# Table of Contents

# **Flask**进阶课程课件

本课件是知了课堂《Python web全栈开发》 VIP课程的课件。请勿用于商业用途，如有转载，请注明出处！

# 学前准备

在学习 `Flask` 之前，需要做好以下准备工作：

- 确保已经安装了 `python3.6` 以上的版本，教学以 `python 3.6` 版本进行讲解。
- 通过 `pip install Flask` 安装最新版的 `Flask` 。目前为止，最新版本为 `0.12.2` 。可以通过以下方式进行验证是否安装成功：

  ```
  $ python
  >> import flask
  >> print flask.__version__
  ```

  如果显示 `0.12.2` ，则说明已经安装成功。

- 安装了 `pycharm 2016版` 或者是 `sublime text 3` 等任意一款你喜欢的编辑器（推荐使用 `pycharm` ，如果电脑性能原因，可以退而求其次用 `sublime` ）。
- 两个基本的命令:
  - `ls` ： 列出当前目录下的所有子目录以及文件。
  - `cd` ： 切换目录。

# 虚拟环境

## 为什么需要虚拟环境：

到目前位置，我们所有的第三方包安装都是直接通过 `pip install xx` 的方式进行安装的，这样安装会将那个包安装到你的系统级的 `Python` 环境中。但是这样有一个问题，就是如果你现在用 `Django 1.10.x` 写了个网站，然后你的领导跟你说，之前有一个旧项目是用 `Django 0.9` 开发的，让你来维护，但是 `Django 1.10` 不再兼容 `Django 0.9` 的一些语法了。这时候就会碰到一个问题，我如何在我的电脑中同时拥有 `Django 1.10` 和 `Django 0.9` 两套环境呢？这时候我们就可以通过虚拟环境来解决这个问题。

## 虚拟环境原理介绍：

虚拟环境相当于一个抽屉，在这个抽屉中安装的任何软件包都不会影响到其他抽屉。并且在项目中，我可以指定这个项目的虚拟环境来配合我的项目。比如我们现在有一个项目是基于 `Django 1.10.x` 版本，又有一个项目是基于 `Django 0.9.x` 的版本，那么这时候就可以创建两个虚拟环境，在这两个虚拟环境中分别安装 `Django 1.10.x` 和 `Django 0.9.x` 来适配我们的项目。

## 安装 `virtualenv` ：

`virtualenv` 是用来创建虚拟环境的软件工具，我们可以通过 `pip` 或者 `pip3` 来安装：

```
pip install virtualenv
pip3 install virtualenv
```

## 创建虚拟环境：

创建虚拟环境非常简单，通过以下命令就可以创建了：

```
virtualenv [虚拟环境的名字]
```

如果你当前的 `Python3/Scripts` 的查找路径在 `Python2/Scripts` 的前面，那么将会使用 `python3` 作为这个虚拟环境的解释器。如果 `python2/Scripts` 在 `python3/Scripts` 前面，那么将会使用 `Python2` 来作为这个虚拟环境的解释器。

## 进入环境：

虚拟环境创建好了以后，那么可以进入到这个虚拟环境中，然后安装一些第三方包，进入虚拟环境在不同的操作系统中有不同的方式，一般分为两种，第一种是 `Windows` ，第二种是 `*nix` ：

1. `windows` 进入虚拟环境：进入到虚拟环境的 `Scripts` 文件夹中，然后执行 `activate` 。

2. `*nix` 进入虚拟环境：`source /path/to/virtualenv/bin/activate`
   一旦你进入到了这个虚拟环境中，你安装包，卸载包都是在这个虚拟环境中，不会影响到外面的环境。

### 退出虚拟环境：

退出虚拟环境很简单，通过一个命令就可以完成：`deactivate`。

### 创建虚拟环境的时候指定 `Python` 解释器：

在电脑的环境变量中，一般是不会去更改一些环境变量的顺序的。也就是说比如你的 `Python2/Scripts` 在 `Python3/Scripts` 的前面，那么你不会经常去更改他们的位置。但是这时候我确实是想在创建虚拟环境的时候用 `Python3` 这个版本，这时候可以通过 `-p` 参数来指定具体的 `Python` 解释器：

```
    virtualenv -p C:\Python36\python.exe [virutalenv name]
```

---

## virtualenvwrapper：

`virtualenvwrapper` 这个软件包可以让我们管理虚拟环境变得更加简单。不用再跑到某个目录下通过 `virtualenv` 来创建虚拟环境，并且激活的时候也要跑到具体的目录下去激活。

### 安装 `virtualenvwrapper`：

1. *nix：`pip install virtualenvwrapper`。
2. windows：`pip install virtualenvwrapper-win`。

### `virtualenvwrapper` 基本使用：

1. 创建虚拟环境：

   ```
   mkvirtualenv my_env
   ```

   那么会在你当前用户下创建一个 `Env` 的文件夹，然后将这个虚拟环境安装到这个目录下。
   如果你电脑中安装了 `python2` 和 `python3`，并且两个版本中都安装了 `virtualenvwrapper`，那么将会使用环境变量中第一个出现的 `Python` 版本来作为这个虚拟环境的 `Python` 解释器。

2. 切换到某个虚拟环境：

   ```
   workon my_env
   ```

3. 退出当前虚拟环境：

```
deactivate
```

4. 删除某个虚拟环境：

```
rmvirtualenv my_env
```

5. 列出所有虚拟环境：

```
lsvirtualenv
```

6. 进入到虚拟环境所在的目录：

```
cdvirtualenv
```

修改 **mkvirtualenv** 的默认路径：

在 我的电脑->右键->属性->高级系统设置->环境变量->系统变量 中添加一个参数 WORKON_HOME ，将这个参数的值设置为你需要的路径。

创建虚拟环境的时候指定 **Python** 版本：

在使用 mkvirtualenv 的时候，可以指定 --python 的参数来指定具体的 python 路径：

```
mkvirtualenv --python==C:\Python36\python.exe hy_env
```

# 认识web

## url详解：

`URL` 是 `Uniform Resource Locator` 的简写，统一资源定位符。

一个 `URL` 由以下几部分组成：

```
scheme://host:port/path/?query-string=xxx#anchor
```

- **scheme**：代表的是访问的协议，一般为 `http` 或者 `https` 以及 `ftp` 等。
- **host**：主机名，域名，比如 `www.baidu.com` 。
- **port**：端口号。当你访问一个网站的时候，浏览器默认使用80端口。
- **path**：查找路径。比如： `www.jianshu.com/trending/now` ，后面的 `trending/now` 就是 `path` 。
- **query-string**：查询字符串，比如： `www.baidu.com/s?wd=python` ，后面的 `wd=python` 就是查询字符串。
- **anchor**：锚点，后台一般不用管，前端用来做页面定位的。

注意： `URL` 中的所有字符都是 `ASCII` 字符集，如果出现非 `ASCII` 字符，比如中文，浏览器会进行编码再进行传输。

## web服务器和应用服务器以及web应用框架：

- **web**服务器：负责处理http请求，响应静态文件，常见的有 `Apache` ， `Nginx` 以及微软的 `IIS` .
- 应用服务器：负责处理逻辑的服务器。比如 `php` 、 `python` 的代码，是不能直接通过 `nginx` 这种**web**服务器来处理的，只能通过应用服务器来处理，常见的应用服务器有 `uwsgi` 、 `tomcat` 等。
- **web**应用框架：一般使用某种语言，封装了常用的 `web` 功能的框架就是**web**应用框架， `flask` 、 `Django` 以及Java中的 `SSH(Structs2+Spring3+Hibernate3)` 框架都是web应用框架。

## Content-type和Mime-type的作用和区别：

两者都是指定服务器和客户端之间传输数据的类型，区别如下：

- Content-type：既可以指定传输数据的类型，也可以指定数据的编码类型，例如： `text/html;charset=utf-8`
- Mime-type：不能指定传输的数据编码类型。例如： `text/html`

常用的数据类型如下：

- text/html（默认的，html文件）
- text/plain（纯文本）
- text/css（css文件）
- text/javascript（js文件）
- application/x-www-form-urlencoded（普通的表单提交）
- multipart/form-data（文件提交）
- application/json（json传输）
- application/xml（xml文件）

# Flask简介：

`flask` 是一款非常流行的 `Python Web` 框架，出生于2010年，作者是 `Armin Ronacher` ,本来这个项目只是作者在愚人节的一个玩笑，后来由于非常受欢迎，进而成为一个正式的项目。目前为止最新的版本是 `0.12.2` 。

`flask` 自2010年发布第一个版本以来，大受欢迎，深得开发者的喜爱，并且在多个公司已经得到了应用，flask能如此流行的原因，可以分为以下几点：

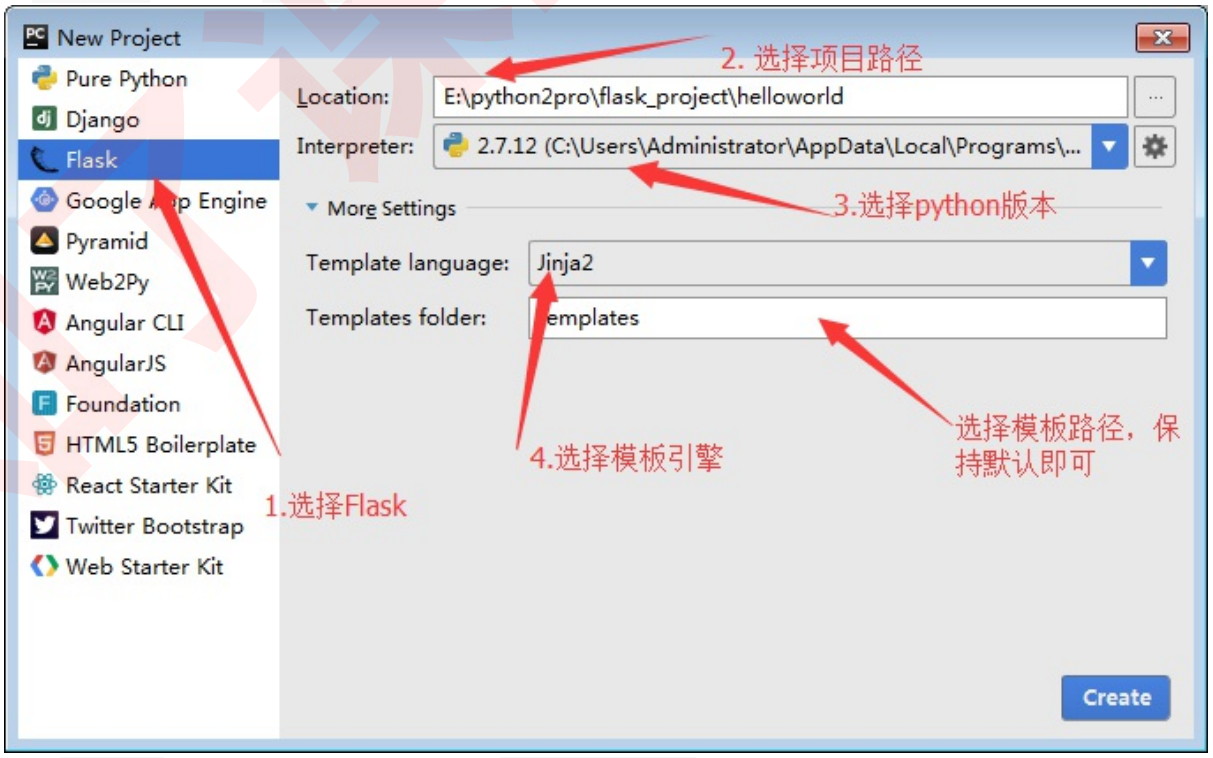
- 微框架、简洁、只做他需要做的，给开发者提供了很大的扩展性。
- Flask和相应的插件写得很好，用起来很爽。
- 开发效率非常高，比如使用 `SQLAlchemy` 的 `ORM` 操作数据库可以节省开发者大量书写 `sql` 的时间。

`Flask` 的灵活度非常之高，他不会帮你做太多的决策，一些你都可以按照自己的意愿进行更改。比如：

- 使用 `Flask` 开发数据库的时候，具体是使用 `SQLAlchemy` 还是 `MongoEngine` ，选择权完全掌握在你自己的手中。区别于 `Django` ， `Django` 内置了非常完善和丰富的功能，并且如果你想替换成你自己想要的，要么不支持，要么非常麻烦。
- 把默认的 `Jinija2` 模板引擎替换成其他模板引擎都是非常容易的。

## 第一个flask程序：

用 `pycharm` 新建一个 `flask` 项目，新建项目的截图如下：




点击 `create` 后创建一个新项目，然后在 `helloworld.py` 文件中书写代码：

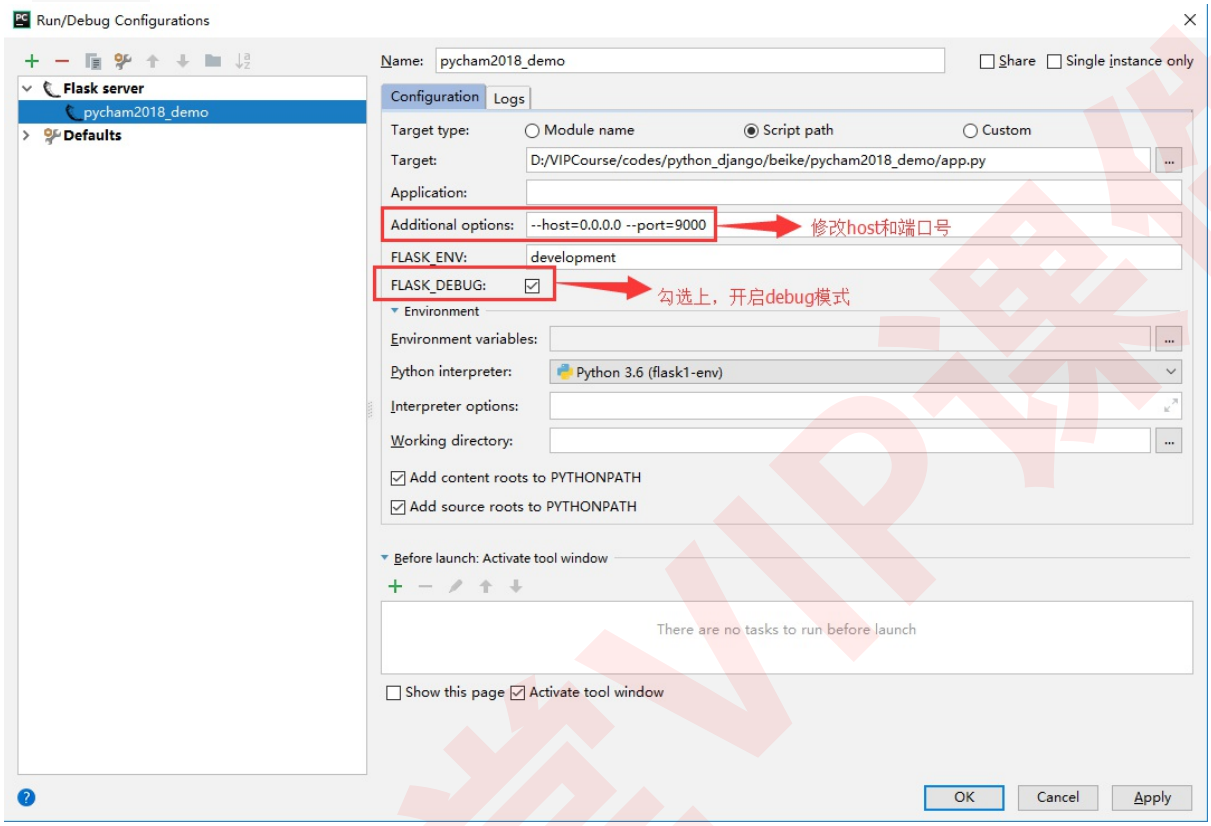
```
#coding: utf8

# 从flask框架中导入Flask类
from flask import Flask

# 传入__name__初始化一个Flask实例
app = Flask(__name__)

# app.route装饰器映射URL和执行的函数。这个设置将根URL映射到了hello_world函数上
@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    # 运行本项目，host=0.0.0.0可以让其他电脑也能访问到该网站，port指定访问的端口。默认的host是127.0.0.1，port为5000
    app.run(host='0.0.0.0',port=9000)
```

然后点击运行，在浏览器中输入 `http://127.0.0.1:9000` 就能看到 `hello world` 了。需要说明一点的是， `app.run` 这种方式只适合于开发，如果在生产环境中，应该使用 `Gunicorn` 或者 `uWSGI` 来启动。如果是在终端运行的，可以按 `ctrl+c` 来让服务停止。

---

## pycharm 2018开启debug模式和修改host：

在 `Pycharm 2018` 中，如果想要开启 `debug` 模式和更改端口号，则需要编辑项目配置。直接在 `app.run` 中更改是无效的。示例图如下：



# 项目配置

## 设置为**DEBUG**模式：

默认情况下 `flask` 不会开启 `DEBUG` 模式，开启 `DEBUG` 模式后，flask会在每次保存代码的时候自动的重新载入代码，并且如果代码有错误，会在终端进行提示。

开启 `DEBUG` 模式有三种方式：

1. 直接在应用对象上设置：

```
app.debug = True
app.run()
```

2. 在执行 `run` 方法的时候，传递参数进去：

```
app.run(debug=True)
```

3. 在 `config` 属性中设置：

```
app.config.update(DEBUG=True)
```

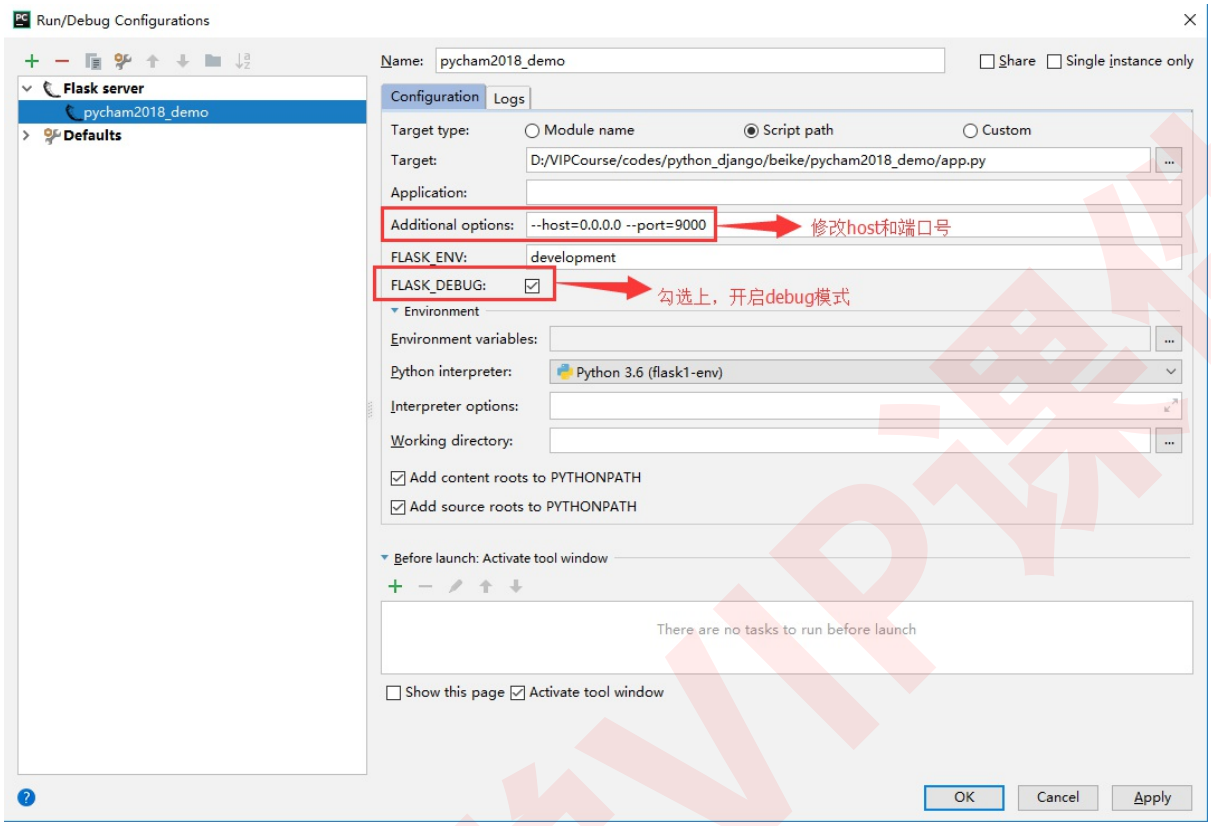
   如果一切正常，会在终端打印以下信息：

```
* Restarting with stat
* Debugger is active!
* Debugger pin code: 294-745-044
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
```

