目录

[第1章 C#调用OpenVINOTM部署Al模型项目开发 1](#_Toc100761670)

[1.1 项目概述 1](#_Toc100761671)

[1.1.1 项目介绍 1](#_Toc100761672)

[1.1.2 OpenVINOTM 1](#_Toc100761673)

[1.1.3 项目方案 1](#_Toc100761674)

[1.1.4 安装方式 2](#_Toc100761675)

[1.2 软件安装 2](#_Toc100761676)

[1.2.1 Microsoft Visual Studio 2022安装 2](#_Toc100761677)

[1.2.2 OpenVINOTM安装 3](#_Toc100761678)

[1.2.3 OpenCV安装 3](#_Toc100761679)

[1.3 OpenVINOTM推理模型与测试数据集 4](#_Toc100761680)

[1.3.1 模型种类与下载方式 4](#_Toc100761681)

[1.3.2 PaddleDetection模型 4](#_Toc100761682)

[1.3.3 PaddleClas 模型 5](#_Toc100761683)

[1.4 创建OpenVINOTM 方法C++动态链接库 5](#_Toc100761684)

[1.4.1 新建解决方案以及项目文件 5](#_Toc100761685)

[1.4.2 配置C++项目属性 6](#_Toc100761686)

[1.4.3 编写C++代码 9](#_Toc100761687)

[1.4.4 编写模块定义文件 13](#_Toc100761688)

[1.4.5 生成dll文件 13](#_Toc100761689)

[1.5 C#构建Core类 14](#_Toc100761690)

[1.5.1 新建C#类库 14](#_Toc100761691)

[1.5.2 引入dll文件中的方法 14](#_Toc100761692)

[1.5.3 创建Core类 14](#_Toc100761693)

[1.5.4 编译Core类库 15](#_Toc100761694)

[1.6 C#实现OpenVINOTM方法的调用 15](#_Toc100761695)

[1.6.1 新建C#项目 15](#_Toc100761696)

[1.6.2 添加OpenCVsharp 15](#_Toc100761697)

[1.6.3 添加项目引用 16](#_Toc100761698)

[1.6.4 编写代码测试花卉分类模型 16](#_Toc100761699)

[1.6.5 编写代码测试车辆识别模型 18](#_Toc100761700)

[1.7 程序时间分析 19](#_Toc100761701)

[1.8 项目总结 20](#_Toc100761702)

# C#调用OpenVINOTM部署Al模型项目开发

## 项目概述

### 项目介绍

C#调用OpenVINOTM部署Al模型项目开发项目，简称OpenVinoSharp，这是一个示例项目，该项目实现在C#编程语言下调用Intel推出的 OpenVINO™ 工具套件，进行深度学习等Al项目在C#框架下的部署。该项目由C++语言编写OpenVINO™ API接口，并在C#语言下实现应用。

项目可以实现在C#编程语言下调用Intel推出的 OpenVINO™ 工具套件，进行深度学习等Al项目在C#框架下的部署，目前可以支持的Al模型格式：

* Paddlepaddle 飞桨模型 (.pdmodel)
* ONNX 开放式神经网络交换模型 (.onnx)
* IR 模型 (.xml, .bin)

目前该项目针对 Paddlepaddle飞桨现有模型进行了测试，主要有：

* PaddleClas 飞桨图像识别套件
* PaddleDetection 目标检测模型套件

### OpenVINOTM

OpenVINOTM是英特尔基于自身现有的硬件平台开发的一种可以加快高性能计算机视觉和深度学习视觉应用开发速度工具套件，支持各种英特尔平台的硬件加速器上进行深度学习，并且允许直接异构执行。支持在Windows与Linux系统，官方支持编程语言为Python与C++语言。

OpenVINOTM 工具套件2022.1版于2022年3月22日正式发布，根据官宣《OpenVINOTM迎来迄今为止最重大更新，2022.1新特性抢先看》，OpenVINOTM  2022.1将是迄今为止最大变化的版本。从开发者的角度来看，对于提升开发效率或运行效率有用的特性有：

* 提供预处理API函数
* ONNX前端API
* AUTO 设备插件
* 支持直接读入飞桨模型

该项目开发环境为OpenVINOTM 2022.1最新版本，因此使用者需在使用时将自己电脑上的OpenVINOTM 版本升级到2022.1版，不然会有较多的问题。

### 项目方案

该项目主要通过调用dll文件方式实现。通过C++调用OpenVINOTM，编写模型推理接口，将我们所用到的推理方法在C++中实现，并将其生成dll文件，在C#调用dll文件，重写dll文件接口，并重新组建Core类，用于在C#中进行模型的推理，其方案如图1- 1所示。

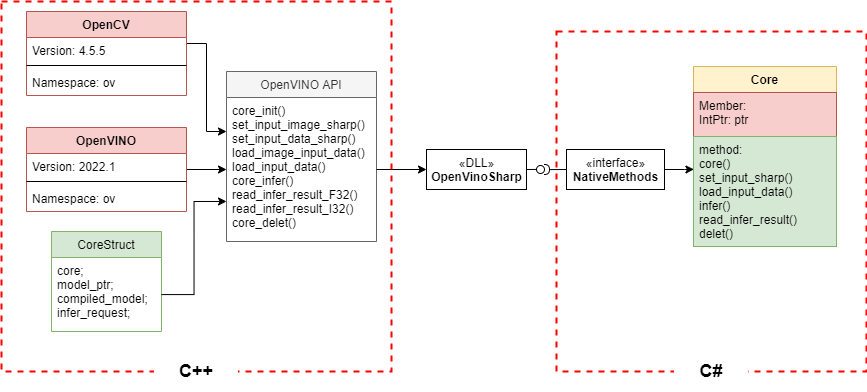


图1- 1 项目解决方案

### 安装方式

该项目所有文件已经上传到Github和Gitee远程代码仓，大家可以通过Gi在本地进行克隆。

* 系统平台：

​ Windows

* 软件要求：

​ Visual Studio 2022 / 2019 / 2017

​ OpenCV 4.5.5

​ OpenVINO 2022.1

* 安装方式

在Github上克隆下载：

git clone https://github.com/guojin-yan/OpenVinoSharp.git

在Gitee上克隆下载：

git clone https://gitee.com/guojin-yan/OpenVinoSharp.git

## 软件安装

### Microsoft Visual Studio 2022安装

Microsoft Visual Studio（简称VS）是美国微软公司的开发工具包系列产品。VS是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如UML工具、代码管控工具、集成开发环境(IDE)等等。其支持C、C++、C#、F#、J#等多门编程语言。

本次项目所使用的编程语言为C++与C#两门编程语言，在VS中完全可以实现，可选择安装版本VS2017、VS2019或VS2022版本。对于VS不同版本的选择，该项目不做较多要求，就笔者使用来说，VS2017版本推出时间较久，不建议使用，其一些编程语言规范有一些变动，对于该项目所提供的范例可能会有部分不兼容；VS2019和VS2022版本相对更新，是由起来差异不大，建议选择这两个版本，并且新版OpenVINOTM 支持VS2022版本Cmake。

笔者电脑安装的为Microsoft Visual Studio Community 2022 版本，其安装包可由VS官网直接下载，下载时选择社区版，按照一般安装步骤进行安装即可。在安装中，工作负荷的选择图1- 2所示。



图1- 2 Visual Studio 2022安装负荷

安装完成后，可以参照网上相关教程，进行学习VS的使用。

### OpenVINOTM安装

该项目所使用的OpenVINOTM版本为2022.1版本，是Intel公司在2022年第一季度发布的最新版本。该版本基于之前版本有了较大变动，不在默认包含OpenCV工具；其次，对代码做了更进一步的优化，使得代码在使用时更加灵活。其具体安装方式如下：

（1）OpenVINOTM下载

访问OpenVINOTM官网<www.openvino.ai>，点击Free Download->，进入到软件下载页面，按照图1- 3所示，选择安装内容

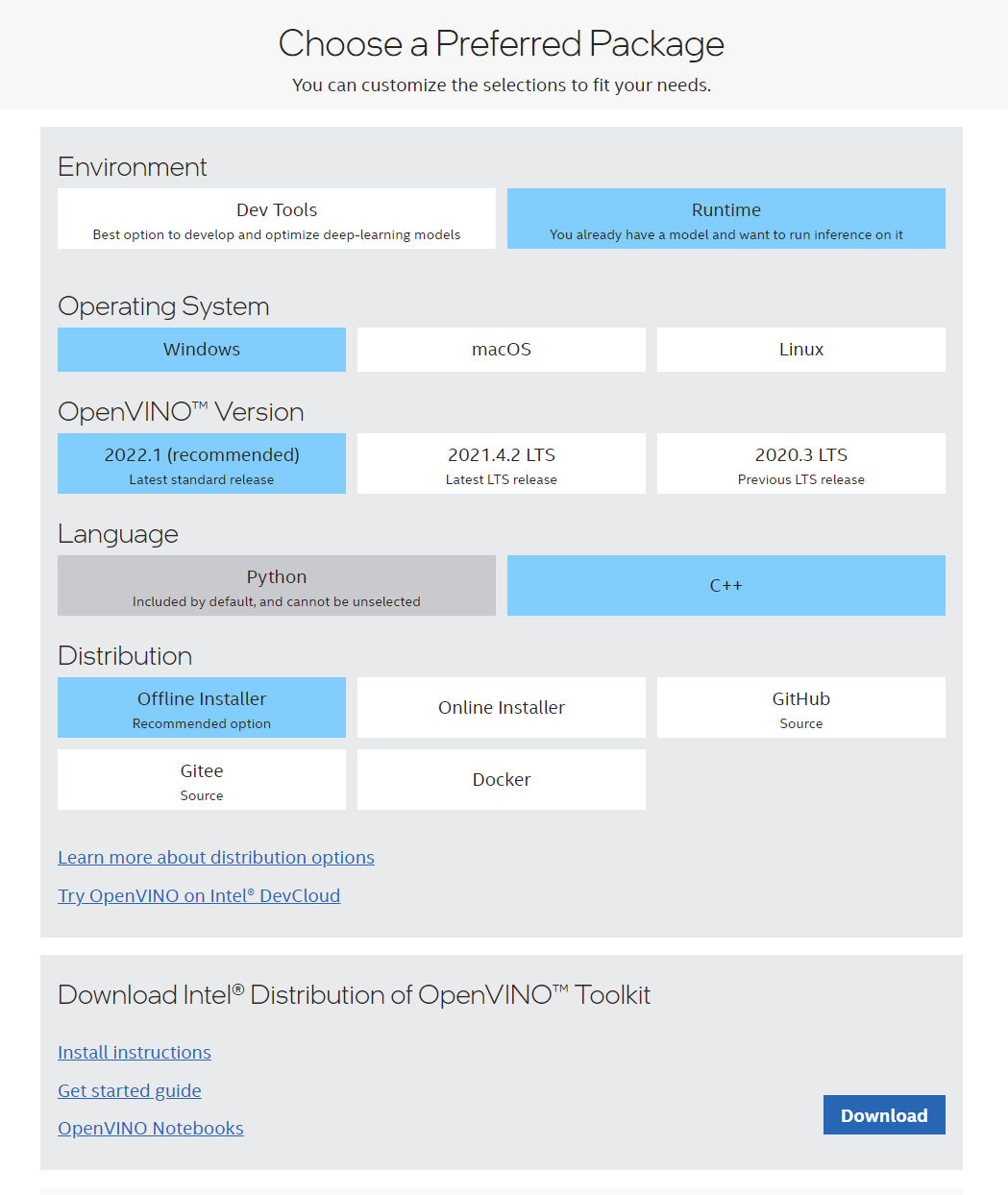


图1- 3 OpenVINOTM安装选择

选择完成后，点击Download，下载安装包。

（2）安装软件

在安装包下载完成后，直接安装软件即可，全程默认软件安装，不需要做任何修改。

（3）配置环境变量

在软件安装完成后，需要配置相关环境变量，防止每次使用都需要运行虚拟环境。右击我的电脑，进入属性设置，选择高级系统设置进入系统属性，点击环境变量，进入到环境变量设置，编辑系统变量下的Path变量，增加以下地址变量：

