

Django 实战

by ThinkInside

ranvane 整理

ranvane 的话:

无意中发现这个系列的文章，粗读了一下，感觉十分好，于是动了保存的念头，打开 word，copy&粘贴下来，并大概整理了一下。文章来自 ThinkInside 的 blog，我是从 csdn 上面 copy'的，现在 ThinkInside 的 blog 已经搬家到了 cnblogs 上，也有。如果大家对我整理的这份不满意可以去看原版。

ThinkInside 's blog :

<http://www.cnblogs.com/holbrook/tag/django/>

<http://blog.csdn.net/ThinkInside/article/category/1068327/1>

目录

1、需求分析和设计	2
2、创建第一个模型类.....	6
3、Django 也可以有 scaffold	10
4、scaffold 生成物分析.....	13
5、引入 bootstrap，设置静态资源	16
6、对比 RoR 和 Django 的模板系统	19
7、-改造 ProductList 界面.....	21
8、对比 RoR 与 Django 的输入校验机制	24
9、实现 Product 的输入校验	26

10、单元测试.....	30
11、修改 Model 类	37
12、增加目录页，设定统一布局	39
13、在 session 中保存购物车	43
14、让页面联动起来	48
15、Django 实现 RESTful web service	50
16、Django+jquery	53
17、ajax !.....	57
18、提交订单.....	64
19、自定义 many-to-many 关系，实现 Atom 订阅	66
20、分页 (Pagination).....	70
21、使用内置的 Amin 管理用户	72
22、处理登录和注销	74
23、权限控制.....	76
24、总结	77

1、需求分析和设计

Depot 是《Agile Web Development with Rails》中的一个购物车应用。

该书中用多次迭代的方法，逐步实现购物车应用，使很多人走上了 rails 开发的道路。

遗憾的是 Django 世界中好像没有类似的指引，也许是因为 pythoner 不需要具体的例子。

但是如果通过这样一个例子能够让更多的人加入 pythoner 的队伍，也是一大幸事。

本文首先回顾一下 depot 的需求，在后续内容中将会按照《Agile Web Development with Rails》中的迭代进度，逐步用 Django 实现 depot 购物车应用。

在原例子的基础上，还会增加一些新的内容，以适应企业级应用的开发。

同时，会尽量展示敏捷开发的特性。

原书中，初始阶段的需求整理如下：

角色

买方，卖方。

用例

买方：浏览产品，创建订单

卖方：管理产品，处理订单，管理发货

界面设计

买方界面包括：

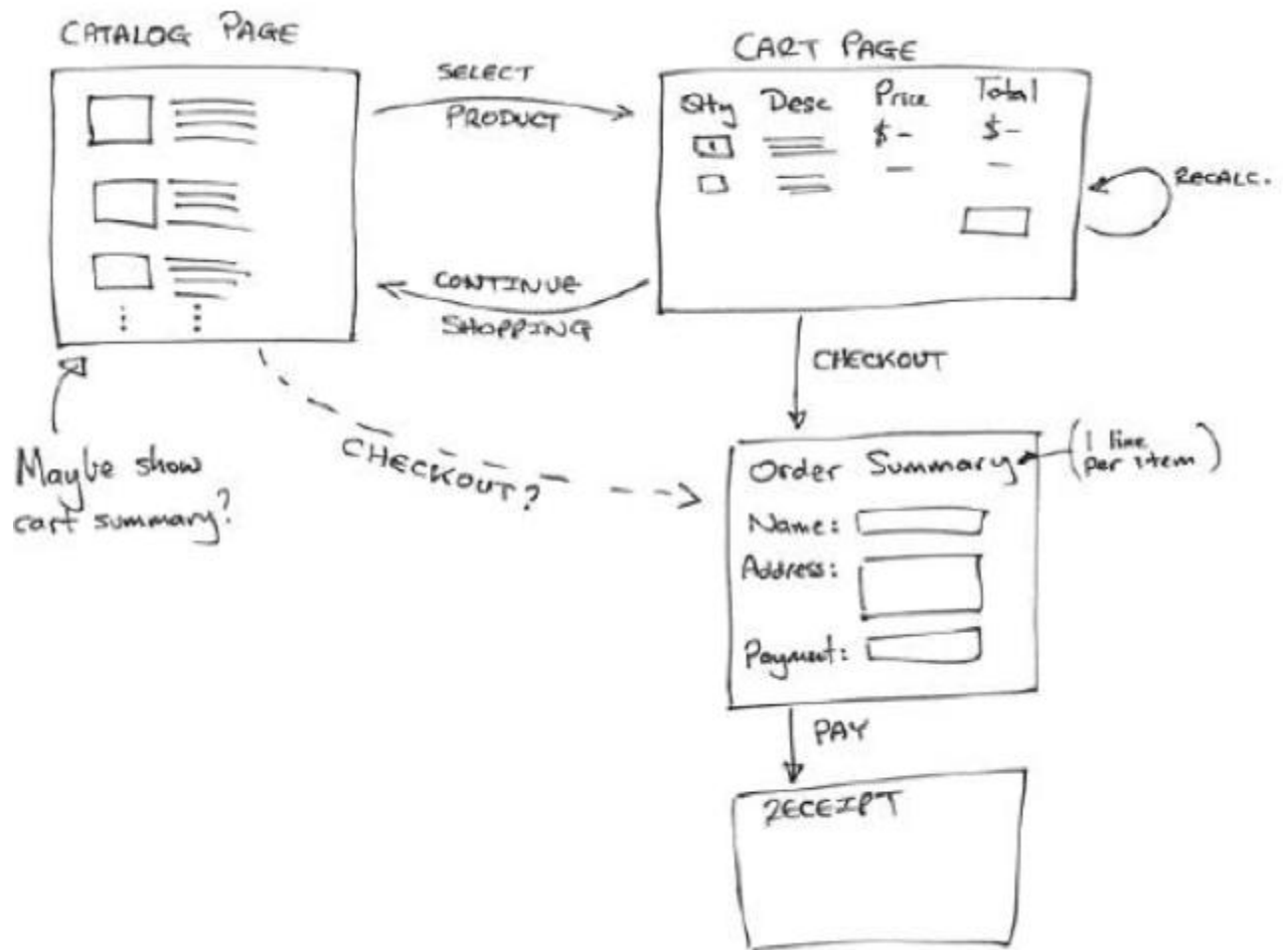
目录页：可以选择一个产品，选中产品会打开购物车页，同时该产品被加入购物车

购物车页：显示所有已选择的产品，可以返回分类页，也可以进入支付页进行支付

订单页：填写一些要素信息，确认支付后显示收据页

收据页：通知买方订单已被接收

买方界面流程如下图所示：



卖方界面包括：

登录页：卖方要登录后才能使用系统，登录后通过菜单选择其要使用的功能

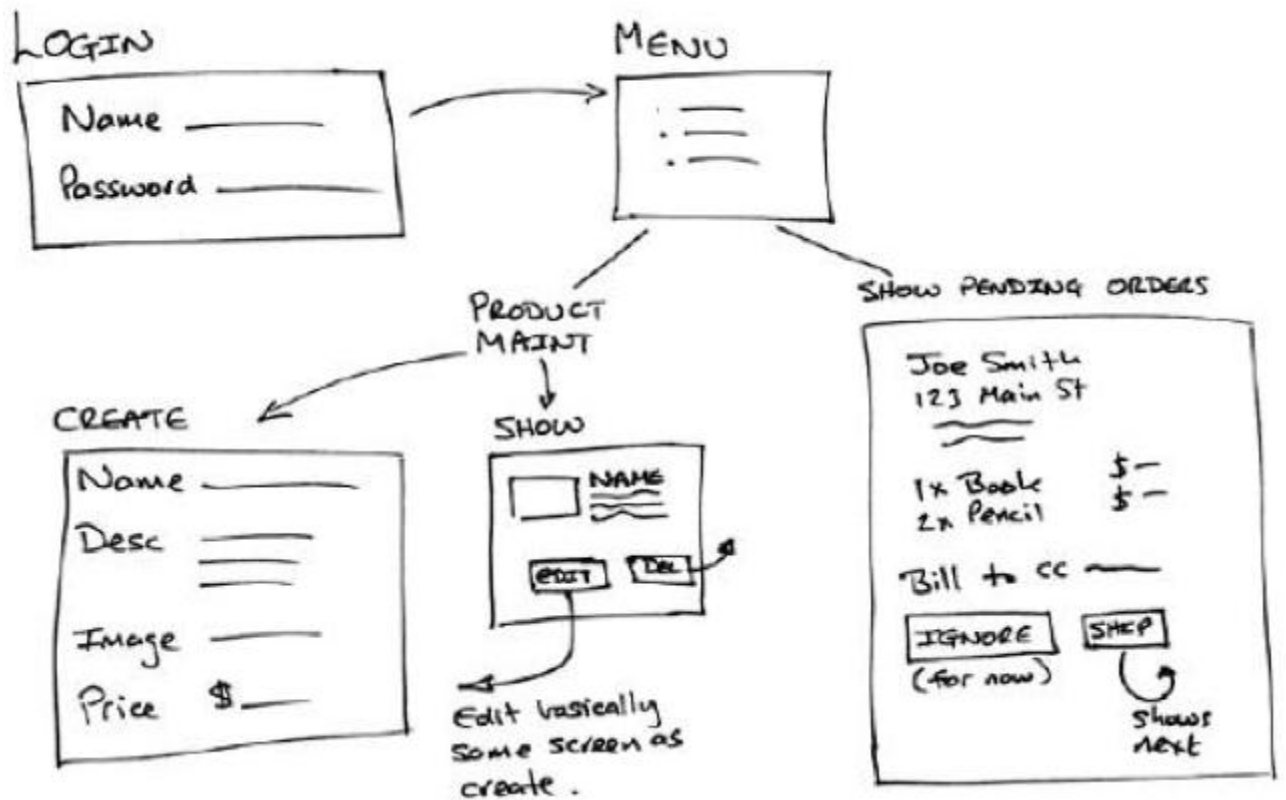
菜单页：选择维护产品或者查看订单

创建产品页：用于加入新的产品

产品信息页：显示已经加入的产品，可以进行修改或者删除

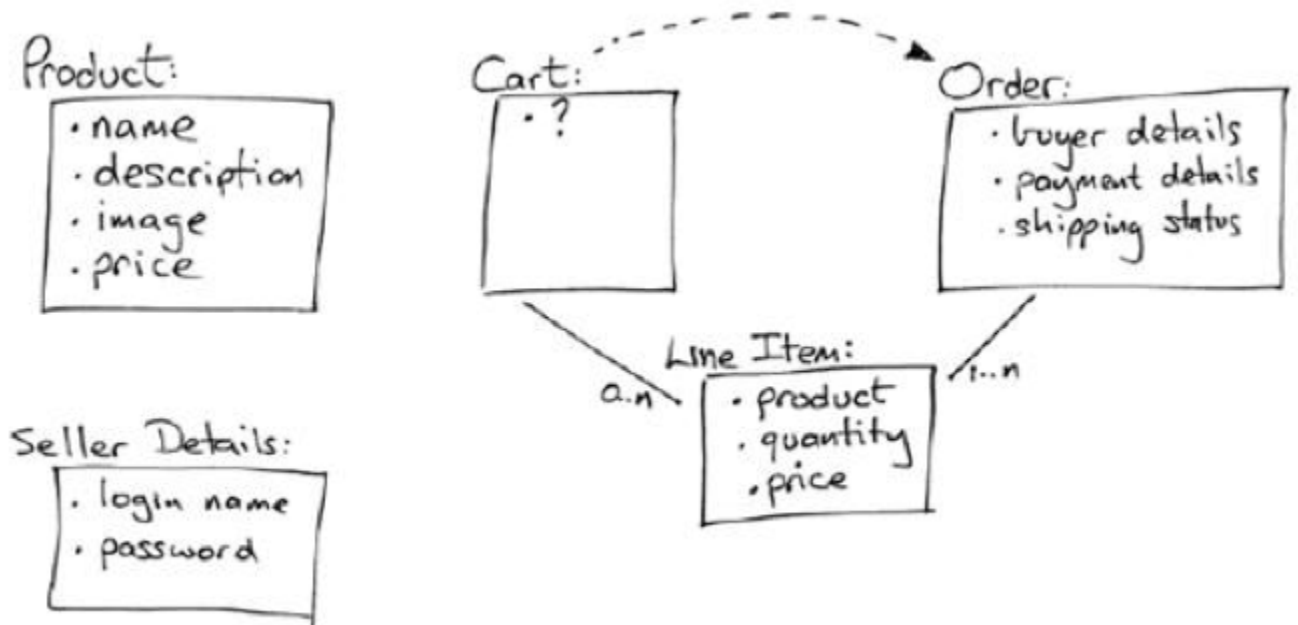
订单页：显示订单信息，可以忽略或者处理

卖方界面流程如下图所示：



领域模型

从界面设计中可以很容易得出初步的模型，如下图：



以上，就是开始阶段所能得到的“需求”。尽管其中还有一些不确定的因素，但是敏捷方法认为应该尽快开始开发，这些不确定的因素会在后续的迭代过程中逐步明确。

接下来，就可以开始第一轮迭代开发了。

2、创建第一个模型类

从模型开始开发似乎是个好主意。一方面模型是整个应用的核心，实现了应用的业务数据和对业务数据进行操作的约束，而视图和模板只是向用户提供操作和展现这些数据的界面；另一方面模型相对于系统的其他部分更加稳定，将模型先确定下来有助于系统其他部分的实现。DDD（领域驱动设计）更进一步将模型中的核心对象抽取出来作为“领域模型”。

从 Depot 应用来看，产品（Product）应该是模型中的核心对象之一。就让我们先来实现 Product 模型。

创建 app

我们可以从《[Django 第一步](#)》中实现的工程开始。在继续之前，还要进行一些准备工作。

Django 约定必须要创建 app 才能使用模型。这也是 Django 的哲学之一：

Django 认为一个 project 包含很多个 Django appl；project 提供配置文件，比如数据库连接信息、安装的 app 清单、模板路径等等；而一个 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。

app 可以在多个 project 之间很容易的复用。比如 Django 自带的注释系统和自动管理界面。

所以我们在原有工程的基础上还需要创建一个 app。现在假设我们只需要一个 app，并将其命名为 depotapp。创建应用的脚本也是使用 project 目录下的 managy.py：

```
$python manage.py startapp depotapp
```

