合并代码

load函数就是从日志中提取合格的数据的生成器函数。
它可以作为dispatcher函数的数据源。
原来写的handler函数处理一个字典的'datetime'字段，不能处理日志抽取函数extract返回的字典，提供一个新的函数。

```
import random
import datetime
import time
from queue import Queue
import threading
import re

# 日志处理正则
pattern = '''(?P<remote>[\d.]{7,}) - - \[(?P<datetime>[\w/: +-]+)\] \
"(?P<method>\w+) (?P<url>\S+) (?P<protocol>[\w\d/.]+)" (?P<status>\d+) (?P<length>\d+) \
".+" "(?P<useragent>.+)"'''
# 编译
regex = re.compile(pattern)

conversion = {
    'datetime': lambda timestr: datetime.datetime.strptime(timestr, '%d/%b/%Y:%H:%M:%S %z'),
    'status': int,
    'length': int
}

def extract(logline: str) -> dict:
    """返回字段的字典，如果返回None说明匹配失败"""
    m = regex.match(logline)
    if m:
        return {k:conversion.get(k, lambda x:x)(v) for k,v in m.groupdict().items()}
    else:
        return None # 或输出日志记录

# 装载日志数据，数据源
def load(path):
    """装载日志文件"""
    with open(path) as f:
        for line in f:
            fields = extract(line)
            if fields:
                yield fields
            else:
                continue # TODO 解析失败就抛弃，或者打印日志


def window(src: Queue, handler, width: int, interval: int):
    """窗口函数
```

```
    :param iterator: 数据源，生成器，用来拿数据
    :param handler: 数据处理函数
    :param width: 时间窗口宽度，秒
    :param interval: 处理时间间隔，秒
    """
    if interval > width:  # width < interval不处理
        return

    start = datetime.datetime.strptime('20170101 000000', '%Y%m%d %H%M%S')
    current = datetime.datetime.strptime('20170101 010000', '%Y%m%d %H%M%S')
    buffer = []  # 窗口中的待计算数据
    delta = datetime.timedelta(seconds=width - interval)

    while True:
        # 从数据源获取数据
        data = src.get()
        if data:  # 攒数据
            buffer.append(data)  # 存入临时缓冲等待计算
            current = data['datetime']

        # 每隔interval计算buffer中的数据一次
        if (current - start).total_seconds() >= interval:
            ret = handler(buffer)
            print('{}'.format(ret))
            start = current

            # 保留buffer中未超出width的数据。如果delta为0，说明width等于interval，buffer直接清空
            buffer = [x for x in buffer if x['datetime'] > current - delta] if delta else []


# 处理函数，送入一批数据计算出一个结果，下为平均值
def handler(iterable):
    return sum(map(lambda x: x['value'], iterable)) / len(iterable)

# 测试函数
def donothing_handler(iterable):
    return iterable

def dispatcher(src):
    # 分发器中记录handler，同时保存各自的队列
    handlers = []
    queues = []

    def reg(handler, width: int, interval: int):
        """注册 窗口函数

        :param handler: 注册的数据处理函数
        :param width: 时间窗口宽度
        :param interval: 时间间隔
        """
        q = Queue()  # 每一个handler自己的数据源queue
        queues.append(q)
```

```
        # 每一个handler都运行在单独的线程中
        t = threading.Thread(target=window, args=(q, handler, width, interval))
        handlers.append(t)

    def run():
        for t in handlers:
            t.start()  # 启动线程，运行所有的处理函数

        for item in src:  # 将数据源取到的数据分发到所有队列中
            for q in queues:
                q.put(item)

    return reg, run


if __name__ == '__main__':
    import sys
    #path = sys.argv[1]
    path = 'test.log'

    reg, run = dispatcher(load(path))

    reg(donothing_handler, 10, 5)  # 注册
    run()  # 运行
```

运行，抛出异常

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "C:\Python3\python366\lib\threading.py", line 916, in _bootstrap_inner
    self.run()
  File "C:\Python3\python366\lib\threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
  File "C:/Users/wayne/PycharmProjects/testo/t2.py", line 66, in window
    if (current - start).total_seconds() >= interval:
TypeError: can't subtract offset-naive and offset-aware datetimes
```

问题在于时间上

```
current 为 2017-04-06 18:09:25+08:00 有时区的称为offset-aware
start 2017-01-01 00:00:00 没有时区称为offset-naive
```

