# 爬虫技术架构及实战

## 课程内容与系统环境介绍

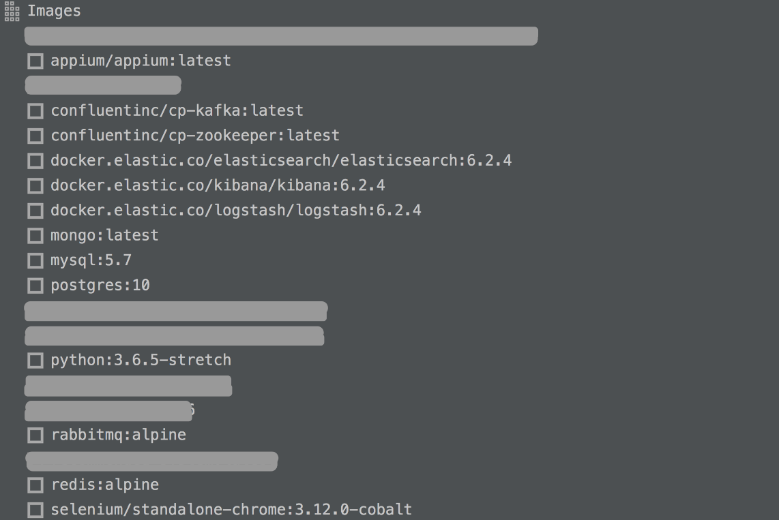
### 课程内容

1. **爬虫系统架构介绍**
2. **爬虫系统架构组件介绍**
3. **爬虫系统架构实现**
4. **项目实战**

### 系统环境介绍

演示开发环境：Centos7 + Docker18.03

使用到的Docker镜像：



使用Python环境基础的Dockerfile：

# 注意：pip.conf文件为了解决，pip安装时可能出现的如certificate verify failed的错误

FROM python:3.6.5-stretch

COPY pip.conf /root/.config/pip/

RUN pip install ipython

# 安装涉及的python第三方模块

RUN pip install requests &&\

pip install aiohttp &&\

pip install tornado &&\

pip install gevent &&\

pip install incremental &&\

pip install twisted[tls] &&\

pip install eventlet &&\

pip install dnspython &&\

pip install celery[redis] &&\

pip install django==1.11 &&\

pip install sqlalchemy &&\

pip install mongoengine &&\

pip install redis &&\

pip install six &&\

pip install pymysql &&\

pip install psycopg2 &&\

pip install lxml &&\

pip install bs4 &&\

pip install pyquery &&\

pip install wxpy &&\

pip install pika &&\

pip install pycurl &&\

pip install selenium &&\

pip install Appium-Python-Client

# 为celery启动设置用户和用户组，当序列化是pickle等类型时，启动celery时，必须指定非root用户运行. 创建一个celery的用户和用户组

RUN groupadd --gid 800 celery \

&& useradd --uid 800 --gid celery --shell /bin/bash --create-home celery

# 安装confluent-kafka时, 需要安装基于c语言的模块librdkafka

# RUN apt-get install git # 如果没有git，需要先安装.

RUN git clone https://github.com/edenhill/librdkafka.git &&\

cd librdkafka &&\

./configure &&\

make &&\

make install &&\

pip install confluent-kafka

# 我们自己开发的爬虫系统架构, 在使用时可以作为第三方模块安装到python中.

# 安装spidersystem时，先将spidersystem文件夹与setup.py拷贝到当前路径的my\_moudle文件夹下

# COPY my\_moudle /my\_moudle/

# RUN cd /my\_moudle &&\

# python setup.py install

## 1. 爬虫系统架构介绍

### 爬虫系统架构了解

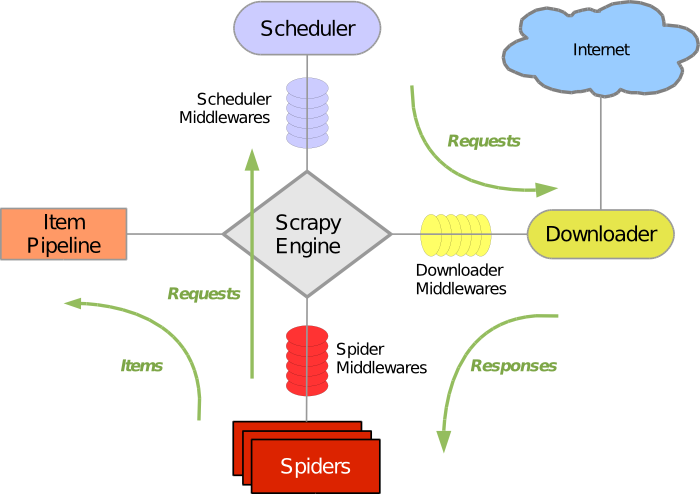
爬虫系统架构可以理解为一套爬虫数据采集的技术解决方案以及为实现该方案而设计的一套程序结构.

爬虫系统会涉及到很多的技术, 要把各个技术方案整合起来, 不同的技术之间要设计一个合理高效的逻辑关系, 使它们结合在一起, 成为一个有机的整体, 才能实现数据采集的目的.

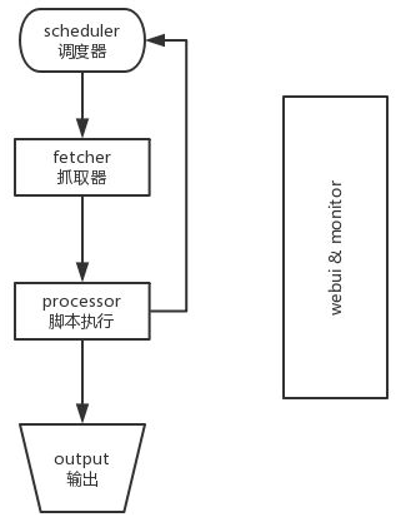
爬虫框架的应用场景：当需要同时抓取成百上千个网站时，并对每一个爬虫进行统一管理. 想要在后期很方便的进行维护和管理, 就要开发自己的系统架构.

常见的爬虫系统架构：

* scrapy就是一套爬虫的数据采集技术解决方案，其实现出来的架构一般是一套单机爬虫结构，属于轻量级爬虫系统架构



* pyspider也是一套爬虫的数据采集技术解决方案，其实现出来的架构可以是分布式的，同时内置了多种技术与功能，属于重量级爬虫系统架构



fetcher抓取器, 类似于scrapy中的Downloader下载器, 专门负责发出网络请求, 获取响应.

processor处理器, 专门处理响应, 从响应中提取信息. 类似于scrapy中的Spider.

output输出, processor从响应中获取数据之后, 输出到指定的位置. 类似于scrapy中的Item Pipeline. pyspider中已经集成了一些数据库连接的引擎, 只需要进行简单的配置就能把数据保存到数据库中.

pyspider还有一个webui & monitor组件, 即可视化web界面, 可以通过web界面控制爬虫的运行并监控爬虫的运行状态.

### 如何设计自己的爬虫系统架构

为什么要自己实现爬虫系统架构：

* 爬虫系统架构一般比较简单，开发效率快. 相比于大数据分析, 机器学习或人工智能的系统架构来说, 爬虫的系统架构比较简单.
* 针对性强，根据公司自身业务情况，要爬取的网站, 使用的爬取工具, 以及主要遇到的问题，定制化功能. scrapy和pyspider是通用性的爬虫架构.
* 扩展性高. 自己设计的架构, 开发的代码, 容易扩展. 而已有的架构较难进行扩展.
* 拥有自主的知识产权
* 学会scrapy、pyspider你可能只会做爬虫；但掌握如何自己搭建爬虫系统架构，你绝对不是只会爬虫

应注意的地方：

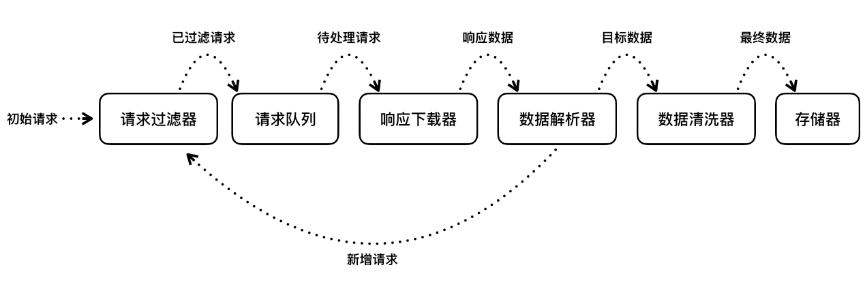
**业务角度**

* 爬虫请求的过滤和调度. 过滤: 请求过滤器, 调度: 请求队列. 下载器是并发的, 请求队列对应多个下载器, 所以要考虑请求的分发与调度的问题.
* 分布式支持. 要爬取成百上千个网站, 请求的数据量非常大, 要进行分布式的设计, 用多个节点来爬取数据.
* 断点续爬以及增量式抓取. 程序出错或硬件出错退出.

**代码角度**

* 代码复用性. 如请求队列中队列的代码也可能会在保存数据时使用, 对于这些使用相同技术的模块, 可以把功能提取出来, 封装成一个单独的模块, 达到代码复用的目的.
* 功能模块化、插件化，使得易扩展
* 程序健壮性、易维护性. 程序运行过程中可能会出现一些难以预料到的异常或错误, 都需要对异常进行捕获和处理, 这就是程序的健壮性.
* 架构简洁. 功能解耦.

**爬虫主要业务流程：**



解析器中只提取, 清洗器中只清洗.

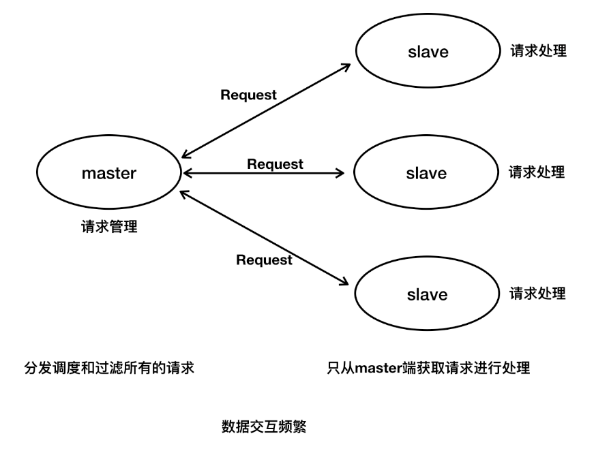
数据分析, 大数据分析, 机器学习, 人工智能一系列渐进式的系统架构设计.

要参考课程中的爬虫业务流程和系统架构, 结合自己对爬虫的理解, 以及所从事的行业的具体业务, 得出自己的爬虫业务流程和爬虫系统架构.

在爬取多个不同的网站时, 同一个代理在其有效时间内, 尽可能多的向多个网站发送请求, 而相同的网站发送的请求则相隔一定的时间. 如何在爬虫运行的过程中动态添加代理. 如何实现代理地址的分发. 一次性提取多个代理, 对代理在不同的爬虫节点中进行分配. 先获取每个网站总体上的数据量, 再根据网站的数据量, 动态的分配爬虫爬取的节点数. tumblr three. 以一个web界面显示出总爬取量和已爬取量. 如tumblr three一样, 多线程处理, 在爬取的过程中更新网站的总爬取量, 并根据总的爬取量来动态的分配爬取节点, 动态的生成爬取的容器. 动态的分配代理. 爬虫下载速度的检测. 不只是每秒钟爬取的网页的个数, 也要对爬虫节点的带宽进行检测, 一旦发现带宽完全占用, 就要启用新的物理机器并动态创建容器爬取节点.

数据保存到分布式的redis中, 达到一定的数据量时再写入到mysql数据库中. 在各个redis数据库中集中进行数据的清洗和处理.

#### 爬虫系统架构方案1



master和每个slave都相当于是一个服务器或主机, 一个master端连接多个slave端进行任务的分发和调度, 由slave端进行数据的请求和处理. 所以这个系统架构方案本身就是分布式的方案. 主-从结构.

master端实现请求管理的功能, 具体包括请求的过滤和分发. 而slave端则要实现具体请求的处理, 具体包括从master获取请求, 向网站发送请求, 获取响应, 对响应进行处理, 从中提取出数据和新的请求, 把新的请求交给master进行过滤入队, 把提取到的数据提交给对应的模块进行清洗和保存等工作.

slave端要能够访问到master端的请求队列和请求过滤器, 当slave端中提取到新的请求时, 再发送到master端进行过滤和入队列, 再由master端统一进行分配和调度. slave端的请求只能从master端来获取, 不能从自身或其它的slave端进行获取.

