第一章 Django介绍

本书所讲的是Django：一款能够节约你的时间并且让你的开发乐趣横生的web开发框架。使用Django，花极少时间即可构建和维护质量上乘的Web应用。

从好的方面来看，Web 开发激动人心且富于创造性；从另一面来看，它却是份繁琐而令人生厌的工作。通过减少重复的代码，Django 使你能够专注于 web 应用上有 趣的关键性的东西。为了达到这个目标，Django 提供了通用Web开发模式的高度抽象，提供了频繁进行的编程作业的快速解决方法，以及为“如何解决问题”提供了清晰明了的约定。同时，Django 尝试留下一些方法，来让你根据需要在framework之外来开发。

本书的目的是将你培养成Django专家。主要侧重于两方面：第一，我们深度解释 Django 到底做了哪些工作以及如何用她构建Web应用；第二，我们将会在适当的地方讨论更高级的概念，并解释如何在自己的项目中高效的使用这些工具。通过阅读此书，你将学会快速开发功能强大网站的技巧，并且你的代码将会十分 清晰，易于维护。

在这一章中，我们将一览 Django 的全貌。

什么是Web框架？

Django是一个卓越的新一代Web框架，而Django这个词语的意义是什么呢？

要回答这个问题，让我们来看看通过编写标准的CGI程序来开发Web应用，这在大约1998年的时候非常流行。编写CGI Web应用时，你需要自己处理所有的操作，就像你想烤面包，但是都需要自己生火一样。下面是实例，一个简单的CGI脚本，用Python写的，读取数据库并显示最新发布的十本书：

```
#!/usr/bin/python

import MySQLdb

print "Content-Type: text/html"
print
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"

connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"

connection.close()
```

代码十分简单。首先，根据CGI的要求输出一行Content-Type，接下来是一个空行。再接下来是一些HTML的起始标签，然后连接数据库并执行一些查询操作，获取最新的十本书。遍历这些书，同时生成一个HTML的无序序列。最后，输出HTML的结束标签并且关闭数据库连接。

类似这样的一次性动态页面，从头写起并不是最好的办法。其中一点，这些代码简单易懂，就算是一个开发初学者都能理解这16行代码从开始到结束做了些什么，不需要阅读额外的代码。同样这16行代码也很容易部署：只需要将它保存在名为“latestbooks.cgi”的文件里，上传至网络服务器，通过浏览器访问即可。

但是Web应用远远要复杂很多，这种方法就不再适用，而且你将会要面对很多问题：

- 当多个动态页面需要同时连接数据库时，将会发生什么？当然，连接数据库的代码不 应该同时存在于各个独立的CGI脚本中，所以最踏实的做法是把这些代码重新组织到一个公共函数里面。
- 一个开发人员真的需要去关注如何输出Content-Type以及完成所有操作后去关闭数据库么？此类问题只会降低开

发人员的工作效率，增加犯错误的几率。那些初始化和释放相关的工作应该交给一些通用的框架来完成。

- 如果这样的代码被重用到一个复合的环境中会发生什么？每个页面都分别对应独立的数据库和密码吗？从这点看来，就需要一些环境相关的配置文件。
- 如果一个Web设计师，完全没有Python开发经验，但是又需要重新设计页面的话，又将 发生什么呢？理想的情况是，页面显示的逻辑与从数据库中读取书本记录分隔开，这样 Web设计师的重新设计不会影响到之前的业务逻辑。

以上正是Web框架致力于解决的问题。Web框架为应用程序提供了一套程序框架， 这样你可以专注于编写清晰、易维护的代码，而无需从头做起。简单来说，这就是Django所能做的。

MVC 设计模式

让我们来研究一个简单的例子，通过该实例，你可以分辨出，通过Web框架来实现的功能与之前的方式有何不同。下面就是通过使用Django来完成以上功能的例子：

```
# models.py (the database tables)

from django.db import models

class Book(models.Model):
    name = models.CharField(maxlength=50)
    pub_date = models.DateField()


# views.py (the business logic)

from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})


# urls.py (the URL configuration)

from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'latest/$', views.latest_books),
)


# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

先不要担心这个东西是 *如何* 工作的，我们主要是先想让你知道总体的设计，这里关键要注意的是 *分离问题*

- `models.py` 文件主要用一个 Python 类来描述数据表。称为 *模型(model)* 。 运用这个类，你可以通过简单的 Python 的代码来创建、检索、更新、删除 数据库中的记录而无需写一条又一条的SQL语句。
- `view.py` 文件的 latest_books() 函数中包含了该页的业务层逻辑。这个函数叫做 *视图(view)* 。
- `urls.py` 指出了什么样的 URL 调用什么的视图，在这个例子中 `/latest/` URL 将会调用 `latest_books()` 这个函

数

- `latest_books.html` 是 html 模板，它描述了这个页面的设计是如何的。

这些部分松散的组合在一起就是模型－视图－控制器（MVC）的设计模式。简单的说， MVC 是一种软件开发的方法，它把代码的定义和数据访问的方法（模型）与请求逻辑 （控制器）还有用户接口（视图）分开来。

这种设计模式关键的优势在于各种组件都是 *松散结合* 的。这样，每个由 Django驱动 的Web应用都有着明确的目的，并且可独立更改而不影响到其它的部分。比如，开发者 更改一个应用程序中的 URL 而不用影响到这个程序底层的实现。设计师可以改变 HTML 页面 的样式而不用接触 Python 代码。数据库管理员可以重新命名数据表并且只需更改一个地方，无需从一大堆文件中进行查找和替换。

本书中，每个组件都有它自己的一个章节。比如，第三章涵盖了视图，第四章是模板， 而第五章是模型。同时第五章也深入讨论了 Django 的 MVC 思想。

django 历史

在我们讨论代码之前我们需要先了解一下 Django 的历史。知道了一些历史知识有助于理解为什么 Django 要建立这个框架，因为这些历史有助于理解Django为何会这样运作。

如果你曾编写过网络应用程序。那么你很有可能熟悉之前我们的 CGI 例子。传统的 网络开发人员的开发流程是这样的:

1. 从头开始编写网络应用程序。
2. 从头编写另一个网络应用程序。
3. 从第一步中总结（找出其中通用的代码），并运用在第二步中。
4. 重构代码使得能在第 2 个程序中使用第 1 个程序中的通用代码。
5. 重复 2-4 步骤若干次。
6. 意识到你发明了一个框架。

这就是为什么 Django 创建的原因！

Django 是从真实世界的应用中成长起来的，它是由 堪萨斯（Kansas）州 Lawrence 城中的一个 网络开发小组编写的。它诞生于 2003 年秋天，那时 *Lawrence Journal-World* 报纸的 程序员 Adrian Holovaty 和 Simon Willison 开始用 Python 来编写程序。 当时他们的 World Online 小组制作并维护当地的几个新闻站点, 并在以新闻界特有的快节奏开发环境中逐渐发展. 这些站点包括有 LJWorld.com、Lawrence.com 和 KUsports.com， 记者（或管理层） 要求增加的特征或整个程序都能在计划时间内快速的被建立，这些时间通常只有几天 或几个小时。因此为了需要，Adrian 和 Simon 开发了一种节省时间的网络程序开发框架， 这是在截止时间前能完成程序的唯一途径。

2005 年的夏天，当这个框架开发完成时，它已经用来制作了很多个 World Online 的站点。 当时 World Online 小组中的 Jacob Kaplan-Moss 决定把这个框架发布为一个开源软件。 他们在 2005 年的 7 月发布并取名为 Django，来源于一个著名的爵士乐吉他演奏家 Django Reinhardt。

虽然现在 Django 是一个全世界开发者参与的开源项目，但原始的 World Online 开发者们 仍然提供主要的指导来促进这个框架的成长。 World Online 还有其它方面的重要贡献，比如雇员时间、 市场材料以及框架的 Web 网站的主机和带宽 (http://www.djangoproject.com/)。

这些历史都是相关联的，因为她们帮助解释了很重要的两点。第一，Django最可爱的地方， 因为Django诞生于一个新闻环境，她提供了很多的功能（特别是她的管理接口，见第6章）， 特别适合提供内容的网站，例如eBay, craigslist.org 和washingtonpost.com，提供一种 基于数据库的动态网站。（不要看到这就感到沮丧，尽管Django擅长于动态内容管理系统， 但并不表示Django主要的目的就是用来创建动态内容的网站。 某些方面 *特别高效* 与 其他方面 *不高效* 是有区别的）

第二，Django的起源造就她的开源社区，因为Django来自于真实世界中的代码，而不是 来自于一个科研项目或者商业产品，她主要集中力量来解决Web开发中遇到的问题，同样 也是Django的开发者经常遇到的问题。这样，Django每天

在现有的基础上进步。框架的 开发者对于为开发人员节省开发时间具有极大的兴趣，编写更加容易维护的程序，同时保证程序运行的效率。开发人员自我激励，尽量的节省时间和享受他们的工作(To put it bluntly, they eat their own dog food.)

如何阅读本书

在编写本书时，我们努力尝试在可读性和参考性间做一个平衡，当然本书会偏向于可 读性。本书的目标，之前也提过，是要将你培养成一名Django专家，我们相信，最好 的方式就是文章和充足的实例，而不是提供一堆详尽却乏味的关于Django特色的手册。 （曾经有人说过，如果仅仅教字母表是无法教会别人说话的。）

按照这种思路，我们推荐按顺序阅读第 1-7 章。这些章节构成了如何使用 Django 的基础；读过之后，你就可以搭建由 Django 支撑的网站了。剩下的章节重点讲述 Django 的其它一些特性，可以按照任何顺序阅读。

附录部分用作参考资料。要回忆语法或查阅 Django 某部分的功能概要时，你偶尔可能会回来翻翻这些资料以及 http://www.djangoproject.com/ 上的免费文档。

所需编程知识

本书读者需要理解基本的面向过程和面向对象编程：流程控制（ `if` ， `while` 和 `for` ），数据结构（列表，哈希表/字典），变量，类和对象。

Web开发经验，正如你所想的，也是非常有帮助的，但是对于阅读本书，并不是必须的。 通过本书，我们尽量给缺乏经验的开发人员提供在Web开发中最好的实践。

所需python知识

本质上来说， Django 只不过是用 Python 编写的一组类库。用 Django 开发站点就是使用这些类库编写 Python 代码。因此，学习 Django 的关键就是学习如何进行 Python 编程并理解 Django 类库的运作方式。

如果你有Python开发经验，在学习过程中应该不会有任何问题，基本上,Django的代码并 没有使用一些黑色魔法（例如代码中的欺骗行为，某个实现解释或者理解起来十分困难）。 对你来说，学习Django就是学习她的命名规则和API。

如果你没有使用 Python 编程的经验，你一定会学到很多东西。它是非常易学易用的。 虽然这本书没有包括一个完整的 Python 教程， 但也算是一个恰当的介绍了 Python特征和 功能的集锦。当然，我们推荐你读一下官方的 Python 教程，它可 以从 http://docs.python.org/tut/ 在线获得。另外我们也推荐 Mark Pilgrims的 书 *Dive Into Python* （ http://www.diveintopython.org/ ）

Django 之新特性

正如我们之前提到的，Django 改进频繁，到本书出版时，可能有一些甚至是 *非常基本* 的新功能将被推出。因此，作为作者，我们要通过此书达到两个方面的目标：

- 保证本书尽可能的面向未来，因此，不管你在本书中读到什么内容，在未来新 的Django版本中都将会可用的。
- 及时的更新本书的在线版， http://www.djangobook.com/ ，这样在我们完成 新的章节后，你可以获得最新和最好的版本。

如果你想用django来实现某些书中没有提到的功能，请到前面提到的网站上检查这本书 的地最新版本，并且同样记得要去检查官方的django文档。

获取帮助

django的最大的益处是,有一群乐于助人的人在django社区上.你可以毫无约束的提各种 问题在上面如:从django的安装,app设计,db 设计,发布。

- django邮件列表是django用户提出问题，回答问题的地方，可以通过

  http://www.djangoproject.com/r/django-users . 来免费注册。

- django irc channel如果django用户遇到什么棘手的问题希望的及时地回复是可以使用它。 在freenode IRC network加入#django

下一章

下一章，主要讲述如何开始安装和初始化django。

第二章 入门

良好的开端胜过一切。后续章节将充斥着 Django 框架的细节和拓展，不过现在呢，请相信我们，这一章还是蛮有意思的。

Django 安装很简单。因为所有 Python 可运行的地方 Django 都可以运行，所以可以通过多种方式配置 Django 。这一章中，我们将介绍一些常见的 Django 安装方案。第20章中将介绍如何将 Django 部署为产品。

Python 安装

Django 由百分百的纯 Python 代码编写而成，因此必须在系统中安装 Python 。Django 需要 2.3 或更高版本的 Python。

如果使用的是 Linux 或 Mac OS X，系统可能已经预装了 Python 。在命令提示符下 (或 OS X 的终端中) 输入 `python`，如果看到如下信息，说明 Python 已经装好了：

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

不然，如果看见类似 `"command not found"` 的错误，你就不得不下载安装 Python 了。参阅 *http://www.python.org/download/* 获取相关入门知识。 Python 的安装 简单而快捷。

Django 安装

本节中，我们介绍两种安装方式：官方发布版安装和通过 Subversion 安装。

官方发布版安装

大多数人会考虑从 http://www.djangoproject.com/download/ 下载安装最新的官方发布版。Django 使用了 Python 标准的 `distutils` 安装法，在 Linux 平台可能包括如下步骤：

1. 下载 tar 安装包，其文件名可能会是 `Django-0.96.tar.gz` 。
2. `tar xzvf Django-*.tar.gz` 。
3. `cd Django-*` 。
4. `sudo python setup.py install` 。

在windows平台下，我们建议使用 **7-Zip** 来处理各种格式的压缩文档，包括 ".tar.gz" 格式。7-Zip 可以从 http://www.djangoproject.com/r/7zip/ 下载得到 。

切换到另一个目录并启动 `python` 。如果一切就绪，你就可以导入 `django` 模块了：

```
>>> import django
>>> django.VERSION
(0, 96, None)
```

注意

Python 交互解释器是命令行窗口的程序，通过它可以交互式编写 Python 程序。 要启动它只需运行 `python` 命令。本书所有的 Python 示例代码均以在交互解释器中输入的方式展示。那三个大于号（ `>>>` ）是 Python 的提示符。

通过 subversion 安装 Django

如果想体验有一定风险的新特性，或者想为 Django 贡献代码的话，应该从Subverion 仓库下载安装 Django。

Subversion 是一种与 CVS 类似的免费开源版本控制系统，Django 开发团队使用它管理 Django 代码库的更新。你可以使用 Subversion 客户端获取最新的 Django 源代码，并可任何时候使用 *local checkout* 更新本地 Django 代码的版本，以获取 Django 开发者所做的最近更新和改进。

最新和最好的Django代码通常叫做*trunk（主干）*。Django开发团队就是使用trunk代码来运行产品级站点，并全力确保其稳定性。

遵循以下步骤以获取最新的 Django 主流代码：

1. 确保安装了 Subversion 客户端。可以从 http://subversion.tigris.org/ 免费下载该软件，并从 http://svnbook.red-bean.com/ 获取出色的文档。
2. 使用 `svn co http://code.djangoproject.com/svn/django/trunk djtrunk` 命令检出主流代码。
3. 创建 `site-packages/django.pth` 并将 `djtrunk` 目录添加进去，或者更新 `PYTHONPATH` 设置，将其指向 `djtrunk` 。
4. 将 `djtrunk/django/bin` 加入系统变量 PATH 中。该目录中包含一些像 `django-admin.py` 之类的管理工具。

提示：

如果之前没有接触过 `.pth` 文件，你可以从 http://www.djangoproject.com/r/python/site-module/ 中获取更多相关知识。

从 Subversion 完成下载并执行了前述步骤后，就没有必要再执行 `python setup.py install` 了，你刚才已经手动完成了安装！

由于 Django 主干代码的更新经常包括 bug 修正和特性添加，如果真的着迷的话，你可能每隔一小段时间就想更新一次。在 `djtrunk` 目录下运行 `svn update` 命令即可进行更新。当你使用这个命令时，Subversion 会联络 http://code.djangoproject.com，判断代码是否有更新，然后把上次更新以来的所有变动应用到本地代码。就这么简单。

安装数据库

使用 Django 的唯一先决条件是安装 Python。然而，本书所关注的是 Django 的亮点之一——基于数据库的网站开发，因此你必须安装某种类型的数据库来存储数据。

如果只是想开始摆弄 Django ，可以跳到《开始一个项目》小节。但请相信我们，到最后你还是会想要安装一个数据库。本书中所有例子都假定你安装了一个数据库系统。

本书写作时，Django 支持 3 种数据库引擎：

- PostgreSQL (http://www.postgresql.org/)
- SQLite 3 (http://www.sqlite.org/)
- MySQL (http://www.mysql.com/)

有人正在努力让它能够支持 SQL Server 和 Oracle 。Django 官方网站会经常发布所支持数据库的最新消息。

出于本书所讨论范围之外的原因，我们非常偏爱 PostgreSQL，因此我们将第一个提到它。当然，这里所列出的引擎和 Django 都配合得一样好。

作为开发工具，SQLite 特别值得注意。它是一种非常简单的进程级数据库引擎，无需任何服务器安装或配置工作。如果你只是想摆弄一下 Django ，它是到目前为止最容易安装的，甚至已经被包括在 Python 2.5 的标准类库之中。

在 Windows 平台上，获取数据库二进制驱动是件很棘手的工作。因为你刚开始接触 Django，我们推荐使用 Python 2.5 及其对 SQLite 的内建支持。编译二进制驱动是件令人沮丧的事。

在 Django 中使用 PostgreSQL

使用 PostgreSQL 的话，你需要从 http://www.djangoproject.com/r/python-pgsql/ 下载 `psycopg` 这个开发包。留意你所用的是版本 1 还是 2，稍后你会需要这项信息。

如果在 Windows 平台上使用 PostgreSQL，可以从 http://www.djangoproject.com/r/python-pgsql/windows/ 获取预编译的 `psycopg` 开发包的二进制文件。

在 Django 中使用 SQLite 3

如果使用 2.5 及更高版本的 Python，你无需再安装 SQLite 。但如果使用的是 2.4 或者更低版本的 Python，你所需要的 SQLite 3 不是从 http://www.djangoproject.com/r/sqlite/ 下载到的版本 2 以及从 http://www.djangoproject.com/r/python-sqlite/ 下载 `pysqlite` 。必须确保使用的是 2.0.3 或者更高版本的 `pysqlite` 。

在 Windows 平台上，可以跳过单独的 SQLite 二进制包安装工作，因为它们已被静态链接到 `pysqlite` 二进制开发包中。

在 Django 中使用 MySQL

Django 需要 4.0 或者更高版本的 MySQL，3.x 版不支持嵌套子查询以及其它一些相当标准的 SQL 语句。你还需要从 http://www.djangoproject.com/r/python-mysql/ 下载安装 `MySQLdb` 。

使用无数据库支持的 Django

正如之前提及过的，Django 并不是非得要数据库才可以运行。如果只用它提供一些不涉及数据库的动态页面服务，也同样可以完美运行。

尽管如此，还是要记住：Django 所捆绑的一些附加工具 *一定* 需要数据库，因此如果选择不使用数据库，你将不能使用这些功能。 (我们会在全书中标出这些功能。)

开始一个项目

项目 是 Django 实例一系列设置的集合，它包括数据库配置、Django 特定选项以及应用程序的特定设置。

如果第一次使用 Django，必须进行一些初始化设置工作。新建一个工作目录，例如 `/home/username/djcode/` ，然后进入该目录。

备忘

如果用的是 `setup.py` 工具进行的 Django 安装， `django-admin.py` 应该已被加入了系统路径中。如果是从 Subversion 检出的代码，则可以在 `djtrunk/django/bin` 中找到它。因为会经常用到 `django-admin.py` ，可以考虑把它加入系统搜索路径。在 Unix 上，你可以用 `sudo ln -s /path/to/django/bin/django-admin.py /usr/local/bin/django-admin.py` 这样的命令从 `/usr/local/bin` 中建立符号连接。在 Windows 平台上则需要更新 `PATH` 环境变量。

运行 `django-admin.py startproject mysite` 命令在当前目录创建一个 `mysite` 目录。

让我们看看 `startproject` 都创建了哪些内容：

```
mysite/
    __init__.py
    manage.py
    settings.py
    urls.py
```

包括下列这些文件：

- `__init__.py` ：让 Python 把该目录当成一个开发包 (即一组模块)所需的文件。
- `manage.py` ：一种命令行工具，可让你以多种方式与该 Django 项目进行交互。
- `settings.py` ：该 Django 项目的设置或配置。
- `urls.py` ：该 Django 项目的 URL 声明，即 Django 所支撑站点的内容列表

这个目录应该放哪儿？

有过 PHP 编程背景的话，你可能习惯于将代码都放在 Web 服务器的文档根目录 (例如 `/var/www` 这样的地方)。而在 Django 中，你不能这样做。把任何 Python 代码放到 Web 服务器的文档根目录中都不是个好主意，因为这样一来，你就要冒着别人透过页面直接看到代码的风险。这对于安全可不是件好事。

把代码放置在文档根目录 **之外** 的某些目录中。

开发服务器

Django 带有一个内建的轻量级 Web 服务器，可供站点开发过程中使用。我们提供这个服务器是为了让你快速开发站点，也就是说在准备发布产品之前，无需进行产品级 Web 服务器（比如 Apache）的配置工作。该开发服务器会监测代码变动并将其自动重载，这样一来，你可快速进行项目修改而无需作任何重启。

如果还没有进入 `mysite` 目录的话，现在进入其中，并运行 `python manage.py runserver` 命令。你将看到如下输出：

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

尽管对于开发来说，这个开发服务器非常得棒，但一定要打消在产品级环境中使用该服务器的念头。在同一时间，该服务器只能可靠地处理一次单个请求，并且没有进行任何类型的安全审计。发布站点前，请参阅第 20 章了解如何部署 Django 。

更改主机或端口

默认情况下， `runserver` 命令在 8000 端口启动开发服务器，且只监听本机连接。要想要更改服务器端口的话，可将端口作为命令行参数传入：

```
python manage.py runserver 8080
```

还可以改变服务器监听的 IP 地址。要和其他开发人员共享同一开发站点的话，该功能特别有用。下面的命令：

```
python manage.py runserver 0.0.0.0:8080
```

会让 Django 监听所有网络接口，因此也就让其它电脑可连接到开发服务器了。

既然服务器已经运行起来了，现在用网页浏览器访问 http://127.0.0.1:8000/ 。 你应该可以看到一个欢快的淡蓝色所笼罩的 Django 欢迎页面。一切正常！

下一章讲

既然一切都已经安装完毕，开发服务器也已经运行起来了，下一章中，让我们编写一些基础代码，演示如何使用 Django 提供网页服务。

第三章：动态网页基础

前一章中，我们解释了如何建立一个 Django 项目并启动 Django 开发服务器。当然，那个网站实际并没有干什么有用的事情，它所做的只是显示 It worked! 消息。让我们来做些改变。本章将介绍如何使用 Django 创建动态网页。

第一份视图：动态内容

我们的第一个目标是创建一个显示当前日期和时间的网页。这是一个不错的 *动态* 网页范例，因为该页面的内容不是静态的。相反，其内容是随着计算(本例中是对当前时间的计算)的结果而变化的。这个简单的范例既不涉及数据库，也不需要任何用户输入，仅输出服务器的内部时钟。

我们将编写一个 *视图函数* 以创建该页面。所谓的视图函数（或 *视图*），只不过是一个接受 Web 请求并返回 Web 响应的 Python 函数。实际上，该响应可以是一份网页的 HTML 内容、一次重定向、一条 404 错误、一份 XML 文档、一幅图片，或其它任何东西。视图本身包含返回该响应所需的任意逻辑。该段代码可以随意放置，只要在 Python 的路径设置中就可以了。没有其它要求——也可以说是没有任何奇特之处。为了给这些代码一个 *存身之处*，让我们在上一章所创建的 `mysite` 目录中新建一份名为 `views.py` 的文件。

以下是一个以 HTML 方式返回当前的日期与时间的视图 (view)，：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

我们逐行逐句地分析一遍这段代码：

> 首先，我们从 `django.http` 模块导入（import） `HttpResponse` 类。参阅附录 H 了解更多关于 `HttpRequest` 和 `HttpResponse` 的细节。
>
> 然后我们从 Python 标准库(Python 自带的实用模块集合)中导入（import) `datetime` 模块。`datetime` 模块包含几个处理日期和时间的函数（functions）和类(classes)，其中就包括返回当前时间的函数。
>
> 接下来，我们定义了一个叫做 `current_datetime` 的函数。这就是所谓的视图函数（view function）。每个视图函数都以一个 `HttpRequest` 对象为第一个参数，该参数通常命名为 `request` 。
>
> 注意视图函数的名称并不重要；并不一定非得以某种特定的方式命名才能让 Django 识别它。此处，我们称之为 `current_datetime` ，只是因为该名字明确地指出了它的功能，而它也可以被命名为 `super_duper_awesome_current_time` 或者其它同样莫名其妙的名字。Django 并不关心其名字。下一节将解释 Django 如何查找该函数。
>
> 函数中的第一行代码计算当前日期和时间，并以 `datetime.datetime` 对象的形式保存为局部变量 `now` 。

函数的第二行代码用 Python 的格式化字符串（format-string）功能构造了一段 HTML 响应。字符串里面的 `%s` 是占位符，字符串之后的百分号表示使用变量 `now` 的值替换 `%s` 。(是的，这段 HTML 不合法，但我们只不过是想让范例尽量保持简短而已。）

> 最后，该视图返回一个包含所生成响应的 `HttpResponse` 对象。每个视图函数都负责返回一个 `HttpResponse` 对象。（也有例外，但是我们稍后才会接触到。）

Django 时区 (Time Zone)

Django 包含一个默认为 `America/Chicago` 的 `TIME_ZONE` 设置。这可能不是你所居住的时区，因此你可以在 `settings.py` 文件中修改它。请参阅附录 E 了解更多细节。

将 URL 映射到视图

那么概括起来，该视图函数返回了包含当前日期和时间的一段 HTML 页面。但是如何告诉 Django 使用这段代码呢？这就是 *URLconfs* 登场的地方了。

URLconf 就像是 Django 所支撑网站的目录。它的本质是 URL 模式以及要为该 URL 模式调用的视图函数之间的映射表。你就是以这种方式告诉 Django，对于这个 URL 调用这段代码，对于那个 URL 调用那段代码。但必须记住的是视图函数必须位于 Python 搜索路径之中。

Python 搜索路径

Python 搜索路径 就是使用 `import` 语句时，Python 所查找的系统目录清单。

举例来说，假定你将 Python 路径设置为 `['','/usr/lib/python2.4/site-packages','/home/username/djcode/']`。如果执行代码 `from foo import bar`，Python 将会首先在当前目录查找 `foo.py` 模块(Python 路径第一项的空字符串表示当前目录)。如果文件不存在，Python将查找 `/usr/lib/python2.4/site-packages/foo.py` 文件。如果文件也不存在，它将尝试 `/home/username/djcode/foo.py`。最后，如果 *这个* 文件还不存在，它将引发 `ImportError` 异常。

如果对了解 Python 搜索路径值感兴趣，可以启动 Python 交互式解释程序，输入 `import sys`，接着输入 `print sys.path`。

通常，你不必关心 Python 搜索路径的设置。Python 和 Django 会在后台自动帮你处理好。(如果有兴趣了解的话，Python 搜索路径的设置工作是 `manage.py` 文件的职能之一。)

前一章中执行 `django-admin.py startproject` 时，该脚本会自动为你建了一份 URLconf（即 `urls.py` 文件）。让我们编辑一下这份文件。預設情况下它是下面这个样子：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.apps.foo.urls.foo')),

    # Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
)
```

让我们逐行逐句分析一遍这段代码：

- 第一行从 `django.conf.urls.defaults` 模块引入了所有的对象，其中包括了叫做 `patterns` 的函数。
- 第二行调用 `patterns()` 函数并将返回结果保存到 `urlpatterns` 变量。 `patterns()` 函数只传入了一个空字符串参数。其他代码行都被注释掉了。 (该字符串可用作视图函数的通用前缀，但目前我们将略过这种高级用法。)

当前应该注意是 `urlpatterns` 变量， Django 期望能从 `ROOT_URLCONF` 模块中找到它。该变量定义了 URL 以及用于处理这些 URL 的代码之间的映射关系。

默认情况下，URLconf 所有内容都被注释起来了——Django 应用程序还是白版一块。（旁注：这也就是上一章中 Django 显示“It worked!”页面的原因。如果 URLconf 为空，Django 会认定你才创建好新项目，因此也就显示那种信息。）

现在编辑该文件用来展示我们的 `current_datetime` 视图:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
)
```

我们做了两处修改。首先，我们从模块 (在 Python 的 import 语法中， `mysite/views.py` 转译为 `mysite.views`) 中引入了 `current_datetime` 视图。接着，我们加入了 `(r'^time/$', current_datetime)`, 这一行。该行就是所谓的 *URLpattern* ，它是一个 Python 元组,其第一个元素是简单的正则表达式，第二个元素是为该模式应用的视图函数。

简单来说，我们只是告诉 Django，所有指向 URL `/time/` 的请求都应由 `current_datetime` 这个视图函数来处理。

下面是一些需要注意的地方：

> 注意，该例中，我们将 `current_datetime` 视图函数作为对象传递，而不是调用它。这是 Python (及其它动态语言的) 的一个重要特性：函数是一级对象（first-class objects）， 也就是说你可以像传递其它变量一样传递它们。很酷吧？
>
> `r'^time/$'` 中的 `r` 表示 `'^time/$'` 是一个原始字符串。这样一来就可以避免

System Message: WARNING/2 (`<string>`, line 232)

Block quote ends without a blank line; unexpected unindent.

正则表达式有过多的转义字符。

> 不必在 `'^time/$'` 前加斜杠（/）来匹配 `/time/` ， 因为 Django 会自动在每个表 达式前添加一个斜杠。乍看起来，这好像有点奇怪，但是 URLconfs 可能由其它的 URLconfs 所引用, 所以不加前面的斜杠可让事情简单一些。这一点在第 8 章中将有进一步阐述。
>
> 上箭头 `^` 和美元符号 `$` 符号非常重要。上箭头要求表达式对字符串的头部进行匹配，美元符号则要求表达式对字符串的尾部进行匹配。
>
> 最好还是用范例来说明一下这个概念。如果我们用 `'^time/'` （结尾没有$), 那么以 `time/` 开始的 *任意* URL 都会匹配，比如 `/time/foo` 和 `/time/bar` ， 不仅仅是 `/time/` 。同样的，如果我们去掉最前面的 ^ (`'time/$'`), Django 一样会匹配由 `time/` 结束的 *任意* URL `/time/` ，比如 `/foo/bar/time/` 。 因此，我们必须同时用上 ^ 和 $ 来精确匹配 URL `/time/` 。不能多也不能少。
>
> 你可能想如果有人请求 `/time` 也可以同样处理。如果 `APPEND_SLASH` 的 设置是 `True` 的话，系统会重定向到 `/time/` ，这样就可以一样处理了。 (有关内容请查看附录 E)

启动Django开发服务器来测试修改好的 URLconf, 运行命令行 `python manage.py runserver` 。 (如果你让它一直运行也可以，开发服务器会自动监测代码改动并自动重新载入，所以不需要手工重启） 开发服务器的地址是 `http://127.0.0.1:8000/` ，打开你的浏览器访问 `http://127.0.0.1:8000/time/` 。 你就可以看到输出结果了。

万岁！你已经创建了第一个Django的web页面。

正则表达式

正则表达式 (或 *regexes*) 是通用的文本模式匹配的方法。Django URLconfs 允许你 使用任意的正则表达式来做强有力的URL映射，不过通常你实际上可能只需要使用很少的一 部分功能。下面就是一些常用通用模式：

符号	匹配
. (dot)	任意字符
`\d`	任意数字
`[A-Z]`	任意字符, A-Z (大写)
`[a-z]`	任意字符, a-z (小写)
`[A-Za-z]`	任意字符, a-z (不区分大小写)
`+`	匹配一个或更多 (例如, `\d+` 匹配一个或 多个数字字符)
`[^/]+`	不是/的任意字符
`*`	匹配0个或更多 (例如, `\d*` 匹配0个 或更多数字字符)
`{1,3}`	匹配1个到3个（包含）

有关正则表达式的更多内容，请访问 http://www.djangoproject.com/r/python/re-module/.

Django是怎么处理请求的

我们必须对刚才所发生的几件事情进行一些说明。它们是运行Django开发服务器和构造Web页面请求的本质所在。

> 命令 `python manage.py runserver` 从同一目录载入文件 `settings.py`。 该文件包含了这个特定的Django实例所有的各种可选配置，其中一个最重要的配置就是 `ROOT_URLCONF`。 `ROOT_URLCONF` 告诉Django哪个Python模块应该用作本网站的 URLconf。
>
> 还记得 `django-admin.py startproject` 创建的文件 `settings.py` 和 `urls.py` 吗？ 这是系统自动生成的 `settings.py` 里 `ROOT_URLCONF` 默认设置是 `urls.py`。
>
> 当访问 URL `/time/` 时，Django 根据 `ROOT_URLCONF` 的设置装载 URLconf。 然后按顺序逐个匹配URLconf里的URLpatterns，直到找到一个匹配的。当找到这个匹配 的URLpatterns就调用相关联的view函数，并把 `HttpRequest` 对象作为第一个参数。 （稍后再给出 `HttpRequest` 的更多信息）

该 view 函数负责返回一个 `HttpResponse` 对象。

你现在知道了怎么做一个 Django-powered 页面了，真的很简单，只需要写视图函数并用 URLconfs把它们和URLs对应起来。你可能会认为用一系列正则表达式将URLs映射到函数也许会比较慢，但事实却会让你惊讶。

Django如何处理一个请求：完整的细节

除了刚才所说到的简明URL-to-view映射方式之外，Django在请求处理方面提供了大量的灵活性。

通过 URLconf 解析到哪个视图函数来返回 `HttpResponse` 可以通过中间件（middleware) 来短路或者增强。关于中间件的细节将在第十五章详细谈论，这里给出 图3-1 让你先了解 大体概念.

图3-1：Django请求和回应的完整流程

当服务器收到一个HTTP请求以后，一个服务器特定的 *handler* 会创建 `HttpRequest` 并传递给下一个组件并处理。

这个 handler 然后调用所有可用的Request或者View中间件。这些类型的中间件通常是用来 增强 `HttpRequest` 对象来对一些特别类型的request做些特别处理。只要其中有一个 返回 `HttpResponse`，系统就跳过对视图的处理。

即便是最棒的程序员也会有出错的时候， 这个时候 *异常处理中间件（exception middleware）* 可以帮你的大忙。如果一个视图函数抛出异常，控制器会传递给异常处理中间件处理。如果这个 中间件没有返回 `HttpResponse`，意味着它不能处理这个异常，这个异常将会再次抛出。

即便是这样，你也不用担心。Django包含預設的视图来生成友好的404 和 500 回应（response）。

最后， *response middleware* 做发送 `HttpResponse` 给浏览器之前的后处理或者清除 请求用到的相关资源。

URL配置和弱配對

现在是好时机来指出Django和URL配置背后的哲学： *松耦合* 原则。简单的说，松耦合是一个 重要的保证互换性的软件开发方法。如果两段代码是松耦合的，那么改动其中一段代码不会 影响另一段代码，或者只有很少的一点影响。

Django的URL配置就是一个很好的例子。在Django的应用程序中，URL的定义和视图函数之间是松 耦合的，换句话说，决定URL返回哪个视图函数和实现这个视图函数是在两个不同的地方。这使得 开发人员可以修改一块而不会影响另一块。

相比之下，其他的Web开发平台紧耦合和URL到代码中。在典型的PHP (http://www.php.net/) 应用，URL的设计是通过放置代码的目录来实现。在早期的 CherryPy Python Web framework (http://www.cherrypy.org/) 中，URL对应处理的方法名。这可能在短期看起来是便利之举， 但是长期会带来难维护的问题。

比方说，考慮先前写的视图函数，这个函数显示当前日期和时间。如果我们想把它的URL 从原来的 `/time/` 改变到 `/currenttime/` ，我们只需要快速的修改一下URL配置即可， 不用担心这个函数的内部实现。同样的，如果我们想要修改这个函数的内部实现也不用担心会影响 到对应的URL。此外，如果我们想要输出这个函数到 *一些* URL， 我们只需要修改URL配置而不用 去改动视图的代码。

Django大量应用松耦合。我们将在本书中继续给出这一重要哲学的相关例子。

404 错误

In our URLconf thus far, weve defined only a single URLpattern: the one that handles requests to the URL `/time/` . What happens when a different URL is requested?

让我们试试看，运行Django开发服务器并访问类似 `http://127.0.0.1:8000/hello/` 或者 `http://127.0.0.1:8000/does-not-exist/` ，甚至 `http://127.0.0.1:8000/` (网站根目录)。你将会看到一个 “Page not found” 页面（图 3－2）。（挺漂亮的，是吧？ 你会喜欢上我们的配色方案的;-) 如果请求的URL没有在URL配置里设置，Django就会显示这个页面。

图 3-2. Django的 404 页面

这个页面的功能不只是显示404的基本错误信息，它同样精确的告诉你Django使用了哪个URL配置和 这个配置里的每一个模式。这样，你应该能了解到为什么这个请求会抛出404错误。

当然，这些敏感的信息应该只呈现给你－开发者。如果是部署到了因特网上的站点就不应该暴露 这些信息。出于这个考虑，这个“Page not found”页面只会在 *调试模式（debug mode）* 下 显示。我们将在以后说明怎么关闭调试模式。现在，你只需要知道所有的Django项目在创建后都 是在调试模式下的，如果关闭了调试模式，结果将会不一样。

第二个视图：动态URL

在我们的第一个视图范例中，尽管内容是动态的，但是URL （ `/time/` ）是静态的。在 大多数动态web应用程序，URL通常都包含有相关的参数。

让我们创建第二个视图来显示当前时间和加上时间偏差量的时间，设计是这样的： `/time/plus/1/` 显示当前时间＋1个小时的页面 `/time/plus/2/` 显示当前时间＋2个小时的页面 `/time/plus/3/` 显示当前时间＋3个小时的页面，以此类推。

新手可能会考虑写不同的视图函数来处理每个时间偏差量，URL配置看起来就象这样：

```
urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/1/$', one_hour_ahead),
    (r'^time/plus/2/$', two_hours_ahead),
    (r'^time/plus/3/$', three_hours_ahead),
    (r'^time/plus/4//$', four_hours_ahead),
)
```

很明显，这样处理是不太妥当的。不但有很多冗余的视图函数，而且整个应用也被限制了只支持 预先定义好的时间段，2小时，3小时，或者4小时。如果哪天我们要实现 *5* 小时，我们就 不得不再单独创建新的视图函数和配置URL，既重复又混乱。我们需要在这里做一点抽象，提取 一些共同的东西出来。

关于漂亮URL的一点建议

如果你有其他Web开发平台的经验，例如PHP或者JAVA，你可能会想，好吧，让我们来用一个 查询字符串参数来表示它们吧，例如 `/time/plus?hours=3` ，哪个时间段用 `hours` 参数代表，URL的查询字符串(query string)是URL里 `?` 后面的字符串。

你 *可以* 在Django里也这样做 (如果你真的想要这样做，我们稍后会告诉你怎么做）， 但是Django的一个核心理念就是URL必须看起来漂亮。URL `/time/plus/3/` 更加清晰， 更简单，也更有可读性，可以很容易的大声念出来，因为它是纯文本，没有查询字符串那么 复杂。漂亮的URL就像是高质量的Web应用的一个标志。

Django的URL配置系统可以使你很容易的设置漂亮的URL，而不是相反

带通配符的URL匹配模式

继续我们的 `hours_ahead` 范例，让我们在URL模式里使用通配符。我们前面讲到，URL模式 是一个正则表达式，因此，我们可以使用正则表达式模式 `\d+` 来匹配一个或多个数字：

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/\d+/$', hours_ahead),
)
```

这个URL模式将匹配类似 `/time/plus/2/` , `/time/plus/25/` ,甚至 `/time/plus/100000000000/` 的任何URL。更进一步，让我们把它限制在最大允许99个小时， 这样我们就只允许一个或两个数字，正则表达式的语法就是 `\d{1,2}` :

```
(r'^time/plus/\d{1,2}/$', hours_ahead),
```

备注

在建造Web应用的时候，尽可能多考虑可能的数据输入是很重要的，然后决定哪些我们可以接受。 在这里我们就设置了99个小时的时间段限制。

现在我们已经设计了一个带通配符的URL，我们需要一个方法把它传递到视图函数里去，这样 我们只用一个视图函数就可以处理所有的时间段了。我们使用圆括号把参数在URL模式里标识 出来。在这个例子中，我们想要把这些数字作为参数，用圆括号把 `\d{1,2}` 包围起来：

```
(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

如果你熟悉正则表达式，那么你应该已经了解，正则表达式也是用圆括号来从文本里 *提取* 数据的。

最终的 `current_datetime` URLconf，包含我们前面的视图，看起来像这样：

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

现在开始写 `hours_ahead` 视图。

编码次序

這個例子中，我們先寫了URLpattern，然後是視圖，但是在前面的例子中，我們先寫了視圖，然後是URLpattern。哪種技術更好？嗯，怎麼說呢，每個開發者是不一樣的。

如果你是喜欢从总体上来把握事物（注：或译为“大局观”）类型的人，你应该会想在项目开始 的时候就写下所有的URL配置。这会给你带来一些好处，例如，给你一个清晰的to-do列表，让你 更好的定义视图所需的参数。

如果你从更像是一个自底向上的开发者，你可能更喜欢先写视图， 然后把它们挂接到URL上。这同样是可以的。

最后，取决与你喜欢哪种技术，两种方法都是可以的。

`hours_ahead` 和我们以前写的 `current_datetime` 很象，关键的区别在于： 它多了一个额外参数，时间差。 `views.py` 修改如下:

```
def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
```

```
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

让我们一次一行通一下这些代码：

就像我们在我们的 `current_datetime` 视图中所作的一样，我们导入 `django.http.HttpResponse` 类和 `datetime` 模块。

视图函数, `hours_ahead` , 有 *两个* 参数: `request` 和 `offset` .

`request` 是一个 `HttpRequest` 对象, 就像在 `current_datetime` 中一样. 再说一次好了: 每一个视图 *总是* 以一个 `HttpRequest` 对象作为 它的第一个参数。

`offset` 是从匹配的URL里提取出来的。例如： 如果URL是 `/time/plus/3/` 那么 `offset` 是字符串 `'3'`， 如果URL是 `/time/plus/21/` ，那么 `offset` 是字符串 `'21'`， 注意，提取的字符串总是 *字符串* ，不是整数，即便都是数字组成，就象 `'21'` 。

在这里我们命名变量为 `offset` ，你也可以任意命名它，只要符合Python 的语法。变量名是无关紧要的，重要的是它的位置，它是这个函数的第二个 参数 (在 `request` 的后面）。你还可以使用关键字来定义它，而不是用 位置。详情请看第八章。

我们在这个函数中要做的第一件事情就是在 `offset` 上调用 `int()` . 这会把这个字符串值转换为整数。

注意Python可能会在你调用 `int()` 来转换一个不能转换成整数时抛出 `ValueError` 异常，例如字符串 `'foo'` 。 当然，在这个范例中我们不用担心这个问题，因为我们已经确定 `offset` 是 只包含数字字符的字符串。因为正则表达式 `(\d{1,2})` 只提取数字字符。 这也是URL配置的另一个好处：提供了清晰的输入数据有效性确认。

下一行显示了我们为什么调用 `int()` 来转换 `offset` 。 这一行我们要 计算当前时间加上这个时间差 `offset` 小时，保存结果到变量 `dt` 。 `datetime.timedelta` 函数的参数 `hours` 必须是整数类型。

这行和前面的那行的一个微小差别就是，它使用带有两个值的Python的格式化字符串功能， 而不仅仅是一个值。因此，在字符串中有两个 `%s` 符号和一个以进行插入的值的元组： `(offset, dt)` 。

最后，我们再一次返回一个HTML的 `HttpResponse` ，就像我们在 `current_datetime` 做的一样。

在完成视图函数和URL配置编写后，启动Django开发服务器，用浏览器访问 `http://127.0.0.1:8000/time/plus/3/` 来确认它工作正常。 然后是 `http://127.0.0.1:8000/time/plus/5/` 。再然后是 `http://127.0.0.1:8000/time/plus/24/` 。最后，访问 `http://127.0.0.1:8000/time/plus/100/` 来检验URL配置里设置的模式是否只 接受一个或两个数字; Django会显示一个 Page not found error 页面, 和以前看到的 404 错误一样。访问URL `http://127.0.0.1:8000/time/plus/` (*没有* 定义时间差) 也会抛出404错误。

你现在已经注意到 `views.py` 文件中包含了两个视图， `views.py` 看起来象这样：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)

def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Django 漂亮的出错页面

花几分钟时间欣赏一下我们写好的Web应用程序，然后我们再来搞点小破坏。我们故意在 `views.py` 文件中引入一项Python 错误，注释掉 `hours_ahead` 视图中的 `offset = int(offset)` 一行。

```
def hours_ahead(request, offset):
    #offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

启动开发服务器，然后访问 `/time/plus/3/` 。你会看到一个包含大量信息的出错页，最上面 的一条 `TypeError` 信息是： `"unsupported type for timedelta hours component: str"` 。

怎么回事呢？是的， `datetime.timedelta` 函数要求 `hours` 参数必须为整型， 而我们注释掉了将 `offset` 转为整型的代码。这样导致 `datetime.timedelta` 弹出 `TypeError` 异常。这是所有程序员某个时候都可能碰到的一种典型错误。

这个例子是为了展示 Django 的出错页面。我们来花些时间看一看这个出错页，了解一下其中 给出了哪些信息。

以下是值得注意的一些要点：

在页面顶部，你可以得到关键的异常信息：异常数据类型、异常的参数 (如本例中的 `"unsupported type"` ）、在哪个文件中引发了异常、出错的行号等等。

在关键异常信息下方，该页面显示了对该异常的完整 Python 追踪信息。这类似于你在 Python 命令行解释器中获得的追溯信息，只不过后者更具交互性。对栈中的每一帧，Django 均显示了其文件名、函数或方法名、行号及该行源代码。

点击该行代码 (以深灰色显示)，你可以看到出错行的前后几行，从而得知相关上下文情况。

点击栈中的任何一帧的“Local vars”可以看到一个所有局部变量的列表，以及在出错 那一帧时它们的值。这些调试信息是无价的。

注意“Traceback”下面的“Switch to copy-and-paste view”文字。点击这些字，追溯会 切换另一个视图，它让你很容易地复制和粘贴这些内容。当你想同其他人分享这些异常 追溯以获得技术支持时（比如在 Django 的 IRC 聊天室或邮件列表中），可以使用它。

接下来的“Request information”部分包含了有关产生错误的 Web 请求的大量信息： GET 和 POST、cookie 值、元数据（象 CGI 头）。在附录H里给出了request的对象的 完整参考。

Request信息的下面，“Settings”列出了 Django 使用的具体配置信息。同样在附录E 给出了settings 配置的完整参考。现在，大概浏览一下，对它们有个大致印象就好了。

Django 的出错页某些情况下有能力显示更多的信息，比如模板语法错误。我们讨论 Django 模板系统时再说它们。现在，取消 `offset = int(offset)` 这行的注释，让它重新正常 工作。

不知道你是不是那种使用小心放置的 `print` 语句来帮助调试的程序员？你其实可以用 Django 出错页来做这些，而不用 `print` 语句。在你视图的任何位置，临时插入一个 `assert False` 来触发出错页。然后，你就可以看到局部变量和程序语句了。（还有更高级的办法来调试 Django 视图，我们后来再说，但这个是最快捷最简单的办法了。）

最后，很显然这些信息很多是敏感的，它暴露了你 Python 代码的内部结构以及 Django 配置，在 Internet 上公开这信息是很愚蠢的。不怀好意的人会尝试使用它攻击你的 Web 应用程序，做些下流之事。因此，Django 出错信息仅在 debug 模式下才会显现。我们稍后 说明如何禁用 debug 模式。现在，你只要知道 Django 服务器在你开启它时默认运行在 debug 模式就行了。（听起来很熟悉？“Page not found”错误，“404 错误”一节也这样描述过。）

下一章将要讲?

System Message: WARNING/2 (`<string>`, line 995)

Title underline too short.

```
下一章将要讲?
``````````
```

我们现在已经学会了怎么在Python代码里硬编码HTML代码来处理视图。可惜的是，这种方法通常不是一个好方法。幸运的是，Django内建有一个简单有强大的模板处理引擎来让你分离两种工作：设计HTML页面和编写Python代码。下一
```

章我们将深入到Django的模板引擎里去。

第四章 Django模板系统

在之前的章节中，你可能觉得例子中视图返回文本有点不妥。即是， HTML 是直接写在 Python 代码中的。

这种做法会导致这些问题：

- 要做任何设计上的更改就必须改写 Python 代码。网站的设计风格的更变一般来说会比更改后台的

  Ptyhon 代码来得频繁，因此如果能够更改设计而不用更改 Python 变得尤为方便。

- Python 代码编写和 HTML 设计是两项不同的工作，大多数专业的网站开发环境都将他们分配给不同的人员（甚至不同部门）来完成。设计人员和 HTML/CSS 编写人员都不应该通过编辑 Python 代码来完成自己的工作；他们应该处理的是 HTML。

- 同理，程序员编写 Python 代码和设计人员制作模板同时进行的工作方式效率是最高的，远胜于让一个人等待另一个人完成对某个既包含 Python 又包含 HTML 的文件的编辑工作。

基于这些原因，将页面的设计和Python的代码分离开会更干净简洁更容易维护。我们可以使用 Django的 *模板系统* (Template System)来实现这种模式，这就是本章要具体讨论的问题。

模板系统基本知识

模板系统基本知识

让我们深入分析一个简单的例子模板。该模板描述了一个向某个与公司签单人员致谢 HTML 页面。可将其视为一个格式信函：

```
<html>
<head><title>Ordering notice</title></head>

<body>

<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>

{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

该模板是一段添加了些许变量和模板标签的基础 HTML 。让我们逐句过一遍：

> 用两个大括号括起来的文字（例如 `{{ person_name }}` ）是 *变量(variable)* 。这意味着将按照给定的名字插入变量的值。如何指定变量的值呢？稍后就会说明。
>
> 被大括号和百分号包围的文本(例如 `{% if ordered_warranty %}`)是 *模板标签(template tag)* 。标签(tag)定

义比较明确，即：仅通知模板系统完成某些工作。

这个示例模板包含两个标签(tag): `{% for item in item_list %}` 标签(一个 `for` 标签) 和 `{% if ordered_warranty %}` 标签 (一个 `if` 标签)。

`for` 标签用于构建简单的循环，允许你遍历循环中的每一项。 `if` 标签，正如你所料，是用来执行逻辑判断的。在这个例子中标签检测 `ordered_warranty` 变量值是否为 `True` 。 如果是，模板系统将显示 `{% if ordered_warranty %}` 与 `{% endif %}` 之间的所有内容。 如果不是模板系统不会显示它。它当然也支持 `{% else %}` 以及其他多种逻辑判断方式。

最后，这个模板的第二段落有一个 *filter* 过滤器的例子,它能让你用来转换变量的输出， 在这个例子中, `{{ship_date|date:"F j, Y" }}` 将变量 `ship_date` 用 `date` 过滤器来转换,转换的参数是 `"F j, Y"` . `date` 过滤器根据指定的参数进行格式输出。过滤器是用管道字符(`|`)来调用的,就和Unix管道一样.

Django 模板含有很多内置的tags和filters,我们将陆续进行学习. 附录F列出了很多的tags和filters的列表,熟悉这些列表对你来说是个好建议. 学习完第十章，你就明白怎么去创建自己的filters和tags了.

如何使用模板系统

想要在Python代码中使用模板系统，只需遵循下面两个步骤：

1. 可以用原始的模板代码字符串创建一个 `Template` 对象， Django同样支持用指定模板文件路径的方式来创建 `Template` 对象;
2. 调用 `Template` 对象的 `render()` 方法并提供给他变量(i.e., 内容). 它将返回一个完整的模板字符串内容,包含了所有标签块与变量解析后的内容.

