目　录

致谢

当前文档 《Sanic使用教程(Sanic For Pythoneer)》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2006-01-02。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/Sanic-For-Pythoneer

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Introduction

第一部分：技巧

1.初使用

快速开始

在安装Sanic之前，让我们一起来看看Python在支持异步的过程中，都经历了哪些比较重大的更新。

首先是Python3.4版本引入了 `asyncio` ，这让Python有了支持异步IO的标准库，而后3.5版本又提供了两个新的关键字 `async/await` ，目的是为了更好地标识异步IO，让异步编程看起来更加友好，最后3.6版本更进一步，推出了稳定版的 `asyncio` ，从这一系列的更新可以看出，Python社区正迈着坚定且稳重的步伐向异步编程靠近。

安装

Sanic是一个支持 `async/await` 语法的异步无阻塞框架，这意味着我们可以依靠其处理异步请求的新特性来提升服务性能，如果你有 `Flask` 框架的使用经验，那么你可以迅速地使用 `Sanic` 来构建出心

中想要的应用，并且性能会提升不少，我将同一服务分别用Flask和Sanic编写，再将压测的结果进行对比，发现Sanic编写的服务大概是 `Falsk` 的1.5倍。

仅仅是Sanic的异步特性就让它的速度得到这么大的提升么？是的，但这个答案并不标准，更为关键的是Sanic使用了 `uvloop` 作为 `asyncio` 的事件循环， `uvloop` 由Cython编写，它的出现让 `asyncio` 更快，快到什么程度？这篇文章中有介绍，其中提出速度至少比 nodejs、gevent 和其他Python异步框架要快两倍，并且性能接近于用Go编写的程序，顺便一提，Sanic的作者就是受这篇文章影响，这才有了Sanic。

怎么样？有没有激起你学习Sanic的兴趣，如果有，就让我们一起开始学习吧，在开始之前，你只需要有一台安装了Python的电脑即可。

> 说明：由于*Windows*下暂不支持安装*uvloop*，故在此建议使用*Mac*或*Linux*

虚拟环境

程序世界一部分是对应着现实的，在生活中，我们会在不同的环境完成不同的任务，比如在厨房做饭、卧室休息，分工极其明确。

其实用Python编写应用服务也是如此，它们同样希望应用服务与开发环境是一对一的关系，这样做的好处在于，每个独立的环境都可以简洁高效地管理自身对应服务所依赖的第三方库，如若不然，各个服务都安排在同一环境，这样不仅会造成管理上的麻烦，还会使第三方库之间产生冲突。

通过上面的叙述，我们是不是可以得出这样一个核心观点：应该在不同的环境下做不同的事 ，以此类推，写项目的时候，我们也需要为每个不同的项目构建一个无干扰的的环境，发散思维，总结一下：

> 不同的项目，需要为其构建不同的虚拟环境，以免互相干扰

构建虚拟环境的工具很多，如下：

- virtualenv
- pyenv
- anaconda

.....

以上三个工具都可以快速地帮助我们构建当前需要的Python环境，如果你之前没有使用过，可直接点开链接进行下载，如果你正在使用其它的环境管理工具，也不要紧，因为不论你使用哪一种方式，我们最终目的都是针对一个新项目构建一个新的环境。

安装配置好之后，简单看看官方提供的使用方法，就可以开始了，比如我本机使用的是 `anaconda` ，安装完成后可以很方便地创建一个虚拟环境，比如这里使用Python3.6来作为本书项目的默认环境：

```
# 新建一个python3.6环境
conda create --name python36 python=3.6
# 安装好之后 输入下面命令进入名为python36的环境
source activate python36
```

若安装速度比较慢，可以考虑换国内源，比如 国内镜像 ，至于为什么选择python3.6作为默认环境，一是因为Sanic只支持Python3.5+，二则是我们构建的项目最终是要在生产环境下运行的，所以建议最好安装Python3.6下稳定版本的 `asyncio` 。

安装Sanic

Python安装第三方模块都是利用 `pip` 工具进行安装，这里也不例外，首先进入上一步我们新建的 `python3.6` 虚拟环境，然后安装：

```
# 安装Sanic, 请先使用 source activate python36 进入虚拟环境
```

```
pip install sanic
# 如果不想使用uvloop和ujson 可以这样安装
SANIC_NO_UVLOOP=true SANIC_NO_UJSON=true pip install sanic
```

通过上面的命令，你就可以在 `python3.6` 虚拟环境中安装Sanic以及其依赖的第三方库了，若想查看Sanic是否已经正确安装，可以进入终端下对应的虚拟环境，启动Python解释器，导入Sanic库：

```
# 启动Python解释器
python
>>> import sanic
>>>
```

如果没有出现错误，就说明你已经正确地安装了Sanic，请继续阅读下一节，了解下如何利用Sanic来构建一个Web项目吧。

踏出第一步

我们将正式使用Sanic来构建一个web项目，让我们踏出第一步，利用Sanic来编写一个返回 `Hello World!` 字符串的服务程序。

新建一个文件，名为 `run.py` ：

```
#!/usr/bin/env python
from sanic import Sanic
from sanic.response import text

app = Sanic()


@app.route("/")
async def test(request):
    return text('Hello World!')


```

```
13. if __name__ == "__main__":
14.     app.run(host="0.0.0.0", port=8000)
```

Sanic的目标是让编写服务更加简单易用，请看上面仅用不到10行的代码，就编写好了一个简单的Web服务，运行此文件，在浏览器输入 `http://0.0.0.0:8000` ，出现的字符会让你回想起当年学c的恐惧 ^_^。

如果你是第一次使用Sanic，上面的代码可能会让你产生一些困扰，不用担心，接下来，我们将一起用Sanic编写一个简单的资讯阅读的web服务，在这过程中，你将逐渐地了解到Sanic的一些基本用法，如路由的构建、接受请求数据以及返回响应的内容等。

本次示例的源代码全部在github上，见 examples/demo01/news.py。

编写一个资讯阅读项目

在开始编写之前，第一步最好写一下需求，哪怕是个简单不过的玩具项目也不能略过这个步骤，比如现在编写的资讯阅读项目，需求就一个，在页面中展示一些资讯新闻。

既然是展示资讯新闻，那么解决数据来源的问题最为重要，对于这个问题你也不用担心，因为在本次示例的源码中我编写了一个名为 `get_news()` 的函数专门用来返回资讯新闻数据，简化代码如下：

```
1. async def get_news(size=10):
2.     """
3.     Sanic是一个异步框架，为了更好的发挥它的性能，有些操作最好也要用异步的
4.     比如这里发起请求就必须要用异步请求框架aiohttp
5.     所以使用本服务的时候请先执行: pip install aiohttp
6.     数据使用的是readhub网站的api接口
7.     为了使这个数据获取函数正常运行，我会保持更新，所以具体代码：
   examples/demo01/news.py
```

```
    """
    async with aiohttp.ClientSession() as client:
        async with client.get(readhub_api, params=params,
headers=headers) as response:
            assert response.status == 200
            text = await response.json()
        return text
```

这样各位就可以只专注于Sanic的代码实现，而不必考虑其他问题，我会一直维护这个数据获取函数，以保证数据正常输出，各位请放心使用。

构建路由

数据的问题解决之后，我们可以开始着手于需求的实现了，根据前面的描述，此时的需求是当客户端（Web浏览器）访问 `http://0.0.0.0:8000/` 的时候，浏览器会立马展示服务端响应返回的10条资讯新闻（假设内容由index()函数返回），若浏览器访问的是 `http://0.0.0.0:8000/2` ，此时返回的就是第二页的10条资讯新闻，以此类推……

当Sanic程序实例接收到一个请求，比如前面提到的 `http://0.0.0.0:8000/` ，它是怎么知道这个URL可以对应到 `index()` 函数呢？

Sanic有一个机制来保存URL和函数（一般称之为视图函数）之间的映射关系，就像 `dict` 中 `key` 和 `value` ，这样当服务端接收到请求 `http://0.0.0.0:8000/` ，就会立马知道，接下来需要调用 `index()` 函数了，我们将其称之为路由。

Sanic中可以用 `app.route` 修饰器来定义路由，当Sanic服务启动的时候， `app.route` 就会将其中传入的参数与装饰的函数自动注册好，比如下面这段代码：

```
@app.route("/")
async def index(request):
    """当服务端接收到客户端的/请求时，就会调用此函数"""
    return text('Hello World!')
```

此时请求 `http://0.0.0.0:8000/` 就会返回 `Hello World!` ，很显然，这不是我们想要的需求，我们的需求是展示10条资讯新闻，数据怎么来？你只需要调用 `get_news()` 函数，就会获取到你想要的资讯数据：

```
@app.route("/")
async def index(request):
    # html页面模板
    html_tem = """
    <div style="width: 80%; margin-left: 10%">
        <p><a href="{href}" target="_blank">{title}</a></p>
        <p>{summary}</p>
         <p>{updated_at}</p>
    </div>
    """
    html_list = []
    # 获取数据
    all_news = await get_news()
    # 生成在浏览器展示的html页面
    for each_news in all_news:
        html_list.append(html_tem.format(
            href=each_news.get('news_info', [{}])[0].get('url',
'#'),
            title=each_news.get('title'),
            summary=each_news.get('summary'),
            updated_at=each_news.get('updated_at'),
        ))

    return html('<hr>'.join(html_list))
```

运行此服务：

```
python run news.py
```

此时，访问 `http://0.0.0.0:8000/` ，你就会获得Sanic服务程序返回的资讯新闻，如下图，可以看到返回服务端提供的最新资讯：

页面成功地呈现出我们想要的结果，实在是令人兴奋，等等，不能高兴太早，我们还有一个需求，要根据浏览器输入的页数来展示内容，如： `http://0.0.0.0:8000/2` ，思考一下，应该怎样优雅地完成这个需求，或许你会想，再构建一对URL与视图函数的映射关系，像下面这样：

```
@app.route("/2")
async def page_2(request):
```

不得不说，这是一个糟糕的解决方案，这样没法解决接下来的第3页、第4页、甚至第n页（虽然目前这个服务程序只展示到第2页），最佳实践应该是把页数当做变量来获取，Sanic的路由机制自然提供了获取动态请求参数的功能，如下：

```
@app.route("/<page:int>")
@app.route("/")
async def index(request, page=1):
    """
    支持/请求与/page请求方式
    具体的代码逻辑也会有一点改变，可参考：examples/demo01/news.py
    """
```

再次运行此服务：

```
python run news.py
```

不论是请求 `http://0.0.0.0:8000/` 或者 `http://0.0.0.0:8000/2` ，都是我们想要的结果。

请求数据

细心的你可能会发现，每次编写一个视图函数的时候，总是有一个 `request` 参数：

```
async def index(request, page=1):
```

