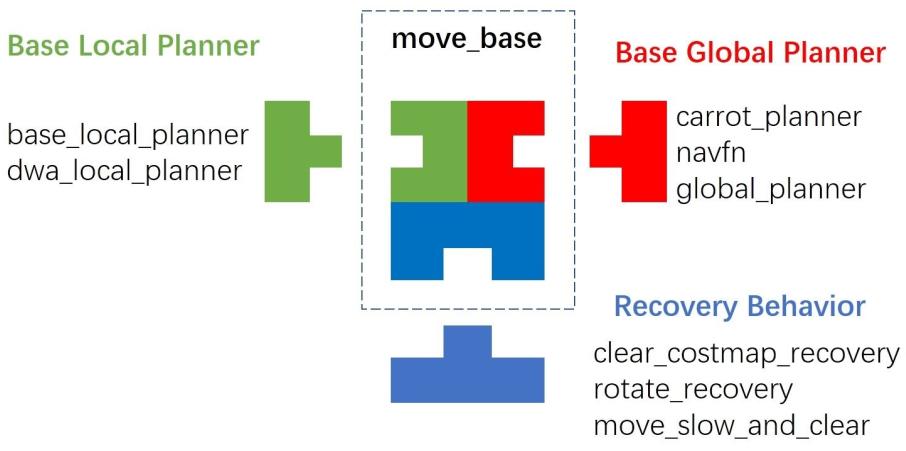
# 一、pluginlib

在ROS开发中，经常会接触到一个名词——plugin(插件)。简单来说，ROS中的插件就是一个“像动态链接库一样可动态加载的扩展功能“类”。ROS中的pluginlib功能包提供了加载和卸载“插件类”的C++库。开发者在插件开发完成后，需要将插件注册到ROS系统中；使用者在使用插件类时，就不需要考虑插件类位于文件系统中的具体位置，即可直接动态加载。

在软件编程中，**插件是一种遵循一定规范的使用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. 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。
2. 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。  
   例如：



在开发ROS应用时，假如我们已经有了一个类——baseClass，它在功能包base\_package中实现。我们可以通过插件机制来扩展这个基类的功能，一个简单直观的实现方式是通过面向对象编程中的继承或者派生的概念，从基类baseClass中派生出一个子类（derived\_package）。

class derivedClass : public baseClass

{

//...

}

在派生类中，我们可以定义新的成员变量和成员函数，也可以重载基类的虚拟函数。

前面在我的讲义中，已经介绍了如何在ROS中使用C++类和编写库函数，这些库函数我们是将他们作为静态链接库直接链接到应用程序中的，而非动态加载。

如果要做到“类”的动态加载，我们还需要完成将功能包中的类“导出”，并且让ROS知道有这个“类”存在（那就需要注册）。

### 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；

# 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；//接口
3. 创建插件类；
4. 注册插件; //将功能包中的类导出
5. 构建插件库;
6. 使插件可用于ROS工具链;
7. 配置xml //注册
8. 导出插件
9. 使用插件;
10. 执行。

### 1.准备

创建功能包demo\_plugin，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。//用到了自定义的头文件

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

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

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

"/${workspace}/<功能包名>/src/include/\*\*"

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

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

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4}

### 2.创建基类

在 demo\_plugin/include/dmo\_plugin下新建C++头文件: polygon\_base.h，你要实现的插件类都需要继承此基类，内容如下:

#ifndef POLYGON\_BASE\_H\_

#define 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

**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 demo\_plugin/include/demo\_plugin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**

该文件中创建了正方形与三角形两个派生类继承基类。

### 4.注册插件（导出类）

在 src 目录下新建 文件，内容如下:

*//pluginlib 宏，可以注册插件类*

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

**#include** <demo\_plugin/polygon\_base.h> //demo\_plugin需要替换成功能包的名字

**#include** <demo\_plugin/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)

该文件会将两个派生类注册为插件。

### 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)

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——libpolygon\_plugins.so

### 6.使插件可用于ROS工具链(供开发者使用)

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>

6.2导出插件

package.xml文件中设置内容如下:

<export>

<**demo\_plugin** plugin="${prefix}/polygon\_plugins.xml" />

</export>

标签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 demo\_plugin //<插件功能包的名称>

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("demo\_plugin", "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;

}

### 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})

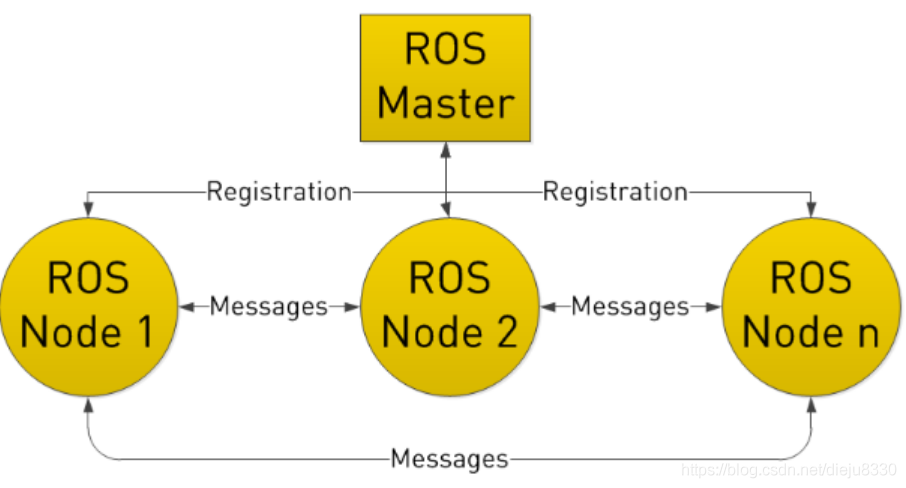
编译然后执行:polygon\_loader，结果如下:

