ROS程序基础

朱笑笑

2017年12月

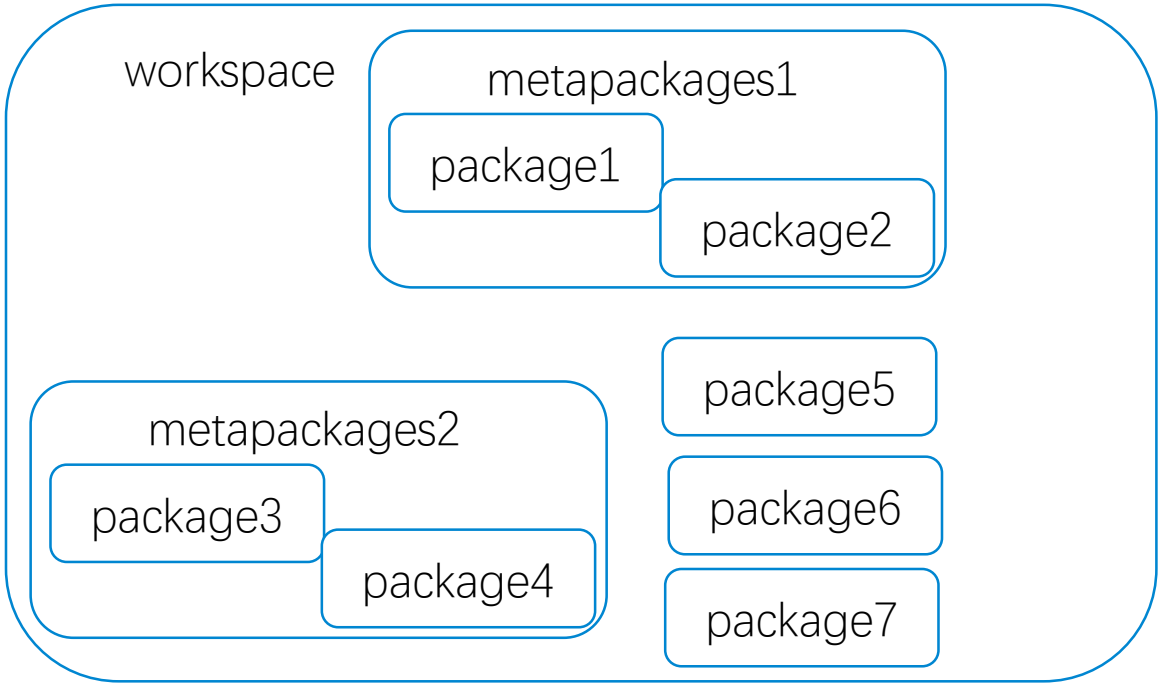
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

目录 Contents

ROS程序创建

- 如前所述，ROS程序的文件结构包括工作空间（workspace）、堆（stack）、包（package）。
- 最终的程序在ROS包内部，但是整个文件系统也非常重要。



ROS程序创建

- 首先需要创建，工作空间（workspace）。
  - $ mkdir -p ~/catkin_ws/src　　//创建一个文件夹
  - $ cd ~/catkin_ws/
  - $ catkin_make　　//利用catkin_make来生成工作空间文件
- 与ROS安装时类似，要让ROS认可这个工作空间，我们需要对环境变量进行配置。对应环境变量为ROS_PACKAGE_PATH
  - $ echo $ROS_PACKAGE_PATH 配置前 /home/youruser/catkin_ws/src:/opt/ros/kinetic/share
  - 配置$ source devel/setup.bash
  - 配置后/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
  - 长期配置： $ echo "source /home/youruser/catkin_ws/devel/setup.bash" >> ~/.bashrc

ROS程序创建

- 创建综合包（metapackages）。
  - 综合包相当于将有相同含义或者接近功能的包进行一个归类整合，让系统程序结构更加清晰。
  - 综合包并不是必须的，并且在形式上只是一个文件夹。
  - 创建综合包用mkdir（或在新建文件夹）命令即可。