以下部分逐步的详细介绍

创建模板对象

创建一个 `Template` 对象最简单的方法就是直接实例化它。 `Template` 类就在 `django.template` 模块中，构造函数接受一个参数，原始模板代码。让我们深入挖掘一下 Python的解释器看看它是怎么工作的。

关于交互式解释器的例子

在本书中，我们喜欢用和Python解释器的交互来举例。你可以通过三个> (`>>>`) 识别它们，它们相当于Python解释器的提示符。如果你要拷贝例子，请不要拷贝这3个>字符。

多行语句则在前面加了3个小数点(. . .)，例如：

```
>>> print """This is a
... string that spans
... three lines."""
This is a
string that spans
three lines.
>>> def my_function(value):
...     print value
>>> my_function('hello')
hello
```

这3个点是Python解释器自动加入的，不需要你的输入。我们包含它们是为了忠实呈现解释器的 真实输出。同样道理，拷贝时不要拷贝这3个小数点符号。

转到project目录（在第二章由 `django-admin.py startproject` 命令创建），输入命令 `python manage.py shell` 启动交互界面。下面是一些基本操作：

```
>>> from django.template import Template
>>> t = Template("My name is {{ name }}.")
>>> print t
```

如果你跟我们一起做，你将会看到下面的内容：

```
<django.template.Template object at 0xb7d5f24c>
```

`0xb7d5f24c` 每次都会不一样，这没什么关系；这只是Python运行时 `Template` 对象的ID。

Django 设置

当你使用Django时，你需要告诉Django使用哪个配置。在交互模式下，通常运行命令 `python manage.py shell` 来做这个，附录E里还有一些其他的一些选项。

> 当你创建一个 `Template` 对象，模板系统在内部编译这个模板到内部格式，并做优化，做好渲染的准备。如果你的模板语法有错误，那么在调用 `Template()` 时就会抛出``TemplateSyntaxError`` 异常：

```
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ...
  django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

系统会在下面的情形抛出 `TemplateSyntaxError` 异常：

- 无效的块标签
- 无效的参数
- 无效的过滤器
- 过滤器的参数无效
- 无效的模板语法
- 未封闭的块标签 （针对需要封闭的块标签）

模板渲染

一旦你创建一个 `Template` 对象，你可以用 *context* 来传递数据给它。一个context是一系列变量和它们值的集合。模板使用它来赋值模板变量标签和执行块标签。

context在Django里表现为 `Context` 类，在 `django.template` 模块里。 它的构造函数有一个可选参数：一个字典（映射变量和它们的值）。调用 `Template` 对象 的 `render()` 方法并传递context来填充模板：

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ name }}.")
>>> c = Context({"name": "Stephane"})
>>> t.render(c)
'My name is Stephane.'
```

字典和Contexts

Python的字典数据类型就是关键字和它们值的一个映射。 `Context` 和字典很类似， `Context` 还提供更多的功能，请看第十章。

变量名必须由英文字符开始 （A-Z或a-z）并可以包含数字字符、下划线和小数点。 （小数点在这里有特别的用途，稍后我们会讲到）变量是大小写敏感的。

下面是编写模板并渲染的示例：

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled
... to ship on {{ ship_date|date:"F j, Y" }}.</p>
```

```
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'product': 'Super Lawn Mower',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': True})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for ordering Super Lawn Mower from
Outdoor Equipment. It's scheduled \nto ship on April 2, 2009.</p>\n\n\n
<p>Your warranty information will be included in the packaging.</p>\n\n\n
<p>Sincerely,<br />Outdoor Equipment</p>"
```

让我们逐句看看这段代码：

首先我们导入（import）类 `Template` 和 `Context`，它们都在模块 `django.template` 里。

我们把模板原始文本保存到变量 `raw_template`。注意到我们使用了三个引号来 标识这些文本，因为这样可以包含多行。这是Python的一个语法。

接下来，我们创建了一个模板对象 `t`，把 `raw_template` 作为 `Template` 类 的构造的参数。

我们从Python的标准库导入 `datetime` 模块，以后我们将会使用它。

然后，我们创建一个 `Context` 对象， `c`。 `Context` 构造的参数是Python 字典数据类型，在这里，我们给的参数是 `person_name` 值为 `'John Smith'`, `product` 值为 `'Super Lawn Mower'`，等等。

最后，我们在模板对象上调用 `render()` 方法，传递 context参数给它。 这是返回渲染后的模板的方法，它会替换模板变量为真实的值和执行块标签。

注意，warranty paragraph显示是因为 `ordered_warranty` 的值为 `True`. 注意时间的显示， `April 2, 2009`, 它是按 `'F j, Y'` 格式显示的。（我们很快就会在 `date` 过滤器解释这些格式）

如果你是Python初学者，你可能在想为什么输出里有回车换行的字符(`'\n'`)而不是 显示回车换行？因为这是Python交互解释器的缘故：调用 `t.render(c)` 返回字符串， 解释器缺省显示这些字符串的 *真实内容呈现*，而不是打印这个变量的值。 要显示换行而不是 `'\n'`，使用 `print` 语句： `print t.render(c)`。

这就是使用Django模板系统的基本规则：写模板，创建 `Template` 对象，创建 `Context`， 调用 `render()` 方法。

同一模板，多个上下文

一旦有了 `模板` 对象，你就可以通过它渲染多个背景（context），例如：

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print t.render(Context({'name': 'John'}))
Hello, John
>>> print t.render(Context({'name': 'Julie'}))
Hello, Julie
>>> print t.render(Context({'name': 'Pat'}))
Hello, Pat
```

无论何时像这样使用同一模板源渲染多个背景，只创建 *一次* `模板` 对象，然后对它多次调用 `render()` 将会更加高效。

```
# Bad
for name in ('John', 'Julie', 'Pat'):
    t = Template('Hello, {{ name }}')
    print t.render(Context({'name': name}))

# Good
```

```
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
    print t.render(Context({'name': name}))
```

Django 模板解析非常快捷。大部分的解析工作都是在后台通过对简短正则表达式一次性调用来完成。这和基于 XML 的模板引擎形成鲜明对比，那些引擎承担了 XML 解析器的开销，且往往比 Django 模板渲染引擎要慢上几个数量级。

背景变量的查找

在到目前为止的例子中，我们通过 context 传递的简单参数值主要是字符串，还有一个 `datetime.date` 范例。然而，模板系统能够非常简洁地处理更加复杂的数据结构，例如list、dictionary和自定义的对象。

在 Django 模板中遍历复杂数据结构的关键是句点字符 (.)。使用句点可以访问字典的键值、属性、索引和对象的方法。

最好是用几个例子来说明一下。比如，假设你要向模板传递一个 Python 字典。要通过字典键访问该字典的值，可使用一个句点：

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

同样，也可以通过句点来访问对象的属性。比方说， Python 的 `datetime.date` 对象有 `year` 、 `month` 和 `day` 几个属性，你同样可以在模板中使用句点来访问这些属性：

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
>>> d.day
2
>>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

下例使用了一个自定义类：

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

句点还用于调用对象的方法。例如，每个 Python 字符串都有 `upper()` 和 `isdigit()` 方法，你在模板中可以使用同样的句点语法来调用它们：

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

注意你不能在方法调用中使用圆括号。而且也无法给该方法传递参数；你只能调用不需参数的方法。（我们将在本章稍后部分解释该设计观。）

最后，句点也可用于访问列表索引，例如：

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

不允许使用负数列表索引。像 `{{ items.-1 }}` 这样的模板变量将会引发

`TemplateSyntaxError` 异常。

Python 列表类型

Python列表类型的索引是从0开始的，第一个元素的索引是0，第二个是1，以此类推。

句点查找规则可概括为：当模板系统在变量名中遇到点时，按照以下顺序尝试进行查找：

- 字典查找 （比如 `foo["bar"]`)
- 属性查找 (比如 `foo.bar`)
- 方法调用 （比如 `foo.bar()`)
- 列表类型索引查找 (比如 `foo[bar]`)

系统使用所找到的第一个有效类型。这是一种短路逻辑。

句点查找可以多级深度嵌套。例如在下面这个例子中 `{{person.name.upper}}` 会转换成字典类型查找（ `person['name']`) 然后是方法调用（ `upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

方法调用行为

方法调用比其他类型的查找略为复杂一点。以下是一些注意事项：

在方法查找过程中，如果某方法抛出一个异常，除非该异常有一个 `silent_variable_failure` 属性并且值为 `True` ，否则的话它将被传播。如果该异常 *确有* 属性 `silent_variable_failure` ，那么（所查找）变量将被渲染为空字符串，例如：

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError, "foo"
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

仅在方法无需传入参数时，其调用才有效。否则，系统将会转移到下一个查找类型（列表索引查找）。

显然，有些方法是有副作用的，好的情况下允许模板系统访问它们可能只是干件蠢事，坏的情况下甚至会引发安全漏洞。

例如，你的一个 `BankAccount` 对象有一个 `delete()` 方法。不应该允许模板包含像 `{{account.delete}}` 这样的方法调用。

要防止这样的事情发生，必须设置该方法的 `alters_data` 函数属性：

```
def delete(self):
    # Delete the account
delete.alters_data = True
```

模板系统不会执行任何以该方式进行标记的方法。也就是说，如果模板包含了 `{{account.delete}}`,该标签不会调用 `delete()` 方法。它只会安静地失败（并不会引发异常）。

如何处理无效变量

默认情况下，如果一个变量不存在，模板系统会把它展示为空字符串，不做任何事情地表示失败，例如：

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

系统静悄悄地表示失败，而不是引发一个异常，因为这通常是人为错误造成的。这种情况下，因为变量名有错误的状况或名称， 所有的查询都会失败。现实世界中，对于一个web站点来说，如果仅仅因为一个小的模板语法错误而造成无法访问，这是不可接受的。

注意，我们是可以有机会通过更改Django的配置以在这点上改变Django的默认行为的。 我们会在第10章进行进一步的讨论的。

玩一玩上下文(context)对象

多数时间，你可以通过传递一个完全填充(full populated)的字典给 `Context()` 来初始化 `上下文(Context)` 。 但是初始化以后，你也可以从``上下文(Context)`` 对象添加或者删除条目，使用标准的Python字典语法(syntax):

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

基本的模板标签和过滤器

像我们以前提到过的，模板系统带有内置的标签和过滤器。 下面的章节提供了一个多数通用标签和过滤器的简要说明。

标签

if/else

`{% if %}` 标签检查(evaluate)一个变量，如果这个变量为真（即，变量存在，非空，不是布尔值假），系统会显示在 `{% if %}` 和 `{% endif %}` 之间的任何内容，例如：

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

`{% else %}` 标签是可选的：

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% endif %}
```

Python 的“真值”

在python中空的列表 (`[]`)，tuple(`()`)，字典(`{}`)，字符串(`''`)，零(`0`)，还有 `None` 对象，在逻辑判断中都为假，其他的情况都为真。

`{% if %}` 标签接受 `and` ， `or` 或者 `not` 关键字来对多个变量做判断 ，或者对变量取反（ `not`)，例如：

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches. (OK, so
    writing English translations of Boolean logic sounds
    stupid; it's not our fault.)
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

`{% if %}` 标签不允许在同一个标签中同时使用 `and` 和 `or` ，因为逻辑上可能模糊的，例如，如下示例是错误的：

```
{% if athlete_list and coach_list or cheerleader_list %}
```

系统不支持用圆括号来组合比较操作。如果你发现需要组合操作，你可以考虑用逻辑语句来简化 模板的处理。例如，你需要组合 `and` 和 `or` 做些复杂逻辑判断，可以使用嵌套的 `{% if %}` 标签，示例如下：

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

多次使用同一个逻辑操作符是没有问题的，但是我们不能把不同的操作符组合起来。比如这样的代码是没问题的：

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

并没有 `{% elif %}` 标签，请使用嵌套的 `{% if %}` 标签来达成同样的效果：

```
{% if athlete_list %}
    <p>Here are the athletes: {{ athlete_list }}.</p>
{% else %}
    <p>No athletes are available.</p>
    {% if coach_list %}
```

```
        <p>Here are the coaches: {{ coach_list }}.</p>
    {% endif %}
{% endif %}
```

一定要用 `{% endif %}` 关闭每一个 `{% if %}` 标签。否则Django会抛出 `TemplateSyntaxError` 。

for

`{% for %}` 允许我们在一个序列上迭代。与Python的 `for` 语句的情形类似，循环语法是 `for X in Y` ，Y是要迭代的序列而X是在每一个特定的循环中使用的变量名称。每一次循环中，模板系统会渲染在 `{% for %}` 和 `{% endfor %}` 之间的所有内容。

例如，给定一个运动员列表 `athlete_list` 变量，我们可以使用下面的代码来显示这个列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

给标签增加一个 `reversed` 使得该列表被反向迭代：

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

可以嵌套使用 `{% for %}` 标签：

```
{% for country in countries %}
    <h1>{{ country.name }}</h1>
    <ul>
    {% for city in country.city_list %}
        <li>{{ city }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

Django不支持退出循环操作。如果我们想退出循环，可以改变正在迭代的变量，让其仅仅包含需要迭代的项目。同理，Django也不支持continue语句，我们无法让当前迭代操作跳回到循环头部。（请参看本章稍后的理念和限制小节，了解下决定这个设计的背后原因）

`{% for %}` 标签在循环中设置了一个特殊的 `forloop` 模板变量。这个变量能提供一些当前循环进展的信息：

`forloop.counter` 总是一个表示当前循环的执行次数的整数计数器。这个计数器是从1开始的，所以在第一次循环时 `forloop.counter` 将会被设置为1。例子如下：

```
{% for item in todo_list %}
    <p>{{ forloop.counter }}: {{ item }}</p>
{% endfor %}
```

`forloop.counter0` 类似于 `forloop.counter` ，但是它是从0计数的。第一次执行循环时这个变量会被设置为0。

`forloop.revcounter` 是表示循环中剩余项的整型变量。在循环初次执行时 `forloop.revcounter` 将被设置为序列中项的总数。最后一次循环执行中，这个变量将被置1。

`forloop.revcounter0` 类似于 `forloop.revcounter` ，但它以0做为结束索引。在第一次执行循环时，该变量会被置为序列的项的个数减1。在最后一次迭代时，该变量为0。

`forloop.first` 是一个布尔值。在第一次执行循环时该变量为True，在下面的情形中这个变量是很有用的。

```
{% for object in objects %}
    {% if forloop.first %}<li class="first">{% else %}<li>{% endif %}
    {{ object }}
    </li>
{% endfor %}
```

forloop.last 是一个布尔值；在最后一次执行循环时被置为True。一个常见的用法是在一系列的链接之间放置管道符（|）

```
{% for link in links %}{{ link }}{% if not forloop.last %} | {% endif %}{% endfor %}

The above template code might output something like this::

        Link1 | Link2 | Link3 | Link4
```

forloop.parentloop 是一个指向当前循环的上一级循环的 forloop 对象的引用（在嵌套循环的情况下）。例子在此：

```
{% for country in countries %}
    <table>
    {% for city in country.city_list %}
        <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
        </tr>
    {% endfor %}
    </table>
{% endfor %}
```

forloop 变量仅仅能够在循环中使用，在模板解析器碰到 {% endfor %} 标签时， forloop 就不可访问了。

Context和forloop变量

在一个 {% for %} 块中，已存在的变量会被移除，以避免 forloop 变量被覆盖。Django会把这个变量移动到 forloop.parentloop 中。通常我们不用担心这个问题，但是一旦我们在模板中定义了 forloop 这个变量（当然我们反对这样做），在 {% for %} 块中它会在 forloop.parentloop 被重新命名。

ifequal/ifnotequal

Django模板系统压根儿就没想过实现一个全功能的编程语言，所以它不允许我们在模板中执行Python的语句（还是那句话，要了解更多请参看理念和限制小节）。但是比较两个变量的值并且显示一些结果实在是个太常见的需求了，所以Django提供了 {% ifequal %} 标签供我们使用。

{% ifequal %} 标签比较两个值，当他们相同时，显示在 {% ifequal %} 和 {% endifequal %} 之中所有的值。

下面的例子比较两个模板变量 user 和 currentuser：

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

参数可以是硬编码的字符串，随便用单引号或者双引号引起来，所以下列代码都是正确的：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
    <h1>Community</h1>
{% endifequal %}
```

和 {% if %} 类似， {% ifequal %} 支持可选的 {% else%} 标签：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

只有模板变量，字符串，整数和小数可以作为 {% ifequal %} 标签的参数。这些是正确的例子：

```
{% ifequal variable 1 %}
```

```
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

其他的一些类型，例如Python的字典类型、列表类型、布尔类型，不能用在 `{% ifequal %}` 中。 下面是些错误的例子：

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

如果你需要判断变量是真还是假，请使用 `{% if %}` 来替代 `{% ifequal %}` 。

注释

象HTML和其他的语言例如python一样，Django模板系统也允许注释。 注释使用 `{# #}` ：

```
{# This is a comment #}
```

注释的内容不会在模板渲染时输出。

注释不能跨多行。这个限制是为了提高模板解析的性能。在下面这个模板中，输出结果和模板本身是 完全一样的（也就是说，注释标签并没有被解析为注释）：

```
This is a {# this is not
a comment #}
test.
```

过滤器

就象本章前面提到的一样，模板过滤器是在变量被显示前修改它的值的一个简单方法。 过滤器看起来是这样的：

```
{{ name|lower }}
```

显示的内容是变量 `{{ name }}` 被过滤器 `lower` 处理后的结果，它功能是转换文本为小写。 使用 `|` 来应用过滤器。

过滤器可以被 *串联* ,就是说一个过滤器的输出可以被输入到下一个过滤器。这里有一个常见的 惯用语法，先转义文本到HTML，再转换每行到 `<p>` 标签：

```
{{ my_text|escape|linebreaks }}
```

有些过滤器有参数。过滤器参数看起来是这样的：

```
{{ bio|truncatewords:"30" }}
```

这个将显示变量 `bio` 的前30个词。过滤器参数总是使用双引号标识。

下面是一些最重要的过滤器；附录F有完整的过滤器列表。

`addslashes` : 添加反斜杠到任何反斜杠、单引号或者双引号前面。 这在处理包含JavaScript的文本时是非常有用的。

`date` : 按指定的格式字符串参数格式化 `date` 或者 `datetime` 对象， 范例：

```
{{ pub_date|date:"F j, Y" }}
```

格式参数的定义在附录F中。

`escape` : 转义 &符号，引号，<，> 符号。 这在确保用户提交的数据是有效的XML或XHTML时是非常有用的。 具体上， `escape` 做下面这些转换:

- 转换 `&` 到 `&`;
    - 转换 < 到 `<`

- 转换 > 到 `>`
- 转换 " (双引号) 到 `&quot;`
- 转换 ' (单引号) 到 `&#39;`

`length`：返回变量的长度。你可以对列表或者字符串，或者任何知道怎么测定长度的Python对象使用这个方法（也就是说，有 `__len__()` 方法的对象）。

理念与局限

现在你已经对Django的模板语言有一些认识了，我们将指出一些特意设置的限制和为什么要这样做 背后的一些设计哲学。

相对Web应用中的其他组件，程序员们对模板系统的分歧是最大的。事实上，Python有成十上百的 开放源码的模板语言实现。每个实现都是因为开发者认为现存的模板语言不够用。（事实上，对一个 Python开发者来说，写一个自己的模板语言就象是某种"成人礼"一样！如果你还没有完成一个自己的 模板语言，好好考虑写一个，这是一个非常有趣的锻炼。）

明白了这个，你也许有兴趣知道事实上Django并不强制要求你必须使用它的模板语言。因为Django 虽然被设计成一个FULL-Stack的Web框架，它提供了开发者所必需的所有组件，而且在大多数情况 使用Django模板系统会比其他的Python模板库要 *更方便* 一点，但是并不是严格要求你必须使用 它。就象你将在后续的章节中看到的一样，你也可以非常容易的在Django中使用其他的模板语言。

虽然如此，很明显，我们对Django模板语言的工作方式有着强烈的偏爱。这个模板语言来源于World Online的开发经验和Django创造者们集体智慧的结晶。下面是关于它的一些设计哲学理念：

业务逻辑应该和表现逻辑相对分开 。我们将模板系统视为控制表现及表现相关逻辑的工具，仅此而已。模板系统不应提供超出此基本目标的功能。

出于这个原因，在 Django 模板中是不可能直接调用 Python 代码的。所有的编程工作基本上都被局限于模板标签的能力范围。当然， *是* 有可能写出自定义的模板标签来完成任意工作，但这些"超范围"的 Django 模板标签有意地不允许执行任何 Python 代码。

语法不应受到 HTML/XML 的束缚 。尽管 Django 模板系统主要用于生成 HTML，它还是被有意地设计为可生成非 HTML 格式，如纯文本。一些其它的模板语言是基于 XML 的，将所有的模板逻辑置于 XML 标签与属性之中，而 Django 有意地避开了这种限制。强制要求使用有效 XML 编写模板将会引发大量的人为错误和难以理解的错误信息，而且使用 XML 引擎解析模板也会导致令人无法容忍的模板处理开销。

假定设计师精通 HTML 编码 。模板系统的设计意图并不是为了让模板一定能够很好地显示在 Dreamweaver 这样的所见即所得编辑器中。这种限制过于苛刻，而且会使得语法不能像目前这样的完美。Django 要求模板创作人员对直接编辑 HTML 非常熟悉。

假定设计师不是 Python 程序员 。模板系统开发人员认为：网页模板通常由 *设计师* 而不是 *程序员* 编写，因而假定这些人并不掌握 Python 相关知识。

当然，系统同样也特意地提供了对那些 *由* Python 程序员进行模板制作的小型团队的支持。它提供了一种工作模式，允许通过编写原生 Python 代码进行系统语法拓展。（详见第十章）

Django架构的目标并不是要发明一种编程语言 。而是恰到好处地提供如分支和循环这一类编程式功能，这是进行与表现相关判断的基础。

采用这些设计理念的结果是导致 Django 模板语言有以下几点限制：

- *模板中不能设置变量和改变变量的值* 。可以通过编写自定义模板标签做到这一点（参见第十章），但正宗的 Django 模板标签做不到。
- *模板中不能调用任何的Python代码* 。不存在转入 Python 模式或使用原生 Python 数据结构的方法。和前面一样，可以编写自定义模板标签来实现这个目标，但正统的 Django 模板标签做不到。

在视图中使用模板

在学习了模板系统的基础之后，现在让我们使用相关知识来创建视图。重新打开我们在前一章在 `mysite.views` 中创建的 `current_datetime` 视图。以下是其内容：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

让我们用 Django 模板系统来修改该视图。第一步，你可能已经想到了要做下面这样的修改：

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

没错，它确实使用了模板系统，但是并没有解决我们在本章开头所指出的问题。也就是说，模板依然内嵌在 Python 代码之中。让我们将模板置于一个 *单独的文件* 中，并且让视图加载该文件来解决此问题。

你可能首先考虑把模板保存在文件系统的某个位置并用 Python 内建的文件操作函数来读取文件内容。假设文件保存在 `/home/djangouser/templates/mytemplate.html` 中的话，代码就会像下面这样:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    # Simple way of using templates from the filesystem.
    # This doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

然而，基于以下几个原因，该方法还算不上简洁：

- 它没有对文件丢失的情况做出处理。如果文件 `mytemplate.html` 不存在或者不可读， `open()` 函数调用将会引发 `IOError` 异常。
- 这里对模板文件的位置进行了硬编码。如果你在每个视图函数都用该技术，就要不断复制这些模板的位置。更不用说还要带来大量的输入工作！
- 它包含了大量令人生厌的重复代码。与其在每次加载模板时都调用 `open()` 、 `fp.read()` 和 `fp.close()` ，还不如做出更佳选择。

要解决此问题，我们将使用 *模板加载* 和 *模板目录* ，这是我们在接下来的章节中要讨论的两个话题。

模板加载

为了减少模板加载调用过程及模板本身的冗余代码，Django 提供了一种使用方便且功能强大的 API ，用于从磁盘中加载模板，

要使用此模板加载API，首先你必须将模板的保存位置告诉框架。该项工作在 *设置文件* 中完成。

Django 设置文件是存放 Django 实例（也就是 Django 项目）配置的地方。它是一个简单的 Python 模块，其中包含了一些模块级变量，每个都是一项设置。

第二章中执行 `django-admin.py startproject mysite` 命令时，它为你创建了一个的缺省配置文件，并恰如其分地将其名为 `settings.py` 。查看一下该文件内容。其中包含如下变量（但并不一定是这个顺序）：

```
DEBUG = True
TIME_ZONE = 'America/Chicago'
USE_I18N = True
ROOT_URLCONF = 'mysite.urls'
```

这里无需更多诠释；设置项与值均为简单的 Python 变量。同时由于配置文件只不过是纯 Python 模块，你可以完成一些动态工作，比如在设置某变量之前检查另一变量的值。（这也意味着你必须避免配置文件出现 Python 语法错误。）

我们将在附录 E 中详述配置文件，目前而言，仅需关注 `TEMPLATE_DIRS` 设置。该设置告诉 Django 的模板加载机制在哪里查找模板。缺省情况下，该设置的值是一个空的元组。选择一个目录用于存放模板并将其添加到 `TEMPLATE_DIRS` 中：

```
TEMPLATE_DIRS = (
    '/home/django/mysite/templates',
)
```

下面是一些注意事项：

你可以任意指定想要的目录，只要运行 Web 服务器的用户账号可以读取该目录的子目录和模板文件。如果实在想不出合适的位置来放置模板，我们建议在 Django 项目中创建一个 `templates` 目录（也就是说，如果你一直都按本书的范例操作的话，在第二章创建的 `mysite` 目录中）。

不要忘记模板目录字符串尾部的逗号！Python 要求单元素元组中必须使用逗号，以此消除与圆括号表达式之间的歧义。这是新手常犯的错误。

想避免此错误的话，你可以将列表而不是元组用作 `TEMPLATE_DIRS` ，因为单元素列表并不强制要求以逗号收尾：

```
TEMPLATE_DIRS = [
    '/home/django/mysite/templates'
]
```

从语义上看，元组比列表略显合适（元组在创建之后就不能修改，而配置被读取以后就不应该有任何修改）。因此，我们推荐对 `TEMPLATE_DIRS` 设置使用元组。

如果使用的是 Windows 平台，请包含驱动器符号并使用Unix风格的斜杠（/）而不是反斜杠（\）,就像下面这样：

```
TEMPLATE_DIRS = (
    'C:/www/django/templates',
)
```

最省事的方式是使用绝对路径（即从文件系统根目录开始的目录路径）。如果想要更灵活一点并减少一些负面干扰，可利用 Django 配置文件就是 Python 代码这一点来动态构建 `TEMPLATE_DIRS` 的内容，如：

```
import os.path

TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\','/'),
)
```

这个例子使用了神奇的 Python 内部变量 `__file__` ，该变量被自动设置为代码所在的 Python 模块文件名。

完成 `TEMPLATE_DIRS` 设置后，下一步就是修改视图代码，让它使用 Django 模板加载功能而不是对模板路径硬编码。返回 `current_datetime` 视图，进行如下修改：

```
from django.template.loader import get_template
from django.template import Context
```

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

此范例中，我们使用了函数 `django.template.loader.get_template()`，而不是手动从文件系统加载模板。该 `get_template()` 函数以模板名称为参数，在文件系统中找出模块的位置，打开文件并返回一个编译好的 `Template` 对象。

如果 `get_template()` 找不到给定名称的模板，将会引发一个 `TemplateDoesNotExist` 异常。要了解究竟会发生什么，让我们按照第三章内容，在 Django 项目目录中运行 `python manage.py runserver` 命令，再次启动Django开发服务器。。然后，用浏览器访问页面（如：`http://127.0.0.1:8000/time/`）激活 `current_datetime` 视图。假如 `DEBUG` 设置为 `True` 而又未创建 `current_datetime.html` 模板，你将会看到 `TemplateDoesNotExist` 错误信息页面。

图 4-1: 无法找到模板时的出错页面

该页面与我们在第三章解释过的错误页面相似，只不过多了一块调试信息区：模板加载器事后检查区。该区域显示 Django 要加载哪个模板、每次尝试出错的原因（如：文件不存在等）。在调试模板加载错误时，这些信息的价值是不可估量的。

正如你从图 4-1 中的错误信息中所看到，Django 尝试通过组合 `TEMPLATE_DIRS` 设置以及传递给 `get_template()` 的模板名称来查找模板。因此如果 `TEMPLATE_DIRS` 为 `'/home/django/templates'`，Django 将会 查找 `'/home/django/templates/current_datetime.html'`。如果 `TEMPLATE_DIRS` 包含多个目录，它将会查找每个目录直至找到模板或找遍所有目录。

接下来，在模板目录中创建包括以下模板代码 `current_datetime.html` 文件：

```
<html><body>It is now {{ current_date }}.</body></html>
```

在网页浏览器中刷新该页，你将会看到完整解析后的页面。

render_to_response()

由于加载模板、填充 `context`、将经解析的模板结果返回为 `HttpResponse` 对象这一系列操作实在太常用了，Django 提供了一条仅用一行代码就完成所有这些工作的捷径。该捷径就是位于 `django.shortcuts` 模块中名为 `render_to_response()` 的函数。大多数时候，你将使用 `render_to_response()`，而不是手动加载模板、创建 `Context` 和 `HttpResponse` 对象。

下面就是使用 `render_to_response()` 重新编写过的 `current_datetime` 范例。

```
from django.shortcuts import render_to_response
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

大变样了！让我们逐句看看代码发生的变化：

- 我们不再需要导入 `get_template`、`Template`、`Context` 和 `HttpResponse`。相反，我们导入 `django.shortcuts.render_to_response`。`import datetime` 继续保留.
- 在 `current_datetime` 函数中，我们仍然进行 `now` 计算，但模板加载、上下文创建、模板解析和 `HttpResponse` 创建工作均在对 `render_to_response()` 的调用中完成了。由于 `render_to_response()` 返回 `HttpResponse` 对象，因此我们仅需在视图中 `return` 该值。

`render_to_response()` 的第一个参数必须是要使用的模板名称。如果要给定第二个参数，那么该参数必须是为该模板创建 `Context` 时所使用的字典。如果不提供第二个参数， `render_to_response()` 使用一个空字典。

locals() 技巧

思考一下我们对 `current_datetime` 的最后一次赋值:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

很多时候，就像在这个范例中那样，你发现自己一直在计算某个变量，保存结果到变量中（比如：前面代码中的 `now` ），然后将这些变量发送给模板。特别懒的程序员可能注意到给这些临时变量 *和* 模板变量命名显得有点多余。不但多余，而且还要进行额外的键盘输入。

如果你是个喜欢偷懒的程序员并想让代码看起来更加简明，可以利用 Python 的内建函数 `locals()` 。它返回的字典对所有局部变量的名称与值进行映射。因此，前面的视图可以重写成下面这个样子：

```
def current_datetime(request):
    current_date = datetime.datetime.now()
    return render_to_response('current_datetime.html', locals())
```

在此，我们没有像之前那样手工指定 context 字典，而是传入了 `locals()` 的值，它囊括了函数执行到该时间点时所定义的一切变量。因此，我们将 `now` 变量重命名为 `current_date` ，因为那才是模板所预期的变量名称。在本例中，`locals()` 并没有带来多 *大* 的改进，但是如果有多个模板变量要界定而你又想偷懒，这种技术可以减少一些键盘输入。

使用 `locals()` 时要注意是它将包括 *所有* 的局部变量，组成它的变量可能比你想让模板访问的要多。在前例中，`locals()` 还包含了 `request` 。对此如何取舍取决你的应用程序。

最后要考虑的是在你调用 `locals()` 时，Python 必须得动态创建字典，因此它会带来一点额外的开销。如果手动指定 context 字典，则可以避免这种开销。

get_template()中使用子目录

把所有的模板都存放在一个目录下可能会让事情变得难以掌控。你可能会考虑把模板存放在你模板目录的子目录中，这非常好。事实上，我们推荐这样做；一些Django的高级特性（例如将在第九章讲到的通用视图系统）的缺省约定就是期望使用这种模板布局。

把模板存放于模板目录的子目录中是件很轻松的事情。只需在调用 `get_template()` 时，把子目录名和一条斜杠添加到模板名称之前，如：

```
t = get_template('dateapp/current_datetime.html')
```

由于 `render_to_response()` 只是对 `get_template()` 的简单封装， 你可以对 `render_to_response()` 的第一个参数做相同处理。

对子目录树的深度没有限制，你想要多少层都可以。

注意

Windows用户必须使用斜杠而不是反斜杠。 `get_template()` 假定的是 Unix 风格的文件名符号约定。

include 模板标签

在讲解了模板加载机制之后，我们再介绍一个利用该机制的内建模板标签： `{% include %}` 。该标签允许在（模板中）包含其它的模板的内容。标签的参数是所要包含的模板名称，可以是一个变量，也可以是用单/双引号硬编码的字符串。每当在多个模板中出现相同的代码时，就应该考虑是否要使用 `{% include %}` 来减少重复。

下面这两个例子都包含了 `nav.html` 模板。两个例子的作用完全相同，只不过是为了说明单、双引号都可以通用。

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

下面的例子包含了 `includes/nav.html` 模板的内容:

```
{% include 'includes/nav.html' %}
```

下面的例子包含了以变量 `template_name` 的值为名称的模板内容：

```
{% include template_name %}
```

和在 `get_template()` 中一样， 对模板的文件名进行判断时会在所调取的模板名称之前加上来自 `TEMPLATE_DIRS` 的模板目录。

所包含的模板执行时的 context 和包含它们的模板是一样的。

如果未找到给定名称的模板文件，Django 会从以下两件事情中择一而为之：

- 如果 `DEBUG` 设置为 `True` ，你将会在 Django 错误信息页面看到 `TemplateDoesNotExist` 异常。
- 如果 `DEBUG` 设置为 `False` ，该标签不会引发错误信息，在标签位置不显示任何东西。

模板继承

到目前为止，我们的模板范例都只是些零星的 HTML 片段，但在实际应用中，你将用 Django 模板系统来创建整个 HTML 页面。这就带来一个常见的 Web 开发问题：在整个网站中，如何减少共用页面区域（比如站点导航）所引起的重复和冗余代码？

解决该问题的传统做法是使用 *服务器端的 includes* ，你可以在 HTML 页面中使用该指令将一个网页嵌入到另一个中。事实上， Django 通过刚才讲述的 `{% include %}` 支持了这种方法。但是用 Django 解决此类问题的首选方法是使用更加简洁的策略—— *模板继承* 。

本质上来说，模板继承就是先构造一个基础框架模板，而后在其子模板中对它所包含站点公用部分和定义块进行重载。

让我们通过修改 `current_datetime.html` 文件，为 `current_datetime` 创建一个更加完整的模板来体会一下这种做法：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

这看起来很棒，但如果我们要为第三章的 `hours_ahead` 视图创建另一个模板会发生什么事情呢？如果我们再次创建一个漂亮、有效且完整的 HTML 模板，我们可能会创建出下面这样的东西：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
```

```
</body>
</html>
```

很明显，我们刚才重复了大量的 HTML 代码。想象一下，如果有一个更典型的网站，它有导航条、样式表，可能还有一些 JavaScript 代码，事情必将以向每个模板填充各种冗余的 HTML 而告终。

解决这个问题的服务器端 include 方案是找出两个模板中的共同部分，将其保存为不同的模板片段，然后在每个模板中进行 include。也许你会把模板头部的一些代码保存为 `header.html` 文件：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

你可能会把底部保存到文件 `footer.html`：

```
    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

对基于 include 的策略，头部和底部的包含很简单。麻烦的是中间部分。在此范例中，每个页面都有一个 `<h1>My helpful timestamp site</h1>` 标题，但是这个标题不能放在 `header.html` 中，因为每个页面的 `<title>` 是不同的。如果我们将 `<h1>` 包含在头部，我们就不得不包含 `<title>`，但这样又不允许在每个页面对它进行定制。何去何从呢？

Django 的模板继承系统解决了这些问题。你可以将其视为服务器端 include 的逆向思维版本。你可以对那些 *不同* 的代码段进行定义，而不是 *共同* 代码段。

第一步是定义 *基础模板*， 该框架之后将由 *子模板* 所继承。以下是我们目前所讲述范例的基础模板：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

这个叫做 `base.html` 的模板定义了一个简单的 HTML 框架文档，我们将在本站点的所有页面中使用。子模板的作用就是重载、添加或保留那些块的内容。（如果一直按我们的范例做话，请将此文件保存到模板目录。）

我们使用一个以前没有见过的模板标签：`{% block %}`。 所有的 `{% block %}` 标签告诉模板引擎，子模板可以重载这些部分。

现在我们已经有了一个基本模板，我们可以修改 `current_datetime.html` 模板来 使用它：

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

再为 `hours_ahead` 视图创建一个模板，看起来是这样的：

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}
```

```
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock %}
```

看起来很漂亮是不是？每个模板只包含对自己而言 *独一无二* 的代码。无需多余的部分。如果想进行站点级的设计修改，仅需修改 `base.html` ，所有其它模板会立即反映出所作修改。

以下是其工作方式。在加载 `current_datetime.html` 模板时，模板引擎发现了 `{% extends %}` 标签， 注意到该模板是一个子模板。模板引擎立即装载其父模板，即本例中的 `base.html` 。

此时，模板引擎注意到 `base.html` 中的三个 `{% block %}` 标签，并用子模板的内容替换这些 block 。因此，引擎将会使用我们在 `{ block title %}` 中定义的标题，对 `{% block content %}` 也是如此。

注意由于子模板并没有定义 `footer` 块，模板系统将使用在父模板中定义的值。父模板 `{% block %}` 标签中的内容总是被当作一条退路。

继承并不改变 context 的工作方式，而且你可以按照需要使用多层继承。使用继承的一种常见方式是下面的三层法：

1. 创建 `base.html` 模板，在其中定义站点的主要外观感受。这些都是不常修改甚至从不修改的部分。
2. 为网站的每个区域创建 `base_SECTION.html` 模板(例如, `base_photos.html` 和 `base_forum.html`)。这些模板对 `base.html` 进行拓展，并包含区域特定的风格与设计。
3. 为每种类型的页面创建独立的模板，例如论坛页面或者图片库。这些模板拓展相应的区域模板。

这个方法可最大限度地重用代码，并使得向公共区域（如区域级的导航）添加内容成为一件轻松的工作。

以下是使用模板继承的一些诀窍：

- 如果在模板中使用 `{% extends %}` ，必须保证其为模板中的第一个模板标记。否则，模板继承将不起作用。
- 一般来说，基础模板中的 `{% block %}` 标签越多越好。记住，子模板不必定义父模板中所有的代码块，因此你可以用合理的缺省值对一些代码块进行填充，然后只对子模板所需的代码块进行（重）定义。俗话说，钩子越多越好。
- 如果发觉自己在多个模板之间拷贝代码，你应该考虑将该代码段放置到父模板的某个 `{% block %}` 中。
- 如果需要获得父模板中代码块的内容，可以使用 `{{ block.super }}` 变量。如果只想在上级代码块基础上添加内容，而不是全部重载，该变量就显得非常有用了。
- 不可同一个模板中定义多个同名的 `{% block %}` 。存在这样的限制是因为block 标签的工作方式是双向的。也就是说，block 标签不仅挖了一个要填的坑，也定义了在 *父* 模板中这个坑所填充的内容。如果模板中出现了两个相同名称的 `{% block %}` 标签，父模板将无从得知要使用哪个块的内容。
- `{% extends %}` 对所传入模板名称使用的加载方法和 `get_template()` 相同。也就是说，会将模板名称被添加到 `TEMPLATE_DIRS` 设置之后。
- 多数情况下， `{% extends %}` 的参数应该是字符串，但是如果直到运行时方能确定父模板名，这个参数也可以是个变量。这使得你能够实现一些很酷的动态功能。

接下来？

时下大多数网站都是 *数据库驱动* 的：网站的内容都是存储在关系型数据库中。这使得数据和逻辑能够彻底地分开（视图和模板也以同样方式对逻辑和显示进行了分隔。）

在下一章里将讲述 Django 提供的数据库交互工具。

第五章：与数据库的交互：数据建模

在第三章，我们讲述了用 Django 建造网站的基本途径：建立视图和 URLConf。正如我们所阐述的，视图负责处理 一*些任意逻辑*，然后返回响应结果。在范例中，我们的任意逻辑就是计算当前的日期和时间。

在当代 Web 应用中，任意逻辑经常牵涉到与数据库的交互。 *数据库驱动网站* 在后台连接数据库服务器，从中取出一些数据，然后在 Web 页面用漂亮的格式展示这些数据。或者，站点也提供让访问者自行填充数据库的功能。

许多复杂的网站都提供了以上两个功能的某种结合。例如 Amazon.com 就是一个数据库驱动站点的良好范例。本质上，每个产品页都是从数据库中取出的数据被格式化为 HTML，而当你发表客户评论时，该评论被插入评论数据库中。

由于先天具备 Python 简单而强大的数据库查询执行方法，Django 非常适合开发数据库驱动网站。本章深入介绍了该功能：Django 数据库层。

（注意：尽管对 Django 数据库层的使用中并不特别强调，我们还是强烈建议掌握一些数据库和 SQL 原理。对这些概念的介绍超越了本书的范围，但就算你是数据库方面的菜鸟，我们也建议你继续阅读。你也许能够跟上进度，并在上下文学习过程中掌握一些概念。）

在视图中进行数据库查询的笨方法

正如第三章详细介绍的那个在视图中输出 HTML 的笨方法（通过在视图里对文本直接硬编码HTML），在视图中也有笨方法可以从数据库中获取数据。很简单：用现有的任何 Python 类库执行一条 SQL 查询并对结果进行一些处理。

在本例的视图中，我们使用了 `MySQLdb` 类库（可以从 http://www.djangoproject.com/r/python-mysql/ 获得）来连接 MySQL 数据库，取回一些记录，将它们提供给模板以显示一个网页：

```
from django.shortcuts import render_to_response
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render_to_response('book_list.html', {'names': names})
```

这个方法可用，但很快一些问题将出现在你面前：

- 我们将数据库连接参数硬行编码于代码之中。理想情况下，这些参数应当保存在 Django 配置中。
- 我们不得不重复同样的代码：创建数据库连接、创建数据库游标、执行某个语句、然后关闭数据库。理想情况下，我们所需要应该只是指定所需的结果。
- 它把我们栓死在 MySQL 之上。如果过段时间，我们要从 MySQL 换到 PostgreSQL，就不得不使用不同的数据库适配器（例如 `psycopg` 而不是 `MySQLdb`），改变连接参数，根据 SQL 语句的类型可能还要修改SQL。理想情况下，应对所使用的数据库服务器进行抽象，这样一来只在一处修改即可变换数据库服务器。

正如你所期待的，Django数据库层正是致力于解决这些问题。以下提前揭示了如何使用 Django 数据库 API 重写之前那个视图。

```
from django.shortcuts import render_to_response
from mysite.books.models import Book

def book_list(request):
    books = Book.objects.order_by('name')
    return render_to_response('book_list.html', {'books': books})
```

我们将在本章稍后的地方解释这段代码。目前而言，仅需对它有个大致的认识。

MTV 开发模式

在钻研更多代码之前，让我们先花点时间考虑下 Django 数据驱动 Web 应用的总体设计。

我们在前面章节提到过，Django 的设计鼓励松耦合及对应用程序中不同部分的严格分割。遵循这个理念的话，要想修改应用的某部分而不影响其它部分就比较容易了。在视图函数中，我们已经讨论了通过模板系统把业务逻辑和表现逻辑分隔开的重要性。在数据库层中，我们对数据访问逻辑也应用了同样的理念。

把数据存取逻辑、业务逻辑和表现逻辑组合在一起的概念有时被称为软件架构的 *Model-View-Controller* (MVC)模式。在这个模式中， Model 代表数据存取层，View 代表的是系统中选择显示什么和怎么显示的部分，Controller 指的是系统中根据用户输入并视需要访问模型，以决定使用哪个视图的那部分。

为什么使用缩写？

像 MVC 这样的明确定义模式的主要用于改善开发人员之间的沟通。与其告诉同事："让我们对数据存取进行抽象，用单独一层负责数据显示，然后在中间放置一层来进行控制"，还不如利用通用的词汇告诉他们："让我们在这里使用 MVC 模式吧"。

Django 紧紧地遵循这种 MVC 模式，可以称得上是一种 MVC 框架。以下是 Django 中 M、V 和 C 各自的含义：

- *M*，数据存取部分，由django数据库层处理，本章要讲述的内容。
- *V*，选择显示哪些数据要及怎样显示的部分，由视图和模板处理。
- *C*，根据用户输入委派视图的部分，由 Django 框架通过按照 URLconf 设置，对给定 URL 调用合适的 python 函数来自行处理。

由于 C 由框架自行处理，而 Django 里更关注的是模型（Model）、模板(Template)和视图（Views），Django 也被称为 *MTV 框架* 。在 MTV 开发模式中：

- *M* 代表模型（Model），即数据存取层。该层处理与数据相关的所有事务：如何存取、如何确认有效性、包含哪些行为以及数据之间的关系等。
- *T* 代表模板(Template)，即表现层。该层处理与表现相关的决定：如何在页面或其他类型文档中进行显示。

V代表View，业务逻辑层。这一层包含访问模型的逻辑和按照模板显示。你可以认为它是模型和模板的桥梁。

如果你熟悉其它的 MVC Web开发框架，比方说 Ruby on Rails，你可能会认为 Django 视图是控制器，而 Django 模板是视图。很不幸，这是对 MVC 不同诠释所引起的错误认识。在 Django 对 MVC 的诠释中，视图用来描述要展现给用户的数据；不是数据看起来 *怎么样* ,而是要呈现 *哪些* 数据。相比之下，Ruby on Rails 及一些同类框架提倡控制器负责决定向用户展现哪些数据，而视图则仅决定 *如何* 展现数据，而不是展现 *哪些* 数据。

两种诠释中没有哪个更加正确一些。重要的是要理解底层概念。

数据库配置

记住这些理念之后，让我们来开始 Django 数据库层的探索。首先，我们需要搞定一些初始化设置：我们必须告诉 Django 要用哪个数据库服务器及如何连接上它。

我们将假定你已经完成了数据库服务器的安装和激活，并且已经在其中创建了数据库（例如，用 `CREATE DATABASE` 语句）。SQLite 数据库有点特别，用它的话不需要创建数据库，因为 SQLite 使用文件系统中的单个文件来保存数据。

象前面章节提到的 `TEMPLATE_DIRS` 一样，数据库配置也是在Django的配置文件里，缺省 是 `settings.py` 。编辑打开这个文件并查找数据库配置：

```
DATABASE_ENGINE = ''
DATABASE_NAME = ''
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

配置纲要如下。

DATABASE_ENGINE 告诉Django使用哪个数据库引擎。如果你在 Django 中使用数据库， DATABASE_ENGINE 必须是 Table 5-1 中所列出的值。

表 5-1. 数据库引擎设置

设置	数据库	适配器
postgresql	PostgreSQL	psycopg 版本 1.x, http://www.djangoproject.com/r/python-pgsql/1/.
postgresql_psycopg2	PostgreSQL	psycopg 版本 2.x, http://www.djangoproject.com/r/python-pgsql/.
mysql	MySQL	MySQLdb , http://www.djangoproject.com/r/python-mysql/.
sqlite3	SQLite	Python 2.5+ 内建。 其他, pysqlite , http://www.djangoproject.com/r/python-sqlite/.
ado_mssql	Microsoft SQL Server	adodbapi 版本 2.0.1+, http://www.djangoproject.com/r/python-ado/.
oracle	Oracle	cx_Oracle , http://www.djangoproject.com/r/python-oracle/.

要注意的是无论选择使用哪个数据库服务器，都必须下载和安装对应的数据库适配器。访问表 5-1 中“所需适配器”一栏中的链接，可通过互联网免费获取这些适配器。

DATABASE_NAME 将数据库名称告知 Django 。如果使用 SQLite，请对数据库文件指定完整的文件系统路径。（例如 '/home/django/mydata.db' ）。

DATABASE_USER 告诉 Django 用哪个用户连接数据库。如果用SQLite，空白即可。

DATABASE_PASSWORD 告诉Django连接用户的密码。SQLite 用空密码即可。

DATABASE_HOST 告诉 Django 连接哪一台主机的数据库服务器。如果数据库与 Django 安装于同一台计算机（即本机），可将此项保留空白。使用 SQLite ，也可保留空白。

此处的 MySQL 是一个特例。如果使用的是 MySQL 且该项设置值由斜杠（ '/' ）开头，MySQL 将通过 Unix socket 来连接指定的套接字，例如：

```
DATABASE_HOST = '/var/run/mysql'
```

如果用 MySQL 而该项设置的值 *不是* 以正斜线开始的，系统将假定该项值是主机名。

DATABASE_PORT 告诉 Django 连接数据库时使用哪个端口。如果用SQLite，空白即可。其他情况下，如果将该项设置保留空白，底层数据库适配器将会连接所给定数据库服务器的缺省端口。在多数情况下，使用缺省端口就可以了，因此你可以将该项设置保留空白。

输入完设置后，测试一下配置情况。首先，转到在第二章创建的 mysite 项目目录，运行 python manage.py shell 命令。

你会看到该命令启动了一个 Python 交互界面。运行命令 python manage.py shell 启动的交互界面和标准的 python 交互界面有很大的区别。看起来都是基本的python外壳（shell），但是前者告诉Django使用哪个配置文件启动。这对数据库操作来说很关键：Django需要知道使用哪个配置文件来获得数据库连接信息。

python manage.py shell 假定你的配置文件就在和 manage.py 一样的目录中。 以后将会讲到使用其他的方式来告诉Django使用其他的配置文件。

输入下面这些命令来测试你的数据库配置：

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

如果没有显示什么错误信息，那么你的数据库配置是正确的。否则，你就得 查看错误信息来纠正错误。表 5-2 是一些常见错误。

表 5-2. 数据库配置错误信息

错误信息	解决方案
You havent set the DATABASE_ENGINE setting yet.	设置正确的 DATABASE_ENGINE 配置
Environment variable DJANGO_SETTINGS_MODULE is undefined.	运行命令行 python manage.py shell 而不是 python .
Error loading _____ module: No module named _____.	你没有安装相关的数据库适配器 (例如, psycopg 或 MySQLdb).
_____ isnt an available database backend.	设置正确的 DATABASE_ENGINE 配置 也许是拼写错误？
database _____ does not exist	设置 DATABASE_NAME 配置到一个已有的数据库， 或者使用 CREATE DATABASE 语句创建数据库。
role _____ does not exist	修改 DATABASE_USER 配置到一个有效用户
could not connect to server	确认 DATABASE_HOST 和 DATABASE_PORT 设置是正确的，并 确认服务器是在运行的。

你的第一个应用程序

你现在已经确认数据库连接正常工作了，让我们来创建一个 *Django app* ，开始编码模型和视图。这些文件放置在同一个包中并且形成为一个完整的Django应用程序。

在这里要先解释一些术语，初学者可能会混淆它们。在第二章我们已经创建了 *project* ,

System Message: WARNING/2 (<string>, line 402)

Block quote ends without a blank line; unexpected unindent.

那么 *project* 和 *app* 之间到底有什么不同呢？ 它们的区别就是一个是配置另一个是代码：

一个project包含很多个Django app以及对它们的配置。

技术上，project的作用是提供配置文件，比方说哪里定义数据库连接信息, 安装的app列表， TEMPLATE_DIRS ，等等。

一个app是一套Django功能的集合，通常包括模型和视图，按Python的包结构的方式存在。

例如，Django本身内建有一些app，如注释系统和自动管理界面。值得关注的是，这些app很容易移植到其他project和被多个project重用。

如果你只是建造一个简单的web站点，那么可能你只需要一个app就可以了。如果是复杂的象 电子商务之类的Web站点，你可能需要把这些功能划分成不同的app，以便以后重用。

确实，你还可以不用创建app，例如以前写的视图，只是简单的放在 views.py ，不需要app。

当然，系统对app有一个约定：如果你使用了Django的数据库层（模型），你 必须创建一个django app。模型必须在这个app中存在。因此，为了开始建造 我们的模型，我们必须创建一个新的app。

转到 mysite 项目目录，执行下面的命令来创建一个新app叫做books：

```
python manage.py startapp books
```

这段命令行代码运行时并不输出任何结果，但是它在 mysite 目录下创建了一个 books 的文件夹。我们现在来看看这个文件夹里的内容

```
books/
    __init__.py
    models.py
    views.py
```

这些文件里面就包含了这个app的模型和视图。

看一下 `models.py` 和 `views.py` 文件。它们都是空的，除了 `models.py` 里有一个 import。

在Python代码里定义模型

我们早些时候谈到。MTV里的M代表模型。Django模型是用Python代码形式表述的数据在数据库中的定义。对数据层来说它等同于 `CREATE TABLE` 语句，只不过执行的是Python代码而不是SQL，而且还包含了比数据库字段定义更多的含义。Django用模型在后台执行SQL代码并把结果用Python的数据结构来描述，这样你可以很方便的使用这些数据。Django还用模型来描述SQL不能处理的高级概念。

如果你对数据库很熟悉，你可能马上就会想到，用Python *和* SQL来定义数据模型是不是有点多余？ Django这样做是有下面几个原因的：

自省（运行时自动识别数据库）会导致过载和有数据完整性问题。为了提供方便的数据访问API，Django需要以 *某种方式* 知道数据库层内部信息，有两种实现方式。第一种方式是用Python明确的定义数据模型，第二种方式是通过运行时扫描数据库来自动侦测识别数据模型。

第二种方式看起来更清晰，因为数据表信息只存放在一个地方-数据库里，但是会带来一些问题。首先，运行时扫描数据库会带来严重的系统过载。如果每个请求都要扫描数据库的表结构，或者即便是服务启动时做一次都是会带来不能接受的系统过载。（Django尽力避免过载，而且成功做到了这一点）其次，有些数据库，例如老版本的MySQL，没有提供足够的元数据来完整地重构数据表。

编写Python代码是非常有趣的，保持用Python的方式思考会避免你的大脑在不同领域来回切换。这可以帮助你提高生产率。不得不去重复写SQL，再写Python代码，再写SQL，…，会让你头都要裂了。

把数据模型用代码的方式表述来让你可以容易对它们进行版本控制。这样，你可以很容易了解数据层的变动情况。

SQL只能描述特定类型的数据字段。例如，大多数数据库都没有数据字段类型描述Email地址、URL。而用Django的模型可以做到这一点。好处就是高级的数据类型带来高生产力和更好的代码重用。

> SQL还有在不同数据库平台的兼容性问题。你必须为不同的数据库编写不同的SQL脚本， 而Python的模块就不会有这个问题。

当然，这个方法也有一个缺点，就是Python代码和数据库表的同步问题。如果你修改了一个Django模型， 你要自己做工作来保证数据库和模型同步。我们将在稍后讲解解决这个问题的几种策略。

最后,我们要提醒你Django提供了实用工具来从现有的数据库表中自动扫描生成模型。 这对已有的数据库来说是非常快捷有用的。

你的第一个模型

在本章和后续章节里，我们将集中到一个基本的 书籍/作者/出版商 数据层上。我们这样做是因为 这是一个众所周知的例子，很多SQL有关的书籍也常用这个举例。你现在看的这本书也是由作者 创作再由出版商出版的哦！

我们来假定下面的这些概念、字段和关系：

trailcoojy@hotmail.com

- 出版商有名称，地址，所在城市、省，国家，网站。
- 书籍有书名和出版日期。它有一个或多个作者（和作者是多对多的关联关系[many-to-many]）， 只有一个出版商（和出版商是一对多的关联关系[one-to-many]，也被称作外键[foreign key]）

在Django中使用该数据库布局的第一步是将其表述为Python代码。在通过“startapp”命令创建的“models.py”文件中，输入如下代码：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
```

```
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

让我们来快速讲解一下这些代码的含义。首先要注意的事是每个数据模型都是 `django.db.models.Model` 的子类。它的父类 `Model` 包含了所有和数据库 打交道的方法，并提供了一个简洁漂亮的定义语法。不管你相信还是不相信， 这就是我们用Django写的数据基本存取功能的全部代码。

每个模型通常对应于单个数据库表,每个属性模型　　通常对应于数据库表的列。属性名称对应　　列名称和类型的字段(如。”、“CharField”)对应于数据库列　　类型(如。”、“varchar””)。例如,“出版商“模型相当于以下　　表(假设PostgreSQL‘创建表的语法):

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

事实上，正如过一会儿我们所要展示的，Django 可以自动生成这些 `CREATE TABLE` 语句。

“每个数据库表对应一个类”这条规则的例外情况是多对多关系。在我们的范例模型中， `Book` 有一个 *多对多字段* 叫做 `authors` 。 该字段表明一本书籍有一个或多个作者，但 `Book` 数据库表却并没有 `authors` 字段。相反，Django创建了一个额外的表（多对多连接表）来处理书籍和作者之间的映射关系。

请查看附录 B 了解所有的字段类型和模型语法选项。

最后需要注意的是：我们并没有显式地为这些模型定义任何主键。除非你指定，否则 Django 会自动为每个模型创建一个叫做 `id` 的主键。每个 Django 模型必须要有一个单列主键。

模型安装

完成这些代码之后，现在让我们来在数据库中创建这些表。要完成该项工作，第一步是在 Django 项目中 *激活* 这些模型。将 `books` app 添加到配置文件的已 installed apps 列表中即可完成此步骤。

再次编辑 `settings.py` 文件， 找到 `INSTALLED_APPS` 设置。 `INSTALLED_APPS` 告诉 Django 项目哪些 app 处于激活状态。缺省情况下如下所示：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

把这四个设置前面加#临时注释起来。（它们是一些缺省的公用设置，现在先不管 它们，以后再来讨论）同样的，修改缺省的 `MIDDLEWARE_CLASSES` 和 `TEMPLATE_CONTEXT_PROCESSORS` 设置，都注释起来。 然后添加 `'mysite.books'` 到 `INSTALLED_APPS` 列表，现在看起来是这样：

```
MIDDLEWARE_CLASSES = (
#    'django.middleware.common.CommonMiddleware',
#    'django.contrib.sessions.middleware.SessionMiddleware',
#    'django.contrib.auth.middleware.AuthenticationMiddleware',
#    'django.middleware.doc.XViewMiddleware',
)

TEMPLATE_CONTEXT_PROCESSORS = ()
#...