需要注意的是，只能在开发环境下开启 `DEBUG` 模式，因为 `DEBUG` 模式会带来非常大的安全隐患。

另外，在开启了 `DEBUG` 模式后，当程序有异常而进入错误堆栈模式，你第一次点击某个堆栈想查看变量值的时候，页面会弹出一个对话框，让你输入 `PIN` 值，这个 `PIN` 值在你启动的时候就会出现，比如在刚刚启动的项目中的 `PIN` 值为294-745-044，你输入这个值后， `Werkzeug` 会把这个 `PIN` 值作为 `cookie` 的一部分保存起来，并在8小时候过期，8小时以内不需要再输入PIN值。这样做的目的是为了更加的安全，让调试模式下的攻击者更难攻击到本站。

## **pycharm**开启**debug**模式：




## 配置文件：

`Flask` 项目的配置，都是通过 `app.config` 对象来进行配置的。比如要配置一个项目处于 `DEBUG` 模式下，那么可以使用 `app.config['DEBUG] = True` 来进行设置，那么 `Flask` 项目将以 `DEBUG` 模式运行。在 `Flask` 项目中，有四种方式进行项目的配置：

1. 直接硬编码：

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

2. 因为 `app.config` 是 `flask.config.Config` 的实例，而 `Config` 类是继承自 `dict` ，因此可以通过 `update` 方法：

```
app.config.update(
    DEBUG=True,
    SECRET_KEY='...'
)
```

3. 如果你的配置项特别多，你可以把所有的配置项都放在一个模块中，然后通过加载模块的方式进行配置，假设有一个 `settings.py` 模块，专门用来存储配置项的，此时你可以通过 `app.config.from_object()` 方法进行加载，并且该方法既可以接收模块的的字符串名称，也可以模块对象：

```
# 1. 通过模块字符串
app.config.from_object('settings')
# 2. 通过模块对象
import settings
app.config.from_object(settings)
```

4. 也可以通过另外一个方法加载，该方法就是 `app.config.from_pyfile()` ，该方法传入一个文件名，通常是以 `.py` 结尾的文件，但也不限于只使用 `.py` 后缀的文件：

```
app.config.from_pyfile('settings.py',silent=True)
# silent=True表示如果配置文件不存在的时候不抛出异常，默认是为False，会抛出异常。
```

`Flask` 项目内置了许多的配置项，所有的内置配置项，可以在这里查看。

# URL与视图

## URL与函数的映射：

从之前的 `helloworld.py` 文件中，我们已经看到，一个 `URL` 要与执行函数进行映射，使用的是 `@app.route` 装饰器。 `@app.route` 装饰器中，可以指定 `URL` 的规则来进行更加详细的映射，比如现在要映射一个文章详情的 `URL` ，文章详情的 `URL` 是 `/article/id/` ，id有可能为1、2、3...,那么可以通过以下方式：

```
@app.route('/article/<id>/')
def article(id):
    return '%s article detail' % id
```

其中 `<id>` ，尖括号是固定写法，语法为 `<variable>` ， `variable` 默认的数据类型是字符串。如果需要指定类型，则要写成 `<converter:variable>` ，其中 `converter` 就是类型名称，可以有以下几种：

- string: 默认的数据类型，接受没有任何斜杠 `/` 的字符串。
- int: 整形
- float: 浮点型。
- path： 和 `string` 类似，但是可以传递斜杠 `/` 。
- uuid： `uuid` 类型的字符串。
- any：可以指定多种路径，这个通过一个例子来进行说明:

```
@app.route('/<any(article,blog):url_path>/')
def item(url_path):
    return url_path
```

以上例子中， `item` 这个函数可以接受两个 `URL` ,一个是 `/article/` ，另一个是 `/blog/` 。并且，一定要传 `url_path` 参数，当然这个 `url_path` 的名称可以随便。

如果不想定制子路径来传递参数，也可以通过传统的 `?=` 的形式来传递参数，例如： `/article?id=xxx` ，这种情况下，可以通过 `request.args.get('id')` 来获取 `id` 的值。如果是 `post` 方法，则可以通过 `request.form.get('id')` 来进行获取。

## 构造URL（url_for）：

一般我们通过一个 `URL` 就可以执行到某一个函数。如果反过来，我们知道一个函数，怎么去获得这个 `URL` 呢？ `url_for` 函数就可以帮我们实现这个功能。 `url_for()` 函数接收两个及以上的参数，他接收函数名作为第一个参数，接收对应**URL**规则的命名参数，如果还出现其他的参数，则会添加到 `URL` 的后面作为查询参数。

通过构建 `URL` 的方式而选择直接在代码中拼 `URL` 的原因有两点：

1. 将来如果修改了 `URL` ，但没有修改该 `URL` 对应的函数名，就不用到处去替换 `URL` 了。
2. `url_for()` 函数会转义一些特殊字符和 `unicode` 字符串，这些事情 `url_for` 会自动的帮我们搞定。

下面用一个例子来进行解释：

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

@app.route('/article/<id>/')
def article(id):
    return '%s article detail' % id

@app.route('/')
def index(request):
    print(url_for("article",id=1))
    return "首页"
```

在访问index的时候，会打印出`/article/1/`。

## 自定义URL转换器：

刚刚在URL映射的时候，我们看到了Flask内置了几种数据类型的转换器，比如有int/string等。如果Flask内置的转换器不能满足你的需求，此时你可以自定义转换器。自定义转换器，需要满足以下几个条件：

1. 转换器是一个类，且必须继承自werkzeug.routing.BaseConverter。
2. 在转换器类中，实现to_python(self,value)方法，这个方法的返回值，将会传递到view函数中作为参数。
3. 在转换器类中，实现to_url(self,values)方法，这个方法的返回值，将会在调用url_for函数的时候生成符合要求的URL形式。

比如，拿一个官方的例子来说，Reddit可以通过在URL中用一个加号（+）隔开社区的名字，方便同时查看来自多个社区的帖子。比如访问“www.reddit.com\/r\/flask+lisp\/”的时候，就同时可以查看flask和lisp两个社区的帖子，现在我们自定义一个转换器来实现这个功能：

```
#coding: utf-8
from flask import Flask,url_for
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def __init__(self,url_map,separator='+'):
```

```
        super(ListConverter,self).__init__(url_map)
        self.separator = separator

    def to_python(self, value):
        return value.split(self.separator)

    def to_url(self, values):
        return self.separator.join(BaseConverter.to_url(self,value) for value in values
)

app.url_map.converters['list'] = ListConverter

@app.route('/community1/<list:page_names>')
def community1(page_names):
    return '%s+%s' % tuple(page_names)

@app.route('/community2/<list('|'):page_names>/')
def community2(page_names):
    return "%s|%s" % tuple(page_names)
```

communityu1使用的是默认的+号进行连接，而第二种方式使用了|进行连接。

## 指定URL末尾的斜杠：

有些 `URL` 的末尾是有斜杠的，有些 `URL` 末尾是没有斜杠的。这其实是两个不同的 `URL` 。

举个例子：

```
@app.route('/article/')
def articles():
    return '文章列表页'
```

上述例子中，当访问一个结尾不带斜线的 `URL` ： `/article` ，会被重定向到带斜线的 `URL` ： `/article/` 上去。但是当我们在定义 `article` 的 `url` 的时候，如果在末尾没有加上斜杠，但是在访问的时候又加上了斜杠，这时候就会抛出一个 `404` 错误页面了：

```
@app.route("/article")
def articles(request):
    return "文章列表页面"
```

以上没有在末尾加斜杠，因此在访问 `/article/` 的时候，就会抛出一个 `404` 错误。

## 指定HTTP方法：

在 `@app.route()` 中可以传入一个关键字参数 `methods` 来指定本方法支持的 `HTTP` 方法，默认情况下，只能使用 `GET` 请求，看以下例子：

```
@app.route('/login/',methods=['GET','POST'])
def login():
    return 'login'
```

以上装饰器将让 `login` 的 `URL` 既能支持 `GET` 又能支持 `POST` 。

## 页面跳转和重定向：

重定向分为永久性重定向和暂时性重定向，在页面上体现的操作就是浏览器会从一个页面自动跳转到另外一个页面。比如用户访问了一个需要权限的页面，但是该用户当前并没有登录，因此我们应该给他重定向到登录页面。

- 永久性重定向： `http` 的状态码是 `301` ，多用于旧网址被废弃了要转到一个新的网址确保用户的访问，最经典的就是京东网站，你输入 `www.jingdong.com` 的时候，会被重定向到 `www.jd.com` ，因为 `jingdong.com` 这个网址已经被废弃了，被改成 `jd.com` ，所以这种情况下应该用永久重定向。
- 暂时性重定向： `http` 的状态码是 `302` ，表示页面的暂时性跳转。比如访问一个需要权限的网址，如果当前用户没有登录，应该重定向到登录页面，这种情况下，应该用暂时性重定向。

在 `flask` 中，重定向是通过 `flask.redirect(location,code=302)` 这个函数来实现的， `location` 表示需要重定向到的 `URL` ，应该配合之前讲的 `url_for()` 函数来使用， `code` 表示采用哪个重定向，默认是 `302` 也即 `暂时性重定向` ，可以修改成 `301` 来实现永久性重定向。

以下来看一个例子，关于在 `flask` 中怎么使用重定向：

```
from flask import Flask,url_for,redirect

app = Flask(__name__)
app.debug = True

@app.route('/login/',methods=['GET','POST'])
def login():
    return 'login page'

@app.route('/profile/',methods=['GET','POST'])
def profile():
    name = request.args.get('name')

    if not name:
    # 如果没有name，说明没有登录，重定向到登录页面
        return redirect(url_for('login'))
    else:
        return name
```

# 关于响应（Response）：

视图函数中可以返回以下类型的值：

- `Response` 对象。
- 字符串。其实 `Flask` 是根据返回的字符串类型，重新创建一个 `werkzeug.wrappers.Response` 对象， `Response` 将该字符串作为主体，状态码为200， `MIME` 类型为 `text/html` ，然后返回该 `Response` 对象。
- 元组。元组中格式是 `(response,status,headers)` 。 `response` 为一个字符串， `status` 值是状态码， `headers` 是一些响应头。
- 如果不是以上三种类型。那么 `Flask` 会通过 `Response.force_type(rv,request.environ)` 转换为一个请求对象。

以下将用例子来进行说明：

第一个例子：直接使用 `Response` 创建：

```
from werkzeug.wrappers import Response

@app.route('/about/')
def about():
    resp = Response(response='about page',status=200,content_type='text/html;charset=u
tf-8')
    return resp
```

第二个例子：可以使用 `make_response` 函数来创建 `Response` 对象，这个方法可以设置额外的数据，比如设置cookie，header信息等：

```
from flask import make_response

@app.route('/about/')
def about():
    return make_response('about page')
```

第三个例子：通过返回元组的形式：

```
@app.errorhandler(404)
def not_found():
    return 'not found',404
```

# 模板简介：

模板是一个 `web` 开发必备的模块。因为我们在渲染一个网页的时候，并不是只渲染一个纯文本字符串，而是需要渲染一个有富文本标签的页面。这时候我们就需要使用模板了。在 `Flask` 中，配套的模板是 `Jinja2` ， `Jinja2` 的作者也是 `Flask` 的作者。这个模板非常的强大，并且执行效率高。以下对 `Jinja2` 做一个简单介绍！

## Flask渲染 `Jinja` 模板：

要渲染一个模板，通过 `render_template` 方法即可，以下将用一个简单的例子进行讲解：

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

@app.route('/about/')
def about():
    return render_template('about.html')
```

当访问 `/about/` 的时候， `about()` 函数会在当前目录下的 `templates` 文件夹下寻找 `about.html` 模板文件。如果想更改模板文件地址，应该在创建 `app` 的时候，给 `Flask` 传递一个关键字参数 `template_folder` ，指定具体的路径，再看以下例子：

```
from flask import Flask,render_template
app = Flask(__name__,template_folder=r'C:\templates')

@app.route('/about/')
def about():
    return render_template('about.html')
```

以上例子将会在C盘的templates文件夹中寻找模板文件。还有最后一点是，如果模板文件中有参数需要传递，应该怎么传呢，我们再来看一个例子：

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

@app.route('/about/')
def about():
    # return render_template('about.html',user='zhiliao')
    return render_template('about.html',**{'user':'zhiliao'})
```

以上例子介绍了两种传递参数的方式，因为 `render_template` 需要传递的是一个关键字参数，所以第一种方式是顺其自然的。但是当你的模板中要传递的参数过多的时候，把所有参数放在一个函数中显然不是一个好的选择，因此我们使用字典进行包装，并且加两个 `*` 号，来转换成关键字参数。

# Jinja2模版概述

## 概要：

先看一个简单例子：

```
1. <html lang="en">
2. <head>
3.    <title>My Webpage</title>
4. </head>
5. <body>
6.     <ul id="navigation">
7.     {% for item in navigation %}
8.         <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
9.     {% endfor %}
10.    </ul>
11.
12.    {{ a_variable }}
13.    {{ user.name }}
14.    {{ user['name'] }}
15.
16.    {# a comment #}
17. </body>
18.</html>
```

以上示例有需要进行解释：

- 第12~14行的 `{{ ... }}` ：用来装载一个变量，模板渲染的时候，会把这个变量代表的值替换掉。并且可以间接访问一个变量的属性或者一个字典的 `key` 。关于点 `.` 号访问和 `[]` 中括号访问，没有任何区别，都可以访问属性和字典的值。
- 第7~9行的 `{% ... %}` ：用来装载一个控制语句，以上装载的是 `for` 循环，以后只要是要用到控制语句的，就用 `{% ... %}` 。
- 第14行的 `{# ... #}` ：用来装载一个注释，模板渲染的时候会忽视这中间的值。

## 属性访问规则：

1. 比如在模板中有一个变量这样使用： `foo.bar` ，那么在 `Jinja2` 中是这样进行访问的：
    - 先去查找 `foo` 的 `bar` 这个属性，也即通过 `getattr(foo,'bar')` 。
    - 如果没有，就去通过 `foo.__getitem__('bar')` 的方式进行查找。
    - 如果以上两种方式都没有找到，返回一个 `undefined` 。
2. 在模板中有一个变量这样使用： `foo['bar']` ，那么在 `Jinja2` 中是这样进行访问：
    - 通过 `foo.__getitem__('bar')` 的方式进行查找。

- 如果没有，就通过 `getattr(foo,'bar')` 的方式进行查找。
- 如果以上没有找到，则返回一个 `undefined` 。

# Jinja2模版过滤器

过滤器是通过管道符号（ | ）进行使用的，例如： {{ name|length }} ，将返回name的长度。过滤器相当于是一个函数，把当前的变量传入到过滤器中，然后过滤器根据自己的功能，再返回相应的值，之后再将结果渲染到页面中。 Jinja2 中内置了许多过滤器，在这里可以看到所有的过滤器，现对一些常用的过滤器进行讲解：

- abs(value) ：返回一个数值的绝对值。 例如： -1|abs 。
- default(value,default_value,boolean=false) ：如果当前变量没有值，则会使用参数中的值来代替。 name|default('xiaotuo') ——如果name不存在，则会使用 xiaotuo 来替代。 boolean=False 默认是在只有这个变量为 undefined 的时候才会使用 default 中的值，如果想使用 python 的形式判断是否为 false ，则可以传递 boolean=true 。也可以使用 or 来替换。
- escape(value)或e ：转义字符，会将 < 、 > 等符号转义成HTML中的符号。例如： content|escape 或 content|e 。
- first(value) ：返回一个序列的第一个元素。 names|first 。
- format(value,*arags,**kwargs) ：格式化字符串。例如以下代码：

  ```
  {{ "%s" - "%s"|format('Hello?',"Foo!") }}将输出：Helloo? - Foo!
  ```

- last(value) ：返回一个序列的最后一个元素。示例： names|last 。
- length(value) ：返回一个序列或者字典的长度。示例： names|length 。
- join(value,d=u'') ：将一个序列用 d 这个参数的值拼接成字符串。
- safe(value) ：如果开启了全局转义，那么 safe 过滤器会将变量关掉转义。示例： content_html|safe 。
- int(value) ：将值转换为 int 类型。
- float(value) ：将值转换为 float 类型。
- lower(value) ：将字符串转换为小写。
- upper(value) ：将字符串转换为小写。
- replace(value,old,new) ： 替换将 old 替换为 new 的字符串。
- truncate(value,length=255,killwords=False) ：截取 length 长度的字符串。
- striptags(value) ：删除字符串中所有的HTML标签，如果出现多个空格，将替换成一个空格。

- `trim`：截取字符串前面和后面的空白字符。
- `string(value)`：将变量转换成字符串。
- `wordcount(s)`：计算一个长字符串中单词的个数。

# 控制语句

所有的控制语句都是放在 `{% ... %}` 中，并且有一个语句 `{% endxxx %}` 来进行结束， `Jinja` 中常用的控制语句有 `if/for..in..` ，现对他们进行讲解：

1. `if` ：if语句和 `python` 中的类似，可以使用 `>`，`<`，`<=`，`>=`，`==`，`!=` 来进行判断，也可以通过 `and`，`or`，`not`，`()` 来进行逻辑合并操作，以下看例子：

```
{% if kenny.sick %}
   Kenny is sick.
{% elif kenny.dead %}
 You killed Kenny!  You bastard!!!
{% else %}
 Kenny looks okay --- so far
{% endif %}
```

2. `for...in...` ： `for` 循环可以遍历任何一个序列包括列表、字典、元组。并且可以进行反向遍历，以下将用几个例子进行解释：

   - 普通的遍历：

```
<ul>
{% for user in users %}
<li>{{ user.username|e }}</li>
{% endfor %}
</ul>
```

   - 遍历字典：

```
<dl>
{% for key, value in my_dict.iteritems() %}
<dt>{{ key|e }}</dt>
<dd>{{ value|e }}</dd>
{% endfor %}
</dl>
```

   - 如果序列中没有值的时候，进入 `else` ：

```
<ul>
{% for user in users %}
<li>{{ user.username|e }}</li>
{% else %}
<li><em>no users found</em></li>
{% endfor %}
```

```
    </ul>
```

并且 `Jinja` 中的 `for` 循环还包含以下变量，可以用来获取当前的遍历状态：

| 变量 | 描述 | | --- | --- | | loop.index | 当前迭代的索引（从1开始） | | loop.index0 | 当前迭代的索引（从0开始） | | loop.first | 是否是第一次迭代，返回True或False | | loop.last | 是否是最后一次迭代，返回True或False | | loop.length | 序列的长度 |

另外，不可以使用 `continue` 和 `break` 表达式来控制循环的执行。

# 测试器

测试器主要用来判断一个值是否满足某种类型，并且这种类型一般通过普通的 `if` 判断是有很大的挑战的。语法是： `if...is...` ，先来简单的看个例子：

```
{% if variable is escaped%}
    value of variable: {{ escaped }}
{% else %}
    variable is not escaped
{% endif %}
```

以上判断 `variable` 这个变量是否已经被转义了， `Jinja` 中内置了许多的测试器，看以下列表：

| 测试器 | 说明 |
| --- | --- |
| `callable(object)` | 是否可调用 |
| `defined(object)` | 是否已经被定义了。 |
| `escaped(object)` | 是否已经被转义了。 |
| `upper(object)` | 是否全是大写。 |
| `lower(object)` | 是否全是小写。 |
| `string(object)` | 是否是一个字符串。 |
| `sequence(object)` | 是否是一个序列。 |
| `number(object)` | 是否是一个数字。 |
| `odd(object)` | 是否是奇数。 |
| `even(object)` | 是否是偶数。 |

# 宏和import语句

## 宏：

模板中的宏跟python中的函数类似，可以传递参数，但是不能有返回值，可以将一些经常用到的代码片段放到宏中，然后把一些不固定的值抽取出来当成一个变量，以下将用一个例子来进行解释：

```
{% macro input(name, value='', type='text') %}
    <input type="{{ type }}" name="{{ name }}" value="{{ value|e }}">
{% endmacro %}
```

以上例子可以抽取出了一个input标签，指定了一些默认参数。那么我们以后创建 `input` 标签的时候，可以通过他快速的创建：

