

安装

0.1 需要准备的安装包

0.2 安装Python(x,y)

0.3 安装PCV库

0.4 VLfeat

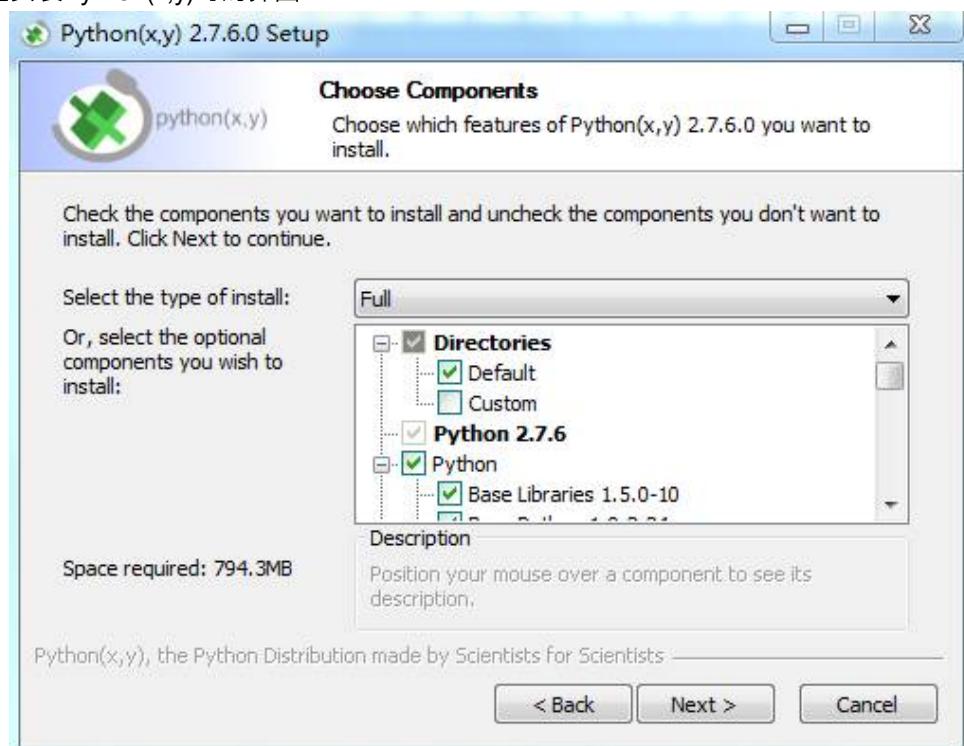
为顺利帮助读者完成本书中实例的学习，译者已对代码做了相应整理，下面给出在对本书实例学习前，你需要做的前期安装工作。注意，下面译者给出的安装过程是针对**Windows**下的，其他平台如Linux、Mac请查阅中译本附录。

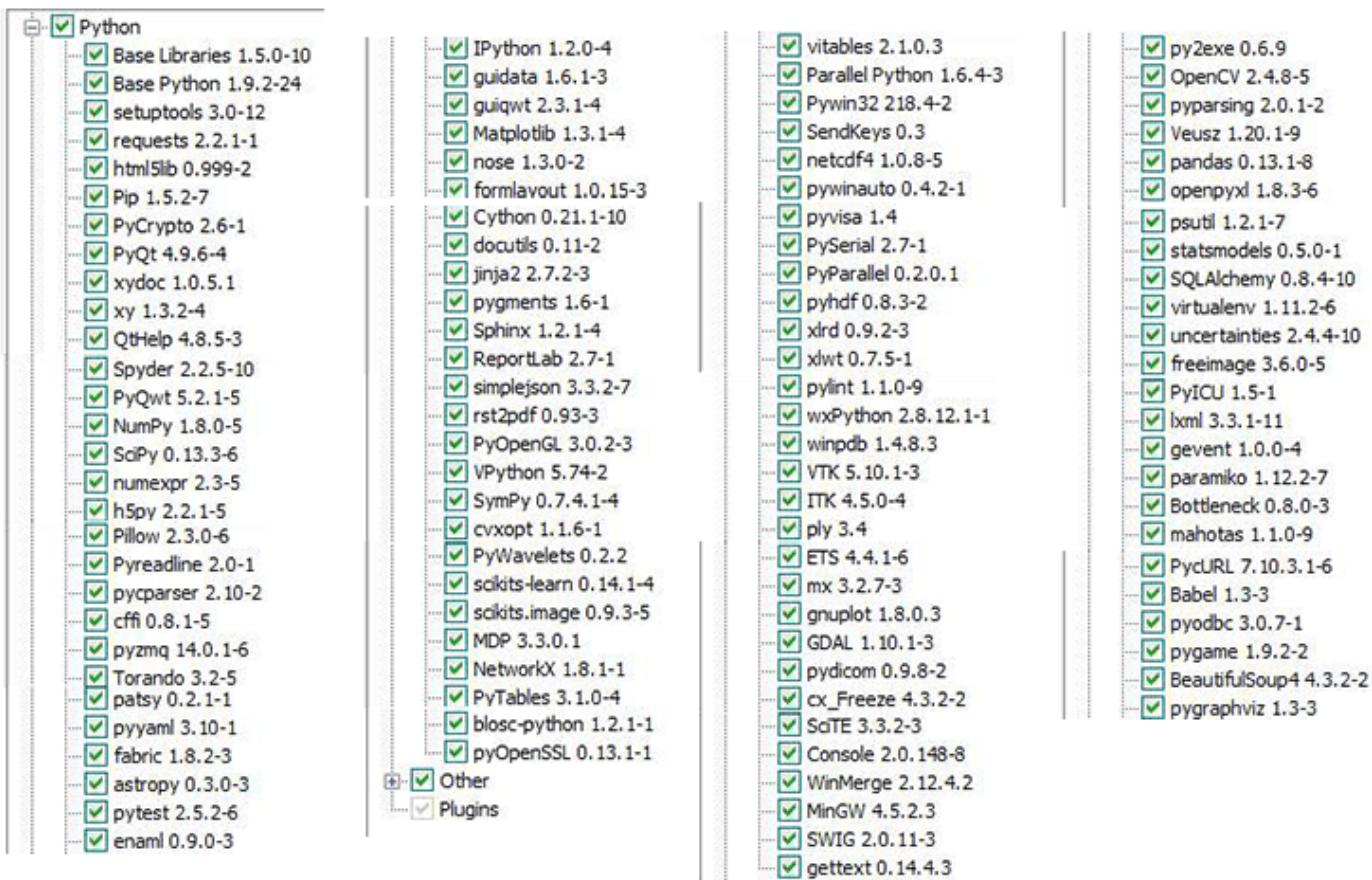
0.1 需要准备的安装包

要完整复现书中的实例，你需要的主要四个文件包括Python(x,y) 2.7.x安装包、PCV库、[VLfeat](http://www.vlfeat.org/download.html)(<http://www.vlfeat.org/download.html>)和本书用到的数据库。Python(x,y)可以在[Google Code](https://code.google.com/p/pythonxy/)(<https://code.google.com/p/pythonxy/>)，PCV库、本书整理出来的实例代码以及本书用到的所有图像数据可以从[首页](http://yuanyong.org/pcvwithpython/)(<http://yuanyong.org/pcvwithpython/>)给出的链接下载。

0.2 安装Python(x,y)

在Windows下，译者推荐你安装Python(x,y) 2.7.x。Python(x,y) 2.7.x是一个库安装包，除了包含Python自身外，还包含了很多第三方库，下面是安装Python(x,y)时的界面：





从上面第二幅图可以看出，pythonxy不仅包含了SciPy、NumPy、PyLab、OpenCV、Matplotlib,还包含了机器学习库scikits-learn。为避免出现运行实例时出现的依赖问题，译者建议将上面的库全部选上，也就是选择“full”(译者也是用的全部安装的方式进行后面的实验的)。安装完成后，为验证安装是否正确，可以在Python shell里确认一下OpenCV是否已安装来进行验证，在Python Shell里输入下面命令：

```
from cv2 import __version__
__version__
```

输入上面命令，如果可以看到OpenCV的版本信息，则说明python(x,y)已安装正确。

另外，需要提醒读者的是，Python是没有平台区分的，这里指的平台不是指Linux和Mac这样的平台概念，而是在Windows上没有位数的区分。举个简单的例子，比如你是64位的Windows系统，你可以安装32位的Python。对于这一部分的详细说明，可以参阅译者的一篇博文[Django配置MySQL \(<http://yuanyong.org/blog/config-mysql-for-django.html>\)](http://yuanyong.org/blog/config-mysql-for-django.html)最后一段的说明。好了，关于Python(x,y)的安装说明就说到这里。

0.3 安装PCV库

PCV库是原书作者写的一个第三方库，书中几乎所有的实例到要用到改库。假设你已从下载本书由译者整理的中译版源码，从Windows cmd终端进入PCV所在目录：

```
cd PCV
python setup.py install
```

运行上面命令，即可完成PCV库的安装。为了验证PCV库是否安装成功，在运行上面命令后，可以打开Python自带的Shell，在Shell输入：

```
import PCV
```

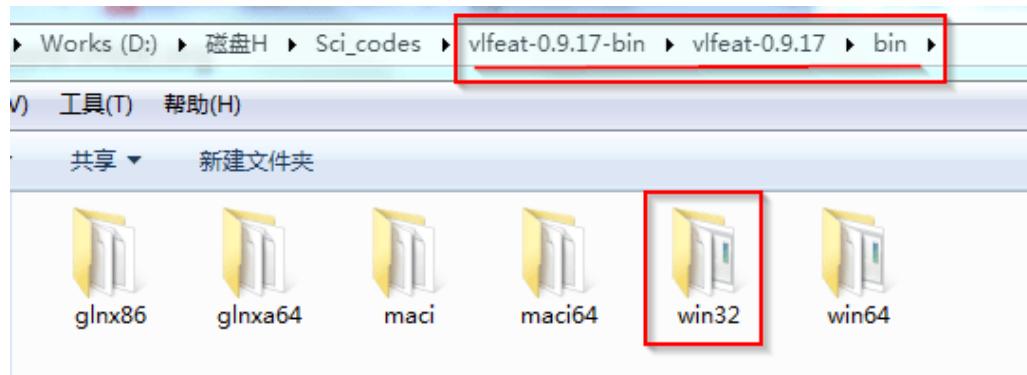
如果未报错，则表明你已成功安装了该PCV库。

0.4 VLfeat

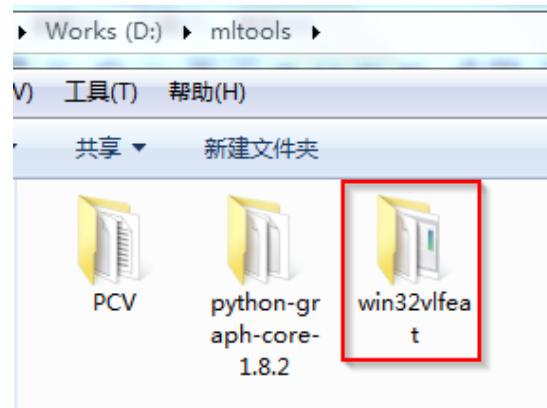
VLFeat是一个跨平台的开源机器视觉库，它囊括了当前流行的机器视觉算法，如SIFT, MSER, HOG, 同时还包含了诸如K-MEANS, Hierarchical K-means的聚类算法。本书中主要在提取sift特征时用到了VLfeat。

The screenshot shows the VLFeat.org website. At the top, there is a navigation bar with links for Home, Download, Tutorials, Applications, and Documentation. Below the navigation bar, there is a main content area with a brief description of the library and an ACM OpenSource Award 2010 badge. On the left, there is a 'Download' section with a list of links: 'VLFeat (Windows, Mac, Linux)' (which is highlighted with a red box), 'Source code and installation', and 'git repository, bug tracking'. On the right, there is a 'Documentation' section with links to 'MATLAB commands', 'C API with algorithm descriptions', and 'Command line tools'.

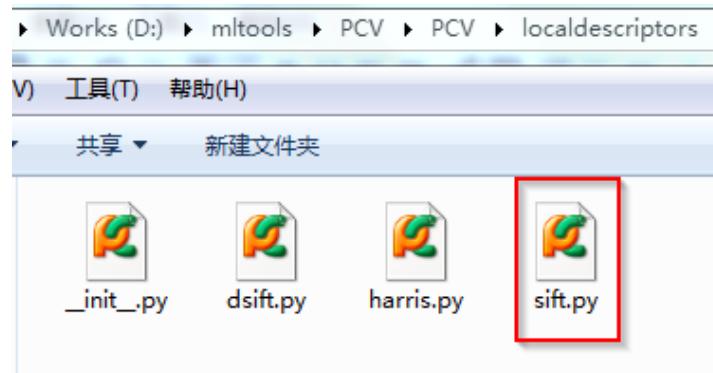
如上图所示，从红色框标的地方下载VLFeat,解压：



你需要的仅是对应平台的可执行文件，译者系统是32位的，所以选用的是win32。注意目前VLFeat最新发布版已到0.9.18了。对于0.9.18，目录结构和0.9.17的一样，所以你也仅需bin下对应的文件夹下的可执行文件。将该win32拷贝到你想放置的某个目录，译者将其放置在计算机的如下目录：



需要注意的是，译者将原来的“bin”文件名重新“win32vlfeat”。完成该步骤后，进入PCV所在目录：



打开sift.py，找到下面代码：

```
def process_image(imagename,resultname,params="--edge-thresh 10 --peak-thresh 5"):  
    """ process an image and save the results in a file"""\n  
  
    if imagename[-3:] != 'pgm':  
        #create a pgm file  
        im = Image.open(imagename).convert('L')  
        im.save('tmp.pgm')  
        imagename = 'tmp.pgm'  
  
    cmmnd = str("D:\\mltools\\win32vlfeat\\sift.exe "+imagename+" --output="+resultname+  
                " "+params)  
    os.system(cmmnd)  
    print 'processed', imagename, 'to', resultname
```

将cmmnd中的目录修改为你自己放置的Vlfeat bin所在目录。这里稍微解释一下os.system(cmmnd)这句话的意思，这里Python通过os.system () 调用外部可执行文件，也就是Vlfeat bin目录下的sift.exe。

好了，安装完后，你便可以运行书中的大部分实例代码了。这里之所以是“大部分”是因为书中的某些实例，还要用到别的库。

[« 作者译者 \(author.html\)](#)

[第一章 图像处理基础 » \(chapter1.html\)](#)

第一章 图像处理基础

1.1 PIL-Python图像库

- 1.1.1 对图片进行格式转换
- 1.1.2 创建缩略图
- 1.1.3 拷贝并粘贴区域
- 1.1.4 调整尺寸及旋转

1.2 Matplotlib库

- 1.2.1 画图、描点和线
- 1.2.2 图像轮廓和直方图
- 1.2.4 交互注释

1.3 NumPy库

- 1.3.1 图像数组表示
- 1.3.2 灰度变换
- 1.3.3 调整图像尺寸
- 1.3.3 直方图均衡化
- 1.3.4 图像平均
- 1.3.5 对图像进行主成分分析
- 1.3.6 Pickle模块

1.4 SciPy模块

- 1.4.1 图像模糊
- 1.4.2 图像差分
- 1.4.3 形态学-物体计数
- 1.4.4 有用的SciPy模块

1.5 更高级的例子：图像降噪

1.1 PIL-Python图像库

PIL (Python Imaging Library)图像库提供了很多常用的图像处理及很多有用的图像基本操作。PIL库下载地址[\[www.pythonware.com/products/pil/\]](http://www.pythonware.com/products/pil/) (<http://www.pythonware.com/products/pil/>)。下面演示原书P001-Fig1-1读入一幅图片的例子：

```

# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)
figure()

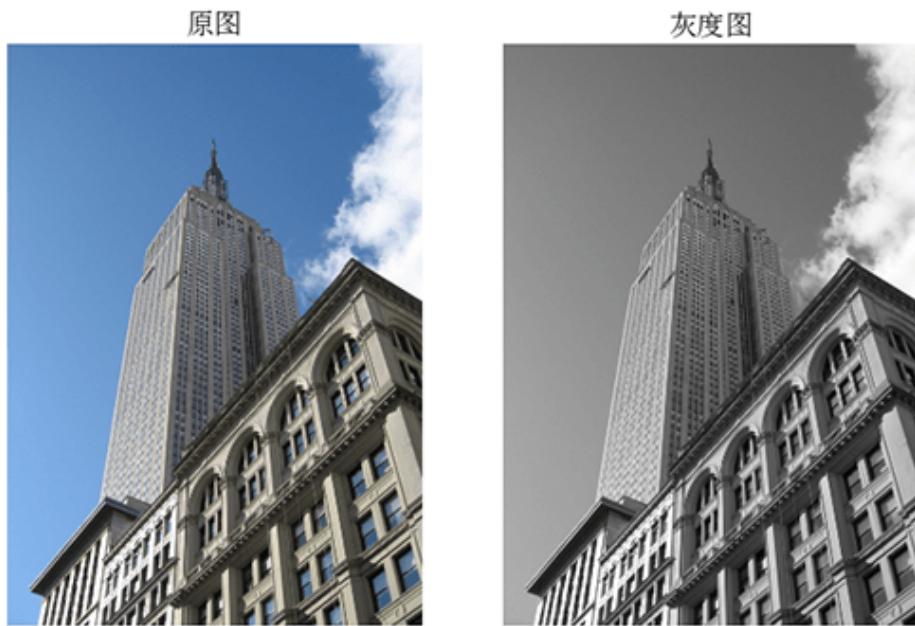
pil_im = Image.open('..../data/empire.jpg')
gray()
subplot(121)
title(u'原图', fontproperties=font)
axis('off')
imshow(pil_im)

pil_im = Image.open('..../data/empire.jpg').convert('L')
subplot(122)
title(u'灰度图', fontproperties=font)
axis('off')
imshow(pil_im)

show()

```

运行上面的代码，可以得出原书P002-Fig1-1中的前两幅图片，如下：



更多关于PIL的实例，可以参阅PIL在线文档[\[www.pythonware.com/library/pil/handbook/index.htm.\]](http://www.pythonware.com/library/pil/handbook/index.htm)
[\(<http://www.pythonware.com/library/pil/handbook/index.htm>\)](http://www.pythonware.com/library/pil/handbook/index.htm)。

1.1.1 对图片进行格式转换

利用`save()`方法，PIL可以将图片保存问很多不同的图像格式。下面演示原书P002中对图片进行转换的例子。

```

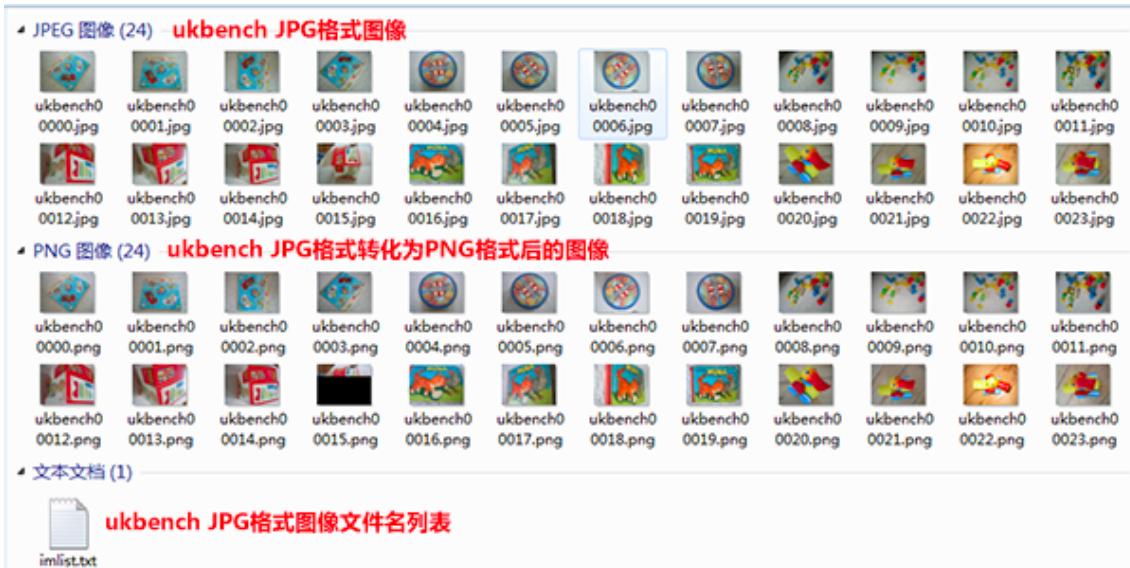
# -*- coding: utf-8 -*-
from PCV.tools.imtools import get_imlist #导入原书的PCV模块
from PIL import Image
import os
import pickle

filelist = get_imlist('../data/convert_images_format_test/') #获取convert_images_format_test文件夹下的图片文件名(包括后缀名)
imlist = file('../data/convert_images_format_test/imlist.txt','w') #将获取的图片文件列表保存到imlist.txt中
pickle.dump(filelist,imlist) #序列化
imlist.close()

for infile in filelist:
    outfile = os.path.splitext(infile)[0] + ".png" #分离文件名与扩展名
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print "cannot convert", infile

```

上面convertimagesformat_test文件夹是译者放的测试图片，共24幅图像，如下图示，测试图片全部为.jpg格式的。译者在源代码中添加了部分代码以便将获取的图像文件名列表保存下来，同时将原来的所有图像转化为.png格式的图像。注意，在载入模块时，载入了原书的PCV模块，关于PCV模块的安装，详见[PCV模块的安装]。运行上面代码，可以得到转化格式后的图像，运行结果为：



1.1.2 创建缩略图

利用PIL可以很容易的创建缩略图，设置缩略图的大小，并用元组保存起来，调用thumbnail()方法即可生成缩略图。创建缩略图的代码见下面。

1.1.3 拷贝并粘贴区域

调用crop()方法即可从一幅图像中进行区域拷贝，拷贝出区域后，可以对区域进行旋转等变换。关于拷贝、旋转粘贴的代码见下面。

1.1.4 调整尺寸及旋转

要对一幅图像的尺寸进行调整，可以调用`resize()`方法，元组中放置的便是你要调整尺寸的大小。如果要对图像进行旋转变换的话，可以调用`rotate()`方法。

下面代码显示上面提到的所有的图像处理操作，即原图显示、RGB图像转为灰度图像、拷贝粘贴区域、生成缩略图、调整图像尺寸、图像旋转变换的实例代码：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)
figure()

# 显示原图
pil_im = Image.open('../data/empire.jpg')
print pil_im.mode, pil_im.size, pil_im.format
subplot(231)
title(u'原图', fontproperties=font)
axis('off')
imshow(pil_im)

# 显示灰度图
pil_im = Image.open('../data/empire.jpg').convert('L')
gray()
subplot(232)
title(u'灰度图', fontproperties=font)
axis('off')
imshow(pil_im)

#拷贝粘贴区域
pil_im = Image.open('../data/empire.jpg')
box = (100,100,400,400)
region = pil_im.crop(box)
region = region.transpose(Image.ROTATE_180)
pil_im.paste(region,box)
subplot(233)
title(u'拷贝粘贴区域', fontproperties=font)
axis('off')
imshow(pil_im)

# 缩略图
pil_im = Image.open('../data/empire.jpg')
size = 128, 128
pil_im.thumbnail(size)
print pil_im.size
subplot(234)
```

```

title(u'缩略图', fontproperties=font)
axis('off')
imshow(pil_im)
pil_im.save('../images/ch01/thumbnail.jpg') #保存缩略图

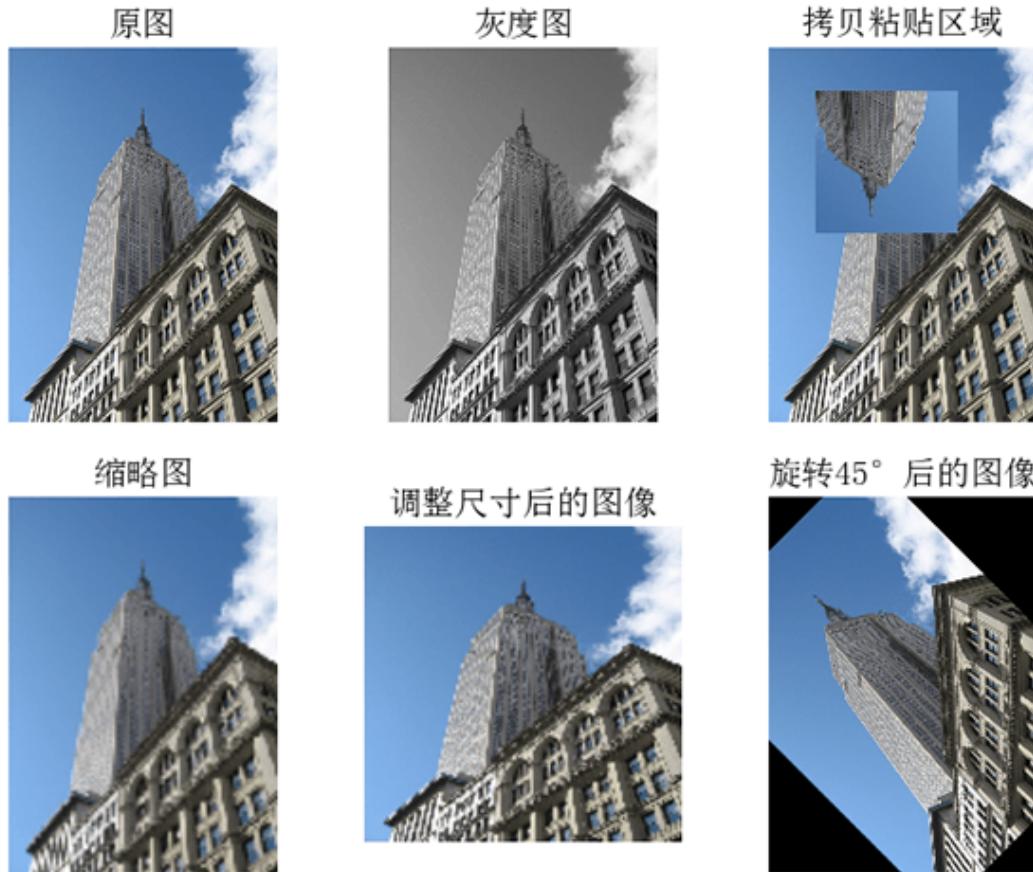
# 调整图像尺寸
pil_im = Image.open('../data/empire.jpg')
pil_im = pil_im.resize(size)
print pil_im.size
subplot(235)
title(u'调整尺寸后的图像', fontproperties=font)
axis('off')
imshow(pil_im)

# 旋转图像45°
pil_im = Image.open('../data/empire.jpg')
pil_im = pil_im.rotate(45)
subplot(236)
title(u'旋转45°后的图像', fontproperties=font)
axis('off')
imshow(pil_im)

show()

```

运行上面代码，可得P002 Figure 1-1中出现的所有实例图，结果如下：



1.2 Matplotlib库

当在处理数学及绘图或在图像上描点、画直线、曲线时，Matplotlib是一个很好的绘图库，它比PIL库提供了更有力的特性。Matplotlib是开源的，可以在[\[matplotlib.sourceforge.net\]](http://matplotlib.sourceforge.net) (<http://matplotlib.sourceforge.net>) 上下载，并且它还提供了详细的文档及教程。这里，会展示一些我们在本书后面会用到的函数的一些实例。

1.2.1 画图、描点和线

虽然Matplotlib可以创建漂亮的条状图、饼图、散点图等，但是在很多计算机视觉应用场合，其实只用到了一些常用的命令。下面展示在一幅图像上描一些点和画一条直线的例子。

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttf", size=14)

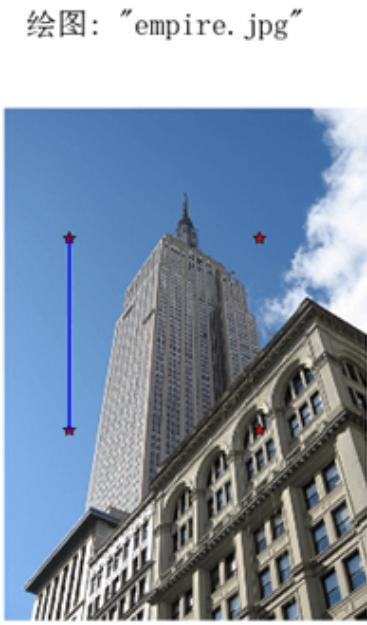
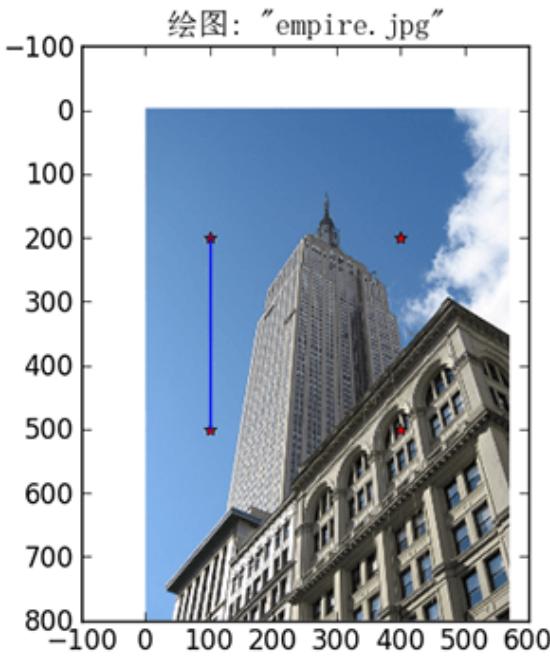
im = array(Image.open('../data/empire.jpg'))
figure()

# 画有坐标轴的
subplot(121)
imshow(im)
x = [100, 100, 400, 400]
y = [200, 500, 200, 500]
plot(x, y, 'r*')
plot(x[:2], y[:2])
title(u'绘图: "empire.jpg"', fontproperties=font)

# 不显示坐标轴
subplot(122)
imshow(im)
x = [100, 100, 400, 400]
y = [200, 500, 200, 500]
plot(x, y, 'r*')
plot(x[:2], y[:2])
axis('off') #显示坐标轴
title(u'绘图: "empire.jpg"', fontproperties=font)

show()
```

运行上面代码，即可得原书P005中 Figure 1-2中左边的结果。去掉上面代码中坐标轴的注释，即可得 Figure 1-2中右边的结果。运行结果如下：



1.2.2 图像轮廓和直方图

下面我们看两个特别的例子：图像轮廓线和图线等高线。在画图像轮廓前需要转换为灰度图像，因为轮廓需要获取每个坐标 $[x,y]$ 位置的像素值。下面是画图像轮廓和直方图的代码：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *

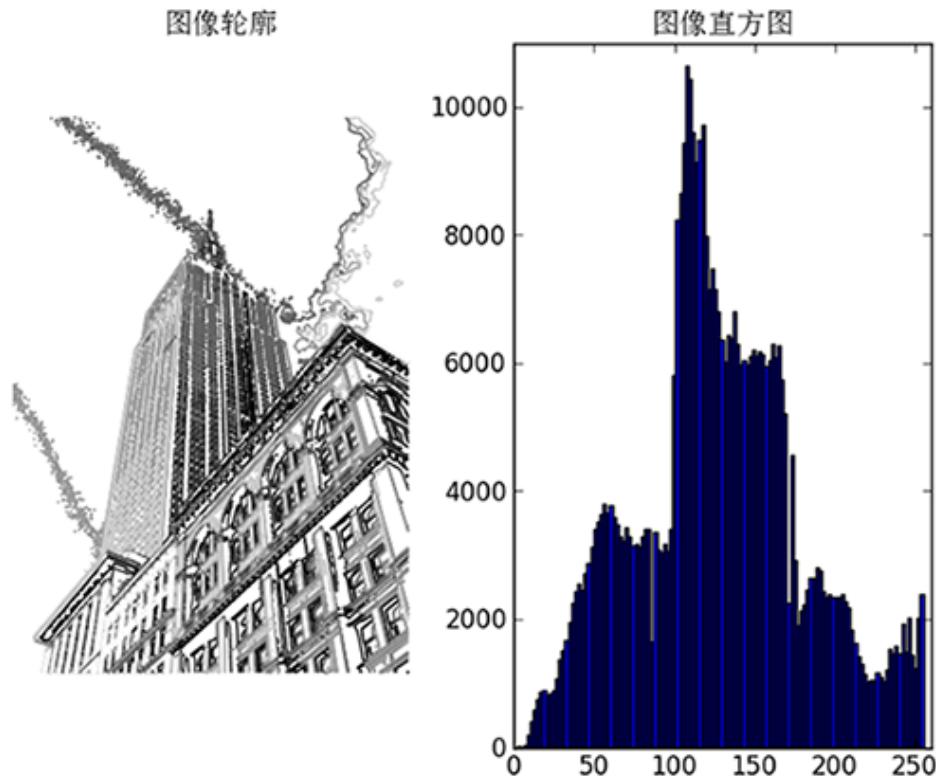
# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)
im = array(Image.open('../data/empire.jpg').convert('L')) # 打开图像，并转成灰度图像

figure()
subplot(121)
gray()
contour(im, origin='image')
axis('equal')
axis('off')
title(u'图像轮廓', fontproperties=font)

subplot(122)
hist(im.flatten(), 128)
title(u'图像直方图', fontproperties=font)
plt.xlim([0,260])
plt.ylim([0,11000])

show()
```

运行上面代码，可以得到书中的图1-3所示的结果：



1.2.4 交互注释

有时，用户需要和应用进行交互，比如在图像中用点做标识，或者在一些训练数据中进行注释。PyLab提供了一个很简洁好用的函数ginput(),它可以完成该任务，下面是一个演示交互注释的简短示例：

```
from PIL import Image
from pylab import *

im = array(Image.open('../data/empire.jpg'))
imshow(im)
print 'Please click 3 points'
imshow(im)
x = ginput(3)
print 'You clicked:', x

show()
```

上面代码先读取empire.jpg图像，显示读取的图像，然后用ginput()交互注释，这里设置的交互注释数据点设置为3个，用户在注释后，会将注释点的坐标打印出来。

1.3 NumPy库

[NumPy \(<http://www.scipy.org/NumPy/>\)](http://www.scipy.org/NumPy/)是Python一个流行的用于科学计算包。它包含了很多诸如矢量、矩阵、图像等其他非常有用的对象和线性代数函数。在本书中几乎所有的例子都用到了NumPy数组对象。NumPy可以在[scipy.org/Download](http://www.scipy.org/Download) (<http://www.scipy.org/Download>)下载，在线文档 (<http://docs.scipy.org/doc/numpy/>)包含了很多常见问题的答案。

1.3.1 图像数组表示

在前面载入图像的示例中，我们将图像用array()函数转为NumPy数组对象，但是并没有提到它表示的含义。数组就像列表一样，只不过它规定了数组中的所有元素必须是相同的类型。下面的例子用于说明图像数组表示：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *

im = array(Image.open('../data/empire.jpg'))
print im.shape, im.dtype
im = array(Image.open('../data/empire.jpg').convert('L'), 'f')
print im.shape, im.dtype
```

运行上面代码，会给出下面结果：

```
(800, 569, 3) uint8
(800, 569) float32
```

数组可以通过索引访问和操作其中的元素。比如：`value=im[i,j,k]`。`i,j`是坐标，`k`是颜色通道。对于多个元素，可以用切片操作，如：

```
im[i,:] = im[j,:]
im[:,i] = 100
im[:100,:50].sum()
im[50:100,50:100]
im[i].mean()
im[:, -1]
im[-2, :]
```

在使用数组时有很多操作和方式，我们会在后面介绍贯穿于本书所需要的操作。

1.3.2 灰度变换

在读入图像到NumPy数组后，就可以对它进行任何我们想要的操作了。对图像进行灰度变换便是一个简单的例子。这里给出一些进行灰度变换的例子：

```

# -*- coding: utf-8 -*-
from PIL import Image
from numpy import *
from pylab import *

im = array(Image.open('../data/empire.jpg').convert('L'))
print int(im.min()), int(im.max())

im2 = 255 - im # invert image
print int(im2.min()), int(im2.max())

im3 = (100.0/255) * im + 100 # clamp to interval 100...200
print int(im3.min()), int(im3.max())

im4 = 255.0 * (im/255.0)**2 # squared
print int(im4.min()), int(im4.max())

figure()
gray()
subplot(1, 3, 1)
imshow(im2)
axis('off')
title(r'$f(x)=255-x$')

subplot(1, 3, 2)
imshow(im3)
axis('off')
title(r'$f(x)=\frac{100}{255}x+100$')

subplot(1, 3, 3)
imshow(im4)
axis('off')
title(r'$f(x)=255(\frac{x}{255})^2$')
show()

