# 3 ROS进阶

上节课我们介绍了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。

**3.1 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:

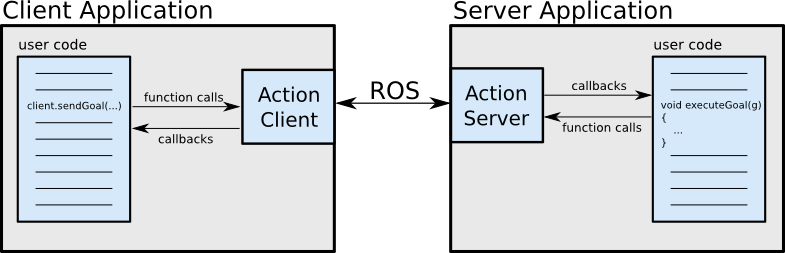
机器人导航到某个目标点,此过程需要一个节点A发布目标信息，然后一个节点B接收到请求并控制移动，最终响应“目标达成”状态信息。

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**。

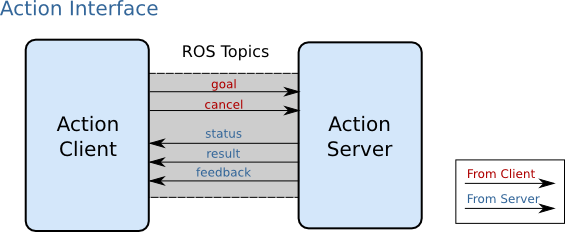
### 概念

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:**



\*机械臂经常会使用到“Action”通信。

基于“请求-响应”的，并且是一个“耗时”的过程。

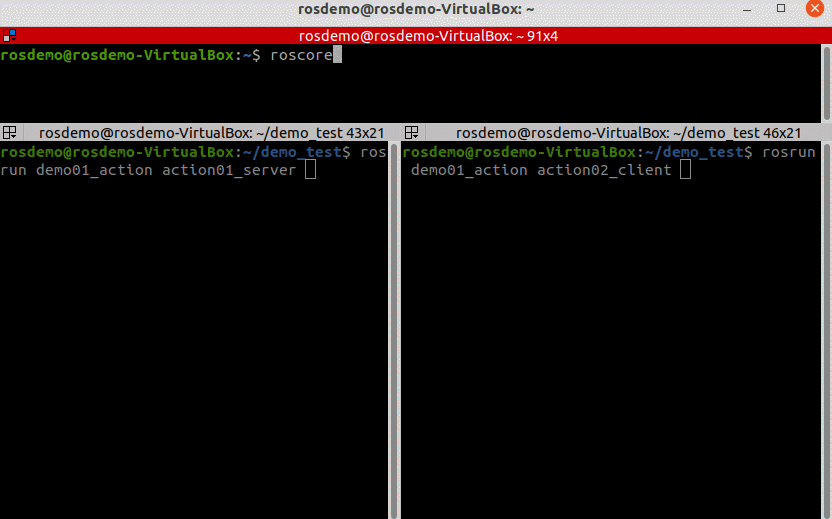
* goal:目标任务;
* cacel:取消任务;
* status:服务端状态;
* result:最终执行结果(只会发布一次);
* feedback:连续反馈(可以发布多次)。

### 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。



# 3.1.1action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

### 2.编辑配置文件

**CMakeLists.txt**

find\_package

(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

**actionlib**

**actionlib\_msgs**

)

add\_action\_files(

FILES

**AddInts.action**

)

generate\_messages(

DEPENDENCIES

std\_msgs

**actionlib\_msgs**

)

catkin\_package(

*# INCLUDE\_DIRS include*

*# LIBRARIES demo04\_action*

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

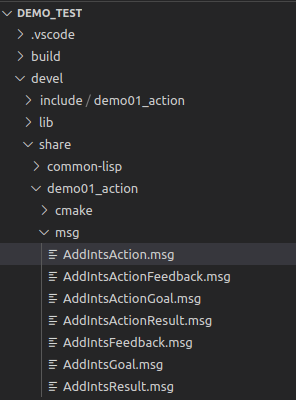
*# DEPENDS system\_lib*

)

### 3.编译

**编译后会生成一些中间文件。**

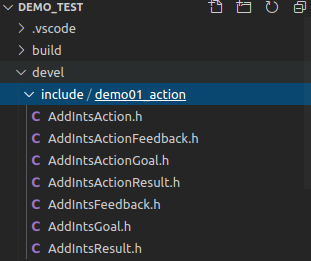
msg文件(.../工作空间/devel/share/包名/msg/xxx.msg):

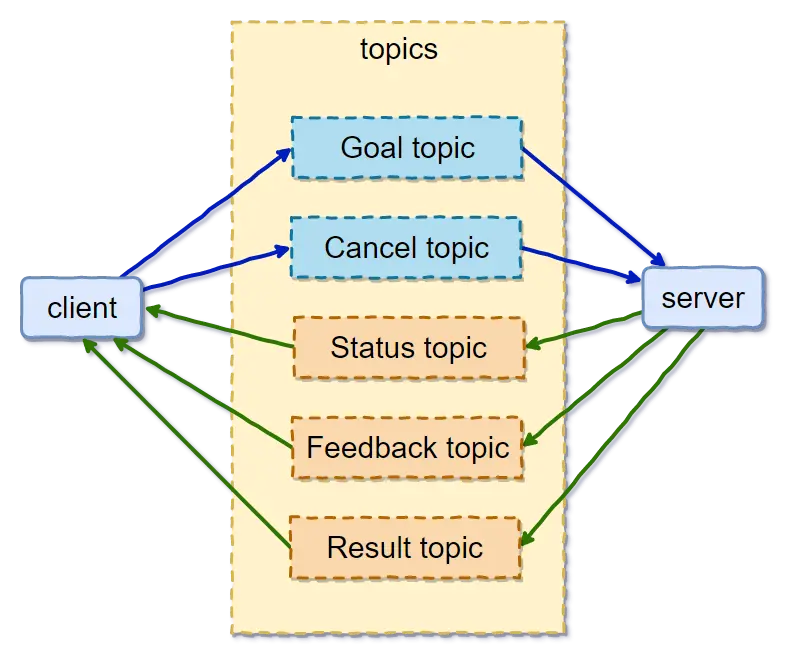


与创建的功能包相关的目录——demo01\_action。展开msg子目录，可以看到生成的中间文件——一些msg文件，以“action文件名”开头的，<action文件名>Action.msg、<action文件名>Goal.msg、<action文件名>Feedback.msg、<action文件名>Result.msg。<action文件名>ActionGoal.msg、<action文件名>ActionResult.msg、<action文件名>ActionFeedback.msg。

