第3章

ROS 通信机制进阶

第 2 章主要介绍了 ROS 通信的实现，内容偏向于粗粒度的通信框架的讲解，没有详细介绍涉及的 API，也没有封装代码。鉴于此，本章主要内容如下：

- ROS 常用 API 介绍。
- ROS 中自定义头文件与源文件的使用。

本章预期达成的学习目标如下：

- 熟练掌握 ROS 常用 API。
- 掌握 ROS 中自定义头文件与源文件的配置。

3.1 常 用 API

首先，建议参考官方的以下 API 文档或源码：

- 与 ROS 节点的初始化相关的 API。
- 与 NodeHandle 的基本使用相关的 API。
- 与话题的发布方、订阅方对象相关的 API。
- 与服务的服务器端，客户端对象相关的 API。
- 与时间相关的 API。
- 与日志输出相关的 API。

与参数服务器相关的 API 在第 2 章已经做了详细介绍和应用，在此不再赘述。

3.1.1 初始化

C++ 实现代码如下：

```
/** @brief ROS 初始化函数。
 *
 * 该函数可以解析并使用节点启动时传入的参数(通过参数设置节点名称、命名空间等)
 *
 * 该函数有多个重载版本。如果使用 NodeHandle，建议调用本版本
 *
 * \param argc: 参数个数
 * \param argv: 参数列表
 * \param name: 节点名称，需要保证其唯一性，不允许包含命名空间
 * \param options: 节点启动选项，被封装进了 ros::init_options
```

```
 *
 */
void init(int &argc, char **argv, const std::string& name, uint32_t options = 0);
```

Python 实现代码如下：

```
Def init_node(name, argv=None, anonymous=False, log_level=None, disable_
rostime=False, disable_rosout=False, disable_signals=False, xmlrpc_port=0,
tcpros_port=0):
    """
    在 ROS msater 中注册节点

    @param name: 节点名称,必须保证唯一性,名称中不能使用命名空间(不能包含 "/")
    @type name: str

    @param anonymous: 取值为 true 时,为节点名称后缀随机编号
    @type anonymous: bool
    """
```

3.1.2 话题与服务相关对象

1. C++ 实现

在 roscpp 中,话题和服务的相关对象一般由 NodeHandle 创建。

NodeHandle 的一个重要作用是设置命名空间,这是后面的重点,本章暂不介绍。

1）发布对象

对象获取代码如下：

```
/**
 * \brief: 根据话题生成发布对象
 * 在 ROS Master 注册并返回一个发布方对象,该对象可以发布消息
 * 示例:
 * ros::Publisher pub = handle.advertise<std_msgs::Empty>("my_topic", 1);
 *
 * \param topic: 发布消息使用的话题
 *
 * \param queue_size: 等待发送给订阅方的最大消息数量
 *
 * \param latch (optional): 如果为 true,该话题发布的最后一条消息将被保存,并且后期
   当有订阅方连接时会将该消息发送给订阅方
 *
 * \return: 调用成功时,会返回一个发布对象
 */
template <class M>
Publisher advertise(const std::string& topic, uint32_t queue_size, bool latch =
false)
```

消息发布函数代码如下：

```
/**
 * 发布消息
 */
```

```
template <typename M>
void publish(const M& message) const
```

2）订阅对象

对象获取代码如下：

```
/**
 * \brief:生成某个话题的订阅对象
 * 该函数将根据给定的话题在 ROS Master 注册,并自动连接相同主题的发布方
 * 每接收到一条消息,都会调用回调函数,并且传入该消息的共享指针
 * 该消息不能被修改,因为可能其他订阅对象也会使用该消息
 * 示例:
 * void callback(const std_msgs::Empty::ConstPtr& message)
 * {
 * }
 * ros::Subscriber sub = handle.subscribe("my_topic", 1, callback);
 *
 * \param M [template]: M 是消息类型
 * \param topic: 订阅的话题
 * \param queue_size: 消息队列长度。超出长度时,头部的消息将被弃用
 * \param fp: 当订阅到一条消息时需要执行的回调函数
 * \return: 调用成功时返回一个订阅方对象,失败时返回空对象
 *
 * void callback(const std_msgs::Empty::ConstPtr& message){…}
 * ros::NodeHandle nodeHandle;
 * ros::Subscriber sub = nodeHandle.subscribe("my_topic", 1, callback);
 * if (sub)                                    //订阅者有效时进入
 * {
 *     …
 * }
 */
template<class M>
Subscriber subscribe(const std::string& topic, uint32_t queue_size, void( * fp)
(const boost::shared_ptr<M const> &), const TransportHints& transport_hints =
TransportHints())
```

