深蓝学院
shenlanxueyuan.com

机器人操作系统ROS理论与实践

—— 第二讲：ROS基础

主讲人 胡春旭

机器人博客“古月居”博主（www.guyuehome.com）
深圳星河智能科技有限公司 联合创始人
华中科技大学自动化学院 硕士
Email：huchunxu@hust.edu.cn

课程概览

1. 认识ROS

- 课程介绍
- ROS现状与起源
- ROS总体架构
- ROS系统实现
- 第一个ROS例程

2. ROS基础

- 创建工作空间
- ROS通信编程
- 实现分布式通讯
- ROS中的关键组件

3. 机器人系统设计

- 机器人的定义与组成
- 机器人系统构建
- URDF机器人建模

4.机器人仿真

- 机器人模型优化
- ArbotiX+rviz功能仿真
- gazebo物理仿真

5.机器人感知

- 机器视觉
(图像校准、图像识别等)
- 机器语音
(科大讯飞SDK)

6.机器人SLAM与自主导航

- 机器人必备条件
- SLAM功能包的应用
- ROS中的导航框架
- 导航框架的应用

7. MoveIt!机械臂控制

- MoveIt!系统架构
- 创建机械臂模型
- MoveIt!编程学习
- Gazebo机械臂仿真
- ROS-I框架介绍

8. ROS机器人综合应用

- ROS机器人实例介绍
(PR2、Turtlebot、HRMRP、Kungfu Arm)
- 构建综合机器人平台

9. ROS 2.0

- 为什要有ROS 2
- 什么是ROS 2
- 如何安装ROS 2
- 话题与服务编程
- ROS 2与ROS 1的集成
- 课程总结与展望

目录

1. 创建工作空间

1. 什么是工作空间

工作空间（workspace）是一个存放工程开发相关文件的文件夹。

- **src**：代码空间（Source Space）
- **build**：编译空间（Build Space）
- **devel**：开发空间（Development Space）
- **install**：安装空间（Install Space）

```
workspace_folder/          -- WORKSPACE
  src/                     -- SOURCE SPACE
    CMakeLists.txt         -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CMakeLists.txt
      package.xml
      ...
  build/                   -- BUILD SPACE
    CATKIN_IGNORE          -- Keeps catkin from walking this directory
  devel/                   -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/                 -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
```

catkin编译系统下的工作空间结构

1. 创建工作空间

创建工作空间

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

编译工作空间

```
$ cd ~/catkin_ws/
$ catkin_make
```

设置环境变量

```
$ source devel/setup.bash
```

检查环境变量

```
$ echo $ROS_PACKAGE_PATH
```

```
➜  ~ echo $ROS_PACKAGE_PATH
/home/hcx/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks
```

1. 创建功能包

$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]

创建功能包

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg learning_communication std_msgs rospy roscpp
```

编译功能包

```
$ cd ~/catkin_ws
$ catkin_make
$ source ~/catkin_ws/devel/setup.bash
```

同一个工作空间下，不允许存在同名功能包
不同工作空间下，允许存在同名功能包

1. 工作空间的覆盖

ROS工作空间的Overlaying机制，即工作空间的覆盖。

```
➜  ~ env | grep ros
PATH=/home/hcx/catkin_ws/devel/bin:/opt/ros/kinetic/bin:/home/hcx/bin:/home/hcx/.local/bin:
/opt/jdk1.8.0_144/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/gam
es:/usr/local/games:/snap/bin
CMAKE_PREFIX_PATH=/home/hcx/catkin_ws/devel:/opt/ros/kinetic
LD_LIBRARY_PATH=/home/hcx/catkin_ws/devel/lib:/opt/ros/kinetic/lib:/opt/ros/kinetic/lib/x86
_64-linux-gnu
PKG_CONFIG_PATH=/home/hcx/catkin_ws/devel/lib/pkgconfig:/opt/ros/kinetic/lib/pkgconfig:/opt
/ros/kinetic/lib/x86_64-linux-gnu/pkgconfig
PYTHONPATH=/home/hcx/catkin_ws/devel/lib/python2.7/dist-packages:/opt/ros/kinetic/lib/pytho
n2.7/dist-packages
ROS_PACKAGE_PATH=/home/hcx/catkin_ws/src:/opt/ros/kinetic/share
ROS_ETC_DIR=/opt/ros/kinetic/etc/ros
ROS_ROOT=/opt/ros/kinetic/share/ros
TURTLEBOT_GAZEBO_MAP_FILE=/opt/ros/kinetic/share/turtlebot_gazebo/maps/playground.yaml
TURTLEBOT_GAZEBO_WORLD_FILE=/opt/ros/kinetic/share/turtlebot_gazebo/worlds/playground.world
TURTLEBOT_MAP_FILE=/opt/ros/kinetic/share/turtlebot_navigation/maps/willow-2010-02-18-0.10.
yaml
TURTLEBOT_STAGE_MAP_FILE=/opt/ros/kinetic/share/turtlebot_stage/maps/maze.yaml
TURTLEBOT_STAGE_WORLD_FILE=/opt/ros/kinetic/share/turtlebot_stage/maps/stage/maze.world
TURTLEBOT_STDR_MAP_FILE=/opt/ros/kinetic/share/turtlebot_stdr/maps/sparse_obstacles.yaml
```

查看ROS相关的环境变量

1. 工作空间的覆盖

- 工作空间的路径依次在ROS_PACKAGE_PATH环境变量中记录
- 新设置的路径在ROS_PACKAGE_PATH中会自动放置在最前端
- 运行时，ROS会优先查找最前端的工作空间中是否存在指定的功能包
- 如果不存在，就顺序向后查找其他工作空间

```
→  ~ rospack find roscpp_tutorials
/opt/ros/kinetic/share/roscpp_tutorials
```

系统路径下的功能包

```
→  catkin_ws rospack find roscpp_tutorials
/home/hcx/catkin_ws/src/ros_tutorials/roscpp_tutorials
```

工作空间下的功能包

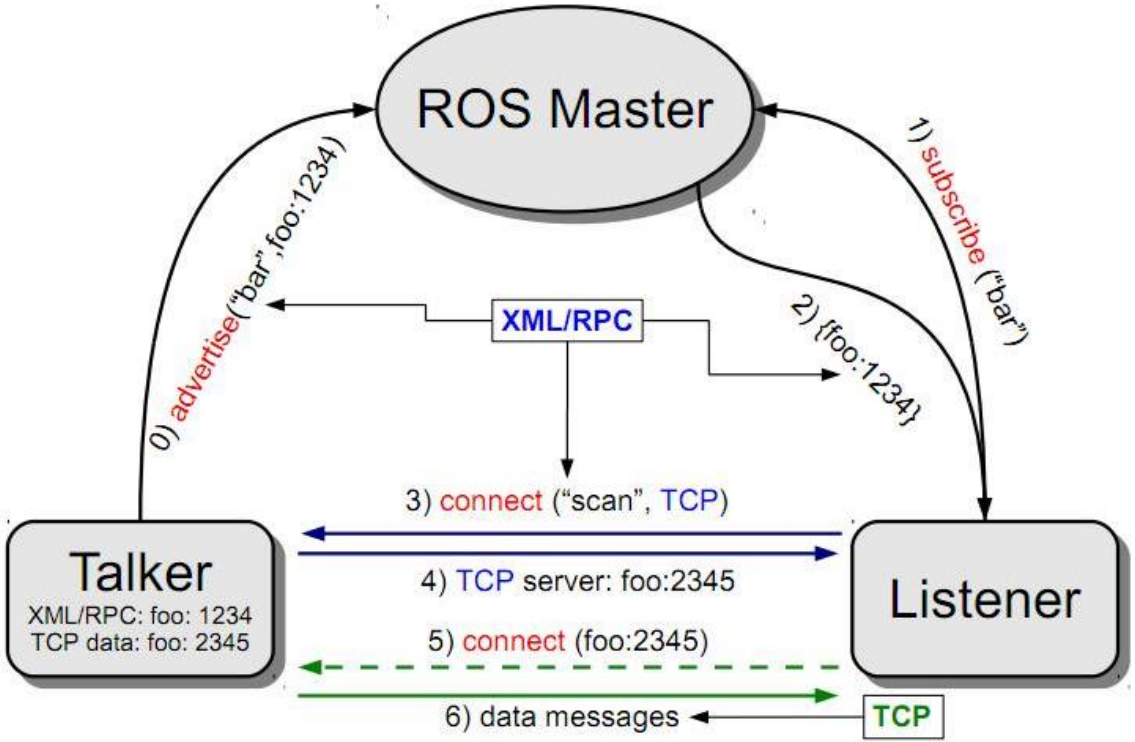
2. ROS通信编程

- 话题编程
- 服务编程
- 动作编程

2. ROS通信编程——话题编程

话题编程流程

- 创建发布者
- 创建订阅者
- 添加编译选项
- 运行可执行程序



话题通信模型

2. ROS通信编程——话题编程

如何实现一个发布者

- 初始化ROS节点；
- 向ROS Master注册节点信息，包括发布的话题名和话题中的消息类型；
- 按照一定频率循环发布消息。

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

int main(int argc, char **argv)
{
    // ROS节点初始化
    ros::init(argc, argv, "talker");

    // 创建节点句柄
    ros::NodeHandle n;

    // 创建一个Publisher，发布名为chatter的topic，消息类型为std_msgs::String
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

    // 设置循环的频率
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok())
    {
        // 初始化std_msgs::String类型的消息
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();

        // 发布消息
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);

        // 循环等待回调函数
        ros::spinOnce();

        // 按照循环频率延时
        loop_rate.sleep();
        ++count;
    }

    return 0;
}
```

talker.cpp

2. ROS通信编程——话题编程

如何实现一个订阅者

- 初始化ROS节点；
- 订阅需要的话题；
- 循环等待话题消息，接收到消息后进入回调函数；
- 在回调函数中完成消息处理。

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

// 接收到订阅的消息后，会进入消息回调函数
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    // 将接收到的消息打印出来
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // 初始化ROS节点
    ros::init(argc, argv, "listener");

    // 创建节点句柄
    ros::NodeHandle n;

    // 创建一个Subscriber，订阅名为chatter的topic，注册回调函数chatterCallback
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

    // 循环等待回调函数
    ros::spin();

    return 0;
}
```

listener.cpp

如何编译代码

- 设置需要编译的代码和生成的可执行文件；
- 设置链接库；
- 设置依赖。

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
# add_dependencies(talker ${PROJECT_NAME}_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
# add_dependencies(talker ${PROJECT_NAME}_generate_messages_cpp)
```

