# 系统设计文档

**文档名称：单页应用搜索引擎优化系统可行性分析报告**

**项目名称：单页应用搜索引擎优化系统**

**编写人：赖信涛**

1.1 系统的概要设计

1.1.1 系统设计

本系统为单页应用的SEO优化而设计，整个系统主要分为两部分。第一部分是爬虫引擎，主要负责从目标网站抓取页面内容，将爬取的任务信息存到数据库。第二部分是网页服务器部分，主要负责给搜索引擎返还真实的页面内容。

为了达到“部署快捷”的原则，整个系统运行在docker之上，以镜像的形式发布，做到了一键安装。总体的系统架构如图1-1所示。

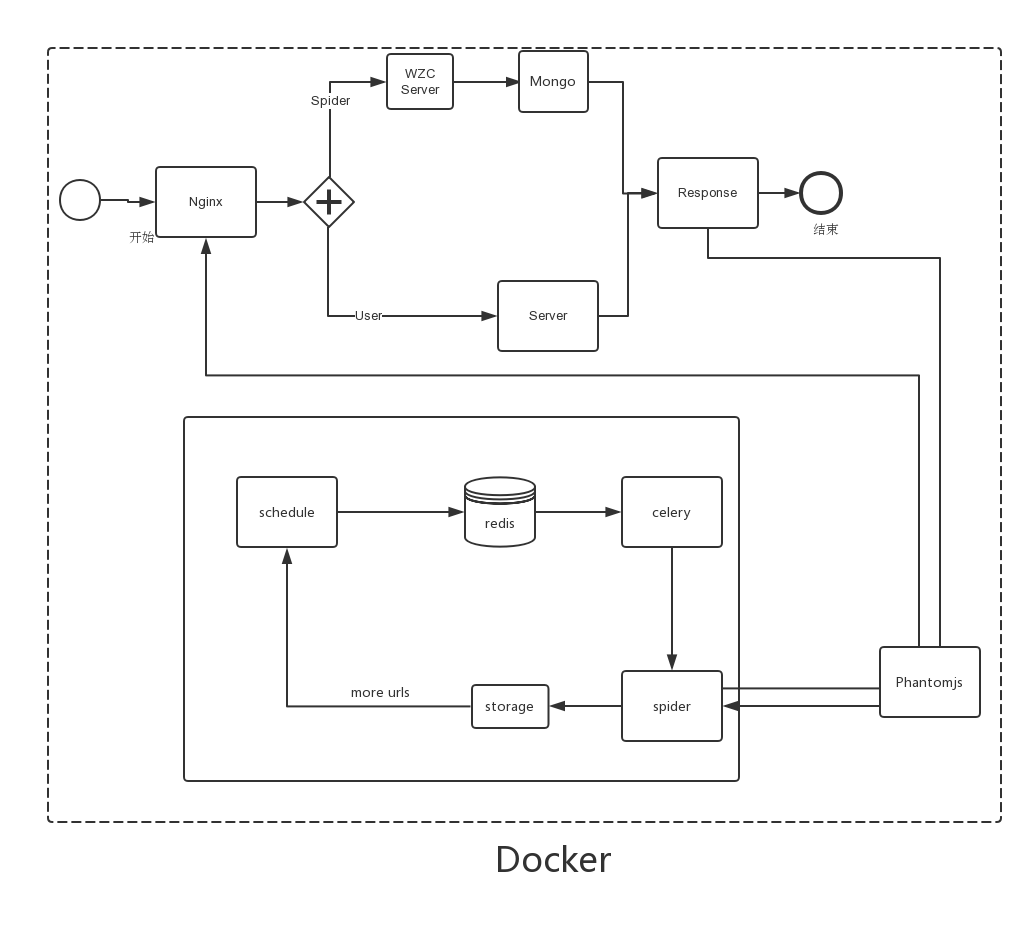


图1-1 系统总体架构图

1.1.1 系统代码的组织

在代码的组织中，将项目的代码当道wzc目录下，文档，及其他资料都放到单独的目录下。代码组织的部分，所有模块都作为文件夹存放（也就是Python的模块）。系统所有代码如图1-2所示：

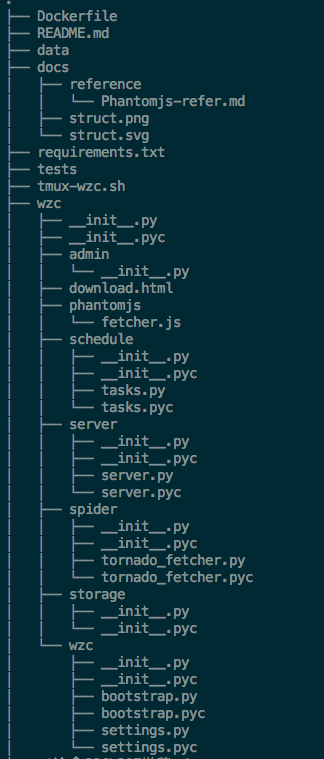


图1-2 系统代码的组织

其中，DockerFile是Dock制作镜像所使用的脚本，requirements.txt中注明了本系统所用到的所有pip依赖。Tmux-wzc.sh是快速启动系统的脚本。Wzc文件夹下的每一个文件夹与本系统的模块所对应。

系统需要爬虫模块、存储模块，JavaScript执行模块、web服务器模块一共四个主要模块，另外需要对容器以及如何部署进行设计，还需要监控系统的运行状况，下面对这七个模块分别进行讨论。

由于系统组件结构复杂，涉及三种编程语言，为了松耦合以及让各模块之间配合方便，采用端口的形式相互协调。其中，用作任务队列的缓存Redis使用6379端口，Phantomjs模块使用12306端口，web服务器监听80端口。端口的部署情况如图1-2所示。

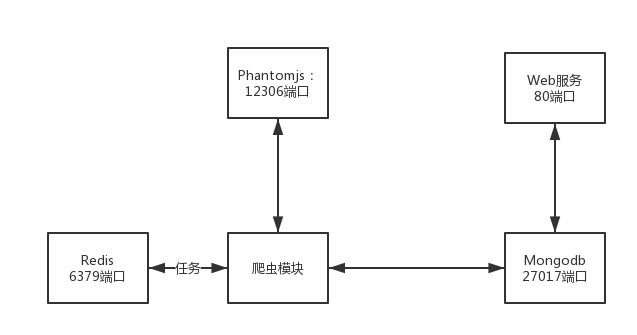


图1-2 组件的端口

1.1.1爬虫模块

本模块主要负责运行爬虫，调度爬虫任务，以及提供任务队列的监控和管理，这个模块由项目的启动入口拉起主要的调度函数，使用这个函数部署抓取任务，抓取网站的页面，通过js执行模块执行页面的js内容。然后将执行之后的页面通过网页的存储模块存储下来。之后，需要查找本页面中存在的链接，将这些链接以任务的形式放到队列中（即保存到redis中）。提供可视化的任务队列管理页面。

爬虫需要通过通过12306端口（在系统中可以更改端口）与phantomjs通讯，以post请求的方式发给phantomjs页面解析的请求，由phantomjs执行解析页面的任务，然后作为response返回。

1.1.2页面的JavaScript执行模块

本模块主要负责解析和执行页面内的JavaScript，是系统的核心模块之一。首先，需要在本机监听一个端口，对这个端口收到的post请求认为是一个任务，收到任务之后完成解析然后返回结果。

随后设置网页的meta信息，例如载入时间，状态，请求的headers等。解析析页面上的JavaScript脚本（包括Ajax请求）。如果页面内写了专门优化SEO的函数document-start，还需要执行这些函数。

1.1.3网页的存储模块

本模块主要负责将爬虫获得的页面存储到本机中。主要使用文件系统配合Mongo数据库来进行存储，主要的目标是将页面信息存入数据库，然后将页面内容作为文件存储。

除此之外，给出一个URL路径，能快速索引到这个页面的收录状态，最后更新时间等。

由于系统使用Python开发，所以使用pymongo包进行对mongo数据库的驱动。将其封装一下，作为系统的一个单独模块，能够自主处理爬虫发来的消息，自动保存。

在章节1.3.2中，提到存储要面临的一个难点是比较两个页面是否有不同。如果对字符串进行比较将耗费大量的内存和计算资源。根据1.3.2章提出的方法，本系统采用文件存储的方式，使用MD5值作为文件名，mongo数据库中保存一个MD5字段，这样mongo数据库的信息就能和页面的内容关联起来了。在保证查询快的同时，又能用很少的资源就完成页面的对比。

