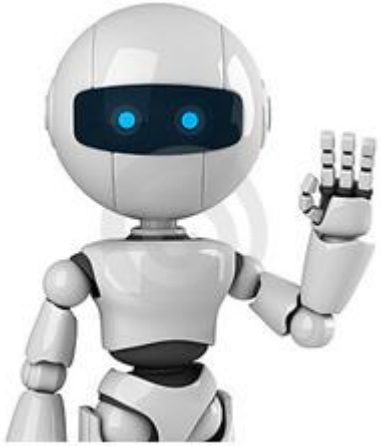
2(2)：ROS系统的基本操作



东北大学 张云洲

内 容 提 纲

- 工作空间
- 功能包的创建与编译
- 使用节点
- 使用参数服务
- 创建和编译节点
- 创建和编译msg/srv文件
- 动态参数

2.1 ROS功能包指令

为了获得功能包和功能包集的信息，或移动文件，需要用

rospack profile: 新安装的功能包

rospack find : 寻找功能包的路径

rosstack depends：功能包的依赖项

roscd：更改目录

rosls：列出文件

例：$ rospack find turtlesim

$ rosls turtlesim

创建工作空间

catkin：（1）卡婷是一个广告公司？（2）菜黄花？

catkin：一个ROS中的工具。

……

catkin_make是一个非常方便的命令行工具。

```
$ mkdir -p ~/dev/catkin_ws/src
$ cd ~/dev/catkin_ws/src
$ catkin_init_workspace
……
$ cd ~/dev/catkin_ws
$ catkin_make
```

工作空间

初始化后：

仅有CMakeList.txt，没有功能包。

编译后：

文件夹：build、devel、src

主要保存个人的一些个性化设置，如命令别名、路径等。

完成配置：

$ source devel/setup.bash

运行工作空间中的ROS节点频繁使用source devel/setup.bash，建议将该命令加到.bashrc中：

$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

创建ROS功能包

用roscreate-pkg命令行工具更方便。

$ cd ~/dev/catkin_ws/src

$ catkin_create_pkg chapter2_tutorials std_msgs roscpp

格式包括功能包的名称和依赖项：

catkin_create_pkg [package_name] [depend1] [depend2] [depend3]

std_msgs：包含常见消息类型，表示基本数据类型和其他基本的消息构造，如多维数组。

rospy　一个ROS的纯Python客户端库。

roscpp　使用C++实现ROS的各种功能。

使用ROS节点

都是可执行程序，位于packagename/bin目录。

启动之前：roscore

获得节点信息：rosnode。具体：

$ rosnode list

$ rosnode info

$ rosnode ping

$ rosnode machine

$ rosnode kill

$ rosnode cleanup

例：

rosnode info /turtlesim

rosrun turtlesim turtlesim_node

rosrun指令：

rosrun + package名 + node 可以直接运行pacakge下的node，避免了输入具体路径。

```
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /spawn
 * /clear
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill

contacting node http://127.0.0.1:43753/ ...
Pid: 32298
Connections:
 * topic: /rosout
 * to: /rosout
 * direction: outbound
 * transport: TCPROS
```

ROS节点Node及相关概念的补充

1. 相关概念的补充

Nodes—A node is an executable that uses ROS to communicate with other nodes.

Messages—ROS data type used when subscribing or publishing to a topic.

Topics—Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.

Master—Name service for ROS (i.e. helps nodes find each other)

rosout—ROS equivalent of stdout/stderr

roscore—Master + rosout + parameter server (parameter server will be introduced later)

ROS节点Node及相关概念的补充

2. 节点Nodes

节点其实就是一个ROS package中的可执行文件，它使用client library与其他Node通信。

nodes可以publish或者subscribe一个Topic，还可以提供或者使用Service。

3. 客户端库 Cilent Libraries

rospy = python cilent library

roscpp = c++ cilent library

5. roscore : 使用ROS时，第一件事情就是运行roscore。

6. 如果没有initialize roscore，可能会出现network configuration issue问题。

7. 如果roscore 没有初始化并且发了一条消息说缺少permissions，则运行：

sudo chown -R <your_username> ~/.ros

使用主题与ROS节点交互

1. 设置

- 运行 roscore
- 在新的 terminal 运行 turtlesim

  $ rosrun turtlesim turtlesim_node
- 用键盘遥控turtle

  $ rosrun turtlesim turtle_teleop_key

2. ROS Topics

turtle_teleop_key node 发布（publish） Topic

turtlesim_node node 订阅（subscribe） Topic，两者相互通信。

使用主题与ROS节点交互

使用rqt_graph

rqt_graph是rqt package的一部分，创建系统运行的动态图（dynamic graph）。

rqt package的安装：

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

运行

```
$ rosrun rqt_graph rqt_graph
```

结果：

使用主题与ROS节点交互

进行交互并获取主题信息，可使用rostopic工具，其接受以下参数：

rostopic bw　　显示主题所使用的带宽。
rostopic echo　将消息输出到屏幕。
rostopic find　按照类型查找主题。
rostopic hz　　显示主题的发布频率。
rostopic info　输出活动主题的信息。
rostopic list　输出活动主题的列表。
rostopic pub　将数据发布到主题。
rostopic type　输出主题的类型。