为什么必须定义这个参数，它从哪来？它有什么作用，下面我将一一为你解答。

如果你在客户端请求 `http://0.0.0.0:8000/` 的时候，顺手在视图函数里面打印下参数 `request` ，会有如下输出：

```
<Request: GET />
```

看终端的输出可以了解到 `request` 参数实际上是一个名为 `Request` 的实例对象，每当服务端接收到一个请求，Sanic的 `handle_request` 函数必定会接收一个 `Request` 实例对象，这个实例对象包含了一系列请求信

息。

前面说到，每个URL对应一个视图函数，而Sanic的 `handle_request` 接下来会将接收的 `Request` 实例对象作为参数传给URL对应的视图函数，也就是上面 `index` 的 `request` 参数，这样一来，就必须定义 `request` 来接收 `Request` 实例对象，其中包含的一些请求信息对视图函数来说非常重要，目前 `Request` 对象提供了以下属性：

- json
- token
- form
- files
- args
- raw_args
- cookies
- ip
- port
- socket
- remote_addr
- path
- url

上面只是列出了一部分属性，如果你想了解更多，可查看request.py源码文件了解。

为了可以实际使用下 `request` ，我们可以再加一个需求，比如增加一个 `GET` 请求的接口 `http://0.0.0.0:8000/json` ，如果请求不设置参数 `nums` 的值，则默认返回一条资讯新闻，如果设置了 `nums` 参数，则该接口返回的新闻数量由参数值决定，参数最大值为10：

```
1.  @app.route('/json')
```

```
async def index_json(request):
    """
    默认返回一条资讯，最多十条
    """
    nums = request.args.get('nums', 1)
    # 获取数据
    all_news = await get_news()
    try:
        return json(random.sample(all_news, int(nums)))
    except ValueError:
        return json(all_news)
```

运行此服务：

```
python run news.py
```

此时视图函数 `index_json` 就可以根据接受的参数 `nums` 来返回对应数量的新闻，访问 `http://0.0.0.0:8000/json?nums=2` ，效果如下：

响应

不论哪个Web框架，都是需要构建响应对象的，Sanic自然也不例外，

它用的是 `sanic.response` 来构建响应对象，像上面的代码中可以看到：

```
1.  from sanic.response import html, json
```

这表示我们目前构建的资讯阅读服务，分别返回了 `body` 格式为 `html` 以及 `json` 的响应对象，除了这两种格式，Sanic还提供了下面几种格式：

- json
- text
- raw
- html
- file
- file_stream
- stream

更多属性请看response.py，我们可以根据实际需求来构建响应对象，最后再返回给客户端。

继续深入

不要以为现在编写的资讯服务已经很完善了，其实还有许多问题需要我们解决，比如访问 `http://0.0.0.0:8000/html` 这个URL会返回：

```
1.  Error: Requested URL /html not found
```

服务程序为什么会抛出这个错误？因为程序中并路由没有注册 `html`，并且没有进行错误捕捉（比如此时的404），解决这个问题也很方便，比如把这个错误全部跳转到首页，代码如下：

```
1.  @app.exception(NotFound)
2.  def ignore_404s(request, exception):
```

```
3.      return redirect('/')
```

此时访问一些没有注册于路由的URL，比如此时的 `http://0.0.0.0:8000/html` 都会自动跳转到 `http://0.0.0.0:8000/` 。

现在，我们已经用Sanic编写了一个简单的资讯阅读服务，在编写的过程中使用了路由、数据请求、处理以及响应对象，这些基础知识足够你编写一些基本的服务，但这还远远不够，比如模板引、引入静态文件等，这些都等着我们在实践中继续深入了解。

总结

本章介绍了Sanic的安装以及基本的使用，目标是希望诸位可以迅速的了解并掌握Sanic的基本使用方法，并为阅读接下来的章节打一下基础。

文档以及代码：

- `Sanic` github地址：https://github.com/channelcat/sanic
- 官方教程：http://sanic.readthedocs.io/en/latest/
- demo地址：demo01

2.配置

配置

对于一个项目来说，配置是一个很严肃的问题，比如说：在开发环境和生产环境中，配置是不同的，那么一个项目该如何自由地在不同的配置环境中进行切换呢，思考下，然后带着答案或者疑问往下阅读。

单一配置

撸起袖子，开始吧，新建文件夹 `demo2` ，内部建立这样的文件结构：

```
demo02
├── config
│   ├── __init__.py
│   └── config.py
└── run.py
```

其中 `run.py` 内容如下：

```
#!/usr/bin/env python
from sanic import Sanic
from sanic.response import text

app = Sanic()


```

```
@app.route("/")
async def test(request):
    return text('Hello World!')


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

代码示例中开启了 `debug` 模式，假设我们需要通过 `config.py` 配置文件来实现控制服务的 `debug` 模式开启与否，那该怎么实现呢。

在 `config.py` 中添加一行: `DEBUG=True` ，然后 `run.py` 内容改为:

```
#!/usr/bin/env python
from sanic import Sanic
from sanic.response import text
from config import DEBUG

app = Sanic()


@app.route("/")
async def test(request):
    return text('Hello World!')


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=DEBUG)
```

表面上看，功能确实实现了，但这实际上却不是很好的做法，若部署在生产环境中，难道还要特地再将 `debug` 改为 `False` 么，这显然很浪费时间，如果需要改变的参数有很多，那就很难维护了。

多配置

那么，正确的做法应该是怎么样的呢？

我们应当依据不同的环境来编写各自对应的环境，举个例子，比如生产环境就对应 `pro_config` ，开发环境就对应 `dev_config.py` 等等

具体该怎么实施？首先在文件夹 `demo2` ，内部建立这样的文件结构：

```
demo02
├── config
│   ├── __init__.py
│   ├── config.py
│   ├── dev_config.py
│   └── pro_config.py
└── run.py
```

然后使用类继承的方式使这三个配置文件联系起来，比如在 `config.py` 中就只放公有配置，如：

```
#!/usr/bin/env python
import os


class Config():
    """
    Basic config for demo02
    """
    # Application config
    TIMEZONE = 'Asia/Shanghai'
    BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

而在 `pro_config.py或dev_config.py` 中就可以自由地编写不同的配置了：

```
# dev_config
```

```
#!/usr/bin/env python
from .config import Config


class DevConfig(Config):
    """
    Dev config for demo02
    """

    # Application config
    DEBUG = True

# pro_config
#!/usr/bin/env python
from .config import Config


class ProConfig(Config):
    """
    Pro config for demo02
    """

    # Application config
    DEBUG = False
```

配置文件还需要根据系统环境变量的设置进行不同配置环境的切换，比如设置 `MODE` 系统环境变量，这里从系统环境变量得到配置也是个不错的方法，一般说利用 `gunicorn` 配置 `worker` 数目之类的，都可以使用这种方案。

然后可以根据其不同的值切换到不同的配置文件，因此在 `__init__.py` 中需要这么写：

```
#!/usr/bin/env python
import os

```

```
4.
5.  def load_config():
6.      """
7.      Load a config class
8.      """
9.
10.     mode = os.environ.get('MODE', 'DEV')
11.     try:
12.         if mode == 'PRO':
13.             from .pro_config import ProConfig
14.             return ProConfig
15.         elif mode == 'DEV':
16.             from .dev_config import DevConfig
17.             return DevConfig
18.         else:
19.             from .dev_config import DevConfig
20.             return DevConfig
21.     except ImportError:
22.         from .config import Config
23.         return Config
24.
25.
26. CONFIG = load_config()
```

默认 `MODE` 设置为 `DEV`，在 `run.py` 文件中就可以这么调用：

```
1.  #!/usr/bin/env python
2.  from sanic import Sanic
3.  from sanic.response import text
4.  from config import CONFIG
5.
6.  app = Sanic()
7.  app.config.from_object(CONFIG)
8.
9.  @app.route("/")
10. async def test(request):
11.     return text('Hello World!')
12.
```

```
13.
14. if __name__ == "__main__":
15.     app.run(host="0.0.0.0", port=8000, debug=app.config['DEBUG'])
```

而在生产环境的服务器上，直接通过设置系统变量就可以达到配置修改的目的了，如下：

```
1. # 通过设置MODE的值进行配置文件的选择
2. export MODE=PRO
```

若是利用 `supervisor` 来启动服务，可通过添加 `environment = MODE="PRO"` 来设置环境变量，是不是很方便呢。

说明

其实我编写这种微服务，配置更新是很正常且很频繁的需求，这样的话我就必须要求我的代码可以实现热更新，也就是可以迅速的修改配置，且迅速的生效，目前我使用的是 `ZooKeeper` 来实现这个需求，有兴趣的朋友可以详细了解，或许你也是用这个方案呢？

如果你有更好的方案，不妨告知一二。

3.项目结构

项目结构

通过前面的讲解，我们了解了 `Sanic` 的运行方式以及编写一个好的配置方案，是不是想要立马编写一个应用练练手呢？别急，请先看完这一章节，了解一下你要写的应用得用什么样的结构。

在 `github` 上也看了不少的 `Python` 项目吧，相信你也清楚，一个项目，在最外层他们应该是一样的，简单概括下，大概是下面这样的结构：

```
pro_name
├── docs               # 项目文档说明
├── src or pro_name/# 项目名称
├── tests              # 测试用例
├── README.md          # 项目介绍
└──requirements.txt # 该项目依赖的第三方库
```

那接下来需要讨论的，就是 `src` 或者说 `pro_name`（这个就看你心情命名了，一般与最外层一样的名字）的内部结构该是什么样的呢？

本章将写一个 `rss` 解析展示的项目用做演示。

普通的项目结构

一个普通的项目：

- 不需要添加后续模块功能
- 快速开发使用，不需要维护
- 无较复杂的前端需求
- 用完就走

那么就可以像 `demo01` 中一样，只需要添加一个 `run.py` 或者叫做 `app.py` 文件（反正这是一个启动文件，命名可随意），不论是配置、路由都写在一起就好了。

新建一个项目如下：

```
sample01
├── docs
│   └── demo.md
├── src
│   └── run.py
├── tests
├── .gitignore
└──requirements.txt
```

任意一个 `rss` 源，假设项目需要将其中的文章标题以及链接提取并展示出来，比如以json格式返回，这属于很简单的功能，可以说只有一段逻辑，`run.py` 内容如下：

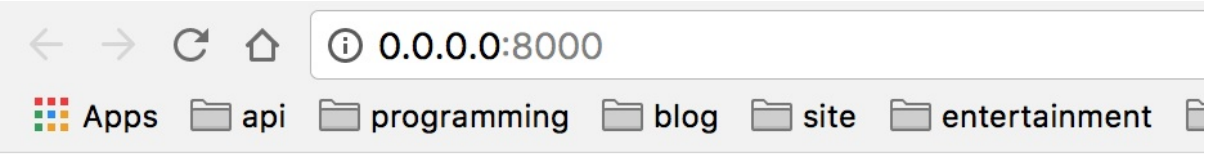
```
#!/usr/bin/env python
from sanic import Sanic
from sanic.response import json
from feedparser import parse

app = Sanic()


@app.route("/")
async def index(request):
    url = "http://blog.howie6879.cn/atom.xml"
```

```
12.     feed = parse(url)
13.     articles = feed['entries']
14.     data = []
15.     for article in articles:
16.         data.append({"title": article["title_detail"]["value"],
    "link": article["link"]})
17.     return json(data)
18.
19.
20. if __name__ == "__main__":
21.     app.run(host="0.0.0.0", port=8000)
```

