# 第三章 知识获取

### 3.1 网络爬虫介绍

#### 3.1.1 基本原理介绍

网络爬虫，没接触过的人听起来又是一个高大上的东西，网络博客、微信公众号文章有跟多关于网络爬虫的案例。我个人所看到的关于网络爬虫主要分为两种，一种是抓取某个或多个网站的美女照片，另外一种是抓取知乎、网易云音乐等网站下对某个作品的评论数据，但本质上他们都是一样的。

我们都会使用浏览器访问网页，我自己平时主要用谷歌搜索引擎查询编程中遇到的问题，也通过在Youtube上面观看有趣的视频来获得愉悦与放松。这是一个新生词汇不断出现的时代，在日常生活中与朋友聊天时，偶尔会碰到自己不熟悉的网络用语，这是朋友就会给我回一句：“问度娘啊！”实际上我们手机上的APP基本都会访问网络，而它们访问网络数据的方式和浏览器是一样的。

前面说的都是人类通过网络访问工具来获取网络数据的，网络爬虫实现就是把这些操作都通过程序实现，通过HTTP协议把网页下载，然后分析下载下来的页面。比如人类在浏览器地址栏输入地址敲击回车，然后等待浏览器显示出html页面，再在上面找寻自己所需要的信息；而爬虫就是通过程序发送请求，服务器同样会给程序返回html页面，再通过程序从返回的页面中截取所需要的信息即可。

#### 3.1.2 爬虫与反爬虫的斗争

需要指出的是，我们所使用的谷歌、百度等搜索引擎本质上就是一个网络爬虫程序，照这样看来，网络爬虫给我们生活带来了很大的便利。随着技术的不断更新，网络爬虫也遇到各种各样的问题。

在网络发展的初始阶段，html页面都是纯文本的，爬虫时只需要对服务器返回的html页面进行解析即可，至多不过是有些html是受到保护的，需要登录才能访问，比如年轻人常使用的QQ空间，对于这样的页面只需要在爬取之前进行登录即可。

在爬虫界是有一定规范的，一般在网站网站的根目录下都会存放一个robots.txt文件，这个文件中定义了哪些内容是对爬虫开放的，哪些内容是不想被爬虫访问的，这样网站的部分或全部内容就可以不被搜索引擎访问和收录了。但是我估计你也不会愿意遵循这些规范吧，我们要的就是别人轻易不能拿到的东西，都遵守这个规范那不就和大家一样了吗，所以我们还是选择不一样，只要不做违法的事就行，而且这个规范只是约定俗成的，并不是一个标准，所以经常会有不遵守这个规范的爬虫程序。

我们都清楚，程序的速度是人类速度的很多倍，人类需要用鼠标在界面上点击，而程序访问网络是直接拿到URL就直接、迅速的访问，比人类快了不知多少倍。这样一来，服务器的压力就大了，尤其是爬虫程序很多的时候，会给网站带来巨大的流量，如果服务器扛不住，网站可能就会立马瘫痪了。企业做网站也是为了赚钱的，他们是想为正常用户提供服务，而不是为爬虫程序提供服务。

因此有很多网站都使用了反爬虫手段，最常见的一个门槛就是通过HTTP头部识别是正常用户还是爬虫程序，在HTTP头部有一个User-Agent字段，它可以让服务器识别出客户端的操作系统、浏览器版本、语言等信息，然后再针对性的进行处理，比如发送手机版网页还是桌面版网页。这是我从我的个人电脑中粘下来的*User-Agent：Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.62 Safari/537.36*；如果没有这个字段，那就说明当前请求是爬虫程序发出的，服务器会给这样的请求返回错误码或者直接禁止访问。

我们可以通过设置User-Agent字段，把程序伪装成客户端访问，甚至还可以伪装成谷歌、百度等搜索引擎的爬虫，各个网站自然都希望这些搜索引擎对自己进行收录，这可是出名的好机会呀。但是爬虫程序和人类还有一个区别，那就是爬虫程序的速度快，服务器还可以利用这个特点来识别爬虫程序，这个很简单，每访问几次就让程序休眠几秒不就识别不出来了吗，然而如果把爬虫程序设置的和人工访问一样慢，我们写爬虫程序还有什么意义呢？

还有一些网站会采取更高明的手法来反爬虫，比如在一个像素大小的图片中放置链接，而如此小的图片人类是看不见的，自然不会去点击，但是爬虫程序是分析链接并访问，而一旦访问了这种链接，服务器就可以认为请求是爬虫程序发出来的。还有就是单位时间内访问次数达到某个阀值就弹出验证码，相信很多人都经历了验证码的升级，最开始是在指定输入框填入图片上对应的字符串就能通过，对于这种验证码完全可以用OCR识别，然而现在出现了滑块验证、数学题验证等等，给爬虫技术又增加了难度。

换一个角度思考，服务器是通过流量来判别爬虫程序的，通过IP来定位爬虫程序，那么可以使用分布式爬虫，让每个机器都稍微慢一点爬虫，但是机器多，速度依旧很快；也可以使用代理IP，让IP隔一段时间就变换一次，这样就算IP被封了，依旧有新的IP来用。技术的不断发展，现在很多网站都使用javaScript通过Ajax方式来渲染页面，直接通过URL请求返回的html页面中基本没有数据，所以就不得不分析这些js代码了，js也是通过Ajax来访问服务器后台API，我们把js代码读懂之后，就可以直接访问服务器API获取数据了，但是这些Ajax都是带上参数的，而且参数经常都是经过加密的，要想使用REST API来爬取数据，就必须知道它使用的加密方式，这个过程更耗时。

为了在不需要分析js代码的前提下也能进行数据的抓取，爬虫又有了新的招数，那就是内嵌一个浏览器，让浏览器把所有数据都请求下来后再进行分析，这个浏览器可以显示界面也可以不显示页面（从Chrome 59版本开始支持Headless模式，即无界面模式），所有的渲染都可以在这个浏览器中进行，不过使用这种方式爬虫的缺点就是读读慢。

前文把爬虫与反爬虫的历史稍微梳理了一下，实际上也是一个技术的大体介绍，我们在爬虫的时候，可以根据具体的情况来选择不同的技术。

#### 3.1.3 网络爬虫分类

网络爬虫常分为四种类型，分别为通用网络爬虫、聚焦网络爬虫、增量式网络爬虫、深层网络爬虫。而在实际的运用过程同，通常是根据需要将不同的类型进行结合。

（1）通用网络爬虫常用于商业领域，因此它的技术细节公开资料较少。一般是从一些种子URL扩充到整个web，这类网络爬虫对爬行速度和存储空间要求较高，而对于页面的顺序相对要求较低，通常采用并行工作方式，主要为搜索引擎和一些大型Web服务提供商采集数据。

如图3.1通用网络爬虫体系所示，它分为页面爬取模块、页面分析模块和链接过滤模块，常采用的爬取策略为深度优先策略和广度优先策略，即图搜索算法中的深度优先和广度优先，把每个链接看作结点，那么待爬取内容就是一张巨大的网络，所以基础的深度优先和广度优先同样适用于此处。

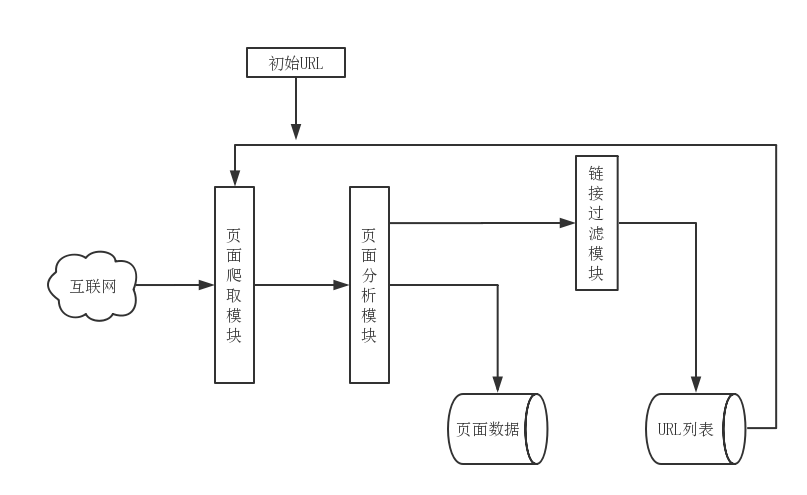


图3.1 通用网络爬虫体系结构

（2）聚焦网络爬虫也称为主题网络爬虫，是有选择性的爬行与主题相关的页面，所以相比通用网络爬虫，聚焦网络爬虫增加了链接评价和内容评价模块，因为不需要把所有页面存储下来，也不需要访问所有的链接，因此节省了硬件和网络资源，可以较好的满足特定领域信息的获取。图3.2给出了聚焦网络爬虫的体系结构，其关键是链接评价模块和内容评价模块，不同的评价策略会导致链接的访问顺序和抓取的页面数据不同。

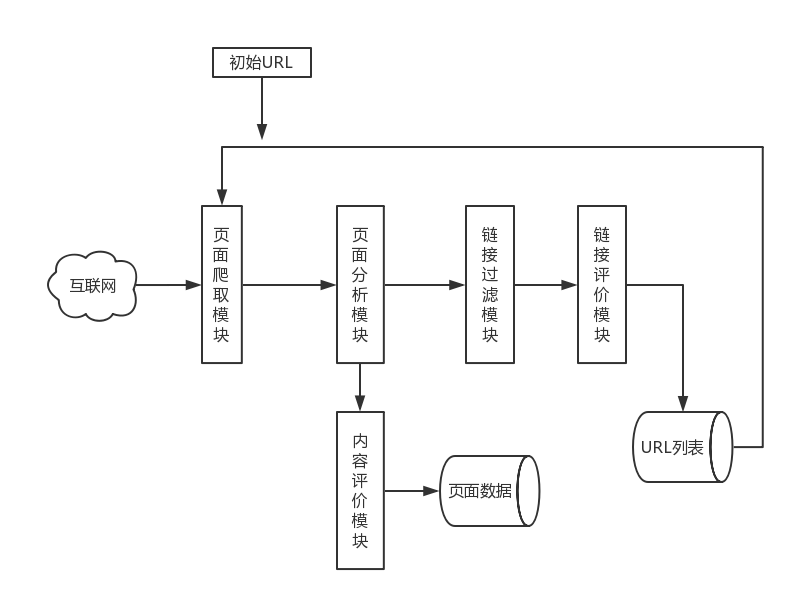


图3.2 聚焦网络爬虫体系结构

（3）增量式网络爬虫是对已下载网页采取增量式更新，即只爬行新产生或发生过变化的网页。它的优点在于，可以保证爬取的页面都是新的，可以减少数据下载量，当然，其算法的复杂度和实现难度也会相应增加。

增量式网络爬虫的目标在于保持本地集中存储的页面是最新的和提高本地页面的质量，因此需要采用一定的方法对本地页面进行更新；同时也需要按重要性对网页进行排序，来保证本地页面的质量。增量式网络爬虫的体系结构如图3.3所示，它包含待爬行URL集、本地页面集、排序模块、本地页面URL集、更新模块和爬取模块。

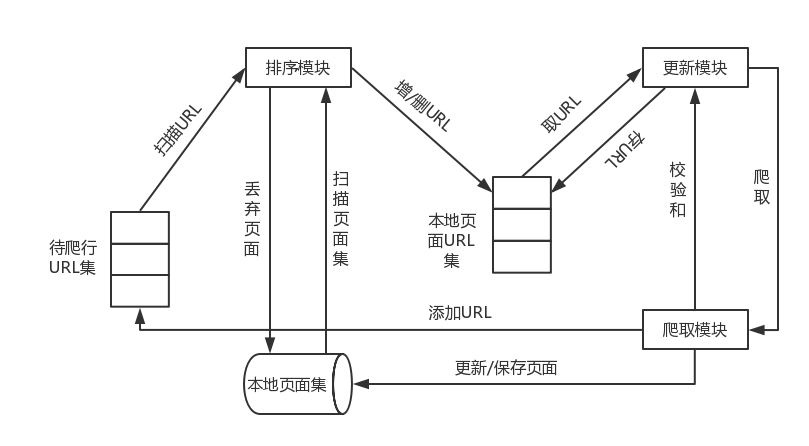


图3.3 增量式网络爬虫体系结构

（4）深层网络爬虫是爬行那些不可以通过静态链接访问的页面，比如一些要求用户注册后才可以发现的网页内容。深层网络爬虫可访问的数据是表层网页的几百倍，是互联网最大的信息资源。