```
<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```

## import语句：

在真实的开发中，会将一些常用的宏单独放在一个文件中，在需要使用的时候，再从这个文件中进行导入。 `import` 语句的用法跟 `python` 中的 `import` 类似，可以直接 `import...as...` ，也可以 `from...import...` 或者 `from...import...as...` ，假设现在有一个文件，叫做 `forms.html` ，里面有两个宏分别为 `input` 和 `textarea` ，如下：

```
forms.html:
{% macro input(name, value='', type='text') %}
    <input type="{{ type }}" value="{{ value|e }}" name="{{ name }}">
{% endmacro %}

{% macro textarea(name, value='', rows=10, cols=40) %}
    <textarea name="{{ name }}" rows="{{ rows }}" cols="{{ cols
    }}">{{ value|e }}</textarea>
{% endmacro %}
```

导入宏的例子：

1. `import...as...` 形式：

```
{% import 'forms.html' as forms %}
<dl>
  <dt>Username</dt>
  <dd>{{ forms.input('username') }}</dd>
```

```
  <dt>Password</dt>
  <dd>{{ forms.input('password', type='password') }}</dd>
</dl>
<p>{{ forms.textarea('comment') }}</p>
```

2. `from...import...as.../from...import...` 形式：

```
{% from 'forms.html' import input as input_field, textarea %}
<dl>
  <dt>Username</dt>
  <dd>{{ input_field('username') }}</dd>
  <dt>Password</dt>
  <dd>{{ input_field('password', type='password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

另外需要注意的是，导入模板并不会把当前上下文中的变量添加到被导入的模板中，如果你想要导入一个需要访问当前上下文变量的宏，有两种可能的方法:

- 显式地传入请求或请求对象的属性作为宏的参数。
- 与上下文一起（with context）导入宏。

与上下文中一起（with context）导入的方式如下:

```
{% from '_helpers.html' import my_macro with context %}
```

# include和set语句

## include语句：

`include` 语句可以把一个模板引入到另外一个模板中，类似于把一个模板的代码copy到另外一个模板的指定位置，看以下例子：

```
{% include 'header.html' %}
    主体内容
{% include 'footer.html' %}
```

## 赋值（set）语句：

有时候我们想在在模板中添加变量，这时候赋值语句（set）就派上用场了，先看以下例子：

```
{% set name='zhiliao' %}
```

那么以后就可以使用 `name` 来代替 `zhiliao` 这个值了，同时，也可以给他赋值为列表和元组：

```
{% set navigation = [('index.html', 'Index'), ('about.html', 'About')] %}
```

赋值语句创建的变量在其之后都是有效的，如果不想让一个变量污染全局环境，可以使用 `with` 语句来创建一个内部的作用域，将 `set` 语句放在其中，这样创建的变量只在 `with` 代码块中才有效，看以下示例：

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}           foo is 42 here
{% endwith %}
```

也可以在 `with` 的后面直接添加变量，比如以上的写法可以修改成这样：

```
{% with foo = 42 %}
    {{ foo }}
{% endwith %}
```

这两种方式都是等价的，一旦超出 `with` 代码块，就不能再使用 `foo` 这个变量了。

# 模版继承

`Flask` 中的模板可以继承，通过继承可以把模板中许多重复出现的元素抽取出来，放在父模板中，并且父模板通过定义 `block` 给子模板开一个口，子模板根据需要，再实现这个 `block` ，假设现在有一个 `base.html` 这个父模板，代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="base.css" />
    <title>{% block title %}{% endblock %}</title>
    {% block head %}{% endblock %}
</head>
<body>
    <div id="body">{% block body %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>
        {% endblock %}
    </div>
</body>
</html>
```

以上父模板中，抽取了所有模板都需要用到的元素 `html` 、 `body` 等，并且对于一些所有模板都要用到的样式文件 `style.css` 也进行了抽取，同时对于一些子模板需要重写的地方，比如 `title` 、 `head` 、 `body` 都定义成了 `block` ，然后子模板可以根据自己的需要，再具体的实现。以下再来看子模板的代码：

```
{% extends "base.html" %}
{% block title %}首页{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .detail{
            color: red;
        }
    </style>
{% endblock %}
{% block content %}
    <h1>这里是首页</h1>
    <p class="detail">
      首页的内容
    </p>
{% endblock %}
```

首先第一行就定义了子模板继承的父模板，并且可以看到子模板实现了 `title` 这个 `block`，并填充了自己的内容，再看 `head` 这个 `block`，里面调用了 `super()` 这个函数，这个函数的目的是执行父模板中的代码，把父模板中的内容添加到子模板中，如果没有这一句，则父模板中处在 `head` 这个 `block` 中的代码将会被子模板中的代码给覆盖掉。

另外，模板中不能出现重名的 `block`，如果一个地方需要用到另外一个 `block` 中的内容，可以使用 `self.blockname` 的方式进行引用，比如以下示例：

```
<title>
    {% block title %}
        这是标题
    {% endblock %}
</title>
<h1>{{ self.title() }}</h1>
```

以上示例中 `h1` 标签重用了 `title` 这个 `block` 中的内容，子模板实现了 `title` 这个 `block`， `h1` 标签也能拥有这个值。

另外，在子模板中，所有的文本标签和代码都要添加到从父模板中继承的 `block` 中。否则，这些文本和标签将不会被渲染。

# 转义

转义的概念是，在模板渲染字符串的时候，字符串有可能包括一些非常危险的字符比如 `<` 、 `>` 等，这些字符会破坏掉原来 `HTML` 标签的结构，更严重的可能会发生 `XSS` 跨域脚本攻击，因此如果碰到 `<` 、 `>` 这些字符的时候，应该转义成 `HTML` 能正确表示这些字符的写法，比如 `>` 在 `HTML` 中应该用 `<` 来表示等。

但是 `Flask` 中默认没有开启全局自动转义，针对那些以 `.html` 、 `.htm` 、 `.xml` 和 `.xhtml` 结尾的文件，如果采用 `render_template` 函数进行渲染的，则会开启自动转义。并且当用 `render_template_string` 函数的时候，会将所有的字符串进行转义后再渲染。而对于 `Jinja2` 默认没有开启全局自动转义，作者有自己的原因：

1. 渲染到模板中的字符串并不是所有都是危险的，大部分还是没有问题的，如果开启自动转义，那么将会带来大量的不必要的开销。
2. `Jinja2` 很难获取当前的字符串是否已经被转义过了，因此如果开启自动转义，将对一些已经被转义过的字符串发生二次转义，在渲染后会破坏原来的字符串。

在没有开启自动转义的模式下（比如以 `.conf` 结尾的文件），对于一些不信任的字符串，可以通过 `{{ content_html|e }}` 或者是 `{{ content_html|escape }}` 的方式进行转义。在开启了自动转义的模式下，如果想关闭自动转义，可以通过 `{{ content_html|safe }}` 的方式关闭自动转义。而 `{%autoescape true/false%}...{%endautoescape%}` 可以将一段代码块放在中间，来关闭或开启自动转义，例如以下代码关闭了自动转义：

```
{% autoescape false %}
  <p>autoescaping is disabled here
  <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

# 数据类型和运算符

## 数据类型：

`Jinja` 支持许多数据类型，包括：字符串、整型、浮点型、列表、元组、字典、**True/False**。

## 运算符：

- `+` 号运算符：可以完成数字相加，字符串相加，列表相加。但是并不推荐使用 `+` 运算符来操作字符串，字符串相加应该使用 `~` 运算符。
- `-` 号运算符：只能针对两个数字相减。
- `/` 号运算符：对两个数进行相除。
- `%` 号运算符：取余运算。
- `*` 号运算符：乘号运算符，并且可以对字符进行相乘。
- `**` 号运算符：次幂运算符，比如2**3=8。
- `in` 操作符：跟python中的 `in` 一样使用，比如 `true` 返回 `true` 。
- `~` 号运算符：拼接多个字符串，比如 `HelloWorld` 将返回 `HelloWorld` 。

# 静态文件的配置

`Web` 应用中会出现大量的静态文件来使得网页更加生动美观。类似于 `CSS` 样式文件、 `JavaScript` 脚本文件、图片文件、字体文件等静态资源。在 `Jinja` 中加载静态文件非常简单，只需要通过 `url_for` 全局函数就可以实现，看以下代码：

```
<link href="{{ url_for('static',filename='about.css') }}">
```

`url_for` 函数默认会在项目根目录下的 `static` 文件夹中寻找 `about.css` 文件，如果找到了，会生成一个相对于项目根目录下的 `/static/about.css` 路径。当然我们也可以把静态文件不放在 `static` 文件夹中，此时就需要具体指定了，看以下代码：

```
app = Flask(__name__,static_folder='C:\static')
```

那么访问静态文件的时候，将会到 `/static` 这个文件夹下寻找。

# 类视图

之前我们接触的视图都是函数，所以一般简称视图函数。其实视图也可以基于类来实现，类视图的好处是支持继承，但是类视图不能跟函数视图一样，写完类视图还需要通过 `app.add_url_rule(url_rule,view_func)` 来进行注册。以下将对两种类视图进行讲解：

## 标准类视图：

标准类视图是继承自 `flask.views.View` ，并且在子类中必须实现 `dispatch_request` 方法，这个方法类似于视图函数，也要返回一个基于 `Response` 或者其子类的对象。以下将用一个例子进行讲解：

```
from flask.views import View
class PersonalView(View):
    def dispatch_request(self):
        return "知了课堂"
# 类视图通过add_url_rule方法和url做映射
app.add_url_rule('/users/',view_func=PersonalView.as_view('personalview'))
```

## 基于调度方法的视图：

`Flask` 还为我们提供了另外一种类视图 `flask.views.MethodView` ，对每个HTTP方法执行不同的函数（映射到对应方法的小写的同名方法上），以下将用一个例子来进行讲解：

```
    class LoginView(views.MethodView):
        # 当客户端通过get方法进行访问的时候执行的函数
        def get(self):
            return render_template("login.html")

        # 当客户端通过post方法进行访问的时候执行的函数
        def post(self):
            email = request.form.get("email")
            password = request.form.get("password")
            if email == 'xx@qq.com' and password == '111111':
                return "登录成功！"
            else:
                return "用户名或密码错误！"

    # 通过add_url_rule添加类视图和url的映射，并且在as_view方法中指定该url的名称，方便url_for函数调用

    app.add_url_rule('/myuser/',view_func=LoginView.as_view('loginview'))
```

用类视图的一个缺陷就是比较难用装饰器来装饰，比如有时候需要做权限验证的时候，比如看以下例子：

```
from flask import session
def login_required(func):
    def wrapper(*args,**kwargs):
        if not session.get("user_id"):
            return 'auth failure'
        return func(*args,**kwargs)
    return wrapper
```

装饰器写完后，可以在类视图中定义一个属性叫做 `decorators`，然后存储装饰器。以后每次调用这个类视图的时候，就会执行这个装饰器。示例代码如下：

```
class UserView(views.MethodView):
    decorators = [user_required]
    ...
```

# 蓝图和子域名

## 蓝图：

之前我们写的 `url` 和视图函数都是处在同一个文件，如果项目比较大的话，这显然不是一个合理的结构，而蓝图可以优雅的帮我们实现这种需求。以下看一个使用蓝图的文件的例子：

```
from flask import Blueprint
bp = Blueprint('user',__name__,url_prefix='/user/')

@bp.route('/')
def index():
    return "用户首页"

@bp.route('profile/')
def profile():
    return "个人简介"
```

然后我们在主程序中，通过 `app.register_blueprint()` 方法将这个蓝图注册进url映射中，看下主 `app` 的实现：

```
from flask import Flask
import user

app = Flask(__name__)
app.register_blueprint(user.bp)

if __name__ == '__main__':
    app.run(host='0.0.0.0',port=9000)
```

以后访问 `/user/` ， `/user/profile/` ，都是执行的 `user.py` 文件中的视图函数，这样就实现了项目的模块化。

以上是对蓝图的一个简单介绍，但是使用蓝图还有几个需要注意的地方，就是在蓝图如何寻找静态文件、模板文件， `url_for` 函数如何反转 `url` ，以下分别进行解释：

寻找静态文件：

默认不设置任何静态文件路径， `Jinja2` 会在项目的 `static` 文件夹中寻找静态文件。也可以设置其他的路径，在初始化蓝图的时候， `Blueprint` 这个构造函数，有一个参数 `static_folder` 可以指定静态文件的路径，如：

```
bp = Blueprint('admin',__name__,url_prefix='/admin',static_folder='static')
```

`static_folder` 可以是相对路径（相对蓝图文件所在的目录），也可以是绝对路径。在配置完蓝图后，还有一个需要注意的地方是如何在模板中引用静态文件。在模板中引用蓝图，应该要使用 `蓝图名+.+static` 来引用，如下所示：

```
<link href="{{ url_for('admin.static',filename='about.css') }}">
```

寻找模板文件：

跟静态文件一样，默认不设置任何模板文件的路径，将会在项目的 `templates` 中寻找模板文件。也可以设置其他的路径，在构造函数 `Blueprint` 中有一个 `template_folder` 参数可以设置模板的路径，如下所示：

```
bp = Blueprint('admin',__name__,url_prefix='/admin',template_folder='templates')
```

模板文件和静态文件有点区别，以上代码写完以后，如果你渲染一个模板 `return render_template('admin.html')` ， `Flask` 默认会去项目根目录下的 `templates` 文件夹中查找 `admin.html` 文件，如果找到了就直接返回，如果没有找到，才会去蓝图文件所在的目录下的 `templates` 文件夹中寻找。

**url_for**生成 `url` ：

用 `url_for` 生成蓝图的 `url` ，使用的格式是： `蓝图名称+.+视图函数名称` 。比如要获取 `admin` 这个蓝图下的 `index` 视图函数的 `url` ，应该采用以下方式：

```
url_for('admin.index')
```

其中这个蓝图名称是在创建蓝图的时候，传入的第一个参数。 `bp = Blueprint('admin',__name__,url_prefix='/admin',template_folder='templates')`

## 子域名：

子域名在许多网站中都用到了，比如一个网站叫做 `xxx.com` ，那么我们可以定义一个子域名 `cms.xxx.com` 来作为 `cms` 管理系统的网址，子域名的实现一般也是通过蓝图来实现，在之前章节中，我们创建蓝图的时候添加了一个 `url_prefix=/user` 作为url前缀，那样我们就可以通过 `/user/` 来访问 `user` 下的url。但使用子域名则不需要。另外，还需要配置 `SERVER_NAME` ，比如 `app.config[SERVER_NAME]='example.com:9000'` 。并且在注册蓝图的时候，还需要添加一个 `subdomain` 的参数，这个参数就是子域名的名称，先来看一下蓝图的实现(admin.py)：

```
from flask import Blueprint
bp = Blueprint('admin',__name__,subdomain='admin')

@bp.route('/')
def admin():
    return 'Admin Page'
```

这个没有多大区别，接下来看主 `app` 的实现：

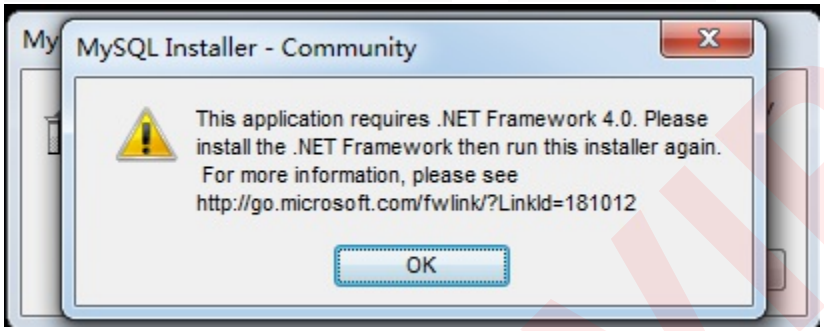
```
from flask import Flask
import admin

# 配置`SERVER_NAME`
app.config['SERVER_NAME'] = 'example.com:8000'
# 注册蓝图，指定了subdomain
app.register_blueprint(admin.bp)

if __name__ == '__main__':
    app.run(host='0.0.0.0',port=8000,debug=True)
```

写完以上两个文件后，还是不能正常的访问 `admin.example.com:8000` 这个子域名，因为我们没有在 `host` 文件中添加域名解析，你可以在最后添加一行 `127.0.0.1 admin.example.com` ，就可以访问到了。另外，子域名不能在 `127.0.0.1` 上出现，也不能在 `localhost` 上出现。

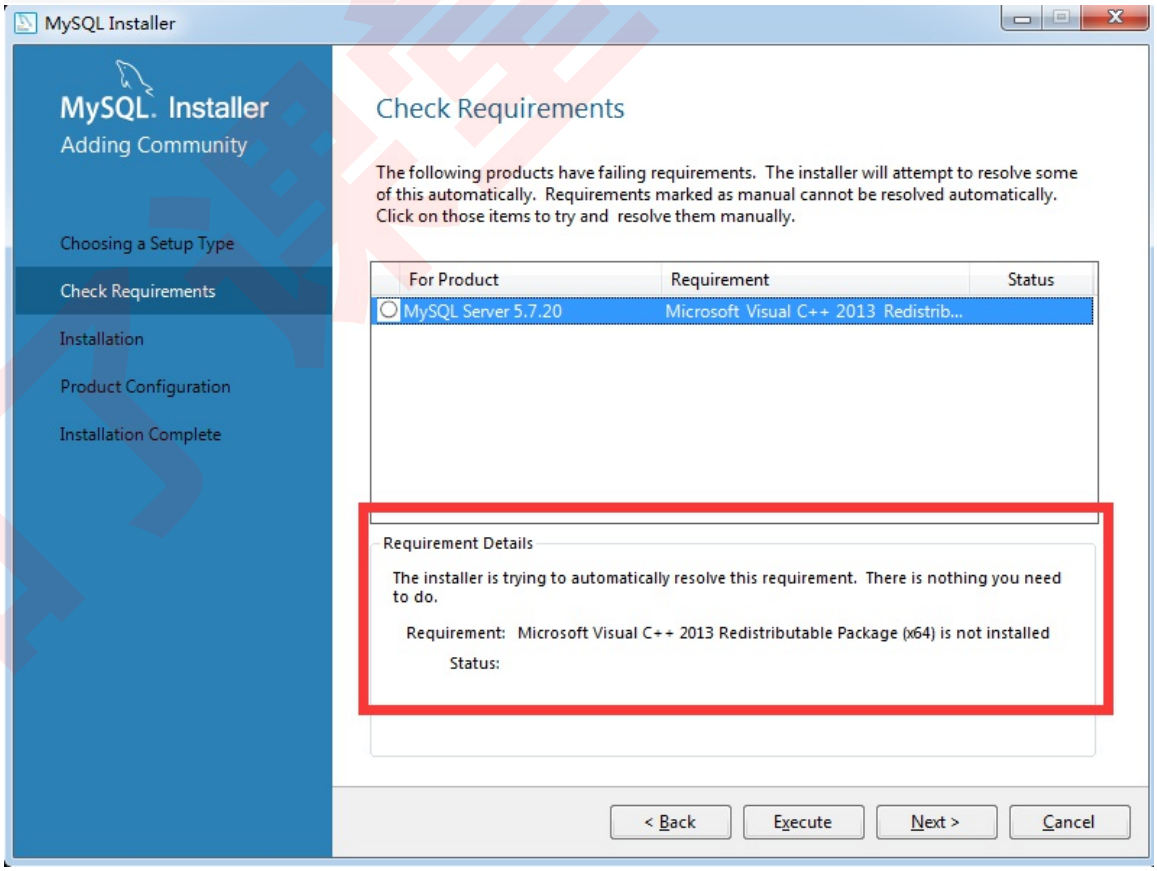
# MySQL数据库的安装

## 在 Windows 下安装 MySQL：

1. 在 MySQL 的官网下载 MySQL 数据库： https://dev.mysql.com/downloads/windows/installer/5.7.html 。
2. 然后双击安装，如果出现以下错误，则到 http://www.microsoft.com/en-us/download/details.aspx?id=17113 下载 .net framework 。然后安装。




3. 在安装过程中，如果提示没有 Microsoft C++ 2013 ，那么就到以下网址下载安装即可： http://download.microsoft.com/download/9/0/5/905DBD86-D1B8-4D4B-8A50-CB0E922017B9/vcredist_x64.exe 。




4. 然后做好用户名与密码的配置即可。

# SQLAlchemy介绍和基本使用

数据库是一个网站的基础。`Flask`可以使用很多种数据库。比如`MySQL`，`MongoDB`，`SQLite`，`PostgreSQL`等。这里我们以`MySQL`为例进行讲解。而在`Flask`中，如果想要操作数据库，我们可以使用`ORM`来操作数据库，使用`ORM`操作数据库将变得非常简单。

