在上一课时我们学习了 Selenium 的基本用法，本课时我们就来结合一个实际的案例来体会一下 Selenium 的适用场景以及使用方法。

准备工作

在本课时开始之前，请确保已经做好了如下准备工作：

- 安装好 Chrome 浏览器并正确配置了 ChromeDriver。
- 安装好 Python（至少为 3.6 版本）并能成功运行 Python 程序。
- 安装好了 Selenium 相关的包并能成功用 Selenium 打开 Chrome 浏览器。

适用场景

在前面的实战案例中，有的网页我们可以直接用 requests 来爬取，有的可以直接通过分析 Ajax 来爬取，不同的网站类型有其适用的爬取方法。

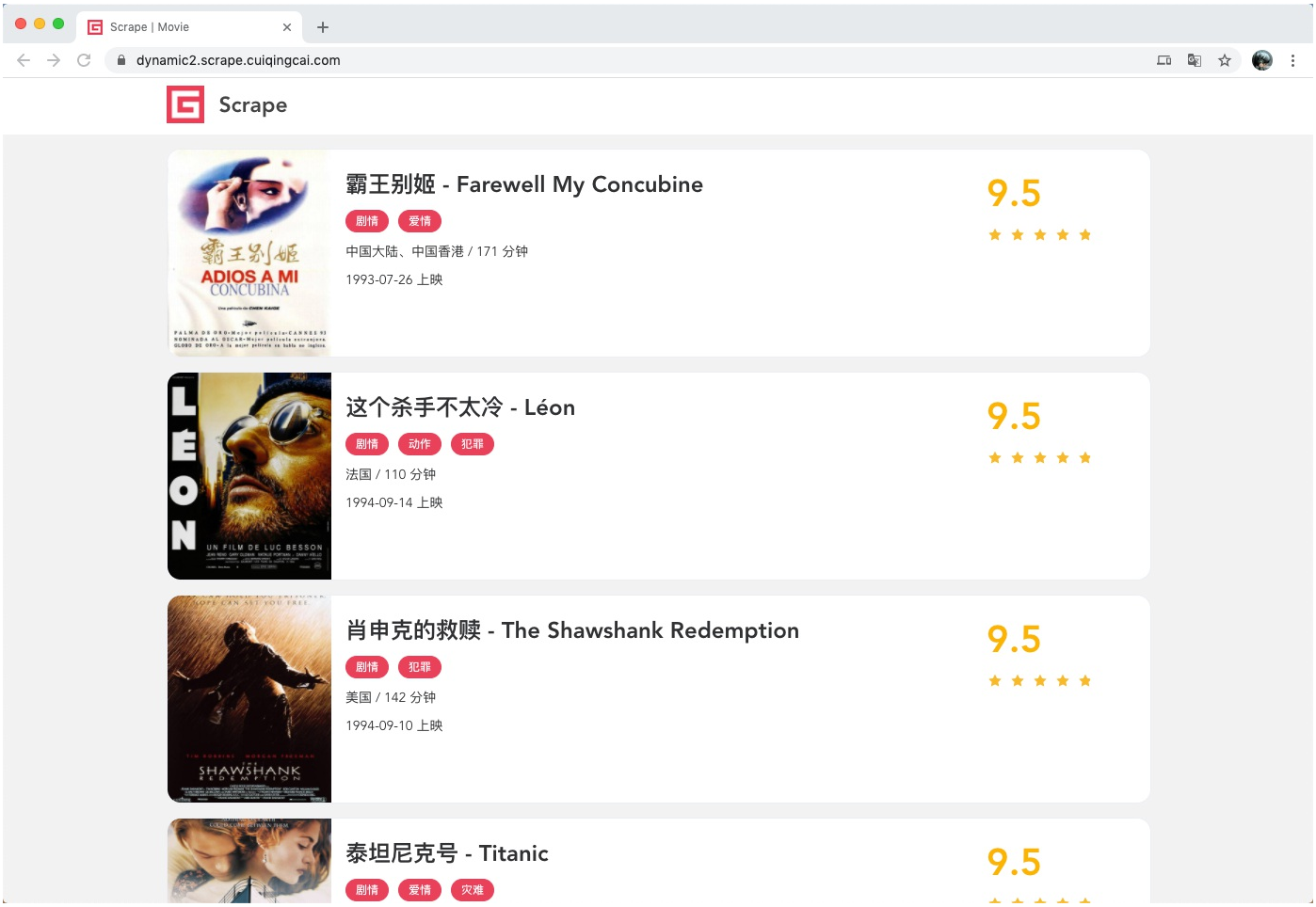
Selenium 同样也有其适用场景。对于那些带有 JavaScript 渲染的网页，我们多数情况下是无法直接用 requests 爬取网页源码的，不过在有些情况下我们可以直接用 requests 来模拟 Ajax 请求来直接得到数据。

