程序的调试与可视化

朱笑笑

2018年3月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

ROS基本开发程序过程

创建 → **编译** → **运行**

创建工作空间

创建综合包

创建包

编写代码

修改package.xml

修改Cmalkelist.txt

编译

运行roscore

运行rosrun

或

建立launch文件

roslaunch

ROS基本开发程序过程

创建

创建工作空间

1，mkdir -p ~/catkin_ws/src　　//创建一个文件夹
2，cd ~/catkin_ws/
3，catkin_make　　//利用catkin_make来生成工作空间文件
4，echo “source /home/youruser/catkin_ws/devel/setup.bash” >> ~/.bashrc
　　//也可以直接编辑~/.bashrc文件

创建综合包

综合包即文件夹，在src文件夹下 mkdir metapackagename

创建包

catkin_create_pkg <package_name> [depend1] [depend2]
例： catkin_create_pkg beginner_tutorials std_msgs roscpp

编写代码

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");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(1);
  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;
}
```

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;
}
```

ROS基本开发程序过程

package.xml是对其它**包**或者库**的依赖项**说明文件。主要根据需要进行修改：
build_depend //编译时的依赖
exec_depend//执行时的依赖

编译

修改package.xml

修改CMakelist.txt

CMakeList.txt是编译规则的设置文件。
需要修改内容：
1、项目输出项
2、项目依赖项
3、项目链接项
4、消息、服务相关项

编译

在工作空间目录下
catkin_make

ROS基本开发程序过程

```
# %Tag(FULLTEXT)%
cmake_minimum_required(VERSION 2.8.3)
project(topic_service)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs message_generation)

## Declare ROS messages and services
add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package(CATKIN_DEPENDS message_runtime)

## Build talker and listener
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker3 src/talker.cpp)
target_link_libraries(talker3 ${catkin_LIBRARIES})
add_dependencies(talker3 ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

add_executable(listener3 src/listener.cpp)
target_link_libraries(listener3 ${catkin_LIBRARIES})
add_dependencies(listener3 ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

是编译
件。
项
项
项
务相关

自身的编译输出项　　find_package的编译的项

ROS基本开发程序过程

打开Master

运行roscore

rosrun packagename nodetype __parameter:=value
例如：rosrun my_package my_node __ns:=value

运行rosrun

或

Launch文件相当于一个脚本，记录了需要执行的命令
例：

