Python Web 框架

使用 Django 和 Python 开发 Web 站点

---Python Django Web 框架，从安装到完成 Web 站点的设计

Ian Maurer (ian@itmaurer.com), 资深顾问, Brulant, Inc

Django 项目是一个定制框架，它源自一个在线新闻 Web 站点，于 2005 年以开源的形式被释放出来。Django 框架的核心组件有：

- 用于创建模型的对象关系映射
- 为最终用户设计的完美管理界面
- 一流的 URL 设计
- 设计者友好的模板语言
- 缓存系统

本文是有关 Python Web 框架的由两篇文章组成的系列文章的第一篇。第二篇文章将向您介绍 TurboGears 框架。

要使用并理解本文中提供的代码，则需要安装 Python，并了解在初学者的水平上如何使用 Python。要查看是否安装了 Python 以及 Python 的版本号，可以输入 python -V。Django 至少需要 2.3.5 版本的 Python，可以从 Python Web 站点上下载它。我们至少还应该顺便熟悉一下 MVC 架构。

安装 Django

本文使用了 Django 的开发版本，以便能够利用 Django 框架的最新改进。建议您在 0.95 版正式发布之前使用这个版本。关于最新发行版本，请参阅 Django 的 Web 站点。

按照以下步骤下载并安装 Django：

清单 1. 下载并安装 Django

~/downloads# svn co http://code.djangoproject.com/svn/django/trunk/ django_src

~/downloads# cd django_src

~/downloads# python setup.py install

Django 管理工具

在安装 Django 之后，您现在应该已经有了可用的管理工具 django-admin.py。清单 2 给出了这个管理工具中可以使用的一些命令：

清单 2. 使用 Django 管理工具

```
~/dev$ django-admin.py

usage: django-admin.py action [options]

actions:

  adminindex [modelmodule ...]

    Prints the admin-index template snippet for the given model

    module name(s).


  ... snip ...


  startapp [appname]

    Creates a Django app directory structure for the given app name

    in the current directory.
```

```
  startproject [projectname]
    Creates a Django project directory structure for the given
    project name in the current directory.

  validate
    Validates all installed models.

options:
  -h, --help            show this help message and exit
  --settings=SETTINGS   Python path to settings module, e.g.
                        "myproject.settings.main". If this isn't
                        provided, the DJANGO_SETTINGS_MODULE
                        environment variable will be used.
  --pythonpath=PYTHONPATH
                        Lets you manually add a directory the Python
                        path, e.g. "/home/djangoprojects/myproject".
```

Django 项目和应用程序

要启动 Django 项，请使用 django-admin startproject 命令，如下所示：

清单 3. 启动项目

```
~/dev$ django-admin.py startproject djproject
```

上面这个命令会创建一个 djproject 目录，其中包含了运行 Django 项目所需要的基本配置文件：

清单 4. djproject 目录的内容

```
__init__.py

manage.py

settings.py

urls.py
```

对于这个项目来说，我们要构建一个职位公告板应用程序 “jobs”。要创建应用程序，可以使用 manage.py 脚本，这是一个特定于项目的 django-admin.py 脚本，其中 settings.py 文件可以自动提供：

清单 5. 使用 manage.py startapp

```
~/dev$ cd djproject

~/dev/djproject$ python manage.py startapp jobs
```

这将创建一个应用程序骨架，其中模型有一个 Python 模块，视图有另外一个 Python 模块。jobs 目录中包含以下文件：

清单 6. jobs 应用程序目录中的内容

__init__.py

models.py

views.py

提供应用程序在项目中的位置纯粹是为新 Django 开发人员建立的一种惯例，并不是必需的。一旦开始在几个项目中混合使用应用程序，就可以将应用程序放到自己的命名空间中，并使用设置和主 URL 文件将它们绑定在一起。现在，请按照下面给出的步骤执行操作。

为了使 Django 认识到新应用程序的存在，还需要向 settings.py 文件中的 INSTALLED_APPS 添加一个条目。对于这个职位公告板应用程序来说，我们必须添加字符串 djproject.jobs：

清单 7. 向 settings.py 中添加一个条目

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

```
    'djproject.jobs',
)
```

