# 1. ros是什么?

robot operating system

虽然叫做操作系统，但还是需要搭建在某个具体的操作系统之上的，功能强大的机器人软件库和工具集。

## 1.1 从ros1到ros2

ros1开发之初是为开发家用机器人：家用机器人无需担心网络连接不畅、算力不足等问题，而且只需要单独一台机器人工作。

ROS1中，所有节点的通信必须经过ros master主节点，一旦主节点崩掉，所有节点之间都无法正常运转。

机器人发展的需要：多机器人协同、在有干扰的情况下进行网络通信、满足实时性、跨平台...

ros2通信通过DDS实现，没有ros master，在同一个数据空间的两个节点之间可以直接通信。

ros1的只支持linux系统，ros2支持linux、windows、macos

ros2支持python3

...

# 2. ros2安装

小鱼一键安装

版本选择：ros2 humble desktop

安装完成后，安装ros2常见包：

```
sudo apt install ros-humble-desktop-full
```

使用ros2 humble时大部分缺失的ros2包都可以通过以下命令安装

```
apt install ros-humble-<package-name>
```

# 3. ros2的重要概念与机制

## 3.1 工作空间及编译

编译：`colcon build`

src 代码空间

build 编译空间 保存编译的中间文件

install 可执行文件和脚本的存放空间

log 存放日志文件

## 3.2 功能包

ros的一大目标就是提高代码复用率，为实现这个目标，我们把实现不同功能的代码划分到不同的功能包中。

创建功能包：

```
ros2 pkg create <package_name> --build-type <build-type> --
dependencies [dependencies1 dependencies2 ...] --node-name <node-
name>
```

