* 跨站请求伪造保护

默认Flask-WTF能保护所有表单免受跨站请求伪造(Cross-Site Request Forgery，CSRF)的攻击。恶意网站把请求发送到被攻击者已登录的其他网站时就会引发 CSRF攻击。为了实现CSRF保护，Flask-WTF需要程序设置一个密钥。Flask-WTF使用这个密钥生成加密令牌，再用令牌验证请求中表单数据的真伪。

设置Flask-WTF密钥：

app = Flask(\_\_name\_\_)  
app.config[**'SECRET\_KEY'**] = **'hoshizora rin'**

app.config字典可用来存储框架、扩展和程序本身的配置变量。

还提供了一些方法，可以从文件或环境中导入配置值。

**SECRET\_KEY**配置变量是通用密钥，可在Flask和多个第三方扩展中使用。加密的强度取决于变量值的机密程度。不同的程序要使用不同的密钥，而且要保证其他人不知道所用的字符串。

注意：为了增强安全性，密钥不应该直接写入代码，而要保存在环境变量中。

* 表单类

每个表单都由一个继承自FlaskForm的类表示。这个类定义表单中的一组字段，每个字段都用对象表示。字段对象可附属一个或多个验证函数。验证函数用来验证用户提交的输入值是否符合要求。

示例：包含一个文本字段和一个提交按钮的简单表单

**from** flask\_wtf **import** FlaskForm  
**from** wtforms **import** StringField, SubmitField  
**from** wtforms.validators **import** DataRequired  
**class** NameForm(FlaskForm):  
 name = StringField(**'你的名字是？'**, validators=[DataRequired()])  
 submit = SubmitField(**'提交'**)

NameForm表单有一个name文本字段和一个submit提交按钮

StringField类表示属性为type="text"的<input>元素。SubmitField类表示属性为 type="submit"的<input>元素。字段构造函数的第一个参数是把表单渲染成 HTML时使用的标号。

可选参数validators指定一个由验证函数组成的列表，在接受用户提交的数据之前验证数据。验证函数DataRequired()确保提交的字段不为空。

WTForms支持的HTML标准字段和验证函数很多，自己查(狗书P35)。

* 表单渲染成HTML

Flask-Bootstrap提供一个非常高端的辅助函数，可以使用Bootstrap中预先定义好的表单样式渲染整个Flask-WTF表单。

{% **import "bootstrap/wtf.html" as wtf** %}  
{{ **wtf**.**quick\_form**(**form**) }}

wtf.quick\_form()函数的参数为Flask-WTF表单对象，使用Bootstrap默认样式渲染传入的表单。

示例：使用Flask-WTF和Flask-Bootstrap渲染表单

templates/index.html：

{% **extends "base.html"** %}  
{% **import "bootstrap/wtf.html" as wtf** %}{% **block page\_content** %}  
 <**div class="page-header"**>  
 <**h1**>Hello, {% **if name** %}{{ **name** }}{% **else** %}Stranger{% **endif** %}!</**h1**></**div**>  
 {{ **wtf**.**quick\_form**(**form**) }}  
{% **endblock** %}

* 视图函数处理表单

修改成表单的index()视图

@app.route(**'/'**, methods=[**'GET'**, **'POST'**]) *# 支持GET和POST***def** index():  
 name = **None** form = NameForm()  
 **if** form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受* name = form.name.data *# 获取字段data属性* form.name.data = **''** *# 清空字段* **return** render\_template(**'index.html'**, form=form, name=name)

第1次访问，服务器收到没有表单数据的GET请求，validate\_on\_submit()返回 False，name为None；模板执行else语句

不输入直接提交，提示：



**20180303**

* 重定向和用户会话

上面表单，提交之后刷新会出现警告，要求再次提交表单之前进行确认。



因为刷新页面，浏览器会重新发送之前已经发送过的最后一个请求。如果是包含表单数据的POST请求，刷新页面会再次提交表单。

最好不要让POST请求成为浏览器发送的最后一个请求。

可以使用重定向作为POST请求的响应。重定向是特殊的响应，响应内容是URL，而不是HTML代码的字符串，浏览器收到重定向响应后，向重定向的URL发送GET请求。称为Post/重定向/Get模式。

