# C++ ROS2 的基本使用



# VSCode

vscode 官网

> https://code.visualstudio.com/Download

安装

```
sudo dpkg -i code_1.58.0-1625728071_amd64.deb
```

## 配置环境

安装插件:

中文语言包、c++插件、Cmake、Cmake Tools、IntelliCode

Msg Language Support、 URDF、 ROS、vscode-icons

---

# 工作空间

工作空间就是写代码的地方，存放多个功能包，其实就是一个文件夹

## 创建工作空间

```
mkdir -p dev_ws/src
cd dev_ws/src
```

---

# 功能包

功能包是用来存放cpp和python文件的地方，也就是存放节点的地方，每个功能包对应不同的功能。

## 创建功能包

```
ros2 pkg create <package-name>  --build-type
{cmake,ament_cmake,ament_python}  --dependencies <依赖名字>
```

pkg：表示功能包相关的功能

create：表示创建功能包

build-type：表示新创建的功能包是C++还是Python的，如果使用C++或者C，那这里就跟ament_cmake，如果使用 Python，就跟ament_python

package_name：新建功能包的名字

---

# 编译

## 安装colcon

colcon是用来编译代码的一个工具

```
sudo apt-get install python3-colcon-common-extensions
```

在功能包中写完代码之后需要编译和配置环境变量才能运行文件。

## 编译和配置环境变量

```
cd ~/dev_ws # 编译前要切换到dev_ws
colcon build  # 编译所有功能包
source install/setup.bash # 配置环境变量
```

---

# 节点 (node)

简单来说就是一个人的手、脚、眼睛、鼻子.....，每个节点对应不同的功能，所有的节点组成了一个人。

## 创建功能包

```
ros2 pkg create node_demo --build-type ament_cmake --dependencies rclcpp
```

## 创建节点

在node_demo/src下创建node_01.cpp文件

## 编写代码

(面向过程)

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

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv); // 初始化rclcpp
    auto node = std::make_shared<rclcpp::Node>("node_01"); // 创建node_01节点
    RCLCPP_INFO(node->get_logger(), "node_01 is running"); // 打印输出
    rclcpp::spin(node); // 循环运行节点
    rclcpp::shutdown(); // 停止运行
    return 0;
}
```

(面向对象)

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

class Node02 : public rclcpp::Node
{
public:
    Node02(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "node02 is running.");
    }
};

```

```cpp
int main(int argc, char **argv)
{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<Node02>("node02");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

一般流程:
1.包含头文件
2.初始化ROS2
3.自定义节点类
4.循环节点
5.释放资源

## 修改CmakeLists

编写完代码之后，需要修改CmakeLists.txt。

在CmakeLists最后一行添加:

```
add_executable(node_01 src/node_01.cpp)
ament_target_dependencies(node_01 rclcpp)

install(TARGETS
  node_01
  DESTINATION lib/${PROJECT_NAME}
)
```

add_executable 表示添加可执行文件
ament_target_dependencies 表示该节点依赖的文件
install 表示将该文件安装到`install`目录，使得编译器编译该文件

## 编译和配置环境变量

