

# 前言

---

## 好处

---

必要性

页面

自动化测试有很多好处，测试最重要的自然是提升代码质量。

以前不喜欢写测试，主要是觉得编写和维护测试用例非常浪费时间。

用例可以通过录制自动生成，节约开发时间而且

自动化测试另外一个重要特点就是快速反馈，反馈越迅速意味着开发效率越高。

主要的目标：让自测不要重复测！让自动化变成低成本！让自动化持续保障您的业务！

过PO用例用时比开发用时还要久。尤其是维护长期迭代的业务，开发不太可能在每次修改之后都去检查所有的用例，长此以往，不常用的场景可能就会有bug。

## 社区方案对比

---

1. UI Recorder (1.7k Star、2020.7.28, 阿里出品)
2. Cypress (22.8k Star, 国外很火, 头条有在用)
3. Nightmare (18.6k Star 以前很火, 现在基本上被后一个方案取代)
4. puppeteer + jest + headless-recorder (10.1k Star, 头条有在用, 上次更新2020年9月2日, 是有维护的状态。)
5. QTA (腾讯出品, 支持多语言)
6. selenium-ide (1.4k Star)

比较项	UI Recorder	Cypress	Nightmare	Puppeteer	QTA	selenium-ide
录制生成脚本	支持	不支持	支持	支持	不支持	支持
开发语言	Mocha	Mocha	chai	jest	Python,c#,ruby,java,不支持JS	jest

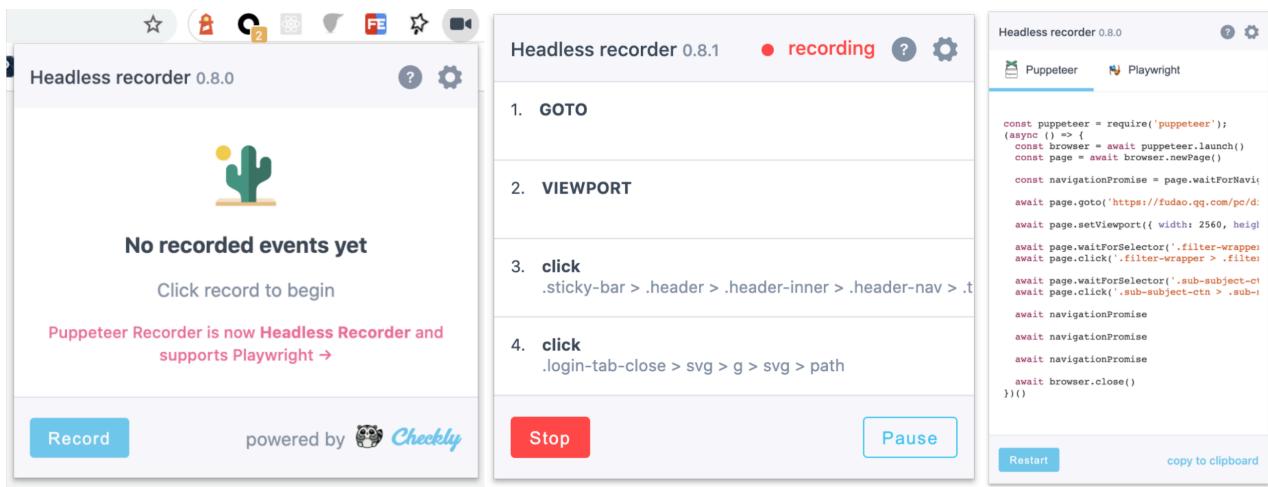
从时间成本的角度，手工编写脚本，成本高，效率低，但维护成本低。录制生成脚本，成本低，效率高，但对脚本的理解成本高。由于“不仅是前端使用，测试也能快速上手”的定位，我们选择录制生成脚本的方案，如果你的团队主要由前端负责自动化测试且有人力负责编写测试脚本，Cypress或许是个不错的选择。其中Nightmare长期没有维护了，所以也不在以下考虑范围内。

比较项	UI Recorder	Puppeteer	selenium-ide
录制生成脚本	支持	支持	支持
开发语言	Mocha	jest	jest
侵入性	低	低	低
底层实现	Webdriver	非Webdriver (chromium)	Webdriver
社区活跃	Yes	Yes	Yes
录制/执行用例效率	★★★	★★	★★

其中侵入性是指测试用例代码对项目代码的影响，比如是不是一定要在项目中安装依赖？可以完全将用例抽离成另外一个单独项目？使用体验后三个方案的侵入性都低。

自动化测试方案的底层实现目前业界主要分为两类，Webdriver和非Webdriver。非Webdriver对应测试单一浏览器。Webdriver开源、多语言、支持跨平台、不局限某个特定浏览器。只要想测的浏览器有Webdriver能力就都可以跑自动化。考虑到后续推广给测试同学，可能要测浏览器的兼容性，所以优先选择Webdriver类的方案。

