Day 2 - 编写数据库模块

2294次阅读

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python-app中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

此外，在一个Web App中，有多个用户会同时访问，系统以多进程或多线程模式来处理每个用户的请求。假设以多线程为例，每个线程在访问数据库时，都必须创建仅属于自身的连接，对别的线程不可见，否则，就会造成数据库操作混乱。

所以，我们还要创建一个简单可靠的数据库访问模型，在一个线程中，能既安全又简单地操作数据库。

为什么不选择SQLAlchemy？SQLAlchemy太庞大，过度地面向对象设计导致API太复杂。

所以我们决定自己设计一个封装基本的SELECT、INSERT、UPDATE和DELETE操作的db模块：transwarp.db。

设计db接口

设计底层模块的原则是，根据上层调用者设计简单易用的API接口，然后，实现模块内部代码。

假设transwarp.db模块已经编写完毕，我们希望以这样的方式来调用它：

首先，初始化数据库连接信息，通过create_engine()函数：

```
from transwarp import db
db.create_engine(user='root', password='password', database='test', host='127.0.0.1', port=3306)
```

然后，就可以直接操作SQL了。

如果需要做一个查询，可以直接调用select()方法，返回的是list，每一个元素是用dict表示的对应的行：

```
users = db.select('select * from user')
# users =>
# [
#     { "id": 1, "name": "Michael"},
#     { "id": 2, "name": "Bob"},
#     { "id": 3, "name": "Adam"}
# ]
```

如果要执行INSERT、UPDATE或DELETE操作，执行update()方法，返回受影响的行数：

```
n = db.update('insert into user(id, name) values(?, ?)', 4, 'Jack')
```

update()函数签名为：

```
update(sql, *args)
```

统一用?作为占位符，并传入可变参数来绑定，从根本上避免SQL注入攻击。

每个select()或update()调用，都隐含地自动打开并关闭了数据库连接，这样，上层调用者就完全不必关心数据库底层连接。

但是，如果要在一个数据库连接里执行多个SQL语句怎么办？我们用一个with语句实现：

```
with db.connection():
    db.select('...')
    db.update('...')
    db.update('...')
```

如果要在一个数据库事务中执行多个SQL语句怎么办？我们还是用一个with语句实现：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

实现db模块

由于模块是全局对象，模块变量是全局唯一变量，所以，有两个重要的模块变量：

```
# db.py

# 数据库引擎对象:
class _Engine(object):
    def __init__(self, connect):
        self._connect = connect
    def connect(self):
        return self._connect()

engine = None

# 持有数据库连接的上下文对象:
class _DbCtx(threading.local):
    def __init__(self):
        self.connection = None
        self.transactions = 0

    def is_init(self):
        return not self.connection is None

    def init(self):
        self.connection = _LasyConnection()
        self.transactions = 0

    def cleanup(self):
        self.connection.cleanup()
        self.connection = None

    def cursor(self):
        return self.connection.cursor()

_db_ctx = _DbCtx()
```

由于_db_ctx是threadlocal对象，所以，它持有的数据库连接对于每个线程看到的都是不一样的。任何一个线程都无法访问到其他线程持有的数据库连接。

有了这两个全局变量，我们继续实现数据库连接的上下文，目的是自动获取和释放连接：

```
class _ConnectionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_cleanup = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_cleanup = True
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        if self.should_cleanup:
            _db_ctx.cleanup()

def connection():
    return _ConnectionCtx()
```

定义了__enter__()和__exit__()的对象可以用于with语句，确保任何情况下__exit__()方法可以被调用。

把_ConnectionCtx的作用域作用到一个函数调用上，可以这么写：

```
with connection():
    do_some_db_operation()
```

但是更简单的写法是写个@decorator：

```
@with_connection
def do_some_db_operation():
    pass
```

这样，我们实现select()、update()方法就更简单了：

```
@with_connection
def select(sql, *args):
    pass

@with_connection
def update(sql, *args):
    pass
```

注意到Connection对象是存储在_DbCtx这个threadlocal对象里的，因此，嵌套使用with connection()也没有问题。_DbCtx永远检测当前是否已存在Connection，如果存在，直接使用，如果不存在，则打开一个新的Connection。

对于transaction也是类似的，with transaction()定义了一个数据库事务：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

函数作用域的事务也有一个简化的@decorator：

```
@with_transaction
def do_in_transaction():
    pass
```

事务也可以嵌套，内层事务会自动合并到外层事务中，这种事务模型足够满足99%的需求。

事务嵌套比Connection嵌套复杂一点，因为事务嵌套需要计数，每遇到一层嵌套就+1，离开一

层嵌套就-1，最后到0时提交事务：

```
class _TransactionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_close_conn = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_close_conn = True
        _db_ctx.transactions = _db_ctx.transactions + 1
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        _db_ctx.transactions = _db_ctx.transactions - 1
        try:
            if _db_ctx.transactions==0:
                if exctype is None:
                    self.commit()
                else:
                    self.rollback()
        finally:
            if self.should_close_conn:
                _db_ctx.cleanup()

    def commit(self):
        global _db_ctx
        try:
            _db_ctx.connection.commit()
        except:
            _db_ctx.connection.rollback()
            raise

    def rollback(self):
        global _db_ctx
        _db_ctx.connection.rollback()
```

最后，把select()和update()方法实现了，db模块就完成了。

Day 3 - 编写ORM

687次阅读

有了db模块，操作数据库直接写SQL就很方便。但是，我们还缺少ORM。如果有了ORM，就可以用类似这样的语句获取User对象：

```
user = User.get('123')
```

而不是写SQL然后再转换成User对象：

```
u = db.select_one('select * from users where id=?', '123')
user = User(**u)
```

所以我们开始编写ORM模块：transwarp.orm。

设计ORM接口

和设计db模块类似，设计ORM也是从上层调用者角度来设计。

我们先考虑如何定义一个User对象，然后把数据库表users和它关联起来。

```
from transwarp.orm import Model, StringField, IntegerField

class User(Model):
    __table__ = 'users'
    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在User类中的__table__、id和name是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述User对象和表的映射关系，而实例属性必须通过__init__()方法去初始化，所以两者互不干扰：

```
# 创建实例:
user = User(id=123, name='Michael')
# 存入数据库:
user.insert()
```

实现ORM模块

有了定义，我们就可以开始实现ORM模块。

首先要定义的是所有ORM映射的基类Model：

