基于无界面浏览器技术的单页应用搜索引擎优化

近年来，兴起了很多使用单页应用开发网页的开发框架。单页应用大量采用Ajax技术，在不刷新网页的情况下就可以更新网页的内容。给用户体验带来了很大的提升。

但是随之也产生很多问题，用单页应用开发网页最大的痛点就是对搜索引擎的收录较差。由于网页的内容是基于执行页面内的JavaScript脚本，使用Ajax填充网页的内容。所以当搜索引擎的爬虫请求页面内容的时候，单页应用所能响应的，只有一个HTML以及JavaScript脚本。在JavaScript脚本之前，网页内并没有实质的内容。

本文提出，使用基于无界面浏览器技术的分布式爬虫对自身网页进行解析，填充内容，对搜索引擎提交完整的网页。以此来优化搜索引擎的收录问题。本文所提出的系统，采用容器方式部署，与系统无关，兼容性强。采用分布式的任务队列，各模块之间以组件的形式相互配合，扩展性强。可以根据网站大小，部署在集群或者单台服务器上。

Single Page Application Search Engine Optimize Based on Headless Broswer

基于无界面浏览器技术的单页应用搜索引擎优化 1

Single Page Application Search Engine Optimize Based on Headless Broswer 2

1 绪论 6

1.1 引言 6

1.2 课题研究现状 6

1.3 课题研究内容 6

1.4 本系统的创新 6

1.5 本章小结 6

2 相关技术 7

2.1 无界面浏览器技术 7

2.2 分布式爬虫技术 7

2.3 NoSQL数据库技术 7

2.4 本章小结 7

3 系统分析 8

3.1 可行性分析 8

3.2 问题定义 8

3.3. 需求分析 8

3.3.1 功能需求 8

3.3.2性能需求 9

3.4 开发环境 9

3.5 本章小结 9

4.单页应用SEO优化系统的设计与实现 11

4.1 系统的结构设计 11

4.1.2 系统设计 11

4.1.2 系统代码的组织 12

4.2 详细设计 13

4.2.1 爬虫模块 13

4.2.2 页面的JavaScript执行模块 13

4.2.3 网页的存储模块 13

4.2.4 响应请求的web服务器模块 14

4.2.5 使用docker部署 14

4.2.6 在服务器上的部署 14

5 系统实现 16

5.1 设计平台搭建 16

5.1.1 开源工具选择 16

5.1.2 设计平台配置 16

5.1.3 生产环境服务器的要求 17

5.2 爬虫模块 17

5.2.1 爬虫模块的实现 17

5.3 Phantomjs模块 26

5.3.1 phantomjs模块的实现 26

5.3.2 Phantomjs的服务方式 33

5.4 网页的存储模块 34

5.4.1 网页存储模块的实现 34

5.4.2 网页更新的难点 36

5.5 Web服务器模块 37

5.5.1 Web服务器的实现 37

5.5.2 对于未收录网站的处理 40

5.6 与Docker有关的部署 40

5.6.1 制作一个镜像 40

5.6.2 开启一个容器 40

5.6.3 镜像的制作及托管 42

6 系统的部署与维护 44

6.1 在Linux内核的机器上使用容器部署 44

6.2 部署一个集群 44

6.3 监控集群的健康度 44

7 总结与展望 45

7.1 总结 45

7.2 展望 45

参考文献 46

致谢 47

1 绪论

* 1. 引言
  2. 课题研究现状
  3. 课题研究内容
  4. 本系统的创新

1.5 本章小结

2 相关技术

2.1 无界面浏览器技术

PhantomJS是一个基于webkit内核、支持[JavaScript](http://wiki.li3huo.com/JavaScript)/[CoffeeScript](http://wiki.li3huo.com/CoffeeScript) API的无界面浏览器，并且原生支持DOM/CSS/JSON/Canvas/SVG等[W3C颁布的互联网技术标准](http://wiki.li3huo.com/W3C#W3C_Recommendation)。

因此，我们可以使用[JavaScript](http://wiki.li3huo.com/JavaScript)或[CoffeeScript](http://wiki.li3huo.com/CoffeeScript)来模拟一个现代浏览器在加载网页时所做的各种事件，例如页面自动化，网络监测，网页截屏，以及无界面测试等。

2.2 分布式爬虫技术

2.3 NoSQL数据库技术

2.4 本章小结

3 系统分析

本系统框架主要面向网站管理员，使采用单页面应用技术的网站能够轻松地使用本框架达到搜索引擎优化的效果。本系统主要的功能是对搜索引擎采取的优化，除此之外，还允许用户进行一些自定义的优化，例如执行页面特定的JavaScript函数等。另外，系统还要向管理员提供监控页面和接口，让本系统方便管理和维护。

3.1 可行性分析

当前，使用vue，react，angular等技术开发单页面web应用已成为一种趋势，但是单页应用使用了大量的ajax技术，如何解决SEO就成了一个难题。

phantomjs本是用来进行自动化测试的一套工具，它可以在没有浏览器的情况下执行页面中的JavaScript代码（包括发送ajax请求）。使用它我们就像一个用户一样，执行页面的js代码，填充页面的内容，然后将解析后的内容发送给搜索引擎。这样我们就可以解决单页引用的搜索引擎优化难题了。

3.2 问题定义

本系统的核心模块是对页面中JavaScript的执行以及爬虫，这是基本的功能。何时进行对网站更新，保证与原网页的实时性又不对网站的访问太过频繁是系统优化的重点。本系统将利用mongo数据库配合爬虫和phantomjs优化抓取策略以及存储策略。

* 1. 需求分析
     1. 功能需求

1. 系统可以对源网站自动进行抓取。
2. 如果提供sitemap，系统可以根据sitemap进行抓取。
3. 系统可以将保存下来的网页与新抓取的网页进行快速对比，判断网页是否已经更新。
4. 系统可以将网页提交给搜索引擎，被搜索引擎正常收录。
5. 系统提供方便，直观的管理界面，方便监控和维护各个服务。
6. 守护进程，确保程序的正常运行。

