ROS节点通信中发布器publisher和订阅器subscriber模块

[1. ROS工作空间创建 2](#_Toc16701)

[2. ROS\_work\_space\_package\_node\_topic的关系 2](#_Toc4368)

[3. 编写发布器代码 3](#_Toc22737)

[4. 编辑订阅器代码 5](#_Toc22735)

[5. CMakeLists\_txt文件 7](#_Toc7236)

[(1) 确认原始CMakeLists.txt文件 7](#_Toc27659)

[(2) 在 CMakeLists.txt 文件末尾加入几条语句: 7](#_Toc16630)

[(3) 说明 7](#_Toc32096)

[6. 编译及运行流程 7](#_Toc16077)

[(1) 进入catkin\_test项目文件夹的根目录； 7](#_Toc24053)

[(2) source ./devel/setup.bash 7](#_Toc16353)

[(3) 输入catkin\_make命令，编译源码； 7](#_Toc24291)

[(4) 设置环境变量： source ./devel/setup.bash 7](#_Toc27329)

[(5) 启动ROS Master：roscore 7](#_Toc9962)

[(6) 启动Publisher: rosrun beginner\_tutorials talker 8](#_Toc17361)

[(7) 启动Subscriber：rosrun beginner\_tutorials listener 8](#_Toc26339)

[(8) 查看例程的节点关系: rqt\_graph 8](#_Toc1671)

[7. 出现问题及解决 8](#_Toc27267)

[8. 单功能包内多节点及多主题生成及传输 8](#_Toc7563)

[(1) 生成主题信息的节点代码talker\_cpp 8](#_Toc12393)

[(2) 接收主题信息的节点1代码listener\_cpp 9](#_Toc22809)

[(3) 接收主题信息的节点2代码listener2\_cpp 9](#_Toc19356)

[(4) CMakeLists\_txt 10](#_Toc15624)

[(5) 运行结果图 10](#_Toc8664)

[(6) 最简单的启动一个节点的Launch文件 12](#_Toc19550)

[(7) 启动多个节点的Launch文件 12](#_Toc30043)

[9. ROS入门之launch 12](#_Toc9835)

[(1) 何为launch 12](#_Toc24086)

[(2) 最简单的launch文件样例 13](#_Toc10676)

[(3) launch标签小结 13](#_Toc13678)

[10. 参考文献 15](#_Toc13110)

# ROS工作空间创建

参考<<[胡春旭. 2021-ROS机器人开发实践](https://www.baidu.com/link?url=RqEFkD376cM-ikOjeYZFgrLQSeJcmrX6t3RkwfAL04-k1uLqY67rXqX6tSqqbUCimjhT7e-K0suFbessM-QIoHRYaZGV9vZhAlkAC6haBOe&wd=&eqid=aac40035001a24af0000000462afd9ef" \t "https://www.baidu.com/_blank)>>section3.2部分。

原文链接：https://blog.csdn.net/qq\_24624539/article/details/102755619

安装完成ROS以后会有一个catkin\_ws文件夹，即为ROS工作空间。

[1]创建并初始化一个新的工作空间catkin\_ws

mkdir -p ~/catkin\_ws/src

cd ~/catkin\_ws/src

catkin\_init\_workspace

catkin\_init\_workspace命令把当前目录初始化为一个ROS工作空间。

[2]编译该工作空间。

cd ~/catkin\_ws

catkin\_make

[3]定义catkin\_ws空间所需要的环境变量。执行此命令后，ros的相关命令（如roscd等）可以找到此工作空间中的package。

source ~/catkin\_ws/devel/setup.bash

[4]验证ROS工作空间的环境变量加载成功：

echo $ROS\_PACKAGE\_PATH

[5] create function package

enter the src directory of function package:

catkin\_create\_pkg beginner\_tutorials std\_msgs rospy roscpp

enter the root directory:

catkin\_make

source ./devel/setup.bash

然后运行rosrun启动节点命令，或roslaunch批量化启动节点命令。

[5]打开环境变量文件sudo gedit .bashrc 加入下面一行：

source ~/catkin\_ws/devel/setup.bash

source ~/.bashrc

[6]最终的环境变量

source /opt/ros/melodic/setup.bash

source ~/catkin\_ws/devel/setup.bash

原文链接：<https://blog.csdn.net/weixin_43297891/article/details/115426161>

# ROS\_work\_space\_package\_node\_topic的关系

ROS中的工作空间（work space）是ROS中非常重要的一个概念，可以把工作空间理解为一个大的工厂，里面的分成几个大的生产车间（package），每一个生产车间中会有若干个具有不同技能的工人（node）。当工厂运转时，每个车间中的工人node同时工作，他们通过话题（topic）进行信息沟通。各个大的车间之间也存在这互相依赖的关系，共同组成一个有机的整体。

