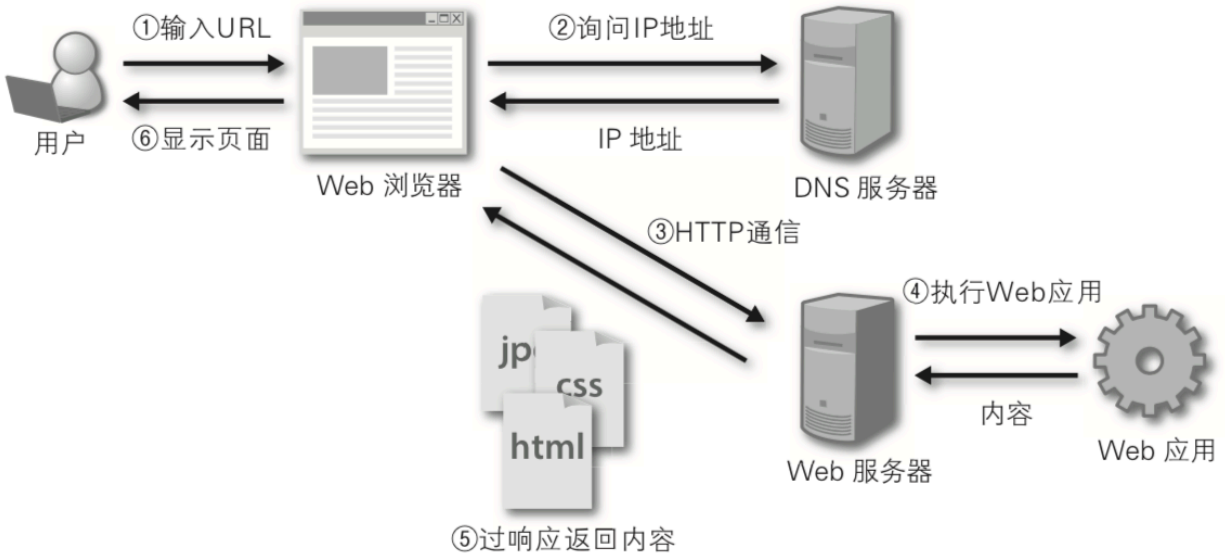
Django知识点概述

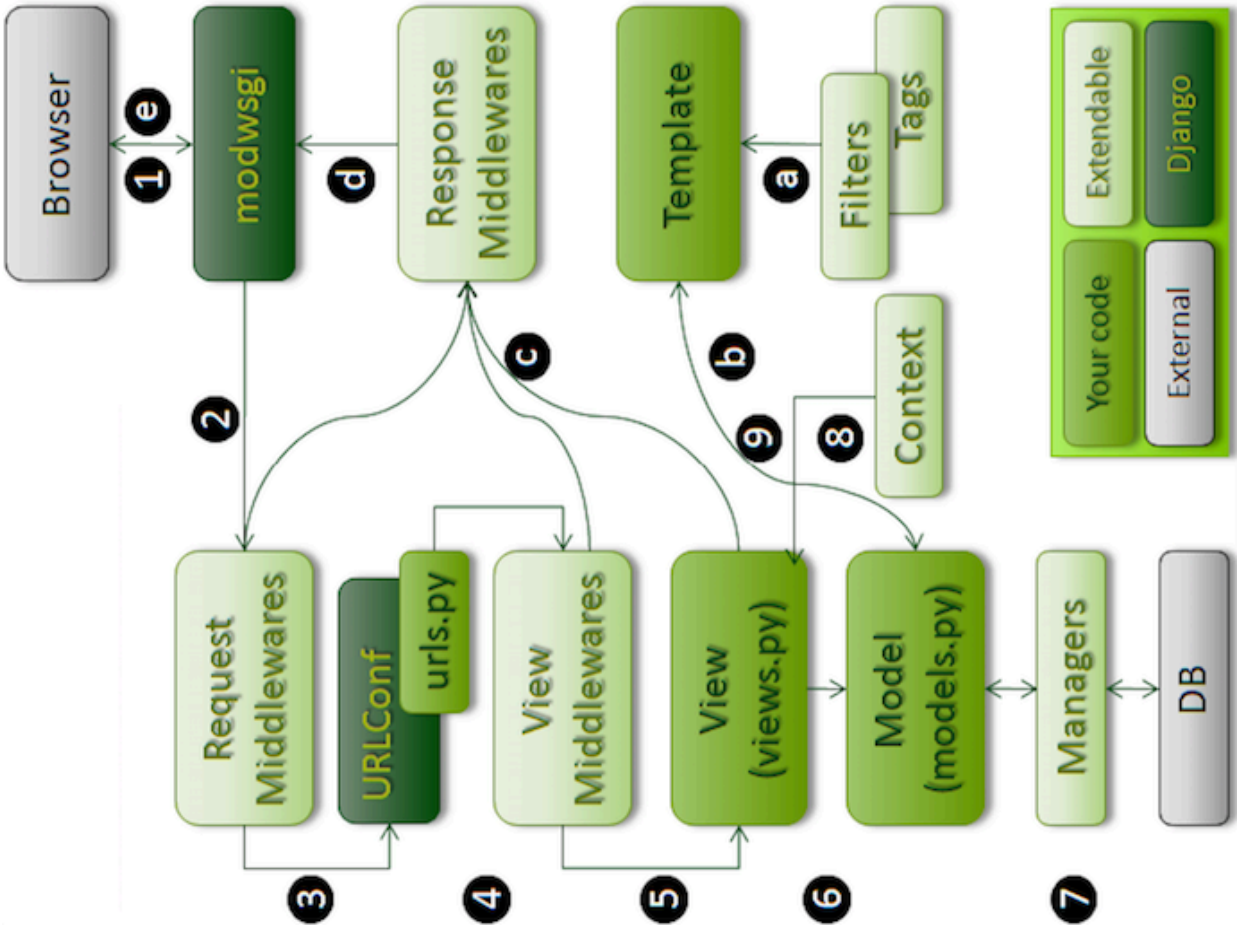
Web应用



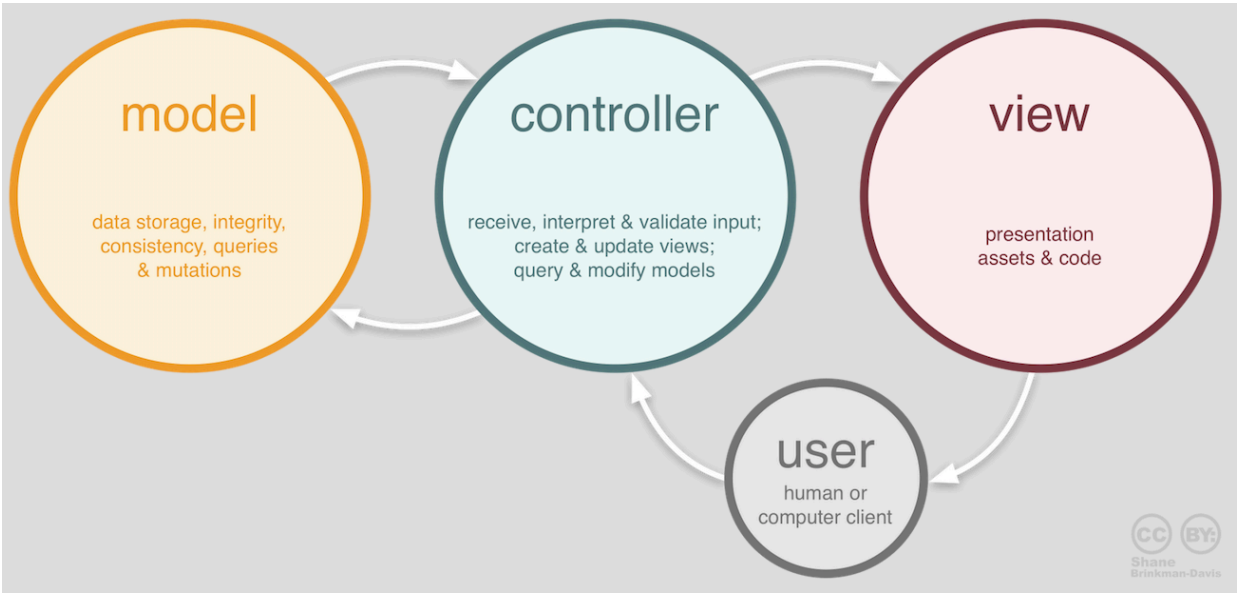
问题1：描述一个Web应用的工作流程。（如上图所示）

问题2：描述项目的物理架构。（上图中补充反向代理服务器、负载均衡服务器、数据库服务器、文件服务器、缓存服务器、防火墙等，每个节点都有可能是多节点构成的集群）

问题3：描述Django项目的工作流程。（如下图所示）



MVC架构模式



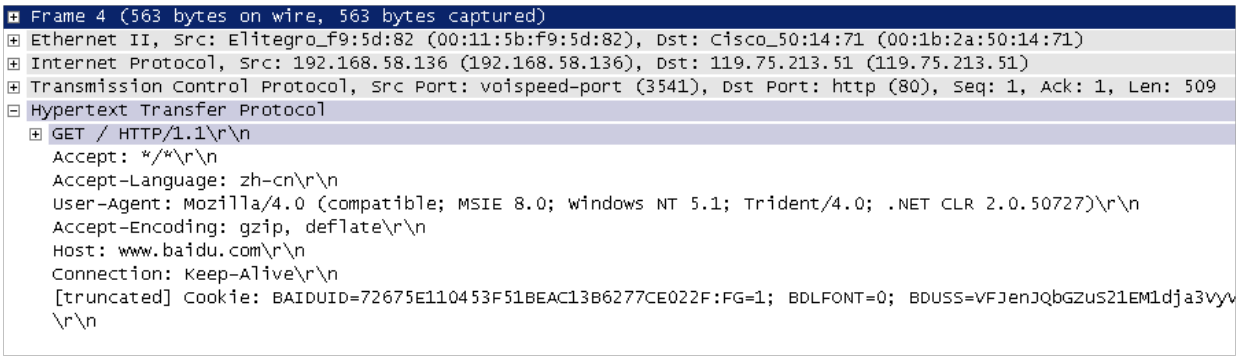
问题1：为什么要使用MVC架构模式？（模型和视图解耦合）

问题2：MVC架构中每个部分的作用？（如上图所示）

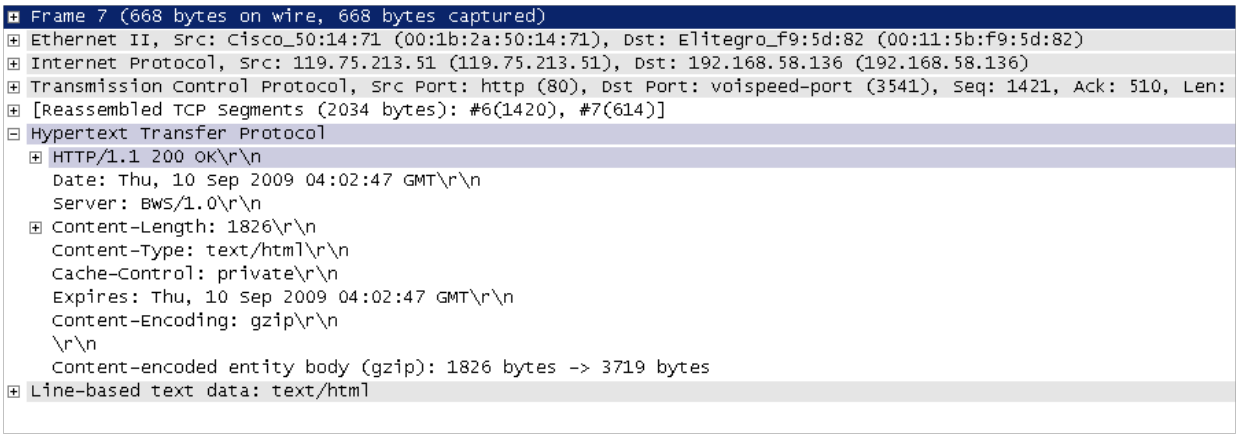
HTTP请求和响应

HTTP请求

HTTP请求 = 请求行+请求头+空行+[消息体]



HTTP响应 = 响应行+响应头+空行+消息体



1. HTTPRequest对象的属性和方法：
    - method
    - path / get_full_path()
    - scheme / is_secure() / get_host() / get_port()
    - META / COOKIES
    - GET / POST / FILES
    - get_signed_cookie()
    - is_ajax()
    - body / content_type / encoding
2. 中间件添加的属性：
    - session / user / site
3. HttpResponse对象的属性和方法：
    - set_cookie() / set_signed_cookie() / delete_cookie()
    - __setitem__ / __getitem__ / __delitem__
    - charset / content / status_code
4. JsonResponse（HttpResponse的子类型）对象

```
class HouseJsonEncoder(JsonEncoder):

    def default(self, o):
        # 定义如何将对象转成dict类型并返回这个字典
        pass
```

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content

>>> response = JsonResponse([1, 2, 3], safe=False)
>>> response = JsonResponse(house, encoder=HouseJsonEncoder)