访问 `http://0.0.0.0:8000/`，会返回一串json，如下：




```
[
  - {
        title: "2.Docker - 实例演示 - owllook",
        link: "http://blog.howie6879.cn/2017/08/22/27/"
    },
  - {
        title: "1.Docker - 初使用",
        link: "http://blog.howie6879.cn/2017/08/15/26/"
    },
  - {
        title: "gRPC使用初试",
        link: "http://blog.howie6879.cn/2017/08/03/25/"
    },
  - {
        title: "talonspider - 简单的爬虫框架",
        link: "http://blog.howie6879.cn/2017/06/07/24/"
    },
  - {
        title: "你的浏览器可好|Chrome插件篇",
        link: "http://blog.howie6879.cn/2017/05/11/23/"
    },
  - {
        title: "owllook -- 一个简洁的网络小说搜索引擎",
        link: "http://blog.howie6879.cn/2017/03/10/22/"
    },
  - {
        title: "sanic使用记录",
        link: "http://blog.howie6879.cn/2017/02/28/21/"
    },
```


和我们想象地一样，返回了一串 `json` ，接下来，问题升级，我想要将标题链接用页面展示，该怎么弄？

很容易想到，我们需要一个页面模板来承载数据，然后将json数据写入到页面模板中，最后用 `jinja2` 的 `template` 将其渲染。

道理我们都懂，`Sanic`具体需要怎么渲染呢？说白了就是对`jinja2`的使用，如下：

```
#!/usr/bin/env python
from sanic import Sanic
from sanic.response import json, text, html
from feedparser import parse
from jinja2 import Template

app = Sanic()

# 后面会使用更方便的模板引用方式
template = Template(
    """
    <!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>rss阅读</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
<article class="markdown-body">
    {% for article in articles %}
    <b><a href="{{article.link}}">{{article.title}}</a></b><br/>
    <i>{{article.published}}</i><br/>
    <hr/>
    {% endfor %}
</article>
</body>
</html>
    """
)


@app.route("/")
async def index(request):
```

```
35.     url = "http://blog.howie6879.cn/atom.xml"
36.     feed = parse(url)
37.     articles = feed['entries']
38.     data = []
39.     for article in articles:
40.         data.append({"title": article["title_detail"]["value"],
    "link": article["link"]})
41.     return json(data)
42.
43.
44. @app.route("/html")
45. async def rss_html(request):
46.     url = "http://blog.howie6879.cn/atom.xml"
47.     feed = parse(url)
48.     articles = feed['entries']
49.     data = []
50.     for article in articles:
51.         data.append(
52.             {"title": article["title_detail"]["value"], "link":
    article["link"], "published": article["published"]})
53.     html_content = template.render(articles=data)
54.     return html(html_content)
55.
56.
57. if __name__ == "__main__":
58.     app.run(host="0.0.0.0", port=8000)
```

具体结构代码见sample01，运行起来，然后输入

`http://0.0.0.0:8000/html` 就可以看到被展示出来的页面^_^

假设需要编写前端页面比较多，那么你就需要添加 `statics` 以及 `templates` 文件夹用来管理各个界面模块，具体下面会介绍。

项目结构具体说明

当编写的项目过于复杂，我都会将其当做一个第三方包来管理项目中涉及的各种模块，比如 `sample02`，目录下面你会发现有个 `__init__.py` 文件，它初始化了当前目录下的应用，然后代码中引用

某个函数可以这么写：

```
from src.views import app
```

这样，你的应用下面的模块引用起来就会特别方便，就像使用一个第三方模块一样，灵巧且方便。

每个项目的内部分布以及命名可能不一样（甚至目录比应该或多或少），但大体意思可能差不多，下面介绍本次项目 `src` 下的一些文件目录结构：

```
sample02
├── docs
│   └── demo.md
├── src
│   ├── config # 配置
│   ├── statics # css、js、img
│   ├── templates # Jinja2模板
│   └── views # 路由、逻辑处理
│   ├── __init__.py
│   ├── run.py # 启动文件
├── tests
└── requirements.txt
```

此处就可以将 `sample02` 当成一个包了，实践是检验真理的唯一标准，让我们来试试看：

首先新建文件 `/views/rss.py` ，具体代码可以看这里 sample02，下面的代码片段可没办法很好的运行：

```
enable_async = sys.version_info >= (3, 6)

app = Sanic()

# jinjia2 config
```

```
env = Environment(
    loader=PackageLoader('views.rss', '../templates'),
    autoescape=select_autoescape(['html', 'xml', 'tpl']),
    enable_async=enable_async)


async def template(tpl, **kwargs):
    template = env.get_template(tpl)
    rendered_template = await template.render_async(**kwargs)
    return html(rendered_template)


@app.route("/html")
async def rss_html(request):
    url = "http://blog.howie6879.cn/atom.xml"
    feed = parse(url)
    articles = feed['entries']
    data = []
    for article in articles:
        data.append(
            {"title": article["title_detail"]["value"], "link": article["link"], "published": article["published"]})
    return await template('rss.html', articles=articles)
```

这里使用异步的方式引入了 `jinja2` ，需要注意的是python版本必须3.6+，否则就得使用同步的方式来引入 `jinja2` ，后面章节会继续介绍。

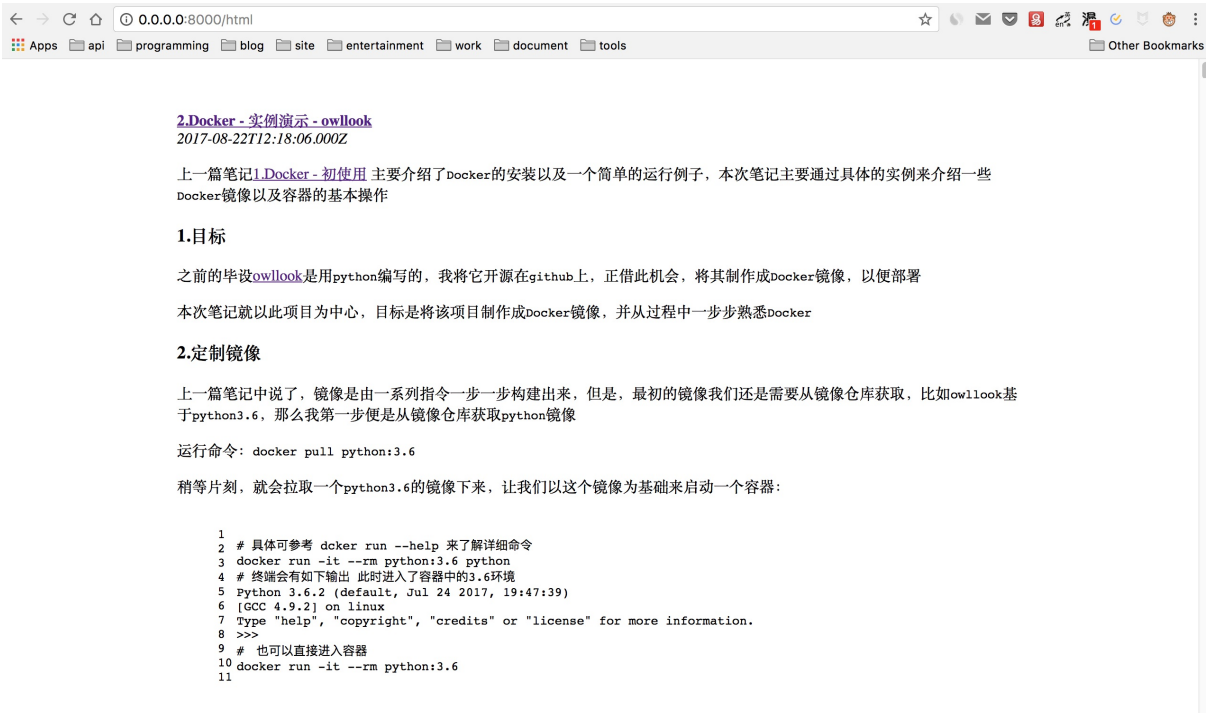
此时启动文件 `run.py` 只要引入 `/views/rss.py` 的 `app` 实例即可：

```
# !/usr/bin/env python
import sys
import os

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__fil
from src.views import app
```

```
7.  from src.config import CONFIG
8.
9.  app.statics('/statics', CONFIG.BASE_DIR + '/statics')
10.
11. if __name__ == "__main__":
12.     app.run(host="0.0.0.0", port=8000)
```

还有一些css文件这里就不介绍，具体代码请看sample02，运行起来，然后输入 `http://0.0.0.0:8000/html` 就好，效果如下图：



说明

关于 `views templates statics` 内部的构造也是值得一写，这就涉及到蓝图 `Blueprint` ，后面介绍蓝图的时候会进行介绍。

代码地址：demo03

4.展示一个页面

展示一个页面

前面一章介绍项目结构的时候，很粗略地讲了下如何将rss的文章内容在网页上进行展示。

相信你应该已经了解清楚， `sanic` 是怎么接收请求并返回被请求的资源的，简单来说概括如下：

- 接收请求
- 找到对应的路由并执行路由对应的视图函数
- Jinja2模板渲染返回视图

路由和视图函数

在此我假设你理解 `python` 中的装饰器，如果你并不清楚，可以看我另写的关于装饰器的介绍，回归正题，还记得第一节中的代码实例么？

```
1. #!/usr/bin/env python
2. from sanic import Sanic
3. from sanic.response import text
4. 
5. app = Sanic()
6. 
7. # 此处将路由 / 与视图函数 test 关联起来
8. @app.route("/")
9. async def test(request):
```

```
10.     return text('Hello World!')
11.
12.
13. if __name__ == "__main__":
14.     app.run(host="0.0.0.0", port=8000)
```

在前言介绍里，出现这几个名词 路由 视图函数 视图 ，在上面那段代码中， `test` 就是视图函数。

这是一段执行逻辑，比如客户端请求 `0.0.0.0:8000/` 此时返回的内容就是由 `test` 这个视图函数提供的。

在我看来，视图函数就是一个纽带，它起到承上启下的作用，那么，到底是怎样的承上启下呢？让我们结合代码(sanic0.1.2源码)来分析下：

```
1.
2. @app.route("/")
3. async def test(request):
4.     return text('Hello World!')
```