然而在有些情况下 Ajax 的一些请求接口可能带有一些加密参数，如 token、sign 等等，如果不分析清楚这些参数是怎么生成的话，我们就难以模拟和构造这些参数。怎么办呢？这时候我们可以直接选择使用 Selenium 驱动浏览器渲染的方式来另辟蹊径，实现所见即所得的爬取，这样我们就无需关心在这个网页背后发生了什么请求、得到什么数据以及怎么渲染页面这些过程，我们看到的页面就是最终浏览器帮我们模拟了 Ajax 请求和 JavaScript 渲染得到的最终结果，而 Selenium 正好也能拿到这个最终结果，相当于绕过了 Ajax 请求分析和模拟的阶段，直达目标。

然而 Selenium 当然也有其局限性，它的爬取效率较低，有些爬取需要模拟浏览器的操作，实现相对烦琐。不过在某些场景下也不失为一种有效的爬取手段。

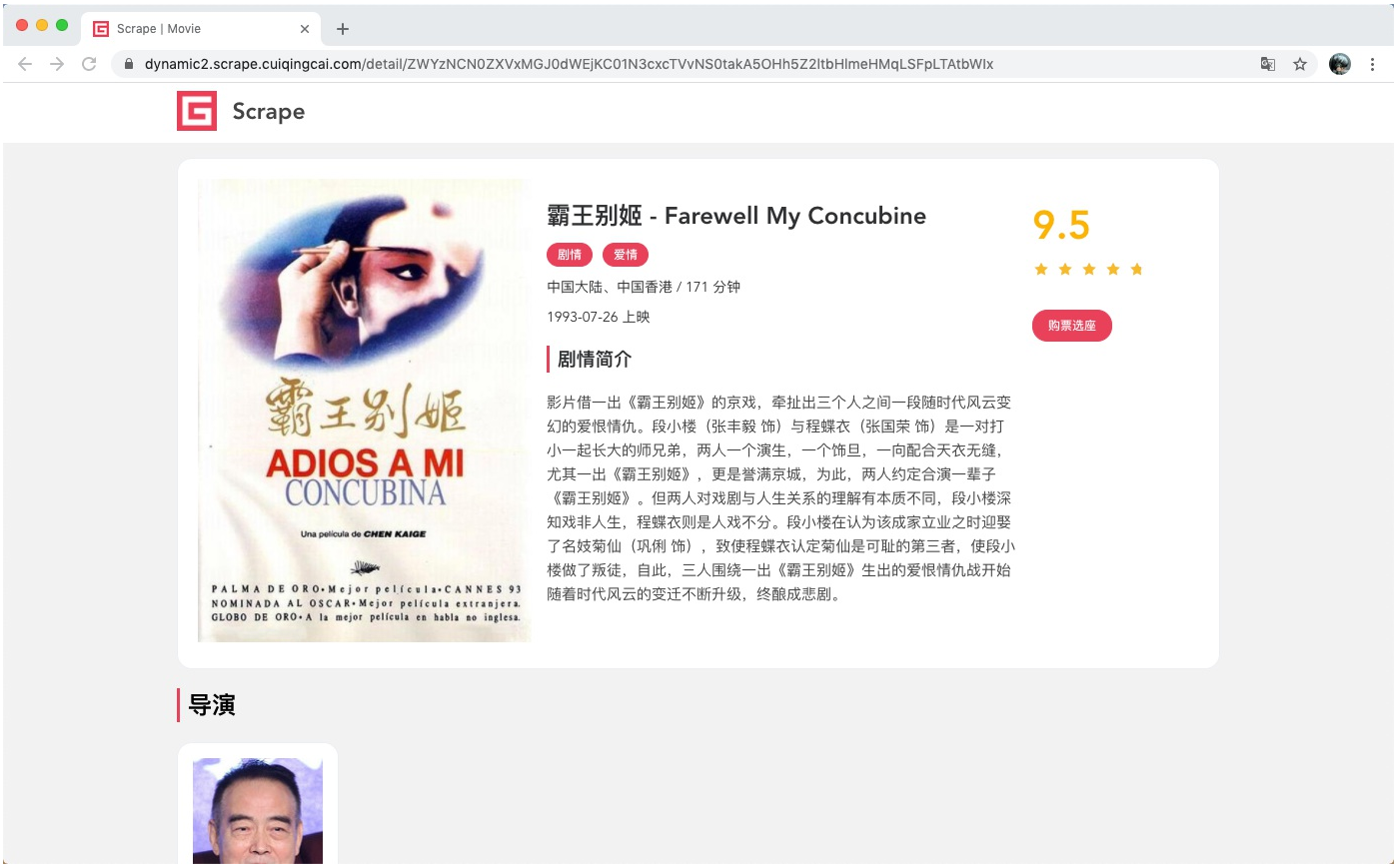
爬取目标

本课时我们就拿一个适用 Selenium 的站点来做案例，其链接为：https://dynamic2.scrape.cuiqingcai.com/，还是和之前一样的电影网站，页面如图所示。



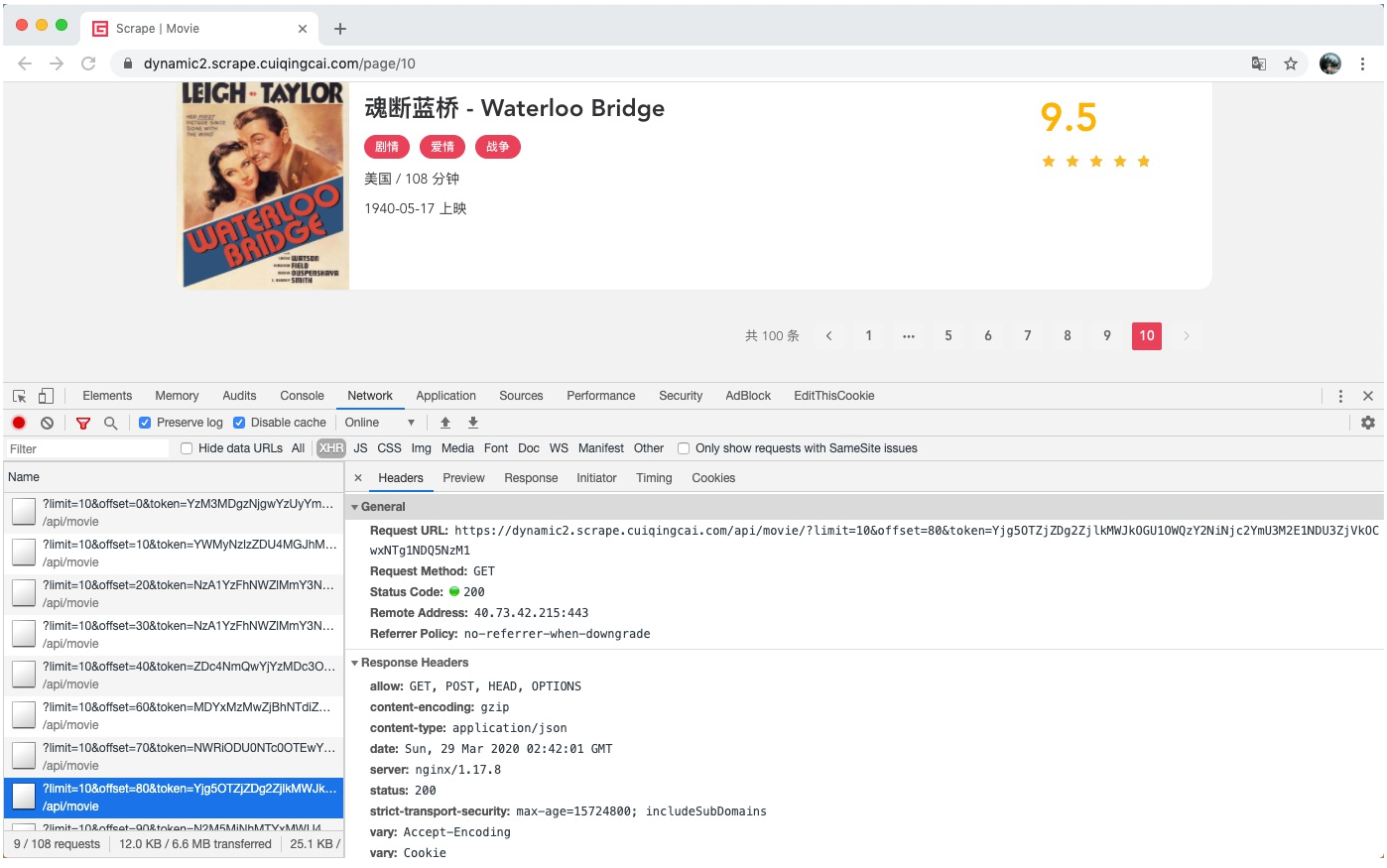
初看之下页面和之前也没有什么区别，但仔细观察可以发现其 Ajax 请求接口和每部电影的 URL 都包含了加密参数。

