Django 源码

1. manage.py 源码
2. URLconf 源码——偏函数
3. view 源码——HttpRequest 与 HttpResponse
4. ORM 源码——元类
5. Template 源码——render 方法的实现

URLconf-URL调度器

典型写法：

```
urlpatterns = [
  path('douban', views.books_short),
   re_path(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
]
```

从源码层面对比path()、re_path() 区别

URLconf-URL 调度器

```
# site-packages/django/urls/conf.py

path = partial(_path, Pattern=RoutePattern)

re_path = partial(_path, Pattern=RegexPattern)
```

官方文档： https://docs.python.org/zh-cn/3.7/library/functools.html

partial 函数的实现

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

partial 函数的实现

1. 闭包(装饰器)

2. 怎么实现参数处理的

3. 除了实现功能，还考虑了哪些额外的功能

partial 函数的实现

```
# 官方文档 demo

from functools import partial

basetwo = partial(int, base=2)

basetwo.__doc__ = 'Convert base 2 string to an int.'

basetwo('10010')
```

输出：18

partial 函数的注意事项

1 partial 第一个参数必须是可调用对象

2 参数传递顺序是从左到右，但不能超过原函数参数个数

3 关键字参数会覆盖 partial 中定义好的参数

include 函数

```
# site-packages/django/urls/conf.py

def include(arg, namespace=None):
    if isinstance(arg, tuple):
        pass
    if isinstance(urlconf_module, str):
        urlconf_module = import_module(urlconf_module)
        patterns = getattr(urlconf_module, 'urlpatterns', urlconf_module)
        app_name = getattr(urlconf_module, 'app_name', app_name)
    if isinstance(patterns, (list, tuple)):
        pass
    return (urlconf_module, app_name, namespace)
```

请求与响应

HttpRequest 创建与 HttpResponse 返回是一次 HTTP 请求的标准行为。

Path 将请求传递给view视图函数，request怎么得到的？如何返回的？

HttpRequest 由 WSGI 创建，HttpResponse 由开发者创建。

View 视图抽象出的两大功能：返回一个包含被请求页面内容的 HttpResponse 对象，或者抛出一个异常，比如 Http404 。

请求

from django.http import HttpRequest 包含大量的属性和方法，如：

```
self.META = {}      # 包含所有的HTTP头部
self.GET = QueryDict(mutable=True)  # 包含HTTP GET的所有参数
```

做如下请求：

http://127.0.0.1:8000/?id=1&id=2&name=wilson

```
def index(request):
    print(request.GET)
    # <QueryDict: {'id': ['1', '2'], 'name': ['wilson']}>
    return HttpResponse("Hello Django!")
```

QueryDict

```
# site-packages/django/utils/datastructures.py
```

QueryDict 继承自 MultiValueDict，MultiValueDict 又继承自 dict

```
class MultiValueDict(dict):
    def __init__(self, key_to_list_mapping=()):
        super().__init__(key_to_list_mapping)

    def __repr__(self):
        return "<%s: %s>" % (self.__class__.__name__, super().__repr__())

    def __getitem__(self, key):

... ...
```

响应

```
def test1(request):
    # 已经引入了HttpResponse
    # from django.http import HttpResponse
    response1 = HttpResponse()
    response2 = HttpResponse("Any Text", content_type="text/plain")
    return response1

def test2(request):
    # 使用HttpResponse的子类
    from django.http import JsonResponse
    response3 = JsonResponse({'foo': 'bar'})  # response.content
    response3['Age'] = 120

    # 没有显式指定 404
    from django.http import HttpResponseNotFound
    response4 = HttpResponseNotFound('<h1>Page not found</h1>')
    return response4
```