通过help查询子命令：　$ rostopic -h

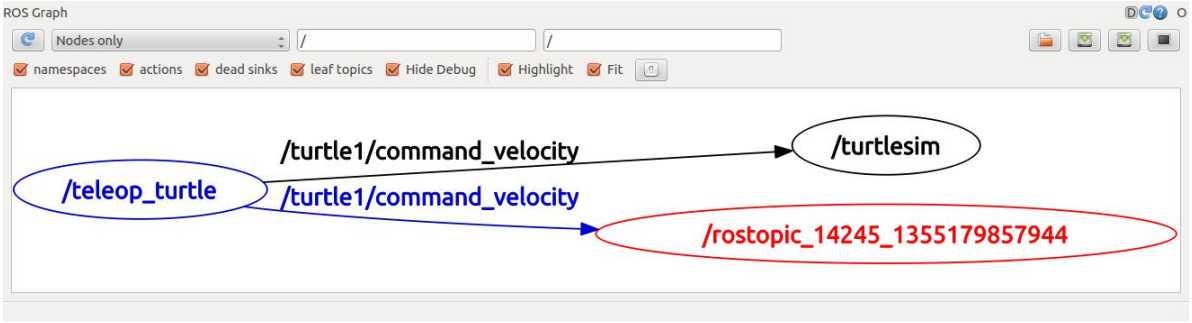
使用主题与ROS节点交互

例：使用rostopic echo显示top发布的数据

$ rostopic echo /turtle1/cmd_vel

若没看到内容，说明没有数据发布，选中turtle_teleop_key的terminal，按键盘上的键盘键控制即可。

此时再看rqt_graph，按左上角refresh键，这时会显示新的node




```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

使用主题与ROS节点交互

例：使用rostopic list 显示所有发布和被订阅的topics

1）查看

$ rostopic list -h

2）得到以下结果

Usage: rostopic list [/topic]

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

使用主题与ROS节点交互

3）若需要详细信息verbose，输入

$ rostopic list -v

得到以下结果

Published topics:

* /turtle1/color_sensor [turtlesim/Color] 1 publisher

* /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher

* /rosout [rosgraph_msgs/Log] 2 publishers

* /rosout_agg [rosgraph_msgs/Log] 1 publisher

* /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:

* /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber

* /rosout [rosgraph_msgs/Log] 1 subscriber

使用主题与ROS节点交互

ROS消息Message

- 关于topic的节点间通信通过发送节点间的ROS Message实现，发布者和订阅者需要发送和接受相同类型的message，topic type由message type定义，发送topic的message类型可通过rostopic type确定。
- rostopic type使用方法

  rostopic type [topic]

  如：

  $ rostopic type /turtle1/cmd_vel

  得到：

  geometry_msgs/Twist

使用主题与ROS节点交互

- 使用rosmsg可以得到message的详细信息：

  $ rosmsg show geometry_msgs/Twist

得到：

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

使用主题与ROS节点交互

- 伴有message的rostopic

  使用rostopic pub

  用法：

  rostopic pub [topibc] [msg_type] [args]

  比如：

  $ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

使用主题与ROS节点交互

解释：

rostopic pub：发布message关于指定的topic

-1: 只发布一条message然后退出

/turtle1/cmd_vel： 要发布的topic的名字

geometry_msgs/Twist： message的类型

--： 告诉选项解析器后面的内容不是可选的，有负数时一定要有这项

'[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]' ：第一行向量是线性值，第二行是角度值。

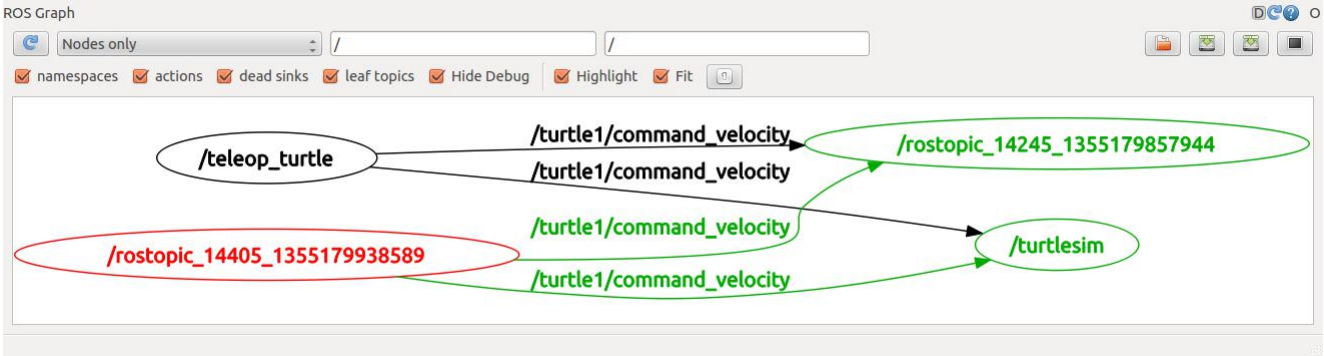
使用主题与ROS节点交互

乌龟停止了运动,因为需要稳定的指令流来保持运动。

$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

在速度topic上发布了速度命令，速率为1Hz

在rqt_graph上按左上角的refresh：



查看被发布的数据：

rostopic echo /turtle1/pose

使用主题与ROS节点交互

- 使用rostopic hz来指导数据发布的速率：

用法：

rostopic hz [topic]

比如：

$ rostopic hz /turtle1/pose

可以通过rosmsg得到更多的信息：

$ rostopic type /turtle1/cmd_vel | rosmsg show

结果：

```
subscribed to [/turtle1/pose]
average rate: 59.354
        min: 0.005s max: 0.027s std dev: 0.00284s window: 58
average rate: 59.459
        min: 0.005s max: 0.027s std dev: 0.00271s window: 118
average rate: 59.539
        min: 0.004s max: 0.030s std dev: 0.00339s window: 177
average rate: 59.492
        min: 0.004s max: 0.030s std dev: 0.00380s window: 237
average rate: 59.463
        min: 0.004s max: 0.030s std dev: 0.00380s window: 290
```

使用ROS节点服务

服务是能够使节点之间相互通信的另一种方法。服务允许节点发送请求和接收响应。

可使用rosservice工具与服务进行交互，接受参数：

rosservice args /service	输出服务参数。
rosservice call /service args	根据命令行参数调用服务。
rosservice find msg-type	根据服务类型查询服务。
rosservice info /service	输出服务信息。
rosservice list	输出活动服务清单。
rosservice type /service	输出服务类型。
rosservice uri /service	输出服务的ROSRPC URI。

使用ROS节点服务

输出活动服务清单

输入：rosservice list

保持上一次练习的turtlesim_node继续运行

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/rostopic_11582_1588085768683/get_loggers
/rostopic_11582_1588085768683/set_logger_level
/rqt_gui_py_node_10627/get_loggers
/rqt_gui_py_node_10627/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

使用ROS节点服务

输出服务类型 rosservice type /service

例如：如查看clear类型