比如我们点击任意一部电影，观察一下 URL 的变化，如图所示。



这里我们可以看到详情页的 URL 和之前就不一样了，在之前的案例中，URL 的 detail 后面本来直接跟的是 id，如 1、2、3 等数字，但是这里直接变成了一个长字符串，看似是一个 Base64 编码的内容，所以这里我们无法直接根据规律构造详情页的 URL 了。

好，那么接下来我们直接看看 Ajax 的请求，我们从列表页的第 1 页到第 10 页依次点一下，观察一下 Ajax 请求是怎样的，如图所示。



可以看到这里接口的参数比之前多了一个 token，而且每次请求的 token 都是不同的，这个 token 同样看似是一个 Base64 编码的字符串。更困难的是，这个接口还是有时效性的，如果我们把 Ajax 接口 URL 直接复制下来，短期内是可以访问的，但是过段时间之后就无法访问了，会直接返回 401 状态码。

那现在怎么办呢？之前我们可以直接用 requests 来构造 Ajax 请求，但现在 Ajax 请求接口带了这个 token，而且还是可变的，现在我们也不知道 token 的生成逻辑，那就没法直接通过构造 Ajax 请求的方式来爬取了。这时候我们可以把 token 的生成逻辑分析出来再模拟 Ajax 请求，但这种方式相对较难。所以这里我们可以直接用 Selenium 来绕过这个阶段，直接获取最终 JavaScript 渲染完成的页面源码，再提取数据就好

了。

所以本课时我们要完成的目标有：

- 通过 Selenium 遍历列表页，获取每部电影的详情页 URL。
- 通过 Selenium 根据上一步获取的详情页 URL 爬取每部电影的详情页。
- 提取每部电影的名称、类别、分数、简介、封面等内容。