```

上面左边灰度变换函数采用的是 $f(x)=255-x$,中间采用的是 $f(x)=(100/255)x+100$,右边采用的是变换函数是 $f(x)=255(x/255)^2$ 。运行上面代码, 可以得到P009 Fig1-5中的结果:

$$f(x) = 255 - x$$



$$f(x) = \frac{100}{255}x + 100$$



$$f(x) = 255\left(\frac{x}{255}\right)^2$$



正如上面代码所示，你可以用通过下面命令检查每幅图像的最小值和最大值：

```
print int(im.min()), int(im.max())
```

如果你对每幅图像用到了打印最小像素值和最大像素值，你会得到下面的输出结果：

```
2 255
0 253
100 200
0 255
```

1.3-3 调整图像尺寸

NumPy数组将成为我们对图像及数据进行处理的最主要工具，但是调整矩阵大小并没有一种简单的方法。我们可以用PIL图像对象转换写一个简单的图像尺寸调整函数：

```
def imresize(im,sz):
    """ Resize an image array using PIL. """
    pil_im = Image.fromarray(uint8(im))

    return array(pil_im.resize(sz))
```

上面定义的调整函数，在imtools.py中你可以找到它。

1.3.3 直方图均衡化

一个极其有用的例子是灰度变换后进行直方图均衡化。图像均衡化作为预处理操作，在归一化图像强度时是一个很好的方式，并且通过直方图均衡化可以增加图像对比度。下面是对图像直方图进行均衡化处理的例子：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from PCV.tools import imtools

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

im = array(Image.open('../data/empire.jpg').convert('L')) # 打开图像，并转成灰度图像
#im = array(Image.open('../data/AquaTermi_Lowcontrast.JPG').convert('L'))
im2, cdf = imtools.histeq(im)

figure()
subplot(2, 2, 1)
axis('off')
gray()
title(u'原始图像', fontproperties=font)
imshow(im)

subplot(2, 2, 2)
axis('off')
title(u'直方图均衡化后的图像', fontproperties=font)
imshow(im2)

subplot(2, 2, 3)
axis('off')
title(u'原始直方图', fontproperties=font)
#hist(im.flatten(), 128, cumulative=True, normed=True)
hist(im.flatten(), 128, normed=True)

subplot(2, 2, 4)
axis('off')
title(u'均衡化后的直方图', fontproperties=font)
#hist(im2.flatten(), 128, cumulative=True, normed=True)
hist(im2.flatten(), 128, normed=True)

show()
```

运行上面代码，可以得到书中的结果：

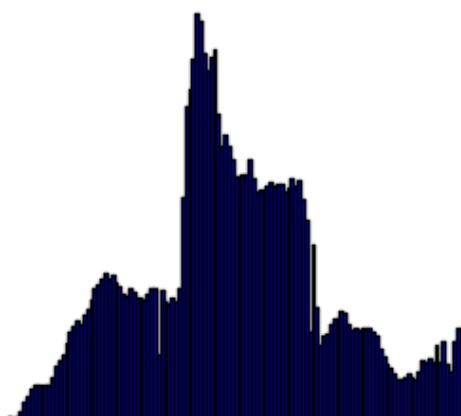
原始图像



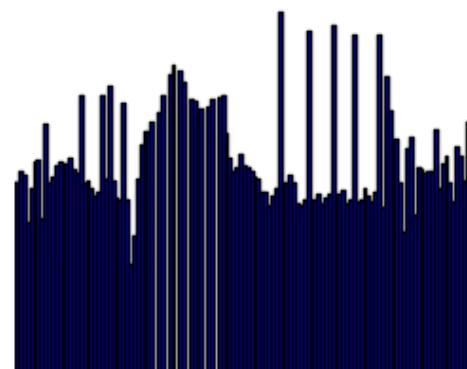
直方图均衡化后的图像



原始直方图



均衡化后的直方图



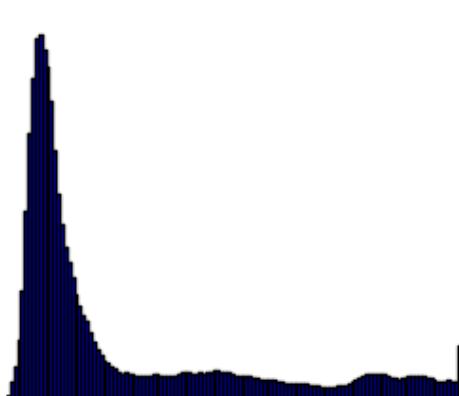
原始图像



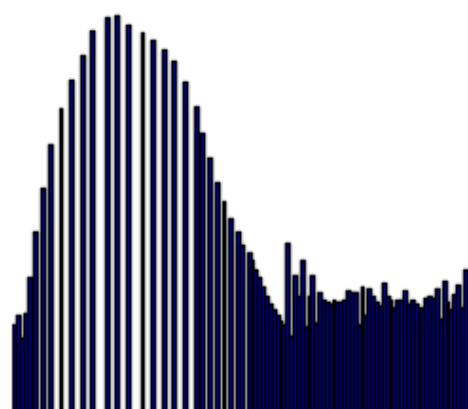
直方图均衡化后的图像



原始直方图



均衡化后的直方图



1.3.4 图像平均

对图像取平均是一种图像降噪的简单方法，经常用于产生艺术效果。假设所有的图像具有相同的尺寸，我们可以对图像相同位置的像素相加取平均，下面是一个演示对图像取平均的例子：

```
# -*- coding: utf-8 -*-
from PCV.tools.imtools import get_imlist
from PIL import Image
from pylab import *
from PCV.tools import imtools

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

filelist = get_imlist('..../data/avg/') #获取convert_images_format_test文件夹下的图片文件名(包括后缀名)
avg = imtools.compute_average(filelist)

for impath in filelist:
    im1 = array(Image.open(impath))
    subplot(2, 2, filelist.index(impath)+1)
    imshow(im1)
    imNum=str(filelist.index(impath)+1)
    title(u'待平均图像'+imNum, fontproperties=font)
    axis('off')
subplot(2, 2, 4)
imshow(avg)
title(u'平均后的图像', fontproperties=font)
axis('off')

show()
```

运行上面代码，可得对3幅图像平均后的效果，如下图：

待平均图像1



待平均图像2



待平均图像3



平均后的图像



1.3.5 对图像进行主成分分析

主成分分析是一项有用的降维技术。对于主成分分析的原理，这里不做具体介绍。下面我们在字体图像上进行降维处理。文件fontimages.zip包含有字母 "a" 的缩略图，共有 2359 个字体图像，可以在 [\[Images courtesy of Martin Solli\]\(http://webstaff.itn.liu.se/%7Emarso/\)](#) 下载。下面代码是显示原书 P14 页对字体图像进行主成分分析的实例代码：

```

# -*- coding: utf-8 -*-
import pickle
from PIL import Image
from numpy import *
from pylab import *
from PCV.tools import imtools, pca

# Uses sparse pca codepath.
imlist = imtools.get_imlist('../data/selectedfontimages/a_selected_thumbs')

# 获取图像列表和他们的尺寸
imlist = imtools.get_imlist('../data/fontimages/a_thumbs') # fontimages.zip is part of the book
data set
im = array(Image.open(imlist[0])) # open one image to get the size
m, n = im.shape[:2] # get the size of the images
imnbr = len(imlist) # get the number of images
print "The number of images is %d" % imnbr

# Create matrix to store all flattened images
immatrix = array([array(Image.open(imname)).flatten() for imname in imlist], 'f')

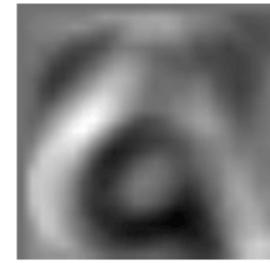
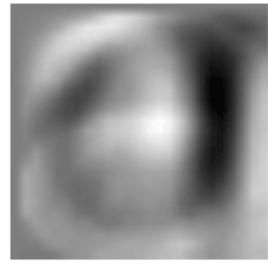
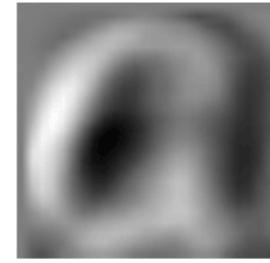
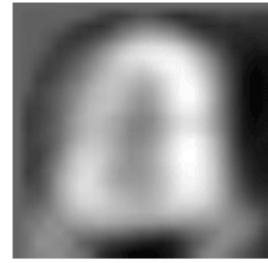
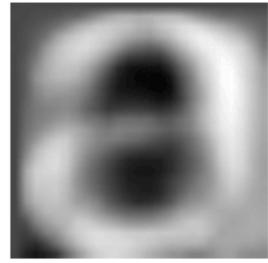
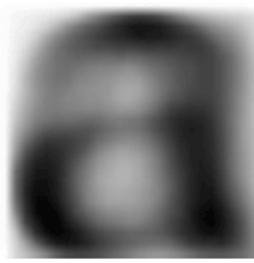
# PCA降维
V, S, immean = pca.pca(immatrix)

# 保存均值和主成分
#f = open('../ch01/font_pca_modes.pkl', 'wb')
#pickle.dump(immean,f)
#pickle.dump(V,f)
#f.close()

# Show the images (mean and 7 first modes)
# This gives figure 1-8 (p15) in the book.
figure()
gray()
subplot(2, 4, 1)
axis('off')
imshow(immean.reshape(m, n))
for i in range(7):
    subplot(2, 4, i+2)
    imshow(V[i].reshape(m, n))
    axis('off')
show()

```

注意，这些图像在拉成一维表示后，必须用`reshape()`函数将它重新转换回来。运行上面代码，可得原书P15 Figure1-8中的结果，即：



1.3.6 Pickle模块

如果你想将结果保存下来，或者将数据保存下来以便后面使用，那么pickle模块是非常有用的。Pickle模块能够获取几乎所有的Python对象，并将它转换成字符串表示，该过程称为封装；从字符串表示重构对象的过程为拆封。下面用一个例子对其进行说明。正如你在上面注释部分看到的一样，假设我们想将前一节字体图像的平均值和主成分保存起来，可以通过下面操作：

```
f = open('../data/fontimages/font_pca_modes.pkl', 'wb')
pickle.dump(immean,f)
pickle.dump(V,f)
f.close()
```

上面在使用封装操作前，需要导入pickle模块。如果要载入保存的.pkl数据，可以通过load()方法，如下：

```
# load mean and principal components
f = open('../data/fontimages/font_pca_modes.pkl', 'rb')
immean = pickle.load(f)
V = pickle.load(f)
f.close()
```

使用with()方法在这里不介绍，具体的可以翻阅原书。关于pickle模块的更多细节可以查阅在线文档
[[docs.python.org/library/pickle.html]](<http://docs.python.org/library/pickle.html>)

1.4 SciPy模块

SciPy (<http://scipy.org/>)是一个开源的数学工具包，它是建立在NumPy的基础上的。它提供了很多有效的常规操作，包括数值综合、最优化、统计、信号处理以及图像处理。正如接下来所展示的，SciPy库包含了很多有用的模块。SciPy库可以在[\[http://scipy.org/Download\]](http://scipy.org/Download) (scipy.org/Download) 下载。

1.4.1 图像模糊

一个经典的并且十分有用的图像卷积例子是对图像进行高斯模糊。高斯模糊可以用于定义图像尺度、计算兴趣点以及很多其他的应用场合。下面是对图像进行模糊显示原书P017 Fig1-9的例子。

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from scipy.ndimage import filters

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

#im = array(Image.open('board.jpeg'))
im = array(Image.open('../data/empire.jpg').convert('L'))

figure()
gray()
axis('off')
subplot(1, 4, 1)
axis('off')
title(u'原图', fontproperties=font)
imshow(im)

for bi, blur in enumerate([2, 5, 10]):
    im2 = zeros(im.shape)
    im2 = filters.gaussian_filter(im, blur)
    im2 = np.uint8(im2)
    imNum=str(blur)
    subplot(1, 4, 2 + bi)
    axis('off')
    title(u'标准差为'+imNum, fontproperties=font)
    imshow(im2)

#如果是彩色图像，则分别对三个通道进行模糊
#for bi, blur in enumerate([2, 5, 10]):
#    im2 = zeros(im.shape)
#    for i in range(3):
#        im2[:, :, i] = filters.gaussian_filter(im[:, :, i], blur)
#    im2 = np.uint8(im2)
#    subplot(1, 4, 2 + bi)
#    axis('off')
#    imshow(im2)

show()
```

运行上面代码，可得P017 Fig1-9中的结果：



上面第一幅图为待模糊图像，第二幅用高斯标准差为2进行模糊，第三幅用高斯标准差为5进行模糊，最后一幅用高斯标准差为10进行模糊。关于该模块的使用以及参数选择的更多细节，可以参阅SciPy `scipy.ndimage` 文档[docs.scipy.org/doc/scipy/reference/ndimage.html] (<http://docs.scipy.org/doc/scipy/reference/ndimage.html>)。

1.4.2 图像差分

图像强度的改变是一个重要的信息，被广泛用以很多应用中，正如它贯穿于本书中。下面是对图像进行差分显示原书P019 Fig1-10的例子。

```

# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from scipy.ndimage import filters
import numpy

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

im = array(Image.open('..../data/empire.jpg').convert('L'))
gray()

subplot(1, 4, 1)
axis('off')
title(u'(a)原图', fontproperties=font)
imshow(im)

# Sobel derivative filters
imx = zeros(im.shape)
filters.sobel(im, 1, imx)
subplot(1, 4, 2)
axis('off')
title(u'(b)x方向差分', fontproperties=font)
imshow(imx)

imy = zeros(im.shape)
filters.sobel(im, 0, imy)
subplot(1, 4, 3)
axis('off')
title(u'(c)y方向差分', fontproperties=font)
imshow(imy)

#mag = numpy.sqrt(imx**2 + imy**2)
mag = 255-numpy.sqrt(imx**2 + imy**2)
subplot(1, 4, 4)
title(u'(d)梯度幅度', fontproperties=font)
axis('off')
imshow(mag)

show()

```

运行上面代码，可得P019 Fig1-10中的运行结果：

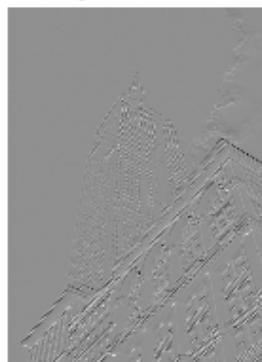
(a) 原图



(b) x方向差分



(c) y方向差分



(d) 梯度幅度



再看一个高斯差分的例子，运行下面代码可得原书P020 Fig1-11页对图像进行高斯差分示例：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from scipy.ndimage import filters
import numpy

# 添加中文字体支持
#from matplotlib.font_manager import FontProperties
#font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

def imx(im, sigma):
    imgx = zeros(im.shape)
    filters.gaussian_filter(im, sigma, (0, 1), imgx)
    return imgx

def imy(im, sigma):
    imgy = zeros(im.shape)
    filters.gaussian_filter(im, sigma, (1, 0), imgy)
    return imgy

def mag(im, sigma):
    # there's also gaussian_gradient_magnitude()
    #mag = numpy.sqrt(imgx**2 + imgy**2)
    imgmag = 255 - numpy.sqrt(imgx ** 2 + imgy ** 2)
    return imgmag

im = array(Image.open('../data/empire.jpg').convert('L'))
figure()
gray()

sigma = [2, 5, 10]

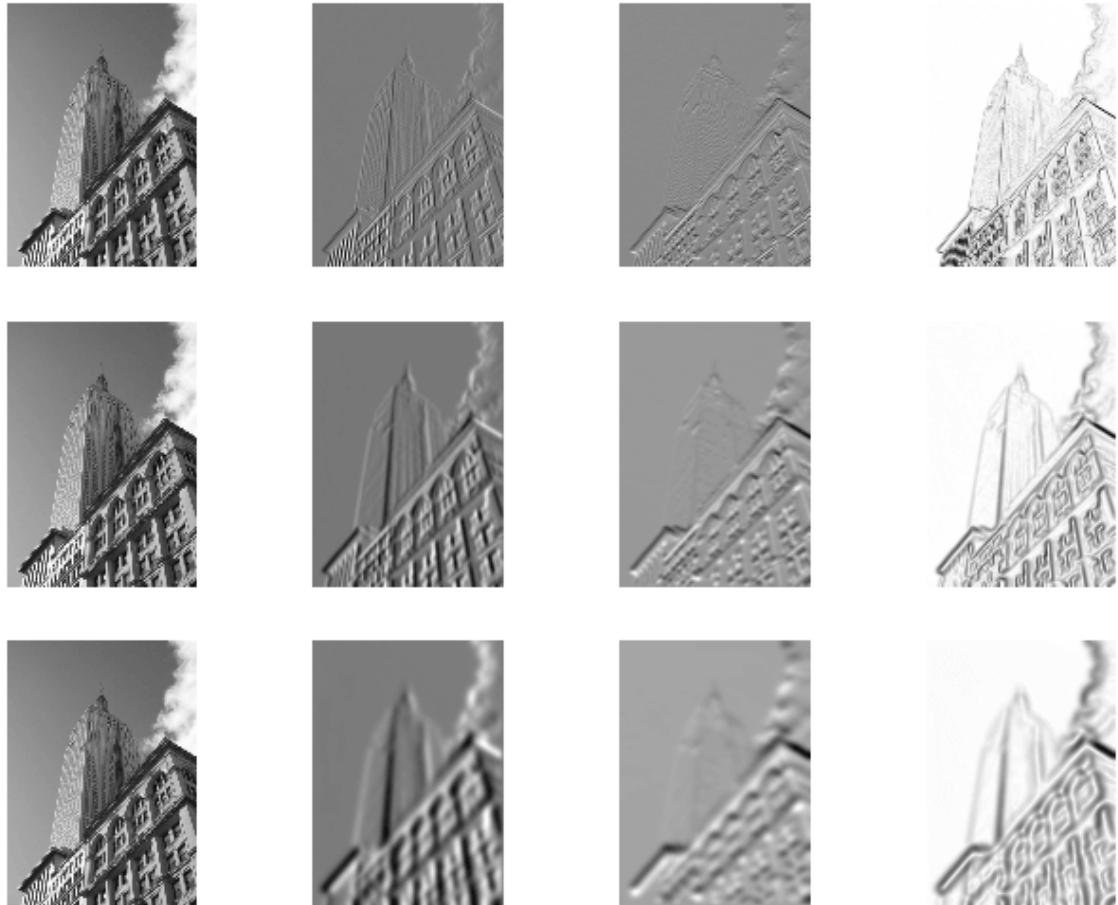
for i in sigma:
    subplot(3, 4, 4*(sigma.index(i))+1)
```

```

axis('off')
imshow(im)
imgx=imx(im, i)
subplot(3, 4, 4*(sigma.index(i))+2)
axis('off')
imshow(imgx)
imgy=imy(im, i)
subplot(3, 4, 4*(sigma.index(i))+3)
axis('off')
imshow(imgy)
imgmag=mag(im, i)
subplot(3, 4, 4*(sigma.index(i))+4)
axis('off')
imshow(imgmag)

show()

```



注意运行的结果在摆放位置时与原书P020 Fig1-11结果稍微不同。上面代码中，第一行标准差为2，列分别表示的是x、y和mag,第二行和第三行依次类推。

1.4.3 形态学-物体计数

形态学常用于二值图像，不过它也可以用于灰度图像。二值图像像素只有两种取值，通常是0和1。二值图像通常是由一幅图像进行二值化处理后的产生的，它可以用于对物体进行计数，或计算它们的大小。对形态学的介绍和较好的介绍是[wiki\[en.wikipedia.org/wiki/Mathematical_morphology\]](http://en.wikipedia.org/wiki/Mathematical_morphology) (http://en.wikipedia.org/wiki/Mathematical_morphology)。

形态学操作包括在sci.ndimage模块morphology中。下面我们看一个简单地怎样使用它们例子。

```

# -*- coding: utf-8 -*-
from PIL import Image
from numpy import *
from scipy.ndimage import measurements, morphology
from pylab import *

""" This is the morphology counting objects example in Section 1.4. """
# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

# Load image and threshold to make sure it is binary
figure()
gray()
im = array(Image.open('../data/houses.png').convert('L'))
subplot(221)
imshow(im)
axis('off')
title(u'原图', fontproperties=font)
im = (im < 128)

labels, nbr_objects = measurements.label(im)
print "Number of objects:", nbr_objects
subplot(222)
imshow(labels)
axis('off')
title(u'标记后的图', fontproperties=font)

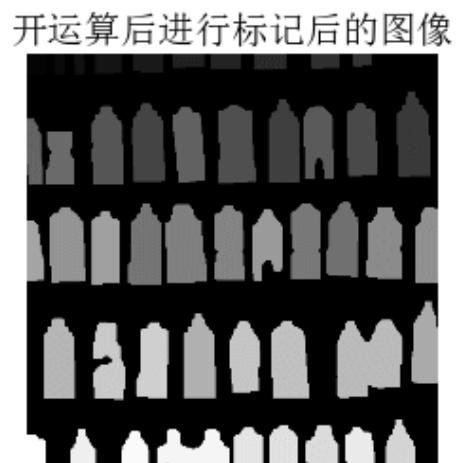
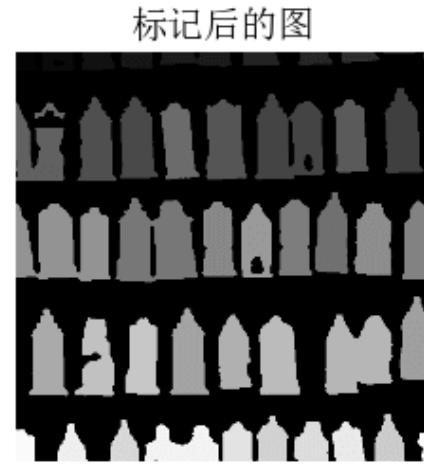
# morphology - opening to separate objects better
im_open = morphology.binary_opening(im, ones((9, 5)), iterations=2)
subplot(223)
imshow(im_open)
axis('off')
title(u'开运算后的图像', fontproperties=font)

labels_open, nbr_objects_open = measurements.label(im_open)
print "Number of objects:", nbr_objects_open
subplot(224)
imshow(labels_open)
axis('off')
title(u'开运算后进行标记后的图像', fontproperties=font)

show()

```

运行上面代码，可得原书P022 Fig1-12的结果：



同时打印计数结果为：

```
Number of objects: 45  
Number of objects: 48
```

更多关于形态学可以参阅scipy.ndimage在线文档[\[docs.scipy.org/doc/scipy/reference/ndimage.html\]](http://docs.scipy.org/doc/scipy/reference/ndimage.html)
[\(<http://docs.scipy.org/doc/scipy/reference/ndimage.html>\)](http://docs.scipy.org/doc/scipy/reference/ndimage.html)。

1.4.4 有用的SciPy模块

SciPy有一些用于输入和输出数据有用的模块，其中两个是io和misc。

读写.mat文件

如果你有一些数据存储在Matlab .mat文件中，可以用scipy.io模块读取：

```
data = scipy.io.loadmat('test.mat')
```

如果要保存到.mat文件中的话，同样也很容易，仅仅只需要创建一个字典，字典中即可保存你想保存的所有变量，然后用savemat()方法即可：

```
#创建字典  
data = {}  
#将变量x保存在字典中  
data['x'] = x  
scipy.io.savemat('test.mat',data)
```

更多关于scipy.io的信息可以参阅在线文档[docs.scipy.org/doc/scipy/reference/io.html] (<http://docs.scipy.org/doc/scipy/reference/io.html>)。

保存数组为图像

在scipy.misc模块中，包含了imsave()函数，要保存数组为一幅图像，可通过下面方式完成：

```
from scipy.misc import imsave
imsave('test.jpg',im)
```

scipy.misc模块中还包含了著名的"Len"测试图像：

```
lena = scipy.misc.lena()
```

上面得到的lena图像是一幅512*512大小的灰度图像。

1.5 更高级的例子：图像降噪

我们以一个非常有用的例子结束本章。图像降噪是一个在尽可能保持图像细节和结构信息时去除噪声的过程。我们采用Rudin-Osher-Fatemi de-noising(ROF)模型。图像去噪可以应用于很多场合，它涵盖了从你的度假照片使之更好看到卫星照片质量提高。

下面我们看一个图像降噪的综合实例：

```
# -*- coding: utf-8 -*-
from pylab import *
from numpy import *
from numpy import random
from scipy.ndimage import filters
from scipy.misc import imsave
from PCV.tools import rof

""" This is the de-noising example using ROF in Section 1.5. """

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

# create synthetic image with noise
im = zeros((500,500))
im[100:400,100:400] = 128
im[200:300,200:300] = 255
im = im + 30*random.standard_normal((500,500))

U,T = rof.denoise(im,im)
G = filters.gaussian_filter(im,10)
```

```

# save the result
#imsave('synth_original.pdf',im)
#imsave('synth_rof.pdf',U)
#imsave('synth_gaussian.pdf',G)

# plot
figure()
gray()

subplot(1,3,1)
imshow(im)
#axis('equal')
axis('off')
title(u'原噪声图像', fontproperties=font)

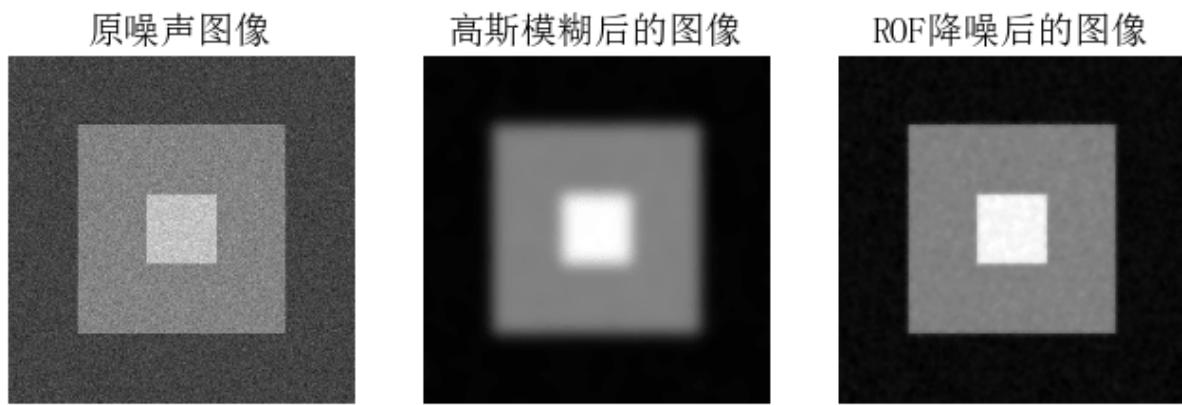
subplot(1,3,2)
imshow(G)
#axis('equal')
axis('off')
title(u'高斯模糊后的图像', fontproperties=font)

subplot(1,3,3)
imshow(U)
#axis('equal')
axis('off')
title(u'ROF降噪后的图像', fontproperties=font)

show()

```

运行上面代码，可得到原书P025 Fig1-13的结果，如下图示：



其中第一幅图示原噪声图像，中间一幅图示用标准差为10进行高斯模糊后的结果，最右边一幅图是用ROF降噪后的图像。上面原噪声图像是模拟出来的图像，现在我们在真实的图像上进行测试：

```

# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from numpy import *
from numpy import random

```

```

from scipy.ndimage import filters
from scipy.misc import imsave
from PCV.tools import rof

""" This is the de-noising example using ROF in Section 1.5. """
# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

im = array(Image.open('../data/empire.jpg').convert('L'))

U,T = rof.denoise(im,im)
G = filters.gaussian_filter(im,10)

# save the result
#imsave('synth_original.pdf',im)
#imsave('synth_rof.pdf',U)
#imsave('synth_gaussian.pdf',G)

# plot
figure()
gray()

subplot(1,3,1)
imshow(im)
#axis('equal')
axis('off')
title(u'原噪声图像', fontproperties=font)

subplot(1,3,2)
imshow(G)
#axis('equal')
axis('off')
title(u'高斯模糊后的图像', fontproperties=font)

subplot(1,3,3)
imshow(U)
#axis('equal')
axis('off')
title(u'ROF降噪后的图像', fontproperties=font)

show()

```

同样，运行上面代码，可得原书P026 Fig1-14的结果，结果如下：

原噪声图像



高斯模糊后的图像



ROF降噪后的图像



正如你所看到的，在去除噪声的同时，ROF降噪能够保持边缘和图像结构。

[« 安装 \(installation.html\)](#)

[第二章 图像局部描述符 » \(chapter2.html\)](#)

第二章 图像局部描述符

2.1 Harris角点检测

2.1.2 在图像间寻找对应点

2.2 sift描述子

2.2.1 兴趣点

2.2.2 描述子

2.2.3 检测感兴趣点

2.2.4 描述子匹配

2.3 地理标记图像匹配

2.3.1 从Panoramio下载地理标记图像

2.3.2 用局部描述子进行匹配

2.3.3 可视化接连的图片

这一章主要介绍两种非常典型的、不同的图像描述子，这两种图像描述子的使用将贯穿于本书，并且作为重要的局部特征，它们应用到了很多应用领域，比如创建全景图、增强现实、3维重建等。

2.1 Harris角点检测

Harris角点检测算法是最简单的角点检测方法之一。关于harris算法的原理，可以参阅本书中译本。下面是harris角点检测实例代码。

```

# -*- coding: utf-8 -*-
from pylab import *
from PIL import Image
from PCV.localdescriptors import harris

"""
Example of detecting Harris corner points (Figure 2-1 in the book).
"""

# 读入图像
im = array(Image.open('../data/empire.jpg').convert('L'))

# 检测harris角点
harrisim = harris.compute_harris_response(im)

# Harris响应函数
harrisim1 = 255 - harrisim

figure()
gray()

#画出Harris 响应图
subplot(141)
imshow(harrisim1)
print harrisim1.shape
axis('off')
axis('equal')

threshold = [0.01, 0.05, 0.1]
for i, thres in enumerate(threshold):
    filtered_coords = harris.get_harris_points(harrisim, 6, thres)
    subplot(1, 4, i+2)
    imshow(im)
    print im.shape
    plot([p[1] for p in filtered_coords], [p[0] for p in filtered_coords], '*')
    axis('off')

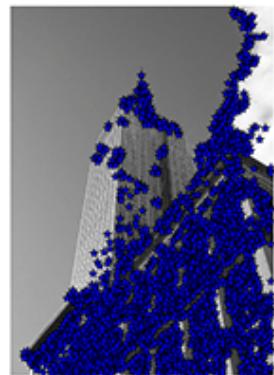
#原书采用的PCV中PCV harris模块
#harris.plot_harris_points(im, filtered_coords)

# plot only 200 strongest
# harris.plot_harris_points(im, filtered_coords[:200])

```

```
show()
```

运行上面代码，可得原书P32页的图：



在上面代码中，先读开一幅图像，将其转换成灰度图像，然后计算相响应函数，通过响应值选择角点。最后，将这些检测的角点在原图上显示出来。如果你想对角点检测方法做一个概览，包括想对Harris检测器做些提高或改进，可以参阅WIKI中的例子WIKI (http://en.wikipedia.org/wiki/Corner_detection)。

2.1.2 在图像间寻找对应点

Harris角点检测器可以给出图像中检测到兴趣点，但它并没有提供在图像间对兴趣点进行比较的方法，我们需要在每个角点添加描述子，以及对这些描述子进行比较。关于兴趣点描述子，见本书中译本。下面再现原书P35页中的结果：

```

# -*- coding: utf-8 -*-
from pylab import *
from PIL import Image

from PCV.localdescriptors import harris
from PCV.tools.imtools import imresize

"""

This is the Harris point matching example in Figure 2-2.

"""

# Figure 2-2上面的图
#im1 = array(Image.open("../data/crans_1_small.jpg").convert("L"))
#im2= array(Image.open("../data/crans_2_small.jpg").convert("L"))

# Figure 2-2下面的图
im1 = array(Image.open("../data/sf_view1.jpg").convert("L"))
im2 = array(Image.open("../data/sf_view2.jpg").convert("L"))

# resize加快匹配速度
im1 = imresize(im1, (im1.shape[1]/2, im1.shape[0]/2))
im2 = imresize(im2, (im2.shape[1]/2, im2.shape[0]/2))

wid = 5
harrisim = harris.compute_harris_response(im1, 5)
filtered_coords1 = harris.get_harris_points(harrisim, wid+1)
d1 = harris.get_descriptors(im1, filtered_coords1, wid)

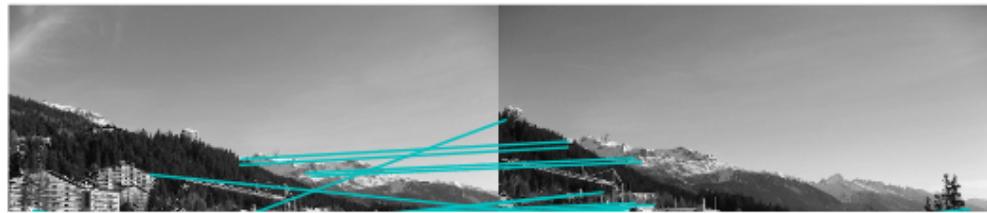
harrisim = harris.compute_harris_response(im2, 5)
filtered_coords2 = harris.get_harris_points(harrisim, wid+1)
d2 = harris.get_descriptors(im2, filtered_coords2, wid)

print 'starting matching'
matches = harris.match_twosided(d1, d2)

figure()
gray()
harris.plot_matches(im1, im2, filtered_coords1, filtered_coords2, matches)
show()

```

运行上面代码，可得下图：



正如你从上图所看到的，这里有很多错配的。近年来，提高特征描述点检测与描述有了很大的发展，在下一节我们会看这其中最好的算法之一——SIFT。

2.2 sift描述子

在过去的十年间，最成功的图像局部描述子之一是尺度不变特征变换(SIFT)，它是由David Lowe发明的。SIFT在2004年由Lowe完善并经受住了时间的考验。关于SIFT原理的详细介绍，可以参阅中译本，在WIKI (http://en.wikipedia.org/wiki/Scale-invariant_feature_transform)上你可以看一个简要的概览。

2.2.1 兴趣点

2.2.2 描述子

2.2.3 检测感兴趣点

为了计算图像的SIFT特征，我们用开源工具包VLFeat。用Python重新实现SIFT特征提取的全过程不会很高效，而且也超出了本书的范围。VLFeat可以在www.vlfeat.org/上下载，它的二进制文件可以用于一些主要的平台。这个库是用C写的，不过我们可以利用它的命令行接口。此外，它还有Matlab接口。下面代码是再现原书P40页的代码：

```
# -*- coding: utf-8 -*-
from PIL import Image
from pylab import *
from PCV.localdescriptors import sift
from PCV.localdescriptors import harris

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

imname = '../data/empire.jpg'
im = array(Image.open(imname).convert('L'))
sift.process_image(imname, 'empire.sift')
l1, d1 = sift.read_features_from_file('empire.sift')

figure()
gray()
subplot(131)
sift.plot_features(im, l1, circle=False)
title(u'SIFT特征', fontproperties=font)
subplot(132)
sift.plot_features(im, l1, circle=True)
title(u'用圆圈表示SIFT特征尺度', fontproperties=font)

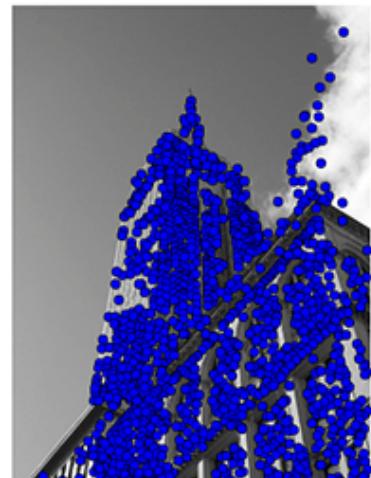
# 检测harris角点
harrisim = harris.compute_harris_response(im)

subplot(133)
filtered_coords = harris.get_harris_points(harrisim, 6, 0.1)
imshow(im)
plot([p[1] for p in filtered_coords], [p[0] for p in filtered_coords], '*')
axis('off')
title(u'Harris角点', fontproperties=font)

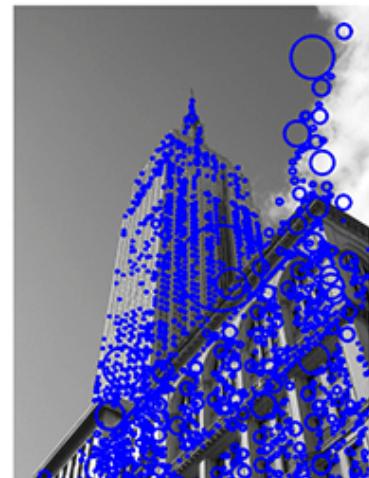
show()
```

运行上面代码，可得下图：

SIFT特征



用圆圈表示SIFT特征尺度



Harris角点



为了将sift和Harris角点进行比较，将Harris角点检测的显示在了图像的最后侧。正如你所看到的，这两种算法选择了不同的坐标。

2.2.4 描述子匹配

```
from PIL import Image
from pylab import *
import sys
from PCV.localdescriptors import sift

if len(sys.argv) >= 3:
    im1f, im2f = sys.argv[1], sys.argv[2]
else:
#    im1f = '../data/sf_view1.jpg'
#    im2f = '../data/sf_view2.jpg'
    im1f = '../data/crans_1_small.jpg'
    im2f = '../data/crans_2_small.jpg'
#    im1f = '../data/climbing_1_small.jpg'
#    im2f = '../data/climbing_2_small.jpg'
im1 = array(Image.open(im1f))
im2 = array(Image.open(im2f))