HttpResponse子类

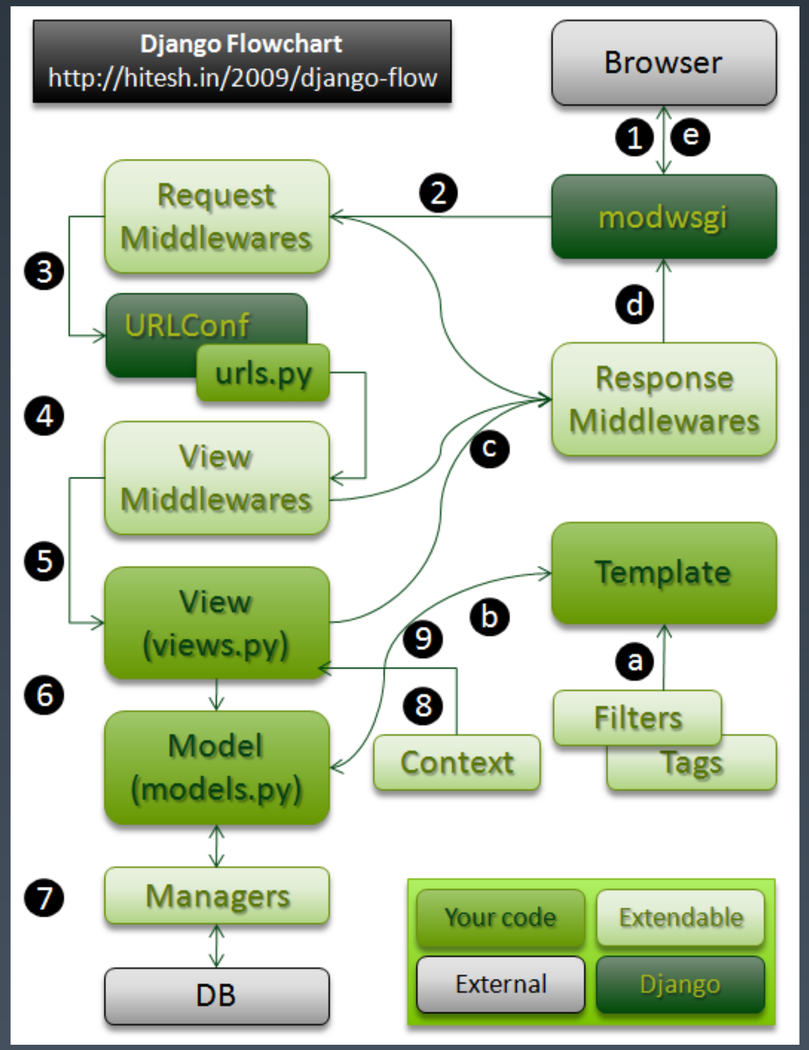
HttpResponse.content：响应内容

HttpResponse.charset：响应内容的编码

HttpResponse.status_code：响应的状态码

JsonResponse 是 HttpResponse 的子类，专门用来生成 JSON 编码的响应。

从请求到响应


Django Flowchart
http://hitesh.in/2009/django-flow
Browser
1
e
2
Request Middlewares
modwsgi
3
URLConf
urls.py
d
Response Middlewares
4
View Middlewares
c
5
View (views.py)
Template
b
9
a
6
8
Filters
Tags
Model (models.py)
Context
7
Managers
DB
Your code
Extendable
External
Django

Model

为什么自定义的 Model 要继承 models.Model ?

- 不需要显式定义主键
- 自动拥有查询管理器对象
- 可以使用 ORM API 对数据库、表实现 CRUD

```
# 作品名称和作者(主演)
class Name(models.Model):
    # id 自动创建
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
    stars = models.CharField(max_length=5)
```

Model

```
# site-packages/django/db/models/base.py

class Model(metaclass=ModelBase):

        ... ...


# site-packages/django/db/models/base.py

class ModelBase(type):
    """Metaclass for all models."""
    def __new__(cls, name, bases, attrs, **kwargs):
        super_new = super().__new__
```

查询管理器

```
def books_short(request):
    ###  从 models 取数据传给 template  ###
    shorts = T1.objects.all()
```

- 如何让查询管理器的名称不叫做 objects?
- 如何利用 Manager(objects) 实现对 Model 的 CRUD?
- 为什么查询管理器返回 QuerySet 对象?

```
# site-packages/django/db/models/manager.py
class Manager(BaseManager.from_queryset(QuerySet)):
    pass
```

Manager 继承自 BaseManagerFromQuerySet 类，拥有 QuerySet 的大部分方法，
get、create、filter 等方法都来自 QuerySet