爬取列表页

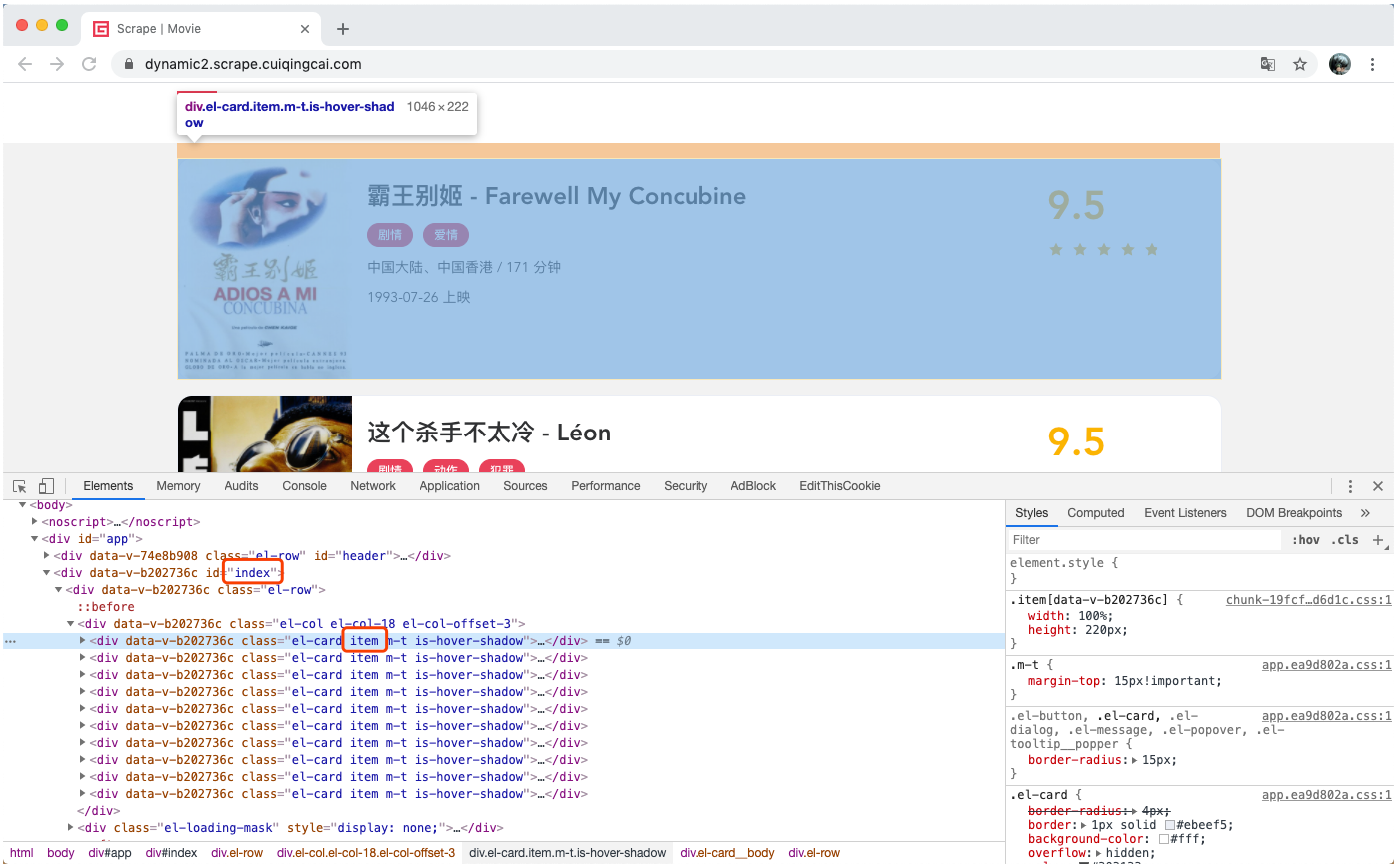
首先要我们要做如下初始化的工作，代码如下：

```
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.wait import WebDriverWait
import logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s: %(message)s')
INDEX_URL = 'https://dynamic2.scrape.cuiqingcai.com/page/{page}'
TIME_OUT = 10
TOTAL_PAGE = 10
browser = webdriver.Chrome()
wait = WebDriverWait(browser, TIME_OUT)
```

首先我们导入了一些必要的 Selenium 模块，包括 webdriver、WebDriverWait 等等，后面我们会用到它们来实现页面的爬取和延迟等待等设置。然后接着定义了一些变量和日志配置，和之前几课时的内容是类似的。接着我们使用 Chrome 类生成了一个 webdriver 对象，赋值为 browser，这里我们可以通过 browser 调用 Selenium 的一些 API 来完成一些浏览器的操作，如截图、点击、下拉等等。最后我们又声明了一个 WebDriverWait 对象，利用它我们可以配置页面加载的最长等待时间。

好，接下来我们就观察下列表页，实现列表页的爬取吧。这里可以观察到列表页的 URL 还是有一定规律的，比如第一页为 https://dynamic2.scrape.cuiqingcai.com/page/1，页码就是 URL 最后的数字，所以这里我们可以直接来构造每一页的 URL。

那么每个列表页要怎么判断是否加载成功了呢？很简单，当页面出现了我们想要的内容就代表加载成功了。在这里我们就可以用 Selenium 的隐式判断条件来判定，比如每部电影的信息区块的 CSS 选择器为 #index .item，如图所示。



所以这里我们直接使用 visibility_of_all_elements_located 判断条件加上 CSS 选择器的内容即可判定页面有没有加载出来，配合 WebDriverWait 的超时配置，我们就可以实现 10 秒的页面的加载监听。如果 10 秒之内，我们所配置的条件符合，则代表页面加载成功，否则则会抛出 TimeoutException 异常。

代码实现如下：

```
def scrape_page(url, condition, locator):
    logging.info('scraping %s', url)
    try:
        browser.get(url)
        wait.until(condition(locator))
    except TimeoutException:
        logging.error('error occurred while scraping %s', url, exc_info=True)
def scrape_index(page):
    url = INDEX_URL.format(page=page)
    scrape_page(url, condition=EC.visibility_of_all_elements_located,
                locator=(By.CSS_SELECTOR, '#index .item'))
```