```
studyon@studyon-virtualPC:~$ rosservice type /clear
std_srvs/Empty
```

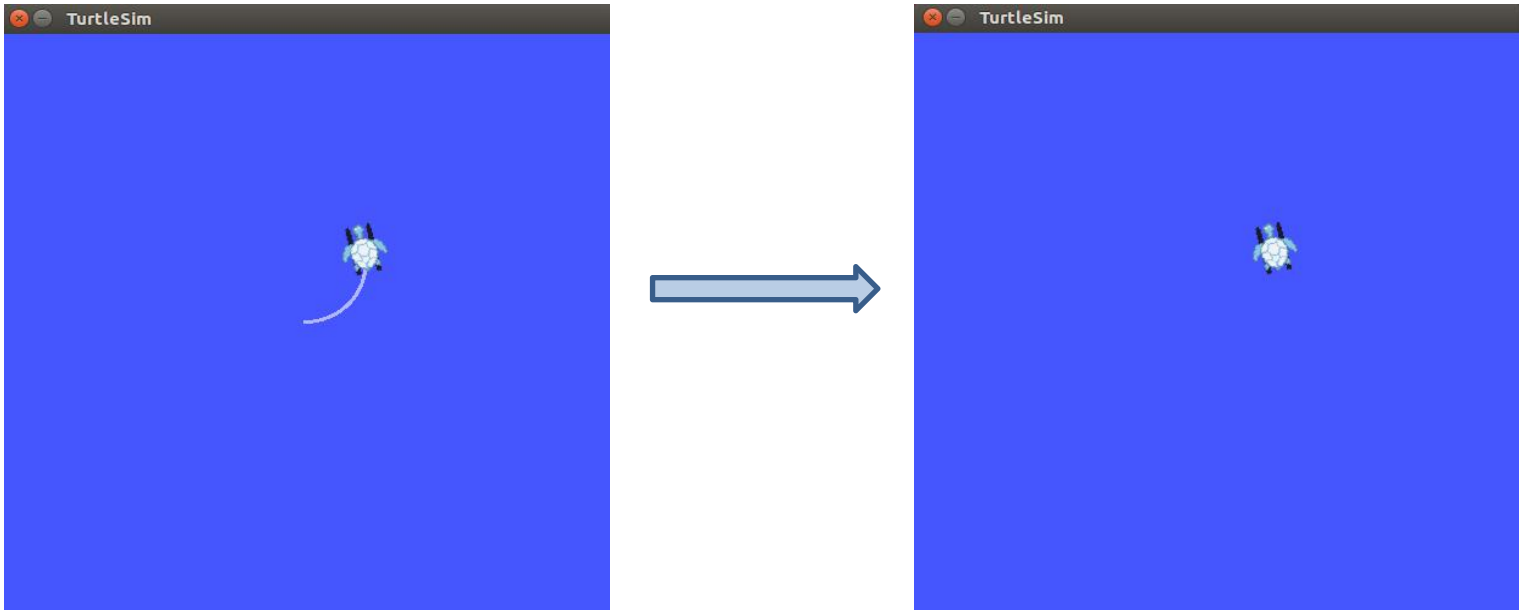
服务的类型为空（empty)，表明在调用该服务时不需要参数（比如，请求不需要发送数据，响应也没有数据）。

使用ROS节点服务

根据命令行参数调用服务 rosservice call /service args

例如：清理turtlesim_node的背景

输入：rosservice call /clear



使用ROS节点服务

查看带有参数的服务/spwan的参数

rosservice type /spawn | rossrv show

```
studyon@studyon-virtualPC:~$ rosservice type /spawn | rossrv show
float32 x
float32 y
float32 theta
string name
---
string name
```

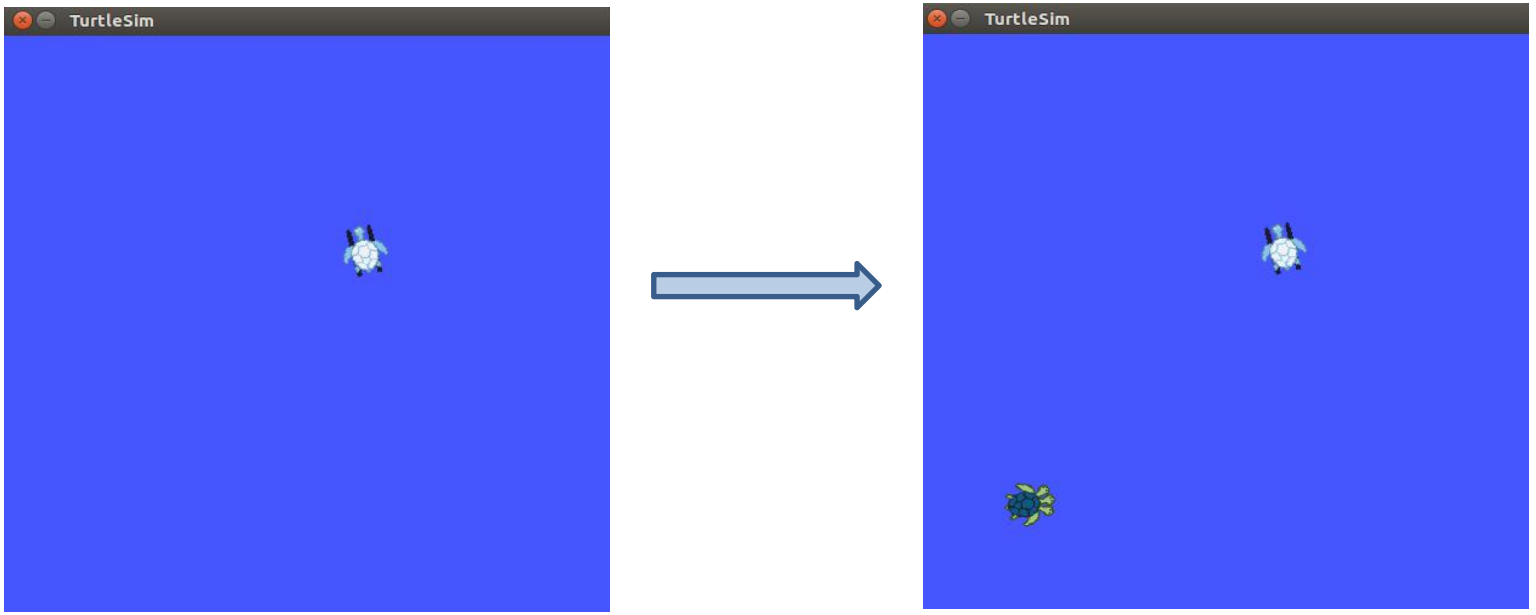
可以看出/spawn服务类型的数据参数为3个float32表示位置（x，y）朝向theta和一个字符串表示名字name。

使用ROS节点服务

根据命令行参数调用服务 /spwan

这里不具体设置名字，让turtlesim自动创建一个

输入：rosservice call /spawn 2 2 0.2 ""



ROS参数服务器

- rosparam set parameter value: 赋值
- rosparam get parameter: 读取
- rosparam load file: 从文件读参数
- rosparam dump file: 将参数写入文件
- rosparam delete parameter: 删除参数
- rosparam list: 列出服务器中的所有参数

使用ROS节点参数

rosparam使得我们能够存储并操作ROS 参数服务器（Parameter Server）上的数据。参数服务器能够存储整型、浮点、布尔、字符串、字典和列表等数据类型。

rosparam使用YAML标记语言的语法。一般而言，YAML的表述很自然：1 是整型, 1.0 是浮点型, one是字符串, true是布尔, [1, 2, 3]是整型列表, {a: b, c: d}是字典。rosparam有很多指令可以用来操作参数，基本用法：

rosparam set 设置参数　　rosparam load 从文件读取参数

rosparam get 获取参数　　rosparam delete 删除参数

rosparam dump 向文件中写入参数 rosparam list 列出参数名

使用ROS节点参数

列出参数名：

输入： rosparam list

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_ylh_tuf_gaming_fx504gm_fx80gm__34349
/rosversion
/run_id
```

