第3章

ROS通信机制

机器人是高度复杂的系统，在机器人上可能需要集成各种传感器（如激光雷达、摄像头、IMU 等）以及对机器人的运动控制。为了解耦合，ROS 中每个功能点都是一个单独的进程，且每个进程都是独立运行的。更确切地讲，ROS 是进程（也称为 Nodes）分布式框架，为用户提供多节点之间的通信服务。这些进程甚至还可以分布于不同主机，且不同主机协同工作，从而分散计算压力。不过随之产生一个问题：不同的进程是如何通信的？即不同进程间如何实现数据交换？这即为 ROS 的通信机制的相关内容。ROS 的通信机制是最底层，也是最核心的技术。

ROS 的基本通信机制主要有以下三种实现方式：

- 话题通信（发布订阅模式）；
- 服务通信（请求响应模式）；
- 参数服务器（参数共享模式）。

本章的主要内容就是介绍各种通信机制的应用场景、理论模型、代码实现以及相关的操作命令。在学习 ROS 通信机制之前，读者需要先了解 ROS 中的节点（node）和节点管理器（master）。

3.1 Node 和 Master

3.1.1 node

在 ROS 中，最小的进程单元就是节点。一个功能包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程（process），这个进程在 ROS 中就叫作节点。从程序角度来说，节点就是一个可执行文件（通常为 C++编译生成的可执行文件、Python 脚本）被执行、加载到内存之中；从功能角度来说，通常一个节点负责机器人的某项单独的功能。由于机器人的功能模块非常复杂，通常不会把所有功能都集中到一个节点上，而会采用分布式的方式。例如，用一个节点来控制底盘轮子的运动，一个节点驱动摄像头获取图像，一个节点驱

动激光雷达，一个节点根据传感器信息进行路径规划等，这样做可以降低程序发生崩溃的可能性，试想一下，如果把所有功能都写到一个程序中，模块间的通信和异常处理将会很麻烦。

在1.2节中执行了小海龟仿真程序和键盘控制程序，这每个程序便是一个节点。ROS系统中不同功能模块之间的通信也就是节点间的通信。读者可以把键盘控制替换为其他控制方式，而小海龟仿真程序则不用变化，这就是一种模块化分工的思想。

3.1.2　master

由于机器人的元器件很多，功能庞大，实际运行时往往会运行众多的节点，分别实现环境感知、运动控制、决策规划等功能。那么如何合理地调配、管理这些节点呢？这就要利用ROS提供的节点管理器master，master在整个网络通信架构里相当于管理中心，管理着各个节点。节点首先在master处进行注册，之后master会将该节点纳入整个ROS程序中。节点之间的通信也是先由master进行“牵线”后，才能两两之间进行点对点通信。当ROS程序启动时，首先启动master，再由节点管理器依次启动节点。

3.1.3　启动 master 与 node

当需要启动ROS时，首先在终端输入roscore命令，此时ROS master启动，同时启动的还有rosout和parameter server。其中rosout是负责日志输出的一个节点，其作用是告知用户当前系统的状态，包括输出系统的错误、警告等，并且将log记录于日志文件中；parameter server即参数服务器，它并不是一个节点，而是存储参数配置的一个服务器，后文会单独介绍。每次运行ROS的节点前，都需要把master启动起来，这样才能够让节点启动和注册。启动master之后，master就开始按照系统的安排协调启动具体的节点。节点就是一个进程，只不过在ROS中它被赋予了专用的名字——node。学习了第2章介绍的ROS文件系统之后，读者知道一个功能包(package)中存放着可执行文件，可执行文件是静态的，当系统执行这些可执行文件，将这些文件加载到内存中，它就成为了动态的节点。启动具体节点的指令是：

```
rosrun pkg_name node_name
```

通常运行ROS就是按照这样的顺序启动，有时节点太多，会选择用launch文件来启动，在后面的章节会专门进行介绍。master和node之间及各node之间的关系如图3-1所示。

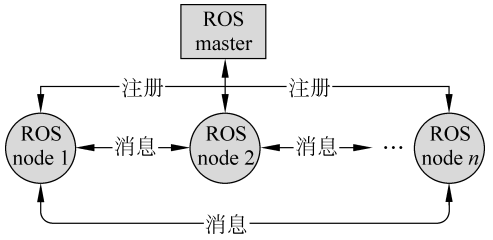

图3-1　node及master之间的关系

3.1.4 rosnode 命令

rosnode 命令用于获取节点信息,其详细用法如表 3-1 所示。

表 3-1 rosnode 命令的用法

rosnode 命令	作 用
rosnode list	列出当前运行的节点信息
rosnode info node_name	显示节点的详细信息
rosnode kill node_name	结束某个节点
rosnode ping	测试到节点的连接状态
rosnode machine	列出指定设备上的节点
rosnode cleanup	清除不可连接的节点

前三个命令较常用,在调试过程中经常需要查看当前节点以及节点信息,所以有必要记住这些命令。也可以通过 rosnode help 来查看 rosnode 命令的用法。

3.2 话题通信机制

话题(topic)通信是 ROS 中使用最频繁的一种通信模式。话题通信是基于发布订阅模式的,即一个节点发布消息,另一个节点订阅该消息。对于实时性、周期性的消息,使用话题来传输是最佳的选择。话题是一种点对点的单向通信方式,这里的"点"指的是节点,也就是说节点之间通过话题方式来传递信息。

像激光雷达、摄像头、GPS 等传感器数据的采集,都使用话题通信,换言之,话题通信适用于不断更新的数据传输相关的应用场景。

3.2.1 话题通信理论模型

话题通信的实现模型是比较复杂的,如图 3-2 所示。模型中涉及三个角色:节点管理者(ROS Master)、发布者(Talker)和订阅者(Listener)。

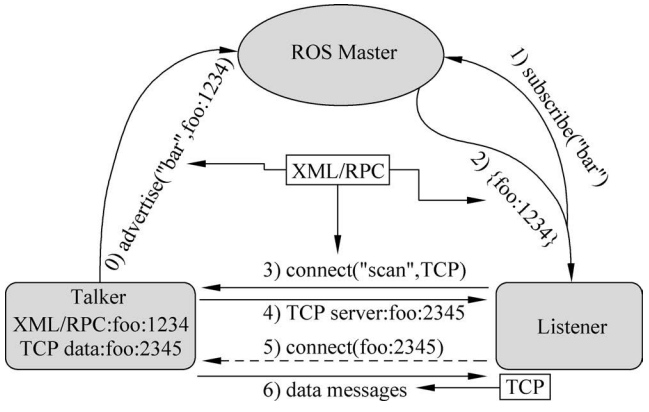

图 3-2 基于发布/订阅模型的话题通信机制

ROS Master 负责保管发布者和订阅者的注册信息，并匹配话题相同的发布者和订阅者，帮助发布者与订阅者建立连接后，发布者可以发布消息，且发布的消息会被订阅者订阅。

话题通信机制的整个流程如下。

1. 发布者注册

发布者启动后，会通过 RPC 在 ROS Master 中注册自身信息和发布消息的话题名称。ROS Master 会将节点的注册信息加入注册表中。

2. 订阅者注册

订阅者启动后，同样会通过 RPC 在 ROS Master 中注册自身信息和需要订阅消息的话题名。ROS Master 会将节点的注册信息加入注册表中。

3. ROS Master 实现信息匹配

ROS Master 会根据注册表中的信息匹配发布者和订阅者，并通过 RPC 向订阅者发送发布者的 RPC 地址信息。

4. 订阅者向发布者发送连接请求

订阅者根据接收到的 RPC 地址，通过 RPC 向发布者发送连接请求，传输订阅的话题名、消息类型及通信协议(TCP/UDP)。

5. 发布者确认连接请求

发布者接收到订阅者的请求后，通过 RPC 向订阅者确认连接信息，并发送自身的 TCP 地址信息。

6. 订阅者与发布者建立网络连接

订阅者根据步骤 4 返回的消息使用 TCP 与发布者建立网络连接。

7. 发布者向订阅者发送消息

成功建立连接后，发布者开始向订阅者发布话题消息。

从话题通信过程可以看出，发布者和订阅者可以有多个。发布者和订阅者建立连接后，不再需要 ROS Master，此后即便关闭 ROS Master，订阅者和发布者照常可以通信。