CMakeLists.txt

2. ROS通信编程——话题编程

如何运行可执行文件

```
→  ~ rosrun learning_communication talker
[ INFO] [1507648965.972490912]: hello world 0
[ INFO] [1507648966.072604444]: hello world 1
[ INFO] [1507648966.172600924]: hello world 2
[ INFO] [1507648966.272562923]: hello world 3
[ INFO] [1507648966.372559424]: hello world 4
[ INFO] [1507648966.472688746]: hello world 5
[ INFO] [1507648966.572691987]: hello world 6
[ INFO] [1507648966.672685917]: hello world 7
[ INFO] [1507648966.772684550]: hello world 8
[ INFO] [1507648966.872679978]: hello world 9
[ INFO] [1507648966.972579883]: hello world 10
[ INFO] [1507648967.072628016]: hello world 11
```

rosrun learning_communication talker

```
→  ~ rosrun learning_communication listener
[ INFO] [1507649032.961619694]: I heard: [hello world 56]
[ INFO] [1507649033.061319417]: I heard: [hello world 57]
[ INFO] [1507649033.161406093]: I heard: [hello world 58]
[ INFO] [1507649033.260928830]: I heard: [hello world 59]
[ INFO] [1507649033.361426764]: I heard: [hello world 60]
[ INFO] [1507649033.461283961]: I heard: [hello world 61]
[ INFO] [1507649033.561306202]: I heard: [hello world 62]
[ INFO] [1507649033.661385283]: I heard: [hello world 63]
[ INFO] [1507649033.761384508]: I heard: [hello world 64]
[ INFO] [1507649033.861318444]: I heard: [hello world 65]
[ INFO] [1507649033.961297490]: I heard: [hello world 66]
[ INFO] [1507649034.061417031]: I heard: [hello world 67]
[ INFO] [1507649034.161411960]: I heard: [hello world 68]
```

rosrun learning_communication listener

2. ROS通信编程——话题编程

如何自定义话题消息

```
string name
uint8  sex
uint8  age

uint8 unknown = 0
uint8 male    = 1
uint8 female  = 2
```

Person.msg

- **定义msg文件；**
- **在package.xml中添加功能包依赖**

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

- **在CMakeLists.txt添加编译选项**
  - find_package(…… message_generation)
  - catkin_package(CATKIN_DEPENDS geometry_msgs roscpp rospy std_msgs message_runtime)
  - add_message_files(FILES Person.msg)
    generate_messages(DEPENDENCIES std_msgs)

*部分ROS版本中的exec_depend需要改成run_depend

2. ROS通信编程——话题编程

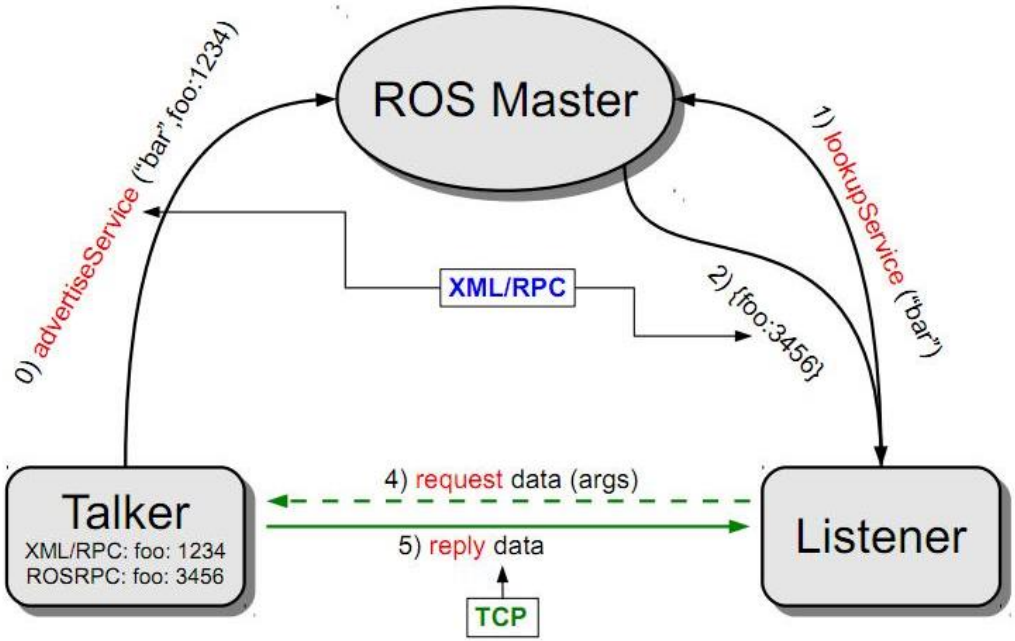
```
→  catkin_ws rosmsg show Person
[learning_communication/Person]:
uint8 unknown=0
uint8 male=1
uint8 female=2
string name
uint8 sex
uint8 age
```

查看自定义的消息

2. ROS通信编程——服务编程

服务编程流程

- 创建服务器
- 创建客户端
- 添加编译选项
- 运行可执行程序



服务通信模型

2. ROS通信编程——服务编程

如何自定义服务请求与应答

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

AddTwoInts.srv

- **定义srv文件；**
- **在package.xml中添加功能包依赖**

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

- **在CMakeLists.txt添加编译选项**
  - find_package(…… message_generation)
  - catkin_package(CATKIN_DEPENDS geometry_msgs roscpp rospy std_msgs message_runtime)
  - add_service_files(FILES AddTwoInts.srv)

*部分ROS版本中的exec_depend需要改成run_depend

2. ROS通信编程——服务编程

如何实现一个服务器

- 初始化ROS节点；
- 创建Server实例；
- 循环等待服务请求，进入回调函数；
- 在回调函数中完成服务功能的处理，并反馈应答数据。

```cpp
#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"

// service回调函数，输入参数req，输出参数res
bool add(learning_communication::AddTwoInts::Request  &req,
         learning_communication::AddTwoInts::Response &res)
{
    // 将输入参数中的请求数据相加，结果放到应答变量中
    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节点初始化
    ros::init(argc, argv, "add_two_ints_server");

    // 创建节点句柄
    ros::NodeHandle n;

    // 创建一个名为add_two_ints的server，注册回调函数add()
    ros::ServiceServer service = n.advertiseService("add_two_ints", add);

    // 循环等待回调函数
    ROS_INFO("Ready to add two ints.");
    ros::spin();

    return 0;
}
```

server.cpp

2. ROS通信编程——服务编程

如何实现一个客户端

- 初始化ROS节点；
- 创建一个Client实例；
- 发布服务请求数据；
- 等待Server处理之后的应答结果。

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

int main(int argc, char **argv)
{
    // ROS节点初始化
    ros::init(argc, argv, "add_two_ints_client");

    // 从终端命令行获取两个加数
    if (argc != 3)
    {
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }

    // 创建节点句柄
    ros::NodeHandle n;

    // 创建一个client，请求add_two_int service
    // service消息类型是learning_communication::AddTwoInts
    ros::ServiceClient client = n.serviceClient<learning_communication::AddTwoInts>("add_two_ints");

    // 创建learning_communication::AddTwoInts类型的service消息
    learning_communication::AddTwoInts srv;
    srv.request.a = atoll(argv[1]);
    srv.request.b = atoll(argv[2]);

    // 发布service请求，等待加法运算的应答结果
    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;
}
```

client.cpp

2. ROS通信编程——服务编程

如何编译代码

- 设置需要编译的代码和生成的可执行文件；
- 设置链接库；
- 设置依赖。

```
add_executable(server src/server.cpp)
target_link_libraries(server ${catkin_LIBRARIES})
add_dependencies(server ${PROJECT_NAME}_gencpp)

add_executable(client src/client.cpp)
target_link_libraries(client ${catkin_LIBRARIES})
add_dependencies(client ${PROJECT_NAME}_gencpp)
```

CMakeLists.txt

2. ROS通信编程——服务编程

如何运行可执行文件

```
→  ~ rosrun learning_communication server
[ INFO] [1507649760.914978873]: Ready to add two ints.
```

Server节点启动后的日志信息

```
→  ~ rosrun learning_communication client 3 5
[ INFO] [1507649815.838663270]: Sum: 8
```

Client启动后发布服务请求，并成功接收到反馈结果

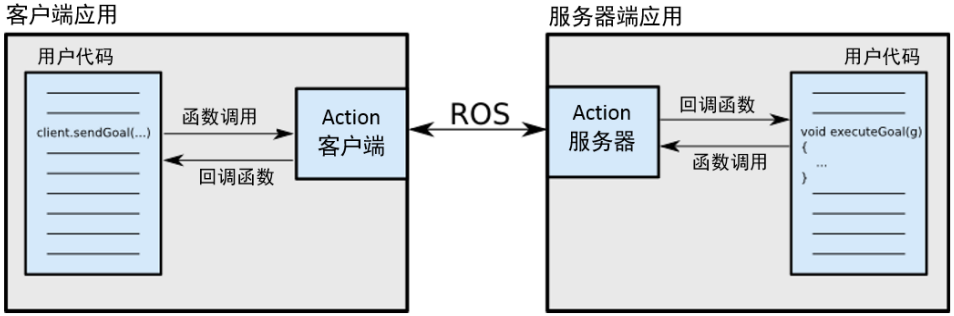
```
→  ~ rosrun learning_communication server
[ INFO] [1507649760.914978873]: Ready to add two ints.

[ INFO] [1507649815.838470408]: request: x=3, y=5
[ INFO] [1507649815.838508903]: sending back response: [8]
```

Server接收到服务调用后完成加法求解，并将结果反馈给Client

2. ROS通信编程——动作编程

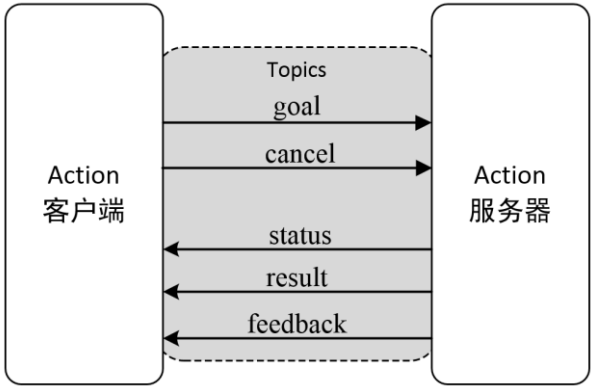
什么是动作（action）

- 一种问答通信机制;
- 带有连续反馈;
- 可以在任务过程中止运行;
- 基于ROS的消息机制实现。



Action的接口

- goal：发布任务目标;
- cancel：请求取消任务;
- status：通知客户端当前的状态;
- feedback：周期反馈任务运行的监控数据;
- result：向客户端发送任务的执行结果，只发布一次。



2. ROS通信编程——动作编程

如何自定义动作消息

```
# 定义目标信息
uint32 dishwasher_id
# Specify which dishwasher we wa
nt to use
---
# 定义结果信息
uint32 total_dishes_cleaned
---
# 定义周期反馈的消息
float32 percent_complete
```

DoDishes.action

- **定义action文件；**
- **在package.xml中添加功能包依赖**

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

- **在CMakeLists.txt添加编译选项**

```
find_package(catkin REQUIRED actionlib_msgs actionlib)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)
```

*部分ROS版本中的exec_depend需要改成run_depend

2. ROS通信编程——动作编程

如何实现一个动作服务器

- 初始化ROS节点；
- 创建动作服务器实例；
- 启动服务器，等待动作请求；
- 在回调函数中完成动作服务功能的处理，并反馈进度信息；
- 动作完成，发送结束信息。

```cpp
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include "learning_communication/DoDishesAction.h"

typedef actionlib::SimpleActionServer<learning_communication::DoDishesAction> Server;