该命令可以列出当前参数服务器所有参数名，可以看到turtlesim节点在参数服务器上有3个参数用于设定背景颜色。

使用ROS节点参数

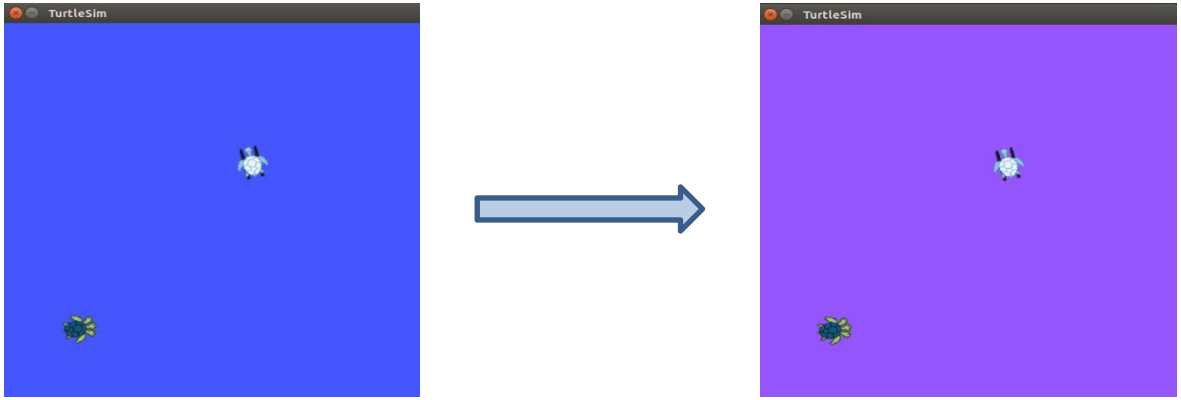
设置参数 rosparam set

例如：修改背景颜色的红色通道

rosparam set /background_r 150

上述指令修改了参数的值，现在调用清除服务使得修改后的参数生效。

rosservice call /clear



使用ROS节点参数

获取参数 rosparam get

例如：获取背景的绿色通道的值

输入：rosparam get /background_g

可以使用rosparam get /来显示参数服务器上的所有内容

输入：rosparam get /

```
background_b: 255
background_g: 86
background_r: 150
rosdistro: 'kinetic

  '
roslaunch:
  uris: {host_ylh_tuf_gaming_fx504gm_fx80gm__34349: 'http://ylh-TUF-GAMING-FX504GM-FX80GM:34349/'}
rosversion: '1.12.14

  '
run_id: cbeb7df0-895f-11ea-9d22-34e12d159429
```

使用ROS节点参数

向文件中写入参数 rosparam dump 从文件读取参数 rosparam load

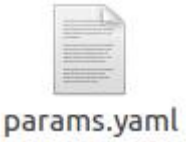
基本用法：

rosparam dump [file_name] [namespace]

rosparam load [file_name] [namespace]

例如：将所有参数写入params.yaml里

rosparam dump params.yaml



params.yaml (~/) - gedit

打开(O)　保存(S)

```
background_b: 255
background_g: 86
background_r: 150
rosdistro: 'kinetic

  '
roslaunch:
  uris: {host_ylh_tuf_gaming_fx504gm_fx80gm__35299: 'http://ylh-TUF-GAMING-FX504GM-FX80GM:35299/'}
rosversion: '1.12.14

  '
run_id: 01940ad6-89af-11ea-81cd-34e12d159429
```

使用ROS节点参数

也可以将yaml文件重载入新的命名空间如copy里

输入：rosparam load params.yaml copy

调用rosparam get命令可以看出已成功载入新的命名空间copy中

输入：rosparam get /

```
background_b: 255
background_g: 86
background_r: 150
copy:
  background_b: 255
  background_g: 86
  background_r: 150
  rosdistro: 'kinetic

    '
  roslaunch:
    uris: {host_ylh_tuf_gaming_fx504gm_fx80gm__35299: 'http://ylh-TUF-GAMING-FX504GM-FX80GM:35299/'}
  rosversion: '1.12.14

    '
  run_id: 01940ad6-89af-11ea-81cd-34e12d159429
rosdistro: 'kinetic

  '
roslaunch:
  uris: {host_ylh_tuf_gaming_fx504gm_fx80gm__35299: 'http://ylh-TUF-GAMING-FX504GM-FX80GM:35299/'}
rosversion: '1.12.14

  '
run_id: 01940ad6-89af-11ea-81cd-34e12d159429
```

