PYAPE

发行版本 *0.3*

zrong

2024 年 03 月 21 日

目录:

PYAPE [paɪp] = A Application Programming Environment of Python.

Pyape 是我在开发 Flask 应用程序过程中积累的一个开发框架。准确的说，这不算一个框架，而是一组集合。我将开发 Web 以及 API 应用程序过程中积累的一些好用的工具和常用功能进行了简单的封装，整合在一起，方便快速启动一个新项目。

PYAPE 的完整文档：https://pyape.rtfd.io/

PYAPE 的 github 主页：https://github.com/zrong/pyape

PYAPE 我的博客上的页面：https://blog.zengrong.net/pyape/

PYAPE 的特点如下：

CHAPTER 1

集成命令行

通过对 Fabric 的集成，使用统一的命令行工具来实现如下功能：

1. 生成配置文件
2. 将程序部署到远程服务器
3. 控制远程服务器的运行

详细说明请阅读：命令行。

CHAPTER 2

多开发环境支持

可配置多套开发环境，方便同时支持本地开发、局域网开发、互联网测试和正式服部署。

详细说明请阅读：多开发环境支持。

CHAPTER 3

模版支持与配置合并

pyape 的命令行工具支持多级配置合并，方便在多个配置中共用数据，不必重复输入配置。

pyape 允许自定义配置生成模版。

CHAPTER 4

环境变量替换支持

pyape 的配置文件模版机制支持从环境变量中获取实际值，这样可以避免将敏感信息写入配置文件提交到 CVS 造成安全隐患。

详细说明请阅读：替换机制。

CHAPTER 5

SQLAlchemy 支持

Pyape 集成了 SQLAlchemy 支持。与 Flask-SQLAlchemy 不同，Pyape 直接使用标准的 SQLAlchemy 语法。这更加方便升级到未来的 SQLAlchemy 2.0 版本。

Use Flask and SQLalchemy, not Flask-SQLAlchemy 这篇文章的观点，我也是赞同的。

CHAPTER 6

Redis 支持

基于 flask-redis 修改，使其支持多个 Redis 数据库。

CHAPTER 7

Logging 集成

支持 ZeroMQHandler、RedisHandler，提供 get_logger 和 get_logging_handler 方便从配置中直接生成 Logger 和 Handler 对象。详情参见 pyape_logging 。

7.1 使用

pyape 支持 python3.9 及以上版本。

7.1.1 安装

安装开发版

国内:

```
pip install git+https://gitee.com/zrong/pyape@develop
```

全球:

```
pip install git+https://github.com/zrong/pyape@develop
```

安装稳定版

由于架构变更，请使用 pyape 开发版。

```
pip install pyape
```

7.1.2 创建项目

创建项目文件夹，项目名称以 project 为例：

```
mkdir project
cd project
python -m venv venv
source venv/bin/activate
pip install git+https://github.com/zrong/pyape@develop

# or

pip install pyape
```

7.1.3 初始化项目

进入虚拟环境，执行初始化命令，在项目文件夹下生成 4 个文件：`README.md/wsgi.py/pyape.toml/.gitignore`。

亦可以用 `-C` 参数指定项目文件夹。详情见*pyape init*。

```
(venv) cd project
(venv) pyape init
```

备注： 这里创建的是一个空项目，需要在这个项目的基础上进行进一步开发。开发遵循 Flask 程序的标准。

进一步阅读：

- 进一步完善项目，请阅读：一个最小化的 *app*；
- 配置*wsgi.py* 作为项目入口；
- 对*pyape.toml* 进行详细配置；
- 阅读开发 文档。

7.1.4 部署项目

若配置文件中存在 `prod` 这个环境，可以使用 pyape deploy 命令将项目部署到远程服务器。

远程服务器配置参见配置文件中的*[FABRIC]* 段落。

```
(venv) pyape deploy --env prod
```

7.1.5 启动项目/停止项目/重载项目

下面的命令使用 gunicorn 作为服务器。

```
# 启动项目
(venv) pyape start --env prod --server gunicorn

# 停止项目
(venv) pyape stop --env prod --server gunicorn

# 优雅重载项目
(venv) pyape reload --env prod --server gunicorn
```

7.1.6 部署并重载项目

在调试时经常需要部署并重载项目，使用 pyape dar 来执行这个操作。

dar = Deploy And Reload

下面的命令使用 gunicorn 作为服务器。

```
(venv) pyape dar --env prod --server gunicorn
```

7.1.7 生成 SECRET_KEY

`SECRET_KEY` 是 Flask 程序必须包含的配置。使用*pyape gen* 命令可以生成一个标准的 `SECRET_KEY`。

不带参数执行命令，可以生成 `secret-key`：

```
pyape gen
{'secret-key': 'HK-VHH4C0ijLjOYBYrO7L2ACsmxcx9UClph-Q8lu3Hk=', 'nonce': 'ZmtcOPhm'}
```