build-type: c++选ament_cmake；python选ament_python。<package_name>是必选参数。

ros2生成的c++功能包的结构：

- package.xml
- CMakeLists.txt
- src/
- include/

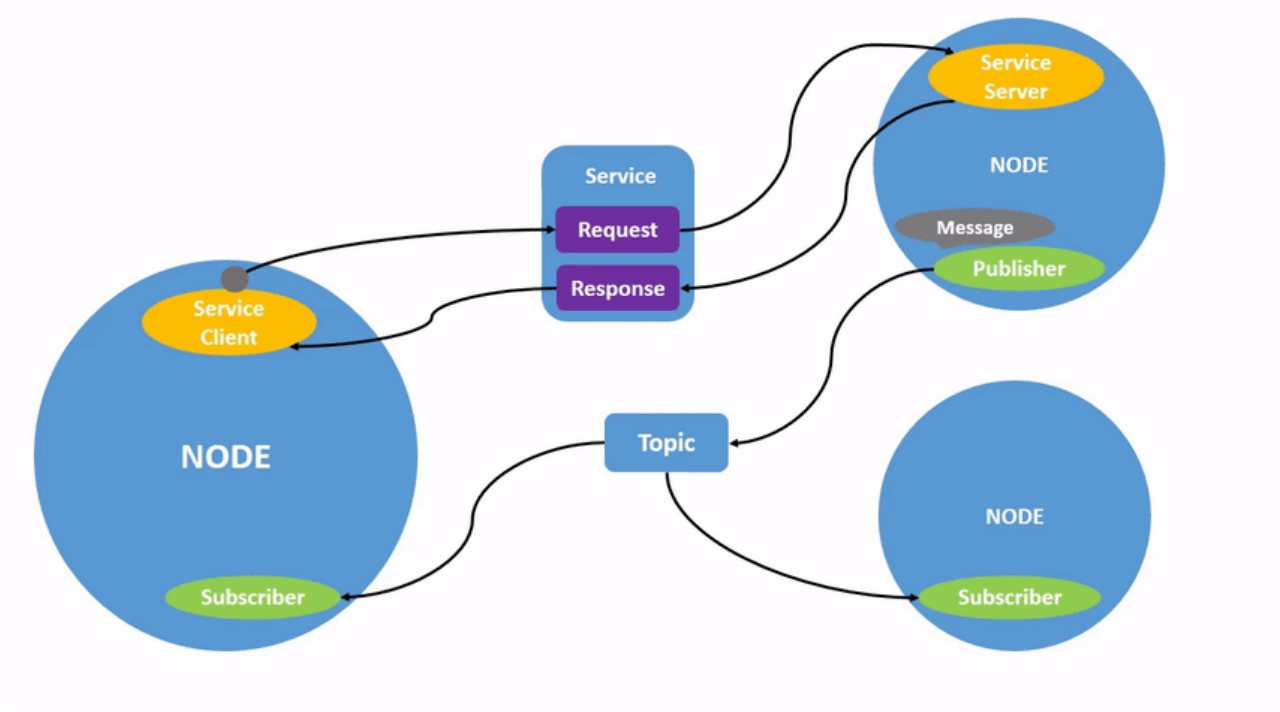
## 3.3 node

ros最重要的一个特点之一就是把代码模块化。一个节点负责一个单一的实现 比如相机节点、识别节点、解算节点...

如果想要一个机器人协调工作，就需要各个节点之间能够进行交互。

ros2为节点和节点之间的通信提供了四种方式：

topic、service、action、parameter




节点的使用：

- 编程接口初始化
- 创建节点并初始化
- 实现节点功能
- 销毁节点并关闭接口

以c++为例：

```
#include "rclcpp/rclcpp.hpp"

int main(int argc, char **argv)
{
    // 初始化rclcpp
    rclcpp::init(argc, argv);
    // 产生一个名为node_01的节点
    auto node = std::make_shared<rclcpp::Node>("node_01");
    // 打印
    RCLCPP_INFO(node->get_logger(), "Node_01 started.");
    // 运行节点，并检测退出信号 Ctrl+C
    rclcpp::spin(node);
```

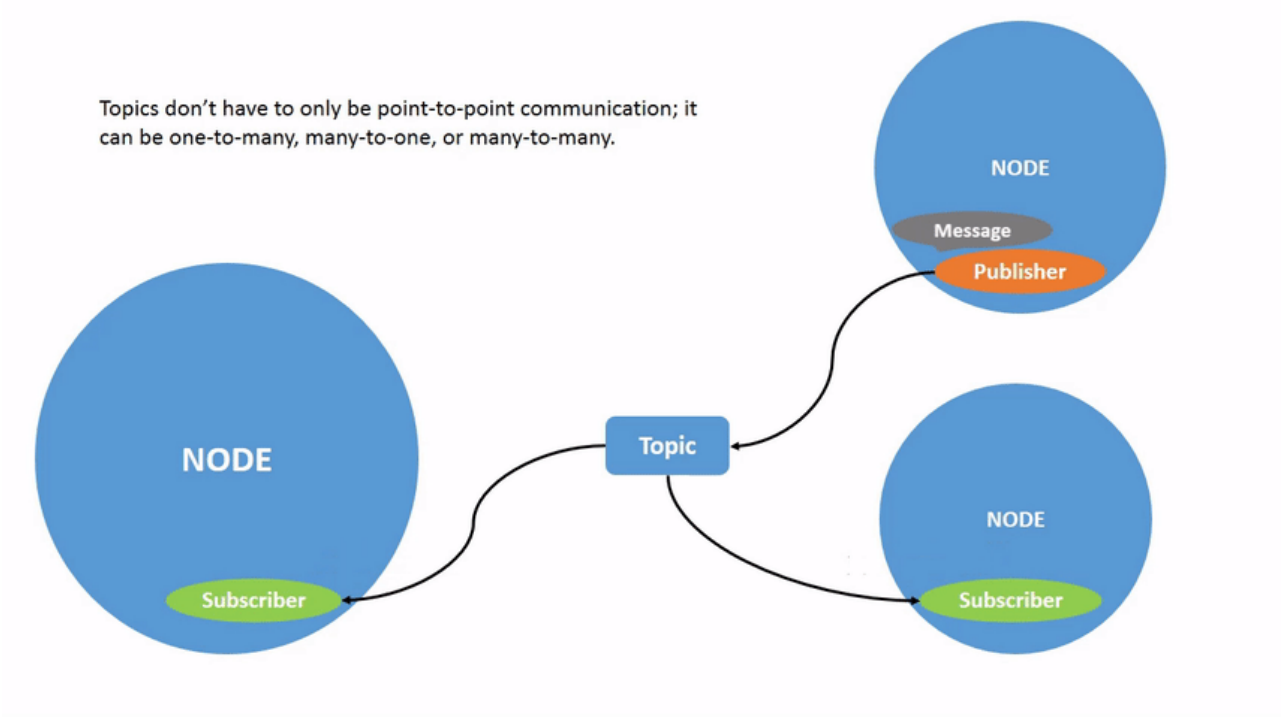
```
    // 停止运行
    rclcpp::shutdown();
    return 0;
}
```

