16 增量式爬虫与增量爬网策略设计

更新时间：2019-06-17 16:09:21

> 机会不会上门来找人，只有人去找机会。
>
> ——狄更斯

对于数据量庞大、更新频繁的网站，一次性的爬网并不能解决问题，这就需要让爬虫进入长期性而且是周期的增量式爬取。本节将介绍应该采用什么样的更新策略去应对此类爬虫项目，对于类似的项目在设计时应用注意哪些设计的原则。

什么是增量式爬虫？

增量式爬虫并不是一种新型的框架或者设计，而是一种具体爬虫技术的应用。增量式爬虫是针对某些会进行不定期更新内容的网站的爬取，此类网站的信息是随时间动态增量变化的爬虫并不能只通过一次的爬取就能获取所有的内容，而是一种异步式的跟随网站更新而增量式地拉取新内容的一种设计。

对于具有持续更新特性的网站我们并不能每次都让爬虫地毯式地全网拉取一次，这样做不但好时长而且每次的数据冗余量也是非常巨大的，由其是对于那些规模庞大的目标网站来说这种一次性拉取的时间可能就得消耗好几天！其次，这种近乎于野蛮的爬取行径是很容易被目标网站所察觉从而可能被直接封禁，更坏的结果可能是会因为干扰他人的网站的正常经营而涉及法律法律责任，所以文明爬网还是很重要的。

增量爬网的策略

增量式爬网主要包括以下几种具体情况：

1. 网站的页面持续增加，原有页面内容不更新。
2. 网站的页面总量不变，页面内容被定期更新。
3. 网站的页面总量增加，页面内容也有可能被更新。

网站的页面持续增加，原有页面内容不更新

这种情况是最容易处理的，只要定期对网站进行一次全面重爬并加入**去重过滤器**，那么每次重爬就不会对已爬取过的页面重新拉取，而只拉取新的页面。

网站的页面总量不变，页面内容被定期更新

对于这种情况有两种策略：

1. 当网站页面总量的规模不大时（万级以下），可采用全面重爬的办法覆盖原有已爬取过的内容。
2. 当网站页面总量有一定规模(万级以上)，则可以考虑先做一次全面的页面爬取并爬取后结果作为历史基线数据。然后可以采用优选法的拉取方式。

什么是优选法呢？其实这个优选法的方式有很多重，目的就是将本需要通过一次性全面无差别的爬取方式改变为针对重点页面，重点数据进行优先获取的目的。当然这种办法实现的前提是目标数据本身存在差异性，可以进行优先级的排序。例如：我们要从豆瓣上爬取每个时期各本书的综合评分，那么就可以根据历史数据中图书的热度（阅读次数）、用户活跃度（评价数）来进行复合优选排序，然后可以用**2/8法则**将最重要的20%的数据先进行快速爬取，而将80%的数据分成多个时段去获取。这是一种比较简单，不需要算法支持的一种优选法。

我也参考过其他一些人的做法，这种会法就点像搜索引擎的方式：

1. 按照反链数来对页面进行悠闲排序 —— 这种做法的好处是反链数越多的内容其质量可能也越高，同时也是各种搜索引擎对页面PV评分的其中一个占比很大的权重值。然而其缺点是作为爬虫你是很难得知到底还有哪些网站还有反向链接指向该页面，充其量只能计算出当前网站内同时指向该页面的链接总数而已，所以我认为这种作法其实并不具备通用性，难以实现。
2. 根据搜索引擎的内容排序来对页面进行优先级划分 —— 将页面链接地址放入搜索引擎排名越靠前的当然越受欢迎，但这种法则需要向搜索引擎发出大量的爬取请求(至少是一个页面则取一次)，为了获取排名的优先级得写多一个针对搜索引擎的爬虫，这样一来感觉就有点得不尝失，工作量倍增。
3. 采用随机数学模型计算网页更新的概率（例如：泊松过程 `scipy.possion`），概率大者优先级高。这种虽然说是很科学的做法，而然这种做法的进入门槛更高，也要求有在统计学方面非常熟悉，开发过程中还得对模型进行各种调节，其实现难度可想而知。

