**第 10 章 ROS进阶**

在本教程的第二章内容介绍了ROS的核心实现:通信机制 ——话题通信、服务通信和参数服务器。三者结合可以满足ROS中的大多数数据传输相关的应用场景，但是在一些特定场景下可能就有些力不从心了，本章主要介绍之前的通信机制存在的问题以及对应的优化策略，本章主要内容如下:

* action通信；
* 动态参数；
* pluginlib；
* nodelet。

本章预期达成的学习目标:

* 了解服务通信应用的局限性(action的应用场景)，熟练掌握action的理论模型与实现流程；
* 了解参数服务器应用的局限性(动态配置参数的应用场景)，熟练掌握动态配置参数的实现流程；
* 了解插件的概念以及使用流程；
* 了解nodelet的应用场景以及使用流程。

## 10.1 action通信

关于action通信，我们先从之前导航中的应用场景开始介绍，描述如下:

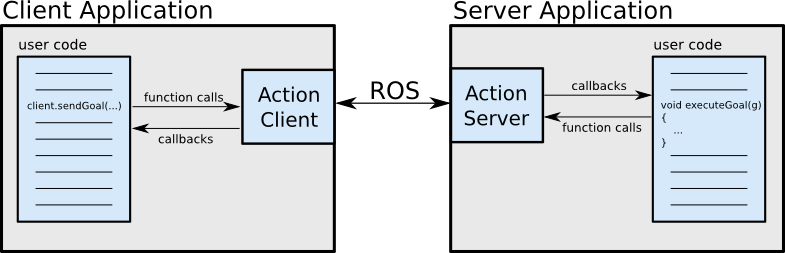
机器人导航到某个目标点,此过程需要一个节点A发布目标信息，然后一个节点B接收到请求并控制移动，最终响应目标达成状态信息。

乍一看，这好像是服务通信实现，因为需求中要A发送目标，B执行并返回结果，这是一个典型的基于请求响应的应答模式，不过，如果只是使用基本的服务通信实现，存在一个问题：**导航是一个过程，是耗时操作，如果使用服务通信，那么只有在导航结束时，才会产生响应结果，而在导航过程中，节点A是不会获取到任何反馈的，从而可能出现程序"假死"的现象，过程的不可控意味着不良的用户体验，以及逻辑处理的缺陷(比如:导航中止的需求无法实现)。**更合理的方案应该是:导航过程中，可以连续反馈当前机器人状态信息，当导航终止时，再返回最终的执行结果。在ROS中，该实现策略称之为:**action 通信**。

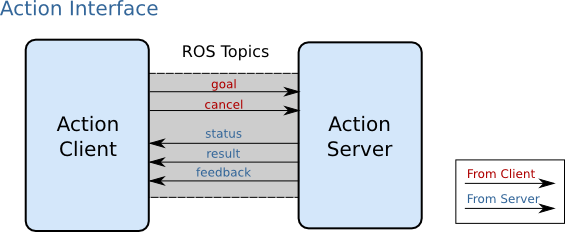
#### ****概念****

在ROS中提供了actionlib功能包集，用于实现 action 通信。action 是一种类似于服务通信的实现，其实现模型也包含请求和响应，但是不同的是，在请求和响应的过程中，服务端还可以连续的反馈当前任务进度，客户端可以接收连续反馈并且还可以取消任务。

**action结构图解:**



**action通信接口图解:**



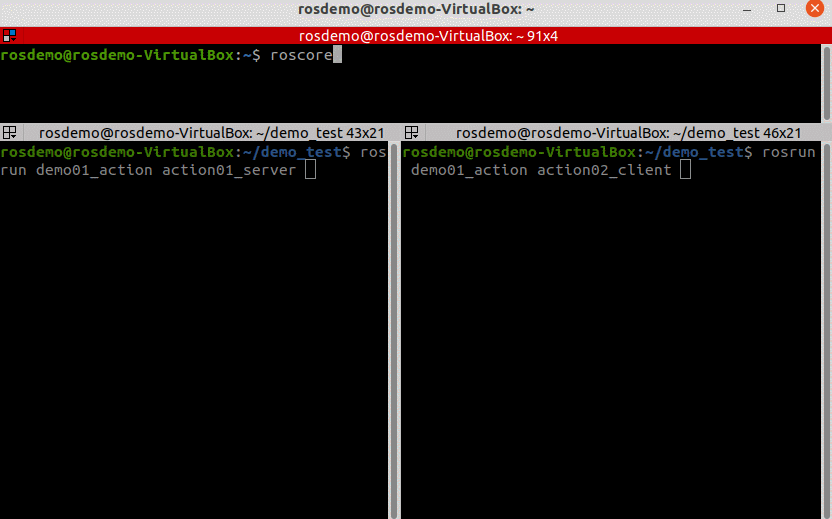
* goal:目标任务;
* cacel:取消任务;
* status:服务端状态;
* result:最终执行结果(只会发布一次);
* feedback:连续反馈(可以发布多次)。

#### ****作用****

一般适用于耗时的请求响应场景,用以获取连续的状态反馈。

#### ****案例****

创建两个ROS 节点，服务器和客户端，客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，为了良好的用户体验，需要服务器在计算过程中，每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。



**另请参考:**

* <http://wiki.ros.org/actionlib>
* <http://wiki.ros.org/actionlib_tutorials/Tutorials>

### 10.1.1action通信自定义action文件

action、srv、msg 文件内的可用数据类型一致，且三者实现流程类似:

1. 按照固定格式创建action文件；
2. 编辑配置文件；
3. 编译生成中间文件。

#### 1.定义action文件

首先新建功能包，并导入依赖: roscpp rospy std\_msgs actionlib actionlib\_msgs；

然后功能包下新建 action 目录，新增 Xxx.action(比如:AddInts.action)。

action 文件内容组成分为三部分:请求目标值、最终响应结果、连续反馈，三者之间使用---分割示例内容如下:

#目标值

int32 num

---

#最终结果

int32 result

---

#连续反馈

float64 progress\_bar

Copy

#### 2.编辑配置文件

**CMakeLists.txt**

find\_package

(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

actionlib

actionlib\_msgs

)

Copy

add\_action\_files(

FILES

AddInts.action

)

Copy

generate\_messages(

DEPENDENCIES

std\_msgs

actionlib\_msgs

)

Copy

catkin\_package(

# INCLUDE\_DIRS include

# LIBRARIES demo04\_action

CATKIN\_DEPENDS roscpp rospy std\_msgs actionlib actionlib\_msgs

# DEPENDS system\_lib

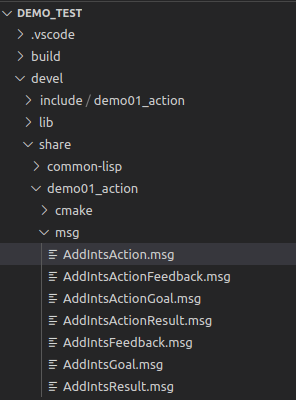
)

Copy

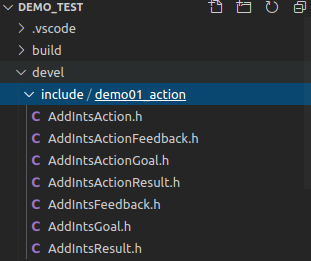
#### 3.编译

编译后会生成一些中间文件。

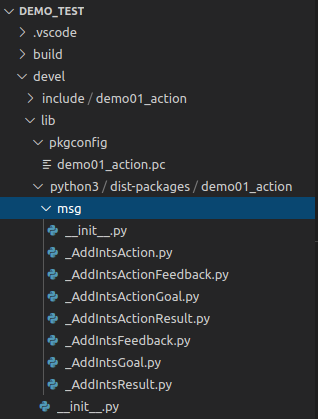
msg文件(.../工作空间/devel/share/包名/msg/xxx.msg):



C++ 调用的文件(.../工作空间/devel/include/包名/xxx.h):



Python 调用的文件(.../工作空间/devel/lib/python3/dist-packages/包名/msg/xxx.py):



### 10.1.2 action通信自定义action文件调用A(C++)

**需求:**

创建两个ROS 节点，服务器和客户端，客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，为了良好的用户体验，需要服务器在计算过程中，每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

**流程:**

1. 编写action服务端实现；
2. 编写action客户端实现；
3. 编辑CMakeLists.txt；
4. 编译并执行。

#### 0.vscode配置

需要像之前自定义 msg 实现一样配置c\_cpp\_properies.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/xxx/yyy工作空间/devel/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

Copy

#### 1.服务端

#include "ros/ros.h"

#include "actionlib/server/simple\_action\_server.h"

#include "demo01\_action/AddIntsAction.h"

/\*

需求:

创建两个ROS节点，服务器和客户端，

客户端可以向服务器发送目标数据N（一个整型数据）

服务器会计算1到N之间所有整数的和，这是一个循环累加的过程，返回给客户端，

这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用action实现。

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action服务对象;

5.处理请求,产生反馈与响应;

6.spin().

\*/

typedef actionlib::SimpleActionServer<demo01\_action::AddIntsAction> Server;

void cb(const demo01\_action::AddIntsGoalConstPtr &goal,Server\* server){

//获取目标值

int num = goal->num;

ROS\_INFO("目标值:%d",num);

//累加并响应连续反馈

int result = 0;

demo01\_action::AddIntsFeedback feedback;//连续反馈

ros::Rate rate(10);//通过频率设置休眠时间

for (int i = 1; i <= num; i++)

{

result += i;

//组织连续数据并发布

feedback.progress\_bar = i / (double)num;

server->publishFeedback(feedback);

rate.sleep();

}

//设置最终结果

demo01\_action::AddIntsResult r;

r.result = result;

server->setSucceeded(r);

ROS\_INFO("最终结果:%d",r.result);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ROS\_INFO("action服务端实现");

// 2.初始化ROS节点;

ros::init(argc,argv,"AddInts\_server");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action服务对象;

/\*SimpleActionServer(ros::NodeHandle n,

std::string name,

boost::function<void (const demo01\_action::AddIntsGoalConstPtr &)> execute\_callback,

bool auto\_start)

\*/

// actionlib::SimpleActionServer<demo01\_action::AddIntsAction> server(....);

Server server(nh,"addInts",boost::bind(&cb,\_1,&server),false);

server.start();

// 5.处理请求,产生反馈与响应;

// 6.spin().

ros::spin();

return 0;

}

Copy

**PS:**

可以先配置CMakeLists.tx文件并启动上述action服务端，然后通过 rostopic 查看话题，向action相关话题发送消息，或订阅action相关话题的消息。

#### 2.客户端

#include "ros/ros.h"

#include "actionlib/client/simple\_action\_client.h"

#include "demo01\_action/AddIntsAction.h"

/\*

需求:

创建两个ROS节点，服务器和客户端，

客户端可以向服务器发送目标数据N（一个整型数据）

服务器会计算1到N之间所有整数的和，这是一个循环累加的过程，返回给客户端，

这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用action实现。

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action客户端对象;

5.发送目标，处理反馈以及最终结果;

6.spin().