1.1.4响应请求的web服务器模块

本模块的角色是一个简单的Web服务器，职责就是接受请求，到数据库中获取相关页面的信息，然后到文件系统中找到相关的页面，返回一个请求的响应给发送者。本模块所要完成的一个主要目的是接受一个请求，能正确返回静态的结果。

搜索引擎请求的页面并不是一定已经被系统收录，如果页面尚未被收录，Web服务模块将马上开启一个高优先级的任务优先处理此页面。如果任务失败，说明此页面存在问题，则记录下错误，返回一个错误信息。对所有请求进行记录，以便于进行SEO优化以及监控使用。

1.1.5 使用docker部署

本系统的模块比较多，而且模块是以服务的形式运行在Linux上，以端口的形式进行通讯，所以部署复杂。

所以本系统采用容器的方式进行发布和部署，可以做到与Linux 的发行版无关，与环境无关，一键拉取到镜像即可以完成部署。

Docker容器的底层是ubuntu 16.04 LTS，本系统所有的模块都基于此进行安装，并且仅安装与系统有关的组件和服务，尽量保持镜像的体积小[10]。

为了保证容器的整洁，本系统采用使用Dockerfile的方式制作镜像，这样可以保持对镜像的改动最小，并完成所有的功能。

1.1.6 在服务器上的部署

上面提到，本系统使用容器部署。所以实际生产服务器部署的时候，会非常简单。首先，需要从Docker Hub（本系统采用官方Docker仓库进行托管和维护）拉取最新的镜像，然后在启动Docker的同时，将一个端口映射到容器内的Web服务器地址即可[11]。

系统像作为开源项目发布，所以镜像仓库和代码仓库都选择了官方的仓库托管，用开放的形式发布。

1.1.7 监控服务状态

由于系统的很多组件都是一个个的进程运行在服务器上，虽然程序内部都对可能发生的异常进行了处理，但是尽管这样，还是可能发生预期之外的异常，这可能终止正在运行的进程。如果某一个进程没有健康地运行，就必然会拖累到整个系统。

所以，系统必须有一定的监控和重启机制，在某个进程终止之后，能自动重启。如果在一定时间内重启失败次数超过了设定值，那么可能就是系统环境或者程序存在问题，便放弃重启。

经过调研，发现supervisor是用Python编写的，带有web管理界面，在web页面便可以查看进程状态，进行重启等操作。符合本项目的需求，所以使用supervisor来进行项目管理。

2详细设计

本章作为系统的实现部分，将要讲述各个模块的代码层面。通过对代码的分析，介绍各个模块的难点和解决方法。

2.1 设计平台搭建

2.1.1 开源工具选择

数据库使用MongoDB。因为数据库要存储的大量有关页面信息的内容，结构化比较差，关系简单。所以在这里选择了NoSQL数据库。管理方面使用官方的mongoDB命令行。

编辑器使用Vim。Vim编辑器支持VimScript，可以实现例如宏这样的高级功能。本系统需要用到各种类型的脚本，例如DockerFile，JavaScript，Python，shell等，所以使用纯文本编辑器，可以兼容编辑各种文档。shell使用zsh，因为zsh自动补全功能强大，为开发带来了不少便利。终端使用tmux，因为本系统涉及各种服务以及后台程序，不可避免地要运行很多终端窗口，用tmux这样对多窗口友好的B/S架构进行开发再好不过。版本控制使用Git[12]。本系统在Github进行开源，使用分布式的版本控制工具控制版本，方便其他开发者参与到本项目中。

2.1.1 设计平台配置

本系统大多数工作都在终端完成，所以对系统并没有特殊的要求，只要支持Linux的shell即可，本系统设计过程中使用的工具都可以通过Linux系统的终端工具编译安装。硬件配置如表5-1所示：

表 5-1 开发环境配置

|  |  |
| --- | --- |
| 开发环境要求 | |
| CPU | 主频2.5Ghz以上 |
| 内存 | 至少4GB |
| 硬盘空间 | 1G左右 |

2.1.3 生产环境服务器的要求

本系统的模块以服务的模式运行，各个模块之间使用端口进行网络通讯，所以耦合行非常低，可以分开部署。故本系统对硬件的要求很低，取决于目标网站的大小，对于硬件，没有明确的配置要求。如果有必要，可以使用集群的方式灵活部署。

软件方面，只要服务器是Linux内核并且安装了Docker即可，除此之外没有其他要求。

2.2 爬虫模块

这是系统的核心模块。爬虫模块的作用是收录本站希望被搜索引擎检索到的页面，供Web服务器使用。

2.2.1任务队列

由于爬虫抓取页面、解析页面、保存页面需要很长的时间（取决于服务器配置和网络带宽，一般在10秒左右），所以使用单线程爬虫是不合理的，本模块引入了任务队列的模式。队列的架构如图5-1所示。一个任务队列系统可以包含多个工作机，以便实现高可用和可水平扩展。其中有一台机器负责调度任务队列，其他的工作机从队列中取出任务执行。

用任务队列的方式可以减少系统阻塞的时间，将任务分时处理。对于耗时较多的任务，就可以相应地增加这部分内容的工作机，专门优化这部分的处理。引入任务队列，提高了系统解决瓶颈的能力。

任务队列的工作流程，如图5-1所示。

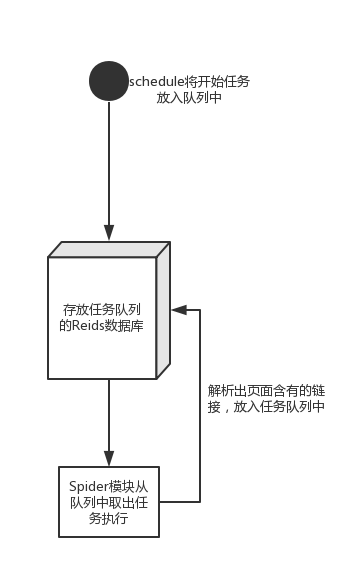


图2-1 任务队列的工作流程

本模块的任务队列架构使用了Celery， 选择了Redis作为Broker。主要原因是，Redis是运行在内存中的数据库，速度相比于以硬盘空间作为存储的数据库快的多。考虑到任务队列需要非常频繁的存储与读取，所以速度是我们考虑的主要因素。另外，本系统用到的组件非常多，为了降低复杂程度，使用了Reids，这样不仅可以将其作为任务队列，还可以作为缓存使用。

爬虫的任务队列部分主要负责两个任务：更新目标URL，即完成目标任务。；将更多的URL放入到任务队列中，即开启更多的任务。核心代码如下：

import logging  
  
from celery import Celery  
from wzc.spider import update\_url  
from wzc.wzc.settings import BASE\_URL, MAX\_RETRY  
  
wzc\_spider = Celery('wzc', broker='redis://localhost')  
logger = logging.getLogger(\_\_name\_\_)  
  
@wzc\_spider.task  
def update(url, retry\_left=MAX\_RETRY):  
 """url update task"""  
 if retry\_left < 0:  
 logger.error("no retry times left")  
 return  
 more\_urls = update\_url(url)  
 if more\_urls:  
 for url in more\_urls:  
 update.delay(BASE\_URL+url)  
 else:  
 retry\_left -= 1  
 update.delay(url, retry\_left)

上面的代码指定了一个名为“wzc”的任务队列实例。任务开始时，首先要判断一下这个任务执行的次数是否达到了重试上限。如果是，则说明这个任务存在问题，使用日志记录，任务直接结束。然后，调用爬虫模块的核心代码解析目标URL，爬虫模块会返回一个URL的列表。第二步，任务队列的worker会将这些URL重新放到任务队列中去。如果返回的URL列表为空，说明任务以不正常的状态结束，重试次数减一，重新把该任务放回队列中。在数据库中使用一个字段存储与上次下载的结果是否有变动，如果有变动，那么下次更新的时间减半，否则时间加倍。这样就可以使更新频繁的页面得到及时更新，不频繁的网页节省资源，工作机处理任务的流程如图5-2所示。