生成后，将 `secret-key` 的值设置为环境变量。如果项目名称为 `sample`，开发环境为 `local`，那么应该设置环境变量 `SAMPLE_LOCAL_SECRET_KEY`：

```
export SAMPLE_LOCAL_SECRET_KEY='HK-VHH4C0ijLjOYBYrO7L2ACsmxcx9UClph-Q8lu3Hk='
```

阅读替换机制 了解 pyape 的环境变量替换机制。

7.2 配置

7.2.1 pyape.toml

pyape.toml 是 pyape 框架的主配置文件。它应该跟随项目源码一起被提交到 VCS。

一个框架需要大量的配置文件，pyape 也不例外。由于 TOML 格式的特点，我们可以把所有的配置文件的生成和替换在一个单独的文件中完成，这让我们的配置更容易理解。感谢 TOML。

关于 TOML 格式的介绍请阅读：https://toml.io/

7.2.2 替换机制

pyape 在生成配置文件的时候，会读取 `pyape.toml` 的所有内容，根据其中的配置做好值的替换，再写入到具体的配置文件中。

pyape 有两个主要的替换机制。

任何时候都可用的变量

`NAME/DEPLOY_DIR/WORK_DIR` 这三个变量在任何时候都可用。关于前两个，需要在根元素 中进行定义。

`WORK_DIR` 不需要定义就能使用，它指向 pyape 项目的工作文件夹。

从环境变量中获取值

为了安全，我们会把敏感的信息写入系统的环境变量中。pyape 可以从环境变量中提取致，在生成配置文件时进行替换。

请阅读根元素 中关于 `REPLACE_ENVIRON` 的介绍。

7.2.3 多开发环境支持

Web 程序开发的过程中，我们一般会有多套开发环境。例如在本地 `local` 环境做调试，在远程测试 `test` 服务器做测试，在正式服 `prod` 环境做部署。

pyape 的多开发环境支持，可以支持在 `pyape.toml` 中进行多套环境的配置。默认配置会被环境配置中的同名变量直接覆盖。这种机制减少了配置内容的数量，也方便信息共用。

环境配置以 *ENV.* *环境名*开头，后接希望被覆盖的配置名称。

例如，根元素下的 *DEPLOY_DIR* 配置，若希望在 `prod` 环境中使用不同的值，可以增加这样的配置：

```
[ENV.prod]
DEPLOY_DIR = '/srv/app/{{NAME}}_prod'
```

若希望在 `local` 环境中使用调试方式启动 Flask，则可以覆盖默认的 `FLASK_ENV`：

```
[ENV.local.'.env']
FLASK_ENV = 'development'
FLASK_RUN_PORT = 5001
```

在开发环境配置中提供的配置，将被 **合并**进入默认的配置。合并规则如下：

- 开发环境配置会 **覆盖**默认配置中的同名参数。
- 开发环境中的新配置，会 **增加**到默认配置中。
- 若希望在开发环境中 **删除**某个默认配置，可以将开发环境中的同名变量设置为空值。

备注： 由于 TOML 自身的规则 限制，TOML 配置中是没有 **空值**的概念的。若希望将某个值设置为空值，可以使用布尔值或者空对象的方法。

下面是几个关于开发环境替换的例子：

```
# 正式环境的 uwsgi 使用 4 进程启动
[ENV.prod.'uwsgi.ini']
processes = 4

# 正式环境的数据库使用 DEPLOY_DIR 来定位
[ENV.prod.'config.toml'.SQLALCHEMY]
URI = 'sqlite:///{{DEPLOY_DIR}}/pyape.sqlite'

# 正式环境的 gunicorn 使用 sock 绑定，并指定 pid 和 log 文件
[ENV.prod.'gunicorn.conf.py']
bind = 'unix:{{DEPLOY_DIR}}/gunicorn.sock'
pidfile = '{{DEPLOY_DIR}}/gunicorn.pid'
```

(续下页)

(接上页)

```
accesslog = '{{DEPLOY_DIR}}/logs/access.log'
errorlog = '{{DEPLOY_DIR}}/logs/error.log
```

7.2.4 根元素

PYE

远程服务器专用。定义 Python 运行时路径，可使用绝对路径。这个配置仅在部署远程服务器时有意义，指定的是远程服务器上的 Python 运行时。使用*pyape venv* 命令部署远程虚拟环境时，会使用这里定义的 Python 运行时。

NAME

项目名称。可以用做替换值。配置文件中所有包含 `{{NAME}}` 的参数都会被这里的值替换。

DEPLOY_DIR

远程服务器专用。设定部署在服务器上的文件夹。可以用做替换值。配置文件中所有包含 `{{DEPLOY_DIR}}` 的参数都会被这里的值替换。

RSYNC_EXCLUDE

远程服务器专用。这是一个列表，定义在使用*pyape deploy* 命令将本地代码同步到远程服务器时的排除文件。详情可参考 fabric-patchwork.transfers。