对应的头文件在功能包的devel/include/<功能包>/目录下。

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





# 3.1.2 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。

在客户端和服务端调用“.action”文件编译生成的头文件，我们需要先设置一下vscode。

### 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

}

### 1.服务端

**#include** "ros/ros.h"

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

**#include** "demo\_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**<**demo\_action**::**AddIntsAction**>** Server;

**void** **cb**(**const** demo\_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;

demo\_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*

demo\_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;

}

**PS:**

可以先配置CMakeLists.txt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**<**demo\_action**::**AddIntsAction**>** Client;

*//处理最终结果*

**void** **done\_cb**(**const** actionlib**::**SimpleClientGoalState **&**state, **const** demo\_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** demo\_action**::AddInts****FeedbackConstPtr** **&**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 demo\_action::AddIntsGoal &goal,*

*boost::function<void (const actionlib::SimpleClientGoalState &state, const demo\_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;

}

**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(action\_server src/action\_server.cpp)

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

...

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

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

...

target\_link\_libraries(action\_server

${catkin\_LIBRARIES}

)

target\_link\_libraries(action\_client

${catkin\_LIBRARIES}

)

### 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。

SimpleActionServer(ros::NodeHandle n,

std::string name,

boost::function<void (const

demo\_action:::AddIntsGoalConstPtr &)> execute\_callback,

bool auto\_start);

//手动启动，需要调用该函数来启动服务

server.start();

# 3.2参数服务器(std::map<string, t>)

参数服务器在ROS中主要用于实现不同节点之间的数据共享。参数服务器相当于是独立于所有节点的一个公共容器，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往其中存储数据，参数服务器一般适用于存在数据共享的一些应用场景。

我们可以认为参数是节点中使用的全局变量。参数的用途与Windows程序中的\*.ini配置文件非常类似。默认情况下，这些设置值是指定的，有需要时可以从外部读取或写入参数。特别是，由于可以通过使用来自外部的写入功能来实时地改变设置值，因此它是非常有用的，因为它可以灵活地应对多变的情况。尽管参数严格的来说并不是消息通信，从大的框架来看，参数也可以看作一种消息通信，因为它们使用消息。例如，用户可以设置要连接的USB端口、摄像机色彩校正值以及速度和命令的最大值和最小值。

## 概念

以共享的方式实现不同节点之间数据交互的通信模式。

## 作用

存储一些多节点共享的数据，类似于全局变量。

## 案例

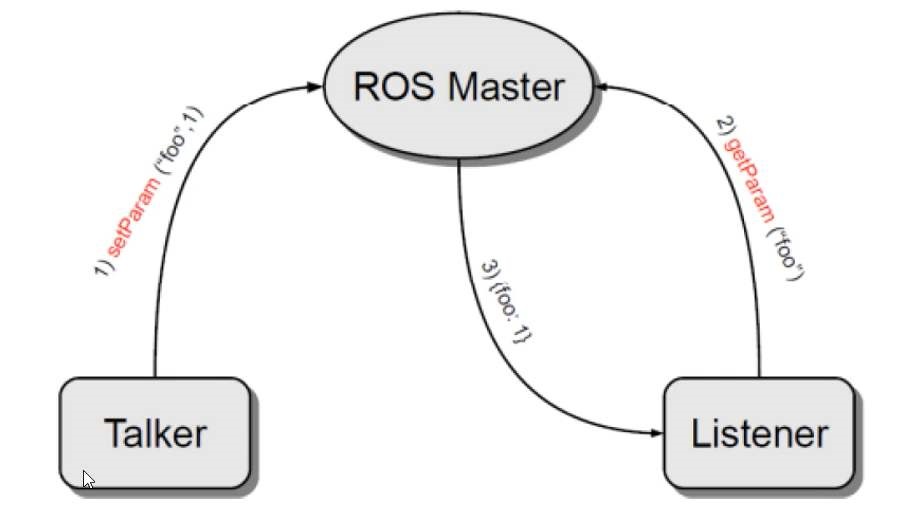
实现参数增删改查操作。

# 3.2.1 参数服务器理论模型

参数服务器实现是最为简单的，该模型如下图所示,该模型中涉及到三个角色:

* ROS Master (管理者)
* Talker (参数设置者)
* Listener (参数调用者)

ROS Master 作为一个公共容器保存参数，Talker 可以向容器中设置参数，Listener 可以获取参数。



整个流程由以下步骤实现:

## 1.Talker 设置参数

Talker 通过 XMLRPC 向参数服务器发送参数(包括参数名与参数值——参数对)，ROS Master 将参数保存到参数列表中。

## 2.Listener 获取参数

Listener 通过 RPC 向参数服务器发送参数查找请求，请求中包含要查找的参数名。

## 3.ROS Master 向 Listener 发送参数值

ROS Master 根据步骤2请求提供的参数名查找参数值，并将查询结果通过 RPC 发送给 Listener。

参数可使用数据类型:

* 32-bit integers(int32\_t)
* booleans
* strings
* doubles
* iso8601 dates
* lists
* base64-encoded binary data
* 字典

注意:参数服务器不是为高性能而设计的，因此最好用于存储静态的非二进制的简单数据

iso8601 dates是什么东西？

日期和时间的表示方法，年为4位数，月为2位数，月中的日为2位数，例如2004年5月3日可写成2004-05-03或20040503。

小时、分和秒都用2位数表示，对UTC时间最后加一个大写字母Z，其他时区用实际时间加时差表示。如UTC时间下午2点30分5秒表示为14:30:05Z或143005Z，当时的北京时间表示为22:30:05+08:00或223005+0800，也可以简化成223005+08。

在ROS Master中，有个参数列表（容器），用于保存参数。

# 3.2.2 参数操作A(C++)

**需求:**实现参数服务器参数的增加、删除、查找和修改操作。

在 C++ 中实现参数服务器数据的增删改查，可以通过两套 API 实现:

* ros::NodeHandle类提供的一套成员函数，
* ros::param

下面为具体操作演示

## 1.参数服务器新增(修改)参数

/\*

参数服务器操作之新增与修改(二者API一样)\_C++实现:

在 roscpp 中提供了两套 API 实现参数操作

ros::NodeHandle

setParam("键",值)

ros::param

set("键","值")

示例:分别设置整形、浮点、字符串、bool、列表、字典等类型参数

修改(相同的键，不同的值)

\*/

#include "ros/ros.h"

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

{

ros::init(argc,argv,"set\_update\_param");