>>> response = HttpResponse('')
>>> response['cotent-type'] = 'application/pdf';
>>> response['content-disposition'] = 'inline;
filename="xyz.pdf"'
>>> response['content-disposition'] = 'attachment;
filename="xyz.pdf"'
>>> response.set_signed_cookie('', '', salt='')
>>> response.status_code = 200
```

数据模型(Model)

问题1：关系型数据库表的设计应该注意哪些问题？（范式理论）

问题2：关系型数据库中数据完整性指的是什么？（实体完整性/参照完整性/域完整性）

问题3：ORM是什么以及解决了什么问题？（对象模型-关系模型双向转换）

1. Field及其子类的属性：
    - 通用选项：
        - db_column / db_tablespace
        - null / blank / default
        - primary_key
        - db_index / unqiue
        - choices / help_text / error_message / editable / hidden
    - 其他选项：
        - CharField: max_length
        - DateField: auto_now / auto_now_add
        - DecimalField: max_digits / decimal_places
        - FileField: storage / upload_to
        - ImageField: height_field / width_field
2. ForeignKey的属性：

- 重要属性：
  - db_constraint（提升性能或者数据分片的情况可能需要设置为False）
  - on_delete
    - CASCADE：级联删除。
    - PROTECT：抛出ProtectedError异常，阻止删除引用的对象。
    - SET_NULL：把外键设置为null，当null属性被设置为True时才能这么做。
    - SET_DEFAULT：把外键设置为默认值，提供了默认值才能这么做。
  - related_name

```
class Dept(models.Model):
    pass


class Emp(models.Model):
    dept = models.ForeignKey(related_name='+', ...)


