Simple Search Engine

简介

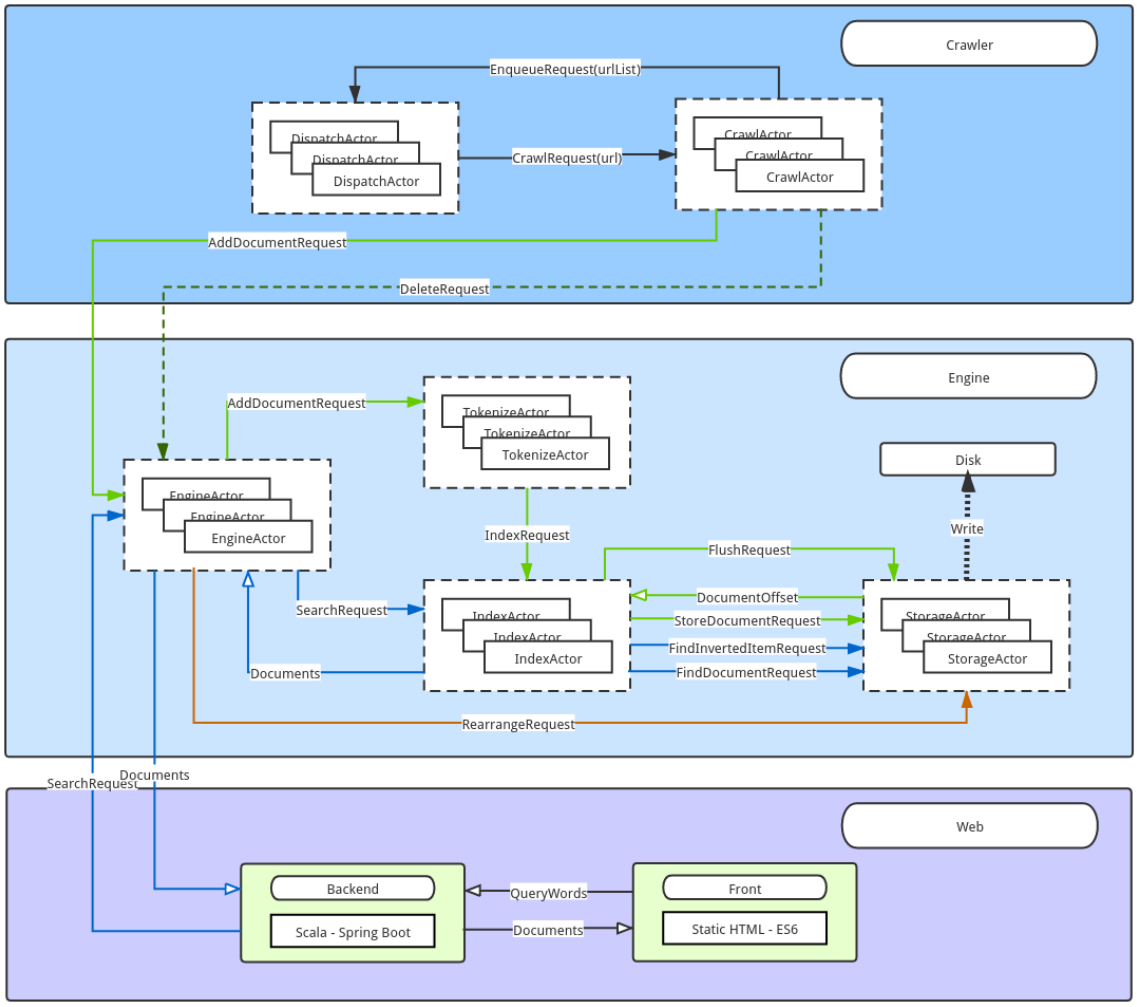
本项目实现了一个简单的搜索引擎，下面简单介绍要点，详细技术要点参考`技术文档`

- 功能要点： 查询网页、异步爬虫、添加文档、缓存快速删除文档、搜索文档、其他数据维护功能
- 技术要点：
    1. 项目包含 web、crawler、engine 三大模块，全部由 scala 编写，使用akka提供Actor并发模型支持
    2. web模块：由 Spring Boot 后端和静态 HTML 前端（原生ES6）组成
    3. crawler模块：基于Actor模型的异步爬取和异步添加/更新文档、定期搜集、种子搜索+维护链接列表策略
    4. engine模块：Actor并发与Future异步方式、缓存快速删除文档设计、缓存索引表和url库、文件读写锁
    5. 其他模块：定时爬取任务，定时缓存持久化，分词功能
- 搜索引擎过程要点：
    1. web搜集：异步爬虫、定期搜集、种子搜索+维护url列表策略
    2. 预处理：分词、正则提取url、正则提取正文
    3. 维护文件：网页库（文档信息、文档正文）、索引表（文档id、内容hash、网页库文件偏移量）、倒排表（词、文档id、文档位置列表）、文档计数表（总文档数、各文档词数）、url库（url、正文hash）
    4. 查询服务：包含信息列表三要素（标题、URL、摘要）、字符串切份生成摘要
    5. 查询排序：BM25排序算法

顶层设计

本项目包含三大模块：web前后端模块（web）、爬虫和爬虫调度模块（crawler）、搜索引擎模块（engine）

数据流图分析



注：

1. 为了便于分析，我将数据流动分为三个颜色：绿色指添加文档、蓝色指搜索文档、红色指缓存写入和整理数据
2. 空心箭头是指Future的异步返回，实心箭头是Actor的请求

Crawler

1. 由DispatchActor向CrawlActor发起爬取请求
2. CrawlActor收到请求后进行爬取，再从文件内容中解析出下一个url列表。
3. 将url列表发送至DispatchActor入队列，同时检查正文是否改变，再决定是否发送Delete请求再发送AddDocument请求

Engine