```
import datetime

# offset-naive
current = datetime.datetime.strptime('20170101 010000', '%Y%m%d %H%M%S')
now = datetime.datetime.now()

print(type(current), current.tzinfo)
print(type(now), now.tzinfo)
```

```
print(now - current)

# offset-aware
start = datetime.datetime.strptime('19/Feb/2013:10:23:29 +0800', '%d/%b/%Y:%H:%M:%S %z')
print(type(start), start.tzinfo)

print(now - start) # TypeError: can't subtract offset-naive and offset-aware datetimes
```

所以，修改window函数中的代码，加上时区即可

```
    start = datetime.datetime.strptime('20170101 000000 +0800', '%Y%m%d %H%M%S %z')
    current = datetime.datetime.strptime('20170101 010000 +0800', '%Y%m%d %H%M%S %z')
```

完成分析功能

分析日志很重要，通过海量数据分析就能够知道是否遭受了攻击，是否被爬取及爬取高峰期，是否有盗链等。
百度(Baidu) 爬虫名称(Baiduspider)
谷歌(Google) 爬虫名称(Googlebot)

状态码分析

状态码中包含了很多信息。例如
304，服务器收到客户端提交的请求参数，发现资源未变化，要求浏览器使用静态资源的缓存
404，服务器找不大请求的资源

304占比大，说明静态缓存效果明显。404占比大，说明网站出现了错误链接，或者尝试嗅探网站资源。
如果400、500占比突然开始增大，网站一定出问题了。

```
# 状态码占比
def status_handler(iterable):
    # 时间窗口内的一批数据
    status = {}
    for item in iterable:
        key = item['status']
        status[key] = status.get(key, 0) + 1
    #total = sum(status.values())
    total = len(iterable)
    return {k:v/total for k,v in status.items()}
```

如果还需要什么分析，增加分析函数handler注册就行了

日志文件的加载

目前实现的代码中，只能接受一个路径，修改为接受一批路径。

可以约定一下路径下文件的存放方式：
如果送来的是一批路径，就迭代其中路径。
如果路径是一个普通文件，就直接加载这个文件。
如果路径是一个目录，就遍历路径下所有指定类型的文件，每一个文件按照行处理。
可以提供参数处理是否递归子目录。

```
# 用户提供一个目录或者一批目录列表，读取下面的`*.log`等文本文件，并逐行加载处理。

from pathlib import Path

def load(*paths, encoding='utf-8', ext="*.log", glob=False):
    """装载日志文件"""
    for p in paths:
        path = Path(p)
        if path.is_dir(): # 只处理目录
            if isinstance(ext, str):
                ext = [ext]
            else:
                ext = list(ext)

            for e in ext:  # 按照扩展名递归
                files = path.rglob(e) if glob else path.glob(e)  # 是否递归
                for file in files:
                    with file.open() as f:
                        for line in f:
                            fields = extract(line)
                            if fields:
                                yield fields
                            else:
                                continue  # TODO 解析失败就抛弃，或者打印日志
```

上面的代码问题是，嵌套层次太多了，结合原来的load函数，得到如下代码

```
from pathlib import Path

def loadfile(filename:str, encoding='utf-8'):
    """装载日志文件"""
    with open(filename, encoding=encoding) as f:
        for line in f:
            fields = extract(line)
            if isinstance(fields, dict):
                yield fields
            else:
                continue # TODO 解析失败就抛弃，或者打印日志

def load(*paths, encoding='utf-8', ext="*.log", glob=False):
    """装载日志文件"""
    for p in paths:
        path = Path(p)
        if path.is_dir(): # 只处理目录
            if isinstance(ext, str):
```

```
                ext = [ext]
            else:
                ext = list(ext)

            for e in ext:  # 按照扩展名递归
                files = path.rglob(e) if glob else path.glob(e)  # 是否递归
                for file in files:
                    yield from loadfile(str(file.absolute()), encoding=encoding)
        elif path.is_file():
            yield from loadfile(str(path.absolute()), encoding=encoding)
```

完整代码

使用队列

```
import random
import datetime
import time
from queue import Queue
import threading
import re

