ROS小课堂

ROS机器人操作系统基础教程

作者： corvin

日期：2017-7-23

是故无冥冥之志者，无昭昭之明；无惛惛之事者，无赫赫之功。--先秦_劝学_荀子

ROS小课堂

一、ROS文件系统和软件包简介

1. ROS文件系统介绍
   - 快速了解ROS文件系统概念
   - 认识文件系统工具
2. 创建和编译ROS软件包
   - 创建工作空间
   - 创建软件包
   - 软件包的依赖关系
   - 删除工作空间的软件包
   - 编写代码
   - 编译软件包
   - 运行节点

1. ROS文件系统介绍

（1）快速了解ROS文件系统概念

- 软件包集（Stack）：如果你将几个具有某些功能的软件包组织在一起，那么你将会获得一个软件包集。在ROS系统中，存在大量的不同用途的软件包集，例如导航软件包集。
- Packages: 软件包：是ROS应用程序代码的组织单元，每个软件包都可以包含程序库、可执行文件、脚本或者其它手动创建的东西。
- package.xml：清单文件，关于'软件包'相关信息的描述,用于定义软件包相关元信息之间的依赖关系，这些信息包括版本、维护者和许可协议，编译依赖和运行依赖等。
- CMakeLists.txt：编译配置文件，使用cmake进行程序编译的时候，会根据CMakeLists.txt这个文件进行一步一步的处理，然后形成一个MakeFile文件，系统再通过这个文件的设置进行程序的编译。

- catkin_make如何编译整个工作空间的软件包呢？

① gcc是GNU Compiler Collection（就是GNU编译器套件），也可以简单认为是编译器，当你有一个源码文件时就可以直接调用gcc来编译;

② 但是当你的程序包含很多个源文件时，用gcc命令逐个去编译时，你就很容易混乱而且工作量大，这时候就需要make工具了，但make本身并没有编译和链接的功能，而是用类似于批处理的方式通过调用makefile文件中用户指定的命令来进行编译和链接的。

③ makefile是什么？简单的说就像一本菜谱，make工具就像厨师，厨师按照菜谱知道了一道菜该如何操作，make工具就根据makefile中的命令进行编译和链接的。

④ makefile在一些简单的工程完全可以人工手下，但是当工程非常大的时候，手写makefile也是非常麻烦的，如果换了个平台makefile又要重新修改。

⑤ 这时候就需要cmake这个工具，cmake可以更加简单的生成makefile文件给make用。当然cmake还可以跨平台生成对应平台能用的makefile，你不用再自己去修改了。

⑥ cmake根据什么生成makefile呢？它又要根据CMakeLists.txt文件去生成makefile，这个CMakeLists.txt大家应该比较熟悉了，就是我们在新创建一个软件包时自动帮我们生成的。

⑦ 最后catkin_make是将cmake与make的编译方式做了一个封装的指令工具，规范了工作路径与生成文件路径，而且在新建个工作空间时就帮我们创建了一个顶层的CMakeLists.txt文件，它会递归的寻找到当前工作空间下的所有软件包内的CMakeLists.txt依次来编译每一个软件包。

```
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  my_pkg  robot_bringup  subscribe_pkg  test_pkg  third_pkg
corvin@workspace:~/ros_workspace/src$ ls -l
total 20
lrwxrwxrwx 1 corvin corvin   50 7月  21 20:07 CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
drwxrwxr-x 5 corvin corvin 4096 7月  18 18:24 my_pkg
drwxrwxr-x 3 corvin corvin 4096 7月  19 09:51 robot_bringup
drwxrwxr-x 4 corvin corvin 4096 7月  21 20:26 subscribe_pkg
drwxrwxr-x 4 corvin corvin 4096 7月  18 15:42 test_pkg
drwxrwxr-x 7 corvin corvin 4096 7月  21 16:33 third_pkg
```

ROS小课堂

- 查看工作空间和软件包目录结构

```
corvin@workspace:~/catkin_ws/src$ ls
arbotix_ros      learning_tf  r2d2  urdf  ros_arduino_bridge     urdf_t
CMakeLists.txt   navigation                 wist_keyboard  usb_cam    voice_system
corvin@workspace:~/catkin_ws/src$ cd navigation/
corvin@workspace:~/catkin_ws/src/navigation$ ls
amcl                    dwa_local_planner  move_slow_and_clear  robot_pose_ekf
base_local_planner      fake_localization  nav_core             rotate_recovery
carrot_planner          global_planner     navfn                voxel_grid
clear_costmap_recovery  map_server         navigation
costmap_2d              move_base          README.md
corvin@workspace:~/catkin_ws/src/navigation$ tree map_server/
map_server/
├── CHANGELOG.rst
├── CMakeLists.txt
├── include
│   └── map_server
│       └── image_loader.h
├── package.xml
├── scripts
│   └── crop_map
├── src
│   ├── image_loader.cpp
│   ├── main.cpp
│   ├── map_saver.cpp
│   └── map_server.dox
└── test
    ├── consumer.py
    ├── rtest.cpp
    ├── rtest.xml
    ├── test_constants.cpp
    ├── test_constants.h
    ├── testmap.bmp
    ├── testmap.png
    ├── testmap.yaml
    └── utest.cpp

5 directories, 18 files
```

功能复杂的软件包集

功能单一的软件包

软件包集里包含很多功能单一软件包，将这些单一的软件包集合起来就能完成复杂的任务。

编译配置文件

头文件目录

软件包的相关描述信息

用于存放可执行脚本文件

src目录存放软件包的源码

（2）认识文件系统工具

程序代码是分布在众多ROS软件包当中，当使用命令行工具（比如ls和cd）来浏览时会非常繁琐，因此ROS提供了专门的命令工具来简化这些操作。

- 使用 rospack：获取软件包的有关信息。

当ROS安装包越来越多时，该命令就会非常有用，尤其是当安装包有上百个之多时，根本无法准确记清各个软件包的路径或是否存在某软件包等。

```
corvin@workspace:~/catkin_ws/src$ rospack list-names|wc -l
268
corvin@workspace:~/catkin_ws/src$ rospack list|wc -l
268
corvin@workspace:~/catkin_ws/src$ rospack list|grep voice
voice_system /home/corvin/catkin_ws/src/voice_system
```

统计当前系统共安装多少个软件包

通过rospack list列出所有软件包，然后通过grep命令来过滤软件包

```
corvin@workspace:~$ rospack help
USAGE: rospack <command> [options] [package]
  Allowed commands:
    help
    cflags-only-I     [--deps-only] [package]
    cflags-only-other [--deps-only] [package]
    depends           [package] (alias: deps)
    depends-indent    [package] (alias: deps-indent)
    depends-manifests [package] (alias: deps-manifests)
    depends-msgsrv    [package] (alias: deps-msgsrv)
    depends-on        [package]
    depends-on1       [package]
    depends-why --target=<target> [package] (alias: deps-why)
    depends1          [package] (alias: deps1)
    export [--deps-only] --lang=<lang> --attrib=<attrib> [package]
    find [package]    查询软件包的路径信息
    langs
    libs-only-L     [--deps-only] [package]
    libs-only-l     [--deps-only] [package]
    libs-only-other [--deps-only] [package]
    list    列出当前系统中安装的软件包名及其路径信息
    list-duplicates
    list-names    只列出当前系统中安装的软件包名称
    plugins --attrib=<attrib> [--top=<toppkg>] [package]
    profile [--length=<length>] [--zombie-only]
    rosdep  [package] (alias: rosdeps)
    rosdep0 [package] (alias: rosdeps0)
    vcs  [package]
    vcs0 [package]
  Extra options:
    -q     Quiets error reports.

 If [package] is omitted, the current working directory
 is used (if it contains a package.xml or manifest.xml).
```

- 使用 roscd：直接切换(cd)工作目录到某个软件包或者软件包集当中。

```
corvin@workspace:~$ rospack list|grep voice
voice_system /home/corvin/catkin_ws/src/voice_system
corvin@workspace:~$ roscd voice_system
corvin@workspace:~/catkin_ws/src/voice_system$ cd
corvin@workspace:~$ cd catkin_ws/src/voice_system/
corvin@workspace:~/catkin_ws/src/voice_system$
```

- 使用 rosls：直接按软件包的名称而不是绝对路径执行ls命令（罗列目录）。

```
corvin@workspace:~$ rosls voice_system
CMakeLists.txt  include  package.xml  src
corvin@workspace:~$ ls catkin_ws/src/voice_system/
CMakeLists.txt  include  package.xml  src
```

- Tab 自动补全输入：当要输入一个完整的软件包名称有时会比较繁琐，可以输入前面一部分然后敲击tab键即可自动根据前半部分的输入来补全。

```
corvin@workspace:~$ roscd turtle
turtle_actionlib/  turtlesim/          turtle_tf/          turtle_tf2/
corvin@workspace:~$ roscd turtlesim/
corvin@workspace:/opt/ros/kinetic/share/turtlesim$
```

敲击tab自动补全，列出所有可能的输入

根据提示，找到想要的名称

- 使用 rosed：直接编辑某个软件包中的可编辑文件。

```
corvin@workspace:~$ rosed voice_system
CMakeLists.txt          msp_errors.h          speech_recognizer.cpp  xf_asr.cpp
formats.h               msp_types.h           speech_recognizer.h    xf_asr_node
linuxrec.cpp            package.xml           tags                   xf_tts.cpp
linuxrec.h              qisr.h                tuling_nlu.cpp         xf_tts_node
msp_cmn.h               qtts.h                tuling_nlu_node
corvin@workspace:~$ rosed voice_system xf_tts.cpp
corvin@workspace:~$ vim catkin_ws/src/voice_system/src/xf_tts.cpp
```

敲击tab列出所有可以编辑的文件

rosed 包名 文件名

vim编辑需要输入完整的编辑文件路径，比较麻烦

- 使用 roscp：直接从某个软件包中复制某个文件到指定目录下。

```
corvin@workspace:~$ roscp help
usage: roscp package filename target

Copy a file from a package to target location.
corvin@workspace:~$ roscp voice_system xf_tts.cpp .
corvin@workspace:~$ cp catkin_ws/src/voice_system/src/xf_tts.cpp .
```

将voice_system软件包下的xf_tts.cpp复制到当前目录下

2.创建和编译ROS软件包

（1）创建软件包

所有的 ROS 软件，包括我们创建的软件，都被组织成软件包。在我们写任何程序之前，第一步是创建一个容纳我们的软件包的工作区，然后再创建软件包本身。

- 创建工作空间

我们创建的工作区其实就是一个文件夹，用于存储我们的软件包，工作区可以任意命名成自己喜欢的名字，任意选择存储在home目录下的自己喜欢的位置。

ROS小课堂

```
corvin@workspace:~$ mkdir -p ros_workspace/src
corvin@workspace:~$ tree ros_workspace/
ros_workspace/
└── src

1 directory, 0 files
corvin@workspace:~$ cd ros_workspace/
corvin@workspace:~/ros_workspace$ catkin_make
Base path: /home/corvin/ros_workspace
Source space: /home/corvin/ros_workspace/src
Build space: /home/corvin/ros_workspace/build
Devel space: /home/corvin/ros_workspace/devel
Install space: /home/corvin/ros_workspace/install
C                                          e/src/CM
C
####
```

创建工作空间目录

编译命令

即使没有软件包，空的工作空间也可以编译，会在当前目录下生成build和devel目录

对于许多用户来说，没有必要使用多个ROS工作区。但是ROS的catkin编译系统，试图一次性编译同一个工作区中的所有功能包。因此，如果你的工作涉及大量的功能包，或者涉及几个相互独立的的项目，则维护数个独立的工作区才可能是有必要的。

- 创建软件包：创建一个新ROS软件包的命令应该在你工作区中的src目录下运行

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg test_pkg std_msgs rospy roscpp
Created file test_pkg/package.xml
Created file test_pkg/CMakeLists.txt
Created folder test_pkg/include/test_pkg
Created folder test_pkg/src
Successfully created files in /home/corvin/ros_workspace/src/test_pkg. Please adjust the values in package.xml.
corvin@workspace:~/ros_workspace/src$ tree test_pkg/
test_pkg/
├── CMakeLists.txt
├── include
│   └── test_pkg
├── package.xml
└── src

3 directories, 2 files
```

创建命令　软件包名　创建软件包需要的依赖关系

编译配置文件，一个cmake的脚本文件

清单文件，软件包相关描述文件

创建的软件包目录结构

其实这个功能包创建命令没有做太多工作，它只不过创建了一个存放这个功能包的目录，并在那个目录下生成了两个配置文件。

- 程序包依赖关系

a) 一级依赖：在使用catkin_create_pkg命令时提供了几个程序包作为依赖包，现在我们可以使用rospack命令工具来查看一级依赖包。

```
corvin@workspace:~/ros_workspace$ rospack depends1 test_pkg
[rospack] Error: no such package test_pkg
corvin@workspace:~/ros_workspace$ source devel/setup.bash
corvin@workspace:~/ros_workspace$ rospack depends1 test_pkg
roscpp
rospy
std_msgs
```

使用rospack depends1命令列出一级依赖包

当新创建了工作空间时，ROS知道该工作空间的环境变量才能定位到该工作空间下的软件包，因此，为了方便每次打开终端都自动配置环境变量需要在home目录下的.bashrc文件最下面添加如下信息：

```
#ros config
source /opt/ros/kinetic/setup.bash
source ~/ros_workspace/devel/setup.bash
```

rospack列出了在运行catkin_create_pkg命令时作为参数的依赖包，这些依赖包随后保存在package.xml文件中。

ROS小课堂

b) 间接依赖：在很多情况中，一个依赖包还会有它自己的依赖包，比如，rospy还有其它依赖包。

```
corvin@workspace:~$ rospack depends1 rospy
genpy
roscpp
rosgraph
rosgraph_msgs
roslib
std_msgs
corvin@workspace:~$ rospack depends test_pkg
cpp_common
rostime
roscpp_traits
roscpp_serialization
catkin
genmsg
genpy
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
rospack
roslib
rospy
```

列出test_pkg的一级依赖包rospy所依赖的一级依赖包

一个程序包还可以有好几个间接的依赖包，幸运的是使用rospack可以递归检测出所有的依赖包。

ROS 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，而且首字符必须是一个小写字母。一些 ROS工具，包括 catkin，不支持那些不遵循此命名规范的包。

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg ROS_PKG std_msgs rospy roscpp
WARNING: Catkin package name "ROS_PKG" does not follow the naming conventions. It should start with a lower case l
etter and only contain lower case letters, digits, and underscores.
Created file ROS_PKG/CMakeLists.txt
Created file ROS_PKG/package.xml
Created folder ROS_PKG/include/ROS_PKG
Created folder ROS_PKG/src
Successfully created files in /home/corvin/ros_workspace/src/ROS_PKG. Please adjust the values in package.xml.
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  ROS_PKG  test_pkg
```