深层网络爬虫体系结构如图3.4所示，其最重要的部分就是表单填写，通过表单填写完成表单验证后，进一步获取相应的网页数据。实际上，只要完成了表单验证工作，剩下的工作就和前文介绍的几种爬虫大同小异了，为了节省篇幅，此处不再赘述。

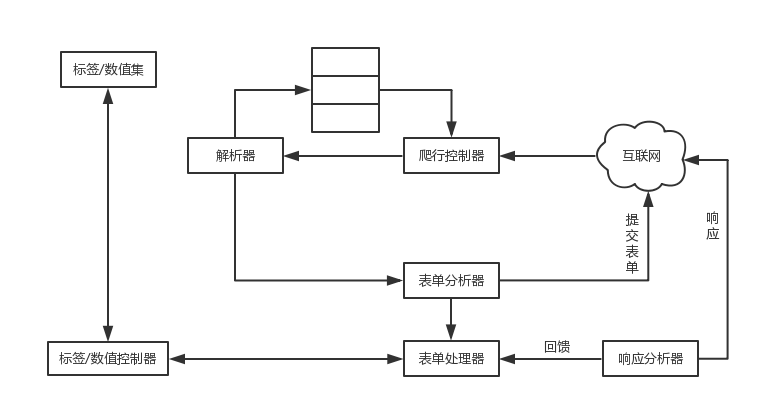


图3.4 深层网络爬虫体系结构

**参考资料：**

[1] 我是一个爬虫: <https://mp.weixin.qq.com/s/Q5hgqFLgmBAbskxLPaQkag>

[2] 孙立伟, 何国辉, 吴礼发. 网络爬虫技术的研究[J]. 电脑知识与技术, 2010, (15):4112-4115.

### 3.2 爬虫准备工作

为了方便后续的爬虫工作，我们需要了解一些关于 HTML、HTTP、浏览器调试工具等知识，不需要多深入，但是一定要有了解，尤其是 HTML 相关知识，下面对将要用到的知识做简要介绍。

#### 3.2.1 相关工具安装

要使用 python 编写网络爬虫程序，那在此之前肯定是先安装 Python 了，你可以选择到 Python 官网下载相应的安装包进行安装，本文推荐使用 Anaconda 进行安装，它提供了 Windows、Linux、MacOS 三种平台的安装包，Anaconda 已经为我们集成了很多的库，给编程工作提供了极大的便利，而如果仅仅到 Python 官网下载安装，后续就需要手动安装很多依赖库，我个人觉得麻烦，当然选择如何安装 Python 取决于你个人喜好，本文在此不浪费时间。Anaconda 下载地址为：https://www.anaconda.com/download/，图 3.5 是 Anaconda 的下载界面。

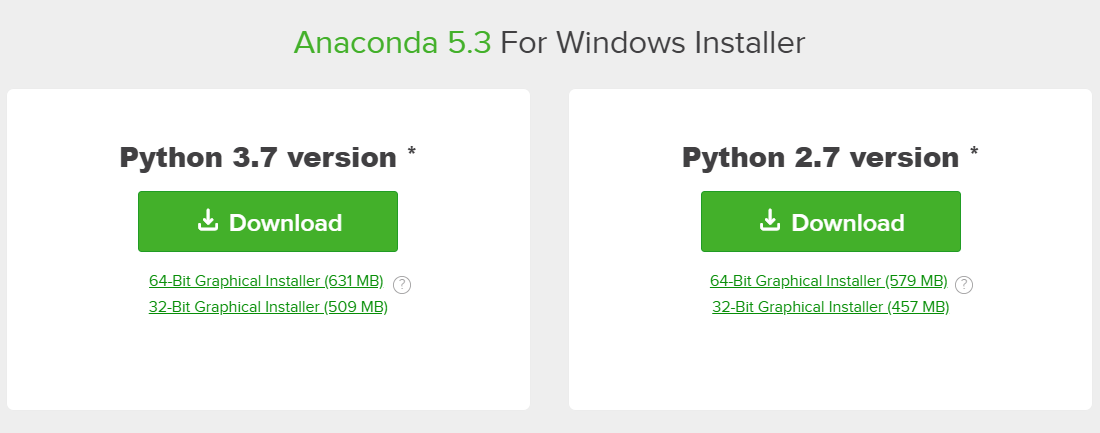


图3.5 Anaconda 下载界面

在介绍网络爬虫时我们提到，网络爬虫就是使用程序来模拟人类上网的过程，除了 Python 我们还需要一个类似于浏览器的工具，因为需要用这个工具来发送网络请求，即 Requests 库。Requests 是用Python语言编写，基于 urllib，采用 Apache2 Licensed 开源协议的 HTTP 库，此部分我们将使用 Requests 库来获取网络数据。安装 Requests 库直接使用pip install requests即可，当然你也可以选择下载源码进行本地安装或者使用 wheel 文件进行安装。

Requests 相关链接：

GitHub：<https://github.com/requests/requests>

PyPI：<https://pypi.python.org/pypi/requests>

官方文档：<http://www.python-requests.org>

中文文档：<http://docs.python-requests.org/zh_CN/latest>

Requests 是用来获取网页的，我们在上网的时候，通过眼睛就能立刻捕捉到自己想要的信息，但是对于程序来说要怎么才能捕捉到我们希望得到的信息呢？我们都知道，一个网页上面有很多我们不需要的信息，甚至还有很多我们并不那么喜欢的广告信息，为了把这些信息过滤掉，需要安装一个网页解析库，我们选择安装 BeautifulSoup 库，其安装方式和 Requests 的安装方式一致。

BeautifulSoup 相关链接：

官方文档：<https://www.crummy.com/software/BeautifulSoup/bs4/doc>

中文文档：<https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh>

PyPI：<https://pypi.python.org/pypi/beautifulsoup4>

Scrapy是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。其最初是为了 [页面抓取](http://en.wikipedia.org/wiki/Screen_scraping) (更确切来说, [网络抓取](http://en.wikipedia.org/wiki/Web_scraping) )所设计的， 也可以应用在获取API所返回的数据(例如 [Amazon Associates Web Services](http://aws.amazon.com/associates/) ) 或者通用的网络爬虫。

Scrapy 是一个很强大的爬虫框架，它依赖的库很多，可以使用直接使用 pip install scrapy 安装，我自己在使用这种方式安装的时候没有成功，所以我采用的是 whl 文件进行安装，这些库的安装方式都差不多，所以不作过多介绍了。

Scrapy 相关链接：

官方网站：[https://scrapy.org](https://scrapy.org/)

官方文档：[https://docs.scrapy.org](https://docs.scrapy.org/)

PyPI：<https://pypi.python.org/pypi/Scrapy>

GitHub：<https://github.com/scrapy/scrapy>

中文文档：[http://scrapy-chs.readthedocs.io](http://scrapy-chs.readthedocs.io/)

#### 3.2.2 HTTP 基本原理

##### 3.2.2.1 HTTP 和 HTTPS

为了更好的开始爬虫工作，我们还需要对 HTTP 原理有一定的了解。HTTP 是 Hyper Text Transfer Protocol（超文本传输协议）的缩写，它是用于从 WWW 服务器传输超文本到本地浏览器的传送协议，可以使浏览器更加高效，使网络传输减少。万维网协会（World Wide Web Consortium）和 Internet 工作小组 IETF（Internet Engineering Task Force）于 1945 年定义了 HTTP/1.0 版本，现在广泛使用的是 HTTP/1.1 版本。

在访问网页时，地址栏都是以 HTTP 或 HTTPS 开头，HTTPS 是在 HTTP 的基础上加入了 SSL 层，使用 HTTPS 传输的内容是经过加密的，现在大部分网站和 APP 都已经采用了 HTTPS，各大互联网公司也在积极的推动使用 HTTPS，我们爬虫过程中抓取的页面通常都是 HTTP 和 HTTPS 协议的。

我们在浏览器地址栏输入一个 URL，敲击回车之后，页面就会在本地逐渐显现出来，这个过程就是通过 HTTP 协议向服务端发送了一个请求，服务端将对应的响应返回回来后，由浏览器解析并渲染出来，便将网页呈现出来了，图 3.6 是一个简单的 HTTP 请求模型图，实际情况远比这复杂的多。

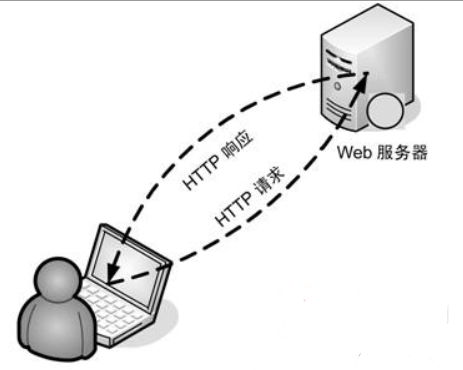


图3.6 HTTP 请求模型图

##### 3.2.2.2 URI 和 URL

你肯定听说过 URL，它是 Universal Resource Locator（统一资源定位符）的缩写，我们浏览器输入 URL 就能访问到对应的资源；URI 是 Uniform Resource Identifier（统一资源标志符）的缩写，URL 是 URI 的子集。还有一个 URN（Uniform Resource Name：统一资源命名）的概念可能听说的少一些，它也是 URI 的子集，可以唯一标识一个实体，但是不给出实体的位置，通过引用一个或多个URL来实现标识指针功能。现实中可见的使用是 P2P 下载中使用的磁力链接。

我们平时基本都是使用的 URL，在后续爬虫过程中也基本使用 URL，对 URL 还需要了解一点的是绝对路径表示的 URL 和相对路径表示的 URL 之间的区别，比如https://www.baidu.com/img/bd\_logo1.png就是一个绝对路径表示的 URL，你可以通过这个地址直接访问到百度的 logo，而 /img/bd\_logo1.png则是相对路径表示的 URL（我个人觉得此时叫 URI 应该更准确），你无法通过它获取到具体的资源，我们需要将其补全后才能获取到具体的资源。后面我们将会遇到需要由我们补全 URL 的情况。

##### 3.2.2.3 HTTP 请求

HTTP 请求包含四部分，分别为：请求方法、请求头、请求体、请求地址。常见的请求方法有：GET、POST、DELETE等。我们主要使用 GET 和 POST 方式发送请求，它们的区别在于 GET 请求的参数直接包含在 URL 中，并且 GET 请求提交的数据有字节限制，我们常用的百度搜索发起的就是一个 GET 请求；POST 请求不会将参数暴露在 URL 中，同时也不会限制传输数据的大小，所以可以用 POST 请求来上传文件。

请求头说明服务器要使用的附加信息，就爬虫而言比较重要的有 User-Agent、Content-Type、Cookie 等，大部分情况下，爬虫程序都需要写请求头，如果不写请求头，就相当于直接告诉服务器这是一个爬虫程序，很有可能会被禁止访问，通过设置请求头可以给爬虫程序戴上一个面具。

POST 请求中的表单数据一般放在请求体中，如果是 GET 请求则请求体为空。

关于 HTTP 请求更加深入的知识可自行查阅相关资料，限于篇幅，本文不作详解。

#### 3.2.3 浏览器调试工具的使用

现代浏览器基本都提供了调试工具供开发者使用，本文的介绍以 Windows 10 下的 Chrome 69.0.3497.100 为例。

在浏览器页面点击鼠标右键，会弹出一个菜单列表，选择最后一项**检查**，就能打开调试工具；或者直接在浏览器页面按 F12，也会打开调试工具，图3.7为 Chrome 浏览器菜单选项。



图3.7 Chrome 菜单选项

下面以过人熟知的百度进行介绍，打开调试工具后，浏览器会自动在底部打开调试工具界面，如图3.8所示，如果你的你的调试工具不是在底部显示，而是在右侧或者左侧显示，可以点击图中红色箭头指示部分，进行切换，如图3.9所示。



