OpenCV 入门教程

作者：于仕琪

shiqi.yu@szu.edu.cn

http://www.opencv.org.cn

2012 年 8 月

前　言

OpenCV 是一个广受欢迎的开源计算机视觉库，它提供了很多函数，实现了很多计算机视觉算法，算法从最基本的滤波到高级的物体检测皆有涵盖。很多初学者希望快速掌握 OpenCV 的使用方法，但往往会遇到各种各样的困难。其实仔细分析，造成这些困难的原因有两类：第一类是 C/C++编程基础不过关；第二类是不了解算法原理。解决这些困难无非提升编程能力，以及提升理论基础知识。提升编程能力需要多练习编程，提升理论知识需要系统学习《数字图像处理》、《计算机视觉》和《模式识别》等课程，所有这些都不能一蹴而就，需要耐下心来认真修炼。

同时我们也需要认识到 OpenCV 只是一个算法库，能为我们搭建计算机视觉应用提供“砖头”。我们并不需要完全精通了算法原理之后才去使用 OpenCV，只要了解了“砖头”的功能，就可以动手了。在实践中学习才是最高效的学习方式。本小册子希望为初学者提供引导，使初学者快速了解 OpenCV 的基本数据结构以及用法。

此外，如您发现有错误之处，欢迎来信指正。

于仕琪

深圳大学

插播广告：欢迎有能力、有激情以及对计算机视觉有兴趣的同学报考我的研究生。欲了解详情可以访问深圳大学招生网 http://zsb.szu.edu.cn/ 或者给我发 email。

目录

第1章 预备知识

OpenCV 是一个功能强大的计算机视觉库，要用好它，除了要具有相关的计算机视觉理论知识外，还需要具有一定的编程能力。本书作者通过对 OpenCV 中文论坛中的大量问题观察，发现有很大比例的问题是因为用户对 C/C++语言不熟练，导致出错，或出错后不知如何解决。如果对 C/C++语言不熟悉，那使用 OpenCV 时会满头雾水瞎摸索，费心费力。

在这一章中，将介绍一些编程的基本概念，让读者对编程的流程有一个基本了解。这样在出现错误时，可以快速确定错误的类型，并知道该如何解决。

1.1 编程的流程

一个编程的基本流程包括编辑、编译和连接三大步骤。其流程图如图 1.1 所示。

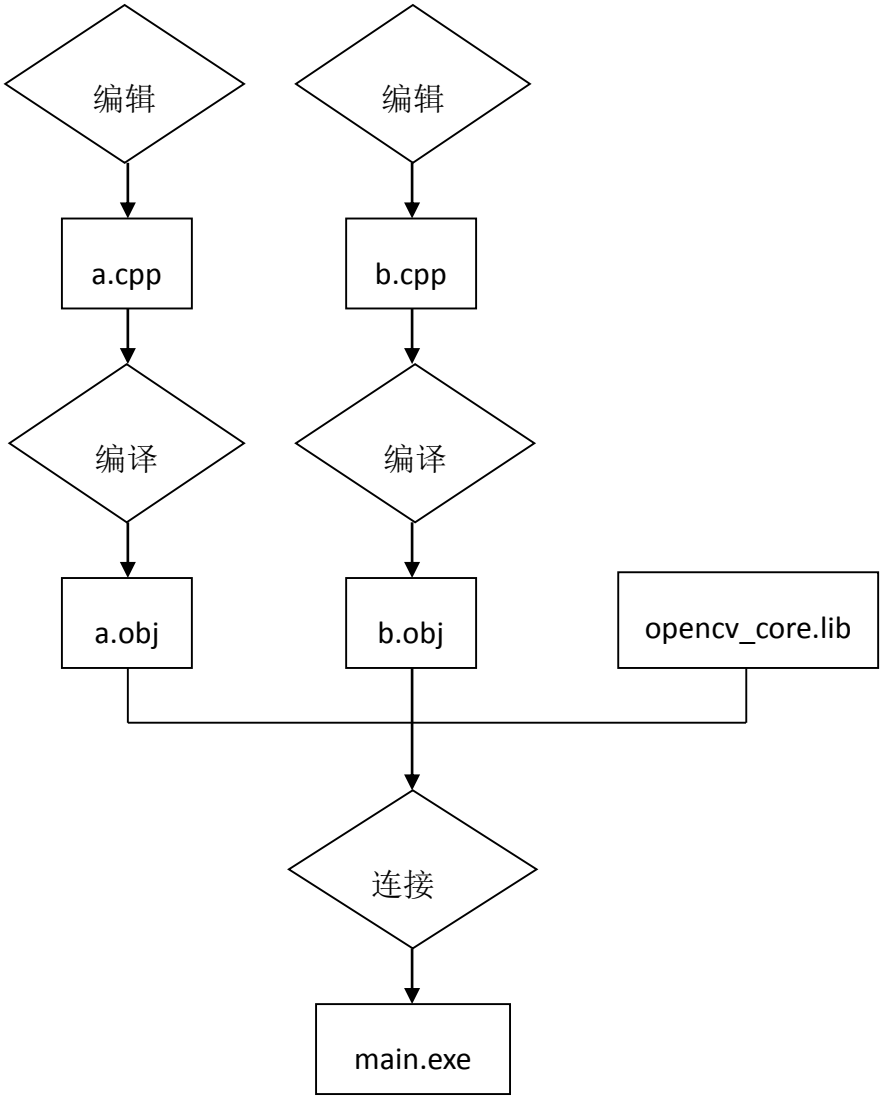

图 1.1 编程的基本流程

1.2 什么叫编辑

编辑（edit）代码即编写代码，是编程的第一步。你可以任意一个编辑器进行代码的编写。你可以使用 Windows 自带的“记事本”来编写代码，也可以使用 Notepad++，或者 Visual Studio 提供的编辑器。

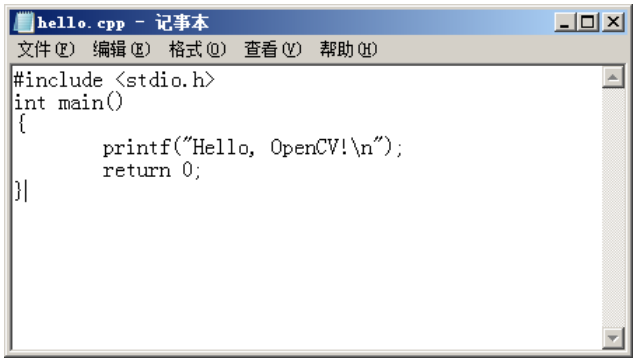

图 1.2 使用 Windows 自带的记事本编辑代码

虽然可以使用记事本软件编辑代码，但是记事本软件的功能非常有限。缺少常用的语法高亮，自动缩进等功能。所以可以使用其他功能更丰富的编辑器，如 Notepad++（图 1.3）等。

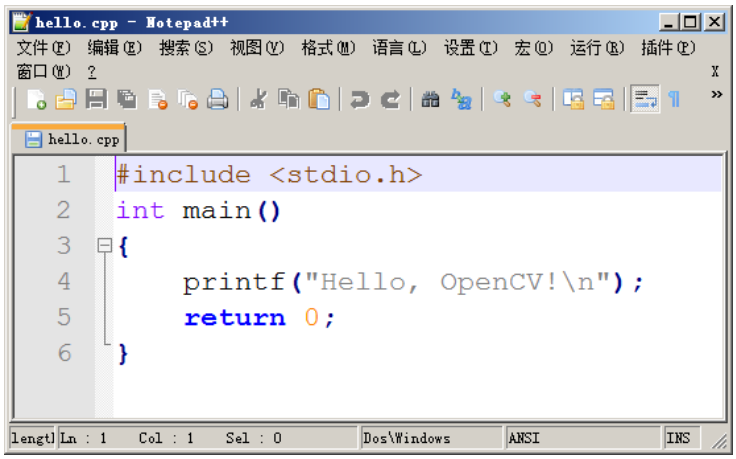

图 1.3 使用 Notepad++软件编辑代码

1.3 什么叫编译

编译（compile）是将用某种编程语言（如 C++语言）写成的源代码，转换成目标文件。目标文件包含着机器代码（可直接被计算机中央处理器执行）以及代码在运行时使用的数据。编译器（compiler）是实现这一目的的软件。编译器有很多，如在 Windows 下有微软公司的 cl.exe，在 Linux 下有 gcc 和 g++。在命令行

下使用 cl.exe 对 hello.cpp 源代码进行编译，如图 1.4 所示。编译后，将得到目标文件 hello.obj，如图 1.5 所示。

```
c:\source\chapter1>cl.exe /c hello.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 16.00.30319.01 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

hello.cpp
```

图 1.4 在命令行下使用 cl.exe 对 hello.cpp 进行编译

图 1.5 编译后，将新生成 hello.obj 目标文件

1.4 什么叫连接

连接（link）是将多个目标文件，以及库文件生成可执行的文件（或静态库、或动态库）的过程。连接器（linker）是实现这一目的的软件。常用的连接器有 Windows 下的 link.exe，Linux 下的 ld 等。

在 Windows 下可以使用 link.exe 将前面生成的 hello.obj 连接为可执行文件。在命令行下效果如所图 1.6 示。连接后，将生成可执行文件，如图 1.7 所示。

```
c:\source\chapter1>link.exe hello.obj
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.
```

图 1.6 在命令行下使用 link.exe 对 hello.obj 进行连接

图 1.7 连接后，将新生成 hello.exe 可执行文件

1.5 什么叫运行

运行（run）较容易理解，我们在 Windows 资源管理器里用鼠标双击 exe 可执行程序，可以使程序被载入 CPU 运行。我们也可以在命令行窗口中输入可执

行程序的文件名运行，如图 1.8 所示。

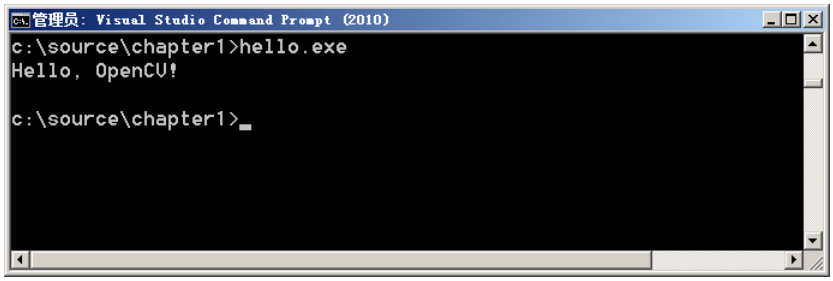

图 1.8 在命令行窗口中运行 hello.exe，可以看到程序打印到标准输出的结果。

1.6 Visual C++是什么

通过前面的介绍，可以看到一个编程的流程：编辑->编译->连接->运行。更具体来说，完成这个流程需要你：

1. 打开记事本软件，编辑代码，并保存；
2. 在命令行下运行编译器，对代码进行编译，生成目标文件；
3. 在命令行下运行连接器，将目标文件连接起来，生成可执行程序；
4. 在命令行下，或 Windows 资源管理器中运行程序，验证程序的正确性。

如果你的项目只有一个源代码文件，完成上面四个步骤尚可接受。但是如果你的项目包括几十个甚至几百个源文件，如无其他软件辅助，只用上面四个非常基本的步骤进行编程开发，会让人抓狂。

集成开发环境（Integrated Development Environment，简称 IDE）可以帮助你对项目进行管理。常用的 IDE 有微软公司的 Visual Studio，里面包含 Visual C++，Visual C#等，其他的还有 Eclipse、NetBeans、Delphi 等。因此我们平时所说的 VC 不是一种编程语言，也不是编译器，它只是一个 IDE。

IDE 一般包含编辑器。IDE 自带的编辑器一般都针对编程语言进行了定制，实现语法高亮、自动缩进、自动补全等方便的功能。IDE 还提供丰富的菜单和按钮工具，如图 1.9、图 1.10 和图 1.11 所示。

