Django

第一章：介绍 *Django*

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。 本书的代码清晰，易维护，通过学习，可以快速开发功能强大的网站。

框架是什麼*?*

像这样的一次性的动态页面，从头写起的方法并非一定不好。 其中一点： 这些代码简单易懂，就算是一个初起步的 开发者都能读明白这16行的Python的代码，而且这些代码从头到尾做了什么都能了解得一清二楚。 不需要学习额外 的背景知识，没有额外的代码需要去了解。 同样，也易于部署这16行代码，只需要将它保存为一个 `latestbooks.cgi` 的 文件，上传到网络服务器上，通过浏览器访问即可。

尽管实现很简单，还是暴露了一些问题和不便的地方。 问你自己这几个问题：

- 应用中有多处需要连接数据库会怎样 呢？ 每个独立的cgi脚本，不应该重复写数据库连接的代码。 比较实用的办法是写一个共享函数，可被多个代码调用。
- 一个开发人员 确实 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？
- 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？ 一个字符写错了，可能导致整个应用崩溃 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： 首先，我们分成4个python的文件，(`models.py` , `views.py` , `urls.py`) 和html模板文件 (`latest_books.html`)

`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。

- views.py文件包含了页面的业 务逻辑。 `latest_books()`函 数叫做视图。

- `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。 换句话说，如果你的域名是 example.com，任何人浏览网址 http://example.com/latest/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。 使用带基本逻辑声明的模板语言，如`{% for book in book_list %}`

jango 历史

在我们讨论代码之前我们需要先了解一下 Django 的历史。 从上面我们注意到：我们将向你展示如何不使用捷径来完成工作，以便能更好的理解捷径的原理 同样，理解 django 产生的背景，历史有助于理解 django 的实现方式。

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。

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。

第二章:入门

安装

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： 在命令行窗口中输入 `python` （或是在 OS X 的程序/工具/终端中）。 如果你看到这样的信息,说明 python 已经安装好了.

```
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.
>>>
```

否则, 你需要下载并安装 Python. 它既快速又方便，而详细说明可参考 http://www.python.org/download/

安装 *Django*

我们推荐选定一个正式发布版本，但重要的是了解到主干开发版本的存在，因为在文档和社区成员中你会发现它被提到。

安装官方发布版

官方发布的版本带有一个版本号，例如 1.0.3 或 1.1，而最新的一个总是可在 http://www.djangoproject.com/download/找 到。

如果您在用 Linux 系统，其中包括 Django 的包，使用默认的版本是个好主意。 这样，你将会通过系统的包管理得到安全的升级

如果你的系统没有自带 Django，你可以自己下载然后安装框架 首先，下载名字类似于 `Django-1.0.2-final.tar.gz` 压 缩文件。（下载到哪里无所谓，安装程序会把 Django 文件放到正确的地方。）解压缩之后运行 `setup.py install`，像操作大多 数 Python 库一样。

以下是如何在 Unix 系统上安装的方法：

1. `tar xzvf Django-*.tar.gz` 。
2. `cd Django-*` 。
3. `sudo python setup.py install` 。

安装 **Trunk** 版本

最新最好的 django 的开发版本称为 trunk,可以从 django 的 subversion 处获得。 如果你想尝鲜，或者想为 django 贡献代码，那么你应当安装这个版本。

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。

请记住，即使是使用 trunk 版本，也是有保障的。 因为很多 django 的开发者在正式网站上就是用的 trunk 版本，他们会保证 trunk 版本的稳定性。

测试 ***Django*** 安装

让我们花点时间去测试 django 是否安装成功，并工作良好。同时也可以了解到一些明确的安装后的反馈信息。 在 Shell 中，更换到另外一个目录（不是包含 Django 的目录），然后输入 `python` 来打开 Python 的交互解释器。如果安装成功，你应该可以导入 `django` 模块了：

```
>>> import django
>>> django.VERSION
(1, 1, 0, final', 1)
```

交互解释器示例

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。

我们在交互解释器中演示 Python 示例将贯穿整本书。 你可以以三个大于号 (>>>)来分辨出示例，三个大于号就表示交互提示符 如果你要从本书中拷贝示例，请不要拷贝提示符。

安装数据库

这会儿，你可以使用 django 写 web 应用了，因为 django 只要求 python 正确安装后就可以 跑起来了。 不过，当你想开发一个数据库驱动的 web 站点时，你应当需要配置一个数据库服务器。

如果你只想玩一下，可以不配置数据库，直接跳到 开始一个 project 部分去，不过你要注意本书

的例子都是假设你配置好了一个正常工作的数据库。

Django 支持四种数据库：

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

如果你只是玩一下，不想安装数据库服务，那么可以考虑使用 SQLite。 如果你用 python2.5 或更高版本的话，SQLite 是唯一一个被支持的且不需要以上安装步骤的数据库。 它仅对你的文件系统中的单一文件读写数据，并且 Python2.5 和以后版本内建了对它的支持。

开始一个项目

一但你安装好了 python，django 和（可选的）数据库及相关库，你就可以通过创建一个 *project*， 迈出开发 django 应用的第一步。

转到你创建的目录，运行命令 **`django-admin.py startproject mysite`**。 这样会在你的当前目录下创建一个目录。**`Mysite`**

如果在运行时，你看到权限拒绝的提示，你应当修改这个文件的权限。`django-admin.py startproject` 为此， 键入 `cd /usr/local/bin` 转到 `django-admin.py` 所在的目录，运行命令 `chmod +x django-admin.py`

`startproject` 命令创建一个目录，包含 4 个文件：

```
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。 这是一个空文件，一般你不需要修改它。
- `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。 键入 `python manage.py help`，看一下它能做什么。 你应当不需要编辑这个文件；在这个目录下生成它纯是为了方便。
- `settings.py`：该 Django 项目的设置或配置。 查看并理解这个文件中可用的设置类型及其默认值。
- `urls.py`：django 项目的 URL 设置。 可视其为你的 django 网站的目录。 目前，它是空的。

除了尺寸小些，这些文件已经构成了一个可运行的 Django 应用。

运行开发服务器

`django` 开发服务是可用在开发期间的，一个内建的，轻量的 `web` 服务。 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。 开发服务器监测你的代码并自动加载它，这样你会很容易修改

代码而不用重启动服务。

要运行你的服务器，请切换到你的项目目录里 **(cd mysite)**。如果你还没准备好，那么运行下面的命令：python manage.py runserver

你会看到些像这样的

```
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.
```

这将会在端口 8000 启动一个本地服务器, 并且只能从你的这台电脑连接和访问。 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令人赏心悦目的淡蓝色 Django 欢迎页面。 它开始工作了。

默认情况下， runserver 命令在 8000 端口启动开发服务器，且仅监听本地连接。 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
```

通过指定一个 IP 地址，你可以告诉服务器–允许非本地连接访问。 如果你想和其他开发人员共享同一开发站点的话，该功能特别有用。 `` 0.0.0.0`` 这个 IP 地址，告诉服务器去侦听任意的网络接口。

```
python manage.py runserver 0.0.0.0:8000
```

完成这些设置后，你本地网络中的其它计算机就可以在浏览器中访问你的 IP 地址了。比如: http://192.168.1.103:8000/ . (注意，你将需要校阅一下你的网络配置来决定你在本地网络中的 IP 地址) Unix 用户可以在命令提示符中输入 ifconfig 来获取以上信息。 使用 Windows 的用户，请尝试使用 ipconfig 命令。

第三章 视图和 URL 配置

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。 在这一章，你将会学到用 Django 创建动态网页的基本知识。

你的第一个基于 *Django* 的页面： *Hello World*

如果你曾经发布过 Hello world 页面，但是没有使用网页框架，只是简单的在 hello.html 文本文件中输入 Hello World，然后上传到任意的一个网页服务器上。 注意，在这个过程中，你已经说明了两个关于这个网页的关键信息： 它包括（字符串 "Hello world"）和它的 URL(http://www.example.com/hello.html , 如果你把文件放在子目录，也可能是 http://www.example.com/files/hello.html)。

使用 Django，你会用不同的方法来说明这两件事 页面的内容是靠 *view function*（视图函数） 来产生，URL 定义在 *URLconf* 中。首先，我们先写一个 Hello World 视图函数。

第一份视图：

在上一章使用 django-admin.py startproject 制 作的 mysite 文件夹中， 创建一个叫做 views.py 的 空文件。这个 Python 模块将包含这一章的视图。 请留意，Django 对于 view.py 的文件命名没有特别的要求，它不在乎这个文件叫什么。但是根据约定，把它命名成

view.py 是个好主意，这样有 利于其他开发者读懂你的代码，正如你很容易的往下读懂本文。

我们的 Hello world 视图非常简单。 这些是完整的函数和导入声明，你需要输入到 `views.py` 文件：

#!/usr/bin/env python

-*- coding: UTF-8 -*-

#view.py

from django.http import HttpResponse

def hello(request):

  return HttpResponse("Hello world!")

这里主要讲的是： 一个视图就是 Python 的一个函数。这个函数第一个参数的类型是 HttpRequest；它返回一个 HttpResponse 实例。为了使一个 Python 的函数成为一个 Django 可识别的视图，它必须满足这两个条件。 （也有例外，但是我们稍后才会接触到。

你的第一个 URLconf

现在，如果你再运行：python manage.py runserver，你还将看到 Django 的欢迎页面，而看不到我们刚才写的 Hello world 显示页面。 那是因为我们的 mysite 项目还对 hello 视图一无所知。 我们需要通过一个详细描述的 URL 来显式的告诉它并且激活这个视图。 （继续我们刚才类似发布静态 **HTML** 文件的例子。现在我们已经创建了 **HTML** 文件，但还没有把它上传至服务器的目录。）为了绑定视图函数和 **URL**，我们使 用 **URLconf**。

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。 例如，当用户访问/foo/时，调用视图函数 foo_view()，这个视图函数存在于 Python 模块文件 view.py 中。

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）。

默认的 URLconf 包含了一些被注释起来的 Django 中常用的功能，仅仅只需去掉这些注释就可以开 启这些功能. 下面是 URLconf 中忽略被注释的行后的实际内容

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

urlpatterns = patterns('',
)
```

让我们逐行解释一下代码：

- 第一行导入 django.conf.urls.defaults 下的所有模块，它们是 Django URLconf 的基本构造。 这包含了一个 patterns 函数。
- 第二行调用 `patterns()` 函数并将返回结果保存到 `urlpatterns` 变量。patterns 函数当前只有一个参数—一个空的字符串。（这个字符串可以被用来表示一个视图函数的通用前缀。具体我们将在第八章里面介绍。）

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。 （注：那是上一节中 Django 怎么知道显示欢迎页面的原因。 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。

如果想在 URLconf 中加入 URL 和 view，只需增加映射 URL 模式和 view 功能的 Python tuple 即可. 这里演示如何添加 view 中 hello 功能.

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

urlpatterns = patterns('',
    ('^hello/$', hello),
)
```

请留意：为了简洁，我们移除了注释代码。 如果你喜欢的话，你可以保留那些行。）

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 `hello` 视图。 （这假设 mysite/views.py 在你的 Python 搜索路径上。关于搜索路径的解释，请参照下文。）
- 接下来，我们为 urlpatterns 加上一行： (‘^hello/$’, hello), 这行被称作 URLpattern，它是一个 Python 的元组。元组中第一个元素是模式匹配字符串（正则表达式）；第二个元素是那个模式将使用的视图函 数。

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

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。

如果你想看 Python 搜索路径的值，运行 Python 交互解释器，然后输入：

```
>>> import sys
>>> print sys.path
```

你大多数的 URL 模式会以^开始、以$结束，但是拥有复杂匹配的灵活性会更好。

你可能会问：如果有人申请访问/hello（尾部没有斜杠/）会怎样。 因为我们的 URL 模式要求尾部有一个斜杠(/)，那个申请 URL 将不匹配。 然而，默认地，任何不匹配或尾部没有斜杠(/)的申请 URL，将被重定向至尾部包含斜杠的相同字眼的 URL。 （这是受配置文件 setting 中

APPEND_SLASH 项控制的，参见附件 D。）

如果你是喜欢所有 URL 都以'/'结尾的人（Django 开发者的偏爱），那么你只需要在每个 URL 后 添加斜杠，并且设置"APPEND_SLASH"为"True". 如果不喜欢 URL 以斜杠结尾或者根据每个 URL 来决定，那么需要设置"APPEND_SLASH"为"False",并且根据你自己的意愿来添加结尾斜杠 /在 URL 模式后.

另外需要注意的是，我们把 hello 视图函数作为一个对象传递，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？

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/hello/ 。 你就可以看到输出结果了。 开发服务器将自动检测 Python 代码的更改来做必要的重新加载， 所以你不需要重启 Server 在代码更改之后。服务器运行地址`` http://127.0.0.1:8000/`` ，所以打开浏览器直接输入`` http://127.0.0.1:8000/hello/`` ，你将看到由你的 Django 视图输出的 Hello world。

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。 这里是一些基本的语法。

符号	匹配
`.` (dot)	任意单一字符
`\d`	任意一位数字
`[A-Z]`	`A` 到 `Z` 中任意一个字符（大写）
`[a-z]`	`a` 到 `z` 中任意一个字符（小写）
`[A-Za-z]`	`a` 到 `z` 中任意一个字符（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)
`[^/]+`	一个或多个不为'/'的字符
`*`	零个或一个之前的表达式（例如：`\d?` 匹配零个或一个数字）
`*`	匹配 0 个或更多 (例如, `\d*` 匹配 0 个 或更多数字字符)
`{1,3}`	介于一个和三个（包含）之前的表达式（例如，`\d{1,3}`匹 配一个或两个或三个数字）

关于"404 错误"的快速参考

目前，我们的 URLconf 只定义了一个单独的 URL 模式： 处理 URL `/hello/` 。 当请求其他 URL 会怎么样呢？

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/goodbye/` 或者 `http://127.0.0.1:8000/hello/subdirectory/` ，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）。 因为你的 URL 申请在 URLconf 中没有定义，所以 Django 显示这条信息。

这个页面比原始的 **404** 错误信息更加实用。 它同时精确的告诉你 **Django** 调用哪个 **URLconf** 及其包含的每个模式。 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。

关于网站根目录的快速参考。

在最后一节，如果你想通过 http://127.0.0.1:8000/看网站根目录你将看到一个 404 错误消息。Django 不会增加任何东西在网站根目录，在任何情况下这个 URL 都不是特殊的 就像在 URLconf 中的其他条目一样，它也依赖于指定给它的 URL 模式.

尽管匹配网站根目录的 URL 模式不能想象，但是还是值得提一下的. 当为网站根目录实现一个视图，你需要使用 URL 模式`` '^$'`` , 它代表一个空字符串。 例如:

```
from mysite.views import hello, my_homepage_view

urlpatterns = patterns('',
    ('^$', my_homepage_view),
    # ...
)
```

Django 是怎么处理请求的

在继续我们的第二个视图功能之前，让我们暂停一下去了解更多一些有关 Django 是怎么工作的知识. 具体地说，当你通过在浏览器里敲 http://127.0.0.1:8000/hello/来访问 Hello world 消息得时候，Django 在后台有些什么动作呢？

所有均开始于 setting 文件。当你运行 python manage.py runserver，脚本将在于 manage.py 同一个目录下查找名为 setting.py 的文件。这个文件包含了所有有关这个 Django 项目的配置 信息，均大写： TEMPLATE_DIRS , DATABASE_NAME , 等. 最重要的设置时 ROOT_URLCONF，它将作为 URLconf 告诉 Django 在这个站点中那些 Python 的模块将被用到

还记得什么时候 django-admin.py startproject 创建文件 settings.py 和 urls.py 吗？自动创建的 settings.py 包含一个 ROOT_URLCONF 配置用 来指向自动产生的 urls.py. 打开文件 settings.py 你将看到如下：

ROOT_URLCONF = 'mysite.urls'

相对应的文件是 mysite/urls.py

总结一下：

1. 进来的请求转入/hello/.
2. Django 通过在 ROOT_URLCONF 配置来决定根 URLconf.
3. Django 在 URLconf 中的所 有 URL 模式中，查找第一个匹配/hello/的条目。
4. 如果找到匹配，将调用相应的视图函数
5. 视图函数返回一个 HttpResponse
6. Django 转换 HttpResponse 为一个适合的 HTTP response， 以 Web page 显示出来

第二个视图： 动态内容

我们的第二个视图，将更多的放些动态的东西例如当前日期和时间显示在网页上 这将非常好，简单的下一步，因为它不引入数据库或者任何用户的输入，仅仅是输出显示你的服务器的内部时钟. 它仅仅有限度的比 Helloworld 刺激一些，但是它将演示一些新的概念

为了让 Django 视图显示当前日期和时间，我们仅需要把语句：datetime.datetime.now()放入视图函数，然后返回一个 HttpResponse 对象即可

让我们分析一下改动后的 views.py：

在文件顶端，我们添加了一条语句：import datetime。这样就可以计算日期了。

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。 字符串中的%s 是占位符，字符串后面的百分号表示用它后面的变量 now 的值来代替%s。变量%s 是一个 datetime.datetime 对象。它虽然不 是一个字符串，但是%s（格式化字符串）会把它转换成字符串，如：2008-12-13 14:09:39.002731。这将导致 HTML 的输出字符串为：It is now 2008-12-13 14:09:39.002731。

（目前 HTML 是有错误的，但我们这样做是为了保持例子的简短。）

添加上述代码之后，还要在 urls.py 中添加 URL 模式，以告诉 Django 由哪一个 URL 来处理这个视图。 用/time/之类的字眼易于理解：

Django 时区

视乎你的机器，显示的日期与时间可能和实际的相差几个小时。 这是因为 Django 是有时区意识的，并且默认时区为 America/Chicago。 （它必须有个值，它的默认值是 Django 的诞生地：美国/芝加哥）如果你处在别的时区，你需要在 settings.py 文件中更改这个值。请参见它里面 的注释，以获得最新世界时区列表。

URL 配置和松耦合

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。

例如，考虑一下 current_datetime 视图。 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。

```
urlpatterns = patterns('',
    ('^hello/$', hello),
    ('^time/$', current_datetime),
    ('^another-time-page/$', current_datetime),
)
```

第三个视图 动态 *URL*

在我们的`` current_datetime`` 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。 举个例子，一家在线书店会为每一本书提供一个 URL，如：/books/243/、/books/81196/。

让我们创建第三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('',
```

```
    ('^time/$', current_datetime),
    ('^time/plus/1/$', one_hour_ahead),
    ('^time/plus/2/$', two_hours_ahead),
    ('^time/plus/3/$', three_hours_ahead),
    ('^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。

那么，我们如何设计程序来处理任意数量的时差？ 答案是：使用通配符（wildcard URLpatterns）。正如我们之前提到过，一个 URL 模式就是一个正则表达式。因此，这里可以使用 d+ 来匹配 1 个以上的数字。

```
urlpatterns = patterns('',

    # ...
    (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),
```

最终的 URLconf 包含上面两个视图，如：

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

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

现在开始写 **`hours_ahead`** 视图。

编码次序

这个例子中，我们先写了 URLpattern ，然后是视图，但是在前面的例子中， 我们先写了视图，然后是 URLpattern 。 哪一种方式比较好？

嗯，怎么说呢，每个开发者是不一样的。

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。

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。 (见上)

`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。 以下是 view 代码：

```
    from django.http import Http404, HttpResponse
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    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)
```

视图函数, `hours_ahead` , 有 两个 参数: `request` 和 `offset` . (见上)

> `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。 (见上)
>
> `offset` 是从匹配的 URL 里提取出来的。 例如：如果请求 URL 是/time/plus/3/，那么 offset 将会是 3；如果请求 URL 是/time/plus/21/，那么 offset 将会是 21。请注意：捕获值永远都是字符串（string）类型，而不会是整数（integer）类型，即使这个字符串全由数字构成（如：“21”）。
>
> （从技术上来说，捕获值总是 Unicode objects，而不是简单的 Python 字节串，但目前不需要担心这些差别。）
>
> 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。

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。

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):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()
    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:  unicode"` .