**std::vector<std::string> stus;**

stus.push\_back("data 1");

stus.push\_back("data 2");

stus.push\_back("data 3");

stus.push\_back("data 4");

**std::map<std::string,std::string> friends;**

friends["guo"] = "huang";

friends["yuang"] = "xiao";

//NodeHandle--------------------------------------------------------

ros::NodeHandle nh;

nh.setParam("nh\_int",10); //整型

nh.setParam("nh\_double",3.14); //浮点型

nh.setParam("nh\_bool",true); //bool

nh.setParam("nh\_string","hello NodeHandle"); //字符串

//向参数服务器内存储标准**容器**对象

nh.setParam("nh\_vector",stus); // vector

nh.setParam("nh\_map",friends); // map

//修改(相同的键，不同的值)

nh.setParam("nh\_int",10000);

//param-----------------------------------------------------------

ros::param::set("param\_int",20);

ros::param::set("param\_double",3.14);

ros::param::set("param\_string","Hello Param");

ros::param::set("param\_bool",false);

ros::param::set("param\_vector",stus);

ros::param::set("param\_map",friends);

//修改演示(相同的键，不同的值)

ros::param::set("param\_int",20000);

return 0;

}

## 2.参数服务器获取参数

/\*

参数服务器操作——参数查询\_C++实现:

在 roscpp 中提供了两套 API 实现参数操作

ros::NodeHandle

param(键,默认值)

存在，返回对应结果，否则返回默认值

getParam(键,存储结果的变量)

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamCached(键,存储结果的变量)--提高变量获取效率

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamNames(std::vector<std::string>)

获取所有的键,并存储在参数 vector 中

hasParam(键)

是否包含某个键，存在返回 true，否则返回 false

searchParam(参数1，参数2)

搜索键，参数1是被搜索的键，参数2存储搜索结果的变量

ros::param ----- 与 NodeHandle 类似

\*/

#include "ros/ros.h"

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

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"get\_param");

//NodeHandle--------------------------------------------------------

ros::NodeHandle nh;

// param 函数

int res1 = nh.param("nh\_int",100); // 键存在，返回键值

int res2 = nh.param("nh\_int2",100); // 键不存在，返回默认值100

ROS\_INFO("param获取结果:%d,%d",res1,res2);

// getParam 函数

int nh\_int\_value;

double nh\_double\_value;

bool nh\_bool\_value;

std::string nh\_string\_value;

std::vector<std::string> stus;

std::map<std::string, std::string> friends;

nh.getParam("nh\_int",nh\_int\_value);

nh.getParam("nh\_double",nh\_double\_value);

nh.getParam("nh\_bool",nh\_bool\_value);

nh.getParam("nh\_string",nh\_string\_value);

nh.getParam("nh\_vector",stus);

nh.getParam("nh\_map",friends);

ROS\_INFO("getParam获取的结果:%d,%.2f,%s,%d",

nh\_int\_value,

nh\_double\_value,

nh\_string\_value.c\_str(),

nh\_bool\_value

);

for (auto &&stu : stus)

{

ROS\_INFO("stus 元素:%s",stu.c\_str());

}

for (auto &&f : friends)

{

ROS\_INFO("map 元素:%s = %s",f.first.c\_str(), f.second.c\_str());

}

// getParamCached()

nh.getParamCached("nh\_int",nh\_int\_value);

ROS\_INFO("通过缓存获取数据:%d",nh\_int\_value);

//getParamNames()

std::vector<std::string> param\_names1;

nh.getParamNames(param\_names1);

for (auto &&name : param\_names1)

{

ROS\_INFO("名称解析name = %s",name.c\_str());

}

ROS\_INFO("----------------------------");

ROS\_INFO("存在 nh\_int 吗? %d",nh.hasParam("nh\_int"));

ROS\_INFO("存在 nh\_intttt 吗? %d",nh.hasParam("nh\_intttt"));

std::string key;

nh.searchParam("nh\_int",key);

ROS\_INFO("搜索键:%s",key.c\_str());

//param--------------------------------------------------------

ROS\_INFO("++++++++++++++++++++++++++++++++++++++++");

int res3 = ros::param::param("param\_int",20); //存在

int res4 = ros::param::param("param\_int2",20); // 不存在返回默认

ROS\_INFO("param获取结果:%d,%d",res3,res4);

// getParam 函数

int param\_int\_value;

double param\_double\_value;

bool param\_bool\_value;

std::string param\_string\_value;

std::vector<std::string> param\_stus;

std::map<std::string, std::string> param\_friends;

ros::param::get("param\_int",param\_int\_value);

ros::param::get("param\_double",param\_double\_value);

ros::param::get("param\_bool",param\_bool\_value);

ros::param::get("param\_string",param\_string\_value);

ros::param::get("param\_vector",param\_stus);

ros::param::get("param\_map",param\_friends);

ROS\_INFO("getParam获取的结果:%d,%.2f,%s,%d",

param\_int\_value,

param\_double\_value,

param\_string\_value.c\_str(),

param\_bool\_value

);

for (auto &&stu : param\_stus)

{

ROS\_INFO("stus 元素:%s",stu.c\_str());

}

for (auto &&f : param\_friends)

{

ROS\_INFO("map 元素:%s = %s",f.first.c\_str(), f.second.c\_str());

}

// getParamCached()

ros::param::getCached("param\_int",param\_int\_value);

ROS\_INFO("通过缓存获取数据:%d",param\_int\_value);

//getParamNames()

std::vector<std::string> param\_names2;

ros::param::getParamNames(param\_names2);

for (auto &&name : param\_names2)

{

ROS\_INFO("名称解析name = %s",name.c\_str());

}

ROS\_INFO("----------------------------");

ROS\_INFO("存在 param\_int 吗? %d",ros::param::has("param\_int"));

ROS\_INFO("存在 param\_intttt 吗? %d",ros::param::has("param\_intttt"));

std::string key;

ros::param::search("param\_int",key);

ROS\_INFO("搜索键:%s",key.c\_str());

return 0;

}

## 3.参数服务器删除参数

/\*

参数服务器操作之删除\_C++实现:

ros::NodeHandle

deleteParam("键")

根据键删除参数，删除成功，返回 true，否则(参数不存在)，返回 false

ros::param

del("键")

根据键删除参数，删除成功，返回 true，否则(参数不存在)，返回 false

\*/

#include "ros/ros.h"

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

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"delete\_param");

ros::NodeHandle nh;

bool r1 = nh.deleteParam("nh\_int");

ROS\_INFO("nh 删除结果:%d",r1);

bool r2 = ros::param::del("param\_int");

ROS\_INFO("param 删除结果:%d",r2);

return 0;