创建软件包过程中的警告信息，虽然也能创建成功，但是最好还是需要遵守规范。

```
-- BUILD_SHARED_LIBS is on
WARNING: Catkin package name "ROS_PKG" does not follow the naming conventions. It should start with a lower case l
etter and only contain lower case letters, digits, and underscores.
-- ~~  traversing 2 packages in topological order:
-- ~~  - ROS_PKG
-- ~~  - test_pkg
-- +++ processing catkin package: 'ROS_PKG'
-- ==> add_subdirectory(ROS_PKG)
WARNING: Catkin package name "ROS_PKG" does not follow the naming conventions. It should start with a lower case l
etter and only contain lower case letters, digits, and underscores.
-- +++ processing catkin package: 'test_pkg'
-- ==> add_subdirectory(test_pkg)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/corvin/ros_workspace/build
####
#### Running command: "make -j4 -l4" in "/home/corvin/ros_workspace/build"
####
```

不符合命名规范的软件包虽然也能编译通过，但是编译过程中会有警告信息，建议遵守软件包命名规范。
即软件包名称仅能由小写字母、数字和下划线组成。

- 删除工作空间的软件包

删除不想要的工作空间的软件包也很简单，直接rm整个软件包目录即可，然后重新编译整个工作空间。

```
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  ROS_PKG  test_pkg
corvin@workspace:~/ros_workspace/src$ rm -rf ROS_PKG/
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  test_pkg
corvin@workspace:~/ros_workspace/src$ cd ..
corvin@workspace:~/ros_workspace$ catkin_make
Base path: /home/corvin/ros_workspace
Source space: /home/corvin/ros_workspace/src
Build space: /home/corvin/ros_workspace/build
Devel space: /home/corvin/ros_workspace/devel
Install space: /home/corvin/ros_workspace/install
####
#### Running command: "cmake /home/corvin/ros_workspace/src -
 -DCMAKE_IN                    corvin/ros_workspace/install -G
"
####
```

删除软件包

重新编译整个工作空间

编译过程输出...

- 编写测试代码

为了演示如何正确的编译软件包，我们需要在测试软件包test_pkg中编写一个测试代码才能演示如何编译软件包。

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

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);
  int count = 0;

  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;

    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

在test_pkg软件包的src目录下创建test.cpp源码文件，这里的测试代码就是创建一个话题，并向其中发送字符串hello world。后面教程会详细介绍什么是话题。

ROS小课堂

- 编译软件包

我们主要需要修改CMakeLists.txt这个编译配置文件。

```
133 ## Declare a C++ executable
134 ## With catkin_make all packages are built within a single CMake context
135 ## The recommended prefix ensures that target names across packages don't collide
136 add_executable(${PROJECT_NAME}_node src/test.cpp)
137
138 ## Rename C                    prefi
139 ## The above recommended prefix causes long target names, the following renames the
140 ## target back to the shorter version for ease of user use
141 ## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
142 # set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")
143
144 ## Add cmake target dependencies of the executable
145 ## same as for the library above
146 # add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPO#
147
148 ## Specify libraries to link a library or executable target against
149 target_link_libraries(${PROJECT_NAME}_node
150   ${catkin_LIBRARIES}
151 )
```

最后的执行文件节点名

根据需要修改成自己的源码文件

编译执行文件所需要的依赖库

当修改完CMakeLists.txt文件后，就可以在工作空间的根目录下使用catkin_make进行编译了。

```
corvin@workspace:~/ros_workspace$ catkin_make
Base path: /home/corvin/ros_workspace
Source space: /home/corvin/ros_workspace/src
Build space: /home/corvin/ros_workspace/build
Devel space: /home/corvin/ros_workspace/devel
Install space: /home/corvin/ros_workspace/install
####
#### Running command: "make cmake_check_build_system" in "/home/corvin/ros_workspace/build"
####
-- Using CATKIN_DEVEL_PREFIX: /home/corvin/ros_workspace/devel
-- Using CMAKE_PREFIX_PATH: /home/corvin/ros_workspace/devel;/home/corvin/catkin_ws/devel;/hom
e/corvin/YoYoRobot/ROSCode/devel;/opt/ros/kinetic
-- This workspace overlays: /home/corvin/ros_workspace/devel;/home/corvin/catkin_ws/devel;/hom
e/corvin/YoYoRobot/ROSCode/devel;/opt/ros/kinetic
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/corvin/ros_workspace/build/test_results
-- Found gtest sources under '/usr/src/gtest': gtests will be built
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.7.6
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 1 packages in topological order:
-- ~~  - test_pkg
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'test_pkg'
-- ==> add_subdirectory(test_pkg)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/corvin/ros_workspace/build
####
#### Running command: "make -j4 -l4" in "/home/corvin/ros_workspace/build"
####
Scanning dependencies of target test_pkg_node
[ 50%] Building CXX object test_pkg/CMakeFiles/test_pkg_node.dir/src/test.cpp.o
[100%] Linking CXX executable /home/corvin/ros_workspace/devel/lib/test_pkg/test_pkg_node
[100%] Built target test_pkg_node
corvin@workspace:~/ros_workspace$
```

在工作空间的根目录下执行catkin_make编译命令

当最后出现100%时说明编译完成，编译过程未出现错误，当出现错误时我们需要根据提示信息进行相应的修改，使其可以正常编译即可。

- 运行节点

当编译过程没有错误时，接下来就可以使用rosrun命令来启动节点了。

```
corvin@workspace:~/ros_workspace$ rosrun test_pkg test_pkg_node
[ERROR] [1500209196.888631784]: [registerPublisher] Failed to contact master at [localhost:11311].  Retrying...
^Ccorvin@workspace:~/ros_workspace$ roscore
... logging to /home/corvin/.ros/log/d675c4c2-6a24-11e7-aa59-1002b5ca5e31/roslaunch-workspace-28824.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://workspace:38961/
ros_comm version 1.12.7


SUMMARY
========
```

在使用rosrun运行节点时，需要首先启动roscore，直接在终端中输入roscore回车即可启动master节点。
rosrun的使用规范是：

rosrun 包名 节点名

```
corvin@workspace:~/ros_workspace$ rosrun test_pkg test_pkg_node
[ INFO] [1500209220.416887289]: hello world 0
[ INFO] [1500209220.51694
[ INFO] [1500209220.61691
[ INFO] [1500209220.71693
[ INFO] [1500209220.816948734]: hello world 4
[ INFO] [1500209220.916899138]: hello world 5
[ INFO] [1500209221.016909205]: hello world 6
[ INFO] [1500209221.116916534]: hello world 7
[ INFO] [1500209221.216898658]: hello world 8
[ INFO] [1500209221.316896364]: hello world 9
[ INFO] [1500209221
[ INFO] [1500209221
```

当启动了master节点后，才可以启动其他的节点。

节点的打印输出...

ROS小课堂

二、如何引用头文件、库文件和如何编写launch文件

1. 如何引用自定义头文件
   - 引用当前软件包内的头文件
   - 引用同工作空间中其他软件包的头文件
2. 如何引用第三方库文件
   - 手工自己创建测试用的动态库
   - 编辑代码和配置文件来引用动态库
3. 如何编写launch文件
   - 编写简单的launch文件
   - 介绍常用的节点属性
   - 编写launch时的注意事项

1. 如何引用自定义头文件

- 引用当前软件包内的头文件

在编写代码时我们经常会需要引用头文件，引用公用的头文件很容易，因为它们已经在标准库头文件路径中。但是如果要引用自定义的头文件就稍微麻烦点，我们首先查看软件包的目录结构，需要在正确的目录下创建头文件并修改CMakeLists.txt文件这样才能正确编译，下面来举例说明：

```
corvin@workspace:~/ros_workspace/src$ tree test_pkg/
test_pkg/
├── CMakeLists.txt
├── include
│   └── test_pkg
├── package.xml
└── src
    └── main.cpp

3 directories, 3 files
```

当创建软件包时默认提供的头文件存放目录，我们需要在此目录下创建自定义的头文件

源码文件

```
corvin@workspace:~/ros_workspace/src/test_pkg$ tree
.
├── CMakeLists.txt
├── include
│   └── test_pkg
│       └── test_pkg.h
├── package.xml
└── src
    └── main.cpp

3 directories, 4 files
```

创建自定义的头文件，这个名称可以任意修改成自己喜欢的文件名，我这里命名为test_pkg

```
1 #ifndef _TEST_PKG_
2 #define _TEST_PKG_
3
4 #define TEST_PKG_VER   "1.0.0"
5 #define INIT_COUNT     100
6
7 int addTwoNum(int a, int b);
8
9 #endif
```

防止头文件被重复包含引用

版本号

随便一个函数声明

- 修改源码文件，引用自定义头文件

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include "test_pkg/test_pkg.h"

int addTwoNum(int a, int b)
{
    return a+b;
}

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);
  int count = INIT_COUNT;

  ROS_INFO("test_pkg version:%s", TEST_PKG_VER);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

引用自定义的头文件

头文件中函数声明的实现

使用头文件中宏定义来初始化count

输出头文件中定义的版本号字符串

修改后的main.cpp源码文件

ROS小课堂

```
112 ############
113 ## Build ##
114 ############
115
116 ## Specify additional locations of header files
117 ## Your package locations should be listed before other locations
118 include_directories(
119   include
120   ${catkin_INCLUDE_DIRS}
121 )
122
123 ## Declare a C++ library
124 # add_library(${PROJECT_NAME}
125 #   src/${PROJECT_NAME}/test_pkg
126 # )
127
```

指定附加的头文件存放位置，自己头文件位置需要在其他包的前面。

修改后的CMakeLists.txt文件

```
corvin@workspace:~/ros_workspace$ rosrun test_pkg test_pkg_node
[ INFO] [1500348463.931086475]: test_pkg version:1.0.0
[ INFO] [1500348463.931121868]: hello world 100
[ INFO] [1500348463.xxxxxxxxx]: hello world 101
[ INFO] [1500348463.xxxxxxxxx]: hello world 102
[ INFO] [1500348464.231373999]: hello world 103
[ INFO] [1500348464.331365050]: hello world 104
```

编译完成后，运行效果演示

打印版本号

输出从100开始

- 引用同一工作空间中其他软件包的头文件

在一些情况我们需要引用其他软件包中提供的函数或宏定义，这样可以一定程度上减少我们在两个节点之间需要进行通信的话题个数，下面我们通过举例来进行说明：

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg my_pkg std_msgs roscpp rospy test_pkg
Created file my_pkg/package.xml
Created file my_pkg/CMakeLists.txt
Created folder my_pkg/include/my_pkg
Created folder my_pkg/src
Successfully created files in /home/corvin/ros_workspace/src/my_pkg. Please adjust the values in package.xml.
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  my_pkg  test_pkg
```

创建命令　新包名称　依赖的软件包

```
corvin@workspace:~/ros_workspace/src/my_pkg$ tree
.
├── CMakeLists.txt
├── include
│   └── my_pkg
├── package.xml
└── src
    └── my_pkg.cpp

3 directories, 3 files
```

源码文件名称，可任意命名

- 修改源码文件

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include "test_pkg/test_pkg.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_pkg");
  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("my_chatter", 1000);
  ros::Rate loop_rate(10);
  int count = INIT_COUNT;

  ROS_INFO("test_pkg version:%s,init count:%d", TEST_PKG_VER, INIT_COUNT);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "my_pkg: " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

引用其他软件包的头文件

将其他软件包头文件中的宏定义在本地软件包中打印输出测试。

my_pkg.cpp源码文件内容

ROS小课堂

- 修改两个软件包的编译配置文件

```
## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
add_executable(${PROJECT_NAME}_node src/my_pkg.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the execu
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${P

## Specify libraries to link a library or executable target against
target_link_libraries(${PROJECT_NAME}_node
  ${catkin_LIBRARIES}
)
```

my_pkg软件包的CMakeLists.txt文件需要修改的部分，跟其他一般的软件包修改没有什么区别。

```
#####################################
## catkin specific configuration ##
#####################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
INCLUDE_DIRS include
#  LIBRARIES test_pkg
#  CATKIN_DEPENDS roscpp rospy std_msgs
#  DEPENDS system_lib
)
```

依赖包test_pkg的CMakeLists.txt需要修改的部分，通知其他软件包当前软件包包含有自定义头文件,在include目录下

ROS小课堂

- 运行效果演示

```
corvin@workspace:~/ros_workspace$ rosrun my_pkg my_pkg_node
[ INFO] [1500363875.212986851]: test_pkg version:1.0.0,init count:100
[ INFO] [1500363875.213024080]: my_pkg: 100
[ INFO] [1500363875.213049749]: my_pkg: 101
[ INFO] [1500363875.213073873]: my_pkg: 102
[ INFO] [1500363875.213097226]: my_pkg: 103
[ INFO] [1500363875.613260599]: my_pkg: 104
```

编译完成后，运行效果演示

打印依赖软件包头文件宏定义信息

```
1 #ifndef _TEST_PKG_
2 #define _TEST_PKG_
3
4 #define TEST_PKG_VER   "1.0.1"
5 #define INIT_COUNT     120
6
7 #endif
```

修改后的test_pkg.h

```
corvin@workspace:~/ros_workspace$ rosrun my_pkg my_pkg_node
[ INFO] [1500364320.136588315]: test_pkg version:1.0.1,init count:120
[ INFO] [1500364320.136629259]: my_pkg: 120
[ INFO] [1500364320.236734168]: my_pkg: 121
[ INFO] [1500364320.336770677]: my_pkg: 122
[ INFO] [1500364320.436837112]: my_pkg: 123
```

重新编译后运行

打印修改后的宏定义

2. 如何引用第三方库文件

我们经常在开发软件包时需要引入第三方的so动态库，但是将其放到系统默认库路径中或者使用绝对路径比较简单省事。但是这样的话我们软件包的移植性就会变差，当需要移植到其他机器上时需要重新配置该软件包的依赖库路径。

- 手工自己创建测试用的动态库

我们创建一个简单的动态库，里面只有一个简单的函数来计算两个输入整数的乘积并返回，我们首先创建multiply.cpp和multiply.h两个文件。