Dept.objects.get(no=10).emp_set.all()
Emp.objects.filter(dept__no=10)
```

- 其他属性：
  - to_field / limit_choices_to / swappable

3. Model的属性和方法
   - objects/ pk
   - save() / delete()
   - from_db() / get_XXX_display() / clean() / full_clean()
4. QuerySet的方法
   - get() / all() / values()
   - count() / order_by() / exists() / reverse()
   - filter() / exclude()
     - exact / iexact：精确匹配/忽略大小写的精确匹配查询
     - contains / icontains / startswith / istartswith / endswith / iendswith：基于like的模糊查询
     - in：集合运算
     - gt / gte / lt / lte：大于/大于等于/小于/小于等于关系运算
     - range：指定范围查询（SQL中的between…and…）

- year / month / day / week_day / hour / minute / second：查询时间日期
- isnull：查询空值（True）或非空值（False）
- search：基于全文索引的全文检索
- regex / iregex：基于正则表达式的模糊匹配查询
- aggregate() / annotate()
- Avg / Count / Sum / Max / Min

```
>>> from django.db.models import Avg
>>> Emp.objects.aggregate(avg_sal=Avg('sal'))
(0.001) SELECT AVG(`TbEmp`.`sal`) AS `avg_sal` FROM
`TbEmp`; args=()
{'avg_sal': 3521.4286}
```

```
>>>
Emp.objects.values('dept').annotate(total=Count('dept
'))
(0.001) SELECT `TbEmp`.`dno`, COUNT(`TbEmp`.`dno`) AS
`total` FROM `TbEmp` GROUP BY `TbEmp`.`dno` ORDER BY
NULL LIMIT 21; args=()
<QuerySet [{'dept': 10, 'total': 4}, {'dept': 20,
'total': 7}, {'dept': 30, 'total': 3}]
```

- first() / last()
- only() / defer()

```
>>> Emp.objects.filter(pk=7800).only('name', 'sal')
(0.001) SELECT `TbEmp`.`empno`, `TbEmp`.`ename`,
`TbEmp`.`sal` FROM `TbEmp` WHERE `TbEmp`.`empno` = 7800
LIMIT 21; args=(7800,)
<QuerySet [<Emp: Emp object (7800)>]>
>>> Emp.objects.filter(pk=7800).defer('name', 'sal')
(0.001) SELECT `TbEmp`.`empno`, `TbEmp`.`job`,
`TbEmp`.`mgr`, `TbEmp`.`comm`, `TbEmp`.`dno` FROM `TbEmp`
WHERE `TbEmp`.`empno` = 7800 LIMIT 21; args=(7800,)
<QuerySet [<Emp: Emp object (7800)>]>
```

- create() / update() / raw()

```
>>> Emp.objects.filter(dept__no=20).update(sal=F('sal') +
100)
(0.011) UPDATE `TbEmp` SET `sal` = (`TbEmp`.`sal` + 100)
WHERE `TbEmp`.`dno` = 20; args=(100, 20)
>>>
>>> Emp.objects.raw('select empno, ename, job from TbEmp
where dno=10')
<RawQuerySet: select empno, ename, job from TbEmp where
dno=10>
```

5. Q对象和F对象

```
>>> from django.db.models import Q
>>> Emp.objects.filter(
...     Q(name__startswith='张'),
...     Q(sal__lte=5000) | Q(comm__gte=1000)
... ) # 查询名字以“张”开头且工资小于等于5000或补贴大于等于1000的员工
<QuerySet [<Emp: 张三丰>]>
```

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

6. 原生SQL查询

```
from django.db import connection

with connection.cursor() as cursor:
    cursor.execute("UPDATE TbEmp SET sal=sal+10 WHERE dno=30")
    cursor.execute("SELECT ename, job FROM TbEmp WHERE dno=10")
    row = cursor.fetchall()
```

7. 模型管理器

```
class BookManager(models.Manager):

    def title_count(self, keyword):
        return self.filter(title__icontains=keyword).count()
```

视图函数(Controller)

如何设计视图函数

1. 用户的每个操作对应一个视图函数。
2. 每个视图函数构成一个事务边界。
    - 事务的概念。
    - 事务的ACID特性。
    - 事务隔离级别。

      Read Uncommitted < Read Committed < Repeatable Read < Serializable

      ```
      set global transaction isolation level repeatable read;
      set session transaction isolation level read committed;


      select @@tx_isolation;
      ```

    - Django中的事务控制。
        - 给每个请求绑定事务环境（反模式）。

          ```
          ATOMIC_REQUESTS = True
          ```

        - 使用事务装饰器。

          ```
          @transaction.non_atomic_requests
          @transaction.atomic
          ```

        - 使用上下文语法。

          ```
          with transaction.atomic():
              pass
          ```

        - 关闭自动提交。

          ```
          AUTOCOMMIT = False
          ```

          ```
          transaction.commit()
          transaction.rollback()
          ```

URL配置

1. 可以让部分URL只在调试模式下生效。

```
from django.conf import settings

urlpatterns = [
    ...
]

if settings.DEBUG:
    urlpatterns += [ ... ]
```

2. 可以使用命名捕获组捕获路径参数。

```
url(r'api/code/(?P<mobile>1[3-9]\d{9})'),
path('api/code/<str:mobile>'),
```

3. URL配置不关心请求使用的方法（一个视图函数可以处理不同的请求方式）。
4. 如果使用`url`函数捕获的路径参数都是字符串，`path`函数可以指定路径参数类型。
5. 可以使用`include`函数引入其他URL配置，捕获的参数会向下传递。
6. 在`url`和`path`函数甚至是`include`函数中都可以用字典向视图传入额外的参数，如果参数与捕获的参数同名，则使用字典中的参数。
7. 可以用`reverse`函数实现URL的逆向解析（从名字解析出URL），在模板中也可以用`{% url %}`实现同样的操作。

```
path('', views.index, name='index')

return redirect(reverse('index'))
return redirect('index')
```

模板(View)

后端渲染

1. 模板的配置和渲染函数。

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

```
resp = render(request, 'foo.html', {'foo': 'bar'})
```

2. 模板遇到变量名的查找顺序。
    - 字典查找（如：`foo['bar']`）
    - 属性查找（如：`foo.bar`）
    - 方法调用（如：`foo.bar()`）
        - 方法不能有必须传值的参数
        - 在模板中不能够给方法传参
        - 如果方法的`alters_data`被设置为`True`则不能调用该方法（避免误操作的风险），模型对象动态生成的`delete()`和`save()`方法都设定了`alters_data = True`。
    - 列表索引查找（如：`foo[0]`）
3. 模板标签的使用。
    - `{% if %}` / `{% else %}` / `{% endif %}`
    - `{% for %}` / `{% endfor %}`
    - `{% ifequal %}` / `{% endifequal %}` / `{% ifnotequal %}` / `{% endifnotequal %}`
    - `{# comment #}` / `{% comment %}` / `{% endcomment %}`
4. 过滤器的使用。
    - `lower` / `upper` / `first` / `last` / `truncatewords` / `date`/ `time` / `length` / `pluralize` / `center` / `ljust` / `rjust` / `cut` / `urlencode` / `default_if_none` / `filesizeformat` / `join` / `slice` / `slugify`
5. 模板的包含和继承。
    - `{% include %}` / `{% block %}`
    - `{% extends %}`
6. 模板加载器（后面优化部分会讲到）。
    - 文件系统加载器