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。

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？ 页面没有发现错误，如前所述，工作正常。）

第四章 模板

在前一章中，你可能已经注意到我们在例子视图中返回文本的方式有点特别。 也就是说，HTML 被硬性地直接写入 Python 代码之中。

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

尽管这种技术便于解释视图是如何工作的，但直接将 HTML 硬编码到你的视图里却并不是一个好主意。 让我们来看一下为什么：

- 对页面设计进行的任何改变都必须对 Python 代码进行相应的修改。 站点设计的修改往往比底层 Python 代码的修改要频繁得多，因此如果可以在不进行 Python 代码修改的情况下变更设计，那将会方便得多。
- 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。 设计者和 HTML/CSS 的编码人员不应该被要求去编辑 Python 的代码来完成他们的工作。
- 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

模板是一个文本，用于分离文档的表现形式和内容。 模板定义了占位符以及各种用于规范文档该如何显示的各部分基本逻辑（模板标签）。 模板通常用于产生 **HTML**，但是 **Django** 的模板也能产生任何基于文本格式的文档。

让我们从一个简单的例子模板开始。 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>
```

```
<h1>Ordering notice</h1>

<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>
{% else %}
    <p>You didn't order a warranty, so you're on your own when
    the products inevitably stop working.</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 。 让我们逐步分析一下：

用两个大括号括起来的文字（例如 `{{ person_name }}` ）称为 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的标签。

这个例子中的模板包含一个 `for` 标签（ `{% for item in item_list %}` ）和一个 `if` 标签（`{% if ordered_warranty %}` ）

`if` 标签，正如你所料，是用来执行逻辑判断的。 在这里，tag 标签检查 ordered_warranty 值是否为 True。如果是，模板系统将显示{% if ordered_warranty %}和{% else %}之间的内容；否则将显示{% else %}和{% endif %}之间的内容。{% else %}是可选的。

如何使用模板系统

让我们深入研究模板系统，你将会明白它是如何工作的。但我们暂不打算将它与先前创建的视图结合在一起， 因为我们现在的目的是了解它是如何独立工作的 。 （换言之， 通常你会将模板和视图一起使用，但是我们为了突出模板系统是一个 Python 库，你可以在任何地方使用它，而不仅仅是在 Django 视图中。）

在 Python 代码中使用 Django 模板的最基本方式如下：

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. 调用模板对象的 render 方法，并 且传入一套变量 context。它将返回一个基于模板的展现字符串，模板中的变量和标签会被 context 值替换。

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。

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。

一个特殊的 Python 提示符

如果你曾经使用过 Python，你一定好奇，为什么我们运行 python manage.py shell 而不是 python。这两个命令都会启动交互解释器，但是 manage.py shell 命 令有一个重要的不同： 在启动解释器之前，它告诉 Django 使用哪个设置文件。 Django 框架的大部分子系统，包括模板系统，都依赖于配置文件；如果 Django 不知道使用哪个配置文件，这些系统将不能工作。

让我们来了解一些模板系统的基本知识：

```
>>> 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。

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'
```

这里，块标签(block tag)指向的是`` {% notatag %}``，块标签与模板标签是同义的。

系统会在下面的情形抛出 TemplateSyntaxError 异常：

- 无效的 tags
- 标签的参数无效
- 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。

context 在 Django 里表现为 `Context` 类，在 `django.template` 模块里。 她的构造函数带有一个可选的参数： 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)
u'My name is Stephane.'
```

我们必须指出的一点是，`t.render(c)`返 回的值是一个 Unicode 对象，不是普通的 Python 字符串。 你可以在字符串前加上 `u` 来声明。 在框架中，Django 会一直使用 Unicode 对象而不是普通的字符串。 如果你明白这样做给你带来了多大便利的话，尽可能地感激 Django 在幕后有条不紊地为你所做这这么多工作吧。 如果不明白你从中获益了什么，别担心。你只需要知道 Django 对 Unicode 的支持，将让你的应用程序轻松地处理各式各样的字符集，而不仅仅是基本的 A-Z 英文字符。

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 placing an order 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>
... {% else %}
... <p>You didn't order a warranty, so you're on your own when
... the products inevitably stop working.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': False})
>>> t.render(c)
u"<p>Dear John Smith,</p>\n\n<p>Thanks for placing an order from Outdoor
Equipment. It's scheduled to\nship on April 2, 2009.</p>\n\n\n<p>You
didn't order a warranty, so you're on your own when\nthe products
inevitably stop working.</p>\n\n\n<p>Sincerely,<br />Outdoor Equipment
</p>"
```

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 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
```

无论何时我们都可以像这样使用同一模板源渲染多个context，只进行 一次模板创建然后多次调用 render()方法渲染会更为高效：

```
# 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}))
```

背景变量的查找

在 Django 模板中遍历复杂数据结构的关键是句点字符 (.)。

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)
u'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)
u'The month is 5 and the year is 1993.'
```

这个例子使用了一个自定义的类，演示了通过实例变量加一点(dots)来访问它的属性，这个方法适用于 任意的对象。

```
>>> 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)
u'Hello, John Smith.'
```

点语法也可以用来引用对象的* 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'}))
u'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
u'123 -- 123 -- True'
```

Python 列表类型

一点提示： Python 的列表是从 0 开始索引。 第一项在索引 0 位置上，第二项在索引 1 位置上，依此类推。

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：

- 字典类型查找 （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)
u'SALLY is 43 years old.'
```

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。

```
>>> 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}))
u'My name is .'
```

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： 例如：

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

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']
Traceback (most recent call last):
  ...
KeyError: 'foo'
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

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 %}
```

`{% 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： 比如这样的代码是不合法的：

```
{% 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。 如果你确实需要用到圆括号来组合表达你的逻辑式，考虑将它移到模板之外处理，然后以模板变量的形式传入结果吧。 或者，仅仅用嵌套的`{% if %}`标签替换吧，就像这样：

```
{% 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。 例如，这些事合法的：

```
{% 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。

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：**reversed(seq)**

接受一个序列作为参数,返回一个以逆序访问的迭代器在 **python** 中 **for i in reversed(seq)**

:

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

```
...
{% endfor %}
```

可以嵌套使用 `{% for %}` 标签

在执行循环之前先检测列表的大小是一个通常的做法，当列表为空时输出一些特别的提示。

因为这种做法十分常见，所以`` for`` 标签支持一个可选的`` {% empty %}`` 分句，通过它我们可以定义当列表为空时的输出内容 下面的例子与之前那个等价：

```
{% for athlete in athlete_list %}
    <p>{{ athlete.name }}</p>
{% empty %}
    <p>There are no athletes. Only computer programmers.</p>
{% 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）

`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。

```
{% 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。
>
> `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。

`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（|）

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。

注释

就像 HTML 或者 Python，Django 模板语言同样提供代码注释。 注释使用 `{# #}`：

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

注释的内容不会在模板渲染时输出。

用这种语法的注释不能跨越多行。 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.
```

如果要实现多行注释，可以使用`` {% comment %}`` 模板标签，就像这样：

```
{% comment %}
This is a
multi-line comment.
{% endcomment %}
```

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。 过滤器使用管道字符，如下所示：

```
{{ 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。

过滤管道可以被* 套接* ，既是说，一个过滤器管道的输出又可以作为下一个管道的输入，如此下去。 下面的例子实现查找列表的第一个元素并将其转化为大写。

```
{{ my_list|first|upper }}
```

有些过滤器有参数。 过滤器的参数跟随冒号之后并且总是以双引号包含。 例如：

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

这个将显示变量 `bio` 的前 30 个词。

以下几个是最为重要的过滤器的一小部分。 附录 F 包含其余的过滤器。

`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。

`length` : 返回变量的长度。 对于列表，这个参数将返回列表元素的个数。 对于字符串，这个参数将返回字符串中字符的个数。 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

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：模板通常由设计师来编写而非程序员，因此不应被假定拥有 Python 开发知识。

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）

目标并不是要发明一种编程语言 。目标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。

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，模板仍然嵌入在 Python 代码里，并未真正的实现数据与表现的分离。 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 is BAD because it 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。

为了解决这些问题，我们采用了 模板自加载 跟 模板目录 的技巧.

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。 设置的保存文件就是我们前一章节中提到的 settings.py，在 讲述 ROOT_URLCONF 配 置的时候。

如果你是一步步跟随我们学习过来的，马上打开你的 settings.py 配置文件，找到 TEMPLATE_DIRS 这项设置 吧。 它的默认设置是一个空元组（tuple），加上一些自动生成的注释。

TEMPLATE_DIRS = (

"/home/jhjguxin/Desktop/djcode/mysite/templates"

,

Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".

Always use forward slashes, even on Windows.

Don't forget to use absolute paths, not relative paths.

)

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）。

如果你的 TEMPLATE_DIRS 只包含一个目录，别忘了在该目录后加上个逗号。

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： 例如：

```
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。 `` os.path.dirname(__file__)`` 将会获取自身所在的文件，即 settings.py 所在的目录，然后由 os.path.join 这个方法将这目录与 templates 进行连接。如果在 windows 下，它会智能地选择正确的后向斜杠"“进行连接，而不是前向斜杠"/"。

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。

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()

我们已经告诉你如何载入一个模板文件，然后用 Context 渲染它，最后返回这个处理好的 HttpResponse 对象给用 户。 我们已经优化了方案，使用 get_template() 方法代替繁杂

的用代码来处理模板及其路径的工作。 但这仍然需要一定量的时间来敲出这些简化的代码。这是一个普遍存在的重复苦力劳动。Django 为此提供了一个捷径，让你一次性地载入某个模板文件，渲染它，然后将此作为 `HttpResponse` 返 回。

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})
```

`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。

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())
```

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。

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。

```
return render_to_response('dateapp/current_datetime.html', {'current_date': now})
```

对子目录树的深度没有限制，你想要多少层都可以。 只要你喜欢，用多少层的子目录都无所谓。

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。

```
{% 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。 举例说，考虑下面两个模板文件：

```
    # mypage.html

<html>
<body>
{% include "includes/nav.html" %}
<h1>{{ title }}</h1>
</body>
</html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

如果你用一个包含 `current_section` 的上下文去渲染 `mypage.html` 这个模板文件，这个变量将存在于它所包含 （include）的模板里，就像你想象的那样。

若果碰到一个没有定义名字的 `{% include %}`标签，Django 将会在下面两个处理方法中选择一个：

- 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：

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。 （如果你是跟着来的话，保存这个文件到你的 template 目录下，命名为 `base.html` .）

我们使用一个以前已经见过的模板标签： `{% block %}` 。 所有的 `{% block %}` 标签告诉模板引擎，子模板可以重载这些部分。 每个`{% block %}`标签所要做的是告诉模板引擎，该模板下的这一块内容将有可能被子模板覆盖。

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。

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。

继承并不会影响到模板的上下文。 换句话说，任何处在继承树上的模板，将可以访问到你传到模板中的每一个模板变量。

你可以根据需要使用任意多的继承次数。 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。

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。
- 如果你需要访问父模板中的块的内 容，使用 **{{ block.super }}**这个标 签吧，这一个魔法变量将会表现出父模板中的内容。 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。

第 5 章 模型

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。 作为例子之一，我们的主观逻辑是要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。 这个网站也许也提供网站访问者取得自己数据库的方法。

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。 （如果你正在建立一个开源的 Django 应用程序来尽可能让更多人使用的话，这个特性是非常适当的。）

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

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。

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* 代表视图（View），即业务逻辑层。 该层包含存取模型及调取恰当模板的相关逻辑。 你可以把它看作模型与模板之间的桥梁。

数据库配置

数据库配置也是在 Django 的配置文件里，缺省 是 `settings.py` 。

```
DATABASES = {

    'default': {

        'ENGINE': 'mysql',

                # Add 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.

        'NAME':'mysql',                    # Or path to database file if using sqlite3.

        'USER': 'root',                    # Not used with sqlite3.

        'PASSWORD': 'jhjwwew12315123',                # Not used with sqlite3.

        'HOST': '/var/run/mysqld/mysqld.sock',                    # Set to empty string for localhost. Not used with sqlite3.

        'PORT': '3306',                    # Set to empty string for default. Not used with sqlite3.

    }

}
```

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。 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* , 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，例如注释系统和自动管理界面。 app 的一个关键点是它们是很容易移植到其他 project 和被多个 project 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 。 在之前那些例子中，我们只是简单的创建了一个称为 views.py 的文件，编写了一些函数并将在 URLconf 中设置各个函数的映射。 这些情况都不需要使用 apps。

但是，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。 模型必须存放在 apps 中。 因此，为了开始建造 我们的模型，我们必须创建一个新的 app。

在`` mysite`` 项目文件下输入下面的命令来创建`` books`` app

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

这个命令并没有输出什么，它只在 `mysite` 的目录里创建了一个 `books` 目录。 让我们来看看这个目录的内容：

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

这个目录包含了这个 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。

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。

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。 我们将在第 18 章中对此进行讨论。

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：

- 一个作者有姓，有名及 email 地址。
- 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]）

第一步是用 Python 代码来描述它们。 打开由`` startapp`` 命令创建的 `models.py` 并输入下面的内容：

```
#!/usr/bin/env python


# -*- coding: UTF-8 -*-


from django.db import models


"""


我们来假定下面的这些概念、字段和关系：




 * 一个作者有姓，有名及 email 地址。




 * 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]）

"""

```
# Create your models here.

#出版商有名称，地址，所在城市、省，国家，网站。

class Publisher(models.Model):

  name=models.CharField(max_length=30)

  address = models.CharField(max_length=50)

  city=models.CharField(max_length=60)

  state_province=models.CharField(max_length=30)

  country=models.CharField(max_length=50)

  website=models.URLField()

#一个作者有姓，有名及 email 地址。

class Author(models.Model):

  first_name=models.CharField(max_length=30)

  last_name=models.CharField(max_length=40)
```

email=models.EmailField()

#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]）

class Book(models.Model):

title=models.CharField(max_length=100)

authors=models.ManyToManyField(Author)

publisher=models.ForeignKey(Publisher)

publication_date=models.DateField()

每个模型相当于单个数据库表，每个属性也是这个表中的一个字段。 属性名就是字段名，它的类型（例如 CharField ）相当于数据库的字段类型 （例如 varchar ）。例如， Publisher 模块等同于下面这张表（用 Postgresql 的 CREATE TABLE 语法描述）：

```
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。

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。（这四个 app 是经常使用到的，我们将在后续章节里讨论如何使用它们）。同时，注释掉 MIDDLEWARE_CLASSES 的默认设置条目，因为这些条目是依赖于刚才我们刚在 INSTALLED_APPS 注释掉的 apps。 然后，添加`` 'mysite.books''`` 到`` INSTALLED_APPS`` 的末尾，此时设置的内容看起来应该是这样的：

```
MIDDLEWARE_CLASSES = (

#   'django.middleware.common.CommonMiddleware',

#   'django.contrib.sessions.middleware.SessionMiddleware',

#   'django.middleware.csrf.CsrfViewMiddleware',

    #'django.contrib.auth.middleware.AuthenticationMiddleware',

#   'django.contrib.messages.middleware.MessageMiddleware',

)


INSTALLED_APPS = (

#   'django.contrib.auth',

#   'django.contrib.contenttypes',

#   'django.contrib.sessions',

#   'django.contrib.sites',

#   'django.contrib.messages',

    # Uncomment the next line to enable the admin:
```

'django.contrib.admin',

Uncomment the next line to enable admin documentation:

'django.contrib.admindocs',

'mysite.books',

)

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。

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
```

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。
- 我们前面已经提到，Django 为每个表格自动添加加了一个 `id` 主键， 你可以重新设置它。
- 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，字段名称也是自动处理（例如单引号还好是双引号）。

`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。 如果要创建数据表，你可以复制那些 SQL 语句到你使用数据库客户端后执行，或者通过 Unix 管道直接进行操作（例如，`` python manager.py sqlall books | psql mydb`` ）。不过，Django 提供了一种更为简易的提交 SQL 语句至数据库的方法： `` syncdb`` 命令

```
python manage.py syncdb
```

执行这个命令后，将看到类似以下的内容：

jhjguxin@jhjguxin-HP-Pavilion-dv2000-RQ116PA-AB2:~/Desktop/djcode/mysite$ python manage.py syncdb

Creating table books_publisher

Creating table books_author

Creating table books_book_authors

Creating table books_book

Installing index for books.Book_authors model

Installing index for books.Book model

No fixtures found.

jhjguxin@jhjguxin-HP-Pavilion-dv2000-RQ116PA-AB2:~/Desktop/djcode/mysite$ python manage.py syncdb

No fixtures found.

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。 你可以手动启动命令行客户端（例如，执行 PostgreSQL 的`` psql`` 命令），也可以执行 `` python manage.py dbshell`` ，这个命令将依据`` DATABASE_SERVER`` 的里设置自动检测使用何种命令行客户端。 常言说，后来者居上。

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 Python 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='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', 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：

- 首先，导入 Publisher 模 型类， 通过这个类我们可以与包含 出版社 的数据表进行交互。
- 接着，创建一个`` Publisher`` 类的实例并设置了字段`` name, address`` 等的值。

- 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。
- 最后，使用`` Publisher.objects`` 属性从数据库取出出版商的信息，这个属性可以认为是包含出版商的记录集。 这个属性有许多方法， 这里先介绍调用`` Publisher.objects.all()`` 方法获取数据库中`` Publisher`` 类的所有对象。这个操作的幕后，Django 执行了一条 SQL `` SELECT`` 语句。

这里有一个值得注意的地方，在这个例子可能并未清晰地展示。 当你使用 Django modle API 创建对象时 Django 并未将对象保存至数据库内，除非你调用`` save()`` 方法：

```
p1 = Publisher(...)
# At this point, p1 is not saved to the database yet!
p1.save()
# Now it is.
```

如果需要一步完成对象的创建与存储至数据库，就使用`` **objects.create()**`` 方法。 下面的例子与之前的例子等价：

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
```

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 `__unicode__()` 到 `Publisher` 对象。 `__unicode__()` 方法告诉 Python 如何实现对象的 unicode 表示。 请看下面：

#出版商有名称，地址，所在城市、省，国家，网站。

class Publisher(models.Model):

name=models.CharField(max_length=30)

address = models.CharField(max_length=50)

city=models.CharField(max_length=60)

```
  state_province=models.CharField(max_length=30)

  country=models.CharField(max_length=50)

  website=models.URLField()

  def __unicode__(self):

    return self.name
```

#一个作者有姓，有名及 email 地址。

```
class Author(models.Model):

  first_name=models.CharField(max_length=30)

  last_name=models.CharField(max_length=40)

  email=models.EmailField()

  def __unicode__(self):

      return u'%s %s'%(self.first_name,self.last_name)
```

#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]）

```
class Book(models.Model):

  title=models.CharField(max_length=100)

  authors=models.ManyToManyField(Author)
```

```
publisher=models.ForeignKey(Publisher)

publication_date=models.DateField()

def __unicode__(self):

  return self.title
```

就象你看到的一样， `__unicode__()` 方法可以进行任何处理来返回对一个对象的表示。 `__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。

对__unicode__()的唯一要求就是它要返回一个 unicode 对象 如果`` __unicode__()`` 方法未返回一个 Unicode 对象，而返回比如说一个整型数字，那么 Python 将抛出一个`` TypeError`` 错误，并提示：”coercing to Unicode: need string or buffer, int found” 。

普通的 python 字符串是经过**， 意思就是它们已经经过某种编码（如 ASCII，ISO-8859-1 或者 UTF-8）来编码。 如果你在保存奇特的字符（其它任何超出标准 128 个如 0-9 和 A-Z 之类的 ASCII 字符）在一个普通的 python 字符串里，你一定要跟踪你的字符串是 用什么编码的，否则这些奇特的字符可能会在显示或者打印的时候出现乱码。 当你尝试要将用某种编码保存的数据结合到另外一种编码的数据中，或者你想要把它显示在已经假定了某种编码的程序中的时候，问题就会发生。 我们都已经见到过网页和邮件被???弄得乱七八糟。 ?????? 或者其它出现在奇怪位置的字符：这一般来说就是存在编码问题了。

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。

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

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。

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


请确保你的每一个模型里都包含 `__unicode__()` 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 `__unicode__()` 来显示对象。

最后， `__unicode()__` 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。 `__unicode__()` 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。 在调用`` save()`` 方法之前，记录并没有保存至数据库，像这样：

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

在 SQL 里，这大致可以转换成这样：

```
INSERT INTO books_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 books_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

当然，创建新的数据库，并更新之中的数据是必要的，但是，对于 Web 应用程序来说，更多的时候是在检索查询数据库。 我们已经知道如何从一个给定的模型中取出所有记录：

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

这相当于这个 SQL 语句：

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

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？ 它被称为管理器，我们将在第 10 章中详细讨论它。 目前，我们只需了解管理器管理着所有针对数据包含、还有最重要的数据查询的表格级操作。

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。

数据过滤

我们很少会一次性从数据库中取出所有的数据；通常都只针对一部分数据进行操作。 在 Django API 中，我们可以使用`` filter()`` 方法对数据进行过滤：

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

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 books_publisher
WHERE name = 'Apress';
```

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>]
```

多个参数会被转换成 AND SQL 语句， 因此上面的代码可以转化成这样：

```
SELECT id, name, address, city, state_province, country, website
FROM books_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， 包含性查找可以使用：

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

在 name 和 contains 之间有双下划线。

```
SELECT id, name, address, city, state_province, country, website
FROM books_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）。

获取单个对象

上面的例子中`` filter()`` 函数返回一个记录集，这个记录集是一个列表。 相对列表来说，有些时候我们更需要获取单个的对象， `` get()`` 方法就是在此时使用的：

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

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):
    ...
MultipleObjectsReturned: get() returned more than one Publisher --
    it returned 2! Lookup parameters were {'country': 'U.S.A.'}
```

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.
```

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.
```


这个 `DoesNotExist` 异常 是 Publisher 这个 model 类的一个属性，即 `Publisher.DoesNotExist`。 在你的应用中，你可以捕获并处理这个异常，像这样：

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print "Apress isn't in the database yet."
else:
    print "Apress is in the database."
```

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。

在你的 Django 应用中，你或许希望对检索结果，根据某字段的值，按字母顺序排序。 那么，使用 `order_by()` 这个方法就可以搞定了。

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress>, <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 books_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>]
```

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

如果需要以多个字段为标准进行排序（第二个字段会在第一个字段的值相同的情况下被使用到），使用多个 参数就可以了，如下：

```
>>> Publisher.objects.order_by("state_province", "address")
 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

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>]
```

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(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

      class Meta:
        ordering = ['name']
```

现在，让我们来接触一个新的概念。 `class Meta`，内嵌于 `Publisher` 这个类的定义中（如果 `class Publisher` 是顶格的，那么 `class Meta` 在它之下要缩进 4 个空格－－按 Python 的传统 ）。你可以在任意一个 模型 类中使用 `Meta` 类，来设置一些与特定模型相关的选项。 在 附录 B 中有 `Meta` 中所有可选项的完整参考，现在，我们关注 `ordering` 这个选项就够了。 如果你设置了这个选项，那么除非你检索时特意额外地使用了 `order_by()`，否则，当你使用 Django 的数据库 API 去检索时，`Publisher` 对象的相关返回值 默认地都会按 `name` 字段排序。