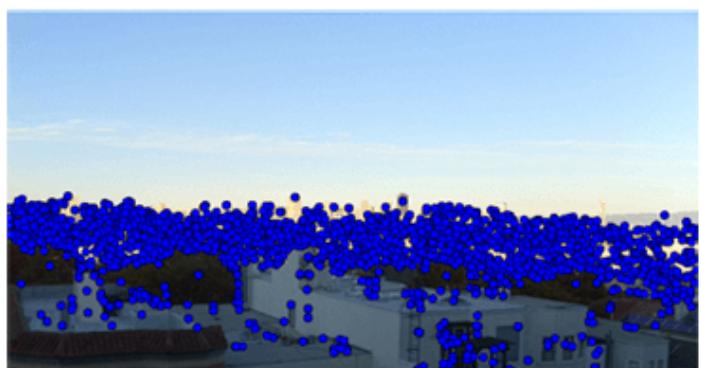
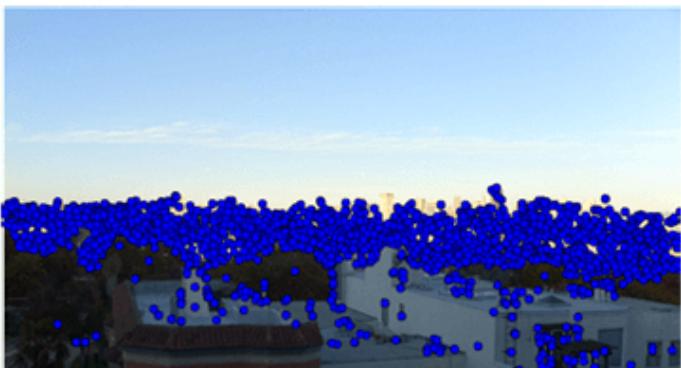
sift.process_image(im1f, 'out_sift_1.txt')
l1, d1 = sift.read_features_from_file('out_sift_1.txt')
figure()
gray()
subplot(121)
sift.plot_features(im1, l1, circle=False)

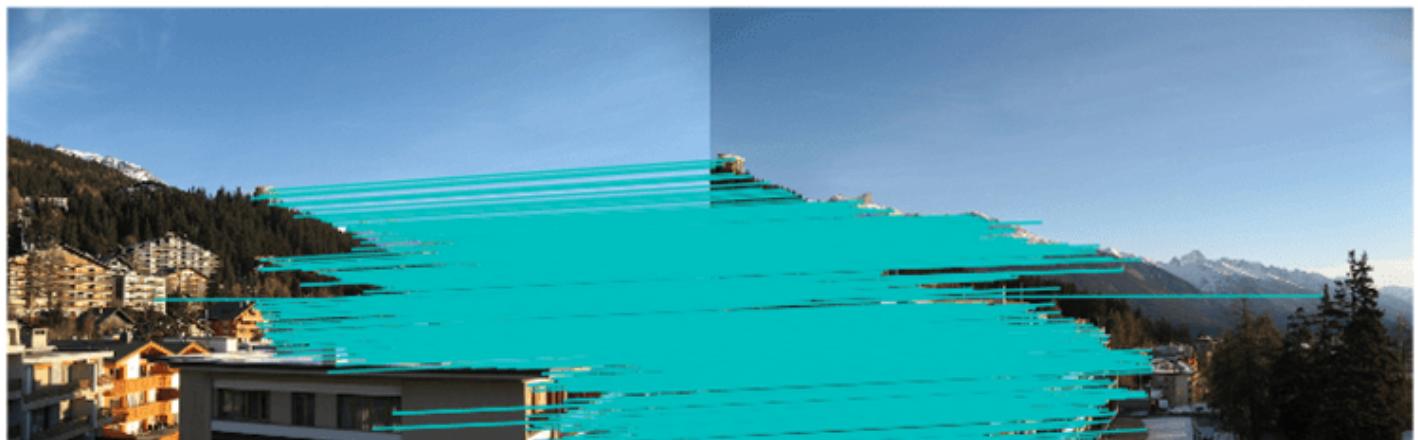
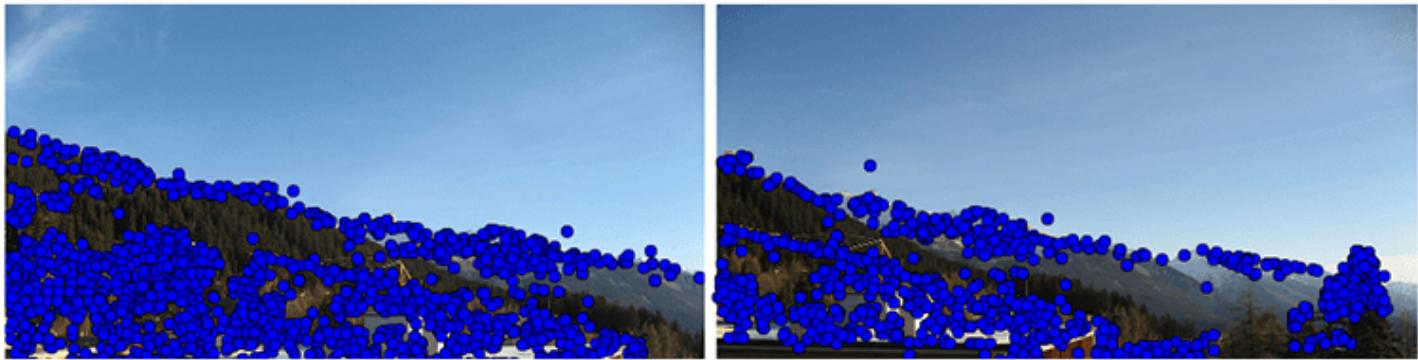
sift.process_image(im2f, 'out_sift_2.txt')
l2, d2 = sift.read_features_from_file('out_sift_2.txt')
subplot(122)
sift.plot_features(im2, l2, circle=False)

#matches = sift.match(d1, d2)
matches = sift.match_twosided(d1, d2)
print '{} matches'.format(len(matches.nonzero()[0]))

figure()
gray()
sift.plot_matches(im1, im2, l1, l2, matches, show_below=True)
show()
```

运行上面代码，可得下图：





2.3 地理标记图像匹配

在结束本章前，我们看一个用局部描述子对地理标记图像进行匹配的例子。

2.3.1 从Panoramio下载地理标记图像

利用谷歌的图片分享服务Panoramio (<http://www.panoramio.com/>)，可以下载地理标记图像。像很多其他的web服务一样，Panoramio提供了API接口，通过提交HTTP GET请求url：

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&size=medium
```

上面minx、miny、maxx、maxy定义了获取照片的地理区域。下面代码是获取白宫地理区域的照片实例：

```
# -*- coding: utf-8 -*-
import json
import os
import urllib
import urlparse
from PCV.tools.imtools import get_imlist
from pylab import *
from PIL import Image

#change the Longitude and Latitude here
#here is the Longitude and Latitude for Oriental Pearl
minx = '-77.037564'
maxx = '-77.035564'
miny = '38.896662'
maxy = '38.898662'

#number of photos
numfrom = '0'
numto = '20'
url = 'http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&from=' +
numfrom + '&to=' + numto + '&minx=' + minx + '&miny=' + miny + '&maxx=' + maxx + '&maxy=' + maxy +
'&size=medium'

#this is the url configured for downloading whitehouse photos. Uncomment this, run and see.
#url = 'http://www.panoramio.com/map/get_panoramas.php?order=popularity&
#set=public&from=0&to=20&minx=-77.037564&miny=38.896662&
#maxx=-77.035564&maxy=38.898662&size=medium'

c = urllib.urlopen(url)

j = json.loads(c.read())
imurls = []
for im in j['photos']:
    imurls.append(im['photo_file_url'])

for url in imurls:
    image = urllib.URLopener()
    image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
    print 'downloading:', url

#显示下载到的20幅图像
figure()
gray()
```

```
filelist = get_imlist('./')
for i, imlist in enumerate(filelist):
    im=Image.open(imlist)
    subplot(4,5,i+1)
    imshow(im)
    axis('off')
show()
```

译者稍微修改了原书的代码，上面numto是设置下载照片的数目。运行上面代码可在脚本所在的目录下得到下载到的20张图片，代码后面部分为译者所加，用于显示下载到的20幅图像：现在我们便可以用这些图片利用局部特征对其进行匹配了。

2.3.2 用局部描述子进行匹配

在下载完上面的图片后，我们便可提取他们的描述子。这里，我们用前面用到的SIFT描述子。

```

# -*- coding: utf-8 -*-
from pylab import *
from PIL import Image
from PCV.localdescriptors import sift
from PCV.tools import imtools
import pydot

""" This is the example graph illustration of matching images from Figure 2-10.
To download the images, see ch2_download_panoramio.py."""

#download_path = "panoimages" # set this to the path where you downloaded the panoramio images
#path = "/FULLPATH/panoimages/" # path to save thumbnails (pydot needs the full system path)

download_path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages" # set this to the
path where you downloaded the panoramio images
path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages/" # path to save thumbnails
(pydot needs the full system path)

# List of downloaded filenames
imlist = imtools.get_imlist(download_path)
nbr_images = len(imlist)

# extract features
featlist = [imname[:-3] + 'sift' for imname in imlist]
for i, imname in enumerate(imlist):
    sift.process_image(imname, featlist[i])

matchscores = zeros((nbr_images, nbr_images))

for i in range(nbr_images):
    for j in range(i, nbr_images): # only compute upper triangle
        print 'comparing ', imlist[i], imlist[j]
        l1, d1 = sift.read_features_from_file(featlist[i])
        l2, d2 = sift.read_features_from_file(featlist[j])
        matches = sift.match_twosided(d1, d2)
        nbr_matches = sum(matches > 0)
        print 'number of matches = ', nbr_matches
        matchscores[i, j] = nbr_matches
print "The match scores is: \n", matchscores

# copy values
for i in range(nbr_images):
    for j in range(i + 1, nbr_images): # no need to copy diagonal
        matchscores[j, i] = matchscores[i, j]

```

上面将两两进行特征匹配后的匹配数保存在matchscores中，最后一部分将矩阵填充完整，它并不是必须的，原因是该“距离度量”矩阵是对称的。运行上面代码，可得到下面的结果：

```

662 0 0 2 0 0 0 1 0 0 1 2 0 3 0 19 1 0 2
0 901 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 2
0 0 266 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
2 1 0 1481 0 0 2 2 0 0 0 2 2 0 0 0 2 3 2 0
0 0 0 0 1748 0 0 1 0 0 0 0 0 2 0 0 0 0 0 1
0 0 0 0 0 1747 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 2 0 0 555 0 0 0 1 4 4 0 2 0 0 5 1 0
0 1 0 2 1 0 0 2206 0 0 0 1 0 0 1 0 2 0 1 1
1 1 0 0 0 1 0 0 629 0 0 0 0 0 0 1 0 0 2 0
0 0 0 0 0 0 0 829 0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 0 1025 0 0 0 0 0 1 1 1 0
1 1 0 2 0 0 4 1 0 0 0 528 5 2 15 0 3 6 0 0
2 0 0 2 0 0 4 0 0 1 0 5 736 1 4 0 3 37 1 0
0 0 1 0 2 0 0 0 0 0 0 2 1 620 1 0 0 1 0 0
3 0 0 0 0 0 2 1 0 0 0 15 4 1 553 0 6 9 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2273 0 1 0 0
19 0 0 2 0 0 0 2 1 0 1 3 3 0 6 0 542 0 0 0
1 0 0 3 0 1 5 0 0 0 1 6 37 1 9 1 0 527 3 0
0 1 0 2 0 1 1 1 0 0 1 0 1 0 1 0 0 3 1139 0
2 2 0 0 1 0 0 1 20 2 0 0 0 0 0 0 0 0 0 499

```

注意：这里译者为排版美观起见，用的是原书运行的结果，上面代码时间运行的结果跟原书得到的结果是有差异的。

2.3.3 可视化接连的图片

这节我们对上面匹配后的图像进行连接可视化，要做到这样，我们需要在一个图中用边线表示它们之间是相连的。我们采用 [pydot](http://code.google.com/p/pydot/) 工具包 (<http://code.google.com/p/pydot/>)，它提供了 GraphViz graphing 库的 Python 接口。不要担心，它们安装起来很容易。

Pydot很容易使用，下面代码演示创建一个图：

```

import pydot

g = pydot.Dot(graph_type='graph')

g.add_node(pydot.Node(str(0), fontcolor='transparent'))
for i in range(5):
    g.add_node(pydot.Node(str(i + 1)))
    g.add_edge(pydot.Edge(str(0), str(i + 1)))
    for j in range(5):
        g.add_node(pydot.Node(str(j + 1) + '0' + str(i + 1)))
        g.add_edge(pydot.Edge(str(j + 1) + '0' + str(i + 1), str(j + 1)))
g.write_png('../images/ch02/ch02_fig2-9_graph.png', prog='neato')

```

运行上面代码，在images/ch02/下生成一幅名字为ch02fig2-9graph的图，如下所示：

现在，我们回到那个地理图像的例子，我们同样将匹配后对其进行可视化。为了是得到的可视化结果比较好看，我们对每幅图像用100*100的缩略图缩放它们。

```

# -*- coding: utf-8 -*-
from pylab import *
from PIL import Image
from PCV.localdescriptors import sift
from PCV.tools import imtools
import pydot

""" This is the example graph illustration of matching images from Figure 2-10.
To download the images, see ch2_download_panoramio.py."""

#download_path = "panoimages" # set this to the path where you downloaded the panoramio images
#path = "/FULLPATH/panoimages/" # path to save thumbnails (pydot needs the full system path)

download_path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages" # set this to the
path where you downloaded the panoramio images
path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages/" # path to save thumbnails
(pydot needs the full system path)

# List of downloaded filenames
imlist = imtools.get_imlist(download_path)
nbr_images = len(imlist)

# extract features
featlist = [imname[:-3] + 'sift' for imname in imlist]
for i, imname in enumerate(imlist):
    sift.process_image(imname, featlist[i])

matchscores = zeros((nbr_images, nbr_images))

for i in range(nbr_images):
    for j in range(i, nbr_images): # only compute upper triangle
        print 'comparing ', imlist[i], imlist[j]
        l1, d1 = sift.read_features_from_file(featlist[i])
        l2, d2 = sift.read_features_from_file(featlist[j])
        matches = sift.match_twosided(d1, d2)
        nbr_matches = sum(matches > 0)
        print 'number of matches = ', nbr_matches
        matchscores[i, j] = nbr_matches
print "The match scores is: \n", matchscores

# copy values
for i in range(nbr_images):
    for j in range(i + 1, nbr_images): # no need to copy diagonal
        matchscores[j, i] = matchscores[i, j]

#可视化

threshold = 2 # min number of matches needed to create link

```

```
g = pydot.Dot(graph_type='graph') # don't want the default directed graph

for i in range(nbr_images):
    for j in range(i + 1, nbr_images):
        if matchscores[i, j] > threshold:
            # first image in pair
            im = Image.open(imlist[i])
            im.thumbnail((100, 100))
            filename = path + str(i) + '.png'
            im.save(filename) # need temporary files of the right size
            g.add_node(pydot.Node(str(i), fontcolor='transparent', shape='rectangle',
image=filename))

            # second image in pair
            im = Image.open(imlist[j])
            im.thumbnail((100, 100))
            filename = path + str(j) + '.png'
            im.save(filename) # need temporary files of the right size
            g.add_node(pydot.Node(str(j), fontcolor='transparent', shape='rectangle',
image=filename))

            g.add_edge(pydot.Edge(str(i), str(j)))
g.write_png('whitehouse.png')
```

运行上面代码，可以得到下面的结果：

正如上图所示，我们可以看到三组图像，前两组是白宫不同的侧面图片。上面这个例子只是一个利用局部描述子进行匹配的很简单的例子，我们并没有对匹配进行核实，在后面两个章节中，我们便可以对其进行核实了。

第三章 图像映射

3.1 图像局部描述子

下面是显示原书图58页的例子：

```
# -*- coding: utf-8 -*-
from scipy import ndimage
from PIL import Image
from pylab import *

im = array(Image.open('../data/empire.jpg').convert('L'))
H = array([[1.4,0.05,-100],[0.05,1.5,-100],[0,0,1]])
im2 = ndimage.affine_transform(im,H[:2,:2],(H[0,2],H[1,2]))

figure()
gray()
subplot(121)
axis('off')
imshow(im)
subplot(122)
axis('off')
imshow(im2)
show()
```



从本章开始我们要开发一个大型的示例程序，本书后续内容都会基于这个示例程序。最终完成的程序会包含用户、微博功能，以及完整的登录和用户身份验证系统，不过我们会从一个看似功能有限的话题出发——创建静态页面。这看似简单的一件事却是一个很好的锻炼，极具意义，对这个初建的程序而言也是个很好的开端。

虽然 Rails 是被设计用来开发基于数据库的动态网站的，不过它也能胜任使用纯 HTML 创建的静态页面。其实，使用 Rails 创建动态页面还有一点好处：我们可以方便的添加一小部分动态内容。这一章就会教你怎么做。在这个过程中我们还会一窥自动化测试（automated testing）的面目，自动化测试可以让我们确信自己编写的代码是正确的。而且，编写一个好的测试用例还可以让我们信心十足的重构（refactor）代码，修改实现过程但不影响最终效果。

本章有很多的代码，特别是在 [3.2 节](#) 和 [3.3 节](#)，如果你是 Ruby 初学者先不用担心没有理解这些代码。就像在 [1.1.1 节](#) ([chapter1.html#sec-1-1-1](#)) 中说过的，你可以直接复制粘贴测试代码，用来验证程序中代码的正确性而不用担心其工作原理。[第四章](#) ([chapter4.html](#)) 会更详细的介绍 Ruby，你有的是机会来理解这些代码。还有 RSpec 测试，它在本书中会被反复使用，如果你现在有点卡住了，我建议你硬着头皮往下看，几章过后你就会惊奇地发现，原本看起来很费解的代码已经变得很容易理解了。

类似第二章，在开始之前我们要先创建一个新的 Rails 项目，这里我们叫它 `sample_app`：

```
$ cd ~/rails_projects  
$ rails new sample_app --skip-test-unit  
$ cd sample_app
```

上面代码中传递给 `rails` 命令的 `--skip-test-unit` 选项的意思是让 Rails 不生成默认使用的 `test::Unit` 测试框架对应的 `test` 文件夹。这样做并不是说我们不用写测试，而是从 [3.2 节](#) 开始我们会使用另一个测试框架 RSpec 来写整个的测试用例。

类似 2.1 节 ([chapter2.html#sec-2-1](#))，接下来我们要用文本编辑器打开并编辑 `Gemfile`，写入程序所需的 gem。这个示例程序会用到之前没用过的两个 gem：RSpec 所需的 gem 和针对 Rails 的 RSpec 库 gem。代码 3.1 所示的代码会包含这些 gem。（注意：如果此时你想安装这个示例程序用到的所有 gem，你应该使用代码 9.49 中的代码。）

代码 3.1 示例程序的 `Gemfile`

```
source 'https://rubygems.org'

gem 'rails', '3.2.13'

group :development, :test do
  gem 'sqlite3', '1.3.5'
  gem 'rspec-rails', '2.11.0'
end

# Gems used only for assets and not required
# in production environments by default.

group :assets do
  gem 'sass-rails',    '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier',     '1.2.3'
end

gem 'jquery-rails', '2.0.2'

group :test do
  gem 'capybara', '1.1.2'
end

group :production do
  gem 'pg', '0.12.2'
end
```

上面的代码将 `rspec-rails` 放在了开发组中，这样我们就可以使用 RSpec 相关的生成器了，同样我们还把它放到了测试组中，这样才能在测试时使用。我们没必要单独的安装 RSpec，因为它是 `rspec-rails` 的依赖件（dependency），会被自动安装。我们还加入了 [Capybara](#) (<https://github.com/jnicklas/capybara>)，这个 gem 允许我们使用类似英语中的句法编写模拟与应用程序交互的代码。¹ 和第二章 ([chapter2.html](#)) 一样，我们还要把 PostgreSQL 所需的 gem 加入生产组，这样才能部署到 Heroku：

```
group :production do
  gem 'pg', '0.12.2'
end
```

Heroku 推荐在开发环境和生产环境使用相同的数据库，不过对我们的示例程序而言没什么影响，SQLite 比 PostgreSQL 更容易安装和配置。在你的电脑中安装和配置 PostgreSQL 会作为一个练习。（参见 [3.5 节](#)）

要安装和包含这些新加的 gem，请运行 `bundle install`：

```
$ bundle install --without production
```

和第二章一样，我们使用 `-without production` 禁止安装生产环境所需的 gem。这个选项会被记住，所以后续调用 Bundler 就不用再指定这个选项，直接运行 `bundle install` 就可以了。²

接着我们要设置一下让 Rails 使用 RSpec 而不用 `test::Unit`。这个设置可以通过 `rails generate rspec:install` 命令实现：

```
$ rails generate rspec:install
```

如果系统提示缺少 JavaScript 运行时，你可以访问 [execjs 在 GitHub 的页面](https://github.com/sstephenson/execjs) (<https://github.com/sstephenson/execjs>) 查看可以使用的运行时。我一般都建议安装 [Node.js](http://nodejs.org/) (<http://nodejs.org/>)。

然后剩下的就是初始化 Git 仓库了：³

```
$ git init  
$ git add .  
$ git commit -m "Initial commit"
```

和第一个程序一样，我建议你更新一下 `README` 文件，更好的描述这个程序，还可以提供一些帮助信息，可参照代码 3.2。

代码 3.2 示例程序改善后的 `README` 文件

```
# Ruby on Rails Tutorial: sample application  
  
This is the sample application for  
[*Ruby on Rails Tutorial: Learn Rails by Example*] (http://railstutorial.org/)  
by [Michael Hartl] (http://michaelhartl.com/).
```

然后添加 `.md` 后缀将其更改为 Markdown 格式，再提交所做的修改：

```
$ git mv README.rdoc README.md  
$ git commit -a -m "Improve the README"
```



图 3.1：为示例程序在 GitHub 新建一个仓库

这个程序在本书的后续章节会一直使用，所以建议你在 GitHub 新建一个仓库（如图 3.1），然后将代码推送上去：

```
$ git remote add origin git@github.com:<username>/sample_app.git  
$ git push -u origin master
```

我自己也做了这一步，你可以在 GitHub 上找到这个示例程序的代码 (https://github.com/railstutorial/sample_app_2nd_ed)。（我用了一个稍微不同的名字）⁴

当然我们也可以选择在这个早期阶段将程序部署到 Heroku：

```
$ heroku create --stack cedar  
$ git push heroku master
```

在阅读本书的过程中，我建议你经常地推送并部署这个程序：

```
$ git push  
$ git push heroku
```

这样你可在远端做个备份，也可以尽早的获知生成环境中出现的错误。如果你在 Heroku 遇到了问题，可以看一下生产环境的日志文件尝试解决：

```
$ heroku logs
```

所有的准备工作都结束了，下面要开始开发这个示例程序了。

3.1 静态页面

Rails 中有两种方式创建静态页面。其一，Rails 可以处理真正只包含 HTML 代码的静态页面。其二，Rails 允许我们定义包含纯 HTML 的视图，Rails 会对其进行渲染，然后 Web 服务器会将结果返回浏览器。

现在回想一下 [1.2.3 节 \(chapter1.html#sec-1-2-3\)](#) 中讲过的 Rails 目录结构（图 1.2）会对我们有点帮助。本节主要的工作都在 app/controllers 和 app/views 文件夹中。（[3.2 节](#)中我们还会新建一个文件夹）

本节你会第一次发现在文本编辑器或 IDE 中打开整个 Rails 目录是多么有用。不过怎么做却取决于你的系统，大多数情况下你可以在命令行中用你选择的浏览器命令打开当前应用程序所在的目录，在 Unix 中当前目录就是一个点号（.）：

```
$ cd ~/rails_projects/sample_app  
$ <editor name> .
```

例如，用 Sublime Text 打开示例程序，你可以输入：

```
$ subl .
```

对于 Vim 来说，针对你使用的不同变种，你可以输入 vim ..、gvim . 或 mvim .。

3.1.1 真正的静态页面

我们先来看一下真正静态的页面。回想一下 [1.2.5 节 \(chapter1.html#sec-1-2-5\)](#)，每个 Rails 应用程序执行过 rails 命令后都会生成一个小型的可以运行的程序，默认的欢迎页面地址是 <http://localhost:3000/>（图 1.3）。



图 3.2: public/index.html 文件

如果想知道这个页面是怎么来的，请看一下 public/index.html 文件（如图 3.2）。因为文件中包含了一些样式信息，所以看起来有点乱，不过其效果却达到了：默认情况下 Rails 会直接将 public 目录下的文件发送给浏览器。⁵ 对于特殊的 index.html 文件，你不用在 URI 中指定它，因为它是默认显示的文件。如果你想在 URI 中包含这个文件的名字也可以，不过 http://localhost:3000/ 和 http://localhost:3000/index.html 的效果是一样的。

如你所想的，如果你需要的话也可以创建静态的 HTML 文件，并将其放在和 `index.html` 相同的目录 `public` 中。举个例子，我们要创建一个文件显示一个友好的欢迎信息（参见代码 3.3）：⁶

```
$ subl public/hello.html
```

代码 3.3 一个标准的 HTML 文件，包含一个友好的欢迎信息

```
public/hello.html
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

从代码 3.3 中我们可以看到 HTML 文件的标准结构：位于文件开头的文档类型（document type，简称 doctype）声明，告知浏览器我们所用的 HTML 版本（本例使用的是 HTML5）；⁷ `head` 部分：本例包含一个 `title` 标签，其内容是“Greeting”；`body` 部分：本例包含一个 `p`（段落）标签，其内容是“Hello,world!”。（缩进是可选的，HTML 并不强制要求使用空格，它会忽略 Tab 和空格，但是缩进可以使文档的结构更清晰。）

现在执行下述命令启动本地浏览器

```
$ rails server
```

然后访问 <http://localhost:3000/hello.html> (`http://localhost:3000/hello.html`)。就像前面说过的，Rails 会直接渲染这个页面（如图 3.3）。注意图 3.3 浏览器窗口顶部显示的标题，它就是 `title` 标签的内容，“Greeting”。



图 3.3：一个新的静态 HTML 文件

这个文件只是用来做演示的，我们的示例程序并不需要它，所以在体验了创建过程之后最好将其删掉：

```
$ rm public/hello.html
```

现在我们还要保留 `index.html` 文件，不过最后我们还是要将其删除的，因为我们不想把 Rails 默认的页面（如图 1.3）作为程序的首页。[5.3 节 \(chapter5.html#sec-5-3\)](#) 会介绍如何将 <http://localhost:3000/> (<http://localhost:3000/>) 指向 `public/index.html` 之外的地方。

3.1.2 Rails 中的静态页面

能够显示静态 HTML 页面固然很好，不过对动态 Web 程序却没有什么用。本节我们要向创建动态页面迈出第一步，我们会创建一系列的 Rails 动作（action），这可比通过静态文件定义 URI 地址要强大得多。⁸ Rails 的动作会按照一定的目的性归属在某个控制器（[1.2.6 节 \(chapter1.html#sec-1-2-6\)](#) 介绍的 MVC 中的 C）中。在[第二章 \(chapter2.html\)](#) 中已经简单介绍了

控制器，当我们更详细的介绍 REST 架构 (http://en.wikipedia.org/wiki/Representational_State_Transfer) 后（从第六章 ([chapter6.html](#)) 开始）你会更深入的理解它。大体而言，控制器就是一组网页的（也许是动态的）容器。

开始之前，回想一下 [1.3.5 节 \(chapter1.html#sec-1-3-5\)](#) 中的内容，使用 Git 时，在一个有别于主分支的独立从分支中工作是一个好习惯。如果你使用 Git 做版本控制，可以执行下面的命令：

```
$ git checkout -b static-pages
```

Rails 提供了一个脚本用来创建控制器，叫做 `generate`，只要提供控制器的名字就可以运行了。如果想让 `generate` 同时生成 RSpec 测试用例，需要执行 RSpec 生成器命令，如果在阅读本章前面内容时没有执行这个命令的话，请执行下面的命令：

```
$ rails generate rspec:install
```

因为我们要创建一个控制器来处理静态页面，所以我们就叫它 `StaticPages` 吧。我们计划创建“首页”（Home）、“帮助”（Help）和“关于”（About）页面的动作。`generate` 可以接受一个可选的参数列表，指明要创建的动作，我们现在只通过命令行创建两个动作（参见代码 3.4）。

代码 3.4 创建 StaticPages 控制器

```
$ rails generate controller StaticPages home help --no-test-framework
  create  app/controllers/static_pages_controller.rb
  route   get "static_pages/help"
  route   get "static_pages/home"
  invoke  erb
  create    app/views/static_pages
  create    app/views/static_pages/home.html.erb
  create    app/views/static_pages/help.html.erb
  invoke  helper
  create    app/helpers/static_pages_helper.rb
  invoke  assets
  invoke  coffee
  create    app/assets/javascripts/static_pages.js.coffee
  invoke  scss
  create    app/assets/stylesheets/static_pages.css.scss
```

注意，我们使用了 `--no-test-framework` 选项禁止生成 RSpec 测试代码，因为我们不想自动生成，在 [3.2 节](#) 会手动创建测试。同时我们还故意从命令行参数中省去了 `about` 动作，稍后我们会看到如何通过 TDD 添加它（[3.2 节](#)）。

顺便说一下，如果在生成代码时出现了错误，知道如何撤销操作就很有用了。[旁注 3.1](#) 中介绍了一些如何在 Rails 中撤销操作的方法。

旁注 3.1 撤销操作

即使再小心，在开发 Rails 应用程序过程中仍然可能犯错。幸运的是，Rails 提供了一些工具能够帮助你进行复原。

举例来说，一个常见的原因是，你想更改控制器的名字，这时你就要撤销生成的代码。生成控制器时，除了控制器文件本身之外，Rails 还会生成很多其他的文件（参见代码 3.4）。撤销生成的文件不仅仅要删除主要的文件，还要删除一些辅助的文件。（事实上，我们还要撤销对 `routes.rb` 文件自动做的一些改动。）在 Rails 中，我们可以通过 `rails`

`destroy` 命令完成这些操作。一般来说，下面的两个命令是相互抵消的：

```
$ rails generate controller FooBars baz quux  
$ rails destroy controller FooBars baz quux
```

同样的，在第六章 (chapter6.html)中会使用下面的命令生成模型：

```
$ rails generate model Foo bar:string baz:integer
```

生成的模型可通过下面的命令撤销：

```
$ rails destroy model Foo
```

(对模型来说我们可以省略命令行中其余的参数。当阅读到第六章 (chapter6.html)时，看看你能否发现为什么可以这么做。)

对模型来说涉及到的另一个技术是撤销迁移。第二章 (chapter2.html)已经简要的介绍了迁移，第六章 (chapter6.html)开始会更深入的介绍。迁移通过下面的命令改变数据库的状态：

```
$ rake db:migrate
```

我们可以使用下面的命令撤销一个迁移操作：

```
$ rake db:rollback
```

如果要回到最开始的状态，可以使用：

```
$ rake db:migrate VERSION=0
```

你可能已经猜到了，将数字 0 换成其他的数字就会回到相应的版本状态，这些版本数字是按照迁移顺序排序的。

拥有这些技术，我们就可以得心的应对开发过程中遇到的各种混乱 (snafu) (<http://en.wikipedia.org/wiki/SNAFU>)了。

代码 3.4 中生成 `StaticPages` 控制器的命令会自动更新路由文件（route），叫做 `config/routes.rb`，Rails 会通过这个文件寻找 URI 和网页之间的对应关系。这是我们第一次讲到 `config` 目录，所以让我们看一下该目录的结构吧（如图 3.4）。`config` 目录如其名字所示，是存储 Rails 应用程序中的设置文件的。



图 3.4：示例程序的 `config` 文件夹

因为我们生成了 `home` 和 `help` 动作，路由文件中已经为它们生成了配置，如代码 3.5。

代码 3.5 `StaticPages` 控制器中 `home` 和 `help` 动作的路由配置

```
config/routes.rb
```

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  .
  .
  .
end
```

如下的规则

```
get "static_pages/home"
```

将来自 `/staticpages/home` 的请求映射到 `StaticPages` 控制器的 `home` 动作上。另外，当使用 `get` 时会将其对应到 `GET` 请求方法上，`GET` 是 `HTTP`（超文本传输协议，*Hypertext Transfer Protocol*）支持的基本方法之一（参见 [旁注 3.2](#)）。在我们这个例子中，当我们在 `StaticPages` 控制器中生成 `home` 动作时，就自动的在 `/staticpages/home` 地址上获得了一个页面。访问 [/static_pages/home](http://localhost:3000/static_pages/home) (`http://localhost:3000/static_pages/home`) 可以查看这个页面（如图 3.5）。

图 3.5：简陋的“首页”视图 ([/static_pages/home](http://localhost:3000/static_pages/home) (`http://localhost:3000/static_pages/home`))

旁注 3.2 GET 等

超文本传输协议（`HTTP`）定义了四个基本的操作，对应到四个动词上，分别是 `get`、`post`、`put` 和 `delete`。这四个词表现了客户端电脑（通常会运行一个浏览器，例如 `Firefox` 或 `Safari`）和服务器（通常会运行一个 `Web` 服务器，例如 `Apache` 或 `Nginx`）之间的操作。（有一点很重要需要你知道，当在本地电脑上开发 `Rails` 应用程序时，客户端和服务端是在同一个物理设备上的，但是二者是不同的概念。）受 `REST` 架构影响的 `Web` 框架（包括 `Rails`）都很重视对 `HTTP` 动词的实现，我们在 [第二章 \(chapter2.html\)](#) 已经简要介绍了 `REST`，从 [第七章 \(chapter7.html\)](#) 开始会做更详细的介绍。

`GET` 是最常用的 `HTTP` 操作，用来从网络上读取数据，它的意思是“读取一个网页”，当你访问 `google.com` 或 `wikipedia.org` 时，你的浏览器发出的就是 `GET` 请求。`POST` 是第二种最常用的操作，当你提交表单时浏览器发送的就是 `POST` 请求。在 `Rails` 应用程序中，`POST` 请求一般被用来创建某个东西（不过 `HTTP` 也允许 `POST` 进行更新操作）。例如，你提交注册表单时发送的 `POST` 请求就会在网站中创建一个新用户。剩下的两个动词，`PUT` 和 `DELETE` 分别用来更新和销毁服务器上的某个东西。这两个操作比 `GET` 和 `POST` 少用一些，因为浏览器没有内建对这两种请求的支持，不过有些 `Web` 框架（包括 `Rails`）通过一些聪明的处理方式，看起来就像是浏览器发出的一样。

要想弄明白这个页面是怎么来的，让我们在浏览器中看一下 `StaticPages` 控制器文件吧，你应该会看到类似代码 3.6 的内容。你可能已经注意到了，不像第二章中的 `Users` 和 `Microposts` 控制器，`StaticPages` 控制器没有使用标准的 `REST` 动作。这对静态页面来说是很常见的，`REST` 架构并不能解决所有的问题。

代码 3.6 代码 3.4 生成的 `StaticPages` 控制器

```
app/controllers/static_pages_controller.rb
```

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

从上面代码中的 `class` 可以看到 `static_pages_controller.rb` 文件定义了一个类（`class`），叫做 `StaticPagesController`。类是一种组织函数（也叫方法）的有效方式，例如 `home` 和 `help` 动作就是方法，使用 `def` 关键字定义。尖括号 `<` 说明 `StaticPagesController` 是继承自 Rails 的 `ApplicationController` 类，这就意味着我们定义的页面拥有了 Rails 提供的大量功能。（我们会在 [4.4 节 \(chapter4.html#sec-4-4\)](#) 中更详细的介绍类和继承。）

在本例中，`StaticPages` 控制器的两个方法默认都是空的：

```
def home
end

def help
end
```

如果是普通的 Ruby 代码，这两个方法什么也做不了。不过在 Rails 中就不一样了，`StaticPagesController` 是一个 Ruby 类，因为它继承自 `ApplicationController`，它的方法对 Rails 来说就有特殊的意义了：访问 `/staticpages/home` 时，Rails 在 `StaticPages` 控制器中寻找 `home` 动作，然后执行该动作，再渲染相应的视图 ([1.2.6 节 \(chapter1.html#sec-1-2-6\)](#) 中介绍的 MVC 中的 `V`)。在本例中，`home` 动作是空的，所以访问 `/staticpages/home` 后只会渲染视图。那么，视图是什么样子，怎么才能找到它呢？

如果你再看一下代码 3.4 的输出，或许你能猜到动作和视图之间的对应关系：`home` 动作对应的视图叫做 `home.html.erb`。[3.3 节](#) 将告诉你 `.erb` 是什么意思。看到 `.html` 你或许就不会奇怪了，它基本上就是 HTML（代码 3.7）。