但是处理POST请求时，使用form.name.data获取用户输入，但请求结束，数据随之丢失。所以需要程序能保存输入数据。

程序可以把数据存储在用户会话中。用户会话是私有存储，存在每个连接到服务器的客户端中。用户会话是请求上下文中的变量session，类似于字典。

注：默认用户会话保存在客户端cookie中，使用设置的SECRET\_KEY进行加密签名。如果篡改了cookie的内容，签名就会失效，会话也随之失效。

* 示例：修改index()，实现重定向和用户会话：

**from** flask **import** session, redirect, url\_for  
@app.route(**'/'**, methods=[**'GET'**, **'POST'**]) *# 支持GET和POST***def** index():form = NameForm()  
 **if** form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受* session[**'name'**] = form.name.data *# 获取字段data属性存入session* **return** redirect(url\_for(**'index'**)) *# 重定向,GET方式请求index*  
 **return** render\_template(**'index.html'**, form=form, name=session.get(**'name'**))

第1次GET请求，没有session没有name字段，输出stranger；POST提交表单后，name被存入session中，随即重定向再次GET请求，输出name的数据；下次再刷新，仍然是同一个name。浏览器的cookies保存有session字段。

session在不同请求之间共享数据。

* Flash消息

flash()函数实现请求完成后告诉用户状态发生了变化，比如用户提交错误登录表单后，服务器发回响应重新渲染表单，显示消息提示用户名或密码错误。

* 示例：

①hello.py添加Flash消息：

**from** flask **import** session, redirect, url\_for, flash  
@app.route(**'/'**, methods=[**'GET'**, **'POST'**]) *# 支持GET和POST***def** index():  
 form = NameForm()  
 **if** form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受* old\_name = session.get(**'name'**)  
 **if** old\_name **and** old\_name != form.name.data:  
 flash(**'你似乎改名字了！'**)  
 session[**'name'**] = form.name.data *# 获取字段data属性存入session* **return** redirect(url\_for(**'index'**))  
 **return** render\_template(**'index.html'**, form=form, name=session.get(**'name'**))

每次提交名字都会和之前存储在session的名字比较，如果不一样就调用flash()函数，发给客户端下一个响应中显示消息。

需要在模板渲染才能显示消息。最好在父模板中渲染flash消息。

②templates/base.html渲染Flash消息：

修改content坑位：

{% **block content** %}  
 <**div class="container"**>  
 {% **for message in get\_flashed\_messages**() %}  
 <**div class="alert alert-warning"**>  
 <**button type="button" class="close" data-dismiss="alert"**>**&times;**</**button**>  
 {{ **message** }}</**div**>{% **endfor** %}  
 {% **block page\_content** %}{% **endblock** %}</**div**>{% **endblock** %}

Flask把get\_flashed\_messages()函数开放给模板，用来获取并渲染消息。

在模板中使用循环是因为在之前的请求循环中每次调用flash()函数都会生成一个消息，可能有多个消息在排队等待显示。

结果：



* 数据库

数据库按照一定规则保存程序数据， 程序再发起查询取回所需的数据。Web程序最常用基于关系模型的数据库，也称为SQL数据库，因为它们使用结构化查询语言。不过最近几年文档数据库和键值对数据库成了流行的替代选择，这两种合称NoSQL数据库。

关系型数据库存储数据高效，但使用复杂；

NoSQL不用联结，数据重复量增加，但查询速度快，操作简单，无须联结。

关系型数据库擅于用高效且紧凑的形式存储结构化数据，需要花费大量精力保证数据的一致性。NoSQL数据库放宽了要求，性能上有优势。

对中小型程序，SQL和NoSQL都是很好的选择，性能相当。

* Python数据库框架

选择数据库框架时要考虑的因素：

①易用性

如果直接比较数据库引擎和数据库抽象层，显然后者取胜。

抽象层，也称为对象关系映射(Object-Relational Mapper，ORM)或对象文档映射

(Object-Document Mapper，ODM)，在用户不知觉的情况下把高层的面向对象操

作转换成低层的数据库指令。

②性能

ORM和ODM把对象业务转换成数据库业务会有一定的损耗。

大多数情况下，ORM和ODM对生产率的提升远远超过性能的损耗。

关键在于如何选择一个能直接操作低层数据库的抽象层，以防特定的操作需要直接使用数据库原生指令优化。

③可移植性

④Flask集成度

不一定非得选择已经集成了Flask的框架，但选择这些框架可以节省编写集成代码的时间，简化配置和操作。

狗书推荐使用Flask-SQLAlchemy(http://pythonhosted.org/Flask-SQLAlchemy/)，其集成了SQLAlchemy(http://www.sqlalchemy.org/)框架。

