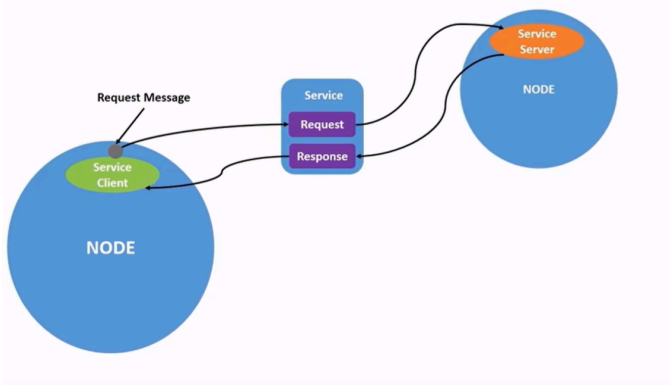


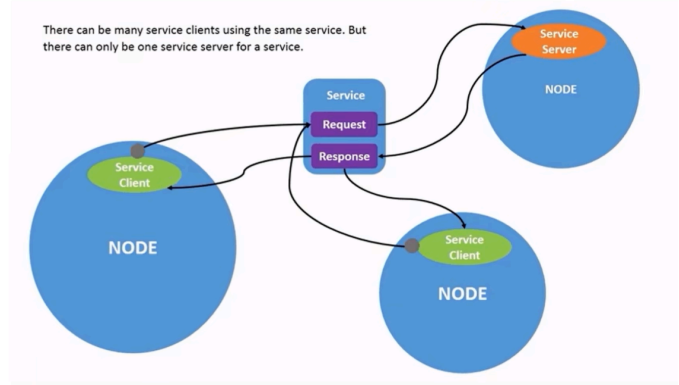
一、服务

1. service

- 客户端/服务器 (C/S) 模型
- 同步通信机制
- 服务器端唯一，客户端可以不唯一
- .srv文件定义请求和应答数据结构



客户端/服务器 (C/S) 模型



一对多

2. server

示例程序：

```
4 """
5 @作者: 古月居(www.guyuehome.com)
6 @说明: ROS2服务示例-提供加法器的服务器处理功能
7 """
8
9 import rclpy # ROS2 Python接口库
10 from rclpy.node import Node # ROS2 节点类
11 from learning_interface.srv import AddTwoInts # 自定义的服务接口
12
13 class adderServer(Node):
14     def __init__(self, name):
15         super().__init__(name) # ROS2节点父类初始化
16         self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.adder_callback) # 创建服务器对象(接口类型、服务名、服务器回调函数)
17
18     def adder_callback(self, request, response):
19         response.sum = request.a + request.b # 创建回调函数, 执行收到请求后对数据的处理
20         self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b)) # 完成加法求和计算, 将结果放到反馈的数据中
21         return response # 输出日志信息, 提示已经完成加法求和计算
22         # 反馈应答信息
23
24 def main(args=None):
25     # ROS2节点主入口main函数
26     rclpy.init(args=args) # ROS2 Python接口初始化
27     node = adderServer("service_adder_server") # 创建ROS2节点对象并进行初始化
28     rclpy.spin(node) # 循环等待ROS2退出
29     node.destroy_node() # 销毁节点对象
30     rclpy.shutdown() # 关闭ROS2 Python接口
```

