aCrawler

Release v0.0.5

Nov 18, 2019

Contents:

A powerful web-crawling framework, based on aiohttp.

CHAPTER 1

Feature

- Write your crawler in one Python script with asyncio
- Schedule task with priority, fingerprint, exetime, recrawl...
- Middleware: add handlers before or after task's execution
- Simple shortcuts to speed up scripting
- Parse html conveniently with Parsel
- Parse with rules and chained processors
- Support JavaScript/browser-automation with pyppeteer
- Stop and Resume: crawl periodically and persistently
- Distributed work support with Redis

CHAPTER 2

Installation

To install, simply use pipenv (or pip):

```
$ pipenv install acrawler

(Optional)
$ pipenv install uvloop      #(only Linux/macOS, for faster asyncio event loop)
$ pipenv install aioredis    #(if you need Redis support)
$ pipenv install motor       #(if you need MongoDB support)
$ pipenv install aiofiles    #(if you need FileRequest)
```

2.1 Tutorial

In this tutorial, we will scrape information of popular movies from IMDB

Code is avaliable at Examples.

2.1.1 Start Requests

First, we start our script with rewritting `start_requests()`:

```
from acrawler import Crawler, Request
class IMDBCrawler(Crawler):
    config = {'MAX_REQUESTS': 6}

    async def start_requests(self):
        yield Request('https://www.imdb.com/chart/moviemeter')
```

Here we don't explictly pass `callback` parameter to `Request` because the default `parse()` will automatically be binded as callback function to it for any request yielded from `start_requests()`.

2.1.2 First Callback Parse

Then we rewrite `parse()` to parse the response:

```
class IMDBCrawler(Crawler):

    def parse(self, response):
        for tr in response.sel.css('.lister-list tr'):
            link = tr.css('.titleColumn a::attr(href)').get()
            if link:
                yield Request(response.urljoin(link), callback=self.parse_movie)

        # or using a shortcut method
        yield from response.follow(
            ".lister-list tr .titleColumn a::attr(href)", callback=self.parse_movie
        )
```

During parsing, the most important attribute is *`acrawler.http.Response.sel`*. It is a Parsel `Selector`. In this callback function, we also yield many new tasks *`Request`* and we explictly pass `callback` parameter to them.

2.1.3 Define MovieItem

Then we need to define a new *`ParselItem`* to store results:

```
from acrawler import ParselItem
from pprint import pprint

def process_time(value):
    # a self-defined field processing function
    # process time to minutes
    # '3h 1min' -> 181
    if value:
        res = 0
        segs = value.split(' ')
        for seg in segs:
            if seg.endswith('min'):
                res += int(seg.replace('min',''))
            elif seg.endswith('h'):
                res += 60*int(seg.replace('h',''))
        return res
    else:
        return value

class MovieItem(ParselItem):
    log = True
    css = {
        # just some normal css rules
        # see Parsel for detailed information
        "date": ".subtext a[href*=releaseinfo]::text",
        "rating": "span[itemprop=ratingValue]::text",
        "rating_count": "span[itemprop=ratingCount]::text",
        "metascore": ".metacriticScore span::text",

        # if you provide a list with additional functions,
        # they are considered as field processor function
        "title": ["h1::text", str.strip],
```

(continues on next page)

(continued from previous page)

```
        "time": [".subtext time::text", process_time],

        # the following four fules is for getting all matching values
        # the rule starts with [ and ends with ] comparing to normal rules
        "genres": "[.subtext a[href*=genres]::text]",
        "director": "[h4:contains(Director) ~ a[href*=name]::text]",
        "writers": "[h4:contains(Writer) ~ a[href*=name]::text]",
        "stars": "[h4:contains(Star) ~ a[href*=name]::text]",
    }

    def custom_process(self):
        pprint(self)
```

2.1.4 Parse Movie Page

Then we write our callback function for movie page:

```
class IMDBCrawler(Crawler):

    async def parse_movie(self, response):
        url = response.url_str
        yield MovieItem(response.sel, extra={'url': url.split('?')[0]})
```

Here in this callback function, we yield a new task *MovieItem*, which will execute and collect all information from the page.

We also pass a dictionary to *extra*. During initialing, item's content will be updated from *extra* at first.

