ROS基本元素的使用

朱笑笑

2018年3月

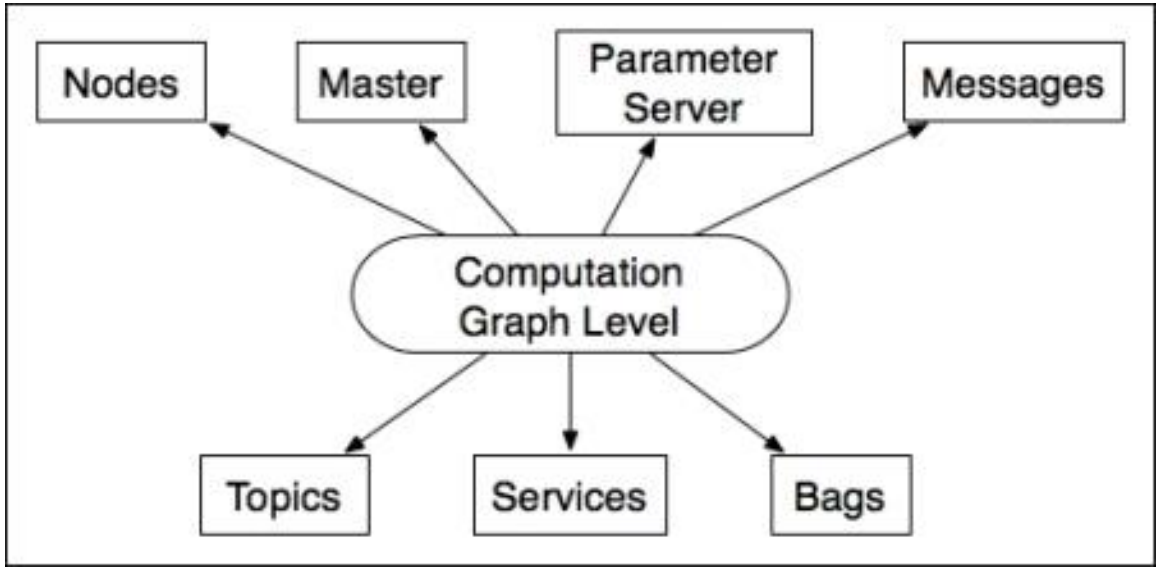
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

目录 Contents

ROS计算图结构

- 如上节所述ROS具有计算图结构。
- 包括节点（Node）、主控（Master）、参数服务器（Parameter Server）、消息（Messages）、主题（Topics）、服务（Service）、消息包（Bag）



Node的使用

- 节点都是各自独立的可执行文件，能够通过主题、服务或参数服务器与其他进程（节点）通信。ROS通过使用节点将代码和功能解耦，提高了系统容错能力和可维护性，使系统简化。
- 同时，节点允许了ROS 系统能够布置在任意多个机器上并同时运行。节点在系统中必须有唯一的名称。节点使用特定名称与其他节点进行通信而不产生歧义。节点可以使用不同的库进行编写．如roscpp 和rospy。roscpp 基于C++，而rospy 基于Python 。在本课程，我们将使用roscpp。

Node的使用

- 例如，咱们有一个机器人，和一个遥控器，那么这个机器人和遥控器开始工作后，就是两个节点。遥控器起到了下达指 令的作用；机器人负责监听遥控器下达的指令，完成相应动作。从这里我们可以看出，节点是一个能执行特定工作任 务的工作单元，并且能够相互通信，从而实现一个机器人系统整体的功能。在这里我们把遥控器和机器人简单定义为两个节点，实际上在机器人中根据控制器、传感器、执行机构等不同组成模块，还可以将其进一步细分为更多的节点，这个是根据用户编写的程序来定义的。）

Node的使用

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");  //初始化 node
  ros::NodeHandle n;                //node的句柄
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(1);     //控制程序周期的，帮助功能
  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;
}
```

Node的使用

- 调用格式void ros::init (<command line or remapping arguments>, std::string node_name, uint32_t options);
  - ros::init (argc, argv, "my_node_name");
  - ros::init (argc, argv, "my_node_name", ros::init_options::AnonymousName);
- 初始化选项Initialization Options
  - ros::init_options::NoSigintHandler
  - ros::init_options::AnonymousName
  - ros::init_options::NoRosout
- 节点的启动
  - ros::NodeHandle nh;
  - ros::start()

Node的使用

- 节点的关闭
  - ros::shutdown()
  - 1，检查节点是否关闭