```
<?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 -->
</launch> <!-- 根元素结束 -->
```

建立launch文件

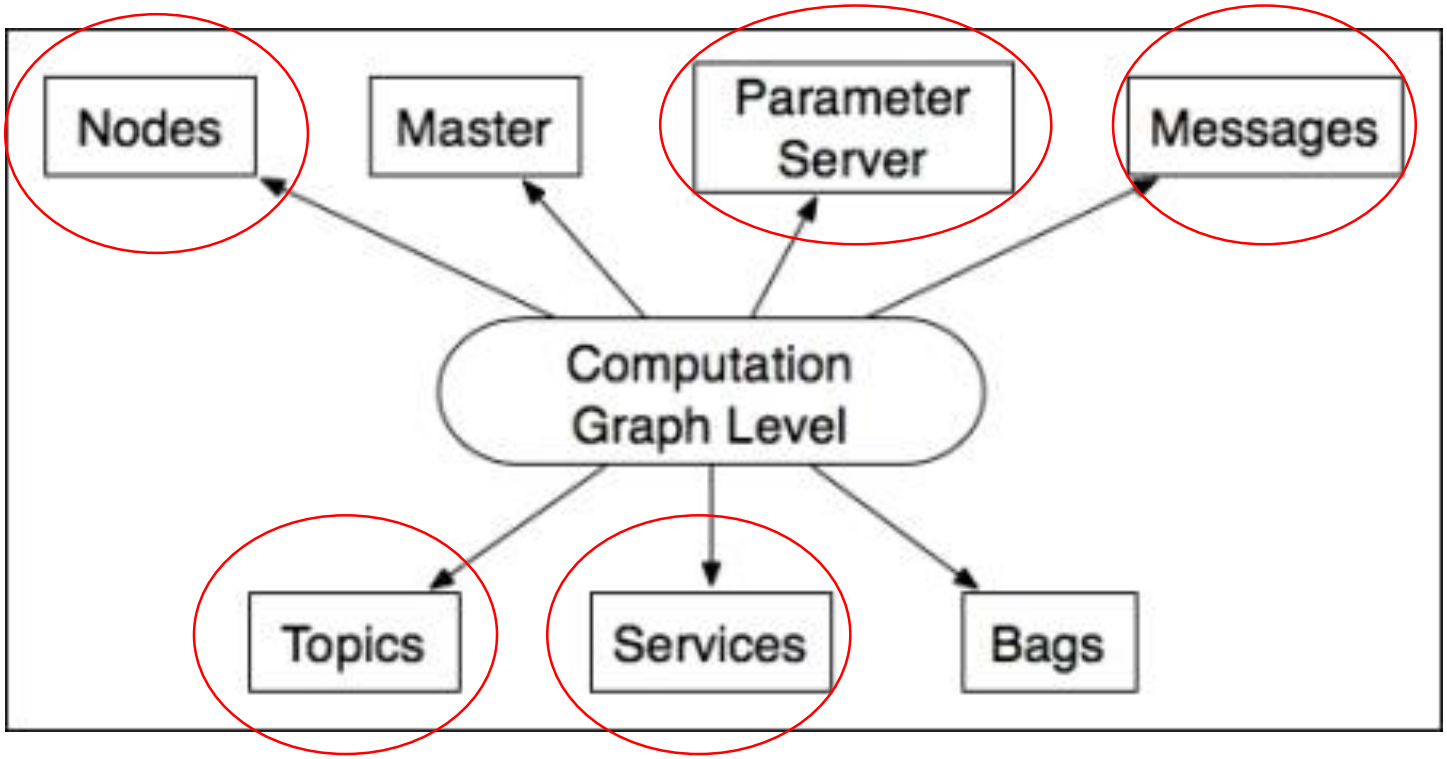
roslaunch

$ roslaunch package_name file.launch 或者
$ roslaunch path/file.launch　　//也可以带参数

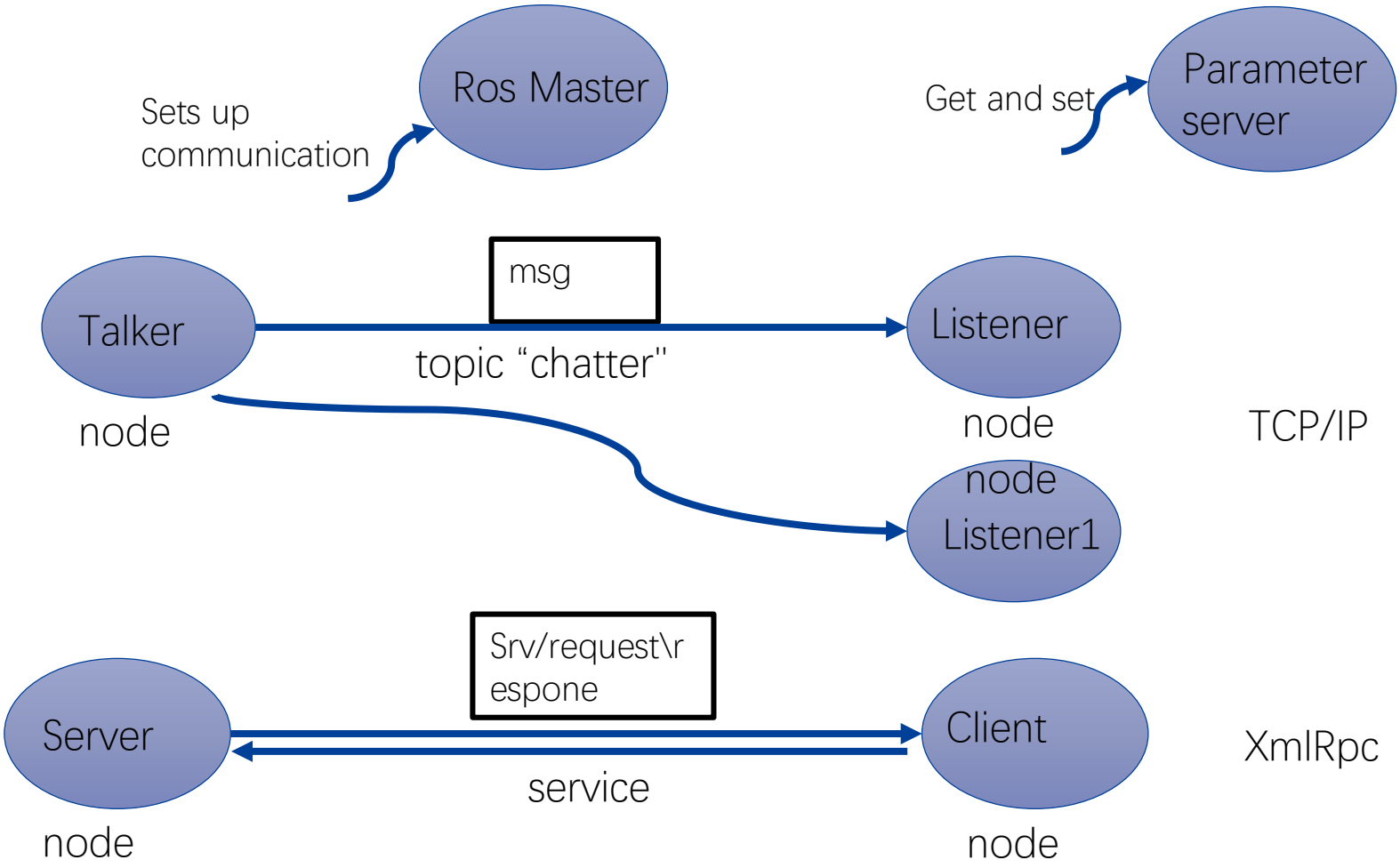
上机实践

- 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。

ROS计算图结构


Nodes
Master
Parameter Server
Messages
Computation Graph Level
Topics
Services
Bags

ROS计算图结构


Ros Master
Sets up
communication
Parameter
server
Get and set
msg
Talker
topic “chatter"
Listener
node
node
TCP/IP
node
Listener1
Srv/request\r
espone
Server
Client
service
XmlRpc
node
node

ROS程序开发

- ROS程序包含了，node相关程序，以及topic、service、parameter server等相关程序

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");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(1);
  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;
}
```

