Django 讲解

Beyking@gmail.com

目　录

1 使用入门

1.1 Sys Environ

首先确保您的电脑已经安装了 Python，其次就是确保已经安装了 Django，Django 可以在官方网站 http:// www.djangoproject.com/download 下载，不过我推荐最好使用 Subversion 软件下载最新的 django，下载地址：http://code.djangoproject.com/svn/django/trunk/。

然后请设置一下环境变量 PYTHONPATH，让该变量值为您自己希望自行放置 Python 软件包的目录。

例如，如果是 Linux 操作系统的话我们可以编辑 /etc/environment 文件，增加一句:

PYTHONPATH=”/home/beyking/python”。

最后，我们需要把解压或者使用 Subversion 下载的 django 当中的 django 目录复制到 PYTHONPATH 的目录下面。

当然，上面只是推荐的 Django 安装方式，您也可以直接把 django 目录放在 Python 安装目录下面的 Lib/site-packages/下面，或者是 /usr/local/lib/python/site-packages/下面.

1.2 Start Project

我们使用 django 的第一步可能就是想创建一个 django 的工程看看，通过 django 的帮助，我们知道需要执行 django-admin.py startproject mysite 这样的命令。

OK，我们进入命令行且当前目录为 PYTHONPATH 的目录，执行如下命令：

```
beyking@python:~/python $ python django/bin/django-admin.py hello
Unknown command: 'startproject'
Type 'django-admin.py help' for usage.
```

我们居然发现出错了，不知道的命令'startproject'，这到底是怎么回事呢？原来我的环境变量当中已经存在有 DJANGO_SETTINGS_MODULE 了，通过查看源码我们看到，

Django 如果发现您已经已经有这个设置之后，会把这 startproject 这个子命令删除不让用的。

可能 django 觉得，您现在当前的正在做 DJANGO_SETTINGS_MODULE 这个环境变量设置的 project，所以还是专心做这个吧，不要创建新的 project 了;但是我们不使用环境变量的话，又需要每次运行之前首先要手动的指定需要使用那个 settings（调用 django.conf.setting.configure(self, default_settings=global_settings, **options))也够麻烦的话。

我觉得这是不太合理的，无论如何应该能让我们创建新的工程，如果我想使用新创建的这个工程，我可以创建完之后再去修改 DJANGO_SETTINGS_MODULE 为新创建的工程的 settgins 都不迟，所以我们直接把 django/core/management/__init__.py 文件当中 get_commands 函数中的#del _commands['startproject']这一行注释掉，然后我们再执行 python django/bin/django-admin.py hello 就一切 OK 了。

我们进去 hello 目录看到，只有 4 个文件，其中__init__.py 文件一个字节没有.

manage.py 文件基本就是 django-admin.py 的复制，只是加多了检查当前目录 settings.py 文件是否存在，不存在就退出。

setting.py 文件当中就是把 django/conf/global_settings.py 文件当中的，django 觉得有必要让您亲自修改的一些设置放在了当前工程的 settings.py 文件当中，当然我们也可以自己动手把更多 global_settings.py 的当中的一些设置放在 settings.py 当中，然后修改为其他值，最终 django.conf.settings 模块会合并当前工程的 settings 和 global_settings 的所有设置，如果有相同的设置名称，会以前工程的 settings 的为准，我们也可以随意添加自己的一些设置，但是需要注意：所有的设置名称的字母都必须全部大写，django 只认大写的设置，不然就视而不见。

urls.py 文件是是负责 url 的解析分配的，熟悉 cherrypy 的同仁都很清楚，在 cherrypy 当中每个页面都是一个 class 的对象，每个页面下一级的 url 的页面也是一个 class 对象，同时又都是上一级 class 的对象的一个属性，从而用 class 构件了一棵 URL 的树。

在 django 当中采用了不同的处理方式，她使用了正则表达式的方式来匹配 url，从而决定应该调用哪个页面显示函数来显示，函数参数是什么等等。我个人觉得 djagno 的方式更加合理和灵活。

1.3 Show View

接下来，演示一下如何显示一个页面，首先我们创建一个 views.py 在当前的工程目录，然后输入如下文字：

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello Django")
```

其中 index 函数是用来显示一个页面的，这个名字是随便命名的，index 不代表什么特殊含义，这个和 cherrypy 不同;

该类函数至少有一个参数，且第一个参数必须固定是 django/http/__init__.py 当中定义的 HttpRequest 这个 class 的子类，至于是哪个子类就不一定了，这个我们以后章节再讲。

该类函数必须返回一个 HttpResponse 类型的结果。

然后我们在修改一下 urls.py 的文件，增加一行 (r'^$', 'views.index')，其中 r'^$'这个正则表达式就是表示匹配一个空字符串，多一个空格都不行，因为一开始^就结束$嘛！

然后后面的'views.index'表示调用 views 模块的 index 函数

最后我们运行这个程序，在运行之前不要忘记修改一下环境变量的 DJANGO_SETTINGS_MODULE 为 hello.settings, 命令行的当前目录为 hello 的工程目录，在命令行建入如下命令:

```
beyking@python:~/python/hello$ python manage.py runserver
Validating models...
0 errors found
Django version 0.97-pre-SVN-6783, using settings 'hello.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

在浏览器当中输入 http://127.0.0.1:8000/ 我们就可以看到 Hello Django 了。

以上是一个最简单的 Django 工程，简单到了无以复加的地步，只是为了让大家先感受一下 Django 而已。

1.4 Start App

实际的项目当中，我们都知道，一个项目可以分为不同的模块，例如：系统登陆、用户校验、留言板 等等。

在 Django 当中，这些不同的模块就是不同的 App 了，可以通过 startapp 这个命令来创建。不同 Django 的 App 关注不同的具体业务，且每个 App 都有很大程度的独立性，以后您做其他的 django 的 project 的时候，如果某个功能和现有的类似，完全可以 copy 过去，简单设置一下就可以了。

OK，下面我们创建一个 app, 命令如下:

```
beyking@python:~/python/hello$ python manage.py startapp world
```

我们发现创建了一个 world 目录，里面有 models.py 和 views.py, 其中 views.py 和我们刚才讲的 views.py 一样，主要用于页面显示或者说页面控制的，而 models.py 是用于写针对 django 自带的面向对象数据库的数据模型的，数据模型可以通过 syncdb 命令来产生相应的物理数据库表，我们在 models.py 输入如下代码：

```
from django.db import models


class World(models.Model):
    name = models.CharField(max_length=96)
    value = models.IntegerField(default=0)
```

以上代码创建了一个叫做 World 的 Django 的 ORM 的模型对象，具体 django 的 ORM 的使用方法请参考相关帮助，这里不详细讲述了.

然后我们修改一下 settings.py 文件当中的 DATABASE 开头的一下属性设置的值，例如，修改为如下：

```
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'hello'
DATABASE_USER = 'root'
DATABASE_PASSWORD = 'sa'
```

还要确保您的 mysql 当中存在 hello 这个数据库，如果不存在请先创建一个。

接下来我们在命令行里面输入命令:

```
beyking@python:~/python/hello$ python manage.py syncdb
Creating table auth_message
Creating table auth_group
Creating table auth_user
....
```

我们去 mysql 数据库查看一下，会发现创建了一些数据库表，但是就是找不到 world 字样的数据库表，为什么呢？

因为我们没有在 settings.py 的 INSTALLED_APPS 属性当中添加上我们自己的 app,只有在告诉了 settings 我们安装了哪些 app，哪个 app 才能工作，OK，我们只需要在 INSTALLED_APPS 加多一行'hello.world'就可以了，默认的 django 已经为我们添加了一些非常通用的 app。

这时我们再去 python manage.py syncdb 就可以添加上我们自己定义的模型的数据库表了。该数据库表名字为"world_world"，它的命名为 app 的名称 + '_' + 模型的名称。

为了下面的演示，我们先在 Python 的命令行里面输入异侠代码，创建一条关于 World 模型的数据库记录，代码如下：

```
>>> from hello.world.models import World
>>> o = World(name = 'test world', value = 100)
>>> o.save()
```

然后我们也在 views 里面写点什么，为了表明我们可以操作数据吧，代码如下：

```
from django.http import HttpResponse
```

```
from hello.world.models import World
def index(request):
    a = World.objects.all()[0]
    return HttpResponse(a.name)
```

为了能让这个 app 工作，我们还需要配置一下 urls.py，我们可以在当前工程目录的 urls.py 里面加上一句(r'^world/$', 'hello.world.views.index')，但是我觉得更好的方式应该是每个 app 本身应该带有自己的 urls.py 文件，所以我们添加 (r'^world/', include('hello.world.urls')),这句话, 相应的在 world 目录下面创建一个 urls.py 文件，内容如下:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('hello.world.views',
    (r'^$', 'index')
)
```

这里需要注意，我们可以看到工程目录的 urls.py 文件当中的 urlpatterns = patterns('', ...) 第一个参数是空格，而我们这里的 urls.py 的是'hello.world.views'，这个参数很重要，它决定了下面所有要解析的 url 调用的函数的所在的模块路径的前缀，所以上面的代码我们也可以这样写：