server和话题中的subscriber类似，不知道信息什么时候来，所以会有个回调函数的机制

程序流程：

- 编程接口初始化
- 创建节点并初始化
- 创建服务器端对象
- 通过回调函数处进行服务
- 向客户端反馈应答结果
- 销毁节点并关闭接口

3. client

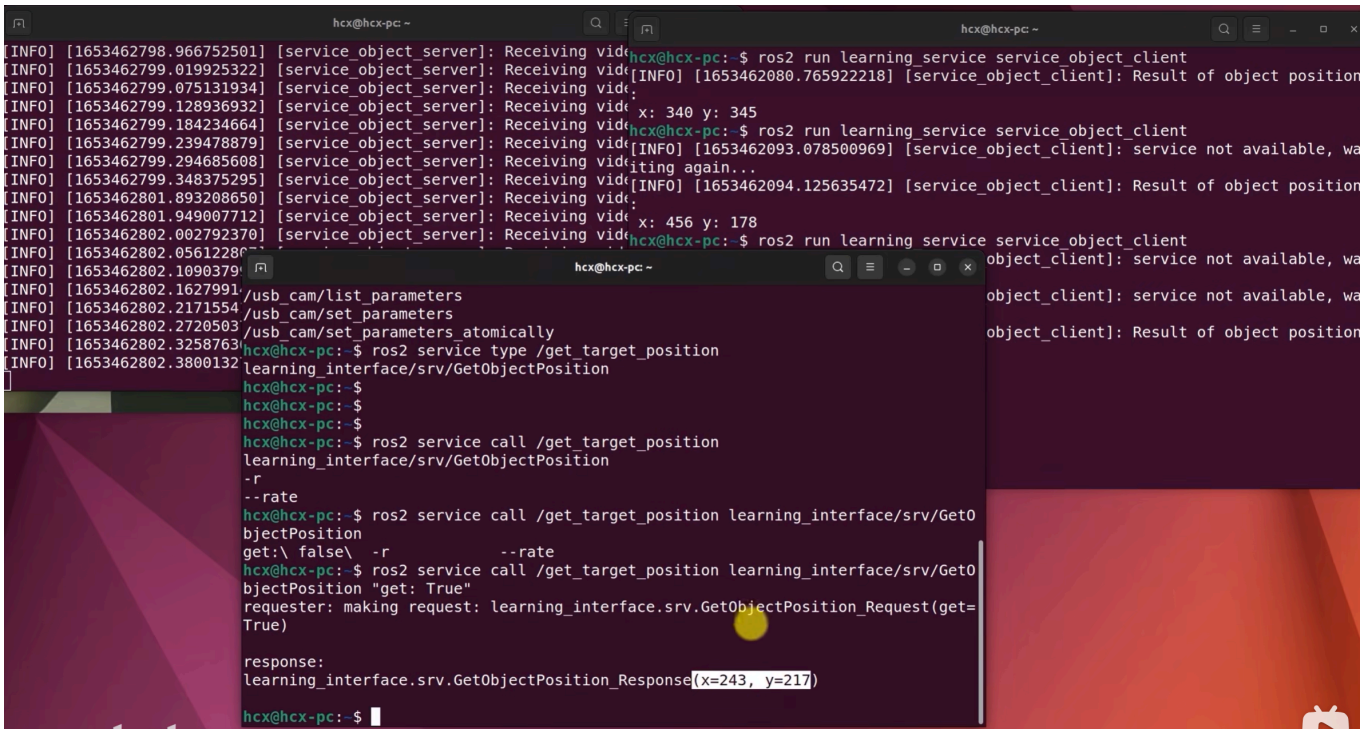
示例程序：

```
learning_service > learning_service > service_adder_client.py > adderClient > send_request
11 import rclpy
12 from rclpy.node import Node
13 from learning_interface.srv import AddTwoInts
14
15 class adderClient(Node):
16     def __init__(self, name):
17         super().__init__(name)
18         self.client = self.create_client(AddTwoInts, 'add_two_ints')
19         while not self.client.wait_for_service(timeout_sec=1.0):
20             self.get_logger().info('service not available, waiting again...')
21         self.request = AddTwoInts.Request()
22
23     def send_request(self):
24         self.request.a = int(sys.argv[1])
25         self.request.b = int(sys.argv[2])
26         self.future = self.client.call_async(self.request)
27
28 def main(args=None):
29     rclpy.init(args=args)
30     node = adderClient("service_adder_client")
31     node.send_request()
32
33     while rclpy.ok():
34         rclpy.spin_once(node)
35
36         if node.future.done():
37             try:
38                 response = node.future.result()
39             except Exception as e:
40                 node.get_logger().info(
41                     'Service call failed %r' % (e,))
42             else:
43                 node.get_logger().info(
44                     'Result of add_two_ints: for %d + %d = %d' %
45                     (node.request.a, node.request.b, response.sum))
46                 break
47
48     node.destroy_node()
49     rclpy.shutdown()
```