Node是指ROS中运行的最小处理单元。在ROS中，建议为一个目的创建一个节点，设计时注重可重用性。节点运行的同时，向主节点注册节点名称，并且还注册发布者(publisher)/订阅者(subscriber)，服务服务器(service server)/服务客户端(service client), 且注册消息形式，URI地址和端口。每个节点可以使用话题(topic)和服务(service)与其他节点交换消息。发布者(publisher)/订阅者(subscriber)之间传递topic，服务服务器(service server)/服务客户端(service client)之间传递service。

ROS node 之间的通讯形式主要包括两种：topic 和 service。

通过 topic 通讯时，不同的 node 可以向同一个 topic 上发送、接收数据，发送数据的 node 不知道数据是从哪个 node 发送过来的，同样地，发送数据的 node 也不知道是哪个 node 接收了数据。因此，每个 node 都是相对独立的，只需要负责自己的功能实现以及外部接口，不需要关心其他 node 的行为。这是一种开放式的收、发数据的方式，也是 node 之间通讯的主要形式，有利于构造分布式大系统。

service 则是一种请求+反馈的通信机制。消息的传输只涉及两个 node：发送请求的一方称为 client，提供服务的一方叫做 server。在通过 service 形式进行通讯时，client 首先向 server 请求服务, 收到消息之后 server 运行事先设置好的服务功能，并返回消息给 client。service 通讯一般用在事件触发情景中，例如满足某个条件就令 node 开启某项功能，并希望确认功能确实顺利开启。

通常情况下，一个cpp文件中定义一个节点(node)；但是在一个cpp文件中，可以在所定义的节点中，发布或订阅多条主题(topic)或一条主题(topic)。

https://blog.csdn.net/puqian13/article/details/91360102

<https://blog.csdn.net/qq_41972382/article/details/95209741>

# 编写发布器代码

初始化 ROS 系统

在 ROS 网络内广播我们将要在 chatter 话题上发布 [std\_msgs/String](https://link.zhihu.com/?target=http://docs.ros.org/api/std_msgs/html/msg/String.html" \t "https://zhuanlan.zhihu.com/p/_blank) 类型的消息

以每秒 10 次的频率在 chatter 上发布消息

在 beginner\_tutorials package 里创建 src/talker.cpp 文件：

#include <sstream>

#include "ros/ros.h"

#include "std\_msgs/String.h"

int main(int argc, char \*\*argv){

ros::init(argc, argv, "talker");

ros::NodeHandle n;

ros::Publisher chatter\_pub = n.advertise<std\_msgs::String>("chatter", 1000);

ros::Rate loop\_rate(10);

/\* a unique string for each message. \*/

int count = 0;

while (ros::ok()) //ros::ok()

{

std\_msgs::String msg;

std::stringstream ss;

ss << "hello world, I am a genius! " << count;

msg.data = ss.str();

ROS\_INFO("%s", msg.data.c\_str());

chatter\_pub.publish(msg);

ros::spinOnce();

loop\_rate.sleep();

++count;

}

return 0;}

talker.cpp中各行代码意义如下：

#include "ros/ros.h"

//一个实用的头文件，它引用了 ROS 系统中大部分常用的头文件

#include "std\_msgs/String.h"

//引用了 std\_msgs/String 消息, 它存放在 std\_msgs package 里，是由 String.msg 文件自动生成的头文件。

#include <sstream>

/\*\* \* This tutorial demonstrates simple sending of messages over the ROS system. \*/

int main(int argc, char \*\*argv){

/\*\* \* The ros::init() function needs to see argc and argv so that it can perform any ROS arguments and name remapping that were provided at the command line. For programmatic remappings you can use a different version of init() which takes remappings directly, but for most command-line programs, passing argc and argv is the easiest way to do it. The third argument to init() is the name of the node. You must call one of the versions of ros::init() before using any other part of the ROS system. \*/

ros::init(argc, argv, "talker");