在讲解`Flask`中的数据库操作之前，先确保你已经安装了以下软件：

- `mysql`：如果是在`windows`上，到官网下载。如果是`ubuntu`，通过命令`sudo apt-get install mysql-server libmysqlclient-dev -yq`进行下载安装。
- `MySQLdb`：`MySQLdb`是用`Python`来操作`mysql`的包，因此通过`pip`来安装，命令如下：`pip install mysql-python`。
- `pymysql`：`pymysql`是用`Python`来操作`mysql`的包，因此通过`pip`来安装，命令如下：`pip3 install pymysql`。如果您用的是`Python 3`，请安装`pymysql`。
- `SQLAlchemy`：`SQLAlchemy`是一个数据库的`ORM`框架，我们在后面会用到。安装命令为：`pip3 install SQLAlchemy`。

## 通过`SQLAlchemy`连接数据库：

首先来看一段代码：

```
from sqlalchemy import create_engine

# 数据库的配置变量
HOSTNAME = '127.0.0.1'
PORT     = '3306'
DATABASE = 'xt_flask'
USERNAME = 'root'
PASSWORD = 'root'
DB_URI = 'mysql+mysqldb://{}:{}@{}:{}/{}'.format(USERNAME,PASSWORD,HOSTNAME,PORT,DATABA
SE)

# 创建数据库引擎
engine = create_engine(DB_URI)

#创建连接
with engine.connect() as con:
    rs = con.execute('SELECT 1')
    print rs.fetchone()
```

首先从`sqlalchemy`中导入`create_engine`，用这个函数来创建引擎，然后用`engine.connect()`来连接数据库。其中一个比较重要的一点是，通过`create_engine`函数的时候，需要传递一个满足某种格式的字符串，对这个字符串的格式来进行解释：

```
dialect+driver://username:password@host:port/database?charset=utf8
```

`dialect` 是数据库的实现，比如 `MySQL` 、 `PostgreSQL` 、 `SQLite` ，并且转换成小写。 `driver` 是 `Python` 对应的驱动，如果不指定，会选择默认的驱动，比如MySQL的默认驱动是 `MySQLdb` 。 `username` 是连接数据库的用户名， `password` 是连接数据库的密码， `host` 是连接数据库的域名， `port` 是数据库监听的端口号， `database` 是连接哪个数据库的名字。

如果以上输出了 `1` ，说明 `SQLAlchemy` 能成功连接到数据库。

## 用SQLAlchemy执行原生SQL：

我们将上一个例子中的数据库配置选项单独放在一个 `constants.py` 的文件中，看以下例子：

```
from sqlalchemy import create_engine
from constants import DB_URI

#连接数据库
engine = create_engine(DB_URI,echo=True)

# 使用with语句连接数据库，如果发生异常会被捕获
with engine.connect() as con:
    # 先删除users表
    con.execute('drop table if exists authors')
    # 创建一个users表，有自增长的id和name
    con.execute('create table authors(id int primary key auto_increment,'name varchar(2
5))')
    # 插入两条数据到表中
    con.execute('insert into persons(name) values("abc")')
    con.execute('insert into persons(name) values("xiaotuo")')
    # 执行查询操作
    results = con.execute('select * from persons')
    # 从查找的结果中遍历
    for result in results:
        print(result)
```

# SQLAlchemy

## 使用SQLAlchemy：

要使用 `ORM` 来操作数据库，首先需要创建一个类来与对应的表进行映射。现在以 `User表` 来做为例子，它有 `自增长的id` 、 `name` 、 `fullname` 、 `password` 这些字段，那么对应的类为：

```
    from sqlalchemy import Column,Integer,String
    from constants import DB_URI
    from sqlalchemy import create_engine
    from sqlalchemy.ext.declarative import declarative_base

    engine = create_engine(DB_URI,echo=True)

    # 所有的类都要继承自`declarative_base`这个函数生成的基类
    Base = declarative_base(engine)
    class User(Base):
        # 定义表名为users
        __tablename__ = 'users'

        # 将id设置为主键，并且默认是自增长的
        id = Column(Integer,primary_key=True)
        # name字段，字符类型，最大的长度是50个字符
        name = Column(String(50))
        fullname = Column(String(50))
        password = Column(String(100))

        # 让打印出来的数据更好看，可选的
        def __repr__(self):
            return "<User(id='%s',name='%s',fullname='%s',password='%s')>" % (self.id,s
elf.name,self.fullname,self.password)
```

`SQLAlchemy` 会自动的设置第一个 `Integer` 的主键并且没有被标记为外键的字段添加自增长的属性。因此以上例子中 `id` 自动的变成自增长的。以上创建完和表映射的类后，还没有真正的映射到数据库当中，执行以下代码将类映射到数据库中：

```
Base.metadata.create_all()
```

在创建完数据表，并且做完和数据库的映射后，接下来让我们添加数据进去：

```
ed_user = User(name='ed',fullname='Ed Jones',password='edspassword')
# 打印名字
print ed_user.name
> ed
```

```
# 打印密码
print ed_user.password
> edspassword
# 打印id
print ed_user.id
> None
```

可以看到，name和password都能正常的打印，唯独 `id` 为 `None` ，这是因为 `id` 是一个自增长的主键，还未插入到数据库中， `id` 是不存在的。接下来让我们把创建的数据插入到数据库中。和数据库打交道的，是一个叫做 `Session` 的对象：

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
# 或者
# Session = sessionmaker()
# Session.configure(bind=engine)
session = Session()
ed_user = User(name='ed',fullname='Ed Jones',password='edspassword')
session.add(ed_user)
```

现在只是把数据添加到 `session` 中，但是并没有真正的把数据存储到数据库中。如果需要把数据存储到数据库中，还要做一次 `commit` 操作：

```
session.commit()
# 打印ed_user的id
print ed_user.id
> 1
```

这时候， `ed_user` 就已经有id。说明已经插入到数据库中了。有人肯定有疑问了，为什么添加到 `session` 中后还要做一次 `commit` 操作呢，这是因为，在 `SQLAlchemy` 的 `ORM` 实现中，在做 `commit` 操作之前，所有的操作都是在事务中进行的，因此如果你要将事务中的操作真正的映射到数据库中，还需要做 `commit` 操作。既然用到了事务，这里就并不能避免的提到一个回滚操作了，那么看以下代码展示了如何使用回滚（接着以上示例代码）：

```
# 修改ed_user的用户名
ed_user.name = 'Edwardo'
# 创建一个新的用户
fake_user = User(name='fakeuser',fullname='Invalid',password='12345')
# 将新创建的fake_user添加到session中
session.add(fake_user)
# 判断`fake_user`是否在`session`中存在
print fake_user in session
> True
# 从数据库中查找name=Edwardo的用户
tmp_user = session.query(User).filter_by(name='Edwardo')
```

```
# 打印tmp_user的name
print tmp_user
# 打印出查找到的tmp_user对象，注意这个对象的name属性已经在事务中被修改为Edwardo了。
> <User(name='Edwardo', fullname='Ed Jones', password='edspassword')>
# 刚刚所有的操作都是在事务中进行的，现在来做回滚操作
session.rollback()
# 再打印tmp_user
print tmp_user
> <User(name='ed', fullname='Ed Jones', password='edspassword')>
# 再看fake_user是否还在session中
print fake_user in session
> False
```

接下来看下如何进行查找操作，查找操作是通过 `session.query()` 方法实现的，这个方法会返回一个 `Query` 对象， `Query` 对象相当于一个数组，装载了查找出来的数据，并且可以进行迭代。具体里面装的什么数据，就要看向 `session.query()` 方法传的什么参数了，如果只是传一个 `ORM` 的类名作为参数，那么提取出来的数据就是都是这个类的实例，比如：

```
for instance in session.query(User).order_by(User.id):
    print instance
# 输出所有的user实例
> <User (id=2,name='ed',fullname='Ed Json',password='12345')>
> <User (id=3,name='be',fullname='Be Engine',password='123456')>
```

如果传递了两个及其两个以上的对象，或者是传递的是 `ORM` 类的属性，那么查找出来的就是元组，例如：

```
for instance in session.query(User.name):
    print instance
# 输出所有的查找结果
> ('ed',)
> ('be',)
```

以及：

```
for instance in session.query(User.name,User.fullname):
    print instance
# 输出所有的查找结果
> ('ed', 'Ed Json')
> ('be', 'Be Engine')
```

或者是：

```
for instance in session.query(User,User.name).all():
    print instance
```

```
# 输出所有的查找结果
> (<User (id=2,name='ed',fullname='Ed Json',password='12345')>, 'Ed Json')
> (<User (id=3,name='be',fullname='Be Engine',password='123456')>, 'Be Engine')
```

另外，还可以对查找的结果（ `Query` ）做切片操作：

```
for instance in session.query(User).order_by(User.id)[1:3]
    instance
```

如果想对结果进行过滤，可以使用 `filter_by` 和 `filter` 两个方法，这两个方法都是用来做过滤的，区别在于， `filter_by` 是传入关键字参数， `filter` 是传入条件判断，并且 `filter` 能够传入的条件更多更灵活，请看以下例子：

```
# 第一种：使用filter_by过滤：
for name in session.query(User.name).filter_by(fullname='Ed Jones'):
    print name
# 输出结果：
> ('ed',)

# 第二种：使用filter过滤：
for name in session.query(User.name).filter(User.fullname=='Ed Jones'):
    print name
# 输出结果：
> ('ed',)
```

# SQLAlchemy的ORM（2）

## Column常用参数：

- `default`：默认值。
- `nullable`：是否可空。
- `primary_key`：是否为主键。
- `unique`：是否唯一。
- `autoincrement`：是否自动增长。
- `onupdate`：更新的时候执行的函数。
- `name`：该属性在数据库中的字段映射。

## sqlalchemy常用数据类型：

- `Integer`：整形。
- `Float`：浮点类型。
- `Boolean`：传递`True/False`进去。
- `DECIMAL`：定点类型。
- `enum`：枚举类型。
- `Date`：传递`datetime.date()`进去。
- `DateTime`：传递`datetime.datetime()`进去。
- `Time`：传递`datetime.time()`进去。
- `String`：字符类型，使用时需要指定长度，区别于`Text`类型。
- `Text`：文本类型。
- `LONGTEXT`：长文本类型。

## query可用参数：

1. 模型对象。指定查找这个模型中所有的对象。
2. 模型中的属性。可以指定只查找某个模型的其中几个属性。
3. 聚合函数。
    - `func.count`：统计行的数量。
    - `func.avg`：求平均值。
    - `func.max`：求最大值。
    - `func.min`：求最小值。
    - `func.sum`：求和。

## 过滤条件：

过滤是数据提取的一个很重要的功能，以下对一些常用的过滤条件进行解释，并且这些过滤条件都是只能通过`filter`方法实现的：

1. `equals`：

```
query.filter(User.name == 'ed')
```

2. `not equals`：

```
query.filter(User.name != 'ed')
```

3. `like`：

```
query.filter(User.name.like('%ed%'))
```

4. `in`：

```
query.filter(User.name.in_(['ed','wendy','jack']))
# 同时，in也可以作用于一个Query
query.filter(User.name.in_(session.query(User.name).filter(User.name.like('%ed%')))
)
```

5. `not in`：

```
query.filter(~User.name.in_(['ed','wendy','jack']))
```

6. `is null`：

```
query.filter(User.name==None)
# 或者是
query.filter(User.name.is_(None))
```

7. `is not null`：

```
query.filter(User.name != None)
# 或者是
query.filter(User.name.isnot(None))
```

8. `and`：

```
from sqlalchemy import and_
query.filter(and_(User.name=='ed',User.fullname=='Ed Jones'))
# 或者是传递多个参数
query.filter(User.name=='ed',User.fullname=='Ed Jones')
# 或者是通过多次filter操作
```

```
query.filter(User.name=='ed').filter(User.fullname=='Ed Jones')
```

9. `or`：

```
from sqlalchemy import or_  query.filter(or_(User.name=='ed',User.name=='wendy'))
```

# SQLAlchemy的ORM（3）

## 查找方法：

介绍完过滤条件后，有一些经常用到的查找数据的方法也需要解释一下：

1. `all()`：返回一个 Python 列表（ `list` ）：

```
query = session.query(User).filter(User.name.like('%ed%').order_by(User.id)
# 输出query的类型
print type(query)
> <type 'list'>
# 调用all方法
query = query.all()
# 输出query的类型
print type(query)
> <class 'sqlalchemy.orm.query.Query'>
```

2. `first()`：返回 `Query` 中的第一个值：

```
user = session.query(User).first()
print user
> <User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

3. `one()`：查找所有行作为一个结果集，如果结果集中只有一条数据，则会把这条数据提取出来，如果这个结果集少于或者多于一条数据，则会抛出异常。总结一句话：有且只有一条数据的时候才会正常的返回，否则抛出异常：

```
# 多于一条数据
user = query.one()
> Traceback (most recent call last):
> ...
> MultipleResultsFound: Multiple rows were found for one()
# 少于一条数据
user = query.filter(User.id == 99).one()
> Traceback (most recent call last):
> ...
> NoResultFound: No row was found for one()
# 只有一条数据
> query(User).filter_by(name='ed').one()
```

4. `one_or_none()`：跟 `one()` 方法类似，但是在结果集中没有数据的时候也不会抛出异常。

5. `scalar()`：底层调用 `one()` 方法，并且如果 `one()` 方法没有抛出异常，会返回查询到的第一列的数据：

```
session.query(User.name,User.fullname).filter_by(name='ed').scalar()
```

## 文本SQL：

`SQLAlchemy` 还提供了使用文本**SQL**的方式来进行查询，这种方式更加的灵活。而文本SQL要装在一个 `text()` 方法中，看以下例子：

```
from sqlalchemy import text
for user in session.query(User).filter(text("id<244")).order_by(text("id")).all():
    print user.name
```

如果过滤条件比如上例中的244存储在变量中，这时候就可以通过传递参数的形式进行构造：

```
session.query(User).filter(text("id<:value and name=:name")).params(value=224,name='ed'
).order_by(User.id)
```

在文本SQL中的变量前面使用了 `:` 来区分，然后使用 `params` 方法，指定需要传入进去的参数。另外，使用 `from_statement` 方法可以把过滤的函数和条件函数都给去掉，使用纯文本的SQL:

```
sesseion.query(User).from_statement(text("select * from users where name=:name")).param
s(name='ed').all()
```

使用 `from_statement` 方法一定要注意， `from_statement` 返回的是一个 `text` 里面的查询语句，一定要记得调用 `all()` 方法来获取所有的值。

## 计数（Count）：

`Query` 对象有一个非常方便的方法来计算里面装了多少数据：

```
session.query(User).filter(User.name.like('%ed%')).count()
```

当然，有时候你想明确的计数，比如要统计 `users` 表中有多少个不同的姓名，那么简单粗暴的采用以上 `count` 是不行的，因为姓名有可能会重复，但是处于两条不同的数据上，如果在原生数据库中，可以使用 `distinct` 关键字，那么在 `SQLAlchemy` 中，可以通过 `func.count()` 方法来实现：

```
from sqlalchemy import func
session.query(func.count(User.name),User.name).group_by(User.name).all()
# 输出的结果
> [(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

另外，如果想实现 `select count(*) from users` ，可以通过以下方式来实现：

```
session.query(func.count(*)).select_from(User).scalar()
```

当然，如果指定了要查找的表的字段，可以省略 `select_from()` 方法：

```
session.query(func.count(User.id)).scalar()
```

# SQLAlchemy的ORM（4）

## 表关系：

表之间的关系存在三种：一对一、一对多、多对多。而 `SQLAlchemy` 中的 `ORM` 也可以模拟这三种关系。因为一对一其实在 `SQLAlchemy` 中底层是通过一对多的方式模拟的，所以先来看下一对多的关系：

## 外键：

在Mysql中，外键可以让表之间的关系更加紧密。而SQLAlchemy同样也支持外键。通过ForeignKey类来实现，并且可以指定表的外键约束。相关示例代码如下：

```
class Article(Base):
    __tablename__ = 'article'
    id = Column(Integer,primary_key=True,autoincrement=True)
    title = Column(String(50),nullable=False)
    content = Column(Text,nullable=False)
    uid = Column(Integer,ForeignKey('user.id'))

    def __repr__(self):
        return "<Article(title:%s)>" % self.title

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer,primary_key=True,autoincrement=True)
    username = Column(String(50),nullable=False)
```

外键约束有以下几项：

1. `RESTRICT`：父表数据被删除，会阻止删除。默认就是这一项。
2. `NO ACTION`：在MySQL中，同 `RESTRICT`。
3. `CASCADE`：级联删除。
4. `SET NULL`：父表数据被删除，子表数据会设置为NULL。

## 一对多：

拿之前的 `User` 表为例，假如现在要添加一个功能，要保存用户的邮箱帐号，并且邮箱帐号可以有多个，这时候就必须创建一个新的表，用来存储用户的邮箱，然后通过 `user.id` 来作为外键进行引用，先来看下邮箱表的实现：

```
    from sqlalchemy import ForeignKey
    from sqlalchemy.orm import relationship
```

```
class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer,primary_key=True)
    email_address = Column(String,nullable=False)
    # User表的外键，指定外键的时候，是使用的是数据库表的名称，而不是类名
    user_id = Column(Integer,ForeignKey('users.id'))
    # 在ORM层面绑定两者之间的关系，第一个参数是绑定的表的类名，
    # 第二个参数back_populates是通过User反向访问时的字段名称
    user = relationship('User',back_populates="addresses")

    def __repr__(self):
        return "<Address(email_address='%s')>" % self.email_address

# 重新修改User表，添加了addresses字段，引用了Address表的主键
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer,primary_key=True)
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(100))
    # 在ORM层面绑定和`Address`表的关系
    addresses = relationship("Address",order_by=Address.id,back_populates="user")
```

其中，在 `User` 表中添加的 `addresses` 字段，可以通过 `User.addresses` 来访问和这个user相关的所有address。在 `Address` 表中的 `user` 字段，可以通过 `Address.user` 来访问这个user。达到了双向绑定。表关系已经建立好以后，接下来就应该对其进行操作，先看以下代码：

```
jack = User(name='jack',fullname='Jack Bean',password='gjffdd')
jack.addresses = [Address(email_address='jack@google.com'),
                  Address(email_address='j25@yahoo.com')]
session.add(jack)
session.commit()
```

首先，创建一个用户，然后对这个 `jack` 用户添加两个邮箱，最后再提交到数据库当中，可以看到这里操作 `Address` 并没有直接进行保存，而是先添加到用户里面，再保存。

## 一对一：

一对一其实就是一对多的特殊情况，从以上的一对多例子中不难发现，一对应的是 `User` 表，而多对应的是 `Address` ，也就是说一个 `User` 对象有多个 `Address` 。因此要将一对多转换成一对一，只要设置一个 `User` 对象对应一个 `Address` 对象即可，看以下示例：

```
class User(Base):
  __tablename__ = 'users'
  id = Column(Integer,primary_key=True)
```

```
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(100))
    # 设置uselist关键字参数为False
    addresses = relationship("Address",back_populates='addresses',uselist=False)
  class Address(Base):
    __tablename__ = 'addresses'
    id = Column(Integer,primary_key=True)
    email_address = Column(String(50))
    user_id = Column(Integer,ForeignKey('users.id')
    user = relationship('Address',back_populates='user')
```

从以上例子可以看到，只要在 `User` 表中的 `addresses` 字段上添加 `uselist=False` 就可以达到一对一的效果。设置了一对一的效果后，就不能添加多个邮箱到 `user.addresses` 字段了，只能添加一个：

```
user.addresses = Address(email_address='ed@google.com')
```

## 多对多：

多对多需要一个中间表来作为连接，同理在 `sqlalchemy` 中的 `orm` 也需要一个中间表。假如现在有一个 `Teacher` 和一个 `Classes` 表，即老师和班级，一个老师可以教多个班级，一个班级有多个老师，是一种典型的多对多的关系，那么通过 `sqlalchemy` 的 `ORM` 的实现方式如下：