ROS程序创建

- 创建包（package）。
  - 一个程序包要想称为catkin程序包必须符合以下要求：
    - 该程序包必须包含package.xml文件
      - 这个package.xml文件提供有关程序包的元信息。
    - 程序包必须包含一个catkin 版本的CMakeLists.txt文件，而Catkin metapackages中必须包含一个对CMakeList.txt文件的引用。
    - 每个目录下只能有一个程序包。
      - 这意味着在同一个目录下不能有嵌套的或者多个程序包存在。
    - 最简单的程序包也许看起来就像这样（不包含程序）：

```
my_package/
  CMakeLists.txt
  package.xml
```

ROS程序创建

- 创建包（package）。
  - catkin_create_pkg用来创建包：
  - # catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
  - $ cd ~/catkin_ws/src
  - $ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
- 创建好的包及工作空间结构大致如下

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

ROS程序创建

- 创建包（package）。
  - 程序包的依赖项　　可以用rospack命令来查看
  - $ rospack depends1 beginner_tutorials
  - 依赖项保持在package.xml
  - 可以在需要的时候进行修改

```
std_msgs
rospy
roscpp
```

```
<package>
...
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
...
</package>
```

目录 Contents

ROS程序编译

- rosbuild和catkin是两个ROS使用过的两个编译系统
  - 将**白名单**的文件集作为安装目标安装
  - 兼容FHS**文件布局**
  - **自动生成**符合cmake的配置文件
  - 外源构建（也用于交叉编译）
  - 仅使用一个命令，就可以以正确的顺序构建多个项目
  - 编译时**允许使用标准cmake命令**，而不是包装器的命令
  - 在开发周期**无需安装**
  - 加速编译周期，使用**并行编译**，对c ++文件跨相互依赖的项目编译
  - 拆分**配置**和**构建**阶段，以提高构建速度
  - 通过避免对资源进行多次类似rospack的查找，来**提高构建速度**（使用单个cmake查找）

ROS程序编译

- 程序包的编译使用catkin_make命令实现。
- 如上创建工作空间是，
  - $ cd ~/catkin_ws/
  - $ catkin_make
- 编译前需要修改确认：
  - package.xml。包的依赖项确认，涉及到头文件，链接库等。
  - CMakeList.txt。编译指令、源文件，编译结果的修改。

package.xml

ROS程序编译

- package.xml修改。
  - 首先更新**描述**标签：

```
<description>The beginner_tutorials package</description>
```

  - 接下来是**维护者**标签：

```
<!-- One maintainer tag required, multiple allowed, one person per tag -->
<!-- Example:  -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
<maintainer email="user@todo.todo">user</maintainer>
```

  - 再接下来是**许可**标签：

```
<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
<license>TODO</license>
```

ROS程序编译

- package.xml修改。
  - 依赖项标签：

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

  - 添加依赖项至run_depend

```
<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

ROS程序编译

- CMakeList.txt修改。
- 涉及到以下的命令
  - 需要Cmake的版本(cmake_minimum_required)
  - 包的名字(project())
  - 查找其它本包会调用的包(find_package())
  - 打开Python模块支持 (catkin_python_setup())
  - 配置Message/Service/Action 文件(add_message_files(), add_service_files(), add_action_files())
  - 调用message/service/action的生成指令 (generate_messages())
  - 指定本包的编译输出(catkin_package())
  - 编译输出项设定 Libraries/Executables(add_library()/add_executable()/target_link_libraries())
  - 添加编译测试(catkin_add_gtest())
  - 编译后结果的安装(install())

ROS程序编译

- CMakeList.txt修改。
- Message/Service/Action相关配置
  - ROS中的消息（.msg），服务（.srv）和动作（.action）文件在ROS包构建和使用之前需要特殊的预处理器构建步骤。 这些宏的目的是生成编程语言特定的文件，以便可以使用他们所选择的编程语言中的消息，服务和动作。 构建系统将使用所有可用的生成器（例如gencpp，genpy，genlisp等）生成绑定。
  - 提供了三个宏来分别处理消息，服务和操作：
    - add_message_files
    - add_service_files
    - add_action_files

ROS程序编译

- CMakeList.txt修改。
- Message/Service/Action相关配置