3.3.2性能需求

1. 能够对新产生的网页进行快速收录。
2. 对搜索引擎爬虫发送过来的请求，要保证及时响应（100ms以内）。
3. 健壮性：对自身网站存在的错误，要能够处理，不能导致系统崩溃。
4. 要尽量减少对源网站的请求次数，避免使原网站的负载压力过大。
5. 易监控，提供监控页面，可以快速、直观地查看系统运行情况。
6. 易维护，系统出现问题后，可以快速定位到问题并排查。

3.4 开发环境

硬件要求： 苹果笔记本电脑。

软件： zsh shell（配合oh-my-zsh），tmux终端工具，vim编辑器，git版本控制工具

开发语言：JavaScript，Python

3.5 本章小结

本章对本系统和框架进行了详细介绍。详细地分析了系统的功能，并且定义了性能要求。证明了该方案是可能的，为后面的开发增加了信心，奠定了基础。此外，还介绍了开发环境。

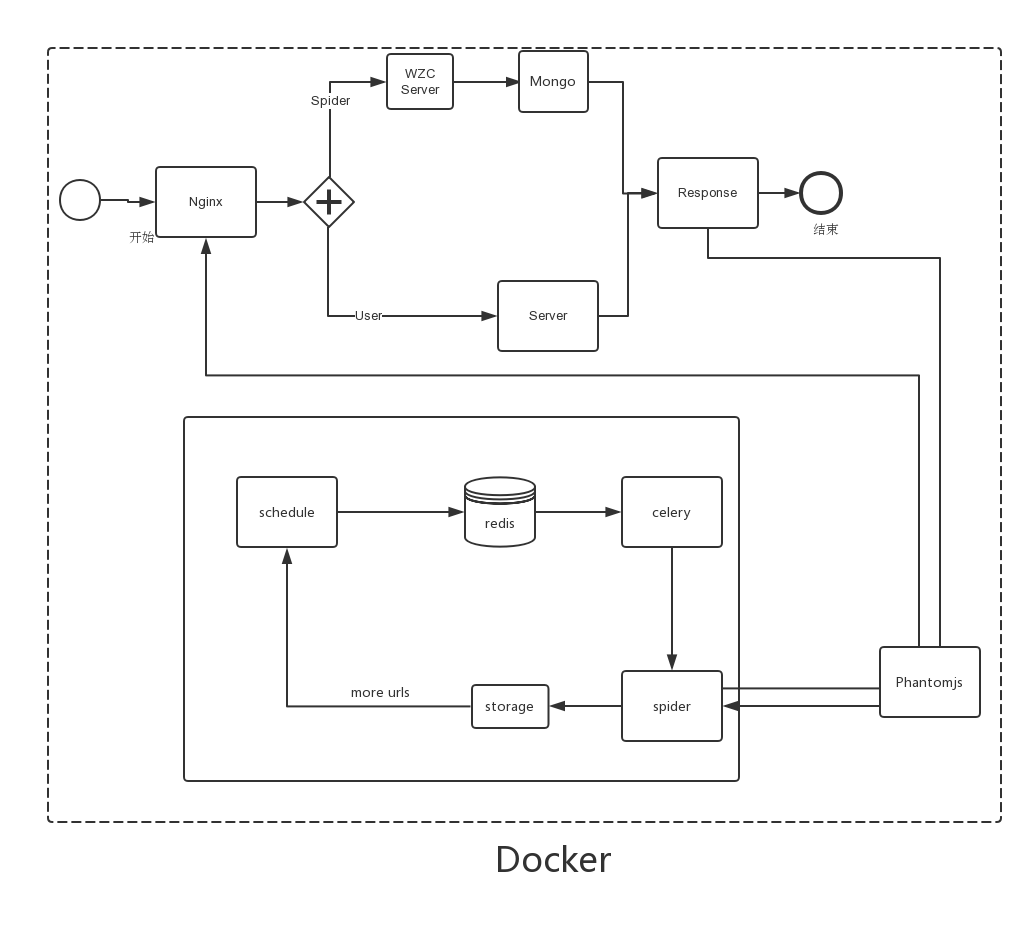
4.单页应用SEO优化系统的设计与实现

4.1 系统的结构设计

4.1.2 系统设计

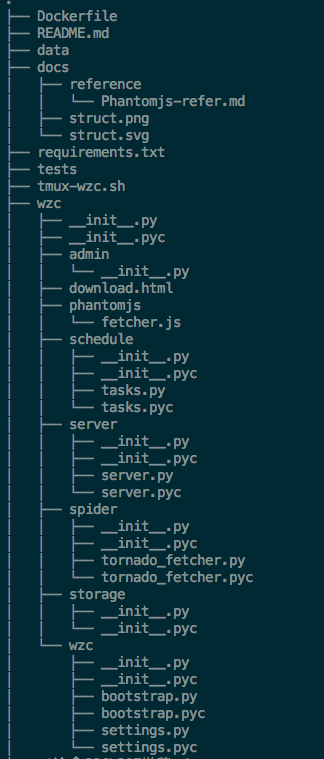
本系统为单页应用的SEO优化而设计，整个系统主要分为两部分。第一部分是爬虫引擎，主要负责从目标网站抓取页面内容，将爬取的任务信息存到数据库。第二部分是网页服务器部分，主要负责给搜索引擎返还真实的页面内容。

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4-1所示。



4.1.2 系统代码的组织

系统所有代码如下图所示：



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。

4.2 详细设计

4.2.1 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，的主要功能是：

1. 抓取网站的页面，通过js执行模块执行页面的js内容。
2. 将执行之后的页面通过网页的存储模块存储下来。

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）

1. 提供可视化的任务队列管理页面

4.2.2 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，主要功能是：

1. 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。
2. 设置网页的meta信息，例如载入时间，状态，请求的headers等。
3. 能解析页面上的JavaScript脚本（包括Ajax请求）。
4. 能执行页面上的document-start，支持有特别需求的SEO。

4.2.3 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，需要实现的功能有：

1. 将页面信息存入数据库。
2. 将页面内容作为文件存储。
3. 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。