}

# 3.2.3 实际操作\_参数设置

**需求描述:** 修改turtlesim乌龟显示节点窗体的背景色，已知背景色是通过参数服务器的方式以 rgb 方式设置的。

**实现分析:**

1. 首先，需要启动乌龟显示节点。
2. 要通过ROS命令，来获取参数服务器中设置背景色的参数。
3. 编写参数设置节点，修改参数服务器中的参数值。

**实现流程:**

1. 通过ros命令获取参数。
2. 编码实现服参数设置节点。
3. 启动 roscore、turtlesim\_node 与参数设置节点，查看运行结果。

## 1.参数名获取

**获取参数列表:**

rosparam list

**响应结果:**

/turtlesim/background\_b

/turtlesim/background\_g

/turtlesim/background\_r

## 2.参数修改

**实现方案A:**C++

/\*

注意命名空间的使用。

\*/

#include "ros/ros.h"

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

{

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

ros::NodeHandle nh("turtlesim");

//ros::NodeHandle nh;

// ros::param::set("/turtlesim/background\_r",0);

// ros::param::set("/turtlesim/background\_g",0);

// ros::param::set("/turtlesim/background\_b",0);

nh.setParam("background\_r",0);

nh.setParam("background\_g",0);

nh.setParam("background\_b",0);

return 0;

}

## 3.运行

首先，启动 roscore；

然后启动背景色设置节点；

最后启动乌龟显示节点；

最终执行结果与演示结果类似。

PS: 注意节点启动顺序，如果先启动乌龟显示节点，后启动背景色设置节点，那么颜色设置不会生效。

## 4.其他设置方式

**方式1:修改小乌龟节点的背景色(命令行实现)**

rosparam set /turtlesim/background\_b 自定义数值

rosparam set /turtlesim/background\_g 自定义数值

rosparam set /turtlesim/background\_r 自定义数值

修改相关参数后，重启 turtlesim\_node 节点，背景色就会发生改变了

**方式2:启动节点时，直接设置参数**

rosrun turtlesim turtlesim\_node \_background\_r:=100 \_background\_g=0 \_background\_b=0

**方式3:通过launch文件传参**

<launch>

<node pkg="turtlesim" type="turtlesim\_node" name="set\_bg" output="screen">

<!-- launch 传参策略1 -->

<!-- <param name="background\_b" value="0" type="int" />

<param name="background\_g" value="0" type="int" />

<param name="background\_r" value="0" type="int" /> -->

<!-- launch 传参策略2 -->

<rosparam command="load" file="$(find demo03\_test\_parameter)/cfg/color.yaml" />

</node>

</launch>

# 3.2.4 与参数服务器相关的ROS命令——rosparam

rosparam包含rosparam命令行工具，用于使用YAML编码文件在参数服务器上获取和设置ROS参数。

rosparam set 设置参数

rosparam get 获取参数

rosparam load 从外部文件加载参数

rosparam dump 将参数写出到外部文件

rosparam delete 删除参数

rosparam list 列出所有参数

* rosparam list

列出所有参数

rosparam list

//默认结果

/rosdistro

/roslaunch/uris/host\_helloros\_virtual\_machine\_\_42911

/rosversion

/run\_id

* rosparam set

设置参数

rosparam set name huluwa

//再次调用 rosparam list 结果

/name

/rosdistro

/roslaunch/uris/host\_helloros\_virtual\_machine\_\_42911

/rosversion

/run\_id

* rosparam get

获取参数

rosparam get name

//结果

* rosparam delete

删除参数

rosparam delete name

//结果

//去除了name

* rosparam load(先准备 yaml 文件)

从外部文件加载参数

rosparam load xxx.yaml

* rosparam dump

将参数写出到外部文件

rosparam dump yyy.yaml

# 3.3动态配置参数

参数服务器的数据被修改时，如果节点不主动地去重新访问，那么就不能获取修改后的数据。而一些特殊场景下，是要求要能做到动态获取的，也就是说参数一旦修改，能够通知节点“参数已经修改”并读取修改后的数据。

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，另一个节点实时解析修改后的数据。

# 3.3.1 动态配置参数的客户端

**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)。

chmod +x <\*>.cfg

我们首先看如何实现参数配置文件。

1、在工作空间的src目录下，创建一个功能包——demo02\_dr

2、导入依赖项：roscpp rospy std\_msgs dynamic\_reconfigure

3、在新建功能包下，创建目录——cfg

4、在cfg目录下，新建一个配置文件——<filename>.cfg

在这个py文件中，首先声明“释伴”行（解释伴随行）格式如下所示：

#! <解释器的绝对路径>

#！/usr/bin/python3

#！/usr/bin/env python3

#! /usr/bin/env python

# 这是python语言中的单行注释，以#开头

"""

这是python的多行注释，使用三重双引号。

"""

*# 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)

*#gen.enum([,,...], "")*

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")

*#名为list\_param的参数，类型为“整型”，参数描述——列表参数，默认值为0，最小值为*

*#0，最大值为2，编辑方法为many\_enum*

gen**.**add("size\_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("demo02\_dr","dr\_client","dr"))

generate有三个参数：分别是(packname, nodename, name)

第三个参数name: 是生成文件所使用的前缀，name的值必须和配置文件的第一部分相匹配，例如：若配置文件名为 dr.cfg，那么这个name参数的值为“dr”

编译生成的中间文件

devel/include/demo02\_dr/drConfig.h

# 3.3.2 动态参数服务端A(C++)

**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

}

### 1.服务器代码实现

新建cpp文件dr\_server.cpp，内容如下:

**#include** "ros/ros.h"

**#include** "dynamic\_reconfigure/server.h"

**#include** "dr\_server/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**(dr\_server**::**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**<**dr\_server**::**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**<**dr\_server**::**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;

}

### 2.编译配置文件

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

...