//初始化 ROS.可指定节点的名称。节点的名称必须唯一(名称内不能包含 / 等符号)

/\*\* \* NodeHandle is the main access point to communications with the ROS system. \* The first NodeHandle constructed will fully initialize this node, and the last \* NodeHandle destructed will close down the node. \*/

ros::NodeHandle n;

//为这个进程的节点创建一个句柄。

//第一个创建的 NodeHandle 会为节点进行初始化，最后一个销毁的 NodeHandle 则会释放该节点所占用的所有资源 /\*\* \* The advertise() function is how you tell ROS that you want to \* publish on a given topic name. This invokes a call to the ROS master node, which keeps a registry of who is publishing and who \* is subscribing. After this advertise() call is made, the master \* node will notify anyone who is trying to subscribe to this topic name, \* and they will in turn negotiate a peer-to-peer connection with this \* node. advertise() returns a Publisher object which allows you to \* publish messages on that topic through a call to publish(). Once \* all copies of the returned Publisher object are destroyed, the topic \* will be automatically unadvertised. \* \* The second parameter to advertise() is the size of the message queue \* used for publishing messages. If messages are published more quickly \* than we can send them, the number here specifies how many messages to \* buffer up before throwing some away. \*/

ros::Publisher chatter\_pub = n.advertise<std\_msgs::String>("chatter", 1000);

//告诉master 我们将要在 chatter（话题名） 上发布 std\_msgs/String 消息类型的消息。　　　　　　　　　　　　　//如果我们发布的消息的频率太高，缓冲区中的消息在大于 1000 个的时候就会开始丢弃先前发布的消息。

//advertise返回一个 ros::Publisher 对象,它有两个作用： 1) 它有一个 publish() 成员函数可以让你在topic上发布消息； 2) 如果消息类型不对,它会拒绝发布。

ros::Rate loop\_rate(10);

///\*\* \* A count of how many messages we have sent. This is used to create

\* a unique string for each message. \*/

int count = 0;

while (ros::ok())　　　　　　//ros::ok()

{ /\*\* \* This is a message object. You stuff it with data, and then publish it. \*/

std\_msgs::String msg;

std::stringstream ss;

ss << "hello world " << count;

msg.data = ss.str();

ROS\_INFO("%s", msg.data.c\_str());

//ROS\_INFO 和其他类似的函数可以用来代替 printf/cout 等函数。

/\*\* \* The publish() function is how you send messages. The parameter \* is the message object. The type of this object must agree with the type \* given as a template parameter to the advertise<>() call, as was done \* in the constructor above. \*/

chatter\_pub.publish(msg);

//向所有订阅 chatter 话题的节点发送消息。

ros::spinOnce();

//ros::spinOnce()这一语句，否则你的回调函数就永远也不会被调用

loop\_rate.sleep();

//调用 ros::Rate 对象来休眠一段时间以使得发布频率为 10Hz。

++count;

}

return 0;}

# 编辑订阅器代码

订阅器代码流程如下所述：

初始化ROS系统

订阅 chatter 话题

进入自循环，等待消息的到达

当消息到达，调用 chatterCallback() 函数

在 beginner\_tutorials package 目录下创建 src/listener.cpp 文件：

#include "ros/ros.h"

#include "std\_msgs/String.h"

void chatterCallback(const std\_msgs::String::ConstPtr& msg)

//是一个回调函数，当接收到 chatter 话题的时候就会被调用。

{ ROS\_INFO("I heard: [%s]", msg->data.c\_str());

}

int main(int argc, char \*\*argv)

{ ros::init(argc, argv, "listener");

ros::NodeHandle n;

ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

ros::spin();

return 0;

}

Listener.cpp中各行代码意义如下：

#include "ros/ros.h"

#include "std\_msgs/String.h"

/\* This tutorial demonstrates simple receipt of messages over the ROS system. \*/

void chatterCallback(const std\_msgs::String::ConstPtr& msg)

//是一个回调函数，当接收到 chatter 话题的时候就会被调用。

{ ROS\_INFO("I heard: [%s]", msg->data.c\_str());}

int main(int argc, char \*\*argv)

{ /\*\*

\* The ros::init() function needs to see argc and argv so that it can perform

\* any ROS arguments and name remapping that were provided at the command line. For programmatic

\* remappings you can use a different version of init() which takes remappings

\* directly, but for most command-line programs, passing argc and argv is the easiest

\* way to do it. The third argument to init() is the name of the node.