3）服务对象

对象获取代码如下：

```
/**
 * \brief:生成服务器端对象
 * 该函数可以连接到 ROS Master,并提供一个具有给定名称的服务对象
 * 示例:
 * \verbatim
 * bool callback(std_srvs::Empty& request, std_srvs::Empty& response)
 * {
 *     return true;
 * }
 *  ros:: ServiceServer service = handle. advertiseService ( " my _ service ",
    callback);
 * \endverbatim
 *
```

```
 * \param service: 服务的主题名称
 * \param srv_func: 接收到请求时需要处理请求的回调函数
 * \return: 请求成功时返回服务对象,否则返回空对象:
 * 使用示例如下:
 * \verbatim
 * bool Foo::callback(std_srvs::Empty& request, std_srvs::Empty& response)
 * {
 *     return true;
 * }
 * ros::NodeHandle nodeHandle;
 * Foo foo_object;
 * ros::ServiceServer service = nodeHandle.advertiseService("my_service",
   callback);
 * if (service)                          //发布方的服务有效时进入
 * {
 *     ...
 * }
 * \endverbatim
 */
template<class MReq, class MRes>
ServiceServer advertiseService(const std::string& service, bool(*srv_func)
(MReq&, MRes&))
```

4）客户端对象

对象获取代码如下：

```
/**
 * @brief:创建一个服务的客户端对象
 * 当清除最后一个连接的引用句柄时,连接将被关闭
 * @param service_name(服务主题名称)
 */
template<class Service>
ServiceClient serviceClient(const std::string& service_name, bool persistent =
false, const M_string& header_values = M_string())
```

请求发送函数代码如下：

```
/**
 * @brief:发送请求
 * 返回值为 bool 类型,true 表示请求处理成功,false 表示请求处理失败
 */
template<class Service>
bool call(Service& service)
```

等待服务函数 1 代码如下：

```
/**
 * ros::service::waitForService("addInts");
 * \brief: 等待服务可用,否则一直处于阻塞状态
 * \param service_name: 被等待的服务的话题名称
 * \param timeout: 等待的最大时长,默认为-1,可以永久等待直至节点关闭
 * \return: 成功返回 true,否则返回 false
 */
```

```
ROSCPP_DECL bool waitForService(const std::string& service_name, ros::Duration
timeout = ros::Duration(-1));
```

等待服务函数 2 代码如下：

```
/**
 * client.waitForExistence();
 * \brief: 等待服务可用,否则一直处于阻塞状态
 * \param timeout: 等待最大时长,默认为-1,可以永久等待直至节点关闭
 * \return: 成功返回 true,否则返回 false。
 */
bool waitForExistence(ros::Duration timeout = ros::Duration(-1));
```

2. Python 实现

1）发布对象

对象获取代码如下：

```
class Publisher(Topic):
    """
    在 ROS Master 注册为相关话题的发布方
    """

    def __init__(self, name, data_class, subscriber_listener=None, tcp_nodelay
=False, latch=False, headers=None, queue_size=None):
        """
        Constructor
        @param name: 话题名称
        @type name: str
        @param data_class: 消息类型
        @param latch: 如果为 true,该话题发布的最后一条消息将被保存,并且后期当有订阅
         方连接时会将该消息发送给订阅方
        @type latch: bool
        @param queue_size: 等待发送给订阅方的最大消息数量
        @type queue_size: int
        """
```

消息发布函数代码如下：

```
def publish(self, *args, **kwds):
        """
        发布消息
        """
```

2）订阅对象

对象获取代码如下：

```
class Subscriber(Topic):
    """
    类注册为指定主题的订阅方,其中消息是给定类型的
    """
    def __init__(self,name, data_class, callback=None, callback_args=None,
queue_size=None,buff_size=DEFAULT_BUFF_SIZE, tcp_nodelay=False):
```

```
        """
        Constructor.
        @param name: 话题名称
        @type name: str
        @param data_class: 消息类型
        @type data_class: L{Message} class
        @param callback: 处理订阅到的消息的回调函数
        @type callback: fn(msg, cb_args)
        @param queue_size: 消息队列长度,超出长度时,头部的消息将被弃用
        """
```

