Grab Documentation

Выпуск 0.6.22

Grigority Petukhov

16 August 2015

Оглавление

Предупреждение: Документация на русском языке устарела и может содержать ошибки. Пожалуйста, используйте английскую документацию для получения актуальной информации о библиотеке Grab.

Grab - библиотека для работы с сетевыми документами. Основные области использования Grab:

- извлечение данных с веб-сайтов (site scraping)
- работа с сетевыми API
- автоматизация работы с веб-сайтами, например, регистратор профилей на каком-либо сайте

Grab состоит из двух частей:

- Главный интерфейс Grab для создания сетевого запроса и работы с его результатом. Этот интерфейс удобно использовать в простых скриптах, где не нужна большая многопоточность, или непосредственно в python-консоли.
- Интерфейс Spider, позволяющий разрабатывать асинхронные парсеры. Этот интерфейс позволяет, во-первых, более строго описать логику парсера, во-вторых, разрабатывать парсеры с большим числом сетевых потоков.

Глава 1

Grab сайты

- Официальный сайт: http://grablib.org
- Репозиторий на github: http://github.com/lorien/grab
- Группа рассылки: http://groups.google.com/group/python-grab

Глава 2

Документация Grab

2.1 Введение в Grab

Для начала нужно проимпортировать нужные вещи:

```
from grab import Grab
```

Теперь создадим рабочий объект:

```
g = Grab()
```

Запросим главную страницу сайта livejournal:

```
g.go('http://livejournal.com')
```

И выведем содержимое тэга title:

```
print g.xpath_text('//title')
```

Если вы хотите отправить POST-запрос, это можно сделать так:

```
g.setup(post={'key1': 'value1})
g.go('http://...')
```

Посмотреть кукисы, заголовки, код ответы можно в объекте *response*:

```
g.go('http://...')
print g.response.cookies['sid']
print g.response.headers['Content-Type']
print g.response.code
```

По-умолчанию, Grab сам обрабатывает кукисы. Например, если вы залогинитесь на какой-либо сайт, сессия будет поддерживаться автоматически.

С помощью Grab удобно обрабатывать формы:

```
g.go('some log-in page')
g.set_input('user', 'foo')
g.set_input('password', 'bar')
g.submit()
```

Вот так можно найти информацию в теле ответа по XPATH:

```
print g.xpath('//div[@id="error"]').text_content()
```

А так можно пробежаться по элементам:

```
for elem in g.xpath_list('//h3'):
    print elem.text
```

Об этих и многих других вещах читайте в *Документация Grab*

2.2 Установка библиотеки Grab

2.2.1 Установка под Linux

Установите зависимости любым удобным для вас способом. Вы можете воспользоваться пакетным менеджером либо утилитами *easy_install* или *pip*:

```
pip install pycurl lxml
```

Если у вас есть проблемы при установке lxml, возможно, вам нужно установить дополнительные пакеты. Пример для Debian/Ubuntu систем:

```
sudo apt-get install libxml2-dev libxslt-dev
```

Далее установите Grab:

```
pip install grab
```

2.2.2 Установка под Windows

Скачайте и установите lxml библиотеку с сайта http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml:

Скачайте и установите pycurl библиотеку с сайта http://www.lfd.uci.edu/~gohlke/pythonlibs/#pycurl:

Скачайте и установите grab с http://pypi.python.org/pypi/grab:

- качаем tar.gz архив
- распаковываем
- запускаем команду python.exe setup.py install

Если у вас python 2.7.6, то команда *python setup.py install* выполнится с ошибкой из-за бага в версии python 27.6. Вам нужно удалить Python версии 2.7.6 и установить версию 2.7.5

2.3 Настройка Grab-объекта

2.3.1 Способы задания настроек

Вы можете изменить свойства Grab объекта различными путями.

Во-первых, вы можете передать настройки через конструктор:

```
g = Grab(log_file='...', url='...')
```

Далее вы можете использовать метод setup:

```
g = Grab()
g.setup(log_file='...', url='...')
```

Самое позднее, где вы можете передать настройки, в методах, которые инициализируют сетевой запрос:

```
g.request(log_file='...')
```

или:

```
g.go('http://...', log_file='...')
```

Разница между методами *go* и *request* в том, что метод *go* требует обязательным первым параметром сетевой адрес, который в других случаях передаётся с помощью настройки *url*. Я часто использую метод *go* т.к. это придаёт выразительности программе.

Полный список настроек вы можете посмотреть в документе *Полный список настроек*

2.3.2 Клонирование

Если вам нужно создать ещё один Grab объект со свойствами существующего объекта, вы можете использовать метод `clone()`:

```
g2 = g.clone()
```

Клонирование сохраняет кукисы, что позволяет например, залогиниться и бродить по сайту с помщью нескольких Grab объектов. Также клонирование полезно, когда нужно запросить картинку капчи через отдельный объект.

Также существует метод `adopt()`, который позволяет привести состояние Grab-объекта к состоянию Grab-объекта, переданного аргументом методу *adopt*.

2.4 Отладка запросов

2.4.1 Использование logging-системы

Самый простой способ увидеть информацию об отсылаемых запросах - это включить вывод logging-сообщений с уровнем DEBUG:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Конечно это будет выводить все logging-сообщения, не только те, что сгенерировала библиотека Grab. Если вам нужны только её сообщения, то настройте вывод сообщенний только от логгера “grab”:

```
import logging
logger = logging.getLogger('grab')
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

Также вы можете воспользоваться функцией *default_logging* которая настраивает logging-систему на вывод всех сообщений Grab в файл:

```
from grab.tools.logs import default_logging
default_logging()
```

После вызова этой функции вы можете через отдельную консоль наблюдать за активностью Grab с помощью команды *tail -f /tmp/grab.log*.

На каждый сетевой запрос Grab генерирует logging-сообщение следующего вида:

```
[5864] GET http://www.kino-govno.com/movies/rusichi via 188.120.244.68:8080 proxy of type http with authorization
```

В начале мы видим номер запроса, далее типа запроса, затем адрес документа и в конце указана информация об используемом прокси сервере (если он используется). Если запрос был сделан не из главного thread-потока, то будет казано также имя thread-потока.

Можно включить вывод дополнительных данных о POST-запросах с помощью опции *debug_post*. Тогда на каждый POST-запрос будет выводиться его содержимое:

```
[01] POST http://yandex.ru
POST request:
foo                       : bar
name                      : Ivan
```

2.4.2 Нумерация запросов

Каждый сетевой запрос осуществлённый с помощью Grab имеет свой номер. Информация о запросе хранится на уровне модуля, так что запросы различных Grab-объектов имеют тем не менее общий счётчик. Общая нумерация сохраняется даже в случае использования тредов. Номер запроса очень удобно использовать вкупе с опцией *log_dir*, которая сохраняет содержимое сетевых ответов в файлы. Читайте об этом в следующем разделе.

2.4.3 Сохранение запросов и ответов в файлы

Обратите внимание на номер запроса, это важная составляющая системы отладки. Если мы включим сохранение содержимого запросов и ответов в файлы, по номеру запроса мы сможем найти нужный файл.

Сперва рассмотрим самую простую опцию *log_file*:

```
g.setup(log_file='/tmp/log.html')
```

Она включает сохранение содержимого последнего ответа в файл. Каждый новый ответ будет перезатирать содержимое старого ответа. Если вам нужно проанализировать содержимое нескольких запросов, то вам понадобится опция *log_dir*:

```
g.setup(log_dir='/tmp/some_dir')
```

После включения опции *log_dir* Grab начинает сохранять в указанную директорию содержимое запросов и ответов. Выше я писал о номере запроса, так вот имя файла будет содержать этот номер. Например, для запроса с номером 28 будут созданы два файла: 01.log и 02.html В log файле будут сохранены HTTP-заголовки как запроса, так и соответствующего ответа. В html файле будет сохранено содержимое ответа (без заголовков).

Счётчик запросов работает на уровне модуля. Это значит, что если вы создадите несколько Grab-объектов, то у них будет общий счётчик запросов и номера запросов не будут пересекаться. Это помогает отлаживать мультитредовые программы или программы, где создаётся несколько Grab объектов в разное время.

2.5 Полный список настроек

О том как изменять настройки, читайте в *Настройка Grab-объекта*.

2.5.1 Настройки

url

Сетевой адрес запрашиваемого документа. Можно использовать относительный адрес, в таком случае полный адрес будет получен путём соединения с полным адресом предыдущего сетевого запроса. Grab ожидает адрес в корректном формате. Это ваша обязанность - преобразовать все нестдартные символы в escape-последовательности (RFC 2396).

Type string

Default None

timeout

Максимальное время, отведённое на получение документа.

Type int

Default 15

connect_timeout

Максимальное время, отведённое на ожидание начала получения данных от сервера.

Type int

Default 10

user_agent

Содержимое HTTP-заголовка *User-Agent*. По-умолчанию, случайный выбор из множества реальных *User-Agent* значений, заложенных в Grab.

Type string

Default см. выше

user_agent_file

Путь к текстовому файлу с *User-Agent* строками. При указании этой опции, будет выбран случайный вариант из указанного файла.

Type string

Default None

method

Выбор метода HTTP-запроса. По-умолчанию, используется *GET* метод. Если заданы непустые опции *post* или *multipart_post*, то используется *POST* метод. Возможные варианты: *GET*, *POST*, *PUT*, *DELETE*.

Type string

Default "GET"

post

Данные для отправки запроса методом *POST*. Значением опции может быть словарь или последовательность пар значений или просто строка. В случае словаря или последовательности, каждое значение обрабатывается по следующему алгоритму: * объекты класса *UploadFile* преобразовываются во внутреннее представление библиотеки pycurl * unicode-строки преобразовываются в байтовые строки * *None*-значения преобразовываются в пустые строки

Если значением *post* опции является строка, то она передаётся в сетевой запрос без изменений.

Type sequence or dict or string

Default None

multipart_post

Данные для отправки запроса методом *Post*. Значением опции может быть словарь или последовательность пар значений. Данные запроса будут отформатированы в соотвествии с методом "multipart/form-data".

Type sequence or dict

Default None

headers

Дополнительные HTTP-заголовки. Значение этой опции будут склеено с заголовками, которые Grab отправляет по-умолчанию. Смотрите подробности в *Изменение HTTP-заголовков*.

Type dict

Default None

reuse_cookies

Если *True*, то кукисы из ответа сервера будут запомнены и отосланы в последующем запросе на сервер. Если *False* то кукисы из ответа сервера запоминаться не будут.

Type bool

Default True

cookies

Кукисы для отправки на сервер. Если включена также опция *reuse_ cookies*, то кукисы из опции *cookies* будут склеены с кукисами, запомненными из ответов сервера.

Type dict

Default None

cookiefile

Перед каждым запросом Grab будет считывать кукисы из этого файла и объединять с теми, что он уже помнит. После каждого запроса, Grab будет сохранять все кукисы в указанный файл.

Формат данных в файле: JSON-сериализованный словарь.

referer

Указание *Referer* заголовка. По-умолчанию, Grab сам формирует этот заголовок из адреса предыдущего запроса.

Type string

Default см. выше

reuse_referer

Если *True*, то использовать адрес предыдущего запроса для формирования заголовка *Refeer*.

Type bool

Default True

proxy

Адрес прокси-сервера в формате “server:port”.

Type string

Default None

proxy_userpwd

Данные авторизации прокси-сервера в формате “username:password”.

Type string

Default None

proxy_type

Тип прокси-сервера. Возможные значения: “http”, “socks4” и “socks5”.

Type string

Default None

encoding

Метод сжатия трафика. По-умолчанию, значение этой опции равно “gzip”. С некоторыми серверами возможны проблемы в работе pycurl, когда gzip включен. В случае проблем передайте в качестве значения опции пустую строку, чтобы выключить сжатие.

Type string

Default “gzip”

charset

Указание кодировки документа. По-умолчанию, кодировка определяется автоматически. Если определение кодировки проходит неправильно, вы можете явно указать нужную кодировку. Значение кодировки будет использовано для приведения содержимого документ в unicode-вид, а также для кодирования строковых не-ascii значений в *POST* данных.

Type string

Default None

log_file

Файл для сохранения полученного с сервера документа. Каждый новый запрос будет перезатить сохранённый ранее документ.

Type string

Default None

log_dir

Директория для сохранения ответов сервера. Каждый ответ сохраняется в двух файлах: * XX.log содержит HTTP-заголовки запроса и ответа * XX.html содержите тело ответа XX - это номер запроса. Смотрите подробности в *Отладка запросов*.

Type string

Default None

follow_refresh

Автоматическая обработка тэга <meta http-equiv=”refresh”>.

Type bool

Default False

follow_location

Автоматическая обработка редиректов в ответах со статусом 301 и 302.

Type bool

Default True

nobody

Игнорирование тела ответа сервера. Если опция включена, то соединение сервером будет разорвано после получения всех HTTP-заголовков ответа. Эта опция действует для любого метода: GET, POST и т.д.

Type bool

Default False

body_maxsize

Ограничение на количество принимаемых данных от сервера.

Type int

Default None

debug_post

Вывод через logging-систему содержимого POST-запросов.

Type bool

Default False

hammer_mode

Режим повторных запросов. Смотрите подробности в *Режим повторных запросов*.

Type bool

Default False

hammer_timeouts

Type list

Default ((2, 5), (5, 10), (10, 20), (15, 30))

Настройка таймаутов для режима повторных запросов.

userpwd

Имя пользователя и пароль для прохождения http-авторизации. Значение опции - это строка вида "username:password"

Type string

Default None

lowercased_tree

Приведение HTML-код документа к нижнему регистру перед построением DOM-дерева. Эта опция не влияет на содержимое *response.body*.

type bool

Default False

strip_null_bytes

Удаление нулевых байтов из HTML-кода документа перед построением DOM-дерева. Эта опция не влияет на содержимое *response.body*. Если в теле документа встретится нулевой байт, то библиотека LXML построит DOM-дерево только по фрагменту, следующему до первого нулевого байта.

Type bool

Default True

strip_xml_declaration

Удаление XML declaration из тела документа перед тем, как строить его unicode-представление. Я забыл зачем это нужно :) Попозже допишу помощь.

Type bool

Default True

2.6 Настройка HTTP-заголовков

2.6.1 Изменение HTTP-заголовков

Для управления отсылаемыми HTTP-заголовками используйте опцию *headers*, её значением должен быть словарь. По-умолчанию, Grab сам настраивает несколько HTTP-заголовков: Accept, Accept-Language, Accept-Charset, Keep-Alive и User-Agent. Их вы также можете переопределить опцией `headers`.

2.6.2 Настройка User-Agent заголовка

Для изменения *User-Agent* заголовка вы можете использовать как опцию *headers*, так и отдельную опцию *user_agent*. По-умолчанию, Grab генерирует значение для *User-Agent* заголовка на основе случайного выбора из множества значений *User-Agent* реальных браузеров. Вы также можете передать своё множество значений *User-Agent* с помощью опции *user_agent_file*, значением которой должен быть путь к текстовому файлу с *User-Agent* строками.

2.6.3 Настройка Referer заголовка

Для изменения *Referer* заголовка вы можете использовать как опцию *headers*, так и отдельную опцию *referer*. Для того, чтобы для заголовка *Referer* использовался адрес прыдыдущего запрошенного документа, включите опцию *reuse_referer*. Кстати, по-умолчанию, она и так включена.

2.7 Методы HTTP-запросов

2.7.1 Выбор метода

По-умолчанию, создаётся GET-запрос. Если вы указываете POST-данные, то тип запроса автоматически изменяется на POST:

```
g.setup(post={'user': 'root'})
g.request() # будет сгенерирован POST-запрос
```

Если вам нужен более экзотический типа запроса, вы можете указать его опцией *method*:

```
g.setup(method='PUT')
```

2.7.2 POST-запрос

Рассмотрим более подробно создание POST-запросов. По-умолчанию, когда вы задаёте *post* опцию, тип запроса меняется на POST, а *Content-Type* становится равен *application/x-www-form-urlencoded*. Опция *post* принимает данные в различных форматах. Если вы передаёте *dict* или список пар значений, то данные будут преобразованы в "key1=value1&key2=value2..." строку. Если же вы передаёте строку, то она будет отпралена в неизменном виде:

```
g.setup(post={'user': 'root', 'pwd': '123'})
g.setup(post=[('user', 'root'), ('pwd', '123')])
g.setup(post='user=root&pwd=123')
```

Чтобы отправить POST запрос с *Content-Type* равным *multipart/form-data*, используйте опцию *multipart_post* вместо *post*.

2.7.3 Отправка файлов

Чтобы отправить файл используйте специальный класс `UploadFile`, а также опцию *multipart_post*:

```
g.setup(multipart_post={'foo': bar', 'image': UploadFile('/tmp/image.gif')})
```

2.8 Прочие возможности

2.8.1 Ограничение тела ответа

Опцией *nobody* вы можете запретить принимать тело ответа. Соединение будет разорвано сразу после того, как будут получены все http-заголовки ответа.

Опцией *body_maxsize* вы можете управлять максимальным количеством информации, получаемой с сервера. В случае превышения указанного размера, соединение с сервером разрывается, никаких исключений не генерируется, а данные, что были получены, доступны для работы.

2.8.2 Сжатие ответа

С помощью опции *encoding* вы можете управлять сжатием ответа сервера. По-умолчанию, значение опции равно "gzip", что означает посылку заголовка "Accept-Encoding: gzip" и автоматическое распаковывание ответа (если он таки пришёл в загзипованном виде).

2.8.3 HTTP-Авторизация

С помощью опции *userpwd* можно передать имя пользователя и пароль для прохождения http-авторизации. Значение опции - это строка вида “username:password”

2.8.4 Работа с pycurl-дескриптором

Если вам нужно какая-либо возможность *pycurl*, интерфейс к которой отсутствует в Grab, вы можете работать с pycurl-дескриптором напрямую. Пример:

```
from grab import Grab
import pycurl

g = Grab()
g.curl.setopt(pycurl.RANGE, '100-200')
g.go('http://some/url')
```

2.8.5 301 и 302 редиректы

По-умолчанию, ответы со статусами 301 и 302 обрабатываются автоматически т.е. происходит переход по адресу, указанному в “Location:” заголовке ответа сервера:

```
HTTP/1.1 301 Moved Permanently
Content-Type: text/html
Content-Length: 174
Location: http://www.example.org/
```

Запретить автоматический переход можно опцией *follow_location*.

2.8.6 Meta Refresh редиректы

Один из способов перенаправить посетителя страницы на другой адрес - использование мета-тэга:

```
<meta http-equiv="Refresh" content="0; url=http://some/url" />
```

Grab может автоматически обрабатывать такой редирект. По-умолчанию, это поведение отключено. Включить его можно опцией *follow_refresh*.

2.9 Кодировка документа

2.9.1 Для чего нужно знать кодировку

По-умолчанию, кодировка данных, полученных с сервера, определяется автоматически. Естественно, это имеет смысл только для текстовых данных. Grab использует кодировку документа, чтобы:

- построить DOM-дерево документа
- получить текст документа в unicode-виде.
- провести поиск unicode-строки в документе
- преобразовать unicode-данные в байтовую строку для отправки в запросе

Оригинальное содержимое документа доступно в аттрибуте *body* объекта *response*, unicode-представление документа можно получить методом *unicode_body()* объекта *response*:

```
>>> g.go('http://yandex.ru')
<grab.response.Response object at 0x11bea90>
>>> type(g.response.body)
<type 'str'>
>>> type(g.response.unicode_body())
<type 'unicode'>
```

2.9.2 Алгоритм определения кодировки

Алгоритм определения кодировки документа проверяет несколько источников, в следующем порядке:

- мета-тэг <meta content="Http-Equiv" >
- xml-декларация (в случае XML-документа)
- значение HTTP-заголовка "Content-Type:"

Если кодировку определить не удалось или было найдено некорректное имя кодировки, то по-умолчанию, используется кодировка UTF-8.

2.9.3 Опция задания кодировки

Вы можете принудительно задать кодировку документа (отлючив её автоматическое определение) опцией *charset*.

2.10 Работа с кукисами

2.10.1 Настройка кукисов

Для того, чтобы отправить в запросе кукисы, используйте опцию *cookies*. Для того, чтобы кукисы, полученные в ответе сервера, автоматически подставлялись в следующие запросы, используйте опцию *reuse_cookies*. По-умолчанию, она включена.

Если включены обе опции *cookies* и *reuse_cookies*, то запомненные кукисы будут объединяться с теми, что указаны в *cookies*.

2.10.2 Работа с файлом кукисов

Вы можете указать путь к файлу в опции *cookiefile*. Перед каждым запросом Grab будет считывать кукисы из этого файла и объединять с теми, что он уже помнит. После каждого запроса, Grab будет сохранять все кукисы в указанный файл. Эта опция полезна, если вам нужно сохранить сессию авторизованного пользователя между различными запусками программы. Формат данных в файле: JSON-сериализованный словарь.

Для того, чтобы выгрузить кукисы Grab-объекта в файл, используйте метод *dump_cookies*. Для загрузки кукисов из файла используйте *load_cookies*.

Если вам нужно очистить все запомненные кукисы, воспользуйтесь методом *clear_cookies*, конкретно, он обнулит опцию *cookies*. Однако помните, если вы задали опцию *cookiefile*, то для следующего запроса, кукисы повторно загрузятся из этого файла.

2.11 Обработка сетевых ошибок, таймауты

2.11.1 Сетевые ошибки

Если в результате сетевого запроса происходит ошибка, Grab генерирует *GrabNetworkError* исключение. Что такое сетевая ошибка? Сервер может вообще не устанавливать соединение по нашему запросу или он может вернуть HTTP-ответ с кодом отличным от 2** (см Статусы HTTP ответа). Любой запрос с кодом отличным от 2** или 404 считается ошибкой и генерируется *GrabNetworkError* исключение. Нужно ли считать 404 ошибкой - спорный вопрос. Мне удобнее обрабатывать этот случай как нормальный ответ сервера. Возможно, нужна опция, устанавливающая какие коды ответа считать неошибочными.

2.11.2 Таймауты

Вы можете настроить максимальное время ожидания начала передачи данных от сервера опцией *connect_timeout* и максимальное время на весь процесс отправки запроса-прёма данных опцией *timeout*.

В случае превышение заданного времени, будет сгенерировано *GrabTimeoutError* исключение.

2.11.3 Режим повторных запросов

В случае активации опции *hammer_mode* Grab переключается в режим повторной отсылки того же запроса в случае сетевой ошибки. Количество повторных запросов и их таймауты настраиваются опцией *hammer_timeouts*. Рассмотрим пример:

```
g.setup(hammer_mode=True, hammer_timeouts=((2, 5), (10, 15), (20, 30)))
g.go('http://some/url')
```

Это значит, что первый сетевой запрос будет произведён с настройками *connect_timeout=2, timeout=5*. В случае timeout-ошибки или получения ответа с кодом отличным от успешного (см. обработка сетевых ошибок), запрос будет произведён ещё раз, но на этот раз с настройками *connect_timeout=10, timeout=15*. Если повторный запрос окончится неудачей будет проивзденё последний третий запрос с натройками *connect_timeout=20, timeout=30*. Надеюсь, вы поняли принцип.

2.12 Работа с прокси-серверами

2.12.1 Настройка прокси-сервера

Для использования проки-сервера вам нужно задать две опции: *proxy* и *proxy_type*. Опция *proxy* принимает значения в виде строки *server:port*. Опция *proxy_type* допускает значения трёх типов: *http*, *socks4* и *socks5*. Пример:

```
g.setup(proxy='gate.somhost.com:444', proxy_type='http')
```

Если прокси-сервер требует авторизации, используйте опцию *proxy_userpwd*, которая принимает значение в виде строки *username:password*.

Обратите внимание, что в случае использования прокси-сервера, информация о нём будет отображаться в logging-сообщениях, соответствующих конкретному запросу.

2.12.2 Работа со списками прокси

Grab поддерживает работу со списком-прокси. Используйте метод `setup_proxylist()` для задания списка проксей:

```
g.setup_proxylist(proxy_file='/path/to/file.txt', proxy_type='http')
```

Следующими аргументами метода `setup_proxylist()` вы можете настроить работу со списком проксей:

proxy_file путь к файлу со списком прокси-серверов

proxy_type тип прокси-серверов. Возможные варианты: "http", "socks4", "socks5".

read_timeout Время, через которое файл с проксями, будет перечитан.

auto_init Один раз выбирает случайный прокси-сервер, который используется для всех дальнейших запросов.

auto_change Включает постоянную смену прокси-сервера для каждого запроса

server_list Вы можете передать непосредственно список-проксей в python-списке, вместо указания файла с прокси-серверами. Аргументы *proxy_file* и *server_list* нельзя использовать одновременно.

Строки в файле, передаваемом через *proxy_file*, или в списке, передаваемом через *server_list* могут быть в двух форматах:

- Простой формат "server:port"
- Сложный формат "server:port:username:password". Используйте его, если прокси-сервер требует авторизации.

2.13 Работа с ответом

2.13.1 Объект Reponse

Результатом обработки запроса является объект класса *Response*. Вы можете получить к нему доступ через аттрибут *response*:

```
g.go('http://mail.ru')
print g.response.headers['Content-Type']
```

Смотрите полный перечень аттрибутов и методов объекта *Response* в API справочнике *Response*. Самое важное, что вам нужно знать:

body оригинальное тело ответа

unicode_body() тело ответа, приведённое к unicode-представлению

code HTTP-статус ответа

headers HTTP-заголовки ответа

charset кодировка документа

cokies кукисы ответа

url URL документа. В случае автоматической обработки редиректов, этот URL может отличаться от запрашиваемого.

2.14 Технические детали устройства Grab

2.14.1 Используемые библиотеки

По сути, Grab - это удобный интерфейс к двум библиотекам: pycurl и lxml.

Pycurl предоставляет возможность настраивать и осуществлять синхронные и асинхронные сетевые запросы.

Lxml даёт удобный способ работы с HTML-документами. В первую очередь это построение DOM-дерева, выборка элементов через XPATH и автоматический разбор форм.

Кроме этих двух библиотек в Grab использует множество стандартных модулей языка Python.

Для полноценной работы с Grab вам практически не нужно знать API библиотеки pycurl, но вот как устроен lxml знать вам нужно обязательно. Спасибо разработчикам lxml - они создали прекрасную документацию: http://lxml.de Рекомендую начать с The lxml.etree Tutorial

2.14.2 Структура расширений

Изначально исходники Grab представляли из себя один файл. Далее файл становился больше и больше и, наконец, стало ясно, что нужно разбивать функциональность по модулям. Grab спроектирован как базовый класс, наследующий свойства множества модулей, называемых расширениями. Работа с сетевыми функциями вынесена в модули, называемые транспортами. Основной транспорт Grab это pycurl. Также ведутся работы по прикручиванию urllib и Selenium. Основная идея транспортов в том, что можно заменить один транспорт другим и программа останется рабочей.

2.14.3 Поддержка python 3

Grab тестируется под python 2.7. Насчёт работоспособности в py3k ничего не могу сказать пока.

2.15 Работа с формами

2.15.1 Автоматическая обработка форм

Если запрошенный документ содержит форму, вы можете заполнить её поля с помощью метода `set_input()` и отослать методом `submit()`. Значения для полей, которые вы не заполнили явно, будут вычислены автоматически. В первую, очередь это касается hidden-полей, но для select, checkbox, radio полей Grab также попытается подставить какое-либо значение. Все методы работают с формой, которая выбрана по-умолчанию. Если в документе несколько форм, то будет выбрана та форма, в которой больше всего полей. В методах *set_ input_ ** автоматический выбор работает несколько по иному, выбирается та форма, которая содержит указанное поле.

Простейший пример поиска на яндексе:

```
>>> g = Grab()
>>> g.go('http://ya.ru')
<grab.response.Response object at 0x1a51ad0>
>>> g.set_input('text', 'grab python')
>>> g.submit()
<grab.response.Response object at 0x1ad41d0>
>>> g.xpath('//li[@class="b-serp-item"]//a/@href')
'http://packages.python.org/grab/'
```

Форма, выбранная по-умолчанию, доступна через аттрибут *form*. Это объект *lxml.etree.Element* с дополнительными свойствами, подробнее вы можете прочитать в lxml мануале.

Если по какой-либо причине вы не можете использовать метод *set_input* для задания значения элемента по его имени, попробуйте методы: *set_input_by_id* или универсальный *set_input_by_xpath*.

Смотрите полный список доступных методов в extensions_form.

2.15.2 Отправка формы

Как было уже сказано выше, при отправке формы методом *submit* значения полей, которые в не задали явно, вычисляются автоматически. Это избавляет от рутинной обработки hidden-полей. Также автоматически вычисляется нужные метод HTTP-запроса (POST или GET), нужная кодировка данных (*www/url-encoded* или *multipart/form-data*), полный адрес формы (аттрибут *action*). У метода *submit* есть несколько полезных аргументов:

- *submit_name* - позволяет "нажать" нужный вам submit-элементов (полезно, если их несколько)
- *make_request* - передав False в этом аргументе вы отлючите автоматическую посылку данных формы. Все параметры запроса будут подготовлены в точности, как описано выше, но запрос не будет сгенерирован. Вы можете подправить его параметры по своему вкусу и уже затем отослать его на сервер.
- *url* - переопределение адреса, куда будет направлен запрос с данными формы
- *extra_post* - словарь с данными полей, которые переопределят автоматически вычисленные значения.

2.15.3 Отправка файлов

Для отправки файлов используйте специальный класс `UploadFile`. Его конструктор принимает единственный элемент - путь к файлу. Вы можете использовать объект класса *UploadFile* в методе `submit()`:

```
g.set_input('file', UploadFile('/path/to/file'))
g.submit()
```

2.16 Работа с DOM-деревом

2.16.1 Интерфейс к LXML библиотеке

Первое, что вам нужно усвоить, это то, что Grab предоставляет всего лишь удобный интерфейс к функциям библиотеки lxml. Крайне желательно, знать и понимать API библиотеки lxml. Grab предоставляет множество функций поиска данных в документе. Большинство этих функций представляют из себя xpath-запрос к DOM-дереву и последующую его обработку.

Далее описаны основные принципы использования lxml-расширения. Полный список методов (и их описание) вы можете посмотреть в API справочнике extensions_lxml.

2.16.2 DOM-дерево

DOM-дерево доступно через аттрибут `tree <LXMLExtension.tree()`:

```
>>> g.go('http://vk.com')
<grab.response.Response object at 0x1c9ae10>
>>> g.tree
<Element html at 1c96940>
>>> print g.tree.xpath('//title/text()')[0]
Welcome!
```

Вычисление DOM-дерева требует значительных ресурсов процессора, поэтому оно не вычисляется сразу после получения тела документа, а лишь только при первом вызове какого-либо xpath/css метода или обращении к аттрибуту `tree <LXMLExtension.tree()`. DOM-дерево вычисляется один раз и затем кэшируется.

2.16.3 XPATH-методы

Самый часто используемый метод, это `xpath()`. В качестве аргумента он принимает xpath-выражение и возвращает найденный узел DOM-дерева. Пожалуйста, не путайте `xpath()` метод объекта Grab и *xpath* метод *lxml.html.etree.Element* объекта. Последний возвращает список элементов, в то время как *xpath* метод Grab-объекта возвращает первый найденный элемент. Если вам нужен список объектов, используйте `xpath_list()` метод. Приведу наглядный пример:

```
>>> g.go('http://google.com')
<grab.response.Response object at 0x1ae73d0>
>>> g.xpath_list('//*[@type="submit"]')
[<InputElement 1a35e88 name='btnG' type='submit'>, <InputElement 1c96530 name='btnI' type='submit'>]
>>> g.xpath('//*[@type="submit"]')
<InputElement 1a35e88 name='btnG' type='submit'>
>>> from lxml.html import fromstring
>>> fromstring(g.response.body).xpath('//*[@type="submit"]')
[<InputElement 1c966d0 name='btnG' type='submit'>, <InputElement 1c96598 name='btnI' type='submit'>]
```

Методом *xpath_text* вы можете извлечь текстовое содержимое из найденного DOM-элемента. Метод *xpath_number* извлекает в начале текстовое содержимое, затем ищет там число:

```
>>> g.go('http://rambler.ru')
<grab.response.Response object at 0x1c9a650>
>>> print g.xpath_text('//td[@class="Spine"]//nobr')
1996-2012
>>> print g.xpath_number('//td[@class="Spine"]//nobr')
1996
```

2.16.4 CSS-методы

Благодаря модулю cssselect, можно искать элементы в DOM-дереве с помощью CSS-выражений. Поддерживаются основные CSS2-селекторы, не все. Список и название методов для работы с CSS аналогичен списку методов для работы с xpath.