4.2.4 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。本模块的功能有：

1. 接受一个请求，能正确返回静态的结果。
2. 如果页面尚未被收录，马上开启一个高优先级的任务优先处理此页面。
3. 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。
4. 对所有请求进行记录，以便于进行SEO优化以及监控使用。

4.2.5 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誓和部署，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。

4.2.6 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，讲一个端口映射到容器内的Web服务器地址即可。

5 系统实现

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。

5.1 设计平台搭建

5.1.1 开源工具选择

1. 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。
2. 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。
3. 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。
4. 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。
5. 版本控制使用Git。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。

5.1.2 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，硬件配置如下：

|  |  |
| --- | --- |
| 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 |

5.1.3 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。如果有必要，可以使用集群的方式灵活部署。这里，以一个小型的网站为例，硬件要求大致如下。

|  |  |
| --- | --- |
| 服务器硬件要求 | |
|  |  |
|  |  |
|  |  |
|  |  |

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。

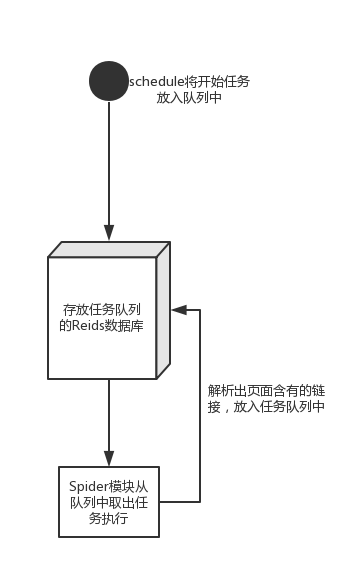
5.2 爬虫模块

5.2.1 爬虫模块的实现

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。

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。队列的架构如下图所示。任务队列是一种在进程或机器之间分发任务的机制。 任务队列的输入是被称为任务（task）的工作单元。专用的工作进程会时刻监控任务队列，来获取要执行的任务。 celery的client和worker通过消息来“沟通“。Celery需要依靠RabbitMQ等作为消息代理，同时也支持Redis甚至是Mysql，Mongo等，当然，官方默认推荐的是RabbitMQ。 为了开始一个任务，client需要向队列中发送任务消息，然后broker会把任务投递给worker处理。一个celery系统可以包含多个worker和broker，以便实现高可用和可水平扩展。



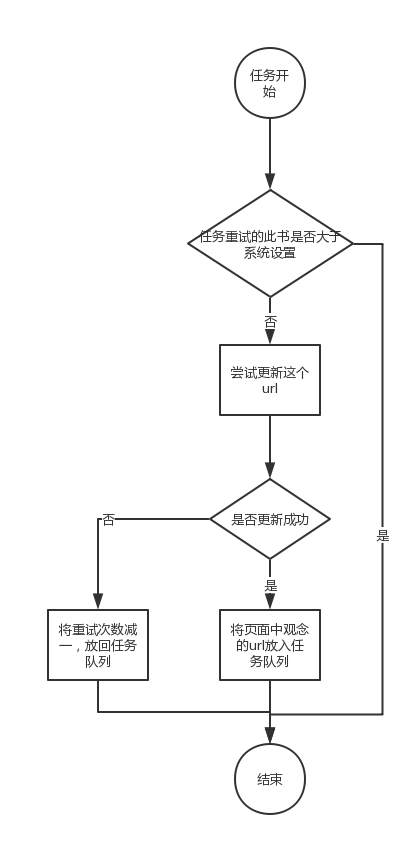
本模块的任务队列架构使用了Celery，Celery是一个使用Python开发的分布式任务调度模块，因此对于大量使用Python构建的系统，可以说是无缝衔接，使用起来很方便。Celery专注于实时处理任务，同时也支持任务的定时调度。因此适合实时异步任务定时任务等调度场景。

本系统选择了Redis作为Broker。原因有二，第一，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。第二，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做两件事：第一件，更新目标URL，即完成目标任务。第二件，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。流程如下图所示。



1. 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。

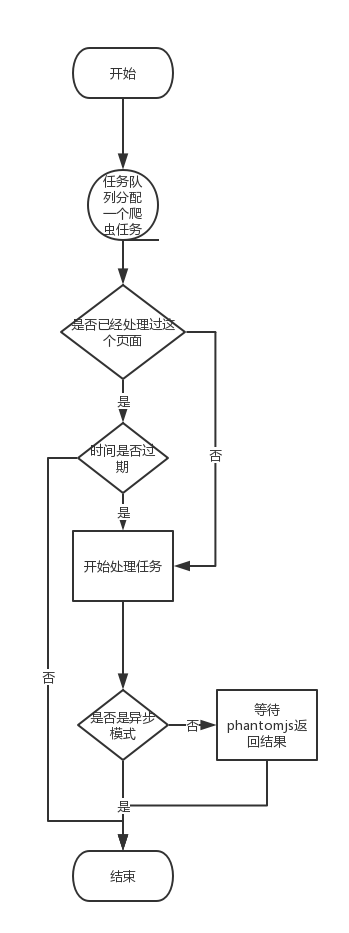
受网络、系统资源等因素影响，更新一个URL的任务并不是总是可以成功执行。如果任务失败，将会触发异常机制，报告异常，并将异常的页面信息存放如数据库。对于调用者，返回None，以进行安排任务重试。