就会在工程目录下创建一个 depotapp 目录：

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

用 python 代码定义数据库

在《Django 的第一印象》中介绍过，Django 的设计是以 Python 类的形式定义数据模型。之所以没有采用 rails 的运行时自动获取数据库 schema 的“魔术方式”，是出于以下的考虑：

1. 效率。运行时扫描数据库可能会带来性能问题。
2. 明确性。只通过 Model 类就完全知道数据库中有哪些字段，而不需要再切换到 migration 或 schema 文件中去查看，更不需要去查看数据库结构。
3. 一致性。你看到的只是 Python 代码，完全不需要将大脑切换到“数据库模式”，能极大提高开发效率。
4. 版本控制。rails 中的数据库结构版本保存在一个个的 migration 文件中，这简直就是版本管理的“反模式”。Django 的方式是管理 Model 代码文件的版本。
5. 可扩展性。可以定义数据库中不存在的“字段类型”。比如 Email，URL，等等。当然，Django 也提供从现有数据库表中自动扫描生成模型的工具。

so，《Agile Web Development with Rails》中的做法是先创建数据库表：

```
[sql] view plaincopy

drop table if exists products;

create table products (

id int not null auto_increment,

title varchar(100) not null,

description text not null,

image_url varchar(200) not null,

price decimal(10,2) not null,

primary key (id)

);
```

然后再生成 scaffold（包括 model，controller，test，4 个 views 等等）。

而 Django 的做法是，编写下面的 Model 类：

```
[python] view plaincopy

depot/depotapp/models.py:

from django.db import models
```

```
class Product(models.Model):

    title      = models.CharField(max_length=100)

    description = models.TextField()

    image_url   = models.CharField(max_length=200)

    price       = models.DecimalField(max_digits=8,decimal_places=2)
```

如同其他的 ORM，ID 字段是默认声明的，不需要单独处理。

部署模型

Django 中的每一件事情都需要明确声明，也就是说，没有你的允许，Django 不会主动去碰你的代码。所以我们还需要在 project 中进行一些配置工作才能让 app 生效。不过这样的配置只需要做一次。

首先要创建数据库并配置整个 project 的数据库连接，为了简单起见，使用 sqlite 数据库。

在工程文件夹下创建 db 文件夹和 sqlite 数据库文件：

```
$mkdir db
$cd db
$sqlite3 development.sqlite3
```

然后修改配置文件 settings.py, 将 DATABASES 改为：

[python] view plaincopy

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': 'db/development.sqlite3',

        'USER': '',

        'PASSWORD': '',

        'HOST': '',

        'PORT': '',

    }

}
```

就完成了数据库的配置。

还需要配置 project 让 depotapp 生效，还是在 settings.py 中，将 INSTALLED_APPS 改

为：

[python] view plaincopy

```
INSTALLED_APPS = (  
    # 'django.contrib.auth',  
    # 'django.contrib.contenttypes',  
    # 'django.contrib.sessions',  
    # 'django.contrib.sites',  
    # 'django.contrib.messages',  
    # 'django.contrib.staticfiles',  
  
    # Uncomment the next line to enable the admin:  
    # 'django.contrib.admin',  
    # Uncomment the next line to enable admin documentation:  
    # 'django.contrib.admindocs',  
    'depot.depotapp',  
)
```

接下来就可以使用模型了。先验证一下：

```
$python manage.py validate  
0 errors found
```

然后可以看一下这个 Model 将会生成什么样的数据库：

```
$ python manage.py sqlall depotapp  
BEGIN;  
CREATE TABLE "depotapp_product" (  
    "id" integer NOT NULL PRIMARY KEY,  
    "title" varchar(100) NOT NULL,  
    "description" text NOT NULL,  
    "image_url" varchar(200) NOT NULL,  
    "price" decimal NOT NULL  
)  
;  
COMMIT;
```

最后，将模型导入数据库：

```
$ python manage.py syncdb  
Creating tables ...  
Creating table depotapp_product
```

Installing custom SQL ...
Installing indexes ...
No fixtures found.

至此，完成了第一个模型类的创建。

3、Django 也可以有 scaffold

rails 有一个无用的“神奇”功能，叫做 scaffold。能够在 model 基础上，自动生成 CRUD 的界面。

说它无用，是因为 rails 的开发者 David 说，scaffold“不是应用程序开发的目的。它只是在我们构建应用程序时提供支持。当你设计出产品的列表该如何工作时，你依赖于“支架”“生成器”产生创建,更新,和删除的行为。然后在保留这个“动作”时你要替换由“生成器”生成的行为。有时候当你需要一个快速接口时,并且你并不在乎界面的丑陋,“支架”就足够用了。不要指望 scaffold 能满足你程序的所有需要”。

说它神奇，是因为在 rails 中你不清楚他是怎么实现的。只告诉你一句话：约定优先于配置。只要名字 xxx，就会 xxx。说得人云里雾里，认为 rails 真是一个伟大的框架。

在 Django 的世界中没有这种无用的东西。但是如果你一定要，可以很容易地创建这么一套东西。下面我们就在 project 中引入一个“插件”。前面说过，[app 可以在多个 project 之间很容易的复用](#)，我们要引入的就是一个第三方的 app，无需修改，只需要简单配置即可使用。

这个 app 叫做 [django-groundwork](#)。它不实现具体的功能，而是扩展了 manage.py 的命令，使得通过命令行可以生成一些代码/文件。

下载 django-groundwork 的代码：

```
$git clone https://github.com/madhusudancs/django-groundwork.git
```

```
$ls django-groundwork
```

```
AUTHORS      LICENSE      README.rst    django-groundwork
```

将**其中的** django-groundwork 文件夹复制到 project 文件夹，然后在 settings.py 中加入该 app:

[\[python\]](#) [view plaincopy](#)

```
INSTALLED_APPS = (  
    # 'django.contrib.auth',  
    # 'django.contrib.contenttypes',  
    # 'django.contrib.sessions',  
    # 'django.contrib.sites',  
    # 'django.contrib.messages',  
    # 'django.contrib.staticfiles',  
  
    # Uncomment the next line to enable the admin:  
    # 'django.contrib.admin',  
    # Uncomment the next line to enable admin documentation:  
    # 'django.contrib.admindocs',  
    'depot.depotapp',  
    'django-groundwork',  
)
```

即完成了安装。（如果遇到了什么麻烦，也可以下载[本文附带的源代码包](#)）

安装后，使用\$python manage.py help 可以看到，列出的可用命令中多了一个 groundwork。其语法是：

```
$python manage.py groundwork appname ModelName1 ModelName2
```

接下来使用这个 app 为 Product 生成 scaffold:

```
$python manage.py groundwork depotapp Product
```

，就会生成所谓的 scaffold。

此时运行开发服务器（python manage.py runserver），就可以访问下面的地址：

http://localhost:8000/depotapp/product/list/ 访问 Product 列表，并链接到 create,edit,view 等界面。

Product		
List Records		
Record	Actions	
Product object	Show	Edit
Product object	Show	Edit
Add New		
Page 1 of 1.		
django-groundwork		

Product	
Edit	
Title:	<input type="text" value="aaa"/>
Description:	<div><div>bbb</div></div>
Image url:	<input type="text" value="acc"/>
Price:	<input type="text" value="12"/>
<div>Save</div>	
django-groundwork	

可以下载本次迭代的源代码：
<http://download.csdn.net/detail/thinkinside/4035662>

4、scaffold 生成物分析

在上一节用一个插件生成了类似 rails 的 scaffold，其实无非就是 URLconf+MTV。让我们看看具体都生成了哪些东西。
首先是“入口”的定义即 URLconf，打开 urls.py:

[python] view plaincopy

```
from django.conf.urls.defaults import patterns, include, url
```

```
from depot.views import hello
```

```
urlpatterns = patterns("",
    url(r'^hello/ hello),
)
urlpatterns += patterns ("",
    (r'^depotapp/', include('depotapp.urls')),
)
```

上面的代码中增加的配置行表示：以 depotapp 开头的 url 由 depotapp/urls.py 文件进行处理。

django 的 url 配置中，除了（正则表达式，view 函数）的方式外，还支持（正则表达式，include 文件）的方式。通常把 app 自身相关的 url 写到自己的 url 配置文件中，然后在 project 中引用。

接下来看一下生成的 depotapp/urls.py 的内容：

[python] view plaincopy

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

```
from models import *
```

```
from views import *
```

```
urlpatterns = patterns("",
    (r'product/create/$', create_product),
    (r'product/list/$', list_product ),
    (r'product/edit/(?P<id>[^/]+)/$', edit_product),
    (r'product/view/(?P<id>[^/]+)/$', view_product),
)
```

将 CRU (没有 D)的 URL 映射到了视图。而视图在 depotapp/views.py 中定义：

[python] view plaincopy

```
from django import forms

from django.template import RequestContext

from django.http import HttpResponseRedirect

from django.template.loader import get_template

from django.core.paginator import Paginator

from django.core.urlresolvers import reverse
```

```
# app specific files
```

```
from models import *

from forms import *
```

```
def create_product(request):

    form = ProductForm(request.POST or None)

    if form.is_valid():

        form.save()

        form = ProductForm()

    t = get_template('depotapp/create_product.html')

    c = RequestContext(request,locals())

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

```
def list_product(request):

    list_items = Product.objects.all()

    paginator = Paginator(list_items ,10)
```

try:

page = int(request.GET.get('page', '1'))

except ValueError:

page = 1

try:

list_items = paginator.page(page)

except :

list_items = paginator.page(paginator.num_pages)

t = get_template('depotapp/list_product.html')

c = RequestContext(request,locals())

return HttpResponse(t.render(c))

def view_product(request, id):

product_instance = Product.objects.get(id = id)

t=get_template('depotapp/view_product.html')

c=RequestContext(request,locals())

return HttpResponse(t.render(c))

def edit_product(request, id):

product_instance = Product.objects.get(id=id)

form = ProductForm(request.POST **or** None, instance = product_instance)

if form.is_valid():

form.save()

t=get_template('depotapp/edit_product.html')

```
c=RequestContext(request,locals())
```

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

视图中的相关内容比较多，主要的是模板，其次还有模型类、Paginator 分页器、Form 表单等等。

基本涵盖了典型的 web 应用交互的内容。

5、引入 bootstrap，设置静态资源

之前生成了 Product 类的 scaffold，但是如同 rails 的开发者 David 所讲的那样，scaffold 几乎没什么用。所以按照《Agile Web Development with Rails 4th》中的迭代计划，下一步的修改是美化 list 页面：



但是这个界面还是太丑陋了。其实，有了 bootstrap 后，很多站点都变成了“又黑又硬”的工具条+“小清新”风格。我们即不能免俗，又懒得自己设计风格，不妨用 bootstrap 将产品清单界面重新设计成如下的风格：

产品清单

	<p>黄西的幽默</p> <p>本书是著名华裔笑星黄西的第一本书，涉及他在中国的成长、在美国的经历以及从生化博士到脱口秀演员的道路。黄西从小有礼貌但不是很听老师的话。中学老师说他考不上大学，后来因为不想让一位慈母般的老师失望而努力学习，考上吉林大学，并以第一名的成绩考入中科院研究生院。为了考GRE去美国留学，他把牛津词典背了8遍；在莱斯大学求学期间他经常在实验室里努力工作到凌晨一两点，有连续两年的时间只休过一个周末，最后拿到生化博士学位；后来工作期间，他白天去公司的做医药实验，晚上去俱乐部说相声；为了能到俱乐部演出，他在大雪天里站在俱乐部外面询问路过的人是否愿意进去听他讲相声……</p>	查看 编辑 删除
	<p>考拉小巫的英语学习日记</p> <p>这本书的出版是献给考拉小巫粉丝以及所有英语学习爱好者的一份礼物。在这本书里，考拉小巫分享了她人生中各个阶段学习英语的所有方法途径及心得体会，和她为了从根本上提高英语听、说、读、写、译能力所做的一切失败和成功的尝试及努力，以及她考过的所有英语考试的备考方案、计划安排、所用书籍及资料，书后附录中还有作者总结的英语学习资料及资源推荐。</p>	查看 编辑 删除
	<p>突然就走到了西藏</p> <p>陈坤动情自述从出身贫寒的“北漂”青年一夜成名到找回自我，继而重新上路的成长历程。关于亲情与友情、家庭与事业、名利与信仰的心灵全纪录。本书记录了陈坤带领十名大学生志愿者行走西藏过程中，对生命的点滴感悟。穿插了他成名前后生活中的小故事。笔端情感真挚，读来亲切自然。“只要你行走，就能与你生命中的真相相遇。”——陈坤这不只是一本明星写的带有自传色彩的书，更是一个在世间行走的人老实实在的心灵告白，你可以读到他的挣扎，他的茫然和转化。这本书也不仅关乎心灵，更关乎现世人生，关乎如何看待这个世界，看待这个无常世界里的命运变幻。</p>	查看 编辑 删除

下面让我们来实现这个界面。显然 web 界面会使用一些静态资源（css,js,image 等），

要在 Django 中引入静态资源）。Django 在正式部署的时候对于静态资源有特殊的处理，在开发阶段，可以有简单的方式让静态资源起作用。

首选在 project 目录下面创建一个 static 目录，并将静态资源按合理的组织方式放入其中：

```
static/  
    css/  
        bootstrap.min.css  
    js/  
    images/  
productlist.html
```

其中 productlist.html 是请界面设计师实现的产品清单静态页面；css/bootstrap.min.css 是该页面使用的样式表，来自 bootstrap，将来整个系统都将使用这一套样式风格；js 目录现在为空，以后可以将 javascript 代码放在这里；images 文件夹同理。

我们可以看到，Django 对于静态内容的管理非常符合管理。相比之下，rails 要求你将静态内容放到很怪异的结构中：

```
app/assets/  
  images/  
  javascripts/  
  stylesheets/
```

界面设计师实现的界面要想运行起来，还需要修改相关的路径，或者改变自己的目录设置习惯。这种设计让人难以理解。

回到 Django，让静态资源起作用只需要简单的配置（下面的做法只适用于开发阶段）：

修改 settings.py 的 static files 小节：

[\[python\] view plaincopy](#)

```
import os  
  
... ..  
  
# Additional locations of static files  
HERE = os.path.dirname(__file__)  
STATICFILES_DIRS = (  
    # Put strings here, like "/home/html/static" or "C:/www/django/static".  
    # Always use forward slashes, even on Windows.  
    # Don't forget to use absolute paths, not relative paths.  
    HERE+STATIC_URL,  
)
```

然后在 urls.py 中增加 static 的 url 映射：

[\[python\] view plaincopy](#)

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns  
  
... ..  
  