```
urlpatterns = patterns('hello.world',
    (r'^$', 'views.index')
)
```

假如您的 world 这个 app 有不同的 view 处理单元的话。

最后我们 beyking@python:~/python/hello$ python manage.py runserver 之后，打开浏览器输入输入 http://127.0.0.1:8000/world 我们就可以看到 test world 了

通过这一节，我们知道了如何创建和配置一个 app，我们的代码应该尽量安装不同业务逻辑划分到不同的 app 当中。

2 工作原理

2.1 Socket&Server

为了能继续下面的章节，我们在这里现简单介绍一下如何创建一个最简单的网络服务器和客户端, 代码如下：

```
import socket

host = '127.0.0.1'
port = 8000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))

while True:
    clientsocket, clientaddr = s.accept()
    clienfile = clientsocket.makefile('rw', 0)
    #do something with clienfile ...
    #clienfile.read()
    #clienfile.write(somestr)
    clientsocket.close()
    clientsocket.close()
```

接下来我们再创建一个最简单的客户端，代码如下:

```
import socket

host = '127.0.0.1'
port = 8000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
```

```
s.sendall('some commond \r\n')

while True:
    buf = s.recv(2048)
    if not buf: break
```

通过上面的代码我们看到,不管是客户端还是服务器都要创建 Socket 对象，而且参数相同，其中 AF_INET 表示为 IPv4 的 Internet 通信类型,SOCK_STREAM 表示使用 TCP 的协议类型。所不同的是，客户端 socket 是 connect 一对地址和接口，服务器端则需要 bind 一对地址和接口。剩下的就是客户端 send 或者 sendall 信息，然后服务器 accept 到以后客户的的请求后，可以 read 到客户端发送的信息，然后 write 信息给客户端，最后客户端就 recv 到了内容。

当然，实际的服务器和客户端需要处理的过程要复杂的多，但基本原理就是这样。

大家可以参考一下 Python 的 Lib 目录下面的 SocketServer.py 文件，该文件定义了服务器类的基类：BaseServer 以及 TCP 类型的服务器基类:TCPServer(从 BaseServer 继承)，BaseServer 的主要接口就是 serve_forever，handle_request，以及构造函数的初始化类 RequestHandlerClass。调用 serve_forever 就开始了 handle_request 的循环，然后在 handle_request 当中，通过调用 get_request()之后获得客户端的请求和地址之后，就把这个请求和地址交给了构造函数初始化的传入的 RequestHandlerClass 类来处理了。在 TCPServer 当中，和我们上面的代码类似，创建了 socket，并且在 get_request()当中就是返回的 socket.accept()得到的客户端的 socket 和地址。

同时该文件也定义了 RequestHandlerClass 的基类 BaseRequestHandler，该类除了定义了 setup()，handle()，finish()三个接口以及在初始化的时候调用之外，几乎没有做其他的事情。真正处客户端请求的代码，需要子类重载 handle 方法来实现。

其中 StreamRequestHandler 是从 BaseRequestHandler 继承的，该类只是把客户端的 socket 调用 makefile，和我们上面的代码类似，不过是按 rb 和 wb 参数 makefile 了两个而已。

2.2 Runserver

如何启动一个 Django 的应用有多种方式，例如 Mod_Python、FastCGI 以及自带的 WSGI 的方式，甚至可以自行扩展任何其他的方式。接下来我们以 Django 自带的 WSGI 的

方式来追踪一下，Django 启动过程的来龙去脉，其他的方式和 WSGI 都大同小异。

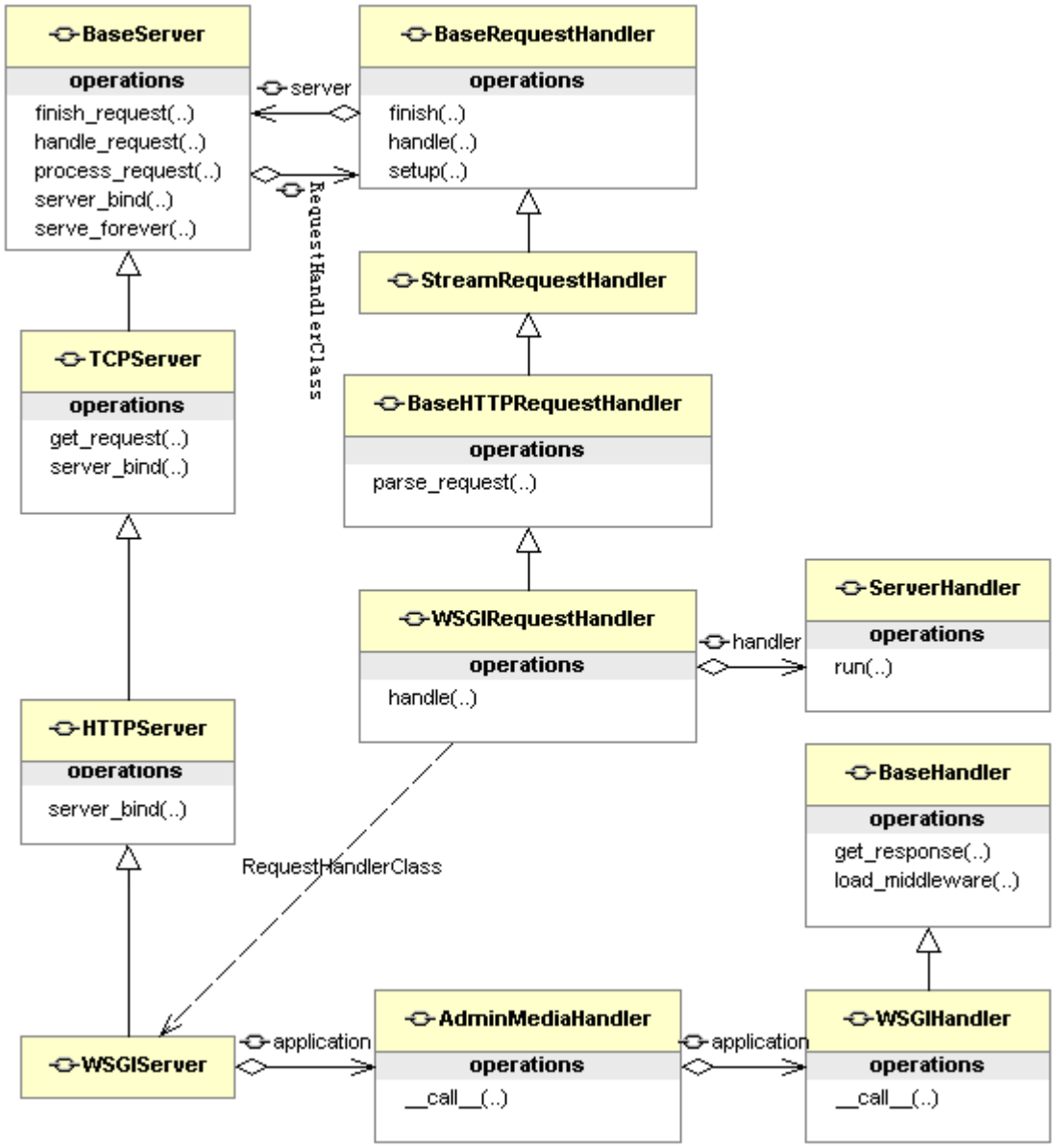
当我们在命令行运行 python manage.py runserver 的时候，会调用 django/core/management/commonds/runserver.py 文件的 inner_run 函数，该函数当中关键的几句如下：

