基于无界面浏览器技术的单页应用搜索引擎优化

摘要

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。本文提出的系统采用分布式的爬虫，使用任务队列分配网页的采集任务，任务调度灵活，扩展性高。针对搜索引擎的爬虫单独设置网络服务器，SEO针对性高。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，并且部署起来方便快速。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。

关键词：搜索引擎优化，单页应用，分布式爬虫，NNoSQL数据库

Single Page Application Search Engine Optimize Based on Headless Broswer

ABSTRACT

In recent years, the development of a lot of use of single-page application development web development framework. Single-page applications using Ajax technology, do not refresh the page in the case can update the contents of the page. To the user experience has brought a great upgrade.

But then also have a lot of problems, with a single page application development web page is the biggest pain point is poor search engine. Since the content of the page is based on executing the JavaScript script within the page, use Ajax to populate the contents of the page. So when the search engine crawler requests page content, the single page application can respond with only one HTML and JavaScript script. Before the JavaScript script, there is no real content on the page.

This article proposes a distributed crawler based on the interfaceless browser technology to parse its own web pages, populate the content, and submit a complete web page to the search engine. In order to optimize the search engine included. The system proposed in this paper uses a distributed crawler, the use of task queue allocation of web collection tasks, task scheduling flexibility, high scalability. For the search engine crawler alone set up a network server, SEO targeted high. The system proposed in this paper, the use of container deployment, and system-independent, compatible, and deployed quickly and easily. Using distributed task queue, the modules in the form of components with each other, scalability. Can be deployed on a cluster or a single server, depending on the size of the site.

**Key words:** Single Page Application, SEO, Distributed task queue

目录

基于无界面浏览器技术的单页应用搜索引擎优化 1

摘要 1

Single Page Application Search Engine Optimize Based on Headless Broswer 2

ABSTRACT 2

目录 4

1 绪论 7

1.1 引言 7

1.2 课题研究内容 7

1.3 本系统的创新 8

1.4 本章小结 9

2 相关技术 10

2.1 无界面浏览器技术 10

2.2 分布式爬虫技术 10

2.3 NoSQL数据库技术 10

2.3.1 NoSQL数据库的概念 10

2.3.2 NOSQL的优势 11

2.4 MD5加密技术 11

2.5本章小结 12

3系统分析 13

3.1 可行性分析 13

3.2 问题定义 13

3.3 需求分析 13

3.3.1 功能需求 13

3.3.2性能需求 14

3.4 开发环境 14

3.5 本章小结 14

4 单页应用SEO优化系统的设计与实现 15

4.1 系统的结构设计 15

4.1.2 系统设计 15

4.1.2 系统代码的组织 16

4.2 详细设计 17

4.2.1 爬虫模块 17

4.2.2 页面的JavaScript执行模块 17

4.2.3 网页的存储模块 17

4.2.4 响应请求的web服务器模块 18

4.2.5 使用docker部署 18

4.2.6 在服务器上的部署 18

5 系统实现 19

5.1 设计平台搭建 19

5.1.1 开源工具选择 19

5.1.2 设计平台配置 19

5.1.3 生产环境服务器的要求 19

5.2 爬虫模块 20

5.2.1 爬虫模块的实现 20

5.3 Phantomjs模块 29

5.3.1 phantomjs模块的实现 29

5.3.2 Phantomjs的服务方式 31

5.4 网页的存储模块 32

5.4.1 网页存储模块的实现 32

5.4.2 网页更新的难点 34

5.5 Web服务器模块 35

5.5.1 Web服务器的实现 35

5.5.2 对于未收录网站的处理 38

5.6 与Docker有关的部署 38

5.6.1 制作一个镜像 38

5.6.2 开启一个容器 38

5.6.3 镜像的制作及托管 39

6 系统的部署与维护 41

6.1 在Linux内核的机器上使用容器部署 41

6.2 部署一个集群 41

6.3 监控集群的健康度 41

7 总结与展望 42

7.1 总结 42

7.2 展望 42

参考文献 43

致谢 44

1. 绪论

1.1 引言

单页Web应用（single page web application，SPA），就是只有一张Web页面的应用，是加载单个HTML 页面并在用户与应用程序交互时动态更新该页面的Web应用程序。单页应用大量通过JavaScript脚本用AJAX技术与网页进行交互，改变部分页面而不必重新刷新整个页面。能给用户体验带来很大提升。

单页应用并不是一个全新发明的技术，而是随着互联网的发展，越来越受web开发者欢迎，单页应用的体验可以模拟原生应用，一次开发，多端兼容，效果酷炫，节省成本。然而，由于单页应用基本全部使用JS，受制于SEO效果，目前国内使用单页应页技术的网站还是少之又少。

目前，使用单页应用的项目来解决SEO问题的主要方式是，后端渲染出另外一套页面出来。前端的负载均衡服务器判断请求来自真实用户还是搜索引擎的爬虫。如果是真实用户，就正常使用单页应用。如果是搜索引擎的爬虫，则返回渲染的模板。