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

...

target\_link\_libraries(dr\_server

${catkin\_LIBRARIES}

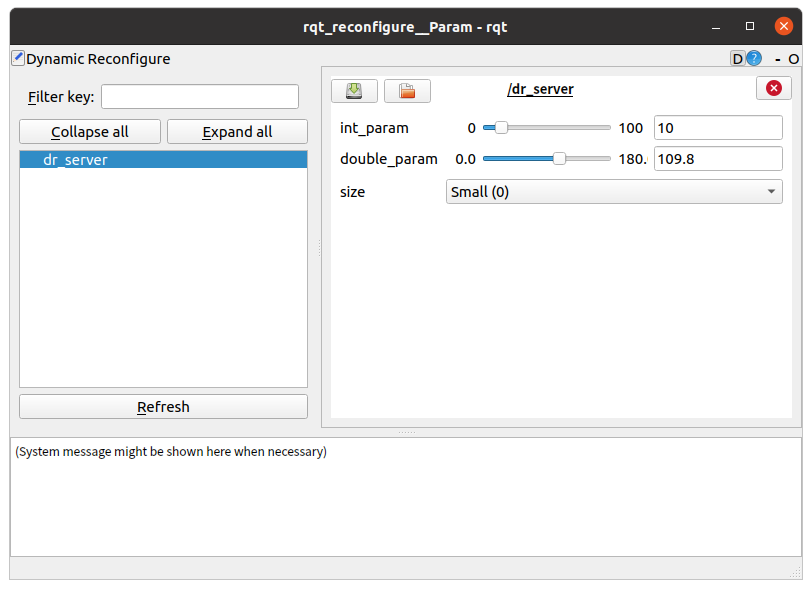
)

### 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**



如果后启动rqt\_reconfigure节点，则需要单击“Refresh”按钮后，选择响应的服务器节点。

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。

附录：boost::function和boost::bind使用

boost::function用于封装另一个函数。这个被封装的函数的最大用处是它可以被多次调用。

告诉function，它将保存的函数的原型和返回类型。

bool some\_func(int a, int b)

{

return a > b;

}

//类似于函数指针，f可以指向任何具有两个整型输入参数和一个bool类型返回值的函数

boost::function<bool(int, double)> f;

f = &some\_func;

f(10, 1.1);

定义一个函数指针：

bool (\*f)(int, int);

f = some\_func;

C语言（非C++语言）中只有普通函数的概念。C++语言中，除了普通函数外，还有类的成员函数和函数对象。

什么叫函数对象：在C++中，我们把所有能当做函数使用的对象统称为函数对象。

一般来说，如果我们列出一个对象，而它的后面又跟有由括号包裹的参数列表，就像f(arg1, arg2, ...)，这个对象就被称为函数对象。

## 函数类

一个函数类，即一个重载了括号操作符()的类。当用该类的对象调用此操作符时，其表现形式如同普通函数调用一般，因此取名叫函数类。举个最简单的例子：

**class** **FuncObjType**

{

**public:**

**void** **operator**() ()

{

cout**<<**"Hello C++!"**<<**endl;

}

};

类FuncObjType中重载了()操作符，因此对于一个该类的对象val，可以这样调用该操作符：val()。调用结果即输出以上代码中的内容。该调用语句在形式上跟以下函数的调用完全一样：

**void** **val**()

{

cout**<<**"Hello C++!"**<<**endl;

}

既然用函数对象与调用普通函数有相同的效果，为什么还有搞这么麻烦定义一个类来使用函数对象？主要在于函数对象有以下的优势：

* 函数对象可以有自己的状态。我们可以在类中定义状态变量，这样一个函数对象在多次的调用中可以共享这个状态。但是函数调用没这种优势，除非它使用全局变量来保存状态。
* 函数对象有自己特有的类型，而普通函数无类型可言。这种特性对于使用C++标准库来说是至关重要的。这样我们在使用STL中的函数时，可以传递相应的类型作为参数来实例化相应的模板，从而实现我们自己定义的规则。

## Lambda

C++11的一大亮点就是引入了Lambda表达式。利用Lambda表达式，可以方便的定义和创建匿名函数。Lambda表达式通过在最前面的方括号[]来明确指明其内部可以访问的外部变量，这一过程也称过Lambda表达式“捕获”了外部变量。类似参数传递方式（值传递、引入传递、指针传递），在Lambda表达式中，外部变量的捕获方式也有值捕获、引用捕获、隐式捕获。

**#include** <vector>

**#include** <vector>

**#include** <iostream>

**#include** <algorithm>

​

**int** main()

{

**int** count **=** 0;

std**::**vector**<**std**::**string**>** words{ "An", "ancient", "pond" };

std**::**for\_each(words.cbegin(), words.cend(),

[**&**count](**const** std**::**string**&** word)

{

**if**(isupper(word[0])) {

std**::**cout **<<** word **<<** " " **<<** count **<<** std**::**endl;

count**++**;

}

});

}

这样的代码在编译时，lambda表达式会自动转变成一个类——它每一个成员变量都对应着一个捕获的变量。这个类根据lambda表达式的参数列表重载了operator()，一个可能的实现如下：

*// 匿名类名由编译器自动生成*

**class** **lambda\_implementation** {

**public:**

lambda\_implementation(**int&** n) **:** count\_(n) {}

**void** **operator**()(**const** std**::**string**&** word) **const**

{

**if**(isupper(word[0]))

{

std**::**cout **<<** word **<<** " " **<<** count **<<** std**::**endl;

count\_**++**;

}

}

**private:**

**int&** count\_;

}

Lambdas 并没有允许你做任何之前的代码做不到的事情，但是它在帮助你保持代码的逻辑精简的同时不需要写一个普普通通的函数类。

## std::function类

当你需要一个非模板函数对象作为类的成员或者函数参数时，你必须指定函数对象的具体类型。C++中的函数对象并没有一个基类，但是标准库提供了一个模板类std::function来代表所有的函数对象。

std**::**function**<float**(**float**, **float**)**>** test\_function;

​test\_function **=** std**::**fmaxf; *// Ordinary function*

test\_function **=** std**::**multiplies**<float>**(); *// class with a call operator*

test\_function **=** std**::**multiplies**<>**(); *// class with a generic call operator*

test\_function **=** [x](**float** a, **float** b) { **return** a **\*** x **+** b; }; *// lambda*

test\_function **=** [x](**auto** a, **auto** b) { **return** a **\*** x **+** b; }; *// generic lambda*

test\_function **=** [](std**::**string s) { **return** s.empty(); }; *//ERROR!*

std::function并不像std::vector<T>等容器对包含的类型做了抽象，而是抽象了函数对象的参数和返回值。无论是普通函数，还是函数指针、lambdas，又或是任何可以被当做函数使用的对象，只要拥有相同参数和返回值，均可以用同一类std::function表示。

尽管std::function非常有用，但是它也带来了性能损失。这是因为为了隐藏包含的函数对象类型，提供通用的调用接口，std::function使用了叫做type erasure的技术。简单来说是通过虚函数的调用在运行期来决定具体调用，因此编译器无法内联（inline）函数调用，也无法进行更多优化。

