Django 基础知识

一. settings.py 说明

二. 日志

django 中的日志记录使用设定好的格式进行 logger、handler、filter、format 的设置以及关联，具体的配置在 settings.py 中完成，关于记录器、处理器、过滤器、格式器的说明请自行查看《pythong 基础知识》文档。

2.1 配置

django 使用 settings.py 中的全局变量 LOGGING 来配置 logger/handler/filter/format，从而将日志记录存入日志系统中。

2.1.1 disable_existing_loggers

是否禁用已经存在的日志配置

2.1.2 formatters

格式器，在该字典中命名子项或者子对象，用于后面的记录器、处理器调用

2.1.3 filters

过滤器，可以被记录器、处理器使用，用于过滤、修改日志记录

2.1.4 handlers

处理器，决定日志记录会被发送到哪里，记录那些等级的日志（和 logger 互相冲突哦）

2.1.5 loggers

记录器，包含多个 Handler，有选择的记录日志，记录的实践者。

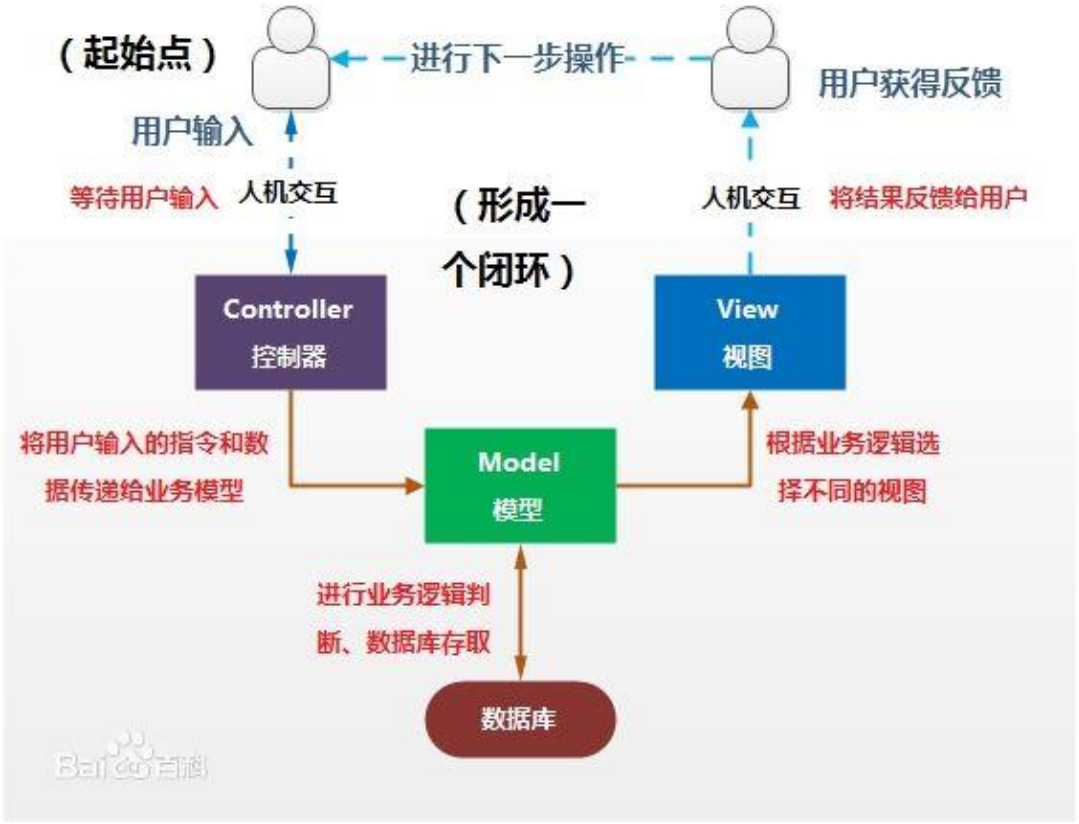
2.2 例子

```
LOGGING = {
    'version': 1,#指明dictConnfig的版本，目前就只有一个版本，哈哈
    'disable_existing_loggers': True,#禁用所有的已经存在的日志配置
    'formatters': {#格式器
        'verbose': {#详细
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {#简单
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {#过滤器
        'special': {#使用project.logging.SpecialFilter，别名special，可以接受其他的参数
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',#参数，名为foo，值为bar
        }
    },
    'handlers': {#处理器，在这里定义了三个处理器
        'null': {#Null处理器，所有高于（包括）debug的消息会被传到/dev/null
            'level':'DEBUG',
            'class':'django.utils.log.NullHandler',
        },
        'mail_admins': {#AdminEmail处理器，所有高于（包括）而error的消息会被发送给站点管理员，使用的是special格式器
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },

    'loggers': {#定义了三个记录器
        'django': {#使用null处理器，所有高于（包括）info的消息会被发往null处理器，向父层次传递信息
            'handlers':['null'],
            'propagate': True,
            'level':'INFO',
        },
        'django.request': {#所有高于（包括）error的消息会被发往mail_admins处理器，消息不向父层次发送
            'handlers': ['mail_admins','null'],
            'level': 'ERROR',
            'propagate': False,
        },
    }
}
```