```cpp
#include "multiply.h"

int multiply(int a, int b)
{
    return a*b;
}
```

库文件源码只包含一个函数

```cpp
#ifndef _MULTIPLY_H_
#define _MULTIPLY_H_

#ifndef __cpluscplus
extern "C"
{
#endif

    int multiply(int a, int b);

#ifndef __cpluscplus
}
#endif

#endif
```

防止头文件重复包含引用

告诉编译器需要用C语言链接方式来链接这个库，这样就可以g++来编译了

C++编译器的一个宏标志

接下来准备开始将multiply.cpp编译成为动态库libmultiply.so供我们测试用，注意-shared参数和-fPIC参数很重要：

-shared：告诉gcc要生成的是动态链接库；

-fPIC：告诉gcc生成的生成的代码是非位置依赖的，方便用于动态链接。

```
corvin@workspace:~/Public$ g++ -shared -fPIC -o libmultiply.so multiply.cpp
corvin@workspace:~/Public$ ls
libmultiply.so  multiply.cpp  multiply.h
corvin@workspace:~/Public$ file libmultiply.so
libmultiply.so: ELF 64-bit LSB  shared object, x86-64, version 1 (SYSV), dynam
ically linked, BuildID[sha1]=fbb016cd9ef90ffbf7ad4a76c25cd6b1d9ebde00, not str
ipped
```

通过file命令来查看该动态库文件类型

将编译好的动态库复制到当前软件包对应的目录下。

```
corvin@workspace:~/ros_workspace/src$ tree my_pkg/
my_pkg/
├── CMakeLists.txt
├── include
│   └── my_pkg
│       └── multiply.h
├── lib
│   └── libmultiply.so
├── package.xml
└── src
    └── my_pkg.cpp

4 directories, 5 files
```

将动态库对应的头文件放到当前软件包的include目录内

首先创建lib目录，然后将生成的测试动态库放到该目录下。

- 编辑代码和配置文件来引用动态库

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include "test_pkg/test_pkg.h"
#include "my_pkg/multiply.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_pkg");
  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("my_chatter", 1000);
  ros::Rate loop_rate(10);
  int count = multiply(30, 5);

  ROS_INFO("test_pkg version:%s,init count:%d", TEST_PKG_VER, INIT_COUNT);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "my_pkg: " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

引用其他软件包的头文件

引用动态库对应的头文件

直接调用动态库头文件提供的函数

打印其他软件包头文件中的宏定义

修改后的my_pkg.cpp,增加了引用第三方动态库的代码，用于测试。

ROS小课堂

- 修改编译配置文件，编译后运行效果演示

```
117 ## Specify additional locations of header files
118 ## Your package locations should be listed before other locations
119 include_directories(
120   include
121   ${catkin_INCLUDE_DIRS}
122 )
123
124 link_directories(
125   lib
126   ${catkin_LIB_DIRS}
127 )
128
129 ## Declare a C++ library
130 # add_library(${PROJECT_NAME}
131 #   src/${PROJECT_NAME}/my_pkg.cpp
132 # )
154 ## Specify libraries to link a library or executable target against
155  target_link_libraries(${PROJECT_NAME}_node
156    ${catkin_LIBRARIES}
157    multiply
158  )
```

link_directories（）用于添加第三方库路径，catkin_LIB_DIRS 表示创建的环境变量,${catkin_LIB_DIRS} 为取该环境变量的值，lib是相对于当前软件包所在路径的相对路径

引用第三方动态库,在这里需要去掉lib前缀和后面.so

```
corvin@workspace:~/ros_workspace$ rosrun my_pkg my_pkg_node
[ INFO] [1500373473.404964275]: test_pkg version:1.0.1,init count:120
[ INFO] [1500373473.404999900]: my_pkg: 150
[ INFO] [1500373473.505228054]: my_pkg: 151
[ INFO] [1500373473.605195409]: my_pkg: 152
[ INFO] [1500373473.705195100]: my_pkg: 153
```

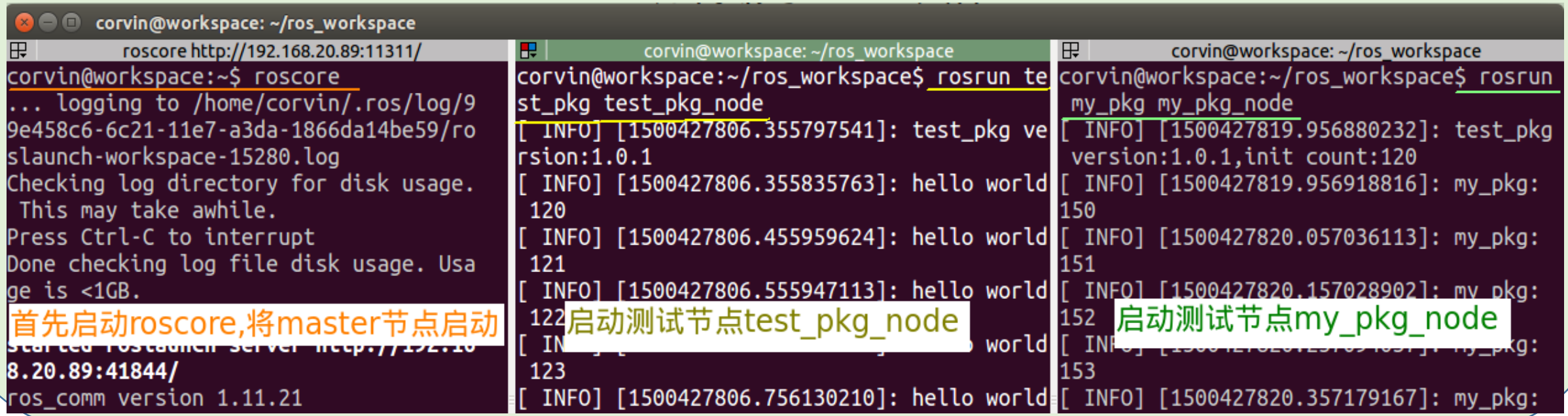
编译完成后运行的效果

调用动态库的输出

3. 如何编写launch文件

在ROS中一个节点程序一般只能完成功能单一的任务，但是一个完整的ROS机器人一般由很多个节点程序同时运行、相互协作才能完成复杂的任务，因此这就要求在启动机器人时就必须要启动很多个节点程序，一般的ROS机器人由十几个节点程序组成，复杂的几十个都有可能。

这就要求我们必须高效率的启动很多节点，而不是通过rosrun命令来依次启动十几个节点程序，launch文件就是为解决这个需求而生，launch文件就是一个xml格式的脚本文件，我们把需要启动的节点都写进launch文件中，这样我们就可以通过roslaunch工具来调用launch文件，执行这个脚本文件就一次性启动所有的节点程序。



接下来我们首先创建存放launch文件的软件包，然后编写一个简单的launch文件来一次性启动这两个节点：

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg robot_bringup
Created file robot_bringup/package.xml
Created file robot_bringup/CMakeLists.txt
Successfully created files in /home/corvin/r
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  my_pkg  robot_bringup  test_pkg
corvin@workspace:~/ros_workspace/src$ cd robot_bringup/
corvin@workspace:~/ros_workspace/src/robot_bringup$ ls
CMakeLists.txt  package.xml
corvin@workspace:~/ros_workspace/src/robot_bringup$ mkdir launch
corvin@workspace:~/ros_workspace/src/robot_bringup$ ls
CMakeLists.txt  launch  package.xml
```

一般每个ROS机器人程序都包含有一个软件包命名为XXX_bringup，里面存放启动所有节点的launch文件。

在bringup软件包中创建launch文件夹，来存放最终的launch文件。

每个XML文件都必须要包含一个根元素，根元素由一对launch标签定义：<launch> … </launch>

<node pkg="package-name" type="executable-name" name="node-name" />

在节点标签末尾的斜杠“/”是必须的，但很容易忘，你也可以这样显式地给出结束标签：

<node pkg="…" type="…" name="…"> </node>

```
1 <launch>
2   <node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" output="screen"/>
3   <node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" output="screen"/>
4 </launch>
5
```

包名　节点的执行文件　节点命名　节点的输出到屏幕上

在launch文件中，启动节点时name属性给节点指派了名称，它将覆盖任何通过调用 ros::int来赋予节点的名称。在默认状态下，从启动文件启动节点的标准输出被重定向到一个日志文件中，而不是在控制台显示。该日志文件的名称是： ~/.ros/log/run_id/node_name-number-stout.log 其中，run_id 是节点管理器（master）启动时生成的一个唯一标示符。

某个单独的节点在控制台中输出信息，只需在节点元素中配置：output="screen"配置了该属性的节点会将标准输出显示在屏幕上而不是记录到日志文档。

接下来我们就可以将整个工作空间编译，这样我们就可以直接通过roslaunch命令来执行launch文件了。

roslaunch的用法是直接在终端中执行：

```
roslaunch package_name xxx.launch
```

下面来查看实际的运行效果：

ROS小课堂

- 编译完成后的运行效果演示

```
corvin@workspace:~/ros_workspace$ roslaunch robot_bringup startup.launch
... logging to /hom
roslaunch-workspace
Checking log direct
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.20.89:39415/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    my_pkg_node (my_pkg/my_pkg_node)
    test_pkg_node (test_pkg/test_pkg_node)

auto-starting new master
process[master]: started with pid [16913]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 8a675384-6c
process[rosout-1]: started wit
started core service [/rosout]
process[test_pkg_node-2]: started with pid [16929]
process[my_pkg_node-3]: started with pid [16935]
[ INFO] [1500432421.749163032]: test_pkg version:1.0.1
[ INFO] [1500432421.749196307]: hello world 120
[ INFO] [1500432421.751033932]: test_pkg version:1.0.1,init count:120
[ INFO] [1500432421.751064021]: my_pkg: 150
[ INFO] [1500432421.849395737]: hello world 121
[ INFO] [1500432421.851188960]: my_pkg: 151
[ INFO] [1500432421.949306711]: hello world 122
[ INFO] [1500432421.951176800]: my_pkg: 152
```

roslaunch来启动launch文件，用法是roslaunch后面加上包名，然后是包里的launch文件名

启动的两个节点名

我们通过执行一个文件就将两个节点启动，这是节点的打印输出信息。

- 介绍常用的节点属性

① 节点重生属性（respawn）

当roslaunch开启所有nodes后，roslaunch会监视每个node，记录那些仍然活动的nodes。对于每个node，当其终止后，我们可以要求roslaunch重启该node，通过使用respawn属性。

```
1 <launch>
2   <node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" respawn="true" output="screen"/>
3   <node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" output="screen"/>
4 </launch>
5
```

设置节点的重生属性，即当节点意外停止时会自动重新启动

```
[ INFO] [1500434318.044015201]: my_pkg: 196
[ WARN] [1500434318.087214853]: Shutdown request received.
[ WARN] [1500434318.087234889]: Reason given for shutdown:
 [user request]
[test_pkg_node-2] process has finished cleanly
log file: /home/corvin/.ros/log/f1ee0d00-6c30-11e7-8325-18
66da14be59/test_pkg_node-2*.log
[test_pkg_node-2] restarting process
process[test_pkg_node-2]: started with pid [17897]
[ INFO] [1500434318.444076503]: my_pkg: 200
[ INFO] [1500434318.517660108]: hello world 122
[ INFO] [1500434318.544264144]: my_pkg: 201
```

节点在正常运行中...

检测到test_pkg_node节点异常退出

roslaunch重新启动test_pkg_node节点

```
corvin@workspace:~$ rosnode list
/my_pkg_node
/rosout
/test_pkg_node
corvin@workspace:~$ rosnode kill /test_pkg_node
killing /test_pkg_node
killed
corvin@workspace:~$ rosnode list
/my_pkg_node
/rosout
/test_pkg_node
```

查看当前运行的节点列表

模拟节点异常退出

发现被杀死的节点依然存在

② 必要节点属性(required)

当一个节点被声明为必要节点即 required="true"终止的时候，roslaunch 会终止所有其他活跃节点并退出。比如在依赖激光雷达的机器人导航中，若激光雷达节点意外退出时，roslaunch将会终止其他节点然后退出。

```
1 <launch>
2   <node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" respawn="true" output="screen"/>
3   <node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" required="true" output="screen"/>
4 </launch>
5
```

设置该节点为必要节点，若该节点意外退出，其他节点都需要退出。

```
[ INFO] [1500435675.117601924]: hello world 226
[ INFO] [1500435675.120445009]: my_pkg: 256
[ INFO] [1500435675.217527043]: hello world 227
[ INFO] [1500435675.220361684]: my_pkg: 257
[ WARN] [1500435675.239087480]: Shutdown request received.
[ WARN] [1500435675.239108220]: Reason given for shutdown: [user request]
[ INFO] [1500435675.317...40]: hello world 228
[ INFO] [1500435675.417...29]: hello world 229
```

节点在正常运行中...

监测到某节点异常退出

```
QUIRED process [my_pkg_node-3] has died!
rocess has finished cleanly
og file: /home/corvin/.ros/log/173e83ac-6c34-11e7-8987-1866da14be59/my_pkg_node-
.log
nitiating shutdown!
```

发现退出的节点是必要节点

```
[ INFO] [1500435675.517767862]: hello world 230
[my_pkg_node-3] killing on exit
[test_pkg_node-2] killing on exit
[ INFO] [1500435675.617831084]: hello world 231
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

依次关闭其他节点...

