第三章、ROS通信进阶

一、常用API

官方链接：http://wiki.ros.org/APIs

1.初始化

C++

```
/** @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...)
 * 该函数有多个重载版本，如果使用NodeHandle建议调用该版本。
 * \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、解决结点名称重名
 实例:
 init(argc, argv, "Node", ros::init_options::AnonymousName);
 */
void init(int &argc, char **argv, const std::string& name, uint32_t
options = 0);
```

Python

```
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，必须保证节点名称唯一，节点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
"""
```

2.话题服务相关

C++

1. **发布对象**

对象获取

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

发布消息

```
/**
* 发布消息
*/
template <typename M>
    void publish(const M& message) const
```

2. **订阅对象**

对象获取

```
/**
    * \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。
    * 使用示例如下:
void callback(const std_msgs::Empty::ConstPtr& message)
{
}
ros::Subscriber sub = handle.subscribe("my_topic", 1, callback);
* \param M [template] M 是指消息类型
* \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，返回一个订阅者对象，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) // Enter if subscriber is valid
{
...
}
*/
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

```
/**
* \brief 生成服务端对象
* 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。
* 使用示例如下:
\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:
\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) // Enter if advertised service is valid
{
...
}
\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

```
/**
  * @brief 创建一个服务客户端对象
  * 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

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

等待服务函数

```
/**       1
 * \brief 等待服务可用，否则一致处于阻塞状态
 * \param service_name 被"等待"的服务的话题名称
 * \param timeout 等待最大时常，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。
 实例:
 ros::service::waitForService("addInts");
 */
ROSCPP_DECL bool waitForService(const std::string& service_name,
ros::Duration timeout = ros::Duration(-1));
/**        2
* \brief 等待服务可用，否则一致处于阻塞状态
* \param timeout 等待最大时常，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。
实例:
client.waitForExistence();
*/
bool waitForExistence(ros::Duration timeout = ros::Duration(-1));
```

3.回旋函数

1. `spinOnce();`

```
/**
 * \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();
```

2. `spin()`

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

相同点: 二者都用于处理回调函数;

不同点: ros::spin() 是进入了循环执行回调函数，而 ros::spinOnce() 只会执行一次回调函数(没有循环)，在 ros::spin() 后的语句不会执行到，而 ros::spinOnce() 后的语句可以执行。

4.时间

1. *时刻*

- 获取时刻、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());//获取距离 1970年01月01日 00:00:00 的秒数
ROS_INFO("当前时刻:%d",right_now.sec);//获取距离 1970年01月01日 00:00:00 的秒数
// 设置指定时间
ros::Time n1(100,100000000);// 参数1:秒数  参数2:纳秒
ROS_INFO("时刻:%.2f",n1.toSec()); //100.10
ros::Time n2(100.3);//直接传入 double 类型的秒数
ROS_INFO("时刻:%.2f",n2.toSec()); //100.30
```

2. *持续时间*

- 让程序停顿

```
ROS_INFO("当前时刻:%.2f",ros::Time::now().toSec());
ros::Duration du(10);//持续10秒钟,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*

```
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 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 之间相互运算
ros::Duration du3 = du1 + du2;
```

```
ros::Duration du4 = du1 - du2;
ROS_INFO("du3 = %.2f",du3.toSec());
ROS_INFO("du4 = %.2f",du4.toSec());
//PS: time 与 time 不可以运算
// ros::Time nn = now + before_now;//异常
```

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::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，设置为 false,就循环执行。
* \param autostart 如果为true，返回已经启动的定时器,设置为 false，需要手动启动,用start()函数启动。
*/
/*
Timer createTimer(Duration period, void(T::*callback)(const TimerEvent&) const, T* obj, bool oneshot = false, bool autostart = true) const
{
    return createTimer(period, boost::bind(callback, obj, boost::placeholders::_1), oneshot, autostart);
}
*/
void doSomeThing(const ros::TimerEvent event){
    ROS_INFO("------------------");
}
// 循环执行回调函数
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 spin
```