// 收到action的goal后调用该回调函数
void execute(const learning_communication::DoDishesGoalConstPtr& goal, Server* as)
{
    ros::Rate r(1);
    learning_communication::DoDishesFeedback feedback;

    ROS_INFO("Dishwasher %d is working.", goal->dishwasher_id);

    // 假设洗盘子的进度，并且按照1hz的频率发布进度feedback
    for(int i=1; i<=10; i++)
    {
        feedback.percent_complete = i * 10;
        as->publishFeedback(feedback);
        r.sleep();
    }

    // 当action完成后，向客户端返回结果
    ROS_INFO("Dishwasher %d finish working.", goal->dishwasher_id);
    as->setSucceeded();
}

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

    // 定义一个服务器
    Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);

    // 服务器开始运行
    server.start();

    ros::spin();

    return 0;
}
```

DoDishes_server.cpp

2. ROS通信编程——动作编程

如何实现一个动作客户端

- 初始化ROS节点；
- 创建动作客户端实例；
- 连接动作服务端；
- 发送动作目标；
- 根据不同类型的服务端反馈处理回调函数。

```cpp
#include <actionlib/client/simple_action_client.h>
#include "learning_communication/DoDishesAction.h"

typedef actionlib::SimpleActionClient<learning_communication::DoDishesAction> Client;

// 当action完成后会调用该回调函数一次
void doneCb(const actionlib::SimpleClientGoalState& state,
        const learning_communication::DoDishesResultConstPtr& result)
{
    ROS_INFO("Yay! The dishes are now clean");
    ros::shutdown();
}

// 当action激活后会调用该回调函数一次
void activeCb()
{
    ROS_INFO("Goal just went active");
}

// 收到feedback后调用该回调函数
void feedbackCb(const learning_communication::DoDishesFeedbackConstPtr& feedback)
{
    ROS_INFO(" percent_complete : %f ", feedback->percent_complete);
}

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

    // 定义一个客户端
    Client client("do_dishes", true);

    // 等待服务器端
    ROS_INFO("Waiting for action server to start.");
    client.waitForServer();
    ROS_INFO("Action server started, sending goal.");

    // 创建一个action的goal
    learning_communication::DoDishesGoal goal;
    goal.dishwasher_id = 1;

    // 发送action的goal给服务器端，并且设置回调函数
    client.sendGoal(goal,  &doneCb, &activeCb, &feedbackCb);

    ros::spin();

    return 0;
}
```

DoDishes_client.cpp

2. ROS通信编程——动作编程

如何编译代码

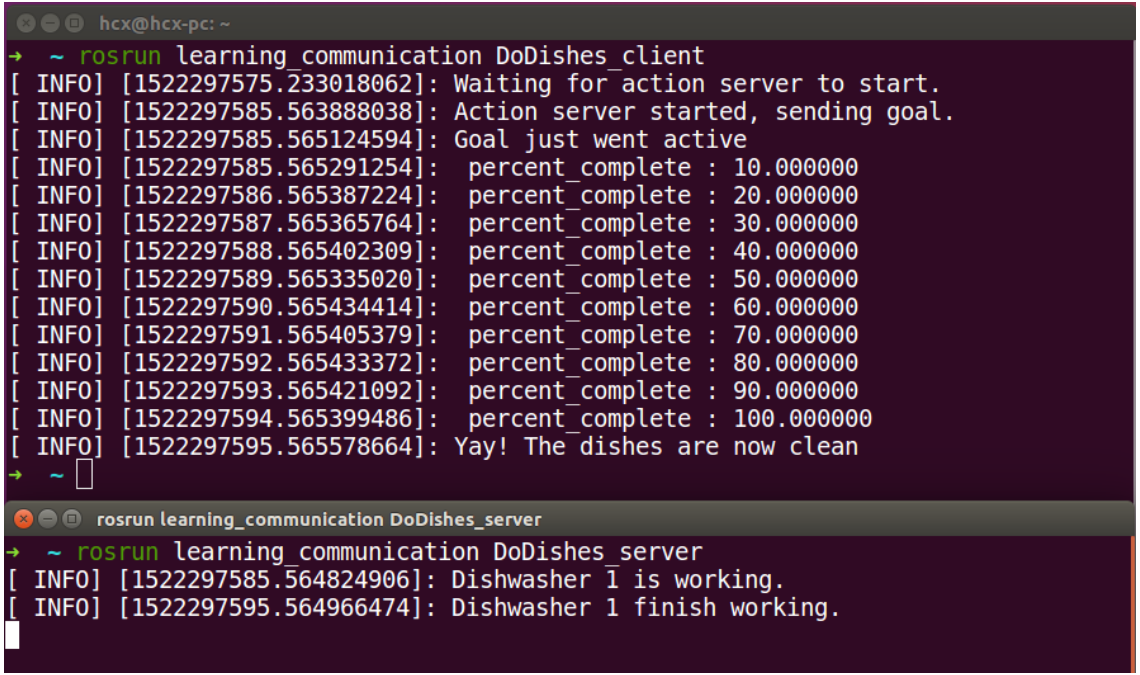
- 设置需要编译的代码和生成的可执行文件；
- 设置链接库；
- 设置依赖。

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

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

CMakeLists.txt

2. ROS通信编程——动作编程

如何运行可执行文件



action例程的运行效果

3. 分布式通信

3. 分布式通信

ROS是一种分布式软件框架，节点之间通过松耦合的方式进行组合。

如何实现分布式多机通信

（1）设置IP地址，确保底层链路的联通

```
wlp2s0    Link encap:Ethernet  HWaddr ac:2b:6e:5d:cc:85
          inet addr:192.168.31.198  Bcast:192.168.31.255  Mask:255.255.255.0
          inet6 addr: fe80::6751:3a89:703f:7e9c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11169 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7686 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10038665 (10.0 MB)  TX bytes:1153112 (1.1 MB)
```

```
wlxc83a35b01533 Link encap:Ethernet  HWaddr c8:3a:35:b0:15:33
          inet addr:192.168.31.14  Bcast:192.168.31.255  Mask:255.255.255.0
          inet6 addr: fe80::2c61:6fea:37e4:69fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:287 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:97160 (97.1 KB)  TX bytes:11068 (11.0 KB)
```

两台计算机（hcx-pc、raspi2）的IP地址

```
hcx@hcx-pc:~$ ping raspi2
PING raspi2 (192.168.31.14) 56(84) bytes of data.
64 bytes from raspi2 (192.168.31.14): icmp_seq=1 ttl=64 time=2.24 ms
64 bytes from raspi2 (192.168.31.14): icmp_seq=2 ttl=64 time=2.37 ms
64 bytes from raspi2 (192.168.31.14): icmp_seq=3 ttl=64 time=2.25 ms
64 bytes from raspi2 (192.168.31.14): icmp_seq=4 ttl=64 time=2.20 ms
64 bytes from raspi2 (192.168.31.14): icmp_seq=5 ttl=64 time=2.14 ms
```

```
robot@raspi2:~$ ping hcx-pc
PING hcx-pc (192.168.31.198) 56(84) bytes of data.
64 bytes from hcx-pc (192.168.31.198): icmp_seq=1 ttl=64 time=9.68 ms
64 bytes from hcx-pc (192.168.31.198): icmp_seq=2 ttl=64 time=6.39 ms
64 bytes from hcx-pc (192.168.31.198): icmp_seq=3 ttl=64 time=1.76 ms
64 bytes from hcx-pc (192.168.31.198): icmp_seq=4 ttl=64 time=5.32 ms
64 bytes from hcx-pc (192.168.31.198): icmp_seq=5 ttl=64 time=7.05 ms
64 bytes from hcx-pc (192.168.31.198): icmp_seq=6 ttl=64 time=7.02 ms
```

在两台计算机上分别使用ping命令测试网络是否联通

3. 分布式通信

如何实现分布式多机通信

（2）在从机端设置ROS_MASTER_URI，让从机找到ROS Master

$ export ROS_MASTER_URI=http://hcx-pc:11311 （当前终端有效）

或

$ echo “export ROS_MASTER_URI=http://hcx-pc:11311” >> ~/.bzshrc （所有终端有效）

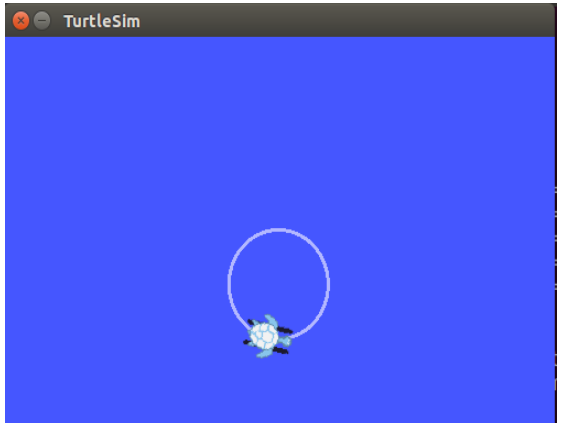
3. 分布式通信

主机端启动ROS Master与海龟仿真节点

```
$ roscore
$ rosrun turtlesim turtlesim_node
```

从机端发布一个速度控制消息

```
$ rostopic pub -r 10 /turtle1/cmd_vel
  geometry_msgs/Twist "linear:
    x: 0.5
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.5"
```



4. ROS中的关键组件

- Launch文件
- TF坐标变换
- Qt工具箱
- Rviz可视化平台
- Gazebo物理仿真环境

4. ROS中的关键组件——Launch文件

```
<launch>
  <!-- local machine already has a definition by default.
       This tag overrides the default definition with
       specific ROS_ROOT and ROS_PACKAGE_PATH values -->
  <machine name="local_alt" address="localhost" default="true" ros-root="/u/user/ros/ros/" ros-package-path="/u/user/ros/ros-pkg" />
  <!-- a basic listener node -->
  <node name="listener-1" pkg="rospy_tutorials" type="listener" />
  <!-- pass args to the listener node -->
  <node name="listener-2" pkg="rospy_tutorials" type="listener" args="-foo arg2" />
  <!-- a respawn-able listener node -->
  <node name="listener-3" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start listener node in the 'wg1' namespace -->
  <node ns="wg1" name="listener-wg1" pkg="rospy_tutorials" type="listener" respawn="true" />
  <!-- start a group of nodes in the 'wg2' namespace -->
  <group ns="wg2">
    <!-- remap applies to all future statements in this scope. -->
    <remap from="chatter" to="hello"/>
    <node pkg="rospy_tutorials" type="listener" name="listener" args="--test" respawn="true" />
    <node pkg="rospy_tutorials" type="talker" name="talker">
      <!-- set a private parameter for the node -->
      <param name="talker_1_param" value="a value" />
      <!-- nodes can have their own remap args -->
      <remap from="chatter" to="hello-1"/>
      <!-- you can set environment variables for a node -->
      <env name="ENV_EXAMPLE" value="some value" />
    </node>
  </group>
</launch>
```

Launch文件：通过XML文件实现多节点的配置和启动（**可自动启动ROS Master**）

4. ROS中的关键组件——Launch文件