```
corvin@workspace:~$ rosnode list
/my_pkg_node
/rosout
/test_pkg_node
corvin@workspace:~$ rosnode kill /my_pkg_node
killing /my_pkg_node
killed
corvin@workspace:~$ rosnode list
ERROR: Unable to communicate with master!
```

模拟必要节点异常退出

所有节点全退出

③ 设定节点所需的参数（param）

节点中的一些参数可以直接在launch文件中设定，这样就没有必要修改源码和编译了，每次只要修改一下launch文件就可以直接修改节点运行的参数。<param>标签定义了在参数服务器上设置的参数，放在<node>标签中，在这种情况下，参数被当成了私有参数。

首先我们来修改launch文件中my_pkg节点的启动代码，增加相应的私有参数，如下图所示：

```
1  <launch>
2    <node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" respawn="true" output="screen"/>
3    <node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" required="true" output="screen">
4      <param name="name" type="str" value="corvin" />
5      <param name="age" type="int" value="20" />
6      <param name="handsome" type="bool" value="true" />
7      <param name="salary" type="double" value="1234.56" />
8    </node>
9  </launch>
10
```

给my_pkg_node节点设定参数，
str类型：name
int类型：年龄
bool类型　:是否长的帅
double类型:每月工资

ROS小课堂

修改源码文件，获取当前节点的私有参数

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include "test_pkg/test_pkg.h"
#include "my_pkg/multiply.h"

using namespace std;

int main(int argc, char **argv)
{
  string my_name="";
  int my_age = 0;
  bool my_handsome = false;
  double my_salary = 0.0;

  ros::init(argc, argv, "my_pkg");
  ros::NodeHandle n;
  ros::param::get("~name", my_name);
  ros::param::get("~age", my_age);
  ros::param::get("~handsome", my_handsome);
  ros::param::get("~salary", my_salary);

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("my_chatter", 1000);
  ros::Rate loop_rate(10);
  int count = multiply(30, 5);

  ROS_INFO("Get param,name:%s,age:%d,isHandsom:%d,salary:%f",my_name.c_str(),my_age, my_handsome,my_salary);
  ROS_INFO("test_pkg version:%s,init count:%d", TEST_PKG_VER, INIT_COUNT);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "my_pkg: " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

声明并初始化变量用来存储获得的私有参数

从参数服务器获取当前节点的私有参数信息
注意前面的波浪号～,表示此参数是私有参数

打印输出获得的参数信息

修改后的my_pkg.cpp源码文件

ROS小课堂

当编译完成后，加载运行后的效果

```
corvin@workspace:~/ros_workspace$ roslaunch robot_bringup startup.launch
... logging to /home/corvin/.ros/log/dd32875a-6c4a-11e7-8ea2-1866da14be59/roslaunch-workspace-19625.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.20.89:45392/

SUMMARY
========

PARAMETERS
 * /my_pkg_node/age: 20
 * /my_pkg_node/handsome: True
 * /my_pkg_node/name: corvin
 * /my_pkg_node/salary: 1234.56
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    my_pkg_node (my_pkg/my_pkg_node)
    test_pkg_node (test_pkg/test_pkg_node)

auto-starting new master
process[master]: started with pid [19637]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to dd32875a-6c4a-11e7-8ea2-1866da14be59
process[rosout-1]: started with pid [19650]
started core service [/rosout]
process[test_pkg_node-2]: started with pid [19654]
process[my_pkg_node-3]: started with pid [19666]
[ INFO] [1500445445.557600400]: test_pkg version:1.0.1
[ INFO] [1500445445.557637204]: hello world 120
[ INFO] [1500445445.560742899]: Get param,name:corvin,age:20,isHandsom:1,salary:1234.560000
[ INFO] [1500445445.560773235]: test_pkg version:1.0.1,init count:120
[ INFO] [1500445445.560791466]: my_pkg: 150
[ INFO] [1500445445.657831606]: hello world 121
[ INFO] [1500445445.661045308]: my_pkg: 151
```

启动launch文件

加载my_pkg_node的私有参数

launch文件启动的两个节点

在节点中打印输出获取的参数信息

④ 包含其他launch文件(include)

在当前 launch 文件中调用另一个 launch 文件，方便代码的复用，可以使用包含（include）元素<include file="$(find package-name)/launch-file-name">由于直接输入路径信息很繁琐且容易出错，大多数包含元素都使用查找（find）命令搜索功能包的位置来替代直接输入绝对路径。

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg third_pkg std_msgs roscpp rospy
Created file third_pkg/package.xml
Created file third_pkg/CMakeLists.txt
Created folder third_pkg/include/third_p
Created folder third_pkg/src
Successfully created files in /home/corv                    t the v
alues in package.xml.
corvin@workspace:~/ros_workspace/src$ mkdir third_pkg/launch
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  my_pkg  robot_bringup  test_pkg  third_pkg
corvin@workspace:~/ros_workspace/src$ tree third_pkg/
third_pkg/
├── CMakeLists.txt
├── include
│   └── third_pkg
├── launch
├── package.xml
└── src

4 directories, 2 files
```

创建测试软件包，在该包内创建launch目录，创建launch文件，使bringup包中的launch文件可以包含调用该包内的launch文件。

在软件包内创建launch目录，用来放launch启动文件。

我们在src目录下仍然创建一个简单的测试源码

- 编写测试源码

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "third_pkg");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("third_pkg", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "third pkg: " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

编写third_pkg.cpp源码，仍然是简单的测试代码

接下来编写third_pkg.launch文件：

```
<launch>
  <node pkg="third_pkg" type="third_pkg_node" name="third_pkg" output="screen" />
</launch>
```

- 然后修改配置文件

```
133 ## Declare a C++ executable
134 ## With catkin_make all packages are built within a single CMake context
135 ## The recommended prefix ensures that target names across packages don't collide
136 add_executable(${PROJECT_NAME}_node src/third_pkg.cpp)
137
138 ## Rename C++ executable without prefix
139 ## The above recommended prefix causes long target names, the following renames the
140 ## target back to the shorter version for ease of user use
141 ## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
142 # set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")
143
144 ## Add cmake target dependencies of the executable
145 ## same as for the library above
146 # add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
147
148 ## Specify libraries to link a library or executable target against
149 target_link_libraries(${PROJECT_NAME}_node
150   ${catkin_LIBRARIES}
151 )
```

修改软件包内的CMakeLists.txt编译配置文件

然后修改robot_bringup软件包的launch文件，将third_pkg软件包的launch文件添加到总启动文件中：

```
<launch>
  <node pkg="test_pkg" type="test_pkg_node" name="test_pkg_node" respawn="true" output="screen"/>
  <node pkg="my_pkg" type="my_pkg_node" name="my_pkg_node" required="true" output="screen">
    <param name="name" type="str" value="corvin" />
    <param name="age" type="int" value="20" />
    <param name="handsome" type="bool" value="true" />
    <param name="salary" type="double" value="1234.56" />
  </node>
  <include file="$(find third_pkg)/launch/third_pkg.launch" />
</launch>
```

修改后的startup.launch文件，添加了启动third_pkg的launch文件

最后编译整个工作空间，当编译完成后，启动总launch文件来测试效果：

```
corvin@workspace:~/ros_workspace$ catkin_make
Base path: /home/corvin/ros_workspace
Source space: /home/corvin/ros_workspace/src
Build space: /home/corvin/ros_workspace/build
Devel space: /home/corvin/ros_workspace/devel
Install space: /home/corvin/ros_workspace/install
####
#### Running command: "make cmake_check_build_system" in "/home/corvin/ros_wor
kspace/build"
####
```

在工作空间的根目录下执行catkin_make编译命令

- 编译完成后运行效果演示

```
corvin@workspace:~/ros_workspace$ roslaunch robot_bringup startup.launch
... logging to /home/corvin/.ros/log/9e5f109e-6c58-11e7-9102-1866da14be59/rosl
aunch-workspace-20979.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.20.89:38496/

SUMMARY
========

PARAMETERS
 * /my_pkg_node/age: 20
 * /my_pkg_node/handsome: True
 * /my_pkg_node/name: corvin
 * /my_pkg_node/salary: 1234.56
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    my_pkg_node (my_pkg/my_pkg_node)
    test_pkg_node (test_pkg/test_pkg_node)
    third_pkg (third_pkg/third_pkg_node)
```

加载启动launch文件

该launch文件加载启动三个节点，包含刚才创建的third_pkg_node

- 编写launch时的注意事项

① roslaunch 不提供节点开始的顺序保证。这是特意的：没有办法知道哪个节点完全初始化了，所以启动代码必须在启动顺序上鲁棒性比较强。这个行为体现了ROS哲学：每一个节点与其他的节点都应该尽可能的独立、不相关，节点间耦合性尽可能低。

② 在开始任何一个节点前，roslaunch 将会确定 roscore是否已经在运行，如果没有则自动启动它，因此在使用roslaunch启动节点时不用再提前启动roscore了。

③ 大多数 ROS 节点在启动时连接到master节点管理器上，如果没有在launch中配置该节点respawn属性为true运行中若连接中断，则不会尝试重新连接。因此如果 roscore被终止，当前运行的其他节点将无法建立新的连接，即使稍后重启 roscore 也无济于事。

ROS小课堂

三、理解ROS话题及其发布和订阅机制

1. 理解ROS话题topic
   - ROS节点通过话题通信流程简图
   - 通过示例理解话题及其命令行工具
2. 理解topic的发布和订阅机制
   - 解析发布者publisher节点代码
   - 解析订阅者subscriber节点代码

ROS节点间通过话题通信流程简图

ROS小课堂

发布话题频率

话题消息格式

每周末发布一集ROS教程

订阅ROS小课堂话题
成为订阅者subscriber

芸芸众生
ROS中的众多节点们

corvin_node
发布者publisher

ROS小课堂
微信公众号
ROS中话题ros_topic

王大锤
hammer_node
对ROS感兴趣想要学习ROS

1.发送建立链接请求

2. 反馈确认建立链接

3.请求ros_topic数据

4.发送ros_topic数据

去master节点注册
自己发布和订阅的topic

所有节点去master注册
自己发布和订阅的topic

URI:统一资源定位符，定位node在ROS系统中的位置
格式为: protocol://host:port

腾讯云
腾讯微信服务器
ROS的master节点，
管理所有的节点且负责节点间通信

master节点会根据所有的注册信息进行检索，
*发现corvin_node发布了ros_topic
*发现hammer_node订阅了ros_topic
则master节点会将corvin_node的URI发送给hammer_node,这样hammer_node收到URI后会向corvin_node发送链接请求并请求数据。

WARNING
该简图仅供大家更容易理解ROS的通信流程，对其有个初步认识。不必深究其原理的科学性及准确性。

- 通过示例理解话题及其命令行工具

我们继续接着前面的示例来深入理解一下什么是话题，由于前面的三个软件包都是发布者，我们需要酌情修改一下，然后增加一个订阅者节点，这样就能真实的感受到ROS中节点如何通过话题进行通信了。

```
corvin@workspace:~/ros_workspace/src$ catkin_create_pkg subscribe_pkg std_msgs rospy roscpp
Created file subscribe_pkg/package.xml
Created file subscribe_pkg/CMakeLists.txt
Created folder subscribe_pkg/include/subscribe_pkg
Created folder subscribe_pkg/src
Successfully created files in /home/corvin/ros_workspace/src/subscribe_pkg. Please adjust the values in package.xml.
corvin@workspace:~/ros_workspace/src$ ls
CMakeLists.txt  my_pkg  robot_bringup  subscribe_pkg  test_pkg  third_pkg
```

创建subscribe_pkg软件包

接下来修改一下third_pkg软件包，由于最初的代码编写时发送的话题类型为字符串，这里我们需要将其修改为整型，这样方便我们后面使用rqt_plot画图软件将话题数据可视化演示，在这里我们修改如下：

（1）将默认的话题类型从String修改为Int32;

（2）将话题名称从third_pkg修改为third_pkg_topic，方便我们与节点third_pkg区别开;

（3）将发送话题的频率从10Hz改为2Hz，500ms发送一次;

（4）话题中的数据内容从0到4循环发送，这样方便我们查看动态发布过程。

ROS小课堂

- 修改源码文件

```cpp
#include "ros/ros.h"
#include "std_msgs/Int32.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "third_pkg");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::Int32>("third_pkg_topic", 1000);
  ros::Rate loop_rate(2);
  int count = 0;

  while (ros::ok())
  {
    std_msgs::Int32 msg;
    msg.data = count;
    ROS_INFO("%d", msg.data);
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    count = (++count)%5;
  }
  return 0;
}
```

修改头文件

修改话题的数据类型

修改话题名称

修改发布话题的频率

将数据填充到话题的数据字段中

使数据可以循环

修改后的third_pkg.cpp

ROS小课堂

subscribe.cpp
源码文件

```cpp
#include "ros/ros.h"
#include "std_msgs/Int32.h"

void chatterCallback(const std_msgs::Int32::ConstPtr& msg)
{
  ROS_INFO("I heard: [%d]", msg->data);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "subscribe_node");
  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("third_pkg_topic", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

订阅者的回调函数当话题中有数据时自动调用该函数进行处理。

订阅的话题名称

- 修改的配置文件

```
## Declare a C++ executable
## With catkin_make all packages are built within a sin
## The recommended prefix ensures that target names acr
add_executable(${PROJECT_NAME}_node src/subscribe.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target name
## target back to the shorter version for ease of user
## e.g. "r                                  of "rosrun s
# set_targ                                  de PROPERTIES

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAM

## Specify libraries to link a library or executable ta
target_link_libraries(${PROJECT_NAME}_node
  ${catkin_LIBRARIES}
)
```

修改subscribe_pkg软件包的CMakeLists.txt编译配置文件

修改third_pkg的launch文件，增加启动subscribe_pkg_node：