这样做的弊端显而易见。开发成本比较高，相当于重新开发一套网页出来，同样也需要额外的精力去维护。而且这样抓取到的页面与原生的单页应用页面有很大不同，可能会遭到搜索引擎的惩罚，效果也并不是很好。

本文提出，使用爬虫的方式对单页应用的网站进行索引。用以响应搜索引擎的请求。换句话说，本文所提出的系统，相当于是建立一个镜像网站，这个系统能保证对原站的收录及时，对搜索引擎响应迅速，在完全不影响原站的情况下，彻底、完美地解决单页应用的搜索引擎优化问题。

1.2 课题研究内容

本文讨论的是针对单页应用的搜索引擎优化问题。搜索引擎优化（英语：search engine optimization，[缩写](https://zh.wikipedia.org/wiki/%E7%B8%AE%E5%AF%AB)为SEO），是一种通过了解[搜索引擎](https://zh.wikipedia.org/wiki/%E6%90%9C%E5%B0%8B%E5%BC%95%E6%93%8E)的运作规则来调整[网站](https://zh.wikipedia.org/wiki/%E7%B6%B2%E7%AB%99)，以及提高目的[网站](https://zh.wikipedia.org/wiki/%E7%B6%B2%E7%AB%99)在有关搜索引擎内排名的方式。一般来说，一个网站80%的流量都会来自于搜索引擎。很多研究表明，搜索引擎的用户往往只会留意搜索结果最前面的几个条目，所以，网站在搜索引擎的结果列表中显示的位置，对一个网站来说就显得非常重要，尤其是对以流量展示为主的网站，就更成了重中之重。

要做好搜索引擎的优化，首先需要保证的一点就是，网站的内容能正常被搜索引擎收录。只有搜索引擎能知道网站的内容，才会被索引，在搜索结果的列表页中呈现给用户。

而在单页应用的搜索引擎优化中，这部分就有很多额外的困难。因为单页应用的HTML文档基本上没有内容，而只是大量的JavaScript代码，通过执行代码发送AJAX请求，然后填充页面的内容。而搜索引擎只会下载页面的HTML文档，并不会去执行JavaScript脚本，所以对于搜索引擎来说，抓到的就是毫无意义的脚本。这样搜索引擎索引到的数据毫无意义，导致网站不可能被用户通过搜索引擎搜索到。

本文提出，使用部署一个分布式爬虫系统的方法，运行一个动态的镜像网站。基本上可以完全“自动”地完成SEO的任务。通过定时更新网页，索引到数据库，判断请求来源，分发响应内容的方式，来针对搜索引擎的爬虫返回特定的响应，解决搜索引擎优化的问题。

1.3 本系统的创新

本文提出，使用分布式爬虫的技术通过构建动态的镜像网站来解决但页面应用的SEO问题。通常，爬虫技术用于数据采集，搜索收录等场景。但是配合无界面浏览器技术，可以打破爬虫只能下载页面内容的局限性，能够使用JavaScript解释器引擎执行页面内的JavaScript代码，从而获得真实的、有内容的页面，而不仅仅是无意义的HTML文档内容。

本文提出的系统采用容器的方式部署，屏蔽了不同操作系统之间底层的不同，从而使部署变得简单。组件与组件之间通过网络地址加端口的方式通讯，非常灵活，采用分布式的爬虫和任务队列，扩展性高。

所有组件的设计方式，保证了本系统的高可用性，适用于各种不同规模的网站。对于小型网站，可以采用单台甚至与原网站使用同一台服务器的方式。对于大型网站，可以使用两台到几十台的服务器集群，专门解决SEO的问题。

1.4 本章小结

本章讨论了单页应用的搜索引擎优化问题所面临的挑战，目前的解决方案，以及本文提出的方法。从总体上定义了问题，是本文的最终目的。

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)。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。

可以认为，PhantomJS就是一个浏览器，只是没有界面而已。它是我们能够执行HTML文档中的JavaScript的基础保证。

PhantomJS提供了集成度比较高的命令行工具，也可以执行一个特定的JavaScript脚本。借此，我们可以制作一个JavaScript脚本，监听特定的一个端口，以服务的方式运行。

2.2 分布式爬虫技术

网络爬虫（英语：web crawler），也叫网络蜘蛛（spider），是一种用来自动浏览[万维网](https://zh.wikipedia.org/wiki/%E4%B8%87%E7%BB%B4%E7%BD%91)的[网络机器人](https://zh.wikipedia.org/w/index.php?title=%E7%BD%91%E7%BB%9C%E6%9C%BA%E5%99%A8%E4%BA%BA&action=edit&redlink=1)。其目的一般为编纂[网络索引](https://zh.wikipedia.org/w/index.php?title=%E7%BD%91%E7%BB%9C%E7%B4%A2%E5%BC%95&action=edit&redlink=1)。[网络](https://zh.wikipedia.org/w/index.php?title=%E7%BD%91%E7%BB%9C%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E&action=edit&redlink=1)[搜索引擎](https://zh.wikipedia.org/wiki/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E)等站点通过爬虫软件更新自身的[网站内容](https://zh.wikipedia.org/w/index.php?title=%E7%B6%B2%E7%AB%99%E5%85%A7%E5%AE%B9&action=edit&redlink=1)或其对其他网站的索引。网络爬虫可以将自己所访问的页面保存下来，以便搜索引擎事后生成[索引](https://zh.wikipedia.org/w/index.php?title=%E7%B4%A2%E5%BC%95_(%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E)&action=edit&redlink=1)供用户搜索。

