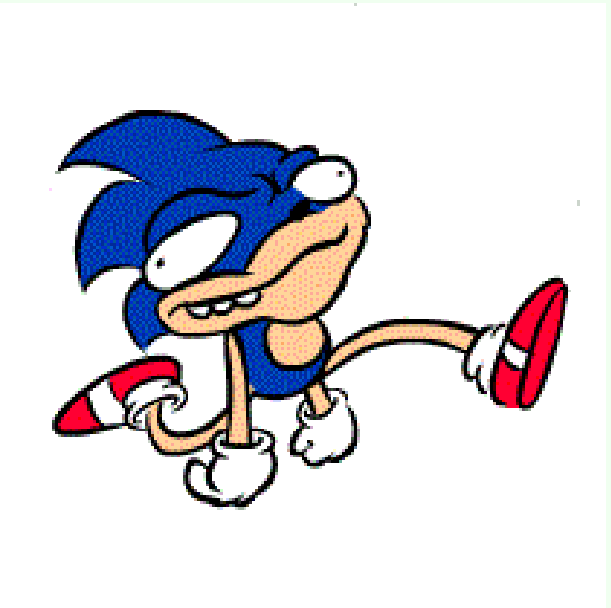
Introdução ao Sanic

O Flask Assíncrono

Danilo J. S. Bellini
@danilobellini



2018-08-25

Flask Conf @ Developer Hub – SP

Origem do nome

Sonic: The Hedgehog é um personagem bastante conhecido do mundo dos games... *Sanic* é o nome dos seus memes feitos às pressas, de qualquer jeito!

"Gotta go fast"

Referência à música de abertura do anime Sonic X na versão estadunidense, foi em 2008 associado a um desenho de fã presente no (agora extinto) Sonic Central.

"Sanic"

Meme criado em março de 2010 por 0nyxheart em um vídeo com o título "How 2 Draw Sanic Hegehog" desenhando o personagem no Microsoft Paint.

1º *commit* em 2016-05-26, *post* no Reddit em 2016-10-14

E se nativamente o Flask pudesse ser assíncrono e 6 (seis) vezes mais rápido?

Eis o Sanic, para quando você "gotta go fast" (precisa ir rápido)!

- Assíncrono — `async def`
- "Reluzente" — Python 3.5+
- Simples — rotas como no Flask
- Leve — não precisa de ferramentas especiais

Dados fornecidos pelo criador, no Reddit:

Framework	Requisições/segundo	Latência média
Sanic (Python 3.5 + uvloop)	30,601	3.23ms
Flask (gunicorn + meinheld)	4,988	20.08ms

Link: `https://www.reddit.com/r/Python/comments/57i301/`

Citações

- (2016-10-14) *Sanic was created because I love the freedom of Flask, but dislike deploying it and its lack of native async support.* (Channel Cat, criador do Sanic)
- (2018-03-26) *Sanic: python web server that's written to die fast* (Andrew Svetlov, criador do `aiohttp`)
- (2018-03-28) *I'm a fan of Sanic, keep up the good work.* (Phil Jones, autor do Quart)

Hello World!

Flask

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def root():
    return "Sync!"
```

Sanic

```python
from sanic import response, Sanic
app = Sanic(__name__)

@app.route("/")
async def root(request):
    return response.text("Async!")
```

Rodando os servidores (porta 1337):

```
# Nativamente
FLASK_APP=01_flask.py flask run -h 0.0.0.0 -p 1337
python -m sanic --host 0.0.0.0 --port 1337 01_sanic.app

# Sanic c/ 4 workers
python -m sanic --host 0.0.0.0 --port 1337 --workers 4 01_sanic.app

# Via Gunicorn (4 workers)
gunicorn -b 0.0.0.0:1337 -w 4 01_flask:app
gunicorn -b 0.0.0.0:1337 -w 4 01_sanic:app \
         --worker-class sanic.worker.GunicornWorker
```

Rodando como um script

Caso se queira personalizar a execução rodando como um script, isto é, rodando algo como `python -m 02_xxxxx_script` diretamente do Python, podemos escrever esse script Python como:

Flask

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def root():
    return "Sync!"

if __name__ == "__main__":
    app.run(
        host="127.0.0.1",
        port="1337",
        debug=True,
    )
```