REPLACE_ENVIRON

这是一个列表。定义允许被替换的环境变量的名称。若配置文件中包含下面的名称，并使用 `{{}}` 包裹，则会被替换成环境变量中的值。

例如：

1. 项目 NAME 为 `pyape`，作为环境变量替换时，会被转换为全大写 `PYAPE`。
2. 环境变量中包含 `PYAPE_LOCAL_SECRET_KEY`。
3. 使用 `--env local` 生成配置文件时，将替换 `{{SECRET_KEY}}` 的值为环境变量中的 `PYAPE_LOCAL_SECRET_KEY`。

默认提供了四个环境变量替换：

- `{{ADMIN_NAME}}` 管理员帐号
- `{{ADMIN_PASSWORD}}` 管理员密码
- `{{SECRET_KEY}}` flask 框架加密使用
- `{{SQLALCHEMY_URI}}` 数据库地址和密码定义

亦可自行增加环境变量，保证配置文件中的变量名称相同即可。

7.2.5 [FABRIC]

pyape 使用 Fabric 作为部署工具。在部署时，会直接读取这个段落的配置作为 Fabric 调用的参数。

> **警告：** 强烈建议在本地 `~/.ssh/config` 中配置好 host 地址、端口和公钥。此处的 host 可以使用配置好的地址，避免真实的地址提交到版本库造成信息泄露。

host
: 远程服务器地址。

user
: 远程服务器登录用户。

7.2.6 ['.env']

`.env` 是一个配置文件，在使用*pyape config* 生成配置文件，或使用*pyape deploy* 进行远程部署时，会自动生成。

其内容为 FLASK 运行需要的配置。默认值为：

```
FLASK_APP = 'wsgi:{{NAME}}_app'
FLASK_ENV = 'production'
FLASK_RUN_PORT = 500
```

请参考 Flask 官方文档中的 dotenv 部分了解此处的配置。

pyape 会调用 flask.cli.load_env 将 `.env` 文件载入为环境变量。

7.2.7 ['gunicorn.conf.py']

`gunicorn.conf.py` 是 Gunicorn 的配置文件。

默认值为：

```
wsgi_app = 'wsgi:{{NAME}}_app'
proc_name = '{{NAME}}'
bind = '127.0.0.1:5001'
umask = 0
daemon = true
capture_output = true
```

配置中可用的参数，通过阅读 `pyape.tpl.gunicorn.conf.py.jinja2` 源码获取。

7.2.8 [‘uwsgi.ini’]

uwsgi.ini 是 uWSGI 的配置文件。

默认值为：

```
callable = 'wsgi:{{NAME}}_app'
processes = 2
threads = 1
venv = '%dvenv'
# 是否切换到后台，本地调试的时候可以设为 False，直接查看控制台输出
daemonize = true
# socket 和 http 参数二选一，如果同时选择，以 socket 参数为准
# 端口转发可能引发 nginx 499 问题（推测是端口转发 limit 没有打开）
# 改为使用 sock 文件 （同样需要打开 limit 限制）
socket = '%d%n.sock'
# http_socket = '127.0.0.1:5002'
# http = '127.0.0.1:5002'
# Stat Server
stats = '%d%nstats.sock
```

配置中可用的参数，通过阅读 pyape.tpl.uwsgi.ini.jinja2 源码获取。

7.2.9 [‘config.toml’]

config.toml 是 pyape 框架在作为 Web App 运行时使用的配置文件。数据库定义、endpoint 支持等均在此定义。

[‘config.toml’.FLASK]

定义 flask 框架使用的变量，默认值为：

```
SECRET_KEY = '{{SECRET_KEY}}'
```

定义在这里的变量会进入 flask.config。

[‘config.toml’.SQLALCHEMY]

定义数据库，默认值为：

```
# 单数据库地址配置，{{WORK_DIR}} 被替换为 pyape 运行文件夹的绝对路径
['config.toml'.SQLALCHEMY]
URI = 'sqlite:///{{WORK_DIR}}/pyape.sqlite
```

配置数据库的引擎参数：

```
# 单数据库配置数据库引擎参数
['config.toml'.SQLALCHEMY.ENGINE_OPTIONS]
pool_timeout = 10
pool_recycle = 360
```

URI 也可以作为多数据库存在：

```
['config.toml'.SQLALCHEMY.URI]
test1000 = 'mysql+pymysql://test:123456@127.0.0.1/test1000'
test2000 = 'mysql+pymysql://test:123456@127.0.0.1/test2000
```

配置 test1000 这个数据库的引擎参数：

```
['config.toml'.SQLALCHEMY.ENGINE_OPTIONS.test1000]
pool_timeout = 10
pool_recycle = 360
```

[‘config.toml’.REDIS]