程序流程：

- 编程接口初始化
- 创建节点并初始化
- 创建客户端对象
- 创建并发送请求数据
- 等待服务器端应答数据
- 销毁节点并关闭接口

另：用命令行也可以发起一个服务请求（起到和client程序类似的作用）



```
hcx@hcx-pc ~
[INFO] [1653462798.966752501] [service_object_server]: Receiving vid:
hcx@hcx-pc: $ ros2 run learning_service service_object_client
[INFO] [1653462799.019925322] [service_object_server]: Receiving vid:
[INFO] [1653462799.075131934] [service_object_server]: Receiving vid:
[INFO] [1653462799.128936932] [service_object_server]: Receiving vid:
[INFO] [1653462799.184234664] [service_object_server]: Receiving vid:
[INFO] [1653462799.239478879] [service_object_server]: Receiving vid:
hcx@hcx-pc: $ ros2 run learning_service service_object_client
[INFO] [1653462799.294685608] [service_object_server]: Receiving vid:
[INFO] [1653462799.348375295] [service_object_server]: Receiving vid:
[INFO] [1653462801.893208650] [service_object_server]: Receiving vid:
[INFO] [1653462801.949007712] [service_object_server]: Receiving vid:
[INFO] [1653462802.002792370] [service_object_server]: Receiving vid:
[INFO] [1653462802.056122800] [service_object_server]: Receiving vid:
hcx@hcx-pc: $ ros2 run learning_service service_object_client
[INFO] [1653462802.109037900] [service_object_server]: Receiving vid:
[INFO] [1653462802.162799100] [service_object_server]: Receiving vid:
[INFO] [1653462802.217155400] [service_object_server]: Receiving vid:
[INFO] [1653462802.272050300] [service_object_server]: Receiving vid:
[INFO] [1653462802.325876300] [service_object_server]: Receiving vid:
[INFO] [1653462802.380013200] [service_object_server]: Receiving vid:
hcx@hcx-pc: $ ros2 service type /get_target_position
learning_interface/srv/GetObjectPosition
hcx@hcx-pc: $
hcx@hcx-pc: $
hcx@hcx-pc: $
hcx@hcx-pc: $ ros2 service call /get_target_position learning_interface/srv/GetObjectPosition
learning_interface/srv/GetObjectPosition
-r
--rate
hcx@hcx-pc: $ ros2 service call /get_target_position learning_interface/srv/GetObjectPosition
get:\ false\ -r --rate
hcx@hcx-pc: $ ros2 service call /get_target_position learning_interface/srv/GetObjectPosition "get: True"
requester: making request: Learning_interface.srv.GetObjectPosition_Request(get=
True)

response:
learning_interface.srv.GetObjectPosition_Response(x=243, y=217)
hcx@hcx-pc: $
```

