ROS编程简要指导

前言

ROS WIKI 网址：http://wiki.ros.org/，有相关问题可以在这里查找，善用搜索引擎可以解决大部分的问题！

相关课程推荐：https://www.bilibili.com/video/BV1mJ411R7Ni，看到P30即可。

下文中的示例文件下载地址：https://jbox.sjtu.edu.cn/l/V1oz8J

若有问题，请邮件联系：frenkiedejong@sjtu.edu.cn

如何安装ROS系统

首先，需要安装ubuntu操作系统，需要注意的是ubuntu版本与使用ROS版本相关联，如ubuntu 16.04对应ROS Kinetic, ubuntu 18.04对应ROS Melodic，为防止程序差异的出现，统一使用ubuntu 18.04+ROS Melodic的搭配。

安装ubuntu操作系统的途径主要分为两种：安装双系统或虚拟机，相对来说双系统较为稳定但是安装较为麻烦，虚拟机的一些特定功能可能存在问题但是比较方便，可自行斟酌。

双系统安装方法：https://blog.csdn.net/weixin_44623637/article/details/106723462

虚拟机安装方法：

- 下载ubuntu 18.04的系统镜像：http://mirrors.aliyun.com/ubuntu-releases/18.04/
- 下载并安装虚拟机软件，例如VMware Workstation Pro：https://blog.csdn.net/davidhzq/article/details/101480668
- 在虚拟机软件使用下载的镜像新建虚拟机：https://blog.csdn.net/davidhzq/article/details/102575343
- 注意：为虚拟机分配空间和处理器核心时务必多分配一些，硬盘太少后期不够用非常麻烦，内存太少会使得一些程序无法正常运行，处理器核心太少会使得运行速度较慢，一般选择4核或8核。

由于ubuntu系统默认的软件源均为国外服务器，因此下载常常会因为无法建立可靠链接或速度太慢而报错，解决办法有科学上网和换源两种，换源可换为阿里云或中科大的源，参考网址：https://www.cnblogs.com/ssxblog/p/11357126.html

在ubuntu系统安装完毕后，开始安装ROS Melodic，打开终端/命令行(Terminal)：

- 添加ROS软件源

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

- 添加公钥

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --
recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

- 更新，确保包索引是最新的：

```
 sudo apt update
```

- 检查`cmake`的版本，尽量升级到最新版本：https://blog.csdn.net/RNG_uzi_/article/details/107016899
- 安装ROS Melodic 桌面完整版，其功能较为全面

```
sudo apt install ros-melodic-desktop-full
```

- 初始化 rosdep

```
sudo rosdep init
rosdep update
```

- 若碰到sudo rosdep init错误

```
ERROR: cannot download default sources list from:
https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/sourc
es.list.d/20-default.list
Website may be down
```

解决办法1

```
打开hosts文件
sudo gedit /etc/hosts
#在文件末尾添加
151.101.84.133  raw.githubusercontent.com
#保存后退出再尝试
```

解决办法2

```
sudo c_rehash /etc/ssl/certs
sudo -E rosdep init
rosdep update
```

如果还是提示错误，请将源更更换为**清华源**，然后`sudo apt update`(请将网络换成手机热点) **rosdep update**出错

```
reading in sources list data from /etc/ros/rosdep/sources.list.d
ERROR: unable to process source
[https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/osx-
homebrew.yaml]:
<urlopen error _ssl.c:495: The handshake operation timed out>
(https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/osx-
homebrew.yaml)
Hit
https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/base.
yaml
ERROR: error loading sources list:
The read operation timed out
```

运行：（网络换成手机热点，如果是time out，那就多尝试几次）

```
sudo apt-get update
sudo apt-get install python-rosdep
```

- 设置环境，使得每次source .bashrc时均可刷新setup.bash

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

- 安装rosinstall,便利的工具

```
sudo apt-get install python-rosinstall python-rosinstall-generator
python-wstool build-essential
```