REDIS 的配置与 SQLALCHEMY 拥有完全相同的规则。

单个 REDIS 数据库：

```
['config.toml'.REDIS]
URI = 'redis://localhost:6379/0'
```

多 REDIS 配置，与单个 REDIS 地址配置方式二选一：

```
['config.toml'.REDIS.URI]
# 对 REDIS 的使用遵循了最大利用率+最大灵活性原则，可能出现：
# 1. 单个 Regional 使用单个 REDIS 实例（少量情况）
# 2. 多个 Regional 使用同一个 REDIS 实例，分 DB 存储（多数情况）
# 3. 多个 Regional 使用同一个 REDIS 实例和同一个 DB（测试情况）
# 4. 单个 Regional 使用多个 REDIS 实例（暂未如此部署）
db0 = 'redis://localhost:6379/0'
db1 = 'redis://localhost:6379/1'
```

[‘config.toml’.PATH]

配置 Flask 对象创建时的三个路径参数，例如：

```
STATIC_URL_PATH = '/static'
TEMPLATE_FOLDER = 'client/dist/template'
STATIC_FOLDER = 'client/dist/static'
```

详情参见 flask.Flask 的参数。

[‘config.toml’.PATH.modules]

pyape 会自动根据这个配置下的名称导入项目文件夹 app 下的模块，每个模块作为一个 Blueprint 存在。

在下面的配置中，main 这个模块对应的 endpoint 为‘/’，user 这个模块对应的 endpoint 为‘/user’：

```
main = ''
user = '/user'
oauth = '/oauth
```

关于使用 pyape 开发 Web 项目的更多信息，请参见：开发 。

7.3 命令行

参考安装 安装最新版本，使用下面的命令行进行操作。

7.3.1 pyape

```
Usage: pyape [OPTIONS] COMMAND [ARGS]...

  管理和部署使用 pyape 构建的项目。

Options:
  --help  Show this message and exit.

Commands:
  config        「本地」生成配置文件。
  copy          「本地」复制 pyape 配置文件到当前项目中
  dar           「远程」在服务器上部署代码，然后执行重载。也就是 deploy and reload␣
↪的组合。
  deploy        「远程」部署项目到远程服务器。
  gen           「本地」生成器，生成一个 Flask 可用的 SECRET_KEY，一个 NONCE␣
```

(续下页)

(接上页)

```
↪字符串，和一个加盐密码。
  init          「本地」初始化 pyape 项目
  pipoutdated   「远程」打印所有的过期的 python package。
  putconf       「远程」生成并上传配置文件到远程服务器。
  reload        「远程」在服务器上重载项目进程。
  setup         「本地」创建 pyape␣
↪项目运行时必须的环境，例如数据库建立等。需要自行在项目根文件夹创建 setup.py。
  start         「远程」在服务器上启动项目进程。
  stop          「远程」在服务器上停止项目进程。
  supervisor    「本地」生成 Supervisor 需要的配置文件。
  top           「远程」展示 uwsgi 的运行情况。
  venv          「远程」部署远程服务器的虚拟环境。
```

7.3.2 pyape init

```
Usage: pyape init [OPTIONS]

  「本地」初始化 pyape 项目

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -F, --force          覆盖已存在的文件
  --help               Show this message and exit.
```

7.3.3 pyape config

```
Usage: pyape config [OPTIONS]
                          [[.env|uwsgi.ini|gunicorn.conf.py|config.toml]]...

  「本地」生成配置文件。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  -P, --env_postfix    在生成的配置文件名称末尾加上环境名称后缀。
  -F, --force          是否强制替换已存在的文件。
  --help               Show this message and exit.
```

7.3.4 pyape copy

```
Usage: pyape copy [OPTIONS] [NAME]...

  「本地」复制 pyape 配置文件到当前项目中

Options:
  -C, --cwd DIRECTORY  工作文件夹，也就是复制目标文件夹。
  -F, --force          覆盖已存在的文件
  -R, --rename         若目标文件存在则重命名
  --help               Show this message and exit.
```

7.3.5 pyape deploy

```
Usage: pyape deploy [OPTIONS]

  「远程」部署项目到远程服务器。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  --help               Show this message and exit.
```

7.3.6 pyape venv

```
Usage: pyape venv [OPTIONS] [UPGRADE]...

  「远程」部署远程服务器的虚拟环境。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  -I, --init           是否初始化虚拟环境。
  --help               Show this message and exit.
```

7.3.7 pyape start

```
Usage: pyape start [OPTIONS]

  「远程」在服务器上启动项目进程。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  --help               Show this message and exit.
```

7.3.8 pyape stop

```
Usage: pyape stop [OPTIONS]

  「远程」在服务器上停止项目进程。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  --help               Show this message and exit.
```

7.3.9 pyape reload

```
Usage: pyape reload [OPTIONS]

  「远程」在服务器上重载项目进程。

Options:
  -C, --cwd DIRECTORY  工作文件夹。
  -E, --env TEXT       输入支持的环境名称。  [required]
  --help               Show this message and exit.
```