这个路由装饰器的作用很简单，就是将 `/` 这个 `uri` 与视图函数 `test` 关联起来，或许你可以将路由想象成一个 `dict` ，当客户端若请求 `0.0.0.0:8000/` ，路由就会 `get` `/` 对应的视图函数 `test` ，然后执行。

其实真实情况和我们想象的差不多，请看 `sanic.py` 中的第三十行：

```
1.
2. # Decorator
3. def route(self, uri, methods=None):
4.
5.     def response(handler):
6.         # 路由类的add 方法将视图函数handler 与uri 关联起来
7.         # 然后整个路由列表会新增一个 namedtuple 如下：
```

```
8.          # Route(handler=handler, methods=methods_dict,
    pattern=pattern, parameters=parameters)
9.          self.router.add(uri=uri, methods=methods, handler=handler)
10.         return handler
11.
12.     return response
```

此时，路由就和 `uri` 对应的视图函数关联起来了，这就是承上，路由和视图函数就是这样对应的关系。
103行有个 `handle_request` 函数：

```
1.
2.  async def handle_request(self, request, response_callback):
3.      """
4.      Takes a request from the HTTP Server and returns a response
    object to be sent back
5.      The HTTP Server only expects a response object, so exception
    handling must be done here
6.      :param request: HTTP Request object
7.      :param response_callback: Response function to be called with
    the response as the only argument
8.      :return: Nothing
9.      """
```

当服务器监听到某个请求的时候， `handle_request` 可以通过参数中的 `request.url` 来找到视图函数并且执行，随即生成视图返回，这便是所谓的启下。

其实浏览器显示的内容就是我们所谓的视图，视图是视图函数生成的，其中Jinja2起到模板渲染的作用。

蓝图

到这里，你一定已经很明白sanic框架是怎么处理一个请求并将视图返回给客户端，然后也掌握了如何编写自己定义的模板(html)以及样式

(css)，通过前面一节项目结构的介绍，我们可以总结出如下经验：

- 对于css、js等静态文件，常规操作是将其放在自己建立的statics下面
- 对于html模板，常规操作是将其放在自己建立的templates下面
- 视图函数(即服务的逻辑实现)放在自己建立的views下面

是时候考虑以下这种情况了，你需要编写一个比较复杂的http服务，你需要定义几百个路由，在编写过程中你会发现有许多不顺心的地方：

- 各种不同类型的路由互相交杂在一起，命名困难，管理困难
- 不同页面的css文件同样堆积在一起，html也是如此
- …

实在是令人烦恼，可能你想要一个模块化的编写方式，一个文件编写后台，url统一是 `/admin/***` ，一个文件编写发帖， `/post/***` 等等，各个文件下面url自动会带上自定义的前缀，不用考虑命名问题，不用重复写url前缀，多么美好

Blueprint，就是sanic为你提供的解决方案，依然是上节rss的例子，让我们利用Blueprint来简单实现一下我们的需求，比如我们要构建的网站分为两个部分：

- `/json/index` 返回json格式的数据
- `/html/index` 返回html视图

我们在上节代码的基础上添加Blueprint，先看看定好的项目结构 demo04，和前面相比，现在的项目结构是不是感觉很丰富？继续往下看：

```
1.
2. .
3. ├── __init__.py
```

```
├── config
│   ├── __init__.py
│   ├── config.py
│   ├── dev_config.py
│   └── pro_config.py
├── run.py
├── statics
│   ├── rss_html # rss_html蓝图的 css js 文件存放目录
│   │   ├── css
│   │   │   └── main.css
│   │   └── js
│   │       └── main.js
│   └── rss_json # rss_json蓝图的 css js 文件存放目录
│   │   ├── css
│   │   │   └── main.css
│   │   └── js
│   │       └── main.js
├── templates
│   ├── rss_html # rss_html蓝图的 html 文件存放目录
│   │   ├── index.html
│   │   └── rss.html
│   └── rss_json # rss_json蓝图的 html 文件存放目录
│       └── index.html
└── views
    ├── __init__.py
    ├── rss_html.py # rss_html 蓝图
    └── rss_json.py # rss_json 蓝图
```

蓝图的目的就是让我们构建的项目灵活，可扩展，容易阅读，方便管理，一个个小的蓝图就构建了一个大型的项目，run.py里面可以随意组合注册蓝图，可抽插式的构建我们的服务，sample01里是本次示例的代码，建议先读一遍，十分简单，主要就是讲路由根据你自己定义的特性分割成一个个蓝图，比如这里的 `rss_html` 和 `rss_json` ，请特别注意项目中对于静态文件以及模板文件的路径配置：

```
1.
```

```
#!/usr/bin/env python
# 部分代码
# rss_html.py
import sys

from sanic import Blueprint
from sanic.response import html

from src.config import CONFIG


html_bp = Blueprint('rss_html', url_prefix='html')
html_bp.static('/statics/rss_html', CONFIG.BASE_DIR +
'/statics/rss_html')

# jinjia2 config
env = Environment(
    loader=PackageLoader('views.rss_html',
'../templates/rss_html'),
    autoescape=select_autoescape(['html', 'xml', 'tpl']),
    enable_async=enable_async)

@html_bp.route("/")
async def index(request):
    return await template('index.html')

#!/usr/bin/env python
# 部分代码
# rss_json.py
import sys

from sanic import Blueprint
from sanic.response import html

from src.config import CONFIG


json_bp = Blueprint('rss_json', url_prefix='json')
```

```
38. json_bp.static('/statics/rss_json', CONFIG.BASE_DIR +
    '/statics/rss_json')
39.
40. # jinjia2 config
41. env = Environment(
42.     loader=PackageLoader('views.rss_json',
    '../templates/rss_json'),
43.     autoescape=select_autoescape(['html', 'xml', 'tpl']),
44.     enable_async=enable_async)
45.
46. @json_bp.route("/")
47. async def index(request):
48.     return await template('index.html')
```

不知你是否感受到这样编写服务的好处，让我们列举下：

- 每个蓝图都有其自己定义的前缀，如 /html/ /json/
- 不同蓝图下面route可相同且互不冲突
- html以及css等文件可根据蓝图名称目录引用，条理清晰易扩展
- 模块化
- …

不多说，看运行效果：

```
1.
2. cd /Sanic-For-Pythoneer/examples/demo04/sample01/src
3. python run.py
```

现在，请访问：

- http://0.0.0.0:8000/html/
- http://0.0.0.0:8000/json/
- http://0.0.0.0:8000/html/index
- http://0.0.0.0:8000/json/index

感觉这个设计方便了全世界^_^

说明

蓝图的具体使用还请诸位多多摸索，说不定能发掘出更多好玩的玩法以及更加优雅的编码方式，本章的代码地址：demo04

5.数据库使用

数据库使用

介绍中说的很明白，`Sanic` 是一个可以使用 `async/await` 语法编写项目的异步非阻塞框架，既然是异步框架，那么在使用过程中用到的第三方包也最好是异步的，比如http请求，最好就使用 `aihttp` 而非 `requests`，对于数据库的连接，也是同样如此，下面我将用代码的形式来说明下如何在Sanic中连接数据库。

操作Mysql

对于mysql数据库的异步操作，我只在一些脚本中用过，用的是aiomysql，其中官方文档中讲得很清楚，也支持结合 `sqlalchemy` 编写 `ORM`，然后aiomysql提供了自己编写的异步引擎。

```
from aiomysql.sa import create_engine
# 这个才是关键
```

下面我编写一个具体的例子来用异步语句操作下数据库，首先建立如下目录:

```
aio_mysql
├── demo.py
```

```
├── model.py
└── requirements.txt
```

建立表：

```
create database test_mysql;

CREATE TABLE user
(
  id        INT AUTO_INCREMENT
    PRIMARY KEY,
  user_name VARCHAR(16) NOT NULL,
  pwd       VARCHAR(32) NOT NULL,
  real_name VARCHAR(6)  NOT NULL
);
```

一切准备就绪，下面编写代码：

```
# script: model.py
import sqlalchemy as sa

metadata = sa.MetaData()

user = sa.Table(
    'user',
    metadata,
    sa.Column('id', sa.Integer, autoincrement=True, primary_key=True),
    sa.Column('user_name', sa.String(16), nullable=False),
    sa.Column('pwd', sa.String(32), nullable=False),
    sa.Column('real_name', sa.String(6), nullable=False),
)

# script: demo.py
import asyncio

from aiomysql.sa import create_engine
```

```
19.
20. from model import user,metadata
21.
22.
23. async def go(loop):
24.     """
25.     aiomysql项目地址：https://github.com/aio-libs/aiomysql
26.     :param loop:
27.     :return:
28.     """
29.     engine = await create_engine(user='root', db='test_mysql',
30.                                  host='127.0.0.1',
    password='123456', loop=loop)
31.     async with engine.acquire() as conn:
32.         await
    conn.execute(user.insert().values(user_name='user_name01',
    pwd='123456', real_name='real_name01'))
33.         await conn.execute('commit')
34.
35.         async for row in conn.execute(user.select()):
36.             print(row.user_name, row.pwd)
37.
38.     engine.close()
39.     await engine.wait_closed()
40.
41.
42. loop = asyncio.get_event_loop()
43. loop.run_until_complete(go(loop))
```

运行 python demo.py，会看到如下输出：

```
1. user_name01 123456
```

很简单吧，具体示例见aio_mysql，如果你比较喜欢类似SQLAlchemy的操作方式，这里推荐一个异步ORM，gino。

操作MongoDB

我业余写的一个项目，基本用的就是 `MongoDB` 来储存数据，对于异步操作 `MongoDB` ，目前Python主要用的是motor，使用起来依旧很简单，但是结合具体功能，就有不同的需求，最后就会形成各种各样的连接方案，这里我主要分享下自己是如何使用的，目录如下所示：

```
aio_mongo
├── demo.py
└── requirements.txt
```

`MongoDB` 是一个基于分布式文件存储的数据库，它介于关系数据库和非关系数据库之间，所以它使用起来也是比较灵活的，打开 `demo.py` ：

```
#!/usr/bin/env python
import os

from functools import wraps

from motor.motor_asyncio import AsyncIOMotorClient

MONGODB = dict(
    MONGO_HOST=os.getenv('MONGO_HOST', ""),
    MONGO_PORT=os.getenv('MONGO_PORT', 27017),
    MONGO_USERNAME=os.getenv('MONGO_USERNAME', ""),
    MONGO_PASSWORD=os.getenv('MONGO_PASSWORD', ""),
    DATABASE='test_mongodb',
)

class MotorBaseOld:
    """
    默认实现了一个db只创建一次，缺点是更换集合麻烦
    """
    _db = None
    MONGODB = MONGODB

    def client(self, db):
        # motor
```

```
25.         self.motor_uri = 'mongodb://{account}{host}:
    {port}/{database}'.format(
26.             account='{username}:{password}@'.format(
27.                 username=self.MONGODB['MONGO_USERNAME'],
28.                 password=self.MONGODB['MONGO_PASSWORD']) if
    self.MONGODB['MONGO_USERNAME'] else '',
29.             host=self.MONGODB['MONGO_HOST'] if
    self.MONGODB['MONGO_HOST'] else 'localhost',
30.             port=self.MONGODB['MONGO_PORT'] if
    self.MONGODB['MONGO_PORT'] else 27017,
31.             database=db)
32.         return AsyncIOMotorClient(self.motor_uri)
33.
34.     @property
35.     def db(self):
36.         if self._db is None:
37.             self._db = self.client(self.MONGODB['DATABASE'])
    [self.MONGODB['DATABASE']]
38.
39.         return self._db
```

