Django 请求处理流程总结

应用程序入口 WSGIHandler

WSGI 有三个部分，分别为服务器(server)，应用程序(application) 和中间件(middleware).

已经知道，服务器方面会调用应用程序来处理请求，在应用程序中有真正的处理逻辑，在这里面几乎可以做任何事情，其中的中间件就会在里面展开.

任何的 WSGI 应用程序,都必须是一个 start_response(status, response_headers, exc_info=None) 形式的函数或者定义了 __call__ 的类. 而 django.core.handlers 就用后一种方式实现了应用程序: WSGIHandler.

def __call__(self, environ, start_response) 方法中调用了 WSGIHandler.get_response() 方法以获取响应数据对象 response. 从 WSGIHandler 的实现来看，它并不是最为底层的: WSGIHandler 继承自 base.BaseHandler

核心类 BaseHandler

BaseHandler 类有两个核心方法，load_middlware() 和 get_response().load_middleware() 函数会根据 settings.py 中的 MIDDLEWARE_CLASSES 导入所有的中间件. 每一个中间件都是一个类，其内部会实现 process_request(), process_view(), process_template_response(), process_response() 或者 process_exception() 方法. 不一定都实现，看需求. 而这些方法如果存在，都会被保存响应的函数列表中，待将来调用. get_response() 方法会按顺序调用保存在各个中间件列表总中的方法， 顺序是 _*request*_middleware, _*view*_middleware, _*exception*_middleware, _*template*_response_middleware (不一定都有)。

url dispatcher

django 内部的 url 调度机制说白了就是给一张有关匹配信息的表，这张表中有着 url -> action 的映射，当请求到来的时候，一个一个(遍历)去匹配. 中，则调用 action，产生相应数据返回; 不中，则会产生 404 等的错误。

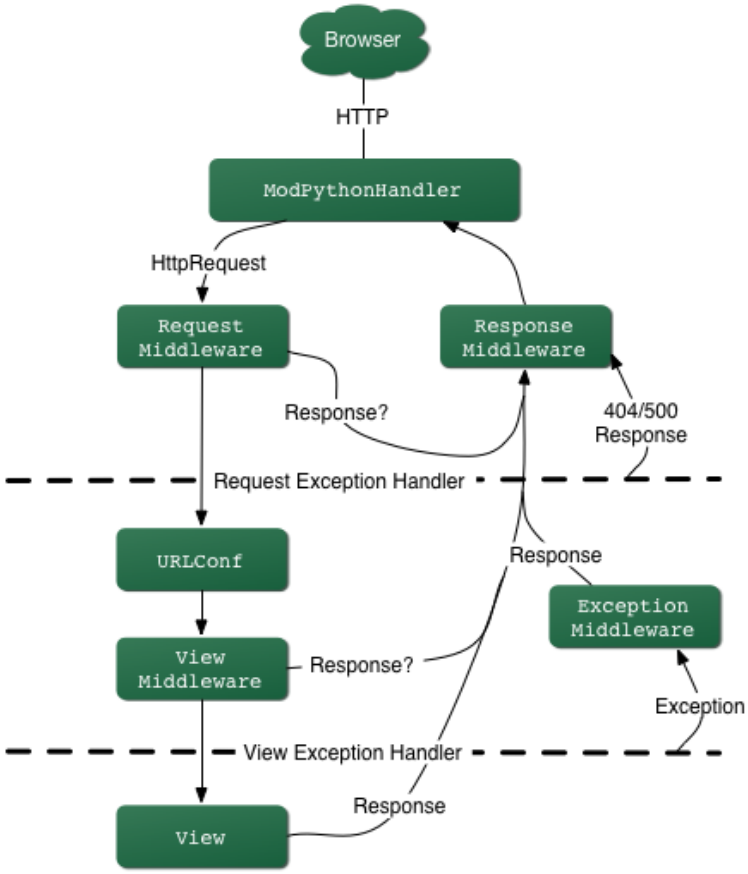
同样在 get_response()方法中实例化 RegexURLResolver，我将其理解为一个 url 的匹配处理器，然后启动匹配的函数，调用 RegexURLResolver.resolve()，返回 resolve_match(ResolverMatch 实例), resolver_match 对象中存储了有用的信息，比如 callback 就是我们在 views.py 中定义的函数，参数 callback_args, callback_kwargs。