创建一个模型

Django 提供了自己的对象关系型数据映射组件（object-relational mapper，ORM）库，它可以通过 Python 对象接口支持动态数据库访问。这个 Python 接口非常有用，功能十分强大，但如果需要，也可以灵活地不使用这个接口，而是直接使用 SQL。

ORM 目前提供了对 PostgreSQL、MySQL、SQLite 和 Microsoft® SQL 数据库的支持。

这个例子使用 SQLite 作为后台数据库。SQLite 是一个轻量级数据库，它不需要进行任何配置，自身能够以一个简单文件的形式存在于磁盘上。要使用 SQLite，可以简单地使用 setuptools 来安装 pysqlite：

```
easy_install pysqlite
```

在使用这个模型之前，需要在设置文件中对数据库进行配置。SQLite 只需要指定数据库引擎和数据库名即可。

清单 8. 在 settings.py 中配置数据库

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = '/path/to/dev/djproject/database.db'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

这个职位公告板应用程序有两种类型的对象：Location 和 Job。Location 包含 city、state（可选）和 country 字段。Job 包含 location、title、description 和 publish date 字段。

清单 9. jobs/models.py 模块

```
from django.db import models


class Location(models.Model):
    city = models.CharField(maxlength=50)
    state = models.CharField(maxlength=50, null=True, blank=True)
    country = models.CharField(maxlength=50)

    def __str__(self):
        if self.state:
            return "%s, %s, %s" % (self.city, self.state, self.country)
        else:
            return "%s, %s" % (self.city, self.country)

class Job(models.Model):
    pub_date = models.DateField()
    job_title = models.CharField(maxlength=50)
    job_description = models.TextField()
    location = models.ForeignKey(Location)
```

```
    def __str__(self):
        return "%s (%s)" % (self.job_title, self.location)
```

__str__ 方法是 Python 中的一个特殊类，它返回对象的字符串表示。Django 在 Admin 工具中显示对象时广泛地使用了这个方法。

要设置这个模型的模式，请返回 manage.py 的 sql 命令。此时模式尚未确定。

清单 10. 使用 manage.py sql 命令查看数据库模式

```
~/dev/djproject$ python manage.py sql jobs

BEGIN;
CREATE TABLE "jobs_job" (
    "id" integer NOT NULL PRIMARY KEY,
    "pub_date" date NOT NULL,
    "job_title" varchar(50) NOT NULL,
    "job_description" text NOT NULL,
    "location_id" integer NOT NULL
);
CREATE TABLE "jobs_location" (
    "id" integer NOT NULL PRIMARY KEY,
    "city" varchar(50) NOT NULL,
```

```
    "state" varchar(50) NULL,
    "country" varchar(50) NOT NULL
);
COMMIT;
```

为了初始化并安装这个模型，请运行数据库命令 syncdb：

```
~/dev/djproject$ python manage.py syncdb
```

注意，syncdb 命令要求我们创建一个超级用户帐号。这是因为 django.contrib.auth 应用程序（提供基本的用户身份验证功能）默认情况下是在 INSTALLED_APPS 设置中提供的。超级用户名和密码用来登录将在下一节介绍的管理工具。记住，这是 Django 的超级用户，而不是系统的超级用户。

查询集

Django 模型通过默认的 Manager 类 objects 来访问数据库。例如，要打印所有 Job 的列表，则应该使用 objects 管理器的 all 方法：

清单 11. 打印所有的职位

```
>>> from jobs.models import Job
>>> for job in Job.objects.all():
...     print job
```

Manager 类还有两个过滤方法：一个是 filter，另外一个是 exclude。过滤方法可以接受满足某个条件的所有方法，但是排除不满足这个条件的其他方法。下面的查询应该可以给出相同的结果（“gte” 表示 “大于或等于”，而 “lt” 表示 “小于”）。

清单 12. 排除和过滤职位

```
>>> from jobs.models import Job

>>> from datetime import datetime

>>> q1 = Job.objects.filter(pub_date__gte=datetime(2006, 1, 1))

>>> q2 = Job.objects.exclude(pub_date__lt=datetime(2006, 1, 1))
```

filter 和 exclude 方法返回一些 QuerySet 对象，这些对象可以链接在一起，甚至可以执行连接操作。下面的 q4 查询会查找从 2006 年 1 月 1 日开始在俄亥俄州的 Cleveland 张贴的职位：