```
class Model(dict):
    __metaclass__ = ModelMetaclass

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

Model从dict继承，所以具备所有dict的功能，同时又实现了特殊方法__getattr__()和__setattr__()，所以又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

Model只是一个基类，如何将具体的子类如User的映射信息读取出来呢？答案就是通过metaclass：ModelMetaclass：

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        mapping = ... # 读取cls的Field字段
        primary_key = ... # 查找primary_key字段
        __table__ = cls.__talbe__ # 读取cls的__table__字段
        # 给cls增加一些字段：
        attrs['__mapping__'] = mapping
        attrs['__primary_key__'] = __primary_key__
        attrs['__table__'] = __table__
        return type.__new__(cls, name, bases, attrs)
```

这样，任何继承自Model的类（比如User），会自动通过ModelMetaclass扫描映射关系，并存储到自身的class中。

然后，我们往Model类添加class方法，就可以让所有子类调用class方法：

```
class Model(dict):

    ...

    @classmethod
    def get(cls, pk):
        d = db.select_one('select * from %s where %s=?' % (cls.__table__, cls.__primary_key__.name), pk)
        return cls(**d) if d else None
```

User类就可以通过类方法实现主键查找：

```
user = User.get('123')
```

往Model类添加实例方法，就可以让所有子类调用实例方法：

```
class Model(dict):

    ...

    def insert(self):
        params = {}
        for k, v in self.__mappings__.iteritems():
            params[v.name] = getattr(self, k)
        db.insert(self.__table__, **params)
        return self
```

这样，就可以把一个User实例存入数据库：

```
user = User(id=123, name='Michael')
user.insert()
```

最后一步是完善ORM，对于查找，我们可以实现以下方法：

- find_first()

- find_all()
- find_by()

对于count，可以实现：

- count_all()
- count_by()

以及update()和delete()方法。

最后看看我们实现的ORM模块一共多少行代码？加上注释和doctest才仅仅300多行。用Python写一个ORM是不是很容易呢？

Day 4 - 编写Model

363次阅读

有了ORM，我们就可以把Web App需要的3个表用Model表示出来：

```
import time, uuid

from transwarp.db import next_id
from transwarp.orm import Model, StringField, BooleanField, FloatField, TextField

class User(Model):
    __table__ = 'users'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    email = StringField(updatable=False, ddl='varchar(50)')
    password = StringField(ddl='varchar(50)')
    admin = BooleanField()
    name = StringField(ddl='varchar(50)')
    image = StringField(ddl='varchar(500)')
    created_at = FloatField(updatable=False, default=time.time)

class Blog(Model):
    __table__ = 'blogs'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    user_id = StringField(updatable=False, ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(updatable=False, default=time.time)

class Comment(Model):
    __table__ = 'comments'

    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    blog_id = StringField(updatable=False, ddl='varchar(50)')
    user_id = StringField(updatable=False, ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(updatable=False, default=time.time)
```

在编写ORM时，给一个Field增加一个default参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用insert()时自动计算。

例如，主键id的缺省值是函数next_id，创建时间created_at的缺省值是函数time.time，可以自动设置当前日期和时间。

日期和时间用float类型存储在数据库中，而不是datetime类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个float到str的转换，也非常容易。

初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

```
-- schema.sql
```

```
drop database if exists awesome;

create database awesome;

use awesome;

grant select, insert, update, delete on awesome.* to 'www-data'@'localhost' identified by 'www-data';

create table users (
    `id` varchar(50) not null,
    `email` varchar(50) not null,
    `password` varchar(50) not null,
    `admin` bool not null,
    `name` varchar(50) not null,
    `image` varchar(500) not null,
    `created_at` real not null,
    unique key `idx_email` (`email`),
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table blogs (
    `id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `name` varchar(50) not null,
    `summary` varchar(200) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table comments (
    `id` varchar(50) not null,
    `blog_id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;
```

如果表的数量很多，可以从Model对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

```
$ mysql -u root -p < schema.sql
```

我们就完成了数据库表的初始化。

编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于User对象，我们就可以做如下操作：

```
# test_db.py

from models import User, Blog, Comment
```

```
from transwarp import db

db.create_engine(user='www-data', password='www-data', database='awesome')

u = User(name='Test', email='test@example.com', password='1234567890', image='about:blank')

u.insert()

print 'new user id:', u.id

u1 = User.find_first('where email=?', 'test@example.com')
print 'find user\'s name:', u1.name

u1.delete()

u2 = User.find_first('where email=?', 'test@example.com')
print 'find user:', u2
```

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

Day 5 - 编写Web框架

542次阅读

在正式开始Web开发前，我们需要编写一个Web框架。

为什么不选择一个现成的Web框架而是自己从头开发呢？我们来考察一下现有的流行的Web框架：

Django：一站式开发框架，但不利于定制化；

web.py：使用类而不是更简单的函数来处理URL，并且URL映射是单独配置的；

Flask：使用@decorator的URL路由不错，但框架对应用程序的代码入侵太强；

bottle：缺少根据URL模式进行拦截的功能，不利于做权限检查。

所以，我们综合几种框架的优点，设计一个简单、灵活、入侵性极小的Web框架。

设计Web框架

一个简单的URL框架应该允许以@decorator方式直接把URL映射到函数上：

```
# 首页:
@get('/')
def index():
    return '<h1>Index page</h1>'

# 带参数的URL:
@get('/user/:id')
def show_user(id):
    user = User.get(id)
    return 'hello, %s' % user.name
```

有没有@decorator不改变函数行为，也就是说，Web框架的API入侵性很小，你可以直接测试函数show_user(id)而不需要启动Web服务器。

函数可以返回str、unicode以及iterator，这些数据可以直接作为字符串返回给浏览器。

其次，Web框架要支持URL拦截器，这样，我们就可以根据URL做权限检查：

```
@interceptor('/manage/')
def check_manage_url(next):
    if current_user.isAdmin():
        return next()
    else:
        raise seeother('/signin')
```

拦截器接受一个next函数，这样，一个拦截器可以决定调用next()继续处理请求还是直接返回。

为了支持MVC，Web框架需要支持模板，但是我们不限定使用哪一种模板，可以选择jinja2，也可以选择mako、Cheetah等等。

要统一模板的接口，函数可以返回dict并配合@view来渲染模板：