## std::bind和闭包

大多数的编程范式都提供了提高代码重用的方式，比如在面向对象编程中，我们可以通过抽象出特定类来将复杂系统拆分成小的组件，在降低耦合的同时也可以分开设计、测试代码。

在函数式编程中，通过组合现有的函数，我们可以创造出新的函数。标准库中的std::bind就是可以创造闭包（closure）的工具。

**#include** <algorithm>

​

**class** **Foo**

{

**public:**

**void** methodA();

**void** **methodInt**(**int** a);

};

**class** **Bar**

{

**public:**

**void** methodB();

};

​

**void** main()

{

std**::**function**<void**()**>** f1; *// 无参数，无返回值*

​

Foo foo;

f1 **=** std**::**bind(**&**Foo**::**methodA, **&**foo);

f1(); *// 调用 foo.methodA();*

Bar bar;

f1 **=** std**::**bind(**&**Bar**::**methodB, **&**bar);

f1(); *// 调用 bar.methodB();*

​

f1 **=** std**::**bind(**&**Foo**::**methodInt, **&**foo, 42);

f1(); *// 调用 foo.methodInt(42);*

​

std**::**function**<void**(**int**)**>** f2; *// int 参数，无返回值*

f2 **=** std**::**bind(**&**Foo**::**methodInt, **&**foo, \_1);

f2(53); *// 调用 foo.methodInt(53);*

}

通过std::bind，我们可以为同一个类的不同对象可以分派不同的实现，从而实现不同的行为。这种方式使得我们不在需要设计通过继承与虚函数来实现多态，无疑为程序库设计提供的新的方式。

扩展1：

如何实现ROS自定义消息的嵌套？

具体实现过程：

1. 创建功能包，并在功能包的目录下创建msg子目录。
2. 在msg目录下，创建消息文件，消息文件名.msg
3. 定义的消息文件，可以使用其他功能包中定义的消息类型。
4. 修改package.xml文件，添加如下内容：

<build\_depend>message\_generation</build\_depend>

<exec\_depend>message\_runtime</exec\_depend>

1. 修改CMakeList.txt

generate\_message(

DEPENDENCIES

std\_msgs

<嵌套的消息所在的功能包名称>

)

catkin\_package(

CATKIN\_DEPENDS roscpp rospy std\_msgs message\_runtime

)

扩展2：

如何在ROS中定义变长度的消息？

1. 新建功能包custom\_msgs，并在功能包下新建目录msg；
2. 在msg目录下，新建消息文件，例如VecOfDouble.msg；
3. 修改package.xml文件，添加如下内容：

<build\_depend>message\_generation</build\_depend>

<exec\_depend>message\_runtime</exec\_depend>

1. 修改CMakeList.txt，修改如下内容：

#添加message\_generation

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

message\_generation

)

#添加消息文件VecOfDouble.msg

add\_message\_files(

FILES

VecOfDouble.msg

)

#打开如下语句的注释

generate\_messages(

DEPENDENCIES

std\_msgs

)

如何使用变长的消息？

#include <ros/ros.h>

#include <custom\_msgs/VecOfDouble.h>

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

{

setlocale(LC\_ALL, "");

ros::init(argc, argv, "custom\_msgs");

ros::NodeHandle nh;

ros::Publisher pub = nh.advertise<custom\_msgs::VecOfDouble>("custom\_msgs", 10);

ros::Rate loop\_rate(10);

double count = 0.0;

custom\_msgs::VecOfDouble msg;

msg.dbl\_vec.resize(3);

msg.dbl\_vec[0] = 3.1415926;

msg.dbl\_vec[1] = 2.71828;

msg.dbl\_vec[2] = 1.414;

while(ros::ok())

{

count += 1.0;

msg.dbl\_vec.push\_back(count);

pub.publish(msg);

loop\_rate.sleep();

ros::spinOnce();