三. web 原理

3.1 mvc



3.1.1 含义

controller——根据用户输入委派视图的部分

model——数据存取部分

views——选择显示那些数据以及怎样显示数据，views 和 template 联合处理

3.1.2 MVC（django）

controller：由 django 框架根据 URLConf 设置并调用响应的 views 函数，无需人工

models：models.py 中的 ORM 设置

views：用来描述展现给用户的数据

注意：

- django 中 Views 用来**描述展现给用户的数据**，而并非**数据如何展现并且展现那些数据**
- ruby 以及同类框架则提倡 Controller 负责决定向用户展现那些数据，视图决定如何展现数据，而并非展现数据

3.2 MTV

Django 更加关注的 Models-Template-Views，所以 django 也被称为 MTV 框架

M——Models（数据存取层）

T——template（表现层）

v——View（业务逻辑层）

3.3 models 知识点

3.3.1 记录和对象

3.3.1.1 说明

django 中每一个 models.Model 类（子类）的对象都对应数据库中的某一条记录，仅仅是某一条记录，对于某一个对象的修改和保存都是对该记录操作。

3.3.1.2 例子

```
>>> p1.name = 'bamboo33'
>>> p1.save()
>>> p1.name = 'bamboo44444'
>>> p1.save()
```

```
mysql> select * from books_publisher;
+----+----------+-----------+--------+----------------+---------+--------------------+
| id | name     | address   | city   | state_province | country | website            |
+----+----------+-----------+--------+----------------+---------+--------------------+
|  1 | bamboo33 | 中关村    | 北京   | BJ             | CHINA   | http://bamboo.com/ |
|  2 | bamboo1  | 中关村    | 北京   | BJ             | CHINA   | http://bamboo.com/ |
+----+----------+-----------+--------+----------------+---------+--------------------+
```

```
mysql> select * from books_publisher;
+----+-------------+-----------+--------+----------------+---------+
| id | name        | address   | city   | state_province | country |
+----+-------------+-----------+--------+----------------+---------+
|  1 | bamboo44444 | 中关村    | 北京   | BJ             | CHINA   |
|  2 | bamboo1     | 中关村    | 北京   | BJ             | CHINA   |
+----+-------------+-----------+--------+----------------+---------+
```

3.3.2 结构和行为

django 中的 Models 不仅仅为对象定义了数据库表的结构，而且通过__unicode__方法更好的展现了 Models 对于对象行为的影响（各个对象的具体数据信息）：

```
>>> p1
<Publisher: bamboo44444 中关村 北京 BJ CHINA http://bamboo.com/>
>>>
```

3.3.3 模型和管理器

- 模型：一个 models.Model 的子类，类似于抽象类的定义
- 管理器：objects 属性，类似于所有对象的中枢管理机构，管理着所有针对数据包含、数据查询的表格及操作

注意：**为了强制分离数据表级别和数据记录级别的操作（操作对象为表—类、操作对象为记录—实例），manager 只能存在 Models 类中，而不能存在实例中，即 p1.objects 无法访问**。

