# 实验二：ROS基础

## 实验目的

* ROS工作空间构建
* ROS通信层原理与重要概念解析。
* ROS消息通信基本过程。
* ROS编译构建系统
* ROS编程基础
* ROS工具

## 实验原理

## 1 工作空间

下面讲解本课的第二点内容：ROS的工作空间。ROS一般默认安装在/opt/ros/目录下，以后通过debian包方式安装的ROS功能包也位于此。但平时下载的源码和自己编写的功能包需存放到其它一个或多个目录，以便于修改和编译，我们把这样的工作目录称作工作空间。对于下载的功能来讲，建立工作空间可以根据需求自己修改代码实现多版本共存又不影响ROS安装目录下功能包。ROS中不同的编译构建系统有着不同的工作空间。下面我们分别介绍ROS的编译构建系统、工作空间创建以及工作空间overlay三个方面。因为ROS的发展，前后共有两种不同的编译构建系统——**rosbuild**和**catkin**。catkin是官方推荐的ROS编译构建系统，是原ROS编译构建系统rosbuild的继承者。catkin通过综合CMake宏和Python脚本在CMake正常工作流程之前实现一些功能操作。与rosbuild相比，其设计更具有规则性，以便能更好地发布功能包、更好地支持交叉编译、有更好的可移植性。Groovy版本之后的功能包基本都是使用catkin构建的。而rosbuild之所以官方还在支持，是因为以前有很多功能包都是通过rosbuild的创建的，功能包之间的依赖关系复杂，很多功能包由不同人员维护，功能包从rosbuild编译构建系统移植到catkin编译构建系统还需时日，这也是在这里介绍过时编译构建系统的原因。在这段过渡时期，通常把由rosbuild构建的功能包称为干包（Dry packages），把由catkin构建的功能包称为湿包（Wet packages）。

那catkin和rosbuild在宏观上又有什么不同呢？我们从以下三个方面进行分析

可移植性方面 用rosbuild构建的包不易移植到其它操作系统，特别是Windows。因为rosbuild混合使用bash脚本、GNU的Make和CMake构建代码。在rosbuild中编译构建功能包时，我们要调用rosbuild提供的如*rosmake*这样的自定义脚本。rosmake是一个bash脚本，调用*make*命令，*make*又调用CMake生成另一个makefile，最后执行*make*操作。而catkin则简洁得多，只简单调用CMake。而且catkin是通过自定义的CMake宏和部分Python代码实现。因为CMake和Python是可移植的，所以catkin也很容易移植到任何支持Python和CMake的系统。

编译构建方面 rosbuild编译构建功能包用rosmake，将编译目标功能包所有的依赖包，且生成的文件全部位于相应功能包的目录中，与代码源文件混在一起，没有安装的概念，不便于发布功能包；而CMake则使用cmake和make来编译构建功能包，默认编译构建工作空间的所有功能包，也可指定构建单独的功能包，建构目标存放于工作空间的devel目录下，可选择make install根据用户的预设执行功能包安装。除此之外，两者在编译构建参数的定义与依赖关系的处理也有所不同。

环境变量方面 rosbuild和catkin两者的运作都要依赖于环境变量。但是rosbuild中使用的环境变量对于catkin已无足轻重。最典型的就是环境变量ROS\_PACKAGE\_PATH，是包含所有功能包的路径列表。没有这个变量，rosmake和rosrun无法找到功能包资源。而在catkin中，最重要的环境变量是CMAKE\_PREFIX\_PATH，是包含CMake项目的路径列表，CMake能根据此变量找其他的CMake项目。Catkin包只是一个特殊CMake项目。

多功能包组织方面 在rosbuild中，有功能包集（stack）的概念，一个目录下有多个功能包；而catkin中使用元功能包进行替代，元功能包只是一个不包含任何功能包的目录加一个功能包清单文件。

### 1.1 创建工作空间

单个catkin功能包可以像普通的CMake项目一样建成一个独立的项目，但创建catkin工作空间可以一次编译构建多个有依赖关系的功能包。catkin工作空间的创建由catkin本身完成。catkin一般在安装ROS已经安装完成，使用时需要设置环境变量。

source /opt/ros/noetic/setup.bash

Catkin工作空间的创建命令如下：

mkdir -p ~/catkin\_ws/src

cd ~/catkin\_ws/src

catkin\_init\_workspace

此时catkin工作空间已经创建，已具备catkin\_make条件，catkin\_make命令必须在catkin工作空间的最顶层执行。

cd ~/catkin\_ws/

catkin\_make

此时catkin工作空间的文件结构如下：

catkin\_ws/ -- 工作空间

src/ -- 源码空间

CMakeLists.txt -- 到catkin/cmake/toplevel.cmake文件的链接

build/ -- 构建空间

devel/ -- 开发空间 (由CATKIN\_DEVEL\_PREFIX变量设定)

install/ -- 安装空间 (此时其实没有install目录，为了显示catkin工作

从上面的文件结构可以看出，catkin工作空间有四个组成部分，分别为源码空间、构建空间、开发空间和安装空间。

* 源码空间存放功能包的所有源码文件，编译构建时就从这个目录寻找功能包。其显著标志就是CMakeLists.txt文件，链接到.cmake文件。
* 构建空间存放编译构建过程中的中间文件。
* 开发空间存放构建好的可执行程序和运行库，可由CATKIN\_DEVEL\_PREFIX变量重新设定，但目录的选择要求必须满足子目录没有功能包。
* 安装空间就是功能包的安装目录，可由CATKIN\_INSALL\_PREFIX变量重新设定。因为新建工作空间，所以默认的install目录并未创建，执行catkin\_make install，此目录就会创建。

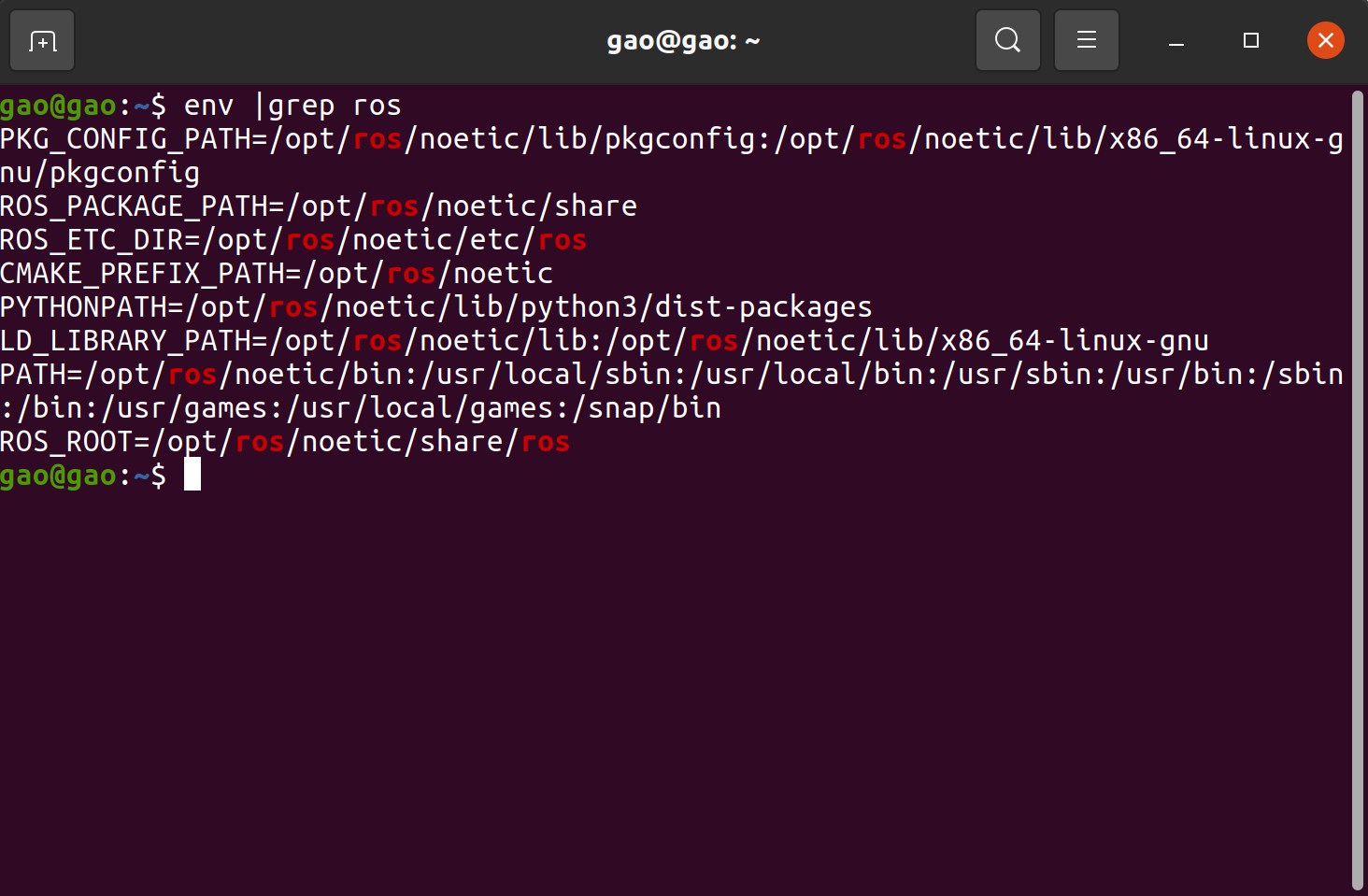
### 1.2 工作空间overlay

工作空间的overlay实际上就是系统环境链的overlay。先来说明下工作空间为什么要overlay。通常ROS核心及其它以二进制格式发行的功能包通过二进制安装的方式安装在/opt/ros/ROS\_DISTRO目录下，而用户的工作空间一般设置在用户目录下，用户常常因为各种自身需求改动定制已发行的功能包或创建自己的功能包，并让所有依赖于此功能包的调用都选择工作空间中的同名定制功能包而非系统安装目录中的。overlay的概念正好能满足这一需求，系统环境链overlay能让ROS找到不管是安装在系统目录中还是编译在用户工作空间的所有功能包，而对于两处同名的功能包则采取遮蔽下层的策略。这一策略的实现主要通过环境变量，overlay的顶层相关路径放置在各环境变量的最前面，也就优先搜索。除了overlay ROS系统安装目录，工作区之间也可以overlay，以实现多层overlay。

#### **1.2.1环境变量**

既然工作空间的overlay就是环境变量链的overlay，那先谈谈ROS系统变量。不管是ROS非独立发布的工具还是功能包，其运行都是要依赖系统环境变量的设置，前面也讲到安装ROS系统后要设置环境变量。ROS默认的环境变量都是通过ROS中形如setup.XXX（XXX为所使用SHELL的名称）这样的脚本来完成设置。使用bash，在kinetic版本中，命令如下：> source /opt/ros/noetic/setup.bash

为了避免新开一个shell就执行一次source命令，我们通常把上面的命令添加到~/.bashrc文件中。新开的bash shell会自动执行.bashrc中的脚本。另外提醒一下，熟悉Linux系统的朋友应该知道前缀为.的文件是隐藏的。 我们可以用env命令查看与ROS相关的系统环境变量： > env | grep ros