Sanic

```python
from sanic import response, Sanic
app = Sanic(__name__)

@app.route("/")
async def root(request):
    return response.text("Async!")

if __name__ == "__main__":
    app.run(
        host="127.0.0.1",
        port="1337",
        debug=True,
        workers=4,
    )
```

JSON I/O (POST): Índice de Massa Corporal

```python
from flask import jsonify, request, Flask
app = Flask(__name__)

@app.route("/", methods=["POST"])
def root():
    data = request.get_json()
    imc = data["mass"] / data["height"] ** 2
    return jsonify({"imc": imc})
```

```python
from sanic import response, Sanic
app = Sanic(__name__)

@app.route("/", methods=["POST"])
async def root(request):
    data = request.json
    imc = data["mass"] / data["height"] ** 2
    return response.json({"imc": imc})
```

A sintaxe p/ selecionar os verbos do HTTP é a mesma: o argumento nominado `methods` com a lista de métodos (apenas GET, por padrão).

O primeiro uso do `request.json` avalia o corpo (*body*/*payload*) da requisição, devolvendo `None` caso não consiga ler o JSON. A diferença é que o Sanic ignora o `Content-Type` de entrada e utiliza uma `property`.

As respostas possuem o header `Content-Type: application/json`

JSON I/O (POST) — Resultados

```
$ # Flask
$ curl localhost:1337 -H 'Content-Type: application/json' \
                      -d '{"mass": 85, "height": 1.82}'
{"imc":25.661152034778407}
$ # Sanic (a rigor, o response.json ignora o header Content-Type)
$ !!
{"imc":25.6611520348}
```

Flask + Gunicorn

```
HTTP/1.1 200 OK
Server: gunicorn/19.9.0
Date: Sat, 25 Aug 2018 09:07:00 GMT
Connection: close
Content-Type: application/json
Content-Length: 27
```

Sanic

```
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: 5
Content-Length: 21
Content-Type: application/json
```

Conexões persistentes, baixa latência!

Status HTTP diferente de 200 (OK)

Entrada inválida? A resposta é "`HTTP/1.1 400 Bad Request`"!

```python
@app.route("/", methods=["POST"])                    # Flask
def root():
    data = request.get_json()
    try:
        imc = data["mass"] / data["height"] ** 2
    except (ValueError, KeyError, TypeError):
        return jsonify({"error": "bad_request"}), 400
    return jsonify({"imc": imc})
```

```python
@app.route("/", methods=["POST"])                    # Sanic
async def root(request):
    data = request.json
    try:
        imc = data["mass"] / data["height"] ** 2
    except (ValueError, KeyError, TypeError):
        return response.json({"error": "bad_request"},
                             status=400)
    return response.json({"imc": imc})
```

As funções do módulo `response` aceitam dois argumentos nominados:

- `status` — Inteiro com o status HTTP
- `headers` — Dicionário com modificações a serem feitas no cabeçalho HTTP

Duas rotas no mesmo *handler* c/ Flask

Uma rota GET sem *query string* (i.e., a entrada está no próprio caminho) e uma POST (JSON) tratadas pela mesma função.

```
from flask import jsonify, request, Flask
app = Flask(__name__)

@app.route("/imc", methods=["POST"])
@app.route("/imc/<float:mass>/<float:height>")
def imc(mass=None, height=None):
    try:
        if request.method == "POST":
            data = request.get_json()
            mass = data["mass"]
            height = data["height"]
        imc = mass / height ** 2
    except (ValueError, KeyError, TypeError):
        return jsonify({"error": "bad_request"}), 400
    return jsonify({"imc": imc})
```

No Flask, blocos <TIPO:NOME> são lidos, convertidos e passados como argumentos nominados.