```
<launch>
    <node pkg="turtlesim" name="sim1" type="turtlesim_node"/>
    <node pkg="turtlesim" name="sim2" type="turtlesim_node"/>
</launch>
```

<launch> launch文件中的根元素采用<launch>标签定义

<node>

启动节点

<node pkg="package-name" type="executable-name" name="node-name" />

- pkg：节点所在的功能包名称
- type：节点的可执行文件名称
- name：节点运行时的名称
- output、respawn、required、ns、args

4. ROS中的关键组件——Launch文件

参数设置

<param> / <rosparam>

设置ROS系统运行中的参数，存储在参数服务器中。

<param name="output_frame" value="odom"/>

- name：参数名
- value：参数值

加载参数文件中的多个参数：

<rosparam file="params.yaml" command="load" ns= “params" />

<arg>

launch文件内部的局部变量，仅限于launch文件使用

<arg name="arg-name" default="arg-value" />

- name：参数名
- value：参数值

调用：

<param name="foo" value="$(arg arg-name)" />

<node name="node" pkg="package" type="type " args="$(arg arg-name)" />

4. ROS中的关键组件——Launch文件

重映射

<remap >

重映射ROS计算图资源的命名。

<remap from="/turtlebot/cmd_vel" to="/cmd_vel"/>

- from：原命名
- to：映射之后的命名

嵌套

<include>

包含其他launch文件，类似C语言中的头文件包含。

<include file="$(dirname)/other.launch" />

- file：包含的其他launch文件路径

更多标签可参见：http://wiki.ros.org/roslaunch/XML

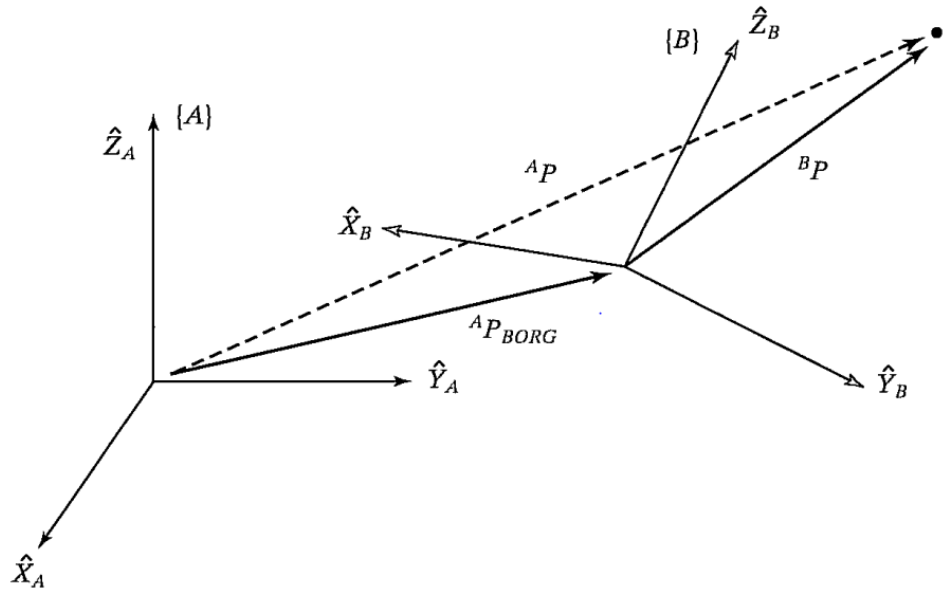
4. ROS中的关键组件——TF坐标变换

机器人中的坐标变换

$$ {}^{A}P = {}^{A}_{B}R\ {}^{B}P + {}^{A}P_{BORG}. $$

$$ {}^{A}P = {}^{A}_{B}T\ {}^{B}P. $$

$$ \begin{bmatrix} {}^{A}P \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & {}^{A}_{B}R & & {}^{A}P_{BORG} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} {}^{B}P \\ 1 \end{bmatrix}. $$



某位姿在A、B两个坐标系下的坐标变换

参考：《机器人学导论》

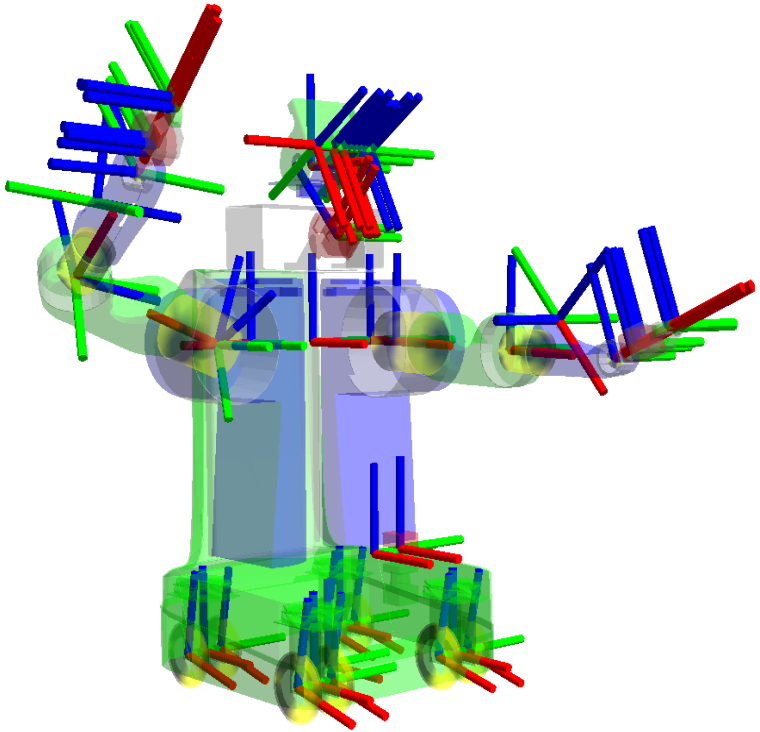
4. ROS中的关键组件——TF坐标变换

TF功能包能干什么?

- 五秒钟之前，机器人头部坐标系相对于全局坐标系的关系是什么样的?
- 机器人夹取的物体相对于机器人中心坐标系的位置在哪里?
- 机器人中心坐标系相对于全局坐标系的位置在哪里?

TF坐标变换如何实现?

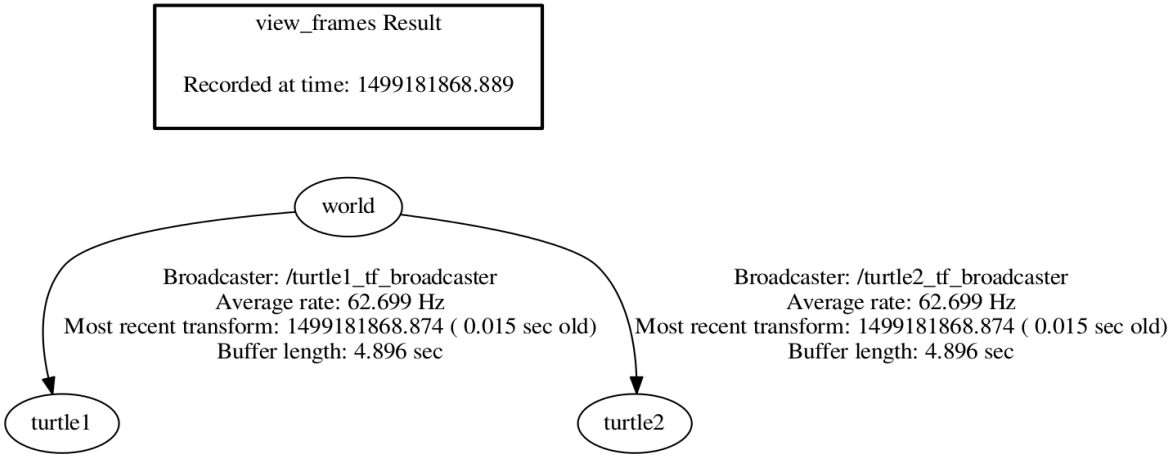
- 广播TF变换
- 监听TF变换

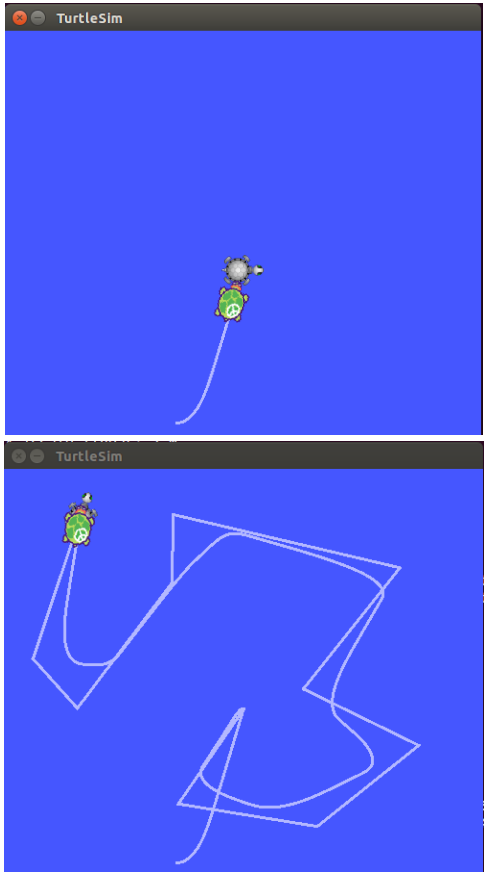


机器人系统中繁杂的坐标系

4. ROS中的关键组件——TF坐标变换

```
$ sudo apt-get install ros-kinetic-turtle-tf
$ roslaunch turtle_tf turtle_tf_demo.launch
$ rosrun turtlesim turtle_teleop_key
$ rosrun tf view_frames
```





小海龟跟随实验

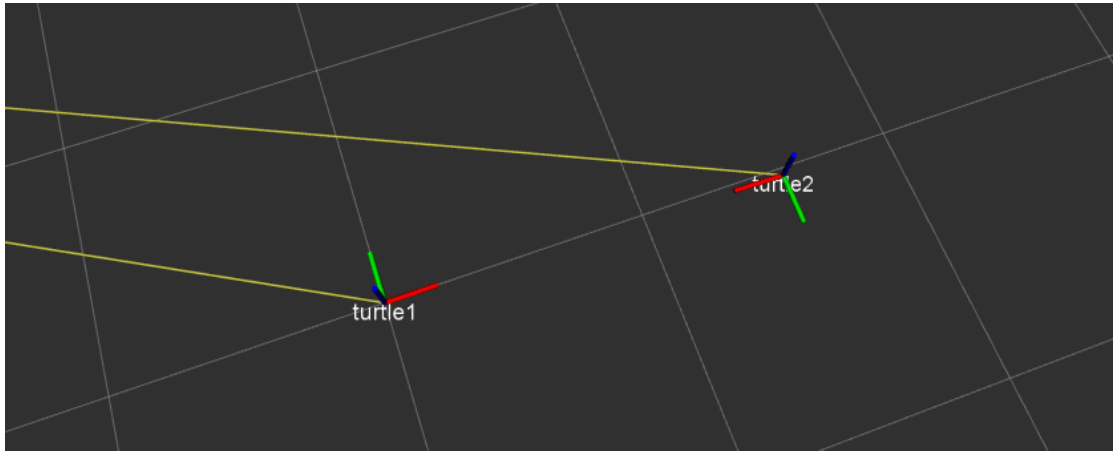
4. ROS中的关键组件——TF坐标变换

命令行工具