3.2.2　话题创建示例(C++版)


视频讲解 1


视频讲解 2

分别编写发布者和订阅者程序，实现发布者以 10Hz(每秒 10 次)的频率发布文本消息(“Hello World!”)，订阅者订阅消息并将该消息内容输出至终端。

1. 编写发布者程序

首先在前面章节创建的 catkin_ws 工作空间下，使用如下命令创建功能包和新建文件：

```
cd ~/catkin_ws/src
catkin_create_pkg learning_communication roscpp rospy std_msgs std_srvs
cd learning_communication/src
gedit string_publisher.cpp
```

然后在 string_publisher.cpp 文件中写入如下程序：

```
/**

    消息发布方:
        循环发布信息:Hello World 后缀数字编号

    实现流程:
```

```
    (1)包含头文件
    (2)初始化 ROS 节点:命名(唯一)
    (3)实例化 ROS 句柄
    (4)实例化发布者对象
    (5)组织被发布的数据,并编写逻辑发布数据

 * 该例程将发布 chatter 话题,消息类型为 String
 */

//(1)包含头文件
#include <sstream>
#include "ros/ros.h"
#include "std_msgs/String.h"   //普通文本类型的消息

int main(int argc, char **argv)
{
    //设置编码
    setlocale(LC_ALL,"");

    //(2)初始化 ROS 节点
    //参数 1 和参数 2 在后期为节点传值时使用
    //参数 3 是节点名称,是一个标识符,需要保证运行后在 ROS 网络拓扑中唯一
    ros::init(argc, argv, "string_publisher");

    //(3)实例化 ROS 句柄
    ros::NodeHandle n;   //该类封装了 ROS 中的部分常用功能

    //(4)实例化发布者对象
    //泛型: 发布的消息类型
    //参数 1: 要发布的话题
    //参数 2: 队列中保存的最大消息数,超出此阈值时,先进队列的先销毁(时间早的先销毁)
    //创建一个 Publisher 对象,发布名为 chatter 的话题,消息类型为 std_msgs::String
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

    //设置循环的频率(一秒 10 次)
    ros::Rate loop_rate(10);

int count = 0;
while (ros::ok())
{
    //(5)组织被发布的数据,并编写逻辑发布数据
    // 初始化 std_msgs::String 类型的消息
    //使用 stringstream 拼接字符串与编号
    std_msgs::String msg;
    std::stringstream ss;
    ss << "Hello World! " << count;
    msg.data = ss.str();

    //发布消息
    ROS_INFO("发送的消息:%s", msg.data.c_str());
    chatter_pub.publish(msg);

    //按照循环频率延时
    loop_rate.sleep();
```

```
        ++count;   //循环结束前使 count 自增
    }

    return 0;
}
```

2. 编写订阅者程序

使用 gedit string_subscriber.cpp 命令新建订阅者 CPP 文件，然后在该文件内写入如下程序：

```
/**
    消息订阅方:
        订阅话题并输出接收到的消息

    实现流程:
        (1)包含头文件
        (2)初始化 ROS 节点:命名(唯一)
        (3)实例化 ROS 句柄
        (4)实例化订阅者对象
        (5)处理订阅的消息(回调函数)
        (6)设置循环调用回调函数

 * 该例程将订阅 chatter 话题,消息类型为 String
 */

//(1)包含头文件
#include "ros/ros.h"
#include "std_msgs/String.h"

//(5)处理订阅的消息(回调函数)
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    //将接收到的消息输出
    ROS_INFO("我听见: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    setlocale(LC_ALL,"");

    //(2)初始化 ROS 节点:命名(唯一)
    ros::init(argc, argv, "string_subscriber");

    //(3)实例化 ROS 句柄
    ros::NodeHandle n;

    //(4)实例化订阅者对象
    //创建一个 Subscriber,订阅名为 chatter 的话题,注册回调函数 chatterCallback
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

    //(6)设置循环调用回调函数
    ros::spin();

    return 0;
```

```
}
```

3. 修改配置文件 CMakeLists.txt 的内容,具体如下:

```
add_executable(string_publisher src/string_publisher.cpp)
target_link_libraries(string_publisher ${catkin_LIBRARIES})

add_executable(string_subscriber src/string_subscriber.cpp)
target_link_libraries(string_subscriber ${catkin_LIBRARIES})
```

4. 编译并执行。

CMakeLists.txt 修改完成后,在工作空间的根路径下开始编译:

```
cd ~/catkin_ws
catkin_make
```

编译完成后就可以运行发布者和订阅者这两个节点。在运行节点之前,需要在终端中设置环境变量,否则将无法找到功能包编译生成的可执行文件。即在终端中执行如下语句:

```
source devel/setup.bash
```

也可以将环境变量的配置脚本添加到终端的配置文件中,这样以后就不必每次都先设置环境变量。即执行如下语句:

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

环境变量设置成功后,就可以运行刚编写的发布者和订阅者程序。

(1) 启动 roscore。

在运行节点之前,首先必须确保 ROS Master 已经成功启动。命令如下:

```
roscore
```

(2) 启动发布者(Publisher)。

发布者和订阅者节点的启动顺序在 ROS 中没有要求,这里先启动发布者,命令如下:

```
rosrun learning_communication string_publisher
```

如果发布者节点正常运行,终端中会显示如图 3-3 所示的日志信息。

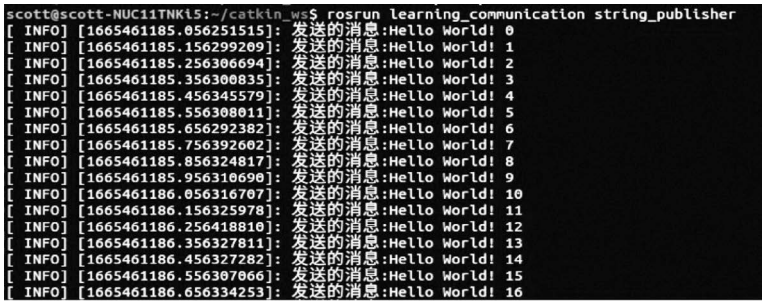

图 3-3 发布者节点成功启动后的日志信息

(3) 启动订阅者(Subscriber)。

成功运行发布者节点后,接下来运行订阅者节点,订阅发布者发布的消息,命令如下:

```
rosrun learning_communication string_subscriber
```

如果订阅者节点成功运行，在终端会显示接收到的消息内容，如图 3-4 所示。

```
scott@scott-NUC11TNKi5:~/catkin_ws$ rosrun learning_communication string_subscriber
[ INFO] [1665461234.757055505]: 我听见: [Hello World! 497]
[ INFO] [1665461234.856849302]: 我听见: [Hello World! 498]
[ INFO] [1665461234.956799012]: 我听见: [Hello World! 499]
[ INFO] [1665461235.056746933]: 我听见: [Hello World! 500]
[ INFO] [1665461235.156667493]: 我听见: [Hello World! 501]
[ INFO] [1665461235.256624766]: 我听见: [Hello World! 502]
[ INFO] [1665461235.356582518]: 我听见: [Hello World! 503]
[ INFO] [1665461235.456665647]: 我听见: [Hello World! 504]
[ INFO] [1665461235.556645235]: 我听见: [Hello World! 505]
[ INFO] [1665461235.656611860]: 我听见: [Hello World! 506]
[ INFO] [1665461235.756577297]: 我听见: [Hello World! 507]
[ INFO] [1665461235.856661418]: 我听见: [Hello World! 508]
[ INFO] [1665461235.956802039]: 我听见: [Hello World! 509]
[ INFO] [1665461236.056891066]: 我听见: [Hello World! 510]
[ INFO] [1665461236.156693399]: 我听见: [Hello World! 511]
[ INFO] [1665461236.256762960]: 我听见: [Hello World! 512]
[ INFO] [1665461236.356802641]: 我听见: [Hello World! 513]
[ INFO] [1665461236.456656445]: 我听见: [Hello World! 514]
[ INFO] [1665461236.556681010]: 我听见: [Hello World! 515]
[ INFO] [1665461236.656655944]: 我听见: [Hello World! 516]
```