```
<launch>
  <node pkg="third_pkg" type="third_pkg_node" name="third_pkg" output="screen" />
  <node pkg="subscribe_pkg" type="subscribe_pkg_node" name="subscribe_pkg" output="screen"/>
</launch>
```

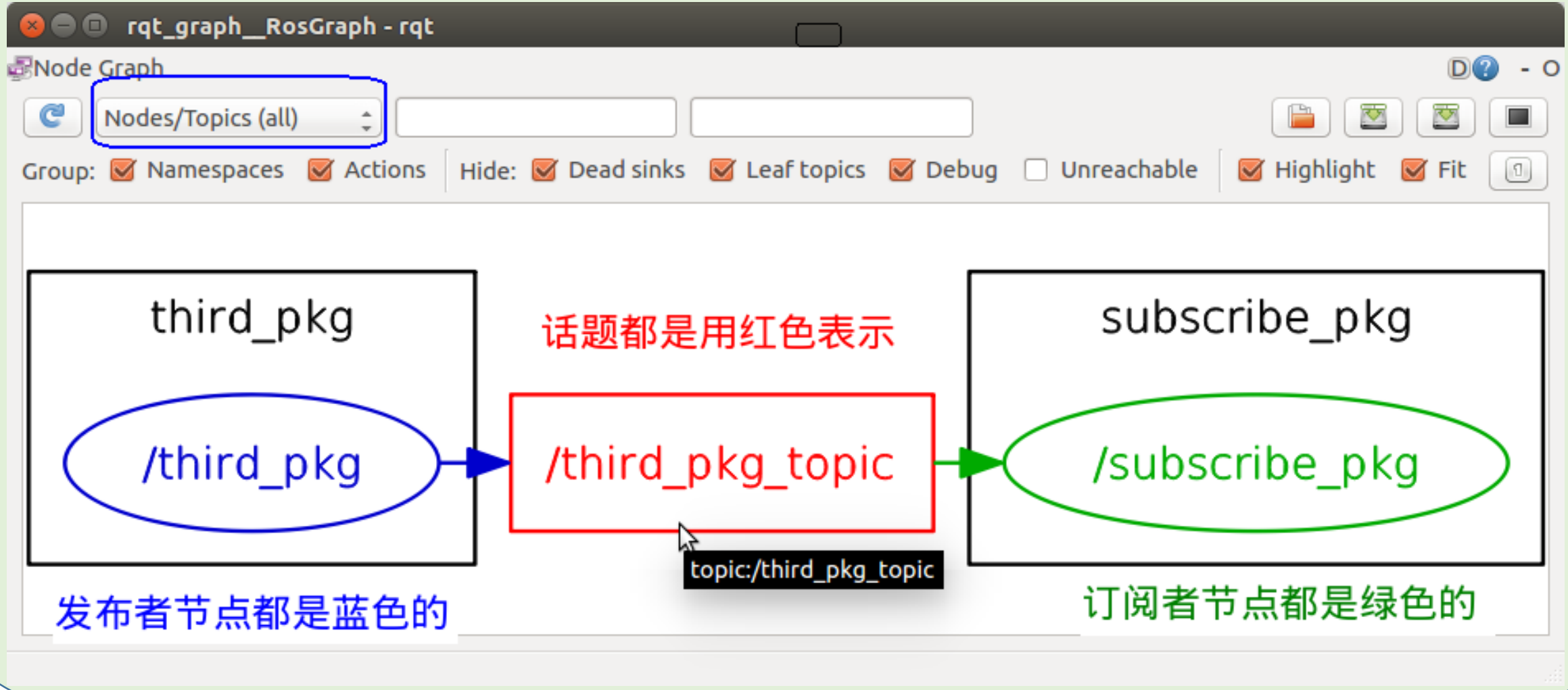
当编译工作空间完成后，通过执行third_pkg.launch文件查看运行后的效果：

roslaunch third_pkg third_pkg.launch

```
process[third_pkg-2]: started with pid [7305]
process[subscribe_pkg-3]: started with pid [7312]
[ INFO] [1500531665.072379745]: 0
[ INFO] [1500531665.572678916]: 1
[ INFO] [1500531665.573329992]: I heard: [1]
[ INFO] [1500531666.072627177]: 2
[ INFO] [1500531666.073108780]: I heard: [2]
[ INFO] [1500531666.572522815]: 3
[ INFO] [1500531666.573012046]: I heard: [3]
[ INFO] [1500531667.072615283]: 4
[ INFO] [1500531667.073100076]: I heard: [4]
                                : 0
                                : I heard: [0]
                                : 1
[ INFO] [1500531668.073177014]: I heard: [1]
```

当节点一发布消息延时1ms后订阅者节点就收到消息。

接下来通过rqt_graph来查看当前ROS系统中运行的节点及其直接话题通信概览图，直接在终端中输入rqt_graph即可启动该工具。



- rostopic命令工具介绍

```
corvin@workspace:~$ rostopic -h
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
        rostopic bw     display bandwidth used by topic
        rostopic delay  display delay of topic from timestamp in header
        rostopic echo   print messages to screen
        rostopic find   find topics by type
        rostopic hz     display publishing rate of topic
        rostopic info   print information about active topic
        rostopic list   list active topics
        rostopic pub    publish data to topic
        rostopic type   print topic type

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'
```

查看rostopic的帮助选项

常用的选项

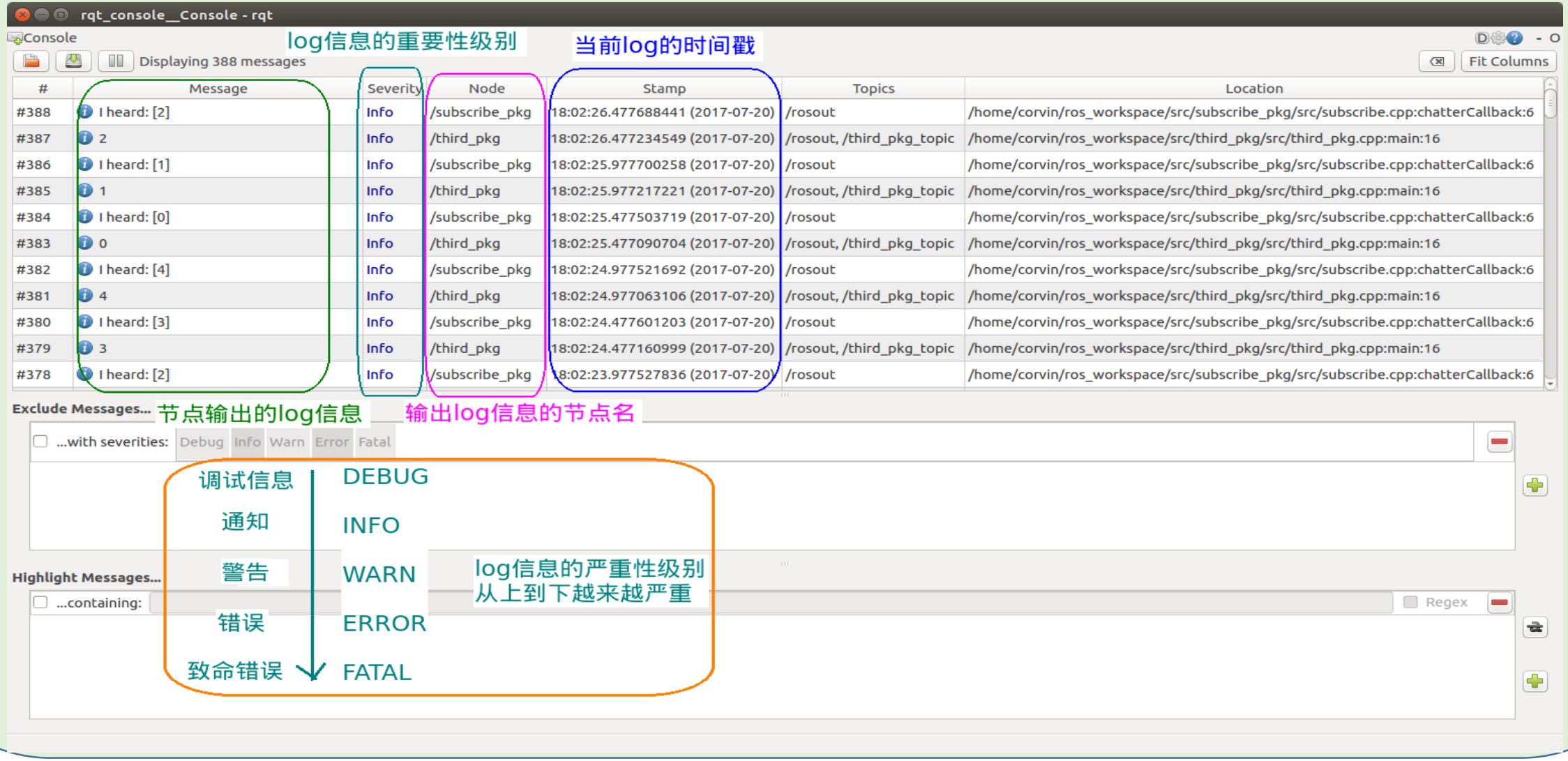
（1）rostopic echo可以显示在某个话题上发布的数据。

```
corvin@workspace:~$ rostopic echo /third_pkg_topic
data: 3
---
data: 4
---
data: 0
---
data: 1
---
data: 2
```

输出/third_pkg_topic话题里的数据内容

ROS小课堂

rostopic echo是在终端命令行中显示节点的log信息，在ROS中还有一个可视化界面rqt_console也可以显示各节点的log信息，直接在终端中输入rqt_console即可启动该软件：



（2）rostopic hz命令可以用来查看话题数据发布的频率。

```
corvin@workspace:~$ rostopic hz /third_pkg_topic
subscribed to [/third_pkg_topic]
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00000s window: 2
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00009s window: 4
a
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00009s window: 8
```

发布的话题频率2Hz,与代码中规定的发布频率相同

（3）rostopic info命令可以查看当前话题的消息类型及订阅和发布者信息

```
corvin@workspace:~$ rostopic info /third_pkg_topic
Type: std_msgs/Int32

Publishers:
 * /third_pkg (http://192.168.20.89:37655/)

Subscribers:
 * /subscribe_pkg (http://192.168.20.89:35004/)
```

话题数据类型

该话题的发布者和订阅者的信息

ROS小课堂

（4）rostopic list能够列出所有当前订阅和发布的话题。

```
corvin@workspace:~$ rostopic list
/rosout
/rosout_agg
/statistics
/third_pkg_topic
corvin@workspace:~$ rostopic list -h
Usage: rostopic list [/namespace]

Options:
  -h, --help            show this help message and exit
  -b BAGFILE, --bag=BAGFILE
                        list topics in .bag file
  -v, --verbose         list full details about each topic
  -p                    list only publishers
  -s                    list only subscribers
  --host                group by host name
corvin@workspace:~$ rostopic list -v

Published topics:
 * /third_pkg_topic [std_msgs/Int32] 1 publisher
 * /rosout [rosgraph_msgs/Log] 3 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher

Subscribed topics:
 * /third_pkg_topic [std_msgs/Int32] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /statistics [rosgraph_msgs/TopicStatistics] 1 subscriber
```

查看list命令的帮助选项

输出更详细信息

（5）rostopic pub可以把数据发布到当前某个正在广播的话题上，该命令用法如下：

rostopic pub [topic] [msg_type] [args]

```
[ INFO] [1500536829.859970654]: I heard: [2]
[ WARN] [1500536830.164190784]: Shutdown request
received.
[ WARN] [1500536830.164212572]: Reason given for
shutdown: [user request]
[third_pkg-2] process has finished cleanly
log file: /home/corvin/.ros/log/9d6ae5e2-6d1f-11e
7-9b6c-1866da14be59/third_pkg-2*.log
[ INFO] [1500536835.530276100]: I heard: [11111]
[ INFO] [1500536853.669758880]: I heard: [2222]
[ INFO] [1500536860.990784016]: I heard: [333]
```

将发布节点关闭，我们通过rostopic pub命令来手动发送消息

```
corvin@workspace:~$ rosnode kill /third_pkg
killing /third_pkg
killed
corvin@workspace:~$ rostopic pub -1 /third_pkg_topic std_msgs/Int32 11111
publishing and latching message for 3.0 seconds
corvin@workspace:~$ rostopic pub -1 /third_pkg_topic std_msgs/Int32 2222
publishing and latching message for 3.0 seconds
corvin@workspace:~$ rostopic pub -1 /third_pkg_topic std_msgs/Int32 333
publishing and latching message for 3.0 seconds
```

只发布一次　话题名　消息类型　消息内容

如果要想周期性的往话题中发布消息可以添加-r选项：

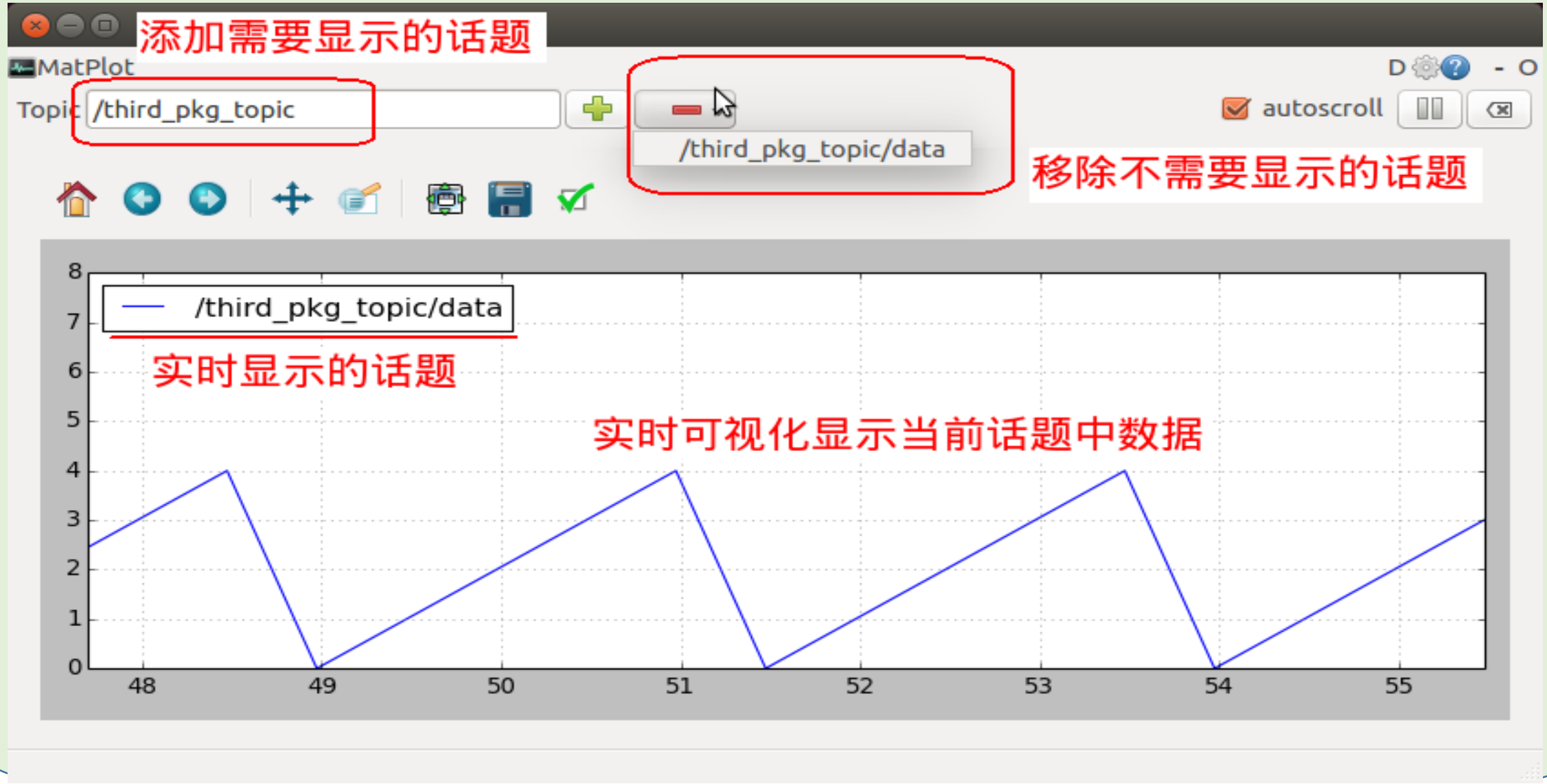
```
[ INFO] [1500537641.223341147]: I heard: [123]
[ INFO] [1500537641.723340507]: I heard: [123]
[ INFO] [1500537642.223309117]: I heard: [123]
[ INFO] [1500537642.723312987]: I heard: [123]
[ INFO] [1500537643.223424594]: I heard: [123]
```

间隔500ms