一个slave端获取到的新的请求交给master过滤入队之后, 很可能不会由原来的slave端进行处理, 而是交给分发给其它的slave端进行处理. 如果多个slave端的结构和代码都是相同的, 它们处理的请求也是相同的, 此时请求交给哪个slave端处理都是一样的, 上面的系统架构就能满足此种需求. 但如果不同的slave端的结构, 代码和功能都是不同的, 如一个slave端使用requests向发送pc端的请求获取响应, 一个slave端使用selenium发送pc端的请求获取经过浏览器渲染之后的响应, 一个slave端使用appium发送移动端的请求, 此时就不能使用上面的系统架构方案, 必须要设计新的架构方案. 此时就需要重新设计主从关系, 可以对slave端进行分类, 把相同结构, 功能和代码的slave放在一起, 使用相同master进行管理, 设计出多套master-slave的主从结构. 也可以设计新的数据结构, 在不同类型的slave端获取的请求中添加一个字段, 记录其产生的slave端的类型, slave端获取请求时, 只能获取并处理自己对应的类型的请求.

slave只从master端获取请求, 并且一个slave只对应一个master. 因为不同的master具有不同的过滤器和队列, 如果一个slave对应多个master, 就可能会造成请求的重复和数据的冗余.

多网站的情况下能否使用多个master?

爬虫系统架构方案1中可能会存在的问题.

slave产生的新的请求要交给master进行过滤, 如果是新的不重复的请求, 就把请求放入队列交给某个slave进行处理, 也就是说, 一个新的请求, 要进出master端两次, 在请求量特别大或者slave节点数量非常多的情况下, 要考虑master端的存储能力, 并发能力, 数据处理能力和网络能力等能否满足整个项目的需求.

在程序运行时, 先向master端中添加一个初始的请求, master对初始请求进行过滤和分发, slave端接收到初始请求进行处理.

master端使用单例模式来设计.

数据结构

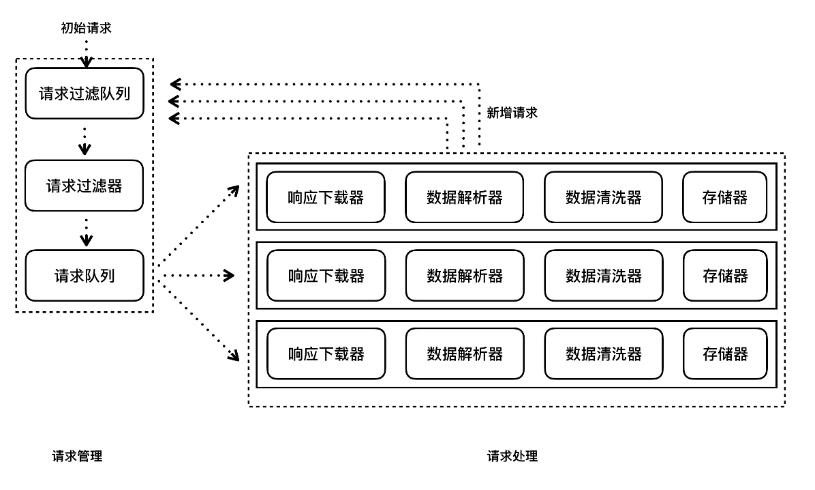
常用算法.

以上系统架构方案整体上结构清晰, 功能简洁易于实现. 在实现时只需要进行解耦, 把代码分为master和slave端来实现即可, master端只需要设计出过滤器和队列, slave端只需要实现请求处理的功能即可. 而在程序开始运行时, 只需要启动一个master端, 多个slave端. 无论是先启动master还是先启动slave都是可以的, 先启动master时, 由于没有slave端处理请求, 程序的多个请求会处于阻塞状态, 而先启动slave时, slave端会处于挂起状态, 等待从master端的队列中获取请求.

在运行的过程中, 如果其中一个slave端出错退出了, 并不会影响到其它slave端的运行, 出错slave端中丢失的请求会再次由master交给其它slave进行处理. 唯一需要考虑的是master端需在运行时的瓶颈压力.

##### 基础策略：

功能设计解耦合，易维护、易扩展



内部功能的具体实现原理和过程.

master端

多出来一个 "请求过滤队列", 因为master端请求过滤是同步的方式执行的, 所以会产生堵塞, 为了解决堵塞, 提高程序整体的运行效率, 需要设置一个 "请求过滤队列", 存放未过滤的请求.

slave端

每个slave端实现的功能是类似的, 这里以一个slave为例进行说明.

每个slave内部有多条线程异步的进行请求的处理, 所以从整体上来看, 一个slave中的多条线程是异步的. 但在一个slave的内部, 每一条线程中的各个功能则是以同步的方式执行的, 也就是说, 在一个线程的内部, 发送请求, 获取响应, 数据清洗, 数据存储这些功能是以同步的方式执行的.

如果在对请求的数据进行解析时, 提取到了新的请求, 就会把新的请求以异步的方式发送到master端进行请求的过滤的, 由于是异步的, 在有多个slave端, 每个slave端中又有多个请求处理线程时, 可能就会同时向master端发送大量的新增请求, 而请求过滤器是以单线程同步的方式进行请求过滤的, 必须要一个请求一个请求的进行过滤, 如果新增请求非常多, master端的请求过滤器就会出现长时间的堵塞, slave端也就不能向master端发送新增的请求, 这样就会造成程序整体上的阻塞. 为了解决这个问题, 可以在master中设置一个请求过滤队列, 把slave端发送过来的新增请求缓存起来. 请求过滤器会由master端主动的进行调用, 不断的从请求过滤队列中逐个取出来新增的请求进行过滤.

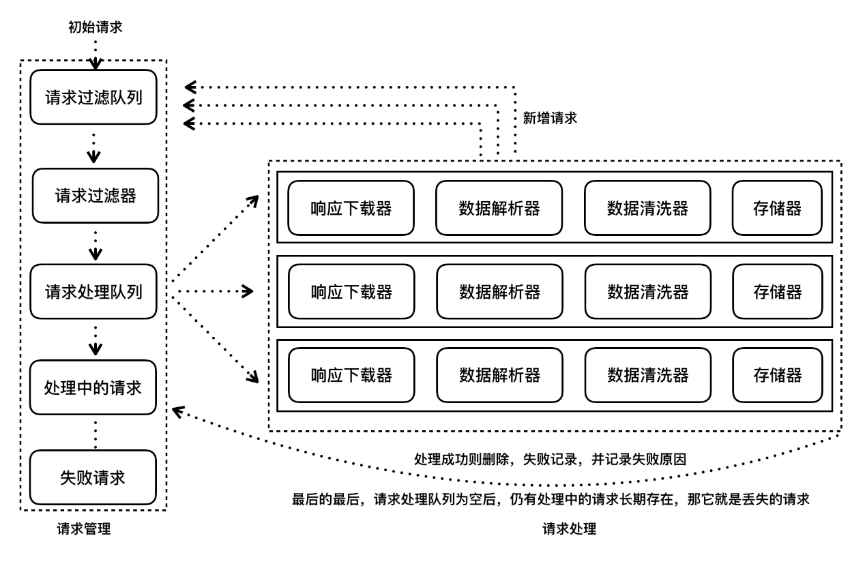
master中的 "请求队列" 以及 "请求过滤队列", 既然能够被多个slave端进行访问, 一定要是基于redis或消息队列实现的分布式的队列, 可以使用第三方的服务如kafka消息队列来实现. 所以master端需要手动实现的功能, 就只是对请求进行过滤的功能了. 从整体上看, master端就是一个大的无限循环, 不停的循环判断 "请求过滤队列" 中是否有新的请求, 并从 "请求过滤队列" 中取出来请求, 进行过滤, 然后把未处理过的新增请求发送到请求队列中.

slave端, 表面上是与master端进行交互的, 但实际上却是与redis或消息队列实现的 "请求队列" 和 "请求过滤队列" 进行直接通信的.

整个爬虫系统架构方案的功能是解耦的方式设计的, 易于维护和扩展. 在需要对某一个功能进行替换, 修改或升级时, 只要这个功能对外提供的接口是一致的, 其它功能都不会受到影响.

##### 升级策略1：

解决丢失请求、失败请求的捕获



由于网络中断, 网站反爬等原因导致slave中某个请求处理过程中未能按正常流程完成, 此请求就相当于丢失了, 此时需要找到这个请求处理失败的原因, 进行对应的处理, 不找到出现问题的原因, 可能多次的尝试也不能解决问题. 在这里使用的解决方法是先进行2或3次的尝试, 避免是由于网络等临时的原因导致的请求处理失败. 如果2, 3次尝试后请求依然失败, 就可能是由于系统原因造成的, 需要把这个请求记录下来, 连同失败的原因一起保存到master中的失败请求队列中. 到最后就可以集中对失败的请求进行处理, 进一步完善代码和逻辑, 然后对失败的请求再次发起请求进行处理.

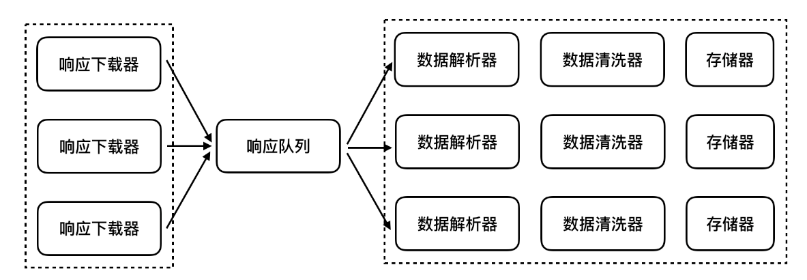
在这个过程中, 把失败请求保存到失败请求队列中的工作是由slave端来完成的, 如果由于slave端系统崩溃, 或者断电导致的请求处理的失败, slave端就无法捕获到这个失败的请求, 更无法把失败的请求保存到失败请求队列中, 这个请求就会完全丢失.

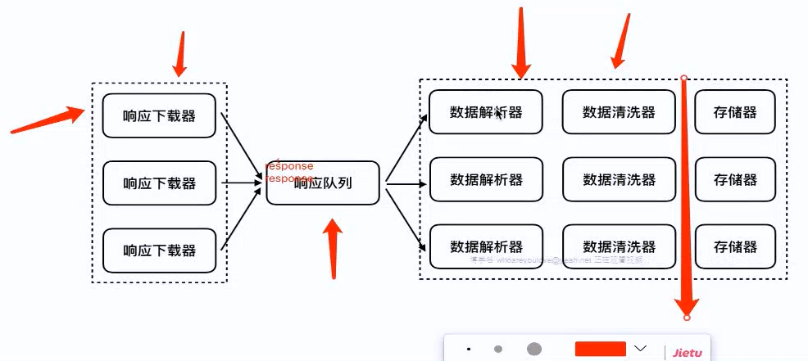
为了解决这个问题, 可以在master端中再添加一个 "处理中的请求" 队列, 记录正在处理中的请求. 当slave端从请求处理队列中获取一个请求, 请求一出队列, 就把它也存到 "处理中的请求" 队列中, 当slave端对请求的一系列处理正常完成并把数据保存到存储器中之后, 整个过程中没有出现异常和错误, 此时再由slave端向master端发送消息, 删除 "处理中的请求" 这个已经正常处理完成的请求. 如果请求处理失败而slave端系统的运行正常, slave端就会主动把这个失败的请求连同失败的原因一起保存到 "失败请求" 队列中, 同时删除掉 "处理中的请求" 队列中的同一个请求. 当slave端是由于系统问题而导致的请求处理的失败时, 就无法进行错误的捕获和提交, 此时既不会向失败的请求队列中添加此请求, 又不会删除掉 "处理中的请求" 中的此请求, 结果就是此请求会在 "处理中的请求" 队列中长期存在. 可以设定一定的时间, 如10\*60s, 来确定此请求是长期存在的请求. 对于在 "处理中的请求" 队列中长期存在的请求, 就可以认为是因为slave端系统错误产生的错误请求, 此时可以用发邮件或微信的方法通知用户立即解决这个问题.

负载均衡的方法来自动启动slave客户端.

##### 升级策略2：

分离响应异步下载与响应异步处理，避免一方阻塞耗时影响另一方





python中的多线程, 实际上是同一个进程中的多线程, 并且每次只有一个线程在运行.