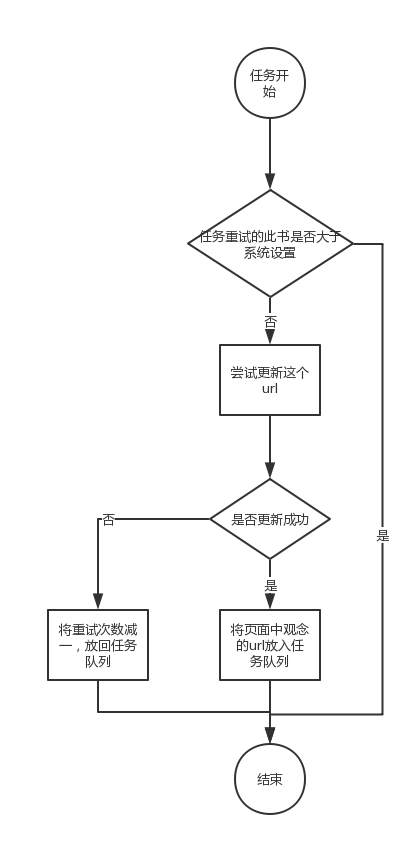


图2-2 爬虫更新一个页面的流程

2.2.2爬虫的核心模块

这是本系统的核心模块之一。当一个URL更新任务发给爬虫模块之后，需要先检查这个URL是否已经更新过，如果已经更新过，目前的内容是否已经过时了。如果没有过时，则直接结束任务，如果内容已经过时，则开启一个更新任务。

受网络、系统资源等因素影响，更新一个URL的任务并不是总是可以成功执行。如果任务失败，将会触发异常机制，报告异常，并将异常的页面信息存放如数据库。对于调用者，返回None，以进行安排任务重试。

爬虫支持同步和异步两种形式。如果是同步的模式，任务会以阻塞的形式执行，直到页面的处理结束，返回一个结果。如果是异步的模式，不等结果出来，进程就会结束，然后去处理下一个任务。等这个函数执行完之后，会触发回调函数，进行后续的处理。

异步是通过协程（coroutine）实现的。协程的概念应该是从进程和线程演变而来的，他们都是独立的执行一段代码，但是不同是线程比进程要轻量级，协程比线程还要轻量级。多线程在同一个进程中执行，而协程通常也是在一个线程当中执行[13]。协程与线程的关系，如图5-2所示。



图2-3 协程与线程的关系

综上，整个任务队列系统的工作流程如上所述。整个流程需要工作机和任务队列之间的服务器相互配合。工作机不断从任务队列中取出任务，处理之后，将更多的任务放入队列。如此，便形成一个不断在更新的镜像网站。这个过程可以用图5-4的工作流程图表示。

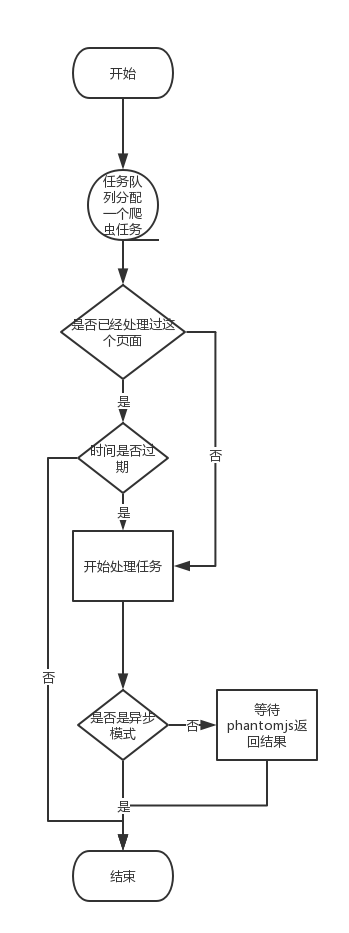


图2-4 任务队列和工作机的工作流程

1. 使用分布式集群爬虫

由于爬虫所消耗的资源非常高，在本系统中，爬虫模块是最占用系统资源（带宽，内存，CPU等）的一部分。对于大型的网站来说，解析任务会比较多，为了能更快速地索引页面，使网站内容更新的更即使，也可以使用分布式的爬虫。

使用分布式的爬虫，需要保持Redis和Mongo的唯一。否则，任务队列调度会失控，导致同一个任务被多次执行。

使用的方法非常简单，只需要在设置里指明所使用的Broker的地址，以及mongo的地址便可。作为Broker的机器就是Master，其余的机器从Master里面取任务执行，将可能存在的更多的任务放回Master主机里。这个架构如图5-5所示。

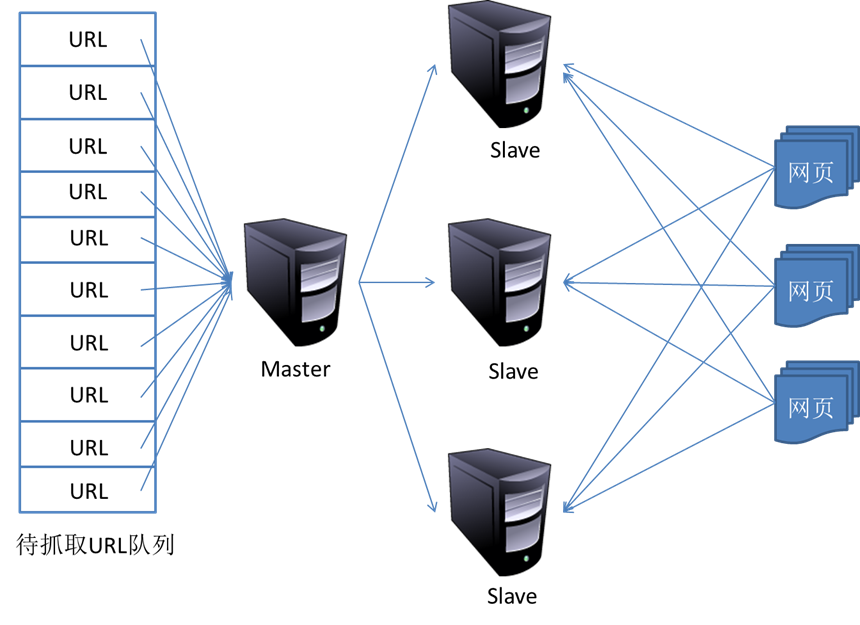


图2-5 分布式的爬虫集群架构

2.2.3对任务队列的监控

爬虫是本系统最主要的部分，所以对系统的维护工作主要是爬虫的运行情况上。本系统提供了一个可以监控爬虫运行情况的Web界面。模式监听5555端口，通过这个端口呈现的信息，可以直观地查看集群的健康值，爬虫任务信息等。

从首页可以看到，正在运行的worker以及他们的状态，这些worker完成的任务，失败的任务等等。还可以快速重启、关闭指定的worker。首页的展示如图5-6所示。这个页面有一个展示所有工作机的表格，可以显示工作机的状态，当前任务的状态等。