网站的页面总量增加，页面内容也有可能被更新

这种情况则是1，2两种情况的综合。同时需要执行两次爬取过程，第一次采用**去重过滤器**的的办法爬取新增加的页面；第二次则是采用优选法进行分时分量的拉取。

增量爬网的设计原则

针对增量式爬虫具有持久性、数据量爬大、更新频繁的特性，我认为设计时应该遵从以下的几条基本原则：

1. 避免重复爬取 —— 采用高效、精确的去重手段尽可能避免在同一爬取周期内发出重复的请求。
2. 分布爬取分布存储——可以采用分布式结构将爬虫分散到多台具有不同IP的机器上，充分利用硬件资源，同时存储的数据也可视具体应用的体量采用分布式存储的方式分切储存空间。
3. 爬取时间碎片化 —— 将一次性的耗时长的爬取任务在分切到多个时间点上，形成多个不同时间段上的计划任务，可以减少由于爬取过程中出现的不可预期的情况导致全过程失败带来的损失。

总结起来就是分时，分量免重复原则。

APScheduler

避免重复爬取原则所采用的实现手段：去重过滤器和分布式爬虫会在本专栏后面专门的章节中配合实例应用进行讲述。在本节中将侧重介绍**爬取时间碎片化**的方法。

将爬取时间碎片化从技术上是通过**定时任务**的方式实现，Python的定时任务有三个方法

1. 内置的 sched 类
2. 使用Celery
3. APScheduler

在上述的三者中从方便性与实用性上APScheduler是最好的，所以我们还是以APScheduler作为爬虫的定时器。

我们可以通过pip指令将APScheduler安装到爬虫项目的虚环境下:

```
(venv) $ pip install apscheduler
```

APScheduler由四种组件构成，它们分别是：

1. triggers（触发器）：触发器包含调度逻辑，每一个作业有它自己的触发器，用于决定接下来哪一个作业会运行，除了他们自己初始化配置外，触发器完全是无状态的。
2. job stores（作业存储）：用来存储被调度的作业，默认的作业存储器是简单地把作业任务保存在内存中，其它作业存储器可以将任务作业保存到各种数据库中，支持MongoDB、Redis、SQLAlchemy存储方式。当对作业任务进行持久化存储的时候，作业的数据将被序列化，重新读取作业时在反序列化。
3. executors（执行器）：执行器用来执行定时任务，只是将需要执行的任务放在新的线程或者线程池中运行。当作业任务完成时，执行器将会通知调度器。对于执行器，默认情况下选择`ThreadPoolExecutor`就可以了，但是如果涉及到一下特殊任务如比较消耗CPU的任务则可以选择`ProcessPoolExecutor`，当然根据根据实际需求可以同时使用两种执行器。
4. schedulers（调度器）：调度器是将其它部分联系在一起，一般在应用程序中只有一个调度器，应用开发者不会直接操作触发器、任务存储以及执行器，相反调度器提供了处理的接口。通过调度器完成任务的存储以及执行器的配置操作，如可以添加。修改、移除任务作业。

APScheduler提供了多种调度器，可以根据具体需求来选择合适的调度器，常用的调度器有：

- `BlockingScheduler`：适合于只在进程中运行单个任务的情况，通常在调度器是你唯一要运行的东西时使用。
- `BackgroundScheduler`: 适合于要求任何在程序后台运行的情况，当希望调度器在应用后台执行时使用。
- `AsyncIOScheduler`：适合于使用asyncio框架的情况
- `GeventScheduler`: 适合于使用gevent框架的情况
- `TornadoScheduler`: 适合于使用Tornado框架的应用
- `TwistedScheduler`: 适合使用Twisted框架的应用
- `QtScheduler`: 适合使用QT的情况