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\bin\intel64\Debug

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\bin\intel64\Release

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\3rdparty\tbb\bin

该路径为默认安装路径，如果不更改安装地址直接使用上方路径即可，如果进行了修改，请将< C:\Program Files (x86)\Intel\>替换为更改的安装路径。

### OpenCV安装

由于最新版的OpenVINOTM 2022.1 版本不在默认附带OpenCV工具，所以我们需要额外安装OpenCV工具。

（1）下载并安装OpenCV

访问OpenCV官网 <https://opencv.org/> ，选择Library下的Releases，进入到下载页面，或直接访问<https://opencv.org/releases/> 进入下载页面。

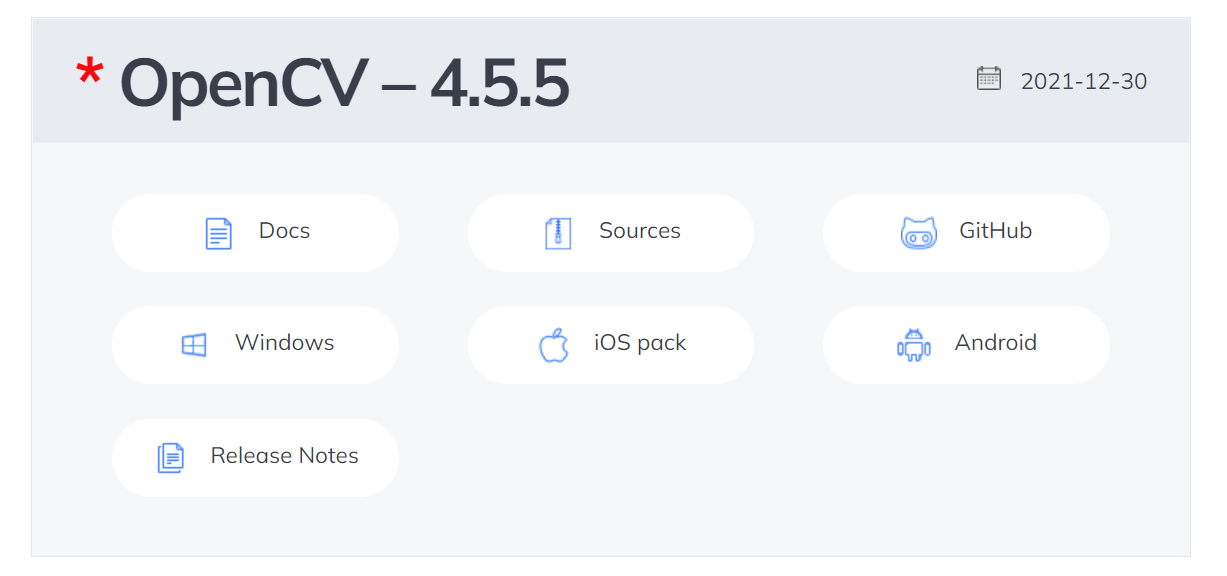


图1- 4 OpenCV-4.5.5 版本页面

根据负载使用情况，选择Windows版本，如图1- 4所示，跳转页面后，下载文件名为：opencv-4.5.5-vc14\_vc15.exe。下载完成后，直接双击打开安装文件，安装完成后，打开安装文件夹，该文件夹下 build、sources文件夹以及LICENSE相关文件，我们所使用的文件在build文件夹中。

（2）配置Path环境变量

右击我的电脑，进入属性设置，选择高级系统设置进入系统属性，点击环境变量，进入到环境变量设置，编辑系统变量下的Path变量，增加以下地址变量：

E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\bin

E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib

E:\OpenCV Source\opencv-4.5.5\build\include

E:\OpenCV Source\opencv-4.5.5\build\include\opencv2

其中<E:\OpenCV Source\opencv-4.5.5>为本机安装OpenCV安装路径。安装完成后，如想在VS中使用，还需要再VS编辑器中设置相关配置，具体配置会在下文中说明。

## OpenVINOTM推理模型与测试数据集

### 模型种类与下载方式

为了测试该项目，我们提供并整合了训练好的Paddlepaddle模型，主要针对PaddleClas以及PaddleDetection现有的模型，提供了PaddleClas下的花卉分类模型以及PaddleDetection中的Vehicle Detection模型，并针对该模型，提供了pdmodel、onnx以及IR格式。

该项目所使用的测试模型以及数据集，均可以在本文下的gitee上下载，下载链接为：

### PaddleDetection模型

PaddleDetection为飞桨PaddlePaddle的端到端目标检测套件，提供多种主流目标检测、实例分割、跟踪、关键点检测算法，配置化的网络模块组件、数据增强策略、损失函数等。该项目在此处主要使用的为PaddleDetection应用中的目标检测功能，使用的网络为YOLOv3网络，表1- 1 给出了YOLOv3网络输出与输入的相关信息。

表1- 1 YOLOv3模型输入与输出节点信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 节点类型 | 节点名 | 节点形状 | 数据类型 | 备注 |
| Input | image | [None, 3, H, W] | float32 | 输入网络的图像 |
| im\_shape | [None, 2] | float32 | 图像经过resize后的大小 |
| scale\_factor | [None, 2] | float32 | 输入图像比真实图像大小 |
| Output | multiclass\_nms3\_0.tmp\_0 | [Num, 6] | float32 | 识别结果 |
| multiclass\_nms3\_0.tmp\_2 | [None] | int32 | 识别结果数量 |

注：None表示batch维度，H、W 分别为图片的高和宽，Num表示识别结果的数量。

本次测试使用的为PaddleDetection中Vehicle Detection模型，我们可以在PaddleDetection gitee上下载。该模型输入图片要求为3×608×608大小，输出为预测框信息，其信息组成为[class\_id, score, x1, y1, x2, y2]，分别代表分类编号、分类得分以及预测框对角顶点坐标。

### PaddleClas 模型

飞桨图像识别套件PaddleClas是飞桨为工业界和学术界提供的的一个图像识别任务的工具集，该模型经过数据集训练，可以识别多种物品。在该项目中，我们使用flower数据集，使用ResNet50网络训练识别102种花卉，关于该模型的输入与输出节点信息如所示

表1- 2 ResNet50模型输入与输出节点信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 节点类型 | 节点名 | 节点形状 | 数据类型 | 备注 |
| Input | x | [None, 3, H, W] | float32 | 输入网络的图像 |
| Output | softmax\_1.tmp\_0 | [None, Class] | float32 | 各个识别结果概率 |

注：None表示batch维度，H、W 分别为图片的高和宽，Class表示分类数量。

花卉训练模型要求图片输入为3×224×224大小，输出结果为102中预测结果概率。

## 创建OpenVINOTM 方法C++动态链接库

### 新建解决方案以及项目文件

打开vs2022，首先新建一个C++空项目文件，并将同时新建一个解决方案命名为：OpenVinoSharp，用于存放后续其他项目文件。将C++项目命名为：CppOpenVinoAPI。

进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的文件的添加。具体操作如图1- 5所示。

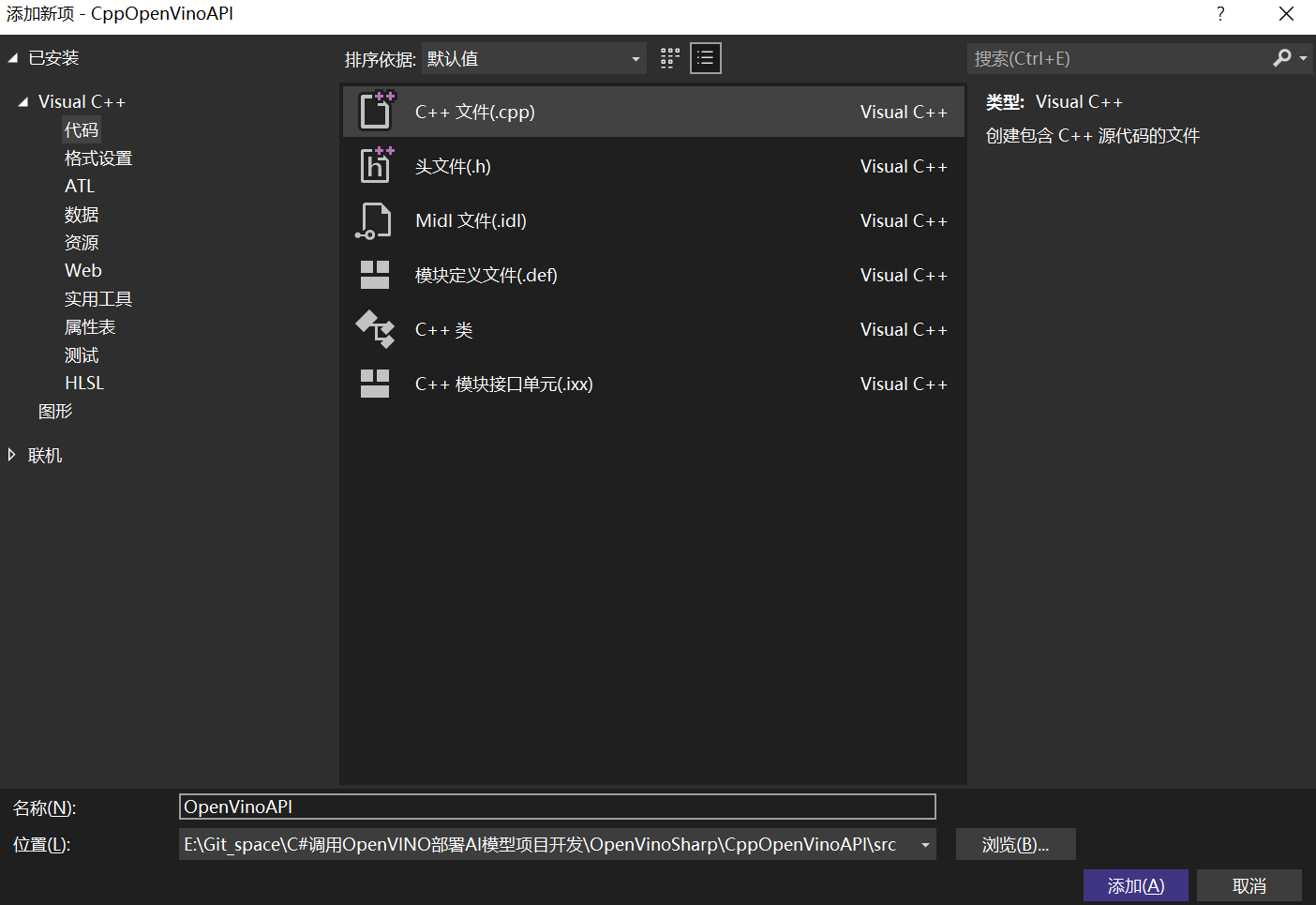


图1- 5 新建项目解决方案及C++项目

本次我们需要添加OpenVinoAPIcpp以及Source.def两个文件，如图1- 6所示。

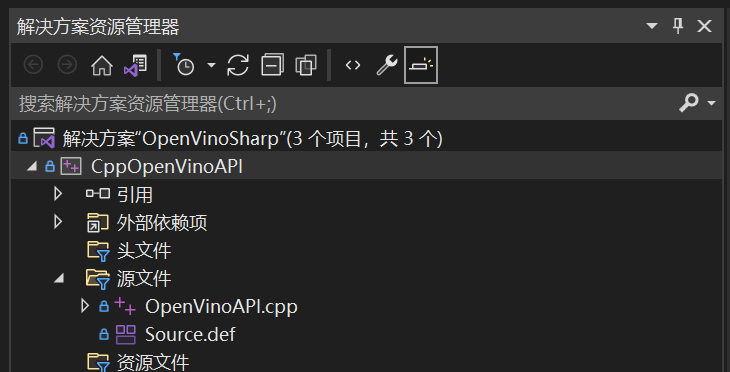


图1- 6 CppOpenVinoAPIl函数方法所需文件

### 配置C++项目属性

右击项目，点击属性，进入到属性设置，此处需要设置项目的配置类型包含目录、库目录以及附加依赖项，本次项目选择Release模式下运行，因此以Release情况进行配置。