\* You must call one of the versions of ros::init() before using any other

\* part of the ROS system.

\*/

ros::init(argc, argv, "listener");

/\* NodeHandle is the main access point to communications with the ROS system.

\* The first NodeHandle constructed will fully initialize this node, and the last

\* NodeHandle destructed will close down the node. \*/

ros::NodeHandle n;

/\* The subscribe() call is how you tell ROS that you want to receive messages

\* on a given topic. This invokes a call to the ROS

\* master node, which keeps a registry of who is publishing and who

\* is subscribing. Messages are passed to a callback function, here

\* called chatterCallback. subscribe() returns a Subscriber object that you

\* must hold on to until you want to unsubscribe. When all copies of the Subscriber

\* object go out of scope, this callback will automatically be unsubscribed from

\* this topic. \*

\* The second parameter to the subscribe() function is the size of the message

\* queue. If messages are arriving faster than they are being processed, this

\* is the number of messages that will be buffered up before beginning to throw

\* away the oldest ones.

\*/

ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

//告诉 master 我们要订阅 chatter 话题上的消息。当有消息发布到这个话题时，ROS 就会调用 chatterCallback() 函数。第二个参数是队列大小，以防我们处理消息的速度不够快，当缓存达到 1000 条消息后，再有新的消息到来就将开始丢弃先前接收的消息。

/\*\* \* ros::spin() will enter a loop, pumping callbacks. With this version, all \* callbacks will be called from within this thread (the main one). ros::spin() \* will exit when Ctrl-C is pressed, or the node is shutdown by the master. \*/

ros::spin();

//ros::spin() 进入自循环，可以尽可能快的调用消息回调函数。

return 0; }

# [CMakeLists\_txt](https://link.zhihu.com/?target=http://wiki.ros.org/catkin/CMakeLists.txt" \t "https://zhuanlan.zhihu.com/p/_blank)文件

## 确认原始[CMakeLists.txt](https://link.zhihu.com/?target=http://wiki.ros.org/catkin/CMakeLists.txt" \t "https://zhuanlan.zhihu.com/p/_blank)文件

原始文件中保留的有用代码如下：

cmake\_minimum\_required(VERSION 2.8.3)

project(beginner\_tutorials)

## Find catkin and any catkin packages

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

genmsg)

catkin\_package()

## 在 [CMakeLists.txt](https://link.zhihu.com/?target=http://wiki.ros.org/catkin/CMakeLists.txt" \t "https://zhuanlan.zhihu.com/p/_blank) 文件末尾加入几条语句:

include\_directories(include ${catkin\_INCLUDE\_DIRS})

add\_executable(talker src/talker.cpp)

target\_link\_libraries(talker ${catkin\_LIBRARIES})

add\_dependencies(talker ${PROJECT\_NAME}\_generate\_messages\_cpp)

add\_executable(listener src/listener.cpp)

target\_link\_libraries(listener ${catkin\_LIBRARIES})

add\_dependencies(listener ${PROJECT\_NAME}\_generate\_messages\_cpp)

## 说明

1-2会生成两个可执行文件, talker 和 listener, 默认存储到 [devel space](https://link.zhihu.com/?target=http://wiki.ros.org/catkin/workspaces%23Development_.28Devel.29_Space" \t "https://zhuanlan.zhihu.com/p/_blank) 目录下,具体在~/catkin\_ws/devel/lib/<package name> 中.