我最开始，使用的是这种方式来连接 `MongoDB` ，上面代码保证了集合中的db被_db维护，保证只会创建一次，如果你项目中不会随意更改集合的话，也没什么大问题，如果不是，我推荐使用下面这样的连接方式，可以自由地更换集合与db：

```
1.
2.  def singleton(cls):
3.      """
4.      用装饰器实现的实例 不明白装饰器可见附录 装饰器：
    https://github.com/howie6879/Sanic-For-
    Pythoneer/blob/master/docs/part2/%E9%99%84%E5%BD%95%EF%BC%9A%E5%85%B3
5.      :param cls: cls
6.      :return: instance
7.      """
8.      _instances = {}
9.
```

```
10.     @wraps(cls)
11.     def instance(*args, **kw):
12.         if cls not in _instances:
13.             _instances[cls] = cls(*args, **kw)
14.         return _instances[cls]
15.
16.     return instance
17.
18. @singleton
19. class MotorBase:
20.     """
21.     更改mongodb连接方式 单例模式下支持多库操作
22.     About motor's doc: https://github.com/mongodb/motor
23.     """
24.     _db = {}
25.     _collection = {}
26.     MONGODB = MONGODB
27.
28.     def __init__(self):
29.         self.motor_uri = ''
30.
31.     def client(self, db):
32.         # motor
33.         self.motor_uri = 'mongodb://{account}{host}:
    {port}/{database}'.format(
34.             account='{username}:{password}@'.format(
35.                 username=self.MONGODB['MONGO_USERNAME'],
36.                 password=self.MONGODB['MONGO_PASSWORD']) if
    self.MONGODB['MONGO_USERNAME'] else '',
37.             host=self.MONGODB['MONGO_HOST'] if
    self.MONGODB['MONGO_HOST'] else 'localhost',
38.             port=self.MONGODB['MONGO_PORT'] if
    self.MONGODB['MONGO_PORT'] else 27017,
39.             database=db)
40.         return AsyncIOMotorClient(self.motor_uri)
41.
42.     def get_db(self, db=MONGODB['DATABASE']):
43.         """
```

```
44.         获取一个db实例
45.         :param db: database name
46.         :return: the motor db instance
47.         """
48.         if db not in self._db:
49.             self._db[db] = self.client(db)[db]
50.
51.         return self._db[db]
52.
53.     def get_collection(self, db_name, collection):
54.         """
55.         获取一个集合实例
56.         :param db_name: database name
57.         :param collection: collection name
58.         :return: the motor collection instance
59.         """
60.         collection_key = db_name + collection
61.         if collection_key not in self._collection:
62.             self._collection[collection_key] = self.get_db(db_name)
    [collection]
63.
64.         return self._collection[collection_key]
```

为了避免重复创建MotorBase实例，可以实现一个单例模式来保证资源的有效利用，具体代码以及运行demo见aio_mongo

操作Redis

对于Redis的异步操作，我选用的是 `asyncio_redis` ，你大可不必非要使用这个，或许其他的库实现地更好，我只是用这个举个例子，建立如下目录：

```
1. aio_redis
2. ├── demo.py
3. └── requirements.txt
```

建立一个redis连接池:

```
#!/usr/bin/env python
import os
import asyncio_redis

REDIS_DICT = dict(
    IS_CACHE=True,
    REDIS_ENDPOINT=os.getenv('REDIS_ENDPOINT', "localhost"),
    REDIS_PORT=os.getenv('REDIS_PORT', 6379),
    REDIS_PASSWORD=os.getenv('REDIS_PASSWORD', None),
    DB=0,
    POOLSIZE=10,
)


class RedisSession:
    """
    建立redis连接池
    """
    _pool = None

    async def get_redis_pool(self):
        if not self._pool:
            self._pool = await asyncio_redis.Pool.create(
                host=str(REDIS_DICT.get('REDIS_ENDPOINT', "localhost")), port=int(REDIS_DICT.get('REDIS_PORT', 6379)),
                poolsize=int(REDIS_DICT.get('POOLSIZE', 10)), password=REDIS_DICT.get('REDIS_PASSWORD', None),
                db=REDIS_DICT.get('DB', None)
            )

        return self._pool
```

具体见aio_redis，使用起来很简单，不做多叙述。

说明

如果你使用其它类型的数据库，其实使用方式也是类似。

本章代码地址，见demo05

6.常用的技巧

常用的技巧

结合前面讲的配置、项目结构、页面渲染、数据库连接，构造一个优雅的Sanic应用对你来说估计没什么大问题了，但是在实际使用过程中，可能你会碰到各种各样的需求，与之对应，你也会遇到千奇百怪的问题，除了在官方pro提issue，你大部分问题都需要自己去面对，看官方的介绍：

> *Async Python 3.5+ web server that's written to go fast*

大概就可以明白 `Sanic` 框架的重心不会放在诸如 `session cache reload authorized` 这些问题上。

此篇我会将我遇到的一些问题以及解决方案一一记录下来，估计会持续更新，因为问题是不断的哈哈，可能有些问题与前面讲的有些重复，你大可略过，我将其总结成一些小技巧，供大家参考，具体如下：

- api请求json参数以及api接口验证
- gRPC的异步调用方式
- Blueprint

- html&templates编写
- cache
- 热加载
- session

对于一些问题，我将编写一个小服务来演示这些技巧，具体见demo06，依旧使用前面rss的那个例子，经过修改一番后的rss例子现在目录变成这样子了，里面加了我遇到的各种问题的解决方案，或许你只需要关注你想要了解的就好，如果你有其他的问题，欢迎issue提问，目录大概如下所示：

```
src
├── config
│   ├── __init__.py
│   ├── config.py
│   ├── dev_config.py
│   └── pro_config.py
├── database
│   ├── __init__.py
│   └── redis_base
├── grpc_service
│   ├── __init__.py
│   ├── grpc_asyncio_client.py
│   ├── grpc_client.py
│   ├── grpc_server.py
│   ├── hello_grpc.py
│   ├── hello_pb2.py
│   ├── hello_pb2_grpc.py
│   └── proto
├── statics
│   ├── rss_html
│   │   ├── css
│   │   └── js
│   └── rss_json
│       ├── css
```

```
25. |       └── js
26. ├── templates
27. |   ├── rss_html
28. |   └── rss_json
29. ├── tools
30. |   ├── __init__.py
31. |   └── mid_decorator.py
32. ├── views
33. |   ├── __init__.py
34. |   ├── rss_api.py
35. |   ├── rss_html.py
36. |   └── rss_json.py
37. └── run.py
```

和前面相比，可以看到目录里面增加了几个目录和文件，一个是关于数据库的 `database` ，以及关于gRPC的 `grpc_service` ，其实主要是为了演示而已，如果想了解，就可以看关于gRPC的部分，不喜欢就看其他的。

增加的文件是 `rss_json.py` ，假设这是个接口文件吧，将会根据你的请求参数返回一串json，具体还是建议直接去看代码吧，点这里 sample。

验证问题

假设你正在编写一个api服务，比如根据传的blog名字返回其rss数据，比如 `rss_json.py` ：

```
1. #!/usr/bin/env python
2. from feedparser import parse
3. from sanic import Blueprint
4. from sanic.response import json
5.
6. api_bp = Blueprint('rss_api', url_prefix='v1')
7.
```

```
8.
9.  @api_bp.route("/get/rss/<param>")
10. async def get_rss_json(request, param):
11.     if param == 'howie6879':
12.         url = "http://blog.howie6879.cn/atom.xml"
13.         feed = parse(url)
14.         articles = feed['entries']
15.         data = []
16.         for article in articles:
17.             data.append({"title": article["title_detail"]["value"],
    "link": article["link"]})
18.         return json(data)
19.     else:
20.         return json({'info': '请访问
    http://0.0.0.0:8000/v1/get/rss/howie6879'})
```

启动服务后，此时用 `GET` 方式访问 `http://0.0.0.0:8000/v1/get/rss/howie6879` ，就会返回一串你需要的json，这样使用，没什么问题，好，下面改成post请求，其中请求的参数如下：

```
1. {
2.     "name": "howie6879"
3. }
```

在 `rss_json.py` 中加入一个新的视图函数：

```
1. @api_bp.route("/post/rss/", methods=['POST'])
2. async def post_rss_json(request, **kwargs):
3.     post_data = json_loads(str(request.body, encoding='utf-8'))
4.     name = post_data.get('name')
5.     if name == 'howie6879':
6.         url = "http://blog.howie6879.cn/atom.xml"
7.         feed = parse(url)
8.         articles = feed['entries']
9.         data = []
```

```
        for article in articles:
            data.append({"title": article["title_detail"]["value"], "link": article["link"]})
        return json(data)
    else:
        return json({'info': '参数错误'})
```

发送post请求到 `http://0.0.0.0:8000/v1/post/rss/` ，依旧和上面的结果一样，代码里面有个问题，我们需要判断 `name` 参数是不是存在于post过来的data中，解决方案很简单，加一个判断就好了，可这个是最佳方案么？

显然并不是的，如果有十个参数呢？然后再增加十几个路由呢？每个路由里有十个参数需要判断，想象一下，这样实施下来，难道要在代码里面堆积满屏幕的判断语句么？这样实在是太可怕了，我们需要更好更通用的解决方案，是时候写个装饰器来验证参数了，打开 `mid_decorator.py` 文件，添加一个验证的装饰器函数：

```
def auth_params(*keys):
    """
    api请求参数验证
    :param keys: params
    :return:
    """

    def wrapper(func):
        @wraps(func)
        async def auth_param(request=None, rpc_data=None, *args, **kwargs):
            request_params, params = {}, []
            if isinstance(request, Request):
                # sanic request
                if request.method == 'POST':
                    try:
                        post_data = json_loads(str(request.body,
```

```
    encoding='utf-8'))
17.                     except Exception as e:
18.                         return response_handle(request, {'info':
    'error'})
19.                     else:
20.                         request_params.update(post_data)
21.                         params = [key for key, value in
    post_data.items() if value]
22.                 elif request.method == 'GET':
23.                     request_params.update(request.args)
24.                     params = [key for key, value in
    request.args.items() if value]
25.                 else:
26.                     return response_handle(request, {'info':
    'error'})
27.             else:
28.                 pass
29.
30.             if set(keys).issubset(set(params)):
31.                 kwargs['request_params'] = request_params
32.                 return await dec_func(func, request, *args,
    **kwargs)
33.             else:
34.                 return response_handle(request, {'info': 'error'})
35.
36.         return auth_param
37.
38.     return wrapper
39.
40.
41. async def dec_func(func, request, *args, **kwargs):
42.     try:
43.         response = await func(request, *args, **kwargs)
44.         return response
45.     except Exception as e:
46.         return response_handle(request, {'info': 'error'})
```