- 检验安装是否成功

```
打开第一个终端，输入
roscore
打开第二个终端，输入
rosrun turtlesim turtlesim_node
打开第三个终端
rosrun turtlesim turtle_teleop_key
选中第三个终端才能进行控制，若正常移动则说明安装成功
```

创建工作空间和功能包，放置源代码

ROS程序运行于称为工作空间(workspace)的环境中，其中一个ROS程序需要被囊括于功能包(package)中。

在完整的工作空间中，存在三个子目录，分别为build，devel和src，前两者通过编译自动生成，不需要作多少改动，功能包源文件放置于src文件夹中，未来的操作也集中在src上。

下面以一个简单的示例程序进行说明：

```
mkdir catkin_ws                 # 创建工作空间
cd catkin_ws                    # 进入工作空间
mkdir src                       # 创建src子目录
cd src                          # 进入子目录
catkin_create_pkg hello_ros     # 创建名为hello_ros的功能包
```

在完成上述操作后，能在~/catkin_ws/src/hello_ros中看到package.xml和CMakeLists.txt两个文件。CMakeLists.txt是一个CMake的脚本文件，这个文件包含了一系列的编译指令，包括生成可执行文件，需要哪些源文件，以及哪里可以找到所需头文件和链接库，package.xml主要描述了文件涉及其他包的使用情况，将在后文中做详细说明。

编写一个测试程序，不妨称为hello.cpp，首先在~/catkin_ws/src/hello_ros中创建名为src的文件夹，将cpp文件放入其中。

```
cd hello_ros                    # 进入功能包中
mkdir src                       # 新建src文件夹
cd src
touch hello.cpp                 # 创建hello.cpp文件
```

```
#include <ros/ros.h>
int main(int argc, char **argv)
{
    ros::init(argc,argv,"hello");        // 初始化ROS客户端
库，"hello"为节点默认名
    ros::NodeHandle n;                   // 创建句柄，把程序注册为节点管理
器的节点
    ROS_INFO("Hello ROS!");              // ROS流信息，输出到控制台并打印
    ros::spinOnce();                     // 信息回调处理函数
}
```

修改相关配置文件

在放置好源文件后，需要修改相关配置文件，确保编译程序时，能够建立正确的链接关系。

首先修改CMakeLists.txt，默认生成文件中包含非常全面的解释，在案例中只需要保留以下部分:

```
```

```
cmake_minimum_required(VERSION 2.8.3)
# 程序的名字
project(hello_ros)
# 使用的内置包，这里用到了roscpp
find_package(catkin REQUIRED COMPONENTS  roscpp)

# 链接本地库
# INCLUDE_DIRS - 声明给其它package的include路径
# LIBRARIES - 声明给其它package的库
# CATKIN_DEPENDS - 本包依赖的catkin package
# DEPENDS - 本包依赖的非catkin package
# CFG_EXTRAS - 其它配置参数
catkin_package(
   CATKIN_DEPENDS roscpp
)

# 头文件位置，一般不需要改动，就是include
include_directories( include ${catkin_INCLUDE_DIRS})

# 生成可执行文件，将src/hello.cpp生成可执行文件hello
add_executable(hello src/hello.cpp)
# 将可执行文件与其依赖的库相关联，第一个参数为可执行文件，其余所有参数均为库
target_link_libraries(hello ${catkin_LIBRARIES})
```

由于此程序引入了roscpp，因此需要修改package.xml：

```
<?xml version="1.0"?>
<package format="2">
  <name>hello_ros</name>
  <version>0.0.0</version>
  <description>The hello_ros package</description>
  <maintainer email="chang@todo.todo">chang</maintainer>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <exec_depend>roscpp</exec_depend>
  <export>
  </export>
</package>
```

其中各项的含义如下图所示，主要修改depend相关部分：

<pacakge>	根标记文件
<name>	包名
<version>	版本号
<description>	内容描述
<maintainer>	维护者
<license>	软件许可证
<buildtool_depend>	编译构建工具，通常为catkin
<depend>	指定依赖项为编译、导出、运行需要的依赖，最常用
<build_depend>	编译依赖项
<build_export_depend>	导出依赖项
<exec_depend>	运行依赖项
<test_depend>	测试用例依赖项
<doc_depend>	文档依赖项