3.如果在 \*Groovy\* 版本下，你可以使用下边的这个变量来添加对所有必须的文件依赖:

add\_dependencies(talker ${catkin\_EXPORTED\_TARGETS})

# 编译及运行流程

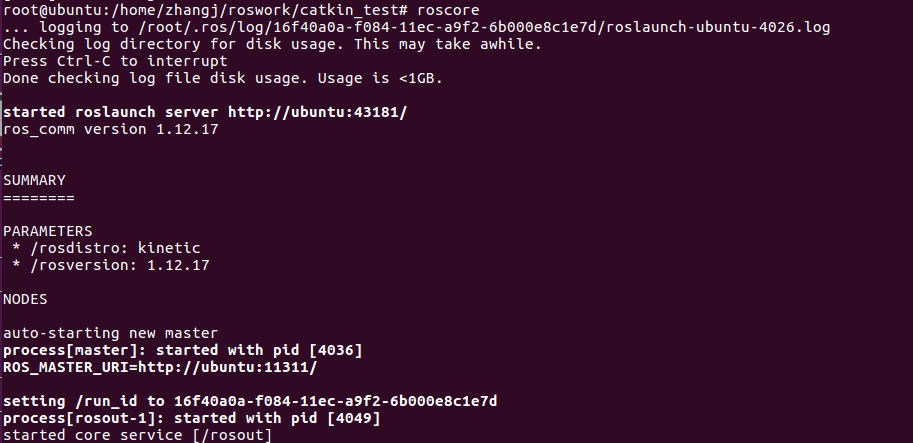
## 进入catkin\_test项目文件夹的根目录；

## source ./devel/setup.bash

## 输入catkin\_make命令，编译源码；

## 设置环境变量： source ./devel/setup.bash

## 启动ROS Master：roscore

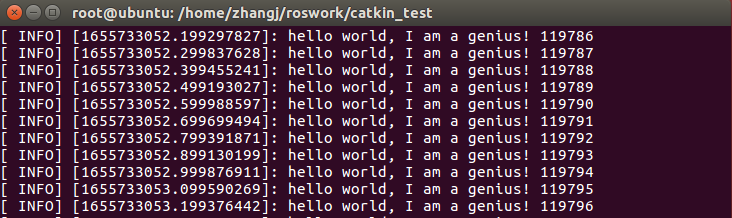


## 启动Publisher: rosrun beginner\_tutorials talker

In the “catkin\_ws” directory:

source ./devel/setup.bash

rosrun beginner\_tutorials talker

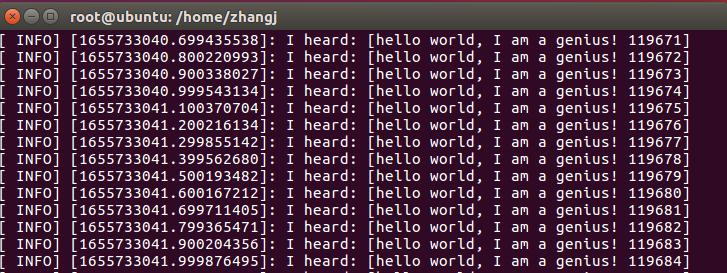


## 启动Subscriber：rosrun beginner\_tutorials listener

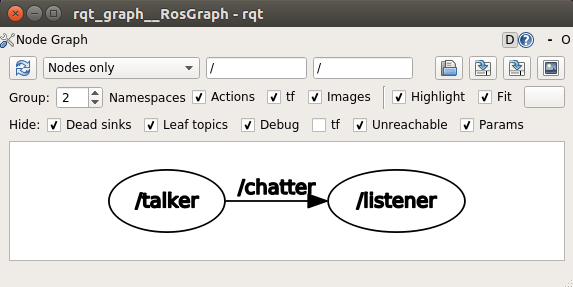
Create a new termial

source ./devel/setup.bash

rosrun beginner\_tutorials listener



## 查看例程的节点关系: rqt\_graph



# 出现问题及解决

[rospack] Error: package 'learning\_communication' not found

在最后运行时，出现error: package ‘\*\*\*’ not found错误

解决方法：source devel/setup.bash

# 单功能包内多节点及多主题生成及传输

## 生成主题信息的节点代码talker\_cpp

#include <sstream>

#include "ros/ros.h"

#include "std\_msgs/String.h"

int main(int argc, char \*\*argv){

ros::init(argc, argv, "talker");

ros::NodeHandle n;

ros::Publisher chatter\_pub = n.advertise<std\_msgs::String>("chatter", 1000);

ros::Publisher testmsg\_pub = n.advertise<std\_msgs::String>("testmsg", 1000);

ros::Rate loop\_rate(10);