7.3.10 pyape gen

```
Usage: pyape gen [OPTIONS] [NAME]...

  「本地」生成器，生成一个 Flask 可用的 SECRET_KEY，一个 NONCE␣
↪字符串，和一个加盐密码。

Options:
  -C, --cwd DIRECTORY  工作文件夹，也就是复制目标文件夹。
  --password TEXT      返回加盐之后的 PASSWORD，需要提供密码，同时在 NAME␣
↪参数中提供一个盐值。
  --nonce INTEGER      返回一个 nonce 字符串。  [default: 8]
  --help               Show this message and exit.
```

7.4 开发

7.4.1 一个最小化的 app

下面的工作完成后，项目文件夹结构将如下所示：

```
sample
├── app
│   ├── __init__.py
│   ├── main.py
│   └── user.py
├── config.toml
├── .gitignore
├── .env
├── pyape.toml
└── wsgi.py
```

使用*pyape init* 创建项目后，我们需要给项目增加内容使其可以工作。下面以创建一个在本地调试的 app 为例，描述项目搭建过程。

创建 app 文件夹，在其中加入 __init__.py 文件使其成为标准的 python 模块。

创建两个子模块 app/main.py 和 app/user.py。

main.py 的内容如下：

```
from flask import Blueprint

main = Blueprint('main', __name__)

```

(续下页)

(接上页)

```
@main.get('/')
def home():
    return 'HELLO WORLD
```

user.py 的内容如下：

```
from flask import Blueprint

user = Blueprint('user', __name__)

@user.get('/friend')
def friend():
    return 'HELLO MY FRIEND'
```

在*pyape.toml* 中创建 local 环境，进行如下配置：

```
[ENV.local.'.env']
FLASK_ENV = 'development'
FLASK_RUN_PORT = 5001

[ENV.local.'config.toml'.PATH]
STATIC_URL_PATH = '/static'

[ENV.local.'config.toml'.PATH.modules]
main = ''
user = '/user'
```

使用*pyape config* 生成配置文件，由于我们的目标是在本地环境调试运营，因此只需要生成 config.toml 和 .env：

```
pyape config --env local --force config.toml .env
```

> **警告：** 在生成 config.toml 之前，在环境变量中必须包含 SAMPLE_LOCAL_SECRET_KEY 这个环境变量。详情请阅读生成 *SECRET_KEY* 。

生成的 .env 文件内容为：

```
FLASK_APP = wsgi:sample_app
FLASK_ENV = development
FLASK_RUN_PORT = 5001
```

生成的 config.toml 文件内容为：

```
[FLASK]
SECRET_KEY = "HK-VHH4C0ijLjOYBYrO7L2ACsmxcx9UClph-Q8lu3Hk="

[SQLALCHEMY]
URI = "sqlite:///sample/sample.sqlite"

[PATH]
STATIC_URL_PATH = "/static"
STATIC_FOLDER = "dist"
TEMPLATE_FOLDER = "dist/template"

[SQLALCHEMY.ENGINE_OPTIONS]
pool_timeout = 10
pool_recycle = 3600

[PATH.modules]
main = ""
user = "/user"
```

参照*wsgi.py* 对已有文件进行修改。

执行 `flask run` 启动开发服务器。

测试页面访问：

```
$ curl http://127.0.0.1:5001/
HELLO WORLD
$ curl http://127.0.0.1:5001/user/friend
HELLO MY FRIEND%
```

完整的 sample 项目请访问 sample 。

7.4.2 wsgi.py

`wsgi.py` 是 Flask 项目的入口文件。执行*pyape init* 后，项目文件夹中会自动生成这个文件。我们需要修改这个文件，使其符合我们自己项目的需要。

最简单的 `wsgi.py` 内容如下：

```
import pyape.app
from pyape.flask_extend import PyapeFlask

pyape_app: PyapeFlask = pyape.app.init()
```

备注： 在*['.env']* 中要设置 `FLASK_APP = wsgi:pyape_app` 。

在使用 Gunicorn 部署时，要确保[*'gunicorn.conf.py'*] 中的 `wsgi_app = 'wsgi:pyape_app'` 。

在使用 uWSGI 部署时，要确保[*'uwsgi.ini'*] 中的 `callable = 'wsgi:pyape_app'`。

7.4.3 wsgi.py 加强版

为了方便理解，我们可以做得更多一些。

导入必要的模块：

```
from pathlib import Path
from functools import partial

import pyape.app
import pyape.config
from pyape.flask_extend import PyapeFlask, PyapeRespons
```

明确指定主配置文件：

```
work_dir = Path(__file__).parent.resolve()
gconfig = pyape.config.GlobalConfig(work_dir, 'config.toml')
```

测试期间，可以用继承 `PyapeRespone` 的方式来实现跨域：

```
class CustomResponse(PyapeResponse):
    @property
    def cors_config(self):
        return PyapeResponse.CORS_DEFAUL
```

创建一个 app 实例，使用支持跨域的 Response：

```
pyape_app: PyapeFlask = pyape.app.init(gconfig, create_app, cls_config={'ResponseClass
↪': CustomResponse})
```