* Flask-SQLAlchemy管理数据库

Flask-SQLAlchemy简化了在Flask程序中使用SQLAlchemy的操作。

SQLAlchemy是一个强大的关系型数据库框架，支持多种数据库后台。

其提供了高层ORM和原生SQL的低层功能。

在Flask-SQLAlchemy中，数据库使用URL指定。

* 常用FLask-SQLAlchemy数据库URL

|  |  |
| --- | --- |
| 数据库引擎 | URL |
| MySQL | *mysql://username:password@hostname/database* |
| Postgres | *postgresql://username:password@hostname/database* |
| SQLite(Unix) | *sqlite:////absolute/path/to/database* |
| SQLite(Windows) | *sqlite:///c:/absolute/path/to/database* |

SQLite数据库不需要使用服务器，因此不用指定hostname、username和password。 URL中的database是硬盘上文件的文件名。

程序使用的数据库URL必须保存到Flask配置对象的 SQLALCHEMY\_DATABASE\_URI键中。配置对象中还有一个很有用的选项， 即SQLALCHEMY\_COMMIT\_ON\_TEARDOWN键，设为True时，每次请求结束后都会自动提交数据库中的变动。

* 示例：配置简单的SQLite数据库

*# 配置sqlite数据库***from** flask\_sqlalchemy **import** SQLAlchemy  
**import** os  
base\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))  
app = Flask(\_\_name\_\_)  
app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **'sqlite:///{}'**.format(os.path.join(base\_dir, **'data.sqlite'**))*# 配置sqlite的URL,数据库名字data.sqlite*  
app.config[**'SQLALCHEMY\_COMMIT\_ON\_TEARDOWN'**] = **True** *# 有变化自动提交*db = SQLAlchemy(app)

db对象是SQLAlchemy类的实例，表示程序使用的数据库，同时还获得了Flask-SQLAlchemy提供的所有功能。

* 定义模型

模型表示程序使用的持久化实体。在ORM中，模型一般是一个类，类的属性对应数据库表中的列。

Flask-SQLAlchemy创建的数据库实例为模型提供了一个父类以及一系列辅助类和辅助函数，可用于定义模型的结构。

* 示例：定义Role和User模型

**class** Role(db.Model):  
 \_\_tablename\_\_ = **'roles'** *#表名*id = db.Column(db.Integer, primary\_key=**True**) *# 主键* name = db.Column(db.String(64), unique=**True**)  
 **def** \_\_repr\_\_(self):  
 **return '<Role %r>'** % self.name  
  
**class** User(db.Model):  
 \_\_tablename\_\_ = **'users'** id = db.Column(db.Integer, primary\_key=**True**) *# 主键* username = db.Column(db.String(64), unique=**True**, index=**True**)  
 **def** \_\_repr\_\_(self):  
 **return '<User %r>'** % self.username

自定义模型继承于db.Model类，一个模型对应一张表，模型的属性对应表中的列，是db.Column类的实例。

\_\_repr()\_\_返回一个具有可读性的字符串表示模型，可在调试和测试时使用。

常见SQLAlchemy列(字段)类型有String、Integer、Text、Date等，见狗书P48

常见SQLAlchemy列选项(约束)有：

|  |  |
| --- | --- |
| 选项名 | 说明 |
| primary\_key | 设为True表示主键 |
| unique | 设为True不允许出现重复的值 |
| index | 设为True为这列创建索引，提升查询效率 |
| nullable | 设为True，允许使用空值；设为False，不允许使用空值 |
| default | 为这列设置默认值 |

每个模型必须要有主键，一般是id。

* 关系

关系型数据库使用关系把不同表中的行联系起来。roles表示用户可用角色，有主键id和name字段；users表示用户信息，有主键id、username、password字段，还有一个外键role\_id表示用户选择的什么角色。显然一个角色对应n个用户，一个用户只能有一个角色，是一对多的关系