连锁查询

我们已经知道如何对数据进行过滤和排序。 当然，通常我们需要同时进行过滤和排序查询的操作。 因此，你可以简单地写成这种"链式"的形式：

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

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 books_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

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.order_by('name')[0]
<Publisher: Apress>
```

这相当于：

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

类似的，你可以用 python 的 range-slicing 语法来取出数据的特定子集

```
>>> Publisher.objects.order_by('name')[0:2]
```

这个例子返回两个对象，等同于以下的 SQL 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

注意，不支持 **Python** 的负索引 **(negative slicing)**：

```
>>> Publisher.objects.order_by('name')[-1]
Traceback (most recent call last):
  ...
AssertionError: Negative indexing is not supported.
```

虽然不支持负索引，但是我们可以使用其他的方法。 比如，稍微修改 **order_by()** 语句来实现：

```
>>> Publisher.objects.order_by('-name')[0]
```

更新多个对象

在“插入和更新数据”小节中，我们有提到模型的 save()方法，这个方法会更新一行里的所有列。 而某些情况下，我们只需要更新行里的某几列。

例如说我们现在想要将 Apress Publisher 的名称由原来的”Apress”更改为”Apress Publishing”。若使用 save()方法，如：

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

这等同于如下 SQL 语句：

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

UPDATE books_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;
```

（注意在这里我们假设 Apress 的 ID 为 52）

在这个例子里我们可以看到 Django 的 save()方法更新了不仅仅是 name 列的值，还有更新了 所有的列。 若 name 以外的列有可能会被其他的进程所改动的情况下，只更改 name 列显然是更加明智的。 更改某一指定的列，我们可以调用结果集（QuerySet）对象的 update()方法： 示例如下：

```
>>> Publisher.objects.filter(id=1).update(name='Apress Pulishing')



1L




>>> Publisher.objects.order_by()[0]<Publisher: Apress Pulishing>
```

与之等同的 SQL 语句变得更高效，并且不会引起竞态条件。

```
UPDATE books_publisher
SET name = 'Apress Publishing'
WHERE id = 52;
```

update()方法对于任何结果集（QuerySet）均有效，这意味着你可以同时更新多条记录。 以下示例演示如何将 Publisher 的 country 字段值为'U.S.A'更改为'USA'：

```
>>> Publisher.objects.all().update(country='USA')
2L
```

删除对象

删除数据库中的对象只需调用该对象的 delete()方法即可：

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

同样我们可以在结果集上调用 delete()方法同时删除多条记录。这一点与我们上一小节提到的 update()方法相似：

```
>>> Publisher.objects.filter(country='USA').delete()
>>> Publisher.objects.all().delete()
>>> Publisher.objects.all()
[]
```

删除数据时要谨慎！ 为了预防误删除掉某一个表内的所有数据，Django 要求若要删除表内所有数据时要使用 all()来明确指定。 比如，下面的操作将会出错：

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'Manager' object has no attribute 'delete'
```

而一旦使用 all()方法，所有数据将会被删除：

```
>>> Publisher.objects.all().delete()
```

如果只需要删除部分的数据，就不需要调用 all()方法。再看一下之前的例子：

```
>>> Publisher.objects.filter(country='USA').delete()
```

第六章 **Django** 站点管理

对于某一类网站， 管理界面 是基础设施中非常重要的一部分。 这是以网页和有限的可信任管理者为基础的界面，它可以让你添加，编辑和删除网站内容。 一些常见的例子： 你可以用这个界面发布博客，后台的网站管理者用它来润色读者提交的内容，你的客户用你给他们建立的界面工具更新新闻并发布在网站上，这些都是使用管理界面 的例子。

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。

django.contrib 包

Django 自动管理工具是 django.contrib 的一部分。django.contrib 是一套 庞大的功能集，它是 Django 基本代码的组成部分，Django 框架就是由众多包含附加组件(add-on)的基本代码构成的。 你可以把 django.contrib 看作是可选的 Python 标准库或普遍模式的实际实现。 它们与 Django 捆绑在一起，这样你在开发中就不用“重复发明轮子”了。

管理工具是本书讲述 django.contrib 的第一个部分。从技术层面上讲，它被称作 django.contrib.admin。django.contrib 中其它可用的特性，如用户鉴别系统 (django.contrib.auth)、支持匿名会话(django.contrib.sessioins)以及用户评注系统 (django.contrib.comments)。这些，我们将在第十六章详细讨论。在成为一个 Django 专家以前，你将会知道更多 django.contrib 的特性。 目前，你只需要知道 Django 自带很多优秀的附加组件，它们都存在于 django.contrib 包里。

激活管理界面

Django 管理站点完全是可选择的，因为仅仅某些特殊类型的站点才需要这些功能。 这意味着你需要在你的项目中花费几个步骤去激活它。

第一步，对你的 settings 文件做如下这些改变：

1. 将`'django.contrib.admin'`加 入 setting 的 `INSTALLED_APPS` 配 置中（`INSTALLED_APPS` 中 的配置顺序是没有关系的, 但是我们喜欢保持一定顺序以方便人来阅读）
2. 保证 `INSTALLED_APPS` 中包含`'django.contrib.auth'`，`'django.contrib.contenttypes'`和`'django.contrib.sessions'`，Django 的管理工具需要这 3 个包。 (如果你跟随本文制作 mysite

项目的话，那么请注意我们在第五章的时候把这三项 INSTALLED_APPS 条目注释了。现在，请把注释取消。)

3. 确保 `MIDDLEWARE_CLASSES` 包含`'django.middleware.common.CommonMiddleware'`、`'django.contrib.sessions.middleware.SessionMiddleware'` 和`'django.contrib.auth.middleware.AuthenticationMiddleware'`。(再次提醒，如果有跟着做 mysite 的话，请把在第五章做的注释取消。)

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。 当你把`'django.contrib.auth'`加 进 `INSTALLED_APPS` 后， 第一次运行 `syncdb` 命令 时, 系统会请你创建一个超级用户。 如果你不这么作，你需要运行 `python manage.py createsuperuser` 来另外创建一个 admin 的用户帐号，否则你将不能登入 admin (提醒一句: 只有当 `INSTALLED_APPS` 包 含`'django.contrib.auth'`时，`python manage.py createsuperuser` 这 个命令才可用.)

jhjguxin@jhjguxin-HP-Pavilion-dv2000-RQ116PA-AB2:~/Desktop/djcode/mysite$ python manage.py syncdb

Creating table django_admin_log

Creating table auth_permission

Creating table auth_group_permissions

Creating table auth_group

Creating table auth_user_user_permissions

Creating table auth_user_groups

Creating table auth_user

Creating table auth_message

Creating table django_content_type

Creating table django_session

You just installed Django's auth system, which means you don't have any superusers defined.

Would you like to create one now? (yes/no): y

Please enter either "yes" or "no": yes

Username (Leave blank to use 'jhjguxin'): admin

E-mail address: admin@admin.com

Password:

Password (again):

Superuser created successfully.

Installing index for admin.LogEntry model

Installing index for auth.Permission model

Installing index for auth.Group_permissions model

Installing index for auth.User_user_permissions model

Installing index for auth.User_groups model

Installing index for auth.Message model

No fixtures found

第三，将 `admin` 访问配置在 `URLconf`(记住，在 `urls.py` 中). 默认情况下，命令 `django-admin.py startproject` 生成的文件 `urls.py` 是将 `Django admin` 的路径注释掉的，你所要做的就是取消注释。 请注意，以下内容是必须确保存在的：

```
# Include these import statements...
from django.contrib import admin
admin.autodiscover()
```

```
# And include this URLpattern...
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

当这一切都配置好后，现在你将发现 Django 管理工具可以运行了。 启动开发服务器(如前：`` python manage.py runserver``)，然后在浏览器中访问：http://127.0.0.1:8000/admin/

使用管理工具。

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。

在 Django 管理页面中，每一种数据类型都有一个* change list* 和* edit form* 。前者显示数据库中所有的可用对象；后者可让你添加、更改和删除数据库中的某条记录。

其它语言

如果你的母语不是英语，而你不想用它来配置你的浏览器，你可以做一个快速更改来观察 Django 管理工 具是否被翻译成你想要的语言。 仅需添加`` 'django.middleware.locale.LocaleMiddleware''`` 到`` MIDDLEWARE_CLASSES`` 设置中，并确保它在'django.contrib.sessions.middleware.SessionMiddleware'* 之后* 。 （见上）

完成后，请刷新页面。 如果你设置的语言可用，一系列的链接文字将被显示成这种语言。这些文字包括页面顶端的 Change password 和 Log out，页面中部的 Groups 和 Users。 Django 自带了多种语言的翻译。

关于 Django 更多的国际化特性，请参见第十九章。

将你的 *Models* 加入到 *Admin* 管理中

有一个关键步骤我们还没做。 让我们将自己的模块加入管理工具中，这样我们就能够通过这个漂亮的界面添加、修改和删除数据库中的对象了。 我们将继续第五章中的`` book`` 例子。在其中，我们定义了三个模块： Publisher 、 Author 和 Book 。

在`` books`` 目录下(`` mysite/books``)，创建一个文件：`` admin.py`` ，然后输入以下代码：

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

这些代码通知管理工具为这些模块逐一提供界面。

完成后，打开页面 `` http://127.0.0.1:8000/admin/`` ，你会看到一个 Books 区域，其中包含 Authors、Books 和 Publishers。 （你可能需要先停止，然后再启动服务(`` runserver``)，才能使其生效。）

现在你拥有一个功能完整的管理界面来管理这三个模块了。 很简单吧！

花点时间添加和修改记录，以填充数据库。 如果你跟着第五章的例子一起创建 Publisher 对象的话（并且没有删除），你会在列表中看到那些记录。

这里需要提到的一个特性是，管理工具处理外键和多对多关系（这两种关系可以在`` Book`` 模块中找到）的方法。 作为提醒，这里有个`` Book`` 模块的例子：

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

    def __unicode__(self):
        return self.title
```

在 Add book 页面中（`` http://127.0.0.1:8000/admin/books/book/add/`` ），`` 外键`` publisher 用一个选择框显示，`` 多对多`` 字段 author 用一个多选框显示。 点击两个字段后面的绿色加号，可以让你添加相关的记录。 举个例子，如果你点击 Publisher 后面的加号，你将会得到一个弹出窗口来添加一个 publisher。 当你在那个窗口中成功创建了一个 publisher 后，Add book 表单会自动把它更新到字段上去 花巧.

Admin 是如何工作的

在幕后，管理工具是如何工作的呢？ 其实很简单。

当服务启动时，Django 从`` url.py`` 引导 URLconf，然后执行`` admin.autodiscover()`` 语句。 这个函数遍历 `INSTALLED_APPS` 配置，并且寻找相关的 `admin.py` 文件。 如果在指定的 app 目录下找到 `admin.py`，它就执行其中的代 码。

在`` books`` 应用程序目录下的`` admin.py`` 文件中，每次调用`` admin.site.register()`` 都将那个模块注册到管理工具中。 管理工具只为那些明确注册了的模块显示一个编辑/修改的界面。

应用程序`` django.contrib.auth`` 包含自身的`` admin.py`` ，所以 Users 和 Groups 能在管理工具中自动显示。 其它的 `django.contrib` 应用程序，如 `django.contrib.redirects`，其它从网上下在的第三方 Django 应用程序一 样，都会自行添加到管理工具。

综上所述，管理工具其实就是一个 Django 应用程序，包含自己的模块、模板、视图和 URLpatterns。 你要像添加自己的视图一样，把它添加到 URLconf 里面。 你可以在 Django 基本代码中的 `django/contrib/admin` 目录下，检查它的模板、视图和 URLpatterns，但你不要尝试直接修改其中的任何代码，因为里面有很多地方可以让你自定义管理工具的工作方式。（如果你确实想浏览 Django 管理工具的代码，请谨记它在读取关于模块的元数据过程中做了些不简单的工作，因此最好花些时间阅读和理解那些代码。）

为什么在管理界面修改添加出版社时，**url** 报错

设置字段可选

在摆弄了一会之后，你或许会发现管理工具有个限制：编辑表单需要你填写每一个字段，然而在有些情况下， 你想要某些字段是可选的。 举个例子，我们想要 `Author` 模 块中的 `email` 字段成为可 选，即允许不填。 在现实世界中，你可能没有为每个作者登记邮箱地址。

为了指定 `email` 字段为可选，你只要编辑 `Book` 模块（回想第五章，它在 `mysite/books/models.py` 文件里），在 `email` 字段上加上 `blank=True`。代码如下：

设置日期型和数字型字段可选

虽然 `blank=True` 同 样适用于日期型和数字型字段，但是这里需要详细讲解一些背景知识。

SQL 有指定空值的独特方式，它把空值叫做 NULL。NULL 可以表示为未知的、非法的、或其它程序指 定的含义。

在 SQL 中， NULL 的值不同于空字符串，就像 Python 中 None 不同于空字符串（""）一样。这意味着某个字符型字段（如 VARCHAR）的值不可能同时包含 NULL 和空字符串。

这会引起不必要的歧义或疑惑。 为什么这条记录有个 NULL，而那条记录却有个空字符串？ 它们之间有区别，还是数据输入不一致？ 还有： 我怎样才能得到全部拥有空值的记录，应该按 NULL 和空字符串查找么？还是仅按字符串查找？

为了消除歧义，Django 生成 CREATE TABLE 语句自动为每个字段显 式加上 NOT NULL。 这里有个第五章中生成 Author 模块的例子：

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
)
;
```

在大多数情况下，这种默认的行为对你的应用程序来说是最佳的，因为它可以使你不再因数据一致性而头痛。 而且它可以和 Django 的其它部分工作得很好。如在管理工具中，如果你留空一个字符型字段，它会为此插入一个空字符串（而* 不是*NULL）。

但是，其它数据类型有例外：日期型、时间型和数字型字段不接受空字符串。 如果你尝试将一个空字符串插入日期型或整数型字段，你可能会得到数据库返回的错误，这取决于那个数据库的类型。 （PostgreSQL 比较严禁，会抛出一个异常；MySQL 可能会也可能不会接受，这取决于你使用的版本和运气了。）在这种情况下，NULL 是唯一指定空值的方法。 在 Django 模块中，你可以通过添加 null=True 来 指定一个字段允许为 NULL。

因此，这说起来有点复杂： 如果你想允许一个日期型（DateField、TimeField、DateTimeField）或数字型（IntegerField、DecimalField、FloatField）字段为空，你 需要使用 null=True * 和* blank=True。

为了举例说明，让我们把 Book 模块修改成允许 publication_date 为 空。修改后的代码如下：

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

添加 null=True 比 添加 blank=True 复 杂。因为 null=True 改 变了数据的语义，即改变了 CREATE TABLE 语句，把 publication_date 字段上的 NOT NULL 删 除了。 要完成这些改动，我们还需要更新数据库。

出于某种原因，Django 不会尝试自动更新数据库结构。所以你必须执行 ALTER TABLE 语句将模块的改动更新至数据库。 像先前那样，你可以使用 manage.py dbshell 进入数据库服务环境。 以下是在这个特殊情况下如何删除 NOT NULL:

```
ALTER TABLE books_book ALTER COLUMN publication_date DROP NOT NULL;
```

（注意：以下 SQL 语法是 PostgreSQL 特有的。）

我们将在第十章详细讲述数据库结构更改。

现在让我们回到管理工具，添加 book 的编辑页面允许输入一个空的 publication date。

自定义字段标签

在编辑页面中，每个字段的标签都是从模块的字段名称生成的。 规则很简单： 用空格替换下划线；首字母大写。例如：`Book` 模 块中 `publication_date` 的 标签是 Publication date。

然而，字段名称并不总是贴切的。有些情况下，你可能想自定义一个标签。 你只需在模块中指定 `verbose_name`。

举个例子，说明如何将 `Author.email` 的标签改为 e-mail，中间有个横线。

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, **verbose_name='e-mail'** )
```

修改后重启服务器，你会在 author 编辑页面中看到这个新标签。

请注意，你不必把 `verbose_name` 的首字母大写，除非是连续大写（如：`"USA state"`）。 Django 会自动适时将首字母大写，并且在其它不需要大写的地方使用 `verbose_name` 的精确值。

最后还需注意的是，为了使语法简洁，你可以把它当作固定位置的参数传递。 这个例子与上面那个的效果相同。

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(**'e-mail',**  blank=True)
```

但这不适用于 `ManyToManyField` 和 `ForeignKey` 字段，因为它们第一个参数必须是模块类。 那种情形，必须显式使用 `verbose_name` 这个参数名 称。

自定义 *ModelAdmi* 类

迄今为止，我们做的 `blank=True`、`null=True` 和 `verbose_name` 修改其实是模块级别，而不是管理级别的。 也就是说，这些修改实质上是构成模块的一部分，并且正好被管理工具使用，而不是专门针对管理工具的。

除了这些，Django 还提供了大量选项让你针对特别的模块自定义管理工具。 这些选项都在 *ModelAdmin classes* 里面，这些类包含了管理工具中针对特别模块的配置。

我们可以在这基础上改进，添加其它字段，从而改变列表的显示。 这个页面应该提供便利，比如说：在这个列表中可以看到作者的邮箱地址。如果能按照姓氏或名字来排序，那就更好了。

为了达到这个目的，我们将为 `Author` 模块定义一个 `ModelAdmin` 类。 这个类是自定义管理工具的关键，其中最基本的一件事情是允许你指定列表中的字段。 打开 `admin.py` 并修改：

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

**class AuthorAdmin(admin.ModelAdmin):**
    **list_display = ('first_name', 'last_name', 'email')**
```

```
admin.site.register(Publisher)
**admin.site.register(Author, AuthorAdmin)**
admin.site.register(Book)
```

解释一下代码：

> 我们新建了一个类 AuthorAdmin，它是从 django.contrib.admin.ModelAdmin 派生出来的子类，保存着一个类的自 定义配置，以供管理工具使用。 我们只自定义了一项：list_display， 它是一个字段名称的元组，用于列表显示。 当然，这些字段名称必须是模块中有的。
>
> 我们修改了 admin.site.register() 调用，在 Author 后面添加了 AuthorAdmin。你可以这样理解： 用 AuthorAdmin 选项注册 Author 模块。
>
> admin.site.register() 函 数接受一个 ModelAdmin 子 类作为第二个参数。 如果你忽略第二个参数，Django 将使用默认的选项。Publisher 和 Book 的注册就属于这种情况。

接下来，让我们添加一个快速查询栏。 向 AuthorAdmin 追加 search_fields，如：

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    **search_fields = ('first_name', 'last_name')**
```

刷新浏览器，你会在页面顶端看到一个查询栏。 （见图 6-9.）我们刚才所作的修改列表页面，添加了一个根据姓名查询的查询框。 正如用户所希望的那样，它是大小写敏感，并且对两个字段检索的查询框。如果查询"bar"，那么名字中含有 Barney 和姓氏中含有 Hobarson 的作者记录将被检索出来。

接下来，让我们为 Book 列表页添加一些过滤器。

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')

**class BookAdmin(admin.ModelAdmin):**
    **list_display = ('title', 'publisher', 'publication_date')**
    **list_filter = ('publication_date',)**

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
**admin.site.register(Book, BookAdmin)**
```

另外一种过滤日期的方式是使用 date_hierarchy 选项，如：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    **date_hierarchy = 'publication_date'**
```

最后，让我们改变默认的排序方式，按 publication date 降序排列。 列表页面默认按照模块 class Meta（详见第五章）中的 ordering 所指的列排序。但目前没有指定 ordering 值，所以当前排序是没有定义的。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    **ordering = ('-publication_date',)**
```

自定义编辑表单

正如自定义列表那样，编辑表单多方面也能自定义。

首先，我们先自定义字段顺序。 默认地，表单中的字段顺序是与模块中定义是一致的。 我们可以通过使用 `ModelAdmin` 子 类中的 `fields` 选项来改 变它：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **fields = ('title', 'authors', 'publisher', 'publication_date')**
```

完成之后，编辑表单将按照指定的顺序显示各字段。 它看起来自然多了——作者排在书名之后。 字段顺序当然是与数据条目录入顺序有关， 每个表单都不一样。

通过 fields 这个选项，你可以排除一些不想被其他人编辑的 fields 只要不选上不想被编辑的 field(s)即可。 当你的 admin 用户只是被信任可以更改你的某一部分数据时，或者，你的数据被一些外部的程序自动处理而改变了了，你就可以用这个功能。 例如，在 book 数据库中，我们可以隐藏 `publication_date`， 以防止它被编辑。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **fields = ('title', 'authors', 'publisher')**
```

这样，在编辑页面就无法对 publication date 进行改动。 如果你是一个编辑，不希望作者推迟出版日期的话，这个功能就很有用。 （当然，这纯粹是一个假设的例子。）

当一个用户用这个不包含完整信息的表单添加一本新书时，Django 会简单地将 `publication_date` 设 置为 `None`，以确保这个字 段满足 `null=True` 的 条件。

另一个常用的编辑页面自定义是针对多对多字段的。 真如我们在 book 编辑页面看到的那样，`` 多对多字段`` 被展现成多选框。虽然多选框在逻辑上是最适合的 HTML 控件，但它却不那么好用。 如果你想选择多项，你必须还要按下 Ctrl 键（苹果机是 command 键）。 虽然管理工具因此添加了注释（help_text），但是当它有几百个选项时，它依然显得笨拙。

更好的办法是使用 `filter_horizontal`。让我们把它添加到 `BookAdmin` 中，然后看看它的效果。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **filter_horizontal = ('authors',)**
```