编译并运行程序

下一步回退到目录~/catkin_ws/，进行编译：

```
catkin_make
```

编译成功后，便可以运行可执行文件：

```
打开一个终端，启动ros内核，输入
roscore
打开第二个终端，输入
cd ~/catkin_ws
source devel/setup.bash
rosrun hello_ros hello
其中hello_ros为CMakeLists.txt中project()自定义的项目名称，hello为可执行文件名称
在每一次使用rosrun前，均需要使用source更新环境，否则会出现奇奇怪怪的错误
```

若一切正常，应在第二个终端中看到以下信息：

```
[ INFO] [1630826936.067177265]: Hello ROS!
```

引入自定义头文件的方式

由于示例程序较为简单，主要目的是初步了解程序编写和运行方式，并未涉及到头文件，但是在使用中不可避免地需要引入自定义的头文件，下面介绍加入头文件的方法。

第一步，新建include文件夹，在其中建立一个与之前package同名的文件夹：

```
cd ~/catkin_ws/src/hello_ros
mkdir include
cd include
mkdir hello_ros
```

接下来就可以把自定义的.h文件放入其中，注意头文件中定义的.cpp文件依旧需要放在src文件夹中。

示例文件header.h, header.cpp的定义和hello.cpp的改动情况如下：

```
// header.h
#ifndef HEADER
#define HEADER

#include <iostream>

class Header{
public:
  void print();
};

#endif

// header.cpp
#include "hello_ros/header.h"

void Header::print()
{
  std::cout << "Header Test Success!" << std::endl;
}

//hello.cpp
#include <ros/ros.h>
#include "hello_ros/header.h"

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

```
{
    ros::init(argc,argv,"hello");           // 初始化ROS客户端
库，"hello"为节点默认名
    ros::NodeHandle n;                      // 创建句柄，把程序注册为节点
管理器的节点
    Header test;
    test.print();
    ROS_INFO("Hello ROS!");                 // ROS流信息，输出到控制台并打
印
    ros::spinOnce();                        // 信息回调处理函数
}
```

在加入头文件后，需要对CMakeLists.txt进行改动，注意与上一次定义的不同点：

```
cmake_minimum_required(VERSION 2.8.3)
# 程序的名字
project(hello_ros)
# 使用的内置包，这里用到了roscpp
find_package(catkin REQUIRED COMPONENTS  roscpp)

# 链接本地库
# INCLUDE_DIRS - 声明给其它package的include路径
# LIBRARIES - 声明给其它package的库
# CATKIN_DEPENDS - 本包依赖的catkin package
# DEPENDS - 本包依赖的非catkin package
# CFG_EXTRAS - 其它配置参数
catkin_package(
   INCLUDE_DIRS include
   CATKIN_DEPENDS roscpp
)

# 头文件位置，一般不需要改动，就是include
include_directories( include ${catkin_INCLUDE_DIRS})

# 生成静态库
add_library(Header
    include/hello_ros/header.h
    src/header.cpp
)
add_dependencies(Header ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(Header ${catkin_LIBRARIES})
```

```
# 生成可执行文件，将src/hello.cpp生成可执行文件hello
add_executable(hello src/hello.cpp)
target_link_libraries(hello Header ${catkin_LIBRARIES})
```

修改完成后，同上文中编译运行，可以看到以下输出信息：

```
Header Test Success!
[ INFO] [1630893156.953213151]: Hello ROS!
```

关于ROS中的信息传递：message

ROS使用简化的消息描述语言来描述ROS节点发布的数据（即消息）。消息描述存储在ROS包的msg 子目录中的.msg文件中。.msg文件就是简单的文本文件，每行都有一个字段类型和字段名称。在ROS库中存在一些已定义，可直接使用的msg类型，包括std_mags，geometry_msgs等。

在.msg文件中，可以使用的类型包括但不局限于：

- int8, int16, int32, int64
- float32, float64
- string
- time, duration
- 其他的msg文件（嵌套）
- variable-length array[] 和 fixed-length array[C]

  如果需要使用库中的msg文件，只需要在源文件中添加例如以下的引用语句：