（1）设置配置与平台

进入属性设置后，在最上面，将配置改为Release，平台改为x64。具体操作如图1- 7所示。

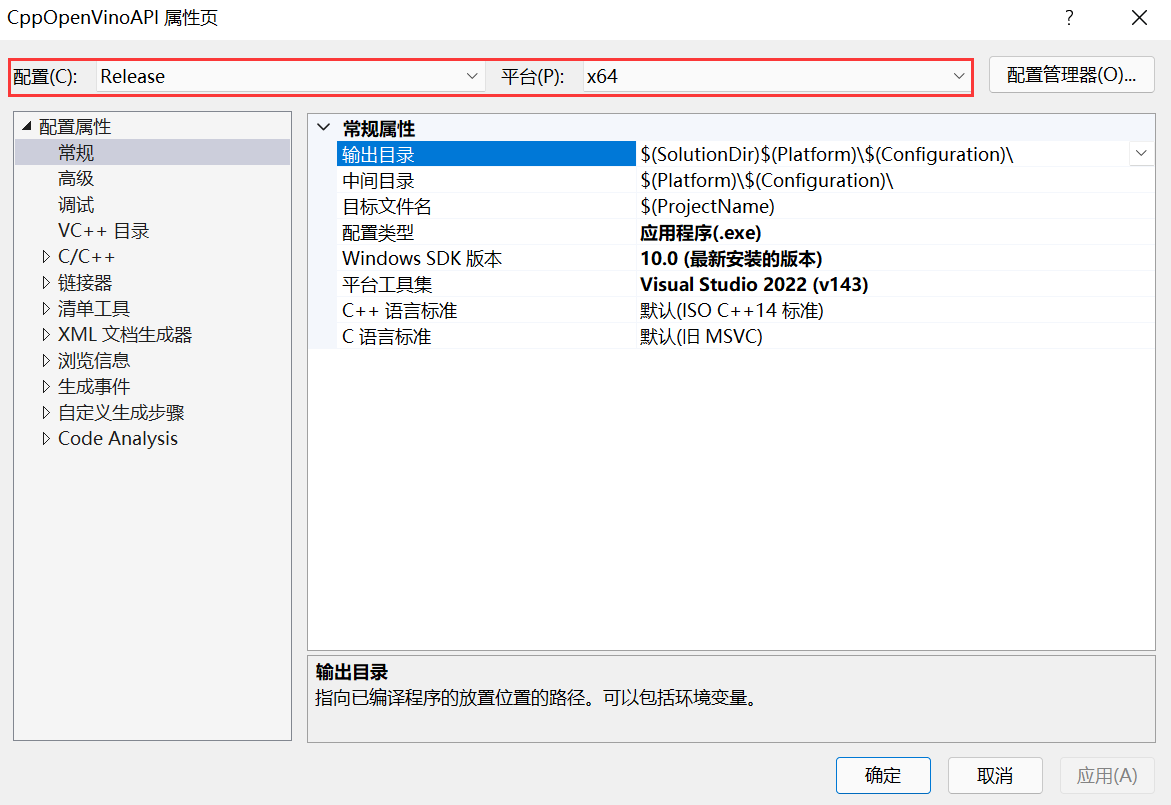


图1- 7 C++项目属性配置与平台设置

（2）设置常规属性

常规设置下，点击输出目录，将输出位置设置为<dll\>，即将生成文件放置在项目文件夹下的dll文件夹下；其次将目标文件名修改为：OpenVinoSharp；最后将配置类型改为：动态库(.dll)，让其生成dll文件。具体操作如图1- 8所示。

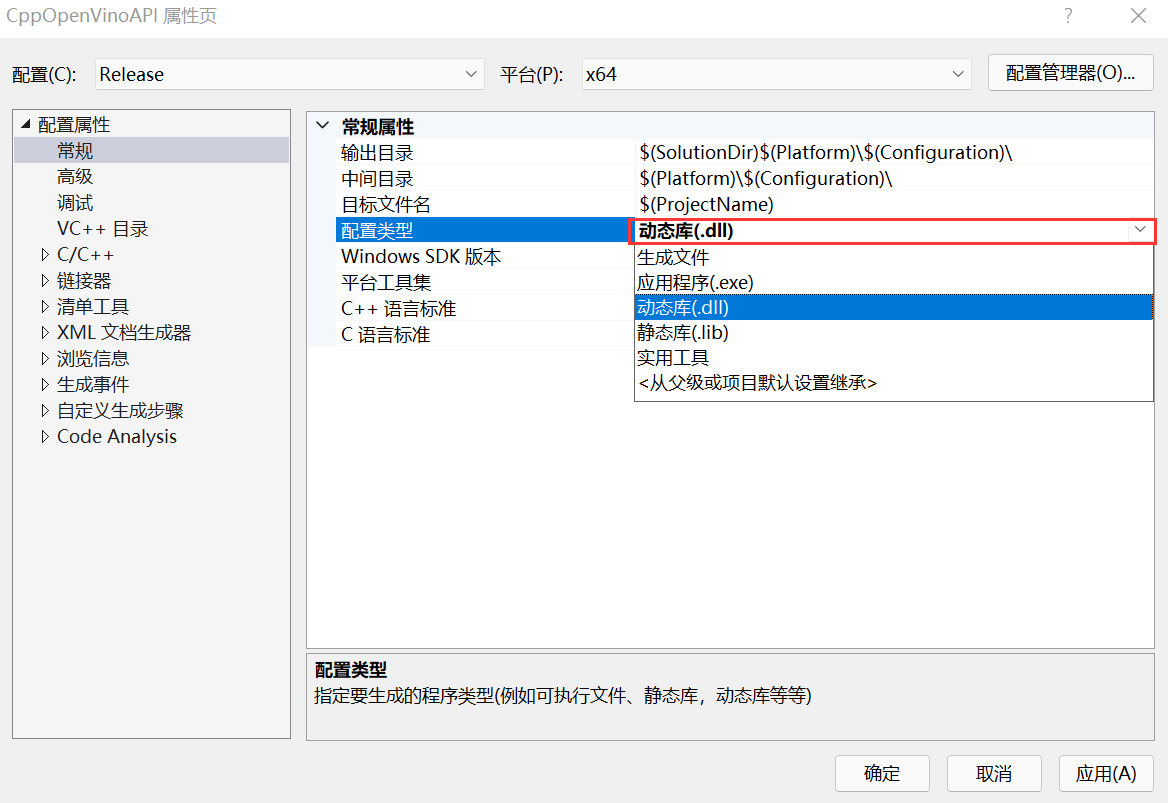
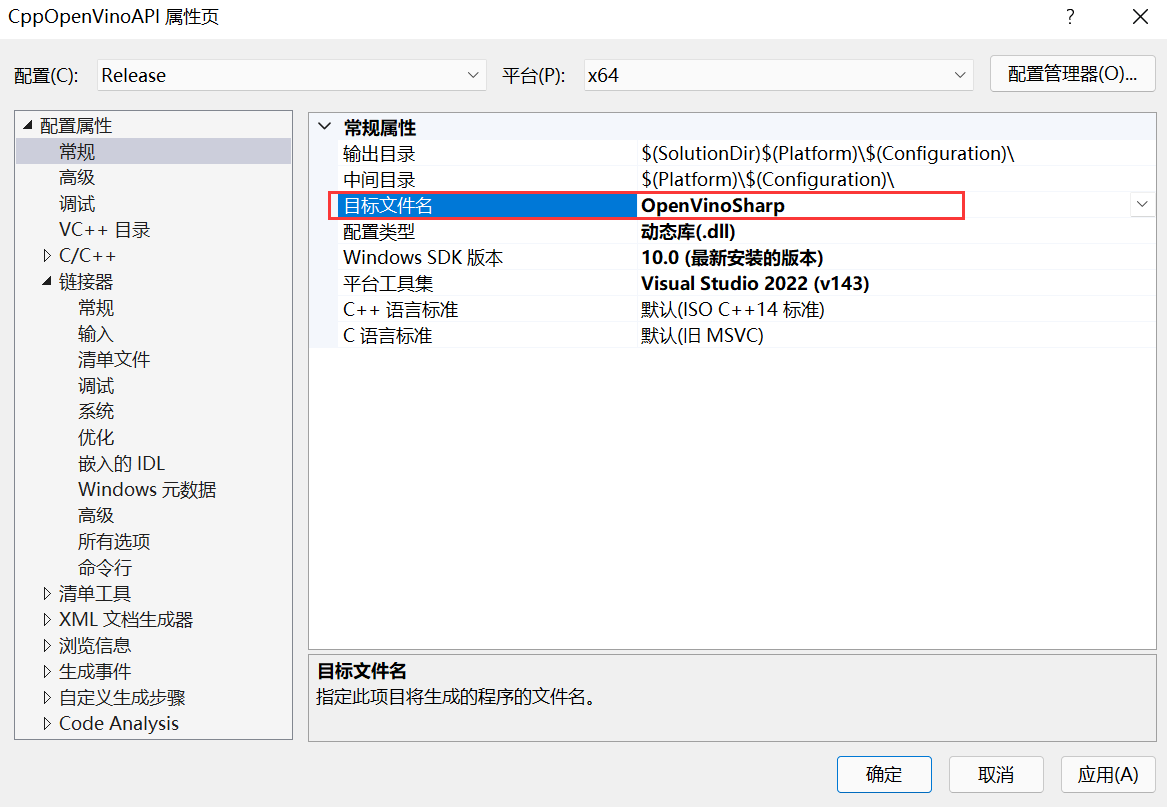
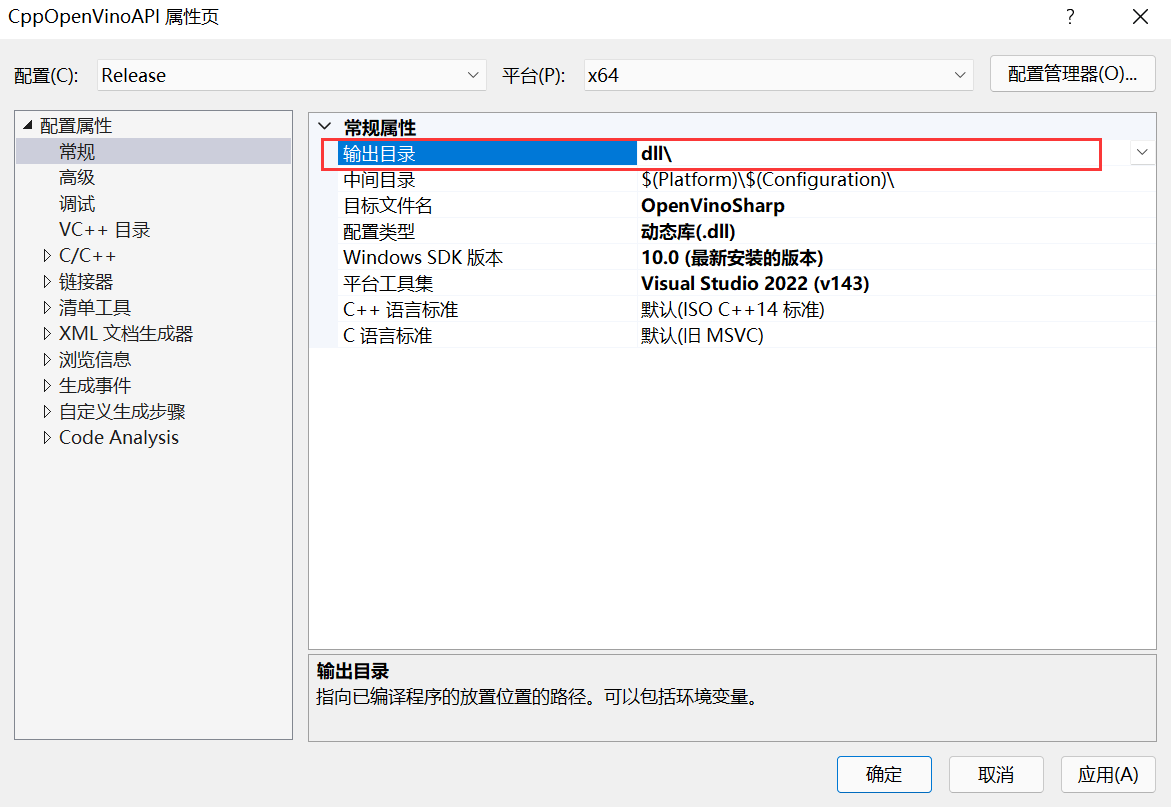


