

# 一、文件上传

## 原生实现

- 模板文件

```
<form method="post" enctype="multipart/form-data">
  <input type="file" name="photo" /><br />
  <input type="submit" value="上传" />
</form>
```

- 视图函数

```
import os

# 配置上传文件保存目录
app.config['UPLOADED_FOLDER'] = os.path.join(os.getcwd(),
'static/upload')

@app.route('/upload/', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # 获取上传对象
        photo = request.files.get('photo')
        if photo:
            # 拼接保存路径名
            pathname =
os.path.join(app.config['UPLOADED_FOLDER'], photo.filename)
            # 保存上传文件
            photo.save(pathname)
            return '上传成功'
        else:
            return '上传失败'
    return render_template('upload.html')
```

- 上传限制设置

```
# 允许上传的文件后缀
ALLOWED_SUFFIX = set(['png', 'jpg', 'jpeg', 'gif'])
```

```

# 判断是否是允许的后缀
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1] in
ALLOWED_SUFFIX

# 限制请求大小
app.config['MAX_CONTENT_LENGTH'] = 1024 * 1024 * 8

# 展示上传的图片
@app.route('/uploaded/<filename>')
def uploaded(filename):
    return send_from_directory(app.config['UPLOADED_FOLDER'],
filename)

@app.route('/upload/', methods=['GET', 'POST'])
def upload():
    img_url = None

    if request.method == 'POST':
        # 获取上传对象
        photo = request.files.get('photo')
        if photo and allowed_file(photo.filename):
            # 拼接保存路径名
            pathname =
os.path.join(app.config['UPLOADED_FOLDER'], photo.filename)
            # 保存上传文件
            photo.save(pathname)
            # 构造上传文件的url
            img_url = url_for('uploaded',
filename=photo.filename)
        return render_template('upload.html', img_url=img_url)

```

## flask-uploads

- 说明：极大的的简化了文件上传相关的操作，使用非常方面。
- 安装： `pip install flask-uploads`
- 使用：
  - 配置

```

from flask_uploads import UploadSet, IMAGES

```

```

from flask_uploads import configure_uploads
from flask_uploads import patch_request_class
import os

app.config['UPLOADED_PHOTOS_DEST'] = os.getcwd()
app.config['MAX_CONTENT_LENGTH'] = 8 * 1024 * 1024
# 创建上传对象
photos = UploadSet('photos', IMAGES)
# 配置上传对象
configure_uploads(app, photos)
# 配置上传文件大小，默认为64M，
# 若设置为None，则以MAX_CONTENT_LENGTH配置为准
patch_request_class(app, size=None)

```

#### ○ 视图函数

```

@app.route('/upload/', methods=['GET', 'POST'])
def upload():
    img_url = None
    if request.method == 'POST':
        # 获取上传对象
        photo = request.files.get('photo')
        if photo:
            # 保存上传文件，返回文件名
            filename = photos.save(photo)
            # 根据文件名获取上传文件的URL
            img_url = photos.url(filename)
        return render_template('upload.html', img_url=img_url)

```

## 二、发送邮件

flask-mail

- 说明：专门用于邮件发送的扩展库，使用非常方便。
- 安装： `pip install flask-mail`
- 使用：

```

from flask_mail import Mail, Message
import os

# 邮件发送配置，——一定要放在创建Mail对象之前

```

```

# 邮件服务器配置
app.config['MAIL_SERVER'] = "smtp.126.com" # 用用户帐号

mail = Mail(app)
@app.route('/send/')
def send():
    # 创建邮件消息对象
    msg = Message('账户激活',
                  recipients=['shuai_fmzj@163.com'],
                  sender=app.config['MAIL_USERNAME'])
    msg.html = '恭喜你，中奖了!!!'
    # 发送邮件 mail.send(msg) return '邮件已发送'

```

- 封装函数发送邮件

```

def send_mail(subject, to, template, *args, **kwargs):
    if isinstance(to, list):
        recipients = to
    elif isinstance(to, str):
        recipients = to.split(',')
    else:
        raise Exception('邮件接收者参数类型有误')
    # 创建邮件消息对象
    msg = Message(subject,
                  recipients=recipients,
                  sender=app.config['MAIL_USERNAME'])
    # 将邮件模板渲染后作为邮件内容
    msg.html = render_template(template, *args, **kwargs)
    # 发送邮件
    mail.send(msg)

```

- 异步发送邮件

```

from flask import current_app

# 异步发送邮件任务
def async_send_mail(app, msg):
    # 邮件发送必须在程序上下文
    # 新的线程中没有上下文，因此需要手动创建
    with app.app_context():
        mail.send(msg)