INSTALLED_APPS = (
    #'django.contrib.auth',
    #'django.contrib.contenttypes',
    #'django.contrib.sessions',
    #'django.contrib.sites',
    'mysite.books',
)
```

（尽管这是单个tuple元素，我们也不要忘了结尾的逗号[,]。 另外，本书的作者喜欢在 *每一个* tuple元素后面加一个逗号，不管它是不是只有一个元素。这是为了避免忘了加逗号)

`'mysite.books'` 标识 `books` app。 `INSTALLED_APPS` 中的每个app都用 Python的路径描述，包的路径，用小数点(.)区分。

现在我们可以创建数据库表了。首先，用下面的命令对校验模型的有效性：

```
python manage.py validate
```

`validate` 命令检查你的模型的语法和逻辑是否正确。如果一切正常，你会看到 `0 errors found` 消息。如果有问题，它会给出非常有用的错误信息来帮助你 修正你的模型。

一旦你觉得你的模型可能有问题，运行 `python manage.py validate` 。 它可以帮助你捕获一些常见的模型定义错误。

模型确认没问题了，运行下面的命令来生成 `CREATE TABLE` 语句：

```
python manage.py sqlall books
```

在这个命令行中， `books` 是app的名称。和你运行 `manage.py startapp` 中的一样。 运行命令的结果是这样的：

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL
);
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "salutation" varchar(10) NOT NULL,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL,
    "headshot" varchar(100) NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE INDEX books_book_publisher_id ON "books_book" ("publisher_id");
```

```
COMMIT;
```

注意：

- 自动生成的表名是app名称（ `books` ）和模型的小写名称 （ `publisher` , `book` , `author`)的组合。 你可以指定不同的表名，详情请看附录 B。
- 我们前面已经提到，Django为自动加了一个 `id` 主键，你一样可以修改它。
- 按约定，Django添加 `"_id"` 后缀到外键字段名。这个同样也是可自定义的。
- 外键是用 `REFERENCES` 语句明确定义的。
- 这些 `CREATE TABLE` 语句会根据你的数据库而作调整，这样象数据库特定的一些字段例如： `auto_increment` (MySQL), `serial` (PostgreSQL), `integer primary key` (SQLite) 可以自动处理。 同样的，字段名称的引号也是自动处理（例如单引号还是双引号）。 这个给出的例子是Postgresql的语法。

`sqlall` 命令并没有在数据库中真正创建数据表，只是把SQL语句段打印出来。 你可以把这些语句段拷贝到你的SQL客户端去执行它。当然，Django提供了更简单的 方法来执行这些SQL语句。运行 `syncdb` 命令：

```
python manage.py syncdb
```

你将会看到这样的内容：

```
Creating table books_publisher
Creating table books_book
Creating table books_author
Installing index for books.Book model
```

`syncdb` 命令是同步你的模型到数据库的一个简单方法。它会根据 `INSTALLED_APPS` 里设置的app来检查数据库， 如果表不存在，它就会创建它。 需要注意的是， `syncdb` 并 *不能* 同步模型的修改到数据库。如果你修改了模型，然后你想更新 数据库， `syncdb` 是帮不了你的。（稍后我们再讲这些。）

如果你再次运行 `python manage.py syncdb` ，什么也没发生，因为你没有添加新的模型或者 添加新的app。所以，运行 `python manage.py syncdb` 总是安全的，它不会把事情搞砸。

如果你有兴趣，花点时间用你的SQL客户端登录进数据库服务器看看刚才Django创建的数据表。 Django带有一个命令行工具， `python manage.py dbshell` 。

基本数据访问

一旦你创建了模型，Django自动为这些模型提供了高级的Pyhton API。 运行 `python manage.py shell` 并输入下面的内容试试看：

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Addison-Wesley', address='75 Arlington Street',
...     city='Boston', state_province='MA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

这短短几行代码干了不少的事。这里简单的说一下：

- 要创建对象，只需 import 相应模型类，并传入每个字段值将其实例化。
- 调用该对象的 `save()` 方法，将对象保存到数据库中。Django 会在后台执行一条 `INSERT` 语句。
- 使用属性 `Publisher.objects` 从数据库中获取对象。调用 `Publisher.objects.all()` 获取数据库中所有的

`Publisher` 对象。此时，Django 在后台执行一条 `SELECT` SQL语句。

自然，你肯定想执行更多的Django数据库API试试看，不过，还是让我们先解决一点烦人的小问题。

添加模块的字符串表现

当我们打印整个publisher列表时，我们没有得到想要的有用的信息：

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

我们可以简单解决这个问题，只需要添加一个方法 `__str__()` 到 `Publisher` 对象。 `__str__()` 方法告诉Python要怎样把对象当作字符串来使用。 请看下面：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    **def __str__(self):**
        **return self.name**

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

    **def __str__(self):**
        **return '%s %s' % (self.first_name, self.last_name)**

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    **def __str__(self):**
        **return self.title**
```

就象你看到的一样， `__str__()` 方法返回一个字符串。 `__str__()` 必须返回字符串， 如果是其他类型，Python将会抛出 `TypeError` 错误消息 "`__str__ returned non-string`" 出来。

为了让我们的修改生效，先退出Python Shell，然后再次运行 `python manage.py shell` 进入。 现在列出 `Publisher` 对象就很容易理解了：

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

请确保你的每一个模型里都包含 `__str__()` 方法，这不只是为了交互时方便，也是因为 Django会在其他一些地方用 `__str__()` 来显示对象。

最后， `__str()__` 也是一个很好的例子来演示我们怎么添加 *行为* 到模型里。 Django的模型不只是为对象定义了数据库表的结构，还定义了对象的行为。 `__str__()` 就是一个例子来演示模型知道怎么显示它们自己。

插入和更新数据

你已经知道怎么做了：先使用一些关键参数创建对象实例，如下：

```
>>> p = Publisher(name='Apress',
...         address='2855 Telegraph Ave.',
...         city='Berkeley',
...         state_province='CA',
...         country='U.S.A.',
...         website='http://www.apress.com/')
```

这个对象实例并 *没有* 对数据库做修改。

要保存这个记录到数据库里（也就是执行 `INSERT` SQL 语句），调用对象的 `save()` 方法：

```
>>> p.save()
```

在SQL里，这大致可以转换成这样：

```
INSERT INTO book_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

因为 `Publisher` 模型有一个自动增加的主键 `id`，所以第一次调用 `save()` 还多做了一件事： 计算这个主键的值并把它赋值给这个对象实例：

```
>>> p.id
52    # this will differ based on your own data
```

接下来再调用 `save()` 将不会创建新的记录，而只是修改记录内容（也就是 执行 `UPDATE` SQL语句，而不是 `INSERT` 语句）：

```
>>> p.name = 'Apress Publishing'
>>> p.save()
```

前面执行的 `save()` 相当于下面的SQL语句：

```
UPDATE book_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

选择对象

我们已经知道查找所有数据的方法了：

```
>>> Publisher.objects.all()
[<Publisher: Addison-Wesley>, <Publisher: O'Reilly>, <Publisher: Apress Publishing>]
```

这相当于这个SQL语句：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher;
```

注意

注意到Django在选择所有数据时并没有使用 `SELECT*`，而是显式列出了所有字段。 就是这样设计的： `SELECT*` 会更慢，而且最重要的是列出所有字段遵循了Python 界的一个信条：明确比不明确好。

有关Python之禅(戒律) :-），在Python提示行输入 `import this` 试试看。

让我们来仔细看看 `Publisher.objects.all()` 这行的每个部分：

首先，我们有一个已定义的模型 `Publisher` 。没什么好奇怪的：你想要查找数据， 你就用模型来获得数据。

其次， `objects` 是干什么的？技术上，它是一个 *管理器（manager）* 。 管理器 将在附录B详细描述，在这里你只要知道它处理有关数据表的操作，特别是数据查找。

所有的模型都自动拥有一个 `objects` 管理器；你可以在想要查找数据时是使用它。

最后，还有 `all()` 方法。这是 `objects` 管理器返回所有记录的一个方法。 尽管这个对象 *看起来* 象一个列表（list），它实际是一个 *QuerySet* 对象， 这个对象是数据库中一些记录的集合。附录C将详细描述 QuerySet，现在，我们 就先当它是一个仿真列表对象好了。

所有的数据库查找都遵循一个通用模式：调用模型的管理器来查找数据。

数据过滤

如果想要获得数据的一个子集，我们可以使用 `filter()` 方法：

```
>>> Publisher.objects.filter(name="Apress Publishing")
[<Publisher: Apress Publishing>]
```

`filter()` 根据关键字参数来转换成 `WHERE` SQL语句。前面这个例子 相当于这样：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name = 'Apress Publishing';
```

你可以传递多个参数到 `filter()` 来缩小选取范围：

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress Publishing>]
```

多个参数会被转换成 `AND` SQL语句，例如象下面这样：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A.' AND state_province = 'CA';
```

注意，SQL缺省的 `=` 操作符是精确匹配的，其他的查找类型如下：

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress Publishing>]
```

在 `name` 和 `contains` 之间有双下划线。象Python自己一样，Django也使用 双下划线来做一些小魔法，这个 `__contains` 部分会被Django转换成 `LIKE` SQL语句：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name LIKE '%press%';
```

其他的一些查找类型有： `icontains` (大小写无关的 `LIKE`), `startswith` 和 `endswith` , 还有 `range` (SQL `BETWEEN` 查询）。 附录C详细列出了这些类型的详细资料。

获取单个对象

有时你只想获取单个对象，这个时候使用 `get()` 方法：

```
>>> Publisher.objects.get(name="Apress Publishing")
<Publisher: Apress Publishing>
```

这样，就返回了单个对象，而不是列表（更准确的说，QuerySet)。 所以，如果结果是多个对象，会导致抛出异常：

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
AssertionError: get() returned more than one Publisher -- it returned 2!
```

如果查询没有返回结果也会抛出异常：

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

数据排序

在运行前面的例子中，你可能已经注意到返回的结果是无序的。我们还没有告诉数据库 怎样对结果进行排序，所以我们返回的结果是无序的。

当然，我们不希望在页面上列出的出版商的列表是杂乱无章的。我们用 `order_by()` 来 排列返回的数据：

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

跟以前的 `all()` 例子差不多，SQL语句里多了指定排序的部分：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name;
```

我们可以对任意字段进行排序：

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>]
```

多个字段也没问题：

```
>>> Publisher.objects.order_by("state_provice", "address")
 [<Publisher: Apress Publishing>, <Publisher: O'Reilly>, <Publisher: Addison-Wesley>]
```

我们还可以指定逆向排序，在前面加一个减号 - 前缀：

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]
```

每次都要用 `order_by()` 显得有点啰嗦。 大多数时间你通常只会对某些 字段进行排序。在这种情况下，Django让你可以指定模型的缺省排序方式：

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    **class Meta:**
        **ordering = ["name"]**
```

这个 `ordering = ["name"]` 告诉Django如果没有显示提供 `order_by()` , 就缺省按名称排序。

Meta是什么？

Django使用内部类Meta存放用于附加描述该模型的元数据。 这个类完全可以不实现，不过他能做很多非常有用的事情。查看附录B，在Meta项下面，获得更多选项信息，

排序

你已经知道怎么过滤数据了，现在让我们来排序它们。你可以同时做这 过滤和排序，很简单，就象这样：

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>]
```

你应该没猜错，转换成SQL查询就是 `WHERE` 和 `ORDER BY` 的组合：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

你可以任意把它们串起来，多长都可以，这里没有限制。

限制返回的数据

另一个常用的需求就是取出固定数目的记录。想象一下你有成千上万的出版商在你的数据库里， 但是你只想显示第一个。你可以这样做：

```
>>> Publisher.objects.all()[0]
<Publisher: Addison-Wesley>
```

这相当于：

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name
LIMIT 1;
```

还有更多

我们只是刚接触到模型的皮毛，你还必须了解更多的内容以便理解以后的范例。 具体请看附录C。

删除对象

要删除对象，只需简单的调用对象的 `delete()` 方法：

```
>>> p = Publisher.objects.get(name="Addison-Wesley")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>]
```

你还可以批量删除对象，通过对查询的结果调用 `delete()` 方法：

```
>>> publishers = Publisher.objects.all()
>>> publishers.delete()
>>> Publisher.objects.all()
[]
```

注意

删除是 *不可恢复* 的，所以要小心操作！事实上，应该尽量避免删除对象，除非你 确实需要删除它。数据库的数据恢复的功能通常不太好，而从备份数据恢复是很痛苦的。

通常更好的方法是给你的数据模型添加激活标志。你可以只在激活的对象中查找， 对于不需要的对象，将激活字段值设为 `False` , 而不是删除对象。这样， 如果一旦你认为做错了的话，只需把标志重设回来就可以了。

修改数据库表结构

当我们在这一章的前面介绍 `syncdb` 命令的时候，我们强调 `syncdb` 仅仅创建数据库中不存在的表，而不会同步模型的修改或者删除到数据库。如果你添加或者修改了模型的一个字段，或者删除一个模型，你必须手动改变你的数据库。下面我们看看怎么来做。

当我们处理表结构的修改时，要时刻想着 Django 的数据库层是如何工作的：

- 如果模型中包含一个在数据库中并不存在的字段，Django会大声抱怨的。这样当你第一次调用Django的数据库API来查询给定的表时就会出错（也就是说，它会在执行的时候出错，而不是编译的时候）
- Django并不关心数据库表中是否存在没有在模型中定义的列
- Django并不关心数据库中是否包含没有被模型描述的表

修改表结构也就是按照正确的顺序修改各种Python代码和数据库本身

添加字段

当按照产品需求向一个表/模型添加字段时，Django不关心一个表的列是否在模型中定义，我们可以利用这个小技巧，先在数据库中添加列，然后再改变模型中对应的字段。

然而，这里总是存在先有鸡还是先有蛋的问题，为了弄清新的数据列怎么用SQL描述，你需要查看 `manage.py sqlall` 的执行结果，它列出了模型中已经存在的字段。（注意：你不需要像Django中的SQL一模一样的创建你的列，但是这确实是一个好主意，从而保证所有都是同步的）

解决鸡和蛋的问题的方法就是先在开发环境而不是发布服务器上修改。（你现在用的就是测试/开发环境，不是吗？）下面是详细的步骤。

首先，在开发环境中执行下面的步骤（也就是说，不是在发布服务器上):

1. 把这个字段添加到你的模型中.
2. 运行 `manage.py sqlall [yourapp]` 会看到模型的新的 `CREATE TABLE` 语句。 注意新的字段的列定义。
3. 启动您的数据库交互shell(也就是 `psql` 或 `mysql` ， 或者您也可以使用 `manage.py dbshell`)。 执行一个 `ALTER TABLE` 语句，添加您的新列。

4. (可选)用 `manage.py shell` 启动Python交互式shell，并通过引入模型并选择表 验证新的字段已被正确添加(比如，`MyModel.objects.all()[:5]`)。

然后在发布服务器上执行下面的步骤：

1. 启动你的数据库的交互式命令行；
2. 执行 `ALTER TABLE` 语句，也就是在开发环境中第3步执行的语句；
3. 添加字段到你的模型中。如果你在开发时使用了版本控制系统并checkin了你的修改，现在可以更新 代码到发布服务器上了（例如，使用Subverison的话就是 `svn update`)。
4. 重启Web服务器以使代码修改生效。

例如，让我们通过给 `Book` 模型添加一个 `num_pages` 字段来演示一下。 首先，我们在开发环境中这样修改模型：

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    **num_pages = models.IntegerField(blank=True, null=True)**

    def __str__(self):
```

```
        return self.title
```

（注意：我们这里为什么写 `blank=True` 和 `null=True` 呢？阅读题为“添加非空字段”的侧边栏获取更多信息。）

然后我们运行命令 `manage.py sqlall books` 来得到 `CREATE TABLE` 语句。它们看起来 是这样的：

```
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

新加的字段SQL描述是这样的：

```
"num_pages" integer NULL
```

接下来，我们启动数据库交互命令界面，例如Postgresql是执行 `psql` , 并执行下面的语句：

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

添加非空字段（NOT NULL）

这里有一个要注意的地方。在添加 `num_pages` 字段时我们使用了 `blank=True` 和 `null=True` 可选项。 我们之所以这么做是因为在数据库创建时我们想允许字段值为NULL。

当然，也可以在添加字段时设置值不能为NULL。要实现这个，你不得不先创建一个 `NULL` 字段， 使用缺省值，再修改字段到 `NOT NULL` 。例如：

```
BEGIN;
ALTER TABLE books_book ADD COLUMN num_pages integer;
UPDATE books_book SET num_pages=0;
ALTER TABLE books_book ALTER COLUMN num_pages SET NOT NULL;
COMMIT;
```

如果你这样做了， 记得要把 `blank=True` 和 `null=True` 从你的模型中拿掉。

执行完 `ALTER TABLE` 语句， 我们确认一下修改是否正确，启动Python交互界面并执行下面语句：

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

如果没有错误，我们就可以转到发布服务器来在数据库上执行 `ALTER TABLE` 语句了。然后， 再更新模型并重启WEB服务器。

删除字段

从模型里删除一个字段可要比增加它简单多了。删除一个字段仅需要做如下操作：

从你的模型里删除这个字段，并重启Web服务器。

使用如下面所示的命令，从你的数据库中删掉该列：

```
ALTER TABLE books_book DROP COLUMN num_pages;
```

删除 Many-to-Many 字段

因为many-to-many字段同普通字段有些不同，它的删除过程也不一样：

删除掉你的模型里的 `ManyToManyField` ，并且重启Web服务器。

使用如下面所示的命令，删除掉你数据库里的many-to-many表：

```
DROP TABLE books_books_publishers;
```

删除模型

完全删除一个模型就像删除一个字段一样简单。删除模型仅需要做如下步骤：

将此模型从你的 `models.py` 文件里删除，并且重启Web服务器。

使用如下的命令，将此表从你的数据库中删除：

```
DROP TABLE books_book;
```

下一步？

一旦你定义了你的模型，接下来就是要把数据导入数据库里了。你可能已经有现成的数据了，请看第十六章，如何集成现有的数据库。也可能数据是用户提供的，第七章中还会教你怎么处理用户提交的数据。

有时候，你和你的团队成员也需要手工输入数据，这时候如果能有一个基于Web的数据输入和管理的界面 就很有帮助。下一章将讲述Django的管理界面，它就是专门干这个活的。

第六章：Django管理站点

对于某些网站来说，一个“管理员界面”是基础功能中的重要部分。这是一个基于Web的界面，仅限受信任的站点管理员来添加，编辑和删除网站的内容。您用于发表博客的界面，用于审核读者评论的站点管理员后台，你所建立的使您的客户端使用更新网站上的新闻稿的工具，这些都是“管理员界面”的实际运用。

但是管理界面有一问题：创建它太繁琐。当你开发对公众的功能时，网页开发是有趣的，但是创建管理界面通常是千篇一律的。你必须认证用户，显示并管理表格，验证输入的有效性诸如此类。这很繁琐而且是重复劳动。

那么，来看看Django是怎么处理这些枯燥的，重复性的任务吧。只要短短的几行代码，它就会为你做这一切，一点也不会欠缺。使用Django，建立一个管理界面，是一个已经解决了的问题。

这一章是关于 Django 的自动管理界面。这个特性是这样起作用的：它读取你模式中的元数据，然后提供给你一个强大而且可以使用的界面，网站管理者可以用它立即工作。在这里我们将讨论如何激活，使用和定制这个特性。

激活管理员界面

我们认为管理界面是 Django 中最酷的一部分，大部分 Django 用户也这么想。但是不是所有人都需要它，所以它是可选的。这也就意味着你需要跟着三个步骤来激活它。

在你的 models 中加入admin metadata。

不是所有的models都能够(或应该)被管理员编辑，你需要给models标记一个管理员接口(interface)，通过给models添加一个内部类‘admin’完成接口标记。所以，给上一章我们的“book”models添加管理员接口就像下面这样:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return self.title

    **class Admin:**
        **pass**
```

Admin 声明标志了该类有一个管理界面。在 Admin 之下你可以放很多选项，但目前我们只关注缺省的东西，所以我们只在那写上 pass 让 Python 知道 Admin 类是空的。

如果你正跟着例子在写你的代码，现在你可以在 Publisher 和 Author 类中加入 Admin 声明。

安装管理应用程序。在你的 INSTALLED_APPS 的设置中加入 "django.contrib.admin" 。

如果你是一直照步骤做下来的，请确认 "django.contrib.sessions" , "django.contrib.auth" , 和 "django.contrib.contenttypes" 前面的注释已去掉，因为管理程序需要它们。请同时去掉所有 MIDDLEWARE_CLASSES 设置行中的注释，并清除 TEMPLATE_CONTEXT_PROCESSOR 设置，以便它可以重新使用缺省值。

运行 python manage.py syncdb 。这一步将生成管理界面使用的额外数据库表。

注释

When you first run syncdb with "django.contrib.auth" in INSTALLED_APPS, youll be asked about creating a superuser. If you didnt do so at that time, youll need to run django/contrib/auth/bin/create_superuser.py to create an admin user. Otherwise, you wont be able to log in to the admin interface.

在你的 `urls.py` 中加入模板。如果你仍在用 `startproject` 生成的 `urls.py` 文件，管理 URL 模板已经在里面了，你需要去掉注释。任何一个方式的 URL 模板应该像下面这样：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    **(r'^admin/', include('django.contrib.admin.urls')),**
)
```

就是这样。现在运行 `python manage.py runserver` 以启动开发服务器。你将看到像下面这样的东西：

```
Validating models...
0 errors found.

Django version 0.96, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

现在你可以访问 Django 给你的URL (`http://127.0.0.1:8000/admin/` 在进行的例子中)，登录，随便看看。

使用管理员界面

管理界面的设计是针对非技术人员的，所以它应该是自我解释的。无论如何，有关管理界面特性的一些注释是完善的。

你看到的第一件事是如图6-1所示的登录屏幕.

图 6-1.Django 登录屏幕

使用当你添加超级用户时创建的用户名和密码，在你登入时，你将会看到你可以管理用户，群组和权限(会有更多介紹)

每一个有 `Admin` 声明的对象都在主索引页显示，见图 6-2。

图 6-2。Django 主管理索引

添加和更改对像的链接将导出两个页面，这两个页面是指向 *更改列表* 和 *编辑表格* 两个对像。如图6-3所示，更改列表主要是系统对像的索引页面。

图 6-3. 典型的改变列表视图

在这些列表里的栏目有不少选项控制，这些显示出一些额外的特性，比如下拉式日期选择控制，搜索栏，过滤界面。我们稍后讨论这些特性的细节。

编辑表格是用来修改现有对象和创建新对象的（见图6-4）。在你模式中定义的域在这里都显示出来，你也许注意到不同类型的域用不同的控件显示（如：日期/时间域有日历控件，外键用选择栏等等）。

图 6-4. 典型的编辑表格

你还能看到管理界面也控制着你输入的有效性。你可以试试不填必需的栏目或者在时间栏里填错误的时间，你会发现当你要保存时会出现错误信息，如图6-5所示。

图6-5. 编辑表格显示错误信息

当你编辑已有的对像时，你在窗口的右上角可以看到一个历史按钮。通过管理界面做的每一个改变都留有记录，你可以按历史键来检查这个记录（见图6-6）。

图6-6. Django 对像历史页面

当你删除现有对象时，管理界面会要求你确认删除动作以免引起严重错误。删除也是*联级*的：删除确认页面会显示所有将要删除的关联对象（见图6-7）。

图 6-7. Django 删除确认页面

用户、组和许可

因为你是用超级用户登录的，你可以创建，编辑和删除任何对像。然而管理界面有一个用户许可系统，你可以用它来给其它用户授与他们需要的部分权力。

你通过管理界面编辑用户及其许可就像你编辑别的对象一样。 `用户` 和 `组` 模式的链接和你自己定义的所有对像一样列在管理索引页面。

用户对像有你期望的标准用户名，密码，e-mail和真实姓名域，同时它还有在管理界面里这个用户可以做什么。首先，这有一组三个标记：

- 这是激活标志，它用来控制用户是否已经激活。如果这个标志关闭，这个用户就不能浏览任何需要登录的URL。
- 这是成员标志，它用来控制这个用户是否可以登录管理界面（如：这个用户是不是你组织的成员）。由于同一个用户系统也用来控制公共（如：非管理）站点的访问（见十二章），本标志区分公共用户和管理员。
- 这是超级用户标志，它给用户所有权限，在管理界面可以自由进入，常规许可无效。

普通的活跃，非超级用户的管理用户可以根据一套设定好的许可进入。通过管理界面编辑的每个对像有三个许可： *创建* 许可， *编辑* 许可和 *删除* 许可。给一个用户授权许可也就表明该用户可以进行许可描述的操作。

注

权限管理系统也控制编辑用户和权限。如果你给某人编辑用户的权限，他可以编辑自己的权限，这种能力可能不是你希望的。

你也可以给组中分配用户。一个 *组* 简化了给组中所有成员应用一套许可的动作。 组在给大量用户特定权限的时候很有用。

定制管理界面

你可以通过很多方法来定制管理界面的外观和行为。在本节我们只谈及与我们 `Book` 相关的一些方法，第十七章将讨论定制管理界面的细节问题。

目前为止我们书的改变列表只显示一个字符串，这个字符串是在模式中的 `__str__` 中加入来代表这个模式的。这种方式在只有几本书的情况下工作得很好，但如果有成百上千中书的时候，找一本书就像大海捞针。但是我们可以很容易地在界面中加入搜索和过滤功能。改变 `Admin` 声明如下：

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

```
class Admin:
    **list_display = ('title', 'publisher', 'publication_date')**
    **list_filter = ('publisher', 'publication_date')**
    **ordering = ('-publication_date',)**
    **search_fields = ('title',)**
```

这四行代码戏剧性地改变了我们的列表界面，如图6-8所示。

图 6-8. 修改过的变化列表页面

每一行说明管理界面的不同部分：

`list_display` 选项控制变更列表所显示的列。缺省情况下变更列表只显示对像包含的 表征字符串。我们在这改变成显示标题，出版商和出版日期。

`list_filter` 选项在右边创建一个过滤条。我们允许它按日期过滤（它可以让你只显示过去一周，一个月等等出版的书籍）和按出版商过滤。

你可以在管理界面中指定任何域做为过滤器，但是用外键，日期，布尔值和有 `choices` 属性的域是最适合的。过滤至少显示2个值。

`list_filter` 选项在右边创建一个过滤条。我们允许它按日期过滤（它可以让你只显示过去一周，一个月等等出版的书籍）和按出版商过滤。

最后， `search_fields` 选项创建了一个允许搜索文本内容的域。它可以搜索 `title` 字段中的内容（所以您可以输入 **Django** 以显示所有题名中包含有 Django 的书籍）。

通过使用这些选项（还有一些是在十二章描述的）你能够用很少的代码实现很强大，产品级的数据编辑界面。

定制管理界面的外观和感觉

显然，如果在每个管理页面的头部都包含“Django administration”这行字是搞笑的。这行字只是块标签的占位符。

通过Django模板系统可以很容易的修改它。Django管理站点同样是用Django编写的，它的用户 界面使用Django自己的模板系统。（关于Django模板系统请参见第四章。）

我们在第四章已经讲到， `TEMPLATE_DIRS` 配置设置了Django加载模板的目录列表。 要自定义Django的管理模板，只需要拷贝Django发行版中的整个管理模板到你在 `TEMPLATE_DIRS` 里设置的模板目录里。

管理站点的头部区域在模板 `admin/base_site.html` 里。缺省情况下，这个模板在 Django管理模板目录 `django/contrib/admin/templates` 里，你可以在Django的安装 目录找到它，例如Python的 `site-packages` 目录或者你安装的其他目录。要自定义 这个 `base_site` 模板，把这个模板拷贝到你的模板目录下的 `admin` 子目录。 例如，假定你的模板目录是 `"/home/mytemplates"` ，拷贝 `django/contrib/admin/templates/admin/base_site.html` 到 `/home/mytemplates/admin/base_site.html` 。不要忘了有 `admin` 子目录。

然后，编辑这个新 `admin/base_site.html` 文件，替换你自己站点的名称上去。

备注 每个Django缺省的管理模板都可以重载。要重载一个模板，就象 ``base_site.html``一样的去做：把它从缺省目录中拷贝到你自己的模板目录中然后修改它

System Message: WARNING/2 (`<string>`, line 450); *backlink*

Inline literal start-string without end-string.

你可能会想到是这么一回事，如果 `TEMPLATE_DIRS` 缺省是空的，Django就使用缺省的管理 模板。正确的回答是，缺省情况下，Django自动在每个app里的 `templates/` 子目录里搜索 模板来做后备。具体请看第十章中的编写自定义模板加载器章节。

定制管理索引页面

你同样可以自定义Django管理的索引页面（index page)。缺省情况下，它将显示在 `INSTALL_APPS` 配置里设置的所有应用程序，按应用程序的名称排序。你可能想要修改 排序方式来让你更容易找到你想要的应用程序。毕竟，索引可能是管理界面中最重要的页面， 所以要容易使用才行。

要自定义的模板是 `admin/index.html` 。（记得象前面例子一样拷贝 `admin/index.html` 到你的模板目录。）打开这个文件，你会看到一个叫做 `{% get_admin_app_list as app_list %}` 的模板标签，你可以在这里硬编码你想要的管理页面连接。如果你不喜欢硬编码的方式，请参看 第十章中实现你自己的模板标签章节。

在这里，Django提供了一个快捷方式。运行命令 `python manage.py adminindex <app>` 来获取可以包含在管理索引模板里的一段代码。这是一个很有用的起点。

有关于Django管理站点自定义的详细内容，请参看第十七章。

使用管理员界面的时机和原因

我们认为Django的管理界面是很有吸引力的。事实上，我们称它为Django的杀手锏之一。当然， 我们也经常被问道 *我们* 应该在什么情况下使用管理界面，为什么呢？多年实践经验让我们 发现了一些使用管理界面的模式，会对大家很有帮助。

显然，Django管理界面对编辑数据特别有用（难以置信的棒！）。如果你有任何需要输入数据的 任务，管理界面是再合适不过了。我相信大家肯定都有很多要输入数据的任务吧？

Django的管理界面对非技术用户要输入他们的数据时特别有用；事实上这个特性就是专门为这个 实现的。在Django最开始开发的新闻报道的行业应用中，有一个典型的在线自来水的水质专题报道 应用，它的实现流程是这样的：

- 负责这个报道的记者和要处理数据的开发者碰头，提供一些数据给开发者。
- 开发者围绕这些数据设计模型然后配置一个管理界面给记者。
- 在记者输入数据到Django中去的时候，编程人员就可以集中注意力到开发公共访问界面上 （这可是有趣的部分啊！）。

换句话说，Django的管理界面为内容输入人员和编程人员都提供了便利的工具。

当然，除了数据输入方面，我们发现管理界面在下面这些情景中也是很有用的：

- *检查数据模型*：在我们定义了数据模型后做的第一件事就是输入一些测试数据。 这可以帮助我们检查数据模型的错误；有一个图形界面可以很快的发现错误。
- *管理已输入的数据*：象 `http://chicagocrime.org` 这样的网站，通常只有少部分 数据是手工输入的，大部分数据是自动导入的。如果自动导入的数据有问题，就可以用管理 界面来编辑它。

下一步是什么？

现在我们已经创建了一些模式，为编辑数据配置了一个顶尖的界面。在下一章里，我们将要转到真正的网站开发上：表单的创建和处理。

第七章 表单处理

本章作者是Simon Willison

经过上一章，你应该对简单网站有个全面的认识。这一章，来处理web开发的下一个难题：建立用户输入的视图。

我们会从手工打造一个简单的搜索页面开始，看看怎样处理浏览器提交而来的数据。然后我们开始使用Django的forms框架。

搜索

在web应用上，有两个关于搜索获得巨大成功的故事：Google和Yahoo，通过搜索，他们建立了几十亿美元的业务。几乎每个网站都有很大的比例访问量来自这两个搜索引擎。甚至，一个网站是否成功取决于其站内搜索的质量。因此，在我们这个网站添加搜索功能看起来好一些。

开始，在URLconf(`mysite.urls`)添加搜索视图。添加类似 `(r'^search/$','mysite.books.views.search')` 设置URL模式。

下一步，在视图模块(`mysite.books.views`)中写这个 `search` 视图:

```
from django.db.models import Q
from django.shortcuts import render_to_response
from models import Book

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

这里有一些需要注意的，首先 `request.GET`，这是如何从Django中访问GET数据；POST数据通过类似的 `request.POST` 对象访问。这些对象行为与标准Python字典很像，在附录H中列出来其另外的特性。

什么是 GET and POST 数据?

GET 和POST 是浏览器使用的两个方法，用于发送数据到服务器端。 一般来说，会在html表单里面看到:

```
<form action="/books/search/" method="get">
```

它指示浏览器向/books/search/以GET的方法提交数据

关于GET和POST这两个方法之间有很大的不同，不过我们暂时不深入它，如果你想了解更多，可以访问：http://www.w3.org/2001/tag/doc/whenToUseGet.html 。

所以下面这行：

```
query = request.GET.get('q', '')
```

寻找名为 `q` 的GET参数，而且如果参数没有提交，返回一个空的字符串。

注意在 `request.GET` 中使用了 `get()` 方法，这可能让大家不好理解。这里的 `get()` 是每个python的的字典数据类型都

有的方法。使用的时候要小心：假设 `request.GET` 包含一个 `'q'` 的key是不安全的，所以我们使用 `get('q', '')` 提供一个缺省的返回值 `''` (一个空字符串)。如果只是使用 `request.GET['q']` 访问变量，在Get数据时 `q` 不可得,可能引发 `KeyError`.

其次,关于 `Q`, `Q` 对象在这个例子里用于建立复杂的查询,搜索匹配查询的任何书籍.技术上 `Q` 对象包含QuerySet,可以在附录C中进一步阅读.

在这个查询中， `icontains` 使用SQL的 `LIKE` 操作符，是大小写不敏感的。

既然搜索依靠多对多域来实现，就有可能对同一本书返回多次查询结果（例如：一本书有两个作者都符合查询条件）。因此添加 `.distinct()` 过滤查询结果，消除重复部分。

现在仍然没有这个搜索视图的模板，可以如下实现:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Search{% if query %} Results{% endif %}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>

  {% if query %}
    <h2>Results for "{{ query|escape }}":</h2>

    {% if results %}
      <ul>
      {% for book in results %}
        <li>{{ book|escape }}</l1>
      {% endfor %}
      </ul>
    {% else %}
      <p>No books found</p>
    {% endif %}
  {% endif %}
</body>
</html>
```

希望你已经很清楚地明白这个实现。不过，有几个细节需要指出：

> 表单的action是 `.`， 表示当前的URL。这是一个标准的最佳惯常处理方式：不使用独立 的视图分别来显示表单页面和结果页面；而是使用单个视图页面来处理表单并显示搜索结果。
>
> 我们把返回的查询值重新插入到 `<input>` 中去，以便于读者可以完善他们的搜索内容， 而不必重新输入搜索内容。
>
> 在所有使用 `query` 和 `book` 的地方，我们通过 `escape` 过滤器来确保任何 可能的恶意的搜索文字被过滤出去，以保证不被插入到页面里。
>
> 这对处理任何用户提交数据来说是 *必须* 的！否则的话你就开放你的网站允许跨站点脚本 （XSS）攻击。在第十九章中将详细讨论了XSS和安全。
>
> 不过，我们不必担心数据库对可能有危害内容的查询的处理。 Django的数据库层在这方面已经做过安全处理。 【译注：数据库层对查询数据自动Escape，所以不用担心】

现在我们已经作了搜索。进一步要把搜索表单加到所有的页面（例如，在base模板）；这个可以由你自己完成。

下面，我们看一下更复杂的例子。事先我们讨论一个抽象的话题：完美表单。

完美表单

表单经常引起站点用户的反感。我们考虑一下一个假设的完美的表单的行为：

- 它应该问用户一些信息，显然，由于可用性的问题， 使用HTML `<label>` 元素和有用的 上下文帮助是很重要的。
- 所提交的数据应该多方面的验证。Web应用安全的金科玉律是从不要相信进来的数据，所以验证是必需的。
- 如果用户有一些错误，表单应该重新显示详情，错误信息。原来的数据应该已经填好，避免用户重新录入，
- 表单应该在所有域验证正确前一直重新显示。

建立这样的表单好像需要做很多工作！幸好，Django的表单框架已经设计的可以为你做绝大部分的工作。你只需要提供表单域的描述，验证规则和简单的模板即可。这样就只需要一点的工作就可以做成一个完美的表单。

创建一个回馈表单

做好一个网站需要注意用户的反馈，很多站点好像忘记这个。他们把联系信息放在FAQ后面，而且好像很难联系到实际的人。

一个百万用户级的网站，可能有些合理的策略。如果建立一个面向用户的站点，需要鼓励回馈。我们建立一个简单的回馈表单，用来展示Django的表单框架。

开始，在URLconf里添加 `(r'^contact/$', 'mysite.books.views.contact')` ，然后定义表单。 在Django中表单的创建类似MODEL:使用Python类来声明。这里是我们简单表单的类。为了方便，把它写到新的 `forms.py` 文件中，这个文件在app目录下。

```
from django import newforms as forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)

class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField()
    sender = forms.EmailField(required=False)
```

New Forms是什么？

当Django最初推出的时候，有一个复杂而难用的form系统。用它来构建表单简直就是噩梦，所以它在新版本里面被一个叫做newforms的系统取代了。但是鉴于还有很多代码依赖于老的那个form系统，暂时Django还是同时保有两个forms包。

在本书写作期间，Django的老form系统还是在 `django.forms` 中，新的form系统位于 `django.newforms` 中。这种状况迟早会改变， `django.forms` 会指向新的form包。 但是为了让本书中的例子尽可能广泛地工作，所有的代码中仍然会使用 `django.newforms` 。

一个Django表单是 `django.newforms.Form` 的子类，就像Django模型是 `django.db.models.Model` 的子类一样。在django.newforms模块中还包含很多Field类；Django的文档（ http://www.djangoproject.com/documentation/0.96/newforms/ ）中包含了一个可用的Field列表。

我们的 `ContactForm` 包含三个字段：一个topic，它是一个三选一的选择框；一个message，它是一个文本域；还有一个sender，它是一个可选的email域（因为即使是匿名反馈也是有用的）。还有很多字段类型可供选择，如果它们都不满足要求，你可以考虑自己写一个。

form对象自己知道如何做一些有用的事情。它能校验数据集合，生成HTML“部件”，生成一集有用的错误信息，当然，如果你确实很懒，它也能绘出整个form。现在让我们把它嵌入一个视图，看看怎么样使用它。在views.py里面：

```
from django.db.models import Q
from django.shortcuts import render_to_response
from models import Book
**from forms import ContactForm**
```

```
def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })

**def contact(request):**
    **form = ContactForm()**
    **return render_to_response('contact.html', {'form': form})**
```

添加contact.html文件：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    <form action="." method="POST">
        <table>
            {{ form.as_table }}
        </table>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

最有意思的一行是 {{ form.as_table }}。form是ContactForm的一个实例，我们通过render_to_response方法把它传递给模板。as_table是form的一个方法，它把form渲染成一系列的表格行(as_ul和as_p也是起着相似的作用)。生成的HTML像这样：

```
<tr>
    <th><label for="id_topic">Topic:</label></th>
    <td>
        <select name="topic" id="id_topic">
            <option value="general">General enquiry</option>
            <option value="bug">Bug report</option>
            <option value="suggestion">Suggestion</option>
        </select>
    </td>
</tr>
<tr>
    <th><label for="id_message">Message:</label></th>
    <td><input type="text" name="message" id="id_message" /></td>
</tr>
<tr>
    <th><label for="id_sender">Sender:</label></th>
    <td><input type="text" name="sender" id="id_sender" /></td>
</tr>
```

请注意：<table>和<form>标签并没有包含在内；我们需要在模板里定义它们，这给予我们更大的控制权去决定form提交时的行为。Label元素是包含在内的，令访问性更佳（因为label的值会显示在页面上）。

我们的form现在使用了一个<input type="text">部件来显示message字段。但我们不想限制我们的用户只能输入一行文本，所以我们用一个<textarea>部件来替代：

```
class ContactForm(forms.Form):
```

```
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(**widget=forms.Textarea()** )
    sender = forms.EmailField(required=False)
```

forms框架把每一个字段的显示逻辑分离到一组部件（widget）中。每一个字段类型都拥有一个默认的部件，我们也可以容易地替换掉默认的部件，或者提供一个自定义的部件。

现在，提交这个form没有在后台做任何事情。让我们把我们的校验规则加进去：

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

一个form实例可能处在两种状态：绑定或者未绑定。一个绑定的实例是由字典（或者类似于字典的对象）构造而来的，它同样也知道如何验证和重新显示它的数据。一个未绑定的form是没有与之联系的数据，仅仅知道如何显示其自身。

现在可以试着提交一下这个空白的form了。页面将会被重新显示出来，显示一个验证错误，提示我们message字段是必须的。

现在输入一个不合法的email地址，EmailField知道如何验证email地址，大多数情况下这种验证是合理的。

设置初始数据

向form的构造器函数直接传递数据会把这些数据绑定到form，指示form进行验证。我们有时也需要在初始化的时候预先填充一些字段——比方说一个编辑form。我们可以传入一些初始的关键字参数：

```
form = CommentForm(initial={'sender': 'user@example.com'})
```

如果我们的form总是会使用相同的默认值，我们可以在form自身的定义中设置它们

```
message = forms.CharField(widget=forms.Textarea(),
                          **initial="Replace with your feedback"** )
```

处理提交

当用户填完form，完成了校验，我们需要做一些有用的事情了。在本例中，我们需要构造并发送一个包含了用户反馈的email，我们将会使用Django的email包来完成

首先，我们需要知道用户数据是不是真的合法，如果是这样，我们就要访问已经验证过的数据。forms框架甚至做的更多，它会把它们转换成对应的Python类型。我们的联系方式form仅仅处理字符串，但是如果我们使用IntegerField或者DataTimeField，forms框架会保证我们从中取得类型正确的值。

测试一个form是否已经绑定到合法的数据，使用is_valid()方法：

```
form = ContactForm(request.POST)
if form.is_valid():
    # Process form data
```

现在我们要访问数据了。我们可以从request.POST里面直接把它们取出来，但是这样做我们就丧失了由framework为我们自动做类型转换的好处了。所以我们要使用form.clean_data:

```
if form.is_valid():
    topic = form.clean_data['topic']
    message = form.clean_data['message']
    sender = form.clean_data.get('sender', 'noreply@example.com')
    # ...
```

请注意因为sender不是必需的，我们为它提供了一个默认值。终于，我们要记录下用户的反馈了，最简单的方法就是把它发送给站点管理员，我们可以使用send_mail方法：

```
from django.core.mail import send_mail
```

```
# ...

send_mail(
    'Feedback from your site, topic: %s' % topic,
    message, sender,
    ['administrator@example.com']
)
```

send_mail方法有四个必须的参数：主题，邮件正文，from和一个接受者列表。send_mail是Django的EmailMessage类的一个方便的包装，EmailMessage类提供了更高级的方法，比如附件，多部分邮件，以及对于邮件头部的完整控制。 发送完邮件之后，我们会把用户重定向到确认的页面。完成之后的视图方法如下：

发送完邮件之后，我们会把用户重定向到确认的页面。完成之后的视图方法如下：

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.core.mail import send_mail
from forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            topic = form.clean_data['topic']
            message = form.clean_data['message']
            sender = form.clean_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message, sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

在POST之后立即重定向

在一个POST请求过后，如果用户选择刷新页面，这个请求就重复提交了。这常常会导致我们不希望的行为，比如重复的数据库记录。在POST之后重定向页面是一个有用的模式，可以避免这样的情况出现：在一个POST请求成功的处理之后，把用户导引到另外一个页面上去，而不是直接返回HTML页面。

1d0p2u azetemwaeinp, [url=http://lcaytpsbsxkh.com/]lcaytpsbsxkh[/url], [link=http://hgjnhjgnqlch.com/]hgjnhjgnqlch[/link], http://curhltadaokb.com/

假设我们已经发布了反馈页面了，email已经开始源源不断地涌入了。只有一个问题：一些email只有寥寥数语，很难从中得到什么详细有用的信息。所以我们决定增加一条新的校验：来点专业精神，最起码写四个字，拜托。

我们有很多的方法把我们的自定义校验挂在Django的form上。如果我们的规则会被一次又一次的使用，我们可以创建一个自定义的字段类型。大多数的自定义校验都是一次性的，可以直接绑定到form类.

我们希望message字段有一个额外的校验，我们增加一个clean_message方法：

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)

    def clean_message(self):
        message = self.clean_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

这个新的方法将在默认的字段校验器之后被调用（在本例中，就是CharField的校验器）。因为字段数据已经被部分地

处理掉了，我们需要从form的clean_data字典中把它弄出来。

我们简单地使用了len()和split()的组合来计算单词的数量。如果用户输入了过少的词，我们扔出一个ValidationError。这个exception的错误信息会被显示在错误列表里。

在函数的末尾显式地返回字段的值非常重要。我们可以在我们自定义的校验方法中修改它的值（或者把它转换成另一种Python类型）。如果我们忘记了这一步，None值就会返回，原始的数据就丢失掉了。

自定义视感

修改form的显示的最快捷的方式是使用CSS。错误的列表可以做一些视觉上的增强，<ul>标签的class属性为了这个目的。下面的CSS让错误更加醒目了：

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

虽然我们可以方便地使用form来生成HTML，可是默认的渲染在多数情况下满足不了我们的应用。{{form.as_table}}和其它的方法在开发的时候是一个快捷的方式，form的显示方式也可以在form中被方便地重写。

每一个字段部件(<input type="text">, <select>, <textarea>, 或者类似)都可以通过访问{{form.字段名}}进行单独的渲染。任何跟字段相关的错误都可以通过{{form.fieldname.errors}}访问。我们可以同这些form的变量来为我们的表单构造一个自定义的模板：

```
<form action="." method="POST">
    <div class="fieldWrapper">
        {{ form.topic.errors }}
        <label for="id_topic">Kind of feedback:</label>
        {{ form.topic }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
    <div class="fieldWrapper">
        {{ form.sender.errors }}
        <label for="id_sender">Your email (optional):</label>
        {{ form.sender }}
    </div>
    <p><input type="submit" value="Submit"></p>
</form>
```

`{{ form.message.errors }}` 会在 `<ul class="errorlist">` 里面显示，如果字段是合法的，或者form没有被绑定，就显示一个空字符串。我们还可以把 `form.message.errors` 当作一个布尔值或者当它是list在上面做迭代：

```
<div class="fieldWrapper{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ol>
        {% for error in form.message.errors %}
            <li><strong>{{ error|escape }}</strong></li>
        {% endfor %}
        </ol>
    {% endif %}
    {{ form.message }}
```

```
</div>
```

在校验失败的情况下, 这段代码会在包含错误字段的div的class属性中增加一个”errors”，在一个有序列表中显示错误信息。

从模型创建表单

我们弄个有趣的东西吧：一个新的form，提交一个新出版商的信息到我们第五章的book应用。

一个非常重要的Django的开发理念就是不要重复你自己(DRY)。Any Hunt和Dave Thomas在《实用主义程序员》里定义了这个原则：

在系统内部，每一条（领域相关的）知识的片断都必须有一个单独的，无歧义的，正式的表述。

我们的出版商模型拥有一个名字，地址，城市，州（省），国家和网站。在form中重复这个信息无疑违反了DRY原则。我们可以使用一个捷径：form_for_model()：

```
from models import Publisher
from django.newforms import form_for_model

PublisherForm = form_for_model(Publisher)
```

PublisherForm是一个Form子类，像刚刚手工创建的ContactForm类一样。我们可以像刚才一样使用它：

```
from forms import PublisherForm

def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render_to_response('books/add_publisher.html', {'form': form})
```

`add_publisher.html` 文件几乎跟我们的contact.html模板一样，所以不赘述了。记得在URLConf里面加上：`(r'^add_publisher/$', 'mysite.books.views.add_publisher')`.

还有一个快捷的方法。因为从模型而来的表单经常被用来把新的模型的实例保存到数据库，从 `form_for_model` 而来的表单对象包含一个 `save()` 方法。一般情况下够用了；你想对提交的数据作进一步的处理的话，无视它就好了。

`form_for_instance()` 是另外一个方法，用于从一个模型对象中产生一个初始化过的表单对象，这个当然给“编辑”表单提供了方便。

下一步？

这一章已经完成了这本书的介绍性的材料。接下来的十三个章节讨论了一些高级的话题，包括生成非html内容（第11章），安全（第19章）和部署（第20章）。

在本书最初的七章后，我们（终于）对于使用Django构建自己的网站已经知道的够多了，接下来的内容可以在需要的时候阅读。

第八章里我们会更进一步地介绍视图和URLConfs（介绍见第三章）。

第八章 高级视图和URL配置

第三章，我们讲到DJANGO基本的视图功能和URL配置，这一章将涉及更多细节和高级功能。

URLconf 技巧

psiOnM nbrzcxnliggw, [url=http://nbeqcslpbjht.com/]nbeqcslpbjht[/url], [link=http://xmxroewwggzy.com/]xmxroewwggzy[/link], http://nabarnvmapgz.com/

Yfphr6 wkchavkykyhr, [url=http://muokhwblpfjl.com/]muokhwblpfjl[/url], [link=http://ploxqieojkeb.com/]ploxqieojkeb[/link], http://xhutfvdzimfx.com/

看下这个 URLconf，它是建立在第三章的例子上:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead, hours_behind, now_in_chicago, now_in_london

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus(\d{1,2})hours/$', hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', hours_behind),
    (r'^now/in_chicago/$', now_in_chicago),
    (r'^now/in_london/$', now_in_london),
)
```

正如第三章中所解释的，在 URLconf 中的每一个入口包括了它所联系的视图函数，直接传入了一个函数对象。这就意味着需要在模块开始处导入视图函数。

但随着 Django 应用变得复杂，它的 URLconf 也在增长，并且维护这些导入可能使得管理变麻烦。(对每个新的view函数，你不得不记住要导入它，并且如果采用这种方法导入语句将变得相当长。)有可能通过导入 `views` 模块本身来避免这个麻烦。这个 URLconf 示例同上一个是等价的:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^now/$', views.current_datetime),
    (r'^now/plus(\d{1,2})hours/$', views.hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', views.hours_behind),
    (r'^now/in_chicago/$', views.now_in_chicago),
    (r'^now/in_london/$', views.now_in_london),
)
```

Django 还提供了另一种方法可以在 URLconf 中为某个特别的模式指定视图函数：你可以传入一个包含模块名和函数名的字符串，而不是函数对象本身。继续示例:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^now/$', 'mysite.views.current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'mysite.views.hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'mysite.views.hours_behind'),
    (r'^now/in_chicago/$', 'mysite.views.now_in_chicago'),
    (r'^now/in_london/$', 'mysite.views.now_in_london'),
)
```

(注意视图名前后的引号。应该使用带引号的 `'mysite.views.current_datetime'` 而不是 `mysite.views.current_datetime` 。)

使用这个技术，就不必导入视图函数了；Django 会在第一次需要它时导入合适的视图函数，根据字符串所描述的视图函数的名字和路径。

当使用字符串技术时，你可以采用更简化的方式：提取出一个公共视图前缀。在我们的 URLconf 例子中，每一个视图字符串都是以 `'mysite.views'` 开始的，造成过多的输入。我们可以提取出公共前缀然后把它作为第一个参数传给 `patterns()`，如:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^now/$', 'current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'hours_behind'),
    (r'^now/in_chicago/$', 'now_in_chicago'),
    (r'^now/in_london/$', 'now_in_london'),
)
```

注意既不要在前缀后面跟着一个点号(`"."`)，也不要在视图字符串前面放一个点号。 Django 会自动处理它们。

牢记这两种方法，哪种更好一些呢？这取决于你的个人编码习惯和需要。

字符串方法的好处如下:

N9gqns wteejridtwke, [url=http://lxotonbmgclb.com/]lxotonbmgclb[/url], [link=http://hkhsettjexoq.com/]hkhsettjexoq[/link], http://dlapxtivyork.com/

- 如果你的视图函数存在于几个不同的 Python 模块的话，它可以使得 URLconf 更易读和管理。

WdAon8 gnuhdlcakrdi, [url=http://mkbdhjkqvlqt.com/]mkbdhjkqvlqt[/url], [link=http://rdpcyyxxithz.com/]rdpcyyxxithz[/link], http://rsbyearxigne.com/

- 更容易对视图函数进行包装(wrap)。参见本章后面的《包装视图函数》一节。
- 更 Pythonic，更符合 Python 的传统，如把函数当成对象传递。

两个方法都是有效的，甚至你可以在同一个 URLconf 中混用它们。决定权在你。

使用多个视图前缀

在实践中，如果你使用字符串技术，特别是当你的 URLconf 中没有一个公共前缀时，你最终可能混合视图。然而，你仍然可以利用视图前缀的简便方式来减少重复。只要增加多个 `patterns()` 对象，象这样:

旧的:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^/?$', 'mysite.views.archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

新的:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^/?$', 'archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'archive_month'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

整个框架关注的是存在一个名为 `urlpatterns` 的模块级别的变量。这个变量可以动态构建，正如本例中我们所做的一样。

调试模式中的特例

当谈到动态构建 `urlpatterns` 时，你可能想利用这一技术，在 Django 的调试模式时，来修改 URLconf 的行为。为了做到这一点，只要在运行时检查 `DEBUG` 配置项的值即可，如:

```
from django.conf.urls.defaults import*
from django.conf import settings

urlpatterns = patterns('',
    (r'^$', 'mysite.views.homepage'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
)

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo$', 'mysite.views.debug'),
    )
```

在这个例子中，URL `/debuginfo/` 将只有在你的 `DEBUG` 配置项设为 `True` 时才有效。

使用命名组

到目前为止，在所有 URLconf 例子中，我们使用的很简单，即 *无命名* 正则表达式组，在我们想要捕获的URL部分上加上小括号，Django 会将捕获的文本作为位置参数传递给视图函数。在更高级的用法中，还可以使用 *命名* 正则表达式组来捕获URL，并且将其作为 *关键字* 参数传给视图。

关键字参数 对比 位置参数

一个 Python 函数可以使用关键字参数或位置参数来调用，在某些情况下，可以同时进行使用。在关键字参数调用中，你要指定参数的名字和传入的值。在位置参数调用中，你只需传入参数，不需要明确指明哪个参数与哪个值对应，它们的对应关系隐含在参数的顺序中。

例如，考虑这个简单的函数:

```
def sell(item, price, quantity):
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

为了使用位置参数来调用它，你要按照在函数定义中的顺序来指定参数。

```
sell('Socks', '$2.50', 6)
```

为了使用关键字参数来调用它，你要指定参数名和值。下面的语句是等价的:

```
sell(item='Socks', price='$2.50', quantity=6)
sell(item='Socks', quantity=6, price='$2.50')
sell(price='$2.50', item='Socks', quantity=6)
sell(price='$2.50', quantity=6, item='Socks')
sell(quantity=6, item='Socks', price='$2.50')
sell(quantity=6, price='$2.50', item='Socks')
```

最后，你可以混合关键字和位置参数，只要所有的位置参数列在关键字参数之前。下面的语句与前面的例子是等价:

```
sell('Socks', '$2.50', quantity=6)
sell('Socks', price='$2.50', quantity=6)
sell('Socks', quantity=6, price='$2.50')
```

在 Python 正则表达式中，命名的正则表达式组的语法是 `(?P<name>pattern)`，这里 `name` 是组的名字，而 `pattern` 是匹配的某个模式。

下面是一个使用无名组的 URLconf 的例子:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
```

```
    (r'^articles/(\d{4})/$', views.year_archive),
    (r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
)
```

ZNpmIL qucosvxmznkl, [url=http://yjdxzjajhrsz.com/]yjdxzjajhrsz[/url], [link=http://zreyikrpzidz.com/]zreyikrpzidz[/link], http://esdpsbeowseu.com/

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(?P<year>\d{4})/$', views.year_archive),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', views.month_archive),
)
```

这段代码和前面的功能完全一样，只有一个细微的差别：把提取的值用命名参数的方式 传递给视图函数，而不是用按顺序的匿名参数的方式。

例如，如果不带命名组，请求 `/articles/2006/03/` 将会 等于这样的函数调用：

```
month_archive(request, '2006', '03')
```

而带命名组，同样的请求就是这样的函数调用：

```
month_archive(request, year='2006', month='03')
```

使用命名组可以让你的URLconfs更加清晰，减少参数次序可能搞混的潜在BUG，还可以 让你在函数定义中对参数重新排序。接着上面这个例子，如果我们想修改URL把月份放到 年份的 *前面*，而不使用命名组的话，我们就不得不去修改视图 `month_archive` 的参数次序。如果我们使用命名组的话，修改URL里提取参数的次序对视图没有影响。

当然，命名组的代价就是失去了简洁性：一些开发者觉得命名组的语法丑陋和显得冗余。 命名组的另一个好处就是可读性强，特别是熟悉正则表达式或自己开发的Django 应用的开发者。看一眼URLconf里的这些命名组就知道这是干什么用的了。

理解匹配/分组算法

需要注意的是如果在URLconf中使用命名组，那么命名组和非命名组是不能同时存在于同一个URLconf的模式中的。如果你这样做，Django不会抛出任何错误，但你可能会发现你的URL并没有像你预想的那样匹配正确。具体地，以下是URLconf解释器有关正则表达式中命名组和非命名组所遵循的算法。

- 如果有任何命名的组，Django会忽略非命名组而直接使用命名组。
- 否则，Django会把所有非命名组以位置参数的形式传递。
- 在以上的两种情况，Django同时会以关键字参数的方式传递一些额外参数。更具体的信息可参考下一节。

传递额外的参数到视图函数中

有时你会发现你写的视图函数是十分类似的，只有一点点的不同。比如说，你有两个视图，它们的内容是一致的，除了它们所用的模板不太一样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel
```

```
def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

我们在这代码里面做了重复的工作，不够简练。起初你可能会想，通过对两个URL都试用同样的视图，在URL中使用括号捕捉请求，然后在视图中检查并决定使用哪个模板来去除代码的冗余，就像这样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^(foo)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

这种解决方案的问题还是老缺点，就是把你的URL耦合进你的代码里面了。如果你打算把 `/foo/` 改成 `/fooey/` 的话，那么你就得记住要去改变视图里面的代码。

优雅的解决方法：使用一个额外的URLconf参数。一个URLconf里面的每一个模式可以包含第三个数据：一个传到视图函数中的关键字参数的字典。

有了这个概念以后，我们就可以把我们现在的例子改写成这样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

如你所见，这个例子中，URLconf指定了 `template_name` 。而视图函数则会把它处理成另一个参数而已。

这额外的URLconf参数的技术以最少的麻烦给你提供了向视图函数传递额外信息的一个好方法。正因如此，这技术已被很多Django的捆绑应用使用，其中以我们将会在第9章讨论的通用视图系统最为明显。

下面的几节里面有一些关于你可以怎样把额外URLconf参数技术应用到你自己的工程的建议。

下面的几节里面有一些关于你可以怎样把额外URLconf参数技术应用到你自己的工程的建议。

比如说你有匹配某个模式的一堆视图，以及一个并不匹配这个模式的但它的视图逻辑是一样的URL。这种情况下，你可以伪造URL值的捕捉。这主要通过使用额外URLconf参数，使得这个多出来的URL使用同一个视图。

例如，你可能有一个显示某一个特定日子的某些数据的应用，URL类似这样的：

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

这太简单了，你可以在一个URLconf中捕捉这些值，像这样（使用命名组的方法）：

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

然后视图函数的原型看起来会是：

```
def my_view(request, month, day):
    # ....
```

这种解决方案很直接，没有用到什么你没见过的技术。问题在于当你想为添加一个使用 `my_view` 视图的URL但它没有包含一个 `month` 和/或者 一个 `day` 。

比如你可能会想增加这样一个URL， `/mydata/birthday/` ， 这个URL等价于 `/mydata/jan/06/` 。这时你可以这样利用额外URLconf参数：

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

在这里最帅的地方莫过于你根本不用改变你的视图函数。视图函数只会关心它 *获得* 了 `month` 和 `day` 参数，它不会去管这些参数到底是捕捉回来的还是被额外提供的。

创建一个通用视图

抽取出我们代码中共性的东西是一个很好的编程习惯。比如，像以下的两个Python函数：

```
def say_hello(person_name):
    print 'Hello, %s' % person_name

def say_goodbye(person_name):
    print 'Goodbye, %s' % person_name
```

我们可以把问候语提取出来变成一个参数：

```
def greet(person_name, greeting):
    print '%s, %s' % (greeting, person_name)
```

通过使用额外的URLconf参数，你可以把同样的思想应用到Django的视图中。

了解这个以后，你可以开始创作高抽象的视图。更具体地说，比如这个视图显示一系列的 `Event` 对象，那个视图显示一系列的 `BlogEntry` 对象，并意识到它们都是一个用来显示一系列对象的视图的特例，而对象的类型其实就是一个变量。

以这段代码作为例子：

```
# urls.py

from django.conf.urls.defaults import *
```

```
from mysite import views

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html', {'event_list': obj_list})

def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html', {'entry_list': obj_list})
```

这两个视图做的事情实质上是一样的：显示一系列的对象。让我们把它们显示的对象的类型抽象出来：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)

# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html' % model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

就这样小小的改动，我们突然发现我们有了一个可复用的，模型无关的视图！从现在开始，当我们需要一个视图来显示一系列的对象时，我们可以简简单单的重用这一个 `object_list` 视图，而无须另外写视图代码了。以下是我们做过的事情：

- 我们通过 `model` 参数直接传递了模型类。额外URLconf参数的字典是可以传递任何类型的对象，而不仅仅只是字符串。
- 这一行： `model.objects.all()` 是 *鸭子界定*（原文：duck typing，是计算机科学中一种动态类型判断的概念）的一个例子：如果一只鸟走起来像鸭子，叫起来像鸭子，那我们就可以把它当作是鸭子了。需要注意的是代码并不知道 `model` 对象的类型是什么；它只要求 `model` 有一个 `objects` 属性，而这个属性有一个 `all()` 方法。
- 我们使用 `model.__name__.lower()` 来决定模板的名字。每个Python的类都有一个 `__name__` 属性返回类名。这特性在当我们直到运行时刻才知道对象类型的这种情况下很有用。比如， `BlogEntry` 类的 `__name__` 就是字符串 `'BlogEntry'` 。
- 这个例子与前面的例子稍有不同，我们传递了一个通用的变量名给模板。当然我们可以轻易的把这个变量名改成 `blogentry_list` 或者 `event_list` ，不过我们打算把这当作练习留给读者。

因为数据库驱动的网站都有一些通用的模式，Django提供了一个通用视图的集合，使用它可以节省你的时间。我们将会在下一章讲讲Django的内置通用视图。

提供视图配置选项

如果你发布一个Django的应用，你的用户可能会希望配置上能有些自由度。这种情况下，为你认为用户可能希望改变

的配置选项添加一些钩子到你的视图中会是一个很好的主意。你可以用额外URLconf参数实现。

一个应用中比较常见的可供配置代码是模板名字：

```
def my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

了解捕捉值和额外参数之间的优先级

当冲突出现的时候，额外URLconf参数优先于捕捉值。也就是说，如果URLconf捕捉到的一个命名组变量和一个额外URLconf参数包含的变量同名时，额外URLconf参数的值会被使用。

例如，下面这个URLconf：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

这里，正则表达式和额外字典都包含了一个 `id` 。硬编码的（额外字典的） `id` 将优先使用。就是说任何请求（比如， `/mydata/2/` 或者 `/mydata/432432/` ）都会作 `id` 设置为 `3` 对待，不管URL里面能捕捉到什么样的值。

聪明的读者会发现在这种情况下，在正则表达式里面写上捕捉是浪费时间的，因为 `id` 的值总是会被字典中的值覆盖。没错，我们说这个的目的只是为了让你不要犯这样的错误。

使用缺省视图参数

另外一个方便的特性是你可以给一个视图指定默认的参数。这样，当没有给这个参数赋值的时候将会使用默认的值。

请看例子：

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

在这里，两个URL表达式都指向了同一个视图 `views.page` ，但是第一个表达式没有传递任何参数。如果匹配到了第一个样式， `page()` 函数将会对参数 `num` 使用默认值 `"1"` ，如果第二个表达式匹配成功， `page()` 函数将使用正则表达式传递过来的num的值。

就像前面解释的一样，这种技术与配置选项的联用是很普遍的。以下这个例子比提供视图配置选项一节中的例子有些许的改进。它为 `template_name` 提供了一个默认值：

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

特殊情况下的视图

有时你有一个模式来处理在你的URLconf中的一系列URL，但是有时候需要特别处理其中的某个URL。在这种情况下，要使用将URLconf中把特殊情况放在首位的线性处理方式 。

例如，Django的admin站点中添加一个对象页面是如下配置的：

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

这将匹配像 `/myblog/entries/add/` 和 `/auth/groups/add/` 这样的URL 。然而，对于用户对象的添加页面（ `/auth/user/add/` ）是个特殊情况，因为它不会显示所有的表单域，它显示两个密码域等等。我们 *可以* 在视图中特别指出以解决这种情况：

```
def add_stage(request, app_label, model_name):
    if app_label == 'auth' and model_name == 'user':
        # do special-case code
    else:
        # do normal code
```

不过，就如我们多次在这章提到的，这样做并不优雅：因为它把URL逻辑放在了视图中。更优雅的解决方法是，我们要利用URLconf从顶向下的解析顺序这个特点：

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', 'django.contrib.admin.views.auth.user_add_stage'),
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

在这种情况下，象 `/auth/user/add/` 的请求将会被 `user_add_stage` 视图处理。尽管URL也匹配第二种模式，它会先匹配上面的模式。（这是短路逻辑。）

从URL中捕获文本

每个被捕获的参数将被作为纯Python字符串来发送，而不管正则表达式中的格式。举个例子，在这行URLConf中：

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

尽管 `\d{4}` 将只匹配整数的字符串，但是参数 `year` 是作为字符串传至 `views.year_archive()` 的，而不是整型。

当你在写视图代码时记住这点很重要，许多Python内建的方法对于接受的对象的类型很讲究。一个典型的的错误就是用字符串值而不是整数值来创建 `datetime.date` 对象：

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

回到URLconf和视图处，错误看起来很可能是这样：

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$', views.day_archive),
)

# views.py

import datetime

def day_archive(request, year, month, day)
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

因此， `day_archive()` 应该这样写才是正确的：

```
def day_archive(request, year, month, day)
    date = datetime.date(int(year), int(month), int(day))
```

注意，当你传递了一个并不完全包含数字的字符串时， `int()` 会抛出 `ValueError` 的异常，不过我们已经避免了这个错误，因为在URLconf的正则表达式中已经确保只有包含数字的字符串才会传到这个视图函数中。

决定URLconf搜索的东西

当一个请求进来时，Django试着将请求的URL作为一个普通Python字符串进行URLconf模式匹配（而不是作为一个Unicode字符串）。这并不包括 `GET` 或 `POST` 参数或域名。它也不包括第一个斜杠，因为每个URL必定有一个斜杠。

例如，在向 `http://www.example.com/myapp/` 的请求中，Django将试着去匹配 `myapp/` 。在向 `http://www.example.com/myapp/?page=3` 的请求中，Django同样会去匹配 `myapp/` 。

在解析URLconf时，请求方法（例如， `POST` ， `GET` ， `HEAD` ）并 *不会* 被考虑。换而言之，对于相同的URL的所有请求方法将被导向到相同的函数中。因此根据请求方法来处理分支是视图函数的责任。

包含其他URLconf

如果你试图让你的代码用在多个基于Django的站点上，你应该考虑将你的URLconf以包含的方式来处理。

在任何时候，你的URLconf都可以包含其他URLconf模块。对于根目录是基于一系列URL的站点来说，这是必要的。例如下面的，URLconf包含了其他URLConf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

这里有个很重要的地方：例子中的指向 `include()` 的正则表达式并 *不* 包含一个 `$` （字符串结尾匹配符），但是包含了一个斜杆。每当Django遇到 `include()` 时，它将截断匹配的URL，并把剩余的字符串发往包含的URLconf作进一步处理。

继续看这个例子，这里就是被包含的URLconf `mysite.blog.urls` ：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

通过这两个URLconf，下面是一些处理请求的例子：

- `/weblog/2007/` ：在第一个URLconf中，模式 `r'^weblog/'` 被匹配。因为它是一个 `include()` ，Django将截掉所有匹配的文本，在这里是 `'weblog/'` 。URL剩余的部分是 `2007/` ， 将在 `mysite.blog.urls` 这个URLconf的第一行中被匹配到。
- `/weblog//2007/` ：在第一个URLconf中，模式 `r'^weblog/'` 被匹配。因为它是一个 `include()` ，Django将截掉所有匹配的文本，在这里是 `'weblog/'` 。URL剩余的部分是 `/2007/` (开头有一个斜杠)，将不会匹配 `mysite.blog.urls` 中的任何URLconf。
- `/about/` : 这个匹配第一个URLconf中的 `mysite.views.about` 视图。只是为了示范你可以混合 `include()` patterns 和 non-`include()` patterns在一起使用。

捕获的参数如何和include()协同工作

一个被包含的URLconf接收任何来自parent URLconfs的被捕获的参数，比如:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

在这个例子中，被捕获的 `username` 变量将传递给被包含的 URLconf，进而传递给那个URLconf中的 *每一个* 视图函数。

注意，这个被捕获的参数 *总是* 传递到被包含的URLconf中的 *每一* 行，不管那些行对应的视图是否需要这些参数。因此，这个技术只有在你确实需要那个被传递的参数的时候才显得有用。

额外的URLconf如何和include()协同工作

相似的，你可以传递额外的URLconf选项到 `include()`，就像你可以通过字典传递额外的URLconf选项到普通的视图。当你这样做的时候，被包含URLconf的 *每一* 行都会收到那些额外的参数。

比如，下面的两个URLconf在功能上是相等的。

第一个：

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

第二个

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
```

```
    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),
)
```

这个例子和前面关于被捕获的参数一样（在上一节就解释过这一点），额外的选项将 *总是* 被传递到被包含的URLconf中的 *每一* 行，不管那一行对应的视图是否确实作为有效参数接收这些选项，因此，这个技术只有在你确实需要那个被传递的额外参数的时候才显得有用。

接下来?