配置调度器

APScheduler提供了许多不同的方式来配置调度器，你可以使用一个配置字典或者作为参数关键字的方式传入。你也可以先创建调度器，再配置和添加作业，这样你可以在不同的环境中得到更大的灵活性。

1)下面一个简单的示例：

```
import time
from apscheduler.schedulers.blocking import BlockingScheduler

def test_job():
    print time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time()))

scheduler = BlockingScheduler()

#该示例代码生成了一个BlockingScheduler调度器，使用了默认的默认的任务存储MemoryJobStore，以及默认的执行器ThreadPoolExecutor，并且最大线程数为10。
scheduler.add_job(test_job, 'interval', seconds=5, id='test_job')

#该示例中的定时任务采用固定时间间隔（interval）的方式，每隔5秒钟执行一次。
#并且还为该任务设置了一个任务id

scheduler.start()
```

2）如果想执行一些复杂任务，如上边所说的同时使用两种执行器，或者使用多种任务存储方式，并且需要根据具体情况对任务的一些默认参数进行调整。可以参考下面的方式。（http://apscheduler.readthedocs.io/en/latest/userguide.html）

第一种方式：参数式定义

```
from pytz import utc
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.mongodb import MongoDBJobStore
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ThreadPoolExecutor, ProcessPoolExecutor


jobstores = {
    'mongo': MongoDBJobStore(), # 创建MongoDB的存储作为后备数据库
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite') # 默认采用SQLite作为任务数据库
}

executors = {
    'default': ThreadPoolExecutor(20),
    'processpool': ProcessPoolExecutor(5)
}

job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler(jobstores=jobstores, executors=executors, job_defaults=job_defaults, timezone=utc)
```

第二种方式：

```
from apscheduler.schedulers.background import BackgroundScheduler

scheduler = BackgroundScheduler({
    'apscheduler.jobstores.mongo': {
         'type': 'mongodb'
    },
    'apscheduler.jobstores.default': {
        'type': 'sqlalchemy',
        'url': 'sqlite:///jobs.sqlite'
    },
    'apscheduler.executors.default': {
        'class': 'apscheduler.executors.pool:ThreadPoolExecutor',
        'max_workers': '20'
    },
    'apscheduler.executors.processpool': {
        'type': 'processpool',
        'max_workers': '5'
    },
    'apscheduler.job_defaults.coalesce': 'false',
    'apscheduler.job_defaults.max_instances': '3',
    'apscheduler.timezone': 'UTC',
})
```

第三种方式：

```
from pytz import utc

from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ProcessPoolExecutor


jobstores = {
    'mongo': {'type': 'mongodb'},
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': {'type': 'threadpool', 'max_workers': 20},
    'processpool': ProcessPoolExecutor(max_workers=5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler()
scheduler.configure(jobstores=jobstores, executors=executors, job_defaults=job_defaults, timezone=utc)
```

对任务作业的基本操作：

添加作业有两种方式：

可以直接调用`add_job()`

使用`scheduled_job()`修饰器

而`add_job()`是使用最多的，它可以返回一个`apscheduler.job.Job`实例，因而可以对它进行修改或者删除，而使用修饰器添加的任务添加之后就不能进行修改。