图 3-4　订阅者节点成功启动后的日志信息

3.2.3　话题创建示例(Python 版)


视频讲解

本节采用 Python 来实现 3.2.2 节中的案例，达到同样的效果。

1. 编写发布者程序

首先在 learning_communication 文件下新建一个 scripts 文件夹，用于存放 Python 文件。然后在 scripts 文件夹内新建 string_publisher.py 文件，并在该文件内写入如下内容：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    消息发布方:
    循环发布信息:Hello World 后缀数字编号

    实现流程:
        (1)导包
        (2)初始化 ROS 节点:命名(唯一)
        (3)实例化发布者对象
        (4)组织被发布的数据,并编写逻辑发布数据

"""

# 该例程将发布 chatter 话题,消息类型为 String

#(1)导包
import rospy
from std_msgs.msg import String

def string_publisher():
    #(2)初始化 ROS 节点:命名(唯一)
    rospy.init_node('talker', anonymous=True)

    #(3)实例化发布者对象
    # 创建一个 Publisher 对象,发布名为/chatter 的话题,消息类型为 String,队列长度为 10
    pub = rospy.Publisher('chatter', String, queue_size=10)

    #(4)组织被发布的数据,并编写逻辑发布数据
```

```
    msg = String()   #创建 msg 对象
    msg_front = "Hello World!"
    count = 0        #计数器

    #设置循环的频率
    rate = rospy.Rate(10)

    while not rospy.is_shutdown():
        #拼接字符串
        msg.data = msg_front + str(count)

    #发布消息
        pub.publish(msg)
        rospy.loginfo("写出的数据:%s", msg.data)
        count += 1

    #按照循环频率延时
        rate.sleep()

if __name__ == '__main__':
    try:
        string_publisher()
    except rospy.ROSInterruptException:
        pass
```

2. 编写订阅者程序

在 scripts 文件夹内新建 string_subscriber.py 文件,并在该文件内写入如下内容:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    消息订阅方:
        订阅话题并输出接收到的消息

    实现流程:
        (1)导包
        (2)初始化 ROS 节点:命名(唯一)
        (3)实例化订阅者对象
        (4)处理订阅的消息(回调函数)
        (5)设置循环调用回调函数

"""

# 该例程将订阅/person_info 话题,消息类型为 String

#(1)导包
import rospy
from std_msgs.msg import String

#(4)处理订阅的消息(回调函数)
def stringCallback(msg):
    rospy.loginfo(rospy.get_caller_id() + "我听见 %s", msg.data)
```

```
def person_subscriber():
    #(2)初始化 ROS 节点:命名(唯一)
    rospy.init_node('listener', anonymous = True)

    #(3)实例化订阅者对象
    #创建一个 Subscriber 对象,订阅名为/chatter 的话题,注册回调函数 stringCallback
    rospy.Subscriber("chatter", String, stringCallback)

    #(5)设置循环调用回调函数
    rospy.spin()

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

3. 为 Python 文件添加可执行权限

进入 scripts 目录后,打开终端,输入 chmod +x *.py,或右击该 Python 文件,勾选“设置为可执行文件”选项。

4. 编辑配置文件 CMakeLists.txt

```
catkin_install_python(PROGRAMS
  scripts/talker_p.py
  scripts/listener_p.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5. 编译并执行


视频讲解

3.2.4　自定义 msg 话题通信示例

在上面的程序中,chatter 话题的消息类型是 ROS 中预定义的 String 类型。在 ROS 的元功能包 common_msgs 中提供了许多不同消息类型的功能包,如 std_msgs(标准数据类型)、geometry_msgs(几何学数据类型)、sensor_msgs(传感器数据类型)等。这些功能包可以满足大部分场景下的常用消息,但很多情况下,用户依然需要针对自己的机器人应用设计特定的消息类型。

ROS 中通过 std_msgs 封装了一些原生的数据类型,如 String、Int32、Int64、Char、Bool、Empty 等,但是这些数据一般只包含一个 data 字段,结构的单一性意味着功能上的局限性,当传输一些复杂的数据,如激光雷达的信息时,std_msgs 由于描述性较差而显得力不从心,在这种场景下,用户可以使用自定义的消息类型。

msgs 只是简单的文本文件,每行具有字段类型和字段名称,可以使用的字段类型包括:int8, int16, int32, int64(或者无符号类型 uint*),float32, float64、string,time, duration, other msg files、variable-length array[]和 fixed-length array[C]。

ROS 中还有一种特殊类型:Header(标头),包含时间戳和 ROS 中常用的坐标帧信息。用户经常可以看到 msg 文件的第一行具有 Header。

msg 文件就是 ROS 中定义消息类型的文件,一般放置在功能包根目录下的 msg 文件夹中。在功能包编译过程中,可以使用 msg 文件生成不同编程语言的代码文件。下面通过创建自定义消息如要求该消息包含人的信息,如姓名、身高、年龄等。

编写流程如下。

1. 按照固定格式创建 msg 文件

在功能包下新建 msg 文件夹，新建文件 PersonMsg.msg，并在该文件中写入如下信息：

```
string name
uint16 age
float64 height
```

注：这里使用的基础数据类型 string、uint16、float64 都是与编程语言无关的，在编译阶段会变成各种语言对应的数据类型。

2. 编辑配置文件

(1) 在 package.xml 中添加编译依赖与执行依赖。

```
< build_depend > message_generation </build_depend >
< exec_depend > message_runtime </exec_depend >
<!--
exec_depend 以前对应的是 run_depend,现在非法
-->
```

(2) CMakeLists.txt 编辑 msg 相关配置。

打开功能包的 CMakeLists.txt 文件，在 find_package 中添加消息生成依赖的功能包 message_generation，这样在编译时才能找到所需的文件。

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
#需要加入 message_generation,必须有 std_msgs
```

设置需要编译的 msg 文件：

```
## 配置 msg 源文件
add_message_files(
  FILES
  PersonMsg.msg
)
#生成消息时依赖于 std_msgs
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

catkin 依赖也需要进行以下设置：

```
#执行时依赖
catkin_package(
#  INCLUDE_DIRS include
   CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
#  DEPENDS system_lib
)
```

3. 编译、生成可以被 Python 或 C++调用的中间文件

以上配置工作完成后，回到工作空间的根路径下，使用 catkin_make 命令进行编译。编译完成后，可以使用如下命令查看自定义的 PersonMsg 消息类型，如图 3-5 所示。

```
rosmsg show PersonMsg
```

```
scott@scott-NUC11TNKi5:~/catkin_ws$ rosmsg show PersonMsg
[learning_communication/PersonMsg]:
string name
uint16 age
float64 height
```

图 3-5　自定义 PersonMsg 消息类型的数据信息

PersonMsg 消息类型定义成功后，在代码中就可以按照以上 String 类型的使用方法使用 PersonMsg 类型的消息。

3.2.5　自定义 msg 话题消息调用(C++版)

3.2.4 小节已经完成了 PersonMsg 消息类型的定义，本小节将通过编写发布者和订阅者程序，实现发布者以 10Hz(每秒 10 次)的频率发布 PersonMsg 自定义消息，订阅者订阅 PersonMsg 自定义消息并将消息内容输出至终端。实现流程如下。

1. 编写发布者程序

```
/**
 * 该例程将发布/person_info 话题,learning_communication::PersonMsg
 */

#include <ros/ros.h>
#include "learning_communication/PersonMsg.h"

int main(int argc, char **argv)
{
    setlocale(LC_ALL,"");

    //(1)初始化 ROS 节点
    ros::init(argc, argv, "person_publisher");

    //(2)创建 ROS 句柄
    ros::NodeHandle n;

    //(3)创建发布者对象
    // 创建一个 Publisher 对象,发布名为/person_info 的话题,消息类型为 learning_communication::
    //PersonMsg,队列长度为 10
    ros::Publisher person_info_pub = n.advertise<learning_communication::PersonMsg>
    ("/person_info", 10);

    //设置循环的频率
    ros::Rate loop_rate(1);

    int count = 0;
    while (ros::ok())
    {
        //(4)组织被发布的消息,编写发布逻辑并发布消息
        //初始化 learning_communication::PersonMsg 类型的消息
    learning_communication::PersonMsg person_msg;
    person_msg.name = "Mike";
    person_msg.age = 20;
    person_msg.height = 1.65;
```

```
    //发布消息
    person_info_pub.publish(person_msg);

        ROS_INFO("我叫%s,今年%d岁,身高%.2f米",
            person_msg.name.c_str(), person_msg.age, person_msg.height);

        //按照循环频率延时
        loop_rate.sleep();
    }

    return 0;
}
```

