实验七：动态网页数据爬取实战

一、实验基本信息

- **实验名称**：动态网页数据爬取实战
- **实验学时**：2 学时
- **实验课程**：网络爬虫与数据采集/大数据采集技术
- **适用专业**：计算机科学与技术、大数据技术、人工智能、网络工程等
- **实验类型**：验证性+设计性实验

二、实验目的

传统静态网页爬取无法获取 JavaScript 动态渲染、Ajax 异步加载的网页数据，是网络爬虫学习的核心难点。本实验通过两种主流动态网页爬取方案实战训练，帮助学生攻克动态数据采集难题，具体实验目的如下：

1. 掌握动态网页的核心特征，能够区分静态网页与 JS 动态加载网页的差异。
2. 熟练使用浏览器开发者工具 Network 面板，定位网页 Ajax 异步接口，抓取接口 URL、请求方式、请求参数、请求头核心信息。
3. 掌握 Ajax 接口爬取原理，编写 Python 代码实现分页请求、JSON 数据解析、批量数据采集。
4. 掌握 Selenium 自动化爬虫环境配置，理解浏览器渲染原理，运用显式等待机制解决动态元素加载延迟问题。
5. 实现 Selenium 自动化爬取 JS 渲染后的网页数据，完成动态元素精准提取。
6. 对比分析 Ajax 接口爬取与 Selenium 自动化爬取的运行效率、优缺点及适用场景，建立动态爬虫选型思维。
7. 总结动态网页爬取核心技巧，提升复杂网页数据采集的实战能力。

三、实验原理

3.1 动态网页基本原理

动态网页的数据并非随网页源代码一次性加载完成，而是客户端浏览器加载基础 HTML 框架后，通过 JavaScript 代码异步向服务器发送请求，获取数据后动态渲染到

页面中。因此直接请求网页源代码无法获取有效业务数据，必须针对性采用动态爬取方案。常见动态加载方式以 Ajax 异步加载、JS 动态渲染为主。

3.2 Ajax 接口爬取原理

Ajax（异步 JavaScript 和 XML）是网页异步数据请求技术，网页通过后台接口向服务器发送 HTTP 请求，获取 JSON/XML 格式数据后前端渲染展示。该爬取方式核心为**抓包接口直连**：通过开发者工具捕获前端真实请求的 API 接口，模拟浏览器请求参数、请求头，直接向服务器发送请求获取原始 JSON 数据，无需加载完整网页资源，具有高效、轻便的特点。

3.3 Selenium 自动化爬取原理

Selenium 是一款自动化浏览器测试工具，可驱动真实浏览器（Chrome、Edge 等）完成打开网页、等待加载、点击、滚动等人工操作。其原理是**模拟真实用户浏览行为**，完整执行网页 JS 代码、加载全部动态资源，等待页面元素渲染完成后，再解析页面提取数据。针对接口加密、参数复杂、接口难以抓取的网页，该方案兼容性极强。

3.4 显式等待机制原理

动态网页元素加载存在延迟，固定延时等待稳定性差。Selenium 显式等待可设置指定等待时间，持续监测目标元素是否加载完成，元素就绪则立即执行后续操作，超时则抛出异常，有效解决动态元素加载不稳定、爬取报错的问题。

四、实验环境

4.1 硬件环境

计算机一台，Windows/macOS/Linux 系统，网络正常联网

4.2 软件环境

- Python 3.7 及以上版本
- 代码编辑器：PyCharm、VS Code、IDLE 均可
- 浏览器：Google Chrome（推荐）/Edge
- 第三方库：requests、selenium、json、time
- 浏览器驱动：对应浏览器版本的 ChromeDriver/EdgeDriver

五、实验内容与实验步骤

5.1 实验准备：环境配置

1. **安装依赖库**：打开命令行，执行安装命令：

```
pip install requests selenium
```

2. **配置浏览器驱动**：下载与本地 Chrome 浏览器版本匹配的 ChromeDriver，将驱动文件放置在 Python 安装目录或项目根目录，完成环境适配。