为了便于分析，我将数据流动分为三个颜色：绿色指添加文档、蓝色指搜索文档、红色指缓存写入和整理数据

- 添加文档：
    1. 一般由CrawlActor请求添加文档，也可以将Engine作为屏蔽底层的数据库，由用户手动添加
    2. TokenizeActor提取正文，分词，将结果发送至IndexActor
    3. IndexActor受到分词结果和正文后，异步发送正文给StorageActor，获得该文档存储时的文件偏移量；同时分析出该文档对应的倒排表存入缓存；更新文档和词数计数；最后条件判断进行缓存写入

4. StorageActor进行网页库、索引表、倒排表、文档计数表的缓存写入和读取；同时进行缓存快速删除文档、整理所有数据文件

- 搜索文档：
  1. 一般由web后端请求搜索文档，也可以作为数据库手动查找
  2. 首先由EngineActor调用TokinzeActor进行搜索语句的分词
  3. IndexActor对分词结果进行遍历，向StorageActor发起查找倒排表表项的请求；同时向StorageActor发起查找正文的请求；最后结合文档和词数计数、倒排表表项和文档信息计算BM25的排序值，进行排序
  4. 查询结果最终回到后端或用户手中
- 缓存写入和整理数据：
  1. 包括网页库、索引表、倒排表、文档计数表的缓存写入和整理

Web

- 后端：由Spring Boot编写，接口包括查询、手动启动爬取、手动刷新缓存
- 前端：静态HTML，同时使用ES6（fetch）进行数据交互

其他

- 定时任务：由Spring Boot提供静态定时任务框架，用于定时启动爬取和刷新缓存
- 一些工具类
- 一些数据实体和Actor间通信协议
- 配置项

详细设计

Actor并发模型

在Engine和Crawler模块中大量使用Actor模型进行并发控制，对整个业务逻辑提供并发的抽象，能够快速编写高并发的程序。

Actor 的基础就是消息传递，一个 Actor 可以认为是一个基本的计算单元，它能接收消息并基于其执行运算，它也可以发送消息给其他 Actor。Actors 之间相互隔离，它们之间并不共享内存。

Actor 本身封装了状态和行为，在进行并发编程时，Actor 只需要关注消息和它本身。而消息是一个不可变对象，所以 Actor 不需要去关注锁和内存原子性等一系列多线程常见的问题。