大部分情况下我们创建自己的类，使这个类继承Node类，再进行ros开发。

```

#include "rclcpp/rclcpp.hpp"

//创建一个类节点，名字叫做Node03,继承自Node.
class Node03 : public rclcpp::Node
{

public:
    // 构造函数,有一个参数为节点名称
    Node03(std::string name) : Node(name)
    {
        // 打印
        RCLCPP_INFO(this->get_logger(), "Node started .");
    }

private:

};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    //产生一个node_03的节点
    auto node = std::make_shared<Node03>("node_03");
    // 运行节点，并检测退出信号
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

## 3.4 topic

发布订阅机制。话题只能由一个节点来发送，但可以多个节点订阅。发送方只知道自己在发送，但并不知道接收方是否接收到消息，属于单向通信。




创建一个发送方并发布消息

```
//create a publisher
rclcpp::Publisher::<message_type>::SharedPtr pub=this-
>create_publisher<message_type>(topic_name,qos);
//publish the message
pub->publish(msg)
```

创建一个订阅方接收消息

```
//create a subscription
rclcpp::Subscription<message_type>::SharedPtr
sub=this_>create_subscription<message_type>
(topic_name,qos,callback_function);
```

订阅方在回调函数中对接收到的消息进行处理

```
//callback function
void callback(const message_type::SharedPtr msg){
    RCLCPP_INFO(this->get_logger(),"receieved!");
}
```

qos是什么?

quality of service 服务质量。可以参考这一篇
https://mp.weixin.qq.com/s/J63fO4c_QIseLGQd5W2fAw

需要注意的是：两个节点的Qos设置不兼容将无法进行通信。

对于传感器数据等实时性要求很高的数据，Reliability这一参数应设置为best effort，因为偶尔丢弃几个包是可以接受的，但为确保数据被传送到而导致的重传延时是无法容忍的。

例：创建一个talker包和一个listener包，talker包中的talker_node节点负责发布信息，listener包中的listener_node节点负责订阅，并将接收到的消息输出。

（为方便起见，这两个节点只负责了订阅和发布其中的一个，实际上一个节点可以发布多个话题、接收多个话题，但同一个话题只能由一个节点来发布。）

下面是两个节点的完整代码：

创建包：

build-type: c++选ament_cmake；python选ament_python。

```
ros2 pkg create talker --build-type ament_cmake --dependencies
rclcpp std_msgs --node-name talker_node

ros2 pkg create listener --build-type ament_cmake --dependencies
rclcpp std_msgs --node-name listener_node
```

发布节点：

```
#include "rclcpp/rclcpp.hpp"
#include <memory>
#include <rclcpp/executors.hpp>
#include <rclcpp/publisher.hpp>
#include <rclcpp/qos.hpp>
```

```
#include <rclcpp/subscription.hpp>
#include <std_msgs/msg/float32.hpp>

/*
    创建一个类节点，名字叫做Node03,继承自Node.
*/
class Node03 : public rclcpp::Node
{

public:
    // 构造函数,有一个参数为节点名称
    Node03(std::string name) : Node(name)
    {
        // 打印一句
        RCLCPP_INFO(this->get_logger(), "%s started
.",name.c_str());
        f_.data=1;
        talker_pub_=this->create_publisher<std_msgs::msg::Float32>
("/cnt",rclcpp::SensorDataQoS());
        while(true){
          talker_pub_->publish(f_);
          f_.data+=1;
          RCLCPP_INFO(this->get_logger(), " %s publish :
%f",name.c_str(),f_.data);
        }
    }
private:
  rclcpp::Publisher<std_msgs::msg::Float32>::SharedPtr
talker_pub_;
  std_msgs::msg::Float32 f_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    /*产生一个node_03的节点*/
    auto talker_node = std::make_shared<Node03>("talker_node");
    /* 运行节点，并检测退出信号*/
    rclcpp::spin(talker_node);
    rclcpp::shutdown();
```