注意，上面增加的路由函数改为这样：

```
@api_bp.route("/post/rss/", methods=['POST'])
@auth_params('name')
async def post_rss_json(request, **kwargs):
```

这样一个请求进来，就会验证参数 `name` 是否存在，而在视图函数里面，就可以放心大胆地使用传进来的参数了，而且对于其他不同的参数验证，只要按照这个写法，直接增加验证参数就好，十分灵活方便。

对于请求验证的问题，解决方法也是类似，就是利用装饰器来实现，我自己也实现过，在上面的代码链接里面可以找到，不过现在的 `Sanic` 官方已经提供了 `demo` ，见这里

gRPC的异步调用方式

在编写微服务的时候，除了需要支持 `http` 请求外，一般还需要支持 `gRPC` 请求，我在使用 `Sanic` 编写微服务的时候，遇到关于异步请求 `RPC` 的需求，当时确实困扰了我，意外发现了这个库grpclib，进入 `src/grpc_service` 目录，里面就是解决方案，很简单，这里就不多说了，直接看代码就好。

Blueprint

Blueprint前面的章节有仔细聊过，这里不多说，借用官方文档的例子，一个简单的sanic服务就搭好了：

```
# main.py
from sanic import Sanic
from sanic.response import json

app = Sanic()

@app.route("/")
async def test(request):
```

```
    return json({"hello": "world"})

#访问http://0.0.0.0:8000/即可
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

上面的例子可以当做一个完整的小应用，关于Blueprint的概念，可以这么理解，一个蓝图可以独立完成某一个任务，包括模板文件，静态文件，路由都是独立的，而一个应用可以通过注册许多蓝图来进行构建。
比如我现在编写的项目，我使用的是功能式架构，具体如下：

```
├── server.py
├── static
│   └── novels
│       ├── css
│       │   └── result.css
│       ├── img
│       │   └── read_content.png
│       └── js
│           └── main.js
├── template
│   └── novels
│       └── index.html
└── views
    └── novels_blueprint.py
```

可以看到，总的templates以及静态文件还是放在一起，但是不同的blueprint则放在对应的文件夹中，还有一种分区式架构，则是将不同的templats以及static等文件夹全都放在不同的的blueprint中。
最后只要将每个单独的blueprint在主启动文件进行注册就好，上面的目录树是我以前的一个项目owllook的，也是用 `Sanic` 编写的，有兴趣可以看看，不过现在结构改变挺大。

html&templates编写

编写web服务，自然会涉及到html，sanic自带有html函数，但这并不能满足有些需求，故引入jinja2迫在眉睫，使用方法也很简单：

```
# 适用python3.5+
# 代码片段，明白意思就好
from sanic import Blueprint
from jinja2 import Environment, PackageLoader, select_autoescape

# 初始化blueprint并定义静态文件夹路径
bp = Blueprint('novels_blueprint')
bp.static('/static', './static/novels')

# jinjia2 config
env = Environment(
    loader=PackageLoader('views.novels_blueprint',
'../templates/novels'),
    autoescape=select_autoescape(['html', 'xml', 'tpl']))

def template(tpl, **kwargs):
    template = env.get_template(tpl)
    return html(template.render(kwargs))

@bp.route("/")
async def index(request):
    return template('index.html', title='index')
```

如果是 `python3.6` ，可以试试下面的写法：

```
# 适用python3.5+
# 代码片段，明白意思就好
#!/usr/bin/env python
import sys

from feedparser import parse
from jinja2 import Environment, PackageLoader, select_autoescape
```

```
from sanic import Blueprint
from sanic.response import html

from src.config import CONFIG

# https://github.com/channelcat/sanic/blob/5bb640ca1706a42a012109dc3d81
# 开启异步特性  要求3.6+
enable_async = sys.version_info >= (3, 6)

html_bp = Blueprint('rss_html', url_prefix='html')
html_bp.static('/statics/rss_html', CONFIG.BASE_DIR +
'/statics/rss_html')

# jinjia2 config
env = Environment(
    loader=PackageLoader('views.rss_html',
'../templates/rss_html'),
    autoescape=select_autoescape(['html', 'xml', 'tpl']),
    enable_async=enable_async)


async def template(tpl, **kwargs):
    template = env.get_template(tpl)
    rendered_template = await template.render_async(**kwargs)
    return html(rendered_template)
```

cache

我在项目中主要使用redis作为缓存，使用aiocache很方便就完成了我需要的功能，当然自己利用aioredis编写也不会复杂到哪里去。

比如上面的例子，每次访问 `http://0.0.0.0:8000/v1/get/rss/howie6879` ，都要请求一次对应的rss资源，如果做个缓存那岂不是简单很多？

改写成这样：

```
@cached(ttl=1000, cache=RedisCache, key="rss",
serializer=PickleSerializer(), port=6379, namespace="main")
async def get_rss():
    print("第一次请求休眠3秒...")
    await asyncio.sleep(3)
    url = "http://blog.howie6879.cn/atom.xml"
    feed = parse(url)
    articles = feed['entries']
    data = []
    for article in articles:
        data.append({"title": article["title_detail"]["value"],
"link": article["link"]})
    return data


@api_bp.route("/get/rss/<name>")
async def get_rss_json(request, name):
    if name == 'howie6879':
        data = await get_rss()
        return json(data)
    else:
        return json({'info': '请访问
http://0.0.0.0:8000/v1/get/rss/howie6879'})
```

为了体现缓存的速度，首次请求休眠3秒，请求过后，redis中就会将此次json数据缓存进去了，下次去请求就会直接冲redis读取数据。

带上装饰器，什么都解决了。

热加载

我发现有个pr实现了这个需求，如果你有兴趣，可以看这里

session

sanic对此有一个第三方插件 `sanic_session` ，用法非常简单，有兴趣

可以看看

7. 可靠的扩展

可靠的扩展

目前开源社区有不少人为 `Sanic` 框架编写了插件，这些插件很可能会在将来的某个时间帮助到你，比如缓存、模板渲染、api文档生成、Session…等等

官方也维护了一个扩展列表，见extensions

8.测试与部署

测试与部署

在项目结构那一节说过，一个服务的基本结构大概是怎么样的，这里再列出来回顾下：

```
pro_name
├── docs              # 项目文档说明
├── src or pro_name/# 项目名称
├── tests             # 测试用例
├── README.md         # 项目介绍
└──requirements.txt # 该项目依赖的第三方库
```

一个服务编写完成后，在部署之前，你需要做的一步就是进行单元测试，首先你要确定目前的代码是可以完美运行的，然后测试用例还可以让你在下次修改代码逻辑进行版本迭代的时候，只要再跑一次对应的测试用例就可以快速地确定此次的版本依旧是完美的，大大节省时间，一般集成测试的时候都需要跑测试用例的脚本。

本次使用的例子还是继续在demo06的基础上进行演示，提醒一下诸位，在继续阅读前可以先大致看下目录中 `test` 的代码哈。

测试

单元测试

Sanic进行单元测试的时候，官方推荐使用的是pytest，具体怎么对Sanic构建的服务进行测试呢，别急，Sanic开发团队提供了关于 `pytest` 的插件，见pytest-sanic，使用起来也是非常简单。

让我们结合前面的例子，利用pytest-sanic测试一下demo06中的 `rss api` 服务，先看下目录结构：

```
tests
├── setting.py
└── test_rss.py
```

首先在 `setting.py` 中定好请求的数据：

```
# setting.py
def rss_data():
    return {
        "name": "howie6879"
    }
```

然后编写对应的测试用例，这里是关于 `/v1/post/rss/` 的一个 `POST` 请求测试，代码如下：

```
# test_rss.py
async def test_http_rss(test_cli):
    data = setting.rss_data()
    response = await test_cli.post('/v1/post/rss/', data=ujson.dumps(data))
    resp_json = await response.json()
    assert resp_json['status'] == 1

# 运行测试 pytest tests/test_rss.py
"""
============================================== test session starts ==============================================
```

```
platform darwin -- Python 3.6.0, pytest-3.2.3, py-1.4.34, pluggy-0.4.0
rootdir: /Users/howie/Documents/programming/python/git/Sanic-For-Pythoneer/examples/demo06/sample, inifile:
plugins: celery-4.0.2, sanic-0.1.5
collected 2 items

tests/test_rss.py .s

========================================= 1 passed, 1 skipped in 2.13 seconds ==========================================
"""
```

可以看到测试通过，全部测试代码在这里，最好可以直接clone下来跑一遍，细心的朋友可能注意到了测试用例结果中的这句话 `1 passed, 1 skipped in 2.13 seconds` ，为什么会有一个测试跳过呢？

因为在实际编写项目的过程中，你的测试用例很可能会分好多种，比如在编写微服务的过程中，同样一套处理逻辑，你需要分别实现 `HTTP` 和 `gRPC` 两种调用方式，测试代码里面我就多写了一个测试 `gRPC` 的配置，不过我设置了参数： `DIS_GRPC_TEST = True` ，没有启用 `gRPC` 的测试，这里只是举个例子，具体还是要看诸位的需求，用本次的例子作为参考，就算改动起来也并没什么难度。

压力测试

说完了如何对Sanic编写的服务进行单元测试，接下来稍微讲下如何进行压力测试，压力测试最好在内外网都进行测试下，当然服务器配置是你定，然后在多个服务器上部署好服务，启动起来，利用负载均衡给压测代码一个固定的ip，这样对于服务的水平扩展测试就会很方便。

压测可以考虑使用locust，看看现在 `tests` 下的目录结构：

```
├── locust_rss
```

```
2. │   ├── __init__.py
3. │   ├── action.py
4. │   ├── locust_rss_http.py
5. │   ├── locustfile.py
6. │   └── utils.py
7. ├── setting.py
8. └── test_rss.py
```

新增了 `locust_rss` 文件夹，首先在 `action.py` 定义好请求地址与请求方式：

```
1. HTTP_URL = "http://0.0.0.0:8000/v1/post/rss/"
2. GRPC_URL = "0.0.0.0:8990"
3.
4.
5. def json_requests(client, data, url):
6.     func_name = inspect.stack()[1][3]
7.     headers = {'content-type': 'application/json'}
8.     return post_request(client, data=json.dumps(data), url=url,
   func_name=func_name, headers=headers)
9.
10.
11. def action_rss(client):
12.     data = {
13.         "name": "howie6879"
14.     }
15.     json_requests(client, data, HTTP_URL)
```

压测怎么个压测法，请求哪些接口，接口请求怎么分配，都在 `locust_rss_http.py` 里定好了：