3.3.4 querySet

querySet 缓存集在整个 django 的 Models 中处于非常重要的作用，合理的使用缓存从而高效的替代 mysql 的复杂语句，例如连锁查询、限制查询、级联查询等等。

3.3.4.1 何时访问数据库

在操作 querySet 的过程中，不涉及任何的数据库查询操作，这才是最高效的原因。

一般在执行 querySet 的赋值操作、打印操作、切片步长操作时，会涉及到查询数据库操作（**实在顶不住了，得马上获取数据**）。

3.3.4.1.1 构造 querySet

```
>>> list = Publisher.objects.all()
```

3.3.4.1.2 修改数据库内容

```
mysql> update books_publisher set name='kuang' where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

3.3.4.1.3 获取 querySet 中的值

```
>>> list[0]
<Publisher: Apress 2855 Telegraph Avenue Berkeley CA U.S.A. http://www.apress.com/>
>>> list[1]
<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/>
>>> list[2]
<Publisher: kuang 中关村 北京 BJ CHINA http://bamboo.com/>
```

由此可以知道，querySet 只有在需要数据库中的值时才会执行数据库查询操作，“所用即所查”。

3.3.4.2 获取 querySet

过滤器（filter）、反向过滤器（exclude）、级联过滤器（filter().filter）、限量查询（切片[:5]）、排序过滤器（order_by）、去重过滤器（distinct）等等

四. 模型

4.1 save 操作

- 预存信号通知，相应“回调函数”处理
- 预处理某些数据，进行某些更改以适应数据库
- 将所有的数据的属性更改（转化）为数据库适应的类型
- 构造 SQL 语句，执行 sql 操作
- 发送完毕信号。

4.2 delete 操作

删除对象或者批量删除对象

4.3 update 操作

批量更改 querySet 中的所有字段的值

4.4 querySet 操作函数

4.4.1 排序

4.4.1.1 函数

order_by(name)

或者

order_by(name1, name2)

或者

order_by('-name1', 'name2', 'name3')

4.4.1.2 默认排序

使用内嵌类 Meta，指定默认的排序方式：

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
    def __unicode__(self):
        """unicode str"""
        return u'{0} {1} {2} {3} {4} {5}'.format(\
                self.name, self.address, self.city, \
                self.state_province, self.country, \
                self.website)
    class Meta:
        ordering = ['name']
```

```
>>> from books.models import Publisher
>>> list = Publisher.objects.all()
>>> list[0]
<Publisher: Apress 2855 Telegraph Avenue Berkeley CA U.S.A. http://www.apress.com/>
>>> list[1]
<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/>
>>> list[2]
<Publisher: bamboo44444 中关村 北京 BJ CHINA http://bamboo.com/>
```

4.4.1.3 sql 语句

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name;
```

4.4.1.4 例子

正向排序：

```
>>> plist = Publisher.objects.order_by("name")
>>> plist[0]
<Publisher: Apress 2855 Telegraph Avenue Berkeley CA U.S.A. http://www.apress.com/>
>>> plist[1]
<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/>
>>> plist[2]
<Publisher: bamboo44444 中关村 北京 BJ CHINA http://bamboo.com/>
```

双重条件：

```
>>> plist = Publisher.objects.order_by("name", "address")
>>> plist[0]
<Publisher: Apress 2855 Telegraph Avenue Berkeley CA U.S.A. http://www.apress.com/>
>>> plist[1]
<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/>
>>> plist[2]
<Publisher: bamboo44444 中关村 北京 BJ CHINA http://bamboo.com/>
```

逆向排序：