```
    return 0;
}
```

订阅节点:

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/float32.hpp"
#include <rclcpp/qos.hpp>
#include <std_msgs/msg/detail/float32__struct.hpp>

class TopicSubscribe01 : public rclcpp::Node {
public:
  TopicSubscribe01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "%s started.", name.c_str());
    // 创建一个订阅者订阅话题
    sub_ = this->create_subscription<std_msgs::msg::Float32>(
        "/cnt", rclcpp::SensorDataQoS(),
        std::bind(&TopicSubscribe01::callback, this,
                  std::placeholders::_1));
  }

private:
  // 声明一个订阅者
  rclcpp::Subscription<std_msgs::msg::Float32>::SharedPtr sub_;
  // 收到话题数据的回调函数
  void callback(const std_msgs::msg::Float32::SharedPtr msg) {
    RCLCPP_INFO(this->get_logger(), " I received %f", msg->data);
  }
};

int main(int argc, char **argv) {
  rclcpp::init(argc, argv);
  //创建对应节点的共享指针对象
  auto node = std::make_shared<TopicSubscribe01>("listener_node");
  // 运行节点，并检测退出信号
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
```

```
}
```

根据实际情况需要，CMakeLists.txt和package.xml文件也需要作出相应修改。

ros2的运行命令：

```
ros2 run <package-name> <excutable-name>
```

编译完成后，运行：

```
source install/setup.bash
ros2 run talker talker_node
```

在新终端：

```
source install/setup.bash
ros2 run listener listener_node
```

通过rqt可以查看各个节点之间话题的订阅与发布情况。

```
rqt
```

通过以下命令查看所有话题名称

```
ros2 topic list
```

通过以下命令查看某一话题发送的内容

```
ros2 topic echo <topic-name>
```

通过以下命令查看某一话题的发布频率

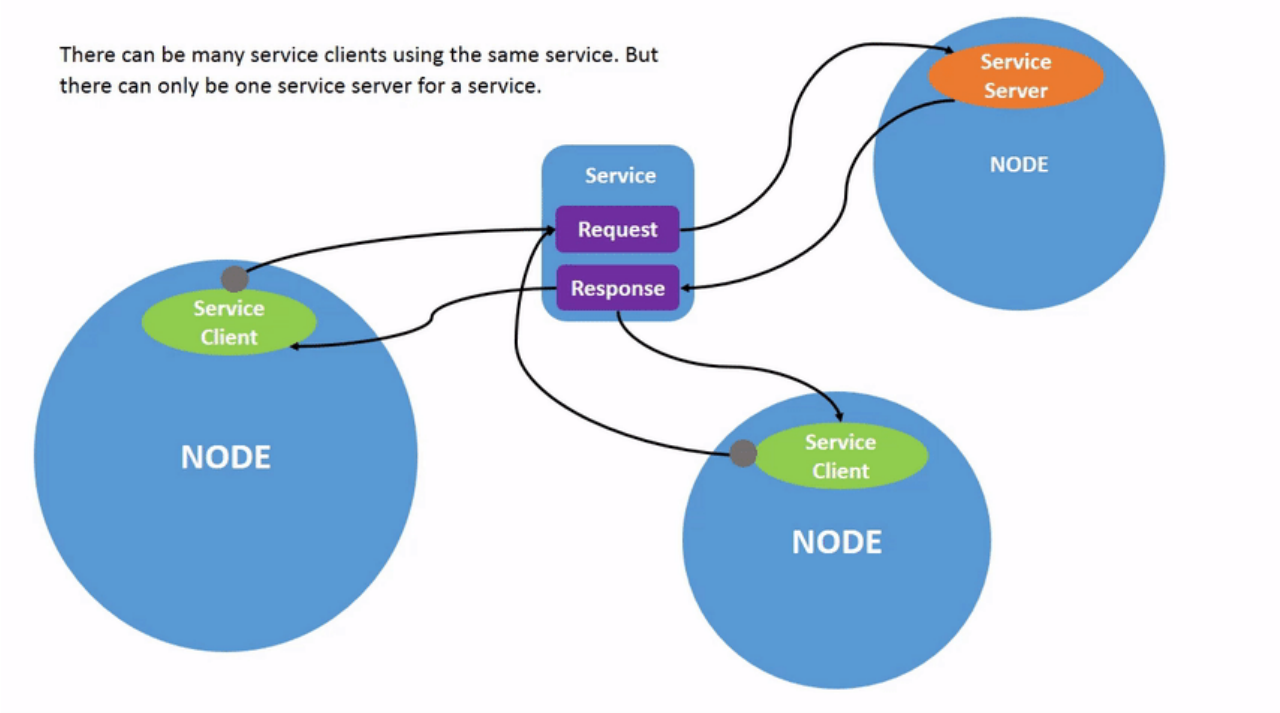
```
ros2 topic hz <topic-name>
```

## 3.5 service

服务与话题不同，话题是一种单向通讯模式，一个节点发布可以被一个或多个订阅者使用的信息，而话题的发布方并不知道订阅方是否接收到了消息。

服务使用的是客户端 /服务器模型，客户端发布请求 request，服务器端提供服务响应 response，服务是一种双向通信。

根据请求数据的不同，得到不同的应答结果。




我们以ros2的example_interfaces中的add_two_ints.srv为例，这个srv文件内容如下，客户端向服务器端发送a和b，服务器端返回它们的和：

```
int64 a
int64 b
---
int64 sum
```

创建包的部分请参考topic一节中，只是需要添加一个依赖 `example_interfaces`

创建客户端节点：

```
#include "example_interfaces/srv/add_two_ints.hpp"
#include "rclcpp/rclcpp.hpp"