图3.8 在百度首页打开调试工具

可以看到，你可以选择在浏览器底部、左边、右边显示调试工具栏，也可以独立出来一个窗口进行显示，此菜单栏包含的其他选项可自行了解。

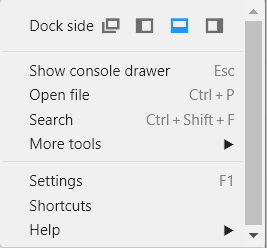


图3.9 切换调试工具显示方式

我们首先进入的是调试工具的控制台（百度在其首页放置了一个彩蛋，可以在图3.8看到），控制台可以调试 javascript 代码，它的作用类似于eclispse、pycharm 等 IDE 的控制台，可以直接在调试工具的控制台输入 javascript 代码执行，例如输入 console.info(“hello world”); ，敲击回车键后，立刻就能看到执行结果，如图3.10所示。



图3.10 使用控制台执行 javascript 代码

在控制台有很多导航栏（如图3.11所示），我们选择会用到的导航栏作介绍，上面已经介绍了 Console，下面就我们爬虫过程最常用的 Elements 和 Network 做简要介绍。



图3.11 调试工具导航栏

Elements 标签页如图3.12所示，左侧显示页面的 HTML 元素，右侧显示 CSS 样式表、事件监听器等信息，右侧信息在 web 开发过程中会经常用到，但是我们并不是做网页开发，而是从网页中获取对自己有用的信息，所以对样式表不作介绍，你可以尝试修改右侧样式表观察实际效果。

页面所呈现的内容即调试工具左侧 HTML 元素，如果你想查看某一特定元素，可以先点击导航栏左上角那个小箭头按钮，然后将鼠标移动到移动到你所想查看的元素上去，比如百度的 logo，显示框就会自动将你所指定元素的 HTML 代码呈现出来，当然你也可以选择自己分析整个 body 标签中的内容，同样能找出它们与页面元素的一一对应关系，只是我个人觉得费时间。



图3.12 Elements 标签页

如图3.13所示，点击左上角的小箭头之后，然后将鼠标光标移动到百度的 logo 上面，会发现 HTML 标签会自动展开，并将你所指定的元素呈现在页面中间。可以看到红色方框内有连个 img 标签，你可以把它们的 src 属性所对应的值（即 URL）复制到浏览器地址栏查看，就能得到一张百度的 logo 图片；同样你也可以通过将鼠标移动到标签页对应的元素上去，浏览器界面会将对应的元素设置为蓝色背景突出显示，比如你讲鼠标移动到此处的 img 标签上去的时候，百度的 logo 图片就会被涂上蓝色背景突出显示。



图3.13 选择指定元素

对于没有 CSS 或 XPath 基础的同学来说，写选择器的时候可能有一点头疼，调试工具也给我们设置快捷的方式。如图3.14所示，在指定元素点击鼠标右键会发现可以复制选择器或者 XPath，是不是有一种幸福来得太突然，一时无法接受的感觉。

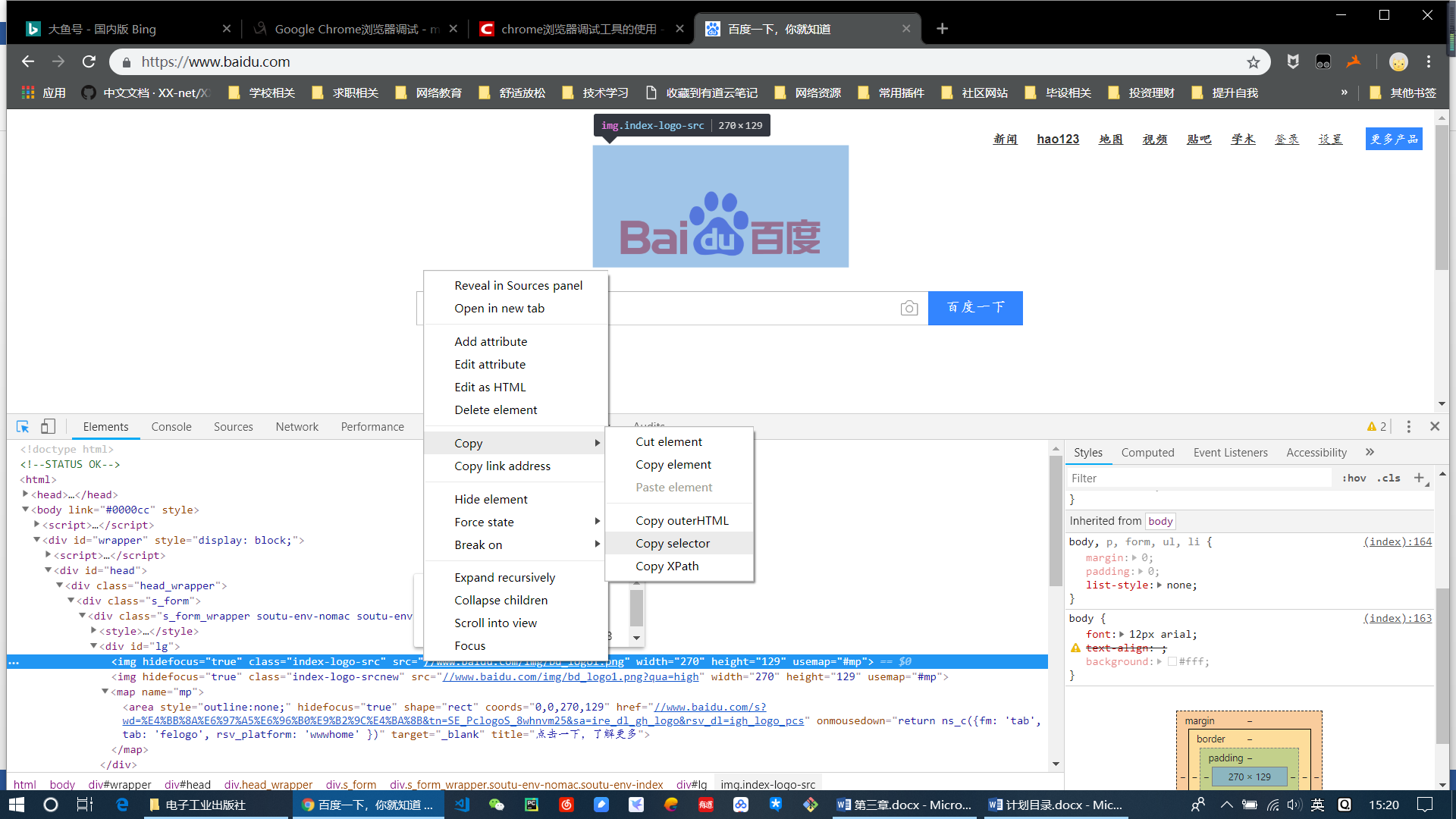


图3.14 使用调试工具提供的选择器功能

下面我们开始介绍 Network 选项卡的内容，其打开页面如图3.15所示，图中标注了两个常用的区域，左上角的小按钮可以清空已发送的请求列表，而且我们也可以使用调试提供的过滤功能来筛选我们想要查看的请求。

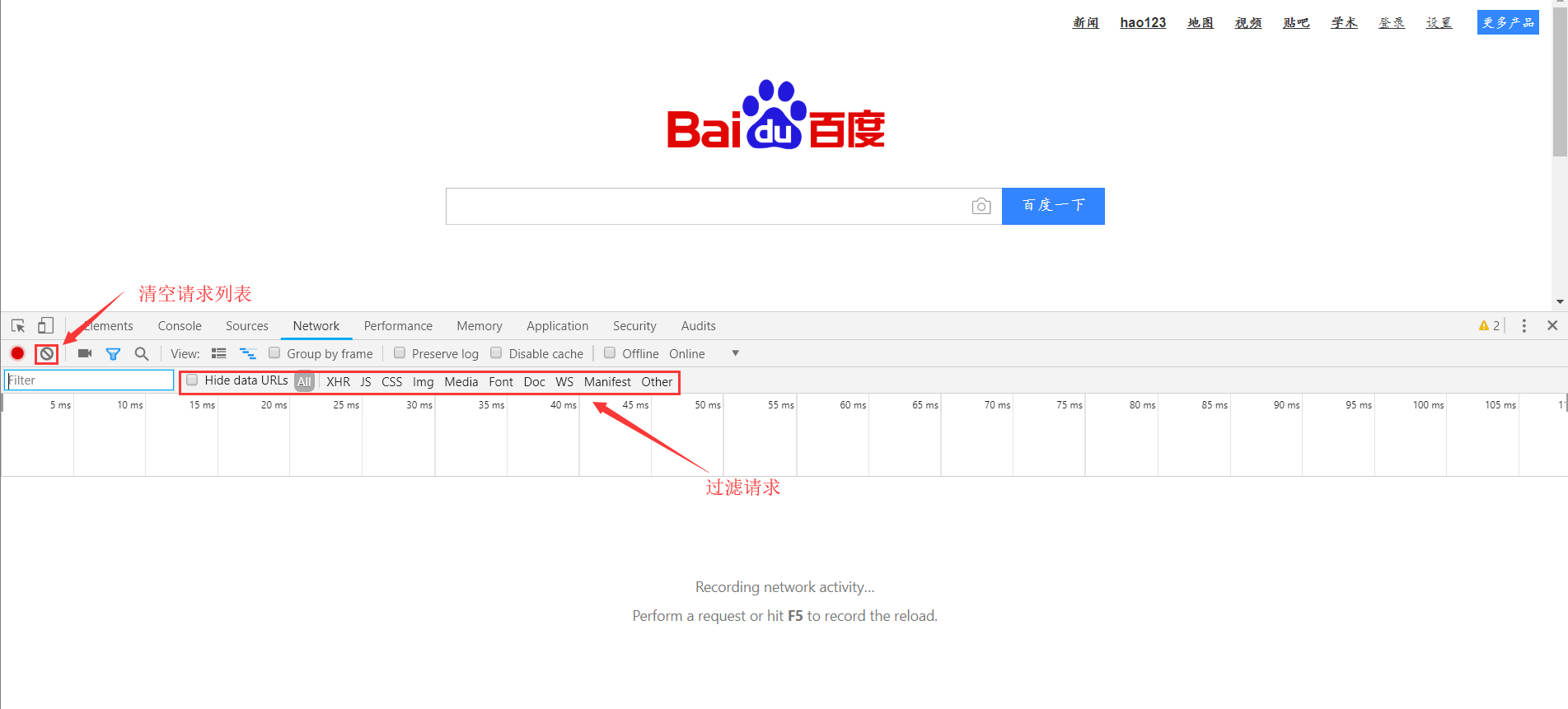


图3.15 Network 选项卡信息

在前一步的基础上，我们可以按 F5 或者点击浏览器左上角的刷新按钮刷新一下页面，查看调试工具信息的变化，如图3.16所示，我们可以看到每个请求的名称、请求所对应的状态码、资源的类型、每个请求的发起者、资源来源于网络还是本地缓存、以及时间等信息。