代码 3.7 为“首页”生成的视图

`app/views/static_pages/home.html.erb`

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

`help` 动作的视图代码类似（参见代码 3.8）。

代码 3.8 为“帮助”页面生成的视图

`app/views/static_pages/help.html.erb`

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

这两个视图只是占位用的，它们的内容都包含了一个一级标题（`h1` 标签）和一个显示视图文件完整的相对路径的段落（`p` 标签）。我们会在 [3.3 节](#) 中添加一些简单的动态内容。这些静态内容的存在是为了强调一个很重要的事情：Rails 的视图可以只包含静态的 HTML。从浏览器的角度来看，[3.1.1 节](#) 中的原始 HTML 文件和本节通过控制器和动作的方式渲染的页面没

有什么差异，浏览器能看到的只有 HTML。

在本章剩下的内容中，我们会为“首页”和“帮助”页面添加一些内容，然后补上 3.1.2 节中丢下的“关于”页面。然后会添加少量的动态内容，在每个页面显示不同的标题。

在继续下面的内容之前，如果你使用 Git 的话最好将 StaticPages 控制器相关的文件加入仓库：

```
$ git add .
$ git commit -m "Add a StaticPages controller"
```

3.2 第一个测试

本书采用了一种直观的测试应用程序表现的方法，而不关注具体的实现过程，这是 TDD 的一个变种，叫做 BDD（行为驱动开发，Behavior-driven Development）。我们使用的主要工具是集成测试（integration test）和单元测试(unit test)。集成测试在 RSpec 中叫做 request spec，它允许我们模拟用户在浏览器中和应用程序进行交互的操作。和 Capybara 提供的自然语言句法（natural-language syntax）一起使用，集成测试提供了一种强大的方法来测试应用程序的功能，而不用在浏览器中手动检查每个页面。（BDD 另外一个受欢迎的选择是 Cucumber，在 8.3 节 ([chapter8.html#sec-8-3](#)) 中会介绍。）

TDD 的好处在于测试优先，比编写应用程序的代码还早。刚接触的话要花一段时间才能适应这种方式，不过好处很明显。我们先写一个失败测试（failing test），然后编写代码使这个测试通过，这样我们就会相信测试真的是针对我们设想的功能。这种“失败-实现-通过”的开发循环包含了一个心流 ([http://en.wikipedia.org/wiki/Flow_\(psychology\)](http://en.wikipedia.org/wiki/Flow_(psychology)))，可以提高编程的乐趣并提高效率。测试还扮演着应用程序代码客户的角色，会提高软件设计的优雅性。

关于 TDD 有一点很重要需要你知道，它不是万用良药，没必要固执的认为总是要先写测试、测试要囊括程序所有的功能、所有情况都要写测试。例如，当你不确定如何处理某些编程问题时，通常推荐你跳过测试先编写代码看一下解决方法能否解决问题。（在极限编程 (http://en.wikipedia.org/wiki/Extreme_Programming) 中，这个过程叫做“探针实验（spike）”。）一旦看到了解决问题的曙光，你就可以使用 TDD 实现一个更完美的版本。

本节我们会使用 RSpec 提供的 rspec 命令运行测试。初看起来这样做是理所当然的，不过却不完美，如果你是个高级用户我建议你按照 3.6 节的内容设置一下你的系统。

3.2.1 测试驱动开发

在测试驱动开发中，我们先写一个会失败的测试，在很多测试工具中会将其显示为红色。然后编写代码让测试通过，显示为绿色。最后，如果需要的话，我们还会重构代码，改变实现的方式（例如消除代码重复）但不改变功能。这样的开发过程叫做“遇红，变绿，重构（Red, Green, Refactor）”。

我们先来使用 TDD 为“首页”增加一些内容，一个内容为 Sample App 的顶级标题（`<h1>`）。第一步要做的是为这些静态页面生成集成测试（request spec）：

```
$ rails generate integration_test static_pages
invoke rspec
```

第四章 Rails 背后的 Ruby

有了第三章 (chapter3.html)中的例子做铺垫，本章将为你介绍一些对 Rails 来说很重要的 Ruby 知识。Ruby 语言的知识点很多，不过对一个 Rails 开发者而言需要掌握的很少。我们采用的是有别于常规的 Ruby 学习过程，我们的目标是开发动态的 Web 应用程序，所以我建议你先学习 Rails，在这个过程中学习一些 Ruby 知识。如果要成为一个 Rails 专家，你就要更深入的掌握 Ruby 了。本书会为你在成为专家的路途上奠定一个坚实的基础。如 1.1.1 节 (chapter1.html#sec-1-1-1)中说过的，读完本书后我建议你阅读一本专门针对 Ruby 的书，例如《Ruby 入门 (<http://www.amazon.com/gp/product/1430223634>)》、《The Well-Grounded Rubyist (<http://www.amazon.com/gp/product/1933988657>)》或《Ruby 之道 (<http://www.amazon.com/gp/product/0672328844>)》。

本章介绍了很多内容，第一遍阅读没有掌握全部是可以理解的。在后续的章节我会经常提到本章的内容。

4.1 导言

从上一章我们可以看到，即使不懂任何背后用到的 Ruby 语言，我们也可以创建一个 Rails 应用程序骨架，也可以进行测试。不过我们依赖的是本教程中提供的测试代码，得到错误信息，然后让其通过。我们不能总是这样做，所以这一章我们要暂别网站开发学习，正视我们的 Ruby 短肋。

上次接触应用程序时，我们已经使用 Rails 布局去掉了几乎是静态的页面中的代码重复。（参见代码 4.1）

代码 4.1 示例程序的网站布局

`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

让我们把注意力集中在代码 4.1 中的这一行：

```
<%= stylesheet_link_tag "application", :media => "all" %>
```

这行代码使用 Rails 内置的方法 `stylesheet_link_tag` (更多内容请查看 [Rails API 文档](http://api.rubyonrails.org/v3.2.0/classes/ActionView/Helpers/AssetTagHelper/StylesheetTagHelpers.html#method-i-stylesheet_link_tag) (http://api.rubyonrails.org/v3.2.0/classes/ActionView/Helpers/AssetTagHelper/StylesheetTagHelpers.html#method-i-stylesheet_link_tag) 为所有的媒介类型 (<http://www.w3.org/TR/CSS2/media.html>) 引入了 `application.css`。对于经验丰富的 Rails 开发者来说，这一行很简单，但是这里却至少包含了困惑着你的四个 Ruby 知识点：内置的 Rails 方法，不用括号的方法调用，`Symbol` 和 `Hash`。这几点本章都会介绍。

除了提供很多内置的方法供我们在视图中使用之外，Rails 还允许我们自行创建。自行创建的这些方法叫做帮助方法 (`helper`)。要说明如何自行创建一个帮助方法，我们要来看看代码 4.1 中标题那一行：

```
Ruby on Rails Tutorial Sample App | <%= yield(:title) %>
```

这行代码依赖于每个视图中定义的页面标题（使用 `provide`），例如

```
<% provide(:title, 'Home') %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

那么如果我们不提供标题会怎样呢？我们的标题一般都包含一个公共部分，如果想更具体些就要加上一个变动的部分了。我们在布局中用了个小技巧，基本上已经实现了这样的标题。如果我们删除视图中的 `provide` 方法调用，输出的标题就没有了变动的那部分：

```
Ruby on Rails Tutorial Sample App |
```

公共部分已经输出了，而且后面还有一个竖杠 |。

为了解决这个标题问题，我们会自定义一个帮助方法，叫做 `full_title`。如果视图中没有定义标题，`full_title` 会返回标题的公共部分，即“Ruby on Rails Tutorial Sample App”；如果定义了，则会在公共部分后面加上一个竖杠，然后再接上该页面的标题（如代码 4.2）。¹

代码 4.2 定义 `full_title` 帮助方法

`app/helpers/application_helper.rb`

```
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title)
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      "#{base_title} | #{page_title}"
    end
  end
end
```

现在我们已经定义了一个帮助方法，我们可以用它来简化布局，将

```
<title>Ruby on Rails Tutorial Sample App | <%= yield(:title) %></title>
```

替换成

```
<title><%= full_title(yield(:title)) %></title>
```

如代码 4.3 所示。

代码 4.3 示例程序的网站布局

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

为了让这个帮助方法起作用，我们要在“首页”视图中将不必要的“Home”这个词删掉，让标题只保留公共部分。首先我们要按照代码 4.4 的内容更新现有的测试，增加对没包含 ‘Home’ 的标题测试。

代码 4.4 更新“首页”标题的测试

spec/requests/static_pages_spec.rb

```

require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the h1 'Sample App'" do
      visit '/static_pages/home'
      page.should have_selector('h1', :text => 'Sample App')
    end

    it "should have the base title" do
      visit '/static_pages/home'
      page.should have_selector('title',
                                :text => "Ruby on Rails Tutorial Sample App")
    end

    it "should not have a custom page title" do
      visit '/static_pages/home'
      page.should_not have_selector('title', :text => '| Home')
    end
  end

  .
  .
  .

end

```

试试看你能否猜到为什么我们添加了一个新测试而不是直接修改之前的测试。 (提示: 答案在 [3.3.1 节 \(chapter3.html#sec-3-3-1\)](#) 中。)

运行测试, 查看是否有一个测试失败了:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

为了让测试通过, 我们要将“首页”视图中的 `provide` 那行删除, 如代码 4.5 所示。

代码 4.5 删除标题定义后的“首页”

`app/views/static_pages/home.html.erb`

```

<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>

```

现在测试应该可以通过了:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

和引入应用程序样式表那行代码一样，代码 4.2 的内容对经验丰富的 Rails 开发者来说看起来很简单，但是充满了很多会让人困惑的 Ruby 知识：module，注释，局部变量的赋值，布尔值，流程控制，字符串插值，还有返回值。这章也会介绍这些知识。

4.2 字符串和方法

学习 Ruby 我们主要使用的工具是 Rails 控制台，它是用来和 Rails 应用程序交互的命令行，在 [2.3.3 节 \(chapter2.html#sec-2-3-3\)](#) 中介绍过。这个控制台是基于 Ruby 的交互程序（irb）开发的，因此也就能使用 Ruby 语言的全部功能。（在 [4.4.4 节](#) 中会介绍，控制台还可以进入 Rails 环境。）使用下面的方法在命令行中启动控制台：

```
$ rails console
Loading development environment
>>
```

默认情况下，控制台是以开发环境启用的，这是 Rails 定义的三个独立的环境之一（其他两个是测试环境和生产环境）。三个环境的区别在本章还不需要知道，我们会在 [7.1.1 节](#) 中更详细的介绍。

控制台是个很好的学习工具，你不用有所畏惧尽情的使用吧，没必要担心，你（几乎）不会破坏任何东西。如果你在控制器中遇到问题了可以使用 Ctrl-C 结束当前执行的命令，或者使用 Ctrl-D 直接退出控制台。在阅读本章后面的内容时，你会发现查阅 [Ruby API \(http://ruby-doc.org/core-1.9.3/\)](#) 会很有用。API 包含很多信息，例如，如果你想查看关于 Ruby 字符串更多的内容，可以查看其中的 string 类页面。

4.2.1 注释

Ruby 中的注释以井号 #（也叫“Hash Mark”，或者更诗意的叫“散列字元”）开头，一直到行尾结束。Ruby 会忽略注释，但是注释对代码阅读者（包括代码的创作者）却很有用。在下面的代码中

```
# Returns the full title on a per-page basis.
def full_title(page_title)
  .
  .
  .
end
```

第一行就是注释，说明了后面方法的作用。

一般无需在控制台中写注释，不过为了说明代码，我会按照下面的形式加上注释，例如：

```
$ rails console
>> 17 + 42  # Integer addition
=> 59
```

在本节的阅读过程中，在控制台中输入或者复制粘贴命令时，如果愿意你可以不复制注释，反正控制台会忽略注释。

4.2.2 字符串

字符串算是 Web 应用程序中最有用的数据结构了，因为网页的内容就是从数据库发送到浏览器的字符串。我们先在控制台中体验一下字符串，这次我们使用 `rails c` 启动控制台，这是 `rails console` 的简写形式：

```
$ rails c
>> ""          # 空字符串
=> ""
>> "foo"        # 非空的字符串
=> "foo"
```

上面的字符串是字面量（字面量字符串，literal string），通过双引号（"）创建。控制器回显的是每一行的计算结果，本例中字符串字面量的结果就是字符串本身。

我们还可以使用 + 号连接字符串：

```
>> "foo" + "bar"    # 字符串连接
=> "foobar"
```

"foo" 连接 "bar" 的运行结果是字符串 "foobar"。²

另外一种创建字符串的方式是通过一个特殊的句法（#{ }）进行插值操作：³

```
>> first_name = "Michael"      # 变量赋值
=> "Michael"
>> "#{first_name} Hartl"      # 字符串插值
=> "Michael Hartl"
```

我们先把“Michael”赋值给变量 `first_name`，然后将其插入到字符串 "`#{first_name} Hartl`" 中。我们可以将两个字符串都赋值给变量：

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name    # 字符串连接，中间加了空格
=> "Michael Hartl"
>> "#{first_name} #{last_name}"    # 作用相同的插值
=> "Michael Hartl"
```

注意，两个表达式的结果是相同的，不过我倾向使用插值的方式。在两个字符串中加入一个空格 (" ") 显得很别扭。

打印字符串

打印字符串最常用的 Ruby 方法是 `puts`（读作“put ess”，意思是“打印字符串”）：

```
>> puts "foo"      # 打印字符串
foo
=> nil
```

`puts` 方法还有一个副作用 (side-effect)：`puts "foo"` 首先会将字符串打印到屏幕上，然后再返回空值字面量 (<http://www.answers.com/nil>)：`nil` 是 Ruby 中的“什么都没有”。(后续内容中为了行文简洁我会省略 => `nil`。)

`puts` 方法会自动在输出的字符串后面加入换行符 `\n`，功能类似的 `print` 方法则不会：

```
>> print "foo"      # 打印字符串 (和 puts 类似，但没有添加换行符)
foo=> nil
>> print "foo\n"   # 和 puts "foo" 一样
=> nil
```

单引号字符串

目前介绍的例子都是使用双引号创建的字符串，不过 Ruby 也支持用单引号创建字符串。大多数情况下这两种字符串的效果是一样的：

```
>> 'foo'           # 单引号创建的字符串
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

不过两种方法还是有个很重要的区别：Ruby 不会对单引号字符串进行插值操作：

```
>> '#{foo} bar'    # 单引号字符串不能进行插值操作
=> "\#{foo} bar"
```

注意控制台是如何使用双引号返回结果的，需要使用反斜线转义特殊字符，例如 `#`。

如果双引号字符串可以做单引号所做的所有事，而且还能进行插值，那么单引号字符串存在的意义是什么呢？单引号字符串的用处在于它们真的就是字面值，只包含你输入的字符。例如，反斜线在很多系统中都很特殊，就像换行符 (`\n`) 一样。如果有一个变量需要包含一个反斜线，使用单引号就很简单：

```
>> '\n'           # 反斜线和 n 字面值
=> "\\n"
```

和前例的 `#` 字符一样，Ruby 要使用一个额外的反斜线来转义反斜线，在双引号字符串中，要表达一个反斜线就要使用两个反斜线。对简单的例子来说，这省不了多少事，不过如果有很多需要转义的字符就显得出它的作用了：

```
>> 'Newlines (\n) and tabs (\t) both use the backslash character \.'
=> "Newlines (\\\n) and tabs (\\\t) both use the backslash character \\".
```

4.2.3 对象及向其传递消息

Ruby 中一切皆对象，包括字符串和 `nil` 都是。我们会在 [4.4.2 节](#) 介绍对象技术层面上的意义，不过一般很难通过阅读一本书就理解对象，你要多看一些例子才能建立对对象的感性认识。

不过说出对象的作用就很简单：它可以响应消息。例如，一个字符串对象可以响应 `length` 这个消息，它返回字符串包含的字符数量：

```
>> "foobar".length          # 把 Length 消息传递给字符串  
=> 6
```

这样传递给对象的消息叫做方法，它是在对象中定义的函数。⁴ 字符串还可以响应 `empty?` 方法：

```
>> "foobar".empty?  
=> false  
>> "".empty?  
=> true
```

注意 `empty?` 方法末尾的问号，这是 Ruby 的一个约定，说明方法的返回值是布尔值：`true` 或 `false`。布尔值在流程控制中特别有用：

```
>> s = "foobar"  
>> if s.empty?  
>>   "The string is empty"  
>> else  
>>   "The string is nonempty"  
>> end  
=> "The string is nonempty"
```

布尔值还可以使用 `&&`（和）、`||`（或）和 `!`（非）操作符结合使用：

```
>> x = "foo"  
=> "foo"  
>> y = ""  
=> ""  
>> puts "Both strings are empty" if x.empty? && y.empty?  
=> nil  
>> puts "One of the strings is empty" if x.empty? || y.empty?  
"One of the strings is empty"  
=> nil  
>> puts "x is not empty" if !x.empty?  
"x is not empty"  
=> nil
```

因为 Ruby 中的一切都是对象，那么 `nil` 也是对象，所以它也可以响应方法。举个例子，`to_s` 方法基本上可以把任何对象转换成字符串：

```
>> nil.to_s  
=> ""
```

结果显然是个空字符串，我们可以通过下面的方法串联（chain）验证这一点：

```
>> nil.empty?  
NoMethodError: You have a nil object when you didn't expect it!  
You might have expected an instance of Array.  
The error occurred while evaluating nil.empty?  
>> nil.to_s.empty?      # 消息串联  
=> true
```

我们看到，`nil` 对象本身无法响应 `empty?` 方法，但是 `nil.to_s` 可以。

有一个特殊的方法可以测试对象是否为空，你应该能猜到这个方法：

```
>> "foo".nil?  
=> false  
>> "".nil?  
=> false  
>> nil.nil?  
=> true
```

下面的代码

```
puts "x is not empty" if !x.empty?
```

说明了关键词 `if` 的另一种用法：你可以编写一个当且只当 `if` 后面的表达式为真时才执行的语句。对应的，关键词 `unless` 也可以这么用：

```
>> string = "foobar"  
>> puts "The string '#{string}' is nonempty." unless string.empty?  
The string 'foobar' is nonempty.  
=> nil
```

我们需要注意一下 `nil` 的特殊性，除了 `false` 本身之外，所有的 Ruby 对象中它是唯一一个布尔值为“假”的：

```
>> if nil  
>>   true  
>> else  
>>   false      # nil 是假值  
>> end  
=> false
```

基本上所有其他的 Ruby 对象都是“真”的，包括 0：

```
>> if 0
>>   true      # 0 (除了 nil 和 false 之外的一切对象) 是真值
>> else
>>   false
>> end
=> true
```

4.2.4 定义方法

在控制台中，我们可以像定义 `home` 动作（代码 3.6）和 `full_title` 帮助方法（代码 4.2）一样进行方法定义。（在控制台中定义方法有点麻烦，我们一般会在文件中定义，不过用来演示还行。）例如，我们要定义一个名为 `string_message` 的方法，可以接受一个参数，返回值取决于参数是否为空：

```
>> def string_message(string)
>>   if string.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> nil
>> puts string_message("")
It's an empty string!
>> puts string_message("foobar")
The string is nonempty.
```

注意 Ruby 方法会非显式的返回值：返回最后一个语句的值。在上面的这个例子中，返回的值会根据参数是否为空而返回两个字符串中的一个。Ruby 也支持显式的指定返回值，下面的代码和上面的效果一样：

```
>> def string_message(string)
>>   return "It's an empty string!" if string.empty?
>>   return "The string is nonempty."
>> end
```

细心的读者可能会发现其实这里第二个 `return` 不是必须的，作为方法的最后一个表达式，不管有没有 `return`，字符串 `"The string is nonempty."` 都会作为返回值。不过两处都加上 `return` 看起来更好看。

4.2.5 回顾一下标题的帮助方法

下面我们来理解一下代码 4.2 中的 `full_title` 帮助方法：⁵

```

module ApplicationHelper

# 根据所在页面返回完整的标题 # 在文档中显示的注释
def full_title(page_title) # 方法定义
  base_title = "Ruby on Rails Tutorial Sample App" # 变量赋值
  if page_title.empty? # 布尔测试
    base_title # 非显式返回值
  else
    "#{base_title} | #{page_title}" # 字符串插值
  end
end
end

```

方法定义、变量赋值、布尔测试、流程控制和字符串插值——组合在一起定义了一个可以在网站布局中使用的帮助方法。还用到了 `module ApplicationHelper`: `module` 为我们提供了一种把相关方法组织在一起的方式，稍后我们可以使用 `include` 把它插入其他的类中。编写一般的 Ruby 程序时，你要自己定义一个 `module` 然后再显式的将其引入类中，但是对于帮助方法所在的 `module` 就交由 Rails 来处理引入了，最终的结果是 `full_title` 方法自动的 (<http://catb.org/jargon/html/A/automagically.html>) 就可以在所有的视图中使用了。

4.3 其他的数据类型

虽然 Web 程序一般都是处理字符串，但也需要其他的数据类型来生成字符串。本节我们就来介绍一些对开发 Rails 应用程序很重要的 Ruby 中的其他数据类型。

4.3.1 数组和 Range

数组就是一组顺序特定的元素。本书尚且没有用过数组，不过理解了数组就能很好的理解 Hash（4.3.3 节），也有助于理解 Rails 中的数据模型（例如 [2.3.3 节 \(chapter2.html#sec-2-3-3\)](#) 中用到的 `has_many` 关联，[10.1.3 节 \(chapter10.html#sec-10-1-3\)](#) 会做详细介绍）。

目前我们已经花了很多的时间理解字符串，从字符串过渡到数组可以从 `split` 方法开始：

```

>> "foo bar     baz".split      # 把字符串分割成有三个元素的数组
=> [

```

第五章 完善布局

[第四章 \(chapter4.html\)](#)对 Ruby 做了简单的介绍，我们讲解了如何在应用程序中引入样式表，不过，就像在 [4.3.4 节 \(chapter4.html#sec-4-3-4\)](#)中说过的，这个样式表现在还是空的。本章我们会做些修改，把 Bootstrap 框架引入应用程序中，然后再添加一些自定义的样式。¹ 我们还会把已经创建的页面（例如“首页”和“关于”页面）添加到布局中（[5.1 节](#)）。在这个过程中，我们会介绍局部视图（partial）、Rails 路由和 asset pipeline，还会介绍 Sass（[5.2 节](#)）。我们还会用最新的 RSpec 技术重构[第三章 \(chapter3.html\)](#)中的测试。最后，我们还会向前迈出很重要的一步：允许用户在我们的网站中注册。

5.1 添加一些结构

本书是关于 Web 开发而不是 Web 设计的，不过在一个看起来很垃圾的应用程序中开发会让人提不起劲，所以本书我们要向布局中添加一些结构，再加入一些 CSS 构建基本的样式。除了使用自定义的 CSS 之外，我们还会使用 [Bootstrap \(<http://twitter.github.com/bootstrap/>\)](#)，由 Twitter 开发的开源 Web 设计框架。我们还要按照一定的方式组织代码，即使用局部视图来保持布局文件的结构清晰，避免大量的代码混杂在布局文件中。

开发 Web 应用程序时，尽早的对用户界面有个统筹安排往往对你有所帮助。在本书后续内容中，我会经常插入网页的构思图（mockup）（在 Web 领域经常称之为“线框图（wireframe）”），这是对应应用程序最终效果的草图设计。² 本章大部分内容都是在开发 [3.1 节 \(chapter3.html#sec-3-1\)](#)中介绍的静态页面，页面中包含一个网站 LOGO、导航条头部和网站底部。这些网页中最重要的一个是“首页”，它的构思图如图 5.1 所示。图 5.7 是最终实现的效果。你会发现二者之间的某些细节有所不同，例如，在最终实现的页面中我们加入了一个 Rails LOGO——这没什么关系，因为构思图没必要画出每个细节。

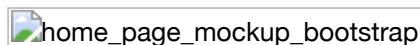


图 5.1：示例程序“首页”的构思图

和之前一样，如果你使用 Git 做版本控制的话，现在最好创建一个新分支：

```
$ git checkout -b filling-in-layout
```

5.1.1 网站导航

在示例程序中加入链接和样式的第一步，要修改布局文件 `application.html.erb`（上次使用是在代码 4.3 中），添加一些 HTML 结构。我们要添加一些区域，一些 CSS class，以及网站导航。布局文件的内容参见代码 5.1，对各部分代码的说明紧跟其后。如果你迫不及待的想看到结果，请查看图 5.2。（注意：结果（还）不是很让人满意。）

代码 5.1 添加一些结构后的网站布局文件

```
app/views/layouts/application.html.erb
```

```

<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag "application", media: "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
    <!--[if lt IE 9]>
      <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>
  <body>
    <header class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <%= link_to "sample app", '#', id: "logo" %>
          <nav>
            <ul class="nav pull-right">
              <li><%= link_to "Home", '#' %></li>
              <li><%= link_to "Help", '#' %></li>
              <li><%= link_to "Sign in", '#' %></li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>

```

需要特别注意一下 Hash 风格从 Ruby 1.8 到 Ruby 1.9 的转变（参见 [4.3.3 节 \(chapter4.html#sec-4-3-3\)](#)）。即把

```
<%= stylesheet_link_tag "application", :media => "all" %>
```

换成

```
<%= stylesheet_link_tag "application", media: "all" %>
```

有一点很重要需要注意一下，因为旧的 Hash 风格使用范围还很广，所以两种用法你都要能够识别。

我们从上往下看一下代码 5.1 中新添加的元素。[3.1 节 \(chapter3.html#sec-3-1\)](#)简单的介绍过，Rails 3 默认会使用 HTML5（如 `<!DOCTYPE html>` 所示），因为 HTML5 标准还很新，有些浏览器（特别是较旧版本的 IE 浏览器）还没有完全支持，所以我们加载了一些 JavaScript 代码（称作“[HTML5 shim \(http://code.google.com/p/html5shim/\)](http://code.google.com/p/html5shim/)”）来解决这个问题：

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

如下有点古怪的句法

```
<!--[if lt IE 9]>
```

只有当 IE 浏览器的版本小于 9 时 (`if lt IE 9`) 才会加载其中的代码。这个奇怪的 `[if lt IE 9]` 句法不是 Rails 提供的，其实它是 IE 浏览器为了解决兼容性问题而特别支持的条件注释 (http://en.wikipedia.org/wiki/Conditional_comment) (conditional comment)。这就带来了一个好处，因为这说明我们只会在 IE9 以前的版本中加载 HTML5 shim，而 Firefox、Chrome 和 Safari 等其他浏览器则不会受到影响。

后面的区域是一个 `header`，包含网站的 LOGO（纯文本）、一些小区域（使用 `div` 标签）和一个导航列表元素：

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", '#', id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", '#' %></li>
          <li><%= link_to "Help", '#' %></li>
          <li><%= link_to "Sign in", '#' %></li>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

`header` 标签的意思是放在网页顶部的内容。我们为 `header` 标签指定了两个 CSS class³，`navbar` 和 `navbar-fixed-top`，用空格分开：

```
<header class="navbar navbar-fixed-top">
```

所有的 HTML 元素都可以指定 `class` 和 `id`，它们不仅是个标注，在 CSS 样式中也有用（[5.1.2 节](#)）。`class` 和 `id` 之间主要的区别是，`class` 可以在同一个网页中多次使用，而 `id` 只能使用一次。这里的 `navbar` 和 `navbar-fixed-top` 在 Bootstrap 框架中有特殊的含义，我们会在 [5.1.2 节](#) 中安装并使用 Bootstrap。`header` 标签内是一些 `div` 标签：

```
<div class="navbar-inner">
  <div class="container">
```

`div` 标签是常规的区域，除了把文档分成不同的部分之外，没有特殊的意义。在以前的 HTML 中，`div` 标签被用来划分网站中几乎所有的区域，但是 HTML5 增加了 `header`、`nav` 和 `section` 元素，用来划分大多数网站中都有用到的区域。本例中，每个 `div` 也都指定了一个 CSS class。和 `header` 标签的 `class` 一样，这些 `class` 在 Bootstrap 中也有特殊的意义。

在这些 `div` 之后，有一些 ERb 代码：

```

<%= link_to "sample app", '#', id: "logo" %>
<nav>
  <ul class="nav pull-right">
    <li><%= link_to "Home", '#' %></li>
    <li><%= link_to "Help", '#' %></li>
    <li><%= link_to "Sign in", '#' %></li>
  </ul>
</nav>

```

这里使用了 Rails 中的 `link_to` 帮助方法来创建链接（在 [3.3.2 节 \(chapter3.html#sec-3-3-2\)](#) 中我们是直接创建 `a` 标签来实现的）。`link_to` 的第一个参数是链接文本，第二个参数是链接地址。在 [5.3.3 节](#) 中我们会指定链接地址为设置好的路由，这里我们用的是 Web 设计中经常使用的占位符 `#`。第三个参数是可选的，为一个 Hash，本例使用这个参数为 LOGO 添加了一个 `logo` id。（其他三个链接没有使用这个 Hash 参数，没关系，因为这个参数是可选的。）Rails 帮助方法经常这样使用 Hash 参数，可以让我们仅使用 Rails 的帮助方法就能灵活的添加 HTML 属性。

第二个 `div` 中是个导航链接列表，使用无序列表标签 `ul`，以及列表项目标签 `li`:

```

<nav>
  <ul class="nav pull-right">
    <li><%= link_to "Home", '#' %></li>
    <li><%= link_to "Help", '#' %></li>
    <li><%= link_to "Sign in", '#' %></li>
  </ul>
</nav>

```

上面代码中的 `nav` 标签以前是不需要的，它的目的是显示导航链接。`ul` 标签指定的 `nav` 和 `pull-right` class 在 Bootstrap 中有特殊的意义。Rails 处理这个布局文件并执行其中的 ERb 代码后，生成的列表如下面的代码所示：

```

<nav>
  <ul class="nav pull-right">
    <li><a href="#">Home</a></li>
    <li><a href="#">Help</a></li>
    <li><a href="#">Sign in</a></li>
  </ul>
</nav>

```

布局文件的最后一个 `div` 是主内容区域：

```

<div class="container">
  <%= yield %>
</div>

```

和之前一样，`container` class 在 Bootstrap 中有特殊的意义。[3.3.4 节 \(chapter3.html#sec-3-3-4\)](#) 已经介绍过，`yield` 会把各页面中的内容插入网站的布局中。

除了网站的底部（在 [5.1.3 节](#) 添加）之外，布局现在就完成了，访问一下“首页”就能看到结果了。为了利用后面添加的样式，我们要向 `home.html.erb` 视图中加入一些元素。（参见代码 5.2。）

代码 5.2 “首页”的代码，包含一个到注册页面的链接

app/views/static_pages/home.html.erb

```
<div class="center hero-unit">
  <h1>Welcome to the Sample App</h1>

  <h2>
    This is the home page for the
    <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </h2>

  <%= link_to "Sign up now!", '#', class: "btn btn-large btn-primary" %>
</div>

<%= link_to image_tag("rails.png", alt: "Rails"), 'http://rubyonrails.org/' %>
```

上面代码中第一个 `link_to` 创建了一个占位链接，指向第七章 (`chapter7.html`) 中创建的用户注册页面

```
<a href="#" class="btn btn-large btn-primary">Sign up now!</a>
```

`div` 标签中的 `hero-unit` class 在 Bootstrap 中有特殊的意义，注册按钮的 `btn`、`btn-large` 和 `btn-primary` 也是一样。

第二个 `link_to` 用到了 `image_tag` 帮助方法，第一个参数是图片的路径；第二个参数是可选的，一个 Hash，本例中这个 Hash 参数使用一个 Symbol 键设置了图片的 `alt` 属性。为了更好的理解，我们来看一下生成的 HTML：⁴

```

```

`alt` 属性的内容会在图片无法加载时显示，也会在针对视觉障碍人士的屏幕阅读器中显示。人们有时懒得加上 `alt` 属性，可是在 HTML 标准中却是必须的。幸运的是，Rails 默认会加上 `alt` 标签，如果你没有在调用 `image_tag` 时指定的话，Rails 就会使用图片的文件名（不包括扩展名）。本例中，我们自己设定了 `alt` 文本，显示一个首字母大写的“Rails”。

现在我们终于可以看到劳动的果实了（如图 5.2）。你可能会说，这并不很美观啊。或许吧。不过也可以小小的高兴一下，我们已经为 HTML 结构指定了合适的 `class`，可以用来添加 CSS。

顺便说一下，你可能会奇怪 `rails.png` 这个图片为什么可以显示出来，它是怎么来的呢？其实每个 Rails 应用程序中都有这个图片，存放在 `app/assets/images/` 目录下。因为我们使用的是 `image_tag` 帮助方法，Rails 会通过 asset pipeline 找到这个图片。（[5.2 节](#)）

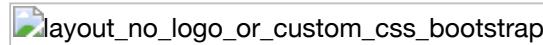


图 5.2：没有定义 CSS 的“首页” (`/static_pages/home` (http://localhost:3000/static_pages/home))

5.1.2 Bootstrap 和自定义的 CSS

在 [5.1.1 节](#) 我们为很多 HTML 元素指定了 CSS class，这样我们就可以使用 CSS 灵活的构建布局了。[5.1.1 节](#) 中已经说过，很多 class 在 Bootstrap 中都有特殊的意义。Bootstrap 是 Twitter 开发的框架，可以方便的把精美的 Web 设计和用户界面元素添加到使用 HTML5 开发的应用程序中。本节，我们会结合 Bootstrap 和一些自定义的 CSS 为示例程序添加样式。

首先要安装 Bootstrap，在 Rails 程序中可以使用 bootstrap-sass 这个 gem，参见代码 5.3。Bootstrap 框架本身使用 LESS 来动态的生成样式表，而 Rails 的 asset pipeline 默认支持的是（非常类似的）Sass，bootstrap-sass 会将 LESS 转换成 Sass 格式，而且 Bootstrap 中必要的文件都可以在当前的应用程序中使用。⁵

代码 5.3 把 bootstrap-sass 加入 Gemfile

```
source 'https://rubygems.org'

gem 'rails', '3.2.13'
gem 'bootstrap-sass', '2.0.4'
.
```

像往常一样，运行 bundle install 安装 Bootstrap：

```
$ bundle install
```

然后重启 Web 服务器，改动才能在应用程序中生效。（在大多数系统中可以使用 Ctrl-C 结束服务器，然后再执行 rails server 命令。）

要向应用程序中添加自定义的 CSS，首先要创建一个 CSS 文件：

```
app/assets/stylesheets/custom.css.scss
```

（使用你喜欢的文本编辑器或者 IDE 创建这个文件。）文件存放的目录和文件名都很重要。其中目录

```
app/assets/stylesheets
```

是 asset pipeline 的一部分（[5.2 节](#)），这个目录中的所有样式表都会自动的包含在网站的 application.css 中。custom.css.scss 文件的第一个扩展名是 .css，说明这是个 CSS 文件；第二个扩展名是 .scss，说明这是个“Sassy CSS”文件。asset pipeline 会使用 Sass 处理这个文件。（在 [5.2.2 节](#) 中才会使用 Sass，有了它 bootstrap-sass 才能运作。）创建了自定义 CSS 所需的文件后，我们可以使用 @import 引入 Bootstrap，如代码 5.4 所示。

代码 5.4 引入 Bootstrap

```
app/assets/stylesheets/custom.css.scss
```

```
@import "bootstrap";
```

这行代码会引入整个 Bootstrap CSS 框架，结果如图 5.3 所示。（或许你要通过 Ctrl-C 来重启服务器。）可以看到，文本的位置还不是很合适，LOGO 也没有任何样式，不过颜色搭配和注册按钮看起来还不错。

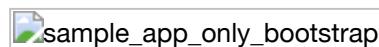


图 5.3：使用 Bootstrap CSS 后的示例程序

下面我们要加入一些整站都会用到的 CSS，用来样式化网站布局和各单独页面，如代码 5.5 所示。代码 5.5 中定义了很多样式规则。为了说明 CSS 规则的作用，我们经常会加入一些 CSS 注释，放在 `/*...*/` 之中。代码 5.5 的 CSS 加载后的效果如图 5.4 所示。

代码 5.5 添加全站使用的 CSS

`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";

/* universal */

html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```



图 5.4：添加一些空白和其他的全局性样式

注意代码 5.5 中的 CSS 格式是很统一的。一般来说，CSS 规则通过 class、id、HTML 标签或者三者结合起来定义的，后面会跟着一些样式声明。例如：

```
body {
  padding-top: 60px;
}
```

把页面的上内边距设为 60 像素。我们在 `header` 标签上指定了 `navbar-fixed-top` class，Bootstrap 就把这个导航条固定在页面的顶部。所以页面的上内边距会把主内容区和导航条隔开一段距离。下面的 CSS 规则：

```
.center {  
    text-align: center;  
}
```

把 .center class 的样式定义为 `text-align: center;`。.center 中的点号说明这个规则是样式化一个 class。（我们会在代码 5.7 中看到，# 是样式化一个 id。）这个规则的意思是，任何 class 为 .center 的标签（例如 div），其中包含的内容都会在页面中居中显示。（代码 5.2 中有用到这个 class。）

虽然 Bootstrap 中包含了很精美的文字排版样式，我们还是要为网站添加一些自定义的规则，如代码 5.6 所示。（并不是所有的样式都会应用于“首页”，但所有规则都会在网站中的某个地方用到。）代码 5.6 的效果如图 5.5 所示。

代码 5.6 添加一些精美的文字排版样式

app/assets/stylesheets/custom.css.scss

```
@import "bootstrap";  
. . .  
  