```
# Get the information about this package's buildtime dependencies
find_package(catkin REQUIRED
  COMPONENTS message_generation std_msgs sensor_msgs)

# Declare the message files to be built
add_message_files(FILES
  MyMessage1.msg
  MyMessage2.msg
)

# Declare the service files to be built
add_service_files(FILES
  MyService.srv
)

# Actually generate the language-specific message and service files
generate_messages(DEPENDENCIES std_msgs sensor_msgs)
```

ROS程序编译

- CMakeList.txt修改。
- 项目的输出
  - 可执行文件

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

  - Lib文件

```
add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
```

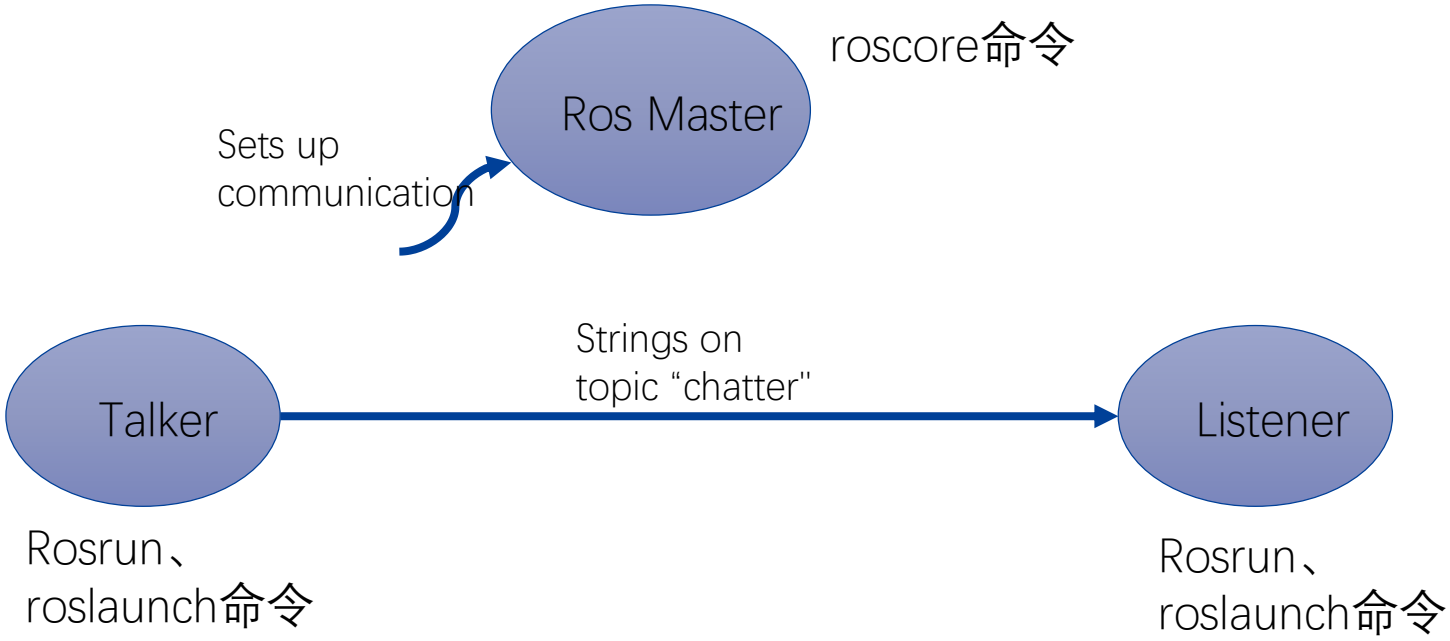
- 项目链接的文件