```
  association_table = Table('teacher_classes',Base.metadata,
  Column('teacher_id',Integer,ForeignKey('teacher.id')),
  Column('classes_id',Integer,ForeignKey('classes.id'))
  )
  class Teacher(Base):
    __tablename__ = 'teacher'
    id = Column(Integer,primary_key=True)
    tno = Column(String(10))
    name = Column(String(50))
    age = Column(Integer)
    classes = relationship('Classes',secondary=association_table,back_populates='teache
rs')
  class Classes(Base):
    __tablename__ = 'classes'
    id = Column(Integer,primary_key=True)
    cno = Column(String(10))
    name = Column(String(50))
    teachers = relationship('Teacher',secondary=association_table,back_populates='class
es')
```

要创建一个多对多的关系表，首先需要一个中间表，通过 `Table` 来创建一个中间表。上例中第一个参数 `teacher_classes` 代表的是中间表的表名，第二个参数是 `Base` 的元类，第三个和第四个参数就是要连接的两个表，其中 `Column` 第一个参数是表示的是连接表的外键名，第二个参数表示这个外键的类型，第三个参数表示要外键的表名和字段。
创建完中间表以后，还需要在两个表中进行绑定，比如在 `Teacher` 中有一个 `classes` 属性，来绑定 `Classes` 表，并且通过 `secondary` 参数来连接中间表。同理， `Classes` 表连接 `Teacher` 表也是如此。定义完类后，之后就是添加数据，请看以下示例：

```
teacher1 = Teacher(tno='t1111',name='xiaotuo',age=10)
teacher2 = Teacher(tno='t2222',name='datuo',age=10)
classes1 = Classes(cno='c1111',name='english')
classes2 = Classes(cno='c2222',name='math')
teacher1.classes = [classes1,classes2]
teacher2.classes = [classes1,classes2]
classes1.teachers = [teacher1,teacher2]
classes2.teachers = [teacher1,teacher2]
session.add(teacher1)
session.add(teacher2)
session.add(classes1)
session.add(classes2)
```

# SQLAlchemy的ORM（6）

## ORM层面的CASCADE：

如果将数据库的外键设置为 `RESTRICT` ，那么在 `ORM` 层面，删除了父表中的数据，那么从表中的数据将会 `NULL` 。如果不想要这种情况发生，那么应该将这个值的 `nullable=False` 。

在 `SQLAlchemy` ，只要将一个数据添加到 `session` 中，和他相关联的数据都可以一起存入到数据库中了。这些是怎么设置的呢？其实是通过 `relationship` 的时候，有一个关键字参数 `cascade` 可以设置这些属性：

1. `save-update` ：默认选项。在添加一条数据的时候，会把其他和他相关联的数据都添加到数据库中。这种行为就是 `save-update` 属性影响的。
2. `delete` ：表示当删除某一个模型中的数据的时候，是否也删掉使用 `relationship` 和他关联的数据。
3. `delete-orphan` ：表示当对一个ORM对象解除了父表中的关联对象的时候，自己便会被删除掉。当然如果父表中的数据被删除，自己也会被删除。这个选项只能用在一对多上，不能用在多对多以及多对一上。并且还需要在子模型中的 `relationship` 中，增加一个 `single_parent=True` 的参数。
4. `merge` ：默认选项。当在使用 `session.merge` ，合并一个对象的时候，会将使用了 `relationship` 相关联的对象也进行 `merge` 操作。
5. `expunge` ：移除操作的时候，会将相关联的对象也进行移除。这个操作只是从session中移除，并不会真正的从数据库中删除。
6. `all` ：是对 `save-update, merge, refresh-expire, expunge, delete` 几种的缩写。

## 排序：

1. order_by：可以指定根据这个表中的某个字段进行排序，如果在前面加了一个 `-` ，代表的是降序排序。
2. 在模型定义的时候指定默认排序：有些时候，不想每次在查询的时候都指定排序的方式，可以在定义模型的时候就指定排序的方式。有以下两种方式：
    - relationship的order_by参数：在指定 `relationship` 的时候，传递 `order_by` 参数来指定排序的字段。
    - 在模型定义中，添加以下代码：

    ```
    __mapper_args__ = {
        "order_by": title
      }
    ```

    即可让文章使用标题来进行排序。

3. 正向排序和反向排序：默认情况是从小到大，从前到后排序的，如果想要反向排序，可以调用排序的字段的 `desc` 方法。

## limit、offset和切片：

1. `limit`：可以限制每次查询的时候只查询几条数据。
2. `offset`：可以限制查找数据的时候过滤掉前面多少条。
3. 切片：可以对 `Query` 对象使用切片操作，来获取想要的数据。

## 懒加载：

在一对多，或者多对多的时候，如果想要获取多的这一部分的数据的时候，往往能通过一个属性就可以全部获取了。比如有一个作者，想要或者这个作者的所有文章，那么可以通过 `user.articles` 就可以获取所有的。但有时候我们不想获取所有的数据，比如只想获取这个作者今天发表的文章，那么这时候我们可以给 `relationship` 传递一个 `lazy='dynamic'` ，以后通过 `user.articles` 获取到的就不是一个列表，而是一个 `AppendQuery` 对象了。这样就可以对这个对象再进行一层过滤和排序等操作。

## 查询高级：

### group_by：

根据某个字段进行分组。比如想要根据性别进行分组，来统计每个分组分别有多少人，那么可以使用以下代码来完成：

```
session.query(User.gender,func.count(User.id)).group_by(User.gender).all()
```

### having：

having是对查找结果进一步过滤。比如只想要看未成年人的数量，那么可以首先对年龄进行分组统计人数，然后再对分组进行having过滤。示例代码如下：

```
result = session.query(User.age,func.count(User.id)).group_by(User.age).having(User.age
 >= 18).all()
```

### join方法：

`join` 查询分为两种，一种是 `inner join` ，另一种是 `outer join` 。默认的是 `inner join` ，如果指定 `left join` 或者是 `right join` 则为 `outer join` 。如果想要查询 `User` 及其对应的 `Address` ，则可以通过以下方式来实现：

```
for u,a in session.query(User,Address).filter(User.id==Address.user_id).all():
  print u
```

```
    print a
# 输出结果：
> <User (id=1,name='ed',fullname='Ed Jason',password='123456')>
> <Address id=4,email=ed@google.com,user_id=1>
```

这是通过普通方式的实现，也可以通过 `join` 的方式实现，更加简单：

```
for u,a in session.query(User,Address).join(Address).all():
    print u
    print a
# 输出结果：
> <User (id=1,name='ed',fullname='Ed Jason',password='123456')>
> <Address id=4,email=ed@google.com,user_id=1>
```

当然，如果采用 `outerjoin` ，可以获取所有 `user` ，而不用在乎这个 `user` 是否有 `address` 对象，并且 `outerjoin` 默认为左外查询：

```
for instance in session.query(User,Address).outerjoin(Address).all():
    print instance
# 输出结果：
(<User (id=1,name='ed',fullname='Ed Jason',password='123456')>, <Address id=4,email=e
d@google.com,user_id=1>)
(<User (id=2,name='xt',fullname='xiaotuo',password='123')>, None)
```

## 别名：

当多表查询的时候，有时候同一个表要用到多次，这时候用别名就可以方便的解决命名冲突的问题了：

```
from sqlalchemy.orm import aliased
adalias1 = aliased(Address)
adalias2 = aliased(Address)
for username,email1,email2 in session.query(User.name,adalias1.email_address,adalias2
.email_address).join(adalias1).join(adalias2).all():
    print username,email1,email2
```

## 子查询：

`sqlalchemy` 也支持子查询，比如现在要查找一个用户的用户名以及该用户的邮箱地址数量。要满足这个需求，可以在子查询中找到所有用户的邮箱数（通过group by合并同一用户），然后再将结果放在父查询中进行使用：

```
from sqlalchemy.sql import func
# 构造子查询
stmt = session.query(Address.user_id.label('user_id'),func.count(*).label('address_co
```

```
unt')).group_by(Address.user_id).subquery()
  # 将子查询放到父查询中
  for u,count in session.query(User,stmt.c.address_count).outerjoin(stmt,User.id==stmt.
c.user_id).order_by(User.id):
    print u,count
```

从上面我们可以看到，一个查询如果想要变为子查询，则是通过 `subquery()` 方法实现，变成子查询后，通过 `子查询.c` 属性来访问查询出来的列。以上方式只能查询某个对象的具体字段，如果要查找整个实体，则需要通过 `aliased` 方法，示例如下：

```
  stmt = session.query(Address)
  adalias = aliased(Address,stmt)
  for user,address in session.query(User,stmt).join(stmt,User.addresses):
    print user,address
```

## 老段的作业：

http://blog.csdn.net/laoduan_78/article/details/44259245

# Flask-SQLAlchemy插件

另外一个框架，叫做 `Flask-SQLAlchemy` ， `Flask-SQLAlchemy` 是对 `SQLAlchemy` 进行了一个简单的封装，使得我们在 `flask` 中使用 `sqlalchemy` 更加的简单。可以通过 `pip install flask-sqlalchemy` 。使用 `Flask-SQLAlchemy` 的流程如下：

- 数据库初始化：数据库初始化不再是通过 `create_engine` ，请看以下示例：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from constants import DB_URI
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = DB_URI
db = SQLAlchemy(app)
```

- `ORM` 类：之前都是通过 `Base = declarative_base()` 来初始化一个基类，然后再继承，在 `Flask-SQLAlchemy` 中更加简单了（代码依赖以上示例）：

```
class User(db.Model):
  id = db.Column(db.Integer,primary_key=True)
  username = db.Column(db.String(80),unique=True)
  email = db.Column(db.String(120),unique=True)
  def __init__(self,username,email):
      self.username = username
      self.email = email
  def __repr__(self):
      return '<User %s>' % self.username
```

- 映射模型到数据库表：使用 `Flask-SQLAlchemy` 所有的类都是继承自 `db.Model` ，并且所有的 `Column` 和数据类型也都成为 `db` 的一个属性。但是有个好处是不用写表名了， `Flask-SQLAlchemy` 会自动将类名小写化，然后映射成表名。
  写完类模型后，要将模型映射到数据库的表中，使用以下代码创建所有的表：

```
db.create_all()
```

- 添加数据：这时候就可以在数据库中看到已经生成了一个 `user` 表了。接下来添加数据到表中：

```
admin = User('admin','admin@example.com')
guest = User('guest','guest@example.com')
db.session.add(admin)
db.session.add(guest)
db.session.commit()
```

添加数据和之前的没有区别，只是 `session` 成为了一个 `db` 的属性。

- 查询数据：查询数据不再是之前的 `session.query` 了，而是将 `query` 属性放在了 `db.Model` 上，所以查询就是通过 `Model.query` 的方式进行查询了：

```
users = User.query.all()
# 再如：
admin = User.query.filter_by(username='admin').first()
# 或者：
admin = User.query.filter(User.username='admin').first()
```

- 删除数据：删除数据跟添加数据类似，只不过 `session` 是 `db` 的一个属性而已：

```
db.session.delete(admin)
db.session.commit()
```

# Flask-Script：

`Flask-Script` 的作用是可以通过命令行的形式来操作 `Flask` 。例如通过命令跑一个开发版本的服务器、设置数据库，定时任务等。要使用 `Flask-Script` ，可以通过 `pip install flask-script` 安装最新版本。首先看一个最简单的例子：

```
# manage.py

from flask_script import Manager
from your_app import app

manager = Manager(app)

@manager.command
def hello():
    print 'hello'

if __name__ == '__main__':
    manager.run()
```

我们把脚本命令代码放在一个叫做 `manage.py` 文件中，然后在终端运行 `python manage.py hello` 命令，就可以看到输出 `hello` 了。

## 定义命令的三种方法：

1. 使用 `@command` 装饰器：这种方法之前已经介绍过。就不过多讲解了。
2. 使用类继承自 `Command` 类：

```
from flask_script import Command,Manager
from your_app import app

manager = Manager(app)

class Hello(Command):
    "prints hello world"

    def run(self):
        print "hello world"

manager.add_command('hello',Hello())
```

使用类的方式，有三点需要注意：

- 必须继承自 `Command` 基类。

- 必须实现 `run` 方法。
- 必须通过 `add_command` 方法添加命令。

3. 使用 `option` 装饰器：如果想要在使用命令的时候还传递参数进去，那么使用 `@option` 装饰器更加的方便：

```
@manager.option('-n','--name',dest='name')
def hello(name):
    print 'hello ',name
```

这样，调用 `hello` 命令：

```
python manage.py -n xt
```

或者

```
python manage.py --name xt
```

就可以输出：

```
hello xt
```

## 添加参数到命令中：

- option装饰器：以上三种创建命令的方式都可以添加参数， `@option` 装饰器，已经介绍过了。看以下示例介绍展示添加多个参数的方式：

```
@manager.option('-n', '--name', dest='name', default='joe')
@manager.option('-u', '--url', dest='url', default=None)
def hello(name, url):
  if url is None:
      print "hello", name
  else:
      print "hello", name, "from", url
```

- command装饰器：command装饰器也可以添加参数，但是不能那么的灵活，如下示例：

```
@manager.command
def hello(name="Fred")
  print "hello", name
```

- 类继承：类继承也可以添加参数，看以下示例：

```
  from flask_Flask import Comman,Manager,Option
```

```
class Hello(Command):
    option_list = (
        Option('--name','-n',dest='name'),
    )

    def run(self,name):
        print "hello %s" % name
```

如果要在指定参数的时候，动态的做一些事情，可以使用 `get_options` 方法，示例如下：

```
class Hello(Command):
    def __init__(self,default_name='Joe'):
self.default_name = default_name

    def get_options(self):
        return [
            Option('-n','--name',dest='name',default=self.default_name),
        ]

    def run(self,name):
        print 'hello',name
```

# alembic教程：

`alembic` 是 `sqlalchemy` 的作者开发的。用来做 `OMR` 模型与数据库的迁移与映射。 `alembic` 使用方式跟 `git` 有点了类似，表现在两个方面，第一个， `alembic` 的所有命令都是以 `alembic` 开头；第二， `alembic` 的迁移文件也是通过版本进行控制的。首先，通过 `pip install alembic` 进行安装。以下将解释 `alembic` 的用法：

1. 初始化 `alembic` 仓库：在终端中， `cd` 到你的项目目录中，然后执行命令 `alembic init alembic` ，创建一个名叫 `alembic` 的仓库。
2. 创建模型类：创建一个 `models.py` 模块，然后在里面定义你的模型类，示例代码如下：

```
from sqlalchemy import Column,Integer,String,create_engine,Text
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer,primary_key=True)
    username = Column(String(20),nullable=False)
    password = Column(String(100),nullable=False)

class Article(Base):
    __tablename__ = 'article'

    id = Column(Integer,primary_key=True)
    title = Column(String(100),nullable=False)
    content = Column(Text, nullable=False)
```

3. 修改配置文件：
   - 在 `alembic.ini` 中设置数据库的连接， `sqlalchemy.url = driver://user:pass@localhost/dbname` ，比如以 `mysql` 数据库为例，则配置后的代码为：

     ```
     sqlalchemy.url = mysql+mysqldb://root:root@localhost/alembic_demo?charset=utf8
     ```

   - 为了使用模型类更新数据库，需要在 `env.py` 文件中设置 `target_metadata` ，默认为 `target_metadata=None` 。使用 `sys` 模块把当前项目的路径导入到 `path` 中：

     ```
     import os
     import sys
     ```

```
sys.path.append(os.path.dirname(os.path.abspath(__file__)) + "/../")
from models import Base
... #省略代码
target_metadata = Base.metadata # 设置创建模型的元类
... #省略代码
```

4. 自动生成迁移文件：使用 `alembic revision --autogenerate -m "message"` 将当前模型中的状态生成迁移文件。
5. 更新数据库：使用 `alembic upgrade head` 将刚刚生成的迁移文件，真正映射到数据库中。同理，如果要降级，那么使用 `alembic downgrade head` 。
6. 修改代码后，重复4~5的步骤。
7. 命令和参数解释：
    - init：创建一个 `alembic` 仓库。
    - revision：创建一个新的版本文件。
    - --autogenerate：自动将当前模型的修改，生成迁移脚本。
    - -m：本次迁移做了哪些修改，用户可以指定这个参数，方便回顾。
    - upgrade：将指定版本的迁移文件映射到数据库中，会执行版本文件中的 `upgrade` 函数。如果有多个迁移脚本没有被映射到数据库中，那么会执行多个迁移脚本。
    - [head]：代表最新的迁移脚本的版本号。
    - downgrade：会执行指定版本的迁移文件中的 `downgrade` 函数。
    - heads：展示head指向的脚本文件版本号。
    - history：列出所有的迁移版本及其信息。
    - current：展示当前数据库中的版本号。

另外，在你第一次执行 `upgrade` 的时候，就会在数据库中创建一个名叫 `alembic_version` 表，这个表只会有一条数据，记录当前数据库映射的是哪个版本的迁移文件。

## 经典错误：

| 错误描述 | 原因 | 解决办法 |
| --- | --- | --- |
| `FAILED: Target database is not up to date.` | 主要是 `heads` 和 `current` 不相同。 `current` 落后于 `heads` 的版本。 | 将current移动到head上。alembic upgrade head |
| `FAILED: Can't locate revision identified by '77525ee61b5b'` | 数据库中存的版本号不在迁移脚本文件中 | 删除数据库的 `alembic_version` 表中的数据，重新执行 `alembic upgrade head` |

# Flask-Migrate

在实际的开发环境中，经常会发生数据库修改的行为。一般我们修改数据库不会直接手动的去修改，而是去修改 `ORM` 对应的模型，然后再把模型映射到数据库中。这时候如果有一个工具能专门做这种事情，就显得非常有用了，而 `flask-migrate` 就是做这个事情的。 `flask-migrate` 是基于 `Alembic` 进行的一个封装，并集成到 `Flask` 中，而所有的迁移操作其实都是 `Alembic` 做的，他能跟踪模型的变化，并将变化映射到数据库中。

使用 `Flask-Migrate` 需要安装，命令如下：

```
pip install flask-migrate
```

要让 `Flask-Migrate` 能够管理 `app` 中的数据库，需要使用 `Migrate(app,db)` 来绑定 `app` 和数据库。假如现在有以下 `app` 文件：

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from constants import DB_URI
from flask_migrate import Migrate

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = DB_URI
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
db = SQLAlchemy(app)
# 绑定app和数据库
migrate = Migrate(app,db)

class User(db.Model):
    id = db.Column(db.Integer,primary_key=True)
    username = db.Column(db.String(20))

    addresses = db.relationship('Address',backref='user')

class Address(db.Model):
    id = db.Column(db.Integer,primary_key=True)
    email_address = db.Column(db.String(50))
    user_id = db.Column(db.Integer,db.ForeignKey('user.id'))

db.create_all()

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

if __name__ == '__main__':
```

```
    app.run()
```

之后，就可以在命令行中映射 `ORM` 了。要操作当前的 `flask app`，首先需要将当前的 `app` 导入到环境变量中：

```
# windows
$env:FLASK_APP='your_app.py'

#linux/unix
export FLASK_APP='your_app.py'
```

将当前的 `app` 导入到环境变量中后，接下来就是需要初始化一个迁移文件夹：

```
    flask db init
```

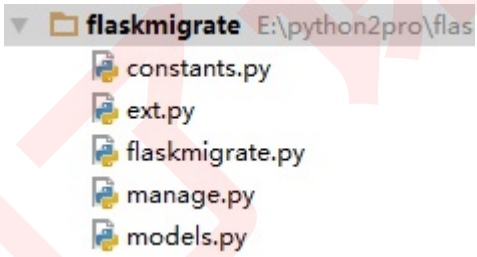
然后再把当前的模型添加到迁移文件中：

```
    flask db migrate
```

最后再把迁移文件中对应的数据库操作，真正的映射到数据库中：

```
    flask db upgrade
```

以上是通过加载当前的 `app` 到环境变量的做法，这种做法有点麻烦，每次都要设置环境变量。还有一种方法，可以直接通过 `flask-script` 的方式进行调用。现在重构之前的项目，设置为以下的目录结构：




以下对各个文件的作用进行解释：

## constants.py文件：

常量文件，用来存放数据库配置。

```
# constants.py
HOSTNAME = '127.0.0.1'
PORT = '3306'
DATABASE = 'xt_flask_migrate'
USERNAME = 'root'
```

```
PASSWORD = 'root'
DB_URI = 'mysql+mysqldb://{}:{}@{}:{}/{}'.format(USERNAME,PASSWORD,HOSTNAME,PORT,DATABA
SE)
```

## ext.py文件：

把 db 变量放到一个单独的文件，而不是放在主 app 文件。这样做的目的是为了在大型项目中如果 db 被多个模型文件引用的话，会造成 from your_app import db 这样的方式，但是往往也在 your_app.py 中也会引入模型文件定义的类，这就造成了循环引用。所以最好的办法是把它放在不依赖其他模块的独立文件中。