查询管理器

```
# site-packages/django/db/models/manager.py
class Manager(BaseManager.from_queryset(QuerySet)):
    pass

class BaseManager:
    @classmethod
    def from_queryset(cls, queryset_class, class_name=None):
        if class_name is None:
            # class_name = BaseManagerFromQuerySet

        return type(class_name, (cls,), {
            '_queryset_class': queryset_class,
            **cls._get_queryset_methods(queryset_class),
        })

    # 增加了很多方法给Manager
    @classmethod
    def _get_queryset_methods(cls, queryset_class):
```

模板引擎

- 模版引擎怎样通过 render() 加载 HTML 文件?
- 模版引擎怎样对模版进行渲染?

```
def books_short(request):
    return render(request, 'result.html', locals())


# site-packages/django/shortcuts.py
def render(request, template_name, context=None, content_type=None,
status=None,
    content = loader.render_to_string(template_name, context, request,
using=using)
    return HttpResponse(content, content_type, status)
```

模板引擎

```
def render_to_string(template_name, context=None, request=None, using=None):

    if isinstance(template_name, (list, tuple)):
       ...
    else:
        template = get_template(template_name, using=using)
    return template.render(context, request)

# get_template使用了_engine_list方法获得后端模板
def _engine_list(using=None):
    # 该方法返回Template文件列表，
    # engines是一个EngineHandler类的实例
    return engines.all()
```

模板引擎

```
class EngineHandler:
    @cached_property
    def templates(self):
            self._templates = settings.TEMPLATES
            # 遍历模板后端配置
            for tpl in self._templates:
                tpl = {
              'NAME': default_name,
              'DIRS': [],
              'APP_DIRS': False,
              'OPTIONS': {},
              **tpl,
            }
        templates[tpl['NAME']] = tpl
        backend_names.append(tpl['NAME'])
        return templates
```

加载模板文件

```
# site-packages/django/template/loader.py
def get_template(template_name, using=None):
    ...
    # engine定义在初始化函数中，是Engine类的实例
    # Engine类在 site-packages/django/template/engine.py 文件中
    return engine.get_template(template_name)


#  site-packages/django/template/engine.py
class Engine:
    def get_template(self, template_name):
        template, origin = self.find_template(template_name)
        if not hasattr(template, 'render'):
            # template needs to be compiled
            template = Template(template, origin, template_name, engine=self)
            return template
```

加载模板文件

```
def find_template(self, name, dirs=None, skip=None):
    tried = []
    for loader in self.template_loaders:
        try:
            template = loader.get_template(name, skip=skip)
            return template, template.origin
            except TemplateDoesNotExist as e:
                tried.extend(e.tried)
        raise TemplateDoesNotExist(name, tried=tried)
```

加载模板文件

通过 get_template() 获得 template 对象

注意：

1. get_template 的实现来自 FilesystemLoader 的父类，找到 contents 对象并构造了 Template 对象进行返回。
2. get_template_loaders() 增加了一个列表：
   ['django.template.loaders.filesystem.Loader','django.template.loaders.app_directories.Loader’]
   并把这个列表里的元素实例化成了 Loader 对象的实例化实现底层文件的加载。

加载模板文件

```
# site-packages/django/template/backends/base.py

class BaseEngine
    @cached_property
    def template_dirs(self):
        template_dirs = tuple(self.dirs)
        if self.app_dirs:
            template_dirs += get_app_template_dirs(self.app_dirname)
        return template_dirs

# site-packages/django/template/utils.py
@functools.lru_cache()
def get_app_template_dirs(dirname):
    template_dirs = [
        str(Path(app_config.path) / dirname)
        for app_config in apps.get_app_configs()
                    if app_config.path and (Path(app_config.path) / dirname).is_dir()
    ]
    return tuple(template_dirs)
```

模板渲染

```
# site-packages/django/template/backends/django.py

class Template:
  def __init__(self, template, backend):

  def render(self, context=None, request=None):
      return self.template.render(context)

# 调用了 site-packages/django/template/base.py
class Template:
  def __init__():
            # source存储的是模版文件中的内容
      self.source = str(template_string)  # May be lazy

   def render(self, context):
       return self._render(context)

   def _render(self, context):
       return self.nodelist.render(context)
```