```
path = admin_media_path or django.__path__[0] + '/contrib/admin/media'
handler = AdminMediaHandler(WSGIHandler(), path)
run(addr, int(port), handler)
```

其中最后一行的 run 函数代码如下(在 django/core/server/basehttp.py)：

```
def run(addr, port, wsgi_handler):
    server_address = (addr, port)
    httpd = WSGIServer(server_address, WSGIRequestHandler)
    httpd.set_app(wsgi_handler)
    httpd.serve_forever()
```

从 httpd.serve_forever()开始，整个程序算是启动起来了。启动过程中用到的主要 Class 及其继承和引用关系，以及主要的方法，如下图所示：



通过上图我们看到，WSGIServer 继承自 HTTPServer， 从 WSGIServer 的对象 httpd 的 serve_forever()调用，就开始了不断调用 handle_reqeust 函数的循环，在 handle_reqeust 当中主要就是通过 request, client_address = self.get_request() 得到了 requset 即 client socket 和 client address，然后把这两个变量当作参数传送给了 process_request 函数，而该函数又调用了 finish_request 函数，参数还是那两个，终于在该函数当中调用了 RequestHandlerClass 这个 class 来处理 client socket 和 client address。

在 http 创建的时候，指定的 RequestHandlerClass 是 WSGIRequestHandler 这个 class，所以最终会调用 WSGIRequestHandler 这个 class 来处理来自客户端的请求 。WSGIRequestHandler 的父类当中有 StreamRequestHandler，在上一节我们介绍过，在该类

把客户端的 socket 按参数 rb 和 wb 调用了 makefile，得到两个变量 rfile 和 wfile。

在 WSGIRequestHandler 重载的 handle 方法中，首先调用了一下父类 BaseHTTPRequestHandler 的 parse_request 的方法，用于得到是否是一个合法的 HTTP 请求，如果是一个合法的 HTTP 请求，顺便得到该请求的 command(POST or GET), path(url path), request_version(HTTP 协议的版本)等，如果不是合法的 HTTP 请求则则直接退出了；如果是合法的请求，则又把处理任务转嫁给了，ServerHandler 类的对象 handler 来处理，并且 ServerHandler 的初始化中 RequestHandlerClass 的 rfile 和 wfile 都是初始化参数，然后又调用 ServerHandler 的 run 函数，且把 WSGIServer 的 applicaton 当作参数传入，该 applicaton 即 AdminMediaHandler 的对象。

然后我们再来看看 ServerHandler 的 run 方法吧，经过千折百转之后，终于到了要真正处理我们来自客户端的请求的地方了，ServerHandler 的主要代码如下（代码是精简过后的，只留下了最为关心的部分）：