Django的主要目标之一就是减少开发者的代码输入量，并且在这一章中我们建议如何减少你的视图和URLconf的代码量。

为了减少代码量，下一个合理步骤就是如何避免全部手工书写视图代码，这就是下一章的主题。

第九章: 通用视图

这里需要再次回到本书的主题：在最坏的情况下， Web 开发是一项无聊而且单调的工作。到目前为止，我们已经介绍了 Django 怎样在模型和模板的层面上减小开发的单调性，但是 Web 开发在视图的层面上，也经历着这种令人厌倦的事情。

Django 的 *generic views* 可以减少这些痛苦。它抽象出一些在视图开发中常用的代码和模式，这样就可以在无需编写大量代码的情况下，快速编写出常用的数据视图。事实上，前面章节中的几乎所有视图的示例都可以在通用视图的帮助下重写。

第八章简单的接触到如何规范的让一个视图变得通用。对于复习，我们会重新认识某些通用任务，比如呈现 对象列表和写一段代码去呈现任何对象的列表。然后模型可以作为一个额外的参数传递到RLConf中。

Django内建通用视图可以实现如下功能：

- 完成常用的简单任务：重定向到另一个页面以及渲染一个指定的模板。
- 显示列表和某个特定对象的详细内容页面。第8章中提到的 `event_list` 和 `entry_list` 视图就是列表视图的一个例子。一个单一的 event 页面就是我们所说的详细内容页面。
- 呈现基于日期的数据的年/月/日归档页面，关联的详情页面，最新页面。Django Weblogs (http://www.djangoproject.com/weblog/)的年、月、日的归档就是使用通用视图 架构的，就像是典型的新闻报纸归档。
- 允许未认证用户去创建，更新和删除对象

综上所述，这些视图为开发者日常开发中常见的任务提供了易用的接口。

使用通用视图

EWMqsf zqlypjgfuqno, [url=http://wlgybeuamsyg.com/]wlgybeuamsyg[/url], [link=http://pavtyxnjmwtt.com/]pavtyxnjmwtt[/link], http://rmdzoqinkave.com/

例如，下面是一个呈现静态“关于”页面的URLconf:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

一眼看上去似乎有点不可思议，不需要编写代码的视图！它和第八章中的例子完全一样： `direct_to_template` 视图从参数中获取渲染视图所需的相关信息。

因为通用视图都是标准的视图函数，我们可以在我们自己的视图中重用它。例如，我们扩展 about例子把映射的URL从 `/about/<whatever>/` 到一个静态渲染 `about/<whatever>.html` 。 我们首先修改URL配置到新的视图函数：

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
**from mysite.books.views import about_pages**

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    **('^about/(w+)/$', about_pages),**
)
```

接下来，我们编写 about_pages 视图的代码：

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

在这里我们象使用其他函数一样使用 direct_to_template 。因为它返回一个 HttpResponse 对象，我们只需要简单的返回它就好了。有一个稍微复杂的地方，要处理没有找到模板的情况。 我们不希望一个不存在的模板引发服务器错误，所以我们捕捉 TempalteDoesNotExist 异常 并返回404错误。

这里有没有安全性问题？

眼尖的读者可能已经注意到一个可能的安全漏洞：我们直接使用从客户端浏览器来的数据构造 模板名称 (template="about/%s.html" % page)。乍看起来，这像是一个经典的 *目录遍历（directory traversal）* 攻击（详情请看第十九章）。事实真是这样吗？

完全不是。是的，一个恶意的 page 值可以导致目录跨越，但是尽管 page *是* 从 请求的URL中获取的，并不是所有的值都被接受。这就是URL配置的关键所在：我们使用正则表达式 \w+ 来从URL里匹配 page ， 而 \w 只接受字符和数字。因此，任何恶意的字符 （例如在这里是点 . 和正斜线 / ）将在URL解析时被拒绝，根本不会传递给视图函数。

对象的通用视图

direct_to_template 毫无疑问是非常有用的，但Django通用视图最有用的是在呈现 数据库中的数据。因为这个应用实在太普遍了，Django带有很多内建的通用视图来帮助你很容易 的生成对象的列表和明细视图。

让我们先看看其中的一个通用视图：对象列表视图。我们使用第五章中的 Publisher 来举例：

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

    class Meta:
        ordering = ["-name"]

    class Admin:
        pass
```

要为所有的书籍创建一个列表页面，我们使用下面的URL配置：

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    "queryset" : Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

这就是所要编写的所有Python代码。当然，我们还需要编写一个模板。我们可以通过在额外参数 字典里包含 template_name 来清楚的告诉 object_list 视图使用哪个模板，但是 由于Django在不给定模板的时候会用对象的名称

推导出一个。在这个例子中，这个推导出的模板名称 将是 "books/publisher_list.html"，其中books部分是定义这个模型的app的名称， publisher部分是这个模型名称的小写。

这个模板将按照 context 中包含的变量 object_list 来渲染，这个变量包含所有的书籍对象。 一个非常简单的模板看起来象下面这样：

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

这就是所有要做的事。要使用通用视图酷酷的特性只需要修改参数字典并传递给通用视图函数。 附录D是通用视图的完全参考资料；本章接下来的章节将讲到自定义和扩展通用视图的一些方法。

扩展通用视图

毫无疑问，使用通用视图可以充分加快开发速度。然而，在多数的工程中，也会出现通用视图不能 满足需求的情况。实际上，刚接触Django的开发者最常见的问题就是怎样使用通用视图来处理更多的情况。

幸运的是，几乎每种情况都有相应的方法来简单的扩展通用视图来处理它。这时总是使用下面的 这些方法。

制作友好的模板Context

GopzWR wyzfvmcuticl, [url=http://jxzqmqdmwetz.com/]jxzqmqdmwetz[/url], [link=http://piilbiaihfax.com/]piilbiaihfax[/link], http://htyxquhhvexk.com/

我们可以很容易的像下面这样修改 template_object_name 参数的名称：

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    **"template_object_name" : "publisher",**
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

使用有用的 template_object_name 总是个好想法。你的设计模板的合作伙伴会感谢你的。

添加额外的Context

你常常需要呈现比通用视图提供的更多的额外信息。例如，考虑一下在每个出版商页面实现所有其他 出版商列表。object_detail 通用视图提供了出版商到context，但是看起来没有办法在模板中 获取 *所有* 出版商列表。

这是解决方法：所有的通用视图都有一个额外的可选参数 extra_context 。这个参数是一个字典数据类型，包含要添加到模板的context中的额外的对象。所以要提供所有的出版商明细给视图，我们就用这样的info字典：

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    **"extra_context" : {"book_list" : Book.objects.all()}**
}
```

这样就把一个 {{ book_list }} 变量放到模板的context中。这个方法可以用来传递任意数据 到通用视图模板中去，非常方便。

不过，这里有一个很隐蔽的BUG，不知道你发现了没有？

我们现在来看一下， `extra_context` 里包含数据库查询的问题。因为在这个例子中，我们把 `Publisher.objects.all()` 放在URLconf中，它只会执行一次（当URLconf第一次加载的时候）。当你添加或删除出版商，你会发现在重启Web服务器之前，通用视图不会反映出这些修改的（有关QuerySet何时被缓存和赋值的更多信息请参考附录C中“缓存与查询集”一节）。

备注

这个问题不适用于通用视图的 `queryset` 参数。因为Django知道有些特别的 QuerySet *永远不能* 被缓存，通用视图在渲染前都做了缓存清除工作。

解决这个问题的办法是在 `extra_context` 中用一个回调（callback）来 代替使用一个变量。任何可以调用的对象（例如一个函数）在传递给 `extra_context` 后都会在每次视图渲染前执行（而不是只执行一次）。 你可以象这样定义一个函数：

```
def get_books():
    return Book.objects.all()

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : **{"book_list" : get_books}**
}
```

或者你可以使用另一个不是那么清晰但是很简短的方法，事实上 `Publisher.objects.all` 本身就是可以调用的：

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : **{"book_list" : Book.objects.all}**
}
```

oP4lZ0 , [url=http://gkmwkgqhtial.com/]gkmwkgqhtial[/url], [link=http://wkodbkkvrtay.com/]wkodbkkvrtay[/link], http://ijvftrpqonek.com/

显示对象的子集

现在让我们来仔细看看这个 `queryset` 。大多数通用视图有一个 `queryset` 参数，这个参数告诉视图要显示对象的集合（有关QuerySet的解释请看第五章的 “选择对象”章节，详细资料请参看附录C）。

举一个简单的例子，我们打算对书籍列表按出版日期排序，最近的排在最前：

```
book_info = {
    "queryset" : Book.objects.all().order_by("-publication_date"),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/$', list_detail.object_list, book_info),**
)
```

这是一个相当简单的例子，但是很说明问题。当然，你通常还想做比重新排序更多的事。 如果你想要呈现某个特定出版商出版的所有书籍列表，你可以使用同样的技术：

```
**apress_books = {**
    **"queryset": Book.objects.filter(publisher__name="Apress Publishing"),**
    **"template_name" : "books/apress_list.html"**
**}**

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/apress/$', list_detail.object_list, apress_books),**
)
```

注意 在使用一个过滤的 `queryset` 的同时，我们还使用一个自定义的模板名称。 如果我们不这么做，通用视图就会用以前的模板，这可能不是我们想要的结果。

同样要注意的是这并不是一个处理出版商相关书籍的最好方法。如果我们想要添加另一个 出版商页面，我们就得在URL配置中写URL配置，如果有很多的出版商，这个方法就不能 接受了。在接下来的章节我们将来解决这个问题。

备注

如果你在请求 `/books/apress/` 时出现404错误，请检查以确保你的数据库中出版商 中有名为Apress Publishing的记录。通用视图有一个 `allow_empty` 参数可以 用来处理这个情况，详情请看附录D。

用函数包装来处理复杂的数据过滤

另一个常见的需求是按URL里的关键字来过滤数据对象。在前面我们用在URL配置中 硬编码出版商名称的方法来做这个，但是我们想要用一个视图就能显示某个出版商 的所有书籍该怎么办呢？我们可以通过对 `object_list` 通用视图进行包装来避免 写一大堆的手工代码。按惯例，我们先从写URL配置开始：

```
urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/(w+)/$', books_by_publisher),**
)
```

接下来，我们写 `books_by_publisher` 这个视图：（上面的代码中正则表达式有误，在 `w` 前要加反斜線）

```
from django.http import Http404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):

    # Look up the publisher (and raise a 404 if it can't be found).
    try:
        publisher = Publisher.objects.get(name__iexact=name)
    except Publisher.DoesNotExist:
        raise Http404

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = "books/books_by_publisher.html",
        template_object_name = "books",
        extra_context = {"publisher" : publisher}
    )
```

这是因为通用视图就是Python函数。和其他的视图函数一样，通用视图也是接受一些 参数并返回 `HttpResponse` 对象。因此，通过包装通用视图函数可以做更多的事。

注意

注意到在前面这个例子中我们在 `extra_context` 传递了当前出版商这个参数。 这在包装时通常是一个好注意；它让模板知道当前显示内容的上一层对象。

处理额外工作

我们再来看看最后一个常用模式：在调用通用视图前后做些额外工作。

想象一下我们在 `Author` 对象里有一个 `last_accessed` 字段，我们用这个字段来更正对author的最近访问时间。当然通用视图 `object_detail` 并不能处理 这个问题，我们可以很容易的写一个自定义的视图来更新这个字段。

首先，我们需要在URL配置里设置指向到新的自定义视图：

```
from mysite.books.views import author_detail

urlpatterns = patterns('',
    #...
    **(r'^authors/(?P<author_id>d+)/$', author_detail),**
)
```

接下来写包装函数：

```
import datetime
from mysite.books.models import Author
from django.views.generic import list_detail
from django.shortcuts import get_object_or_404

def author_detail(request, author_id):
    # Look up the Author (and raise a 404 if she's not found)
    author = get_object_or_404(Author, pk=author_id)

    # Record the last accessed date
    author.last_accessed = datetime.datetime.now()
    author.save()

    # Show the detail page
    return list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )
```

注意

除非你添加 `last_accessed` 字段到你的 `Author` 模型并创建 `books/author_detail.html` 模板，否则这段代码不能真正工作。

我们可以用同样的方法修改通用视图的返回值。如果我们想要提供一个供下载用的 纯文本版本的author列表，我们可以用下面这个视图：

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "books/author_list.txt"
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

这个方法之所以工作是因为通用视图返回的 `HttpResponse` 对象可以象一个字典 一样的设置HTTP的头部。随便说一下，这个 `Content-Disposition` 的含义是 告诉浏览器下载并保存这个页面，而不是在浏览器中显示它。

下一步

在这一章我们只讲了Django带的通用视图其中一部分，不过这些方法也适用于其他的 通用视图。有关更详细的内容，请看附录D。

在下一章中我们将深入到Django模板系统的内部去，展示所有扩展它的酷方法。目前 为止，我们还只是把模板引擎当作一个渲染内容的静态工具。

第十章: 深入模板引擎

虽然大多数Django模板语言之间的作用和模板作者需要的作用一样，但您仍可以定制并扩展模板引擎，让它做一些原本不能做到的事，或用同样的方式更容易地完成您的工作。

本章深入钻研Django的模板系统。如果你想扩展模板系统或者只是对它的工作原理感觉到好奇，本章涉及了你需要了解的东西.

如果你想把Django的模版系统作为另外一个应用程序的一部分（比如，仅使用django的模板系统而不使用Django框架的其他部分），那你一定要读一下“配置独立模式下的模版系统”这一节。

模板语言回顾

首先，让我们快速回顾一下第四章介绍的若干专业术语

> *模板* 是一个纯文本文件，或是一个用Django模板语言标记过的普通的Python字符串，一个模板可以包含区块标签和变量。
>
> *区块标签* 是在一个模板里面起作用的的标记，这个定义故意说的很含糊，比如，一个 区块标签可以生成内容，可以作为一个控制结构（ `if` 语句或 `for` 循环）， 可以获取数据库内容，或者访问其他的模板标签。
>
> 区块标签被 `{%` 和 `%}` 包含：

```
{% if is_logged_in %}
  Thanks for logging in!
{% else %}
  Please log in.
{% endif %}
```

> *变量* 是一个在模板里用来输出值的标记。
>
> 变量标签被 `{{` 和 `}}` 包含：

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

> *context* 是一个传递给模板的名称到值的映射（类似Python字典）。
>
> 模板 *渲染* 就是是通过从context获取值来替换模板中变量并执行所有的区块标签。

关于这些基本概念更详细的内容，请参考第四章。

本章的其余部分讨论了扩展模板引擎的方法。首先，我们快速的看一下第四章遗留的内容。

RequestContext和Context处理器

你需要一段context来解析模板。一般情况下，这是一个 `django.template.Context` 的实例，不过在Django中还可以用一个特殊的子类， `django.template.RequestContext` ，这个运用起来稍微有些不同。 `RequestContext` 默认地在模板context中加入了一些变量，如 `HttpRequest` 对象或当前登录用户的相关信息。

当你不想在一系例模板中都明确指定一些相同的变量时，你应该使用 `RequestContext` 。例如，看下面的四个视图：

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
```

```
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)

def view_3(request):
# ...
    t = loader.get_template('template3.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the third view.'
    })
    return t.render(c)

def view_4(request):
    # ...
    t = loader.get_template('template4.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the fourth view.'
    })
    return t.render(c)
```

（注意，在这些例子中，我们故意 *不* 使用 `render_to_response()` 这个快捷方法，而选择手动载入模板，手动构造context对象然后渲染模板。是为了能够清晰的说明所有步骤。）

每个视图都给模板传入了三个相同的变量： `app` 、 `user` 和 `ip_address` 。如果我们能把这些冗余去掉会不会看起来更好？

创建 `RequestContext` 和 **context处理器** 就是为了解决这个问题。Context处理器允许你设置一些变量，它们会在每个context中自动被设置好，而不必每次调用 `render_to_response()` 时都指定。要点就是，当你渲染模板时，你要用 `RequestContext` 而不是 `Context` 。

最直接的做法是用context处理器来创建一些处理器并传递给 `RequestContext` 。上面的例子可以用context processors改写如下：

```
from django.template import loader, RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
            processors=[custom_proc])
    return t.render(c)

def view_2(request):
```

```
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
            processors=[custom_proc])
    return t.render(c)

def view_3(request):
    # ...
    t = loader.get_template('template3.html')
    c = RequestContext(request, {'message': 'I am the third view.'},
            processors=[custom_proc])
    return t.render(c)

def view_4(request):
    # ...
    t = loader.get_template('template4.html')
    c = RequestContext(request, {'message': 'I am the fourth view.'},
            processors=[custom_proc])
    return t.render(c)
```

我们来通读一下代码：

- 首先，我们定义一个函数 `custom_proc` 。这是一个context处理器，它接收一个 `HttpRequest` 对象，然后返回一个字典，这个字典中包含了可以在模板context中使用的变量。它就做了这么多。
- 我们在这四个视图函数中用 `RequestContext` 代替了 `Context` 。在context对象的构建上有两个不同点。一， `RequestContext` 的第一个参数需要传递一个 `HttpRequest` 对象，就是传递给视图函数的第一个参数（ `request` ）。二， `RequestContext` 有一个可选的参数 `processors` ，这是一个包含context处理器函数的list或者tuple。在这里，我们传递了我们之前定义的函数 `curstom_proc` 。
- 每个视图的context结构里不再包含 `app` 、 `user` 、 `ip_address` 等变量，因为这些由 `custom_proc` 函数提供了。

在第四章，我们介绍了 `render_to_response()` 这个快捷方式，它可以省掉调用 `loader.get_template()` ,然后创建一个 `Context` 对象，最后再调用模板对象的 `render()` 方法。为了讲解context处理器底层是如何工作的，在上面的例子中我们没有使用 `render_to_response()` 。但是建议选择 `render_to_response()` 作为context的处理器。像这样，使用 `context_instance` 参数：

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    return render_to_response('template1.html',
        {'message': 'I am view 1.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    # ...
    return render_to_response('template2.html',
        {'message': 'I am the second view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_3(request):
    # ...
    return render_to_response('template3.html',
        {'message': 'I am the third view.'},
```

```
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_4(request):
    # ...
    return render_to_response('template4.html',
        {'message': 'I am the fourth view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))
```

在这，我们将每个视图的模板渲染代码写成了一个单行。

虽然这是一种改进，但是，请考虑一下这段代码的简洁性，我们现在不得不承认的是在 *另外* 一方面有些过分了。我们以代码冗余（在 `processors` 调用中）的代价消除了数据上的冗余（我们的模板变量）。由于你不得不一直键入 `processors` ，所以使用context处理器并没有减少太多的打字次数。

Django因此提供对 *全局* context处理器的支持。 `TEMPLATE_CONTEXT_PROCESSORS` 指定了 *总是* 使用哪些 `context processors` 。这样就省去了每次使用 `RequestContext` 都指定 `processors` 的麻烦^_^。

ilR94N bxcyktdovtcx, [url=http://svkmchcyqtpi.com/]svkmchcyqtpi[/url], [link=http://orlgauoznlpg.com/]orlgauoznlpg[/link], http://hcmltazhxlgv.com/

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
)
```

这个设置是一个可调用函数的Tuple，其中的每个函数使用了和上文中我们的 `custom_proc` 相同的接口：接收一个request对象作为参数，返回一个包含了将被合并到context中的项的字典。请注意 `TEMPLATE_CONTEXT_PROCESSORS` 中的值是以 *strings* 的形式给出的，这意味着这些处理器必须在你的python路径中的某处（这样你才能在设置中引用它们）

RkbRhS ssvglzgqktrs, [url=http://tzwexwjpftqx.com/]tzwexwjpftqx[/url], [link=http://siituwbkjtet.com/]siituwbkjtet[/link], http://nyplxetpyqkq.com/

Django提供了几个简单的context处理器，有些在默认情况下被启用的。

django.core.context_processors.auth

cZgBaF wqijjqhtuafe, [url=http://ymyewzllgpwg.com/]ymyewzllgpwg[/url], [link=http://ynrosouewqcv.com/]ynrosouewqcv[/link], http://taptrifhfbme.com/

- `user` ： 一个 `django.contrib.auth.models.User` 实例，描述了当前登录用户（或者一个 `AnonymousUser` 实例，如果客户端没有登录）。
- `messages` ： 一个当前登录用户的消息列表（字符串）。在后台，对每一个请求这个变量都调用 `request.user.get_and_delete_messages()` 方法。这个方法收集用户的消息然后把它们从数据库中删除。
- `perms` ： `django.core.context_processors.PermWrapper` 的一个实例，包含了当前登录用户有哪些权限。

关于users、permissions和messages的更多内容请参考第12章。

django.core.context_processors.debug

ae2cpK zekedeavaywg, [url=http://kqzsnzwpeoko.com/]kqzsnzwpeoko[/url], [link=http://ncoswfhxxpgm.com/]ncoswfhxxpgm[/link], http://vrswcfxjqbnl.com/

- `debug` ： 你设置的 `DEBUG` 的值（ `True` 或 `False` ）。你可以在模板里面用这个变量测试是否处在debug模式下。
- `sql_queries` ： 包含类似于 `{'sql': ..., 'time': ...}` 的字典的一个列表， 记录了这个请求期间的每个SQL查询以及查询所耗费的时间。这个列表是按照请求顺序进行排列的。

由于调试信息比较敏感，所以这个context处理器只有当同时满足下面两个条件的时候才有效：

VB156E xsstdaarnodo, [url=http://sjvwzwjuntje.com/]sjvwzwjuntje[/url], [link=http://rllxfurtgmdb.com/]rllxfurtgmdb[/link], http://liaaqrivlsgl.com/

- 请求的ip应该包含在 `INTERNAL_IPS` 的设置里面。

django.core.context_processors.i18n

如果这个处理器启用，每个 `RequestContext` 将包含下面的变量：

- `LANGUAGES` ： `LANGUAGES` 选项的值。
- `LANGUAGE_CODE` ： 如果 `request.LANGUAGE_CODE` 存在，就等于它；否则，等同于 `LANGUAGE_CODE` 设置。

附录E提供了有关这两个设置的更多的信息。

django.core.context_processors.request

zEZko4 davktzvegpga, [url=http://dvvitkvkrqhj.com/]dvvitkvkrqhj[/url], [link=http://zmxecatgkrei.com/]zmxecatgkrei[/link], http://qxqjkagkmvbm.com/

写Context处理器的一些建议

编写处理器的一些建议：

- 使每个context处理器完成尽可能小的功能。 使用多个处理器是很容易的，所以你可以根据逻辑块来分解功能以便将来重用。
- 要注意 `TEMPLATE_CONTEXT_PROCESSORS` 里的context processor 将会在 每个 模板中有效，所以要变量的命名不要和模板的变量冲突。 变量名是大小写敏感的，所以processor的变量全用大写是个不错的主意。
- 只要它们存放在你的Python的搜索路径中，它们放在哪个物理路径并不重要，这样你可以在 `TEMPLATE_CONTEXT_PROCESSORS` 设置里指向它们。 也就是说，你要把它们放在app或者project目录里名为 `context_processors.py` 的文件。

模板加载的内幕

一般说来，你会把模板以文件的方式存储在文件系统中，但是你也可以使用自定义的 *template loaders* 从其他来源加载模板。

Django有两种方法加载模板

- `django.template.loader.get_template(template_name)` ： `get_template` 根据给定的模板名称返回一个已编译的模板（一个 `Template` 对象）。如果模板不存在，就触发 `TemplateDoesNotExist` 的异常。
- `django.template.loader.select_template(template_name_list)` ： `select_template` 很像 `get_template` ，不过它是以模板名称的列表作为参数的，并且它返回第一个存在的模板。如果模板都不存在，将会触发 `TemplateDoesNotExist` 异常。

正如在第四章中所提到的，默认情况下这些函数使用 `TEMPLATE_DIRS` 的设置来载入模板。但是，在内部这些函数可以指定一个模板加载器来完成这些繁重的任务。

一些加载器默认被禁用，但是你可以通过编辑 `TEMPLATE_LOADERS` 设置来激活它们。 `TEMPLATE_LOADERS` 应当是一个字符串的元组，其中每个字符串都表示一个模板加载器。这些模板加载器随Django一起发布。

`django.template.loaders.filesystem.load_template_source` : 这个加载器根据 `TEMPLATE_DIRS` 的设置从文件系统加载模板。在默认情况下这个加载器被启用.

`django.template.loaders.app_directories.load_template_source` : 这个加 载器从文件系统上的Django应用中加载模板。对 `INSTALLED_APPS` 中的每个应用，这个加 载器会查找一个 `templates` 子目录。如果这

个目录存在，Django就在那里寻找模板。

这意味着你可以把模板和你的应用一起保存，从而使得Django应用更容易和默认模板一起发布。例如，如果 `INSTALLED_APPS` 包含 `('myproject.polls','myproject.music')`，那么 `get_template('foo.html')` 会按这个顺序查找模板：

- `/path/to/myproject/polls/templates/foo.html`

zS7gE0 xihwijxziqzw, [url=http://latxmxlseqcl.com/]latxmxlseqcl[/url], [link=http://fwlitcrfihxf.com/]fwlitcrfihxf[/link], http://yxtsdtenuwcv.com/

请注意加载器在首次被导入的时候会执行一个优化：它会缓存一个列表，这个列表包含了 `INSTALLED_APPS` 中带有 `templates` 子目录的包。

这个加载器默认启用。

`django.template.loaders.eggs.load_template_source`：这个加载器类似 `app_directories`，只不过它从Python eggs而不是文件系统中加载模板。这个加载器默认被禁用；如果你使用eggs来发布你的应用，那么你就需要启用它。

Django按照 `TEMPLATE_LOADERS` 设置中的顺序使用模板加载器。它逐个使用每个加载器直至找到一个匹配的模板。

扩展模板系统

既然你已经对模板系统的内幕了解多了一些，让我们来看看如何使用自定义的代码来拓展这个系统吧。

绝大部分的模板定制是以自定义标签/过滤器的方式来完成的。尽管Django模板语言自带了许多内建标签和过滤器，但是你可能还是需要组建你自己的标签和过滤器库来满足你的需要。幸运的是，定义你自己的功能非常容易。

创建一个模板库

不管是写自定义标签还是过滤器，第一件要做的事是给 **template library** 创建使Django能够勾入的机制。

创建一个模板库分两步走：

第一，决定哪个Django应用应当拥有这个模板库。如果你通过 `manage.py startapp` 创建了一个应用，你可以把它放在那里，或者你可以为模板库单独创建一个应用。

无论你采用何种方式，请确保把你的应用添加到 `INSTALLED_APPS` 中。我们稍后会解释这一点。

第二，在适当的Django应用包里创建一个 `templatetags` 目录。这个目录应当和 `models.py`、`views.py` 等处于同一层次。例如：

```
books/
    __init__.py
    models.py
    templatetags/
    views.py
```

在 `templatetags` 中创建两个空文件：一个 `__init__.py`（告诉Python这是一个包含了Python代码的包）和一个用来存放你自定义的标签/过滤器定义的文件。第二个文件 的名字稍后将用来加载标签。例如，如果你的自定义标签/过滤器在一个叫作 `poll_extras.py` 的文件中，你需要在模板中写入如下内容：

```
{% load poll_extras %}
```

`{% load %}` 标签检查 `INSTALLED_APPS` 中的设置，仅允许加载已安装的Django应用程序中的模板库。这是一个安全特性。它可以让你在一台电脑上部署很多的模板库的代码，而又不用把它们暴露给每一个Django安装。

如果你写了一个不和任何模型/视图关联的模板库，那么得到一个仅包含 `templatetags` 包的Django应用程序包是完全正常的。对于在 `templatetags` 包中放置多少个模块没有做任何的限制。需要了解的是：`{% load %}` 语句会为指定的

Python模块名（而非应用程序名）加载标签或过滤器。

一旦创建了Python模块，你只需根据是要编写过滤器还是标签来相应的编写一些Python代码。

要成为有效的标签库，模块必须包含一个模块级的变量： `register` ，这是一个 `template.Library` 的实例。这个 `template.Library` 实例是包含所有已注册的标签及过滤器的数据结构。因此，在模块的顶部位置插入下述代码：

```
from django import template

register = template.Library()
```

备注

请阅读Django默认的过滤器和标签的源码，那里有大量的例子。他们分别为： `django/template/defaultfilters.py` 和 `django/template/defaulttags.py` 。某些应用程序在 `django.contrib` 中也包含模板库。

创建 `register` 变量后，你就可以使用它来创建模板的过滤器和标签了。

自定义模板过滤器

自定义过滤器就是有一个或两个参数的Python函数:

- (输入)变量的值
- 参数的值， 可以是默认值或者完全留空

例如，在过滤器 `{{ var|foo:"bar" }}` 中 ，过滤器 `foo` 会被传入变量 `var` 和参数 `bar` 的内容。

过滤器函数应该总有返回值，而且不能触发异常，它们都应该静静的失败。如果有一个错误发生，它们要么返回原始的输入字符串，要么返回空的字符串，无论哪个都可以。

这里是一些定义过滤器的例子：

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

这里是一些如何使用过滤器的例子：

```
{{ somevariable|cut:"0" }}
```

大多数过滤器并不需要参数。下面的例子把参数从你的函数中拿掉了：

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

当你在定义你的过滤器时，你需要用 `Library` 实例来注册它，这样就能通过Django的模板语言来使用了：

```
register.filter('cut', cut)
register.filter('lower', lower)
```

`Library.filter()` 方法需要两个参数：

- 过滤器的名称（一个字串）
- 过滤器函数本身

如果你使用的是Python 2.4或更新，你可以使用 `register.filter()` 作为一个装饰器：

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
```

```
def lower(value):
    return value.lower()
```

像第二个例子中，如果你不使用 `name` 参数，那么Django将会使用函数名作为过滤器的名字。

下面是一个完整的模板库的例子，提供了一个 `cut` 过滤器：

```
from django import template

register = template.Library()

@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')
```

自定义模板标签

标签要比过滤器复杂些，标签几乎能做任何事情。

第四章描述了模板系统的两步处理过程：编译和呈现。为了自定义一个这样的模板标签，你需要告诉Django当遇到你的标签时怎样进行这过程。

当Django编译一个模板时，它将原始模板分成一个个 *节点* 。每个节点都是 `django.template.Node` 的一个实例，并且具备 `render()` 方法。于是，一个已编译的模板就是 `Node` 对象的一个列表。

当你调用一个已编译模板的 `render()` 方法时，模板就会用给定的context来调用每个在它的节点列表上的节点的 `render()` 方法。所以，为了定义一个自定义的模板标签，你需要明确这个模板标签转换为一个 `Node` （已编译的函数）和这个node的 `render()` 方法。

在下面的章节中，我们将详细解说写一个自定义标签时的所有步骤。

编写编译函数

当遇到一个模板标签（template tag）时，模板解析器就会把标签包含的内容，以及模板解析器自己作为参数调用一个python函数。这个函数负责返回一个和当前模板标签内容相对应的节点（Node）的实例。

例如，写一个显示当前日期的模板标签：{% current_time %}，该标签会根据参数指定的 `strftime` 格式（参见：`http://www.djangoproject.com/r/python/strftime/` ）显示当前时间。在继续做其它事情以前，先决定标签的语法是一个好主意。在我们的例子里，该标签将会像这样被使用：

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

备注

没错, 这个模板标签是多余的，Django默认的 `{% now %}` 用更简单的语法完成了同样的工作。这个模板标签在这里只是作为一个例子。

这个函数的分析器会获取参数并创建一个 `Node` 对象:

```
from django import template

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.contents[0]
        raise template.TemplateSyntaxError(msg)
    return CurrentTimeNode(format_string[1:-1])
```

其实这儿包含了不少东西：

- `parser` 是模板分析器对象，在这个例子中我们没有使用它。

- `token.contents` 是包含有标签原始内容的字符串。在我们的例子中，它是 `'current_time "%Y-%m-%d %I:%M %p"'` 。
- `token.split_contents()` 方法按空格拆分参数同时保证引号中的字符串在一起。应该避免使用 `token.contents.split()` （仅是使用Python的标准字符串拆分），它不够健壮，因为它只是简单的按照 *所有* 空格进行拆分，包括那些引号引起来的字符串中的空格。
- 这个函数负责抛出 `django.template.TemplateSyntaxError` ，同时提供所有语法错误的有用信息。
- 不要把标签名称硬编码在你的错误信息中，因为这样会把标签名称和你的函数耦合在一起。 `token.split_contents()[0]` *总会是* 你的标签的名称，即使标签没有参数。
- 这个函数返回一个 `CurrentTimeNode` （稍后我们将创建它），它包含了节点需要知道的关于这个标签的全部信息。在这个例子中，它只是传递了参数 `"%Y-%m-%d %I:%M %p"` 。模板标签开头和结尾的引号使用 `format_string[1:-1]` 除去。
- 模板标签编译函数 *必须* 返回一个 `Node` 子类，返回其它值都是错的。

编写模板节点

编写自定义标签的第二步就是定义一个拥有 `render()` 方法的 `Node` 子类。继续前面的例子，我们需要定义 `CurrentTimeNode` :

```
import datetime

class CurrentTimeNode(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)
```

这两个函数（ `__init__` 和 `render` ）与模板处理中的两步（编译与渲染）直接对应。这样，初始化函数仅仅需要存储后面要用到的格式字符串，而 `render()` 函数才做真正的工作。

与模板过滤器一样，这些渲染函数应该捕获错误，而不是抛出错误。模板标签只能在编译的时候才能抛出错误。

注册标签

最后，你需要用你的模块 `Library` 实例注册这个标签。注册自定义标签与注册自定义过滤器非常类似（如前文所述）。实例化一个 `template.Library` 实例然后调用它的 `tag()` 方法。例如：

```
register.tag('current_time', do_current_time)
```

`tag()` 方法需要两个参数:

模板标签的名字（字符串）。如果被遗漏的话，将会使用编译函数的名字。

编译函数。

和注册过滤器类似，也可以在Python2.4及其以上版本中使用 `register.tag` 修饰：

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

如果你像在第二个例子中那样忽略 `name` 参数的话，Django会使用函数名称作为标签名称。

在上下文中设置变量

前一节的例子只是简单的返回一个值。很多时候设置一个模板变量而非返回值也很有用。那样，模板作者就只能使用你的模板标签所设置的变量。

要在上下文中设置变量，在 `render()` 函数的context对象上使用字典赋值。这里是一个修改过的 `CurrentTimeNode`，其中设定了一个模板变量 `current_time`，并没有返回它：

```
class CurrentTimeNode2(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        now = datetime.datetime.now()
        context['current_time'] = now.strftime(self.format_string)
        return ''
```

注意 `render()` 返回了一个空字符串。 `render()` 应当总是返回一个字符串，所以如果模板标签只是要设置变量，`render()` 就应该返回一个空字符串。

你应该这样使用这个新版本的标签：

```
{% current_time2 "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

但是 `CurrentTimeNode2` 有一个问题: 变量名 `current_time` 是硬编码的。这意味着你必须确定你的模板在其它任何地方都不使用 `{{ current_time }}`，因为 `{% current_time2 %}` 会盲目的覆盖该变量的值。

一种更简洁的方案是由模板标签来指定需要设定的变量的名称，就像这样：

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

为此，你需要重构编译函数和 `Node` 类，如下所示：

```
import re

class CurrentTimeNode3(template.Node):

    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name

    def render(self, context):
        now = datetime.datetime.now()
        context[self.var_name] = now.strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r tag requires arguments' % token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*?) as (\w+)', arg)
    if m:
        fmt, var_name = m.groups()
    else:
        msg = '%r tag had invalid arguments' % tag_name
        raise template.TemplateSyntaxError(msg)

    if not (fmt[0] == fmt[-1] and fmt[0] in ('"', "'")):
        msg = "%r tag's argument should be in quotes" % tag_name
```

```
        raise template.TemplateSyntaxError(msg)

    return CurrentTimeNode3(fmt[1:-1], var_name)
```

现在 `do_current_time()` 把格式字符串和变量名传递给 `CurrentTimeNode3` 。

分析直至另一个块标签

模板标签可以像包含其它标签的块一样工作（想想 `{% if %}` 、 `{% for %}` 等）。 要创建一个这样的模板标签，在你的编译函数中使用 `parser.parse()` 。

标准的 `{% comment %}` 标签是这样实现的：

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

`parser.parse()` 接收一个包含了需要分析块标签名的元组作为参数. 它返回一个 `django.template.NodeList` 实例, 它是一个包含了所有 `Node` 对象的列表,这些对象代表了分析器在遇到元组中任一标签名之 *前* 的内容.

因此在前面的例子中， `nodelist` 是在 `{% comment %}` 和 `{% endcomment %}` 之间所有节点的列表，不包括 `{% comment %}` 和 `{% endcomment %}` 自身。

在 `parser.parse()` 被调用之后，分析器还没有清除 `{% endcomment %}` 标签，因此代码需要显式地调用 `parser.delete_first_token()` 来防止该标签被处理两次。

之后 `CommentNode.render()` 只是简单地返回一个空字符串。在 `{% comment %}` 和 `{% endcomment %}` 之间的所有内容都被忽略。

分析直至另外一个块标签并保存内容

在前一个例子中， `do_comment()` 抛弃了在 `{% comment %}` 和 `{% endcomment %}` 之间的所有内容。同样，也可以对块标签之间的代码进行处理。

例如，这个自定义模板标签： `{% upper %}` ，它把自己和 `{% endupper %}` 之间的所有内容都变成大写：

```
{% upper %}
    This will appear in uppercase, {{ your_name }}.
{% endupper %}
```

就像前面的例子一样，我们将使用 `parser.parse()` 。这次，我们将产生的 `nodelist` 传递给 `Node` ：

```
@register.tag
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):

    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

这里唯一的一个新概念是 `UpperNode.render()` 中的 `self.nodelist.render(context)` 。它对节点列表中的每个 `Node` 简单的调用 `render()` 。

更多的复杂渲染示例请查看 `django/template/defaulttags.py` 中的 `{% if %}`、 `{% for %}`、 `{% ifequal %}` 和 `{% ifchanged %}` 的代码。

简单标签的快捷方式

许多模板标签接收单一的字符串参数或者一个模板变量 引用,然后独立地根据输入变量和一些其它外部信息进行处理并返回一个字符串. 例如, 我们先前写的 `current_time` 标签就是这样一个例子. 我们给它格式字符串, 然后它把时间作为字符串返回.

为了简化这类标签，Django 提供了一个帮助函数： `simple_tag` 。这个函数是 `django.template.Library` 的一个方法，它接受一个只有一个参数的函数作参数，把它包装在 `render` 函数和之前提及过的其他的必要单位中，然后通过模板系统注册标签。

我们之前的的 `current_time` 函数于是可以写成这样：

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

在Python 2.4中，也可以使用修饰语法：

```
@register.simple_tag
def current_time(token):
    ...
```

有关 `simple_tag` 辅助函数，需要注意下面一些事情：

- 传递给我们的函数的只有（单个）参数。
- 在我们的函数被调用的时候，检查必需参数个数的工作已经完成了，所以我们不需要再做这个工作。
- 参数两边的引号（如果有的话）已经被截掉了，所以我们会接收到一个普通字符串。

包含标签

另外一类常用的模板标签是通过渲染 *其他* 模板显示数据的。比如说，Django的后台管理界面，它使用了自定义的模板标签来显示新增/编辑表单页面下部的按钮。那些按钮看起来总是一样的，但是链接却随着所编辑的对象的不同而改变。这就是一个使用小模板很好的例子，这些小模板就是当前对象的详细信息。

这些排序标签被称为 *包含标签* 。如何写包含标签最好通过举例来说明。我们来写一个可以生成一个选项列表的多选项对象 `Poll` 。标签这样使用：

```
{% show_results poll %}
```

结果将会像下面这样：

```
<ul>
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

首先，我们定义一个函数，通过给定的参数生成一个字典形式的结果。 需要注意的是，我们只需要返回字典类型的结果就行了，它将被用做模板片断的context。 (译注：dict 的 key 作为变量名在模板中被使用)

```
def show_books_for_author(author):
    books = author.book_set.all()
    return {'books': books}
```

接下来，我们创建用于渲染标签输出的模板。在我们的例子中，模板很简单：

```
<ul>
{% for book in books %}
```

```
    <li> {{ book }} </li>
{% endfor %}
</ul>
```

最后，我们通过对一个 `Library` 对象使用 `inclusion_tag()` 方法来创建并注册这个包含标签。

在我们的例子中，如果先前的模板在 `polls/result_snippet.html` 文件中，那么我们这样注册标签：

```
register.inclusion_tag('books/books_for_author.html')(show_books_for_author)
```

和往常一样，我们也可以使用Python 2.4中的修饰语法，所以我们还可以这么写：

```
@register.inclusion_tag('books/books_for_author.html')
def show_books_for_author(show_books_for_author):
    ...
```

有时候，你的包含标签需要访问父模板的context。为了解决这个问题，Django提供了一个 `takes_context` 选项。如果你在创建模板标签时，指明了这个选项，这个标签就不需要参数，并且下面的Python函数会带一个参数：就是当这个标签被调用时的模板context。

例如，你正在写一个包含标签，该标签包含有指向主页的 `home_link` 和 `home_title` 变量。Python函数会像这样：

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

备注

函数的第一个参数 *必须* 是 `context` 。

模板 `link.html` 可能包含下面的东西：

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

然后您想使用自定义标签时，就可以加载它的库，然后不带参数地调用它，就像这样：

```
{% jump_link %}
```

编写自定义模板加载器

Djangos 内置的模板加载器（在先前的模板加载内幕章节有叙述）通常会满足你的所有的模板加载需求，但是如果你有特殊的加载需求的话，编写自己的模板加载器也会相当简单。比如：你可以从数据库加载模板，或者使用 Subversions 的Python实现直接从Subversion库加载模板，再或者（稍后展示）从zip文件加载模板。

一个模板加载器，也就是 `TEMPLATE_LOADERS` 中的每一项，都要能被下面这个接口所调用：

```
load_template_source(template_name, template_dirs=None)
```

参数 `template_name` 是所加载模板的名称 (和传递给 `loader.get_template()` 或者 `loader.select_template()` 一样), 而 `template_dirs` 是一个可选的包含除去 `TEMPLATE_DIRS` 之外的搜索目录列表。

如果加载器能够成功加载一个模板, 它应当返回一个元组： `(template_source, template_path)` 。在这里的 `template_source` 就是将被模板引擎编译的的模板字符串，而 `template_path` 是被加载的模板的路径。由于那个路径可能会出于调试目的显示给用户，因此它应当很快的指明模板从哪里加载而来。

如果加载器加载模板失败，那么就会触发 `django.template.TemplateDoesNotExist` 异常。

每个加载函数都应该有一个名为 `is_usable` 的函数属性。这个属性是一个布尔值，用于告知模板引擎这个加载器是否在当前安装的Python中可用。例如，如果 `pkg_resources` 模块没有安装的话，eggs加载器（它能够从python eggs中加载模板）就应该把 `is_usable` 设为 `False` ，因为必须通过 `pkg_resources` 才能从eggs中读取数据。

一个例子可以清晰地阐明一切。这儿是一个模板加载函数，它可以从ZIP文件中加载模板。它使用了自定义的设置 `TEMPLATE_ZIP_FILES` 来取代了 `TEMPLATE_DIRS` 用作查找路径，并且它假设在此路径上的每一个文件都是包含模板的ZIP文件：

```
import zipfile
from django.conf import settings
from django.template import TemplateDoesNotExist

def load_template_source(template_name, template_dirs=None):
    """Template loader that loads templates from a ZIP file."""