  ```
  #include <geometry_msgs/Twist.h>
  ```

  当然，已存在的文件类型不能完全囊括我们的需求，当需要自定义msg文件时，应该如何操作？

  首先，进入工作空间的包内，创建名为msg的新文件夹，并在其中创建test.msg，定义如下：

```
cd ~/catkin_ws/src/hello_ros
mkdir msg
cd msg
touch test.msg
```

```
int32 x
int32 y
```

下面修改package.xml，为使用message文件，需要添加一些依赖项：

```
  <build_depend>message_generation</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend> message_runtime</exec_depend>
```

再修改CMakeLists.txt，修改为以下形式，注意与前面的区别：

```
cmake_minimum_required(VERSION 2.8.3)
# 程序的名字
project(hello_ros)
# 使用的内置包，这里用到了roscpp
find_package(catkin REQUIRED COMPONENTS
  roscpp
  message_generation
  std_msgs
)

## Generate messages in the 'msg' folder
add_message_files(
   FILES
   test.msg
)

## Generate added messages and services with any
dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

```
# 链接本地库
# INCLUDE_DIRS - 声明给其它package的include路径
# LIBRARIES - 声明给其它package的库
# CATKIN_DEPENDS - 本包依赖的catkin package
# DEPENDS - 本包依赖的非catkin package
# CFG_EXTRAS - 其它配置参数
catkin_package(
   INCLUDE_DIRS include
   CATKIN_DEPENDS message_runtime std_msgs roscpp
)

# 头文件位置，一般不需要改动，就是include
include_directories( include ${catkin_INCLUDE_DIRS})

# 生成静态库
add_library(Header
    include/hello_ros/header.h
    src/header.cpp
)
add_dependencies(Header ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(Header ${catkin_LIBRARIES})


# 生成可执行文件，将src/hello.cpp生成可执行文件hello
add_executable(hello src/hello.cpp)
target_link_libraries(hello Header ${catkin_LIBRARIES})
```

经历上述步骤后，进行编译，便可以完成了自定义msg文件的过程，那么如何在程序中使用它们呢？

在hello.cpp中，加入以下语句，便可引用自定义test.msg内容，其用法与一般的头文件区别不大，其中需要注意的是形如int32[]的数据，其为不定长数组，相当于vector类型。

```
#include "hello_ros/test.h"

hello_ros::test msg;
```

可以用指令验证是否成功定义消息类型，打开命令行输入：

```
rosmsg show hello_ros/test
```

可以看到输出信息：

```
int32 x
int32 y
```

如果你忘记msg定义在哪个功能包中，也可以输入：

```
rosmsg show test
```

此时输出信息会指示存放的功能包名称：

```
[hello_ros/test]:
int32 x
int32 y
```

ROS中的talker和listener

既然已经说完了ROS中消息的格式，那么显然还需要消息发送者和接受者来构成完整的消息收发结构，我们把消息发送者称为talker，消息接收者称为listener。

进入~/catkin_ws/src/hello_ros/src目录，创建talker.cpp和listener.cpp，将以下内容复制到其中：

```
// talker.cpp
#include "ros/ros.h"
#include "hello_ros/test.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  // publish test.msg, a cache of 1000 message
  ros::Publisher chatter_pub = n.advertise<hello_ros::test>
("chatter", 1000);
  // frequency of 10hz
  ros::Rate loop_rate(10);

  /**
```

```
   * A count of how many messages we have sent. This is used to
create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then
publish it.
     */
    hello_ros::test msg;
    msg.x = 1;
    msg.y = 2;
    ROS_INFO("send x = %d, y = %d", msg.x, msg.y);
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}

// listener.cpp
#include "ros/ros.h"
#include "hello_ros/test.h"

// callback function for listener, notice that msg is a pointer
void chatterCallback(const hello_ros::test::ConstPtr& msg)
{
  ROS_INFO("I heard: x = %d, y = %d", msg->x, msg->y);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  // subscribe from "chatter", use callback function
chatterCallback
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);
  ros::spin();
```

```
  return 0;
}
```

