深入浅出 CDP (Chrome DevTools Protocol)

Original Clericpy Clericpy 2020年01月15日 00:00

背景

自从 Chrome 59 发布支持 –headless 启动参数以后 (Windows 上是 60 版本), 轻量级浏览器内核就不再是 webdriver 一家独大, 甚至 phantomjs 作者也发文表示不再维护该项目, 国外也有越来越多的文章推荐使用 headless Chrome 代替过去 selenium + webdriver 的方式进行 Web 测试或者爬虫相关工作.
目前国内实际上使用 headless Chrome 的并不少, 只不过目前大量营销号的存在, 导致为了点击量频繁刷文, 进而把早年间 selenium 用作爬虫的旧文章重新翻到读者眼前, 所以遇到各种稀奇古怪的问题, 初学者使用体验较差.
selenium 作为老牌 Web 测试手段闻名已久, 在高级功能 API 层面非常成熟, 后来也加强了对 Chrome headless 模式下 CDP 的支持, 目前依然拥有大量用户在使用.
这里, 简单提一下 selenium + webdriver 方式的一些不足:

- 默认参数启动时很容易被服务端发现
- 性能与 Chrome headless 相比, 较差
- 存在了无数年的内存泄漏问题
- 不像 Chrome 有大厂在背后支撑, 上千 issues 解决不完
- 无法作为完整浏览器使用和调试

简而言之, 都 2020 年了, 不要再抱着 selenium 不放了

概述

CDP

Chrome DevTools Protocol 的简称, 通过 CDP, 可以检查/调试/监听网络流量, Chrome 浏览器的调试工具 Chrome DevTools 使用的也是这套协议, 支持 Chrome, Chromium 等所有基于 Blink 的浏览器.
CDP 官方文档: https://chromedevtools.github.io/devtools-protocol/

交流方式

通过 HTTP, WebSocket 两种方式, 对添加了远程调试接口参数(`--remote-debugging-port=9222`)的浏览器进行远程调试, 大部分功能其实与浏览器手机打开的 devtools 一致

```
1. HTTP 负责总览当前 Tabs 信息
2. 每个 Tab 的对话使用 WebSocket 建立连接, 并接收已开启功能 (enabled domain)
```

复制

Headless Browser

俗称的无头浏览器, 实际上就是没有图形界面的浏览器, 因为省去了视觉渲染的工作, 性能和开销有较大优化, 粗略估计, 原本只能启动十个浏览器的内存, 使用 Headless 模式可以至少启动三倍的数量

常见用途

- 主要还是 Web 测试
- 少数情况会用来做爬虫, 所见即所得的调试体验非常容易上手
- 有一些 Web 自动化的工作, 可以替代自己写扩展或者 tampermonkey JS 脚本, 毕竟权限更高更全面, GUI 模式调试完以后, 无人参与操作的多数情况, 则可以无痛改成 –Headless 模式来提高性能

常见问题

Chrome 浏览器有一个并发连接数的限制. 即对同一个网站, 只允许建立最多 6 个连接(纯静态情况下, 可以看作是 6 个同 domain 的 Tabs). 如果真的遇到超过 6 个连接的需求, 可以通过新开一个浏览器实例来解决.
对于 Linux 来说, 子进程处理不正确会导致出现僵尸进程/孤儿进程, 导致白白浪费资源, 时间长了整台服务器的内存都会垮掉. 常见解决方案有 3 种

a. 调用 Browser.close 功能 gracefully 地关闭浏览器
b. 然后 terminate 子进程后, 记得 wait 一下消息
c. 最后保险起见可以再加个 kill, 虽然实际没什么用
   i. 将 Chrome 守护进程 (Daemon) 与业务代码隔离, 随需要启动对应数量的 Chrome 实例
   ii. 就 Python subprocess 这个内置模块来说, 确定每次关闭的时候执行正确的姿势
   iii. 最简单的其实是找到 chrome 实例的进程 ID 来杀, 毕竟杀死以后, subprocess 那边立刻就结束了

神奇的是, 除了 chrome 实例有僵尸进程, 连 tab 也会存在一些看不见 (**/json** 里那些非 “page” 类型的就是了)或关不掉(僵尸标签页)的 tab 页

i. 目前这种 tab 不确定会不会自己关闭, 访问 B 站遇到过
ii. 以前我处理这种 tab 的方式是给每个 tab 设定一个 lifespan, 异步一个循环, 扫描并关闭那些非 page 类型或者寿命超时了的 tab
iii. 然而 tab 数量多了以后, 反而会出现很多无法关闭的僵尸 tab, 通过 **/json/close** 或者发送 Page.close 事件都无效, 暂时只好重启 chrome 实例来清理

