C++

1. 在使用 fstream 从文件中读入内容时，每个 string 用空格或换行符隔开。（fstream 读入内容、空格或换行符隔开）

```
fstream finout;
string s1,s2,n;
finout.open("myFile.txt",ios::in);
finout>>s1>>s2;
cout<<"s1 的内容是:"<<s1<<endl;
cout<<"s2 的内容是:"<<s2<<endl;
```

假设 myFile.txt 中的内容如下：

```
linux 19891114
```

则输出如下：

```
s1 的内容是:linux
s2 的内容是:19891114
```

2. QT 无法输入中文可能是由于文件的编码文件，windows 的命令行使用 utf-8 编码无法输出中文，可以使用记事本来修改文件编码为 GBK。当在程序中使用中文字符时，需要把 QT 的默认编码也改成 GBK。（QT 默认编码）

Django

1. request.Get.get()用于获取 request 请求中的请求参数：（request.Get.get、获取请求参数）

```
def search(request):
    q = request.GET.get('q')              #其中 q 是搜索框的 name
    error_msg = ''

    if not q:
        error_msg = "请输入关键词"
        return render(request, 'blog/index.html', {'error_msg': error_msg})

    post_list = Post.objects.filter(Q(title__icontains=q) | Q(body__icontains=q))
    return render(request, 'blog/index.html', {'error_msg': error_msg,
```

2. 使用 Django 自带的 Paginator 类可以实现分页效果。（Paginator、分页效果）

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
from django.shortcuts import render

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25)       # 每页 25 条内容

    page = request.GET.get('page')
    contacts = paginator.get_page(page)
```

```
    return render(request, 'list.html', {'contacts': contacts})
```

3. 调用了 Paginator 分页之后，只是把列表分成了几页，还需要在前端代码显示分页栏样式:（Paginator 分页后、前端显示）

```
<div class="pagination">
    <span class="step-links">
        {% if contacts.has_previous %}
            <a href="?page=1">&laquo; 第 1 页</a>
            <a href="?page={{ contacts.previous_page_number }}">前一页</a>
        {% endif %}

        <span class="current">
            Page {{ contacts.number }} of {{ contacts.paginator.num_pages }}.
        </span>

        {% if contacts.has_next %}
            <a href="?page={{ contacts.next_page_number }}">下一页</a>
            <a href="?page={{ contacts.paginator.num_pages }}">最后一页 &raquo;</a>
        {% endif %}
    </span>
</div>
```

4. Paginator 类拥有以下方法和属性:

(1)方法:

- Paginator.page(number)：返回指定页面的对象列表，比如第 7 页的所有内容，下标以 1 开始。如果提供的页码不存在，抛出 InvalidPage 异常。（page、指定页面、对象列表）

(2)属性:

- Paginator.count：所有页面的对象总数。（count、对象总数）
- Paginator.num_pages：页面总数。（num_pages、页面总数）
- Paginator.page_range：基于 1 的页数范围迭代器。（页数范围）

5. Paginator.page()将返回一个 Page 对象，我们主要的操作都是基于 Page 对象的，它具有下面的方法和属性:

(1)方法:

- Page.has_next()：如果有下一页，则返回 True。（has_next()、有下一页）
- Page.has_previous()：如果有上一页，返回 True。（has_previous、有上一页）
- Page.has_other_pages()：如果有上一页或下一页，返回 True。（has_other_pages、有下一页或上一页）
- Page.next_page_number()：返回下一页的页码。如果下一页不存在，抛出 InvalidPage 异常。（next_page_number、返回下一页）
- Page.previous_page_number()：返回上一页的页码。如果上一页不存在，抛出 InvalidPage 异常。（previous_page_number、返回上一页）
- Page.start_index()：返回当前页上的第一个对象，相对于分页列表的所有对象的

序号，从 1 开始计数。 比如，将五个对象的列表分为每页两个对象，第二页的 start_index()会返回 3。（start_index、当前页、第一个对象）

- Page.end_index():返回当前页上的最后一个对象，相对于分页列表的所有对象的序号，从 1 开始。 比如，将五个对象的列表分为每页两个对象，第二页的 end_index()会返回 4。

(2)属性:

- Page.object_list:当前页上所有对象的列表。（object_list、对象列表）
- Page.number:当前页的序号，从 1 开始计数。（number、当前页号）
- Page.paginator：当前 Page 对象所属的 Paginator 对象。

6. 使用类似下列代码自定义标签：（自定义标签、template.Library、simple_tag）

```
blog/templatetags/blog_tags.py

from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def get_recent_posts(num=5):
    return Post.objects.all().order_by('-created_time')[:num]
```

首先导入 template 这个模块，然后实例化了一个 template.Library 类，并将函数 get_recent_posts 装饰为 register.simple_tag。这样就可以在模板中使用语法 {% get_recent_posts %} 调用这个函数了。

Simple_tag 是 django 提供的用于创建自定义标签的关键字。（Simple_tag、创建标签）

7. 为了安全起见，在生产环境下需要关闭 DEBUG 选项以及设置允许访问的域名。打开 settings.py 文件，找到 DEBUG 和 ALLOWED_HOSTS 这两个选项，将它们设置成如下的值：（生产环境、关闭 DEBUG）

```
blogproject/settings.py

DEBUG = False
ALLOWED_HOSTS = ['127.0.0.1', 'localhost ', '.zmrenwu.com']
```

ALLOWED_HOSTS 是允许访问的域名列表。

8. 处理表单时，如果需要验证输入的内容，使用 model 处理表单数据会比较麻烦，如果输入不符合格式则提交后的输入就会丢失，使用 form 可以解决这个问题。form 可以验证数据是否符合格式，某一些输入不合法也不会丢失已经输入的数据。

form 获取到表单内容后，对其进行判断是否符合要求，符合要求则保存到关联的 model 中去。（form、验证输入）

9. django 有两个处理表单的类：Form 和 ModelForm。如果表单内容有专门的 model，则使用 ModelForm，否则使用 Form。这两个类会自动为我们创建常规的表单代码。例如下例使用了 ModelForm 类：（Form、有专门的 model、ModelForm）

- model