Duas rotas no mesmo *handler* c/ Sanic

No Sanic, o tipo `float` apenas possui outro nome: `number`.

```
from sanic import response, Sanic
app = Sanic(__name__)

@app.route("/imc", methods=["POST"])
@app.route("/imc/<mass:number>/<height:number>")
async def imc(request, mass=None, height=None):
    try:
        if request.method == "POST":
            mass = request.json["mass"]
            height = request.json["height"]
        imc = mass / height ** 2
    except (ValueError, KeyError, TypeError):
        return response.json({"error": "bad_request"},
                             status=400)
    return response.json({"imc": imc})
```

No Sanic, blocos <NOME:TIPO> (o inverso do Flask) são lidos, convertidos e passados como argumentos nominados.

Tipos permitidos nas rotas

- Flask — `string`, `int`, `float`, `path` e `uuid`
- Sanic — `int`, `number` e *regexes*

No Flask o uso de expressões regulares (*regexes*) poderia ser feito processando o conteúdo de uma variável definida no caminho com o tipo path, ou então customizando um conversor parametrizado e inserindo-o em `app.url_map.converters`.

```
from sanic import response, Sanic
app = Sanic(__name__)

@app.route("/imc/<mass:\d+(?:.\d+)?>/<height:\d+(?:.\d+)?>")
async def imc(request, mass, height):
    try:
        imc = float(mass) / float(height) ** 2
    except (ValueError, TypeError):
        return response.json({"error": "bad_request"},
                             status=400)
    return response.json({"imc": imc})
```

Middleware

```python
from datetime import datetime, timezone
from email.utils import formatdate

from sanic import exceptions, response, Sanic
app = Sanic(__name__)

@app.middleware("request")
async def before_handling(request):
    request["start"] = datetime.now(tz=timezone.utc)
    if request.headers.get("Authorization", "") != "TEST":
        exceptions.abort(401)

@app.middleware("response")
async def after_handling(request, response):
    end = datetime.now(tz=timezone.utc)
    response.headers.update({
        "Duration": str(end - request["start"]),
        "Path": request.path,
        "Date": formatdate(end.timestamp()),
    })

@app.route("/")
@app.route("/another/path")
async def root(request):
    return response.json({"status": "authorized"})
```

Similar ao `before_request` e ao `after_request` do Flask.

Há várias coisas acontecendo:

- *Timestamps* e tempo de processamento
- Inserção de 3 *headers*
- Autorização (*bearer token*)
- Acesso ao caminho da requisição

Middleware — *Headers* dos resultados

```
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: 5
Duration: 0:00:00.000025
Path: /
Date: Sat, 25 Aug 2018 12:46:59 -0000
Content-Length: 23
Content-Type: application/json
```

Os caminhos testados foram:

- /
  (com *Authorization: TEST*)
- /unknown
  (com *Authorization: TEST*)
- /another/path
  (apenas *headers* do cURL)

```
HTTP/1.1 404 Not Found
Connection: keep-alive
Keep-Alive: 5
Duration: 0:00:00.000533
Path: /unknown
Date: Sat, 25 Aug 2018 12:51:52 -0000
Content-Length: 39
Content-Type: text/plain; charset=utf-8
```

```
HTTP/1.1 401 Unauthorized
Connection: keep-alive
Keep-Alive: 5
Duration: 0:00:00.000549
Path: /another/path
Date: Sat, 25 Aug 2018 13:19:31 -0000
Content-Length: 19
Content-Type: text/plain; charset=utf-8
```

Rodando um banco de dados PostgreSQL

```
docker run --rm -d \
  -p 5432:5432 \
  -e POSTGRES_USER=flask \
  -e POSTGRES_PASSWORD=conf \
  -e POSTGRES_DB=sanic \
  --name pgdb \
  postgres

export PGSQL_URL=postgres://flask:conf@localhost:5432/sanic
```

O comando `docker run` acima roda o PostgreSQL localmente, e o segundo comando define a variável de ambiente PGSQL_URL com o DSN (*data source name*), permitindo que a conexão com o banco seja realizada por meio dessa variável ao invés de um valor *hardcoded* no código Python.