    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])

    # Try each ZIP file in TEMPLATE_ZIP_FILES.
    for fname in template_zipfiles:
        try:
            z = zipfile.ZipFile(fname)
            source = z.read(template_name)
        except (IOError, KeyError):
            continue
        z.close()
        # We found a template, so return the source.
        template_path = "%s:%s" % (fname, template_name)
        return (source, template_path)

    # If we reach here, the template couldn't be loaded
    raise TemplateDoesNotExist(template_name)

# This loader is always usable (since zipfile is included with Python)
load_template_source.is_usable = True
```

我们要想使用它，还差最后一步，就是把它加入到 `TEMPLATE_LOADERS` 。如果我们把这部分代码放到一个叫做 `mysite.zip_loader` 的包中，我们就需要把 `mysite.zip_loader.load_template_source` 加入到 `TEMPLATE_LOADERS` 中去。

使用内置的模板参考

Django管理界面包含一个完整的参考资料，里面有所有的可以在特定网站上使用的模板标签和过滤器。它设计的初衷是Django程序员提供给模板开发人员的一个工具。你可以点击管理页面右上角的文档链接来查看这些资料。

参考说明分为4个部分：标签、过滤器、模型和视图。 *标签* 和 *过滤器* 部分描述了所有内置的标签（实际上，第4章中用到的标签和过滤器都直接来源于那几页）以及一些可用的自定义标签和过滤器库。

视图 页面是最有价值的。网站中的每个URL都在这儿有独立的入口。如果相关的视图包含一个 文档字符串， 点击URL，你就会看到：

- 生成本视图的视图函数的名字
- 视图功能的一个简短描述
- 上下文或一个视图模板中可用的变量的列表
- 视图使用的模板的名字

要想查看关于视图文档的更详细的例子，请阅读Django的通用 `object_list` 视图部分的源代码，它位于 `django/views/generic/list_detail.py` 文件中。

通常情况下，由Django构建的网站都会使用数据库对象， *模型* 页面描述了系统中所有类型的对象，以及该对象对应的所有可用字段。

总之，这些文档告诉你在模板中的所有可用的标签、过滤器、变量和对象。

配置独立模式下的模板系统

备注

这部分只针对于对在其他应用中使用模版系统作为输出组件感兴趣的人。如果你是在Django应用中使用模版系统，请略过此部分。

通常，Django会从它的默认配置文件和由 `DJANGO_SETTINGS_MODULE` 环境变量所指定的模块中加载它需要的所有配置信息。但是当你想在非Django应用中使用模版系统的时候，采用环境变量并不是很好的方法。比起为模版系统单独采用配置文件并用环境变量来指向它，你可能更希望能够在你的应用中采用一致的配置方法来配置模版系统和其他部分

为了解决这个问题，你需要使用附录E中所描述的手动配置选项。简单来说，你需要引入合适的模板系统，并且在调用任何模板函数 *之前* 调用 `django.conf.settings.configure()` 来指定任何你想要的设置。

你可能会考虑至少要设置 `TEMPLATE_DIRS` （如果你打算使用模板加载器）， `DEFAULT_CHARSET` （尽管默认的 `utf-8` 编码相当好用），以及 `TEMPLATE_DEBUG` 。所有可用的选项都在附录E中详细描述，所有以 `TEMPLATE_` 开头的选项都可能使你感兴趣的。

接下来？

迄今为止，本书假定您想展示的内容为HTML。对于一个有关Web开发的书来说，这不是一个 不好的假设，但有时你想用Django输出其他数据格式。

下一章将讲解如何使用Django生成图像、PDF、还有你可以想到的其他数据格式。

第十一章 输出非HTML内容

通常当我们谈到开发网站时，主要谈论的是HTML。当然，Web远不只有HTML，我们在Web上用多种格式来发布数据：RSS、PDF、图片等。

到目前为止，我们的注意力都是放在常见 HTML 代码生成上，但是在这一章中，我们将会对使用 Django 生成其它格式的内容进行简要介绍。

Django拥有一些便利的内建工具帮助你生成常见的非HTML内容：

- RSS/Atom 聚合文件
- 站点地图 （一个XML格式文件，最初由Google开发，用于给搜索引擎提示线索）

我们稍后会逐一研究这些工具，不过首先让我们来了解些基础原理。

基础: 视图和MIME类型

还记得第三章的内容吗？

一个视图函数（view function），或者简称 *view*，只不过是一个可以处理一个Web请求并且返回 一个Web响应的Python函数。这个响应可以是一个Web页面的HTML内容，或者一个跳转，或者一个404 错误，或者一个XML文档，或者一幅图片，或者映射到任何东西上。

更正式的说，一个Django视图函数 *必须*

- 接受一个 `HttpRequest` 实例作为它的第一个参数
- 返回一个 `HttpResponse` 实例

从一个视图返回一个非 HTML 内容的关键是在构造一个 `HttpResponse` 类时，需要指定 `mimetype` 参数。通过改变 MIME 类型，我们可以告知浏览器将要返回的数据是另一种不同的类型。

下面我们以返回一张PNG图片的视图为例。为了使事情能尽可能的简单，我们只是读入一张存储在磁盘上的图片：

```
from django.http import HttpResponse

def my_image(request):
    image_data = open("/path/to/my/image.png", "rb").read()
    return HttpResponse(image_data, mimetype="image/png")
```

就是这么简单。如果改变 `open()` 中的图片路径为一张真实图片的路径，那么就可以使用这个十分简单的视图来提供一张图片，并且浏览器可以正确的显示它。

另外我们必须了解的是"HttpResponse"对象应用了Python标准的文件应用程序接口(API)。这就是说你可以在Python（或第三方库）任何用到文件的地方使用"HttpResponse"实例。

下面将用 Django 生成 CSV 文件为例，说明它的工作原理。

生成 CSV 文件

CSV 是一种简单的数据格式，通常为电子表格软件所使用。它主要是由一系列的表格行组成，每行中单元格之间使用逗号(CSV 是 *逗号分隔数值(comma-separated values)* 的缩写)隔开。例如，下面是以 CSV 格式记录的一些违规航班乘客的数据。

```
Year,Unruly Airline Passengers
1995,146
1996,184
```

```
1997,235
1998,200
1999,226
2000,251
2001,299
2002,273
2003,281
2004,304
2005,203
```

备注

前面的列表是真实的数据，数据由美国联邦航空管理处提供。具体内容请参见 http://www.faa.gov/data_statistics/passengers_cargo/unruly_passengers/.

虽然 CSV 看上去简单，以至于简单到这个格式甚至都没有正式的定义。但是不同的软件会生成和使用不同的 CSV 的变种，在使用上会有一些不便。幸运的是， Python 使用的是标准 CSV 库， `csv` ，所以它更通用。

因为 `csv` 模块操作的是类似文件的对象，所以可以使用 `HttpResponse` 替换：

```
import csv
from django.http import HttpResponse

# Number of unruly passengers each year 1995 - 2005. In a real application
# this would likely come from a database or some other back-end data store.
UNRULY_PASSENGERS = [146,184,235,200,226,251,299,273,281,304,203]

def unruly_passengers_csv(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=unruly.csv'

    # Create the CSV writer using the HttpResponse as the "file"
    writer = csv.writer(response)
    writer.writerow(['Year', 'Unruly Airline Passengers'])
    for (year, num) in zip(range(1995, 2006), UNRULY_PASSENGERS):
        writer.writerow([year, num])

    return response
```

代码和注释可以说是很清楚，但还有一些事情需要特别注意：

- 响应返回的是 `text/csv` MIME类型（而非默认的 `text/html` ）。这会告诉浏览器，返回的文档是CSV文件。
- 响应会有一个附加的 `Content-Disposition` 头部，它包含有CSV文件的文件名。这个头部（或者说，附加部分）会指示浏览器弹出对话框询问文件存放的位置（而不仅仅是显示）。这个文件名是任意的，它会用在浏览器的另存为对话框中。
- 与创建CSV的应用程序界面（API）挂接是很容易的：只需将 `response` 作为第一个变量传递给 `csv.writer` 。 `csv.writer` 函数希望获得一个文件类的对象， `HttpResponse` 正好能达成这个目的。
- 调用 `writer.writerow` ，并且传递给它一个类似 list 或者 tuple 的可迭代对象，就可以在 CSV 文件中写入一行。
- CSV 模块考虑到了引用的问题，所以您不用担心逸出字符串中引号和逗号。只要把信息传递给 `writerow()` ，它会处理好所有的事情。

在任何需要返回非 HTML 内容的时候，都需要经过以下几步：创建一个 `HttpResponse` 响应对象（需要指定特殊的 MIME 类型）。将它作为参数传给一个需要文件的方法，然后返回这个响应。

下面是一些其它的例子

生成 PDF 文件

便携文件格式 (PDF) 是由 Adobe 开发的格式，主要用于呈现可打印的文档，包含有 pixel-perfect 格式，嵌入字体以及2D 矢量图像。PDF 文件可以被认为是一份打印文档的数字等价物；实际上，PDF 文件通常用于需要将文档交付给其他人

去打印的场合。

便携文件格式 (PDF) 是由 Adobe 开发的格式，主要用于呈现可打印的文档，包含有 pixel-perfect 格式，嵌入字体以及2D 矢量图像。PDF 文件可以被认为是一份打印文档的数字等价物；实际上，PDF 文件通常用于需要将文档交付给其他人去打印的场合。

下面的例子是使用 Django 和 ReportLab 在 KUSports.com 上生成个性化的可打印的 NCAA 赛程表 (tournament brackets) 。

安装 ReportLab

在生成 PDF 文件之前，需要安装 ReportLab 库。这通常是个很简单的过程：从 http://www.reportlab.org/downloads.html 下载并且安装这个库即可。

使用手册（原始的只有 PDF 格式）可以从 http://www.reportlab.org/rsrc/userguide.pdf 下载，其中包含有一些其它的安装指南。

注意

如果使用的是一些新的 Linux 发行版，则在安装前可以先检查包管理软件。多数软件包仓库中都加入了 ReportLab 。

比如，如果使用（杰出的） Ubuntu 发行版，只需要简单的 `apt-get install python-reportlab` 一行命令即可完成安装。

在 Python 交互环境中导入这个软件包以检查安装是否成功。

```
>>> import reportlab
```

如果刚才那条命令没有出现任何错误，则表明安装成功。

编写视图

和 CSV 类似，由 Django 动态生成 PDF 文件很简单，因为 ReportLab API 同样可以使用类似文件对象。

下面是一个 Hello World 的示例：

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

需要注意以下几点：

- 这里我们使用的 MIME 类型是 `application/pdf` 。这会告诉浏览器这个文档是一个 PDF 文档，而不是 HTML 文档。如果忽略了这个参数，浏览器可能会把这个文件看成 HTML 文档，这会使浏览器的窗口中出现很奇怪的文字。

- 使用 ReportLab 的 API 很简单：只需要将 `response` 对象作为 `canvas.Canvas` 的第一个参数传入。 `Canvas` 类需要一个类似文件的对象， `HttpResponse` 对象可以满足这个要求。

- 所有后续的 PDF 生成方法需要由 PDF 对象调用（在本例中是 `p`），而不是 `response` 对象。
- 最后需要对 PDF 文件调用 `showPage()` 和 `save()` 方法（否则你会得到一个损坏的 PDF 文件）。

复杂的 PDF 文件

如果您在创建一个复杂的 PDF 文档（或者任何较大的数据块），请使用 `cStringIO` 库存放临时生成的 PDF 文件。`cStringIO` 提供了一个用 C 编写的类似文件对象的接口，从而可以使系统的效率最高。

下面是使用 `cStringIO` 重写的 Hello World 例子：

```
from cStringIO import StringIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    temp = StringIO()

    # Create the PDF object, using the StringIO object as its "file."
    p = canvas.Canvas(temp)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the StringIO buffer and write it to the response.
    response.write(temp.getvalue())
    return response
```

其它的可能性

使用 Python 可以生成许多其它类型的内容，下面介绍的是一些其它的想法和一些可以用以实现它们的库。

> *ZIP 文件*：Python 标准库中包含有 `zipfile` 模块，它可以读和写压缩的 ZIP 文件。它可以用于按需生成一些文件的压缩包，或者在需要时压缩大的文档。如果是 TAR 文件则可以使用标准库 `tarfile` 模块。
>
> *动态图片*： Python 图片处理库 (PIL; http://www.pythonware.com/products/pil/) 是极好的生成图片(PNG, JPEG, GIF 以及其它许多格式)的工具。它可以用于自动为图片生成缩略图，将多张图片压缩到单独的框架中，或者是做基于 Web 的图片处理。
>
> *图表*： Python 有许多出色并且强大的图表库用以绘制图表，按需地图，表格等。我们不可能将它们全部列出，所以下面列出的是个中的翘楚。
>
> - `matplotlib` (http://matplotlib.sourceforge.net/) 可以用于生成通常是由 matlab 或者 Mathematica 生成的高质量图表。
> - `pygraphviz` (https://networkx.lanl.gov/wiki/pygraphviz) 是一个 Graphviz 图形布局的工具 (http://graphviz.org/) 的 Python 接口，可以用于生成结构化的图表和网络。

总之，所有可以写文件的库都可以与 Django 同时使用。请相信一切皆有可能。

我们已经了解了生成“非HTML”内容的基本知识，让我们进一步总结一下。Django拥有很多用以生成各类“非HTML”内容的内置工具。

内容聚合器应用框架

Django带来了一个高级的聚合生成框架，它使得创建RSS和Atom feeds变得非常容易。

什么是RSS？什么是Atom?

RSS和Atom都是基于XML的格式，你可以用它来提供有关你站点内容的自动更新的feed。了解更多关于RSS的可以访问 http://www.whatisrss.com/, 更多Atom的信息可以访问 http://www.atomenabled.org/.

想创建一个联合供稿的源(syndication feed)，所需要做的只是写一个简短的python类。你可以创建任意多的源(feed)。

高级feed生成框架是一个默认绑定到/feeds/的视图，Django使用URL的其它部分(在/feeds/之后的任何东西)来决定输出 哪个feed

要创建一个feed, 您将创建一个 `Feed` 类, 并在您的 URLconf 中指向它. (查看第3章和第8章, 可以获取更多有关URLconfs 的更多信息)

初始化

为了在您的Django站点中激活syndication feeds, 添加如下的 URLconf:

```
(r'^feeds/(?P<url>.*)/$',
 'django.contrib.syndication.views.feed',
 {'feed_dict': feeds}
),
```

这一行告诉Django使用RSS框架处理所有的以 `"feeds/"` 开头的URL. (你可以修改 `"feeds/"` 前缀以满足您自己的要求.)

URLConf里有一行参数：``{‘feed_dict’: feeds}``，这个参数可以把对应URL需要发布的feed内容传递给 syndication framework

特别的，feed_dict应该是一个映射feed的slug(简短URL标签)到它的Feed类的字典 你可以在URL配置本身里定义 feed_dict，这里是一个完整的例子

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

前面的例子注册了两个feed:

- `LatestEntries``表示的内容将对应到``feeds/latest/`.
- `LatestEntriesByCategory``的内容将对应到 ``feeds/categories/`.

以上的设定完成之后，接下来需要自己定义 `Feed` 类

一个 `Feed` 类是一个简单的python类，用来表示一个syndication feed. 一个feed可能是简单的 (例如一个站点新闻feed，或者最基本的，显示一个blog的最新条目)，也可能更加复杂(例如一个显示blog某一类别下所有条目的feed。这里类别 category 是个变量).

Feed类必须继承django.contrib.syndication.feeds.Feed，它们可以在你的代码树的任何位置

一个简单的Feed

例子来自于chicagocrime.org，描述最近5项新闻条目的feed:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]
```

要注意的重要的事情如下所示:

子类 `django.contrib.syndication.feeds.Feed`.

`title`, `link`, 和 `description` 对应一个标准 RSS 里的 `<title>`, `<link>`, 和 `<description>` 标签.

`items()` 是一个方法，返回一个用以包含在包含在feed的 `<item>` 元素里的 list 虽然例子里用Djangos database API返回的 `NewsItem` 对象, `items()` 不一定必须返回 model的实例

你可以利用 Django models免费实现一定功能，但是 `items()` 可以返回你想要的任意类型的对象.

还有一个步骤，在一个RSS feed里，每个(item)有一个(title)，(link)和(description)，我们需要告诉框架 把数据放到这些元素中

如果要指定 `<title>` 和 `<description>`，可以建立一个Django模板（见Chapter 4）名字叫 `feeds/latest_title.html` 和 `feeds/latest_description.html`，后者是URLConf里为对应feed指定的 `slug`。注意 `.html` 后缀是必须的。

RSS系统模板渲染每一个条目，需要给传递2个参数给模板上下文变量：

- `obj`: 当前对象 (返回到 `items()` 任意对象之一)。
- `site`: 一个表示当前站点的 `django.models.core.sites.Site` 对象。 这对于 `{{ site.domain }}` 或者 `{{ site.name }}` 很有用。

如果你在创建模板的时候，没有指明标题或者描述信息，框架会默认使用 `"{{ obj }}"`，对象的字符串表示。

你也可以通过修改 `Feed` 类中的两个属性 `title_template` 和 `description_template` 来改变这两个模板的名字。

你有两种方法来指定 `<link>` 的内容。 Django 首先执行 `items()` 中每一项的 `get_absolute_url()` 方法。 如果该方法不存在，就会尝试执行 `Feed` 类中的 `item_link()` 方法，并将自身作为 `item` 参数传递进去。

`get_absolute_url()` 和 `item_link()` 都应该以Python字符串形式返回URL。

对于前面提到的 `LatestEntries` 例子，我们可以实现一个简单的feed模板。 `latest_title.html` 包括：

```
{{ obj.title }}
```

并且 `latest_description.html` 包含:

```
{{ obj.description }}
```

这真是 *太* 简单了!

一个更复杂的Feed

框架通过参数支持更加复杂的feeds。

举个例子，chicagocrime.org提供了一个RSS源以跟踪每一片区域的犯罪近况。如果为每一个单独的区域建立一个 `Feed`

类就显得很不明智。这样做就违反了DRY原则了，程序逻辑也会和数据耦合在一起。

取而代之的方法是，使用聚合框架来产生一个通用的源，使其可以根据feeds URL返回相应的信息。

在chicagocrime这个例子中，区域信息可以通过这样的URL方式来访问:

- `http://www.chicagocrime.org/rss/beats/0613/` ：返回0613号地区的犯罪数据
- `http://www.chicagocrime.org/rss/beats/1424/` ：返回1424号地区的犯罪数据

固定的那一部分是 `"beats"` （区域）。聚合框架看到了后面的不同之处 `0613` 和 `1424` ，它会提供给你一个钩子函数来描述这些URL的意义，以及会对feed中的项产生的影响。

举个例子会澄清一切。下面是每个地区特定的feeds：

```
from django.core.exceptions import ObjectDoesNotExist

class BeatFeed(Feed):
    def get_object(self, bits):
        # In case of "/rss/beats/0613/foo/bar/baz/", or other such
        # clutter, check that bits has only one member.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Beat.objects.get(beat__exact=bits[0])

    def title(self, obj):
        return "Chicagocrime.org: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        crimes =  Crime.objects.filter(beat__id__exact=obj.id)
        return crimes.order_by('-crime_date')[:30]
```

以下是RSS框架的基本算法，我们假设通过URL `/rss/beats/0613/` 来访问这个类：

> 框架获得了URL `/rss/beats/0613/` 并且注意到URL中的slug部分后面含有更多的信息。它将斜杠(`"/"`)作为分隔符，把剩余的字符串分割开作为参数，调用 `Feed` 类的 `get_object()` 方法。
>
> 在这个例子中，添加的信息是 `['0613']` 。对于 `/rss/beats/0613/foo/bar/` 的一个URL请求， 这些信息就是 `['0613', 'foo', 'bar']` 。
>
> `get_object()` 就根据给定的 `bits` 值来返回区域信息。
>
> 在这个例子中，它使用了Django的数据库API来获取信息。注意到如果给定的参数不合法， `get_object()` 会抛出 `django.core.exceptions.ObjectDoesNotExist` 异常。在 `Beat.objects.get()` 调用中也没有出现 `try` /`except` 代码块。函数在出错时抛出 `Beat.DoesNotExist` 异常，而 `Beat.DoesNotExist` 是 `ObjectDoesNotExist` 异常的一个子类型。 而在 `get_object()`

System Message: WARNING/2 (`<string>`, line 798)

Block quote ends without a blank line; unexpected unindent.

中抛出 `ObjectDoesNotExist` 异常又会使得Django引发404错误。

> 为产生 `<title>` ， `<link>` ， 和 `<description>` 的feeds， Django使用 `title()` , `link()` , 和 `description()` 方法。 在上面的例子中，它们都是简单的字符串类型的类属性，而这个例子表明，它们既可以是字符串，*也可以是* 方法。对于每一个 `title` ， `link` 和 `description` 的组合，Django使用以下的算法：
>
> 1. 试图调用一个函数，并且以 `get_object()` 返回的对象作为参数传递给 `obj` 参数。

2. 如果没有成功，则不带参数调用一个方法。

3. 还不成功，则使用类属性。

最后，值得注意的是，这个例子中的 `items()` 使用 `obj` 参数。对于 `items` 的算法就如同上面第一步所描述的那样，首先尝试 `items(obj)`， 然后是 `items()`，最后是 `items` 类属性（必须是一个列表）。

`Feed` 类所有方法和属性的完整文档，请参考官方的Django文档 (http://www.djangoproject.com/documentation/0.96/syndication_feeds/) 。

指定Feed的类型

默认情况下, 聚合框架生成RSS 2.0. 要改变这样的情况, 在 `Feed` 类中添加一个 `feed_type` 属性.

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

注意你把 `feed_type` 赋值成一个类对象，而不是类实例。目前合法的Feed类型如表11-1所示。

表 11-1. Feed 类型

Feed类	类型
`django.utils.feedgenerator.Rss201rev2Feed`	RSS 2.01 (default)
`django.utils.feedgenerator.RssUserland091Feed`	RSS 0.91
`django.utils.feedgenerator.Atom1Feed`	Atom 1.0

闭包

为了指定闭包（例如，与feed项比方说MP3 feeds相关联的媒体资源信息），使用 `item_enclosure_url`， `item_enclosure_length`， 以及 `item_enclosure_mime_type`，比如

```
from myproject.models import Song

class MyFeedWithEnclosures(Feed):
    title = "Example feed with enclosures"
    link = "/feeds/example-with-enclosures/"

    def items(self):
        return Song.objects.all()[:30]

    def item_enclosure_url(self, item):
        return item.song_url

    def item_enclosure_length(self, item):
        return item.song_length

    item_enclosure_mime_type = "audio/mpeg"
```

当然，你首先要创建一个包含有 `song_url` 和 `song_length` （比如按照字节计算的长度）域的 `Song` 对象。

语言

聚合框架自动创建的Feed包含适当的 `<language>` 标签(RSS 2.0) 或 `xml:lang` 属性(Atom). 他直接来自于您的 `LANGUAGE_CODE` 设置.

URLs

`link` 方法/属性可以以绝对URL的形式（例如， `"/blog/"` ）或者指定协议和域名的URL的形式返回（例如 `"http://www.example.com/blog/"` ）。如果 `link` 没有返回域名，聚合框架会根据 `SITE_ID` 设置，自动的插入当前站点的域信息。

Atom feeds需要 `<link rel="self">` 指明feeds现在的位置。聚合框架根据 `SITE_ID` 的设置，使用站点的域名自动完成这些功能。

同时发布Atom and RSS

一些开发人员想 *同时* 支持Atom和RSS。这在Django中很容易实现：只需创建一个你的 `feed` 类的子类，然后修改 `feed_type` ，并且更新URLconf内容。下面是一个完整的例子：

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
```

这是与之相对应那个的URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

Sitemap 框架

sitemap 是你服务器上的一个XML文件，它告诉搜索引擎你的页面的更新频率和某些页面相对于其它页面的重要性。这个信息会帮助搜索引擎索引你的网站。

例如，这是 Django 网站(http://www.djangoproject.com/sitemap.xml)sitemap的一部分：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.djangoproject.com/documentation/</loc>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.djangoproject.com/documentation/0_90/</loc>
    <changefreq>never</changefreq>
    <priority>0.1</priority>
  </url>
  ...
</urlset>
```

需要了解更多有关 sitemaps 的信息, 请参见 http://www.sitemaps.org/.

Django sitemap 框架允许你用 Python 代码来表述这些信息，从而自动创建这个XML文件。要创建一个 sitemap，你只需要写一个 `Sitemap` 类然后配置你的URLconf指向它。

安装

要安装 sitemap 应用程序, 按下面的步骤进行:

1. 将 `'django.contrib.sitemaps'` 添加到您的 `INSTALLED_APPS` 设置中.
2. 确保 `'django.template.loaders.app_directories.load_template_source'` 在您的 `TEMPLATE_LOADERS` 设置中。默认情况下它在那里, 所以, 如果你已经改变了那个设置的话, 只需要改回来即可。
3. 确定您已经安装了 sites 框架 (参见第14章).

备注

sitemap 应用程序没有安装任何数据库表. 它需要加入到 `INSTALLED_APPS` 中的唯一原因是: 这样 `load_template_source` 模板加载器可以找到默认的模板.

初始化

要在您的Django站点中激活sitemap生成, 请在您的 URLconf 中添加这一行:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
```

这一行告诉 Django, 当客户访问 `/sitemap.xml` 的时候, 构建一个 sitemap.

sitemap文件的名字无关紧要，但是它在服务器上的位置却很重要。搜索引擎只索引你的sitemap中当前URL级别及其以下级别的链接。用一个实例来说，如果 `sitemap.xml` 位于你的根目录，那么它将引用任何的URL。然而，如果你的sitemap位于 `/content/sitemap.xml` ，那么它只引用以 `/content/` 打头的URL。

sitemap视图需要一个额外的必须的参数： `{'sitemaps': sitemaps}` 。 `sitemaps` 应该是一个字典，它把一个短的块标签(例如, `blog` 或 `news`)映射到它的 `Sitemap` 类(例如， `BlogSitemap` 或 `NewsSitemap`)。它也可以映射到一个 `Sitemap` 类的实例(例如， `BlogSitemap(some_var)`)。

Sitemap 类

`Sitemap` 类展示了一个进入地图站点简单的Python类片断.例如,一个 `Sitemap` 类能展现所有日志入口，而另外一个能够调度所有的日历事件。

在最简单的例子中，所有部分可以全部包含在一个 `sitemap.xml` 中，也可以使用框架来产生一个站点地图，为每一个独立的部分产生一个单独的站点文件。

`Sitemap` 类必须是 `django.contrib.sitemaps.Sitemap` 的子类. 他们可以存在于您的代码树的任何地方。

例如假设你有一个blog系统，有一个 `Entry` 的model，并且你希望你的站点地图包含所有连到你的blog入口的超链接。你的 `Sitemap` 类很可能是这样的：

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

声明一个 `Sitemap` 和声明一个 `Feed` 看起来很类似；这都是预先设计好的。

如同 `Feed` 类一样， `Sitemap` 成员也既可以是方法，也可以是属性。想要知道更详细的内容，请参见上文 《一个复杂的例子》章节。

一个 Sitemap 类可以定义如下 方法/属性:

items (**必需**): 提供对象列表。框架并不关心对象的 *类型*；唯一关心的是这些对象会传递给 location() ， lastmod() ， changefreq() ，和 priority() 方法。

location （可选）：给定对象的绝对URL。绝对URL不包含协议名称和域名。下面是一些例子：

- 好的: '/foo/bar/'
- 差的: 'example.com/foo/bar/'
- 差的: 'http://example.com/foo/bar/'

如果没有提供 location , 框架将会在每个 items() 返回的对象上调用 get_absolute_url() 方法.

lastmod (可选): 对象的最后修改日期, 作为一个Python datetime 对象.

changefreq （可选）：对象变更的频率。可选的值如下（详见Sitemaps文档）：

- 'always'
- 'hourly'
- 'daily'
- 'weekly'
- 'monthly'
- 'yearly'
- 'never'

priority （可选）：取值范围在 0.0 and 1.0 之间，用来表明优先级。默认值为 0.5 ；请详见 http://sitemaps.org 文档。

快捷方式

sitemap框架提供了一些常用的类。在下一部分中会看到。

FlatPageSitemap

django.contrib.sitemaps.FlatPageSitemap 类涉及到站点中所有的flat page，并在sitemap中建立一个入口。但仅仅只包含 location 属性，不支持 lastmod， changefreq，或者 priority 。

参见第16章获取有关flat page的更多的内容.

GenericSitemap

GenericSitemap 与所有的通用视图一同工作（详见第9章）。

你可以如下使用它，创建一个实例，并通过 info_dict 传递给通用视图。唯一的要求是字典包含 queryset 这一项。也可以用 date_field 来指明从 queryset 中取回的对象的日期域。这会被用作站点地图中的 lastmod 属性。你也可以向 GenericSitemap 的构造函数传递 priority 和 changefreq 来指定所有URL的相应属性。

下面是一个使用 FlatPageSitemap and GenericSiteMap （包括前面所假定的 Entry 对象）的URLconf:

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry
```

```
info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap.xml$',
     'django.contrib.sitemaps.views.sitemap',
     {'sitemaps': sitemaps})
)
```

创建一个Sitemap索引

sitemap框架同样可以根据 `sitemaps` 字典中定义的单独的sitemap文件来建立索引。用法区别如下：

- 您在您的URLconf中使用了两个视图: `django.contrib.sitemaps.views.index` 和 `django.contrib.sitemaps.views.sitemap` .
- `django.contrib.sitemaps.views.sitemap` 视图需要带一个 `section` 关键字参数.

这里是前面的例子的相关的 URLconf行看起来的样子:

```
(r'^sitemap.xml$',
 'django.contrib.sitemaps.views.index',
 {'sitemaps': sitemaps}),

(r'^sitemap-(?P<section>.+).xml$',
 'django.contrib.sitemaps.views.sitemap',
 {'sitemaps': sitemaps})
```

这将自动生成一个 `sitemap.xml` 文件, 它同时引用 `sitemap-flatpages.xml` 和 `sitemap-blog.xml` . `Sitemap` 类和 `sitemaps` 目录根本没有更改.

通知Google

当你的sitemap变化的时候，你会想通知Google，以便让它知道对你的站点进行重新索引。框架就提供了这样的一个函数： `django.contrib.sitemaps.ping_google()` 。

备注

在本书写成的时候, 只有Google可以响应sitemap更新通知。然而，Yahoo和MSN可能很快也会支持这些通知。

到那个时候，把“ping_google()”这个名字改成“ping_search_engines()”会比较好。所以还是到 http://www.djangoproject.com/documentation/0.96/sitemaps/ 去检查一下最新的站点地图文档。

`ping_google()` 有一个可选的参数 `sitemap_url` ，它应该是你的站点地图的URL绝对地址（例如： `/sitemap.xml` ）。如果不提供该参数， `ping_google()` 将尝试通过反查你的URLconf来找到你的站点地图。

如果不能够确定你的sitemap URL, `ping_google()` 会引发 `django.contrib.sitemaps.SitemapNotFound` 异常。

我们可以通过模型中的 `save()` 方法来调用 `ping_google()` :

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
```

```
    def save(self):
        super(Entry, self).save()
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            pass
```

一个更有效的解决方案是用 `cron` 脚本或任务调度表来调用 `ping_google()` ，该方法使用Http直接请求Google服务器，从而减少每次调用 `save()` 时占用的网络带宽。

接下来？

下面, 我们要继续深入挖掘所有的Django给你的很好的内置工具。 在第12章，您将看到提供用户自定义站点所需要的所有工具： sessions, users 和authentication.

继续前行!

十二章: Sessions, Users和 Registration

是时候承认了：我们有意的避开了web开发中极其重要的方面。到目前为止，我们都在假定，网站流量是大量的匿名用户带来的。

这当然不对，浏览器的背后都是活生生的人(至少某些时候是)。我们忽略了一件重要的事情：互联网服务于人而不是机器。要开发一个真正令人心动的网站，我们必须面对浏览器后面活生生的人。

很不幸，这并不容易。HTTP被设计为"无状态"，每次请求都处于相同的空间中。在一次请求和下一次请求之间没有任何状态保持，我们无法根据请求的任何方面(IP地址，用户代理等)来识别来自同一人的连续请求。

在本章中你将学会如何搞定状态的问题。好了，我们会从较低的层次(*cookies*)开始，然后过渡到用高层的工具来搞定会话，用户和注册的问题。

Cookies

浏览器的开发者在很早的时候就已经意识到， HTTP's 的无状态会对Web开发者带来很大的问题，于是(*cookies*)应运而生。cookies 是浏览器为 Web 服务器存储的一小段信息。每次浏览器从某个服务器请求页面时，它向服务器回送之前收到的cookies

来看看它是怎么工作的。当你打开浏览器并访问 `google.com`，你的浏览器会给Google发送一个HTTP请求，起始部分就象这样：

```
GET / HTTP/1.1
Host: google.com
...
```

当 Google响应时，HTTP的响应是这样的：

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
            expires=Sun, 17-Jan-2038 19:14:07 GMT;
            path=/; domain=.google.com
Server: GWS/2.1
...
```

注意 `Set-Cookie` 的头部。你的浏览器会存储cookie值(`PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671`)，而且每次访问google 站点都会回送这个cookie值。因此当你下次访问Google时，你的浏览器会发送像这样的请求：

```
GET / HTTP/1.1
Host: google.com
Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...
```

于是 `Cookies` 的值会告诉Google，你就是早些时候访问过Google网站的人。这个值可能是数据库中存储用户信息的key，可以用它在页面上显示你的用户名。

存取Cookies

在Django中处理持久化，大部分时候你会更愿意用高层些的session 和/或 后面要讨论的user 框架。但在此之前，我们需要停下来在底层看看如何读写cookies。这会帮助你理解本章节后面要讨论的工具是如何工作的，而且如果你需要自己操作cookies，这也会有所帮助。

读取已经设置好的cookies极其简单，每个request对象都有一个 `COOKIES` 对象，可以象使用字典般使用它，你可以读取任何浏览器发给视图(view)的任何cookies:

```
def show_color(request):
    if "favorite_color" in request.COOKIES:
```

```
        return HttpResponse("Your favorite color is %s" % \
            request.COOKIES["favorite_color"])
    else:
        return HttpResponse("You don't have a favorite color.")
```

写cookies稍微复杂点，需要用 `HttpResponse` 对象的 `set_cookie()` 方法来写。这儿有个基于 `GET` 参数来设置 `favorite_color` cookie的例子：

```
def set_color(request):
    if "favorite_color" in request.GET:

        # Create an HttpResponse object...
        response = HttpResponse("Your favorite color is now %s" % \
            request.GET["favorite_color"])

        # ... and set a cookie on the response
        response.set_cookie("favorite_color",
                            request.GET["favorite_color"])

        return response

    else:
        return HttpResponse("You didn't give a favorite color.")
```

你可以给 `response.set_cookie()` 传递一些可选的参数来控制cookie的行为，详见表12-1。

表 12-1: Cookie 选项

参数	缺省值	描述
`max_age`	`None`	cookies的持续有效时间（以秒计），如果设置为 `None` cookies 在浏览器关闭的时候就失效了。
`expires`	`None`	cookies的过期时间，格式： `"Wdy, DD-Mth-YY HH:MM:SS GMT"` 如果设置这个参数，它将覆盖 `max_age` 参数。
`path`	`"/"`	cookie生效的路径前缀，浏览器只会把cookie回传给带有该路径的页 面，这样你可以避免将cookie传给站点中的其他的应用。 当你的应用不处于站点顶层的时候，这个参数会非常有用。
`domain`	`None`	cookie生效的站点。你可用这个参数来构造一个跨站cookie。如， `domain=".example.com"` 所构造的cookie对下面这些站点都是可 读的： `www.example.com` 、 `www2.example.com` 和 `an.other.sub.domain.example.com` 。 如果该参数设置为 `None` ，cookie只能由设置它的站点读取。
`secure`	`False`	如果设置为 `True` ，浏览器将通过HTTPS来回传cookie。

好坏参半的Cookies

也许你已经注意到了，cookies的工作方式可能导致的问题，一起来看看其中一些重要的方面：

cookies存取完全是非强制性的，浏览器不保证这一点。事实上，所有的浏览器都让用户自己控制 是否接受cookies。如果你想知道cookies对于web应用有多重要，你可以试着打开这个浏览器的 选项：提示我接受每次cookie。

尽管cookies广为使用，但仍被认为是不可靠的的。这意味着，开发者使用cookies之前必须 检查用户是否可以接收cookie。

更重要的是，*永远* 也不要在cookie中存储重要的数据。开发者在cookie中存储了不可恢复 的数据，而浏览器却因为某种原因将cookie中的数据清得一干二净，这样令人发指的故事在 Web世界中比比皆是。

Cookie(特别是那些没通过HTTPS传输的)是非常不安全的。因为HTTP数据是以明文发送的，所以 特别容易

受到嗅探攻击。也就是说，嗅探攻击者可以在网络中拦截并读取cookies，因此你要 绝对避免在cookies中存储敏感信息。

还有一种被称为"中间人"的攻击更阴险，攻击者拦截一个cookie并将其用于另一个用户。 第19章将深入讨论这种攻击的本质以及如何避免。

即使从预想中的接收者返回的cookie也是不安全的，因为大多数浏览器都提供了很方便的方法来 修改cookies的内容，有技术背景的用户甚至可以用像mechanize (http://wwwsearch.sourceforge.net/mechanize/) 这样的工具来手工构造HTTP请求。

因此不能在cookies中存储可能会被篡改的敏感数据，"经典"错误是：在cookies中存储 `IsLoggedIn=1` ，以标识用户已经登录。犯这类错误的站点数量多的令人难以置信； 绕过这些网站的安全系统也是易如反掌。

Django的 Session 框架

由于存在的限制与安全漏洞，cookies和持续性会话已经成为Web开发中令人头疼的典范。好消息是，Django的目标正是高效的"头疼杀手"，它自带的session框架会帮你搞定这些问题。

你可以用session 框架来存取每个访问者任意数据，这些数据在服务器端存储，并用通过cookie来传输数据摘要。cookies只存储数据的哈希会话ID，而不是数据本身，从而避免了大部分的常见cookie问题。

下面我们来看看如何打开session功能，并在视图中使用它。

打开 Sessions功能

Sessions 功能是通过一个中间件(middleware)和一个模型(model)来实现的。要打开sessions功能，需要以下几步操作：

1. 编辑 `MIDDLEWARE_CLASSES` 配置，确保 `MIDDLEWARE_CLASSES` 中包含 `'django.contrib.sessions.middleware.SessionMiddleware'`
2. 确认 `INSTALLED_APPS` 中有 `'django.contrib.sessions'` (如果你是刚打开这个应用，别忘了运行 `manage.py syncdb`)

如果项目是用 `startproject` 来创建的，配置文件中都已经安装了这些东西，除非你自己删除，正常情况下，你无需任何设置就可以使用session功能。

如果不需要session功能，你可以删除 `MIDDLEWARE_CLASSES` 设置中的 `SessionMiddleware` 和 `INSTALLED_APPS` 设置中的 `'django.contrib.sessions'` 。虽然这只会节省很少的开销，但积少成多啊。

在视图中使用Session

`SessionMiddleware` 激活后，每个传给视图(view)函数的第一个参数``HttpRequest`` 对象都有一个 `session` 属性，这是一个字典型的对象。你可以象用普通字典一样来用它。例如，在视图(view)中你可以这样用：

```
# Set a session value:
request.session["fav_color"] = "blue"

# Get a session value -- this could be called in a different view,
# or many requests later (or both):
fav_color = request.session["fav_color"]

# Clear an item from the session:
del request.session["fav_color"]

# Check if the session has a given key:
if "fav_color" in request.session:
    ...
```

其他的映射方法，如 `keys()` 和 `items()` 对 `request.session` 同样有效：

下面是一些有效使用Django sessions的简单规则：

- 用正常的字符串作为key来访问字典 `request.session`， 而不是整数、对象或其它什么的。 这不是什么强硬的条规，但值得遵循。
- Session字典中以下划线开头的key值是Django内部保留key值。框架只会用很少的几个下划线 开头的session变量，除非你知道他们的具体含义，而且愿意跟上Django的变化，否则，最好 不要用这些下划线开头的变量，它们会让Django搅乱你的应用。
- 不要用一个新对象来替换掉 `request.session`，也不要存取其属性，象用普通Python字典一样用它。

我们来看个简单的例子。这是个简单到不能再简单的例子：在用户发了一次评论后将 `has_commented` 设置为 `True`，这是个简单（但不很安全）的、防止用户多次评论的方法。

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

下面是一个很简单的站点登录视图(view)：

```
def login(request):
    try:
        m = Member.objects.get(username__exact=request.POST['username'])
        if m.password == request.POST['password']:
            request.session['member_id'] = m.id
            return HttpResponse("You're logged in.")
    except Member.DoesNotExist:
        return HttpResponse("Your username and password didn't match.")
```

这是退出登录，根据 `login()`：

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

注意

在实践中，这是很烂的用户登录方式，稍后讨论的认证(authentication)框架会帮你以更健壮和有利的方式来处理这些问题。这些非常简单的例子只是想让你知道这一切是如何工作的。

设置测试Cookies

就像前面提到的，你不能指望所有的浏览器都可以接受cookie，因此，Django为了方便，也提供了检查用户浏览器是否接受cookie的简单方法。你只需在视图(view)中调用 `request.session.set_test_cookie()`，并在后续的视图(view)、而不是当前的视图(view)中检查 `request.session.test_cookie_worked()`。

虽然把 `set_test_cookie()` 和 `test_cookie_worked()` 分开的做法看起来有些笨拙，但由于cookie的工作方式，这无可避免。当设置一个cookie时候，只能等浏览器下次访问的时候，你才能知道浏览器是否接受cookie。

检查cookie是否可以正常工作后，你得自己用 `delete_test_cookie()` 来清除它，这是个好习惯。

这是个典型例子：

```
def login(request):

    # If we submitted the form...
    if request.method == 'POST':

        # Check that the test cookie worked (we set it below):
        if request.session.test_cookie_worked():
```

```
            # The test cookie worked, so delete it.
            request.session.delete_test_cookie()

            # In practice, we'd need some logic to check username/password
            # here, but since this is an example...
            return HttpResponse("You're logged in.")

        # The test cookie failed, so display an error message. If this
        # was a real site we'd want to display a friendlier message.
        else:
            return HttpResponse("Please enable cookies and try again.")

    # If we didn't post, send the test cookie along with the login form.
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

注意

再次强调，内置的认证函数会帮你做检查的。

在视图(View)外使用Session

从内部来看，每个session都只是一个普通的Django model（在 `django.contrib.sessions.models` 中定义)。每个session都由一个随机的32字节哈希串来标识，并存储于数据库中。由于这是一个普通的model，你可以用一般的Django 数据库API来读取session。

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

你得用 `get_decoded()` 来读取实际的session数据，因为session字典经过了编码存储。

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

何时保存Session

缺省的情况下，Django只会在session发生变化的时候才会存入数据库，比如说，字典赋值或删除。

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

你可以设置 `SESSION_SAVE_EVERY_REQUEST` 为 `True` 来改变这一缺省行为。如果 `SESSION_SAVE_EVERY_REQUEST` 设置为 `True` ，Django 会在每次请求的时候都把session存到数据库中，即使没有任何改变。

注意，会话cookie只会在创建和修改的时候才会送出。但如果 `SESSION_SAVE_EVERY_REQUEST` 设置为 `True` ，会话cookie会在每次请求的时候都会送出。同时，每次会话cookie送出的时候，其 `expires` 参数都会更新。

浏览器关闭即失效会话 vs. 持久会话

你可能注意到了，Google给我们发送的cookie中有 `expires=Sun, 17-Jan-2038 19:14:07 GMT;` cookie可以有过期时间，这样浏览器就知道什么时候可以删除cookie了。如果cookie没有设置过期时间，当用户关闭浏览器的时候，cookie就自

动过期了。你可以改变 SESSION_EXPIRE_AT_BROWSER_CLOSE 的设置来控制session框架的这一行为。

缺省情况下， SESSION_EXPIRE_AT_BROWSER_CLOSE 设置为 False ，这样，会话cookie可以在用户浏览器中保持有效达 SESSION_COOKIE_AGE 秒（缺省设置是两周，即1,209,600 秒）。如果你不想用户每次打开浏览器都必须重新登陆的话，用这个参数来帮你。

如果 SESSION_EXPIRE_AT_BROWSER_CLOSE 设置为 True ，当浏览器关闭时，Django会使cookie失效。

其他的Session设置

除了上面提到的设置，还有一些其他的设置可以影响Django session框架如何使用cookie，详见表 12-2.

表 12-2. 影响cookie行为的设置

设置	描述	缺省值
SESSION_COOKIE_DOMAIN	session cookie生效的站点，跨站点生效的 cookie可以这样设置：``".lawrence.com"`` None 为标准cookie	None
SESSION_COOKIE_NAME	用于session 的cookie名称，可以是任何 字符串	"sessionid"
SESSION_COOKIE_SECURE	是否在session中使用安全cookie，如果设置 True , cookie就会标记为安全， 这意味着cookie只会通过HTTPS来传输	False

技术细节

如果你还是好奇的话，下面是一些关于session框架内部工作方式的技术细节：

session 字典和普通Python对象一样，支持序列化，详见Python文档中内置 pickle 模块的部分。

Session 数据存在数据库表 django_session 中

Session 数据在需要的时候才会读取，如果你从不使用 request.session ， Django不会动相关数据库表的一根毛。

Django 只在需要的时候才送出cookie。如果你压根儿就没有设置任何会话数据，它不会 送出会话cookie(除非 SESSION_SAVE_EVERY_REQUEST 设置为 True)

Django session 框架完全而且只能基于cookie，不会后退到把会话ID编码在URL中。（像某些工具(PHP,JSP)那样）

这是一个有意而为之的设计，把session放在URL中不只是难看，更重要的是这让你的站点 很容易受到攻击——通过 Referer header进行session ID"窃听"而实施的攻击。

如果你还是好奇，阅读源代码是最直接办法，详见 django.contrib.sessions 。

用户与Authentication

现在，我们通过浏览器连接真实用户的目标已经完成一半了。通过session，我们可以在多次浏览器请求中保持数据，接下来的部分就是用session来处理用户登录了。当然，不能仅凭用户的一面之词，我们就相信，所以我们需要认证。

当然了，Django 也提供了工具来处理这样的常见任务（就像其他常见任务一样）。Django 用户认证系统处理用户帐号，组，权限以及基于cookie的用户会话。这个系统一般被称为 *auth/auth* (认证与授权)系统，这个系统的名称同时也表明了用户常见的两步处理。我们需要

1. 验证 (*认证*) 用户是否是他所宣称的用户(一般通过查询数据库验证其用户名和密码)
2. 验证用户是否拥有执行某种操作的 *授权* (通常会通过检查一个权限表来确认)

根据这些需求，Django 认证/授权 系统会包含以下的部分：

- *用户* : 在网站注册的人

- *权限* : 用于标识用户是否拥有某种操作的二进制(yes/no)标志
- *组* :一种可以将标记和权限应用于多个用户的常用方法
- *Messages* : 向用户显示队列式的系统消息的常用方法
- *Profiles* : 通过自定义字段扩展用户对象的机制

如果你已经用了admin工具(详见第6章)，就会看见这些工具的大部分。如果已经用了admin工具来编辑用户和组，你实际上就已经在编辑认证系统中数据库表。

打开认证支持

像session工具一样，认证支持也是一个Django应用，放在 `django.contrib` 中，所以也需要安装。与session系统相似，它也是缺省安装的，但如果它已经被删除了，通过以下步骤也能重新安装上：

1. 根据本章早前的部分确认已经安装了session 框架，需要确认用户使用cookie，这样sesson 框架才能正常使用。
2. 将 `'django.contrib.auth'` 放在你的 `INSTALLED_APPS` 设置中，然后运行 `manage.py syncdb`
3. 确认 `SessionMiddleware` 后面的 `MIDDLEWARE_CLASSES` 设置中包含 `'django.contrib.auth.middleware.AuthenticationMiddleware'`

这样安装后，我们就可以在视图(view)的函数中用处理user了。在视图中存取users，主要用 `request.user` ；这个对象表示当前已登录的用户，如果用户还没登录，这就是一个 `匿名` 对象(细节见下)

你可以很容易的通过 `is_authenticated()` 方法来判断一个用户是否已经登录了

```
if request.user.is_authenticated():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

使用User对象

`User` 实例一般从 `request.user` ，或是其他下面即将要讨论到的方法取得，它有很多属性和方法。 `AnonymousUser` 对象模拟了 *部分* 的接口，但不是全部，在把它当成真正的user对象 使用前，你得检查一下 `user.is_authenticated()`

表 12-3. `User` 对象属性

属性	描述
`username`	必填; 少于等于30字符. 只允许字符，数字，下划线
`first_name`	可选; 少于等于30字符.
`last_name`	可选; 少于等于30字符.
`email`	可选. 邮件地址.
`password`	必填. 密码的摘要hash(Django不会存储原始密码)，详见密码章节部分
`is_staff`	布尔值. 用户是否拥有网站的管理权限.
`is_active`	布尔值. 是否允许用户登录, 设置为``False``，可以不用删除用户来禁止 用户登录
`is_superuser`	布尔值. 用户是否拥有所有权限，而无需任何显式的权限分配定义
`last_login`	用户最后登录的时间，缺省会设置为当前时间
`date_joined`	创建用户的时间，当用户创建时，缺省的设置为当前的时间

表 12-4. `User` 对象方法

方法	描述
`is_authenticated()`	如果是真正的 `User` 对象，返回值恒为 `True` 。 用于检查用户是否已经通过了认证。通过认证并不意味着 用户拥有任何权限，甚至也不检查该用户是否处于激活状 态，这只是表明用户成功的通过了认证。
`is_anonymous()`	如果是个 `AnonymousUser` ，返回值为 `True` ， 如果是 `User` 对象，返回值

方法	描述
	为 False 。一般来 说， is_authenticated() 会比这个方法更常用些。
get_full_name()	返回值为： first_name 加上 last_name ，以 空格分隔。
set_password(passwd)	将用户的密码设置为给定的字符串，实际密码已被哈希 处理。这时并不会真正保存 User 对象。
check_password(passwd)	如果给定的字符串通过了密码检查，返回 True 。 密码比较已进行了哈希处理。
get_group_permissions()	返回用户通过所属组获得的权限列表
get_all_permissions()	返回用户通过所属组和用户自身权限所获得的所有权限 列表。
has_perm(perm)	如果用户拥有给定的权限，返回 True ， perm 应形如 "package.codename" 的格式。如果用户处于 非激活状态，则总是返回 False 。
has_perms(perm_list)	如果用户拥有所有给定的权限，返回 True 。 如果用户处于非激活状态，则总是返回 False 。
has_module_perms(app_label)	如果用户拥有任何给定 app_label 的权限，返回 True 。如果用户处于非激活状态，则总是返回 False
get_and_delete_messages()	返回用户的 Message 对象列表，并从队列中删除。
email_user(subj, msg)	给用户发送电子邮件，用 DEFAULT_FROM_EMAIL 的设 置作为发件人。也可以用第3个参数 from_email 来 覆盖设置。
get_profile()	返回用户的网站自定义profile，详见Profile章节

最后， User 对象有两个多对多的属性： groups 和 permissions 。 User 对象可以 象使用其他多对多属性的方法一样使用它们。

```
# Set a user's groups:
myuser.groups = group_list

# Add a user to some groups:
myuser.groups.add(group1, group2,...)

# Remove a user from some groups:
myuser.groups.remove(group1, group2,...)

# Remove a user from all groups:
myuser.groups.clear()

# Permissions work the same way
myuser.permissions = permission_list
myuser.permissions.add(permission1, permission2, ...)
myuser.permissions.remove(permission1, permission2, ...)
myuser.permissions.clear()
```

登录和退出

Django 提供内置的视图(view)函数用于处理登录和退出 (以及其他奇技淫巧)，但在开始前，我们来看看如何手工登录和退出，Django 在 django.contrib.auth 中提供了两个函数来处理这些事情——authenticate() 和 login() 。

认证给出的用户名和密码，使用 authenticate() 函数。它接受两个参数，用户名 username 和 密码 password ，并在密码对用给出的用户名是合法的情况下返回一个 User 对象。当给出的密码不合法的时候 authenticate() 函数返回 None ：

```
>>> from django.contrib import auth
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...     print "Correct!"
... else:
...     print "Oops, that's wrong!"
```

authenticate() 只是验证一个用户的证书而已。而要登录一个用户，使用 login() 。该函数接受一个 HttpRequest 对象和一个 User 对象作为参数并使用Django的会话（ session ）框架把用户的ID保存在该会话中。

下面的例子演示了如何在一个视图中同时使用 `authenticate()` 和 `login()` 函数：

```
from django.contrib import auth

def login(request):
    username = request.POST['username']
    password = request.POST['password']
    user = auth.authenticate(username=username, password=password)
    if user is not None and user.is_active:
        # Correct password, and the user is marked "active"
        auth.login(request, user)
        # Redirect to a success page.
        return HttpResponseRedirect("/account/loggedin/")
    else:
        # Show an error page
        return HttpResponseRedirect("/account/invalid/")
```

注销一个用户，在你的视图中使用 `django.contrib.auth.logout()` 。该函数接受一个 `HttpRequest` 对象作为参数，没有返回值。

```
from django.contrib import auth

def logout(request):
    auth.logout(request)
    # Redirect to a success page.
    return HttpResponseRedirect("/account/loggedout/")
```

注意，即使用户没有登录， `logout()` 也不会抛出任何异常。

在实际中，你一般不需要自己写登录/登出的函数；认证系统提供了一系例视图用来处理登录和登出。

使用认证视图的第一步是把它们写在你的URLconf中。 你需要这样写：

```
from django.contrib.auth.views import login, logout

urlpatterns = patterns('',
    # existing patterns here...
    (r'^accounts/login/$',  login),
    (r'^accounts/logout/$', logout),
)
```

`/accounts/login/` 和 `/accounts/logout/` 是Django提供的视图的默认URL。

缺省情况下， `login` 视图渲染 `registragiton/login.html` 模板(可以通过视图的额外参数 `template_name` 修改这个模板名称)。这个表单必须包含 `username` 和 `password` 域。如下示例：

```
{% extends "base.html" %}

{% block content %}

  {% if form.errors %}
    <p class="error">Sorry, that's not a valid username or password</p>
  {% endif %}

  <form action='.' method='post'>
    <label for="username">User name:</label>
    <input type="text" name="username" value="" id="username">
    <label for="password">Password:</label>
    <input type="password" name="password" value="" id="password">

    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next|escape }}" />
  <form action='.' method='post'>

{% endblock %}
```

如果用户登录成功，缺省会重定向到 `/accounts/profile` 。表单中有一个hidden字段叫 `next` ，可以用在登录后指定url。也可以把这个值（指定的url）作为 `GET` 参数传递给login视图，这个参数会成为Context中名为 `next` 的变量，你可以

把这个变量设置给表单中对应的隐含字段。

logout视图有一些不同。缺省的它渲染 `registration/logged_out.html` 模板（这个视图一般包含你已经成功退出的信息）。视图中还可以包含一个参数 `next_page` 用于退出后重定向。

限制已登录用户的访问

有很多原因需要控制用户访问站点的某部分。

一个简单原始的限制方法是检查 `request.user.is_authenticated()` ,然后重定向到登陆页面：

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s' % request.path)
    # ...
```

或者显示一个出错信息：

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

作为一个快捷方式, 你可以使用便捷的 `login_required` 修饰符:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # ...
```

`login_required` 做下面的事情:

- 如果用户没有登录, 重定向到 `/accounts/login/` , 把当前绝对URL作为 `next` 在查询字符串中传递过去, 例如: `/accounts/login/?next=/polls/3/` .
- 如果用户已经登录, 正常地执行视图函数. 视图代码就可以假定用户已经登录了.

对通过测试的用户限制访问

限制访问可以基于某种权限，某些检查或者为login视图提供不同的位置，这些实现方式大致相同

一般的方法是直接在视图的 `request.user` 上运行检查。例如，下面视图检查用户登陆并是否有 `polls.can_vote` 的权限：

```
def vote(request):
    if request.user.is_authenticated() and request.user.has_perm('polls.can_vote')):
        # vote here
    else:
        return HttpResponse("You can't vote in this poll.")
```

并且Django有一个称为 `user_passes_test` 的简洁方式。它根据情况使用参数并且产生特殊装饰符。

```
def user_can_vote(user):
    return user.is_authenticated() and user.has_perm("polls.can_vote")

@user_passes_text(user_can_vote, login_url="/login/")
def vote(request):
    # Code here can assume a logged-in user with the correct permission.
    ...
```

`user_passes_test` 使用一个必需的参数：一个可调用的方法，当存在 `User` 对象并当此用户允许查看该页面时返回 `True` 。 注意 `user_passes_test` 不会自动检查 `User` 是否认证，你应该自己做这件事。

例子中我们也展示了第二个可选的参数 `login_url` ，它让你指定你的登录页面的URL（默认为 `/accounts/login/` ）。

既然检查用户是否有一个特殊权限是相对常见的任务，Django为这种情形提供了一个捷径： `permission_required()` 装饰器 使用这个装饰器，前面的例子可以这样写:

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url="/login/")
def vote(request):
    # ...
```

注意, `permission_required()` 也有一个可选的 `login_url` 参数, 这个参数默认为 `'/accounts/login/'` 。

限制通用视图的访问

在Django用户邮件列表中问到最多的问题是关于对通用视图的限制性访问。为实现这个功能，你需要自己包装视图，并且在URLconf中，将你自己的版本替换通用视图：

```
from dango.contrib.auth.decorators import login_required
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

当然, 你可以用任何其他限定修饰符来替换 `login_required` 。

管理 Users, Permissions 和 Groups

管理认证系统最简单的方法是通过管理界面。 第六章讨论了怎样使用Django的管理界面来编辑用户和控制他们的权限和可访问性，并且大多数时间你都会只使用这个界面。

然而，当你需要绝对的控制权的时候，有一些低层 API 需要深入专研，我们将在下面的章节中讨论它们。

创建用户

使用 `create_user` 辅助函数创建用户:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user(username='john',
...                                 email='jlennon@beatles.com',
...                                 password='glass onion')
```

在这里， `user` 是 `User` 类的一个实例，准备用于向数据库中存储数据。 `create_user()` 函数并没有在数据库中创建记录，在保存数据之前，你仍然可以继续修改它的属性值。

```
>>> user.is_staff = True
>>> user.save()
```

修改密码

你可以使用 `set_password()` 来修改密码：

```
>>> user = User.objects.get(username='john')
>>> user.set_password('goo goo goo joob')
>>> user.save()
```

除非你清楚的知道自己在做什么，否则不要直接修改 `password` 属性。其中保存的是密码的 *加入salt的hash值* ，所以不能直接编辑。

一般来说， `User` 对象的 `password` 属性是一个字符串，格式如下：

```
hashtype$salt$hash
```

这是哈希类型，salt和哈希本身，用美元符号（$）分隔。

`hashtype` 是 `sha1` （默认）或者 `md5` ，它是用来处理单向密码哈希的算法，Salt是一个用来加密原始密码来创建哈希的随机字符串，例如:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

`User.set_password()` 和 `User.check_password()` 函数在后台处理和检查这些值。

一个加入salt的哈希算法是某种毒品吗？

不是，一个 *加入salt值的哈希算法* 可不是什么毒品；事实上它是一种确保密码存储安全的常用方法。一次 *哈希* 是一次单向的加密过程，你能容易地计算出一个给定值的哈希码，但是几乎不可能从一个哈希码解出它的原值。

如果我们以普通文本存储密码,任何能进入数据库的人都能轻易的获取每个人的密码。使用哈希方式来存储密码相应的减少了数据库泄露密码的可能。

然而，攻击者仍然可以使用 *暴力破解* 使用上百万个密码与存储的值对比来获取数据库密码，这需要花一些时间，但是智能电脑惊人的速度超出了你的想象

更糟糕的是我们可以公开地得到 *rainbow tables* （一种暴力密码破解表）或预备有上百万哈希密码值的数据库。使用rainbow tables可以在几秒之内就能搞定最复杂的一个密码。

在存储的hash值的基础上，加入 *salt* 值（一个随机值），增加了密码的强度，使得破解更加困难。因为每个密码的salt值都不相同，这也限制了rainbow table的使用，使得攻击者只能使用最原始的暴力破解方法。而加入的salt值使得hash的熵进一步获得增加，使得暴力破解的难度又进一步加大。

加入salt值得hash并不是绝对安全的存储密码的方法，然而在安全和方便之间有很大的中间地带需要我们来做决定。

处理注册

我们可以使用这些底层工具来创建允许用户注册的视图。最近每个开发人员都希望实现各自不同的注册方法，所以Django把写一个注册视图的工作留给了你。幸运的是，这很容易。

作为这个事情的最简化处理, 我们可以提供一个小视图, 提示一些必须的用户信息并创建这些用户. Django为此提供了可用的内置表单, 在下面这个例子中很好地使用了:

```
from django import oldforms as forms
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.contrib.auth.forms import UserCreationForm

def register(request):
    form = UserCreationForm()

    if request.method == 'POST':
        data = request.POST.copy()
        errors = form.get_validation_errors(data)
        if not errors:
            new_user = form.save(data)
            return HttpResponseRedirect("/books/")
    else:
        data, errors = {}, {}

    return render_to_response("registration/register.html", {
        'form' : forms.FormWrapper(form, data, errors)
    })
```

这个表单构想了一个叫 `registration/register.html` 的模板. 这里是一个这个模板的可能的样子的例子:

```
{% extends "base.html" %}

{% block title %}Create an account{% endblock %}

{% block content %}
```

```
  <h1>Create an account</h1>
  <form action="." method="post">
    {% if form.error_dict %}
      <p class="error">Please correct the errors below.</p>
    {% endif %}

    {% if form.username.errors %}
      {{ form.username.html_error_list }}
    {% endif %}
    <label for="id_username">Username:</label> {{ form.username }}

    {% if form.password1.errors %}
      {{ form.password1.html_error_list }}
    {% endif %}
    <label for="id_password1">Password: {{ form.password1 }}

    {% if form.password2.errors %}
      {{ form.password2.html_error_list }}
    {% endif %}
    <label for="id_password2">Password (again): {{ form.password2 }}

    <input type="submit" value="Create the account" />
  </label>
{% endblock %}
```

备注

在本书出版之时, `django.contrib.auth.forms.UserCreationForm` 是一个 *oldforms* 表单. 参看 http://www.djangoproject.com/documentation/0.96/forms/ 可以获取有关 oldforms 的详细信息. 转换到有关 newforms 的内容在第7章中将会讲述, newforms 功能 将会在不远的将来完成.

在模板中使用认证数据

当前登入的用户以及他（她）的权限可以通过 `RequestContext` 在模板的context中使用（详见第10章）。

备注

从技术上来说，只有当你使用了 `RequestContext` *并且* `TEMPLATE_CONTEXT_PROCESSORS` 设置包含了 `"django.core.context_processors.auth"` （默认情况就是如此）时，这些变量才能在模板context中使用。更详细的内容，也请参考第10章。

当使用 `RequestContext` 时, 当前用户 (是一个 `User` 实例或一个 `AnonymousUser` 实例) 存储在模板变量 `{{ user }}` 中:

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
  <p>Welcome, new user. Please log in.</p>
{% endif %}
```

这些用户的权限信息存储在 `{{ perms }}` 模板变量中。这是一个在模板中使用很方便的代理，其中包含一些权限相关函数的简写。

你有两种方式来使用 `perms` 对象。你可以使用类似于 `{{ perms.polls }}` 的形式来检查，对于某个特定的应用，一个用户是否具有 *任意* 权限；你也可以使用 `{{ perms.polls.can_vote }}` 这样的形式，来检查一个用户是否拥有特定的权限。

这样你就可以在模板中的 `{% if %}` 语句中检查权限:

```
{% if perms.polls %}
  <p>You have permission to do something in the polls app.</p>
  {% if perms.polls.can_vote %}
    <p>You can vote!</p>
  {% endif %}
{% else %}
  <p>You don't have permission to do anything in the polls app.</p>
{% endif %}
```

其他一些功能：权限，组，消息和档案

在认证框架中还有其他的一些功能。我们会在接下来的几个部分中进一步地了解它们。

权限

权限可以很方便地标识用户和用户组可以执行的操作。它们被Django的admin管理站点所使用，你也可以在你自己的代码中使用它们。

Django的admin站点如下使用权限：

- 只有设置了 *add* 权限的用户才能使用添加表单，添加对象的视图。
- 只有设置了 *change* 权限的用户才能使用变更列表，变更表格，变更对象的视图。
- 只有设置了 *delete* 权限的用户才能删除一个对象。

权限是根据每一个类型的对象而设置的，并不具体到对象的特定实例。例如，我们可以允许Mary改变新故事，但是目前还不允许设置Mary只能改变自己创建的新故事，或者根据给定的状态，出版日期或者ID号来选择权限。

这三个基本权限：添加，变更和删除，会被自动添加到所有的Django模型中，只要该模型包含 `class Admin` 。当你执行 `manage.py syncdb` 的时候，这些就被自动添加到 `auth_permission` 数据表中。

权限以 `"<app>.<action>_<object_name>"` 的形式出现。如果你有一个 `polls` 的应用，包含一个 `Choice` 模型，你就有以下三个权限，分别叫做 `"polls.add_choice"` ， `"polls.change_choice"` ，和 `"polls.delete_choice"` 。

注意，如果当你运行 `syncdb` 时，模型中没有包含 `class Admin` ，该模型对应的权限就不会被创建。如果你在初始化数据库以后，又在自己的模型中加入了 `class Admin` ，你就需要重新运行 `syncdb` 来为应用加入权限。

你也可以通过设置 `Meta` 中的 `permissions` 属性，来为给定的模型定制权限。下面的例子创建了三个自定义的权限：

```
class USCitizen(models.Model):
    # ...
    class Meta:
        permissions = (
            # Permission identifier     human-readable permission name
            ("can_drive",               "Can drive"),
            ("can_vote",                "Can vote in elections"),
            ("can_drink",               "Can drink alcohol"),
        )
```

当你运行 `syncdb` 时，额外的权限才会被加入；你需要自己在视图中添加权限相关的代码。

就跟用户一样，权限也就是Django模型中的 `django.contrib.auth.models` 。因此如果你愿意，你也可以通过Django的数据库API直接操作权限。

组

组提供了一种通用的方式来让你按照一定的权限规则和其他标签将用户分类。一个用户可以隶属于任何数量的组。

在一个组中的用户自动获得了赋予该组的权限。例如， `Site editors` 组拥有 `can_edit_home_page` 权限，任何在该组中的用户都拥有这个权限。

组也可以通过给定一些用户特殊的标记，来扩展功能。例如，你创建了一个 `'Special users'` 组，并且允许组中的用户访问站点的一些VIP部分，或者发送VIP的邮件消息。

和用户管理一样，admin接口是管理组的最简单的方法。然而，组也就是Django模型 `django.contrib.auth.models` ，因此你可以使用Django的数据库API，在底层访问这些组。

消息

消息系统会为给定的用户接收消息。每个消息都和一个 `User` 相关联。其中没有超时或者时间戳的概念。

在每个成功的操作以后，Django的admin管理接口就会使用消息机制。例如，当你创建了一个对象，你会在admin页面的顶上看到 `The object was created successfully` 的消息。

你也可以使用相同的API在你自己的应用中排队接收和显示消息。API非常地简单：

- 要创建一条新的消息，使用 `user.message_set.create(message='message_text')` 。
- 要获得/删除消息，使用 `user.get_and_delete_messages()` ，这会返回一个 `Message` 对象的列表，并且从队列中删除返回的项。

在例子视图中，系统在创建了播放单（playlist）以后，为用户保存了一条消息。

```
def create_playlist(request, songs):
    # Create the playlist with the given songs.
    # ...
    request.user.message_set.create(
        message="Your playlist was added successfully."
    )
    return render_to_response("playlists/create.html",
        context_instance=RequestContext(request))
```

当使用 `RequestContext` ，当前登录的用户以及他（她）的消息，就会以模板变量 `{{ messages }}` 出现在模板的context中。下面是显示消息的一个例子模板代码：

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

需要注意的是 `RequestContext` 会在后台调用 `get_and_delete_messages` ，因此即使你没有显示它们，它们也会被删除掉。

最后注意，这个消息框架只能服务于在用户数据库中存在的用户。如果要向匿名用户发送消息，请直接使用会话框架。

档案

最后一个难题是档案系统.为了理解什么是档案,让我们先看看问题.