```
@view('index.html')
@get('/')
```

```
def index():
    return dict(blogs=get_recent_blogs(), user=get_current_user())
```

如果需要从form表单或者URL的querystring获取用户输入的数据，就需要访问request对象，如果要设置特定的Content-Type、设置Cookie等，就需要访问response对象。request和response对象应该从一个唯一的ThreadLocal中获取：

```
@get('/test')
def test():
    input_data = ctx.request.input()
    ctx.response.content_type = 'text/plain'
    ctx.response.set_cookie('name', 'value', expires=3600)
    return 'result'
```

最后，如果需要重定向、或者返回一个HTTP错误码，最好的方法是直接抛出异常，例如，重定向到登陆页：

```
raise seeother('/signin')
```

返回404错误：

```
raise notfound()
```

基于以上接口，我们就可以实现Web框架了。

实现Web框架

最基本的几个对象如下：

```
# transwarp/web.py

# 全局ThreadLocal对象：
ctx = threading.local()

# HTTP错误类:
class HttpError(Exception):
    pass

# request对象:
class Request(object):
    # 根据key返回value:
    def get(self, key, default=None):
        pass

    # 返回key-value的dict:
    def input(self):
        pass

    # 返回URL的path:
    @property
    def path_info(self):
        pass

    # 返回HTTP Headers:
    @property
    def headers(self):
        pass

    # 根据key返回Cookie value:
    def cookie(self, name, default=None):
        pass
```

```
# response对象:
class Response(object):
    # 设置header:
    def set_header(self, key, value):
        pass

    # 设置Cookie:
    def set_cookie(self, name, value, max_age=None, expires=None, path='/'):
        pass

    # 设置status:
    @property
    def status(self):
        pass
    @status.setter
    def status(self, value):
        pass

# 定义GET:
def get(path):
    pass

# 定义POST:
def post(path):
    pass

# 定义模板:
def view(path):
    pass

# 定义拦截器:
def interceptor(pattern):
    pass

# 定义模板引擎:
class TemplateEngine(object):
    def __call__(self, path, model):
        pass

# 缺省使用jinja2:
class Jinja2TemplateEngine(TemplateEngine):
    def __init__(self, templ_dir, **kw):
        from jinja2 import Environment, FileSystemLoader
        self._env = Environment(loader=FileSystemLoader(templ_dir), **kw)

    def __call__(self, path, model):
        return self._env.get_template(path).render(**model).encode('utf-8')
```

把上面的定义填充完毕，我们就只剩下一件事情：定义全局WSGIApplication的类，实现WSGI接口，然后，通过配置启动，就完成了整个Web框架的工作。

设计WSGIApplication要充分考虑开发模式（Development Mode）和产品模式（Production Mode）的区分。在产品模式下，WSGIApplication需要直接提供WSGI接口给服务器，让服务器调用该接口，而在开发模式下，我们更希望能通过app.run()直接启动服务器进行开发调试：

```
wsgi = WSGIApplication()
if __name__ == '__main__':
    wsgi.run()
else:
    application = wsgi.get_wsgi_application()
```

因此，WSGIApplication定义如下：

```
class WSGIApplication(object):
    def __init__(self, document_root=None, **kw):
        pass

    # 添加一个URL定义:
    def add_url(self, func):
        pass

    # 添加一个Interceptor定义:
    def add_interceptor(self, func):
        pass

    # 设置TemplateEngine:
    @property
    def template_engine(self):
        pass

    @template_engine.setter
    def template_engine(self, engine):
        pass

    # 返回WSGI处理函数:
    def get_wsgi_application(self):
        def wsgi(env, start_response):
            pass
        return wsgi

    # 开发模式下直接启动服务器:
    def run(self, port=9000, host='127.0.0.1'):
        from wsgiref.simple_server import make_server
        server = make_server(host, port, self.get_wsgi_application())
        server.serve_forever()
```

把WSGIApplication类填充完毕，我们就得到了一个完整的Web框架。

Day 6 - 添加配置文件

234次阅读

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的.properties或者.yaml等配置文件。

默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置文件命名为config_default.py：

```
# config_default.py

configs = {
    'db': {
        'host': '127.0.0.1',
        'port': 3306,
        'user': 'www-data',
        'password': 'www-data',
        'database': 'awesome'
    },
    'session': {
        'secret': 'AwEsOmE'
    }
}
```

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的host等信息，直接修改config_default.py不是一个好办法，更好的方法是编写一个config_override.py，用来覆盖某些默认设置：

```
# config_override.py

configs = {
    'db': {
        'host': '192.168.0.100'
    }
}
```

把config_default.py作为开发环境的标准配置，把config_override.py作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从config_override.py读取。为了简化读取配置文件，可以把所有配置读取到统一的config.py中：

```
# config.py
configs = config_default.configs

try:
    import config_override
    configs = merge(configs, config_override.configs)
except ImportError:
    pass
```

这样，我们就完成了App的配置。

Day 7 - 编写MVC

310次阅读

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的@decorator和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

```
# urls.py
from transwarp.web import get, view
from models import User, Blog, Comment

@view('test_users.html')
@get('/')
def test_users():
    users = User.find_all()
    return dict(users=users)
```

@view指定的模板文件是test_users.html，所以我们在模板的根目录templates下创建test_users.html：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test users - Awesome Python Webapp</title>
</head>
<body>
    <h1>All users</h1>
    {% for u in users %}
    <p>{{ u.name }} / {{ u.email }}</p>
    {% endfor %}
</body>
</html>
```