#include <rclcpp/client.hpp>
#include <rclcpp/logging.hpp>

class ServiceClient01 : public rclcpp::Node {
public:
  // 构造函数,有一个参数为节点名称
  ServiceClient01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(), "client node started .");
    // 创建客户端
```

```
    client_ = this-
>create_client<example_interfaces::srv::AddTwoInts>(
        "add_two_ints_srv");
  }
  void send_request(int a, int b) {
    RCLCPP_INFO(this->get_logger(), "计算%d+%d", a, b);

    // 1.等待服务端上线
    // be sure that the server had began to work.
    while (!client_->wait_for_service(std::chrono::seconds(1))) {
      //等待时检测rclcpp的状态
      if (!rclcpp::ok()) {
        RCLCPP_ERROR(this->get_logger(), "等待服务的过程中被打
断...");
        return;
      }
      RCLCPP_INFO(this->get_logger(), "等待服务端上线中");
    }

    // 2.构造请求
    auto request =

std::make_shared<example_interfaces::srv::AddTwoInts_Request>();
    request->a = a;
    request->b = b;
    RCLCPP_INFO(this->get_logger(), "now!");
    // 3.发送异步请求，然后等待返回，返回时调用回调函数
    client_->async_send_request(request,

std::bind(&ServiceClient01::result_callback_,
                                            this,
std::placeholders::_1));
  };

private:
  // 声明客户端
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr
client_;
  void result_callback_(
```

```
rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedFuture
          result_future) {
    auto response = result_future.get();
    RCLCPP_INFO(this->get_logger(), "计算结果：%ld", response->sum);
  }
};

int main(int argc, char **argv) {
  rclcpp::init(argc, argv);
  //创建对应节点的共享指针对象
  auto node = std::make_shared<ServiceClient01>("client_node");
  // 运行节点，并检测退出信号
  //增加这一行，node->send_request(5, 6);，计算5+6结果
  node->send_request(5, 6);
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

创建服务器端节点：

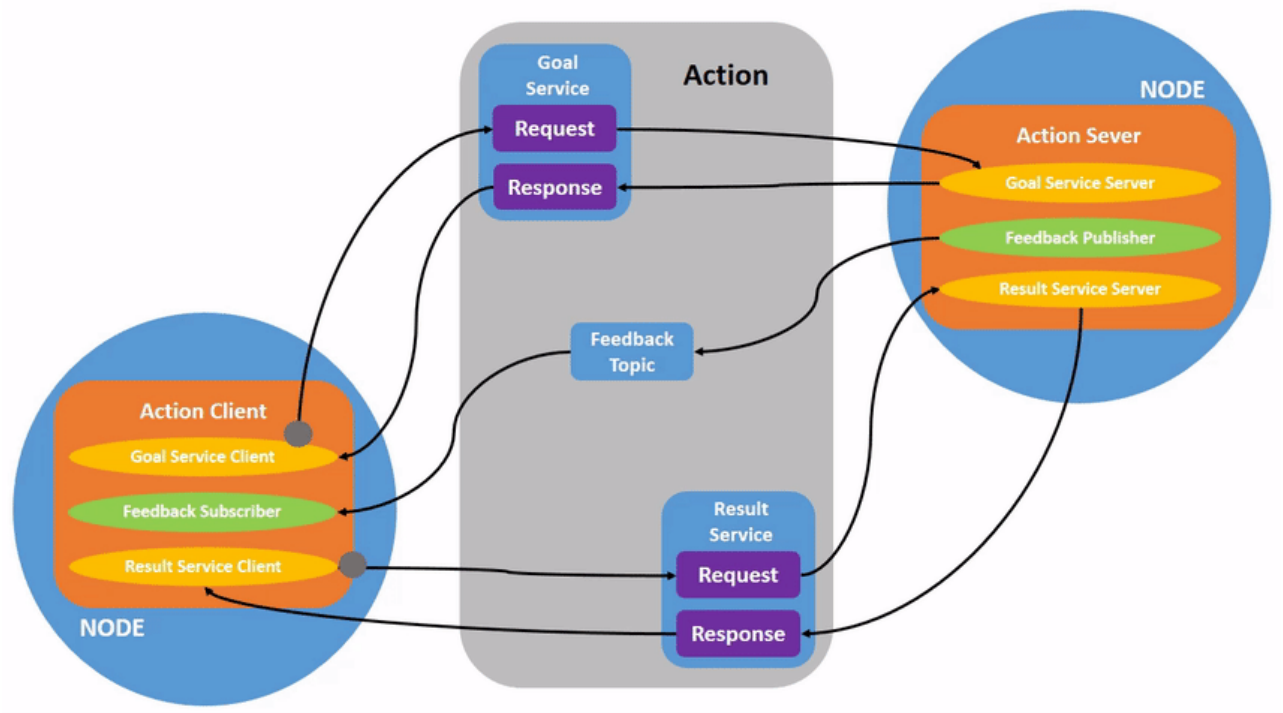
```
#include "example_interfaces/srv/add_two_ints.hpp"
#include "rclcpp/rclcpp.hpp"
#include <rclcpp/logging.hpp>
#include "rclcpp/rclcpp.hpp"