# for development only  
# This will only work if DEBUG is True.  
urlpatterns += staticfiles_urlpatterns()
```

启动 server，就可以通过 <http://127.0.0.1:8000/static/productlist.html> 看到设计好的界面了。

源代码：<http://download.csdn.net/detail/thinkinside/4036963>

在下一节，终于可以修改模板，美化产品清单页的样式了。

6、对比 RoR 和 Django 的模板系统

[scaffold](#) 的生成物虽然用处不大，但是给我们带来一些最佳实践。其中就有模板的继承和分区。

如果你深入使用过 rails 的模板体系，那么恭喜你：你有超强的忍耐力！而且更重要的是，你只需要 3 分钟就可以理解 Django 的模板体系。

让我们先回顾一下 rails 的模板系统：

1. 你创建了一个 xxxview，展现出一些数据。

2. 你意识到，各个 view 都有一些共同的内容。因为 rails 也强调 DRY，所以你决定将这些共同的部分抽取出来。rails 也看到了这点，所以你很高兴的看到，rails 支持 layout。

3. rails 的 layout 很简单，类似 html 的代码，`<%= yield %>` 部分会被具体视图替代，于是你很欣慰。

4. 但是等等，如何指定 layout？你又兴奋地发现：默认的 layout 是 `views/layouts/application.html.erb`，你可以在 controller、action 去指定特定的 layout，甚至这种指定支持变量。在兴奋之余，你完全忽视了这等于让 controller 去做了 view 该做的事情。

5. 你实现了一个左右结构的 layout，左侧是导航，右侧是内容。你认为这个 layout 应该可以被多个 view 使用。但是你又发现不同的 view 需要的导航是不同的。由于存在几个 view 使用一种导航、另外几个 view 使用另一个导航的情况，由于 DRY，rails 说，我有 partial。在 view 中可以使用 `<%= render "foo/bar" %>`，甚至可以使用变量：`<%= render @mypartial %>`然后在 controller/action 中指定具体的

partial。 : render :partial => 'foo/bar'。 尽管， controller 更进一步干预了 view 的细节； 尽管， 你又要记住： partial: foo/bar 意味着 views/foo/_bar.html.erb。

6. 如果 view 中的多块内容要插到 layout 的不同地方怎么办？ 除了主要的内容外， 你还可以在 view 中定义：

```
[ruby] view plaincopy
```

```
<% content_for :foo do %>
```

```
  foo foo foo
```

```
<% end %>
```

```
<% content_for :bar do %>
```

```
  bar bar bar
```

```
<% end %>
```

```
<div>下面居然就是主要内容了</div>
```

然后这些内容块会分别插入到 layout 的<%= yield :foo %> <%= yield :bar %> 和<%= yield %>的地方。

7. 还有， 还有， <%= stylesheet_link_tag "application" %>， <%= javascript_include_tag "html5" %>

到这里， 你可以说自己已经了解 rails 的模板系统了吗？

接下来我们可以放松心情了， 因为 Django 的模板很容易理解， 除了基本的变量、 标签、 过滤器等之外， 模板的关系只有两个：

1. 包含。 将模板中的相同部分提取出来共用。

可以使用硬编码的字符串{% include 'foo/bar.html' %} 或者变量名 {% include template_name %}， 变量当然是在 view 中赋值（ 注意， 不是 controller 中）

2. 继承。 模板继承是 Django 解决共用页面区域 DRY 的一个优雅的方案。 简单地说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（ override）。 基础模板中， 将内容不同的部分指定各个内容块：

```
...
```

```
{% block foo %}
```

```

<div>default content of foo</div>

{% endblock %}

...

{% block bar %}

<div>default content of bar</div>

{% endblock %}

...

```

在子模板中指定继承关系并 override 各个内容块即可。继承的写法是`{% extends "base.html" %}`，注意一定要放在模板的开头部分。

好了，你已经理解了 Django 的模板系统，下面对产品清单界面的改造就非常容易理解了。分成两个部分：base 和 productlist。

抱歉，写到这里，发现篇幅已经不短了。只好界面的实现放到下一节了。

7、-改造 ProductList 界面

有了上一节关于 Django 模板的基础，改造界面就很容易理解了。将界面设计师设计的页面中的内容根据复用程度分别放到基础模板 base.html 和专用模板 productlist.html 中。

depot/templates/base.html

[html] view plaincopy

```

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

  <meta name="description" content="a depot implement with Django"/>

  <meta name="keywords" content="django,depot" />

  <meta name="author" content="Holbrook(http://hi.csdn.net/space-2668.html)" />

  <title>{% block title %} 标题 {% endblock %}</title>

  <link rel="stylesheet" href="/static/css/bootstrap.min.css">

```

```

</head>

<body>

<div class="container">
    {% block content %}

    内容

    {% endblock %}

</div>

</body>

</html>

```

base 作为整个网站的基础布局，包含了所有页面都需要的 bootstrap.min.css。同时设置了两个内容块（title, content）。在 productlist.html 中替换这两个内容块：

depot/templates/depotapp/list_product.html

[html] view plaincopy

```

{% extends "base.html" %}

{% block title %} 产品清单 {% endblock %}

{% block content %}

<div class="container">
    <div class="page-header">
        <h2>产品清单</h2>
    </div>
    {% for item in list_items.object_list %}
    <div class="row" style="padding-top:10">
        <div class="span3 media-grid">
            <a href="#">
                
            </a>
        </div>
        <div class="span-two-thirds">
            <h4>{{item.title}}</h4>
            {{item.description}}

```

```

</div>

<div class="span2" style="align:right">

  <p><a class="btn primary" href="{% url depotapp.views.view_product item.id %}">查看
</a></a> </p>

  <p><a class="btn success" href="{% url depotapp.views.edit_product item.id %}">编辑
</a> </p>

  <p><a class="btn danger" href="#">删除</a></p>

</div>

</div>

{% endfor %}

{% if list_items.has_previous %}

  <a href="?page={{ list_items.previous_page_number }}">上一页</a>

{% endif %}

<span class="current">

  第{{ list_items.number }}页，共{{ list_items.paginator.num_pages }}页

</span>

{% if list_items.has_next %}

  <a href="?page={{ list_items.next_page_number }}">下一页</a>

{% endif %}

<p>

<a href="{% url depotapp.views.create_product %}">新增产品</a>

</p>

{% endblock %}

```

先是声明这个模板继承自 base.html，然后是两个内容块的实现。

注意其中链接的写法：href="{% url depotapp.views.view_product item.id %}"。这样定义的 href 是关联到 view 函数，而不是硬编码的 URL。在以后如果改变了 URLconf 的定义，不需要再更改模板。**这个功能不是 rails 特有的！**

关于分页的部分，无需关注，以后再说。

最后，认真填写一下表单，将真正的数据存到数据库，就可以在 <http://localhost:8000/depotapp/product/list/> 看到漂亮的界面了。

例子中使用的书籍信息和图片链接均来自[豆瓣读书](#)

8、对比 RoR 与 Django 的输入校验机制

rails 有一个“简洁、完美的验证机制，无比强大的表达式和验证框架”。在《Agile Web Development with Rails 4th》一书的 7.1 节向我们展示了如何验证 Product：

[ruby] view plaincopy

```
class Product < ActiveRecord::Base
  validates :title, :description, :image_url, :presence => true
  validates :price, :numericality => { :greater_than_or_equal_to => 0.01}
  validates :title, :uniqueness => true
  validates :image_url, :format => {
    :with => %r{\.(gif|jpg|png)$}i,
    :message => 'must be a URL for GIF, JPG or PNG image.'
  }
end
```

还是需要解释一下：

`validates :title, :description, :image_url, :presence => true`：这三个字段不能为空。

rails 默认是允许为空。而且由于 model 与 migration 是分开定义的，**你可以在 migration 中定义字段不能为空而 model 中可以为空**，或者反之。

`validates :price, :numericality => { :greater_than_or_equal_to => 0.01}`：price 字段应该是有效的数字并且不小于 0.01

`validates :image_url, :format => {...}`：image_url 必须以三种扩展名结尾，这里没有验证是否为有效的 url

更加可怕的是，这个验证语法是 rails3.0 开始支持的，而在此之前的版本要写成这样：

[ruby] view plaincopy

```
class Product < ActiveRecord::Base
  validates_presence_of :title, :description, :image_url
```



```

validates_numericality_of :price

validates_format_of :image_url, :with => %r{^http:.(gif|jpg|png)$}i,
  :message => "must be a URL for a GIF, JPG, or PNG image"

protected

def validate
  errors.add(:price, "should be positive") unless price.nil? || price > 0.0
end

end

```

再让我们看看“简洁”的 rails 验证还有哪些功能（旧版语法）：

`validates_acceptance_of`: 验证指定 checkbox 应该选中。这个怎么看都应该是 form 中的验证而与 model 无关

`validates_associated`：验证关联关系

`validates_confirmation_of`：验证 xxx 与 xxx_confirmation 的值应该相同。这个怎么看也应该是 form 中的验证而与 model 无关

`validates_length_of`：检查长度

`validates_each` 使用 block 检验一个或一个以上参数

`validates_exclusion_of` 确定被检对象不包括指定数据

`validates_inclusion_of` 确认对象包括在指定范围

`validates_uniqueness_of` 检验对象是否不重复

也许还有 more and more, and more, and more...

回到 Django。Django 的验证有 3 层机制：

1. Field 类型验证。除了能够对应到数据库字段类型的 Field 类型外，还有 EmailField，FileField，FilePathField，ImageField，IPAddressField, PhoneNumberField、URLField、XMLField 等，
2. Field 选项验证。如，null=true, blank=true, choices, editable, unique，unique_for_date, unique_for_month，unique_for_year 等等。有些 Field 还有自己独特的选项，也可以用来约束数据。
3. 表单（Form）验证。还可以在 Form 中定义验证方法。可以定义整个 Form 的验证方法 clean，或者针对某个表单项的验证方法：clean_xxx。

前面建立的 Product 模型中，已经默认加入了不能为空、要求符合数字等验证，所以还需要进行如下验证：

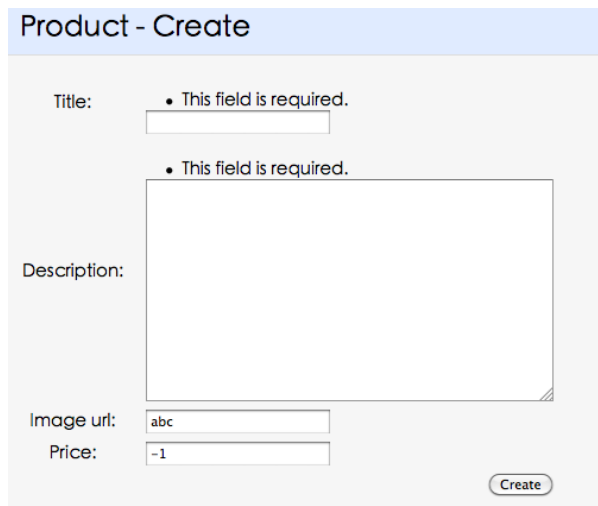
1. 验证 price>0: 需要在 Form 中验证；
2. 验证 title 唯一：在 Model 中验证；
3. 验证 image_url 的扩展名：在 Form 中验证，还可以顺便在 Model 中将其改为 URLField 类型。

9、实现 Product 的输入校验

让我们完成[上一节](#)中的任务：

1. 验证 price>0: 需要在 Form 中验证；
2. 验证 title 唯一：在 Model 中验证；
3. 验证 image_url 的扩展名：在 Form 中验证，还可以顺便在 Model 中将其改为 URLField 类型。

之前生成的 scaffold 中已经实现了属性不能为空的验证：



但是对于 url 格式，url 的后缀，title 的唯一性都没有验证。首先在 model 中增加 URL 格式和 title 唯一性的校验：

`[python]` [view plaincopy](#)

```
from django.db import models
```

```
class Product(models.Model):
```

```
    title = models.CharField(max_length=100, unique=True)
```

```
description = models.TextField()

image_url = models.URLField(max_length=200)

price = models.DecimalField(max_digits=8,decimal_places=2)
```

在 title 上增加 `unique=True`, 并将 `image_url` 的类型改为 `URLField` , 就完成了 :

Title: • Product with this Title already exists.

雷瓜的雷 西瓜的西

Description: 一些描述

Image url: • Enter a valid URL.

Is this a URL ?

Price: -1

Create

剩下的图片格式后缀、价格>0 的校验需要在 form 中实现:

depot/depotapp/forms.py

[python] view plaincopy

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

```
#coding: utf8
```

```
from django import forms
```

```
from models import *
```

```
import itertools
```

```
def anyTrue(predicate, sequence):  
    return True in itertools.imap(predicate, sequence)  
  
def endsWith(s, *endings):  
    return anyTrue(s.endswith, endings)
```

```
class ProductForm(forms.ModelForm):
```

```
    class Meta:
```

```
        model = Product
```

```
    def __init__(self, *args, **kwargs):  
        super(ProductForm, self).__init__(*args, **kwargs)
```

```
    def clean_price(self):  
        price = self.cleaned_data['price']  
        if price <= 0:  
            raise forms.ValidationError("价格必须大于零")  
        return price
```

```
    def clean_image_url(self):  
        url = self.cleaned_data['image_url']  
        if not endsWith(url, '.jpg', '.png', '.gif'):  
            raise forms.ValidationError('图片格式必须为 jpg、png 或 gif')  
        return url
```

ProductForm 继承自 ModelForm，可以根据 model 属性自动生成表单。

在生成的 ProductForm 上增加了 clean_price 和 clean_image_url 验证。结果如下：

Title:	<ul style="list-style-type: none"> Product with this Title already exists. <input type="text" value="西瓜的甜 西瓜的甜"/>
Description:	<input type="text" value="一些描述"/>
Image url:	<ul style="list-style-type: none"> 图片格式必须为jpg、png或gif <input type="text" value="http://a.b.com/index.html"/>
Price:	<ul style="list-style-type: none"> 价格必须大于零 <input type="text" value="-1"/>

那么，表单是如何展现的呢？看一下 template：

depot/depotapp/templates/depotapp/create_product.html

[html] view plaincopy

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