简单来说，许多网站需要存储比标准 `User` 对象更多的用户信息。为了解决这个问题，大多数网站都会有不同的额外字段。所以，Django提供一个轻量级的方式定义档案对象链接到指定的用户。这个档案对象在每个项目中可以是不同的，甚至可以为同一数据库服务的不同的站点处理不同的档案。

创建档案的第一步是定义一个模型（model）来存储档案信息。Django对这个模型所做的唯一的限制是，必须要包含唯一的一个对 `User` 模型的 `ForeignKey` ，而且这个字段必须要叫做 `user` 。其他的字段可以由你自己掌控。下面是一个档案模型的例子：

```
from django.db import models
from django.contrib.auth.models import User

class MySiteProfile(models.Model):
    # This is the only required field
    user = models.ForeignKey(User, unique=True)

    # The rest is completely up to you...
    favorite_band = models.CharField(maxlength=100, blank=True)
    favorite_cheese = models.CharField(maxlength=100, blank=True)
    lucky_number = models.IntegerField()
```

下一步，你需要告诉Django去哪里查找档案对象。你可以通过设置模型中的 `AUTH_PROFILE_MODULE` 变量达到这个目

的。因此，如果你的模型包含在 myapp 这个应用中，你就需要如下编写你的设置文件：

```
AUTH_PROFILE_MODULE = "myapp.mysiteprofile"
```

一旦完成，你就可以通过调用 user.get_profile() 函数来获得用户档案。如果 AUTH_PROFILE_MODULE 变量没有设置，这个函数可能会抛出 SiteProfileNotAvailable 异常；如果这个用户不存在档案，也可能会抛出 DoesNotExist 异常（通常情况下，你会捕获这个异常并在当时创建一个新的档案）。

接下来?

是的，会话和认证系统有太多的东西要学。大多数情况下，你并不需要本章所提到的所有功能。然而当你需要允许用户之间复杂的互操作时，所有的功能都能使用就显得很重要了。

在下一章节中，我们会来深入了解Django建立在会话/用户系统之上的一个系统：评论应用。它允许你很方便地以匿名用户或者注册用户的身份，向任意类型的对象添加评论。让我们继续向前吧。

第十三章 缓存机制

静态的网站的内容都是些简单的静态网页直接存储在服务器上，可以非常容易地达到非常惊人的访问量。但是动态网站因为是动态的，也就是说每次用户访问一个页面，服务器要执行数据库查询，启动模板，执行业务逻辑到最终生成一个你所看到的网页，这一切都是动态即时生成的。从处理器资源的角度来看，这是比较昂贵的。

对于大多数网络应用来说，过载并不是大问题。因为大多数网络应用并不是washingtonpost.com或Slashdot；它们通常是很小很简单，或者是中等规模的站点，只有很少的流量。但是对于中等至大规模流量的站点来说，尽可能地解决过载问题是非常必要的。这就需要用到缓存了。

缓存的目的是为了避免重复计算，特别是对一些比较耗时间、资源的计算。下面的伪代码演示了如何对动态页面的结果进行缓存。

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

为此，Django提供了一个稳定的缓存系统让你缓存动态页面的结果，这样在接下来有相同的请求就可以直接使用缓存中的数据，避免不必要的重复计算。另外Django还提供了不同粒度数据的缓存，例如：你可以缓存整个页面，也可以缓存某个部分，甚至缓存整个网站。

Django也和"上游"缓存工作的很好，例如Squid(http://www.squid-cache.org)和基于浏览器的缓存，这些类型的缓存你不直接控制，但是你可以提供关于你的站点哪部分应该被缓存和怎样缓存的线索(通过HTTP头部)给它们

继续阅读来研究如何使用Django的缓存系统。当你的网站变成象Slashdot的时候，你会很很高兴理解了这部分材料

设定缓存

缓存系统需要一些少量的设定工作，即你必需告诉它你的缓存数据在哪里—在数据库，文件系统或者直接在内存中，这是影响你的缓存性能的重要决定，是的，一些缓存类型要比其它的快，内存缓存通常比文件系统或数据库缓存快，因为前者没有访问文件系统或数据库的过度连接

你的缓存选择在你的settings文件的 `CACHE_BACKEND` 设置中，如果你使用缓存但没有指定 `CACHE_BACKEND` ，Django将默认使用 `simple:///` ，下面将解释 `CACHE_BACKEND` 的所有可得到的值

内存缓冲

目前为止Django可得到的最快的最高效的缓存类型是基于内存的缓存框架Memcached，它起初开发来为LiveJournal.com处理高负荷并随后被Danga Interactive(http://www.danga.com)开源，它被Slashdot和Wikipedia等站点采用以减少数据库访问并极大的提升了站点性能

Memcached可以在http://danga.com/memcached/免费得到，它作为后台进程运行并分配一个指定数量的RAM.它能为你提供在缓存中*如闪电般快速的*添加，获取和删除任意数据，所有的数据直接存储在内存中，所以没有数据库和文件系统使用的过度使用

在安装了Memcached本身之后，你将需要安装Memcached Python绑定，它没有直接和Django绑定，这些绑定在一个单独的Python模块中，'memcache.py'，可以在http://www.djangoproject.com/thirdparty/python-memcached得到

设置 `CACHE_BACKEND` 为 `memcached://ip:port/` 来让Django使用Memcached，这里的 `ip` 是Memcached后台进程的IP地址， `port` 则是Memcached运行所在的端口

在这个例子中，Memcached运行在本地主机 (127.0.0.1)上,端口为11211：

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Memcached的一个极好的特性是它在多个服务器分享缓存的能力，这意味着你可以在多台机器上运行Memcached进程，程序将会把这组机器当作一个*单独的*缓存，而不需要在每台机器上复制缓存值，为了让Django利用此特性，需要在CACHE_BACKEND里包含所有的服务器地址并用分号分隔

这个例子中，缓存在运行在172.19.26.240和172.19.26.242的IP地址和11211端口的Memcached实例间分享:

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

这个例子中，缓存在运行在172.19.26.240(端口11211)，172.19.26.242(端口11212)，172.19.26.244(端口11213)的Memcached实例间分享:

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11212;172.19.26.244:11213/'
```

最后关于Memcached的是基于内存的缓存有一个重大的缺点，因为缓存数据只存储在内存中，则如果服务器死机的话数据会丢失，显然内存不是为持久数据存储准备的，Django没有一个缓存后端是用来做持久存储的，它们都是缓存方案，而不是存储.但是我们在这里指出是因为基于内存的缓存特别的短暂 .

数据库缓存

为了将数据库表作为缓存后端，需要在数据库中创建一个缓存表并将Django的缓存系统指向该表

首先,使用如下语句创建一个缓存用数据表:

```
python manage.py createcachetable [cache_table_name]
```

这里的[cache_table_name]是要创建的数据库表名，名字可以是任何你想要的，只要它是合法的在你的数据库中没有被使用，这个命令在你的数据库创建一个遵循Django的数据库缓存系统期望形式的单独的表.

一旦你创建了数据库表，设置你的CACHE_BACKEND设置为"db://tablename"，这里的tablename是数据库表的名字，在这个例子中，缓存表名为my_cache_table:

```
CACHE_BACKEND = 'db://my_cache_table'
```

数据库缓存后端使用你的settings文件指定的同一数据库，你不能为你的缓存表使用不同的数据库后端.

文件系统缓存

使用"file://"缓存类型作为CACHE_BACKEND并指定存储缓存数据的文件系统目录来在文件系统存储缓存数据.

例如，使用下面的设置来在/var/tmp/django_cache存储缓存数据:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

注意例子中开头有三个前斜线，前两个是file://，第三个是目录路径的第一个字符，/var/tmp/django_cache，如果你使用Windows系统，把盘符字母放在file://后面，像这样:'file://c:/foo/bar'.

目录路径应该是*绝对*路径，即应该以你的文件系统的根开始，你在设置的结尾放置斜线与否无关紧要.

确认该设置指向的目录存在并且你的Web服务器运行的系统的用户可以读写该目录，继续上面的例子，如果你的服务器以用户apache运行，确认/var/tmp/django_cache存在并且用户apache可以读写/var/tmp/django_cache目录

每个缓存值将被存储为单独的文件，其内容是Python的pickle模块以序列化("pickled")形式保存的缓存数据，每个文件的文件名是缓存键，以规避开安全文件系统的使用

本地内存缓存

如果你想要内存缓存的速度优势但没有能力运行Memcached，可以考虑使用本地存储器缓存后端，该缓存是多线程和线程安全 的，但是由于其简单的锁和内存分配策略它没有Memcached高效

设置 `CACHE_BACKEND` 为 `locmem:///` 来使用它，例如:

```
CACHE_BACKEND = 'locmem:///'
```

简易缓存（用于开发阶段）

可以通过配置 `'simple:///'` 来使用一个简单的单进程内存缓存，例如：

```
CACHE_BACKEND = 'simple:///'
```

这个缓存仅仅是将数据保存在进程内，因此它应该只在开发环境或测试环境中使用.

仿缓存（供开发时使用）

最后，Django提供一个假缓存的设置：它仅仅实现了缓存的接口而不做任何实际的事情

这是个有用的特性，如果你的线上站点使用了很多比较重的缓存，而在开发环境中却不想使用缓存，那么你只要修改配置文件，将 `CACHE_BACKEND` 设置为 `'dummy:///'` 就可以了，例如：

```
CACHE_BACKEND = 'dummy:///'
```

这样的结果就是你的开发环境没有使用缓存，而线上环境依然在使用缓存.

CACHE_BACKEND参数

每个缓存后端都可能使用参数，它们在CACHE_BACKEND设置中以查询字符串形式给出，合法的参数为:

`timeout`:用于缓存的过期时间，以秒为单位。这个参数默认被设置为300秒（五分钟）

> `max_entries` : 对于simple, local-memory与database类型的缓存,这个参数是指定缓存中存放的最大条目数，大于这个数时，旧的条目将会被删除。这个参数默认是300.
>
> `cull_frequency` :当达到 `max_entries` 的时候,被接受的访问的比率。实际的比率是 `1/cull_frequency` ,所以设置cull_frequency=2就是在达到 `max_entries` 的时候去除一半数量的缓存
>
> 把 `cull_frequency` 的值设置为 `0` 意味着当达到 `max_entries` 时,缓存将被清空。这将以很多缓存丢失为代价,大大提高接受访问的速度。这个值默认是3

在这个例子中， `timeout` 被设成 `60`

```
CACHE_BACKEND = "locmem:///?timeout=60"
```

而在这个例子中， `timeout` 设为 `30` 而 `max_entries` 为 `400`：

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

其中，非法的参数与非法的参数值都将被忽略。

站点级 Cache

一旦你指定了"CACHE_BACKEND"，使用缓存的最简单的方法就是缓存你的整个网站。这意味着所有不包含GET或POST参数的页面在第一次被请求之后将被缓存指定好的一段时间（就是设置的timeout参数）。

要激活每个站点的cache，只要将```django.middleware.cache.CacheMiddleware``` 添加到 `MIDDLEWARE_CLASSES` 的设置里，就像下面这样：

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.CacheMiddleware',
    'django.middleware.common.CommonMiddleware',
)
```

注意

关于 `MIDDLEWARE_CLASSES` 顺序的一些事情。请看本章节后面的MIDDLEWARE_CLASSES顺序部分。

然后，在你的Django settings文件里加入下面所需的设置：

`CACHE_MIDDLEWARE_SECONDS` :每个页面应该被缓存的秒数

- `CACHE_MIDDLEWARE_KEY_PREFIX` ：如果缓存被多个使用相同Django安装的网站所共享，那么把这个值设成当前网站名，或其他能代表这个Django实例的唯一字符串，以避免key发生冲突。如果你不在意的话可以设成空字符串。

缓存中间件缓存每个没有GET或者POST参数的页面，即如果用户请求页面并在查询字符串里传递GET参数或者POST参数，中间件将不会尝试得到缓存版本的页面，如果你打算使用整站缓存，设计你的程序时牢记这点，例如，不要使用拥有查询字符串的URLs，除非那些页面可以不缓存

缓存中间件（ cache middleware）支持另外一种设置选项, `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` 。如果你把它设置为“True”,那么缓存中间件就只会对匿名请求进行缓存， 匿名请求是指那些 没有登录的用户发起的请求。如果想取消用户相关页面（user-specific pages）的缓存，例如Djangos 的管理界面，这是一种既简单又有效的方法。另外，如果你要使用 `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` 选项，你必须先激活 `AuthenticationMiddleware` 才行，也就是在你的配置文件 `MIDDLEWARE_CLASSES` 的地方， `AuthenticationMiddleware` 必须出现在 `CacheMiddleware` 前面。

最后，再提醒一下： `CacheMiddleware` 在每个 `HttpResponse` 中都会自动设置一些头部信息（headers）

- 当一个新(没缓存的)版本的页面被请求时设置Last-Modified头部为当前日期/时间
- 设置Expires头部为当前日期/时间加上定义的CACHE_MIDDLEWARE_SECONDS
- 设置Cache-Control头部来给页面一个最大的时间—再一次，根据CACHE_MIDDLEWARE_SECONDS设置

视图级缓存

更加颗粒级的缓存框架使用方法是对单个视图的输出进行缓存。这和整站级缓存有一样的效果（包括忽略对有 GET 和 POST 参数的请求的缓存）。它应用于你所指定的视图，而不是整个站点。

完成这项工作的方式是使用 *修饰器* ，其作用是包裹视图函数，将其行为转换为使用缓存。视图缓存修饰器称为 `cache_page` ，位于 `django.views.decorators.cache` 模块中，例如：

```
from django.views.decorators.cache import cache_page

def my_view(request, param):
    # ...
my_view = cache_page(my_view, 60 * 15)
```

It’s not my first time to go to see this website, i am visiting this web page dailly and obtain nice facts from here daily.

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request, param):
    # ...
```

`cache_page` 只接受一个参数：以秒计的缓存超时。在前例中， “my_view()” 视图的结果将被缓存 15 分钟。（注意：为了提高可读性，该参数被书写为 `60 * 15` 。 `60 * 15` 将被计算为 `900` ，也就是说15 分钟乘以每分钟 60 秒。）

和站点缓存一样，视图缓存与 URL 无关。如果多个 URL 指向同一视图，每个视图将会分别缓存。继续 `my_view` 范例，如果 URLconf 如下所示：

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

那么正如你所期待的那样，发送到 `/foo/1/` 和 `/foo/23/` 的请求将会分别缓存。但一旦发出了特定的请求（如： `/foo/23/` ），之后再度发出的指向该 URL 的请求将使用缓存。

在 URLconf 中指定视图缓存

前一节中的范例将视图硬编码为使用缓存，因为 cache_page 在适当的位置对 my_view 函数进行了转换。该方法将视图与缓存系统进行了耦合，从几个方面来说并不理想。例如，你可能想在某个无缓存的站点中重用该视图函数，或者你可能想将该视图发布给那些不想通过缓存使用它们的人。解决这些问题的方法是在 URLconf 中指定视图缓存，而不是紧挨着这些视图函数本身来指定。

完成这项工作非常简单：在 URLconf 中用到这些视图函数的时候简单地包裹一个 cache_page 。以下是刚才用到过的 URLconf：

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

以下是同一个 URLconf，不过用 cache_page 包裹了 my_view：

```
from django.views.decorators.cache import cache_page

urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', cache_page(my_view, 60 * 15)),
)
```

如果采取这种方法, 不要忘记在 URLconf 中导入 cache_page .

低层次缓存API

有些时候，对整个经解析的页面进行缓存并不会给你带来太多，事实上可能会过犹不及。

比如说，也许你的站点所包含的一个视图依赖几个费时的查询，每隔一段时间结果就会发生变化。在这种情况下，使用站点级缓存或者视图级缓存策略所提供的整页缓存并不是最理想的，因为你可能不会想对整个结果进行缓存（因为一些数据经常变化），但你仍然会想对很少变化的部分进行缓存。

在像这样的情形下， Django 展示了一种位于 django.core.cache 模块中的简单、低层次的缓存 API 。你可以使用这种低层次的缓存 API 在缓存中以任何级别粒度进行对象储存。你可以对所有能够安全进行 pickle 处理的 Python 对象进行缓存：字符串、字典和模型对象列表等等；查阅 Python 文档可以了解到更多关于 pickling 的信息。）

下面是如何导入这个 API :

```
>>> from django.core.cache import cache
```

基本的接口是 set(key, value, timeout_seconds) 和 get(key)：

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

timeout_seconds 参数是可选的, 并且默认为前面讲过的 CACHE_BACKEND 设置中的 timeout 参数.

如果对象在缓存中不存在, 或者缓存后端是不可达的, cache.get() 返回 None :

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_key')
None

>>> cache.get('some_unset_key')
None
```

我们不建议在缓存中保存 None 常量，因为你将无法区分所保存的 None 变量及由返回值 None 所标识的缓存未中。

cache.get() 接受一个 缺省 参数。其指定了当缓存中不存在该对象时所返回的值：

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

要想一次获取多个缓存值，可以使用 cache.get_many() 。如果可能的话，对于给定的缓存后端， get_many() 将只访

问缓存一次，而不是对每个缓存键值都进行一次访问。 `get_many()` 所返回的字典包括了你所请求的存在于缓存中且未超时的所有键值。

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

如果某个缓存关键字不存在或者已超时, 它将不会被包含在字典中。 下面是范例的延续:

```
>>> cache.get_many(['a', 'b', 'c', 'd'])
{'a': 1, 'b': 2, 'c': 3}
```

最后,你可以用 `cache.delete()` 显式地删除关键字。这是在缓存中清除特定对象的简单途径。

```
>>> cache.delete('a')
```

> `cache.delete()` 没有返回值, 不管给定的缓存关键字对应的值存在与否, 它都将以同样方式工作。

上游缓存

目前为止，本章的焦点一直是对你 *自己的* 数据进行缓存。但还有一种与 Web 开发相关的缓存：由 *上游* 高速缓存执行的缓冲。有一些系统甚至在请求到达站点之前就为用户进行页面缓存。

下面是上游缓存的几个例子：

- 你的 ISP (互联网服务商)可能会对特定的页面进行缓存，因此如果你向 http://example.com/ 请求一个页面，你的 ISP 可能无需直接访问 example.com 就能将页面发送给你。而 example.com 的维护者们却无从得知这种缓存，ISP 位于 example.com 和你的网页浏览器之间，透明地处理所有的缓存。
- 你的 Django 网站可能位于某个 *代理缓存* 之后，例如 Squid 网页代理缓存 (http://www.squid-cache.org/)，该缓存为提高性能而对页面进行缓存。在此情况下 ，每个请求将首先由代理服务器进行处理，然后仅在需要的情况下才被传递至你的应用程序。
- 你的网页浏览器也对页面进行缓存。如果某网页送出了相应的头部，你的浏览器将在为对该网页的后续的访问请求使用本地缓存的拷贝，甚至不会再次联系该网页查看是否发生了变化。

上游缓存将会产生非常明显的效率提升，但也存在一定风险。许多网页的内容依据身份验证以及许多其他变量的情况发生变化，缓存系统仅盲目地根据 URL 保存页面，可能会向这些页面的后续访问者暴露不正确或者敏感的数据。

举个例子，假定你在使用网页电邮系统，显然收件箱页面的内容取决于登录的是哪个用户。如果 ISP 盲目地缓存了该站点，那么第一个用户通过该 ISP 登录之后，他（或她）的用户收件箱页面将会缓存给后续的访问者。这一点也不好玩。

幸运的是， HTTP 提供了解决该问题的方案。已有一些 HTTP 头标用于指引上游缓存根据指定变量来区分缓存内容，并通知缓存机制不对特定页面进行缓存。我们将在本节后续部分将对这些头标进行阐述。

使用 Vary 头标

> `Vary` 头标定义了缓存机制在构建其缓存键值时应当将哪个请求头标考虑在内。例如，如果网页的内容取决于用户的语言偏好，该页面被称为根据语言而不同。

缺省情况下，Django 的缓存系统使用所请求的路径（比如： `"/stories/2005/jun/23/bank_robbed/"` ）来创建其缓存键。这意味着对该 URL 的每个请求都将使用同一个已缓存版本，而不考虑 cookies 或语言偏好之类的 user-agent 差别。然而，如果该页面基于请求头标的区别（例如 cookies、语言或者 user-agent)产生不同内容，你就不得不使用

> `Vary` 头标来通知缓存机制：该页面的输出取决于这些东西。

要在 Django 完成这项工作，可使用便利的 `vary_on_headers` 视图修饰器，如下所示：

```
from django.views.decorators.vary import vary_on_headers
```

```
# Python 2.3 syntax.
def my_view(request):
    # ...
my_view = vary_on_headers(my_view, 'User-Agent')

# Python 2.4+ decorator syntax.
@vary_on_headers('User-Agent')
def my_view(request):
    # ...
```

在这种情况下，缓存装置（如 Django 自己的缓存中间件）将会为每一个单独的用户浏览器缓存一个独立的页面版本。

使用 `vary_on_headers` 修饰器而不是手动设置 `Vary` 头标（使用像 `response['Vary'] = 'user-agent'` 之类的代码）的好处是修饰器在（可能已经存在的） `Vary` 之上进行 *添加*，而不是从零开始设置，且可能覆盖该处已经存在的设置。

你可以向 `vary_on_headers()` 传入多个头标：

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

该段代码通知上游缓存对 *两者* 都进行不同操作，也就是说 user-agent 和 cookie 的每种组合都应获取自己的缓存值。举例来说，使用 `Mozilla` 作为 user-agent 而 `foo=bar` 作为 cookie 值的请求应该和使用 `Mozilla` 作为 user-agent 而 `foo=ham` 的请求应该被视为不同请求。

由于根据 cookie 而区分对待是很常见的情况，因此有 `vary_on_cookie` 修饰器。以下两个视图是等效的：

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

传入 `vary_on_headers` 头标是大小写不敏感的； `"User-Agent"` 与 `"user-agent"` 完全相同。

你也可以直接使用帮助函数： `django.utils.cache.patch_vary_headers` 。该函数设置或增加 `Vary header` ，例如：

```
from django.utils.cache import patch_vary_headers

def my_view(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

`patch_vary_headers` 以一个 `HttpResponse` 实例为第一个参数，以一个大小写不敏感的头标名称列表或元组为第二个参数。

其它缓存头标

关于缓存剩下的问题是数据的私隐性以及关于在级联缓存中数据应该在何处储存的问题。

通常用户将会面对两种缓存：他或她自己的浏览器缓存（私有缓存）以及他或她的提供者缓存（公共缓存）。公共缓存由多个用户使用，而受其他某人的控制。这就产生了你不想遇到的敏感数据的问题，比如说你的银行账号被存储在公众缓存中。因此，Web 应用程序需要以某种方式告诉缓存那些数据是私有的，哪些是公共的。

解决方案是标示出某个页面缓存应当是私有的。要在 Django 中完成此项工作，可使用 `cache_control` 视图修饰器：

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

该修饰器负责在后台发送相应的 HTTP 头标。

还有一些其他方法可以控制缓存参数。例如, HTTP 允许应用程序执行如下操作:

- 定义页面可以被缓存的最大次数。
- 指定某个缓存是否总是检查较新版本，仅当无更新时才传递所缓存内容。（一些缓存即便在服务器页面发生变化的情况下都可能还会传送所缓存的内容，只因为缓存拷贝没有过期。）

在 Django 中，可使用 `cache_control` 视图修饰器指定这些缓存参数。在本例中， `cache_control` 告诉缓存对每次访问都重新验证缓存并在最长 3600 秒内保存所缓存版本：

```
from django.views.decorators.cache import cache_control
@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    ...
```

在 `cache_control()` 中，任何有效 `Cache-Control` HTTP 指令都是有效的。以下是一个完整的清单：

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

小提示

要了解有关 `Cache-Control` HTTP 指令的相关解释, 可以查阅 http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9 的规范文档。

注意

缓存中间件已经使用 `CACHE_MIDDLEWARE_SETTINGS` 设置设定了缓存头标 `max-age` 。如果你在 `cache_control` 修饰器中使用了自定义的 `max_age` ，该修饰器将会取得优先权，该头标的值将被正确地被合并。）

其他优化

Django 带有一些其它中间件可帮助您优化应用程序的性能:

- `django.middleware.http.ConditionalGetMiddleware` 为现代浏览器增加了有条件地 GET 基于 `ETag` 和 `Last-Modified` 头标的响应的相关支持。
- `django.middleware.gzip.GZipMiddleware` 为所有现代浏览器压缩响应内容，以节省带宽和传送时间。

MIDDLEWARE_CLASSES 的顺序

如果使用缓存中间件，一定要将其放置在 MIDDLEWARE_CLASSES 设置的正确位置，因为缓存中间件需要知道用于产生不同缓存存储的的头标。

将 `CacheMiddleware` 放置在所有可能向 `Vary` 头标添加内容的中间件之后，包括下列中间件：

- 添加 `Cookie` 的 `SessionMiddleware`
- 添加 `Accept-Encoding` 的 `GZipMiddleware`，

接下来？

Django 带有一些功能包装了一些很酷的,可选的特色. 我们已经讲了一些: admin系统(第6章)和session/user框架(第11章).

下一章中，我们将讲述Django中其他的子框架，将会有很多很酷的工具出现，你一定不想错过它们。

第十四章 集成的子框架

ifRqur boillucronlx, [url=http://pfkayzbowvrx.com/]pfkayzbowvrx[/url], [link=http://onajxszobhba.com/]onajxszobhba[/link], http://spfvpxdqczbk.com/

Django标准库

Django的标准库存放在 `django.contrib` 包中。每个子包都是一个独立的附加功能包。 它们互相之间一般没有必然的关联，但是有些 `django.contrib` 子包可能依赖其他的包。

在 `django.contrib` 中对函数的类型并没有强制要求 。其中一些包中带有模型（因此需要你在数据库中安装对应的数据表），但其它一些由独立的中间件及模板标签组成。

`django.contrib` 开发包共有的特性是:就算你将整个 `django.contrib` 开发包删除，你依然可以使用 Django 的基础功能而不会遇到任何问题。当 Django 开发者向框架增加新功能的时，他们会严格根据这一教条来决定是否把新功能放入 `django.contrib` 中。

`django.contrib` 由以下开发包组成：

- `admin` : 自动化的站点管理工具。请查看第6章和第18章
- `auth` : Django的用户验证框架。请查看第12章
- `comments` : 一个评论应用，目前，这个应用正在紧张的开发中，因此在本书出版的时候还不能给出一个完整的说明，关于这个应用的更多信息请参见Django的官方网站.
- `contenttypes` : 这是一个用于文档类型钩子的框架，每个安装的Django模块作为一种独立的文档类型。这个框架主要在Django内部被其他应用使用，它主要面向Django的高级开发者。可以通过阅读源码来了解关于这个框架的更多信息，源码的位置在 `django/contrib/contenttypes/` .
- `csrf` : 这个模块用来防御跨站请求伪造(CSRF).参见后面标题为"CSRF 防御"的小节。
- `flatpages` : 一个在数据库中管理单一HTML内容的模块，参见后面标题为"Flatpages"的小节。
- `humanize` : 一系列 Django 模块过滤器，用于增加数据的人性化。参阅稍后的章节《人性化数据》。
- `markup` : 一系列的 Django 模板过滤器，用于实现一些常用标记语言。参阅后续章节《标记过滤器》。
- `redirects` : 用来管理重定向的框架。参见后面标题为《重定向》的小节。
- `sessions` : Django 的会话框架，参见12章。
- `sitemaps` : 用来生成网站地图的 XML 文件的框架。参见 11 章。
- `sites` : 一个让你可以在同一个数据库与 Django 安装中管理多个网站的框架。参见下一节：站点。
- `syndication` : 一个用 RSS 和 Atom 来生成聚合订阅源的的框架。参阅第 11 章。

本章接下来将详细描述前面没有介绍过的 `django.contrib` 开发包内容。

多个站点

Django 的多站点系统是一种通用框架，它让你可以在同一个数据库和同一个Django项目下操作多个网站。这是一个抽象概念，理解起来可能有点困难，因此我们从几个让它能派上用场的实际情景入手。

情景1：对多个站点重用数据

正如我们在第一章里所讲，Django 构建的网站 LJWorld.com 和 Lawrance.com 是用由同一个新闻组织控制的：肯萨斯州劳伦斯市的 *劳伦斯日报世界* 报纸。 LJWorld.com 主要做新闻，而 Lawrence.com 关注本地娱乐。然而有时，编辑可能需要把一篇文章发布到 *两个* 网站上。

解决此问题的死脑筋方法可能是使用每个站点分别使用不同的数据库，然后要求站点维护者把同一篇文章发布两次：一次为 LJWorld.com，另一次为Lawrence.com。但这对站点管理员来说是低效率的，而且为同一篇文章在数据库里保留多个副本也显得多余。

更好的解决方案？两个网站用的是同一个文章数据库，并将每一篇文章与一个或多个站点用多对多关系关联起来。Django 站点框架提供数据库记载哪些文章可以被关联。它是一个把数据与一个或多个站点关联起来的钩子。

情景2：把你的网站名称/域存储到唯一的位置

LJWorld.com 和 Lawrence.com 都有邮件提醒功能，使读者注册后可以在新闻发生后立即收到通知。这是一种完美的的机制：某读者提交了注册表单，然后马上就受到一封内容是“感谢您的注册”的邮件。

把这个注册过程的代码实现两遍显然是低效、多余的，因此两个站点在后台使用相同的代码。但感谢注册的通知在两个网站中需要不同。通过使用 `Site` 对象，我们通过使用当前站点的 `name` (例如 `'LJWorld.com'`)和 `domain` (例如 `'www.ljworld.com'`)可以把感谢通知抽提出来。

Django 的多站点框架为你提供了一个位置来存储 Django 项目中每个站点的 `name` 和 `domain` ，这意味着你可以用同样的方法来重用这些值。

如何使用多站点框架

多站点框架与其说是一个框架，不如说是一系列约定。所有的一切都基于两个简单的概念：

- 位于 `django.contrib.sites` 的 `Site` 模型有 `domain` 和 `name` 两个字段。
- `SITE_ID` 设置指定了与特定配置文件相关联的 `Site` 对象之数据库 ID。

如何运用这两个概念由你决定，但 Django 是通过几个简单的约定自动使用的。

安装多站点应用要执行以下几个步骤：

1. 将 `'django.contrib.sites'` 加入到 `INSTALLED_APPS` 中。
2. 运行 *manage.py syncdb* 命令将 *django_site* 表安装到数据库中。
3. 通过 Django 管理后台或通过 Python API 添加一个或者多个 ‘Site’ 对象。为该 Django 项目支撑的每个站（或域）创建一个 `Site` 对象。
4. 在每个设置文件中定义一个 `SITE_ID` 变量。该变量值应当是该设置文件所支撑的站点之 `Site` 对象的数据库 ID 。

多站点框架的功能

下面几节讲述的是用多站点框架能够完成的几项工作。

多个站点的数据重用

正如在情景一中所解释的，要在多个站点间重用数据,仅需在模型中为 `Site` 添加一个 `多对多字段` 即可，例如：

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    sites = models.ManyToManyField(Site)
```

这是在数据库中为多个站点进行文章关联操作的基础步骤。在适当的位置使用该技术，你可以在多个站点中重复使用

同一段 Django 视图代码。继续 `Article` 模型范例，下面是一个可能的 `article_detail` 视图：

```
from django.conf import settings

def article_detail(request, article_id):
    try:
        a = Article.objects.get(id=article_id, sites__id=settings.SITE_ID)
    except Article.DoesNotExist:
        raise Http404
    # ...
```

该视图方法是可重用的，因为它根据 `SITE_ID` 设置的值动态检查 articles 站点。

例如， LJWorld.coms 设置文件中有有个 `SITE_ID` 设置为 `1` ，而 Lawrence.coms 设置文件中有个 `SITE_ID` 设置为 `2` 。如果该视图在 LJWorld.coms 处于激活状态时被调用，那么它将把查找范围局限于站点列表包括 LJWorld.com 在内的文章。

将内容与单一站点相关联

同样，你也可以使用 外键 在多对一关系中将一个模型关联到 `Site` 模型。

举例来说，如果某篇文章仅仅能够出现在一个站点上，你可以使用下面这样的模型：

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    site = models.ForeignKey(Site)
```

这与前一节中介绍的一样有益。

从视图钩挂当前站点

在底层，通过在 Django 视图中使用多站点框架，你可以让视图根据调用站点不同而完成不同的工作，例如：

```
from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.
```

当然，像那样对站点 ID 进行硬编码是比较难看的。略为简洁的完成方式是查看当前的站点域：

```
from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

从 `Site` 对象中获取 `settings.SITE_ID` 值的做法比较常见，因此 `Site` 模型管理器 (`Site.objects`) 具备一个 `get_current()` 方法。下面的例子与前一个是等效的：

```
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
```

```
    else:
        # Do something else.
```

注意

在这个最后的例子里，你不用导入 `django.conf.settings` 。

获取当前域用于呈现

正如情景二中所解释的那样，对于储存站名和域名的 DRY (Dont Repeat Yourself) 方法（在一个位置储存站名和域名）来说，只需引用当前 `Site` 对象的 `name` 和 `domain` 。例如：

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % current_site.name,
        'editor@%s' % current_site.domain,
        [user_email])
    # ...
```

继续我们正在讨论的 LJWorld.com 和 Lawrence.com 例子，在Lawrence.com 该邮件的标题行是“感谢注册 Lawrence.com 提醒信件”。在 LJWorld.com，该邮件标题行是“感谢注册 LJWorld.com 提醒信件”。这种站点关联行为方式对邮件信息主体也同样适用。

完成这项工作的一种更加灵活（但重量级也更大）的方法是使用 Django 的模板系统。假定 Lawrence.com 和 LJWorld.com 各自拥有不同的模板目录（ `TEMPLATE_DIRS` ），你可将工作轻松地转交给模板系统，如下所示：

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'do-not-reply@example.com', [user_email])
    # ...
```

本例中，你不得不在 LJWorld.com 和 Lawrence.com 的模板目录中都创建一份 `subject.txt` 和 `message.txt` 模板。正如之前所说，该方法带来了更大的灵活性，但也带来了更多复杂性。

尽可能多的利用 `Site` 对象是减少不必要的复杂、冗余工作的好办法。

获取当前域的完整 URL

Django 的 `get_absolute_url()` 约定对与获取不带域名的对象 URL 非常理想，但在某些情形下，你可能想显示某个对象带有 `http://` 和域名以及所有部分的完整 URL 。要完成此工作，你可以使用多站点框架。下面是个简单的例子：

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

当前站点管理器

如果 站点 在你的应用中扮演很重要的角色，请考虑在你的模型中使用方便的 `CurrentSiteManager` 。这是一个模型管

理器（见附录B），它会自动过滤使其只包含与当前 站点 相关联的对象。

通过显示地将 CurrentSiteManager 加入模型中以使用它。例如：

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

通过该模型， Photo.objects.all() 将返回数据库中所有的 Photo 对象，而 Photo.on_site.all() 仅根据 SITE_ID 设置返回与当前站点相关联的 Photo 对象。

换言之，以下两条语句是等效的：

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

> CurrentSiteManager 是如何知道 Photo 的哪个字段是 Site 呢？缺省情况下，它会查找一个叫做 site 的字段。如果模型中有个 外键 或 多对多字段 叫做 site *之外* 的名字，你必须显示地将它作为参数传递给 CurrentSiteManager 。下面的模型中有个叫做 publish_on 的字段，如下所示：

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

如果试图使用 CurrentSiteManager 并传入一个不存在的字段名， Django 将引发一个 ValueError 异常。

注意事项

即便是已经使用了 CurrentSiteManager ，你也许还想在模型中拥有一个正常的（非站点相关）的 管理器 。正如在附录B 中所解释的，如果你手动定义了一个管理器，那么 Django 不会为你创建全自动的 objects = models.Manager() 管理器。

同样，Django 的特定部分——即 Django 超级管理站点和通用视图——使用的管理器 *首先* 在模型中定义，因此如果希望超级管理站点能够访问所有对象（而不是仅仅站点特有对象），请于定义 CurrentSiteManager 之前在模型中放入 objects = models.Manager() 。

Django如何使用多站点框架

尽管并不是必须的，我们还是强烈建议使用多站点框架，因为 Django 在几个地方利用了它。即使只用 Django 来支持单个网站，你也应该花一点时间用 domain 和 name 来创建站点对象，并将 SITE_ID 设置指向它的 ID 。

以下讲述的是 Django 如何使用多站点框架：

- 在重定向框架中（见后面的重定向一节），每一个重定向对象都与一个特定站点关联。当 Django 搜索重定向的时候，它会考虑当前的 SITE_ID 。
- 在注册框架中，每个注释都与特定站点相关。每个注释被张贴时，其 site 被设置为当前的 SITE_ID ，而当通过适当的模板标签列出注释时，只有当前站点的注释将会显示。

- 在 flatpages 框架中 (参见后面的 Flatpages 一节），每个 flatpage 都与特定的站点相关联。创建 flatpage 时，你都将指定它的 `site`，而 flatpage 中间件在获取 flatpage 以显示它的过程中，将查看当前的 `SITE_ID`。
- 在 syndication 框架中（参阅第 11 章）， `title` 和 `description` 的模板自动访问变量 `{{ site }}`，它就是代表当前着桨的 `Site` 对象。而且，如果你不指出一个完全合格的domain的话，提供目录URLS的钩子将会使用当前“Site”对象的domain。
- 在身份验证框架（参见第 12 章）中， `django.contrib.auth.views.login` 视图将当前 `Site` 名称作为 `{{ site_name }}` 传递给模板。

Flatpages - 简单页面

尽管通常情况下总是建造和运行数据库驱动的 Web 应用，你还是会需要添加一两张一次性的静态页面，例如“关于”页面，或者“隐私策略”页面等等。可以用像 Apache 这样的标准Web服务器来处理这些静态页面，但却会给应用带来一些额外的复杂性，因为你必须操心怎么配置 Apache，还要设置权限让整个团队可以修改编辑这些文件，而且你还不能使用 Django 模板系统来统一这些页面的风格。

这个问题的解决方案是使用位于 `django.contrib.flatpages` 开发包中的 Django 简单页面（flatpages）应用程序。该应用让你能够通过 Django 超级管理站点来管理这些一次性的页面，还可以让你使用 Django 模板系统指定它们使用哪个模板。它在后台使用了 Django 模型，也就是说它将页面存放在数据库中，你也可以像对待其他数据一样用标准 Django 数据库 API 存取简单页面。

简单页面以它们的 URL 和站点为键值。当创建简单页面时，你指定它与哪个URL以及和哪个站点相关联 。（有关站点的更多信息，请查阅《站点》一节）

使用简单页面

安装简单页面应用程序必须按照下面的步骤：

1. 添加 `'django.contrib.flatpages'` 到 `INSTALLED_APPS` 设置。 `django.contrib.flatpages` 依赖于 `django.contrib.sites`，所以确保这两个开发包都包括在 `INSTALLED_APPS` 设置中。
2. 将 `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` 添加到 `MIDDLEWARE_CLASSES` 设置中。
3. 运行 `manage.py syncdb` 命令在数据库中创建必需的两个表。

简单页面应用程序在数据库中创建两个表： `django_flatpage` 和 `django_flatpage_sites`。 `django_flatpage` 只是将 URL 映射到标题和一段文本内容。 `django_flatpage_sites` 是一个多对多表，用于关联某个简单页面以及一个或多个站点。

该应用所带来的 `FlatPage` 模型在 `django/contrib/flatpages/models.py` 进行定义，如下所示：

```
from django.db import models
from django.contrib.sites.models import Site

class FlatPage(models.Model):
    url = models.CharField(maxlength=100)
    title = models.CharField(maxlength=200)
    content = models.TextField()
    enable_comments = models.BooleanField()
    template_name = models.CharField(maxlength=70, blank=True)
    registration_required = models.BooleanField()
    sites = models.ManyToManyField(Site)
```

让我们逐项看看这些字段的含义：

- `url`：该简单页面所处的 URL，不包括域名，但是包含前导斜杠 (例如 `/about/contact/`)。
- `title`：简单页面的标题。框架不对它作任何特殊处理。由你通过模板来显示它。

- content：简单页面的内容 (即 HTML 页面)。框架不会对它作任何特别处理。由你负责使用模板来显示。
- enable_comments：是否允许该简单页面使用评论。框架不对此做任何特别处理。你可在模板中检查该值并根据需要显示评论窗体。
- template_name：用来解析该简单页面的模板名称。这是一个可选项；如果未指定模板或该模板不存在，系统会退而使用默认模板 flatpages/default.html 。
- registration_required：是否注册用户才能查看此简单页面。该设置项集成了 Djangos 验证/用户框架，该框架于第十二章详述。
- sites：该简单页面放置的站点。该项设置集成了 Django 多站点框架，该框架在本章的《多站点》一节中有所阐述。

你可以通过 Django 超级管理界面或者 Django 数据库 API 来创建简单页面。要了解更多内容，请查阅《添加、修改和删除简单页面》一节。

一旦简单页面创建完成， FlatpageFallbackMiddleware 将完成（剩下）所有的工作。每当 Django 引发 404 错误，作为终极手段，该中间件将根据所请求的 URL 检查平页面数据库。确切地说，它将使用所指定的 URL以及 SITE_ID 设置对应的站点 ID 查找一个简单页面。

如果找到一个匹配项，它将载入该简单页面的模板（如果没有指定的话，将使用默认模板 flatpages/default.html ）。同时，它把一个简单的上下文变量—— flatpage （一个简单页面对象）传递给模板。在模板解析过程中，它实际用的是 RequestContext 。

如果 FlatpageFallbackMiddleware 没有找到匹配项，该请求继续如常处理。

注意

该中间件仅在发生 404 （页面未找到）错误时被激活，而不会在 500 （服务器错误）或其他错误响应时被激活。还要注意的是必须考虑 MIDDLEWARE_CLASSES 的顺序问题。通常，你可以把 FlatpageFallbackMiddleware 放在列表最后，因为它是一种终极手段。

添加、修改和删除简单页面

可以用两种方式增加、变更或删除简单页面：

通过超级管理界面

如果已经激活了自动的 Django 超级管理界面，你将会在超级管理页面的首页看到有个 Flatpages 区域。你可以像编辑系统中其它对象那样编辑简单页面。

通过 Python API

前面已经提到，简单页面表现为 django/contrib/flatpages/models.py 中的标准 Django 模型。因此，你可以通过 Django 数据库 API 来存取简单页面对象，例如：

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.sites.models import Site
>>> fp = FlatPage(
...     url='/about/',
...     title='About',
...     content='<p>About this site...</p>',
...     enable_comments=False,
...     template_name='',
...     registration_required=False,
... )
>>> fp.save()
>>> fp.sites.add(Site.objects.get(id=1))
>>> FlatPage.objects.get(url='/about/')
<FlatPage: /about/ -- About>
```

使用简单页面模板

缺省情况下，系统使用模板 `flatpages/default.html` 来解析简单页面，但你也可以通过设定 `FlatPage` 对象的 `template_name` 字段来覆盖特定简单页面的模板。

你必须自己创建 `flatpages/default.html` 模板。只需要在模板目录创建一个 `flatpages` 目录，并把 `default.html` 文件置于其中。

简单页面模板只接受有一个上下文变量—— `flatpage` ，也就是该简单页面对象。

以下是一个 `flatpages/default.html` 模板范例:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

重定向

通过将重定向存储在数据库中并将其视为 Django 模型对象，Django 重定向框架让你能够轻松地管理它们。比如说，你可以通过重定向框架告诉Django，把任何指向 `/music/` 的请求重定向到 `/sections/arts/music/` 。当你需要在站点中移动一些东西时，这项功能就派上用场了——网站开发者应该穷尽一切办法避免出现坏链接。

使用重定向框架

安装重定向应用程序必须遵循以下步骤：

1. 将 `'django.contrib.redirects'` 添加到 `INSTALLED_APPS` 设置中。
2. 将 `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` 添加到 `MIDDLEWARE_CLASSES` 设置中。
3. 运行 `manage.py syncdb` 命令将所需的表安装到数据库中。

`manage.py syncdb` 在数据库中创建了一个 `django_redirect` 表。这是一个简单的查询表，只有 `site_id` 、 `old_path` 和 `new_path` 三个字段。

你可以通过 Django 超级管理界面或者 Django 数据库 API 来创建重定向。要了解更多信息，请参阅《增加、变更和删除重定向》一节。

一旦创建了重定向， `RedirectFallbackMiddleware` 类将完成所有的工作。每当 Django 应用引发一个 404 错误，作为终极手段，该中间件将为所请求的 URL 在重定向数据库中进行查找。确切地说，它将使用给定的 `old_path` 以及 `SITE_ID` 设置对应的站点 ID 查找重定向设置。（查阅前面的《多站点》一节可了解关于 `SITE_ID` 和多站点框架的更多细节） 然后，它将执行以下两个步骤：

- 如果找到了匹配项，并且 `new_path` 非空，它将重定向到 `new_path` 。
- 如果找到了匹配项，但 `new_path` 为空，它将发送一个 410 (Gone) HTTP 头信息以及一个空（无内容）响应。
- 如果未找到匹配项，该请求将如常处理。

该中间件仅为 404 错误激活，而不会为 500 错误或其他任何状态码的响应所激活。

注意必须考虑 `MIDDLEWARE_CLASSES` 的顺序。通常，你可以将 `RedirectFallbackMiddleware` 放置在列表的最后，因为它是一种终极手段。

注意

如果同时使用重定向和简单页面回退中间件，必须考虑先检查其中的哪一个（重定向或简单页面）。我们建议将简单页面放在重定向之前（因此将简单页面中间件放置在重定向中间件之前），但你可能有不同想法。

增加、变更和删除重定向

你可以两种方式增加、变更和删除重定向：

通过超级管理界面

如果已经激活了全自动的 Django 超级管理界面，你应该能够在超级管理首页看到重定向区域。可以像编辑系统中其它对象一样编辑重定向。

通过 Python API

`django/contrib/redirects/models.py` 中的一个标准 Django 模型代表了重定向。因此，你可以通过 Django 数据库 API 来存取重定向对象，例如：

```
>>> from django.contrib.redirects.models import Redirect
>>> from django.contrib.sites.models import Site
>>> red = Redirect(
...     site=Site.objects.get(id=1),
...     old_path='/music/',
...     new_path='/sections/arts/music/',
... )
>>> red.save()
>>> Redirect.objects.get(old_path='/music/')
<Redirect: /music/ ---> /sections/arts/music/>
```

CSRF 防护

`django.contrib.csrf` 开发包能够防止遭受跨站请求伪造攻击 (CSRF).

CSRF, 又叫进程跳转，是一种网站安全攻击技术。当某个恶意网站在用户未察觉的情况下将其从一个已经通过身份验证的站点诱骗至一个新的 URL 时，这种攻击就发生了，因此它可以利用用户已经通过身份验证的状态。开始的时候，要理解这种攻击技术比较困难，因此我们在本节将使用两个例子来说明。

一个简单的 CSRF 例子

假定你已经登录到 `example.com` 的网页邮件账号。该网页邮件站点上有一个登出按钮指向了 URL `example.com/logout` ，换句话说，要登出的话，需要做的唯一动作就是访问 URL： `example.com/logout` 。

通过在（恶意）网页上用隐藏一个指向 URL `example.com/logout` 的 `<iframe>` ,恶意网站可以强迫你访问该 URL 。因此，如果你登录 `example.com` 的网页邮件账号之后，访问了带有指向 `example.com/logout` 之 `<iframe>` 的恶意站点，访问该恶意页面的动作将使你登出 `example.com` 。

诚然，对你而言登出一个网页邮件站点并不会构成多大的安全破坏，但同样的攻击可能发生在 *任何* 信任用户的站点之上，比如在线银行网站或者电子商务网站。

稍微复杂一点的CSRF例子

在上一个例子中， `example.com` 应该负部分责任，因为它允许通过 HTTP `GET` 方法进行状态变更（即登入和登出）。如果对服务器的状态变更要求使用 HTTP `POST` 方法，情况就好得多了。但是，即便是强制要求使用 `POST` 方法进行状态变更操作也易受到 CSRF 攻击。

假设 `example.com` 对登出功能进行了升级，登出 `<form>` 按钮是通过一个指向 URL `example.com/logout` 的 `POST` 动作完成，同时在 `<form>` 中加入了以下隐藏的字段：

```
<input type="hidden" name="confirm" value="true" />
```

这就确保了用简单的 POST 到 example.com/logout 不会让用户登出；要让用户登出，用户必须通过 POST 向 example.com/logout 发送请求 *并且* 发送一个值为 'true' 的 POST 变量。

尽管增加了额外的安全机制，这种设计仍然会遭到 CSRF 的攻击——恶意页面仅需一点点改进而已。攻击者可以针对你的站点设计整个表单，并将其藏身于一个不可见的 <iframe> 中，然后使用 Javascript 自动提交该表单。

防止 CSRF

那么，是否可以让站点免受这种攻击呢？第一步，首先确保所有 GET 方法没有副作用。这样以来，如果某个恶意站点将你的页面包含为 <iframe>，它将不会产生负面效果。

该技术没有考虑 POST 请求。第二步就是给所有 POST 的 <form> 一个 隐藏字段，它的值是保密的并根据用户进程的 ID 生成。这样，从服务器端访问表单时，可以检查该保密的字段，不吻合时可以引发一个错误。

这正是 Django CSRF 防护层完成的工作，正如下面的小节所介绍的。

使用CSRF中间件

> django.contrib.csrf 开发包只有一个模块： middleware.py 。该模块包含了一个 Django 中间件类——CsrfMiddleware，该类实现了 CSRF 防护功能。

在设置文件中将 'django.contrib.csrf.middleware.CsrfMiddleware' 添加到 MIDDLEWARE_CLASSES 设置中可激活 CSRF 防护。该中间件必须在 SessionMiddleware *之后* 执行，因此在列表中 CsrfMiddleware 必须出现在 SessionMiddleware *之前*（因为响应中间件是自后向前执行的）。同时，它也必须在响应被压缩或解压之前对响应结果进行处理，因此 CsrfMiddleware 必须在 GZipMiddleware 之后执行。一旦将它添加到 MIDDLEWARE_CLASSES 设置中，你就完成了工作。参阅第 13 章中的《MIDDLEWARE_CLASSES 的顺序》一节了解更多诠释。

如果感兴趣的话，下面是 CsrfMiddleware 的工作模式。它完成以下两项工作：

1. 它修改当前处理的请求，向所有的 POST 表单增添一个隐藏的表单字段，使用名称是 csrfmiddlewaretoken，值为当前会话 ID 加上一个密钥的散列值。如果未设置会话 ID，该中间件将 *不会* 修改响应结果，因此对于未使用会话的请求来说性能损失是可以忽略的。

2. 对于所有含会话 cookie 集合的传入 POST 请求，它将检查是否存在 csrfmiddlewaretoken 及其是否正确。如果不是的话，用户将会收到一个 403 HTTP 错误。403 错误页面的内容是消息：检测到跨站伪装请求。请求被终止。”

该步骤确保只有源自你的站点的表单才能将数据 POST 回来。

该中间件特意只针对 HTTP POST 请求（以及对应的 POST 表单）。如我们所解释的，永远不应该因为使用了 GET 请求而产生负面效应，你必须自己来确保这一点。

未使用会话 cookie 的 POST 请求无法受到保护，但它们也不 *需要* 受到保护，因为恶意网站可用任意方法来制造这种请求。

为了避免转换非 HTML 请求，中间件在编辑响应结果之前对它的 Content-Type 头标进行检查。只有标记为 text/html 或 application/xml+xhtml 的页面才会被修改。

CSRF中间件的局限性

CsrfMiddleware 的运行需要 Django 的会话框架。（参阅第 12 章了解更多关于会话的内容）如果你使用了自定义会话或者身份验证框架手动管理会话 cookies，该中间件将帮不上你的忙。

如果你的应用程序以某种非常规的方法创建 HTML 页面（例如：在 Javascript 的 document.write 语句中发送 HTML 片段），你可能会绕开了向表单添加隐藏字段的过滤器。在此情况下，表单提交永远无法成功。（这是因为在页面被发送到客户端之前， CsrfMiddleware 使用正则表达式向 HTML 中添加 csrfmiddlewaretoken 字段，而有时正则表达式无法处理非常规的 HTML。）如果你怀疑发生这类事情，只需在浏览器中查看源码的表单中是否已经插入了 csrfmiddlewaretoken 。

想了解更多关于 CSRF 的信息和例子的话，可以访问 http://en.wikipedia.org/wiki/CSRF 。

人性化数据

该应用程序包括一系列 Django 模板过滤器，用于增加数据的人性化。要激活这些过滤器，仅需将 `'django.contrib.humanize'` 加入到 `INSTALLED_APPS` 设置中。一旦完成该项工作，在模板中使用 `{% load humanize %}` 就能访问后续小节中讲述的过滤器了。

apnumber

对于 1 到 9 的数字，该过滤器返回了数字的拼写形式。否则，它将返回数字。这遵循的是美联社风格。

举例：

- 1 变成 one 。
- 2 变成 two 。
- 10 变成 10 。

你可以传入一个整数或者表示整数的字符串。

intcomma

该过滤器将整数转换为每三个数字用一个逗号分隔的字符串。

例如：

- 4500 变成 4,500 。
- 45000 变成 45,000 。
- 450000 变成 450,000 。
- 4500000 变成 4,500,000 。

你可以传入整数或者表示整数的字符串。

intword

该过滤器将一个很大的整数转换成友好的文本表示方式。它对于超过一百万的数字最好用。

例如：

- 1000000 变成 1.0 million 。
- 1200000 变成 1.2 million 。
- 1200000000 变成 1.2 billion 。

最大支持不超过一千的五次方（1,000,000,000,000,000）。

你可以传入整数或者表示整数的字符串。

ordinal

该过滤器将整数转换为序数词的字符串形式。

例如：

- 1 变成 1st 。

- 2 变成 2nd 。
- 3 变成 3rd 。

你可以传入整数或着表示整数的字符串。

标记过滤器

下列模板过滤器集合实现了常见的标记语言：

- `textile` : 实现了 Textile (http://en.wikipedia.org/wiki/Textile_%28markup_language%29)
- `markdown` : 实现了 Markdown (http://en.wikipedia.org/wiki/Markdown)
- `restructuredtext` : 实现了 ReStructured Text (http://en.wikipedia.org/wiki/ReStructuredText)

每种情形下，过滤器都期望字符串形式的格式化标记，并返回表示标记文本的字符串。例如： `textile` 过滤器把以 Textile 格式标记的文本转换为 HTML 。