```
def __init__(self, stdin, stdout, stderr, environ, multithread=True, multiprocess=False):
    self.stdin = stdin
    self.stdout = stdout

def run(self, application):
    self.result = application(self.environ, self.start_response)
    self.finish_response()

def finish_response(self):
    for data in self.result:
        self.write(data)
    self.finish_content()
    self.close()
def write(self, data):
    self._write(data)
    self._flush()

def _write(self,data):
```

```
    self.stdout.write(data)

def _flush(self):
    self.stdout.flush()
```

在 WSGIRequestHandler 的 run 方法中，创建 ServerHandler 的对象的代码如下：

```
handler = ServerHandler(self.rfile, self.wfile, self.get_stderr(), self.get_environ())
```

所以 ServerHandler 中的 stdin 和 stdout 其实就是 client socket 的读和写的两个文件对象 rfile 和 wfile，当调用了 application 来处理了客户端的请求之后，会得到针对客户端请求的返回内容，然后通过调用 finish_response 函数把所有返回内容 write 到 client socket 的 wfile 当中，最终客户端就受到了来自服务器的发送的内容了。

看到这里我们才知道，原来 ServerHandler 主要是负责返回客户端需要的结果，而真正产生得到客户端需要的结果的另有其人，那就是 WSGIRequestHandle 当中的 run 方法中 handler.run(self.server.get_app()) 这句话中的 self.server.get_app()得到的那个 application，即 AdminMediaHandler 的对象。OK，接下来我们在来看看 AdminMediaHandler 的主要代码（代码是精简过后的，只留下了最为关心的部分）：

```
def __init__(self, application, media_dir):
    self.application = application
    self.media_dir = media_dir

def __call__(self, environ, start_response):
    import os.path

    if self.media_url.startswith('http://') or self.media_url.startswith('https://') \
            or not environ['PATH_INFO'].startswith(self.media_url):
            return self.application(environ, start_response)

    relative_url = environ['PATH_INFO'][len(self.media_url):]
    file_path = os.path.join(self.media_dir, relative_url)
    fp = open(file_path, 'rb')
```

```
        mime_type = mimetypes.guess_type(file_path)[0]
        if mime_type: headers['Content-Type'] = mime_type
        output = [fp.read()]
        fp.close()
        start_response(status, headers.items())
        return output
```

AdminMediaHandler 里面居然还有一个 application，这个 application 是 WSGIHandler 的对象。

在__call__方法当中我们看到，如果 media_url 是以 http://或 https://开头，或者 url path 不是以 media_url 开头的的话，则直接调用 WSGIHandler 来进行处理；

其中 environ['PATH_INFO'] 当中存放的是当前 url 的 path，该值是在 WSGIRequestHandler 类的时候，就被赋值了的，且一路传送至此，并且将继续传送下去，直至我们后面介绍到的 SWGIHandler 和 WSGIRequest。

如果 media_url 是以 http://或 https://开头则表示这是一个文件服务器，专门处理处理文件的请求；如果不是，且 url path 也不是以 media_url 开头则表示普通的 HTTP 请求，自然也是通过 WSGIHandler 来进行处理，最后如果 url path 以 media_url 开头，则表示想取得本服务器上的文件，所以执行__call__下面的代码，把要取得的文件的内容返回出去。

最后，我们在来看看 SWGIHandler 及其父类 BaseHandle 两个类，不容易啊！真正处理请求并产生结果的类终于出现了，代码如下（代码是精简过后的，只留下了最为关心的部分）：

```
class WSGIHandler(BaseHandler):
    request_class = WSGIRequest
    def __call__(self, environ, start_response):
        if self._request_middleware is None:
            self.load_middleware()
        request = self.request_class(environ)
        response = self.get_response(request)
        for middleware_method in self._response_middleware:
```

```
                response = middleware_method(request, response)
        return response


# WSGIHandler 的父类 BaseHandler 代码如下:
class BaseHandler(object):
    def __init__(self):
        self._request_middleware = self._view_middleware = \
        self._response_middleware = self._exception_middleware = None


    def load_middleware(self):
        from django.conf import settings
        self._request_middleware = []
        self._view_middleware = []
        self._response_middleware = []
        self._exception_middleware = []
        for middleware_path in settings.MIDDLEWARE_CLASSES:
            dot = middleware_path.rindex('.')
            mw_module, mw_classname = middleware_path[:dot], \
                                      middleware_path[dot+1:]
            mod = __import__(mw_module, {}, {}, [''])
            mw_class = getattr(mod, mw_classname)
            mw_instance = mw_class()
            if hasattr(mw_instance, 'process_request'):
                self._request_middleware.append(mw_instance.process_request)
            if hasattr(mw_instance, 'process_view'):
                self._view_middleware.append(mw_instance.process_view)
            if hasattr(mw_instance, 'process_response'):
                self._response_middleware.insert(0, mw_instance.process_response)
            if hasattr(mw_instance, 'process_exception'):
                self._exception_middleware.insert(0, mw_instance.process_exception)
```