/\* a unique string for each message. \*/

int count = 0;

while (ros::ok()) //ros::ok()

{ std\_msgs::String msg,msg1;

std::stringstream ss,ss1;

ss << "\nhello world, I am a genius! " << count;

msg.data = ss.str();

ROS\_INFO("%s", msg.data.c\_str());

chatter\_pub.publish(msg);

ss1<< "\nTest Message, Please Reply! " << count;

msg1.data = ss1.str();

ROS\_INFO("%s", msg1.data.c\_str());

testmsg\_pub.publish(msg1);

ros::spinOnce();

loop\_rate.sleep();

++count;

}

return 0;}

## 接收主题信息的节点1代码listener\_cpp

#include "ros/ros.h"

#include "std\_msgs/String.h"

void chatterCallback(const std\_msgs::String::ConstPtr& msg)

//是一个回调函数，当接收到 chatter 话题的时候就会被调用。

{ ROS\_INFO("I heard: [%s]", msg->data.c\_str());

}

int main(int argc, char \*\*argv)

{ ros::init(argc, argv, "listener");

ros::NodeHandle n;

ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

ros::spin();

return 0;

}

## 接收主题信息的节点2代码listener2\_cpp

#include "ros/ros.h"

#include "std\_msgs/String.h"

void chatterCallback(const std\_msgs::String::ConstPtr& msg)

//是一个回调函数，当接收到 chatter 话题的时候就会被调用。

{ ROS\_INFO("I heard: [%s]", msg->data.c\_str());

}

int main(int argc, char \*\*argv)

{ ros::init(argc, argv, "listener2");

ros::NodeHandle n;

ros::Subscriber sub = n.subscribe("testmsg", 1000, chatterCallback);

ros::spin();

return 0;

}

## CMakeLists\_txt

cmake\_minimum\_required(VERSION 3.0.2)

project(beginner\_tutorials)

## Compile as C++11, supported in ROS Kinetic and newer

# add\_compile\_options(-std=c++11)

## Find catkin macros and libraries

## if COMPONENTS list like find\_package(catkin REQUIRED COMPONENTS xyz)

## is used, also find other catkin packages

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

genmsg

)

catkin\_package()

## System dependencies are found with CMake's conventions

# find\_package(Boost REQUIRED COMPONENTS system)

include\_directories(include **${catkin\_INCLUDE\_DIRS}**)

add\_executable(talker src/talker.cpp)

target\_link\_libraries(talker **${catkin\_LIBRARIES}**)

add\_dependencies(talker **${PROJECT\_NAME}**\_generate\_messages\_cpp)

add\_executable(listener src/listener.cpp)

target\_link\_libraries(listener **${catkin\_LIBRARIES}**)

add\_dependencies(listener **${PROJECT\_NAME}**\_generate\_messages\_cpp)

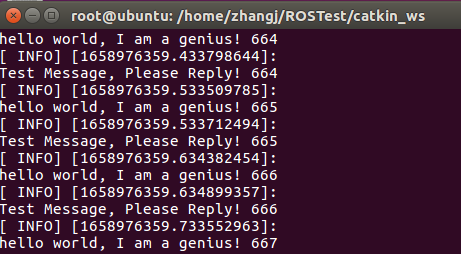
add\_executable(listener2 src/listener2.cpp)

target\_link\_libraries(listener2 **${catkin\_LIBRARIES}**)

add\_dependencies(listener2 **${PROJECT\_NAME}**\_generate\_messages\_cpp)

## 运行结果图

talker节点发布的主题信息



包括两条主题信息“hello world，I am a genius！”和“Test Message， Please Reply！”