```
target_link_libraries(<executableTargetName>, <lib1>, <lib2>, ... <libN>)
```

目录 Contents

ROS程序执行

- 在上一课中已经介绍了ROS的图结构，一个系统运行的基本要素。



ROS程序执行

- roscore命令，可以打开ros master服务器，这是各个node相互通信的基础。
- 同时还会运行一个rosout的node，用于收集和记录节点调 试输出信息。
- 可以使用rosnode命令，查看系统中运行的node。
- 在/rosout后，运行rosnode list可得到以下输出

ROS程序执行

- Rosrun可以实现在任何地方执行一个包的node而无需给出全局的路径。
- 用法：rosrun <package> <executable>
- 例如：rosrun roscpp_tutorials talker
- 传递参数：rosrun package node _parameter:=value
- 例如：rosrun my_package my_node _my_param:=value

ROS程序执行

- Roslaunch可以利用launch文件来配置需要启动的节点，可以是多个节点，因此会大大方便系统的启动。
- 用法：$ roslaunch package_name file.launch 或者
- $ roslaunch path/file.launch
- 可以使用多种参数 roslaunch - [args]
- -p port， --wait， --local， --screen， -v， --dump-params

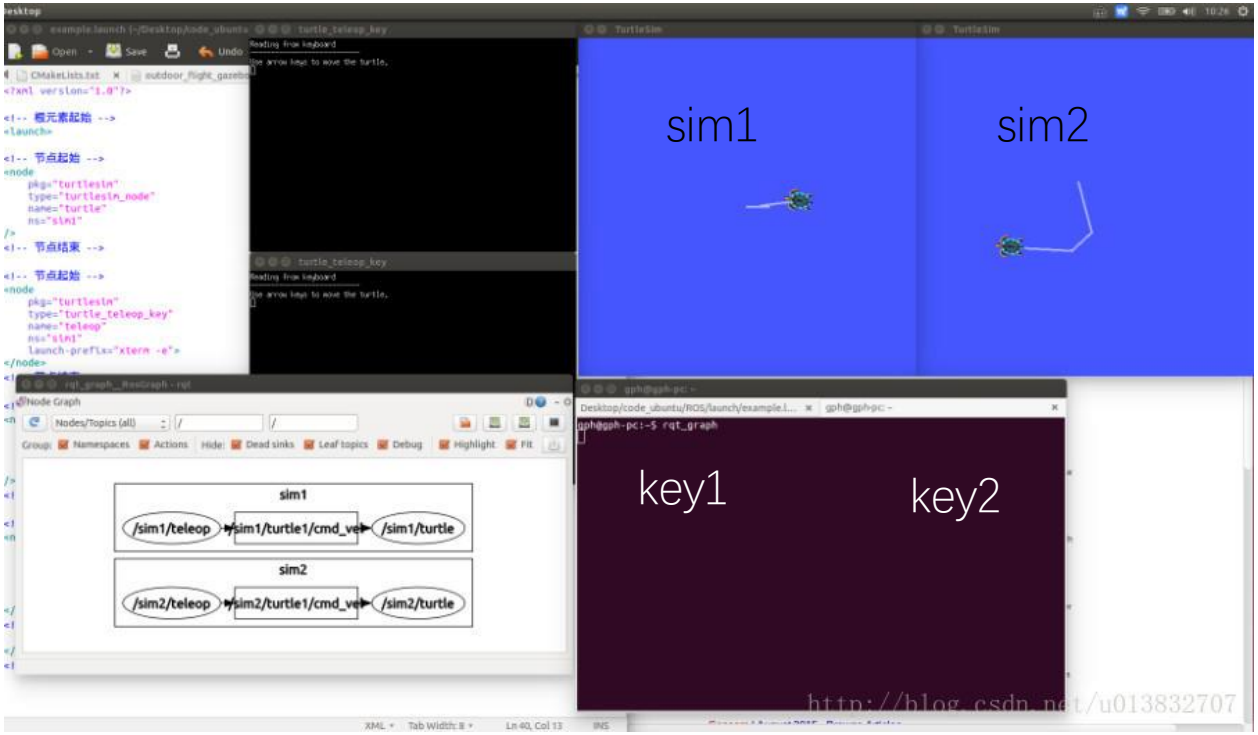
ROS程序执行

- Launch文件格式
  - 一个简单例子