加强版的完整内容 `wsgi.py`：

```
from pathlib import Path

import pyape.app
import pyape.config
from pyape.flask_extend import PyapeFlask, PyapeRespons

work_dir = Path(__file__).parent.resolve()
gconfig = pyape.config.GlobalConfig(work_dir, 'config.toml')

```

(续下页)

(接上页)

```
class CustomResponse(PyapeResponse):
    @property
    def cors_config(self):
        return PyapeResponse.CORS_DEFAUL

pyape_app: PyapeFlask = pyape.app.init(gconfig, None, cls_config={'ResponseClass':␣
↪CustomResponse})
```

7.4.4 wsgi.py 加加强版

基于加强版，可以做更多事。例如增加可以在 flask shell 环境中调用的上下文方法。以及对数据库进行初始化：

```
from pathlib import Path
from functools import partial

import pyape.app
import pyape.config
from pyape.flask_extend import PyapeFlask, PyapeRespons

work_dir = Path(__file__).parent.resolve()
gconfig = pyape.config.GlobalConfig(work_dir, 'config.toml')

class CustomResponse(PyapeResponse):
    @property
    def cors_config(self):
        return PyapeResponse.CORS_DEFAUL


def setup_app(pyape_app: PyapeFlask, **kwargs):
""" 初始化 app 项目，这个方法被嵌入 flask shell 上下文中执行，可以使用 kwargs 传递参数
"""
# 在这里可以进行数据库的初始化工作
# pyape_app._gdb.create_all()
return pyape_app


def create_app(pyape_app: PyapeFlask):
    """ 被 pyape.app.init 调用，用于处理 app 初始化
    """
    # 加入上下文处理器
    pyape_app.shell_context_processor(lambda: {
        'gdb': pyape_app._gdb,
        # 这里可以传递更多促使给 setup_app
```

(续下页)

(接上页)

```
        'setup': partial(setup_app, pyape_app),
    })
    pyape.app.logger.info(pyape_app.config)

pyape_app: PyapeFlask = pyape.app.init(gconfig, create_app, cls_config={'ResponseClass
↪': CustomResponse})
```

7.4.5 使用 SQLAlchemy 操作数据库

在使用*wsgi.py* 初始化框架的时候，数据库会自动创建。可以在 pyape.app.init_db 中找到创建代码：

```
def init_db(pyape_app: PyapeFlask):
""" 初始化 SQLAlchemy
"""
    sql_uri = pyape_app._gconf.getcfg('SQLALCHEMY', 'URI')
    if sql_uri is None:
        return
    global gdb
    if gdb is not None:
        raise ValueError('gdb 不能重复定义！')
    gdb = PyapeDB(app=pyape_app)
    pyape_app._gdb = gdb
```

备注： pyape.flask_extend.PyapeDB 是*pyape.db.SQLAlchemy* 的子类。

pyape 默认使用 SQLAlchemy 的 ORM 模式工作。让我们构建一个 Model 用于创建 Table。

创建子模块 app/model.py 用于 Table 定义。

```
# app/model.py
import time
from sqlalchemy import Column, INT, VARCHAR
from pyape.app import gdb

Model = gdb.Model()

class User(Model):
    __tablename__ = 'user'

    id = Column(INT, autoincrement=True, primary_key=True)
    name = Column(VARCHAR(100), nullable=False)
    createtime = Column(INT, nullable=False, default=lambda: int(time.time()))
```

在模块*app/user.py* 中增加两个方法，用于读写数据库。

备注： 涉及到 webargs 用法，请参考其 官方文档。

SQLAlchemy 语法基于最新的 SQLAlchemy 2.0，请阅读 SQLAlchemy 2.0 Tutorial。若你是 SQLAlchemy 1.x 用户，请阅读 Migrating to SQLAlchemy 2.0。

```
from flask import Blueprint, abort
from webargs.flaskparser import use_args
from webargs import fields

from pyape.app import gdb, logger
from app.model import User


@user.get('/get')
@use_args({'id': fields.Int(required=True)}, location='query')
def get_user(args):
    id = args['id']
    userobj = gdb.session().get(User, id)
    if userobj is None:
        logger.warning(f'user {id} is not found.')
        abort(404)
    return f'User id: {userobj.id}, name: {userobj.name}'


@user.post('/set')
@use_args({'id': fields.Int(required=True), 'name': fields.Str(required=True)},␣
↪location='form')
def set_user(args):
    userobj = User(**args)
    gdb.session().add(userobj)
    gdb.session().commit()
    return f'User id: {userobj.id}, name: {userobj.name}'
```

7.4.6 测试 Sample 项目的 local 环境（单数据库支持）

若 pyape 项目位于 `~/storage/pyape`：

```
# 进入虚拟环境
cd ~/storage/pyape
python3 -m venv venv
source venv/bin/active
```

(续下页)