模板渲染

如果是 Node 类型，则会调用 render_annotated 方法获取渲染结果，否则直接将元素本身作为结果，继续跟踪 bit = node.render_annotated(context)。

```
# Node类的两个子类
class TextNode(Node):
    def render(self, context):
        # 返回对象(字符串)本身
        return self.s

class VariableNode(Node):
    def render(self, context):
        try:
            # 使用resolve()解析后返回
            output = self.filter_expression.resolve(context)
```

模板渲染

```
class FilterExpression:
  def resolve(self, context, ignore_failures=False):

# 如何解析引用了类class Lexer:
class Lexer:
  def tokenize(self):
      # split分割匹配的子串并返回列表
      # tag_re是正则表达式模式对象
      for bit in tag_re.split(self.template_string):
      ... ...
      return result
   # 定义四种token类型
  def create_token(self, token_string, position, lineno, in_tag):
```

模板渲染

```
# 定义四种token类型
def create_token(self, token_string, position, lineno, in_tag):
    if in_tag and not self.verbatim:
     # 1 变量类型，开头为{{
     if token_string.startswith(VARIABLE_TAG_START):
       return Token(TokenType.VAR, token_string[2:-2].strip(), position, lineno)
     # 2 块类型，开头为{%
     elif token_string.startswith(BLOCK_TAG_START):
       if block_content[:9] in ('verbatim', 'verbatim '):
         self.verbatim = 'end%s' % block_content
         return Token(TokenType.BLOCK, block_content, position, lineno)
     # 3 注释类型，开头为{#
     elif token_string.startswith(COMMENT_TAG_START):
       content = ''
       if token_string.find(TRANSLATOR_COMMENT_MARK):
         content = token_string[2:-2].strip()
         return Token(TokenType.COMMENT, content, position, lineno)
    else:
     # 0 文本类型， 字符串字面值
     return Token(TokenType.TEXT, token_string, position, lineno)
```

DjangoWeb相关功能

1. 管理页面
2. 表单与Auth
3. 信号
4. 中间件

Django管理页面

管理页面的设计哲学：

- 管理后台是一项缺乏创造性和乏味的工作，Django 全自动地根据模型创建后台界面。
- 管理界面不是为了网站的访问者，而是为管理者准备的。

创建管理员账号：

```
$ python manage.py createsuperuser
```

Django管理页面

增加模型：

./index/admin.py

```
from .models import Type, Name
# 注册模型
admin.site.register(Type)
admin.site.register(Name)
```

表单

```
<form action="result.html" method="post">
    username:<input type="text" name="username" /><br>
    password:<input type="password" name="password" /> <br>
    <input type="submit" value="登录">
</form>
```

表单

使用Form对象定义表单

```
# form.py
from django import forms
class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput, min_length=6)
```

表单

```
<form action="/login2" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Login">
</form>
```

表单与内部auth功能结合

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('tom', 'tom@tom.com', 'tompassword')
>>> user.save()
>>> from django.contrib.auth import authenticate
>>> user = authenticate(username='tom', password='tompassword’)
```

auth功能

```
def login2(request):
    if request.method == 'POST':
     login_form = LoginForm(request.POST)
     if login_form.is_valid():
      # 读取表单的返回值
      cd = login_form.cleaned_data
      user = authenticate(username=cd['username'], password=cd['password'])
      if user:
       # 登陆用户
       login(request, user)
       return HttpResponse('登录成功')
      else:
        return HttpResponse('登录失败')
```

信号

信号：

- 发生事件，通知应用程序
- 支持若干信号发送者通知一组接收者
- 解耦

内建信号有哪些？

https://docs.djangoproject.com/zh-hans/2.2/ref/signals/

信号

信号怎么用?

函数方式注册回调函数

```
from django.core.signals import request_started
request_started.connect(my_callback1)
```

装饰器方式注册回调函数