```
corvin@workspace:~$ rostopic pub -r 2 /third_pkg_topic std_msgs/Int32 123
^Ccorvin@workspace:~$
corvin@workspace:~$
corvin@workspace:~$
corvin@workspace:~$
```

往话题中以固定频率发布消息

- 使用rqt_plot工具使话题数据实时显示可视化。



2. 理解topic的发布和订阅机制

- 解析发布者publisher节点代码

ros/ros.h是一个实用的头文件，它引用了ROS系统中大部分常用的头文件。

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "third_pkg");
  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<std_msgs::Int32>("third_pkg_topic", 1000);
  ros::Rate loop_rate(2);
  int count = 0;

  while (ros::ok())
  {
    std_msgs::Int32 msg;
    msg.data = count;
    chatter_pub.publish(msg);
    ROS_INFO("%d", msg.data);

    ros::spinOnce();
    loop_rate.sleep();
    count = (++count)%5;
  }
  return 0;
}
```

包含std_msgs/Int32消息，它是系统提供的标准消息类型

初始化ROS

为当前节点创建一个句柄，即实例化当前节点的一个对象

指定自循环的频率

向master注册本节点将要发布int类型的话题，话题名称为third_pkg_topic,消息队列大小为1000个

判断ROS 节点是否仍处于运行良好的状态

使循环休眠一段时间，使其频率符合2Hz

- 解析订阅者subscriber节点代码

```
1  #include "ros/ros.h"
2  #include "std_msgs/Int32.h"
3
4  void chatterCallback(const std_msgs::Int32::ConstPtr& msg)
5  {
6    ROS_INFO("I heard: [%d]", msg->data);
7  }
8
9  int main(int argc, char **argv)
10 {
11   ros::init(argc, argv, "subscribe_node");
12   ros::NodeHandle n;
13
14   ros::Subscriber sub = n.subscribe("third_pkg_topic", 1000, chatterCallback);
15   ros::spin();
16   return 0;
17 }
```

回调函数，当订阅的话题中有收到消息时会自动的调用，消息是以指针的形式传输，这样就省去数据的复制过程又可以获取其中内容，这样加快了消息获取的速度又节省存储空间。

消息队列大小，防止我们处理话题中速度不够快，起到一个临时缓存的作用

通知master要订阅话题

话题名

要调用的处理函数指针

进入自循环，可以尽可能快的调用消息回调函数。如果没有消息到达，它不会占用很多CPU

ROS小课堂

四、理解ROS的自定义消息、服务和actionlib

1. 理解ROS的自定义消息
   - 认识msg的基本概念
   - 通过示例来理解如何创建和使用msg
2. 理解ROS的服务
   - 认识服务的基本概念
   - 通过示例来理解如何创建和使用srv
3. 理解ROS的actionlib软件包
   - 认识actionlib存在的意义
   - 通过示例来理解如何使用actionlib

1. 理解ROS的自定义消息msg

- 了解msg的基本概念

msg文件：是一个描述ROS中所使用消息类型的简单文本，它们会被用来生成不同语言的源代码。msg文件存放在软件包的msg目录下，msg文件实际上就是每行声明一个数据类型和变量名。可以使用的数据类型如下：

- int8, int16, int32, int64 (plus uint*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array[] and fixed-length array[C]

下面介绍rosmsg命令行工具：

```
corvin@workspace:~$ rosmsg -h
rosmsg is a command-line tool for displaying information about ROS Message types.

Commands:
        rosmsg show     Show message description
        rosmsg list     List all messages
        rosmsg md5      Display message md5sum
        rosmsg package  List messages in a package
        rosmsg packages List packages that contain messages

Type rosmsg <command> -h for more detailed usage
```

rosmsg的命令选项

下面来介绍几个常用的rosmsg命令：

① rosmsg list：列出当前ros系统中所有的消息格式，并统计共有多少个。

```
corvin@workspace:~$ rosmsg list | wc -l
668
corvin@workspace
```

统计共有多少个消息格式

```
corvin@workspace:~$ rosmsg list
actionlib/TestAction
actionlib/TestActionFe
actionlib/TestActionGoal
actionlib/TestActionResult
actionlib/TestFeedback
actionlib/TestGoal
actionlib/TestRequestAction
actionlib/TestRequestActionFeedback
actionlib/TestRequestActionGoal
actionlib/TestRequestActionResult
```

打印消息列表

非常多

② rosmsg show：显示出消息的定义格式。

linear 和 angular 都是复合域，其数据类型是 geometry_msgs/Vector3。

一个复合域是由简单的一个或多个子域组合而成，其中的每一个子域可能是另一个复合域或者独立的域，而且它们也都由基本数据类型组成。

```
corvin@workspace:~$ rosmsg show std_msgs/Int32
int32 data

corvin@workspace:~$ rosmsg show std_msgs/String
string data

corvin@workspace:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

ROS中使用的速度信息，包括线速度和角速度

```
corvin@workspace:~$ rosmsg show -h
Usage: rosmsg show [options] <message type>

Options:
  -h, --help            show this help message and exit
  -r, --raw             show raw message text, including comments
  -b BAGFILE, --bag=BAGFILE
                        show message from .bag file
corvin@workspace:~$ rosmsg show -r geometry_msgs/Twist
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

show命令的二级命令选项列表

- 通过示例来理解如何创建和使用msg

现在我们来创建一个自定义的消息来演示整个过程如何操作：

① 首先来创建目录来存放我们将要创建的msg文件

```
corvin@workspace:~/ros_workspace/src/third_pkg$ tree
.
├── CMakeLists.txt
├── include
│   └── third_pkg
├── launch
│   └── third_pkg.launch
├── msg
│   └── myTestMsg.msg
├── package.xml
└── src
    └── third_pkg.cpp

5 directories, 5 files
```

创建msg目录并创建myTestMsg.msg文件用于自定义消息

myTestMsg.msg内容

```
1 string name
2 int32 age
3 bool handsome
4 float32 salary
```

ROS小课堂

② 接下来要确保msg文件被转换成为C++，Python和其他语言的源代码，要修改package.xml和CMakeLists.txt：

```
31  <!-- The *_depend tags are used to specify dependencies -->
32  <!-- Dependencies can be catkin packages or system dependencies -->
33  <!-- Examples: -->
34  <!-- Use build_depend for packages you need at compile time: -->
35  <build_depend>message_generation</build_depend>
36  <!-- Use buildtool_depend for build tool packages: -->
37  <!--   <buildtool_depend>catkin</buildtool_depend> -->
38  <!-- Use run_depend for packages you need at r
39  <run_depend>message_runtime</run_depend>
40  <!-- Use test_depend for packages you need only for testing: -->
41  <!--   <test_depend>gtest</test_depend> -->
42  <buildtool_depend>catkin</buildtool_depend>
43  <build_depend>roscpp</build_depend>
44  <build_depend>rospy</build_depend>
45  <build_depend>std_msgs</build_depend>
46  <run_depend>roscpp</run_depend>
47  <run_depend>rospy</run_depend>
48  <run_depend>std_msgs</run_depend>
```

需要增加这两条

修改后的package.xml

```
 7  ## Find catkin macros and libraries
 8  ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
 9  ## is used, also find other catkin packages
10  find_package(catkin REQUIRED COMPONENTS
11    roscpp
12    rospy
13    std_msgs
14    message_generation
15  )
```

CMakeLists.txt文件

增加的选项

在 CMakeLists.txt文件中，利用find_packag函数，增加对message_generation的依赖，这样就可以生成消息了。 你可以直接在COMPONENTS的列表里增加message_generation

```
50  ## Generate messages in the 'msg' folder
51  add_message_files(
52    FILES
53    myTestMsg.msg
54  #   Message2.msg
55  )
```

CMakeLists.txt
增加的选项

③ 然后就可以先将刚才添加的消息编译，然后查看ROS是否已经可以识别我们添加的自定义消息，编译过程跟编译软件包相同，仍然是catkin_make编译整个工作空间，当编译成功后开始利用学到的rosmsg list查看有没有我们的消息。

（编译完成后记得执行source devel/setup.bash，使环境变量生效）

```
corvin@workspace:~/ros_workspace$ rosmsg list | grep myTestMsg
third_pkg/myTestMsg
corvin@workspace:~/ros_workspace$ rosmsg show third_pkg/myTestMsg
string name
int32 age
bool handsome
float32 salary
```

ros系统已经可以识别我们自定义的消息格式，且跟我们刚才在msg文件中定义的相同。

④ 当系统可以识别我们自定义的消息后，就可以开始修改测试代码了，往话题中发布我们自定义格式的消息：

```cpp
#include "ros/ros.h"
#include "third_pkg/myTestMsg.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "third_pkg");
  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<third_pkg::myTestMsg>("third_pkg_topic", 1000);
  ros::Rate loop_rate(2);
  while (ros::ok())
  {
    third_pkg::myTestMsg msg;
    msg.name = "corvin";
    msg.age = 20;
    msg.handsome = true;
    msg.salary = 123.45;
    chatter_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

添加自定义消息的头文件

更新发布话题的消息格式为自定义的消息格式

声明一个自定义消息的对象

往将要发布的消息中填充数据

将消息发送到话题中

修改后的third_pkg.cpp源码

⑤ 接下来修改订阅者的代码，修改接收话题的数据格式：

```
1  #include "ros/ros.h"
2  #include "third_pkg/myTestMsg.h"
3
4  void chatterCallback(const third_pkg::myTestMsg::ConstPtr& msg)
5  {
6    ROS_INFO("I heard-name:%s,age:%d,ishandsome:%d,salary:%f", msg->name.c_str(),
7             msg->age,msg->handsome,msg->salary);
8  }
9
10 int main(int argc, char **argv)
11 {
12   ros::init(argc, argv, "subscribe_node");
13   ros::NodeHandle n;
14
15   ros::Subscriber sub = n.subscribe("third_pkg_topic", 1000, chatterCallback);
16   ros::spin();
17   return 0;
18 }
```

添加自定义消息头文件

在打印字符串时需要将C++格式转换为C格式，所以要加后缀

修改后的subscribe.cpp源码

⑥最后我们就可以编译代码，然后查看测试效果

```
corvin@workspace:~$ roslaunch third_pkg third_pkg.launch
... logging to /home/corvin/.ros/log/47feee5c-6ddb-11e7-a56b-1866da14be59/roslaunch-workspa
ce-9056.log
Checking log directory for disk usage.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.20.89:40728/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    subscribe_pkg (subscribe_pkg/subscribe_pkg_node)
    third_pkg (third_pkg/third_pkg_node)

auto-starting new master
process[master]: started with pid [9068]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 47feee5c-6ddb-11e7-a56b-1866da14be59
process[rosout-1]: started with pid [9081]
started core service [/rosout]
process[third_pkg-2]: started wit
process[subscribe_pkg-3]: started with pid [9094]
[ INFO] [1500617423.956294205]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500617424.456038175]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500617424.955963605]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500617425.456047670]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500617425.955963775]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500617426.455875345]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
```

通过launch文件启动发布和订阅节点

订阅者节点打印的信息就是我们自定义格式的消息

2. 理解ROS的服务

- 认识服务的基本概念

消息传递机制尽管是 ROS 系统中节点通信的主要方法，但确实受到了一定的限制，因此引入另一种通信的方法，称之为服务调用（ service calls），服务调用与消息的区别主要体现在两个方面：

*服务调用是双向的，一个节点给另一个节点发送信息并等待响应，因此信息流是双向的。作为对比，当消息发布后，并没有响应的概念，甚至不能保证系统内有节点订阅了这些消息。

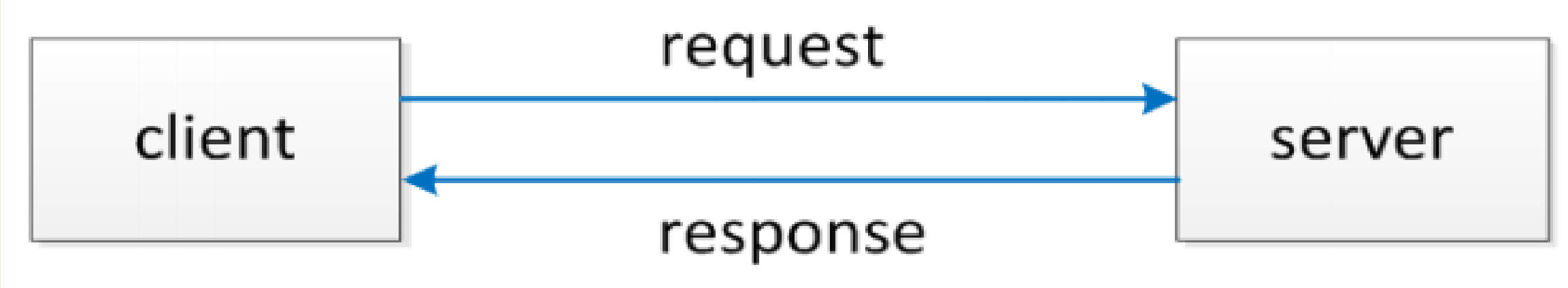
*服务调用实现的是一对一通信。每一个服务由一个节点发起，对这个服务的响应返回同一个节点。对于消息传递来说，每一个消息都和一个话题相关，这个话题可能有很多的发布者和订阅者。

简单来理解消息传递和服务调用区别的话就是：

*通过话题来传递消息的机制就雷同于通过发送微信公众号来广播消息，不管有没有人订阅ROS小课堂我都会往公众号里发布教程，你愿意学习ROS你就订阅查看，没人订阅不影响我发公众号。

*服务调用就雷同于你给我打电话询问问题，等我接通电话后询问你的问题后，然后我经过思考后再把答案告诉你，我们之间的通信是一对一的，而且是及时的。不同于发话题，如果没有发公众号你想看教程都没有。

服务调用的基本信息流如下图所示：



其过程是一个客户端（client）节点发送一些称为请求（request）的数据到一个服务器（server）节点， 并且等待回应。服务器节点接收到请求后，采取一些行动（计算、配置软件或硬件、改变自身行为等），然后发送一些称为响应（response） 的数据给客户端节点。

请求和响应数据携带的特定内容由服务数据类型（service data type）来决定，它与决定消息内容的消息类型是类似的。唯一的区别就在于服务数据类型分为两部分，分别表示请求（客户端节点提供给服务器节点）和响应（服务其节点反馈给客户端节点）。

- 通过示例来理解如何创建和使用srv

我们创建一个可以查询机器人当前电量的服务来进行演示。

① 我们首先来创建存放srv的目录：

一个srv文件描述一项服务。它包含两个部分：请求和响应。