\*/

typedef actionlib::SimpleActionClient<demo01\_action::AddIntsAction> Client;

//处理最终结果

void done\_cb(const actionlib::SimpleClientGoalState &state, const demo01\_action::AddIntsResultConstPtr &result){

if (state.state\_ == state.SUCCEEDED)

{

ROS\_INFO("最终结果:%d",result->result);

} else {

ROS\_INFO("任务失败！");

}

}

//服务已经激活

void active\_cb(){

ROS\_INFO("服务已经被激活....");

}

//处理连续反馈

void feedback\_cb(const demo01\_action::AddIntsFeedbackConstPtr &feedback){

ROS\_INFO("当前进度:%.2f",feedback->progress\_bar);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化ROS节点;

ros::init(argc,argv,"AddInts\_client");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action客户端对象;

// SimpleActionClient(ros::NodeHandle & n, const std::string & name, bool spin\_thread = true)

// actionlib::SimpleActionClient<demo01\_action::AddIntsAction> client(nh,"addInts");

Client client(nh,"addInts",true);

//等待服务启动

client.waitForServer();

// 5.发送目标，处理反馈以及最终结果;

/\*

void sendGoal(const demo01\_action::AddIntsGoal &goal,

boost::function<void (const actionlib::SimpleClientGoalState &state, const demo01\_action::AddIntsResultConstPtr &result)> done\_cb,

boost::function<void ()> active\_cb,

boost::function<void (const demo01\_action::AddIntsFeedbackConstPtr &feedback)> feedback\_cb)

\*/

demo01\_action::AddIntsGoal goal;

goal.num = 10;

client.sendGoal(goal,&done\_cb,&active\_cb,&feedback\_cb);

// 6.spin().

ros::spin();

return 0;

}

Copy

**PS:**等待服务启动，只可以使用client.waitForServer();,之前服务中等待启动的另一种方式ros::service::waitForService("addInts");不适用

#### 3.编译配置文件

add\_executable(action01\_server src/action01\_server.cpp)

add\_executable(action02\_client src/action02\_client.cpp)

...

add\_dependencies(action01\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

add\_dependencies(action02\_client ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

...

target\_link\_libraries(action01\_server

${catkin\_LIBRARIES}

)

target\_link\_libraries(action02\_client

${catkin\_LIBRARIES}

)

Copy

#### 4.执行

首先启动 roscore，然后分别启动action服务端与action客户端，最终运行结果与案例类似。

### 10.1.3 action通信自定义action文件调用(Python)

**需求:**

创建两个ROS 节点，服务器和客户端，客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，为了良好的用户体验，需要服务器在计算过程中，每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

**流程:**

1. 编写action服务端实现；
2. 编写action客户端实现；
3. 编辑CMakeLists.txt；
4. 编译并执行。

#### 0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"python.autoComplete.extraPaths": [

"/opt/ros/noetic/lib/python3/dist-packages",

"/xxx/yyy工作空间/devel/lib/python3/dist-packages"

]

}

Copy

#### 1.服务端

#! /usr/bin/env python

import rospy

import actionlib

from demo01\_action.msg import \*

"""

需求:

创建两个ROS 节点，服务器和客户端，

客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,

这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

流程:

1.导包

2.初始化 ROS 节点

3.使用类封装，然后创建对象

4.创建服务器对象

5.处理请求数据产生响应结果，中间还要连续反馈

6.spin

"""

class MyActionServer:

def \_\_init\_\_(self):

#SimpleActionServer(name, ActionSpec, execute\_cb=None, auto\_start=True)

self.server = actionlib.SimpleActionServer("addInts",AddIntsAction,self.cb,False)

self.server.start()

rospy.loginfo("服务端启动")

def cb(self,goal):

rospy.loginfo("服务端处理请求:")

#1.解析目标值

num = goal.num

#2.循环累加，连续反馈

rate = rospy.Rate(10)

sum = 0

for i in range(1,num + 1):

# 累加

sum = sum + i

# 计算进度并连续反馈

feedBack = i / num

rospy.loginfo("当前进度:%.2f",feedBack)

feedBack\_obj = AddIntsFeedback()

feedBack\_obj.progress\_bar = feedBack

self.server.publish\_feedback(feedBack\_obj)

rate.sleep()

#3.响应最终结果

result = AddIntsResult()

result.result = sum

self.server.set\_succeeded(result)

rospy.loginfo("响应结果:%d",sum)

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("action\_server\_p")

server = MyActionServer()

rospy.spin()

Copy

**PS:**

可以先配置CMakeLists.tx文件并启动上述action服务端，然后通过 rostopic 查看话题，向action相关话题发送消息，或订阅action相关话题的消息。

#### 2.客户端

#! /usr/bin/env python

import rospy

import actionlib

from demo01\_action.msg import \*

"""

需求:

创建两个ROS 节点，服务器和客户端，

客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,

这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

流程:

1.导包

2.初始化 ROS 节点

3.创建 action Client 对象

4.等待服务

5.组织目标对象并发送

6.编写回调, 激活、连续反馈、最终响应

7.spin

"""

def done\_cb(state,result):

if state == actionlib.GoalStatus.SUCCEEDED:

rospy.loginfo("响应结果:%d",result.result)

def active\_cb():

rospy.loginfo("服务被激活....")

def fb\_cb(fb):

rospy.loginfo("当前进度:%.2f",fb.progress\_bar)

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("action\_client\_p")

# 3.创建 action Client 对象

client = actionlib.SimpleActionClient("addInts",AddIntsAction)