2. 编写订阅者程序

```
/**
 * 该例程将订阅/person_info话题,自定义消息类型learning_communication::PersonMsg
 */

#include <ros/ros.h>
#include "learning_communication/PersonMsg.h"

//(1)回调函数中处理personMsg
// 接收到订阅的消息后,会进入消息回调函数
void personInfoCallback(const learning_communication::PersonMsg::ConstPtr& msg)
{
    // 将接收到的消息输出
    ROS_INFO("订阅的人信息:我的名字是%s, 年龄%d, 身高%.2f",
             msg->name.c_str(), msg->age, msg->height);
}

int main(int argc, char **argv)
{
    setlocale(LC_ALL,"");

    //(2)初始化ROS节点
    ros::init(argc, argv, "person_subscriber");

    //(3)创建ROS句柄
    ros::NodeHandle n;

    //(4)创建订阅对象
    //创建一个Subscriber对象,订阅名为/person_info的话题,注册回调函数personInfoCallback
    ros::Subscriber person_info_sub = n.subscribe("/person_info", 10, personInfoCallback);

    //(5)循环等待回调函数
    ros::spin();

    return 0;
}
```

3. 编辑配置文件CMakeLists.txt

需要添加add_dependencies用以设置所依赖的消息相关的中间文件。

```
add_executable(person_publisher src/person_publisher.cpp)
```

```
target_link_libraries(person_publisher ${catkin_LIBRARIES})
add_dependencies(person_publisher ${PROJECT_NAME}_gencpp)

add_executable(person_subscriber src/person_subscriber.cpp)
target_link_libraries(person_subscriber ${catkin_LIBRARIES})
add_dependencies(person_subscriber ${PROJECT_NAME}_gencpp)
```

4. 编译并执行

编译和执行过程与3.2.4小节的案例类似。

3.2.6　自定义msg话题消息调用(Python版)

与3.2.5小节的案例类似,本小节将采用Python编程实现自定义消息PersonMsg的调用,流程如下。

1. 编写发布者代码

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# 该例程将发布/person_info话题,自定义消息类型 learning_communication::PersonMsg

import rospy
from learning_communication.msg import PersonMsg

def velocity_publisher():
    #(1)初始化ROS节点
    rospy.init_node('person_publisher', anonymous=True)

    #(2)创建发布者对象
    # 创建一个Publisher对象,发布名为/person_info的话题,消息类型为PersonMsg,队列长度为10
    person_info_pub = rospy.Publisher('/person_info', PersonMsg, queue_size=10)

    #设置循环的频率
    rate = rospy.Rate(10)

    while not rospy.is_shutdown():
    #(3)组织消息
    # 初始化PersonMsg类型的消息
        person_msg = PersonMsg()
        person_msg.name = "Mike";
        person_msg.age = 20;
        person_msg.height = 1.65;

    #(4)发布消息
        person_info_pub.publish(person_msg)
        rospy.loginfo("Publish person message[姓名%s, 年龄%d, 身高%.2f]",
                person_msg.name, person_msg.age, person_msg.height)

    # 按照循环频率延时
        rate.sleep()

    if __name__ == '__main__':
```

```
        try:
            velocity_publisher()
        except rospy.ROSInterruptException:
            Pass
```

2. 编写订阅者代码

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# 该例程将订阅/person_info 话题,自定义消息类型 PersonMsg

import rospy
from learning_communication.msg import PersonMsg

def personInfoCallback(msg):
    rospy.loginfo("接收到的人的信息: name: %s age: %d height: %.2f",
                  msg.name, msg.age, msg.height)

def person_subscriber():
    #(1)初始化节点
    rospy.init_node('person_subscriber', anonymous = True)

    #(2)创建订阅者对象
    # 创建一个 Subscriber 对象,订阅名为/person_info 的话题,注册回调函数
    # personInfoCallback
    rospy.Subscriber("/person_info", PersonMsg, personInfoCallback)

    #(3)循环等待回调函数
    rospy.spin()

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

3. 为 Python 文件添加可执行权限

进入 scripts 文件夹后,打开终端执行 chmod +x *.py,或者右击 Python 文件,勾选“可执行文件”选项。

4. 编辑配置文件 CMakeLists.txt

```
catkin_install_python(PROGRAMS
  scripts/person_publisher.py
  scripts/person_subscriber.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

5. 编译并执行

执行过程与前面的案例类似。

3.3 常见的消息类型

本节主要介绍常见的消息(message)类型,包括 std_msgs、geometry_msgs、nav_msgs、sensor_msgs 等。

1. Header.msg

```
#定义数据的参考时间和参考坐标
#文件位置:std_msgs/Header.msg
uint32 seq          #数据 ID
time stamp          #数据时间戳
string frame_id     #数据的参考坐标系
```

2. Vector3.msg

```
#文件位置:geometry_msgs/Vector3.msg
float64 x
float64 y
float64 z
```

3. Accel.msg

```
#定义加速度项,包括线性加速度和角加速度
#文件位置:geometry_msgs/Accel.msg
Vector3 linear
Vector3 angular
```

4. Quaternion.msg

```
#消息代表空间中旋转的四元数
#文件位置:geometry_msgs/Quaternion.msg
float64 x
float64 y
float64 z
float64 w
```

5. Point.msg

```
#空间中的点的位置
#文件位置:geometry_msgs/Point.msg
float64 x
float64 y
float64 z
```

6. Pose.msg

```
#消息定义自由空间中的位姿信息,包括位置和指向信息
#文件位置:geometry_msgs/Pose.msg
Point position
Quaternion orientation
```

7. PoseStamped.msg

```
#定义有时空基准的位姿
#文件位置:geometry_msgs/PoseStamped.msg
Header header
Pose pose
```

8. PoseWithCovariance.msg

```
#表示空间中含有不确定性的位姿信息
#文件位置:geometry_msgs/PoseWithCovariance.msg
Pose pose
float64[36] covariance
```

9. Twist.msg

```
#定义空间中物体运动的线速度和角速度
#文件位置:geometry_msgs/Twist.msg
Vector3 linear
Vector3 angular
```

10. TwistWithCovariance.msg

```
#消息定义了包含不确定性的速度量,协方差矩阵按行分别表示:
#沿 x 方向速度的不确定性,沿 y 方向速度的不确定性,沿 z 方向速度的不确定性
#绕 x 转动角速度的不确定性,绕 y 轴转动的角速度的不确定性,绕 z 轴转动的
#角速度的不确定性
#文件位置:geometry_msgs/TwistWithCovariance.msg
Twist twist
float64[36] covariance     #分别表示[x; y; z; Rx; Ry; Rz]
```

11. Odometry.msg

```
#消息描述了自由空间中位置和速度的估计值
#文件位置:nav_msgs/Odometry.msg
Header header
string child_frame_id
PoseWithCovariance pose
TwistWithCovariance twist
```

12. Power.msg

```
#表示电源状态,是否开启
#文件位置:自定义 msg 文件
Header header
bool power
##################
bool ON = 1
 bool OFF = 0
```

13. Echos.msg

```
#定义超声传感器
#文件位置:自定义 msg 文件
Header header
uint16 front_left
uint16 front_center
uint16 front_right
uint16 rear_left
uint16 rear_center
uint16 rear_right
```