3）服务对象

对象获取代码如下：

```
class Service(ServiceImpl):
    """
    声明一个 ROS 服务

    示例:
    s = Service('getmapservice', GetMap, get_map_handler)
    """
    def __init__(self, name, service_class, handler,
                 buff_size=DEFAULT_BUFF_SIZE, error_handler=None):
        """
        @param name: 服务主题名称(``str``)
        @param service_class:服务消息类型
        @param handler: 回调函数,处理请求数据,并返回响应数据
        @type handler: fn(req)->resp
        """
```

4）客户端对象

对象获取代码如下：

```
class ServiceProxy(_Service):
    """
    创建一个 ROS 服务的句柄
    示例:
    add_two_ints = ServiceProxy('add_two_ints', AddTwoInts)
    resp = add_two_ints(1, 2)
    """
    def __init__(self, name, service_class, persistent=False, headers=None):
        """
        ctor.
        @param name: 服务主题名称
        @type name: str
        @param service_class: 服务消息类型
        @type service_class: Service class
        """
```

请求发送函数代码如下：

```
def call(self, *args, **kwds):
        """
```

```
        发送请求,返回值为响应数据
        """
```

等待服务函数代码如下：

```
def wait_for_service(service, timeout=None):
    """
    调用该函数时,程序会处于阻塞状态,直到服务可用
    @param service: 被等待的服务话题名称
    @type service: str
    @param timeout: 超时时间
    @type timeout: double|rospy.Duration
    """
```

3.1.3　回旋函数

1. C++ 实现

在 ROS 程序中,频繁地使用 ros::spin()和 ros::spinOnce()两个回旋函数,可以用于处理回调函数。

spinOnce()用法如下：

```
/**
 * \brief: 处理一轮回调
 * 一般应用场景:
 *     在循环体内,处理所有可用的回调函数
 */
ROSCPP_DECL void spinOnce();
```

spin()用法如下：

```
/**
 * \brief: 进入循环处理回调
 */
ROSCPP_DECL void spin();
```

二者的相同点是都用于处理回调函数。

二者的不同点：ros::spin()是进入循环执行回调函数,而 ros::spinOnce()只会执行一次回调函数(没有循环);ros::spin()后的语句不会执行,而 ros::spinOnce()后的语句可以执行。

2. Python 实现

spin()用法如下：

```
def spin():
    """
    进入循环处理回调
    """
```

3.1.4　时间

ROS 中与时间相关的 API 极其常用,例如获取当前时刻、持续时间的设置、执行频率、

休眠、定时器等，都与时间相关。

1. C++ 实现

1）时刻

获取时刻或者设置指定时刻：

```
ros::init(argc,argv,"hello_time");
ros::NodeHandle nh;          //必须创建句柄，否则时间没有初始化，导致后续 API 调用失败
ros::Time right_now = ros::Time::now();       //将当前时刻封装成对象
ROS_INFO("当前时刻:%.2f",right_now.toSec()); //获取距 1970 年 1 月 1 日 00:00:00 的秒数
ROS_INFO("当前时刻:%d",right_now.sec);        //获取距 1970 年 1 月 1 日 00:00:00 的秒数
ros::Time someTime(100,100000000);            //参数 1 单位为秒，参数 2 单位为纳秒
ROS_INFO("时刻:%.2f",someTime.toSec());      //100.10
ros::Time someTime2(100.3);                   //直接传入 double 类型的秒数
ROS_INFO("时刻:%.2f",someTime2.toSec());     //100.30
```

2）持续时间

设置一个时间区间（间隔）：

```
ROS_INFO("当前时刻:%.2f",ros::Time::now().toSec());
ros::Duration du(10);                    //持续 10s，参数是 double 类型的，以秒为单位
du.sleep();                              //按照指定的持续时间休眠
ROS_INFO("持续时间:%.2f",du.toSec());  //将持续时间换算成秒
ROS_INFO("当前时刻:%.2f",ros::Time::now().toSec());
```