```
while (ros::ok())
{
  ...
}
```

  - 2，自定义信号处理函数

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

void mySigintHandler(int sig)
{
  // Do some custom action.
  // For example, publish a stop message to some other nodes.

  // All the default sigint handler does is call shutdown()
  ros::shutdown();
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "my_node_name", ros::init_options::NoSigintHandler);
  ros::NodeHandle nh;

  // Override the default ros sigint handler.
  // This must be set after the first NodeHandle is created.
  signal(SIGINT, mySigintHandler);

  //...
  ros::spin();
  return 0;
}
```

Node的使用

- 如上节课提到的，node可由rosrun或者roslaunch启动。
- Rosnode工具是一个node相关的命令行工具：
  - rosnode info node 输出当前节点信息。
  - rosnode kill node 结束当前运行节点进程或发送给定信号。
  - rosnode list 列出当前活动节点。
  - rosnode machine hostname 列出某一特定计算机上运行的节点或列出主机名称。
  - rosnode pi 口q node 测试节点间的连通性。
  - rosnode cleanup 将无法访问节点的注册信息清除。

目录 Contents

Topic的使用

- topic是节点间用来传输数据的总线。通过主题进行消息路由不需要节点之间直接手动连接。这就意味着发布者和订阅者之间不需要知道彼此是否存在。同一个主题也可以有很多个订阅者。
- 每个主题都是强类型的，发布到主题上的消息必须与主题的ROS 消息类型相匹配。

Topic的使用

- 消息
  - ROS 使用了一种简化的消息类型描述语言来描述ROS 节点发布的数据值。通过这样的描述语言， ROS 能够使用多种编程语言生成不同类型消息的源代码。
  - ROS 提供了很多**预定义消息类型**。如果你创建了一种新的消息类型，那么就要把消息的类型定义放到功能包的msg / 文件夹下。在该文件夹中，有用于定义各种消息的文件。这些文件都以.msg 为扩展名。
  - 消息类型必须具有两个主要部分：字段和常量。字段定义了要在消息中传输的数据的类型，例如int32 、float32 、string 或之前创建的自定义类型，如叫做typel 和type2 的新类型。常量用于定义字段的名称。

Topic的使用

- 消息（message）

基本类型	串行化	C++	Python
bool	Unsigned 8-bit int	uint8_t	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ASCII string (4-bit)	std::string	string
time	Secs/nsecs signed 32- bit ints	ros::Time	rospy.Time
duration	Secs/nsecs signed 32- bit ints	ros::Duration	rospy.Duration

- variable-length array[] and fixed-length array[C]

Topic的使用

- 消息
  - ROS 消息中的一种特殊数据类型是报文头(header)，主要用于添加时间戳、坐标位置等。报文头还允许对消息进行编号。通过在报文头内部附加信息，我们可以知道是哪个节点发出的消息，或者可以添加一些能够被ROS处理的其他功能。
  - 报文头类型包含以下字段：
    - uint32 seq
    - time stamp
    - string frame_id

Topic的使用

- 消息
  - 在一个msg文件夹下面进行定义，文件名称为“消息类型名称 .msg”
    - 如 msg/Num.msg　内容： int64 num
    - 类似一个结构体

```
string first_name
string last_name
uint8 age
uint32 score
```

  - 编译时将CMakeList.txt中的

```
# Do not just add this to your CMakeLists
n before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

```
add_message_files(
  FILES
  Num.msg
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

  - package.xml

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

Topic的使用

- 消息
  - 消息的类型在ROS 中按照以下标准命名方式进行约定：功能包名称I .m sg 文件名称。例如std_msgs/msg/String . msg 的消息类型是std_msgs/String 。
  - ROS 使用命令行工具rosmsg 来获取有关消息的信息。惯用参数如下所示：
    - rosmsg show 显示一条消息的宇段。
    - rosmsg list 列出所有消息。
    - rosmsg package 列出功能包的所有消息。
    - rosmsg packages 列出所有具有该消息的功能包。
    - rosmsg users 搜索使用该消息类型的代码文件。
    - rosmsg md5 显示一条消息的MD5 求和结果。