```
1. class RssBehavior(TaskSet):
2.     @task(1)
3.     def interface_rss(self):
4.         action.action_rss(self.client)
```

然后需要发送请求给目标，还需要判断是否请求成功，这里将其封装成函数，放在 `utils.py` 里，比如 `post_request` 函数：

```
def post_request(client, data, url, func_name=None, **kw):
    """
    发起post请求
    """
    func_name = func_name if func_name else inspect.stack()[1][3]
    with client.post(url, data=data, name=func_name,
catch_response=True, timeout=2, **kw) as response:
        result = response.content
        res = to_json(result)
        if res['status'] == 1:
            response.success()
        else:
            response.failure("%s-> %s" % ('error', result))
        return result
```

`locustfile.py` 是压测的启动文件，必不可少，我们先请求一次，看看能不能请求成功，如果成功了再将其正式运行起来：

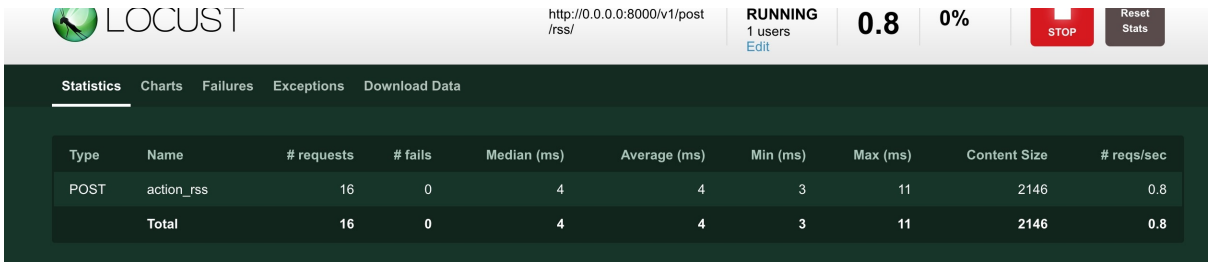
```
cd Sanic-For-Pythoneer/examples/demo06/sample/tests/locust_rss

# 只想跑一次看看有没有问题 记得先将你编写的服务启动起来哦
locust -f locustfile.py --no-web -c 1 -n 1

# Output: 表示没毛病
[2018-01-14 14:54:30,119] 192.168.2.100/INFO/locust.main: Shutting
down (exit code 0), bye.
 Name                                                          #
reqs      # fails     Avg     Min     Max  |  Median   req/s
--------------------------------------------------------------------
--------------------------------------------------------------------
------
 POST action_rss
1     0(0.00%)    1756    1756    1756  |    1800    0.00
--------------------------------------------------------------------
```

```
    ---------------------------------------------------------------------
    ------
 12.  Total
    1     0(0.00%)                                              0.00
 13.
 14. Percentage of the requests completed within given times
 15.  Name                                                           #
    reqs    50%    66%    75%    80%    90%    95%    98%    99%   100%
 16. ---------------------------------------------------------------------
    ---------------------------------------------------------------------
    ------
 17.  POST action_rss
    1   1800   1800   1800   1800   1800   1800   1800   1800   1756
 18. ---------------------------------------------------------------------
    ---------------------------------------------------------------------
    ------
```

好了，没问题了，可以执行 `locust -f locustfile.py` ，然后访问 `http://0.0.0.0:8089/` ，如下图：



当然，这里只是大概讲解下如何进行压测，至于真实环境下，还是需要诸位继续摸索。

部署

千辛万苦，终于到了这一步，我们历经代码编写、单元测试、压力测试终于到了这一步，将我们的服务正式部署！

在继续阅读之前，请你万万先读一遍官方的Deploying。

好了，你现在肯定知道了Sanic服务的两种启动方式，分别如下：

- python -m sanic server.app –host=0.0.0.0 –port=8000 –workers=4
- gunicorn myapp:app –bind 0.0.0.0:8000 –worker-class sanic.worker.GunicornWorker

至于选哪种启动方式，我觉得都可以，看你心情了，下面直接说下如何部署：

- Gunicorn + Supervisor + Caddy
- Docker

对于用Gunicorn启动，可以将配置写在自己定义的配置文件中，比如 `config/gunicorn.py` ：

```
# gunicorn.py
bind = '127.0.0.1:8001'
backlog = 2048

workers = 2
worker_connections = 1000
timeout = 30
keepalive = 2

spew = False
daemon = False
umask = 0
```

然后直接运行 `gunicorn -c config/gunicorn.py --worker-class sanic.worker.GunicornWorker server:app` 就启动了。

为了方便对此服务的管理，可以使用 `Supervisor` 来对服务进行启动、停止，比如使用如下配置：

```
[program:demo]
command      = gunicorn -c config/gunicorn.py --worker-class
sanic.worker.GunicornWorker server:app
directory    = /your/path/
user         = root
process_name = %(program_name)s
autostart    = true
autorestart  = true
startsecs    = 3
redirect_stderr         = true
stdout_logfile_maxbytes = 500MB
stdout_logfile_backups  = 10
stdout_logfile          = ~/supervisor/demo.log
environment             = MODE="PRO"
```

最后，你需要对该服务(假设是一个网站)的”站点”进行配置，推荐使用Caddy服务器，Caddy是使用Go编写的Web服务器，它简单易用且支持自动化HTTPS，你只需按照官方文档编写好你自己的Caddyfile，比如目前的例子：

```
www.your.domain.com {
    proxy / 127.0.0.1:8001
    timeouts none
    gzip
}

your.domain.com {
    redir http://www.your.domain.com
}
```

在利用 `Supervisor` 守护一个Caddy的服务进程，至此，你的服务站点就搭建好了。

现在 `Docker` 的崛起，使得我们的部署方式也发生了改变，我们完全可以将上面编写的服务 `Docker` 化，然后构建自己的集群，一个服务器启

动一个服务节点，再启动一个镜像做负载均衡，岂不是美滋滋。

这个例子中我已经写了一个Dockerfile，你可以按照如下方式进行启动：

```
1. docker build -t demo:0.1 .
2. docker run -d -p 8001:8001 demo:0.1
```

我建议使用 `daocloud` 来体验一下，你可以关联自己主机，不一定非要用我这个例子中的服务镜像，你大可随意下载一个镜像

说明

代码见demo06

第二部分：源码及附录

Sanic源码阅读：基于0.1.2

Sanic源码阅读: 基于0.1.2

`Sanic`是一个可以使用`async/await`语法编写项目的异步非阻塞框架，它写法类似于`Flask`，但使用了异步特性，而且还使用`uvloop`作为事件循环，其底层使用的是`libuv`，从而使`Sanic`的速度优势更加明显。

本章，我将和大家一起看看`Sanic`里面的运行机制是怎样的，它的`Router Blueprint`等是如何实现的。

如果你有以下的需求:

- 想深入了解Sanic，迫切想知道它的运行机制
- 直接阅读源码，做一些定制
- 学习

将Sanic-0.1.2阅读完后的一些建议，我觉得你应该有以下基础再阅读源码才会理解地比较好:

- 理解装饰器，见附录
- 理解协程

Sanic-0.1.2 的核心文件如下:

```
1.  .
```

```
2.  ├── __init__.py
3.  ├── blueprints.py
4.  ├── config.py
5.  ├── exceptions.py
6.  ├── log.py
7.  ├── request.py
8.  ├── response.py
9.  ├── router.py
10. ├── sanic.py
11. ├── server.py
12. └── utils.py
```

通过运行下面的示例，这些文件都会被我们看到它的作用，拭目以待吧，为了方便诸位的理解，我已将我注解的一份 `Sanic` 代码上传到了 `github` ，见sanic_annotation。

simple_server.py

让我们从simple_server开始吧，代码如下：

```
1.
2.  from sanic_0_1_2.src import Sanic
3.  from sanic_0_1_2.src.response import json
4.
5.  app = Sanic(__name__)
6.
7.
8.  @app.route("/")
9.  async def test(request):
10.     return json({"test": True})
11.
12.
13. app.run(host="0.0.0.0", port=8000)
```

或许你直接把sanic_annotation项目直接clone到本地比较方便调试+理解：

```
1. git clone https://github.com/howie6879/sanic_annotation
2. cd sanic_annotation/sanic_0_1_2/examples/
```

那么，现在一切准备就绪，开始阅读吧。

前两行代码导入包：

- `Sanic`：构建一个 Sanic 服务必须要实例化的类
- `json`：以json格式返回结果，实际上是HTTPResponse类，根据实例化参数content_type的不同，构建不同的实例，如：
  - `text`：`content_type="text/plain; charset=utf-8"`
  - `html`：`content_type="text/html; charset=utf-8"`

实例化一个 `Sanic` 对象，`app = Sanic(__name__)`，可见sanic.py，我已经在这个文件里面做了一些注释，这里也详细说下 `Sanic` 类：

- route()：装饰器，构建uri和视图函数的映射关系，调用Router().add()方法
- exception()：装饰器，和上面差不多，不过针对的是错误处理类Handler
- middleware()：装饰器，针对中间件
- register_blueprint()：注册视图的函数，接受第一个参数是视图类 `blueprint`，再调用该类下的 `register` 方法实现将此蓝图下的 `route、exception、middleware` 统一注册到 `app.route、app.exception、app.exception`
- handle_request()：这是一个很重要的异步函数，当服务启动后，如果客户端发来一个有效的请求，会自动执行 `on_message_complete` 函数，该函数的目的是异步调用 `handle_request` 函数，`handle_request` 函数会回

调 `write_response` 函数， `write_response` 接受的参数是此uri请求对应的视图函数，比如上面demo中，如果客户端请求'/'，那么这里 `write_response` 就会接受 `json({"test": True})` ，然后进一步处理，再返回给客户端

- `run()`: Sanic服务的启动函数，必须执行，实际上会继续调用 `server.serve` 函数，详情下面会详细讲
- `stop()`: 终止服务

其实上面这部分介绍已经讲了Sanic基本的运行逻辑，如果你理解了，那下面的讲解对你来说是轻轻松松，如果不怎么明白，也不要紧，这是只是一个大体的介绍，跟着步骤来，也很容易理解，继续看代码：

```
1.  # 此处将路由 / 与视图函数 test 关联起来
2.  @app.route("/")
3.  async def test(request):
4.      return json({"test": True})
```

`app.route` ，上面介绍过，随着Sanic服务的启动而启动，可定义参数 `uri, methods`

目的是为 `url` 的 `path` 和视图函数对应起来，构建一对映射关系，本例中 `Sanic.router` 类下的 `Router.routes = []`

会增加一个名为 `Route` 的 `namedtuple` ，如下：