/* typography */  
  
h1, h2, h3, h4, h5, h6 {  
    line-height: 1;  
}  
  
h1 {  
    font-size: 3em;  
    letter-spacing: -2px;  
    margin-bottom: 30px;  
    text-align: center;  
}  
  
h2 {  
    font-size: 1.7em;  
    letter-spacing: -1px;  
    margin-bottom: 30px;  
    text-align: center;  
    font-weight: normal;  
    color: #999;  
}  
  
p {  
    font-size: 1.1em;  
    line-height: 1.7em;  
}
```



图 5.5：添加了一些文字排版样式

第六章 图像聚类

6.1 K-Means聚类

6.1.1 SciPy聚类包

6.1.2 图像聚类

6.1.3 在主成分上可视化图像

6.1.4 像素聚类

6.2 层次聚类

6.2.1 图像聚类

6.3 谱聚类

这一章会介绍几种聚类方法，并就怎么使用它们对图像进行聚类找出相似的图像组进行说明。聚类可以用于识别，划分图像数据集、组织导航等。同时，我们也会用聚类相似的图像进行可视化。

6.1 K-Means聚类

K-means是一种非常简单的聚类算法，它能够将输入数据划分成k个簇。关于K-means聚类算法的介绍可以参阅中译本。

6.1.1 SciPy聚类包

尽管K-means聚类算法很容易实现，但我们没必要自己去实现。SciPy矢量量化包`sci.cluter.vq`中有k-means的实现。这里我们演示怎样使用它。

我们以2维示例样本数据进行说明：

```
# coding=utf-8
"""
Function: figure 6.1
An example of k-means clustering of 2D points
"""

from pylab import *
from scipy.cluster.vq import *

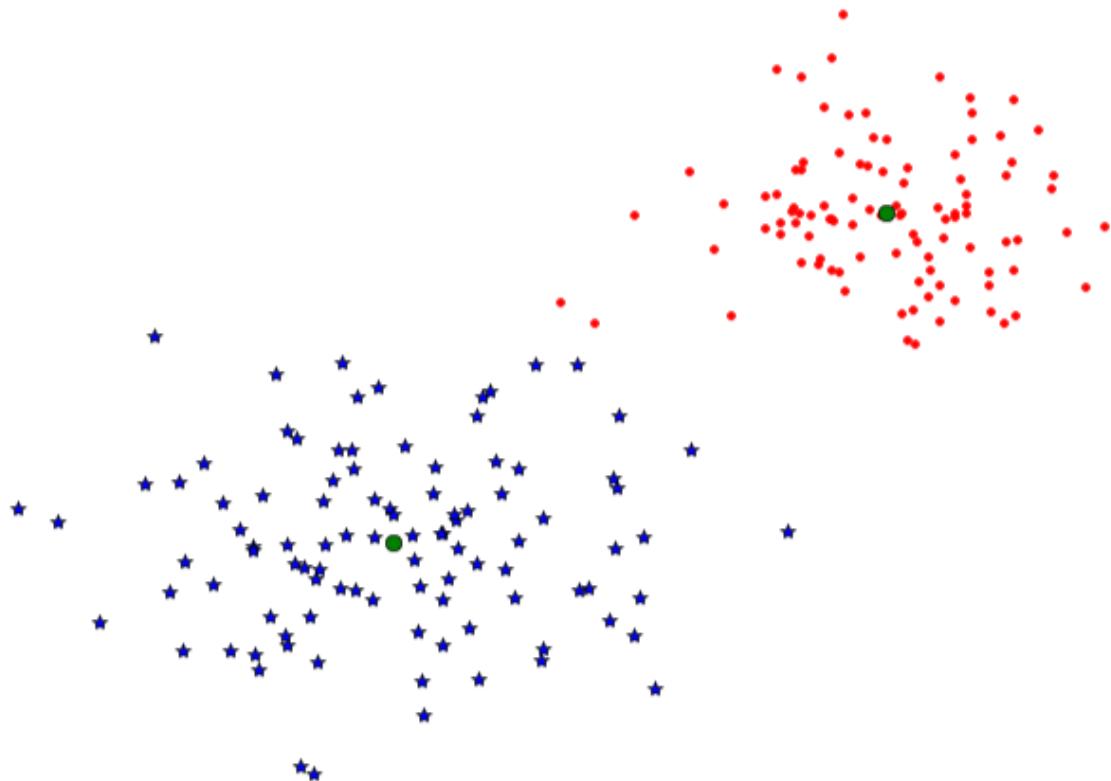
# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

class1 = 1.5 * randn(100, 2)
class2 = randn(100, 2) + array([5, 5])
features = vstack((class1, class2))
centroids, variance = kmeans(features, 2)
code, distance = vq(features, centroids)

figure()
ndx = where(code == 0)[0]
plot(features[ndx, 0], features[ndx, 1], '*')
ndx = where(code == 1)[0]
plot(features[ndx, 0], features[ndx, 1], 'r.')
plot(centroids[:, 0], centroids[:, 1], 'go')

title(u'2维数据点聚类', fontproperties=font)
axis('off')
show()
```

上面代码中where()函数给出每类的索引。运行上面代码，可得到原书P129页图6-1，即：



6.1.2 图像聚类

现在我们用k-means对原书14页的图像进行聚类，文件selectedfontimages.zip包含了66张字体图像。对于每一张图像，我们用在前40个主成分上投影后的系数作为特征向量。下面为对其进行聚类的代码：

```
# -*- coding: utf-8 -*-
from PCV.tools import imtools
import pickle
from scipy import *
from pylab import *
from PIL import Image
from scipy.cluster.vq import *
from PCV.tools import pca

# Uses sparse pca codepath.
imlist = imtools.get_imlist('../data/selectedfontimages/a_selected_thumbs/')

# 获取图像列表和他们的尺寸
im = array(Image.open(imlist[0])) # open one image to get the size
m, n = im.shape[:2] # get the size of the images
imnbr = len(imlist) # get the number of images
print "The number of images is %d" % imnbr

# Create matrix to store all flattened images
immatrix = array([array(Image.open(imname)).flatten() for imname in imlist], 'f')
```

```

# PCA降维
V, S, immean = pca.pca(immatrix)

# 保存均值和主成分
#f = open('./a_pca_modes.pkl', 'wb')
#f = open('./a_pca_modes.pkl', 'wb')
pickle.dump(immean,f)
pickle.dump(V,f)
f.close()

# get list of images
imlist = imtools.get_imlist('../data/selectedfontimages/a_selected_thumbs/')
imnbr = len(imlist)

# Load model file
with open('../data/selectedfontimages/a_pca_modes.pkl','rb') as f:
    immean = pickle.load(f)
    V = pickle.load(f)
# create matrix to store all flattened images
immatrix = array([array(Image.open(im)).flatten() for im in imlist],'f')

# project on the 40 first PCs
immean = immean.flatten()
projected = array([dot(V[:40],immatrix[i]-immean) for i in range(imnbr)])

# k-means
projected = whiten(projected)
centroids,distortion = kmeans(projected,4)
code,distance = vq(projected,centroids)

# plot clusters
for k in range(4):
    ind = where(code==k)[0]
    figure()
    gray()
    for i in range(minimum(len(ind),40)):
        subplot(4,10,i+1)
        imshow(immatrix[ind[i]].reshape((25,25)))
        axis('off')
show()

```

运行上面代码，可得到下面的聚类结果：

аааааааааа **аааааааааа**
аааааа **аааааа**

аааааааааа **аааааааааа**
 аааааааааа
 ааааааа

注：这里的结果译者截的是原书上的结果，上面代码实际运行出来的结果可能跟上面有出入。

6.1.3 在主成分上可视化图像

```
# -*- coding: utf-8 -*-
from PCV.tools import imtools, pca
from PIL import Image, ImageDraw
from pylab import *
from PCV.clustering import hcluster

imlist = imtools.get_imlist('../data/selectedfontimages/a_selected_thumbs')
imnbr = len(imlist)

# Load images, run PCA.
immatrix = array([array(Image.open(im)).flatten() for im in imlist], 'f')
V, S, immean = pca.pca(immatrix)

# Project on 2 PCs.
projected = array([dot(V[[0, 1]], immatrix[i] - immean) for i in range(imnbr)]) # P131 Fig6-3左图
#projected = array([dot(V[[1, 2]], immatrix[i] - immean) for i in range(imnbr)]) # P131 Fig6-3右图

# height and width
h, w = 1200, 1200

# create a new image with a white background
img = Image.new('RGB', (w, h), (255, 255, 255))
draw = ImageDraw.Draw(img)

# draw axis
draw.line((0, h/2, w, h/2), fill=(255, 0, 0))
draw.line((w/2, 0, w/2, h), fill=(255, 0, 0))

# scale coordinates to fit
scale = abs(projected).max(0)
scaled = floor(array([(p/scale) * (w/2 - 20, h/2 - 20) + (w/2, h/2)
                     for p in projected])).astype(int)
```

```

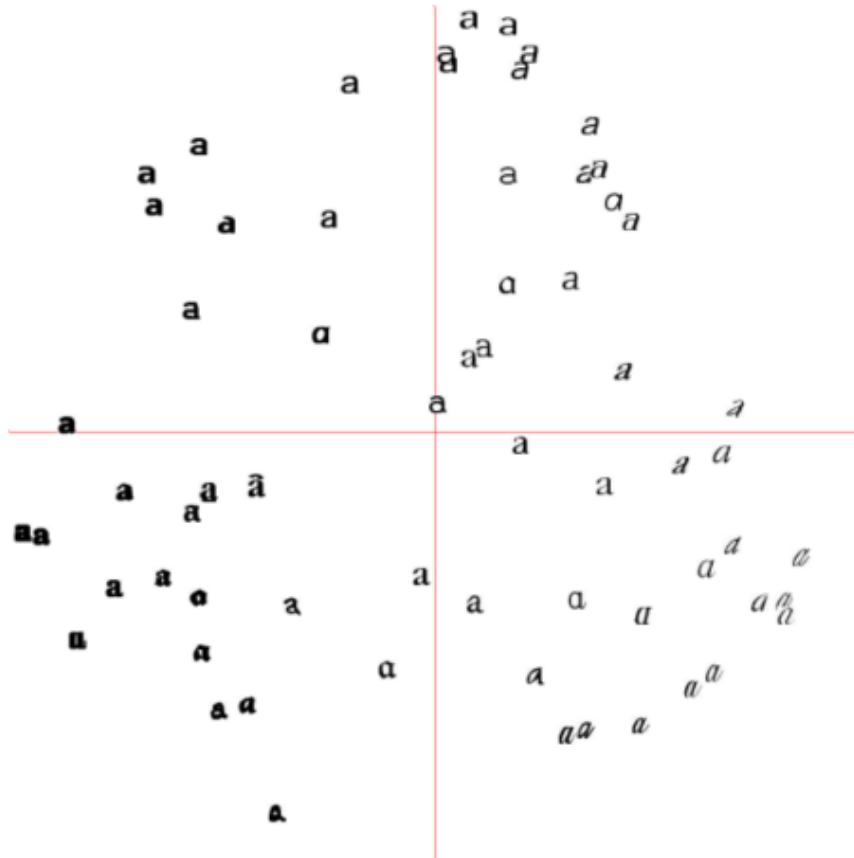
# paste thumbnail of each image
for i in range(imnbr):
    nodeim = Image.open(imlist[i])
    nodeim.thumbnail((25, 25))
    ns = nodeim.size
    box = (scaled[i][0] - ns[0] // 2, scaled[i][1] - ns[1] // 2,
           scaled[i][0] + ns[0] // 2 + 1, scaled[i][1] + ns[1] // 2 + 1)
    img.paste(nodeim, box)

tree = hcluster.hcluster(projected)
hcluster.draw_dendrogram(tree, imlist, filename='fonts.png')

figure()
imshow(img)
axis('off')
img.save('../images/ch06/pca_font.png')
show()

```

运行上面代码，可画出原书P131图6-3中的实例结果。



6.1.4 像素聚类

在结束这节前，我们看一个对像素进行聚类而不是对所有的图像进行聚类的例子。将图像区域归并成“有意义的”组件称为图像分割。在第九章会将其单独列作为一个主题。在像素级水平进行聚类除了可以用在一些很简单的图像，在其他图像上进行聚类是没有意义的。这里，我们将k-means应用到RGB颜色值上，关于分割问题会在第九章第二节会给出分割的方法。下面是对两幅图像进行像素聚类的例子(注：译者对原书中的代码做了调整)：

```

# -*- coding: utf-8 -*-
"""

```

Function: figure 6.4

Clustering of pixels based on their color value using k-means.

```
"""
from scipy.cluster.vq import *
from scipy.misc import imresize
from pylab import *
import Image

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

def clusterpixels(infile, k, steps):
    im = array(Image.open(infile))
    dx = im.shape[0] / steps
    dy = im.shape[1] / steps
    # compute color features for each region
    features = []
    for x in range(steps):
        for y in range(steps):
            R = mean(im[x * dx:(x + 1) * dx, y * dy:(y + 1) * dy, 0])
            G = mean(im[x * dx:(x + 1) * dx, y * dy:(y + 1) * dy, 1])
            B = mean(im[x * dx:(x + 1) * dx, y * dy:(y + 1) * dy, 2])
            features.append([R, G, B])
    features = array(features, 'f')      # make into array
    # 聚类, k是聚类数目
    centroids, variance = kmeans(features, k)
    code, distance = vq(features, centroids)
    # create image with cluster labels
    codeim = code.reshape(steps, steps)
    codeim = imresize(codeim, im.shape[:2], 'nearest')
    return codeim

k=3
infile_empire = '../data/empire.jpg'
im_empire = array(Image.open(infile_empire))
infile_boy_on_hill = '../data/boy_on_hill.jpg'
im_boy_on_hill = array(Image.open(infile_boy_on_hill))
steps = (50, 100)  # image is divided in steps*steps region
print steps[0], steps[-1]

#显示原图empire.jpg
figure()
subplot(231)
title(u'原图', fontproperties=font)
axis('off')
imshow(im_empire)

# 用50*50的块对empire.jpg的像素进行聚类
```

```

codeim= clusterpixels(infile_empire, k, steps[0])
subplot(232)
title(u'k=3,steps=50', fontproperties=font)
#ax1.set_title('Image')
axis('off')
imshow(codeim)

# 用100*100的块对empire.jpg的像素进行聚类
codeim= clusterpixels(infile_empire, k, steps[-1])
ax1 = subplot(233)
title(u'k=3,steps=100', fontproperties=font)
#ax1.set_title('Image')
axis('off')
imshow(codeim)

#显示原图empire.jpg
subplot(234)
title(u'原图', fontproperties=font)
axis('off')
imshow(im_boy_on_hill)

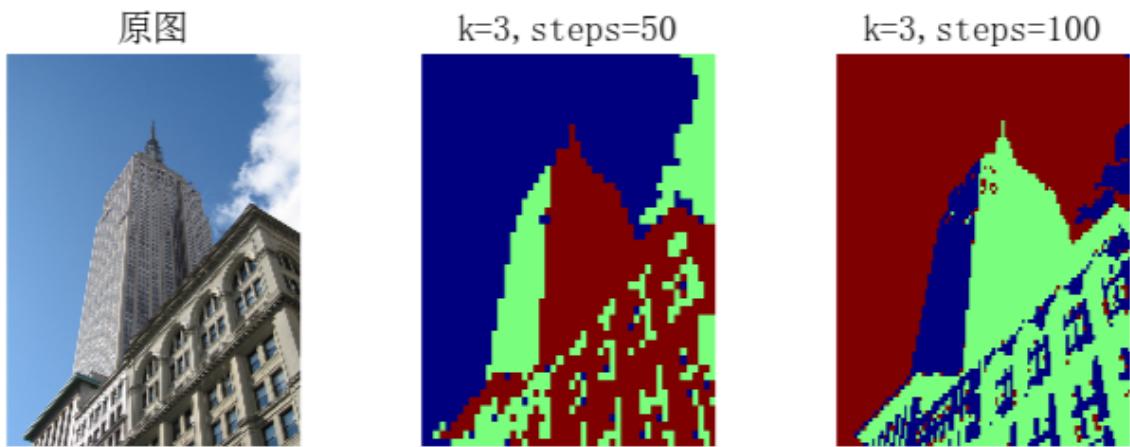
# 用50*50的块对empire.jpg的像素进行聚类
codeim= clusterpixels(infile_boy_on_hill, k, steps[0])
subplot(235)
title(u'k=3,steps=50', fontproperties=font)
#ax1.set_title('Image')
axis('off')
imshow(codeim)

# 用100*100的块对empire.jpg的像素进行聚类
codeim= clusterpixels(infile_boy_on_hill, k, steps[-1])
subplot(236)
title(u'k=3, steps=100', fontproperties=font)
axis('off')
imshow(codeim)

show()

```

上面代码中，先载入一幅图像，然后用一个 $steps \times steps$ 的方块在原图中滑动，对窗口中的图像值求和取平均，将它下采样到一个较低的分辨率，然后对这些区域用k-means进行聚类。运行上面代码，即可得出原书P133页图6-4中的图。



6.2 层次聚类

层次聚类(或称凝聚聚类)是另一种简单但有效的聚类算法。下面我们通过一个简单的实例看看层次聚类是怎样进行的。

```
from pylab import *
from PCV.clustering import hcluster

class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))

tree = hcluster.hcluster(features)
clusters = tree.extract_clusters(5)
print 'number of clusters', len(clusters)
for c in clusters:
    print c.get_cluster_elements()
```

上面代码首先创建一些2维数据点，然后对这些数据点聚类，用一些阈值提取列表中的聚类后的簇群，并将它们打印出来，译者在自己的笔记本上打印出的结果为：

```
number of clusters 2  
[197, 107, 176, 123, 173, 189, 154, 136, 183, 113, 109, 199, 178, 129, 163, 100, 148, 111, 143,  
118, 162, 169, 138, 182, 193, 116, 134, 198, 184, 181, 131, 166, 127, 185, 161, 171, 152, 157,  
112, 186, 128, 156, 108, 158, 120, 174, 102, 137, 117, 194, 159, 105, 155, 132, 188, 125, 180,  
151, 192, 164, 195, 126, 103, 196, 179, 146, 147, 135, 139, 110, 140, 106, 104, 115, 149, 190,  
170, 172, 121, 145, 114, 150, 119, 142, 122, 144, 160, 187, 153, 167, 130, 133, 165, 191, 175,  
177, 101, 141, 124, 168]  
[0, 39, 32, 87, 40, 48, 28, 8, 26, 12, 94, 5, 1, 61, 24, 59, 83, 10, 99, 50, 23, 58, 51, 16, 71,  
25, 11, 37, 22, 46, 60, 86, 65, 2, 21, 4, 41, 72, 80, 84, 33, 56, 75, 77, 29, 85, 93, 7, 73, 6,  
82, 36, 49, 98, 79, 43, 91, 14, 47, 63, 3, 97, 35, 18, 44, 30, 13, 67, 62, 20, 57, 89, 88, 9, 54,  
19, 15, 92, 38, 64, 45, 70, 52, 95, 69, 96, 42, 53, 27, 66, 90, 81, 31, 34, 74, 76, 17, 78, 55,  
68]
```

6.2.1 图像聚类

```

# -*- coding: utf-8 -*-
import os
import Image
from PCV.clustering import hcluster
from matplotlib.pyplot import *
from numpy import *

# create a list of images
path = '../data/sunsets/flickr-sunsets-small/'
imlist = [os.path.join(path, f) for f in os.listdir(path) if f.endswith('.jpg')]
# extract feature vector (8 bins per color channel)
features = zeros([len(imlist), 512])
for i, f in enumerate(imlist):
    im = array(Image.open(f))
    # multi-dimensional histogram
    h, edges = histogramdd(im.reshape(-1, 3), 8, normed=True, range=[(0, 255), (0, 255), (0, 255)])
    features[i] = h.flatten()
tree = hcluster.hcluster(features)

# visualize clusters with some (arbitrary) threshold
clusters = tree.extract_clusters(0.23 * tree.distance)
# plot images for clusters with more than 3 elements
for c in clusters:
    elements = c.get_cluster_elements()
    nbr_elements = len(elements)
    if nbr_elements > 3:
        figure()
        for p in range(minimum(nbr_elements, 20)):
            subplot(4, 5, p + 1)
            im = array(Image.open(imlist[elements[p]]))
            imshow(im)
            axis('off')
show()

hcluster.draw_dendrogram(tree, imlist, filename='sunset.pdf')

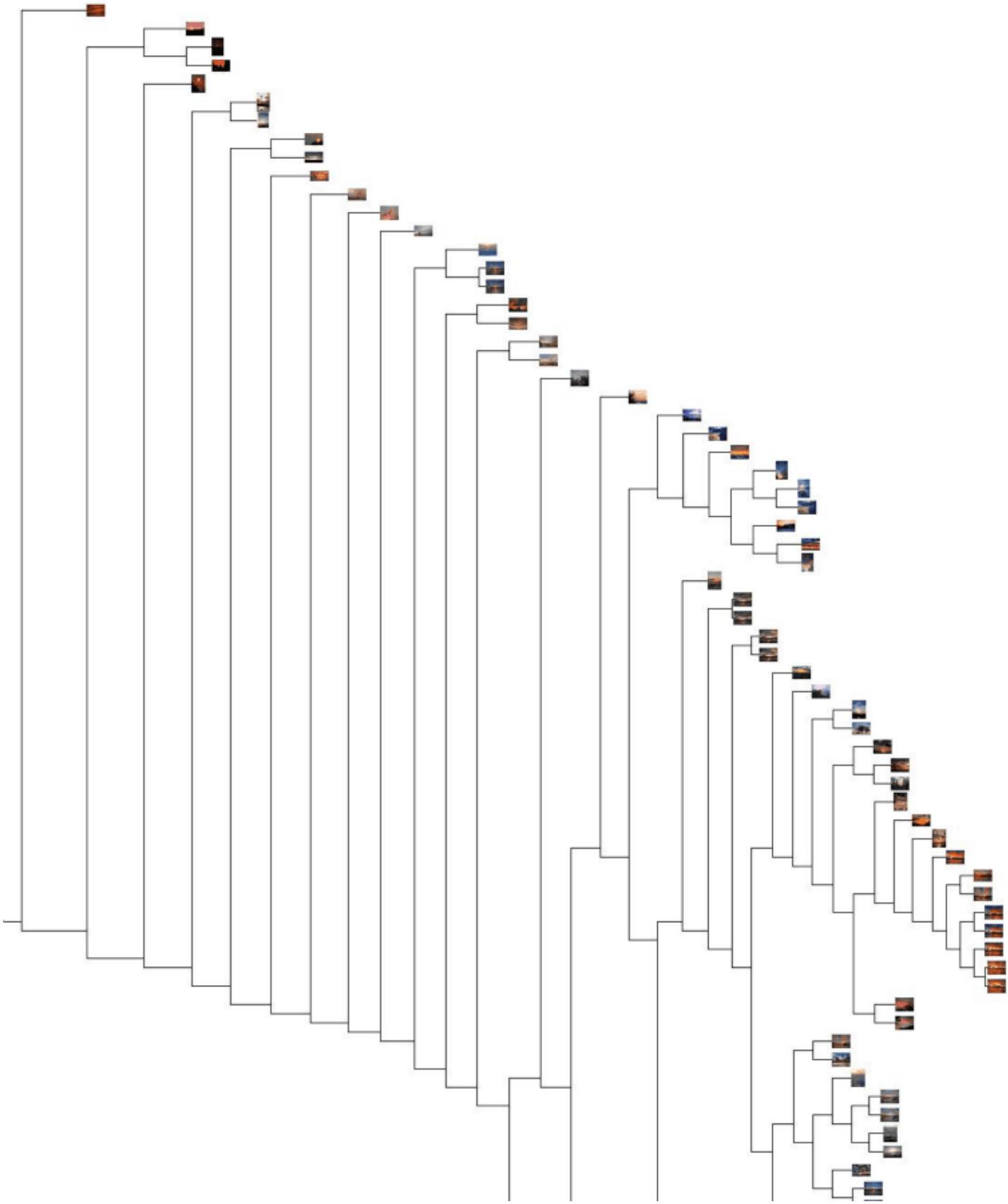
```

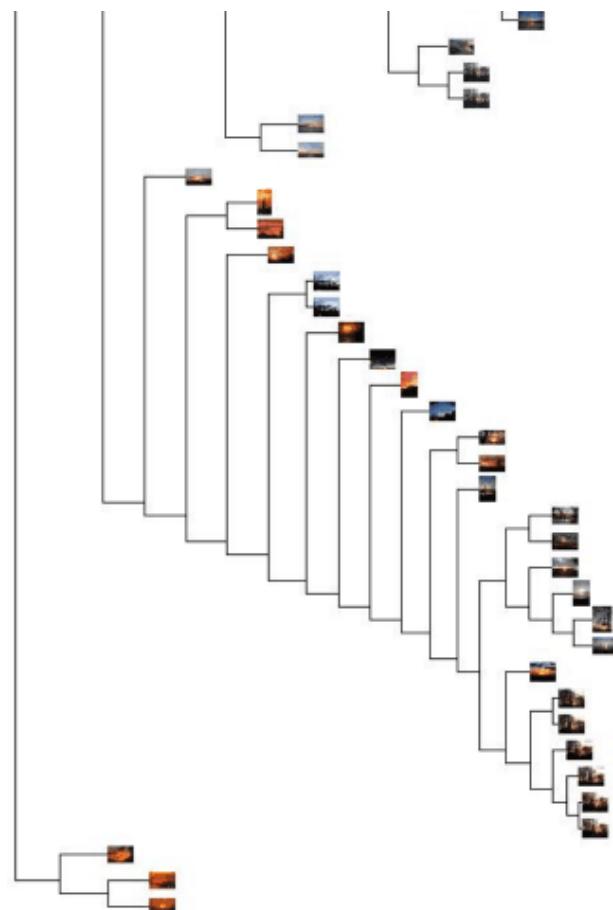
运行上面代码，可得原书P140图6-6。





同时会在上面脚本文件所在的文件夹下生成层次聚类后的簇群树:

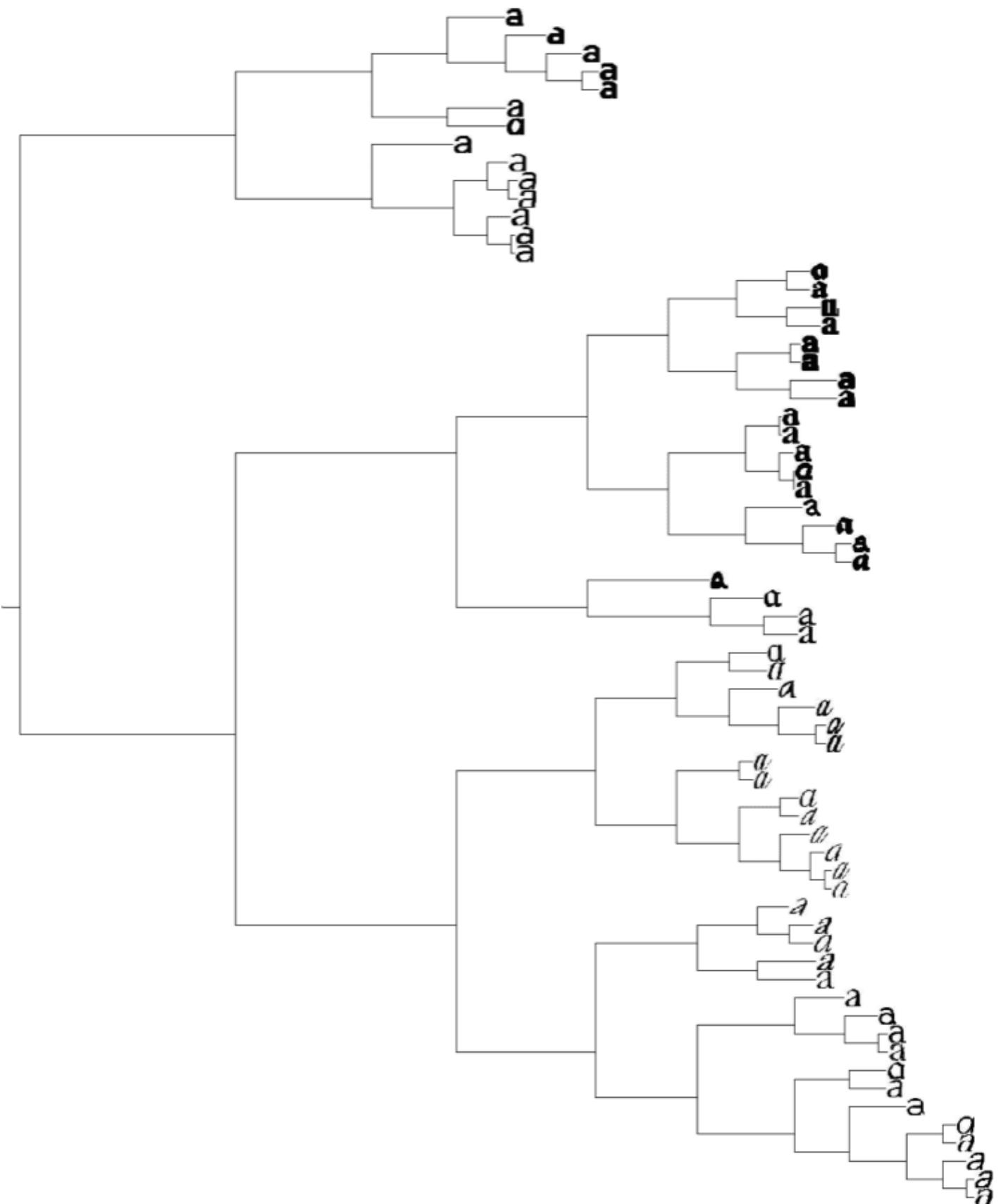




我们对前面字体图像同样创建一个树，正如前面在主成分可视化图像中，我们添加了下面代码：

```
tree = hcluster.hcluster(projected)
hcluster.draw_dendrogram(tree, imlist, filename='fonts.png')
```

运行添加上面两行代码后前面的例子，可得对字体进行层次聚类后的簇群树：



6.3 谱聚类

谱聚类是另一种不同于k-means和层次聚类的聚类算法。关于谱聚类的原理，可以参阅中译本。这里，我们用原来k-means实例中用到的字体图像。

```
# -*- coding: utf-8 -*-
from PCV.tools import imtools, pca
from PIL import Image, ImageDraw
from pylab import *
```

```

from scipy.cluster.vq import *

imlist = imtools.get_imlist('../data/selectedfontimages/a_selected_thumbs')
imnbr = len(imlist)

# Load images, run PCA.
immatrix = array([array(Image.open(im)).flatten() for im in imlist], 'f')
V, S, immean = pca.pca(immatrix)

# Project on 2 PCs.
projected = array([dot(V[[0, 1]], immatrix[i] - immean) for i in range(imnbr)]) # P131 Fig6-3左图
#projected = array([dot(V[[1, 2]], immatrix[i] - immean) for i in range(imnbr)]) # P131 Fig6-3右图

n = len(projected)
# compute distance matrix
S = array([[sqrt(sum((projected[i]-projected[j])**2)) for j in range(n)], 'f'])
for i in range(n) [for j in range(n)], 'f')
# create Laplacian matrix
rowsum = sum(S, axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D, dot(S, D))
# compute eigenvectors of L
U, sigma, V = linalg.svd(L)
k = 5
# create feature vector from k first eigenvectors
# by stacking eigenvectors as columns
features = array(V[:, :k]).T
# k-means
features = whiten(features)
centroids, distortion = kmeans(features, k)
code, distance = vq(features, centroids)
# plot clusters
for c in range(k):
    ind = where(code == c)[0]
    figure()
    gray()
    for i in range(minimum(len(ind), 39)):
        im = Image.open(imlist[ind[i]])
        subplot(4, 10, i+1)
        imshow(array(im))
        axis('equal')
        axis('off')
show()

```

上面我们在前个特征向量上计算标准的k-means。下面是运行上面代码的结果：

a a a a a a a a a a

a a a a

a a a a a a o a a a

a u a a a a a a

a a a a a a a a a a

a @ a a a

注意，由于在k-means阶段会给出不同的聚类结果，所以你运行上面代码出来的结果可能跟译者的是不一样的。

同样，我们可以在不知道特征向量或是没有严格相似性定义的情况下进行谱聚类。原书44页的位置地理图像是通过它们之间有多少局部描述子匹配相连接的。48页的相似性矩阵中的元素是为规范化的匹配特征点数。我们同样可以对其进行谱聚类，

完整的代码如下：

```

# -*- coding: utf-8 -*-
from PCV.tools import imtools, pca
from PIL import Image, ImageDraw
from PCV.localdescriptors import sift
from pylab import *
import glob
from scipy.cluster.vq import *

#download_path = "panoimages" # set this to the path where you downloaded the panoramio images
#path = "/FULLPATH/panoimages/" # path to save thumbnails (pydot needs the full system path)

download_path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages" # set this to the
path where you downloaded the panoramio images
path = "F:/dropbox/Dropbox/translation/pcv-notebook/data/panoimages/" # path to save thumbnails
(pydot needs the full system path)

# List of downloaded filenames
imlist = imtools.get_imlist('../data/panoimages/')
nbr_images = len(imlist)

# extract features
#featlist = [imname[:-3] + 'sift' for imname in imlist]
#for i, imname in enumerate(imlist):
#    sift.process_image(imname, featlist[i])

featlist = glob.glob('../data/panoimages/*.sift')

matchscores = zeros((nbr_images, nbr_images))

for i in range(nbr_images):
    for j in range(i, nbr_images): # only compute upper triangle
        print 'comparing ', imlist[i], imlist[j]
        l1, d1 = sift.read_features_from_file(featlist[i])
        l2, d2 = sift.read_features_from_file(featlist[j])
        matches = sift.match_twosided(d1, d2)
        nbr_matches = sum(matches > 0)
        print 'number of matches = ', nbr_matches
        matchscores[i, j] = nbr_matches
print "The match scores is: \n", matchscores

# copy values
for i in range(nbr_images):
    for j in range(i + 1, nbr_images): # no need to copy diagonal
        matchscores[j, i] = matchscores[i, j]

n = len(imlist)
# Load the similarity matrix and reformat
S = matchscores

```

```

S = 1 / (S + 1e-6)
# create Laplacian matrix
rowsum = sum(S, axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D, dot(S, D))
# compute eigenvectors of L
U, sigma, V = linalg.svd(L)
k = 2
# create feature vector from k first eigenvectors
# by stacking eigenvectors as columns
features = array(V[:k]).T
# k-means
features = whiten(features)
centroids, distortion = kmeans(features, k)
code, distance = vq(features, centroids)
# plot clusters
for c in range(k):
    ind = where(code==c)[0]
    figure()
    gray()
    for i in range(minimum(len(ind), 39)):
        im = Image.open(imlist[ind[i]])
        subplot(5,4,i+1)
        imshow(array(im))
        axis('equal')
        axis('off')
show()

```

改变聚类数目k，可以得到不同的结果。译者分别测试了原书中k=2和k=10的情况，运行结果如下： **k=2**



k=10



注：对于聚类后，图像小于或等于1的类，在上面没有显示。

第七章 图像搜索

7.0 安装CherryPy

7.1 创建词汇

7.2 添加图像

7.3 获取候选图像

7.4 建立演示程序及Web应用

7.5 配置service.conf

本章将展示如何利用文本挖掘技术基于图像视觉内容进行图像搜索。在本章中，阐明了利用视觉单词的基本思想，完整解释了的安装细节，并且还在一个示例数据集上进行测试。

本章图像搜索模型是建立在BoW词袋基础上，先对图像数据库提取sift特征，对提取出来的所有sift特征进行kmeans聚类得到视觉单词(每个视觉单词用逆文档词频配以一定的权重)，然后对每幅图像的sift描述子进行统计得到每幅图像的单词直方图表示，最后对给定的查询图像，将其对应的单词直方图与数据库中的单词直方图进行欧式距离匹配，并由大到小进行排序，最后显示靠前的图像。

7.0 安装CherryPy

在接着下面示例的学习前，先介绍CherryPy的安装，以供后面建立web演示实例使用。

7.1 创建词汇

为创建视觉单词词汇，首先需要提取特征描述子，这里，我们使用SIFT描述子。

```

# -*- coding: utf-8 -*-
import pickle
from PCV.imagesearch import vocabulary
from PCV.tools.imtools import get_imlist
from PCV.localdescriptors import sift

#获取图像列表
imlist = get_imlist('./first500/')
nbr_images = len(imlist)
#获取特征列表
featlist = [imlist[i][:-3] + 'sift' for i in range(nbr_images)]

