视图函数

视图函数（Function-based View），即视图功能由函数实现。

JSON响应

```
from django.http import HttpRequest, HttpResponse, JsonResponse

def test_index(request:HttpRequest):
    data = [1, 2, 3]
    return JsonResponse(data)
```

抛出异常`TypeError: In order to allow non-dict objects to be serialized set the safe parameter to False.`。意思是，safe参数为False才可使用非字典数据，所以，除非有必要，否则还是使用字典

```
from django.http import HttpRequest, HttpResponse, JsonResponse

def test_index(request:HttpRequest):
    data = {'a':100, 'b':'abc'}
    return JsonResponse(data)
```

请求方法限制装饰器

如果需要对请求方法限制，例如只允许GET方法请求怎么办？当然可以自己判断，也可以使用Django提供的装饰器函数。

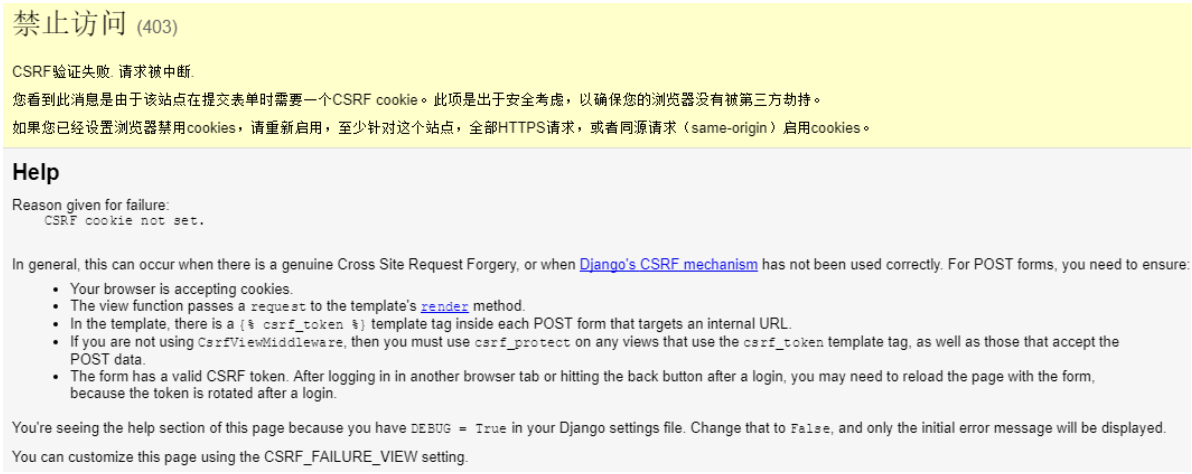
```
from django.http import HttpRequest, HttpResponse, JsonResponse
from django.views.decorators.http import require_http_methods, require_GET, require_POST

# @require_http_methods(['GET', 'POST'])
# @require_GET
@require_POST
def test_index(request:HttpRequest):
    data = {'a':100, 'b':'abc'}
    return JsonResponse(data)
```

