# 爬虫中的数据处理

## 课程介绍

1. **爬虫数据解析与提取.**

从响应中提取出真正想要的数据

1. **爬虫数据清洗**

格式处理和统一

1. **爬虫数据存储**

## 课程与开发环境介绍

课程涉及系统环境介绍：

centos7系统 + Docker18.03

[docker hub](https://github.com/docker-library/docs)镜像地址<https://github.com/docker-library/docs>

用到的镜像：python:3.6.5-stretch、mysql:5.7、postgres:latest、mongo:latest

python3.7有些模块与之前的有些差异, 可能运行结果不同

所使用的Python环境的Dockerfile

# 注意：pip.conf文件为了解决, pip安装时可能出现的如certificate verify failed的错误

FROM python:3.6.5-stretch

COPY pip.conf /root/.config/pip/

RUN pip install ipython

# 使用到的Python第三方模块

RUN pip install requests &&\

pip install lxml &&\

pip install jsonpath &&\

pip install bs4 &&\

pip install pyquery &&\

pip install js2py &&\

pip install pyexecjs &&\

pip install html5lib &&\

pip install redis &&\

pip install pymysql &&\

pip install sqlalchemy &&\

pip install django==1.11 &&\

pip install psycopg2 &&\

pip install psmongo &&\

pip install mongoengine

# 安装node8, 供pyexecjs使用

RUN groupadd --gid 1000 node \

&& useradd --uid 1000 --gid node --shell /bin/bash --create-home node

# gpg keys listed at https://github.com/nodejs/node#release-team

RUN set -ex \

&& for key in \

94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \

FD3A5288F042B6850C66B31F09FE44734EB7990E \

71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \

DD8F2338BAE7501E3DD5AC78C273792F7D83545D \

C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \

B9AE9905FFD7803F25714661B63B535A4C206CA9 \

56730D5401028683275BD23C23EFEFE93C4CFFFE \

77984A986EBC2AA786BC0F66B01FBB92821C587A \

8FCCA13FEF1D0C2E91008E09770F7A9A5AE15600 \

; do \

gpg --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys "$key" || \

gpg --keyserver hkp://ipv4.pool.sks-keyservers.net --recv-keys "$key" || \

gpg --keyserver hkp://pgp.mit.edu:80 --recv-keys "$key" ; \

done

ENV NODE\_VERSION 8.11.4

RUN ARCH= && dpkgArch="$(dpkg --print-architecture)" \

&& case "${dpkgArch##\*-}" in \

amd64) ARCH='x64';; \

ppc64el) ARCH='ppc64le';; \

s390x) ARCH='s390x';; \

arm64) ARCH='arm64';; \

armhf) ARCH='armv7l';; \

i386) ARCH='x86';; \

\*) echo "unsupported architecture"; exit 1 ;; \

esac \

&& curl -fsSLO --compressed "https://nodejs.org/dist/v$NODE\_VERSION/node-v$NODE\_VERSION-linux-$ARCH.tar.xz" \

&& curl -fsSLO --compressed "https://nodejs.org/dist/v$NODE\_VERSION/SHASUMS256.txt.asc" \

&& gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \

&& grep " node-v$NODE\_VERSION-linux-$ARCH.tar.xz\$" SHASUMS256.txt | sha256sum -c - \

&& tar -xJf "node-v$NODE\_VERSION-linux-$ARCH.tar.xz" -C /usr/local --strip-components=1 --no-same-owner \

&& rm "node-v$NODE\_VERSION-linux-$ARCH.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \

&& ln -s /usr/local/bin/node /usr/local/bin/nodejs

ENV YARN\_VERSION 1.6.0

RUN set -ex \

&& for key in \

6A010C5166006599AA17F08146C2130DFD2497F5 \

; do \

gpg --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys "$key" || \

gpg --keyserver hkp://ipv4.pool.sks-keyservers.net --recv-keys "$key" || \

gpg --keyserver hkp://pgp.mit.edu:80 --recv-keys "$key" ; \

done \

&& curl -fsSLO --compressed "https://yarnpkg.com/downloads/$YARN\_VERSION/yarn-v$YARN\_VERSION.tar.gz" \

&& curl -fsSLO --compressed "https://yarnpkg.com/downloads/$YARN\_VERSION/yarn-v$YARN\_VERSION.tar.gz.asc" \

&& gpg --batch --verify yarn-v$YARN\_VERSION.tar.gz.asc yarn-v$YARN\_VERSION.tar.gz \

&& mkdir -p /opt \

&& tar -xzf yarn-v$YARN\_VERSION.tar.gz -C /opt/ \

&& ln -s /opt/yarn-v$YARN\_VERSION/bin/yarn /usr/local/bin/yarn \

&& ln -s /opt/yarn-v$YARN\_VERSION/bin/yarnpkg /usr/local/bin/yarnpkg \

&& rm yarn-v$YARN\_VERSION.tar.gz.asc yarn-v$YARN\_VERSION.tar.gz

RUN npm install crypto-js crypto

## 1. 爬虫数据解析与提取

### 爬虫数据解析与提取简介

#### 什么是爬虫数据解析与提取？

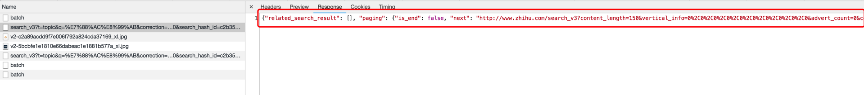
一般情况下, 爬虫最终想要的数据不是获取到的原始数据全部, 而是其中的某一部分的内容。

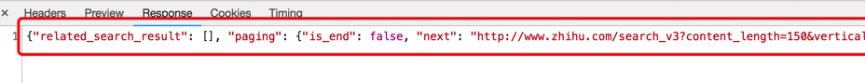
在爬虫中使用一切手段, 从原始数据中定位、截取出目标数据的过程就是爬虫数据解析与提取。

有的情况下, 爬虫最终想要的数据就是直接获取到的原始数据, 如当抓取图片、视频、CSV、PDF等媒体/文本文件时, 但更多情况下, 我们拿到的响应数据都是HTML、JSON等类型的字符串数据。

#### 爬虫抓取的原始数据有哪些格式或类型？

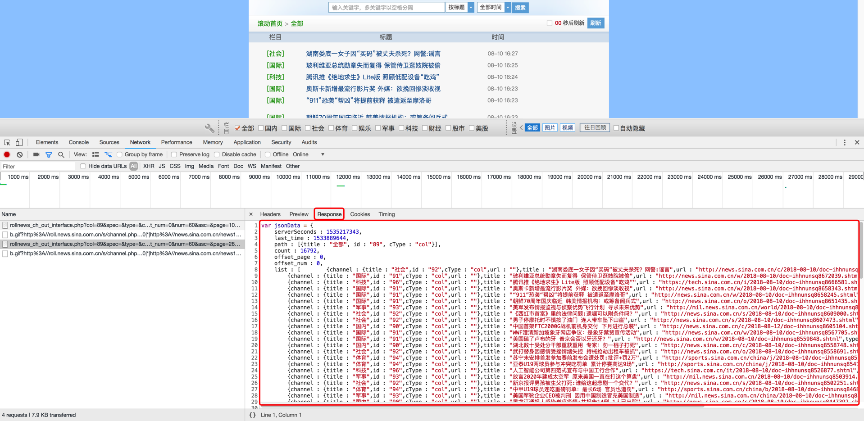
* HTML类型字符串
* JSON类型字符串





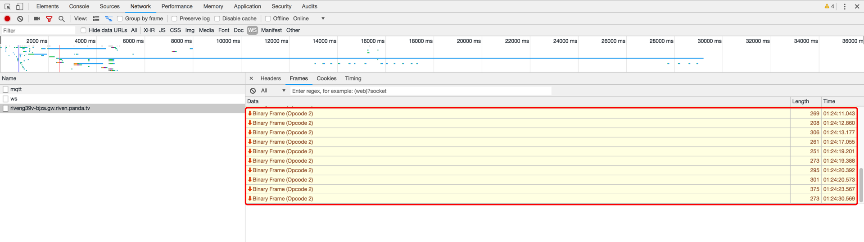
* 图片、视频、PDF等二进制流式类型字符串
* 其他类型
  + JavaScript代码

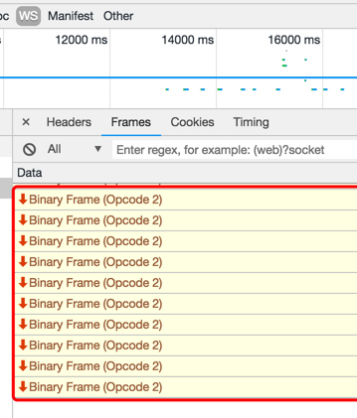
使用python中的模块来执行js代码, 就得到了js中包含的数据. 如果是复杂的js代码, 还需要进行分析, 找到规律, 进行简单的还原.





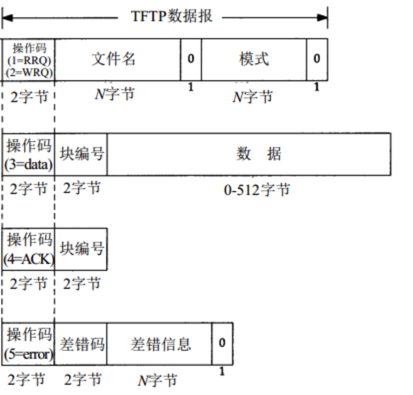
* + 网站自定义格式





弹幕使用的是websocket协议. 发送的是字节数据. 需要对js代码进行解析.

如TFTP协议传输数据一样。



但由网站自己定义每个字节代表的含义, 如果想要获得数据, 那么首先需要弄清楚网站该数据的解析规则, 也就是那些字节表示如操作码数据, 那个字节开始才表示真正的数据(通常都在其前端js代码中可以找到对应的js处理逻辑代码)

#### 有哪些手段, 进行爬虫数据解析与提取？

* 无固定格式字符串(普通字符串)
  + 正则表达式
* HTML类型字符串
  + 正则表达式
  + xpath规则
  + css选择器规则
* JSON类型字符串
  + 正则表达式
  + 转换python对象
  + jsonpath规则
* 图片、视频、PDF等二进制流式类型字符串
  + 无需解析提取(原始数据就是最终想要的数据)
  + 利用图片、视频等识别技术, 识别出其中的信息作为最终想要的数据
* JavaScript代码, 运行js代码, 得到结果.
  + 正则表达式
  + js代码执行器：nodejs、phantomjs、pyv8等可运行js的环境
* 网站自定义格式（难点在分析对方的js代码, 找出解析数据的那段js代码）
  + js代码翻译
  + 正则表达式
  + js代码执行器：nodejs、phantomjs、chrome等可运行js的环境

总的来说, 爬虫的数据解析需要掌握：

* 爬虫数据四大解析规则：**正则表达式规则**、**Xpath规则**、**CSS选择器规则**、JsonPath规则
* 前端的三大语法：**HTML、CSS、JavaScript**

### 1.1. 正则表达式规则与运用

应用场景：一切字符串数据的解析提取操作

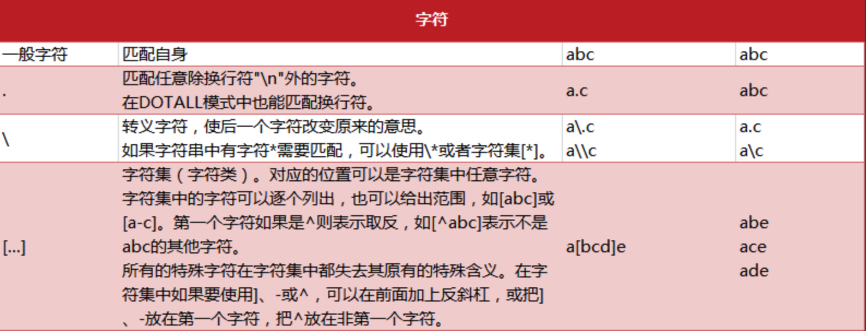
学习两部分的内容: 正则表达式的规则和python re模块api的使用方法. 正则表达式的解析速度最快.

#### 正则表达式规则

**注意：不同语言有不同的正则表达式语法, 规则不通用, 但基本大同小异**

#### Python正则表达式语法规则

1. 普通字符语法



import re

# re.match(表达式语法, 匹配的字符串)

**"\*\*\*\* 1.1 一般字符 \*\*\*\*"**

pattern = "python"

re.match(pattern, "python") # python

re.match(pattern, "python2.7") # python

**"\*\*\*\* 1.2 "." 万能匹配, 匹配任意一个字符 \*\*\*\*"**

pattern = "python."

re.match(pattern, "python2") # python2

re.match(pattern, "python3") # python3

re.match(pattern, "python3.7") # python3

re.match(pattern, "python\n") # None (无法匹配)

re.match(pattern, "python\n", re.S) # python\n, 使用re.S使点能够匹配到换行符

**"\*\*\*\* 1.3 "\" 正则转义字符 \*\*\*\*"**

# 想要匹配 ".", 如果直接使用 "." 作为正则表达式, 则表示匹配任意字符, 可以使用转义字符, 使用 "\." 来只匹配 "."

re.match("\.", ".")

pattern = "\.\\*\?\+\^\$"

re.match(pattern, ".\*?+^$abcd") # .\*?+^$

# python中想要保存 "\", 如果直接输入一个 "\", 会报错.

# "\"

# 想要保存一个字符串格式的 "\", 也需要用2个"\", 第1个反斜杠表示转义, 把第2个反斜杠的特殊含义取消掉.

"\\"

# 匹配一个反斜线字符 \

# 字符串中表示单个反斜线

len("\\") # 1

pattern = "\\"

re.match(pattern, "\\") # 抛出异常, 正则中使用4个反斜杠来匹配1个反斜杠

# 正则表达式本质上就是一个字符串

# 而正则中的\和普通字符串的\作用是有区别的, 因此正则中表示一个\

# 需要将字符串形式的\\中的反斜线进行正则转义（各自加上一个反斜线）, 才能表示它的正则形式

pattern = "\\\\" # 正则中表示单个反斜线

re.match(pattern, "\\") # \

# r作用：将字符串转为纯正则模式, 表示原生字符串, 忽略字符串对其的干扰

pattern = r"\\" # 正则中表示单个反斜线

re.match(pattern, "\\") # \

re.match(r"\\\\", "\\\\") # \\ print() 与 直接输入的差别

re.match("\\\\", "\\")

re.match(r"\\", "\\")

**"\*\*\*\* 1.4 中括号字符集里的特殊符号将失去所有特殊含义, 除了\ ,放在首位的^, 放在结尾的$, 放在2个字母或2个数字之间的-\*\*\*\*"**

pattern = "[.\*?+^$]"

re.match(pattern, "\*")

re.match(pattern, "$")