响应数据返回：

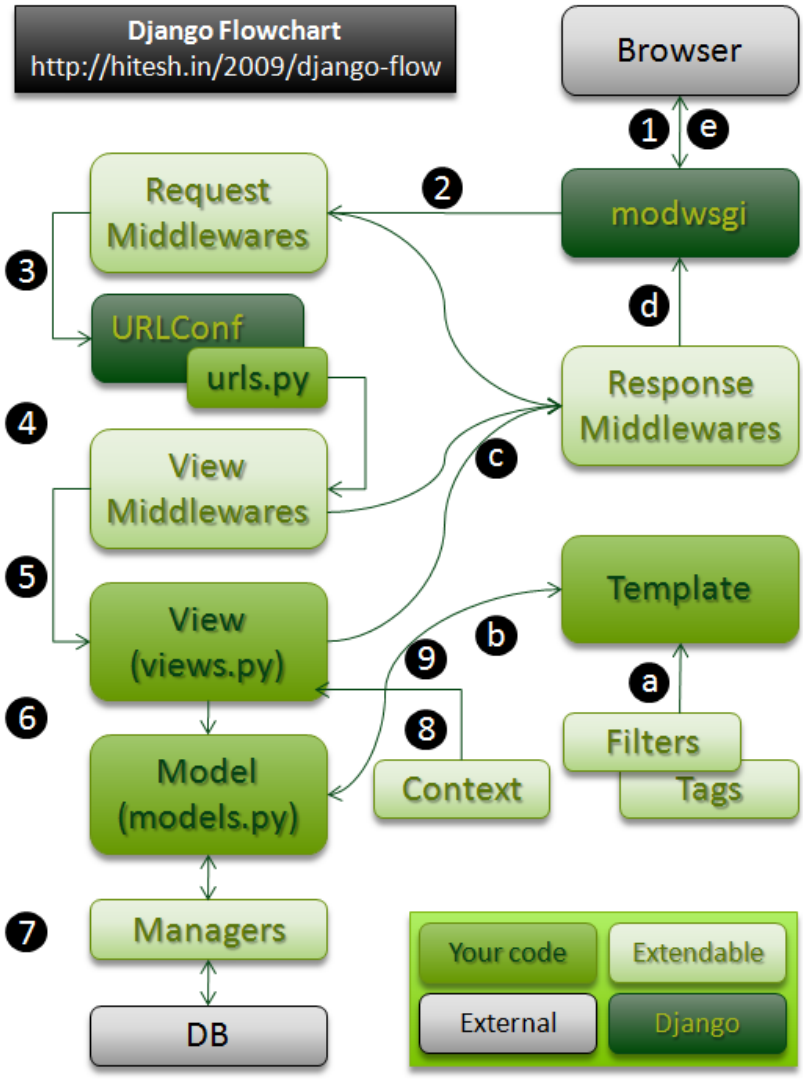
在 WSGIHandler.call(self, environ, start_response) 方法调用了 WSGIHandler.get_response() 方法，由此得到响应数据对象 response. ServerHandler 产生响应数据对象后，调用 finish_response()方法，向 socket 写数据，返还给上游，关闭连接。

Django 处理 Request 的详细流程

首先分享两个网上看到的 Django 流程图：



Django 流程图 1



Django 流程图 2

上面的两张流程图可以大致描述 Django 处理 request 的流程，按照流程图 2 的标注，可以分为以下几个步骤：

1. 用户通过浏览器请求一个页面
2. 请求到达 Request Middlewares，中间件对 request 做一些预处理或者直接 response 请求
3. URLConf 通过 urls.py 文件和请求的 URL 找到相应的 View
4. View Middlewares 被访问，它同样可以对 request 做一些处理或者直接返回 response
5. 调用 View 中的函数
6. View 中的方法可以选择性的通过 Models 访问底层的数据
7. 所有的 Model-to-DB 的交互都是通过 manager 完成的
8. 如果需要，Views 可以使用一个特殊的 Context
9. Context 被传给 Template 用来生成页面

a. Template 使用 Filters 和 Tags 去渲染输出

b. 输出被返回到 View

c. HTTPResponse 被发送到 Response Middlewares

d. 任何 Response Middlewares 都可以丰富 response 或者返回一个完全不同的 response

e. Response 返回到浏览器，呈现给用户

下面对重要部分中间件再展开一下

Middleware(中间件)

Middleware 并不是 Django 所独有的东西，在其他的 Web 框架中也有这种概念。在 Django 中，Middleware 可以渗入处理流程的四个阶段: request，view，response 和 exception，相应的 ,在每个 Middleware 类中都有 process_request ,process_view， process_response 和 process_exception 这四个方法。可以定义其中任意一个活多个方法，这取决于希望该 Middleware 作用于哪个处理阶段。每个方法都可以直接返回 response 对象。

Middleware 是在 Django BaseHandler 的 load_middleware 方法执行时加载的，加载之后会建立四个列表作为处理器的实例变量：

_request_middleware：process_request 方法的列表

_view_middleware：process_view 方法的列表

_response_middleware：process_response 方法的列表

_exception_middleware：process_exception 方法的列表

Django 的中间件是在其配置文件(settings.py)的 MIDDLEWARE_CLASSES 元组中定义的。在 MIDDLEWARE_CLASSES 中 ,中间件组件用字符串表示 指向中间件类名的完整 Python 路径。