测试过程中，当使用不被允许的方法请求时，返回405状态码，表示`Method Not Allowed`

装饰完后，test_index就是新的视图函数，装饰器内部的inner函数。这类似于在装饰器一章学过的logger装饰器。

CSRF处理

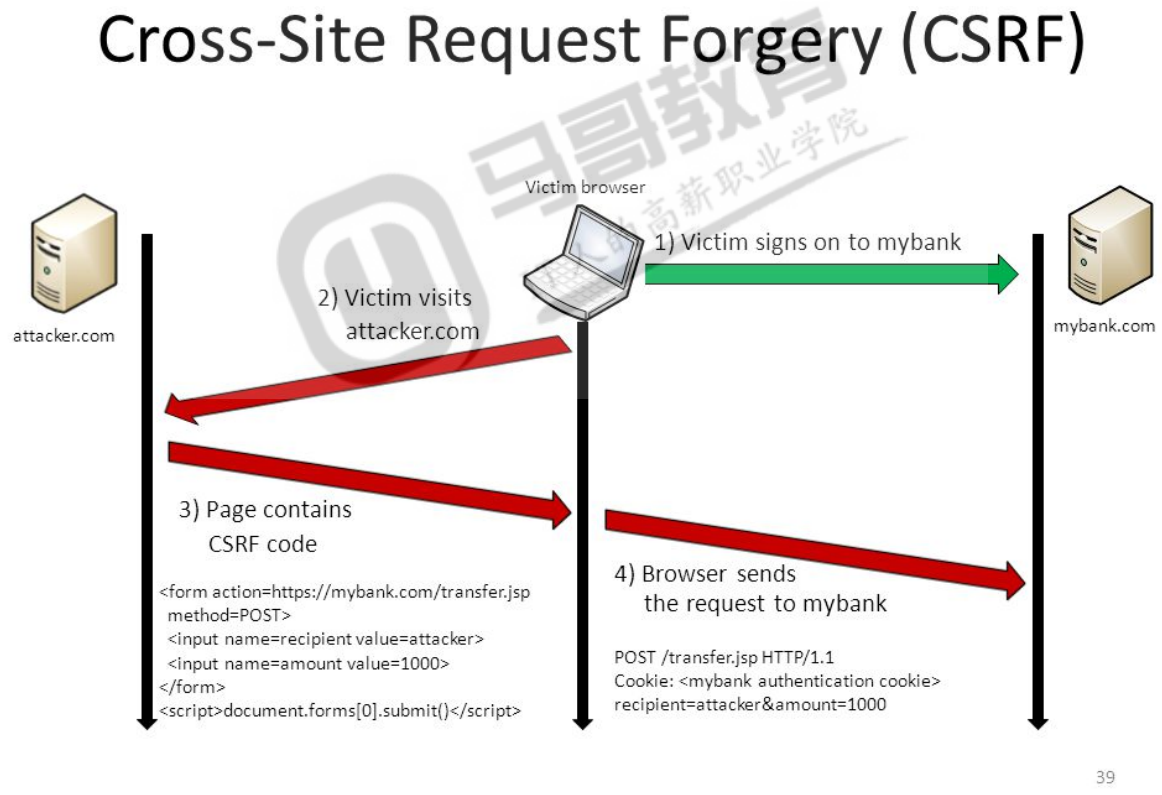
在Post数据的时候，发现出现了下面的提示



原因：**默认Django CsrfViewMiddleware中间件会对所有POST方法提交的信息做CSRF校验。**

CSRF或XSRF（Cross-site Request Forgery），即跨站请求伪造。它也被称为：one click attack/session riding，是一种对网站的恶意利用。它伪装成来自受信任用户发起请求，难以防范。

原理



1. 用户登录某网站A完成登录认证，网站返回敏感信息的Cookie，即使是会话级的Cookie
2. 用户没有关闭浏览器，或认证的Cookie一段时间内不过期还持久化了，用户就访问攻击网站B
3. 攻击网站B看似一切正常，但是某些页面里面有一些隐藏运行的代码，或者诱骗用户操作的按钮等
4. 这些代码一旦运行就是悄悄地向网站A发起特殊请求，由于网站A的Cookie还有效，且访问的是网站A，则其Cookie就可以一并发给网站A
5. 网站A看到这些Cookie就只能认为是登录用户发起的合理合法的请求，就会执行