```
def get_response(self, request):
    from django.core import exceptions, urlresolvers
    from django.conf import settings
    for middleware_method in self._request_middleware:
        response = middleware_method(request)
        if response: return response
    urlconf = getattr(request, "urlconf", settings.ROOT_URLCONF)
    resolver = urlresolvers.RegexURLResolver(r'^/', urlconf)
    callback, callback_args, callback_kwargs = resolver.resolve(request.path)
    for middleware_method in self._view_middleware:
        response = middleware_method(request, callback, \
                                     callback_args, callback_kwargs)
        if response: return response
    try:
        response = callback(request, *callback_args, **callback_kwargs)
    except Exception, e:
        for middleware_method in self._exception_middleware:
            response = middleware_method(request, e)
            if response: return response
    return response
```

以上代码的灰色字体部分，我们暂且不用理会，都是和 middleware 相关的，后面的章节我们在专门介绍，届时您再返回来专门看这些灰色字体的部分。 剩下的黑色字体的代码就是处理客户端请求的主要部分。

首先我们看 SWGIHandler 的__call__方法中，最为关键的几句代码，如下：

```
request_class = WSGIRequest
request = self.request_class(environ)
response = self. get_response (request)
return response
```

就是创建了一个 WSGIRequest 对象，并且作为参数调用父类的 get_response，然后返

回的 get_response 产生的结果 response。

WSGIRequest 的基类是 HttpRequest，按照 Django 的要求，所以这样的 Requset 的基类都应该是 HttpRequest；同样，所以真正处理请求并产生结果的 Handler 也都应该是 BaseHandler。

WSGIRequest 类就是负责读取从 client socket 调用 makefile 产生的’rb’参数的 rfile 的内容，并且把这些内容转化能比较为比较友好、符合 HTTP 协议所规定的标准的属性，GET 、POST、FILES、COOKIES、REQUEST 等。

注意，还有一个 raw_post_data 属性，该属性值就是直接读取的 client socket 的 rfile 的内容，在 ServerHandler 类当中把该类的字段 stdin（即 StreamRequestHandler 的 rfile 字段）赋值给了 environ 这个 dict， key 值为'wsgi.input'，并且该 environ 一路传到了这个 WSGIRequest。

WSGIRequest 的 path 字段也是从 environ 的’ PATH_INFO’这个 key 获取的，前面我们已经介绍了该’ PATH_INFO’ key 对应的内容的产生，它保存的就是当前 url 请求的路径。

那么我们再来看一下 SWGIHandler 的父类 BaseHandle 当中的 get_response 最为关键的几句代码，如下：