```
TEMPLATES = [{
    'BACKEND':
'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
}]
```

- 应用目录加载器

```
TEMPLATES = [{
    'BACKEND':
'django.template.backends.django.DjangoTemplates',
    'APP_DIRS': True,
}]
```

前端渲染

1. 前端模板引擎：Handlebars / Mustache。
2. 前端MV*框架。
    - MVC - AngularJS
    - MVVM - Vue.js

其他视图

1. MIME类型。

Content-Type	说明
application/json	JSON（JavaScript Object Notation）
application/pdf	PDF（Portable Document Format）
audio/mpeg	MP3或其他MPEG音频文件
audio/vnd.wave	WAV音频文件
image/gif	GIF图像文件
image/jpeg	JPEG图像文件
image/png	PNG图像文件
text/html	HTML文件

text/xml	XML
video/mp4	MP4视频文件
video/quicktime	QuickTime视频文件

2. 如何处置生成的内容。

```
response['content-type'] = 'application/pdf'
response['content-disposition'] = 'attachment;
filename="xyz.pdf"'
```

> *提醒：URL以及请求和响应中的中文都应该处理成百分号编码。*

3. 生成CSV / Excel / PDF。
   - 向浏览器传输二进制数据。

```
buffer = ByteIO()

resp = HttpResponse(content_type='...')
resp['Content-Disposition'] = 'attachment;filename="..."'
resp.write(buffer.getvalue())
```

   - 大文件的流式处理：`StreamingHttpResponse`。

```
class EchoBuffer(object):

    def write(self, value):
        return value


def some_streaming_csv_view(request):
    rows = (["Row {}".format(idx), str(idx)] for idx in
range(65536))
    writer = csv.writer(EchoBuffer())
    resp = StreamingHttpResponse((writer.writerow(row) for
row in rows),
                                 content_type="text/csv")
    resp['Content-Disposition'] = 'attachment;
filename="data.csv"'
    return resp
```

- 生成PDF：需要安装`reportlab`。
- 生成Excel：需要安装`openpyxl`。

4. 统计图表。
    - ECharts或Chart.js。
    - 思路：后端只提供JSON格式的数据，前端JavaScript渲染生成图表。

中间件

问题1：中间件背后的设计理念是什么？（分离横切关注功能/拦截过滤器模式）

问题2：中间件有哪些不同的实现方式？（参考下面的代码）

问题4：描述Django内置的中间件及其执行顺序。

推荐阅读：Django官方文档 - 中间件 - 中间件顺序。

激活中间件

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'common.middlewares.block_sms_middleware',
]
```

自定义中间件

```
def simple_middleware(get_response):

    def middleware(request):

        response = get_response(request)

        return response

    return middleware
```

```
class MyMiddleware(object):

    def __init__(self, get_response):
```

```
        self.get_response = get_response

    def __call__(self, request):

        response = self.get_response(request)

        return response

    def process_view(self, request, view_func, view_args,
view_kwargs):
        response = view_func(*view_args, **view_kwargs)
        return response
```

```
class MyMiddleware(object):

    def __init__(self):
        pass

    def process_request(request):
        pass

    def process_view(request, view_func, view_args, view_kwargs):
        pass

    def process_template_response(request, response):
        pass

    def process_response(request, response):
        pass

    def process_exception(request, exception):
        pass
```

内置中间件

1. CommonMiddleware
    - DISALLOWED_USER_AGENTS
    - APPEND_SLASH
    - USE_ETAG
2. SecurityMiddleware
    - SECURE_HSTS_SECONDS
    - SECURE_HSTS_INCLUDE_SUBDOMAINS
    - SECURE_CONTENT_TYPE_NOSNIFF
    - SECURE_BROWSER_XSS_FILTER
    - SECURE_SSL_REDIRECT
    - SECURE_REDIRECT_EXEMPT

3. SessionMiddleware
4. CsrfViewMiddleware - 防范跨栈身份伪造。
5. XFrameOptionsMiddleware - 防范点击劫持攻击

类	说明
UpdateCacheMiddleware	在修改 Vary 首部的中间件（SessionMiddleware、GZipMiddleware、LocaleMiddleware）前面。
GZipMiddleware	在任何可能修改或使用响应主体的中间件前面。在 UpdateCacheMiddleware 后面，因为要修改 Vary 首部。
ConditionalGetMiddleware	在 CommonMiddleware 前面，因为设定 USE_ETAGS = True 时，要使用 Etag 首部。
SessionMiddleware	在 UpdateCacheMiddleware 后面，因为要修改 Vary 首部。
LocaleMiddleware	在前部，SessionMiddleware（使用会话数据）和 CacheMiddleware（修改 Vary 首部）后面。
CommonMiddleware	在任何可能修改首部的中间件前面（要计算 Etag）。在 GZipMiddleware 后面，因为不为压缩的内容计算 Etag。靠近顶部，因为 APPEND_SLASH 或 PREPEND_WWW 设为 True 时要重定向。
CsrfViewMiddleware	在任何假定已经防范了 CSRF 攻击的视图中间件前面。
AuthenticationMiddleware	在 SessionMiddleware 后面，因为要使用会话存储器。
MessageMiddleware	在 SessionMiddleware 后面，这样才能使用基于会话的存储器。
FetchFromCacheMiddleware	在任何修改 Vary 首部的中间件后面，因为创建缓存哈希键时要从那个首部中选一个值。
FlatpageFallbackMiddleware	应该放在靠近底部的位置，做最后一搏。
RedirectFallbackMiddleware	应该放在靠近底部的位置，做最后一搏。

表单

1. 用法：通常不要用来生成页面上的表单控件（耦合度太高不容易定制），主要用来验证数据。
2. Form的属性和方法：
    - `is_valid()` / `is_multipart()`
    - `errors` / `fields` / `is_bound` / `changed_data` / `cleaned_data`
    - `add_error()` / `has_errors()` / `non_field_errors()`
    - `clean()`
3. Form.errors的方法：
    - `as_data()` / `as_json()` / `get_json_data()`

问题1：Django中的`Form`和`ModelForm`有什么作用？（通常不用来生成表单主要用来验证数据）

问题2：表单上传文件时应该注意哪些问题？（表单的设置、多文件上传、图片预览、Ajax上传文件、上传后的文件如何存储）

Cookie和Session

问题1：使用Cookie能解决什么问题？（用户跟踪，解决HTTP协议无状态问题）

1. URL重写
2. 隐藏域（隐式表单域）
3. Cookie

问题2：Cookie和Session之间关系是什么？（Session的标识会通过Cookie记录）

Session的配置

1. Session对应的中间件：`django.contrib.sessions.middleware.SessionMiddleware`。
2. Session引擎。
    - 基于数据库（默认方式）

    ```
    INSTALLED_APPS = [
        'django.contrib.sessions',
    ]
    ```

    - 基于缓存（推荐使用）

    ```
    SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
    SESSION_CACHE_ALIAS = 'default'
    ```

