяющих границы управления обогревателем.

Изменить настройки программы через параметры на `20` и `25` градусов.