14. Imu.msg

```
#消息包含了从惯性元件中得到的数据,加速度单位为 m/s^2,角速度单位为 rad/s
#如果所有的测量协方差已知,则需要全部填充进来。如果只知道方差,则只填充协方差矩阵的对角
#数据即可
#位置:sensor_msgs/Imu.msg
Header header
Quaternion orientation
float64[9] orientation_covariance
Vector3 angular_velocity
```

```
float64[9] angular_velocity_covariance
Vector3 linear_acceleration
float64[] linear_acceleration_covariance
```

15. LaserScan.msg

```
#平面内的激光测距扫描数据,注意此消息类型仅仅适配激光测距设备
#如果有其他类型的测距设备(如超声波传感器),需要另外创建不同类型的消息
#位置:sensor_msgs/LaserScan.msg
Header header            #时间戳为接收到第一束激光的时间
float32 angle_min        #扫描开始时的角度(单位为 rad)
float32 angle_max        #扫描结束时的角度(单位为 rad)
float32 angle_increment  #两次测量之间的角度增量(单位为 rad)
float32 time_increment   #两次测量之间的时间增量(单位为 s)
float32 scan_time        #两次扫描之间的时间间隔(单位为 s)
float32 range_min        #距离最小值(单位为 m)
float32 range_max        #距离最大值(单位为 m)
float32[] ranges         #测距数据(单位为 m,如果数据不在最小数据和最大数据之间,则抛弃)
float32[] intensities    #强度,具体单位由测量设备确定,如果仪器没有强度测量,则数组为空即可
```

3.4　服务通信机制

前面介绍了 ROS 通信方式中的话题通信。话题是 ROS 中一种单向的异步通信方式，然而有时单向的通信满足不了通信要求，例如机器人巡逻过程中，控制系统分析传感器数据，发现可疑物体或人，此时需要拍摄照片并留存，如果用话题通信方式时就会消耗大量不必要的系统资源，造成系统的低效率高功耗。

这种情况下，就需要有另外一种请求-查询式的通信模型。本小节将介绍 ROS 通信中的另一种通信方式——service(服务)。Service 是节点之间同步通信的一种方式，允许客户端(Client)节点发布请求(Request)，由服务端节点处理后反馈应答。服务通信更适用于对实时性有要求、具有一定逻辑处理能力的应用场景。

3.4.1　服务通信的理论模型

服务通信相较于话题通信更简单，其理论模型如图 3-6 所示。该模型涉及三个角色：节点管理者(ROS Master)、服务端(Server)和客户端(Client)。

ROS Master 负责保管 Server 和 Client 注册的信息，并匹配话题相同的 Server 与 Client，帮助 Server 与 Client 建立连接后，客户端发送请求信息，服务端返回响应信息。

服务通信机制的实现流程如下。

1. 服务端注册

服务端启动后，会通过 RPC 在 ROS Master 中注册自身信息和提供的服务名称。ROS Master 会将节点的注册信息加入注册表中。

2. 客户端注册

客户端启动后，也会通过 RPC 在 ROS Master 中注册自身信息和需要请求的服务名称。ROS Master 会将节点的注册信息加入注册表中。

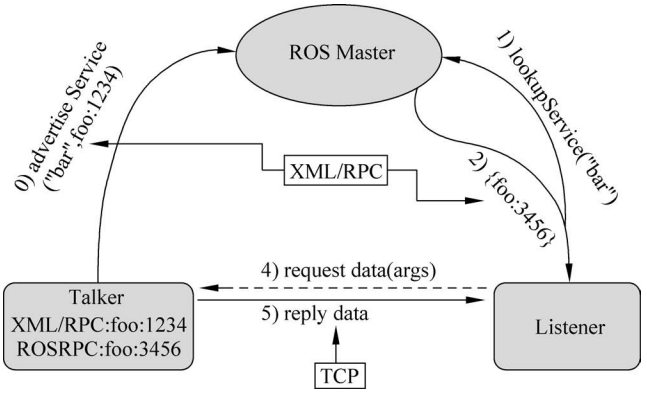

图 3-6　基于服务端/客户端模型的服务通信机制

3. ROS Master 实现信息匹配

ROS Master 会根据注册表中的信息匹配 Server 和 Client,并通过 RPC 向 Client 发送 Server 的 TCP 地址信息。

4. 客户端发送请求

客户端根据步骤 2 响应的信息,使用 TCP 与服务端建立网络连接,并发送请求数据。

5. 服务端发送响应

服务端接收、解析请求的数据,并产生响应结果返回给客户端。

3.4.2　服务通信机制示例(C++版)

分别编写服务端和客户端程序,实现文本消息("Hello ROS!")服务通信的过程。

1. 编写服务端实现代码(string_server.cpp)

```
/**
 * 该例程将提供 print_string 服务,std_srvs::SetBool
 */

//(1)包含头文件
#include "ros/ros.h"
#include "std_srvs/SetBool.h"

// service 回调函数,输入参数为 req,输出参数为 res
bool print(std_srvs::SetBool::Request &req,
           std_srvs::SetBool::Response &res)
{
    // 输出字符串
    if(req.data)
{
    ROS_INFO("服务器收到请求信息为:Hello ROS!");
    res.success = true;
    res.message = "Print Successfully";
}
else
{
    res.success = false;
    res.message = "Print Failed";
```

```
}

    return true;
}

int main(int argc, char **argv)
{
    setlocale(LC_ALL,"");

    //(2)初始化 ROS 节点
    ros::init(argc, argv, "string_server");

    //(3)创建 ROS 句柄
    ros::NodeHandle n;

    //(4)创建服务对象
    // 创建一个名为 print_string 的 server,注册回调函数 print()
    ros::ServiceServer service = n.advertiseService("print_string", print);

    //(5)回调函数处理请求并产生响应
    //(6)由于请求有多个,需要调用 ros::spin()
    ROS_INFO("服务已经启动,准备输出字符串");
    ros::spin();

    return 0;
}
```

2. 编写客户端实现代码(string_client.cpp)

```
/**
 * 该例程将请求 print_string 服务,std_srvs::SetBool
 */

//(1)包含头文件
#include "ros/ros.h"
#include "std_srvs/SetBool.h"

int main(int argc, char **argv)
{
    setlocale(LC_ALL,"");

    //(2)初始化 ROS 节点
    ros::init(argc, argv, "string_client");

    //(3)创建 ROS 句柄
    ros::NodeHandle n;

    //(4)创建客户端对象
    // 创建一个 client 对象,service 消息类型是 std_srvs::SetBool
    ros::ServiceClient client = n.serviceClient<std_srvs::SetBool>("print_string");

    //(5)组织请求数据
    // 创建 std_srvs::SetBool 类型的 service 消息
    std_srvs::SetBool srv;
    srv.request.data = true;
```

```
    //(6)发送请求,返回 bool 值,标记是否成功
    bool flag = client.call(srv);

    //(7)处理响应
    // 发布 service 请求,等待应答结果
    if (flag)
    {
        ROS_INFO("请求正常处理,响应结果 : [ %s] %s", srv.response.success?"True":
"False",
                                        srv.response.message.c_str());
    }
    else
    {
        ROS_ERROR("请求 print_string 失败 ");
        return 1;
    }

    return 0;
}
```

3. 编辑 CMakeLists.txt 配置文件

```
add_executable(string_server src/string_server.cpp)
target_link_libraries(string_server ${catkin_LIBRARIES})

add_executable(string_client src/string_client.cpp)
target_link_libraries(string_client ${catkin_LIBRARIES})
```