接下来，我们创建一个Web App的启动文件wsgiapp.py，负责初始化数据库、初始化Web框架，然后加载urls.py，最后启动Web服务：

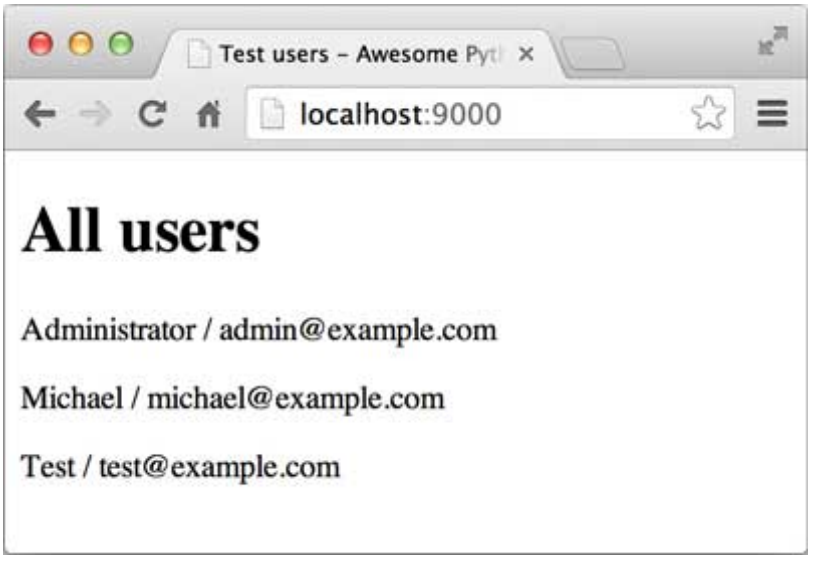
```
# wsgiapp.py
import logging; logging.basicConfig(level=logging.INFO)
import os

from transwarp import db
from transwarp.web import WSGIApplication, Jinja2TemplateEngine

from config import configs

# 初始化数据库:
db.create_engine(**configs.db)

# 创建一个WSGIApplication:
wsgi = WSGIApplication(os.path.dirname(os.path.abspath(__file__)))
# 初始化jinja2模板引擎:
template_engine = Jinja2TemplateEngine(os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates'))
wsgi.template_engine = template_engine

# 加载带有@get/@post的URL处理函数:
import urls
wsgi.add_module(urls)

# 在9000端口上启动本地测试服务器:
if __name__ == '__main__':
    wsgi.run(9000)
```

如果一切顺利，可以用命令行启动Web服务器：

```
$ python wsgiapp.py
```

然后，在浏览器中访问http://localhost:9000/。

如果数据库的users表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给users表添加几条记录，然后再访问：



Day 8 - 构建前端

338次阅读

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择uikit这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从uikit首页下载打包的资源文件。

所有的静态资源文件我们统一放到www/static目录下，并按照类别归类：

```
static/
+- css/
|  +- addons/
|  |  +- uikit.addons.min.css
|  |  +- uikit.almost-flat.addons.min.css
|  |  +- uikit.gradient.addons.min.css
|  +- awesome.css
|  +- uikit.almost-flat.addons.min.css
|  +- uikit.gradient.addons.min.css
|  +- uikit.min.css
+- fonts/
|  +- fontawesome-webfont.eot
|  +- fontawesome-webfont.ttf
|  +- fontawesome-webfont.woff
|  +- FontAwesome.otf
+- js/
   +- awesome.js
   +- html5.js
   +- jquery.min.js
   +- uikit.min.js
```

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

```
<html>
    <% include file="inc_header.html" %>
    <% include file="index_body.html" %>
    <% include file="inc_footer.html" %>
</html>
```

这样，相同的部分inc_header.html和inc_footer.html就可以共享。

但是include方法不利于页面整体结构的维护。jinjia2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

```
<!-- base.html -->
<html>
    <head>
        <title>{% block title%} 这里定义了一个名为title的block {% endblock %}</title>
    </head>
    <body>
        {% block content %} 这里定义了一个名为content的block {% endblock %}
    </body>
</html>
```

对于子模板a.html，只需要把父模板的title和content替换掉：

```
{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}
<h1>Chapter A</h1>
<p>blablabla...</p>
{% endblock %}
```

对于子模板b.html，如法炮制：

```
{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}
<h1>Chapter B</h1>
```

```
<ul>
   <li>list 1</li>
   <li>list 2</li>
</ul>
{% endblock %}
```

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板__base__.html的编写：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    {% block meta %}<!-- block meta  -->{% endblock %}
    <title>{% block title %} ? {% endblock %} - Awesome Python Webapp</title>
    <link rel="stylesheet" href="/static/css/uikit.min.css">
    <link rel="stylesheet" href="/static/css/uikit.gradient.min.css">
    <link rel="stylesheet" href="/static/css/awesome.css" />
    <script src="/static/js/jquery.min.js"></script>
    <script src="/static/js/md5.js"></script>
    <script src="/static/js/uikit.min.js"></script>
    <script src="/static/js/awesome.js"></script>
    {% block beforehead %}<!-- before head  -->{% endblock %}
</head>
<body>
    <nav class="uk-navbar uk-navbar-attached uk-margin-bottom">
        <div class="uk-container uk-container-center">
            <a href="/" class="uk-navbar-brand">Awesome</a>
            <ul class="uk-navbar-nav">
                <li data-url="blogs"><a href="/"><i class="uk-icon-home"></i> 日志</a></li>
                <li><a target="_blank" href="#"><i class="uk-icon-book"></i> 教程</a></li>
                <li><a target="_blank" href="#"><i class="uk-icon-code"></i> 源码</a></li>
            </ul>
            <div class="uk-navbar-flip">
                <ul class="uk-navbar-nav">
                {% if user %}
                    <li class="uk-parent" data-uk-dropdown>
                        <a href="#0"><i class="uk-icon-user"></i> {{ user.name }}</a>
                        <div class="uk-dropdown uk-dropdown-navbar">
                            <ul class="uk-nav uk-nav-navbar">
                                <li><a href="/signout"><i class="uk-icon-sign-out"></i> 登出</a></li>
                            </ul>
                        </div>
                    </li>
                {% else %}
                    <li><a href="/signin"><i class="uk-icon-sign-in"></i> 登陆</a></li>
                    <li><a href="/register"><i class="uk-icon-edit"></i> 注册</a></li>
                {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <div class="uk-container uk-container-center">
        <div class="uk-grid">
            <!-- content -->
            {% block content %}
            {% endblock %}
            <!-- // content -->
        </div>
    </div>

    <div class="uk-margin-large-top" style="background-color:#eee; border-top:1px solid #ccc;">
        <div class="uk-container uk-container-center uk-text-center">
            <div class="uk-panel uk-margin-top uk-margin-bottom">
                <p>
                    <a target="_blank" href="#" class="uk-icon-button uk-icon-weibo"></a>
                    <a target="_blank" href="#" class="uk-icon-button uk-icon-github"></a>
                    <a target="_blank" href="#" class="uk-icon-button uk-icon-linkedin-square"></a>
                    <a target="_blank" href="#" class="uk-icon-button uk-icon-twitter"></a>
                </p>
                <p>Powered by <a href="#">Awesome Python Webapp</a>. Copyright &amp;copy; 2014. [<a href="/manage/" target="_blank">Manage</a>]</p>
                <p><a href="http://www.liaoxuefeng.com/" target="_blank">www.liaoxuefeng.com</a>. All rights reserved.</p>
                <a target="_blank" href="#"><i class="uk-icon-html5" style="font-size:64px; color: #444;"></i></a>
            </div>
        </div>
    </div>
</body>
</html>
```

__base__.html定义的几个block作用如下：

用于子页面定义一些meta，例如rss feed：