分布式爬虫，就是利用多台服务器或集群进行网站抓取的任务。分布式爬虫的核心是一台负责任务分发的服务器，其他服务器从这台服务器取得任务，执行任务然后保存结果。分布式的爬虫可以减少任务的等待时间，从而加快网站的收录以及更新的速度。分布式爬虫的集群是可扩展的，只要保证负责任务队列的服务器唯一，即可以添加多台工作机。

2.3 NoSQL数据库技术

2.3.1 NoSQL数据库的概念

随着web2.0的快速发展，非关系型、分布式数据存储得到了快速的发展，它们不保证关系数据的ACID特性。[NoSQL](http://nosql-database.org/)概念在2009年被提了出来。NoSQL最常见的解释是“non-relational”，“Not Only SQL”也被很多人接受。（“NoSQL”一词最早于1998年被用于一个轻量级的关系数据库的名字。）

NoSQL被我们用得最多的当数key-value存储，还有其他的文档型的、列存储、图型数据库、xml数据库等。在NoSQL概念提出之前，这些数据库就被用于各种系统当中，但是却很少用于web互联网应用。比如cdb、qdbm、bdb数据库。

2.3.2 NOSQL的优势

(1) 易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

(2) 大数据量，高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

(3) 灵活的数据模型

NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的web2.0时代尤其明显。

(4) 高可用

NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

2.4 MD5加密技术

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。

MD5算法可以保证，如果两个网页内容存在不同，生成的MD32位数据就会不同。在本文提出的系统中，使用这个原理作为网页比较的算法，从而保证，我们可以快速地验证此网页与上一次的网页相比，是否更新过。还可以用来索引，页面内容可以存放在数据库中，占用数据库的空间。通过MD5值将网页文件与数据库内容索引起来，即可以使用文件存放网页。

2.5本章小结

本章简述了本文所提出的系统所涉及的一些技术，通过了解这些技术的原理，可以更好地理解本系统。正是有了这些技术的支撑，本系统的实现才成为可能。在接下来的几章，将讨论如何运用这些技术，来解决单页应用的搜索引擎优化难题。

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

3.3.1 功能需求

系统要保证基本的功能，即可以对源网站自动进行抓取。此外，如果提供sitemap，让系统可以根据sitemap进行抓取。从而打到与搜索引擎一致的效果。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。系统可以将网页提交给搜索引擎，被搜索引擎正常收录。系统提供方便，直观的管理界面，方便监控和维护各个服务。守护进程，确保程序的正常运行。

3.3.2性能需求

要保证系统的可用性，最基本的是保证系统能够对新产生的网页进行快速收录。对搜索引擎爬虫发送过来的请求，要保证及时响应，对于已经收录的网页，这点非常简单，只要读取数据库一次，读取文件系统一次，即可以完成对请求的响应。还要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。初次之外，系统还应该保证以下几点。

1. 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。
2. 易监控，提供监控页面，可以快速、直观地查看系统运行情况。
3. 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。