4. 编译并执行

(1) 先对程序进行编译,并设置环境变量;

```
catkin_make
source devel/setup.bash
```

(2) 启动 roscore。

(3) 启动服务端节点。

打开一个新的终端,运行服务端节点:

```
rosrun learning_communication string_server
```

如果运行正常,则在终端会显示如图 3-7 所示的信息。

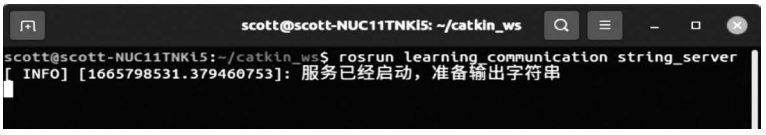

图 3-7 服务器节点启动后的日志信息

(4) 运行客户端节点。

打开一个新的终端,运行客户端节点:

```
rosrun learning_communication string_client
```

客户端发布服务请求,服务端完成服务功能后反馈结果给客户端。在客户端和服务端

的终端中分别可以看到如图 3-8 和图 3-9 所示的日志信息。

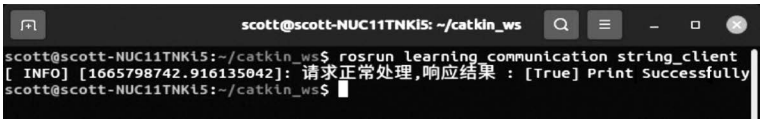

图 3-8　客户端启动后发布服务请求后的日志信息

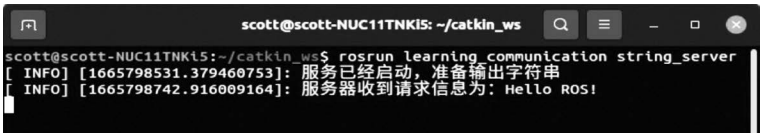

图 3-9　服务端接收到服务调用后输出的日志信息

3.4.3　服务通信机制示例(Python 版)

使用 Python 语言分别编写服务端和客户端程序,实现文本消息("Hello ROS!")服务通信的过程如下。

1. 编写服务端实现代码(string_server.py)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# 该例程将提供 print_string 服务,std_srvs::SetBool

#(1)导包
import rospy
from std_srvs.srv import SetBool, SetBoolResponse

# 回调函数的参数是请求对象,返回值是响应对象
def stringCallback(req):
    # 显示请求数据
    if req.data:
        rospy.loginfo("服务器收到请求信息为:Hello ROS!")

    # 反馈数据
        return SetBoolResponse(True, "Print Successfully")
    else:
        # 反馈数据
        return SetBoolResponse(False, "Print Failed")

def string_server():
    #(2)初始化 ROS 节点
    rospy.init_node('string_server')

    #(3)创建服务对象
    # 创建一个名为/print_string 的 server,注册回调函数 stringCallback
    s = rospy.Service('print_string', SetBool, stringCallback)

    #(4)回调函数处理请求并产生响应
    # 循环等待回调函数
    print ("服务已经启动,准备输出字符串")
    rospy.spin()
```

```
if __name__ == "__main__":
    string_server()
```

2. 编写客户端实现代码(string_client.py)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# 该例程将请求 print_string 服务,std_srvs::SetBool

#(1)导包
import sys
import rospy
from std_srvs.srv import SetBool, SetBoolRequest

def string_client():
    #(2)初始化 ROS 节点
    rospy.init_node('string_client')

    #(3)创建请求对象
    # 发现 print_string 服务后,创建一个服务客户端,连接名为 print_string 的 service
    rospy.wait_for_service('print_string')
    try:
        string_client = rospy.ServiceProxy('print_string', SetBool)

    #(4)请求服务调用,输入请求数据
        response = string_client(True)
        return response.success, response.message
    except (rospy.ServiceException, e):
        print ("Service call failed: %s" % e)

if __name__ == "__main__":
#服务调用并输出调用结果
    print ("响应结果 : [%s] %s" %(string_client()))
```

3. 为 Python 文件添加可执行权限

进入 scripts 文件夹后,打开终端,输入 chmod +x *.py,或右击该 Python 文件,勾选“设置为可执行文件”选项。

4. 编辑配置文件 CMakeLists.txt

```
catkin_install_python(PROGRAMS
  scripts/string_server.py
  scripts/string_client.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5. 编译并执行

与 3.4.3 小节执行过程类似,分别启动 roscore、服务端和客户端。


视频讲解

3.4.4 自定义 srv 服务数据

本小节以一个简单的加法运算为例,介绍 ROS 中自定义服务的应用。本例中,客户端提交两个整数至服务端,服务端接收请求后完成求和运算,并返回结果给客户端。

与话题消息类似,ROS 中的服务数据可以通过 srv 进行语言无关的接口定义,一般放

置在功能包根路径下的 srv 文件夹中。实现自定义 srv 服务数据的流程如下。

1. 按照固定格式创建 srv 文件

服务通信中，数据分成两部分，请求与应答两个数据域，数据域中的内容与话题消息的数据类型相同，在 srv 文件中请求和应答描述之间使用“---”分割，具体实现如下。

在功能包下新建 srv 文件夹，并创建 AddTwoInts.srv 文件，内容如下：

```
# 客户端请求时发送的两个数字
int32 num1
int32 num2
---
# 服务器响应发送的数据
int32 sum
```

2. 编辑配置文件

完成服务数据类型的描述后，与前面的话题消息类似，也需要在功能包的 package.xml 和 CMakeLists.txt 文件中配置依赖及编译规则，编译后将该描述文件转换成编程语言所能识别的代码。

打开 package.xml 文件，添加编译依赖与执行依赖（在定义话题消息的时候已经添加过）。

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

打开 CMakeLists.txt 文件，编辑 srv 相关配置。

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  std_srvs
  message_generation
)
# 需要加入 message_generation,必须有 std_msgs
add_service_files(
  FILES
  AddTwoInts.srv
)
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

3. 编译生成中间文件

功能包编译成功后，在服务的服务端节点和客户端节点的代码实现中就可以直接调用这些定义好的服务消息。接下来编写客户端和服务端节点的代码，完成两个数求和的服务过程。

3.4.5　自定义 srv 服务通信调用（C++版）

在 3.4.4 小节中已经完成了自定义服务数据的定义，本小节将通过编写客户端和服务器，实现两数相加的服务过程。实现流程如下。

1. 编写服务端实现代码(AddTwoInts_Server.cpp)

```
/*
    服务端实现:
        (1)包含头文件
        (2)初始化 ROS 节点
        (3)创建 ROS 句柄
        (4)创建服务对象
        (5)回调函数处理请求并产生响应
        (6)由于请求有多个,则需要调用 ros::spin()

*/

//(1)包含头文件
#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"

//(5)回调函数处理请求并产生响应
//返回 bool 值,标记是否处理成功
bool doReq(learning_communication::AddTwoInts::Request& req,
           learning_communication::AddTwoInts::Response& resp){
    int num1 = req.num1;
    int num2 = req.num2;

    ROS_INFO("服务器接收到的请求数据为:num1 = %d, num2 = %d",num1, num2);

    //逻辑处理
    if (num1 < 0 || num2 < 0)
    {
        ROS_ERROR("提交的数据异常:数据不可以为负数");
        return false;
    }

    //如果没有异常,那么相加并将结果赋值给 resp
    resp.sum = num1 + num2;
    return true;

}

int main(int argc, char *argv[])
{
    setlocale(LC_ALL,"");
    //(2)初始化 ROS 节点
    ros::init(argc,argv,"AddTwoInts_Server");
    //(3)创建 ROS 句柄
    ros::NodeHandle nh;
    //(4)创建服务对象
    ros::ServiceServer server = nh.advertiseService("AddTwoInts",doReq);
    ROS_INFO("服务已经启动....");

    //(6)由于请求有多个,则需要调用 ros::spin()
    ros::spin();
    return 0;
}
```

2. 编写客户端实现代码(AddTwoInts_Client.cpp)

```
/*
    服务端实现:
        (1)包含头文件
        (2)初始化 ROS 节点
        (3)创建 ROS 句柄
        (4)创建客户端对象
        (5)~(7)请求服务,接收响应
*/
//(1)包含头文件
#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"

int main(int argc, char *argv[])
{
    setlocale(LC_ALL,"");

    // 调用时动态传值,如果通过 launch 的 args 传参,需要传递的参数个数为 + 3
    if (argc != 3)
        {
        ROS_ERROR("请提交两个整数");
        return 1;
    }