范例：创建ROS节点

- 编写程序
- 编译节点
- 编译功能包
- 运行

1. 编写程序

打开功能包的src文件夹

/dev/catkin_ws/chapter2_tutorials/src/

输入：$ roscd chapter2_tutorials/src/

在该文件夹中创建两个新文件（节点）：一个发送数据、一个接收数据

example1_a.cpp、example1_b.cpp

编写程序：

```
//example1_a.cpp: 发送数据

#include "ros/ros.h"                    //"ros/ros.h"包含ROS节点所有节点的必要文件
#include "std_msgs/String.h"            //"std_msgs/String.h"包含消息类型
#include <sstream>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "example1_a"); //启动该节点并设置其名称（example1_a），该名称是唯一的
    ros::NodeHandle n;                  //设置节点进程的句柄
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("message", 1000);
//把这个节点设置成发布者，并把发布主题的类型告诉节点管理器。第一个参数是消息名称“message”，第二个参数将缓冲区设置为1000个消息

    ros::Rate loop_rate(10);            //设置频率10Hz
```

ROS节点设计

1. 编写程序

//example1_a.cpp: 发送数据

```
    while (ros::ok())                          //一直运行，直到CTRL+C停止运行
    {
        std_msgs::String msg;                  //创建消息变量，变量类型必须符合发送的要求
        std::stringstream ss;
        ss << " I am the example1_a node "; //要发布的消息内容
        msg.data = ss.str();
        chatter_pub.publish(msg);              //发布消息
        ros::spinOnce();                       //如果出现订阅者，ROS会更新和读取所有主题
        loop_rate.sleep();                     //按频率挂起
    }
    return 0;
}
```

```
//example1_b.cpp: 接收数据
#include "ros/ros.h"
#include "std_msgs/String.h"

/*接收消息然后发布*/
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());  //在命令行窗口显示消息内容
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example1_b");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("message", 1000, chatterCallback);
//创建一个订阅者，从主题获取以“message”为名称的消息，缓冲区1000，处理消息句柄的回调函数chatterCallback
    ros::spin();  /ros::spin()库是响应循环，消息到达时调用函数chatterCallback，CTRL+C结束循环
    return 0;
}
```

打开CMakelists.txt:

$ rosed chapter2_tutorials CMakeLists.txt

//rosed 节点名 CMakeLists.txt

在最后加上

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)

//原书这里把节点名写成char2_example1_a/b导致后面编译不成功
//添加可执行文件
//add_executable(节点名 src/节点名.cpp)
add_executable(chap2_example1_a src/example1_a.cpp)
add_executable(chap2_example1_b src/example1_b.cpp)

//添加依赖项
//add_dependencies(节点名 功能包名_generate_messages_cpp)
add_dependencies(chap2_example1_a chapter2_tutorials_generate_messages_cpp)
add_dependencies(chap2_example1_b chapter2_tutorials_generate_messages_cpp)

//链接
//target_link_librariex(节点名 ${catkin_LIBRARIES})
target_link_libraries(chap2_example1_a ${catkin_LIBRARIES})
target_link_libraries(chap2_example1_b ${catkin_LIBRARIES})
```

2. 编译ROS节点

3. 编译功能包

回到上层：

$ cd ~/dev/catkin_ws/

$ catkin_make --pkg chapter2_tutorials

打开roscore，启动ROS

$ roscore

4.在两个不同的窗口运行以下命令：

$ rosrun chapter2_tutorials chap2_example1_a

$ rosrun chapter2_tutorials chap2_example1_b

教材勘误：Chapter2原书部分错误与修正：

P56：

问题：

$ roscd chapter2_tutorials/src/

roscd: No such package/stack 'chapter2_tutorials'

解决：没有将工作空间路径： /home/catkin_ws/devel/setup.bash 添加到 .bashrc中

$ gedit ~/.bashrc

在最后加上：source /home/studyon/dev/catkin_ws/devel/setup.bash

P60：

问题：

$ rosrun chapter2_tutorials example1_a

[rosrun] Couldn't find executable named example1_a below /home/rushaonan/dev/catkin_ws/src/chapter2_tutorials

解决：

$ cd ~/dev/catkin_ws

$ catkin_make --pkg chapter2_tutorials

$ source devel/setup.bash

注：可参考前面的CMakeLists.txt修改。

教材勘误：Chapter2原书部分错误与修正：

P61（P62同）：

修改package.xml文件：

Now, edit package.xml and remove <!-- --> from

the <build_depend>message_generation</build_depend> and <run_depend>message_runtime</run_depend>

P69：

chapter2.launch中：

<node name ="example1_a" pkg="chapter2_tutorials" type="example1_a"/>

改为<node name ="example1_a" pkg="chapter2_tutorials" type="chap2_example1_a"/>

<node name ="example1_b" pkg="chapter2_tutorials" type="example1_b"/>

改为<node name ="example1_b" pkg="chapter2_tutorials" type="chap2_example1_b"/>

教材勘误：Chapter2原书部分错误与修正：

P61（P62同）：

修改package.xml文件：

Now, edit package.xml and remove <!-- --> from

the <build_depend>message_generation</build_depend> and <run_depend>message_runtime</run_depend>

P69：

chapter2.launch中：

<node name ="example1_a" pkg="chapter2_tutorials" type="example1_a"/>

改为<node name ="example1_a" pkg="chapter2_tutorials" type="chap2_example1_a"/>

<node name ="example1_b" pkg="chapter2_tutorials" type="example1_b"/>

改为<node name ="example1_b" pkg="chapter2_tutorials" type="chap2_example1_b"/>

创建msg和srv文件

在节点中创建msg和srv文件，ROS会根据这些文件内容自动地创建所需的代码，以便msg和srv文件能够被节点使用。

创建 msg文件

在chapter2_tutorials功能包下创建msg文件夹，并在其中创建一个新的文件chapter2_msg1.msg，添加以下内容：