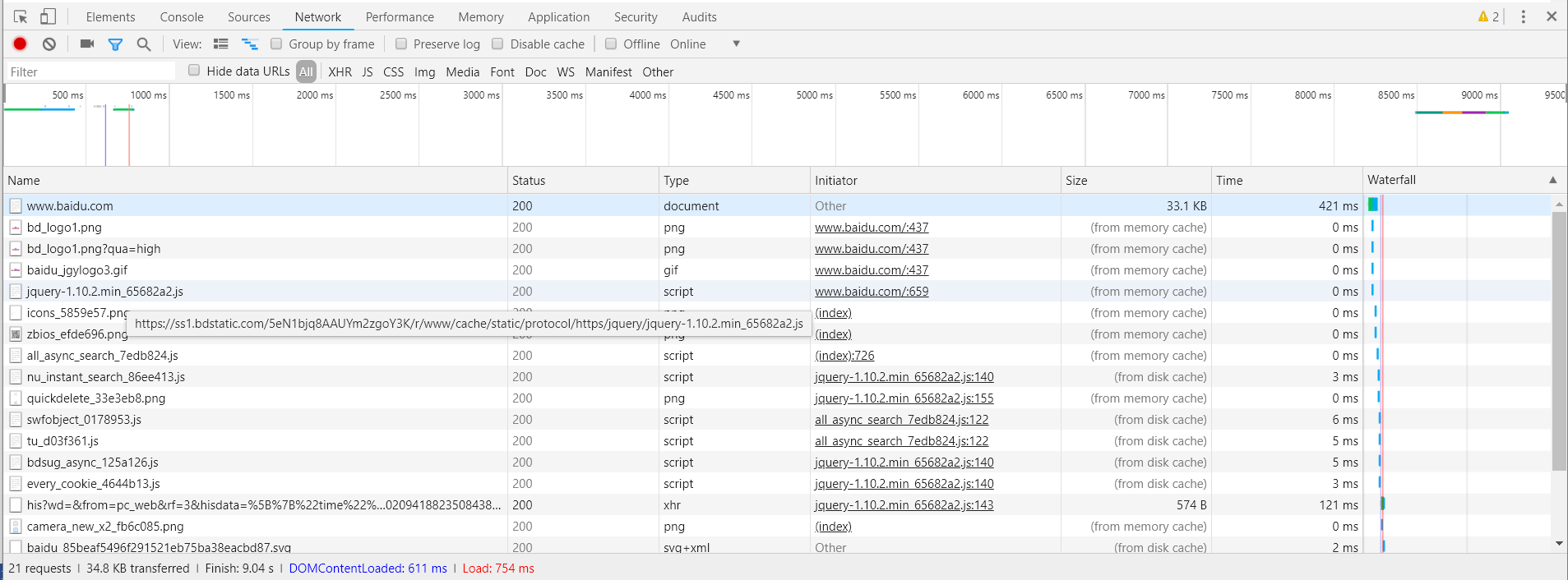


图3.16 查看请求列表

你可以随机选择一个请求点击查看其详细信息，图3.17是名为 [www.baidu.com](http://www.baidu.com)（请求列表第一个）请求的详细信息，可以看到请求以及响应的详细信息，提交的表单信息、Cookie 等详细信息都可以看到。有一些网站我们并不需要去分析它的 html 结构，可以直接分析它的接口，通过程序调用其接口就可以获取到后台数据了，而分析接口时我们就需要去挖掘对应请求的详细信息。

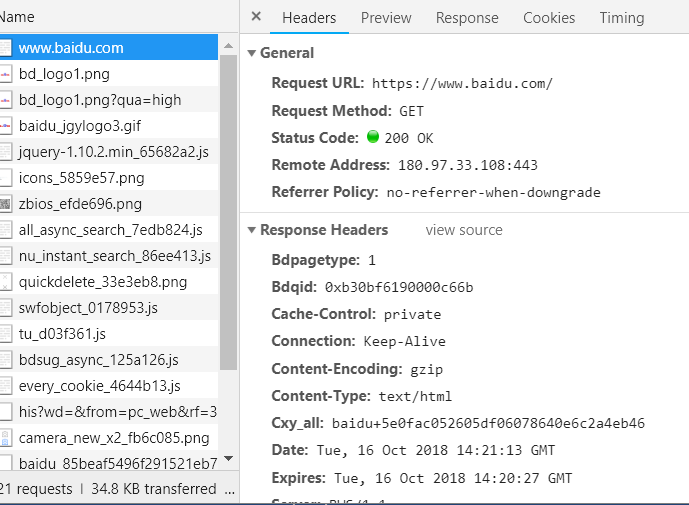


图3.17 请求的详细信息

#### 3.2.4 HTML 和 CSS 选择器

HTML 的全称为超文本标记语言（Hyper Text Markup Language），不同类型的元素通过标签来表示，比如图片用 <img> 标签来表示，整个网页文件是一棵 dom 树，如图 3.18 所示，根结点是 html，然后是 head、body，你仔细想一下，它的组织方式和我们的文件系统是一样的，都是以树的形式组织，文件系统是一层一层的目录，而 html 不过换成了规定的标签而已。

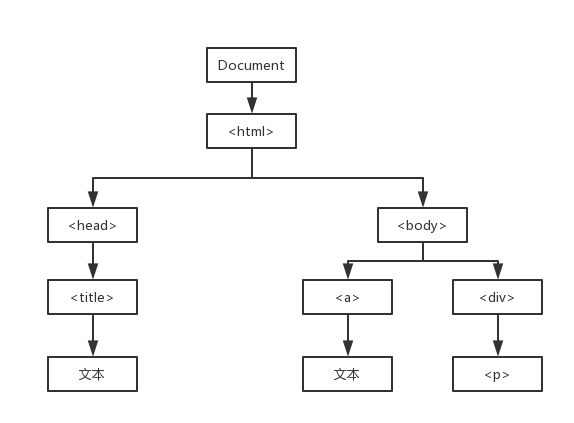


图3.18 DOM 树示意图

我们爬虫所要抓取的内容就包含在 html 中。你可能也已经发现了，虽然网页在浏览器上呈现出来时是很美观的，但是我们程序分析的只是 html，而网页显示的效果是 html、css、javascript 三者共同协作的结果，为了从 html 源码中拿到我们需要的信息，脑子里崩出来的第一想法应该是解析 html，自然而然就会想到自己写一个能解析 html 文件的程序。时间就成为了一个巨大的问题，实际上我们并不需要重复造轮子，对于 html 的解析已经有现成的工具可以实现了。

希望你没有忘掉正则表达式，它在很多地方都能方便快捷的解决我们的问题，同样我们也可以用它去匹配 html 文件中的信息（对正则表达式语法不熟悉的同学可以查看本章的附录，我个人总结了正则表达式的基本语法），而写过正则表达式的人都知道，简洁的另一面就是难于理解，要写出一个符合具体业务的正则表达式是不简单的，因此不推荐使用正则表达式。

其实还有专门用于过滤 html 文件内容的工具，那就是 XPath 和 CSS 选择器，推荐使用这两种方式，我个人不懂 XPath，不敢多言，下面着重介绍一下 CSS 选择器。

前文已经介绍了，网页是由一个一个的结点组成，CSS 样式表用于定义不同结点的样式规则，那么如何来定位结点呢？就需要用到 CSS 选择器了，它不仅语法简单易学，而且还支持嵌套选择，不同选择器之间使用空格隔开，即可实现嵌套选择，比如下面这个单一结点

<input id=”username” class=”input” name=”username” />

要定位到这个结点，可以有以下四种方法：

（1）、#username ID选择器

（2）、.input 类选择器

（3）、input 标签选择器

（4）、input[name=’username’] 属性选择器

表3.1给出了 CSS 选择器的语法规则，应能根据具体业务合理的应用这些语法规则，在能实现业务的前提下保持简洁。

表3.1 CSS 选择器语法规则

|  |  |  |  |
| --- | --- | --- | --- |
| **选择器** | **例子** | **例子描述** | **CSS** |
| [.*class*](http://www.w3school.com.cn/cssref/selector_class.asp) | .intro | 选择 class="intro" 的所有元素。 | 1 |
| [#*id*](http://www.w3school.com.cn/cssref/selector_id.asp) | #firstname | 选择 id="firstname" 的所有元素。 | 1 |
| [\*](http://www.w3school.com.cn/cssref/selector_all.asp) | \* | 选择所有元素。 | 2 |
| [*element*](http://www.w3school.com.cn/cssref/selector_element.asp) | p | 选择所有 <p> 元素。 | 1 |
| [*element*,*element*](http://www.w3school.com.cn/cssref/selector_element_comma.asp) | div,p | 选择所有 <div> 元素和所有 <p> 元素。 | 1 |
| [*element* *element*](http://www.w3school.com.cn/cssref/selector_element_element.asp) | div p | 选择 <div> 元素内部的所有 <p> 元素。 | 1 |
| [*element*>*element*](http://www.w3school.com.cn/cssref/selector_element_gt.asp) | div>p | 选择父元素为 <div> 元素的所有 <p> 元素。 | 2 |
| [*element*+*element*](http://www.w3school.com.cn/cssref/selector_element_plus.asp) | div+p | 选择紧接在 <div> 元素之后的所有 <p> 元素。 | 2 |
| [[*attribute*]](http://www.w3school.com.cn/cssref/selector_attribute.asp) | [target] | 选择带有 target 属性所有元素。 | 2 |
| [[*attribute*=*value*]](http://www.w3school.com.cn/cssref/selector_attribute_value.asp) | [target=\_blank] | 选择 target="\_blank" 的所有元素。 | 2 |
| [[*attribute*~=*value*]](http://www.w3school.com.cn/cssref/selector_attribute_value_contain.asp) | [title~=flower] | 选择 title 属性包含单词 "flower" 的所有元素。 | 2 |
| [[*attribute*|=*value*]](http://www.w3school.com.cn/cssref/selector_attribute_value_start.asp) | [lang|=en] | 选择 lang 属性值以 "en" 开头的所有元素。 | 2 |
| [:link](http://www.w3school.com.cn/cssref/selector_link.asp) | a:link | 选择所有未被访问的链接。 | 1 |
| [:visited](http://www.w3school.com.cn/cssref/selector_visited.asp) | a:visited | 选择所有已被访问的链接。 | 1 |
| [:active](http://www.w3school.com.cn/cssref/selector_active.asp) | a:active | 选择活动链接。 | 1 |
| [:hover](http://www.w3school.com.cn/cssref/selector_hover.asp) | a:hover | 选择鼠标指针位于其上的链接。 | 1 |
| [:focus](http://www.w3school.com.cn/cssref/selector_focus.asp) | input:focus | 选择获得焦点的 input 元素。 | 2 |
| [:first-letter](http://www.w3school.com.cn/cssref/selector_first-letter.asp) | p:first-letter | 选择每个 <p> 元素的首字母。 | 1 |
| [:first-line](http://www.w3school.com.cn/cssref/selector_first-line.asp) | p:first-line | 选择每个 <p> 元素的首行。 | 1 |
| [:first-child](http://www.w3school.com.cn/cssref/selector_first-child.asp) | p:first-child | 选择属于父元素的第一个子元素的每个 <p> 元素。 | 2 |
| [:before](http://www.w3school.com.cn/cssref/selector_before.asp) | p:before | 在每个 <p> 元素的内容之前插入内容。 | 2 |
| [:after](http://www.w3school.com.cn/cssref/selector_after.asp) | p:after | 在每个 <p> 元素的内容之后插入内容。 | 2 |
| [:lang(*language*)](http://www.w3school.com.cn/cssref/selector_lang.asp) | p:lang(it) | 选择带有以 "it" 开头的 lang 属性值的每个 <p> 元素。 | 2 |
| [*element1*~*element2*](http://www.w3school.com.cn/cssref/selector_gen_sibling.asp) | p~ul | 选择前面有 <p> 元素的每个 <ul> 元素。 | 3 |
| [[*attribute*^=*value*]](http://www.w3school.com.cn/cssref/selector_attr_begin.asp) | a[src^="https"] | 选择其 src 属性值以 "https" 开头的每个 <a> 元素。 | 3 |
| [[*attribute*$=*value*]](http://www.w3school.com.cn/cssref/selector_attr_end.asp) | a[src$=".pdf"] | 选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。 | 3 |
| [[*attribute*\*=*value*]](http://www.w3school.com.cn/cssref/selector_attr_contain.asp) | a[src\*="abc"] | 选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。 | 3 |
| [:first-of-type](http://www.w3school.com.cn/cssref/selector_first-of-type.asp) | p:first-of-type | 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。 | 3 |
| [:last-of-type](http://www.w3school.com.cn/cssref/selector_last-of-type.asp) | p:last-of-type | 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-of-type](http://www.w3school.com.cn/cssref/selector_only-of-type.asp) | p:only-of-type | 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-child](http://www.w3school.com.cn/cssref/selector_only-child.asp) | p:only-child | 选择属于其父元素的唯一子元素的每个 <p> 元素。 | 3 |
| [:nth-child(*n*)](http://www.w3school.com.cn/cssref/selector_nth-child.asp) | p:nth-child(2) | 选择属于其父元素的第二个子元素的每个 <p> 元素。 | 3 |
| [:nth-last-child(*n*)](http://www.w3school.com.cn/cssref/selector_nth-last-child.asp) | p:nth-last-child(2) | 同上，从最后一个子元素开始计数。 | 3 |
| [:nth-of-type(*n*)](http://www.w3school.com.cn/cssref/selector_nth-of-type.asp) | p:nth-of-type(2) | 选择属于其父元素第二个 <p> 元素的每个 <p> 元素。 | 3 |
| [:nth-last-of-type(*n*)](http://www.w3school.com.cn/cssref/selector_nth-last-of-type.asp) | p:nth-last-of-type(2) | 同上，但是从最后一个子元素开始计数。 | 3 |
| [:last-child](http://www.w3school.com.cn/cssref/selector_last-child.asp) | p:last-child | 选择属于其父元素最后一个子元素每个 <p> 元素。 | 3 |
| [:root](http://www.w3school.com.cn/cssref/selector_root.asp) | :root | 选择文档的根元素。 | 3 |
| [:empty](http://www.w3school.com.cn/cssref/selector_empty.asp) | p:empty | 选择没有子元素的每个 <p> 元素（包括文本节点）。 | 3 |
| [:target](http://www.w3school.com.cn/cssref/selector_target.asp) | #news:target | 选择当前活动的 #news 元素。 | 3 |
| [:enabled](http://www.w3school.com.cn/cssref/selector_enabled.asp) | input:enabled | 选择每个启用的 <input> 元素。 | 3 |
| [:disabled](http://www.w3school.com.cn/cssref/selector_disabled.asp) | input:disabled | 选择每个禁用的 <input> 元素 | 3 |
| [:checked](http://www.w3school.com.cn/cssref/selector_checked.asp) | input:checked | 选择每个被选中的 <input> 元素。 | 3 |
| [:not(*selector*)](http://www.w3school.com.cn/cssref/selector_not.asp) | :not(p) | 选择非 <p> 元素的每个元素。 | 3 |
| [::selection](http://www.w3school.com.cn/cssref/selector_selection.asp) | ::selection | 选择被用户选取的元素部分。 | 3 |