Listeners — *Timestamp* do banco de dados... *Await*!!!

```python
import os
from asyncpgsa import pg
from sanic import  response, Sanic
app = Sanic(__name__)

@app.listener("before_server_start")
async def setup_db(app, loop):
    await pg.init(os.environ["PGSQL_URL"])

@app.route("/")
async def tstamper(request):
    ip = request.remote_addr or request.ip
    query = "SELECT timezone('UTC', CURRENT_TIMESTAMP)"
    now = await pg.fetchval(query)
    return response.json({
        str(ip): now.isoformat(timespec="microseconds")
    })
```

Há 4 listeners possíveis, e todos possuem os mesmos parâmetros de entrada:

- `before_server_start`
- `after_server_start`
- `before_server_stop`
- `after_server_stop`

Finalmente assíncrono!

```
$ curl localhost:1337
{"127.0.0.1":"2018-08-25T14:40:57.873015"}
```

Cuidados com o código síncrono

```python
import time
from sanic import response, Sanic
app = Sanic(__name__)

def slow_func(x):
    time.sleep(1)
    return x[::-1]

@app.route("/")
async def root(request):
    result = slow_func(request.args["input"][0])
    return response.json({"reversed": result})
```

```
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}

real  0m10.044s
user  0m0.063s
sys  0m0.036s
```

Com 1 worker, 5 processos realizando requisições simultâneas, e um total de 10 requisições. O comando abaixo foi utilizado para realizar as execuções:

```
command='curl -s "localhost:1337?input=flaskconf"'
time sh -c "seq 10 | sed 'c $command' | xargs -d'\n' -n1 -P5 sh -c | jq -c"
```

asyncio.sleep

```python
import asyncio
from sanic import response, Sanic
app = Sanic(__name__)

async def slow_func(x):
    await asyncio.sleep(1)
    return x[::-1]

@app.route("/")
async def root(request):
    result = await slow_func(request.args["input"][0])
    return response.json({"reversed": result})
```

```
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}

real  0m2.039s
user  0m0.068s
sys  0m0.035s
```

Que tal 5 vezes mais rápido? Não use o `time.sleep` em código assíncrono, pois ele é bloqueante!

concurrent.futures.ThreadPoolExecutor

```python
import time
from sanic import response, Sanic
app = Sanic(__name__)

def slow_func(x):
    time.sleep(1)
    return x[::-1]

@app.route("/")
async def root(request):
    def slow_runner():
        return slow_func(request.args["input"][0])
    future = app.loop.run_in_executor(None, slow_runner)
    result = await future
    return response.json({"reversed": result})
```

```
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}
{"reversed":"fnocksalf"}

real  0m2.041s
user  0m0.053s
sys  0m0.045s
```

Alternativa híbrida! Por que não usar um pool de threads para rodar o código bloqueante enquanto o restante continua assíncrono?

O `None` desse excerto denota o `ThreadPoolExecutor`.

O `app.loop` é o *loop* do uvloop, que também pode ser obtido através do `asyncio.get_event_loop()`.

Blueprints — Rotas (*request handlers*)

Se trocarmos a classe `Sanic` pela classe `Blueprint`, a princípio nada muda...

Módulo `imc10.py`

```python
from sanic import response, Blueprint
bp = Blueprint(__name__)

@bp.route("/imc")
async def imc(request):
    try:
        mass = request.args["mass"][0]
        height = request.args["height"][0]
        imc = float(mass) / float(height) ** 2
    except (ValueError, KeyError, TypeError):
        return response.json({"error": "bad_request"},
                             status=400)
    return response.json({"imc": imc})
```

Blueprints podem fazer parte da aplicação, trata-se de uma forma de "picotar" em diferentes módulos sem o problema de importação cíclica, dado que o objeto `app` é tipicamente criado no contexto do módulo principal.

Blueprints — Middleware

A rigor, mesmo *middlewares* e *listeners* podem fazer parte do blueprint, porém eles serão sempre globais (isto é, o *middleware* não é acionado apenas com as rotas do módulo com o blueprint em que ele se encontra, mas com todas as rotas tratadas pela aplicação).