在一个slave中, 会使用多线程来实现对请求的异步并发处理, 响应下载器, 数据解析器, 数据清洗器, 存储器之间是同步的关系, 如果某一个步骤的处理耗时较长 如数据解析或清洗时有一些耗时的计算操作, 或存储时有耗时的IO操作, 整个线程就会阻塞到这个步骤中, 直到它的执行完成, 才会进行下一步的操作. 这时, 就降低了slave端整体上的并发量. 为了解决这个问题, 可以在这个步骤之前添加一个队列, 上一个步骤处理的结果添加到队列中, 此时上一个步骤就能够接着去处理上上一步的结果, 就不会整体上都阻塞在一个步骤的操作处, 而下一步耗时的操作就从队列中取出来数据. 这样就能提高整体上的并发量. 假设数据解析器中有一个耗时较长的操作, 可以在响应下载器和数据解析器之间添加一个响应队列, 响应下载器可以使用线程池/协程池来实现异步并发, 数据解析器-数据清洗器-存储器这部分操作可以使用线程池/线程池的方式实现异步并发, 响应下载器把获取的响应保存到响应队列中之后, 就可以接着去发起下一个请求获取响应了, 此时响应下载器就不会受到后面耗时操作的阻塞的影响了.

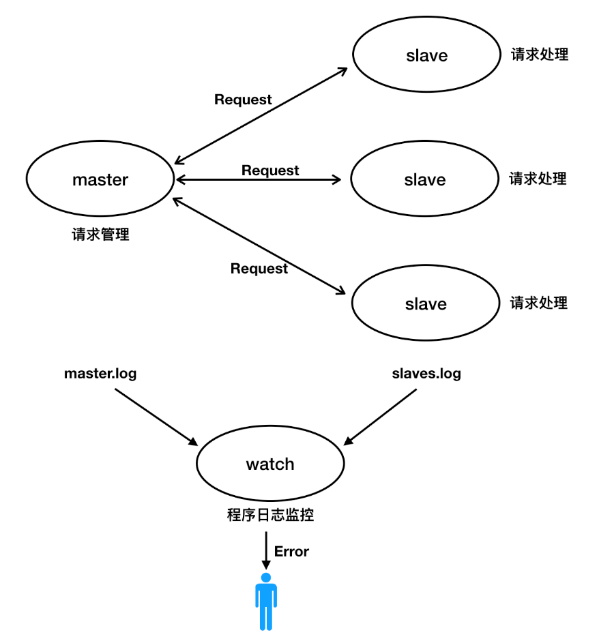
如果slave中每一个步骤的处理都没有太长的耗时操作, 就可以使用原来的同步的方式来处理, 也不会对整体性能有太多的影响.

如果其它步骤中, 如存储器中有一些耗时较长的IO操作, 也可以在数据清洗器和存储器之间添加一个队列.

响应队列的实现, 可以使用基于mysql和redis的数据库队列, 由于slave中所有的异步任务都处于同一个进程中, 也可以使用内存队列来实现, 如使用python中的queue模块来实现.

##### 升级策略3：

日志监控捕获错误，并实时通报

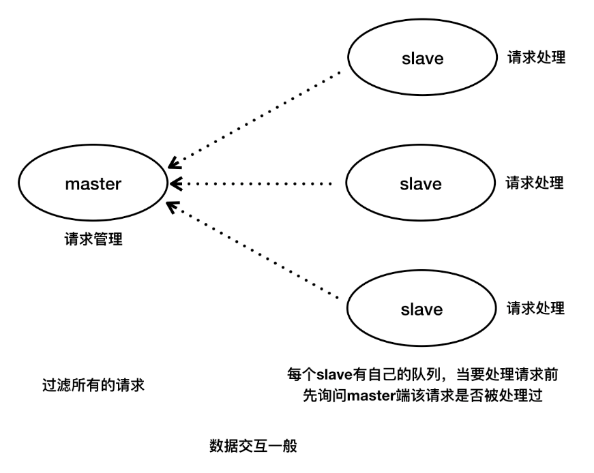


爬虫的运行时间非常长, 甚至是长年累月的运行. 包括爬虫在内的所有的程序, 在运行过程中一旦出现了错误, 就要及时的处理, 否则程序就会崩溃, 或者错误一步步传递下去可能会给之后的步骤的运行带来更多的错误, 或者导致它本身和其它结果的大面积错误. 所以要对程序的运行状态进行监控. 最常见的监控是日志监控. 可以在程序中预见的可能出现异常或错误的地方进行 "埋点", 使用try-except来对异常或错误进行捕获, 同时在except中通过日志把错误的信息记录下来, 然后对日志信息进行实时的监控, 如果发现某一个级别如error级别的错误, 就立即通知用户对错误进行处理.

埋点 > 监控 > 通知

可以使用ELK日志处理系统来实现日志的监控和通知. ELK可以实现亿级的日志信息的处理.

#### 爬虫系统架构方案2

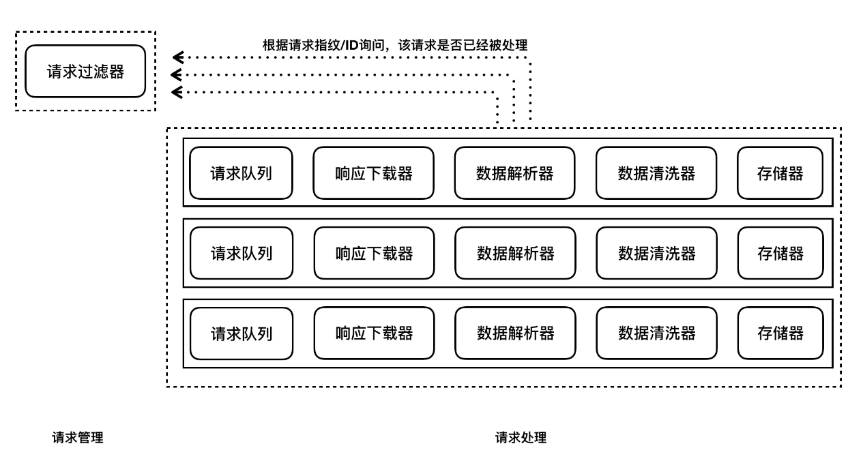


系统架构方案1中, 请求对象会在master端和slave端之间频繁的传递, 当数据量大或slave端的数量比较多时, 总体性能就有可能会降低.

爬虫架构方案2中, master端只负责过滤所有的请求, 并不保存请求队列, slave端除了要处理请求之外, 还有自己的队列, 每个slave获取到新的请求时, 都生成请求的指纹, 把指纹发送给master端进行查询, 看这个请求是否处理过, 如果没有处理过, 就把这个请求保存到自己的队列中, 然后从自己的队列中取出请求进行处理, 在这个请求处理成功之后, 再向master中发送此请求的指纹, master端是把这个请求的指纹保存到指纹队列中, 表示这个请求已经进行了处理.

每个slave产生的请求只能被它本身处理, 这时就能实现一个master对应多个不同功能, 不同结构, 不同实现的slave的架构. 但此时请求丢失和失败请求的处理逻辑就与方案1中的方法不同了, 方案1中是在master中实现的, 方案2中就要在slave端进行实现了.

##### 基础策略：



## 2. 爬虫系统架构组件

### 爬虫系统架构主要组件及其相关技术

1. **队列组件**

**队列功能: 缓存及排序**

* + **队列类型：**
    - FIFO 广度优先
    - LIFO 深度优先
    - 优先级队列. 最高级列表页的优先级最高, 然后是第二级的列表页, 详情页的优先级最低.
  + **内存队列：单机模式**
    - Python内置队列模块
    - Asyncio中的队列模块
    - Gevent中的队列模块
    - Tornado中的队列模块
  + **持久化队列：分布式、断点续爬**
    - Redis队列 百万至千万级数据量
    - 消息队列(Kafka、RabbitMQ) 千万级及更多

1. **过滤器组件**
   * 指纹过滤器（redis等）：千万级数据去重（无误判） 请求+数据去重
   * simhash过滤器：只能用于文本去重, 相似文本去重 数据去重
   * 布隆过滤器（redis）：亿级数据去重（存在极小概率的误判） 请求+数据去重
2. **下载器组件**

发起请求, 获取响应, 一般与异步组件配合起来使用

* + urllib/requests
  + aiohttp
  + tornado.httpclient
  + twisted.web.client
  + websocket-client
  + selenium+Chrome-Headless
  + appium+android-app

1. **异步组件**

不同的模块代码结构有很大不同, 很难使用相同的接口

* + asyncio 推荐使用
  + celery+eventlet/gevent
  + selenium+chrome-headless Pool (多个浏览器实例)
  + appium+android-app Pool (多台设备)

1. **数据解析提取组件**
   * **语法规则：**
     + 正则表达式
     + Xpath
     + Css选择器
     + Jsonpath
     + html/css/JS 前端中js代码经常会进行混淆, 使用a, b, c这样的无意义的名字来命名
   * **解析提取工具：**
     + re
     + lxml
     + lxml+bs4
     + lxml+pyquery
     + jsonpath
     + js2py/pyexecjs+node
2. **数据清洗组件**
   * 自定义清洗规则
3. **数据存储组件**
   * **存储介质：**
     + file: csv/json/...
     + DB: mysql/postgresql/mongodb 当数据量达到上亿级时要考虑使用大数据的存储方案. 如hdfs
   * **存储工具：**
     + csv、json
     + sqlalchemy/django-orm/mongoengine
4. **程序监控组件**

使用dockerfile运行即搭建好了ELK系统, 但更重要的是在程序中进行埋点, 在程序中记录日志, ELK系统才能捕获日志并在后期进行处理.

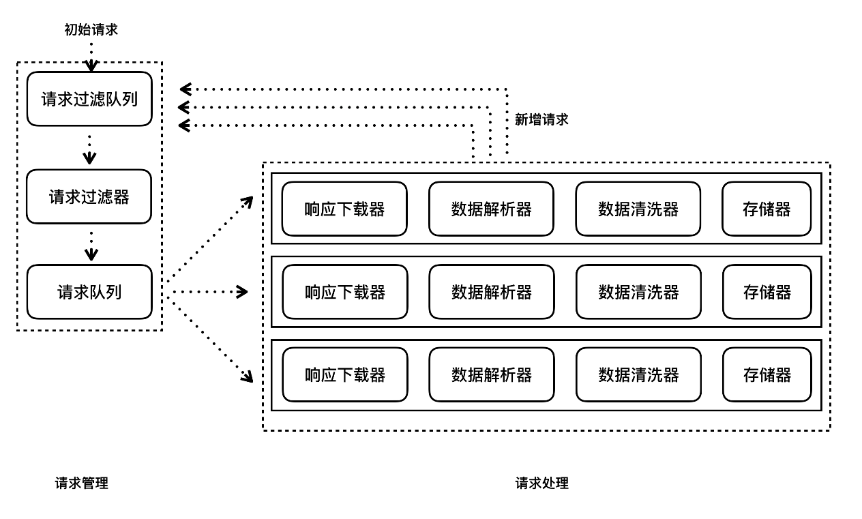
* + ELK 运维
    - elasticsearch: 日志数据存储
    - logstash: 日志收集工具
    - kibana: 日志可视化

1. **可视化控制组件 \*\*\*\***
   * web界面(django\flask)
   * GUI界面(pyqt)

这些组件里涉及的技术并不是在同一个爬虫系统架构中都要用到。因为它们就相当于是我们的武器弹药库，针对不同的敌人选择使用不同的武器和弹药来提高杀伤力。且也绝不仅仅只有上面列出来的这些，只是相对来说，它们更常用一些。

## 3. 爬虫系统架构实现

### 3.1. 基础策略实现



#### 同步模式实现

**同步模式代码结构：以同步模式实现主要核心逻辑**



所有的代码都放在内层的一个spidersystem目录中, 形成一个独立的模块. 在使用时就可以直接调用这个模块中的功能.

##### request\_manager 请求管理器

注意: request\_manager是**爬虫请求管理阶段**所实现的代码，其中实现了请求过滤器和请求队列的功能, 这里直接引用过来使用.

未加锁版request\_manager和加锁版request\_manager, 这里使用的是未加锁版request\_manager, 加锁版使用的场景是多个slave端同时向一个master端的请求过滤器中进行请求是否重复的判断, 由于是异步进行判断的, 多个slave端就有可能同时判断一个相同的请求不重复的结果. 为了解决这个问题, 加一个redis锁把去重判断转换为同步的, 实现请求的不重复判断. 而在这里, 由于请求是添加到 "请求过滤队列" 中, 然后在 "请求过滤器" 中使用单线程或同步的方式对 "请求过滤队列" 中的请求一一进行判断, 此时就不会出现冲突和误判的情况, 也就不需要使用加锁版的request\_manager了.

把未加锁版request\_manager代码复制到SpiderSystem中, 修改request\_manager/\_\_init\_\_.py, 把class RequestScheduler(object) 修改为 class RequestManager, 即修改为请求管理器.