拿来做爬虫还有几个问题没解决

i. chronium 开发团队本着 “你并不是真的特别需要” 原则, 没有动态 UA 和 动态挂代理的开发意向, 毕竟人家也不太希望人们拿它来做爬虫, 只能指望不同代理 IP 启动多个 chrome 实例来解决
ii. 在 “非 headless” 情况下, 可以通过代理扩展, 或者 pac 文件, 来搞定动态代理的问题
iii. 在 headless 的模式, 那就只好从 upstream 角度搞了, 甚至挂上 mitmproxy 也行吧
iv. 至于动态修改 UA, 暂时可以用扩展来搞, 不过如果喜欢钻研, 可以发现 CDP 里支持动态修改 Request 的各项属性, 在这里改 headers 是有效的

文档

常用功能

Chrome DevTools Protocol 文档的使用, 主要还是使用里面的检索功能, 不过最常用的还是以下几个领域

Page

i. 简单地理解, 可以把一个 Page 看成一个 Page 类型的 Tab
ii. 对 Tab 的刷新, 跳转, 停止, 激活, 截图等功能都可以找到
iii. 也会有很多有用的事件需要 enable Page 以后才能监听到, 比如 loadEventFired
iv. 多个网站的任务, 可以在同一个浏览器里打开多个 Tab 进行操作, 通过不同的 Websocket 地址进行连接, 相对隔离, 并且托异步模型的福, Chrome 多个标签操作的抗压能力还不错
v. 然而并发操作多个 Tab 的时候, 可能会出现一点小问题需要注意: 同一个浏览器实例, 对一个域名只能建立 6 个连接, 这个不太好改; 过快生成大量 Tab, 可能会导致有的 Tab 无法正常关闭(zombie tabs)

Network

i. 和产生网络流量有关系的大都在这个 Domain

ii. 比如 setExtraHTTPHeaders / setUserAgentOverride 对当前标签页的所有请求修改原是参数
iii. 比如对 cookie 的各种操作
iv. 通过 responseReceived + getResponseBody 来监听流量, 只用前者就能嗅探到 mp4 这种特殊类型的 url 了, 而后者可以把流量里已经 base64 化的数据进行其他操作, 比如验证码图片的处理

其他功能也基本和 devtools 一致

常规姿势

和某个 Tab 建立连接
通过 send 发送你想使用的 methods
通过 recv 监听你发送 methods 产生的事件, 或者其他 enable 的事件, 并执行对应回调

实践

准备工作

安装 chrome 浏览器

安装 Python3.7

ichrome 库是可选的, 主要是为了演示通过 HTTP / Websocket client 与 chrome 实例实现通信
ichrome 库除了协程实现, 也有一个同步实现, 观察它的源码比协程版本的更直观一点, 也易于学习
pip install ichrome -U

启动调试模式下的 chrome

复制

```
from ichrome import ChromeDaemon


def launch_chrome():
    with ChromeDaemon(host="127.0.0.1", port=9222, max_deaths=1) as chrome
        chromed.run_forever()


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

以上代码负责启动 chrome 调试模式的守护进程, 具体参数如下:

chrome_path: 表示 chrome 的可执行路径 / 命令, 默认为 None 的时候, 会自动根据操作系统去尝试找寻 chrome 路径, 如 **linux** 下的 google-chrome 和 google-chrome-stable, **macOS** 下的 /Applications/Google Chrome.app/Contents/MacOS/Google Chrome, 或者 Windows 下的

i. C:/Program Files (x86)/Google/Chrome/Application/chrome.exe
ii. C:/Program Files/Google/Chrome/Application/chrome.exe
iii. "%s\AppData\Local\Google\Chrome\Application\chrome.exe" % os.getenv("USERPROFILE")

host: 默认为 127.0.0.1, 之所以不用 localhost, 是因为很多 Windows / macOS 的 etc/hosts 文件里被强制绑定到了 ipv6 地址上

port: 默认为 9222

headless: 常见参数 –headless, –hide-scrollbars, 放在初始化参数里了

user_agent: 常见参数 –user-agent

proxy: 常见参数 –proxy-server

user_data_dir: 避免 chrome 到处乱放 user data, 所以默认会放到 user 目录下的 ichrome_user_data 文件夹下, 命名按端口号 chrome_9222

disable_image: 常用参数 –blink-settings=imagesEnabled=false, 从 blink 层面禁用, 比其他禁止图片加载的方式要靠谱

max_deaths: 用来自动重启, max_deaths=2 表示快速杀死 chrome 实例 2 次才能避免再次自动重启, 所以默认为 1

extra_config: 就是添加其他更多 chrome 启动的参数, 参数类型为 list

启动带图形界面的 chrome 之后, 可以手动尝试下通过 http 请求和 chrome 实例通信了

访问 http://127.0.0.1:9222/json , 会拿到一个列出当前 tabs 信息的 json

其他操作参考 https://chromedevtools.github.io/devtools-protocol/ (HTTP Endpoints 部分)

复制