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。它们的关系图如下：



整个爬虫任务的流程图如下：

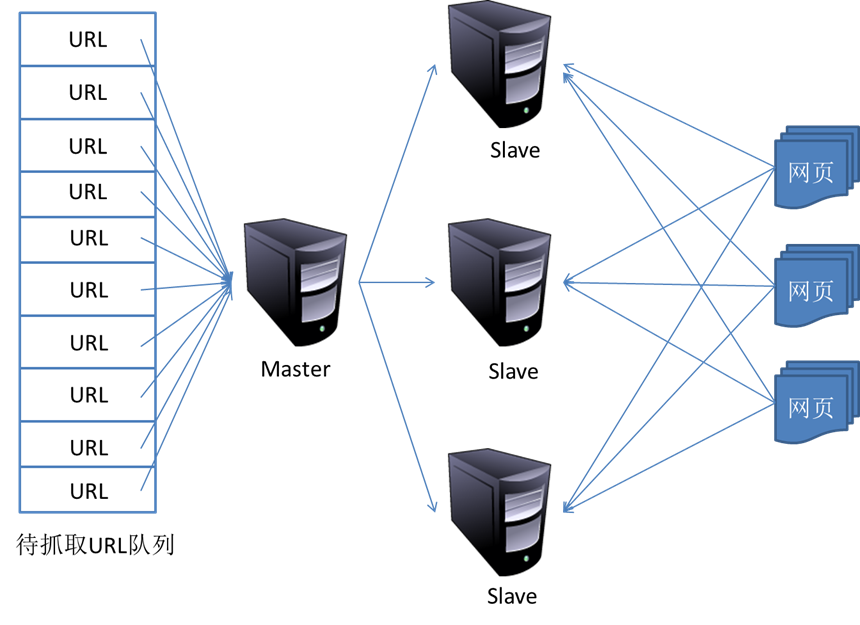


1. 使用分布式集群爬虫

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。

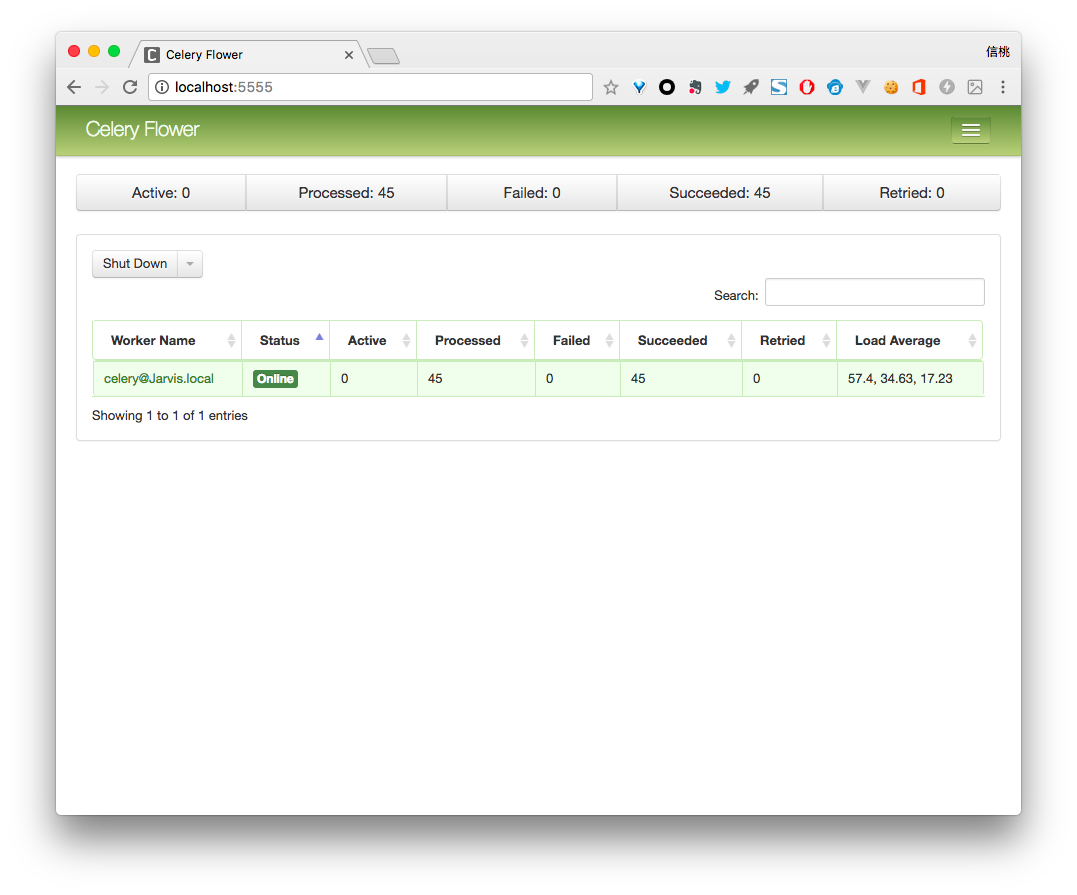
使用的方法非常简单，只需要在设置里指明所使用的Broker的地址，以及mongo的地址便可。作为Broker的机器就是Master，其余的机器从Master里面取任务执行，将可能存在的更多的任务放回Master主机里。这个架构如下图所示。