2.1.5 Start Crawling

To start crawling, simply write:

```
if __name__ == "__main__":
    IMDBCrawler().run()
```

Here is one of the items:

```
{'date': '26 April 2019 (USA)',
'director': ['Anthony Russo', 'Joe Russo'],
'genres': ['Action', 'Adventure', 'Sci-Fi'],
'metascore': '78',
'rating': '8.8',
'rating_count': '407,691',
'stars': ['Robert Downey Jr.', 'Chris Evans', 'Mark Ruffalo'],
'time': 181,
'title': 'Avengers: Endgame',
'url': 'https://www.imdb.com/title/tt4154796/',
'writers': ['Christopher Markus', 'Stephen McFeely']}
```

2.1.6 Register a Handler

We can define a dummy handler to send a warning if the movie is a horror movie:

```
@register()
class HorrorHandler(Handler):
    family = 'MovieItem'
    logger = get_logger('horrorlogger')

    async def handle_after(self, item):
        if item['genres'] and 'Horror' in item['genres']:
            self.logger.warning(
                "({}) is a horror movie!!!!".format(item['title']))
```

In this case, handler is register to *MovieItem* with a specific family provided:

```
2019-05-24 18:37:22,888 acrawler.horrorlogger WARNING  (Midsommar) is a horror movie!!
↪!!
```

2.1.7 Periodical & Persistent

If we want the crawler supports keyboard interupt(Ctrl-C) and resumes crawling next time, the config *PERSISTENT* should be set.

If we want to recrawl the index page every 4 hour starting from a specific time, we can provide `recrawl` and `exetime` parameters:

```
import time
class IMDBCrawler(Crawler):
    config = {
        'MAX_REQUESTS': 6,
        'PERSISTENT': True,
        'PERSISTENT_NAME': 'IMDBv0.1'
    }

    async def start_requests(self):
        yield Request('https://www.imdb.com/chart/moviemeter',
                      exetime=time.mktime((2019,5,24,18,30,0,0,0,0)),
                      recrawl=4*60*60)
```

2.2 Basic Concepts

2.2.1 About Tasks

Anything in aCrawler is a *Task*, which *execute()* and then may yield new *Task*.

There are several basic Tasks defined here.

- *Request* task executes its default `fetch()` method to make HTTP request. Then task will automatically yield a corresponding *Response* task. You can pass a function to *callback* argument and provide a *family*, which are all passed to the response task.
- *Response* task executes `callback()`. It call all functions in *callbacks* with http response and may yield new task. A *Response* may have several callback functions (which are passed from decorator *callback()* or corresponding request's parameter).
- *Item* task executes its `custom_process()` method, which you can rewrite.
- *ParselItem* extends from *Item*. It accepts a `Selector` and uses Parsel to parse content.

- Any new `Task` yielded from an existing `Task` 's execution will be catched and delivered to scheduler.
- Any new `dictionary` yielded from an existing `Task`'s execution will be catched as *`DefaultItem`*.

2.2.2 About Families

- Each *`Handler`* has only one family. If a handler's family is in a task's families, this handler matches the task and then somes fuctions will be called before and after the task.
- Each task has `families` (defaults to names of all base classes and itself). If you pass `family` to a task, it will be appended to task's families. Specially, a *`Request`* 's user-passed `family` will be passed to its corresponding *`Response`*'s family.
- `family` is also used for decorator *`callback()`* and *`register()`*
  - You can use decorator `@register()` to add a `handler` to crawler. It is also allowed to register a function but you should provide family, position as parameters. If a `handler`'s family is in a `task`'s families, then `handler` matches `task`.
  - You can use decorator `@callback(family='')` to add a callback to response. If `family` in `@callback()` is in a response's families, then callback will be combined to this response.

2.3 Main Interface

2.3.1 Crawler

class acrawler.crawler.**Crawler**(*config=None*, *middleware_config=None*, *request_config=None*)

This is the base crawler, from which all crawlers that you write yourself must inherit.

Attributes:

start_urls = []

Ready for vanilla *`start_requests()`*

parsers = []

Shortcuts for parsing response.

Crawler will automatically append *`acrawler.parser.Parser.parse()`* to response's callbacks list for each parser in parsers.

config = {}

Config dictionary for this crawler. See avaliable options in *setting*.

middleware_config = {}

Key-value pairs for handler-priority.