头文件

Node 相关

Topic 相关

一个无限循环直到在终端按下ctrl+c

Topic 相关

ROS程序开发

node

初始化

句柄定义

启动和关闭

帮助函数

topic

发布/提供

定义发布变量

初始化msg/srv变量

发布topic

msg

Srv/request\r
espone

定义msg/srv文件

编译msg/srv

service

接收/调用

定义回调函数

定义接收变量

等待topic/或者
请求service

parameter

获取/设定

检查参数是否存在

搜索参数

从文件载入参数

rosnode

Rosmsg、rostopic、 rossrv、rosservice

rosparam

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rostopic,rosmsg查看turtlesim(仿真小乌龟)的发送类型。1）利用rostopic直接发送topic，2）编写一个发送相应topic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。

程序的调试与可视化

- 调试与可视化功能对于系统开发非常重要，是缩短开发周期的关键因素之一。
- ROS 软件框架附带了大量功能强大的工具，帮助用户和开发人员检测软硬件问题并完成代码调试。这包括了调试工具（如日志消息记录与展示工具）、数据可视化工具和系统监测组件。这些工具使得用户轻松地以自己喜欢的方式查看系统运行状态。

目录 Contents

目录 Contents

调试信息

- C语言 printf、C++的cout，cerr
- 日志库log4j
- ROS基于log4cxx开发了日志记录API 我们有不同层级的调试信息输出，每条信息都有自己的名称，并根据相应条件输出消息，它们对性能没有任何影响并与ROS 框架中的其他工具完全集成。
- 此外，它们无缝集成在并发执行的节点上，也就是说信息没有被打散，而是完全可以根据它们的时间戳进行交叉展示。

调试信息

- ROS中roscpp中包含了rosconsole包提供的客户端API。该API以一些ros_macros的形式
  - #include <ros/console.h>
- rosconsole提供的日志语句5种不同的详细级别，2种格式，8种类型。4种输出地。
- ROS_<LEVEL>[_STREAM]_COND[_NAMED]

5种级别	2种格式	8种类型	4种输出地
DEBUG 、INFO 、WARN 、ERROR 和 FATAL	C语言Printf类型 C++的Stream类型	基本、带名字、带条件、带名字和条件、单次、定时、延时、自定义滤波器	stdout、stderr、Node log file、/rosout topic

调试信息——5个级别

- 调试信息带有5个级别，DEBUG 、INFO 、WARN 、ERROR 和FATAL 。
  - #include <ros/console.h>
  - ROS_INFO (” My INFO message with argument : %f”, val) ;
- 可以为特定节点配置调试信息级别，调试信息的级别高于该节点的级别才会被记录。
- 参考example3

调试信息——2种格式

- 类似于C 语言中的printf 函数的方式：
  - const double val = 3 . 14;
  - ROS_INFO (” My INFO message with argument : %f”, val) ;
- 类似于C++STL 流支持＊ STREAM 函数。 ：
  - ROS_INFO_STREAM(”My INFO stream message with argument : ”<<val);
  - 它一般默认为cout或cerr 。调试信息级别会决定具体是哪→个。我们将在下一节中做详细介绍。
- 参考example1

调试信息——8种类型

- 基本、带名字、带条件、带名字和条件、单次、定时、延时、自定义滤波器
  - 基本型
    - ROS_DEBUG< _STREAM >(args);
  - 带名字
    - ROS_DEBUG < _STREAM >_NAMED(name, args);
  - 带条件
    - ROS_DEBUG < _STREAM >_COND(cond, args);
  - 带名字和条件
    - ROS_DEBUG < _STREAM >_COND_NAMED(cond, args);

调试信息——8种类型

- 基本、带名字、带条件、带名字和条件、单次、定时、延时、自定义滤波器
  - 单次
    - ROS_DEBUG<_STREAM>_ONCE <_NAMED>(<name,>args);
  - 定时
    - ROS_DEBUG<_STREAM>_THROTTLE<_NAMED>(period,< name,> args)
  - 延时
    - ROS_DEBUG<_STREAM>_DELAYED_THROTTLE<_NAMED>(period, <name,> args)
  - 自定义滤波器
    - ROS_DEBUG<_STREAM>_FILTER<_NAMED>(filter,< name,> args)
- 参考example2

调试信息——8种类型

（1）基本

- ROS_DEBUG(...)
- ROS_DEBUG_STREAM(args)

- 基本版本只打印输出消息，即
  - #include <ros/console.h>
  - ROS_DEBUG("Hello %s", "World");
  - ROS_DEBUG_STREAM("Hello " << "World");
- 基础版本输出到名叫“ros.<your_package_name>”的记录中。

调试信息——8种类型

（2）NAMED指定版

- ROS_DEBUG_NAMED(name, ...)
- ROS_DEBUG_STREAM_NAMED(name, args)