#### 1.2.2 catkin工作空间overlay（覆盖）

catkin工作空间执行catkin\_make会在devel目录下生成系统环境设置脚本（setup.XXX），执行catkin\_make install会在安装目录生成类似的系统环境设置脚本，这些脚本已包含ROS系统安装目录环境设置脚本的设置内容，也会保留其它脚本已经设置的环境变量。通过这些脚本就可以实现catkin工作空间的overlay。如overlay系统安装目录：

source /opt/ros/noetic/setup.bash env | grep ros

mkdir -p ~/catkin\_ws/src

cd ~/catkin\_ws/src

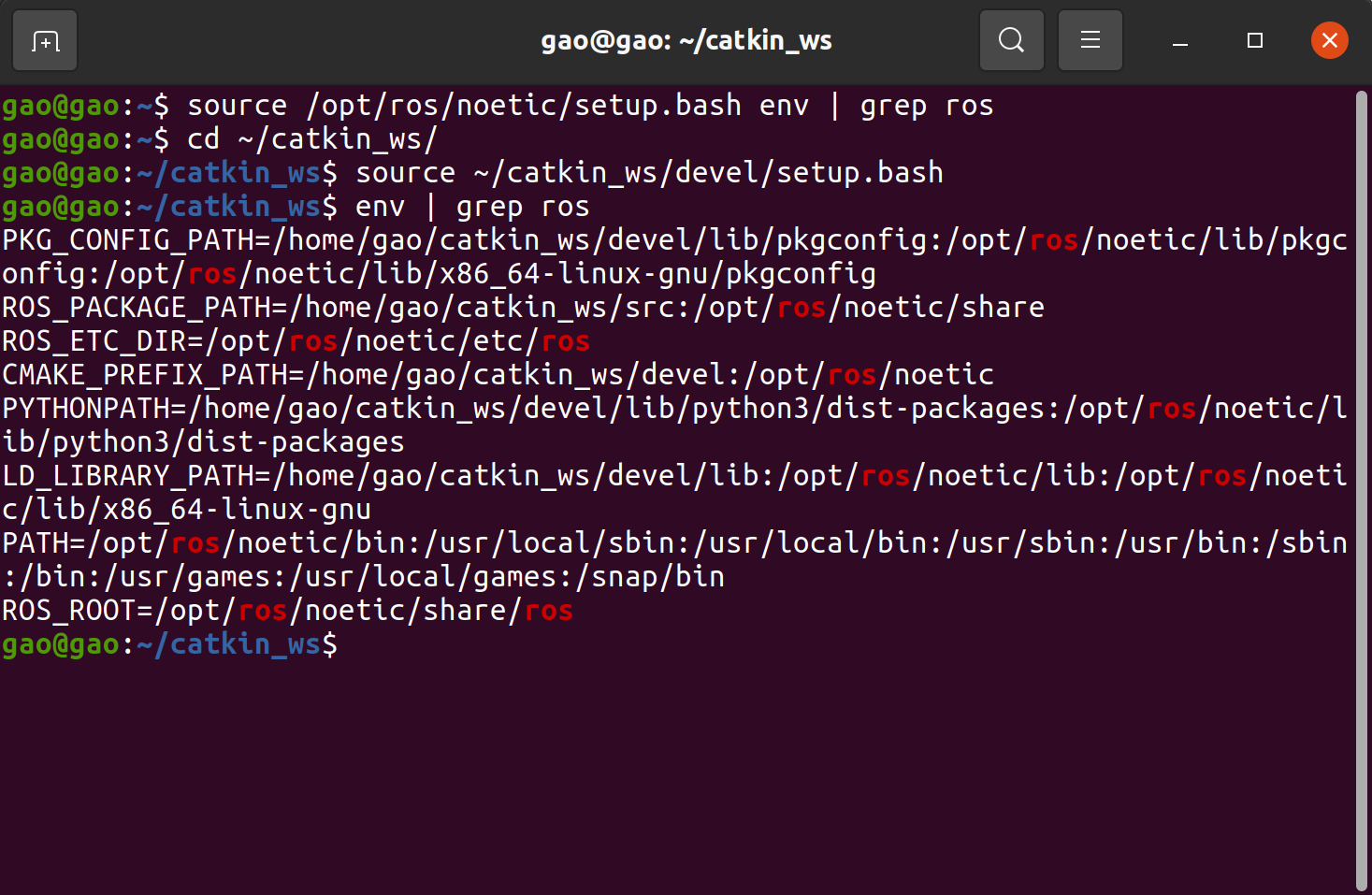
catkin\_init\_workspace

cd ~/catkin\_ws

catkin\_make

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

env | grep ros



因为catkin功能包编译默认是对整个工作空间所有功能包的编译，所以大多用户会有需求创建一个以上的catkin工作空间。

如此，前面我们在~/.bashrc中添加的source命令就显得没用了，如果catkin\_ws是主用工作空间，只要source了其devel目录下的setup脚本就可以设置好所有的环境参数（具体请查看\_setup\_util.py，环境变量设置脚本由其生成）。所以可以把source ~/catkin\_ws/devel/setup.bash替换掉原有的source命令。

## 2. ROS 通信层原理及重要概念

ROS通信层中间件可以说是ROS系统的核心，我们之前介绍的ROS系统诸多特性与功能都基于此，我们这节课就要仔细讨论ROS通信中间件的原理与topic/service 具体实现，为我们之后进一步理解ROS框架，使用ROS 通信api打好基础。

之前介绍过，ROS的通信层设计是基于网络运输层tcp/udp协议，而最为重要的一点是，ROS的computational graph（计算图）概念。ROS的网络拓扑结构为P2P，节点间互相独立。下面进行ROS通信层中的术语解析.

### 2.1 Node

node是一个处理计算的进程。node的实体是操作系统中的一个进程，因为要占用端口号进行通信，多机分布式系统中还要标明ip，详见后面的uri，node在graph中使用topic service和parameter相互通信，协调工作。在不同的系统设计中，node承载的功能粒度也不一样。比如大部分的设备驱动都是单独占用一个node，为了提高代码复用率经常会有细粒度的划分，而在很多视觉处理的系统避免使用多节点结构，传递图像数据会造成系统资源浪费（nodelet的应用）。node的使用极大的增加了系统模块化和代码封装的程度，也给系统带来了一些容错率的提高。

在ROS的世界里，最小的进程单元就是节点（node）。一个软件包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程(process)，这个进程在ROS中就叫做节点。 从程序角度来说，node就是一个可执行文件（通常为C++编译生成的可执行文件、Python脚本）被执行，加载到了内存之中；从功能角度来说，通常一个node负责者机器人的某一个单独的功能。由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个node上，而会采用分布式的方式，把鸡蛋放到不同的篮子里。例如有一个node来控制底盘轮子的运动，有一个node驱动摄像头获取图像，有一个node驱动激光雷达，有一个node根据传感器信息进行路径规划。这样做可以降低程序发生崩溃的可能性，试想一下如果把所有功能都写到一个程序中，模块间的通信、异常处理将会很麻烦。

### 2.2 Master

由于机器人的元器件很多，功能庞大，因此实际运行时往往会运行众多的node，负责感知世界、控制运动、决策和计算等功能。那么如何合理的进行调配、管理这些node？这就要利用ROS提供给我们的节点管理器master, master在整个网络通信架构里相当于管理中心，管理着各个node。node首先在master处进行注册，之后master会将该node纳入整个ROS程序中。node之间的通信也是先由master进行“牵线”，才能两两的进行点对点通信。当ROS程序启动时，第一步首先启动master，由节点管理器处理依次启动node。

master给ros系统中其他节点提供命名与注册的服务。可以说它负责管理ROS系统中所有的name。它跟踪节点中的publisher与subscriber，service的server与client，记录其他节点的位置（uri标明，host与port），并将这些信息通知给需要建立链接的节点。（从分布式系统的角度分析，ros这样P2P网络中集中式管理peer信息，ROS多机系统中，ros的全局时钟仅是本机的系统时钟，但不负责时间上的同步，仿真环境中会由仿真器节点发布/clock topic其他节点订阅来实现同步。重要：多机需要同步系统时间的工具来实现ros内时钟同步，一般是ntp，pr2应用linux中的chrony进行基于ntp的系统时间同步）

### 2.3 Topic

ROS中广为使用的是异步的 publish-subscribe 通信模式。这种方式将信息的产生和使用双方解耦，首发双方的信息都是由master node 进行管理。Node从需要的topic那取得消息，同一个topic 可以有多个 subscriber与publisher。Topic 一般用于单向，消息流通信。Node 需要同步通信交换信息时一般使用service。Topic 一般拥有很强的类型定义：一种类型的topic只能接受/发送特定数据类型（message type）的message。Publisher 没有被要求类型一致性，但是接受时subscriber会检查类型的md5，进而报错。

### 2.4 Service

service 用于处理ros通信中的同步通信，采用server/client 语义。每个service type拥有 request 与 response两部分，对于service中的 server，ros不会检查重名（name conflict），只有最后注册的server会生效，与client建立连接。

### 2.5 Parameter

parameter 可以看作为ros系统运行时中定义的全局变量，而master node 中有parameter server 来维护这些变量。而namespace的存在使得parameter 拥有了非常清晰的层次划分，避免了重名，而且使得parameter访问可以单独访问也可以树状访问（层层解析namespace）

### 2.6 URI

定位node在分布式系统中的位置，格式为： protocol://host:port，protocol一般为http或者 rosrpc， host为 hostname 或者 ip 地址，port则为端口号。

### 2.7 XML-RPC

XML-RPC是一个通用的远程调用协议，主要特性是使用XML语言将函数以及参数封装，XML优势在于支持其的语言类型很多，而本身传输的文本方便人进行调试，XML-RPC 调用时无状态的（stateless）:远程过程调用由两个阶段组成：请求（request）与应答（response）。一个调用者将方法调用请求发给 被调用者，然后被调用者返回调用是否成功与相应返回值。其间没有状态信息需要追踪，简化了控制逻辑。ROS中topic/service p2p传输时的node name信息交换与parameter server 都是由XML-RPC进行支持，可见XML-RPC在ROS网络中的重要地位。

数据类型如下：

在XML-RPC中，方法参数和其返回值被封装在value 的实体中，value有以下几种固定的类型可以选择。（xml中体现为value的子标签）

* string 为ascii 码的字符串，为value的默认格式。不合法的字符只有&和<，它们会被编码成&amp;和&lt;。
* int 或者 i4 是32位的有符号整型，十进制表示中，前缀-号则为负数。
* boolean 只有两种取值，0和1，用于表示布尔类型。
* double 实数类型，前缀-号表示负数。
* dataTime. ISO8601日期时间，用ISO-8601格式表示。
* base64 用 base64算法编码的二进制字符串。
* array values组成的表，在data实体下一层
* Struct又称为map表示关系的集合，每一个struct实体是由一个name与 value的键值对组成。

## 2 ROS topic/service 数据传输过程

对于ROS的用户来说，这一部分是最为神秘的，一般大家都会忽视这一方面的，我们在这里将其剖析开来，揭秘这些过程，让大家对ROS整个系统有一个概念上的认识，之后接下来我们会给出一些常见bug来加强这些过程的理解。 Topic 链接过程：

topic

以一个最简易topic 模型为例。 当一个talker 启动时，它会发起一次RPC，将自己的信息（发布的topic的name，node的URI）传给master node。 当一个listener 启动时，它会发起一次RPC，将自己的信息（node的URI，订阅的topic name）注册到master node中，如果有此topic 有新的publisher，则master node 发起一次RPC，将publisher所在的node信息传给listener，收到这些信息后，listener本身会对talker发起一次TCP链接，传输topic name与数据类型，而talker返回talker 接受数据的tcp server 的信息，具体为ip与端口号，然后listener 以这些信息再发起连接，传输相应数据。 总的来说会发生三次RPC，交换node间信息，两次TCP链接，第一次交换TCP server与topic 信息，第二次开始传输数据。 值得注意的是：RPC使用http协议，而传输数据使用TCPROS协议，这个协议使用二进制数据流，相对于XML-RPC来说更快更轻量级，关于TCPROS的细节之后会讲到。