    - 基于文件（基本不考虑）
    - 基于Cookie（不靠谱）

    ```
    SESSION_ENGINE =
    'django.contrib.sessions.backends.signed_cookies'
    ```

3. Cookie相关的配置。

```
SESSION_COOKIE_NAME = 'djang_session_id'
SESSION_COOKIE_AGE = 1209600
SESSION_EXPIRE_AT_BROWSER_CLOSE = False
SESSION_SAVE_EVERY_REQUEST = False
SESSION_COOKIE_HTTPONLY = True
```

4. session的属性和方法。
    - `session_key` / `session_data` / `expire_date`
    - `__getitem__` / `__setitem__` / `__delitem__` / `__contains__`
    - `set_expiry()` / `get_expiry_age()` / `get_expiry_date()`
    - `flush()`
    - `set_test_cookie()` / `test_cookie_worked()` / `delete_test_cookie()`
5. session的序列化。

```
SESSION_SERIALIZER =
django.contrib.sessions.serializers.JSONSerializer
```

    - JSONSerializer（默认）- 如果想将自定义的对象放到session中，会遇到“Object of type 'XXX' is not JSON serializable”的问题。
    - PickleSerializer（1.6以前的默认，但是因为安全问题不推荐使用，但是只要不去反序列化用户构造的恶意的Payload就行了。关于这种方式的安全漏洞，请参考《Python Pickle的任意代码执行漏洞实践和Payload构造》。）

缓存

配置缓存

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': [
            'redis://120.77.222.217:6379/0',
            'redis://120.77.222.217:6380/0',
            'redis://120.77.222.217:6381/0',
            # 'redis://120.77.222.217:6382/0',
        ],
        # 'KEY_PREFIX': 'fang',
        # 'TIMEOUT': 120,
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
            'CONNECTION_POOL_KWARGS': {
                'max_connections': 100,
            },
            'PASSWORD': '1qaz2wsx',
            # 'COMPRESSOR':
'django_redis.compressors.zlib.ZlibCompressor'
        }
    },
}
```

全站缓存

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

CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 300
CACHE_MIDDLEWARE_KEY_PREFIX = 'djang:cache'
```

视图层缓存

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


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

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

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```

其他内容

1. 模板片段缓存。
    - {% load cache %}
    - {% cache %} / {% endcache %}
2. 使用底层API访问缓存。

```
>>> from django.core.cache import cache
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
>>> cache.clear()
```

```
>>> from django.core.cache import caches
>>> cache1 = caches['myalias']
>>> cache2 = caches['myalias']
>>> cache1 is cache2
True
```

日志

日志级别

NOTSET < DEBUG < INFO < WARNING < ERROR < FETAL

日志配置

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'simple': {
            'format': '%(asctime)s %(module)s.%(funcName)s: %
(message)s',
            'datefmt': '%Y-%m-%d %H:%M:%S',
        },
        'verbose': {
            'format': '%(asctime)s %(levelname)s [%(process)d-%
(threadName)s] '
                      '%(module)s.%(funcName)s line %(lineno)d: %
(message)s',
            'datefmt': '%Y-%m-%d %H:%M:%S',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'DEBUG',
        },
        'inf': {
            'class': 'logging.handlers.TimedRotatingFileHandler',
            'filename': 'info.log',
            'when': 'W0',
            'backupCount': 12,
            'formatter': 'simple',
            'level': 'INFO',
        },
        'err': {
            'class': 'logging.handlers.TimedRotatingFileHandler',
```

```
            'filename': 'error.log',
            'when': 'D',
            'backupCount': 31,
            'formatter': 'verbose',
            'level': 'WARNING',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
        'inform': {
            'handlers': ['inf'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'error': {
            'handlers': ['err'],
            'level': 'DEBUG',
            'propagate': True,
        }
    }
}
```

日志配置官方示例。

日志分析

1. Linux相关命令：head、tail、grep、awk、uniq、sort

   ```
   tail -10000 access.log | awk '{print $1}' | uniq -c | sort -r
   ```

2. 实时日志文件分析：Python + 正则表达式 + Crontab
3. 《Python日志分析工具》。
4. 《集中式日志系统ELK》。
   - ElasticSearch
   - Logstash
   - Kibana

RESTful

问题1：RESTful架构到底解决了什么问题？（URL具有自描述性、资源表述与视图的解耦和、互操作性利用构建微服务以及集成第三方系统、无状态性提高水平扩展能力）

问题2：项目在使用RESTful架构时有没有遇到一些问题或隐患？（对资源访问的限制、资源从属关系检查、避免泄露业务信息、防范可能的攻击）

> *补充：下面的几个和安全性相关的响应头在前面讲中间件的时候提到过的。*
>
> - *X-Frame-Options: DENY*
> - *X-Content-Type-Options: nosniff*
> - *X-XSS-Protection: 1; mode=block;*
> - *Strict-Transport-Security: max-age=31536000;*

问题3：如何保护API中的敏感信息以及防范重放攻击？（摘要和令牌）

推荐阅读：《如何有效防止API的重放攻击》。

修改配置文件

```
INSTALLED_APPS = [

    'rest_framework',

]
```

编写序列化器

```
from rest_framework import serializers


class ProvinceSerializer(serializers.ModelSerializer):

    class Meta:
        model = Province
        fields = ('prov_id', 'prov_name')
```

最怂的做法

```
@csrf_exempt
def list_provinces(request):
    if request.method == 'GET':
        serializer = ProvinceSerializer(Province.objects.all(),
many=True)
    elif request.method = 'POST':
        serializer = ProvinceSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
    return HttpResponse(json.dumps(serializer.data),
                        content_type="application/json; charset=utf-
8")
```

使用装饰器

```
@api_view(['GET', 'POST'])
def list_provinces(request):
    if request.method == 'GET':
        serializer = ProvinceSerializer(Province.objects.all(),
many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = ProvinceSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET'])
def list_cities_by_prov(request, prov_id):
    serializer =
CitySerializer(City.objects.filter(prov__prov_id=prov_id),
many=True)
    return Response(serializer.data)
```

```
urlpatterns = [
    path('provinces/', views.list_provinces),
    path('provinces/<int:prov_id>', views.list_cities_by_prov),
]
```

问题1：如何让浏览器能够发起DELETE/PUT/PATCH?

```
<form method="post">

    <input type="hidden" name="_method" value="delete">

</form>
```

```
if request.method == 'POST' and '_method' in request.POST:
    request.method = request.POST['_method'].upper()
```

```
<script>
    $.ajax({
        'url': '/api/provinces',
        'type': 'put',
        'data': {},
        'dataType': 'json',
        'success': function(json) {
            // Web = 标签(内容) + CSS(显示) + JS(行为)
            // JavaScript = ES + BOM + DOM
            // DOM操作实现页面的局部刷新
        },
        'error': function() {}
    });
    $.getJSON('/api/provinces', function(json) {
        // DOM操作实现页面的局部刷新
    });
</script>
```

问题2：如何解决多个JavaScript库之间某个定义（如$函数）冲突的问题?

