第6章介绍了Django的admin界面，现在是回过头来仔细看看这个的时候了   
我们前面讲的几次admin是Django的"杀手级特性"，并且大多数Django开发人员很快爱上了它节省时间的所有特性   
这样自然而然的大部分Django开发人员开始寻找自定义或者扩展admin的方法   
第6章最后几部分讲到了一些定制admin界面某一部分的简单方法，重新阅读一下那些资料是个好主意   
它讲述了一些定制admin的更改列表，编辑表单以及logo等等的简单方法   
第6章也讨论了何时和为什么你想使用admin界面，这些资料跳跃到了其他章节，我们这里重新介绍一下:   
显然，admin对编辑数据非常有用(fancy that)，如果你有一些录入数据的任务，则admin不可能被其它东西打败   
我们料想大多数本书的读者都将有很多数据录入的任务   
Django的admin在非技术用户需要录入数据时特别闪耀，这是这个特性的最初起源   
尽管如此，我们发现除了显而易见的数据录入任务，admin也在下面一些情况下有用:   
1，检查数据模型，我们定义了一个新模型后第一件事就是在admin里调用它并输入一些模拟数据，这对我们发现数据   
模型的错误并有一个图形界面来显示这些错误很有帮助   
2，管理必须的数据，对于chicagocrime.org来说很少有数据录入的任务，因为它的数据都来自于一个自动的数据源   
尽管如此，当自动获取数据的模块出问题时，通过admin可以轻松的编辑数据，这是很有用的   
Django的admin不需要或者需要很少配置就可以处理这些常见的情况，但是，处理这些常见的情况如此的好意味着   
Django的admin在处理其它情形时不一定很好   
我们后面将谈到Django的admin不适合做的一些事情，但是现在我们先离题来看看它的一些哲学:   
  
**admin的禅宗**   
作为它的核心，Django的admin设计用来为如下的一个单独的活动:   
受信任的用户编辑结构化的内容   
是的，很简单，但是这简单的一行隐藏着很多内容，Django的admin的整个哲学都基于此   
让我们深入了解这个句子的子内容:   
"受信任的用户"   
admin设计来被你(开发者)信任的人用，这不仅仅表示那些被授权的用户，它表示Django假设你的内容编辑者可以   
被信任来做正确的事情，这意味着编辑内容没有批准的过程，如果你信任你的用户，没有人需要对编辑的批准   
这也表明了权限系统不支持基于一个对象的限制访问   
如果你信任某人来编辑他自己的故事，你也将信任他不会在没有权限的情况下编辑别人的故事   
"编辑"   
Django的admin的首要目的是让人们编辑内容，这最初看起来很显而易见，但是也存在一些细小而强大的影响   
例如，尽管admin对重新视查数据很有用，但是它不是设计来干这个的，注意缺少"can view"权限(参考第12章)   
Django假设如果用户被允许在admin里查看内容，他们也被允许编辑它   
另外一个很值得注意的地方是admin缺少一些例如"工作流"的东西，如果一些任务需要几步来完成，admin不支持   
特别的顺序来做这件事情，admin关注于编辑，而不是围绕编辑的其它活动   
对于工作流的缺乏支持也起源于信任的原则，admin的哲学是，工作流属于个人问题，而不应该用代码实现   
最后，注意admin缺乏统计的支持，它不支持显示总数，平均数等等   
再一次说明，admin是用来编辑的，它期望你写自定义的视图来完成其它的任务   
"结构化的内容"   
因为Django其它部分的关系，admin希望你与结构化的数据工作，这样，admin仅仅支持编辑用Django模型存储的数据   
对于其它形式的数据，你则需要自定义视图   
总结   
现在应该很清楚了，Django的admin不是给任何用户来做任何事情的，而是牢牢的关注一点并且把这一点做的非常好   
当我们需要扩展Django的admin时，同一哲学的大部分内容存在与此(注意扩展性无处不在)   
因为自定义的Django视图可以做任何事情，而且它们可以可视化的集成到admin(参看下面内容)，内建的定制admin的   
机会在一定程序上被设计所限制   
  
**定制admin模板**   
我们下面将看到，你有几种工具来定制内建的admin模板，但是对于其它任务，例如需要自定义工作流或者细粒度权限   
你将需要阅读本章末尾讲到的定制admin视图   
现在我们来看看快速定制admin的外观和行为，第6章讲到了一些常见的任务，如更改logo样式和提供自定义admin表单   
就这点来说，我们通常需要更改一个特殊项的一些模板   
admin的每一个视图，如更改列表，编辑表单，删除确认页面，历史视图等都有一个分配的模板   
而这个模板可以通过几种方式来覆盖   
首先，你可以全局覆盖模板，admin视图使用标准模板载入机制来寻找模板，所以如果你在你的模板目录里创建模板   
Django将载入并使用这些模板而不是使用Django绑定的默认admin模板   
这些全局模板如下:   
视图 基本模板名   
更改列表 admin/change\_list.html   
增加/编辑表单 admin/change\_form.html   
删除确认 admin/delete\_confirmation.html   
对象历史 admin/object\_history.html   
尽管如此，大多数情况下你只想更改一个单独的对象或者app的模板而不是全局的模板   
这样的话，每个admin视图首先寻找模型和app专有的模板，这些视图按下面的顺序寻找模板:   
admin/<app\_lable>/<object\_name>/<template>.html   
admin/<app\_lable>/<template>.html   
admin/<template>.html   
例如，在bookstore app的Book模型的增加/编辑表单的视图(第6章的例子)按下面的顺序寻找模板:   
admin/bookstore/book/change\_form.html   
admin/bookstore/change\_form.html   
admin/change\_form.html   
  