所以 Actor 是由状态（State）、行为（Behavior）和邮箱（MailBox，可以认为是一个消息队列）三部分组成：

1. 状态：Actor 中的状态指 Actor 对象的变量信息，状态由 Actor 自己管理，避免了并发环境下的锁和内存原子性等问题。
2. 行为：Actor 中的计算逻辑，通过 Actor 接收到的消息来改变 Actor 的状态。
3. 邮箱：邮箱是 Actor 和 Actor 之间的通信桥梁，邮箱内部通过 FIFO（先入先出）消息队列来存储发送方 Actor 消息，接受方 Actor 从邮箱队列中获取消息。

Actor 模型特点在于：

1. 对并发模型进行了更高的抽象。
2. 使用了异步、非阻塞、高性能的事件驱动编程模型。
3. 轻量级事件处理（1 GB 内存可容纳百万级别 Actor）。

下面是一个本项目中Actor的简单例子：

```
class EngineActor extends Actor with ActorLogging {

  // 此处进行请求的模式匹配，并进行相关操作
  override def receive: Receive = {
    case AddRequest(response) =>
      val documentId: Long = Engine.getDocumentId
      Engine.documentUrlToId(response.getUri.toString) = documentId
      Engine.documentIdToUrl(documentId) = response.getUri.toString
      Engine.tokenizeActor ! TokenizeDocumentRequest(documentId, response)

    case SearchRequest(sentence) =>
      ...

    case AsyncSearchRequest(sentence, callback) =>
      ...

    case DeleteRequest(url) =>
      ...
  }

}
```

Future异步模型

在Actor模型提供异步并发，并且各种操作间的同步控制时，需要Future模型进行同步或者异步的描述。

举例来说，Crawler在请求页面的时候使用了Future异步爬取；在IndexActor中取得文档偏移量是与其他操作间无关的，所以直接异步的使用Future；同步的情况出现在IndexActor的搜索过程，需要首先取得倒排表才能进行之后的操作。

Future提供了一套高效便捷的非阻塞并行操作管理方案。其基本思想很简单，所谓Future，指的是一类占位符对象，用于指代某些尚未完成的计算的结果。一般来说，由Future指代的计算都是并行执行的，计算完毕后可另行获取相关计算结果。以这种方式组织并行任务，便可以写出高效、异步、非阻塞的并行代码。

默认情况下，future和promise并不采用一般的阻塞操作，而是依赖回调进行非阻塞操作。为了在语法和概念层面更加简明扼要地使用这些回调，Scala还提供了flatMap、foreach和filter等算子，使得我们能够以非阻塞的方式对future进行组合。当然，future仍然支持阻塞操作——必要时，可以阻塞等待future。

下面是两个本项目中Future的简单例子，一个同步一个异步：

```
// 这里开始异步请求，返回的是一个将返回文档列表的Future
val documentsFuture: Future[List[Document]] =
  (Engine.indexActor ? IndexSearchRequest(words, xs =>
())).mapTo[List[Document]]

// 这里直接等待返回值
val documents: List[Document] =
  Await.result(documentsFuture, Config.DEFAULT_AWAIT_TIMEOUT)
```

```
// 这里开始异步请求，返回的是一个装有长整型的Future
val offsetFuture: Future[Long] =
  (Engine.storageActor ? StoreDocumentRequest(documentHash,
documentInfo)).mapTo[Long]

// 这里异步处理Future，当成功时向缓存的索引表中添加数据
offsetFuture onComplete {
  case Success(offset) =>
    Engine.indexTable(id) = (documentHash, offset)

  case Failure(exception) =>
    exception.printStackTrace()
}
```