使用时, 先导入, 再传入一些配置信息. 确定使用的队列类型和过滤器类型. 队列类型有 fifo, lifo, priority, 而过滤器类型有 redis, memory, mysql, bloom

usage\_demo.py

**from** **six.moves** **import** queue

**from** **reuquest\_manager.request** **import** Request

**from** **reuquest\_manager** **import** RequestManager

*# request\_manager.\_\_init\_\_.RequestManager.\_\_init\_\_中有4个参数. queue\_type, filter\_type, queue\_kwargs, filter\_kwargs, 所以单独提供这4种参数.*

REQUEST\_SCHEDULER\_CONFIG = {

"queue\_type": "fifo",

*# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中. redis\_tools.base.BaseRedisQueue.\_\_init\_\_中*

*# 注意：redis的key会被queue\_name重写，因此此处不用写*

"queue\_kwargs": {"host":"172.17.0.2", "port":6379},

"filter\_type": "redis",

*# 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中*

*# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写*

"filter\_kwargs": {"redis\_key":"redis\_filter", "redis\_host":"172.17.0.2"},

}

request\_scheduler = RequestManager(\*\*REQUEST\_SCHEDULER\_CONFIG)

##### request.py: 封装请求对象

把请求地址, 请求方式, 请求参数, 请求头, 请求体等与请求相关的内容封装成一个请求对象.

# 构建一个请求对象

import urllib.parse

class Request(object):

def \_\_init\_\_(self, url, method="GET", query={}, body={}, name="test", headers=None):

self.url = url

self.method = method

if not isinstance(query, dict):

raise Exception("query myst be a dict")

self.query = query

if not isinstance(body, dict):

raise Exception("body myst be a dict")

self.body = body

self.name = name # 表示该请求属于哪个爬虫

self.headers = headers

@property

def url\_with\_query(self):

'''把字典的请求参数与url地址里的请求参数进行整合'''

url = self.url

\_ = urllib.parse.urlparse(url)

url\_without\_query = \_.scheme + "://" + \_.hostname + \_.path

url\_query = urllib.parse.parse\_qsl(\_.query)

query = self.query.items()

all\_query = sorted(set(list(query) + url\_query))

return url\_without\_query + "?" + urllib.parse.urlencode(all\_query)

*# 构建一个请求对象*

**import** **urllib.parse**

**class** **Request**(object):

**def** \_\_init\_\_(self, url, method="GET", query={}, body={}, name="test", headers=None):

self.url = url

self.method = method

**if** **not** isinstance(query, dict):

**raise** **Exception**("query myst be a dict")

self.query = query

**if** **not** isinstance(body, dict):

**raise** **Exception**("body myst be a dict")

self.body = body

*# 表示当前请求属于哪个爬虫, 如淘宝爬虫, 京东爬虫. 每个爬虫的解析, 去重和存储等规则都是不同的*

self.name = name

self.headers = headers

@property

**def** url\_with\_query(self):

*'''*

*功能: 把字典的请求参数与url地址里的请求参数进行整合.*

*请求中包括请求行, 请求行中包括请求地址, 请求参数, 请求协议. 请求参数本身属于请求地址的一部分, 而在代码中, 为了修改的方法, 一般把查询参数写为字典形式的query={}, 而在发送请求的时候, 程序会自动把字典形式的查询参数转换为url地址中的查询参数.*

*requests, urllib, 把字典格式的查询参数转换为url地址中的查询参数. 而tornado中则不支持字典格式的查询参数, 所以这里定义一个单独的方法, 把字典格式的查询参数和url地址中的查询参数进行合并.*

*这个方法相当于request\_manager.request\_filter.\_\_init\_\_中的\_get\_request\_filter\_data方法中把3. query和url地址合并的功能单独提取了出来.*

*在使用tornado时会用到这个属性, 在requests, urllib中则不会用到这个属性*

*'''*

url = self.url

\_parsed\_url = urllib.parse.urlparse(url)

url\_without\_query = \_parsed\_url.scheme + "://" + \_parsed\_url.hostname + \_parsed\_url.path

url\_query = urllib.parse.parse\_qsl(\_parsed\_url.query)

query = self.query.items()

all\_query = sorted(set(list(query) + url\_query))

**return** url\_without\_query + "?" + urllib.parse.urlencode(all\_query)

##### response.py：封装响应对象

封装响应url, 响应体, 响应头, 响应对应的请求对象

import re

import lxml.etree

from bs4 import BeautifulSoup

from pyquery import PyQuery

class Response(object):

'''响应对象'''

def \_\_init\_\_(self, request, status\_code, url, headers, body):

self.request = request

self.status\_code = status\_code

self.url = url

self.headers = headers

self.body = body

def xpath(self, rule):

html = lxml.etree.HTML(self.body)

return html.xpath(rule)

def select(self, rule):

soup = BeautifulSoup(self.body, "lxml")

return soup.select(rule)

def d(self, rule):

d = PyQuery(lxml.etree.HTML(self.body))

return d(rule)

def re\_match(self, rule):

return re.match(rule, self.body)

def re\_search(self, rule):

return re.search(rule, self.body)

def re\_findall(self, rule):

return re.findall(rule, self.body)

**import** **re**

**import** **lxml.etree**

**from** **bs4** **import** BeautifulSoup

**from** **pyquery** **import** PyQuery

**class** **Response**(object):

*'''响应对象'''*

**def** \_\_init\_\_(self, request, status\_code, url, headers, body):

*# 响应对应的请求*

self.request = request

self.status\_code = status\_code

*# 这里的url为响应地址, 如果有重定向, 这里的地址就是重定向后的地址*

self.url = url

self.headers = headers

self.body = body

*# 定义多种解析方法, 把解析方法定义在响应对象中, 这样在用户使用时就不用再从外部导包, 不用从0开始写代码了*

*# xpath方法, 传入一个解析规则 rule, 从响应的body中提取出规则对应的内容*

**def** xpath(self, rule):

html = lxml.etree.HTML(self.body)

**return** html.xpath(rule)

**def** select(self, rule):

soup = BeautifulSoup(self.body, "lxml")

**return** soup.select(rule)

**def** d(self, rule):

d = PyQuery(lxml.etree.HTML(self.body))

**return** d(rule)

**def** re\_match(self, rule):

**return** re.match(rule, self.body)

**def** re\_search(self, rule):

**return** re.search(rule, self.body)

**def** re\_findall(self, rule):

**return** re.findall(rule, self.body)

##### downloader.py：封装响应下载器对象

下载器对象实现的功能: 给下载器中的方法提供一个request.py中定义的请求对象, 下载器对象根据提供的请求对象发起请求, 获取响应, 获取响应之后要构建并返回一定格式的response响应对象.

可以使用不同的模块来实现下载器, 每个模块都有自己独特的功能逻辑和代码结构, 把这些模块封装到下载器对象中.

import requests

from spidersystem.response import Response

class RequestsDownloader(object):

def fetch(self, request):

if request.method.upper() == "GET":

response = requests.get(request.url\_with\_query, headers=request.headers)

elif request.method.upper() == "POST":

response = requests.get(request.url\_with\_query, data=request.body, headers=request.headers)

else:

raise Exception("Only support GET or POST method ")

return Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

**import** **requests**

*# 由于要把所有的代码都放在spidersystem这个独立的模块中, 所以导包的时候要以spidersystem为根目录来导包.*

*# pycharm可能会识别为错误. 因为pycharm中导包时先从当前目录中查找, 再从系统环境中查找此模块. 在运行代码时, 都是在spidersystem这个目录的角度来运行的, 所以spidersystem中导入的所有包都要从spidersystem这个路径导入.*

**from** **spidersystem.response** **import** Response

**class** **RequestsDownloader**(object):

*# 根据不同的模块来实现下载器的功能. 这里是基于requests模块来实现的下载器, 所以定义为RequestDownloader, 还可以定义urllib downloader, tornado downloader等*

**def** fetch(self, request):

*# 定义一个单独的方法, 提供一个请求对象, 返回一个响应对象. 注意这里的请求对象必须是request.py中的类实例化出来的请求对象, 而返回的response对象也必须是response.py中对应的类实例化出来的响应对象, 这样才能根据response.py中定义的方法从响应中提取出网页中的内容.*

**if** request.method.upper() == "GET":

*# requests.get支持字典格式的params参数. 但由于在request.py请求对象中已经定义了一个url\_with\_query方法, 就可以直接调用这个方法, 把所有的请求参数都整合到url地址中.*

*# response = requests.get(request.url, headers=request.headers, params=request.params)*

response = requests.get(request.url\_with\_query, headers=request.headers)

**elif** request.method.upper() == "POST":

response = requests.post(request.url\_with\_query, data=request.body, headers=request.headers)

**else**:

**raise** **Exception**("Only support GET or POST method ")

*# 返回response.py请求对象中定义的特定格式的response对象.*

**return** Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

##### spider.py：定义爬虫基类

整个爬虫系统架构是帮助用户实现请求的处理, 请求的异步发送, 响应的获取等功能, 但在解析器中如何从响应中提取数据, 同样, 数据清洗和存储的逻辑都是由用户自己实现的. spider.py中定义了爬虫的基类, 实现数据解析, 清洗和存储的基本框架, 留出对应的api接口, 在具体的爬虫代码中再实现具体的功能.

from spidersystem.request import Request

class BaseSpider(object):

'''爬虫基类'''

def start\_requests(self):

yield Request("http://www.example.com")

def parse(self, response):

'''生成器'''

yield

def data\_clean(self, data):

return data

def data\_save(self, data):

pass

**from** **spidersystem.request** **import** Request

**class** **BaseSpider**(object):

*'''爬虫基类'''*

name="demo"

*# 发起初始的请求. 返回一个请求对象. 可能会返回一个, 也可能会返回多个请求对象, 为了统一, 这里使用生成器返回, 就避免了一个请求对象和多个请求对象时的判断.*

**def** start\_requests(self, start\_url):

*# yield Request("http://www.example.com", name=self.name)*

**yield** Request(start\_url)

*# 数据解析. 从响应中解析数据. 可能返回新的请求, 也可能返回提取到的数据. 为了方便外部的调用, 定义为一个生成器.*

**def** parse(self, response):

*'''生成器'''*

**yield** Request

**yield** Data

*# 数据清洗*

**def** data\_clean(self, data):

**return** data

*# 数据保存*

**def** data\_save(self, data):

**pass**

##### main.py：实现程序主逻辑（主从端）

程序的入口, 程序的主逻辑.

master端要实现的内容, 从请求过滤队列中取出一个请求, 使用请求过滤器进行判断, 如果已经存在, 就把它忽略或保存到日志中, 如果不存在, 就把它加入到请求队列中, 而使用过滤器进行过滤和处理在request\_manager中已经存在了, 这里只需要实现从请求过滤队列中取出来请求就可以了.

import threading

from spidersystem.downloader import RequestsDownloader

from spidersystem.request\_manager import RequestManager

from spidersystem.request import Request

from spidersystem.response import Response

from spidersystem.request\_manager.utils.redis\_tools import get\_redis\_queue\_cls

Queue = get\_redis\_queue\_cls("fifo")

class Master(object):

def \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

self.spiders = spiders

self.request\_queue = request\_queue

def run\_start\_requests(self):

for spider in self.spiders.values():

for r in spider().start\_requests():

self.request\_manager.add\_request(r, self.request\_queue)

def run\_filter\_request(self):

while True:

for request in self.filter\_queue.get():

self.request\_manager.add\_request(request, self.request\_queue)

def run(self):

# self.run\_start\_requests()

# self.run\_filter\_request()

threading.Thread(target=self.run\_start\_requests).start()

threading.Thread(target=self.run\_filter\_request).start()

class Slave(object):

def \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

self.downloader = RequestsDownloader()

self.spiders = spiders

self.request\_queue = request\_queue

def run(self):

while True:

# 1. 获取请求对象

request = self.request\_manager.get\_request(self.request\_queue)

# 2. 发起请求

response = self.downloader.fetch(request)

# 3. 处理响应

spider = self.spiders[request.name]()

for result in spider.parse(response):

if result is None:

raise Exception("parse方法未返回任何结果")

elif isinstance(result, Request):

self.filter\_queue.put(result)

else:

clean\_result = spider.data\_clean(result)

spider.data\_save(clean\_result)