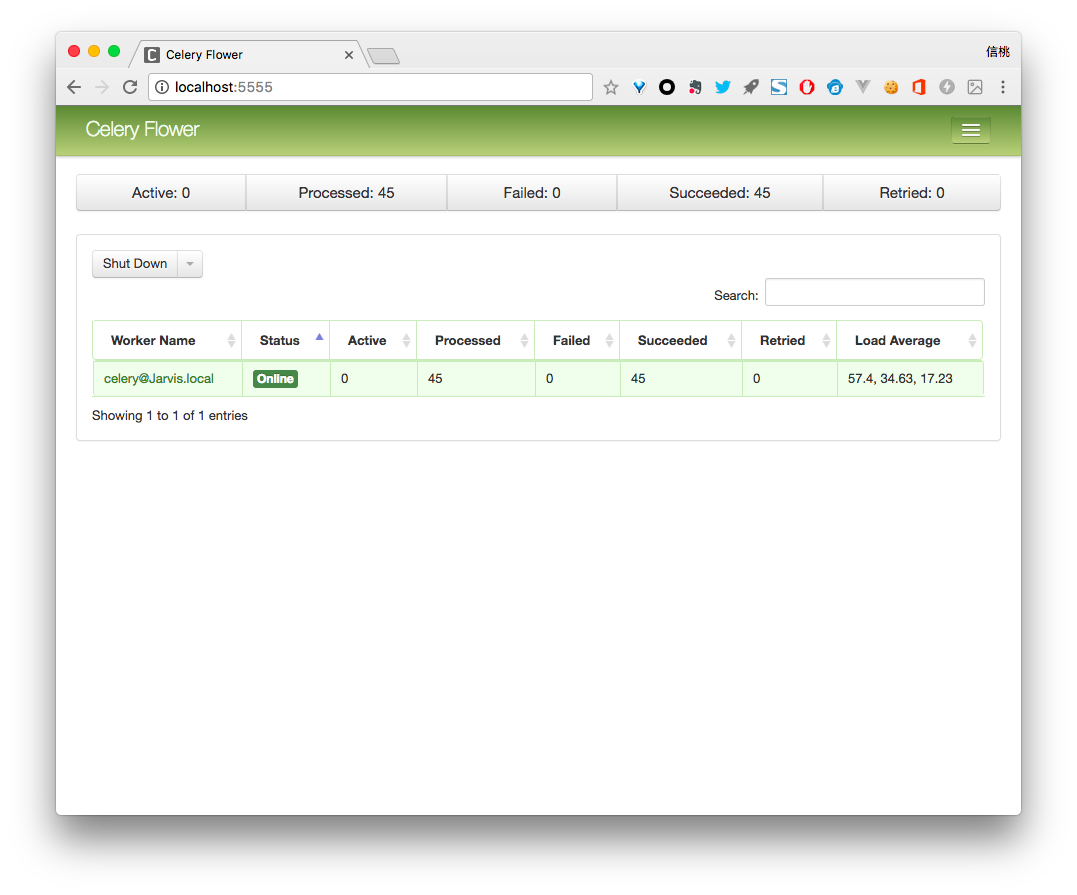


图2-6 监控界面的首页

在监控页面，可以看到各个worker工作状态的图表。从这里可以直观地看到这些worker前后一段时间的执行情况，任务花费的时间。从而得知他们的健康程度。监控的表格页面如图5-7所示，第一幅图展示了执行任务成功的数量，第二幅图展示了任务失败的数量，从这个表格中可以监控集群的健康程度，代码中是否存在错误等。第三幅图展示了完成一个任务所需要的时间，第四幅图展示了任务队列中正在等待被执行的任务。由此可以看出集群的性能。如果执行的时间过长，那么可能是页面的性能有问题，网页的JavaScript代码还有优化的空间。如果任务队列中在等待的任务不断增加，说明集群的负载压力过大，处理任务比较吃力。这时候如果要保证网站更新的及时，应该升级集群的配置。

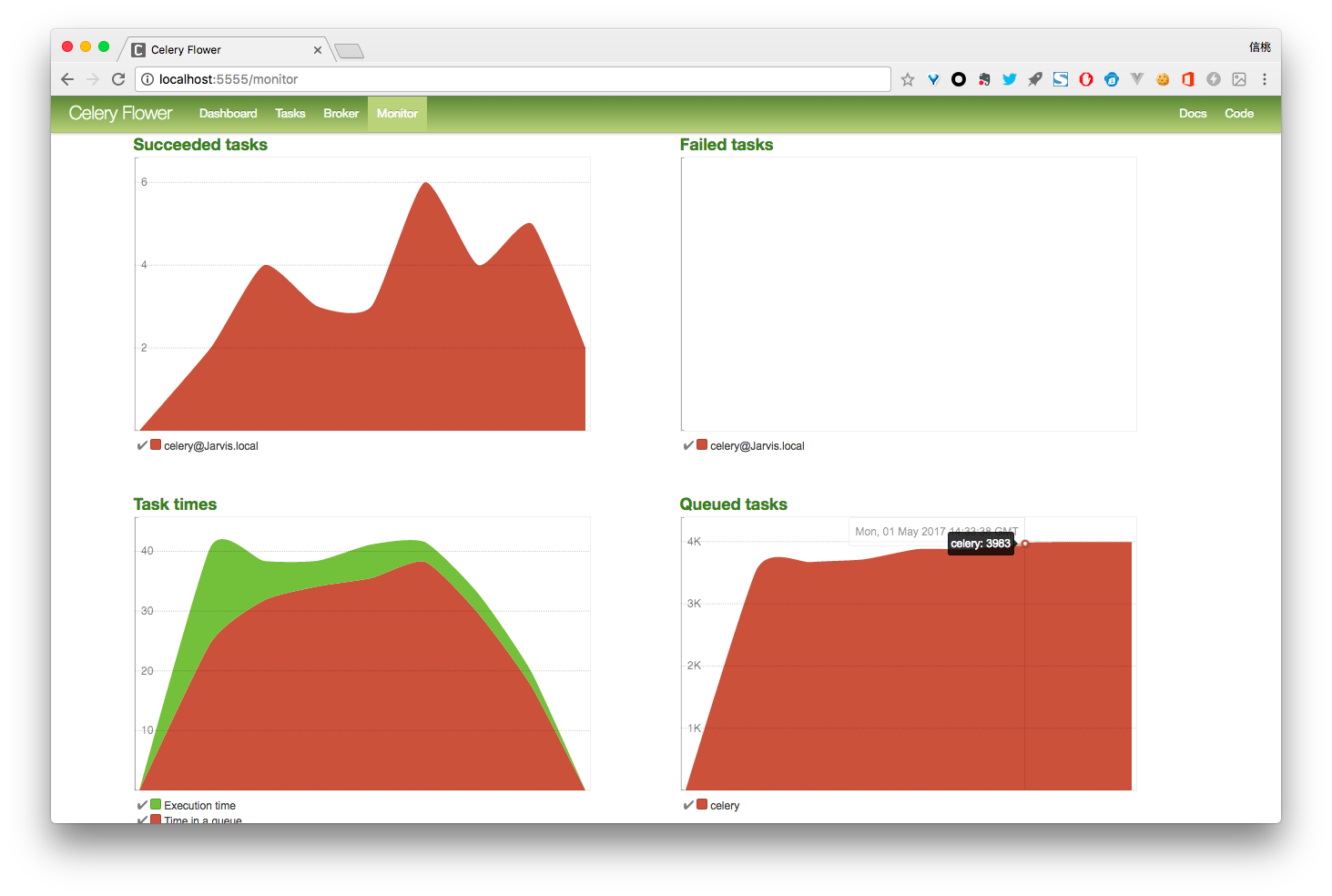


图2-7 任务队列的运行情况监控界面

在tasks页面，也可看到每一个执行过的任务状态，以及执行信息。想知道某一个任务的详细信息，可以在这里查看。任务详情页面的界面如图5-8所示。这个页面可以选择每页展示的任务数量，提供了搜索功能。在列表里面，提供了一个查看一个任务的状态所需要的详细信息。包括任务的名字（NAME），唯一标志符（UUID），状态（STATUS），参数列表（ARGS），字典参数（KWARGS），返回结果（RESULT），接受者（RECEIVES），开始时间（STARTED），执行它的工作机（WORKER）。