```
colcon build
source install/setup.bash
```

## 运行节点

```
ros2 run node_demo node_01
```

## 节点命令行操作

```
ros2 node list               # 查看节点列表
ros2 node info <node_name>   # 查看节点信息
```

# 话题 (topic)

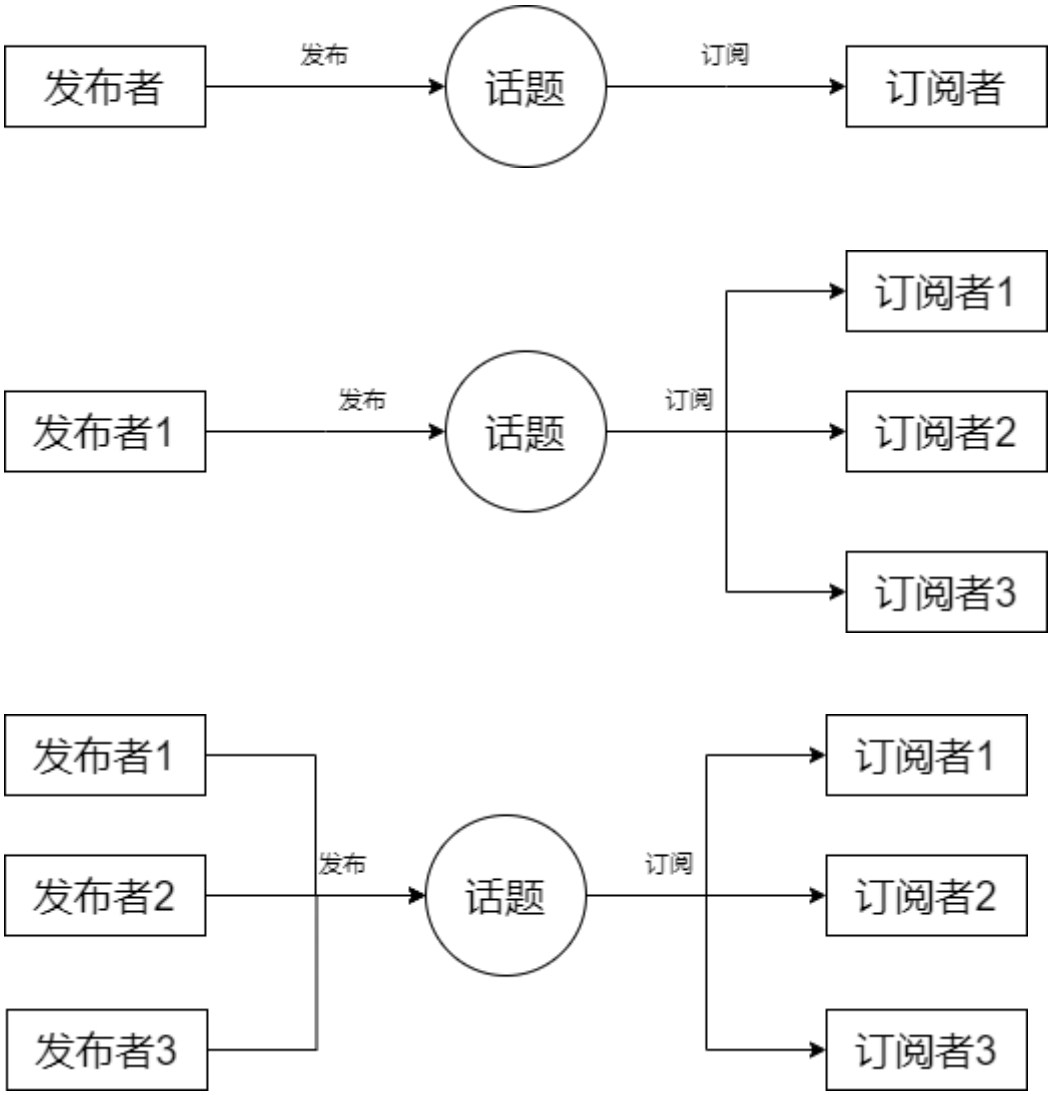
话题通信:是一种单向通信模型，发布者发布信息，订阅者接收信息。

每个话题都有一个特定的名称。

比如：电视台发布信息；新闻周刊发布新闻。

## 订阅发布模型

可以是1对1，1对n, n对n




## 创建功能包

```
ros2 pkg create topic_demo --build-type ament_cmake --dependencies rclcpp
```

## 编写代码

创建节点

在 topic_demo/src 下创建 pub.cpp 和 sub.cpp 文件

代码实现（发布者）

```cpp
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class Pub : public rclcpp::Node
{
public:
    Pub(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "node is running.");

        // 2.创建发布者
        pub = this->create_publisher<std_msgs::msg::String>("name", 10);

        // 创建定时器发布信息
        timer_ = this->create_wall_timer(std::chrono::milliseconds(1000), std::bind(&Pub::send_msg, this));
    }

private:
    // 1. 声明发布者
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr pub;