- NAMED版本输出到日志，这允许您根据其名称配置启用/禁用的不同日志记录语句。例如:
  - #include <ros/console.h>
  - ROS_DEBUG_NAMED("test_only", "Hello %s", "World");
  - ROS_DEBUG_STREAM_NAMED("test_only", "Hello " << "World");
- 它会输出到名叫"ros.<your_package_name>.test_only"的记录者。
- 有关此信息的更多信息可在配置文件节中获得
- 注意：不要使用可变值的变量作为名称。
- 每个Named的日志存储在一个静态变量，使用宏来初始化。

调试信息——8种类型

(3) Conditional条件版

- ROS_DEBUG_COND(cond, ...)
- ROS_DEBUG_STREAM_COND(cond, args)

- conditional版本当条件为真是否就会输出日志信息。
  - #include <ros/console.h>
  - ROS_DEBUG_COND(x < 0, "Uh oh, x = %d, this is bad", x);
  - ROS_DEBUG_STREAM_COND(x < 0, "Uh oh, x = " << x << ", this is bad");

调试信息——8种类型

(4) Conditional Named指定条件版

- ROS_DEBUG_COND_NAMED(cond, name, ...)
- ROS_DEBUG_STREAM_COND_NAMED(cond, name, args)

- 指定条件版结合上面两种类型：
  - #include <ros/console.h>
  - ROS_DEBUG_COND_NAMED(x < 0, "test_only", "Uh oh, x = %d, this is bad", x);
  - ROS_DEBUG_STREAM_COND_NAMED(x < 0, "test_only", "Uh oh, x = " << x << ", this is bad");

调试信息——8种类型

(5) Once [1.1+]

- ROS_DEBUG_ONCE(...)
- ROS_DEBUG_STREAM_ONCE(args)
- ROS_DEBUG_ONCE_NAMED(name, ...)
- ROS_DEBUG_STREAM_ONCE_NAMED(name, args)

- 这些宏定义会激活时只输出一次
  - #include <ros/console.h>
  - for (int i = 0; i < 10; ++i)
  - {
  - ROS_DEBUG_ONCE("This message will only print once");
  - }

调试信息——8种类型

(6) Throttle [1.1+]

- ROS_DEBUG_THROTTLE(period, ...)
- ROS_DEBUG_STREAM_THROTTLE(period, args)
- ROS_DEBUG_THROTTLE_NAMED(period, name, ...)
- ROS_DEBUG_STREAM_THROTTLE_NAMED(period, name, args)

- 这些宏定义会定期输出日志信息
  - while (true)
  - {
  - ROS_DEBUG_THROTTLE(60, "This message will print every 60 seconds");
  - }

调试信息——8种类型

(7) Delayed throttle (added in Indigo as of rosconsole version 1.11.11)

- ROS_DEBUG_DELAYED_THROTTLE(period, ...)
- ROS_DEBUG_STREAM_DELAYED_THROTTLE(period, args)
- ROS_DEBUG_DELAYED_THROTTLE_NAMED(period, name, ...)
- ROS_DEBUG_STREAM_DELAYED_THROTTLE_NAMED(period, name, args)

- 这些宏会按一定间隔延迟发送日志
  - while (!ros::service::waitForService("add_two_ints", ros::Duration(0.1)) && ros::ok())
  - {
  - // This message will print every 10 seconds.
  - // The macro will have no effect the first 10 seconds.
  - // In other words, if the service is not available, the message will be
  - // printed at times 10, 20, 30, ...
  - ROS_DEBUG_DELAYED_THROTTLE(10, "Waiting for service 'add_two_ints'");
  - }

调试信息——8种类型

(8) Filter [1.1+]

- ROS_DEBUG_FILTER(filter, ...)
- ROS_DEBUG_STREAM_FILTER(filter, args)
- ROS_DEBUG_FILTER_NAMED(filter, name, ...)
- ROS_DEBUG_STREAM_FILTER_NAMED(filter, name, args)

- 过滤输出允许你使用自定义的过滤器，这些过滤器扩展自ros::console::FilterBase类，过滤器必需是指针类型。

调试信息——4种输出

根据不同级别，有四种潜在输出的日志的地方

(1) stdout

- 激活后，DEBUG和INFO消息输出到stdout。
- 注意：这可能不会发送的屏幕，依赖在roslaunch/XML/node输出参数。

(2) stderr

- 激活后，WARN, ERROR 和 FATAL 消息输出到stderr。

(3) Node log file， 节点日志文件

- 节点的所有内容都会记录的日志文件，在不改写ROS_HOME和ROS_LOG_DIR环境变量情况下，节点文件位于~/.ros/log。
- 如果使用roslaunch，你可以用roslaunch-logs来设置日志目录。

(4) /rosout topic

- 所有内容可以输出到/rosout 话题。
- 注意：节点日志要完全启动消息才会发送，因此你可能不能看到初始化的信息。

调试信息

Debug	Info	Warn	Error	Fatal	
stdout	X	X			
stderr			X	X	X
log file	X	X	X	X	X
/rosout	X	X	X	X	X