# 4.等待服务

client.wait\_for\_server()

# 5.组织目标对象并发送

goal\_obj = AddIntsGoal()

goal\_obj.num = 10

client.send\_goal(goal\_obj,done\_cb,active\_cb,fb\_cb)

# 6.编写回调, 激活、连续反馈、最终响应

# 7.spin

rospy.spin()

Copy

#### 3.编辑配置文件

先为 Python 文件添加可执行权限:chmod +x \*.py

catkin\_install\_python(PROGRAMS

scripts/action01\_server\_p.py

scripts/action02\_client\_p.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

Copy

#### 4.执行

首先启动 roscore，然后分别启动action服务端与action客户端，最终运行结果与案例类似。

## 10.2动态参数

参数服务器的数据被修改时，如果节点不重新访问，那么就不能获取修改后的数据，例如在乌龟背景色修改的案例中，先启动乌龟显示节点，然后再修改参数服务器中关于背景色设置的参数，那么窗体的背景色是不会修改的，必须要重启乌龟显示节点才能生效。而一些特殊场景下，是要求要能做到动态获取的，也即，参数一旦修改，能够通知节点参数已经修改并读取修改后的数据，比如：

机器人调试时，需要修改机器人轮廓信息(长宽高)、传感器位姿信息....，如果这些信息存储在参数服务器中，那么意味着需要重启节点，才能使更新设置生效，但是希望修改完毕之后，某些节点能够即时更新这些参数信息。

在ROS中针对这种场景已经给出的解决方案: dynamic reconfigure 动态配置参数。

动态配置参数，之所以能够实现即时更新，因为被设计成 CS 架构，客户端修改参数就是向服务器发送请求，服务器接收到请求之后，读取修改后的是参数。

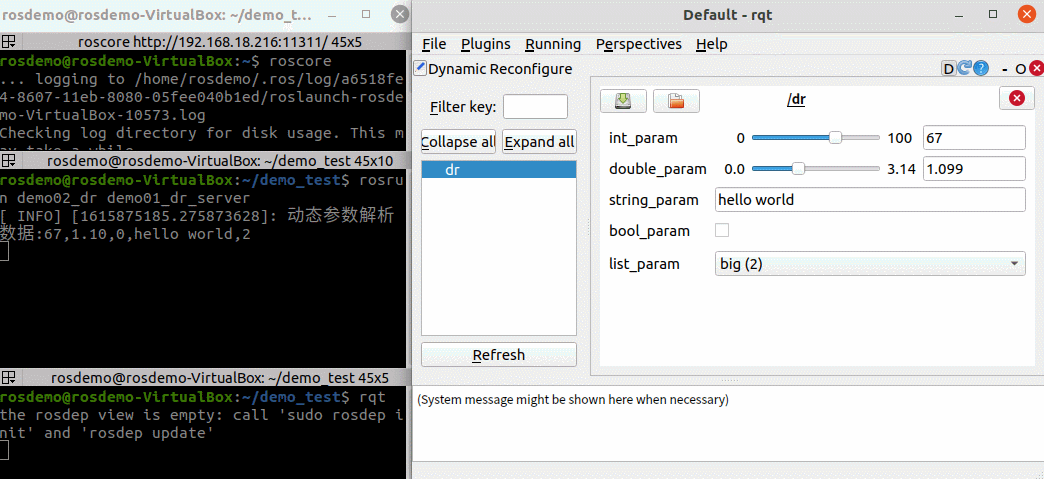
#### ****概念****

一种可以在运行时更新参数而无需重启节点的参数配置策略。

#### ****作用****

主要应用于需要动态更新参数的场景，比如参数调试、功能切换等。典型应用:导航时参数的动态调试。

#### ****案例****

编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

**另请参考:**

* <http://wiki.ros.org/dynamic_reconfigure>
* <http://wiki.ros.org/dynamic_reconfigure/Tutorials>

### 10.2.1 动态参数客户端

**需求:**

编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

**客户端实现流程:**

* 新建并编辑 .cfg 文件;
* 编辑CMakeLists.txt;
* 编译。

#### 1.新建功能包

新建功能包，添加依赖:roscpp rospy std\_msgs dynamic\_reconfigure。

#### 2.添加.cfg文件

新建 cfg 文件夹，添加 xxx.cfg 文件(并添加可执行权限)，cfg 文件其实就是一个 python 文件,用于生成参数修改的客户端(GUI)。

#! /usr/bin/env python

"""

4生成动态参数 int,double,bool,string,列表

5实现流程:

6 1.导包

7 2.创建生成器

8 3.向生成器添加若干参数

9 4.生成中间文件并退出

10

"""

# 1.导包

from dynamic\_reconfigure.parameter\_generator\_catkin import \*

PACKAGE = "demo02\_dr"

# 2.创建生成器

gen = ParameterGenerator()

# 3.向生成器添加若干参数

#add(name, paramtype, level, description, default=None, min=None, max=None, edit\_method="")

gen.add("int\_param",int\_t,0,"整型参数",50,0,100)

gen.add("double\_param",double\_t,0,"浮点参数",1.57,0,3.14)

gen.add("string\_param",str\_t,0,"字符串参数","hello world ")

gen.add("bool\_param",bool\_t,0,"bool参数",True)

many\_enum = gen.enum([gen.const("small",int\_t,0,"a small size"),

gen.const("mediun",int\_t,1,"a medium size"),

gen.const("big",int\_t,2,"a big size")

],"a car size set")

gen.add("list\_param",int\_t,0,"列表参数",0,0,2, edit\_method=many\_enum)