CSRF解决

1. 关闭CSRF中间件（不推荐）

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

2. csrftoken验证

   - 在表单POST提交时，需要发给服务器一个csrf_token
   - 模板中的表单Form中增加{% csrf_token %}，它返回到了浏览器端就会为cookie增加csrftoken 字段，还会在表单中增加一个名为csrfmiddlewaretoken隐藏控件`<input type='hidden' name='csrfmiddlewaretoken' value='jZTxUOv5mPoLvugcfLbS1B6vT8COYrKuxMzodWv8oNAr3a4ouWlb5AaYG2tQi3dD' />`
   - POST提交表单数据时，需要将csrfmiddlewaretoken一并提交，Cookie中的csrf_token 也一并会提交，最终在中间件中比较，相符通过，不相符就看到上面的403提示
   - 假设正常网站为A，攻击网站为B，在访问网站B网页时，这个网页并不是来自网站A的网页，而只是在这个网页中包含着提交到网站A的请求的代码，注意只有访问网站A返回的HTML页面，才会有{% csrf_token %}产生set-cookie和input hidden。网站B的网页恶意代码执行，由于发起对网站A的请求，会带上cookie，但是没有input hidden带的值，验证失败

3. 双cookie验证

   - 访问本站先获得csrftoken的cookie
   - 如果使用AJAX进行POST，需要在每一次请求Header中增加自定义字段X-CSRFTOKEN，其值来自cookie中获取的csrftoken值
   - 在服务器端比较cookie和X-CSRFTOKEN中的csrftoken，相符通过
   - 假设正常网站为A，攻击网站为B，双Cookie验证中，用户访问攻击网站B时，网站B网页中代码悄悄发起对A的请求，由于跨域不能获得正常网站A的Cookie值，它只能发起请求时，浏览器自动带上A的Cookie，但是A检查请求头中并没有X-CSRFTOKEN的值，或这个随机token值对不上，验证失败

现在没有前端代码，为了测试方便，可以选择第一种方法先禁用中间件，测试完成后开启。

视图类

视图类（Class-based View），即视图功能由一个类和其方法实现

参考 https://docs.djangoproject.com/en/3.2/topics/class-based-views/

View类原理

> django.views.View类本质就是一个对请求方法分发到与请求方法同名函数的调度器。

```
from django.urls import path
from .views import TestIndex # 视图类

urlpatterns = [
    path('', TestIndex.as_view()), # 二级路由包含前缀emp/，对应URL是/emp/
]
```

django.views.View类，定义了http的方法的小写名称列表，这些小写名称其实就是处理请求的方法名的小写。

View类的类方法as_view()方法调用后**返回**一个内建的`view(request, *args, **kwargs)`**新函数**（为了后面叙述方便，称它为fn），本质上其实还是url映射到了这个fn函数上。注意这个fn函数的签名，就是视图函数的签名。

请求request到来后，直接发给fn函数，fn函数内部

- 构建TestIndex实例self。注意：阅读源码可以看到，每一个请求创建一个实例
- dispatch派发请求，`self.dispatch(request, *args, **kwargs)`

dispatch方法内部比对请求方法method，如果存在请求的get、post等方法，则调用，否则返回405

看到了getattr等反射函数，说明基于**反射**实现的。

> 本质上，as_view()方法还是把一个类伪装成了一个视图函数。
>
> 这个视图函数，内部使用了一个分发函数，使用请求方法名称把请求分发给存在的同名函数处理。

视图类实现

```
from django.http import HttpRequest, HttpResponse, JsonResponse
from django.views import View


class TestIndex(View):
    def get(self, request): # 支持GET
        data = {'a':100, 'b':'abc'}
        return JsonResponse(data)

    def post(self, request): # 支持POST
        data = {'a':200, 'b':'xyz'}
        return JsonResponse(data)
```

方法装饰器

由上面的原理分析，as_view()后，就可以看做是一个普通的视图函数。由此，得到方法装饰器的一种用法。

```
from django.urls import path
from .views import TestIndex # 视图类
from django.views.decorators.http import require_http_methods, require_GET

urlpatterns = [
    #path('', require_GET(TestIndex.as_view())),
    path('', require_http_methods(['POST'])(TestIndex.as_view())),
]
```