然后再修改CMakeLists.txt，在结尾加入talker.cpp和listener.cpp产生可执行文件的命令即可：

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker Header ${catkin_LIBRARIES})

add_executable(listener src/listener.cpp)
target_link_libraries(listener Header ${catkin_LIBRARIES})
```

进行编译后即可运行，需要三个命令行窗口：

- 第一个运行roscore命令
- 第二个运行rosrun hello_ros talker
- 第三个运行rosrun hello_ros listener
- 若出现package hello_ros not found之类的报错，说明环境变量没有刷新，进入~/catkin_ws/devel中source一下即可。

如果一切正常，应该能在talker的窗口看到重复的“send x = 1, y = 2”，在listener窗口出现“I heard: x = 1, y = 2”。

关于功能包间调用和依赖关系

介绍完ROS中头文件和消息类型的使用方法，请想象这样一种情景：工程分为两个模块，每个模块被放置在一个功能包中，模块间共用了一些头文件，并且模块间使用message进行通信，你是否能看出存在哪些问题？

首先对于头文件，一种自然的想法是在包含两个功能包中分别定义一次头文件，但是这样的改动较为复杂，并且在未来需要更改头文件的内容时，需要在两处进行相同的改动，对于文件的同步造成了不小的困难。

其次是对于message，注意到程序中定义talker和listener时，对于消息类型的引用是hello_ros::test，那么如果还是在两个功能包中分别定义的话，其所处的域名自然也就不同，会造成错误。

上述两种问题的解决方案很简单：额外创建功能包放置头文件和消息类型，模块所在功能包只需要引用这一个功能包，便可以实现期望的功能。在实际的应用中，常常将所有的消息都放置于一个功能包中，将自定义的头文件放置于另一个功能包中，如此使得文件功能明确，在调试中较为方便。

假设需要为功能包pkg_d添加对功能包pkg_a的依赖，方法主要有两种：

1. 创建包时就添加依赖：

```
# 对pkg_d添加对pkg_a的依赖
catkin_create_pkg pkg_d std_msgs roscpp rospy pkg_a
# 在编译pkg_d之前，单独编译pkg_a
catkin_make -DCATKIN_WHITELIST_PACKAGES='pkg_a'
```

这样就可以在pkg_d程序中引用pkg_a的内容，只需要include "pkg_a/..."，pkg_a::...即可。

当然，还需要修改配置文件，但是由于在创建包时已经添加了依赖项，因此已经完成了对pkg_a的依赖配置，只需要生成可执行文件即可。

2. 手动修改依赖关系

可以很明显地看出，在创建包时就添加依赖非常的简单有效，但是在创建包的时候，往往并不完全清楚之后是否需要添加依赖。或者单纯因为忘记就并未添加依赖项。在这种情况下，就需要手动修改依赖关系。

首先修改CMakeLists.txt：

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  pkg_a
)
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```

其次修改package.xml：

```
  <build_depend>pkg_a</build_depend>
  <build_export_depend>pkg_a</build_export_depend>
  <exec_depend>pkg_a</exec_depend>
```

同样先单独编译pkg_a，再完整编译即可。

关于roslaunch

从talker和listener的实践过程中，相信你已经感受到，使用rosrun同时启动多个节点是一件非常复杂且痛苦的事，不仅需要使用一个窗口运行roscore，每一个节点还需要单独占用一个命令行，如果遇到大工程，其繁杂程度不可想象。解决这种问题的方法也很简单，那就是roslaunch。只需要编写一个launch文件，就可以按照指令运行工程，并且roscore是自动启动和关闭的。

首先添加一个launch文件：

