Flask+Vue前后端分离与docker部署

2020年12月13日 11:31

序

在这之前，已经完成了后端应用部分利用docker在服务端的完整部署。但是在之后的项目模块整合工作推进过程中，由于之前在忙着做后端与中间层算法部分的推进，对于前端vue的生态了解一直很有限，一直以为其只是提供了后端数据交互的框架，整合的时候直接用静态文件做整合就好。

但是事实并非如此。

那么前后端是如何交互的，它们之间如何连接？连接之后docker又将作何配置将它们部署？前后端分离式开发的完整工作流与我之前的认识有何不同？这是我这个阶段主要需要解决的问题。

概述

Vue项目部署

目前而言，我对Vue的理解只是从后端与它协同工作的角度去理解。具体的开发过程我尚不了解。

安装nodejs（提供npm管理资源）-> npm安装cnpm（npm的国内镜像） -> 打包好的Vue应用 -> cnpm安装必要包 -> 生成需要的库文件 -> 生成目标静态文件 -> 部署入nginx根目录

Nginx/Dockerfile再配置

nginx配置文件之前的两次部署已经配置过两次，这次的情况同样发生了变化，需要重新配置。

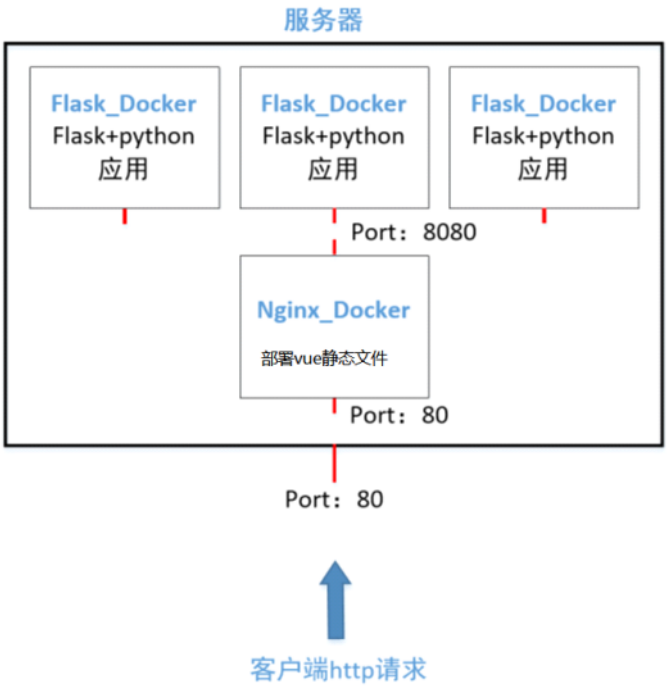
flask应用加入CORS处理同源请求

加入flask_cors包。

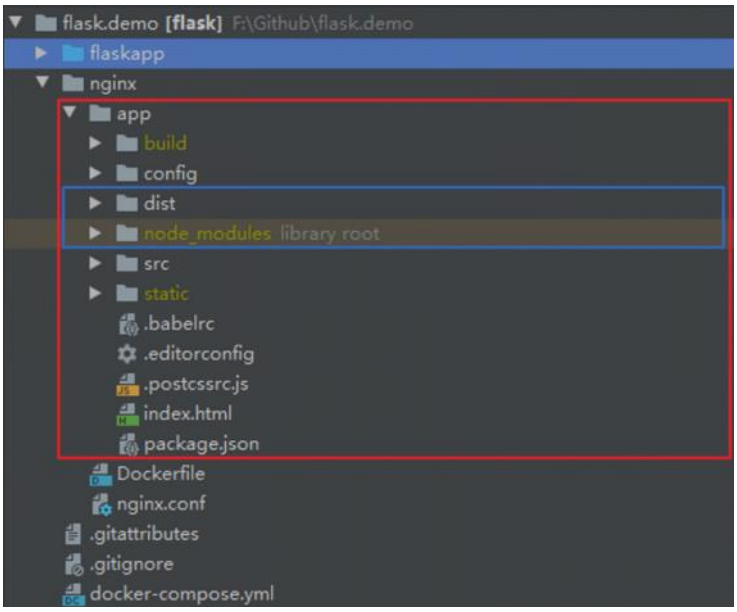
系统结构

系统的结构上变化并不大，基本可以沿用上一次的结构图，不过需要稍作修改。

与上次不同的是，现在客户端的请求可以被vue的页面直接响应，而不需要直接去请求flask应用返回页面。在页面需要向后台请求数据或是提交数据到后台时才需要flask应用响应。



文件结构



Github:https://github.com/vanot313/flask.demo

flask应用整体配置上并无变化，主要变化在于nginx的结构变化。

红框内即为完整的Vue前端应用。不过这只是前端同学给我提供的早期版本，不代表标准的结构。

蓝框中，/dist文件为执行命令后生成的目标静态文件，/node_modules为生成的运行环境（即必要的库文件）。/dist文件夹内的静态文件为真正部署到nginx根目录的内容。