3. **验证环境**：编写简易代码测试库与驱动是否正常运行，无报错即为配置成功。

5.2 任务一：识别动态网页，抓取 Ajax 接口信息

选取典型 Ajax 动态加载网页（如新闻列表、商品列表、数据榜单类网页），完成接口抓包分析。

1. 打开目标动态网页，按下 F12 打开浏览器开发者工具，切换至 **Network（网络）** 面板。

2. 勾选「XHR/Fetch」筛选异步请求数据，刷新页面或滑动页面加载更多数据。

3. 在请求列表中筛选业务数据接口，点击对应请求查看详情，记录核心信息：请求 URL、请求方式（GET/POST）、请求参数（页码、条数、时间戳等）、请求头（User-Agent、Referer 等）、响应数据格式。

4. 分析接口分页规则，明确分页参数变化规律，为批量爬取做准备。

5. 记录动态网页核心特征：网页源代码无业务数据、数据通过异步接口动态加载、分页加载无页面刷新。

5.3 任务二：编写 Ajax 接口分页爬取代码

基于抓取的 Ajax 接口信息，编写 Python 代码，实现分页请求、JSON 数据解析、数据批量提取与保存。

1. 导入 requests、json 库，模拟浏览器请求头，规避网站基础反爬机制。

2. 定义基础接口 URL、分页参数，通过循环修改页码参数，实现多页数据请求。

3. 发送 HTTP 请求，获取接口返回的 JSON 格式响应数据。

4. 解析 JSON 字典数据，提取目标字段（如标题、内容、时间、作者、价格等）。

5. 添加异常捕获机制，处理请求超时、接口报错、数据为空等问题。

6. 运行代码，完成多页动态数据批量爬取，保存数据至控制台或本地文件。

5.4 任务三：Selenium 动态渲染爬取实战

针对 JS 全渲染动态网页，使用 Selenium 模拟浏览器加载页面，结合显式等待提取动态数据。

1. 导入 selenium 相关模块，配置浏览器参数（无头模式、禁用弹窗等，可选）。
2. 初始化浏览器驱动，打开目标动态网页。
3. **配置显式等待机制**：设置最大等待时长，通过定位元素 ID、XPATH、CSS 选择器，等待目标动态元素加载完成。
4. 页面元素加载完成后，批量定位并提取网页渲染后的文本、属性等有效数据。
5. 实现分页操作（模拟下滑、点击下一页），完成多页动态数据采集。
6. 关闭浏览器驱动，释放资源，输出爬取结果。

5.5 任务四：两种爬取方案对比分析

统一爬取相同数量、相同内容的网页数据，从**爬取效率、资源占用、代码难度、稳定性、适用场景**五个维度对比两种方案，记录实验数据。

1. 统计两种方案爬取 10 页、50 条数据的耗时时间。
2. 对比 CPU、内存资源占用情况。
3. 分析代码编写难度、反爬适配能力、数据准确率。
4. 总结两种方案的优缺点及适配场景。

六、实验核心代码示例

6.1 Ajax 接口分页爬取核心代码

```
python
import requests
import json

# 请求头模拟浏览器
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0
Safari/537.36"
}

# 接口基础地址（替换为自己抓取的真实接口）
base_url = "https://xxx.com/api/data"
```

```
# 分页批量爬取
def ajax_crawl(page_num):
    data_list = []
    for page in range(1, page_num + 1):
        # 分页参数（根据实际接口修改）
        params = {
            "page": page,
            "size": 10
        }
        try:
            res = requests.get(url=base_url, headers=headers,
params=params, timeout=10)
            res.encoding = "utf-8"
            # 解析 JSON 数据
            json_data = json.loads(res.text)
            # 遍历提取目标数据（根据实际数据结构修改）
            for item in json_data["data"]["list"]:
                info = {
                    "标题": item["title"],
                    "发布时间": item["time"],
                    "内容": item["content"]
                }
                data_list.append(info)
                print("爬取成功：", info)
        except Exception as e:
            print(f"第{page}页爬取失败，错误信息：{e}")
    return data_list

if __name__ == "__main__":
    result = ajax_crawl(5)
    print("数据爬取完成，总条数：", len(result))
```