```
{% block meta %} ... {% endblock %}
```

覆盖页面的标题：

```
{% block title %} ... {% endblock %}
```

子页面可以在标签关闭前插入JavaScript代码：

```
{% block beforehead %} ... {% endblock %}
```

子页面的content布局和内容：

```
{% block content %}
...
{% endblock %}
```

我们把首页改造一下，从__base__.html继承一个blogs.html：

```
{% extends '__base__.html' %}

{% block title %}日志{% endblock %}

{% block content %}

    <div class="uk-width-medium-3-4">
    {% for blog in blogs %}
        <article class="uk-article">
            <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></h2>
            <p class="uk-article-meta">发表于{{ blog.created_at}}</p>
            <p>{{ blog.summary }}</p>
            <p><a href="/blog/{{ blog.id }}">继续阅读 <i class="uk-icon-angle-double-right"></i></a></p>
        </article>
        <hr class="uk-article-divider">
    {% endfor %}
    </div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">编程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">读书</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Python教程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Git教程</a></li>
            </ul>
        </div>
    </div>

{% endblock %}
```

相应地，首页URL的处理函数更新如下：

```
@view('blogs.html')
@get('/')
def index():
    blogs = Blog.find_all()
    # 查找登陆用户:
    user = User.find_first('where email=?', 'admin@example.com')
    return dict(blogs=blogs, user=user)
```

往MySQL的blogs表中手动插入一些数据，我们就可以看到一个真正的首页了。但是Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

```
<p class="uk-article-meta">发表于{{ blog.created_at }}</p>
```

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个datetime的filter，在模板里用法如下：

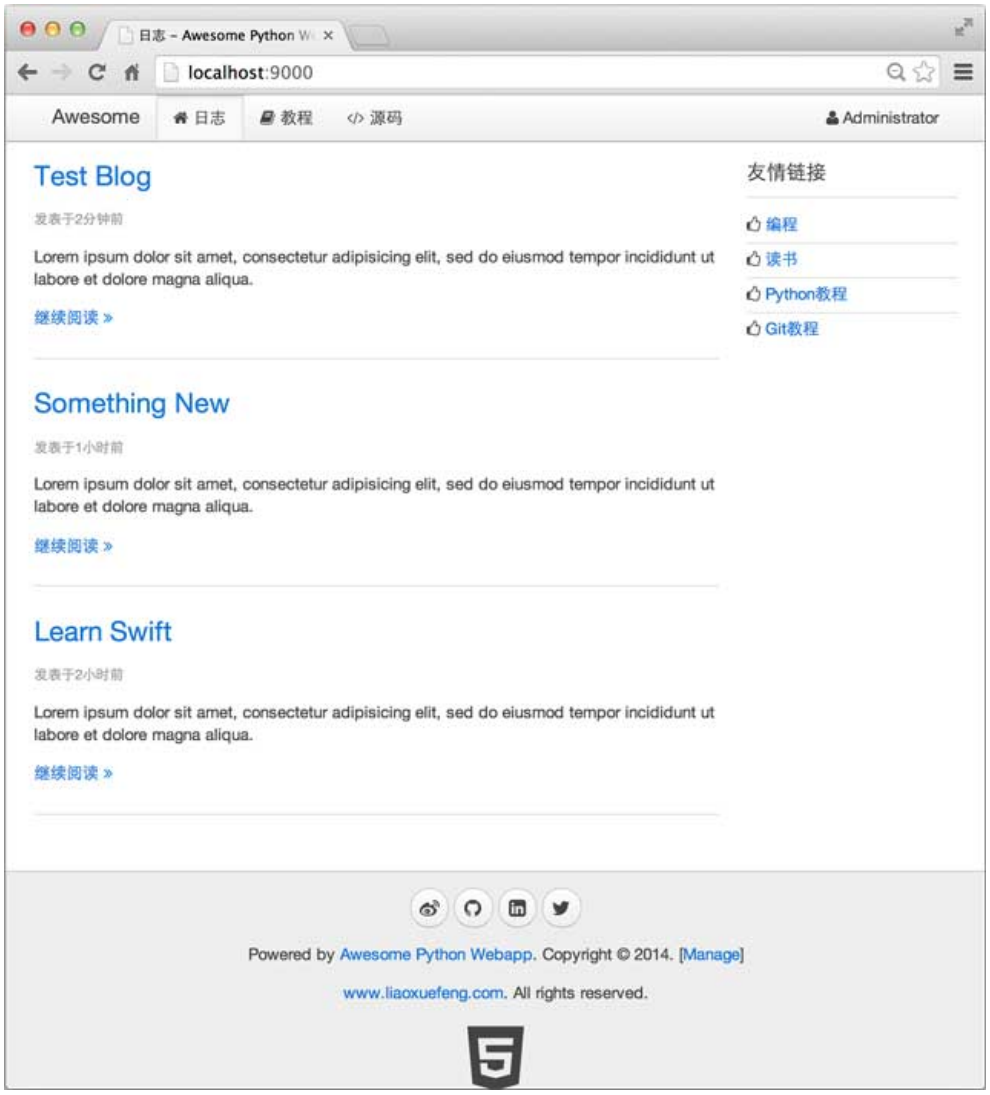
```
<p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
```

filter需要在初始化jinja2时设置。修改wsgiapp.py相关代码如下：

```
# wsgiapp.py:

...