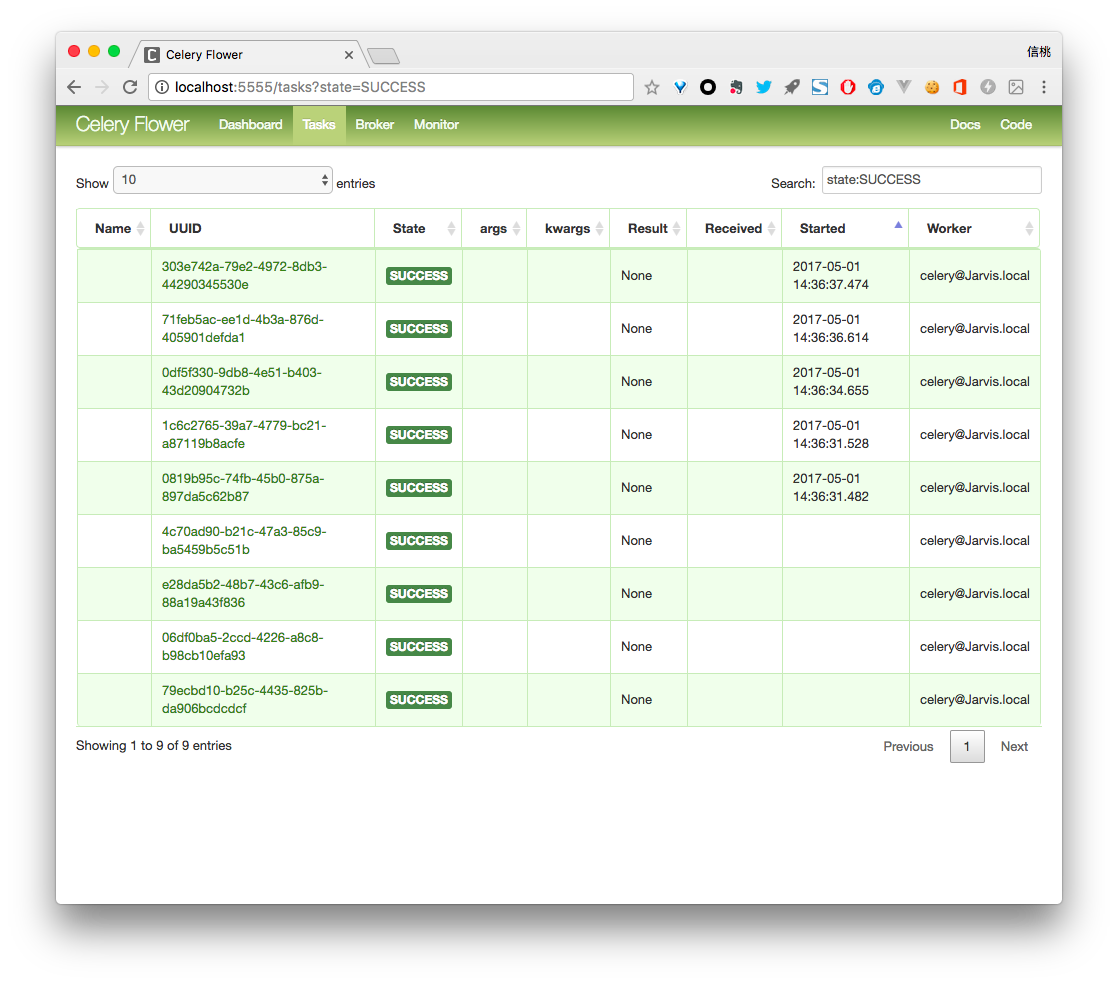


图2-8 任务详情监控界面

2.3 Phantomjs模块

2.3.1 phantomjs模块的实现

本模块主要负责处理爬虫的请求，作为一个代理，去抓去页面并进行解析。由于本系统的目标是动态的单页应用，所以依赖于大量的AJAX请求，通常，要想得到一个完整的页面，需要发送多次请求，对页面的数据进行填充。

PhantomJS是一个可编程的无头浏览器。它有一个完整的浏览器内核,包括js解析引擎,渲染引擎,请求处理等,但是不包括显示和用户交互页面的浏览器。

本系统中有关Phantomjs的脚本使用JavaScript写成，使用Phantomjs的接口，定义了下载一个页面的时候触发的一系列事件。该模块以监听端口的方式运行，执行的流程图如图5-9所示。

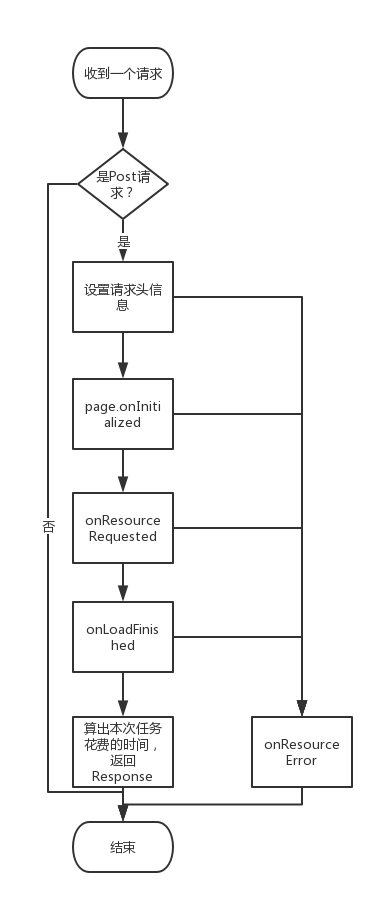


图2-9 PHANTOMJS执行的流程图

Phantomjs模块运行之后。通过监听一个端口（默认为12306）来与其他模块进行交互。爬虫将下载的文档发到这个端口，Phantomjs便开启一个任务，执行文档的JavaScript脚本，执行完成后，以一个字典返回。后面的操作，爬虫会调用存储模块去完成。

当收到请求的时候，Phantomjs模块会记录下当前的时间戳，请求处理完成，将要返回结果的时候，用当前系统的时间戳与之前记录的比较，得到整个任务花费的时间，放到字典中作为其中的一个字段放回。后面会存放到数据库中，可以供后序的研究和监控使用。

return {  
 orig\_url: fetch.url,  
 status\_code: first\_response.status || 599,  
 error: first\_response.errorString,  
 content: page.content,  
 headers: headers,  
 url: page.url,  
 cookies: cookies,  
 time: (Date.now() - start\_time) / 1000,  
 js\_script\_result: script\_result,  
 save: fetch.save  
}

返回数据的格式，如代码所示，其中headers是响应的头部，java\_script\_result是执行文档中的JavaScript脚本获得的结果。如果本次处理请求的过程中遇到了错误，则error就会记录下错误信息，status\_code会记录下错误代码，在结果中返回。爬虫会判断本次请求是否成功。如果不成功将会试图获得错误的代码和错误信息，并开启一个重试任务。

Phantomjs中对任务最大执行时间进行了限制。如果不限制，假设有一个任务发生了阻塞，那么Phantomjs将会永远被卡在这个地方，后面的任务可能永远停在了等待状态。

对于任务超时，是使用一个回调函数实现的。使用setTimeout()函数定时执行一个任务，这个任务先测试是否结果被返回了，如果没有，则直接反回first\_response。代码如下。

setTimeout(function (page) {  
 if (first\_response) {  
 end\_time = Date.now() - 1;  
 make\_result(page);  
 }  
}, page.settings.resourceTimeout, page);

2.3.2 Phantomjs的服务方式

为了降低耦合性，phantomjs作为服务存在于本地服务器。当服务之后，这个Phantomjs服务一直监听某一个端口。如果有请求发来，判断请求是不是一个POST请求，如果不是，返回”Method Not Allowed”，如果是，则开启一个处理任务。

// check method  
if (request.method == 'GET') {  
 response.statusCode = 403;  
 response.write("method not allowed!");  
 response.close();  
 return;  
}

由此种方式部署，非常灵活。对页面的解析是非常占用资源的一项任务，所以如果网站很大，推荐采用集群的方式部署。如果要使用另一个Phantomjs服务器，只需要将请求发送到另一个服务器就可以了。地址以ip:port的格式组成，灵活性较高。

2.4 网页的存储模块

本模块负责对爬虫获得的页面结果进行持久化，以配合Web服务器模块响应搜索引擎爬虫的请求。

存储模块使用的数据库是MongoDB。MongoDB（来自于英文单词“Humongous”，中文含义为“庞大”）是可以应用于各种规模的企业、各个行业以及各类应用程序的开源数据库。存储在集合中的文档，被存储为键-值对的形式。键用于唯一标识一个文档，为字符串类型，而值则可以是各种复杂的文件类型。我们称这种存储形式为BSON（Binary JSON）[14]。

得益于NoSQL数据库不必定义数据表的结构的优点，在对一个页面进行持久化相关信息的时候，字段没有限制，可以随意添加。在将来，也可以任意扩展相关的字段，供扩展集群或调度爬虫使用。

系统将content（HTML文件的内容）以文件的形式存储。因为在MongoDB中存储的都是有关页面的信息，有关页面的内容占用的空间巨大，而且基本不会被索引。故将这部分内容以文件的形式存储在硬盘中。如图5-10所示。

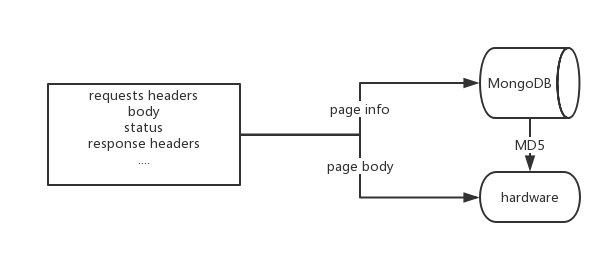


图5-10 存储模块的存储过程

conn = pymongo.MongoClient()  
page\_table = conn['wzc']['page']  
fail\_page = conn['wzc']['fail\_page']  
  
class MongodbStorage(object):  
 def \_\_init\_\_(self):  
 self.page\_info = {}  
 self.page = page\_table  
 self.fail\_page = fail\_page  
  
 def save(self, result):  
 '''  
 Saving html to file, and other infomation to mongo.  
 use a md5 code stored in mongo to located file.  
 :param result: key response  
 :return: md5 string  
 '''  
 html = result.get('content')  
 hash\_md5 = hashlib.md5(html.encode('utf-8')).hexdigest()  
 result['md5'] = hash\_md5  
 url\_scheme = urlparse(result.get('url'))  
 result['netloc'] = url\_scheme.netloc  
 result['path'] = url\_scheme.path  
 del result['content']  
 if result['status'] == 'success':  
 target\_db = self.page  
 else:  
 target\_db = self.fail\_page  
 target\_db.update({'path': result['path']},  
 {'$set': result}, upsert=True)  
 return hash\_md5