```
# ext.py
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

**models.py**文件：模型文件，用来存放所有的模型，并且注意，因为这里使用的是 flask-script 的方式进行模型和表的映射，因此不需要使用 db.create_all() 的方式创建数据库。

```
# models.py
from ext import db
class User(db.Model):
    id = db.Column(db.Integer,primary_key=True)
    username = db.Column(db.String(50))
    addresses = db.relationship('Address',backref='user')

    def __init__(self,username):
        self.username = username

class Address(db.Model):
    id = db.Column(db.Integer,primary_key=True)
    email_address = db.Column(db.String(50))
    user_id = db.Column(db.Integer,db.ForeignKey('user.id'))

    def __init__(self,email_address):
        self.email_address = email_address
```

## manage.py文件：

这个文件用来存放映射数据库的命令， MigrateCommand 是 flask-migrate 集成的一个命令，因此想要添加到脚本命令中，需要采用 manager.add_command('db',MigrateCommand) 的方式，以后运行 python manage.py db xxx 的命令，其实就是执行 MigrateCommand 。

```
# manage.py
from flask_migrate import Migrate,MigrateCommand
```

```
from ext import db
from flask_script import Manager
from flask import Flask
from constants import DB_URI
import models
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = DB_URI
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
db.init_app(app)
migrate = Migrate(app,db)
manager = Manager(app)
manager.add_command('db',MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

## flaskmigrate.py文件：

这个是主 `app` 文件，运行文件。并且因为 `db` 被放到另外一个文件中，所以使用 `db.init_app(app)` 的方式来绑定数据库。

```
# flaskmigrate.py
from flask import Flask
from ext import db

app = Flask(__name__)
db.init_app(app)

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

if __name__ == '__main__':
    app.run()
```

之后运行命令来初始化迁移文件：

```
python manage.py db init
```

然后运行命令来将模型的映射添加到文件中：

```
python manage.py db migrate
```

最后添加将映射文件真正的映射到数据库中：

```
python manage.py db upgrade
```

# Flask-WTF

`Flask-WTF` 是简化了 `WTForms` 操作的一个第三方库。 `WTForms` 表单的两个主要功能是验证用户提交数据的合法性以及渲染模板。当然还包括一些其他的功能： `CSRF保护` ，文件上传等。安装 `Flask-WTF` 默认也会安装 `WTForms` ，因此使用以下命令来安装 `Flask-WTF` ：

```
pip install flask-wtf
```

## 表单验证：

安装完 `Flask-WTF` 后。来看下第一个功能，就是用表单来做数据验证，现在有一个 `forms.py` 文件，然后在里面创建一个 `RegistForm` 的注册验证表单：

```
class RegistForm(Form):
    name = StringField(validators=[length(min=4,max=25)])
    email = StringField(validators=[email()])
    password = StringField(validators=[DataRequired(),length(min=6,max=10),EqualTo('con
firm')])
    confirm = StringField()
```

在这个里面指定了需要上传的参数，并且指定了验证器，比如 `name` 的长度应该在 `4-25` 之间。 `email` 必须要满足邮箱的格式。 `password` 长度必须在 `6-10` 之间，并且应该和 `confirm` 相等才能通过验证。

写完表单后，接下来就是 `regist.html` 文件：

```
    <form action="/regist/" method="POST">
        <table>
            <tr>
                <td>用户名：</td>
                <td><input type="text" name="name"></td>
            </tr>
            <tr>
                <td>邮箱：</td>
                <td><input type="email" name="email"></td>
            </tr>
            <tr>
                <td>密码：</td>
                <td><input type="password" name="password"></td>
            </tr>
            <tr>
                <td>确认密码：</td>
                <td><input type="password" name="confirm"></td>
            </tr>
```

```
            <tr>
                <td></td>
                <td><input type="submit" value="提交"></td>
            </tr>
        </table>
    </form>
```

再来看视图函数 `regist`：

```
@app.route('/regist/',methods=['POST','GET'])
def regist():
    form = RegistForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(name=form.name.data,email=form.email.data,password=form.password.da
ta)
        db.session.add(user)
        db.session.commit()
        return u'注册成功!'
    return render_template('regist.html')
```

`RegistForm` 传递的是 `request.form` 进去进行初始化，并且判断 `form.validate` 会返回用户提交的数据是否满足表单的验证。

## 渲染模板：

`form` 还可以渲染模板，让你少写了一丢丢的代码，比如重写以上例子， `RegistForm` 表单代码如下：

```
class RegistForm(Form):
    name = StringField(u'用户名：',validators=[length(min=4,max=25)])
    email = StringField(u'邮箱：'validators=[email()])
    password = StringField(u'密码：',validators=[DataRequired(),length(min=6,max=10),Eq
ualTo('confirm')])
    confirm = StringField(u'确认密码：')
```

以上增加了第一个位置参数，用来在html文件中，做标签提示作用。

在 `app` 中的视图函数中，修改为如下：

```
@app.route('/regist/',methods=['POST','GET'])
def regist():
    form = RegistForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(name=form.name.data,email=form.email.data,password=form.password.da
ta)
        db.session.add(user)
```

```
            db.session.commit()
            return u'注册成功!'
        return render_template('regist.html',form=form)
```

以上唯一的不同是在渲染模板的时候传入了 `form` 表单参数进去，这样在模板中就可以使用表单 `form` 变量了。

接下来看下 `regist.html` 文件：

```
<form action="/regist/" method="POST">
    <table>
        <tr>
            <td>{{ form.name.label }}</td>
            <td>{{ form.name() }}</td>
        </tr>
        <tr>
            <td>{{ form.email.label }}</td>
            <td>{{ form.email() }}</td>
        </tr>
        <tr>
            <td>{{ form.password.label }}</td>
            <td>{{ form.password() }}</td>
        </tr>
        <tr>
            <td>{{ form.confirm.label }}</td>
            <td>{{ form.confirm() }}</td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" value="提交"></td>
        </tr>
    </table>
</form>
```

## Field常用参数：

在使用 `Field` 的时候，经常需要传递一些参数进去，以下将对一些常用的参数进行解释：

- label（第一个参数）： `Field` 的label的文本。
- validators：验证器。
- id： `Field` 的id属性，默认不写为该属性名。
- default：默认值。
- widget：指定的 `html` 控件。

## 常用Field：

- BooleanField：布尔类型的Field，渲染出去是 `checkbox` 。
- FileField：文件上传Field。

```
# forms.py
from flask_wtf.file import FileField,FileAllowed,FileRequired
class UploadForm(FlaskForm):
    avatar = FileField(u'头像：',validators=[FileRequired(),FileAllowed([])])

# app.py
@app.route('/profile/',methods=('POST','GET'))
def profile():
    form = ProfileForm()
    if form.validate_on_submit():
        filename = secure_filename(form.avatar.data.filename)
        form.avatar.data.save(os.path.join(app.config['UPLOAD_FOLDER'],filename))
    return u'上传成功'

    return render_template('profile.html',form=form)
```

- FloatField：浮点数类型的Field，但是渲染出去的时候是 `text` 的input。
- IntegerField：整形的Field。同FloatField。
- RadioField： `radio` 类型的 `input` 。表单例子如下：

```
# form.py
class RegistrationForm(FlaskForm):
    gender = wtforms.RadioField(u'性别：',validators=[DataRequired()])
```

模板文件代码如下：

```
<tr>
    <td>
        {{ form.gender.label }}
    </td>
    <td>
        {% for gender in form.gender %}
            {{ gender.label }}
            {{ gender }}
        {% endfor %}
    </td>
</tr>
```

`app.py` 文件的代码如下，给 `gender` 添加了 `choices` ：

```
@app.route('/register/',methods=['POST','GET'])
def register():
    form = RegistrationForm()
    form.gender.choices = [('1',u'男'),('2',u'女')]
    if form.validate_on_submit():
        return u'success'

    return render_template('register.html',form=form)
```

- SelectField：类似于 `RadioField` 。看以下示例：

```
# forms.py
class ProfileForm(FlaskForm):
    language = wtforms.SelectField('Programming Language',choices=[('cpp','C++'),('py','python'),('text','Plain Text')],validators=[DataRequired()])
```

再来看 `app.py` 文件：

```
@app.route('/profile/',methods=('POST','GET'))
def profile():
    form = ProfileForm()
    if form.validate_on_submit():
        print form.language.data
        return u'上传成功'
    return render_template('profile.html',form=form)
```

模板文件为：

```
<form action="/profile/" method="POST">
    {{ form.csrf_token }}
    {{ form.language.label }}
    {{ form.language() }}
    <input type="submit">
</form>
```

- StringField：渲染到模板中的类型为 `<input type='text'>` ，并且是最基本的文本验证。
- PasswordField：渲染出来的是一个 `password` 的 `input` 标签。
- TextAreaField：渲染出来的是一个 `textarea` 。

## 常用的验证器：

数据发送过来，经过表单验证，因此需要验证器来进行验证，以下对一些常用的内置验证器进行讲解：

- Email：验证上传的数据是否为邮箱。
- EqualTo：验证上传的数据是否和另外一个字段相等，常用的就是密码和确认密码两个字段是否相等。
- InputRequired：原始数据的需要验证。如果不是特殊情况，应该使用 `InputRequired` 。
- Length：长度限制，有min和max两个值进行限制。
- NumberRange：数字的区间，有min和max两个值限制，如果处在这两个数字之间则满足。
- Regexp：自定义正则表达式。
- URL：必须要是 `URL` 的形式。
- UUID：验证 `UUID` 。

## 自定义验证字段：

使用 `validate_fieldname(self,field)` 可以对某个字段进行更加详细的验证，如下：

```
class ProfileForm(FlaskForm):
    name = wtforms.StringField('name',[validators.InputRequired()])
    def validate_name(self,field):
        if len(field.data) > 5:
            raise wtforms.ValidationError(u'超过5个字符')
```

## CSRF保护：

在flask的表单中，默认是开启了 `csrf` 保护功能的，如果你想关闭表单的 `csrf` 保护，可以在初始化表单的时候传递 `csrf_enabled=False` 进去来关闭 `csrf` 保护。如果你想关闭这种默认的行为。如果你想在没有表单存在的请求视图函数中也添加 `csrf` 保护，可以开启全局的 `csrf` 保护功能：

```
csrf = CsrfProtect()
csrf.init_app(app)
```

或者是针对某一个视图函数，使用 `csrf.protect` 装饰器来开启 `csrf` 保护功能。并且如果已经开启了全局的 `csrf` 保护，想要关闭某个视图函数的 `csrf` 保护功能，可以使用 `csrf.exempt` 装饰器来取消本视图函数的保护功能。

## AJAX的CSRF保护：

在 `AJAX` 中要使用 `csrf` 保护，则必须手动的添加 `X-CSRFToken` 到 `Header` 中。但是 `CSRF` 从哪里来，还是需要通过模板给渲染，而 `Flask` 比较推荐的方式是在 `meta` 标签中渲染 `csrf` ，如下：

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

如果要发送 `AJAX` 请求，则在发送之前要添加 `CSRF` ,代码如下（使用了jQuery）：

```
var csrftoken = $('meta[name=csrf-token]').attr('content')
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!/^(GET|HEAD|OPTIONS|TRACE)$/i.test(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken)
        }
    }
})
```

# cookie和session

1. cookie：在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。`cookie`的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的`cookie`数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。`cookie`存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用`cookie`只能存储一些小量的数据。

2. session: session和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，`cookie`是存储在本地浏览器，`session`是一个思路、一个概念、一个服务器存储授权信息的解决方案，不同的服务器，不同的框架，不同的语言有不同的实现。虽然实现不一样，但是他们的目的都是服务器为了方便存储数据的。`session`的出现，是为了解决`cookie`存储数据不安全的问题的。

3. cookie和session结合使用：`web`开发发展至今，`cookie`和`session`的使用已经出现了一些非常成熟的方案。在如今的市场或者企业里，一般有两种存储方式：

    - 存储在服务端：通过`cookie`存储一个`session_id`，然后具体的数据则是保存在`session`中。如果用户已经登录，则服务器会在`cookie`中保存一个`session_id`，下次再次请求的时候，会把该`session_id`携带上来，服务器根据`session_id`在`session`库中获取用户的`session`数据。就能知道该用户到底是谁，以及之前保存的一些状态信息。这种专业术语叫做`server side session`。存储在服务器的数据会更加的安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，一些`session`信息还是绰绰有余的。
    - 将`session`数据加密，然后存储在`cookie`中。这种专业术语叫做`client side session`。`flask`采用的就是这种方式，但是也可以替换成其他形式。

# flask中使用cookie和session

1. cookies：在`Flask`中操作`cookie`，是通过`response`对象来操作，可以在`response`返回之前，通过`response.set_cookie`来设置，这个方法有以下几个参数需要注意：
    - key：设置的cookie的key。
    - value：key对应的value。
    - max_age：改cookie的过期时间，如果不设置，则浏览器关闭后就会自动过期。
    - expires：过期时间，应该是一个`datetime`类型。
    - domain：该cookie在哪个域名中有效。一般设置子域名，比如`cms.example.com`。
    - path：该cookie在哪个路径下有效。
2. session：`Flask`中的`session`是通过`from flask import session`。然后添加值key和value进去即可。并且，`Flask`中的`session`机制是将`session`信息加密，然后存储

在 `cookie` 中。专业术语叫做 `client side session` 。

# Flask上下文

`Flask` 项目中有两个上下文，一个是应用上下文（app），另外一个是请求上下文（request）。请求上下文 `request` 和应用上下文 `current_app` 都是一个全局变量。所有请求都共享的。 `Flask` 有特殊的机制可以保证每次请求的数据都是隔离的，即A请求所产生的数据不会影响到B请求。所以可以直接导入 `request` 对象，也不会被一些脏数据影响了，并且不需要在每个函数中使用request的时候传入 `request` 对象。这两个上下文具体的实现方式和原理可以没必要详细了解。只要了解这两个上下文的四个属性就可以了：

- request：请求上下文上的对象。这个对象一般用来保存一些请求的变量。比如 `method` 、 `args` 、 `form` 等。
- session：请求上下文上的对象。这个对象一般用来保存一些会话信息。
- current_app：返回当前的app。
- g：应用上下文上的对象。处理请求时用作临时存储的对象。

## 常用的钩子函数

- before_first_request：处理第一次请求之前执行。例如以下代码：

```
@app.before_first_request
def first_request():
    print 'first time request'
```

- before_request：在每次请求之前执行。通常可以用这个装饰器来给视图函数增加一些变量。例如以下代码：

```
@app.before_request
def before_request():
    if not hasattr(g,'user'):
        setattr(g,'user','xxxx')
```

- teardown_appcontext：不管是否有异常，注册的函数都会在每次请求之后执行。

```
@app.teardown_appcontext
def teardown(exc=None):
    if exc is None:
        db.session.commit()
    else:
        db.session.rollback()
    db.session.remove()
```

- template_filter：在使用 `Jinja2` 模板的时候自定义过滤器。比如可以增加一个 `upper` 的过滤器（当然Jinja2已经存在这个过滤器，本示例只是为了演示作用）：

```
@app.template_filter
def upper_filter(s):
    return s.upper()
```

- context_processor：上下文处理器。返回的字典中的键可以在模板上下文中使用。例如：

```
@app.context_processor
return {'current_user':'xxx'}
```

- errorhandler：errorhandler接收状态码，可以自定义返回这种状态码的响应的处理方法。例如：

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist',404
```

# flask信号：

## 安装：

`flask` 中的信号使用的是一个第三方插件，叫做 `blinker` 。通过 `pip list` 看一下，如果没有安装，通过以下命令即可安装 `blinker` ：

```
pip install blinker
```

## 内置信号：

`flask` 内置集中常用的信号：

1. `flask.template_rendered` ：模版渲染完毕后发送，示例如下：

```
from flask import template_rendered
def log_template_renders(sender,template,context,*args):
    print 'sender:',sender
    print 'template:',template
    print 'context:',context

template_rendered.connect(log_template_renders,app)
```

2. `flask.request_started` ：请求开始之前，在到达视图函数之前发送，订阅者可以调用 `request` 之类的标准全局代理访问请求。示例如下：

```
def log_request_started(sender,**extra):
    print 'sender:',sender
    print 'extra:',extra
request_started.connect(log_request_started,app)
```

3. `flask.request_finished` ：请求结束时，在响应发送给客户端之前发送，可以传递 `response` ，示例代码如下：

```
def log_request_finished(sender,response,*args):
    print 'response:',response
request_finished.connect(log_request_finished,app)
```

4. `flask.got_request_exception` ：在请求过程中抛出异常时发送，异常本身会通过 `exception` 传递到订阅的函数。示例代码如下：

```
    def log_exception_finished(sender,exception,*args):
        print 'sender:',sender
        print type(exception)
    got_request_exception.connect(log_exception_finished,app)
```

5. `flask.request_tearing_down`：请求被销毁的时候发送，即使在请求过程中发生异常，也会发送，示例代码如下：

```
    def log_request_tearing_down(sender,**kwargs):
        print 'coming...'
    request_tearing_down.connect(log_request_tearing_down,app)
```

6. `flask.appcontext_tearing_down`：在应用上下文销毁的时候发送，它总是会被调用，即使发生异常。示例代码如下：

```
    def log_appcontext_tearing_down(sender,**kwargs):
        print 'coming...'
    appcontext_tearing_down.connect(log_appcontext_tearing_down,app)
```

## 自定义信号：

自定义信号分为3步，第一是定义一个信号，第二是监听一个信号，第三是发送一个信号。以下将对这三步进行讲解：

1. 定义信号：定义信号需要使用到 `blinker` 这个包的 `Namespace` 类来创建一个命名空间。比如定义一个在访问了某个视图函数的时候的信号。示例代码如下：

```
    from blinker import Namespace

    mysignal = Namespace()
    visit_signal = mysignal.signal('visit-signal')
```

2. 监听信号：监听信号使用 `singal` 对象的 `connect` 方法，在这个方法中需要传递一个函数，用来接收以后监听到这个信号该做的事情。示例代码如下：

```
  def visit_func(sender,username):
    print(sender)
    print(username)

  mysignal.connect(visit_func)
```

3. 发送信号：发送信号使用 `singal` 对象的 `send` 方法，这个方法可以传递一些其他参数过去。示例代码如下：

```
mysignal.send(username='zhiliao')
```

# Restful API规范

`restful api`是用于在前端与后台进行通信的一套规范。使用这个规范可以让前后端开发变得更加轻松。以下将讨论这套规范的一些设计细节。

## 协议：

采用`http`或者`https`协议。

## 数据传输格式：

数据之间传输的格式应该都使用`json`，而不使用`xml`。

## url链接：

url链接中，不能有动词，只能有名词。并且对于一些名词，如果出现复数，那么应该在后面加`s`。

比如：获取文章列表，应该使用`/articles/`，而不应该使用/get_article/

## HTTP请求的方法：

1. `GET`：从服务器上获取资源。
2. `POST`：在服务器上新创建一个资源。
3. `PUT`：在服务器上更新资源。（客户端提供所有改变后的数据）
4. `PATCH`：在服务器上更新资源。（客户端只提供需要改变的属性）
5. `DELETE`：从服务器上删除资源。

示例如下：

- GET /users/：获取所有用户。
- POST /user/：新建一个用户。
- GET /user/id/：根据id获取一个用户。
- PUT /user/id/：更新某个id的用户的信息（需要提供用户的所有信息）。
- PATCH /user/id/：更新某个id的用户信息（只需要提供需要改变的信息）。
- DELETE /user/id/：删除一个用户。

## 状态码：

| 状态码 | 原生描述 | 描述 |
| --- | --- | --- |
| 200 | OK | 服务器成功响应客户端的请求。 |

| | | |
|---|---|---|
| 400 | INVALID REQUEST | 用户发出的请求有错误，服务器没有进行新建或修改数据的操作 |
| 401 | Unauthorized | 用户没有权限访问这个请求 |
| 403 | Forbidden | 因为某些原因禁止访问这个请求 |
| 404 | NOT FOUND | 用户发送的请求的url不存在 |
| 406 | NOT Acceptable | 用户请求不被服务器接收（比如服务器期望客户端发送某个字段，但是没有发送）。 |
| 500 | Internal server error | 服务器内部错误，比如出现了bug |