如果你点击 IDE 中的“生成（build）”按钮（图 1.11），或者点击菜单“生成（build）”中的菜单项“生成项目（build project）”，那么 IDE 会去调用编译器 cl.exe 和连接器 link.exe 来生成可执行程序。如果你在调试状态下，还会去调用调试器（debugger）。IDE 会提升程序开发的效率，特别是调试程序的效率。

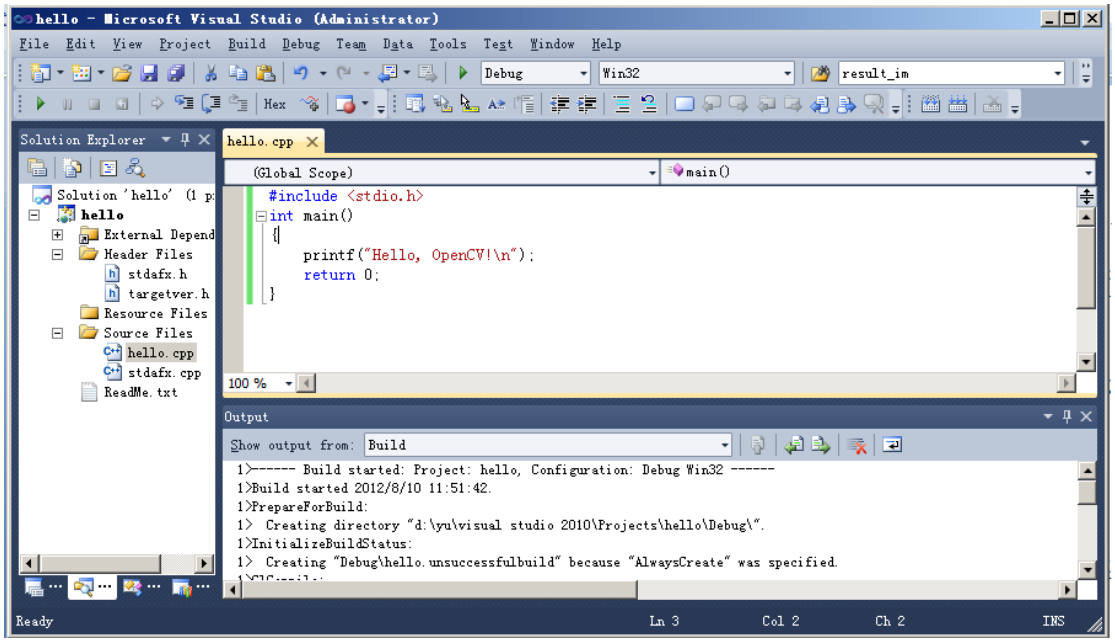

图 1.9 微软 Visual Stdio 集成开发环境

图 1.10 Visual Stdio 中的编辑按钮

图 1.11 Visual Stdio 中的生成程序按钮

1.7 头文件

在编程过程中，程序代码往往被拆成很多部分，每部分放在一个独立的源文件中，而不是将所有的代码放在一个源文件中。考虑一个简单的小例子：程序中有两个函数 main()和 foo()。main()函数位于 main.cpp，foo()函数位于 foo.cpp，main()函数中调用 foo()函数。在编译阶段，由于编译是对单个文件进行编译，所以编译 main.cpp 时，编译器不知道是否存在 foo()函数以及 foo()调用是否正确，因此需要头文件辅助。也就是说，在编译命令：

```
cl.exe /c main.cpp
```

运行时，编译器不知道 foo 的用法是否正确（因为 foo 在另一个文件 foo.cpp 中），只有借助头文件中的函数声明来判断。对 main.cpp 进行编译时，不会涉及 foo.cpp 文件，只会涉及 main.cpp 和 foo.h（因为 foo.h 被 include）文件。头文件的作用如图 1.1 所示。

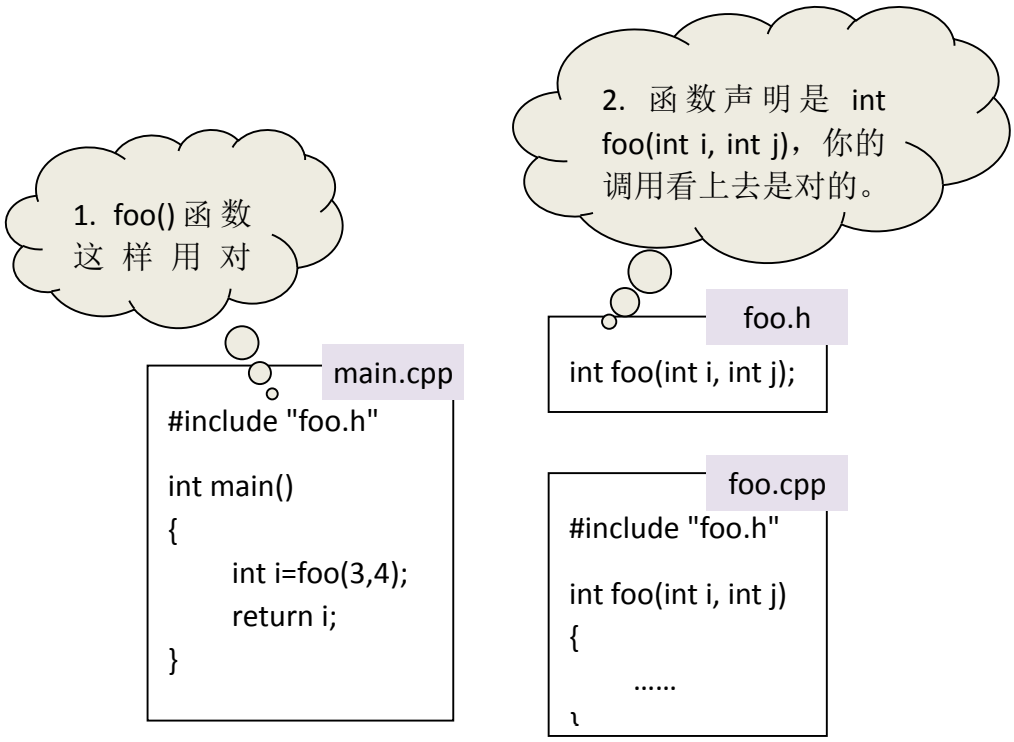

图 1.12 对 main.cpp 进行编译时，需要利用头文件中的 foo()函数声明来确认 main.cpp 中对 foo()的调用是正确的

1.8 库文件

库文件中包含一系列的子程序。例如在上一节的例子中，foo.cpp 源文件中实现了 foo()函数，我们假设 foo()函数是包含重要算法的函数，我们需要将 foo()函数提供给客户使用，但是不希望客户看到算法源代码。为了达到这一目的，我们可以将 foo.cpp 编译程库文件（图 1.13），库文件是二进制的，在库文件中是看不到原始的源代码的。库和可执行文件的区别是，库不是独立程序，他们是向其他程序提供服务的代码。

当然使用库文件的好处不仅仅是对源代码进行保密，使用库文件还可以减少重复编译的时间，增强程序的模块化。将库文件连接到程序中，有两种方式，一种是静态连接库，另一种是动态连接库。如果希望了解更多关于库文件的知识，请查阅相关资料，再次不详细分析它们之间的异同。

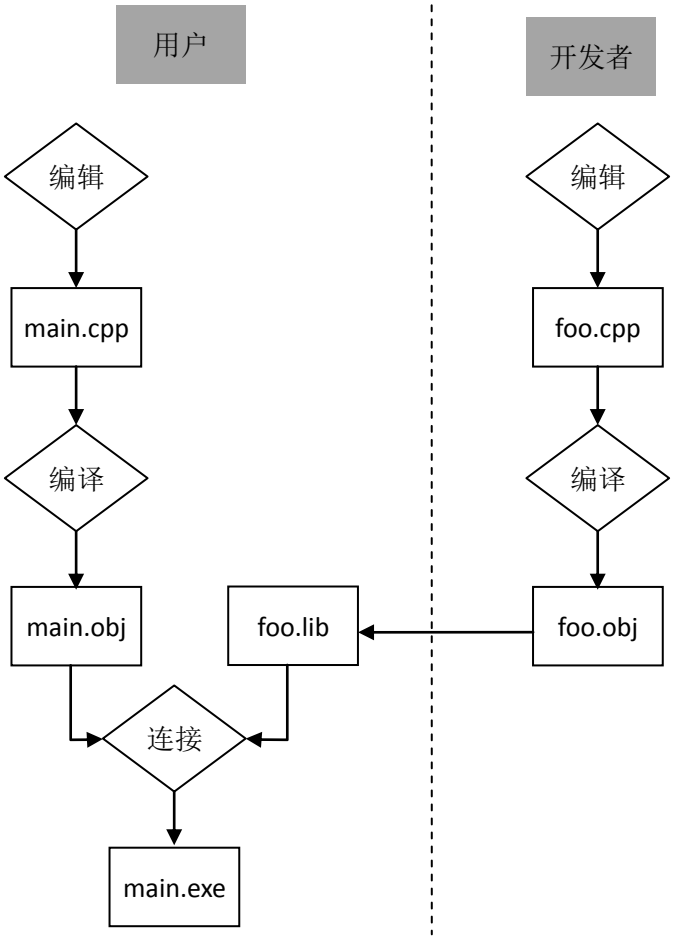

图 1.13 库是二进制的文件，里面包含一系列子程序（图有问题）

1.9 OpenCV 是什么

OpenCV 其实就是一堆 C 和 C++语言的源代码文件，这些源代码文件中实现了许多常用的计算机视觉算法。例如 C 接口函数 cvCanny()实现了 Canny 边缘提取算法。可以直接将这些源代码添加到我们自己的软件项目中，而不需要自己再去写代码实现 Canny 算法，也就是不需要重复“造轮子”。

由于 OpenCV 中源代码文件巨多，根据算法的功能，将这些源文件分到多个模块中：core、imgproc、highgui 等。将每个模块中的源文件编译成一个库文件（如 opencv_core.lib、opencv_imgproc.lib、opencv_highgui.lib 等），用户在使用时，仅将所需的库文件添加到自己的项目中，与自己的源文件一起连接成可执行程序则可。

1.10 什么是命令行参数

C/C++语言中的 main 函数，经常带有参数 argc，argv，如下：

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

或者

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

在上面代码中，argc 表示命令行输入参数的个数（以空白符分隔），argv 中存储了所有的命令行参数。假如你的程序是 hello.exe，如果在命令行运行该程序（如图 1.14。首先应该在命令行下用 cd 命令进入到 hello.exe 文件所在目录），运行命令为：

```
hello.exe Shiqi Yu
```

那么，argc 的值是 3，argv[0]是"hello.exe"，argv[1]是"Shiqi"，argv[2]是"Yu"。

图 1.14 使用命令行参数运行程序

下面的程序演示 argc 和 argv 的使用：

```
#include <stdio.h>
int main(int argc, char ** argv)
{
    int i;
    for (i=0; i < argc; i++)
        printf("Argument %d is %s.\n", i, argv[i]);
    return 0;
}
```

假如上述代码编译为 hello.exe，那么运行

```
hello.exe a b c d e
```

将得到

```
Argument 0 is hello.exe.
Argument 1 is a.
Argument 2 is b.
Argument 3 is c.
Argument 4 is d.
Argument 5 is e.
```

运行

```
hello.exe lena.jpg
```

将得到

```
Argument 0 is hello.exe.
Argument 1 is lena.jpg.
```

1.11 常见编译错误

在编程中，经常会出现各种错误。出现错误后，不要闭眼抱头作痛苦状。出现错误后，需要做的第一件事情是阅读出错信息。出错信息虽然看似凌乱，但是能够提供很多有价值的信息，帮你解决问题。

1.11.1找不到头文件

找不到头文件往往会提示如下错误：

```
hello.cpp(2): fatal error C1083: Cannot open include file:
'opencv2/opencv.hppp': No such file or directory
```

找不到头文件一般有两个原因：一个是头文件的文件名拼写错误；或者未将头文件所在路径添加到开发环境中。上例中的错误是文件名拼写错误，opencv2/opencv.hpp 被错误地拼写为 opencv2/opencv.hppp。如果文件名拼写正确，编译器还是找不到头文件，则需要将头文件所在路径添加到相应的变量中。如在 Visual Studio 2010 中，需要在项目属性（Project Property）对话框中设置头文件路径。具体位置在对话框“VC++ Directories”里面的“Include Directories”中，如图 1.15 所示。