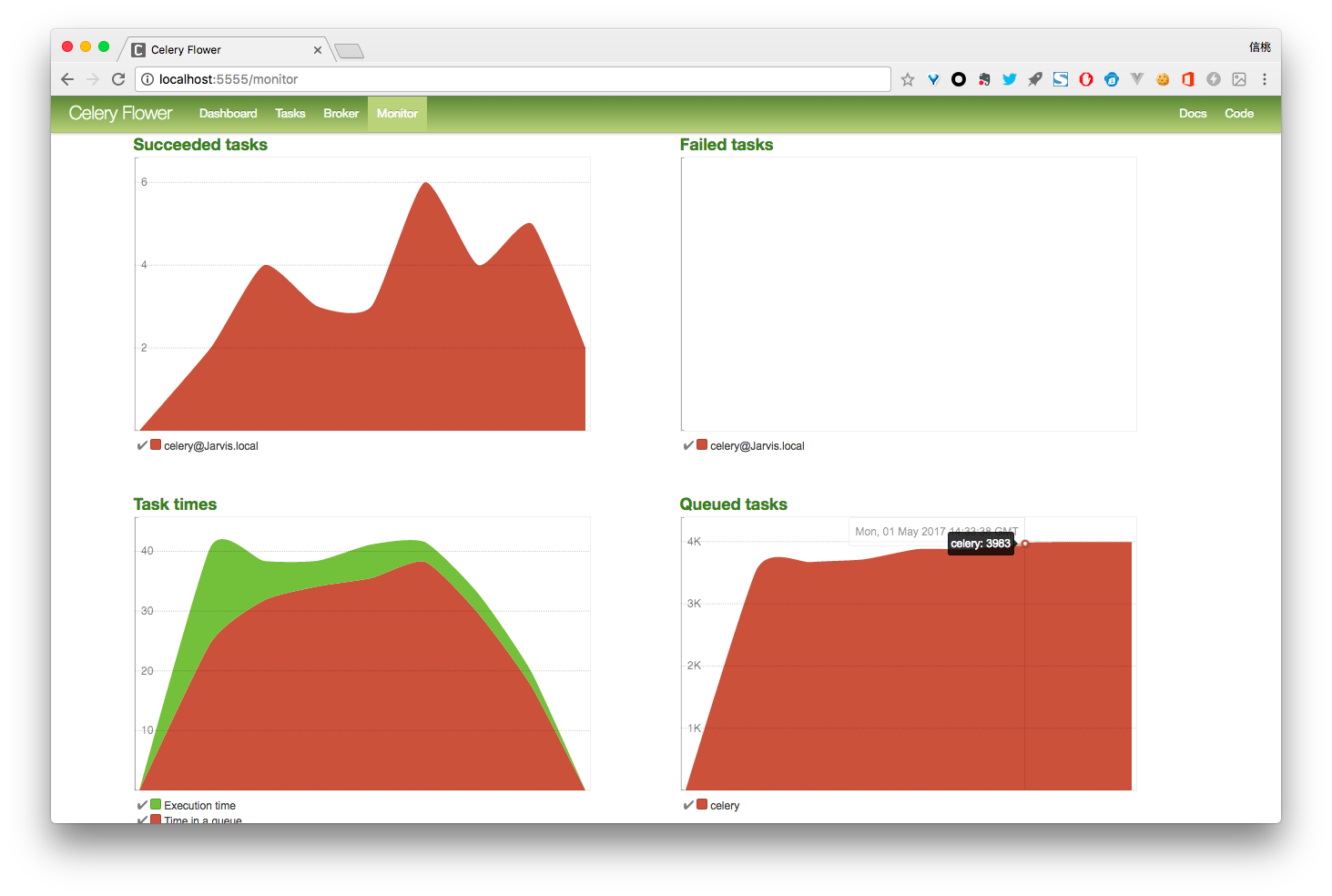
1. 对任务队列的监控

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。

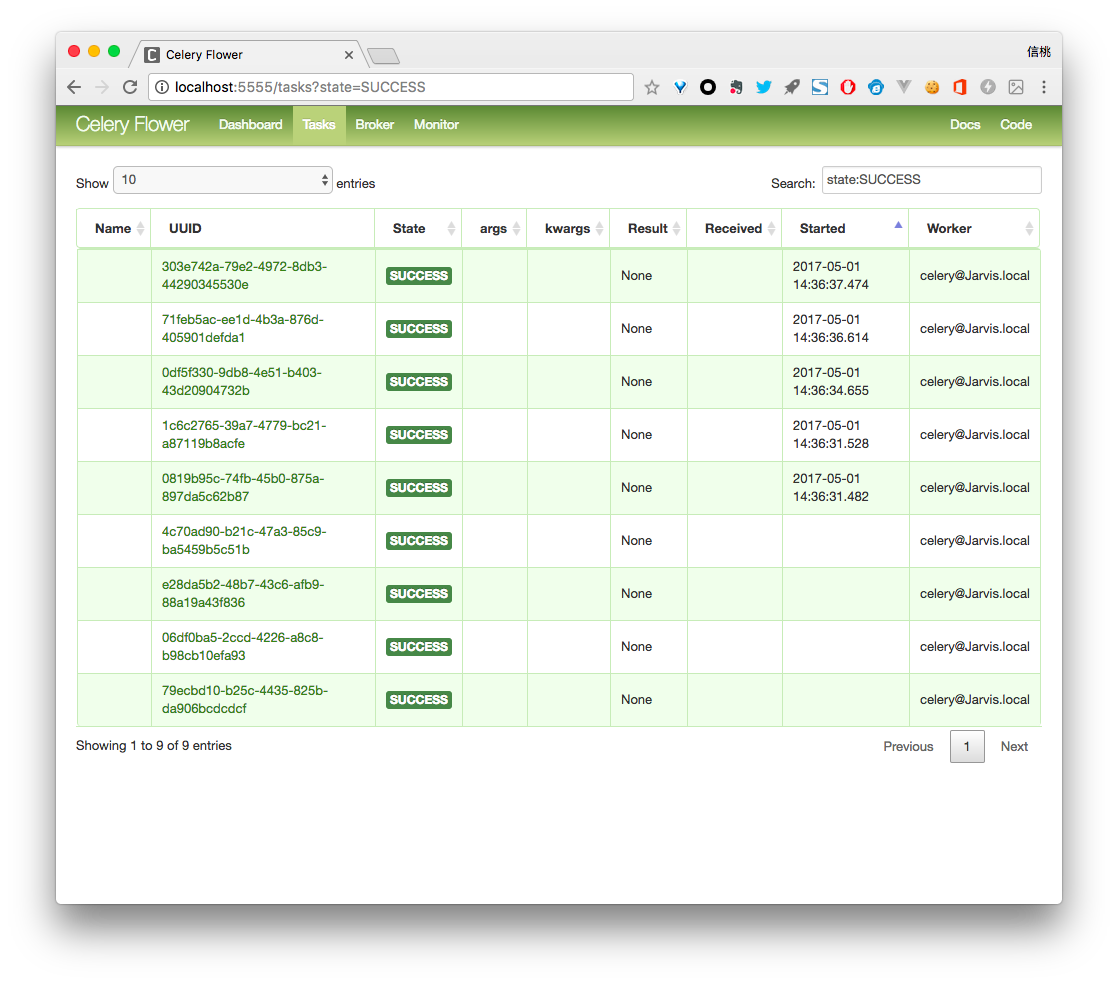
从首页可以看到，正在运行的worker以及他们的状态，这些worker完成的任务，失败的任务等等。还可以快速重启、关闭指定的worker。