# 日志处理正则
pattern = '''(?P<remote>[\d.]{7,}) - - \[(?P<datetime>[\w/: +]+)\] \
"(?P<method>\w+) (?P<url>\S+) (?P<protocol>[\w\d/.]+)" (?P<status>\d+) (?P<length>\d+) \
".+" "(?P<useragent>.+)"'''
# 编译
regex = re.compile(pattern)

conversion = {
    'datetime': lambda timestr: datetime.datetime.strptime(timestr, '%d/%b/%Y:%H:%M:%S %z'),
    'status': int,
    'length': int
}

def extract(logline: str) -> dict:
    """返回字段的字典，如果返回None说明匹配失败"""
    m = regex.match(logline)
    if m:
        return {k:conversion.get(k, lambda x:x)(v) for k,v in m.groupdict().items()}
    else:
        return None # 或输出日志记录

# 装载日志数据，数据源
from pathlib import Path

def loadfile(filename:str, encoding='utf-8'):
    """装载日志文件"""
    with open(filename, encoding=encoding) as f:
        for line in f:
```

```
            fields = extract(line)
            if isinstance(fields, dict):
                yield fields
            else:
                continue # TODO 解析失败就抛弃，或者打印日志

def load(*paths, encoding='utf-8', ext="*.log", glob=False):
    """装载日志文件"""
    for p in paths:
        path = Path(p)
        if path.is_dir(): # 只处理目录
            if isinstance(ext, str):
                ext = [ext]
            else:
                ext = list(ext)

            for e in ext:  # 按照扩展名递归
                files = path.rglob(e) if glob else path.glob(e)  # 是否递归
                for file in files:
                    yield from loadfile(str(file.absolute()), encoding=encoding)
        elif path.is_file():
            yield from loadfile(str(path.absolute()), encoding=encoding)



def window(src: Queue, handler, width: int, interval: int):
    """窗口函数

    :param iterator: 数据源，生成器，用来拿数据
    :param handler: 数据处理函数
    :param width: 时间窗口宽度，秒
    :param interval: 处理时间间隔，秒
    """
    if interval > width:  # width < interval不处理
        return

    start = datetime.datetime.strptime('20170101 000000 +0800', '%Y%m%d %H%M%S %z')
    current = datetime.datetime.strptime('20170101 010000 +0800', '%Y%m%d %H%M%S %z')
    buffer = []  # 窗口中的待计算数据
    delta = datetime.timedelta(seconds=width - interval)

    while True:
        # 从数据源获取数据
        data = src.get()
        if data:  # 攒数据
            buffer.append(data)  # 存入临时缓冲等待计算
            current = data['datetime']

        # 每隔interval计算buffer中的数据一次
        if (current - start).total_seconds() >= interval:
            ret = handler(buffer)
            print('{}'.format(ret))
            start = current
```

```
            # 保留buffer中未超出width的数据。如果delta为0，说明width等于interval，buffer直接清空
            buffer = [x for x in buffer if x['datetime'] > current - delta] if delta else []


# 处理函数，送入一批数据计算出一个结果，下为平均值
def handler(iterable):
    return sum(map(lambda x: x['value'], iterable)) / len(iterable)

# 测试函数
def donothing_handler(iterable):
    return iterable

# 状态码占比
def status_handler(iterable):
    # 时间窗口内的一批数据
    status = {}
    for item in iterable:
        key = item['status']
        status[key] = status.get(key, 0) + 1
    #total = sum(status.values())
    total = len(iterable)
    return {k:v/total for k,v in status.items()}

def dispatcher(src):
    # 分发器中记录handler，同时保存各自的队列
    handlers = []
    queues = []

    def reg(handler, width: int, interval: int):
        """注册 窗口函数

        :param handler: 注册的数据处理函数
        :param width: 时间窗口宽度
        :param interval: 时间间隔
        """
        q = Queue()  # 每一个handler自己的数据源queue
        queues.append(q)

        # 每一个handler都运行在单独的线程中
        t = threading.Thread(target=window, args=(q, handler, width, interval))
        handlers.append(t)

    def run():
        for t in handlers:
            t.start()  # 启动线程，运行所有的处理函数

        for item in src:  # 将数据源取到的数据分发到所有队列中
            for q in queues:
                q.put(item)

    return reg, run
```

```
if __name__ == '__main__':
    import sys
    #path = sys.argv[1]
    path = '.'

    reg, run = dispatcher(load(path))

    reg(status_handler, 10, 5)  # 注册
    run()  # 运行
```

到这里，一个离线日志分析项目基本完成。

1、可以指定文件或目录，对日志进行数据分析
2、分析函数可以动态注册
3、数据可以分发给不同的分析处理程序处理