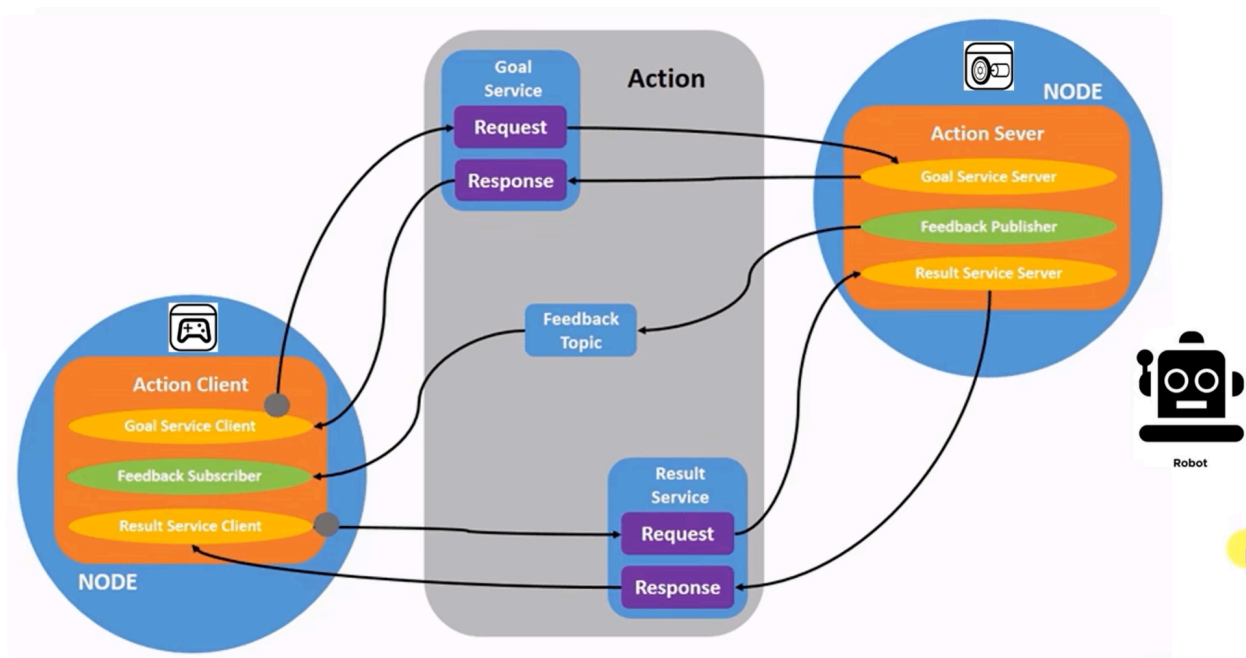
二、动作

1. 基本概念

动作：完整行为的流程管理

- 客户端/服务器（C/S）模型
- 服务器端唯一，客户端可以不唯一
- 同步通信机制
- .action文件定义通信接口的数据结构

（发送控制命令的是client，受控的机器人是server）



bilibili

客户端/服务器 (C/S) 模型

从上图可以看到，action是一个应用层的模型，底层仍然是用topic和service实现通信的。

2. 程序实现

虽然action是由service和topic实现的，但编程时我们并不用service和topic去编，而是action也有一套封装好的编程接口。

示例程序：

server：

```
"""
```

```
@作者：古月居(www.guyuehome.com)  
@说明：ROS2动作示例-负责执行圆周运动动作的服务端  
"""
```

```
import time
```

```
import rclpy # ROS2 Python接口库  
from rclpy.node import Node # ROS2 节点类  
from rclpy.action import ActionServer # ROS2 动作服务器类  
from learning_interface.action import MoveCircle # 自定义的圆周运动接口
```

```
class MoveCircleActionServer(Node):
```

```
    def __init__(self, name):  
        super().__init__(name) # ROS2节点父类初始化  
        self.action_server = ActionServer( # 创建动作服务器 (接口类型、动作名、回调函数)  
            self,  
            MoveCircle,  
            'move_circle',  
            self.execute_callback)
```

```
    def execute_callback(self, goal_handle): # 执行收到动作目标之后的处理函数  
        self.get_logger().info('Moving circle...')  
        feedback_msg = MoveCircle.Feedback() # 创建一个动作反馈信息的信息  
  
        for i in range(0, 360, 30): # 从0到360度, 执行圆周运动, 并周期反馈信息  
            feedback_msg.state = i # 创建反馈信息, 表示当前执行到的角度  
            self.get_logger().info('Publishing feedback: %d' % feedback_msg.state)  
            goal_handle.publish_feedback(feedback_msg) # 发布反馈信息  
            time.sleep(0.5)  
  
        goal_handle.succeed() # 动作执行成功  
        result = MoveCircle.Result() # 创建结果消息  
        result.finish = True  
        return result # 反馈最终动作执行的结果
```

client:

```
def send_goal(self, enable):
    goal_msg = MoveCircle.Goal()
    goal_msg.enable = enable

    self._action_client.wait_for_server()
    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)

def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected :(')
        return

    self.get_logger().info('Goal accepted :)')

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


def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info('Result: {%d}' % result.finish)

def feedback_callback(self, feedback_msg):
    feedback = feedback_msg.feedback
    self.get_logger().info('Received feedback: {%d}' % feedback.state)
```

(client这里会有比较多的callback回调函数)

FAQ:

在ROS2中，我知道action是由request service, feedback topic, result service组成的。我不理解result是什么时候client请求的？

 在 ROS 2 的 Action 通信机制中，Client 请求 Result 的时机是非常明确的：**在 Action Goal 被 Server 接收 (Accept) 之后，Client 会立即发送一个异步的 Result 请求。**

为了让你透彻理解，我们可以把 Action 的执行分为三个阶段：

1. 目标传递阶段 (Goal Handling)

- Client 发送 `SendGoal` 请求。
- Server 收到后决定是“接受”还是“拒绝”。
- **关键点：** 如果 Server **接受了**目标 (Status 变为 `ACCEPTED`)，ROS 2 的 Action Client 库底层就会**自动且立即**发起一个 `GetResult` 的 Service 请求。

2. 执行与反馈阶段 (Execution & Feedback)

- 此时，Server 开始执行任务，并通过 `Feedback Topic` 持续发送进度。
- **与此同时**，上一步发出的 `GetResult` 请求正处于“挂起 (Pending)”状态。Server 并不会立刻响应这个 Service 请求，而是把它“拿在手里”，直到任务结束。

3. 结果返回阶段 (Result Delivery)

- 当 Server 完成任务（成功、失败或被取消）时，它会填充 Result 数据。
- 此时，Server 才会对之前那个一直挂着的 `GetResult` 请求做出 Response。
- Client 收到 Response 后，触发对应的 Result 回调函数。

为什么要这么设计？（解除你的误区）

你可能直觉上觉得应该是“任务完了，Client 再去问结果”，但 ROS 2 采用了“提前订阅结果”的模式：

1. **异步非阻塞**：Client 不需要自己去轮询（Polling）任务完没完。通过在开始时就发送 Result 请求，Client 只需要在自己的事件循环中等待那个 Service Response 回来即可。
2. **状态一致性**：这种设计确保了 Client 能够第一时间抓住任务结束的瞬间，拿到最终状态。

总结

Client 请求 Result 的时间点：只要 Goal 被 Server 接受了，Client 就会立刻发出 Result 请求。

Server 返回 Result 的时间点：只有当任务真正结束（SUCCEEDED / ABORTED / CANCELED）时，Server 才会给这个请求发回响应。

所以，在代码层面，你通常在发送 Goal 的同时就会设置一个 `result_callback`，这个回调函数对应的底层 Service 请求在目标被接受那一刻就已经发出去了。



这样异步的话，是不是client不用一直等server完成，可以干别的事，虽然同时一直能拿到feedback

 是的，你的理解完全正确！这正是 ROS 2 Action 设计的核心优势：**全双工、非阻塞的异步通信**。

由于 Client 在发送 Goal 并得到 Server “接受”的确认后，就立即发出了 Result 请求，这个请求在底层是**异步 (Asynchronous)** 的。