    // 3.发布信息
    void send_msg()
    {
        // 创建消息
        std_msgs::msg::String name;
        name.data = "zhang san";
        // 日志打印
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", name.data.c_str());
        // 发布消息
        pub->publish(name);
    }

    // 4. 声明定时器
    rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    auto node = std::make_shared<Pub>("Pub");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

代码实现（订阅者）

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

```cpp
#include "std_msgs/msg/string.hpp"

class Sub : public rclcpp::Node
{
public:
    Sub(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "node is running.");

        // 3. 创建订阅者
        sub = this->create_subscription<std_msgs::msg::String>("name", 10, std::bind(&Sub::sub_callback, this, std::placeholders::_1));
    }

private:
    // 1.声明订阅者
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub;

    // 2.订阅者回调函数
    void sub_callback(const std_msgs::msg::String::SharedPtr name)
    {
        RCLCPP_INFO(this->get_logger(), "Receiving, '%s'", name->data.c_str());
    }
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    auto node = std::make_shared<Sub>("Pub");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

## 修改CmakeLists

```
find_package(std_msgs REQUIRED)

add_executable(pub src/pub.cpp)
ament_target_dependencies(
  pub
  "rclcpp"
  "std_msgs"
)

add_executable(sub src/sub.cpp)
ament_target_dependencies(
  sub
  "rclcpp"
  "std_msgs"
)

```

```
install(TARGETS
  pub
  sub
  DESTINATION lib/${PROJECT_NAME}
)
```

## 修改package

```
<depend>std_msgs</depend>
```

## 编译和配置环境变量

```
colcon build --packages-select topic_demo
source install/setup.bash
```

## 运行

```
ros2 run topic_demo pub
ros2 run topic_demo sub
```

---

# 通信接口 (interfaces)

在传输数据的时候，会涉及到数据载体，可以用ROS2自带的一些数据载体，也可以根据实际需要创建对应的数据载体，这些载体成为接口(interfaces)。通信接口与编程语言无关，自定义接口的数据类型由ROS2中基本的数据类型组成。

## ROS2基本数据类型

```
bool
byte
char
float32,float64
int8,uint8
int16,uint16
int32,uint32
int64,uint64
string
```

## ROS2接口文件

ROS2有三种接口文件，如下：

1.msg文件

用于自定义话题通信的数据载体，如:

```
string name
int64 age
```

2.srv文件

用于自定义服务通信的数据载体，如:

```
# request
int64 num1
int64 nun2
---
# response
int64 sum
```

3.action文件

用于自定义动作通信的数据载体，如:

```
int32 order  # goal
---
int32[] sequence # result
---
int32[] partial_sequence # feedback
```

# 自定义接口(话题)

1.创建功能包

```
ros2 pkg create interfaces_demo --build-type ament_cmake
```

**(这里 --build-type 必须是ament_cmake)**

2.创建msg文件夹

新建PersonInfo.msg 文件，每个单词首字母大写。

```
string name
int64 age
```

3.修改CMakeLists

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/PersonInfo.msg"
  DEPENDENCIES
)
```

4.修改package

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

5.编译

```
colcon build --packages-select interfaces_demo
```

6.验证

```
ros2 interface show interfaces_demo/msg/PersonInfo
```

7.使用该自定义接口

修改topic_demo文件

修改CmakeLists

```
find_package(interfaces_demo REQUIRED)

add_executable(pub src/pub.cpp)
ament_target_dependencies(
  pub
  "rclcpp"
  "std_msgs"
  "interfaces_demo"
)

add_executable(sub src/sub.cpp)
ament_target_dependencies(
  sub
  "rclcpp"
  "std_msgs"
  "interfaces_demo"
)
```