# 定义datetime_filter，输入是t，输出是unicode字符串:
def datetime_filter(t):
    delta = int(time.time() - t)
    if delta < 60:
        return u'1分钟前'
    if delta < 3600:
        return u'%s分钟前' % (delta // 60)
    if delta < 86400:
        return u'%s小时前' % (delta // 3600)
    if delta < 604800:
        return u'%s天前' % (delta // 86400)
    dt = datetime.fromtimestamp(t)
    return u'%s年%s月%s日' % (dt.year, dt.month, dt.day)

template_engine = Jinja2TemplateEngine(os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates'))
# 把filter添加到jinjia2，filter名称为datetime，filter本身是一个函数对象:
```

```
template_engine.add_filter('datetime', datetime_filter)

wsgi.template_engine = template_engine
```

现在，完善的首页显示如下：



Day 9 - 编写API

374次阅读

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入http://localhost:9000/blog/123，就可以看到id为123的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取http://localhost:9000/api/blogs/123，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个@api来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

```
@api
@get('/api/users')
def api_get_users():
    users = User.find_by('order by created_at desc')
    # 把用户的口令隐藏掉:
    for u in users:
        u.password = '******'
    return dict(users=users)
```

所以，@api这个decorator只要编写好了，就可以把任意的URL处理函数变成API调用。

新建一个apis.py，编写@api负责把函数的返回结果序列化为JSON：

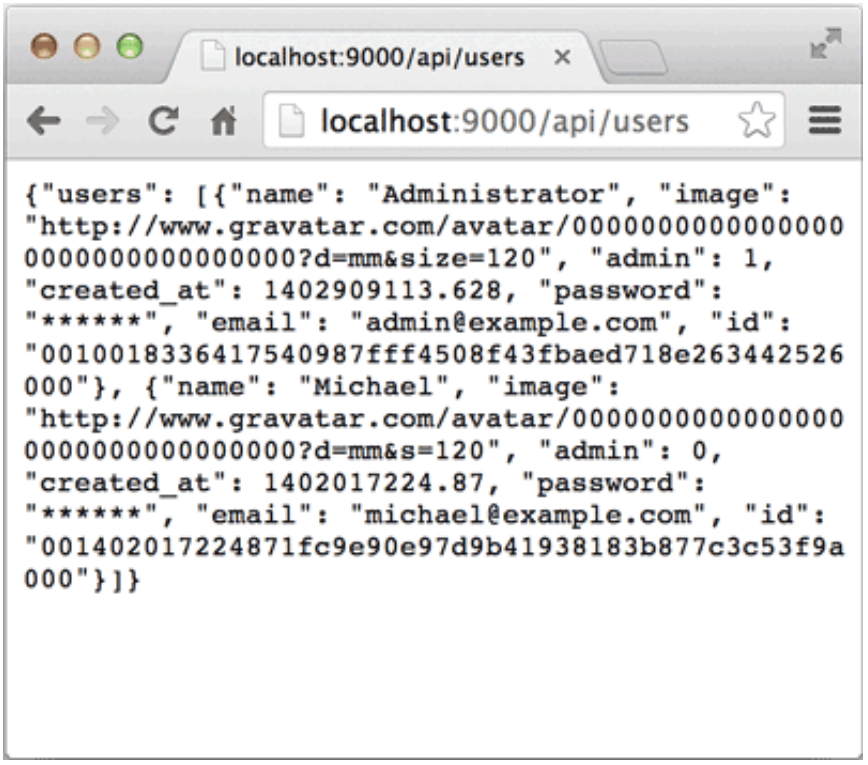
```
def api(func):
    @functools.wraps(func)
    def _wrapper(*args, **kw):
        try:
            r = json.dumps(func(*args, **kw))
        except APIError, e:
            r = json.dumps(dict(error=e.error, data=e.data, message=e.message))
        except Exception, e:
            r = json.dumps(dict(error='internalerror', data=e.__class__.__name__, message=e.message))
        ctx.response.content_type = 'application/json'
        return r
    return _wrapper
```

@api需要对Error进行处理。我们定义一个APIError，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为internalerror。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还

需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入http://localhost:9000/api/users，就可以看到返回的JSON：



Day 10 - 用户注册和登录

305次阅读

用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

```
_RE_MD5 = re.compile(r'^[0-9a-f]{32}$')

@api
@post('/api/users')
def register_user():
    i = ctx.request.input(name='', email='', password='')
    name = i.name.strip()
    email = i.email.strip().lower()
    password = i.password
    if not name:
        raise APIValueError('name')
    if not email or not _RE_EMAIL.match(email):
        raise APIValueError('email')
    if not password or not _RE_MD5.match(password):
        raise APIValueError('password')
    user = User.find_first('where email=?', email)
    if user:
        raise APIError('register:failed', 'email', 'Email is already in use.')
    user = User(name=name, email=email, password=password, image='http://www.gravatar.com/avatar/%s?d=mm&s=120' % hashlib.md5(email).hexdigest())
    user.insert()
    return user
```

注意用户口令是客户端传递的经过MD5计算后的32位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

```
{% extends '__base__.html' %}

{% block title %}注册{% endblock %}

{% block beforehead %}

<script>
function check_form() {
    $('#password').val(CryptoJS.MD5($('#password1').val()).toString());
    return true;
}
</script>

{% endblock %}

{% block content %}

<div class="uk-width-2-3">
    <h1>欢迎注册！</h1>
    <form id="form-register" class="uk-form uk-form-stacked" onsubmit="return check_form()">
        <div class="uk-alert uk-alert-danger uk-hidden"></div>
        <div class="uk-form-row">
            <label class="uk-form-label">名字:</label>
            <div class="uk-form-controls">
                <input name="name" type="text" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">电子邮件:</label>
            <div class="uk-form-controls">
                <input name="email" type="text" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">输入口令:</label>
            <div class="uk-form-controls">
                <input id="password1" type="password" class="uk-width-1-1">
                <input id="password" name="password" type="hidden">
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">重复口令:</label>
            <div class="uk-form-controls">
                <input name="password2" type="password" maxlength="50" placeholder="重复口令" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <button type="submit" class="uk-button uk-button-primary"><i class="uk-icon-user"></i> 注册</button>
        </div>
    </form>
</div>

{% endblock %}
```

这样我们就把用户注册的功能完成了：

用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如MD5），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

```
"用户id" + "过期时间" + MD5("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

- 用户id
- 过期时间
- MD5值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

```
MD5("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

并与浏览器cookie中的MD5进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于MD5是一种单向算法，即可以通过原始字符串计算出MD5，但无法通过MD5反推出原始字符串。

所以登录API可以实现如下：