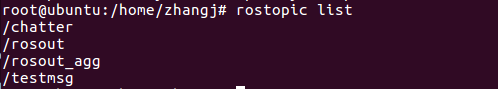
节点listener.cpp所接收到的信息



节点listener2.cpp所接收到的信息



查看所有的主题topic



rqt\_graph



查看chatter这个topic的数据类型：

Screenshot from 2022-07-27 20-23-39

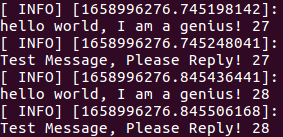
## 最简单的启动一个节点的Launch文件

<launch>

<node name="talker" pkg="beginner\_tutorials"

type="talker" output="screen"/>

</launch>



运行roslaunch和rosrun的位置相同，都要在ros项目文件的根目录下，且先运行如下命令：

Source ./devel/setup.bash

## 启动多个节点的Launch文件

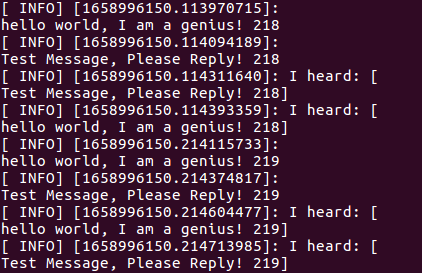
<launch>

<node name="talker" pkg="beginner\_tutorials" type="talker" output="screen"/>

<node name="listener" pkg="beginner\_tutorials" type="listener" output="screen"/>

<node name="listener2" pkg="beginner\_tutorials" type="listener2" output="screen"/>

</launch>



# ROS入门之launch

## 何为launch

launch中文含义是启动，launch文件顾名思义就是启动文件，要说这launch文件啊，那还得从roslaunch说起。相传，在程序猿们还没有使用roslaunch之前，需要手动rosrun逐个启动node，遇到大工程的时候，其繁杂程度不可想象，一度民不聊生。直到神实在看不下去了，于是送给程序猿们一份礼物，唤名roslaunch。roslaunch法力无边，只需程序猿们编写一个launch文件，启动大量节点弹指之间。从此一传十，十传百，程序猿们过上了幸福快乐的生活……

这故事虽假，但roslaunch大法却不虚。roslaunch是一个用于自动启动ros节点的命令行工具，从命令的字面上看还跟rosrun有些相似：

roslaunch PACKAGE\_NAME LAUNCH\_FILE\_NAME

只不过roslaunch的操作对象并非节点，而是launch文件。launch文件是描述一组节点及其话题重映射和参数的XML文件。根据规范，这些文件的后缀都是.launch

## 最简单的launch文件样例

<launch>

<node name="talker" pkg="beginner\_tutorials"

type="talker" output="screen"/>

</launch>

## launch标签小结

1.0 <launch>

launch标签就像一个有容乃大的括号，规定了一片区域，所有的launch文件都由<launch>开头，由</launch>结尾，所有的描述标签都要写在<launch></launch>之间

<launch>

……

</launch>

1.1 <node>

node标签可以说是launch文件里最常见的标签了，每个node标签里包括了ROS图中节点的名称属性name、该节点所在的包名pkg以及节点的类型type（其实就是可执行文件的名称，如果是用Python编写的就填写xxx.py，如果是cpp就写编译生成的可执行文件名），在上面的例子中还出现了output这个东西，这个属性在调试阶段很有用，output=“screen”表示了将终端输出转储在当前的控制台上，而不是在日志文件中。

属性 属性作用

name="NODE\_NAME" 为节点指派名称，这将会覆盖掉ros::init()定义的node\_name

pkg="PACKAGE\_NAME" 节点所在的包名

type="FILE\_NAME" 执行文件的名称如果是用Python编写的就填写xxx.py，如果是cpp就写编译生成的可执行文件名

output="screen" 终端输出转储在当前的控制台上，而不是在日志文件中

respawn="true" 当roslaunch启动完所有该启动的节点之后，会监测每一个节点，保证它们正常的运行状态。对于任意节点，当它终止时，roslaunch 会将该节点重启

required="true" 当被此属性标记的节点终止时，roslaunch会将其他的节点一并终止。注意此属性不可以与respawn="true"一起描述同一个节点

launch-prefix = "command-prefix" 我的理解是，相当于在执行启动命令时加上一段命令前缀

ns = "NAME\_SPACE" 这个属性可以让你在自定义的命名空间里运行节点

1.2<include>

该标签可以导入另一个roslaunch XML文件到当前文件。

属性 属性作用

file ="$(find pkg-name)/path/filename.xml" 指明我们想要包含进来的文件

ns="NAME\_SPACE" 相对NAME\_SPACE命名空间导入文件

使用起来就像下面这样

<include file="$(find demo)/launch/demo.launch" ns="demo\_namespace"/>

1.3<remap>

remap标签顾名思义重映射，ROS支持topic的重映射，remap标签里包含一个original-name和一个new-name，及原名称和新名称。

比如现在你拿到一个节点，这个节点订阅了"/chatter"topic，然而你自己写的节点只能发布到"/demo/chatter"topic，由于这两个topic的消息类型是一致的，你想让这两个节点进行通讯，那么可以在launch文件中这样写：