图1- 8 C++项目常规属性设置

（3）设置包含目录

点击VC++目录，然后点击包含目录，进行编辑，在弹出的新页面中，添加以下路径：

E:\OpenCV Source\opencv-4.5.5\build\include

E:\OpenCV Source\opencv-4.5.5\build\include\opencv2

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\include

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\include\ie

其中路径<E:\OpenCV Source\opencv-4.5.5\>为OpenCV的安装路径，<C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\>为OpenVINO安装路径，个人根据自己安装的位置及版本确定。操作过程如图1- 9所示。

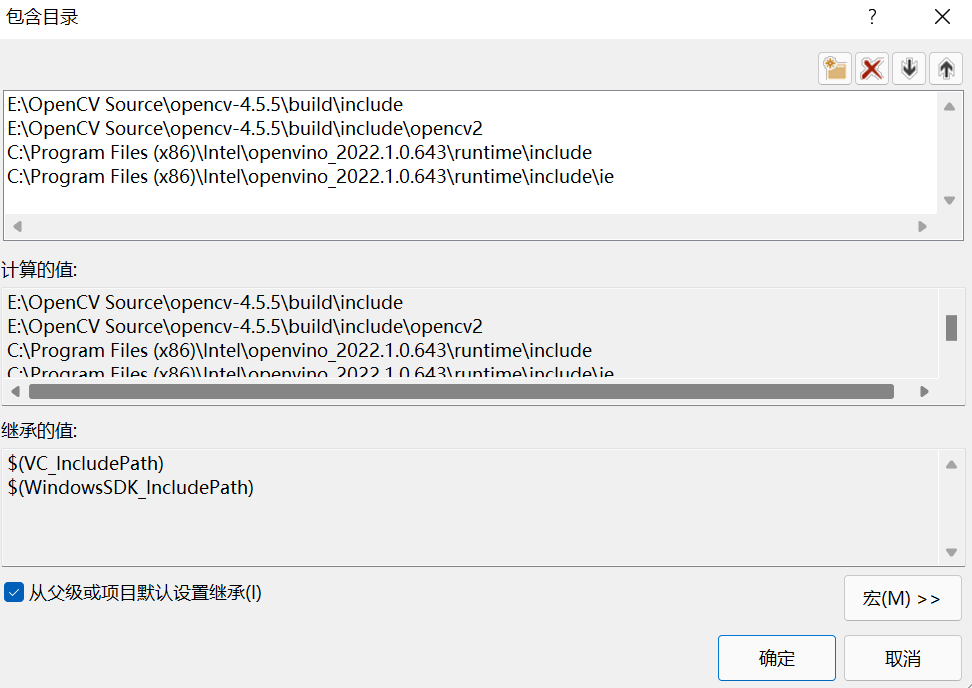
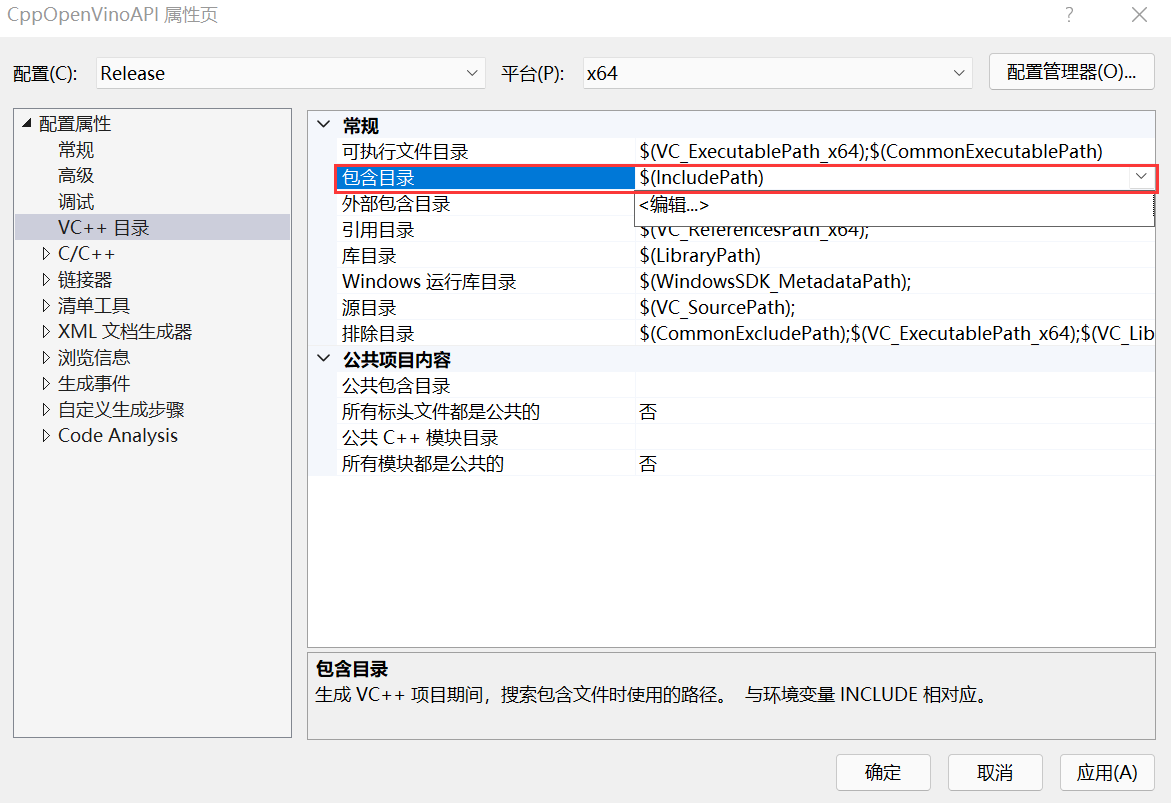


图1- 9 C++项目属性库目录设置

（4）设置库目录

同样的方式，在VC++目录下，点击下方的库目录，点击编辑，在弹出来的页面中增加以下路径：

E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib

C:\Program Files (x86)\Intel\openvino\_2022.1.0.643\runtime\lib\intel64\Release

如果时配置Debug模式，则需要将Release文件路径改为Debug即可。

（5）设置附加依赖项

点击展开链接器，点击输入，在附加依赖项中点击编辑，在弹出来的新的页面，添加以下文件名：

opencv\_world455.lib

openvino.lib

具体操作步骤参考如图1- 10所示。新版OpenCV与OpenVINOTM 都将依赖库文件合成到了一个文件中，这极大地简化了使用，如果使用老版本的，需要将所有的.lib文件放置在此处即可。

