

13 去重处理— 高性能爬虫调优技术

更新时间：2019-06-14 14:35:18



没有引发任何行动的思想都不是思想，而是梦想。

更多一手资源请+V：Andyqc1——马丁
qq：3118617541

目标网站中的网页之间的链接可能是有序的也可能是无序的，甚至可能是具有很大重复性的，甚至某些网站采用页面互链的方式形成“链接死循环”诱杀爬虫，对于增量式爬网而言那些被爬取过的数据则没有必要重新获取，由其像对于网易此类老牌的信息门户内链接错综复杂，循环链接极多，面对这些情况我们就需要采用“去重处理”过滤那些没有必要爬取的页面跳出链接的死循环。

什么时候应该进行去重处理？

去重处理可以避免将重复性的数据保存到数据库中以造成大量的冗余性数据。

不要在获得蜘蛛爬网结果后进行内容过滤，这样做只不过是避免后端数据库出现重复数据。去重处理对于一次性爬取是有效的，但对于增量式爬网则恰恰相反。对于持续性长的增量式爬网，应该进行“前置过滤”，这样可以有效地减少蜘蛛出动的次数。

在发出请求之前检查蜘蛛是否曾爬取过该 URL，如果已爬取过，则让蜘蛛直接跳过该请求以避免重复出动。除了重复的 URL 指纹，还应该加上404与500错误的URL过滤，因为即使目标网站上没有反爬网机制，但绝大多数的Web服务器程序都会有对404与500错误的记录。过多的404与500很容易暴露蜘蛛的痕迹，因此加入对异常URL的筛选是非常有必要的。

Scrapy 是默认加载内置的去重过滤器的，那么我们可以先从这个去重过滤器的源代码中学习一下它的基本实现原理。Scrapy提供了一个很好的请求指纹过滤器（Request Fingerprint duplicates filter - `scrapy.dupefilters.RFPDuperFilter`），当它被启用后，会自动记录所有成功返回响应的请求的URL并将其以文件（`requests.seen`）方式保存在项目目录中。请求指纹过滤器的原理是为每个URL生成一个指纹并记录下来，一旦当前请求的URL在指纹库中有记录，就自动跳过该请求。

接下来我从 `scrapy.dupefilters.RFPDufilter` 源码中抽取其核心代码（其中的部分辅助性的代码被我略去）来了解这个去重的原理：

```
class RFPDufilter(BaseDufilter):
    """Request Fingerprint duplicates filter"""

    def __init__(self, path=None, debug=False):
        self.file = None # 指纹文件
        self.fingerprints = set() # 指纹变量
        # 这里部分内容略去
        if path:
            self.file = open(os.path.join(path, 'requests.seen'), 'a+')
            self.file.seek(0)
            self.fingerprints.update(x.rstrip() for x in self.file)

    def request_seen(self, request):
        """
        读取或更新重复的URL指纹
        """
        fp = self.request_fingerprint(request)
        if fp in self.fingerprints:
            return True
        self.fingerprints.add(fp)
        if self.file:
            self.file.write(fp + os.linesep)

    def request_fingerprint(self, request):
        return request_fingerprint(request)

# close(), log() 与 from_settings() 三个函数的代码略去
```

`RFPDufilter` 的代码非常容易理解，分成两部实现：

1. 在构造时(`__init__`)读取现有的URL指纹库文件(没有就创建一个)到一个 `fingerprints` 变量内，该变量是 `set` 类型所以它是一个具有唯一性的集合，指纹不会重复。
2. 每次发出请求之前Scrapy会调用 `request_seen` 方法，如果该方法返回 `False` 请求就会被正常发出，相反则会被剔除。