Service

Service链接过程： Server与caller的注册过程与topic是类似的。只不过区别在于同一个service能被多个node注册，但是只有最后一个能被其他节点接受。而最后TCP链接也只会发生一次，这一次会caller传输request 数据，server 返回 response 数据。

TCPROS协议中我们提一下重点：

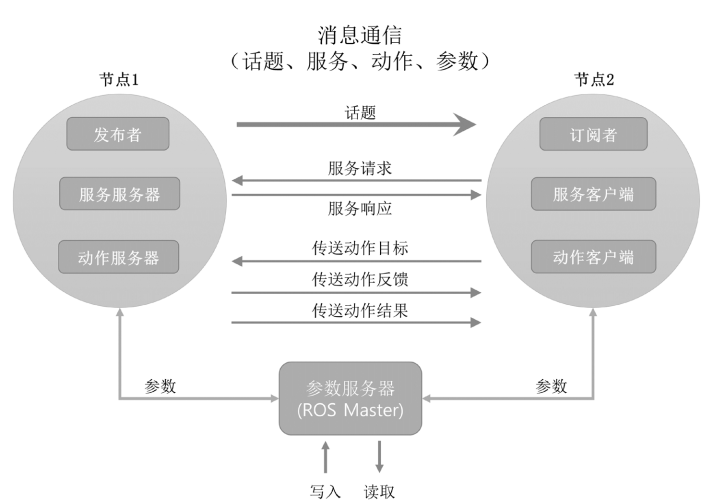
* Md5：TCPROS为了保证两边传输数据类型一致，会在协议头中给出topic name 的md5 hash算法处理过的值，而每次你生成新的msg 时，md5的值都会因为你内部数据类型的变动而改变，这样就避免了新msg与 旧msg 传输类型不一致的问题。
* Subscriber 选项tcp\_nodelay :如果是“1” 则给socket 设置 TCP\_NODELAY 选项，降低延迟，可能会降低传输效率。
* Service client 选项：persistent 设置为1，则service的链接会一直开放给多个 service request

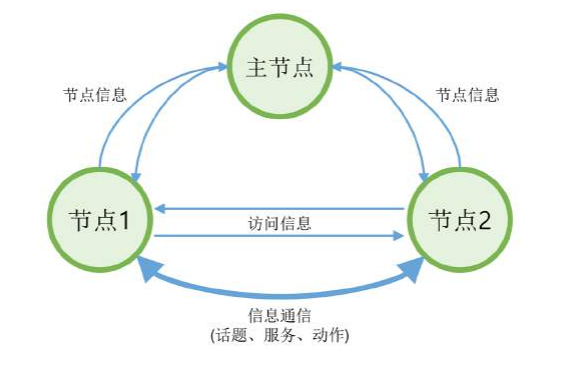
我们可以通过一些小实验来验证这些过程：

* 比如，使用第一章的rostopic 命令行工具发布一个消息
* $ rostopic pub -r 10 /chatter std\_msgs/String "data: 'hello'"
* 再使用rostopic echo 订阅这个publisher
* $ rostopic echo /chatter

这个时候关闭master node，发现传输不受影响，而再使用rostopic echo建立另一个listener时，会报错: > ERROR: Unable to communicate with master!

## 3 ROS消息通信

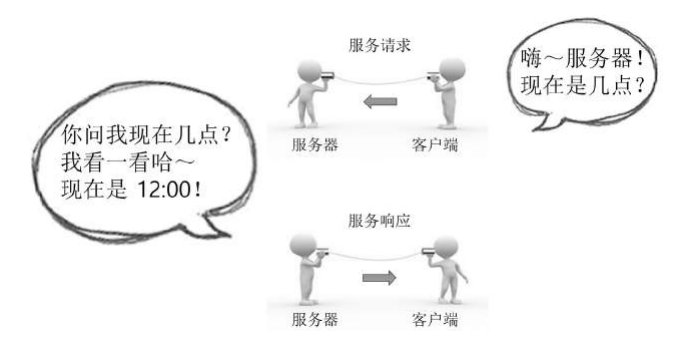




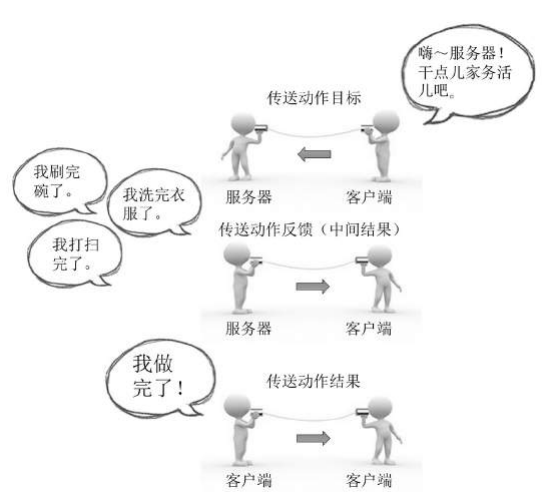
话题消息通信是指发送信息的发布者和接收信息的订阅者以话题消息的形式发送和接收信息。



服务消息通信是指请求服务的服务客户端与负责服务响应的服务服务器之间的同步双向服务消息通信



动作消息通信是在如下情况使用的消息通信方式：服务器收到请求后直到响应所需的时间较长，且需要中途反馈值。这与服务非常相似，服务具有与请求和响应分别对应的目标（goal）和结果（result）。除此之外动作中还多了反馈（feedback）。



## 4 ROS编译构建系统

功能包编写一般包括下面几个步骤：先使用建构系统工具创建功能包、功能包功能编写和配置文件编写、功能包编译、环境变量设置及功能包运行。下面及下课分别介绍这几个步骤。从这里开始只讨论用catkin编译构建系统。catkin功能包主要有以下几部分组成：

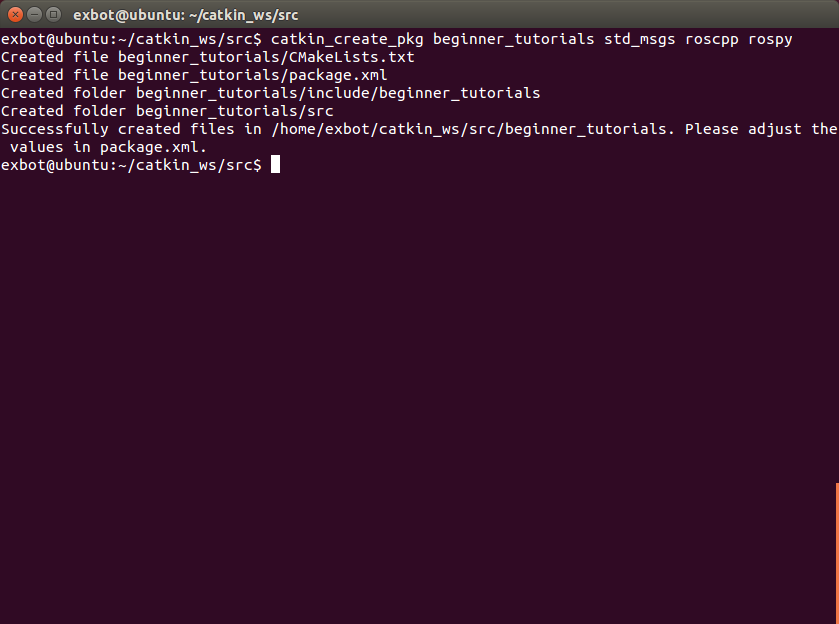
* 必须包含package.xml文件；
* 必须包含CMakeLists.txt文件；
* 在每一个文件夹下只能有一个功能包，且功能包之间不允许嵌套。

catkin功能包使用**catkin\_create\_pkg**脚本创建，其用法如下：> catkin\_create\_pkg [depend1] [depend2] [depend3] ...

其中，package\_name为功能包的名字，必填，后面为此功能包的依赖列表。下面创建一个名为**beginner\_tutorials**，依赖std\_msgs、rospy、roscpp的功能包，在创建之前进入源码空间：

cd ~/catkin\_ws/src

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



这将会在~/catkin\_ws/src下创建beginner\_tutorials文件夹及相应文件，其结构如下：

beginner\_tutorials/

* src\ 存放ROS的源代码，包括C++的源码和(.cpp)以及Python的module(.py)

include\ 存放C++源码对应的头文件

CMakeLists.txt 定义package的包名、依赖、源文件、目标文件等编译规则，是package不可少的成分

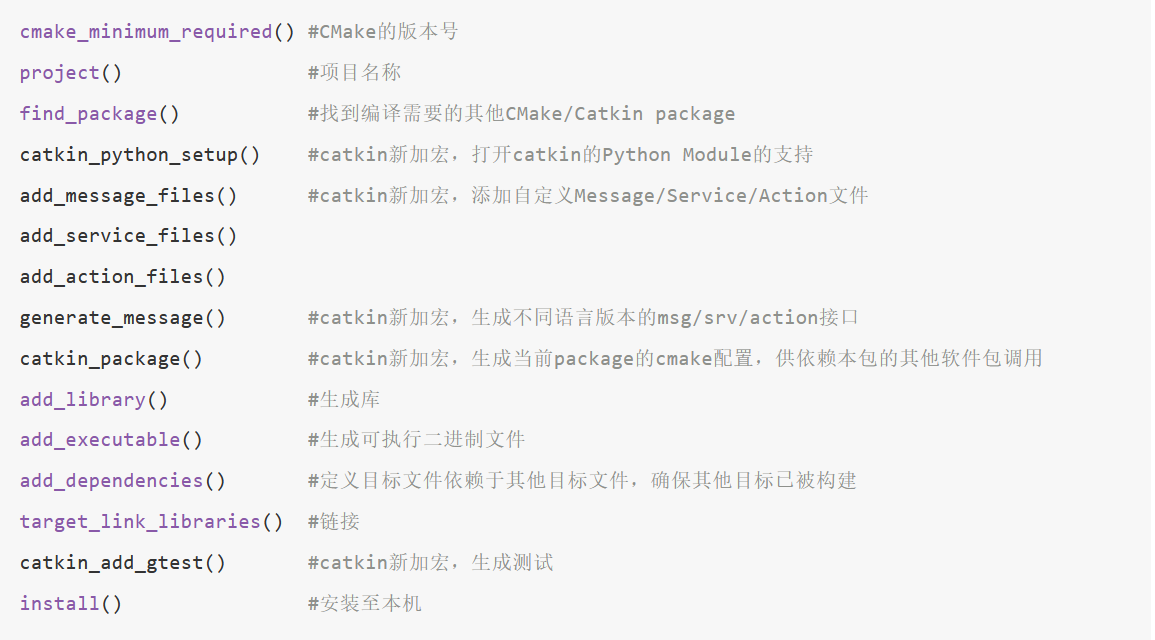
* package.xml 描述package的包名、版本号、作者、依赖等信息，是package不可少的成分