```
comments/forms.py
```

```
from django import forms
from .models import Comment


class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ['name', 'email', 'url', 'text']
```

- Form：

```
comments/models.py

from django.db import models
from django.utils.six import python_2_unicode_compatible

# python_2_unicode_compatible 装饰器用于兼容 Python2
@python_2_unicode_compatible
class Comment(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(max_length=255)
    url = models.URLField(blank=True)
    text = models.TextField()
    created_time = models.DateTimeField(auto_now_add=True)

    post = models.ForeignKey('blog.Post')

    def __str__(self):
        return self.text[:20]
```

生成的 Form 类将按照 fields 属性中指定的顺序为每个指定的模型字段指定一个表单字段。

定义表单类后，在模板上可以直接使用该类：（定义后、模板上使用）

```
<form action="{% url 'comments:post_comment' post.pk %}" method="post"
class="comment-form">
  {% csrf_token %}
  <div class="row">
    <div class="col-md-4">
      <label for="{{ form.name.id_for_label }}">名字：</label>
      {{ form.name }}
      {{ form.name.errors }}
    </div>
    <div class="col-md-4">
      <label for="{{ form.email.id_for_label }}">邮箱：</label>
      {{ form.email }}
      {{ form.email.errors }}
    </div>
    <div class="col-md-4">
      <label for="{{ form.url.id_for_label }}">URL：</label>
```

```
    {{ form.url }}
    {{ form.url.errors }}
  </div>
  <div class="col-md-12">
    <label for="{{ form.text.id_for_label }}">评论：</label>
    {{ form.text }}
    {{ form.text.errors }}
    <button type="submit" class="comment-btn">发表</button>
  </div>
 </div>    <!-- row -->
</form>
```

{{ form.name }}、{{ form.email }}、{{ form.url }} 等将自动渲染成表单控件，例如 <input> 控件。

{{ form.name.errors }}、{{ form.email.errors }} 等将渲染表单对应字段的错误（如果有的话），例如用户 email 格式填错了，那么 Django 会检查用户提交的 email 的格式，然后将格式错误信息保存到 errors 中，模板便将错误信息渲染显示。（form.name.errors、填错了、渲染显示）

10. ModelForm 完整示例：

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
  ('MR', 'Mr.'),
  ('MRS', 'Mrs.'),
  ('MS', 'Ms.'),
)

class Author(models.Model):
  name = models.CharField(max_length=100)
  title = models.CharField(max_length=3, choices=TITLE_CHOICES)
  birth_date = models.DateField(blank=True, null=True)

  def __str__(self):              # __unicode__ on Python 2
    return self.name

class Book(models.Model):
  name = models.CharField(max_length=100)
  authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
  class Meta:
    model = Author
    fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
  class Meta:
    model = Book
    fields = ['name', 'authors']
```

11. 使用 Feed 类可以方便地实现 RSS 订阅。（Feed、RSS 订阅）

```
from django.contrib.syndication.views import Feed

from .models import Post


class AllPostsRssFeed(Feed):
    # 显示在聚合阅读器上的标题，用于在 item_title()中返回
    title = "Django 博客教程演示项目"

    # 通过聚合阅读器跳转到网站的地址
    link = "/"

    # 显示在聚合阅读器上的描述信息
    description = "Django 博客教程演示项目测试文章"

    # 需要显示的内容条目
    def items(self):
        return Post.objects.all()

    # 聚合器中显示的内容条目的标题
    def item_title(self, item):
        return '[%s] %s' % (item.category, item.title)

    # 聚合器中显示的内容条目的描述
    def item_description(self, item):
        return item.body
```

12. Django 中有两种视图，一种是函数式视图，另一种是类视图。视图的作用主要是，用于填充逻辑，返回响应体。函数式视图难以扩展，代码复用率低。而类视图可以利用继承、Mixins，快速复用、扩展功能。（函数式视图、类视图、类视图快速复用）

(1)函数式视图

views.py

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # 填充逻辑
        return HttpResponse('result')
    if request.method == 'POST':
        # 填充逻辑
        return HttpResponse('result')
```

urls.py

```
# urls.py
from django.conf.urls import patterns
import .views as home_view

urlpatterns = patterns('',
```

```
    (r'^my_view/', home_view.my_view)
)
```

(2)类视图

views.py

```
from django.http import HttpResponse
from django.views.generic.base import View

class MyView(View):
    def get(self, request):
        # 填充逻辑
        return HttpResponse('result')
    def post(self, request):
        # 填充逻辑
        return HttpResponse('result')
```

urls.py

```
from django.conf.urls import patterns
from .views import MyView

urlpatterns = patterns('',
    (r'^about/', MyView.as_view()),
)
```

类视图（CBV）提供了一个 as_view()静态方法，调用该方法，会创建一个类的实例。然后调用实例的 dispatch()方法，dispatch 查看请求以确定它是否为 GET、POST 等，并在请求被中继到匹配的方法（如果已定义），或者如果不是，则引发 HttpResponseNotAllowed。（类视图、ad_view()）

13. 类属性在很多的基于类的设计中都很有用。设置类属性有两个方法。

(1)第一个方法是标准的 python 方法：在子类中重写类的属性和方法，比如：（类属性、重写）

```
from django.http import HttpResponse
from django.views.generic.base import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

(2)第二个方法是在 URLconf 中将类属性作为参数传递给 as_view()：（类属性、作为参数传递）

```
urlpatterns = patterns('',
    (r'^about/', GreetingView.as_view(greeting="G'day")),
)
```