```
<?xml version="1.0"?>
<launch>  <!-- 根元素起始 -->
<!-- 节点起始 -->
    <node pkg="turtlesim" type="turtlesim_node" name="turtle"
ns="sim1" /> <!-- 节点结束 -->
<!-- 节点起始 -->
    <node pkg="turtlesim" type="turtle_teleop_key" name="teleop"
ns="sim1" launch-prefix="xterm -e"> </node> <!-- 节点结束 -->
<!-- 节点起始 -->
    <node pkg="turtlesim" type="turtlesim_node" name="turtle"
ns="sim2" /> <!-- 节点结束 -->
<!-- 节点起始 -->
    <node pkg="turtlesim" type="turtle_teleop_key" name="teleop"
launch-prefix="xterm -e" ns="sim2"> </node> <!-- 节点结束 -->
</launch> <!-- 根元素结束 -->
```

ROS程序执行

- Launch文件格式
  - 利用roslaunch 这个文件可以得到以下效果



ROS程序执行

- Launch文件格式

- 每个节点元素由三个必须的属性：
  - pkg 该节点属于哪个包，相当于rosrun命令后面的第一个参数
  - type 可执行文件的名字，rosrun命令的第二个参数
  - name 该节点的名字，相当于代码中ros::int的命名信息，有了它代码中的名称会被覆盖。

ROS程序执行

- Launch文件格式
  - 其他属性：
  - output 将标准输出显示在屏幕上而不是记录在日志中
  - respawn 请求复位，当该属性的值为respawn="true"时，roslaunch会在该节点崩溃时重新启动该节点
  - required 必要节点，当该值为required="true"时，roslaunch会在该节点终止时终止其他活跃节点。
  - 启动前缀 在启动命令加上前缀。例如当其设置为launch-prefix="xterm -e"时，效果类似于xterm -e rosrun X X。也就是为该节点保留独立的终端。
  - ns 在命名空间中启动节点。
  - 重映射 使用方法remap from="original-name(turtle/pose)"to"new-name(tim)"
  - 包含其他文件include file="path to launch file" 在启动文件中包含其他启动文件的内容（包括所有的节点和参数），可使用如下命令使路径更为简单include file="($find package-name)/launch-file-name"
  - 启动参数（launch arguments） 为了使启动文件便于配置,roslaunch还支持启动参数,有时也简称为参数甚至args,其功能有点像可执行程序中的局部变量。
  - 声明参数：arg name="arg-name" 然而这样的声明并不是必须的(除非你想要给它赋值或设置为默认值,见后续内容),但是这是一个好的做法,因为这样能使读者比较清楚启动文件需要哪些参数
  - 参数赋值： roslaunch package-name launch-file-name arg-name:=arg-value
    - <arg name=”arg-name” default=”arg-value”/>
    - <arg name=”arg-name” value=”arg-value”/>
  - 获取参数：一旦参数值被声明并且被赋值,你就可以利用下面的arg 替换(arg substitution)语法来使用该参数值了:$(arg arg-name)每个该替换出现的地方,roslaunch 都将它替换成参数值。在示例中,我们在 group 元素中的 if 属性使用了一次 use_sim3 参数。

目录 Contents

一个简单的ROS例子

talker.cpp (intro_to_ros)

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

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");     //init node
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000); //init a topic
  ros::Rate loop_rate(1);              //A loop in 1HZ
  int count = 0;

  while (ros::ok()) {
    std_msgs::String msg;              //define a message
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();               //init a message
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);          //Pulish a message into ros network
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
    }
  return 0;
}
```