Topic的使用

- 发布者
  - 通过NodeHandle::advertise()创建一个发布者，并向ros master进行注册

```
ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
```

名称　发送队列大小

  - 通过publish()函数发送，参数类型需要和创建的时候一致

```
std_msgs::String msg; msg.data = “message”;
chatter_pub.publish(msg);
```

```
Num msg; msg.num = 4;
chatter_pub.publish(msg);
```

Topic的使用

- 接收者
  - 利用ros::subscribe()创建一个接收者。

```
ros::Subscriber sub = node.subscribe("chatter", 1000, messageCallback);
```

名称　接收队列大小　回调函数

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

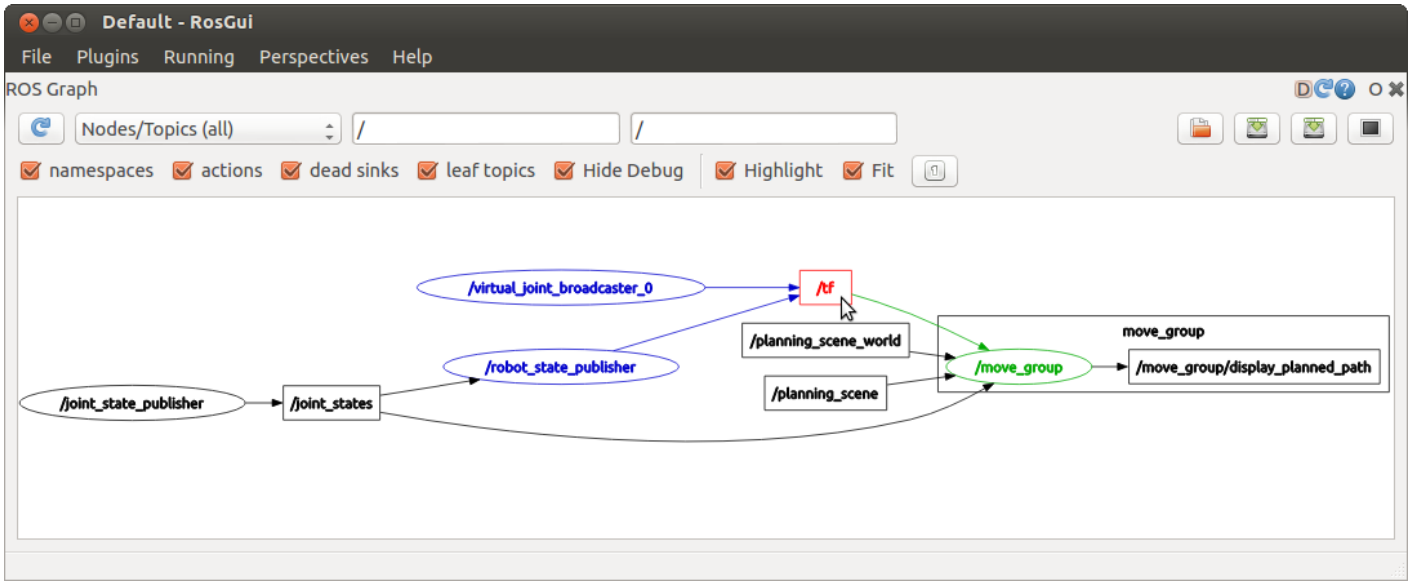
  - 可以有多个接收者
  - 收到主题后，回调函数回进入队列等待需要调用ros::spin()和ros::spinonce() 处理等待

Topic的使用

- Rostopic可以进行交互并获取主题的信息
  - 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。

Topic的使用

- rqt_graph 。提供了一个可视化ROS计算图的GUI插件。它的组件是通用的，所以你想实现图形表示的其他包可以依赖于这个pkg（使用rqt_dep来找出依赖的pkgs，rqt_dep本身也依赖于rqt_graph）。