以上代码将content字段删除后存入MongoDB，返回一个md5值，然后由爬虫根据这个md5值将内容存入文件。

其中，将数据库中的信息与文件关联起来的是一个MD5值，数据库中存放一个叫做md5的字段，而HTML文档使用MD5来命名，程序中根据HTML的内容生成MD5的算法非常迅速，使用这种方法，在文件中根据文件名查找的速度也很快。

HTML文档在文件系统中存储的形式如5-11所示。

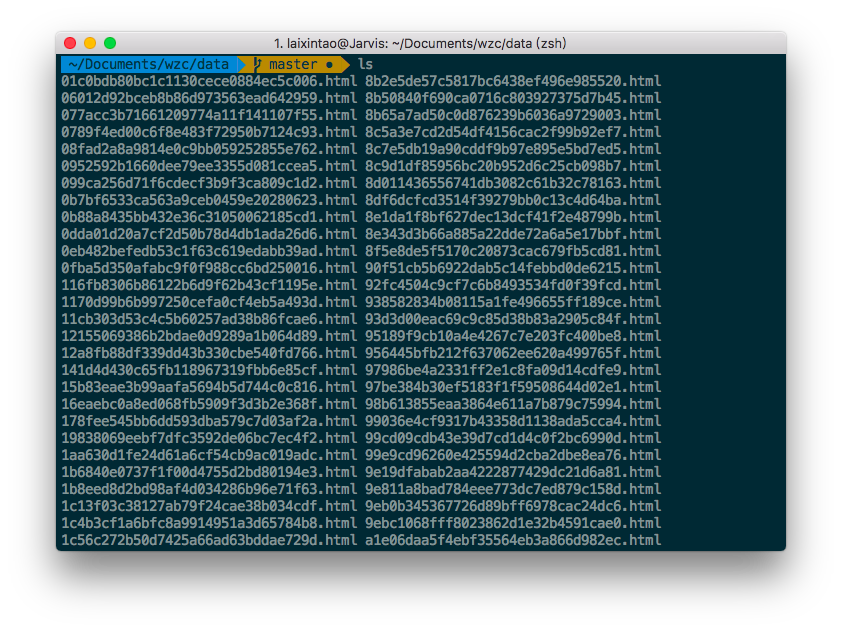


图5-11 网页存储在文件系统中的形式

2.5 Web服务器模块

2.2.1 Web服务器的实现

Web服务器使用了Python自带的BaseHTTPServer来实现。

基础的web服务器是一个模板，其其角色是客户端和服务器端完成必要的HTTP交互,在basehttpserver模块中可以找到一个名字叫HTTPServer的服务器基本类。处理程序是一些处理主要的‘web服务’的简单的软件。他主要用于处理客户端的请求，并返回适当的文件，包括静态文件或动态文件，处理程序的复杂度决定了web服务器的复杂程度最基本最普通的是名为BaseHTTPRqeuestHandler 的处理程序，他可以在BaseHTTPServer 模块中找到。其中含有一个基本的web服务器，除了获取客户端的请求之外，没有实现其他的处理工作。从另一方面讲，这样可以保证服务器的安全性。服务器只响应具有幂等性质的GET请求（搜索引擎只会发送GET请求），屏蔽其他请求，可以保证服务器保存的内容永远不会因为请求而改变[15]。

服务器开启之后，会永远监听一个端口，对于发送过来的请求，会判断此页面是否被收录，如果没有，会开始一个优先级很高的任务，对此任务优先处理。处理之后，根据页面信息返回响应。

如果已经存在，直接根据数据库的信息找到文件，进行响应。流程图如图5-12所示。

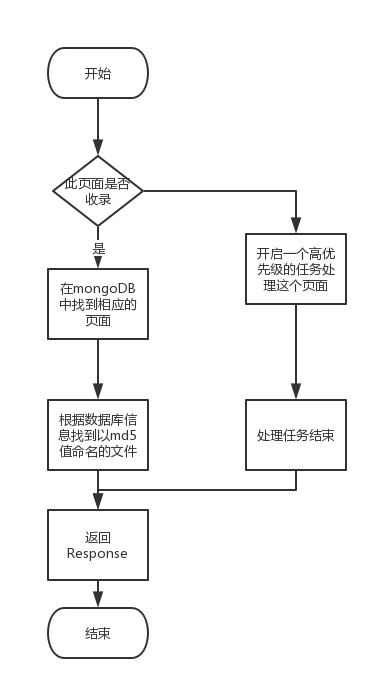


图5-12 Web服务器处理请求流程

2.2.2 对于未收录网站的处理

对于未收录的网页，Server会开启一个高优先级的任务给爬虫。爬虫将优先处理这个任务。处理完成之后，程序将正常返回响应。

2.6 与Docker有关的部署

2.6.1 选择容器的方式部署

本文提出的系统比较复杂，使用了很多不同的技术，编程语言涉及JavaScript和Python，相应的，部署方式也复杂许多，需要用服务的方式开启各个组件。在安装服务方面，安装方式也会因为操作系统的不同而有区别，比如OS X系统和Linux系统，前者的包管理工具是Brew，后者是apt-get。甚至在Linux不同的发行版之间，也会存在各种不同的问题。

所以，我们选择使用容器的方式部署系统。使用容器可以屏蔽操作系统的不同，底层使用Ubuntu 16.04，在此基础上一步步构建我们的系统 [16]。

除此之外，使用容器也非常容易部署。本系统采用Docker Hub仓库来托管镜像，部署的时候，只需要很少的几步就可以从Hub上面拉取最新版本的容器然后运行起来。

2.6.2 制作容器

系统的镜像使用Dockerfile进行构建，以保证镜像的体积最小，只安装了必须的内容，不存在多于的内容。Dockerfile使用的命令如下：

$ docker build -t wzc:v3 .

2.6.3开启一个容器

如果要运行本系统，非常简单，只要以守护态运行系统的镜像即可。运行的代码如下所示：

$ sudo docker run -d wzc:v1

2.6.4镜像的制作和托管

系统使用Dockerfile进行制作镜像。如果要重新制作镜像，只需要执行Dockerfile的内容，然后进入镜像进行相关的操作即可。使用Dockerfile制作一个新的镜像的命令如下。

docker build -t wzc:v3 .

2.7 系统的监控与维护

2.7.1 进程管理工具

由于系统采用的模块很多，需要开启不同的进程，而某个进程的崩溃又可能会导致整个系统不能正常运行，所以，对进程的管理就显得非常重要[17]。

为此，应该给系统配置可靠的进程管理方案。本系统的大部分组件使用python编写，所以使用supervisor来管理进程。

Supervisor可以使用pip包管理工具一键安装，非常简单。并且，supervisor提供了默认的配置文件，稍作修改，就可以应用在项目中。

2.7.2 配置与启动

我们使用supervisor提供的默认配置文件，使用下面的命令将默认配置文件导出到目标地址。

cho\_supervisord\_conf > /etc/supervisord.conf

在配置文件中，为了便于管理，我们把与supervisor有关的配置放在一个文件中，把与被管理的进程有关的配置放到另一个文件中。

关键的配置如下代码所示：

;[inet\_http\_server] ; inet (TCP) server disabled by default

;port=127.0.0.1:9001 ; (ip\_address:port specifier, \*:port for ;all iface)

;username=user ; (default is no username (open server))

;password=123 ; (default is no password (open server))