（如果你一着跟着做练习，请注意移除 `fields` 选项，以使得编辑页面包含所有字段。）

`ModelAdmin` 类还支 持 `filter_vertical` 选 项。 它像 `filter_horizontal` 那 样工作，除了控件都是垂直排列，而不是水平排列的。 至于使用哪个，只是个人喜好问题。

解决这个问题的办法是使用`` raw_id_fields`` 选项。它是一个包含外键字段名称的元组，它包含的字段将被展现成`` 文本框`` ，而不再是`` 下拉框`` 。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
    **raw_id_fields = ('publisher',)**
```

用户、用户组和权限

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。 我们在本章的前面，浏览用户和用户组区域的时候已经见过这些了。 如你所想，用户对象有标准的用户名、密码、邮箱地址和真实姓名，同时它还有关于使用管理界面的权限定义。 首先，这有一组三个布尔型标记：

- 活动标志，它用来控制用户是否已 经激活。 如果一个用户帐号的这个标记是关闭状态，而用户又尝试用它登录时，即使密码正确，他也无法登录系统。
- 成员标志，它用来控制这个用户是 否可以登录管理界面（即：这个用户是不是你们组织里的成员） 由于用户系统可以被用于控制公众页面（即：非管理页面）的访问权限（详见第十四章），这个标志可用来区分公众用户和管理用户。
- 超级用户标志，它赋予用户在管理 界面中添加、修改和删除任何项目的权限。 如果一个用户帐号有这个标志，那么所有权限设置（即使没有）都会被忽略。

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。 管理界面中每种可编辑的对象（如：books、authors、publishers）都有三种权限： 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。

当你创建一个用户时，它没有任何权限，该有什么权限是由你决定的。 例如，你可以给一个用户添加和修改 publishers 的权限，而不给他删除的权限。 请注意，这些权限是定义在模块级别上，而不是对象级别上的。据个例子，你可以让小强修改任何图书，但是不能让他仅修改由机械工业出版社出版的图书。 后面这种基于对象级别的权限设置比较复杂，并且超出了本书的覆盖范围，但你可以在 Django documentation 中寻找答案。

注释

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。 赋予一个用户修改用户的权限，本质上说就是把他变成一个超级用户。

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。

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。
- 记者检查管理界面，尽早指出缺少 或多余的字段。 开发者来回地修改模块。
- 当模块认可后，记者就开始用管理 界面输入数据。 同时，程序员可以专注于开发公众访问视图和模板（有趣的部分）。

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：

- 检查模块*：当你定义好了若干个模块，在管理页面中把他们调出来然后输入一些虚假的数据，这是相当有用的。 有时候，它能显示数据建模的错误或者模块中其它问题。
- 管理既得数据*：如果你的应用程序依赖外部数据（来自用户输入或网络爬虫），管理界面提供了一个便捷的途径，让你检查和编辑那些数据。 你可以把它看作是一个功能不那么强大，但是很方便的数据库命令行工具。
- 临时的数据管理程序*：你可以用管理工具建立自己的轻量级数据管理程序，比如说开销记录。 如果你正在根据自己的，而不是公众的需要开发些什么，那么管理界面可以带给你很大的帮助。 从这个意义上讲，你可以把它看作是一个增强的关系型电子表格。

第 7 章 表单

从 Google 的简朴的单个搜索框，到常见的 Blog 评论提交表单，再到复杂的自定义数据输入接口，HTML 表单一直是交互性网站的支柱。 本章介绍如何用 Django 对用户通过表单提交的数据进行访问、有效性检查以及其它处理。 与此同时，我们将介绍 HttpRequest 对象和 Form 对象。

从 ***Request*** 对象中获取数据

我们在第三章讲述 View 的函数时已经介绍过 HttpRequest 对象了，但当时并没有讲太多。 让我们回忆下：每个 view 函数的第一个参数是一个 HttpRequest 对象，就像下面这个 hello()函数:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

HttpRequest 对象，比如上面代码里的 request 变量，会有一些有趣的、你必须让自己熟悉的 属性和方法，以便知道能拿它们来做些什么。 在 view 函数的执行过程中，你可以用这些属性来获取当前 request 的一些信息（比如，你正在加载这个页面的用户是谁，或者用的是什么浏览器）。

URL 相关信息

HttpRequest 对象包含当前请求 URL 的一些信息：

属性/方法	说明	举例
`request.path`	除域名以外的请求路径，以正斜杠开头	`"/hello/"`
`request.get_host()`	主机名（比如，通常所说的域名）	`"127.0.0.1:8000"` or `"www.example.com"`

属性/方法	说明	举例
`request.get_full_path()`	请求路径，可能包含查询字符串	`"/hello/?print=true"`
`request.is_secure()`	如果通过HTTPS访问，则此方法返回True， 否则返回False	`True` 或者 `False`

在view函数里，要始终用这个属性或方法来得到URL，而不要手动输入。 这会使得代码更加灵活，以便在其它地方重用。 下面是一个简单的例子：

```
# BAD!
def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")

# GOOD
def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

有关 **request** 的其它信息

request.META 是一个Python字典，包含了所有本次HTTP请求的Header信息，比如用户IP地址和用户Agent（通常是浏览器的名称和版本号）。 注意，Header信息的完整列表取决于用户所发送的Header信息和服务器端设置的Header信息。 这个字典中几个常见的键值有：

- `HTTP_REFERER`，进站前链 接网页，如果有的话。 （请注意，它是`REFERRER`的 笔误。）
- `HTTP_USER_AGENT`，用 户浏览器的user-agent字符串，如果有的话。 例如： `"Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17"` .
- `REMOTE_ADDR` 客户端IP，如：`"12.345.67.89"` 。(如果申请是经过代理服务器的话，那么它可能是以逗号分割的多个IP地址，如：`"12.345.67.89,23.456.78.90"` 。)

```
# GOOD (VERSION 1)
def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 2)
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

我们鼓励你动手写一个简单的view函数来显示 request.META 的所有数据，这样你就知道里面有什么了。 这个view函数可能是这样的：

```
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

做为一个练习，看你自己能不能把上面这个 view 函数改用 Django 模板系统来实现，而不是上面这样 来手动输入 HTML 代码。 也可以试着把前面提到的 request.path 方法或 HttpRequest 对象的其它方法加进去。

提交的数据信息

除了基本的元数据，HttpRequest 对象还有两个属性包含了用户所提交的信息： request.GET 和 request.POST。二者都是类字典对象，你可以通过它们来访问 GET 和 POST 数据。

类字典对象

我们说"request.GET 和 request.POST 是类字典对象"，意思是他们的行为像 Python 里标准的字典对象，但在技术底层上他们不是标准字典对象。 比如说，request.GET 和 request.POST 都有 get()、keys()和 values()方法，你可以用用 for key in request.GET 获取所有的键。

一个简单的表单处理示例

继续本书一直进行的关于书籍、作者、出版社的例子，我们现在来创建一个简单的 view 函数以便让用户可 以通过书名从数据库中查找书籍。

通常，表单开发分为两个部分： 前端 HTML 页面用户接口和后台 view 函数对所提交数据的处理过程。 第一部分很简单；现在我们来建立个 view 来显示一个搜索表单：

```
from django.shortcuts import render_to_response

def search_form(request):
```

在第三章已经学过，这个 view 函数可以放到 Python 的搜索路径的任何位置。 为了便于讨论，咱们将它放在 books/views.py 里。

这个 search_form.html 模板，可能看起来是这样的：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

而 urls.py 中的 URLpattern 可能是这样的：

```
from mysite.books import views

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    # ...
)
```

（注意，我们直接将 views 模块 import 进来了，而不是用类似 from mysite.views import search_form 这样的语句，因为前者看起来更简洁。 我们将在第 8 章讲述更多的关于 import 的

用法。）

现在，如果你运行 `runserver` 命令，然后访问 `http://127.0.0.1:8000/search-form/`，你会看到搜索界面。 非常简单。

不过，当你通过这个 form 提交数据时，你会得到一个 Django 404 错误。 这个 Form 指向的 URL `/search/` 还没有被实现。 让我们添加第二个视图函数并设置 URL：

urls.py

```
urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    (r'^search/$', views.search),
    # ...
)
```

```
#!/usr/bin/env python

# -*- coding: UTF-8 -*-

#views.py

# Create your views here.

from django.http import HttpResponse,Http404

from django.shortcuts import render_to_response



def search_form(request):

  return render_to_response('search_form.html')#查找模板文件在 templates 里面



def search(request):

  if 'q' in request.GET and request.GET['q']:

    message = 'You searched for: %r' % request.GET['q']

  else:

    message = 'You submitted an empty form.'

  return HttpResponse(message)
```

因为使用 GET 方法的数据是通过查询字符串的方式传递的（例如/search/?q=django）， 所以我们可以使用 requet.GET 来获取这些数据。 第三章介绍 Django 的 URLconf 系统时我们比较了 Django 的简洁的 URL 与 PHP/Java 传统的 URL，我们提到将在第七章讲述如何使用传 统的

URL。通过刚才的介绍，我们知道在视图里可以使用 request.GET 来获取传统 URL 里的查询字符串（例如 hours=3）。

获取使用 POST 方法的数据与 GET 的相似，只是使用 request.POST 代替了 request.GET。那么，POST 与 GET 之间有什么不同？当我们提交表单仅仅需要获取数据时就可以用 GET； 而当我们提交表单时需要更改服务器数据的状态，或者说发送 e-mail，或者其他不仅仅是获取并显示数据的时候就使用 POST。 在这个搜索书籍的例子里，我们使用 GET，因为这个查询不会更改服务器数据的状态。 (如果你有兴趣了解更多关于 GET 和 POST 的知识，可以参见 http://www.w3.org/2001/tag/doc/whenToUseGet.html。)

既然已经确认用户所提交的数据是有效的，那么接下来就可以从数据库中查询这个有效的数据（同样，在 **views.py** 里操作）：

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

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render_to_response('search_results.html',
            {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

改进表单

同上一章一样，我们先从最为简单、有效的例子开始。 现在我们再来找出这个简单的例子中的不足，然后改进他们。

首先，search()视图对于空字符串的处理相当薄弱——仅显示一条"Please submit a search term."的提示信息。 若用户要重新填写表单必须自行点击"后退"按钮， 这种做法既糟糕又不专业。如果在现实的案例中，我们这样子编写，那么 Django 的优势将荡然无存。

在检测到空字符串时更好的解决方法是重新显示表单，并在表单上面给出错误提示以便用户立刻重新填写。 最简单的实现方法既是添加 else 分句重新显示表单，代码如下：

```
def search(request):

  if 'q' in request.GET and request.GET['q']:

    q=request.GET['q']

    books=Book.objects.filter(title__icontains=q)

    return render_to_response('search_result.html',{'books':books,'query':q})

  else:

#    return HttpResponse('Please submit a search term.')

    return render_to_response('search_form.html',{'error':True})
```

这段代码里，我们改进来 search()视图：在字符串为空时重新显示 search_form.html。 并且给这个模板传递了一个变量 error，记录着错误提示信息。 现在我们编辑一下 search_form.html，检测变量 error：

```
<html>

<head>

   <title>Search Book</title>

</head>

<body>

   {% if error %}

      <p style='color:red;'>Please submit a search term.</p>

   {% endif %}

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

      <input type="text" name="q">

      <input type="submit" value="Search">

   </form>

</body>

</html>
```

我们修改了 **search_form()**视图所使用的模板，因为 **search_form()**视图没有传递 **error** 变量，所以在条用 **search_form** 视图时不会显示错误信息。

通过上面的一些修改，现在程序变的好多了，但是现在出现一个问题： 是否有必要专门编写 search_form()来显示表单？ 按实际情况来说，当一个请求发送至/search/（未包含 GET 的数据）后将会显示一个空的表单（带有错误信息）。 所以，只要我们改变 search()视图：当用户访问/search/并未提交任何数据时就隐藏错误信息，这样就移去 search_form()视图以及 对应的 URLpattern。

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {'error': error})
```

在改进后的视图中，若用户访问/search/并且没有带有 GET 数据，那么他将看到一个没有错误信息 的表单； 如果用户提交了一个空表单，那么它将看到错误提示信息，还有表单； 最后，若用户提交了一个非空的值，那么他将看到搜索结果。

最后，我们再稍微改进一下这个表单，去掉冗余的部分。 既然已经将两个视图与 URLs 合并起来，/search/视图管理着表单的显示以及结果的显示，那么在 search_form.html 里表单的 action 值就没有必要硬编码的指定 URL。 原先的代码是这样：

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

现在改成这样：

```
<form action="" method="get">
```

action="“意味着表单将提交给与当前页面相同的 URL。 这样修改之后，如果 `search()`视图不指向其它页面 的话，你将不必再修改 `action`。

简单的验证

我们的搜索示例仍然相当地简单，特别从数据验证方面来讲；我们仅仅只验证搜索关键值是否为空。 然后许多 HTML 表单包含着比检测值是否为空更为复杂的验证。 我们都有在网站上见过类似以下的错误提示信息：

- 请输入一个有效的 email 地址， foo’ 并不是一个有效的 e-mail 地址。
- 请输入 5 位数的 U.S 邮政编码， 123 并非是一个有效的邮政编码。
- 请输入 YYYY-MM-DD 格式的日 期。
- 请输入 8 位数以上并至少包含一个数字 的密码。

关于 JavaScript 验证

可以使用 Javascript 在客户端浏览器里对数据进行验证，这些知识已超出本书范围。 要注意：即使在客户端已经做了验证，但是服务器端仍必须再验证一次。 因为有些用户会将 JavaScript 关闭掉，并且还有一些怀有恶意的用户会尝试提交非法的数据来探测是否有可以攻击的机会。

除了在服务器端对用户提交的数据进行验证（例如在视图里验证），我们没有其他办法。 JavaScript 验证可以看作是额外的功能，但不能作为唯一的验证功能。

我们来调整一下 search()视图，让她能够验证搜索关键词是否小于或等于 20 个字符。 （为来让例子更为显著，我们假设如果关键词超过 20 个字符将导致查询十分缓慢）。那么该如何实现呢？ 最简单的方式就是将逻辑处理直接嵌入到视图里，就像这样：

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        **elif len(q) > 20:**
            **error = True**
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
```

```
        {'error': error})
```

现在，如果尝试着提交一个超过 20 个字符的搜索关键词，系统不会执行搜索操作，而是显示一条错误提示 信息。 但是，search_form.html 里的这条提示信息是：”Please submit a search term.”，这显然是错误的， 所以我们需要更精确的提示信息：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term 20 characters or
shorter.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

但像这样修改之后仍有一些问题。 我们包含万象的提示信息很容易使人产生困惑： 提交一个空表单怎么会出现一个关于 20 个字符限制的提示？ 所以，提示信息必须是详细的，明确的，不会产生疑议。

问题的实质在于我们只使用来一个布尔类型的变量来检测是否出错，而不是使用一个列表来记录相应的错误 信息。 我们需要做如下的调整：

```
def search(request):
    **errors = []**
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            **errors.append('Enter a search term.')**
        elif len(q) > 20:
            **errors.append('Please enter at most 20 characters.')**
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {**'errors': errors** })
```

接着，我们要修改一下 search_form.html 模板，现在需要显示一个 errors 列表而不 是一个布尔判断。

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    **{% if errors %}**
        **<ul>**
            **{% for error in errors %}**
            **<li>{{ error }}</li>**
            **{% endfor %}**
```

```
        **</ul>**
    **{% endif %}**
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

使用 **south** 代替 **syncdb** 在 **settings** 文件 **INSTALLED_APPS = (**

'south',

)

The First Migration

python manage.py schemamigration appname –initial

python manage.py migrate appname

Changing the model

manage.py schemamigration appname --auto

第一个 Form 类

Django 带有一个 form 库，称为 django.forms，这个库可以处理我们本章所提到的包 括 HTML 表单显示以及验证。 接下来我们来深入了解一下 form 库，并使用她来重写 contact 表单应用。

表单框架最主要的用法是，为每一个将要处理的 HTML 的`` <Form>`` 定义一个 `Form` 类。 在这个例子中，我们只有一个`` <Form>`` ，因此我们只需定义一个 `Form` 类。 这个类可以存在于任何地方，甚至直接写在`` views.py`` 文件里也行，但是社区的惯例是把 `Form` 类 都放到一个文件中：`forms.py`。 在存放`` views.py`` 的目录中，创建这个文件，然后输入：

```
from django import forms

class ContactForm(forms.Form):

   subject = forms.CharField()

   email = forms.EmailField(required=False)

   message = forms.CharField()
```

让我们钻研到 Python 解释器里面看看这个类做了些什么。 它做的第一件事是将自己显示成 HTML：

```
>>> from books.forms import ContactForm

>>> f=ContactForm()

>>> print f

<tr><th><label for="id_subject">Subject:</label></th><td><input type="text" name="subject" id="id_subject" /></td></tr>
```

<tr><th><label for="id_email">Email:</label></th><td><input type="text" name="email" id="id_email" /></td></tr>

<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_message" /></td></tr>

为了便于访问，Django 用`` <label>`` 标志，为每一个字段添加了标签。 这个做法使默认行为尽可能合适。

默认输出按照 HTML 的<`` table`` >格式，另外有一些其它格式的输出：

```
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subject" /></li>
<li><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" /></li>
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></li>
>>> print f.as_p()
<p><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subject" /></p>
<p><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" /></p>
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></p>
```

请注意，标签<table>、<ul>、<form>的开闭合 标记没有包含于输出当中，这样你就可以添加额外的行或者自定义格式。

这些类方法只是一般情况下用于快捷显示完整表单的方法。 你同样可以用 HTML 显示个别字段：

```
>>> print f['subject']
<input type="text" name="subject" id="id_subject" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
```

`Form` 对 象做的第二件事是校验数据。 为了校验数据，我们创建一个新的对 `Form` 象，并且传入一个与定义匹配的字典类型数据：

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com', 'message': 'Nice site!'})
```

一旦你对一个 `Form` 实体赋值，你就得到了一个绑定 form：

```
>>> f.is_bound
True
```

调用任何绑定 form 的 `is_valid()` 方法，就可以知道它的数据是否合法。 我们已经为每个字段传入了值，因此整个 `Form` 是合法的：

```
>>> f.is_valid()
True
```

如果我们不传入 email 值，它依然是合法的。因为我们指定这个字段的属性 required=False：

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
>>> f.is_valid()
True
```

但是，如果留空 subject 或 message，整个 Form 就不再合法了：

```
>>> f = ContactForm({'subject': 'Hello'})
>>> f.is_valid()
False
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.is_valid()
False
```

你可以逐一查看每个字段的出错消息：

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f['message'].errors
[u'This field is required.']
>>> f['subject'].errors
[]
>>> f['email'].errors
[]
```

每一个邦定 Form 实体都有一个 errors 属性，它为你提供了一个字段与错误消息相映射的字典表。

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.errors
{'message': [u'This field is required.']}
```

最终，如果一个 Form 实体的数据是合法的，它就会有一个可用的 cleaned_data 属性。 这是一个包含干净的提交数据的字典。 Django 的 form 框架不但校验数据，它还会把它们转换成相应的 Python 类型数据，这叫做清理数据。

```
>>> f = ContactForm({subject': Hello, email: adrian@example.com, message: Nice
site!})
>>> f.is_valid()
True
>>> f.cleaned_data
{message': uNice site!, email: uadrian@example.com, subject: uHello}
```

我们的 contact form 只涉及字符串类型，它们会被清理成 Unicode 对象。如果我们使用整数型或日期型，form 框架会确保方法使用合适的 Python 整数型或 datetime.date 型对 象。

在视图中使用 *Form* 对象

在学习了关于 Form 类的基本知识后，你会看到我们如何把它用到视图中，取代 contact()代码中不整齐的部 分。 一下示例说明了我们如何用 forms 框架重写 contact()：

import pdb

from mysite.books.forms import ContactForm

```
def contact(request):

#  pdb.set_trace()

  if request.method=='POST':

    form=ContactForm(request.POST)

    if form.is_valid():

      ct=form.cleaned_data

      send_mail(

       ct['subject'],

       ct['message'],

       ct.get('email','noreply@example.com'),

       ['siteowner@example.com'],

)

      return HttpResponseRedirect('/contact/thanks/')

  else:

    form=ContactForm()

  return render_to_response('contact_form.html',{'form':form})

<html>
```

```
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form action="/contact/" method="post">
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Submit">
```

</form>

</body>

</html>

改变字段显示

你可能首先注意到：当你在本地显示这个表单的时，message 字段被显示成`` input type="text"`` ，而它应该被显示成<`` textarea`` >。我们可以通过设置* widget* 来修改它：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField(**widget=forms.Textarea** )
```

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。

考虑一下 Field 类表现* 校验逻辑* ，而部件表现* 显示逻辑* 。

设置最大长度

一个最经常使用的校验要求是检查字段长度。 另外，我们应该改进 ContactForm，使 subject 限制在 100 个字符以 内。 为此，仅需为 CharField 提 供 max_length 参 数，像这样：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(**max_length=100** )
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

选项 min_length 参数同样可用。

设置初始值

让我们再改进一下这个表单：为字 subject 段添加* 初始值* ： "I love your site!" （一点建议，但没坏处。）为此，我们可以在创建 Form 实体时，使用 initial 参数：

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
```

```
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', `'noreply@example.com`_'),
                [`'siteowner@example.com`_'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm(
            **initial={'subject': 'I love your site!'}**
        )
    return render_to_response('contact_form.html', {'form': form})
```