14. 上下文数据：实际上就是外部变量。一段依赖外部变量的代码如果没有数据，则程序

无法运行，这些外部变量就是上下文数据。（上下文数据、外部变量）

15. 类视图：

- TemplateView：可以在类中指定模板文件。（TemplateView、指定模板）
- ListView ：通过 context_object_name 来指定从 model 中返回的内容列表。执行这个视图的时候，self.object_list 将包含视图正在操作的对象列表。（ListView、内容列表）
- DetailView：执行这个视图的时候，self.object 将包含视图正在操作的对象。（DetailView、正在操作的对象）

```
def get_context_data(self, **kwargs):
    # 覆写 get_context_data 的目的是因为除了将 post 传递给模板外（DetailView 已经帮我们完成），
    # 还要把评论表单、post 下的评论列表传递给模板。
    context = super(PostDetailView, self).get_context_data(**kwargs)
    form = CommentForm()
    comment_list = self.object.comment_set.all()
    context.update({
       'form': form,
       'comment_list': comment_list
    })
    return context
```

覆写方法是先调用父类的函数，然后在后面更新该返回值。（覆写、先父类、后更新）

也就是说，对于一个博客来说，文章列表使用 ListView、而文章详情页使用 DetailView。（文章列表、ListView、文章详情页、DetailView）

- FormView：显示表单的视图，验证错误时，重新显示表单并显示错误信息；成功时，重定向到一个新的 URL。（FormView、成功、重定向）
- CreateView：显示用于创建对象的表单的视图，通过验证错误信息重新显示视图，并且保存对象。（CreateView、创建对象、视图）

16. DetailView：

(1) 属性：

- model：视图要显示的 model。
- queryset:：表示对象的一个查询集。queryset 的值优先于 model 的值。
- template_name：字符串表示的模板名称。
- context_object_name：指定在上下文中使用的变量的名称。
- pk_url_kwarg：URLconf 中，包含逐渐的关键字参数的名称.默认为'pk'。
- get_context_data：返回显示对象的上下文数据。

(2) 方法：

- get_queryset()：获取此视图的对象列表。必须是可迭代或者可以使查询集。默认返回 queryset 属性。可以通过重写该方法实现动态过滤。（get_queryset()、项目列表、可迭代）
- get_context_object_name()：获取此视图正在操作的数据的上下文变量名称。（get_context_object_name（）、上下文变量名称）
- get_context_data ()：返回显示对象的上下文数据，通过覆盖该方法返回额外的上下文。

（get_context_data()、上下文数据）

- get_object()：返回该视图将要显示的单个对象。（get_object()、将要显示）

➢ 说明：DatailView 的 get()实际上先调用了 get_object()获取到对象，然后调用 get_context_data()获取该对象相关的上下文数据。可以覆写这两个函数实现相关操作。例如如果要对获取的博客文章格式进行处理，则覆写 get_object()；如果要获取博客相关的评论、标签等上下文内容，则覆写 get_context_data()。

```
def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        return self.render_to_response(context)
```

17. ListView

(1) 属性

- model：指定模型
- tempalta_name：模板文件
- queryset：指定一个经过过滤的对象列表，将取代 model 提供的值
- context_object_name：指定要在上下文中使用的变量的名称

(2) 方法：

- get_queryset()：获取此视图的对象列表。必须是可迭代或者可以使查询集。默认返回 queryset 属性。（get_queryset、获取对象列表）
- get_context_data(**kwargs)：返回显示对象的上下文数据，通过覆盖该方法返回额外的上下文。（get_context_data、上下文数据）

➢ 与 DatailView 的 get()函数类似，ListView 的 get()首先调用 get_queryset()获取 queryset，然后将返回结果送到 get_context_data()获取上下文数据。重写这两个函数可以实现更多功能。

18. Django 出于安全方面的考虑，任何的 HTML 代码在 Django 的模板中都会被转义（即显示原始的 HTML 代码，而不是经浏览器渲染后的格式）。为了解除转义，只需在模板标签使用 safe 过滤器即可，例如{{ post.body|safe }}。

19. *args 和 **kwargs 是 python 中的两个自带变量，用于支持可变参数。只有前面的星号是必须的，也可以写成*var 和**kvar。其中*args 表示任何多个无名参数，它是一个 tuple；而**kwargs 表示关键字参数，它是一个 dict。（*args、无名参数、tuple、**kwargs、关键字参数、dict）

```
def foo(*args, **kwargs):
    print 'args = ', args
    print 'kwargs = ', kwargs
    print '----------------------------------------'

if __name__ == '__main__':
    foo(1,2,3,4)
    foo(a=1,b=2,c=3)
    foo(1,2,3,4, a=1,b=2,c=3)
    foo('a', 1, None, a=1, b='2', c=3)
```

输出结果如下：

```
args =  (1, 2, 3, 4)
kwargs =  {}
---------------------------------------
args =  ()
kwargs =  {'a': 1, 'c': 3, 'b': 2}
---------------------------------------
args =  (1, 2, 3, 4)
kwargs =  {'a': 1, 'c': 3, 'b': 2}
---------------------------------------
args =  ('a', 1, None)
kwargs =  {'a': 1, 'c': 3, 'b': '2'}
---------------------------------------
```

20. django 中有时候需要使用 save(commit=False)方法，更新对象属性，但并不向数据库真正提交数据，举个博客的例子吧，登录之后，写博客，然后保存博客的内容。commit=False 表示生成实例模型，但暂时不保存到数据库。如果需要保存到数据库中，则再次调用 save()：（commit=False、更新但不提交、保存到数据库、再调用 save）

```
def post_comment(request, post_pk):
    post = get_object_or_404(Post, pk=post_pk)
    if request.method == 'POST':
        form = CommentForm(request.POST)
        if form.is_valid():
            comment = form.save(commit=False)
            comment.post = post
            comment.save()
            return redirect(post)

        else:
            comment_list = post.comment_set.all()
            context = {'post': post,
                'form': form,
                'comment_list': comment_list
                }
            return render(request, 'blog/detail.html', context=context)
    return redirect(post)
```