修改package

```
<depend>interfaces_demo</depend>
```

sub.cpp 和 pub.cpp

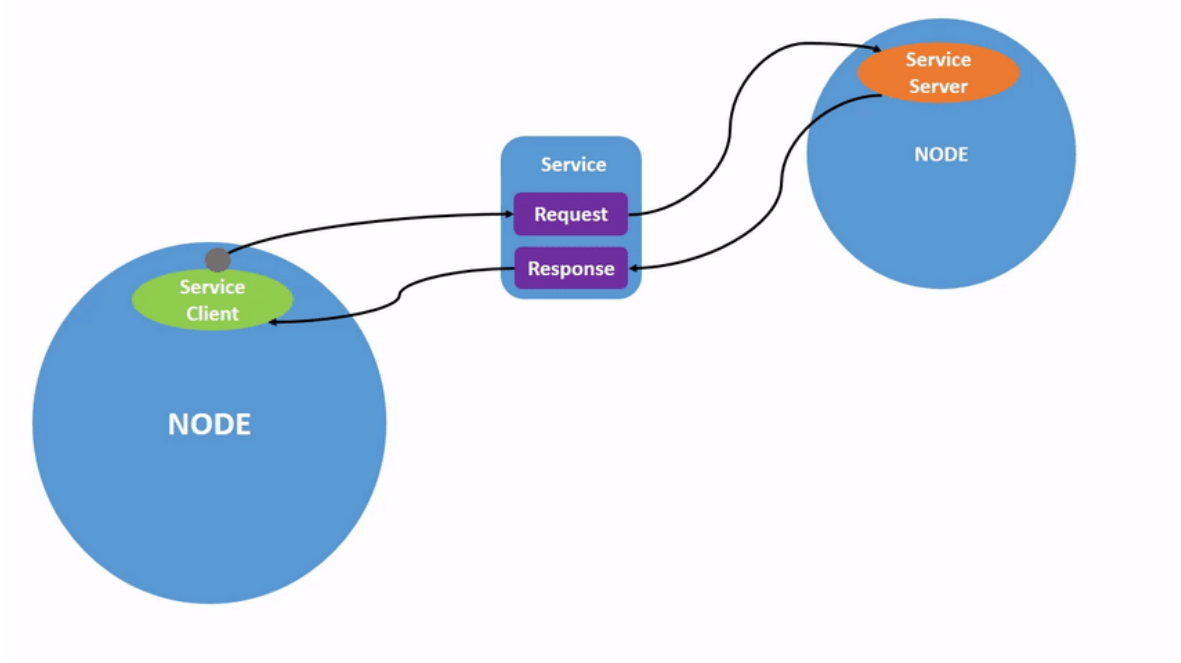
```
#include "interfaces_demo/msg/person_info.hpp"
```

---

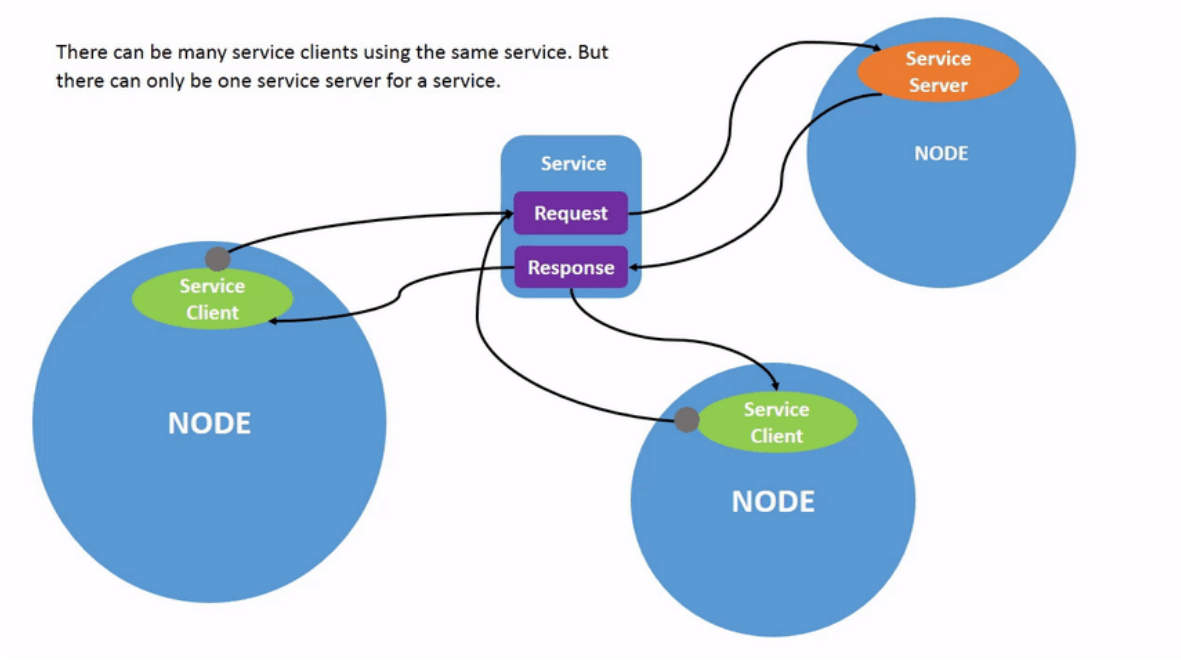
# 服务 (service)