# 但当位于首位的^、作为连字符的-、]具备单独的含义

re.match("[a-z]", str1) # 只匹配小写字母, 连字符, 大小写字母和数字

re.match("[^a-z]", str2) # 只匹配非小写字母, 取反操作, 不是[]中的内容才匹配

re.match("[\^a-z]", str3) # 只匹配以小写字母或^

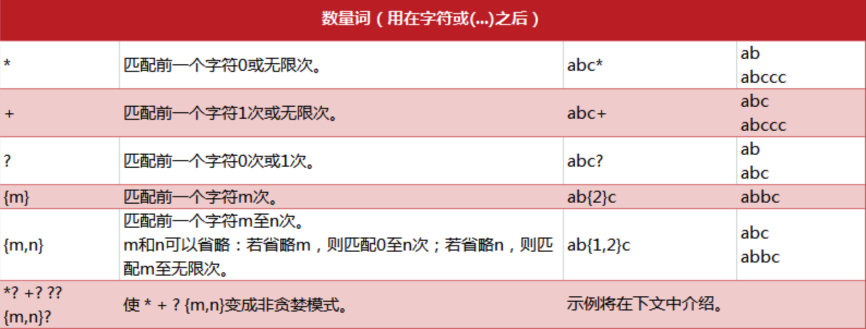
re.match("[a\-z]","-") # 只匹配a、-、z 这三个字母

re.match("[\]]","]") # 只匹配"]"

1. 预定义字符集语法



1. 数量词语法



数据词是用在字符、字符串集、分组之后, 控制它的匹配次数。

数量词的贪婪与非贪婪：

* + 作用：控制数量词的匹配个数
  + 用法：默认是使用贪婪模式；在数量词后加上一个?后, 则表示使用非贪婪模式

匹配后贪婪与非贪婪模式的匹配个数：

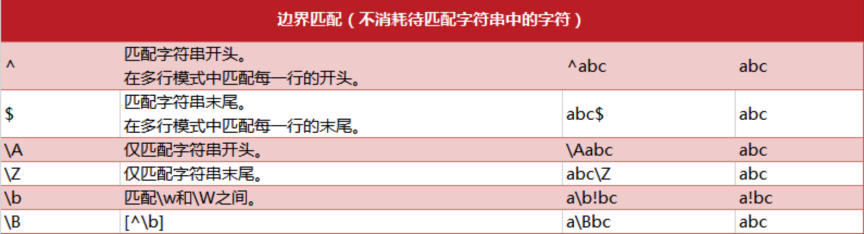


import re

re.match("\d+\.\d{2,6}", "3.1415926") # 3.141592

re.match("\d+\.\d{2,6}?", "3.1415926") # 3.14

1. 边界匹配语法



# \b相当于\w和\W的边界, 可以用来匹配出单词；\B则常用来判断单词的连贯性

import re

re.match(r"\w\W\w", "q q") # q q

re.match(r"\w\b\W\b\w", "q q") # q q

re.match(r"\w+\b\W+\b\w+", "hello-world") # hello-world

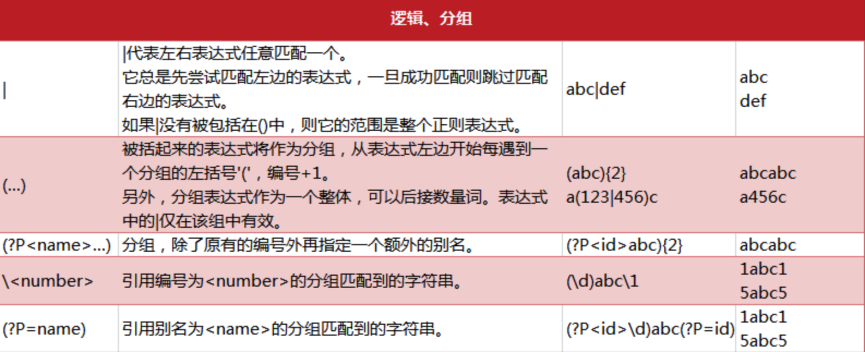
re.match(r"\w+\b\W+\b\w+", "hello - -- -world") # hello - -- -world

# 注意：必须使用r, 将字符串变为正则模式, 因为\b在普通字符串中也是具有含义的

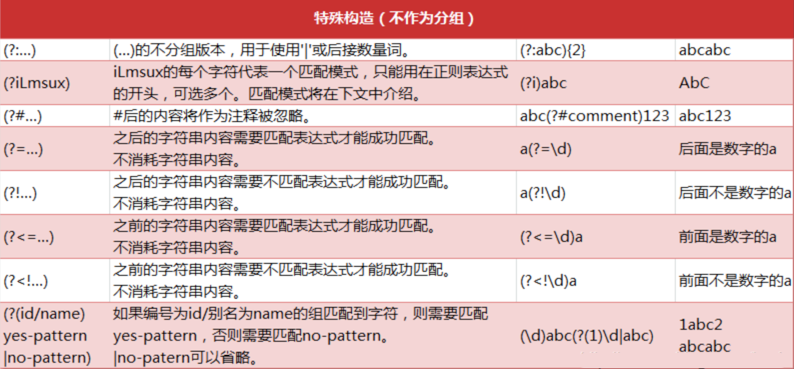
re.match(r"Tom\B[A-Z][a-z]+", "TomCruise") # 可以匹配

re.match(r"Tom\B[A-Z][a-z]+", "Tom Cruise") # 无法匹配

1. 逻辑、分组语法



1. 特殊构造语法



import re

"\*\*\*\* 6.1 (?aiLmsux) 作用：替换flags参数 \*\*\*\*"

# aLu, 已经几乎不使用了

re.match(r"(?i)ABCD", "abcd") # (?i), 替换flags=re.I参数, 与flags=re.I所起的作用相同, 忽略大小写进行匹配

# abcd

re.match(r"ABcd", "abcd", flags=re.I) # abcd

"\*\*\*\* 6.2 (?:...) 作用：进行匹配, 但不计入分组 \*\*\*\*")

re.match(r"(ab)(cd)", "abcd").group(1) # ab

re.match(r"(?:ab)(cd)", "abcd").group(1) # cd

"\*\*\*\* 6.3 (?imsx-imsx:...) 作用：为组内设置或去除flags效果 \*\*\*\*"

# 局部匹配规则, 在整体匹配的基础上, 局部的字符设置或去除flags的效果

# Python3.6新增正则语法

re.match(r"(?i:AB)cd", "abcd") # ?i忽略大小写, AB是以忽略大小写的方式进行匹配的

# abcd

re.match(r"(?-i:AB)cd", "abcd", flags=re.I) # ?-i去除忽略大小写的效果, AB是以大写字母的方式进行匹配的

# 无法匹配

re.match(r"(?s-i:AB)cd", "abcd", flags=re.I) # ?s 使点能够匹配换行符

# 无法匹配

# 注意不能在内部即开启又关闭同一模式, 如：i-i、imsx-imsx都是错误的

"\*\*\*\* 6.4 (?#...) 作用：添加注释 \*\*\*\*")

re.match(r"(?#这是注释)abcd", "abcd")

"\*\*\*\* 6.5 (?=...) 作用：向右正向断言匹配, 后面如果是 \*\*\*\*"

#如果后面是cd就匹配, 否则不匹配, 相当于if判断

re.match(r"\w+(?=cd)", "abcd") # 可以匹配 abcd

re.match(r"\w+(?=cd)", "abcf") # 无法匹配

"\*\*\*\* 6.6 (?!...) 作用：向右反向断言匹配, 后面如果不是 \*\*\*\*"

# 后面如果不是cd就匹配

re.match(r"ab(?!cd)\w+", "abcd") # 无法匹配

re.match(r"ab(?!cd)\w+", "abcf") # 可以匹配 abcf

"\*\*\*\* 6.7 (?<=...) 作用：向左正向断言匹配, 前面如果是 \*\*\*\*"

# cd前面如果是ab就匹配

re.match(r"\w+(?<=ab)cd", "abcd") # 可以匹配 abcd

re.match(r"\w+(?<=ab)cd", "afcd") # 无法匹配

"\*\*\*\* 6.8 (?<!...) 作用：向左反向断言匹配, 前面如果不是 \*\*\*\*"

# 后面如果不是ab就匹配

re.match(r"\w+(?<!ab)cd", "abcd") # 无法匹配

re.match(r"\w+(?<!ab)cd", "afcd") # 可以匹配 afcd

"\*\*\*\* 6.9 (?(id/name)yes-pattern|no-pattern) 作用：(id/name)表示分组的id或名称, 如果给定的分组存在, 则使用yes规则进行匹配, 否则使用no规则进行匹配 \*\*\*\*")

re.match(r"(<){0,1}abcd(?(1)>|)", "<abcd>") # 可以匹配 <abcd>. 首先匹配0个或1个尖括号<, 如果分组存在, 就匹配右尖括号, 否则就不进行匹配.

re.match(r"(<){0,1}abcd(?(1)>|)", "<abcd") # 无法匹配, 因为分组存在, 就要匹配右尖括号, 但右尖括号不存在, 所以无法匹配.

re.match(r"(<){0,1}abcd(?(1)>|)", "abcd") # 可以匹配 abcd, 因为分组不存在, 就不会去匹配右尖括号, 所以能匹配到abcd.

re.match(r"(<){0,1}abcd(?(1)>|)", "abcd") # 可以匹配 abcd, 这个规则表示匹配要么两边都有尖括号, 要么都没有尖括号

#### Python的正则模块 -- re

注意：以下介绍以Python3.6为基础

python3.6 re模块文档介绍

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/>

1. [re模块内容（属性和方法）](https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#module-contents)

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#module-contents>

* + 只掌握I、M、S、X四种模式
  + 重点掌握compile、match、search、findall/finditer、sub/subn、split方法, 注意各自的参数和返回值

1. [re中的正则表达式对象](https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#regular-expression-objects)

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#regular-expression-objects>

* + 重点掌握match、search、findall/finditer、sub/subn、split方法, 并注意其余re模块对应方法的使用区别
  + 了解groups、groupindex、pattern等属性

1. [re中的匹配对象](https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#match-objects)

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/re/#match-objects>

* + 重点掌握group、groups、\_\_getitem\_\_、groupdict等方法
  + 了解lastindex、span、expand等属性或方法

import re

match = re.match("(\w+) (?P<name>\w+)", "Tom Cruise")

match.expand(r"My name is '\1 \2'")

match.expand(r"My name is '\1 \g<name>'")

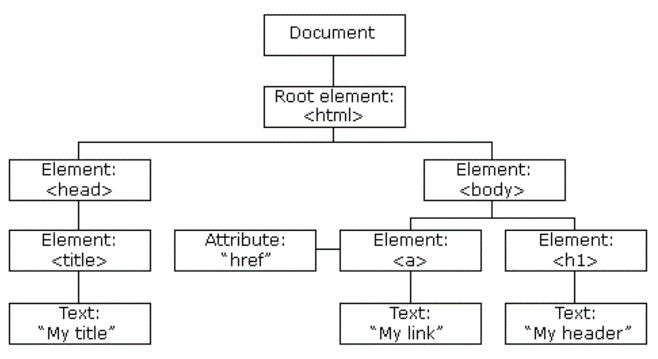
# My name is 'Tom Cruise'

### 1.2. Xpath规则与运用

Xpath：XML Path Language

应用场景：用于确定XML、HTML文档中某部分的位置

文档树（DOM Tree）：



#### Xpath规则

**节点选取基本路径表达式：如何选取路径**

|  |  |
| --- | --- |
| 表达式 | 描述 |
| nodename | 选取此节点的所有子节点 |
| / | 从根节点选取 |
| // | 从匹配选择的当前节点选择文档中的节点, 而不考虑它们的位置 |
| . | 选取当前节点 |
| .. | 选取当前节点的父节点 |
| @ | 选取属性 |

html/body//div/@class

**节点选取通配符：**

|  |  |
| --- | --- |
| 通配符 | 描述 |
| \* | 匹配任何元素节点 |
| @\* | 匹配任何属性节点 |
| node() | 匹配任何类型的节点 |

**实例**

在下面的表格中, 我们列出了一些路径表达式, 以及这些表达式的结果：

|  |  |
| --- | --- |
| 路径表达式 | 结果 |
| /bookstore/\* | 选取 bookstore 元素的所有子元素。 |
| //\* | 选取文档中的所有元素。 |
| //title[@\*] | 选取所有带有属性的 title 元素。 |

**Xpath谓语条件（Predicates）：**

所谓"谓语条件", 就是对路径表达式的附加条件, 都写在方括号"[]"中, 表示对节点进行进一步的筛选

**实例**

在下面的表格中, 我们列出了带有谓语的一些路径表达式, 以及表达式的结果：

|  |  |
| --- | --- |
| 路径表达式 | 结果 |
| /bookstore/book[1] | 选取属于 bookstore 子元素的第一个 book 元素。 |
| /bookstore/book[last()] | 选取属于 bookstore 子元素的最后一个 book 元素。 |
| /bookstore/book[last()-1] | 选取属于 bookstore 子元素的倒数第二个 book 元素。 |
| /bookstore/book[position()<3] | 选取最前面的两个属于 bookstore 元素的子元素的 book 元素。 |
| //title[@lang] | 选取所有拥有名为 lang 的属性的 title 元素。 |
| //title[@lang='eng'] | 选取所有 title 元素, 且这些元素拥有值为 eng 的 lang 属性。 |
| /bookstore/book[price>35.00] | 选取 bookstore 元素的所有 book 元素, 且其中的 price 元素的值须大于 35.00。 |
| /bookstore/book[price>35.00]/title | 选取 bookstore 元素中的 book 元素的所有 title 元素, 且其中的 price 元素的值须大于 35.00。 |

谓语条件语法：

1. 函数调用 [查看更多函数](http://www.w3school.com.cn/xpath/xpath_functions.asp)

<http://www.w3school.com.cn/xpath/xpath_functions.asp>

节点函数

|  |  |  |
| --- | --- | --- |
| Node functions |  | Examples |
| name() | 取当前标签的名称 | //[starts-with(name(), 'h'] |
| text() | 取当前标签的文本内容 | //button[text()="Submit"]  //button/text() |
| lan(str) |  |  |
| namespace-uri() |  |  |
| count() | 计数 | //table[count(tr)=1] |
| position() | 定位 | //ol/li[position()=2] |

类型转换

|  |  |
| --- | --- |
| Type conversion | Examples |
| string() | 把一个对象转换为字符串 |
| number() |  |
| boolean() |  |

布尔函数

|  |  |
| --- | --- |
| Boolean functions | Examples |
| not(expr) | button[not(starts-with(text(), "Submit"))] |

字符串函数

|  |  |
| --- | --- |
| String functions | Examples |
| contains() | font[contains(@class, "head")] |
| starts-with() | font[starts-with(@class, "head")] |
| ends-with() | font[ends-with(@class, "head")] |
| concat(x, y) |  |
| substring(str, start, len) |  |
| substring-before("01/02", "/") #=> 01 |  |
| substring-after("01/02", "/") #=> 02 |  |
| translate() |  |
| normalize-space() |  |
| string-length() |  |

1. 运算操作 [查看更多运算符](http://www.w3school.com.cn/xpath/xpath_operators.asp)

**Operators**

# comparision 比较运算

//a[@id = "xyz"]

//a[@id != "xyz"]

# Logic (and/or) 逻辑运算

//div[@id="head" and position()=2]

//div[(x and y) or not(z)]

Use comparision and logic operators to make conditionals

1. Indexing索引操作

//a[1] # first <a>

//a[last()] # last <a>

//ol/li[2] # second <li>

//ol/li[position()=2] # same as above

//ol/li[position()>1] # :not(:first-child)

Use [] with a number, or last() or position()

1. 链式谓语  Chaining order

a[1][@href='/'] # 先选择第一个元素, 再进行过滤

a[@href='/'][1] # 先进行过滤, 再选择过滤的结果中的第1个元素

Order is significant, these two are different

1. 嵌套谓语 Nesting predicates

//section[//h1[@id='hi']] # 选择section标签, 此section中必须有id="hi"的h1标签.