在此提醒一点，ROS功能包的命名遵循一个命名规范，只允许使用小写字母、数字和下划线，而且首字符必须是小写字母。包括 catkin在内的一些ROS工具，不支持不遵循此命名规范的功能包。

CMakeLists.txt原本是Cmake编译系统的规则文件，而Catkin编译系统基本沿用了CMake的编译风格，只是针对ROS工程添加了一些宏定义。所以在写法上，catkin的CMakeLists.txt与CMake的基本一致。

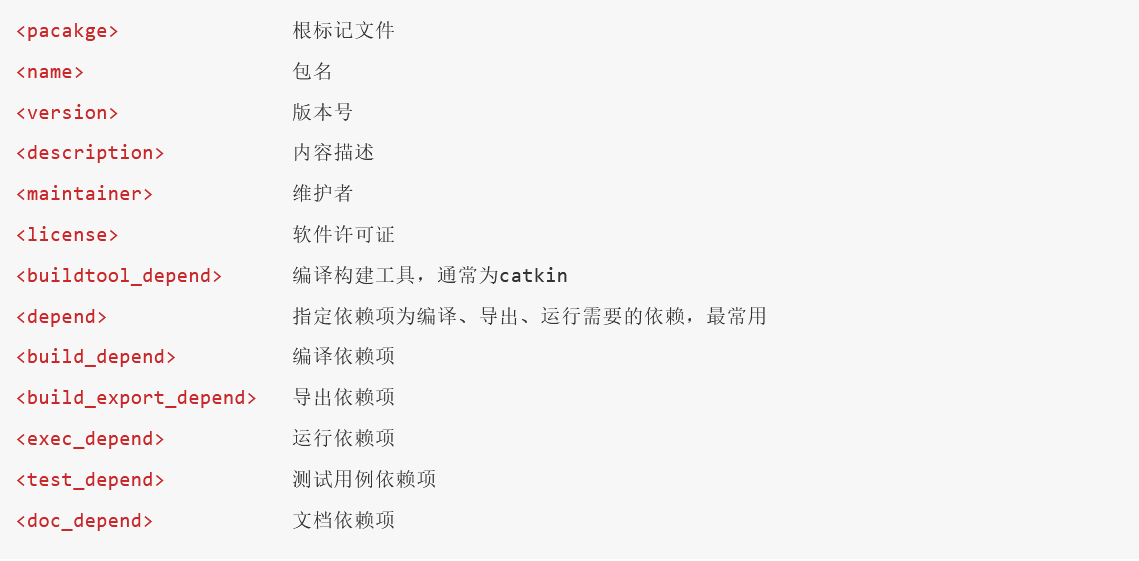
这个文件直接规定了这个package要依赖哪些package，要编译生成哪些目标，如何编译等等流程。所以CMakeLists.txt非常重要，它指定了由源码到目标文件的规则，catkin编译系统在工作时首先会找到每个package下的CMakeLists.txt，然后按照规则来编译构建。

CMakeLists.txt的基本语法都还是按照CMake，而Catkin在其中加入了少量的宏，总体的结构如下：



**package.xml也是一个catkin的package必备文件，它是这个软件包的描述文件**，在较早的ROS版本(rosbuild编译系统)中，这个文件叫做manifest.xml，用于描述pacakge的基本信息。如果你在网上看到一些ROS项目里包含着manifest.xml，那么它多半是hydro版本之前的项目了。

pacakge.xml包含了package的名称、版本号、内容描述、维护人员、软件许可、编译构建工具、编译依赖、运行依赖等信息。实际上rospack find、rosdep等命令之所以能快速定位和分析出package的依赖项信息，就是直接读取了每一个pacakge中的package.xml文件。它为用户提供了快速了解一个pacakge的渠道。



## 5 ROS编程基础

### 5.1 发布者节点和订阅者节点的创建和运行

**创建功能包**

$ cd ~/catkin\_ws/src

$ catkin\_create\_pkg ros\_tutorials\_topic message\_generation std\_msgs roscpp

**修改构建配置文件（CMakeLists.txt）**

$ gedit CMakeLists.txt

以下是为了匹配正在创建的功能包而修改的CMakeLists.txt。同样，有关每个选项的详细说明

cmake\_minimum\_required(VERSION 3.02

project(ros\_tutorials\_topic)

## catkin构建时需要的组件包。

## 是依赖包，是message\_generation、 std\_msgs和roscpp。

## 如果这些功能包不存在，在构建过程中会发生错误。

find\_package(catkin REQUIRED COMPONENTS message\_generation std\_msgs roscpp)

## 消息声明：MsgTutorial.msg

add\_message\_files(FILES MsgTutorial.msg)

## 这是设置依赖性消息的选项。

## 如果未安装std\_msgs，则在构建过程中会发生错误。

generate\_messages(DEPENDENCIES std\_msgs)

## catkin功能包选项，描述了库、catkin构建依赖项和系统依赖的功能包。

catkin\_package(

LIBRARIES ros\_tutorials\_topic

CATKIN\_DEPENDS std\_msgs roscpp

)

## 设置包含目录。

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

## topic\_publisher节点的构建选项。

## 配置可执行文件、目标链接库和其他依赖项。

add\_executable(topic\_publisher src/topic\_publisher.cpp)

add\_dependencies(topic\_publisher ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(topic\_publisher ${catkin\_LIBRARIES})

## topic\_subscriber节点的构建选项。

add\_executable(topic\_subscriber src/topic\_subscriber.cpp)

add\_dependencies(topic\_subscriber ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(topic\_subscriber ${catkin\_LIBRARIES})

**创建发布者节点**

#include "ros/ros.h" // ROS默认头文件

#include "ros\_tutorials\_topic/MsgTutorial.h" // MsgTutorial消息头文件（ 构建后自动生成）

int main(int argc, char \*\*argv) // 节点主函数

{

ros::init(argc, argv, "topic\_publisher"); // 初始化节点名称

ros::NodeHandle nh; // 声明一个节点句柄来与ROS系统进行通信

// 声明发布者，创建一个使用ros\_tutorials\_topic功能包的MsgTutorial 消息文件的

// 发布者ros\_tutorial\_pub。话题名称是"ros\_tutorial\_msg"，

// 消息文件发布者队列（queue）的大小设置为100

ros::Publisher ros\_tutorial\_pub =

nh.advertise<ros\_tutorials\_topic::MsgTutorial>("ros\_tutorial\_msg", 100);

// 设定循环周期。"10"是指10Hz，是以0.1秒间隔重复

ros::Rate loop\_rate(10);

ros\_tutorials\_topic::MsgTutorial msg; // 以MsgTutorial消息文件格式声明一个 叫做msg的消息

int count = 0; // 声明要在消息中使用的变量

while (ros::ok())

{

msg.stamp = ros::Time::now(); // 把当前时间传给msg的下级消息stamp

msg.data = count; // 将变量count的值传给下级消息data

ROS\_INFO("send msg = %d", msg.stamp.sec); // 显示stamp.sec消息

ROS\_INFO("send msg = %d", msg.stamp.nsec); // 显示stamp.nsec消息

ROS\_INFO("send msg = %d", msg.data); // 显示data消息

ros\_tutorial\_pub.publish(msg); // 发布消息。

loop\_rate.sleep(); // 按照上面定义的循环周期进行暂歇

++count; // 变量count增加1

}

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(topic\_publisher src/topic\_publisher.cpp)

**创建订阅者节点**

#include "ros/ros.h" // ROS的默认头文件

#include "ros\_tutorials\_topic/MsgTutorial.h" // MsgTutorial消息头文件（构建后自动生成）

// 这是一个消息后台函数，

// 此函数在收到一个下面设置的名为ros\_tutorial\_msg的话题时候被调用。

// 输入的消息是从ros\_tutorials\_topic功能包接收MsgTutorial消息。

void msgCallback(const ros\_tutorials\_topic::MsgTutorial::ConstPtr& msg)

{

ROS\_INFO("recieve msg = %d", msg->stamp.sec); // 显示stamp.sec消息

ROS\_INFO("recieve msg = %d", msg->stamp.nsec); // 显示stamp.nsec消息

ROS\_INFO("recieve msg = %d", msg->data); // 显示data消息

}

int main(int argc, char \*\*argv) // 节点主函数

{

ros::init(argc, argv, "topic\_subscriber"); // 初始化节点名称

ros::NodeHandle nh; // 声明用于ROS系统和通信的节点句柄

// 声明订阅者，创建一个订阅者ros\_tutorial\_sub，

// 它利用ros\_tutorials\_topic功能包的的MsgTutorial消息文件。

// 话题名称是"ros\_tutorial\_msg"，订阅者队列（queue)的大小设为100。

ros::Subscriber ros\_tutorial\_sub = nh.subscribe("ros\_tutorial\_msg", 100, msgCallback);

// 用于调用后台函数，等待接收消息。在接收到消息时执行后台函数。

ros::spin();

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(topic\_subscriber src/topic\_subscriber.cpp)

**创建消息文件**

time stamp

int32 data

在上述的CMakeLists.txt文中添加了如下选项

add\_message\_files(FILES MsgTutorial.msg)