```
→ ~ rosrun tf tf_echo turtle1 turtle2
At time 1504942486.329
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.311, 0.950]
            in RPY (radian) [0.000, -0.000, 0.633]
            in RPY (degree) [0.000, -0.000, 36.290]
At time 1504942487.018
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.311, 0.950]
            in RPY (radian) [0.000, -0.000, 0.633]
            in RPY (degree) [0.000, -0.000, 36.290]
```

$$T_{turtle1\_turtle2} = T_{turtle1\_world} * T_{world\_turtle2}$$

可视化工具



$ rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz

4. ROS中的关键组件——TF坐标变换

如何实现一个TF广播器

- 定义TF广播器；（TransformBroadcaster）
- 创建坐标变换值；
- 发布坐标变换。（sendTransform）

```cpp
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>

std::string turtle_name;

void poseCallback(const turtlesim::PoseConstPtr& msg)
{
    // tf广播器
    static tf::TransformBroadcaster br;

    // 根据乌龟当前的位姿，设置相对于世界坐标系的坐标变换
    tf::Transform transform;
    transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
    tf::Quaternion q;
    q.setRPY(0, 0, msg->theta);
    transform.setRotation(q);

    // 发布坐标变换
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", turtle_name));
}

int main(int argc, char** argv)
{
    // 初始化节点
    ros::init(argc, argv, "my_tf_broadcaster");
    if (argc != 2)
    {
        ROS_ERROR("need turtle name as argument");
        return -1;
    };
    turtle_name = argv[1];

    // 订阅乌龟的pose信息
    ros::NodeHandle node;
    ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10, &poseCallback);

    ros::spin();

    return 0;
};
```

turtle_tf_broadcaster.cpp

4. ROS中的关键组件——TF坐标变换

如何实现一个TF监听器

- 定义TF监听器；（TransformListener）
- 查找坐标变换；（waitForTransform、 lookupTransform）

```cpp
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Spawn.h>

int main(int argc, char** argv)
{
    // 初始化节点
    ros::init(argc, argv, "my_tf_listener");

    ros::NodeHandle node;

    // 通过服务调用，产生第二只乌龟turtle2
    ros::service::waitForService("spawn");
    ros::ServiceClient add_turtle =
    node.serviceClient<turtlesim::Spawn>("spawn");
    turtlesim::Spawn srv;
    add_turtle.call(srv);

    // 定义turtle2的速度控制发布器
    ros::Publisher turtle_vel =
    node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);

    // tf监听器
    tf::TransformListener listener;

    ros::Rate rate(10.0);
    while (node.ok())
    {
        tf::StampedTransform transform;
        try
        {
            // 查找turtle2与turtle1的坐标变换
            listener.waitForTransform("/turtle2", "/turtle1", ros::Time(0), ros::Duration(3.0));
            listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0), transform);
        }
        catch (tf::TransformException &ex)
        {
            ROS_ERROR("%s",ex.what());
            ros::Duration(1.0).sleep();
            continue;
        }

        // 根据turtle1和turtle2之间的坐标变换，计算turtle2需要运动的线速度和角速度
        // 并发布速度控制指令，使turtle2向turtle1移动
        geometry_msgs::Twist vel_msg;
        vel_msg.angular.z = 4.0 * atan2(transform.getOrigin().y(),
                                        transform.getOrigin().x());
        vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                      pow(transform.getOrigin().y(), 2));
        turtle_vel.publish(vel_msg);

        rate.sleep();
    }
    return 0;
};
```

turtle_tf_listener.cpp

4. ROS中的关键组件——TF坐标变换

如何编译代码

- 设置需要编译的代码和生成的可执行文件；
- 设置链接库。

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

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

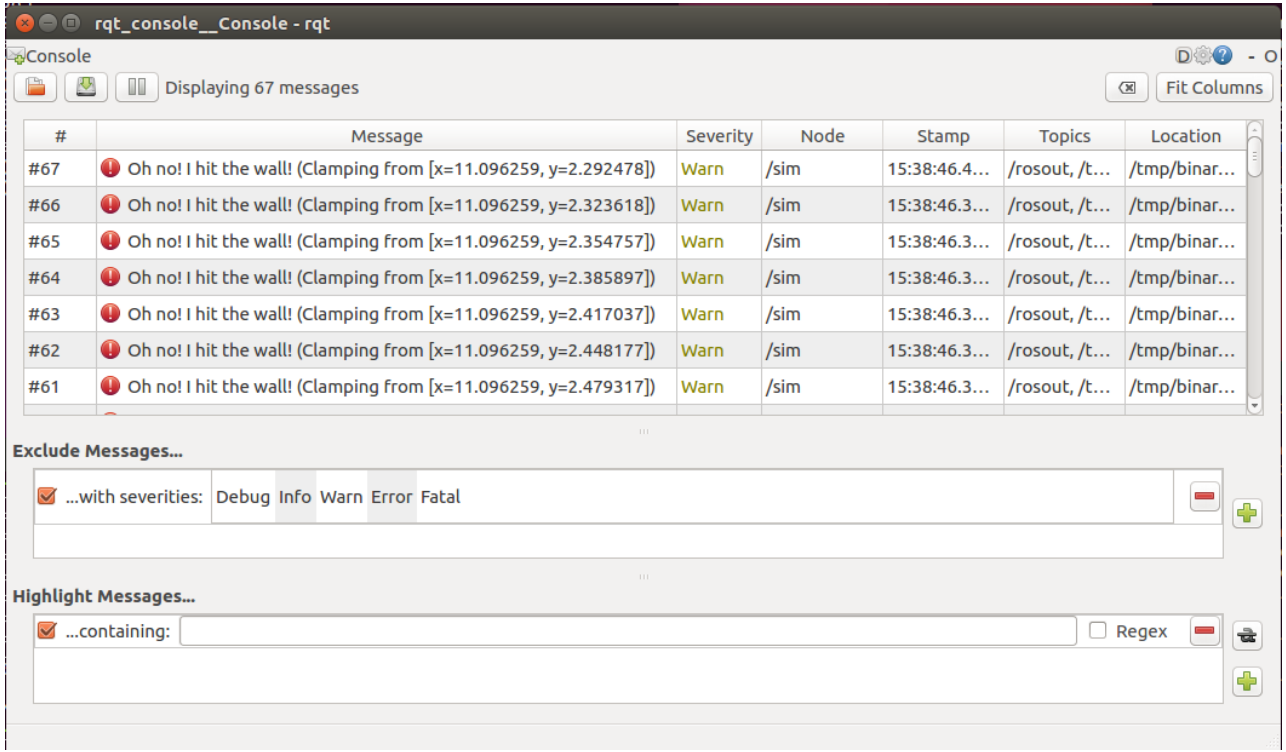
CMakeLists.txt

4. ROS中的关键组件——TF坐标变换

```
<launch>

    <!-- 海龟仿真器 -->
    <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

    <!-- 键盘控制 -->
    <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

    <!-- 两只海龟的tf广播 -->
    <node pkg="learning_tf" type="turtle_tf_broadcaster" args="/turtle1" name="turtle1_tf_broadcaster" />
    <node pkg="learning_tf" type="turtle_tf_broadcaster" args="/turtle2" name="turtle2_tf_broadcaster" />

    <!-- 监听tf广播，并且控制turtle2移动 -->
    <node pkg="learning_tf" type="turtle_tf_listener" name="listener" />