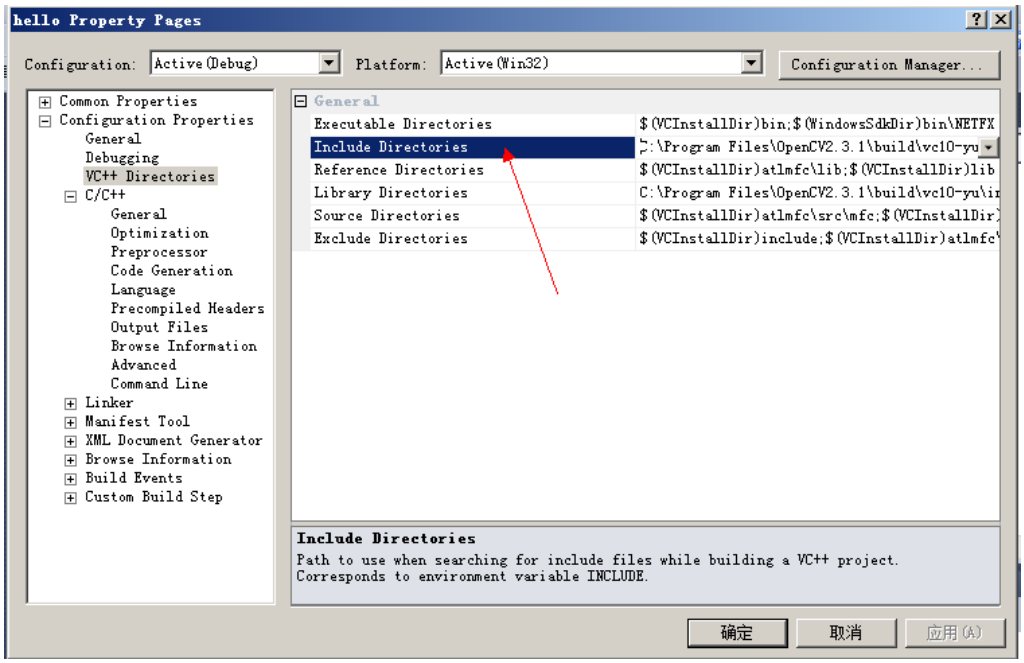

图 1.15 头文件所在路径设置

1.11.2拼写错误

在编程中，拼写错误也是一类常见错误。如图 1.16 所示代码中，将 imread 函数错误地拼成 imreadd，编译器会提示错误：

```
hello.cpp(9): error C3861: 'imreadd': identifier not found
```

这句错误提示的意思是说无法找到 imreadd 标识符，因此我们需要仔细检查 imreadd 找不到的原因。假如你真的有一个函数是 imreadd，但是找不到，可能的原因是声明 imreadd 的头文件未使用 include 语句包含到源文件中。

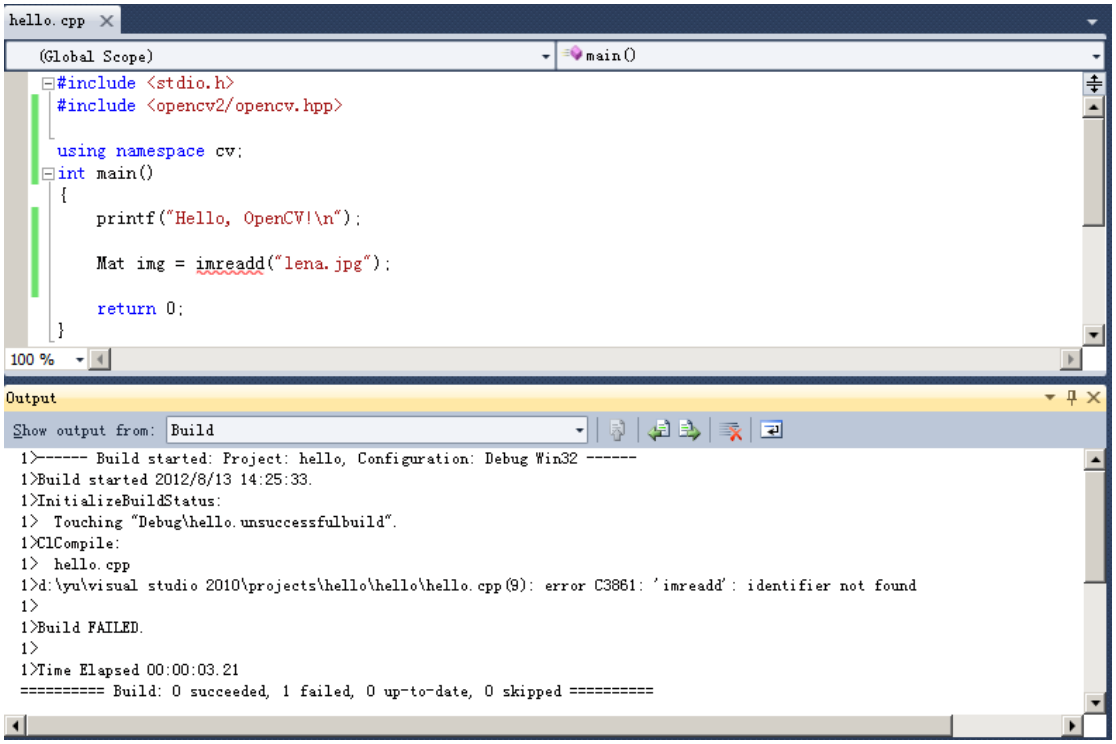

图 1.16 拼写错误，将 imread 拼成了 imreadd，会造成编译时错误。

如果源代码不符合语法规则，则会造成编译错误。编译错误往往是由于编写代码不仔细造成，比如拼写错误、漏了半个括号、漏了分号等。因此一旦遇到便宜错误，你需要按照错误提示，定位到出错的位置，仔细检查语法是否符合规范。

1.12 常见链接错误

如果你的代码符合语法规则，则会通过编译过程。编译完所有源代码之后，下一步是连接目标文件，以形成可执行文件。连接过程中最常见的错误如下（图 1.17）：

```
1>hello.obj : error LNK2019: unresolved external symbol "class cv::Mat __cdecl cv::imread(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &,int)" (?imread@cv@@YA?AVMat@1@ABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@H@Z) referenced in function _main
```

这个错误信息里最关键的词是“unresolved external symbol”，更具体的意思是在 main 函数中使用了 imread 函数，但是无法从外部找到 imread。imread 函数是 OpenCV 的函数，不是用户自己实现的函数。opencv.hpp 头文件告诉编译器有个 imread 函数可以用，编译通过；但是到了连接时，连接器却找不到 imread 的具体实现，故出错。

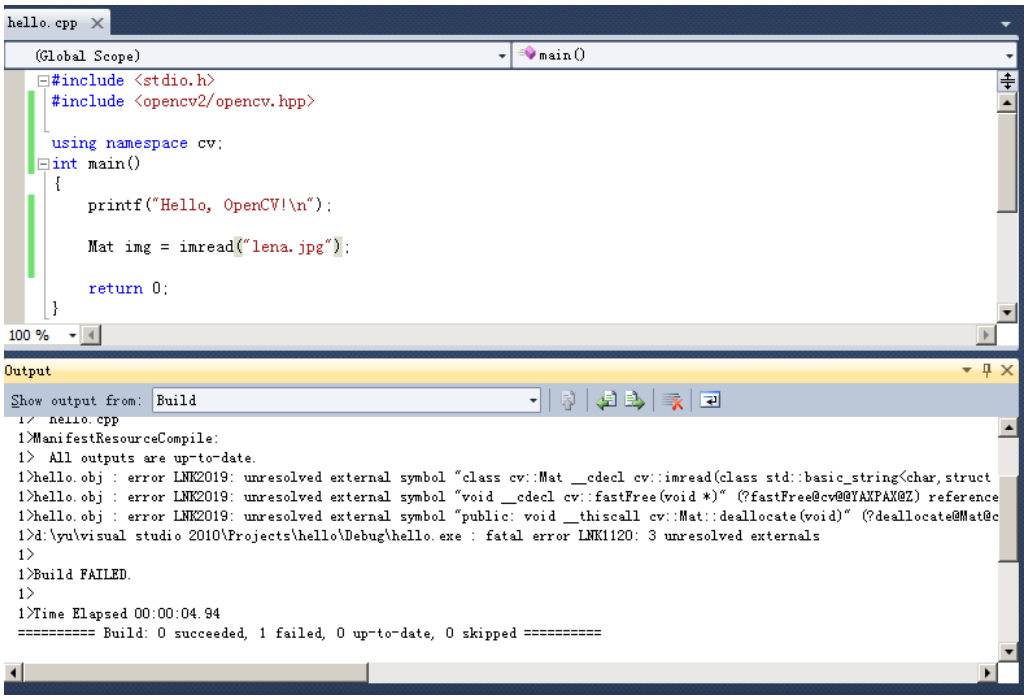

图 1.17 连接错误，无法找到 imread 等函数的实现

要解决这一问题，需要将依赖的库文件添加到项目设置中。具体位置在对话框“Linker - Input”里面的“Additional Dependencies”中，如图 1.18 所示。

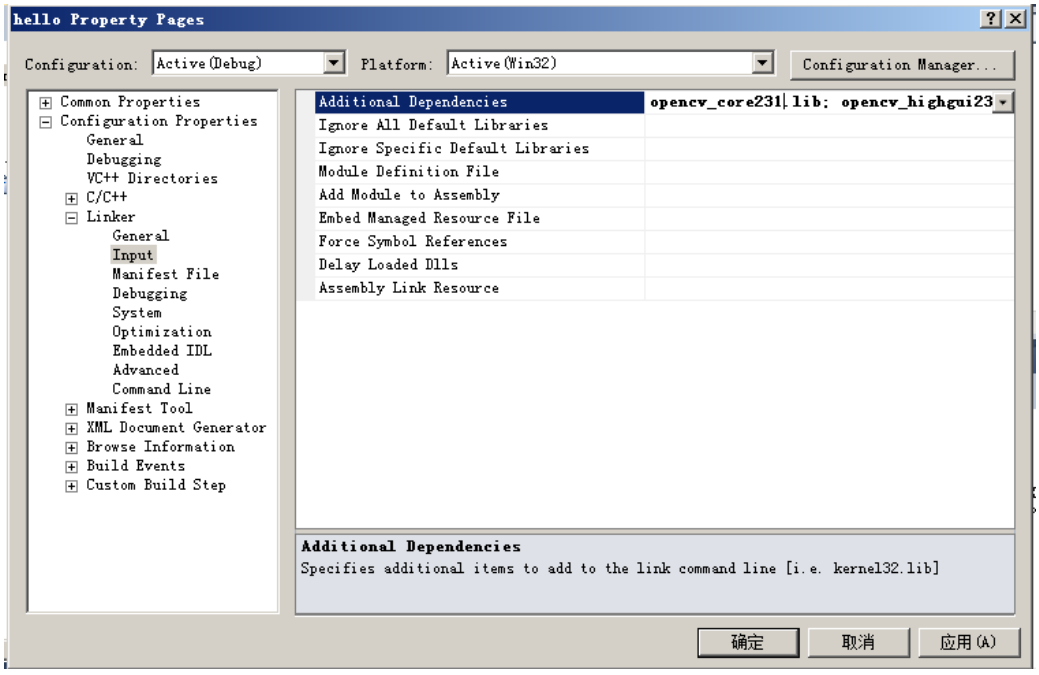

图 1.18 添加依赖的库文件

1.13 运行时错误

经过编译和连接过程，生成了可执行的文件（如 exe 文件）之后，在运行这个可执行文件所产生的错误是运行时错误。比较常见的运行时错误是内存错误。比如下面这段代码：

```
#include <stdio.h>
#include <opencv2/opencv.hpp>

using namespace cv;
int main()
{
    printf("Hello, OpenCV!\n");

    Mat img = imread("lena.jpg");
    Mat gray;
    cv::cvtColor(img, gray, CV_BGR2GRAY);

    return 0;
}
```