注意，这个表不同于rospy

调试信息——设置日志级别

- 有三种方法设置日志级别：
  - 第一个是通过配置文件设置所有节点的日志级别
  - 第二个是在运行时，通过rqt_logger_level或rqt_console工具来设置日志级别
  - 第三个通过 log4cxx API来设置：

```
#include <log4cxx/logger.h>
#define OVERRIDE_NODE_VERBOSITY_LEVEL 1

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

  ros::init( argc, argv, "example1" );

#if OVERRIDE_NODE_VERBOSITY_LEVEL
  // Se the logging level manually to DEBUG
  ROSCONSOLE_AUTOINIT;
  log4cxx::LoggerPtr my_logger =
    log4cxx::Logger::getLogger( ROSCONSOLE_DEFAULT_NAME );
  my_logger->setLevel(
    ros::console::g_level_lookup[ros::console::levels::Debug]
  );
#endif
```

调试信息——设置日志级别

- 有三种方法设置日志级别：example1.launch
  - 第一个是通过配置文件设置所有节点的日志级别

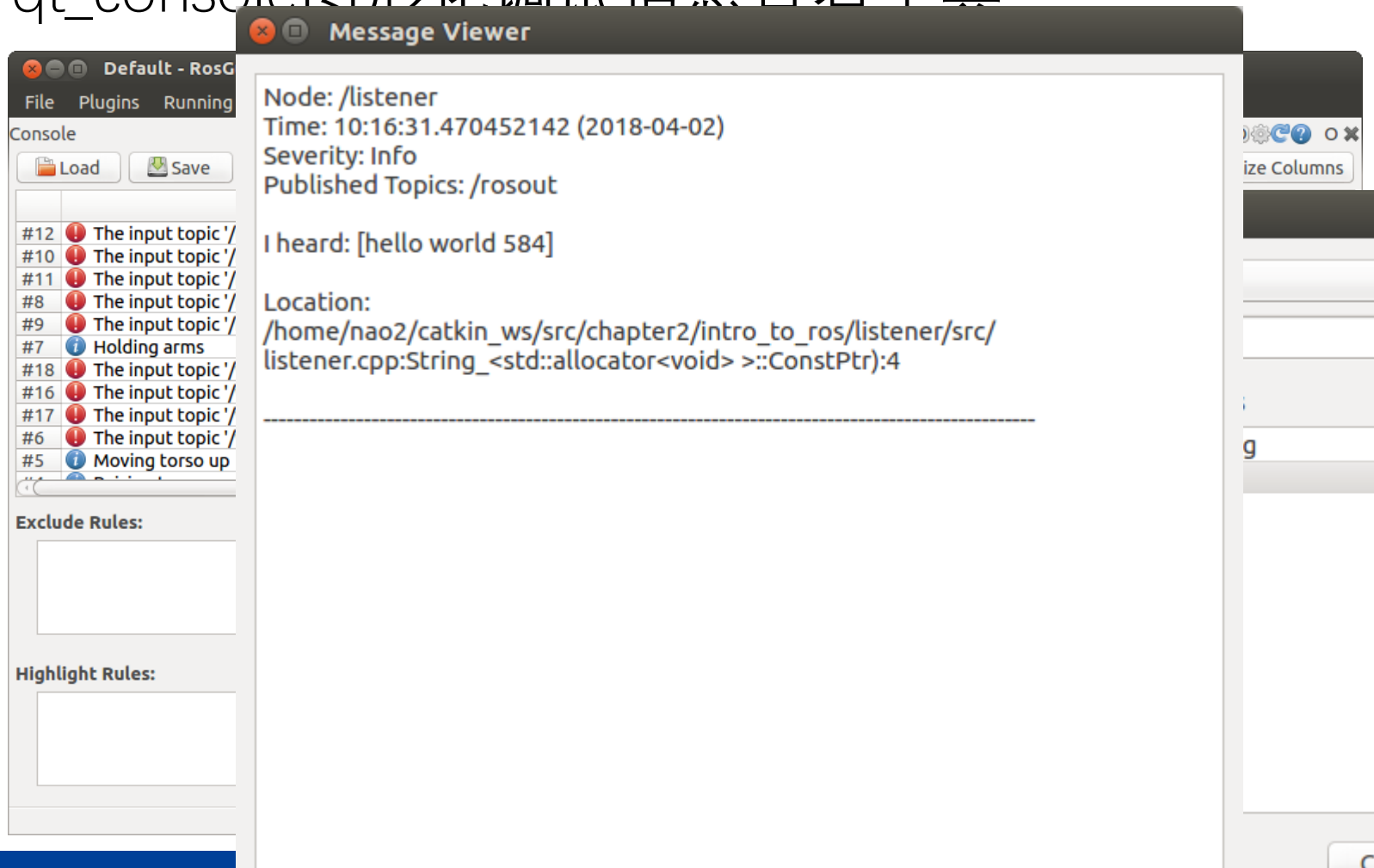
```
# Set the default ros output to warning and higher
log4j.logger.ros=WARN
# Override my package to output everything
log4j.logger.ros.my_package_name=DEBUG
```

  - 在launch 文件中

```
<launch>
    <env name="ROSCONSOLE_CONFIG_FILE"
        value="$(find mypackage)/custom_rosconsole.conf"/>
    <node pkg="mypackage" type="mynode" name="mynode"
output="screen"/>
</launch>
```