```
1.  [Route(handler=<function test at 0x10a0f6488>, methods=None,
    pattern=re.compile('^/$'), parameters=[])]
```

看到没， `uri '/'` 和视图函数 `test` 对应起来了，如果客户端请求 `'/'` ，当服务器监听到这个请求的时候， `handle_request` 可以通过参数中的 `request.url` 来找到视图函数 `test` 并且执行，随即生成视图

返回

那么这里 `write_response` 就会接受视图函数test返回的 `json({"test": True})`

说下 `Router` 类，这个类的目的就是添加和获取路由对应的视图函数，把它想象成 `dict` 或许更容易理解：

- `add(self, uri, methods, handler)`：添加一个映射关系到`self.routes`
- `get(self, request)`：获取`request.url`对应的视图函数

最后一行，`app.run(host="0.0.0.0", port=8000)`，Sanic 下的 `run` 函数，启动一个 `http server`，主要是启动 `run` 里面的 `serve` 函数，参数如下：

```
1.
2.  try:
3.      serve(
4.          host=host,
5.          port=port,
6.          debug=debug,
7.          # 服务开始后启动的函数
8.          after_start=after_start,
9.          # 在服务关闭前启动的函数
10.         before_stop=before_stop,
11.         # Sanic(__name__).handle_request()
12.         request_handler=self.handle_request,
13.         # 默认读取Config
14.         request_timeout=self.config.REQUEST_TIMEOUT,
15.         request_max_size=self.config.REQUEST_MAX_SIZE,
16.     )
17. except:
18.     pass
```

让我们将目光投向server.py，这也是Sanic框架的核心代码：

- serve(): 里面会创建一个TCP服务的协程，然后通过 `loop.run_forever()` 运行这个事件循环，以便接收客户端请求以及处理相关事件，每当一个新的客户端建立连接服务就会创建一个新的 `Protocol` 实例，接受请求与返回响应离不开其中的 `HttpProtocol` ，里面的函数支持接受数据、处理数据、执行视图函数、构建响应数据并返回给客户端
- HttpProtocol: `asyncio.Protocol` 的子类，用来处理与客户端的通信，我在server.py里写了对应的注释

至此，Sanic 服务启动了

不要小看这一个小小的demo，执行一下，竟然涉及到下面这么多个文件，让我们总结一下：

- sanic.py
- server.py
- router.py
- request.py
- response.py
- exceptions.py
- config.py
- log.py

除去 `__init__.py` ， `Sanic` 项目一共就10个文件，这个小demo不显山不露水地竟然用到了8个，虽然其中几个没有怎么用到，但也足够说明，你如果理解了这个demo， `Sanic` 的运行逻辑以及框架代码你已经了解地很深入了

blueprints.py

这个例子看完，我们就能轻易地明白什么是 `blueprints` ，以及 `blueprints` 的运行方式，代码如下：

```
1.
2.  from sanic_0_1_2.src import Sanic
3.  # 引入Blueprint
4.  from sanic_0_1_2.src import Blueprint
5.  from sanic_0_1_2.src.response import json, text
6.
7.  app = Sanic(__name__)
8.  blueprint = Blueprint('name', url_prefix='/my_blueprint')
9.  blueprint2 = Blueprint('name2', url_prefix='/my_blueprint2')
10.
11.
12. @blueprint.route('/foo')
13. async def foo(request):
14.     return json({'msg': 'hi from blueprint'})
15.
16.
17. @blueprint2.route('/foo')
18. async def foo2(request):
19.     return json({'msg': 'hi from blueprint2'})
20.
21.
22. app.register_blueprint(blueprint)
23. app.register_blueprint(blueprint2)
24.
25. app.run(host="0.0.0.0", port=8000, debug=True)
```

让我们从这两行开始：

```
1.
2.  blueprint = Blueprint('name', url_prefix='/my_blueprint')
3.  blueprint2 = Blueprint('name2', url_prefix='/my_blueprint2')
```

显然， `blueprint` 以及 `blueprint2` 是 `Blueprint` 根据不同的参数生成

的不同的实例对象，接下来要干嘛？没错，分析blueprints.py：

- BlueprintSetup: 蓝图注册类
  - add_route: 添加路由到app
  - add_exception: 添加对应抛出的错误到app
  - add_middleware: 添加中间件到app
- Blueprint: 蓝图类，接收两个参数: name(蓝图名称) url_prefix 该蓝图的url前缀
  - route: 路由装饰器，将会生成一个匿名函数到self.deferred_functions列表里稍后一起处理注册到app里
  - middleware: 同上
  - exception: 同上
  - record: 注册一个回调函数到self.deferred_functions列表里面，
  - make_setup_state: 实例化BlueprintSetup
  - register: 注册视图，实际就是注册route、middleware、exception到app，此时会利用make*setup_state*返回的*BlueprintSetup*示例进行对于的*add**一系列操作，相当于Sanic().route()效果

请看下 `route` 和 `register` 函数，然后再看下面的代码：

```
1.  # 生成一个匿名函数到self.deferred_functions列表里 包含三个参数 handler(foo), uri, methods
2.  @blueprint.route('/foo')
3.  async def foo(request):
4.      return json({'msg': 'hi from blueprint'})
5.
6.
7.  @blueprint2.route('/foo')
8.  async def foo2(request):
```

```
    return json({'msg': 'hi from blueprint2'})

# 上一个例子说过这个函数，Sanic().register_blueprint() 注册蓝图
app.register_blueprint(blueprint)
app.register_blueprint(blueprint2)
```

怎么样，现在来看，是不是很轻松，这一行 `app.run(host="0.0.0.0", port=8000, debug=True)` 服务启动代码不用多说吧？

总结

看到这里，相信你已经完全理解了 `Sanic` 的运行机制，虽然还有 `middleware&exception` 的注册以及调用机制没讲，但这和 `route` 的运行机制一样，如果你懂了 `route` 那么这两个也很简单。

如果诸位一遍没怎么看明白，这里我建议可以多看几遍，多结合编辑器 `Debug` 下源码，坚持下来，会发下 `Sanic` 真的很简单，当然，这只是第一个小版本的 `Sanic` ，和目前的版本相比，不论是代码结构的复杂程度以及功能对比，都有很大差距，毕竟， `Sanic` 一直在开源工作者的努力下，慢慢成长。

本人技术微末，若有错误，请指出，不胜感激.

- 注解地址: sanic_annotation
- 博客地址: http://blog.howie6879.cn/post/31/

附录：关于装饰器

附录：关于装饰器

认识装饰器

在python中，对于一个函数，若想在其运行前后做点什么，那么装饰器是再好不过的选择，这种语法在一些项目中十分常见，是Python语言的黑魔法，用处颇多，话不多说，让我们看一下代码：

```
1.
2. #!/usr/bin/env
3. # -*-coding:utf-8-*-
4. # script: 01.py
5. __author__ = 'howie'
6. from functools import wraps
7. def decorator(func):
8.     @wraps(func)
9.     def wrapper(*args, **kwargs):
10.         print("%s was called" % func.__name__)
11.         func(*args, **kwargs)
12.     return wrapper
13. @decorator
14. def hello(name="howie"):
15.     print("Hello %s!" % name)
16. hello()
```

```
1.
```

```
outputs:
hello was called
Hello howie!
```

这段代码，初看之下，确实不是很理解，接下来一步一步分析，看看装饰器到底是怎么工作的。

装饰器原理

在python中，方法允许作为参数传递，想在某个函数执行前后加点料，也可以这样简单实现。

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-1.py
__author__ = 'howie'
def decorator(func):
    print("%s was called" % func.__name__)
    func()
def hello(name="howie"):
    print("Hello %s!" % name)
decorator(hello)
```

由此，上面代码也可以这样写：

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-2.py
__author__ = 'howie'
def decorator(func):
    print("%s was called" % func.__name__)
    func()
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello
```

两段代码执行后：

```
outputs: shell
hello was called
Hello howie!
```

表面上看来，`02-2.py`代码看起来也可以很好地执行啊，可请注意，在末尾处，`hello`只是函数名称，它并不能被调用，若执行`hello()`，就会报`TypeError: 'NoneType' object is not callable`对象不能调用的错误，这是自然，因为在`decorator`中`func()`直接将传入的函数实例化了，有人会想，那如果这样改呢？

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-3.py
__author__ = 'howie'
def decorator(func):
    print("%s was called" % func.__name__)
    return func
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello()
```

确实，这样改是可以，可有没有想过，若想在函数执行结束后加点装饰呢？这样便行不通了，可能又有人会想，若这样改呢？

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-4.py
__author__ = 'howie'
def decorator(func):
    print("%s was called" % func.__name__)
    func()
    return bye
```

```
def bye():
    print("bye~")
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello()
```

这样写看起来，恩，怎么说呢，总有种没有意义的感觉，不如直接将在外部的函数放进 `decorator` 中，如下：

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-5.py
__author__ = 'howie'
def decorator(func):
    def wrapper():
      print("%s was called" % func.__name__)
      func()
      print("bye~")
    return wrapper
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello()
```

执行：

```
outputs: shell
hello was called
Hello howie!
bye~
```

怎么样，输出的结果是不是符合要求，其实简单来看的话，可以这样理解 `hello()==decorator(hello)()==wrapper()` ，最后其实就是执行 `wrapper()` 函数而已，事实就是如此的简单，不妨来验证一下：

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-6.py
__author__ = 'howie'
def decorator(func):
    def wrapper():
      print("%s was called" % func.__name__)
      func()
      print("bye~")
    return wrapper
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello()
print(hello.__name__)
```

```
outputs: shell
hello was called
Hello howie!
bye~
wrapper
```

果然就是执行了wrapper函数，解决问题的同时也会出现新的问题，那便是代码中本来定义的hello函数岂不是被wrapper函数覆盖了，又该如何解决这个问题呢？这时候 `functions.wraps` 就可以登场了，代码如下：

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 02-7.py
__author__ = 'howie'
from functools import wraps
def decorator(func):
    @wraps(func)
    def wrapper():
      print("%s was called" % func.__name__)
```

```
        func()
        print("bye~")
    return wrapper
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello()
print(hello.__name__)
```

执行代码：

```
outputs: shell
hello was called
Hello howie!
bye~
hello
```

`functions.wraps` 作用是不是一目了然哈~到了这一步，再看01.py的代码，是不是代码结构清晰明了，只不过多了个参数~

```
#!/usr/bin/env
# -*-coding:utf-8-*-
# script: 01.py
__author__ = 'howie'
from functools import wraps
def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("%s was called" % func.__name__)
        func(*args, **kwargs)
    return wrapper
@decorator
def hello(name="howie"):
    print("Hello %s!" % name)
hello('world')
```

猜都猜得到执行后输出什么了。

结语

只要了解装饰器原理，不管是带参数的装饰器，还是装饰器类，都是小菜一碟。