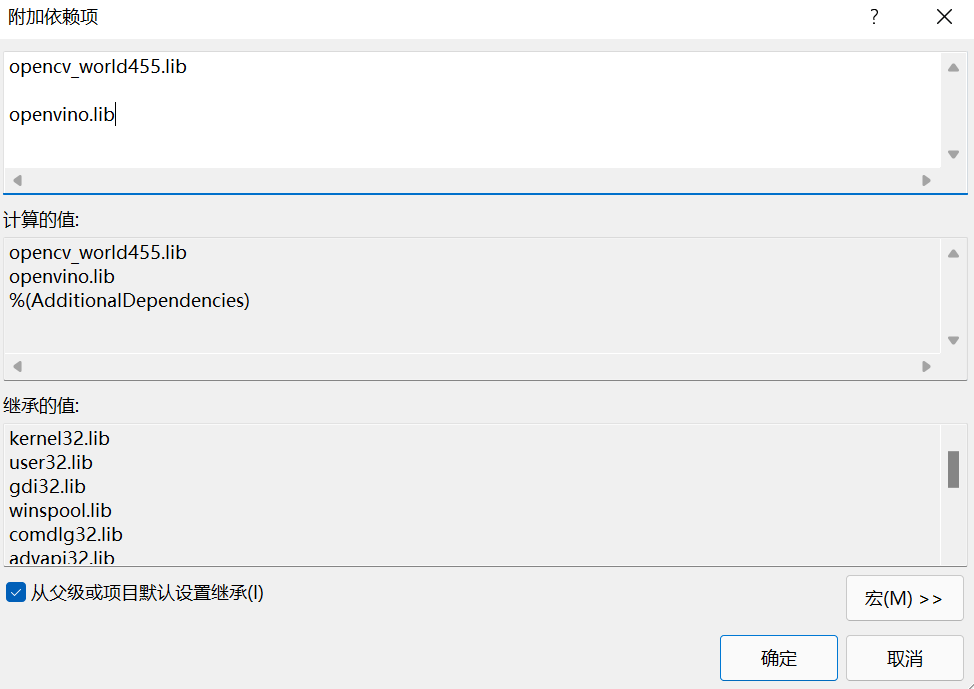
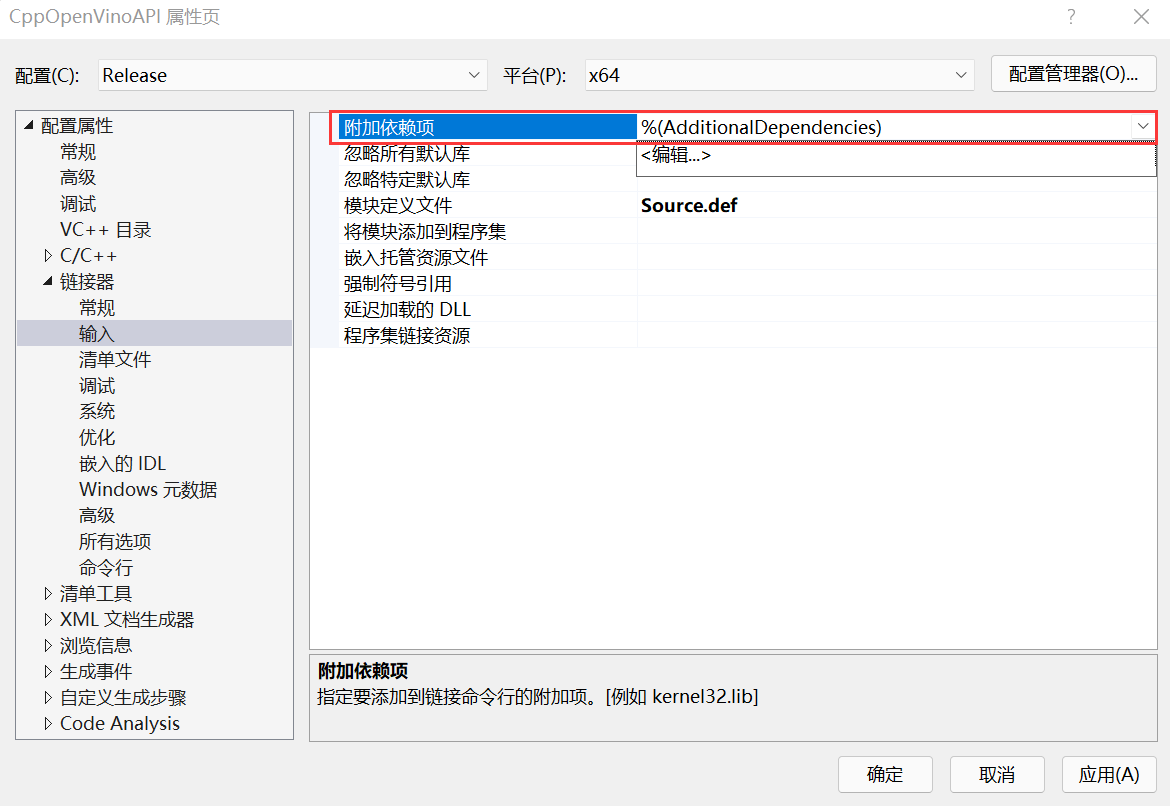


图1- 10 C++项目属性附加依赖项设置

### 编写C++代码

**（1）推理引擎结构体**

Core是OpenVINOTM 工具套件里的推理核心类，该类下包含多个方法，可用于创建推理中所使用的其他类。在此处，需要在各个方法中传递的仅仅是所使用的几个变量，因此选择构建一个推理引擎结构体，用于存放各个变量。

// @brief 推理核心结构体

typedef struct openvino\_core {

ov::Core core; // core对象

std::shared\_ptr<ov::Model> model\_ptr; // 读取模型指针

ov::CompiledModel compiled\_model; // 模型加载到设备对象

ov::InferRequest infer\_request; // 推理请求对象

} CoreStruct;

其中Core是OpenVINOTM 工具套件里的推理机核心，该模块只需要初始化； shared\_ptr<ov::Model>是读取本地模型的方法，新版更新后，该方法发生了较大改动，可支持读取Paddlepaddle飞桨模型、onnx模型以及IR模型；CompiledModel指的是一个已编译的模型类，其主要是将读取的本地模型应用多个优化转换，然后映射到计算内核，由所指定的设备编译模型；InferRequest是一个推理请求类，在推理中主要用于对推理过程的操作。

**（2）接口方法规划**

经典的OpenVINOTM 进行模型推理，一般需要八个步骤，主要是：初始化Core对象、读取本地推理模型、配置模型输入&输出、载入模型到执行硬件、创建推理请求、准备输入数据、执行推理计算以及处理推理计算结果。我们根据原有的八个步骤，对步骤进行重新整合，并根据推理步骤，调整方法接口。

对于方法接口，主要设置为：推理初始化、配置输入数据形状、配置输入数据、模型推理、读取推理结果数据以及删除内存地址六个大类，其中配置输入数据形状要细分为配置图片数据形状以及普通数据形状，配置输入数据要细分为配置图片输入数据与配置普通数据输入，读取推理结果数据细分为读取float数据和int数据，因此，总共有6类方法接口，9个方法接口。

**（3）初始化推理模型**

OpenVINOTM推理引擎结构体是联系各个方法的桥梁，后续所有操作都是在推理引擎结构体中的变量上操作的，为了实现数据在各个方法之间的传输，因此在创建推理引擎结构体时，采用的是创建结构体指针，并将创建的结构体地址作为函数返回值返回。推理初始化接口主要整合了原有推理的初始化Core对象、读取本地推理模型、载入模型到执行硬件和创建推理请求步骤，并将这些步骤所创建的变量放在推理引擎结构体中。

初始化推理模型接口方法为：

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall core\_init(const wchar\_t\* model\_file\_wchar, const wchar\_t\* device\_name\_wchar);

该方法返回值为CoreStruct结构体指针，其中model\_file\_wchar为推理模型本地地址字符串指针，device\_name\_wchar为模型运行设备名指针，在后面使用上述变量时，需要将其转换为string字符串，利用wchar\_to\_string()方法可以实现将其转换为字符串格式：

std::string model\_file\_path = wchar\_to\_string(model\_file\_wchar);

std::string device\_name = wchar\_to\_string(device\_name\_wchar);

模型初始化功能主要包括：初始化推理引擎结构体和对结构体里面定义的其他变量进行赋值操作，其主要是利用InferEngineStruct中创建的Core类中的方法，对各个变量进行初始化操作：

CoreStruct\* p = new CoreStruct(); // 创建推理引擎指针

p->model\_ptr = p->core.read\_model(model\_file\_path); // 读取推理模型

p->compiled\_model = p->core.compile\_model(p->model\_ptr, "CPU"); // 将模型加载到设备

p->infer\_request = p->compiled\_model.create\_infer\_request(); // 创建推理请求

**（4）配置输入数据形状**

在新版OpenVINOTM 2022.1 中，新增加了对Paddlepaddle模型以及onnx模型的支持，Paddlepaddle模型不支持指定指定默认bath通道数量，因此需要在模型使用时指定其输入；其次，对于onnx模型，也可以在转化时不指定固定形状，因此在配置输入数据前，需要配置输入节点数据形状。其方法接口为：

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall set\_input\_image\_sharp(void\* core\_ptr, const wchar\_t\* input\_node\_name\_wchar, size\_t \* input\_size);

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall set\_input\_data\_sharp(void\* core\_ptr, const wchar\_t\* input\_node\_name\_wchar, size\_t \* input\_size);

由于需要配置图片数据输入形状与普通数据的输入形状，在此处设置了两个接口，分别设置两种不同输入的形状。该方法返回值是CoreStruct结构体指针，但该指针所对应的数据中已经包含了对输入形状的设置。第一个输入参数core\_ptr是CoreStruct指针，在当前方法中，我们要读取该指针，并将其转换为CoreStruct类型：

CoreStruct\* p = (CoreStruct\*)core\_ptr;

input\_node\_name\_wchar 为待设置网络节点名，input\_size 为形状数据数组，对图片数据，需要设置 [batch, dim, height, width] 四个维度大小，所以input\_size数组传入4个数据，其设置在形状主要使用Tensor类下的set\_shape()方法：

std::string input\_node\_name = wchar\_to\_string(input\_node\_name\_wchar); // 将节点名转为string类型

ov::Tensor input\_image\_tensor = p->infer\_request.get\_tensor(input\_node\_name); // 读取指定节点Tensor

input\_image\_tensor.set\_shape({ input\_size[0],input\_size[1],input\_size[2],input\_size[3] }); // 设置节点数据形状

**（5）配置输入数据**

在新版OpenVINOTM 中，Tensor类的T\* data()方法，其返回值为当前节点Tensor的数据内存地址，通过填充Tensor的数据内存，实现推理数据的输入。对于图片数据，其最终也是将其转为一维数据进行输入，不过为方便使用，此处提供了配置图片数据和普通数据的接口，对于输入为图片的方法接口：

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall load\_image\_input\_data(void\* core\_ptr, const wchar\_t\* input\_node\_name\_wchar, uchar \* image\_data, size\_t image\_size)；

该方法返回值是CoreStruct结构体指针，但该指针所对应的数据中已经包含了加载的图片数据。第一个输入参数core\_ptr是CoreStruct指针，在当前方法中，我们要读取该指针，并将其转换为CoreStruct类型；第二个输入参数input\_node\_name\_wchar为待填充节点名，先将其转为string字符串：

std::string input\_node\_name = wchar\_to\_string(input\_node\_name\_wchar);

在该项目中，我们主要使用的是以图片作为模型输入的推理网络，模型主要的输入为图片的输入。其图片数据主要存储在矩阵image\_data和矩阵长度image\_size两个变量中。需要对图片数据进行整合处理，利用创建的data\_to\_mat () 方法，将图片数据读取到OpenCV中：

cv::Mat input\_image = data\_to\_mat(image\_data, image\_size);

接下来就是配置网络图片数据输入，对于节点输入是图片数据的网络节点，其配置网络输入主要分为以下几步：

**首先，获取网络输入图片大小。**

使用InferRequest类中的get\_tensor ()方法，获取指定网络节点的Tensor，其节点要求输入大小在Shape容器中，通过获取该容器，得到图片的长宽信息：

ov::Tensor input\_image\_tensor = p->infer\_request.get\_tensor(input\_node\_name);

int input\_H = input\_image\_tensor.get\_shape()[2]; //获得"image"节点的Height

int input\_W = input\_image\_tensor.get\_shape()[3]; //获得"image"节点的Width

**其次，按照输入要求，处理输入图片。**

在这一步，我们除了要按照输入大小对图片进行放缩之外，还要根据PaddlePaddle对模型输入的要求进行处理。因此处理图片其主要分为交换RGB通道、放缩图片以及对图片进行归一化处理。在此处我们借助OpenCV来实现。

OpenCV读取图片数据并将其放在Mat类中，其读取的图片数据是BGR通道格式，PaddlePaddle要求输入格式为RGB通道格式，其通道转换主要靠一下方式实现：

cv::cvtColor(input\_image, blob\_image, cv::COLOR\_BGR2RGB);

接下来就是根据网络输入要求，对图片进行压缩处理：

cv::resize(blob\_image, blob\_image, cv::Size(input\_H, input\_W), 0, 0, cv::INTER\_LINEAR);