Examples

Handler with higer priority handles the task earlier. Priority 0 will disable the handler:

```
{
    'some_old_handler': 0,
    'myhandler_first': 1000,
    'myhandler': 500
}
```

request_config = {}
Key-Value pairs will be passed as keyword arguments to every sended *aiohttp.request*.

acceptable keyword params - Dictionary or bytes to be sent in the query string of the new request

data - Dictionary, bytes, or file-like object to send in the body of the request

json - Any json compatible python object

headers - Dictionary of HTTP Headers to send with the request

cookies - Dict object to send with the request

allow_redirects - If set to False, do not follow redirects

timeout - Optional ClientTimeout settings structure, 5min total timeout by default.

middleware = None
Singleton object `acrawler.middleware.middleware`

run()
Core method of the crawler. Usually called to start crawling.

web_add_task_query(*query*)
This method is to deal with web requests if you enable the web service. New tasks should be yielded in this method. And Crawler will finish tasks to send response. Should be overwritten.

Parameters **query** (`dict`) – a multidict.

web_action_after_query(*items*)
Action to be done after the web service finish the query and tasks. Should be overwritten.

add_task(*new_task*, *dont_filter=False*, *ancestor=None*, *flag=1*)
Interface to add new Task to scheduler.

Parameters **new_task** (*Task*) – a Task or a dictionary which will be catched as *DefaultItem* task.

Return type `bool`

Returns True if the task is successfully added.

manager()
Create multiple workers to execute tasks.

next_requests()
This method will be binded to the event loop as a task. You can add task manually in this method.

parse(*response*)
Default callback function for Requests generated by default *start_requests()*.

Parameters **response** (*Response*) – the response task generated from corresponding request.

start_requests()
Should be rewritten for your custom spider.

Otherwise it will yield every url in *start_urls*. Any Request yielded from *start_requests()* will combine *parse()* to its callbacks and passes all callbacks to Response

2.3.2 Base Task

class acrawler.task.**Task**(*dont_filter=False*, *ignore_exception=False*, *priority=0*, *meta=None*, *family=None*, *recrawl=0*, *exetime=0*)

Task is scheduled by crawler to execute.

Parameters

- **dont_filter** (bool) – if True, every instance of the Task will be considered as new task. Otherwise fingerprint will be checked to prevent duplication.
- **ignore_exception** (bool) – if True, any exception catched from the task's execution will not retry the task.
- **fingerprint_func** – A function that receives Task as parameter.
- **priority** (int) – Tasks are scheduled in priority order (higher first). If priorities are same, the one initialized earlier executes first.(FIFO)
- **meta** (Optional[dict]) – additional information about a task. It can be used with *fingerprint*, *execute()* or middleware's methods. If a task's execution yields new task, old task's meta should be passed to the new one.
- **family** – used to distinguish task's type. Defaults to *__class__.__name__*.

tries = None

Every execution increase it by 1. If a task's *tries* is larger than scheduler's max_tries, it will fail. Defaults to 0.

init_time = None

The timestamp of task's initializing time.

last_crawl_time = None

The timestamp of task' last execution time.

exceptions = None

a list to store exceptions occurs during execution

score

Implements its real priority based on expecttime and priority

fingerprint

returns value of _fingerprint().

execute(***kwargs*)

main entry for a task to start working.

Parameters

- **middleware** – needed to call custom functions before or after executing work.
- **kwargs** (Any) – additional keyword args will be passed to _execute()

Return type AsyncGenerator[*Task*, None]

Returns an asyncgenerator yields Task.