排他锁 & 共享锁 & 基于乐观锁的原子操作

当缓存写入文件时，不免会有两个线程（抽象的说是Actor）对同一个文件进行同时的写写/读写操作。这样的操作有很大可能出现错误。于是引入读锁和写锁进行响应处理。

```
// 建立writer
val writer = new RandomAccessFile(file, "rw")
val channel: FileChannel = writer.getChannel
val offset: Long = writer.length()

// 使用排他锁进行写操作
val xLock: FileLock = channel.lock()
writer.seek(offset)
writer.write(s"${documentInfo.title}${Config.CRLF}".getBytes())
writer.write(s"${documentInfo.url}${Config.CRLF}".getBytes())
writer.write(s"${documentInfo.content}${Config.CRLF + Config.CONTENT_SPLITTER +
Config.CRLF}".getBytes())

// 向发起请求的Actor返回值，同时释放锁关闭writer
sender ! offset
xLock.release()
writer.close()
```

在添加文档时，会进行文档数和词汇数的统计，这里也会有并发的写写/读写问题。解决方法是使用原子操作，即底层实现为乐观锁。

```
// 例如总词数统计，这是总词数变量，定义为AtomicLong
val totalWordCount: AtomicLong = new AtomicLong(0)

// 原子添加数量
Engine.totalWordCount.addAndGet(wordCount)
```

爬取策略

简要来说，爬取策略是定期搜集和种子搜索+维护链接列表

种子搜索：在首次进行爬取时，选定某些链接作为种子进行搜索。具体来说是获取某一页面、解析出符合要求的链接列表、加入队列中。

维护链接列表：因为互联网中页面数量实在太大，同时也存在某些页面多变的情况。当首次爬取结束后，获得包含定量链接的列表，接下来的爬取任务就是定时爬取这些网页，如果内容变化（hash值变化，存储于链接表中）则删除搜索引擎中的文档，同时再次写入新的内容。

定期搜集：可在配置中设定间隔，DEMO中是每天进行爬取。由Spring Boot中的Schedule实现。

正文提取 & 摘要生成

采用正则表达式的方式进行正文提取：

1. 过滤掉js代码部分
2. 删除标签的信息，如`<p style="color: red">正文的一部分</p>`的尖括号内容
3. 最后合并重复的空格、制表符和回车

摘要生成的方法是按倒排表的词汇位置进行切分正文，以关键词为中心半径20个字符（可配置）进行切分。

BM25排序算法

`BM25`是搜索引擎的经典排序函数，用于衡量一组关键词和某文档的相关程度。

定义为：

```
                IDF * TF * (k1 + 1)
BM25 = sum ----------------------------
           TF + k1 * (1 - b + b * D / L)
```

其中`sum`是所有关键词求和，`TF`（term frequency）为某关键词在该文档中出现的词频，`D`为该文档的词数，`L`为所有文档的平均词数，`k`和`b`为常数，默认值为2.0和0.75，可以修改。

`IDF`（inverse document frequency）衡量关键词是否常见，这里使用带平滑的IDF：

```
                      总文档数目
IDF = log2( ------------------------  + 1 )
               出现该关键词的文档数目
```

数据文件设计

原始网页库

每个文档会有对应的正文hash码，这个hash码的一个功能是平均分配原始网页库的id。

如一个文章的hash码为`1654859314`，同时配置中的最大文件数量为`5`时，那么对应的原始网页库id为`1654859314 % 5 = 4`，即网页将存入`4.content`中。

原始网页库包含文档信息、文档正文和一个分隔符`{{SPLIT}}`，其中文档信息包括文档标题和链接。

例子`7.content`：

```
2020 年 1月 5 日 随笔档案 - 糖栗子 - 博客园
https://www.cnblogs.com/tanglizi/archive/2020/01/05.html

2020年1月5日随笔档案-糖栗子-博客园糖栗子順番に殴るね随笔-167,
...
沪ICP备09004260号©2004-2020博客园Poweredby.NETCoreonKubernetes

{{SPLIT}}
...
```

索引表

包含文档id、内容hash、原始网页库文件偏移量。

如`40 2038333635 82824`，指40号文档的hash值为`2038333635`，其在`0.content`原始网页库中的偏移量为`82824`

`indexTable.data`例：

```
40 2038333635 82824
41 920180810 21252
42 1096021219 19365
43 984874545 92671
44 1551583241 43029
45 1109568460 27422
46 2005139154 18550
47 789807010 33238
48 904327696 544
...
```