以下是 Client 在这个过程中能“一心多用”的具体表现：

1. Client 可以自由执行本地代码

当 `send_goal_async()` 发出后，Client 的主线程（或控制循环）不会被卡住。你可以继续：

- 处理用户的输入。
- 运行其他的控制算法。
- 调用其他的 Service 或发布其他的 Topic。

2. Feedback 是“被动接收”的（通过 Callback）

你不需要去“取”反馈。每当 Server 发布一条 Feedback，Client 预设好的 `feedback_callback` 就会被触发。这就像是一个独立的后台任务，完全不干扰你主逻辑的运行。

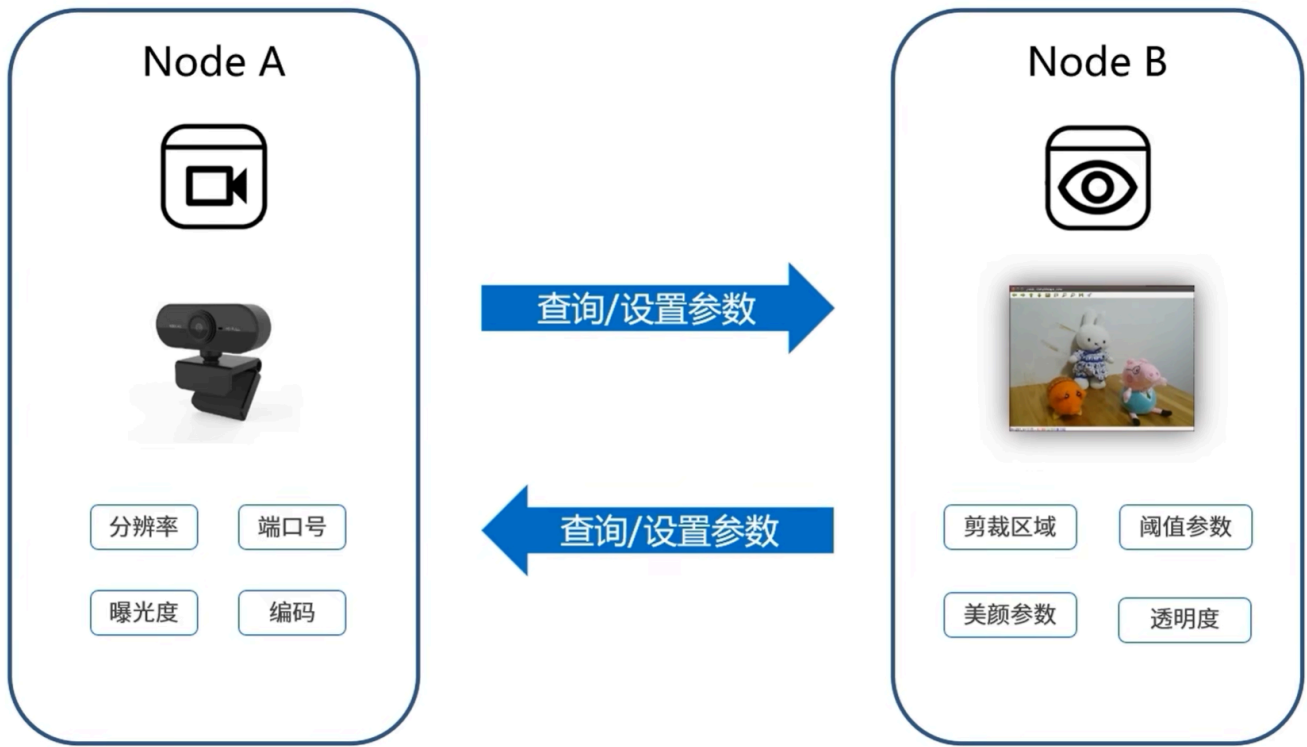
3. Result 是“静候通知”的（通过 Future/Callback）

Client 发出的 Result 请求就像是给 Server 留了个“回执单”：

- 在 C++ 中，这通常表现为一个 `shared_future`。你可以随时检查 `future.get_status()` 看看任务完没完，或者干脆不管它，等回调函数执行。
- 在 Python 中，这通常是一个 `Task` 或 `Future` 对象。

三、参数

1. 基本概念



参数：机器人系统的全局字典



- 全局共享字典
- 由键与值组成
- 可实现动态监控

2. 命令行使用

```

alpharos2@alpharos2-VMware-Virtual-Platform:~$ ros2 param
usage: ros2 param [-h] Call `ros2 param <command> -h` for more detailed usage. ...

Various param related sub-commands

options:
  -h, --help            show this help message and exit

Commands:
  delete                Delete parameter
  describe              Show descriptive information about declared parameters
  dump                  Show all of the parameters of a node in a YAML file format
  get                   Get parameter
  list                  Output a list of available parameters
  load                  Load parameter file for a node
  set                   Set parameter

Call `ros2 param <command> -h` for more detailed usage.