```
>>> plist = Publisher.objects.order_by("-name")
```

4.4.2 去重

4.4.2.1 函数

distinct()

4.4.2.2 sql 语句

select distinct * from A;

4.4.3 限量查找

4.4.3.1 定义

pList[3:8]

或者

pList[:5]

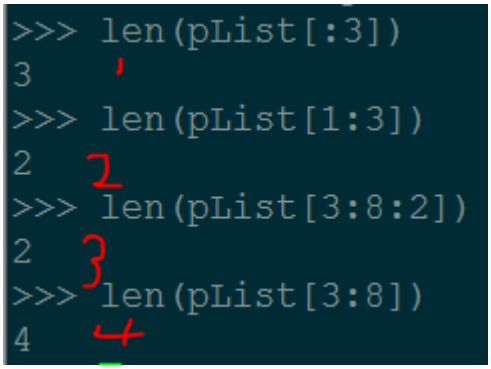
或者

pList[3:8:2]

4.4.3.2 sql 语句

select * from A limit 5 offset 3;

4.4.3.3 例子



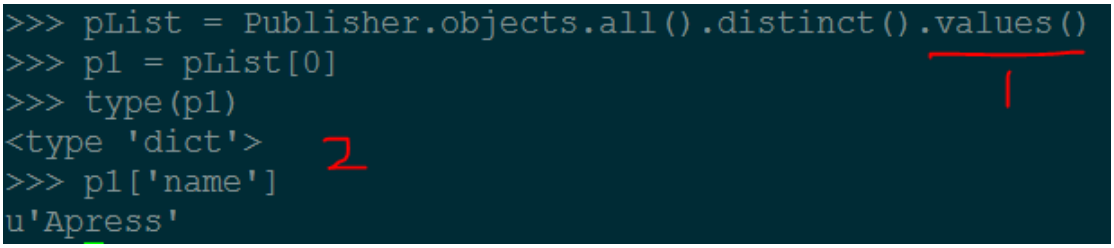
类似于 python 中列表的切片操作，注意和 limit/offset 的对应关系。

4.4.4 字典返回

4.4.4.1 定义

values()

4.4.4.2 例子



4.4.5 extra

在某些情况下，django 的 query syntax 无法准确的描述复杂的 SQL 语法，django 提供了 extra 函数，用于直接执行 sql 语句并返回 QuerySet，但是尽可能的少用该方式进行数据库与查询操作。

注意：**请区分 extra 和 exact 的区别，后者是精确匹配**

4.4.5.1 定义

extra(select={…}, …)，其中参数有 params/select/where/tables

4.4.5.2 例子

```
>>> pList = Publisher.objects.extra(where=["city IN ('北京', '深圳')"])
>>> pList
[<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/ 2015-11-07>,
p://bamboo.com/ 2015-12-07>]
>>> pList = Publisher.objects.extra(where=['city=%s'], params=['北京'])
>>> pList
[<Publisher: bamboo1 中关村 北京 BJ CHINA http://bamboo.com/ 2015-11-07>,
p://bamboo.com/ 2015-12-07>]
```

4.5 单一对象

4.5.1 get

4.5.1.1 定义

get(**lookup)，返回的某一个特定的对象，并非一个 querySet

4.5.1.2 异常

- 返回值超过一个对象，抛出 MultipleObjectsReturned 异常
- 返回值为空，抛出 DoesNotExist 异常

4.5.1.3 例子

```
>>> p1 = Publisher.objects.get(city='北京')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/usr/lib/python2.6/site-packages/django/db/models/manager.py",
    return self.get_query_set().get(*args, **kwargs)
  File "/usr/lib/python2.6/site-packages/django/db/models/query.py", li
    % (self.model._meta.object_name, num, kwargs))
MultipleObjectsReturned: get() returned more than one Publisher -- it r
xe5\x8c\x97\xe4\xba\xac'}
>>> p1 = Publisher.objects.get(id=1)
>>> p1
<Publisher: kuang 中关村 北京 BJ CHINA http://bamboo.com/ 2015-12-07>
```

4.5.2 create

4.5.2.1 定义

create(**kwargs)

4.5.2.2 例子

```
>>> p2 = Publisher.objects.create(name='bifeng', address='西二旗', city='BJ')
>>> p2.save
<bound method Publisher.save of <Publisher: [Bad Unicode data]>>
>>> p2.save()
```

4.5.3 get_or_create

4.5.3.1 定义

get_or_create(**kwargs)，如果获取键值失败，就创建一个新的对象，一般用于 GET 请求中，如果非要在 POST 请求中，则必须注意对数据的影响

4.5.3.2 例子

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=date(1940, 10, 9))
    obj.save()
```

