**4 ROS 常用API介绍**

上一章内容，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。

预期达成的学习目标:

* 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。

# 4.1 常用API

首先，建议参考官方API文档或参考源码:

官方网站：wiki.ros.org 和wiki.ros.org/APIs

* ROS节点的初始化相关API;
* NodeHandle 的基本使用相关API;
* 话题的发布方，订阅方对象相关API;
* 服务的服务端，客户端对象相关API;
* 时间相关API;
* 日志输出相关API。

参数服务器相关API在第二章已经有详细介绍和应用，在此不再赘述。

# 4.1.1 初始化

### C++

### 初始化

/\*\* @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

\*

\*/

void init(int &argc, char \*\*argv, const std::string& name, uint32\_t options = 0);

# 4.1.2 话题与服务相关对象

## C++

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

NodeHandle有一个重要作用是可以用于设置命名空间。

### 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

\*

\* 使用示例如下:

\*

\* 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 等待发送给订阅者的最大消息数量

\*

\* \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

\*

\*

\*/

template <class M>

Publisher advertise(const std::string& topic, uint32\_t queue\_size, bool latch = false)

**消息发布函数:**

/\*\*

\* 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:

/\*\*

\* ros::service::waitForService("addInts");

\* \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。

\*/

ROSCPP\_DECL bool waitForService(const std::string& service\_name, ros::Duration timeout = ros::Duration(-1));

### 等待服务函数2:

/\*\*

\* client.waitForExistence();

\* \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。

\*/

bool waitForExistence(ros::Duration timeout = ros::Duration(-1));

# 4.1.3 回旋函数

## C++

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

### 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();

### 3.二者比较

**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。

# 4.1.4 时间

ROS中时间相关的API是极其常用，比如:获取当前时刻、持续时间的设置、执行频率、休眠、定时器...都与时间相关。

### 1.时刻

获取时刻，或是设置指定时刻:

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

ros::Time someTime(100,100000000);// 参数1:秒数 参数2: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

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 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，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。

\*/

//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 spin

定时器的回调函数:

void doSomeThing(const ros::TimerEvent &event){

ROS\_INFO("-------------");

ROS\_INFO("event:%s",std::to\_string(event.current\_real.toSec()).c\_str());

}

# 4.1.5 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来控制，C++中可以通过 ros::ok() 来判断节点状态是否正常，而 python 中则通过 rospy.is\_shutdown() 来实现判断，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，比如常用的 ctrl + c 快捷键就是关闭节点的信号；
* 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节点关闭相关的API(C++中是ros::shutdown()，python中是rospy.signal\_shutdown())

另外，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.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();
* 3.日志函数
* 使用示例
* 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