```
{% block title %} 创建产品 {% endblock %}
```

```
{% block content %}
```

```
<table>
```

```
<form action="" method="POST"> {% csrf_token %}
```

```
{{form}}
```

```
<tr>
```

```
<td colspan="2" align="right"><input type="submit" value="Create"/></td>
```

```
</tr>
```

```
</form>
```

```
</table>
```

```
{% endblock %}
```

直接输出 form 对象 (`{{fom}}`) 就会将 Form 格式化表单 (默认使用 table , 也可以通过 `as_p,as_ul` 方法指定为 `<p>` 或 ``) , 并且包含了错误提示信息。

`{% csrf_token %}` 的作用是增加 token 表单项 , 避免重复提交。

10、单元测试

尽早进行单元测试 (UnitTest) 是比较好的做法 , 极端的情况甚至强调“测试先行”。现在我们已经有了第一个 model 类和 Form 类 , 是时候开始写测试代码了。

Django 支持 python 的单元测试 (unit test) 和文本测试 (doc test) , 我们这里主要讨论单元测试的方式。这里不对单元测试的理论做过多的阐述 , 假设你已经熟悉了下列概念 : test suite, test case, test/test action, test data , assert 等等。

在单元测试方面 , Django 继承 python 的 `unittest.TestCase` 实现了自己的 `django.test.TestCase` , 编写测试用例通常从这里开始。测试代码通常位于 app 的 `tests.py` 文件中 (也可以在 `models.py` 中编写 , 但是我不建议这样做) 。在 Django 生成的 depotapp 中 , 已经包含了这个文件 , 并且其中包含了一个测试用例的样例 :

depot/depotapp/tests.py

[\[python\] view plaincopy](#)

```
from django.test import TestCase

class SimpleTest(TestCase):

    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 2)
```

你可以有几种方式运行单元测试 :

`python manage.py test` : 执行所有的测试用例

`python manage.py test app_name`, 执行该 app 的所有测试用例

`python manage.py test app_name.case_name`: 执行指定的测试用例

用第三中方式执行上面提供的样例，结果如下：

```
$ python manage.py test depotapp.SimpleTest
Creating test database for alias 'default'...
```

```
.
```

```
Ran 1 test in 0.012s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

你可能会注意到，输出信息中包括了创建和删除数据库的操作。为了避免测试数据造成的影响，测试过程会使用一个单独的数据库，关于如何指定测试数据库的细节，请查阅 Django 文档。在我们的例子中，由于使用 sqlite 数据库，Django 将默认采用内存数据库来进行测试。

下面就让我们来编写测试用例。在《Agile Web Development with Rails 4th》中，7.2 节，最终实现的 ProductTest 代码如下：

[[ruby](#)] [view plaincopy](#)

```
class ProductTest < ActiveSupport::TestCase

  test "product attributes must not be empty" do

    product = Product.new

    assert product.invalid?

    assert product.errors[:title].any?

    assert product.errors[:description].any?

    assert product.errors[:price].any?

    assert product.errors[:image_url].any?

  end

  test "product price must be positive" do

    product = Product.new(:title => "My Book Title",
                          :description => "yyy",
                          :image_url => "zzz.jpg")

    product.price = -1

    assert product.invalid?

    assert_equal "must be greater than or equal to 0.01",
      product.errors[:price].join('; ')
```

```
product.price = 0
assert product.invalid?
assert_equal "must be greater than or equal to 0.01",
  product.errors[:price].join('; ')
```

```
product.price = 1
assert product.valid?
```

end

```
def new_product(image_url)
  Product.new(:title    => "My Book Title",
             :description => "yyy",
             :price     => 1,
             :image_url => image_url)
```

end

test "image url" **do**

```
ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
  http://a.b.c/x/y/z/fred.gif }
bad = %w{ fred.doc fred.gif/more fred.gif.more }
```

```
ok.each do |name|
  assert new_product(name).valid?, "#{name} shouldn't be invalid"
```

end

```
bad.each do |name|
  assert new_product(name).invalid?, "#{name} shouldn't be valid"
```

end

end

test "product is not valid without a unique title" **do**

```
product = Product.new(:title    => products(:ruby).title,
```



```

        :description => "yyy",
        :price       => 1,
        :image_url   => "fred.gif")

    assert !product.save

    assert_equal "has already been taken", product.errors[:title].join('; ')
end

test "product is not valid without a unique title - i18n" do
  product = Product.new(:title => products(:ruby).title,
    :description => "yyy",
    :price       => 1,
    :image_url   => "fred.gif")

  assert !product.save

  assert_equal I18n.translate('activerecord.errors.messages.taken'),
    product.errors[:title].join('; ')
end

end

```

对 Product 测试的内容包括：

1. title , description , price , image_url 不能为空；
2. price 必须大于零；
3. image_url 必须以 jpg , png , jpg 结尾，并且对大小写不敏感；
4. title 必须唯一；

让我们在 Django 中进行这些测试。由于 [ProductForm](#) 包含了模型校验和表单校验规则，使用 ProductForm 可以很容易的实现上述测试：

depot/depotapp/tests.py

[\[python\]](#) [view plaincopy](#)

```

#!/usr/bin/python
#coding: utf8

```

```
"""
```

This file demonstrates writing tests using the unittest module. These will pass when you run "manage.py test".

Replace this with more appropriate tests for your application.

```
"""
```

```
from django.test import TestCase
```

```
from forms import ProductForm
```

```
class SimpleTest(TestCase):
```

```
    def test_basic_addition(self):
```

```
        """
```

Tests that 1 + 1 always equals 2.

```
        """
```

```
        self.assertEqual(1 + 1, 2)
```

```
class ProductTest(TestCase):
```

```
    def setUp(self):
```

```
        self.product = {
```

```
            'title': 'My Book Title',
```

```
            'description': 'yyy',
```

```
            'image_url': 'http://google.com/logo.png',
```

```
            'price': 1
```

```
        }
```

```
        f = ProductForm(self.product)
```

```
        f.save()
```

```
        self.product['title'] = 'My Another Book Title'
```

```
##### title , description , price , image_url 不能为空
```

```
    def test_attrs_cannot_empty(self):
```

```
        f = ProductForm({})
```

```
self.assertFalse(f.is_valid())

self.assertTrue(f['title'].errors)

self.assertTrue(f['description'].errors)

self.assertTrue(f['price'].errors)

self.assertTrue(f['image_url'].errors)
```

price 必须大于零

```
def test_price_positive(self):

    f = ProductForm(self.product)

    self.assertTrue(f.is_valid())

    self.product['price'] = 0

    f = ProductForm(self.product)

    self.assertFalse(f.is_valid())

    self.product['price'] = -1

    f = ProductForm(self.product)

    self.assertFalse(f.is_valid())

    self.product['price'] = 1
```

image_url 必须以 jpg , png , jpg 结尾 , 并且对大小写不敏感 ;

```
def test_imgae_url_endwiths(self):

    url_base = 'http://google.com/'

    oks = ('fred.gif', 'fred.jpg', 'fred.png', 'FRED.JPG', 'FRED.Jpg')

    bads = ('fred.doc', 'fred.gif/more', 'fred.gif.more')

    for endwith in oks:

        self.product['image_url'] = url_base+endwith

        f = ProductForm(self.product)

        self.assertTrue(f.is_valid(),msg='error when image_url endwith '+endwith)

    for endwith in bads:

        self.product['image_url'] = url_base+endwith
```

```

        f = ProductForm(self.product)

        self.assertFalse(f.is_valid(),msg='error when image_url endwith '+endwith)

self.product['image_url'] = 'http://google.com/logo.png'

### titile 必须唯一

def test_title_unique(self):

    self.product['title'] = 'My Book Title'

    f = ProductForm(self.product)

    self.assertFalse(f.is_valid())

    self.product['title'] = 'My Another Book Title'

```

然后运行 `python manage.py test depotapp.ProductTest`。如同预想的那样，测试没有通过：

```

    Creating test database for alias 'default'...
.F..
=====
=====
FAIL: test_imgae_url_endwiths (depot.depotapp.tests.ProductTest)
-----
Traceback (most recent call last):
  File "/Users/holbrook/Documents/Dropbox/depot/./depot/depotapp/tests.py", line
65, in test_imgae_url_endwiths
    self.assertTrue(f.is_valid(),msg='error when image_url endwith '+endwith)
AssertionError: False is not True : error when image_url endwith FRED.JPG
-----

Ran 4 tests in 0.055s

FAILED (failures=1)
Destroying test database for alias 'default'...

```

因为我们之前并没有考虑到 `image_url` 的图片扩展名可能会大写。修改 `ProductForm` 的相关部分如下：

`[python]` [view plaincopy](#)

```

def clean_image_url(self):

    url = self.cleaned_data['image_url']

    if not endsWith(url.lower(), '.jpg', '.png', '.gif'):

```

```
raise forms.ValidationError('图片格式必须为 jpg、png 或 gif')

return url
```

然后再运行测试：

```
$ python manage.py test depotapp.ProductTest
Creating test database for alias 'default'...
....
-----
Ran 4 tests in 0.060s

OK
Destroying test database for alias 'default'...
```

测试通过，并且通过单元测试，我们发现并解决了一个 bug。

11、修改 Model 类

我们已经实现了卖方的产品维护界面，根据最初的需求，还要为买方实现一个目录页：买方通过这个界面浏览产品并可以加入购物车。通过进一步需求调研，了解到产品有一个“上架时间”，在这个时间之后的产品才能被买方看到。并且买方应该先看到最新的产品。

我们注意到，这个“新需求”需要对 Product 进行调整，增加一个日期属性 `date_available` 来保存“上架时间”。如同开发新功能一样，在修改的时候也应该从 model 开始。

为 Model 类增加一个属性很容易：

[python] view plaincopy

```
class Product(models.Model):

    title = models.CharField(max_length=100,unique=True)

    description = models.TextField()

    image_url = models.URLField(max_length=200)

    price = models.DecimalField(max_digits=8,decimal_places=2)

    date_available = models.DateField()
```

问题在于，model 类的改变需要对数据库表也进行同样的修改。你可能想到前面介绍过的 `python manage.py syncdb`，但是很遗憾这个命令现在不起作用了。因为 syncdb 仅仅创建数据库里还没有的表，它并不 对你数据模型的修改进行同步，也不处理数据模型的删除。如果你新增或修改数据模型里的字段，或是删除了一个数据模型，你需要手动在数据库里进行相应的修改。当然，在开发环境你可以 drop 相应的表，然后运行 syncdb 重新创建。但是这样做对于发布环境没有任何帮助，所以我们最好这样做：

1. 使用 sqlall 查看模型新的 CREATE TABLE 语句。查看新创建的字段：

```
$ python manage.py sqlall depotapp
BEGIN;
CREATE TABLE "depotapp_product" (
  "id" integer NOT NULL PRIMARY KEY,
  "title" varchar(100) NOT NULL UNIQUE,
  "description" text NOT NULL,
  "image_url" varchar(200) NOT NULL,
  "price" decimal NOT NULL,
  "date_available" date NOT NULL
)
;
COMMIT;
```

2. 使用数据库命令行工具，或客户端工具，或者 django 提供的 dbshell 工具增加字段：

```
$ python manage.py dbshell
sqlite> begin;
sqlite> alter table depotapp_product add column date_available date not null default 0;
sqlite> commit;
```

3. 验证数据库：

```
$ python manage.py shell
>>> from depot.depotapp.models import Product
>>> Product.objects.all()
```

如果没有异常发生，则可以在发布环境进行上述修改。

以上是在模型中增加一个属性的步骤。其他对模型的修改可能还有：

删除字段——drop column

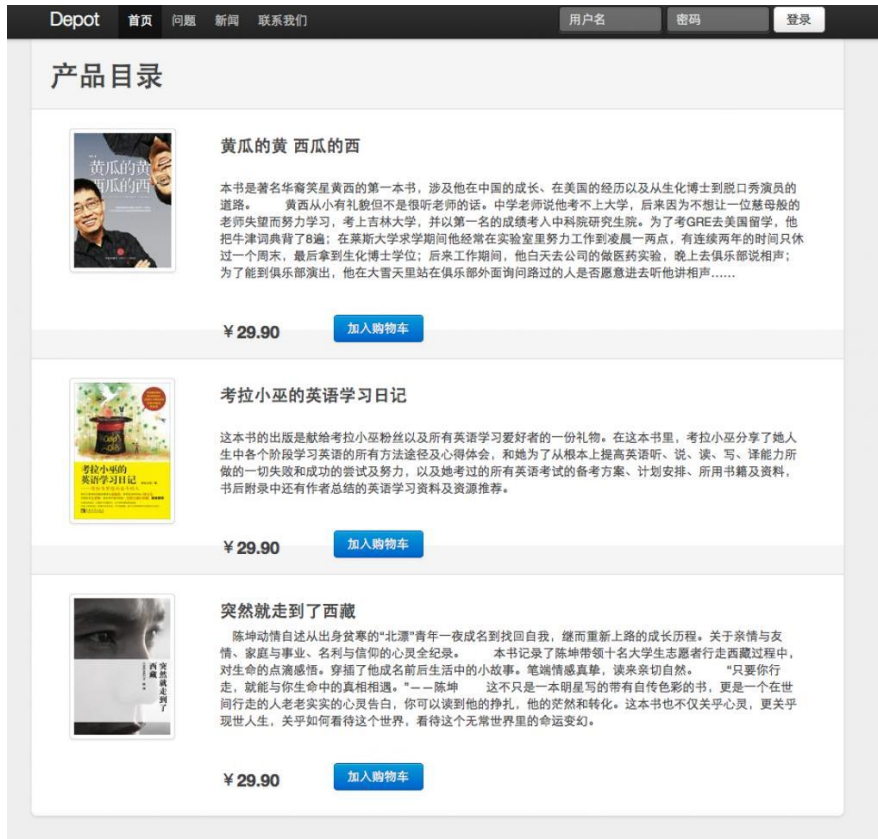
删除模型——drop table

删除多对多关联——drop table (自动生成的关联表)

也需要参考上述的步骤进行。

12、增加目录页，设定统一布局

针对上一节的新需求，界面设计师还为我们设计了一个新的界面，不仅仅是目录页，还包含了站点的整体风格，如下图：



感谢界面设计师为我们提供的“又黑又硬”的工具条，这个看起来真的很酷。下面，让我们来享用她的工作成果吧。

我们前面的 `scaffold` 已经生成了有继承关系模板，显然对于一些公用的内容应该放到 `base.html` 之中。但是我们先把手头的事情放到一边，先来实现目录页。