This pattern gets quite unwieldy as the number of fields in a model increases. The previous example can be rewritten using `get_or_create()` like so:

```
obj, created = Person.objects.get_or_create(
    first_name = 'John',
    last_name  = 'Lennon',
    defaults   = {'birthday': date(1940, 10, 9)}
)
```

4.5.4 latest

4.5.4.1 定义

latest(field_name=...)，返回数据库中的最后一条记录

4.5.4.2 例子

```
>>> p1 = Publisher.objects.latest('name')
>>> pList = Publisher.objects.all()
>>> pList.count()
8
>>> pList[7]
<Publisher: test1 sand 纽约 CA U.S.A  2014-10-07>
>>> p1
<Publisher: test1 sand 纽约 CA U.S.A  2014-10-07>
```

4.6 field 操作

4.6.1 格式

filed__lookuptype=value，其中 field 表示数据库中的某一个字段，lookuptype 表示进行查找的操作，value 即相应的值，__表示调用魔术方法

4.6.1.1 LIKE

4.6.1.2 exact

精确匹配字段

4.6.1.3 iexact

大小写不敏感的精确匹配字段，相关 SQL 语法为 LIKE

4.6.1.4 contains

内容包含检测类型，相关 SQL 语句为 LIKE

4.6.1.5 icontains

大小写不敏感的内容包含检测，相关 SQL 语句为 LIKE

4.6.1.6 startswith

精确开头匹配，相关 SQL 语句为 LIKE

4.6.1.7 istartswith

大小写不敏感的精确开头匹配，相关 SQL 语句为 LIKE

4.6.1.8 endswith

精确字段结尾匹配，相关 SQL 语句为 LIKE

4.6.1.9 iendswith

大小写不敏感的精确结尾匹配，相关 SQL 语句为 LIKE

4.6.2 logical

4.6.2.1 gt

大于

4.6.2.2 gte

大于等于

4.6.2.3 lt

小于

4.6.2.4 lte

小于等于

4.6.2.5 =

等于

4.6.2.6 range

所有包含在指定范围的所有数据对象集合

```
>>> start_date = datetime.date(2005, 1, 1)
>>> end_date = datetime.date(2005, 3, 31)
>>> Entry.objects.filter(pub_date__range=(start_date, end_date))
```

4.6.3 in

所有包含在列表中的任意一个元素的所有集合

4.6.4 时间操作

4.6.4.1 dates 排序

4.6.4.1.1 定义

dates(field, kind, order)：

- field:属性字段
- kind:关注的日期字段，按照该字段以及前面的值进行排序比较
- order:排序，默认为 ASC，另外有 DESC

4.6.4.1.2 例子

```
>>> pList.dates('put_date', 'year')
[None, datetime.datetime(2014, 1, 1, 0, 0, tzinfo=<UTC>), datetime.datetime(2015, 1, 1, 0, 0, tzinfo=<UTC>)]
>>> pList.dates('put_date', 'month')
[None, datetime.datetime(2014, 10, 1, 0, 0, tzinfo=<UTC>), datetime.datetime(2014, 11, 1, 0, 0, tzinfo=<UTC>), datet
ime.datetime(2015, 11, 1, 0, 0, tzinfo=<UTC>), datetime.datetime(2015, 12, 1, 0, 0, tzinfo=<UTC>)]
>>> pList.dates('put_date', 'day')
[None, datetime.datetime(2014, 10, 6, 0, 0, tzinfo=<UTC>), datetime.datetime(2014, 10, 7, 0, 0, tzinfo=<UTC>), datet
ime.datetime(2014, 11, 6, 0, 0, tzinfo=<UTC>), datetime.datetime(2014, 11, 7, 0, 0, tzinfo=<UTC>), datetime.datetime
(2015, 11, 7, 0, 0, tzinfo=<UTC>), datetime.datetime(2015, 12, 7, 0, 0, tzinfo=<UTC>)]
```