```
corvin@workspace:~/ros_workspace/src/third_pkg$ tree
.
├── CMakeLists.txt
├── include
│   └── third_pkg
├── launch
│   └── third_pkg.launch
├── msg
│   └── myTestMsg.msg
├── package.xml
├── src
│   └── third_pkg.cpp
└── srv
    └── myTestSrv.srv

6 directories, 6 files
```

查看最新的目录结构

创建srv目录，然后创建服务文件myTestSrv.srv

myTestSrv.srv文件内容如下

```
int32 index
---
int32 result
```

srv文件分为请求和响应两部分，由'---'分隔。
上面的int32 index是服务的请求;
下面的int32 result是服务的响应反馈;

ROS小课堂

② 修改配置文件，使其可以正常编译

```
57 ## Generate services in the 'srv' folder
58 add_service_files(
59    FILES
60    myTestSrv.srv
61 #   Service2.srv
62 )
63
64 ## Generate actions in the 'action' folder
65 # add_action_files(
66 #   FILES
67 #   Action1.action
68 #   Action2.action
69 # )
70
71 ## Generate added messages and services with any dependencies listed here
72 generate_messages(
73    DEPENDENCIES
74    std_msgs
75 )
```

添加编译服务的配置选项

third_pkg软件包中修改后的CMakeLists.txt

编译过程跟编译软件包相同，仍然是catkin_make编译整个工作空间，当编译成功后开始利用rossrv list查看有没有我们自定义的服务。

下面来简单介绍rossrv这个命令，并查看我们生成的自定义服务：

```
corvin@workspace:~$ rossrv -h
rossrv is a command-line tool for displaying information about ROS Service types.

Commands:
        rossrv show     Show service description
        rossrv list     List all services
        rossrv md5      Display service md5sum
        rossrv package  List services in a package
        rossrv packages List packages that contain services

Type rossrv <command> -h for more detailed usage
```

列出所有rossrv的命令

```
corvin@workspace:~$ rossrv list | wc -l
241
corvin@workspace:~$ rossrv list | grep myTestSrv
third_pkg/myTestSrv
corvin@workspace:~$ rossrv show third_pkg/myTestSrv
int32 index
---
int32 result
```

```
corvin@workspace:~$ rossrv show -h
Usage: rossrv show [options] <service type>

Options:
  -h, --help            show this help message and exit
  -r, --raw             show raw message text, including comments
  -b BAGFILE, --bag=BAGFILE
                        show message from .bag file
corvin@workspace:~$ rossrv package third_pkg
third_pkg/myTestSrv
```

show命令的二级命令选项

③接下来修改发布者的代码，增加服务端代码

```cpp
#include "ros/ros.h"
#include "third_pkg/myTestMsg.h"
#include "third_pkg/myTestSrv.h"

static int battery = 100;

bool checkBattery(third_pkg::myTestSrv::Request &req,
                  third_pkg::myTestSrv::Response &res)
{
    if(1 == req.index)
    {
        ROS_WARN("Service sending response:[%d]",battery);
        res.result = battery;
    }
    return battery;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "third_pkg");
  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<third_pkg::myTestMsg>("third_pkg_topic", 1000);
  ros::ServiceServer service = n.advertiseService("batteryStatus", checkBattery);
  ros::Rate loop_rate(2);
  while (ros::ok())
  {
    battery--;
    if(battery<10) battery=100;
    third_pkg::myTestMsg msg;
    msg.name = "corvin";
    msg.age = 20;
    msg.handsome = true;
    msg.salary = 123.45;
    chatter_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

增加服务的头文件,由编译系统自动根据我们先前创建的srv文件生成的对应该srv文件的头文件.

服务器端判断请求是否为1,只有为1才返回当前电池的电量状态。index和result是我们在srv文件中定义的请求和反馈字段。

向master节点注册,当前节点将发布服务

电量递减,当少于10重新更新为100

修改后的third_pkg.cpp源码

④ 修改订阅者代码，增加服务的客户端代码

修改后的subscribe.cpp源码

```cpp
#include "ros/ros.h"
#include "third_pkg/myTestMsg.h"
#include "third_pkg/myTestSrv.h"

void chatterCallback(const third_pkg::myTestMsg::ConstPtr& msg)
{
  ROS_INFO("I heard-name:%s,age:%d,ishandsome:%d,salary:%f", msg->name.c_str(),
           msg->age,msg->handsome,msg->salary);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "subscribe_node");
  ros::NodeHandle n;
  ros::Rate loop_rate(1);
  ros::Subscriber sub = n.subscribe("third_pkg_topic", 1000, chatterCallback);

  ros::ServiceClient client = n.serviceClient<third_pkg::myTestSrv>("batteryStatus");
  third_pkg::myTestSrv mySrv;
  mySrv.request.index = 1;

  while(ros::ok())
  {
    if(client.call(mySrv))
    {
      ROS_WARN("Client Get Battery:%d",mySrv.response.result);
    }
    else
    {
      ROS_ERROR("Failed to call batteryStatus service");
      return 1;
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

实例化一个由ROS编译系统自动生成的service类,并给其request成员赋值。一个service类包含两个成员request和response.

发送的服务请求：1，获取电量

获取输出结果

service的调用是阻塞(调用的时候占用进程阻止其他代码的执行),一旦调用完成,将返回调用结果.如果service调用成功,call()函数将返回true,srv.response里面的值将是合法的值。如果调用失败，call()函数将返回false,srv.response里面的值将是非法的。

ROS小课堂

⑤最后编译整个工作空间，然后运行查看效果

```
corvin@workspace:~/ros_workspace$ roslaunch third_pkg third_pkg.launch
... logging to /home/corvin/.ros/log/be39de22-6df6-11e7-ae24-1866da14be59/roslaunch-workspace-14
138.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.20.89:44722/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES
  /
    subscribe_pkg (subscribe_pkg/subscribe_pkg_node)
    third_pkg (third_pkg/third_pkg_node)

auto-starting new master
process[master]: started with pid [14150]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to be39de22-6df6-11e7-ae2
process[rosout-1]: started with pid [141
started core service [/rosout]
process[third_pkg-2]: started with pid [1
process[subscribe_pkg-3]: started with pid [14175]
[ WARN] [1500629218.698387433]: Service sending response:[98]
[ WARN] [1500629218.698888545]: Client Get Battery:98
[ INFO] [1500629218.699098827]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ WARN] [1500629219.698210947]: Service sending response:[96]
[ WARN] [1500629219.698512126]: Client Get Battery:96
[ INFO] [1500629219.698623378]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500629219.698687490]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ WARN] [1500629220.698309627]: Service sending response:[94]
[ WARN] [1500629220.698698897]: Client Get Battery:94
[ INFO] [1500629220.698801998]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
[ INFO] [1500629220.698865589]: I heard-name:corvin,age:20,ishandsome:1,salary:123.449997
```

启动launch进行测试

由于发布者节点的频率是2Hz,1秒钟内将电量值减2,但是订阅者的运行频率是1,每一秒才向服务器请求一次电量值，因此打印值每次少2。

ROS小课堂

⑥使用命令行工具来查看当前系统中服务

```
corvin@workspace:~$ rosservice -h
Commands:
        rosservice args print service arguments
        rosservice call call the service with the provided args
        rosservice find find services by service type
        rosservice info print information about service
        rosservice list list active services
        rosservice type print service type
        rosservice uri  print service ROSRPC uri

Type rosservice <command> -h for more detailed usage, e.g. 'rosservice call -h'