装饰器本质就是函数调用，`require_http_methods(['POST'])(TestIndex.as_view())`返回一个新的视图函数。

虽然，TestIndex有get、post方法，但是之前却要现经过require_http_methods函数检查。

中间件

洋葱模型

中间件和视图，如同洋葱一层层包裹着最中心的视图，想要见到视图函数或返回给浏览器端很不容易，需要在来路和去路都要经过这些中间件。

get_response非常重要，表示去调用下一层的对象。对象可能是下一级中间件，也可能是洋葱心儿——视图。

在阅读wsgi.py源码中，进入`django.core.handlers.wsgi.WSGIHandler`类，可以看到`__init__`中加载了中间件，在`__call__`中调用了get_response。

中间件定义

Django1.10版本开始，中间件帮助文档已经不能很好的体现其技术原理了。在官网切换到1.8版本帮助，看到下面内容 https://docs.djangoproject.com/en/1.8/topics/http/middleware/

Hooks and application order

During the request phase, before calling the view, Django applies middleware in the order it's defined in **MIDDLEWARE_CLASSES**, top-down. Two hooks are available:

- **process_request()**
- **process_view()**

During the response phase, after calling the view, middleware are applied in reverse order, from the bottom up. Three hooks are available:

- **process_exception()** (only if the view raised an exception)
- **process_template_response()** (only for template responses)
- **process_response()**

再看看目前的版本的文档 https://docs.djangoproject.com/en/3.2/topics/http/middleware/

从文档中可以看出保留process_view、process_exception、process_template_response这些钩子函数。

process_view参考 https://docs.djangoproject.com/en/3.2/topics/http/middleware/#process-view

```python
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # request请求去视图的路上
        response = self.get_response(request)
        # 视图函数调用完成返回response的路上
        return response
```

新建包utils，在里面增加一个middlewares.py

```python
from django.http import HttpResponse

class MagMiddleware1:
    def __init__(self, get_response):
        """执行一次"""
        print(self.__class__.__name__, "init~~~~")
        self.get_response = get_response
```