4.6.4.2 date 查找

查找特定的 dates 字段，返回 querySet 集合

```
>>> pList = Publisher.objects.filter(put_date__month=11)
>>> pList
[<Publisher: Apress 2855 Telegraph Avenue Berkeley CA U.S.A. http://www.apress.com/ 2014-11-07>,
 中关村 北京 BJ CHINA http://bamboo.com/ 2015-11-07>, <Publisher: Lucy sunshine 纽约 CA U.S.A
```

4.7 杂项

4.7.1 count

获取 querySet 中对象的个数

4.7.2 in_bulk

4.7.2.1 定义

in_bulk(id_list)，传入 primary-key 列表，返回对应的 id-record 字典

4.7.2.2 例子

```
>>> pList = Publisher.objects.in_bulk([1, 3])
>>> pList
{1L: <Publisher: kuang 中关村 北京 BJ CHINA http://bamboo.com/ 2015-12-07>, 3L: <Publisher:
enue Berkeley CA U.S.A. http://www.apress.com/ 2014-11-07>}
```

4.8 异常

4.8.1 DoesNotExist

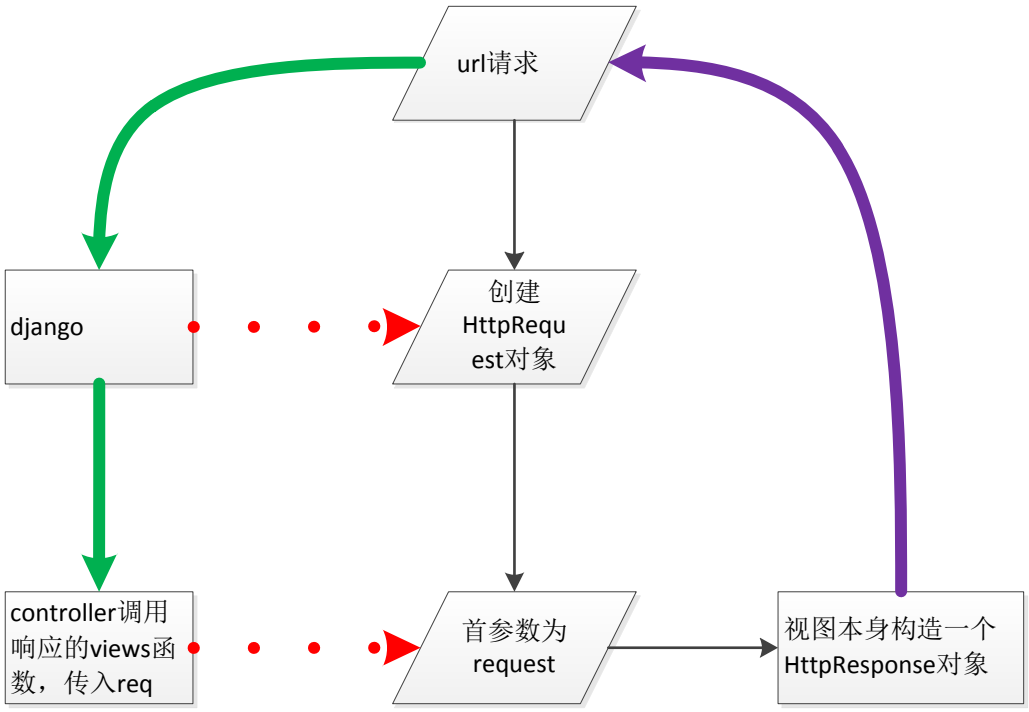
key 值（数据库列值）不存在异常，为类 Models 的内置属性

4.8.2 MultipleObjectsReturned

返回多个值异常

五. request 数据

5.1 介绍



其中每一个 HttpRequest 都代表了一个来自客户端的 HTTP 请求，包含了连接相关的各种详细信息。

5.2 QueryDict

类似字典的类（字典的一个特殊子类），用于处理一键多值情况（字典中可没有哦），其中 Request 中的 GET 和 POST 属性都是该类实例对象。

5.2.1 只读

该类实例是不可更改的，除非创建了一个 copy()副本，否则无法更改，即 Request.GET 以及 Request.POST 都是不可更改的属性实例。

5.2.2 方法

请自行查看《django book》文档。

5.3 属性

所有这些属性在 Request 中都是延迟加载，即用到的时候才会去加载这个类，否则 django 不会花费计算资源去处理这些属性值。

5.3.1 path

不包含域名的 URL 路径字符串

5.3.2 method

提交请求使用的 HTTP 方法，例如”GET”,”POST”,”PUT”等。

5.3.3 GET

QueryDict 实例对象。

5.3.4 POST

QueryDict 实例对象，有些时候 request.POST 为空。

5.3.5 REQUEST

为了方便而创建，一个类字典对象。

5.3.6 COOKIES

标准的 python 字典，包含所有的 cookies 键值对。

5.3.7 FILES

类字典对象，包含所有上传的文件

5.3.7.1 filename

上传文件名，请求中 filename 字段。

5.3.8 META

标准的 python 字典类型，包含了所有的 HTTP 头信息，存储的值类似于 nginx 处理过程中 ngx_http_request_s→ngx_http_headers_in_t 头信息。

5.3.8.1 客户端服务器相关

- CONTENT_LENGTH——报文实体当前的总大小
- CONTENT_TYPE——例如 json 格式等