# 4.生成中间文件并退出

exit(gen.generate(PACKAGE,"dr\_node","dr"))

Copy

chmod +x xxx.cfg添加权限

#### 3.配置 CMakeLists.txt

generate\_dynamic\_reconfigure\_options(

cfg/mycar.cfg

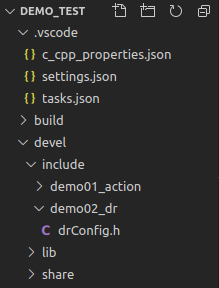
)

Copy

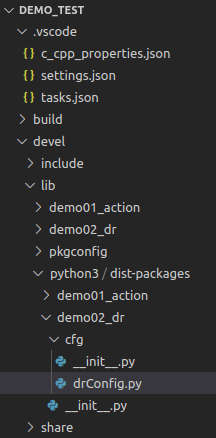
#### 4.编译

编译后会生成中间文件

C++ 需要调用的头文件:



Python需要调用的文件:



### 10.2.2 动态参数服务端A(C++)

**需求:**

编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

**服务端实现流程:**

* 新建并编辑 c++ 文件;
* 编辑CMakeLists.txt;
* 编译并执行。

#### 0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/xxx/yyy工作空间/devel/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

Copy

#### 1.服务器代码实现

新建cpp文件，内容如下:

#include "ros/ros.h"

#include "dynamic\_reconfigure/server.h"

#include "demo02\_dr/drConfig.h"

/\*

动态参数服务端: 参数被修改时直接打印

实现流程:

1.包含头文件

2.初始化 ros 节点

3.创建服务器对象

4.创建回调对象(使用回调函数，打印修改后的参数)

5.服务器对象调用回调对象

6.spin()

\*/

void cb(demo02\_dr::drConfig& config, uint32\_t level){

ROS\_INFO("动态参数解析数据:%d,%.2f,%d,%s,%d",

config.int\_param,

config.double\_param,

config.bool\_param,

config.string\_param.c\_str(),

config.list\_param

);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ros 节点

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

// 3.创建服务器对象

dynamic\_reconfigure::Server<demo02\_dr::drConfig> server;

// 4.创建回调对象(使用回调函数，打印修改后的参数)

dynamic\_reconfigure::Server<demo02\_dr::drConfig>::CallbackType cbType;

cbType = boost::bind(&cb,\_1,\_2);

// 5.服务器对象调用回调对象

server.setCallback(cbType);

// 6.spin()

ros::spin();

return 0;

}

Copy

#### 2.编译配置文件

add\_executable(demo01\_dr\_server src/demo01\_dr\_server.cpp)

...

add\_dependencies(demo01\_dr\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

...

target\_link\_libraries(demo01\_dr\_server

${catkin\_LIBRARIES}

)

Copy

#### 3.执行

先启动roscore

启动服务端:rosrun 功能包 xxxx

启动客户端:rosrun rqt\_gui rqt\_gui -s rqt\_reconfigure或rosrun rqt\_reconfigure rqt\_reconfigure

最终可以通过客户端提供的界面修改数据，并且修改完毕后，服务端会即时输出修改后的结果，最终运行结果与示例类似。

**PS:**ROS版本较新时，可能没有提供客户端相关的功能包导致rosrun rqt\_reconfigure rqt\_reconfigure调用会抛出异常。

### 10.2.3 动态参数服务端B(Python)

**需求:**

编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

**服务端实现流程:**

* 新建并编辑 Python 文件;
* 编辑CMakeLists.txt;
* 编译并执行。

#### 0.vscode配置

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"python.autoComplete.extraPaths": [

"/opt/ros/noetic/lib/python3/dist-packages",

"/xxx/yyy工作空间/devel/lib/python3/dist-packages"

]

}

Copy

#### 1.服务器代码实现

新建python文件，内容如下:

#! /usr/bin/env python

import rospy

from dynamic\_reconfigure.server import Server

from demo02\_dr.cfg import drConfig

"""

动态参数服务端: 参数被修改时直接打印

实现流程:

1.导包

2.初始化 ros 节点

3.创建服务对象

4.回调函数处理

5.spin

"""

# 回调函数

def cb(config,level):

rospy.loginfo("python 动态参数服务解析:%d,%.2f,%d,%s,%d",

config.int\_param,

config.double\_param,

config.bool\_param,

config.string\_param,

config.list\_param

)

return config

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ros 节点

rospy.init\_node("dr\_p")

# 3.创建服务对象

server = Server(drConfig,cb)

# 4.回调函数处理

# 5.spin

rospy.spin()

Copy

#### 2.编辑配置文件

先为 Python 文件添加可执行权限:chmod +x \*.py

catkin\_install\_python(PROGRAMS

scripts/demo01\_dr\_server\_p.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

Copy

#### 3.执行

先启动roscore

启动服务端:rosrun 功能包 xxxx.py

启动客户端:rosrun rqt\_gui rqt\_gui -s rqt\_reconfigure或rosrun rqt\_reconfigure rqt\_reconfigure

最终可以通过客户端提供的界面修改数据，并且修改完毕后，服务端会即时输出修改后的结果，最终运行结果与示例类似。

**PS:**ROS版本较新时，可能没有提供客户端相关的功能包导致rosrun rqt\_reconfigure rqt\_reconfigure调用会抛出异常。

## 10.3 pluginlib

**pluginlib**直译是插件库，所谓插件字面意思就是可插拔的组件，比如:以计算机为例，可以通过USB接口自由插拔的键盘、鼠标、U盘...都可以看作是插件实现，其基本原理就是通过规范化的USB接口协议实现计算机与USB设备的自由组合。同理，在软件编程中，插件是一种遵循一定规范的应用程序接口编写出来的程序，插件程序依赖于某个应用程序，且应用程序可以与不同的插件程序自由组合。在ROS中，也会经常使用到插件，场景如下:

1.导航插件:在导航中，涉及到路径规划模块，路径规划算法有多种，也可以自实现，导航应用时，可能需要测试不同算法的优劣以选择更合适的实现，这种场景下，ROS中就是通过插件的方式来实现不同算法的灵活切换的。

2.rviz插件:在rviz中已经提供了丰富的功能实现，但是即便如此，特定场景下，开发者可能需要实现某些定制化功能并集成到rviz中，这一集成过程也是基于插件的。

#### ****概念****

**pluginlib**是一个c++库， 用来从一个ROS功能包中加载和卸载插件(plugin)。插件是指从运行时库中动态加载的类。通过使用Pluginlib，不必将某个应用程序显式地链接到包含某个类的库，Pluginlib可以随时打开包含类的库，而不需要应用程序事先知道包含类定义的库或者头文件。

#### ****作用****

* 结构清晰；
* 低耦合，易修改，可维护性强；
* 可移植性强，更具复用性；
* 结构容易调整，插件可以自由增减；

**另请参考:**

* <http://wiki.ros.org/pluginlib>
* <http://wiki.ros.org/pluginlib/Tutorials/Writing%20and%20Using%20a%20Simple%20Plugin>

### 10.3.1 pluginlib使用

**需求:**

以插件的方式实现正多边形的相关计算。

**实现流程:**

1. 准备；
2. 创建基类；
3. 创建插件类；
4. 注册插件;
5. 构建插件库;
6. 使插件可用于ROS工具链;
   * 配置xml
   * 导出插件
7. 使用插件;
8. 执行。

##### 1.准备

创建功能包xxx导入依赖: roscpp pluginlib。

在 VSCode中需要配置 .vascode/c\_cpp\_properties.json文件中关于 includepath 选项的设置。

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/.../yyy工作空间/功能包/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

Copy

##### 2.创建基类

在 xxx/include/xxx下新建C++头文件: polygon\_base.h，所有的插件类都需要继承此基类，内容如下:

#ifndef XXX\_POLYGON\_BASE\_H\_

#define XXX\_POLYGON\_BASE\_H\_

namespace polygon\_base

{

class RegularPolygon

{

public:

virtual void initialize(double side\_length) = 0;

virtual double area() = 0;

virtual ~RegularPolygon(){}

protected:

RegularPolygon(){}

};

};

#endif

Copy

**PS:**基类必须提供无参构造函数，所以关于多边形的边长没有通过构造函数而是通过单独编写的initialize函数传参。

##### 3.创建插件

在 xxx/include/xxx下新建C++头文件:polygon\_plugins.h，内容如下:

#ifndef XXX\_POLYGON\_PLUGINS\_H\_

#define XXX\_POLYGON\_PLUGINS\_H\_

#include <xxx/polygon\_base.h>

#include <cmath>

namespace polygon\_plugins

{

class Triangle : public polygon\_base::RegularPolygon

{

public:

Triangle(){}

void initialize(double side\_length)

{

side\_length\_ = side\_length;

}

double area()

{

return 0.5 \* side\_length\_ \* getHeight();

}

double getHeight()

{

return sqrt((side\_length\_ \* side\_length\_) - ((side\_length\_ / 2) \* (side\_length\_ / 2)));

}

private:

double side\_length\_;

};

class Square : public polygon\_base::RegularPolygon

{

public:

Square(){}

void initialize(double side\_length)

{

side\_length\_ = side\_length;

}

double area()

{

return side\_length\_ \* side\_length\_;

}

private:

double side\_length\_;

};

};

#endif

Copy

该文件中创建了正方形与三角形两个衍生类继承基类。

##### 4.注册插件

在 src 目录下新建 polygon\_plugins.cpp 文件，内容如下:

//pluginlib 宏，可以注册插件类

#include <pluginlib/class\_list\_macros.h>

#include <xxx/polygon\_base.h>

#include <xxx/polygon\_plugins.h>

//参数1:衍生类 参数2:基类

PLUGINLIB\_EXPORT\_CLASS(polygon\_plugins::Triangle, polygon\_base::RegularPolygon)

PLUGINLIB\_EXPORT\_CLASS(polygon\_plugins::Square, polygon\_base::RegularPolygon)

Copy

该文件会将两个衍生类注册为插件。

##### 5.构建插件库

在 CMakeLists.txt 文件中设置内容如下:

include\_directories(include)

add\_library(polygon\_plugins src/polygon\_plugins.cpp)

Copy

至此，可以调用 catkin\_make 编译，编译完成后，在工作空间/devel/lib目录下，会生成相关的 .so 文件。

##### 6.使插件可用于ROS工具链

6.1配置xml

功能包下新建文件:polygon\_plugins.xml,内容如下:

<!-- 插件库的相对路径 -->

<library path="lib/libpolygon\_plugins">

<!-- type="插件类" base\_class\_type="基类" -->

<class type="polygon\_plugins::Triangle" base\_class\_type="polygon\_base::RegularPolygon">

<!-- 描述信息 -->

<description>This is a triangle plugin.</description>

</class>

<class type="polygon\_plugins::Square" base\_class\_type="polygon\_base::RegularPolygon">

<description>This is a square plugin.</description>

</class>

</library>

Copy

6.2导出插件

package.xml文件中设置内容如下:

<export>

<xxx plugin="${prefix}/polygon\_plugins.xml" />

</export>

Copy

标签<xxx />的名称应与基类所属的功能包名称一致，plugin属性值为上一步中创建的xml文件。

编译后，可以调用rospack plugins --attrib=plugin xxx命令查看配置是否正常，如无异常，会返回 .xml 文件的完整路径，这意味着插件已经正确的集成到了ROS工具链。

##### 7.使用插件

src 下新建c++文件:polygon\_loader.cpp，内容如下:

//类加载器相关的头文件

#include <pluginlib/class\_loader.h>

#include <xxx/polygon\_base.h>

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

{

//类加载器 -- 参数1:基类功能包名称 参数2:基类全限定名称

pluginlib::ClassLoader<polygon\_base::RegularPolygon> poly\_loader("xxx", "polygon\_base::RegularPolygon");

try

{

//创建插件类实例 -- 参数:插件类全限定名称

boost::shared\_ptr<polygon\_base::RegularPolygon> triangle = poly\_loader.createInstance("polygon\_plugins::Triangle");

triangle->initialize(10.0);

boost::shared\_ptr<polygon\_base::RegularPolygon> square = poly\_loader.createInstance("polygon\_plugins::Square");

square->initialize(10.0);

ROS\_INFO("Triangle area: %.2f", triangle->area());

ROS\_INFO("Square area: %.2f", square->area());

}

catch(pluginlib::PluginlibException& ex)

{

ROS\_ERROR("The plugin failed to load for some reason. Error: %s", ex.what());

}

return 0;

}

Copy

##### 8.执行

修改CMakeLists.txt文件，内容如下:

add\_executable(polygon\_loader src/polygon\_loader.cpp)

target\_link\_libraries(polygon\_loader ${catkin\_LIBRARIES})

Copy

编译然后执行:polygon\_loader，结果如下:

[ INFO] [WallTime: 1279658450.869089666]: Triangle area: 43.30

[ INFO] [WallTime: 1279658450.869138007]: Square area: 100.00

## 10.4 nodelet

ROS通信是基于Node(节点)的，Node使用方便、易于扩展，可以满足ROS中大多数应用场景，但是也存在一些局限性，由于一个Node启动之后独占一根进程，不同Node之间数据交互其实是不同进程之间的数据交互，当传输类似于图片、点云的大容量数据时，会出现延时与阻塞的情况，比如：

现在需要编写一个相机驱动，在该驱动中有两个节点实现:其中节点A负责发布原始图像数据，节点B订阅原始图像数据并在图像上标注人脸。如果节点A与节点B仍按照之前实现，两个节点分别对应不同的进程，在两个进程之间传递容量可观图像数据，可能就会出现延时的情况，那么该如何优化呢？

ROS中给出的解决方案是:Nodelet，通过Nodelet可以将多个节点集成进一个进程。

#### ****概念****

nodelet软件包旨在提供在同一进程中运行多个算法(节点)的方式，不同算法之间通过传递指向数据的指针来代替了数据本身的传输(类似于编程传值与传址的区别)，从而实现零成本的数据拷贝。

nodelet功能包的核心实现也是插件，是对插件的进一步封装:

* 不同算法被封装进插件类，可以像单独的节点一样运行；
* 在该功能包中提供插件类实现的基类:Nodelet；
* 并且提供了加载插件类的类加载器:NodeletLoader。

#### ****作用****

应用于大容量数据传输的场景，提高节点间的数据交互效率，避免延时与阻塞。

**另请参考:**

* <http://wiki.ros.org/nodelet/>
* <http://wiki.ros.org/nodelet/Tutorials/Running%20a%20nodelet>
* <https://github.com/ros/common_tutorials/tree/noetic-devel/nodelet_tutorial_math>

### 10.4.1 使用演示

在ROS中内置了nodelet案例，我们先以该案例演示nodelet的基本使用语法，基本流程如下:

1. 案例简介；
2. nodelet基本使用语法；
3. 内置案例调用。

#### 1.案例简介

以“ros- [ROS\_DISTRO] -desktop-full”命令安装ROS时，nodelet默认被安装，如未安装，请调用如下命令自行安装:

sudo apt install ros-<<ROS\_DISTRO>>-nodelet-tutorial-math

Copy

在该案例中，定义了一个Nodelet插件类:Plus，这个节点可以订阅一个数字，并将订阅到的数字与参数服务器中的 value 参数相加后再发布。

**需求:**再同一线程中启动两个Plus节点A与B，向A发布一个数字，然后经A处理后，再发布并作为B的输入，最后打印B的输出。

#### 2.nodelet 基本使用语法

使用语法如下:

nodelet load pkg/Type manager - Launch a nodelet of type pkg/Type on manager manager

nodelet standalone pkg/Type - Launch a nodelet of type pkg/Type in a standalone node

nodelet unload name manager - Unload a nodelet a nodelet by name from manager

nodelet manager - Launch a nodelet manager node

Copy

#### 3.内置案例调用

##### 1.启动roscore

roscore

Copy

##### 2.启动manager

rosrun nodelet nodelet manager \_\_name:=mymanager

Copy

\_\_name:= 用于设置管理器名称。

##### 3.添加nodelet节点

添加第一个节点:

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n1 \_value:=100

Copy

添加第二个节点:

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n2 \_value:=-50 /n2/in:=/n1/out

Copy

PS: 解释

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n1 \_value:=100

1. rosnode list 查看，nodelet 的节点名称是: /n1；
2. rostopic list 查看，订阅的话题是: /n1/in，发布的话题是: /n1/out；
3. rosparam list查看，参数名称是: /n1/value。

rosrun nodelet nodelet standalone nodelet\_tutorial\_math/Plus mymanager \_\_name:=n2 \_value:=-50 /n2/in:=/n1/out