服务通信:是一种基于请求响应的通信模型，在通信双方中，客户端发送请求数据到服务端，服务端响应结果给客户端。

## 服务端/客户端模型




多对多




## 自定义服务接口

创建 srv文件夹 和 AddTwoInt.srv 文件

```
int64 num1
int64 num2
---
int64 sum
```

## 创建功能包

```
ros2 pkg create service_demo --build-type ament_cmake --dependencies rclcpp
interfaces_demo
```

## 编写代码

(服务端)

```
#include "rclcpp/rclcpp.hpp"
#include "interfaces_demo/srv/add_two_int.hpp"

class Server : public rclcpp::Node
{
public:
    Server(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "node is running.");

        // 3.创建服务端
        server = this->create_service<interfaces_demo::srv::AddTwoInt>
("service", std::bind(&Server::server_callback, this,
std::placeholders::_1, std::placeholders::_2));
    }

private:
    // 1.声明服务端
    rclcpp::Service<interfaces_demo::srv::AddTwoInt>::SharedPtr server;

    // 2.服务端回调函数
    void server_callback(const
interfaces_demo::srv::AddTwoInt::Request::SharedPtr request,
                         const
interfaces_demo::srv::AddTwoInt::Response::SharedPtr response)
    {
        RCLCPP_INFO(this->get_logger(), "Receiving a request");
        response->sum = request->num1 + request->num2;
    }
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    auto node = std::make_shared<Server>("server");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

(客户端)

```
#include "rclcpp/rclcpp.hpp"
#include "interfaces_demo/srv/add_two_int.hpp"

class Client : public rclcpp::Node
{
public:
```

```
    Client(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "node is running");

        // 3.创建客户端
        client = this->create_client<interfaces_demo::srv::AddTwoInt>
("service");

        // 6.发送请求
        timer_ = this->create_wall_timer(std::chrono::seconds(1),
std::bind(&Client::send_request, this));
    }

private:
    // 1.声明客户端
    rclcpp::Client<interfaces_demo::srv::AddTwoInt>::SharedPtr client;

    // 2.客户端回调函数
    void
client_callback(rclcpp::Client<interfaces_demo::srv::AddTwoInt>::SharedFutu
re response)
    {
        RCLCPP_INFO(this->get_logger(), "Receiving a response");
        auto result = response.get();
        RCLCPP_INFO(this->get_logger(), "sum: %d", result->sum);
    }

    // 4.发送请求函数
    void send_request()
    {
        while (!client->wait_for_service(std::chrono::seconds(2)))
        {
            RCLCPP_WARN(this->get_logger(), "等待服务端上线...");
        }

        // 构造 request
        auto request =
std::make_shared<interfaces_demo::srv::AddTwoInt_Request>();
        request->num1 = 10;
        request->num2 = 20;