6.2 Selenium 显式等待爬取核心代码

```
python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time
```

```
# 初始化浏览器
driver = webdriver.Chrome()
driver.maximize_window()
# 显式等待最大时长 10 秒
wait = WebDriverWait(driver, 10)

def selenium_crawl():
    url = "https://xxx.com/dynamic-page" # 替换为目标动态网页
    driver.get(url)
    data_list = []
    try:
        # 等待目标动态元素加载完成
        items =
wait.until(EC.presence_of_all_elements_located((By.CLASS_NAME,
"item")))
        # 遍历提取数据
        for item in items:
            title = item.find_element(By.CLASS_NAME, "title").text
            time_text = item.find_element(By.CLASS_NAME,
"time").text
            data_list.append({"标题": title, "时间": time_text})
            print("爬取数据：", title)
    except Exception as e:
        print("数据提取失败：", e)
    finally:
        driver.quit()
    return data_list

if __name__ == "__main__":
    selenium_crawl()
```

七、实验结果与分析

7.1 实验结果要求

1. 提交 Ajax 抓包截图，包含接口 URL、请求参数、响应 JSON 数据。
2. 提交两种爬取方案的运行代码及成功爬取的数据结果截图。
3. 记录两种方案爬取相同数据的耗时、资源占用数据，填写对比表格。

7.2 方案对比分析

对比维度	Ajax 接口爬取	Selenium 自动化爬取
爬取效率	极高，直接请求接口原始数据，无冗余资源加载	较低，需要加载完整浏览器页面、渲染 JS 资源，耗时更长
资源占用	占用低，仅发送 HTTP 请求、解析数据	占用高，需启动浏览器进程，占用 CPU 和内存
代码难度	需抓包分析接口，参数复杂时调试难度大	无需分析接口，直接解析页面元素，上手简单
稳定性	接口更新、参数加密后会失效，稳定性一般	适配性强，页面小幅修改不影响爬取，稳定性高
适用场景	接口公开、参数简单、需要大批量高效爬取的场景	接口加密、参数复杂、JS 强渲染、反爬严格的网页

八、实验常见问题与解决方案

- **问题 1：Ajax 接口请求返回空数据/403 报错**

解决方案：补充完整请求头（User-Agent、Referer、Cookie），校验请求参数是否与抓包信息一致，排查接口分页、时间戳参数是否失效。

- **问题 2：Selenium 元素定位不到，报错找不到元素**

解决方案：使用显式等待替代固定 sleep 延时，确认元素定位方式（XPATH/CSS）准确，排查页面是否存在 iframe 嵌套、动态弹窗遮挡。

- **问题 3：浏览器驱动启动失败**

解决方案：匹配浏览器与驱动版本，将驱动放置项目根目录，关闭浏览器自动更新，禁用浏览器后台进程。

- **问题 4：分页爬取数据重复/缺失**

解决方案：核对分页参数逻辑，确认每页数据条数、页码起始值，添加数据去重逻

辑。

九、实验报告要求

实验完成后，需独立撰写实验报告，包含以下核心内容：

1. 实验基本信息、实验目的、实验原理简述；
2. 实验环境配置过程、核心操作步骤截图；
3. 完整实验代码、代码注释及运行结果展示；
4. 两种动态爬取方案的对比数据分析；
5. 实验总结：动态网页爬取核心技巧、实验遇到的问题及解决方法、实验心得；
6. 拓展思考：简述动态爬虫的反爬规避思路及技术优化方向。

十、实验思考题

1. 为什么静态网页 requests 直接爬取无法获取动态加载数据？核心原因是什么？
2. Selenium 中显式等待和隐式等待、固定延时 sleep 的区别是什么？各自适用什么场景？
3. 针对接口加密、签名校验的动态网页，如何优化 Ajax 爬取方案？
4. 大批量数据爬取时，如何提升 Selenium 爬虫的运行效率？