现在，`subject` 字段将被那个句子填充。

请注意，传入* 初始值* 数据和传入数据以* 绑定* 表单是有区别的。 最大的区别是，如果仅传入* 初始值* 数据，表单是 *unbound* 的，那意味着它没有错误消息。

自定义校验规则

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。 这里有一个问题： 一些提交的消息只有一两个字，我们无法得知详细的信息。 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()`` 方法到`` Form`` 类：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Django 的 form 系统自动寻找匹配的函数方法，该方法名称以 `clean_` 开头，并以字段名称结束。 如果有这样的方法，它将在校验时被调用。

特别地，`clean_message()` 方法将在指定字段的默认校验逻辑执行* 之后* 被调用。（本例中，在必填 `CharField` 这 个校验逻辑之后。）因为字段数据已经被部分处理，所以它被从 `self.cleaned_data` 中提取出来了。同样，我们不必担心数据是否为空，因为它已经被校验过了。

我们简单地使用了 len()和 split()的组合来计算单词的数量。 如果用户输入字数不足，我们抛出一个 `forms.ValidationError` 型 异常。这个异常的描述会被作为错误列表中的一项显示给用户。

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。

指定标签

HTML 表单中自动生成的标签默认是按照规则生成的：用空格代替下划线，首字母大写。如 `email` 的标签是`"Email"` 。（好像在哪听到过？ 是的，同样的逻辑被用于模块（model）中字段的 `verbose_name` 值。 我们在第五章谈到过。）

像在模块中做过的那样，我们同样可以自定义字段的标签。 仅需使用 `label`，像这样：

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, **label='Your e-mail address'** )
    message = forms.CharField(widget=forms.Textarea)
```

定制 *Form* 设计

在上面的`` contact_form.html`` 模板中我们使用`` {{form.as_table}}`` 显示表单，不过我们可以使用其他更精确控制表单显示的方法。

修改 form 的显示的最快捷的方式是使用 CSS。 尤其是错误列表，可以增强视觉效果。自动生成的错误列表精确的使用`` <ul class=”errorlist”>``，这样，我们就可以针对它们使用 CSS。 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，自动生成 HTML 是很方便的，但是在某些时候，你会想覆盖默认的显示。 {{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。

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
```

```
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
        <div class="field">
            {{ form.email.errors }}
            <label for="id_email">Your e-mail address:</label>
            {{ form.email }}
        </div>
        <div class="field">
            {{ form.message.errors }}
            <label for="id_message">Message:</label>
            {{ form.message }}
        </div>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

`{{ 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， 例如：

```
<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ul>
        {% for error in form.message.errors %}
            <li><strong>{{ error }}</strong></li>
        {% endfor %}
        </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ 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。

第八章：高级视图和 **URL** 配置

在第三章，我们已经对基本的 Django 试图和 URL 配置做了介绍。 在这一章，将进一步说明框架中这两个部分的高级机能。

URLconf 技巧

URLconf 没什么特别的，就象 Django 中其它东西一样，它们只是 Python 代码。 你可以在几方面从中得到好处，正如下面所描述的。

流线型化(Streamlining)函数导入

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 hello, current_datetime, hours_ahead

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

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。

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

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

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'^hello/$', **'mysite.views.hello'** ),
    (r'^time/$', **'mysite.views.current_datetime'** ),
    (r'^time/plus/(d{1,2})/$', **'mysite.views.hours_ahead'** ),
)
```

(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'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'mysite.views.hours_ahead'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

新的:

from django.conf.urls.defaults import *

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

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

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)
    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')
```

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),
)
```

下面是相同的 URLconf，使用命名组进行了重写:

```
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。

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：

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。 视图函数对待它就像另一个参数。

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。

伪造捕捉到的 **URLconf** 值

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。

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),
)
```

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。 `` month`` 和 `` day`` 参数来自 URL 捕捉或是指定参数都可以。

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：
- 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。

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 额外的选项

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 *
from mysite import views

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。

例子：

```
# urls.py

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

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。

```
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 。

比方说，你可以考虑通过下面这个 URLpattern 所描述的方式来向 Django 的管理站点添加一 个目标页面

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', views.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/$', views.user_add_stage),
    ('^([^/]+)/([^/]+)/add/$', views.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。

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。 许多内置 Python 函数是挑剔的（这是理所当然的）只接受特定类型的对象。 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 *
from mysite import views

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))
```

视图函数的高级概念

说到关于请求方法的分支，让我们来看一下可以用什么好的方法来实现它。 考虑这个URLconf/view 设计:

```
# urls.py

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

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.some_page),
    # ...
)

# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def some_page(request):
    if request.method == 'POST':
        do_something_for_post()
        return HttpResponseRedirect('/someurl/')
    elif request.method == 'GET':
        do_something_for_get()
        return render_to_response('page.html')
    else:
        raise Http404()
```

在这个示例中，`` some_page()`` 视图函数对`` POST`` 和`` GET`` 这两种请求方法的处理完全不同。 它们唯一的共同点是共享一个 URL 地址： `` /somepage/.``正如大家所看到的，在同一个视图函数中对`` POST`` 和`` GET`` 进行处理是一种很初级也很粗糙的做法。 一个比较好的设计习惯应该是，用两个分开的视图函数——一个处理`` POST`` 请求，另一个处理`` GET`` 请求，然后在相应的地方分别进行调用。

我们可以像这样做：先写一个视图函数然后由它来具体分派其它的视图，在之前或之后可以执行一些我们自 定的程序逻辑。 下边的示例展示了这个技术是如何帮我们改进前边那个简单的`` some_page()`` 视图的：

```
# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def method_splitter(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404

def some_page_get(request):
```

```
    assert request.method == 'GET'
    do_something_for_get()
    return render_to_response('page.html')

def some_page_post(request):
    assert request.method == 'POST'
    do_something_for_post()
    return HttpResponseRedirect('/someurl/')

# urls.py

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

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.method_splitter, {'GET': views.some_page_get, 'POST':
views.some_page_post}),
    # ...
)
```

让我们从头看一下代码是如何工作的：

> 我们写了一个新的视图，`` method_splitter()`` ，它根据`` request.method`` 返回的值来调用相应的视图。可以看到它带有两个关键参数，`` GET`` 和`` POST`` ，也许应该是* 视图函数* 。如果`` request.method`` 返回`` GET`` ，那它就会自动调用`` GET`` 视图。 如果`` request.method`` 返回的是`` POST`` ，那它调用的就是`` POST`` 视图。 如果`` request.method`` 返回的是其它值（如：`` HEAD`` ），或者是没有把`` GET`` 或`` POST`` 提交给此函数，那它就会抛出一个`` Http404`` 错误。
>
> 在 URLconf 中，我们把`` /somepage/`` 指到`` method_splitter()`` 函数，并把视图函数额外需要用到的`` GET`` 和`` POST`` 参数传递给它。

然而，当我们做到这一步时，我们仍然可以改进`` method_splitter`` 。从代码我们可以看到，它假设`` Get`` 和`` POST`` 视图除了`` request`` 之外不需要任何其他的参数。那么，假如我们想要使用`` method_splitter`` 与那种会从 URL 里捕捉字符，或者会接收一些可选参数的视图一起工作时该怎么办呢？

为了实现这个，我们可以使用 Python 中一个优雅的特性 带星号的可变参数 我们先展示这些例子，接着再进行解释

```
def method_splitter(request, *args, **kwargs):
    get_view = kwargs.pop('GET', None)
    post_view = kwargs.pop('POST', None)
    if request.method == 'GET' and get_view is not None:
        return get_view(request, *args, **kwargs)
    elif request.method == 'POST' and post_view is not None:
        return post_view(request, *args, **kwargs)
    raise Http404
```

这里,我们重构 method_splitter(),去掉了 GET 和 POST 两个关键字参数,改而支 持使用*args 和和**kwargs(注意*号) 这是一个 Python 特性，允许函数接受动态的、可变数量的、参数名只在运行时可知的参数。 如果你在函数定义时,只在参数前面加一个*号,所有传递给函数的参数将会保存为一个元组. 如果你在函数定义时,在参数前面加两个*号,所有传递给函数的关键字参数,将会保存为一个字典

包装视图函数

我们最终的视图技巧利用了一个高级 python 技术。 假设你发现自己在各个不同视图里重复了大量代码，

这里，每一个视图开始都检查 `request.user` 是否是已经认证的，是的话，当前用户已经成功登陆站点否则就重定向`/accounts/login/` (注意,虽然我们还没有讲到 request.user,但是 14 章将要讲到它.就如你所想像的,request.user 描述当前用户是登陆的还是匿名)

如果我们能够从每个视图里移除那些 重复代，并且只在需要认证的时候指明它们，那就完美了。 我们能够通过使用一个视图包装达到目的。 花点时间来看看这个：

```
def requires_login(view):
    def new_view(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return new_view
```

函数 requires_login,传入一个视图函数 view,然后返回一个新的视图函数 new_view.这个新的视图函数 new_view 在函数 requires_login 内定义 处理 request.user.is_authenticated()这个验证,从而决定是否执行原来的 view 函数

现在,我们可以从 views 中去掉 if not request.user.is_authenticated()验证.我们可以在 URLconf 中很容易的用 requires_login 来包装实现.

```
from django.conf.urls.defaults import *
from mysite.views import requires_login, my_view1, my_view2, my_view3

urlpatterns = patterns('',
    (r'^view1/$', requires_login(my_view1)),
    (r'^view2/$', requires_login(my_view2)),
    (r'^view3/$', requires_login(my_view3)),
)
```

优化后的代码和前面的功能一样,但是减少了代码冗余 现在我们建立了一个漂亮,通用的函数 requires_login()来帮助我们修饰所有需要它来验证的视图

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'),
)
```

在前面第 6 章介绍 Django 的 admin 模块时我们曾经见过 include. admin 模块有他自己的 URLconf,你仅仅只需要在你自己的代码中加入 include 就可以了.

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 /(包含两个斜杠) 在第一个 URLconf 中,r'^weblog/'匹配 因为它有一个 include(),django 去掉了匹配的部,在这个例子中匹配的部分是'weblog/' 剩下的部分是/2007/ (最前面有一个斜杠),不匹配 mysite.blog.urls 中的任何一行.
- `/about/` : 这个匹配第一个 URLconf 中的 `mysite.views.about` 视图。

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}),
)
```

第九章 模板高级进阶

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。 它也包含一个自动转意特征，当你继续使用 django 的时候随着时间推移你一定会注意这个安全考虑。

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

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。 模板可以包含模板标签和变量。

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。 这个定义故意搞得模糊不清。 例如，一个模版标签能够产生作为控制结构的内容（一个 iffor 循环),

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。

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： 例如，考虑这两个视图：

```
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)
```

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： 例如，考虑这两个视图：

```
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)
```

（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)
```

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。
- 每个视图 仍然 具有很大的灵活性，可以引入我们需要的任何模板变量。 在这个例子中， message 模板变量在每个视图中都不一样。

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。

```
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]))
```

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^_^。

默认情况下， `TEMPLATE_CONTEXT_PROCESSORS` 设置如下：

```
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。

每个处理器将会按照顺序应用。 也就是说如果你在第一个处理器里面向 **context** 添加了一个变量，而第二个处理器添加了同样名字的变量，那么第二个将会覆盖第一个。

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。

html 自动转意

从模板生成 html 的时候，总是有一个风险——变量包了含会影响结果 html 的字符。 例如，考虑这个模板片段：

```
Hello, {{ name }}.
```

首先，这看起来是显示用户名的一个无害的途径，但是考虑如果用户输入如下的名字将会发生什么：

```
<script>alert('hello')</script>
```

用这个用户名，模板将被翻译成：

```
Hello, <script>alert('hello')</script>
```

这意味着浏览器将弹出 JavaScript 警告框！

Similarly, what if the name contained a `'<'` symbol, like this?

<b>username

那样的话模板结果被翻译成这样：

```
Hello, <b>username
```

页面的剩余部分变成了粗体！

为了避免这个问题，你有两个选择：

- 一是你可以确保每一个不被信任的变量 通过过滤器，它把潜在有害的 html 字符转换为无害的。`escape` 这是最初几年 django 的默认方案，但是这样的问题是把责任推给 *you* 自己，开发者、模版作者，来确保转意每一件事情。 很容易就忘记转意数据。
- 二是，你可以利用 django 的自动 html 转意。 这一章的剩余部分描述自动转意是如何工作的。

在 django 里默认情况下，每一个模板自动转意每一个变量标签的输出。 尤其是这五个字符。

- `<` is converted to `<`
- > 被转换为>
- `'` (single quote) is converted to `'`
- `"` (double quote) is converted to `"`
- `&` is converted to `&`

另外，我强调一下这个行为默认是开启的。 如果你正在使用 django 的模板系统，那么你是被保护的。

如何关闭它

如果你不想数据被自动转意，在每一站点级别、每一模板级别或者每一变量级别你都能用两三中方法来关闭 它。

用 safe 过滤器为单独的变量关闭自动转意：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML* . In this example, if `data` contains `'<b>'` , the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For Template Blocks

为了控制模板的自动转意,用标签 autoescape 来包装整个模板(或者模板中常用的部分),就像 这样

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

autoescape 标签有两个参数 on 和 off 同时,你可能想阻止一部分自动转意,对另一部分自动转意 这是一个模板的例子

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

auto-escaping 标签的作用域不仅可以影响到当前模板还可以通过 include 标签作用到其他标签,就像 block 标签一样 例如：

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}

# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

由于在 base 模板中自动转意被关闭,所以在 child 模板中自动转意也会关闭.因此,在下面一段 HTML 被提交时,变量 greeting 的值就为字符串 Hello!Removing plasma-scriptengine-javascript ...

Removing shared-desktop-ontologies ...

Removing virtuoso-opensource-6.1-bin ...

Removing virtuoso-opensource-6.1-common ...

Removing odbcinst1debian2 ...

Removing odbcinst ...

Automatic Escaping of String Literals in Filter Arguments

就像我们前面提到的,过滤器也可以是字符串.

```
{{ data|default:"This is a string literal." }}
```

所有字符常量没有经过转义就被插入模板,就如同它们都经过了 safe 过滤 这是由于字符常量完全由模板作者决定,因此编写模板的时候必须确保文本的正确性.

这意味着你必须这样写

```
{{ data|default:"3 &lt; 2" }}
```

而不是这样

```
{{ data|default:"3 < 2" }}  <-- Bad! Don't do this.
```

来源于自变量的数据不受影响 如果必要,变量内容会自然的转义,因为它们始终都在模板作者的控制下.

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。 If none of the templates exist, a TemplateDoesNotExist exception will be raised.

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。

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。

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**
- **/path/to/myproject/music/templates/foo.html**

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。 We'd recommend the latter, because your filters might be useful to you in future projects.

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。 We'll explain this shortly.

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。 For example:

```
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。

Note

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。 For example, consider this template:

```
Hello, {{ person.name }}.

{% ifequal name.birthday today %}
    Happy birthday!
{% else %}
    Be sure to come back on your birthday
    for a splendid surprise message.
{% endifequal %}
```

#!/usr/bin/env python

```
# -*- coding: UTF-8 -*-

from django import template


register = template.Library()

import pdb


@register.tag(name='current_time')

def do_current_time(parser, token):

#  pdb.set_trace()

  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.split_contents()[0]

    raise template.TemplateSyntaxError(msg)

  return CurrentTimeNode(format_string[1:-1])
```

```
import datetime

class CurrentTimeNode(template.Node):
  def __init__(self, format_string):
    self.format_string = str(format_string)

  def render(self, context):
#    pdb.set_trace()
    now = datetime.datetime.now()
    return now.strftime(self.format_string)
#register.tag('current_time', do_current_time)#实例化
{%load custom_tag%}

<p>The time is    '{% current_time "%Y-%m-%d %I:%M %p" %}'.</p>
```

详见：http://docs.djangoproject.com/en/1.2/howto/custom-template-tags/

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：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：

```
from django.conf import settings
from django.template import TemplateDoesNotExist
import zipfile

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。 If we put this code in a package called mysite.zip_loader , then we add mysite.zip_loader.load_template_source to TEMPLATE_LOADERS .

第 10 章： 数据模型高级进阶

在第 5 章里，我们介绍了 Django 的数据层如何定义数据模型以及如何使用数据库 API 来创建、检索、更 新以及删除记录 在这章里，我们将向你介绍 Django 在这方面的一些更高级功能

如我们在第 5 章的讲解,在数据库对象获取特定字段的值只需直接使用属性那么简单 例如,要确定 ID 为 50 的书本的标题,我们这样做:

```
>>> from mysite.books.models import Book
>>> b = Book.objects.get(id=50)
>>> b.title
u'The Django Book'
```

但是,在之前有一件我们没提及到的是表示为 `ForeignKey` 或 `ManyToManyField` 的相互关联对象字段,它们的作用稍有不同

访问外键(Foreign Key)值

当你获取一个为 `ForeignKey` 字段时,你会得到相关的数据模型对象. 例如:

```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
u'http://www.apress.com/'
```

对于用`` ForeignKey`` 来定义的关系来说，在关系的另一端也能反向的追溯回来，只不过由于不对称性的关系而稍有不同。 通过一个`` publisher`` 对象，直接获取 books ，用 publisher.book_set.all() ，如下：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.all()
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

实际上，`book_set` 只是一个 `QuerySet`（参考第 5 章的介绍），所以它可以像 `QuerySet` 一样,能实现数据过滤和分切，例如：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.filter(name__icontains='django')
[<Book: The Django Book>, <Book: Pro Django>]
```

属性名称 `book_set` 是由模型名称的小写(如 book)加`_set` 组成的。

访问多对多值**(Many-to-Many Values)**

Many-to-many values work like foreign-key values, except we deal with `QuerySet` values instead of model instances. 例如,这里是如何查看书籍的作者

```
>>> b = Book.objects.get(id=50)
>>> b.authors.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.authors.filter(first_name='Adrian')
[<Author: Adrian Holovaty>]
>>> b.authors.filter(first_name='Adam')
[]
```

反方向也可以 要查看一个作者的所有书籍,使用 `author.book_set` ,就如这样:

```
>>> a = Author.objects.get(first_name='Adrian', last_name='Holovaty')
>>> a.book_set.all()
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

这里,就像使用 `ForeignKey` 字段,属性名 `book_set` 是在数据模型(model)名后追加`_set` 而产生的.

Manager 增加额外的方法

增加额外 manager 的方法是首选的办法来添加表级功能的你的模块。 (For row-level functionality i.e., functions that act on a single instance of a model object use model methods, which are explained later in this chapter.)

例如,我们为 Book 模型定义了一个 title_count()方法,它需要一个关键字,返回包含这 个关键字的书的数量.（这个例子稍微做作，但它演示了如何管理工作。）

```
# models.py

from django.db import models

# ... Author and Publisher models here ...

**class BookManager(models.Manager):**
    **def title_count(self, keyword):**
        **return self.filter(title__icontains=keyword).count()**

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

    def __unicode__(self):
        return self.title
```

有了这个 manager 地方，我们现在可以这样做：

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

第 11 章 通用视图

使用通用视图

使用通用视图的方法是在 URLconf 文件中创建配置字典，然后把这些字典作为 URLconf 元组的第 三个成员。 (See Passing Extra Options to View Functions in Chapter 8 for an overview of this technique.)