This returns <section> if it has an <h1> descendant with id='hi'

**Xpath轴（axis）：**

轴其实就是以当前节点(自己)为中心, 来选取其他节点

|  |  |
| --- | --- |
| 轴名称 | 结果 |
| ancestor | 选取当前节点的所有先辈（父、祖父等）。 |
| ancestor-or-self | 选取当前节点的所有先辈（父、祖父等）以及当前节点本身。 |
| attribute | 选取当前节点的所有属性。 |
| child | 选取当前节点的所有子元素。 |
| descendant | 选取当前节点的所有后代元素（子、孙等）。 |
| descendant-or-self | 选取当前节点的所有后代元素（子、孙等）以及当前节点本身。 |
| following | 选取文档中当前节点的结束标签之后的所有节点。 |
| namespace | 选取当前节点的所有命名空间节点。 |
| parent | 选取当前节点的父节点。 |
| preceding | 选取文档中当前节点的开始标签之前的所有节点。 |
| preceding-sibling | 选取当前节点之前的所有同级节点。 |
| self | 选取当前节点。 |

xpath轴使用示例：

//ul/li # ul > li # 选择ul的直接子节点

//ul/child::li # ul >li (same) # 选择ul的直接子节点

//ul/following-sibling::li # ul - li # 向下选择和ul同级的li

//ul/descendant-or-self::li # ul li # 选择所有子孙li节点 //ul//li

//ul/ancestor-or-self::li # $('ul').closest('li') # 选择所有先辈li节点

以ul为中心, 分别在子、向后同胞、后辈、先辈中寻找li元素节点

following、preceding既包括“亲兄弟”也包括“堂兄弟”

following-sibling、preceding-sibling只包括“亲兄弟”

#### lxml模块

lxml模块是一个在Python中用于处理XML和HTML的工具。特点就是快、性能强。

**安装**

lxml依赖于两个C语言编写的库libxml2-dev、libxslt-dev, 以ubuntu为例：

sudo apt-get install libxml2-dev libxslt-dev

然后在虚拟环境或其他Python解释器环境安装pip install lxml

更多详情, 请查看其[github地址](https://github.com/lxml/lxml)、[官方文档地址](https://lxml.de/)

lxml工具功能强大, 可对XML、HTML里的内容进行非常精确的增删改查操作, 但我们爬虫中基本只用到**查询(搜索)**操作。根据一定规则（如xpath）获取某个元素节点的属性文本内容

lxml默认使用内置的解析器, 也就是libxml2-dev进行解析。特点：**快、快、快**

**爬虫中lxml模块常用方法或对象**

lxml模块[API介绍](https://lxml.de/api/index.html)

常用：

* lxml
  + Element, html文档使用lxml解析之后生成的对象, 可以从此对象中选择节点和节点内容
  + ElementTree, 是Element的父对象, 可以通过 .getroot()来获得到Element对象.
* lxml.etree
  + HTMLParser, 解析html文档
  + XMLParser, 解析xml文档
  + XPath, 类似于正则表达式中compile方法返回的正则对象, 把xpath规则编译之后返回的对象, 可以实现xpath规则的复用
  + parse: 对文件或二进制流数据(类文件对象)进行解析, 返回ElementTree对象
  + fromstring: 对字符串进行解析, 返回Element对象. fromstring在解析时需要指定使用的解析器.
  + tostring: 对Element对象进行格式化, 返回bytes类型字符串数据
  + HTML: 相当于fromstring方法指定HTMLParser解析器, 解析字符串.
  + XML: 相当于fromstring方法指定XMLParser解析器

**lxml数据解析**

数据解析：将原始数据进行转换为格式完整的DOM对象

对于xml、html数据, lxml默认使用内置的解析器, 也就是libxml2-dev进行解析。特点：**快、快、快**

1. 使用方法一(常用)：利用fromstring或HTML方法解析XML数据

import lxml.etree

# 待解析数据

xml\_data = '''<?xml version="1.0" encoding="utf-8"?>

<note>

<to>George</to>

<from>John</from>

<heading>Reminder</heading>

<body>Don't forget the meeting!</body>

</note>'''.encode("utf-8")

# 1. 定义解析器

parser = lxml.etree.XMLParser()

# 2. 数据解析, 默认就是使用的XMLParser(), 返回Element对象

root = lxml.etree.fromstring(xml\_data, parser)

print(root) # Element 对象

xml\_data2 = '''<note>

<to>George</to>

<from>John</from>

<heading>Reminder</heading>

<body>Don't forget the meeting!</body>

</note>

'''

# 默认使用的就是XMLParser

root2 = lxml.etree.fromstring(xml\_data2)

print(root2) # Element 对象

'''

<Element note at 0x7f8aad54ee88>

<Element note at 0x7f8aad54ee48>

'''

HTML数据的解析

import lxml.etree

# 待解析数据

html\_data = '''<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Hello World</title>

</head>

<body>

<div>Hello World</div>

</body>

</html>'''

# 1. 定义解析器, 使用HTMLParser()

parser = lxml.etree.HTMLParser()

# 2. 数据解析

# 默认使用的就是XMLParser, 解析HTML文档时必须要指定解析器, 如果不指定解析器, 将抛出异常

root = lxml.etree.fromstring(html\_data, parser)

print(lxml.etree.tostring(root))

'''

b'<html lang="en">\n<head>\n <meta charset="UTF-8"/>\n <title>Hello World</title>\n</head>\n<body>\n<div>Hello World</div>\n</body>\n</html>'

'''

# 使用HTMLParser来解析HTML文档时会自动对标签进行补全

html\_data2 = '''<meta charset="UTF-8">

<title>Hello World</title>

<div>Hello World</div>'''

# 默认使用的就是XMLParser

root2 = lxml.etree.fromstring(html\_data2, parser)

print(lxml.etree.tostring(root2))

'''

b'<html><head><meta charset="UTF-8"/>\n<title>Hello World</title>\n</head><body><div>Hello World</div></body></html>'

'''

简写（更常用）：

import lxml.etree

# 解析XML文档

root = lxml.etree.XML(xml\_data)

# 等价于

parser = lxml.etree.XMLParser()

root = lxml.etree.fromstring(xml\_data, parser)

# 解析HTML文档

root = lxml.etree.HTML(html\_data)

# 等价于

parser = lxml.etree.HTMLParser()

root = lxml.etree.fromstring(html\_data, parser)

1. 使用方法二：parse方法解析xml、html文件或相应的类文件对象

import io

import lxml.etree

# 待解析数据

html\_data = '''<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Hello World</title>

</head>

<body>

<div>Hello World</div>

</body>

</html>'''

root = lxml.etree.HTML(html\_data)

print(root) # Element 对象

# 使用parse方法

parser = lxml.etree.HTMLParser()

# 传入一个文件路径或类文件对象, 使用io.StringIO()转换为类文件对象

root2 = lxml.etree.parse(io.StringIO(html\_data), parser)

print(root2) # ElementClass 对象

print(root2.getroot()) # 通过getroot方法返回Element对象

'''

<Element html at 0x7f7ce7407ec8>

<lxml.etree.\_ElementTree object at 0x7f7ce7407f08>

<Element html at 0x7f7ce7407f48>

'''

对于HTML文件的解析, 或者直接使用lxml.etree.parse()方法, 传入文件的路径, 或者是先把文件读取到内存中, 再使用lxml.etree.HTML()方法来进行解析.

**lxml数据提取**

```python

import lxml.etree

# 待解析数据

html\_data = '''<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Hello World</title>

</head>

<body>

<div>Hello World</div>

</body>

</html>'''

''' 解析数据 '''

root = lxml.etree.HTML(html\_data) # root: Element 对象

''' 提取数据 '''

# 1. 直接使用Element对象的xpath方法, 提取

print(root.xpath(".//\*"))

# 2. 利用Xpath对象, 进行提取

xpath\_obj = lxml.etree.XPath(".//\*")

print(xpath\_obj(root))

'''

[<Element head at 0x7f9c052dae88>, <Element meta at 0x7f9c052dae48>, <Element title at 0x7f9c052daf48>, <Element body at 0x7f9c052daf88>, <Element div at 0x7f9c052dafc8>]

[<Element head at 0x7f9c052dae88>, <Element meta at 0x7f9c052dae48>, <Element title at 0x7f9c052daf48>, <Element body at 0x7f9c052daf88>, <Element div at 0x7f9c052dafc8>]

'''

```

# 注意xpath方法返回的永远是一个列表, 如果提取到了内容, 列表中的元素也是Element对象, 可以进一步的使用xpath方法进行选择. 如果没有提取到内容, 则返回一个空列表.

### 1.3. CssSelect规则与运用

[css选择器规则参考手册](http://www.w3school.com.cn/cssref/css_selectors.ASP)

<http://www.w3school.com.cn/cssref/css_selectors.ASP>

#### BeautifulSoup4模块

[BeautifulSoup4文档](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html)

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html>

安装

pip install bs4

bs4内置对象：

* [Tag](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#tag)：Tag对象与XML或HTML原生文档中的tag相同, 代表标签节点
* [BeautifulSoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#beautifulsoup)：BeautifulSoup 对象表示的是一个文档的全部内容, 整个DOM树, 类似于xpath中的Document对象. 使用bs模块对html文档进行解析, 得到一个BeautifulSoup对象, 再从BeautifulSoup对象中提取数据.
* [NavigableString](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#id13)：NavigableString类中包装有tag中的字符串, 从BeautifulSoup对象中提取到的字符串.
* [Comment](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#id14)：表示文档中的注释部分字符串

**bs4数据解析**

bs4可以使用多种解析器, 下表列出了主要的解析器,以及它们的优缺点:

| **解析器** | **使用方法** | **优势** | **劣势** |
| --- | --- | --- | --- |
| Python标准库 | BeautifulSoup(markup, "html.parser") | * Python的内置标准库 * 执行速度适中 * 文档容错能力强 | * Python 2.7.3 or 3.2.2)前 的版本中文档容错能力差 |
| lxml HTML 解析器 | BeautifulSoup(markup, "lxml") | * 速度快 * 文档容错能力强 | * 需要安装C语言库 |
| lxml XML 解析器 | BeautifulSoup(markup, ["lxml", "xml"])  BeautifulSoup(markup, "xml") | * 速度快 * 唯一支持XML的解析器 | * 需要安装C语言库 |
| html5lib | BeautifulSoup(markup, "html5lib") | * 最好的容错性 * 以浏览器的方式解析文档 * 生成HTML5格式的文档 | * 速度慢 * 不依赖外部扩展 |

一般情况下, 使用bs4时需要显示指明使用什么解析器, 否则会默认选择最优或最可行的解析器

使用：

from bs4 import BeautifulSoup

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 显示指出使用html.parser作为解析器. 解析的过程一是对数据进行格式化, 二是转换为树结构的DOM对象, 然后才能对其进行数据的提取

soup = BeautifulSoup(html\_doc, "html.parser") # 返回BeautifulSoup对象

print(soup.prettify())

'''

<meta charset="utf-8"/>

<title>

Hello World

</title>

<div>

<p id="hello">

Hello

</p>

<p>

World

</p>

</div>

'''

# 显示指出使用lxml作为解析器

soup2 = BeautifulSoup(html\_doc, "lxml") # 返回BeautifulSoup对象

print(soup2.prettify())

'''

<html>

<head>

<meta charset="utf-8"/>

<title>

Hello World

</title>

</head>

<body>

<div>

<p id="hello">

Hello

</p>

<p>

World

</p>

</div>

</body>

</html>

'''

**bs4数据提取**

方法一：搜索文档树之[select方法](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#id37)

from bs4 import BeautifulSoup

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 显示指出使用lxml作为解析器

soup = BeautifulSoup(html\_doc, "lxml") # 返回BeautifulSoup对象

# 利用select方法, 根据css选择器规则进行提取, 返回的结果集一个列表

print(soup.select("p#hello")) # 查询id为hello的p标签

'''

[<p id="hello">Hello</p>]

'''

# 结果集中的元素是Tag对象

print(type(soup.select("p#hello")[0])) # 每一个单独元素就是一个tag对象

'''

<class 'bs4.element.Tag'>

'''

# 获取id为hello的p标签内部的字符串

print(soup.select("p#hello")[0].string)

'''

Hello

'''

方法二：搜索文档树之[使用find家族方法](https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#id24)

from bs4 import BeautifulSoup

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 显示指出使用lxml作为解析器

soup = BeautifulSoup(html\_doc, "lxml") # 返回BeautifulSoup对象

# 利用find家族API进行标签定位和搜索

print(soup.find("p", id="hello", class="nav")) # 查询id为hello的p标签

'''

<p id="hello">Hello</p>

'''

# find方法如果查询到结果, 返回的结果就是一个Tag对象

print(type(soup.find("p", id="hello2")))

'''

<class 'bs4.element.Tag'>

'''

# 获取id为hello的p标签内部的字符串

print(soup.find("p", id="hello").string)

'''

Hello

'''

方法三：遍历文档树方法

from bs4 import BeautifulSoup

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 显示指出使用lxml作为解析器

soup = BeautifulSoup(html\_doc, "lxml") # 返回BeautifulSoup对象

# 通过点取方式定位

# 注意：点选方式只能获取到第一个属性

print(soup.body.div.p) # 查询id为hello的p标签

'''

<p id="hello">Hello</p>

'''

# soup下第一个p标签

print(soup.p)

'''

<p id="hello">Hello</p>

'''

# 通过特殊属性

# 如利用contents属性获取soup下第一个div的全部子节点

print(soup.div.contents)

'''

[<p id="hello">Hello</p>, <p>World</p>]

'''

#### pyquery模块

pyquery模块：一个类似jquery的python库, 主要实现了jquery中查询的操作.