**from** **spidersystem.downloader** **import** RequestsDownloader, TornadoDownloader

**from** **spidersystem.request\_manager** **import** RequestManager

**from** **spidersystem.request** **import** Request

**from** **spidersystem.response** **import** Response

**from** **spidersystem.request\_manager.utils.redis\_tools** **import** get\_redis\_queue\_cls

*# 由于master端的请求过滤队列需要提供给多个slave端使用, 所以它要支持分布式的功能, 这里直接使用redis队列. 可以使用fifo, lifo, priority 三种redis队列. 查看 SpiderSystem\spidersystem\request\_manager\utils\redis\_tools\\_\_init\_\_.py*

Queue = get\_redis\_queue\_cls("fifo")

**class** **Master**(object):

*"""实现请求管理, 包含请求过滤队列, 过滤器, 请求队列, 由于已经通过RquestManager实现了过滤器和请求队列, 所以这里可以直接导入使用. 还需要定义请求过滤队列, 可以在\_\_init\_\_方法中定义"""*

**def** \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

*# 请求过滤队列*

*# 实例化请求过滤队列时, 需要传递参数, 查看 SpiderSystem\spidersystem\request\_manager\utils\redis\_tools\base.py 中的 BaseRedisQueue 中的 def \_\_init\_\_() 确定要传递的参数*

*# def \_\_init\_\_(self, name, host='localhost', port=6379, db=0, maxsize=0, lazy\_limit=True, password=None, cluster\_nodes=None, redis\_lock\_config={}, use\_lock=False)*

*# 其中的name参数是必须要传递的, 指定 请求过滤队列 在reids中的key的名称*

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

*# 请求管理对象: 包含请求过滤器和请求队列, 使用方法参考 request\_manager.usage\_demo.py*

self.request\_manager = RequestManager(\*\*request\_manager\_config)

*# 爬虫对象*

self.spiders = spiders

*# 请求队列*

self.request\_queue = request\_queue

*# 可以把start\_requests方法单独提取出来*

**def** run\_start\_requests(self):

*# 执行所有爬虫中的初始的start\_requests方法, 获取初始请求的响应*

*# spiders为字典格式, 想要获取spider对象, 就要对spiders的values进行遍历*

**for** spider **in** self.spiders.values():

**for** r **in** spider().start\_requests():

self.request\_manager.add\_request(r, self.request\_queue)

*# 也可以把过滤请求的方法也单独提取出来*

**def** run\_filter\_queue(self):

*# 死循环, 让master端始终从请求过滤队列中取出来请求进行处理, 如果filter\_queue为空, 则会处于堵塞等待状态.*

**while** True:

*# master端要实现的内容, 从请求过滤队列中取出一个请求, 使用请求过滤器进行判断, 如果已经存在, 就把它忽略或保存到日志中, 如果不存在, 就把它加入到请求队列中, 而使用过滤器进行过滤和处理在request\_manager中已经存在了, 这里只需要实现从请求过滤队列中取出来请求就可以了.*

*# 使用for ... in ... 来获取filter\_queue中的内容时, 如果内容为空, 就会报错, 所以这里使用 while True死循环, 如果filter\_queue为空, 就会处于堵塞状态*

*# for request in self.filter\_queue.get():*

request = self.filter\_queue.get()

*# 调用请求管理器, 传入请求对象和队列名称, 对请求对象进行处理. 队列名称, 对于一个网站的爬虫, 可以使用一个队列名称, 如淘宝爬虫, 队列名称可以设置为 "taobao"*

self.request\_manager.add\_request(request, self.request\_queue)

*# 启动master端的方法*

**def** run(self):

*# 执行所有爬虫中的初始的start\_requests方法, 获取初始请求的响应*

*# spiders为字典格式, 想要获取spider对象, 就要对spiders的values进行遍历*

**for** spider **in** self.spiders.values():

**for** r **in** spider().start\_requests():

self.request\_manager.add\_request(r, self.request\_queue)

*# 死循环, 让master端始终从请求过滤队列中取出来请求进行处理, 如果filter\_queue为空, 则会处于堵塞等待状态.*

**while** True:

*# master端要实现的内容, 从请求过滤队列中取出一个请求, 使用请求过滤器进行判断, 如果已经存在, 就把它忽略或保存到日志中, 如果不存在, 就把它加入到请求队列中, 而使用过滤器进行过滤和处理在request\_manager中已经存在了, 这里只需要实现从请求过滤队列中取出来请求就可以了.*

*# 使用for ... in ... 来获取filter\_queue中的内容时, 如果内容为空, 就会报错, 所以这里使用 while True死循环, 如果filter\_queue为空, 就会处于堵塞状态*

*# for request in self.filter\_queue.get():*

request = self.filter\_queue.get()

*# 调用请求管理器, 传入请求对象和队列名称, 对请求对象进行处理. 队列名称, 对于一个网站的爬虫, 可以使用一个队列名称, 如淘宝爬虫, 队列名称可以设置为 "taobao"*

self.request\_manager.add\_request(request, self.request\_queue)

*# 如果把两个方法都单独提取出来了, 就可以在这里进行调用*

*# self.run\_start\_requests()*

*# self.run\_filter\_queue()*

**class** **Slave**(object):

*# slave 端要实现的功能为, 从master端的请求队列中取出来一个请求, 交给响应下载器来获取响应*

**def** \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

*# 实例化filter\_queue, 即请求过滤队列, 以把响应中提取到的新的请求保存到请求过滤队列中, 这里传递的参数与对应的master中的参数相同, 才能把请求保存到master的请求过滤队列中*

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

*# 实例化下载器对象*

self.downloader = RequestsDownloader()

self.downloader = TornadoDownloader()

*# 爬虫一般是在外部编写的, 编写好之后传递过来使用, 并且一般是多个爬虫. 新建 SpiderSystem/usage.py 爬虫文件, 在usage.py爬虫文件中实例化master和slave, 在实例化slave时把爬虫spiders传递进来. 因为运行时是从 usage.py 来运行slave端和master端的, 而usage.py与spidersystem包是在同一目录下的, 所以spidersystem中导包的时候都要从spidersysytem根目录下来导入*

self.spiders = spiders

self.request\_queue = request\_queue

*# 启动slave端的方法*

**def** run(self):

*# slave端也要进行死循环, 当队列中有新的请求时, 立即逐个获取并进行处理. 此时slave端是同步模式执行的.*

**while** True:

*# 1. 获取请求对象*

*# 根据队列名称 self.request\_queue, 获取对应的请求对象, 默认情况下是堵塞的, 这里的队列名称要与master端中的队列名称相同, 才能取出来数据*

request = self.request\_manager.get\_request(self.request\_queue)

*# 2. 发起请求*

*# 把请求交给下载器进行下载, 返回一个响应对象*

response = self.downloader.fetch(request)

*# 3. 获取爬虫对象*

*# 在爬虫spider中对响应进行处理, 在所编写的爬虫文件中, spiders = {BaiduSpider.name: BaiduSpider}, 所以self.spiders[request.name]获取的是类对象, 所以要进行实例化.*

*# 在爬虫文件中, 实例化Request对象时, 把name传递进去, yield Request("http://www.baidu.com", name=self.name), 所以这里的request.name和爬虫文件usage.py中的BaiduSpider.name是相同的, 这样才能从中取出来对应的爬虫类*

spider = self.spiders[request.name]()

*# 4. 处理响应对象*

*# 如果用户编写的爬虫中没有使用yield返回任何内容, 这里就为None*

**if** spider.parse(response) **is** None:

**raise** **Exception**("parse方法未返回任何结果")

*# 因为parse方法返回的是一个生成器, 所以要进行遍历*

**for** result **in** spider.parse(response):

*# 响应中可能会提取到数据, 也可能会提取到新的请求, 所以要进行判断.*

**if** result **is** None:

**raise** **Exception**("parse方法未提取到任何内容")

*# 如果是新的请求, 就把请求保存到master的请求过滤队列中*

**elif** isinstance(result, Request):

self.filter\_queue.put(result)

*# 如果提取到的是数据*

**else**:

clean\_result = spider.data\_clean(result)

spider.data\_save(clean\_result)

##### usage.py: 使用，该模块与spidersystem处于同级目录

爬虫文件, 在其中实例化master和slave端, 运行usage.py来分别启动slave端和master端. 可以同时运行多个slave端, 此时就相当于是分布式的了.

from spidersystem.spider import BaseSpider

from spidersystem.request import Request

from spidersystem.main import Master, Slave

REQUEST\_MANAGER\_CONFIG = {

"queue\_type": "fifo",

# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中

# 注意：redis的key会被queue\_name重写，因此此处不用写

"queue\_kwargs": {"host":"10.211.55.3", "port":6379},

"filter\_type": "redis",

# 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中

# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写

"filter\_kwargs": {"redis\_key":"redis\_filter", "redis\_host":"10.211.55.3"},

}

REQUEST\_QUEUE = "request\_queue"

class BaiduSpider(BaseSpider):

name = "baidu"

def start\_requests(self):

yield Request("http://www.baidu.com", name=self.name)

yield Request("http://www.baidu.com/s?wd=python", name=self.name)

yield Request("http://www.baidu.com/s?wd=python2", name=self.name)

yield Request("http://www.baidu.com/s?wd=python3", name=self.name)

yield Request("http://www.baidu.com/s?wd=python4", name=self.name)

def parse(self, response):

print(response)

print(response.url)

yield response.url

if \_\_name\_\_ == '\_\_main\_\_':

spiders = {BaiduSpider.name: BaiduSpider}

# Master(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG).run()

Slave(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG).run()

**from** **spidersystem.spider** **import** BaseSpider

**from** **spidersystem.request** **import** Request

**from** **spidersystem.main** **import** Master, Slave

REQUEST\_MANAGER\_CONFIG = {

"queue\_type": "fifo",

*# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中. redis\_tools.base.BaseRedisQueue.\_\_init\_\_中*

*# 注意：redis的key会被queue\_name重写，因此此处不用写*

"queue\_kwargs": {"host":"172.17.0.2", "port":6379},

*# 基于redis的指纹过滤器, 如果想要使用其它的过滤器, 查看 request\_manager\reuquest\_manager\utils\filter\_class\\_\_init\_\_.py, 可以使用 bloom, memory, mysql, redis*

"filter\_type": "redis",

*# 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中*

*# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写*

"filter\_kwargs": {"redis\_key":"redis\_filter", "redis\_host":"172.17.0.2"},

}

*# 定义请求队列的名称, 此名称会作为redis中的key*

REQUEST\_QUEUE = "request\_queue"

**class** **BaiduSpider**(BaseSpider):

name = "baidu"

*# 发起初始请求*

**def** start\_requests(self):

**yield** Request("http://www.baidu.com", name=self.name)

**yield** Request("http://www.baidu.com", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python2", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python3", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python4", name=self.name)

*# 对响应进行解析*

**def** parse(self, response):

**print**(response)

**print**(response.url)

**yield** response.url

**if** \_\_name\_\_ == '\_\_main\_\_':

*# 在实例化master时可以传递多个不同网站的爬虫, 所以可以把爬虫定义为字典格式的多个爬虫, 字典的值为爬虫类*

spiders = {BaiduSpider.name: BaiduSpider}

*# 实例化master*

master = Master(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG)

*# 运行master, 注意master和slave的运行是分开的, 不能同时开始运行, 要先运行一个, 再运行另一个*

master.run()

*# 实例化slave*

slave = Slave(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG)

slave.run()

运行一个redis容器, 同时把redis代码中的redis地址修改为redis容器的地址.

先后启动master端和slave端, 对代码进行测试.

运行usage.py爬虫文件, 此时会报错, request\_manager.\_\_init\_\_.py中的

from request\_manager.utils.redis\_tools import get\_redis\_queue\_cls 导包时错误. 要修改为

from spidersystem.request\_manager.utils.redis\_tools import get\_redis\_queue\_cls

##### downloader.py：添加其他下载器

使用tornado的httpclient来发起请求获取响应.

class tornado.httpclient.HTTPClient(async\_client\_class=None, \*\*kwargs) 是发送同步的请求, class tornado.httpclient.AsyncHTTPClient 发送的是异步的请求.