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('',
    (r'^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：

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('',
    (r'^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    **(r'^about/(\w+)/$', 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。 这里唯一有点棘手的事情是处理找不到模板的情况 我们不希望一个不存在的模板导致一个服务端错误，用 TemplateDoesNotExist 异常并且升起 404 errors 取而代之。

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(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
```

```
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

    class Meta:
        ordering = ['name']
```

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。 我们能够通过在额外参数字典中包含一个 `template_name` 键 来显式的告诉 `object_list` 视 图使用那个模板

```
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(),
    **'template_name': 'publisher_list_page.html',**
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

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 %}
```

制作友好的模板 **Context**

你也许已经注意到范例中的出版商列表模板在变量 object_list 里保存所有的书籍。 这个方法工作的很好，只是对编写模板的人不太友好： 他们必须知道这里正在处理的是书籍。 他们不得不去了解他们现在处理的数据是什么， 比方说在这里是书籍。 更好的变量名应该是 publisher_list， 这样变量所代表的内容就显而易见了。

我们可以很容易的像下面这样修改 template_object_name 参数的名称：

```
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(),
    'template_name': 'publisher_list_page.html',
    'template_object_name': 'Publisher',
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

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

{% block content %}

  <h2>Publishers</h2>

  <ul>

    {% for publisher in Publisher_list %}

      <li>{{ publisher.name }}</li>

    {% endfor %}

  </ul>

{% endblock %}
```

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。 这是非常方便的

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}**
}
```

注意 `Book.objects.all` 后面没有括号；这表示这是一个函数的引用，并没有 真调用它（通用视图将会在渲染时调用它）。

```
<h2>Books</h2>

<ul>

    {% for book in book_list %}

        <li>{{ book.title }}</li>

    {% endfor %}

</ul>
```

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）。 大部分通用视图都带有一个参数，这是视图知道显示那一组对象的原因。

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）。 大部分通用视图都带有一个参数，这是视图知道显示那一组对象的原因。

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.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),**
```

```
)

{% extends "base.html" %}

{% block content %}

    <h2>Apress_list</h2>

    <ul>

        {% for book in apress_list %}

            <li>{{ book.title }}</li>

        {% endfor %}

    </ul>

{% endblock %}
```

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。

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('mysite.books.views',

    (r"^books/([\w'\s]+)$",'books_by_publisher'),


)
```

接下来，我们写 `books_by_publisher` 这个视图：

```
from django.shortcuts import get_object_or_404
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).
    publisher = get_object_or_404(Publisher, name__iexact=name)

    # 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 = 'book',
        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。

处理额外工作

我们再来看看最后一个常用模式：

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 django.shortcuts import get_object_or_404
from django.views.generic import list_detail
from mysite.books.models import Author

def author_detail(request, author_id):
```

```
    # Delegate to the generic view and get an HttpResponse.
    response = list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )

    # Record the last accessed date. We do this *after* the call
    # to object_detail(), not before it, so that this won't be called
    # unless the Author actually exists. (If the author doesn't exist,
    # object_detail() will raise Http404, and we won't reach this point.)
    now = datetime.datetime.now()
    Author.objects.filter(id=author_id).update(last_accessed=now)

    return response
```

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

{% block content %}

    <h2>{{author.first_name}}*{{author.last_name}}</h2>

    <ul>

            <li>id={{ author.id }}</li>

            <li>email={{ author.email }}</li>

            <li>更新最后访问时间为={{ author.last_accessed }}</li>

    </ul>
```

```
{% endblock %}
```

注释

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。

第十二章： 部署 **Django**

本章包含创建一个 django 程序最必不可少的步骤 在服务器上部署它

如果你一直跟着我们的例子做，你可能正在用 `runserver` 但是 `runserver` 要部署你的 django 程序，你需要挂接到工业用的服务器 如：Apache 在本章，我们将展示如何做，但是，在做之前我们要给你一个(要做的事的)清单.

准备你的代码库

很幸运，`runserver` 但是，在开始前，有一些 *essential things*

关闭 **Debug** 模式.

我们在第 2 章创建了一个 project ,命令 `django-admin.py startproject` created a `settings.py` file with `DEBUG` set to `True` . django 会检查这个设置和改变他们的行为， 如果 `DEBUG` 模式被开启. 例如， 如果 `DEBUG` 被设置成 `True` , 那么:

- 所有的数据库查询将被保存在内存中， 以 `django.db.connection.queries` 的形式. 你可以想象，这个吃内存!
- 任何 404 错误都将呈现 django 的特 殊的 404 页面(第 3 章有)而不是普通的 404 页面。 这个页面包含潜在的敏感信息，但是不会暴露在公共互联网。
- 你的应用中任何未捕获的异常，从基本的 python 语法错误到数据库错误以及模板语法错误都会返回漂亮的 Django 错误页面。 这个页面包含了比 404 错误页面更多的敏感信息，所以这个页面绝对不要公开暴露。

简单的说，把`` DEBUG`` 设置成`` True`` 相当于告诉 Django 你的网站只会被可信任的开发人员

使用。 Internet 里充满了不可信赖的事物，当你准备部署你的应用时，首要的事情就是把`` DEBUG`` 设置为`` False`` 。

来关闭模板 **Debug** 模式。

类似地，你应该在生产环境中把 `TEMPLATE_DEBUGFalse` 如果这个设为`` True`` ，为了在那个好看的错误页面上显示足够的东西，Django 的模版系统就会为每一个模版保存一些额外的信息。

实现一个 **404** 模板

如果`` DEBUG`` 设置为`` True`` ，Django 会显示那个自带的 404 错误页面。 但如果`` DEBUG`` 被设置成`` False`` ，那它的行为就不一样了： 他会显示一个在你的模版根目录中名字叫`` 404.html`` 的模版 所以，当你准备部署你的应用时，你会需要创建这个模版并在里面放一些有意义的"页面未找到"的信息

这里有一个示例的`` 404.html`` ，你可以用它作为一个出发点。 It assumes you're using template inheritance and have defined a `base.html` with blocks called `title` and `content` .

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

{% block title %}Page not found{% endblock %}

{% block content %}
<h1>Page not found</h1>

<p>Sorry, but the requested page could not be found.</p>
{% endblock %}
```

To test that your `404.html` is working, just change `DEBUG` to `False` and visit a nonexistent URL. (This works on the `runserver` just as well as it works on a production server.)

Implementing a 500 Template

Similarly, if `DEBUG` is `False` , then Django no longer displays its useful error/traceback pages in case of an unhandled Python exception. Instead, it looks for a template called `500.html` and renders it. Like `404.html` , this template should live in your root template directory.

There's one slightly tricky thing about `500.html` . You can never be sure *why* this template is being rendered, so it shouldn't do anything that requires a database connection or relies on any potentially broken part of your infrastructure. (For example, it should not use custom template tags.) If it uses template inheritance, then the parent template(s) shouldn't rely on potentially broken infrastructure, either. Therefore, the best approach is to avoid template inheritance and use something very simple. Here's an example `500.html` as a starting point:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
    <title>Page unavailable</title>
</head>
<body>
    <h1>Page unavailable</h1>

    <p>Sorry, but the requested page is unavailable due to a
```

```
    server hiccup.</p>

    <p>Our engineers have been notified, so check back later.</p>
</body>
</html>
```

Setting Up Error Alerts

When your Django-powered site is running and an exception is raised, you'll want to know about it, so you can fix it. By default, Django is configured to send an e-mail to the site developers whenever your code raises an unhandled exception but you need to do two things to set it up.

First, change your `ADMINS` setting to include your e-mail address, along with the e-mail addresses of any other people who need to be notified. This setting takes a tuple of `(name, email)` tuples, like this:

```
ADMINS = (
    ('John Lennon', 'jlennon@example.com'),
    ('Paul McCartney', 'pmacca@example.com'),
)
```

Second, make sure your server is configured to send e-mail. Setting up `postfix` , `sendmail` or any other mail server is outside the scope of this book, but on the Django side of things, you'll want to make sure your `EMAIL_HOST` setting is set to the proper hostname for your mail server. It's set to `'localhost'` by default, which works out of the box for most shared-hosting environments. You might also need to set `EMAIL_HOST_USER` , `EMAIL_HOST_PASSWORD` , `EMAIL_PORT` or `EMAIL_USE_TLS` , depending on the complexity of your arrangement.

Also, you can set `EMAIL_SUBJECT_PREFIX` to control the prefix Django uses in front of its error e-mails. It's set to `'[Django] '` by default.

用 *apache* 和 *mod_python* 来部署 *Django*

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：

目前，Apache 和 mod_python 是在生产服务器上部署 Django 的最健壮搭配。

在/etc/apache2/httpd.conf 或 site-avaible 下也可，

添加

LoadModule python_module /usr/lib/apache2/modules/mod_python.so

PythonPath "['/home/jhjguxin/Desktop/djcode', '/usr/lib/pymodules/python2.7/django'] + sys.path"

<Location "/">

SetHandler python-program

PythonHandler django.core.handlers.modpython

SetEnv DJANGO_SETTINGS_MODULE mysite.settings

PythonDebug On

</Location>

启动阿帕奇：sudo apache2ctl -k restart

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要需要重启 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>
```

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。

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。

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。

RAM 怎么也不嫌多

Even the really expensive RAM is relatively affordable these days. 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。 This shouldn’t be too hard; weve developed a site with more than half a million newspaper articles, and it took under 2GB of space.

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。 Chapter 15 is your best friend here: 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。

第十三章： 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 ***views and MIME-types***

Recall from Chapter 3 that a view function is simply a Python function that takes a Web request and returns a Web response. 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。

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。

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。 It will be used by browsers in the Save As dialog.
>
> To assign a header on an `HttpResponse` , just treat the `HttpResponse` as a dictionary and set a key/value.
>
> 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。

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 Django uses the remainder of the URL (everything after `/feeds/`) to determine which feed to return.

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。

一个简单的 Feed

URLconf:

```
from django.conf.urls.defaults import *
from mysite.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
#    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

#!/usr/bin/env python

-*- coding: UTF-8 -*-

#feeds.py

from django.contrib.syndication.feeds import Feed

from mysite.books.models import Book,Author

class LatesEntries(Feed):

```
  title="书名"

  authors="作者"

  publisher='出版社'

  publication_date='出版日期'

  link="http://127.0.0.1:8000/books/archive/"

  description="The latest news about stuff."

  item_link='/books/'

  def items(self):

    return Book.objects.order_by('-publication_date')[:5]

#  def items(self):

#    return Author.objects.order_by('-last_accessed')[:5]
```

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！

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。 To create a sitemap, you just need to write a `Sitemap` class and point to it in your URLconf.

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。

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.index` and `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.

说的就是'Site' object has no attribute 'flatpage_set',看了 django 的源代码,也不知道是怎么回事.

后来看了 google 上的一些 django apps,我终于把上面这个错误解决了.

方法是:

在 settings.py 中增加

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.sites',<<<<增加这个
    'django.contrib.contenttypes',
    'django.contrib.sessions',
......
    'django.contrib.flatpages', <<<<增加这个

MIDDLEWARE_CLASSES = (

....
  'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware', <<<增加这个

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()` 。

`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：

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 Book(models.Model): # 在模块中添加save函数
    # ...
    def save(self, *args, **kwargs):
        super(Book, self).save(*args, **kwargs)
        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。 The function makes an HTTP request to Google’s servers, so you may not want to introduce that network overhead each time you call `save()` .

Finally, if `'django.contrib.sitemaps'` is in your `INSTALLED_APPS` , then your `manage.py` will include a new command, `ping_google` . This is useful for command-line access to pinging. For example:

```
python manage.py ping_google /sitemap.xml
```

第十四章： 会话、用户和注册

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。 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。 Google 会（以及目前）使用它在网页上显示你账号的用户名。

存取 **Cookies**

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: 每一个`` HttpRequest`` 对象都有一个`` COOKIES`` 对象，该对象的行为类似一个字典，你可以使用它读取任何浏览器发送给视图（view）的 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

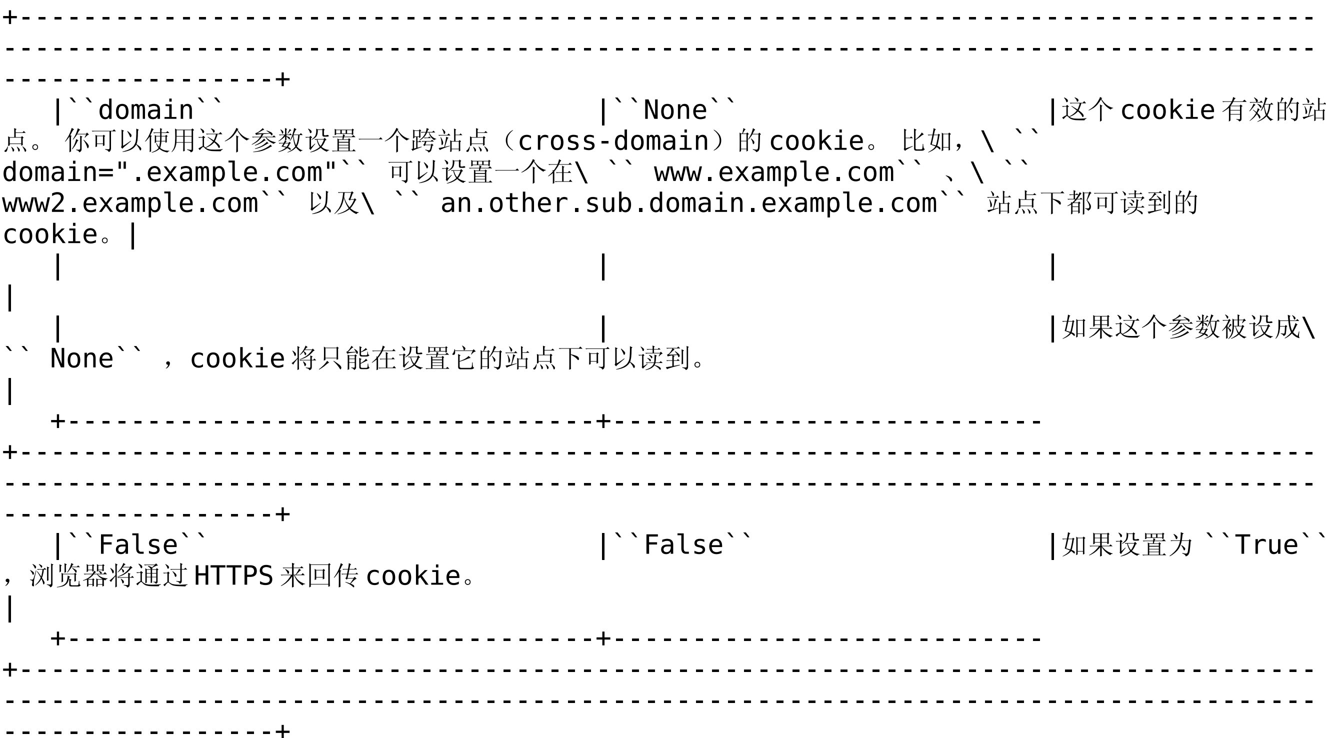
基于 GET 参数来设置 favorite_color cookie 的例子： 这里有一个设置了基于`` GET`` 参数的`` favorite_color`` cookie 的例子：

```
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 14-1。

table:: 表 14-1: Cookie 选项

参数	缺省值	Description
``max_age``	``None``	cookie 需要延续的时间（以秒为单位） 如果参数是\ `` None`` ，这个 cookie 会延续到浏览器关闭为止。
``expires``	``None``	cookie 失效的实际日期/时间。 它的格式必须是：\ `` "Wdy, DD-Mth-YY HH:MM:SS GMT"`` 。如果给出了这个参数，它会覆盖\ `` 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。 当你不是控制你的站点的顶层时，这样做是特别有用的。



``domain``	``None``	这个 cookie 有效的站点。 你可以使用这个参数设置一个跨站点（cross-domain）的 cookie。 比如，\ `` domain=".example.com"`` 可以设置一个在\ `` www.example.com`` 、\ `` www2.example.com`` 以及\ `` an.other.sub.domain.example.com`` 站点下都可读到的 cookie。
		如果这个参数被设成\ `` None`` ，cookie 将只能在设置它的站点下可以读到。
``False``	``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：让我们看一下其中一些比较重要的问题：

cookie 的存储是自愿的，一个客户端不一定要去接受或存储 cookie。 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：

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。

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。 这就意味着您不应该使用 cookie 来在存储任何敏感信息。

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。

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。

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`)

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

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):
    if request.method != 'POST':
        raise Http404('Only POSTs are allowed')

    if 'comment' not in request.POST:
        raise Http404('Comment not submitted')

    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")

    c = comments.Comment(comment=request.POST['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):
    if request.method != 'POST':
        raise Http404('Only POSTs are allowed')
    try:
        m = Member.objects.get(username=request.POST['username'])
        if m.password == request.POST['password']:
            request.session['member_id'] = m.id
```

```
        return HttpResponseRedirect('/you-are-logged-in/')
    except Member.DoesNotExist:
        return HttpResponse("Your username and password didn't match.")
```

下面的例子将登出一个在上面已通过`` login()`` 登录的用户：

```
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。 在你证实了测试 cookie 已工作了之后这样操作。

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。(pk 为 sessionid 的值)

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='78e7eb67d49af0bebc692dbc30d92de1')
>>> s.expire_date
datetime.datetime(2011, 3, 25, 15, 21, 44)#为该个 cook 失效的时期
```

>>> s.session_data

u'gAJ9cQEoVRJfYXV0aF91c2VyX2JhY2tlbmRxAlUpZGphbmdvLmNvbnRyaWIuYXV0aC5iY-WNrZW5k\ncy5Nb2RlbEJhY2tlbmRxA1UNX2F1dGhfdXNlcl9pZHEEigEBdS5hYTY2NTFmN-mZjNjQ0M2VmYmY2\nMzE5NWZhOTk0YTY0Ng==\n'

>>> s.get_decoded()

{'_auth_user_id': 1L, '_auth_user_backend': 'django.contrib.auth.backends.ModelBackend'}

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。

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。

配置 ***Session*** 引擎¶

New in Django 1.0..

缺省情况下，Django 将 Session 存储在数据库中 (使用模型 `django.contrib.sessions.models.Session`)。尽管这很方便，但在某些情况下，把 Session 放在其它的地方速度会更快。因此 Django 允许您通过配置让它将 Session 数据保存在文件系统或缓冲区中。

使用基于文件的 **Session**¶

要使用基于文件的 Session，请将 `SESSION_ENGINE` 设置为 `"django.contrib.sessions.backends.file"` 。

您可能还需要修改 `SESSION_FILE_PATH` 这一设置以便控制 Django 存储 Session 文件的位置，缺省情况下，它使用 `tempfile.gettempdir()` ，通常是 `/tmp` 。

使用基于缓冲区的 **Session**¶

要使用 Django 的缓冲区系统来保存 Session，需要将 `SESSION_ENGINE` 设置为 `"django.contrib.sessions.backends.cache"` 。您必须确保您已经配置了缓冲区，详情请参考 缓冲区文档 。

Note

只有在使用 Memcached 作为缓冲后台时，才能使用基于缓冲区的 Session。因为以本地内存作

为缓冲后台时，它存储缓冲数据的时间太短了，这样直接访问文件或数据库的速度，要比通过缓冲区访问文件或数据库的速度更快一些。

浏览器关闭即失效会话 vs. 持久会话 Persistent Sessions

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

用户与 *Authentication*

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

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

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。 Keeping track of users obviously requires cookies, and thus builds on the session framework.
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'` `SessionMiddleware` .

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) If the user isn’t logged in, this will instead be an `AnonymousUser` object (see below for more details).

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 14-3 和 14-4 分别列出了`` User`` 对象中的属性（fields)和方法。

表 14-3. `User` 对象属性

属性	**Description**
`username`	必需的，不能多于 30 个字符。 仅用字母数字式字符（字母、数字和下划线）。
`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`	必需的。 A hash of, and metadata about, the password (Django doesn’t store the raw password). See the Passwords section for more about this value.
`is_staff`	布尔值. 用户是否拥有网站的管理权限.
`is_active`	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 用户最后登录的时间，缺省会设置为当前时间
`is_superuser`	Boolean. Designates that this user has all permissions without explicitly assigning them.
`last_login`	用户上次登录的时间日期。 它被默认设置为当前的日期/时间。
`date_joined`	账号被创建的日期时间 当账号被创建时，它被默认设置为当前的日期/时间。

```
   table:: 表 14-4. ``User`` 对象方法


   +------------------------------------------------------------------------------+
   |方法                          |Description
|
   +==============================================================================+
   |``is_authenticated()``        |对于真实的 User 对象，总是返回\ `` True`` 。
                                  |这是一个分辨用户是否已被鉴证的方法。 它并不意味着任何权限，也不
检查用户是否仍是活动的。 它仅说明此用户已被成功鉴证。                          |
   +------------------------------------------------------------------------------+
   |``is_anonymous()``            |对于\ `` AnonymousUser`` 对象返回\ `` True`` （对于真实
的\ `` User`` 对象返回\ `` False`` ）。
                                  |总的来说，比较于这个方法，你应该倾向于使用\ ``
```

```
is_authenticated()`` 方法。
    +-------------------------------------------------------------------------------+
    |``get_full_name()``       |返回\ `` first_name`` 加上\ `` last_name`` ，中间插入一
个空格。
    +-------------------------------------------------------------------------------+
    |``set_password(passwd)`` |Sets the user's password to the given raw string,
taking care of the password hashing. ``has_perm(perm)``
    +-------------------------------------------------------------------------------+
    |``check_password(passwd)`|如果用户拥有给定的权限，返回 ``True`` ， ``perm`` 应形如
``"package.codename"`` 的格式。 比较时会使用密码哈希表。
    +-------------------------------------------------------------------------------+
    |``get_group_permissions()`|返回一个用户通过其所属组获得的权限字符串列表。
    +-------------------------------------------------------------------------------+
    |``get_all_permissions()`|返回一个用户通过其所属组以及自身权限所获得的权限字符串列表。
    +-------------------------------------------------------------------------------+
    |``has_perm(perm)``       |如果用户有指定的权限，则返回\ `` True`` ，此时\ `` perm``
的格式是\ `` "package.codename"`` 。如果用户已不活动，此方法总是返回\ `` False`` 。     |
    +-------------------------------------------------------------------------------+
    |``has_perms(perm_list)``|如果用户拥有\ * 全部* 的指定权限，则返回\ `` True`` 。 如果用
户是不活动的，这个方法总是返回\ `` False`` 。
    +-------------------------------------------------------------------------------+
    |``has_module_perms(app_label)`` |如果用户拥有给定的\ `` app_label`` 中的任何权限，则
返回\ `` True`` 。如果用户已不活动，这个方法总是返回\ `` False`` 。
    +-------------------------------------------------------------------------------+
    |``get_and_delete_messages()`` |返回一个用户队列中的\ `` Message`` 对象列表，并从队列
中将这些消息删除。
    +-------------------------------------------------------------------------------+
    |``email_user(subj, msg)`` |向用户发送一封电子邮件。 这封电子邮件是从\ ``
DEFAULT_FROM_EMAIL`` 设置的地址发送的。 你还可以传送一个第三参数：\ `` from_email`` ，以覆盖
电邮中的发送地址。
    +-------------------------------------------------------------------------------+
```

最后，`` User`` 对象有两个 many-to-many 属性。 `` groups`` 和`` permissions`` 。正如其他的 many-to-many 属性使用的方法一样，`` User`` 对象可以获得他们相关的对象：

最后，`` User`` 对象有两个 many-to-many 属性。 `` groups`` 和`` permissions`` 。正如其他的 many-to-many 属性使用的方法一样，`` User`` 对象可以获得他们相关的对象：

```
# 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() 。 Django provides two functions to perform these actions in django.contrib.auth : authenticate() and 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

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。

```
>>> from django.contrib import auth
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...     print "Correct!"
... else:
...     print "Invalid password."
```

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_view(request):
    username = request.POST.get('username', '')
    password = request.POST.get('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。 It takes an HttpRequest object and has no return value:

```
from django.contrib import auth

def logout_view(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),
)
```

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： 一个简单的 template 看起来是这样的

```
{% 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 }}" /-->

    <input type="hidden" name="next" value="/books/" />

  </form>



{% endblock %}
```

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('/accounts/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/` . `/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. The view code can then assume that the user is logged in.