```
{% load markup %}
{{ object.content|textile }}
```

要激活这些过滤器，仅需将 `'django.contrib.markup'` 添加到 `INSTALLED_APPS` 设置中。一旦完成了该项工作，在模板中使用 `{% load markup %}` 就能使用这些过滤器。要想掌握更多信息的话，可阅读 `django/contrib/markup/templatetags/markup.py`. 内的源代码。

下一步？

这些继承框架（CSRF、身份验证系统等等）通过提供 *中间件* 来实现其奇妙的功能。本质上，中间件就是在每个请求之前或/和之后运行的代码，它们可随意修改每个请求或响应。接下来，我们将讨论 Django 的内建中间件，并解释如何编写自己的中间件。

第十五章 中间件

在有些场合，需要对Django处理的每个request都执行某段代码。这类代码可能是在view处理之前修改传入的request，或者记录日志信息以便于调试，等等。

这类功能可以用Django的中间件框架来实现，该框架由切入到Django的request/response处理过程中的钩子集合组成。这个轻量级低层次的plug-in系统，能用于全面的修改Django的输入和输出。

每个中间件组件都用于某个特定的功能。如果顺序阅读这本书(谨对后现代主义者表示抱歉)，你可能已经多次看到中间件了:

- 第12章中所有的session和user工具都籍由一小簇中间件实现(例如，由中间件设定view中可见的 `request.session` 和 `request.user`)。
- 第13章讨论的站点范围cache实际上也是由一个中间件实现，一旦该中间件发现与view相应的response已在缓存中，就不再调用对应的view函数。
- 第14章所介绍的 `flatpages` , `redirects` , 和 `csrf` 等应用也都是通过中间件组件来完成其魔法般的功能。

这一章将深入到中间件及其工作机制中，并阐述如何自行编写中间件。

什么是中间件

中间件组件是遵循特定API规则的简单Python类。在深入到该API规则的正式细节之前，先看一下下面这个非常简单的例子。

高流量的站点通常需要将Django部署在负载平衡proxy(参见第20章)之后。这种方式将带来一些复杂性，其一就是每个request中的远程IP地址(`request.META["REMOTE_IP"]`)将指向该负载平衡proxy，而不是发起这个request的实际IP。负载平衡proxy处理这个问题的方法在特殊的 `X-Forwarded-For` 中设置实际发起请求的IP。

因此，需要一个小小的中间件来确保运行在proxy之后的站点也能够在 `request.META["REMOTE_ADDR"]` 中得到正确的IP地址：

```
class SetRemoteAddrFromForwardedFor(object):
    def process_request(self, request):
        try:
            real_ip = request.META['HTTP_X_FORWARDED_FOR']
        except KeyError:
            pass
        else:
            # HTTP_X_FORWARDED_FOR can be a comma-separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

一旦安装了该中间件(参见下一节)，每个request中的 `X-Forwarded-For` 值都会被自动插入到 `request.META['REMOTE_ADDR']` 中。这样，Django应用就不需要关心自己是否位于负载平衡proxy之后；简单读取 `request.META['REMOTE_ADDR']` 的方式在是否有proxy的情形下都将正常工作。

实际上，为针对这个非常常见的情形，Django已将该中间件内置。它位于 `django.middleware.http` 中, 下一节将给出这个中间件相关的更多细节。

安装中间件

如果按顺序阅读本书，应当已经看到涉及到中间件安装的多个示例,因为前面章节的许多例子都需要某些特定的中间件。出于完整性考虑，下面介绍如何安装中间件。

要启用一个中间件，只需将其添加到配置模块的 `MIDDLEWARE_CLASSES` 元组中。在 `MIDDLEWARE_CLASSES` 中，中间件组

件用字符串表示：指向中间件类名的完整Python路径。例如，下面是 `django-admin.py startproject` 创建的缺省 `MIDDLEWARE_CLASSES`：

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.doc.XViewMiddleware'
)
```

Django项目的安装并不强制要求任何中间件，如果你愿意， `MIDDLEWARE_CLASSES` 可以为空。但我们建议启用 `CommonMiddleware` ，稍后做出解释。

这里中间件出现的顺序非常重要。在request和view的处理阶段，Django按照 `MIDDLEWARE_CLASSES` 中出现的顺序来应用中间件，而在response和异常处理阶段，Django则按逆序来调用它们。也就是说，Django将 `MIDDLEWARE_CLASSES` 视为view函数外层的顺序包装子：在request阶段按顺序从上到下穿过，而在response则反过来。关于Django处理阶段的详细信息，请参见第三章”Django怎么处理一个请求: 完整细节”这一节。

中间件方法

现在，我们已经知道什么是中间件和怎么安装它，下面将介绍中间件类中可以定义的所有方法。

Initializer: __init__(self)

在中间件类中， `__init__()` 方法用于执行系统范围的设置。

出于性能的考虑，每个已启用的中间件在每个服务器进程中只初始化 一次。也就是说 `__init__()` 仅在服务进程启动的时候调用，而在针对单个request处理时并不执行。

对一个middleware而言，定义 `__init__()` 方法的通常原因是检查自身的必要性。如果 `__init__()` 抛出异常 `django.core.exceptions.MiddlewareNotUsed` ,则Django将从middleware栈中移出该middleware。可以用这个机制来检查middleware依赖的软件是否存在、服务是否运行于调试模式、以及任何其它环境因素。

在中间件中定义 `__init__()` 方法时，除了标准的 `self` 参数之外，不应定义任何其它参数。

Request预处理函数: process_request(self, request)

这个方法的调用时机在Django接收到request之后，但仍未解析URL以确定应当运行的view之前。Django向它传入相应的 `HttpRequest` 对象，以便在方法中修改。

`process_request()` 应当返回 `None` 或 `HttpResponse` 对象.

- 如果返回 `None` , Django将继续处理这个request,执行后续的中间件， 然后调用相应的view.
- 如果返回 `HttpResponse` 对象, Django 将不再执行 *任何* 其它的中间件(而无视其种类)以及相应的view。 Django将立即返回该 `HttpResponse` .

View预处理函数: process_view(self, request, view, args, kwargs)

这个方法的调用时机在Django执行完request预处理函数并确定待执行的view之后，但在view函数实际执行之前。

表15-1列出了传入到这个View预处理函数的参数。

SoSjGG yiqzndzralaf, [url=http://gjemutbubtxu.com/]gjemutbubtxu[/url], [link=http://zgdgecnxtllq.com/]zgdgecnxtllq[/link], http://gtldsopsugix.com/

如同 `process_request()` , `process_view()` 应当返回 `None` 或 `HttpResponse` 对象。

- 如果返回 `None` , Django将继续处理这个 `request` ,执行后续的中间件， 然后调用相应的view.

- 如果返回 `HttpResponse` 对象, Django 将不再执行 *任何* 其它的中间件(不论种类)以及相应的view. Django将立即返回该 `HttpResponse` .

Response后处理函数: process_response(self, request, response)

这个方法的调用时机在Django执行view函数并生成response之后。这里，该处理器就能修改response的内容；一个常见的用途是内容压缩，如gzip所请求的HTML页面。

这个方法的参数相当直观: `request` 是request对象，而 `response` 则是从view中返回的response对象。

不同可能返回 `None` 的request和view预处理函数, `process_response()` *必须* 返回 `HttpResponse` 对象. 这个response对象可以是传入函数的那一个原始对象(通常已被修改)，也可以是全新生成的。

Exception后处理函数: process_exception(self, request, exception)

这个方法只有在request处理过程中出了问题并且view函数抛出了一个未捕获的异常时才会被调用。这个钩子可以用来发送错误通知，将现场相关信息输出到日志文件, 或者甚至尝试从错误中自动恢复。

这个函数的参数除了一贯的 `request` 对象之外，还包括view函数抛出的实际的异常对象 `exception` 。

`process_exception()` 应当返回 None 或 HttpResponse 对象.

- 如果返回 `None` , Django将用框架内置的异常处理机制继续处理相应request。
- 如果返回 `HttpResponse` 对象, Django 将使用该response对象，而短路框架内置的异常处理机制。

备注

Django自带了相当数量的中间件类(将在随后章节介绍)，它们都是相当好的范例。阅读这些代码将使你对中间件的强大有一个很好的认识。

在Djangos wiki上也可以找到大量的社区贡献的中间件范例: http://code.djangoproject.com/wiki/ContributedMiddleware

内置的中间件

Django自带若干内置中间件以处理常见问题，将从下一节开始讨论。

认证支持中间件

中间件类: `django.contrib.auth.middleware.AuthenticationMiddleware` .

wSOx5R hztnqpaboavm, [url=http://vmvmmjrsqxic.com/]vmvmmjrsqxic[/url], [link=http://sngrfxoajwdo.com/]sngrfxoajwdo[/link], http://eautnmbfurjc.com/

完整的细节请参见第12章。

SP3df8 bfqhoouwrdmm, [url=http://hzujzeyighho.com/]hzujzeyighho[/url], [link=http://mbrgpoykhljx.com/]mbrgpoykhljx[/link], http://qqgcbhgtklwg.com/

mBh1AN vmlyatgebrhc, [url=http://dvatqsqhoxsu.com/]dvatqsqhoxsu[/url], [link=http://zuchupmuwhub.com/]zuchupmuwhub[/link], http://mxgkuqikazko.com/

这个中间件为完美主义者提供了一些便利:

> *禁止 ``DISALLOWED_USER_AGENTS`` 列表中所设置的user agent访问* ：一旦提供，这一列表应当由已编译的正则表达式对象组成，这些对象用于匹配传入的request请求头中的user-agent域。下面这个例子来自某个配置文件片段：

```
import re
```

```
DISALLOWED_USER_AGENTS = (
    re.compile(r'^OmniExplorer_Bot'),
    re.compile(r'^Googlebot')
)
```

请注意 `import re` ,因为 `DISALLOWED_USER_AGENTS` 要求其值为已编译的正则表达式(也就是 `re.compile()` 的返回值)。配置文件是常规的python文件，所以在其中包括Python `import` 语句不会有任何问题。

依据 ``APPEND_SLASH`` 和 ``PREPEND_WWW`` 的设置执行URL重写：如果 `APPEND_SLASH` 为 `True` , 那些尾部没有斜杠的URL将被重定向到添加了斜杠的相应URL，除非path的最末组成部分包含点号。因此，`foo.com/bar` 会被重定向到 `foo.com/bar/` , 但是 `foo.com/bar/file.txt` 将以不变形式通过。

如果 `PREPEND_WWW` 为 True , 那些缺少先导www.的URLs将会被重定向到含有先导www.的相应URL上。

这两个选项都是为了规范化URL。其后的哲学是每个URL都应且只应当存在于一处。技术上来说，URL `example.com/bar` 与 `example.com/bar/` 及 `www.example.com/bar/` 都互不相同。搜索引擎编目程序将把它们视为不同的URL,这将不利于该站点的搜索引擎排名，因此这里的最佳实践是将URL规范化。

依据 ``USE_ETAGS`` 的设置处理Etag : ETags 是HTTP级别上按条件缓存页面的优化机制。如果 `USE_ETAGS` 为 `True` ，Django针对每个请求以MD5算法处理页面内容，从而得到Etag, 在此基础上，Django将在适当情形下处理并返回 `Not Modified` 回应(译注：或者设置response头中的Etag域)。

请注意，还有一个条件化的 `GET` 中间件, 处理Etags并干得更多，下面马上就会提及。

压缩中间件

中间件类: `django.middleware.gzip.GZipMiddleware` .

这个中间件自动为能处理gzip压缩(包括所有的现代浏览器)的浏览器自动压缩返回]内容。这将极大地减少Web服务器所耗用的带宽。代价是压缩页面需要一些额外的处理时间。

相对于带宽，人们一般更青睐于速度，但是如果你的情形正好相反，尽可启用这个中间件。

条件化的GET中间件

中间件类: `django.middleware.http.ConditionalGetMiddleware` .

这个中间件对条件化 `GET` 操作提供支持。如果response头中包括 `Last-Modified` 或 `ETag` 域，并且request头中包含 `If-None-Match` 或 `If-Modified-Since` 域，且两者一致，则该response将被response 304(Not modified)取代。对 `ETag` 的支持依赖于 `USE_ETAGS` 配置及事先在response头中设置 `ETag` 域。稍前所讨论的通用中间件可用于设置response中的 `ETag` 域。

此外，它也将删除处理 `HEAD` request时所生成的response中的任何内容，并在所有request的response头中设置 `Date` 和 `Content-Length` 域。

反向代理支持 (X-Forwarded-For中间件)

中间件类: `django.middleware.http.SetRemoteAddrFromForwardedFor` .

这是我们在 什么是中间件 这一节中所举的例子。 在 `request.META['HTTP_X_FORWARDED_FOR']` 存在的前提下，它根据其值来设置 `request.META['REMOTE_ADDR']` 。在站点位于某个反向代理之后的、每个request的 `REMOTE_ADDR` 都被指向 `127.0.0.1` 的情形下，这一功能将非常有用。

红色警告!

这个middleware并 *不* 验证 `HTTP_X_FORWARDED_FOR` 的合法性。

如果站点并不位于自动设置 `HTTP_X_FORWARDED_FOR` 的反向代理之后，请不要使用这个中间件。否则，因为任何人都能够伪造 `HTTP_X_FORWARDED_FOR` 值，而 `REMOTE_ADDR` 又是依据 `HTTP_X_FORWARDED_FOR` 来设置，这就意味着任何人都能够伪造IP地址。

只有当能够绝对信任 HTTP_X_FORWARDED_FOR 值得时候才能够使用这个中间件。

会话支持中间件

中间件类: django.contrib.sessions.middleware.SessionMiddleware .

这个中间件激活会话支持功能. 细节请参见第12章。

站点缓存中间件

ZV3FvR haiujfhxgmcm, [url=http://nvlhddespdgs.com/]nvlhddespdgs[/url], [link=http://siljtaebxpgm.com/]siljtaebxpgm[/link], http://vlrjbtvuqedl.com/

这个中间件缓存Django处理的每个页面。已在第13章中详细讨论。

事务处理中间件

中间件类: django.middleware.transaction.TransactionMiddleware .

这个中间件将数据库的 COMMIT 或 ROLLBACK 绑定到request/response处理阶段。如果view函数成功执行，则发出 COMMIT 指令。如果view函数抛出异常，则发出 ROLLBACK 指令。

这个中间件在栈中的顺序非常重要。其外层的中间件模块运行在Django缺省的 保存-提交 行为模式下。而其内层中间件(在栈中的其后位置出现)将置于与view函数一致的事务机制的控制下。

关于数据库事务处理的更多信息，请参见附录C。

X-View 中间件

中间件类: django.middleware.doc.XViewMiddleware .

这个中间件将对来自 INTERNAL_IPS 所设置的内部IP的HEAD请求发送定制的 X-View HTTP头。Django的自动文档系统使用了这个中间件。

下一章

Web开发者和数据库模式设计人员并不总是享有白手起家打造项目的奢侈机会。下一章将阐述如何集成遗留系统，比如继承自1980年代的数据库模式。

第十六章 集成已有的数据库和应用

Django最适合于所谓的green-field开发，即从头开始的一个项目，正如你在一块还长着青草的未开垦的土地上从零开始建造一栋建筑一般。然而，尽管Django偏爱从头开始的项目，将这个框架和以前遗留的数据库和应用相整合仍然是可能的。本章就将介绍一些整合的技巧。

与遗留数据库整合

Django的数据库层从Python代码生成SQL schemas—但是对于遗留数据库，你已经拥有SQL schemas，这种情况下你需要为你 已经存在的数据库表写模型(由于性能的原因，Django的数据库层不支持通过运行时自省数据库的不工作的对象-关系映射， 为了使用数据库API，你需要写模型代码)，幸运的是，Django带有通过阅读你的数据库表规划来生成模型代码的辅助工具 该辅助工具称为manage.py inspectdb

使用 `inspectdb`

The `inspectdb` 工具内省检查你的配置文件（setting file）指向的数据库，针对你的每一个表生成一个Django model的表现，然后将这些Python model的代码显示在系统的标准输出里面。

下面是一个从头开始的针对一个典型的遗留数据库的整合过程

通过运行django-admin.py startproject mysite (这里 `mysite` 是你的项目的名字)建立一个Django项目。好的，那我们在这个例子中就用这个 `mysite` 作为项目的名字。

> 编辑项目中的配置文件, `mysite/settings.py` ,告诉Django你的数据库连接参数和数据库名。具体的说，要提供 `DATABASE_NAME` , `DATABASE_ENGINE` , `DATABASE_USER` , `DATABASE_PASSWORD` , `DATABASE_HOST` , 和 `DATABASE_PORT` 这些配置信息. (注意,这里面有些配置项是可选的，更多信息参考第五章）
>
> 通过运行 `python mysite/manage.py startapp myapp` (这里 `myapp` 是你的应用的名字)创建一个Django应用. 那么,我们就以 `myapp` 做为这个应用的名字.

Mgi4P7 egawnqpydxgr, [url=http://pqxrfbeebwlx.com/]pqxrfbeebwlx[/url], [link=http://ayerpdvfcibx.com/]ayerpdvfcibx[/link], http://oiclqzyfemdn.com/

> 将标准shell的输出重定向，保存输出到你的应用的 `models.py` 文件里：

```
python mysite/manage.py inspectdb > mysite/myapp/models.py
```

> 编辑 `mysite/myapp/models.py` 文件以清理生成的 models 以及一些必要的定制化。 下一个章节对此有些好的建议。

BJuyYO gudqbthrquuw, [url=http://nnferpaaqtza.com/]nnferpaaqtza[/url], [link=http://pkxlmlxjxtow.com/]pkxlmlxjxtow[/link], http://hvuszhvyijlw.com/

如你可能会预料到的，数据库自省不是完美的，你需要对产生的模型代码做些许清理。 这里提醒一点关于处理生成 models 的要点:

dMtfgT ofpjyxnfjlnj, [url=http://dpixdvamofrx.com/]dpixdvamofrx[/url], [link=http://ivaqvrambdds.com/]ivaqvrambdds[/link], http://owvunzpidrey.com/

所生成的每一个model中的每个字段都拥有自己的属性，包括id主键字段。但是，请注意，如果某个model没有主键的话，那么Django会自动为其增加一个Id主键字段。这样一来，你也许希望使用如下代码来对任意行执行删除操作：

```
id = models.IntegerField(primary_key=True)
```

这样做并不是仅仅因为这些行是冗余的，而且如果当你的应用需要向这些表中增加新记录时，这些行会导致某些问题。而inspectdb命令并不能检测出一个字段是否自增长的，因此必要的时候，你必须将他们修改为AutoField.

每一个字段类型，如CharField、DateField， 是通过查找数据库列类型如VARCHAR,DATE来确定的。如果inspectdb无法对某个model字段类型根据数据库列类型进行映射，那么它会使用TextField字段进行代替，并且会在所生成model字段后面加入Python注释“该字段类型是猜的”。因此，请特别注意这一点，并且在必要的时候相应的修改这些字段类型。

如果你的数据库中的某个字段在Django中找不到合适的对应物，你可以放心的略过它，因为Django层并没有要求必须包含你的表中的每一个字段。

如果数据库中某个列的名字是Python的保留字，比如pass、class或者for等，inspectdb会在每个属性名后附加上_field，并将db_column属性设置为真实的字段名，比如pass,class或者for等。

例如，某张表中包含一个INT类型的列，其列名为for，那么所生成的model将会包含如下所示的一个字段：

```
for_field = models.IntegerField(db_column='for')
```

jBI57S jillralfgjok, [url=http://eofkzwzlxabw.com/]eofkzwzlxabw[/url], [link=http://zbzzidjvvvxl.com/]zbzzidjvvvxl[/link], http://dsxsfstfqzfy.com/

如果数据库中某张表引用了其他表（正如大多数数据库系统所做的那样），你需要适当的修改所生成model的顺序，以使得这种引用能够正确映射。例如，model Book拥有一个针对于model Author的外键，那么后者应该先于前者被定义。如果你需要为一个还没有被定义的model创建一个关系，那么你可以使用该model的名字，而不是model对象本身。

对于PostgreSQL,MySQL和SQLite数据库系统，inspectdb能够自动检测出主键关系。也就是说，它会在合适的位置插入primary_key=True。而对于其他数据库系统，你必须为每一个model中至少一个字段插入这样的语句，因为Django的model要求必须拥有一个primary_key=True的字段。

外键检测仅对PostgreSQL，还有MySQL表中的某些特定类型生效。至于其他数据库，外键字段都将在假定其为INT列的情况下被自动生成为IntegerField。

与认证系统的整合

将Django与其他现有认证系统的用户名和密码或者认证方法进行整合是可以办到的。

例如，你所在的公司也许已经安装了LDAP，并且为每一个员工都存储了相应的用户名和密码。如果用户在LDAP和基于Django的应用上拥有独立的账号，那么这时无论对于网络管理员还是用户自己来说，都是一件很令人头痛的事儿。

为了解决这样的问题，Django认证系统能让您以插件方式与其他认证资源进行交互。您可以覆盖Diangos的默认基于数据库模式，您还可以使用默认的系统与其他系统进行交互。

指定认证后台

在后台，Django维护了一个用于检查认证的后台列表。当某个人调用 `django.contrib.auth.authenticate()` (如12章中所述)时，Django会尝试对其认证后台进行遍历认证。如果第一个认证方法失败，Django会尝试认证第二个，以此类推，一直到尝试完。

认证后台列表在AUTHENTICATION_BACKENDS设置中进行指定，它应该是指向知道如何认证的Python类的Python路径的名字数组，这些类可以放置在您的Python路径的任何位置上。

默认情况下，AUTHENTICATION_BACKENDS被设置为如下：

```
('django.contrib.auth.backends.ModelBackend',)
```

那就是检测Django用户数据库的基本认证模式。

对于多个顺序组合的AUTHENTICATION_BACKENDS，如果其用户名和密码在多个后台中都是有效的，那么Django将会在第一个正确通过认证后停止进一步的处理。

如何写一个认证后台

一个认证后台其实就是一个实现了如下两个方法的类： `get_user(id)` 和 `authenticate(**credentials)` 。

方法 `get_user` 需要一个参数 `id` ，这个 `id` 可以是用户名，数据库ID或者其他任何数值，该方法会返回一个 `User` 对

象。

方法 `authenticate` 使用证书作为关键参数。大多数情况下，该方法看起来如下：

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
```

但是有时候它也可以认证某个令牌，例如：

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
```

每一个方法中， `authenticate` 都应该检测它所获取的证书，并且当证书有效时，返回一个匹配于该证书的 `User` 对象，如果证书无效那么返回 `None` 。

如12章中所述，Django管理系统紧密连接于其自己后台数据库的 `User` 对象。实现这个功能的最好办法就是为您的后台数据库（如LDAP目录，外部SQL数据库等）中的每个用户都创建一个对应的Django User对象。您可以提前写一个脚本来完成这个工作，也可以在某个用户第一次登陆的时候在 `authenticate` 方法中进行实现。

以下是一个示例后台程序，该后台用于认证定义在 `setting.py` 文件中的username和password变量，并且在该用户第一次认证的时候创建一个相应的Django `User` 对象。

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """
    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. Note that we can set password
                # to anything, because it won't be checked; the password
                # from settings.py will.
                user = User(username=username, password='get from settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

和遗留Web应用集成

同由其他技术驱动的应用一样，在相同的Web服务器上运行Django应用也是可行的。最简单直接的办法就是利用Apaches配置文件httpd.conf，将不同的URL类型代理至不同的技术。（请注意，第20章包含了在Apache/mod_python上配置Django的相关内容，因此在尝试本章集成之前花些时间去仔细阅读第20章或许是值得的。）

关键在于只有在您的httpd.conf文件中进行了相关定义，Django对某个特定的URL类型的驱动才会被激活。在第20章中解

释的缺省部署方案假定您需要Django去驱动某个特定域上的每一个页面。

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

这里, `<Location "/">` 这一行表示用Django处理每个以根开头的URL.

精妙之处在于Django将<location>指令值限定于一个特定的目录树上。举个例子，比如说您有一个在某个域中驱动大多数页面的遗留PHP应用，并且您希望不中断PHP代码的运行而在../admin/位置安装一个Django域。要做到这一点，您只需将<location>值设置为/admin/即可。

```
<Location "/admin/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

KshJiK hychpznxvvpz, [url=http://idhrtvjehost.com/]idhrtvjehost[/url], [link=http://yssfucouapzc.com/]yssfucouapzc[/link], http://hsoazioekrvi.com/

请注意，把Diango绑定到的合格的URL（比如在本章例子中的 `/admin/` ）并不会影响其对URL的解析。绝对路径对Django才是有效的（例如 `/admin/people/person/add/` ），而非截断后的URL（例如 `/people/person/add/` ）。这意味着你的根URLconf必须包含前缀 `/admin/` 。

aoRVhS rxhmltxgkdkj, [url=http://wmhujqrawkcu.com/]wmhujqrawkcu[/url], [link=http://blebolfmttir.com/]blebolfmttir[/link], http://tvqjecsbdnap.com/

谈到Django管理站点和让整个框架服从于遗留Web集成需求时，另一个常见任务就是定制Django管理站点。我们会在下一章中聚焦类似的定制。

第17章 扩展 Django 管理界面

第六章介绍了 Django 的管理界面，现在是该回过头来仔细了解一下的时候了。

第六章的最后几节介绍了定制部分管理界面的一些简单方法。进入本章之前，请先复习一下那部分资料；其中涵盖了如何定制管理接口的 change list 和 edit forms ，以及如何将管理界面冠以与站点一致的风格。

第六章还讨论了何时以及如何使用管理界面，由于那些资料对本章剩下内容是个好的起点，在此我们将重温一遍：

显而易见，对数据编辑工作来说，该管理界面极为有用（想象一下）。如果用于完成某种数据的录入工作，该管理界面实在是无人能及。我们猜想本书绝大多数读者都有成堆数据录入任务。

Django管理接口特别关注那些没有技术背景的用户来使用数据录入；这也是该功能的开发目的。在Django最初开发地报社，开发一个典型的在线市政供水质量报告系统，需求如下：

- 负责该题材的记者与某个开发人员会面，提交现有数据。
- 开发人员围绕该数据设计一个模型，并为该记者开发出管理界面。
- 在记者将数据录入 Django 的同时，程序员就可以专注于开发公众访问界面了（最有趣的部分！）。

换句话说，Django 管理接口之所以存在的首要目的是为了方便内容编辑人员和程序员同时开展工作。

当然，除了显而易见的数据录入任务之外，我们发现管理界面在其他一些情况下有是很有用处的。

- *管理获得的数据*：很少有真实数据输入会和像 `http://chicagocrime.org` 这样的站点相关联，因为多数数据来自自动生成的源头。然而，当所获取的数据出错而导致麻烦时，能够便捷地找到并修改出错数据将会有助于问题解决。

无需或者仅需略为定制之后， Django 管理界面就能处理绝大部分常见情形。然而，正是因为在设计上极力折衷，Django 管理界面能够很好地处理这种常见情形也就意味着它无法同样处理其它一些编辑模型。

管理之道

受信任用户编辑结构化的内容。

是的，这非常的简单，但这种简单是建立在一整堆假定之上的。Django 管理界面的全部设计理念均直接遵循这些假定，因此让我们深入理解一下这些后续小节中所出现术语的含义。

受信任用户

管理界面被设计成由你这样的开发人员所 *信任* 的人使用。这里所指的并非只是通过身份验证的人；而是说 Django 假定可以相信内容编辑者只会做对的事情。

反过来说，这也就意味着如果你信任用户，他们无需征得许可就能编辑内容，也没有人需要对他们的编辑行为进行许

可。另一层含义是，尽管认证系统功能强大，但到本书写作时为止，它并不支持对象级基础的访问限制。如果你允许某人对自己的新闻报道进行编辑，你必须能够确信该用户不会未经许可对其他人的报道进行编辑。

编辑

Django 管理界面的首要目的是让用户编辑数据。乍一看这是显而易见的，但仔细一想却又变得有点难以捉摸和不同凡响。

举例来说，虽然管理界面非常便于查验数据（如刚才所讨论的那样），但这并不是它的设计初衷。比如我们在第 12 章中谈到的，它缺少视图许可。Django 假定如果某人在管理界面中可以查看内容，那么也可以进行编辑。

还有件更重要的事情要注意，那就是对于远程调用工作流的缺乏。如果某个特定任务由一系列步骤组成，没有任何机制确保这些步骤能够以某个特定顺序完成。 Django 管理界面专注于 *编辑* ，而不关心修改周边的活动。对工作流的这种回避也源自于信任原则：管理界面的设计理念是工作流乃人为事物，无需在代码中实现。

最后，要注意的是管理界面中缺少聚合。也就是说，不支持显示总计、平均值之类的东西。再次重申，管理界面只用于编辑——它预期你将通过定义视图来完成其它所有工作。

结构化的内容

30CEN9 bxyzitjhvwis, [url=http://ylhgcrwsnuue.com/]ylhgcrwsnuue[/url], [link=http://mecoeqqxpail.com/]mecoeqqxpail[/link], http://iuvxeciajpoz.com/

就此打住

现在可以肯定的是，Django 的管理界面 *并不* 打算成为所有人的万能工具；相反我们选择了专心做一件事情，并把它完成得尽善尽美。

SaPQz9 fmvbpdngtcts, [url=http://tvwreqwomvco.com/]tvwreqwomvco[/url], [link=http://nwfgxlxpodeq.com/]nwfgxlxpodeq[/link], http://khcvnasfavcq.com/

必须记住，尽管管理界面很复杂，但它始终只是一个应用程序。只要有充足的时间，任何Django的开发者都能做到admin接口做到的所有事。 因此，我们需要寄希望于将来会有一个完全不同的admin接口会出现，这个新的接口拥有一系列不同的前提假设，并且工作方式也完全不同。

最后要指出的是，在本文写作之时，Django 开发者们正在进行一个新的管理界面的开发工作，该版本将提供更多定制灵活性。当你阅读本文时，这些新特性也许已经进入了真实的 Django 发布之中。你可以向 Django 社区的某些人了解是否已经整合了 newforms-admin 主干代码。

定制管理模板

lrYxwt uokhfgxirpcl, [url=http://wqhpuiouinaf.com/]wqhpuiouinaf[/url], [link=http://fpatpucympci.com/]fpatpucympci[/link], http://qvtnfmmjkvrf.com/

现在，我们来看看如何来快速定制admin管理接口的外观。 第6章讲到了一些最常见的任务：修改商标（为那些讨厌蓝色的尖发老板），或者提供一个自定义的form。

更进一步的目标常常会包含，改变模板中的一些特殊的项。每一种admin的视图，包括修改列表、编辑表单、删除确认页以及历史视图，都有一个与之相关联的模板可以以多种方式来进行覆盖。

首先，你可以在全局上覆盖模板。admin视图使用标准的模板载入机制来查找模板。所以如果你在模板目录中创建了一个新的模板，Django会自动地加载它。全局的模板在表17-1中列出。

Fd2cvv fphafdvvmhea, [url=http://mompqgnztmob.com/]mompqgnztmob[/url], [link=http://oamzxjphpsxv.com/]oamzxjphpsxv[/link], http://ktqzcuqdddlb.com/

大多数时候，你可能只是想修改一个单独的对象或应用程序，而不是修改全局性的设定。因此，每个admin视图总是先去查找与模型或应用相关的模板。这些视图寻找模板的顺序如下：

- `admin/<app_label>/<object_name>/<template>.html`

JRmAl2 kqeldaxlpkep, [url=http://ihzkjznjrsqj.com/]ihzkjznjrsqj[/url], [link=http://sqhirlipxjfo.com/]sqhirlipxjfo[/link], http://vksbmtrxzjbj.com/

- `admin/<template>.html`

例如，在 `books` 这个应用程序中， `Book` 模块的添加/编辑表单的视图会按如下顺序查找模板：

- `admin/books/book/change_form.html`
- `admin/books/change_form.html`
- `admin/change_form.html`

自定义模型模板

大多数时候，你想使用第一个模板来创建特定模型的模板。 通常，最好的办法是扩展基模板和往基模板中定义的区块中添加信息。

例如，我们想在那个书籍页面的顶部添加一些帮助文本。 可能是像图17-1所示的表单一样的东西。

图 17-1. 一个自定义管理编辑表单.

这做起来非常容易：只要建立一个 `admin/bookstore/book/change_form.html` 模板，并输入下面的代码：

```
{% extends "admin/change_form.html" %}

{% block form_top %}
  <p>Insert meaningful help message here...</p>
{% endblock %}
```

mGrlX4 omnhyzkldddt, [url=http://ydlrelntzksj.com/]ydlrelntzksj[/url], [link=http://zzizrvybyaol.com/]zzizrvybyaol[/link], http://iudthlpesvpv.com/

自定义JavaScript

这些自定义模型模板的常见用途包括，给admin页面增加自定义的javascript代码来实现一些特殊的视图物件或者是客户端行为。

幸运的是，这可以更简单。每一个admin模板都定义了 `{% block extrahead %}` ，你可以在 `<head>` 元素中加入新的内容。例如你想要增加jQuery(http://jquery.com/) 到你的admin历史中，可以这样做：

```
{% extends "admin/object_history.html" %}

{% block extrahead %}
    <script src="http://media.example.com/javascript/jquery.js" type="text/javascript"></script>
    <script type="text/javascript">

        // code to actually use jQuery here...

    </script>
{% endblock %}
```

备注

我们并不知道你为什么需要把jQuery放入到历史页中，但是这个例子可以被用到任何的模板中。

TYy7is hpswriedwzmw, [url=http://bvkzcktnoefm.com/]bvkzcktnoefm[/url], [link=http://wkbudpcvacdz.com/]wkbudpcvacdz[/link], http://arujugnwqwuh.com/

创建自定义管理视图

现在，想要往Django的admin管理接口添加自定义行为的人，可能开始觉得有点奇怪了。我们这里所讲的都是如何改变admin管理接口的外观。他们都在喊：如何才能改变admin管理接口的内部工作机制。

首先要提的一点是，这并不神奇。admin管理接口并没有做任何特殊的事情，它只不过是和其他一些视图一样，简单地处理数据而已。

确实，这里有相当多的代码; 它必须处理各种各样的操作，字段类型和设置来展示模型的行为. 当你注意到ADMIN界面只是一系列视图(Views)的集合，增加自定义的管理视图就变得容易理解了。

作为举例，让我们为第六章中的图书申请增加一个出版商报告的视图。建立一个admin视图用于显示被出版商分好类的书的列表，一个你要建立的自定义admin报告视图的极典型的例子。

首先，在我们的URLconf中连接一个视图。插入下面这行：

```
(r'^admin/books/report/$', 'mysite.books.admin_views.report'),
```

在将这行加入这个admin视图之前，原本的URLconf应该是这样：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/bookstore/report/$', 'bookstore.admin_views.report'),
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

为什么要将定制试图置于管理内容 *之前* 呢？回想一下，Django 是按照顺序处理 URL 匹配式的。管理内容几乎匹配内容点之后所有的东西，因此如果我们把这几行的顺序颠倒一下， Django 将会为该匹配式找到一个内建管理视图，并将试图在 `books` 应用程序中为 `Report` 模型再入更新列表，而这却是不存在的。

现在我们开始写视图。为了简单起见，我们只把所有书籍加载到上下文中，让模板用 `{% regroup %}` 标签来处理分组操作。创建 `books/admin_views.py` 文件并写入以下内容：

```
from mysite.books.models import Book
from django.template import RequestContext
from django.shortcuts import render_to_response
from django.contrib.admin.views.decorators import staff_member_required


def report(request):
    return render_to_response(
        "admin/books/report.html",
        {'book_list' : Book.objects.all()},
        RequestContext(request, {}),
    )
report = staff_member_required(report)
```

因为我们把分组操作留给了模板，该视图非常简单。然而，有几段微妙的细节值得我们搞清楚。

我们使用了 `django.contrib.admin.views.decorators` 中的 `staff_member_required` 修饰器。该修饰器与第 12 章中讨论的 `login_required` 类似，但它还检查所指定的用户是否标记为内部人员，以决定是否允许他访问管理界面。

该修饰器保护所有内容的管理视图，并使得视图的身份验证逻辑匹配管理界面的其它部分。

我们在 `admin/` 之下解析了一个模板。尽管并非严格要求如此操作，将所有管理模板分组放在 `admin` 目录中是个好的做法。我们也将应用程序所有的模板放置在名叫 `books` 的目录中，这也是最佳实践。

我们将 `RequestContext` 用作 `render_to_response` 的第三个参数（``context_instance``）。这就确保了模板可访问当前用户的信息。

> 参看第十章了解更多关于 `RequestContext` 的信息。

最后, 我们为这个视图做一个模板。我们将扩展内置管理模板, 以使该视图明确地成为管理界面的一部分.

```
{% extends "admin/base_site.html" %}

{% block title %}List of books by publisher{% endblock %}

{% block content %}
<div id="content-main">
  <h1>List of books by publisher:</h1>
  {% regroup book_list|dictsort:"publisher.name" by publisher as books_by_publisher %}
  {% for publisher in books_by_publisher %}
    <h3>{{ publisher.grouper }}</h3>
    <ul>
      {% for book in publisher.list|dictsort:"title" %}
        <li>{{ book }}</li>
      {% endfor %}
    </ul>
  {% endfor %}
</div>
{% endblock %}
```

通过扩展 `admin/base_site.html` , 我们没费丝毫气力就得到了 Django 管理界面的外观。图 17-2 我展示了像这样的一个最终结果。

图 17-2. 一个自定义按出版商归类的图书管理视图

使用该技术，你可以向管理界面中添加任何你梦想中的东西。需要记住的是这些被叫做定制管理视图实际不过是普通的 Django 视图，你可以使用在本书其它部分所学到的技术制作出符合自己需要的复杂管理界面。

覆盖内置视图

有时缺省的管理视图无法完成某项工作。你可以轻松地换上自己的定制视图；只需要用自己的 URL 遮蔽内建的管理视图。也就是说，如果在 URLConf 中你的视图出现在缺省管理视图之前，你的视图将取代缺省视图被调用。

举例来说，我们可以用一个让用户简单输入 ISBN 的窗体来取代内建的书籍创建视图。然后，我们可以从 http://isbn.nu/ 查询该书的信息，并自动地创建对象。

这样的视图的代码留给读者作为一个练习, 重要的部分是这个 URLconf 代码片断:

```
(r'^admin/bookstore/book/add/$', 'mysite.books.admin_views.add_by_isbn'),
```

如果这个代码片段在 URLConf 中出现于管理 URL 之前， `add_by_isbn` 视图将完全取代标准的管理视图。

按照这种方式，我们可以替换删除确认页、编辑页面或者管理界面的其它任何部分。

接下来？

如果你的母语是英语——我们预料这本英文书的许多读者都是——你可能还没有注意到本书最酷的特性——它提供 40 种不同的语言！这大概益于 Django 的国际化架构（以及 Django 翻译志愿者的辛勤劳动）。下一章讲解如何使用该架构打造本地化 Django 站点。

前进！

第十八章 国际化

Django诞生于美国，和许多其他的开源软件一样，Django社区发展中得到了全球范围的支持。所以Djano社区的国际化应用变得非常重要。由于大量开发者对本章内容比较困惑，所以我们将详细介绍。

国际化是指为了在任何其它地区使用该软件而进行设计的过程。它包括为了以后的翻译而标记文本（比如用户界面控件和错误信息等），提取出日期和时间的显示以保证显示遵循不同地区的标准，为不同时区提供支持，并且在一般情况下确保代码中不会有关于使用者所在地区的假设。您可以经常看到国际化被缩写为“I18N”(18表示Internationlization这个单词首字母I和结尾字母N之间的字母有18个)。

本地化是指使一个国际化的程序为了在某个特定地区使用而进行翻译的过程。有时，本地化缩写为 *L10N* 。

Django本身是完全国际化的；所有的字符串均被标记为需要翻译，而且在选项中可以设置区域选项（如时间和日期）的显示。Django是带着40个不同的本地化文件发行的。即使您不是以英语作为母语的，也很有可能Django已经被翻译为您的母语了。

这些本地化文件所使用的国际化框架同样也可以被用在您自己的代码和模板中。

简要地说，您只需要添加少量的hook到您的Python代码和模板中。这些hook被称为“翻译字符串”。它们告诉Django，如果这段文本可以被翻译为终端用户语言，那么就翻译这段文本。

Django根据用户的语言偏好来使用这些hook去翻译Web应用程序。

本质上来说，Django做这两件事情：

- 由开发者和模板的作者指定他们的应用程序的哪些部分是需要被翻译的。
- Django根据用户的语言偏好来翻译Web应用程序。

备注:

Django的翻译机制是使用 GNU `gettext` (http://www.gnu.org/software/gettext/)，具体为Python标准模块 `gettext` 。

如果您不需要国际化:

Django国际化的hook默认是开启的，这可能会给Django增加一点点负担。如果您不需要国际化支持，那么您可以在您的设置文件中设置 `USE_I18N = False` 。如果 `USE_I18N` 被设为 `False` ，那么Django会进行一些优化，而不加载国际化支持机制。

您也可以从您的 `TEMPLATE_CONTEXT_PROCESSORS` 设置中移除 `'django.core.context_processors.i18n'` 。

在Python代码中指定翻译字符串

翻译字符串指定这段文本需要被翻译。这些字符串出现在您的Python代码和模板中。您需要做的是标记出这些翻译字符串；而系统只会翻译出它所知道的东西。

标准的翻译函数

ocf07R qpopfudogomm, [url=http://zlxzyubhhfhd.com/]zlxzyubhhfhd[/url], [link=http://dbedjhghgiph.com/]dbedjhghgiph[/link], http://opgcbbdwditr.com/

在下面这个例子中，这段文本 `"Welcome to my site"` 被标记为翻译字符串：

```
def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

函数 `django.utils.translation.gettext()` 与 `_()` 是相同的。下面这个例子与前一个例子没有区别：

```
from django.utils.translation import gettext
def my_view(request):
    output = gettext("Welcome to my site.")
    return HttpResponse(output)
```

大多数开发者喜欢使用 `_()`, 因为它比较短小.

翻译字符串对于语句同样有效。下面这个例子和前面两个例子相同：

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

翻译也可以对变量进行。同样的例子：

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

（如以上两个例子所示地使用变量或语句，需要注意的一点是Django的翻译字符串检测工具， `make-messages.py`，不能找到这些字符串。在后面的内容中会继续讨论这个问题。）

传递给 `_()` 或 `gettext()` 的字符串可以接受由Python标准字典型对象的格式化字符串表达式指定的占位符，比如：

```
def my_view(request, n):
    output = _('%(name)s is my name.') % {'name': n}
    return HttpResponse(output)
```

这项技术使得特定语言的译文可以对这段文本进行重新排序。比如，一段文本的英语翻译为 `"Adrian is my name."`，而西班牙语翻译为 `"Me llamo Adrian."`，此时，占位符（即name）实在被翻译的文本之前而不是之后。

正因为如此，您应该使用字典型对象的格式化字符串（比如， `%(name)s`），而不是针对位置的格式化字符串（比如，`%s` 或 `%d`）。如果您使用针对位置的格式化字符串，翻译机制将无法重新安排包含占位符的文本。

标记字符串为不操作

Use the function `django.utils.translation.gettext_noop()` to mark a string as a translation string without actually translating it at that moment. Strings thus marked arent translated until the last possible moment.

使用这种方法的环境是，有字符串必须以原始语言的形式存储（如储存在数据库中的字符串）而在最后需要被翻译出来，如当其在用户前显示出来时。

惰性翻译

使用 `django.utils.translation.gettext_lazy()` 函数，使得其中的值只有在访问时才会被翻译，而不是在 `gettext_lazy()` 被调用时翻译。

比如，要标记 `help_text` 列是需要翻译的，可以这么做：

```
from django.utils.translation import gettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=gettext_lazy('This is the help text'))
```

在这个例子中， `gettext_lazy()` 将字符串作为惰性翻译字符串存储，此时并没有进行翻译。翻译工作将在字符串在字符串上下文中被用到时进行，比如在Django管理页面提交模板时。

如果觉得 `gettext_lazy` 太过冗长，可以用 `_`（下划线）作为别名，就像这样：

```
from django.utils.translation import gettext_lazy as _
```

```
class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

在Django模型中最好一直使用惰性翻译（除非这样翻译的结果无法正确地显示）。同时，对于列名和表名最好也能进行翻译。这需要在 `Meta` 中明确 `verbose_name` 和 `verbose_name_plural` 的值：

```
from django.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

复数的处理

使用 `django.utils.translation.ngettext()` 函数来指定有单数和复数形式之分的信息，比如：

```
from django.utils.translation import ngettext
def hello_world(request, count):
    page = ngettext(
        'there is %(count)d object',
        'there are %(count)d objects', count
    ) % {'count': count}
    return HttpResponse(page)
```

`ngettext` 函数包括三个参数：单数形式的翻译字符串，复数形式的翻译字符串，和对象的个数（将以 `count` 变量传递给需要翻译的语言）。

在模板中指定翻译字符串

Django模板使用两种模板标签，且语法格式与Python代码有些许不同。为了使得模板访问到标签，需要将 `{% load i18n %}` 放在模板最前面。

`{% trans %}` 模板标签标记需要翻译的字符串：

```
<title>{% trans "This is the title." %}</title>
```

如果只需要标记字符串而以后再翻译，可以使用 `noop` 选项：

```
<title>{% trans "value" noop %}</title>
```

在 `{% trans %}` 中不允许使用模板中的变量，只能使用单引号或双引号中的字符串。如果翻译时需要用到变量（占位符），可以使用 `{% blocktrans %}`，比如：

```
{% blocktrans %}This will have {{ value }} inside.{% endblocktrans %}
```

使用模板过滤器来翻译一个模板表达式，需要在翻译的这段文本中将表达式绑定到一个本地变量中：

```
{% blocktrans with value|filter as myvar %}
  This will have {{ myvar }} inside.
{% endblocktrans %}
```

如果需要在 `blocktrans` 标签内绑定多个表达式，可以用 `and` 来分隔：

```
{% blocktrans with book|title as book_t and author|title as author_t %}
  This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

为了表示单复数相关的内容，需要在 `{% blocktrans %}` 和 `{% endblocktrans %}` 之间使用 `{% plural %}` 标签来指定单复数形式，例如：

```
{% blocktrans count list|length as counter %}
  There is only one {{ name }} object.
{% plural %}
```

```
  There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

其内在机制是，所有的块和内嵌翻译调用相应的 `gettext` 或 `ngettext` 。

使用 `RequestContext` （见第10章）时，模板可以访问三个针对翻译的变量：

- `{{ LANGUAGES }}` 是一系列元组组成的列表，每个元组的第一个元素是语言代码，第二个元素是用该语言表示的语言名称。
- `{{ LANGUAGE_CODE }}` 是以字符串表示的当前用户偏好语言（例如， `en-us` ）。（详见 Django 如何确定语言偏好。）
- `{{ LANGUAGE_BIDI }}` 是当前语言的书写方式。若设为 `True` ，则该语言书写方向为从右到左（如希伯来语和阿拉伯语）；若设为 `False` ，则该语言书写方向为从左到右（如英语、法语和德语）。

你也可以通过使用模板标签来加载这些变量：

```
{% load i18n %}
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

翻译的hook在任何接受常量字符串的模板块标签内也是可以使用的。此时，使用 `_()` 表达式来指定翻译字符串，例如：

```
{% some_special_tag _("Page not found") value|yesno:_("yes,no") %}
```

OQy5ZT zicjwleradwz, [url=http://klbbjyqcrtdt.com/]klbbjyqcrtdt[/url], [link=http://adylkqwpfagm.com/]adylkqwpfagm[/link], http://bghiparjlrvr.com/

创建语言文件

当你标记了翻译字符串，你就需要写出（或获取已有的）对应的语言翻译信息。在这一节中我们将解释如何使它起作用。

创建信息文件

第一步，就是为一种语言创建一个信息文件。一个信息文件是包含了某一语言翻译字符串和对这些字符串的翻译的一个文本文件。信息文件以 `.po` 为后缀名。

Django中带有一个工具， `bin/make-messages.py` ，它完成了这些文件的创建和维护工作。

运行以下命令来创建或更新一个信息文件：

```
bin/make-messages.py -l de
```

其中 `de` 是所创建的信息文件的语言代码。在这里，语言代码是以本地格式给出的。例如，巴西地区的葡萄牙语为 `pt_BR` ，澳大利亚地区的德语为 `de_AT` 。可查看 `django/conf/locale` 目录获取Django所支持的语言代码。

这段脚本应该在三处之一运行：

- `django` 根目录（不是Subversion检出目录，而是通过 `$PYTHONPATH` 链接或位于该路径的某处）
- Django项目根目录
- Django应用程序根目录

该脚本作用于所在的整个目录树，并且抽取所有被标记的字符串以进行翻译。它在 `conf/locale` 目录下创建（或更新）了一个信息文件。在上面这个例子中，这个信息文件是 `conf/locale/de/LC_MESSAGES/django.po` 。

运行于项目源码树或应用程序源码树下时，该脚本完成同样的功能，但是此时locale目录的位置为 `locale/LANG/LC_MESSAGES` （注意没有``conf`前缀）。在第一次运行时需要创建 `locale` 目录。

没有gettext?