```
from django.core.signals import request_finished
from django.dispatch import receiver
@receiver(request_finished)
def my_callback2(sender, **kwargs):
    pass
```

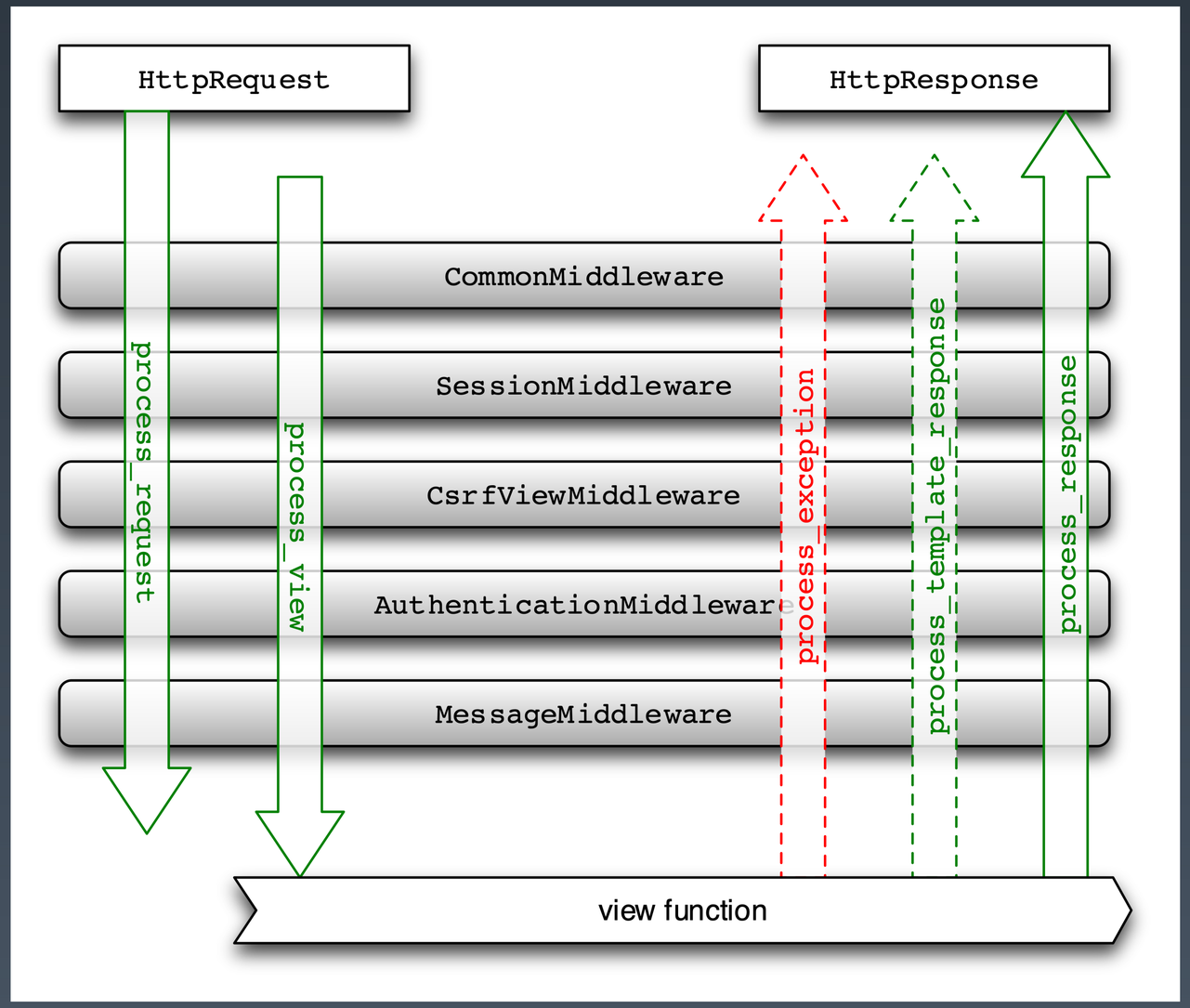
中间件

Django中间件是什么?

全局改变输入或输出

轻量级的、低级的“插件”系统

对请求、响应处理的钩子框架



中间件