默认情况下这个过滤器是被自动启用的。当然也可以根据自身的需求编写自定义的过滤器，继承 `scrapy.dupefilters.BaseDufilter` 来开发自定义的过滤器。

```

class BaseDupeFilter(object):

    @classmethod
    def from_settings(cls, settings):
        """
        我们可以通过这个方法从settings.py文件中读取过滤器的配置
        """
        return cls()

    def request_seen(self, request):
        """
        返回当前的请求是否已重复
        """
        return False

    def open(self):
        """
        当过滤器被打开时执行的代码
        """
        pass

    def close(self, reason):
        """
        当过滤器被关闭时执行的代码
        """
        pass

    def log(self, request, spider):
        """
        记录请求已被过滤
        """
        pass

```

由于 `scrapy.dupefilters.RFPDupeFilter` 采用文件方式保存指纹库，对于增量爬取且只用于短期运行的项目还能应对。一旦遇到爬取量巨大的场景时，这个过滤器就显得不太适用了，因为指纹库文件会变得越来越大会，过滤器在启动时会一次性将指纹库中所有的 URL 读入，导致消耗大量内存。

`scrapy.dupefilters.RFPDupeFilter` 是被默认加载的，当然我们也可以在 `settings.py` 内显式地将它改成我们自己的定义的去重过滤器：

```

DUPEFILTER_CLASS = 'netease_crawler.dupefilters.RFPDupeFilter'

```

编写基于 Redis 的去重过滤器

对于本示例中的增量式的网易爬虫而言 `RFPDupeFilter` 当爬取数据量变得越来越大时就很容易因为读取指纹库时使用内存过大而导致整个爬虫性能下降，最终会因为内存耗尽而死机。

所以我们可以采用相同的原理将 URL 指纹存储到 Redis 数据库中，借助 Redis 超高的读写性能来优化由于 `RFPDupeFilter` 所带来的性能瓶颈。

Redis 的基本介绍

Redis 是一款开源的，基于 BSD 许可的，高级键值 (key-value) 缓存 (cache) 和存储 (store) 系统。由于 Redis 的键包括 string, hash, list, set, sorted set, bitmap 和 hyperloglog，所以常常被称为数据结构服务器。你可以在这些类型上面运行原子操作，例如，追加字符串，增加哈希中的值，加入一个元素到列表，计算集合的交集、并集和差集，或者是从有序集合中获取最高排名的元素。

Redis 的作用与好处

为了满足高性能，Redis 采用内存 (in-memory) 数据集 (dataset)。根据你的使用场景，你可以通过每隔一段时间转储数据集到磁盘，或者追加每条命令到日志来持久化。持久化也可以被禁用，如果你只是需要一个功能丰富，网络化的内存缓存。

Redis 是由 ANSI C 语言编写的，在无需额外依赖下，运行于大多数 POSIX 系统，如 Linux、*BSD、OS X。Redis 是在 Linux 和 OS X 两款操作系统下开发和充分测试的，所以我推荐 Linux 为部署环境。Redis 也可以运行在 Solaris 派生系统上，如 SmartOS，但是支持有待加强。没有官方支持的 Windows 构建版本，但是微软开发和维护了一个 64 位 Windows 的版本。

更详细的内容可以参考[Redis的中文网](#)或者[Redis英文官网](#)

安装 Redis | Docker

比起手工安装Redis我更愿意使用Docker的镜像来起动一个Redis实例，这既不会"污染"我的开发环境也不会给我的开发机器带来过多的性能损耗。

Docker是一种现代最流行的虚化容器技术，它不像虚拟机那样需要独立分走你系统的资源，而是采用"共享"的方式来使用你的机器。对于完全没有接触过Docker的童鞋也可以保持淡定，就当是执行几个命令行运行起来的一个服务，免去你安装经常下载、安装工具的痛苦。