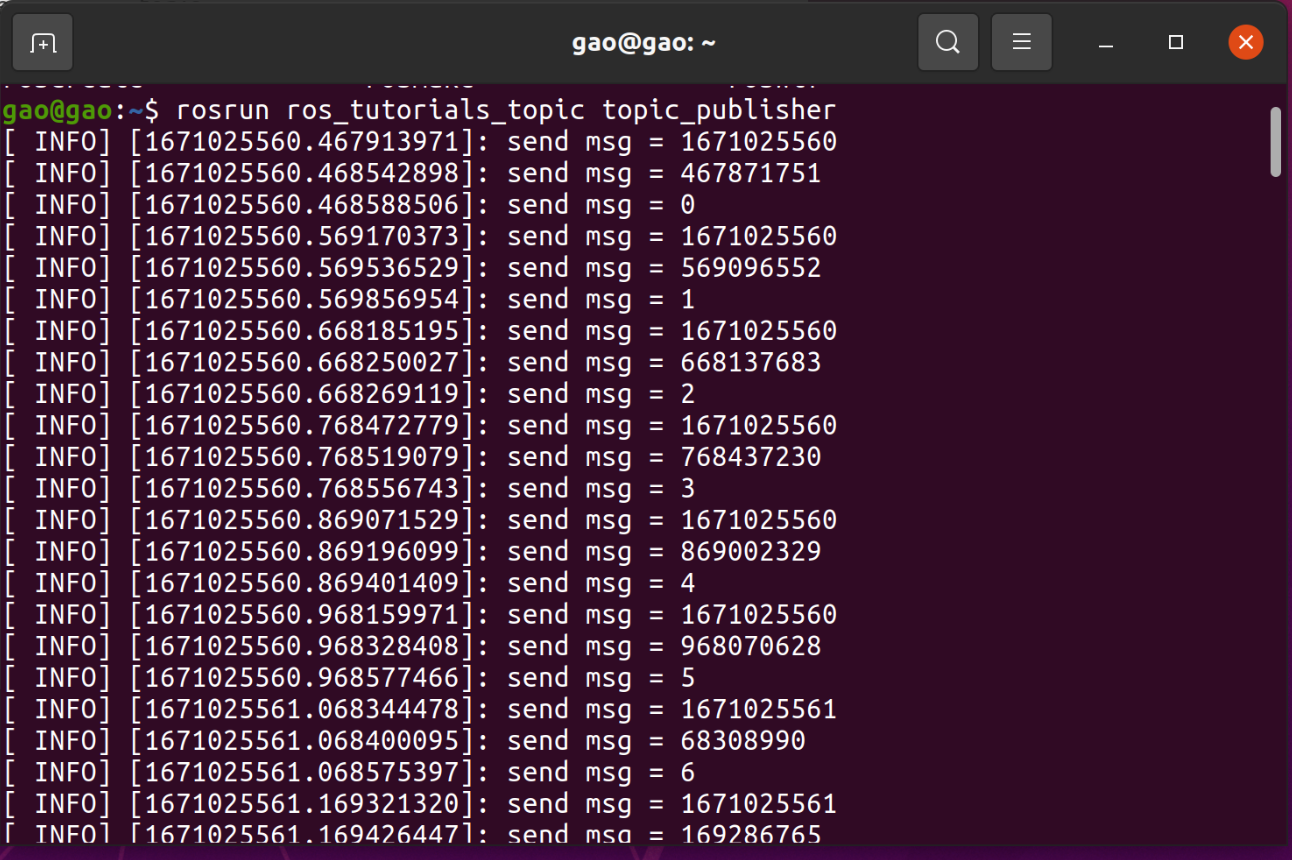
**构建并运行节点**

$ cd ~/catkin\_ws

$ catkin\_make

$ roscore

$ rosrun ros\_tutorials\_topic topic\_publisher



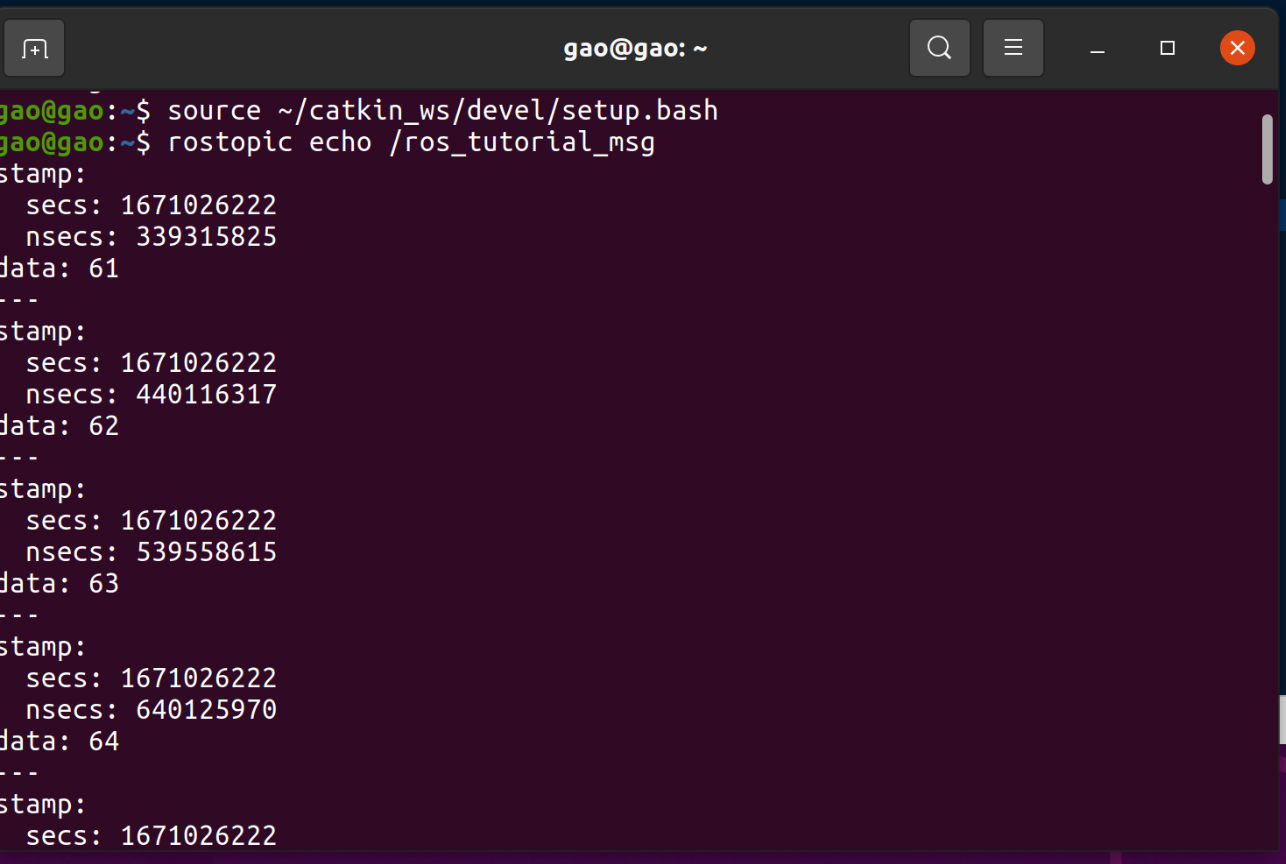
$ rostopic list

/ros\_tutorial\_msg

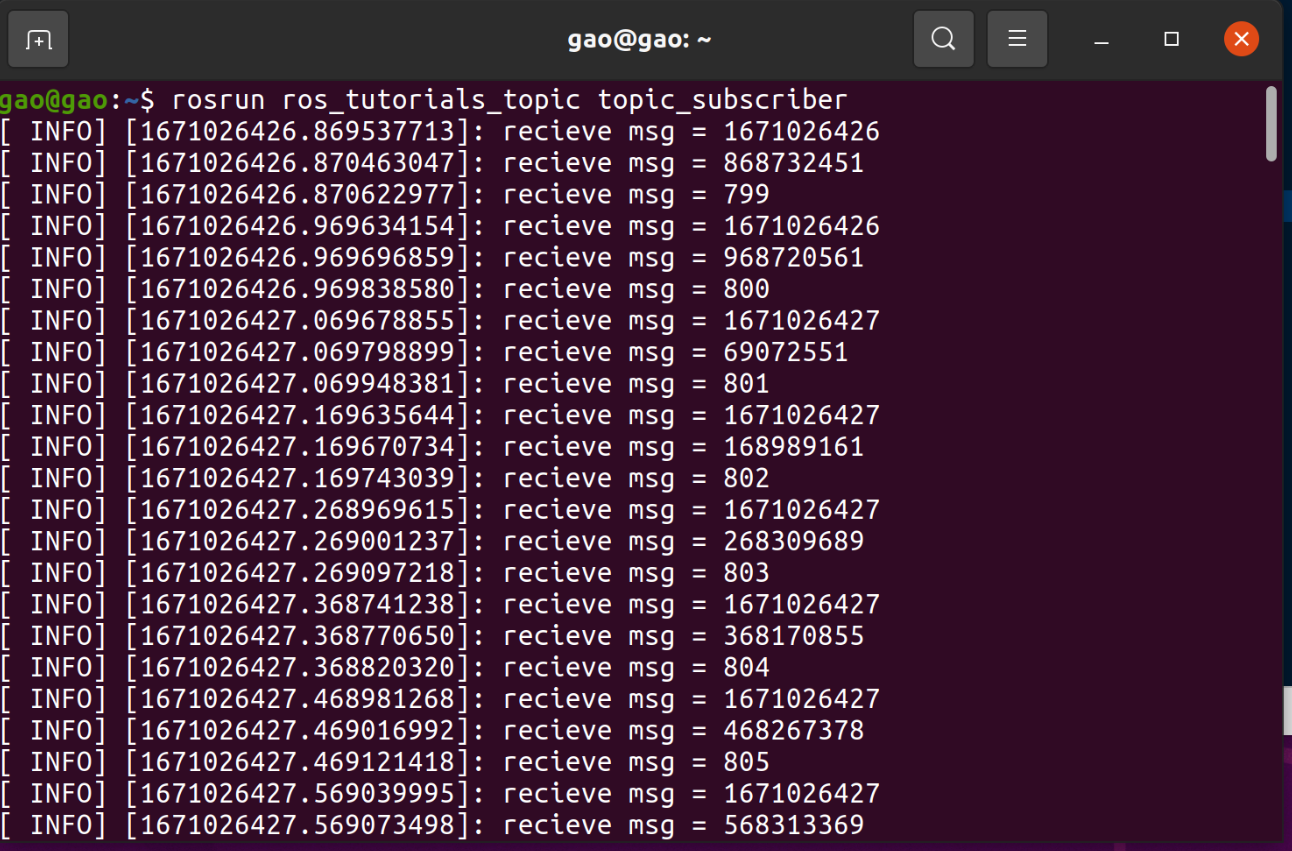
/rosout

/rosout\_agg

$ rostopic echo /ros\_tutorial\_msg



rosrun ros\_tutorials\_topic topic\_subscriber



### 5.2创建和运行服务服务器和客户端节点

**创建功能包**

$ cd ~/catkin\_ws/src

$ catkin\_create\_pkg ros\_tutorials\_service message\_generation std\_msgs roscpp

**修改构建配置文件（CMakeLists.txt）**

$ gedit CMakeLists.txt

以下是为了匹配正在创建的功能包而修改的CMakeLists.txt。同样，有关每个选项的详细说明

cmake\_minimum\_required(VERSION 3.02

project(ros\_tutorials\_topic)

## catkin构建时需要的组件包。

## 是依赖包，是message\_generation、 std\_msgs和roscpp。

## 如果这些功能包不存在，在构建过程中会发生错误。

find\_package(catkin REQUIRED COMPONENTS message\_generation std\_msgs roscpp)

## 服务声明：SrvTutorial.srv

add\_service\_files(FILES SrvTutorial.srv)

## 这是一个设置依赖消息的选项。

## 如果未安装std\_msgs，则在构建过程中会发生错误。

generate\_messages(DEPENDENCIES std\_msgs)

## 这是catkin功能包选项，它描述了库、catkin构建依赖和依赖系统的功能包。

catkin\_package(

LIBRARIES ros\_tutorials\_service

CATKIN\_DEPENDS std\_msgs roscpp

)

## 设置包含目录。

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

## 这是service\_server节点的构建选项。

## 设置可执行文件、目标链接库和附加依赖项。

add\_executable(service\_server src/service\_server.cpp)

add\_dependencies(service\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(service\_server ${catkin\_LIBRARIES})

## 这是节点的构建选项。

add\_executable(service\_client src/service\_client.cpp)

add\_dependencies(service\_client ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(service\_client ${catkin\_LIBRARIES})

**创建服务服务器节点**

#include "ros/ros.h" // ROS的基本头文件

#include "ros\_tutorials\_service/SrvTutorial.h" // SrvTutorial服务头文件（构建后自动生成）

// 如果有服务请求，将执行以下处理

// 将服务请求设置为req，服务响应则设置为res。

bool calculation(ros\_tutorials\_service::SrvTutorial::Request &req,

ros\_tutorials\_service::SrvTutorial::Response &res)

{

// 在收到服务请求时，将a和b的和保存在服务响应值中

res.result = req.a + req.b;

// 显示服务请求中用到的a和b的值以及服务响应result值

ROS\_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);

ROS\_INFO("sending back response: %ld", (long int)res.result);

return true;

}

int main(int argc, char \*\*argv) // 节点主函数

{

ros::init(argc, argv, "service\_server"); // 初始化节点名称

ros::NodeHandle nh; // 声明节点句柄

// 声明服务服务器

// 声明利用ros\_tutorials\_service功能包的SrvTutorial服务文件的

// 服务服务器ros\_tutorials\_service\_server

// 服务名称是ros\_tutorial\_srv，且当有服务请求时，执行calculation函数。

ros::ServiceServer ros\_tutorials\_service\_server = nh.advertiseService("ros\_tutorial\_srv",

calculation);