```
<script src="js/jquery.js"></script>
<script src="js/abc.js"></script>
<script>
    // $已经被后加载的JavaScript库占用了
    // 但是可以直接用绑定在window对象上的jQuery去代替$
    jQuery(function() {
        jQuery('#okBtn').on('click', function() {});
    });
</script>
```

```
<script src="js/abc.min.js"></script>
<script src="js/jquery.min.js"></script>
<script>
    // 将$让出给其他的JavaScript库使用
    jQuery.noConflict();
    jQuery(function() {
        jQuery('#okBtn').on('click', function() {});
    });
</script>
```

问题3：jQuery对象与原生DOM对象之间如何转换?

```
<button id="okBtn">点我</button>
<script src="js/jquery.min.js"></script>
<script>
    var btn = document.getElementById('okBtn'); // 原生JavaScript对象
    // $(btn) --> jQuery --> 拥有更多的属性和方法而且没有浏览器兼容性问题
    var $btn = $('#okBtn'); // jQuery对象
    // $btn[0] / $btn.get(0) --> JavaScript --> 自己处理浏览器兼容性问题
    $btn.on('click', function() {});
</script>
```

使用类视图

更好的复用代码，不要重“复发明轮子”。

```
class CityView(APIView):

    def get(self, request, pk, format=None):
        try:
            serializer = CitySerializer(City.objects.get(pk=pk))
            return Response(serializer.data)
        except City.DoesNotExist:
            raise Http404

    def put(self, request, pk, format=None):
        try:
            city = City.objects.get(pk=pk)
            serializer = CitySerializer(city, data=request.data)
            if serializer.is_valid():
                serializer.save()
                return Response(serializer.data)
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
        except City.DoesNotExist:
```

```
            raise Http404

    def delete(self, request, pk, format=None):
        try:
            city = City.objects.get(pk=pk)
            city.delete()
            return Response(status=status.HTTP_204_NO_CONTENT)
        except City.DoesNotExist:
            raise Http404
```

```
urlpatterns = [
    path('cities/<int:pk>', views.CityView.as_view()),
]
```

使用ViewSet

```
class DistrictViewSet(viewsets.ReadOnlyModelViewSet):

    queryset = District.objects.all()
    serializer_class = DistrictSerializer
```

```
class DistrictViewSet(viewsets.ModelViewSet):

    queryset = District.objects.all()
    serializer_class = DistrictSerializer
```

```
router = routers.DefaultRouter()
router.register('districts', views.DistrictViewSet)

urlpatterns += router.urls
```

认证用户身份

1. 利用Django自带的User。
2. 自行对用户及其身份验证的摘要进行处理。

```
sha1_proto = hashlib.sha1()


def check_user_sign(get_response):
```

```
    def middleware(request):
        if request.path.startswith('/api'):
            data = request.GET if request.method == 'GET' else
request.data
            try:
                user_id = data['user_id']
                user_token = cache.get(f'fang:user:{user_id}')
                user_sign = data['user_sign']
                hasher = sha1_proto.copy()
                hasher.update(f'{user_id}:
{user_token}'.encode('utf-8'))
                if hasher.hexdigest() == user_sign:
                    return get_response(request)
            except KeyError:
                pass
            return JsonResponse({'msg': '身份验证失败拒绝访问'})
        else:
            return get_response(request)

    return middleware
```

3. 请求的时效性问题。（请求中再加上一个随机的令牌）

其他问题

问题1：如何设计一套权限系统？（RBAC或ACL）

1. RBAC - 基于角色的访问控制（用户-角色-权限，都是多对多关系）。
2. ACL - 访问控制列表（每个用户绑定自己的访问白名单）。

问题2：如何实现异步任务和定时任务？（Celery）

问题3：如何解决JavaScript跨域获取数据的问题？（django-cors-headers）

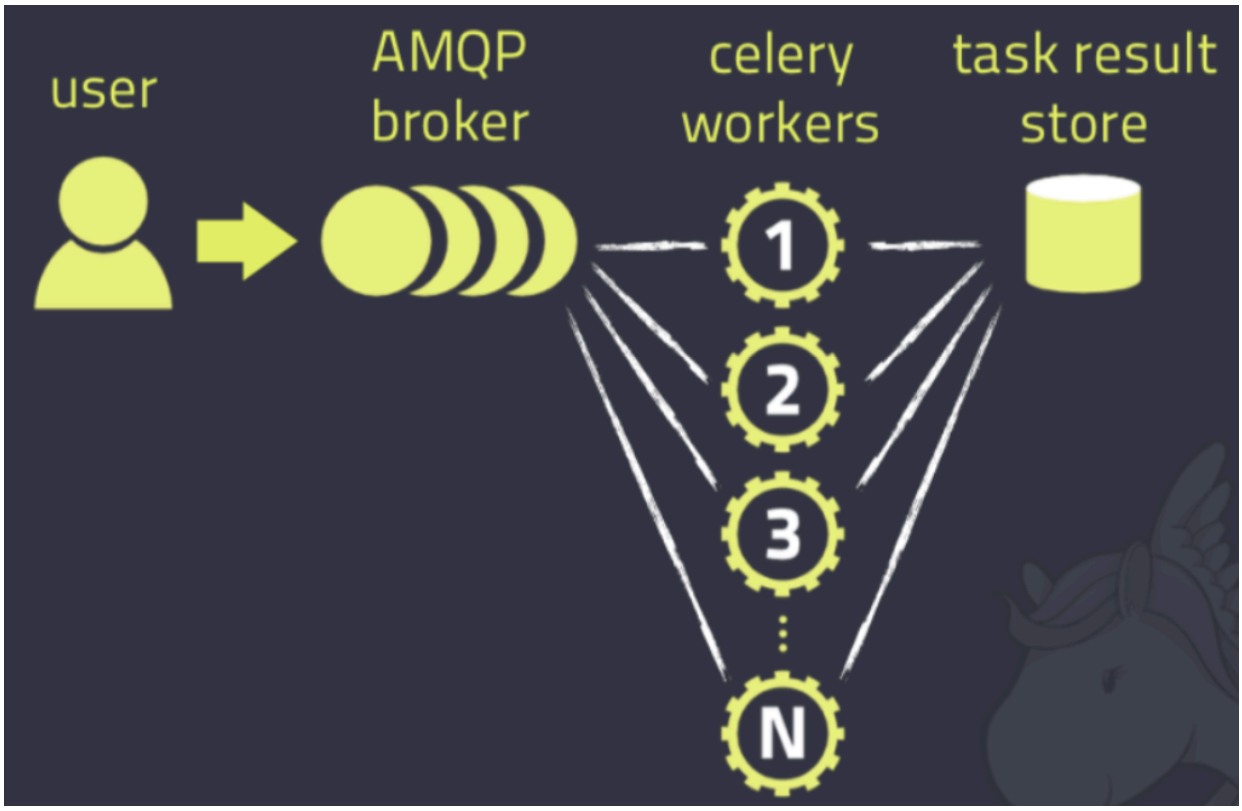
问题4：网站图片（水印、剪裁）和视频（截图、水印、转码）是如何处理的？（云存储、FFmpeg）

问题5：网站如何架设（静态资源）文件系统？

Celery的应用

Celery 是一个简单、灵活且可靠的，处理大量消息的分布式系统，并且提供维护这样一个系统的必需工具。它是一个专注于实时处理的任务队列，同时也支持任务调度。

推荐阅读：《Celery官方文档中文版》，上面有极为详细的配置和使用指南。



Celery是一个本身不提供队列服务，官方推荐使用RabbitMQ或Redis来实现消息队列服务，前者是更好的选择，它对AMQP（高级消息队列协议）做出了非常好的实现。

1. 安装RabbitMQ。