```
[
{
    "description": "",
    "devtoolsFrontendUrl": "/devtools/inspector.html?ws=127.0.0.1:9222
    "id": "E6826ED4A0365605F3234B2A441B1D03",
    "title": "about:blank",
    "type": "page",
    "url": "about:blank",
    "webSocketDebuggerUrl": "ws://127.0.0.1:9222/devtools/page/E6826ED
}
]
```

操作 Tab

建立到 webSocketDebuggerUrl 的 Websocket 连接, 然后监听事件

大部分功能 ichrome 已经打包好了

复制

```
from ichrome import AsyncChrome
import asyncio


async def async_operate_tab():
chrome = AsyncChrome(host='127.0.0.1', port=9222)
if not await chrome.connect():
    raise RuntimeError
tab = (await chrome.tabs)[0]
async with tab():
    # 跳转到 httpbin, 3 秒 loading 超时的话则 stop loading
    await tab.set_url('http://httpbin.org', timeout=3)
    # 注入 js, 并查看返回结果
    result = await tab.js("document.title")
    title = result['result']['result']['value']
    # 打印 title
    print(title)
    # httpbin.org
    # 通过 js 修改 title
    await tab.js("document.title = 'New Title'")
    # click 一个 css 选择器的位置, 跳转到了 Github
```

```
    await tab.click('body > a:first-child')
    # 等待加载完成
    await tab.wait_loading(3)

    async def callback_function(request):
        if request:
            # 监听到经过过滤的流量，等待它加载一会比较保险
            for _ in range(3):
                result = await tab.get_response(request)
                if result.get('error'):
                    await tab.wait_loading(1)
                    continue
                # 拿到整个 html
                body = result['result']['body']
                print(body)

    def filter_func(r):
        url = r['params']['response']['url']
        print('received:', url)
        return url == 'https://github.com/'

    # 监听流量，需要异步处理，则使用 asyncio.ensure_future 即可
    # 监听 10 秒
    task = asyncio.ensure_future(
        tab.wait_response(
            filter_function=filter_func,
            callback_function=callback_function,
            timeout=10),
        loop=tab.loop)
    # 点击一下左上角的小章鱼则会触发流量
    await tab.click('[href="https://github.com/"]')
    # 等待监听流量
    await task


if __name__ == "__main__":
asyncio.run(async_operate_tab())
```

总结

CDP 单单入门的话，其实没想象中那么复杂，chrome 59 刚出的时候，puppeteer 都没的用，更别说 pyppeteer 之类的包装，看了几个早期项目的源码，发现简单使用的话，其实主要就是：

复制

```
1. HTTP
2. Websocket
3 Javascript
4. Protocol
```

pyppeteer 诞生之初曾体验了一下，第一步就因为一些不可抗力导致下载 chromium 失败，所以之后只能阅读一下里面一些有意思的源码，主要看了下如何从 puppeteer 原生事件驱动转为 Python 角度的事件，pyee 的使用也让人眼前一亮

之后自己摸索过程中也碰到了各种各样问题，除了上面提到的，其实还遇到 Websocket 粘包(粘包本身就是个因为理解不足导致的伪命题)，Chrome Headless 阉割掉了很多基础功能也使开发过程中总是无理由地调试失败，甚至关闭 user-dir 使用匿名模式导致一系列不知名故障也是费心费力，不过总体来说收获颇大
用 Python 来操作 chrome 能做的事情挺多，尤其是各路签到爬虫，或者索取微信公众平台大概 20 小时有效期的 cookie / token 给后台爬虫使用，采集视频，自动化测试时候截图，启动 Headless 模式以后节省了很多手动操作的时间，甚至可以丢到无 GUI 的 li

nux server 上去. 要知道以前指望的还是 tampermonkey 或者手写扩展, 很多 Python 的功能只能转 js 再用, 劳心劳力.

Read more