首选为目录页确定一个 url，不妨叫做 `depotapp/store`，在 `depotapp` 的 `urls.py` 中增加一条 `pattern`：

```
[python] view plaincopy
```

```
(r'store/$', store_view),
```

而 `store_view` 是对应的视图函数，在 `depotapp` 的 `views.py` 中定义：

```
[python] view plaincopy
```

```
def store_view(request):
    products = Product.objects.filter(date_available__gt=datetime.datetime.now().date()) \
        .order_by("-date_available")
    t = get_template('depotapp/store.html')
    c = RequestContext(request, locals())
    return HttpResponse(t.render(c))
```

store_view 使用 depotapp/store.html 作为模板：

depot/templates/depotapp/store.html

[html] view plaincopy

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

{% block title %} 产品目录 {% endblock %}

{% block pagename %} 产品目录 {% endblock %}

{% block content %}
{% for item in products %}
<div class="row" style="padding-top:10">
    <div class="span3 media-grid">
        <a href="#">
            
        </a>
    </div>
    <div class="span-two-thirds">
        <h3>{{item.title}}</h3>
        <br/>
        {{item.description}}
        <br/>
        <br/>
        <br/>
    </div>
    <div class="row">
        <div class="span2"><h3>¥ {{item.price|floatformat:"2"}}</h3></div>
        <div class="span"><a class="btn primary" href="#">加入购物车</a></div>
    </div>
{% endfor %}
{% endblock %}
```



```

        </div>
    </div>

</div>
<div class="page-header">
</div>
{% endfor %}
{% endblock %}

```

该模板继承了 base.html，在 base 模板中实现了整个站点的基础布局：

depot/templates/base.html

[html] view plaincopy

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="description" content="a depot implement with Django"/>
    <meta name="keywords" content="django,depot" />
    <meta name="author" content="Holbrook(http://hi.csdn.net/space-2668.html)" />
    <title>{% block title %} 标题 {% endblock %}</title>
    <!-- Le HTML5 shim, for IE6-8 support of HTML elements -->
    <!--[if lt IE 9]>
        <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->

    <!-- Le styles -->
    <link rel="stylesheet" href="/static/css/bootstrap.min.css">
    <link rel="stylesheet" href="/static/css/layout.css">
</head>
<body>
    <div class="topbar">
        <div class="fill">
            <div class="container">
                <a class="brand" href="#">Depot</a>
            
```

```
<ul class="nav">

  <li class="active"><a href="#">首页</a></li>

  <li><a href="#about">问题</a></li>

  <li><a href="#contact">新闻</a></li>

  <li><a href="#contact">联系我们</a></li>

</ul>

<form action="" class="pull-right">

  <input class="input-small" type="text" placeholder="用户名">

  <input class="input-small" type="password" placeholder="密码">

  <button class="btn" type="submit">登录</button>

</form>

</div>

</div>

</div>

<div class="container">

  <div class="content">

    <div class="page-header">

      <h1>{% block pagename %} 页面名称 {% endblock %}</h1>

    </div>

    {% block content %}

    内容

    {% endblock %}

  </div><!-- /content -->

</div><!-- /container -->

</body>

</html>
```

这样，我们用很少的代码就实现了新增一个目录页，并且为整个站点设置了统一的布局。

13、在 session 中保存购物车

现在，我们有了一个[产品目录界面](#)，用户如果看到满意的产品，就可以将其放入购物车。下面就让我们来实现购物车的功能。

首先要做一下简单的分析和设计。购物车应该显示一系列产品的清单，其中列出了买方选中的产品。但是这个清单没有必要马上保存到数据库，因为直到付款之前，用户随时都有可能改变主意。我们只需要在用户的 session 中记录这些产品就可以了。

购物车中的条目

购物车中的条目与[产品 \(Product\)](#) 很类似，但是我们没有必要将这些信息再重复记录一次，而只需要让条目关联到产品即可。此外在条目中还会记录一些产品中没有的信息，比如数量。最后，我们想要在条目中记录一下单价——虽然产品中包含了价格信息，但是有时候可能会有一些折扣，所以需要记录下来用户购买时的价格。基于这些分析，我们设计了条目这一模型类：

`[python] view plaincopy`

```
class LineItem(models.Model):  
    product = models.ForeignKey(Product)  
    unit_price = models.DecimalField(max_digits=8, decimal_places=2)  
    quantity = models.IntegerField()
```

购物车

购物车是这些条目的容器。我们希望实现一个“聪明的”购物车，它可以有自己的一些行为：比如，如果放入已经有的产品，就更改该产品的数量而不是再增加一个条目；能够查询当前购物车中的产品数量，等等。所以购物车也应该是一个模型类。但是与 LineItem 不同，购物车并不需要记录到数据库中，就好像超市并不关注顾客使用了哪辆购物车而只关注他买了什么商品一样。所以购物车不应该继承自 models.Model，而仅仅应该是一个普通类：

`[python] view plaincopy`

```
class Cart(object):  
    def __init__(self, *args, **kwargs):  
        self.items = []
```

```

self.total_price = 0

def add_product(self, product):
    self.total_price += product.price

    for item in self.items:
        if item.product.id == product.id:
            item.quantity += 1

    return

self.items.append(LineItem(product=product, unit_price=product.price, quantity=1))

```

在我们设计模型的同时，界面设计师也为我们设计好了购物车的界面：



接下来就可以实现 url 映射，view 函数和模板。首先我们希望购物车的 url 为“http://localhost:8000/depotapp/cart/view/”，这需要在 depotapp/urls.py 的 urlpatterns 中增加一行：

```

[python] view plaincopy

(r'cart/view/', view_cart),

```

然后在 depotapp/views.py 中定义视图函数。注意购物车是保存在 session 中的，需要通过 request.session.get 获取：

```

[python] view plaincopy

def view_cart(request):
    cart = request.session.get("cart", None)

    t = get_template('depotapp/view_cart.html')

    if not cart:
        cart = Cart()

    request.session["cart"] = cart

```

```
c = RequestContext(request,locals())
```

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

最后实现模板界面：

[\[html\] view plaincopy](#)

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

```
{% block title %} 我的购物车{% endblock %}
```

```
{% block pagename %} 我的购物车 {% endblock %}
```

```
{% block content %}
```

```
<div class="row">
```

```
<div class="span10">
```

```
<table class="condensed-table">
```

```
<thead>
```

```
<tr>
```

```
<th class="header">数量</th>
```

```
<th class="yellow header">名称</th>
```

```
<th class="blue header">单价</th>
```

```
<th class="green header">小计</th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
{% for item in cart.items %}
```

```
<tr>
```

```
<th>{{item.quantity}}</th>
```

```
<td>{{item.product.title}}</td>
```

```
<td>{{item.unit_price}}</td>
```

```

        <td>{% widthratio item.quantity 1 item.unit_price %}</td>

    </tr>

{% endfor %}

<tr>

    <th></th>

    <td></td>

    <th>总计 :</th>

    <td>{{cart.total_price}}</td>

</tr>

</tbody>

</table>

</div>

<div class="span4">

    <p><a class="btn primary span2" href="#">继续购物</a></p>

    <p><a class="btn danger span2" href="#">清空购物车</a></p>

    <p><a class="btn success span2" href="#">结算</a></p>

</div>

</div>

{% endblock %}

```

这里面有一个技巧。因为 Django 模板的设计理念是“业务逻辑应该和表现逻辑相对分开”，所以在 Django 模板中不建议执行过多的代码。在计算条目小计的时候，使用的是 Django 模板的 `widthratio` 标签。该标签的原意是按比例计算宽度：根据当前值（`this_value`）和最大值（`max_value`）之间的比例，以及最大宽度（`max_width`）计算出当前的宽度（`this_width`），即 `{% widthratio this_value max_value max_width %} = max_width * this_value / max_value`。但是如果我们设定 `max_value=1`，就可以通过 `width ratio` 在 Django 模板中进行乘法计算了。同理还可以进行除法计算。

而总计价格的计算是通过模型 (Cart) 类实现的。现在是广告时间，很高兴地发现有人转载了本“实战”系列，但是没有给出原文地址。关于 Django 系列的文章会不断更新和完善，敬请到原始地址 <http://blog.csdn.net/thinkinside> 围观！

为 Django 增加 session 支持

好了，我们已经做了大量的工作，现在让我们欣赏一下自己的作品。但是启动 server 并访问 <http://localhost:8000/depotapp/cart/view/> 时，却提示“找不到 django_session 表”。这是因为 Django 对 session 的支持是通过内置的 django.contrib.sessions 应用实现的，该应用会将 session 数据保存在数据库中。但是创建 project 是 Django 并没有默认安装该 app——Django 不会瞒着你做任何事情。所以如果我们“显式地”决定要使用 session，需要更改 project 的设置，在 depot/settings.py 的 INSTALLED_APPS 中去掉 session 的注释：

`[python]` view plaincopy

```
INSTALLED_APPS = (  
    # 'django.contrib.auth',  
    # 'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    # 'django.contrib.sites',  
    # 'django.contrib.messages',  
    # 'django.contrib.staticfiles',  
  
    # Uncomment the next line to enable the admin:  
    # 'django.contrib.admin',  
    # Uncomment the next line to enable admin documentation:  
    # 'django.contrib.admindocs',  
    'depot.depotapp',  
    'django-groundwork',  
)
```

然后还要同步一下数据库：

```
$ python manage.py syncdb  
Creating tables ...  
Creating table django_session  
Installing custom SQL ...  
Installing indexes ...  
No fixtures found.
```

这是再启动服务器，就可以看到我们实现的购物车了。但是这个购物车能看不能用，无法将商品加入购物车，也无法从中去掉商品。下一节，让我们继续完善购物车的功能。

14、让页面联动起来

上一节我们实现了一个“能看不能用”的购物车，现在我们来使用这个购物车。

首先是产品目录界面中的“加入购物车”链接，我们希望点击这个按钮后，在购物车中添加该产品（添加的规则是如果购物车中已经有该产品就增加数量，如果没有就增加条目），然后显示购物车。首先我们设计一个“RESTful”的加入购物车链接，其形式是：

`http://localhost:8080/depotapp/cart/add/xxx`

其中 xxx 为产品的 ID。

这需要在 `depotapp/urls.py` 中增加一条配置：

```
[python] view plain copy
(r'cart/view/(?P<id>[^\]+)/$', add_to_cart),
```

这条 url 配置中使用了 python 正则表达式中的“命名组”。其语法是 `(?P<name>pattern)`，这里 name 是组的名字，而 pattern 是匹配的某个模式。命名组的作用是将 pattern 匹配到的字符串按照 name 指定的参数传递给 view 函数。在上面的配置中指定参数名称为 id。如果我们的 view 函数定义为 `def add_to_cart(request,id)`。上述的 url 配置会自动调用 `add_to_cart(request, id=xxx)`，其中 xxx 为产品的 ID。（如果不使用命名组，则会调用 `add_to_cart(request,xxx)`，如果有多个参数需要通过 url 匹配，就会带来麻烦）

接下来是定义 view 函数，在 `depotapp/views.py` 中：

```
[python] view plain copy
def add_to_cart(request,id):
    product = Product.objects.get(id = id)
    cart = request.session.get("cart",None)
    if not cart:
        cart = Cart()
        request.session["cart"] = cart
    cart.add_product(product)
    request.session['cart'] = cart
```



```
return view_cart(request)
```

根据 url 中的 id 参数获取产品，加入购物车，然后调用 view_cart 视图函数显示购物车。这里面判断 session 中是否有 cart 对象的写法与上一节的 view_cart 视图函数中的写法相同，不符合 DRY 的原则。为了简单这里暂时不去管它，在后续的内容中再来重构。

要注意，从 session 中获取对象后，对该对象属性的更改不能自动同步到 session 中，而是需要重新写入 session。

最后就是增加链接，使得从产品列表页可以直接加入购物车。使用前面介绍过的 Django 模板中的 url 表达式可以自动生成到视图函数的链接，即使将来 URLconf 有所改变，也不需要再修改模板。同样的道理，在购物车模板中，将“继续购物”的链接改为{% url depotapp.views.store_view %}，就可以自动链接到产品目录视图对应的 url。

最后再实现一个“清空购物车”的功能：

url:

```
[python] view plaincopy  
(r'cart/clean/', clean_cart),
```

view 函数：

```
[python] view plaincopy  
  
def clean_cart(request):  
    request.session['cart'] = Cart()  
    return view_cart(request)
```

view_cart.html 模板中的链接：

```
{% url depotapp.views.clean_cart %}
```

请自己将这些内容加入相关文件，即可实现“清空购物车”的功能。

15、Django 实现 RESTful web service

曾几何时，Ajax 已经统治了 Web 开发中的客户端，而 REST 成为 web 世界中最流行的架构风格（architecture style）。所以我们的选择变得很简单：前端 ajax 访问后端的 RESTful web service 对资源进行操作。

Django 中有一些可选的 REST framework，比如 [django-piston](#)，[django-tasypie](#)。但是我和 google（呵呵，不好意思）推荐这个：[Django REST framework](#)。因为这个框架的几个特点：

1. 名字好！直入主题
2. 因为名字好，所以 google 搜索（Django REST/ Django RESTful）排名第一
3. 当然，前面两个理由都是开玩笑的。最重要的理由是该框架对于 resource，serializer，renderer/parser，view 和 response 的定义很清晰，又很符合 Django 的 MTV 模式（比如，它的 view 就是包装了 Django 的 View 实现的）。在 Django 中使用该框架可以说是顺乎自然。
4. 对于认证和授权有很好的支持。
5. 内置了一系列的 Mixin，可以随意组装。

下面用 Django REST framework 来实现购物车（Cart）的 RESTful web service。

第一步：安装

官方文档说可以用 pip 或 easy_install 安装，但是经过实测使用 easy_install 安装的不是最新版，会损失一些特性。所以建议用源代码的方式安装：

从 <http://pypi.python.org/pypi/django-rest-framework/0.3.2> 下载 v0.3.2，解压后 \$sudo python setup.py install

第二步：配置

在 depot/settings.py 的 INSTALLED_APPS 中加入：

```
'django-rest-framework',
```

第三步：使用

Django REST framework 有很多种“用法”，最常见的用法是：

1. 定义资源。资源将 python 对象（比如 model 对象）进行隔离、组装，生成需要序列化（serialize）的数据。除了基本的 Resource 类型外，框架还提供了 FormResource 和 ModelResource，以便于对 Form 或 Model 的处理。Resource 有助于 View 中的处理，当然你也可以不使用 Resource，而在 View 中去指定要序列化的数据。
2. 创建视图。视图是对 django View 的封装，并定义序列化、反序列化等方法，同时通过 Mixin 的支持来实现 get, post, put, delete 等操作。框架内置了 ModelView，与