关于Docker的详细资料可以到以下的网站中详细学习：

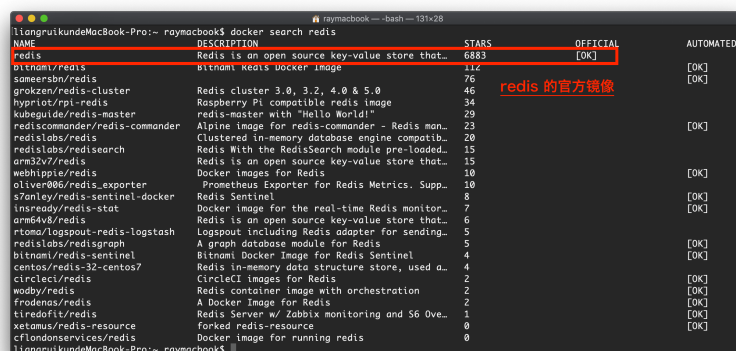
- [菜鸟学院的Docker专栏](#)
- [Docker中文社区站](#)

Docker实在是太火了，它的资料到处都是，为了节省篇幅在此就不过多讲述怎么来安装它了，其它平台的读者可以参考以下的链接：

- [CentOS docker 安装](#)
- [Windows](#)
- [macOS](#)

进入docker的命令行，查找redis的镜像：

```
$ docker search redis
```

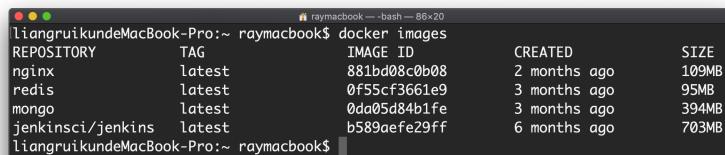


NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
redis	Redis is an open source key-value store that...	6883	[OK]	
bitnami/redis	bitnami Redis Docker Image	214		[OK]
sameersbn/redis		76		[OK]
grokzen/redis-cluster	Redis cluster 3.0, 3.2, 4.0 & 5.0	46		
hyprlat/rpi-redis	Raspberry PI compatible redis image	34		
kubequide/redis-master	redis-master with "Hello World!"	29		
rediscommander/redis-commander	Alpine image for redis-commander - Redis man...	23		[OK]
redislabs/redis	Clustered in-memory database engine compatib...	20		
redislabs/redisearch	Redis With the RedisSearch module pre-loaded...	15		
arm32v7/redis	Redis is an open source key-value store that...	15		
webhippie/redis	Docker images for Redis	10		[OK]
oliver006/redis.exporter	Prometheus Exporter for Redis Metrics. Supp...	10		
37onley/redis-sentinel-docker	Redis Sentinel	8		[OK]
insready/redis-stat	Docker image for the real-time Redis monitor...	7		[OK]
arm64v8/redis	Redis is an open source key-value store that...	6		
rtoma/logspout-redis-logstash	Logspout including Redis adapter for sending...	5		
redislabs/redisgraph	A graph database module for Redis	5		[OK]
bitnami/redis-sentinel	Bitnami Docker Image for Redis Sentinel	4		[OK]
centos/redis-32-centos7	Redis in-memory data structure store, used a...	4		
circlect/redis	CircleCI Images for Redis	2		[OK]
noby/redis	Redis container image with orchestration	2		[OK]
Frodinas/redis	A Docker Image for Redis	2		[OK]
tiredofit/redis	Redis Server w/ Zabbix monitoring and S6 Ove...	1		[OK]
actamus/redis-resource	Forked redis-resource	0		[OK]
cflondonservices/redis	Docker image for running redis	0		

找到官方的 docker 镜像 **redis** (在 OFFICIAL 栏下有 [OK] 字样的都是官方提供的)，然后使用 **pull** 指令从docker的镜像库里拉取 **redis** 到本地机器

```
docker pull redis
```

完成后可以用 `images` 指令查看本地已拉取的镜像:



```
liangruikundeMacBook-Pro:~ raymacbook$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx                latest             881bd08c0b08       2 months ago       109MB
redis                latest             0f55cf3661e9       3 months ago       95MB
mongo                latest             0da05d84b1fe       3 months ago       394MB
jenkinsci/jenkins    latest             b589aefe29ff       6 months ago       703MB
liangruikundeMacBook-Pro:~ raymacbook$
```

使用 `run` 指令启动 `redis` 容器实例:

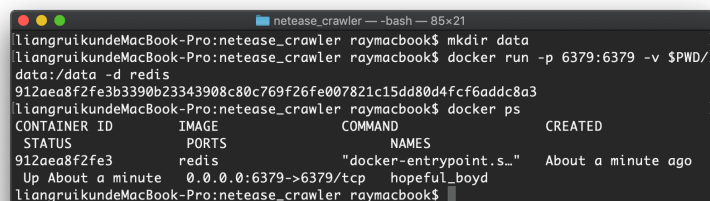
```
$ mkdir data
$ docker run -p 6379:6379 -v $PWD/data:/data -d redis
```

这里要稍微做一个简单的解析, 首先我在当前目录创建一个 `data` 目录, 目的是将 `redis` 的数据文件存到这个目录下, 然后指令的参数是以下的意义:

- `-p 6379:6379` : 将容器的6379端口映射到主机的6379端口
- `-v $PWD/data:/data` : 将主机中当前目录下的 `data` 挂载到容器的 `/data` 目录
- `-d` 就表示在后台运行
- `redis` 是指定采用哪个镜像来实例化容器

执行以上指令以后就容器就启动了, 可以用 `ps` 指令查看当前运行了哪些容器实例:

```
$ docker ps
```



```
liangruikundeMacBook-Pro:netease_crawler raymacbook$ mkdir data
liangruikundeMacBook-Pro:netease_crawler raymacbook$ docker run -p 6379:6379 -v $PWD/
data:/data -d redis
912aea8f2fe3b3390b23343908c80c769f26fe007821c15dd80d4fcf6addc8a3
liangruikundeMacBook-Pro:netease_crawler raymacbook$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
912aea8f2fe3       redis              "docker-entrypoint.s" About a minute ago
Up About a minute  0.0.0.0:6379->6379/tcp hopeful_boyd
liangruikundeMacBook-Pro:netease_crawler raymacbook$
```

现在 `redis` 就摆在那里可以用了, 只要连接到本机的6379端口就可以直接使用 `redis` 的存储服务了。

Python 中连接 Redis

接下的一步是我们应该如何在 Python 里面连接redis呢？在 Python 世界中redis的工具包也非常的多，我个人比较喜欢使用redis这个源自于redis官方的Python客户端，它的代码很清晰简单也非常容易使用而且同时支持Python2.x和Python3。

记得使用前在虚环境先安装redis:

```
$ pip install redis
```

然后按以下的代码，两行就可以实例化redis对象直接在代码里面使用

```
from redis import Redis
redis = Redis(port=6379)
```

Set 类型的介绍

准备好Redis的工具环境，接下来就是要在Redis中建立指纹库了。Redis是一个键-值(Key-Value)型的NoSQL数据库，它的值有很多种类型，其中我们只需要用到Set这个类型就可以实现指纹库的存储与读取。

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 $O(1)$ 。集合中最大的成员数为 $2^{32} - 1$ (4294967295, 每个集合可存储40多亿个成员)。

编写 RedisDupeFilter

RedisDupeFilter 完全可以按照 REPDupeFilter 的逻辑来改写:

1. 从 BaseDupeFilter 直接继承，重写相应方法
2. 从 settings.py 读取 Redis 的连接信息用以实例化 redis 对象
3. 为每个URL生成指纹存到Redis的Set集合中
4. 对每次进行过滤的重复性URL在日志中记录输出

具体代码实现如下:

更多一手资源请+V : AndyqcI
qq: 3118617541

```
# coding: utf-8
from redis import Redis
from scrapy.utils.request import request_fingerprint
from scrapy.dupefilters import BaseDupeFilter
import logging

class RedisDupeFilter(BaseDupeFilter):
    """
    基于Redis的去重过滤器
    """

    def __init__(self, host='localhost', port=6379, db=0):
        self.redis = Redis(host=host, port=port, db=db)
        self.logger = logging.getLogger(__name__)

    @classmethod
    def from_settings(cls, settings):
        """
        读取配置信息
        """
        host = settings.get('REDIS_HOST', 'localhost')
        redis_port = settings.getint('REDIS_PORT')
        redis_db = settings.get('REDIS_DUP_DB')
        return cls(host, redis_port, redis_db)

    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        key = 'UrlFingerprints'
        if not self.redis.sismember(key, fp):
            self.redis.sadd(key, fp)
            return False
        return True

    def request_fingerprint(self, request):
        """
        用Hash生成URL指纹
        :param request:
        :return:
        """
        return request_fingerprint(request)

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered', spider=spider)
```

与 `RFPDupeFilter` 相比两个过滤的代码基本相同，只是将的文件型的指纹库换成了Redis，这里调用了两个redis的常用指令：`sismember` 和 `sadd`：

```
self.redis.sismember(key, fp)
```

这个代码的意思是判断是否具有该的键-值，该方法的调用就相当于在redis-cli的命令行工具中运行 `SMEMBERS` 指令：

```
127.0.0.1:6379>SMEMBERS UrlFingerprints
```

如果没有这个URL在指纹库中就是不存在的，通过 `sadd` 方法将该URL指纹添加到redis的Set指纹库中。

```
self.redis.sadd(key, fp)
```

上述代码就相当于在 `redis-cli` 命令工具中的 `SADD`

`request_fingerprint` 函数

更多一手资源请+V：Andyqc1
qq：3118617541

如果你仔细阅读了上述的代码你会发现有一个老是重复出现的函数 `request_footprint`，这个函数的实际作用是将URL转换为哈希值的。由于 `RFPDupeFilter` 是将指纹存到文件的，当然存固定长度的哈希值会使得指纹库更小而且校对速度会比字符串更高，但是如果用在Redis中的话就会有点多余了，以下是回顾上文中的一段话：

Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 $O(1)$ 。集合中最大的成员数为 $2^{32} - 1$ (4294967295, 每个集合可存储40多亿个成员)。

也就是说使用Redis的Set集合来存储指纹库的话根本我们就不需要耗费多余的CPU资源来调用 `request_footprint` 计算哈希值，直接将URL存到Redis就可以了。下面是简化后的代码：

```
# coding: utf-8
from redis import Redis
from scrapy.dupefilters import BaseDupeFilter
import logging

class RedisDupeFilter(BaseDupeFilter):

    def __init__(self, host='localhost', port=6379, db=0):
        self.redis = Redis(host=host, port=port, db=db)
        self.logger = logging.getLogger(__name__)

    @classmethod
    def from_settings(cls, settings):
        host = settings.get('REDIS_HOST', 'localhost')
        redis_port = settings.getint('REDIS_PORT')
        redis_db = settings.get('REDIS_DUP_DB')
        return cls(host, redis_port, redis_db)

    def request_seen(self, request):
        fp = request.url
        key = 'UrlFingerprints'
        if not self.redis.sismember(key, fp):
            self.redis.sadd(key, fp)
            return False
        return True

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered', spider=spider)
```

如何读取 `settings.py` 中的配置

在上述代码中的 `from_settings` 方法就是用于从 `settings.py` 文件中读取配置值的，所有的配置都将被Scrapy自动读取并存到该方法的 `settings` 字典对象中，本示例为了使 `RedisDupeFilter` 可以变得更加通用，特意将Redis的连接信息放至到 `settings.py` 中。也就是说按上述代码的定义，这个网易爬虫的 `settings.py` 就多了三个可附加的配置项：

```
# settings.py
REDIS_HOST = 'localhost' # 配置Redis的安装地址
REDIS_PORT = 6379        # Redis的连接端口
REDIS_DUP_DB = 0         # Redis去重过滤器的数据库
```

`from_settings` 还附带了一个装饰器 `@classmethod`：

```
@classmethod
def from_settings(self, settings):
    pass
```