```
>>> g.go('http://rambler.ru')
<grab.response.Response object at 0x1c9a650>
>>> print g.css_text('td.Spine nobr')
1996-2012
>>> print g.css_number('td.Spine nobr')
1996
```

2.16.5 Обработка исключений

Если xpath/css метод не нашёл данных, то генерируется исключение *DataNotFound*. Класс этого исключения унаследован от `IndexError`, так что можно просто ловить IndexError на заморачиваясь на импорт *DataNotFound* исключения:

```
>>> try:
...     g.xpath('//foobar')
... except IndexError:
...     print 'not found'
...
not found
```

Все xpath/css методы понимают аргумент *default*, если вы зададите его, то в случае, когда данные не были найден, вместо генерации исключения, xpath/css метод вернёт указанное значение. В качестве значения вы можете передавать даже *None*:

```
>>> print g.xpath('//foobar')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.6/dist-packages/grab/ext/lxml.py", line 134, in xpath
    raise DataNotFound('Xpath not found: %s' % path)
grab.error.DataNotFound: Xpath not found: //foobar
>>> print g.xpath('//foobar', default=None)
None
>>> print g.xpath('//foobar', default='spam')
spam
```

2.17 Поиск в тексте документа

2.17.1 Поиск строк

Методом `search()` можно установить наличие или отсутствие текстового фрагмента в исходном коде документа. Обратите внимание, что поиск проводится именно в HTML-коде, а не в объединение текстового содержимого всех элементов документа.

Может оказаться полезным метод `assert_substring()`, единственное назначение которого выбросить *DataNotFound* исключение, если искомая строка не найдена. Для проверки существования хотя бы одной строки из множества, можно использовать метод `assert_substrings()`.

По-умолчанию, описанные методы ожидают аргумент в unicode-виде и проводят поиск в `grab.response.Response.unicode_body()`. Если вы хотите искать байтовую строку в `grab.response.Response.body`, передайте дополнительный аргумент *byte=True* в поисковый метод:

```
>>> g.go('http://forum.omsk.com')
<grab.response.Response object at 0x1910f10>
>>> g.search(u'<title>')
True
>>> g.search(u'Омск')
True
>>> g.search('Омск')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.6/dist-packages/grab/ext/text.py", line 36, in search
    raise GrabMisuseError('The anchor should be byte string in non-byte mode')
```

```
grab.error.GrabMisuseError: The anchor should be byte string in non-byte mode
>>> g.search(u'Омск'.encode('cp1251'), byte=True)
True
```

2.17.2 Поиск регулярных выражений

Метод `rex()` позволяет искать регулярное выражение в исходном коде документа. В качестве аргумента вы можете передать либо уже скомпилированный объект регулярного выражения либо текстовую строку, которая будет скомпилирована в объект регулярного выражения автоматически.

Если вам нужно извлечь из исхоного кода некоторый фрагмент, может оказаться полезным метод `rex_text()`, который ищет регулярное выражение и возвращает первую группу из него:

```
>>> g.go('http://linode.com')
<grab.response.Response object at 0x20a8150>
>>> g.rex(re.compile('<title>[^>]+</title>')).group(0)
u'<title>Linode - Xen VPS Hosting</title>'
>>> g.rex('<title>[^>]+</title>').group(0)
u'<title>Linode - Xen VPS Hosting</title>'
>>> g.rex_text('<title>([^>]+)</title>')
u'Linode - Xen VPS Hosting'
```

Метод `assert_rex()` по принципу действия аналогичен методу `assert_substring()`.

2.18 Другие расширения

2.18.1 PyQuery расширение

Через аттрибут `pyquery()` вам доступен *PyQuery* объект, связанный с содержимым документа. PyQuery - это наслойка поверх lxml API, позволяющая выбирать элементы с помощью jQuery-селекторов:

```
>>> g = Grab()
>>> g.go('http://yandex.ru')
<grab.response.Response object at 0x1159b10>
>>> print g.pyquery('ol.b-news__news li:eq(0)')[0].text_content()
1. Дальневосточники активно голосуют на выборах президента России
```

2.18.2 BeautifulSoup расширение

Через аттрибут `soup()` вы можете обращаться к DOM-дереву документа, через API BeautifulSoup. Обратите внимание, что это расширение не доступно по-умолчаню. Если оно вам нужно, создайте свой класс, унаследованный от классов *Grab* и `grab.ext.soup.BeautifulSoupExtension`:

```
>>> from grab.ext.soup import BeautifulSoupExtension
>>> class MyGrab(Grab, BeautifulSoupExtension):
...     pass
...
>>> g = MyGrab()
>>> g.go('http://yandex.ru')
<grab.response.Response object at 0x13ea390>
>>> g.soup.title
<title>Яндекс</title>
```

2.19 Сетевые транспорты

2.19.1 Что такое транспорт

Транспортом в Grab называется расширение, которое осуществляет сетевой запрос и обработку полученного ответа. Grab предоставляет универсальный интерфейс для настройки параметра запроса и для обработки результатов запроса, но сама функциональность по отсылке и приёму данных по сетевому каналу скрыта внутри библиотеки и может осуществляться различными способами.

2.19.2 Транспорт pycurl

Сайт библиотеки: http://pycurl.sourceforge.net

По-умолчанию, Grab работает с библиотекой pycurl, это python-интерфейс к библиотеке cURL. Библиотека curl поддерживает огромное количество возможностей по создания, передаче и получению HTTP-запросов. Библиотека curl не ограничена работой с HTTP-протоколом, но Grab в основном ориентирован на HTTP-протокол. Схема использования pycurl в Grab довольно проста: в аттрибуте *pycurl* хранится pycurl объект, который настраивается в соотвествии с заданными опциями и затем используется для передачи данных на сервер и приёма ответных данных. Интерфейс библиотеки pycurl не слишком удобный, собственно, это и было изначальной причиной написать Grab - получить человечкий интерфейс к возможностям curl.

2.19.3 Транспорт urllib

Сайт библиотеки: http://docs.python.org/library/urllib.html

Преимуществом этого транспорта является то, что библиотека urllib является стандартной библиотекой языка python. Разработка данного транспорта находится в зачаточном состоянии. На данный момент я попытался использовать urllib через прослойку Requests, возможно, следует отказаться от неё и соединить Grab и urllib напрямую.

2.19.4 Транспорт selenium

Сайт библиотеки: http://seleniumhq.org/

Преимуществом данного транспорта является то, что selenium позволяет работать с документами в режиме браузера, в том числе выполнять javascript скрипты. На данный момент транспорт seleinum находится в очень зачаточном состоянии. Пощупать можно так:

```
from grab import GrabSelenium

g = GrabSelenium()
g.go('http://ixbt.com')
print g.xpath_text('//title')
```

Естественно вам нужно установить предварительно selenium. Это можно сделать командой: *sudo pip install selenium*.

2.20 Полезные утилиты

В пакете grab.tools содержится множество различных вспомогательных утилит, которые оказываются полезными в разработке Grab и парсеров на основе Grab. Здесь приведён обзор наиболее важных утилит.

2.20.1 Пул заданий

Используя функцию `work.make_work()` вы можете организовать выполнение множества заданий в параллельных тредах, причём количеством тредов можно управлять:

```
def worker(url):
    g = Grab()
    g.go(url)
    return url, g.xpath_text('//title')

task_iterator = open('urls.txt')
for url, title make_work(worker, task_iterator, limit=5):
    print url, title
```

Обратите внимание, что вы можете передавать не только статический список заданий, но и итератор. Результат работы функции `work.make_work()` выполнен также в виде итератора. Если вы хотите использовать процессы, вместо тредов, вам нужна функция `pwork.make_work()`. Она аналогична вышерассмотренной, за тем исключением, что она порождает не треды (*threading.Thread*), но процессы (*multiprocessing.Process*)

2.20.2 Блокировка файла

Для того, чтобы гарантировать то, что в любой момент времени выполняется только один экземпляр вашего парсера, можно использовать функцию `lock.assert_lock()`. Её аргумент - путь до файла, который должен быть залочен. Если залочить файл не удаётся, функция генерирует исключение и программа прекращается. Естественно, в разных скриптах нужно лочить различные файлы.

2.20.3 Логирование Grab-активности в файл

Функция `logs.default_logging()` настраивает logging-систему так, чтобы все сообщения библиотеки Grab направлялись в файл, по-умолчанию, это "/tmp/grab.log". Удобно вызвать эту функцию в начале программы и наблюдать за активностью парсинга с помощью команды *tail -f /tmp/grab.log*, оставляя себе возможность выводить в консоль, где был запущен скрипт, более важные данные.

2.20.4 Фильтрация строк в файле

Функция `files.unique_file()` читает строки из файла, оставляет уникальные строки и записывает их обратно в файл. Функция `files.unique_host()` читайет список URL-строк из файла и оставляет только строки с уникальным hostname, далее записывает строки обратно в файл.

2.20.5 Обработка HTML

- `html.decode_entities()` - преобразовывает все &XXX; и &#XXX; последовательности в тексте в уникод.

- `html.strip_tags()` - вырезает все тэги из текста простым регекспом.
- `html.escape()` - преобразовывает ряд "небезопасных" HTML-символов в "&xxx;" последовательности.

2.20.6 Работа с LXML-элементами

- `lxml_tools.get_node_text()` - возвращает текстовое содержимое элемента и всех его подэлементов, за ислючением элементов script и style.
- `lxml_tools.find_node_number()` - возвращает первое найденное число в текстовом содержимом переданного элемента.

2.20.7 Работа с регулярными выражениями

Функция `rex.rex()` позволяет искать регулярное выражение. Вы можете передать ей как скомпилированный объект регулярного выражения, так и просто текст из которого будет построен объект регулярного выражения. Объекты регулярных выражений кэшируются, так что вам не нужно беспокоиться о том, что выражение будет перекомпилироваться. Если выражение не найдено, функция `rex.rex()` сгенерирует *grab.error.DataNotFound* исключение. Вы можете изменить это поведение, передав в аргументе *default* значение, которое нужно вернуть по-умолчанию:

```
>>> from grab.tools import rex
>>> import re
>>> rex.rex('*** foo +++', re.compile('\w+')).group(0)
'foo'
>>> rex.rex('*** foo +++', '\w+').group(0)
'foo'
>>> rex.rex('***', '\w+')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.6/dist-packages/grab/tools/rex.py", line 44, in rex
    raise DataNotFound('Could not find regexp: %s' % regexp)
grab.error.DataNotFound: Could not find regexp: <_sre.SRE_Pattern object at 0xb83c10>
>>> rex.rex('***', '\w+', default='default value')
'default value'
```

Функция `rex.rex_list()` вернёт список всех найденных регулярных выражений. Функция `rex.rex_text()` найдёт указанный текст и затем вырежет из него все тэги. Функция `rex.rex_text_list()` вернёт список всех найденных текстовых фрагментов с вырезанными тэгами.

2.20.8 Работа с текстом

- `text.find_number()` - поиск числа в строке
- `text.drop_space()` - удаление *всех* пробелов в строке
- `text.normalize_space()` - удаление начальных и конечных пробелов, приведение последовательности пробелов к одному пробелу.

2.20.9 Работа с http-заголовками

- `http.urlencode()` - сериализация словаря или списка пар в строку, которую можно отправить в GET или POST-запросе. В отличие от стандартного *urllib.urlencode* может обрабатывать unicode, None и `grab.upload.UploadFile` объекты.

Глава 3

Документация Grab:Spider

Асинхронный модуль для разработки сложных парсеров.

3.1 Что такое Spider

Модуль Spider это фреймворк, позволяющий описать парсер сайта как набор функций-обработчиков, каждый из которых отвечает за результаты обработки конкретного документа (сетевого запроса). Например, при парсинге форума у вас будут обработчики для главной страницы, страницы подфорума, страницы топика, страницы профиля участника. Изначально такая структура парсера была разработана в силу ограничений асинхронного режима, но впоследствии оказалось, что писать парсеры в таком структурированном виде (один запрос - одна функция) очень удобно.

Модуль Spider работает асинхронно. Это значит, что всегда есть только один рабочий поток программы. Для множественных запросов не создаются ни треды, ни процессы. Все созданные запросы обрабатываются библиотекой multicurl. Суть асинхронного подхода в том, что программа создаёт сетевые запросы и ждёт сигналы о готовности ответа на эти запроссы. Как только готов ответ, вызывается функция-обработчик, которую мы привязали к конкретному запросу. Асинхронный подход позволяет обрабатывать большее количество одновременных соединений, чем подход, связанный с созданием множества тредов или процессов, т.к. память занята всего одним процессом и процессору не нужно постоянно переключаться между различными процессами программы.

Есть один нюанс, который будет очень непривычен тем, кто привык работать в синхронном стиле. Асинхронный подход позволяет вызывать функции-обработчики при готовности сетевого ответа. Если алгоритм парсинга состоит из нескольких последовательных сетевых запросов, то нужно где-то хранить информацию о том, для чего мы создали сетевой запрос и что с ним делать. Spider позволяет достаточно удобно решать эту проблему.

Каждая функция-обработчк получает два аргумента. Первый аргумент - это объект Grab, в котором хранится информация о сетевом ответе. Вся прелесть модуля Spider в том, что он сохранил знакомый вам интерфейс для работы с синхронными запросами. Второй аргумент функции-обработчика это Task объект. Task объекты создаются в Spider для того, чтобы добавить в очередь сетевых запросов новое задание. С помощью Task объекта можно сохранять промежуточные данные между множественными запросами.

Рассмотрим пример простого парсера. Допустим, мы хотим зайти на сайт habrahabr.ru, считать заголовки последних новостей, далее для каждого заголовка найти картинку с помощью images.yandex.ru и сохранить полученные данные в файл:

```
# coding: utf-8
import urllib
import csv
```

```
import logging

from grab.spider import Spider, Task

class ExampleSpider(Spider):
    # Список страниц, с которых Spider начнёт работу
    # для каждого адреса в этом списке будет сгенерировано
    # задание с именем initial
    initial_urls = ['http://habrahabr.ru/']

    def prepare(self):
        # Подготовим файл для записи результатов
        # Функция prepare вызывается один раз перед началом
        # работы парсера
        self.result_file = csv.writer(open('result.txt', 'w'))
        # Этот счётчик будем использовать для нумерации
        # найденных картинок, чтобы создавать им простые имена файлов.
        self.result_counter = 0

    def task_initial(self, grab, task):
        print 'Habrahabr home page'

        # Это функция-обработчик для заданий с именем initial
        # т.е. для тех заданий, что были созданы для
        # адресов указанных в self.initial_urls

        # Как видите интерфейс работы с ответом такой же,
        # как и в обычном Grab
        for elem in grab.xpath_list('//h1[@class="title"]/a[@class="post_title"]'):
            # Для каждой ссылки-заголовка создадим новое задание
            # с именем habrapost
            # Обратите внимание, что мы создаём задания с помощью
            # вызова yield - это сделано исключительно ради красоты
            # По сути, это равносильно следующему коду:
            # self.add_task(Task('habrapost', url=...))
            yield Task('habrapost', url=elem.get('href'))

    def task_habrapost(self, grab, task):
        print 'Habrahabr topic: %s' % task.url

        # Эта функция, как вы уже догадываетесь,
        # получает результаты обработки запросов, которые
        # мы создали для кадого хабратопика, найденного на
        # главной странице хабры

        # Для начала сохраним адрес и заголовок топика в словарь
        post = {
            'url': task.url,
            'title': grab.xpath_text('//h1/span[@class="post_title"]'),
        }

        # Теперь создадим поисковый запрос картинок яндекса, обратите внимание,
        # что мы передаём объекту Task информацию о хабрапосте. Таким образом
        # в функции обработки поиска картинок мы будем знать, для какого именно
        # хабрапоста мы получили результат поиска картинки. Дело в том, что все
        # нестандартные аргументы конструктора Task просто запоминаются в созданном
        # объекте и доступны в дальнейшем как его атрибуты
        query = urllib.quote_plus(post['title'].encode('utf-8'))
```