同步客户端的使用

http\_client = httpclient.HTTPClient()

**try**:

response = http\_client.fetch("http://www.google.com/")

print(response.body)

**except** httpclient.HTTPError **as** e:

*# HTTPError is raised for non-200 responses; the response*

*# can be found in e.response.*

print("Error: " + str(e))

**except** **Exception** **as** e:

*# Other errors are possible, such as IOError.*

print("Error: " + str(e))

http\_client.close()

tornado中使用http\_client对象的fetch方法来发送请求. 查看fetch方法的源码, fetch方法的第一个参数是request对象, 可以是一个字符串格式的url, 或者tornado的HTTPRequest对象. 查看tornado.httpclient.HTTPRequest的参数.

class tornado.httpclient.HTTPRequest(url, method='GET', headers=None, body=None, auth\_username=None, auth\_password=None, auth\_mode=None, connect\_timeout=None, request\_timeout=None, if\_modified\_since=None, follow\_redirects=None, max\_redirects=None, user\_agent=None, use\_gzip=None, network\_interface=None, streaming\_callback=None, header\_callback=None, prepare\_curl\_callback=None, proxy\_host=None, proxy\_port=None, proxy\_username=None, proxy\_password=None, proxy\_auth\_mode=None, allow\_nonstandard\_methods=None, validate\_cert=None, ca\_certs=None, allow\_ipv6=None, client\_key=None, client\_cert=None, body\_producer=None, expect\_100\_continue=False, decompress\_response=None, ssl\_options=None)

如果想要使用tornado来发送网络请求, 就要构建出tornado的请求对象, 交给fetch方法来获取响应

**def** fetch(self, request, \*\*kwargs):

*"""Executes a request, returning an `HTTPResponse`.*

*The request may be either a string URL or an `HTTPRequest` object.*

*If it is a string, we construct an `HTTPRequest` using any additional*

*kwargs: ``HTTPRequest(request, \*\*kwargs)``*

*If an error occurs during the fetch, we raise an `HTTPError` unless*

*the ``raise\_error`` keyword argument is set to False.*

*"""*

response = self.\_io\_loop.run\_sync(functools.partial(

self.\_async\_client.fetch, request, \*\*kwargs))

**return** response

import requests

from tornado.httpclient import HTTPClient, HTTPRequest

from spidersystem.response import Response

class RequestsDownloader(object):

def fetch(self, request):

if request.method.upper() == "GET":

response = requests.get(request.url\_with\_query, headers=request.headers)

elif request.method.upper() == "POST":

response = requests.get(request.url\_with\_query, data=request.body, headers=request.headers)

else:

raise Exception("Only support GET or POST method ")

return Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

class TornadoDownloader(object):

def \_\_init\_\_(self):

self.httpclient = HTTPClient()

def fetch(self, request):

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method.upper(), headers=request.headers)

response = self.httpclient.fetch(tornado\_request)

return Response(request=request, status\_code=response.code, url=response.effective\_url, headers=response.headers, body=response.buffer.read())

def \_\_del\_\_(self):

self.httpclient.close()

**修改downloader.py, 添加如下代码**

**from** **tornado.httpclient** **import** HTTPClient, HTTPRequest, AsyncHTTPClient

**from** **spidersystem.response** **import** Response

**class** **RequestsDownloader**(object):

**def** fetch(self, request):

**if** request.method.upper() == "GET":

response = requests.get(request.url\_with\_query, headers=request.headers)

**elif** request.method.upper() == "POST":

response = requests.get(request.url\_with\_query, data=request.body, headers=request.headers)

**else**:

**raise** **Exception**("Only support GET or POST method ")

**return** Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

**class** **TornadoDownloader**(object):

**def** \_\_init\_\_(self):

self.httpclient = HTTPClient()

**def** fetch(self, request):

**print**("tornado 同步客户端发送的请求")

*# 创建出tornado的HTTPRequest对象. tornado中的method要大写, 所以这里要加上一个upper()转换为大写.*

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method.upper(), headers=request.headers)

*# 使用httpclient对象的fetch方法, 传入tornado\_request对象, 发送请求获取响应.*

*# 这里得到的response是tornado的response*

tornado\_response = self.httpclient.fetch(tornado\_request)

*# 想要返回符合我们自定义的Response类的response对象, 必须手动进行构建, 查看tornado.httpclient.HTTPResponse查看如何获取其中的参数. response.buffer为类文件对象, 必须使用read()方法才能得到响应体的内容*

**return** Response(request= tornado\_request, status\_code= tornado\_response.code, url= tornado\_response.effective\_url, headers= tornado\_response.headers, body= tornado\_response.buffer.read())

**def** \_\_del\_\_(self):

self.httpclient.close()

修改main.py中的class Slave中的downloader为TornadoDownloader

**from** **spidersystem.downloader** **import** RequestsDownloader, TornadoDownloader

**class** **Slave**(object):

*# slave 端要实现的功能为, 从master端的请求队列中取出来一个请求, 交给响应下载器来获取响应*

**def** \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

*# 实例化filter\_queue, 即请求过滤队列, 以把响应中提取到的新的请求保存到请求过滤队列中, 这里传递的参数与对应的master中的参数相同, 才能把请求保存到master的请求过滤队列中*

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

*# 实例化下载器对象*

*# self.downloader = RequestsDownloader()*

*# 实例化tornado下载器对象*

self.downloader = TornadoDownloader()

*# 爬虫一般是在外部编写的, 编写好之后传递过来使用, 并且一般是多个爬虫. 新建 SpiderSystem/usage.py 爬虫文件, 在usage.py爬虫文件中实例化master和slave, 在实例化slave时把爬虫spiders传递进来. 因为运行时是从 usage.py 来运行slave端和master端的, 而usage.py与spidersystem包是在同一目录下的, 所以spidersystem中导包的时候都要从spidersysytem根目录下来导入*

self.spiders = spiders

self.request\_queue = request\_queue

#### 异步模式实现

**使用tornado的异步客户端结合asyncio实现异步**

##### downlower.py: 加入tornado的异步客户端

import requests

from tornado.httpclient import HTTPClient, HTTPRequest, AsyncHTTPClient

from spidersystem.response import Response

class RequestsDownloader(object):

def fetch(self, request):

if request.method.upper() == "GET":

response = requests.get(request.url\_with\_query, headers=request.headers)

elif request.method.upper() == "POST":

response = requests.get(request.url\_with\_query, data=request.body, headers=request.headers)

else:

raise Exception("Only support GET or POST method ")

return Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

class TornadoDownloader(object):

def \_\_init\_\_(self):

self.httpclient = HTTPClient()

def fetch(self, request):

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method, headers=request.headers)

response = self.httpclient.fetch(tornado\_request)

return Response(request=request, status\_code=response.code, url=response.effective\_url, headers=response.headers, body=response.buffer.read())

def \_\_del\_\_(self):

self.httpclient.close()

# 使用pycurl，前提必须安装pycurl

AsyncHTTPClient.configure("tornado.curl\_httpclient.CurlAsyncHTTPClient")

class AsyncTornadoDownloader(object):

def \_\_init\_\_(self):

self.httpclient = AsyncHTTPClient()

async def fetch(self, request):

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method, headers=request.headers)

response = await self.httpclient.fetch(tornado\_request)

return Response(request=request, status\_code=response.code, url=response.effective\_url, headers=response.headers, body=response.buffer.read())

**import** **requests**

**from** **tornado.httpclient** **import** HTTPClient, HTTPRequest, AsyncHTTPClient

**from** **spidersystem.response** **import** Response

*# 因为main.py中的slave方法都修改为异步了, usage.py中也修改为异步执行的方式了, 想要使用异步的方式执行RequestsDownloader, 也要对requests.get和requests.post进行ioloop.run\_in\_executor的改写*

**class** **RequestsDownloader**(object):

**def** fetch(self, request):

**if** request.method.upper() == "GET":

response = requests.get(request.url\_with\_query, headers=request.headers)

**elif** request.method.upper() == "POST":

response = requests.post(request.url\_with\_query, data=request.body, headers=request.headers)

**else**:

**raise** **Exception**("Only support GET or POST method ")

**return** Response(request=request, status\_code=response.status\_code, url=response.url, headers=response.headers, body=response.content)

*# 因为main.py中的slave方法都修改为异步了, usage.py中也修改为异步执行的方式了, 想要使用异步的方式执行TornadoDownloader, 也要对httpclient.fetch进行ioloop.run\_in\_executor的改写*

**class** **TornadoDownloader**(object):

**def** \_\_init\_\_(self):

self.httpclient = HTTPClient()

async **def** fetch(self, request):

**print**("tornado 同步客户端发送的请求")

*# 创建出tornado的HTTPRequest对象*

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method.upper(), headers=request.headers)

*# 使用httpclient对象的fetch方法, 传入tornado\_request对象, 发送请求获取响应.*

*# 这里得到的response是tornado的response*

*# 因为这里的httpclient不是异步的, 所以httpclient方法返回的也不是await支持的三种对象中的任一种, 所以不能直接使用await来改写同步TornadoDownloader中的fetch方法. 必须要使用run\_in\_executor方法来改写*

*# tornado\_response = await self.httpclient.fetch(tornado\_request)*

tornado\_response = run\_in\_executor(self.httpclient.fetch(tornado\_request))

*# 想要返回符合我们自定义的Response类的response对象, 必须手动进行构建, 查看tornado.httpclient.HTTPResponse查看如何获取其中的参数. response.buffer为类文件对象, 必须使用read()方法才能得到响应体的内容*

**return** Response(request=tornado\_request, status\_code=tornado\_response.code, url=tornado\_response.effective\_url, headers=tornado\_response.headers, body=tornado\_response.buffer.read())

**def** \_\_del\_\_(self):

self.httpclient.close()

*# tornado异步发起请求*

*# 异步时会对httpclient进行复用, 所以此时不要求必须手动关闭httpclient的连接了.*

**class** **AsyncTornadoDownloader**(object):

**def** \_\_init\_\_(self):

self.httpclient = AsyncHTTPClient()

*# tornado 5.1 版本的协程是基于asyncio实现的, 想要使用异步, 就要把函数定义为协程函数, 在有耗时的网络io操作时时使用await来实现异步并发*

async **def** fetch(self, request):

tornado\_request = HTTPRequest(request.url\_with\_query, method=request.method, headers=request.headers)

*# await后面要么跟协程对象, 要么是future对象, 要么是task对象, 这里fetch函数返回的就是一个协程对象*

response = await self.httpclient.fetch(tornado\_request)

**return** Response(request=request, status\_code=response.code, url=response.effective\_url, headers=response.headers, body=response.buffer.read())

##### main.py: 修改run对应方法为异步代码

import asyncio

import tornado.ioloop

from spidersystem.downloader import RequestsDownloader, TornadoDownloader, AsyncTornadoDownloader

......

class Slave(object):

def \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

self.downloader = AsyncTornadoDownloader()

self.spiders = spiders

self.request\_queue = request\_queue

async def handle\_request(self):

io\_loop = tornado.ioloop.IOLoop.current()

# 1. 获取请求对象

future = io\_loop.run\_in\_executor(None, self.request\_manager.get\_request, self.request\_queue)

request = await future

# 2. 发起请求

response = await self.downloader.fetch(request)

# 3. 处理响应

spider = self.spiders[request.name]()

for result in spider.parse(response):

if result is None:

raise Exception("parse方法未返回任何结果")

elif isinstance(result, Request):

await io\_loop.run\_in\_executor(None, self.filter\_queue.put, result)

# self.filter\_queue.put(result)

else:

clean\_result = spider.data\_clean(result)

spider.data\_save(clean\_result)

async def run(self):

while True:

# 列表内协程个数，控制着并发数

await asyncio.wait([self.handle\_request(),

self.handle\_request()])

**import** **threading**

**import** **asyncio**

**import** **tornado.ioloop**

**from** **spidersystem.downloader** **import** RequestsDownloader, TornadoDownloader, AsyncTornadoDownloader

**from** **spidersystem.request\_manager** **import** RequestManager

**from** **spidersystem.request** **import** Request

**from** **spidersystem.request\_manager.utils.redis\_tools** **import** get\_redis\_queue\_cls

Queue = get\_redis\_queue\_cls("fifo")

**class** **Master**(object):

*# master中的执行是以同步的方式执行过滤的, 也可以把它改写成异步的方式, 但注意要加上redis锁, 来保证同一时间只有一个slave在进行过滤操作.*

**def** \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

self.spiders = spiders

self.request\_queue = request\_queue

**def** run\_start\_requests(self):

**for** spider **in** self.spiders.values():

**for** r **in** spider().start\_requests():

self.request\_manager.add\_request(r, self.request\_queue)

**def** run\_filter\_request(self):

**while** True:

**for** request **in** self.filter\_queue.get():

self.request\_manager.add\_request(request, self.request\_queue)

**def** run(self):

*# 这两个函数的执行也是同步的, 只有前一个函数执行结束后才能开始执行下一个函数. 假设有100 0000 个请求, 就会消费较长的时间才能执行完第一个函数, 所以要进行异步的改造. 把这两个函数分别放在两个子线程中去执行. 这样这两个函数都是以异步的方式执行了.*