}

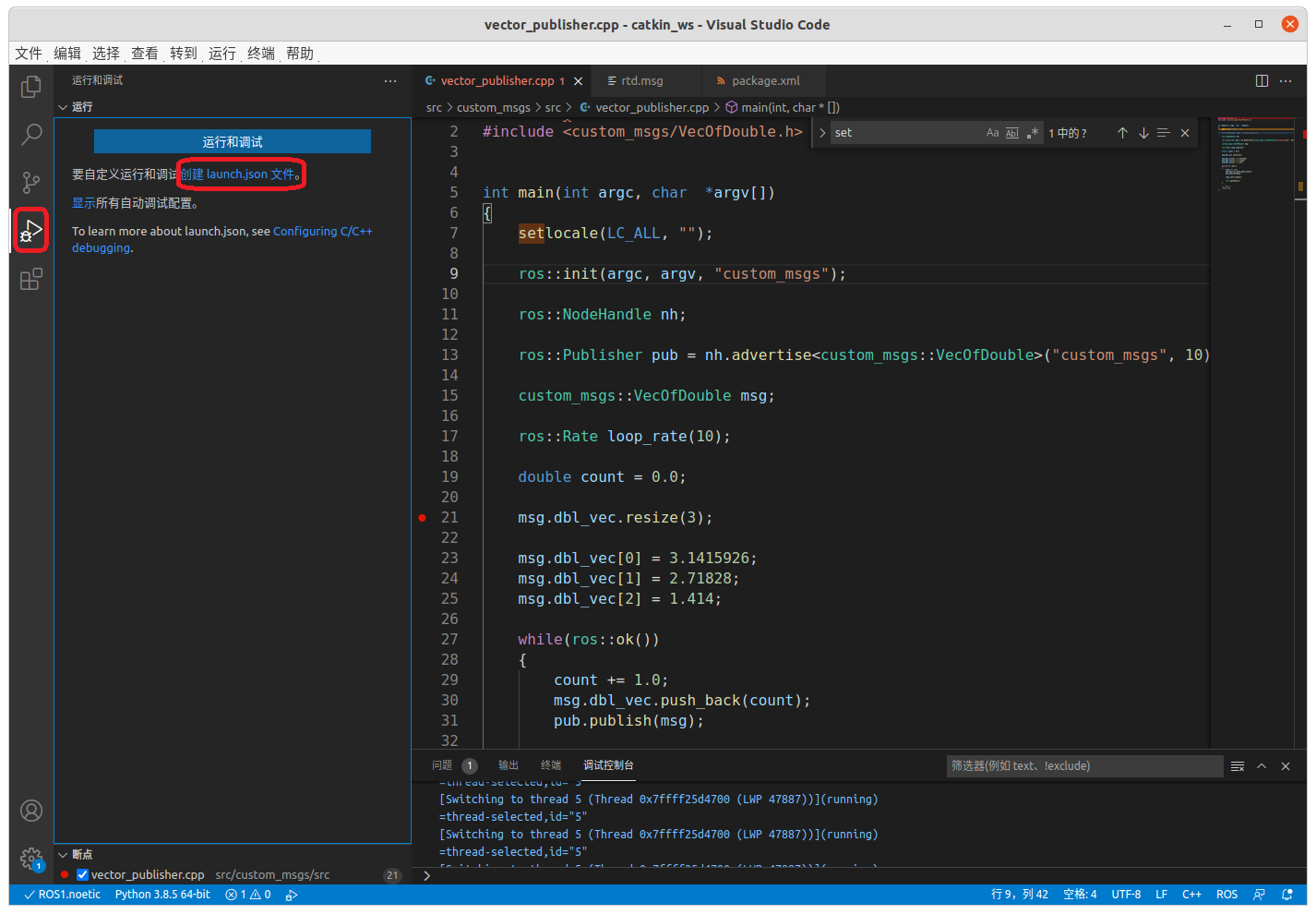
/\* code \*/

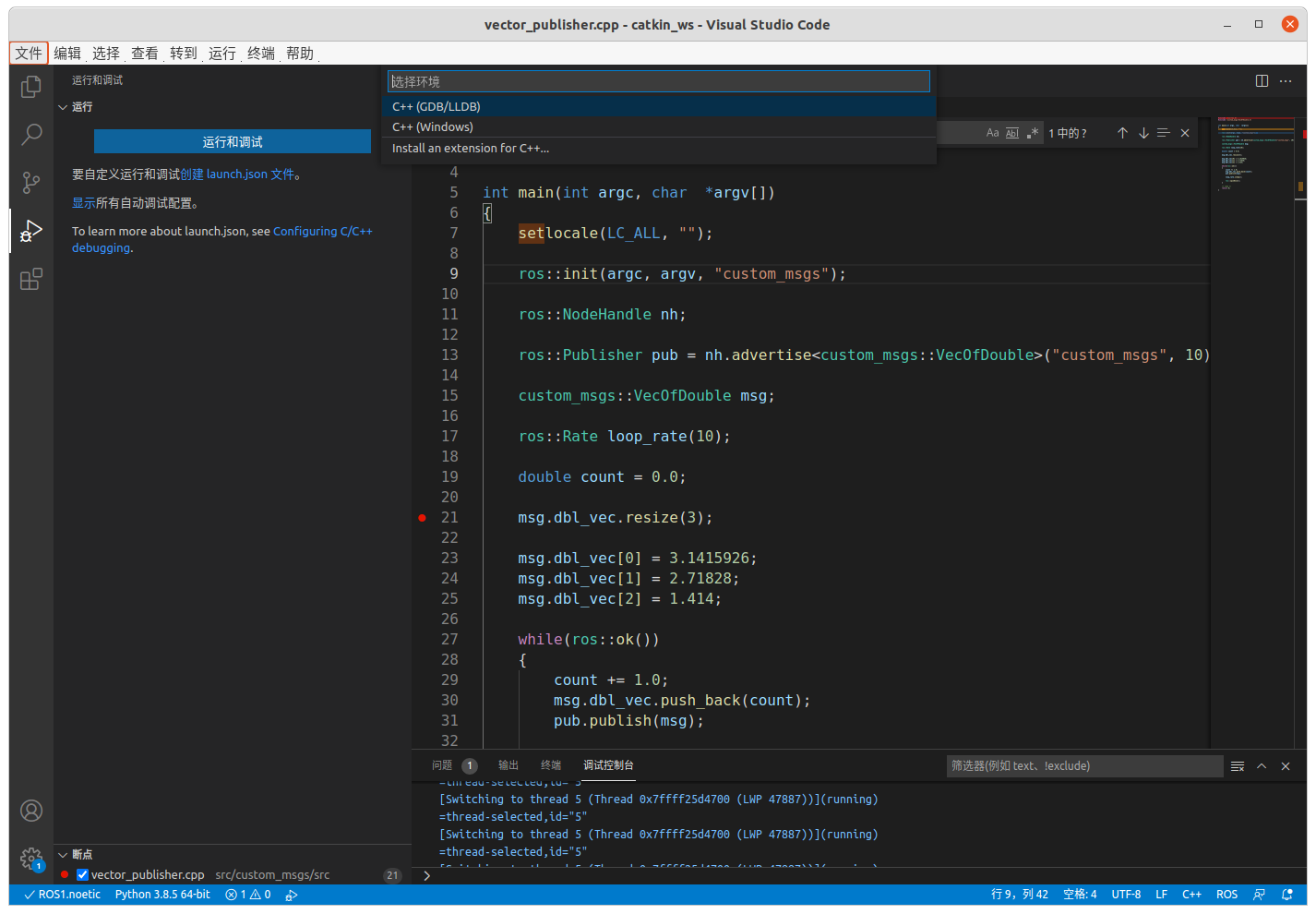
return 0;