编译和连接过程无任何问题，但在运行时弹出如图 1.19 所示对话框，并在命令行窗口输出错误信息（图 1.20）。

图 1.19 运行时错误对话框

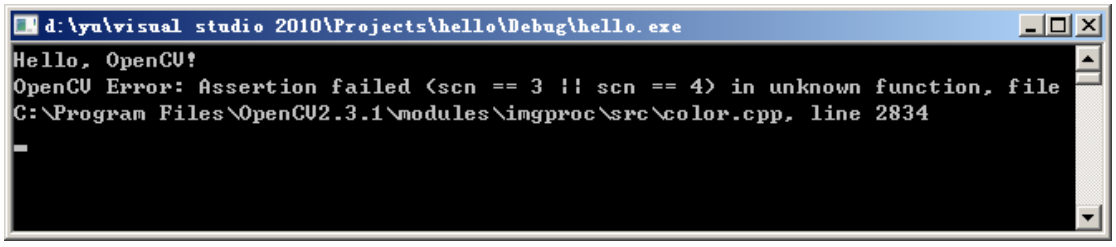

图 1.20 运行时错误的出错信息

错误信息中提示 color.cpp 文件的第 2834 行有错，错误原因是条件(scn==3||scn==4)不成立。很多 OpenCV 用户看到此错误信息一头雾水，不知如何下手解决。根据程序源代码的意思，是将三通道的 BGR 图像 img 转为单通道的图像 gray。但是程序说 img 既不是 3 通道，也不是 4 通道。而根据 imread 函数的文档，imread 将图像作为彩色图像读入，条件(scn==3||scn==4)肯定成立。

这个程序的问题出现在当前目录下无 lena.jpg 文件，这样程序无法读到图像，造成 cvtColor 函数出错。因此对于读入图像时，需要检查图像读入是否成功，以免造成运行时错误。

在程序编写中，对于数组和指针等，要特别地小心。因为对于空指针以及数组越界等问题，编译器无法在编译时给出错误提示。这类错误一旦在运行时发生，排除起来非常困难。

第2章 OpenCV 介绍

OpenCV 的全称是 Open Source Computer Vision Library，是一个开放源代码的计算机视觉库。OpenCV 是最初由英特尔公司发起并开发，以 BSD 许可证授权发行，可以在商业和研究领域中免费使用，现在美国 Willow Garage 为 OpenCV 提供主要的支持。OpenCV 可用于开发实时的图像处理、计算机视觉以及模式识别程序，目前在工业界以及科研领域广泛采用。

2.1 OpenCV 的来源

OpenCV 诞生于 Intel。Intel 最初希望提供一个计算机视觉库，使之能充分发掘 CPU 的计算能力，当然更希望以此促进 Intel 的产品的销售。OpenCV 最初的开发工作是由 Intel 在俄罗斯的团队实现。这里面有两个关键人物，一个是 Intel 性能团队（Intel's Performance Library Team）的李信弘（Shinn Lee）先生，他是团队的经理，负责 IPP 等库，给予 OpenCV 很大的支持。另一个关键人物是 Vadim Pisarevsky，Vadim 在 Intel 负责 OpenCV 的项目管理、代码集成、代码优化等工作。在后期 Intel 支持渐少的时候，是 Vadim Pisarevsky 一直在维护着 OpenCV。2007 年 6 月，受本书作者之邀，李信弘和 Vadim Pisarevsky 作为嘉宾参加了在北京举行的“开放源代码计算机视觉库(OpenCV)研讨会”[1]，并做了非常有价值的报告。

在 2008 年，一家美国公司，Willow Garage[2]，开始大力支持 OpenCV，Vadim Pisarevsky 和 Gary Bradski 都加入了 Willow Garage。Gary Bradski 也是 OpenCV 开发者中的元老级人物，他曾出版《Leaning OpenCV》一书，广受欢迎。

Willow Garage 是一家机器人公司，致力于为个人机器人开发开放的硬件平台和软件。现在已经开发了 PR2 机器人，并支持 ROS、OpenCV、PCL 等软件。ROS（Robot Operating System）是用于机器人的操作系统，是一个开放源代码的软件，OpenCV 作为 ROS 的视觉模块嵌入。

自从获得 Willow Garage 支持后，OpenCV 的更新速度明显加快。大量的新特性被被加入 OpenCV 中，很多算法都是最近一两年的新的科研成果。OpenCV 正日益成为算法研究和产品开发不可缺少的工具。

2.2 OpenCV 的协议

OpenCV 采用 BSD 协议，这是一个非常宽松的协议。简而言之，用户可以修

[1] 研讨会网址：http://www.opencv.org.cn/index.php/OpenCV_Symposium

[2] Willow Garage 公司网站：http://www.willowgarage.com

改 OpenCV 的源代码，可以将 OpenCV 嵌入到自己的软件中，可以将包含 OpenCV 的软件销售，可以用于商业产品，也可以用于科研领域。BSD 协议并不具有“传染性”，如果你的软件中使用了 OpenCV，你不需要公开代码。你可以对 OpenCV 做任何操作，协议对用户的唯一约束是要在软件的文档或者说明中注明使用了 OpenCV，并附上 OpenCV 的协议。

在这个宽松协议下，企业可以在 OpenCV 基础之上进行产品开发，而不需要担心版权问题（当然你要注明使用了 OpenCV，并附上 OpenCV 的协议）。科研领域的研究者，可以使用 OpenCV 快速地实现系统原型。因此可以这样说，OpenCV 的协议保证了计算机视觉技术快速的传播，让更多的人从 OpenCV 受益。

第3章 图像的基本操作

3.1 图像的表示

在正式介绍之前，先简单介绍一下数字图像的基本概念。如图 3.1 中所示的图像，我们看到的是 Lena 的头像，但是计算机看来，这副图像只是一堆亮度各异的点。一副尺寸为 M × N 的图像可以用一个 M × N 的矩阵来表示，矩阵元素的值表示这个位置上的像素的亮度，一般来说像素值越大表示该点越亮。如图 3.1 中白色圆圈内的区域，进行放大并仔细查看，将会如图 3.2 所示。

图 3.1 Lena 的照片

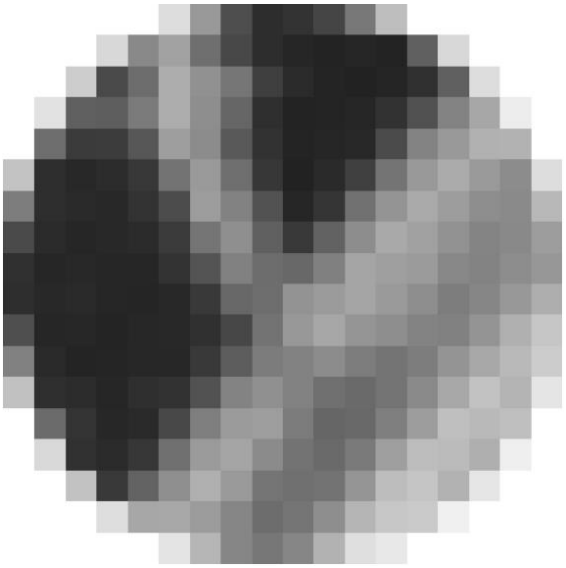

图 3.2 图 3.1 中圆圈处的放大效果

一般来说，灰度图用 2 维矩阵表示，彩色（多通道）图像用 3 维矩阵（M × N × 3）表示。对于图像显示来说，目前大部分设备都是用无符号 8 位整数（类型为 CV_8U）表示像素亮度。

图像数据在计算机内存中的存储顺序为以图像最左上点（也可能是最左下点）开始，存储如表 3-1 所示。

表 3-1 灰度图像的存储示意图

$I_{0\,0}$	$I_{0\,1}$	…	$I_{0\,N-1}$
$I_{1\,0}$	$I_{1\,1}$	…	$I_{1\,N-1}$
…	…	…	…
$I_{M-1\,0}$	$I_{M-1\,1}$	…	$I_{M-1\,N-1}$

I_{ij} 表示第 i 行 j 列的像素值。如果是多通道图像，比如 RGB 图像，则每个像素用三个字节表示。在 OpenCV 中，RGB 图像的通道顺序为 BGR ，存储如表 3-2 所示。

表 3-2 彩色 RGB 图像的存储示意图

B_{00}	G_{00}	R_{00}	B_{01}	G_{01}	R_{01}	…
B_{10}	G_{10}	R_{10}	B_{11}	G_{11}	R_{11}	…

…	…	…	…	…	…	…

3.2 Mat 类

早期的 OpenCV 中，使用 IplImage 和 CvMat 数据结构来表示图像。IplImage 和 CvMat 都是 C 语言的结构。使用这两个结构的问题是内存需要手动管理，开发者必须清楚的知道何时需要申请内存，何时需要释放内存。这个开发者带来了一定的负担，开发者应该将更多精力用于算法设计，因此在新版本的 OpenCV 中引入了 Mat 类。

新加入的 Mat 类能够自动管理内存。使用 Mat 类，你不再需要花费大量精力在内存管理上。而且你的代码会变得很简洁，代码行数会变少。但 C++接口唯一的不足是当前一些嵌入式开发系统可能只支持 C 语言，如果你的开发平台支持 C++，完全没有必要再用 IplImage 和 CvMat。在新版本的 OpenCV 中，开发者依然可以使用 IplImage 和 CvMat，但是一些新增加的函数只提供了 Mat 接口。本书中的例程也都将采用新的 Mat 类，不再介绍 IplImage 和 CvMat。

Mat 类的定义如下所示，关键的属性如下方代码所示：

```
class CV_EXPORTS Mat
{
public:
    //一系列函数
    ...
    /* flag 参数中包含许多关于矩阵的信息，如:
        -Mat 的标识
        -数据是否连续
        -深度
        -通道数目
     */
    int flags;
    //矩阵的维数，取值应该大于或等于 2
    int dims;
    //矩阵的行数和列数，如果矩阵超过 2 维，这两个变量的值都为-1
    int rows, cols;
    //指向数据的指针
    uchar* data;

    //指向引用计数的指针
    //如果数据是由用户分配的，则为 NULL
    int* refcount;
```

```
    //其他成员变量和成员函数
    ...
};
```

3.3 创建 Mat 对象

Mat 是一个非常优秀的图像类，它同时也是一个通用的矩阵类，可以用来创建和操作多维矩阵。有多种方法创建一个 Mat 对象。

3.3.1 构造函数方法

Mat 类提供了一系列构造函数，可以方便的根据需要创建 Mat 对象。下面是一个使用构造函数创建对象的例子。

```
Mat M(3,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl;
```

第一行代码创建一个行数（高度）为 3，列数（宽度）为 2 的图像，图像元素是 8 位无符号整数类型，且有三个通道。图像的所有像素值被初始化为(0, 0, 255)。由于 OpenCV 中默认的颜色顺序为 BGR，因此这是一个全红色的图像。

第二行代码是输出 Mat 类的实例 M 的所有像素值。Mat 重定义了<<操作符，使用这个操作符，可以方便地输出所有像素值，而不需要使用 for 循环逐个像素输出。

该段代码的输出如图 3.3 所示。

```
M =
 [0,  0,  255,  0,  0,  255;
  0,  0,  255,  0,  0,  255;
  0,  0,  255,  0,  0,  255]
```

图 3.3 例程输出内容

常用的构造函数有：

- Mat::Mat()
  无参数构造方法；
- Mat::Mat(int rows, int cols, int type)
  创建行数为 rows，列数为 col，类型为 type 的图像；
- Mat::Mat(Size size, int type)
  创建大小为 size，类型为 type 的图像；
- Mat::Mat(int rows, int cols, int type, const Scalar& s)