```
@api
@post('/api/authenticate')
def authenticate():
    i = ctx.request.input()
    email = i.email.strip().lower()
    password = i.password
    user = User.find_first('where email=?', email)
    if user is None:
        raise APIError('auth:failed', 'email', 'Invalid email.')
    elif user.password != password:
        raise APIError('auth:failed', 'password', 'Invalid password.')
    max_age = 604800
    cookie = make_signed_cookie(user.id, user.password, max_age)
    ctx.response.set_cookie(_COOKIE_NAME, cookie, max_age=max_age)
    user.password = '******'
    return user

# 计算加密cookie:
def make_signed_cookie(id, password, max_age):
    expires = str(int(time.time() + max_age))
    L = [id, expires, hashlib.md5('%s-%s-%s-%s' % (id, password, expires, _COOKIE_KEY)).hexdigest()]
    return '-'.join(L)
```

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用拦截器在处理URL之前，把cookie解析出来，并将登录用户绑定到ctx.request对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

```
@interceptor('/')
def user_interceptor(next):
    user = None
    cookie = ctx.request.cookies.get(_COOKIE_NAME)
    if cookie:
        user = parse_signed_cookie(cookie)
    ctx.request.user = user
    return next()

# 解密cookie:
def parse_signed_cookie(cookie_str):
    try:
        L = cookie_str.split('-')
        if len(L) != 3:
            return None
        id, expires, md5 = L
        if int(expires) < time.time():
            return None
        user = User.get(id)
        if user is None:
            return None
        if md5 != hashlib.md5('%s-%s-%s-%s' % (id, user.password, expires, _COOKIE_KEY)).hexdigest():
            return None
        return user
    except:
        return None
```

这样，我们就完成了用户注册和登录的功能。

Day 11 - 编写日志创建页

98次阅读

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog：

```
@api
@post('/api/blogs')
def api_create_blog():
    i = ctx.request.input(name='', summary='', content='')
    name = i.name.strip()
    summary = i.summary.strip()
    content = i.content.strip()
    if not name:
        raise APIValueError('name', 'name cannot be empty.')
    if not summary:
        raise APIValueError('summary', 'summary cannot be empty.')
    if not content:
        raise APIValueError('content', 'content cannot be empty.')
    user = ctx.request.user
    blog = Blog(user_id=user.id, user_name=user.name, name=name, summary=summary, content=content)
    blog.insert()
    return blog
```

编写后端Python代码不但很简单，而且非常容易测试，上面的API：`api_create_blog()`本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

```
s = '<html><head><title>'
  + title
  + '</title></head><body>'
  + body
  + '</body></html>'
```

显然这种方式完全不具备可维护性。所以有第二种模板方式：

```
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    {{ body }}
</body>
</html>
```

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的MVVM：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

```
<script>
var blog = {
    name: 'hello',
    summary: 'this is summary',
    content: 'this is content...'
};
</script>
```

View是纯HTML：

```
<form action="/api/blogs" method="post">
    <input name="name">
    <input name="summary">
    <textarea name="content"></textarea>
    <button type="submit">OK</button>
</form>
```

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择Vue这个简单易用的MVVM框架来实现创建Blog的页面templates/manage_blog_edit.html：

```
{% extends '__base__.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}

<script>
var
    action = '{{ action }}',
    redirect = '{{ redirect }}';

var vm;

$(function () {
    vm = new Vue({
        el: '#form-blog',
        data: {
            name: '',
            summary: '',
            content: ''
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                postApi(action, this.$data, function (err, r) {
                    if (err) {
                        alert(err);
                    }
```

```
                    else {
                        alert('保存成功！');
                        return location.assign(redirect);
                    }
                });
            }
        }
    });
});
</script>
{% endblock %}