* 示例：一对多关系：

**class** Role(db.Model):  
 \_\_tablename\_\_ = **'roles'** id = db.Column(db.Integer, primary\_key=**True**) *# 主键* name = db.Column(db.String(64), unique=**True**)  
 users = db.relationship(**'User'**, backref=**'role'**)  
**class** User(db.Model):  
 \_\_tablename\_\_ = **'users'** id = db.Column(db.Integer, primary\_key=**True**) *# 主键* username = db.Column(db.String(64), unique=**True**, index=**True**)  
 role\_id = db.Column(db.Integer, db.ForeignKey(**'roles.id'**)) *# 外键,roles表中的id*

添加到Role模型的users属性代表这个关系的面向对象视角。对于Role类的一个实例，其users属性将返回与角色相关联的用户组成的列表。

db.relationship()的第一个参数表示关系的另一端是哪个模型。如果模型类尚未定义，可使用字符串形式指定。backref参数向User模型中添加一个role属性，从而定义反向关系。这一属性可替代role\_id访问Role模型，获取的是模型对象，而不是外键的值。

* 常用SQLAlchemy关系选项

|  |  |
| --- | --- |
| **选项名** | **说明** |
| backref | 在关系的另一个模型中添加反向引用 |
| primaryjoin | 明确指定两个模型之间使用的联结条件。只在模棱两可的关系中需要指定 |
| lazy | 指定如何加载相关记录。可选值有select(首次访问时按需加载)、 immediate(源对象加载后就加载)、joined(加载记录，但使用联结)、 subquery(立即加载，但使用子查询)、noload(永不加载)和dynamic(不加载记录，但提供加载记录的查询) |
| uselist | 如果设为 False，不使用列表，而使用标量值 |
| order\_by | 指定关系中记录的排序方式 |
| secondary | 指定多对多关系中关系表的名字 |
| secondaryjoin | SQLAlchemy无法自行决定时，指定多对多关系中的二级联结条件 |

一对一关系可以用一对多关系表示，但调用db.relationship()时要把uselist 设为False，把多变成一。

多对多关系很复杂，需要用到第三张表，称为关系表。

**20180304**

* ずら丸、誕生日おめでとう！



* 数据库操作

学习使用模型最好方法是在Python shell中实际操作。

要使用shell要Flask-Script支持命令行：

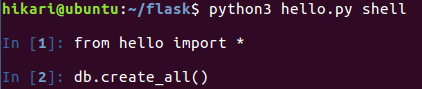
**from** flask\_script **import** Manager  
manager = Manager(app)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:manager.run()

提示SQLALCHEMY\_TRACK\_MODIFICATIONS要设为True或False…

app.config[**'SQLALCHEMY\_TRACK\_MODIFICATIONS'**] = **False**

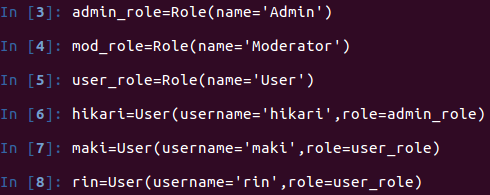
①创建表：create\_all()



程序目录多了一个data.sqlite的文件。

如果数据库表已经存在于数据库，db.create\_all()不会重新创建或者更新这个表

②插入行



这些对象只存在于Python中，还未写入数据库。



id没有被赋值，说明数据的确未写入数据库。

通过数据库会话(也称事务)db.session管理对数据库所做的改动

>>> db.session.add(admin\_role)

>>> db.session.add(mod\_role)

>>> db.session.add(user\_role)

>>> db.session.add(hikari)

>>> db.session.add(maki)

>>> db.session.add(rin)

可以简写为：



调用commit()方法提交会话，将对象写入数据库：





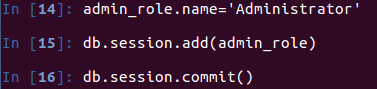
说明id已经被赋值了，数据存入了数据库。

数据库会话能保证数据库的一致性。提交操作使用原子方式把会话中的对象全部写入数据库。如果在写入过程中发生了错误，整个会话都会失效。

db.session.rollback()：数据库会话回滚

③修改行

数据库会话使用add()方法也可以更新模型



将admin\_role的name属性从之前的'Admin'改为'Administrator'