2.3.3 HTTP Task

class acrawler.http.**Request**(*url*, *callback=None*, *method='GET'*, *request_config=None*, *status_allowed=None*, *encoding=None*, *links_to_abs=True*, *dont_filter=False*, *ignore_exception=False*, *meta=None*, *priority=0*, *family=None*, *recrawl=0*, *exetime=0*, ***kwargs*)

Request is a Task that execute `fetch()` method.

url

callback

should be a callable function or a list of functions. It will be passed to the corresponding response task.

family

this family will be appended in families and also passed to corresponding response task.

status_allowed

a list of allowed status integer. Otherwise any response task with *status!=200* will fail and retry.

meta

a dictionary to deliver information. It will be passed to `Response.meta`.

request_config

a dictionary, will be passed as keyword arguments to `aiohttp.ClientSession.request()`.

acceptable keyword:

params - Dictionary or bytes to be sent in the query string of the new request

data - Dictionary, bytes, or file-like object to send in the body of the request

json - Any json compatible python object

headers - Dictionary of HTTP Headers to send with the request

cookies - Dict object to send with the request

allow_redirects - If set to False, do not follow redirects

timeout - Optional ClientTimeout settings structure, 5min total timeout by default.

fetch()

Sends a request and return the response as a task.

send()

This method is used for independent usage of Request without Crawler.

class acrawler.http.**Response**(*url*, *status*, *cookies*, *headers*, *request*, *body*, *encoding*, *links_to_abs=False*, *callbacks=None*, ***kwargs*)

Response is a Task that execute parse function.

status

HTTP status code of response, e.g. 200.

url

url as yarl URL

url_str

url as str

sel

a `Selector`. See Parsel for parsing rules.

doc

a `PyQuery` object.

meta
a dictionary to deliver information. It comes from `Request.meta`.

ok
True if *status==200* or status is allowed from `Request.status_allowed`

cookies
HTTP cookies of response (Set-Cookie HTTP header).

headers
A case-insensitive multidict proxy with HTTP headers of response.

history
Preceding requests (earliest request first) if there were redirects.

body
The whole response's body as *bytes*.

text
Read response's body and return decoded *str*

request
Point to the corresponding request object that generates this response.

callbacks
list of callback functions

ok
If the response is allowed by the config of request.

Return type `bool`

update_sel(*source=None*)
Update response's Selector.

Parameters **source** – can be a string or a PyQuery object. if it's None, use *self.pq* as source by default.

open(*path=None*)
Open in default browser

urljoin(*a*)
Accept a str (can be a relative url) or a Selector that has href attributes.

Return type `str`

Returns return a absolute url.

paginate(*css*, *limit=0*, *pass_meta=False*, ***kwargs*)
Follow links and yield requests with same callback functions. Additional keyword arguments will be used for constructing requests.

Parameters

- **css** (*str*) – css selector
- **limit** (int) – max number of links to follow.

follow(*css*, *callback=None*, *limit=0*, *pass_meta=False*, ***kwargs*)
Yield requests in current page using css selector. Additional keyword arguments will be used for constructing requests.

Parameters

- **css** (*str*) – css selector

- **callback** (*callable, optional*) – Defaults to None.
- **limit** – max number of links to follow.

spawn (*item*, *divider=None*, *pass_meta=True*, ***kwargs*)
Yield items in current page Additional keyword arguments will be used for constructing items.

Parameters

- **divider** (*str*) – css divider
- **item** (ParselItem) – item class

parse ()
Parse the response(call all callback funcs) and return the list of yielded results. This method should be used in independent work without Crawler.

Return type list

class acrawler.http.**FileRequest** (*url*, **args*, *fdir=None*, *fname=None*, *skip_if_exists=True*, *callback=None*, *method='GET'*, *request_config=None*, *dont_filter=False*, *meta=None*, *priority=0*, *family=None*, ***kwargs*)
A derived Request to download files.

class acrawler.http.**BrowserRequest** (*url*, **args*, *page_callback=None*, *callback=None*, *method='GET'*, *request_config=None*, *dont_filter=False*, *meta=None*, *priority=0*, *family=None*, ***kwargs*)
A derived Request using *pyppeteer* to crawl pages.

There are two ways to directly deal with *pyppeteer.page.Page*. You can rewrite method *operate_page()* or pass *page_callback* as parameter. Callback function accepts two parameters: *page* and *resposne*.

fetch ()
Sends a request and return the response as a task.

operate_page (*page*, *response*)
Can be rewritten for customed operation on the page. Should be a asyncgenerator to yield new tasks.