class ServiceServer01 : public rclcpp::Node {
public:
  ServiceServer01(std::string name) : Node(name) {
    RCLCPP_INFO(this->get_logger(),"service node started .");
    // 创建服务
    add_ints_server_ =
      this->create_service<example_interfaces::srv::AddTwoInts>(
        "add_two_ints_srv",
        std::bind(&ServiceServer01::handle_add_two_ints, this,
                  std::placeholders::_1, std::placeholders::_2));
  }

private:
```

```
  // 声明一个服务
  rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr
add_ints_server_;

  // 收到请求的处理函数
  void handle_add_two_ints(
    const
std::shared_ptr<example_interfaces::srv::AddTwoInts::Request>
request,
    std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>
response) {
    RCLCPP_INFO(this->get_logger(), "收到a: %ld b: %ld", request-
>a,
                request->b);
    response->sum = request->a + request->b;
  };
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<ServiceServer01>("server_node");
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

运行后可以观察到终端输出结果。

## 3.6 action

action建立在话题和服务的基础之上，由一个任务目标（Goal，服务），一个执行结果（Result，服务），周期数据反馈（Feedback，话题）组成。action是可抢占式的，由于需要执行一段时间，比如执行过程中你不想跑了，那可以随时发送取消指令，动作终止，如果执行过程中发送一个新的action目标，则会直接中断上一个目标开始执行最新的任务目标。




## 3.7 parameter

在实际应用过程中，存在需要随时调整的参数，如果把这些参数全部写进程序，调参会变得非常麻烦。为了解决这个问题，ros2提供了parameter这一通信机制。

ros2的参数是由key-value对构成的，参数名是key，参数值是value。ros2支持的参数类型包括：string、float64、int64、bool和他们的数组以及byte[]字节数组。

我们可以使用小乌龟例程来感受一下如何使用命令行来查看、设置参数。

```
source /opt/ros/humble/setup.bash
ros2 run turtlesim turtlesim_node
```

再开一个终端

```
source /opt/ros/humble/setup.bash
//和参数有关的操作
//查看所有参数
ros2 param list
//获取参数值
ros2 param get <node-name> <param-name>
//设置参数值（更改后的参数值不会保存）
ros2 param set <node-name> <param-name>
//查看参数描述信息
ros2 param describe <node-name> <param-name>
//保存参数
```

```
ros2 param dump <node-name>
//节点启动后加载参数
ros2 param load <node-name> <param.yaml path>
//节点启动时加载参数
ros2 run <package_name> <executable_name> --ros-args --params-file
<file_name>
```