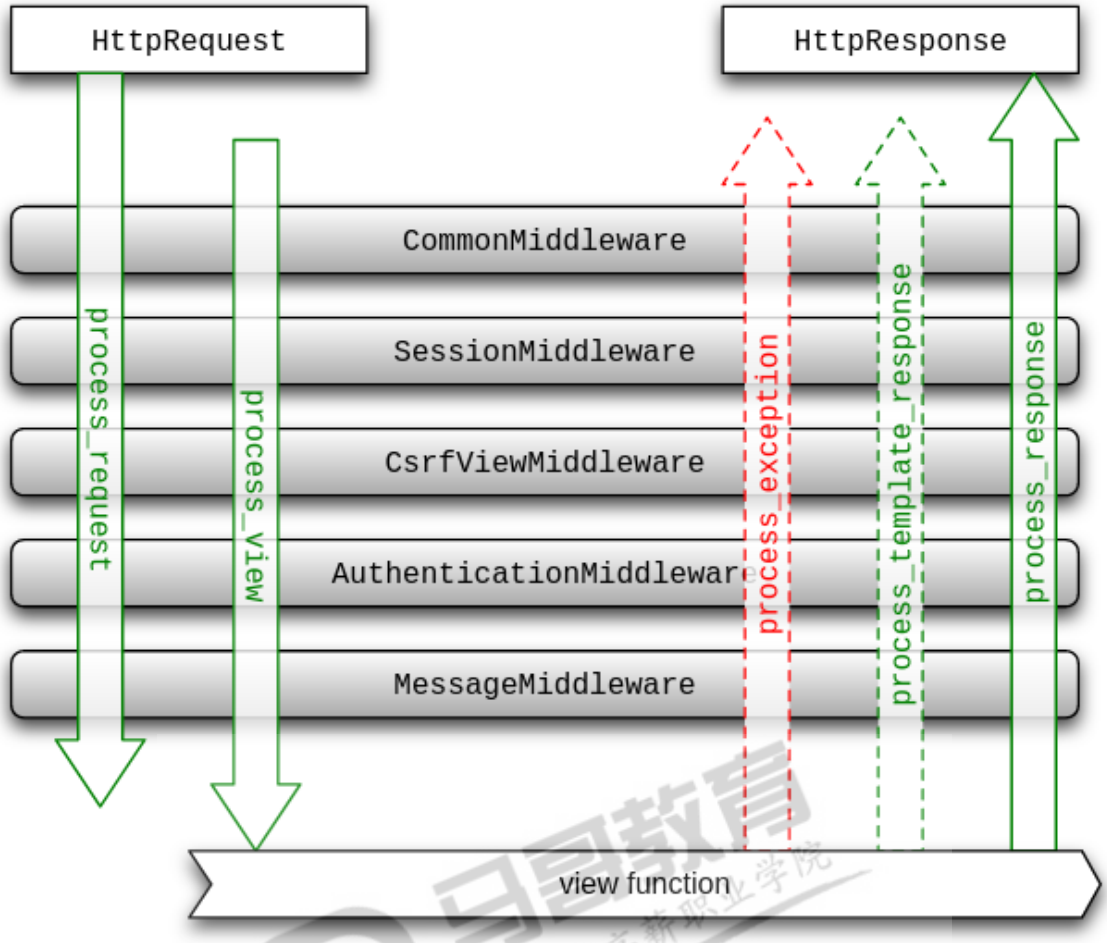
```
    def __call__(self, request):
        # request请求去视图的路上
        print(self.__class__.__name__, "__call__~~~~")
        # return HttpResponse(self.__class__.__name__) # 测试点
        response = self.get_response(request)
        # 视图函数调用完成返回response的路上
        print(self.__class__.__name__, "__call__#####")
        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        """调用视图前被调用，返回值是None或HttpResponse对象"""
        print(self.__class__.__name__, "process_view~~~~",
view_func.__name__, view_args, view_kwargs)
        # return HttpResponse(self.__class__.__name__ + ' process_view') #
测试点


class MagMiddleware2:
    def __init__(self, get_response):
        """执行一次"""
        print(self.__class__.__name__, "init~~~~")
        self.get_response = get_response

    def __call__(self, request):
        # request请求去视图的路上
        print(self.__class__.__name__, "__call__~~~~")
        # return HttpResponse(self.__class__.__name__) # 测试点
        response = self.get_response(request)
        # 视图函数调用完成返回response的路上
        print(self.__class__.__name__, "__call__#####")
        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        """调用视图前被调用，返回值是None或HttpResponse对象"""
        print(self.__class__.__name__, "process_view~~~~",
view_func.__name__, view_args, view_kwargs)
        # return HttpResponse(self.__class__.__name__ + ' process_view') #
测试点
```

定义2个中间件，注册

```
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',
    'utils.middlewares.MagMiddleware1',
    'utils.middlewares.MagMiddleware2'
]
```

将测试代码中`测试点`打开，观察效果，理解中间件原理。

原理



结论

- Django中间件使用的洋葱式，但有特殊的地方
- settings.py中配置着中间件的顺序
- 中间件初始化一次，但是初始化顺序和配置序相反，如同包粽子，先从芯开始包
- 新版中间件先在`__call__`中get_response(request)之前代码（相当于老版本中的process_request）
- 按照配置顺序先后执行所有中间件的get_response(request)之前代码
- 全部执行完解析路径映射得到view_func
- process_view函数按照配置顺序，依次向后执行
  - return None 继续向后执行
  - return HttpResponse() 就不在执行其它函数的preview函数了，此函数返回值作为浏览器端的响应
- 执行view函数，前提是前面的所有中间件process_view都返回None
- 逆序执行所有中间件的get_response(request)之后代码
- 特别注意，如果get_response(request)之前代码中return HttpResponse()，将从当前中间件立即返回给浏览器端，从洋葱中依次反弹

应用

应用场景：如果绝大多数请求或响应都需要拦截，个别例外，采用中间件较为合适。

中间件有很多用途，适合拦截所有请求和响应。例如浏览器端的IP是否禁用、UserAgent分析、异常响应的统一处理

内建中间件

- SessionMiddleware 从请求报文中提取sessionid，提供request.session属性
- AuthenticationMiddleware 依赖SessionMiddleware，提供request.user属性。根据session认证，如果成功，request.user就是可用的用户对象，is_authenticated为True；如果失败，返回一个匿名用户对象，is_authenticated为False。