```
from django.http import HttpResponse
from django.utils.deprecation import MiddlewareMixin
class Middle1(MiddlewareMixin):
    def process_request(self,request):
        print('中间件请求')
    def process_view(self, request, callback, callback_args, callback_kwargs):
        print('中间件视图')
    def process_exception(self, request, exception):
        print('中间件异常')
    def process_response(self, request, response):
        print('中间件响应')
        return response
```

Django的其他功能

1. 生产环境部署
2. 定时任务

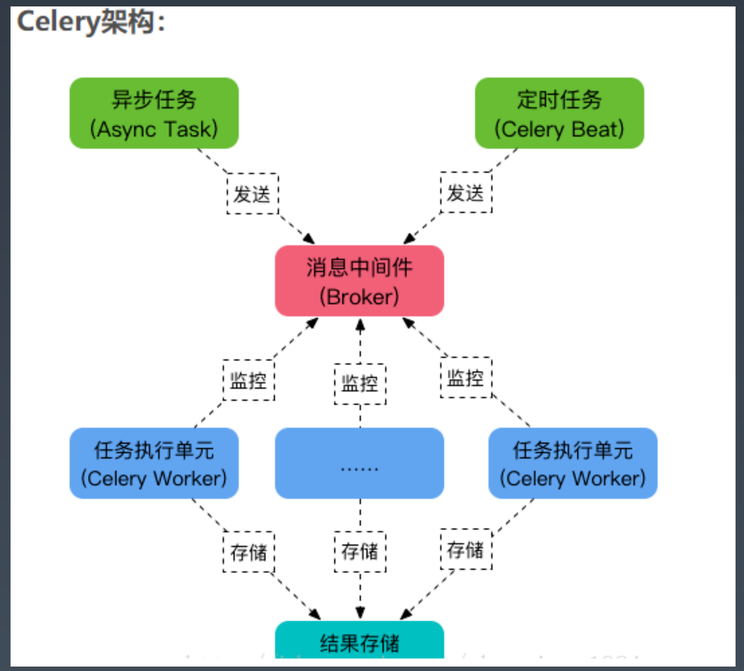
gunicorn

```
# 安装gunicorn
pip install gunicorn

# 在项目目录执行
gunicorn MyDjango.wsgi
```

Celery

- Celery 是分布式消息队列
- 使用 Celery 实现定时任务



Celery

1. Redis 安装和启动

```
redis-server /path/to/redis.conf
```

2. 安装 Celery

```
pip install celery
pip install redis==2.10.6
pip install celery-with-redis
pip install django-celery
```

Celery

3. 添加app

```
django-admin startproject MyDjango
python manager.py startapp djcron

INSTALL_APPS=[
'djcelery',
'djcron'
]
```

Celery

4. 迁移生成表

```
python manage.py migrate
```

5. 配置django时区

```
from celery.schedules import crontab
from celery.schedules import timedelta
import djcelery
djcelery.setup_loader()
BROKER_URL = 'redis://:123456@127.0.0.1:6379/'  # 代理人
CELERY_IMPORTS = ('djcron.tasks')  # app
CELERY_TIMEZONE = 'Asia/Shanghai' # 时区
CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler' # 定时任务调度器
```

Celery

6. 在 MyDjango 下建立 celery.py

```
import os
from celery import Celery, platforms
from django.conf import settings
os.environ.setdefault('DJANGO_SETTINGS_MODULE','MyDjango.settings')
app = Celery('MyDjango')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
platforms.C_FORCE_ROOT = True
```

Celery

在 __init__.py 增加

```
# 使用绝对引入，后续使用import引入会忽略当前目录下的包
from __future__ import absolute_import
from .celery import app as celery_app
```

Celery

```
from MyDjango.celery import app


@app.task()
def task1():
    return 'test1'


@app.task()
def task2():
    return 'test2'
```

Celery

启动 Celery

celery -A MyDjango beat -l info

celery -A MyDjango worker -l info

通过 admin 增加定时任务

Flask

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

```
$ export FLASK_APP=hello.py
$ flask run
```

Flask的上下文与信号

上下文：request 上下文与 session 上下文

信号：Flask 从 0.6 开始，通过 Blinker 提供了信号支持

pip install blinker

Tornado

Tornado 的同步 IO 与异步 IO：

- http_client = HTTPClient()
- http_client = AsyncHTTPClient()

Tornado 路由映射

路由映射

```
application = tornado.web.Application([
    (r"/", MainHandler),
])
```

Tornado 上下文

```
import tornado.ioloop

ioloop = tornado.ioloop.IOLoop.instance()

def callback():

    print('callback')

def async_task():

    ioloop.add_callback(callback=callback)
```

THANKS!