ROS\_INFO("ready srv server!");

ros::spin(); // 等待服务请求

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(service\_server src/service\_server.cpp)

**创建服务客户端节点**

#include "ros/ros.h" // ROS的基本头文件

#include "ros\_tutorials\_service/SrvTutorial.h" // SrvTutorial服务头文件（构建后自动生成）

#include <cstdlib> // 使用atoll函数所需的库

int main(int argc, char \*\*argv) // 节点主函数

{

ros::init(argc, argv, "service\_client"); // 初始化节点名称

if (argc != 3) // 处理输入值错误

{

ROS\_INFO("cmd : rosrun ros\_tutorials\_service service\_client arg0 arg1");

ROS\_INFO("arg0: double number, arg1: double number");

return 1;

}

ros::NodeHandle nh; // 声明与ROS系统通信的节点句柄

// 声明客户端，声明利用ros\_tutorials\_service功能包的SrvTutorial服务文件的

// 服务客户端ros\_tutorials\_service\_client。

// 服务名称是"ros\_tutorial\_srv"

ros::ServiceClient ros\_tutorials\_service\_client =

nh.serviceClient<ros\_tutorials\_service::SrvTutorial>("ros\_tutorial\_srv");

// 声明一个使用SrvTutorial服务文件的叫做srv的服务

ros\_tutorials\_service::SrvTutorial srv;

// 在执行服务客户端节点时用作输入的参数分别保存在a和b中

srv.request.a = atoll(argv[1]);

srv.request.b = atoll(argv[2]);

// 请求服务，如果请求被接受，则显示响应值

if (ros\_tutorials\_service\_client.call(srv))

{

ROS\_INFO("send srv, srv.Request.a and b: %ld, %ld", (long int)srv.request.a, (long int)srv.request.b);

ROS\_INFO("receive srv, srv.Response.result: %ld", (long int)srv.response.result);

}

else

{

ROS\_ERROR("Failed to call service ros\_tutorial\_srv");

return 1;

}

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(service\_client src/service\_client.cpp)

**创建服务文件**

int64 a

int64 b

---

int64 result

在上述的CMakeLists.txt文中添加了如下选项

add\_service\_files(FILES SrvTutorial.srv)