ModelResource 配合使用非常简单方便。

3. 定义 url，将正则表达式匹配的 View 类的 as_view 方法，该方法会返回 django 的 view 函数。

在我们的例子中，要处理的不是购物车本身，而是购物车中的 line_item,属于 model 类，所以使用 ModelResource 和 ModelView 是最方便的。具体实现：

创建 depotapp/resources.py

`[python] view plaincopy`

```
from django.core.urlresolvers import reverse

from djangorestframework.views import View

from djangorestframework.resources import ModelResource

from models import *
```

```
class LineItemResource(ModelResource):

    model = LineItem

    fields = ('product', 'unit_price', 'quantity')

    def product(self, instance):

        return instance.product.title
```

其中重新定义关联的对象。比如 LineItem 关联到了 Product，但我们在 resource 中将 product 属性重新定义为 product.title

然后使用 ModelView 定义 url：在 depot/depotapp/urls.py 的 urlpatterns 中增加 url 映射，当然首选要引入相关的模块：

depot/depotapp/urls.py

`[python] view plaincopy`

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

from models import *

from views import *

from djangorestframework.views import ListOrCreateModelView, InstanceModelView

from resources import *

urlpatterns = patterns("",

    (r'product/create/$', create_product),

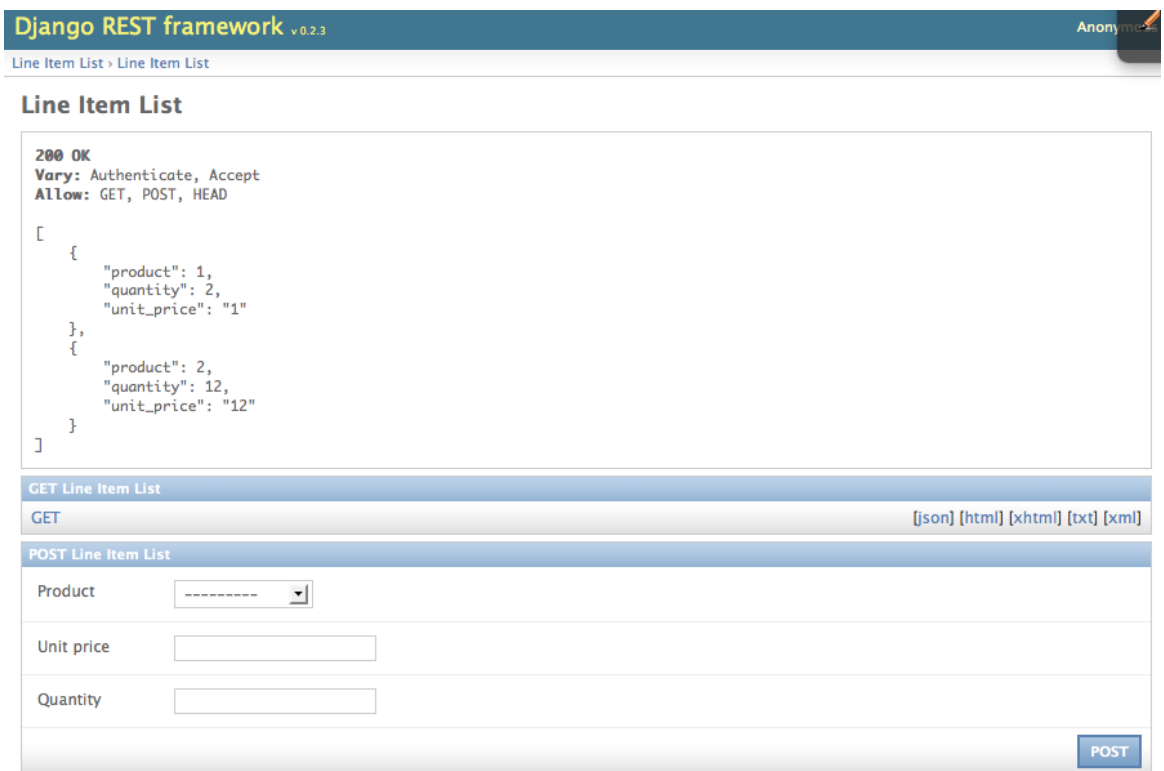
    (r'product/list/$', list_product ),
```

```

(r'product/edit/(?P<id>[^\]+)/$', edit_product),
(r'product/view/(?P<id>[^\]+)/$', view_product),
(r'store/$', store_view),
(r'cart/view/', view_cart),
(r'cart/clean/', clean_cart),
(r'cart/add/(?P<id>[^\]+)/$', add_to_cart),
(r'API/cart/items', ListOrCreateModelView.as_view(resource=LineItemResource)),
)

```

此时访问 <http://localhost:8000/depotapp/API/cart/items/> 就可以看到生成的 RESTful API 了：



如图所示，可以渲染（render）成 json,html, xhtml,txt,xml 等格式，当然你也可以增加自己的渲染，比如 YAML

对于一般的情况来说，这样做就已经足够了。但是我们这里由于 LineItem 不是从数据库中获取的，而是从 session 中的 cart 对象中获取，所以还需要进行一些改造。框架提供的 ListOrCreateModelView 继承了 ModelView，同时混合了 ListModelMixin 和 CreateModelMixin。而 ListModelMixin 定义了 get 方法，该方法使用 model.objects.all() 即从数据库中获取数据，所以我们应该修改一下 View 的行为，让其从 session 中获取数据，不妨在 depotapp/views.py 中自定义一个 View 类：

[python] view plaincopy

```
from djangorestframework.views import View
```

```
class RESTforCart(View):
```

```
    def get(self, request, *args, **kwargs):
```

```
        return request.session['cart'].items
```

然后将 url 改为 : (r'API/cart/items',
RESTforCart.as_view(resource=LineItemResource)),

这时再访问 <http://localhost:8000/depotapp/API/cart/items/> , 就可以显示购物车中的 item 了。默认的是 html 渲染 , 你可以通过
<http://localhost:8000/depotapp/API/cart/items/?format=json> 访问 json 渲染:

[javascript] view plaincopy

```
[{"product": "\u7a81\u7136\u5c31\u8d70\u5230\u4e86\u897f\u85cf", "unit_price": "12", "quantity": 2}, {"product": "\u9ec4\u74dc\u7684\u9ec4\u897f\u74dc\u7684\u897f", "unit_price": "12", "quantity": 38}]
```

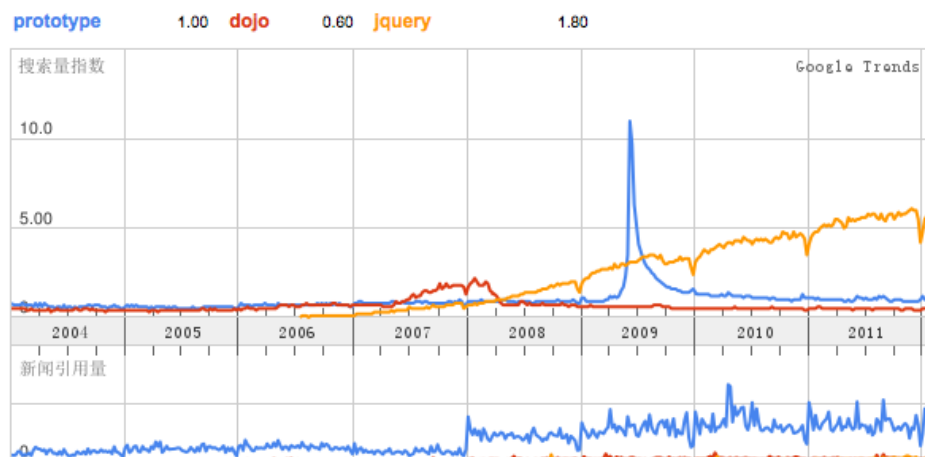
用 Django REST framework 实现 RESTful web service , 可以说即简单 , 又灵活。

16、Django+jquery

现在我们有了一个使用 json 格式的 RESTful API , 可以实现这样的功能了 : 为了避免在产品列表和购物车之间来回切换 , 需要在产品列表界面显示购物车 , 并且通过 ajax 的方式不刷新界面就更新购物车的显示内容。

ajax 框架的选择

关于 ajax 框架的选择 , 看图不说话 :



我不管你选的是什么，反正我是选了 jquery。

在 Django 中使用 jquery

这个实在是简单得不能在简单了，在 `depot/static` 下面创建 `js` 文件夹，放入 [jquery 库](#)，如 `jquery-1.7.1.min.js`。然后在模板界面中引入即可。我们假定所有的界面都使用 jquery，而且希望我们能够编写出 [Unobtrusive JavaScript](#)，所以在 `base.html` 中，在 `</body>` 之前加入如下几行内容：

[html] view plaincopy

```
<script src="/static/js/jquery-1.7.1.min.js"></script>

{% block js %}

    <!--这里插入具体页面引用的 js 库，或者 js 代码-->

{% endblock %}

<script>

$(function () {

    //这里编写 base 界面的 on_ready 代码

    {% block on_ready %}

        //这里插入具体界面的 on_ready 代码

    {% endblock %}

});

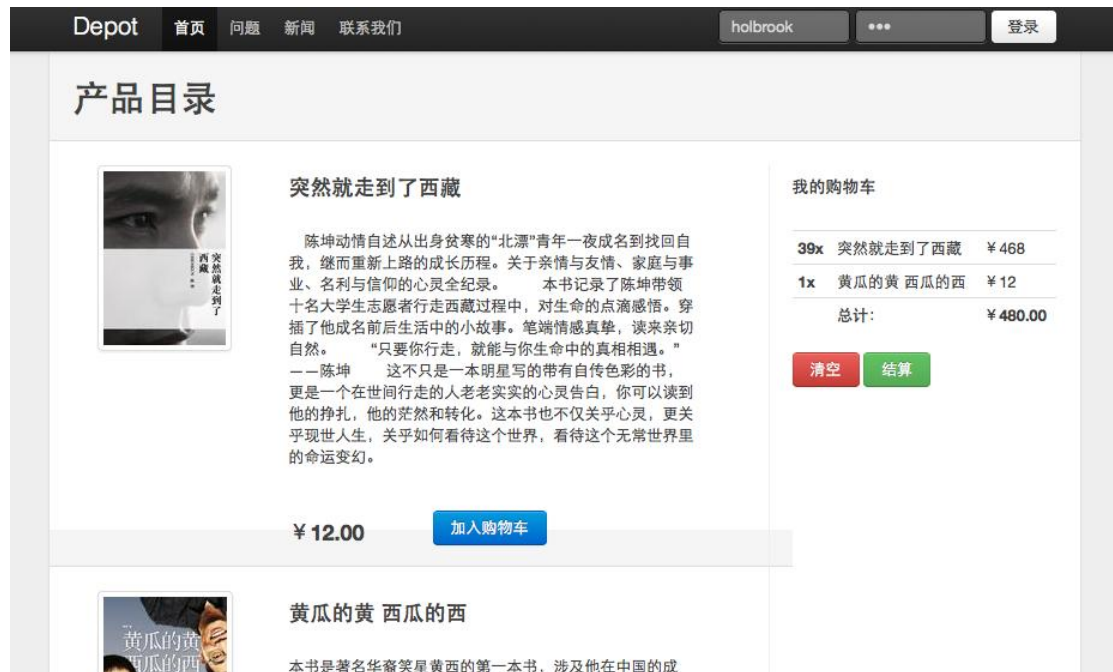
</script>

</body>
```

注：jquery 的 `$(function(){...})` 是 `$(document).ready(function(){...})` 的简写。

嵌入购物车界面

为了实现如下图所示的嵌入购物车的产品目录界面，我们需要做两件事情：



1. 修改模板：

depot/templates/depotapp/store.html:

[html] view plaincopy

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

{% block title %} 产品目录 {% endblock %}

{% block pagename %} 产品目录 {% endblock %}

{% block content %}

<div class="row">

  <div class="span10">

{% for item in products %}

<div class="row" style="padding-top:10">

  <div class="span3 media-grid">

    <a href="#">
```

```

        
    </a>
</div>
<div class="span6">
    <h3>{{item.title}}</h3>
    <br/>
    {{item.description}}
    <br/>
    <br/>
    <br/>
    <div class="row">
        <div class="span2"><h3>¥ {{item.price|floatformat:"2"}}</h3></div>
        <div class="span"><a class="btn primary" href="{% url depotapp.views.add_to_cart item.id %}">加入购物车</a></div>
    </div>
</div>

</div>
<div class="page-header">
</div>
{% endfor %}

</div><!--span10-->
<div class="span4">
    <h5>我的购物车</h5><br/>
    <table class="condensed-table">
        <tbody>
            {% for item in cart.items %}
                <tr>
                    <th>{{item.quantity}}x</th>
                    <td>{{item.product.title}}</td>
                    <td>¥ {% widthratio item.quantity 1 item.unit_price %}</td>
                </tr>
            {% endfor %}

```



```

        <tr>
            <td></td>

            <th>总计 : </th>

            <th>¥ {{cart.total_price|floatformat:"2"}}</th>

        </tr>
    </tbody>
</table>

<a class="btn danger" href="{% url depotapp.views.clean_cart %}">清空</a>

<a class="btn success" href="#">结算</a>

</div><!--span4-->

{% endblock %}

```

2. 在 depotapp/views.py 中的 `view_cart` `store_view` 视图函数中增加一行：

```
cart = request.session.get("cart",None)
```

就可以显示出如上的界面了。

编写 javascript 实现 ajax

最后一步就是通过 javascript 实现：点击“加入购物车”时，调用 server 端的 RESTful API，同时更改界面上购物车里的内容。

这部分内容，留到下一节。

17、ajax !