最后就是对图片进行归一化处理，其主要处理步骤就是减去图像数值均值，并除以方差。查询PaddlePaddle模型对图片的处理，其均值mean = [0.485, 0.456, 0.406]，方差std = [0.229, 0.224, 0.225]，利用OpenCV中现有函数，对数据进行归一化处理：

std::vector<float> mean\_values{ 0.485 \* 255, 0.456 \* 255, 0.406 \* 255 };

std::vector<float> std\_values{ 0.229 \* 255, 0.224 \* 255, 0.225 \* 255 };

std::vector<cv::Mat> rgb\_channels(3);

cv::split(blob\_image, rgb\_channels); // 分离图片数据通道

for (auto i = 0; i < rgb\_channels.size(); i++){

//分通道依此对每一个通道数据进行归一化处理

rgb\_channels[i].convertTo(rgb\_channels[i], CV\_32FC1, 1.0 / std\_values[i], (0.0 - mean\_values[i]) / std\_values[i]);

}

cv::merge(rgb\_channels, blob\_image); // 合并图片数据通道

**最后，将图片数据输入到模型中。**

在此处，我们重写了网络赋值方法，并将其封装到 fill\_tensor\_data\_image(ov::Tensor& input\_tensor, const cv::Mat& input\_image)方法中，input\_tensor为模型输入节点Tensor类，input\_image为处理过的图片Mat数据。因此节点赋值只需要调用该方法即可：

fill\_tensor\_data\_image(input\_image\_tensor, blob\_image);

对于普通数据的输入，其方法接口如下：

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall load\_input\_data(void\* core\_ptr, const wchar\_t\* input\_node\_name\_wchar, float\* input\_data);

与配置图片数据不同点，在于输入数据只需要输入input\_data数组即可。其数据处理哦在外部实现，只需要将处理后的数据填充到输入节点的数据内存中即可，通过调用自定义的fill\_tensor\_data\_float(ov::Tensor& input\_tensor, float\* input\_data, int data\_size) 方法即可实现：

std::string input\_node\_name = wchar\_to\_string(input\_node\_name\_wchar);

ov::Tensor input\_image\_tensor = p->infer\_request.get\_tensor(input\_node\_name); // 读取指定节点tensor

int input\_size = input\_image\_tensor.get\_shape()[1]; //获得输入节点的长度

fill\_tensor\_data\_float(input\_image\_tensor,input\_data, input\_size); // 将数据填充到tensor数据内存上

**（6）模型推理**

上一步中我们将推理内容的数据输入到了网络中，在这一步中，我们需要进行数据推理，这一步中我们留有一个推理接口：

extern "C" \_\_declspec(dllexport) void\* \_\_stdcall core\_infer(void\* core\_ptr)

进行模型推理，只需要调用CoreStruct结构体中的infer\_request对象中的infer()方法即可：

CoreStruct\* p = (CoreStruct\*)core\_ptr;

p->infer\_request.infer();

**（7）读取推理数据**

上一步我们对数据进行了推理，这一步就需要查询上一步推理的结果。对于我们所使用的模型输出，主要有float数据和int数据，对此，留有了两种数据的查询接口，其方法为：

extern "C" \_\_declspec(dllexport) void \_\_stdcall read\_infer\_result\_F32(void\* core\_ptr, const wchar\_t\* output\_node\_name\_wchar, int data\_size, float\* infer\_result);

extern "C" \_\_declspec(dllexport) void \_\_stdcall read\_infer\_result\_I32(void\* core\_ptr, const wchar\_t\* output\_node\_name\_wchar, int data\_size, int\* infer\_result);

其中data\_size为读取数据长度，infer\_result 为输出数组指针。读取推理结果数据与加载推理数据方式相似，依旧是读取输出节点处数据内存的地址：

const ov::Tensor& output\_tensor = p->infer\_request.get\_tensor(output\_node\_name);

const float\* results = output\_tensor.data<const float>();

针对读取整形数据，其方法一样，只是在转换类型时，需要将其转换为整形数据即可。我们读取的初始数据为二进制数据，因此要根据指定类型转换，否则数据会出现错误。将数据读取出来后，将其放在数据结果指针中，并将所有结果赋值到输出数组中：

for (int i = 0; i < data\_size; i++) {

\*inference\_result = results[i];

inference\_result++;

}

**（8）删除推理核心结构体指针**

推理完成后，我们需要将在内存中创建的推理核心结构地址删除，防止造成内存泄露，影响电脑性能，其接口该方法为：

extern "C" \_\_declspec(dllexport) void \_\_stdcall core\_delet(void\* core\_ptr);

在该方法中，我们只需要调用delete命令，将结构体指针删除即可。

### 编写模块定义文件

我们在定义接口方法时，在原有方法的基础上，增加了extern "C" 、 \_\_declspec(dllexport) 以及\_\_stdcall 三个标识，其主要原因是为了让编译器识别我们的输出方法。其中，extern "C"是指示编译器这部分代码按C语言（而不是C++）的方式进行编译；\_\_declspec(dllexport)用于声明导出函数、类、对象等供外面调用；\_\_stdcall是一种函数调用约定。通过上面三个标识，我们在C++种所写的接口方法，会在dll文件中暴露出来，并且可以实现在C#中的调用。

不过上面所说内容，我们在编辑器中可以通过模块定义文件(.def)所实现，在模块定义文件中，添加以下代码：

LIBRARY

"OpenVinoSharp"

EXPORTS

core\_init

set\_input\_image\_sharp

set\_input\_data\_sharp

load\_image\_input\_data

load\_input\_data

core\_infer

read\_infer\_result\_F32

read\_infer\_result\_I32

core\_delet

LIBRARY后所跟的为输出文件名，EXPORTS后所跟的为输出方法名。仅需要以上语句便可以替代extern "C" 、 \_\_declspec(dllexport) 以及\_\_stdcall的使用。

### 生成dll文件

前面我们将项目配置输出设置为了生成dll文件，因此该项目不是可以执行的exe文件，只能生成不能运行。右键项目，选择重新生成/生成。在没有错误的情况下，会看到项目成功的提示。可以看到dll文件在解决方案同级目录下\x64\Release\文件夹下。

使用dll文件查看器打开dll文件，如图1- 11所示；可以看到，我们创建的四个方法接口已经暴露在dll文件中。

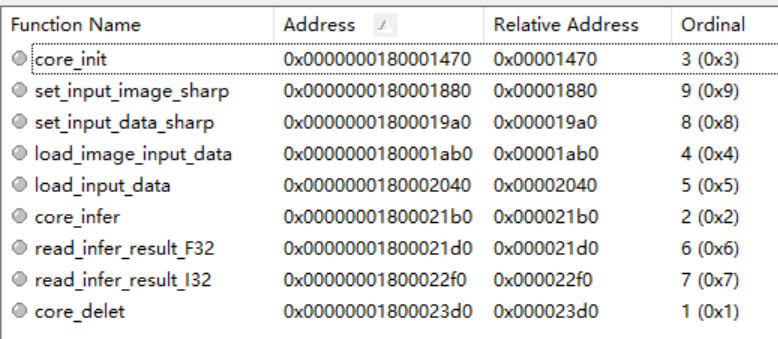


图1- 11 dll文件方法输出目录

## C#构建Core类

### 新建C#类库

右击解决方案，添加->新建项目，选择添加C#类库，项目名命名为OpenVinoSharp，项目框架根据电脑中的框架选择，此处使用的是.NET 5.0。新建完成后，然后右击项目，选择添加->新建项，选择类文件，添加Core.cs和NativeMethods.cs两个类文件。

### 引入dll文件中的方法

在NativeMethods.cs文件下，我们通过[DllImport()]方法，将dll文件中所有的方法读取到C#中。读取方式如下：

[DllImport(openvino\_dll\_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]

public extern static IntPtr core\_init(string model\_file, string device\_name);

其中openvino\_dll\_path为dll文件路径，CharSet = CharSet.Unicode代表支持中文编码格式字符串，CallingConvention = CallingConvention.Cdecl指示入口点的调用约定为调用方清理堆栈。

上述所列出的为初始化推理模型，dlii文件接口在匹配时，是通过方法名字匹配的，因此，方法名要保证与dll文件中一致。其次就是方法的参数类型要进行对应，在上述方法中，函数的返回值在C++中为void\* ，在C#中对应的为IntPtr类型，输入参数中，在C++中为wchar\_t\* 字符指针，在C#中对应的为string字符串。通过方法名与参数类型一一对应，在C#可以实现对方法的调用。其他方法的引用类似，在此处不在一一赘述，具体可以参照项目提供的源代码。

### 创建Core类

为了更方便地调用我们通过dll引入的OpenVINOTM 方法，减少使用时的函数方法接口，我们在C#中重新组建我们自己的推理类，命名为Class Core，其主要成员变量和方法如图1- 12所示。

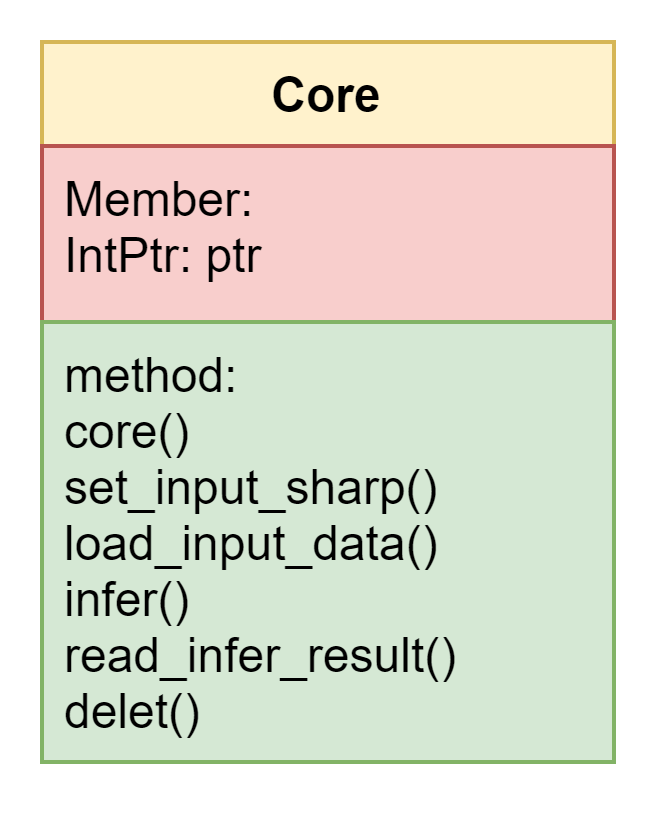


图1- 12 Core类图

在Core.类中，我们只需要创建一个地址变量，作为Core类的成员变量，用于接收接口函数返回的推理核心指针，该成员变量我们只需要在当前类下访问，因此将其设置为私有变量：

private IntPtr ptr = new IntPtr();

接下来，构建类的构造函数，在类的初始化时，我们需要输入模型地址以及设备类型，通过掉用dll文件中引入的方法，获取初始化指针，对成员变量进行赋值，实现类的初始化：

public Core(string model\_file, string device\_name) {

ptr = NativeMethods.core\_init(model\_file, device\_name);

}

然后构其中的方法，在构建设置数据输入形状时，我们需要提供的为节点名以及形状数据，为了简化该方法，我们合并了图片形状设置与普通数据形状设置接口，通过判断输入数组的长度，来确定是对那一个形状的设置：

public void set\_input\_sharp(string input\_node\_name, ulong[] input\_size);