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。



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。



5.3 Phantomjs模块

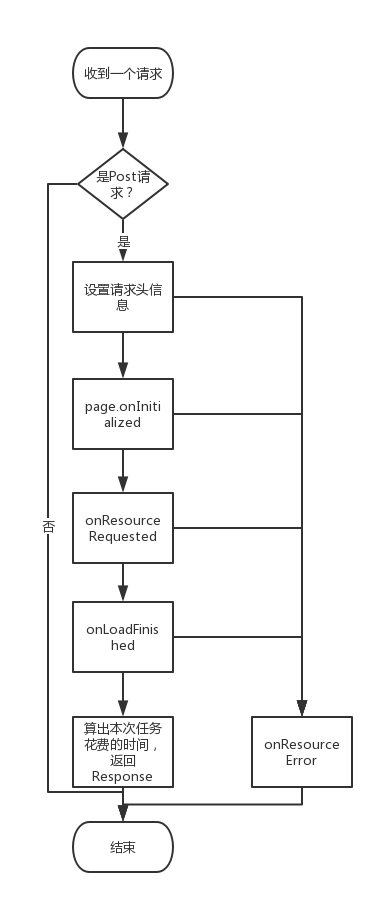
5.3.1 phantomjs模块的实现

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。

[PhantomJS](http://phantomjs.org/)是一个基于[WebKit](http://www.webkit.org/)的服务器端JavaScript API，它基于 [BSD](http://opensource.org/licenses/BSD-3-Clause)开源协议发布。PhantomJS无需浏览器的支持即可实现对Web的支持，且原生支持各种Web标准，如DOM 处理、JavaScript、CSS选择器、JSON、Canvas和可缩放矢量图形SVG。PhantomJS主要是通过JavaScript和[CoffeeScript](http://coffeescript.org/)控制WebKit的CSS选择器、可缩放矢量图形SVG和HTTP网络等各个模块。PhantomJS主要支持Windows、Mac OS、Linux三个平台，并且提供了对应的二进制安装包，读者可在[下载页面](http://phantomjs.org/download.html)选择对应的版本下载解压即可使用。GitHub上还提供了GitHub、Google Code、[BitBucket](https://bitbucket.org/)等各种方式的[源码下载地址](https://github.com/ariya/phantomjs/wiki/Source-Code)。

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。



onResourceRequested和onResourceReceived是两个核心的回调函数。

onResourceRequested

onResourceRequested属性用来指定一个回调函数，当页面请求一个资源时，会触发这个回调函数。它的第一个参数是HTTP请求的元数据对象，第二个参数是发出的网络请求对象。

HTTP请求包括以下字段。

id：所请求资源的编号

method：使用的HTTP方法

url：所请求的资源 URL

time：一个包含请求时间的Date对象

headers：HTTP头信息数组

网络请求对象包含以下方法。

abort()：终止当前的网络请求，这会导致调用onResourceError回调函数。

changeUrl(newUrl)：改变当前网络请求的URL。

setHeader(key, value)：设置HTTP头信息。

var webPage = require('webpage');

var page = webPage.create();

page.onResourceRequested = function(requestData, networkRequest) {

console.log('Request (#' + requestData.id + '): ' + JSON.stringify(requestData));

};

onResourceReceived

onResourceReceived属性用于指定一个回调函数，当网页收到所请求的资源时，就会执行该回调函数。它的参数就是服务器发来的HTTP回应的元数据对象，包括以下字段。

id：所请求的资源编号

url：所请求的资源的URL r- time：包含HTTP回应时间的Date对象

headers：HTTP头信息数组

bodySize：解压缩后的收到的内容大小

contentType：接到的内容种类

redirectURL：重定向URL（如果有的话）

stage：对于多数据块的HTTP回应，头一个数据块为start，最后一个数据块为end。

status：HTTP状态码，成功时为200。

statusText：HTTP状态信息，比如OK。

如果HTTP回应非常大，分成多个数据块发送，onResourceReceived会在收到每个数据块时触发回调函数。

var webPage = require('webpage');

var page = webPage.create();

page.onResourceReceived = function(response) {

console.log('Response (#' + response.id + ', stage "' + response.stage + '"): ' + JSON.stringify(response));

};

创建一个phantomjs server核心的代码如下：

service **=** server.listen(port, {  
 'keepAlive'**: true**}, **function** (request, response) {  
 phantom.clearCookies();  
  
 *//console.debug(JSON.stringify(request, null, 4));  
 // check method* **if** (request.method **==** 'GET') {  
 response.statusCode **=** 403;  
 response.write("method not allowed!");  
 response.close();  
 **return**;  
 }  
  
 **var** fetch **=** *JSON*.parse(request.postRaw);  
 console.debug(*JSON*.stringify(fetch, **null**, 2));  
  
 *// create and set page* **var** page **=** webpage.create();  
 page.viewportSize **=** {  
 width**:** 1024,  
 height**:** 768  
 }  
 **if** (fetch.headers **&&** fetch.headers['User-Agent']) {  
 page.settings.userAgent **=** fetch.headers['User-Agent'];  
 }  
 page.settings.loadImages **=** fetch.load\_images **? true : false**;  
 page.settings.resourceTimeout **=** fetch.timeout **?** fetch.timeout **\*** 1000 **:** 120 **\*** 1000;  
 **if** (fetch.headers) {  
 page.customHeaders **=** fetch.headers;  
 }  
  