上面是体验如何使用命令行进行参数的设置和查看，我们来看一下在程序中参数是如何声明和使用的。

```
//声明参数
this->declare_parameter("thresh",19);
//读取参数值
thresh=this->get_parameter("thresh").as_double();
//设置参数
this->set_parameter("thresh",60);
```

## 3.8 interfaces

ROS如何实现跨平台、跨设备之间的数据收发，依赖于消息接口文件。

同一个话题的发送者和所有接收者，必须使用相同的消息接口，我们在发布的话题中需要指定消息接口类型。

ros2提供了一些标准的消息类型 比如std_msgs包中的Float32、Uint8；sensor_msgs中的Image等等，但是这些已经规定好的消息类型可能有时候无法适配我们的需求，比如如果我们想通过一个话题同时发布位姿和装甲板编号该怎么办？ros2中并没有一个这样的消息类型供我们使用。

但ros2提供了自定义消息接口，让我们可以根据自己的需要，以标准消息接口为基础，定制自己需要的消息接口。

例：

我们创建一个interfaces包，这个包并不需要节点，依赖中的rosidl_default_generators是必需的，其他的根据需要而定。

rosidl_default_generators是一个转换包，它将我们自定义的这些.msg、.action、.srv文件转换成.hpp/.h文件，对于python而言则转换为.py文件

```
ros2 pkg create interfaces --build-type ament_cmake --dependencies
std_msgs rosidl_default_generators geometry_msgs
```

在msg文件夹下创建一个Armor.msg，假设我们需要通过这个消息类型发布装甲板矩形框左上和右下两点的图像坐标系的坐标和装甲板编号，我们可以在这个.msg文件里这样写：

```
uint8 id
float32 x1
float32 y1
float32 x2
float32 y2
```

如果我们需要通过一个话题发布一组识别到的装甲板，可以使用我们自定义好的消息类型。在msg文件夹下创建Armors.msg文件。

```
Armor[] armors
```

和话题一样，我们也可以自定义service类型，

例：实际应用中我们需要切换视觉模式 打符or自瞄?

在interfaces/srv文件夹下创建.srv文件。

请求和应答的数据通过---分割开。

```
# 请求数据
uint8 mode
---
# 应答数据
bool success
```

对于action:

在interfaces/action文件夹下创建.action文件

```
# Goal: 要移动的距离
float32 distance
---
# Result: 最终的位置
float32 pose
---
# Feedback: 中间反馈的位置和状态
float32 pose
uint32 status
uint32 STATUS_MOVEING = 3
uint32 STATUS_STOP = 4
```

CMakeLists.txt中需要增加：

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Armor.msg"
  "msg/Armors.msg"
  "srv/SetMode.srv"
  "action/Robot.action"
  DEPENDENCIES
    std_msgs
    geometry_msgs
)
```

package.xml中需要添加：

```
  <member_of_group>rosidl_interface_packages</member_of_group>
```

最终这个文件夹的结构如下：

```
├── action
│   └── Robot.action
├── CMakeLists.txt
├── msg
│   ├── Armor.msg
│   └── Armors.msg
├── package.xml
└── srv
    └── SetMode.srv
```

单独编译interfaces一个包：

```
colcon build --packages-select interfaces
```

如果之后创建包时，需要使用到interfaces作为依赖，则在创建包命令中的dependencies参数中添加这个包。对于已经创建的包，如果要使用，需要在package.xml中添加:

```
<depend>interfaces</depend>
```

当编译通过，所有的环境配置问题都已经解决后，我们使用这个包中的自定义消息类型，以发布一个Armor.msg接口类型的消息为例：

```
//create a publisher
rclcpp::Publisher::<interfaces::msg::Armor>::SharedPtr pub=this-
>create_publisher<interfaces::msg::Armor>(topic_name,qos);

//publish the message
interfaces::msg::Armor armor_msg;
armor_msg.id=3;
armor_msg.x1=30;
armor_msg.x2=100;
armor_msg.y1=90;
armor_msg.y2=160;
pub->publish(msg);
```