corvin@workspace:~$ rosservice list
/batteryStatus
/rosout/get_loggers
/rosout/set_logger_level
/subscribe_pkg/get_loggers
/subscribe_pkg/set_logger_level
/third_pkg/get_loggers
/third_pkg/set_logger_level
corvin@workspace:~$ rosservice info /batteryStatus
Node: /third_pkg
URI: rosrpc://192.168.20.89:39332
Type: third_pkg/myTestSrv
Args: index
corvin@workspace:~$ rosservice call /batteryStatus 1
result: 90
corvin@workspace:~$ rosservice call /batteryStatus 1
result: 84
```

当服务在运行时可以通过该命令查看当前系统中所有的service

查看自定义服务的详细信息

可以通过命令行来向服务发起请求

3. 理解ROS的actionlib软件包

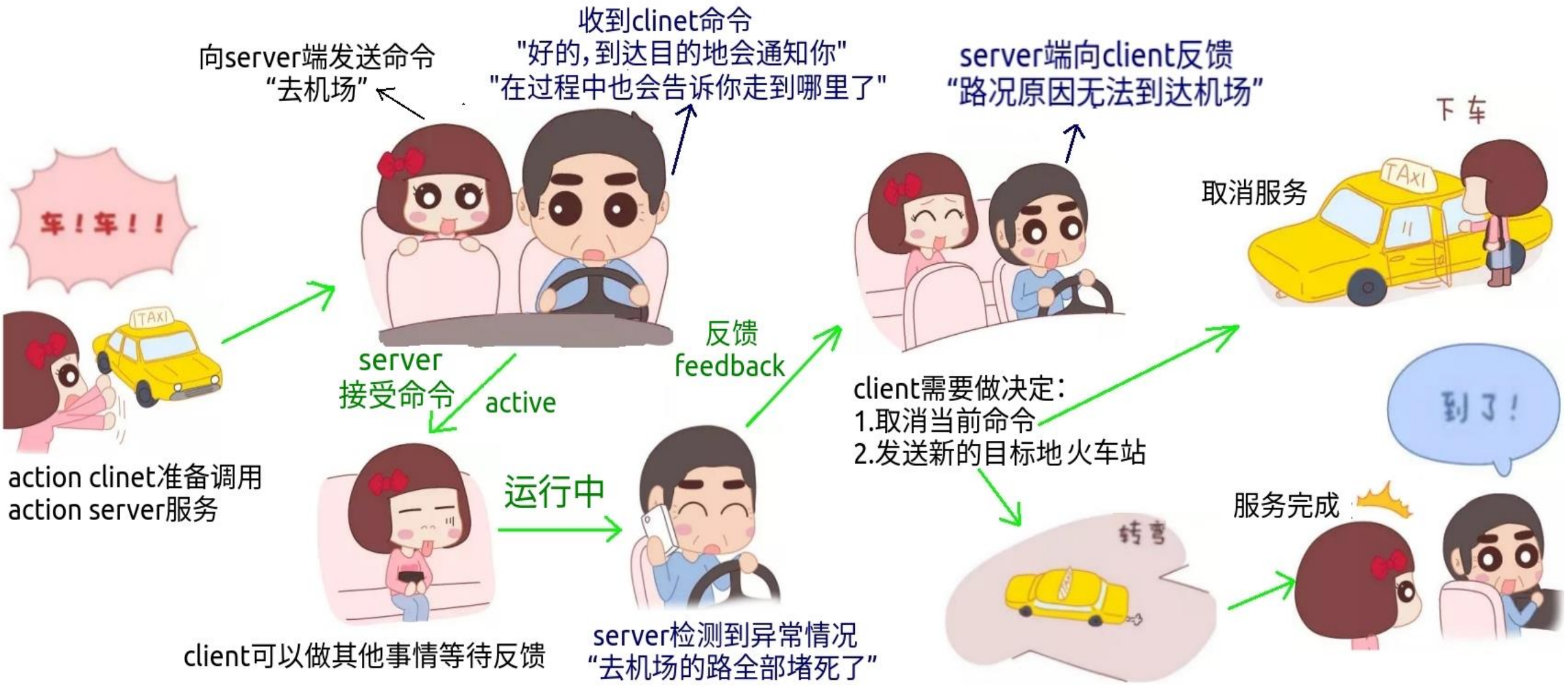
- 认识actionlib存在的意义

在一个复杂的ROS系统中我们会需要各种各有的服务要求，向某个节点发送请求执行某一个任务，并返回相应的执行结果，这种用ROS的服务（services）完成。然而有一些情况服务执行的时间很长，在执行中想要获得任务处理的进度，或可能取消执行任务，actionlib就能实现这样的功能，它是ROS的一个非常重要的库，算是对service在某些情况下能力不足的一种补充。

在前面介绍过服务，因为它在执行的过程中是阻塞的，会阻止程序进一步执行，必须等待服务器返回结果才会继续执行后面的程序，对于一些计算量较轻的任务使用服务是可以满足的。但是对于一些伏在的任务，例如让机器人从房间的某一点A运动到指定的地点B，在此运动过程中需要花费的时间未定，任务时间较长而且中途会遇到各种情况，例如机器人电量不足时，这就需要取消或暂停当前任务等待机器人充满电再继续该任务。

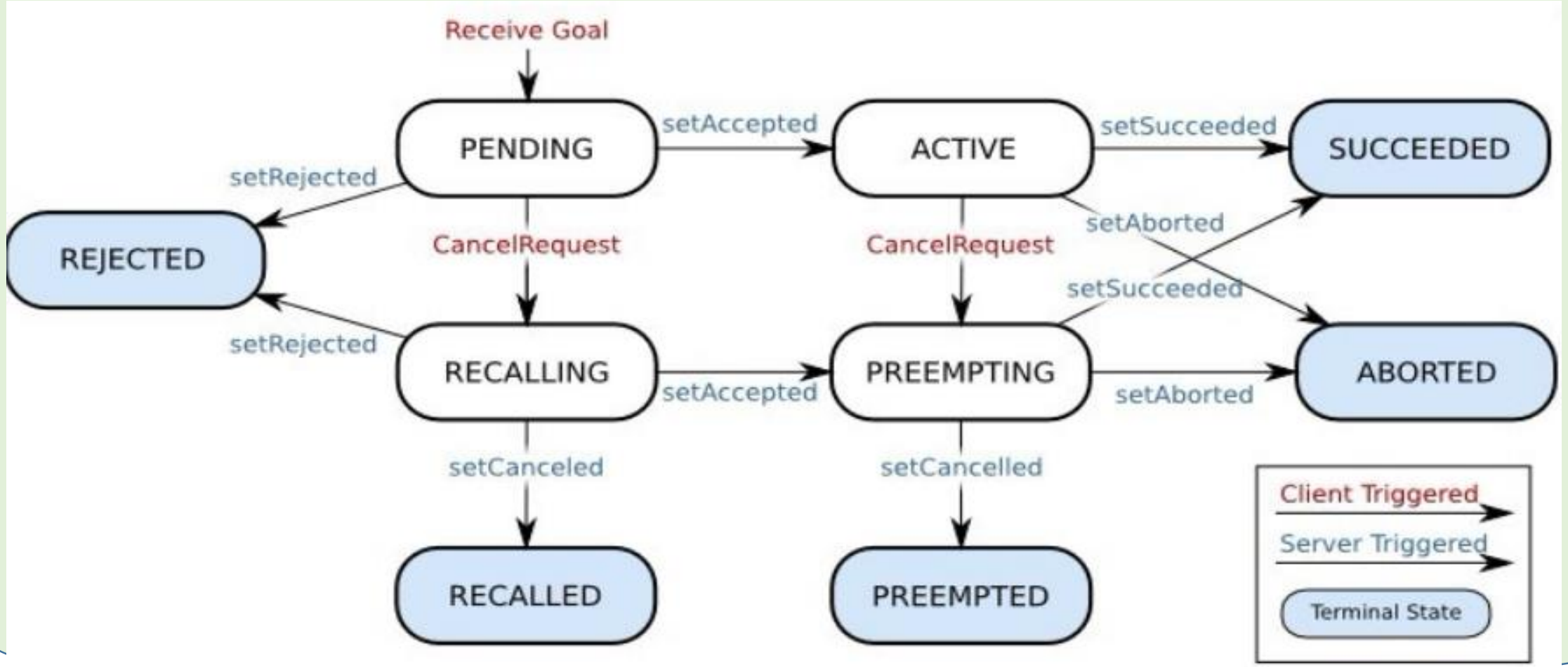
ROS小课堂

action client和server交互流程简图



- 服务端描述

goal是在ActionClient端启动的，一旦ActionServer接收到goal请求，它就会为这个goal创建一个状态机来追踪goal的状态转换：



这些状态的转换大多是服务器端触发的：

setAccepted - 检查到有goal之后，决定开始处理它

setRejected - 检察到goal后，决定不去处理它，因为它是个无效请求（溢出，资源不可用，无效等）

setSucceeded - 告知goal被正确执行

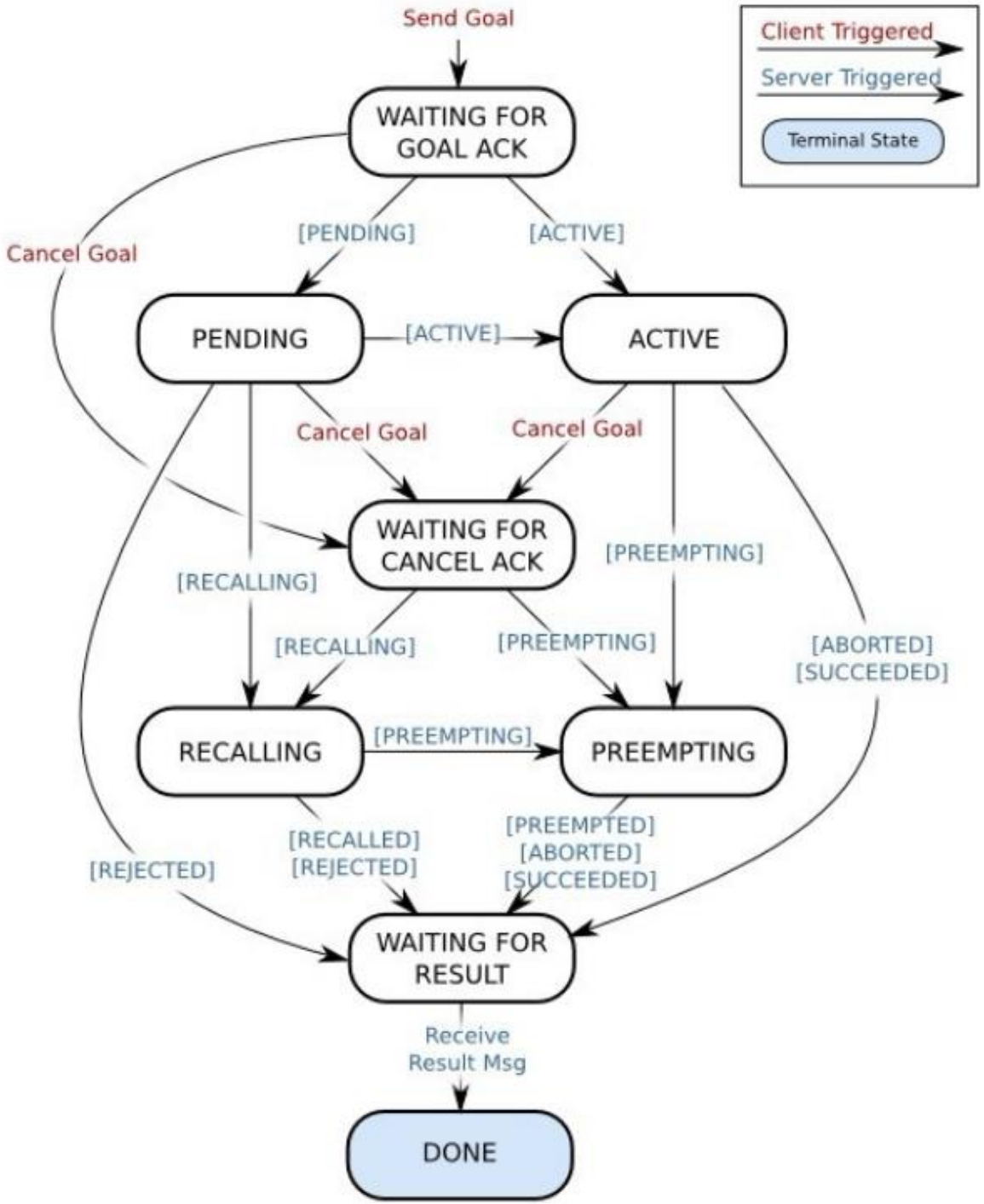
setAborted - 告知goal在处理时遇到了问题不得不被终止了

setCanceled - 告知因cancle请求，goal不再被执行了

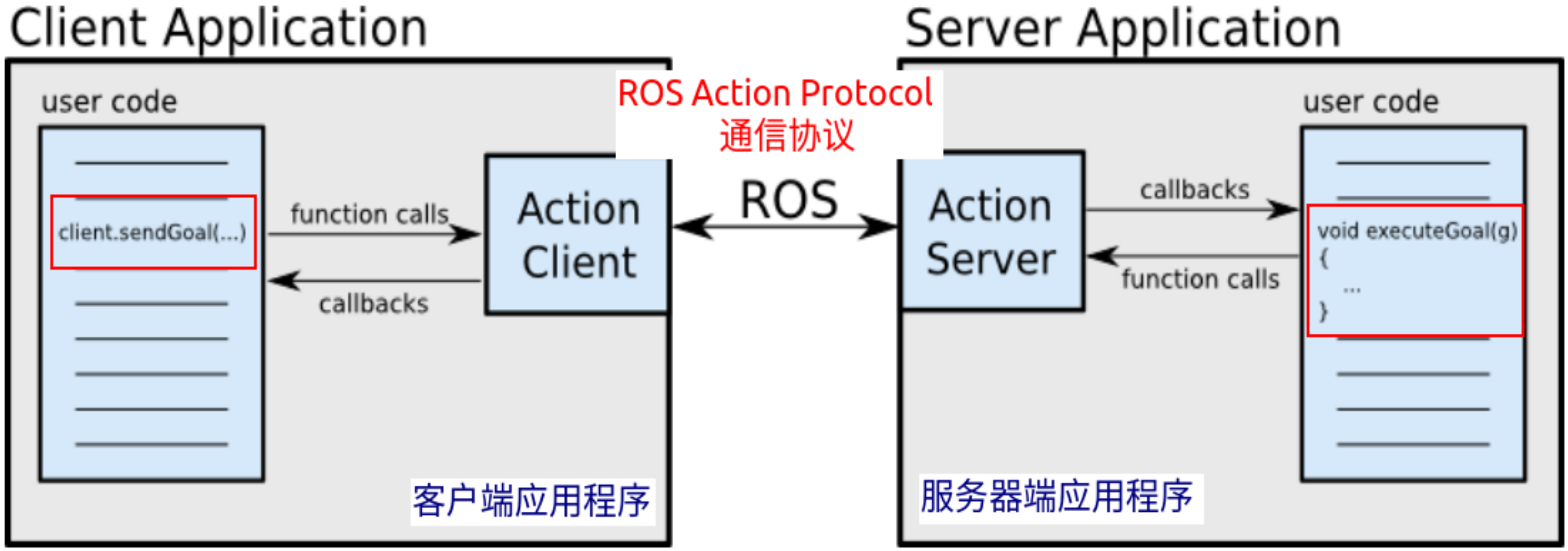
action client也能异步触发状态转换：

CancelRequest: 客户端通知server它要server停止处理这个goal服务端状态

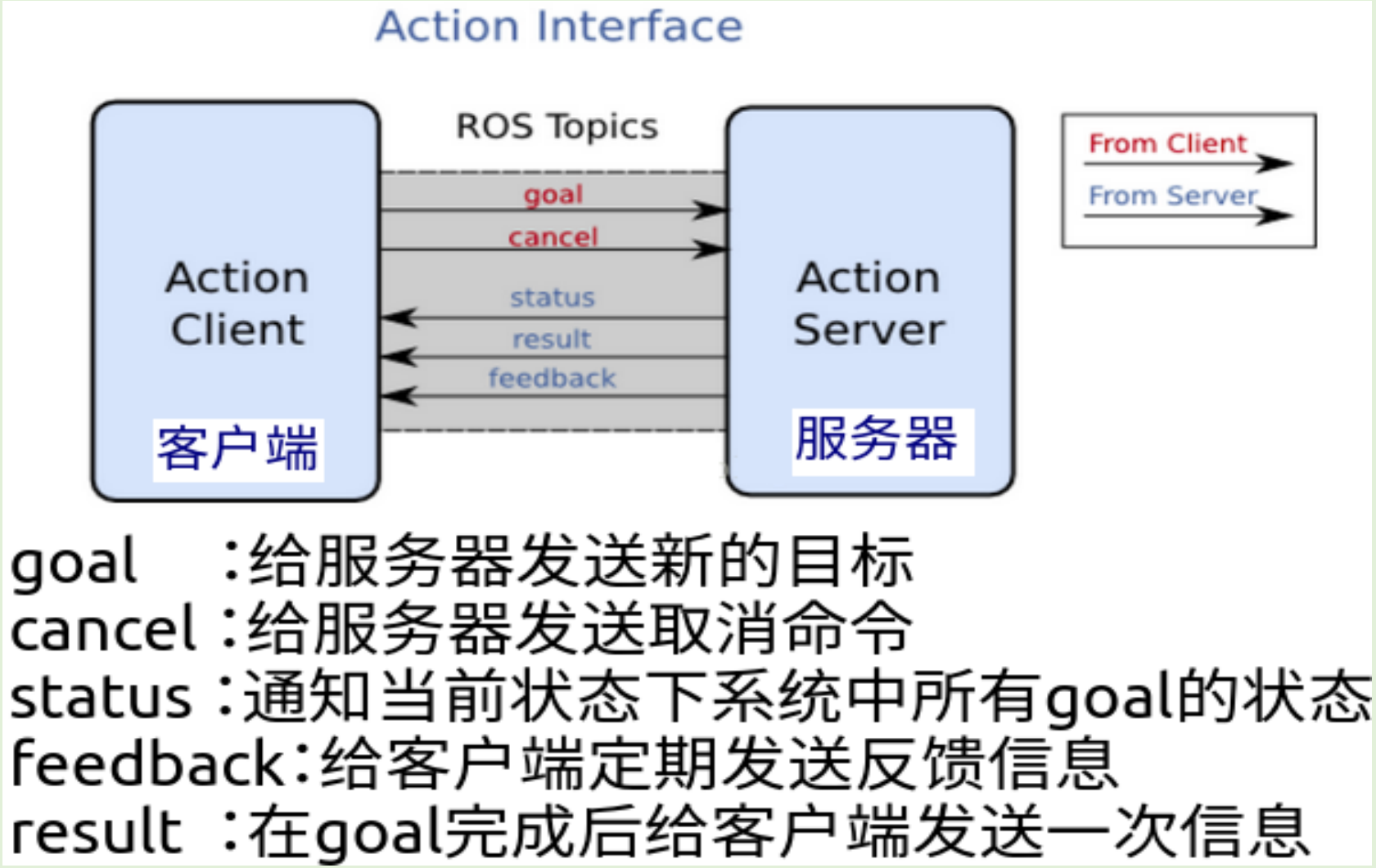
- 客户端描述


Send Goal
Client Triggered
Server Triggered
Terminal State
WAITING FOR GOAL ACK
[PENDING]
[ACTIVE]
Cancel Goal
PENDING
[ACTIVE]
ACTIVE
Cancel Goal
Cancel Goal
WAITING FOR CANCEL ACK
[RECALLING]
[PREEMPTING]
[RECALLING]
[PREEMPTING]
[ABORTED]
[SUCCEEDED]
RECALLING
[PREEMPTING]
PREEMPTING
[RECALLED]
[REJECTED]
[PREEMPTED]
[ABORTED]
[SUCCEEDED]
[REJECTED]
WAITING FOR RESULT
Receive Result Msg
DONE

Actionlib的框架实际是一种特殊的客户-服务的模式。除了服务请求的功能外，还可以实时获取服务器执行任务的进度状态，以及强制中断服务的功能。



ActionClient 和ActionServer之间使用action protocol通信，action protocol就是预定义的一组ROS message，这些message被放到ROS topic上在 ActionClient 和ActionServer之间进行传实现二者的沟通



goal ：给服务器发送新的目标
cancel：给服务器发送取消命令
status：通知当前状态下系统中所有goal的状态
feedback:给客户端定期发送反馈信息
result ：在goal完成后给客户端发送一次信息

- 通过示例来理解如何使用actionlib

未完待续...

ROS小课堂

关注“ROS小课堂”微信公众号，会不断更新ROS教程