**定制模型模板**   
大多数情况下，你想使用上面第一个模板来创建模型专有的模板   
通常情况下通过扩展基本模板并在其中的块定义中添加信息会将这个任务完成的最好   
例如我们想在book页面顶端添加一些帮助内容，可能像下面这样:   
[img][/img]   
这很容易做到，创建一个叫admin/bookstore/book/change\_form.html的模板并且插入下面的代码:

Java代码

1. {% **extends** "admin/change\_form.html" %}
3. {% block form\_top %}
4. <p>Insert meaningful help message here..</p>
5. {% endblock %}

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. {% extends "admin/change\_form.html" %}
3. {% block form\_top %}
4. <p>Insert meaningful help message here..</p>
5. {% endblock %}

所有的这些模板都定义了一些块来让你覆盖，对于大多数程序，代码就是最好的文档，所以我们鼓励你浏览admin模板   
(在django/contrib/admin/templates/里面)来得到最新的信息   
  
**定制JavaScript**   
使用这个自定义的模型模板最常见的用途就是添加自定义的JavaScript到admin页面，可能是实现一些特殊的小窗口部件   
或者是客户端行为   
幸运的是，这再简单不过了，每个admin模板定义了一个{% block extrahead %}，你可以把使用它来把其它的内容添加   
到head元素里去，例如你想在你的一个admin历史页面引入jQuery:

Java代码

1. {% **extends** "admin/object\_history.html" %}
3. {% block extrahead %}
4. <script src="http://media.example.com/javascript/jquery.js" type="text/javascript"></script>
5. <script type="text/javascript">
7. // code to actually use jQuery here...
9. </script>
10. {% endblock %}

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. {% extends "admin/object\_history.html" %}
3. {% block extrahead %}
4. <script src="http://media.example.com/javascript/jquery.js" type="text/javascript"></script>
5. <script type="text/javascript">
7. // code to actually use jQuery here...
9. </script>
10. {% endblock %}

我不知道为什么你在对象历史页面需要jQuery，但是这个例子适用于admin的任何模板   
你可以使用这个技术来引入任何其它你可能需要的JavaScript小窗口部件   
  
**定制admin视图**   
到目前为止那些想添加自定义行为到Django的admin中的人们可能开始困惑了，他们会喊，"你所讲述的都是关于怎样改变   
admin的外观，但是我怎样改变admin的工作方式呢?"   
好了，别喊了，这里就是答案   
需要理解的第一件事就是它一点也不神奇，admin做的任何事都不特殊，它只是一些像其它视图一样处理数据的视图罢了   
这些视图在django.contrib.admin.views，当然这里有很多代码，它必须处理所有的选项，域类型和影响模型行为的设置   
同样的，当你意识到admin只是一些视图时，添加自定义的admin视图就变得更容易理解   
让我们添加一个"publisher report"视图到我们第6章的book app中，我们将构建一个admin视图来显示通过publisher   
分组的books列表，这是一个非常典型你可能想构建的自定义admin"report"的例子   
首先我们在URLconf里面包装一个视图，我们需要把这行代码插入到admin视图的引入行之前

Java代码

1. (r'^admin/bookstore/report/$', 'bookstore.admin\_views.report'),

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. (r'^admin/bookstore/report/$', 'bookstore.admin\_views.report'),

完整的URL配置可能像下面这样:

Java代码

1. from django.conf.urls.defaults **import** \*
3. urlpatterns = patterns('',
4. (r'^admin/bookstore/report/$', 'bookstore.admin\_views.report'),
5. (r'^admin/', include('django.contrib.admin.urls')),
6. )

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. from django.conf.urls.defaults import \*
3. urlpatterns = patterns('',
4. (r'^admin/bookstore/report/$', 'bookstore.admin\_views.report'),
5. (r'^admin/', include('django.contrib.admin.urls')),
6. )

为什么把自定义视图放在admin引入之前?回想一下Django处理URL模式的顺序，因为admin的引入URL匹配几乎所有的东西   
如果我们把上面的两行URL配置代码调换顺序，Django将会查找一个内建的视图来匹配这个URL，这将不能工作   
在这种特殊情况下，Django将试图载入bookstore app的Report模型的更改列表，这是不存在的   
现在让我们来写我们的视图，为了简单起见，我们只是载入所有的books在context里并让模板使用{% regroup %}标签处理   
分组，用下面的代码创建一个bookstore/admin\_views.py文件:

Java代码

1. from bookstore.models **import** Book
2. from django.template **import** RequestContext
3. from django.shortcuts **import** render\_to\_response
4. from django.contrib.admin.views.decorators **import** staff\_member\_required
6. @staff\_member\_required
7. def report(request):
8. **return** render\_to\_response(
9. "admin/bookstore/report.html",
10. {'book\_list' : Book.objects.all()},
11. RequestContext(request, {}),
12. )

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. from bookstore.models import Book
2. from django.template import RequestContext
3. from django.shortcuts import render\_to\_response
4. from django.contrib.admin.views.decorators import staff\_member\_required
6. @staff\_member\_required
7. def report(request):
8. return render\_to\_response(
9. "admin/bookstore/report.html",
10. {'book\_list' : Book.objects.all()},
11. RequestContext(request, {}),
12. )

因为我们把分组留给模板来做，这个视图非常简单，尽管如此，这里有一些细小的东西值得解释:   
1，我们使用django.contrib.admin.views.decorators的staff\_member\_required装饰器，它类似于第12章讨论的   
login\_required装饰器，但是这个还检查给定的用户是否标记为"staff"成员来决定是否允许访问admin   
这个装饰器保护所有内建的admin视图，让你的视图的认证逻辑和admin的其它部分匹配   
2，我们渲染在admin/下面的模板，虽然这没有严格的要求，但是保持你所有的admin模板分组在一个admin目录下   
被认为是最佳实践，我们把模板放在我们的app后面叫bookstore的目录下也是最佳实践   
3，我们使用RequestContext作为第3个参数(context\_instance)传递给render\_to\_response   
这保证了关于当前用户的信息可以在模板里得到，参看第10章得到更多关于RequestContext的信息   
最后我们将为这个视图创建一个模板，我们继承内建的admin模板来使这个视图视觉上看起来是admin的一部分:

Java代码

1. {% **extends** "admin/base\_site.html" %}
3. {% block title %}List of books by publisher{% endblock %}
5. {% block content %}
6. <div id="content-main">
7. <h1>List of books by publisher:</h1>
8. {% regroup book\_list|dictsort:"publisher.name" by publisher as books\_by\_publisher %}
9. {% **for** publisher in books\_by\_publisher %}
10. <h3>{{ publisher.grouper }}</h3>
11. <ul>
12. {% **for** book in publisher.list|dictsort:"title" %}
13. <li>{{ book }}</li>
14. {% endfor %}
15. </ul>
16. {% endfor %}
17. </div>
18. {% endblock %}

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. {% extends "admin/base\_site.html" %}
3. {% block title %}List of books by publisher{% endblock %}
5. {% block content %}
6. <div id="content-main">
7. <h1>List of books by publisher:</h1>
8. {% regroup book\_list|dictsort:"publisher.name" by publisher as books\_by\_publisher %}
9. {% for publisher in books\_by\_publisher %}
10. <h3>{{ publisher.grouper }}</h3>
11. <ul>
12. {% for book in publisher.list|dictsort:"title" %}
13. <li>{{ book }}</li>
14. {% endfor %}
15. </ul>
16. {% endfor %}
17. </div>
18. {% endblock %}

通过继承admin/base\_site.html我们"免费"得到Django的admin的外观，它看起来像这样:   
[img][/img]   
  
**今天你需要在哪里使用admin?**   
你可以使用这个技术来向admin添加任何你想到的东西，记住所谓的"定制admin视图"事实上只是普通的Django视图   
你可以使用你在本书其它部分所学的所有技术来构建任意复杂的admin视图   
我们将以一些自定义admin视图的一些好注意结束本章内容   
  
**覆盖内建的视图**   
默认的admin视图不包含这些，你可以很轻松的在admin的任何地方跳转到你的自定义视图，只需让你的URL覆盖掉内建的那些   
例如，我们可以用一个简单的让用户输入ISBN的表单替代内建的book创建视图，然后我们就可以从http://isbn.nu/来查询   
book信息和自动创建对象   
这个视图的代码留给读者做练习，最重要的部分是下面的URL配置:

Java代码

1. (r'^admin/bookstore/book/add/$', 'bookstore.admin\_views.add\_by\_isbn'),

**[java]** [view plaincopy](http://blog.csdn.net/robinliu2010/article/details/7649190)

1. (r'^admin/bookstore/book/add/$', 'bookstore.admin\_views.add\_by\_isbn'),

如果这段代码在你的URL配置中放在admin的URL前面的话，add\_by\_isbn视图将完全替代标准的admin视图   
我们可以遵循类似的动作来替代删除确认页面，编辑页面或者admin的任何其它部分