2.3.4 Item Task

class acrawler.item.**Item** (*extra=None*, *extra_from_meta=False*, *log=None*, *store=None*, *family=None*, ***kwargs*)
Item is a Task that execute *custom_process()* work. Extending from MutableMapping so it provide a dictionary interface. Also you can use *Item.content* to directly access content.

extra
During initialing, *content* will be updated from extra at first.

content
Item stores information in the *content*, which is a dictionary.

custom_process ()
can be rewritten for customed futhur processing of the item.

class acrawler.item.**ParselItem** (*selector=None*, *extra=None*, *css=None*, *xpath=None*, *re=None*, *default=None*, *inline=None*, *inline_divider=None*, *bindmap=None*, ***kwargs*)
The item working with Parsel.

The item receives Parsel's selector and several rules. The selector will process item's fields with these rules. Finally, it will call processors to process each field.

classmethod bind(*field=None*, *map=False*)
Bind field processor.

class acrawler.item.**DefaultItem**(*extra=None*, *extra_from_meta=False*, *log=None*, *store=None*, *family=None*, ***kwargs*)
Any python dictionary yielded from a task's execution will be cathed as `DefaultItem`.

It's the same as `Item`. But its families has one more member 'DefaultItem'.

class acrawler.item.**Processors**
Processors are used to spawn field processing functions for ParselItem. All the methods are static.

static first()
get the first element from the values

static strip()
strip every string in values

classmethod map(*func*)
apply function to every item of filed's values list

static filter(*func=<class 'bool'>*)
pick from those elements of the values list for which function returns true

static drop(*func=<class 'bool'>*)
If func return false, drop the Field.

static drop_item(*func=<class 'bool'>*)
If func return false, drop the Item.

static to_datetime(*error_drop=False*, *error_keep=False*, *with_time=False*, *regex=None*)
extract datetime, return None if not matched

Parameters

- **error_drop** (*bool, optional*) – drop the field if not matched, defaults to False
- **error_keep** (*bool, optional*) – keep the original value if not matched, defaults to False
- **with_time** (*bool, optional*) – regex with time parsing, defaults to False
- **regex** (*str, optional*) – provided custom regex, defaults to None

static to_date(*error_drop=False*, *error_keep=False*, *regex=None*)
extract date, return None if not matched

Parameters

- **error_drop** (*bool, optional*) – drop the field if not matched, defaults to False
- **error_keep** (*bool, optional*) – keep the original value if not matched, defaults to False
- **with_time** (*bool, optional*) – regex with time parsing, defaults to False
- **regex** (*str, optional*) – provided custom regex, defaults to None

static to_float(*error_drop=False*, *error_keep=False*, *regex=None*)
extract float, return None if not matched

Parameters

- **error_drop** (*bool, optional*) – drop the field if not matched, defaults to False
- **error_keep** (*bool, optional*) – keep the original value if not matched, defaults to False

static to_int (*error_drop=False*, *error_keep=False*, *regex=None*)
extract int, return None if not matched

Parameters

- **error_drop** (`bool, optional`) – drop the field if not matched, defaults to False
- **error_keep** (`bool, optional`) – keep the original value if not matched, defaults to False

2.3.5 Parser

class acrawler.parser.**Parser** (*in_pattern=''*, *follow_patterns=None*, *css_divider=None*, *item_type=None*, *extra=None*, *selectors_loader=None*, *callbacks=None*)
A basic parser.

It is a shortcut class for parsing response. If there are parsers int :attr:Crawler.parsers, then crawler will call Parser's parse method with the response to yield new Request Task or Item Task.

Parameters

- **in_pattern** (`str`) – a string as a regex pattern or a function.
- **follow_patterns** (`Optional[List[str]]`) – a list containing strings as regex patterns or a function.
- **item_type** (`Optional[ParselItem]`) – a custom item class to store results.
- **css_divider** (`Optional[str]`) – You may have many pieces in one response. Yield them in different selectors by providing a css_divider.
- **selectors_loader** (`Optional[Callable]`) – a function accepts selector and yield selectors. Default one deals with css_divider.
- **callbacks** (`Optional[List[Callable]]`) – additional callbacks.

parse_links (*response*)
Follow new links and yield Request in the response.

parse_items (*response*)
Get items from all selectors in the loader.

parse (*response*)
Main function to parse the response.