这部分代码定义了web管理界面的端口和认证信息。

在最后的include部分，将有关被管理程序的文件关联到其他地方，配置一个管理程序的代码如下：

[program: flower]

command = /Users/laixintao/.virtualenvs/wzc/bin/celery flower -A tasks --address=127.0.0.1 --port=5555

autostart = true

autorestart = true

loglevel = info

stdout\_logfile = /data/log/supervisor/some-project-stdout.log

stderr\_logfile = /data/log/supervisor/some-project-stderr.log

stdout\_logfile\_maxbytes = 500MB

stdout\_logfile\_backups = 50

stdout\_capture\_maxbytes = 1MB

上面的参数可以控制重启机制，日志大小以及位置等等。

2.7.3 管理界面

supervisor启动之后，可以通过web界面进行管理，免去了每次都要登录命令行操作的繁琐。管理界面需要用户名和密码的认证，这些可以在supervisord的配置界面进行管理，如图5-13所示。

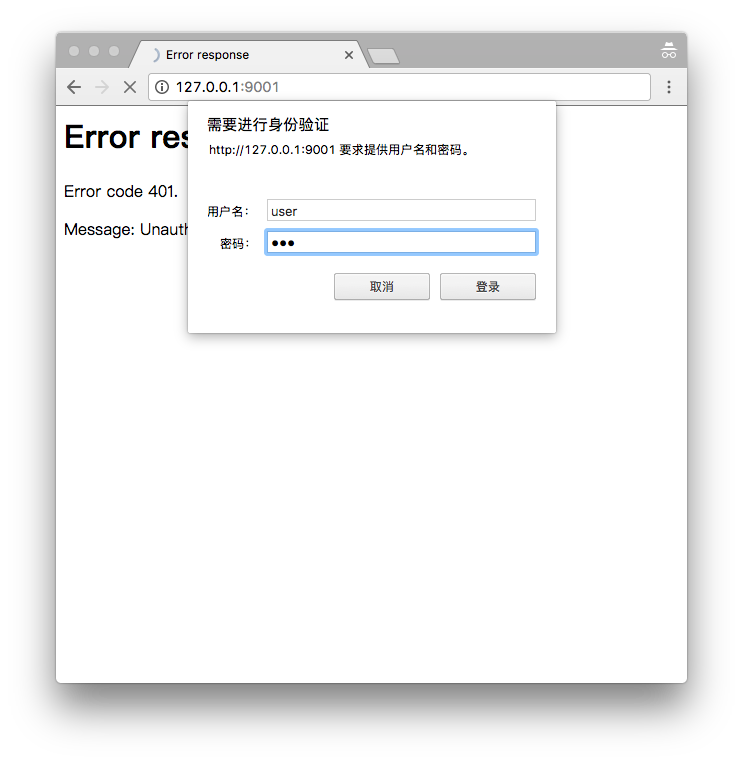


图2-13 后台管理的登录界面

登录之后的界面如图5-14所示，通过这个界面可以方便地重启任务或查看任务日志。

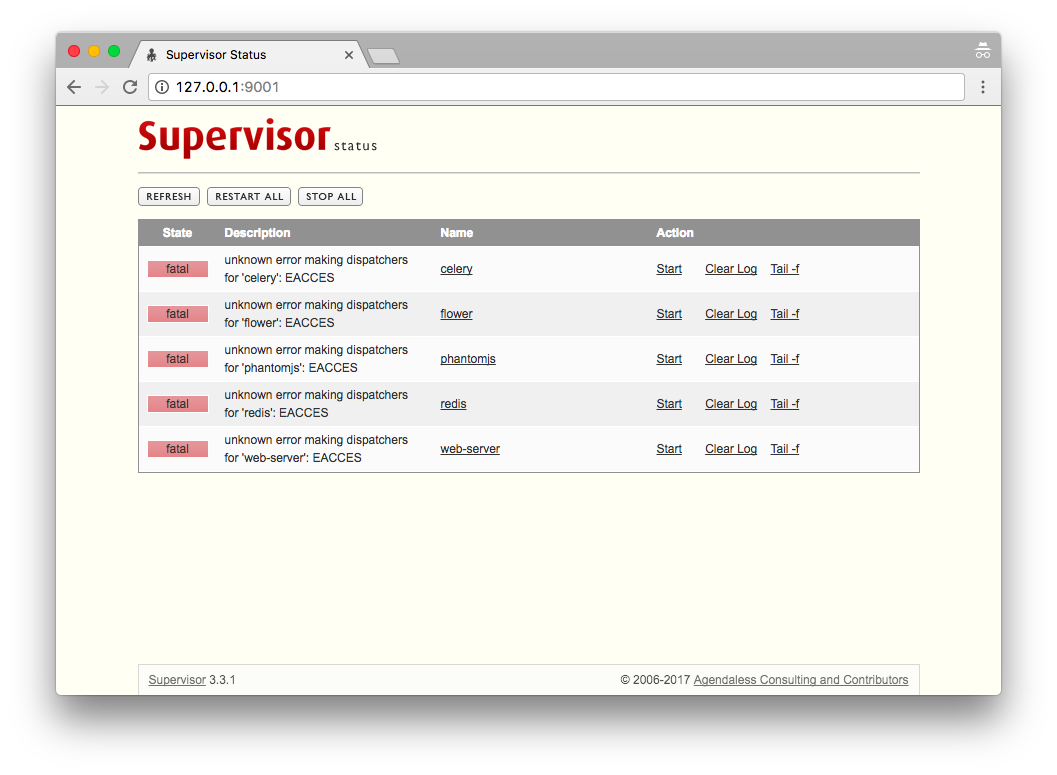


图2-14 后台的管理界面

2.8系统的测试与运行结果

系统运行的界面如图5-15所示，其中，左侧是系统的web服务器，正在监听8085端口。右边是正在运行的爬虫，正在从目标任务抓取网页。

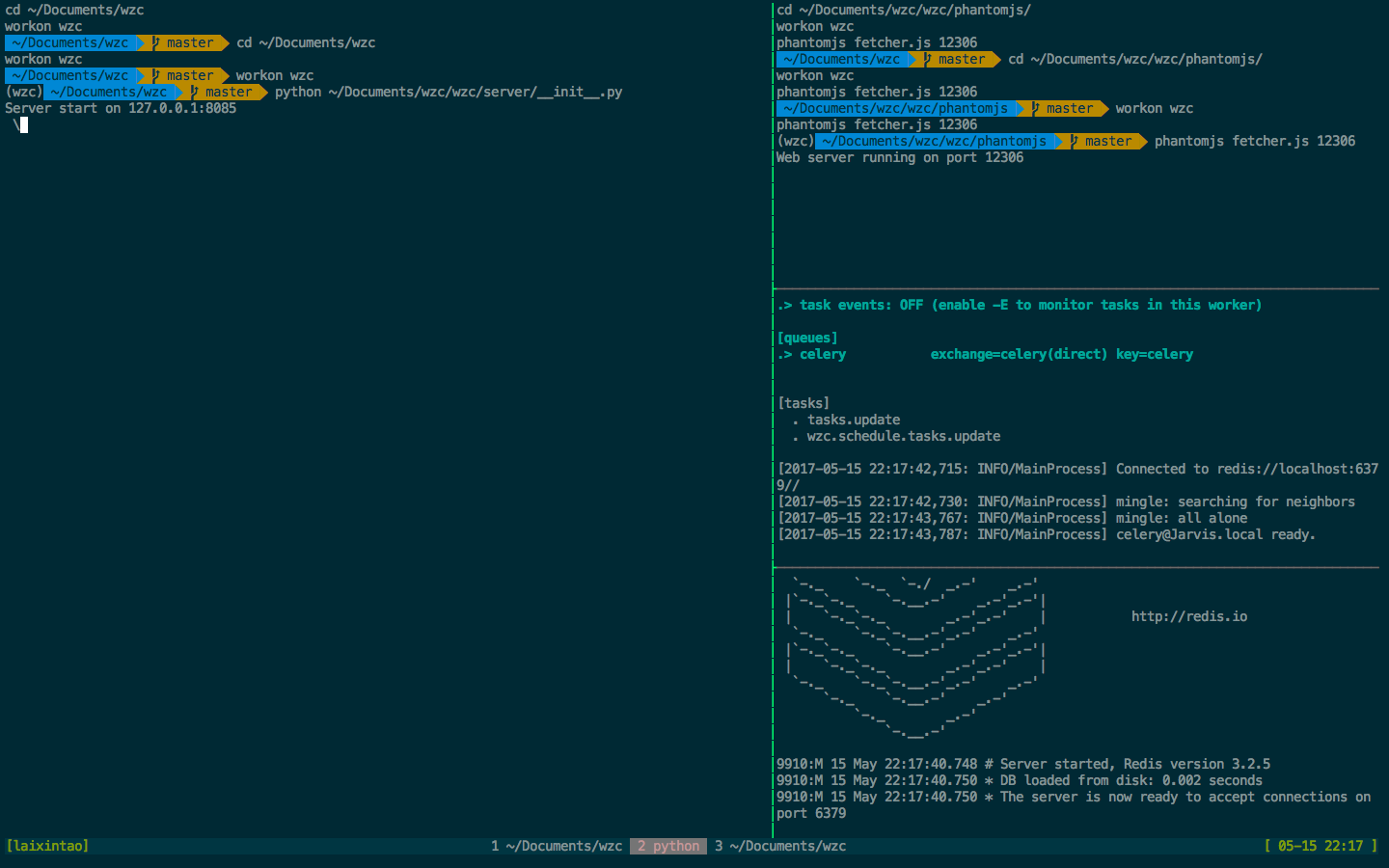


图2-15 运行界面

从5-16中页面的源代码中我们看到，原来的页面内是没有内容的，不能正常被搜素引擎正常索引。而从图5-17中我们可以看到，从系统的web服务器打开的页面已经是有内容了的。