{% block content %}
<div class="uk-width-1-1">
    <form id="form-blog" v-on="submit: submit" class="uk-form uk-form-stacked">
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <input v-model="name" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <textarea v-model="summary" rows="4" class="uk-width-1-1"></textarea>
            </div>
        </div>
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <textarea v-model="content" rows="8" class="uk-width-1-1"></textarea>
            </div>
        </div>
        <div class="uk-form-row">
            <button type="submit" class="uk-button uk-button-primary">保存</button>
        </div>
    </form>
</div>
{% endblock %}
```

初始化Vue时，我们指定3个参数：

el：根据选择器查找绑定的View，这里是#form-blog，就是id为form-blog的DOM，对应的是一个<form>标签；

data：JavaScript对象表示的Model，我们初始化为{ name: '', summary: '', content: ''}；

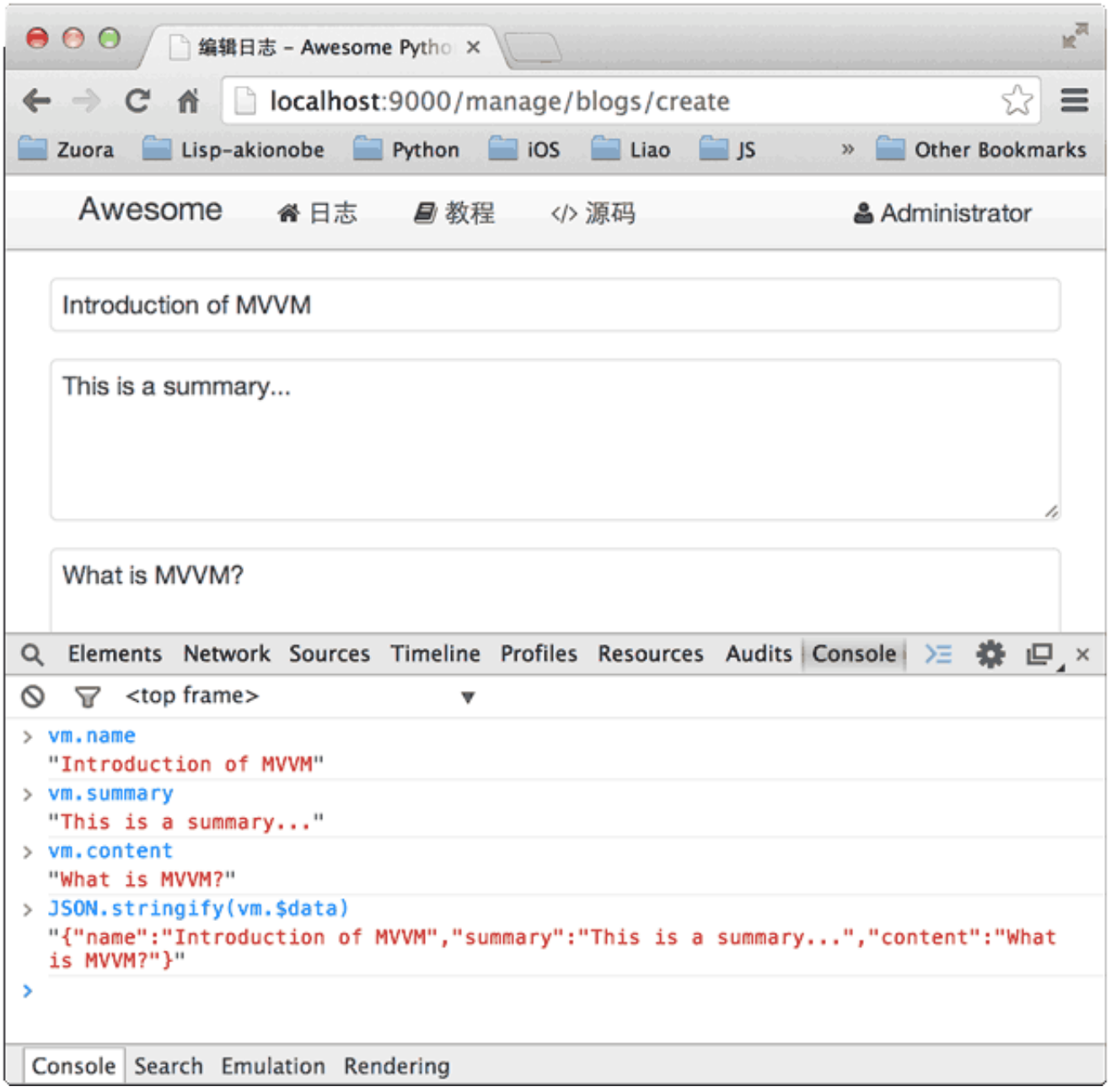
methods：View可以触发的JavaScript函数，submit就是提交表单时触发的函数。

接下来，我们在<form>标签中，用几个简单的v-model，就可以让Vue把Model和View关联起来：

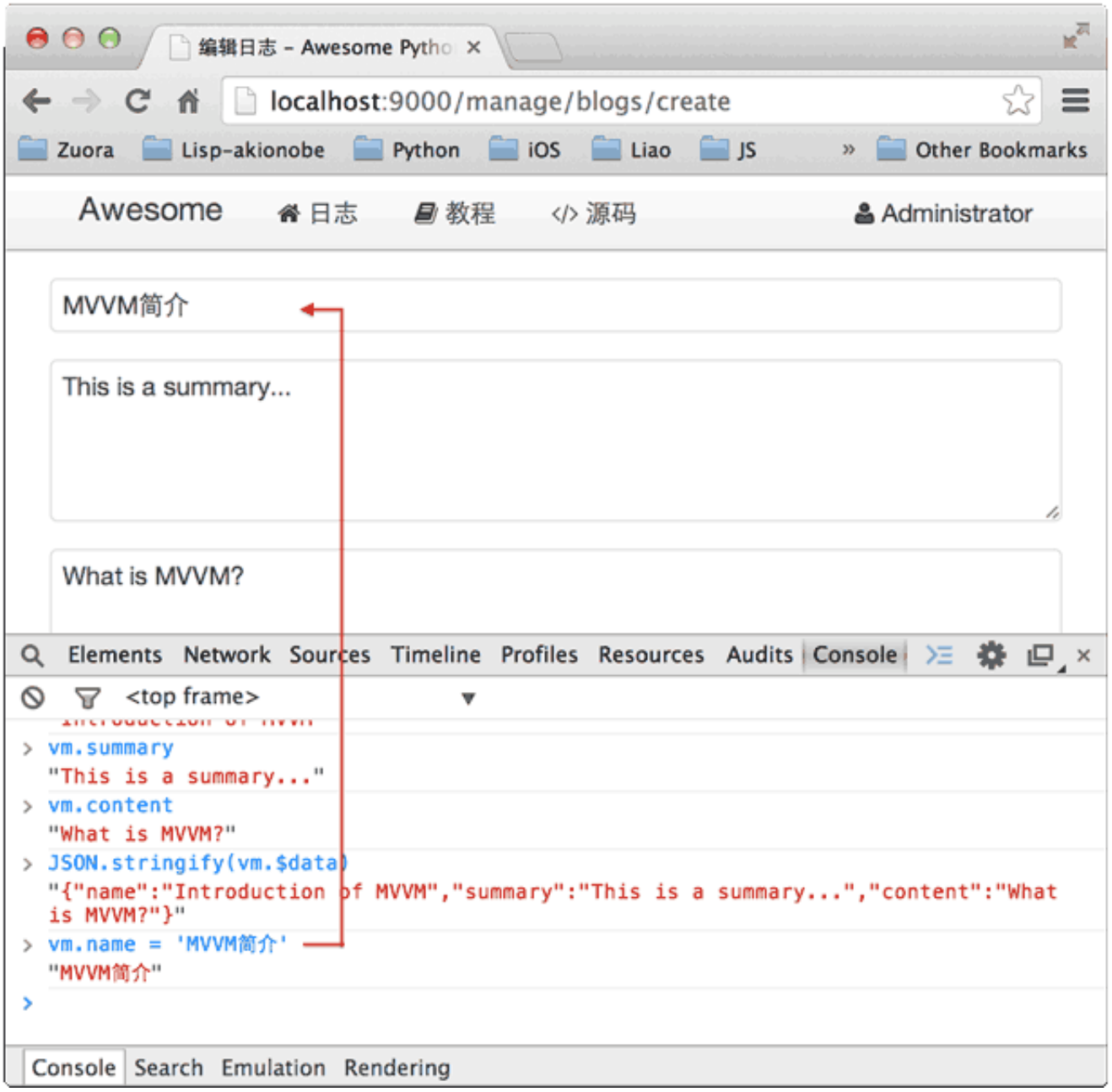
```
<!-- input的value和Model的name关联起来了 -->
<input v-model="name" class="uk-width-1-1">
```

Form表单通过<form v-on="submit: submit">把提交表单的事件关联到submit方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过vm.name访问单个属性，或者通过vm.$data访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入`vm.name = 'MVVM简介'`，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。