```
cd ~/catkin_ws/src/hello_ros
mkdir launch
cd launch
touch hello.launch
```

下面介绍launch文件中的一些重要标签：

- ：文件的核心内容被...的结构包括在其中
- ：包含的部分属性和其作用如下表所示，主要用到前四条

属性	属性作用
name="NODE_NAME"	为节点指派名称，这将会覆盖掉**ros::init()**定义的**node_name**
pkg="PACKAGE_NAME"	节点所在的包名
type="FILE_NAME"	执行文件的名称，即编译生成的可执行文件名
output="screen"	终端输出转储在当前的控制台上，而不是在日志文件中
respawn="true"	当**roslaunch**启动完所有该启动的节点之后，会监测每一个节点，保证它们正常的运行状态。对于任意节点，当它终止时，**roslaunch** 会将该节点重启
required="true"	当被此属性标记的节点终止时，**roslaunch**会将其他的节点一并终止。注意此属性不可以与**respawn="true"**一起描述同一个节点

- ：param标签用来传递单个参数
- ：rosparam标签允许从YAML文件中一次性导入大量参数

介绍完了launch文件的基本结构，便可以使用它实现一些简单的功能，这里介绍同时启动多个节点和导入参数的方法。

根据上面介绍的标签，使用和来启动多个节点。注意name值若与程序中的定义名不同，以name为准，一般取name值和后面type值一致，那么以talker和listener为例，就可以这样编写：

```
<launch>
    <node name="talker" pkg="hello_ros"
          type="talker" output="screen"/>
    <node name="listener" pkg="hello_ros"
          type="listener" output="screen"/>
</launch>
```

使用命令roslaunch hello_ros hello.launch运行launch文件，可以看到命令行中首先启动了roscore，并且在打印几次talker信息后，交替出现talker和listener的信息。这是由于设定talker先于listener启动，于是存在了一定的时间差。

下面介绍单个参数的添加方式，使用标签。首先同上建立param.launch，将下面内容复制到其中。

```
<launch>
    <param name = "test"  type = "double"
       value = "10.0" />
    <node  name = "hello" pkg = "hello_ros"
       type = "hello" output = "screen"/>
</launch>
```

再修改hello.cpp为下面的形式，使用getParam函数，获取名为"test"的参数并打印。

```
#include <ros/ros.h>
#include "hello_ros/header.h"
#include "hello_ros/test.h"

int main(int argc, char **argv)
{
    ros::init(argc,argv,"hello");          // 初始化ROS客户端
库，"hello"为节点默认名
```

```
    ros::NodeHandle n;                          // 创建句柄，把程序注册为节点管理器的节点
    Header test;
    test.print();
    ROS_INFO("Hello ROS!");                     // ROS流信息，输出到控制台并打印

    double ret;
    n.getParam("test", ret);
    ROS_INFO("value of the parameter is.............. %f", ret);
    ros::spinOnce();                            // 信息回调处理函数
}
```

运行roslaunch hello_ros param.launch后，在终端显示：

```
[ INFO] [1630981772.794011484]: value of the parameter
is.............. 10.000000
```

当然，也可以传入数组形式，使用标签为，在程序中使用vector进行接收即可。

```
<rosparam param = "vector_param">[1, 2, 3, 4]</rosparam>
```

当一次需要导入非常多参数时，像上面一样使用标签导入，每一条参数都需要一个标签，显得比较复杂，因此使用标签，利用一个额外的yaml文件进行大量参数的传递。

首先，新建一个名为config的文件夹，在其中新建yaml文件，yaml文件的固定格式为变量名: 变量值。

```
noise: 10.0
string_var: abc
vector_var: [1,2,3]
```

创建launch文件read_param.launch，其中$(find hello_ros)用来定位功能包hello_ros目录的位置：

```
<launch>
    <rosparam file="$(find hello_ros)/config/param.yaml"
          command="load" />
    <node name="hello" pkg = "hello_ros"
      type = "hello" output = "screen"/>
</launch>
```

修改hello.cpp为以下形式，获取参数的方式与标签一致，使用getParam函数：