```
int32 A
int32 B
int32 C
```

1）编辑package.xml：

从<build_depend>message_generation</build_depend>和<run_depend>message_runtime</run_depend>行删除<！---->

2）编辑CMakeList.txt：

加入message_generation：

```
find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation
 )
```

找到如下行，取消注释并加入新消息名称：

```
## Generate messages in the 'msg' folder
add_message_files(
FILES
chapter2_msg1.msg
)
## Generate added messages and services with any dependencies
listed here
generate_messages(
DEPENDENCIES
std_msgs
)
```

在命令行进行编译：

$ cd ~/dev/catkin_ws/

$ catkin_make

检查编译是否成功，使用rosmsg命令：

$ rosmsg show chapter2_tutorials/chapter2_msg1

在chapter2_msg1.msg文件中看到一样的内容，则编译正确。

创建srv文件

在chapter2_tutorials文件夹下创建名为srv的文件夹，新建文件chapter2_srv1.srv并添加：

```
int32 A
int32 B
int32 C
---
int32 sum
```

为了编译新的msg和srv文件，必须取消在package.xml和CMakeLists.txt中的某些注释，它们包含消息和服务的配置信息，并告诉ROS如何编译。

1）修改package.xml：

从chapter2_tutorials功能包中打开：

$ rosed chapter2_tutorials package.xml

找到下面的代码行并取消注释：

<build_depend>message_generation</build_depend>

<run_depend>message_runtime</run_depend>

2）修改CMakeLists.txt：

$ rosed chapter2_tutorials CMakeLists.txt

找到如下的代码行并取消注释，改为正确数据：

```
catkin_package(
CATKIN_DEPENDS message_runtime
)
```

在find_package部分添加message_generation行：

```
find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation
)
```

在add_message_files的相应位置添加消息和服务文件的名字：

```
## Generate messages in the 'msg' folder
add_message_files(
FILES
chapter2_msg1.msg
)
## Generate services in the 'srv' folder
add_service_files(
FILES
chapter2_srv1.srv
)
```

取消generate_messages部分的注释：

Generate added messages and services with any dependencies listed here

generate_messages(

DEPENDENCIES

std_msgs

)

测试编译是否成功：

$ rossrv show chapter2_tutorials/chapter2_srv1

在chapter2_srv1.srv文件中看到相同的内容，则说明编译正确。

使用新建的srv和msg文件：

对三个整数求和。需要两个节点，一个服务器、一个客户端。

在chapter2_tutorials功能包中，新建两个节点example2_a和example2_b。

在src文件夹下创建两个源代码文件：

1）example2_a.cpp中，添加以下代码：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
bool add(chapter2_tutorials::chapter2_srv1::Request &req,
chapter2_tutorials::chapter2_srv1::Response &res)
{
   res.sum = req.A + req.B + req.C;
   ROS_INFO("request: A=%ld, B=%ld C=%ld", (int)req.A, (int)req.B,(int)req.C);
   ROS_INFO("sending back response: [%ld]", (int)res.sum);
   return true;
}
```

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_3_ints_server");
  ros::NodeHandle n;
  ros::ServiceServer service = n.advertiseService("add_3_ints", add);
  ROS_INFO("Ready to add 3 ints.");
  ros::spin();
  return 0;
}
```

代码解析：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
```

这些行包含必要的头文件和创建的srv文件：

```
bool add(chapter2_tutorials::chapter2_srv1::Request &req,
chapter2_tutorials::chapter2_srv1::Response &res)
```

该函数会对3个变量求和，并将计算结果发送给其他节点：

```
ros::ServiceServer service = n.advertiseService("add_3_ints", add);
```

创建服务并在ROS中发布广播。

2）编辑example2_b.cpp中，添加代码：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
#include <cstdlib>

int main(int argc, char **argv)
{
   ros::init(argc, argv, "add_3_ints_client");
   if (argc != 4)
    {
       ROS_INFO("usage: add_3_ints_client A B C ");
       return 1;
     }

   ros::NodeHandle n;

   ros::ServiceClient client =
                  n.serviceClient<chapter2_tutorials::chapter2_srv1>("add_3_ints");
   chapter2_tutorials::chapter2_srv1 srv;
```

```
 srv.request.A = atoll(argv[1]);
 srv.request.B = atoll(argv[2]);
 srv.request.C = atoll(argv[3]);

if (client.call(srv))
{
   ROS_INFO("Sum: %ld", (long int)srv.response.sum);
}
else
{
   ROS_ERROR("Failed to call service add_3_ints");
   return 1;
}
   return 0;
}
```

代码解析：

```
ros::ServiceClient client = n.serviceClient<chapter2_
tutorials::chapter2_srv1>("add_3_ints");
```

以add_3_ints为名称创建一个服务的客户端。

```
chapter2_tutorials::chapter2_srv1 srv;
```

```
srv.request.A = atoll(argv[1]);
srv.request.B = atoll(argv[2]);
srv.request.C = atoll(argv[3]);
```

下面代码创建srv文件的一个实例，并且加入需要发送的数据值。

```
    if (client.call(srv)
```

这行代码会调用服务并发送数据。

若调用成功，call()函数会返回true；否则，返回false。

为了编译节点，在CMakeList.txt文件中增加以下行：

```
add_executable(chap2_example2_a src/example2_a.cpp)
add_executable(chap2_example2_b src/example2_b.cpp)

add_dependencies(chap2_example2_a chapter2_tutorials_generate_
messages_cpp)
add_dependencies(chap2_example2_b chapter2_tutorials_generate_
messages_cpp)

target_link_libraries(chap2_example2_a ${catkin_LIBRARIES})
target_link_libraries(chap2_example2_b ${catkin_LIBRARIES})
```

现在执行以下命令：

$ cd ~/dev/catkin_ws
$ catkin_make

启动节点，需要执行以下命令行：

$ rosrun chapter2_tutorials chap2_example2_a
$ rosrun chapter2_tutorials chap2_example2_b 1 2 3

显示如下：

Node example2_a

[INFO] [1355256113.014539262]: Ready to add 3 ints.

[INFO] [1355256115.792442091]: request: A=1, B=2 C=3

[INFO] [1355256115.792607196]: sending back response: [6]

Node example2_b

[INFO] [1355256115.794134975]: Sum: 6

用自定义的msg文件来创建节点。

创建example3_a.cpp和example3_b.cpp文件，同时调用chapter2_msg1.msg。

1）example3_a.cpp：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h"
#include <sstream>

int main(int argc, char **argv)
{
   ros::init(argc, argv, "example3_a");

   ros::NodeHandle n;
   ros::Publisher pub =n.advertise<chapter2_tutorials::chapter2_msg1>("message",  1000);
   ros::Rate loop_rate(10);

   while (ros::ok())
   {
     chapter2_tutorials::chapter2_msg1 msg;
     msg.A = 1;
     msg.B = 2;
     msg.C = 3;
     pub.publish(msg);
     ros::spinOnce();
     loop_rate.sleep();
   }
  return 0;
}
```