```
docker pull rabbitmq
docker run -d -p 5672:5672 --name myrabbit rabbitmq
docker container exec -it myrabbit /bin/bash
```

2. 创建用户、资源以及分配操作权限。

```
rabbitmqctl add_user jackfrued 123456
rabbitmqctl set_user_tags jackfrued administrator
rabbitmqctl add_vhost myvhost
rabbitmqctl set_permissions -p myvhost jackfrued ".*" ".*" ".*"
```

3. 创建Celery实例。

```
project_name = 'fang'
project_settings = '%s.settings' % project_name

# 注册环境变量
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
project_settings)
```

```
app = celery.Celery(
    project_name,

 backend='amqp://jackfrued:123456@120.77.222.217:5672/myvhost',

 broker='amqp://jackfrued:123456@120.77.222.217:5672/myvhost'
)

# 从默认的配置文件读取配置信息
app.config_from_object('django.conf:settings')

# Celery加载所有注册的应用
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

4. 启动Celery创建woker。

```
celery -A fang worker -l info &
```

5. 执行异步任务。

```
@app.task
def send_email(from, to, cc, subject, content):
    pass


async_result = send_email.delay('', [], [], '', '')
async_result.get()
```

6. 创建定时任务。

```
from celery.schedules import crontab
from celery.task import periodic_task


@periodic_task(run_every=crontab('*', '12,18'))
def print_dummy_info():
    print('你妈喊你回家吃饭啦')
```

7. 检查定时任务并交给worker执行。

```
celery -A fang beat -l info
```

8. 检查消息队列状况。

```
rabbitmqctl list_queues -p myvhost
```

安全保护

问题1：什么是跨站脚本攻击，如何防范？（对提交的内容进行消毒）

问题2：什么是跨站身份伪造，如何防范？（使用随机令牌）

问题3：什么是SQL注射攻击，如何防范？（不拼接SQL语句，避免使用单引号）

问题4：什么是点击劫持攻击，如何防范？（不允许<iframe>加载非同源站点内容）

Django提供的安全措施

签名数据的API

```
    >>> from django.core.signing import Signer
    >>> signer = Signer()
    >>> value = signer.sign('hello, world!')
    >>> value
    'hello, world!:BYMlgvWMTSPLxC-DqxByleiMVXU'
    >>> signer.unsign(value)
    'hello, world!'
    >>>
    >>> signer = Signer(salt='1qaz2wsx')
    >>> signer.sign('hello, world!')
    'hello, world!:9vEvG6EA05hjMDB5MtUr33nRA_M'
    >>>
    >>> from django.core.signing import TimestampSigner
    >>> signer = TimestampSigner()
    >>> value = signer.sign('hello, world!')
    >>> value
    'hello, world!:1fpmcQ:STwj464IFE6eUB-_-hyUVF3d2So'
    >>> signer.unsign(value, max_age=5)
    Traceback (most recent call last):
      File "<console>", line 1, in <module>
      File "/Users/Hao/Desktop/fang.com/venv/lib/python3.6/site-
packages/django/core/signing.py", line 198, in unsign
        'Signature age %s > %s seconds' % (age, max_age))
    django.core.signing.SignatureExpired: Signature age
21.020604848861694 > 5 seconds
    >>> signer.unsign(value, max_age=120)
    'hello, world!'
```

5. CSRF令牌和小工具

```
{% csrf_token %}
```

```
@csrf_exempt
@csrf_protect
@require_csrf_token
@ensure_csrf_cookie
```

用户敏感信息的保护

1. 哈希摘要（签名）

```
>>> import hashlib
>>>
>>> md5_proto = hashlib.md5()
>>> md5_hasher = md5_proto.copy()
>>> md5_hasher.update('hello, world!'.encode())
>>> md5_hasher.hexdigest()
'3adbbad1791fbae3ec908894c4963870'
>>>
>>> sha1_proto = hashlib.sha1()
>>> sha1_hasher = sha1_proto.copy()
>>> sha1_hasher.update('hello, world!'.encode())
>>> sha1_hasher.hexdigest()
'1f09d30c707d53f3d16c530dd73d70a6ce7596a9'
```

2. 加密和解密（对称加密和非对称加密）

```
>>> pub_key, pri_key = rsa.newkeys(1024)
>>> message = 'hello, world!'
>>> crypto = rsa.encrypt(message.encode(), pub_key)
>>> crypto
b'Ou{gH\xa9\xa8}O\xe3\x1d\x052|M\x9d9?
\xdc\xd8\xecF\xd3v\x9b\xde\x8e\x12\xe6M\xebvx\x08\x08\x8b\xe8\x
86~\xe4^)w\xf2\xef\x9e\x9fOg\x15Q\xb7\x7f\x1d\xcfV\xf1\r\xbe^+\
x8a\xbf
}\x10\x01\xa4U9b\x97\xf5\xe0\x90T\'\xd4(\x9b\x00\xa5\x92\x17\xa
d4\xb0\xb0"\xd4\x16\x94*s\xe1r\xb7L\xe2\x98\xb7\x7f\x03\xd9\xf2
\t\xee*\xe6\x93\xe6\xe1o\xfd\x18\x83L\x0cfL\xff\xe4\xdd%\xf2\xc
0/\xfb'
>>> origin = rsa.decrypt(crypto, pri_key).decode()
>>> origin
'hello, world!'
```

安全相关建议

1. 虽然 Django 自带了稳固的安全保护措施，但是依然要采用正确的方式部署应用程序，利用 Web 服务器、操作系统和其他组件提供的安全保护措施。
2. 记得把 Python 代码放在 Web 服务器的文档根目录之外，避免代码意外泄露。
3. 谨慎处理用户上传的文件。
4. Django本身没有对请求次数加以限制（包括验证用户身份的请求），为了防止暴力攻击和破解，可以考虑使用具有一次消费性的验证码或对这类请求的次数进行限制。
5. 将缓存系统、数据库服务器以及重要的资源服务器都放在第二级防火墙之后（不要放在DMZ）。

测试相关

1. 测试为项目带来的好处有哪些？（更早的发现问题，更好的团队协作）
2. 什么是黑盒测试和白盒测试？（方法是黑盒子，不知道实现细节，通过设定输入和预期输出进行来断言方法是否实现了应有的功能）

单元测试

目标：测试函数和对象的方法（程序中最基本的单元）。

实施：通过对实际输出和预期输出的比对以及各种的断言条件来判定被测单元是否满足设计需求。

```
class UtilTest(TestCase):

    def setUp(self):
        self.pattern = re.compile(r'\d{6}')

    def test_gen_mobile_code(self):
```

```
        for _ in range(100):

 self.assertIsNotNone(self.pattern.match(gen_mobile_code()))

    def test_to_md5_hex(self):
        md5_dict = {
            '123456': 'e10adc3949ba59abbe56e057f20f883e',
            '123123123': 'f5bb0c8de146c67b44babbf4e6584cc0',
            '1qaz2wsx': '1c63129ae9db9c60c3e8aa94d3e00495',
        }
        for key, value in md5_dict.items():
            self.assertEqual(value, to_md5_hex(key))
```