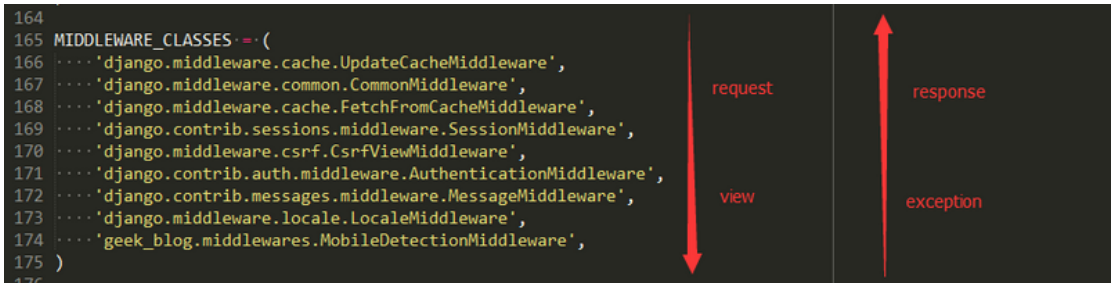
```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.locale.LocaleMiddleware',
```

)

Django 项目的安装并不强制要求任何中间，MIDDLEWARE_CLASSES 可以为空。中间件出现的顺序非常重要，在 request 和 view 的处理阶段，Django 按照 MIDDLEWARE_CLASSES 中出现的顺序来应用中间件，而在 response 和 exception 异常处理阶段，Django 则按逆序来调用它们。也就是说，Django 将 MIDDLEWARE_CLASSES 视为 view 函数外层的顺序包装子，在 request 阶段按顺序从上到下穿过，而在 response 则反过来。

Django Middleware 流程

python 重要特性总结：



- **with 调用上下文管理器**

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

通常使用 with 语句调用上下文管理器，上下文管理器（Context Manager）：支持上下文管理协议的对象，这种对象实现了__enter__() 和 __exit__() 方法。上下文管理器定义执行 with 语句时要建立的运行时上下文，负责执行 with 语句块上下文中的进入与退出操作。

```
with context_expression [as target(s)]:
        with-body
```

这里 context_expression 要返回一个上下文管理器对象，该对象并不赋值给 as 子句中的 target(s) ，如果指定了 as 子句的话，会将上下文管理器的 __enter__() 方法的返回值赋值给 target(s)。target(s) 可以是单个变量，或者由“()”括起来的元组（不能是仅仅由“,”分隔的变量列表，必须加“()” ）。

with 语句执行过程：

```
context_manager = context_expression
exit = type(context_manager).__exit__
```

```
    value = type(context_manager).__enter__(context_manager)
    exc = True     # True 表示正常执行，即便有异常也忽略；False 表示重新抛出异常，需要对异常进行处理
    try:
        try:
            target = value   # 如果使用了 as 子句
            with-body        # 执行 with-body
        except:
            # 执行过程中有异常发生
            exc = False
            # 如果 __exit__ 返回 True，则异常被忽略；如果返回 False，则重新抛出异常
            # 由外层代码对异常进行处理
            if not exit(context_manager, *sys.exc_info()):
                raise
    finally:
        # 正常退出，或者通过 statement-body 中的 break/continue/return 语句退出
        # 或者忽略异常退出
        if exc:
            exit(context_manager, None, None, None)
        # 缺省返回 None，None 在布尔上下文中看做是 False
```

1. 执行 context_expression，生成上下文管理器 context_manager
2. 调用上下文管理器的 __enter__() 方法；如果使用了 as 子句，则将 __enter__() 方法的返回值赋值给 as 子句中的 target(s)
3. 执行语句体 with-body
4. 不管是否执行过程中是否发生了异常，执行上下文管理器的 __exit__() 方法，__exit__() 方法负责执行“清理”工作，如释放资源等。如果执行过程中没有出现异常，或者语句体中执行了语句 break/continue/return，则以 None 作为参数调用 __exit__(None, None, None) ；如果执行过程中出现异常，则使用 sys.exc_info 得到的异常信息为参数调用 __exit__(exc_type, exc_value, exc_traceback)
5. 出现异常时，如果 __exit__(type, value, traceback) 返回 False，则会重新抛出异常，让 with 之外的语句逻辑来处理异常，这也是通用做法；如果返回 True，则忽略异常，不再对异常进行处理

Python 对一些内建对象进行改进，加入了对上下文管理器的支持，可以用于 with 语句中，比如可以自动关闭文件、线程锁的自动获取和释放等。

实例定义一个上下文管理器，将文件里读取到的 key 存进 redis：

```
## contextmanager
class UserKeyProcessor:
    def __init__(self):
        self.pipeline = RedisClient.pipeline()
    def store(self, name):
        self.pipeline.setex(self, name, value)
        ...
    def delete(self, name):
        self.pipeline.delete(name)
        ...
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.pipeline.execute()
```