* pyquery允许在xml、html文档上进行jquery查询
* pyquery的API设计与jquery非常类似
* pyquery可以使用lxml进行快速的xml和html解析

安装

pip install pyquery

**pyquery数据解析**

import lxml.etree

from pyquery import PyQuery

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 使用默认解析器, 解析html. 可以把d看成是JQuery中的$, 在JQuery中使用$来对元素进行选择, PyQuery中则使用d来选择html元素

d = PyQuery(html\_doc)

# 如果是xml数据, 需指明：PyQuery(html\_doc, parser="xml")

print(d)

'''

<html><head><meta charset="UTF-8"/>

<title>Hello World</title>

</head><body><div><p id="hello">Hello</p><p>World</p></div>

</body></html>

'''

# 直接传入lxml解析出来的对象, lxml.etree.HTML() 返回一个Element对象, 即PyQuery可以接收一个Element对象.

d = PyQuery(lxml.etree.HTML(html\_doc))

print(d)

'''

<html><head><meta charset="UTF-8"/>

<title>Hello World</title>

</head><body><div><p id="hello">Hello</p><p>World</p></div>

</body></html>

'''

[其他解析方式](https://pyquery.readthedocs.io/en/latest/#full-documentation)

**pyquery数据提取**

import lxml.etree

from pyquery import PyQuery

html\_doc = """

<meta charset="UTF-8">

<title>Hello World</title>

<div><p id="hello">Hello</p><p>World</p></div>

"""

# 直接传入lxml解析出来的对象

d = PyQuery(lxml.etree.HTML(html\_doc))

# d就相当于jquery中的$一样去使用

print(d("div")) # 返回的对象仍然是PyQuery对象, 也就是jquery中的$对象

print(type(d("div")))

'''

<div><p id="hello">Hello</p><p>World</p></div>

<class 'pyquery.pyquery.PyQuery'>

'''

print(d("div")("p#hello")) # 可继续进行链式选取

'''

<p id="hello">Hello</p>

'''

print(d("div").html()) # 获取div下的html内容

'''

<p id="hello">Hello</p>

<p id="hello">Hello</p><p>World</p>

'''

print(d("div").text()) # 只获取div下的文本内容

'''

Hello

World

'''

[更多完整API使用介绍](https://pyquery.readthedocs.io/en/latest/api.html)

### 1.4. JsonPath规则与运用

jsonpath是一个专门用于快速处理json数据的模块。

jsonpath规则之于json, 如xpath规则之于xml/html

特别大的json数据, 如有上千上万行时使用jsonpath, 小的json数据转换为python处理.

**json数据特征**

* json中只有两种结构：对象和数组
* 对象：在json中是“{}”括起来的内容, 数据结构为{"key":"vlaue1","key2": "value2"...........}的键值对结构, key只能是数字, 小数或字符串.
* 数组：在json中是‘[]’括起来的内容, 数据结构为["python","jsonpath","html"......], 与所有的语言一样, 使用索引获取值
* json的字段的类型可以是数字, 字符串、数组、对象。其中字符串必须使用双引号表示。

[jsonpath规则文档地址](http://goessner.net/articles/JsonPath/)

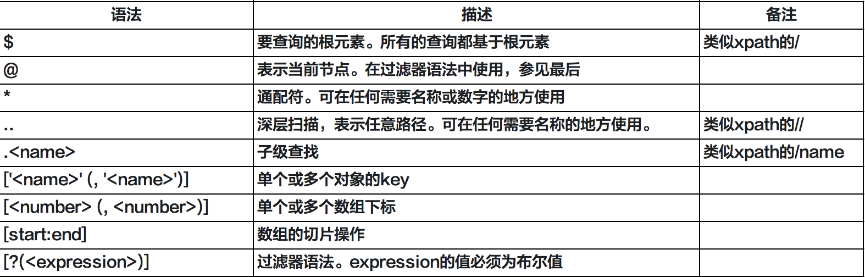
[jsonpath的Python实现](https://pypi.org/project/jsonpath/)

安装

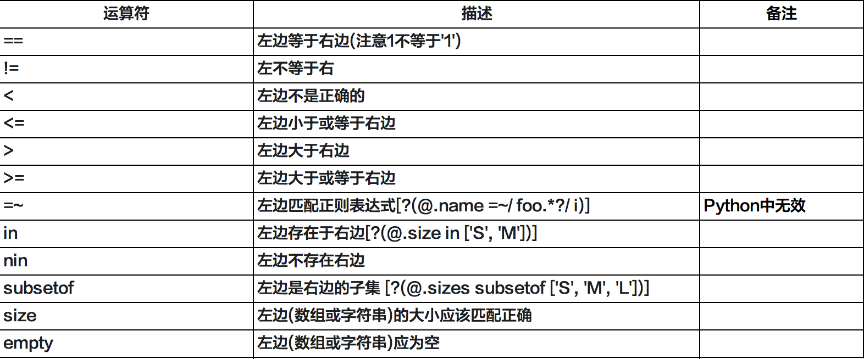
pip install jsonpath

#### jsonpath规则

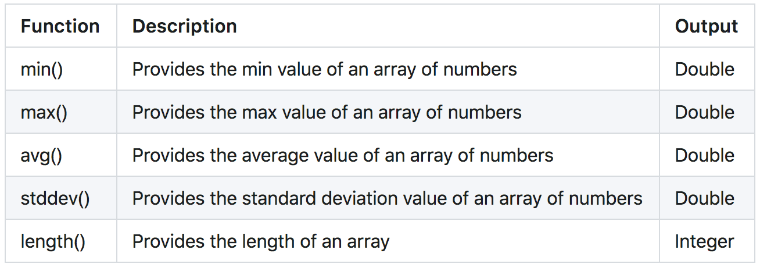
1. **jsonpath基础语法**



1. **jsonpath运算符**



1. jsonpath函数（Python中jsonpath模块暂不支持该语法）



因为jsonpath本身是js中的数据结构, 这些函数和运算本来是应该用到js中的, python中的jsonpath只是实现了一部分的功能, 一些jsonpath中的规则在python中并不支持.

#### jsonpath模块

**jsonpath数据解析：** Python中使用jsonpath模块时, 必须将json字符串转换为python内置对象, 如对象-->字典, 数组-->列表。 因为python中需要使用json.loads或load方法将json数据进行简单转换。

**jsonpath数据提取**

import json

from jsonpath import jsonpath

data = '''

{

"store": {

"book": [

{

"category": "reference",

"author": "Nigel Rees",

"title": "Sayings of the Century",

"price": 8.95

},

{

"category": "fiction",

"author": "Evelyn Waugh",

"title": "Sword of Honour",

"price": 12.99

},

{

"category": "fiction",

"author": "Herman Melville",

"title": "Moby Dick",

"isbn": "0-553-21311-3",

"price": 8.99

},

{

"category": "fiction",

"author": "J. R. R. Tolkien",

"title": "The Lord of the Rings",

"isbn": "0-395-19395-8",

"price": 22.99

}

],

"bicycle": {

"color": "red",

"price": 19.95

}

},

"expensive": 10

}'''

# 转换, 先用json.loads或load(如果是类文件对象)方法装换为python字典或列表

json\_data = json.loads(data)

# 选取所有的作者, jsonpath中接收的对象必须是python中的字典或列表.

# jsonpath的所有规则都是从根路径开始选取的. 所以都要以 $ 开头

print(jsonpath(json\_data,"$..author"))

'''

['Nigel Rees', 'Evelyn Waugh', 'Herman Melville', 'J. R. R. Tolkien']

'''

# 选择book价格小于10块的

print(jsonpath(json\_data,"$..book[?(@.price<10)]"))

'''

[{'category': 'reference', 'author': 'Nigel Rees', 'title': 'Sayings of the Century', 'price': 8.95}, {'category': 'fiction', 'author': 'Herman Melville', 'title': 'Moby Dick', 'isbn': '0-553-21311-3', 'price': 8.99}]

'''

# 选择book分类是fiction或news的

print(jsonpath(json\_data, "$..store.book[?(@.category in ['fiction', 'news'])]"))

'''

[{'category': 'fiction', 'author': 'Evelyn Waugh', 'title': 'Sword of Honour', 'price': 12.99}, {'category': 'fiction', 'author': 'Herman Melville', 'title': 'Moby Dick', 'isbn': '0-553-21311-3', 'price': 8.99}, {'category': 'fiction', 'author': 'J. R. R. Tolkien', 'title': 'The Lord of the Rings', 'isbn': '0-395-19395-8', 'price': 22.99}]

'''

# 选择book标题中包含'The'的

print(jsonpath(json\_data, "$..store.book[?('The' in @.title)]"))

'''

[{'category': 'fiction', 'author': 'J. R. R. Tolkien', 'title': 'The Lord of the Rings', 'isbn': '0-395-19395-8', 'price': 22.99}]

'''

# 注意：如果jsonpath规则写错、或查询不到结果, jsonpath返回的结果都是False

print(jsonpath(json\_data, "...")) # False

print(jsonpath(json\_data, "$..books")) # False

### 1.5. Python中运行JS

Python中执行JS代码, 通常两个库: js2py、pyexecjs. 都需要第3方的依赖互环境, 如nodejs. 在Dockerfile中已经实现了nodejs安装的代码. 想要把mysql, redis, nodejs等安装到一个容器中, 可以在docker的github主页, 找到各个环境的Dockerfile, 然后把各个Dockerfile整合到一个文件中就可以了. 注意各个环境使用的基础镜像要相同, 如都是基于stretch. stretch是基于ubuntu的镜像, Dockerfile也可以用于ubuntu中配置开发环境.

#### js2py

js2py是纯的Python实现的库, 用于在python中运行js代码。本质上是将js代码翻译成Python代码。

[源码地址](https://github.com/PiotrDabkowski/Js2Py)

**例1：js2py简单使用**

import js2py

js2py.eval\_js('console.log( "Hello World!" )')

# 'Hello World!'

func\_js = '''

function add(a, b) {

return a + b

}

'''

add = js2py.eval\_js(func\_js)

print(add(1,2))

# 3

**例2：js代码翻译**

import js2py

print(js2py.translate\_js("console.log('hello world')"))

# demo.js的内容

# console.log('hello world');

# 将js文件翻译为python脚本

# js2py.translate\_file('demo.js', 'demo.py')

如果python环境是在远程linux中的docker中的话, 需要手动把代码下载到本地. 在文件夹上点击右键 | Deployment | Download from ubuntu-linux, 才会看到生成的文件.

**例3：在js代码中使用python函数**

把js的代码翻译为python语句后, 想要把翻译后的python语句在js中运行, 就可以使用这个模块.

import js2py

# 将sum函数传入js的上下文执行环境中进行使用

print("sum: ", sum([1,2,3]))

context = js2py.EvalJs({'python\_sum': sum})

print("context.python\_sum: ", context.python\_sum)

js\_code = '''

python\_sum([1,2,3])

'''

print("js\_code运行结果: ", context.eval(js\_code))

'''运行结果

sum: 6

context.python\_sum: 'function sum() { [python code] }'

js\_code运行结果: 6

'''

例4：js代码中导入Python模块并使用

import js2py

# 在js代码中导入python模块并使用

# 使用pyimport语法

js\_code = '''

pyimport requests

console.log("导入成功");

var response = requests.get("http://www.baidu.com");

console.log(response.url);

console.log(response.content);

'''

# 执行js代码,

js2py.eval\_js(js\_code)

'''运行结果

True

'导入成功'

'http://www.baidu.com/'

PyObjectWrapper(b'<!DOCTYPE html>\r\n<!--STATUS OK--><html> ......

'''

例5：导入nodejs的安装的相关模块（前提系统环境中有node, 并有对应的node模块）

有的网站前端代码需要nodejs这个第三方模块才能正常运行. 如果使用纯js环境, 因为缺少nodejs库, 就无法在python中运行.

import js2py

# 前提：先安装好nodejs, 并使用 npm install crypto-js 安装

# crypto-js是node中数据加密、解密算法库

# 获取nodejs中环境中安装的crypto-js模块

CryptoJS = js2py.require('crypto-js')

data = [{'data1': "hello world"}, {'data2': 666}]

JSON = js2py.eval\_js('JSON') # 获取js中的JSON对象

ciphertext = CryptoJS.AES.encrypt(JSON.stringify(data), 'secret key 123') # 加密数据

print("ciphertext: ", ciphertext)

bytes = CryptoJS.AES.decrypt(ciphertext.toString(), 'secret key 123') # 解密成bytes类型的数据.

print("bytes: ", bytes)

decryptedData = JSON.parse(bytes.toString(CryptoJS.enc.Utf8)).to\_list() # 转换为json数据.

print("decryptedData: ", decryptedData)

'''运行结果

v8.11.4

npm WARN deprecated babel-preset-es2015@6.24.1: 🙌 Thanks for using Babel: we recommend using babel-preset-env now: please read babeljs.io/env to update!

...

...

...如果出现以上信息, 属于正常, 因为js2py模块中使用的一个库babel-preset-es2015, 被node遗弃了, 所以出现这个警告, 程序会自动联网进行升级. 可以去修改js2py模块的源码, 把使用bable-preset-es2015修改为babel-preset-env. 或者使用其它类似的python模块.

ciphertext: {'$super': {......

bytes: {'sigBytes': 39, 'words': [1534796388,......

decryptedData: [{'data1': 'hello world'}, {'data2': 666}]

'''

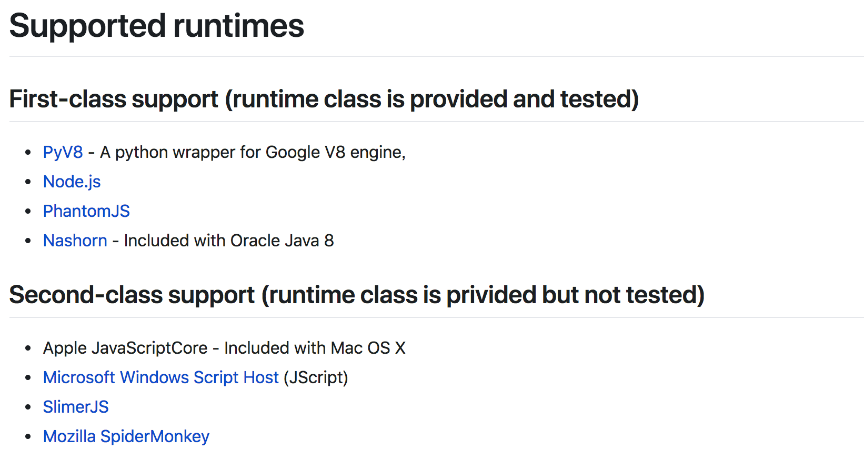
#### pyexecjs模块

pyexecjs同js2py一样, 用于在python中执行js代码。

但其原理和js2py不同, 因为相当于作为python与其他的js运行环境(如pyv8、nodejs等)的桥梁, js代码的运行是直接在对应的js运行环境中的, 而不是将js代码进行翻译, 因此理论上pyexecjs的执行速度是比js2py快的。

但是, 遗憾的是, pyexecjs模块的作者已经不维护该模块了, 具体原因[查看](https://gist.github.com/doloopwhile/8c6ec7dd4703e8a44e559411cb2ea221)。尽管如此, 至少当前版本已经能满足我们的大致需求。

具体运行环境:



虽然有那么多支持的运行环境, 但这里只推荐使用node环境。因为node可以安装很多第三方模块, 帮助解决问题, 同时有的网站前端代码本身就借助了node的一些第三方模块, 因此node是最合适的选择。

模块安装：

pip install pyexecjs

运行环境安装, 即node。

如果当前安装了多个运行环境, pyexecjs默认会自动选择最优的运行环境。 但也可以用过手动指定：

**例1：运行环境获取**

import execjs

# execjs.get() 从上面支持的环境中自动选择一个环境, 如果上面支持的环境都没有安装, 就会抛出异常

default\_runtime = execjs.get()

print(default\_runtime) # ExternalRuntime(Node.js (V8))

print(default\_runtime.name) # Node.js (V8)

import execjs.runtime\_names

# 也可以显示指定名称

node = execjs.get(execjs.runtime\_names.Node)

print(node) # ExternalRuntime(Node.js (V8))

**例2：execjs简单使用**

import execjs

# 简单使用：直接调用方法/函数

ret = execjs.eval("'red yellow blue'.split(' ')")

print(ret)

# 由于现在console命令是node的环境下运行, node的进程和python进程是不同的进程, 因此该语句不会出现在python的终端上. 而上面的js2py模块则是把js代码翻译成python代码再执行的, 所以能在python终端上打印出信息. 如果使用pyexecjs模块时想要在终端中输出一些内容, 可以把execjs.eval执行的结果返回过来, 然后再在python终端中执行并显示.

execjs.eval("console.log(123)")

'''运行结果

['red', 'yellow', 'blue']

'''

**例3：js代码预编译**

import execjs

# 复杂使用：需要先对js代码进行预编译

js\_code = '''

function f(temp) {

console.log(temp);

return "Hello World!";

};

'''

# 首先必须要编译代码

js\_context = execjs.compile(js\_code)

# 利用call方法调用对应的方法, 要调用f 函数, 传入test参数. 因为console.log是在node的进程中打印的, 所以python中不会显示出内容.

ret2 = js\_context.call("f", "test")

print(ret2)

'''运行结果

Hello World!