```

```

# 封装函数发送邮件
def send_mail(subject, to, template, *args, **kwargs):
    if isinstance(to, list):
        recipients = to
    elif isinstance(to, str):
        recipients = to.split(',')
    else:
        raise Exception('邮件接收者参数类型有误')
    # 创建邮件消息对象
    msg = Message(subject,
                   recipients=recipients,
                   sender=app.config['MAIL_USERNAME'])
    # 将邮件模板渲染后作为邮件内容
    msg.html = render_template(template, *args, **kwargs)
    # 异步发送邮件
    # current_app是app的代理对象
    # 根据代理对象current_app找到原始的app
    app = current_app._get_current_object()
    # 创建线程
    thr = Thread(target=async_send_mail, args=(app, msg))
    # 启动线程
    thr.start()
    # 返回线程
    return thr

```

- QQ邮件发送额外配置：需要配置QQ邮箱开启smtp服务，然后设置授权码

```

# 邮箱端口
app.config['MAIL_PORT'] = 465
# 使用SSL(加密传输)
app.config['MAIL_USE_SSL'] = True
# 不是QQ邮箱的密码，而是授权码
app.config['MAIL_PASSWORD'] = '授权码'

```

## 三、验证码

### 6.1 安装Pillow库

PIL:Python Imaging Library, 已经是Python平台事实上的图像处理标准库了。PIL功能非非常强大, 但API却非非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫Pillow，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow。

```
$ pip install pillow
```

## 6.2 创建验证码步骤

- 1)、创建画布
- 2)、生成验证码字符串
- 3)、画验证码
- 4)、画干扰点
- 5)、画干扰线
- 6)、返回验证码图片

## 6.3 常用方法

方法名	说明
Image.new()	创建图像
ImageDraw.Draw	创建画笔
ImageDraw.point	画点
ImageDraw.line	画线
ImageDraw.text	画文字
ImageFont.truetype	获取字体

## 6.4 实现

```
from io import BytesIO
from random import randint, sample
import string
```

```

from PIL import Image,ImageFont,ImageDraw
class VerifyCode:
def code(self): #获取验证码字符串串的方法
    return self.__code
    def output(self):
        # 1 image pen
        im = Image.new('RGB', (self.width,
self.height),self.__rand_color(160,255))
        self.pen = ImageDraw.Draw(im)
        # 2code string
        self.__code = self.rand_string()
# 3 draw string

self.__draw_string()
    # 4 point
    self.__disturb_point()
    # 5 line
    self.__draw_line()
    for i in range(300):
        x = randint(1,self.width - 1)
        y = randint(1,self.height - 1)
        self.pen.point([(x,y)],fill=self.__rand_color(60,120))
    def __draw_string(self):
        font1 = ImageFont.truetype('SIMLI.TTF',size=20,encoding='utf-
8')
width = (self.width - 20) / self.size
for i in range(len(self.__code)):
    x = 13 + i * width
    y = randint(5,20)
    self.pen.text((x,
y),self.__code[i],fill='black',font=font1)
def rand_string(self): # 数字验证码
    return str(randint(1000,pow(10,self.size)) - 1)
    def __rand_color(self,low,high):
        return randint(low,high),randint(low,high),randint(low,high)

```

在flask中使用用:

```

from VerifyCode import VerifyCode
def yzm():
    vc = VerifyCode()
    result = vc.output()
    session['code'] = image.code
    return HttpResponse(result,'image/png')

```

## 四、分页页显示

方法:paginate, 分页页查询

参数:

page:当前的页码

per\_page:每页的条数

error\_out:当查询出错时是否报错

返回值: Pagination:分页页对象, 包含了了所有的分页页信息

Pagination:

属性:

page:当前页页码

per\_page:每页页的条数, 默认为20条

pages:总页页数

total:总条数

prev\_num:上一页的页码 next\_num:下一页的页码

has\_prev:是否有上一页

has\_next:是否有下一页

items:当前页页的数据

方法:

iter\_pages:返回一个迭代器, 在分页导航条上显示的页码列表, 显示不完  
的时返回None

prev:上一页的分页对象

next:下一页的分页对象

### ● 封装分页显示的宏

```

{% macro show_pagination(pagination, endpoint) %}
    <nav aria-label="Page navigation">
    <ul class="pagination"> {# 上——页页 #}
        <li {% if not pagination.has_prev %}class="disabled"
    {% endif %}>

```



```

        <a href="{% if pagination.has_prev %}{%
url_for(endpoint, page=pagination.prev_num, **kwargs) %}{% else
%}#{% endif %}" aria-label="Previous">
            <span aria-hidden="true">&laquo;</span>
</a> </li>
{# 中间页码 #}
{% for p in pagination.iter_pages() %}
    {% if p %}
        <li {% if pagination.page == p
%}class="active"{% endif %}><a href="{% url_for(endpoint, page=p,
**kwargs) %}">{{ p }}</a></li>
        {% else %}
            <li><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
{# 下一页 #}
    <li {% if not pagination.has_next %}class="disabled"
{% endif %}>
        <a href="{% if pagination.has_next %}{%
url_for(endpoint, page=pagination.next_num, **kwargs) %}{% else
%}#{% endif %}" aria-label="Next">
            <span aria-hidden="true">&raquo;</span>
</a> </li>
</ul> </nav>
{% endmacro %}

```