#提取文件夹下图像的sift特征
for i in range(nbr_images):
    sift.process_image(imlist[i], featlist[i])

#生成词汇
voc = vocabulary.Vocabulary('ukbenchtest')
voc.train(featlist, 1000, 10)
#保存词汇
# saving vocabulary
with open('./first500/vocabulary.pkl', 'wb') as f:
    pickle.dump(voc, f)
print 'vocabulary is:', voc.name, voc.nbr_words

```

上面源码对应ch07_cocabulary.py (<https://github.com/willard-yuan/pcv-book-code/tree/master/ch07>)。在该文件夹下，有一个first500的文件夹，将你从首页下载的数据 (<http://yuanyong.org/pcvwithpython/>) 中文件夹first1000中的图像放在first500中。注意，译者这里实验的时候，由于计算机内存不足，所以只从first1000取出前500张放入first500中。

运行上面代码，会在first500文件夹下生成一个名为vocabulary.pkl的文件，同时在first500会多出500个后缀为.sift的文件，它们分别对应每幅图像提取出来的sift特征描述子。

7.2 添加图像

```

# -*- coding: utf-8 -*-
import pickle
from PCV.imagesearch import imagesearch
from PCV.localdescriptors import sift
from sqlite3 import dbapi2 as sqlite
from PCV.tools.imtools import get_imlist

#获取图像列表
imlist = get_imlist('./first500/')
nbr_images = len(imlist)
#获取特征列表
featlist = [imlist[i][-3:]+'sift' for i in range(nbr_images)]

# Load vocabulary
#载入词汇
with open('./first500/vocabulary.pkl', 'rb') as f:
    voc = pickle.load(f)
#创建索引
indx = imagesearch.Indexer('testImaAdd.db',voc)
indx.create_tables()
# go through all images, project features on vocabulary and insert
#遍历所有的图像，并将它们的特征投影到词汇上
for i in range(nbr_images)[:500]:
    locs,descr = sift.read_features_from_file(featlist[i])
    indx.add_to_index(imlist[i],descr)
# commit to database
#提交到数据库
indx.db_commit()

con = sqlite.connect('testImaAdd.db')
print con.execute('select count (filename) from imlist').fetchone()
print con.execute('select * from imlist').fetchone()

```

运行上面代码后，会在根目录生成建立的索引数据库testImaAdd.db，

7.3 获取候选图像

```

# -*- coding: utf-8 -*-
import pickle
from PCV.imagesearch import imagesearch
from PCV.localdescriptors import sift
from sqlite3 import dbapi2 as sqlite
from PCV.tools.imtools import get_imlist

#获取图像列表
imlist = get_imlist('./first500/')
nbr_images = len(imlist)
#获取特征列表
featlist = [imlist[i][-3:]+'sift' for i in range(nbr_images)]

#载入词汇
f = open('./first500/vocabulary.pkl', 'rb')
voc = pickle.load(f)
f.close()

src = imagesearch.Searcher('testImaAdd.db',voc)
locs,descr = sift.read_features_from_file(featlist[0])
iw = voc.project(descr)

print 'ask using a histogram...'
#获取imlist[0]的前十幅候选图像
print src.candidates_from_histogram(iw)[:10]

src = imagesearch.Searcher('testImaAdd.db',voc)
print 'try a query...'

nbr_results = 12
res = [w[1] for w in src.query(imlist[0])[:nbr_results]]
imagesearch.plot_results(src,res)

```

7.4 建立演示程序及Web应用

```

# -*- coding: utf-8 -*-
import cherrypy
import pickle
import urllib
import os
from numpy import *
#from PCV.tools.imtools import get_imlist
from PCV.imagesearch import imagesearch

"""
This is the image search demo in Section 7.6.
"""

```

```
class SearchDemo:

    def __init__(self):
        # 载入图像列表
        self.path = './first500/'
        #self.path = 'D:/python_web/isoutu/first500/'
        self.imlist = [os.path.join(self.path,f) for f in os.listdir(self.path) if f.endswith('.jpg')]
        #self.imlist = get_imlist('./first500/')
        #self.imlist = get_imlist('E:/python/isoutu/first500/')
        self.nbr_images = len(self.imlist)
        self.ndx = range(self.nbr_images)

        # 载入词汇
        f = open('./first500/vocabulary.pkl', 'rb')
        self.voc = pickle.load(f)
        f.close()

        # 显示搜索返回的图像数
        self.maxres = 49

        # header and footer html
        self.header = """
            <!doctype html>
            <head>
                <title>Image search</title>
            </head>
            <body>
            """

        self.footer = """
            </body>
            </html>
            """

    def index(self, query=None):
        self.src = imagesearch.Searcher('testImaAdd.db', self.voc)

        html = self.header
        html += """
            <br />
            Click an image to search. <a href='?query='> Random selection </a> of images.
            <br /><br />
            """

        if query:
            # query the database and get top images
            #查询数据库，并获取前面的图像
            res = self.src.query(query)[:self.maxres]
            for dist, ndx in res:
```

```

        imname = self.src.get_filename(ndx)
        html += "<a href='?query="+imname+"' >"
        html += "<img src='"+imname+"' width='200' />"
        html += "</a>"

    # show random selection if no query
    # 如果没有查询图像则随机显示一些图像

else:
    random.shuffle(self.ndx)
    for i in self.ndx[:self.maxres]:
        imname = self.imlist[i]
        html += "<a href='?query="+imname+"' >"
        html += "<img src='"+imname+"' width='200' />"
        html += "</a>"

    html += self.footer
    return html

index.exposed = True

#conf_path = os.path.dirname(os.path.abspath(__file__))
#conf_path = os.path.join(conf_path, "service.conf")
#cherrypy.config.update(conf_path)
#cherrypy.quickstart(SearchDemo())

cherrypy.quickstart(SearchDemo(), '/', config=os.path.join(os.path.dirname(__file__),
'service.conf'))

```

7.5 配置service.conf

```

[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 10
tools.sessions.on = True
[/]
tools.staticdir.root = "E:/python/isoutu"
[/first500]
tools.staticdir.on = True
tools.staticdir.dir = "first500"

```

第八章 图像类容分类

8.1 K最近邻

K最近邻是分类中最简单且常用的方法之一。

8.1.1 一个简单的二维例子

```
# -*- coding: utf-8 -*-
from numpy.random import randn
import pickle
from pylab import *

# create sample data of 2D points
n = 200
# two normal distributions
class_1 = 0.6 * randn(n,2)
class_2 = 1.2 * randn(n,2) + array([5,1])
labels = hstack((ones(n),-ones(n)))

# save with Pickle
#with open('points_normal.pkl', 'w') as f:
with open('points_normal.pkl', 'w') as f:
    pickle.dump(class_1,f)
    pickle.dump(class_2,f)
    pickle.dump(labels,f)

# normal distribution and ring around it
class_1 = 0.6 * randn(n,2)
r = 0.8 * randn(n,1) + 5
angle = 2*pi * randn(n,1)
class_2 = hstack((r*cos(angle),r*sin(angle)))
labels = hstack((ones(n),-ones(n)))

# save with Pickle
#with open('points_ring.pkl', 'w') as f:
with open('points_ring.pkl', 'w') as f:
    pickle.dump(class_1,f)
    pickle.dump(class_2,f)
    pickle.dump(labels,f)
```

```

# -*- coding: utf-8 -*-
import pickle
from pylab import *
from PCV.classifiers import knn
from PCV.tools import imtools

pklist=['points_normal.pkl','points_ring.pkl']

figure()

# Load 2D points using Pickle
for i, pklfile in enumerate(pklist):
    with open(pklfile, 'r') as f:
        class_1 = pickle.load(f)
        class_2 = pickle.load(f)
        labels = pickle.load(f)

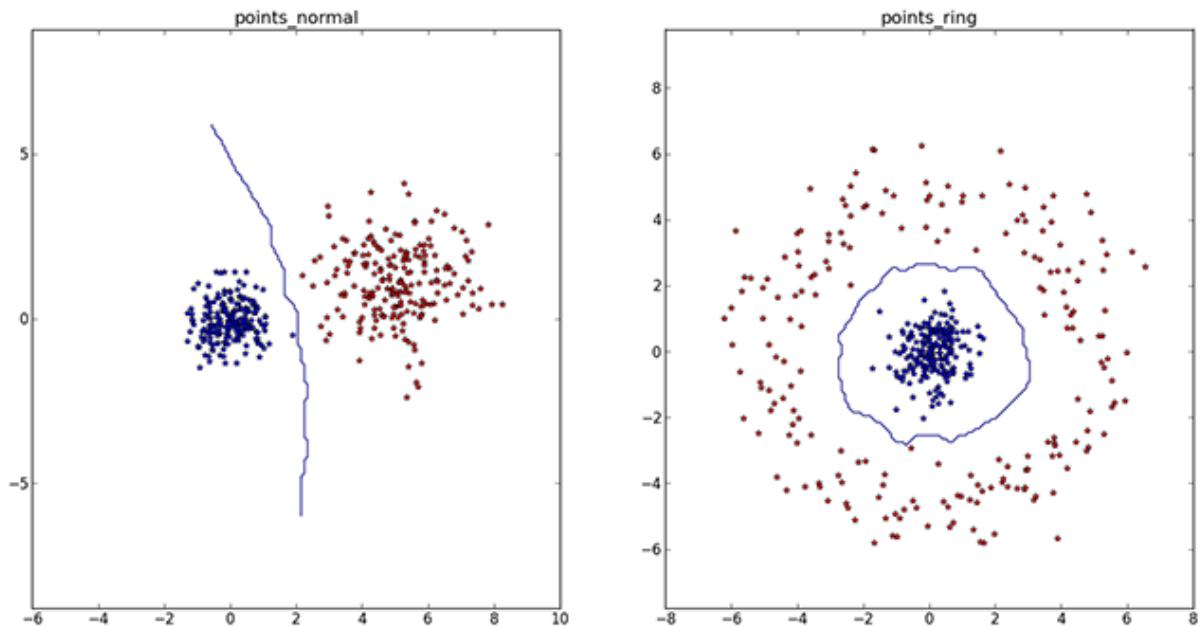
# Load test data using Pickle
with open(pklfile[:-4]+'_test.pkl', 'r') as f:
    class_1 = pickle.load(f)
    class_2 = pickle.load(f)
    labels = pickle.load(f)

model = knn.KnnClassifier(labels,vstack((class_1,class_2)))
# test on the first point
print model.classify(class_1[0])

#define function for plotting
def classify(x,y,model=model):
    return array([model.classify([xx,yy]) for (xx,yy) in zip(x,y)])

# let the classification boundary
subplot(1,2,i+1)
imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
titlename=pklfile[:-4]
title(titlename)
show()

```

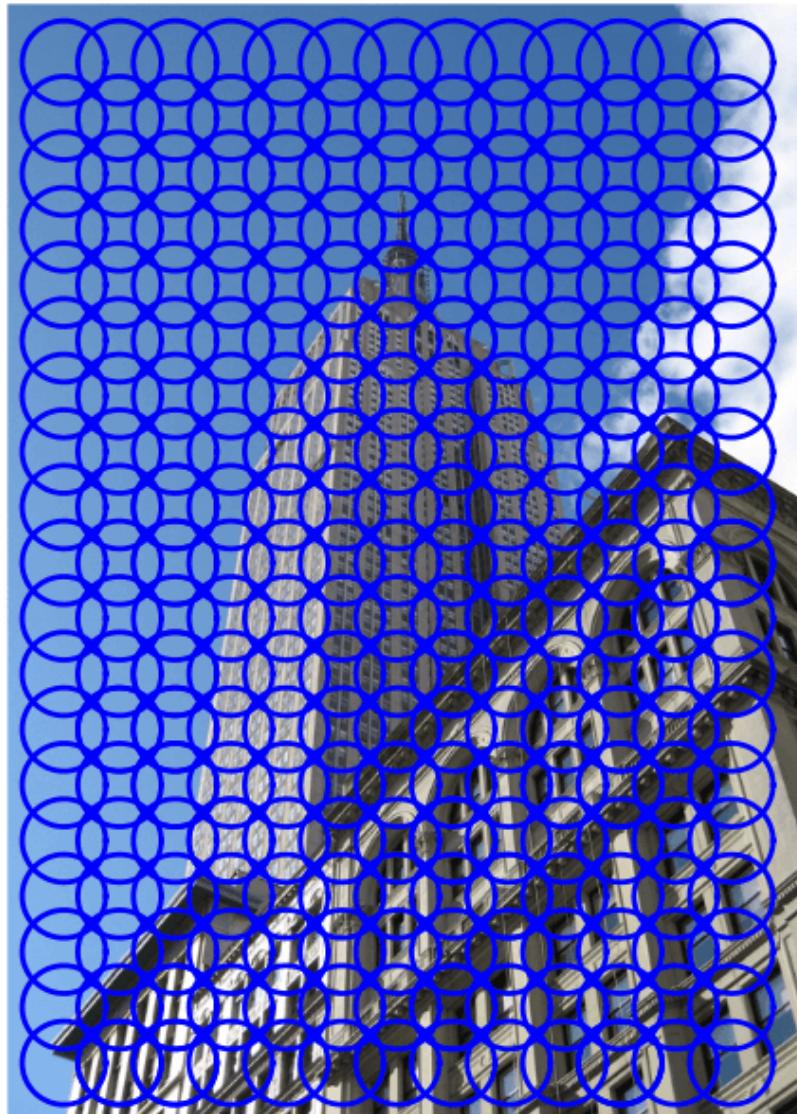


8.1.2 图像稠密(dense)sift特征

```
# -*- coding: utf-8 -*-
from PCV.localdescriptors import sift, dsift
from pylab import *
from PIL import Image

dsift.process_image_dsift('../data/empire.jpg','empire.dsift',90,40,True)
l,d = sift.read_features_from_file('empire.dsift')
im = array(Image.open('../data/empire.jpg'))
sift.plot_features(im,l,True)
title('dense SIFT')
show()
```

dense SIFT



8.1.3 图像分类——手势识别

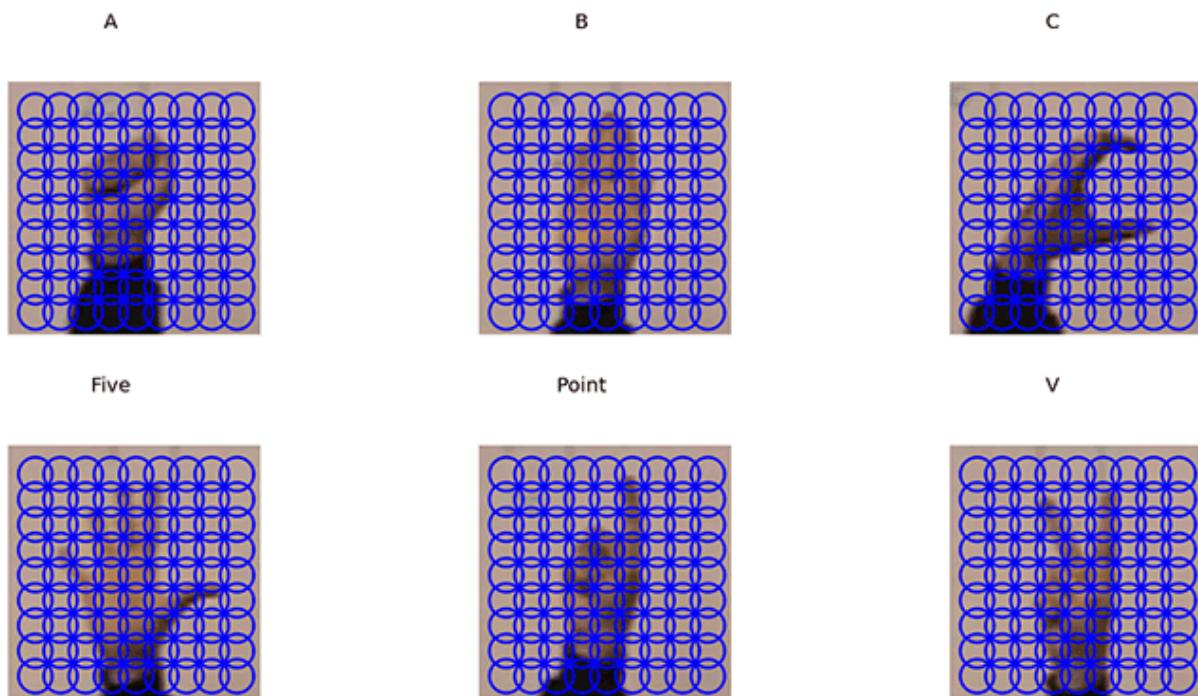
```

# -*- coding: utf-8 -*-
import os
from PCV.localdescriptors import sift, dsift
from pylab import *
from PIL import Image

imlist=['../data/gesture/train/A-uniform01.ppm','../data/gesture/train/B-uniform01.ppm',
        '../data/gesture/train/C-uniform01.ppm','../data/gesture/train/Five-uniform01.ppm',
        '../data/gesture/train/Point-uniform01.ppm','../data/gesture/train/V-uniform01.ppm']

figure()
for i, im in enumerate(imlist):
    dsift.process_image_dsift(im,im[:-3]+'_dsift',90,40,True)
    l,d = sift.read_features_from_file(im[:-3]+'_dsift')
    dirpath, filename=os.path.split(im)
    im = array(Image.open(im))
    #显示手势含义title
    titlename=filename[:-14]
    subplot(2,3,i+1)
    sift.plot_features(im,l,True)
    title(titlename)
show()

```



```

# -*- coding: utf-8 -*-
from PCV.localdescriptors import dsift
import os
from PCV.localdescriptors import sift
from pylab import *
from PCV.classifiers import knn

def get_imagelist(path):

```

```

""" Returns a list of filenames for
all jpg images in a directory. """
return [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.ppm')]

def read_gesture_features_labels(path):
    # create list of all files ending in .dsift
    featlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.dsift')]
    # read the features
    features = []
    for featfile in featlist:
        l,d = sift.read_features_from_file(featfile)
        features.append(d.flatten())
    features = array(features)
    # create labels
    labels = [featfile.split('/')[-1][0] for featfile in featlist]
    return features,array(labels)

def print_confusion(res,labels,classnames):
    n = len(classnames)
    # confusion matrix
    class_ind = dict([(classnames[i],i) for i in range(n)])
    confuse = zeros((n,n))
    for i in range(len(test_labels)):
        confuse[class_ind[res[i]],class_ind[test_labels[i]]] += 1
    print 'Confusion matrix for'
    print classnames
    print confuse

filelist_train = get_imagelist('../data/gesture/train')
filelist_test = get_imagelist('../data/gesture/test')
imlist=filelist_train+filelist_test

# process images at fixed size (50,50)
for filename in imlist:
    featfile = filename[:-3] + 'dsift'
    dsift.process_image_dsift(filename,featfile,10,5,resize=(50,50))

features,labels = read_gesture_features_labels('../data/gesture/train/')
test_features,test_labels = read_gesture_features_labels('../data/gesture/test/')
classnames = unique(labels)

# test kNN
k = 1
knn_classifier = knn.KnnClassifier(labels,features)
res = array([knn_classifier.classify(test_features[i],k) for i in
range(len(test_labels))])
# accuracy
acc = sum(1.0*(res==test_labels)) / len(test_labels)

```

```
print 'Accuracy:', acc  
  
print_confusion(res,test_labels,classnames)
```

```
Accuracy: 0.813471502591  
Confusion matrix for  
['A' 'B' 'C' 'F' 'P' 'V']  
[[ 26.  0.  2.  0.  1.  1.]  
 [ 0.  26.  0.  1.  1.  1.]  
 [ 0.  0.  26.  0.  0.  1.]  
 [ 0.  3.  0.  37.  0.  0.]  
 [ 0.  1.  2.  0.  17.  1.]  
 [ 3.  1.  3.  0.  14.  25.]]
```

第七章 ([chapter7.html](#)) 已经实现了注册新用户的功能，本章我们要为已注册的用户提供登录和退出功能。实现登录功能之后，就可以根据登录状态和当前用户的身份定制网站的内容了。例如，本章我们会更新网站的头部，显示“登录”或“退出”链接，以及到个人资料页面的链接；在第十章中，会根据当前登录用户的 id 创建关联到这个用户的微博；在第十一章，我们会实现当前登录用户关注其他用户的功能，实现之后，在首页就可以显示被关注用户发表的微博了。

实现登录功能之后，还可以实现一种安全机制，即根据用户的身份限制可以访问的页面，例如，在第九章 ([chapter9.html](#)) 中会介绍如何实现只有登入的用户才能访问编辑用户资料的页面。登录系统还可以赋予管理员级别的用户特别的权限，例如删除用户（也会在第九章 ([chapter9.html](#)) 中实现）等。

实现验证系统的功能之后，我们会简要的介绍一下 Cucumber 这个流行的行为驱动开发（Behavior-driven Development, BDD）系统，使用 Cucumber 重新实现之前的一些 RSpec 集成测试，看一下这两种方式有何不同。

和之前的章节一样，我们会在一个新的从分支中工作，本章结束后再将其合并到主分支中：

```
$ git checkout -b sign-in-out
```

8.1 session 和登录失败

[session] ([http://en.wikipedia.org/wiki/Session_\(computerscience\)](http://en.wikipedia.org/wiki/Session_(computerscience))) 是两台电脑（例如运行有网页浏览器的客户端电脑和运行 Rails 的服务器）之间的半永久性连接，我们就是利用它来实现“登录”这一功能的。网络中常见的 session 处理方式有好几种：可以在用户关闭浏览器后清除 session；也可以提供一个“记住我”单选框让用户选择永远保存，直到用户退出后 session 才会失效。¹ 在示例程序中我们选择使用第二种处理方式，即用户登录后，会永久的记住登录状态，直到用户点击“退出”链接之后才清除 session。（在 [8.2.1](#) 节中会介绍“永久”到底有多久。）

很显然，我们可以把 session 视作一个符合 REST 架构的资源，在登录页面中准备一个新的 session，登录后创建这个 session，退出则会销毁 session。不过 session 和 Users 资源有所不同，Users 资源使用数据库（通过 User 模型）持久的存储数据，而 Sessions 资源是利用 [cookie](http://en.wikipedia.org/wiki/HTTP_cookie) (http://en.wikipedia.org/wiki/HTTP_cookie) 来存储数据的。cookie 是存储在浏览器中的简单文本。实现登录功能基本上就是在实现基于 cookie 的验证机制。在本节及接下来的一节中，我们会构建 Sessions 控制器，创建登录表单，还会实现控制器中相关的动作。在 [8.2](#) 节中会加入处理 cookie 所需的代码。

8.1.1 Sessions 控制器

登录和退出功能其实是由 Sessions 控制器中相应的动作处理的，登录表单在 new 动作中处理（本节的内容），登录的过程就是向 create 动作发送 POST 请求（[8.1](#) 节和 [8.2](#) 节），退出则是向 destroy 动作发送 DELETE 请求（[8.2.6](#) 节）。

（HTTP 请求和 REST 动作之间的对应关系可以查看表格 [7.1 \(chapter7.html#sec-7-1\)](#)。）首先，我们要生成 Sessions 控制器，以及验证系统所需的集成测试：

```
$ rails generate controller Sessions --no-test-framework  
$ rails generate integration_test authentication_pages
```

参照 7.2 节 ([chapter7.html#sec-7-2](#)) 中的“注册”页面，我们要创建一个登录表单，用来生成新的 session。注册表单的构思图如图 8.1 所示。

“登录”页面的地址由 `signin_path` (稍后定义) 获取，和之前一样，我们要先编写相应的测试，如代码 8.1 所示。（可以和代码 7.6 中对“注册”页面的测试比较一下。）



图 8.1：注册表单的构思图

代码 8.1 对 new 动作和对应视图的测试

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do

  subject { page }

  describe "signin page" do
    before { visit signin_path }

    it { should have_selector('h1',      text: 'Sign in') }
    it { should have_selector('title',   text: 'Sign in') }
  end
end
```

现在测试是失败的：

```
$ bundle exec rspec spec/
```

要让代码 8.1 中的测试通过，首先，我们要为 `Sessions` 资源设置路由，还要修改“登录”页面具名路由的名称，将其映射到 `Sessions` 控制器的 `new` 动作上。和 `Users` 资源一样，我们可以使用 `resources` 方法设置标准的 REST 动作：

```
resources :sessions, only: [:new, :create, :destroy]
```

因为我们没必要显示或编辑 `session`，所以我们对动作的种类做了限制，为 `resources` 方法指定了 `:only` 选项，只创建 `new`、`create` 和 `destroy` 动作。最终的结果，包括登录和退出具名路由的设置，如代码 8.2 所示。

代码 8.2 设置 session 相关的路由

`config/routes.rb`

```

SampleApp::Application.routes.draw do
  resources :users
  resources :sessions, only: [:new, :create, :destroy]

  match '/signup', to: 'users#new'
  match '/signin', to: 'sessions#new'
  match '/signout', to: 'sessions#destroy', via: :delete
  .
  .
  .

end

```

注意，设置退出路由那行使用了 `via :delete`，这个参数指明 `destroy` 动作要使用 DELETE 请求。

代码 8.2 中的路由设置会生成类似 [表格 7.1 \(chapter7.html#table-7-1\)](#) 所示的 URI 地址和动作的对应关系，如 [表格 8.1](#) 所示。注意，我们修改了登录和退出具名路由，而创建 session 的路由还是使用默认值。

HTTP 请求	URI 地址	具名路由	动作	目的
GET	/signin	signin_path	new	创建新 session 的页面（登录）
POST	/sessions	sessions_path	create	创建 session
DELETE	/signout	signout_path	destroy	删除 session（退出）

表格 8.1：代码 8.2 中的设置生成的符合 REST 架构的路由关系

为了让代码 8.1 中的测试通过，我们还要在 Sessions 控制器中加入 `new` 动作，相应的代码如代码 8.3 所示（同时也定义了 `create` 和 `destroy` 动作）。

代码 8.3 没什么内容的 Sessions 控制器

`app/controllers/sessions_controller.rb`

```

class SessionsController < ApplicationController
  def new
  end

  def create
  end

  def destroy
  end
end

```

接下来还要创建“登录”页面的视图，因为“登录”页面的目的是创建新 session，所以创建的视图位于 `app/views/sessions/new.html.erb`。在视图中我们要显示网页的标题和一个一级标头，如代码 8.4 所示。

代码 8.4 “登录”页面的视图

`app/views/sessions/new.html.erb`

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>
```

现在代码 8.1 中的测试应该可以通过了，接下来我们要编写登录表单。

```
$ bundle exec rspec spec/
```

8.1.2 测试登录功能

对比图 8.1 和图 7.11 之后，我们发现登录表单和注册表单外观上差不多，只是少了两个字段，只有 Email 地址和密码字段。和注册表单一样，我们可以使用 Capybara 填写表单，再点击按钮进行测试。

在测试的过程中，我们不得不向程序中加入相应功能，这也正是 TDD 带来的好处之一。我们先来测试填写不合法数据的登录过程，构思图如图 8.2 所示。

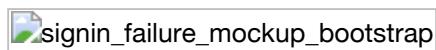


图 8.2：注册失败页面的构思图

从图 8.2 我们可以看出，如果提交的数据不正确，我们会重新渲染“注册”页面，还会显示一个错误提示消息。这个错误提示是 Flash 消息，我们可以通过下面的测试验证：

```
it { should have_selector('div.alert.alert-error', text: 'Invalid') }
```

(在第七章 ([chapter7.html](#)) 练习中的代码 7.32 中出现过类似的代码。) 我们要查找的元素是：

```
div.alert.alert-error
```

前面介绍过，这里的点号代表 CSS 中的 class（参见 [5.1.2 节 \(chapter5.html#sec-5-1-2\)](#)），你也许猜到了，这里我们要查找的是同时具有 alert 和 alert-error class 的 div 元素。而且我们还检测了错误提示消息中是否包含了 "Invalid" 这个词。所以，上述测试是检测页面中是否有下面这个元素的：

```
<div class="alert alert-error">Invalid...</div>
```

代码 8.5 是针对标题和 Flash 消息的测试。我们可以看出，这些代码缺少了一个很重要的部分，会在 [8.1.5 节](#) 中说明。

代码 8.5 登录失败时的测试

```
spec/requests/authentication_pages_spec.rb
```

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_selector('title', text: 'Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }
    end
  end
end

```

测试了登录失败的情况，下面我们要测试登录成功的情况了。我们要测试登录成功后是否转向了用户资料页面（从页面的标题判断，标题中应该包含用户的名字），还要测试网站的导航中是否有以下三个变化：

1. 出现了指向用户资料页面的链接
2. 出现了“退出”链接
3. “登录”链接消失了

（对“设置（Settings）”链接的测试会在 [9.1 节 \(chapter9.html#9-1\)](#) 中实现，对“所有用户（Users）”链接的测试会在 [9.3 节 \(chapter9.html#sec-9-3\)](#) 中实现。）如上变化的构思图如图 8.3 所示。²注意，“退出”和“个人资料”链接位于“账户（Account）”下拉菜单中。在 [8.2.4 节](#) 中会介绍如何通过 Bootstrap 实现这种下拉菜单。

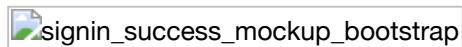


图 8.3：登录成功后显示的用户资料页面构思图

对登录成功时的测试如代码 8.6 所示。

代码 8.6 登录成功时的测试

`spec/requests/authentication_pages_spec.rb`

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    before { visit signin_path }

    .
    .
    .

    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        fill_in "Email", with: user.email
        fill_in "Password", with: user.password
        click_button "Sign in"
      end

      it { should have_selector('title', text: user.name) }
      it { should have_link('Profile', href: user_path(user)) }
      it { should have_link('Sign out', href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
    end
  end
end

```

在代码 8.6 中用到了 `have_link` 方法，它的第一参数是链接文本，第二个参数是可选的 `:href`，指定链接的地址，因此如下的代码

```
it { should have_link('Profile', href: user_path(user)) }
```

确保了页面中有一个 `a` 元素，链接到指定的 URI 地址。这里我们要检测的是一个指向用户资料页面的链接。

8.1.3 登录表单

写完测试之后，我们就可以创建登录表单了。在代码 7.17 中，注册表单使用了 `form_for` 帮助函数，并指定其参数为 `@user` 变量：

```

<%= form_for(@user) do |f| %>
  .
  .
  .
<% end %>

```

注册表单和登录表单的区别在于，程序中没有 Session 模型，因此也就没有类似 `@user` 的变量。也就是说，在构建登录表单时，我们要给 `form_for` 提供更多的信息。一般来说，如下的代码

```
form_for(@user)
```

Rails 会自动向 /users 地址发送 POST 请求。对于登录表单，我们则要明确的指定资源的名称以及相应的 URI 地址：

```
form_for(:session, url: sessions_path)
```

(创建表单还有另一种方法，不用 `form_for`，而用 `form_tag`。`form_tag` 也是 Rails 程序常用的方法，不过换用 `form_tag` 之后就和注册表单有很多不同之处了，我现在是想使用相似的代码构建登录表单。使用 `form_tag` 构建登录表单会留作练习（参见 8.5 节）。

使用上述这种 `form_for` 形式，参照代码 7.17 中的注册表单，很容易的就能编写一个符合图 8.1 的登录表单，如代码 8.7 所示。

代码 8.7 注册表单的代码

app/views/sessions/new.html.erb

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(:session, url: sessions_path) do |f| %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.submit "Sign in", class: "btn btn-large btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

注意，为了访客的便利，我们还加入了到“注册”页面的链接。代码 8.7 中的登录表单效果如图 8.4 所示。



图 8.4：登录表单 ([/signup \(http://localhost:3000/signup\)](#))

用的多了你就不会老是查看 Rails 生成的 HTML（你会完全信任所用的帮助函数可以正确的完成任务），不过现在还是来看一下登录表单的 HTML 吧（如代码 8.8 所示）。

代码 8.8 代码 8.7 中登录表单生成的 HTML

```

<form accept-charset="UTF-8" action="/sessions" method="post">
  <div>
    <label for="session_email">Email</label>
    <input id="session_email" name="session[email]" size="30" type="text" />
  </div>
  <div>
    <label for="session_password">Password</label>
    <input id="session_password" name="session[password]" size="30"
           type="password" />
  </div>
  <input class="btn btn-large btn-primary" name="commit" type="submit"
         value="Sign in" />
</form>

```

你可以对比一下代码 8.8 和代码 7.20。你可能已经猜到了，提交登录表单后会生成一个 `params` Hash，其中 `params[:session][:email]` 和 `params[:session][:password]` 分别对应了 Email 和密码字段。

8.1.4 分析表单提交

和创建用户类似，创建 session 时先要处理提交不合法数据的情况。我们已经编写了对提交不合法数据的测试（参见代码 8.5），也添加了有几处难理解但还算简单的代码让测试通过了。下面我们就来分析一下表单提交的过程，然后为登录失败添加失败提示信息（如图 8.2）。最后，以此为基础，验证提交的 Email 和密码，处理登录成功的情况（参见 [8.2 节](#)）。

首先，我们来编写 Sessions 控制器的 `create` 动作，如代码 8.9 所示，现在只是直接渲染登录页面。在浏览器中访问 `/sessions/new`，然后提交空表单，显示的页面如图 8.5 所示。

代码 8.9 Sessions 控制器中 `create` 动作的初始版本

`app/controllers/sessions_controller.rb`

```

class SessionsController < ApplicationController
  .
  .
  .
  def create
    render 'new'
  end
  .
  .
  .
end

```



图 8.5：代码 8.9 中的 `create` 动作显示的登录失败后的页面

仔细看一下图 8.5 中显示的调试信息，你会发现，如在 [8.1.3 节](#) 末尾说过的，表单提交后会生成 `params` Hash，Email 和密码都在 `:session` 键中：

```
---
```

```
session:  
  email: ''  
  password: ''  
commit: Sign in  
action: create  
controller: sessions
```

和注册表单类似，这些参数是一个嵌套的 Hash，在代码 4.6 中见过。params 包含了如下的嵌套 Hash：

```
{ session: { password: "", email: "" } }
```

也就是说

```
params[:session]
```

本身就是一个 Hash：

```
{ password: "", email: "" }
```

所以，

```
params[:session][:email]
```

就是提交的 Email 地址，而

```
params[:session][:password]
```

就是提交的密码。

也就是说，在 `create` 动作中，`params` 包含了使用 Email 和密码验证用户身份所需的全部数据。幸运的是，我们已经定义了身份验证过程中所需的两个方法，即由 Active Record 提供的 `User.find_by_email`（参见 [6.1.4 节](#) [\(chapter6.html#sec-6-1-4\)](#)），以及由 `has_secure_password` 提供的 `authenticate` 方法（参见 [6.3.3 节](#) [\(chapter6.html#sec-6-3-3\)](#)）。我们之前介绍过，如果提交的数据不合法，`authenticate` 方法会返回 `false`。基于以上的分析，我们计划按照如下的方式实现用户登录功能：

```
def create  
  user = User.find_by_email(params[:session][:email].downcase)  
  if user && user.authenticate(params[:session][:password])  
    # Sign the user in and redirect to the user's show page.  
  else  
    # Create an error message and re-render the signin form.  
  end  
end
```

`create` 动作的第一行，使用提交的 Email 地址从数据库中取出相应的用户。第二行是 Ruby 中经常使用的语句形式：

```
user && user.authenticate(params[:session][:password])
```

我们使用 `&&`（逻辑与）检测获取的用户是否合法。因为除了 `nil` 和 `false` 之外的所有对象都被视作 `true`，上面这个语句可能出现的结果如表格 8.2 所示。我们可以从表格 8.2 中看出，当且仅当数据库中存在提交的 Email 并提交了对应的密码时，这个语句才会返回 `true`。

用户	密码	<code>a && b</code>
不存在	任意值	<code>nil && [anything] == false</code>
存在	错误的密码	<code>true && false == false</code>
存在	正确的密码	<code>true && true == true</code>

表格 8.2: `user && user.authenticate(...)` 可能出现的结果

8.1.5 显示 Flash 消息

在 7.3.2 节 ([chapter7.html#sec-7-3-2](#)) 中，我们使用 User 模型的数据验证信息来显示注册失败时的提示信息。这些错误提示信息是关联在某个 Active Record 对象上的，不过这种方式不可以在 session 上，因为 session 不是 Active Record 模型。我们要采取的方法是，在登录失败时，把错误提示信息赋值给 Flash 消息。代码 8.10 显示的是我们首次尝试实现这种方法所用的代码，其中有个小小的错误。

代码 8.10 尝试处理登录失败（有个小小的错误）

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by_email(params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Sign the user in and redirect to the user's show page.
    else
      flash[:error] = 'Invalid email/password combination' # Not quite right!
      render 'new'
    end
  end

  def destroy
  end
end
```

布局中已经加入了显示 Flash 消息的局部视图，所以无需其他修改，上述 Flash 错误提示消息就会显示出来，而且因为使用了 Bootstrap，这个错误消息的样式也很美观（如图 8.6）。