</launch>
```

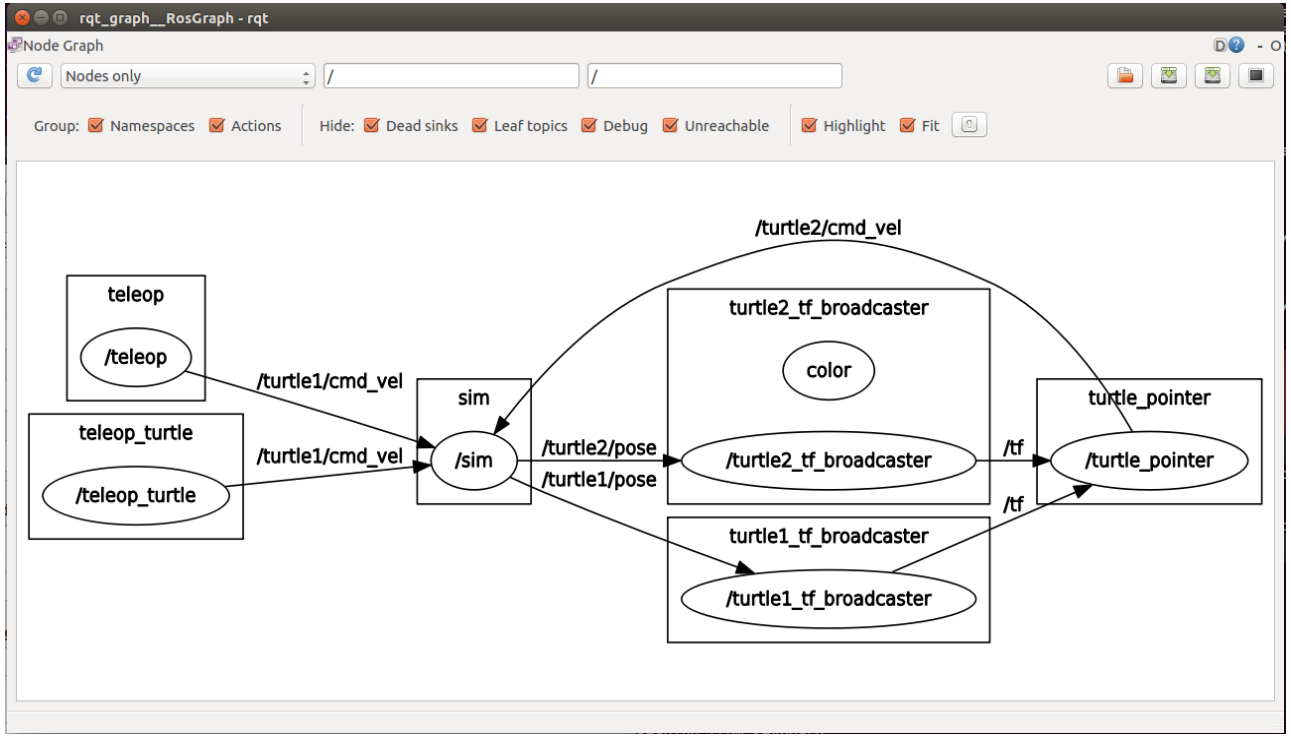
启动小海龟跟随例程

4. ROS中的关键组件——Qt工具箱



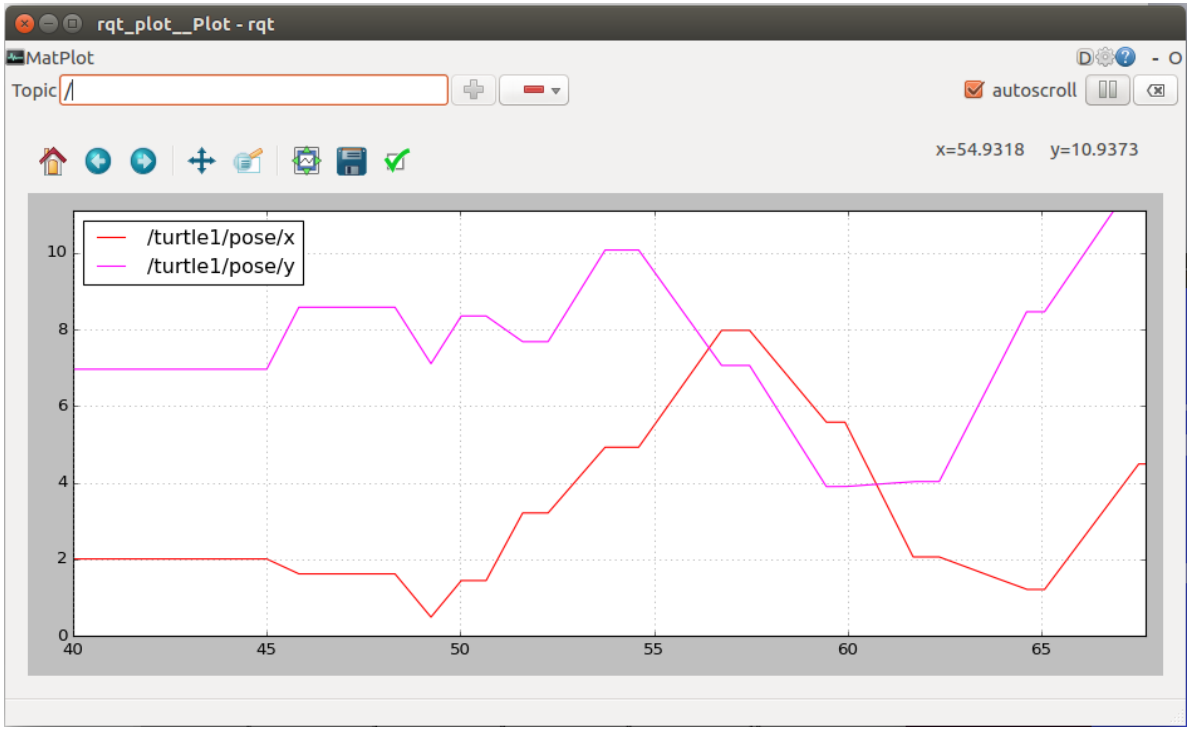
日志输出工具——rqt_console

4. ROS中的关键组件——Qt工具箱



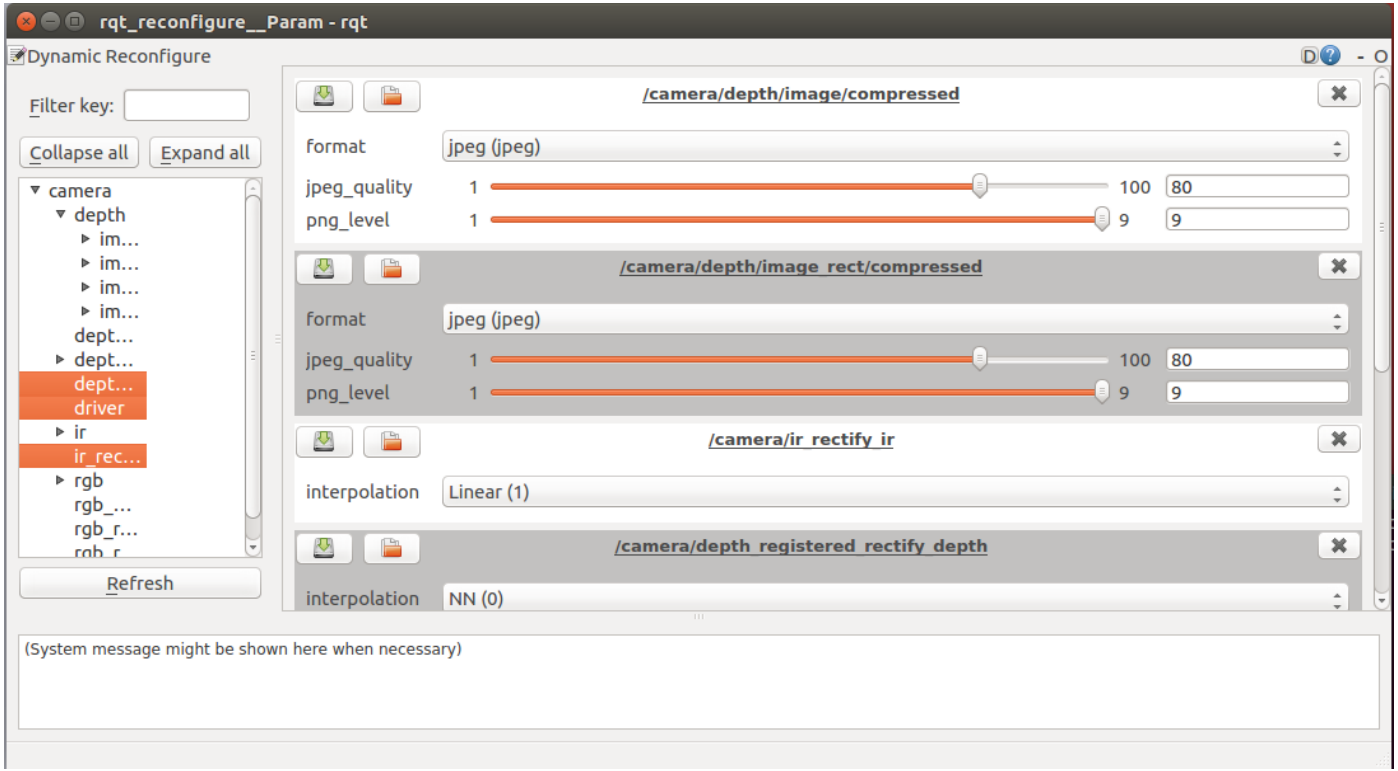
计算图可视化工具——rqt_graph

4. ROS中的关键组件——Qt工具箱



数据绘图工具——rqt_plot

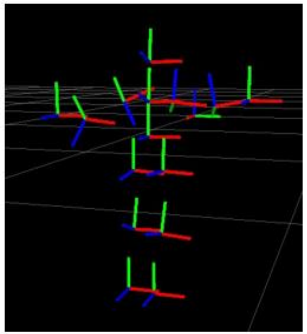
4. ROS中的关键组件——Qt工具箱



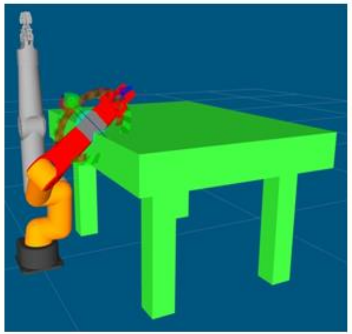
参数动态配置工具——rqt_reconfigure

4. ROS中的关键组件——Rviz可视化平台

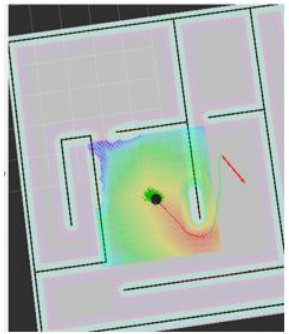
机器人模型



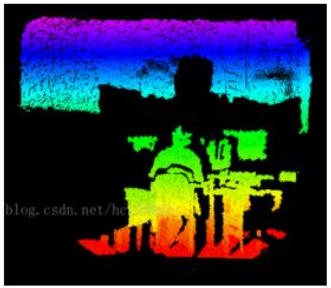
坐标



运动规划

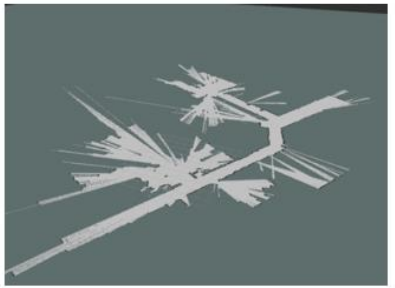


导航



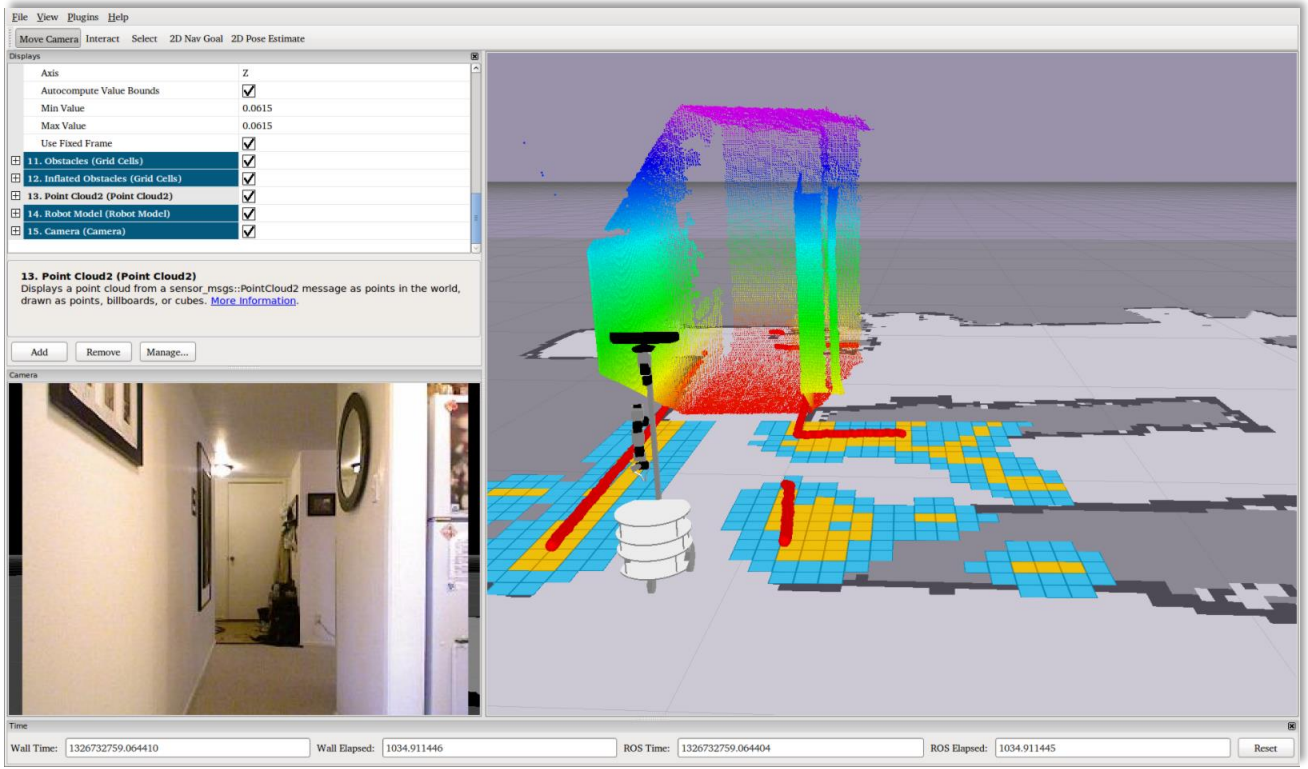
点云

图像



SLAM

4. ROS中的关键组件——Rviz可视化平台



机器人开发过程中的数据可视化界面

4. ROS中的关键组件——Rviz可视化平台

Rviz是一款**三维可视化工具**，可以很好的兼容基于ROS软件框架的机器人平台。

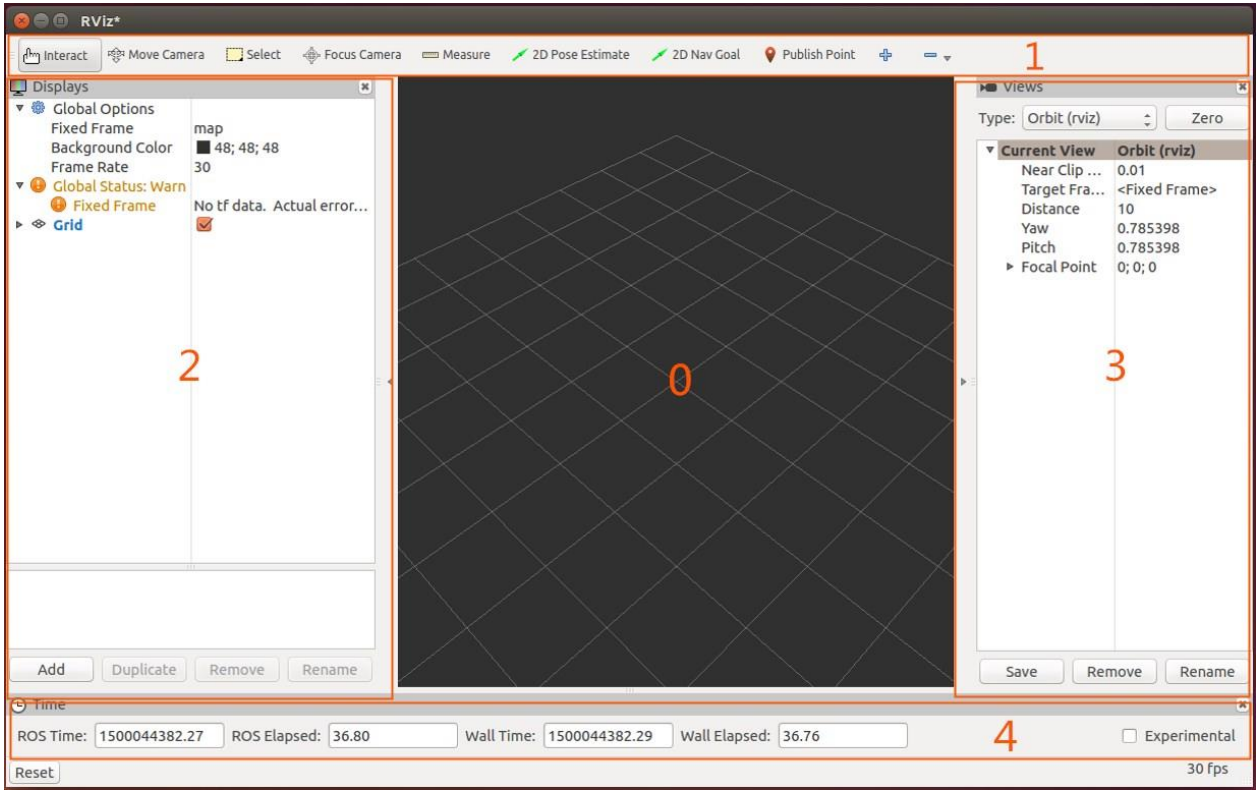
- 在rviz中，可以使用**可扩展标记语言XML**对机器人、周围物体等任何实物进行尺寸、质量、位置、材质、关节等属性的描述，并且在界面中呈现出来。

- 同时，rviz还可以通过**图形化的方式**，实时显示机器人传感器的信息、机器人的运动状态、周围环境的变化等信息。

- 总而言之，rviz通过机器人模型参数、机器人发布的传感信息等数据，为用户进行所有**可监测信息的图形化显示**。用户和开发者也可以在rviz的控制界面下，通过按钮、滑动条、数值等方式，控制机器人的行为。

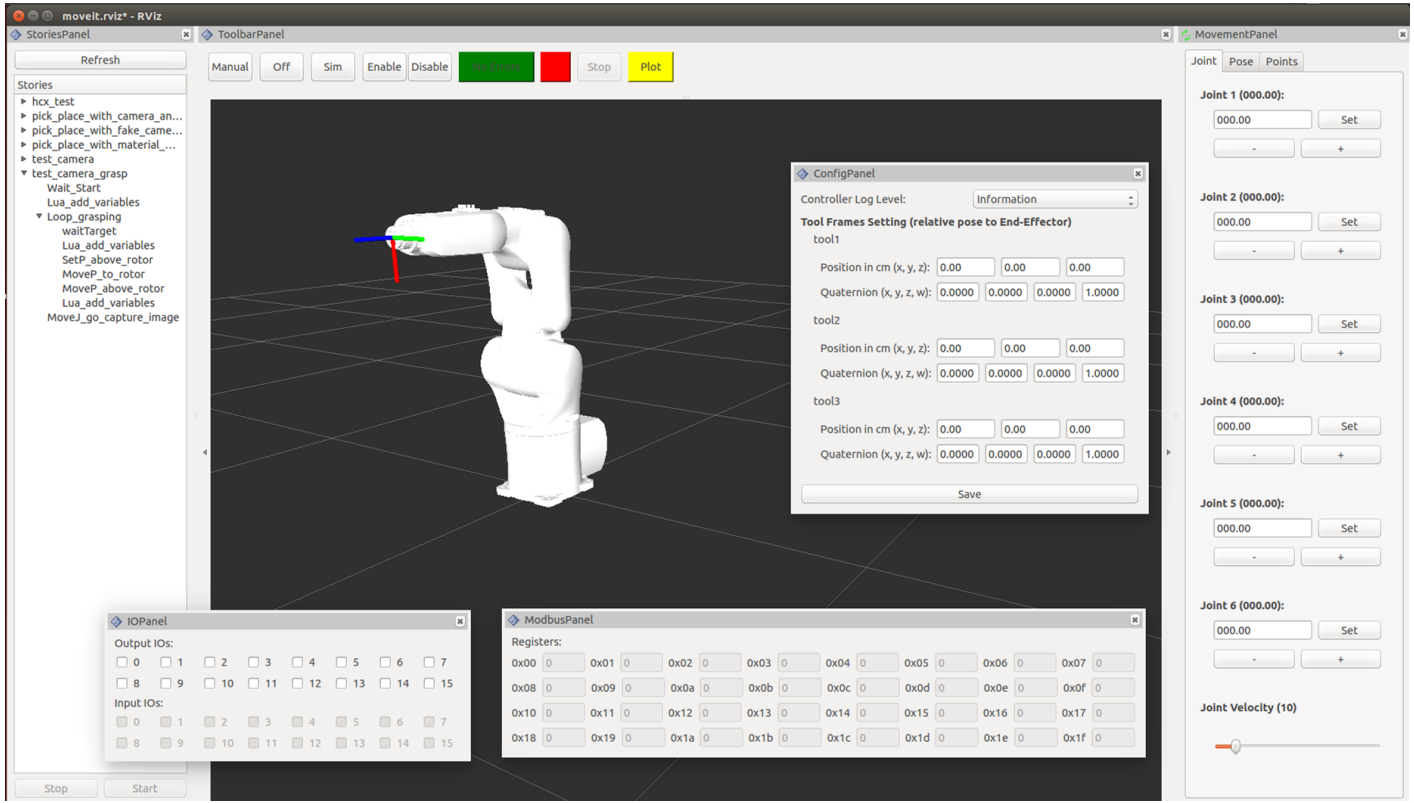
4. ROS中的关键组件——Rviz可视化平台

- 0 ：3D视图区
- 1 ：工具栏
- 2 ：显示项列表
- 3 ：视角设置区
- 4 ：时间显示区



4. ROS中的关键组件——Rviz可视化平台

Rviz的插件机制



基于rviz开发的机械臂控制系统

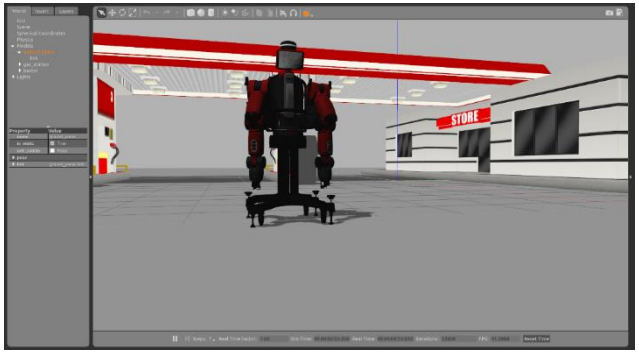
4. ROS中的关键组件——Gazebo物理仿真环境

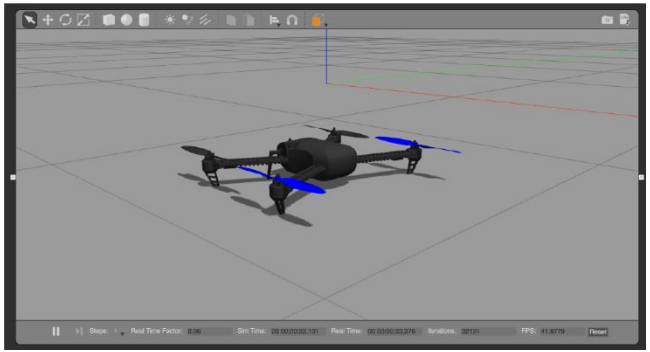
Gazebo是一款功能强大的三维物理仿真平台

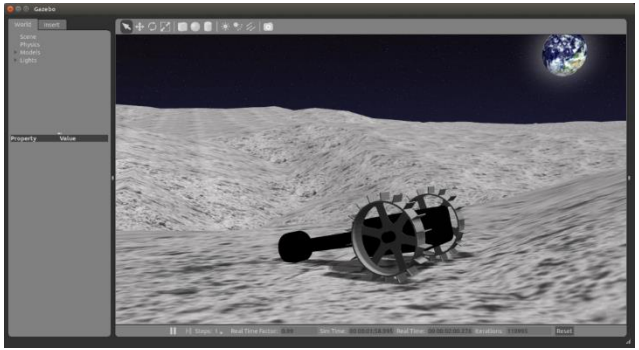