21. 用户提交的数据存在 request.POST 中，这是一个类字典对象。（request.POST、用户提交的数据）

22. django 的中间件是嵌入 django 的 request/response 处理过程的一套钩子框架。它是一个轻量级的底层嵌入系统，可以对 django 的输入输出做整体的修改。 Django 中间件必须是一个类，并提供四个接口：（中间件、类、提供四个接口）

- process_request(self, request)：该方法在请求到来的时候调用。（process_request、请求到来）
- process_view(self ,request, fnc , arg ,kwarg)：在本次将要执行的 View 函数被调用前调用本函数。（process_view、视图被调用前）
- process_response(self,request,response)：在执行完 View 函数准备将响应发到客户端前被执行。（process_response、响应请求前）
- **process_exception(self,request, exception)：**View 函数在抛出异常时该函数被调用，

得到的 exception 参数是实际上抛出的异常实例。通过此方法可以进行很好的错误控制，提供友好的用户界面。

23. 要激活中间件，需要把它添加到 Django 配置文件 settings.py 中的 MIDDLEWARE 中。步骤如下：（激活中间件、setting、MIDDLEWARE）

(1) app 下新建 middleware.py 文件

(2) middleware.py 中实现自定义中间键类

(3) settings.py 中激活自定义中间件

例如，需要检测浏览器版本信息：

```
from django.http import HttpResponse
from django.shortcuts import render

try:
    from django.utils.deprecation import MiddlewareMixin  # Django 1.10.x
except ImportError:
    MiddlewareMixin = object  # Django 1.4.x - Django 1.9.x

class CheckVersionsMiddleware(MiddlewareMixin):
    def process_request(self, request):
        if 'rv:11.0' in request.META["HTTP_USER_AGENT"]:
            return render(request,"xxx.html")
            # return render(request,"upgrade.html")
        return None
```

定义完后，在 MIDDLEWARE 中添加中间件，‘app_name.middleware.CheckVersionsMiddleware’。

24. 上例中的 Request.META 是一个包含所有的 HTTP 头的字典。（Request.META、HTTP 头、字典）

25. 使用 middleware 时应该记住的东西

- middlewares 的顺序非常重要
- 一个 middleware 只需要继承 object 类 （只需继承、object 类）
- 一个 middleware 可以实现一些方法并且不需要实现所有的方法

26. 在 python 里，in 可以判断一个字符串是否在字典里：（in、是否在字典里）

```
z = {'a': 1, 'b': 2, 'c':3}
if 'a' in z:
    ……
```

27. Django 的 settings 文件包含 Django 应用的所有配置项。（settings、所有配置项）

```
from django.conf import settings

settings.DEBUG = False
```

28. 通过子类化 models.Manager 可以增强管理器的功能，例如，下列代码为管理器增加了 recently()和 published()两个函数：（子类化 models.Manager、增强功能）

```
class TutorialManager(models.Manager):
```

```
    def recently(self, num=40):
        return self.published().exclude(column__slug='aboutus'
            ).order_by('-update_time')[0:num]

    def published(self):
        return self.get_queryset().filter(published=True,
            pub_date__lt=timezone.now())

class Tutorial(models.Model):
        objects = TutorialManager()     #重新命名管理器
```

29. 在 setting 文件中使用 CACHE 来设置缓存：（setting 文件、CACHE、设置缓存）

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'TIMEOUT': 3600,
    }
}
```

- BACKEND：设置缓存的种类。其内置的有基于文件的缓存、基于数据库的缓存等等。（BACKEND、种类）
- LOCATION：缓存的位置。（LOCATION、位置）
- TIMEOUT：缓存超时时间。（TIMEOUT、超时时间）
- KEY_PREFIX：Django 服务器使用的所有缓存键将被自动包含（默认预置）的字符串。用于识别不同的 Django 实例的缓存。（KEY_PREFIX、识别不同的缓存）

30. 缓存设置完成后，使用缓存的最简单方法就是缓存整个网站。'django.middleware.cache.UpdateCacheMiddleware'和'django.middleware.cache.FetchFromCacheMiddleware'添加到 MIDDLEWARE 设置来完成。

```
MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

注意 UpdateCacheMiddleware 中间件和 FetchFromCacheMiddleware 中间件的顺序。

31. cache_page 装饰器可以用于缓存单个视图。（cache_page 装饰器、缓存、单个视图）

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

32. 上面这种 cache_page 用法将视图耦合到缓存系统，如果该视图某个请求不缓存，则该函数不可用。解决办法是在 URLconf 中引用视图函数：（解耦、URLconf、引用视图）

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

```
urlpatterns = [
    path('foo/<int:code>/', cache_page(60 * 15)(my_view)),
```

注意 ache_page 的用法。

33. 使用 cache 模板标签模板可以缓存模板片段。（cache 模板标签、缓存模板片段）

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

其中 500 是缓存时间，sidebar 是缓存片段的名称

34. urlpatterns 在 url 文件中是一个 url 映射列表，Django1.8 以后的版本中可直接为列表形式或者也可以用 patterns 函数生成。（url 映射、直接列表形式、patterns 函数生成）

- 列表形式：

```
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),            #使用 as_view()
将 IndexView 类视图转换成函数。这里使用了类视图。
    url(r'^post/(?P<pk>[0-9]+)/$', views.PostDetailView.as_view(),
name='detail'),
    url(r'^archives/(?P<year>[0-9]{4})/(?P<month>[0-9]{1,2})/$',
views.ArchivesView.as_view(), name='archives'),    #博客归档
    url(r'^category/(?P<pk>[0-9]+)/$', views.CategoryView.as_view(),
name='category'),          #博客分类
    url(r'^tag/(?P<pk>[0-9]+)/$', views.TagView.as_view(), name='tag'),
    # url(r'^search/$', views.search, name='search'),
]
```