        // 发送异步数据
        client->async_send_request(request,
std::bind(&Client::client_callback, this, std::placeholders::_1));
    }

    // 5.声明定时器
    rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    auto node = std::make_shared<Client>("client");

    rclcpp::spin(node);

```

```
    rclcpp::shutdown();

    return 0;
}
```

## 修改CmakeLists

```
find_package(rosidl_default_generators REQUIRED)

add_executable(server src/server.cpp)
ament_target_dependencies(
  server
  "rclcpp"
  "interfaces_demo"
)

add_executable(client src/client.cpp)
ament_target_dependencies(
  client
  "rclcpp"
  "interfaces_demo"
)

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME}
)
```

## 修改package

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

## 命令行操作

```
ros2 service list                    # 查看服务列表
ros2 service type <service_name>     # 查看服务数据类型
ros2 service call <service_name> <service_type> <service_data>    # 发送服务请求
```

---

# 动作 (action)

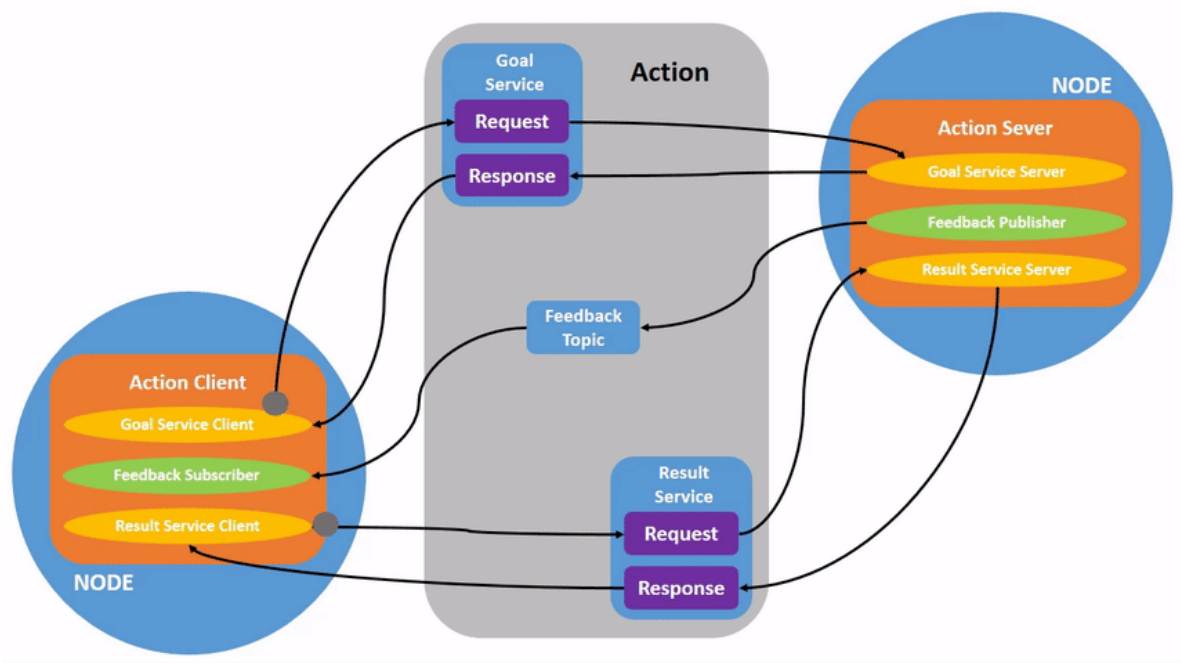
动作通信:是一种带有连续反馈的通信模型，在通信双方中，客户端发送清求数据到服务端，服务端响应结果给客户端，但是在服务端接收到请求到产生最终响应的过程中，会发送连续的反馈信息到客户端。

## 客户端/服务器模型

和服务通信类似，客户端发出指令，服务端接收指令处理后返回。

由三部分组成: 目标，结果，反馈。

底层是基于话题和服务的，由三个服务和两个话题组成。三个服务分别是：1.目标传递服务 2.结果传递服务 3.取消执行服务 两个话题：1.反馈话题（服务发布，客户端订阅） 2.状态话题（服务端发布，客户端订阅）




## 一对多通信

只能有一个服务端，但是可以有多个客户端。

（求1~N的和，返回最终结果，并且每次返回累加进度）

## 自定义接口(动作)

(1) 创建 action文件夹 和 Progress.action文件

