**初始化API**

\* \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)

**发布对象**

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

##### 订阅对象

##### 使用示例

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

{

...

}

##### 服务对象

##### 使用示例

bool callback(std\_srvs::Empty& request, std\_srvs::Empty& response)

{

return true;

}

\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

{

...

}

##### 客户端对象

##### 等待服务函数

client.waitForExistence();

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

##### 回旋函数

##### spin()

##### spinOnce()

**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。

**时间函数**

**1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

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 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());

}

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

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