**参考资料**：

[1] CSS 选择器参考手册 ：<http://www.w3school.com.cn/cssref/css_selectors.asp>

[2] [Python 之网页解析器](Python%20之网页解析器) ：<https://www.cnblogs.com/my1e3/p/6622306.html>

#### 3.2.5 网页解析器

网页解析器简单的说就是用来解析 HTML 的工具，如图 3.19 所示，即用它来提取 HTML 网页中的信息，过滤出我们想要的有价值的信息。

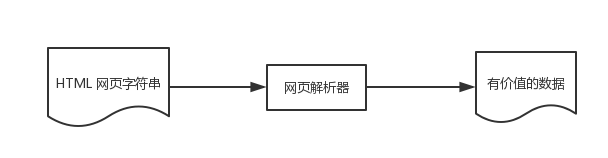


图3.19 网页解析图解

常见的网页解析工具有：正则表达式匹配、Python 自带的 html.parser 模块、lxm 库和 BeautifulSoup 解析器。其中正则表达式属于模糊匹配，即按照指定规则匹配字符串；而后三种属于结构化解析，所谓结构化解析就是将整个 HTML 文档当做一个 Document 对象，然后利用其树形结构的标签形式，对这个对象的标签进行遍历和信息提取操作。

我们主要使用 BeautifulSoup 网页解析器，它的使用方法非常简单，只需要给其构造函数传入 HTML 字符串和一个指定的网页解析器就能得到一个BeautifulSoup 对象，然后就可以使用它的 select()、find()、find\_all() 等方法了，如下所示：

soup = BeautifulSoup(html\_doc, 'html.parser')  
soup.select('a')  
soup.find('a', href='/1234.html')  
soup.find\_all('a', href=re.compile(r'/\d+\.html'))

更多具体关于BeautifulSoup 的使用，我们将在实践中逐渐介绍。

### 3.3 简单爬虫案例实现

本章我们将通过对豆瓣 Top250 的抓取，来介绍三种不同的爬虫方式，为后续的知识获取做准备。

#### 3.3.1 使用 Requests 爬虫

我们通过 Requests 库所提供的接口进行爬虫，案例为抓取网站为豆瓣电影 Top250，使用 Requests 的 get 方法获取页面数据，该方法会返回一个 Response 对象，示例代码如下：

#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-  
# @File : hello\_world.py  
  
import requests  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 response = requests.get('https://movie.douban.com/top250')  
 print(response)  
 print(response.status\_code)  
 print(response.text)

这段代码很简单，首先导入 requests 库，然后调用 requests库所实现的 get 方法，并给它传入一个 url，它就会给我们返回一个 Response 对象，然后打印出了这个对象，以及状态码和响应内容，运行结果如图 3.18 所示。

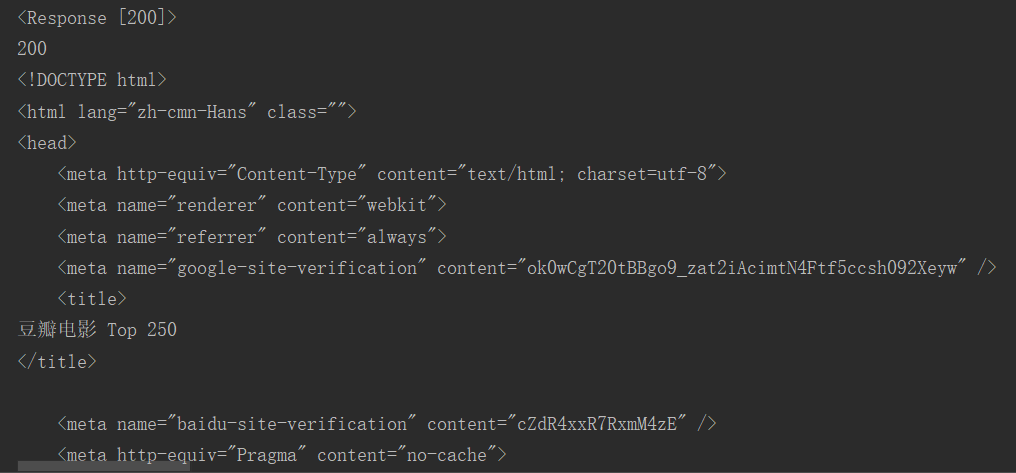


图3.18 hello\_world.py 运行结果

如果你的控制台输出出现乱码情况，你可以将编码设置成 UTF-8 格式，设置方式为：File 🡪 Settings 🡪 Editor 🡪 File Encoding，然后将编码格式格式设置为 UTF-8 就可以了。

除了 get 请求，Requests 还为我们提供了 post、delete、put、head 等请求，为我们的爬虫工作提供了极大的便利性，你只需要根据具体业务调用不同方法就可以了。除了这些基本用法，Requests 还提供了文件上传、会话维持、设置代理、SSL 证书验证、身份认证、超时设置等功能，可以根据情况灵活使用。

细心的你可能会发现，get 方法还可以接收其它参数，我们常设的一个重要阐述就是 headers 信息，如果不设置 headers 信息，那就等于告诉服务器我是一个爬虫程序，所以大多大多时候设置 headers 信息是非常有必要的，如果以后你的爬虫程序运行出错了，那就可以先查查又没有设置请求头信息。将豆瓣 Top250 作为第一个案例，是因为它简单且适合练手，不用设置请求头也能抓取内容，下面我们开始介绍如何抓取豆瓣 Top250的信息。

首先用浏览器打开豆瓣 Top250 的页面，并打开控制台查看网页源代码，如图3.20 所示，所有电影的信息都被组织在一个 <ol></ol> 标签中，其中每一个 <li></li> 标签就是一个电影的具体信息。

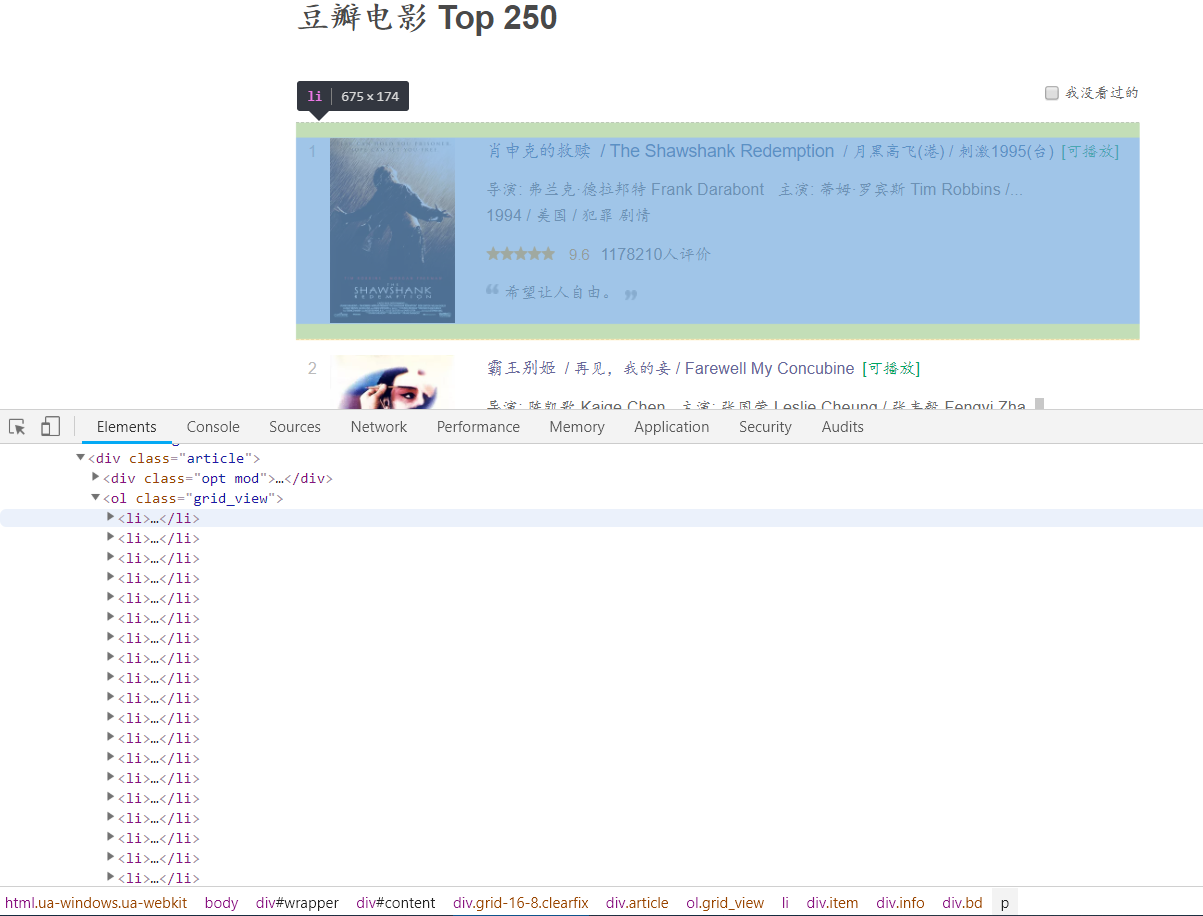


图3.20 豆瓣 Top250 网页分析

下面是抓取程序：

#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-  
# @File : crawl\_douban\_top250.py  
# @Desc : 抓取豆瓣 top2 内容  
  
import requests  
from bs4 import BeautifulSoup  
  
  
def get\_html(url):  
 response = requests.get(url)  
 response.encoding = 'utf-8'  
 return response.text  
  
  
def parse\_html(html):  
 soup = BeautifulSoup(html, 'html.parser')  
 return soup  
  
  
def get\_movies\_soup(soup):  
 movies = soup.select('.grid\_view li')  
 return movies  
  
  
def get\_movie\_details(movie):  
 ranking = movie.select('.pic em')[0].text  
 img = movie.select('.pic img')[0]['src']  
 title = movie.select('.title')[0].text  
 score = movie.select('.rating\_num')[0].text  
 pople\_num = movie.select('.star span')[3].text  
 decsription = movie.select('.inq')[0].text  
  
 info = movie.select('.bd p')[0].text  
 director = info.strip().split('\n')[0].split('\xa0\xa0\xa0')[0]  
  
 yac = info.strip().split('\n')[1].strip().split('\xa0/\xa0')  
 year = yac[0]  
 area = yac[1]  
 clazz = yac[2]  
 return {  
 'title': title,  
 'clazz': clazz,  
 'ranking': ranking,  
 'img': img,  
 'score': score,  
 'pople\_num': pople\_num,  
 'decsription': decsription,  
 'director': director,  
 'year': year,  
 'area': area  
 }  
  
  
def get\_movies\_list(movies\_soup):  
 movies\_list = []  
 for movie\_soup in movies\_soup:  
 movie\_details = get\_movie\_details(movie\_soup)  
 movies\_list.append(movie\_details)  
 return movies\_list  
  
  
def main():  
 url = 'https://movie.douban.com/top250'  
 html = get\_html(url)  
 soup = parse\_html(html)  
 movies\_soup = get\_movies\_soup(soup)  
 movies\_list = get\_movies\_list(movies\_soup)  
 return movies\_list  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 movies = main()  
 for movie in movies:  
 print(movie)