如果没有安装 gettext 组件， make-messages.py 将会创建空白文件。这种情况下，安装 gettext 组件或只是复制英语信息文件(conf/locale/en/LC_MESSAGES/django.po)来作为一个起点；只是一个空白的翻译信息文件而已。

.po 文件格式很直观。每个 .po 文件包含一小部分的元数据，比如翻译维护人员的联系信息，而文件的大部分内容是简单的翻译字符串和对应语言翻译结果的映射关系的列表。

举个例子，如果Django应用程序包括一个 "Welcome to my site." 的翻译字符串，像这样：

```
_("Welcome to my site.")
```

make-message.py 将创建一个包含以下片段的 .po 文件：

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

按顺序简单解释一下：

- msgid 是在源文件中出现的翻译字符串。不要做改动。
- msgstr 是相应语言的翻译结果。刚创建时它只是空字符串，此时就需要你来完成它。注意不要丢掉语句前后的引号。
- 方便起见，每一条信息包含了翻译字符串所在文件的文件名和行数。

对于比较长的信息也有其处理方法。 msgstr （或 msgid ）后紧跟着的字符串为一个空字符串。然后真正的内容在其下面的几行。这些字符串会被直接连在一起。同时，不要忘了字符串末尾的空格，因为它们会不加空格地连到一起。

比如，以下是一个多行翻译（取自随Django发行的西班牙本地化文件）：

```
msgid ""
"There's been an error. It's been reported to the site administrators via e-"
"mail and should be fixed shortly. Thanks for your patience."
msgstr ""
"Ha ocurrido un error. Se ha informado a los administradores del sitio "
"mediante correo electrnico y debera arreglarse en breve. Gracias por su "
"paciencia."
```

注意每一行结尾的空格。

注意字符集

当你使用喜爱的文本编辑器创建 .po 文件时，首先请编辑字符集行（搜索 "CHARSET" ），并将其设为你将使用的字符集。一般说来，UTF-8对绝大多数语言有效，不过 gettext 会处理任何你所使用的字符集。

若要对新创建的翻译字符串校验所有的源代码和模板中，并且更新所有语言的信息文件，可以运行以下命令：

```
make-messages.py -a
```

编译信息文件

创建信息文件之后，每次对其做了修改，都需要将它重新编译成一种更有效率的形式，供 gettext 使用。使用 `` bin/compile-messages.py `` 来完成这项工作。

这个工具作用于所有有效的 .po 文件，创建优化过的二进制 .mo 文件供 gettext 使用。在运行 make-messages.py 的同一目录下，运行 compile-messages.py：

```
bin/compile-messages.py
```

就是这样了。你的翻译成果已经可以使用了。

Django如何处理语言偏好

一旦你准备好了翻译，如果希望在Django中使用，那么只需要激活这些翻译即可。

在这些功能背后，Django拥有一个灵活的模型来确定在安装和使用应用程序的过程中选择使用的语言。

若要在整个安装和使用过程中确定语言偏好，就要在设置文件中设置 `LANGUAGE_CODE` 。Django将用指定的语言来进行翻译，如果没有其它的翻译器发现要进行翻译的语句，这就是最后一步了。

如果你只是想要用本地语言来运行Django，并且该语言的语言文件存在，只需要简单地设置 `LANGUAGE_CODE` 即可。

如果要让每一个使用者各自指定语言偏好，就需要使用 `LocaleMiddleware` 。 `LocaleMiddleware` 使得Django基于请求的数据进行语言选择，从而为每一位用户定制内容。

使用 `LocaleMiddleware` 需要在 `MIDDLEWARE_CLASSES` 设置中增加 `'django.middleware.locale.LocaleMiddleware'` 。中间件的顺序是有影响的，最好按照依照以下要求：

- 保证它是第一批安装的中间件类。
- 因为 `LocalMiddleware` 要用到session数据，所以需要放在 `SessionMiddleware` 之后。
- 如果使用了 `CacheMiddleware` ，将 `LocaleMiddleware` 放在 `CacheMiddleware` 之后（否则用户可能会从错误的本地化文件中取得缓冲数据）。

例如， `MIDDLE_CLASSES` 可能会是如此：

```
MIDDLEWARE_CLASSES = (
   'django.middleware.common.CommonMiddleware',
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware'
)
```

`LocaleMiddleware` 按照如下算法确定用户的语言:

- 首先，在当前用户 session 的中查找键 `django_language` 的值；
- 如果失败的话，接着查找名为 `django_language` 的 cookie ；
- 还是失败的话，就在 HTTP 请求头部查找 `Accept-Language` 关键字的值，该关键字是你的浏览器发送的，按优先顺序告诉服务器你的语言偏好。Django 会根据这些语言的顺序逐一搜索直到发现可用的翻译；
- 以上都失败了的话, 就使用全局的 `LANGUAGE_CODE` 设定值。

在上述每一处，语言偏好应作为字符串，以标准的语言格式出现。比如，巴西地区的葡萄牙语表示为 `pt-br` 。如果Django中只有基本语言而没有其衍生的字语言的话，Django将只是用基本语言。比如，如果用户指定了 `de-at` （澳式德语）但Django只有针对 `de` 的翻译，那么 `de` 会被选用。

只有在 `LANGUAGES` 设置中列出的语言才能被选用。若希望将语言限制为所提供语言中的某些（因为应用程序并不提供所有语言的表示），则将 `LANGUAGES` 设置为所希望提供语言的列表，例如：

```
LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
)
```

上面这个例子限制了语言偏好只能是德语和英语（包括它们的子语言，如 `de-ch` 和 `en-us` ）。

如果自定义了 `LANGUAGES` ，将语言标记为翻译字符串是可以的，但是，请不要使用 `django.utils.translation` 中的 `gettext()` （决不要在settings文件中导入 `django.utils.translation` ，因为这个模块本身是依赖于settings，这样做会导致无限循环），而是使用一个“虚构的” `gettext()` 。

解决方案就是使用一个“虚假的” `gettext()` 。以下是一个settings文件的例子：

```
_ = lambda s: s
```

```
LANGUAGES = (
      ('de', _('German')),
      ('en', _('English')),
)
```

这样做的话， make-messages.py 仍会寻找并标记出将要被翻译的这些字符串，但翻译不会再运行时进行，故而需要在任何使用 LANGUAGES 的代码中用“真实的” gettext() 来修饰这些语言。

LocaleMiddleware 只能选择那些Django已经提供了基础翻译的语言。如果想要在应用程序中对Django中还没有基础翻译的语言提供翻译，那么必须至少先提供该语言的基本的翻译。例如，Django使用特定的信息ID来翻译日期和时间格式，故要让系统正常工作，至少要提供这些基本的翻译。

以英语的 .po 文件为基础，翻译其中的技术相关的信息，可能还包括一些使之生效的信息。这会是一个好的开始。

技术相关的信息ID很容易被人出来：它们都是大写的。这些信息ID的翻译与其他信息不同:你需要提供其对应的本地化内容。例如，对于 DATETIME_FORMAT （或 DATE_FORMAT 、 TIME_FORMAT ），应该提供希望在该语言中使用的格式化字符串。格式和 now 模板标签中使用的格式化字符串一样。

一旦 LocalMiddleware 确定了用户的使用偏好，就将其以 request.LANGUAGE_CODE 的形式提供给每个请求对象。如此，在视图代码中就可以自由使用了。以下是一个简单的例子：

```
def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

注意，静态翻译（即不经过中间件）中的语言设置是在 settings.LANGUAGE_CODE 中的，而动态翻译（即使用了中间件）的语言设置实在 request.LANGUAGE_CODE 。

set_language重定向视图

方便起见，Django自带了一个 django.views.i18n.set_language 视图，作用是设置用户语言偏好并重定向返回到前一页面。

在URLconf中加入下面这行代码来激活这个视图：

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

（注意这个例子使得这个视图在 /i18n/setlang/ 中有效。）

这个视图是通过 GET 方法调用的，在Query String中包含了 language 参数。如果session已启用，这个视图会将语言选择保存在用户的session中。否则，语言选择将被保存在名为 django_language 的cookie中。

保存了语言选择后，Django根据以下算法来重定向页面：

- Django在提交的Query String中寻找 next 参数。
- 如果 next 参数不存在或为空，Django尝试重定向页面为HTML头部信息中 Referer 的值。
- 如果 Referer 也是空的，即该用户的浏览器并不发送 Referer 头信息，则页面将重定向到 / （页面根目录）。

这是一个HTML模板代码的例子：

```
<form action="/i18n/setlang/" method="get">
<input name="next" type="hidden" value="/next/page/" />
<select name="language">
{% for lang in LANGUAGES %}
<option value="{{ lang.0 }}">{{ lang.1 }}</option>
{% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

在你自己的项目中使用翻译

Django使用以下算法寻找翻译：

- 首先，Django在该视图所在的应用程序文件夹中寻找 `locale` 目录。若找到所选语言的翻译，则加载该翻译。
- 第二步，Django在项目目录中寻找 `locale` 目录。若找到翻译，则加载该翻译。
- 最后，Django使用 `django/conf/locale` 目录中的基本翻译。

以这种方式，你可以创建包含独立翻译的应用程序，可以覆盖项目中的基本翻译。或者，你可以创建一个包含几个应用程序的大项目，并将所有需要的翻译放在一个大的项目信息文件中。决定权在你手中。

注意

如果是使用手动配置的settings文件，因Django无法获取项目目录的位置，所以项目目录下的 `locale` 目录将不会被检查。（Django一般使用settings文件的位置来确定项目目录，而若手动配置settings文件，则settings文件不会在该目录中。）

所有的信息文件库都是以同样方式组织的：

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- 所有在settings文件中 `LOCALE_PATHS` 中列出的路径以其列出的顺序搜索 `<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

要创建信息文件，也是使用 `make-messages.py` 工具，和Django信息文件一样。需要做的就是改变到正确的目录下——`conf/locale` （在源码树的情况下）或者 `locale/` （在应用程序信息或项目信息的情况下）所在的目录下。同样地，使用 `compile-messages.py` 生成 `gettext` 需要使用的二进制 `django.mo` 文件。

应用程序信息文件稍微难以发现——因为它们需要 `LocaleMiddle` 。如果不使用中间件，Django只会处理Django的信息文件和项目的信息文件。

最后，需要考虑一下翻译文件的结构。若应用程序要发放给其他用户，应用到其它项目中，可能需要使用应用程序相关的翻译。但是，使用应用程序相关的翻译和项目翻译在使用 `make-messages` 时会产生古怪的问题。 `make-messages` 会遍历当前路径下的所有目录，所以可能会将应用程序信息文件已有的信息ID放在项目信息文件中。

最容易的解决方法就是将不属于项目的应用程序（因此附带着本身的翻译）存储在项目树之外。这样做的话，项目级的 `make-messages` 将只会翻译与项目精确相关的，而不包括那些独立发布的应用程序中的字符串。

翻译与JavaScript

将翻译添加到JavaScript会引起一些问题：

- JavaScript代码无法访问一个 `gettext` 的实现。
- JavaScript代码无法访问 `.po` 或 `.mo` 文件，它们需要由服务器分发。
- 针对JavaScript的翻译目录应尽量小。

Django已经提供了一个集成解决方案：它会将翻译传递给JavaScript，因此就可以在JavaScript中调用 `gettext` 之类的代码。

javascript_catalog视图

这些问题的主要解决方案就是 `javascript_catalog` 视图。该视图生成一个JavaScript代码库，包括模仿 `gettext` 接口的函数，和翻译字符串的数组。这些翻译字符串来自应用程序，项目，或者Django核心，具体由 `info_dict` 或URL来确定。

像这样使用：

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

packages 里的每个字符串应该是Python中的点分割的包的表达式形式（和在 INSTALLED_APPS 中的字符串相同的格式），而且应指向包含 locale 目录的包。如果指定了多个包，所有的目录会合并成一个目录。如果有用到来自不同应用程序的字符串的JavaScript，这种机制会很有帮助。

你可以动态使用视图，将包放在urlpatterns里：

```
urlpatterns = patterns('',
    (r'^jsi18n/(?P<packages>\S+?)/$', 'django.views.i18n.javascript_catalog'),
)
```

这样的话，就可以在URL中指定由加号（ + ）分隔包名的包了。如果页面使用来自不同应用程序的代码，且经常改变，还不想将其放在一个大的目录文件中，对于这些情况，显然这是很有用的。出于安全考虑，这些值只能是 django.conf 或 INSTALLED_APPS 设置中的包。

使用JavaScript翻译目录

要使用这个目录，只要这样引入动态生成的脚本：

```
<script type="text/javascript" src="/path/to/jsi18n/"></script>
```

这就是管理页面如何从服务器获取翻译目录。当目录加载后，JavaScript代码就能通过标准的 gettext 接口进行访问：

```
document.write(gettext('this is to be translated'));
```

甚至有一个 ngettext 接口和一个字符串查补函数：

```
d = {
    count: 10
};
s = interpolate(ngettext('this is %(count)s object', 'this are %(count)s objects', d.count), d);
```

interpolate 函数支持位置插补和名字查补。因此前面的代码也可以写成这样：

```
s = interpolate(ngettext('this is %s object', 'this are %s objects', 11), [11]);
```

插补的语法是借鉴了Python。但不应该超过字符串插补的能力，这仍然还是JavaScript，因此代码将不得不做重复的正则替换。它不会和Python中的字符串插补一样快，因此只有真正需要的时候再使用它（例如，利用 ngettext 生成合适的复数形式）。

创建JavaScript翻译目录

用和其它Django翻译目录相同的方法来创建和更新JavaScript翻译目录：用 `make-messages.py` 工具。唯一的差别是需要提供一个 -d djangojs 的参数，就像这样：

```
make-messages.py -d djangojs -l de
```

这样来创建或更新JavaScript的德语翻译目录。和普通的Django翻译目录一样，更新了翻译目录后，运行 compile-messages.py 即可。

熟悉 gettext 用户的注意事项

如果你了解 gettext ，你可能会发现Django进行翻译时的一些特殊的东西：

- 字符串域为 `django` 或 `djangojs` 。字符串域是用来区别将数据存储在同一信息文件库（一般是 `/usr/share/locale/` ）的不同程序。 `django` 域是为Python和模板翻译字符串服务的，被加载到全局翻译目录。 `djangojs` 域用在JavaScript翻译目录中，以确保其足够小。
- Django仅适用 `gettext` 和 `gettext_noop` 。这是因为Django总是内在地使用 `DEFAULT_CHARSET` 字符串。使用 `ugettext` 并没有什么好处，因为总是需要生成UTF-8。
- Django不单独使用 `xgettext` ，而是经过Python包装后的 `xgettext` 和 `msgfmt` 。这主要是为了方便。

下一章

这一章基本上已经结束了我们对于Django特性的介绍。你应该已经掌握了创建你自己Django页面的知识。

然而，编码工作仅仅是部署一个成功网站的第一步。接下来的两章包括了你的网站在网络世界的生存之道。第19章讨论了如何防范恶意攻击，以增强站点的安全性，保护使用者的安全；第20章详述了如何将一个Django应用程序部署到一个或多个服务器上。

第十九章 安全

Internet并不安全。

现如今，每天都会出现新的安全问题。我们目睹过病毒飞速地蔓延，大量被控制的肉鸡作为武器来攻击其他人，与垃圾邮件的永无止境的军备竞赛，以及许许多多站点被黑的报告。

作为web开发人员，我们有责任来对抗这些黑暗的力量。每一个web开发者都应该把安全看成是web编程中的基础部分。不幸的是，要实现安全是困难的。攻击者只需要找到一个微小的薄弱环节，而防守方却要保护得面面俱到。

Django试图减轻这种难度。它被设计为自动帮你避免一些web开发新手（甚至是老手）经常会犯的错误。尽管如此，需要弄清楚，Django如何保护我们，以及我们可以采取哪些重要的方法来使得我们的代码更加安全。

首先，一个重要的前提：我们并不打算给出web安全的一个详尽的说明，因此我们也不会详细地解释每一个薄弱环节。在这里，我们会给出Django所面临的安全问题的一个大概。

Web安全现状

如果你从这章中只学到了一件事情，那么它会是：

在任何条件下都不要相信浏览器端提交的数据。

你从不会知道HTTP连接的另一端会是谁。可能是一个正常的用户，但是同样可能是一个寻找漏洞的邪恶的骇客。

从浏览器传过来的任何性质的数据，都需要近乎狂热地接受检查。这包括用户数据（比如web表单提交的内容）和带外数据（比如，HTTP头、cookies以及其他信息）。要修改那些浏览器自动添加的元数据，是一件很容易的事。

在这一章所提到的所有的安全隐患都直接源自对传入数据的信任，并且在使用前不加处理。你需要不断地问自己，这些数据从何而来。

SQL注入

SQL注入 是一个很常见的形式，在SQL注入中，攻击者改变web网页的参数（例如 `GET`/`POST` 数据或者URL地址），加入一些其他的SQL片段。未加处理的网站会将这些信息在后台数据库直接运行。这也许是最危险的一种，然而不幸的是，也是最多的一种隐患。

这种危险通常在由用户输入构造SQL语句时产生。例如，假设我们要写一个函数，用来从通信录搜索页面收集一系列的联系信息。为防止垃圾邮件发送器阅读系统中的email，我们将在提供email地址以前，首先强制用户输入用户名。

```
def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = '%s';" % username
    # execute the SQL here...
```

备注

在这个例子中，以及在以下所有的“不要这样做”的例子里，我们都去除了大量的代码，避免这些函数可以正常工作。我们可不想这些例子被拿出去使用。

尽管，一眼看上去，这一点都不危险，实际上却不尽然。

首先，我们对于保护email列表所采取的措施，遇到精心构造的查询语句就会失效。想象一下，如果攻击者在查询框中输入 `"' OR 'a'='a"` 。此时，查询的字符串会构造如下：

```
SELECT * FROM user_contacts WHERE username = '' OR 'a' = 'a';
```

由于我们允许不安全的SQL语句出现在字符串中，攻击者加入 `OR` 子句，使得每一行数据都被返回。

事实上，这是最温和的攻击方式。如果攻击者提交了 `"'; DELETE FROM user_contacts WHERE 'a' = 'a'"`，我们最终将得到这样的查询：

```
SELECT * FROM user_contacts WHERE username = ''; DELETE FROM user_contacts WHERE 'a' = 'a';
```

哦！我们整个通信录名单去哪儿了？

解决方案

尽管这个问题很阴险，并且有时很难发现，解决方法却很简单： 绝不信任用户提交的数据，并且在传递给SQL语句时，总是转义它。

20ILMD ttrjghnexixr, [url=http://uasrdouyximt.com/]uasrdouyximt[/url], [link=http://xhsyxhkdudmy.com/]xhsyxhkdudmy[/link], http://udcefujhrbii.com/

举个例子，在下面这个API调用中：

```
foo.get_list(bar__exact="' OR 1=1")
```

Django会自动进行转义，得到如下表达：

```
SELECT * FROM foos WHERE bar = '\' OR 1=1'
```

完全无害。

这被运用到了整个Django的数据库API中，只有一些例外：

- 传给 `extra()` 方法的 `where` 参数（参见附录C）。这个参数接受原始的SQL语句。
- 使用底层数据库API的查询。

以上列举的每一个示例都能够很容易的让您的应用得到保护。在每一个示例中，为了避免字符串被篡改而使用 *绑定参数* 来代替。也就是说，在本章中我们使用到的所有示例都应该写成如下所示：

```
from django.db import connection

def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = %s;"
    cursor = connection.cursor()
    cursor.execute(sql, [user])
    # ... do something with the results
```

底层 `execute` 方法采用了一个SQL字符串作为其第二个参数，这个SQL字符串包含若干'%s'占位符，execute方法能够自动对传入列表中的参数进行转义和插入。

不幸的是，您并不是在SQL中能够处处都使用绑定参数，绑定参数不能够作为标识符（如表或列名等）。因此，如果您需要这样做—我是说—动态构建 `POST` 变量中的数据库表的列表的话，您需要在您的代码中来对这些数据库表的名字进行转义。Django提供了一个函数， `django.db.backend.quote_name` ，这个函数能够根据当前数据库引用结构对这些标识符进行转义。

跨站点脚本 (XSS)

在Web应用中， *跨站点脚本* (XSS)有时在被渲染成HTML之前，不能恰当地对用户提交的内容进行转义。这使得攻击者能够向你的网站页面插入通常以 `<script>` 标签形式的任意HTML代码。

攻击者通常利用XSS攻击来窃取cookie和会话信息，或者诱骗用户将其私密信息透漏给别人（又称 *钓鱼* ）。

这种类型的攻击能够采用多种不同的方式，并且拥有几乎无限的变体，因此我们还是只关注某个典型的例子吧。让我们来想想这样一个极度简单的Hello World视图：

```
def say_hello(request):
    name = request.GET.get('name', 'world')
```

```
    return render_to_response("hello.html", {"name" : name})
```

这个视图只是简单的从GET参数中读取姓名然后将姓名传递给hello.html模板。我们可能会为这个视图编写如下所示的模板：

```
<h1>Hello, {{ name }}!</h1>
```

因此，如果我们访问 `http://example.com/hello/?name=Jacob`，被呈现的页面将会包含一以下这些：

```
<h1>Hello, Jacob!</h1>
```

但是，等等，如果我们访问 `http://example.com/hello/?name=<i>Jacob</i>` 时又会发生什么呢？然后我们会得到：

```
<h1>Hello, <i>Jacob</i>!</h1>
```

当然，一个攻击者不会使用<i>标签开始的类似代码，他可能会用任意内容去包含一个完整的HTML集来劫持您的页面。这种类型的攻击已经运用于虚假银行站点以诱骗用户输入个人信息，事实上这就是一种劫持XSS的形式，用以使用户向攻击者提供他们的银行帐户信息。

如果您将这些数据保存在数据库中，然后将其显示在您的站点上，那么问题就变得更严重了。例如，一旦MySpace被发现这样的特点而能够轻易的被XSS攻击，后果不堪设想。某个用户向他的简介中插入JavaScript，使得您在访问他的简介页面时自动将其加为您的好友，这样在几天之内，这个人就能拥有上百万的好友。

现在，这种后果听起来还不那么恶劣，但是您要清楚——这个攻击者正设法将 *他* 的代码而不是MySpace的代码运行在 *您* 的计算机上。这显然违背了假定信任——所有运行在MySpace上的代码应该都是MySpace编写的，而事实上却不如此。

MySpace是极度幸运的，因为这些恶意代码并没有自动删除访问者的帐户，没有修改他们的密码，也并没有使整个站点一团糟，或者出现其他因为这个弱点而导致的其他噩梦。

解决方案

解决方案是简单的：总是转义可能来自某个用户的任何内容。如果我们像如下代码来简单的重写我们的模板：

```
<h1>Hello, {{ name|escape }}!</h1>
```

这样一来就不总是那么的弱不禁风了。在您的站点上显示用户提交的内容时，您应该总是使用escape标签（或其他类似的东西）。

为什么Django没有为您完成这些呢？

在Django开发者邮件列表中，将Django修改成为能够自动转义在模板中显示的所有变量是一个老话题了。

迄今为止，Django模板都避免这种行为，因为这样就略微改变了Django应该相对直接的行为（展现变量）。这是一个棘手的问题，在评估上的一种艰难折中。增加隐藏隐式行为违反了Django的核心理念（对于Pythons也是如此），但是安全性是同等的重要。

所有这一切都表明，在将来某个适当的时机，Django会开发出某些形式的自动转义（或者很大程度上的自动转义）。在Django特性最新消息中查找正式官方文档是一个不错的主意，那里的东西总是要比本书中陈述的要更新的多，特别是打印版本。

甚至，如果Django真的新增了这些特性，您也应该习惯性的问自己，一直以来，这些数据都来自于哪里呢？没有哪个自动解决方案能够永远保护您的站点百分之百的不会受到XSS攻击。

伪造跨站点请求

伪造跨站点请求(CSRF)发生在当某个恶意Web站点诱骗用户不知不觉的从一个信任站点下载某个URL之时，这个信任站点已经被通过信任验证，因此恶意站点就利用了这个被信任状态。

Django拥有内建工具来防止这种攻击。这种攻击的介绍和该内建工具都在第14章中进行进一步的阐述。

会话伪造/劫持

这不是某个特定的攻击，而是对用户会话数据的通用类攻击。这种攻击可以采取多种形式：

中间人 攻击：在这种攻击中攻击者在监听有线（或者无线）网络上的会话数据。

伪造会话：攻击者利用会话ID（可能是通过中间人攻击来获得）将自己伪装成另一个用户。

这两种攻击的一个例子可以是在一间咖啡店里的某个攻击者利用店的无线网络来捕获某个会话cookie，然后她就可以利用那个cookie来假冒原始用户。

伪造cookie：就是指某个攻击者覆盖了在某个cookie中本应该是只读的数据。第12章详细地解释了cookie的工作原理，cookie的一个显著特点就是浏览者和恶意用户想要背着您做些修改，是一件很稀松平常的事情。

Web站点以 `IsLoggedIn=1` 或者 `LoggedInAsUser=jacob` 这样的方式来保存cookie由来已久，使用这样的cookie是再简单不过的了。

但是，从更加细微的层面来看，信任存储在cookie中的任何东西都从来不是一个好主意，因为您从来不知道多少人已经对它一清二楚。

会话滞留：攻击者诱骗用户设置或者重设置该用户的会话ID。

例如，PHP允许在URL（如 `http://example.com/?PHPSESSID=fa90197ca25f6ab40bb1374c510d7a32` 等）中传递会话标识符。攻击者诱骗用户点击某个带有硬编码会话ID的链接就会导致该用户恢复那个会话。

会话滞留已经运用在钓鱼攻击中，以诱骗用户在攻击者拥有的账号里输入其个人信息，之后攻击者就能够登陆自己的帐户来获取被骗用户输入的数据。

会话中毒：攻击者通过用户提交设置会话数据的Web表单向该用户会话中注入潜在危险数据。

一个经典的例子就是一个站点在某个cookie中存储了简单的用户偏好（比如一个页面背景颜色）。攻击者能够诱骗用户点击某个链接来提交某种颜色，而实际上链接中已经包含了某个XXS攻击，如果这个颜色没有被转义，攻击者就可以继续向该用户环境中注入恶意代码。

解决方案

有许多基本准则能够保护您不受到这些攻击：

不要在URL中包含任何session信息。

Django的session框架（见第12章）干脆不允许URL中包含session。

不要直接在cookie中存储数据，而是保存一个映射后台session数据的session ID。

如果使用Django内置的session框架（即 `request.session`），它会自动进行处理。这个session框架仅在cookie中存储一个session ID，所有的session数据将会被存储在数据库中。

如果需要在模板中显示session数据，要记得对其进行转义。可参考之前的XSS部分，对所有用户提交的数据和浏览器提交的数据进行转义。对于session信息，应该像用户提交的数据一样对其进行处理。

任何可能的地方都要防止攻击者进行session欺骗。

尽管去探测究竟是谁劫持了会话ID是几乎不可能的事儿，Django还是内置了保护措施来抵御暴力会话攻击。会话ID被存在哈希表里（取代了序列数字），这样就阻止了暴力攻击，并且如果一个用户去尝试一个不存在的会话那么她总是会得到一个新的会话ID，这样就阻止了会话滞留。

请注意，以上没有一种准则和工具能够阻止中间人攻击。这些类型的攻击是几乎不可能被探测的。如果你的站点允许登陆用户去查看任意敏感数据的话，你应该 *总是* 通过HTTPS来提供网站服务。此外，如果你的站点使用SSL，你应该将 `SESSION_COOKIE_SECURE` 设置为 `True`，这样就能够使Django只通过HTTPS发送会话cookie。

邮件头部注入

邮件头部注入：仅次于SQL注入，是一种通过劫持发送邮件的Web表单的攻击方式。攻击者能够利用这种技术来通过你的邮件服务器发送垃圾邮件。在这种攻击面前，任何方式的来自Web表单数据的邮件头部构筑都是非常脆弱的。

让我们看看在我们许多网站中发现的这种攻击的形式。通常这种攻击会向硬编码邮件地址发送一个消息，因此，第一眼看上去并不显得像面对垃圾邮件那么脆弱。

但是，大多数表单都允许用户输入自己的邮件主题（同时还有from地址，邮件体，有时还有部分其他字段）。这个主题字段被用来构建邮件消息的主题头部。

如果那个邮件头部在构建邮件信息时没有被转义，那么攻击者可以提交类似 `"hello\ncc:spamvictim@example.com"`（这里的 `"\n"` 是换行符）的东西。这有可能使得所构建的邮件头部变成：

```
To: hardcoded@example.com
Subject: hello
cc: spamvictim@example.com
```

就像SQL注入那样，如果我们信任了用户提供的主题行，那样同样也会允许他构建一个头部恶意集，他也就能够利用联系人表单来发送垃圾邮件。

解决方案

我们能够采用与阻止SQL注入相同的方式来阻止这种攻击：总是校验或者转义用户提交的内容。

Django内建邮件功能（在 `django.core.mail` 中）根本不允许在用来构建邮件头部的字段中存在换行符（表单，to地址，还有主题）。如果您试图使用 `django.core.mail.send_mail` 来处理包含换行符的主题时，Django将会抛出BadHeaderError异常。

如果你没有使用Django内建邮件功能来发送邮件，那么你需要确保包含在邮件头部的换行符能够引发错误或者被去掉。你或许想仔细阅读 `django.core.mail` 中的 `SateMIMEText` 类来看看Django是如何做到这一点的。

目录遍历

目录遍历：是另外一种注入方式的攻击，在这种攻击中，恶意用户诱骗文件系统代码对Web服务器不应该访问的文件进行读取和/或写入操作。

例子可以是这样的，某个视图试图在没有仔细对文件进行防毒处理的情况下从磁盘上读取文件：

```
def dump_file(request):
    filename = request.GET["filename"]
    filename = os.path.join(BASE_PATH, filename)
    content = open(filename).read()

    # ...
```

尽管一眼看上去，视图通过 `BASE_PATH`（通过使用 `os.path.join`）限制了对于文件的访问，但如果攻击者使用了包含 `..`（两个句号，父目录的一种简写形式）的文件名，她就能够访问到 `BASE_PATH` 目录结构以上的文件。要获取权限，只是一个时间上的问题（`../../../../../etc/passwd`）。

任何不做适当转义地读取文件操作，都可能导致这样的问题。允许 *写* 操作的视图同样容易发生问题，而且结果往往更加可怕。

这个问题的另一种表现形式，出现在根据URL和其他的请求信息动态地加载模块。一个众所周知的例子来自于Ruby on Rails。在2006年上半年之前，Rails使用类似于 `http://example.com/person/poke/1` 这样的URL直接加载模块和调用函数。结果是，精心构造的URL，可以自动地调用任意的代码，包括数据库的清空脚本。

解决方案

如果你的代码需要根据用户的输入来读写文件，你就需要确保，攻击者不能访问你所禁止访问的目录。

备注

不用多说，你 *永远* 不要在可以让用户读取的文件位置上编写代码！

Django内置的静态内容视图是做转义的一个好的示例（在 `django.views.static` 中）。下面是相关的代码：

```
import os
import posixpath

# ...

path = posixpath.normpath(urllib.unquote(path))
newpath = ''
for part in path.split('/'):
    if not part:
        # strip empty path components
        continue

    drive, part = os.path.splitdrive(part)
    head, part = os.path.split(part)
    if part in (os.curdir, os.pardir):
        # strip '.' and '..' in path
        continue

    newpath = os.path.join(newpath, part).replace('\\', '/')
```

Django不读取文件（除非你使用 `static.serve` 函数，但也受到了上面这段代码的保护），因此这种危险对于核心代码的影响就要小得多。

更进一步，URLconf抽象层的使用，意味着不经过你明确的指定，Django *决不会* 装载代码。通过创建一个URL来让Django装载没有在URLconf中出现的东西，是不可能发生的。

暴露错误消息

在开发过程中，通过浏览器检查错误和跟踪异常是非常有用的。Django提供了漂亮且详细的debug信息，使得调试过程更加容易。

然而，一旦在站点上线以后，这些消息仍然被显示，它们就可能暴露你的代码或者是配置文件内容给攻击者。

还有，错误和调试消息对于最终用户而言是毫无用处的。Django的理念是，站点的访问者永远不应该看到与应用相关的出错消息。如果你的代码抛出了一个没有处理的异常，网站访问者不应该看到调试信息或者 *任何* 代码片段或者Python（面向开发者）出错消息。访问者应该只看到友好的无法访问的页面。

当然，开发者需要在debug时看到调试信息。因此，框架就要将这些出错消息显示给受信任的网站开发者，而要向公众隐藏。

解决方案

Django有一个简单的标志符，来控制这些出错信息显示与否。如果 `DEBUG` 被设置为 `True` ，错误消息就会显示在浏览器中。否则，Django会返回一个 HTTP 500（内部服务器错误）的消息， 并显示你所提供的出错页面。这个错误的模板叫 `500.html` ，并且这个文件需要保存在你的某个模板目录的根目录中。

由于开发者仍然需要在上线的站点上看到出错消息，这样的出错信息会向 `ADMINS` 设定选项自动发送email。

在Apache和mod_python下开发的人员，还要保证在Apache的配置文件中关闭 `PythonDebug Off` 选项，这个会在Django被加载以前去除出错消息。

安全领域的总结

我们希望关于安全问题的讨论，不会太让你感到恐慌。Web是一个处处布满陷阱的世界，但是只要有一些远见，你就能拥有安全的站点。

永远记住，Web安全是一个不断发展的领域。如果你正在阅读这本书的停止维护的那些版本，请阅读最新版本的这个部分来检查最新发现的漏洞。事实上，每周或者每月花点时间挖掘web应用安全，并且跟上最新的动态是一个很好的主意。小小的投入，却能收获保护你的站点和用户的无价的回报。

接下来？

下一章中，我们会谈论到一些使用Django的细节问题：如何部署一个站点，并具有良好的伸缩性。

第二十章：部署Django

在这本书中，我们提到了驱使Django发展的很多目标。易用，对初学者友好，重复任务的抽象， 这些都驱使着Django继续发展。

然而，从Django一开始，就有另一个重要的目标：Django应该容易被部署，并且它应该能够用 有限的资源提供大量的服务。

这样的动机是很明显的，当你看到 Django 的背景：堪萨斯州一个小巧的、家族式报纸企业 负担不起高品质的服务器硬件，所以 Django 的最初开发者们都非常的关心如何才能从有限 的资源中挤压出最好的性能。确实，这些年来 Django 的开发者们充当了他们自己的系统管 理员。虽然他们的站点每天处理上千万的点击量，但他们确实没有那么多数量的硬件以至于*需要*专门的系统管理员。

当 Django 成为一个开源项目后，关注在其性能和开发的简易性因为某些原因变得特别重要： 业余爱好者也有同样的需求。有一些人想要花费仅仅 10 美元并使用 Django 来体验制作一个 中小规模流量的站点。

但是小规模的应用只是目标的一半而已。Django 也需要能够增加其规模来满足大型公司和集团 的需求。这里，Django 采取了类似于 LAMP 形式的 Web 集的哲学理论，通常称为 *无共享(shared nothing)* 。

什么是 LAMP?

LAMP 这个缩写最初是创造它来描述一系列驱动 Web 站点的流行的开源软件：

- Linux（操作系统）
- Apache（Web 服务器）
- MySQL（数据库）
- PHP（编程语言）

随着时间的推移，这个缩写已经变得涉及了更多开源软件栈的哲学思想，而不仅仅再是局限于特定的某一种了。所以当 Django 使用 Python 并可以使用多种数据库产品时，LAMP 软件栈证实的一些理论就渗透到 Django 中了。

这里尝试建立一个类似的缩写来描述 Django 的技术栈。本书的作者喜欢用 LAPD（Linux, Apache, PostgreSQL, and Django）或PAID（PostgreSQL, Apache, Internet, and Django）。 使用 Django 并采用 PAID 吧！

无共享

无共享哲学的核心实际上就是应用程序在整个软件栈上的松耦合思想。这个架构的产生直接响应了当时的主流架构背景：将web应用服务器作为不可分的整体，它将语言、数据库、web服务器甚至操作系统的一部分封装到单个进程中(如java)。

当需要伸缩性时，就会碰到下面这个主要问题：几乎不可能把混为一体的进程所干的事情分解到许多不同的物理机器上，因此这类应用就必须要有极为强大的服务器来支撑。这些服务器，需要花费数万甚至数十万美金，从而使这类大规模Web网站远离了财政不宽裕的个人和小公司。

LAMP社区注意到，如果将Web栈分解为多个独立的组件，人们就能从容的从廉价的服务器开始自己的事业，而在发展壮大时只需要添加更多的廉价服务器。如果3000美元的数据库服务器不足以处理负载，只需要简单的购买第二个(或第三、第四个)直到足够。如果需要更多的存储空间，只需增加新的NFS服务器。

但是，为了使这个成为可能，Web应用必须不再假设由同一个服务器处理所有的请求，甚至处理单个请求的所有部分。在大规模LAMP(以及Django)的部署环境中，处理一个页面甚至会涉及到多达半打的服务器！这一条件会对各方面产生诸多影响，但可以归结到以下几点：

- *不能在本地保存状态* 。也就是说，任何需要在多个请求间共享的数据都必须保存在某种形式的持久性存储(如数据库)或集中化缓存中。

- *软件不能假设资源是本地的*。例如，Web平台不能假设数据库运行于同一个服务器上；因此，它必须能够连接到远程数据库服务器。

- *Web栈中的每个部分都必须易于移动或复制*。如果在部署时因为某种原因Apache不能工作，必须能够在最小的代价下切换到其它服务器。或者，在硬件层次，如果Web服务器崩溃，必须能够在最小的宕机时间内替换到另一台机器。请记住，整个哲学基于将应用部署在廉价的、商品化的硬件上。因此，本就应当预见到单个机器的失效。

就所期望的那样，Django或多或少透明的处理好了这件事情，没有一个部分违反这些原则。但是，了解架构设计背后的哲学，在我们处理伸缩性时将会有所裨益。

但这果真解决问题？

这一哲学可能在理论上(或者在屏幕上)看起来不错，但是否真的能解决问题？

好了，我们不直接回答这个问题，先请看一下将业务建立在上述架构上的公司的不完全列表。大家可能已经熟悉其中的一些名字：

- Amazon
- Blogger
- Craigslist
- Facebook
- Google
- LiveJournal
- Slashdot
- Wikipedia
- Yahoo
- YouTube

请允许我用 *当哈利遇见沙莉* 中的著名场景来比喻：他们有的我们也会有！

关于个人偏好的备注

在进入细节之前，短暂跑题一下。

开源运动因其所谓的宗教战争而闻名；许多墨水(以及墨盒、硒鼓等)被挥洒在各类争论上：文本编辑器(`emacs` vs. `vi`), 操作系统(Linux vs.Windows vs. Mac OS),数据库引擎(MySQL vs. PostgreSQL), 当然还包括编程语言。

我们希望远离这些战争。仅仅因为没有足够的时间。

但是，在部署Django确实有很多选择，而且我们也经常被问及个人偏好。表述这些会使社区处于引发上类战争的危险边缘，所以我们在一直以来非常克制。但是，出于完整性和全无保留的目的，我们将在这里表述自己的偏好。以下是我们的优先选择：

- 操作系统: Linux(具体而言是Ubuntu)
- Web服务器: Apache和mod_python
- 数据库服务器：PostgreSQL

当然，我们也看到，许多做了不同选择的Django用户同样也大获成功。

用Apache和mod_python来部署Django

目前，Apache和mod_python是在生产服务器上部署Django的最健壮搭配。

mod_python (http://www.djangoproject.com/r/mod_python/)是一个在Apache中嵌入Python的Apache插件，它在服务器启动时将Python代码加载到内存中。(译注：这里的内存是指虚拟内存) 代码在Apache进程的整个生命周期中都驻留在内存中，与其它服务器的做法相比，这将带来重要的性能提升。

Django要求Apache2.x和mod_python3.x，并且我们优先考虑Apache的prefork MPM模式，而不是worker MPM。

备注

如何配置Apache超出了本书的范围，因此下面将只简单介绍必要的细节。幸运的是，如果需要进一步学习Apache的相关知识，可以找到相当多的绝佳资源。下面是我们所中意的部分资料：

- 开源的Apache在线文档，位于 http://www.djangoproject.com/r/apache/docs/
- *Pro Apache，第三版* (Apress, 2004),作者Peter Wainwright, 位于 http://www.djangoproject.com/r/books/pro-apache/
- *Apache: The Definitive Guide, 第三版* (OReilly, 2002),作者Ben Laurie和Peter Laurie, 位于 http://www.djangoproject.com/r/books/apache-pra/

基本配置

为了配置基于 mod_python 的 Django，首先要安装有可用的 mod_python 模块的 Apache。这通常意味着应该有一个 `LoadModule` 指令在 Apache 配置文件中。它看起来就像是这样：

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

然后，编辑你的Apache的配置文件，添加如下内容：

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

要确保把 `DJANGO_SETTINGS_MODULE` 中的 `mysite.settings` 项目换成与你的站点相应的内容。

它告诉 Apache，任何在 / 这个路径之后的 URL 都使用 Django 的 mod_python 来处理。它 将 `DJANGO_SETTINGS_MODULE` 的值传递过去，使得 mod_python 知道这时应该使用哪个配置。

注意这里使用 `<Location>` 指令而不是 `<Directory>` 。后者用于指向你的文件系统中的一个位置，然而 `<Location>` 指向一个 Web 站点的 URL 位置。

Apache 可能不但会运行在你正常登录的环境中，也会运行在其它不同的用户环境中；也可能会有不同的文件路径或 sys.path。你需要告诉 mod_python 如何去寻找你的项目及 Django 的位置。

```
PythonPath "['/path/to/project', '/path/to/django'] + sys.path"
```

你也可以加入一些其它指令，比如 `PythonAutoReload Off` 以提升性能。查看 mod_python 文档获得详细的指令列表。

注意，你应该在成品服务器上设置 `PythonDebug Off` 。如果你使用 `PythonDebug On` 的话，在程序产生错误时，你的用户会看到难看的（并且是暴露的） Python 回溯信息。

重启 Apache 之后所有对你的站点的请求（或者是当你用了 `<VirtualHost>` 指令后则是虚拟主机）都会由 Django 来处理。

注意

如果你在一个比 / 位置更深的子目录中部署 Django，它 *不会* 对你的 URL 进行修整。所以如果你的 Apache 配置是像这样的：

```
<Location "/mysite/">
```

```
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

则你的 *所有* URL都要以 `"/mysite/"` 起始。基于这种原因我们通常建议将 Django，部署 在主机或虚拟主机的根目录。另外还有一个选择就是，你可以简单的将你的 URL 配置转换成像下面这样：

```
urlpatterns = patterns('',
    (r'^mysite/', include('normal.root.urls')),
)
```

在同一个 Apache 的实例中运行多个 Django 程序

在同一个 Apache 实例中运行多个 Django 程序是完全可能的。当你是一个独立的 Web 开发人员并有多个不同的客户时，你可能会想这么做。

只要像下面这样使用 `VirtualHost` 你可以实现：

```
NameVirtualHost *

<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

如果你需要在同一个 `VirtualHost` 中运行两个 Django 程序，你需要特别留意一下以 确保 mod_python 的代码缓存不被弄得乱七八糟。使用 `PythonInterpreter` 指令来将不 同的 `<Location>` 指令分别解释：

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter mysite_other
    </Location>
</VirtualHost>
```

这个 `PythonInterpreter` 中的值不重要，只要它们在两个 `Location` 块中不同。

用 mod_python 运行一个开发服务器

因为 mod_python 缓存预载入了 Python 的代码，当在 mod_python 上发布 Django 站点时，你每 改动了一次代码都需要重启 Apache 一次。这还真是件麻烦事，所以这有个办法来避免它：只要 加入 `MaxRequestsPerChild 1` 到配置文件中强制 Apache 在每个请求时都重新载入所有的 代码。但是不要在产品服务器上使用这个指令，这会撤销 Django 的特权。

如果你是一个用分散的 `print` 语句（我们就是这样）来调试的程序员，注意这 `print` 语 句在 mod_python 中是无效的；它不会像你希望的那样产生一个 Apache 日志。如果你需要在 mod_python 中打印调试信息，可能需要用到 Python 标准日志包（Pythons standard logging package）。 更多的信息请参见 http://docs.python.org/lib/module-logging.html 。另一个选择是在模板页面中加入调试信息。

使用相同的Apache实例来服务Django和Media文件

Django本身不用来服务media文件；应该把这项工作留给你选择的网络服务器。我们推荐使用一个单独的网络服务器（即没有运行Django的一个）来服务media。想了解更多信息，看下面的章节。

不过，如果你没有其他选择，所以只能在同Django一样的Apache `VirtualHost` 上服务media文件，这里你可以针对这个站点的特定部分关闭mod_python：

```
<Location "/media/">
    SetHandler None
</Location>
```

将 `Location` 改成你的media文件所处的根目录。

你也可以使用 `<LocationMatch>` 来匹配正则表达式。比如，下面的写法将Django定义到网站的根目录，并且显式地将 `media` 子目录以及任何以 `.jpg`， `.gif`， 或者 `.png` 结尾的URL屏蔽掉：

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media/">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

在所有这些例子中，你必须设置 `DocumentRoot`，这样apache才能知道你存放静态文件的位置。

错误处理

当你使用 Apache/mod_python 时，错误会被 Django 捕捉，它们不会传播到 Apache 那里，也不会出现在 Apache 的 `错误日志` 中。

有一个例外就是当确实你的 Django 设置混乱了时。在这种情况下，你会在浏览器上看到一个 内部服务器错误的页面，并在 Apache 的 `错误日志` 中看到 Python 的完整回溯信息。 `错误日志` 的回溯信息有多行。当然，这些信息是难看且难以阅读的。

处理段错误

有时候，Apache会在你安装Django的时候发生段错误。这时，基本上 *总是* 有以下两个与Django本身无关的原因其中之一所造成：

- 有可能是因为，你使用了 `pyexpat` 模块（进行XML解析）并且与Apache内置的版本相冲突。详情请见 http://www.djangoproject.com/r/articles/expat-apache-crash/.
- 也有可能是在同一个Apache进程中，同时使用了mod_python 和 mod_php，而且都使用MySQL作为数据库后端。在有些情况下，这会造成PHP和Python的MySQL模块的版本冲突。在mod_python的FAQ中有更详细的解释。 http://www.djangoproject.com/r/articles/php-modpython-faq/.

如果还有安装mod_python的问题，有一个好的建议，就是先只运行mod_python站点，而不使用Django框架。这是区分mod_python特定问题的好方法。下面的这篇文章给出了更详细的解释。 http://www.djangoproject.com/r/articles/getting-modpython-working/.

下一个步骤应该是编辑一段测试代码，把你所有django相关代码import进去，你的views,models,URLconf,RSS配置，等等。把这些imports放进你的handler函数中，然后从浏览器进入你的URL。如果这些导致了crash，你就可以确定是import的django代码引起了问题。逐个去掉这些imports，直到不再冲突，这样就能找到引起问题的那个模块。 深入了解各模块，看看它们的imports。要想获得更多帮助，像linux的ldconfig，Mac OS的otool和windows的ListDLLs（form sysInternals）都可以帮你识别共享依赖和可能的版本冲突。

使用FastCGI部署Django应用

尽管将使用Apache和mod_python搭建Django环境是最具鲁棒性的，但在很多虚拟主机平台上，往往只能使用FastCGI

此外，在很多情况下，FastCGI能够提供比mod_python更为优越的安全性和效能。针对小型站点，相对于Apache来说FastCGI更为轻量级。

FastCGI 简介

如何能够由一个外部的应用程序很有效解释WEB 服务器上的动态页面请求呢？答案就是使用FastCGI! 它的工作步骤简单的描述起来是这样的： 1、WEB服务器收到客户端的页面请求 2、WEB服务器将这个页面请求委派给一个FastCGI 外部进程（WEB服务器于FastCGI之间是通过socket来连接通讯的） 3、FastCGI外部进程得到WEB服务器委派过来的页面请求信息后进行处理，并且将处理结果（动态页面内容）返回给WEB服务器 4、Web服务器将FastCGI返回回来的结果再转送给客户端浏览器。

和mod_python一样，FastCGI也是驻留在内存里为客户请求返回动态信息,而且也免掉了像传统的CGI一样启动进程时候的时间花销。但于mod_python不同之处是它并不是作为模块运行在web服务器同一进程内的，而是有自己的独立进程。

为什么要在一个独立的进程中运行代码？

在以传统的方式的几种以mod_*方式嵌入到Apache的脚本语言中（常见的例如：PHP，Python/mod_python和Perl/mod_perl），他们都是以apache扩展模块的方式将自身嵌入到Apache进程中的。尽管这种方式可以减低启动时候的时间花销（因为代码不用在每次收到访问请求的时候都读去硬盘数据），但这是以增大内存的开销来作为代价的。

每一个Apache进程都是一个Apache引擎的副本，它完全包括了所有Apache所具有的一切功能特性（哪怕是对Django毫无好处的东西也一并加载进来）。而FastCGI就不一样了，它仅仅把Python和Django等必备的东东弄到内存中。

依据FastCGI自身的特点可以看到，FastCGI进程可以与Web服务器的进程分别运行在不同的用户权限下。对于一个多人共用的系统来说，这个特性对于安全性是非常有好处的，因为你可以安全的于别人分享和重用代码了。

如果你希望你的Django以FastCGI的方式运行，那么你还必须安装 `flup` 这个Python库，这个库就是用于处理FastCGI的。很多用户都抱怨 `flup` 的发布版太久了，老是不更新。其实不是的，他们一直在努力的工作着，这是没有放出来而已。但你可以通过 http://www.djangoproject.com/r/flup/ 获取他们的最新的SVN版本。

运行你的 FastCGI 服务器

FastCGI是以客户机/服务器方式运行的，并且在很多情况下，你得自己去启动FastCGI的服务进程。 Web服务器（例如Apache,lighttpd等等）仅仅在有动态页面访问请求的时候才会去与你的Django-FastCGI进程交互。因为Fast-CGI已经一直驻留在内存里面了的，所以它响应起来也是很快的。

注意

在虚拟主机上使用的话，你可能会被强制的使用Web server-managed FastCGI进程。在这样的情况下，请参阅下面的“在Apache共享主机里运行Django”这一小节。

web服务器有两种方式于FastCGI进程交互：使用Unix domain socket(在win32里面是 *命名管道*)或者使用TCP socket.具体使用哪一个，那就根据你的偏好而定了，但是TCP socket弄不好的话往往会发生一些权限上的问题。

开始你的服务器项目，首先进入你的项目目录下（你的 `manage.py` 文件所在之处），然后使用 `manage.py runfcgi` 命令：

```
./manage.py runfcgi [options]
```

想了解如何使用 `runfcgi` ，输入 `manage.py runfcgi help` 命令。

你可以指定 `socket` 或者同时指定 `host` 和 `port` 。当你要创建Web服务器时，你只需要将服务器指向当你在启动FastCGI服务器时确定的socket或者host/port。

范例：

在TCP端口上运行一个线程服务器：

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

在Unix socket上运行prefork服务器：

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
```

启动，但不作为后台进程（在调试时比较方便）：

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock
```

停止FastCGI的行程

如果你的FastCGI是在前台运行的，那么只需按Ctrl+C就可以很方便的停止这个进程了。但如果是在后台运行的话，你就要使用Unix的 `kill` 命令来杀掉它。

如果你在 `manage.py runfcgi` 中指定了 `pidfile` 这个选项，那么你可以这样来杀死这个FastCGI后台进程：

```
kill `cat $PIDFILE`
```

`$PIDFILE` 就是你在 `pidfile` 指定的那个。

你可以使用下面这个脚本方便地重启Unix里的FastCGI守护进程：

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
  PYTHONPATH="../python:.." \
  ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

在Apache中以FastCGI的方式使用Django

在Apache和FastCGI上使用Django，你需要安装和配置Apache，并且安装mod_fastcgi。请参见Apache和mod_fastcgi文档：http://www.djangoproject.com/r/mod_fastcgi/ 。

当完成了安装，通过 `httpd.conf` （Apache的配置文件）来让Apache和Django FastCGI互相通信。你需要做两件事：

- 使用 `FastCGIExternalServer` 指明FastCGI的位置。
- 使用 `mod_rewrite` 为FastCGI指定合适的URL。

指定 FastCGI Server 的位置

`FastCGIExternalServer` 告诉Apache如何找到FastCGI服务器。 按照FastCGIExternalServer 文档（http://www.djangoproject.com/r/mod_fastcgi/FastCGIExternalServer/ ），你可以指明 `socket` 或者 `host` 。以下是两个例子：

```
# Connect to FastCGI via a socket/named pipe:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

在这两个例子中， /home/user/public_html/ 目录必须存在，而 `/home/user/public_html/mysite.fcgi` 文件不一定存在。它仅仅是一个Web服务器内部使用的接口，这个URL决定了对于哪些URL的请求会被FastCGI处理（下一部分详细讨论）。

使用mod_rewrite为FastCGI指定URL

第二步是告诉Apache为符合一定模式的URL使用FastCGI。为了实现这一点，请使用mod_rewrite 模块，并将这些URL重定向到 `mysite.fcgi` （或者正如在前文中描述的那样，使用任何在 `FastCGIExternalServer` 指定的内容）。

在这个例子里面，我们告诉Apache使用FastCGI来处理那些在文件系统上不提供文件(译者注：也就是指向虚拟文件)和没有从 `/media/` 开始的任何请求。如果你使用Django的admin站点，下面可能是一个最普通的例子：

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

FastCGI 和 lighttpd

lighttpd (http://www.djangoproject.com/r/lighttpd/) 是一个轻量级的Web服务器，通常被用来提供静态页面的访问。它天生支持FastCGI，因此除非你的站点需要一些Apache特有的特性，否则，lighttpd对于静态和动态页面来说都是理想的选择。

确保 `mod_fastcgi` 在模块列表中，它需要出现在 `mod_rewrite` 和 `mod_access` ，但是要在 `mod_accesslog` 之前。你也可能需要 `mod_alias` 模块，来提供管理所用的媒体资源。

将下面的内容添加到你的lighttpd的配置文件中：

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media/" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

在一个lighttpd进程中运行多个Django站点

lighttpd允许你使用条件配置来为每个站点分别提供设置。为了支持FastCGI的多站点，只需要在FastCGI的配置文件中，为每个站点分别建立条件配置项：

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
       ...
    )
```

```
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
       ...
    )
    ...
}
```

你也可以通过 `fastcgi.server` 中指定多个入口，在同一个站点上实现多个Django安装。请为每一个安装指定一个FastCGI主机。

在使用Apache的共享主机服务商处运行Django

许多共享主机的服务提供商不允许运行你自己的服务进程，也不允许修改 `httpd.conf` 文件。尽管如此，仍然有可能通过Web服务器产生的子进程来运行Django。

备注

如果你要使用服务器的子进程，你没有必要自己去启动FastCGI服务器。Apache会自动产生一些子进程，产生的数量按照需求和配置会有所不同。

在你的Web根目录下，将下面的内容增加到 `.htaccess` 文件中：

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

接着，创建一个脚本，告知Apache如何运行你的FastCGI程序。创建一个 `mysite.fcgi` 文件，并把它放在你的Web目录中，打开可执行权限。

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

重启新产生的进程服务器

如果你改变了站点上任何的python代码，你需要告知FastCGI。但是，这不需要重启Apache，而只需要重新上传 `mysite.fcgi` 或者编辑改文件，使得修改时间发生了变化，它会自动帮你重启Django应用。

如果你拥有Unix系统命令行的可执行权限，只需要简单地使用 `touch` 命令：

```
touch mysite.fcgi
```

可扩展性

既然你已经知道如何在一台服务器上运行Django，让我们来研究一下，如何扩展我们的Django安装。这一部分我们将讨论，如何把一台服务器扩展为一个大规模的服务器集群，这样就能满足每小时上百万的点击率。

有一点很重要，每一个大型的站点大的形式和规模不同，因此可扩展性其实并不是一种千篇一律的行为。以下部分会涉及到一些通用的原则，并且会指出一些不同选择。

首先，我们来做一个大的假设，只集中地讨论在Apache和mod_python下的可扩展性问题。尽管我们也知道一些成功的中型和大型的FastCGI策略，但是我们更加熟悉Apache。

运行在一台单机服务器上

大多数的站点一开始都运行在单机服务器上，看起来像图20-1这样的构架。

图 20-1：一个单服务器的Django安装。

这对于小型和中型的站点来说还不错，并且也很便宜，一般来说，你可以在3000美元以下就搞定一切。

然而，当流量增加的时候，你会迅速陷入不同软件的 *资源争夺* 之中。数据库服务器和Web服务器都 *喜欢* 自己拥有整个服务器资源，因此当被安装在单机上时，它们总会争夺相同的资源（RAM, CPU），它们更愿意独享资源。

通过把数据库服务器搬移到第二台主机上，可以很容易地解决这个问题。这将在下一部分介绍。

分离出数据库服务器

对于Django来说，把数据库服务器分离开来很容易：只需要简单地修改 `DATABASE_HOST`，设置为新的数据库服务器的IP地址或者DNS域名。设置为IP地址总是一个好主意，因为使用DNS域名，还要牵涉到DNS服务器的可靠性连接问题。

使用了一个独立的数据库服务器以后，我们的构架变成了图20-2。

图 20-2：将数据库移到单独的服务器上。

这里，我们开始步入 *n-tier* 构架。不要被这个词所吓坏，它只是说明了Web栈的不同部分，被分离到了不同的物理机器上。

我们再来看，如果发现需要不止一台的数据库服务器，考虑使用连接池和数据库备份(冗余）将是一个好主意。不幸的是，本书没有足够的时间来讨论这个问题，所以你参考数据库文档或者向社区求助。

运行一个独立的媒体服务器

使用单机服务器仍然留下了一个大问题：处理动态内容的媒体资源，也是在同一台机器上完成的。

这两个活动是在不同的条件下进行的，因此把它们强行凑和在同一台机器上，你不可能获得很好的性能。下一步，我们要把媒体资源（任何 *不是* 由Django视图产生的东西）分离到别的服务器上（请看图20-3）。

图 20-3：分离出媒体服务器。

理想的情况是，这个媒体服务器是一个定制的Web服务器，为传送静态媒体资源做了优化。lighttpd和tux (http://www.djangoproject.com/r/tux/) 都是极佳的选择，当然瘦身的Apache服务器也可以工作的很好。

对于拥有大量静态内容（照片、视频等）的站点来说，将媒体服务器分离出去显然有着更加重要的意义，而且应该是扩大规模的时候所要采取的 *第一步措施* 。

这一步需要一点点技巧，Django的admin管理接口需要能够获得足够的权限来处理上传的媒体（通过设置 `MEDIA_ROOT` ）。如果媒体资源在另外的一台服务器上，你需要获得通过网络写操作的权限。

最简单的方案是使用NFS（网络文件系统）把媒体服务器的目录挂载到Web服务器上来。只要 `MEDIA_ROOT` 设置正确，

媒体的上传就可以正常工作。

实现负担均衡和数据冗余备份

现在，我们已经尽可能地进行了分解。这种三台服务器的构架可以承受很大的流量，比如每天1000万的点击率。如果还需要进一步地增加，你就需要开始增加冗余备份了。

这是个好主意。请看图 20-3，一旦三个服务器中的任何一个发生了故障，你就得关闭整个站点。因此在引入冗余备份的时候，你并不只是增加了容量，同时也增加了可靠性。

我们首先来考虑Web服务器的点击量。把同一个Django的站点复制多份，在多台机器上同时运行很容易，我们也只需要同时运行多台机器上的Apache服务器。

你还需要另一个软件来帮助你在多台服务器之间均衡网络流量：*流量均衡器（load balancer）*。你可以购买昂贵的专有的硬件均衡器，当然也有一些高质量的开源的软件均衡器可供选择。

Apaches 的 `mod_proxy` 是一个可以考虑的选择，但另一个配置更棒的选择是：Perlbal (http://www.djangoproject.com/r/perlbal/)。Perlbal是一个均衡器，同时也是一个反向代理，它的开发者和memcached的开发者是同一拨人（请见13章）。

备注

如果你使用FastCGI，你同样可以分离前台的web服务器，并在多台其他机器上运行FastCGI服务器来实现相同的负载均衡的功能。前台的服务器就相当于是一个均衡器，而后台的FastCGI服务进程代替了Apache/mod_python/Django服务器。

现在我们拥有了服务器集群，我们的构架慢慢演化，越来越复杂，如图20-4。

图 20-4：负载均衡的服务器设置。

值得一提的是，在图中，Web服务器指的是一个集群，来表示许多数量的服务器。一旦你拥有了一个前台的均衡器，你就可以很方便地增加和删除后台的Web服务器，而且不会造成任何网站不可用的时间。

慢慢变大

下面的这些步骤都是上面最后一个的变体：

- 当你需要更好的数据库性能，你可能需要增加数据库的冗余服务器。MySQL内置了备份功能；PostgreSQL应该看一下Slony (http://www.djangoproject.com/r/slony/) 和 pgpool (http://www.djangoproject.com/r/pgpool/)，这两个分别是数据库备份和连接池的工具。
- 如果单个均衡器不能达到要求，你可以增加更多的均衡器，并且使用轮训（round-robin）DNS来实现分布访问。
- 如果单台媒体服务器不够用，你可以增加更多的媒体服务器，并通过集群来分布流量。
- 如果你需要更多的高速缓存（cache），你可以增加cache服务器。
- 在任何情况下，只要集群工作性能不好，你都可以往上增加服务器。

重复了几次以后，一个大规模的构架会像图20-5。

图 20-5。大规模的Django安装。

尽管我们只是在每一层上展示了两到三台服务器，你可以在上面随意地增加更多。

当你到了这一个阶段，你有一些选择。附录A有一些开发者关于大型系统的信息。如果你想要构建一个高流量的Django站点，那值得一读。

性能优化

如果你有大笔大笔的钱，遇到扩展性问题时，你可以简单地投资硬件。对于剩下的人来说，性能优化就是必须要做的一件事。

备注

顺便提一句，谁要是有大笔大笔的钞票，请捐助一点Django项目。我们也接受未切割的钻石和金币。

不幸的是，性能优化比起科学来说更像是一种艺术，并且这比扩展性更难描述。如果你真想要构建一个大规模的Django应用，你需要花大量的时间和精力学习如何优化构架中的每一部分。

以下部分总结了多年以来的经验，是一些专属于Django的优化技巧。

RAM怎么也不嫌多

写这篇文章的时候，RAM的价格已经降到了每G大约200美元。购买尽可能多的RAM，再在别的上面投资一点点。

高速的处理器并不会大幅度地提高性能；大多数的Web服务器90%的时间都浪费在了硬盘IO上。当硬盘上的数据开始交换，性能就急剧下降。更快速的硬盘可以改善这个问题，但是比起RAM来说，那太贵了。

如果你拥有多台服务器，首要的是要在数据库服务器上增加内存。如果你能负担得起，把你整个数据库都放入到内存中。这不会很难。LJWorld.com等网站的数据库保存了大量从1989年起至今的报纸和文章，内存的消耗也不到2G。

下一步，最大化Web服务器上的内存。最理想的情况是，没有一台服务器进行磁盘交换。如果你达到了这个水平，你就能应付大多数正常的流量。

禁用 Keep-Alive

`Keep-Alive` 是HTTP提供的功能之一，它的目的是允许多个HTTP请求复用一个TCP连接，也就是允许在同一个TCP连接上发起多个HTTP请求，这样有效的避免了每个HTTP请求都重新建立自己的TCP连接的开销。

这一眼看上去是好事，但它足以杀死Django站点的性能。如果你从单独的媒体服务器上向用户提供服务，每个光顾你站点的用户都大约10秒钟左右发出一次请求。这就使得HTTP服务器一直在等待下一次keep-alive 的请求，空闲的HTTP服务器和工作时消耗一样多的内存。

使用 memcached

尽管Django支持多种不同的cache后台机制，没有一种的性能可以 *接近* memcached。如果你有一个高流量的站点，不要犹豫，直接选择memcached。

经常使用memcached

当然，选择了memcached而不去使用它，你不会从中获得任何性能上的提升。第13章将为你提供有用的信息：学习如何使用Django的cache框架，并且尽可能地使用它。大量的可抢占式的高速缓存通常是一个站点在大流量下正常工作的唯一瓶颈。

参加讨论

Django相关的每一个部分，从Linux到Apache到PostgreSQL或者MySQL背后，都有一个非常棒的社区支持。如果你真想从你的服务器上榨干最后1%的性能，加入开源社区寻求帮助。多数的自由软件社区成员都会很乐意地提供帮助。

别忘了Django社区。这本书谦逊的作者只是Django开发团队中的两位成员。我们的社区有大量的经验可以提供。

下一步？

你已经看到了正文的结束部分了。下面的这些附录都包含了许多参考资料，当你构建你的Django项目时，有可能会用到。

我们希望你的Django站点运行良好，无论你的站点是你和你朋友之间的一个小玩具，还是下一个Google。