清单 13. 对职位进行更多的排除和过滤

```
>>> from jobs.models import Job

>>> from datetime import datetime

>>> q3 = Job.objects.filter(pub_date__gte=datetime(2006, 1, 1))

>>> q4 = q3.filter(location__city__exact="Cleveland",

...                location__state__exact="Ohio")
```

QuerySets 是惰性的，这一点非常不错。这意味着只在对数据库进行求值之后才会对它们执行查询，这会比立即执行查询的速度更快。

这种惰性利用了 Python 的*分片（slicing）*功能。下面的代码并没有先请求所有的记录，然后对所需要的记录进行分片，而是在实际的查询中使用了 5 作为 OFFSET、10 作为 LIMIT，这可以极大地提高性能。

清单 14. Python 分片

```
>>> from jobs.models import Job

>>> for job in Job.objects.all()[5:15]

...         print job
```

注意：使用 count 方法可以确定一个 QuerySet 中有多少记录。Python 的 len 方法会进行全面的计算，然后统计那些以记录形式返回的行数，而 count 方法执行的则是真正的 SQL COUNT 操作，其速度更快。我们这样做，数据库管理员会感激我们的。

清单 15. 统计记录数

```
>>> from jobs.models import Job

>>> print "Count = ", Job.objects.count()            # GOOD!

>>> print "Count = ", len(Job.objects.all())       # BAD!
```

管理员工具

Django 的最大卖点之一是其一流的管理界面。这个工具是按照最终用户的思路设计的。它为我们的项目提供了很多数据输入工具。

管理工具是 Django 提供的一个应用程序。与 jobs 应用程序一样，在使用之前也必须进行安装。第一个步骤是将应用程序的模块（django.contrib.admin）添加到 INSTALLED_APPS 设置中：

清单 16. 修改 settings.py

```
INSTALLED_APPS = (

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.sites',

    'djproject.jobs',

    'django.contrib.admin',

)
```

要让该管理工具可以通过 /admin URL 使用，只需要简单地取消项目的 urls.py 文件中提供的对应行的内容即可。下一节将详细介绍 URL 的配置。

清单 17. 使管理工具可以通过 urls.py 使用

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


urlpatterns = patterns('',

    (r'^admin/', include('django.contrib.admin.urls.admin')),

)
```

这个管理应用程序有自己的数据库模型，但也需要进行安装。我们可以再次使用 syncdb 命令来完成这个过程：

python manage.py syncdb

要查看这个管理工具，可以使用 Django 提供的测试服务器。

清单 18. 使用测试服务器来查看管理工具

~/dev/djproject$ python manage.py runserver

Validating models...

0 errors found.

Django version 0.95 (post-magic-removal), using settings 'djproject.settings'

Development server is running at http://127.0.0.1:8000/

Quit the server with CONTROL-C (Unix) or CTRL-BREAK (Windows).

现在可以使用 http://localhost:8000/admin 启动管理工具，并使用前面创建的超级用户帐号进行登录。我们注意到现在还没有可用的模块。

要让一个类可以通过管理工具进行访问，我们需要为其创建一个 Admin 子类。然后可以通过为这个子类添加类属性来定制如何对每个类进行管理。清单 19 展示了如何将 Location 类添加到这个管理工具中。

清单 19. 使用管理工具添加 Location 类

class Location(meta.Model):

    ...

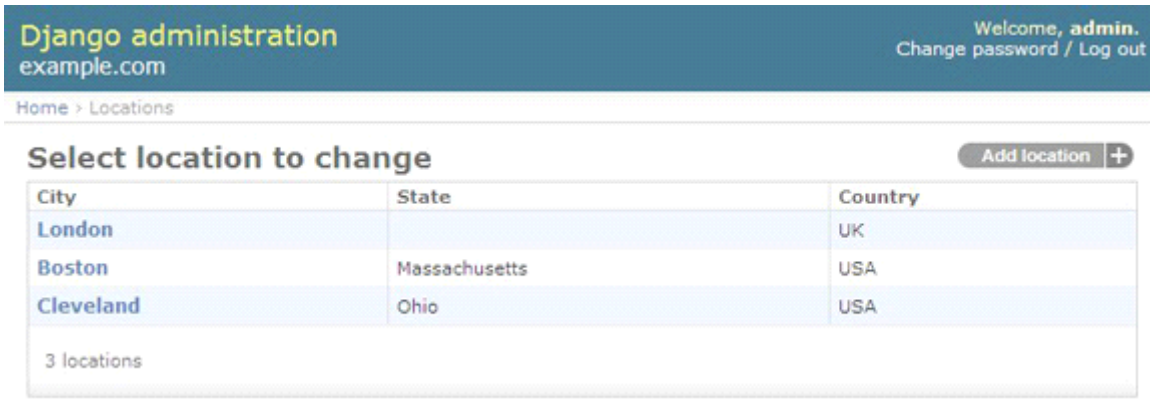
```
class Admin:

    list_display = ("city", "state", "country")
```

现在就可以通过管理界面来创建、更新和删除 Location 记录了。

图 1. 使用管理工具编辑位置

可以按照 list_display 类的属性指定的城市、州和国家来列出记录并对它们进行排序。

图 2. 使用管理工具显示位置



管理工具有无数用来管理每种模块类的选项。清单 20 给出了几个适用于 Job 类的例子：

清单 20. 管理模块类的选项

```
class Job(meta.Model):

    ...

    class Admin:

    list_display = ("job_title", "location", "pub_date")

    ordering = ["-pub_date"]

    search_fields = ("job_title", "job_description")

    list_filter = ("location",)
```

根据以上设置，职位的标题、位置和发布日期都会在显示职位记录时用到。职位可以按照发布时间进行排序，最开始是最近发布的职位（减号表示降序）。用户可以按照标题和说明来查找职位，管理员可以根据位置对记录进行过滤。

图 3. 使用管理工具显示职位

设计 URL 方案

Django URL 分发系统使用了正则表达式配置模块，它可以将 URL 字符串模式映射为 Python 方法 *views*。这个系统允许 URL 与底层代码完全脱节，从而实现最大的控制和灵活性。

urls.py 模块被创建和定义成 URL 配置的默认起点（通过 settings.py 模块中的 ROOT_URLCONF 值）。URL 配置文件的惟一要求是必须包含一个定义模式 urlpatterns 的对象。

这个职位公告板应用程序会在启动时打开一个索引和一个详细视图，它们可以通过以下的 URL 映射进行访问：

- /jobs 索引视图：显示最近的 10 个职位
- /jobs/1详细视图：显示 ID 为 1 的职位信息

这两个视图（索引视图和详细视图）都是在这个 jobs 应用程序的 views.py 模块中实现的。在项目的 urls.py 文件中实现这种配置看起来如下所示：

清单 21. 在 djproject/urls.py 中实现视图的配置

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

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

    (r'^jobs/$', 'djproject.jobs.views.index'),
    (r'^jobs/(?P<job_id>\d+)/$', 'djproject.jobs.views.detail'),
)
```

注意 <job_id> 部分，这在后面非常重要。

最佳实践是提取出应用程序特有的 URL 模式，并将它们放入应用程序自身中。这样可以取消应用程序与项目的耦合限制，从而更好地实现重用。jobs 使用的应用程序级的 URL 配置文件如下所示：

清单 22. 应用程序级的 URL 配置文件 urls.py

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

urlpatterns = patterns('',
    (r'^$', 'djproject.jobs.views.index'),
    (r'^(?P<job_id>\d+)/$', 'djproject.jobs.views.detail'),
```

)

由于 view 方法现在都是来自同一个模块，因此第一个参数可以使用这个模块的根名称来指定 djproject.jobs.views，Django 会使用它来查找 index 方法和 detail 方法：

清单 23. jobs/urls.py：查找 index 和 detail 方法

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

urlpatterns = patterns('djproject.jobs.views',
    (r'^$', 'index'),
    (r'^(?P<object_id>\d+)/$', 'detail'),
)
```

尝试上面的 jobs URL 会返回到这个项目中，因为它们是使用 include 函数将其作为一个整体来实现的。应用程序级的 URL 被绑定到下面的 /jobs 部分：

清单 24. djproject/urls.py：将 URL 送回该项目

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

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

```
    (r'^jobs/', include('djproject.jobs.urls')),

)
```

如果现在尝试使用测试服务器来访问索引页（http://localhost:8000/jobs），会得到一个错误，因为正在调用的视图（djproject.jobs.views.index）不存在。