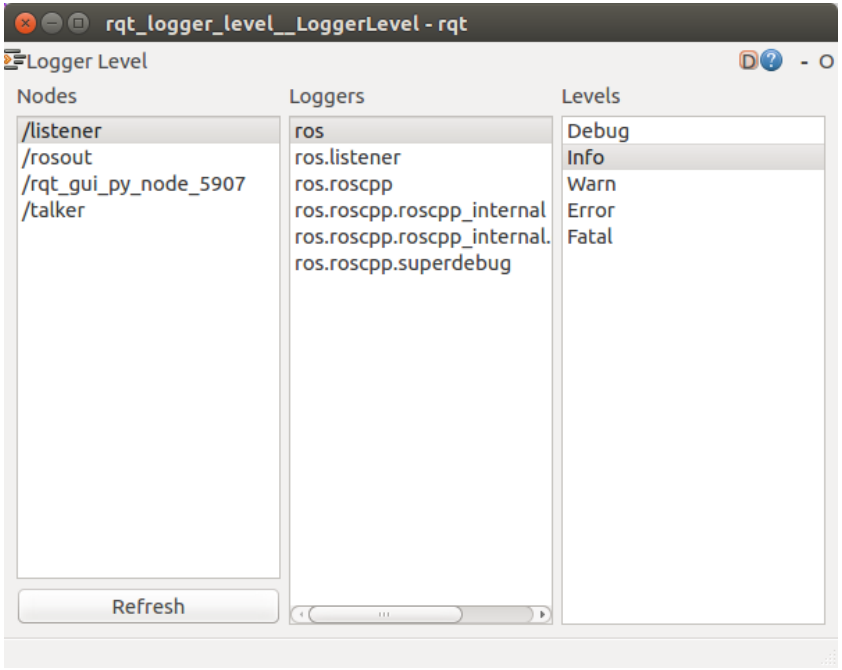
调试信息

- rqt_console图形化调试信息查看工具



调试信息

- rqt_logger_level图形化调试信息查看工具



目录 Contents

系统状态监控

- 在给定时间的运行状态是非常重要的。ROS 提供了一些基本而又非常强大的工具，它们不仅能实现状态监视，还能探测节点状态图中任何节点发生的功能失效。简而言之，使用主题来连接节点展现系统架构的状态。
  - 获得运行中节点的清单： rosnode list
  - 所有节点的主题请使用： rostopic list
  - 类似地，显示所有服务使用： rosservice list
  - 图形化界面rqt_graph

目录 Contents

roswtf

- ROS 还提供了另外一个工具来检测给定功能包中所有组件的潜在问题。使用roscd 移动到你想要分析的功能包路径下，然后运行roswtf 。
- roswtf会检查你的ROS设置，比如你的环境变量，并查找配置问题。 如果你有一个在线的ROS系统，它会查看它并检查是否有任何潜在的问题。

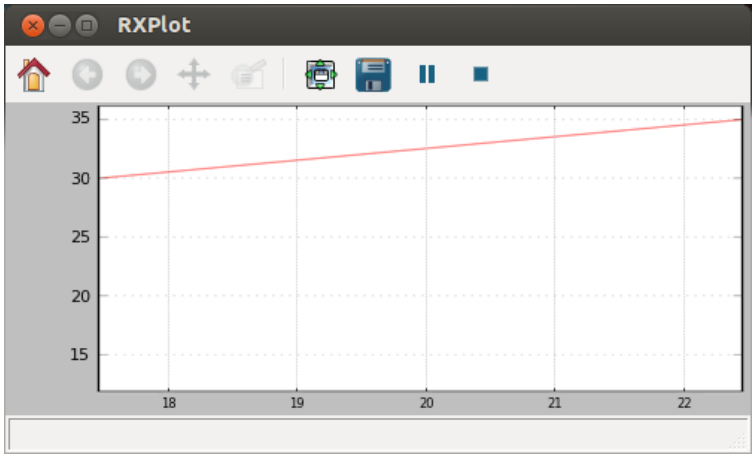
目录 Contents

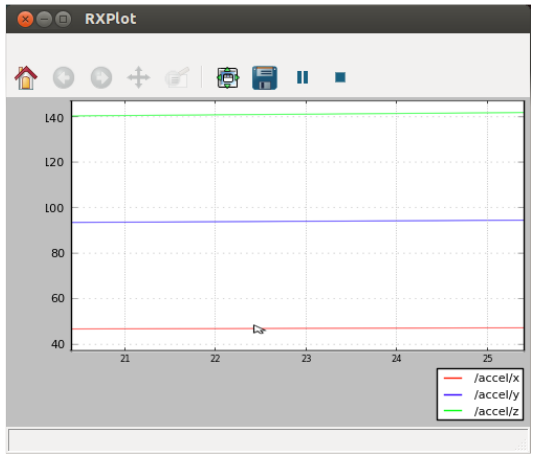
绘制标量图

- 我们可以使用ROS 中现有的一些通用工具，轻松地绘制标量数据图。它要求对每一个标量字段数据分别绘制成二维曲线。
- 大部分非标量数据结构更适合于使用专用的可视化工具，我们会在后面进行部分介绍，例如，图形、位姿、方向和角度。