意思就是说这是一个类方法或者可以理解为"静态方法", 就是当类还没有被实例化时仍然可以被调用的方法类型。这个方法同时也是一个“工厂方法”用于实例化自己, 在实例化的过程中将参数传给自己的构造函数 `__init__`。

Scrapy的其它插件都大量采用了这种模式来进行全局配置的传入。

log 与 stats 收集运行时输出信息

`RedisDupeFilter` 过滤器中的最后个方法 `log` 用于将被过滤的信息添加到全局状态变量 `crawler.states` 中。这是个非常意义的小处理。爬虫本身就是一个非可视化的服务, 每次的爬取时间又非常的短暂, 处理的问题越多、越复杂可能出现异常的机会也就越大, 而且爬虫所面对的数据可以说是千奇百怪什么样的都有, 如果没有一个集中的地方去收集每个处理中的状态, 去记录每个关键操作, 以后就很难去进行深入的数据分析工作了。

由于这个 `log` 方法极为简单, 所以就更容易理解与学习希望不要将其忽略, 本专栏后续的章节中还会专门去讲述关于日志与状态收集的内容, 所以在此暂时不作展开。

配置并启用去重过滤器 `RedisDupeFilter`

现在只要在 `settings.py` 中将 `DUPEFILTER_CLASS` 的配置指向 `RedisDupeFilter` 就可以使用这个高效的去重过滤器了。

```
DUPEFILTER_CLASS = 'netease_crawler.dupefilters.RedisDupeFilter'
```

完成配置后运行:

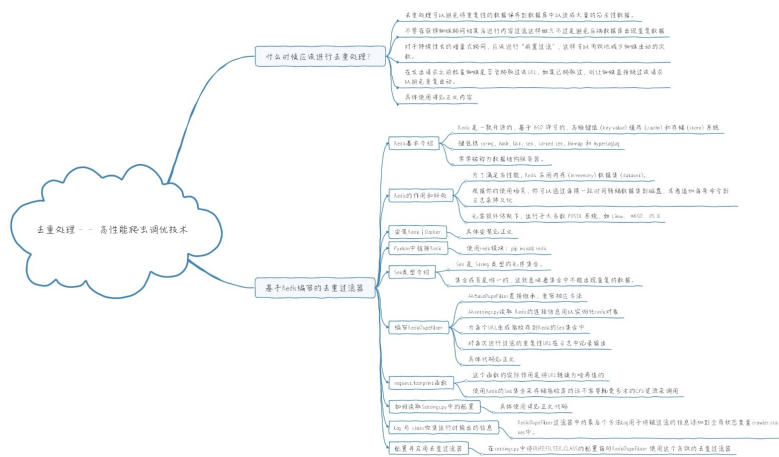
```
$ scrapy crawl netease
```

仅仅通过控制台的LOG输出速度就明显感觉到整个爬虫的性能番了好几番!

小结

至此我们已接触了两种去重过滤器, 接下来将介绍一个更强大的布隆过滤器。在此之前有必要对去重过滤器的应用先做一个简单的小结, 归纳它们适用的场合才能更好地发挥它们的作用。

- 当数据量不大时 (大约在200MB内), 可以直接在内存中进行去重处理 (例如, 可以使用 `set()` 进行去重), 而更省事又能对去重状态进行持久化的办法就是采用 `scrapy.dupefilters.RFPDupeFilter`;
- 当数据量在5GB以内时, 建议采用上文中的 `RedisDupeFilter` 进行去重, 当然这要求服务器的内存必须大于5GB, 否则Redis可能会将机器的内存耗光;
- 当数据量达到10~100GB级别时, 由于内存有限, 就必须用“位”来去重, 才能够满足需求。而布隆过滤器就是将去重对象映射到几个内存“位”, 通过几个位的0/1值来判断一个对象是否已经存在, 以应对海量级的请求数据的重复性校验。



← 12 用 ItemLoader 解决网页数据多样性的问题

14 高效的布隆过滤器 - RedisBloomDuperFilter →

更多一手资源请+V : AndyqcI
aa : 3118617541