# Flask-Restful插件

## 介绍：

`Flask-Restful` 是一个专门用来写 `restful api` 的一个插件。使用他可以快速的集成 `restful api` 功能。在 `app` 的后台以及纯 `api` 的后台中，这个插件可以帮助我们节省很多时间。当然，如果在普通的网站中，这个插件就显得有些鸡肋了，因为在普通的网页开发中，是需要去渲染HTML代码的，而 `Flask-Restful` 在每个请求中都是返回 `json` 格式的数据。

## 安装：

`Flask-Restful` 需要在 `Flask 0.8` 以上的版本，在 `Python2.6` 或者 `Python3.3` 上运行。通过 `pip install flask-restful` 即可安装。

## 定义Restful的视图：

如果使用 `Flask-Restful` ，那么定义视图函数的时候，就要继承自 `flask_restful.Resource` 类，然后再根据当前请求的 `method` 来定义相应的方法。比如期望客户端是使用 `get` 方法发送过来的请求，那么就定义一个 `get` 方法。类似于 `MethodView` 。示例代码如下：

```
from flask import Flask,render_template,url_for
from flask_restful import Api,Resource

app = Flask(__name__)
# 用Api来绑定app
api = Api(app)

class IndexView(Resource):
    def get(self):
        return {"username":"zhiliao"}

api.add_resource(IndexView,'/',endpoint='index')
```

注意事项：

1. `endpoint` 是用来给 `url_for` 反转 `url` 的时候指定的。如果不写 `endpoint` ，那么将会使用视图的名字的小写来作为 `endpoint` 。
2. `add_resource` 的第二个参数是访问这个视图函数的 `url` ，这个 `url` 可以跟之前的 `route` 一样，可以传递参数。并且还有一点不同的是，这个方法可以传递多个 `url` 来指定这个视图函数。

## 参数解析：

`Flask-Restful` 插件提供了类似 `WTForms` 来验证提交的数据是否合法的包，叫做 `reqparse` 。以下是基本用法：

```
parser = reqparse.RequestParser()
parser.add_argument('username',type=str,help='请输入用户名')
args = parser.parse_args()
```

`add_argument` 可以指定这个字段的名字，这个字段的数据类型等。以下将对这个方法的一些参数做详细讲解：

1. `default` ：默认值，如果这个参数没有值，那么将使用这个参数指定的值。
2. `required` ：是否必须。默认为False，如果设置为 `True` ，那么这个参数就必须提交上来。
3. `type` ：这个参数的数据类型，如果指定，那么将使用指定的数据类型来强制转换提交上来的值。
4. `choices` ：选项。提交上来的值只有满足这个选项中的值才符合验证通过，否则验证不通过。
5. `help` ：错误信息。如果验证失败后，将会使用这个参数指定的值作为错误信息。
6. `trim` ：是否要去掉前后的空格。

其中的 `type` ，可以使用 `python` 自带的一些数据类型，也可以使用 `flask_restful.inputs` 下的一些特定的数据类型来强制转换。比如一些常用的：

1. `url` ：会判断这个参数的值是否是一个url，如果不是，那么就会抛出异常。
2. `regex` ：正则表达式。
3. `date` ：将这个字符串转换为 `datetime.date` 数据类型。如果转换不成功，则会抛出一个异常。

## 输出字段：

对于一个视图函数，你可以指定好一些字段用于返回。以后可以使用ORM模型或者自定义的模型的时候，他会自动的获取模型中的相应的字段，生成 `json` 数据，然后再返回给客户端。这其中需要导入 `flask_restful.marshal_with` 装饰器。并且需要写一个字典，来指示需要返回的字段，以及该字段的数据类型。示例代码如下：

```
class ProfileView(Resource):
    resource_fields = {
        'username': fields.String,
        'age': fields.Integer,
        'school': fields.String
    }

    @marshal_with(resource_fields)
    def get(self,user_id):
        user = User.query.get(user_id)
        return user
```

在 `get` 方法中，返回 `user` 的时候， `flask_restful` 会自动的读取 `user` 模型上的 `username` 以及 `age` 还有 `school` 属性。组装成一个 `json` 格式的字符串返回给客户端。

### 重命名属性：

很多时候你面向公众的字段名称是不同于内部的属性名。使用 attribute可以配置这种映射。比如现在想要返回 `user.school` 中的值，但是在返回给外面的时候，想以 `education` 返回回去，那么可以这样写：

```
resource_fields = {
    'education': fields.String(attribute='school')
}
```

### 默认值：

在返回一些字段的时候，有时候可能没有值，那么这时候可以在指定 `fields` 的时候给定一个默认值，示例代码如下：

```
resource_fields = {
    'age': fields.Integer(default=18)
}
```

### 复杂结构：

有时候想要在返回的数据格式中，形成比较复杂的结构。那么可以使用一些特殊的字段来实现。比如要在一个字段中放置一个列表，那么可以使用 `fields.List` ，比如在一个字段下面又是一个字典，那么可以使用 `fields.Nested` 。以下将讲解下复杂结构的用法：

```
class ProfileView(Resource):
    resource_fields = {
        'username': fields.String,
        'age': fields.Integer,
        'school': fields.String,
        'tags': fields.List(fields.String),
        'more': fields.Nested({
            'signature': fields.String
        })
    }
```

# memcached

## 什么是memcached：

1. `memcached` 之前是 `danga` 的一个项目，最早是为LiveJournal服务的，当初设计师为了加速LiveJournal访问速度而开发的，后来被很多大型项目采用。官网是 `www.danga.com` 或者是 `memcached.org` 。
2. `Memcached` 是一个高性能的分布式的内存对象缓存系统，全世界有不少公司采用这个缓存项目来构建大负载的网站，来分担数据库的压力。 `Memcached` 是通过在内存里维护一个统一的巨大的hash表， `memcached` 能存储各种各样的数据，包括图像、视频、文件、以及数据库检索的结果等。简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。
3. 哪些情况下适合使用 `Memcached` ：存储验证码（图形验证码、短信验证码）、登录session等所有不是至关重要的数据。

## 安装和启动 `memcached` ：

1. windows：
   - 安装： `memcached.exe -d install` 。
   - 启动： `memcached.exe -d start` 。
2. linux（ubuntu）：
   - 安装： `sudo apt install memcached`
   - 启动：

```
cd /usr/local/memcached/bin
./memcached -d start
```

3. 可能出现的问题：
   - 提示你没有权限：在打开cmd的时候，右键使用管理员身份运行。
   - 提示缺少 `pthreadGC2.dll` 文件：将 `pthreadGC2.dll` 文件拷贝到 `windows/System32` .
   - 不要放在含有中文的路径下面。
4. 启动 `memcached` ：
   - `-d` ：这个参数是让 `memcached` 在后台运行。
   - `-m` ：指定占用多少内存。以 `M` 为单位，默认为 `64M` 。
   - `-p` ：指定占用的端口。默认端口是 `11211` 。
   - `-l` ：别的机器可以通过哪个ip地址连接到我这台服务器。如果是通过 `service memcached start` 的方式，那么只能通过本机连接。如果想要让别的机器连接，就必须设置 `-l 0.0.0.0` 。

如果想要使用以上参数来指定一些配置信息，那么不能使用 `service memcached start` ，而应该使用 `/usr/bin/memcached` 的方式来运行。比如 `/usr/bin/memcached -u memcache -m 1024 -p 11222 start` 。

## `telnet` 操作 `memcached` ：

`telnet ip地址 [11211]`

1. 添加数据：
   - `set` ：
     - 语法：

       ```
       set key flas(是否压缩) timeout value_length
       value
       ```

     - 示例：

       ```
       set username 0 60 7
       zhiliao
       ```

   - `add` ：
     - 语法：

       ```
       add key flas(0) timeout value_length
       value
       ```

     - 示例：

       ```
       add username 0 60 7
       xiaotuo
       ```

     `set` 和 `add` 的区别： `add` 是只负责添加数据，不会去修改数据。如果添加的数据的 `key` 已经存在了，则添加失败，如果添加的 `key` 不存在，则添加成功。而 `set` 不同，如果 `memcached` 中不存在相同的 `key` ，则进行添加，如果存在，则替换。
2. 获取数据：
   - 语法：

     ```
     get key
     ```

   - 示例：

     ```
     get username
     ```

3. 删除数据：
    - 语法：
    ```
    delete key
    ```
    - 示例：
    ```
    delete username
    ```
    - `flush_all`：删除`memcached`中的所有数据。
4. 查看`memcached`的当前状态：
    - 语法：`stats`。

## 通过`python`操作`memcached`：

1. 安装：`python-memcached`：`pip install python-memcached`。
2. 建立连接：
```
import memcache
mc = memcache.Client(['127.0.0.1:11211','192.168.174.130:11211'],debug=True)
```
3. 设置数据：
```
mc.set('username','hello world',time=60*5)
mc.set_multi({'email':'xxx@qq.com','telphone':'111111'},time=60*5)
```
4. 获取数据：
```
mc.get('telphone')
```
5. 删除数据：
```
mc.delete('email')
```
6. 自增长：
```
mc.incr('read_count')
```
7. 自减少：
```
mc.decr('read_count')
```

## **memcached**的安全性：

`memcached` 的操作不需要任何用户名和密码，只需要知道 `memcached` 服务器的ip地址和端口号即可。因此 `memcached` 使用的时候尤其要注意他的安全性。这里提供两种安全的解决方案。分别来进行讲解：

1. 使用 `-l` 参数设置为只有本地可以连接：这种方式，就只能通过本机才能连接，别的机器都不能访问，可以达到最好的安全性。
2. 使用防火墙，关闭 `11211` 端口，外面也不能访问。

```
ufw enable # 开启防火墙
ufw disable # 关闭防火墙
ufw default deny # 防火墙以禁止的方式打开，默认是关闭那些没有开启的端口
ufw deny 端口号 # 关闭某个端口
ufw allow 端口号 # 开启某个端口
```

# redis教程：

## 概述

`redis` 是一种 `nosql` 数据库,他的数据是保存在内存中，同时 `redis` 可以定时把内存数据同步到磁盘，即可以将数据持久化，并且他比 `memcached` 支持更多的数据结构( `string` ，`list列表[队列和栈]` ，`set[集合]` ，`sorted set[有序集合]` ，`hash(hash表)` )。相关参考文档：http://redisdoc.com/index.html

# redis使用场景：

1. 登录会话存储：存储在 `redis` 中，与 `memcached` 相比，数据不会丢失。
2. 排行版/计数器：比如一些秀场类的项目，经常会有一些前多少名的主播排名。还有一些文章阅读量的技术，或者新浪微博的点赞数等。
3. 作为消息队列：比如 `celery` 就是使用 `redis` 作为中间人。
4. 当前在线人数：还是之前的秀场例子，会显示当前系统有多少在线人数。
5. 一些常用的数据缓存：比如我们的 `BBS` 论坛，板块不会经常变化的，但是每次访问首页都要从 `mysql` 中获取，可以在 `redis` 中缓存起来，不用每次请求数据库。
6. 把前200篇文章缓存或者评论缓存：一般用户浏览网站，只会浏览前面一部分文章或者评论，那么可以把前面200篇文章和对应的评论缓存起来。用户访问超过的，就访问数据库，并且以后文章超过200篇，则把之前的文章删除。
7. 好友关系：微博的好友关系使用 `redis` 实现。
8. 发布和订阅功能：可以用来做聊天软件。

## `redis` 和 `memcached` 的比较：

| | memcached | redis |
|---|---|---|
| 类型 | 纯内存数据库 | 内存磁盘同步数据库 |
| 数据类型 | 在定义value时就要固定数据类型 | 不需要 |
| 虚拟内存 | 不支持 | 支持 |
| 过期策略 | 支持 | 支持 |
| 存储数据安全 | 不支持 | 可以将数据同步到dump.db中 |
| 灾难恢复 | 不支持 | 可以将磁盘中的数据恢复到内存中 |
| 分布式 | 支持 | 主从同步 |
| 订阅与发布 | 不支持 | 支持 |

## `redis` 在 `ubuntu` 系统中的安装与启动

1. 安装：

```
sudo apt-get install redis-server
```

2. 卸载：

```
sudo apt-get purge --auto-remove redis-server
```

3. 启动：`redis` 安装后，默认会自动启动，可以通过以下命令查看：

```
ps aux|grep redis
```

如果想自己手动启动，可以通过以下命令进行启动：

```
sudo service redis-server start
```

4. 停止：

```
sudo service redis-server stop
```

## 对 `redis` 的操作

对 `redis` 的操作可以用两种方式，第一种方式采用 `redis-cli` ，第二种方式采用编程语言，比如 `Python` 、 `PHP` 和 `JAVA` 等。

1. 使用 `redis-cli` 对 `redis` 进行字符串操作：
2. 启动 `redis` ：

```
sudo service redis-server start
```

3. 连接上 `redis-server` ：

```
redis-cli -h [ip] -p [端口]
```

4. 添加：

```
set key value
如：
set username xiaotuo
```

将字符串值 `value` 关联到 `key` 。如果 `key` 已经持有其他值， `set` 命令就覆写旧值，无视其类型。并且默认的过期时间是永久，即永远不会过期。

5. 删除：

```
del key
如：
del username
```

6. 设置过期时间：

```
expire key timeout(单位为秒)
```

也可以在设置值的时候，一同指定过期时间：

```
set key value EX timeout
或：
setex key timeout value
```

7. 查看过期时间：

```
ttl key
如：
ttl username
```

8. 查看当前 `redis` 中的所有 `key` ：

```
keys *
```

9. 列表操作：
   - 在列表左边添加元素：

     ```
     lpush key value
     ```

     将值 `value` 插入到列表 `key` 的表头。如果 `key` 不存在，一个空列表会被创建并执行 `lpush` 操作。当 `key` 存在但不是列表类型时，将返回一个错误。

   - 在列表右边添加元素：

     ```
     rpush key value
     ```

将值value插入到列表key的表尾。如果key不存在，一个空列表会被创建并执行RPUSH操作。当key存在但不是列表类型时，返回一个错误。

- 查看列表中的元素：

```
lrange key start stop
```

返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定,如果要左边的第一个到最后的一个 `lrange key 0 -1` 。

- 移除列表中的元素：
  - 移除并返回列表 `key` 的头元素：

```
lpop key
```

  - 移除并返回列表的尾元素：

```
rpop key
```

  - 移除并返回列表 `key` 的中间元素：

```
lrem key count value
```

将删除 `key` 这个列表中， `count` 个值为 `value` 的元素。

- 指定返回第几个元素：

```
lindex key index
```

将返回 `key` 这个列表中，索引为 `index` 的这个元素。

- 获取列表中的元素个数：

```
llen key
如：
llen languages
```

- 删除指定的元素：

```
lrem key count value
如：
lrem languages 0 php
```

根据参数 count 的值，移除列表中与参数 value 相等的元素。 `count` 的值可以是以下几种：

- count > 0：从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count` 。
- count < 0：从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。
- count = 0：移除表中所有与 `value` 相等的值。

10. `set` 集合的操作：

    - 添加元素：

      ```
      sadd set value1 value2....
      如：
      sadd team xiaotuo datuo
      ```

    - 查看元素：

      ```
      smembeers set
      如：
      smembers team
      ```

    - 移除元素：

      ```
      srem set member...
      如：
      srem team xiaotuo datuo
      ```

    - 查看集合中的元素个数：

      ```
      scard set
      如：
      scard team1
      ```

    - 获取多个集合的交集：

      ```
      sinter set1 set2
      如：
      sinter team1 team2
      ```

    - 获取多个集合的并集：

      ```
      sunion set1 set2
      如：
      sunion team1 team2
      ```

    - 获取多个集合的差集：

```
sdiff set1 set2
如：
sdiff team1 team2
```

11. `hash` 哈希操作：

    - 添加一个新值：

      ```
      hset key field value
      如：
      hset website baidu baidu.com
      ```

      将哈希表 `key` 中的域 `field` 的值设为 `value` 。
      如果 `key` 不存在，一个新的哈希表被创建并进行 `HSET` 操作。如果域 `field` 已经存在于哈希表中，旧值将被覆盖。

    - 获取哈希中的 `field` 对应的值：

      ```
      hget key field
      如：
      hget website baidu
      ```

    - 删除 `field` 中的某个 `field` ：

      ```
      hdel key field
      如：
      hdel website baidu
      ```

    - 获取某个哈希中所有的 `field` 和 `value` ：

      ```
      hgetall key
      如：
      hgetall website
      ```

    - 获取某个哈希中所有的 `field` ：

      ```
      hkeys key
      如：
      hkeys website
      ```

    - 获取某个哈希中所有的值：

      ```
      hvals key
      ```

```
如：
hvals website
```

- 判断哈希中是否存在某个 `field`：

```
hexists key field
如：
hexists website baidu
```

- 获取哈希中总共的键值对：

```
hlen field
如：
hlen website
```

12. 事务操作：Redis事务可以一次执行多个命令，事务具有以下特征：
    - 隔离操作：事务中的所有命令都会序列化、按顺序地执行，不会被其他命令打扰。
    - 原子操作：事务中的命令要么全部被执行，要么全部都不执行。
    - 开启一个事务：

```
multi
```

以后执行的所有命令，都在这个事务中执行的。

- 执行事务：

```
exec
```

会将在 `multi` 和 `exec` 中的操作一并提交。

- 取消事务：

```
discard
```

会将 `multi` 后的所有命令取消。

- 监视一个或者多个 `key`：

```
watch key...
```

监视一个(或多个)key，如果在事务执行之前这个(或这些) key被其他命令所改动，那么事务将被打断。

- 取消所有 `key` 的监视：

```
unwatch
```

13. 发布/订阅操作：

- 给某个频道发布消息：

```
publish channel message
```

- 订阅某个频道的消息：

```
subscribe channel
```

14. 持久化： `redis` 提供了两种数据备份方式，一种是 `RDB` ，另外一种是 `AOF` ，以下将详细介绍这两种备份策略：

| | RDB | AOF | | --- | --- | --- | | 开启关闭 | 开启：默认开启。关闭：把配置文件中所有的save都注释，就是关闭了。 | 开启：在配置文件中 `appendonly yes` 即开启了 `aof` ，为 `no` 关闭。 | | 同步机制 | 可以指定某个时间内发生多少个命令进行同步。比如1分钟内发生了2次命令，就做一次同步。 | 每秒同步或者每次发生命令后同步 | | 存储内容 | 存储的是redis里面的具体的值 | 存储的是执行的更新数据的操作命令 | | 存储文件的路径 | 根据dir以及dbfilename来指定路径和具体的文件名 | 根据dir以及appendfilename来指定具体的路径和文件名 | | 优点 | （1）存储数据到文件中会进行压缩，文件体积比aof小。（2）因为存储的是redis具体的值，并且会经过压缩，因此在恢复的时候速度比AOF快。（3）非常适用于备份。 | （1）AOF的策略是每秒钟或者每次发生写操作的时候都会同步，因此即使服务器故障，最多只会丢失1秒的数据。（2）AOF存储的是Redis命令，并且是直接追加到aof文件后面，因此每次备份的时候只要添加新的数据进去就可以了。（3）如果AOF文件比较大了，那么Redis会进行重写，只保留最小的命令集合。 | | 缺点 | （1）RDB在多少时间内发生了多少写操作的时候就会出发同步机制，因为采用压缩机制，RDB在同步的时候都重新保存整个Redis中的数据，因此你一般会设置在最少5分钟才保存一次数据。在这种情况下，一旦服务器故障，会造成5分钟的数据丢失。（2）在数据保存进RDB的时候，Redis会fork出一个子进程用来同步，在数据量比较大的时候，可能会非常耗时。 | （1）AOF文件因为没有压缩，因此体积比RDB大。 （2）AOF是在每秒或者每次写操作都进行备份，因此如果并发量比较大，效率可能有点慢。（3）AOF文件因为存储的是命令，因此在灾难恢复的时候Redis会重新运行AOF中的命令，速度不及RDB。 | | 更多 | http://redisdoc.com/topic/persistence.html#redis | |

15. 安全：在配置文件中，设置 `requirepass password` ，那么客户端连接的时候，需要使用密码：

```
> redis-cli -p 127.0.0.1 -p 6379
redis> set username xxx
(error) NOAUTH Authentication required.
redis> auth password
redis> set username xxx
```

```
OK
```

## Python操作redis

1. 安装 `python-redis` ：

```
pip install redis
```

2. 新建一个文件比如 `redis_test.py` ，然后初始化一个 `redis` 实例变量，并且在 `ubuntu` 虚拟机中开启 `redis` 。比如虚拟机的 `ip` 地址为 `192.168.174.130` 。示例代码如下：

```
# 从redis包中导入Redis类
from redis import Redis
# 初始化redis实例变量
xtredis = Redis(host='192.168.174.130',port=6379)
```

