# **1、Puppeteer 简介**

Puppeteer 是一个node库，他提供了一组用来操纵Chrome的API, 通俗来说就是一个 headless chrome浏览器 (当然你也可以配置成有UI的，默认是没有的)。既然是浏览器，那么我们手工可以在浏览器上做的事情 Puppeteer 都能胜任, 另外，Puppeteer 翻译成中文是”木偶”意思，所以听名字就知道，操纵起来很方便，你可以很方便的操纵她去实现：

1） 生成网页截图或者 PDF  
2） 高级爬虫，可以爬取大量异步渲染内容的网页  
3） 模拟键盘输入、表单自动提交、登录网页等，实现 UI 自动化测试  
4） 捕获站点的时间线，以便追踪你的网站，帮助分析网站性能问题

如果你用过 PhantomJS 的话，你会发现她们有点类似，但Puppeteer是Chrome官方团队进行维护的，用俗话说就是”有娘家的人“，前景更好。

# **2、运行环境**

查看 Puppeteer 的官方 API 你会发现满屏的 async, await 之类，这些都是 ES7 的规范，所以你需要：

1. Nodejs 的版本不能低于 v7.6.0, 需要支持 async, await.
2. 需要最新的 chrome driver, 这个你在通过 npm 安装 Puppeteer 的时候系统会自动下载的

npm install puppeteer --save

# **3、基本用法**

先开看看官方的入门的 DEMO

const puppeteer = require('puppeteer');

(async () => {

const browser = await puppeteer.launch();

const page = await browser.newPage();

await page.goto('https://example.com');

await page.screenshot({path: 'example.png'});

await browser.close();})();

上面这段代码就实现了网页截图，先大概解读一下上面几行代码：

1. 先通过 puppeteer.launch() 创建一个浏览器实例 Browser 对象
2. 然后通过 Browser 对象创建页面 Page 对象
3. 然后 page.goto() 跳转到指定的页面
4. 调用 page.screenshot() 对页面进行截图
5. 关闭浏览器

是不是觉得好简单？ 反正我是觉得比 PhantomJS 简单，至于跟 selenium-webdriver 比起来， 那更不用说了。下面就介绍一下 puppeteer 的常用的几个 API。

## **3.1 puppeteer.launch(options)**

使用 puppeteer.launch() 运行 puppeteer，它会 return 一个 promise，使用 then 方法获取 browser 实例， 当然高版本的 的 nodejs 已经支持 await 特性了，所以上面的例子使用 await 关键字，这一点需要特殊说明一下，****Puppeteer 几乎所有的操作都是 异步的****, 为了使用大量的 then 使得代码的可读性降低，本文所有 demo 代码都是用 ****async, await**** 方式实现。这个 也是 Puppeteer 官方推荐的写法。对 async/await 一脸懵逼的同学狠狠的[戳这里](http://www.r9it.com/20171108/async-await.html)

##### **options 参数详解**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数类型 | 参数说明 |
| ignoreHTTPSErrors | boolean | 在请求的过程中是否忽略 Https 报错信息，默认为 false |
| headless | boolean | 是否以”无头”的模式运行 chrome, 也就是不显示 UI， 默认为 true |
| executablePath | string | 可执行文件的路劲，Puppeteer 默认是使用它自带的 chrome webdriver, 如果你想指定一个自己的 webdriver 路径，可以通过这个参数设置 |
| slowMo | number | 使 Puppeteer 操作减速，单位是毫秒。如果你想看看 Puppeteer 的整个工作过程，这个参数将非常有用。 |
| args | Array(String) | 传递给 chrome 实例的其他参数，比如你可以使用”–ash-host-window-bounds=1024x768” 来设置浏览器窗口大小。更多参数参数列表可以参考[这里](https://peter.sh/experiments/chromium-command-line-switches/) |
| handleSIGINT | boolean | 是否允许通过进程信号控制 chrome 进程，也就是说是否可以使用 CTRL+C 关闭并退出浏览器. |
| timeout | number | 等待 Chrome 实例启动的最长时间。默认为30000（30秒）。如果传入 0 的话则不限制时间 |
| dumpio | boolean | 是否将浏览器进程stdout和stderr导入到process.stdout和process.stderr中。默认为false。 |
| userDataDir | string | 设置用户数据目录，默认linux 是在 ~/.config 目录，window 默认在 C:\Users{USER}\AppData\Local\Google\Chrome\User Data, 其中 {USER} 代表当前登录的用户名 |
| env | Object | 指定对Chromium可见的环境变量。默认为process.env。 |
| devtools | boolean | 是否为每个选项卡自动打开DevTools面板， 这个选项只有当 headless 设置为 false 的时候有效 |

## **3.2 Browser 对象**

当 Puppeteer 连接到一个 Chrome 实例的时候就会创建一个 Browser 对象，有以下两种方式：

Puppeteer.launch 和 Puppeteer.connect.