Vue

工作模式

Vue + flask 的开发模式是一种前后端分离的开发模式。那么什么是前后端分离？就我目前的理解而言，就是前端与后端可以分别作为独立的应用开发并作独立的部署。

前端应用做好前端相关的所有工作，包括响应接受请求，处理前端页面的跳转路由，并在需要后端数据的时候向后

台请求数据接口。相比于传统的开发模式，不再仅限于编写与美化界面。

后端应用也做好后端相关的所有工作，包括处理前端发送的数据并返回前端需要的数据接口，访问数据库等工作。相比以前，省去了配置前端路由的麻烦。

这种解耦化的工作模式可以让前后端项目互不干扰地更新与部署。不过这不意味着前后端成为两个完全独立的黑盒，前后端的程序员仍然需要通过讨论来确定数据接口。

打包过程

首先，我们需要下载安装node.js。简单地找到windows安装包并下载安装即可，安装过程中基本一路点下去就行了。

接下来要做的事情与python和pip做的事情很像。国内的npm下载速度很慢，而阿里云给我们提供了cnpm，它是一个完整的npm镜像。那么安装cnpm。

```
cnpm  npm install -g cnpm -registry=https://registry.npm.taobao.org
```

就像python项目的运行需要环境，vue项目也不例外。需要安装axios（尚未深入了解，只知道其提供了类似于ajax的功能）等额外的包来提供支持。

```
cnpm install axios --save
cnpm i element-ui -S
```

这些包也许不够用，不过在cnpm install报错的时候你会知道你还缺少哪些包。

进入到vue项目的目录下。并生成库文件（前文提到的/node_modules）。

```
cd path/to/project/
cnpm install
```

就是在这里，你很有可能遭遇报错，不过错误内容大概率能够看懂，无非是缺少了哪些包，接着安装上即可。

生成静态文件（前文提到的/disk）。**前端工程师在目前的预定的工作流中应当在本地生成/disk完成后一同push至git，在部署时直接将生成的disk静态文件部署进入nginx根目录。**

```
cnpm run build
```

前端工程师测试时可以直接通过cnpm run dev指令在本地测试，不过这不在本篇博客的讨论范围之内。

完成了静态文件的生成，vue项目的打包即完成了。

nginx再配置

配置文件

```
server{
     listen 80;
     # flask
     location / data{
          Include uwsgi_params;
          uwsgi_pass flask:5000;
     }
     # vue
     location / {
          root /usr/share/nginx/html;
          index index.html index.htm;
     }
}
```

与之前不同的是，'/'访问根目录被vue项目占据了，flask应用便需要一个额外的目录来作为数据接口目录。需要注意的是配置的前后关系，因为nginx的遍历是从前到后的，因此需要将转发子集排列在前面以防止错误的转发。

并不确定这是标准的配置模式，但是它有效

Dockerfile再配置（nginx）

配置文件

```
FROM nginx
# config
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/
# disk
COPY app/dist /usr/share/nginx/html
```

因为多了部署disk静态文件的操作，因此需要更改nginx的dockerfile文件来更新部署操作。具体是由最后一行指令完成文件的拷贝。

flask_cors 处理同源请求

触发情景

浏览器从一个域名的网页去请求另一个域名的资源时，域名、端口、协议任一不同。如在页面强用ajax获取其他异步请求向其他网络地址发起请求。只要协议、域名、端口有任何一个不同，都被当作是不同的域，之间的请求就是跨域操作 。

Flask做的处理

```
app = Flask(__name__)
CORS(app, supports_credentials=True)
```

目前只是停留在用的层面上，对其中的运作原理尚不清楚，用CORS函数加工一边Flask应用即可。

交互情景

vue请求数据接口

通过axios完成请求

```
import axios from 'axios';
import router from '../router'
axios.defaults.timeout = 10000; //10秒 超时时间
axios.defaults.withCredentials = true; //允许跨域
/*Content-type 响应头*/
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded;charset=UTF-8';
/*基础url*/
axios.defaults.baseURL = "http://49.235.73.129/data";
```

完成必要配置。最后一行baseURL为请求的数据源。直接通过域名访问到flask应用的数据接口。具体的请求操作等待前端同学来做更具体的解答吧（期待）！

这里的数据源的获取方式并不清楚是否符合规范，还是说应当采用docker容器间通信的方案呢?

flask提供的数据接口

```
# 注册蓝图blueprint对象
app.register_blueprint(data, url_prefix="/data")
```

```
# 创建一个蓝图对象
data = Blueprint("data", __name__)
```

```python
# 注册路由
@data.route("/", methods=['GET'])
def index():
    context = {}
    result = Account.query.all()
    # context['result'] = result
    title = 'it works'
    context['title'] = title
    return jsonify(context)
    # return render_template("index.html", **context)
```

完成基本路由配置（对应前面请求的数据接口路径），并在函数中简单返回一个json数据。

简单的demo

这是个登陆页面（假的）

事实上他只完成了点击‘登录’按钮后向后台应用请求一个json数据并通过console打印出来而已。就是前面给出的交互情景完成的内容。

末

该解决的问题应该是得到全部解决了。具体工作流的分析可能可以单独地做一篇博客。有些踩到的坑并没有在这里罗列出来，但是具体的解决方案博客已经做了保存。

对于docker容器的架构可能还能做进一步优化，以及容器间通信的相关内容似乎尚有优化的余地。不过这一次的迭代比较完整地建立了前后端分离的基础设施，先解决0到1的问题。

中间层的推进也有了眉目了，接下来先把后台各模块做一个整合。