#默认登录地址 LOGIN_URL (/accounts/login/ by default)

LOGIN_URL='/login'

As a shortcut, you can use the convenient login_required() decorator:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

login_required() does the following:

- If the user isn't logged in, redirect to `settings.LOGIN_URL(#在 settings 文件里默认登录地址 LOGIN_URL(/accounts/login/ by default) LOGIN_URL='/login')`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.

- If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called `"next"`. If you would prefer to use a different name for this parameter, login_required() takes an optional `redirect_field_name` parameter:

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

Note that if you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as it's key rather than `"next"` (the default).

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。

```
def user_can_vote(user):
    return user.is_authenticated() and user.has_perm("polls.can_vote")

@user_passes_test(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。 Note that `user_passes_test` does not automatically check that the `User` is authenticated; you should do that yourself.

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/` ）。 If the user doesn't pass the test, then the `user_passes_test` decorator will redirect the user to the `login_url` .

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: Using this decorator, the earlier example can be written as follows:

```
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 django.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: `salt` is a random string used to salt the raw password to create the hash, for example:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

`User.set_password()` 和 `User.check_password()` 函数在后台处理和检查这些值。

Salted hashes

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 This takes some time, but less than you might think.

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。

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: Django provides a built-in form you can use for this purpose, which we’ll use in this example:

```
#///处理注册

def register(request):

 if request.method=="POST":

  form=UserCreationForm(request.POST)

  if form.is_valid():
```

```
        new_user=form.save()

        return HttpResponseRedirect("/books")

    else:

        form=UserCreationForm()

#   return render_to_response("registration/register.html",{"form":form,})

    return render_to_response("register.html",{"form":form,})
```

registration.html

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



{% block title %}Create an account{% endblock %}



{% block content %}

  <h1>Create an account</h1>



  <form action="" method="post">

      {{ form.as_p }}

      <input type="submit" value="Create the account">

  </form>
```

{% endblock %}

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 9 章）。

Note

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。

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。

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 %}
```

权限、组和消息

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。

会自动为每一个 Django 模型创建三个基本权限：增加、改变和删除。 Behind the scenes, these permissions are added to the `auth_permission` database table when you run `manage.py syncdb` .

权限以 "<app>.<action>_<object_name>" 的形式出现。

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。

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。

```
{% 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。

第十五章： 缓存机制

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。 This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

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 这些都是缓存类型，您不直接控制，但您可以向其中提供线索（通过 HTTP 标头）关于您网站的哪个部分应被缓存，以及如何。

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
也就是说，你必须告诉它在您的缓存的数据是否应该存在一个数据库中，在文件系统，或直接在内存中。 这是一个重要的决定，影响您的高速缓存的性能，是的，有些类型的缓存比其它类型快。

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 这里是一个为 CACHE_BACKEND 所有可用的值解释

内存缓冲

迄今为止最快，最有效的高速缓存可 Django 的类型，Memcached 的是一个完全基于内存的缓存框架，最初开发来处理高负荷的 LiveJournal.com 并随后开放，由 Danga Interactive 公司采购。它的使用，如 Facebook 和维基百科网站，以减少数据库访问，并大幅提高站点的性能。

Memcached 在 http://danga.com/memcached 是免费的。它作为一个守护 进程运行，并分配了特定数量的内存。 它只是提供了添加，检索和删除缓存中的任意数据的高速接口。 所有数据都直接存储在内存中，所以没有对使用的数据库或文件系统的开销。

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 这两个可用版本。 选择和安装以下模块之一：

- 最快的可用选项是一个模块，称为 cmemcache，在 http://gijsbert.org/cmemcache。
- 如果您无法安装 cmemcache， 您可以安装 python - Memcached，在 ftp://ftp.tummy.com/pub/python-memcached/。如果该网址已不再有效，只要到 Memcached 的网站 http://www.danga.com/memcached/），并从客户端 API 的 Python 绑定。

若要使用 Memcached 的 Django，设置 CACHE_BACKEND 到 memcached：/ / IP：port/，其中 IP 是 Memcached 的守护进程的 IP 地址和 port 的端口 Memcached 的运行。

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 这意味着您可以运行在多台机器 Memcached 的守护进程，该程序会当成一个单一缓存组机器，而无需重复每台机器上的缓存值。 要充分利用此功能，包括 CACHE_BACKEND 所有服务器的地址，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 . 由于缓存的数据存储在内存中的数据将会丢失，如果您的服务器崩溃。 显然，内存不打算永久存储数据，因此不依赖基于内存缓存仅作为您的存储数据缓存。毫无疑问，在 Django 的缓存后端都应该用于永久储存他们都拟用于缓存的解决方案，但我们不是储存指出此点，这里是因为基于内存的缓存尤其暂时的。

数据库缓存

作为您的缓存后端数据库表中，首先在数据库中创建通过运行这个命令缓存表：

```
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.（此名称可以是任何你想的，只要它是一个有效的表名，这不是已经在您的数据库中使用。）此命令创建数据库中的一个表，在正确的格式，**Django** 的数据库 缓存系统所期望的。

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:** 在这个例子中，高速缓存表的名字是 **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.您不能使用您的缓存表在不同的数据库后端。

数据库缓存效果最佳，如果你已经有了一个快速，良好的索引数据库服务器。

文件系统缓存

要存储的文件系统缓存的项目，使用"file://"缓存类型 CACHE_BACKEND。例如，缓存数据存储在/var/tmp/django_cache，使用此设置：

```
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'. 头两项是 file://，第三个是第一个字符的目录路径，/var/tmp/django_cache。如果你是 Windows，之后提出 file://，这样文件的驱动器号：

```
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. 它不管你放在设置的结束斜线。

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 apache 继续上面的例子，如果您的服务器运行作出的用户 apache，确保目录/var/tmp/django_cache 存在并且是可读和由用户可 写。

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 每个文件的名称是缓存键，除了文件系统使用的安全。

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 此缓存的多进程和线程安全。 设置 `CACHE_BACKEND` 为 `locmem:///` 来使用它，例如:

```
CACHE_BACKEND = 'locmem:///'
```

请注意，每个进程都有自己的私自缓存实例，这意味着没有跨进程缓存是可能的。 这显然也意味着本地内存缓存并不特别记忆效率，所以它可能不是一个生产环境不错的选择。 这是很好的发展。

仿缓存（供开发时使用）

最后，Django 提供一个假缓存的设置：

这是有用的，如果你有一个生产基地，使用在不同地方的重型缓存，但开发/测试环境中，你不想来缓存和不希望要改变您的代码的特殊情况后者。 要激活虚拟缓存，就像这样设置 CACHE_BACKEND：

```
CACHE_BACKEND = 'dummy:///'
```

使用自定义缓存后端

尽管 Django 的包括针对缓存的支持后端的即开即用，有时你可能想使用自定义缓存后端。 要使用外部高速缓存与 Django 的后端，在最初使 Python import 作为该计划的一部分（前部分的初步冒号）的 CACHE_BACKEND URI 的，像这样：

```
CACHE_BACKEND = 'path.to.backend://'
```

如果您构建自己的后端，你可以使用标准的参考实现缓存后端。 django/core/cache/backends/你会发现在的 Django 的源目录。

注意 如果没有一个真正令人信服的理由，如主机不支持他们，你要坚持到缓存后端使用 Django 中。 他们一直行之有效，易于使用。

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: 他们给予查询字符串的 CACHE_BACKEND 设置样式。 有效参数如下：

> `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）

filesystemmax_entries：对于 locmem，database 和后端，在高速缓存允许的最大条目数之前的旧值将被删除。 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。

在这个例子中， timeout 被设成 60

```
CACHE_BACKEND = "memcached://127.0.0.1:11211/?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*

一旦高速缓存设置，最简单的方法是使用缓存缓存整个网站。 您 需要添加'django.middleware.cache.UpdateCacheMiddleware'和'django.middleware.cache.FetchFromCacheMiddleware'您的 MIDDLEWARE_CLASSES 设置，在这个例子为：

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

注意：

不，这不是一个错字： 中间件的更新，必须先在列表中，而获取的中间件，必须最后。 的细节有点模糊，但看到下面的 MIDDLEWARE_CLASSES 订阅如果您想看到完整的故事。

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 或者，如果 CACHE_MIDDLEWARE_ANONYMOUS_ONLY 设置为 True，只有匿名请求（即不是由登录的用户）将被缓存作出的声明。 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。 CACHE_MIDDLEWARE_ANONYMOUS_ONLY，你应该确保你已经启动 AuthenticationMiddleware。

此外，缓存中间件自动设置在每个中 HttpResponse 几个标题：

- 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

参阅更多的中间件第 17 章。

如果视图设置自己的缓存到期时间（即 它有一个最大，年龄在其缓存部分，控制头），那么页面将届满时为止，而不是 CACHE_MIDDLEWARE_SECONDS 缓存。使用 django.views.decorators.cache 的装饰，您可以轻松地设置视图的到期时间（使用 cache_control 装饰）或禁用缓存 视图（使用 never_cache 装饰）。 请参阅使用其他头节下面有关这些装饰。

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。 `django.views.decorators.cache` defines a `cache_page` decorator that will automatically cache the view's response for you. Its easy to use:

```
from django.views.decorators.cache import cache_page

def my_view(request):
    # ...

my_view = cache_page(my_view, 60 * 15)
```

Or, using Python 2.4's decorator syntax:

```
@cache_page(60 * 15)
def my_view(request):
    # ...
```

`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：

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 : Here's the old URLconf from earlier:

```
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 .

Template Fragment Caching

If you're after even more control, you can also cache template fragments using the cache template tag. To give your template access to this tag, put {% load cache %} near the top of your template.

The {% cache %} template tag caches the contents of the block for a given amount of time. 它至少需要两个参数: the cache timeout, in seconds, and the name to give the cache fragment. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment. For example, you might want a separate cached copy of the sidebar used in the previous example for every user of your site. Do this by passing additional arguments to the {% cache %} template tag to uniquely identify the cache fragment:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

It's perfectly fine to specify more than one argument to identify the fragment. Simply pass as many arguments to {% cache %} as you need.

The cache timeout can be a template variable, as long as the template variable resolves to an integer value. For example, if the template variable my_timeout is set to the value 600 , then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and just reuse that value.

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。

For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. 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。）

The cache module, `django.core.cache` , has a `cache` object that's automatically created from the `CACHE_BACKEND` setting:

```
>>> 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.

If the object doesn't exist in the cache, `cache.get()` returns `None` :

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_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'
```

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()` , but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

There's also a `get_many()` interface that only hits the cache once. `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}
```

最后,你可以用 `cache.delete()` 显式地删除关键字。

```
>>> cache.delete('a')
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A ValueError will be raised if you attempt to increment or decrement a nonexistent cache key.:

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

Note

`incr()` /`decr()` methods are not guaranteed to be atomic. On those backends that support atomic increment/decrement (most notably, the memcached backend), increment and decrement operations will be atomic. However, if the backend doesn't natively provide an increment/decrement operation, it will be implemented using a two-step retrieve/update.

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：caching performed by upstream caches. 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。

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。

Controlling Cache: Using Other Headers

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： Example:

```
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。

- `public=True`

- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True #重新认证`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

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。 ）

If you want to use headers to disable caching altogether, `django.views.decorators.cache.never_cache` is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```

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

If you use caching middleware, it's important to put each half in the right place within the `MIDDLEWARE_CLASSES` setting. That's because the cache middleware needs to know which headers by which to vary the cache storage. Middleware always adds something to the `Vary` response header when it can.

`UpdateCacheMiddleware` runs during the response phase, where middleware is run in reverse order, so an item at the top of the list runs *last* during the response phase. **Thus, you need to make sure that `UpdateCacheMiddleware` appears *before* any other middleware that might add something to the `Vary` header. The following middleware modules do so:**

- 添加 `Cookie` 的 `SessionMiddleware`
- 添加 `Accept-Encoding` 的 `GZipMiddleware` ,
- `LocaleMiddleware` adds `Accept-Language`

`FetchFromCacheMiddleware` , on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs *first* during the request phase.

The `FetchFromCacheMiddleware` also needs to run after other middleware updates the `Vary` header, so **`FetchFromCacheMiddleware` must be *after* any item that does so.**

第十六章：集成的子框架 django.contrib

Python 有众多优点，其中之一就是“开机即用”原则： 安装 Python 的同时安装好大量的标准软件包，这样 你可以立即使用而不用自己去下载。 Django 也遵循这个原则，它同样包含了自己的标准库。 这一章就来讲 这些集成的子框架。

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。

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。 When the Django developers add new functionality to the framework, they use this rule of thumb in deciding whether the new functionality should live in `django.contrib` or elsewhere.

`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 **12** 章
- **`admindocs`** : **Auto-documentation for the Django admin site. This book doesn’t cover this feature; check the official Django documentation.**
- **`auth`** : **Django** 的用户验证框架。 **See Chapter 14.**
- **`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**. This book doesn’t cover this feature; check the official Django documentation.**
- **`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。 参见后面标题为《重定向》的小节。
- **`databrowse`** : **A Django application that lets you browse your data. This book doesn’t cover this feature; check the official Django documentation.**
- **`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。 **See the later section titled Flatpages.**
- **`formtools`** : **A number of useful higher-level libraries for dealing with common patterns in forms. This book doesn’t cover this feature; check the official Django documentation.**
- **`gis`** : **Extensions to Django that provide for GIS (Geographic Information Systems)**

support. These, for example, allow your Django models to store geographic data and perform geographic queries. This is a large, complex library and isn't covered in this book. See http://geodjango.org/ for documentation.

- `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》。
- `localflavor`：Assorted pieces of code that are useful for particular countries or cultures. For example, this includes ways to validate U.S. ZIP codes or Icelandic identification numbers.
- `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。 See the later section titled Redirects.
- `sessions`：Django 的会话框架，参见 12 章。 See Chapter 14.
- `sitemaps`：用来生成网站地图的 XML 文件的框架。 See Chapter 13.
- `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：
- `syndication`：一个用 RSS 和 Atom 来生成聚合订阅源的的框架。 See Chapter 13.
- `webdesign`：Django add-ons that are particularly useful to Web *designers* (as opposed to developers). As of this writing, this included only a single template tag, `{% lorem %}`. Check the Django documentation for information.

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：

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：

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。

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。 这样也会建立默认的站点对象，域名为 example.com。
3. Change the `example.com` site to your own domain, and add any other `Site` objects, either through the Django admin site or via the Python API. 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(max_length=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
from django.shortcuts import get_object_or_404
from mysite.articles.models import Article

def article_detail(request, article_id):
    a = get_object_or_404(Article, id=article_id, sites__id=settings.SITE_ID)
    # ...
```

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。

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（获取当前站点）。 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 。#直接对当前站点操作

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： 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 LJ-World.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。

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。 It’s a model manager (see Chapter 10) that automatically filters its queries to include only objects associated with the current `Site` .

通过显示地将 `CurrentSiteManager` 加入模型中以使用它。 For example:

```
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(max_length=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。

```
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(max_length=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。

Note

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。
- In the authentication framework (see Chapter 14), the django.contrib.auth.views.login view passes the current Site name to the

template as `{{ site_name }}` and the current `Site` object as `{{ site }}` .

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。

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） (For more on sites, see the Sites section.)

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。
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：已经定义好了的

```
from django.db import models
from django.contrib.sites.models import Site

class FlatPage(models.Model):
    url = models.CharField(max_length=100, db_index=True)
    title = models.CharField(max_length=200)
    content = models.TextField(blank=True)
    enable_comments = models.BooleanField()
    template_name = models.CharField(max_length=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)。 The framework doesn't do anything special

with this. 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。 The framework doesn't do anything special with this. 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 。 It uses RequestContext in rendering the template.

如果 FlatpageFallbackMiddleware 没有找到匹配项，该请求继续如常处理

Note

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。

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。

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.sites.models import Site
>>> fp = FlatPage.objects.create(
...     url='/about/',
...     title='About',
...     content='<p>About this site...</p>',
...     enable_comments=False,
...     template_name='',
...     registration_required=False,
... )
>>> 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|safe }}
</body>
</html>
```

Note that we've used the safe template filter to allow flatpage.content to include raw HTML and bypass auto-escaping.

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。 This is a simple lookup table with site_id , old_path , and new_path fields.

Via the Admin Interface

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。

Via the Python API

`django/contrib/redirects/models.py` 中的一个标准 Django 模型代表了重定向。

```
>>> from django.contrib.redirects.models import Redirect
>>> from django.contrib.sites.models import Site
>>> red = Redirect.objects.create(
...     site=Site.objects.get(id=1),
...     old_path='/music/',
...     new_path='/sections/arts/music/',
... )
>>> 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。

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` 。 Thus, if you're logged in to the `example.com` webmail account and visit the malicious page that has an `<iframe>` to `example.com/logout` , the act of visiting the malicious page will log you out from `example.com` .

Clearly, being logged out of a webmail site against your will is not a terrifying breach of security, but this same type of exploit can happen to *any* site that trusts users, such as an online banking site or an e-commerce site, where the exploit could be used to initiate an order or payment without the user's knowledge

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。 *and* send the `confirm` `POST` variable with a value of `'true'` .

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。 Then, when processing the form on the server side, check for that secret field and raise an error if it doesn't validate.

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。

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。

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。

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

The package `django.contrib.humanize` holds a set of Django template filters useful for adding a human touch to data. To activate these filters, add `'django.contrib.humanize'` to your `INSTALLED_APPS` . Once you've done that, use `{% load humanize %}` in a template, and you'll have access to the filters described in the following sections.

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。

Examples:

- 4500 变成 4,500 。
- 45000 变成 45,000 。
- 450000 变成 450,000 。
- 4500000 变成 4,500,000 。

You can pass in either an integer or a string representation of an integer.

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。

Examples:

- 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）。

You can pass in either an integer or a string representation of an integer.

ordinal

该过滤器将整数转换为序数词的字符串形式。

Examples:

- 1 变成 1st 。
- 2 变成 2nd 。
- 3 变成 3rd 。
- 254 becomes 254th.

You can pass in either an integer or a string representation of an integer.

标记过滤器

The package `django.contrib.markup` includes a handful of Django template filters, each of which implements a common markup languages:

- `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：

```
{% 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。

第十七章： 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(请求/响应)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。 如果你是顺着这本书读下来的话，你应该已经多次见到“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

我们从一个简单的例子开始。

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
```

(Note: Although the HTTP header is called `X-Forwarded-For` , Django makes it available as `request.META['HTTP_X_FORWARDED_FOR']` . With the exception of `content-length` and `content-type` , any HTTP headers in the request are converted to `request.META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name.)

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

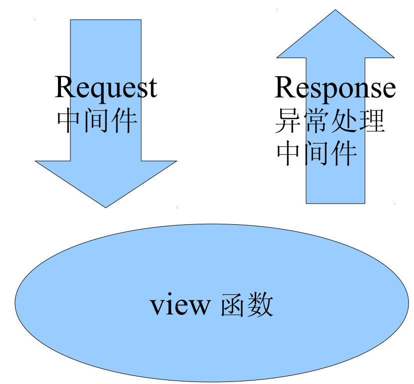
如果按顺序阅读本书，应当已经看到涉及到中间件安装的多个示例,因为前面章节的许多例子都需要某些特定的中间件。 出于完整性考虑，下面介绍如何安装中间件。

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 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。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。



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) __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。

View 预处理函数: process_view(self, request, view, args, kwargs) 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。

表 15-1. 传入 process_view()的参数

参数	说明
`request`	The `HttpRequest` object.
`view`	The Python function that Django will call to handle this request. This is the actual function object itself, not the name of the function as a string.
`args`	将传入 view 的位置参数列表，但不包括 `request` 参数(它通常是传 入 view 的第一个参数)
`kwargs`	将传入 view 的关键字参数字典.

Just like `process_request()` , `process_view()` should return either `None` or an `HttpResponse` object.

- If it returns `None`, Django will continue processing this request, executing any other middleware and then the appropriate view.
- **If it returns an `HttpResponse` object**, Django won't bother calling *any* other middleware (of any type) or the appropriate view. **Django will immediately return that `HttpResponse`.**

Response 后处理函数: process_response(self, request, response) 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。 Here, the processor can modify (修改)the content of a response. One obvious（明显） use case is content compression（压缩）, such as gzipping of the request's HTML.

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。 `request` is the request object, and `response` is the response object returned from the view.

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。 That response could be the original one passed into the function (possibly modified) or a brand-new one.

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
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` . `django.contrib.auth.middleware.AuthenticationMiddleware` .

这个中间件激活认证支持功能. 它在每个传入的 `HttpRequest` 对象中添加代表当前登录用户的 `request.user` 属性。 It adds the `request.user` attribute, representing the currently logged-in user, to every incoming `HttpRequest` object.

完整的细节请参见第 12 章。

通用中间件

Middleware class: `django.middleware.common.CommonMiddleware` .

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)。

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。 will be redirected to the same URL with a leading www..

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。

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：

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

Middleware class: `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

Middleware class: `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。 As discussed above, the `ETag` header is set by the Common middleware.

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)

Middleware class: `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。 It sets request.META['REMOTE_ADDR'] based on request.META['HTTP_X_FORWARDED_FOR'] , if the latter is set. This is useful if you're sitting behind a reverse proxy that causes each request's REMOTE_ADDR to be set to 127.0.0.1 .

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

Middleware class: 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。 See Chapter 14 for details.

站点缓存中间件

Middleware classes: django.middleware.cache.UpdateCacheMiddleware and django.middleware.cache.FetchFromCacheMiddleware .

These middlewares work together to cache each Django-powered page. 已在第 13 章中详细讨论。

事务处理中间件

Middleware class: 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。

第十八章： 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,你需要为已经存在的数据表创建 model. 为此,Django 自带了一个可以通过读取您的数据表结构来生成 model 的工具. 该辅助工具称为 inspectdb,你可以通过执行 manage.py inspectdb 来调用它.

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 The only assumptions are that Django is installed and that you have a legacy database.

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） (Note that some of these settings are optional. Refer to Chapter 5 for more information.)

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. 这里我们使用 myapp 做为应用名.

运行命令 python mysite/manage.py inspectdb . 这将在 DATABASE_NAME 数据库中检查所有的表和打印出为每张表生成的 model class. 看一看输出结果想一下 inspectdb 能做些什么.

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。

清理生成的 **Models**

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:

数据库的每一个表都会被转化为一个 model 类 (也就是说，数据库的表和 model 的类之间做一对一的映射)。 这意味着你需要为多对多连接的表，重构其 models 为 ManyToManyField 的对象。

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。

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”。

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。 The Django model

layer is not required to include every field in your table(s).

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')
```

`inspectdb` 会在该字段后加注 '字段重命名，因为它是一个 `Python` 保留字' 。

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。

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。 This should be a tuple of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

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` 。 If they're not valid, it should return `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
```

For more on authentication backends, see the official Django documentation

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>
```

有了这样的设置，只有那些以/admin/开头的 URL 地址才会触发 Django 去进行处理，而任何其他页面依旧按之前已经存在的那些设置进行处理。 Any other page will use whatever infrastructure already existed.