实现视图

视图是一个简单的 Python 方法，它接受一个请求对象，负责实现：

- 任何业务逻辑（直接或间接）
- 上下文字典，它包含模板数据
- 使用一个上下文来表示模板
- 响应对象，它将所表示的结果返回到这个框架中

在 Django 中，当一个 URL 被请求时，所调用的 Python 方法称为一个*视图（view）*，这个视图所加载并呈现的页面称为*模板（template）*。由于这个原因，Django 小组将 Django 称为一个 MVT（model-view-template）框架。另一方面，TurboGears 把自己的方法称作*控制器（controller）*，将所呈现的模板称为*视图（view）*，因此缩写也是 MVC。其区别在于广义的语义，因为它们所实现的内容是相同的。

最简单的视图可能会返回一个使用字符串初始化过的 HttpResponse 对象。创建下面的方法，并生成一个 /jobs HTTP 请求，以确保 urls.py 和 views.py 文件都已经正确设置。

清单 25. jobs/views.py (v1)

```
from django.utils.httpwrappers import HttpResponse


def index(request):

    return HttpResponse("Job Index View")
```

下面的代码将获取最近的 10 个职位，并通过一个模板呈现出来，然后返回响应。没有 下一节 中的模板文件，这段代码就*无法* 正常工作。

清单 26. jobs/views.py (v2)

```
from django.template import Context, loader

from django.http import HttpResponse

from jobs.models import Job
```


```
from django.template import Context, loader

from django.http import HttpResponse

from jobs.models import Job
```


```
def index(request):

    object_list = Job.objects.order_by('-pub_date')[:10]

    t = loader.get_template('jobs/job_list.html')

    c = Context({

        'object_list': object_list,

    })

    return HttpResponse(t.render(c))
```

在上面的代码中，模板是由 jobs/job_list.html 字符串进行命名的。该模板是使用名为 object_list 的职位列表的上下文呈现的。所呈现的模板字符串随后被传递到 HTTPResponse 构造器中，后者通过这个框架被发送回请求客户机那里。

加载模板、创建内容以及返回新响应对象的步骤在下面都被 render_to_response 方法取代了。新增内容是详细视图方法使用了一个 get_object_or_404 方法，通过该方法使用所提供的参数获取一个 Job 对象。如果没有找到这个对象，就会触发 404 异常。这两个方法减少了很多 Web 应用程序中的样板代码。

清单 27. jobs/views.py (v3)

```
from django.shortcuts import get_object_or_404, render_to_response

from jobs.models import Job


def index(request):

    object_list = Job.objects.order_by('-pub_date')[:10]

    return render_to_response('jobs/job_list.html',

                                        {'object_list': object_list})


def detail(request, object_id):

    job = get_object_or_404(Job, pk=object_id)

    return render_to_response('jobs/job_detail.html',

                                        {'object': job})
```

注意，detail 使用 object_id 作为一个参数。这是前面提到过的 jobs urls.py 文件中 /jobs/ URL 路径后面的数字。它以后会作为主键（pk）传递给 get_object_or_404 方法。

上面的视图仍然会失败，因为它们所加载和呈现的模板（jobs/job_list.html and jobs/job_detail.html）不存在。

创建模板

Django 提供了一种模板语言，该语言被设计为能够快速呈现且易于使用。Django 模板是利用 {{ variables }} 和 {% tags %} 中嵌入的文本创建的。变量会使用它们表示的值进行计算和替换。标记用来实现基本的控制逻辑。模板可以用来生成任何基于文本的格式，包括 HTML、XML、CSV 和纯文本。

第一个步骤是定义将模板加载到什么地方。为了简便起见，我们需要在 djproject 下面创建一个 templates 目录，并将这个路径添加到 settings.py 的 TEMPLATE_DIRS 条目中：

清单 28. 在 settings.py 中创建一个 templates 目录

```
TEMPLATE_DIRS = (
    '/path/to/devdir/djproject/templates/',
)
```

Django 模板支持称为*模板继承（template inheritance）*的概念，它允许站点设计人员创建一个统一的外表，而不用替换每个模板的内容。我们可以通过使用块标记定义骨干文档或基础文档来使用继承。这些块标记都是使用一些包含内容的页面模板来填充的。这个例子给出了一个包含称为 title、extrahead 和 content 的块的 HTML 骨干：