1. 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。

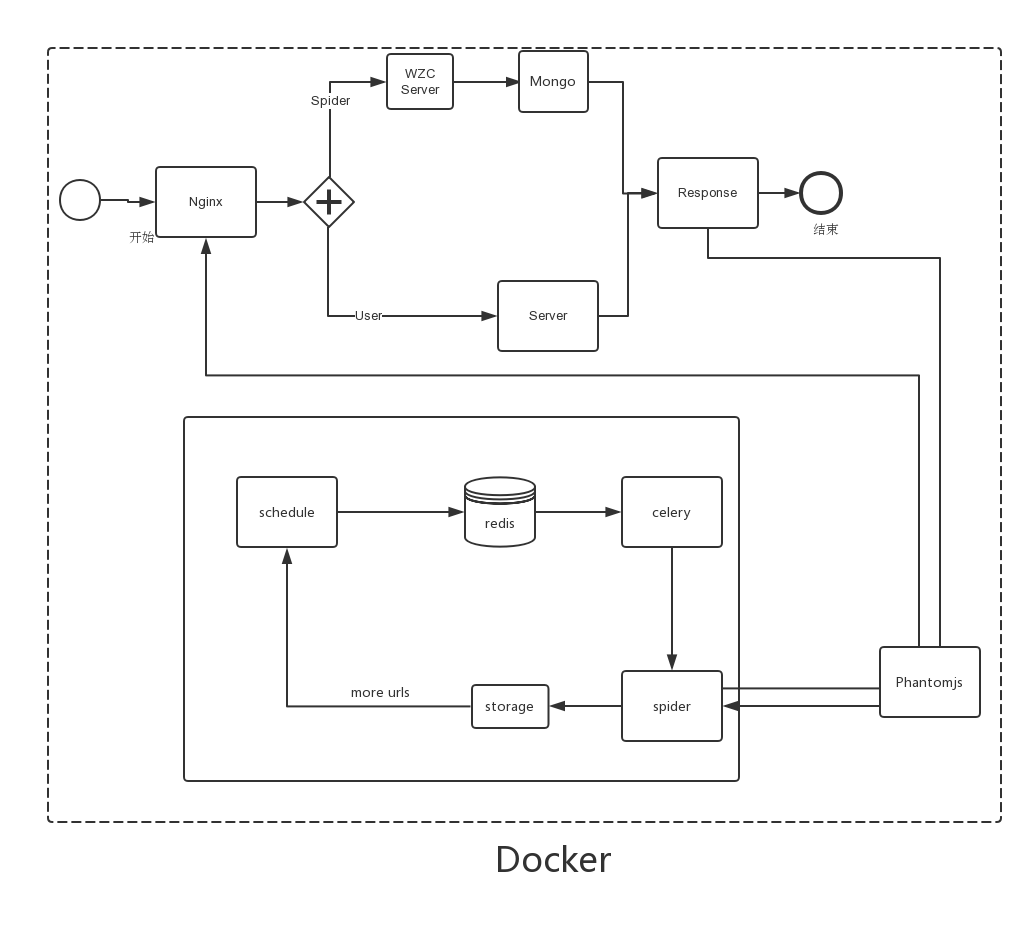


图4-1 系统总体架构图

4.1.2 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4-2所示：

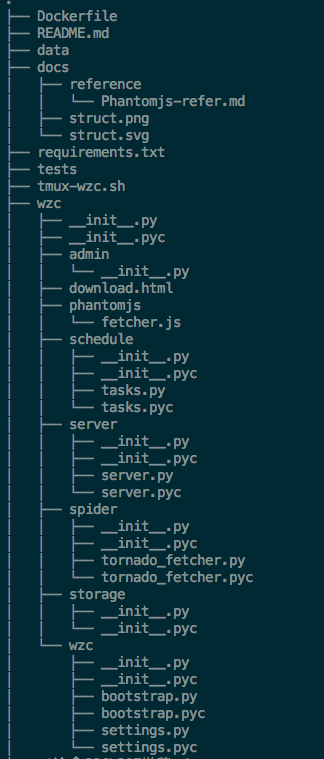


图4-2 系统代码的组织

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。

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。
3. 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）。
4. 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

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。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 |

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，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。