创建行数为 rows，列数为 col，类型为 type 的图像，并将所有元素初始化为值 s；

- Mat::Mat(Size size, int type, const Scalar& s)
  创建大小为 size，类型为 type 的图像，并将所有元素初始化为值 s；
- Mat::Mat(const Mat& m)
  将 m 赋值给新创建的对象，此处不会对图像数据进行复制，m 和新对象共用图像数据；
- Mat::Mat(int rows, int cols, int type, void* data, size_t step=AUTO_STEP)
  创建行数为 rows，列数为 col，类型为 type 的图像，此构造函数不创建图像数据所需内存，而是直接使用 data 所指内存，图像的行步长由 step 指定。
- Mat::Mat(Size size, int type, void* data, size_t step=AUTO_STEP)
  创建大小为 size，类型为 type 的图像，此构造函数不创建图像数据所需内存，而是直接使用 data 所指内存，图像的行步长由 step 指定。
- Mat::Mat(const Mat& m, const Range& rowRange, const Range& colRange)
  创建的新图像为 m 的一部分，具体的范围由 rowRange 和 colRange 指定，此构造函数也不进行图像数据的复制操作，新图像与 m 共用图像数据；
- Mat::Mat(const Mat& m, const Rect& roi)
  创建的新图像为 m 的一部分，具体的范围 roi 指定，此构造函数也不进行图像数据的复制操作，新图像与 m 共用图像数据。

这些构造函数中，很多都涉及到类型 type。type 可以是 CV_8UC1，CV_16SC1，…，CV_64FC4 等。里面的 8U 表示 8 位无符号整数，16S 表示 16 位有符号整数，64F 表示 64 位浮点数（即 double 类型）；C 后面的数表示通道数，例如 C1 表示一个通道的图像，C4 表示 4 个通道的图像，以此类推。

如果你需要更多的通道数，需要用宏 CV_8UC(n)，例如：

```
Mat M(3,2, CV_8UC(5));//创建行数为 3，列数为 2，通道数为 5 的图像
```

3.3.2 create()函数创建对象

除了在构造函数中可以创建图像，也可以使用 Mat 类的 create()函数创建图像。如果 create()函数指定的参数与图像之前的参数相同，则不进行实质的内存申请操作；如果参数不同，则减少原始数据内存的索引，并重新申请内存。使用方法如下面例程所示：

```
Mat M(2,2, CV_8UC3);//构造函数创建图像
M.create(3,2, CV_8UC2);//释放内存重新创建图像
```

需要注意的时，使用 create()函数无法设置图像像素的初始值。

3.3.3 Matlab 风格的创建对象方法

OpenCV 2 中提供了 Matlab 风格的函数，如 zeros()，ones()和 eyes()。这种方法使得代码非常简洁，使用起来也非常方便。使用这些函数需要指定图像的大小和类型，使用方法如下：

```
Mat Z = Mat::zeros(2,3, CV_8UC1);
cout << "Z = " << endl << " " << Z << endl;

Mat O = Mat::ones(2, 3, CV_32F);
cout << "O = " << endl << " " << O << endl;

Mat E = Mat::eye(2, 3, CV_64F);
cout << "E = " << endl << " " << E << endl;
```

该代码中，有些 type 参数如 CV_32F 未注明通道数目，这种情况下它表示单通道。上面代码的输出结果如图 3.4 所示。

```
Z =
 [0, 0, 0;
  0, 0, 0]
O =
 [1, 1, 1;
  1, 1, 1]
E =
 [1, 0, 0;
  0, 1, 0]
```

图 3.4 Matlab 风格的函数例程的输出结果

3.4 矩阵的基本元素表达

对于单通道图像，其元素类型一般为 8U（即 8 位无符号整数），当然也可以是 16S、32F 等；这些类型可以直接用 uchar、short、float 等 C/C++语言中的基本数据类型表达。

如果多通道图像，如 RGB 彩色图像，需要用三个通道来表示。在这种情况下，如果依然将图像视作一个二维矩阵，那么矩阵的元素不再是基本的数据类型。

OpenCV 中有模板类 Vec，可以表示一个向量。OpenCV 中使用 Vec 类预定义了一些小向量，可以将之用于矩阵元素的表达。

```
typedef Vec<uchar, 2> Vec2b;
typedef Vec<uchar, 3> Vec3b;
typedef Vec<uchar, 4> Vec4b;

typedef Vec<short, 2> Vec2s;
typedef Vec<short, 3> Vec3s;
typedef Vec<short, 4> Vec4s;

typedef Vec<int, 2> Vec2i;
typedef Vec<int, 3> Vec3i;
typedef Vec<int, 4> Vec4i;

typedef Vec<float, 2> Vec2f;
typedef Vec<float, 3> Vec3f;
typedef Vec<float, 4> Vec4f;
typedef Vec<float, 6> Vec6f;

typedef Vec<double, 2> Vec2d;
typedef Vec<double, 3> Vec3d;
typedef Vec<double, 4> Vec4d;
typedef Vec<double, 6> Vec6d;
```

例如 8U 类型的 RGB 彩色图像可以使用 Vec3b，3 通道 float 类型的矩阵可以使用 Vec3f。

对于 Vec 对象，可以使用[]符号如操作数组般读写其元素，如：

```
Vec3b color; //用 color 变量描述一种 RGB 颜色
color[0]=255; //B 分量
color[1]=0; //G 分量
color[2]=0; //R 分量
```

3.5 像素值的读写

很多时候，我们需要读取某个像素值，或者设置某个像素值；在更多的时候，我们需要对整个图像里的所有像素进行遍历。OpenCV 提供了多种方法来实现图像的遍历。

3.5.1 at()函数

函数 at()来实现读去矩阵中的某个像素，或者对某个像素进行赋值操作。下面两行代码演示了 at()函数的使用方法。

```
uchar value = grayim.at<uchar>(i,j);//读出第 i 行第 j 列像素值
grayim.at<uchar>(i,j)=128; //将第 i 行第 j 列像素值设置为 128
```

如果要对图像进行遍历，可以参考下面的例程。这个例程创建了两个图像，分别是单通道的 grayim 以及 3 个通道的 colorim，然后对两个图像的所有像素值进行赋值，最后现实结果。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char* argv[])
{
    Mat grayim(600, 800, CV_8UC1);
    Mat colorim(600, 800, CV_8UC3);

    //遍历所有像素，并设置像素值
    for( int i = 0; i < grayim.rows; ++i)
        for( int j = 0; j < grayim.cols; ++j )
            grayim.at<uchar>(i,j) = (i+j)%255;

    //遍历所有像素，并设置像素值
    for( int i = 0; i < colorim.rows; ++i)
        for( int j = 0; j < colorim.cols; ++j )
        {
            Vec3b pixel;
            pixel[0] = i%255; //Blue
            pixel[1] = j%255; //Green
            pixel[2] = 0;     //Red
            colorim.at<Vec3b>(i,j) = pixel;
        }

    //显示结果
    imshow("grayim", grayim);
    imshow("colorim", colorim);
```

```
    waitKey(0);

    return 0;
}
```

需要注意的是，如果要遍历图像，并不推荐使用 at()函数。使用这个函数的优点是代码的可读性高，但是效率并不是很高。

这段代码的运行结果如图 3.5 所示。

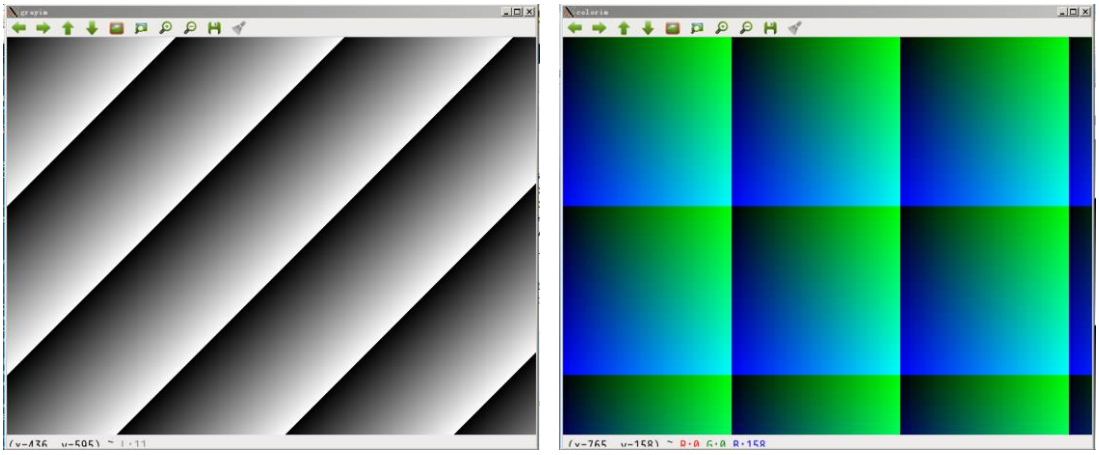

图 3.5 使用 at()函数遍历图像的例程的输出结果

3.5.2 使用迭代器

如果你熟悉 C++的 STL 库，那一定了解迭代器（iterator）的使用。迭代器可以方便地遍历所有元素。Mat 也增加了迭代器的支持，以便于矩阵元素的遍历。下面的例程功能跟上一节的例程类似，但是由于使用了迭代器，而不是使用行数和列数来遍历，所以这儿没有了 i 和 j 变量，图像的像素值设置为一个随机数。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char* argv[])
{
    Mat grayim(600, 800, CV_8UC1);
    Mat colorim(600, 800, CV_8UC3);

    //遍历所有像素，并设置像素值
    MatIterator_<uchar> grayit, grayend;
```

```
        for(    grayit    =    grayim.begin<uchar>(),    grayend    =
grayim.end<uchar>(); grayit != grayend; ++grayit)
            *grayit = rand()%255;

        //遍历所有像素，并设置像素值
        MatIterator_<Vec3b> colorit, colorend;
        for(    colorit    =    colorim.begin<Vec3b>(),    colorend    =
colorim.end<Vec3b>(); colorit != colorend; ++colorit)
        {
            (*colorit)[0] = rand()%255; //Blue
            (*colorit)[1] = rand()%255; //Green
            (*colorit)[2] = rand()%255; //Red
        }

        //显示结果
        imshow("grayim", grayim);
        imshow("colorim", colorim);
        waitKey(0);

        return 0;
    }
```

例程的输出结果如图 3.6 所示。

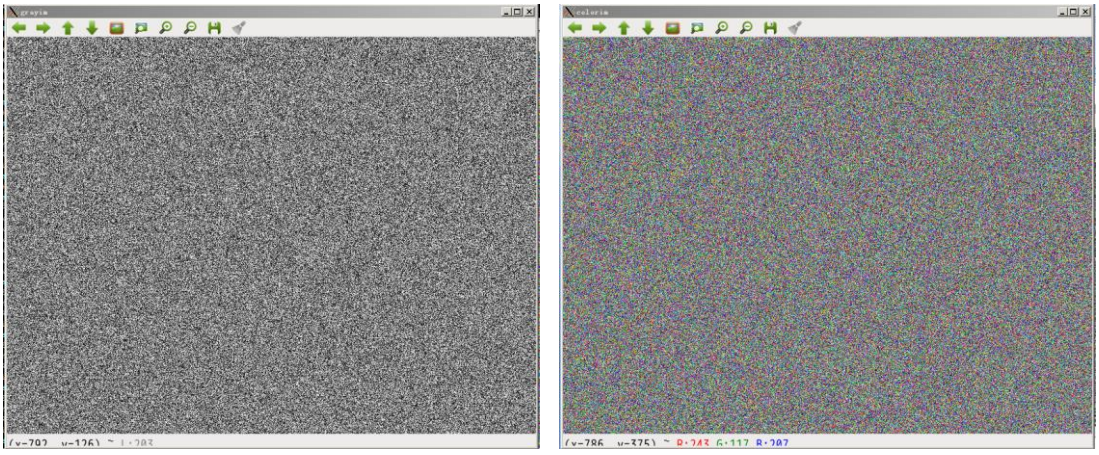

图 3.6 使用迭代器遍历图像的例程的输出结果

3.5.3 通过数据指针

使用 IplImage 结构的时候，我们会经常使用数据指针来直接操作像素。通过指针操作来访问像素是非常高效的，但是你务必十分地小心。C/C++中的指针操作是不进行类型以及越界检查的，如果指针访问出错，程序运行时有时候可能看