```
#include <ros/ros.h>
#include "hello_ros/header.h"
#include "hello_ros/test.h"
#include <string>
#include <vector>

int main(int argc, char **argv)
{
    ros::init(argc,argv,"hello");           // 初始化ROS客户端
库，"hello"为节点默认名
    ros::NodeHandle n;                      // 创建句柄，把程序注册为节点
管理器的节点
    Header test;
    test.print();
    ROS_INFO("Hello ROS!");                 // ROS流信息，输出到控制台并
打印

    // double ret;
    // n.getParam("test", ret);
    // ROS_INFO("value of the parameter is.............. %f",
ret);

    double noise;
    if(n.getParam("noise", noise))
        ROS_INFO("noise is %f", noise);

    std::string string_var;
    if (n.getParam("string_var", string_var))
        ROS_INFO("string_var: %s", string_var.c_str());

    std::vector<int> vector_var;
    if (n.getParam("vector_var",vector_var))
        ROS_INFO("got vector");
```

```
    ros::spinOnce();                                    // 信息回调处理函数
}
```

运行roslaunch后，出现下面的信息：

```
[ INFO] [1630983408.812129605]: noise is 10.000000
[ INFO] [1630983408.813179938]: string_var: abc
[ INFO] [1630983408.813745095]: got vector
```

关于dynamic reconfigure的使用

在程序的测试过程中，存在一些需要利用实际测试才能获得最佳结果的参数，而在实际测试中，每测一次就需要重新构建程序的工作量显然是不可接受的，为应对这种情况，可以考虑使用ROS中动态调参的功能。在完成以下步骤后，便可以得到可视化的调参工具。

首先，在之前创建的config文件夹中，新建dynamic.cfg，其使用python编写，配置了动态调参的参数：

```
#!/usr/bin/env python
PACKAGE = "hello_ros"

from dynamic_reconfigure.parameter_generator_catkin import *

# create a parameter generator
gen = ParameterGenerator()

# add dynamic parameters
gen.add("int_param",    int_t,    0, "An Integer parameter", 50,
 0, 100)
gen.add("double_param", double_t, 0, "A double parameter",    .5,
0,   1)
gen.add("str_param",    str_t,    0, "A string parameter",  "Hello
World")
gen.add("bool_param",   bool_t,   0, "A Boolean parameter",  True)

# another way: enumerate
size_enum = gen.enum([ gen.const("Small",      int_t, 0, "A small
constant"),
```

```
                        gen.const("Medium",     int_t, 1, "A medium
constant"),
                        gen.const("Large",      int_t, 2, "A large
constant"),
                        gen.const("ExtraLarge", int_t, 3, "An extra
large constant")], "An enum to set size")

gen.add("size", int_t, 0, "A size parameter which is edited via an
enum", 1, 0, 3, edit_method=size_enum)

# second parameter: name of the node
# third parameter: consistent with the file
exit(gen.generate(PACKAGE, "dynamic_test", "dynamic"))
```

在完成config文件的编写后，需要为其添加可执行权限：

```
chmod a+x config/dynamic.cfg
```

下面编写一个动态调参的服务器，用于处理参数的改变，称为server.cpp。注意在头文件中"dynamicConfig.h"在上面从未出现，这是以exit()中第三个参数"dynamic"为前缀，添加固定后缀Config.h产生的。

```
#include <ros/ros.h>

#include <dynamic_reconfigure/server.h>
#include "hello_ros/dynamicConfig.h"

// 定义一个回调函数，回调函数的传入参数有两个，一个是新的参数配置值，另外一个表示
参数修改的掩码
void callback(hello_ros::dynamicConfig& config, uint32_t level) {
  ROS_INFO("Reconfigure Request: %d %f %s %s %d",
            config.int_param, config.double_param,
            config.str_param.c_str(),
            config.bool_param?"True":"False",
            config.size);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "dynamic_test");
    // 创建参数动态配置的服务端实例，监听客户端的参数配置请求
```