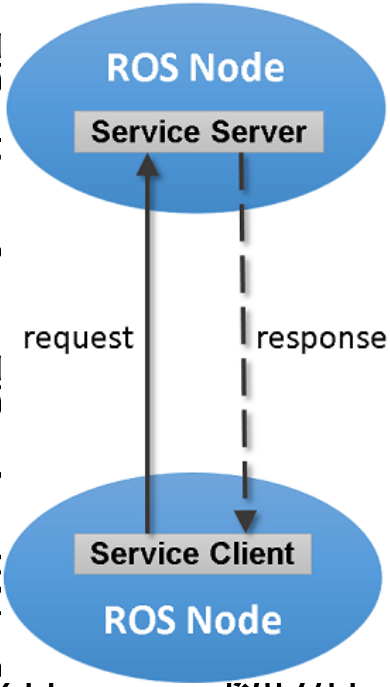
目录 Contents

service的使用

- 当你需要直接与节点通信并获得应答时，将无法通过主题实现，而需要使用服务。服务需要由用户开发，节点并不提供标准服务。包含消息源代码的文件存储在srv 文件夹中。
- 像主题一样，服务关联一个以功能包中.srv 文件名称来命名的服务类型。与其他基于RO S 文件系统的类型一样，服务类型是功能包名称和.srv 文件名称的组合。例如，chapter2_tutorials/srv/chapter2_srv1. srv 文件的服务类型是chapter2_tutorials/chapter2_srvl。

service的使用

- 当你需要直接与节点通信并获得应答时，将无法通过主[illegible]由用户开发，[illegible]代码的文件存[illegible]



Service Name:	**/example_service**
Service Type:	**roscpp_tutorials/TwoInts**
Request Type:	**roscpp_tutorials/TwoIntsRequest**
Response Type:	**roscpp_tutorials/TwoIntsResponse**

- 像主[illegible]/文件名称来[illegible]件系统的类[illegible]/文件名称的组合。例如，chapter2_tutorials/srv/chapter2_srv1. srv 文件的服务类型是chapter2_tutorials/chapter2_srvl。

service的使用

- Srv
  - ROS 使用了一种简化的服务描述语言来描述ROS 的服务类型。这直接借鉴了ROS 消息的数据格式，以实现节点之间的请求 / 响应通信。服务的描述存储在功能包的srv / 子目录下.srv 文件中。
  - 若要调用服务，你需要使用该功能包的名称及服务名称。例如，对于sample_packagel/srv/sample1.srv 文件，可以将它称为sample_packagel/sample1服务。

Srv →

```
int64 a
int64 b
---
int64 sum
```

srv.request.a

srv.response.sum

service的使用

- Srv
  - 编译时将CMakeList.txt中的

```
# Do not just add this to your CMakeLists
n before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

  - package.xml

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

service的使用

- ROS 可以通过rossrv看到有关服务数据结构的信息，并且与rosmsg 具有完全一致的用法。
  - rossrv show 显示一条srv的字段。
  - rossrv list 列出所有srv。
  - rossrv package 列出功能包的所有srv。
  - rossrv packages 列出所有具有该srv的功能包。
  - rossrv md5 显示一条srv的MD5 求和结果。

service的使用

- 提供者

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

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

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

service的使用

- 调用者

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

Service 的调用是阻塞式的 succeeded, call() 返回值表示调用是否成功

service的使用

- ROS 通过rosservice可以列出服务列表和查询某个服务。支持的命令如下所示：
  - 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 。

目录 Contents

Parameter Server的使用

- ROS参数服务器能保存数据类型包括：strings, integers, floats, booleans, lists, dictionaries, iso8601 dates, and base64-encoded data。Dictionaries则必需有字符串key值。
- roscpp参数API能支持全部类型，多数情况容易使用的类型有：strings, integers, floats and booleans，使用其他类型参考XmlRpc::XmlRpcValue class
- roscpp有两个版本的API接口：bare版和handle版。
- bare版：在 ros::param 命令空间下。
- handle版：通过ros::NodeHandle接口使用。

Parameter Server的使用

- **获取参数**
- 从参数服务器获取值，每个版本都支持strings, integers, doubles, booleans 和XmlRpc::XmlRpcValue
- 返回 false代表参数不存在或不是正确的类型，同样有版本是返回默认值。
- ros::NodeHandle::getParam()
- 代码示例：

Parameter Server的使用

- **获取参数**

```
ros::NodeHandle nh;
std::string global_name, relative_name, default_param;
if (nh.getParam("/global_name", global_name))
{
  ...
}
if (nh.getParam("relative_name", relative_name))
{
...
}
// Default value version
nh.param<std::string>("default_param", default_param, "default_value");
```

Parameter Server的使用

- **参数缓存**
- ros::NodeHandle::getParamCached()能提供本地的缓存功能。
- **设置参数**
  - ros::NodeHandle::setParam()