上去一切正常，有时候却突然弹出“段错误”（segment fault）。

当程序规模较大，且逻辑复杂时，查找指针错误十分困难。对于不熟悉指针的编程者来说，指针就如同噩梦。如果你对指针使用没有自信，则不建议直接通过指针操作来访问像素。虽然 at()函数和迭代器也不能保证对像素访问进行充分的检查，但是总是比指针操作要可靠一些。

如果你非常注重程序的运行速度，那么遍历像素时，建议使用指针。下面的例程演示如何使用指针来遍历图像中的所有像素。此例程实现的操作跟第 3.5.1 节中的例程完全相同。例程代码如下：

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char* argv[])
{
    Mat grayim(600, 800, CV_8UC1);
    Mat colorim(600, 800, CV_8UC3);

    //遍历所有像素，并设置像素值
    for( int i = 0; i < grayim.rows; ++i)
    {
        //获取第 i 行首像素指针
        uchar * p = grayim.ptr<uchar>(i);
        //对第 i 行的每个像素(byte)操作
        for( int j = 0; j < grayim.cols; ++j )
            p[j] = (i+j)%255;
    }

    //遍历所有像素，并设置像素值
    for( int i = 0; i < colorim.rows; ++i)
    {
        //获取第 i 行首像素指针
        Vec3b * p = colorim.ptr<Vec3b>(i);
        for( int j = 0; j < colorim.cols; ++j )
        {
            p[j][0] = i%255; //Blue
            p[j][1] = j%255; //Green
            p[j][2] = 0;     //Red
        }
```

```
    }

    //显示结果
    imshow("grayim", grayim);
    imshow("colorim", colorim);
    waitKey(0);

    return 0;
}
```

例程的输出结果如图 3.7 所示。

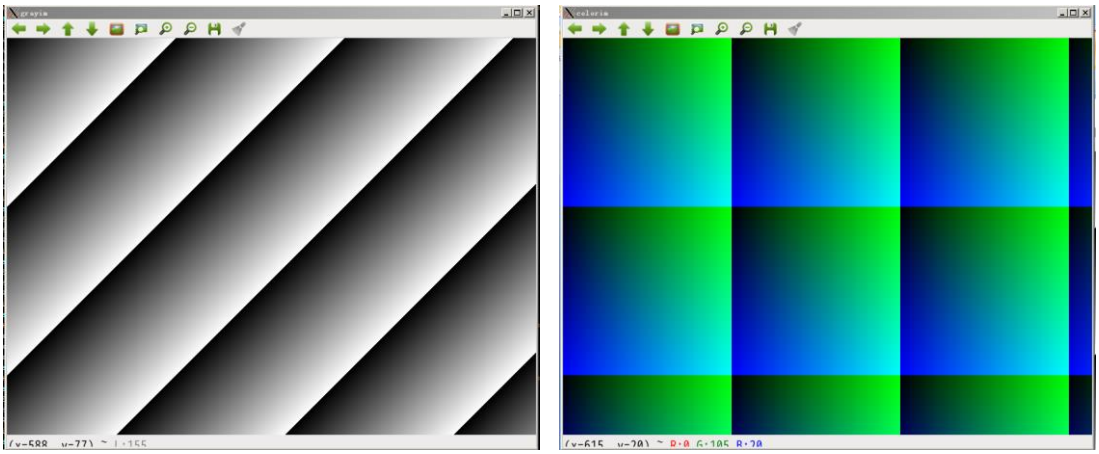

图 3.7 使用指针遍历图像的例程的输出结果

3.6 选取图像局部区域

Mat 类提供了多种方便的方法来选择图像的局部区域。使用这些方法时需要注意，这些方法并不进行内存的复制操作。如果将局部区域赋值给新的 Mat 对象，新对象与原始对象共用相同的数据区域，不新申请内存，因此这些方法的执行速度都比较快。

3.6.1 单行或单列选择

提取矩阵的一行或者一列可以使用函数 row()或 col()。函数的声明如下：

```
Mat Mat::row(int i) const
Mat Mat::col(int j) const
```

参数 i 和 j 分别是行标和列标。例如取出 A 矩阵的第 i 行可以使用如下代码：

```
Mat line = A.row(i);
```

例如取出 A 矩阵的第 i 行，将这一行的所有元素都乘以 2，然后赋值给第 j

行，可以这样写：

```
A.row(j) = A.row(i)*2;
```

3.6.2 用 Range 选择多行或多列

Range 是 OpenCV 中新增的类，该类有两个关键变量 star 和 end。Range 对象可以用来表示矩阵的多个连续的行或者多个连续的列。其表示的范围为从 start 到 end，包含 start，但不包含 end。Range 类的定义如下：

```
class Range
{
public:
    ...
    int start, end;
};
```

Range 类还提供了一个静态方法 all()，这个方法的作用如同 Matlab 中的“:”，表示所有的行或者所有的列。

```
//创建一个单位阵
Mat A = Mat::eye(10, 10, CV_32S);
//提取第 1 到 3 列（不包括 3）
Mat B = A(Range::all(), Range(1, 3));
//提取 B 的第 5 至 9 行（不包括 9）
//其实等价于 C = A(Range(5, 9), Range(1, 3))
Mat C = B(Range(5, 9), Range::all());
```

3.6.3 感兴趣区域

从图像中提取感兴趣区域（Region of interest）有两种方法，一种是使用构造函数，如下例所示：

```
//创建宽度为 320，高度为 240 的 3 通道图像
Mat img(Size(320,240),CV_8UC3);
//roi 是表示 img 中 Rect(10,10,100,100)区域的对象
Mat roi(img, Rect(10,10,100,100));
```

除了使用构造函数，还可以使用括号运算符，如下：

```
Mat roi2 = img(Rect(10,10,100,100));
```

当然也可以使用 Range 对象来定义感兴趣区域，如下：

```
//使用括号运算符
Mat roi3 = img(Range(10,100),Range(10,100));
```

```
//使用构造函数
Mat roi4(img, Range(10,100),Range(10,100));
```

3.6.4 取对角线元素

矩阵的对角线元素可以使用 Mat 类的 diag()函数获取，该函数的定义如下：

```
Mat Mat::diag(int d) const
```

参数 d=0 时，表示取主对角线；当参数 d>0 是，表示取主对角线下方的次对角线，如 d=1 时，表示取主对角线下方，且紧贴主多角线的元素；当参数 d<0 时，表示取主对角线上方的次对角线。

如同 row()和 col()函数，diag()函数也不进行内存复制操作，其复杂度也是 O(1)。

3.7 Mat 表达式

利用 C++中的运算符重载，OpenCV 2 中引入了 Mat 运算表达式。这一新特点使得使用 C++进行编程时，就如同写 Matlab 脚本，代码变得简洁易懂，也便于维护。

如果矩阵 A 和 B 大小相同，则可以使用如下表达式：

```
C = A + B + 1;
```

其执行结果是 A 和 B 的对应元素相加，然后再加 1，并将生成的矩阵赋给 C 变量。

下面给出 Mat 表达式所支持的运算。下面的列表中使用 A 和 B 表示 Mat 类型的对象，使用 s 表示 Scalar 对象，alpha 表示 double 值。

- 加法，减法，取负：A+B，A-B，A+s，A-s，s+A，s-A，-A
- 缩放取值范围：A*alpha
- 矩阵对应元素的乘法和除法： A.mul(B)，A/B，alpha/A
- 矩阵乘法：A*B （注意此处是矩阵乘法，而不是矩阵对应元素相乘）
- 矩阵转置：A.t()
- 矩阵求逆和求伪逆：A.inv()
- 矩阵比较运算：A cmpop B，A cmpop alpha，alpha cmpop A。此处 cmpop 可以是>，>=，==，!=，<=，<。如果条件成立，则结果矩阵（8U 类型矩阵）的对应元素被置为 255；否则置 0。
- 矩阵位逻辑运算：A logicop B，A logicop s，s logicop A，~A，此处 logicop 可以是&，|和^。

- 矩阵对应元素的最大值和最小值：min(A, B)，min(A, alpha)，max(A, B)，max(A, alpha)。
- 矩阵中元素的绝对值：abs(A)
- 叉积和点积：A.cross(B)，A.dot(B)

下面例程展示了 Mat 表达式的使用方法，例程的输出结果如图 3.8 所示。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

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


    Mat A = Mat::eye(4,4,CV_32SC1);

    Mat B = A * 3 + 1;

    Mat C = B.diag(0) + B.col(1);

    cout << "A = " << A << endl << endl;
    cout << "B = " << B << endl << endl;
    cout << "C = " << C << endl << endl;
    cout << "C .* diag(B) = " << C.dot(B.diag(0)) << endl;


    return 0;
}
```

```
A = [1, 0, 0, 0;
  0, 1, 0, 0;
  0, 0, 1, 0;
  0, 0, 0, 1]

B = [4, 1, 1, 1;
  1, 4, 1, 1;
  1, 1, 4, 1;
  1, 1, 1, 4]

C = [5; 8; 5; 5]

C .* diag(B) = 92
```

图 3.8 Mat 表达式例程的输出结果

3.8 Mat_类

Mat_类是对 Mat 类的一个包装，其定义如下：

```
template<typename _Tp> class Mat_ : public Mat
{
public:
    //只定义了几个方法
    //没有定义新的属性
};
```

这是一个非常轻量级的包装，既然已经有 Mat 类，为何还要定义一个 Mat_？下面我们看这段代码：

```
Mat M(600, 800, CV_8UC1);
for( int i = 0; i < M.rows; ++i)
{
    uchar * p = M.ptr<uchar>(i);
    for( int j = 0; j < M.cols; ++j )
    {
        double d1 = (double) ((i+j)%255);
        M.at<uchar>(i,j) = d1;
        double d2 = M.at<double>(i,j);//此行有错
    }
}
```

在读取矩阵元素时，以及获取矩阵某行的地址时，需要指定数据类型。这样首先需要不停地写“<uchar>”，让人感觉很繁琐，在繁琐和烦躁中容易犯错，如

上面代码中的错误，用 at()获取矩阵元素时错误的使用了 double 类型。这种错误不是语法错误，因此在编译时编译器不会提醒。在程序运行时，at()函数获取到的不是期望的(i,j)位置处的元素，数据已经越界，但是运行时也未必会报错。这样的错误使得你的程序忽而看上去正常，忽而弹出“段错误”，特别是在代码规模很大时，难以查错。

如果使用 Mat_类，那么就可以在变量声明时确定元素的类型，访问元素时不再需要指定元素类型，即使得代码简洁，又减少了出错的可能性。上面代码可以用 Mat_实现，实现代码如下面例程里的第二个双重 for 循环。

```
#include <iostream>
#include "opencv2/opencv.hpp"
#include <stdio.h>
using namespace std;
using namespace cv;

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

    Mat M(600, 800, CV_8UC1);

    for( int i = 0; i < M.rows; ++i)
    {
        //获取指针时需要指定类型
        uchar * p = M.ptr<uchar>(i);
        for( int j = 0; j < M.cols; ++j )
        {
            double d1 = (double) ((i+j)%255);
            //用 at()读写像素时，需要指定类型
            M.at<uchar>(i,j) = d1;
            //下面代码错误，应该使用 at<uchar>()
            //但编译时不会提醒错误
            //运行结果不正确，d2 不等于 d1
            double d2 = M.at<double>(i,j);
        }
    }

    //在变量声明时指定矩阵元素类型
    Mat_<uchar> M1 = (Mat_<uchar>&)M;
    for( int i = 0; i < M1.rows; ++i)
    {
        //不需指定元素类型，语句简洁
```

```
            uchar * p = M1.ptr(i);
            for( int j = 0; j < M1.cols; ++j )
            {
                double d1 = (double) ((i+j)%255);
                //直接使用 Matlab 风格的矩阵元素读写，简洁
                M1(i,j) = d1;
                double d2 = M1(i,j);
            }
        }

        return 0;
    }
```