```
from django.core import exceptions, urlresolvers
from django.conf import settings
urlconf = getattr(request, "urlconf", settings.ROOT_URLCONF)
resolver = urlresolvers.RegexURLResolver(r'^/', urlconf)
callback, callback_args, callback_kwargs = resolver.resolve(request.path)
response = callback(request, *callback_args, **callback_kwargs)
return response
```

首先得到 urlconf 的模块，如果 request 当中没有特别指定的话，那默认的就是 settings 的 ROOT_URLCONF 了，该文件默认就是我们的 django project 下面的 urls.py 文件，当然您也可以自行修改您的 project 的 settings.py 的 ROOT_URLCONF 的属性为其他。

然后创建 RegexURLResolver 对象 resolver，并且把 urlconf 当作参数传递进去，在调用 resolver 的 resolve 方法的时候，把当前的 url path 即 request.path 传递进去，如果您的 urls.py 写的内容能匹配到这个 url path 的话，则会返回需要调用的函数和参数，例如：

根据我们前面的 hello 的例子，如果 request.path 是空的话，则会匹配 urls.py 当中的

(r'^$', 'views.index')这一行，callback 即 views.py 这个文件的 index 函数，callback_args, callback_kwargs 则都为空。

最后调用 callback, response = callback(request, *callback_args, **callback_kwargs) , 我们看到第一个参数是 requset，在这里该 request 就是 WSGIQuest 的对象，从这里我们就明白了，为什么我们 views 的函数，第一个参数必须是 HttpRequest 类型的参数。

到此，如果 callback 不出异常，最终产生了结果，并且该结果最终到达了我们前面介绍的 ServerHandle 当中的 result 变量当中，剩下的处理过程，我们已经在 ServerHandle 当中介绍过了。

通过上面的介绍，我们看到了一个从客户端的请求到最终获得请求的结果的整个基本过程，仅仅是基本过程，哈哈，接下来我们会介绍上面这个讲解过程当中我们为了避免干扰，省略的部分。

2.3 Middleware

Middleware 是 Django 当中非常重要的一部分，Django 本身的 Session 和 Authentication(用户验证)都是通过 Middleware 来实现的。

通过 BaseHandler 的 get_response 函数以及 WSGI 的__call__函数的代码（请参考上一节最后，灰色字体的代码），我们可以看到 Django 的 Middleware 按执行的顺序分为以下 4 种：

1．Request middleware

函数名：process_request

参数：request

用途：request 首先由 request middleware 们进行处理，如果 middleware 有返回值 response 的话，则直接返回该 response。例如 CacheMiddleware；

2．View middleware

函数名：process_view

参数：request, callback_args, callback_kwargs

用途：如果 request middleware 没有处理该 requset 的话，那么通过 RegexURLResolve 解析 request.path 后得到 callback, callback_args, callback_kwargs 三个变量，在真正调用

callback 这个变量实际指向的 callable 对象(一般是函数)被调用之前， 给一次机会让 wiew middleware 们来处理一下试试，如果连一个 wiew middleware 都没有，或者没有任何 wiew middleware 理会该 request 以及相关参数 callback_args, callback_kwargs，那么该 requset 及其参数就真正调用 callback 这个 callable 的对象了；

3．Exception middleware

函数名：process_exception

参数：request, e

用途：如果在调用 callback 这个 callable 的对象的时候发生了异常，怎么就把该 request 和异常的对象 e，传送给 exception middleware 来处理。如果您自己不理会这些异常的话，django 也会帮您处理，如果您觉得 django 默认处理的不好的话，那么就写自己的 exception middleware 吧；

4．Response middleware

函数名：process_response

参数：request, response

用途：通过上面的几个回合，request 处理了，就算异常也处理了！response 也产生了，如果您觉得还是有点意犹未尽的话，那么好吧，您甚至还可以对已经产生的最终结果 response 做任何手脚，就用 response middleware 吧，例如 GzipMiddleware 就把最终的 response 压缩了一番。

就像安装了那些 App 需要写在告诉 settings.py 的 INSTALLED_APPS 一样，安装了那些 Middleware 的模块也需要在 settings.py 的 MIDDLEWARE_CLASSES 当中加上。不过和 INSTALLED_APPS 不同的是，MIDDLEWARE_CLASSES 里面的每一项都是有顺序的，必须把依赖其他 Middleware 写被依赖的 Middleware 后面。例如默认就安装的两个 Middleware，AuthenticationMiddleware 就必须写在 SessionMiddleware 后面，因为用户验证是依赖于 Session 机制的。

我们可以使用 request.user 中的 user 就是 AuthenticationMiddleware 给我们的 request 添加的。

其中 Django 的 contrib 目录当中，已经自带了很多常用 Middleware，这一部分的源码都相对比较简单，我们不需要像上一节那样剖析了，如果感兴趣请自行阅读。

2.4 URLResolver

关于 URL Resolver 我们同样返回到 Runserver 最后的在介绍 BaseHandler 类的时候，从该类的 get_reponse 函数当中的 三句代码 开始说起，代码如下：