下面这个 DEMO 实现断开连接之后重新连接浏览器实例

const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {

// 保存 Endpoint，这样就可以重新连接 Chromium

const browserWSEndpoint = browser.wsEndpoint();

// 从Chromium 断开连接

browser.disconnect();

// 使用endpoint 重新和 Chromiunm 建立连接

const browser2 = await puppeteer.connect({browserWSEndpoint});

// Close Chromium

await browser2.close();});

##### **Browser 对象 API**

|  |  |  |
| --- | --- | --- |
| 方法名称 | 返回值 | 说明 |
| browser.close() | Promise | 关闭浏览器 |
| browser.disconnect() | void | 断开浏览器连接 |
| browser.newPage() | Promise(Page) | 创建一个 Page 实例 |
| browser.pages() | Promise(Array(Page)) | 获取所有打开的 Page 实例 |
| browser.targets() | Array(Target) | 获取所有活动的 targets |
| browser.version() | Promise(String) | 获取浏览器的版本 |
| browser.wsEndpoint() | String | 返回浏览器实例的 socket 连接 URL, 可以通过这个 URL 重连接 chrome 实例 |

好了，Puppeteer 的API 就不一一介绍了，官方提供的详细的 API， [戳这里](https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md)

# **4、Puppeteer 实战**

了解 API 之后我们就可以来一些实战了，在此之前，我们先了解一下 Puppeteer 的设计原理，简单来说 Puppeteer 跟 webdriver 以及 PhantomJS 最大的 的不同就是它是站在用户浏览的角度，而 webdriver 和 PhantomJS 最初设计就是用来做自动化测试的，所以它是站在机器浏览的角度来设计的，所以它们 使用的是不同的设计哲学。举个栗子，加入我需要打开京东的首页并进行一次产品搜索，分别看看使用 Puppeteer 和 webdriver 的实现流程：

****Puppeteer 的实现流程：****

1. 打开京东首页
2. 将光标 focus 到搜索输入框
3. 键盘点击输入文字
4. 点击搜索按钮

****webdriver 的实现流程：****

1. 打开京东首页
2. 找到输入框的 input 元素
3. 设置 input 的值为要搜索文字
4. 触发搜索按钮的单机事件

个人感觉 Puppeteer 设计哲学更符合任何的操作习惯，更自然一些。

下面我们就用一个简单的需求实现来进行 Puppeteer 的入门学习。这个简单的需求就是：

在京东商城抓取10个手机商品，并把商品的详情页截图。

首先我们来梳理一下操作流程

1. 打开京东首页
2. 输入“手机”关键字并搜索
3. 获取前10个商品的 A 标签，并获取 href 属性值，获取商品详情链接
4. 分别打开10个商品的详情页，截取网页图片

要实现上面的功能需要用到查找元素，获取属性，键盘事件等，那接下来我们就一个一个的讲解一下。

## **4.1 获取元素**

Page 对象提供了2个 API 来获取页面元素

(1). Page.$(selector) 获取单个元素，底层是调用的是 document.querySelector() , 所以选择器的 selector 格式遵循 [css 选择器规范](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors)

let inputElement = await page.$("#search", input => input);//下面写法等价let inputElement = await page.$('#search');

(2). Page.$$(selector) 获取一组元素，底层调用的是 document.querySelectorAll(). 返回 Promise(Array(ElemetHandle)) 元素数组.

const links = await page.$$("a");//下面写法等价const links = await page.$$("a", links => links);

最终返回的都是 ElemetHandle 对象

## **4.2 获取元素属性**

Puppeteer 获取元素属性跟我们平时写前段的js的逻辑有点不一样，按照通常的逻辑，应该是现获取元素，然后在获取元素的属性。但是上面我们知道 获取元素的 API 最终返回的都是 ElemetHandle 对象，而你去查看 ElemetHandle 的 API 你会发现，它并没有获取元素属性的 API.