这里我们定义了两个方法。

第一个方法 scrape_page 依然是一个通用的爬取方法，它可以实现任意 URL 的爬取和状态监听以及异常处理，它接收 url、condition、locator 三个参数，其中 url 参数就是要爬取的页面 URL；condition 就是页面加载的判定条件，它可以是 expected_conditions 的其中某一项判定条件，如 visibility_of_all_elements_located、visibility_of_element_located 等等；locator 代表定位器，是一个元组，它可以通过配置查询条件和参数来获取一个或多个节点，如 (By.CSS_SELECTOR, '#index .item') 则代表通过 CSS 选择器查找 #index .item 来获取列表页所有电影信息节点。另外爬取的过程添加了 TimeoutException 检测，如果在规定时间（这里为 10 秒）没有加载出来对应的节点，那就抛出 TimeoutException 异常并输出错误日志。

第二个方法 scrape_index 则是爬取列表页的方法，它接收一个参数 page，通过调用 scrape_page 方法并传入 condition 和 locator 对象，完成页面的爬取。这里 condition 我们用的是 visibility_of_all_elements_located，代

表所有的节点都加载出来才算成功。

注意，这里爬取页面我们不需要返回任何结果，因为执行完 scrape_index 后，页面正好处在对应的页面加载完成的状态，我们利用 browser 对象可以进一步进行信息的提取。

好，现在我们已经可以加载出来列表页了，下一步当然就是进行列表页的解析，提取出详情页 URL，我们定义一个如下的解析列表页的方法：

```
from urllib.parse import urljoin
def parse_index():
    elements = browser.find_elements_by_css_selector('#index .item .name')
    for element in elements:
        href = element.get_attribute('href')
        yield urljoin(INDEX_URL, href)
```

这里我们通过 find_elements_by_css_selector 方法直接提取了所有电影的名称，接着遍历结果，通过 get_attribute 方法提取了详情页的 href，再用 urljoin 方法合并成一个完整的 URL。

最后，我们再用一个 main 方法把上面的方法串联起来，实现如下：

```
def main():
    try:
        for page in range(1, TOTAL_PAGE + 1):
            scrape_index(page)
            detail_urls = parse_index()
            logging.info('details urls %s', list(detail_urls))
    finally:
        browser.close()
```

这里我们就是遍历了所有页码，依次爬取了每一页的列表页并提取出来了详情页的 URL。

运行结果如下：

```
2020-03-29 12:03:09,896 - INFO: scraping https://dynamic2.scrape.cuiqingcai.com/page/1
2020-03-29 12:03:13,724 - INFO: details urls ['https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIx',
...
'https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWI5', 'https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWE
2020-03-29 12:03:13,724 - INFO: scraping https://dynamic2.scrape.cuiqingcai.com/page/2
...
```