*# self.run\_start\_requests()*

*# self.run\_filter\_request()*

*# 对于 run\_start\_requests 函数, 遍历完spiders后对应的线程就退出了, 但run\_filter\_request函数是死循环, 它对应的线程会一直在运行.*

threading.Thread(target=self.run\_start\_requests).start()

threading.Thread(target=self.run\_filter\_request).start()

**class** **Slave**(object):

**def** \_\_init\_\_(self, spiders, request\_queue, request\_manager\_config):

self.filter\_queue = Queue("filter\_queue", host="10.211.55.3")

self.request\_manager = RequestManager(\*\*request\_manager\_config)

self.downloader = AsyncTornadoDownloader()

self.spiders = spiders

self.request\_queue = request\_queue

*# fetch是一个协程函数, 则所有调用它的函数也必须是协程函数*

*# 一旦使得asyncio把函数修改为协程之后, 函数中任何存在堵塞或耗时操作的代码都应该使用await改写为异步的. 但await后面只能跟协程对象, future对象, 或task对象.*

async **def** handle\_request(self):

*# 1. 获取请求对象*

*# 获取请求对象时, 是从redis队列中获取的, get\_request方法中的block参数默认为True, 当队列为空时, 获取请求的操作就是堵塞的. 所以这里要进行异步的改写. 但get\_request方法不是一个协程函数. 想要修改, 就要把request\_manager模块中的所有的函数都要修改为协程形式的, 任务量比较大.*

*# 这里可以使用ioloop的run\_in\_executor方法, 把一个函数放在进程或线程池中执行, 然后再把结果返回给之前的线程中使用, 这样也能把函数的执行变成是异步的. 对于没有办法或比较难以改成协程的函数, 都可以使用这种方法来异步执行.*

*# request = self.request\_manager.get\_request(self.request\_queue)*

io\_loop = tornado.ioloop.IOLoop.current()

*# run\_in\_executor第一个参数为None, 就表示使用的是默认的线程池, 第二个参数是要执行的函数, 第三个参数是函数的参数. 返回一个future对象*

future = io\_loop.run\_in\_executor(None, self.request\_manager.get\_request, self.request\_queue)

*# 使用 io\_loop.run\_in\_executor 来改造之后, 遇到堵塞或耗时的操作都会在子线程(池)中去执行, 不会对主线程造成堵塞, 就变成异步的了*

request = await future

*# 2. 发起请求, 使用await发起异步请求*

response = await self.downloader.fetch(request)

*# 3. 处理响应*

spider = self.spiders[request.name]()

**for** result **in** spider.parse(response):

**if** result **is** None:

**raise** **Exception**("parse方法未返回任何结果")

**elif** isinstance(result, Request):

*# put方法也是一个网络io操作, 也需要进行异步改写. 因为没有返回值, 就可以直接使用await来执行了.*

*# self.filter\_queue.put(result)*

await io\_loop.run\_in\_executor(None, self.filter\_queue.put, result)

**else**:

clean\_result = spider.data\_clean(result)

*# data\_save方法因为涉及到磁盘io操作, 也可以用run\_in\_executor修改为异步执行的.*

spider.data\_save(clean\_result)

*# 只要一个地方使用了协程函数, 那么任何调用它的地方也必须是协程函数*

async **def** run(self):

**while** True:

*# 在循环中直接使用await来调用协程函数, 循环中的多个协程函数之间仍然是同步的*

*# await self.handle\_request()*

*# 协程嵌套协程时, 要使用asyncio.wait, asyncio.gather, asyncio.as\_complete方法一次性的await多个协程函数出去, 这些协程函数之间才是异步的. 这里有几个协程函数, 就表示有几个并发, 可以通过列表中协程函数的个数来控制并发量*

await asyncio.wait([

self.handle\_request(),

self.handle\_request()

])

##### 启动slave时：

......

if \_\_name\_\_ == '\_\_main\_\_':

spiders = {BaiduSpider.name: BaiduSpider}

# Master(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG).run()

slave = Slave(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG)

import tornado.ioloop

io\_loop = tornado.ioloop.IOLoop.current()

io\_loop.run\_sync(slave.run)

修改usage.py

**from** **spidersystem.spider** **import** BaseSpider

**from** **spidersystem.request** **import** Request

**from** **spidersystem.main** **import** Master, Slave

REQUEST\_MANAGER\_CONFIG = {

"queue\_type": "fifo",

*# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中*

*# 注意：redis的key会被queue\_name重写，因此此处不用写*

"queue\_kwargs": {"host":"10.211.55.3", "port":6379},

"filter\_type": "redis",

*# 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中*

*# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写*

"filter\_kwargs": {"redis\_key":"redis\_filter", "redis\_host":"10.211.55.3"},

}

REQUEST\_QUEUE = "request\_queue"

**class** **BaiduSpider**(BaseSpider):

name = "baidu"

**def** start\_requests(self):

**yield** Request("http://www.baidu.com", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python2", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python3", name=self.name)

**yield** Request("http://www.baidu.com/s?wd=python4", name=self.name)

**def** parse(self, response):

**print**(response)

**print**(response.url)

**yield** response.url

**if** \_\_name\_\_ == '\_\_main\_\_':

spiders = {BaiduSpider.name: BaiduSpider}

*# master的run方法是普通 函数, 没有使用协程*

Master(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG).run()

slave = Slave(spiders, request\_queue=REQUEST\_QUEUE, request\_manager\_config=REQUEST\_MANAGER\_CONFIG)

**import** **tornado.ioloop**

*# 获取当前线程的循环对象*

io\_loop = tornado.ioloop.IOLoop.current()

*# 异步时就要使用 ioloop.run\_sync 来调用slave的run方法, 类似于asyncio中的 run\_until\_complete 方法, 需要传递进一个协程函数. 因为slave.run方法已经是改写为协程函数了, 所以这里可以在run\_sync中使用.*

io\_loop.run\_sync(slave.run)

### 3.2. 升级策略实现

逐步添加其他功能进行升级

#### 使用布隆过滤器进行过滤

**更改REQUEST\_MANAGER\_CONFIG参数: BloomFilter需要传入的参数**

...

REQUEST\_MANAGER\_CONFIG = {

"queue\_type": "fifo",

# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中

# 注意：redis的key会被queue\_name重写，因此此处不用写

"queue\_kwargs": {"host":"10.211.55.3", "port":6379},

"filter\_type": "bloom",

# 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中

# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写

"filter\_kwargs": {"redis\_key":"bloom\_filter", "redis\_host":"10.211.55.3"},

}

...

修改usage.py, 使用布隆过滤器

REQUEST\_MANAGER\_CONFIG = {

"queue\_type": "fifo",

*# 如何配置需查看redis队列的实例化时需要的参数 redis\_queue模块中*

*# 注意：redis的key会被queue\_name重写，因此此处不用写*

"queue\_kwargs": {"host":"10.211.55.3", "port":6379},

*# 使用 布隆过滤器. 查看 request\_manager/\_\_init\_\_.py中确定如何使用某种过滤器*

"filter\_type": "bloom",

*# filter\_kwargs中配置过滤器的参数. 如何配置需查看对应过滤器实例化时如何传参 filter\_class模块中*

*# 注意：redis的key或mysql的表名称 会被filter\_name重写，因此此处不用写*

*# 布隆过滤器的参数. bloomfilter.py, def \_\_init\_\_(self, salts, redis\_host="localhost", redis\_port=6379, redis\_db=0, redis\_key="bloomfilter")*

*# salts参数的作用: 模拟多个hash函数, 求出多个hash值, 映射到布隆过滤器的hash表中. 对于同一批数据去重时, salts只要确定下来, 就不能再更改了.*

"filter\_kwargs": {"redis\_key":"redis\_filter", "redis\_host":"10.211.55.3", "salts": ["a", "b", "c", "d"]},

}

运行usage.py, 启动master端, 就会报错. request\_manager/\_\_init\_\_() line 69: got an unexpected keyword argument "mysql\_table\_name".

self.\_filter\_kwargs["mysql\_table\_name"] = filter\_name

filter\_obj = self.\_filter\_cls(\*\*self.\_filter\_kwargs)

这里的 filter\_cls 表示的是使用的过滤器的类, 当使用布隆过滤器时, 这里的filter\_cls表示 bloomfilter.py中的BloomFilter类, 当使用redis过滤器时, filter\_cls表示 redis\_filter.py 中的RedisFilter, 当使用mysql过滤器时, 使用的是mysql\_filter.py中的MySQLFilter类, 为了实现代码的统一, 上面添加了一个 mysql\_table\_name的键, 而bloomfilter.py中的BloomFilter类中并不能接收这个参数, 修改修改BloomFilter类中的\_\_init\_\_方法, 接收 mysql\_table\_name的参数, 只接收, 不使用即可, 这样就不会报错了.

修复request\_manager的BUG：添加\*args和\*\*kwargs来处理多余的参数 bloomfilter.py

...

class BloomFilter(object):

''''''

def \_\_init\_\_(self, salts, redis\_host="localhost", redis\_port=6379, redis\_db=0, redis\_key="bloomfilter", \*args, \*\*kwargs):

...

然后运行，重点要注意BloomFilter要传入的参数情况：filter\_kwargs中进行修改.

使用Redis desktop manager查看redis数据库, filter-badu的键中就保存了布隆过滤器的hash表, 但由于之前设置了hash表的长度为2\*\*31, 即256M的大小. 直接在RDM中查看会造成软件的卡死.

#### 标记正在处理的请求

丢失请求和失败请求的捕获和处理.

当slave端从 "请求处理队列" 中取出来一个请求进行处理时, 也把这个请求保存到 "处理中的请求" 这个队列中, 当这个请求处理完成时, 再把 "处理中的请求" 队列中同一个请求删除掉. 为了使多个slave端都能进行处理, 还需要使用redis来实现分布式的请求队列, 为了方便的进行存储和删除, 可以使用redis中的哈希, 类似于python中的字典来保存. 使用请求对象的指纹为哈希的键, 而请求对象本身为哈希的值来存储和删除.

同样, 失败的请求也可以使用redis中的哈希来实现.

思路：当salve取出一个请求以后，立刻将该请求记录下来(如redis中)，当处理完成后，再将之删除

需要为每一个请求的设置一个ID（如指纹），便于区分请求

修改以下代码，为Request对象添加ID属性和对应的值

将请求的指纹作为请求的id，如果是布隆过滤器，那就把各个数据的映射下标拿出来作为指纹也就是ID . 例如, 如果有 ["a", "b", "c", "d"] 这个salts值, 会产生4个offset偏移量, 保存到offsets这个列表中. 这个列表中的元素是唯一的, 可以把列表中的各个元素拼接成一个字符串, 作为request请求对象的指纹.

修改bloomfilter.py中：

...

def save(self, data):

'''将原始数据在hash表中一一映射

返回对应的偏移量'''

hash\_values = self.multiple\_hash.get\_hash\_values(data)

offsets = []

for hash\_value in hash\_values:

offset = self.\_get\_offset(hash\_value)

offsets.append(str(offset))

self.client.setbit(self.redis\_key, offset, 1)

return "-".join(offsets) # 将各下标作为ID

...

如果是redis过滤器, 查看information\_summary\_filter/\_\_init\_\_.py, BaseFilter中的save方法, 最后返回的就是一个hash\_value哈希值, 可能是md5或sha256的值. 此时就不需要进行修改.

同时Request对象也默认设置上一个id属性.

修改request.py中的Request类的\_\_init\_\_方法, 增加id的字段.

**class** **Request**(object):

**def** \_\_init\_\_(self, url, method="GET", query={}, body={}, name="test", headers=None, id=None):

self.url = url

self.method = method

**if** **not** isinstance(query, dict):

**raise** **Exception**("query myst be a dict")

self.query = query

**if** **not** isinstance(body, dict):

**raise** **Exception**("body myst be a dict")

self.body = body

self.name = name *# 表示该请求属于哪个爬虫*

self.headers = headers

self.id = id

修改request\_manager/\_\_init\_\_.py中：

...

def add\_request(self, request\_obj, queue\_name, filter\_name=None):

...

if request\_filter.is\_exists(request):

logging.info("发现重复请求: <%s>" % request.url)

else:

# 先设置id，再保存到队列

request.id = request\_filter.mark\_request(request)

if priority is None:

request\_queue.put(request)

else:

request\_queue.put((priority, request))

logging.info("请求添加成功: <%s>" % request.url)

...