事实上 Puppeteer 专门提供了一套获取属性的 API， Page.$eval() 和 Page.$$eval()

(1). Page.$$eval(selector, pageFunction[, …args]), 获取单个元素的属性，这里的选择器 selector 跟上面 Page.$(selector) 是一样的。

const value = await page.$eval('input[name=search]', input => input.value);const href = await page.$eval('#a", ele => ele.href);const content = await page.$eval('.content', ele => ele.outerHTML);

## **4.3 执行自定义的 JS 脚本**

Puppeteer 的 Page 对象提供了一系列 evaluate 方法，你可以通过他们来执行一些自定义的 js 代码，主要提供了下面三个 API

(1). page.evaluate(pageFunction, …args) 返回一个可序列化的普通对象，pageFunction 表示要在页面执行的函数， args 表示传入给 pageFunction 的参数， 下面的 pageFunction 和 args 表示同样的意思。

const result = await page.evaluate(() => {

return Promise.resolve(8 \* 7);});

console.log(result); // prints "56"

这个方法很有用，比如我们在获取页面的截图的时候，默认是只截图当前浏览器窗口的尺寸大小，默认值是800x600，那如果我们需要获取整个网页的完整 截图是没办法办到的。Page.screenshot() 方法提供了可以设置截图区域大小的参数，那么我们只要在页面加载完了之后获取页面的宽度和高度就可以解决 这个问题了。

(async () => {

const browser = await puppeteer.launch({headless:true});

const page = await browser.newPage();

await page.goto('https://jr.dayi35.com');

await page.setViewport({width:1920, height:1080});

const documentSize = await page.evaluate(() => {

return {

width: document.documentElement.clientWidth,

height : document.body.clientHeight,

}

})

await page.screenshot({path:"example.png", clip : {x:0, y:0, width:1920, height:documentSize.height}});

await browser.close();})();

(2). Page.evaluateHandle(pageFunction, …args) 在 Page 上下文执行一个 pageFunction, 返回 JSHandle 实体

const aWindowHandle = await page.evaluateHandle(() => Promise.resolve(window));

aWindowHandle; // Handle for the window object.

const aHandle = await page.evaluateHandle('document'); // Handle for the 'document'.

从上面的代码可以看出，page.evaluateHandle() 方法也是通过 Promise.resolve 方法直接把 Promise 的最终处理结果返回， 只不过把最后返回的对象封装成了 JSHandle 对象。本质上跟 evaluate 没有什么区别。

下面这段代码实现获取页面的动态（包括js动态插入的元素） HTML 代码.

const aHandle = await page.evaluateHandle(() => document.body);const resultHandle = await page.evaluateHandle(body => body.innerHTML, aHandle);

console.log(await resultHandle.jsonValue());await resultHandle.dispose();

(3). page.evaluateOnNewDocument(pageFunction, …args), 在文档页面载入前调用 pageFunction, 如果页面中有 iframe 或者 frame, 则函数调用 的上下文环境将变成子页面的，即iframe 或者 frame, 由于是在页面加载前调用，这个函数一般是用来初始化 javascript 环境的，比如重置或者 初始化一些全局变量。

## **4.4 Page.exposeFunction**

除此上面三个 API 之外，还有一类似的非常有用的 API， 那就是 ****Page.exposeFunction****，这个 API 用来在页面注册全局函数，非常有用：

因为有时候需要在页面处理一些操作的时候需要用到一些函数，虽然可以通过 Page.evaluate() API 在页面定义函数，比如：

const docSize = await page.evaluate(()=> {

function getPageSize() {

return {

width: document.documentElement.clientWidth,

height : document.body.clientHeight,

}

}

return getPageSize();});

但是这样的函数不是全局的，需要在每个 evaluate 中去重新定义，无法做到代码复用，在一个就是 nodejs 有很多工具包可以很轻松的实现很复杂的功能 比如要实现 md5 加密函数，这个用纯 js 去实现就不太方便了，而用 nodejs 却是几行代码的事情。

下面代码实现给 Page 上下文的 window 对象添加 md5 函数：

const puppeteer = require('puppeteer');const crypto = require('crypto');

puppeteer.launch().then(async browser => {

const page = await browser.newPage();

page.on('console', msg => console.log(msg.text));

await page.exposeFunction('md5', text =>

crypto.createHash('md5').update(text).digest('hex')