2）example3_b.cpp：

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h“

void messageCallback(const chapter2_tutorials::chapter2_msg1::ConstPtr& msg)
{
    ROS_INFO("I heard: [%d] [%d] [%d]", msg->A, msg->B, msg->C);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example3_b");

    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("message", 1000, messageCallback);
    ros::spin();

    return 0;
}
```

运行这两个节点，将会看到如下信息：

…

[INFO] [1355270835.920368620]: I heard: [1] [2] [3]

[INFO] [1355270836.020326372]: I heard: [1] [2] [3]

[INFO] [1355270836.120367449]: I heard: [1] [2] [3]

[INFO] [1355270836.220266466]: I heard: [1] [2] [3]

启动文件

- 命令行操作，太繁琐
- launch文件可以同时启动多个节点。
- launch文件格式

```
<?xml version="1.0"?>
<launch>
<node name ="example1_a" pkg="chapter2_tutorials"
type="example1_a"/>
<node name ="example1_b" pkg="chapter2_tutorials"
type="example1_b"/>
</launch>
```

使用roslaunch

(1) roslaunch按照launch文件中的定义来启动nodes

(2) 进入此前的package文件夹

```
$ cd ~/catkin_ws

$ source devel/setup.bash

$ roscd beginner_tutorials
```

(3) 创建launch文件夹

$ mkdir launch
$ cd launch

注意：文件夹并不是一定要名称为launch，甚至不需要这个文件夹，roslaunch会自动搜索package中的launch文件。

(4) roslaunch的用法：

$ roslaunch [package] [filename.launch]

launch文件的写法

(1)创建文件turtlemimic.launch

(2)写入以下内容：

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

(3) launch文件讲解：

<launch>确保文件被识别为launch文件，接下来有两个组：namespace tag是turtlesim1 和 turtlesim2，并且有turtlesim节点的名字，这可以使我们在启用两个simulator时不冲突。

接下来是mimic节点，topics的输入和输出是turtlesim1和turtlesim2，这样可以让turtlesim2模拟turtlesim1。最后一行xml tag代表关闭这个launch file。

launch file:

(1)输入：

$ roslaunch beginner_tutorials turtlemimic.launch

(2)在新的terminal输入：

$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

(3)使用rqt_graph来更好理解launch file做了什么

方法一：

先输入rqt，然后在窗口中选Plugins > Introspection > Node Graph

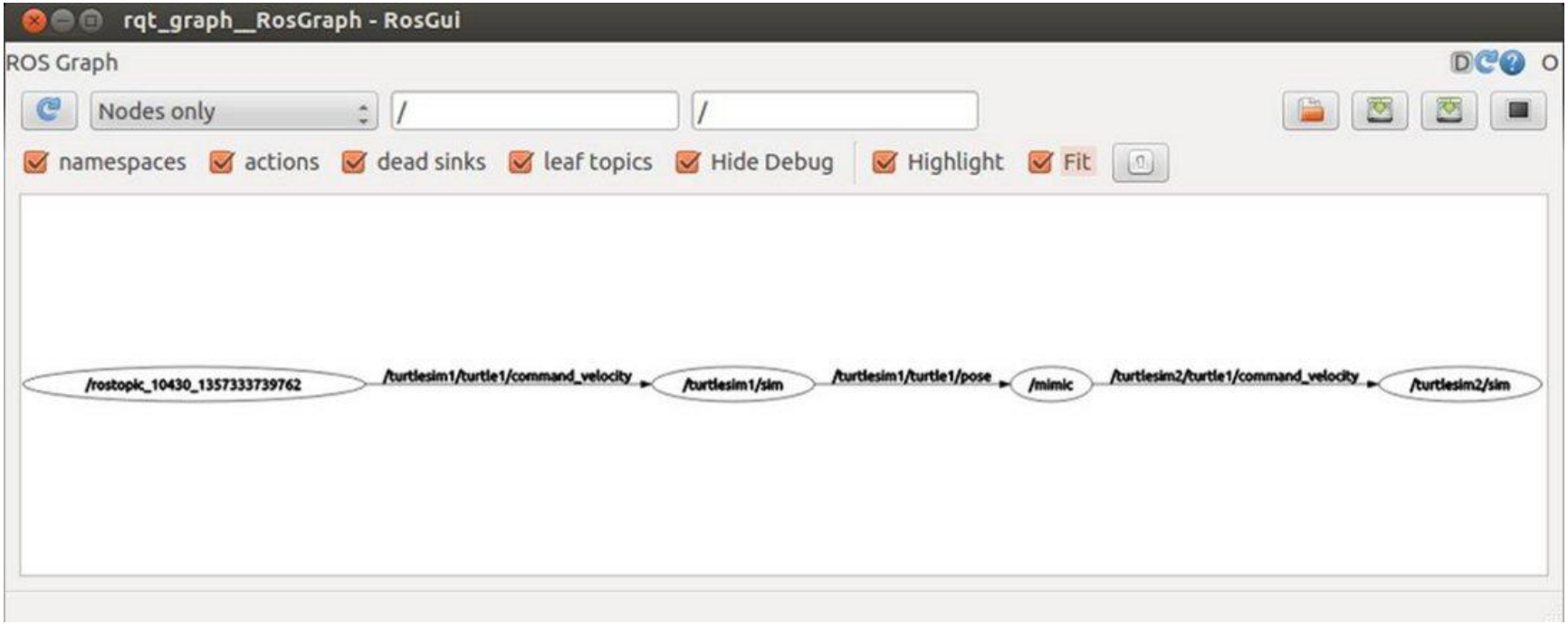
$ rqt

方法二：

直接输入：

$ rqt_graph

效果如下图：


rqt_graph__RosGraph - RosGui
ROS Graph
Nodes only
/
/
namespaces
actions
dead sinks
leaf topics
Hide Debug
Highlight
Fit
/rostopic_10430_1357333739762
/turtlesim1/turtle1/command_velocity
/turtlesim1/sim
/turtlesim1/turtle1/pose
/mimic
/turtlesim2/turtle1/command_velocity
/turtlesim2/sim

ROS动态参数

在节点外部改变参数的方式有：

参数服务器、服务、主题以及动态参数。

在ROS系统中实现某个功能时，时常需要外部动态调整参数。

例如：PID参数调节，或查看机器人在不同参数下的的性能表现。

在ROS中可以通过动态参数功能来实现。

PROJECT_SOURCE_DIR

----所运行的CMakeList.txt所在文件夹下的工程目录

1.创建cfg文件：

roscd chapter2_tutorials; mkdir cfg； vim chapter2.cfg

Cfg文件内容：