```
    dynamic_reconfigure::Server<hello_ros::dynamicConfig> server;

 dynamic_reconfigure::Server<hello_ros::dynamicConfig>::CallbackTyp
e f;
    // 将回调函数和服务端绑定，当客户端请求修改参数时，服务端即可跳转到回调函数中
进行处理
    f = boost::bind(&callback, _1, _2);
    server.setCallback(f);

    ROS_INFO("Spinning node");
    ros::spin();
    return 0;
}
```

下面自然需要修改CMakeLists.txt和package.xml。

```
# CMakeLists.txt的核心改动
find_package(catkin REQUIRED COMPONENTS
  roscpp
  message_generation
  std_msgs
  dynamic_reconfigure
)

generate_dynamic_reconfigure_options(
  config/dynamic.cfg
)

# 链接本地库
# INCLUDE_DIRS - 声明给其它package的include路径
# LIBRARIES - 声明给其它package的库
# CATKIN_DEPENDS - 本包依赖的catkin package
# DEPENDS - 本包依赖的非catkin package
# CFG_EXTRAS - 其它配置参数
catkin_package(
   INCLUDE_DIRS include
   CATKIN_DEPENDS message_runtime std_msgs roscpp
dynamic_reconfigure
)

catkin_package(
  LIBRARIES custom_dynamic_reconfigure
```

```
  CATKIN_DEPENDS roscpp std_msgs dynamic_reconfigure
)

# 头文件位置，一般不需要改动，就是include
include_directories( include ${catkin_INCLUDE_DIRS})

# make sure configure headers are built before any node using them
add_executable(dynamic_reconfigure_node src/server.cpp)
# make sure configure headers are built before any node using them
add_dependencies(dynamic_reconfigure_node ${PROJECT_NAME}_gencfg)
# for dynamic reconfigure
target_link_libraries(dynamic_reconfigure_node ${catkin_LIBRARIES})
```

```
# package.xml的核心改动：
  <build_depend>dynamic_reconfigure</build_depend>
  <build_export_depend>dynamic_reconfigure</build_export_depend>
  <exec_depend>dynamic_reconfigure</exec_depend>
```

下面就可以运行程序：

```
# 第一个终端，启动ROS内核
roscore
# 第二个终端，启动服务器
rosrun hello_ros dynamic_reconfigure_node
# 第三个终端，启动ROS提供的可视化参数配置工具
rosrun rqt_reconfigure rqt_reconfigure
```

若遇到如下报错，则说明对应文件缺失，解决方案详见https://www.cnblogs.com/long5683/p/13847352.html

```
[ERROR] [1631012668.802477557]: Skipped loading plugin with error:
XML Document '/opt/ros/melodic/share/rqt_virtual_joy/plugin.xml'
has no Root Element. This likely means the XML is malformed or
missing..
RosPluginProvider._parse_plugin_xml() plugin file
"/opt/ros/melodic/share/rqt_virtual_joy/plugin.xml" in package
"rqt_virtual_joy" not found
RosPluginProvider._parse_plugin_xml() plugin file
"/opt/ros/melodic/share/rqt_virtual_joy/plugin.xml" in package
"rqt_virtual_joy" not found
```

在启动服务器的终端，在可视化参数调节窗口进行修改，可以看到如下的结果：

```
[ INFO] [1508144642.464050963]: Reconfigure Request: 50 0.500000 Hello World True 1
[ INFO] [1508144642.466430198]: Spinning node
[ INFO] [1508144747.189317033]: Reconfigure Request: 65 0.500000 Hello World True 1
[ INFO] [1508144752.631543877]: Reconfigure Request: 65 0.230000 Hello World True 1
[ INFO] [1508144757.396002236]: Reconfigure Request: 65 0.230000 hcx True 1
[ INFO] [1508144761.109123375]: Reconfigure Request: 65 0.230000 hcx True 2
[ INFO] [1508144762.807916946]: Reconfigure Request: 65 0.230000 hcx False 2
[ INFO] [1508144763.515548408]: Reconfigure Request: 65 0.230000 hcx True 2
```