3）持续时间与时刻运算

为了方便使用，ROS 中提供了时间与时刻的运算：

```
ROS_INFO("时间运算");
ros::Time now = ros::Time::now();
ros::Duration du1(10);
ros::Duration du2(20);
ROS_INFO("当前时刻:%.2f",now.toSec());
//1.time 与 duration 运算
ros::Time after_now = now + du1;
ros::Time before_now = now - du1;
ROS_INFO("当前时刻之后:%.2f",after_now.toSec());
ROS_INFO("当前时刻之前:%.2f",before_now.toSec());
//2.duration 之间的运算
ros::Duration du3 = du1 + du2;
ros::Duration du4 = du1 - du2;
ROS_INFO("du3 = %.2f",du3.toSec());
ROS_INFO("du4 = %.2f",du4.toSec());
//time 之间不可以相加
//ros::Time nn = now + before_now;            //异常
ros::Duration du5 = now - before_now;
ROS_INFO("时刻相减:%.2f",du5.toSec());
```

4）设置运行频率

```
ros::Rate rate(1);                           //指定频率
while (true)
{
```

```
    ROS_INFO("-----------code----------");
    rate.sleep();                              //休眠,休眠时间=1/频率
}
```

5）定时器

ROS 中内置了专门的定时器,可以实现与 ros::Rate 类似的效果：

```
ros::NodeHandle nh;          //必须创建句柄,否则时间没有初始化,导致后续 API 调用失败
//ROS 定时器
/**
 * \brief: 创建一个定时器,按照指定频率调用回调函数
 * \param period: 时间间隔
 * \param callback: 回调函数
 * \param oneshot: 如果设置为 true,只执行一次回调函数;否则循环执行
 * \param autostart: 如果为 true,返回已经启动的定时器;否则需要手动启动
 */
//Timer createTimer (Duration period, const TimerCallback& callback, bool
oneshot = false, bool autostart = true) const;
//ros::Timer timer =nh.createTimer(ros::Duration(0.5),doSomeThing);
ros::Timer timer=nh.createTimer(ros::Duration(0.5),doSomeThing,true);
                             //只执行一次
//ros::Timer timer = nh.createTimer(ros::Duration(0.5), doSomeThing, false,
false);                      //需要手动启动
//timer.start();
ros::spin();                 //必须回旋
```

定时器的回调函数代码如下：

```
void doSomeThing(const ros::TimerEvent &event){
    ROS_INFO("--------------");
ROS_INFO("event:%s",std::to_string(event.current_real.toSec()).c_str());
}
```

2. Python 实现

1）时刻

获取时刻,或是设置指定时刻：

```
# 获取当前时刻
right_now = rospy.Time.now()
rospy.loginfo("当前时刻:%.2f",right_now.to_sec())
rospy.loginfo("当前时刻:%.2f",right_now.to_nsec())
# 自定义时刻
some_time1 = rospy.Time(1234.567891011)
some_time2 = rospy.Time(1234,567891011)
rospy.loginfo("设置时刻 1:%.2f",some_time1.to_sec())
rospy.loginfo("设置时刻 2:%.2f",some_time2.to_sec())
# 从时间创建对象
# some_time3 = rospy.Time.from_seconds(543.21)
some_time3 = rospy.Time.from_sec(543.21) # from_sec 替换了 from_seconds
rospy.loginfo("设置时刻 3:%.2f",some_time3.to_sec())
```

2）持续时间

设置一个时间区间(间隔)：

```
# 与持续时间相关的 API
rospy.loginfo("持续时间测试开始.....")
du = rospy.Duration(3.3)
rospy.loginfo("du1 持续时间:%.2f",du.to_sec())
rospy.sleep(du) #休眠函数
rospy.loginfo("持续时间测试结束.....")
```

3）持续时间与时刻运算

为了方便使用，ROS 提供了时间与时刻的运算：

```
rospy.loginfo("时间运算")
now = rospy.Time.now()
du1 = rospy.Duration(10)
du2 = rospy.Duration(20)
rospy.loginfo("当前时刻:%.2f",now.to_sec())
before_now = now - du1
after_now = now + du1
dd = du1 + du2
# now = now + now #非法
rospy.loginfo("之前时刻:%.2f",before_now.to_sec())
rospy.loginfo("之后时刻:%.2f",after_now.to_sec())
rospy.loginfo("持续时间相加:%.2f",dd.to_sec())
```

4）设置运行频率

```
# 设置执行频率
rate = rospy.Rate(0.5)
while not rospy.is_shutdown():
    rate.sleep() #休眠
    rospy.loginfo("+++++++++++++++")
```