继续上一节未完成任务，现在让我们来通过 ajax 请求后台服务。当然首选要实现后台服务。关于“加入购物车”，我们需要的服务是这样定义的：

[plain] view plaincopy

url: http://localhost:8000/depotapp/API/cart/items/post

post 数据: product = product_id

处理过程: 根据 product_id , 将 product 加入购物车

返回 : 购物车中的所有条目

这个 API 的定义似乎不那么 RESTful , 但是暂且不去管它。实现这个服务需要为 [RESTful web service](#) (depotapp/views.py 中的 RESTforCart 类) 增加一个方法 :

[python] view plaincopy

```
def post(self, request, *args, **kwargs):  
    print request.POST['product']  
    product = Product.objects.get(id=request.POST['product'])  
    cart = request.session['cart']  
    cart.add_product(product)  
    request.session['cart'] = cart  
    return request.session['cart'].items
```

可以通过 <http://localhost:8000/depotapp/API/cart/items/post> 来测试服务接口
(使用 Firebug 调试是非常方便的办法) :

Line Item

204 NO CONTENT
Vary: Authenticate, Accept
Allow: GET, POST, HEAD

GET Line Item
GET [json] [html] [xhtml] [txt] [xml]

POST Line Item
Product
Unit price
Quantity



如同你看到的那样，我们的接口定义不是完全 RESTful，在生成的表单中，我们只需要选择 Product，不用管另外的两个表单项，POST 之后就可以从之前实现的购物车界面中看到新增加的产品项了。

服务接口测试通过，就可以在界面中通过 ajax 调用了。jquery 对 ajax 提供了丰富的支持，为了方便使用 jquery 的 selector，先要对 html 进行改造。将上一节实现的 depot/templates/depotapp/store.html 中，迭代产品的部分改成如下的样子：

[html] view plaincopy

```
{% for item in products %}

<div class="row" style="padding-top:10">

  <div class="span3 media-grid">

    <a href="#">

    </a>

  </div>

  <div class="span6">

    <h3>{{item.title}}</h3>

    <br/>

    {{item.description}}

    <br/>

    <br/>

    <br/>

    <div class="row">

      <div class="span2"><h3>¥ {{item.price|floatformat:"2"}}</h3></div>

      <div class="span"><a class="btn primary" productid="{{item.id}}" href="#">加入购物车</a></div>

    </div>

  </div>

</div>

</div>

<div class="page-header">

</div>

{% endfor %}
```

其中主要更改了“加入购物车”的<a>标签，增加 productid 属性，并将 href 改为“#”。这样我们就可以很方便的为其添加事件：

[javascript] view plaincopy

```
//store.html on ready

$('a.btn[productid]').bind("click",function(){
    alert($(this).attr("productid"));
});
```

这段代码实现的功能是：对于所有的标签<a>，如果 class 包括“btn”，并且拥有“productid”属性的元素，添加 click 事件，弹出对话框显示其“productid”属性值。

打开产品清单界面测试一下，能够正确弹出产品 ID，然后就可以编写 ajax 的处理了。在这里我们使用 `jquery.post()` 方法，`jquery.post()` 是 `jquery.ajax` 的简化写法，如下：

[javascript] view plaincopy

```
//store.html on ready

$('a.btn[productid]').bind("click",function(){
    var product_id=$(this).attr("productid");
    //alert(product_id);
    $.post("/depotapp/API/cart/items/post",
        {product:product_id},
        function(data){
            alert(data);
        }
    );
});
```

弹出对话框显示的 data 就是前面定义的 API 接口的返回值，即现有购物车中的条目列表。

最后，要根据返回的数据更改界面上的购物车显示。这里为了方便也对 html 进行了改造。整个完成的 depot/templates/depotapp/store.html 如下：

[html] view plaincopy

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

```
{% block title %} 产品目录 {% endblock %}
```

```
{% block pagename %} 产品目录 {% endblock %}
```

```
{% block content %}
```

```
<div class="row">
```

```
<div class="span10">
```

```
{% for item in products %}
```

```
<div class="row" style="padding-top:10">
```

```
<div class="span3 media-grid">
```

```
<a href="#">
```

```

```

```
</a>
```

```
</div>
```

```
<div class="span6">
```

```
<h3>{{item.title}}</h3>
```

```
<br/>
```

```
{{item.description}}
```

```
<br/>
```

```
<br/>
```

```
<br/>
```

```
<div class="row">
```

```
<div class="span2"><h3>¥ {{item.price|floatformat:"2"}}</h3></div>
```

```
<div class="span"><a class="btn primary" productid="{{item.id}}" href="#">加入购物车</a></div>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
<div class="page-header">
```

```
</div>
```

```
{% endfor %}
```

```

</div><!--span10-->
<div class="span4">

<h5>我的购物车</h5><br/>

<table id="tabCart" class="condensed-table">

  <tbody id="items">

</tbody>

<tfoot>

  <tr>

    <td></td>

    <th>总计 : </th>

    <td id="totalprice">¥{{cart.total_price|floatformat:"2"}}</td>

  </tr>

</tfoot>

</table>

<a class="btn danger" href="{% url depotapp.views.clean_cart %}">清空</a>

<a class="btn success" href="#">结算</a>

</div><!--span4-->
{% endblock %}
{% block js %}
<!--js from store.html-->
<script>
function refreshCart(items){
  total = 0;
  var tbody = $('tbody#items')[0];
  tbody.innerHTML = "";
  for(var i=0;i<items.length;i++){
    total+=items[i].quantity*items[i].unit_price;
    $('table#tabCart').append('<tr><td>'+items[i].quantity+'x</td>'+
      '<td>'+items[i].product+'</td><td>¥'+items[i].unit_price+
      '</td></tr>');
  }
}

```

```

    }

    $('#totalprice')[0].innerHTML = '$'+total;
}

</script>

{% endblock %}

{% block on_ready %}

//store.html on ready

$.getJSON('/depotapp/API/cart/items/',refreshCart);

$('a.btn[productid]').bind("click",function(){
    var product_id=$(this).attr("productid");

    //alert(product_id);

    $.post("/depotapp/API/cart/items/post",{product:product_id},refreshCart);
});

{% endblock %}

```

定义了一个 refreshCart 函数，根据参数“重绘”购物车界面。在\$(document).ready 部分，首先调用前面实现的 API 显示购物车，这样我们在模板中就可以去掉原来实现的“购物车”，改成 javascript 的方式。

然后为每个“加入购物车”按钮添加点击事件，调用本节开始部分实现的接口，根据返回的最新条目数据调用 refreshCart 函数重绘购物车。

上面的模板中，javascript 的部分划分成了两个 block：{% block js %}用于嵌入具体页面（相对应父模板）的 js 函数；{% block on_ready %}用于嵌入具体页面的\$(document).ready 处理。结合 base.html 中定义的 block，可以使组合在一起的具体页面和模板页面符合 Unobtrusive JavaScript。这样做应该是 Django+jquery 实现 ajax 的最佳实践。

18、提交订单

前面的内容已经基本上涵盖了 Django 开发的主要方面，我们从需求和界面设计出发，创建模型和修改模型，并通过 scaffold 作为开发的起点；在 scaffold 的基础上重新定制模板，并且通过 Model 类和 Form 类对用户输入的数据进行校验。我们也涉及到了单元测试。为了提高开发用户界面的效率，更好地实现模板，我们还讨论了对静态资源（css, js, image 等）的管理，并通过模板继承的方式实现了整个站点的统一布局。作为 web 应用必不可少的部分，我们还演示了如何使用会话（session）。最后，我们还在这些基础上增加了 RESTful web service，将 jquery 集成到 Django，并实现了 ajax。

有了这些基础，可以应付 Django 开发中的绝大多数问题。

下面我们继续实现 depot 购物车的 web 应用。本节要实现提交订单功能。现在买方已经可以挑选需要的产品放入购物车，但是还不能进行购买。我们希望实现这样的功能：

买方点击“结算”按钮，然后输入姓名、地址和 email 信息，就向卖方发出了一张订单，该订单包含上述买方信息及其选购的所有条目。首先还是实现 Model 类。我们要增加一个订单类（Order），并修改原来的条目类（LineItem），增加到 Order 的外键（即 LineItem 到 Order 的 many-to-one 关联）。如下：

[python] view plaincopy

```
1. class Order(models.Model):
2.     name = models.CharField(max_length=50)
3.     address = models.TextField()
4.     email = models.EmailField()
5.
6. class LineItem(models.Model):
7.     product = models.ForeignKey(Product)
8.     order = models.ForeignKey(Order)
9.     unit_price = models.DecimalField(max_digits=8, decimal_places=2)
10.    quantity = models.IntegerField()
```

要注意，因为 LineItem 引用了 Order，所以在定义模型类的顺序上要将 Order 放在前面。

前面介绍过修改模型类之后应该如何处理，这里就不再重复了。

然后祭出 `scaffold`，快速生成一系列“丑陋”的界面。该工具还不够成熟，在生成之前，一定要备份 `depotapp/urls.py`、`views.py` 和 `forms.py`，之后手工合并，切记！

之后将产品目录界面（`depot/templates/depotapp/store.html`）中的“结算”标签的链接改为映射到生成的 `create_order` 视图函数：

[html] view plaincopy

```
1. <a class="btn success" href="{% url depotapp.views.create_order %}">结算</a>
```

两个界面就链接起来了。但是自动生成的订单界面并没有将 `session` 中的条目保存到数据库，也没有清空购物车，并且提交订单后没有返回到产品目录界面，所以还要对其进行修改。主要的修改在 `create_order` 视图函数中进行。为了方便对比，将生成的原始视图函数列出如下：

[python] view plaincopy

```
1. def create_order(request):
2.     form = OrderForm(request.POST or None)
3.     if form.is_valid():
4.         form.save()
5.         form = OrderForm()
6.
7.     t = get_template('depotapp/create_order.html')
8.     c = RequestContext(request, locals())
9.     return HttpResponse(t.render(c))
```

关于事务处理

“保存订单”和“保存订单条目”应该形成一个事务。Django 的事务处理可以通过 `middleware` 自动添加，也可以手工添加。手工添加的方式更加灵活，通过在 `view` 函数前增加修饰符（`decorator`）来实现，有三种修饰符可以选择：

[plain] view plaincopy

1. `@transaction.autocommit` 在 `save()` 或 `delete()` 时自动提交事务。
2. `@transaction.commit_on_success` 当整个 `view` 成功后提交事务，否则回滚（`TransactionMiddleware` 采用的就是这种机制）
3. `@transaction.commit_manually` 需要手动调用 `commit` 或 `rollback`。

我们暂时不想使用 `middleware` 的方式，所以用 `@transaction.commit_on_success` 来实现事务管理。这需要引用 `transaction` 模块：`from django.db import transaction`
最终修改的 `create_order` 视图函数如下：

[python] view plaincopy

```
1. @transaction.commit_on_success
2. def create_order(request):
3.     form = OrderForm(request.POST or None)
4.     if form.is_valid():
5.         order = form.save()
6.         for item in request.session['cart'].items:
7.             item.order = order
8.             item.save()
9.         clean_cart(request)
10.    return store_view(request)
11.
12.    t = get_template('depotapp/create_order.html')
13.    c = RequestContext(request,locals())
14.    return HttpResponse(t.render(c))
```

19、自定义 many-to-many 关系，实现 Atom 订阅

记得有人跟我说过，rails 的 `has_many :through` 是一个“亮点”，在 Django 看来，该功能简直不值一提。rails 中的 many-to-many 关联中，还需要你手工创建关联表（写 migration 的方式），而 `has_many :through` 的“语法”只不过是为了自定义关联关系：通过一个中间的、到两端都是 many-to-one 的模型类实现多对多关联。在 Django 中，many-to-many 的中间关系表是自动创建的，如果你要指定一个自己的 Model 类作为关系对象，只需要在需要获取对端的 Model 类中增加一个 `ManyToManyField` 属性，并指定 `through` 参数。比如现在我们已经有了这样两个 many-to-one 关系，`LineItem ---> Product`，`LineItem --> Order`，如果需要从 `Product` 直接获取关联的 `Order`，只需要增加一行，最终的 `Product` 如下：

[python] view plaincopy

```
class Product(models.Model):
    title = models.CharField(max_length=100,unique=True)
    description = models.TextField()
    image_url = models.URLField(max_length=200)
    price = models.DecimalField(max_digits=8,decimal_places=2)
```

```
date_available = models.DateField()

orders = models.ManyToManyField(Order,through='LineItem')
```

之后就可以通过 product 对象直接找到包含该产品的订单：

[plain] view plaincopy

```
$ python manage.py shell

>>> from depot.depotapp.models import Product

>>> p = Product(id=1)

>>> p.orders

<django.db.models.fields.related.ManyRelatedManager object at 0x10be110>

>>> p.orders.all()

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

实现这个关系的目的是我们要针对每个产品生成一个“订阅”，用于查看谁买了该产品。我们采用 Atom 作为格式的标准。生成的 Atom 发布格式如下：

[html] view plaincopy

```
<?xml version="1.0" encoding="UTF-8"?>

<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">

  <id>tag:localhost,2005:/products/3/who_bought</id>

  <link type="text/html" href="http://localhost:3000/depotapp" rel="alternate"/>

  <link type="application/atom+xml" href="http://localhost:8000/depotapp/product/3/who_bought" rel="self"/>

  <title>谁购买了《黄瓜的黄 西瓜的西》</title>

  <updated>2012-01-02 12:02:02</updated>

  <entry>

    <id>tag:localhost,2005:Order/1</id>

    <published>2012-01-02 12:02:02</published>

    <updated>2012-01-02 12:02:02</updated>

    <link rel="alternate" type="text/html" href="http://localhost:8000/orders/1"/>

    <title>订单 1</title>

    <summary type="xhtml">

      <div xmlns="http://www.w3.org/1999/xhtml">

        <p>我住在北京</p>

      </div>

    </summary>
```

```

<author>

  <name>我是买家</name>

  <email>wanghaikuo@gmail.com</email>

</author>

</entry>

<entry>

  <id>tag:localhost,2005:Order/3</id>

  <published>2012-01-02 12:02:02</published>

  <updated>2012-01-02 12:02:02</updated>

  <link rel="alternate" type="text/html" href="http://localhost:8000/orders/3"/>

  <title>订单 3</title>

  <summary type="xhtml">

    <div xmlns="http://www.w3.org/1999/xhtml">

      <p>我住在哪里？</p>

    </div>

  </summary>

  <author>

    <name>我是买家 2</name>

    <email>2222b@baidu.com</email>

  </author>

</entry>

</feed>

```

你可能想到，Atom 是以 xml 为格式的，我们可以借助 [Django REST framework](#) 来实现，但是这不是一个好主意，因为 REST framework 生成的 xml 有其自身的格式，与 Atom 的格式完全不同。如果使用 REST framework 就需要对其进行定制，甚至要实现一个自己的 renderer（比如，AtomRenderer），而这需要深入了解该框架的大量细节。为了简单起见，我们考虑用 Django 的模板来实现。

首先我们设计 url 为：

`http://localhost:8000/depotapp/product/[id]/who_bought`，先在 `depot/depotapp/urls.py` 中增加 `urlpatterns`：

`[python]` [view plaincopy](#)