```
        search_url = 'http://images.yandex.ru/yandsearch?text=%s&rpt=image' % query
        yield Task('image_search', url=search_url, post=post)

    def task_image_search(self, grab, task):
        print 'Images search result for %s' % task.post['title']

        # В этой функции мы получили результат обработки поиска картинок, но
        # это ещё не сама картинка! Это только список найденных картинок,
        # Теперь возьмём адрес первой картинки и создадим задание для её
        # скачивания. Не забудем передать информацию о хабрапосте, для которого
        # мы ищем картинку, эта информация хранится в `task.post`.
        image_url = grab.xpath_text('//div[@class="b-image"]/a/img/@src')
        yield Task('image', url=image_url, post=task.post)

    def task_image(self, grab, task):
        print 'Image downloaded for %s' % task.post['title']

        # Это последняя функция в нашем парсере.
        # Картинка получена, можно сохранить результат.
        path = 'images/%s.jpg' % self.result_counter
        grab.response.save(path)
        self.result_file.writerow([
            task.post['url'].encode('utf-8'),
            task.post['title'].encode('utf-8'),
            path
        ])
        # Не забудем увеличить счётчик ответов, чтобы
        # следующая картинка записалась в другой файл
        self.result_counter += 1


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    # Запустим парсер в многопоточном режиме - два потока
    # Можно больше, только вас яндекс забанит
    # Он вас и с двумя то потоками забанит, если много будете его беспокоить
    bot = ExampleSpider(thread_number=2)
    bot.run()
```

В примере рассмотрен простейший парсер и не затронуто очень много возможностей Spider. Читайте о них в подробной документации. Обратите внимание, что часть функций обработчиков отработают с ошибкой, например, потому что, яндекс ничего не найдёт по заданному запросу.

Обо этом и многом другом читайте в *Документация Grab:Spider*

3.2 Способы создания заданий

Spider это по сути набор функций-обработчиков сетевых запросов. Каждый обработчик в свою очередь может создать новые запросы или просто сохранить куда-либо данные. Каждый запрос описывается Task-объектом. Паук добавляет каждый новый запрос в очередь и выполняет его по мере освобождения сетевых ресурсов. Каждому Task-объекту присваиватя имя. Когда становится доступен результат сетевого запроса, то с помощью этого имени определяется имя функции-обработчика и вызывается эта функция.

Например, если мы создадим задание с именем “contact_page”, то мы должны будем объявить в нашем классе паука метод с именем “task_contact_page”:

```
    ...
    self.add_task(Task('contact_page', url='http://domain.com/contact.html'))
    ...

def task_contact_page(self, grab, task):
    ...
```

Имя функции-обработчика определяется так: берётся имя задания и добавляется префикс "**task_**".

Рассмотрим различные способы создания Task-заданий.

3.2.1 initial_urls

В атрибуте паука *initial_urls* Можно указать список адресов, с обработки которых паук должен начать свою работу:

```
class ExampleSpider(Spider):
    initial_urls = ['http://google.com/', 'http://yahoo.com/']
```

Для всех адресов, перечисленных в *initial_urls* будет создано задание с именем 'initial'. Это самый простой способ создания заданий, вы не можете управлять ничем кроме адресов запрашиваемых документов.

3.2.2 task_generator

Более сложный способ создания начальных заданий. Метод с именем *task_generator* должен являться python-генератором т.е. функцией выдающей множество значений с помощью инструкции yield. Spider будет обращаться к новым задания из *task_generator* каждый раз, когда его очередь будет опустошаться. Это позволяет не опасаться того, что вы создадите слишком много заданий. Выглядит это так: в начале работы паук извлекает некоторое количество заданий с помощью *task_generator* и помещает их в очередь, далее он выполняет запросы и следит за количеством заданий в очереди. Как только их становится слишком мало, паук обращется ещё раз к *task_generator* и добавляет новые задания.

К примеру, вы можете открыть файл с миллионом записей и последовательно читать строки из него, создавая всё новые и новые задания:

```
class ExampleSpider(Spider):
    def task_generator(self):
        for line in open('var/urls.txt'):
            yield Task('download', url=line.strip())
```

3.2.3 add_task

Независимо от того, каким способом вы создали новое задание, в очередь заданий оно попадёт с помощью метода *add_task*. В случае использования *intial_urls* или *task_generator* метод *add_task* будет вызван неявно, но вы, конечно, можете использовать его напрямую, чтобы добавить новое задание в любом месте выполнения программы. Это можно делать даже до начала работы паука. Например:

```
bot = ExampleSpider()
bot.add_task('google', url='http://google.com')
bot.run()
```

3.2.4 yield

Инструкцию yield для создания заданий вы можете использовать в двух местах, во-первых, в методе *task_ generator*, о чём уже писалось выше, во-вторых, в любой функции-обработчике результата. При вызове функции-обработчика Spider ловит все задания, которые она генерирует и складывает в очередь заданий. Создание заданий с помощью yield ничем не отличается от использования метода *add_ task*, разве что запись получается более короткая:

```
class ExampleSpider(Spider):
    initial_urls = ['http://google.com']

    def task_initial(self, grab, task):
        # Google page was fetched
        # Now let's download yahoo page
        yield Task('yahoo', url='yahoo.com')

    def task_yahoo(self, grab, task):
        pass
```

3.2.5 Резюме

Для задания начальных заданий используйте атрибут *initial_ urls*, если вам нужна более сложная логика создания начальных заданий, используйте метод *task_ generator*. Для создания заданий внутри функций-обработчиков используйте инструкцию yield. Использовать метод *add_ task* напрямую вам практически никога не понадобится.

Есть также ряд методов для типичных случаев генерации новых заданий: обработка пагинации, обработка списка ссылок. Смотрите модуль *grab.spider.pattern*.

3.3 Задания

Spider это по сути набор функций-обработчиков сетевых запросов. Каждый обработчик в свою очередь может создать новые запросы или просто сохранить куда-либо данные. Каждый запрос описывается Task-объектом. Паук добавляет каждый новый запрос в очередь и выполняет его по мере освобождения сетевых ресурсов. Каждому Task-объекту присваиватя имя. Когда становится доступен результат сетевого запроса, то с помощью этого имени определяется имя функции-обработчика и вызывается эта функция.

Например, если мы создадим задание с именем “contact_page”, то мы должны будем объявить в нашем классе паука метод с именем “task_contact_page”:

```
    ...
    self.add_task(Task('contact_page', url='http://domain.com/contact.html'))
    ...

def task_contact_page(self, grab, task):
    ...
```

Имя функции-обработчика определяется так: берётся имя задания и добавляется префикс “**task_**”.

3.3.1 Конструктор Task объекта

Конструктор Task-объекта принимает множество аргументов. Вы должны обязательно указать имя задания и адрес документа, либо настроенный Grab-объект. Далее приведены примеры кода, который

создаёт три одинаковых задания:

```
# Using `url` argument
t = Task('wikipedia', url 'http://wikipedia.org/')

# Using Grab intance
g = Grab()
g.setup(url='http://wikipedia.org/')
t = Task('wikipedia', grab=g)

# Using configured state of Grab instance
g = Grab()
g.setup(url='http://wikipedia.org/')
config = g.dump_config()
t = Task('wikipedia', grab_config=config)
```

Также в конструкторе задания можно задать следующие свойства, работу которых вы сможете понять из описания других частей архитектуры Spider:

priority приоритет задания, целое положительное число, чем меньше число, тем выше приоритет.

disable_cache не использовать кэш паука для этого запроса, сетевой ответ в кэш сохранён не будет

refresh_cache не использовать кэш паука, в случае удачного ответа обновить запись в кэше

valid_status обрабатывать обычным способом указанные статусы. По умолчанию обрабатываются все статусы 2xx, а также статус 404.

use_proxylist использовать заданный глобально для паука список прокси. По умолчанию опция включена.

За исключением вышеуказанных и ещё нескольких аргументов все аргументы просто сохраняются в Task-объект и доступны для дальнейшего использования. Таким образом Task-объект выступает как хранилище данных: можно запоминать данные и передавать их от запроса к запросу.

3.3.2 Task-объект как хранилище данных

В асинхронном окружении часто бывает нужным куда-то записать информацию о запросе, а затем "вспомнить" её, когда будет готов ответ на запрос. Как было сказано выше, все неспециальные аргументы конструктора Task-объекта, просто запоминаются в объекте и доступны в дальнейшем как его атрибуты. Для удобства Task-объект имеет метод *get*, который возвращает None (или указанное вами значение), если запрошенного атрибута в Task-объекте не нашлось. Рассмотрим примеры:

```
t = Task('bing', url='http://bing.com/', disable_cache=True, foo='bar')
t.foo # == "bar"
t.get('foo') # == "bar"
t.get('asdf') # == None
t.get('asdf', 'qwerty') # == "qwerty"
```

3.3.3 Клонирование Task-объекта

Иногда бывает удобно использовать существующий Task-объект для создания нового. Например, когда мы получили ответ на сетевой запрос и хотим сделать похожий запрос:

```
# TODO: придумать вменяемый пример
```

3.4 Очередь заданий

3.4.1 Приоритеты заданий

Все задания помещаются в очередь, откуда они ивзлекаются по мере освобождения сетевых ресурсов и выполняются. У каждого задания есть приоритет, обозначаемый положительным целым числом. Чем меньше это число, тем выше приоритет. Если приоритет не указан явно при создании задания, он назначается автоматически. Есть два алгоритма автоматического задания приоритетов:

random случайные приоритеты

const один и тот же приоритет для всех заданий.

По умолчанию используются случайные приоритеты. Способ выбора приоритетов задаётся аргументом *priority_mode* при создании Spider-объекта:

```
bot = SomeSpider(priority_mode='const')
```

3.4.2 Бэкенды хранилищ

Очередь заданий в Grab выделена в отдельный слой, это позволяет писать реализации очереди для различных систем хранения данных. Из коробки доступны две реализации: хранение заданий в памяти и хранение заданий в mongodb. По умолчанию задания хранятся в памяти. Если объёма вашей памяти не хватает, чтобы хранить все задания, то рекомендуется использовать очередь в mongodb:

```
bot = SomeSpider()
bot.setup_queue() # очередь в памяти
bot.setup_queue(backend='mongo', database='database-name') # очередь в монго
```

3.4.3 Генератор заданий

Также для сокращения потребления памяти очередью заданий вы можете воспользоваться методом *task_generator*, задания из которого будут браться только по мере опустошения очереди заданий.

3.5 Обработка ошибок

3.5.1 Правила обработки запросов

- Если запрос выполнен успешно, то вызывается метод-обработчик, связанный с заданием по его имени.
- Если запрос был нарушен из-за сетевой ошибки, то задание снова отправляется в очередь заданий.
- Если произошла непредвиденная ошибка в обработчике задания, то обработка задания прекращается немедленно. Ошибка в обработчике не фатальна: она не влияет на работу обработчиков других заданий.

3.5.2 Сетевые ошибки

Сетевой ошибокй считаются следующие случаи: * произошла ошибка во время передачи данных, например, сервер отверг запрос на соеденине или оборвал связь до того, как данные были переданы или данные передавались слишком долго * данные были переданы, но HTTP-статус ответа отличен от 2xx или 404

Существует ряд настроек, для задания критериев, по которым сетеовой запрос помечается как ошибочный.

Вы можете управлять таймаутами: *timeout* и *connect_timeout*. Для задания этих настроек вам нужно конструировать Task-обеъкт с помощью настроенного Grab-объекта:

```
g = Grab(timeout=5, connect_timeout=1, url='http://example.com')
t = Task('example', grab=g)
```

Вы можете указывать дополнительный список HTTP-статусов, которые будут считаться успешными:

```
t = Task('example', url='http://example.com', valid_status=(500, 501, 502))
```

3.5.3 Повторно выполнение заданий

Завершившееся сетевой ошибкой задание повторно отправляется в очередь заданий. Количество попыток зависит от атрибута *Spider.network_try_limit* и по умолчанию равно десяти. Номер попытки хранится в атрибуте *Task.network_try_count*. Если все попытки исчерпаны, то задание больше не добавляетя в очередь. Кроме того, если в пауке определён метод *task_<имя задания>_fallback*, то он вызывается и получает в качестве единственного аргумента Task-объект, невыполненного задания.

Также бывает, что хотя HTTP-статус не содержит ошибки, но данные ответа являются неверными, например, когда отправленная форма отображается ещё раз из-за неверно заполненного поля или когда сайт показывает каптчу или сообщение о том, что ваш IP забанен. В таких случаях нужно вручную (из метода-обработчика) отправить это задание ещё раз в очередь. Дабы избежать бесконечного добавления такого задания в очередь существует ещё один счётчик: *Task.task_try_count* и соответсвующее ему ограничение в пауке *Spider.task_try_limit*. Важное замечание, в случае использования *task_try_count* вы должны самостоятельно увеличивать его значение при повторной отправке задания в очередь:

```
def task_google(self, grab, task):
    if captcha_found(grab):
        yield Task('google', url=grab.config['url'], task_try_count=task.task_try_count + 1)

def task_google_fallback(self, task):
    print 'Google is not happy with you IP address'
```

3.5.4 Статистика ошибок

После завершения работы паука, или даже во время его работы, вы можете получить суммарную информацию о количестве сделанных запросов и количестве различных ошибок с помощью метода *Spider.render_stats*.

3.6 Система кэширования сетевых запросов

В целях ускорения тестирования паука в процессе разработки, а также ускорения повторного парсинга данных, была разработана система кэширования. В данный момент есть ограничение - только GET

запросы могут быть закешированными. Важно понимать что кэш в Spider это не полноценный http-прокси-сервер это лишь средство для отладки. Хотя стоит заметить, что даже в таком примитвной реализации система кэширования в большинстве случаев позволяет успешно использовать закэшированные данные для повторного парсинга в случае изменения логики обработки данных.

3.6.1 Бэкенды системы кэширования

Кэш в Spider разработан в виде отдельнього слоя для того, чтобы можно было подключать различные базы данных. В данный момент доступна только одна реализация кэша - хранение данных в mongodb

3.6.2 Исползование кэша

Для того, чтобы паук мог искать запрашиваемые документы в кэше и сохранять в кэш полученные данные, нужно вызывать метод *setup_ cache* до начала работы паука:

```
bot = ExampleSpider()
bot.setup_cache(database='some-database')
bot.run()
```

Вышенаписанный код активирует кэш, документы будут искаться и сохраняться в базе данных mongodb с именем 'some-datbase'. Имя коллекции с документами: "cache".

Есть несколько настроек для регулирования работы кэша:

backend бэкенд кэша, сейчас ничего кроме "mongo" не работает

database имя mongodb базы данных

use_compression использование gzip для сжатия данных перед помещением их в кэш.

3.6.3 Сжатие кэшируемых данных

По умолчанию сжатие включено. Сжатие позволяет на порядок уменьшить размер места в базе данных, необходимого для хранения закешированных документов. Сжатие снижает скорость работы паука, но не намного.

Глава 4

API

Вся нижеследующая информация сгенерирована из комментариев в исходном коде. Поэтому она на английском языке. Документы из раздела API полезны тем, что они показывают описания всех аргументов каждого метода и класса библиотеки Grab.

Базовый интерфейс:

4.1 grab.base: API базового класса

`grab.base.BaseGrab`
псевдоним класса `Grab`

4.2 grab.error: классы исключений

class `grab.error.GrabError`
All custom Grab exception should be children of that class.

class `grab.error.GrabNetworkError`
Raises in case of network error.

class `grab.error.GrabTimeoutError`
Raises when configured time is outed for the request.

In curl transport it is CURLE_OPERATION_TIMEDOUT (28)

class `grab.error.GrabMisuseError`
Indicates incorrect usage of grab API.

class `grab.error.DataNotFound`
Raised when it is not possible to find requested data.

4.3 grab.response: класс ответа сервера

`grab.response.Response`
псевдоним класса `Document`

Утилиты:

4.4 grab.upload

Глава 5

Всякая фигня

- genindex
- modindex
- search

g

B

D

G

R