案例:

```
#include "ros/ros.h"
using namespace ros;
void doSomeThing(const TimerEvent& event){
    ROS_INFO("------------------");
```

```
    ROS_INFO("时间%.2f",event.current_real.toSec());
}
int main(int argc, char *argv[])
{
    setlocale(LC_ALL,"");
    init(argc,argv,"time_demo");
    NodeHandle nh;
    // 获取现在的时间
    Time right_now = Time::now();
    ROS_INFO("当前时刻%.2f",right_now.toSec());
    ROS_INFO("当前时刻%d",right_now.sec);
    // 创建一个时间变量，接收时间
    Time n1(10.236);
    ROS_INFO("当前时刻%.2f",n1.toSec());
    // 时间段
    ROS_INFO("开始");
    right_now = Time::now();
    ROS_INFO("当前时刻%.2f",right_now.toSec());
    Duration du(5);
    du.sleep();
    right_now = Time::now();
    ROS_INFO("当前时刻%.2f",right_now.toSec());
    ROS_INFO("持续的时间%0.2f",du.toSec());
    // 定时器
    Timer nm = nh.createTimer(du,doSomeThing);
    spin();
    return 0;
}
```

```
^Czhf@zhf:~/Learn_ROS/demo03_ws$ rosrun plumbing_apis de
apis_time
[ INFO] [1612375386.434397290]: 当前时刻1612375386.43
[ INFO] [1612375386.435292053]: 当前时刻1612375386
[ INFO] [1612375386.435306613]: 当前时刻10.24
[ INFO] [1612375386.435313738]: 开始
[ INFO] [1612375386.435320710]: 当前时刻1612375386.44
[ INFO] [1612375391.450051777]: 当前时刻1612375391.45
[ INFO] [1612375391.450111115]: 持续的时间5.00
[ INFO] [1612375396.452220413]: ------------------
[ INFO] [1612375396.452627370]: 时间1612375396.45
[ INFO] [1612375401.450931694]: ------------------
[ INFO] [1612375401.450982635]: 时间1612375401.45
[ INFO] [1612375406.450715418]: ------------------
[ INFO] [1612375406.451081075]: 时间1612375406.45
[ INFO] [1612375411.450468110]: ------------------
[ INFO] [1612375411.450547517]: 时间1612375411.45
[ INFO] [1612375416.457989841]: ------------------
[ INFO] [1612375416.458138535]: 时间1612375416.46
^Czhf@zhf:~/Learn_ROS/demo03_ws$
```

5.补、其他函数

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 true 如果节点还健在，false 如果节点已经火化了。
 */
bool ok();
```

2. 节点关闭函数