- 具备强大的物理引擎
- 高质量的图形渲染
- 方便的编程与图形接口
- 开源免费

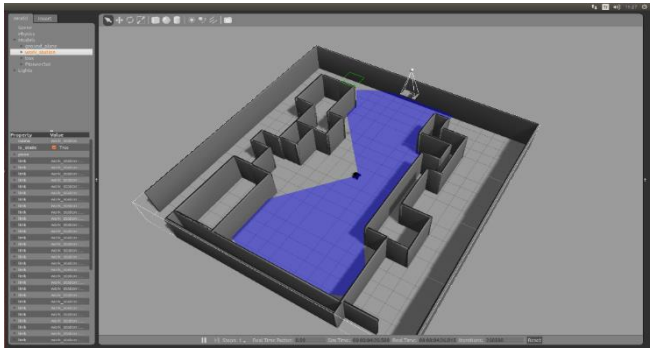
其典型应用场景包括

- 测试机器人算法
- 机器人的设计
- 现实情景下的回溯测试









4. ROS中的关键组件——Gazebo物理仿真环境

Features

Dynamics Simulation

Access multiple high-performance physics engines including ODE, Bullet, Simbody, and DART.

Advanced 3D Graphics

Utilizing OGRE, Gazebo provides realistic rendering of environments including high-quality lighting, shadows, and textures.

Sensors and Noise

Generate sensor data, optionally with noise, from laser range finders, 2D/3D cameras, Kinect style sensors, contact sensors, force-torque, and more.

Plugins

Develop custom plugins for robot, sensor, and environmental control. Plugins provide direct access to Gazebo's API.

Robot Models

Many robots are provided including PR2, Pioneer2 DX, iRobot Create, and TurtleBot. Or build your own using SDF.

TCP/IP Transport

Run simulation on remote servers, and interface to Gazebo through socket-based message passing using Google Protobufs.

Cloud Simulation

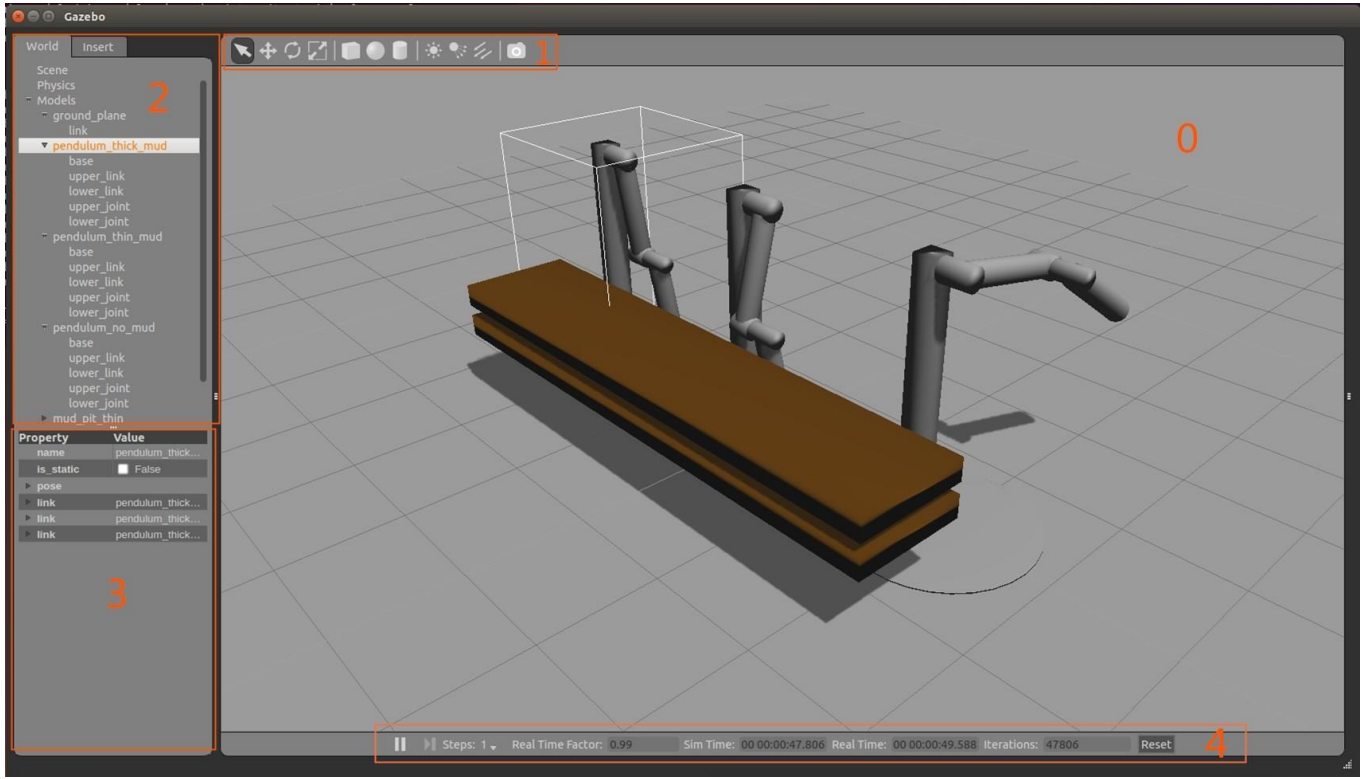
Use CloudSim to run Gazebo on Amazon, Softlayer, or your own OpenStack instance.

Command Line Tools

Extensive command line tools facilitate simulation introspection and control.

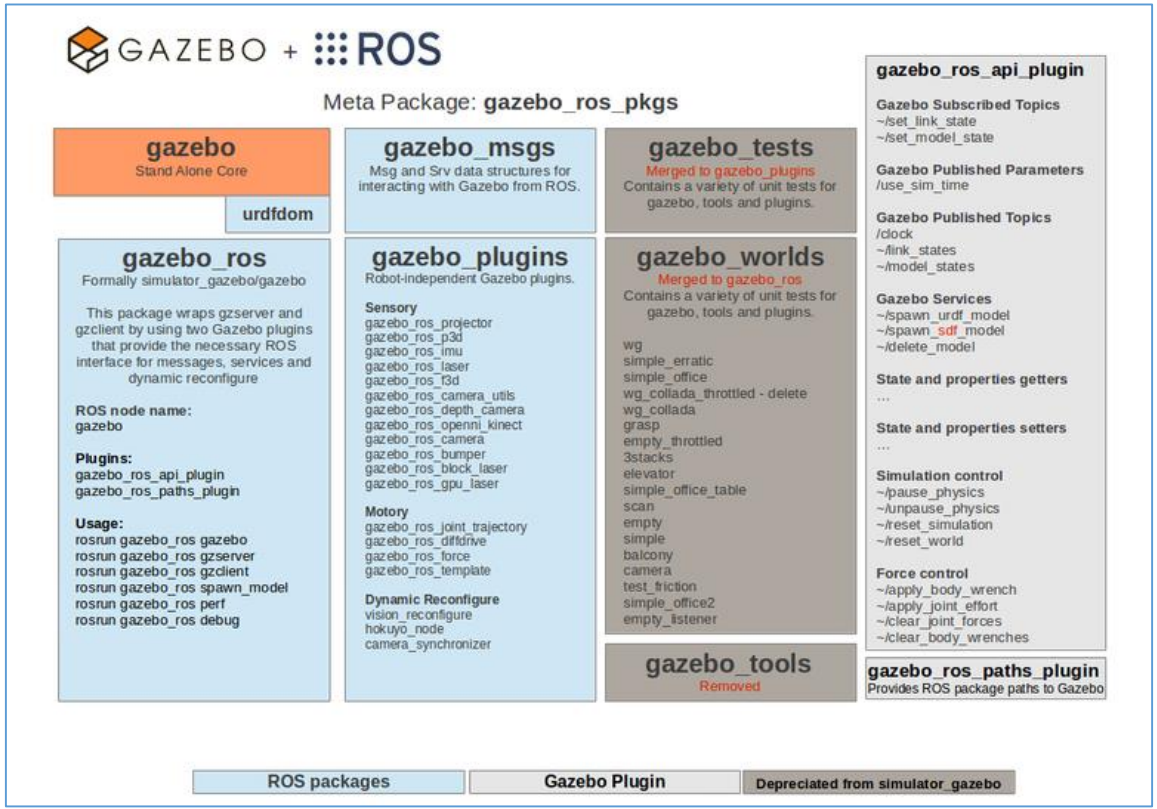
4. ROS中的关键组件——Gazebo物理仿真环境

- 0 ：3D视图区
- 1 ：工具栏
- 2 ：模型列表
- 3 ：模型属性项
- 4 ：时间显示区