 *// add callbacks* **var** first\_response **= null**,  
 finished **= false**,  
 page\_loaded **= false**,  
 start\_time **= Date**.now(),  
 end\_time **= null**,  
 script\_executed **= false**,  
 script\_result **= null**;  
 page.onInitialized **= function** () {  
 **if** (**!**script\_executed **&&** fetch.js\_script **&&** fetch.js\_run\_at **===** "document-start") {  
 script\_executed **= true**;  
 console.log('running document-start script.');  
 script\_result **=** page.evaluateJavaScript(fetch.js\_script);  
 }  
 };  
 page.onLoadFinished **= function** (status) {  
 page\_loaded **= true**;  
 **if** (**!**script\_executed **&&** fetch.js\_script **&&** fetch.js\_run\_at **!==** "document-start") {  
 script\_executed **= true**;  
 console.log('running document-end script.');  
 script\_result **=** page.evaluateJavaScript(fetch.js\_script);  
 }  
 console.debug("waiting " **+** wait\_before\_end **+** "ms before finished.");  
 end\_time **= Date**.now() **+** wait\_before\_end;  
 setTimeout(make\_result, wait\_before\_end **+** 10, page);  
 };  
 page.onResourceRequested **= function** (request) {  
 console.debug("Starting request: #" **+** request.id **+** " [" **+** request.method **+** "]" **+** request.url);  
 end\_time **= null**;  
 };  
 page.onResourceReceived **= function** (response) {  
 console.debug("Request finished: #" **+** response.id **+** " [" **+** response.status **+** "]" **+** response.url);  
 **if** (first\_response **=== null &&** response.status **!=** 301 **&&** response.status **!=** 302) {  
 first\_response **=** response;  
 }  
 **if** (page\_loaded) {  
 console.debug("waiting " **+** wait\_before\_end **+** "ms before finished.");  
 end\_time **= Date**.now() **+** wait\_before\_end;  
 setTimeout(make\_result, wait\_before\_end **+** 10, page);  
 }  
 }  
 page.onResourceError **=** page.onResourceTimeout **= function** (response) {  
 console.info("Request error: #" **+** response.id **+** " [" **+** response.errorCode **+** "=" **+** response.errorString **+** "]" **+** response.url);  
 **if** (first\_response **=== null**) {  
 first\_response **=** response;  
 }  
 **if** (page\_loaded) {  
 console.debug("waiting " **+** wait\_before\_end **+** "ms before finished.");  
 end\_time **= Date**.now() **+** wait\_before\_end;  
 setTimeout(make\_result, wait\_before\_end **+** 10, page);  
 }  
 }

5.3.2 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。

5.4 网页的存储模块

5.4.1 网页存储模块的实现

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。

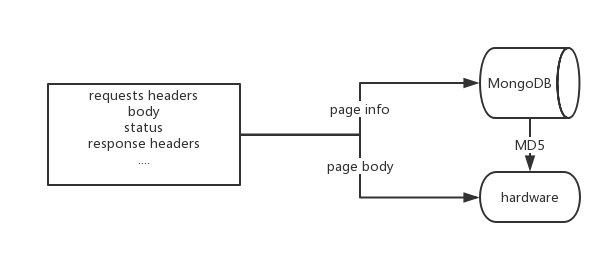
存储模块使用的数据库是MongoDB。MongoDB（来自于英文单词“Humongous”，中文含义为“庞大”）是可以应用于各种规模的企业、各个行业以及各类应用程序的开源数据库。作为一个适用于敏捷开发的数据库，MongoDB的数据模式可以随着应用程序的发展而灵活地更新。与此同时，它也为开发人员 提供了传统数据库的功能：二级索引，完整的查询系统以及严格一致性等等。

所谓“面向集合”（Collenction-Oriented），意思是数据被分组存储在数据集中，被称为一个集合（Collenction)。每个集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库（RDBMS）里的表（table），不同的是它不需要定义任何模式（schema)。

模式自由（schema-free)，意味着对于存储在mongodb数据库中的文件，我们不需要知道它的任何结构定义。如果需要的话，你完全可以把不同结构的文件存储在同一个数据库里。

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）。

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。如下图所示。



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。

5.4.2 网页更新的难点

怎么判断网页内容相比于上一次是更新了，这是一个难点。使用字符串比较不同，肯定是不合理的。因为网页内容太多了，使用字符串比较将占用很大的内存，效率也很低。

所以，我们采用存储MD5值，并用这个值进行比较的方法。

在数据库中，只存储从content生成的md5值，然后将content写入一个文件，这个文件用md5值命名。这样，我们需要网页内容的时候，直接根据数据库的md5值去打开对应的文件即可。

当试图更新一个网页的时候，计算新页面内容的md5值，如果和之前存储的值相同，说明网页内容未发生变化，如果发生了改变，则说明网页更新了。

MD5消息摘要算法（英语：MD5 Message-Digest Algorithm），一种被广泛使用的[密码散列函数](https://zh.wikipedia.org/wiki/%E5%AF%86%E7%A2%BC%E9%9B%9C%E6%B9%8A%E5%87%BD%E6%95%B8)，可以产生出一个128位（16[字节](https://zh.wikipedia.org/wiki/%E5%AD%97%E8%8A%82)）的散列值（hash value），用于确保信息传输完整一致。MD5由[罗纳德·李维斯特](https://zh.wikipedia.org/wiki/%E7%BD%97%E7%BA%B3%E5%BE%B7%C2%B7%E6%9D%8E%E7%BB%B4%E6%96%AF%E7%89%B9)设计，于1992年公开，用以取代[MD4](https://zh.wikipedia.org/wiki/MD4)算法。这套算法的程序在 [RFC 1321](https://tools.ietf.org/html/rfc1321) 中被加以规范。

将[数据](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE)（如一段文字）运算变为另一固定长度值，是散列算法的基础原理。

MD5是输入不定长度信息，输出固定长度128-bits的算法。经过程序流程，生成四个32位数据，最后联合起来成为一个128-bits[散列](https://zh.wikipedia.org/wiki/%E6%95%A3%E5%88%97)。基本方式为，求余、取余、调整长度、与链接变量进行循环运算。得出结果。

一般128位的MD5散列被表示为32位[十六进制](https://zh.wikipedia.org/wiki/%E5%8D%81%E5%85%AD%E8%BF%9B%E5%88%B6)数字。以下是一个43位长的仅[ASCII](https://zh.wikipedia.org/wiki/ASCII)字母列的MD5散列：