'''

#### 总结

两个模块各有各自的特点, 大家应该都应该掌握它们的使用方法, 如果必要, 互相搭配使用可能效果会更好。

## 2. 爬虫数据清洗

数据清洗：简单的说, 就是对数据进行审查和校验。

数据清洗的目的主要是解决数据质量问题, 删除重复信息、纠正存在的错误, 从而保证数据一致性。

根据不同的业务场景有不同的需要解决的问题, 这里以爬虫主要业务为例, 数据清洗大致要解决的问题：

- 数据一致性问题

- 数据一致性是指所获取的数据, 从多个维度考量数据的取值是否合乎预期要求。如数据的单位是否统一、数据取值标准是否统一、数据分类标准/分类依据是否统一等等. 多个网站电商数据, 分类的名称可能不同, 日期格式不同, 价格格式不同. 在入库之前要对数据进行统一. 相当于把多来源的数据变成相同来源的数据.

- 数据完整性问题

- 数据完整性是指数据信息不完整。如图书数据中缺少出版社、出版日期等；人类数据中缺少性别、年龄等。需要对缺失的数据进行补全. 如果数据不敏感, 直接手动补全即可, 如果数据敏感, 想办法从其它来源进行补全.

- 数据唯一性问题

- 数据唯一性是指数据的重复性过滤问题, 重复数据, 只应保留一条。

- 数据准确性问题

- 数据准确性是指根据数据的准确性原则, 判断数据是否准确的问题。如数据有误(格式不对、或超过当前时间)、无效数据、冗余数据无关数据过多等等

## 3. 爬虫数据存储

当数据解析提取完毕, 并经过数据清洗以后, 就可以对数据进行存储。

最简单的形式就是存成文件, 如存成txt、csv、json等格式。另外大部分情况下, 数据为了更好的使用和管理, 其实往往会选择将数据存储到数据库中, 如MySQL、PostgreSQL、MongoDB等。

但如果是一些媒体类型或特殊文本类型数据, 如图片、视频、PDF等, 通常都是存成文件, 然后将文件的名称、路径、大小等信息存入数据库中, 便于后期的使用以及维护和管理。

### 3.1. 文件存储

无论哪种形式的文件存储, 其实其本质都是一个文本的写入操作, 基本操作都是：

file = open(文件名称, 打开模式)

file.write(数据)

file.close()

区别在于不同类型数据写入时, 文件的打开模式不同、存储文件的结构不同。

#### 文件打开模式

* r：以只读模式打开文件。默认模式。
* r+：以读写模式打开文件
* rb：以二进制只读模式打开文件
* rb+：以二进制读写模式打开文件
* w：以写入模式打开文件。文件存在则覆盖, 否则创建
* w+：以读写模式打开文件。文件存在则覆盖, 否则创建
* wb：以二进制写入模式打开文件。文件存在则覆盖, 否则创建
* wb+：以二进制读写模式打开文件。文件存在则覆盖, 否则创建
* a：以追加模式打开文件。文件存在则追加写入, 否则创建
* a+：以读写追加模式打开文件。文件存在则追加写入, 否则创建
* ab：以二进制追加模式打开文件。文件存在则追加写入, 否则创建
* ab+：以二进制读写追加模式打开文件。文件存在则追加写入, 否则创建

记忆点：

* r\w\a的含义
* +号的意义
* b的意义

#### 文件结构类型

1. **CSV文本存储**

CSV文本存储是一种非常常用的存储方式：

- 结构简单：列行数据分别以逗号和换行符来分割

- 使用便捷：可直接被pandas、numpy等数据处理模块或也可直接导入Excel中进行处理

第一列,第二列

col1-data1,col2-data1

col1-data2,col2-data2

col1-data3,col2-data3

import csv # Python内置模块

# col1, col2分别为第1列和第2列的数据

col1 = ["col1-data1","col1-data2","col1-data3"]

col2 = ["col2-data1","col2-data2","col2-data3"]

with open("test.csv","w") as file:

# 定义writer对象

writer = csv.writer(file)

# 先在第一行写入列名称

writer.writerow(["第一列","第二列"])

# 利用writerows一次写入多行数据, 因为数据是按行写入的, 所以要使用zip函数转换为行的结构.

rows = list(zip(col1, col2))

'''rows:

[('col1-data1', 'col2-data1'),

('col1-data2', 'col2-data2'),

('col1-data3', 'col2-data3')]

'''

writer.writerows(rows)

# 查看是否写入成功

import os

os.system("cat test.csv")

'''运行结果

第一列,第二列

col1-data1,col2-data1

col1-data2,col2-data2

col1-data3,col2-data3

'''

1. **JSON文本存储**

json数据写入：

'''利用json模块将python列表或字典装换为json字符串, 再写入'''

import json

json\_data = [

{ "firstName":"Bill" , "lastName":"Gates" },

{ "firstName":"George" , "lastName":"Bush" },

{ "firstName":"Thomas" , "lastName": "Carter" }

]

with open("test.json", "w") as file:

file.write(json.dumps(json\_data, indent=2)) # indent控制json字符串的缩进, 使得结构更美观

file.close()

import os

os.system("cat test.json")

'''运行结果

[

{

"firstName": "Bill",

"lastName": "Gates"

},

{

"firstName": "George",

"lastName": "Bush"

},

{

"firstName": "Thomas",

"lastName": "Carter"

}

]

'''

1. **媒体文件存储**

媒体文件数据（如图片、视频等）存储需注意：

* + 其一：媒体文件数据, 都是二进制类型数据, 因此打开文件式, 必须以二进制方式打开；
  + 其二：如果文件数据较大的情况下, 如视频数据, 可能会考虑分批写入, 那么此时注意必须使用追加模式写入数据；
  + 其三：如果需要考虑维护文件的路径、大小等数据, 那么需要考虑将这些数据单独进行存储并维护。

### 3.2. 数据库存储

#### Docker中使用数据库

一般常用的Docker镜像, 在[docker hub](https://github.com/docker-library/docs)上都有维护, 下载对应的镜像下来, 启动即可。

使用mysql 5.7镜像来构建mysql容器, 并且进行相关配置

Image ID: mysql:5.7

Container name: mysql

Environment variables环境变量: MYSQL\_ROOT\_PASSWORD=password; MYSQL\_DATABASE=test\_db

Command line options: --network host, 即启动时使用host网络模式.

postgres数据库, 使用postgres:10来构建容器

Image ID: postgres:10

Container name: postgres

Environment variables环境变量: POSTGRES\_PASSWORD=password; POSTGRES\_DB=test\_db

Command line options: --network host, 即启动时使用host网络模式.

mongodb数据库不需要设置密码, 也不用指定使用的数据库, 其它的配置与mysql类似.

注意：如果使用host模式启动, 直接使用主机的网络进行通信, 如果需要在容器外部进行访问, 那么通常需要在防火墙开放数据对应的端口, 否则很可能造成无法连接. mysql打开3306, mongodb打开27017, postgres打开5432端口

firewall-cmd --list-ports

firewall-cmd --add-port 3306/tcp --zone=public --permanet

firewall-cmd --add-port 27017/tcp --zone=public --permanet

firewall-cmd --add-port 5432/tcp --zone=public --permanet

service firewalld restart

可以使用navicat来对mysql和postgres连接进行测试, 使用mongodb compass对mongodb数据库进行测试, 因为以上3个数据库都是使用的host网络模式, 所以连接时的ip地址都使用linux服务器的地址.

#### 3.2.1. MySQL与PostgreSQL对比

#### 数据库定位：

MySQL：全世界最流行的关系型数据库, 稳定

PostgreSQL：全世界最先进的关系型数据库, 性能更好, 功能最先进

#### 数据库功能比较

|  |  |  |
| --- | --- | --- |
| 特性 | MySQL 8 | PostgreSQL 10 |
| 查询 & 分析 |  |  |
| 公用表表达式 (CTEs) | ✔ New | ✔ |
| 窗口函数 | ✔ New | ✔ |
| 数据类型 |  |  |
| JSON 支持 | ✔ Improved | ✔ |
| GIS / SRS | ✔ Improved | ✔ |
| 全文检索 | ✔ | ✔ |
| 可扩展性 |  |  |
| 逻辑复制 | ✔ | ✔ New |
| 半同步复制 | ✔ | ✔ New |
| 声明式分区 | ✔ | ✔ New |

MySQL 8 和 PostgreSQL 10最受关注的功能比较, 从功能上, 二者已经几乎没有太大差别

#### 问题：如何选择一个而不选择另外一个？

**MySQL相较于PostgreSQL的优势：**

1. 生态系统。

MySQL比PostgreSQL更流行。 流行对于一个商业软件来说, 也是一个很重要的指标, 流行意味着更多的用户, 意味着经受了更多的考验, 意味着更好的商业支持、意味着更多、更完善的文档资料。易用, 很容易安装。第三方工具, 包括可视化工具更加完善, 同时让用户能够很容易入门。

1. 回滚实现更优

innodb的基于回滚段实现的MVCC机制, 相对PG新老数据一起存放的基于XID的MVCC机制, 是占优的。新老数据一起存放, 需要定时触发VACUUM, 会带来多余的IO和数据库对象加锁开销, 引起数据库整体的并发能力下降。而且VACUUM清理不及时, 还可能会引发数据膨胀。

1. 在Windows上运行更可靠

与PostgreSQL相比, MySQL更适宜在Windows环境下运行。MySQL作为一个本地的Windows应用程序运行(在 NT/Win2000/WinXP下, 是一个服务), 而PostgreSQL是运行在Cygwin模拟环境下。PostgreSQL在Windows下运行没有MySQL稳定。

1. 线程模式相比进程模式的优势

MySQL使用了线程, 而PostgreSQL使用的是进程。在不同线程之间的环境转换和访问公用的存储区域显然要比在不同的进程之间要快得多。 线程模式对资源消耗比较少, 但进程模式对多CPU利用率比较高。

1. 权限设置上更加完善

MySQL在权限系统上比PostgreSQL某些方面更为完善。PostgreSQL只支持对于每一个用户在一个数据库上或一个数据表上的 INSERT、SELECT和UPDATE/DELETE的授权, 而MySQL允许你定义一整套的不同的数据级、表级和列级的权限。

**PostgreSQL相较于MySQL的优势**

1. 集群支持更好

mysql的集群化工具较少, 且使用效果一般。而PostgreSQL有丰富的开源cluster软件支持。

1. 事务隔离做的更好
2. 没有字符串长度限制, 且对字符支持更好一些

MySQL的各种text字段有不同的限制, 要手动区分 small text, middle text, large text；而PostgreSQL 没有这个限制, text 能支持各种大小 MySQL里需要utf8mb4才能显示emoji, PostgreSQL直接支持

1. 提供更多的高级功能

对表连接支持较完整, 支持跨库数据表的连接 存储方式支持更大的数据量 时间精度更高 序列支持更好 地址数据的良好支持 函数化sql编程 支持数组、json数据的存储 可以快速构建REST API 外部数据源支持等等

总的来说：

求好：PostgreSQL在很多细节上是优于MySQL的, 也就是说在数据层面PostgreSQL相较于MySQL会显得更加可靠、信赖, 因此当你绝对需要可靠性和数据完整性的时候、需要数据库执行定制程序、处理的是地理数据等特殊需求时你应该选择PostgreSQL。

求稳：但是由于MySQL的定位是成为最流行的数据库, MySQL运行时稳定性优于PostgreSQL, 并且它强大的生态系统的支持使得MySQL更适合绝大多数的应用场景。而且单论数据库最基本的大批量数据的查询、插入、更新、删除, 其实mysql是相对会更占优势一点。

#### 3.2.2. MySQL与MongoDB对比

提及关系型与非关系型数据库, 其实最容易联想到的, 应该就是MySQL与MongoDB

#### 问题：二者在使用时又该如何去选择呢？

1. 相比较MySQL, MongoDB数据库更适合那些读作业较重的任务模型。

MongoDB能充分利用机器的内存资源。如果机器的内存资源丰富的话, MongoDB的查询效率会快很多。

1. 相比MongoDB, MySQL数据库更适合插入、更新为主的任务模型, 其效率相比于MongoDB优势非常明显。
2. 在带 "\_id" 插入数据的时候, MongoDB的插入效率其实并不高。

如果想充分利用MongoDB性能的话, 推荐采取不带 "\_id" 的插入方式, 然后对相关字段作索引来查询。

1. MongoDB适合那些对数据库具体数据格式不明确或者数据库数据格式经常变化的需求模型, 而且对开发者十分友好。

MongoDB用之前不用想太多；而MySQL使用之前, 必须提前建库建表, 并明确数据的具体格式, 约束较大。

1. 稳定性方面, MongoDB不如MySQL, MySQL是一种非常稳定的数据库, 无论在指定主键还是在不指定主键插入的情况下, 其效率都差不了太多, 其稳定性可以说是毋庸置疑的。
2. 事务支持方面, MySQL占绝对优势, MongoDB对事务关系支持薄弱。

事务可以说是所有NoSQL数据库共同的缺陷, 不过NoSQL并不是为了事务关系而设计的, 具体应用还是很需求。

#### 3.2.3. 关系型数据库使用

关系型数据种类很多, 这里主要介绍MySQL、PostgreSQL的使用. cs的架构, 使用数据库主要是client端的使用

#### MySQL: PyMySQL

Python中使用MySQL建议使用的客户端库是PyMySQL, 跨平台性、兼容性更好。

[PyMySQL的github地址](https://github.com/PyMySQL/PyMySQL)

[PyMySQL的文档地址](https://pymysql.readthedocs.io/en/latest/)

**使用**

使用介绍

import pymysql.cursors

class PyMysqlBaseUsage(object):

def \_\_init\_\_(

self,

host='localhost',

port=3306,

user='user',

password='passwd',

db='db',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor

):