5）定时器

ROS 内置了专门的定时器，可以实现与 ros::Rate 类似的效果：

```
#定时器设置
"""
def __init__(self, period, callback, oneshot=False, reset=False):
    Constructor.
    @param period: 回调函数的时间间隔
    @type period: rospy.Duration
    @param callback: 回调函数
    @type callback: function taking rospy.TimerEvent
    @param oneshot: 设置为 True,就只执行一次;否则循环执行
    @type oneshot: bool
    @param reset: if True, timer is reset when rostime moved backward. [default:
False]
    @type  reset: bool
"""
rospy.Timer(rospy.Duration(1),doMsg)
# rospy.Timer(rospy.Duration(1),doMsg,True)                    # 只执行一次
rospy.spin()
```

回调函数代码如下：

```
def doMsg(event):
    rospy.loginfo("++++++++++")
    rospy.loginfo("当前时刻:%s",str(event.current_real))
```

3.1.5　其他函数

在发布实现时，一般会循环发布消息，循环的判断条件一般由节点状态控制，在 C++ 中可以通过 ros::ok()判断节点状态是否正常，而在 Python 中则通过 rospy.is_shutdown()实现判断。导致节点退出的原因主要有如下几种：

- 节点接收到了关闭信息，例如常用的 Ctrl+C 组合键就会发出关闭节点的信号。
- 同名节点启动，导致现有节点退出。
- 程序中的其他部分调用了与节点关闭相关的 API(在 C++ 中是 ros::shutdown()，在 Python 中是 rospy.signal_shutdown())。

另外，与日志相关的函数也是极其常用的。在 ROS 中日志被划分成如下 5 个级别：

- DEBUG(调试)：只在调试时使用，此类消息不会输出到控制台。
- INFO(信息)：标准消息，一般用于说明系统内正在执行的操作。
- WARN(警告)：提醒一些异常情况，但程序仍然可以执行。
- ERROR(错误)：提示错误信息，此类错误会影响程序运行。
- FATAL(严重错误)：此类错误将阻止节点继续运行。

1. C++ 实现

1) 节点状态判断

```
/** \brief: 检查节点是否已经退出
 *   ros::shutdown(): 被调用且执行完毕后,该函数将会返回 false
 * \return: 返回 true 表示节点还存在, 返回 false 表示节点已经被关闭
 */
bool ok();
```

2) 节点关闭函数

```
/*
 *   关闭节点
 */
void shutdown();
```

3) 日志函数

使用示例如下：

```
ROS_DEBUG("hello,DEBUG");                    //不会输出
ROS_INFO("hello,INFO");                      //默认白色字体
ROS_WARN("Hello,WARN");                      //默认黄色字体
ROS_ERROR("hello,ERROR");                    //默认红色字体
ROS_FATAL("hello,FATAL");                    //默认红色字体
```

2. Python 实现

1) 节点状态判断

```
def is_shutdown():
    """
    @return: 返回 True 表示节点已经被关闭
    @rtype: bool
    """
```

2）节点关闭函数

```
def signal_shutdown(reason):
    """
    关闭节点
    @param reason: 节点关闭的原因,是一个字符串
    @type reason: str
    """
def on_shutdown(h):
    """
    节点被关闭时调用的函数
    @param h: 关闭时调用的回调函数,此函数无参数
    @type h: fn()
    """
```

3）日志函数

使用示例如下：

```
rospy.logdebug("hello,debug")                #不会输出
rospy.loginfo("hello,info")                  #默认白色字体
rospy.logwarn("hello,warn")                  #默认黄色字体
rospy.logerr("hello,error")                  #默认红色字体
rospy.logfatal("hello,fatal")                #默认红色字体
```

3.2 ROS 中的头文件与源文件

本节主要介绍在 ROS 的 C++ 实现中如何使用头文件与源文件的方式封装代码，具体内容如下：

（1）设置头文件，以可执行文件作为源文件。

（2）分别设置头文件、源文件与可执行文件。

在 ROS 中，关于头文件的使用，核心内容在于 CMakeLists.txt 文件的配置。不同的封装方式在配置上也有差异。

3.2.1 自定义头文件调用

需求：设计头文件，以可执行文件作为源文件。

流程：

（1）编写头文件。

（2）编写可执行文件（同时也是源文件）。

（3）编辑配置文件并执行。

1. 头文件

在功能包下的 include/功能包名目录下新建头文件 hello.h，示例内容如下：

```
#ifndef _HELLO_H
#define _HELLO_H
namespace hello_ns{
    class HelloPub {
        public:
            void run();
    };
}
#endif
```

注意：在 VSCode 中，为后续包含头文件不抛出异常，应配置.vscode 下 c_cpp_properties.json 的 includepath 属性，格式如下：

```
"/home/用户/工作空间/src/功能包/include/**"
```