- patterns 函数生成：

```
urlpatterns = patterns('tiy.views',
    url(r'^$', 'tutorial.views.index', name='index'),
    url(r'^j/(?P<token>.+)/$', 'iclick.views.click_count', name='click-count'),
    url(r'^robots\.txt$', TemplateView.as_view(
        template_name='robots.txt', content_type='text/plain'), name='robots'),
    url(r'^error_tutorial\.py$', 'tutorial.views.error_tutorial', name='error-tutorial'),
)
```

其中第一个参数 tiy.views 是指视图函数所在的文件，这里指 tiy 目录下的 views.py 文件。如果第一个参数为空，那么在写 url 列表引用处理函数时则需要 import 引入，传入的是函数名：（第一个参数、视图文件）

```
from myapp.views import *

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

35. urlpatterns 列表支持+= 运算的，这样可以分块完成 urlpatterns，结构更好辨认。（urlpatterns、支持+=）

```
urlpatterns += patterns('tiy.views',
    url(r'^try/color\.py$', 'try_color', name='try-color'),
)
```

36. F()对象允许 Django 在未实际链接数据库的情况下具有对数据库字段的值的引用。（F()、数据库字段、引用）

```
from django.db.models import F

reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed = F('stories_filed') + 1
reporter.save()
```

上面的意思是从 Reporters 数据库中取出 stories_filed 字段的值，加 1 后保存。

F()函数同样可以使用在查询集中，配合 update()方法

```
reporter = Reporters.objects.filter(name='Tintin')
reporter.update(stories_filed=F('stories_filed') + 1)
```

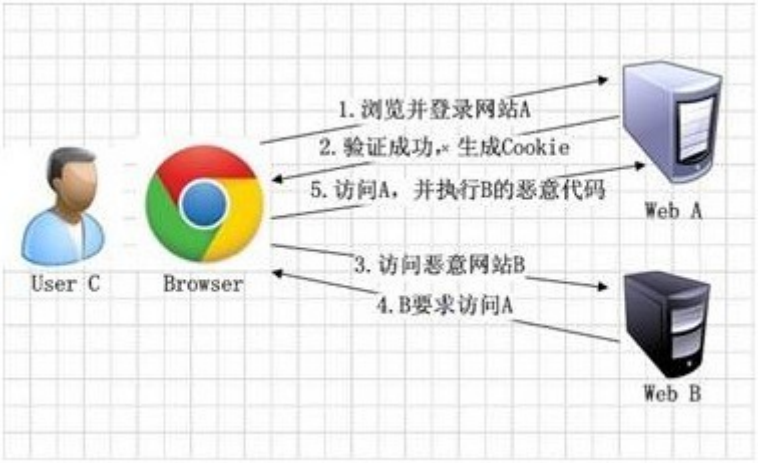
37. 如果在查询的时候需要组合条件，可以使用 Q 查询对象：（查询、组合条件、Q 查询对象）

```
from django.db.models import Q
q = Q(question__startswith = 'Who') | Q(question__startswith = 'What')
```

在 Q 对象的前面使用 ~ 来代表“非”：

```
Q(question__startswith = 'Who') | ~Q(pub_date__year = 2005)
```

38. csrf 攻击，即跨站请求伪造，攻击者盗用了你的网站身份，以你的名义发送恶意请求，攻击者即可得到你相关隐私信息。过程如下：（csrf 攻击、盗用身份）



39. 在 django 防御 csrf 攻击：在客户端页面上添加 csrftoken，服务器端进行验证，服务器端验证的工作通过'django.middleware.csrf.CsrfViewMiddleware'这个中间层来完成。（防御 csrf、csrftoken、验证）

40. GET 请求不需要 CSRF 认证，POST 请求需要正确认证才能得到正确的返回结果。一般在 POST 表单中加入{% csrf_token %}：（POST 请求、csrf_token 模板标签）

```
<form method="POST" action="/post-url/">
    {% csrf_token %}
```

```
    <input name='zqxt' value="自强学堂学习 Django 技术">
</form>
```

如果使用 Ajax 调用的时候，就要麻烦一些。

41. 可以通过 csrf_exempt 装饰器来取消 csrftoken 验证，方式有两种。（csrf_exempt 装饰器、取消 csrftoken 验证）

(1) 在视图函数当中添加 csrf_exempt 装饰器：

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def post_data(request):
    pass
```

(2) 在 urlconf 中：

```
from django.views.decorators.csrf import csrf_exempt
urlpatterns = [
   url(r'^post/get_data/$', csrf_exempt(post_data), name='post_data'),

]
```

42. 当我们想在多个页面都显示同样的内容的时候，例如在每个页面显示当前 ip，可以使用上下文渲染器。上下文渲染器事实上就是个函数，返回值是一个字典。但是该函数要在 setting 文件的 TEMPLATES 里注册。定义上下文渲染器有三步：（上下文渲染器、函数、返回字典）

(1) 在 setting 的同级目录下创建 context_processors.py 文件：（创建 context_processors.py 文件、添加到 setting、使用）

```
from django.conf import settings as original_settings

def settings(request):
   return {'settings': original_settings}

def ip_address_processor(request):
   return {'ip_address': request.META['REMOTE_ADDR']}
```

(2) 添加到 setting 中：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'accouting_project.context_processors.settings',  # add
                'accouting_project.context_processors.ip_address_processor'  # add
```

```
            ],
        },
    },
]
```

注意，accouting_project.context_processors.ip_address_processor 中的几项内容分别是项目名、文件名和函数名。（项目名、文件名、函数名）

(3) 使用：