**main.py: 实现标记正在处理和处理失败的请求逻辑**

...

import json

import redis

...

class RequestWatcher(object):

def \_\_init\_\_(self):

self.client = redis.StrictRedis(host="10.211.55.3")

def json\_serializer(self, request, error=None):

'''json序列化reqeust对象数据'''

\_ = {

"id": request.id,

"url": request.url,

"method": request.method,

"query": request.query,

"body": request.body,

"name": request.name,

"error": error # 当请求失败时，需要提供失败的原因

}

return json.dumps(\_)

def mark\_processing\_request(self, request):

'''标记正在处理的请求'''

self.client.hset("processing\_request", request.id, self.json\_serializer(request))

def unmark\_processing\_request(self, request):

'''删除标记正在处理的请求'''

self.client.hdel("processing\_request", request.id)

def mark\_fail\_request(self, request, error):

'''标记处理失败的请求'''

self.client.hset("fail\_request", request.id, self.json\_serializer(request,error))

class Slave(object):

def \_\_init\_\_(self, spiders, request\_manager\_config, project\_name):

self.filter\_queue = FIFO\_QUEUE("filter\_queue", host='10.211.55.3') # 请求过滤队列

self.request\_manager = RequestManager(\*\*request\_manager\_config) # 请求管理对象：请求过滤器、请求队列

self.downloader = AsyncTornadoDownloader()

self.spiders = spiders

self.project\_name = project\_name

self.request\_watcher = RequestWatcher()

async def handle\_request(self):

io\_loop = tornado.ioloop.IOLoop.current()

# 1. 获取一个请求

future = io\_loop.run\_in\_executor(None, self.request\_manager.get\_request, self.project\_name)

request = await future

try:

# 立刻标记该请求正在处理

self.request\_watcher.mark\_processing\_request(request)

# 2.发起请求

response = await self.downloader.fetch(request)

# 3. 获取爬虫对象

spider = self.spiders[request.name]()

# 4.处理response

for result in spider.parse(response):

if result is None:

raise Exception("不允许返回None")

elif isinstance(result, Request):

await io\_loop.run\_in\_executor(None, self.filter\_queue.put, result)

# self.filter\_queue.put(result)

else:

# 意味着是一个数据

new\_result = spider.data\_clean(result)

spider.data\_save(new\_result)

except Exception as e:

self.request\_watcher.mark\_fail\_request(request, str(e))

raise

finally:

# 最终删除标记该请求

self.request\_watcher.unmark\_processing\_request(request)

async def run(self):

while True:

await asyncio.wait([

self.handle\_request(),

self.handle\_request()

])

...

**downloader.py: 实现基于selenium+chrome-headless的下载器**

......

import asyncio

import tornado.ioloop

from selenium import webdriver

from selenium.webdriver.common.desired\_capabilities import DesiredCapabilities

from selenium.webdriver.common.proxy import Proxy # 设置driver的代理

from selenium.common.exceptions import WebDriverException # 如果chrome崩溃了，所报的异常

from selenium.webdriver import ChromeOptions # 设置自定义请求头

......

class ChromeHeadlessDownloader(object):

def \_\_init\_\_(self, max=4):

# max 建立的最大的driver个数

self.driver\_pool = [self.\_get\_driver() for i in range(max)]

def \_get\_driver(self):

# httpProxy = {"httpProxy": "121.224.40.26:44120"}

# options = ChromeOptions()

# options.add\_argument('user-agent=Mozilla/5.0 (Linux; Android 5.0; SM-G900P Build/LRX21T) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Mobile Safari/537.36')

# 如果需要使用代理，那么需要在创建driver时进行设置

driver = webdriver.Remote(

command\_executor="http://10.211.55.3:4444/wd/hub",

desired\_capabilities=DesiredCapabilities.CHROME,

# proxy=Proxy(httpProxy), # 设置代理

# options=options # 设置请求头

)

return driver

def \_fetch(self, request, driver):

print("Use Chrome Headless")

driver.get(request.url\_with\_query)

# 注意：由于webdriver没有办法提取出状态码和响应头，所以分别默认用200和cookie来代替

return Response(request, status\_code=200, url=driver.current\_url, headers=driver.get\_cookies(), body=driver.page\_source)

async def fetch(self, request):

while True:

# 判断当前是否有可用的driver，如果没有那就等0.5秒后再来判断

# 直到有可用driver了才继续

if len(self.driver\_pool) >= 1:

driver = self.driver\_pool.pop(0)

break

await asyncio.sleep(0.5)

# 使用线程池来运行

io\_loop = tornado.ioloop.IOLoop.current()

response= await io\_loop.run\_in\_executor(None, self.\_fetch, request, driver)

self.driver\_pool.append(driver)

return response

def \_\_del\_\_(self):

for driver in self.driver\_pool:

driver.quit()

注意：如果大家也是使用docker的selenium/chrome镜像，那么启动时需要加上-v /dev/shm:/dev/shm，来共享主机的内存，否则极大概率频繁出现selenium.common.exceptions.WebDriverException异常，报错chrome崩溃等。

如果是使用appium抓取APP，由于其原理是控制手机模拟点击等操作，然后获取APP上UI展示的数据，其原理不同于selenium+chrome-headless，因此无法在此实现API接口的统一，如果需要，只能单独去设计一套采集系统

#### 模块安装

考虑到需要直接使用spidersystem模块，因此直接安装成为python的第三方模块会更方便使用 setup.py: 与spidersystem同级目录，然后使用python setup.py install安装

from setuptools import setup, find\_packages

setup(

name = "spidersystem" ,

version = "0.1" ,

description = "spidersystem module" ,

author = "author" ,

url = "url" ,

license = "license" ,

packages=find\_packages(exclude=[]),

install\_requires= [

"tornado>=5.1",

"pycurl",

"selenium"

]

)

#### 日志完善

**日志模块使用：** log.py

import logging

import time

def set\_logging\_config(

format, datefmt, filename=, filemode='a', level=logging.DEBUG

):

# 设置基本的日志为文件日志输出

logging.basicConfig(level=level,

format=format,

datefmt=datefmt,

filename=filename,

filemode=filemode)

# 添加终端日志输出

console = logging.StreamHandler()

console.setLevel(logging.INFO)

formatter = logging.Formatter(format)

console.setFormatter(formatter)

logging.getLogger('').addHandler(console)

# 使用方法

if \_\_name\_\_ == '\_\_main\_\_':

format = '%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s'

datefmt='%Y-%m-%d %H:%M:%S'

filename = "./spider.log"

set\_logging\_config(format, datefmt, filename)

while True:

time.sleep(1)

logging.debug("hello debug log")

logging.info("hello info log")

logging.warning("hello warning log")

**日志埋点工作(安装摄像头)：在任何你想要进行日志记录的地方调用日志模块进行记录**

#### ELK日志监控与报警（开启摄像头）

ELK工作原理回顾：

logstash工作原理

Docker中配置和启动ELK：

* [Docker中安装启动Elasticsearch](https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html)
* [Docker中安装启动Logstash](https://www.elastic.co/guide/en/logstash/current/docker.html)
* [Docker中安装启动Kibana](https://www.elastic.co/guide/en/kibana/current/docker.html)

**单节点ELK启动：**

ELK-docker-compose.yaml

version: '2.2'

services:

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:6.2.4

container\_name: elasticsearch

environment:

- cluster.name=docker-cluster

- bootstrap.memory\_lock=true

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

ulimits:

memlock:

soft: -1

hard: -1

volumes:

- esdata:/usr/share/elasticsearch/data

network\_mode: "host"

logstash:

image: docker.elastic.co/logstash/logstash:6.2.4

environment:

- "LS\_JAVA\_OPTS=-Xms256m -Xmx256m"

container\_name: logstash

volumes:

- ./logstash.yaml:/usr/share/logstash/config/logstash.yml

- ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf

- 日志路径:/mylog/\*.log

network\_mode: "host"

depends\_on:

- elasticsearch

kibana:

image: docker.elastic.co/kibana/kibana:6.2.4

container\_name: kibana

volumes:

- ./kibana.yaml:/usr/share/kibana/config/kibana.yml

network\_mode: "host"

depends\_on:

- elasticsearch

volumes:

esdata:

driver: local

kibana.yaml

server.name: kibana

server.host: "0"

# 注意IP地址

elasticsearch.url: http://10.211.55.3:9200

#elasticsearch.username: elastic

#elasticsearch.password: changeme

xpack.monitoring.ui.container.elasticsearch.enabled: true

logstash.yaml

http.host: "0.0.0.0"

# 注意IP地址

xpack.monitoring.elasticsearch.url: http://10.211.55.3:9200

#xpack.monitoring.elasticsearch.username: logstash\_system

#xpack.monitoring.elasticsearch.password: changeme

logstash.conf：注意需要在kafka、rabbitmq、redis等启动后才开始启动elk

input {

http {

port => 8080

}

file {

path => "/mylog/\*.log"

}

}

filter {

grok {

add\_tag => [ "warning" ]

match => { "message" => "WARNING" } # 过滤日志

}

}

output {

# 如果tag是warning（对应grok中的过滤规则），存入kafka、rabbitmq

if "warning" in [tags] {

kafka {

codec => json

topic\_id => "mytopic"

bootstrap\_servers => "10.211.55.3:29092" # 注意IP和端口

}

rabbitmq {

exchange => "spider\_log"

exchange\_type => "direct"

host => "10.211.55.3"

key => "logstash"

}

}

# 存入redis

redis {

host => "10.211.55.3"

db => 1

key => "logstash"

data\_type => "list"

}

}

启动rabbitmq服务，打开宿主机5672端口 运行Rabbitmq消费者：

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

# 注意这里交换机的配置参数需要和logstash中的一致

exchange\_name = "spider\_log"

channel.exchange\_declare(exchange=exchange\_name,

exchange\_type='direct',

durable=True)

# 这里如果不写queue参数，那么意味着，由服务端随机产生一个队列

# 由于还加上了exclusive，所以这就实现了临时队列，连接已关闭，该随机队列也会被删除

# result = channel.queue\_declare(exclusive=True)

# queue\_name = result.method.queue

queue = "new\_queue"

channel.queue\_declare(queue=queue)

# 指当前队列要接受routing\_key指为debug、info或warning的消息

binding\_keys = ["logstash"]

for binding\_key in binding\_keys:

channel.queue\_bind(exchange=exchange\_name,

queue=queue,

routing\_key=binding\_key)

print(' [\*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):

print(" [x] %r:%r" % (method.routing\_key, body))

channel.basic\_consume(callback,

queue=queue,

no\_ack=True)

channel.start\_consuming()

if \_\_name\_\_ == '\_\_main\_\_':

pass

启动kafka服务，打开宿主机32181、29092端口 kafka-docker-compose.yaml

version: '2'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

network\_mode: host

environment:

ZOOKEEPER\_CLIENT\_PORT: 32181

ZOOKEEPER\_TICK\_TIME: 2000

# extra\_hosts:

# - "moby:127.0.0.1"

kafka:

image: confluentinc/cp-kafka:latest

network\_mode: host

depends\_on:

- zookeeper

environment:

KAFKA\_BROKER\_ID: 100

KAFKA\_ZOOKEEPER\_CONNECT: 10.211.55.3:32181 # 注意替换掉IP

# host模式下，可以写localhost，但如果要提供外部访问，那么必须使用宿主机的地址

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://10.211.55.3:29092 # 注意替换掉IP

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

KAFKA\_NUM\_PARTITIONS: 3

kafka消费者并结合wxpy进行日志警报通知

from confluent\_kafka import Consumer, KafkaError, TopicPartition

from wxpy import \*

# 先用其他小号登录微信

bot = Bot(console\_qr=True)

# 指明要将消息发给谁

# search第一个参数表示 对方的微信昵称

friend = bot.friends().search("低调程序员的小号")[0] # 注意下标，如果没有此人，此处会报错

c = Consumer({

'bootstrap.servers': '10.211.55.3:29092',

'group.id': 'mygroup',

'default.topic.config': {

'auto.offset.reset': 'smallest' # largest

}

})

c.subscribe(['mytopic'])

while True:

msg = c.poll(1.0)

if msg is None:

continue

if msg.error():

if msg.error().code() == KafkaError.\_PARTITION\_EOF:

continue

else:

print(msg.error())

break

friend.send(msg.value()) # 利用微信发送消息

print('Received message: {} {} {}'.format(msg.value().decode('utf-8'), msg.topic(), msg.partition()))

c.close()

if \_\_name\_\_ == '\_\_main\_\_':

pass

wxpy[使用方法](https://wxpy.readthedocs.io/zh/latest/index.html)，以及[github地址](https://github.com/youfou/wxpy)

## 4. 项目实战

目的：将doubanbook爬虫的django-orm版本改为使用spidersystem进行抓取

## 5. 总结

**送给大家：虽然我们做的是爬虫，但希望大家不仅仅只是会爬虫**，技术都是相通的。