图 8.6：登录失败后显示的 Flash 消息

不过，就像代码 8.10 中的注释所说，这些代码还有问题。显示的页面看起来很正常啊，那么，问题出现在哪儿呢？问题的关键在于，Flash 消息在一个请求的生命周期内是持续存在的，而重新渲染页面（使用 `render` 方法）和代码 7.27 中的转向不同，它不算新的请求，你会发现这个 Flash 消息存在的时间比设想的要长很多。例如，我们提交了不合法的登录信息，Flash 消息生成了，然后在登录页面中显示出来（如图 8.6），这时如果我们点击链接转到其他页面（例如“首页”），这只不过是表单提交后的第一次请求，所以页面中还是会显示 Flash 消息（如图 8.7）。

图 8.7：仍然显示有 Flash 消息的页面

Flash 消息没有按预期消失算是程序的一个 bug，在修正之前，我们最好编写一个测试来捕获这个错误。现在，登录失败时的测试是可以通过的：

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "signin with invalid information"
```

不过程序中有错误，测试应该是失败的，所以我们要编写一个能够捕获这种错误的测试。幸好，捕获这种错误正是集成测试的拿手好戏，所用的代码如下：

```
describe "after visiting another page" do
  before { click_link "Home" }
  it { should_not have_selector(
```

第九章 图像分割

9.1 更新用户

9.1.1 编辑表单

9.1.2 编辑失败

9.1.3 编辑成功

9.2 权限限制

9.2.1 必须先登录

9.2.2 用户只能编辑自己的资料

9.2.3 更友好的转向

9.3 列出所有用户

9.3.1 用户列表

9.3.2 示例用户

9.3.3 分页

9.3.4 视图重构

9.4 删除用户

9.4.1 管理员

9.4.2 destroy 动作

9.5 小结

9.6 练习

图像分割是将一幅图像分割成有意义区域的过程。区域可以是图像的前景与背景或者单个对象。这些区域可以利用诸如颜色、边线或近邻相似性等特征构建。本章中，我们将看到一些不同的分割技术。

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

gr = digraph()
gr.add_nodes([0,1,2,3])
gr.add_edge((0,1), wt=4)
gr.add_edge((1,2), wt=3)
gr.add_edge((2,3), wt=5)
gr.add_edge((0,2), wt=3)
gr.add_edge((1,3), wt=4)
flows,cuts = maximum_flow(gr,0,3)
print 'flow is:', flows
print 'cut is:', cuts
```

```
flow is: {(0, 1): 4, (1, 2): 0, (1, 3): 4, (2, 3): 3, (0, 2): 3}
cut is: {0: 0, 1: 1, 2: 1, 3: 1}
```

```
from scipy.misc import imresize
from PCV.tools import graphcut
from PIL import Image
from pylab import *

im = array(Image.open("../data/empire.jpg"))
im = imresize(im, 0.07)
size = im.shape[:2]

# add two rectangular training regions
labels = zeros(size)
labels[3:18, 3:18] = -1
labels[-18:-3, -18:-3] = 1

# create graph
g = graphcut.build_bayes_graph(im, labels, kappa=1)

# cut the graph
res = graphcut.cut_graph(g, size)

figure()
graphcut.show_labeling(im, labels)

figure()
imshow(res)
gray()
axis('off')
show()
```

P203

```

from PCV.tools import rof
from pylab import *
from PIL import Image
import scipy.misc

#im = array(Image.open('../data/ceramic-houses_t0.png').convert("L"))
im = array(Image.open('../data/flower32_t0.png').convert("L"))
figure()
gray()
subplot(131)
axis('off')
imshow(im)

U, T = rof.denoise(im, im, tolerance=0.001)
subplot(132)
axis('off')
imshow(U)

#t = 0.4 # ceramic-houses_t0 threshold
t = 0.8 # flower32_t0 threshold
seg_im = U < t*U.max()
#scipy.misc.imsave('ceramic-houses_t0_result.pdf', seg_im)
scipy.misc.imsave('flower32_t0_result.pdf', seg_im)
subplot(133)
axis('off')
imshow(seg_im)

show()

```





本章我们要完成表格 7.1 ([chapter7.html#table-7-1](#))所示的Users 资源，添加 `edit`、`update`、`index` 和 `destroy` 动作。首先我们要实现更新用户个人资料的功能，实现这样的功能自然要依靠安全验证系统（基于第八章 ([chapter8.html](#))中实现的权限限制）。然后要创建一个页面列出所有的用户（也需要权限限制），期间会介绍示例数据和分页功能。最后，我们还要实现删除用户的功能，从数据库中删除用户记录。我们不会为所有用户都提供这种强大的权限，而是会创建管理员，授权他们来删除用户。

在开始之前，我们要新建 `updating-users` 分支：

```
$ git checkout -b updating-users
```

9.1 更新用户

编辑用户信息的方法和创建新用户差不多（参见第七章 ([chapter7.html](#))），创建新用户的页面是在 `new` 动作中处理的，而编辑用户的页面则是在 `edit` 动作中；创建用户的过程是在 `create` 动作中处理了 `POST` 请求，而编辑用户要在 `update` 动作中处理 `PUT` 请求（`HTTP` 请求参见旁注 3.2 ([chapter3.html#sec-3-2](#))）。二者之间最大的区别是，任何人都可以注册，但只有当前用户才能更新他自己的信息。所以我们就要限制访问，只有授权的用户才能编辑更新资料，我们可以利用第八章 ([chapter8.html](#))实现的身份验证机制，使用”事前过滤器（before filter）“实现访问限制。

9.1.1 编辑表单

我们先来创建编辑表单，其构思图如图 9.1 所示。¹和之前一样，我们要先编写测试。注意构思图中修改 Gravatar 头像的链接，如果你浏览过 Gravatar 的网站，可能就知道上传和编辑头像的地址是 <http://gravatar.com/emails>，我们就来测试编辑页面中有没有一个链接指向了这个地址。²



图 9.1：编辑用户页面的构思图

对编辑用户表单的测试和第七章练习中的代码 7.31 类似，同样也测试了提交不合法数据后是否会显示错误提示信息，如代码 9.1 所示。

代码 9.1 用户编辑页面的测试

```
spec/requests/user_pages_spec.rb
```

```

require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit edit_user_path(user) }

    describe "page" do
      it { should have_selector('h1',    text: "Update your profile") }
      it { should have_selector('title', text: "Edit user") }
      it { should have_link('change', href: 'http://gravatar.com/emails') }
    end

    describe "with invalid information" do
      before { click_button "Save changes" }
      it { should have_content('error') }
    end
  end
end

```

程序所需的代码要放在 `edit` 动作中，我们在表格 7.1 (chapter7.html#table-7-1) 中列出了，用户编辑页面的地址是 `/users/1/edit`（假设用户的 id 是 1）。我们介绍过用户的 id 是保存在 `params[:id]` 中的，所以我们可以按照代码 9.2 所示的方法查找用户。

代码 9.2 Users 控制器的 `edit` 方法

`app/controllers/users_controller.rb`

```

class UsersController < ApplicationController
  .
  .
  .
  def edit
    @user = User.find(params[:id])
  end
end

```

要让测试通过，我们就要编写编辑用户页面的视图，如代码 9.3 所示。仔细观察一下视图代码，它和代码 7.17 中创建新用户页面的视图代码很相似，这就暗示我们要进行重构，把重复的代码移入局部视图。重构会留作练习，详情参见 9.6 节。

代码 9.3 编辑用户页面的视图

`app/views/users/edit.html.erb`

```

<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.label :password_confirmation, "Confirm Password" %>
      <%= f.password_field :password_confirmation %>

      <%= f.submit "Save changes", class: "btn btn-large btn-primary" %>
    <% end %>

    <%= gravatar_for @user %>
    <a href="http://gravatar.com/emails">change</a>
  </div>
</div>

```

在这段代码中我们再次使用了 [7.3.2 节 \(chapter7.html#sec-7-3-2\)](#) 中创建的 `error_messages` 局部视图。

添加了视图代码，再加上代码 9.2 中定义的 `@user` 变量，代码 9.1 中的“编辑页面”测试应该就可以通过了：

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "edit page"
```

编辑用户页面如图 9.2 所示，我们看到 Rails 会自动读取 `@user` 变量，预先填好了名字和 Email 地址字段。



图 9.2：编辑用户页面，名字和 Email 地址字段已经自动填好了

查看一下编辑用户页面的源码，我们可以发现的确生成了一个 `form` 元素，参见代码 9.4。

代码 9.4 编辑表单的 HTML

```
<form action="/users/1" class="edit_user" id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="put" />
  .
  .
  .
</form>
```

留意一下其中的一个隐藏字段：

```
<input name="_method" type="hidden" value="put" />
```

因为浏览器本身并不支持发送 PUT 请求（[表格 7.1 \(chapter7.html#table-7-1\)](#)中列出的 REST 动作要用），所以 Rails 就在 POST 请求中使用这个隐藏字段伪造了一个 PUT 请求。³

还有一个细节需要注意一下，代码 9.3 和代码 7.17 都使用了相同的 `form_for(@user)` 来构建表单，那么 Rails 是怎么知道创建新用户要发送 POST 请求，而编辑用户时要发送 PUT 请求的呢？这个问题的答案是，通过 Active Record 提供的 `new_record?` 方法可以检测用户是新创建的还是已经存在于数据库中的：

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

所以在使用 `form_for(@user)` 构建表单时，如果 `@user.new_record?` 返回 `true` 则发送 POST 请求，否则就发送 PUT 请求。

最后，我们还要在导航中添加一个指向编辑用户页面的链接（“设置（Settings）”）。因为只有登录之后才会显示这个页面，所以对“设置”链接的测试要和其他的身份验证测试放在一起，如代码 9.5 所示。（如果能再测试一下没登录时不会显示“设置”链接就更完美了，这会留作练习，参见 [9.6 节](#)。）

代码 9.5 添加检测“设置”链接的测试

```
spec/requests/authentication_pages_spec.rb
```

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "with valid information" do
    let(:user) { FactoryGirl.create(:user) }
    before { sign_in user }

    it { should have_selector('title', text: user.name) }
    it { should have_link('Profile', href: user_path(user)) }
    it { should have_link('Settings', href: edit_user_path(user)) }
    it { should have_link('Sign out', href: signout_path) }
    it { should_not have_link('Sign in', href: signin_path) }
    .
    .
    .
  end
end
end

```

为了简化，代码 9.5 中使用 `sign_in` 帮助方法，这个方法的作用是访问登录页面，提交合法的表单数据，如代码 9.6 所示。

代码 9.6 用户登录帮助方法

`spec/support/utilities.rb`

```

.
.
.

def sign_in(user)
  visit signin_path
  fill_in "Email", with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
  # Sign in when not using Capybara as well.
  cookies[:remember_token] = user.remember_token
end

```

如上述代码中的注释所说，如果没有使用 Capybara 的话，填写表单的操作是无效的，所以我们就添加了一行，在不使用 Capybara 时把用户的记忆权标添加到 `cookies` 中：

```

# Sign in when not using Capybara as well.
cookies[:remember_token] = user.remember_token

```

如果直接使用 HTTP 请求方法就必须要有上面这行代码，具体的用法在代码 9.47 中有介绍。（注意，测试中使用的 `cookies` 对象和真实的 `cookies` 对象是有点不一样的，代码 8.19 中使用的 `cookies.permanent` 方法不能在测试中使用。）你可能已经猜到了，`sing_in` 在后续的测试中还会用到，而且还可以用来去除重复代码（参见 [9.6 节](#)）。

在程序中添加“设置”链接很简单，我们就直接使用表格 7.1 ([chapter7.html#table-7-1](#)) 中列出的 `edit_user_path` 具名路由，其参数设为代码 8.22 中定义的 `current_user` 帮助方法：

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

完整的代码如代码 9.7 所示。

代码 9.7 添加“设置”链接

`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

9.1.2 编辑失败

本小节我们要处理编辑失败的情况，让代码 9.1 中对错误提示信息的测试通过。我们要在 Users 控制器的 update 动作中使用 update_attributes 方法，传入提交的 params Hash，更新用户记录，如代码 9.8 所示。如果提交了不合法的数据，更新操作会返回 false，交由 else 分支处理，重新渲染编辑用户页面。我们之前用过类似的处理方式，代码结构和第一个版本的 create 动作类似（参见代码 7.21）。

代码 9.8 还不完整的 update 动作

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  .
  .
  .
  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      # Handle a successful update.
    else
      render 'edit'
    end
  end
end
```

提交不合法信息后显示了错误提示信息（如图 9.3），测试就可以通过了，你可以运行测试组件验证一下：

```
$ bundle exec rspec spec/
```



图 9.3：提交编辑表单后显示的错误提示信息

9.1.3 编辑成功

现在我们要让编辑表单能够正常使用了。编辑头像的功能已经实现了，因为我们把上传头像的操作交由 Gravatar 处理了，如需更换头像，点击图 9.2 中的“change”链接就可以了，如图 9.4 所示。下面我们来实现编辑其他信息的功能。



图 9.4：Gravatar 的剪切图片界面，上传了一个帅哥的图片

对 update 动作的测试和对 create 的测试类似。代码 9.9 介绍了如何使用 Capybara 在表单中填写合法的数据，还介绍了怎么测试提交表单的操作是否正确。测试的代码很多，你可以参考第七章 ([chapter7.html](#)) 中的测试，试一下能不能完全理解。

代码 9.9 测试 Users 控制器的 update 动作

```
spec/requests/user_pages_spec.rb
```

```

require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit edit_user_path(user) }
    .
    .
    .
    describe "with valid information" do
      let(:new_name) { "New Name" }
      let(:new_email) { "new@example.com" }
      before do
        fill_in "Name",           with: new_name
        fill_in "Email",          with: new_email
        fill_in "Password",       with: user.password
        fill_in "Confirm Password", with: user.password
        click_button "Save changes"
      end

      it { should have_selector('title', text: new_name) }
      it { should have_selector('div.alert.alert-success') }
      it { should have_link('Sign out', href: signout_path) }
      specify { user.reload.name.should == new_name }
      specify { user.reload.email.should == new_email }
    end
  end
end

```

上述代码中出现了一个新的方法 `reload`, 出现在检测用户的属性是否已经更新的测试中:

```

specify { user.reload.name.should == new_name }
specify { user.reload.email.should == new_email }

```

这两行代码使用 `user.reload` 从测试数据库中重新加载 `user` 的数据, 然后检测用户名和 Email 地址是否更新成了新的值。

要让代码 9.9 中的测试通过, 我们可以参照最终版本的 `create` 动作 (代码 8.27) 来编写 `update` 动作, 如代码 9.10 所示。我们在代码 9.8 的基础上加入了下面这三行。

```

flash[:success] = "Profile updated"
sign_in @user
redirect_to @user

```

注意, 用户资料更新成功之后我们再次登入了用户, 因为保存用户时, 重设了记忆权标 (代码 8.18), 之前的 `session` 就失

效了（代码 8.22）。这也是一项安全措施，因为如果用户更新了资料，任何会话劫持都会自动失效。

代码 9.10 Users 控制器的 update 动作

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      flash[:success] = "Profile updated"
      sign_in @user
      redirect_to @user
    else
      render 'edit'
    end
  end
end
```

注意，现在这种实现方式，每次更新数据都要提供密码（填写图 9.2 中那两个空的字段），虽然有点烦人，不过却保证了安全。

添加了本小节的代码之后，编辑用户页面应该可以正常使用了，你可以运行测试组件再确认一下，测试应该是可以通过的：

```
$ bundle exec rspec spec/
```

9.2 权限限制

第八章 (chapter8.html) 中实现的身份验证机制有一个很好的作用，可以实现权限限制。身份验证可以识别用户是否已经注册，而权限限制则可以限制用户可以进行的操作。

虽然 9.1 节中已经基本完成了 edit 和 update 动作，但是却有一个安全隐患：任何人（甚至是未登录的用户）都可以访问这两个动作，而且登录后的用户可以更新所有其他用户的资料。本节我们要实现一种安全机制，限制用户必须先登录才能更新自己的资料，而不能更新他人的资料。没有登录的用户如果试图访问这些受保护的页面，会转向登录页面，并显示一个提示信息，构思图如图 9.5 所示。



图 9.5：访问受保护页面转向后的页面构思图

9.2.1 必须先登录

因为对 edit 和 update 动作所做的安全限制是一样的，所以我们就同一个 RSpec describe 块中进行测试。我们从要求登录开始，测试代码要检测未登录的用户试图访问这两个动作时是否转向了登录页面，如代码 9.11 所示。

代码 9.11 测试 edit 和 update 动作是否处于被保护状态

spec/requests/authentication_pages_spec.rb

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .

  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "in the Users controller" do

        describe "visiting the edit page" do
          before { visit edit_user_path(user) }

          it { should have_selector('title', text: 'Sign in') }
        end

        describe "submitting to the update action" do
          before { put user_path(user) }
          specify { response.should redirect_to(signin_path) }
        end
      end
    end
  end
end

```

代码 9.11 除了使用 Capybara 的 `visit` 方法之外，还第一次使用了另一种访问控制器动作的方法：如果需要直接发起某种 HTTP 请求，则直接使用 HTTP 动词对应的方法即可，例如本例中的 `put` 发起的就是 PUT 请求：

```

describe "submitting to the update action" do
  before { put user_path(user) }
  specify { response.should redirect_to(signin_path) }
end

```

上述代码会向 `/users/1` 地址发送 PUT 请求，由 `Users` 控制器的 `update` 动作处理（参见表格 7.1 ([chapter7.html#table-7-1](#))）。我们必须这么做，因为浏览器无法直接访问 `update` 动作，必须先提交编辑表单，所以 Capybara 也做不到。访问编辑资料页面只能测试 `edit` 动作是否有权限继续操作，而不能测试 `update` 动作的授权情况。所以，如果要测试 `update` 动作是否有权限进行操作只能直接发送 PUT 请求。（你可能已经猜到了，除了 `put` 方法之外，Rails 中的测试还支持 `get`、`post` 和 `delete` 方法。）

直接发送某种 HTTP 请求时，我们需要处理更底层的 `response` 对象。和 Capybara 提供的 `page` 对象不同，我们可以使用 `response` 测试服务器的响应。本例我们检测了 `update` 动作的响应是否转向了登录页面：

```

specify { response.should redirect_to(signin_path) }

```

我们要使用 `before_filter` 方法实现权限限制，这个方法会在指定的动作执行之前，先运行指定的方法。为了实现要求用户先登录的限制，我们要定义一个名为 `signed_in_user` 的方法，然后调用 `before_filter :signed_in_user`，如代码 9.12 所示。

代码 9.12 添加 `signed_in_user` 事前过滤器

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:edit, :update]
  .
  .
  .
  private

  def signed_in_user
    redirect_to signin_path, notice: "Please sign in." unless signed_in?
  end

end
```

默认情况下，事前过滤器会应用于控制器中的所有动作，所以在上述代码中我们传入了 `:only` 参数指定只应用在 `edit` 和 `update` 动作上。

注意，在代码 9.12 中我们使用了设定 `flash[:notice]` 的简便方式，把 `redirect_to` 方法的第二个参数指定为一个 Hash。这段代码等同于：

```
flash[:notice] = "Please sign in."
redirect_to signin_path
```

(`flash[:error]` 也可以使用上述的简便方式，但 `flash[:success]` 却不可以。)

`flash[:notice]` 加上 `flash[:success]` 和 `flash[:error]` 就是我们要介绍的三种 Flash 消息，Bootstrap 为这三种消息都提供了样式。退出后再尝试访问 `/users/1/edit`，就会看到如图 9.6 所示的黄色提示框。



图 9.6：尝试访问受保护的页面后显示的登录表单

在尝试让代码 9.11 中检测权限限制的测试通过的过程中，我们却破坏了代码 9.1 中的测试。如下的代码

```
describe "edit" do
  let(:user) { FactoryGirl.create(:user) }
  before { visit edit_user_path(user) }
  .
  .
  .
```

现在会失败，因为必须先登录才能正常访问编辑用户资料页面。解决这个问题的办法是，使用代码 9.6 中定义的 `sign_in` 方法登入用户，如代码 9.13 所示。

代码 9.13 为 edit 和 update 测试加入登录所需的代码

spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .

  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before do
      sign_in user
      visit edit_user_path(user)
    end
    .
    .
    .

  end
end
```

现在所有的测试应该都可以通过了：

```
$ bundle exec rspec spec/
```

9.2.2 用户只能编辑自己的资料

当然，要求用户必须先登录还是不够的，用户必须只能编辑自己的资料。我们的测试可以这么编写，用其他用户的身份登录，然后访问 edit 和 update 动作，如代码 9.14 所示。注意，用户不应该尝试编辑其他用户的资料，我们没有转向登录页面，而是转到了网站的首页。

代码 9.14 测试只有自己才能访问 edit 和 update 动作

spec/requests/authentication_pages_spec.rb

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as wrong user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:wrong_user) { FactoryGirl.create(:user, email: "wrong@example.com") }
      before { sign_in user }

      describe "visiting Users#edit page" do
        before { visit edit_user_path(wrong_user) }
        it { should_not have_selector('title', text: full_title('Edit user')) }
      end

      describe "submitting a PUT request to the Users#update action" do
        before { put user_path(wrong_user) }
        specify { response.should redirect_to(root_path) }
      end
    end
  end
end

```

注意，创建预构件的方法还可以接受第二个参数：

```
FactoryGirl.create(:user, email: "wrong@example.com")
```

上述代码会用指定的 Email 替换默认值，然后创建用户。我们的测试要确保其他的用户不能访问原来那个用户的 `edit` 和 `update` 动作。

我们在控制器中加入了第二个事前过滤器，调用 `correct_user` 方法，如代码 9.15 所示。

代码 9.15 保护 `edit` 和 `update` 动作的 `correct_user` 事前过滤器

```
app/controllers/users_controller.rb
```

```

class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:edit, :update]
  before_filter :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update_attributes(params[:user])
      flash[:success] = "Profile updated"
      sign_in @user
      redirect_to @user
    else
      render 'edit'
    end
  end

  private

  def signed_in_user
    redirect_to signin_path, notice: "Please sign in." unless signed_in?
  end

  def correct_user
    @user = User.find(params[:id])
    redirect_to(root_path) unless current_user?(@user)
  end
end

```

上述代码中的 `correct_user` 方法使用了 `current_user?` 方法，我们要在 Sessions 帮助方法模块中定义一下，如代码 9.16。

代码 9.16 定义 `current_user?` 方法

`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def current_user
    @current_user ||= User.find_by_remember_token(cookies[:remember_token])
  end

  def current_user?(user)
    user == current_user
  end
  .
  .
  .
end
```

代码 9.15 同时也更新了 `edit` 和 `update` 动作的代码。之前在代码 9.2 中，我们是这样写的：

```
def edit
  @user = User.find(params[:id])
end
```

`update` 代码类似。既然 `correct_user` 事前过滤器中已经定义了 `@user`，这两个动作中就不再需要再定义 `@user` 变量了。

在继续阅读之前，你应该验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

9.2.3 更友好的转向

程序的权限限制基本完成了，但是还有一点小小的不足：不管用户尝试访问的是哪个受保护的页面，登录后都会转向资料页面。也就是说，如果未登录的用户访问了编辑资料页面，会要求先登录，登录转到的页面是 `/users/1`，而不是 `/users/1/edit`。如果登录后能转到用户之前想访问的页面就更好了。

针对这种更友好的转向，我们可以这样编写测试，先访问编辑用户资料页面，转向登录页面后，填写正确的登录信息，点击“Sign in”按钮，然后显示的应该是编辑用户资料页面，而不是用户资料页面。相应的测试如代码 9.17 所示。

代码 9.17 测试更友好的转向

```
spec/requests/authentication_pages_spec.rb
```

```

require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email", with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do
          it "should render the desired protected page" do
            page.should have_selector('title', text: 'Edit user')
          end
        end
      end
    end
  end
  .
  .
  .
end

```

下面我们来实现这个设想。⁴要转向用户真正想访问的页面，我们要在某个地方存储这个页面的地址，登录后再转向这个页面。我们要通过两个方法来实现这个过程，`store_location` 和 `redirect_back_or`，都在 Sessions 帮助方法模块中定义，如代码 9.18。

代码 9.18 实现更友好的转向所需的代码

`app/helpers/sessions_helper.rb`

```

module SessionsHelper

  .
  .
  .

  def redirect_back_or(default)
    redirect_to(session[:return_to] || default)
    session.delete(:return_to)
  end

  def store_location
    session[:return_to] = request.fullpath
  end
end

```

地址的存储使用了 Rails 提供的 `session`, `session` 可以理解成和 [8.2.1 节 \(chapter8.html#sec-8-2-1\)](#) 中介绍的 `cookies` 是类似的东西, 会在浏览器关闭后自动失效。 (在 [8.5 节 \(chapter8.html#sec-8-5\)](#) 中介绍过, 其实 `session` 的实现方法正是如此。) 我们还使用了 `request` 对象的 `fullpath` 方法获取了所请求页面的完整地址。在 `store_location` 方法中, 把完整的请求地址存储在 `session[:return_to]` 中。

要使用 `store_location`, 我们要把它加入 `signed_in_user` 事前过滤器中, 如代码 9.19 所示。

代码 9.19 把 `store_location` 加入 `signed_in_user` 事前过滤器
`app/controllers/users_controller.rb`

```

class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:edit, :update]
  before_filter :correct_user,   only: [:edit, :update]

  .
  .
  .

  def edit
  end

  .
  .
  .

  private

    def signed_in_user
      unless signed_in?
        store_location
        redirect_to signin_path, notice: "Please sign in."
      end
    end

    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_path) unless current_user?(@user)
    end
end

```

实现转向操作，要在 Sessions 控制器的 `create` 动作中加入 `redirect_back_or` 方法，用户登录后转到适当的页面，如代码 9.20 所示。如果存储了之前请求的地址，`redirect_back_or` 方法就会转向这个地址，否则会转向参数中指定的地址。

代码 9.20 加入友好转向后的 `create` 动作

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .

  def create
    user = User.find_by_email(params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      sign_in user
      redirect_back_or user
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end
end
.
.
.
```

`redirect_back_or` 方法在下面这行代码中使用了“或”操作符 `||`：

```
session[:return_to] || default
```

如果 `session[:return_to]` 的值不是 `nil`，上面这行代码就会返回 `session[:return_to]` 的值，否则会返回 `default`。注意，在代码 9.18 中，成功转向后就会删除存储在 `session` 中的转向地址。如果不删除的话，在关闭浏览器之前，每次登录后都会转到存储的地址上。（对这一过程的测试留作练习，参见 9.6 节。）

加入上述代码之后，代码 9.17 中对友好转向的集成测试应该可以通过了。至此，我们也就完成了基本的用户身份验证和页面保护机制。和之前一样，在继续阅读之前，最好确认一下所有的测试是否都可以通过：

```
$ bundle exec rspec spec/
```

9.3 列出所有用户

本节我们要添加计划中的倒数第二个用户动作，`index`。`index` 动作不会显示某一个用户，而是显示所有的用户。在这个过程中，我们要学习如何在数据库中生成示例用户数据，以及如何分页显示用户列表，显示任意数量的用户。用户列表、分页链接和“所有用户（Users）”导航链接的构思图如图 9.7 所示。⁵在 9.4 节中，我们还会在用户列表中添加删除链接，这样就可以删除有问题的用户了。



图 9.7：用户列表页面的构思图，包含了分页链接和“Users”导航链接

9.3.1 用户列表

单个用户的资料页面是对外开放的，不过用户列表页面只有注册用户才能访问。我们先来编写测试。在测试中我们要检测 `index` 动作是被保护的，如果访问 `users_path` 会转向登录页面。和其他的权限限制测试一样，我们也会把这个测试放在身份验证的集成测试中，如代码 9.21 所示。

代码 9.21 测试 `index` 动作是否是被保护的

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    describe "for non-signed-in users" do
      .
      .
      .
      describe "in the Users controller" do
        .
        .
        .
        describe "visiting the user index" do
          before { visit users_path }
          it { should have_selector('title', text: 'Sign in') }
        end
      end
      .
      .
      .
    end
  end
end
```

若要这个测试通过，我们要把 `index` 动作加入 `signed_in_user` 事前过滤器，如代码 9.22 所示。

代码 9.22 访问 `index` 动作必须先登录

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
  end
  .
  .
  .
end
```

接下来，我们要测试用户登录后，用户列表页面要有特定的标题和标头，还要列出网站中所有的用户。为此，我们要创建三个用户预构件，以第一个用户的身份登录，然后检测用户列表页面中是否有一个列表，各用户的名字都包含在一个单独的 li 标签中。注意，我们要为每个用户分配不同的名字，这样列表中的用户才是不一样的，如代码 9.23 所示。

代码 9.23 用户列表页面的测试

```
spec/requests/user_pages_spec.rb
```

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do
    before do
      sign_in FactoryGirl.create(:user)
      FactoryGirl.create(:user, name: "Bob", email: "bob@example.com")
      FactoryGirl.create(:user, name: "Ben", email: "ben@example.com")
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1', text: 'All users') }
    it "should list each user" do
      User.all.each do |user|
        page.should have_selector('li', text: user.name)
      end
    end
  end
  .
  .
  .
end
```

你可能还记得，我们在演示程序的相关代码中介绍过（参见代码 2.4），在程序中我们可以使用 `User.all` 从数据库中取回所有的用户，赋值给实例变量 `@users` 在视图中使用，如代码 9.24 所示。（你可能会觉得一次列出所有的用户不太好，你是对的，我们会在 [9.3.3 节](#) 中改进。）

代码 9.24 Users 控制器的 `index` 动作

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

要显示用户列表页面，我们要创建一个视图，遍历所有的用户，把单个用户包含在 `li` 标签中。我们要使用 `each` 方法遍历所有用户，显示用户的 Gravatar 头像和名字，然后把所有的用户包含在无序列表 `ul` 标签中，如代码 9.25 所示。在代码 9.25 中，我们用到了 [7.6 节 \(chapter7.html#sec-7-6\)](#) 练习中代码 7.29 的成果，允许向 Gravatar 帮助方法传入第二个参数，指定头像的大小。如果你之前没有做这个练习题，在继续阅读之前请参照代码 7.29 更新 Users 控制器的帮助方法文件。

代码 9.25 用户列表页面的视图

`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>
```

我们再添加一些 CSS（更确切的说是 SCSS）美化一下，如代码 9.26。

代码 9.26 用户列表页面的 CSS

`app/assets/stylesheets/custom.css.scss`

```
•  
•  
•  
/* users index */  
  
.users {  
  list-style: none;  
  margin: 0;  
  li {  
    overflow: auto;  
    padding: 10px 0;  
    border-top: 1px solid $grayLighter;  
    &:last-child {  
      border-bottom: 1px solid $grayLighter;  
    }  
  }  
}  
}
```

最后，我们还要在头部的导航中加入到用户列表页面的链接，链接的地址为 `users_path`，这是表格 7.1 ([chapter7.html#table-7-1](#)) 中还没介绍的最后一个具名路由了。相应的测试（代码 9.27）和程序所需的代码（代码 9.28）都很简单。

代码 9.27 检测“Users”链接的测试

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'  
  
describe "Authentication" do  
  .  
  .  
  .  
  describe "with valid information" do  
    let(:user) { FactoryGirl.create(:user) }  
    before { sign_in user }  
  
    it { should have_selector('title', text: user.name) }  
  
    it { should have_link('Users', href: users_path) }  
    it { should have_link('Profile', href: user_path(user)) }  
    it { should have_link('Settings', href: edit_user_path(user)) }  
    it { should have_link('Sign out', href: signout_path) }  
  
    it { should_not have_link('Sign in', href: signin_path) }  
    .  
    .  
    .  
  end  
end  
end
```

代码 9.28 添加“Users”链接

app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", users_path %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

至此，用户列表页面的功能就实现了，所有的测试也都可以通过了：

```
$ bundle exec rspec spec/
```

不过，如图 9.8 所示，页面中只显示了一个用户，有点孤单单。下面，让我们来改变一下这种悲惨状况。



图 9.8：用户列表页面，只有一个用户

9.3.2 示例用户

在本小节中，我们要为应用程序添加更多的用户。如果要让用户列表看上去像个列表，我们可以在浏览器中访问注册页面，然后一个一个地注册用户，不过还有更好的方法，让 Ruby 和 Rake 为我们创建用户。

首先，我们要在 `Gemfile` 中加入 `faker`（如代码 9.29 所示），使用这个 `gem`，我们可以使用半真实的名字和 Email 地址创建示例用户。

代码 9.29 把 `faker` 加入 `Gemfile`

```
source 'https://rubygems.org'

gem 'rails', '3.2.13'
gem 'bootstrap-sass', '2.0.0'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.0.1'

.
```

然后和之前一样，运行下面的命令安装：

```
$ bundle install
```

接下来我们要添加一个 Rake 任务创建示例用户。这个 Rake 任务保存在 `lib/tasks` 文件夹中，而且在 `:db` 命名空间中定义，如代码 9.30 所示。（代码中涉及到一些高级知识，现在不必深入了解。）

代码 9.30 在数据库中生成示例用户的 Rake 任务

```
lib/tasks/sample_data.rake
```

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    User.create!(name: "Example User",
                email: "example@railstutorial.org",
                password: "foobar",
                password_confirmation: "foobar")
    99.times do |n|
      name = Faker::Name.name
      email = "example-#{n+1}@railstutorial.org"
      password = "password"
      User.create!(name: name,
                  email: email,
                  password: password,
                  password_confirmation: password)
    end
  end
end
```

上述代码定义了一个名为 `db:populate` 的 Rake 任务，先创建一个用户替代之前存在的那个用户，然后还创建了 99 个用户。下面这行代码

```
task populate: :environment do
```

确保这个 Rake 任务可以获取 Rails 环境的信息，包括 User 模型，所以才能使用 `User.create!` 方法。`create!` 方法和 `create` 方法的作用一样，只不过如果提供的信息不合法不会返回 `false` 而是会抛出异常（参见 [6.1.4 节 \(chapter6.html#sec-6-1-4\)](#)），这样如果出错的话就很容易找到错误发生的地方。

这个任务是定义在 `:db` 命名空间中的，所以我们要按照如下的方式来执行：

```
$ bundle exec rake db:reset  
$ bundle exec rake db:populate  
$ bundle exec rake db:test:prepare
```

执行这三个任务之后，我们的应用程序就有 100 个用户了，如图 9.9 所示。（我牺牲了一点个人时间为前几个用户上传了头像，这样就不都会显示默认的 Gravatar 头像了。）

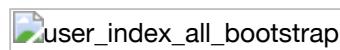


图 9.9：显示了 100 个用户的用户列表页面 ([/users \(http://localhost:3000/users\)](#))

9.3.3 分页

现在，当初的用户不再孤单单了，但是又出现了新的问题：用户太多，全在一个页面中显示。现在的用户数量是 100 个，算是少的了，在真实的网站中，这个数量可能是以千计的。为了避免在一页中显示过多的用户，我们可以使用分页功能，一页只显示 30 个用户。

在 Rails 中有很多实现分页的方法，我们要使用其中一个最简单也最完善的，叫做 `will paginate`。我们要使用 `'will_paginate'` 和 `'bootstrap-will_paginate'` 这两个 `gem`，`bootstrap-will_paginate` 的作用是设置 `will_paginate` 使用 `Bootstrap` 中的分页样式。修改后的 `Gemfile`` 如代码 9.31 所示。

代码 9.31 在 `Gemfile` 中加入 `will_paginate`

```
source 'https://rubygems.org'  
  
gem 'rails', '3.2.13'  
gem 'bootstrap-sass', '2.0.0'  
gem 'bcrypt-ruby', '3.0.1'  
gem 'faker', '1.0.1'  
gem 'will_paginate', '3.0.3'  
gem 'bootstrap-will_paginate', '0.0.6'  
. . .
```

然后执行下面的命令安装：

```
$ bundle install
```

安装后你还要重启 Web 服务器，确保成功加载这两个新 `gem`。

因为 will_paginate 这个 gem 使用的范围很广，所以我们不必做大量的测试，只需简单的测试一下就可以了。首先我们要检测页面中是否包含一个 CSS class 为 pagination 的 div 元素，这个元素就是由 will_paginate 生成的。然后，我们要检测分页的第一页中是否显示有正确的用户列表。在测试中我们要用到 paginate 方法，稍后会做介绍。

和之前一样，我们要使用 Factory Girl 生成用户，但是我们立马就会遇到一个问题：因为用户的 Email 地址必须是唯一的，那么我们就要手动生成 30 个用户，这可是一件很费事的活儿。而且，在测试用户列表时，用户的名字最好也不一样。幸好 Factory Girl 料事如神，提供了 sequence 方法来解决这种问题。在代码 7.8 中，我们是直接输入名字和 Email 地址来创建预构件的：

```
FactoryGirl.define do
  factory :user do
    name      "Michael Hartl"
    email     "michael@example.com"
    password  "foobar"
    password_confirmation "foobar"
  end
end
```