2.3.6 Handlers

class acrawler.middleware.**Handler** (*family=None*, *func_before=None*, *func_after=None*, *func_start=None*, *func_close=None*)
A handler wraps functions for a specific task.

priority = 500
A handler with higher priority will be checked with task earlier. A handler with priority 0 will be disabled.

family = '_Default'
Associated with *Task*'s families. One handler only has one family. If a handler's family is in a task's families, this handler matches the task and then somes fuctions will be called before and after the task.

handle_after (*task*)
Then function called after the execution of the task.

handle_before (*task*)
Then function called before the execution of the task.

on_close ()
When Crawler closes, this method will be called.

on_start ()
When Crawler starts(before *start_requests()*), this method will be called.

acrawler.middleware.**register** (*family=None*, *position=None*, *priority=None*)
Shortcut for middleware.register()

acrawler.handlers.**callback** (*family*)
The decorator to add callback function.

class acrawler.middleware.**_Middleware**

register (*family=None*, *position=None*, *priority=None*)
The factory method for creating decorators to register handlers to middleware. Singledispathed for differenct types of targets.

If you register a function, you must give *position* and *family*. If you register a Handler class, you can register it without explicit parameters:

```
@register(family='Myfamily', position=1)
def my_func(task):
    print("This is called before execution")
    print(task)

@register()
class MyHandler(Handler):
    family = 'Myfamily'

    def handle(self, task):
        print("This is called before execution")
        print(task)
```

Parameters

- **family** (Optional[str]) – received as the *Handler.family* of the Handler.
- **priority** (Optional[int]) – received as the *Handler.priority* of the Handler.
- **position** (Optional[int]) – represents the role of function. Should be a valid int: 0/1/2/3

  0 - *Handler.on_start()*

  1 - *Handler.handle_before()*

  2 - *Handler.handle_after()*

  3 - *Handler.on_close()*

append_func (*func*, *family=None*, *position=None*, *priority=None*)
constructor a handler class from given function and register it.

Parameters

- **func** (*[type]*) –
- **family** (*str, optional*) – , defaults to None

- **position** (*int, optional*) – 0, 1, 2, 3, defaults to None
- **priority** (*int, optional*) – , defaults to None

class acrawler.handlers.**ItemToRedis**(*family=None*, *func_before=None*, *func_after=None*, *func_start=None*, *func_close=None*)

family = 'Item'
Family of this handler.

address = 'redis://localhost'
An address where to connect. Can be one of the following:

- a Redis URI — `"redis://host:6379/0?encoding=utf-8"`;
- a (host, port) tuple — `('localhost', 6379)`;
- or a unix domain socket path string — `"/path/to/redis.sock"`.

maxsize = 10
Maximum number of connection to keep in pool.

items_key = 'acrawler:items'
Key of the list at which item's content is inserted.

handle_after(*item*)
Then function called after the execution of the task.

on_close()
When Crawler closes, this method will be called.

on_start()
When Crawler starts(before `start_requests()`), this method will be called.

class acrawler.handlers.**ItemToMongo**(*family=None*, *func_before=None*, *func_after=None*, *func_start=None*, *func_close=None*)

family = 'Item'
Family of this handler.

address = 'mongodb://localhost:27017'
a full mongodb URI, in addition to a simple hostname

db_name = ''
name of targeted database

col_name = ''
name of targeted collection

handle_after(*item*)
Then function called after the execution of the task.

on_close()
When Crawler closes, this method will be called.

on_start()
When Crawler starts(before `start_requests()`), this method will be called.

class acrawler.handlers.**ResponseAddCallback**(*family=None*, *func_before=None*, *func_after=None*, *func_start=None*, *func_close=None*)
(before execution) add `Parser.parse()` to `Response.callbacks`.

handle_before(*response*)
Then function called before the execution of the task.

class acrawler.handlers.**ExpiredWatcher**(**args*, ***kwargs*)
Maintain a expired Event.

You can set this event and then meth'custom_expired_worker' will be waken up to do bypassing work. You should overwrite meth'custom_on_start' if needed rather than default one.