测试模型

```
class ModelTest(TestCase):

    def test_save_province(self):
        pass

    def test_del_province(self):
        pass

    def test_update_province(self):
        pass

    def test_get_all_provinces(self):
        pass

    def test_get_province_by_id(self):
        pass
```

测试视图

```
class RentViewTest(TestCase):

    def test_index_view(self):
        resp = self.client.get(reverse('index'))
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['num'], 5)
```

运行测试

配置测试数据库

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'localhost',
        'PORT': 3306,
        'NAME': 'House',
        'USER': os.environ['DB_USER'],
        'PASSWORD': os.environ['DB_PASS'],
        'TEST': {
            'NAME': 'House_for_testing',
            'CHARSET': 'utf8',
        },
    }
}
```

```
python manage.py test
python manage.py test common
python manage.py test common.tests.UtilsTest
python manage.py test common.tests.UtilsTest.test_to_md5_hex
```

测试覆盖度

```
pip install coverage
coverage run --source=<path1> --omit=<path2> manage.py test common
coverage report

Name                                Stmts   Miss  Cover
-------------------------------------------------------
common/__init__.py                      0      0   100%
common/admin.py                         1      0   100%
common/apps.py                          3      3     0%
common/forms.py                        16     16     0%
common/helper.py                       32     32     0%
common/middlewares.py                  19     19     0%
common/migrations/__init__.py           0      0   100%
common/models.py                       71      2    97%
common/serializers.py                  14     14     0%
common/tests.py                        14      8    43%
common/urls_api.py                      3      3     0%
common/urls_user.py                     3      3     0%
common/utils.py                        22      7    68%
common/views.py                        69     69     0%
-------------------------------------------------------
TOTAL                                 267    176    34%
```

性能测试

问题1：性能测试的指标有哪些？（）

1. ab

```
ab -c 10 -n 1000 http://www.baidu.com/
...
Benchmarking www.baidu.com (be patient).....done
Server Software:        BWS/1.1
Server Hostname:        www.baidu.com
Server Port:            80
Document Path:          /
Document Length:        118005 bytes
Concurrency Level:      10
Time taken for tests:   0.397 seconds
Complete requests:      100
Failed requests:        98
   (Connect: 0, Receive: 0, Length: 98, Exceptions: 0)
Write errors:           0
Total transferred:      11918306 bytes
HTML transferred:       11823480 bytes
Requests per second:    252.05 [#/sec] (mean)
Time per request:       39.675 [ms] (mean)
Time per request:       3.967 [ms] (mean, across all concurrent
requests)
Transfer rate:          29335.93 [Kbytes/sec] received
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        6    7   0.6      7       9
Processing:    20   27  22.7     24     250
Waiting:        8   11  21.7      9     226
Total:         26   34  22.8     32     258
Percentage of the requests served within a certain time (ms)
  50%     32
  66%     34
  75%     34
  80%     34
  90%     36
  95%     39
  98%     51
  99%    258
 100%    258 (longest request)
```

2. mysqlslap

```
mysqlslap -a -c 100 -h 1.2.3.4 -u root -p
mysqlslap -a -c 100 --number-of-queries=1000 --auto-generate-sql-load-type=read -h <负载均衡服务器IP地址> -u root -p
mysqlslap -a --concurrency=50,100 --number-of-queries=1000 --debug-info --auto-generate-sql-load-type=mixed -h 1.2.3.4 -u root -p
```

3. sysbench

```
sysbench --test=threads --num-threads=64 --thread-yields=100 --thread-locks=2 run
sysbench --test=memory --num-threads=512 --memory-block-size=256M --memory-total-size=32G run
```

4. jmeter

   请查看《使用JMeter进行性能测试》。

5. LoadRunner

部署相关

请参考《Django项目上线指南》。

性能相关

网站优化两大定律：

1. 尽可能的使用缓存 - 牺牲空间换取时间（普适策略）。
2. 能推迟的都推迟 - 使用消息队列将并行任务串行来缓解服务器压力。
   - 服务器CPU利用率出现瞬时峰值 - 削峰（CPU利用率平缓的增长）
   - 上下游节点解耦合（下订单和受理订单的系统通常是分离的）

Django框架

1. 配置缓存来缓解数据库的压力，并有合理的机制应对缓存穿透和缓存雪崩。
2. 开启模板缓存来加速模板的渲染。

```
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',
            ],
            'loaders': [(
                'django.template.loaders.cached.Loader', [

 'django.template.loaders.filesystem.Loader',

 'django.template.loaders.app_directories.Loader',
                ], ),
            ],
        },
    },
]
```

3. 用惰性求值、迭代器、defer()、only()等缓解内存压力。
4. 用select_related()和prefetch_related()执行预加载避免“1+N查询问题”。

数据库

1. 用ID生成器代替自增主键（性能更好、适用于分布式环境）。

```
>>> my_uuid = uuid.uuid1()
>>> my_uuid
UUID('63f859d0-a03a-11e8-b0ad-60f81da8d840')
>>> my_uuid.hex
'63f859d0a03a11e8b0ad60f81da8d840'
```

2. 避免不必要的外键列上的约束（除非必须保证参照完整性），更不要使用触发器之类的机制。
3. 使用索引来优化查询性能（索引放在要用于查询的字段上）。

```
select * from tb_goods where gname like 'iPhone%';
```

```
Goods.objects.filter(name__istartswith='iPhone');
```

4. 使用存储过程（存储在服务器端编译过的一组SQL语句）。

```
drop procedure if exists sp_avg_sal_by_dept;

create procedure sp_avg_sal_by_dept(deptno integer, out avg_sal
float)
begin
    select avg(sal) into avg_sal from TbEmp where dno=deptno;
end;

call sp_avg_sal_by_dept(10, @a);

select @a;
```

```
>>> from django.db import connection
>>> cursor = connection.cursor()
>>> cursor.callproc('sp_avg_sal_by_dept', (10, 0))
>>> cursor.execute('select @_sp_avg_sal_by_dept_1')
>>> cursor.fetchone()
(2675.0,)
```

5. 使用explain来分析查询性能 - 执行计划。

```
explain select * from ...;
```

请参考网易出品的《深入浅出MySQL》上面对应部分的讲解（已经有第二版）。

6. 使用慢查询日志来发现性能低下的查询。

```
mysql> show variables like 'slow_query%';
+---------------------------+---------------------------------
+
| Variable_name             | Value
|
+---------------------------+---------------------------------
+
| slow_query_log            | OFF
|
| slow_query_log_file       | /mysql/data/localhost-slow.log
|
+---------------------------+---------------------------------
+

mysql> show variables like 'long_query_time';
+-----------------+-----------+
| Variable_name   | Value     |
```

```
+-----------------+-----------+
| long_query_time | 10.000000 |
+-----------------+-----------+
```

```
mysql> set global slow_query_log='ON';
mysql> set global
slow_query_log_file='/usr/local/mysql/data/slow.log';
mysql> set global long_query_time=1;
```

```
[mysqld]
slow_query_log = ON
slow_query_log_file = /usr/local/mysql/data/slow.log
long_query_time = 1
```

其他

请参考《Python项目性能调优》。