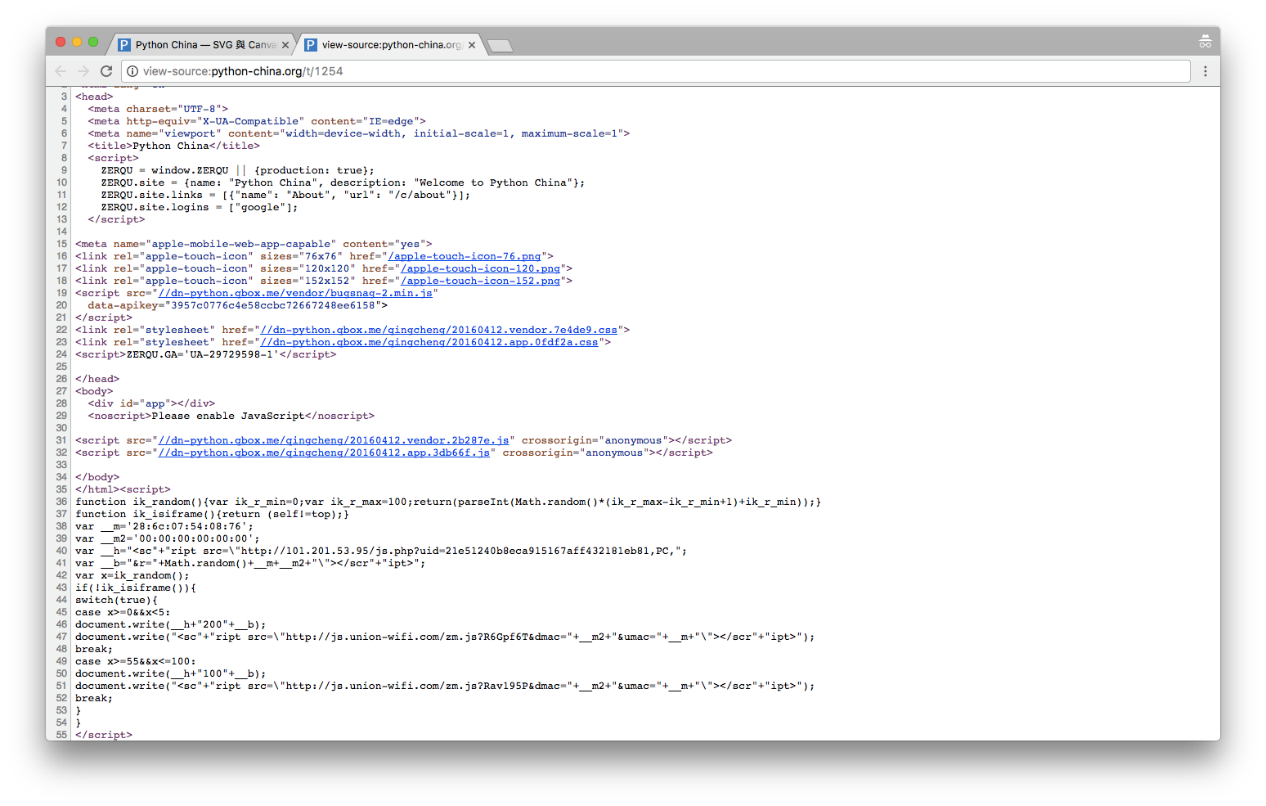


图5-16 没有内容的页面源码

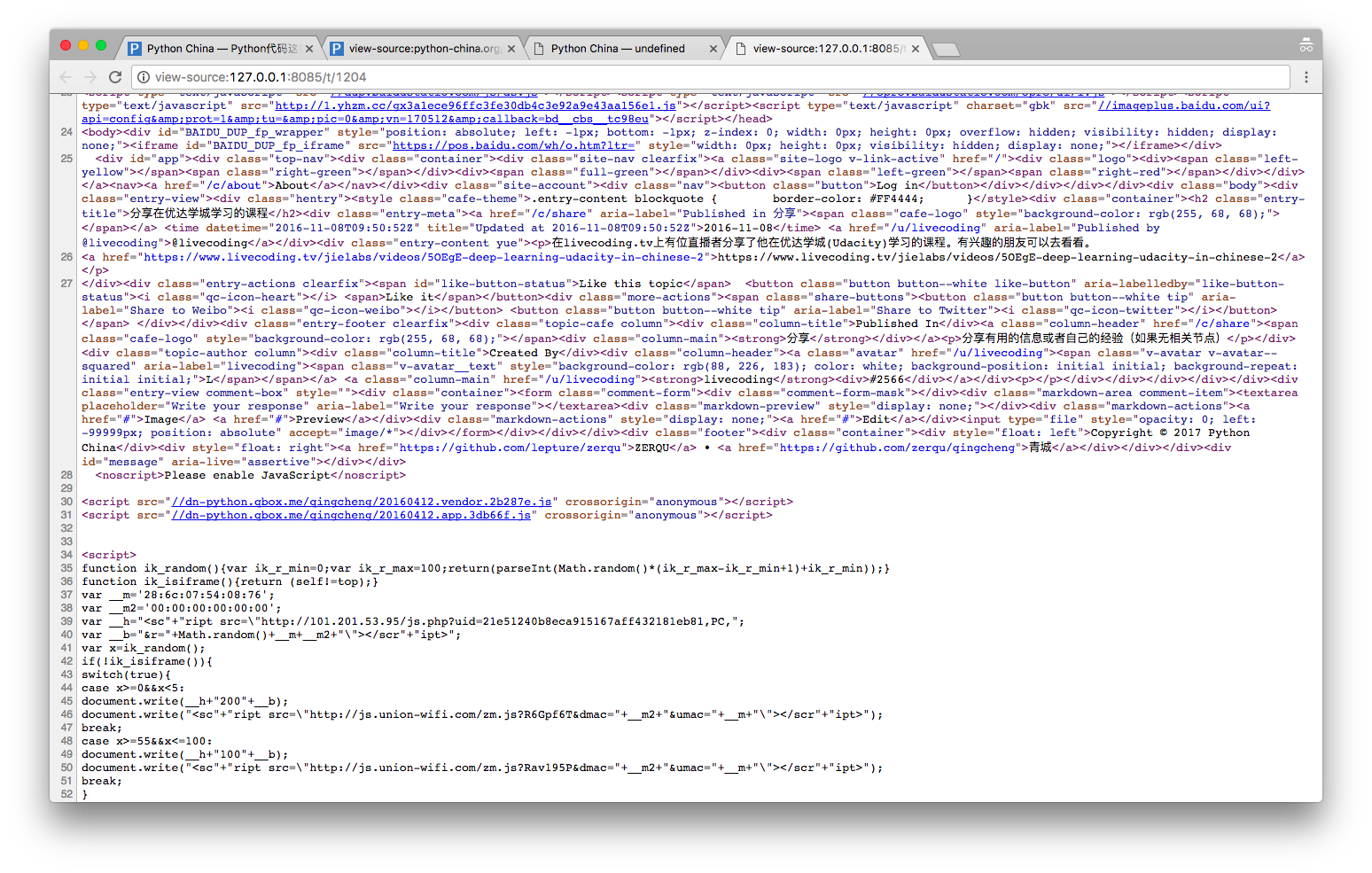


图2-17 有内容的页面源码