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。任务队列的工作流程，如图5-1所示。

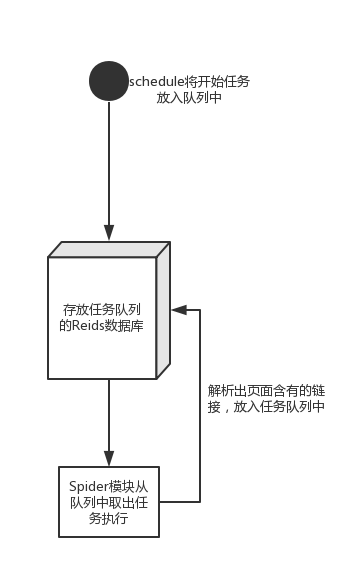


图 5-1 任务队列的工作流程

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。工作机处理任务的流程如图5-2所示。

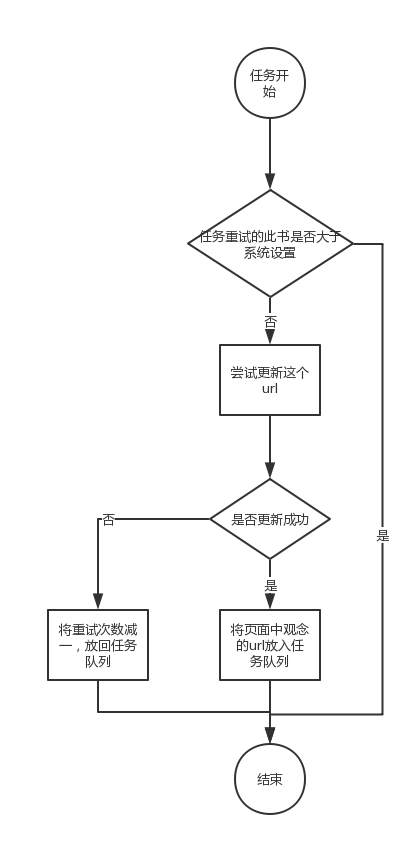


图 5-2

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。协程与线程的关系，如图5-2所示。



图 5-3 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。

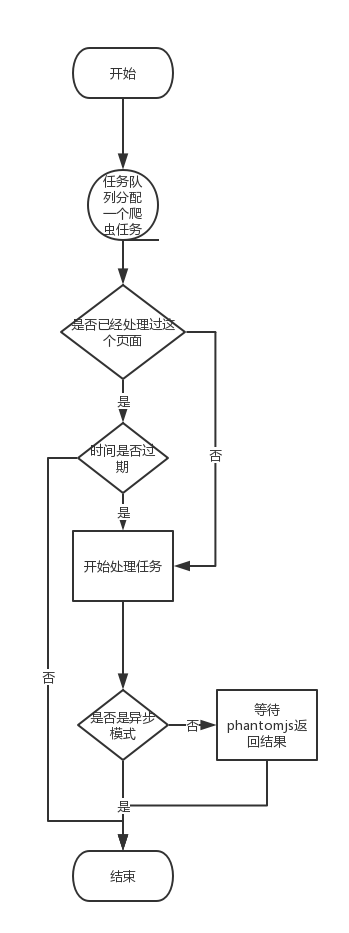


图 5-4 任务队列和工作机的工作流程

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。

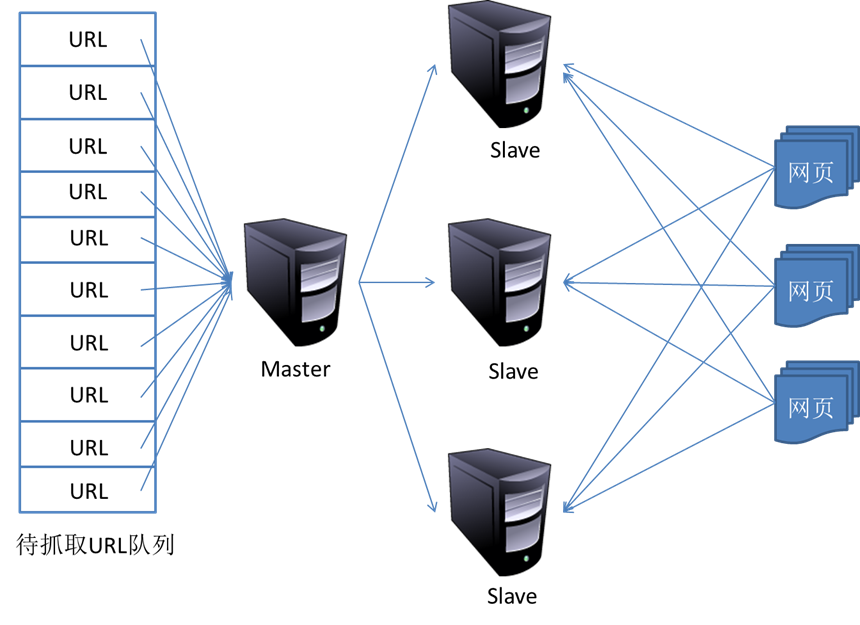


图 5-5 分布式的爬虫集群架构

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。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。