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

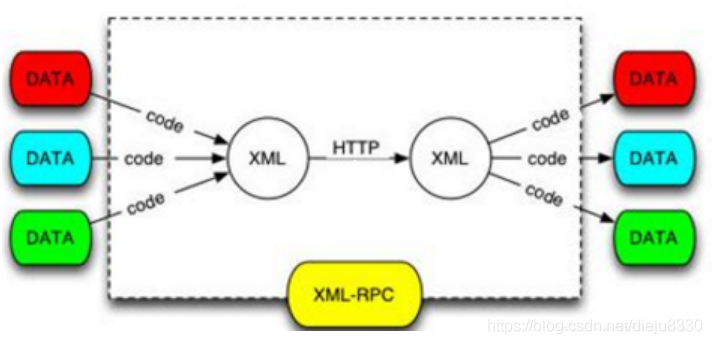
[ INFO] [WallTime: 1279658450.869138007]: Square area: 100.00

# 二、nodelet

ROS是一种基于分布式网络通讯的操作系统，整个机器人控制系统是由一个Master主节点和若干个功能相对独立的Node子节点组成，这也是ROS系统最主要的特点就是分布式以及模块化的设计。在ROS通讯过程中Master节点存储着各个子节点的topics和services的注册信息，每个功能节点在请求服务之前先向主节点进行注册，然后节点之间就可以直接进行信息传递。ROS的底层通信都是基于XML-RPC协议实现的。



XML-RPC协议是XML Remote Prodecure Call的简称，是一种远程过程调用的分布式网络协议。它允许跨平台的软件间通过发送和接收XML格式的消息进行远程调用，即允许不同的操作系统、不同环境中的程序实现基于Internet过程调用的规范和一系列方法的实现。这种远程过程调用使用http作为传输协议，XML作为传送信息的编码格式。



XML-RPC的远程调用过程为：首先客户端发起请求后需要按照协议格式对请求信息进行填充；填充完毕以后XML格式的信息会被转化为数据流，通过传输层进行传输。服务端收到客户端发出来的数据流，会将其再转化为XML格式的信息，然后按照XML-RPC协议获取客户端的请求信息，并对请求信息进行处理，处理完毕以后将反馈信息发送给客户端。

　　以XML-RPC的方式传输数据存在一定的延时和阻塞。在数据量小、频率低的情况下，传输耗费的时间可以忽略不计。但当传输图像流，点云等数据量较大的消息，或者执行有一定的实时性要求的任务时，因传输而耗费的时间就不得不考虑。Nodelet包就是为改善这一状况设计的，它提供一种方法，可以让多个算法程序在一个进程中用 shared\_ptr 实现零拷贝通信（zero copy transport），以降低因为传输大数据而损耗的时间。简单讲就是可以将多个node捆绑在一起管理，使得同一个manager里面的topic的数据传输更快。

# 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启动之后独占一个进程，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。

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-noetic-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-noetic-nodelet-tutorial-math

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

### 3.内置案例调用

### 1.启动roscore

roscore

### 2.启动manager

rosrun nodelet nodelet manager \_\_name:=mymanager

\_\_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

添加第二个节点:

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

PS: 解释

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus nodelet\_manager \_\_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 nodelet\_manager \_\_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="nodelet\_manager" 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 nodelet\_manager" 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 nodelet\_manager" output="screen" >

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

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

</node>

</launch>

### 4.执行

向节点n1发布消息:

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

打印节点n2发布的消息:

rostopic echo /n2/out

最终输出结果应该是:60。

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:**

* 准备；
* 创建插件类并注册插件;
* 构建插件库;
* 使插件可用于ROS工具链；
* 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)

### 3.构建插件库

CMakeLists.txt配置如下：

...

add\_library(myplus

src/myplus.cpp

)

...

target\_link\_libraries(myplus

${catkin\_LIBRARIES}

)

编译后，会在 工作空间/devel/lib/生成文件: libmyplus.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:myplus.xml)，内容如下：

<library path="lib/libmyplus">

<class name="demo\_nodelet/MyPlus" type="nodelet\_demo\_ns::MyPlus"

base\_class\_type="nodelet::Nodelet" >

<description>myplus</description>

</class>

</library>

### 4.2导出插件

<export>

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

<nodelet plugin="${prefix}/myplus.xml" />

</export>

### 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>

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。