- QUERY_STING——原始请求字符串，例如 key=value1&key2=value2
- REMOTE_ADDR——客户端 IP 地址
- REMOTE_HOST——客户端主机名
- SERVER_NAME——服务器主机名
- SERVER_PORT——服务器端口号

5.3.8.2 任意有效头信息

- HTTP_ACCEPT_ENCODING——客户端接收的编码
- HTTP_ACCEPT_LANGUAGE——客户端接收的语言
- HTTP_HOST——host 字段
- HTTP_REFERENCE——reference 页面
- HTTP_USER_AGENT——user-agent 字符串

5.3.9 user

当前登录的用户，一般使用 user. is_authenticated()来区分是否为真正的登录用户。

5.3.10 session

可读写的 session 对象

5.3.11 raw_post_data

原始数据，用于处理复杂数据

5.4 获取请求体

5.4.1 form-encoded data

获取方式：request.POST['data'] 或者 request.POST.get('data')

5.4.2 raw json data

对于 post 请求中（特别是使用 request 模块进行 post 请求）的数据，不能简单的使用 POST['key1']或者 POST.get('key1')来获取。

获取方式：received_json_data=json.loads(request.body)

六. response 数据

视图中自定义构造 HttpResponse 对象或者子类对象，之后返回给客户端

6.1 构造 response

字符串形式

或者

迭代器形式

6.2 headers 操作

6.2.1 删除和添加

```
>>> response = HttpResponse()
>>> response['X-DJANGO'] = "It's the best."
>>> del response['X-PHP']
>>> response['X-DJANGO']
"It's the best."
```

注意：**避免手动设置 cookie 字段**

6.2.2 判断

response.has_header('X-1')

6.3 子类

6.3.1 HttpResponseRedirect

临时重定向响应数据（302），参数为“重定向路径”

6.3.2 HttpResponsePermanentRedirect

永久重定向（301），其他类似 HttpResponseRedirect

6.3.3 HttpResponseNotModified

304 状态码，无参数

6.3.4 HttpResponseBadRequest

400 状态码，参数同 HttpResponse

6.3.5 HttpResponseNotFound

404 状态码，参数同 HttpResponse

6.3.6 HttpResponseForbidden

403 状态码，参数同 HttpResponse

6.3.7 HttpResponseNotAllowed

405 状态码，参数同 HttpResponse，但增加方法列表参数

6.3.8 HttpResponseGone

410 状态码，参数同 HttpResponse

6.3.9 HttpResponseServerError

500 状态码，参数同 HttpResponse

6.4 错误信息

6.4.1 返回错误类响应

上一节中大部分的 HttpResponse 子类都是用于返回某些特定的错误信息，用于标识特定的错误。

6.4.2 404 处理

内置或者自定义

6.4.3 500 处理

内置或者自定义