Session和Cookie

浏览器端和服务器端身份认证的一种方式。简单讲，就是为了让服务端确定你是谁。

无状态，无连接

- 无连接：Http 1.1之前，都是一个请求一个连接，连接用完即刻断开，其实是**无连接。**
  - 有连接：是因为它基于TCP协议，是面向连接的，需要3次握手、4次断开。
  - 短连接：而Tcp的连接创建销毁成本高，对服务器有很大的影响。所以，自Http 1.1开始，支持keep-alive，默认也开启，一个连接打开后，会保持一段时间（可设置），浏览器再访问该服务器就使用这个Tcp连接，减轻了服务器压力，提高了效率。
- 无状态：服务器端没有记录每次客户端请求相关的任何状态数据，服务器无法确定2次请求之间的联系，即使是前后2次同一个浏览器也没有任何数据能够判断出是同一个浏览器的请求。后来可以通过cookie、session来判断。

Cookie技术

- 键值对信息
- 是一种客户端、服务器端传递数据的技术
- 一般来说cookie信息是在服务器端生成，返回给浏览器端的
- 浏览器端可以保持这些值，浏览器对同一域发起每一请求时，都会把Cookie信息发给服务器端
- 服务端收到浏览器端发过来的Cookie，处理这些信息，可以用来判断这次请求是否和之前的请求有关联

曾经Cookie唯一在浏览器端存储数据的手段，目前浏览器端存储数据的方案很多，Cookie正在被淘汰。

当服务器收到HTTP请求时，服务器可以在响应头里面添加一个Set-Cookie键值对。浏览器收到响应后通常会保存这些Cookie，之后对该服务器每一次请求中都通过Cookie请求头部将Cookie信息发送给服务器。

另外，Cookie的过期时间、域、路径、有效期、适用站点都可以根据需要来指定。
可以使用`Set－Cookie: NAME=VALUE；Expires=DATE；Path=PATH；Domain=DOMAIN_NAME；SECURE`

例如：

```
Set-Cookie:aliyungf_tc=AQAAAJDwJ3Bu8gkAHbrHb4zlNZGw4Y; Path=/; HttpOnly
set-cookie:test_cookie=CheckForPermission; expires=Tue, 19-Mar-2018 15:53:02 GMT; path=/; domain=.doubleclick.net

Set-Cookie: BD_HOME=1; path=/
```

key	value说明
Cookie过期	Cookie可以设定过期终止时间，过期后将被浏览器清除。 如果缺省，Cookie不会持久化，浏览器关闭Cookie消失，称为会话级Cookie
Cookie域	域确定有哪些域可以存取这个Cookie。 缺省设置属性值为当前主机，例如`www.magedu.com`。 如果设置为`magedu.com`表示包含子域
Path	确定哪些目录及子目录访问可以使用该Cookie
Secure	表示Cookie随着HTTPS加密过得请求发送给服务端 有些浏览器已经不允许http://协议使用Secure了 这个Secure不能保证Cookie是安全的，Cookie中不要传输敏感信息
HttpOnly	将Cookie设置此标记，就不能被JavaScript访问，只能发给服务器端

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
告诉浏览器端设置这个Cookie的键值对，有过期时间，使用HTTPS加密传输到服务器端，且不能被浏览器中JS脚本访问该Cookie
```

```
Cookie的作用域：Domain和Path定义Cookie的作用域

Domain
domain=www.magedu.com 表示只有该域的URL才能使用
domain=magedu.com 表示可以包含子域，例如www.magedu.com、python.magedu.com等