```
urlconf = getattr(request, "urlconf", settings.ROOT_URLCONF)
resolver = urlresolvers.RegexURLResolver(r'^/', urlconf)
callback, callback_args, callback_kwargs = resolver.resolve(request.path)
```

在 Runserver 那一节我们也对此做了简单的介绍，下面我们看一下 url 解析的详细过程，首先 RegexURLResolver 创建的第一个参数是要解析的正则表达，第二个参数就是要解析的 url patterns 模块。

在这里 `resolver = urlresolvers.RegexURLResolver(r'^/', urlconf)` 这句话自然就是解析 / 开头的，我们当前 django 工程的目录下的 urls.py 这个模块。 在 `resolver.resolve(request.path)` 这句话执行的时候，才真正创建调用/django/conf/urls/defaults.py 当中的 parttents 这个函数，产生了 RegexURLPattern 实例的一个 List，如果在您的 urls.py 当中有用到 include 的话，这个 List 当中实例是 RegexURLResolver 对对象。

不管是 RegexURLPattern 还是 RegexURLResolver 都有 resolve 方法、都按我们 urls.py 当中每一项写的正则表达式来解析，直到最后递归 resolve 到是 RegexURLPattern 对象位置。

其实我们可以认 RegexURLResolver 是 url 解析的节点，RegexURLPattern 是 url 解析的叶子，在 get_reponse 当中的 `resolver = urlresolvers.RegexURLResolver(r'^/', urlconf)` 这句话相当于创建了一个 url 解析树的根节点，然后下面可以有若干节点和叶子，每个节点下面还可以有若干节点和叶子，每个节点和叶子上都有一个正则表达式，每个叶子上还需要记录一个 callable 对象和一些默认参数。

当传送了一个字符串给这棵树的根节点后，就开始了层层解析，每个节点或叶子把传入的字符串和自己记录的正则表达式匹配，如果 OK 而且是叶子的话，那么到此位置，把该叶子记录的 callable 对象和解析的参数和记录的参数都统统返回；如果是节点的话，那么把和传入的字符串匹配的部分去掉，产生一个新的字符串，接着在自己的记录的节点和叶子当中传送匹配下去…….，直到找到一个匹配的叶子为止。如果无论如何都匹配不到，那就异常喽！

以上几节，我们几乎都是围绕着 django/core/handlers/base.py 和 django/http/__init__.py 当中的 BaseHandler、HttpRequest、HttpResponse 展开的，其实我觉得 django 的核心框架部分也就是这些内容了。以上我们只是以 WSGI 的实现方式讲解的，只要有核心定义这几个基类在，不管我们怎么实现都没所谓，HTTPServer 也未必就一定要用 Python 的 Lib 当中默认提供的。

其他的属于外围的 template、orm、forms（newforms）以及 contrib 目录下面的很多非常有用的 app 就不一一介绍了。

3 性能优化

3.1 CacheMiddleware

在这个内存越来越便宜的年代，获得性能提升的最快捷的途径就是：添加几条内存，打开您的页面缓存。

除了类似在线聊天、股票分析之类的需要非常及时的反应当前页面的最新变化的应用，大部分页面根本不必用户浏览哪个页面就去生成哪个页面，并且为此需要大量的操作数据库。即便是需要很好的体验，也可以使用 AJAX 在客户端先做一些手脚，而不必频繁的从服务器更新。

使用页面缓存，我们只需要在 settings.py 当中的 MIDDLEWARE_CLASSES 中增加 django.middleware.cache.CacheMiddleware 这句话就可以了。

如果想更详细的设置，最好那就在 setting.py 当中增加 以下三句话：

```
CACHE_BACKEND = 'locmem://' #设置缓存的方式(内存、文件、数据库 等)
CACHE_MIDDLEWARE_SECONDS = 600 #设置缓存的时间
CACHE_MIDDLEWARE_ANONYMOUS_ONLY=True #只对没注册用户使用页面缓存
```

3.2 Template&Tag

在 Django 的 Template 当中，如果想提高速度，我觉得最主要的手段还是缓存。不过我们上一节说的缓存是页面级别的，而我们这里要说的缓存是局部的。

更多的时候我们可能需要对页面当中某一部分来进行缓存，而不是整个页面。这时候就需要参考 django/core/cache 目录中定义的 Cache 对象来实现自己的 caching 了，然后在自己的 View 当中调用。或者我们可以创建 Tag 来生成页面的某一部分，然后在该 Tag 当中使用我们自己实现的 cahcing，从而减少数据库的读取提高 template 的生成过程。

当然，还有另外一个绝招，可以狠狠的提高 Template 的速度至少一倍，那就是使用 Mako 的 Template，只是该模板的使用习惯和 Django Template 有些不同，但是功能更强、速度更快。

这个 Template 的实现思路和 Django 的 template 相同，都是文法分析、语法分析、生成语法树、最后根据传入的内容进行进行替换，得到最终的结果。但是 Django 的 Templte 在得到最终结果以后就丢弃了，下次再调用模板还是重复一遍这样的操作，而 Mako 则不同，她居然在语法树之后，把模板的语法树生成了真真正正的 Python 代码，只要模板不变化，就不再重新编译，所以她可能是迄今世界上最快的 Python 语言的模板了，强烈推荐！

3.3 Database

数据库的优化，首先考虑的就是数据库结构是否合理，这方面可能需要咨询专家了，根据我们是 Oracle 数据库 还是 MySQL 数据库等等，来看看我们每一个表的字段、索引创建的是否合适。

如果您使用 Django 的 ORM 的话，剩下的就是 ORM 的问题了。Django 的 ORM 使用起来非常简单、自然，但是简单的 ORM 最好处理简单的事情，复杂的数据库操作就不要让她来做了。第一，她可能做不了；第二，即便做得了也未必能做好。所以我推荐，复杂的数据库操作最好还是直接写 SQL 比较好，至于 django 的 sqlalchemy 分支，好像是遥遥无期了，即便真的能使用 sqlalchemy，我觉得也够痛苦的，简单的事情不如使用 django 的 orm 简单；复杂的事情简直比直接写 SQL 还要复杂。