```
<h1>Blog Home Page</h1>

DEBUG: {{ settings.DEBUG }}

ip: {{ ip_address }}
```

43. Django 中自带了 sitemap 框架，用来生成 xml 文件。步骤如下：（自带 sitemap 框架）

(1) 开启 sitemap 功能：（开启、定义 Sitemap 子类、更新 urls.py）

- 在 INSTALLED_APPS 设置中添加'django.contrib.sitemaps' 。
- 确认 settings.py 中的 TEMPLATES 设置包含 DjangoTemplates 后端，并将 APP_DIRS 选项设置为 True。APP_DIRS 选项默认配置就是 True，只有当你曾经修改过这些设置，才需要调整过来。
- 确认你已经安装 sites 框架。即在 INSTALLED_APPS 中添加'django.contrib.sites'，并添加设置变量 SITE_ID=1。

(2) 定义 Sitemap 子类：

```
from django.contrib.sitemaps import Sitemap
from myweb.models import News
from django.urls import reverse

class NewsSitemap(Sitemap):
    changefreq = 'daily'
    priority = 0.6

    def  items(self):
        return News.objects.all()

    def lastmod(self, obj):
        return obj.pub_time

    def location(self, obj):
        return reverse('new', kwargs={'new_id': obj.id})
```

sitemap 类和 GenericSitemap 类可以直接在 urlpatterns 中进行子类化。（sitemap 类和 GenericSitemap 类、urlpatterns 中、子类化）

(3) 更新 urls.py

```
from django.contrib.sitemaps import sitemap    # 导入 sitemap 视图
from xxx.sitemaps import NewsSitemap

sitemaps = {
    'new': NewsSitemap,
}
```

```
urlpatterns = [
        ...
        url(r'^sitemap\.xml$', sitemap, {'sitemap': sitemaps},
name='django.contrib.sitemaps.views.sitemap'),
]
```

其中 sitemaps 是一个字典，保存着所有需要的栏目的页面的地图，如上面的 new 就是栏目。NewsSitemap 是 sitemap 子类。下面的代码更能显示 sitemaps 变量：（sitemaps、字典、栏目、子类）

```
class LimitGenericSitemap(GenericSitemap):
    limit = 200


sitemaps = {
    'tutorials': LimitGenericSitemap(
            {'queryset': Tutorial.objects.all(), 'date_field': 'update_time'},
            priority=1.0),
    'tiy': LimitGenericSitemap(
            {'queryset': Instance.objects.all(), 'date_field': 'update_time'},
            priority=0.5),
}
```

44. 有时候，我们不希望在站点地图中出现一些静态页面，比如商品的详细信息页面，这个可以使用 reverse()解决。（不出现、location 中、调用 reverse()）

```
from django.contrib.sitemaps import Sitemap
from django.urls import reverse

class StaticViewSitmap(Sitemap):
    priority = 0.7
    changefreq = 'daily'

    def items(self):
        return ['main' , 'about' , 'license']

    def location(self, obj):
        return reverse(item)
```

在网站地图的 location 方法中调用 reverse()。

45. Sitemap 类

(1) items()：必须定义。返回对象列表。这些列表将被传递给 location()，lastmod()，changefreq()和 priority()方法。

(2) location：可选，其值可以是一个方法，也可以是属性。如果是一个方法，它应该为 items()返回的对象的绝对路径。如果它是一个属性，它的值应该是一个字符串，表示 items()返回的每个对象的绝对路径。

(3) lastmod：可选。 一个方法或属性。表示当前条目最后的修改时间。

(4) changefreq：可选。 一个方法或属性。表示当前条目修改的频率。其值包括'always'、'hourly'、'daily'等。

(5) priority：可选。表示当前条目在网站中的权重系数，优先级。页面的默认优先级为 0.5，最高为 1.0。

(6) protocol：可选的。定义网站地图中的网址的协议（'http'或'https'）。

(7) limit：可选的。定义网站地图的每个网页上包含的最大超级链接数。

46. 和 Sitemap 不同的是，GenericSitemap 类可以包括多个 queryset 条目：（GenericSitemap、多个 queryset 条目）

```
class LimitGenericSitemap(GenericSitemap):
   limit = 200

sitemaps = {
   'tutorials': LimitGenericSitemap(
         {'queryset': Tutorial.objects.all(), 'date_field': 'update_time'},
         priority=1.0),
   'tiy': LimitGenericSitemap(
         {'queryset': Instance.objects.all(), 'date_field': 'update_time'},
         priority=0.5),
}
```

47. Django2.0 的 urlpatterns 发生了变化，从使用 url 函数变为 path 函数。

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>/', views.year_archive),
    path('articles/<int:year>/<int:month>/', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>/', views.article_detail),
]
```

- 要捕获一段 url 中的值，需要使用尖括号，而不是之前的圆括号；（捕获、尖括号）
- 可以捕获指定类型的值，并保存为指定变量，比如例子中的捕获 int 型值，并保存为 year 变量。其中的 int 是 path 路径转换器。（捕获指定类型、保存为变量、int、转换器）

48. Django 内置下面的 path 路径转换器：（转换器、str、int、slug、uuid、path）

- str：匹配任何非空字符串，但不含斜杠/，如果你没有专门指定转换器，那么这个是默认使用的；
- int：匹配 0 和正整数，返回一个 int 类型
- slug：可理解为注释、后缀、附属等概念，是 url 拖在最后的一部分解释性字符。该转换器匹配任何 ASCII 字符以及连接符和下划线，比如’ building-your-1st-django-site‘；
- uuid：匹配一个 uuid 格式的对象。为了防止冲突，规定必须使用破折号，所有字母必须小写，例如’075194d3-6885-417e-a8a8-6c931e272f00‘ 。返回一个 UUID 对象；
- path：匹配任何非空字符串，重点是可以包含路径分隔符’/‘。这个转换器可以帮助你匹配整个 url 而不是一段一段的 url 字符串。

49. 自定义 path 转换器，其实就是写一个类，并包含下面的成员和属性：（自定义转换器、类）

- regex：一个字符串形式的正则表达式属性；
- to_python(self, value)：一个用来将匹配到的字符串转换为你想要的那个数据类型，并传递给视图函数。如果转换失败，它必须弹出 ValueError 异常；