(接上页)

```
# 安装当前环境下的 pyape
(venv) pip install -e .

# 创建本地配置文件，使用 local 环境
(venv) pyape config -FE local config.toml .env

# 运行单元测试
(venv) pytest tests/test_sample_env_local.py
```

7.4.7 多数据库支持范例

使用 SQLAlchemy 操作数据库 仅包含单数据库支持，得益于 SQLAlchemy 的良好设计以及*pyape.toml* 的多环境支持，我们可以非常容易让不同环境支持不同的数据库。

要理解多数据库支持的原理，请查看：*多数据库支持原理*。

在 `sample/pyape.toml` 中已经包含了多数据库范例的支持。让我们看看 `multidb` 环境的配置，我们使用*app.user_multidb* 模块替代*app.user*。

```
[ENV.multidb.'config.toml'.PATH.modules]
main = ''
user_multidb = '/user2'

[ENV.multidb.'config.toml'.SQLALCHEMY.URI]
# 多数据库配置，直接覆盖默认配置
db1 = 'sqlite:///{WORK_DIR}/sample_db1.sqlite'
db2 = 'sqlite:///{WORK_DIR}/sample_db2.sqlite'
```

创建子模块 `app/model_multidb.py` 用于 Table 定义。在这里，可以使用 *Model()* 方法，传递 `bind_key` 参数来获取对应不同数据库的 Model。下面的 `User1` 和 `User2` 两个 Table 分别位于不同的数据库。

```
# app/model_multidb.py
import time

from sqlalchemy import Column, INT, VARCHAR

from pyape.app import gdb

Model1 = gdb.Model('db1')
Model2 = gdb.Model('db2')
```

(续下页)

(接上页)

```
class User1(Model1):
    __tablename__ = 'user1'

    id = Column(INT, autoincrement=True, primary_key=True)
    name = Column(VARCHAR(100), nullable=False)
    createtime = Column(INT, nullable=False, default=lambda: int(time.time()))


class User2(Model2):
    __tablename__ = 'user2'

    id = Column(INT, autoincrement=True, primary_key=True)
    name = Column(VARCHAR(100), nullable=False)
    createtime = Column(INT, nullable=False, default=lambda: int(time.time()))
```

创建子模块 app/user_multidb.py 提供数据库访问方法。这里的范例简单传递 bind_key 参数用来指定写入不同的数据库。

```
from flask import Blueprint, abort
from webargs.flaskparser import use_args
from webargs import fields
from sqlalchemy import select

from pyape.app import gdb, logger
from app.model_multidb import User1, User2


user_multidb = Blueprint('user_multidb', __name__)


@user_multidb.get('/get')
@use_args({'id': fields.Int(required=True), 'bind_key': fields.Str(required=True)},␣
↪location='query')
def get_user_db1(args):
    id = args['id']
    bind_key = args['bind_key']
    User = User1 if bind_key == 'db1' else User2
    userobj = gdb.session().get(User, id)
    if userobj is None:
        logger.warning(f'user {id} is not found in {bind_key}.')
        abort(404)
    return f'User in {bind_key} id: {userobj.id}, name: {userobj.name}'
```

(续下页)

(接上页)

```
@user_multidb.post('/set')
@use_args({'id': fields.Int(required=True), 'name': fields.Str(required=True), 'bind_
↪key': fields.Str(required=True)}, location='form')
def set_user(args):
    bind_key = args['bind_key']
    User = User1 if bind_key == 'db1' else User2
    userobj = User(**args)
    gdb.session().add(userobj)
    gdb.session().commit()
    return f'User in {bind_key} id: {userobj.id}, name: {userobj.name}'
```

7.4.8 测试 Sample 项目的 multidb 环境（多数据库支持）

若 pyape 项目位于 ~/storage/pyape：

```
# 进入虚拟环境
cd ~/storage/pyape
python3 -m venv venv
source venv/bin/active

# 安装当前环境下的 pyape
(venv) pip install -e .

# 创建本地配置文件，使用 multidb 环境
(venv) pyape config -FE multidb config.toml .env

# 运行单元测试
(venv) pytest tests/test_sample_env_multidb.py
```

7.5 依赖

7.5.1 TOML

Tom’s Obvious, Minimal Language.