```
(r'product/(?P<id>[^/]+)/who_bought$', atom_of_order),
```

然后在 depot/depotapp/views.py 中实现视图函数：

[python] view plaincopy

```
def atom_of_order(request,id):  
    product = Product.objects.get(id = id)  
    t = get_template('depotapp/atom_order.xml')  
    c=RequestContext(request,locals())  
  
    return HttpResponse(t.render(c), mimetype='application/atom+xml')
```

注意其中我们指定了 mimetype，使浏览器知道返回的是 xml 而不是 html。最后的工作就是编写模板了。depot/templates/depotapp/atom_order.xml 如下：

[html] view plaincopy

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">  
    <id>tag:localhost,2005:/product/{{product.id}}/who_bought</id>  
    <link type="text/html" href="{% url depotapp.views.store_view %}" rel="alternate"/>  
    <link type="application/atom+xml" href="{% url depotapp.views.atom_of_order product.id %}" rel="self"/>  
  
    <title>谁购买了 《{{product.title}}》 </title>  
  
    <updated>2012-01-02 12:02:02</updated>  
  
    {% for order in product.orders.all %}  
        <entry>  
            <id>tag:localhost,2005:order/{{order.id}}</id>  
            <published>2012-01-02 12:02:02</published>  
            <updated>2012-01-02 12:02:02</updated>  
            <link rel="alternate" type="text/html" href="{% url depotapp.views.atom_of_order order.id %}" />  
  
            <title>订单{{order.id}}</title>  
  
            <summary type="xhtml">  
                <div xmlns="http://www.w3.org/1999/xhtml">  
                    <p>{{order.address}}</p>  
                </div>  
            </summary>  
  
            <author>  
                <name>{{order.name}}</name>  
                <email>{{order.email}}</email>
```

```
</author>

</entry>

{% endfor %}

</feed>
```

用模板实现 Atom 发布，确实很简单，不是吗？

20、分页 (Pagination)

在上一节我们实现了针对某个产品的订单订阅功能。但是我们可能需要直接在站点上查询所有的订单。显然，随着时间的增长订单会越来越多，所以分页 (Pagination) 是个好办法：每次只显示一部分订单。

分页是 Web 应用常用的手法，Django 提供了一个分页器类 Paginator (django.core.paginator.Paginator)，可以很容易的实现分页的功能。该类有两个构造参数，一个是数据的集合，另一个是每页放多少条数据。Paginator 的基本使用如下：

```
$python manage.py shell

>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)    #每页两条数据的一个分页器
>>> p.count    #数据总数
4
>>> p.num_pages    #总页数
2
>>> p.page_range    #页码的列表
[1, 2]
>>> page1 = p.page(1)    #第 1 页
>>> page1
<Page 1 of 2>
```

```

>>> page1.object_list    #第 1 页的数据
['john', 'paul']
>>> page2 = p.page(2)
>>> page2.object_list    #第 2 页的数据
['george', 'ringo']
>>> page2.has_next()    #是否有后一页
False
>>> page2.has_previous() #是否有前一页
True
>>> page2.has_other_pages() #是否有其他页
True
>>> page2.next_page_number() #后一页的页码
3
>>> page2.previous_page_number() #前一页的页码
1
>>> page2.start_index() # 本页第一条记录的序数 (从 1 开始)
3
>>> page2.end_index() # 本页最后一条记录的序数 (从 1 开始)
4
>>> p.page(0)            #错误的页，抛出异常
...EmptyPage: That page number is less than 1
>>> p.page(3)            #错误的页，抛出异常
...EmptyPage: That page contains no results

```

其实前面 scaffold 生成的内容里面就已经包含了分页的功能，相信有了对 Paginator 的了解，你自己就可以看懂在 view 函数和模板中如何使用分页器了。

21、使用内置的 Amin 管理用户

到目前为止，我们开发的所有功能都是匿名访问的，这显然不够安全。通常会要求注册的用户通过用户名和密码登录，只有登录后的用户才可以管理产品。套用专业的说法就是：第一步是认证，验证用户是否是他所宣称的那个人；第二步是授权，验证用户是否拥有执行某种操作的权限。

Django 已经提供了一个 `django.contrib.auth` 应用来处理登录、登出和权限验证，同时还提供了 `django.contrib.admin` 应用可以进行用户管理（`admin` 应用还有很多其他的功能）。所以我们只需要将这些 app 插入到我们的站点，就可以实现登录和用户管理的大部分功能。

本节先介绍如何进行用户管理。

我们前面已经介绍过在 [Django 中使用 session](#)，方法是打开 `'django.contrib.sessions'` 应用。该 app 是 `django.contrib.auth` 的基础，同时从该节我们也知道了如何在 Django 中加入应用，所以让我们快速行动起来：

首先在 `depot/settings.py` 中，打开 `MIDDLEWARE_CLASSES` 中的 `CommonMiddleware`、`SessionMiddleware` 和 `AuthenticationMiddleware`：

`[python]` view plaincopy

```
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',  
)
```

同样在 `depot/settings.py` 中，打开 `INSTALLED_APPS` 中的 `auth`、`contenttypes`、`sessions` 和 `admin` 应用：

`[python]` view plaincopy

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



```

    #'django.contrib.sites',

    #'django.contrib.messages',

    #'django.contrib.staticfiles',

    # Uncomment the next line to enable the admin:

    'django.contrib.admin',

    # Uncomment the next line to enable admin documentation:

    #'django.contrib.admindocs',

    'depot.depotapp',

    'django-groundwork',

    'djangorestframework',

)

```

在 depot/urls.py 中，在头部增加：

[\[python\] view plaincopy](#)

```

from django.contrib import admin

admin.autodiscover()

```

然后增加 admin 的 urlpattern：(r'^admin/', include(admin.site.urls)),

最后运行 `$python manage.py syncdb`，以创建需要的数据库表。在此过程中会询问你创建一个管理员账号。如果没有创建，也可以手工运行 `$python manage.py createsuperuser` 再次创建：

```

$ python manage.py createsuperuser
Username (Leave blank to use 'holbrook'):
E-mail address: a@b.com
Password:
Password (again):
Superuser created successfully.
holbrook-wongdemacbook-pro:depot holbrook$ python manage.py syncdb
Creating tables ...
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

```

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): yes

Username (Leave blank to use 'holbrook'):
E-mail address: wanghaikuo@gmail.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
No fixtures found.

此时访问 <http://localhost:8000/admin/>，用刚才创建的用户名和密码登录，就可以看到 Django 内置的管理界面，其中就有用户管理的功能。如果你对英文界面很不爽，只需要在 `depot/settings.py` 中设置 `LANGUAGE_CODE = 'zh-CN'`，就可以看到中文的管理界面了。

你可以创建几个用户，在下一节中会用到。

22、处理登录和注销

前一节我们实现了用户管理，本节对用户登录和登出进行处理。由于我们已经引入了 `django.contrib.auth` 应用，用户的登录和登出处理变得非常简单。

我们已经可以在 `view` 函数中判断用户是否已经登录以及获取用户信息：

```
[python] view plaincopy
```

```
1. if request.user.is_authenticated():    #判断用户是否已登录
2.     user = request.user;               #获取已登录的用户
3. else:
4.     user = request.user;               #非登录用户将返回 AnonymousUser 对象
```

Django 的 `User` 对象提供了一系列的属性和方法，其中 `password` 存储的是加密后的密码，`is_staff` 记录用户是否有管理员权限，其他的属性可以参考官方文档。同时 `django.contrib.auth` 模块中提供了 `authenticate()`、`login()`、`logout()` 等函数，分别实现认证、登录、登出等功能。

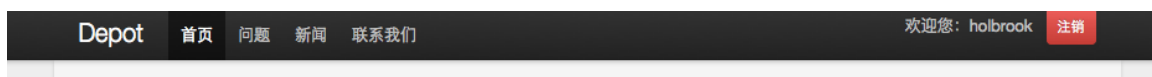
Django 还为我们提供了内置的处理 `login`、`logout` 的 `view` 函数，但是因为其提供的行为与我们这里要的不一样，所以还需要自己实现 `view` 函数。

我们使用 `bootstrap` 模板设计的布局中，在菜单栏已经放入了登录表单,如下图：



如上一节所述，我们希望只有登录后的用户才可以管理产品，所以在右上角输入用户名和密码并登录后，该处显示用户名和“注销”按钮，而页面重定向的产品管理的界面。

。而注销后再恢复成登录表单。



需求已经明确了，现在开始实现。对于 url 配置，由于 login 和 logout 功能应该属于整个 project，而不是特定的 app，所以在 depot/urls.py 中配置：

[python] view plaincopy

```
1. from depotapp.views import login_view,logout_view
2. urlpatterns = patterns('
3.     ...
4.     (r'^accounts/login/$', login_view),
5.     (r'^accounts/logout/$', logout_view),
6. )
```

而我们的设计的界面中没有独立的登录、注销界面，所以 view 函数还是在 depotapp 的 views.py 中实现：

[python] view plaincopy

```
1. from django.contrib.auth import authenticate,login,logout
2.
3. def login_view(request):
4.     user = authenticate(username=request.POST['username'], password=request.POST['password'])
5.     if user is not None:
6.         login(request, user)
7.         print request.user
8.         return list_product(request)
9.     else:
10.        #验证失败，暂时不做处理
11.        return store_view(request)
12.
13. def logout_view(request):
```

```
14.     logout(request)
15.     return store_view(request)
```

所有的界面都在 base.html 模板中，将 base.html 的 topbar 中原来登录表单的部分，改成如下的样子：

[html] view plaincopy

```
1.  {% if request.user.is_authenticated %}
2.      <div class="pull-right">
3.          <a href=#>欢迎您: {{request.user}}</a>
4.          <a class="btn danger small" href="{% url depotapp.views.logout_view %}">注
销</a>
5.      </div>
6.  {% else %}
7.      <form action="{% url depotapp.views.login_view %}" method='post' class="pull-
right">
8.          {% csrf_token %}
9.          <input name='username' class="input-small" type="text" placeholder="用户名
">
10.         <input name='password' class="input-small" type="password" placeholder="
密码">
11.         <button class="btn" type="submit">登录</button>
12.     </form>
13. {% endif %}
```

所有的工作就完成了。

23、权限控制

我们已经实现了登录和注销功能，但是它还没有起作用。因为匿名用户还是可以通过直接输入 url：http://localhost:8000/depotapp/product/list/ 访问到产品管理界面。这就好像你在门上加了把锁，却没有把窗户关上一样。所以我们还需要进行访问控制。

我们这里实现最简单的控制，非登录用户禁止访问产品管理界面。在 Django 里面，只需要在相应的视图函数前面增加@login_required 修饰符即可：

[python] view plaincopy

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

... ..

@login_required

def list_product(request):

list_items = Product.objects.all()

... ..

login_required 实现了如下功能:

如果用户没有登录, 重定向到/accounts/login/, 并且把当前绝对 URL 作为 next 参数用 get 方法传递过去;

如果用户已经登录, 正常地执行视图函数。

这样我们例子中需要的功能就实现了。但是这充其量是“登录限制”, 而更常见的需求是“访问控制”, 即区分已经登录的用户, 对不同的视图有不同的访问权限。因为我们的例子中没有涉及到, 所以只把相关的做法简单列举在下面。

1. 访问控制功能通过 Django 的 request.user.has_perm () 来实现, 该函数返回 True 或 False, 表示该用户是否有权限。而权限是 auth 应用中定义的 Permission 类型; User 与 Permission 是 many-to-many 的关系。

2. Django 还提供了一个@permission_required 修饰符, 来限定 view 函数只有在 User 具有相应权限的情况下才能被访问。

3. Django 对于每个模型类, 自动增加 add、change、delete 三种权限, 以便于权限控制。当然你也可以设定自己的权限。

24、总结

这个实战系列暂时就告一段落了, 时间和能力的关系, 有些地方写的不够清楚明白, 代码也没有整理出来。未尽事宜将在以后逐步补全。

希望通过这个系列, 你能够掌握使用 Django 开发一个 web 应用的基本过程。现在简单整理一下《Django 实战系列》的内容:

0. 如果你以前没有接触过 Django, 你可能需要这些准备知识:

[URLconf+MTV : Django 眼中的 MVC](#)

Django 第一步

1. 实战系列的开发目标

需求分析和设计

2. 从 Model 开始

创建第一个模型类

3. Model 之外，你还需要知道什么

Django 也可以有 scaffold

scaffold 生成物分析

4. 关于界面：静态资源，模板，及其使用

引入 bootstrap，设置静态资源

对比 RoR 和 Django 的模板系统

改造 ProductList 界面

5. 逻辑层

对比 RoR 与 Django 的输入校验机制

实现 Product 的输入校验

单元测试

6. 变更

修改 Model 类

增加目录页，设定统一布局

7. 关于会话

在 session 中保存购物车

让页面联动起来

8. ajax

Django 实现 RESTful web service

[Django+jquery](#)

[ajax !](#)

9. 另一轮变更

[提交订单](#)

[自定义 many-to-many 关系，实现 Atom 订阅](#)

[分页 \(Pagination\)](#)

10. 用户和权限

[使用内置的 Amin 管理用户](#)

[处理登录和注销](#)

[权限控制](#)

通过这个系列，你是否对 Django 有了初步的认识，并希望能够更加深入呢？计划下一个系列写《深入 Django》，目前能想到的内容（顺序无关）包括：

[\[plain\] view plaincopy](#)

[WSGI，Python 的 Web 服务网关接口](#)

[国际化](#)

[部署和发布](#)

[Django 控制台和调试](#)

[Django 最佳实践，比如 project 目录结构的设定等](#)

[深入 model，包括关联关系，Model API，Manager，直接使用 sql 语句等](#)

[Django 中的中间件 \(middleware \)](#)

[站点 \(Site \)](#)

[定制 Admin](#)

[深入模板](#)

[深入 Form](#)

[深入认证和授权](#)

... ..

如果你有好的想法或者建议，请一定留言，谢谢！

从 1 月 29 日至 2 月 10 日，13 天的时间写了关于 Django 的博文 27 篇（包括本实战系列 24 篇和另外 3 篇），并设立了 Django 专栏。先给自己鼓一下劲！