绘制标量图

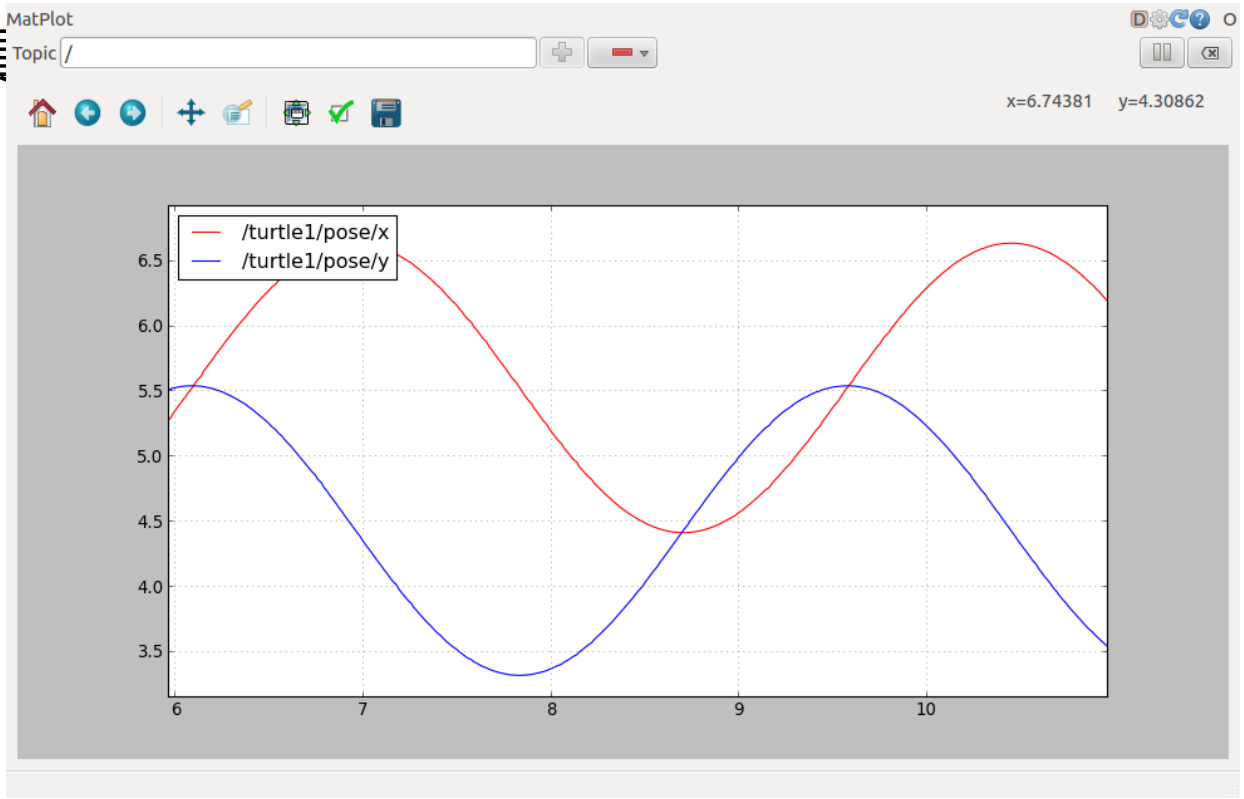
- 在ROS 系统中，标量数据可以根据消息中提供的时间戳作为时间序列绘制图形。然后，我们就能够在 y 轴上使用rqt_plot工具绘制标量数据。 rqt_plot 工具有一套功能强大的参数语法，允许我们在结构化消息中指定多个字段（当然使用了相当简明的方





绘制标量图

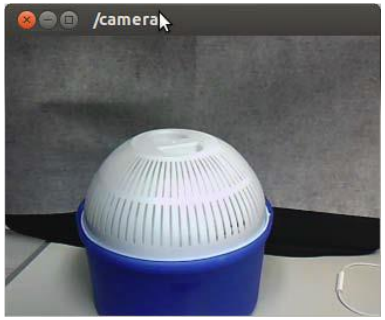
- rqt_plot提供了一个GUI插件，使用不同的绘图后端在二维图中显示数值。
- 运行乌龟仿真



目录 Contents

图像可视化

- 在一些使用图像处理的ROS包中，图像的可视化非常重要。ROS提供了很多的工具来显示图像。
  - rosrun image_view image_view __image:=/camera



  - Rviz中可以添加图像元素来显示图像，具有和image_view相同的效果。
- 安装 Sudo apt-get install ros-kinetic-usb-cam
- 运行rosparam set usb_cam/pixel_format yuyv 做一个参数设置
  - rosrun usb cam usb_cam_node

目录 Contents

3D可视化

- 有很多设备（例如双目摄像头、3D 激光雷达和Kinect 传感器）能够提供3D 数据。它们通常使用点云格式（组织好的或未组织的）。基于上述原因，能够使用工具实现3D 数据可视化就非常有用了。本节我们将介绍ROS 系统中的rviz 工具。它集成了能够完成3D 数据处理的OpenGL 接口，能够将传感器数据在模型化世界（world)中展示。因此，我们将会看到在复杂系统中至关重要的传感器的坐标系建立和坐标变换。
- rosrun rviz rviz
- 参考example7