一个简单的ROS例子

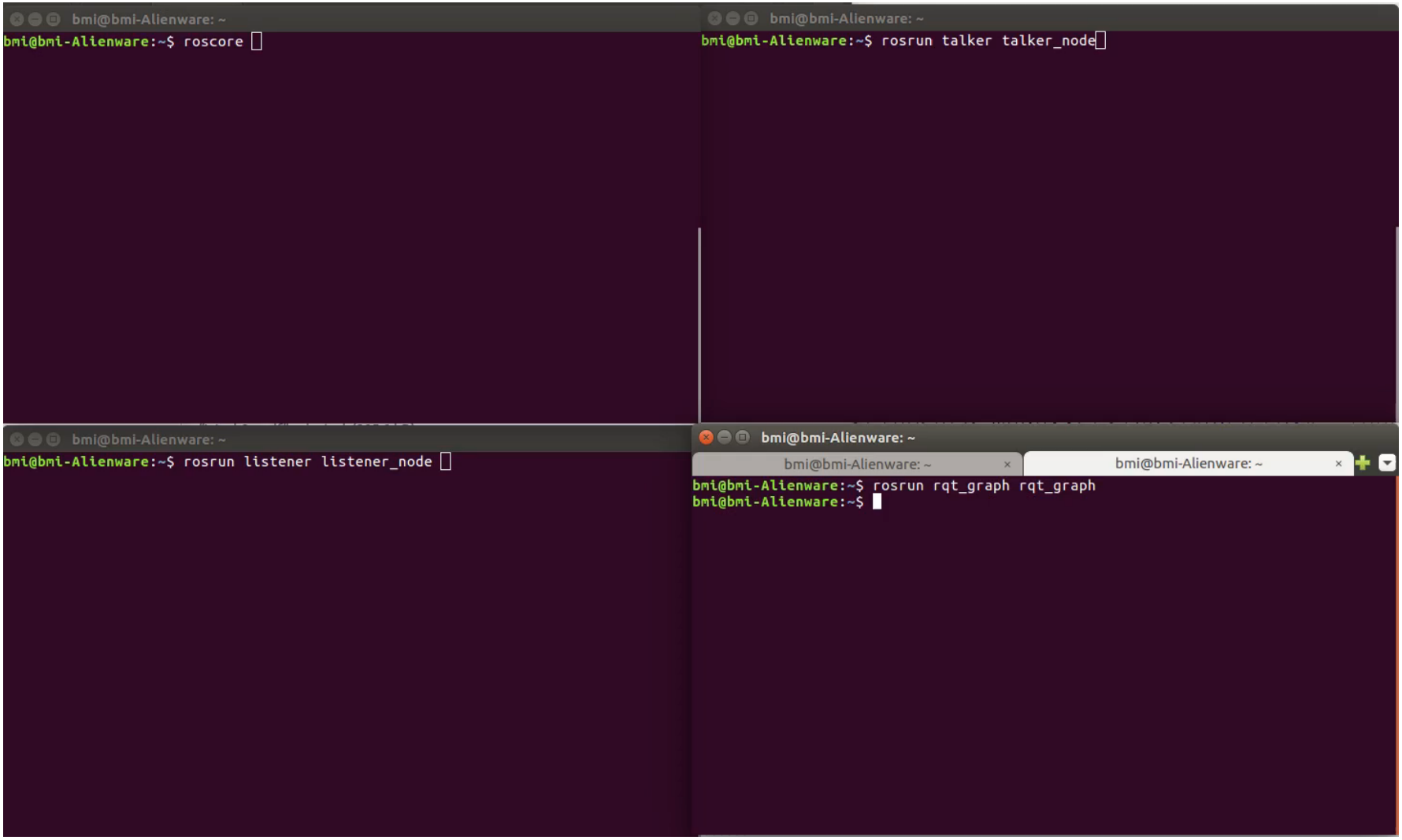
listener.cpp (intro_to_ros)

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

void chatterCallback(const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

//callback function of a topic

//define a subscribe of a topic

一个简单的ROS例子


bmi@bmi-Alienware: ~
bmi@bmi-Alienware:~$ roscore
bmi@bmi-Alienware: ~
bmi@bmi-Alienware:~$ rosrun talker talker_node
bmi@bmi-Alienware: ~
bmi@bmi-Alienware:~$ rosrun listener listener_node
bmi@bmi-Alienware: ~
bmi@bmi-Alienware: ~
bmi@bmi-Alienware: ~
bmi@bmi-Alienware:~$ rosrun rqt_graph rqt_graph
bmi@bmi-Alienware:~$

上机实践

- 1，创建一个工作空间
- 2，创建一个包/二个包
- 3，新建一个例子中的CPP文件将例程代码复制进去。
- 4，修改Cmakelist.txt，编译程序。利用ROSrun运行程序
- 5，利用rosnode，rospack，rqt_graph来观察系统
- 6，编写launch文件，启动所有程序。

谢谢！