    //(2)初始化 ROS 节点
    ros::init(argc,argv,"AddTwoInts_Client");
    //(3)创建 ROS 句柄
    ros::NodeHandle nh;
    //(4)创建客户端对象
    ros::ServiceClient client = nh.serviceClient<learning_communication::AddTwoInts>
("AddTwoInts");
    //等待服务启动成功
    //方式 1
    ros::service::waitForService("AddTwoInts");
    //方式 2
    // client.waitForExistence();
    //(5)组织请求数据
    learning_communication::AddTwoInts ai;
    ai.request.num1 = atoi(argv[1]);
    ai.request.num2 = atoi(argv[2]);
    //(6)发送请求,返回 bool 值,标记是否成功
    bool flag = client.call(ai);
    //(7)处理响应
    if (flag)
    {
        ROS_INFO("请求正常处理,响应结果:%d",ai.response.sum);
    }
    else
    {
        ROS_ERROR("请求处理失败....");
        return 1;
    }
```

```
    return 0;
}
```

3. 编辑 CMakeLists.txt 配置文件

```
add_executable(AddTwoInts_Server src/AddTwoInts_Server.cpp)
target_link_libraries(AddTwoInts_Server ${catkin_LIBRARIES})
add_dependencies(AddTwoInts_Server ${PROJECT_NAME}_gencpp)

add_executable(AddTwoInts_Client src/AddTwoInts_Client.cpp)
target_link_libraries(AddTwoInts_Client ${catkin_LIBRARIES})
add_dependencies(AddTwoInts_Client ${PROJECT_NAME}_gencpp)
```

4. 编译并执行

(1) 先对程序进行编译,并设置环境变量。

(2) 启动 roscore。

(3) 启动服务端节点。

```
rosrun learning_communication AddTwoInts_Server
```

(4) 运行客户端节点。

打开一个新的终端,运行客户端节点,同时需要输入加法运行的两个参数值,具体命令如下:

```
rosrun learning_communication AddTwoInts_Client 1 2
```

客户端发布服务请求,服务端完成服务功能后反馈结果给客户端。在客户端和服务端的终端中分别可以看到如图 3-10 和图 3-11 所示的日志信息。

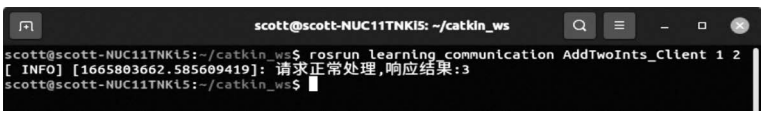

图 3-10　客户端启动后发布服务请求后的日志信息

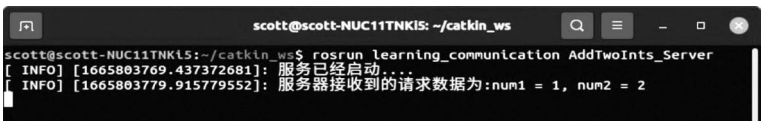

图 3-11　服务端接收到服务调用后完成加法运算后输出的日志信息

3.4.6　自定义 srv 服务通信调用(Python 版)

本小节采用 Python 编程实现与 3.4.5 小节案例同样的效果,具体流程如下。

1. 编写服务端实现代码

```
#! /usr/bin/env python
"""
    服务端实现:
        (1)导包
        (2)初始化 ROS 节点
        (3)创建服务对象
        (4)回调函数处理请求并产生响应
        (5)spin 函数
```

```
"""
#(1)导包
import rospy
from learning_communication.srv import AddTwoInts,AddTwoIntsRequest,AddTwoIntsResponse

#(4)回调函数处理请求并产生响应
# 回调函数的参数是请求对象,返回值是响应对象
def doReq(req):
    # 解析提交的数据
    sum = req.num1 + req.num2
    rospy.loginfo("提交的数据:num1 = %d, num2 = %d, sum = %d",req.num1, req.num2, sum)

    # 创建响应对象,赋值并返回
    # resp = AddIntsResponse()
    # resp.sum = sum
    resp = AddTwoIntsResponse(sum)
    return resp

if __name__ == "__main__":
    #(2)初始化 ROS 节点
    rospy.init_node("addTwoints_server_p")
    #(3)创建服务对象
    server = rospy.Service("AddTwoInts",AddTwoInts,doReq)

    #(5)spin 函数
    rospy.spin()
```

2. 编写客户端实现代码

```
#! /usr/bin/env python

"""
    客户端实现:
        (1)导包
        (2)初始化 ROS 节点
        (3)创建请求对象
        (4)发送请求
        (5)接收并处理响应

    优化:
        加入数据的动态获取
"""
#(1)导包
import rospy
from learning_communication.srv import *
import sys

if __name__ == "__main__":

    #优化实现
    if len(sys.argv) != 3:
        rospy.logerr("请正确提交参数")
        sys.exit(1)
```

```
    #(2)初始化 ROS 节点
    rospy.init_node("AddTwoInts_Client_p")
    #(3)创建请求对象
    client = rospy.ServiceProxy("AddTwoInts",AddTwoInts)
    # 请求前,等待服务已经就绪
    # 方式 1:
    # rospy.wait_for_service("AddTwoInts")
    # 方式 2
    client.wait_for_service()
    #(4)和(5)发送请求,接收并处理响应
    # 方式 1
    # resp = client(3,4)
    # 方式 2
    # resp = client(AddIntsRequest(1,5))
    # 方式 3
    req = AddTwoIntsRequest()
    # req.num1 = 100
    # req.num2 = 200

    #优化
    req.num1 = int(sys.argv[1])
    req.num2 = int(sys.argv[2])

    resp = client.call(req)
    rospy.loginfo("响应结果:%d",resp.sum)
```

3. 为 Python 文件添加可执行权限

进入 scripts 目录,打开终端,执行 chmod +x *.py。

4. 编辑 CMakeLists.txt 配置文件

```
catkin_install_python(PROGRAMS
  scripts/AddTwoInts_Server.py
  scripts/AddTwoInts_Client.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5. 编译并执行

与 3.4.5 小节执行过程类似,分别启动 roscore、服务端和客户端。

3.5 常见的服务通信

本小节介绍常见的 srv 类型和自定义 srv 类型。服务通信相当于两个 message 通道,即一个通道用于发送,另一个通道用于接收。

1. SetBools.srv

```
#文件位置:std_srvs/SetBools.srv
bool data          # 启动或者关闭硬件
---
bool success       # 标示硬件是否成功运行
```

```
string message     # 运行信息
```

2. Trigger.srv

```
# 文件位置:std_srvs/Trigger.srv
---
bool success          # 标示 srv 是否成功运行
string message        # 信息,如错误信息等
```

3. TalkerListener.srv

```
# 文件位置: 自定义 srv 文件
---
bool success          # 标示 srv 是否成功运行
string message        # 信息,如错误信息等
```

4. AddTwoInts.srv

```
# 对两个整数求和,虚线前是输入量,后是返回量
# 文件位置:自定义 srv 文件
int32 a
int32 b
---
int32 sum
```

5. GetMap.srv

```
# 文件位置:nav_msgs/GetMap.srv
# 获取地图,注意请求部分为空
---  nav_msgs/OccupancyGrid map
```

6. GetPlan.srv

```
# 文件位置:nav_msgs/GetPlan.srv
# 得到一条从当前位置到目标点的路径
geometry_msgs/PoseStamped start      # 起始点
geometry_msgs/PoseStamped goal       # 目标点
float32 tolerance      # 到达目标点的 x,y 方向的容错距离
---
nav_msgs/Path plan
```

7. SetMap.srv

```
# 文件位置:nav_msgs/SetMap.srv
# 以初始位置为基准,设定新的地图
nav_msgs/OccupancyGrid map geometry_msgs/PoseWithCovarianceStamped initial_pose
---
bool success
```

8. SetCameraInfo.srv

```
# 文件位置:sensor_msgs/SetCameraInfo.srv
# 通过给定的 CameraInfo 相机信息来对相机进行标定 sensor_msgs/CameraInfo camera_info
                        # 相机信息
---
bool success              # 如果调用成功,则返回 true
string status_message     # 给出调用成功的细节
```

3.6 参数服务器

前面介绍了 ROS 中常见的两种通信方式——主题和服务，本节介绍另一种通信方式——参数服务器(parameter server)。与前两种通信方式不同，参数服务器也可以说是特殊的“通信方式”。参数服务器在 ROS 中主要用于实现不同节点之间的数据共享。