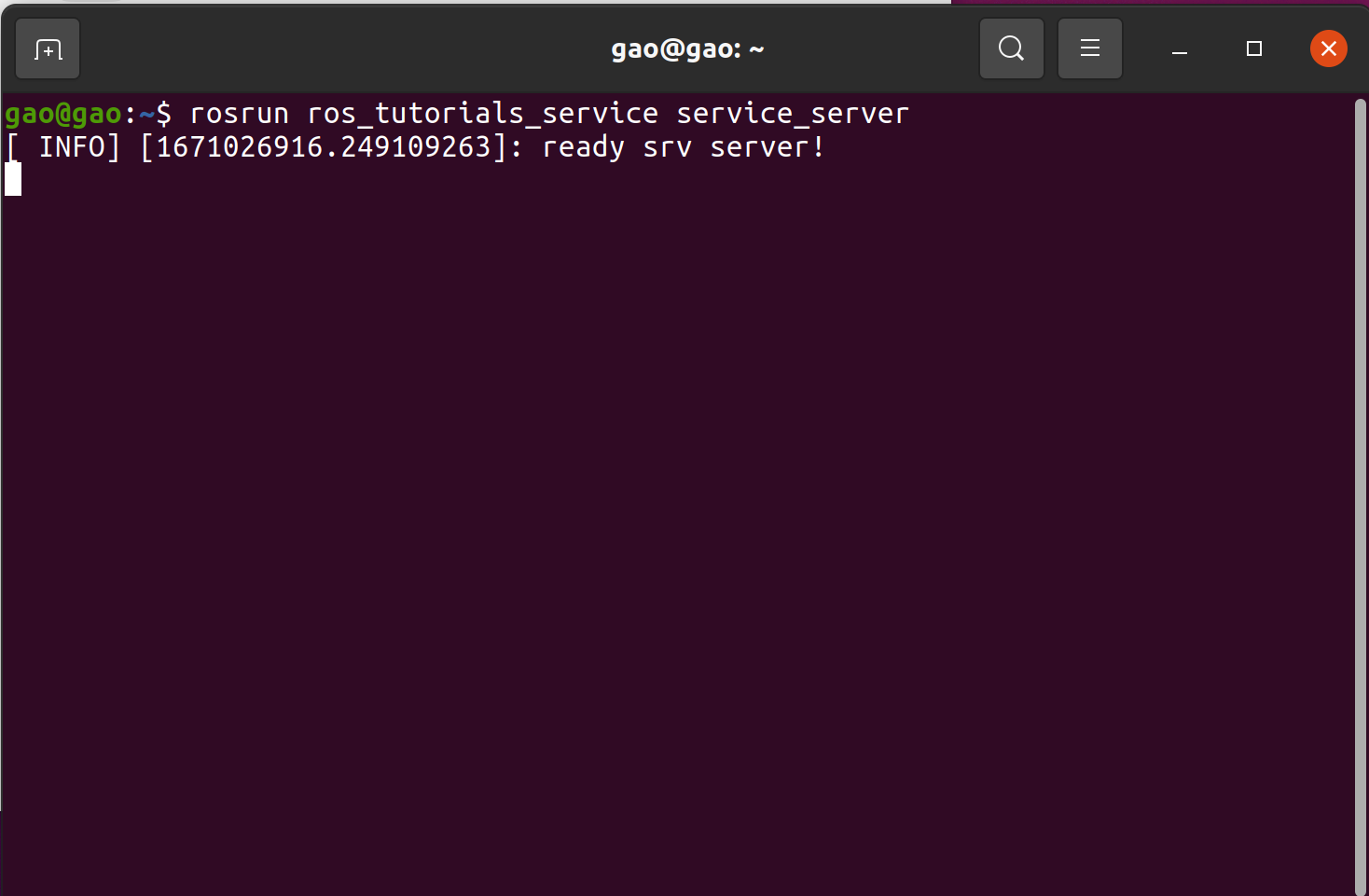
**构建并运行节点**

$ cd ~/catkin\_ws

$ catkin\_make

$ roscore

$ rosrun ros\_tutorials\_service service\_server



$ rosservice list

/ros\_tutorial\_srv

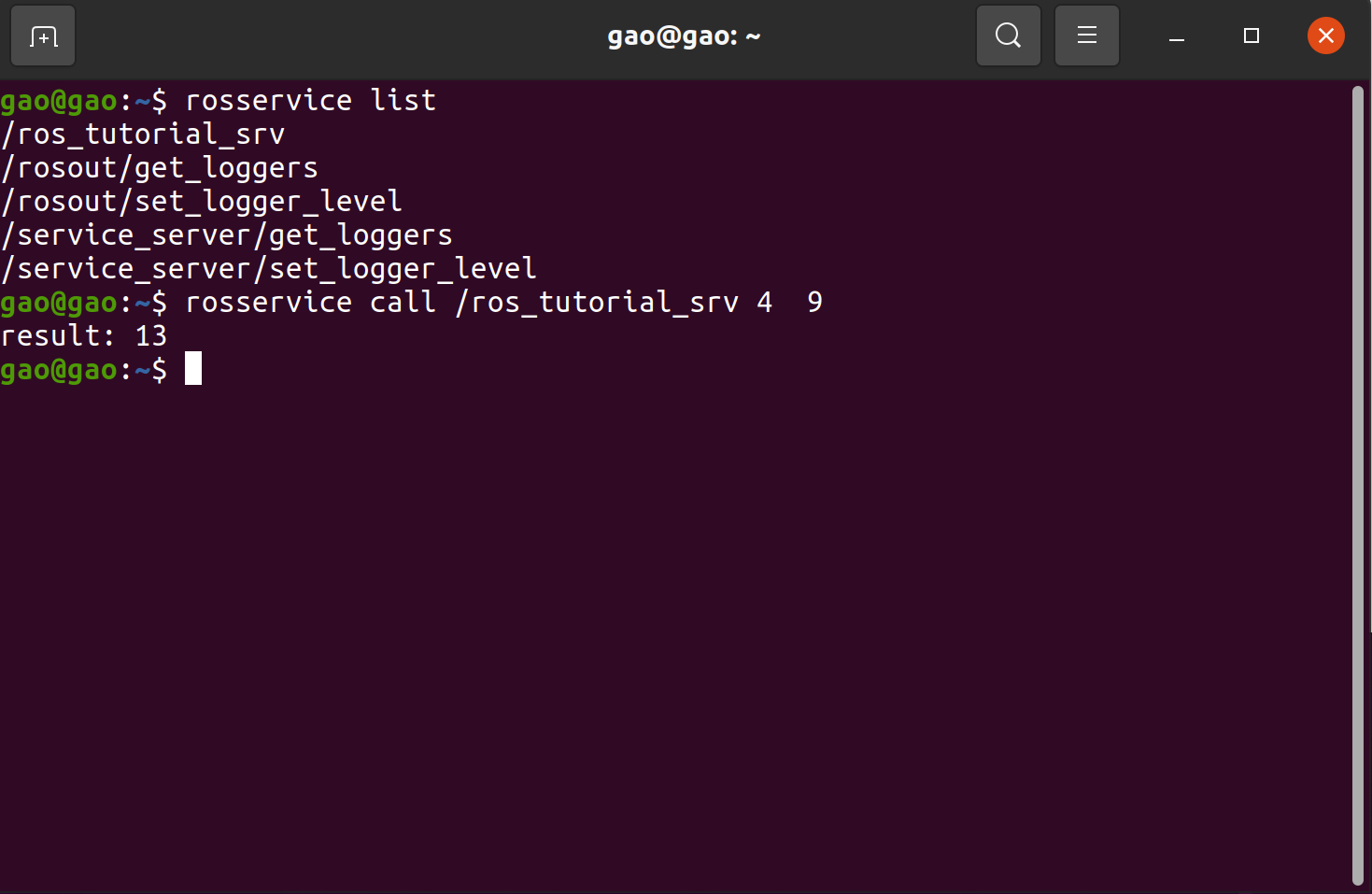
/rosout/get\_loggers

/rosout/set\_logger\_level

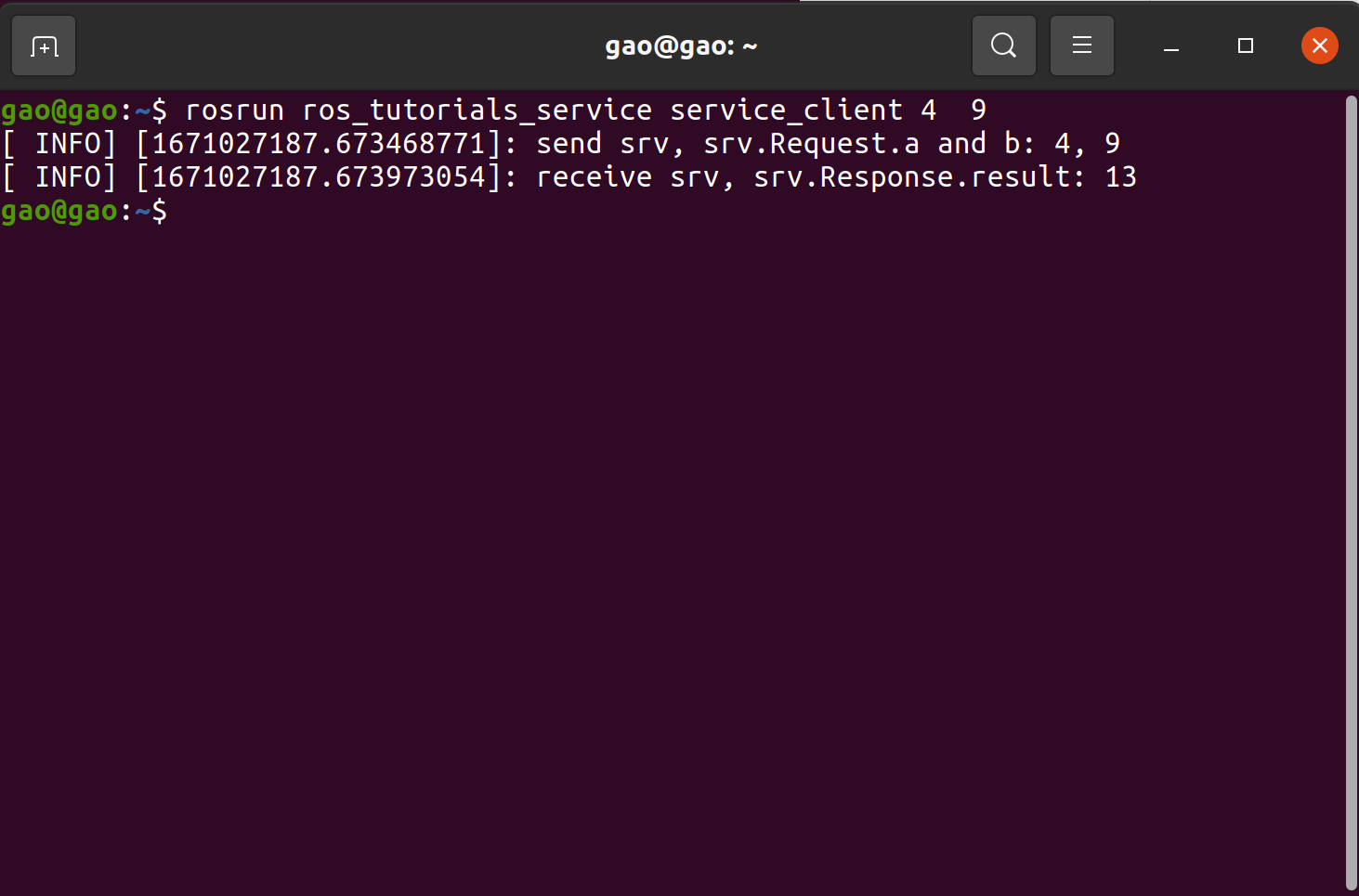
/service\_server/get\_loggers

/service\_server/set\_logger\_level

$ rosservice call /ros\_tutorial\_srv 4 9



rosrun ros\_tutorials\_service service\_client 4 9



### 5.3创建和运行动作服务器和客户端节点

**创建功能包**

$ cd ~/catkin\_ws/src

$ catkin\_create\_pkg ros\_tutorials\_action message\_generation std\_msgs roscpp

**修改构建配置文件（CMakeLists.txt）**

$ gedit CMakeLists.txt

以下是为了匹配正在创建的功能包而修改的CMakeLists.txt。同样，有关每个选项的详细说明

cmake\_minimum\_required(VERSION 3.02

project(ros\_tutorials\_topic)

find\_package(catkin REQUIRED COMPONENTS

message\_generation

std\_msgs

actionlib\_msgs

actionlib

roscpp

)

find\_package(Boost REQUIRED COMPONENTS system)

add\_action\_files(FILES Fibonacci.action)

generate\_messages(DEPENDENCIES actionlib\_msgs std\_msgs)

catkin\_package(

LIBRARIES ros\_tutorials\_action

CATKIN\_DEPENDS std\_msgs actionlib\_msgs actionlib roscpp

DEPENDS Boost

)

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

add\_executable(action\_server src/action\_server.cpp)

add\_dependencies(action\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(action\_server ${catkin\_LIBRARIES})

add\_executable(action\_client src/action\_client.cpp)

add\_dependencies(action\_client ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(action\_client ${catkin\_LIBRARIES})

**创建动作服务器节点**

#include <ros/ros.h> // ROS的基本头文件

#include <actionlib/server/simple\_action\_server.h> // 动作库头文件

#include <ros\_tutorials\_action/FibonacciAction.h> // FibonacciAction动作头文件（生成后自动生成）

class FibonacciAction

{

protected:

// 声明节点句柄

ros::NodeHandle nh\_;

// 声明动作服务器

actionlib::SimpleActionServer<ros\_tutorials\_action::FibonacciAction> as\_;

// 用作动作名称

std::string action\_name\_;

// 声明用于发布的反馈及结果

ros\_tutorials\_action::FibonacciFeedback feedback\_;

ros\_tutorials\_action::FibonacciResult result\_;

public:

// 初始化动作服务器（节点句柄、动作名称、动作后台函数）

FibonacciAction(std::string name) :

as\_(nh\_, name, boost::bind(&FibonacciAction::executeCB, this, \_1), false),

action\_name\_(name)

{

as\_.start();

}

~FibonacciAction(void)

{

}

// 接收动作目标（goal）消息并执行指定动作（此处为斐波那契数列）的函数。

void executeCB(const ros\_tutorials\_action::FibonacciGoalConstPtr &goal)

{

ros::Rate r(1); // 循环周期：1 Hz

bool success = true; // 用作保存动作的成功或失败的变量

// 斐波那契数列的初始化设置，也添加了反馈的第一个（0）和第二个消息（1）

feedback\_.sequence.clear();

feedback\_.sequence.push\_back(0);

feedback\_.sequence.push\_back(1);

// 将动作名称、目标和斐波那契数列的两个初始值通知给用户

ROS\_INFO("%s: Executing, creating fibonacci sequence of order %i with seeds %i, %i",

action\_name\_.c\_str(), goal->order, feedback\_.sequence[0], feedback\_.sequence[1]);

// 动作细节

for(int i=1; i<=goal->order; i++)

{

// 从动作客户端得知动作取消

if (as\_.isPreemptRequested() || !ros::ok())

{

ROS\_INFO("%s: Preempted", action\_name\_.c\_str()); // 通知动作取消

as\_.setPreempted(); // 取消动作

success = false; // 看作动作失败并保存到变量

break;

}

// 除非有动作取消或已达成动作目标

// 将当前斐波纳契数字加上前一个数字的值保存到反馈值。

feedback\_.sequence.push\_back(feedback\_.sequence[i] + feedback\_.sequence[i-1]);

as\_.publishFeedback(feedback\_); // 发布反馈。

r.sleep(); // 按照上面定义的循环周期调用暂歇函数。

}

// 如果达到动作目标值，则将当前斐波那契数列作为结果值传输。

if(success)

{

result\_.sequence = feedback\_.sequence;

ROS\_INFO("%s: Succeeded", action\_name\_.c\_str());

as\_.setSucceeded(result\_);

}

}

};

int main(int argc, char\*\* argv) // 节点主函数

{

ros::init(argc, argv, "action\_server"); // 初始化节点名称

FibonacciAction fibonacci("ros\_tutorial\_action"); // 声明Fibonacci （动作名: ros\_tutorial\_action）

ros::spin(); // 等待动作目标

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(action\_server src/action\_server.cpp)

**创建服务客户端节点**

#include <ros/ros.h> // ROS的基本头文件

#include <actionlib/client/simple\_action\_client.h> // 动作库头文件

#include <actionlib/client/terminal\_state.h> // 动作目标状态头文件

#include <ros\_tutorials\_action/FibonacciAction.h> // FibonacciAction动作头文件（构建后自动生成）

int main (int argc, char \*\*argv) // 节点主函数

{

ros::init(argc, argv, "action\_client"); // 初始化节点名称

// 声明动作客户端（动作名称：ros\_tutorial\_action）

actionlib::SimpleActionClient<ros\_tutorials\_action::FibonacciAction> ac("ros\_tutorial\_action",

true);

ROS\_INFO("Waiting for action server to start.");

ac.waitForServer(); // 等待动作服务器启动

ROS\_INFO("Action server started, sending goal.");

ros\_tutorials\_action::FibonacciGoal goal; // 声明动作目标

goal.order = 20; // 指定动作目标（进行20次斐波那契运算）

ac.sendGoal(goal); // 发送动作目标

// 设置动作完成时间限制（这里设置为30秒）

bool finished\_before\_timeout = ac.waitForResult(ros::Duration(30.0));

// 在动作完成时限内收到动作结果值时

if (finished\_before\_timeout)

{

// 获取动作目标状态值并将其显示在屏幕上

actionlib::SimpleClientGoalState state = ac.getState();

ROS\_INFO("Action finished: %s",state.toString().c\_str());

}

else

ROS\_INFO("Action did not finish before the time out."); // 超过了动作完成时限的情况

//exit

return 0;

}

在前面的CMakeLists.txt文件中，给了生成以下可执行文件的选项

add\_executable(action\_client src/action\_client.cpp)

**创建动作文件**

#goal definition

int32 order

---

#result definition

int32[] sequence

---

#feedback

int32[] sequence

在上述的CMakeLists.txt文中添加了如下选项

add\_action\_files(FILES Fibonacci.action)