4. ROS中的关键组件——Gazebo物理仿真环境

- **gazebo_ros** - 主要用于gazebo接口封装、gazebo服务端和客户端的启动、URDF模型生成等。
- **gazebo_msgs** - 是gazebo的Msg和Srv数据结构。
- **gazebo_plugins** - 用于gazebo的通用传感器插件。
- **gazebo_ros_api_plugin** 和 **gazebo_ros_path_plugin** 这两个Gazebo的插件实现接口封装。



4. ROS中的关键组件——Gazebo物理仿真环境

创建仿真环境

配置机器人模型

开始仿真

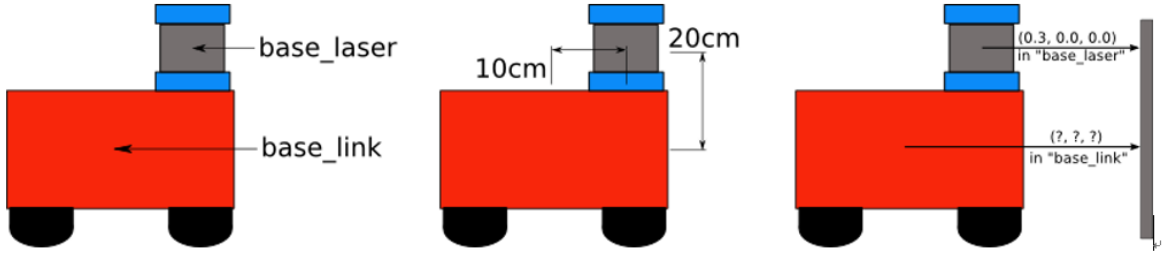
如何使用Gazebo进行仿真

小结

创建工作空间	• mkdir -p ~/catkin_ws/src • catkin_init_workspace • source devel/setup.bash • catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
ROS通信编程	• 话题编程：advertise / subscribe • 服务编程：advertiseService / serviceClient • 动作编程：Server / Client / waitForServer / sendGoal
分布式通信	• 设置IP地址，确保底层链路的联通 • 在从机端设置ROS_MASTER_URI，让从机找到ROS Master
关键组件	• Launch文件：多节点的配置和启动 • TF坐标变换：管理ROS中的坐标系统 • QT工具箱：rqt_console、rqt_graph、rqt_plot、rqt_reconfigure • Rviz可视化平台：数据可视化、插件机制 • Gazebo三维物理仿真环境：机器人仿真、功能/算法验证

作业

1. 创建一个工作空间和功能包，然后在功能包中完成如下工作，并且使用launch文件启动涉及的节点；
2. 话题与服务编程：通过代码新生一只海龟，放置在（5，5）点，命名为“turtle2”；通过代码订阅turtle2的实时位置并在终端打印；控制turtle2实现旋转运动；
3. 动作编程：客户端发送一个运动目标，模拟机器人运动到目标位置的过程，包含服务端和客户端的代码实现，要求带有实时位置反馈；
4. TF编程：广播并监听机器人的坐标变换，已知激光雷达和机器人底盘的坐标关系，求解激光雷达数据在底盘坐标系下的坐标值。



扩展阅读

- ROS Tutorials
http://wiki.ros.org/ROS/Tutorials

- ROS APIs (C++ / Python)
http://wiki.ros.org/APIs

- ROS Launch
http://wiki.ros.org/roslaunch/XML

- 《Introduction to Robotics Mechanics and Control 3rd edition》
《机器人学导论（第三版）》

感谢各位聆听！

Thanks for Listening