```

describe: 详细讲解某一个参数

dump: 比如 `ros2 param dump turtlesim >> filename.yaml` 的话是在当前路径生成一个yaml 配置文件（不加 `>> filename.yaml` 的话就是把该功能包的所有参数打印出来），之后再使用 `ros2 param load turtlesim filename.yaml` 就可以一键导入该文件中设置的所有参数。

get: 得到一个param的值

set: set一个param的值

示例:

```
hcx@hcx-pc:~$ ros2 param describe turtlesim background_b
Parameter name: background_b
Type: integer
Description: Blue channel of the background color
Constraints:
  Min value: 0
  Max value: 255
  Step: 1
hcx@hcx-pc:~$ ros2 param get turtlesim background_b
Integer value is: 255
hcx@hcx-pc:~$ ros2 param set turtlesim background_b 10
Set parameter successful
hcx@hcx-pc:~$ ros2 param dump turtlesim
Saving to: ./turtlesim.yaml
hcx@hcx-pc:~$ ros2 param load turtlesim turtlesim.yaml
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter qos_overrides./parameter_events.publisher.depth failed: parameter 'qos_overrides./parameter_events.publisher.depth' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.durability failed: parameter 'qos_overrides./parameter_events.publisher.durability' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.history failed: parameter 'qos_overrides./parameter_events.publisher.history' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.reliability failed: parameter 'qos_overrides./parameter_events.publisher.reliability' cannot be set because it is read-only
Set parameter use_sim_time successful
hcx@hcx-pc:~$
```

```
turtlesim.yaml
1 /turtlesim:
2   ros__parameters:
3     background_b: 10
4     background_g: 86
5     background_r: 69
6     qos_overrides:
7       /parameter_events:
8         publisher:
9           depth: 1000
10          durability: volatile
11          history: keep_last
12          reliability: reliable
13    use_sim_time: false
```

查看/修改参数

- \$ ros2 param describe turtlesim background_b
- \$ ros2 param get turtlesim background_b
- \$ ros2 param set turtlesim background_b 10
- \$ ros2 param dump turtlesim >> turtlesim.yaml
- \$ ros2 param load turtlesim turtlesim.yaml

3. 示例程序

```
"""
@作者: 古月居(www.guyuehome.com)
@说明: ROS2参数示例 - 创建、读取、修改参数
"""

import rclpy # ROS2 Python接口库
from rclpy.node import Node # ROS2 节点类

class ParameterNode(Node):
    def __init__(self, name):
        super().__init__(name) # ROS2节点父类初始化
        self.timer = self.create_timer(2, self.timer_callback) # 创建一个定时器 (单位为秒的周期, 定时执行的回调函数)
        self.declare_parameter('robot_name', 'mbot') # 创建一个参数, 并设置参数的默认值

    def timer_callback(self):
        robot_name_param = self.get_parameter('robot_name').get_parameter_value().string_value # 从ROS2系统中读取参数的值
        self.get_logger().info('Hello %s!' % robot_name_param) # 输出日志信息, 打印读取到的参数值

        new_name_param = rclpy.parameter.Parameter('robot_name', # 重新将参数值设置为指定值
            rclpy.Parameter.Type.STRING, 'mbot')
        all_new_parameters = [new_name_param]
        self.set_parameters(all_new_parameters) # 将重新创建的参数列表发送给ROS2系统

def main(args=None):
    # ROS2节点主入口main函数
    rclpy.init(args=args) # ROS2 Python接口初始化
    node = ParameterNode("param_declare") # 创建ROS2节点对象并进行初始化
    rclpy.spin(node) # 循环等待ROS2退出
    node.destroy_node() # 销毁节点对象
    rclpy.shutdown() # 关闭ROS2 Python接口
```

FAQ:

1. ROS2中参数实现的原理是什么？一个节点参数是所有节点可见的吗？

1. 参数实现的原理：它是特殊的 Service

ROS 2 的参数并不是一种独立的通信协议，它的底层其实是基于 **Service（服务）** 实现的。

当你为一个节点定义参数时，ROS 2 背后会自动为该节点创建一系列隐形的 **Services**。你可以通过命令 `ros2 service list` 观察到，每个节点下面都会多出类似这样的服务：

- `/<node_name>/get_parameters`
- `/<node_name>/set_parameters`
- `/<node_name>/list_parameters`
- `/<node_name>/describe_parameters`