- **yield 定义生成器 generator**

一个生成器函数的定义很像一个普通的函数，除了当它要生成一个值的时候，使用 yield 关键字而不是 return。如果一个 def 的主体包含 yield，这个函数会自动变成一个生成器（即使它包含一个 return）。

生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。当函数执行结束时，generator 自动抛出 StopIteration 异常，表示迭代完成。在 for 循环里，无需处理 StopIteration 异常，循环会正常结束。

例子， 将读取类文件对象的类 FileWrapper（包括好几个方法） 改用 yield 实现

```
def read_filelike(filelike, blksize):
    while True:
        data = filelike.read(blksize)
        if data:
            yield data
        else:
            return
```

在此，总结一下 迭代器 iterator 和 可迭代对象 iterable。

- 可以直接作用于 for 循环的对象统称为可迭代对象：Iterable。
- 像 generator 这种可以被 next()函数调用并不断返回下一个值的对象称为迭代器：Iterator，iterator 表示的对象是数据流，可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 next()函数实现按需计算下一个数据，所以 Iterator 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

所以 ,集合数据类型 ,如 list、tuple、dict、set、str 等是可迭代对象 iterable，而不是 iterator。

- **闭包和装饰器**

闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。简单的说一个闭包就是调用了一个函数 A ,这个函数 A 返回了一个函数 B 给你。这个返回的函数 B 就叫做闭包。在调用函数 A 的时候传递的参数就是自由变量。

闭包的最大特点是可以将父函数的变量与内部函数绑定 ,并返回绑定变量后的函数(也即闭包），此时即便生成闭包的环境（父函数）已经释放，闭包仍然存在，这个过程很像类生成实例，不同的是父函数只在调用时执行，执行完毕后其环境就会释放，而类则在文件执行时创建，一般程序执行完毕后作用域才释放，因此对一些需要重用的功能且不足以定义为类的行为，使用闭包会比使用类占用更少的资源，且更轻巧灵活。

举个例子

```
def func(name):
    def inner_func(age):
        print 'name:', name, 'age:', age
    return inner_func
bb = func('lxkaka')
bb(26)   # >>> name: lxkaka age: 26
```

这里面调用 func 的时候就产生了一个闭包——inner_func,并且该闭包持有自由变量——name，因此这也意味着，当函数 func 的生命周期结束之后，name 这个变量依然存在，因为

它被闭包引用了，所以不会被回收。

装饰器就是一种的闭包的应用，只不过其传递的是函数。装饰器 decorator 从字面上说就是用来装饰一个函数，decorator 可以增强被修饰函数的功能，而不修改函数，比如可以用来做参数检查，插入日志，性能分析，权限校验等等。有了装饰器，我们就可以抽离出大量函数中与函数功能本身无关的相同代码并继续重用。提高了程序的可重复利用性，并增加了程序的可读性。

实例，为函数添加打印开始时间

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

在函数定义前使用 docorator log

```
@log
def add(x, y):
    return x+y
```

使用中我们当然也可以向装饰器传参，比如@decorator(x); 也可以使用多层 decorator 或者类装饰器。此外需要注意的是使用装饰器的函数的元信息，比如__name__，docstring 会被 decorator 返回的函数元信息替代，我们用 functools.wraps，wraps 本身也是一个装饰器，它能把原函数的元信息拷贝到装饰器函数中，这使得装饰器函数也有和原函数一样的元信息。

```
def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

- property

使用 Python 内建的 property 装饰器。@property 一般应用在 Python 方法上，可以有效地将方法调用（method call）变成属性访问（attribute access）。

举个简单例子：

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
```

```
    @property
    def full_name(self):
        return '{} {}'.format(self.first, self.last)
```

full_name 被定义为一个方法，但却可以通过变量属性的方式访问。用 property 装饰器可以说是创建了某种动态属性。

@property 把 getter 方法变成了属性， @x.setter 则把 setter 方法变成属性赋值。

简单例子：

```
class Student(object):
    @property
    def testname(self):
        return self.name
    @testname.setter
    def testname(self,name):
        self.name = name
```

用 property 一个最大的好处就是比如我们想修改一个类原本的属性，就可以把这个属性修改成方法，用 property 装饰，这样就不用更改调用过这个类的代码，只需要修改类本身。

- **利用 __setattr__， __getattr__ 拓展数据类型**

  对字典的访问支持 dict.key 的方式

```
class ExtendDict(dict):
    def __getattr__(self, key):
        return self[key]
    def __setattr__(self, key, value):
        self[key] = value
    def __call__(self, key):
        return self[key]
```