3.9 Mat 类的内存管理

使用 Mat 类，内存管理变得简单，不再像使用 IplImage 那样需要自己申请和释放内存。虽然不了解 Mat 的内存管理机制，也无碍于 Mat 类的使用，但是如果清楚了解 Mat 的内存管理，会更清楚一些函数到底操作了哪些数据。

Mat 是一个类，由两个数据部分组成：矩阵头（包含矩阵尺寸，存储方法，存储地址等信息）和一个指向存储所有像素值的矩阵的指针，如图 3.9 所示。矩阵头的尺寸是常数值，但矩阵本身的尺寸会依图像的不同而不同，通常比矩阵头的尺寸大数个数量级。复制矩阵数据往往花费较多时间，因此除非有必要，不要复制大的矩阵。

为了解决矩阵数据的传递，OpenCV 使用了引用计数机制。其思路是让每个 Mat 对象有自己的矩阵头信息，但多个 Mat 对象可以共享同一个矩阵数据。让矩阵指针指向同一地址而实现这一目的。很多函数以及很多操作（如函数参数传值）只复制矩阵头信息，而不复制矩阵数据。

前面提到过，有很多中方法创建 Mat 类。如果 Mat 类自己申请数据空间，那么该类会多申请 4 个字节，多出的 4 个字节存储数据被引用的次数。引用次数存储于数据空间的后面，refcount 指向这个位置，如图 3.9 所示。当计数等于 0 时，则释放该空间。

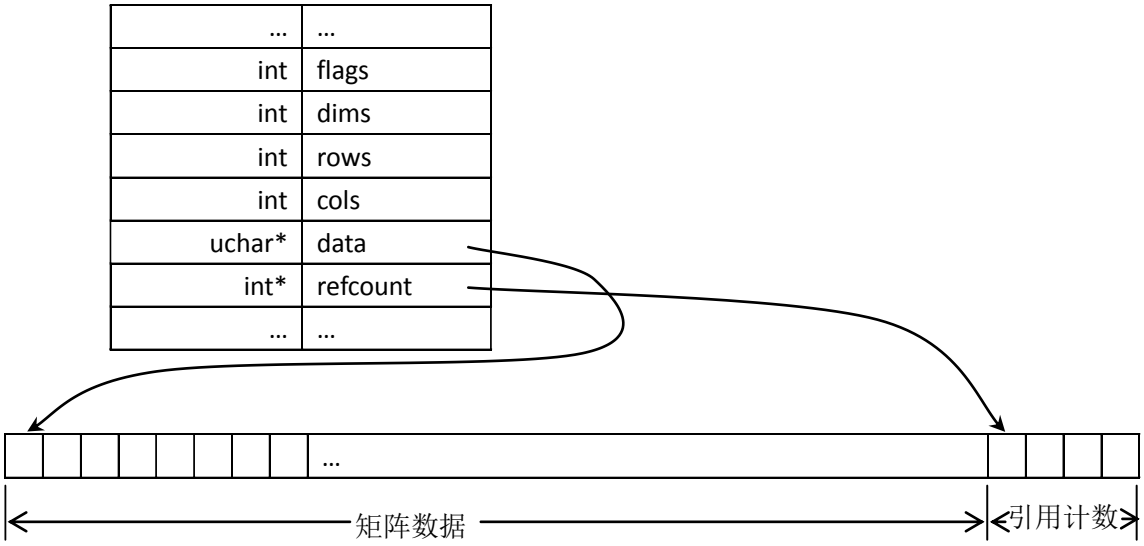

图 3.9 Mat 类中的数据存储示意图，refcount 变量指向数据区后面，用 4 个字节（int 类型）存储引用数目。

关于多个矩阵对象共享同一矩阵数据，我们可以看这个例子：

```
Mat A(100,100, CV_8UC1);

Mat B = A;

Mat C = A(Rect(50,50,30,30));
```

上面代码中有三个 Mat 对象，分别是 A，B 和 C。这三者共有同一矩阵数据，其示意图如图 3.10 所示。

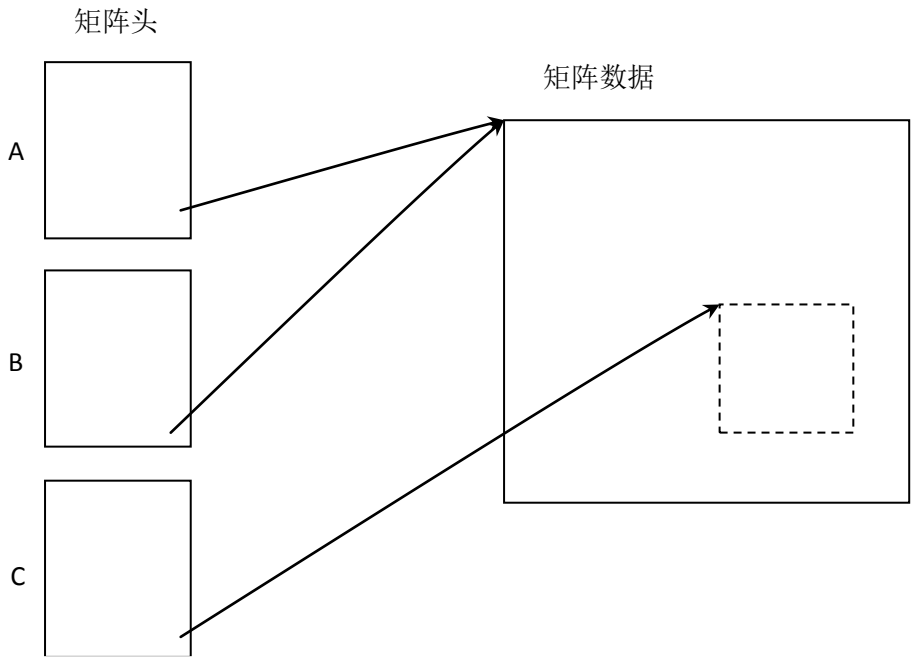

图 3.10 三个矩阵头共用共用同一矩阵数据

3.10 输出

从前面的例程中，可以看到 Mat 类重载了<<操作符，可以方便得使用流操作来输出矩阵的内容。默认情况下输出的格式是类似 Matlab 中矩阵的输出格式。除了默认格式，Mat 也支持其他的输出格式。代码如下：

首先创建一个矩阵，并用随机数填充。填充的范围由 randu()函数的第二个参数和第三个参数确定，下面代码是介于 0 到 255 之间。

```
Mat R = Mat(3, 2, CV_8UC3);
randu(R, Scalar::all(0), Scalar::all(255));
```

默认格式输出的代码如下：

```
cout << "R (default) = " << endl << R << endl << endl;
```

输出结果如图 3.11 所示。

```
R (default) =
[91, 2, 79, 179, 52, 205;
 236, 8, 181, 239, 26, 248;
 207, 218, 45, 183, 158, 101]
```

图 3.11 默认格式的矩阵输出

Python 格式输出的代码如下：

```
cout << "R (python)  = " << endl << format(R,"python") << endl
```

```
<< endl;
```

```
R (python)   =
[[[91, 2, 79], [179, 52, 205]],
  [[236, 8, 181], [239, 26, 248]],
  [[207, 218, 45], [183, 158, 101]]]
```

图 3.12 Python 格式的矩阵输出

以逗号分割的输出的代码如下：

```
    cout << "R (csv)      = " << endl << format(R,"csv"   ) << endl
<< endl;
```

```
R (csv)      =
91, 2, 79, 179, 52, 205
  236, 8, 181, 239, 26, 248
  207, 218, 45, 183, 158, 101
```

图 3.13 以逗号分割格式的矩阵输出

numpy 格式输出的代码如下：

```
    cout << "R (numpy)    = " << endl << format(R,"numpy" ) << endl
<< endl;
```

```
R (numpy)    =
array([[[91, 2, 79], [179, 52, 205]],
  [[236, 8, 181], [239, 26, 248]],
  [[207, 218, 45], [183, 158, 101]]], type='uint8')
```

图 3.14 numpy 格式的矩阵输出

C 语言格式输出的代码如下：

```
    cout << "R (c)       = " << endl << format(R,"C"     ) << endl <<
endl;
```

```
R (c)        =
{91, 2, 79, 179, 52, 205,
  236, 8, 181, 239, 26, 248,
  207, 218, 45, 183, 158, 101}
```

图 3.15 C 语言格式的矩阵输出

除了 Mat 对象可以使用<<符号输出，其他的很多类型也支持<<输出。

二维点：

```
    Point2f P(5, 1);
    cout << "Point (2D) = " << P << endl << endl;
```

```
Point (2D) = [5, 1]
```

图 3.16 二维点的输出结果

三维点：

```
Point3f P3f(2, 6, 7);
cout << "Point (3D) = " << P3f << endl << endl;
```

```
Point (3D) = [2, 6, 7]
```

图 3.17 三维点的输出结果

3.11 Mat 与 IplImage 和 CvMat 的转换

在 OpenCV 2 中虽然引入了方便的 Mat 类，出于兼容性的考虑，OpenCV 依然是支持 C 语言接口的 IplImage 和 CvMat 结构。如果你要与以前的代码兼容，将会涉及 Mat 与 IplImage 和 CvMat 的转换。

3.11.1 Mat 转为 IplImage 和 CvMat 格式

假如你有一个以前写的函数，函数的定义为：

```
void mycvOldFunc(IplImage * p, ...);
```

函数的参数需要 IplImage 类型的指针。Mat 转为 IplImage，可以用简单的等号赋值操作来进行类型转换，这样实现：

```
Mat img(Size(320, 240), CV_8UC3);
...
IplImage iplimg = img; //转为 IplImage 结构
mycvOldFunc( & iplimg, ...);//对 iplimg 取地址
```

如果要转为 CvMat 类型，操作类似：

```
CvMat cvimg = img; //转为 CvMat 结构
```

需要特别注意的是，类型转换后，IplImage 和 CvMat 与 Mat 共用同一矩阵数据，而 IplImage 和 CvMat 没有引用计数功能，如果上例中的 img 中数据被释放，iplimg 和 cvimg 也就失去了数据。因此要牢记不可将 Mat 对象提前释放。

3.11.2 IplImage 和 CvMat 格式转为 Mat

Mat 类有两个构造函数，可以实现 IplImage 和 CvMat 到 Mat 的转换。这两个函数都有一个参数 copyData。如果 copyData 的值是 false，那么 Mat 将与 IplImage 或 CvMat 共用同一矩阵数据；如果值是 true，Mat 会新申请内存，然后将 IplImage 或 CvMat 的数据复制到 Mat 的数据区。

如果共用数据，Mat 也将不会使用引用计数来管理内存，需要开发者自己来管理。本书建议做此转换是将参数置为 true，这样内存管理变得简单。

```
Mat::Mat(const CvMat* m, bool copyData=false)
Mat::Mat(const IplImage* img, bool copyData=false)
```

例子代码如下：

```
IplImage * iplimg = cvLoadImage("lena.jpg");
Mat im(iplimg, true);
```

第4章 数据获取与存储

4.1 读写图像文件

将图像文件读入内存，可以使用 imread()函数；将 Mat 对象以图像文件格式写入内存，可以使用 imwrite()函数。

4.1.1 读图像文件

imread()函数返回的是 Mat 对象，如果读取文件失败，则会返回一个空矩阵，即 Mat::data 的值是 NULL。执行 imread()之后，需要检查文件是否成功读入，你可以使用 Mat::empty()函数进行检查。imread()函数的声明如下：

```
Mat imread(const string& filename, int flags=1 )
```

很明显参数 filename 是被读取或者保存的图像文件名；在 imread()函数中，flag 参数值有三种情况：

- flag>0，该函数返回 3 通道图像，如果磁盘上的图像文件是单通道的灰度图像，则会被强制转为 3 通道；
- flag=0，该函数返回单通道图像，如果磁盘的图像文件是多通道图像，则会被强制转为单通道；
- flag<0，则函数不对图像进行通道转换。

imread()函数支持多种文件格式，且该函数是根据图像文件的内容来确定文件格式，而不是根据文件的扩展名来确定。所只是的文件格式如下：

- Windows 位图文件 - BMP, DIB；
- JPEG 文件 - JPEG, JPG, JPE；
- 便携式网络图片 - PNG；
- 便携式图像格式 - PBM，PGM，PPM；
- Sun rasters - SR，RAS；
- TIFF 文件 - TIFF，TIF;
- OpenEXR HDR 图片 - EXR;
- JPEG 2000 图片- jp2。