Módulo `middle10.py`

```python
from datetime import datetime, timezone
from email.utils import formatdate
from sanic import response, Blueprint
bp = Blueprint(__name__)

@bp.middleware("request")
async def before_handling(request):
    request["start"] = datetime.now(tz=timezone.utc)

@bp.middleware("response")
async def after_handling(request, response):
    end = datetime.now(tz=timezone.utc)
    response.headers.update({
        "Duration": str(end - request["start"]),
        "Date": formatdate(end.timestamp()),
    })
```

Arquivos estáticos e *blueprints*

Bora unificar, aproveitando para fornecer arquivos estáticos!

```
from sanic import Sanic
import imc10, middle10

app = Sanic(__name__, configure_logging=False)
app.blueprint(imc10.bp)
app.blueprint(middle10.bp)
app.static("img", "sanic.png") # route, file/dir name
app.static("gotta_go_fast.png", "gotta_go_fast.png")
```

Há ainda diversos outros recursos (e.g. prefixo de caminho, uso de nomes e do `url_for` para obtenção dos caminhos, etc.).

Fallback de exceções

Nos exemplos em que haviam exceções não tratadas, o servidor devolve um erro interno do servidor (status HTTP 500) com uma mensagem em HTML. Algo similar ocorreu com os erros 404 (não encontrado), 405 (método não permitido) e 401 (não autorizado) nos exemplos anteriores. Essas exceções poderiam ser, todas, filtradas para terem suas mensagens substituídas por mensagens em JSON. Para isso, basta usar o decorador `exception` (na aplicação ou em um *blueprint*). Por exemplo:

```
from sanic import Blueprint, exceptions, response
bp = Blueprint(__name__)

@bp.exception(Exception)
def handle_error(request, exception):
    status = getattr(exception, "status_code", 500)
    reason = exceptions.STATUS_CODES.get(exception.status_code,
                                         b"unknown_error")
    snake_reason = b"_".join(reason.lower().split())
    return response.json({"error": snake_reason}, status=status)
```

Respostas sempre JSON!

Aplicando o novo *blueprint*,

```
from sanic import Sanic
import exc10, imc10, middle10

app = Sanic(__name__, configure_logging=False)
app.blueprint(exc10.bp)
app.blueprint(imc10.bp)
app.blueprint(middle10.bp)
app.static("img", "sanic.png") # route, file/dir name
app.static("gotta_go_fast.png", "gotta_go_fast.png")
```

As requisições para rotas inválidas agora resultam em JSON válido como resposta, acompanhando o cabeçalho HTTP:

```
HTTP/1.1 405 Method Not Allowed
[...]
{"error":"method_not_allowed"}

HTTP/1.1 404 Not Found
[...]
{"error":"not_found"}
```

E muitos outros recursos!

- Websockets (suporte nativo);
- Testes automatizados com `request, response = app.test_client.get("/rota")` (ou com o *plugin* `pytest-sanic`)
- *Class-Based Views*, com `sanic.views.HTTPMethodView` e `app.add_route`;
- Configuração (`app.config`);
- SSL;
- Compartilhamento de loop com outros recursos assíncronos;
- `Exemplos` na wiki do Sanic (GitHub) com `aiohttp` (cliente HTTP assíncrono), Jinja2, aiopeewee, aiopg, Motor, etc.;
- `Plugins` (ou extensões) listadas na wiki do Sanic.

Futuro do Sanic

A versão mais recente do Sanic no PyPI é a `0.7.0`, embora a *tag* `0.8.0` já esteja no repositório.

O status ainda é *Pre-Alpha*, mas o desenvolvimento está bastante rápido!

Uma seleção de issues em andamento:

- `issue1176` — O modo de *streaming* está incompleto;
- `issue1265` — Existe a intenção de prover compatibilidade com o ASGI (*Asynchronous Server Gateway Interface*);

Será que saberemos quem é "Channel Cat", o autor do Sanic?

Referências

- https://youtu.be/VTHsOSGJHN0 — Abertura de Sonic X
- https://youtu.be/3f_cAy8ugvQ — História do meme
- https://youtu.be/of9gIoQpPmA — Reupload do 0nyxheart
- https://knowyourmeme.com/memes/sanic-hegehog
- https://knowyourmeme.com/memes/gotta-go-fast
- https://magic.io/blog/uvloop-blazing-fast-python-networking/
- https://github.com/channelcat/sanic
- https://sanic.readthedocs.io
- https://www.reddit.com/r/Python/comments/57i301/

FIM!