# 使用pymysql连接到数据库, 返回一个连接对象, 必须要指定host, port, user, password, db这5个参数. 又由于mysql对数据的编码方式非常敏感, 所以还需要指定数据库的编码, 使用utf8mb4, 可以支持类似于表情的特殊编码. 还需要指定cursorclass, 如果不指定, fetchall方法中返回的是一个嵌套元组的列表, 如果指定了cursorclass=pymysql.cursors.DictCursor, 则以嵌套字典的列表方式返回fetchall的查询结果.

self.connection = pymysql.connect(

host=host,

port=port,

user=user,

password=password,

db=db,

charset=charset,

cursorclass=cursorclass)

print("已建立数据库连接")

# 使用mysql之前必须要先创建数据表, 传入创建表的sql语句, 使用连接对象的cursort方法执行sql语句, 创建数据表

def create\_table(self, sql):

with self.connection.cursor() as cursor:

cursor.execute(sql)

self.connection.commit()

# 与创建数据表的语句类似, 只是这里需要传入的是删除表的sql语句

def drop\_table(self, sql):

with self.connection.cursor() as cursor:

cursor.execute(sql)

self.connection.commit()

def insert(self, sql, params=None):

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

self.connection.commit()

def select\_all(self, sql, params=None):

results = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

results = cursor.fetchall()

return results

def select\_many(self, sql, params=None, size=1):

results = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

results = cursor.fetchmany(size)

return results

def select\_one(self, sql, params=None):

result = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

result = cursor.fetchone()

return result

# 对象在删除时会断开连接. 必须要执行, 否则数据库服务端的连接会一直存在.

def \_\_del\_\_(self):

self.connection.close()

print("已断开数据库连接")

if \_\_name\_\_ == '\_\_main\_\_':

obj = PyMysqlBaseUsage(host="数据库IP", user="root", password="password", db="test\_db")

# 建表语句, 为了与变量和字符串进行区分, mysql中的字段推荐使用``来括起来.

# create\_sql = '''

# CREATE TABLE `test\_table` (

# `id` int(11) NOT NULL AUTO\_INCREMENT,

# `colume1` varchar(255) NOT NULL,

# `colume2` varchar(255) NOT NULL,

# PRIMARY KEY (`id`)

# ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin

# AUTO\_INCREMENT=1;'''

# obj.create\_table(create\_sql)

# 删表语句

# drop\_sql = '''DROP TABLE test\_table'''

# obj.drop\_table(drop\_sql)

# 插入数据

# insert\_sql = "INSERT INTO `test\_table` (`colume1`, `colume2`) VALUES (%s, %s)"

# obj.insert(insert\_sql, ("data1", "data2"))

# 查询数据

# select\_sql = "SELECT \* FROM `test\_table`"

# results = obj.select\_all(select\_sql)

# print("all: ", results)

#

# results = obj.select\_many(select\_sql, size=10)

# print("many: ", results)

#

# result = obj.select\_one(select\_sql)

# print("one: ", result)

#### PostgreSQL: psycopg2

[psycopg2的github地址](https://github.com/psycopg/psycopg2)

[psycopg2的文档地址](http://initd.org/psycopg/docs/)

使用介绍：postgresql与mysql使用时最大的区别是sql语句不同, python客户端的api使用方法几乎完全相同.

import psycopg2

# conn = psycopg2.connect("dbname=test\_db user=postgres password=password host=10.211.55.3")

class Psycopg2BaseUsage(object):

def \_\_init\_\_(self,

host='localhost',

port=5432,

user='user',

password='passwd',

dbname='dbname'):

# 连接到数据库, postgres对编码的支持已经很完善了, 不需要再设定编码方式了.

self.connection = psycopg2.connect(

host=host,

port=port,

user=user,

password=password,

dbname=dbname)

print("已建立数据库连接")

def create\_table(self, sql):

with self.connection.cursor() as cursor:

cursor.execute(sql)

self.connection.commit()

def drop\_table(self, sql):

with self.connection.cursor() as cursor:

cursor.execute(sql)

self.connection.commit()

def insert(self, sql, params=None):

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

self.connection.commit()

def select\_all(self, sql, params=None):

results = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

results = cursor.fetchall()

return results

def select\_many(self, sql, params=None, size=1):

results = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

results = cursor.fetchmany(size)

return results

def select\_one(self, sql, params=None):

result = None

with self.connection.cursor() as cursor:

cursor.execute(sql, params)

result = cursor.fetchone()

return result

def \_\_del\_\_(self):

self.connection.close()

print("已断开数据库连接")

if \_\_name\_\_ == '\_\_main\_\_':

obj = Psycopg2BaseUsage(host="数据库IP", user="postgres", password="password", dbname="test\_db")

# 建表语句, postgre中不支持``反引号, 不用像mysql中一样把列括起来.

# create\_sql = '''

# CREATE TABLE test\_table (

# id serial NOT NULL,

# colume1 varchar(255) NOT NULL,

# colume2 varchar(255) NOT NULL,

# PRIMARY KEY (id)

# );'''

# obj.create\_table(create\_sql)

# 删表语句

# drop\_sql = '''DROP TABLE test\_table'''

# obj.drop\_table(drop\_sql)

# 插入数据

# insert\_sql = "INSERT INTO test\_table (colume1, colume2) VALUES (%s, %s)"

# obj.insert(insert\_sql, ("data1", "data2"))

# 查询数据

# select\_sql = "SELECT \* FROM test\_table"

# results = obj.select\_all(select\_sql)

# print("all: ", results)

#

# results = obj.select\_many(select\_sql, size=10)

# print("many: ", results)

#

# result = obj.select\_one(select\_sql)

# print("one: ", result)

执行时可能会出现下面的警告信息:

UserWarning: The psycopg2 wheel package will be renamed from release 2.8; in order to keep installing from binary please use "pip install psycopg2-binary" instead. For details see: <http://initd.org/psycopg/docs/install.html#binary-install-from-pypi>

#### ORM

sqlalchemy和django-orm使用, 参见案例

#### 3.2.4. 非关系型数据库使用

非关系型数据库也有多种, 这里以MongoDB为例

#### pymongo

[pymongo的github地址](https://github.com/mongodb/mongo-python-driver)

[pymongo的文档地址](http://api.mongodb.com/python/)

**使用：**

from pymongo import MongoClient

client = MongoClient('10.211.55.3', 27017)

# client = MongoClient('mongodb://IP:27017/')

# 获取数据库, 如果没有, 会自动创建

db = client.test\_db

# db = client['test-db']

# 获取一个collection, mongodb的collection类似mysql的表

collection = db.test\_collection

# collection = db['test-collection']

# 插入数据

# mongodb插入的数据是一个json风格的的文档数据, 如

data = {

"title": "解忧杂货店",

"author": "[日]东野圭吾",

"publisher": "南海出版公司",

"info": {

"producer": "新经典文化",

"original\_title": "ナミヤ雑貨店の奇蹟",

"translator": "李盈春",

"publish\_time": "2014/05/01",

"page\_number": 291,

"price": 39.5,

"pack": "精装",

"series": "新经典文库·东野圭吾作品"

},

"isbn": "9787544270878",

"rating\_num": 8.6,

"another\_data": [1,2,3,"new\_data"]

}

# 插入数据. 在名为book的collection集合中插入一条数据

obj = db.book.insert\_one(data)

print(obj.inserted\_id)

# 查询数据

ret = db.book.find\_one({"title":"解忧杂货店"})

print(ret)

# 更新数据, 使用聚合操作, 把title的字段进行修改

ret = db.book.update({"title":"解忧杂货店"}, {"$set":{"title":"《解忧杂货店》"}})

print(ret)

# 删除数据

# ret = db.book.delete\_one({"rating\_num":8.6})

# ret = db.book.delete\_many({"rating\_num":8.6})

# print(ret.deleted\_count)

#### mongoengine

mongodb的orm. 使用ORM, 就可以直接使用python的语句对数据库进行增删改查的操作, 不用再使用原生的sql或mongo语句.

[github地址](https://github.com/MongoEngine/mongoengine)

[文档地址](http://docs.mongoengine.org/)

具体使用, 参见案例

### 3.3. 案例：豆瓣图书信息抓取

#### 需求介绍

**一、图书大小**[**标签信息**](https://book.douban.com/tag/?view=type&icn=index-sorttags-hot)**抓取, 需存储:**

<https://book.douban.com/tag/?view=type&icn=index-sorttags-hot>

抓取所有的大标签和小标签.

**二、标签列表页抓取, 需要翻页, 不需存储**

只需要从标签列表页获取图片的url地址即可, 具体信息在图书详情页中提取.

**三、图书详情页抓取, 需存储**

包括: 图书名称, 作者, 出版社, 出品方, 原作名, 译者, 出版年, 面粉数, 定价, 装帧, 丛书, ISBN, 评分, 评分人数. 内容简介, 作者简介.



最终数据包括：大标签数据、小标签数据、书籍详细数据

#### 需求分析

从标签页抓取所有标签, 提取大标签, 小标签的名称和url地址.

https://book.douban.com/tag/?view=type&icn=index-sorttags-hot

从小标签的url地址进入到小标签的详情页中, 分页提取所有图书的url信息.

https://book.douban.com/tag/小说

每个小标签的url地址中, 前面的部分都是不变的, 只有小标签名变化, 所以只需要提取出小标签名即可, 不用提取出小标签的url, 只要有了小标签名, 就可以构建出小标签列表页的url地址.

注意需要进行翻页, 但每个小标签最多只会显示1000条左右的信息, 对应50页的信息, 50页以后的页码是无数据的, 所以还要根据页面的结构进行判断, 如果无数据, 就不再提取下一页的内容.

每页数据20条, 第1页start=0, 第2页start=20

<https://book.douban.com/tag/小说?start=20&type=T>

每次提取完一页的信息后, 给start递加20, 直到页面中没有出现图书结构的数据, 或者出现 "没有找到符合条件的图书" 的提示.

不能按照网站中给出的分页结构来提取数据. 一些小标签的列表页, 如小说, "后页" 可用时, 得到的可能是空白页面. 一些小标签的列表页, 如程序, "后页" 不可用时, 通过修改start的值, 还是可以提取到内容的.

从图书的url地址进入到图书的详情页中, 提取出图书的详情信息.

https://book.douban.com/subject/25862578/

提取图书的评价信息, 包括评价的时间, 评价人, 评分.

https://book.douban.com/subject/25862578/collections

提取图书的所有短评信息. 包括短评人, 短评时间, 评分, 短评内容, 有用数量.

https://book.douban.com/subject/25862578/comments/

#### 3.3.1. 数据解析和提取

需要写3个页面信息抓取的接口

在项目根目录中新建douban\_book的python package

Step1. 标签信息抓取

新建douban\_book/spider01\_get\_tags.py

import requests

import lxml.etree

class DoubanBook(object):

# 标签列表页

tag\_list\_url = "https://book.douban.com/tag/?view=type&icn=index-sorttags-hot"

def get\_tags(self):

'''获取tag信息'''

response = requests.get(self.tag\_list\_url)

html = lxml.etree.HTML(response.content)

# 取到每一个大标签, 对大标签进行遍历, 获取其中的所有小标签

for div in html.xpath("//div[@class='article']/div[2]/div"):

big\_tag = div.xpath("./a/@name")

small\_tags = div.xpath("./table//td/a/text()")

print(big\_tag)

print(small\_tags)

if \_\_name\_\_ == '\_\_main\_\_':

spider = DoubanBook()

spider.get\_tags()

Step2. 标签列表页抓取

新建douban\_book/spider02\_get\_one\_page.py

获取一页

import requests

import lxml.etree

class DoubanBook(object):

# 构建初始url. 书列表, 传入tag名称和offset得到完整的url地址.

book\_list\_from\_tag\_url = "https://book.douban.com/tag/{tag\_name}?start={offset}&type=T"

def get\_one\_page(self, tag\_name, offset):

'''获取书籍详情页地址'''

response = requests.get(self.book\_list\_from\_tag\_url.format(tag\_name=tag\_name, offset=offset))

html = lxml.etree.HTML(response.content)

book\_url\_list = html.xpath("//ul[@class='subject-list']/li/div[@class='pic']/a/@href")

return book\_url\_list

if \_\_name\_\_ == '\_\_main\_\_':

spider = DoubanBook()

url\_list = spider.get\_one\_page("小说", 0)

url\_list2 = spider.get\_one\_page("小说", 980)

url\_list3 = spider.get\_one\_page("小说", 1000) # 超过1000没有数据

print(url\_list)

print(url\_list2)

print(url\_list3)

获取多页

新建douban\_book/spider03\_get\_all\_pages.py

import requests

import lxml.etree

class DoubanBook(object):

def \_\_init\_\_(self, end\_offset=980):

# 偏移量超过980, 很可能无法获取数据

self.end\_offset = end\_offset

# 书列表, 根据tag名称和offset 20

book\_list\_from\_tag\_url = "https://book.douban.com/tag/{tag\_name}?start={offset}&type=T"

def get\_one\_page(self, tag\_name, offset):

'''获取书籍主页地址'''

response = requests.get(self.book\_list\_from\_tag\_url.format(tag\_name=tag\_name, offset=offset))

html = lxml.etree.HTML(response.content)

book\_url\_list = html.xpath("//ul[@class='subject-list']/li/div[@class='pic']/a/@href")

# 判断是否还有后页

next = html.xpath("//span[@class='next']/a/@href")

# 如果还有下一页, 那么递归

if next and offset < self.end\_offset:

return True, book\_url\_list

else:

return False, book\_url\_list

def get\_all\_pages(self, tag\_name, offset=0):

'''翻页'''

# 最后一页判断条件：

# 1. 判断页面是否还有下一页

# 2. 判断offset是否超过1000（目前豆瓣超过50页, 拿不到数据）

has\_next, url\_list = self.get\_one\_page(tag\_name, offset)

yield int((offset/20)+1), url\_list

if has\_next: # 如果有下一页, 递归执行

# 为什么使用yield from. 如果不使用yield from, 要递归调用, 需要使用self.get\_all\_pages(), 但由于get\_all\_pages中使用了yield, 即返回的是一个生成器, 想要从生成器中获取数据, 就要对其进行循环遍历. 取出来url\_list, 并再次使用yield来返回数据, 使用yield from, 就相当于下面的for循环和yield语句.