```
#!/usr/bin/env python
PACKAGE = “chapter2_tutorials"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

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("int_param",    int_t,    0, "An Integer parameter", 50,  0, 100)
gen.add("bool_param",   bool_t,   0, "A Boolean parameter",  True)
```

Cfg文件内容（续）：

```
size_enum = gen.enum([ gen.const(“Low",  int_t, 0, "Low is 0"),
                       gen.const("Medium",   int_t, 1, “Medium is 1"),
                       gen.const(“High",  int_t, 2, “High is 2"),
                        “Select from the list")

gen.add("size", int_t, 0, " Select from the list ", 1, 0, 3, edit_method =
                size_enum)

exit(gen.generate(PACKAGE, “chapter2_tutorials", “chapter2"))
```

cfg文件的代码解析：

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

from dynamic_reconfigure.parameter_generator_catkin import *
```

初始化ROS，导入dynamic_reconfigure功能包提供的参数生成器（parameter generator）。

```
gen = ParameterGenerator()
#创建一个动态参数生成器，后面可以往该变量中添加动态参数；
```

添加参数：

gen.add(参数名，数据类型，回调函数掩码，参数说明，默认值，最小值，最大值）——

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("int_param", int_t, 0, "An Integer parameter", 50, 0, 100)

gen.add("bool_param", bool_t, 0, "A Boolean parameter", True)

exit(gen.generate(PACKAGE, “chapter2_tutorials", “chapter2"))

——生成必要的文件并退出。

上述cfg中的代码为python代码，需要修改cfg文件的权限：

chmod a+x cfg/chapter2.cfg（或 chmod 777 cfg/chapter2.cfg ）

2. 修改CMakeLists.txt：

```
find_package(catkin REQUIRED COMPONENTS
   roscpp
   std_msgs
   message_generation
   dynamic_reconfigure
)

generate_dynamic_reconfigure_options(cfg/chapter2.cfg)

add_executable(chap2_example4 src/example4.cpp)
add_dependencies(chap2_example4 chapter2_tutorials_gencfg)
target_link_libraries(chap2_example4 ${catkin_LIBRARIES})
```

3. 编写源码：

```
#include "ros/ros.h"
#include <dynamic_reconfigure/server.h>
#include <chapter2_tutorials/chapter2Config.h>

void callback(chapter2_tutorials::chapter2Config &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);
}
```

4. 编译：

修改CMakelist.txt，添加：

```
add_executable(chap2_example4 src/example4.cpp)
add_dependencies(chap2_example4 chapter2_tutorials_gencfg)
target_link_libraries(chap2_example4 ${catkin_LIBRARIES})
```

编译：catkin_make

运行：rosrun chapter2_tutorials chap2_example4

rosrun rqt_reconfigure rqt_reconfige

boost::bind

标准库函数std::bind1st和std::bind2nd的一种泛化形式，可以支持函数对象、函数、函数指针、成员函数指针，并绑定任意参数到某指定值上或将输入参数传入任意位置。例：

```
int f(int a, int b)
{
return a + b;
}
```

可以绑定所有参数，如：

bind(f, 1, 2)　　等价于f(1, 2);

bind(f, 1, 2, 3)　等价于f(1, 2, 3)。

boost::bind

ROS 许多地方的回调函数（call back）都用到了boost::bind。

其中talk.cpp的21行

```
cb = boost::bind(&NodeExample::configCallback,
node_example, _1, _2);
```

其中的 node_example是指针变量NodeExample

```
*node_example = new NodeExample();
boost::bind(&NodeExample::configCallback, node_example, _1,
_2)
```

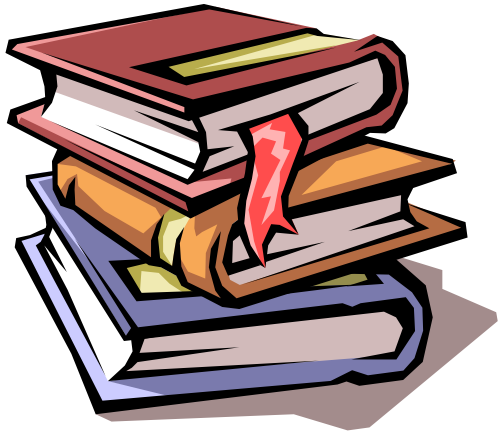
的意思就是 node_example -> configCallback(x, y)

boost::bind

bind(f, _1, 5)(x)is equivalent to f(x, 5); here **_1** is a placeholder argument that means "substitute with the first input argument."

这里的 _1 相当于一个占位符，用来代替第一个输入参数。

```
bind( f, 5, _1)(x);                 // f(5, x)
bind( f, _2, _1)(x, y);             // f(y, x)
bind( g, _1, 9, _1)(x);             // g(x, 9, x)
bind( g, _3, _3, _3)(x, y, z);      // g(z, z, z)
bind( g, _1, _1, _1)(x, y, z);      // g(x, x, x)
```



Q & A
Thanks !