<remap from="chatter" to="demo/chatter"/>

这样就可以直接把/chattertopic重映射到/demo/chatter，这样子不用修改任何代码，就可以让两个节点进行通讯。

如果这个 remap标签写在与 node 元素的同一级，而且在launch 元素内的最顶层。 那么这个重映射将会作用于launch文件中所有的节点。

1.4<param>

param标签的作用相当于命令行中的rosparam set

比如现在在参数服务器中添加一个名为demo\_param，值为666的参数

<param name="demo\_param" type="int" value="666"/>

1.5<rosparam>

rosparam标签允许从YAML文件中一次性导入大量参数。

使用起来就像这样：

<rosparam command="load" file="$(find pkg-name)/path/name.yaml"/>

1.6<arg>

arg标签用来在launch文件中定义参数，arg和param在ROS里有根本性的区别，就像局部变量和全局变量的区别一样。arg不储存在参数服务器中，不能提供给节点使用，只能在launch文件中使用。param则是储存在参数服务器中，可以被节点使用。

<arg name="demo"/>

像上面这样，就简单地声明了一个参数，名叫demo，但是声明不等于定义，我们需要给他赋值，在赋值之后参数才能够发挥作用。

<arg name="demo" value="666"/>

<arg name="demo" default="666"/>

以上是两种简单的赋值方法，两者的区别是使用后者赋值的参数可以在命令行中像下面这样被修改，前者则不行。

roslaunch demo demo.launch demo:=6666

arg还有更加高级，也更加灵活的用法：

$(arg arg\_name)

当$(arg arg\_name)出现在launch文件任意位置时，将会自动替代为所给参数的值。

1.7<group>

group标签可以将若干个节点同时划分进某个工作空间。

<group ns="demo\_1">

<node name="demo\_1" pkg="demo\_1" type="demo\_pub\_1" output="screen"/>

<node name="demo\_1" pkg="demo\_1" type="demo\_sub\_1" output="screen"/>

</group>

<group ns="demo\_2">

<node name="demo\_2" pkg="demo\_2" type="demo\_pub\_2" output="screen"/>

<node name="demo\_2" pkg="demo\_2" type="demo\_sub\_2" output="screen"/>

</group>

group标签还可以做到对node的批量管理。比如我可以同时终止在同一个group中的节点。

<group if="1-or-0">

……

</group>

<group unless="1-or-0">

……

</group>

第一种情况，当if属性的值为0的时候将会忽略掉<group></group>之间的标签。

第二种恰好相反，当if属性的值为1的时候将会忽略掉<group></group>之间的标签。

但是我们通常不会直接用1或0来定义if标签。因为这样不够灵活。

通常会搭配$(arg arg\_name)来使用。

demo.launch文件

<launch>

<include file="include.launch">

<arg name="demo\_arg" value="1"/>

</include>

</launch>

include.launch文件

<launch>

<arg name="demo\_arg"/>

<group if="$(demo\_arg)">

<node name="demo" pkg="demo" type="demo\_pub" output="screen"/>

<node name="demo" pkg="demo" type="demo\_sub" output="screen"/>

</group>

</launch>

# 参考文献

1. <https://zhuanlan.zhihu.com/p/452758643>
2. [胡春旭. 2021-ROS机器人开发实践](https://www.baidu.com/link?url=RqEFkD376cM-ikOjeYZFgrLQSeJcmrX6t3RkwfAL04-k1uLqY67rXqX6tSqqbUCimjhT7e-K0suFbessM-QIoHRYaZGV9vZhAlkAC6haBOe&wd=&eqid=aac40035001a24af0000000462afd9ef" \t "https://www.baidu.com/_blank). section3.6:话题中的Publisherr与Subscriber.
3. https://blog.csdn.net/weixin\_41995979/article/details/81784987