④删除行

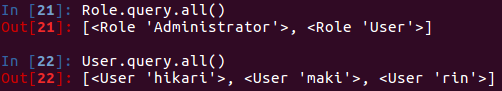
数据库会话使用delete()方法删除模型



将"Moderator"角色(列，字段)从数据库删除

⑤查询行

每个模型类都有query对象，查询字段所有行使用all()方法



使用过滤器可以对query对象进行更精确的数据库查询

查询User中所有角色是user\_role的用户



将query对象转换成字符串可以查看SQLAlchemy生成的原生SQL查询语句：



* 常用的SQLAlchemy查询过滤器

|  |  |
| --- | --- |
| **过滤器** | **说明** |
| filter() | 把过滤器添加到原查询上，返回一个新查询 |
| filter\_by() | 把等值过滤器添加到原查询上，返回一个新查询 |
| limit() | 使用指定的值限制原查询返回的结果数量，返回一个新查询 |
| offset() | 偏移原查询返回的结果，返回一个新查询 |
| order\_by() | 根据指定条件对原查询结果进行排序，返回一个新查询 |
| group\_by() | 根据指定条件对原查询结果进行分组，返回一个新查询 |

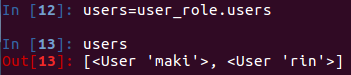
在查询上应用指定的过滤器后，通过调用all()执行查询，以列表的形式返回结果。还有其他方法能触发查询执行。

* 常使用的SQLAlchemy查询执行函数

|  |  |
| --- | --- |
| **方法** | **说明** |
| all() | 以列表形式返回查询的所有结果 |
| first() | 返回查询的第一个结果；没有返回None |
| first\_or\_404() | 返回查询的第一个结果；没有结果则终止请求，返回404错误响应 |
| get() | 返回指定主键对应的行，没有则返回None |
| get\_or\_404() | 返回指定主键对应的行，没有则终止请求，返回404错误响应 |
| count() | 返回查询结果的数量 |
| paginate() | 返回一个Paginate对象，包含指定范围内的结果 |

关系和查询的处理方式类似。

示例：分别从关系的两端查询角色和用户之间的一对多关系：



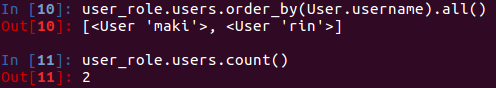


但是此时执行user\_role.users表达式，隐含的查询会调用all()返回一个用户列表。 query对象是隐藏的，无法指定过滤器。

示例：修改Role模型的动态关系

**class** Role(db.Model):  
 \_\_tablename\_\_ = **'roles'** id = db.Column(db.Integer, primary\_key=**True**)name = db.Column(db.String(64), unique=**True**)  
 users = db.relationship(**'User'**, backref=**'role'**, lazy=**'dynamic'**)

加入了lazy = 'dynamic'参数，从而禁止自动执行查询；user\_role.users会返回一个尚未执行的查询，可以添加过滤器：



* 示例：在视图函数中操作数据库

①index()视图函数中操作数据库：

@app.route(**'/'**, methods=[**'GET'**, **'POST'**]) *# 支持GET和POST***def** index():  
 form = NameForm()  
 **if** form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受* user = User.query.filter\_by(username=form.name.data).first() *# 数据库中查询* **if** user **is None**: *# 用户不存在于数据库,加入* user = User(username=form.name.data)  
 db.session.add(user)  
 session[**'known'**] = **False** *# known记录是不是新用户* **else**:  
 session[**'known'**] = **True** session[**'name'**] = form.name.data *# 获取字段data属性存入session* form.name.data = **''** *# 清空字段* **return** redirect(url\_for(**'index'**))  
 **return** render\_template(**'index.html'**, form=form, name=session.get(**'name'**), known=session.get(**'known'**, **False**))

因为上面设置了自动提交，不需要db.session.commit()

②templates/index.html：

{% **extends "base.html"** %}  
{% **import "bootstrap/wtf.html" as wtf** %}  
{% **block styles** %}  
 {{ **super**() }}  
 <**style**>**p** {**font**: 24**px**/24**px "Microsoft YaHei"**; **color**: **red**;}</**style**>  
{% **endblock** %}  
{% **block page\_content** %}  
 <**div class="page-header"**>  
 <**h1**>Hello, {% **if name** %}{{ **name** }}{% **else** %}Stranger{% **endif** %}!</**h1**>  
 {% **if not known** %}  
 <**p**>很高兴遇见你！</**p**> *{# 新用户 #}*  
 {% **else** %}  
 <**p**>非常高兴再次遇见你！</**p**> *{# 数据库中已知用户 #}*  
 {% **endif** %}</**div**>  
 {{ **wtf**.**quick\_form**(**form**) }}  
{% **endblock** %}