Tom 的（语义）明显、（配置）最小化的语言。

https://toml.io/

7.5.2 Flask

受欢迎且现代化的 Python 网络服务微框架。

https://flask.palletsprojects.com/

7.5.3 SQLAlchemy

Python 世界中最好的 ORM(Object Relational Mapper) 框架，简化你的 SQL 工作。

https://www.sqlalchemy.org/

7.5.4 Fabric

部署工具，让你基于 SSH 在远程计算机上执行 shell 命令。

https://www.fabfile.org

7.5.5 webargs & marshmallow

使用 webargs 解析 HTTP 请求中的参数，使用 marshmallow 串行化。

https://webargs.readthedocs.io/

https://marshmallow.readthedocs.io/

7.5.6 uWSGI

内容齐全，性能优秀，功能完善的全栈 WSGI HTTP 服务器。主要使用 c 实现。

https://uwsgi-docs.readthedocs.io/

7.5.7 gunicorn

轻量级的 WSGI HTTP 服务器。完全使用 Python 实现。

https://gunicorn.org/

7.6 架构

7.6.1 多数据库支持原理

在设计多数据库支持时，Flask-SQLAlchemy 的设计给了我很大启发，但我并不太喜欢这种设计。原因有 2：

1. SQLAlchemy 自身就建立了 binds 机制，已经支持不同的 mapper 使用不同的 bind，但 Flask-SQLAlchemy 并没有使用这套机制，这让我感到混乱。

2. Flask-SQLAlchemy 仅使用了一个 Model，通过在 Model 中设置 __bind_key__ 这个巧妙的方式来让不同的 Table 绑定到不同的数据库。而我认为直接使用多个 Model，在定义 Table 的时候通过 Model 来区分，更容易理解。

Flask-SQLAlchemy 对 Metadata 做了一些手脚，自动加入 __bind_key__ 属性。

```
class BindMetaMixin(type):
    def __init__(cls, name, bases, d):
        bind_key = (
            d.pop('__bind_key__', None)
            or getattr(cls, '__bind_key__', None)
        )

        super(BindMetaMixin, cls).__init__(name, bases, d)

        if bind_key is not None and getattr(cls, '__table__', None) is not None:
            cls.__table__.info['bind_key'] = bind_key
```

然后，通过重写 Session.get_bind 方法，让 Query 可以通过 __bind_key__ 来找到对应的 Engine：

```
def get_bind(self, mapper=None, clause=None):
    """Return the engine or connection for a given model or
    table, using the ``__bind_key__`` if it is set.
    """
    # mapper is None if someone tries to just get a connection
    if mapper is not None:
        try:
            # SA >= 1.3
            persist_selectable = mapper.persist_selectable
        except AttributeError:
            # SA < 1.3
            persist_selectable = mapper.mapped_table

        info = getattr(persist_selectable, 'info', {})
        bind_key = info.get('bind_key')
        if bind_key is not None:
```

(续下页)

(接上页)

```
            state = get_state(self.app)
            return state.db.get_engine(self.app, bind=bind_key)
    return SessionBase.get_bind(self, mapper, clause)
```

pyape 采用了另一种方法。下面截取*pyape.db.DBManager* 的一部分代码来说明：

```
class DBManager(object):

    def __build_binds(self) -> None:
        view = None
        if isinstance(self.URI, str):
            view = {None: self.URI}.items()
            self.default_bind_key = None
        else:
            view = self.URI.items()
            self.default_bind_key = list(self.URI.keys())[0]

        # 下面的三个引擎只需要创建一遍，在初始化的时间创建
        for name, uri in view:
            self.__add_bind(name, uri)

        self.__Session_Factory = sessionmaker(
            binds=self.__binds,
            autoflush=False,
            future=True
        )

    def __add_bind(self, bind_key: str, uri: str) -> bool:
        Model = self.set_Model(bind_key)
        if self.__binds.get(Model) is None:
            engine = self.__set_engine(bind_key, uri)
            self.__binds[Model] = engine
            return True
        return False

    def __set_engine(self, bind_key: str, uri: str) -> None:
        engine = create_engine(uri, future=True)
        # 保存 engine
        self.__engines[bind_key] = engine
        return engine

    def set_Model(self, bind_key: str=None):
        """ 设置并保存一个 Model。
```

(续下页)

(接上页)

```
        :param bind_key: 详见
            :ref:`pyape.db.DBManager.set_bind <pyape.db.DBManager>` 中的说明。
        """
        if self.__model_classes.get(bind_key):
            raise KeyError(bind_key)

        Model = declarative_base(name=bind_key or 'Model', metaclass=DefaultMeta)
        Model.bind_key = bind_key
        self.__model_classes[bind_key] = Model
        return Model
```

DBManager 在初始化时自动调用 `__build_binds` 方法，创建必须的 `__Session_Factory` 和 `Model`。Model 是根据*pyape.toml* 中的 `[SQLALCHEMY.URI]` 的值进行创建的。若 `URI` 值为 dict，代表使用多个数据库，dict 的 key 就是 `bind_key`，使用这个 key 就可以获取到不同的 Engine。若 `URI` 为 str，那么默认的 `bind_key` 就是 `None`，这也是一个合法的值。

在创建 `__Session_Factory` 时，使用 SQLAlchemy 提供的标准 `binds` 机制，将 mapper(即 Model) 和 Engine 对应起来。

要获取到不同数据库 Model，只需要使用 `get_Model(bind_key)` 即可。若要获取到对应的 Engine，也可以直接使用 `get_engine(bind_key)`。这简化了使用，也降低了理解成本。

具体案例请查看：多数据库支持范例。

7.7 参考