该程序的运行结果如图 3.21 所示，下面我们对该程序做一个详细的介绍分析。

程序一开始引入了必备的 Requests 和 BeautifulSoup 解析器。get\_html(url)函数的功能是获取指定 url 对应的 html，函数体中为了防止乱码，将 response 的编码格式设置成了 utf-8，返回的是一个字符串对象，该字符串中的内容就是网页的 html 源码。

parse\_html(html) 函数用于将字符串形式的 html 转换为 BeautifulSoup 对象，BeautifulSoup 的构造函数函数中我们选择了 html.parser 作为网页解析器。

有了便于操作的BeautifulSoup 对象，在 get\_movies\_soup(soup) 中，直接使用选择器过滤出电影信息，该函数返回的结果是一个BeautifulSoup 数组，数组中的每个元素都是一部电影的信息，其中第一个元素的信息如图 3.22 所示。

拿到了电影元素的列表，我们再对其中的每个元素使用 get\_movie\_details()函数进行处理，用选择器去过滤我们所需要的信息，需要注意的是，select() 方法返回的结果是一个列表。其中选择图片的时候想要得到的是图片路径信息，BeautifulSoup 给我们提供了非常方便的解决方案，可以像操作字典一样操作 DOM 元素，及传入 ‘src’ 即代表选择该属性。

其中导演、地区、年份、类别等信息又稍显的麻烦，因为我们通过选择器只能过滤出一个字符串，这个字符串里面包含了很多的导演、地区等信息，因此需要对该字符串进行处理，通过观察可以发现，这里面的信息分布是有规律可循的，导演与其他信息用了三个空格分割，程序中表示为 ‘\xa0\xa0\xa0’，而地区、年份、类别三个信息又可以通过 ‘\xa0/\xa0’ 进行分割，自然而然就拿到了电影的详细信息，最后将电影详细信息用一个字典结构返回。

程序的核心部分都写好了，在 get\_movies\_list(movies\_soup) 函数中，直接对传入的 movies\_soup 进行遍历，对其中每个元素都使用get\_movie\_details() 获取详细信息，并压入 movies\_list 中返回。在主函数中直接将 movies\_list 打印出来，就得到了图 3.21 的结果。

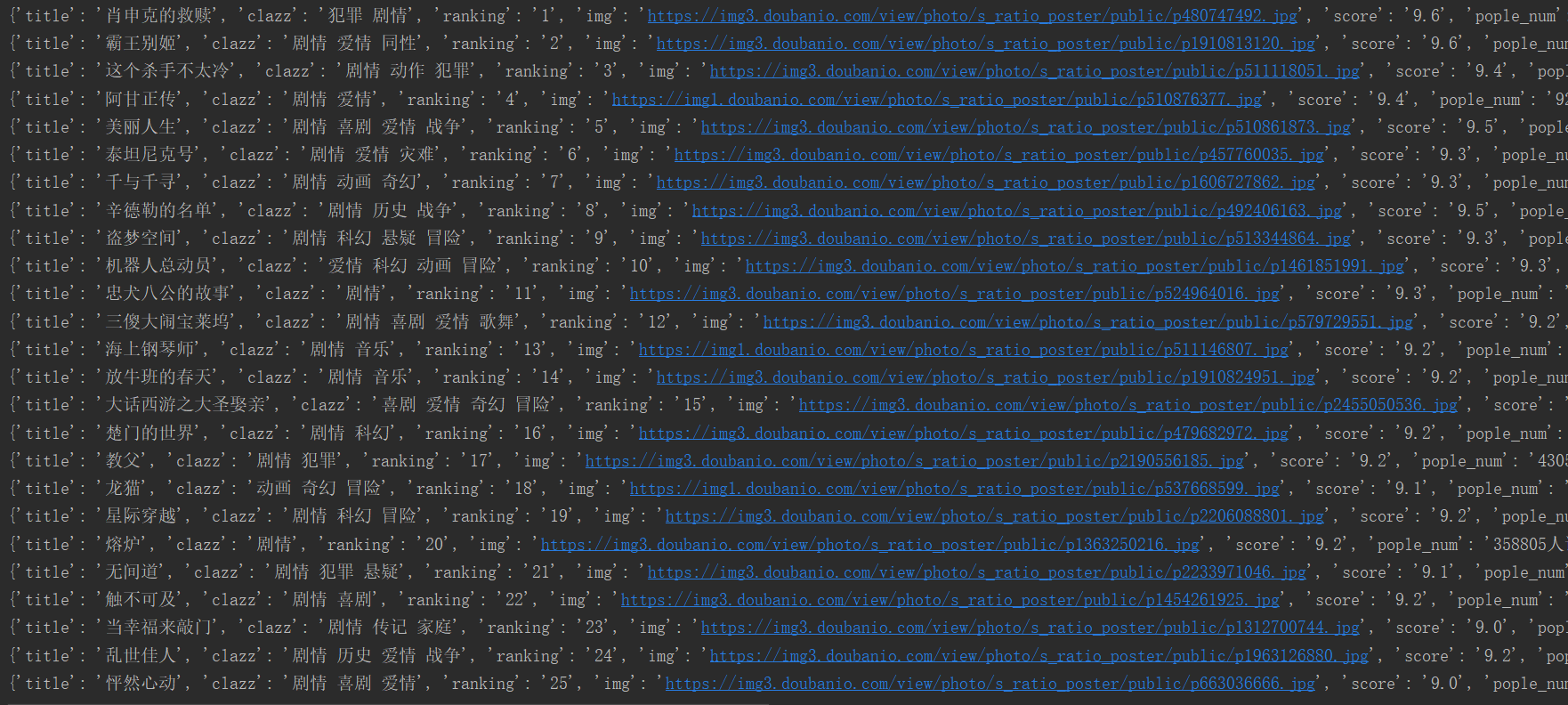


图 3.21 豆瓣爬虫程序运行结果

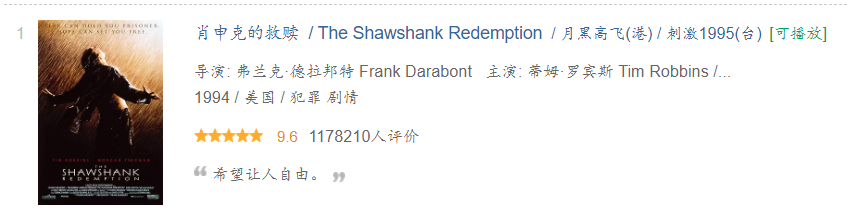


图3.22 <li> 标签对应的内容

可能你已经发现了上述程序的缺点，它只能抓取当前页面内容，而豆瓣 Top250 总共有 10 页，难道抓取每一页的时候都要去手动替换 url 吗？答案肯定不是这样子的，下面我们再一起去分析分析分析网页信息。

进入浏览器调试工具的 Network 标签页，清空请求列表信息，然后回到页面上点击第 2 页，查看调试工具的变化信息，第一个请求信息如图 3.23 所示，可以发现在我们之前的基础上加上了两个数据信息，分别为 start 和 filter，start 表示当前页第一条数据从哪个地方开始，也就是给后台接口传入了一个偏置信息，这和 MySql 分页的原理是一样的；而 filter 没有传值，我们暂且不管它。

实际上这个变化从地址栏也可以直接看到，为了验证我们的分析结果，你可以尝试将 start 的值修改为 0 ，查看页面呈现的结果是不是豆瓣 Top250 的首页信息。

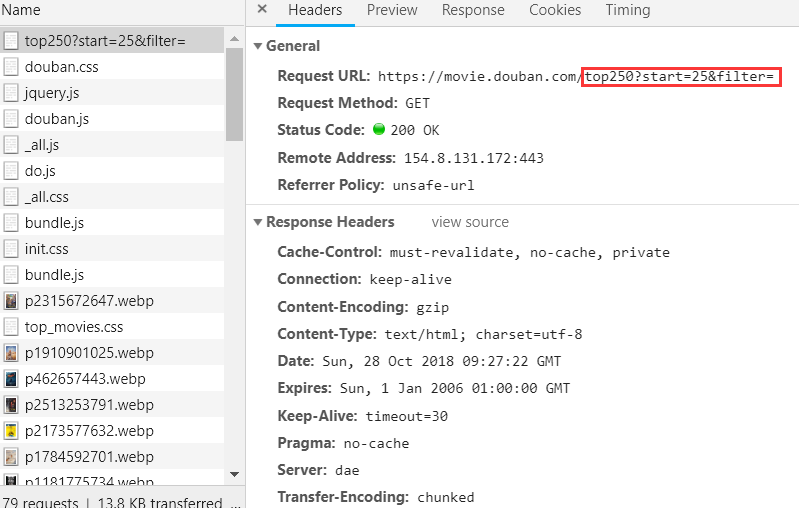


图3.23 豆瓣 Top250 的分页信息

上面分析明白之后，现在就可以修改原来的程序以适应分页的方式了，修改后的 main() 函数如下所示，现在就可以从控制台看到豆瓣 Top250 的所有信息了。

def main():  
 movies\_list = []  
 for i in range(0, 10):  
 url = 'https://movie.douban.com/top250?start=' + str(i\*25) + '&filter='  
 html = get\_html(url)  
 soup = parse\_html(html)  
 movies\_soup = get\_movies\_soup(soup)  
 temp\_list = get\_movies\_list(movies\_soup)  
 movies\_list.extend(temp\_list)  
 return movies\_list

#### 3.3.2 使用 Scrapy 爬虫

##### 3.3.2.1 Scrapy 简介

本来想写一个从零开始的 Scrapy 教程，但是网上查询一番，发现官方已经有了入门案例，并且介绍的很详细，至少从官方教程可以学会如何使用 Scrapy，下面对 Scrapy 的架构在做一个深入的介绍。

图 3.24 是 Scrapy 的大体架构，可以看出 Scrapy 有Scrapy Engine（Scrapy 引擎）、Scheduler（调度器）、Downloader（下载器）、Spiders（爬取器）、Item Pipeline（项目管道）、Downloader middlewares（下载器中间件）、Spider middlewares（爬取器中间件）、Scheduler middlewares（调度器中间件）共 8 个组件，这些组件各司其职，保证了爬虫程序的正确运行，根据具体情况我们可以选择舍弃不同的组件，不是每个工程都完全需要这些组件的，但是 Scrapy Engine（Scrapy 引擎）和Spiders（爬取器）是必备的。

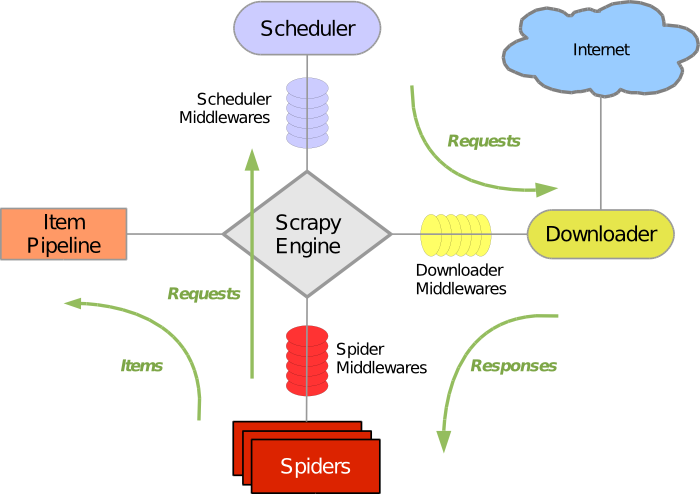


图3.24 Scrapy 架构

Scrapy 引擎用于控制整个整个系统的数据流程，并进行事务处理的触发，相当于司令部。

Scheduler（调度器）负责从 Scrapy 引擎请求并将它们排序加入队列，以便之后引擎请求他们时提供给引擎。

Downloader（下载器）负责获取页面数据并提供给引擎，而后提供给spider（爬取器）。

Spiders（爬取器）是用户自己定义用来解析网页并抓取制定URL返回的内容的类，每个爬取器负责一个网站或者一类网站，即用来定义特定网站的抓取和解析规则。对于Spiders 整个抓取的周期如下：