* 集成Python shell

shell练习用尚可，但每次都要重复导入模型和实例，枯燥、不友善。可以做些配置，让Flask-Script的shell命令自动导入特定的对象。

想把对象添加到导入列表中， 要为shell命令注册一个make\_context回调函数

**from** flask\_script **import** Shell  
**def** make\_shell\_context():  
 **return** dict(app=app, db=db, User=User, Role=Role)  
manager.add\_command(**"shell"**, Shell(make\_context=make\_shell\_context))

E:\练习\flask>hello.py shell

>>> app

<Flask 'hello'>

>>> db

<SQLAlchemy engine=sqlite:///E:\练习\flask\data.sqlite>

>>> User

<class '\_\_main\_\_.User'>

* Flask-Migrate实现数据库迁移

有时需要修改数据库模型，修改之后还需要更新数据库。

仅当数据库表不存在时，Flask-SQLAlchemy才会创建表。因此，更新表的唯一方式就是先删除旧表，不过会丢失数据库中的所有数据。

更好方法是使用数据库迁移框架。数据库迁移框架能跟踪数据库模式的变化，然后增量式的把变化应用到数据库中。

SQLAlchemy的主力开发人员编写了一个迁移框架Alembic；Flask-Migrate扩展对Alembic做了轻量级包装，并集成到Flask-Script中。

①创建迁移仓库

**from** flask\_migrate **import** Migrate, MigrateCommand  
migrate = Migrate(app, db)  
manager.add\_command(**'db'**, MigrateCommand)

Flask-Migrate提供了一个MigrateCommand类，可附加到FlaskScript的manager 对象上。此处，MigrateCommand类使用db命令附加。

在维护数据库迁移之前，要使用init命令创建迁移仓库：

python hello.py db init

然后生成migrations文件夹，所有迁移脚本都存放其中。

②创建迁移脚本

数据库迁移用迁移脚本表示。脚本有两个函数：upgrade()把迁移中的改动应用到数据库中；downgrade()将改动删除。

Alembic可以添加和删除改动，数据库可重设到历史的任意一点。

可以使用revision命令手动创建迁移，也可使用migrate命令自动创建。

手动创建的迁移只是一个骨架，upgrade()和downgrade()函数都是空的，要使用Alembic提供的Operations对象指令实现具体操作。自动创建的迁移会根据模型定义和数据库当前状态之间的差异生成upgrade()和downgrade()函数的内容

**注意**：自动创建的迁移不一定正确，有可能会漏掉一些细节。自动生成迁移脚本后一定要进行检查。

migrate命令自动创建迁移脚本：

python hello.py db migrate -m "initial migration"

③更新数据库

使用upgrade命令把迁移应用到数据库：

python hello.py db upgrade

对第一个迁移，其作用和调用db.create\_all()方法一样。但后续迁移中，upgrade 命令能把改动应用到数据库中，且不影响其中保存的数据。

* 电子邮件

Python标准库的smtplib包可用在Flask程序中发送电子邮件；

包装了smtplib的Flask-Mail扩展能更好地和Flask集成。

Flask-Mail连接到SMTP(Simple Mail Transfer Protocol，简单邮件传输协议)服务器，并把邮件交给这个服务器发送。如果不进行配置，Flask-Mail会连接localhost 上的端口25，无需验证即可发送电子邮件。

* Flask-Mail SMTP服务器的配置

|  |  |  |
| --- | --- | --- |
| **配置** | **默认值** | **说明** |
| MAIL\_SERVER | localhost | 电子邮件服务器的主机名或IP地址 |
| MAIL\_PORT | 25 | 电子邮件服务器的端口 |
| MAIL\_USE\_TLS | False | 启用TLS(Transport Layer Security，传输层安全)协议 |
| MAIL\_USE\_SSL | False | 启用SSL(Secure Sockets Layer，安全套接层)协议 |
| MAIL\_USERNAME | None | 邮件账户的用户名 |
| MAIL\_PASSWORD | None | 邮件账户的密码 |