);

await page.evaluate(async () => {

// use window.md5 to compute hashes

const myString = 'PUPPETEER';

const myHash = await window.md5(myString);

console.log(`md5 of ${myString} is ${myHash}`);

});

await browser.close();});

可以看出，Page.exposeFunction API 使用起来是很方便的，也非常有用，在比如给 window 对象注册 readfile 全局函数：

const puppeteer = require('puppeteer');const fs = require('fs');

puppeteer.launch().then(async browser => {

const page = await browser.newPage();

page.on('console', msg => console.log(msg.text));

await page.exposeFunction('readfile', async filePath => {

return new Promise((resolve, reject) => {

fs.readFile(filePath, 'utf8', (err, text) => {

if (err)

reject(err);

else

resolve(text);

});

});

});

await page.evaluate(async () => {

// use window.readfile to read contents of a file

const content = await window.readfile('/etc/hosts');

console.log(content);

});

await browser.close();});

# **5、Page.emulate 修改模拟器(客户端)运行配置**

Puppeteer 提供了一些 API 供我们修改浏览器终端的配置

1. Page.setViewport() 修改浏览器视窗大小
2. Page.setUserAgent() 设置浏览器的 UserAgent 信息
3. Page.emulateMedia() 更改页面的CSS媒体类型，用于进行模拟媒体仿真。 可选值为 “screen”, “print”, “null”, 如果设置为 null 则表示禁用媒体仿真。
4. Page.emulate() 模拟设备，参数设备对象，比如 iPhone, Mac, Android 等

page.setViewport({width:1920, height:1080}); //设置视窗大小为 1920x1080

page.setUserAgent('Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.90 Safari/537.36');

page.emulateMedia('print'); //设置打印机媒体样式

除此之外我们还可以模拟非 PC 机设备, 比如下面这段代码模拟 iPhone 6 访问google：

const puppeteer = require('puppeteer');const devices = require('puppeteer/DeviceDescriptors');const iPhone = devices['iPhone 6'];

puppeteer.launch().then(async browser => {

const page = await browser.newPage();

await page.emulate(iPhone);

await page.goto('https://www.google.com');

// other actions...

await browser.close();});

Puppeteer 支持很多设备模拟仿真，比如Galaxy, iPhone, IPad 等，想要知道详细设备支持，请戳这里 [DeviceDescriptors.js.](https://github.com/GoogleChrome/puppeteer/blob/master/DeviceDescriptors.js)

# **6、键盘和鼠标**

键盘和鼠标的API比较简单，键盘的几个API如下：

* keyboard.down(key[, options]) 触发 keydown 事件
* keyboard.press(key[, options]) 按下某个键，key 表示键的名称，比如 ‘ArrowLeft’ 向左键，详细的键名映射请[戳这里](https://github.com/GoogleChrome/puppeteer/blob/master/lib/USKeyboardLayout.js)
* keyboard.sendCharacter(char) 输入一个字符
* keyboard.type(text, options) 输入一个字符串
* keyboard.up(key) 触发 keyup 事件

page.keyboard.press("Shift"); //按下 Shift 键

page.keyboard.sendCharacter('嗨');

page.keyboard.type('Hello'); // 一次输入完成

page.keyboard.type('World', {delay: 100}); // 像用户一样慢慢输入

鼠标操作：

* mouse.click(x, y, [options]) 移动鼠标指针到指定的位置，然后按下鼠标，这个其实 mouse.move 和 mouse.down 或 mouse.up 的快捷操作
* mouse.down([options]) 触发 mousedown 事件，options 可配置:
  + options.button 按下了哪个键，可选值为[left, right, middle], 默认是 left, 表示鼠标左键
  + options.clickCount 按下的次数，单击，双击或者其他次数
  + delay 按键延时时间
* mouse.move(x, y, [options]) 移动鼠标到指定位置， options.steps 表示移动的步长
* mouse.up([options]) 触发 mouseup 事件