1. 首先获取第一个 url 的初始请求，当请求返回后调取一个回调函数。第一个请求是通过调用 start\_requests() 方法。该方法默认从 start\_urls 中的 url 中生成请求，并以  [parse](https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/spiders.html#scrapy.spider.Spider.parse)  为回调函数生成  [Request](https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/request-response.html#scrapy.http.Request) 。
2. 在回调函数内分析返回的(网页)内容，返回  [Item](https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/items.html#scrapy.item.Item)  对象或者  [Request](https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/request-response.html#scrapy.http.Request)  或者一个包括二者的可迭代容器。 返回的 Request 对象之后会经过 Scrapy 处理，下载相应的内容，并调用设置的 callback 函数(函数可相同)。
3. 在回调函数内，您可以使用 [选择器(Selectors)](https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/selectors.html#topics-selectors) (您也可以使用 BeautifulSoup 等网页解析器) 来分析网页内容，并根据分析的数据生成 item。
4. 最后，从蜘蛛返回的项目通常会进驻到项目管道。

Item Pipeline（项目管道）负责被 spider 提取出来的 item。常见的处理有清洗、验证解析到的数据、检查是否有重复数据、持久化等。

Downloader middlewares（下载器中间件）是在引擎及下载器之间的特定钩子，主要是处理Scrapy引擎与下载器之间的请求及响应，其提供了一个简便的机制，通过插入自定义代码来扩展Scrapy功能。

Spider middlewares（爬取器中间件）是在引擎及爬取器之间的特定钩子，主要工作是处理 spider 的响应输入和请求输出，和下载器中间件一样，也提供了通过插入自定义代码来扩展Scrapy功能的机制。

Scheduler middlewares（调度器中间件）调度中间件是介于 Scrapy 引擎和调度之间的中间件，主要工作是处从 Scrapy 引擎发送到调度的请求和响应。

##### 3.3.2.2 使用 Scrapy 抓取豆瓣 Top250

首先我们通过命令行工具创建一个 douban 项目，命令为：scrapy startproject douban，该命令会自动帮我们创建好目录，我们只需要修改 items.py 和 spiders 文件夹下的 douban\_spide.py的内容即可。

items.py 中的内容如下所示，它是一个简单的容器，保存了爬去得到的数据。

# -\*- coding: utf-8 -\*-  
  
# Define here the models for your scraped items  
#  
# See documentation in:  
# https://doc.scrapy.org/en/latest/topics/items.html  
  
import scrapy  
  
"""  
Item 对象是种简单的容器，保存了爬取到得数据。  
Item使用简单的class定义语法以及Field对象来声明。  
"""  
  
  
class DoubanItem(scrapy.Item):  
 # 排名  
 ranking = scrapy.Field()  
  
 # 电影名称  
 title = scrapy.Field()  
  
 # 评分  
 score = scrapy.Field()  
  
 # 评论人数  
 pople\_num = scrapy.Field()  
  
 # 导演  
 director = scrapy.Field()  
  
 # 年份  
 year = scrapy.Field()  
  
 # 地区  
 area = scrapy.Field()  
  
 # 类型  
 clazz = scrapy.Field()  
  
 # 一句话描述  
 decsription = scrapy.Field()

在 douban\_spider.py 中，编写我们的爬虫逻辑，然后将抓取盗的内容生成 item。

#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-  
# @File : douban\_spider.py  
# @Desc : 抓取豆瓣 Top250  
  
from scrapy.spider import Spider  
from douban.items import DoubanItem  
from scrapy import Request  
from bs4 import BeautifulSoup  
  
  
class douban\_spider(Spider):  
  
 count = 1  
  
 # 爬虫启动命令  
 name = 'douban'  
  
 headers = {  
 'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36',  
 }  
  
 def start\_requests(self):  
 url = 'https://movie.douban.com/top250'  
 yield Request(url, headers=self.headers)  
  
 # 处理爬取的数据  
 def parse(self, response):  
  
 print('第', self.count, '页')  
 self.count += 1  
  
 item = DoubanItem()  
 soup = BeautifulSoup(response.text, 'html.parser')  
  
 # 选出电影列表  
 movies = soup.select('#content div div.article ol li')  
  
 for movie in movies:  
 item['title'] = movie.select('.title')[0].text  
 item['ranking'] = movie.select('em')[0].text  
 item['score'] = movie.select('.rating\_num')[0].text  
 item['pople\_num'] = movie.select('.star span')[3].text  
  
 # 包含导演、年份、地区、类别  
 info = movie.select('.bd p')[0].text  
 director = info.strip().split('\n')[0].split('   ')  
 yac = info.strip().split('\n')[1].strip().split(' / ')  
  
 item['director'] = director[0].split(': ')[1]  
 item['year'] = yac[0]  
 item['area'] = yac[1]  
 item['clazz'] = yac[2]  
  
 if len(movie.select('.inq')) is not 0:  
 item['decsription'] = movie.select('.inq')[0].text  
 else:  
 item['decsription'] = 'None'  
 yield item  
  
 next\_url = soup.select('.paginator .next a')[0]['href']  
 if next\_url:  
 next\_url = 'https://movie.douban.com/top250' + next\_url  
 yield Request(next\_url, headers=self.headers)

完成上述工作之后，使用命令行工具进入到项目文件夹下面，然后运行命令“scrapy crawl douban -o movies.csv”，待程序运行完毕，就会发现豆瓣 Top250 的信息就被放入到 movies.csv 文件中去了，其内容如图 3.25 所示。

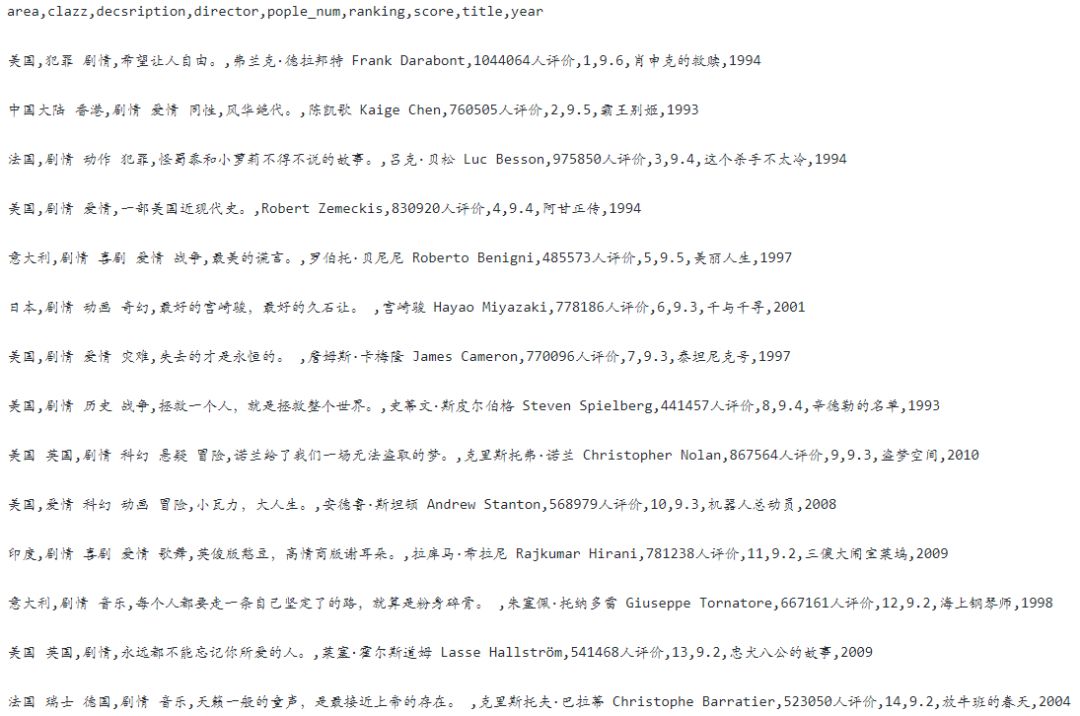


图3.25 Scrapy 抓取结果

上面的item 很简单，不浪费篇幅介绍，可以看到 douban\_spider 前面是基本的配置项，包括爬虫启动命令、请求头、初始 url。

重点部分是 parse 函数，其中大部分代码通过选择器过滤我们想要的信息，这部分与 3.3.1 节中的程序类似，不同的是此处通过 yield 生成新的请求，我们这里的 url 同样可以自己构造，程序的结构很简单，yield 每生成一个 Request 就会被加入到待爬取队列中，而当队列为空时，爬虫就结束了。

我们不需要自己写文件操作相关的程序，而是采用命令行指定输出文件，Scrapy 就会自动将抓取结果写入我们指定的文件，现在是不是就能感受到使用框架的便利性了？

**参考资料**：<https://scrapy-chs.readthedocs.io/zh_CN/0.24/intro/tutorial.html>

### 3.4 抓取审计署新闻

有了上面的基础，下面我们开始获取经济责任审计领域的知识，这里我们选择抓取国家审计署官方网站的内容，图3.4.1是网站首页的截图。我们可以看到，在网站首页右上角右上角有一个搜索框，可以在这个搜索框内输入具体的关键字来过滤我们想要的知识，我们充分利用好这个搜索框就省去了很多不必要的工作，因为我们所需要的经济责任审计方面的知识仅仅属于审计署网站中的一小部分。

图3.4.1 审计署网站首页

在搜索框中输入“经济责任审计”后点击搜索按钮，发现浏览器会打开一个新的窗口，如图3.4.2所示。网页会按与搜索关键字相关度的顺序列出相关内容，每页显示十条，当然你可以通过右侧的高级搜索来修改这些搜索条件，不过这些都不是我们所关心的内容。

图3.4.2 搜索结果

打开控制台，点到Network选项卡下，再点击一下页面中的搜索按钮，这时你就会发现如图3.4.3所展示的内容，幸福来的如此突然。

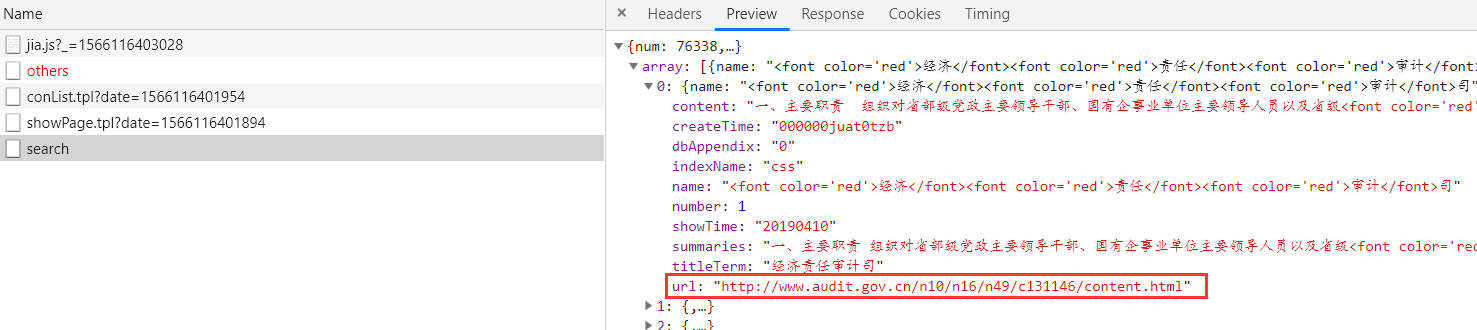


图3.4.3请求信息详情

从图3.4.3中可以看到，审计署网站直接提供了一个搜索接口，该接口的地址为<http://xinxi.audit.gov.cn:8180/searchweb/search>，页面上展示的搜索结果就是这个接口所返回的信息，里面有我们最想要的每条新闻所对应的url，有了url我们下一步就可以获取新闻的详细内容了。下面我们通过代码来验证我们上面的分析是否正确。

import requests  
import json  
  
URL = 'http://xinxi.audit.gov.cn:8180/searchweb/search'  
POST\_DATA = {  
 'fullText': '经济责任审计',  
 'pageSize': 10,  
 'pageNow': 1,  
 'sortType': 0,  
 'searchType': 0,  
 'cateId': '',  
 'keywordNavigation': 1,  
 'checkSearch': 1  
}  
HEADERS = {  
 'Referer': 'www.baidu.com',  
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) '  
 'Chrome/74.0.3729.131 Safari/537.36'  
}  
  
  
def get\_news():  
 res = requests.post(URL, POST\_DATA, HEADERS)  
 news = json.loads(res.text)  
 return news.get('array')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print(get\_news())