现在我们要使用 sequence 方法自动创建一系列的名字和 Email 地址：

```
factory :user do
  sequence(:name) { |n| "Person #{n}" }
  sequence(:email) { |n| "person_#{n}@example.com" }
  .
  .
  .
```

sequence 方法可以接受一个 Symbol 类型的参数，对应到属性上（例如 :name），其后还可以跟着块，有一个块参数，我们将其命名为 n。FactoryGirl.create(:user) 方法执行成功后，块参数会自动增加 1。因此，创建的第一个用户名为“Person 1”，Email 地址为“person1@example.com”；第二个用户名的名字为 Person 2，Email 地址为“person2@example.com”；依此类推。完整的代码如代码 9.32 所示。

代码 9.32 定义 Factory Girl 序列

spec/factories.rb

```
FactoryGirl.define do
  factory :user do
    sequence(:name) { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com" }
    password  "foobar"
    password_confirmation "foobar"
  end
end
```

创建了预构件序列后，在测试中就可以生成 30 个用户了，这 30 个用户就可以产生分页了：

```
before(:all) { 30.times { FactoryGirl.create(:user) } }
after(:all)  { User.delete_all }
```

注意，上述代码使用 `before(:all)` 确保在块中所有测试执行之前，一次性创建 30 个示例用户。这是对速度做的优化，因为在某些系统中每个测试都创建 30 个用户会很慢。对应的，我们调用 `after(:all)` 方法，在测试结束后一次性删除所有的用户。

代码 9.33 检测了页面中是否包含正确的 div 元素，以及是否显示了正确的用户。注意，我们把代码 9.23 中的 `User.all` 换成了 `User.paginate(page: 1)`，这样我们才能从数据库中取回第一页中要显示的用户。还要注意一下，代码 9.33 中使用的 `before(:each)` 方法是和 `before(:all)` 方法相反的操作。

代码 9.33 测试分页

`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do
    let(:user) { FactoryGirl.create(:user) }
    before(:each) do
      sign_in user
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1', text: 'All users') }

    describe "pagination" do
      before(:all) { 30.times { FactoryGirl.create(:user) } }
      after(:all) { User.delete_all }

      it { should have_selector('div.pagination') }

      it "should list each user" do
        User.paginate(page: 1).each do |user|
          page.should have_selector('li', text: user.name)
        end
      end
    end
  end
  .
  .
  .
end
```

要实现分页，我们要在用户列表页面的视图中加入一些代码，告诉 Rails 要分页显示用户，而且要把 `index` 动作中的 `User.all` 换成知道如何分页的方法。我们先在视图中加入特殊的 `will_paginate` 方法，如代码 9.34 所示。稍后我们会看到为什么要在用户列表的前后都加入分页代码。

代码 9.34 在用户列表视图中加入分页

app/views/users/index.html.erb

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>

<%= will_paginate %>
```

`will_paginate` 方法有点小神奇，在 `Users` 控制器的视图中，它会自动寻找名为 `@users` 的对象，然后显示一个分页导航链接。代码 9.34 所示的视图现在还不能正确显示分页，因为现在 `@users` 的值是通过 `User.all` 方法获取的，是个数组；而 `will_paginate` 方法需要的是 `ActiveRecord::Relation` 类对象。`will_paginate` 提供的 `paginate` 方法正好可以返回 `ActiveRecord::Relation` 类对象：

```
$ rails console
>> User.all.class
=> Array
>> User.paginate(page: 1).class
=> ActiveRecord::Relation
```

`paginate` 方法可以接受一个 Hash 类型的参数，键 `:page` 的值指定第几页。`User.paginate` 方法根据 `:page` 的值，一次取回一系列的用户（默认为 30 个）。所以，第一页显示的是第 1-30 个用户，第二页显示的是第 31-60 个，等。如果指定的页数不存在，`paginate` 会显示第一页。

我们可以把 `index` 动作中的 `all` 方法换成 `paginate`，这样页面中就可以显示分页导航了，如代码 9.35 所示。`paginate` 方法所需的 `:page` 参数值由 `params[:page]` 指定，这个 `params` 元素是由 `will_paginate` 自动生成的。

代码 9.35 在 `index` 动作中按分页取回用户

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .

  def index
    @users = User.paginate(page: params[:page])
  end
  .
  .
  .
end
```

现在，用户列表页面应该可以显示分页了，如图 9.10 所示。（在某些系统中，可能需要重启 Rails 服务器。）因为我们在用户列表前后都加入了 `will_paginate` 方法，所以这两个地方都会显示分页链接。



图 9.10：显示了分页链接的用户列表页面 (`/users` (<http://localhost:3000/users>))

如果点击链接“2”，或者“Next”，就会显示第二页，如图 9.11 所示。

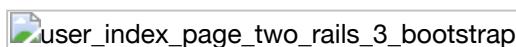


图 9.11：用户列表的第二页 (`/users?page=2` (<http://localhost:3000/users?page=2>))

你还应该验证一下测试是否可以通过：

```
$ bundle exec rspec spec/
```

9.3.4 视图重构

用户列表页面现在已经可以显示分页了，但是有个改进点我不得不介绍一下。Rails 提供了一些很巧妙的方法可以精简视图的结构，本小节我们就要利用这些方法重构一下用户列表页面。因为我们已经做了很好的测试，所以就可以放手去重构，不用担心会破坏网站的功能。

重构的第一步，要把代码 9.34 中的 `li` 换成对 `render` 方法的调用，如代码 9.36 所示。

代码 9.36 重构用户列表视图的第一步

`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <%= render user %>
  <% end %>
</ul>

<%= will_paginate %>
```

在上述代码中，`render` 的参数不再是指定局部视图的字符串，而是代表 `User` 类的 `user` 变量。⁶Rails 会自动寻找一个名为 `_user.html.erb` 的局部视图，我们要手动创建这个视图，然后写入代码 9.37 中的内容。

代码 9.37 显示单一用户的局部视图

`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
</li>
```

这个改进很不错，不过我们还可以做的更好。我们可以直接把 `@users` 变量传递给 `render` 方法，如代码 9.38 所示。

代码 9.38 完全重构后的用户列表视图

`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <%= render @users %>
</ul>

<%= will_paginate %>
```

Rails 会把 `@users` 当作一系列的 `User` 对象，遍历这些对象，然后使用 `_user.html.erb` 渲染每个对象。所以我们就得到了代码 9.38 这样简洁的代码。每次重构后，你都应该验证一下测试组件是否还是可以通过的：

```
$ bundle exec rspec spec/
```

9.4 删除用户

至此，用户索引也完成了。符合 REST 架构的Users 资源就只剩下最后一个 `destroy` 动作了。本节，我们先添加删除用户的链接（构思图如图 9.12 所示），然后再编写适当的 `destroy` 动作代码完成删除操作。不过，首先我们要先创建管理员级别的用户，并授权这些用户进行删除操作。



图 9.12：显示有删除链接的用户列表页面构思图

9.4.1 管理员

我们要通过 User 模型中一个名为 `admin` 的属性来判断用户是否具有管理员权限。`admin` 属性的类型为布尔值，Active Record 会自动生成一个 `admin?` 方法，返回布尔值，判断用户是否为管理员。针对 `admin` 属性的测试如代码 9.39 所示。

代码 9.39 测试 `admin` 属性

`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:admin) }
  it { should respond_to(:authenticate) }

  it { should be_valid }
  it { should_not be_admin }

  describe "with admin attribute set to 'true'" do
    before { @user.toggle!(:admin) }

    it { should be_admin }
  end
  .
  .
  .
end
```

在上述代码中我们使用 `toggle!` 方法把 `admin` 属性的值从 `false` 转变成 `true`。`it { should be_admin }` 这行代码说明用户对象应该可以响应 `admin?` 方法（这是 RSpec 对布尔值属性的一个约定）。

和之前一样，我们要使用迁移添加 `admin` 属性，在命令行中指定其类型为 `boolean`：

```
$ rails generate migration add_admin_to_users admin:boolean
```

这个命令生成的迁移文件（如代码 9.40 所示）会在 `users` 表中添加 `admin` 这一列，得到的数据模型如图 9.13 所示。



图 9.13：添加了 `admin` 属性后的 User 模型

代码 9.40 为 User 模型添加 admin 属性所用的迁移文件

db/migrate/[timestamp]_add_admin_to_users.rb

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

注意，在代码 9.40 中，我们为 `add_column` 方法指定了 `default: false` 参数，添加这个参数后用户默认情况下就不是管理员。（如果没有指定 `default: false`，`admin` 的默认值是 `nil`，也是“假值”，所以严格来说，这个参数不是必须的。不过，指定这个参数，可以更明确地向 Rails 以及代码的阅读者表明这段代码的意图。）

然后，我们要在“开发数据库”中执行迁移操作，还要准备好“测试数据库”：

```
$ bundle exec rake db:migrate
$ bundle exec rake db:test:prepare
```

和预想的一样，Rails 可以自动识别 `admin` 属性的类型为布尔值，而且自动生成了 `admin?` 方法：

```
$ rails console --sandbox
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

执行迁移操作后，针对 `admin` 属性的测试应该可以通过了：

```
$ bundle exec rspec spec/models/user_spec.rb
```

最后，我们要修改一下生成示例用户的代码，把第一个用户设为管理员，如代码 9.41 所示。

代码 9.41 生成示例用户的代码，把第一个用户设为管理员

lib/tasks/sample_data.rake

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    admin = User.create!(name: "Example User",
                         email: "example@railstutorial.org",
                         password: "foobar",
                         password_confirmation: "foobar")
    admin.toggle!(:admin)
    .
    .
    .
  end
end
```

之后还要还原数据库，并且重新生成示例用户：

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake db:test:prepare
```

attr_accessible 再探

你可能注意到了，在代码 9.41 中，我们使用 `toggle!(:admin)` 把用户设为管理员，为什么没有直接在 `User.create!` 的参数中指定 `admin: true` 呢？原因是，直接指定 `admin: true` 不起作用，Rails 就是这样设计的，只有通过 `attr_accessible` 指定的属性才能通过 mass assignment 赋值，而 `admin` 并不是可访问的。代码 9.42 显示的是当前可访问的属性列表，注意其中并没有 `:admin`。

代码 9.42 User 模型中通过 `attr_accessible` 指定的可访问的属性，其中没有 `:admin` 属性
app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  .
  .
  .
end
```

明确指定可访问的属性对网站的安全是很重要的，如果你没有指定，或者傻傻的把 `:admin` 也加进去了，那么心怀不轨的用户就可以发送下面这个 PUT 请求：

```
put /users/17?admin=1
```

这个请求会把 id 为 17 的用户设为管理员，这可是一个很严重的安全隐患。鉴于此，最佳的方法是在每个数据模型中都指定可访问的属性列表。其实，最好再测试一下各属性是否是可访问的，对 `:admin` 属性的可访问性测试留作练习，参见 9.6 节。

9.4.2 destroy 动作

编写完整的 Users 资源还要再添加删除链接和 `destroy` 动作。我们先在用户列表页面每个用户后面都加入一个删除链接，而且限制只有管理员才能看到这些链接。

编写针对删除功能的测试，最好能有个创建管理员的工厂方法，为此，我们可以在预构件中加入一个名为 `:admin` 的块，如代码 9.43 所示。

代码 9.43 添加一个创建管理员的工厂方法

`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    sequence(:name) { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com" }
    password "foobar"
    password_confirmation "foobar"

    factory :admin do
      admin true
    end
  end
end
```

添加了以上代码之后，我们就可以在测试中调用 `FactoryGirl.create(:admin)` 创建管理员用户了。

基于安全考虑，普通用户是看不到删除用户链接的，所以：

```
it { should_not have_link('delete') }
```

只有管理员才能看到删除用户链接，如果管理员点击了删除用户链接，该用户会被删除，用户的数量就会减少 1 个：

```
it { should have_link('delete', href: user_path(User.first)) }
it "should be able to delete another user" do
  expect { click_link('delete') }.to change(User, :count).by(-1)
end
it { should_not have_link('delete', href: user_path(admin)) }
```

注意，我们还添加了一个测试，确保管理员不会看到删除自己的链接。针对删除用户的完整测试如代码 9.44 所示。

代码 9.44 测试删除用户功能

`spec/requests/user_pages_spec.rb`

```

require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do

    let(:user) { FactoryGirl.create(:user) }

    before do
      sign_in user
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1',   text: 'All users') }

    describe "pagination" do
      .
      .
      .
    end

    describe "delete links" do

      it { should_not have_link('delete') }

      describe "as an admin user" do
        let(:admin) { FactoryGirl.create(:admin) }
        before do
          sign_in admin
          visit users_path
        end

        it { should have_link('delete', href: user_path(User.first)) }
        it "should be able to delete another user" do
          expect { click_link('delete') }.to change(User, :count).by(-1)
        end
        it { should_not have_link('delete', href: user_path(admin)) }
      end
    end
  end
end

```

然后在视图中加入代码 9.45。注意链接中的 `method: delete` 参数，它指明点击链接后发送的是 `DELETE` 请求。我们还把各链接放在了 `if` 语句中，这样就只有管理员才能看到删除用户链接。管理员看到的页面如图 9.14 所示。

代码 9.45 删除用户的链接（只有管理员才能看到）

app/views/users/_user.html.erb

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
  <% if current_user.admin? && !current_user?(user) %>
    | <%= link_to "delete", user, method: :delete, data: { confirm: "You sure?" } %>
  <% end %>
</li>
```



图 9.14：显示有删除用户链接的用户列表页面 ([/users \(http://localhost:3000/users\)](#))

浏览器不能发送 DELETE 请求，Rails 通过 JavaScript 进行模拟的。也就是说，如果用户禁用了 JavaScript，那么删除用户的链接就不可用了。如果必须要支持没有启用 JavaScript 的浏览器，你可以使用一个发送 POST 请求的表单来模拟 DELETE 请求，这样即使浏览器的 JavaScript 被禁用了，删除用户的链接还是可用的，更多细节请观看 RailsCasts 第 77 集

[《Destroy Without JavaScript \(http://railscasts.com/episodes/77-destroy-without-javascript\)》](#)。

若要删除用户的链接起作用，我们要定义 destroy 动作（参见表格 7.1 (chapter7.html#table-7-1)）。在 destroy 动作中，先找到要删除的用户，使用 Active Record 提供的 destroy 方法删除这个用户，然后再转向用户列表页面，如代码 9.46 所示。

代码 9.46 加入 destroy 动作

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_filter :correct_user,   only: [:edit, :update]

  def destroy
    User.find(params[:id]).destroy
    flash[:success] = "User destroyed."
    redirect_to users_path
  end
end
```

注意上述 destroy 动作中，把 find 方法和 destroy 方法链在一起使用了：

```
User.find(params[:id]).destroy
```

理论上，只有管理员才能看到删除用户的链接，所以只有管理员才能删除用户。但实际上，还是存在一个严重的安全隐患：只要攻击者有足够的经验，就可以在命令行中发送 `DELETE` 请求，删除网站中的用户。为了保证网站的安全，我们还要限制对 `destroy` 动作的访问，因此我们在测试中不仅要确保只有管理员才能删除用户，还要保证其他用户不能执行删除操作，如代码 9.47 所示。注意，和代码 9.11 中的 `put` 方法类似，在这段代码中我们使用 `delete` 方法向指定的地址 (`user_path`，参见表格 7.1 (`chapter7.html#table-7-1`)) 发送了一个 `DELETE` 请求。

代码 9.47 测试访问受限的 `destroy` 动作

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as non-admin user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:non_admin) { FactoryGirl.create(:user) }

      before { sign_in non_admin }

      describe "submitting a DELETE request to the Users#destroy action" do
        before { delete user_path(user) }
        specify { response.should redirect_to(root_path) }
      end
    end
  end
end
```

理论上来说，网站中还是有一个安全漏洞，管理员可以发送 `DELETE` 请求删除自己。有些人可能会想，这样的管理员是自作自受。不过作为开发人员，我们最好还是要避免这种情况的发生，具体的实现留作练习，参见 9.6 节。

你可能已经知道了，我们要使用一个事前过滤器限制对 `destroy` 动作的访问，如代码 9.48 所示。

代码 9.48 限制只有管理员才能访问 `destroy` 动作的事前过滤器

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_filter :correct_user,   only: [:edit, :update]
  before_filter :admin_user,     only: :destroy
  .
  .
  .
  private
  .
  .
  .
  def admin_user
    redirect_to(root_path) unless current_user.admin?
  end
end
```

至此，所有的测试应该都可以通过了，而且 Users 相关的资源，包括控制器、模型和视图，都已经实现了。

```
$ bundle exec rspec spec/
```

9.5 小结

我们用了好几章来介绍如何实现 Users 资源，在[5.4 节 \(chapter5.html#sec-5-4\)](#)用户还不能注册，而现在不仅可以注册，还可以登录、退出、查看个人资料、修改设置，还能浏览网站中所有的用户列表，某些用户甚至可以删除其他的用户。

本书剩下的内容会以这个 Users 资源为基础（以及相关的权限授权系统），在[第十章 \(chapter10.html\)](#)中为示例程序加入类似 Twitter 的微博功能，在[第十一章 \(chapter11.html\)](#)中实现关注用户的状态列表。最后这两章会介绍几个 Rails 中最为强大的功能，其中包括通过 has_many 和 has_many through 实现的数据模型关联。

在继续阅读之前，先把本章所做的改动合并到主分支：

```
$ git add .
$ git commit -m "Finish user edit, update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
```

你还可以将程序部署到“生产环境”，再生成示例用户（在此之前要使用 pg:reset 命令还原“生产数据库”）：

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:populate
```

（如果你忘了 Heroku 程序的名字，可以直接运行 heroku pg:reset DATABASE，Heroku 会告诉你程序的名字。）

还有一点需要注意，本章我们加入了程序所需的最后一个 gem，最终的 Gemfile 如代码 9.49 所示。

代码 9.49 示例程序所需 Gemfile 的最终版本

```

source 'https://rubygems.org'

gem 'rails', '3.2.13'
gem 'bootstrap-sass', '2.0.0'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.0.1'
gem 'will_paginate', '3.0.3'
gem 'bootstrap-will_paginate', '0.0.6'

group :development do
  gem 'sqlite3', '1.3.5'
  gem 'annotate', '~> 2.4.1.beta'
end

# Gems used only for assets and not required
# in production environments by default.

group :assets do
  gem 'sass-rails', '3.2.4'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.0'

group :test, :development do
  gem 'rspec-rails', '2.10.0'
  gem 'guard-rspec', '0.5.5'
  gem 'guard-spork', '0.3.2'
  gem 'spork', '0.9.0'
end

group :test do
  gem 'capybara', '1.1.2'
  gem 'factory_girl_rails', '1.4.0'
  gem 'cucumber-rails', '1.2.1', require: false
  gem 'database_cleaner', '0.7.0'
end

group :production do
  gem 'pg', '0.12.2'
end

```

9.6 练习

1. 参照代码 10.8，编写一个，测试确保 User 模型的 admin 属性是不可访问的。确保测试先是红色的，然后才会变绿。（提示：先要把 admin 加入可访问属性列表中。）
2. 把代码 9.3 中修改 Gravatar 头像的链接（“change”），使链接在新窗口（或新标签）中打开。提示：请搜索，你会发现一个很常用的方法，涉及到 `_blank` 的用法。

- 现在针对身份验证系统的测试会确保用户登录后能看到“Profile”和“Settings”等导航链接。增加一个测试，确保用户未登录时看不到这些导航链接。
- 在测试中尽量多的使用代码 9.6 中的 `sign_in` 帮助方法。
- 使用代码 9.50 中的代码重构 `new.html.erb` 和 `edit.html.erb` 中的表单。注意，你要明确的传入 `f` 这个表单变量，如代码 9.51 所示。你还要修改相应的测试，因为表单已经不完全一样了。仔细的查找修改前后表单的差异，据此修改测试。
- 已经登录的用户就没必要再访问 `Users` 控制器的 `new` 和 `create` 动作了，修改程序，如果登录后的用户访问这些地址时，转向到网站首页。
- 在网站的布局中插入一些 [Rails API \(http://api.rubyonrails.org/v3.2.0/classes/ActionDispatch/Request.html\)⁸](http://api.rubyonrails.org/v3.2.0/classes/ActionDispatch/Request.html) 中介绍的方法，了解一下 `request` 对象。（如果遇到了困难，可以参考代码 7.1。）
- 编写一个测试，确保友好转向只在第一次转向指定的地址，其后再登录的话就转向默认设定的地址（如资料页面）。代码 9.52 是个提示，其实也就是所需的代码。
- 修改 `destroy` 动作，避免管理员删除自己。（先编写测试。）

代码 9.50 注册和编辑表单字段的局部视图

`app/views/users/_fields.html.erb`

```
<%= render 'shared/error_messages' %>

<%= f.label :name %>
<%= f.text_field :name %>

<%= f.label :email %>
<%= f.text_field :email %>

<%= f.label :password %>
<%= f.password_field :password %>

<%= f.label :password_confirmation, "Confirm Password" %>
<%= f.password_field :password_confirmation %>
```

代码 9.51 使用局部视图后的注册页面视图

`app/views/users/new.html.erb`

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'fields', f: f %>
      <%= f.submit "Create my account", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

代码 9.52 测试友好的转向后，只能转向到默认的页面

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    describe "for non-signed-in users" do
      .
      .
      .
      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email", with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do
          it "should render the desired protected page" do
            page.should have_selector('title', text: 'Edit user')
          end

          describe "when signing in again" do
            before do
              visit signin_path
              fill_in "Email", with: user.email
              fill_in "Password", with: user.password
              click_button "Sign in"
            end

            it "should render the default (profile) page" do
              page.should have_selector('title', text: user.name)
            end
          end
        end
      end
    end
  end
  .
  .
  .
end
end
```

1. 图片来自 <http://www.flickr.com/photos/sashawolff/4598355045/>
(<http://www.flickr.com/photos/sashawolff/4598355045/>) ↵
2. Gravatar 会把这个地址转向 <http://en.gravatar.com/emails>, 我去掉了前面的 en, 这样选择其他语言的用户就会自动转向相应的页面了。 ↵
3. 不要担心实现的细节。具体的实现方式是 Rails 框架的开发者需要关注的, 作为 Rails 程序开发者则无需关心。 ↵
4. 实现的代码来自 thoughtbot 的 Clearance gem。 ↵
5. 婴儿的图片来自 ↵
6. 我们并不是一定要使用 user, 遍历时如果用的是 @users.each do |foobar|, 那么就要用 render foobar。这里的关键是要知道对象的类, 也就是 User。 ↵
7. 类似 curl 的命令行工具可以发送这种 PUT 请求。 ↵
8. <http://api.rubyonrails.org/v3.2.0/classes/ActionDispatch/Request.html>
(<http://api.rubyonrails.org/v3.2.0/classes/ActionDispatch/Request.html>) ↵

00

©2014 Yong Yuan (<http://yuanyong.org>) 保留部分权力。在线阅读版本基于“CC 3.0 BY-SA 协议” (<http://creativecommons.org/licenses/by-sa/3.0/>) 发布

第十章 OpenCV

[10.1 OpenCV Python接口](#)

[10.2 OpenCV基础](#)

[10.2.1 读取、写入图像](#)

[10.2.2 颜色空间](#)

[10.2.3 显示图像和结果](#)

[10.3 视频处理](#)

[10.3.1 视频输入](#)

[10.3.2 读取视频到NumPy数组](#)

[10.4 跟踪](#)

[10.4.1 光流法](#)

[10.4.2 Lucas-Kanade算法](#)

这一章主要讲述通过Python接口使用目前流行的计算机视觉编程库OpenCV。OpenCV是一个C++库，用于实时处理计算机视觉方面的问题。

10.1 OpenCV Python接口

OpenCV是一个C++库，它涵盖了很多计算机视觉领域的模块。可以通过访问[\[http://opencv.willowgarage.com/documentation/python/index.html\]](http://opencv.willowgarage.com/documentation/python/index.html) (<http://opencv.willowgarage.com/documentation/python/index.html>)。

OpenCV目前最新的版本是2.4.8。实际上，OpenCV有两个Python接口，老版本的cv模块使用OpenCV内置的数据类型，新版本的cv2模块使用**NumPy**数组。对于新版本的模块，可以通过下面方式导入：

```
import cv2
```

而老版本的模块则通过下面方式导入：

```
import cv2.cv
```

在本章中，我们使用cv2模块，译者使用的OpenCV版本是2.4.6。

10.2 OpenCV基础

OpenCV提供了读取图像和写入图像，矩阵操作以及数学库函数。我们先来看看这些基本组件并学习怎样使用它们。

10.2.1 读取、写入图像

下面是一个简短的载入图像、打印尺寸、转换格式及保存图像为.png格斯的例子：

```
# -*- coding: utf-8 -*-
import cv2

# 读入图像
im = cv2.imread('../data/empire.jpg')

# 打印图像尺寸
h, w = im.shape[:2]
print h, w

# 保存原jpg格式的图像为png格式图像
cv2.imwrite('../images/ch10/ch10_P210_Reading-and-Writing-Images.png',im)
```

运行上面代码后，在ch10文件下保存有empire.jpg转换成.png格式的图片，即ch10P210Reading-and-Writing-Images.png，下面是转换格式后保存的.png的图像：



函数imread()将图像返回为一个标准的**NumPy**数组，如果你喜欢的话，你可以将该函数用于PIL图像读取的备选函数。函数imwrite()能够根据文件后缀自动的进行格式转换。

10.2.2 颜色空间

在OpenCV中，图像不是用常规的RGB颜色通道来存储的，它们用的是BGR顺序。当读取一幅图像后，默认的是BGR，不过有很多转换方式是可以利用的。颜色空间转换可以用函数cvtColor()函数。比如，下面是一个转换为灰度图像的例子：

```
import cv2

im = cv2.imread('../data/empire.jpg')
# create a grayscale version
gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
```

10.2.3 显示图像和结果

下面我们看一些用OpenCV进行图像处理并用OpenCV绘图及窗口管理功能显示图像后的结果的示例。

第一个例子是从文件中读取一幅图像，并创建积分图像表示：

```
# -*- coding: utf-8 -*-
import cv2
from pylab import *

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttf", size=14)

# 读入图像
im = cv2.imread('../data/fisherman.jpg')
# 转换颜色空间
gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

# 显示积分图像
fig = plt.figure()
subplot(121)
plt.gray()
imshow(gray)
title(u'灰度图', fontproperties=font)
axis('off')

# 计算积分图像
intim = cv2.integral(gray)
# 归一化
intim = (255.0*intim) / intim.max()

# 显示积分图像
subplot(122)
plt.gray()
imshow(intim)
title(u'积分图', fontproperties=font)
axis('off')
show()

# 用OpenCV显示图像
#cv2.imshow("Image", intim)
#cv2.waitKey()

# 用OpenCV保存积分图像
#cv2.imwrite('../images/ch10/ch10_P211_Displaying-Images-and-Results-cv2.jpg', intim)

# 保存figure中的灰度图像和积分图像
fig.savefig("../images/ch10/ch10_P211_Displaying-Images-and-Results.png")
```

运行上面代码，显示如下结果，并在/images/ch10/目录下生成一幅保存有灰度图像和积分图像的图片：

灰度图



积分图



第二个例子从种子像素开始应用泛洪(漫水)填充：

```

# -*- coding: utf-8 -*-
import cv2
import numpy
from pylab import *

# 添加中文字体支持
from matplotlib.font_manager import FontProperties
font = FontProperties(fname=r"c:\windows\fonts\SimSun.ttc", size=14)

# 读入图像
filename = '../data/fisherman.jpg'
im = cv2.imread(filename)
# 转换颜色空间
rgbIm = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

# 显示原图
fig = plt.figure()
subplot(121)
plt.gray()
imshow(rgbIm)
title(u'原图', fontproperties=font)
axis('off')

# 获取图像尺寸
h, w = im.shape[:2]
# 泛洪填充
diff = (6, 6, 6)
mask = zeros((h+2, w+2), numpy.uint8)
cv2.floodFill(im, mask, (10, 10), (255, 255, 0), diff, diff)

# 显示泛洪填充后的结果
subplot(122)
imshow(im)
title(u'泛洪填充', fontproperties=font)
axis('off')

show()
#fig.savefig("../images/ch10/floodFill.png")

# 在OpenCV窗口中显示泛洪填充后的结果
# cv2.imshow('flood fill', im)
# cv2.waitKey()
# 保存结果
# cv2.imwrite('../images/ch10/floodFill.jpg', im)

```

译者使用的是matplotlib显示泛洪填充后的结果，上面代码底下的注释部分是用OpenCV显示泛洪填充的结果。如果你用OpenCV窗口显示上面运行的结果，可以反注释。下面是上面泛洪填充后的结果：

原图



泛洪填充



作为最后一个例子，我们看一下提取图像的SURF(加速稳健特征)

(<http://zh.wikipedia.org/wiki/%E5%8A%A0%E9%80%9F%E7%A8%B3%E5%81%A5%E7%89%B9%E5%BE%81>)特征。SURF是SIFT特征的一个快速版本。这里我们也会展示一些OpenCV绘制命令。

```
# -*- coding: utf-8 -*-
import cv2
import numpy
from pylab import *

# 读入图像
im = cv2.imread('../data/empire.jpg')
# 下采样
im_lowres = cv2.pyrDown(im)
# 转化为灰度图像
gray = cv2.cvtColor(im_lowres, cv2.COLOR_RGB2GRAY)
# 检测特征点
s = cv2.SURF()
mask = numpy.uint8(ones(gray.shape))
keypoints = s.detect(gray, mask)
# 显示图像及特征点
vis = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
for k in keypoints[::10]:
    cv2.circle(vis, (int(k.pt[0]), int(k.pt[1])), 2, (0, 255, 0), -1)
    cv2.circle(vis, (int(k.pt[0]), int(k.pt[1])), int(k.size), (0, 255, 0), 2)
cv2.imshow('local descriptors', vis)
cv2.waitKey()

cv2.imwrite('../images/ch10/ch10_P261_Fig10-3.jpg', vis)
```

上面代码先读入一幅图像，用pyrDown下采样，得到的一幅尺寸是原图像尺寸一半的降采样图像，即im_lowres，然后将图像转换为灰度图像，并将它传递给SURF关键点检测对象。运行上面代码，可得下面SURF特征点检测结果：



10.3 视频处理

单纯利用Python处理视频是比较困难的，因为要考虑到速度、编解码器、摄像机、操作系统以及文件格式等问题。目前Python还没有视频处理库。Python处理视频的接口仅有的较好的就是OpenCV。在这一节，我们会展示一些对视频进行处理的基本例子。

10.3.1 视频输入

OpenCV能够很好地支持视频的读入。下面的例子展示了捕获视频帧，并在OpenCV窗口中显示它们：

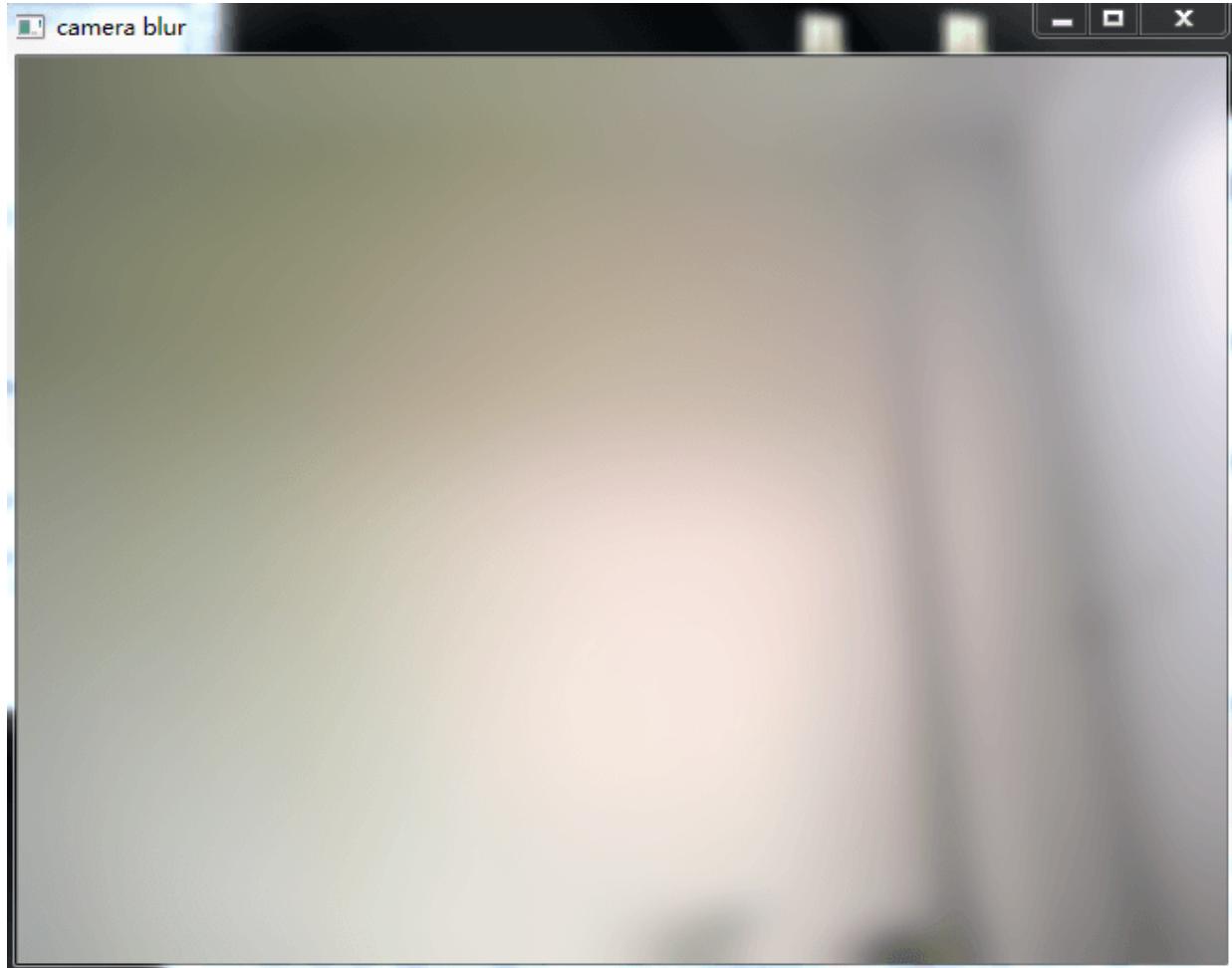
```
import cv2
# setup video capture
cap = cv2.VideoCapture(0)
while True:
    ret,im = cap.read()
    cv2.imshow('video test',im)
    key = cv2.waitKey(10)
    if key == 27:
        break
    if key == ord(' '):
        cv2.imwrite('vid_result.jpg',im)
```

上面VideoCapture从摄像头或文件中捕获视频。这里我们给它传递了一个整数作为初始化参数，它实际是视频设备的ID号，对于单个连着的摄像头，其ID号为0。read()方法解码并返回下一视频帧。waitKey()函数等待用户按下“Esc”(对应的ASCII码为27)终止应用，或按“space”将当前帧保存起来。

我们将上面例子进行拓展，使其能够在OpenCV窗口中对输出的视频进行模糊。对上面的例子进行稍微的修改便可实现该功能：

```
import cv2
# setup video capture
cap = cv2.VideoCapture(0)
# get frame, apply Gaussian smoothing, show result
while True:
    ret,im = cap.read()
    blur = cv2.GaussianBlur(im,(0,0),10)
    cv2.imshow('camera blur',blur)
    if cv2.waitKey(10) == 27:
        break
```

上面对每一帧图像，将其传给GaussianBlur()函数，该函数实现对图像进行高斯滤波。在这个实例中，我们传递的是彩色图像，每一个颜色通道可以分别对其进行模糊。该函数以一个滤波器大小元组及高斯函数的标准差作为输入，如果滤波器大小设置为0，则它自动赋为标准差。运行上面的结果如下：



10.3.2 读取视频到NumPy数组

OpenCV可以从一个文件中读取视频帧序列，并将它们转换成NumPy数组。下面给出了一个从摄像头捕获视频，并将它们存储在NumPy数组中的例子。

```
import cv2
from pylab import *
# setup video capture
cap = cv2.VideoCapture(0)
frames = []
# get frame, store in array
while True:
    ret,im = cap.read()
    cv2.imshow('video',im)
    frames.append(im)
    if cv2.waitKey(10) == 27:
        break
frames = array(frames)
# check the sizes
print im.shape
print frames.shape
```

上面每一帧数组会被添加到列表的末尾直到捕获终止。打印出的数组大小表示的帧数、高度、宽度、3。运行上面代码打印出的结果为：

```
(480, 640, 3)
(40, 480, 640, 3)
```

这里，记录了40帧。类似如这样的视频数据数组非常适合视频处理，比兔计算视频帧差异以及跟踪。

10.4 跟踪

跟踪是对视频帧序列中的物体进行跟踪。

10.4.1 光流法

10.4.2 Lucas-Kanade算法

Lucas-Kanade算法原理这里略，具体可以参阅中译本。

```
import lktrack

imnames = ['../data/bt/bt.003.pgm', '../data/bt/bt.002.pgm', '../data/bt/bt.001.pgm',
'../data/bt/bt.000.pgm']
# create tracker object
lkt = lktrack.LKTracker(imnames)
# detect in first frame, track in the remaining
lkt.detect_points()
lkt.draw()
for i in range(len(imnames)-1):
    lkt.track_points()
    lkt.draw()
```