- to_url(self, value)：将 Python 数据类型转换为一段 url 的方法，上面方法的反向操作。

举例步骤如下：

(1) 新建一个 converters.py 文件，与 urlconf 同目录，写个下面的类：（converters.py 文件、register_converter 注册）

```
class FourDigitYearConverter:
    regex = '[0-9]{4}'

    def to_python(self, value):
        return int(value)

    def to_url(self, value):
        return '%04d' % value
```

(2) 在 URLconf 中注册，并使用它，如下所示，注册了一个 yyyy：

```
from django.urls import register_converter, path

from . import converters, views

register_converter(converters.FourDigitYearConverter, 'yyyy')

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<yyyy:year>/', views.year_archive),
    ...
]
```

50. Django2.0 的 url 使用 re_path()方法代替 path()方法来兼容之前的 url()。re_path()方法在骨子里，根本就是以前的 url()方法，只不过导入的位置变了。（re_path()兼容 url()）

```
from django.urls import path, re_path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    re_path(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$',
views.article_detail),
]
```

51. 在 Python 的正则表达式中，命名组的语法是(?P<name>pattern)，其中 name 是组的名称，pattern 是要匹配的模式。（name、组名）

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
```

```
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$',
views.article_detail),
]
```

52. URL 中捕获的参数为字符串类型，每个捕获的参数都作为一个普通的 Python 字符串传递给视图，即便被捕获的‘100’看起来像个整数，但实际上是个字符串‘100’。 例如，下面这行 URLconf 中：（捕获的参数、字符串）

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
```

传递给 views.year_archive()的 year 参数将是一个字符串，不是整数，即使[0-9]{4}只匹配整数字符串。

53. 错误视图包括 400、403、404 和 500，分别表示请求错误、拒绝服务、页面不存在和服务器错误。自定义错误页面：（自定义错误页面）

(1) 在根 URLconf 中额外增加下面的条目：（增加条目、增加视图、页面文件）

```
# URLconf
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>[0-9]+)/$', views.page),
]

# 增加的条目
handler400 = views.bad_request
handler403 = views.permission_denied
handler404 = views.page_not_found
handler500 = views.page_error
```

- handler400 等等内容是 django 内容的关键字。
- views.bad_request 是自定义的视图函数。

(2) 然后在，views.py 文件中增加四个处理视图：

```
def page_not_found(request):
    return render(request, '404.html')


def page_error(request):
    return render(request, '500.html')


def permission_denied(request):
    return render(request, '403.html')


def bad_request(request):
    return render(request, '400.html')
```

(3) 根据自己的需求，创建 404.html、400.html 等四个页面文件。

54. 路由文件也就是 urls.py 文件，使用 include 进行路由转发。注意转发时已经匹配过的内容无须再匹配。例如：（匹配过、无须再匹配）

```
from django.urls import path,include
from django.contrib import admin

urlpatterns = [
    path(r'^app2/', include('app2.urls')), # 映射到下级路由
```

以 xxx.com/app2/post 为例，在上面的路由匹配后，子路由不需要再匹配 app2，而是直接匹配 post：

```
url(r'^post/, views.detail)
```

55. 用户登陆等提交表单的页面通常需要在提交表单之后重定向到一个指定的页面，这就需要使用 django 的重定向功能。重定向的几种方式：（重定向、HttpResponseRedirect、redirect）

- 使用 HttpResponseRedirect。
- redirect 和 reverse

56. reverse()用于根据指定的视图函数获取 url。通常和重定向配合使用，用于重定向到指定的页面。在重定向中使用 reverse()而不直接写 url 的原因是，修改视图的 url 时不需要在视图函数中，而只需要的路由中即可。（reverse()、根据视图、获取 url）

57. get_object_or_404()：常用于查询某个对象，找到了则进行下一步处理，如果未找到则给用户返回 404 页面。

58. get_list_or_404()：这其实就是 get_object_or_404 多值获取版本。在后台，返回一个给定模型管理器上 filter()的结果，并将结果映射为一个列表，如果结果为空则弹出 Http404 异常。

59. 由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来标识具体的用户，这个机制就是 Session。典型的场景比如购物车，当你点击下单按钮时，由于 HTTP 协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的 Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。（标识用户、session）

60. Session 是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie 是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现 Session 的一种方式。

61. Django session 通过中间件来实现，要启用会话，需要将 django.contrib.sessions.middleware.SessionMiddleware 添加到 setting 文件的 MIDDLEWARE 中。事实上，django 默认就开启了会话。（session、中间件实现）

62. Django 默认情况下会将会话存储在数据库中。将 django.contrib.sessions 添加到 INSTALLED_APPS 中。（session、默认、数据库）

63. HttpResponse.set_cookie()可以用于在服务器中设置指定 cookie 的值而，HttpRequest.COOKIES 是一个字典，里面保存着各种 cookie 的值。（set_cookie、设置 cookie、COOKIES、字典）

- HttpResponse.set_cookie()：

```
from django.template import loader ,Context
from django.http import HttpResponse
```

```
def main(request):
    #不用模板
    response= HttpResponse('test')
    response.set_cookie('my_cookie','cookie value')
    return response
```

- HttpRequest.COOKIES：

```
def my_get_cookie(request):
    ...
    request.COOKIES["cookie"])
    ...
```

64. request.session 对象可以用于设置 session：（request.session、设置 session）

```
def login ( request ):
    m = Member . objects . get ( username = request . POST [ 'username' ])
    if m . password == request . POST [ 'password' ]:
        request . session [ 'member_id' ] = m . id
        return HttpResponse ( "You're logged in." )
    else :
        return HttpResponse ( "Your username and password didn't match." )

def logout ( request ):
    try :
        del request . session [ 'member_id' ]
    except KeyError :
        pass
    return HttpResponse ( "You're logged out." )
```