例如使用`add_job()`添加作业：

```
#!/usr/bin/env python
#-*- coding:UTF-8
import time
import datetime
from apscheduler.schedulers.blocking import BlockingScheduler

def job1(f):
    print time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())), f

def job2(arg1, args2, f):
    print f, args1, args2

def job3(**args):
    print args

'''
APScheduler支持以下三种定时任务：
cron: crontab类型任务
interval: 固定时间间隔任务
date: 基于日期时间的一次性任务
'''
scheduler = BlockingScheduler()
#循环任务示例
scheduler.add_job(job1, 'interval', seconds=5, args=('循环',), id='test_job1')
#定时任务示例
scheduler.add_job(job1, 'cron', second='*/5', args=('定时',), id='test_job2')
#一次性任务示例
scheduler.add_job(job1, next_run_time=(datetime.datetime.now() + datetime.timedelta(seconds=10)), args=('一次',), id='test_
job3')
'''
传递参数的方式有元组(tuple)、列表(list)、字典(dict)
注意：不过需要注意采用元组传递参数时后边需要多加一个逗号
'''
#基于list
scheduler.add_job(job2, 'interval', seconds=5, args=['a','b','list'], id='test_job4')
#基于tuple
scheduler.add_job(job2, 'interval', seconds=5, args=('a','b','tuple',), id='test_job5')
#基于dict
scheduler.add_job(job3, 'interval', seconds=5, kwargs={'f':'dict', 'a':1,'b':2}, id='test_job7')
print scheduler.get_jobs()
scheduler.start()

#带有参数的示例
scheduler.add_job(job2, 'interval', seconds=5, args=['a','b'], id='test_job4')
scheduler.add_job(job2, 'interval', seconds=5, args=('a','b',), id='test_job5')
scheduler.add_job(job3, 'interval', seconds=5, kwargs={'a':1,'b':2}, id='test_job6')
print scheduler.get_jobs()
scheduler.start()
```

或者使用`scheduled_job()`修饰器来添加作业：

```
@sched.scheduled_job('cron', second='*/5' , id='my_job_id',)
def test_task():
    print("Hello world!")
```

启动调度器

可以使用`start()`方法启动调度器，`BlockingScheduler`需要在初始化之后才能执行`start()`,对于其他的Scheduler，调用`start()`方法都会直接返回，然后可以继续执行后面的初始化操作。

例如:

```
from apscheduler.schedulers.blocking import BlockingScheduler

def my_job():
    print "Hello world!"

scheduler = BlockingScheduler()
scheduler.add_job(my_job, 'interval', seconds=5)
scheduler.start()
```

关闭调度器

使用下边方法关闭调度器：

```
scheduler.shutdown()
```

默认情况下调度器会关闭它的任务存储和执行器，并等待所有正在执行的任务完成，如果不想等待，可以进行如下操作：

```
scheduler.shutdown(wait=False)
```

用定时任务启动网易爬虫

如果要将网易爬虫彻底运行一次那可能有得耗费好几天的时间，毕竟这是一个爬通网易全网的爬虫，网易的数据量是何其巨大。除了本章之前所介绍的几种优化这种大规模爬虫的方法以外，要节省时间就应该对爬虫所爬取的范围进行精准的划分，毕竟将对方爬光的这种情况发生的几率并不高。就本示例而言，我们是可以应用增量式爬网的思维，将需要获取的数据进行分区划分，从区域入手分布于多个时间段进行，这样就可以将一个需要耗费巨量时间的任务拆分成多个，对局部获取到的数据进行优先的处理而不是等待所有的任务完成后再进行下一步的数据分析处理工作。

所以我们可以将网易爬虫的爬取目标以栏目的内容来划分，例如：我们只爬取"游戏"、"人间"与"体育"这三个栏目，每次就直接从分栏目进入只爬取该栏目下的内容，这样对数据的获取效率会更高。然后将爬取这三个栏目独立看成执行三次任务并分别在周一早上11点，周三深夜11点和周五的凌晨3点进行，通过对内容与分时的划分后，当爬虫持续地运行一段时间，我们还可以从这种划分中观察到网易对这些频道中的内容的更新频率与更新方式，这样在以后便于以后对爬虫任务的调整，使爬取的时间更短数据目标更加明确。

将频道作为起始点而不是网易首页的话，我们只要按照我在第二章第二节"新闻供稿专用爬虫开发实践"中所介绍的将种子页参数化的办法就可以得以实现了，只要在`NeteaseSpider`中将`starts_url`的内容调整一下即可：

```
class NeteaseSpider(CrawlSpider):
    name = 'netease'
    allowed_domains = ['163.com']
    urls = 'https://www.163.com/'
    start_urls = urls.split(',')
```