2. 可执行文件

在 src 目录下新建文件 hello.cpp，示例内容如下：

```
#include "ros/ros.h"
#include "test_head/hello.h"
namespace hello_ns {
    void HelloPub::run(){
        ROS_INFO("自定义头文件的使用....");
    }
}
int main(int argc, char * argv[])
{
    setlocale(LC_ALL,"");
    ros::init(argc,argv,"test_head_node");
    hello_ns::HelloPub helloPub;
    helloPub.run();
    return 0;
}
```

3. 配置文件

配置 CMakeLists.txt 文件，头文件相关配置如下：

```
include_directories(
include
  ${catkin_INCLUDE_DIRS}
)
```

可执行配置文件配置方式与前面一致：

```
add_executable(hello src/hello.cpp)
add_dependencies(hello${${PROJECT_NAME}_EXPORTED_TARGETS}${catkin_EXPORTED_
TARGETS})
target_link_libraries(hello
  ${catkin_LIBRARIES}
)
```

最后，编译并执行，控制台可以输出自定义的文本信息。

3.2.2 自定义源文件调用

需求：设计头文件与源文件，在可执行文件中包含头文件。

流程：

(1) 编写头文件。

(2) 编写源文件。

(3) 编写可执行文件。

(4) 编辑配置文件并执行。

1. 头文件

头文件设置与 3.2.1 节类似，在功能包下的 include/功能包名目录下新建头文件 haha.h，示例内容如下：

```
#ifndef _HAHA_H
#define _HAHA_H
namespace hello_ns {
    class My {
    public:
        void run();
    };
}
#endif
```

注意：在 VSCode 中，为后续包含头文件不抛出异常，应配置.vscode 下 c_cpp_properties.json 的 includepath 属性，格式如下：

```
"/home/用户/工作空间/src/功能包/include/**"
```

2. 源文件

在 src 目录下新建文件 haha.cpp，示例内容如下：

```
#include "test_head_src/haha.h"
#include "ros/ros.h"
namespace hello_ns{
    void My::run(){
        ROS_INFO("hello,head and src ...");
    }
}
```

3. 可执行文件

在 src 目录下新建文件 use_head.cpp，示例内容如下：

```
#include "ros/ros.h"
#include "test_head_src/haha.h"
int main(int argc, char *argv[])
{
    ros::init(argc,argv,"hahah");
    hello_ns::My my;
```

```
    my.run();
    return 0;
}
```

4. 配置文件

头文件与源文件的相关配置如下：

```
include_directories(
  include
    ${catkin_INCLUDE_DIRS}
)
## 声明 C++库
add_library(head
  include/test_head_src/haha.h
  src/haha.cpp
)
add_dependencies(head${${PROJECT_NAME}_EXPORTED_TARGETS}${catkin_EXPORTED_
TARGETS})
target_link_libraries(head
  ${catkin_LIBRARIES}
)
```

可执行文件配置如下：

```
add_executable(use_head src/use_head.cpp)
add_dependencies(use_head${${PROJECT_NAME}_EXPORTED_TARGETS}${catkin_EXPORTED
_TARGETS})
# 此处需要添加前面设置的 head 库
target_link_libraries(use_head
  head
  ${catkin_LIBRARIES}
)
```

3.3　Python 模块导入

与 C++ 类似的，在 Python 中导入其他模块时也需要相关处理。

需求：首先新建 Python 文件 A，再新建 Python 文件 UseA，在 UseA 中导入 A 并调用 A 的实现。

流程：

(1) 新建两个 Python 文件，使用 import 实现导入关系。

(2) 添加可执行权限，编辑配置文件并执行 UseA。

1. 新建两个 Python 文件并使用 import 导入

文件 A 的实现如下(包含一个变量)：

```
#! /usr/bin/env python
num = 1000
```

文件 UseA 的核心实现如下：

```
import os
import sys
path = os.path.abspath(".")
# 核心
sys.path.insert(0,path + "/src/plumbing_pub_sub/scripts")
import tools
…
…
rospy.loginfo("num = %d",tools.num)
```

2. 添加可执行权限，编辑配置文件并执行

此过程略。

3.4 本章小结

本章主要介绍了 ROS 的常用 API、ROS 中的自定义头文件与源文件的使用以及 Python 模块的导入。本章内容比较简单，多加练习即可。