配置Flask-Mail使用QQ邮箱：

**from** flask\_mail **import** Mail  
app.config[**'MAIL\_SERVER'**] = **'smtp.qq.com'**app.config[**'MAIL\_PORT'**] = 465  
app.config[**'MAIL\_USE\_SSL'**] = **True**app.config[**'MAIL\_USE\_TLS'**] = **False**app.config[**'MAIL\_USERNAME'**] = os.environ.get(**'MAIL\_USERNAME'**) *# xxx@qq.com*app.config[**'MAIL\_PASSWORD'**] = os.environ.get(**'MAIL\_PASSWORD'**) *# 授权码*mail = Mail(app)

邮箱设置-->账户-->开启IMAP/SMTP服务-->手机验证-->授权码

**注意**：千万不要把账户密码直接写入脚本；为了保护账户信息，让脚本从环境变量中导入敏感信息。

设置环境变量：

Linux：export MAIL\_USERNAME=< username >

Windows：set MAIL\_USERNAME=< username >

* 在程序中集成发送电子邮件功能

**from** flask\_mail **import** Message  
app.config[**'FLASKY\_MAIL\_SUBJECT\_PREFIX'**] = **'[Flasky]'**app.config[**'FLASKY\_MAIL\_SENDER'**] = **'Flasky Admin <flasky@example.com>'  
def** send\_email(to, subject, template, \*\*kwargs):  
 *# to为接收方,subject为邮件主题,template为渲染邮件正文的模板* msg = Message(app.config[**'FLASKY\_MAIL\_SUBJECT\_PREFIX'**] + **' '** + subject,  
 sender=app.config[**'FLASKY\_MAIL\_SENDER'**], recipients=[to])  
 msg.body = render\_template(template + **'.txt'**, \*\*kwargs)  
 msg.html = render\_template(template + **'.html'**, \*\*kwargs)  
 mail.send(msg)

index()视图修改为：

app.config[**'FLASKY\_ADMIN'**] = os.environ.get(**'FLASKY\_ADMIN'**)  
@app.route(**'/'**, methods=[**'GET'**, **'POST'**]) *# 支持GET和POST***def** index():  
 form = NameForm()  
 **if** form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受* user = User.query.filter\_by(username=form.name.data).first() *# 数据库中查询* **if** user **is None**: *# 用户不存在于数据库,加入* user = User(username=form.name.data)  
 db.session.add(user)session[**'known'**] = **False** *# known记录是不是新用户* **if** app.config[**'FLASKY\_ADMIN'**]:  
 send\_email(app.config[**'FLASKY\_ADMIN'**], **'New User'**, **'mail/new\_user'**, user=user)  
 **else**:  
 session[**'known'**] = **True** session[**'name'**] = form.name.data *# 获取字段data属性存入session* form.name.data = **''** *# 清空字段* **return** redirect(url\_for(**'index'**))  
 **return** render\_template(**'index.html'**, form=form, name=session.get(**'name'**), known=session.get(**'known'**, **False**))

电子邮件的收件人保存在环境变量FLASKY\_ADMIN中，程序启动加载到同名配置变量中。要创建两个模板文件(templates文件夹下的mail文件夹下)，分别用于渲染纯文本和HTML版本的邮件正文。

电子邮件的模板中要有一个模板参数是user，因此调用send\_mail()函数时要以

关键字参数的形式传入user。

注意：上面代码未亲自测试，不一定靠谱！

* 异步发送电子邮件

**from** threading **import** Thread  
**def** send\_async\_email(app, msg):  
 **with** app.app\_context():  
 mail.send(msg)  
**def** send\_email(to, subject, template, \*\*kwargs):  
 msg = Message(app.config[**'FLASKY\_MAIL\_SUBJECT\_PREFIX'**] + **' '** + subject,  
 sender=app.config[**'FLASKY\_MAIL\_SENDER'**], recipients=[to])  
 msg.body = render\_template(template + **'.txt'**, \*\*kwargs)  
 msg.html = render\_template(template + **'.html'**, \*\*kwargs)  
 t = Thread(target=send\_async\_email, args=[app, msg])  
 t.start()  
 **return** t

很多Flask扩展都假设已经存在激活的程序上下文和请求上下文。 Flask-Mail中的send()函数使用current\_app，因此必须激活程序上下文。但在不同线程中执行 send()函数时，程序上下文要使用app.app\_context()人工创建。