这样我们就可以在命令行执行`crawl`时通过参数传递改变起始页`url`的位置。例如起动专爬游戏栏目：

```
(venv) $ scrapy crawl netease -a urls='https://game.163.com,https://play.163.com'
```

接下来我们就需要编写一个入口文件，在这个文件就启动APScheduler来定时启动相应的爬取任务：

```
# coding:utf-8
import os
from pytz import utc
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ProcessPoolExecutor

jobstores = {
    # 'mongo': {'type': 'mongodb'},
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}

executors = {
    'default': {'type': 'threadpool', 'max_workers': 20},
    'processpool': ProcessPoolExecutor(max_workers=5)
}

job_defaults = {
    'coalesce': False,
    'max_instances': 3
}

scheduler = BlockingScheduler()
scheduler.configure(jobstores=jobstores, executors=executors, job_defaults=job_defaults, timezone=utc)

# 此任务每周1上午11点执行
@scheduler.scheduled_job('cron', day_of_week='1', hour=11)
def crawl_games():
    os.system("scrapy crawl netease -a url='https://play.163.com,https://game.163.com'")

# 此任务每周3晚上11点执行
@scheduler.scheduled_job('cron', day_of_week='3', hour=21)
def crawl_sports():
    os.system("scrapy crawl netease -a url='https://sports.163.com'")

# 此任务每周5凌晨3点执行
@scheduler.scheduled_job('cron', day_of_week='5', hour=3)
def crawl_renjian():
    os.system("scrapy crawl netease -a url='https://renjian.163.com'")

if __name__ == '__main__':
    try:
        print("爬取任务管理已启动按ctrl+C退出执行")
        scheduler.start() # 启动任务管理器
    except (KeyboardInterrupt, SystemExit):
        scheduler.shutdown() # 关闭任务管理器
```

我们只需要执行以下的指命就可以运行这个定时任务了：

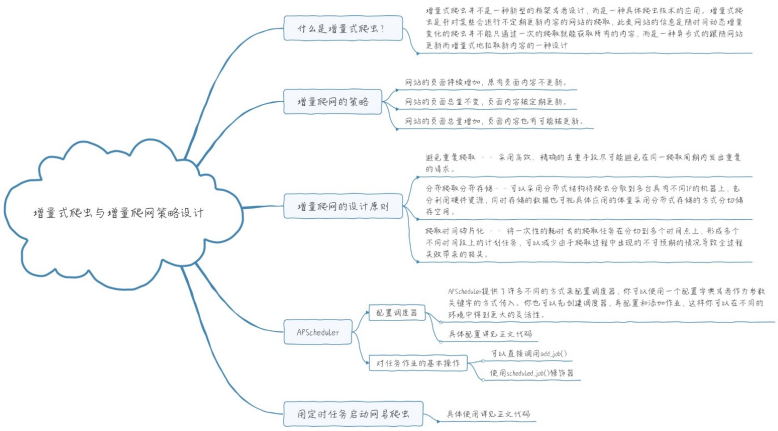
```
(venv) $ python scheduled.py
```

小结

本节是给出了一种"以战养战"以增量、持久的方式长期地爬取目标网站的一种设计思路。在面对复杂的应用时应该紧记以下几点就可以做到"化繁为简"：

1. 区域化爬取的目标内容，尽量让爬虫执行精准的内容爬取任务缩短执行的时长。
2. 碎片化爬取时间，将连续的长任务变成时间上分布的短任务。

这样做就能让我们长期运作的爬虫从根本上提高其运作的效率。



> 注：你可以去GitHub获取本章源代码

15 为网易爬虫配置存储大规模数据存储

17 数据提取过程中的类型化方法

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论