# for has\_next, url\_list in self.get\_all\_pages(tag\_name, offset+20):

# yield int((offset/20)+1), url\_list

yield from self.get\_all\_pages(tag\_name, offset+20)

if \_\_name\_\_ == '\_\_main\_\_':

# 测试时可以只获取前几页的数据, 只需要设置一个小的offset值就可以了.

# spider = DoubanBook(80) # 80代表, 只获取5页

# for page, url\_list in spider.get\_all\_pages("小说"):

# print(page, url\_list)

# 如果不给出offset的值, 则使用默认的980

spider = DoubanBook()

for page, url\_list in spider.get\_all\_pages("UCD"):

print(page, url\_list)

Python：对于yield与yield from 的理解

<https://www.jianshu.com/p/2332279fafdb>

Python中的yield from语法

<https://blog.csdn.net/u014683535/article/details/51865880>

简述python的yield和yield from

<https://juejin.im/post/59f75a3d6fb9a045023aec42>

理解Python协程:从yield/send到yield from再到async/await

<https://blog.csdn.net/soonfly/article/details/78361819>

Step3. 获取详情页数据

新建douban\_book/spider04\_get\_book\_detail.py

import requests

import lxml.etree

import re

class DoubanBook(object):

# book info中的所有字段, 对比不同的书籍得出所有可能存在的字段

book\_info\_details = ["作者", "出版社", "出品方", "原作名", "译者", "出版年", "页数", "定价", "装帧", "丛书", "ISBN", "副标题"]

def get\_book\_detail(self, book\_url):

'''获取书籍详情

'''

response = requests.get(book\_url)

html = lxml.etree.HTML(response.content)

# info内的标签没有规则, xpath很难定位

info\_text\_list = html.xpath("//div[@id='info']//text()")

# 使用正则来处理

info\_str = "".join(info\_text\_list) # 全部拼接为字符串

info\_str = re.sub(r"\s", "", info\_str) # 剔除空白字符

info\_str = re.sub(r"\xa0", "", info\_str) # 剔除无用字符

print(info\_str)

# 作者:[日]东野圭吾出版社:南海出版公司出品方:新经典文化原作名:ナミヤ雑貨店の奇蹟译者:李盈春出版年:2014-5页数:291定价:39.50元装帧:精装丛书:新经典文库·东野圭吾作品ISBN:9787544270878

# 筛选出info中每一个字段的值

for filed in self.book\_info\_details:

# 挨个筛选出字段, 如 '作者:[日]东野圭吾出版社'

ret = re.search("%s:(.\*?):" % filed, info\_str + ":")

if ret:

# 剔除掉 '作者:[日]东野圭吾出版社'末尾的'出版社', 其他同理

value = re.sub("(%s)$" % ("|".join(self.book\_info\_details)), "", ret.group(1))

print(filed, ":", value)

# 如果字段不存在, 就返回None

else:

print(filed, ":", None)

# 选取标题

book\_title = html.xpath("//div[@id='wrapper']/h1/span/text()")[0].strip()

print("书名: ", book\_title)

# 评分：

rating\_num = html.xpath("//div[@id='interest\_sectl']//strong/text()")[0].strip()

print("评分: ", rating\_num)

# 内容简介

book\_summary = html.xpath("//div[@class='related\_info']/div[@class='indent'][1]//div[@class='intro']//text()")

print(book\_summary)

# 作者简介

# 注意indent后面那个神奇的空格, 加了空格后就只能取到1个元素了, 所以不用再选择第几个.

author\_summary = html.xpath("//div[@class='related\_info']/div[@class='indent ']//div[@class='intro']//text()")

print(author\_summary)

if \_\_name\_\_ == '\_\_main\_\_':

book\_url = "https://book.douban.com/subject/25862578/"

spider = DoubanBook()

spider.get\_book\_detail(book\_url)

注意: 如果使用xpath插件在chrome中可以提取到内容, 但在代码中去无法得到内容, 那么就需要查看网页的源代码, 使用源代码来写xpath的规则.

在写xpath规则时, 一是要注意网页中经过渲染后的代码和网页的源码可能不同, xpath规则也可能不同. 二是页面的结构中一些细小的不同, 如class中多写了一个空格, 或者写错了一个单词, 都有可能提取不到内容.

Step3. 各个功能的整合

新建douban\_book/spider05\_full\_functions.py

#### 3.3.2. 数据清洗

1. 字段类型统一：
   * 标签数据：
     + 大标签(big\_tag):字符串
     + 小标签(small\_tag):字符串
   * 书籍详情页数据
     + 书名(title): 字符串. 从列表中取出来
     + 评分(rating\_num): 浮点数. 列表嵌套的字符串转换为浮点数
     + 内容简介(book\_summary): 字符串(LONG)
     + 作者简介(author\_summary): 字符串(LONG)
     + 作者(author): 字符串
     + 出版社(publisher): 字符串
     + 出品方(producer): 字符串
     + 原作名(original\_title): 字符串
     + 译者(translator): 字符串
     + 出版年(publish\_time):时间(YMD), 有年, 年-月, 年-月-日三种格式, 统一转换为YMD的形式
     + 页数(page\_number): 整型
     + 定价(price): 浮点数, 取出来字符串, 并去掉最后的元, 转换为浮点数
     + 装帧(pack): 字符串
     + 丛书(series): 字符串
     + ISBN(isbn)：数值/字符串, 不可重复
     + 副标题(subtitle): 字符串
2. 空数据替换
3. 重复数据判断
   * 书籍信息重复, 本例中可根据ISBN判断
   * 标签数据去重, 数据很少, 直接比对即可

字段整合

import requests

import lxml.etree

import re

class DoubanBook(object):

# 标签列表页

tag\_list\_url = "https://book.douban.com/tag/?view=type&icn=index-sorttags-hot"

def get\_tags(self):

'''获取tag信息'''

response = requests.get(self.tag\_list\_url)

html = lxml.etree.HTML(response.content)

for div in html.xpath("//div[@class='article']/div[2]/div"):

big\_tag = div.xpath("./a/@name")

small\_tags = div.xpath("./table//td/a/text()")

yield big\_tag, small\_tags

def \_\_init\_\_(self, end\_offset=980):

# 偏移量超过980, 很可能无法获取数据

self.end\_offset = end\_offset

# 书列表, 根据tag名称和offset 20

book\_list\_from\_tag\_url = "https://book.douban.com/tag/{tag\_name}?start={offset}&type=T"

def get\_one\_page(self, tag\_name, offset):

'''获取书籍主页地址'''

response = requests.get(self.book\_list\_from\_tag\_url.format(tag\_name=tag\_name, offset=offset))

html = lxml.etree.HTML(response.content)

book\_url\_list = html.xpath("//ul[@class='subject-list']/li/div[@class='pic']/a/@href")

# 判断是否还有后页

next = html.xpath("//span[@class='next']/a/@href")

# 如果还有下一页, 那么递归

if next and offset < self.end\_offset:

return True, book\_url\_list

else:

return False, book\_url\_list

def get\_all\_pages(self, tag\_name, offset=0):

'''翻页'''

# 最后一页判断条件：

# 1. 判断页面是否还有下一页

# 2. 判断offset是否超过1000（目前豆瓣超过50页, 拿不到数据）

has\_next, url\_list = self.get\_one\_page(tag\_name, offset)

yield int((offset/20)+1), url\_list

if has\_next: # 如果有下一页, 递归执行

yield from self.get\_all\_pages(tag\_name, offset+20)

# book info中的所有字段, 使用英文名称为保存时的字段名

book\_info\_details = [("作者", "author"),

("出版社", "publisher"),

("出品方", "producer"),

("原作名", "original\_title"),

("译者", "translator"),

("出版年", "publish\_time"),

("页数", "page\_number"),

("定价", "price"),

("装帧", "pack"),

("丛书", "series"),

("ISBN", "isbn"),

("副标题","subtitle")]

def get\_book\_detail(self, book\_url):

'''获取书籍详情

'''

response = requests.get(book\_url)

html = lxml.etree.HTML(response.content)

# info内的标签没有规则, xpath很难定位

info\_text\_list = html.xpath("//div[@id='info']//text()")

# 使用正则来处理

info\_str = "".join(info\_text\_list) # 全部拼接为字符串

info\_str = re.sub(r"\s", "", info\_str) # 剔除空白字符

info\_str = re.sub(r"\xa0", "", info\_str) # 剔除无用字符

detail = {}

# 筛选出info中每一个字段的值

for filed, filed\_name in self.book\_info\_details:

# 挨个筛选出字段, 如 '作者:[日]东野圭吾出版社'

ret = re.search("%s:(.\*?):" % filed, info\_str + ":")

if ret:

# 剔除掉 '作者:[日]东野圭吾出版社'末尾的'出版社', 其他同理

value = re.sub("(作者|出版社|出品方|原作名|译者|出版年|页数|定价|装帧|丛书|ISBN|副标题)$", "", ret.group(1))

detail[filed\_name] = value

else:

detail[filed\_name] = None

# 选取标题

book\_title = html.xpath("//div[@id='wrapper']/h1/span/text()")

detail["title"] = book\_title

# 评分：

rating\_num = html.xpath("//div[@id='interest\_sectl']//strong/text()")

detail["rating\_num"] = rating\_num

# 内容简介

book\_summary = html.xpath("//div[@class='related\_info']/div[@class='indent'][1]//div[@class='intro']//text()")

detail["book\_summary"] = book\_summary

# 作者简介

# 注意indent后面那个神奇的空格

author\_summary = html.xpath("//div[@class='related\_info']/div[@class='indent ']//div[@class='intro']//text()")

detail["author\_summary"] = author\_summary

return detail

if \_\_name\_\_ == '\_\_main\_\_':

book\_url = "https://book.douban.com/subject/25862578/"

spider = DoubanBook()

detail = spider.get\_book\_detail(book\_url)

import json

print(json.dumps(detail, indent=2))

清洗数据

......

def clean\_detail(self, data):

'''

{

"author": str, must not null

"publisher": str, must not null

"producer": str,

"original\_title": str,

"translator": str,

"publish\_time": datetime, must

"page\_number": int,

"price": float,

"pack": str,

"series": str,

"isbn": str/long int, must

"subtitle": str,

"title": str, must

"rating\_num": float, must

"book\_summary": text, must

"author\_summary": text must

}

'''

# 1. 数据去重, 略

# 可直接根据ISBN值进行比对, 方案较多

# 2. 数据格式清洗

# 对出版时间进行格式化, 包含只有年, 只有年-月, 包含年-月-日三种情况

\_ = data["publish\_time"].split("-")

if len(\_) is 1:

dt = datetime(int(\_[0]), 1, 1)

elif len(\_) is 2:

dt = datetime(int(\_[0]), int(\_[1]), 1)

elif len(\_) is 3:

dt = datetime(int(\_[0]), int(\_[1]), int(\_[2]))

else:

raise Exception("Can't trans %s to datetime"%data["publish\_time"])

# 再把时间格式转换为字符串. 注意mysql中只支持以 "-" 进行连接的时间格式. 为什么要转换为字符串, 因为之后要把数据保存到json中, 而json必须为字符串类型.

# data["publish\_time"] = dt.strftime("%Y/%m/%d")

data["publish\_time"] = dt.strftime("%Y-%m-%d")

data["page\_number"] = int(data["page\_number"])

# price中可能会包含其它的字符, 所以要提取出2位小数的价格.

data["price"] = float(re.search(r"\d+\.\d+", data["price"]).group())

data["title"] = data["title"][0].strip()

data["rating\_num"] = float(data["rating\_num"][0])

# 列表格式, 把两端的空白字符去除掉, 然后再拼接起来

data["book\_summary"] = "".join([i.strip() for i in data["book\_summary"]])

data["author\_summary"] = "".join([i.strip() for i in data["author\_summary"]])

# 3. 空值赋值/ 是否需要赋值, 看需求

# 略

return data

if \_\_name\_\_ == '\_\_main\_\_':

book\_url = "https://book.douban.com/subject/25862578/"

spider = DoubanBook()

detail = spider.get\_book\_detail(book\_url)

detail = spider.clean\_detail(detail)

import json

print(json.dumps(detail, indent=2))

#### 3.3.3. 数据存储

##### 存储到json文件

......

def store\_to\_json(self, data, filename):

import json

with open(filename, "w") as file:

# ensure\_ascii=False, 保证存储进去后, 不是unicode类型数据

file.write(json.dumps(data, indent=2, ensure\_ascii=False))

if \_\_name\_\_ == '\_\_main\_\_':

book\_url = "https://book.douban.com/subject/25862578/"

spider = DoubanBook()

tags\_data = {}

for bt, st in spider.get\_tags():

tags\_data[bt[0]] = st

detail = spider.get\_book\_detail(book\_url)

new\_detail = spider.clean\_detail(detail)

spider.store\_to\_json(tags\_data, "tags.json")

spider.store\_to\_json(new\_detail, "%s.json"%new\_detail["title"])

##### 使用sqlalchemy存储到MySQL或PostgreSQL

使用ORM存储时最重要的是建立关系型数据库的模型类, 创建数据表. 需要先设计出数据库的逻辑关系, 数据分为大标签数据, 小标签数据, 书籍详情数据这三种, 大标签和小标签是一对多的关系, 一个小标签对应多个书籍, 但一个书籍却不一定只对应一个小标签, 如小说中应该能找到钱钟书的书籍, 而在 "钱钟书" 小标签中, 很可能会找到同名的书籍. 也就是说小标签与书籍之间是多对多的关系.

新建 \douban\_book\sqlalchemy\_usage\models.py：建立模型类, 并建表

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy import Column, Integer, String, Text, Float,DateTime, ForeignKey, Index, UniqueConstraint

from sqlalchemy.orm import sessionmaker, relationship

from sqlalchemy import create\_engine

# 创建引擎, 相当于做一个数据库的连接. 传入数据库连接的url地址创建引擎的. url地址的格式: "数据库名+数据库的驱动://root:password@ip:port/db\_name?charset=utf8"

engine = create\_engine("mysql+pymysql://root:password@10.211.55.3:3306/test\_db?charset=utf8", max\_overflow=5)

# 对于postgresql, 只需要修改创建引擎时的语句, 传入postgresql的地址, 其它的都不需要进行更改, 就可以完成对postgresql的操作了.

# engine = create\_engine("postgresql+psycopg2://postgres:password@10.211.55.3:5432/test\_db", max\_overflow=5)

# 创建Base对象, 所有的模型类都是基于Base对象来创建的.

Base = declarative\_base()

class BigTag(Base):

\_\_tablename\_\_ = 'big\_tag'

id = Column(Integer, primary\_key=True, autoincrement=True)

btag = Column(String(31), unique=True, nullable=False)

\_\_table\_args\_\_ = {

"mysql\_engine": "InnoDB", # 表的引擎

"mysql\_charset": "utf8" # 表的编码格式

}

# 一对多

class SmallTag(Base):

\_\_tablename\_\_ = 'small\_tag'

id = Column(Integer, primary\_key=True, autoincrement=True)

stag = Column(String(31), unique=True, nullable=False)

# 设置外键约束. 需要指定绑定哪个数据表的哪个字段上.

btag\_id = Column(Integer, ForeignKey("big\_tag.id"))

\_\_table\_args\_\_ = {

"mysql\_engine": "InnoDB", # 表的引擎

"mysql\_charset": "utf8" # 表的编码格式

}

class Book(Base):

'''

{

"author": str,

"publisher": str,

"producer": str,

"original\_title": str,

"translator": str,

"publish\_time": datetime,

"page\_number": int,

"price": float,

"pack": str,

"series": str,

"isbn": str/long int,

"subtitle": str,

"title": str,

"rating\_num": float,

"book\_summary": text,

"author\_summary": text

}

'''

\_\_tablename\_\_ = 'book'

# nullable=False表示不允许为空

author = Column(String(63), nullable=False)

publisher = Column(String(63), nullable=False)

producer = Column(String(63))

original\_title = Column(String(63))

translator = Column(String(63))

publish\_time = Column(DateTime, nullable=False)

page\_number = Column(Integer)

price = Column(Float, nullable=False)

pack = Column(String(63))

series = Column(String(63))