你所安装的 OpenCV 并不一定能支持上述所有格式，文件格式的支持需要特

定的库，只有在编译 OpenCV 添加了相应的文件格式库，才可支持其格式。

4.1.2 写图像文件

将图像写入文件，可使用 imwrite()函数，该函数的声明如下：

```
bool imwrite(const string& filename, InputArray image,
             const vector<int>& params=vector<int>())
```

文件的格式由 filename 参数指定的文件扩展名确定。推荐使用 PNG 文件格式。BMP 格式是无损格式，但是一般不进行压缩，文件尺寸非常大；JPEG 格式的文件娇小，但是 JPEG 是有损压缩，会丢失一些信息。PNG 是无损压缩格式，推荐使用。

imwrite()函数的第三个参数 params 可以指定文件格式的一些细节信息。这个参数里面的数值是跟文件格式相关的：

- JPEG：表示图像的质量，取值范围从 0 到 100。数值越大表示图像质量越高，当然文件也越大。默认值是 95。
- PNG：表示压缩级别，取值范围是从 0 到 9。数值越大表示文件越小，但是压缩花费的时间也越长。默认值是 3。
- PPM，PGM 或 PBM：表示文件是以二进制还是纯文本方式存储，取值为 0 或 1。如果取值为 1，则表示以二进制方式存储。默认值是 1。

并不是所有的 Mat 对象都可以存为图像文件，目前支持的格式只有 8U 类型的单通道和 3 通道（颜色顺序为 BGR）矩阵；如果需要要保存 16U 格式图像，只能使用 PNG、JPEG 2000 和 TIFF 格式。如果希望将其他格式的矩阵保存为图像文件，可以先用 Mat::convertTo()函数或者 cvtColor()函数将矩阵转为可以保存的格式。

另外需要注意的是，在保存文件时，如果文件已经存在，imwrite()函数不会进行提醒，将直接覆盖掉以前的文件。

下面例程展示了如何读入一副图像，然后对图像进行 Canny 边缘操作，最后将结果保存到图像文件中。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char* argv[])
{
```

```
    //读入图像，并将之转为单通道图像
    Mat im = imread("lena.jpg", 0);
    //请一定检查是否成功读图
    if( im.empty() )
    {
        cout << "Can not load image." << endl;
        return -1;
    }

    //进行 Canny 操作，并将结果存于 result
    Mat result;
    Canny(im, result, 50, 150);

    //保存结果
    imwrite("lena-canny.png", result);

    return 0;
}
```

将 lena.jpg 文件放在当前目录，运行该例程后，lena-canny.png 将会出现在当前目录。lena-canny.png 图像如图 4.1 所示，是 lena.jpg 的边缘提取结果。

图 4.1 Lena 图像的边缘提取结果

4.2 读写视频

介绍 OpenCV 读写视频之前，先介绍一下编解码器（codec）。如果是图像文件，我们可以根据文件扩展名得知图像的格式。但是此经验并不能推广到视频文件中。有些 OpenCV 用户会碰到奇怪的问题，都是 avi 视频文件，有的能用 OpenCV 打开，有的不能。

视频的格式主要由压缩算法决定。压缩算法称之为编码器（**co**der），解压算法称之为解码器（**dec**oder），编解码算法可以统称为编解码器（codec）。视频文件能读或者写，关键看是否有相应的编解码器。编解码器的种类非常多，常用的有 MJPG、XVID、DIVX 等，完整的列表请参考 FOURCC 网站[3]。因此视频文件的扩展名（如 avi 等）往往只能表示这是一个视频文件。

OpenCV 2 中提供了两个类来实现视频的读写。读视频的类是 VideoCapture，写视频的类是 VideoWriter。

4.2.1 读视频

VideoCapture 既可以从视频文件读取图像，也可以从摄像头读取图像。可以使用该类的构造函数打开视频文件或者摄像头。如果 VideoCapture 对象已经创建，也可以使用 VideoCapture::open()打开，VideoCapture::open()函数会自动调用 VideoCapture::release()函数，先释放已经打开的视频，然后再打开新视频。

如果要读一帧，可以使用 VideoCapture::read()函数。VideoCapture 类重载了>>操作符，实现了读视频帧的功能。下面的例程演示了使用 VideoCapture 类读视频。

```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    //打开第一个摄像头
    //VideoCapture cap(0);

    //打开视频文件
    VideoCapture cap("video.short.raw.avi");
```

[3] 网址：http://www.fourcc.org/codecs.php

```
    //检查是否成功打开
    if(!cap.isOpened())
    {
        cerr << "Can not open a camera or file." << endl;
        return -1;
    }

    Mat edges;
    //创建窗口
    namedWindow("edges",1);

    for(;;)
    {
        Mat frame;
        //从 cap 中读一帧，存到 frame
        cap >> frame;
        //如果未读到图像
        if(frame.empty())
            break;
        //将读到的图像转为灰度图
        cvtColor(frame, edges, CV_BGR2GRAY);
        //进行边缘提取操作
        Canny(edges, edges, 0, 30, 3);
        //显示结果
        imshow("edges", edges);
        //等待 30 秒，如果按键则推出循环
        if(waitKey(30) >= 0)
            break;
    }
    //退出时会自动释放 cap 中占用资源
    return 0;
}
```

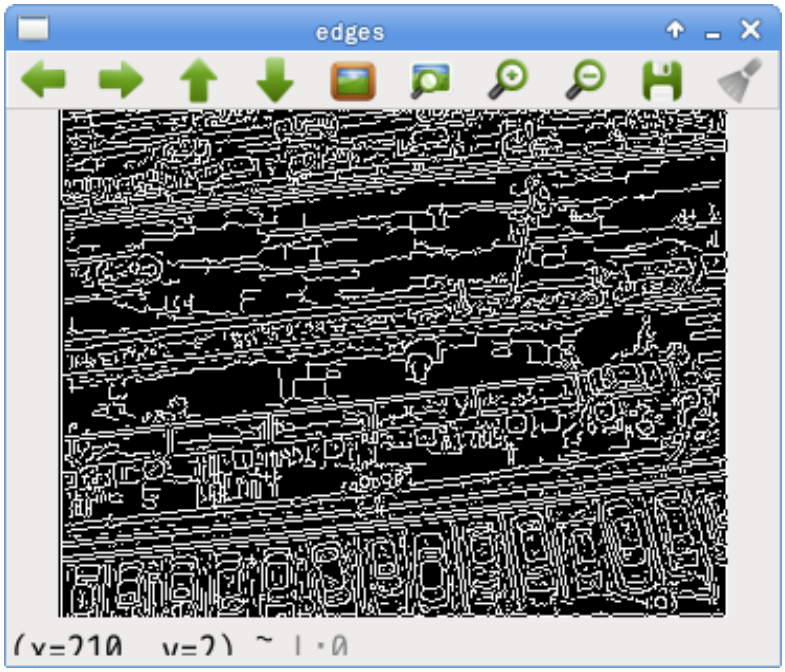

图 4.2 读视频例程的运行界面

4.2.2 写视频

使用 OpenCV 创建视频也非常简单，与读视频不同的是，你需要在创建视频时设置一系列参数，包括：文件名，编解码器，帧率，宽度和高度等。编解码器使用四个字符表示，可以是 CV_FOURCC('M','J','P','G')、CV_FOURCC('X','V','I','D')及 CV_FOURCC('D','I','V','X')等。如果使用某种编解码器无法创建视频文件，请尝试其他的编解码器。

将图像写入视频可以使用 VideoWriter::write()函数，VideoWriter 类中也重载了<<操作符，使用起来非常方便。另外需要注意：待写入的图像尺寸必须与创建视频时指定的尺寸一致。

下面例程演示了如何写视频文件。本例程将生成一个视频文件，视频的第 0 帧上是一个红色的“0”，第 1 帧上是个红色的“1”，以此类推，共 100 帧。生成视频的播放效果如图 4.3 所示。

```
#include <stdio.h>
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

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

```
    {
        //定义视频的宽度和高度
        Size s(320, 240);
        //创建 writer，并指定 FOURCC 及 FPS 等参数
        VideoWriter     writer     =     VideoWriter("myvideo.avi",
CV_FOURCC('M','J','P','G'), 25, s);
        //检查是否成功创建
        if(!writer.isOpened())
        {
            cerr << "Can not create video file.\n" << endl;
            return -1;
        }

        //视频帧
        Mat frame(s, CV_8UC3);

        for(int i = 0; i < 100; i++)
        {
            //将图像置为黑色
            frame = Scalar::all(0);
            //将整数 i 转为 i 字符串类型
            char text[128];
            snprintf(text, sizeof(text), "%d", i);
            //将数字绘到画面上
            putText(frame,   text,   Point(s.width/3,   s.height/3),
FONT_HERSHEY_SCRIPT_SIMPLEX, 3,
                    Scalar(0,0,255), 3, 8);
            //将图像写入视频
            writer << frame;
        }

        //退出程序时会自动关闭视频文件
        return 0;
    }
```

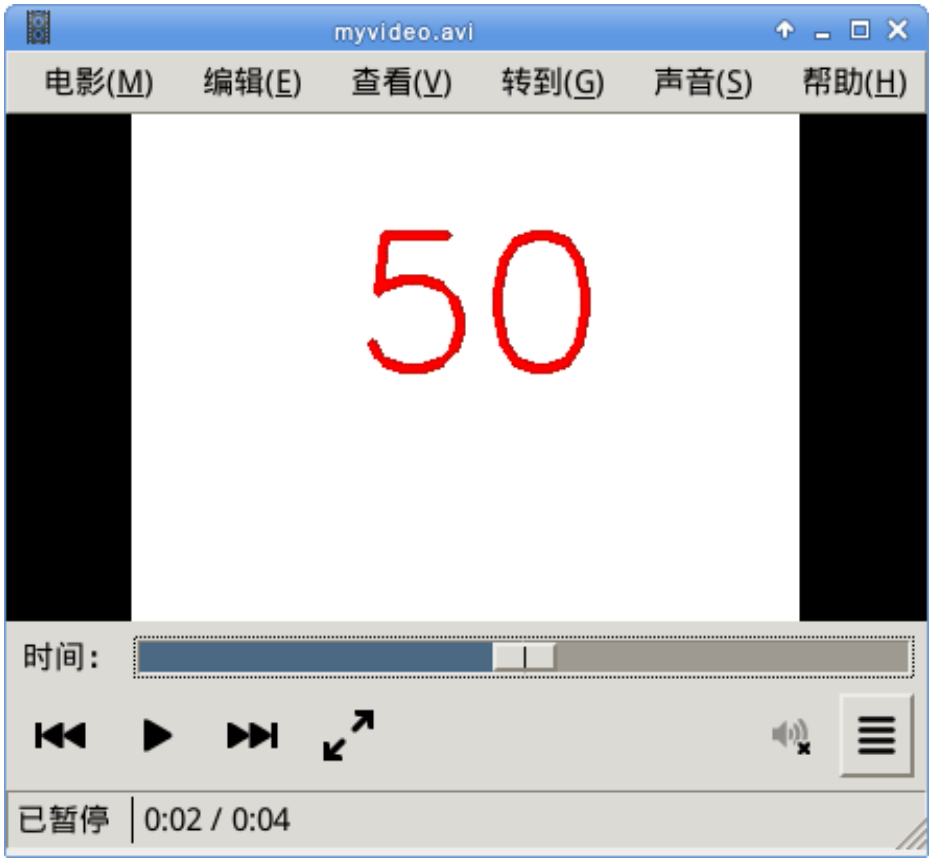

图 4.3 写视频例程生成的视频的播放效果