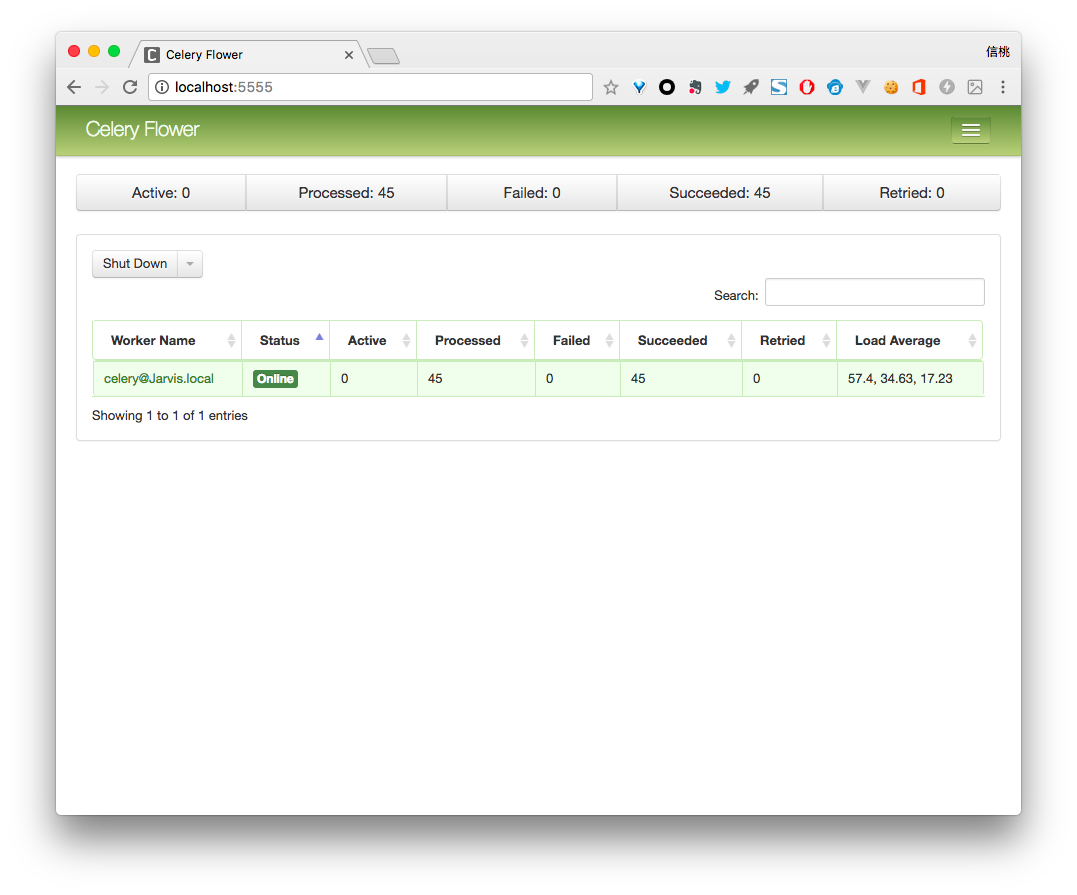


图 5-6 监控界面的首页

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。

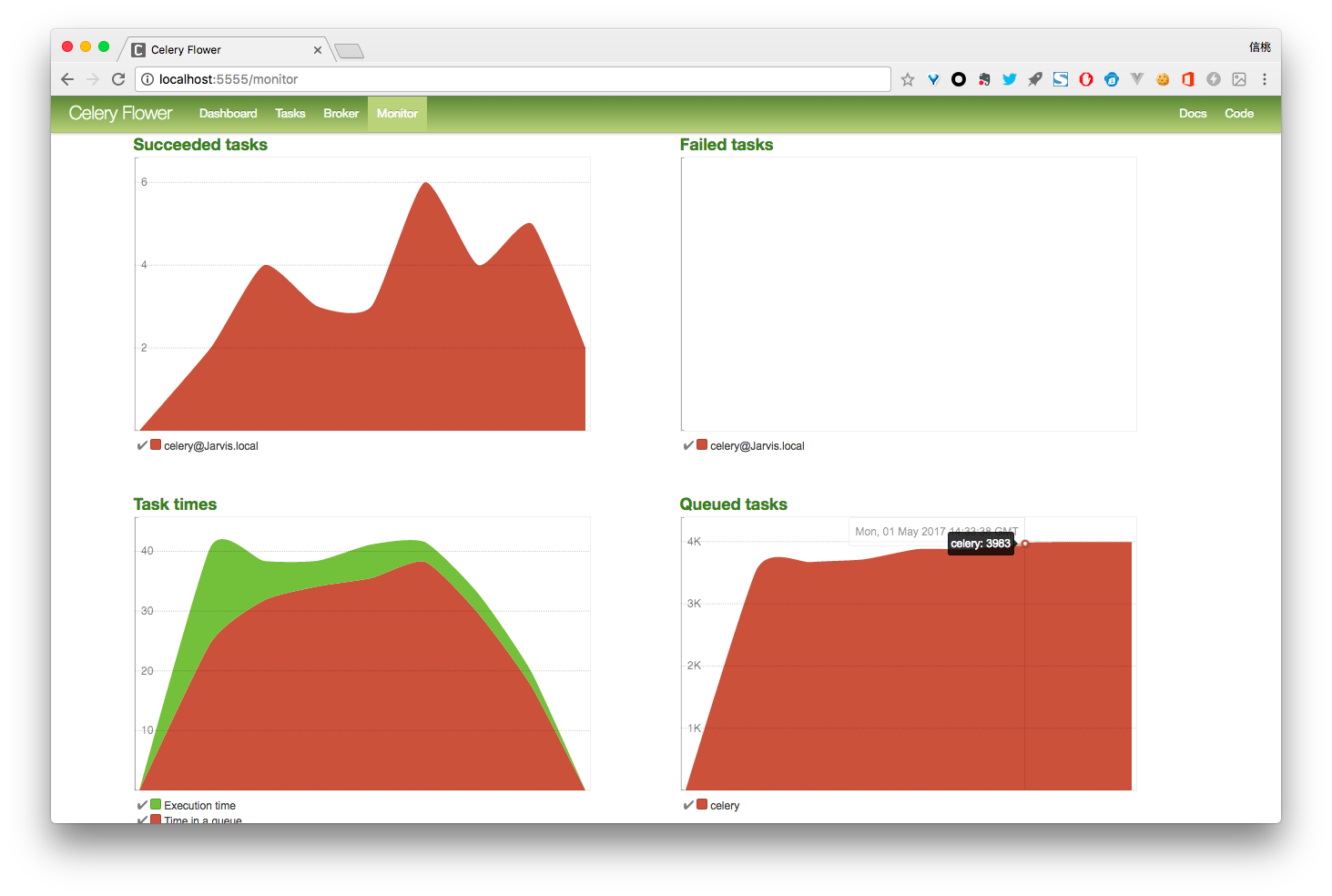


图 5-7 任务队列的运行情况监控界面

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）。

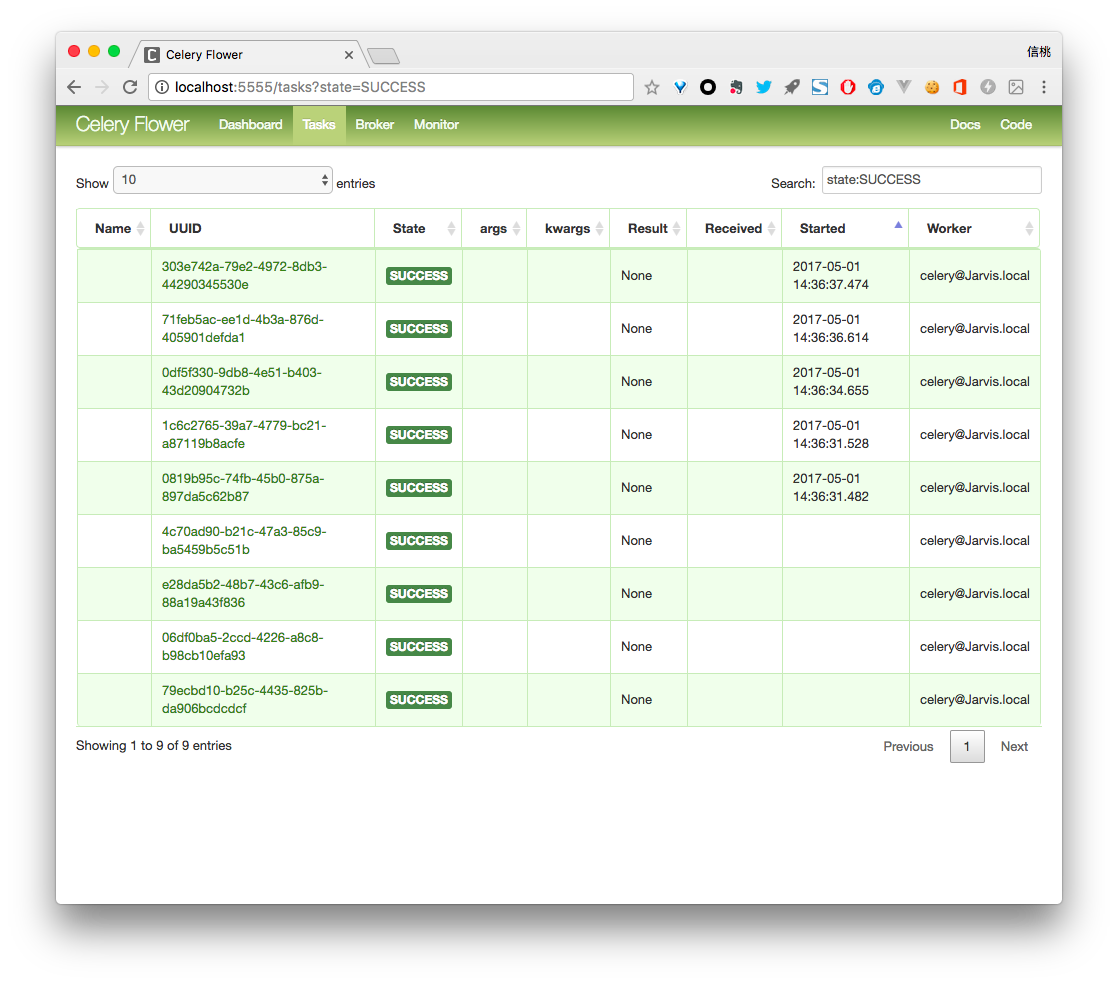


图 5-8 任务详情监控界面

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。

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。

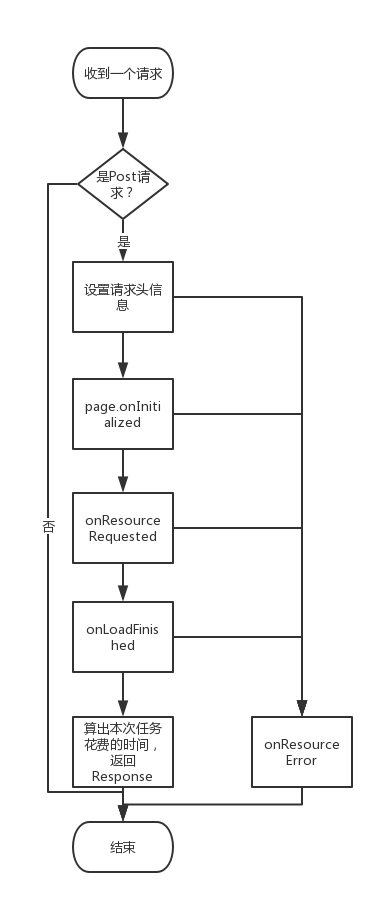


图 5-9 PHANTOMJS执行的流程图

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);

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。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）。

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。如下图所示。

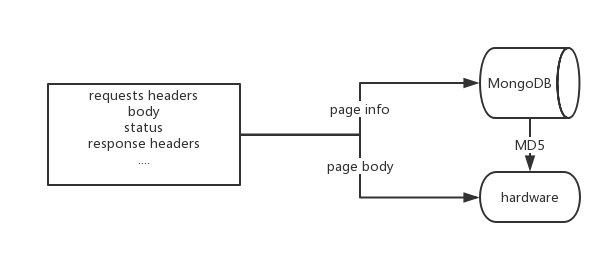


图5-10 存储模块的存储过程

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。

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。

使用MD5值记录来追踪网页的变化，可以保证速度以及唯一性。

首先，MD5算法是一个摘要算法，根据文件的内容生成一个16字节的字符。生成过程中基本上是一系列的逻辑运算，将所有的数据分组，应用到算法，速度能够得到保证。

其次，文件中所有的字符都会参与MD5的生成过程，可以说，如果有两座图书馆，两座图书馆其中只有一本书的一个文字不同，那么生成的MD5值也是不同的。这样，唯一性就得到了保证。