MD5("The quick brown fox jumps over the lazy dog")

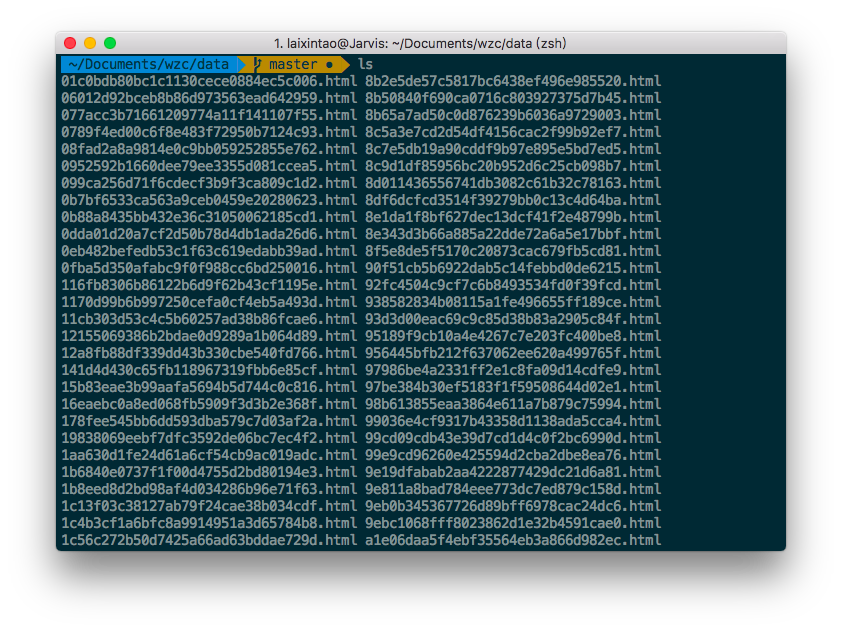
= 9e107d9d372bb6826bd81d3542a419d6

即使在原文中作一个小变化（比如用c取代d）其散列也会发生巨大的变化：

MD5("The quick brown fox jumps over the lazy **c**og")

= 1055d3e698d289f2af8663725127bd4b

下图为存储的文件截图。



5.5 Web服务器模块

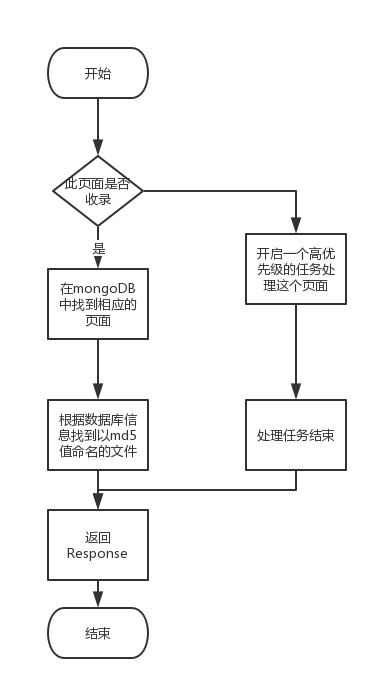
5.5.1 Web服务器的实现

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

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。流程图如下。



5.5.2 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。

5.6 与Docker有关的部署

5.6.1 制作一个镜像

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 [Union FS](https://en.wikipedia.org/wiki/Union_mount) 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

5.6.2 开启一个容器

所需要的命令主要为 docker run。

例如，下面的命令输出一个 “Hello World”，之后终止容器。

$ sudo docker run ubuntu:14.04 /bin/echo 'Hello world'

Hello world

这跟在本地直接执行 /bin/echo 'hello world' 几乎感觉不出任何区别。

下面的命令则启动一个 bash 终端，允许用户进行交互。

$ sudo docker run -t -i ubuntu:14.04 /bin/bash

root@af8bae53bdd3:/#

其中，-t 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上， -i 则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如

root@af8bae53bdd3:/# pwd

/

root@af8bae53bdd3:/# ls

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

当利用 docker run 来创建容器时，Docker 在后台运行的标准操作包括：

检查本地是否存在指定的镜像，不存在就从公有仓库下载

利用镜像创建并启动一个容器

分配一个文件系统，并在只读的镜像层外面挂载一层可读写层

从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去

从地址池配置一个 ip 地址给容器

执行用户指定的应用程序

执行完毕后容器被终止

5.6.3 镜像的制作及托管

用户无需登录即可通过 docker search 命令来查找官方仓库中的镜像，并利用 docker pull 命令来将它下载到本地。

例如以 centos 为关键词进行搜索：

$ sudo docker search centos

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

centos The official build of CentOS. 465 [OK]

tianon/centos CentOS 5 and 6, created using rinse instea... 28

blalor/centos Bare-bones base CentOS 6.5 image 6 [OK]

saltstack/centos-6-minimal 6 [OK]

tutum/centos-6.4 DEPRECATED. Use tutum/centos:6.4 instead. ... 5 [OK]

...

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建。 官方的镜像说明是官方项目组创建和维护的，automated 资源允许用户验证镜像的来源和内容。

根据是否是官方提供，可将镜像资源分为两类。 一种是类似 centos 这样的基础镜像，被称为基础或根镜像。这些基础镜像是由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。 还有一种类型，比如 tianon/centos 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名称前缀。可以通过前缀 user\_name/ 来指定使用某个用户提供的镜像，比如 tianon 用户。

另外，在查找的时候通过 -s N 参数可以指定仅显示评价为 N 星以上的镜像。

下载官方 centos 镜像到本地。

$ sudo docker pull centos

Pulling repository centos

0b443ba03958: Download complete

539c0211cd76: Download complete

511136ea3c5a: Download complete

7064731afe90: Download complete

用户也可以在登录后通过 docker push 命令来将镜像推送到 Docker Hub。

6 系统的部署与维护

6.1 在Linux内核的机器上使用容器部署

6.2 部署一个集群

6.3 监控集群的健康度

7 总结与展望

7.1 总结

7.2 展望

参考文献

致谢