# isbn可以作为每一本书的唯一标识符, 所以可以把isbn设置为书的主键.

isbn = Column(String(20), primary\_key=True)

subtitle = Column(String(63))

title = Column(String(63), nullable=False)

rating\_num = Column(Float, nullable=False)

book\_summary = Column(Text, nullable=False)

author\_summary = Column(Text, nullable=False)

# \_\_table\_args\_\_可以使用字典格式, 也可以使用元组, 如果在元组中还想要加入字典格式的key-value参数, 要把所有key-value格式的参数放在同一个字典中, 最且把字典格式的参数放在元组的最后面.

\_\_table\_args\_\_ = (

# UnqiueConstraint表示联合约束. 以title+author在一起作为约束, 这两个字段合起来不能重复, 即不允许 title和author同时重复的记录存在.

UniqueConstraint('title', 'author', name='uix\_title\_author'),

Index('title', 'author', 'publisher'), # 索引

{"mysql\_engine": "InnoDB","mysql\_charset": "utf8"}, # 表的引擎

)

# 多对多关系. 在mysql中, 如果是多对多的关系, 需要建立第3张数据表来维护对应关系.

class BookToTag(Base):

\_\_tablename\_\_ = 'booktostag'

# 在多对多关系表中, 分别建立两个字段, 以一对多的关系来对应到两个数据表中.

id = Column(Integer, primary\_key=True, autoincrement=True)

book\_isbn = Column(String(20), ForeignKey("book.isbn"))

stag\_id = Column(Integer, ForeignKey("small\_tag.id"))

\_\_table\_args\_\_ = {

"mysql\_engine": "InnoDB", # 表的引擎

"mysql\_charset": "utf8" # 表的编码格式

}

# 以上的操作只是完成了模型类的设计, 还需要创建数据表

def create\_all\_table():

# 创建所有表

Base.metadata.create\_all(engine)

def drop\_all\_table():

# 删除所有表

Base.metadata.drop\_all(engine)

if \_\_name\_\_ == '\_\_main\_\_':

create\_all\_table()

# drop\_all\_table()

spider.py代表DoubanBook类所在文件. 把DoubanBook的爬虫文件复制到sqlalchemy\_usage中, 修改为spider.py.

main.py: 实现数据存储主逻辑

from sqlalchemy.orm import sessionmaker

from spider import DoubanBook

from models import engine, BigTag, SmallTag, Book, BookToTag

# 基于models中的引擎, 使用sqlalchemy.orm的sessionmaker引擎, 创建一个Session类.

Session = sessionmaker(bind=engine)

spider = DoubanBook(20)

def store\_tag\_data():

# 创建session对象, 查询, 插入的操作都是使用session对象来完成的.

session = Session()

# spider.get\_tags()方法返回big\_tag和small\_tag.

for bt, st in spider.get\_tags():

if not session.query(BigTag).filter\_by(btag=bt[0]).first(): # 查询, 判断, 避免重复数据入库

big\_tag = BigTag(btag=bt[0])

session.add(big\_tag)

session.commit() # 因为后面需要用big\_tag的id, 所以这里必须先commit一次, 把大标签的数据保存到数据库中, 后面才能正常使用.

# 逐个保存小标签的内容

for t in st:

small\_tag = SmallTag(stag=t, btag\_id=big\_tag.id)

session.add(small\_tag)

session.commit()

session.close()

def store\_detail\_data():

session = Session()

for t in session.query(SmallTag).all(): # 查询小标签出来, 逐个下载

for page, url\_list in spider.get\_all\_pages(t.stag): # 获取所有翻页

for book\_url in url\_list: # 遍历列表页

detail = spider.get\_book\_detail(book\_url) # 获取详情

detail = spider.clean\_detail(detail) # 清洗数据

book = Book(\*\*detail)

session.add(book)

# 测试代码, 不进行翻页

# break

# 测试代码

# break

session.commit()

session.close()

if \_\_name\_\_ == '\_\_main\_\_':

store\_tag\_data()

store\_detail\_data()

运行main.py进行测试.

##### 使用Django-Orm存储到MySQL或PostgreSQL

使用django的orm思路：将爬虫程序作为django的一个app来进行使用, 如果目前你还不了解django, 建议先去熟悉一下django：

新建douban\_book/django\_orm\_usage(python package)

step.1 创建django项目：

django-amdin startproject doubanBookSpider

cd doubanBookSpider

python manage.py startapp spider

step.2 构建模型类 spider.models.py

from django.db import models

# Create your models here.

class BigTag(models.Model):

id = models.AutoField(primary\_key=True, unique=True)

btag = models.CharField(max\_length=31, unique=True, null=False)

# 一对多

class SmallTag(models.Model):

id = models.AutoField(primary\_key=True, unique=True)

stag = models.CharField(max\_length=31, unique=True, null=False)

btag = models.ForeignKey(BigTag)

class Book(models.Model):

'''

{

"author": str,

"publisher": str,

"producer": str,

"original\_title": str,

"translator": str,

"publish\_time": datetime,

"page\_number": int,

"price": float,

"pack": str,

"series": str,

"isbn": str/long int,

"subtitle": str,

"title": str,

"rating\_num": float,

"book\_summary": text,

"author\_summary": text

}

'''

\_\_tablename\_\_ = 'book'

author = models.CharField(max\_length=63, null=False, db\_index=True)

publisher = models.CharField(max\_length=63, null=False, db\_index=True)

producer = models.CharField(max\_length=63, null=True)

original\_title = models.CharField(max\_length=63, null=True)

translator = models.CharField(max\_length=63, null=True)

publish\_time = models.DateTimeField(null=False)

page\_number = models.IntegerField(null=False)

price = models.FloatField(null=False)

pack = models.CharField(max\_length=63, null=True)

series = models.CharField(max\_length=63, null=True)

isbn = models.CharField(max\_length=20, unique=True, primary\_key=True)

subtitle = models.CharField(max\_length=63, null=True)

title = models.CharField(max\_length=63, null=False, db\_index=True)

rating\_num = models.FloatField(null=False)

book\_summary = models.TextField(null=False)

author\_summary = models.TextField(null=False)

stag = models.ManyToManyField(SmallTag) # 多对多关系

step.3 更改django的settings.py, 配置数据库 注意：如果是使用mysql, 注意一定要把mysql数据库编码先改成utf8

...

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

"spider", # 注册app

]

...

DATABASES = {

# postgresql数据库设置

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': 'test\_db',

'USER': 'postgres',

'PASSWORD': 'password',

'HOST': '10.211.55.3',

'PORT': '5432'

},

# mysql数据库设置

# 'default': {

# 'ENGINE': 'django.db.backends.mysql',

# 'NAME': 'test\_db',

# 'USER': 'root',

# 'PASSWORD': 'password',

# 'HOST': '10.211.55.3',

# 'PORT': '3306',

# 'OPTIONS':{'charset':'utf8'},

# }

}

...

step.4 执行数据库迁移, 并生成数据库表

# 指定只迁移spider对应的数据库, 如果不指定, 就会迁移所有的程序, 在数据库中生成很多默认的表.

python manage.py makemigrations spider

python manage.py migrate spider

step.5 配置django的自定义命令来启动爬虫：

django中可以使用默认的命名来完成一定功能的操作, 如上面执行迁移操作时, python manager.py migrate spider, 就是使用migrate命令来完成数据库的迁移, 可以自定义命令来未完成爬虫app的启动.

在spider目录下新建management的python package, 再在management目录下创建commands的python package, 在commands中创建run.py

首先先把之前定义的含有DoubanBook类的爬虫模块spider.py拷贝到Django的spider应用文件夹下, 与models.py同级； 然后再在models.py同级建立如下结构文件夹

douban\_book目录结构

├───django\_orm\_usage

│ └───doubanBookSpider

│ ├───doubanBookSpider

│ └───spider

│ └───management

│ └───commands

├───mongoengine\_usage

└───sqlalchemy\_usage

DOUBAN\_BOOK

│ demo.py

│ spider01\_get\_tags.py

│ spider02\_get\_one\_page.py

│ spider03\_get\_all\_pages.py

│ spider04\_get\_book\_detail.py

│ spider05\_整合.py

│ spider06\_dataclean.py

│ spider07\_store\_to\_file.py

│ tags.json

│ \_\_init\_\_.py

│ 解忧杂货店.json

│

├───django\_orm\_usage

│ │ \_\_init\_\_.py

│ │

│ └───doubanBookSpider

│ │ main.py

│ │ manage.py

│ │ \_\_init\_\_.py

│ │

│ ├───doubanBookSpider

│ │ settings.py

│ │ urls.py

│ │ wsgi.py

│ │ \_\_init\_\_.py

│ │

│ └───spider

│ │ admin.py

│ │ apps.py

│ │ models.py

│ │ spider.py

│ │ tests.py

│ │ views.py

│ │ \_\_init\_\_.py

│ │

│ └───management

│ │ \_\_init\_\_.py

│ │

│ └───commands

│ run.py

│ \_\_init\_\_.py

│

├───mongoengine\_usage

│ main.py

│ models.py

│ spider.py

│ \_\_init\_\_.py

│

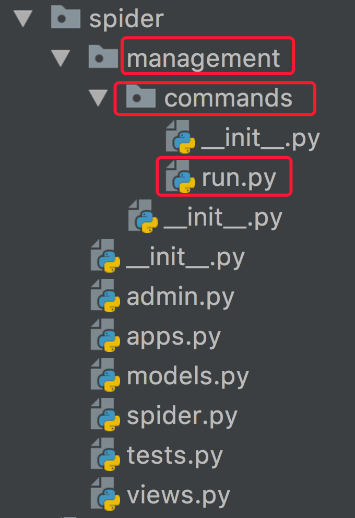
└───sqlalchemy\_usage

main.py

models.py

spider.py

\_\_init\_\_.py



然后编写run.py:

from django.core.management.base import BaseCommand

from spider.spider import DoubanBook

from spider.models import BigTag, SmallTag, Book

spider = DoubanBook()

# 基于BaseCommand来创建一个类.

class Command(BaseCommand):

help = 'run spider'

# 定义handle方法, 与sqlalchemy中的方法类似, 发起get\_tags请求, 获取数据, 处理并存储, 发起get\_all\_pages请求, 获取数据, 处理并存储. 这里的保存就直接使用对象的save方法进行存储了, 不用再使用session方法进行操作了.

def handle(self, \*args, \*\*options):

for bt, st in spider.get\_tags():

if not BigTag.objects.filter(btag=bt[0]):

big\_tag = BigTag(btag=bt[0])

big\_tag.save()

for t in st:

small\_tag = SmallTag(stag=t, btag=big\_tag)

small\_tag.save()

for t in SmallTag.objects.all(): # 查询小标签出来, 逐个下载

for page, url\_list in spider.get\_all\_pages(t.stag): # 获取所有翻页

for book\_url in url\_list: # 遍历列表页

detail = spider.get\_book\_detail(book\_url)

detail = spider.clean\_detail(detail) # 清洗数据

book = Book(\*\*detail)

book.save()

break

break

step.6 回到django项目目录, 即可使用run命令启动爬虫：

# 运行爬虫

python manage.py run

因为是在远程执行的, 为了执行的方便, 可以使用main文件来完成迁移和启动爬虫的操作. 在manage.py同目录下新建main.py文件.

新建 " douban\_book\django\_orm\_usage\doubanBookSpider\main.py"

**import** **os**

*# os.system("python manage.py makemigrations spider")*

*# os.system("python manage.py migrate spider")*

os.system("python manage.py run")

运行过程中如果出现文件没有找到的错误, 就需要手动上传文件. 因为可能不会自动上传新建的文件, 而只会自动上传修改过的文件.

如果出现 "Illegal mix of collations (latin1\_swedish\_ci, ImPLICIT) and (utf8\_general\_ci, COERCIBLE) for operation '='" 的错误, 是因为数据库的编码与爬虫爬取到的数据不匹配引起的. 修改数据库的编码, 字符集使用utf8或utf8mb4, 排序规则使用utf8 generial 或 utf8mb4\_bin, 然后删除所有的数据表, 重新执行生成迁移, 执行迁移, 运行爬虫的操作.

##### 使用mongoengine存储到MongoDB

mongodbengine由于nosql没有数据表的原因, 它也是用起来最简单的一个. 模型类的设计与关系型数据库的设计就不同了. 非关系型数据库不存在表之间的一对一和一对多的关系, 所以大标签和小标签都可以保存到同一张数据表collection中. 使用冗余的方式进行存储, 不再是关系型数据库中big\_tag大标签保存到一个表中, 不重复, small\_tag保存到一个表中, 不重复. 而是把两个tag保存到一个表/collection中, 以small\_tag + big\_tag两个字段结合在一起成为一条记录. 同样, book中也有small\_tag和big\_tab的字段, 每一个book的信息中都包含着它自己对应的big\_tag和small\_tag的信息..

新建doubanbook/mongoengine\_usage(package)/models.py

from mongoengine import \*

class Tag(Document):

stag = StringField(required=True, max\_length=31)

btag = StringField(required=True, max\_length=31)

class Book(Document):

'''

{

"author": str,

"publisher": str,

"producer": str,

"original\_title": str,

"translator": str,

"publish\_time": datetime,

"page\_number": int,

"price": float,

"pack": str,

"series": str,

"isbn": str/long int,

"subtitle": str,

"title": str,

"rating\_num": float,

"book\_summary": text,

"author\_summary": text

}

'''

author = StringField(required=True, max\_length=63)

publisher = StringField(required=True, max\_length=63)

producer = StringField(max\_length=63)

original\_title = StringField(max\_length=63)

translator = StringField(max\_length=63)

publish\_time = DateTimeField(required=True)

page\_number = IntField(required=True)

price = FloatField(required=True)

pack = StringField(max\_length=63)

series = StringField(max\_length=63)

isbn = StringField(required=True, max\_length=63)

subtitle = StringField(max\_length=63)

title = StringField(required=True, max\_length=63)

rating\_num = FloatField(required=True)

book\_summary = StringField(required=True)

author\_summary = StringField(required=True)

# 因为是nosql, 如果这些数据采取冗余的方式存储

small\_tag = StringField(required=True, max\_length=31)

big\_tag = StringField(required=True, max\_length=31)

main.py, 主程序, 用来启动所有的模块. 启动爬虫, 保存到数据库中.

from mongoengine import \*

from models import Tag, Book

from spider import DoubanBook

spider = DoubanBook()

connect('test\_db', host="10.211.55.3")

def run():

for bt, st in spider.get\_tags():

if not Tag.objects(btag=bt[0]).all():

for t in st:

tag = Tag(stag=t, btag=bt[0])

tag.save()

for t in Tag.objects.all(): # 查询小标签出来, 逐个下载

for page, url\_list in spider.get\_all\_pages(t.stag): # 获取所有翻页

for book\_url in url\_list: # 遍历列表页

detail = spider.get\_book\_detail(book\_url)

detail = spider.clean\_detail(detail) # 清洗数据

detail["small\_tag"] = t.stag

detail["big\_tag"] = t.btag

book = Book(\*\*detail)

book.save()

break

break

if \_\_name\_\_ == '\_\_main\_\_':

run()

因为mongodb不需要创建数据库和数据表, 在建立模型类和定义了main.py主程序之后, 可以直接运行.