**构建并运行节点**

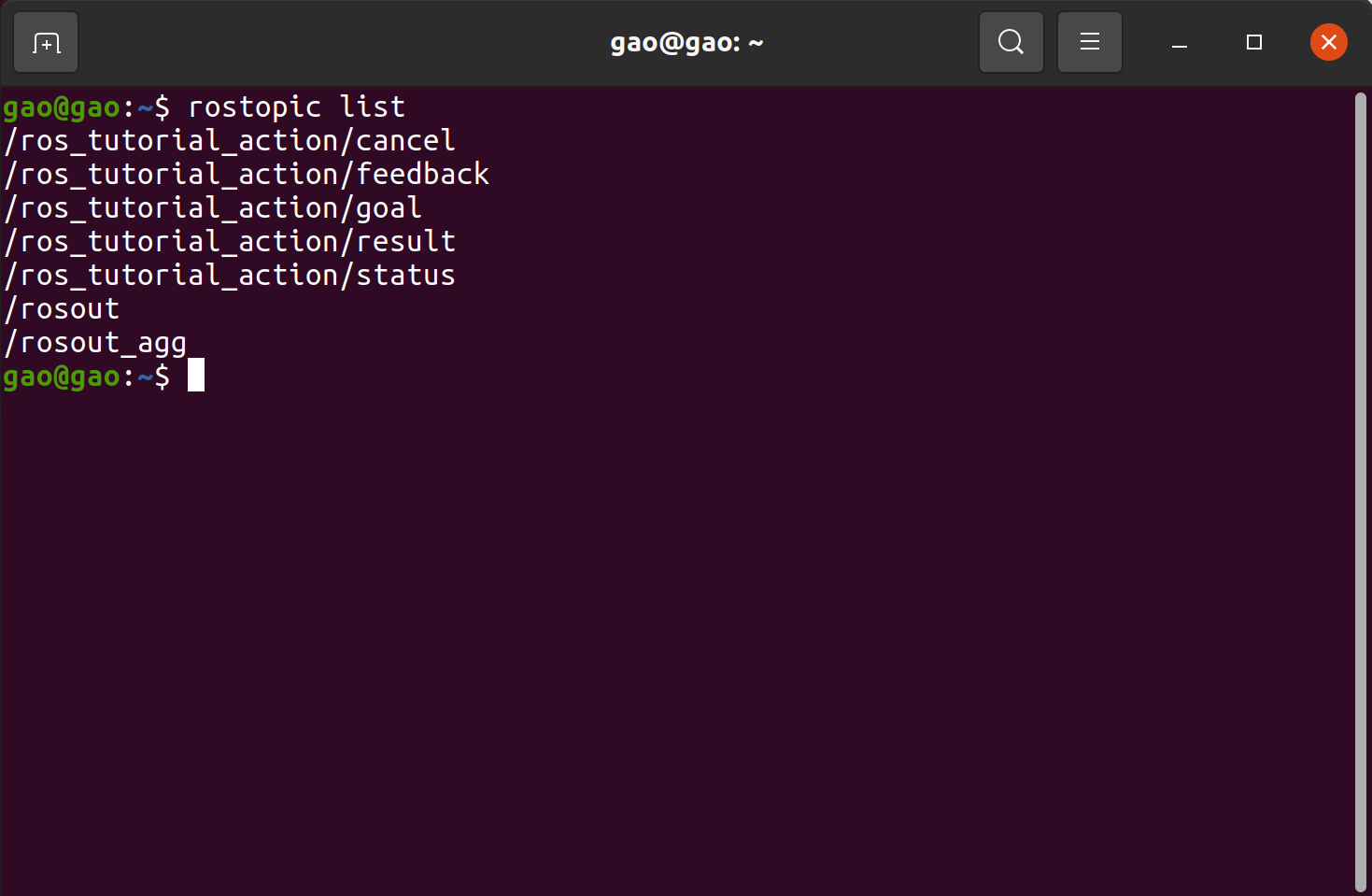
$ cd ~/catkin\_ws

$ catkin\_make

$ roscore

$ rosrun ros\_tutorials\_action action\_server

$ rosservice list

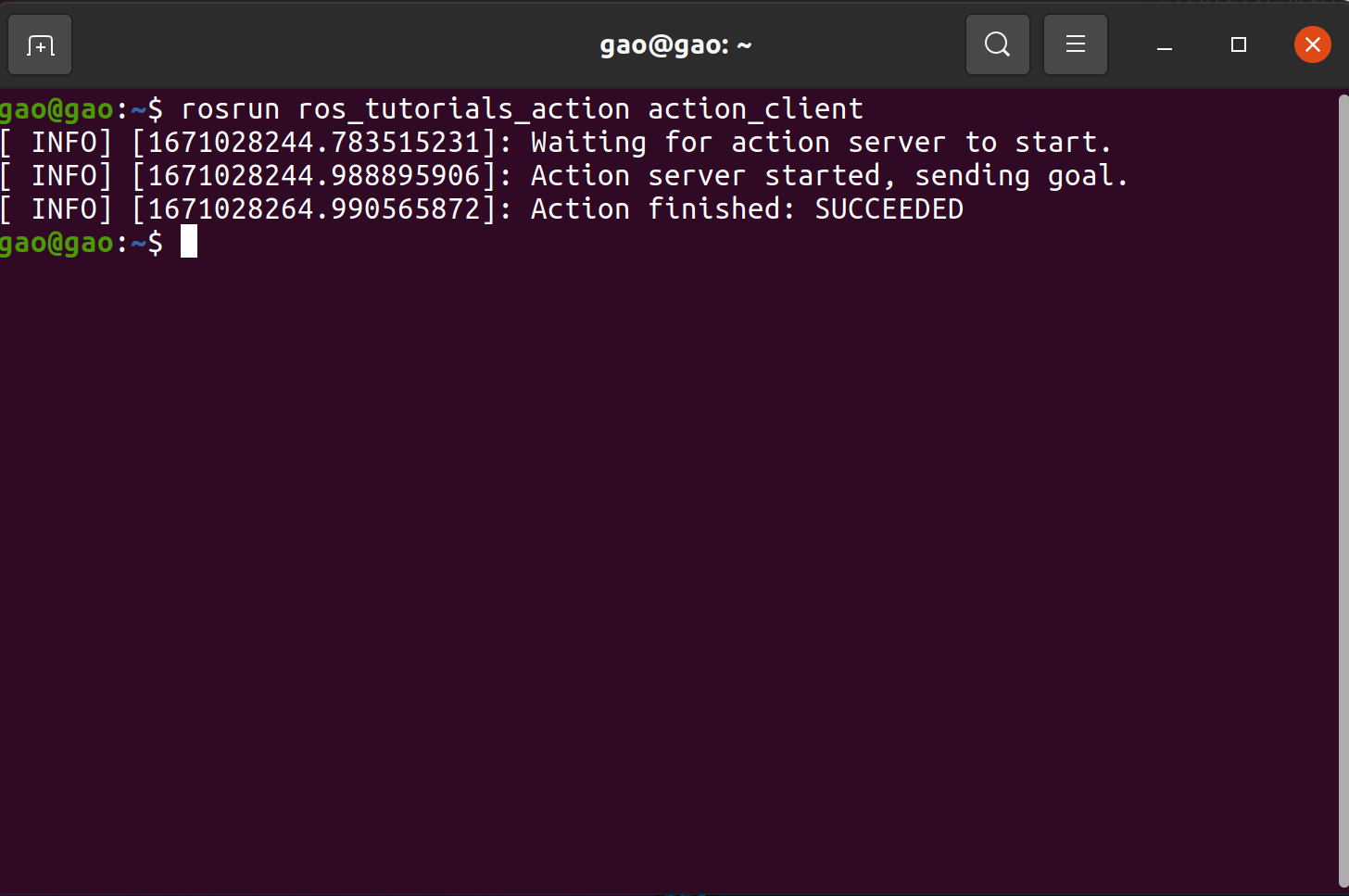


$ rosrun ros\_tutorials\_action action\_client

[INFO] [1495764515.999158825]: Waiting for action server to start.

[INFO] [1495764516.293575887]: Action server started, sending goal.

[INFO] [1495764536.295139830]: Action finished: SUCCEEDED



$ rostopic echo /ros\_tutorial\_action/feedback

## 实验内容

1.话题与服务编程 ：通过代码新生一只海龟，放置在（5,5）点，命名为“turtle2”；通过代码订阅turtle2的实时位置并在终端打印；控制turtle2实现旋转运动 ；

2.动作编程：客户端发送一个运动目标，模拟机器人运动到目标位置的过程，包含服务端和客户端的代码实现，要求带有实时位置反馈

**示例：让小乌龟画圆**

1. 进入工作空间

cd catkin\_ws/src #进入工作空间

catkin\_create\_pkg my\_turtle rospy roscpp #新建my\_turtle\_package包

1. 进入新建的工程文件中

cd ~/catkin\_ws/src/my\_turtle/src

1. 新建画圆程序 circle.cpp

touch circle.cpp

1. 打开circle.cpp，复制以下代码到文件中

gedit circle.cpp

**circle.cpp**

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include<geometry\_msgs/Twist.h> //运动速度结构体类型 geometry\_msgs::Twist的定义文件

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

{

ros::init(argc, argv, "vel\_ctrl"); //对该节点进行初始化操作

ros::NodeHandle n; //申明一个NodeHandle对象n，并用n生成一个广播对象vel\_pub

ros::Publisher vel\_pub = n.advertise<geometry\_msgs::Twist>("/turtle1/cmd\_vel", 10);

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

ros::Rate loop\_rate(10);

//vel\_pub会在主题"/turtle1/cmd\_vel"(机器人速度控制主题)里广播geometry\_msgs::Twist类型的数据

//ros::Rate loopRate(2);

ROS\_INFO("draw\_circle start...");//输出显示信息

while(ros::ok())

{

geometry\_msgs::Twist vel\_cmd; //声明一个geometry\_msgs::Twist 类型的对象vel\_cmd，并将速度的值赋值到这个对象里面

vel\_cmd.linear.x = 2.0;//前后（+-） m/s

vel\_cmd.linear.y = 0.0; //左右（+-） m/s

vel\_cmd.linear.z = 0.0;

vel\_cmd.angular.x = 0;

vel\_cmd.angular.y = 0;

vel\_cmd.angular.z = 1.8; //机器人的自转速度，+左转，-右转，单位是rad/s

vel\_pub.publish(vel\_cmd); //赋值完毕后，发送到主题"/turtle1/cmd\_vel"。机器人的核心节点会从这个主题接受发送过去的速度值，并转发到硬件体上去执行

std\_msgs::String msg;

std::stringstream ss;//定义输出流对象

ss <<vel\_cmd;

msg.data = ss.str();

/\*\*

\* 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);

ros::spinOnce();//调用此函数给其他回调函数得以执行(比例程未使用回调函数)

//loopRate.sleep();

}

return 0;

}

**创建接收程序**

当前目录不改变，新建小乌龟画圆接收程序，用于实时更新位置信息

touch callbackcircle.cpp

打开callbackcircle.cpp 输入如下代码

gedit callbackcircle.cpp

**callbackcircle.cpp**

#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)

{

ROS\_INFO("it is location: [%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, "yuan");

/\*\*

\* 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);

/\*\*

\* 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();

return 0;

}

**修改 CMakelist.txt**

cd ~/catkin\_ws\_src/my\_turtle

打开 CMakelist.txt文件，在文件末尾添加如下文件

gedit CMakelist.txt

add\_executable(circle src/circle.cpp)

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

#add\_dependencies(circle comm\_generate\_messages\_cpp)

add\_executable(callbackcircle src/callbackcircle.cpp)

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

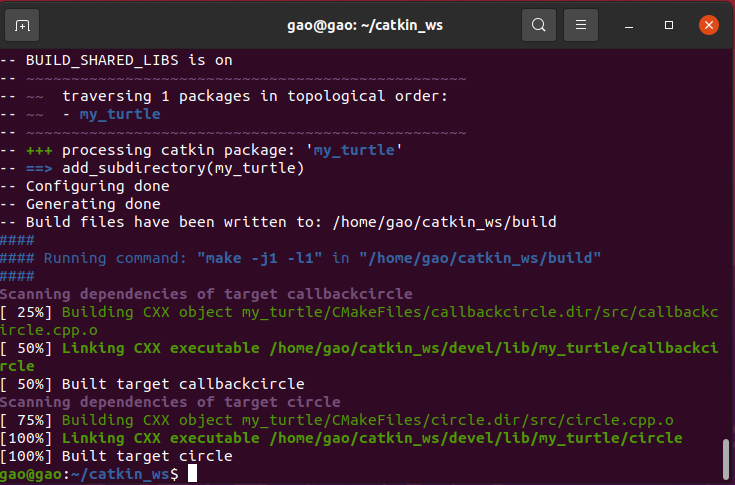
**然后保存文件并关闭**

**编译程序**

cd ~/catkin\_ws

catkin\_make

编译成功显示如下



**运行程序**

新建终端，启动ROS

roscore

新建终端，注册工作空间

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

运行接收程序

rosrun my\_turtle callbackcircle

新建终端，启动小乌龟

rosrun turtlesim turtlesim\_node

新建终端，注册工作空间

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

运行画圆程序

rosrun my\_turtle circle

执行后，具体如下