```
/*
*   关闭节点
*/
void shutdown();
// 调用：
ros::shutdown();
```

3. 日志函数

```
ROS_DEBUG("hello,DEBUG"); //不会输出,调试
ROS_INFO("hello,INFO"); //默认白色字体,标准消息
ROS_WARN("Hello,WARN"); //默认黄色字体,警告
ROS_ERROR("hello,ERROR");//默认红色字体,错误
ROS_FATAL("hello,FATAL");//默认红色字体,严重错误
```

二、ROS中的头文件与源文件

1.自定义头文件并调用

- 流程

> 1. 编写头文件
> 2. 编写可执行问价
> 3. 配置文件并执行

1. 头文件

在`include`文件夹下创建头文件

在`src`文件夹下配置源文件

- 头文件

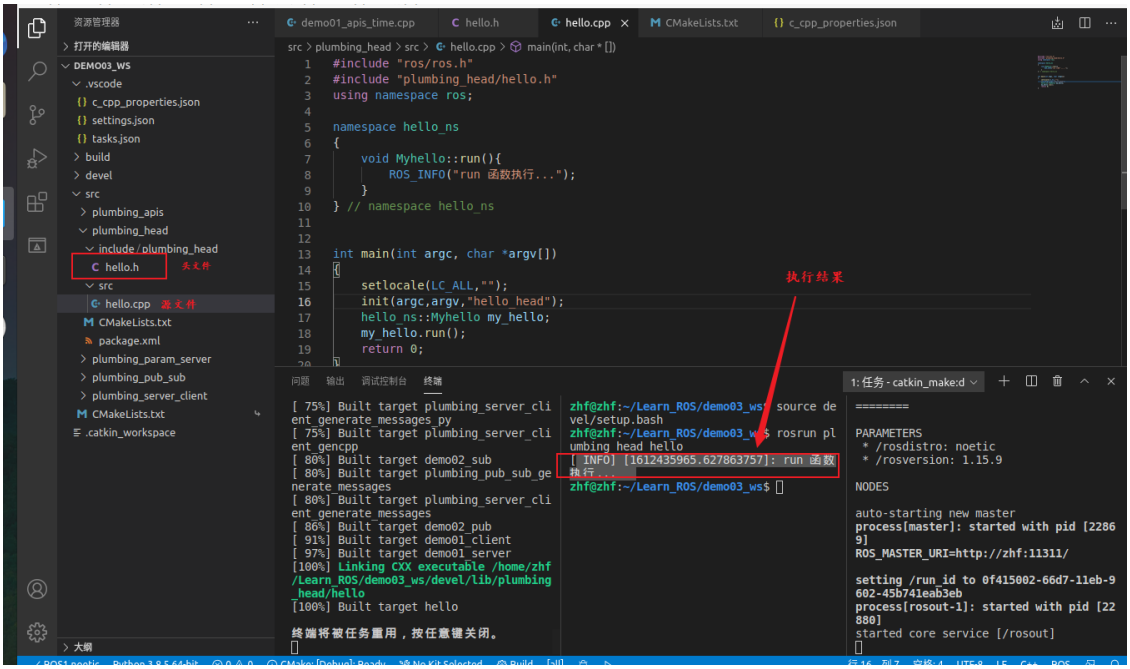
```
#ifndef __HELLO_H_
#define __HELLO_H_
/*
    声明
*/
namespace hello_ns {
    class Myhello {
    public:
        void run();
    };
} // namespace hello_ns
#endif
```

- 源文件

```
#include "ros/ros.h"
#include "plumbing_head/hello.h"
using namespace ros;
namespace hello_ns{
    void Myhello::run(){
        ROS_INFO("run 函数执行...");
    }
} // namespace hello_ns
int main(int argc, char *argv[]){
    setlocale(LC_ALL,"");
    init(argc,argv,"hello_head");
    hello_ns::Myhello my_hello;
    my_hello.run();
    return 0;
}
```

- 配置编译文件

```
include_directories(
include # 放开
  ${catkin_INCLUDE_DIRS}
)
add_executable(hello src/hello.cpp)
target_link_libraries(hello
  ${catkin_LIBRARIES}
)
```



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

- 头文件(和上面的一样)
- 源文件

```
#include "plumbing_head_src/hello.h"
#include "ros/ros.h"
namespace hello_ns{
    void Myhello::run(){
        ROS_INFO("hello_run函数...");
    }
}
```

- 执行文件

```
#include "plumbing_head_src/hello.h"
#include "ros/ros.h"
int main(int argc, char *argv[])
{
    setlocale(LC_ALL,"");
    hello_ns::Myhello he;
    he.run();
    return 0;
}
```

- 配置文件

```
include_directories(
include
  ${catkin_INCLUDE_DIRS}
)
add_library(head_src
  include/${PROJECT_NAME}/hello.h
  src/hello.cpp
)
add_dependencies(head_src ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(head_src
  ${catkin_LIBRARIES}
)
add_dependencies(user_hello ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
add_executable(user_hello src/user_hello.cpp)
target_link_libraries(user_hello
  head_src
  ${catkin_LIBRARIES}
)
```

- 运行结果：