65. Django 使用 Python 内置的 logging 模块实现它自己的日志系统。

66. 用户对象是 Django 认证系统的核心！在 Django 的认证框架中只有一个用户模型也就是 User 模型，它位于 django.contrib.auth.models。

67. 使用 User 对象的 has_perm()函数判断对象是否具有某种权限，有则返回 true。（has_perm、是否具有、权限）

```
def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    user.has_perm('myapp.change_blogpost')
```

68. django 自带一套信号机制来帮助我们在框架的不同位置之间传递信息。Django 内置了一整套信号，下面是一些比较常用的：（信号机制、内置信号）

- django.db.models.signals.pre_save 和 django.db.models.signals.post_save

  在 ORM 模型的 save()方法调用之前或之后发送信号。

- django.db.models.signals.pre_delete 和 django.db.models.signals.post_delete

  在 ORM 模型或查询集的 delete()方法调用之前或之后发送信号。

- django.db.models.signals.m2m_changed
  当多对多字段被修改时发送信号。
- django.core.signals.request_started 和 django.core.signals.request_finished
  当接收和关闭 HTTP 请求时发送信号。

➢ 注意，ORM 模型即对象关系映射，也就是 model 的数据库的内容。

69. 要接收信号，请使用 Signal.connect()方法注册一个接收器。当信号发送后，会调用这个接收器。（Signal.connect()、注册接收器）

```
Signal.connect(receiver, sender=None, weak=True, dispatch_uid=None)[source]
```

70. 监听信号的步骤如下：

(1) 编写接收器，接收器其实就是一个 Python 函数或者方法：

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

(2) 连接接收器。有两种方法可以连接接收器，一种是使用 Signal.connect()的手动方式：

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

另一种是使用 receiver()装饰器：

```
from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

(3) 接收特定发送者的信号。一个信号接收器，通常不需要接收所有的信号，只需要接收特定发送者发来的信号，所以需要在 sender 参数中，指定发送方。

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel


@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

(4) 防止重复信号为了防止重复信号，可以设置 dispatch_uid 参数来标识你的接收器，标识符通常是一个字符串，如下所示：

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

71. 所有的信号都是 django.dispatch.Signal 的实例。使用类似下列方式定义新信号：

（Signal 的实例 、定义新信号）

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

上面的例子定义了 pizza_done 信号，它向接受者提供 size 和 toppings 参数。

72. Django 中使用 Signal.send 和 Signal.send_robust 两种方法用于发送信号。（send 和 send_robust、发送信号）

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
        ...
```

73. 使用 Signal.disconnect()来断开信号的接收器。（disconnect、断开信号）
74. Django 提供了基于 Cookie 或者会话的消息框架 messages，无论是匿名用户还是认证的用户都可使用。这个消息框架允许你临时将消息存储在请求中，并在接下来的请求（通常就是下一个请求）中提取它们并显示。（消息框架、存储在请求中）
75. 通过 django-admin startproject xxx 命令创建工程时，已经默认在 settings.py 中开启了消息框架功能需要的所有的设置。
76. 消息框架的级别是可配置的，与 Python 的 logging 模块类似 Django 内置的 message 级别有下面几种：

- DEBUG：将在生产部署中忽略（或删除）的与开发相关的消息
- INFO：普通提示信息
- SUCCESS：成功信息
- WARNING：警告信息
- ERROR：已经发生的错误信息

77. 有两种方式可以添加消息：（添加消息、add_message()、快捷方式）

- 使用 add_message()：

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

- 使用快捷方式：

```
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

78. 使用 get_messages()来获取消息。（get_messages()、获取消息）

```
storage = get_messages(request)
```

79. 消息级别只是一个整数常量，所以，可以定义消息级别，例如：（add_message、自定义消息级别）

```
CRITICAL = 50

def my_view(request):
    messages.add_message(request, CRITICAL, 'A serious error occurred.')
```

自定义消息级别时，应小心避免覆盖现有级别。内置的消息级别为：

- DEBUG：10
- INFO：20
- SUCCESS：25
- WARNING：30
- ERROR：40

80. 不同的消息级别在前端显示不同的样式，默认为消息级别的小写，如 DEBUG 的前端样式为 debug 类。使用 MESSAGE_TAGS 可以修改前端样式：（前端样式、消息级别的小写、MESSAGE_TAGS、修改）

```
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.INFO: 'myinfo',
}
```

将 INFO 的样式修改为 myinfo。

81. 每个请求都可以通过 set_level()方法设置最小记录级别：（set_level()、最小级别）

```
from django.contrib import messages

# 修改最小级别为 DEBUG
messages.set_level(request, messages.DEBUG)

# 在另外一个视图中修改最小级别为 WARNING
messages.set_level(request, messages.WARNING)
messages.success(request, 'Your profile was updated.') # 被忽略，不记录
```

类似的，当前有效的记录级别可以用 get_level()方法获取。

82. 默认情况下，如果包含消息的迭代器完成迭代后，当前请求中的消息都将被删除。如果你不想这么做，想保留这些消息，那么需要显式的指定 used 参数为 False，如下所示：（迭代后、删除、要保留、指定 used）

```
storage = messages.get_messages(request)
for message in storage:
    do_something_with(message)
storage.used = False
```

83. 要为消息添加自定义的消息 CSS 样式，可以通过 extra_tags 参数：（添加 css、extra_tags）

```
messages.add_message(request, messages.INFO, 'Over 9000!', extra_tags='dragonball')
messages.error(request, 'Email box full', extra_tags='email')
```

84. 在 css 中渲染不同类型 messages 如下：

```
.messages {
    /* Your code */
}
.messages .success {
    /* Your code, e.g. background: green */
}
.messages .error {
    /* Your code, e.g. background: black */
}
.messages .warning {
    /* Your code, e.g. background: red */
}
```