工作流程：

1. **存储：** 参数保存在**每个节点内部**的一个“参数字典”中（通常是一个 Map 结构）。
2. **查询：** 当你想查看参数时（例如 `ros2 param get`），实际上是调用了该节点的 `get_parameters` 服务。
3. **修改：** 当你修改参数时（例如 `ros2 param set`），是向该节点发送了一个 `set_parameters` 的服务请求，节点收到后更新内部字典。

2. 参数是“所有节点可见”的吗？

答案是：**可见（能看到名字和值），但不是“全局共享”的，而是“属于特定节点”的。**

这里有两个关键点需要区分：

A. 归属感：参数属于特定节点

参数是**私有于节点**的。不存在一个叫“全局参数池”的地方存放所有参数。

- 节点 A 的参数 `my_param` 和节点 B 的参数 `my_param` 是完全独立的两个变量。
- 如果你要访问参数，你必须明确指定是**哪个节点**的参数。
- 这就是为什么你在命令里必须写 `ros2 param get <node_name> <param_name>`。

B. 访问权：任何节点都可以去“问”


虽然参数属于某个节点，但因为它是通过 **Service** 实现的，而在 ROS 2 网络中，Service 默认是公开的。

- **节点 B 可以访问节点 A 的参数：** 只要节点 B 知道节点 A 的名字，它就可以创建一个 Service Client 来调用节点 A 的参数服务。
 - **外部工具可以访问：** 比如 `rviz2` 或 `rqt` 可以列出系统中所有节点的参数并修改它们，因为它们扫描到了那些隐藏的参数服务。
-

特性	ROS 2 参数 (Parameters)
底层实现	基于 Service (请求-响应模式)
归属关系	属于单个节点 (Node-specific)
可见性	全网可见 (只要知道节点名, 谁都能去读写)
实时性	支持参数监听回调 (当参数被别人改了, 节点能立刻发现并做出反应)

2. set的权限支持设置吗?

set的权限支持设置吗? 还是说所有节点想set就set?

 在 ROS 2 的默认机制中, 确实没有像数据库或操作系统那样复杂的“用户/组权限控制”。也就是说, 只要一个节点在同一个 ROS 2 网络中 (同一个 `ROS_DOMAIN_ID`), 它理论上可以向任何其他节点发送 `set_parameters` 的请求。

但是, 这并不意味着“谁都能随便改成功”。节点拥有最终的拒绝权。

你可以通过以下三种方式来控制参数的“修改权限”:

1. 参数描述符 (Parameter Descriptor) 中的 `read_only` (最强硬的限制)

在定义参数时, 你可以显式地将其标记为只读。

- 效果: 一旦设置为 `read_only=true`, 无论是节点内部还是外部工具, 都无法在运行时修改这个参数的值。
- 代码示例 (C++):

```
C++  
  
auto descriptor = rcl_interfaces::msg::ParameterDescriptor();  
descriptor.read_only = true;  
this->declare_parameter("my_readonly_param", 10, descriptor);
```

如果你尝试用 `ros2 param set` 修改它, 会直接报错。

2. 参数修改回调函数 (On Set Parameters Callback) —— “逻辑防火墙”

这是最常用的办法。节点可以注册一个回调函数，每当有人（无论是自己还是外部节点）尝试修改参数时，这个函数都会先运行。

- **原理：** 回调函数会收到拟修改的参数值，它必须返回一个 `SetParametersResult`。
- **你可以决定：**
 - **校验数值：** “如果改的值超过 100，我就拒绝（return successful=false）”。
 - **限制逻辑：** 你甚至可以根据业务状态拒绝。比如“如果机器人正在移动，禁止修改最高限速”。
- **示例逻辑：**

```
C++

// 伪代码逻辑
if (new_value < 0) {
    result.successful = false;
    result.reason = "数值不能为负数! ";
} else {
    result.successful = true;
}
}
```

3. 参数的“动态”与“静态”

- **静态声明：** 如果你声明参数时不使用动态描述符，且不提供回调处理，外部修改虽然能成功改变节点内的参数值，但如果你的程序逻辑只在 `on_configure` 时读取一次参数，那么外部的 `set` 虽然成功了，但并不会对程序运行产生实际影响（这被称为“死参数”）。

4. 深度安全：ROS 2 Security (SROS2)

如果你是在做一个正式的机器人产品，担心黑客或恶意节点修改关键参数（比如避障距离），你可以开启 SROS2。

- **原理：** 基于 DDS 的安全插件。
- **功能：** 你可以为每个节点配置 **权限策略 (Policy)**。
- **限制：** 你可以规定“节点 B 只允许调用节点 A 的 `get_parameters` 服务，而不允许调用 `set_parameters` 服务”。
- 这是在**通信层**直接掐死权限，不符合策略的请求会被 DDS 直接丢弃。