运行这个脚本，可以发现控制台的输出结果和我们预期的是一样的了，是一个包含了十条新闻概要的列表，获取审计署相关新闻列表的工作已经算完成了，下一步就是抓取具体新闻的内容了。

<http://www.audit.gov.cn/n10/n16/n49/c131146/content.html>（列表第一项url）可以看到这是经济责任审计司的一项公告，现在通过程序将其中的内容抓取出来并存储。

图3.4.4 新闻详细内容

import requests  
from bs4 import BeautifulSoup  
  
CONTENT\_TABLE\_INDEX = 5  
  
  
def get\_soup(url):  
 res = requests.get(url)  
 res.encoding = "utf-8"  
 soup = BeautifulSoup(res.text, "html.parser")  
 return soup  
  
  
def get\_title(soup):  
 title = soup.select("table")[CONTENT\_TABLE\_INDEX].select("span")[0]  
 return title.text  
  
  
def get\_content(soup):  
 ps = soup.select("table")[CONTENT\_TABLE\_INDEX].select("p")  
 content = ""  
 for p in ps:  
 content += p.text + "\n"  
 return content  
  
  
def get\_news\_info(url):  
 soup = get\_soup(url)  
 result = {}  
 result.update({"title": get\_title(soup)})  
 result.update({"content": get\_content(soup)})  
 return result  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print(get\_news\_info("http://www.audit.gov.cn/n10/n16/n49/c131146/content.html"))

结合前文的内容，我们现在就可以写出一个比较完整的抓取审计署网站中与经济责任审计内容相关的文章了。

def save\_news(news):  
 f = open(".\\news\\" + news["title"] + ".txt", "w")  
 f.write(news["content"])  
 f.flush()  
 f.close()  
  
  
 def main():  
 for i in range(1, 500):  
 error\_set = set()  
 POST\_DATA.update({"pageNow": i})  
 news\_list = get\_news()  
 for news in news\_list:  
 url = news.get("url")  
 print(url)  
 try:  
 news\_info = get\_news\_info(url)  
 save\_news(news\_info)  
 except Exception as e:  
 error\_set.add(url)  
 print(e, url)

上面只是一个简单的示例，我们后文还会抓取其他网站的内容（比如互动百科、维基百科等等），不过抓取经济责任审计网站内容是我们的第一步，所有的工作都从这里开始。爬虫工作更多的是在于实践，上述程序在一些条件下工作的也并不好，你可以在此基础上进行改进，或者也可以使用Scrapy爬虫框架来实现我们的需求。

### 3.4 附录

#### 正则表达式简介

    正则表达式是用来匹配字符串的一系列匹配符，具备简介高效的特点，在很多语言中都有支持（java、python、javascript、php等等）。在windows的cmd命令中也同样支持，例如使用命令dir j\*，那么只会罗列出所有以j开头的文件和文件夹。

#### 正则表达式基本语法

    正则表达式在在不同语言的支持语法略有不同，本文采用js的进行说明。js中使用正则表达式的方法为str.match(/表达式/)，即需要加两个斜杠。以下所有的代码段第一行为代码，第二行为返回结果，实验是在chrome控制台进行的。

    一直认为最好的学习方式就是实际操作，理论谁都能讲一大堆，但是实际做没做出来还真不知道。一个奇葩现象就是教软件工程的老师可能并没有在软件行业待过。

##### 普通匹配符

    普通匹配符能匹配与之对应的字符，默认区分大小写。

"Hello Regx".match(/H/)

["H", index: 0, input: "Hello Regx", groups: undefined]

##### 正则标记符

* i ：不区分大小写
* g ：全局匹配
* m ：多行匹配（暂不管它，我用的少）

    参数直接加在最后一个斜杠的后面，比如"Hello Regx".match(/regx/i)，可以加多个参数。

"Hello Regx".match(/regx/i)

["Regx", index: 6, input: "Hello Regx", groups: undefined]

    之前是表达式一旦匹配成功，就不再向字符串后面查找了，加上g后，表示进行全局查找。最后返回的是一个数组。

"Hello Regx".match(/e/g)

(2) ["e", "e"]

##### 多匹配符

* \d ：匹配数字，即0~9
* \w ：匹配数字、字母、下划线
* . ：匹配除换行的所有字符

    需要注意的是，上面所有的匹配符都只能匹配一个字符。

"Hello 2018".match(/\d/g)

*// 使用\d,匹配字符串中的所有数字*

(4) ["2", "0", "1", "8"]

"Hello 2018".match(/\w/g)

*// 使用\w，匹配所有的数字和字母，需要注意没有匹配到空格*

(9) ["H", "e", "l", "l", "o", "2", "0", "1", "8"]

"Hello 2018".match(/./g)

*// 使用.，匹配所有字符，包括空格*

(10) ["H", "e", "l", "l", "o", " ", "2", "0", "1", "8"]

"Hello 2018".match(/\d\w./g)

*// 分析一下这个为什么匹配到的是201，*

*// 首先\d找到第一个数字2，匹配成功，紧接着\w匹配到0，然后.匹配到1*

*// 整个正则表达式匹配成功，返回201*

["201"]

"Hello 20\n18".match(/\d\w./g)

*// 这里匹配不成功，因为.不能匹配换行符，所以返回null*

null

"Hello 2018".match(/\w.\d/g)

*// 首先看这个正则式，\w.\d，它要求最后一个字符是数字*

*// \w.能一直匹配到空格，但是因为得满足\d，所以第一个匹配成功的是0 2*

*// 因为是全局匹配，所以会接着匹配后面的018，也匹配成功*

(2) ["o 2", "018"]

##### 自定义匹配符

    比如中国的手机号都是以1开头，第二位只能是3、4、5、7、8，第3位只要是数字就行。如何匹配这样的字符串？

* [] ：匹配[]中的任意一个字符

"152".match(/1[34578]\d/)

*// 第二个字符可以选择中括号中的任意一个*

["152", index: 0, input: "152", groups: undefined]

    如果在[]添加了^,代表取反。即[^]表示除了中括号中的字符都满足。

"152".match(/1[^34578]\d/)

null

"1a2".match(/1[^34578]\d/)

*// 只要不是[]中的字符，都满足，包括回车符*

["1a2", index: 0, input: "1a2", groups: undefined]

##### 修饰匹配次数

    我们的手机号有11位，除了前2位有要求，其他9位度没有要求，那么是不是正则表达式就应该这样写呢？

1[^34578]\d\d\d\d\d\d\d\d\d

    很明显，这样写太麻烦，肯定有更好的方式，这里就可以修饰一下匹配次数啦。

* ? ：最多出现1次
* + ：至少出现1次
* \* ：出现任意次数
* {} ：分下面四种情况
  + {n}代表前面的匹配符出现n次
  + {n, m}出现次数在n~m之间
  + {n, }至少出现n次
  + {, m}最多出现m次

    例子很简单，一看就懂，不浪费时间。

"15284750845".match(/1[34578]\d{9}/)

["15284750845", index: 0, input: "15284750845", groups: undefined]

"15".match(/1[34578]\d?/)

["15", index: 0, input: "15", groups: undefined]

"152".match(/1[34578]\d?/)

["152", index: 0, input: "152", groups: undefined]

"152".match(/1[34578]\d+/)

["152", index: 0, input: "152", groups: undefined]

"15".match(/1[34578]\d+/)

null

##### 完整匹配

    按照上面的写法会出现下面的问题。

"ya15284750845".match(/1[34578]\d{9}/)

*// 不是电话号码，也能匹配成功，需要进一步改进*

["15284750845", index: 2, input: "ya15284750845", groups: undefined]

* ^ ：在[]中代表取反，但在外面代表从开始匹配

"ya15284750845".match(/^1[34578]\d{9}/)

*// 现在就能从一开始匹配而且还得符合正则式才算匹配成功*

null

*// 但是依旧会出现下面的问题*

"1528475084523255".match(/^1[34578]\d{9}/)

*// 不是电话号码也能匹配成功，还要改进*

["15284750845", index: 0, input: "1528475084523255", groups: undefined]

* $ ：代表持续匹配到结束

"1528475084523255".match(/^1[34578]\d{9}$/)

*// 现在就能保证正确了，有^表示从开始匹配；*

*// 有$表示持续匹配到结束，即完全匹配*

null

*/\**

*需要注意的是，一个字符串从开始匹配和从结束匹配都没问题，*

*不代表整个字符串就没问题，比如15284750845-15284750845*

*这个字符串从开始和从结束匹配都能成功，但实际上是错的*

*\*/*

##### 特殊符号

    到这里发现正则表达式确实很强大，仅仅几个简单的符号就能匹配字符串，但是如果我们要匹配的字符本身就是前面用到的符号怎么办呢？

* 匹配像$、^等特殊符号时，需要加转义字符\

"1.".match(/./)

*//因为.能匹配除换行的所有字符，所以匹配到1*

*//但实际上我们想匹配.这个字符*

["1", index: 0, input: "1.", groups: undefined]

"1.".match(/\./)

*// 只需要加一个转义字符就可以了，其他类似*

[".", index: 1, input: "1.", groups: undefined]

##### 条件分支

    比如现在想匹配图片的文件名，包括jpg、png、jpeg、gif等等，这是多个选项，所以需要像编程语言一样，应该具备条件分支结构。

* | ：条件分支
* () ：有两层含义
  + 括号中的内容成为一个独立的整体
  + 括号的内容可以进行分组，单独匹配，若不需要此功能，则(?:)

"1.jpg".match(/.+\.jpe?g|gif|png/)

*// 这样就可以满足条件分支了,不过下面又出问题了*

["1.jpg", index: 0, input: "1.jpg", groups: undefined]

"1.png".match(/.+\.jpe?g|gif|png/)

*// 这里没有匹配到.和前面的文件名*

["png", index: 2, input: "1.png", groups: undefined]

*/\**

*其实我们想告诉它的是，.和后面的每一个条件分支的值都是一个独立的整体*

*但是它把.+\.jpe?g、gif、png当成了各自独立的整体*

*我们并不想让它这样切分，所以我们来告诉它怎么分才是正确的*

*\*/*

"1.png".match(/.+\.(jpe?g|gif|png)/)

*// 现在可以匹配成功了，但是它多匹配了一个*

*// 因为括号的内容可以进行分组，单独匹配*

(2) ["1.png", "png", index: 0, input: "1.png", groups: undefined]

*// 所以最终写法如下*

"1.png".match(/.+\.(?:jpe?g|gif|png)/)

["1.png", index: 0, input: "1.png", groups: undefined]

##### 贪婪与懒惰

*// 首先看一个例子*

"aabab".match(/a.\*b/)

["aabab", index: 0, input: "aabab", groups: undefined]

*/\**

*上面的匹配没有什么问题，但实际上aab也是可以的*

*也就是aab也是符合条件的，那又是为什么呢？*

*\*/*

    因为在正则表达式中，默认是贪婪模式，尽可能多的匹配，可以在修饰数量的匹配符后面添加?，则代表懒惰。

*// like this (^\_\_^)*

"aabab".match(/a.\*?b/)

["aab", index: 0, input: "aabab", groups: undefined]

    到这里应该就差不多了，再深入的，就自我查询知识了。