对于该类中方法的构建，可以参考源码文件，在此处不做详述。并且其他方法构建方式基本相似，此处不在一一赘述，具体可以参考源码文档。

### 编译Core类库

右击项目，点击生成/重新生成，出现如下图1- 13所示，表示编译成功。

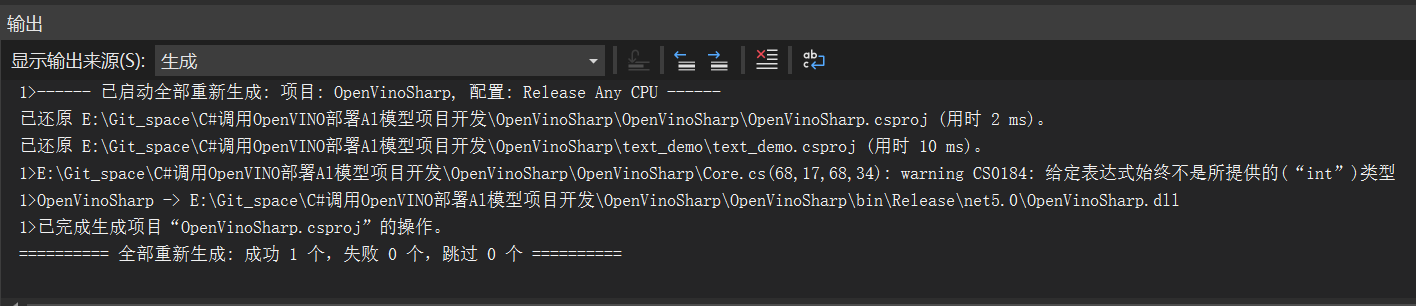


图1- 13 Core类编译输出

## C#实现OpenVINOTM方法的调用

### 新建C#项目

右击解决方案，添加->新建项目，选择添加C#控制台项目，项目框架根据电脑中的框架选择，此处使用的是.NET 5.0。

新建完成后，右击项目，选择属性，点击新页面中的生成，在常规下，将目标平台改为X64。具体操作如图1- 14所示。



图1- 14 C#项目设置

### 添加OpenCVsharp

右击项目，选择管理NuGet程序包，在新页面中选择浏览，在搜索框中输入opencvsharp3，在搜索结果中，找到OpenCvSharp3-AnyCPU，然后右侧点击安装，具体操作步骤如图1- 15所示。

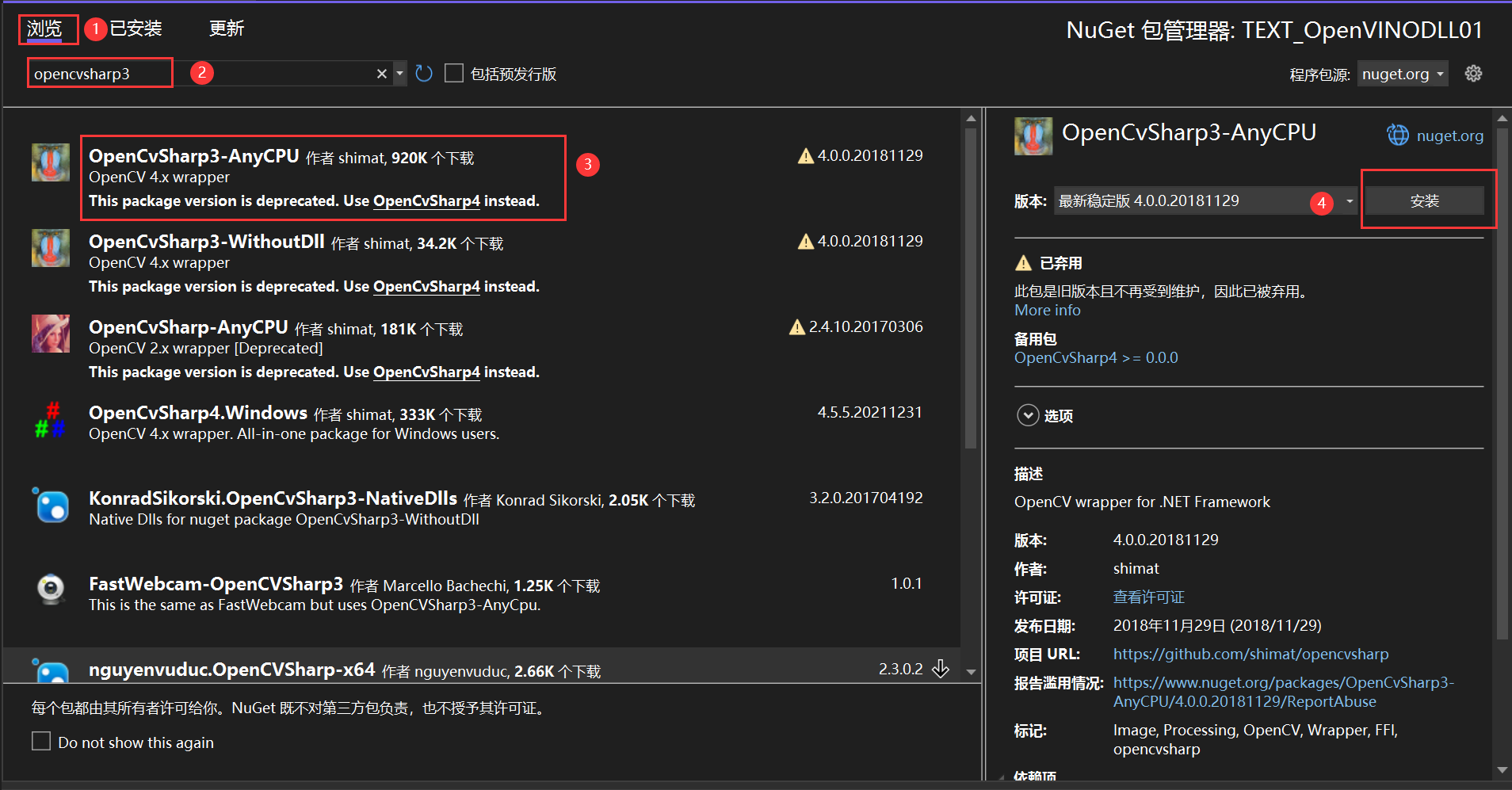


图1- 15 NuGet程序包安装

### 添加项目引用

上一步中我们将dll文件中的方法引入到C#中，并组建了Core类，在这一步中，我们主要通过调用Core类，进行Al模型的部署，所以需要引入上一步的项目。

右击当前项目，选择添加，选择项目引用，在出现的窗体中，选择上一步中创建的项目OpenVinoSharp，点击确定；然后在当前项目下，添加using OpenVinoSharp命名空间。具体操作如图1- 16所示。

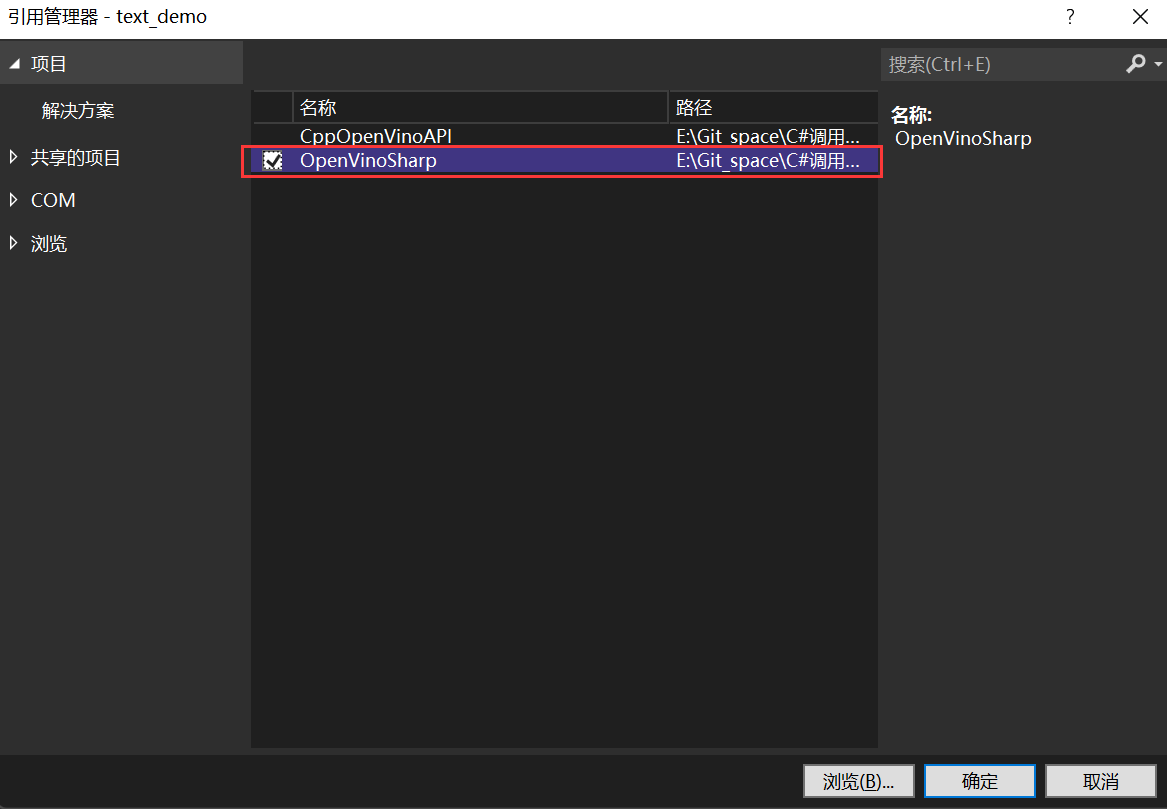


图1- 16 添加项目引用

### 编写代码测试花卉分类模型

在该项目中，我们提供了两种推理模型，此处我们以花卉分类模型为例，简介如何通过C#调用OpenVINOTM 进行Al模型的部署。

（1）引入相关变量

string device\_name = "CPU";

string model\_file = "E:/Text\_Model/flowerclas/flower\_rec.onnx";

string image\_file = "E:/Text\_dataset/flowers102/jpg/image\_00001.jpg";

string input\_node\_name = "x";

string output\_node\_name = "softmax\_1.tmp\_0";

为了让大家更加清晰的看懂后续代码，在此处对引入的相关变量进行解释：

device\_name：设备类型名称，可为CPU、GPU以及AUTO（均可）；

model\_file：模型地址，可以为onnx、pdmodel或者xml格式；

image\_file：测试图片地址；

input\_node\_name：输入模型节点名，当多输入时，可以为数组；

output\_node\_name：输出模型节点名，当多输出时，可以为数组。

（2）初始化Core类

在此处我们直接调用Core类的构造函数，进行初始化：

Core ie = new Core(model\_file\_paddle, device\_name);

（3）配置模型输入

花卉分类模型输入只有一个，即待分类花卉图片。如果我们调用的模型未指定输入大小，需要在输入数据前，调用模型输入数据形状设置方法，设置节点输入数据形状。图片数据为三维数组，再加一个batchsize，最终为四维数据，将形状数据放在数组中，调用set\_input\_sharp()方法：

ulong[] image\_sharp = new ulong[] { 1, 3, 224, 224 };

ie.set\_input\_sharp(input\_node\_name, image\_sharp);

对于图片数据，需要将其转为转为矩阵数据，在此处，我们可以直接使用opencvsharp中的编解码方法，将图片数据放置在byte数组中：

Mat image = new Mat(image\_file);

byte[] image\_data = new byte[2048 \* 2048 \* 3];

ulong image\_size = new ulong();

image\_data = image.ImEncode(".bmp");

image\_size = Convert.ToUInt64(image\_data.Length);