Path
path=/ 所有/的子路径可以使用
domain=www.magedu.com; path=/webapp 表示只有www.magedu.com/webapp下的URL匹配，例如http://www.magedu.com/webapp/a.html就可以
```

缺点

- Cookie一般明文传输（Secure是加密传输），安全性极差，不要传输敏感数据
- 有4kB大小限制
- 每次请求中都会发送Cookie，增加了流量

其它持久化技术

LocalStorage

- 浏览器端持久化方案之一，HTML5标准增加的技术
- 依然采用键值对存储数据
- 数据会存储在不同的域名下面
- 不同浏览器对单个域名下存储数据的长度支持不同，有的最多支持2MB。

https://developer.mozilla.org/zh-CN/docs/Web/API/Window/localStorage

SessionStorage和LocalStorage差不多，它是会话级的，浏览器关闭，会话结束，数据清除。

IndexedDB

- 一个域一个datatable
- key-valuede检索方式
- 建立在关系型的数据模型之上，具有索引表、游标、事务等概念

Session技术

WEB 服务器端，尤其是动态网页服务端Server，有时需要知道浏览器方是谁？但是HTTP是无状态的，怎么办？

服务端会为每一次浏览器端第一次访问生成一个SessionID，用来唯一标识该浏览器，通过响应报文的Set-Cookie发送到浏览器端。

```
Set-Cookie:JSESSIONID=741248A52EEB83DF182009912A4ABD86.Tomcat1; Path=/; HttpOnly
```

浏览器端收到之后并不永久保持这个Cookie，可以是会话级的。浏览器访问服务端时，会使用与请求域相关的Cookies，也会带上这个SessionID的Cookie值。

动态网页技术，也需要知道用户身份，但是HTTP是无状态协议，无法知道。必须提出一种技术，让客户端提交的信息可以表明身份。只能是服务端发出一个凭证，即SessionID，让浏览器端每次请求时发出Cookies的同时带上这个SessionID，且过期作废，浏览器还不能更改。这个技术为了给浏览器发凭证就使用了现有的Cookie技术。

服务端会维持这个SessionID一段时间，如果超时，会清理这些超时没有人访问的SessionID。如果浏览器端发来的SessionID无法在服务端找到，就会自动再次分配新的SessionID，并通过Set-Cookie发送到浏览器端以覆盖原有的存在浏览器中的会话级的SessionID。

也就是说服务器端会为浏览器端在内存开辟空间保存SessionID，同时和这个SessionID关联存储更多键值对。这种为客户端在服务端维护相关状态数据的技术，就是Session技术。

推荐图书《HTTP权威指南》

Session开启后，会为浏览器端设置一个Cookie值，即SessionID。
这个SessionID的Cookie如果是会话级的，浏览器不做持久化存储只放在内存中，并且浏览器关闭自动清除。
浏览器端发起HTTP请求后，这个SessionID会通过Cookie发到服务器端，服务器端就可以通过这个ID查到对应的一个字典结构。如果查无此ID，就为此浏览器重新生成一个SessionID，为它建立一个SessionID和空字典的映射关系。

可以在这个SessionID关联的字典中，存入键值对来保持与当前会话相关的更多信息

- Session会定期过期清除
- Session占用服务器端内存
- Session如果没有持久化，如果服务程序崩溃，那么所有Session信息丢失
- Session可以持久化到数据库中，如果服务程序崩溃，那么可以从数据库中恢复

开启session支持

Django可以使用Session

- 在settings中，MIDDLEWARE设置中，启用'django.contrib.sessions.middleware.SessionMiddleware'
- INSTALLED_APPS设置中，启用'django.contrib.sessions'。它是基于数据库存储的Session
- Session不使用，可以关闭上述配置，以减少开销
- 在数据库的表中的django_session表，记录session信息。但可以使用文件系统或其他cache来存储

session清除

登录成功，为当前session在django_session表中增加一条记录，如果没有显式调用logout函数或request.session.flush()，那么该记录不会消失。Django也没有自动清除失效记录的功能。

request.session.flush()会清除当前session，同时删除表记录。

但Django提供了一个命令clearsessions，建议放在cron中定期执行。

```
$ django-admin.py clearsessions
$ manage.py clearsessions
```