由于输出内容较多，这里省略了部分内容。

观察结果我们可以发现，详情页那一个个不规则的 URL 就成功被我们提取到了！

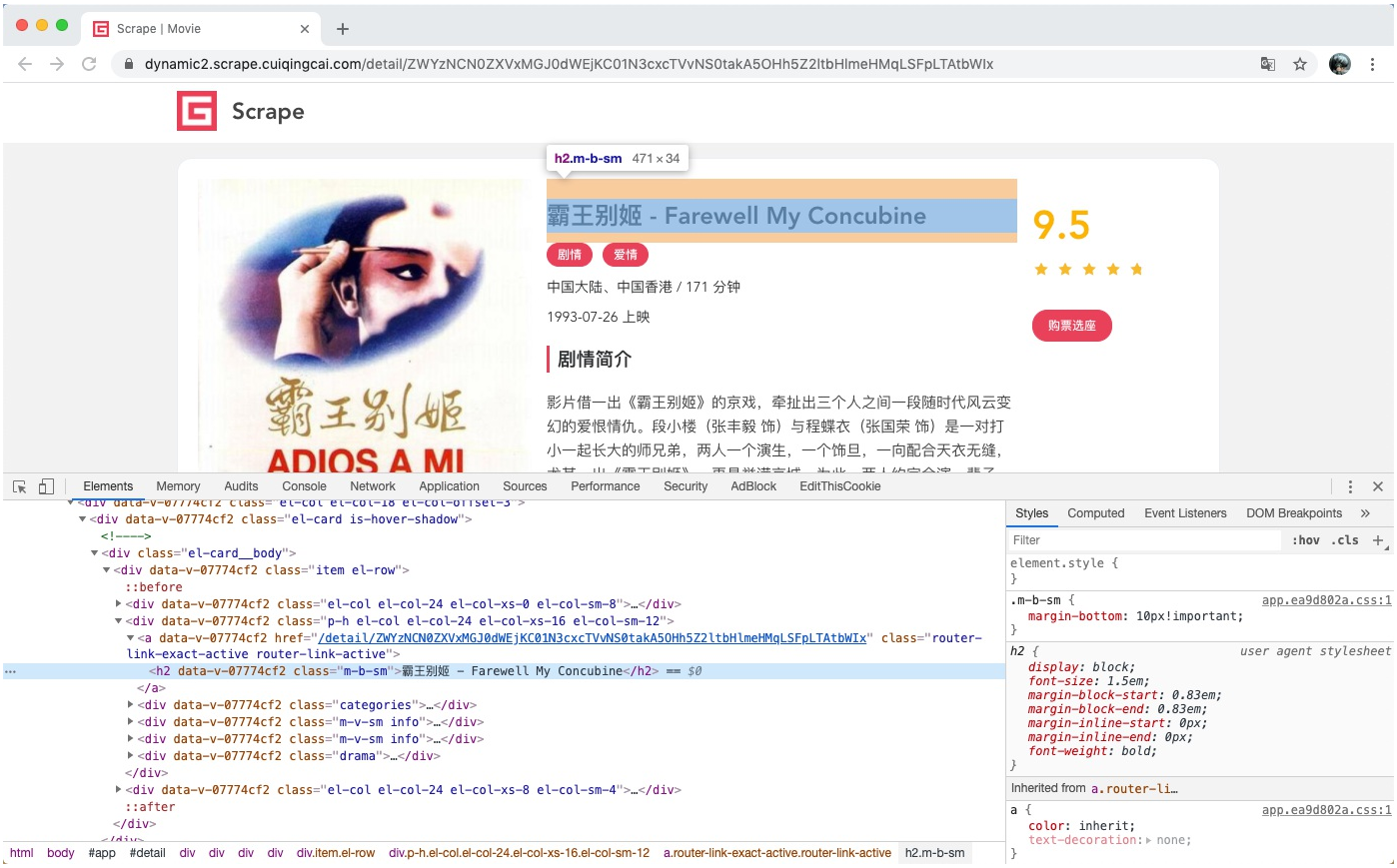
爬取详情页

好了，既然现在我们已经可以成功拿到详情页的 URL 了，接下来我们就进一步完成详情页的爬取并提取对应的信息吧。

同样的逻辑，详情页我们也可以加一个判定条件，如判断电影名称加载出来了就代表详情页加载成功，同样调用 scrape_page 方法即可，代码实现如下：

```
def scrape_detail(url):
    scrape_page(url, condition=EC.visibility_of_element_located,
                locator=(By.TAG_NAME, 'h2'))
```

这里的判定条件 condition 我们使用的是 visibility_of_element_located，即判断单个元素出现即可，locator 我们传入的是 (By.TAG_NAME, 'h2')，即 h2 这个节点，也就是电影的名称对应的节点，如图所示。



如果执行了 scrape_detail 方法，没有出现 TimeoutException 的话，页面就加载成功了，接着我们再定义一个解析详情页的方法，来提取出我们想要的信息就可以了，实现如下：

```
def parse_detail():
    url = browser.current_url
    name = browser.find_element_by_tag_name('h2').text
    categories = [element.text for element in browser.find_elements_by_css_selector('.categories button span')]
    cover = browser.find_element_by_css_selector('.cover').get_attribute('src')
    score = browser.find_element_by_class_name('score').text
    drama = browser.find_element_by_css_selector('.drama p').text
    return {
```

```
        'url': url,
        'name': name,
        'categories': categories,
        'cover': cover,
        'score': score,
        'drama': drama
    }
```