Parameters

- **expired** – a Event to tell the worker that your token is expired.
- **last_handle_time** – a timestamp when the last work happened.
- **ttl** – if *set* signal is sent at a time that less than *last_handle_time* + *ttl*, it will be ignored.

on_start()
When Crawler starts(before `start_requests()`), this method will be called.

class acrawler.handlers.**ItemToMongo**(*family=None*, *func_before=None*, *func_after=None*, *func_start=None*, *func_close=None*)

family = 'Item'
Family of this handler.

address = 'mongodb://localhost:27017'
a full mongodb URI, in addition to a simple hostname

db_name = ''
name of targeted database

col_name = ''
name of targeted collection

handle_after(*item*)
Then function called after the execution of the task.

on_close()
When Crawler closes, this method will be called.

on_start()
When Crawler starts(before `start_requests()`), this method will be called.

2.3.7 Setting/Config

There are default settings for aCrawler.

You can providing settings by writing a new *setting.py* in your working directory or writing in `Crawler`'s attributes.

acrawler.setting.**DOWNLOAD_DELAY = 0**
Every Request worker will delay some seconds before sending a new Request

acrawler.setting.**DOWNLOAD_DELAY_SPECIAL_HOST = {}**
Every Request worker for specific host will delay some seconds before sending a new Request

acrawler.setting.**LOG_LEVEL = 'INFO'**
Default log level.

acrawler.setting.**LOG_TO_FILE = None**
redirect log to a filepath

acrawler.setting.**LOG_TIME_DELTA = 60**
disabled

Type how many seconds to log a new crawling statistics. 0

`acrawler.setting.`**`STATUS_ALLOWED = None`**
A list of intergers representing status codes other than 200.

`acrawler.setting.`**`MAX_TRIES = 3`**
A task will try to execute for *max_tries* times before a complete fail.

`acrawler.setting.`**`MAX_REQUESTS = 4`**
A crawler will obtain *MAX_REQUESTS* request concurrently.

`acrawler.setting.`**`MAX_REQUESTS_PER_HOST = 0`**
Limit simultaneous connections to the same host.

`acrawler.setting.`**`MAX_REQUESTS_SPECIAL_HOST = {}`**
Limit simultaneous connections with a host-limit dictionary.

`acrawler.setting.`**`REDIS_ENABLE = False`**
Set to True if you want distributed crawling support. If it is True, the crawler will obtain *crawler.redis* and lock itself always.

`acrawler.setting.`**`REDIS_START_KEY = None`**
And the crawler will try to get url from redis list *REDIS_START_KEY* if it is not None and send Request(also bind *crawler.parse* as its callback function)

`acrawler.setting.`**`REDIS_QUEUE_KEY = None`**

`acrawler.setting.`**`REDIS_DF_KEY = None`**

`acrawler.setting.`**`REDIS_ADDRESS = 'redis://localhost'`**

`acrawler.setting.`**`WEB_ENABLE = False`**
Set to True if you want web service support. If it is True, the crawler will lock itself always.

`acrawler.setting.`**`WEB_HOST = 'localhost'`**
Host for the web service.

`acrawler.setting.`**`WEB_PORT = 8079`**
Port for the web service.

`acrawler.setting.`**`LOCK_ALWAYS = False`**
Set to True if you don't want the crawler exits after finishing tasks.

`acrawler.setting.`**`PERSISTENT = False`**
Set to True if you want stop-resume support. If you enable distributed support, this conf will be ignored.

`acrawler.setting.`**`PERSISTENT_NAME = None`**
A name tag for file-storage of persistent support

2.3.8 Utils

This module provides utility functions that are used by aCrawler. Some are used for external consumption.

`acrawler.utils.`**`merge_dicts`**(*a*, *b*)
Merges b into a

`acrawler.utils.`**`check_import`**(*name*, *allow_import_error=False*)
Safely import module only if it's not imported

`acrawler.utils.`**`open_html`**(*html*, *path=None*)
A helper function to debug your response. Usually called with *open_html(response.text)*.

`acrawler.utils.`**`get_logger`**(*name='user'*)
Get a logger which has the same configuration as crawler's logger.

`acrawler.utils.`**`redis_push_start_urls`**(*key*, *url=None*, *address='redis://localhost'*)
When you are using redis_based distributed crawling, you can use this function to feed start_urls to redis.

`acrawler.utils.`**`sync_coroutine`**(*coro*, *loop=None*)
Run a coroutine in synchronized way.

`acrawler.utils.`**`redis_push_start_urls_coro`**(*key*, *url=None*, *address='redis://localhost'*)
Coroutine version of `redis_push_start_urls()`

2.4 Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S