参数服务器相当于独立于所有节点的一个公共容器，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往里面存储数据。参数服务器机制如图 3-12 所示。参数服务器一般适用于存在数据共享的一些应用场景，例如实现路径规划时需要参考小车的尺寸，用户可以将这些尺寸信息存储到参数服务器，全局路径规划节点与本地路径规划节点都可以从参数服务器中调用这些参数。

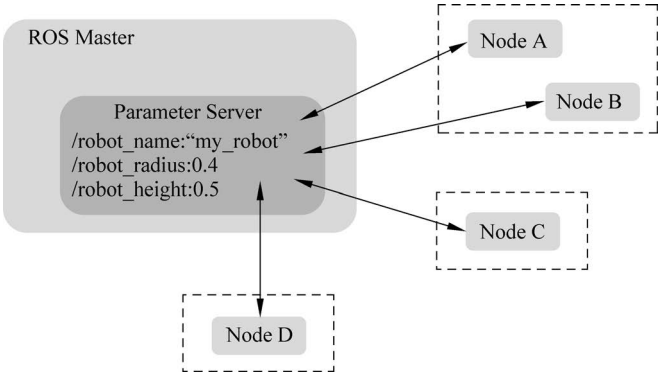

图 3-12 参数服务器机制

参数服务器的维护方式非常简单灵活，总的来讲有三种方式：命令行参数设置、launch 文件内读写参数和节点源码内读写参数。

3.6.1 命令行参数设置

使用命令行来设置参数服务，主要通过 rosparam 命令来进行各种操作设置。rosparam 能够存储并操作 ROS 参数服务器(parameter server)上的数据。参数服务器能够存储整型、浮点、布尔、字符串、字典和列表等数据类型。rosparam 使用 YAML 标记语言的语法。一般而言，YAML 的表述很自然：1 是整型，1.0 是浮点型，one 是字符串，true 是布尔，[1，2，3]是整型列表，{a：b，c：d}是字典。rosparam 有很多命令可以用来操作参数，如表 3-2 所示。

表 3-2 rosparam 命令

rosparam 命令	作　用
rosparam set param_key param_value	设置参数
rosparam get param_key	显示参数
rosparam load file_name	从文件加载参数
rosparam dump file_name	保存参数到文件

续表

rosparam 命令	作　　用
rosparam delete	删除参数
rosparam list	列出参数名称

load 和 dump 文件需要遵守 YAML 格式，YAML 格式具体示例如下：

```
name:'Zhangsan'
age:20
gender:'M'
score{Chinese:80,Math:90}
score_history:[85,82,88,90]
```

就是“名称：值”这样一种常用的解释方式。一般格式如下：

```
key : value
```

其实可以把 YAML 文件的内容理解为 Python 中的字典，采用键值对的形式。

3.6.2　launch 文件内读写参数

launch 文件中有很多标签，而与参数服务器相关的标签只有两个，一个是 < param >，另一个是 < rosparam >。这两个标签功能比较相近，但 < param > 一般只设置一个参数，示例如下：

```
<!-- 直接定义参数并赋值 -->
<param name = "velodyne_frame_id" type = "string" value = "velodyne"/>
<!-- 定义参数并通过 arg 给参数赋值 -->
<arg name = "velodyne_frame_id" default = "velodyne"/>
<param name = "velodyne_frame_id" type = "string" value = " $ (arg velodyne_frame_id)"/>
```

< rosparam >的典型用法是先指定一个 YAML 文件，然后添加 command，将参数批量写入参数服务器，其效果等价于 rosparam load file_name，示例如下：

```
<!-- 从../config/params.yaml 文件中加载参数 -->
<rosparam command = "load" file = " $ (find * package_name)/config/params.yaml" />
```

3.6.3　节点源码内读写参数

除了上述最常用的两种读写参数服务器的方法，还有一种方法就是修改 ROS 的源码，也就是利用 API 对参数服务器进行操作。C++节点中有两套 API 系统可以实现参数服务器的读写。

1. ros::NodeHandle

该方式的示例如下：

```
/**
 * C++中分别通过 ros::NodeHandle 实现参数服务器中参数的读写
 **/
#include "ros/ros.h"

int main(int argc, char *argv[])
{
```

```
    //初始化 ros 节点
    ros::init(argc,argv,"param_demo_node");

    //创建 ros 节点句柄
    ros::NodeHandle nh;
    string node_param;
    std::vector< std::string > param_names;

    //新增参数到参数服务器中
    nh.setParam("param_1", 1.0);

    //读取参数.如果参数服务器中参数"param_2"存在, 就把"param_2"的值传递给变量 node_param;
    //如果"param_2"不存在,就传递默认值"hahaha"
    nh.param < std::string >("param_2",node_param,"hahaha");

    //如果参数"param_2"存在,返回 true,并赋值给变量 node_param;如果"param_2"不存在,返回
    //false,不赋值
    nh.getParam("param_2", node_param);

    //其他功能
    nh.getParamNames(param_names);
    nh.hasParam("param_2");
    nh.searchParam("param_2",node_param);
}
```

2. ros::param

相关示例如下：

```
/**
 * C++中分别通过 ros::param 实现参数服务器中参数的读写
 **/

#include "ros/ros.h"

int main(int argc, char *argv[])
{
    //初始化 ros 节点
    ros::init(argc,argv,"param_demo_node");

    string node_param;
    std::vector< std::string > param_names;

    //新增参数到参数服务器中
    ros::param::set("param_1", 1.0);

    //读取参数.如果参数服务器中参数"param_2"存在,就把"param_2"的值传递给变量 node_param;
    //如果"param_2"不存在,就传递默认值"hahaha"
    node_param = ros::param::param("param_2","hahaha");

    //如果参数"param_2"存在,返回 true,并赋值给变量 node_param;如果"param_2"不存在,返回
    //false,不赋值
  ros::param::get("param_2", node_param);

    //其他功能
```

```
    ros::param::getParamNames(param_names);
    ros::param::has("param_2");
    ros::param::search("param_2",node_param);
}
```

3.7　通信机制比较

三种通信机制中，参数服务器是一种数据共享机制，可以在不同的节点之间共享数据，话题通信与服务通信是在不同的节点之间传递数据。这三种机制是 ROS 中基础的应用广泛的通信机制。其中话题通信和服务通信有一定的相似性，也有本质上的差异，因此有必要对二者进行简单比较。

二者的实现流程是比较相似的，都涉及以下四个要素：

(1) 消息的发布方/客户端(Publisher/Client)；

(2) 消息的订阅方/服务端(Subscriber/Server)；

(3) 话题名称(Topic/Service)；

(4) 数据载体(msg/srv)。

可以概括为：两个节点通过话题关联到一起，并通过某种类型的数据载体实现数据传输。二者的实现方式也有着较大差异，具体比较结果如表 3-3 所示。

表 3-3　话题及服务通信方式的比较

通信机制	通信模式	同步性	底层协议	缓冲区	实时性	节点关系	通信数据	使用场景
话题(Topic)	发布/订阅	异步	ROSTCP/ROSUDP	有	弱	多对多	msg	连续高频的数据发布与接收，如激光雷达、里程计
服务(Service)	请求/响应	同步	ROSTCP/ROSUDP	无	强	一对多(一个服务端)	srv	偶尔调用或执行某一项特定功能，如拍照、语音识别

本章小结

本章主要介绍了 ROS 中三种最基本、最核心的通信机制：话题通信、服务通信和参数服务器。对于每种通信机制都介绍了如下内容：

- 通信机制的理论模型；
- 通信机制的应用场景；
- 通信机制的 C++与 Python 的代码实现。

本章还对话题通信和服务通信进行了比较，对比了话题通信与服务通信的相同点及差异。另外，本章还给出了常用的 msg 和 srv 类型的定义，有助于读者将来在开发中参考使

用。通过本章内容的学习,读者可以理解 ROS 的大部分应用场景。

习题

1. 如何启动 ROS Master?
2. 分别阐述话题通信和服务通信机制的原理,并说明其应用场合。
3. 参数服务器的维护主要有哪几种方式?
4. ROS 中常用的数据通信方式有哪些?
5. 比较话题通信和服务通信的异同点。

实验

1. 编写程序,实现话题通信。
2. 编写程序,实现服务通信。