倒排表

每个词汇会有对应的hash码，这个hash码的一个功能是平均分配倒排表的id。

如一个词汇`北京`的hash码为`14814`，同时配置中的最大文件数量为5时，那么对应的倒排表id为`14814 % 5 = 4`，即该词汇的所有倒排表表项将存入`4.invert`中。

每个倒排表项包括：词、文档id、原始网页库的偏移量。

例：

```
强大 0 1282
强大 0 2655
注意 1 454
注意 1 1113
注意 1 1418
...
```

文档计数表

由于`BM25`排序算法需要总文档数、各文档词数等统计数量的变量。所以这些数据是必要的。

它们作为元数据存储于`metaTable.data`中，`DC`指文档数量，`WC`指总词数，`id count url`则是文档对应的词数量。

例：

```
DC2050
WC3016993
0 2324 https://www.cnblogs.com/
1 967 https://www.cnblogs.com/tanglizi/p/11515409.html
2 1048 https://www.cnblogs.com/tanglizi/
3 1289 https://news.cnblogs.com/n/667106/
4 1712 https://q.cnblogs.com/
6 7366 https://www.cnblogs.com/aggsite
7 4959 http://feed.cnblogs.com/blog/sitehome/rss
8 77 https://home.cnblogs.com/
9 3465 https://news.cnblogs.com
10 83 https://ing.cnblogs.com/
13 966 https://job.cnblogs.com/
14 2543 https://edu.cnblogs.com/
16 2647 https://www.cnblogs.com/pick/
...
```

链接库

包含了需要维护的网页链接和其正文hash，作用是判断爬取过的正文是否有变化。

每行包括`url hash`，存储与`urls.data`中：

```
https://www.cnblogs.com/shirleyya/archive/2020/04/04.html
b6fcd5b1c4e4233feb45f5147e2f9fdb5b4f54223802e66bf189de436023d8b2
https://i.cnblogs.com/EditPosts.aspx?postid=13307712
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
https://edu.cnblogs.com/
15ec30b557b24b135d89d223d989e13b31da3f7cbae62960359c211528cf398a
https://www.cnblogs.com/Dylan7/p/13295209.html#commentform
e1967dd4b108f46db95ad6fd1bb4822c86d03e7e0fe991551031dd213f35b539
https://news.cnblogs.com/n/667254/
d5c319fba34acde1164e801721500f7d0964c37d3eef06cf80c7273034af1e7c
https://www.cnblogs.com/Yunya-Cnblogs/p/13300244.html
5e67dde5789d57e6df7571254c7576e57208586136a23739a2af14202cb1b829
https://news.cnblogs.com/n/tag/%E5%BC%A0%E5%8B%87/
91a8bb9d0068b1987b3cc45a2f8fed69cf0193c9a297ec4961c6c1f88bdb51c1
https://www.cnblogs.com/miro/p/13297147.html
f7bfde1e2641fcec6b231f319d79e9b921f773e102a7d9a109fb63abd78dfb7d
https://www.cnblogs.com/skins/summergarden/bundle-summergarden-mobile.min.css
0fe0ce7024e1fd3fd9842bc9650ba76cf27b128d43149a2c481c656d6f52a144
https://www.cnblogs.com/tanglizi/archive/2019/09/10.html
8b0ab3eaa5fc975e3852c06f49894290bbc2dbf5cba4d1a8da1fcb26b1fdd55a
...
```

缓存 & 缓存快速删除设计

在内存里我们只存储了数据量较小，并且使用频率很高的索引表和文档计数表。

缓存：在内存中我们对倒排表进行了小规模的缓存，作用是对新加入的文档而言，其搜索速度会更快。因为会首先检查内存中的倒排表。

快速删除设计：在内存中还有一个称为`被删除文档id的集合`数据结构，当文档id被包含时，该文档不可被检索到。当这个集合的元素到达某一限度，或者数据刷新定时任务到达时，进行数据整理（rearrange）过程。

数据整理：即读取所有数据文件，找出未被删除的记录，然后将这些记录重新写入文件，同时维护一些数据依赖。如0号文档被删除，则`0.content`文件中将不包含0号文档，同时`indexTable`将更新偏移量。