清单 29. 骨干文档 templates/base.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Company Site: {% block title %}Page{% endblock %}</title>
```

```
    {% block extrahead %}{% endblock %}
  </head>
  <body>
    {% block content %}{% endblock %}
  </body>
</html>
```

为了取消应用程序与项目之间的耦合，我们使用了一个中间基本文件作为 Job 应用程序所有页面文件的基础。对于这个例子来说，为了简便起见，我们将应用程序的 CSS 放到这个基本文件中。在实际的应用程序中，需要有一个正确配置的 Web 服务器，将这个 CSS 提取出来，并将其放到 Web 服务器所服务的静态文件中。

清单 30. 中间基础文件 templates/jobs/base.html

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

{% block extrahead %}
    <style>
        body {
            font-style: arial;
        }
        h1 {
            text-align: center;
        }
```

```
        .job .title {
            font-size: 120%;
            font-weight: bold;
        }
        .job .posted {
            font-style: italic;
        }
    </style>
{% endblock %}
```

默认情况下，Django 测试服务器并不会为静态文件提供服务，因为这是 Web 服务器的工作。但是在开发过程中，如果您希望 Django 可以提供图像、样式表等，那么请参阅 参考资料 中有关如何激活这个特性的链接。

现在我们要创建视图所加载并呈现的两个页面模板。jobs/job_list.html 模板简单地循环遍历 *object_list*，它通过索引视图遍历其内容，并显示一个到每条记录的详细页面的链接。

清单 31. templates/jobs/job_list.html 模板

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

{% block title %}Job List{% endblock %}

{% block content %}
    <h1>Job List</h1>
```

```
    <ul>
    {% for job in object_list %}
        <li><a href="{{ job.id }}">{{ job.job_title }}</a></li>
    {% endfor %}
    </ul>
{% endblock %}
```

jobs/job_detail.html 页面会显示一条称为 *job* 的记录：

清单 32. templates/jobs/job_detail.html 页面

```
{% extends "jobs/base" %}

{% block title %}Job Detail{% endblock %}

{% block content %}
    <h1>Job Detail</h1>

    <div class="job">
        <div class="title">
            {{ job.job_title }}
            -
```

```
            {{ job.location }}
        </div>
        <div class="posted">
            Posted: {{ job.pub_date|date:"d-M-Y" }}
        </div>
        <div class="description">
            {{ job.job_description }}
        </div>
    </div>
{% endblock %}
```

Django 模板语言已经被设计为只能实现有限的功能。这种限制可以为非程序员保持模板的简单性，同时还可以让程序员不会将业务逻辑放到不属于自己的地方，即表示层。

通用视图

Django 提供了 4 种*通用视图（generic view）*，它们可以让开发人员创建遵循典型模式的应用程序：

- 页面列表/详细页面（与上面的例子类似）
- 基于数据的记录分类（对于新闻或 blog 站点非常有用）
- 对象的创建、更新和删除（CRUD）
- 简单直接的模板表示或简单地对 HTTP 重新进行定向

我们没有创建样板视图方法，而是将所有的业务逻辑都放入了 urls.py 文件中，它们都由 Django 提供的通用视图进行处理。

清单 33. jobs/urls.py 中的通用视图

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

from jobs.models import Job



info_dict = {

    'queryset': Job.objects.all(),

}



urlpatterns = patterns('django.views.generic.list_detail',

    (r'^$', 'object_list', info_dict),

    (r'^(?P<object_id>\d+)/$', 'object_detail', info_dict),

)
```

这个 urls.py 文件中的 3 个主要变化如下：

- info_dict 映射对象会为要访问的 Job 提供一个查询集。
- 它使用了 django.views.generic.list_detail，而不是 djproject.jobs.views。
- 真正的视图调用是 object_list 和 object_detail。

这个项目需要遵循一些要求才能让通用视图自动工作：

- 通用详细视图期望获得一个 object_id 参数。
- 模板遵循下面的命名模式：*app_label*/*model_name*_list.html (jobs/job_list.html) *app_label*/*model_name*_detail.html (jobs/job_detail.html)
- 列表模板处理一个名为 object_list 的列表。
- 详细模板处理一个名为 object 的对象。

更多选项可以通过 info_dict 来传递，其中包括指定每个页面中对象个数的 paginate_by 值。