就最后调用Core类中的load \_input\_data()方法，将数据加载到推理网络中：

ie.load\_input\_data(input\_node\_name, image\_data, image\_size);

在配置完输入数据后，调用模型推理方法，对输入数据进行推理：

ie.infer();

接下来就是读取推理结果，对于模型的推理结果输出一般为数组数据，可以通过调用Core类中读取推理数据结果的方法，对与花卉分类模型的输出，其结果为长度为102的浮点型数据，所以直接调用read\_inference\_result<T> ()方法读取即可：

float[] result = new float[102];

result = ie.read\_infer\_result<float>(output\_node\_name, 102);

在读取推理数据时，我们一定要根据模型的书名读取正确的结果数据，因为如果超出实际输出长度，其结果数据会掺杂其他干扰数据。

最后一步就是处理输出数据。对于不同的推理模型，其结果处理方式是不同的，对于花卉分类模型，其输出为102种分类情况打分，因此，在处理数据时，需要找出得分最高的哪一类即可。在此处，我们提供了一个方法，该方法可以实现提取数组中前N个max数据的位置，通过调用该方法，我们可以获取分类结果中分数最高的几个结果，并将结果打印输出：

int[] index = find\_array\_max(result,5);

for (int i = 0; i < 5; i++){

Console.WriteLine("the index is {0} , the score is {1} ", index[i], result[index[i]]);

}

最终输出结果如图1- 17所示，该页面打印出来了推理结果预测分数最大的前五个分数和其对应的索引值，最后可以通过索引值查询flowers102\_label\_list.txt文件中对应的花卉名称。

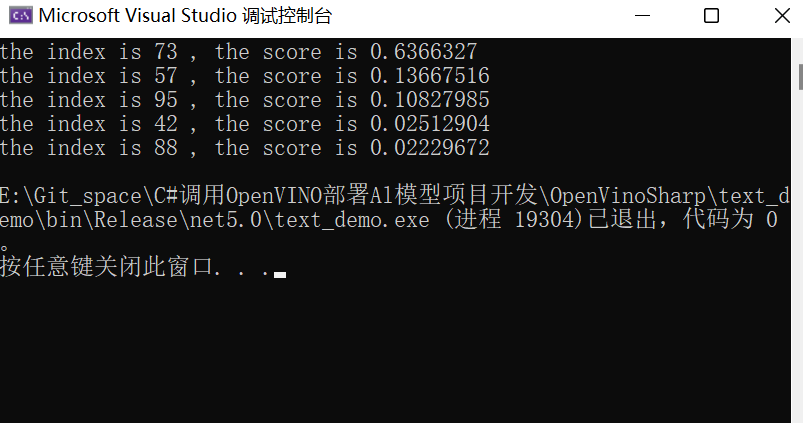


图1- 17 花卉分类结果

在程序最后，我们该需要将前面在内存上创建推理引擎结构体进行删除，只需要调用Core类下的delet()即可。

### 编写代码测试车辆识别模型

对于车辆识别模型此处不再进行详细讲解，具体实现可以参考源码文件，此处只对一些不同点进行分析。

在配置输入时，除了需要配置图片数据输入，还需要配置图片长宽数据以及长宽缩放比例数据，在配置时，只需要将数据放置在数组中，通过调用load\_input\_data()方法实现，对于设置缩放比例数据输入，如下所示：

float scale\_h = 608.00f / image.Height;

float scale\_w = 608.00f / image.Width;

float[] scale\_factor = new float[] { scale\_h, scale\_w };

ie.load\_input\_data(input\_node\_name[1], scale\_factor);

对于该模型推理结果数据，总共有两个节点输出，一个是识别结果数量，一个为识别结果信息。对于识别结果数量，其数据类型为整形数据，对于单图片输入，只需要读取一位即可，利用该数据，确定识别结果信息长度。识别结果信息为6列N行数据，在数据读取时，我们将其转化为一维数据，所以在处理数据时，以6位数据为一组，进行处理。

识别结果信息数据中，第1位为识别标签，第2位为识别得分，第3位到第6位四个数据为位置矩形框对角点坐标，通过每6位读取一次数据，获取识别结果。在此处，我们提供了专门的结果处理方法，通过该方法们可以实现直接将结果绘制在原图片上：

image = draw\_image\_resule(image, resule\_num[0], result, lable, 0.2f);

其中image为原图片，resule\_num[0]为识别结果数量，result为识别结果数组，lable为结果标签，0.2f为评价得分下限。通过结果处理，将识别结果标注在图片中，并把识别结果以及得分情况打印在图片中，最终识别结果如图1- 18所示。

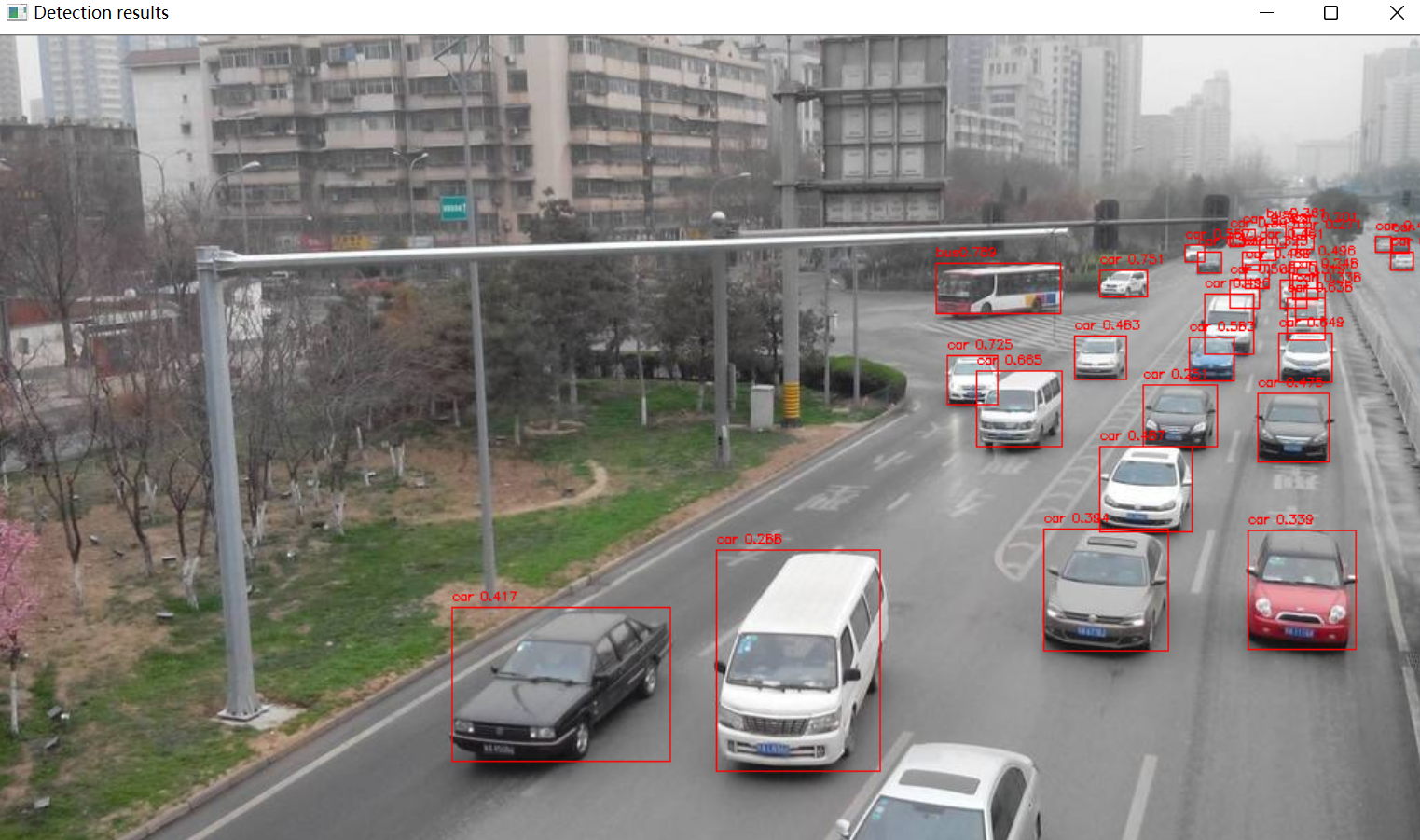


图1- 18 车辆类型识别结果输出

## 程序时间分析

为了对比C++、C#以及Python这三个平台下调用OpenVINOTM所使用的时间，我们通过测试flower\_clas以及vehicle\_yolov3\_darknet模型运行时间进行对比，在同一台电脑相同运行环境之下，以及对模型的处理方式在不同编程语言下尽量做到相同，在程序测试100次之后，得到结果表1- 3如所示。

表1- 3 程序运行时间

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 项目名称 | 编程语言 | 程序各过程运行时间/(ms) | | | | 测试环境 |
| 总时间 | 前处理  (pre-process) | 推理运算  (inference) | 后处理  (NMS) |
| flower\_clas | C++ | 302.41 | 255.19 | 47.22 | 0 | Shape：[1, 3, 224 ,224]  Net：ResNet50 CPU：AMD R7 5800H |
| C# | 306.75 | 255.15 | 51.53 | 0.06 |
| Python | 319.31 | 267.16 | 52.16 | 0 |
| flower\_clas | C++ | 273.06 | 215.93 | 57.12 | 001 | Shape：[1, 3, 224 ,224]  Net：ResNet50 CPU：Intel (R) i9-12900K |
| C# | 272.44 | 208.76 | 63.63 | 0.04 |
| Python | 294.49 | 223.65 | 70.81 | 0. |
| vehicle\_yolov3\_darknet | C++ | 1184.99 | 611.86 | 572.59 | 0.54 | Shape：[1, 3, 608 ,608]  Net：YOLOv3 CPU：AMD R7 5800H |
| C# | 1199.11 | 620.58 | 577.51 | 1.01 |
| Python | 1247.55 | 683.55 | 563.54 | 0.47 |
| vehicle\_yolov3\_darknet | C++ | 970.43 | 512.26 | 457.70 | 0.46 | Shape：[1, 3, 608 ,608]  Net：YOLOv3 CPU：Intel (R) i9-12900K |
| C# | 962.45 | 525.50 | 439.64 | 0.26 |
| Python | 1154.43 | 572.84 | 581.16 | 0 |

在本次检测中，我们通过C++、C#以及Python分别调用OpenVINOTM 进行模型的部署与推理，通过上述表格，一方面可以看出，C#通过调用C++的dll，实现模型的部署与推理，并没有太大的影响程序的运行速度；另一方面，C++与C#部署模型推理，在总时间上来看，运行速度是优于Python的。

测试模型运行时间所使用的测试代码，已同步到远程成代码托管仓库gitee与github中，具体在openvino\_run\_time文件夹下，使用人员可以根据自己的设备对C++、C#和Python三个平台进行测试。

## 项目总结

该项目通过C++调用OpenVINOTM，创建推理方法接口，并通过调用dll文件的方式，在C#中进行重新构建Core模型推理类，并测试了花卉分类模型以及车辆识别模型，在预测结果精度以及预测时间上，和C++相比，并没有较大的差异。

该项目所提供的方法，证实了C#平台调用OpenVINOTM的可行性，为后续在C#部署Al模型项目提供了有效途径。