# **7、另外几个有用的 API**

Puppeteer 还提供几个非常有用的 API， 比如：

## **7.1 Page.waitFor 系列 API**

* page.waitFor(selectorOrFunctionOrTimeout[, options[, …args]]) 下面三个的综合 API
* page.waitForFunction(pageFunction[, options[, …args]]) 等待 pageFunction 执行完成之后
* page.waitForNavigation(options) 等待页面基本元素加载完之后，比如同步的 HTML, CSS, JS 等代码
* page.waitForSelector(selector[, options]) 等待某个选择器的元素加载之后，这个元素可以是异步加载的，这个 API 非常有用，你懂的。

比如我想获取某个通过 js 异步加载的元素，那么直接获取肯定是获取不到的。这个时候就可以使用 page.waitForSelector 来解决：

await page.waitForSelector('.gl-item'); //等待元素加载之后，否则获取不到异步加载的元素const links = await page.$$eval('.gl-item > .gl-i-wrap > .p-img > a', links => {

return links.map(a => {

return {

href: a.href.trim(),

name: a.title

}

});});

其实上面的代码就可以解决我们最上面的需求，抓取京东的产品，因为是异步加载的，所以使用这种方式。

## **7.2 page.getMetrics()**

通过 page.getMetrics() 可以得到一些页面性能数据， 捕获网站的时间线跟踪，以帮助诊断性能问题。

* Timestamp 度量标准采样的时间戳
* Documents 页面文档数
* Frames 页面 frame 数
* JSEventListeners 页面内事件监听器数
* Nodes 页面 DOM 节点数
* LayoutCount 页面布局总数
* RecalcStyleCount 样式重算数
* LayoutDuration 所有页面布局的合并持续时间
* RecalcStyleDuration 所有页面样式重新计算的组合持续时间。
* ScriptDuration 所有脚本执行的持续时间
* TaskDuration 所有浏览器任务时长
* JSHeapUsedSize JavaScript 占用堆大小
* JSHeapTotalSize JavaScript 堆总量

# **8、总结和源码**

本文通过一个实际需求来学习了 Puppeteer 的一些基本的常用的 API， API 的版本是 v0.13.0-alpha. 最新邦本的 API 请参考 [Puppeteer 官方API](https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md).

总的来说，Puppeteer 真是一款不错的 headless 工具，操作简单，功能强大。用来做UI自动化测试，和一些小工具都是很不错的。

下面贴上我们开始的需求实现源码，仅供参考：

//延时函数function sleep(delay) {

return new Promise((resolve, reject) => {

setTimeout(() => {

try {

resolve(1)

} catch (e) {

reject(0)

}

}, delay)

})}

const puppeteer = require('puppeteer');

puppeteer.launch({

ignoreHTTPSErrors:true,

headless:false,slowMo:250,

timeout:0}).then(async browser => {

let page = await browser.newPage();

await page.setJavaScriptEnabled(true);

await page.goto("https://www.jd.com/");

const searchInput = await page.$("#key");

await searchInput.focus(); //定位到搜索框

await page.keyboard.type("手机");

const searchBtn = await page.$(".button");

await searchBtn.click();

await page.waitForSelector('.gl-item'); //等待元素加载之后，否则获取不异步加载的元素

const links = await page.$$eval('.gl-item > .gl-i-wrap > .p-img > a', links => {

return links.map(a => {

return {

href: a.href.trim(),

title: a.title

}

});

});

page.close();

const aTags = links.splice(0, 10);

for (var i = 1; i < aTags.length; i++) {

page = await browser.newPage()

page.setJavaScriptEnabled(true);

await page.setViewport({width:1920, height:1080});

var a = aTags[i];

await page.goto(a.href, {timeout:0}); //防止页面太长，加载超时

//注入代码，慢慢把滚动条滑到最底部，保证所有的元素被全部加载

let scrollEnable = true;

let scrollStep = 500; //每次滚动的步长

while (scrollEnable) {

scrollEnable = await page.evaluate((scrollStep) => {

let scrollTop = document.scrollingElement.scrollTop;

document.scrollingElement.scrollTop = scrollTop + scrollStep;

return document.body.clientHeight > scrollTop + 1080 ? true : false

}, scrollStep);

await sleep(100);

}

await page.waitForSelector("#footer-2014", {timeout:0}); //判断是否到达底部了

let filename = "images/items-"+i+".png";

//这里有个Puppeteer的bug一直没有解决，发现截图的高度最大只能是16384px， 超出部分被截掉了。

await page.screenshot({path:filename, fullPage:true});

page.close();

}

browser.close();});

****《THE END》****