```
int64 num
---
int64 sum
---
float64 progress
```

(2)修改CmakeLists

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/PersonInfo.msg"
  "srv/AddTwoInt.srv"
  "action/Progress.action"
  DEPENDENCIES
)
```

(3)修改package

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

(4)编译

```
colcon build --packages-select interfaces_demo
```

(5)验证

```
ros2 interface show interfaces_demo/action/Progress
```

## 创建功能包

```
ros2 pkg create action_demo --build-type ament_cmake --dependencies rclcpp
rclcpp_action interfaces_demo
```

## 编写代码

(服务端)

```
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "interfaces_demo/action/progress.hpp"

class ActionServer : public rclcpp::Node
{
public:
    ActionServer(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "ActionServer is running.");

        server =
rclcpp_action::create_server<interfaces_demo::action::Progress>(
            this,
            "get_sum",
            std::bind(&ActionServer::handle_goal, this,
std::placeholders::_1, std::placeholders::_2),
            std::bind(&ActionServer::handle_cancel, this,
std::placeholders::_1),
            std::bind(&ActionServer::handle_accepted, this,
std::placeholders::_1));
    }

private:
    // 声明动作服务端
    rclcpp_action::Server<interfaces_demo::action::Progress>::SharedPtr
server;

    rclcpp_action::GoalResponse handle_goal(const rclcpp_action::GoalUUID
&uuid, std::shared_ptr<const interfaces_demo::action::Progress::Goal> goal)
    {
```

```
        if (goal->num <= 1)
        {
            RCLCPP_INFO(this->get_logger(), "数据必须大于1!");
            return rclcpp_action::GoalResponse::REJECT;
        }
        RCLCPP_INFO(this->get_logger(), "成功接收数据");
        return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
    }

    rclcpp_action::CancelResponse
handle_cancel(std::shared_ptr<rclcpp_action::ServerGoalHandle<interfaces_de
mo::action::Progress>> goal_handle)
    {
        RCLCPP_INFO(this->get_logger(), "取消请求");
        return rclcpp_action::CancelResponse::ACCEPT;
    }

    void
execute(std::shared_ptr<rclcpp_action::ServerGoalHandle<interfaces_demo::ac
tion::Progress>> goal_handle)
    {
        auto feedback =
std::make_shared<interfaces_demo::action::Progress::Feedback>();
        auto result =
std::make_shared<interfaces_demo::action::Progress::Result>();

        // 生成连续反馈给客户端
        int num = goal_handle->get_goal()->num;
        int sum = 0;
        rclcpp::Rate rate(1);
        for (int i = 1; i <= num; i++)
        {
            sum += i;
            double progress = i / (double)num;
            feedback->progress = progress;
            goal_handle->publish_feedback(feedback);
            RCLCPP_INFO(this->get_logger(), "连续反馈中，进度%.2f", progress);

            if (goal_handle->is_canceling())
            {
                result->sum = sum;
                goal_handle->canceled(result);
                RCLCPP_INFO(this->get_logger(), "取消请求");
                return;
            }

            rate.sleep();
        }

        // 生成最终相应结果
        if (rclcpp::ok())
        {
            result->sum = sum;
            goal_handle->succeed(result);
        }
    }

    // std::function<void (std::shared_ptr<ServerGoalHandle<ActionT>>)>
```

```
    void handle_accepted(std::shared_ptr<rclcpp_action::ServerGoalHandle<interfaces_demo::action::Progress>> goal_handle)
    {
        // 新建线程处理反馈
        std::thread(std::bind(&ActionServer::execute, this, goal_handle)).detach();
    }
};

int main(int argc, char **argv)
{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<ActionServer>("ActionServer");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

(客户端)