1. 第二个nodelet 与第一个同理；
2. 第二个nodelet 订阅的话题由 /n2/in 重映射为 /n1/out。

**优化:**也可以将上述实现集成进launch文件:

<launch>

<!-- 设置nodelet管理器 -->

<node pkg="nodelet" type="nodelet" name="mymanager" args="manager" output="screen" />

<!-- 启动节点1，名称为 n1, 参数 /n1/value 为100 -->

<node pkg="nodelet" type="nodelet" name="n1" args="load nodelet\_tutorial\_math/Plus mymanager" output="screen" >

<param name="value" value="100" />

</node>

<!-- 启动节点2，名称为 n2, 参数 /n2/value 为-50 -->

<node pkg="nodelet" type="nodelet" name="n2" args="load nodelet\_tutorial\_math/Plus mymanager" output="screen" >

<param name="value" value="-50" />

<remap from="/n2/in" to="/n1/out" />

</node>

</launch>

Copy

##### 4.执行

向节点n1发布消息:

rostopic pub -r 10 /n1/in std\_msgs/Float64 "data: 10.0"

Copy

打印节点n2发布的消息:

rostopic echo /n2/out

Copy

最终输出结果应该是:60。

### 10.4.2 nodelet实现

nodelet本质也是插件，实现流程与插件实现流程类似，并且更为简单，不需要自定义接口，也不需要使用类加载器加载插件类。

**需求:**参考 nodelet 案例，编写 nodelet 插件类，可以订阅输入数据，设置参数，发布订阅数据与参数相加的结果。

**流程:**

1. 准备；
2. 创建插件类并注册插件;
3. 构建插件库;
4. 使插件可用于ROS工具链；
5. 执行。

#### 1.准备

新建功能包，导入依赖: roscpp、nodelet；

#### 2.创建插件类并注册插件

#include "nodelet/nodelet.h"

#include "pluginlib/class\_list\_macros.h"

#include "ros/ros.h"

#include "std\_msgs/Float64.h"

namespace nodelet\_demo\_ns {

class MyPlus: public nodelet::Nodelet {

public:

MyPlus(){

value = 0.0;

}

void onInit(){

//获取 NodeHandle

ros::NodeHandle& nh = getPrivateNodeHandle();

//从参数服务器获取参数

nh.getParam("value",value);

//创建发布与订阅对象

pub = nh.advertise<std\_msgs::Float64>("out",100);

sub = nh.subscribe<std\_msgs::Float64>("in",100,&MyPlus::doCb,this);

}

//回调函数

void doCb(const std\_msgs::Float64::ConstPtr& p){

double num = p->data;

//数据处理

double result = num + value;

std\_msgs::Float64 r;

r.data = result;

//发布

pub.publish(r);

}

private:

ros::Publisher pub;

ros::Subscriber sub;

double value;

};

}

PLUGINLIB\_EXPORT\_CLASS(nodelet\_demo\_ns::MyPlus,nodelet::Nodelet)

Copy

#### 3.构建插件库

CMakeLists.txt配置如下：

...

add\_library(mynodeletlib

src/myplus.cpp

)

...

target\_link\_libraries(mynodeletlib

${catkin\_LIBRARIES}

)

Copy

编译后，会在 工作空间/devel/lib/先生成文件: libmynodeletlib.so。

#### 4.使插件可用于ROS工具链

##### 4.1配置xml

新建 xml 文件，名称自定义(比如:my\_plus.xml)，内容如下：

<library path="lib/libmynodeletlib">

<class name="demo04\_nodelet/MyPlus" type="nodelet\_demo\_ns::MyPlus" base\_class\_type="nodelet::Nodelet" >

<description>hello</description>

</class>

</library>

Copy

##### 4.2导出插件

<export>

<!-- Other tools can request additional information be placed here -->

<nodelet plugin="${prefix}/my\_plus.xml" />

</export>

Copy

#### 5.执行

可以通过launch文件执行nodelet，示例内容如下:

<launch>

<node pkg="nodelet" type="nodelet" name="my" args="manager" output="screen" />

<node pkg="nodelet" type="nodelet" name="p1" args="load demo04\_nodelet/MyPlus my" output="screen">

<param name="value" value="100" />

<remap from="/p1/out" to="con" />

</node>

<node pkg="nodelet" type="nodelet" name="p2" args="load demo04\_nodelet/MyPlus my" output="screen">

<param name="value" value="-50" />

<remap from="/p2/in" to="con" />

</node>

</launch>

Copy

运行launch文件，可以参考上一节方式向 p1发布数据，并订阅p2输出的数据，最终运行结果也与上一节类似。

## 10.5 本章小结

本章介绍了ROS中的一些进阶内容，主要内容如下:

* Action 通信；
* 动态参数；
* pluginlib；
* nodelet。

上述内容其实都是对之前通信机制缺陷的进一步优化：action较之于以往的服务通信是带有连续反馈的，更适用于耗时的请求响应场景；动态参数较之于参数服务器实现，则可以保证参数读取的实时性；最后，nodelet可以动态加载多个节点到同一进程，不再是一个节点独占一个进程，从而可以零成本的实现不同节点之间的数据交互，降低了数据传输的延时，提高了数据传输的效率；当然，nodelet是插件的应用之一，所以在介绍 nodelet 之前，我们又先学习了 pluginlib，借助 pluginlib 可以实现可插拔的设计，让程序更为灵活、易于扩展且方便维护。