这里我们定义了一个 parse_detail 方法，提取了 URL、名称、类别、封面、分数、简介等内容，提取方式如下：

- URL：直接调用 browser 对象的 current_url 属性即可获取当前页面的 URL。
- 名称：通过提取 h2 节点内部的文本即可获取，这里使用了 find_element_by_tag_name 方法并传入 h2，提取到了名称的节点，然后调用 text 属性即提取了节点内部的文本，即电影名称。
- 类别：为了方便，类别我们可以通过 CSS 选择器来提取，其对应的 CSS 选择器为 .categories button span，可以选中多个类别节点，这里我们通过 find_elements_by_css_selector 即可提取 CSS 选择器对应的多个类别节点，然后依次遍历这个结果，调用它的 text 属性获取节点内部文本即可。
- 封面：同样可以使用 CSS 选择器 .cover 直接获取封面对应的节点，但是由于其封面的 URL 对应的是 src 这个属性，所以这里用 get_attribute 方法并传入 src 来提取。
- 分数：分数对应的 CSS 选择器为 .score，我们可以用上面同样的方式来提取，但是这里我们换了一个方法，叫作 find_element_by_class_name，它可以使用 class 的名称来提取节点，能达到同样的效果，不过这里传入的参数就是 class 的名称 score 而不是 .score 了。提取节点之后，我们再调用 text 属性提取节点文本即可。
- 简介：同样可以使用 CSS 选择器 .drama p 直接获取简介对应的节点，然后调用 text 属性提取文本即可。

最后，我们把结果构造成一个字典返回即可。

接下来，我们在 main 方法中再添加这两个方法的调用，实现如下：

```
def main():
    try:
        for page in range(1, TOTAL_PAGE + 1):
            scrape_index(page)
            detail_urls = parse_index()
            for detail_url in list(detail_urls):
                logging.info('get detail url %s', detail_url)
                scrape_detail(detail_url)
                detail_data = parse_detail()
                logging.info('detail data %s', detail_data)
    finally:
        browser.close()
```

这样，爬取完列表页之后，我们就可以依次爬取详情页，来提取每部电影的具体信息了。

```
2020-03-29 12:24:10,723 - INFO: scraping https://dynamic2.scrape.cuiqingcai.com/page/1
2020-03-29 12:24:16,997 - INFO: get detail url https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIx
2020-03-29 12:24:16,997 - INFO: scraping https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIx
2020-03-29 12:24:19,289 - INFO: detail data {'url': 'https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIx', 'name': '霸王别
2020-03-29 12:24:19,291 - INFO: get detail url https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIy
2020-03-29 12:24:19,291 - INFO: scraping https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIy
2020-03-29 12:24:21,524 - INFO: detail data {'url': 'https://dynamic2.scrape.cuiqingcai.com/detail/ZWYzNCN0ZXVxMGJ0dWEjKC01N3cxcTVvNS0takA5OHh5Z2ltbHlmeHMqLSFpLTAtbWIy', 'name': '这个杀
...
```

这样详情页数据我们也可以提取到了。

数据存储

最后，我们再像之前一样添加一个数据存储的方法，为了方便，这里还是保存为 JSON 文本文件，实现如下：

```
from os import makedirs
from os.path import exists
RESULTS_DIR = 'results'
exists(RESULTS_DIR) or makedirs(RESULTS_DIR)
def save_data(data):
    name = data.get('name')
    data_path = f'{RESULTS_DIR}/{name}.json'
    json.dump(data, open(data_path, 'w', encoding='utf-8'), ensure_ascii=False, indent=2)
```

这里原理和实现方式与 Ajax 爬取实战课时是完全相同的，不再赘述。

最后添加上 save_data 的调用，完整看下运行效果。

Headless

如果觉得爬取过程中弹出浏览器有所干扰，我们可以开启 Chrome 的 Headless 模式，这样爬取过程中便不会再弹出浏览器了，同时爬取速度还有进一步的提升。

只需要做如下修改即可：

```
options = webdriver.ChromeOptions()
options.add_argument('--headless')
browser = webdriver.Chrome(options=options)
```

这里通过 ChromeOptions 添加了 --headless 参数，然后用 ChromeOptions 来进行 Chrome 的初始化即可。

修改后再重新运行代码，Chrome 浏览器就不会弹出来了，爬取结果是完全一样的。

总结

本课时我们通过一个案例了解了 Selenium 的适用场景，并结合案例使用 Selenium 实现了页面的爬取，从而对 Selenium 的使用有进一步的掌握。

以后我们就知道什么时候可以用 Selenium 以及怎样使用 Selenium 来完成页面的爬取啦。