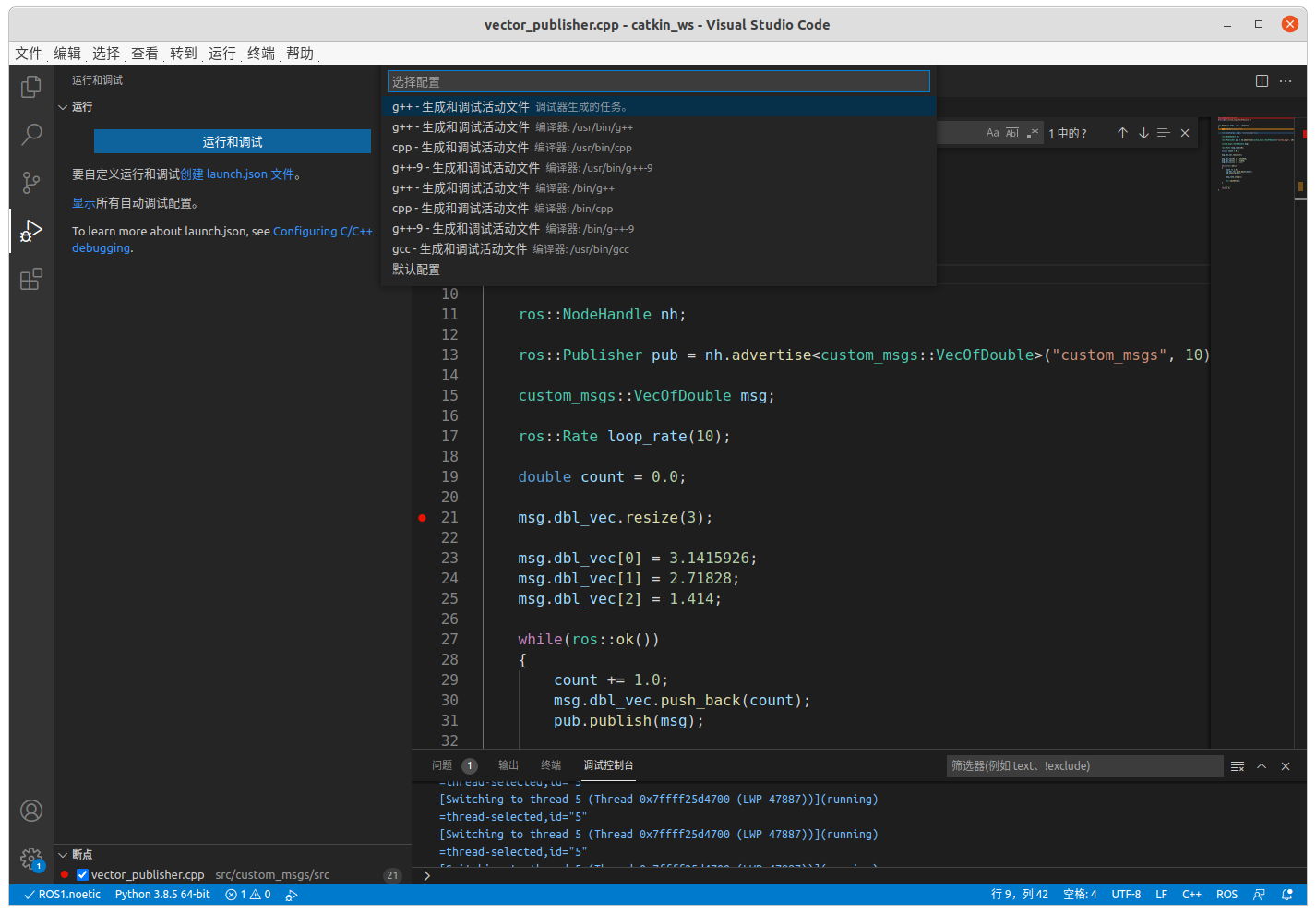
}

扩展3：

如何在VSCode中调试ROS的节点？







配置launch.json文件

{

// 使用 IntelliSense 了解相关属性。

// 悬停以查看现有属性的描述。

// 欲了解更多信息，请访问: https://go.microsoft.com/fwlink/?linkid=830387

"version": "0.2.0",

"configurations": [

{

"name": "g++ - 生成和调试活动文件",

"type": "cppdbg",

"request": "launch",

**"program": "${workspaceFolder}/devel/lib/custom\_msgs/vector\_publisher",**

"args": [],

"stopAtEntry": false,

"cwd": "${workspaceFolder}",

"environment": [],

"externalConsole": false,

"MIMode": "gdb",

"setupCommands": [

{

"description": "为 gdb 启用整齐打印",

"text": "-enable-pretty-printing",

"ignoreFailures": true

}

],

"preLaunchTask": "C/C++: g++ 生成活动文件",

"miDebuggerPath": "/usr/bin/gdb"

}

]

}

# ROS中的头文件与源文件

本节主要介绍ROS的C++实现中，如何使用头文件与源文件的方式封装代码，具体内容如下:

1. 设置头文件，可执行文件作为源文件；
2. 分别设置头文件，源文件与可执行文件。

在ROS中关于头文件的使用，核心内容在于CMakeLists.txt文件的配置，不同的封装方式，配置上也有差异。

# 1 自定义头文件调用

**需求:**设计头文件，可执行文件本身作为源文件。

**流程:**

1. 编写头文件；
2. 编写可执行文件(同时也是源文件)；
3. 编辑配置文件并执行。

## 1.头文件

在功能包下的 include/功能包名 目录下新建头文件: hello.h，示例内容如下:

#ifndef \_HELLO\_H

#define \_HELLO\_H

namespace hello\_ns{

class HelloPub {

public:

void run();

};

}

#endif

**注意:**

在 VScode 中，为了后续包含头文件时不抛出异常，请配置 .vscode 下 c\_cpp\_properties.json 的 includepath属性

"/home/用户/工作空间/src/功能包/include/\*\*"

## 2.可执行文件

在 src 目录下新建文件:hello.cpp，示例内容如下:

#include "ros/ros.h"

#include "test\_head/hello.h"

namespace hello\_ns {

void HelloPub::run(){

ROS\_INFO("自定义头文件的使用....");

}

}

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

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"test\_head\_node");

hello\_ns::HelloPub helloPub;

helloPub.run();

return 0;

}

## 3.配置文件

配置CMakeLists.txt文件，头文件相关配置如下:

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

可执行配置文件配置方式与之前一致:

add\_executable(hello src/hello.cpp)

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

target\_link\_libraries(hello

${catkin\_LIBRARIES}

)

最后，编译并执行，控制台可以输出自定义的文本信息。

# 2 自定义源文件调用

**需求:**设计头文件与源文件，在可执行文件中包含头文件。

**流程:**

1. 编写头文件；
2. 编写源文件；
3. 编写可执行文件；
4. 编辑配置文件并执行。

## 1.头文件

头文件设置于 3.2.1 类似，在功能包下的 include/功能包名 目录下新建头文件: haha.h，示例内容如下:

#ifndef \_HAHA\_H

#define \_HAHA\_H

namespace hello\_ns {

class My {

public:

void run();

};

}

#endif

**注意:**

在 VScode 中，为了后续包含头文件时不抛出异常，请配置 .vscode 下 c\_cpp\_properties.json 的 includepath属性

"/home/用户/工作空间/src/功能包/include/\*\*"

## 2.源文件

在 src 目录下新建文件:haha.cpp，示例内容如下:

#include "test\_head\_src/haha.h"

#include "ros/ros.h"

namespace hello\_ns{

void My::run(){

ROS\_INFO("hello,head and src ...");

}

}

## 3.可执行文件

在 src 目录下新建文件: use\_head.cpp，示例内容如下:

#include "ros/ros.h"

#include "test\_head\_src/haha.h"

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

{

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

hello\_ns::My my;

my.run();

return 0;

}

## 4.配置文件

头文件与源文件相关配置:

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

## 声明C++库

add\_library(head

include/test\_head\_src/haha.h

src/haha.cpp

)

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

target\_link\_libraries(head

${catkin\_LIBRARIES}

)

可执行文件配置:

add\_executable(use\_head src/use\_head.cpp)

add\_dependencies(use\_head ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

#此处需要添加之前设置的 head 库

target\_link\_libraries(use\_head

head

${catkin\_LIBRARIES}

)