总结

1. 默认状态：谁都能发 `set` 请求（全公开）。
2. 开发者防御：
 - 简单的用 `read_only`（不准改）。
 - 复杂的用 `on_set_parameters_callback`（检查后再决定准不准改）。
3. 系统级防御：使用 SROS2 进行数字签名和权限加密（最安全但也最复杂）。

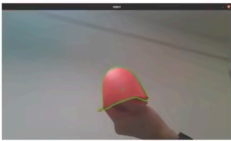
所以，在普通的 ROS 2 学习和开发中，我们通常默认大家都是“文明人”，主要靠回调函数来防止错误的参数输入。



参数在机器人中的应用

```
ros@ros2-ubuntu:~$ ros2 run learning_parameter param_object_detect
[INFO] [1649931235.10564269]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.109857956]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931235.292469832]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.2933306892]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931235.348545144]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.350809914]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931235.409711882]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.410729592]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931235.463662459]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.464533884]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931235.617919080]: [param_object_detect]: Receiving video frame
[INFO] [1649931235.620228353]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931240.370379108]: [param_object_detect]: Receiving video frame
[INFO] [1649931240.473848307]: [param_object_detect]: Get Red H Upper: 0, Lower: 0
[INFO] [1649931240.379486614]: [param_object_detect]: Receiving video frame
[INFO] [1649931240.380860340]: [param_object_detect]: Get Red H Upper: 180, Lower: 0
[INFO] [1649931240.435993381]: [param_object_detect]: Receiving video frame
[INFO] [1649931240.436989834]: [param_object_detect]: Get Red H Upper: 180, Lower: 0
[INFO] [1649931240.493333776]: [param_object_detect]: Receiving video frame
[INFO] [1649931240.492522872]: [param_object_detect]: Get Red H Upper: 180, Lower: 0
```

```
ros@ros2-ubuntu:~$ ros2 param set param_object_detect red_h_upper 180
Set parameter successful
```



```
class ImageSubscriber(Node):
def __init__(self, name):
super().__init__(name) # ros2节点父类初始化
self.sub = self.create_subscription(Image, # 创建订阅者对象(消息类型、话题名、订阅回调函数、队列长度)
"image_raw", self.listener_callback, 10)
self.cv_bridge = CvBridge() # 创建一个图像转换对象，用于opencv图像与ros的图像消息的互相转换
self.declare_parameter("red_h_upper", 0) # 创建一个参数，表示阈值上限
self.declare_parameter("red_h_lower", 0) # 创建一个参数，表示阈值下限

def listener_callback(self, image):
upper_red[0] = self.get_parameter("red_h_upper").get_parameter_value().integer_value # 读取阈值上限的参数值
lower_red[0] = self.get_parameter("red_h_lower").get_parameter_value().integer_value # 读取阈值下限的参数值
self.get_logger().info("Get Red H Upper: %d, Lower: %d" % (upper_red[0], lower_red[0])) # 通过日志打印读取到的参数值

# 图像从bgr颜色模型转换为nsv模型
# 图像二值化
# 图像中轮廓检测
# 去除一些轮廓面积太小的噪声
hav_img = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
mask_red = cv2.inRange(hav_img, lower_red, upper_red)
contours, hierarchy = cv2.findContours(mask_red, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
for cnt in contours:
if cnt.shape[0] < 150:
continue

(x, y, w, h) = cv2.boundingRect(cnt) # 得到苹果所在轮廓的左上角xy像素坐标及轮廓范围的宽和高
cv2.drawContours(image, [cnt], -1, (0, 255, 0), 2) # 将苹果的轮廓勾勒出来
cv2.circle(image, (int(x+w/2), int(y+h/2)), 5, (0, 255, 0), -1) # 将苹果的中心点画出来

cv2.imshow("object", image) # 使用opencv显示处理后的图像效果
cv2.waitKey(50)

def listener_callback(self, data):
self.get_logger().info("Receiving video frame") # 输出日志信息，提示已进入回调函数
image = self.cv_bridge.imgmsg_to_cv2(data, "bgr8") # 将ros的图像消息转化成opencv图像
self.object_detect(image) # 苹果检测

def main(args=None):
# ros2节点主入口main函数
rclpy.init(args=args) # ros2 python接口初始化
node = ImageSubscriber("param_object_detect") # 创建ros2节点对象并进行初始化
rclpy.spin(node) # 循环等待ros2退出
node.destroy_node() # 销毁节点对象
rclpy.shutdown() # 关闭ros2 python接口
```

param_object_detect.py

参数应用

```
$ ros2 run usb_cam usb_cam_node_exe
$ ros2 run learning_parameter param_object_detect
$ ros2 param set param_object_detect red_h_upper 180
```

课程示例中就是参数的一种应用：将阈值设置为参数，对阈值实时调整查看效果，而不用改一次编译一次这样调试。