3. 对字符串的操作：操作 `redis` 的方法名称，跟之前使用 `redis-cli` 一样，现就一些常用的来做个简单介绍，示例代码如下(承接以上的代码)：

```
# 添加一个值进去，并且设置过期时间为60秒，如果不设置，则永远不会过期
xtredis.set('username','xiaotuo',ex=60)
# 获取一个值
xtredis.get('username')
# 删除一个值
xtredis.delete('username')
# 给某个值自增1
xtredis.set('read_count',1)
xtredis.incr('read_count')  # 这时候read_count变为2
# 给某个值减少1
xtredis.decr('read_count') # 这时候read_count变为1
```

4. 对列表的操作：同字符串操作，所有方法的名称跟使用 `redis-cli` 操作是一样的：

```
# 给languages这个列表往左边添加一个python
xtredis.lpush('languages','python')
# 给languages这个列表往左边添加一个php
xtredis.lpush('languages','php')
# 给languages这个列表往左边添加一个javascript
xtredis.lpush('languages','javascript')

# 获取languages这个列表中的所有值
print xtredis.lrange('languages',0,-1)
> ['javascript','php','python']
```

5. 对集合的操作：

```
# 给集合team添加一个元素xiaotuo
xtredis.sadd('team','xiaotuo')
# 给集合team添加一个元素datuo
xtredis.sadd('team','datuo')
# 给集合team添加一个元素slice
xtredis.sadd('team','slice')

# 获取集合中的所有元素
xtredis.smembers('team')
> ['datuo','xiaotuo','slice'] # 无序的
```

6. 对哈希( `hash` )的操作：

```
# 给website这个哈希中添加baidu
xtredis.hset('website','baidu','baidu.com')
# 给website这个哈希中添加google
xtredis.hset('website','google','google.com')

# 获取website这个哈希中的所有值
print xtredis.hgetall('website')
> {"baidu":"baidu.com","google":"google.com"}
```

7. 事务(管道)操作： `redis` 支持事务操作，也即一些操作只有统一完成，才能算完成。否则都执行失败，用 `python` 操作 `redis` 也是非常简单，示例代码如下：

```
# 定义一个管道实例
pip = xtredis.pipeline()
# 做第一步操作，给BankA自增长1
pip.incr('BankA')
# 做第二步操作，给BankB自减少1
pip.desc('BankB')
# 执行事务
pip.execute()
```

以上便展示了 `python-redis` 的一些常用方法，如果想深入了解其他的方法，可以参考 `python-redis` 的源代码（查看源代码 `pycharm` 快捷键提示：把鼠标光标放在 `import Redis` 的 `Redis` 上，然后按 `ctrl+b` 即可进入）。

# CSRF攻击：

## CSRF攻击概述：

CSRF（Cross Site Request Forgery, 跨站域请求伪造）是一种网络的攻击方式，它在 2007 年曾被列为互联网 20 大安全隐患之一。其他安全隐患，比如 SQL 脚本注入，跨站域脚本攻击等在近年来已经逐渐为众人熟知，很多网站也都针对他们进行了防御。然而，对于大多数人来说，CSRF 却依然是一个陌生的概念。即便是大名鼎鼎的 Gmail, 在 2007 年底也存在着 CSRF 漏洞，从而被黑客攻击而使 Gmail 的用户造成巨大的损失。

## CSRF攻击原理：

网站是通过 `cookie` 来实现登录功能的。而 `cookie` 只要存在浏览器中，那么浏览器在访问这个 `cookie` 的服务器的时候，就会自动的携带 `cookie` 信息到服务器上去。那么这时候就存在一个漏洞了，如果你访问了一个别有用心或病毒网站，这个网站可以在网页源代码中插入js代码，使用js代码给其他服务器发送请求（比如ICBC的转账请求）。那么因为在发送请求的时候，浏览器会自动的把 `cookie` 发送给对应的服务器，这时候相应的服务器（比如ICBC网站），就不知道这个请求是伪造的，就被欺骗过去了。从而达到在用户不知情的情况下，给某个服务器发送了一个请求（比如转账）。

## 防御CSRF攻击：

CSRF攻击的要点就是在向服务器发送请求的时候，相应的 `cookie` 会自动的发送给对应的服务器。造成服务器不知道这个请求是用户发起的还是伪造的。这时候，我们可以在用户每次访问有表单的页面的时候，在网页源代码中加一个随机的字符串叫做 `csrf_token` ，在 `cookie` 中加一个也加入一个相同值的 `csrf_token` 字符串。以后给服务器发送请求的时候，必须在 `body` 中以及 `cookie` 中都携带 `csrf_token` ，服务器只有检测到 `cookie` 中的 `csrf_token` 和 `body` 中的 `csrf_token` 都相同，才认为这个请求是正常的，否则就是伪造的。那么黑客就没办法伪造请求了。在Flask中，如果想要防御 `CSRF` 攻击，应该做两步工作。第一个是使用 `flask_wtf.CSRFProtect` 来包裹 `app` 。第二个是在模版代码中添加一个 `input` 标签，加载 `csrf_token` 。示例代码如下：

- 服务器代码：

```
from flask_wtf import CSRFProtect
CSRFProtect(app)
```

- 模版代码：

```
<input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>
```

## iframe相关知识：

1. `iframe` 可以加载嵌入别的域名下的网页。也就是说可以发送跨域请求。比如我可以在我自己的网页中加载百度的网站，示例代码如下：

```
<iframe src="http://www.baidu.com/">
</ifrmae>
```

2. 因为 `iframe` 加载的是别的域名下的网页。根据同源策略， `js` 只能操作属于本域名下的代码，因此 `js` 不能操作通过 `iframe` 加载来的 `DOM` 元素。
3. 如果 `ifrmae` 的 `src` 属性为空，那么就没有同源策略的限制，这时候我们就可以操作 `iframe` 下面的代码了。并且，如果 `src` 为空，那么我们可以在 `iframe` 中，给任何域名都可以发送请求。
4. 直接在 `iframe` 中写 `html` 代码，浏览器是不会加载的。

# 七牛云配置

七牛云是一个对象存储的云服务平台。可以方便的存储一些图片和视频文件。对于一些中小型的公司，把多媒体文件存储在七牛云上是一个比较好的选择。因为技术不需要太负担文件存储的风险，以及如何做cdn加速，以及防盗链和图片的一些处理等。费用也不是很高。因此比较推荐使用。

## 创建空间：

登录七牛云后，创建空间。示例如下：


获取**access_key**和**secret_key**：



# 在网站中使用七牛存储文件：

在网站中使用七牛，可以直接把一些图片和视频文件直接发送给服务器。需要做以下配置：

1. 安装 `Python` 的 `sdk` ： `pip install qiniu` 。
2. 编写获取 `uptoken` 的接口：在后端代码中，写好一个接口，用来获取 `uptoken` 的。

```
@app.route('/uptoken/')
def uptoken():
    access_key = '你的AccessKey'
    secret_key = '你的SecretKey'
    q = qiniu.Auth(access_key,secret_key)
    bucket = 'hyvideo'
    token = q.upload_token(bucket)
    return jsonify({"uptoken":token})
```

1. 在前端添加 `JS` 的 `SDK` ：七牛为 `JavaScript` 提供了一个专门用来上传文件的接口。把以下文

件引入到 `html` 代码中：

```
<script src="https://cdn.staticfile.org/Plupload/2.1.1/moxie.js"></script>
<script src="https://cdn.staticfile.org/Plupload/2.1.1/plupload.dev.js"></script>
<script src="https://cdn.staticfile.org/qiniu-js-sdk/1.0.14-beta/qiniu.js"></script>
```

1. 在前端添加 `zlqiniu.js` 文件：这个文件是封装了七牛的初始化和配置相关的。使用这个文件可以写更少的代码来使用七牛。

```
<script src="{{ url_for('static',filename='zlqiniu.js') }}"></script>
```

2. 初始化七牛：使用以下代码初始化七牛，配置一些参数信息：

```
window.onload = function () {
    zlqiniu.setUp({
        'browse_btn': 'upload-btn',
        'uptoken_url': '/uptoken/',
        'success': function (up,file,info) {
            var url = file.name;
            console.log(url);
        }
    });
};
```

对以上代码做一些解释：

- browse_btn：这个是用来绑定按钮的id的。以后点击这个按钮就可以上传文件。
- uptoken_url：这个是后台写好的获取uptoken的接口。
- success：这个是文件上传成功后执行的回调。

# 项目部署：

这里用的是非常干净的 `ubuntu 16.04` 系统环境，没有使用任何云服务器，原因是因为不同的云服务器环境都不一样。我们就从零开始来完成部署。

# 在开发机上的准备工作：

1. 确认项目没有bug。
2. 用 `pip freeze > requirements.txt` 将当前环境的包导出到 `requirements.txt` 文件中，方便部署的时候安装。
3. 把 `dysms_python` 文件准备好。因为短信验证码的这个包必须通过
4. 将项目上传到服务器上的 `/srv` 目录下。这里以 `git` 为例。使用 `git` 比其他上传方式（比如使用pycharm）更加的安全，因为 `git` 有版本管理的功能，以后如果想要回退到之前的版本， `git` 轻而易举就可以做到。
5. 在 `https://git-scm.com/downloads` 下载 `Windows` 版本的客户端。然后双击一顿点击下一步安装即可。
6. 然后使用码云，在码云上创建一个项目。码云地址：https://gitee.com/
7. 然后进入到项目中，使用以下命令做代码提交：

```
# 初始化一个仓库
* git init
# 添加远程的仓库地址
* git remote add origin xxx.git
# 添加所有的代码到缓存区
* git add .
# 将代码提交到本地仓库
* git commit -m 'first commit'
# 从码云仓库上拉数据下来
* git pull origin master --allow-unrelated-histories
# 将本地仓库中的代码提交到远程服务器的master分支上
* git push origin master
```

---

# 在服务器上的准备工作：

1. ubuntu开启root用户：

```
> sudo passwd root
> 然后输入root用户的密码
```

2. 为了方便 `xshell` 或者 `CRT` 连接服务器，建议安装 `OpenSSH` （一般云服务器上都已经安装了）：

```
sudo apt install openssh-server openssh-client
service ssh restart
```

3. 安装 `vim` ：

```
sudo apt install vim
```

4. 修改一下 `ubuntu` 的 `apt` 源（云服务器一般都有自己的源，可以不用修改）， `apt` 源是用来安装软件的链接：
   先拷贝 `/etc/apt/sources.list` 为 `/etc/apt/sources.list.bak` ，然后用 `vi` 编辑 `/etc/apt/sources.list` ，删除 `sources.list` 中的其他内容，将下面代码粘贴到文件中。然后保存：

```
deb http://mirrors.ustc.edu.cn/ubuntu/ xenial main restricted universe multiverse
deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-security main restricted universe mul
tiverse
deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-updates main restricted universe mult
iverse
deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-proposed main restricted universe mul
tiverse
deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-backports main restricted universe mu
ltiverse
deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial main restricted universe multiver
se
deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-security main restricted universe
 multiverse
deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-updates main restricted universe
multiverse
deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-proposed main restricted universe
 multiverse
deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-backports main restricted univers
e multiverse
```

   然后更新源：

```
sudo apt update
```

5. 安装 `MySQL` 服务器和客户端：

```
sudo apt install mysql-server mysql-client
```

6. 安装 `memcached`：
   通过命令 `apt install memcached` 即可安装。更多的 `memcached` 的知识点请参考 `memcached` 那一章节。

7. 安装好项目要用到的 `Python`：

```
* sudo apt install python3
* sudo apt install python3-pip
* pip install --upgrade pip
```

   如果系统上已经有 `Python3` 了，就无需再安装了。因为 `supervisor` 不支持 `Python3`，所以还需要安装 `Python2`，如果没有，就安装一下：

```
* sudo apt install python2.7
* sudo apt install python-pip
```

   然后输入 `python2.7` 即可使用了。
   如果在输入 `pip` 的时候提示以下错误：

```
Traceback (most recent call last):
  File "/usr/bin/pip", line 9, in <module>
    from pip import main
ImportError: cannot import name main
```

   这是因为 `pip 10` 的一个 `bug`，可以零时使用以下解决方案：
   将 `/usr/bin/pip` 中的：

```
from pip import main
if __name__ == '__main__':
    sys.exit(main())
```

   改成：

```
from pip import __main__
if __name__ == '__main__':
    sys.exit(__main__._main())
```

8. 安装 `virtualenvwrapper`，并创建好项目要用到的虚拟环境：

```
* pip install virtualenvwrapper
```

   安装完 `virtualenvwrapper` 后，还需要配置 `virtualenvwrapper` 的环境变量。

- 首先通过 `which virtualenvwrapper.sh` 命令查看 `virtualenvwrapper.sh` 文件所在的路径。
- 在当前用户目录下创建 `.virtualenv` 文件夹，用来存放所有的虚拟环境目录。
- 在当前用户目录下编辑 `.bashrc` 文件，添加以下代码：

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

然后退出 `bashrc` 文件，输入命令 `source ~/.bashrc` 。

注意：因为我们是把 `virtualenvwrapper` 安装在了 `python2` 的环境中，所以在创建虚拟环境的时候需要使用 `--python` 参数指定使用哪个 `Python` 文件。比如我的 `python3` 的路径是在 `/usr/bin/python3` 。那么示例代码如下：

```
mkvirtualenv --python=/usr/bin/python3 xfz-env
```

9. 安装 `git` ：

```
sudo apt install git
```

10. 使用 `git` 下载项目代码：

```
* git init
* git remote add origin https://gitee.com/hynever/xfz_beike
* git pull origin master
```

11. 进入虚拟环境中，然后进入到项目所在的目录，执行命令： `pip install -r requirements.txt` ，安装项目依赖的包。如果提示 `OSError: mysql_config not found` ，那么再安装 `sudo apt install libmysqld-dev` 即可。
    注意短信验证码的包需要单独安装。把 `dysms_python` 文件夹上传到项目中，然后进入到这个文件夹中。执行命令： `python setup.py install` 。
12. 进入 `mysql` 数据库中，创建好项目的数据库。
13. 执行 `python manage.py migrate/upgrade` 将模型映射到数据库中。
14. 执行 `python zlbbs.py` ，然后在自己电脑上访问这个网站，确保没有BUG。
15. 设置 `DEBUG=False` ，避免如果你的网站产生错误，而将错误信息暴漏给用户。

---

## 安装**uwsgi**：

uwsgi是一个应用服务器，非静态文件的网络请求就必须通过他完成，他也可以充当静态文件服务器，但不是他的强项。uwsgi是使用python编写的，因此通过 `pip3 install uwsgi` 就可以了。(uwsgi必须安装在系统级别的Python环境中，不要安装到虚拟环境中)。然后创建一个叫做 `uwsgi.ini` 的配置文件：

```
[uwsgi]

# 必须全部为绝对路径
# 项目的路径
chdir           = /srv/zlbbs/
# Django的wsgi文件
wsgi-file       = /srv/zlbbs/zlbbs.py
# 回调的app对象
callable        = app
# Python虚拟环境的路径
home            = /root/.virtualenvs/xfz-env

# 进程相关的设置
# 主进程
master          = true
# 最大数量的工作进程
processes       = 10

http            = :8000

# 设置socket的权限
chmod-socket    = 666
# 退出的时候是否清理环境
vacuum          = true
```

然后通过命令 `uwsgi --ini uwsgi.ini` 运行，确保没有错误。然后在浏览器中访问 `http://ip地址:8000` ，如果能够访问到页面（可能没有静态文件）说明 `uwsgi` 配置没有问题。

## 安装和配置**nginx**：

虽然 `uwsgi` 可以正常的部署我们的项目了。但我们还是依然要采用 `nginx` 来作为**web**服务器。使用 `nginx` 来作为web服务器有以下好处：

1. uwsgi对静态文件资源处理并不好，包括响应速度，缓存等。
2. nginx作为专业的web服务器，暴露在公网上会比uwsgi更加安全一点。
3. 运维起来更加方便。比如要将某些IP写入黑名单，nginx可以非常方便的写进去。而uwsgi可能还要写一大段代码才能实现。

### 安装：

通过 `apt install nginx` 即可安装。

## nginx简单操作命令：

- 启动：service nginx start
- 关闭：service nginx stop
- 重启：service nginx restart
- 测试配置文件：service nginx configtest

## 添加配置文件：

在 `/etc/nginx/conf.d` 目录下，新建一个文件，叫做 `zlbbs.conf` ，然后将以下代码粘贴进去：

```
upstream zlbbs{
    server unix:///srv/zlbbs/zlbbs.sock;
}

# 配置服务器
server {
    # 监听的端口号
    listen      80;
    # 域名
    server_name 192.168.0.101;
    charset     utf-8;

    # 最大的文件上传尺寸
    client_max_body_size 75M;

    # 静态文件访问的url
    location /static {
        # 静态文件地址
        alias /srv/zlbbs/static;
    }

    # 最后，发送所有非静态文件请求到django服务器
    location / {
        uwsgi_pass  zlbbs;
        # uwsgi_params文件地址
        include     /etc/nginx/uwsgi_params;
    }
}
```

写完配置文件后，为了测试配置文件是否设置成功，运行命令： `service nginx configtest` ，如果不报错，说明成功。
每次修改完了配置文件，都要记得运行 `service nginx restart` 。

## 使用**supervisor**管理 `uwsgi` 进程：

让supervisor管理uwsgi，可以在uwsgi发生意外的情况下，会自动的重启。

## 安装**supervisor**：

因为 `supervisor` 是用 `python` 写成的，所以通过 `pip` 即可安装。
并且因为 `supervisor` 不支持 `python3` ，因此需要把 `supervisor` 安装在 `python2` 的环境中。
`pip2 install supervisor` 。

## 启动：

在项目的根目录下创建一个文件叫做 `supervisor.conf` ，然后将以下代码填入到配置文件中：

```
# supervisor的程序名字
[program:zlbbs]
# supervisor执行的命令
command=uwsgi --ini uwsgi.ini
# 项目的目录
directory = /srv/zlbbs
# 开始的时候等待多少秒
startsecs=0
# 停止的时候等待多少秒
stopwaitsecs=0
# 自动开始
autostart=true
# 程序挂了后自动重启
autorestart=true
# 输出的log文件
stdout_logfile=/var/log/supervisord.log
# 输出的错误文件
stderr_logfile=/var/log/supervisord.err

[supervisord]
# log的级别
loglevel=debug

[inet_http_server]
# supervisor的服务器
port = :9001
# 用户名和密码
username = admin
password = 123

# 使用supervisorctl的配置
```

```
[supervisorctl]
# 使用supervisorctl登录的地址和端口号
serverurl = http://127.0.0.1:9001

# 登录supervisorctl的用户名和密码
username = admin
password = 123

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
```

然后使用命令 `supervisord -c supervisor.conf` 运行就可以了。
以后如果想要启动 `uwsgi` ，就可以通过命令 `supervisorctl -c supervisor.conf` 进入到管理控制台，然后可以执行相关的命令进行管理：

- status # 查看状态
- start program_name #启动程序
- restart program_name #重新启动程序
- stop program_name # 关闭程序
- reload # 重新加载配置文件
- quit # 退出控制台

## 部署**celery**：

因为celery是需要用到redis，因此需要首先安装redis：

```
apt install redis-server
```

然后通过命令 `pip install celery` ，把 `celery` 安装在虚拟环境中。并
然后因为 `celery` 也是一个非守护进程。那么也可以通过 `supervisor` 来管理。配置文件如下：

```
# supervisor的程序名字
[program:zlbbs_celery]
uid = 1000
# supervisor执行的命令
command=/root/.virtualenvs/flask-env/bin/celery -A tasks.celery worker --loglevel=info
# 项目的目录
directory = /srv/zlbbs
# 开始的时候等待多少秒
startsecs=0
# 停止的时候等待多少秒
stopwaitsecs=0
# 自动开始
autostart=true
# 程序挂了后自动重启
```

```
autorestart=true
# 输出的log文件
stdout_logfile=/var/log/celery_supervisor.log
# 输出的错误文件
stderr_logfile=/var/log/celery_supervisor.err

[supervisord]
# log的级别
loglevel=debug

[inet_http_server]
# supervisor的服务器
port = :9002
# 用户名和密码
username = admin

# 使用supervisorctl的配置
[supervisorctl]
# 使用supervisorctl登录的地址和端口号
serverurl = http://127.0.0.1:9002

# 登录supervisorctl的用户名和密码
username = admin
password = 123

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
```