```
ros::NodeHandle nh;
nh.setParam("/global_param", 5);
nh.setParam("relative_param", "my_string");
nh.setParam("bool_param", false);
```

Parameter Server的使用

- **检查参数是否存在**
  - ros::NodeHandle::hasParam()

```
ros::NodeHandle nh;
if (nh.hasParam("my_param"))
{
  ...
}
```

- **删除参数**
  - ros::NodeHandle::deleteParam()

```
ros::NodeHandle nh;
nh.deleteParam("my_param");
```

Parameter Server的使用

- **访问私有参数**

```
ros::NodeHandle nh("~");
std::string param;
nh.getParam("private_name", param);
```

- **搜索参数**

```
std::string key;
if (nh.searchParam("bar", key))
{
  std::string val;
  nh.getParam(key, val);
}
```

Parameter Server的使用

- **列表参数**

```
ros::NodeHandle nh("~");
std::string param;
nh.getParam("private_name", param);
```

  - **你可以获取或设置**lists、dictionaries**和**strings**作为**std::vector **和**std::map **容器的模板值**
  - **这些模板值类型包括：**bool、int、float、double、string

- **获取或设置方法：**
  - ros::NodeHandle::getParam / ros::NodeHandle::setParam

Parameter Server的使用

- **列表参数**

```
// Create a ROS node handle
ros::NodeHandle nh;
// Construct a map of strings
std::map<std::string,std::string> map_s, map_s2;
map_s["a"] = "foo";
map_s["b"] = "bar";
map_s["c"] = "baz";
// Set and get a map of strings
nh.setParam("my_string_map", map_s);
nh.getParam("my_string_map", map_s2);
// Sum a list of doubles from the parameter server
std::vector<double> my_double_list;
double sum = 0;
nh.getParam("my_double_list", my_double_list);
for(unsigned i=0; i < my_double_list.size(); i++) {
  sum += my_double_list[i];
}
```

Parameter Server的使用

- 参数服务器用于存储所有节点均可访问的共享数据。ROS 中用来管理参数服务器的工具称为rosparam 。接受的参数如下所示：
  - rosparam set parameter value 设置参数值。
  - rosparam get parameter 获取参数值。
  - rosparam load file 加载参数文件到参数服务器。
  - rosparam delete parameter 删除参数。
  - rosparam dump file 将参数服务器保存到一个文件。
  - rosparam list 列出了服务器中的所有参数。

Parameter Server的使用

- Launch中读取文件中的参数

config.yaml

```
camera:
  left:
    name: left_camera
    exposure: 1
  right:
    name: right_camera
    exposure: 1.1
```

package.launch

```
<launch>
  <node name="name" pkg="package" type="node_type">
      <rosparam command="load"
                file="$(find package)/config/config.yaml" />
  </node>
</launch>
```

面向对象的编程

my_package_node.cpp

```
#include <ros/ros.h>
#include "my_package/MyPackage.hpp"
int main(int argc, char** argv)
{
  ros::init(argc, argv, "my_package");
  ros::NodeHandle nodeHandle("~");

  my_package::MyPackage myPackage(nodeHandle);

  ros::spin();
  return 0;
}
```

MyPackage.hpp

MyPackage.cpp

`class MyPackage`

Main node class providing ROS interface (subscribers, parameters, timers etc.)

Algorithm.hpp

Algorithm.cpp

`class Algorithm`

Class implementing the algorithmic part of the node

Note: The algorithmic part of the code could be separated in a (ROS-independent) library

! Specify a function handler to a method from within the class as

```
subscriber_ = nodeHandle_.subscribe(topic, queue_size,
&ClassName::methodName, this);
```

More info
http://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks

上机实践

- 1，在上节课基础上创建，自己创建一个msg类型，创建一个新的Topic，在发送和接收端实现。
- 2，修改topic的频率，用rostopic进行观察。rostopic发送一个数据。//rostopic
- 3，用rosmsg查看turtle_的发送类型，编写一个发送vel_cmd的node可以实现turtle的控制，比如走个圆，正方形等等。
- 4，创建一个srv类型，实现两个节点间的相互调用。用ros
- 5，写一个配置文件，用launch文件载入到参数服务器中，并在程序中读取参数，打印出来。

谢谢！