```
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "interfaces_demo/action/progress.hpp"

class ActionClient : public rclcpp::Node
{
public:
    ActionClient(std::string name) : Node(name)
    {
        RCLCPP_INFO(this->get_logger(), "ActionClient is running.");

        client = rclcpp_action::create_client<interfaces_demo::action::Progress>(this, "get_sum");

        send_goal(10);
    }

private:
    // 声明动作客户端
    rclcpp_action::Client<interfaces_demo::action::Progress>::SharedPtr client;

    void send_goal(int num)
    {
        // 连接服务端
        if (!client->wait_for_action_server(std::chrono::seconds(10)))
        {
            RCLCPP_ERROR(this->get_logger(), "服务器连接失败!");
            return;
        }
```

```
29
30          interfaces_demo::action::Progress::Goal goal;
31          goal.num = num;
32
33
 rclcpp_action::Client<interfaces_demo::action::Progress>::SendGoalOptions
options;
34          options.goal_response_callback =
std::bind(&ActionClient::goal_response_callback, this,
std::placeholders::_1);
35          options.feedback_callback =
std::bind(&ActionClient::feedback_callback, this, std::placeholders::_1,
std::placeholders::_2);
36          options.result_callback = std::bind(&ActionClient::result_callback,
this, std::placeholders::_1);
37
38          // 发送请求
39          client->async_send_goal(goal, options);
40      }
41
42      // 目标响应
43      void
goal_response_callback(rclcpp_action::ClientGoalHandle<interfaces_demo::act
ion::Progress>::SharedPtr goal_handle)
44      {
45          if (!goal_handle)
46          {
47              RCLCPP_INFO(this->get_logger(), "目标请求被拒绝!");
48          }
49          else
50          {
51              RCLCPP_INFO(this->get_logger(), "目标正在被处理中");
52          }
53      }
54
55      // 反馈响应
56      void feedback_callback(
57
 rclcpp_action::ClientGoalHandle<interfaces_demo::action::Progress>::Shared
Ptr goal_handle,
58          const std::shared_ptr<const
interfaces_demo::action::Progress::Feedback> feedback)
59      {
60          double progress = feedback->progress;
61          RCLCPP_INFO(this->get_logger(), "当前进度%.2f%%", progress);
62      }
63
64      // 结果响应
65      void result_callback(const
rclcpp_action::ClientGoalHandle<interfaces_demo::action::Progress>::Wrapped
Result &result)
66      {
67          if (result.code == rclcpp_action::ResultCode::SUCCEEDED)
68          {
69              RCLCPP_INFO(this->get_logger(), "最终结果是:%d", result.result-
>sum);
70          }
71          else if (result.code == rclcpp_action::ResultCode::ABORTED)
```

```
        {
            RCLCPP_INFO(this->get_logger(), "被中断");
        }
        else if (result.code == rclcpp_action::ResultCode::CANCELED)
        {
            RCLCPP_INFO(this->get_logger(), "被取消");
        }
        else
        {
            RCLCPP_INFO(this->get_logger(), "未知错误");
        }
    }
};

int main(int argc, char **argv)
{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<ActionClient>("ActionClient");

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;
}
```

## 修改CmakeLists

```
add_executable(action_server src/action_server.cpp)
ament_target_dependencies(
  action_server
  "rclcpp"
  "interfaces_demo"
  "rclcpp_action"
)

add_executable(action_client src/action_client.cpp)
ament_target_dependencies(
  action_client
  "rclcpp"
  "interfaces_demo"
  "rclcpp_action"
  )

install(TARGETS
  action_server
  action_client
  DESTINATION lib/${PROJECT_NAME}
)
```

## 命令行操作

```
ros2 action list                    # 查看服务列表
ros2 action info <action_name>      # 查看服务数据类型
ros2 action send_goal <action_name> <action_type> <action_data>   # 发送服务请求
```

```
ros2 action send_goal /get_sum interfaces_demo/action/Progress -f "{num: 10}"
```