目录 Contents

数据保存和回放

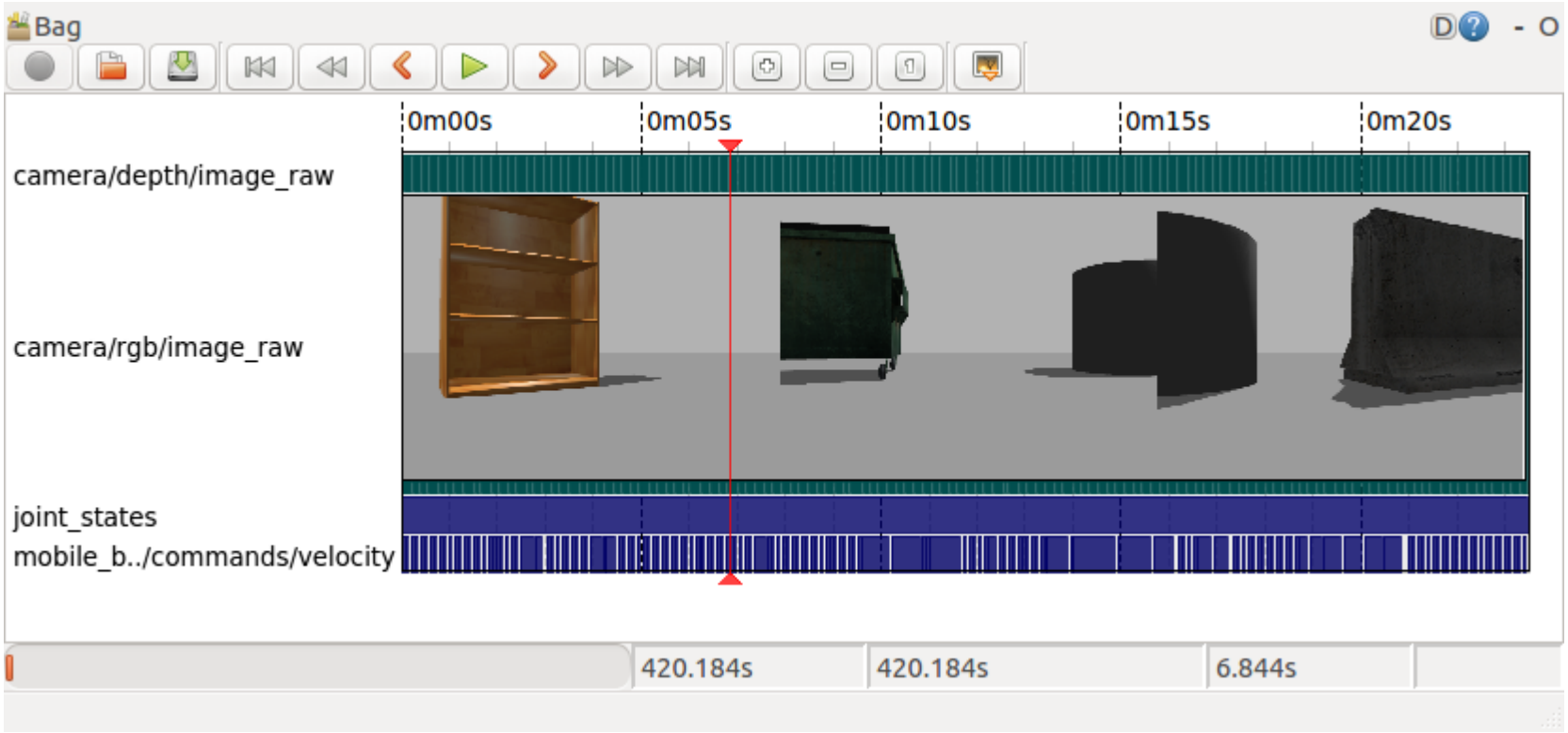
- 有通常情况下，当我们使用机器人系统时，资源都是共享的，并不一定总能提供给你一个人使用，或者因为种种原因实验不能顺利完成，如准备和实施实验就消耗了大量的时间。
- 基于以上原因，将实验会话数据记录下来用于未来的分析、处理、开发和算法验证，就是一个学习者必修的课程。然而要保证存储数据正确并能够用于离线回放并不是容易的事情，因为很多实验并不一定能反复进行。幸运的是，我们拥有ROS 中的强大工具能够完成这个功能。

数据保存和回放

- Rosbag
- 消息记录
  - rosbag record – a　　//所有消息
  - rosbag record /temp /accel　　//指定消息
- 消息回放
  - rosbag play ***.bag
- 参考example4_record.launch

数据保存和回放

- rqt_bag。图形化的bag工具，具有录制，回放，检查显示各种类型消息的功能。



目录 Contents

Rqt插件

- Rqt插件是一些非常实用的图像话工具，从前面的介绍中就提到了几个rqt插件
  - rqt_console
  - rqt_graph
  - rqt_plot
  - rqt_bag
- 更多要说明的是，这些可被单独运行的插件能够更广泛地应用，能够作为命令行(shell ） 、主题的发布者、消息类型的检查者。甚至rviz 都有一个名为rqt_rviz 的插件，还能够被集成在最新的rqt_gui 界面中。

谢谢！