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/` 。

第十九章： 国际化

Django 诞生于美国中部堪萨斯的劳伦斯，距美国的地理中心不到 40 英里。 像大多数开源项目一样，Djano 社区逐渐开始包括来自全球各地的许多参与者。 鉴于 Django 社区逐渐变的多样性，国际化和本地化逐渐变的很重要。 由于很多开发者对这些措辞比较困惑，所以我们将简明的定

义一下它们。

- 国际化* 是指为了该软件在任何地区的潜在使用而进行程序设计的过程。 它包括了为将来翻译而标记的文本（比如用户界面要素和错误信息等）、日期和时间的抽象显示以便保证不同地区的标准得到遵循、为不同时区提供支持，并且一般 确保代码中不会存在关于使用者所在地区的假设。 您会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实际翻译的过程。 有时，本地化缩写为 *L10N* 。

Django 本身是完全国际化了的，所有的字符串均因翻译所需而被标记，并且设定了与地域无关的显示控制值，如时间和日期。 Django 是带着 40 个不同的本地化文件发行的。 即使您的母语不是英语，Django 也很有可能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异常分支指令到您的 Python 代码和模板中。 这些异常分支指令被称为* 翻译字符串* 。它们告诉 Django：这段文本应被翻译为终端用户指定的语言，如果这种语言的译文可以提供的话。

Django 会根据用户的语言偏好，在线地运用这些异常分支指令去翻译 Web 应用程序。

本质上来说，Django 做两件事情：

- 它让开发者和模板的作者指定他们的应用程序的哪些部分应该被翻译。
- Django 根据用户的语言偏好来翻译 Web 应用程序。

如果您不需要国际化:

Django 的国际化异常分支指令是默认开启的，这可能会给 Django 的运行增加一点点开销。 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'` 。

对你的 Django 应用进行国际化的三个步骤:

1. 第一步：在你的 python 代码和模板中嵌入待翻译的字符串。
2. 第二步：把那些字符串翻译成你要支持的语言
3. 第三步：在你的 django settings 文件中激活做要地区的中间件

我们将详细地对以上步骤逐一进行描述。

1、如何指定待翻译字符串

翻译字符串指定这段需要被翻译的文本。 这些字符串可以出现在您的 Python 代码和模板中。 而标记出这些翻译字符串则是您的责任；系统仅能翻译出它所知道的东西。

在 **Python** 代码中

标准翻译

使用函数 `ugettext()` 来指定一个翻译字符串。 作为惯例，使用短别名 _ 来引入这个函数以

节省键入时间.

在下面这个例子中，文本 "Welcome to my site" 被标记为待翻译字符串：

```
from django.utils.translation import ugettext as _

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

显然，你也可以不使用别名来编码。 下面这个例子和前面两个例子相同：

```
from django.utils.translation import ugettext

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

翻译字符串对于语句同样有效。 下面这个例子等同前面一种

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

翻译对变量也同样有效。 这里是一个同样的例子：

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

（以上两个例子中，对于使用变量或计算值，需要注意的一点是 Django 的待翻译字符串检测工具， make-messages.py ，将不能找到这些字符串。 稍后，在 makemessages 中会有更多讨论。

你传递给 _() 或 gettext() 的字符串可以接受占位符，由 Python 标准命名字符串插入句法指定的。 例如：

```
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    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，一段英语译文可能是 "Today is November 26." ，而一段西班牙语译文会是 "Hoy es 26 de Noviembre." 使用占位符（月份和日期）交换他们的位置。

为了这个原因，无论何时当你有多于一个单一参数时，你应当使用命名字符串插入（例如： %(day)s ）来替代位置插入（例如： %s or %d ） 如果你使用位置插入的话，翻译动作将不能重新排序占位符文本。

标记字符串为不操作

使用 django.utils.translation.gettext_noop() 函数来标记一个不需要立即翻译的字符串。 这个串会稍后从变量翻译。

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。

例如：要翻译一个模型的 help_text，按以下进行

```
from django.utils.translation import ugettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=ugettext_lazy('This is the help text'))
```

在这个例子中， ugettext_lazy() 将字符串作为惰性参照存储，而不是实际翻译。 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。

在 Python 中，无论何处你要使用一个 unicode 字符串（一个 unicode 类型的对象），您都可以使用一个 ugettext_lazy() 调用的结果。 一个 ugettext_lazy()对象并不知道如何转换它自己到一个字节串。如果你尝试在一个期待字节串的地方使用它，事情将不会像期待的那样发生。 同样，你也不能在一个字节串中使用一个 unicode 字符串。所以，这同常规的 Python 行为是一致的。 例如：

```
# This is fine: putting a unicode proxy into a unicode string.
u"Hello %s" % ugettext_lazy("people")

# This will not work, since you cannot insert a unicode object
# into a bytestring (nor can you insert our unicode proxy there)
"Hello %s" % ugettext_lazy("people")
```

如果你曾经见到到像"hello"这样的输出，你就尝试过了在一个字节串中插入 ugettext_lazy()的结果。 在您的代码中，那是一个漏洞。

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 ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

在 Django 模型中总是无一例外的使用惰性翻译。 为了翻译，字段名和表名应该被标记。（否则的话，在管理界面中它们将不会被翻译） 这意味着即使在 Meta 类中直截了当的写 verbose_nane 和 verbose_name_plural 选项，也不看着模型的类名，对 Django 的默认的 verbose_name 和 verbose_name_plural 缺省决定进行回复。

```
    from django.utils.translation import ugettext_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.ungettext()来指定以复数形式表示的消息。 例如：

```
from django.utils.translation import ungettext
```

```
def hello_world(request, count):
    page = ungettext('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）。

模板代码

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。

这个{% trans %}模板标记翻译一个常量字符串 (括以单或双引号) 或 可变内容：

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

noop 选项是当前的，变量查询还会取代但翻译会跳过。 当欠缺内容要求将来再翻译时，这很有用。

```
<title>{% trans "myvar" noop %}</title>
```

在一个带 {% trans %} 的字符串中，混进一个模板变量是不可能的。如果你的译文要求字符串带有变量(占位符 placeholders)，请使用 {% blocktrans %} ：

```
{% blocktrans %}This string 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 。

每一个 RequestContext 可以访问三个指定翻译变量：

- {{ 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。

- 作为一二字符串，`LANGUAGE_CODE` 是当前用户的优先语言。 例如： `en-us`。（请参见下面的 Django 如何发现语言偏好）
- `LANGUAGE_BIDI` 就是当前地域的说明。 如果为真（True），它就是从右向左书写的语言，例如： 希伯来语，阿拉伯语。 如果为假（False），它就是从左到右书写的语言，如： 英语，法语，德语等。

如果你不用这个 `RequestContext` 扩展，你可以用 3 个标记到那些值：

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

这些标记亦要求一个 `{% load i18n %}` 。

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") %}
```

在这种情况下，标记和过滤器两个都会看到已经翻译的字符串，所有它们并不需要提防翻译操作。

备注:

在这个例子中，翻译结构将放过字符串`"yes,no"`，而不是单独的字符串`"yes"`和`"no"`。翻译的字符串将需要包括逗号以便过滤器解析代码明白如何分割参数。 例如， 一个德语翻译器可能会翻译字符串 `"yes,no"` 为 `"ja,nein"` (保持逗号原封不动)。

与惰性翻译对象一道工作

在模型和公用函数中，使用 `ugettext_lazy()`和 `ungettext_lazy()`来标记字符串是很普遍的操作。 当你在你的代码中其它地方使用这些对象时，你应当确定你不会意外地转换它们成一个字符串，因为它们应被尽量晚的转换（以便正确的地域生效） 这需要使用及个帮助函数。

拼接字符串： **string_concat()**

标准 Python 字符串拼接(`''.join([...])`) 将不会工作在包括惰性翻译对象的列表上。 作为替代，你可以使用 `django.utils.translation.string_concat()`， 这个函数创建了一个惰性对象，其连接起它的内容 并且 仅当结果被包括在一个字符串中时转换它们为字符串 。 例如：

```
from django.utils.translation import string_concat
# ...
name = ugettext_lazy(u'John Lennon')
instrument = ugettext_lazy(u'guitar')
result = string_concat([name, ': ', instrument])
```

System Message: ERROR/3 (`<string>`, line 521)

Error in “cnid” directive: no content permitted.

```
.. cnid:: 109
```

在这种情况下，当

System Message: WARNING/2 (<string>, line 525)

Explicit markup ends without a blank line; unexpected unindent.

`result` 自己被用与一个字符串时， `result` 中的惰性翻译将仅被转换为字符串(通常在模板渲染时间)。

allow_lazy() 修饰符

Django 提供很多功能函数（如：取一个字符串作为他们的第一个参数并且对那个字符串做些什么）。(尤其在 `django.utils` 中) 这些函数被模板过滤器像在其他代码中一样直接使用。

如果你写你自己的类似函数并且与翻译打交道，当第一个参数是惰性翻译对象是，你会面临"做什么"的难题。 因为你可能在视图之外使用这个函数（并且因此当前线程的本地设置将会不正确），所以你不想立即转换其为一个字符串。

象这种情况，请使用 `django.utils.functional.allow_lazy()` 修饰符。 它修改这个函数以便 *if* 作为第一个参数被一个惰性翻译调用， 这个函数的赋值会被延后直到它需要被转化为一个字符串为止。

例如：

```
from django.utils.functional import allow_lazy

def fancy_utility_function(s, ...):
    # Do some conversion on string 's'
    # ...
fancy_utility_function = allow_lazy(fancy_utility_function, unicode)
```

`allow_lazy()` 装饰符 采用了另外的函数来装饰，以及一定量的，原始函数可以返回的特定类型的额外参数 (`*args`) 。 通常，在这里包括 `unicode` 就足够了并且确定你的函数将仅返回 Unicode 字符串。

使用这个修饰符意味着你能写你的函数并且假设输入是合适的字符串，然后在末尾添加对惰性翻译对象的支持。

2、如何创建语言文件

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。 这里就是它如何工作的。

地域限制

Django 不会支持本地化你的应用到一个连它自己都还没被翻译的地域。 在这种情况下，它将忽略你的翻译文件。 如果你想尝试这个并且 Django 支持它，你会不可避免地见到这样一个混合体--参杂着你的译文和来自 Django 自己的英文。 如果你的应用需要你支持一个 Django 中没有的地域，你将至少需要做一个 Django core 的最小翻译。

消息文件

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：

```
django-admin.py makemessages -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 项目根目录
- 您 Django 应用的根目录。
- `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） 这仅和你为 Django 自己创建一个翻译时有关。

这段脚本遍历你的项目源树或你的应用程序源树并且提取出所有为翻译而被标记的字符串。 它在 `conf/locale` 目录下创建（或更新）了一个信息文件。

作为默认， `django-admin.py makemessages` 检测每一个有 `.html` 扩展名的文件。 以备你要重载缺省值，使用 `--extension` 或 `-e` 选项指定文件扩展来检测。

```
django-admin.py makemessages -l de -e txt
```

用逗号和（或）使用`-e` 或`--extension` 来分隔多项扩展项：

```
django-admin.py makemessages -l de -e html,txt -e xml
```

当创建 JavaScript 翻译目录时，你需要使用特殊的 Django 域：**not** `-e js` 。

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。

工作在 Windows 上么？

如果你正在使用 Windows，且需要安装 GNU gettext 共用程序以便 `django-admin makemessages` 可以工作，请参看下面 Windows 小节中 gettext 部分以获得更多信息。

`.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."` 的待翻译字符串 ，像这样：

```
_("Welcome to my site.")
```

则 `django-admin.py makemessages` 将创建一个 `.po` 文件来包含以下片段的消息：

```
#: path/to/python/module.py:23
```

```
msgid "Welcome to my site."
msgstr ""
```

一个快速解释：

- 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。
- 作为方便之处，每一个消息都包括：以 # 为前缀的一个注释行并且定位上边的 msgid 行，文件名和行号。

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。

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：

```
django-admin.py makemessages -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。

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。

```
django-admin.py compilemessages
```

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。

要设定一个安装阶段的语种偏好，请设定 LANGUAGE_CODE。如果其他翻译器没有找到一个译文，Django 将使用这个语种作为缺省的翻译最终尝试。

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。 它为每一个用户定制内容。

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： 因为中间件的顺序因素，你应当跟从这些准则：

- 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。
- 如果你使用 CacheMiddleware 把 LocaleMiddleware 放在它后面。

例如， MIDDLE_CLASSES 可能会是如此：

```
MIDDLEWARE_CLASSES = (
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
)
```

（更多关于中间件的内容，请参阅第 17 章）

LocaleMiddleware 按照如下算法确定用户的语言:

- 首先，在当前用户的 session 的中查找 django_language 键；
- 如未找到，它会找寻一个 cookie
- 还找不到的话，它会在 HTTP 请求头部里查找 Accept-Language， 该头部是你的浏览器发送的，并且按优先顺序告诉服务器你的语言偏好。 Django 会尝试头部中的每一个语种直到它发现一个可用的翻译。
- 以上都失败了的话, 就使用全局的 LANGUAGE_CODE 设定值。

备注：

> 在上述每一处，语种偏好应作为字符串，以标准的语种格式出现。 例如，巴西葡萄牙语是 pt-br
>
> 如果一个基本语种存在而亚语种没有指定，Django 将使用基本语种。 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： 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： 这里是一个设置文件的样例。

```
ugettext = lambda s: s

LANGUAGES = (
    ('de', ugettext('German')),
```

```
    ('en', ugettext('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。 `ugettext()` 在运行时使用 `LANGUAGES` 的任何代中。

`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。

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。

一旦 `LocaleMiddleware` 决定用户的偏好，它会让这个偏好作为 `request.LANGUAGE_CODE` 对每一个 `HttpRequest` 有效。请随意在你的视图代码中读一读这个值。 以下是一个简单的例子：

```
def hello_world(request):
    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，对于静态翻译（无中间件）而言，此语言在 `settings.LANGUAGE_CODE` 中，而对于动态翻译（中间件），它在 `request.LANGUAGE_CODE` 中。

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。

所有的信息文件库都是以同样方式组织的： 它们是：

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$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。

您亦可运行 django-admin.py compilemessages --settings=path.to.settings 来使编译器处理所有存在于您 LOCALE_PATHS 设置中的目录。

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。

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。

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，它会以缺省名 django_language 在 cookie 中保存这个语言选择。(这个名字可以通过 LANGUAGE_COOKIE_NAME 设置来改变)

保存了语言选择后，Django 根据以下算法来重定向页面：

- Django 在 POST 数据中寻找一个 下一个 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="post">
<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>
```

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 代码并不访问 .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。

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'));
```

亦有一个 `ngettext` 接口：

```
var object_cnt = 1 // or 0, or 2, or 3, ...
s = ngettext('literal for the singular case',
        'literal for the plural case', object_cnt);
```

甚至有一个字符串插入函数：

```
function interpolate(fmt, obj, named);
```

插入句法是从 Python 借用的，所以 `interpolate` 函数对位置和命名插入均提供支持：

> 位置插入 `obj` 包括一个 JavaScript 数组对象，要素值在它们的对应于 `fmt` 的占位符中以它们出现的相同次序顺序插值 。 例如：

```
fmts = ngettext('There is %s object. Remaining: %s',
        'There are %s objects. Remaining: %s', 11);
s = interpolate(fmts, [11, 20]);
// s is 'There are 11 objects. Remaining: 20'
```

> 命名插入 通过传送为真（TRUE）的布尔参数 `name` 来选择这个模式。 `obj` 包括一个 JavaScript 对象或相关数组。 例如：

```
d = {
    count: 10
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
'there are %(count)s of a total of %(total)s objects', d.count);
s = interpolate(fmts, d, true);
```

你不应同字符串插值一起升到顶部，尽管： 这还是 JavaScript，所以这段代码不得不重复做正则表达式置换。 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

你可以像其他人一样以同样的方式创建和升级翻译目录

Django 翻译目录支持 django-admin.py makemessages 工具。 唯一的差别是需要提供一个 `-d djangojs` 的参数，就像这样：

```
django-admin.py makemessages -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。
- 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。

Windows 下的 *gettext*

对于那些要提取消息或编译消息文件(的人们来说，这就是仅需的。翻译工作本身仅仅参与编辑这个类型的现存文件，但如果你要创建你自己的消息文件，或想要测试或编译一个更改过的消息文件，你将需要这个.po)gettext 公用程序。

- 从 http://sourceforge.net/projects/gettext 下载以下 zip 文件
  - gettext-runtime-X.bin.woe32.zip
  - gettext-tools-X.bin.woe32.zip
  - libiconv-X.bin.woe32.zip
- 在同一文件夹下展开这 3 个文件。 C:\Program Files\gettext-utils）
- 更新系统路径：
  - Control Panel > System > Advanced > Environment Variables
  - 在 System variables 列表中，点击 Path，点击 Edit
  - 把;C:\Program Files\gettext-utils\bin 加到 Variable value 字段的末尾。

只要 xgettext --version 命令正常工作，你亦可使用从别处获得的 gettext 的二进制代码。 有些版本的 0.14.4 二进制代码被发现不支持这个命令。 不要试图与 Django 公用程序一起使用一个 gettext。在一个 windows 命令提示窗口输入命令 `` xgettext —version ``将导致出现一个错误弹出窗口–“xgettext.exe 产生错误并且将被 windows 关闭”。

第二十章： 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。

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： In that case, the query that the string interpolation will construct will be:

```
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？ 我们整个通讯录会被立即删除

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。

Django的数据库API帮你做了。 它会根据你所使用的数据库服务器（例如PostSQL或者MySQL）的转换规则，自动转义特殊的SQL参数。

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）。 (参考 附录 C.) 这个参数故意设计成可以接受原始的 SQL.
- 使用底层数据库 API 的查询。 (详见第十章)

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。

```
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。 You should *always* construct custom SQL this way.

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被人（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：

```
from django.http import HttpResponse

def say_hello(request):
    name = request.GET.get('name', 'world')
    return HttpResponse('<h1>Hello, %s!</h1>' % 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。 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？

```
<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。 Within a few days, he had millions of friends.

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。

To guard against this, Django’s template system automatically escapes all variable values. Let’s see what happens if we rewrite our example using the template system:

```
# views.py

from django.shortcuts import render_to_response

def say_hello(request):
    name = request.GET.get('name', 'world')
    return render_to_response('hello.html', {'name': name})

# hello.html

<h1>Hello, {{ name }}!</h1>
```

With this in place, a request to http://example.com/hello/name=<i>Jacob</i> will result in the following page:

```
<h1>Hello, &lt;i&gt;Jacob&lt;/i&gt;!</h1>
```

We covered Django’s auto-escaping back in Chapter 4, along with ways to turn it off. 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。 Both the attack itself and those tools are covered in great detail in Chapter 16.

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：

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。 She could then use that cookie to impersonate the original user.

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。 Chapter 14 explains in detail how cookies work, and one of the salient points is that it's trivial for browsers and malicious users to change cookies without your knowledge.

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。

On a more subtle level, though, it's never a good idea to trust anything stored in cookies. You never know who's been poking at them.

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。

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。 He can later log into that account and retrieve the data.

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）。 An attacker could trick a user into clicking a link to submit a color that actually contains an XSS attack. If that color isn't escaped, the user could again inject malicious code into the user's environment.

The Solution

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's session framework (see Chapter 14) simply doesn't allow sessions to be contained in the URL.

Don't store data in cookies directly. Instead, store a session ID that maps to session data stored on the backend.

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。

The Solution

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。

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。

The Solution

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。

Note

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）。

```
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。

The Solution

As we covered in Chapter 12, Django's `DEBUG` setting controls the display of these error messages. Make sure to set this to `False` when you're ready to deploy.

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。