HTML文档在文件系统中存储的形式如5-11所示。

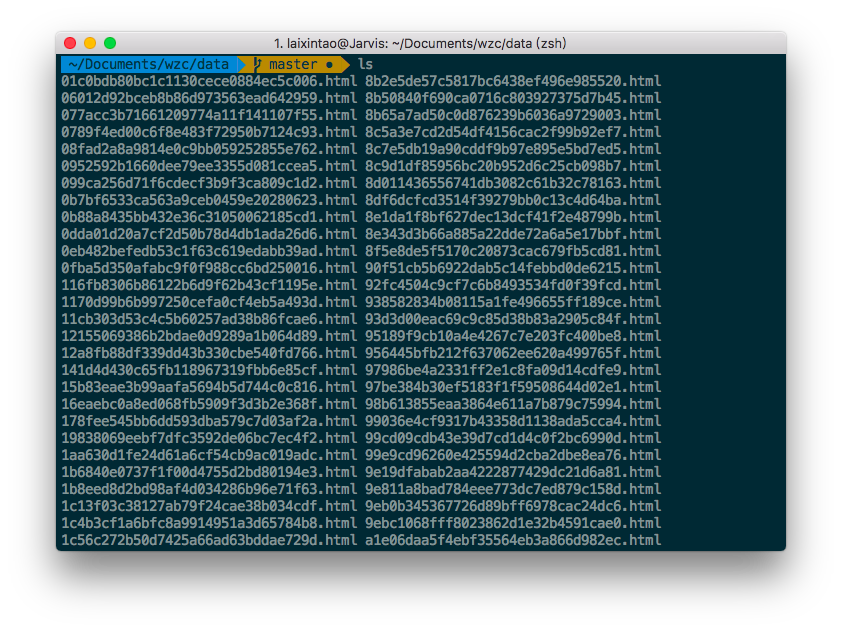


图5-11 网页存储在文件系统中的形式

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。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。

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。

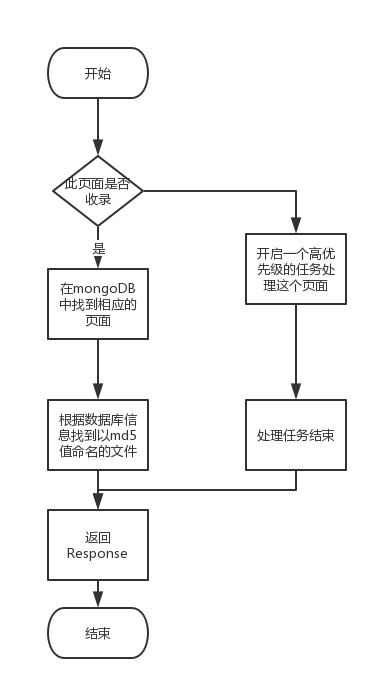


图5-12 Web服务器处理请求流程

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 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。选择了容器而不是虚拟机，是因为Hyper-V、KVM和Xen等虚拟机管理程序都“基于虚拟化硬件仿真机制。这意味着，它们对系统要求很高。然而，容器却使用共享的[操作系统](http://product.it168.com/list/b/0501_1.shtml)。这意味着它们在使用系统资源方面比虚拟机管理程序要高效得多。容器不是对硬件进行虚拟化处理，而是驻留在单单一个Linux实例上。这反过来意味着，你可以“丢弃没有用的99.9%的虚拟机垃圾，剩下一个小巧简洁的胶囊式容器，里面含有你的应用程序。

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。

5.6.2 制作容器

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。

* + 1. 开启一个容器
    2. 镜像的制作和托管

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

6.4 本章小结

7 总结与展望

7.1 总结

本文所提出的系统，在结合任务队列、分布式爬虫、无界面浏览器技术、Web服务器的情况下，提供了一种可用于服务单页面应用搜索引擎优化的系统，该系统能够从目标的单页应用网站抓取页面，并且解析HTML文档中的JavaScript脚本，从而获得有真是内容的网页，用于被搜索引擎收录。

系统使用了很多复杂的技术，但是从总体上来说，各个组件可以比较良好的相互支持，以服务的形式通过监听端口运行在服务器上。使用容器技术解决了因为应用的技术复杂多样而带来的部署困难的问题。在保证高可用性的同时，有具备较高的扩展能力。

7.2 展望

系统中应用的一部分模块（例如Web服务器）没有监控机制，虽然这一部分出错的概率很小，而且恢复起来也比较方便，但是还是应该有一些协程来帮助监控系统的运行状况，在出现意外的情况下，能马上通知系统管理员。

参考文献

致谢