从录制/执行用例的效率来来看，Puppeteer的脚本录制如下图所示通过chrome插件点击“Record”开始录制，出现图2然后操作页面后点击“Stop”结束录制出现图3。需要手动新建文件，复制代码，效率较低。



在调研过程中使用Puppeteer时发现有两个缺点：

1. 打开新页面后的操作无法录制，对于打开的新页面只能单独录制，然后再人工整理成一个脚本。我们的业务较多链接都是打开新页面，所以该方案用起来不太方便。
2. 不支持自动生成断言语句，需要自行在脚本中补充逻辑。

UIRecorder录制是基于浏览器进程，不同于Puppeteer是基于当前页面，所以没有以上问题1。而且UIRecorder是直接生成脚本文件，节省了新建文件复制等人工操作，所以从录制/执行用例效率的角度，UIRecorder优于Puppeteer。

UIRecorder的工具栏提供了断言的功能，可以自动生成断言逻辑（虽然使用的过程中有遇到某些断言逻辑无法通过验证）。从用例覆盖率的角度，UIRecorder相对优于Puppeteer。

授课老师：小豹老师（李泓）

UIRecorder

DomPath: //span[text()="登录"]

属性开关: id text name value data-id data-name type data-type role data-role data-value

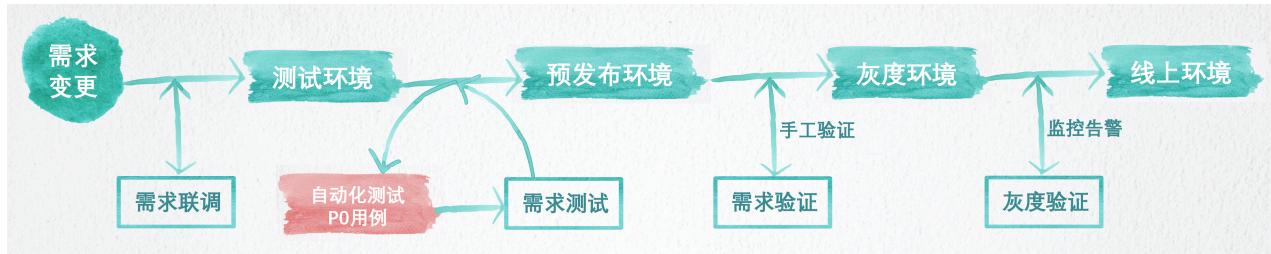
属性黑名单: 请输入过滤属性值的正则表达式, 例如: /black\_val/

添加悬停 添加断言 使用变量 执行JS 添加延迟 脚本跳转 结束录制

综上，我们主要考量：时间成本、侵入性、录制/执行用例效率、用例覆盖率后选择了UIRecorder。

## 自动化测试实践

应该在哪个流程做自动化测试呢？显然问题越早暴露，解决问题的成本越低。所以我们选择在代码部署至测试环境后的开发自测阶段，过PO用例的同时录制新的用例脚本。然后执行自动化测试，确保旧用例和新用例都通过验证。



## 基本操作

### 项目初始化

```

cnpm install uirecorder mocha macaca-reporter -g
uirecorder init //初始化
npm run installdriver //安装工具
  
```

### PC端录制

```

npm run server //录制的时候启动selenium服务, selenium通过协议和Driver进行通信
uirecorder start //开始录制
  
```

### PC端回放

```

npm run singletest 测试脚本名称
npm run paralleltest
npm run moduletest
  
```

### H5录制

修改config.json后执行PC端录制流程

```
"webdriver": {
    "host": "127.0.0.1",
    "port": "4444",
    "chromeOptions": {
        "mobileEmulation": {"deviceName": "iPhone X"},
        "w3c": false
    },
    "browsers": "chrome"
}
```

app端录制和回放流程较多这里不累述。

为了确认UIRecorder到底是不是一个合适的方案，在调研过程中深(shi)度(jin)使(zhe)用(teng)。

我们的开发流程中测试环境主要是借助chrome的[SwitchyOmega](#)插件和nohost。虽然 WebDriverJS 可以在 Node 中运行，但它至今还没有实现本地驱动的支持（也就是说，你的测试必须使用一个远程的 WebDriver 服务）。WebDriver每次启动的浏览器进程都是纯净版无插件无用户storage信息的，所以uirecorder初始化的项目启动的浏览器是不支持nohost的。但是我们可以在 Chromedriver (Chromedriver是用于Chrome的WebDriver) 中配置pac文件来实现代理自动配置。

## 自定义浏览器启动参数

```
let sessionConfig = Object.assign({}, webdriverConfig, {
    'group': group,
    'browserName': browserName,
    'version': browserVersion,
    'ie.ensureCleanSession': true,
    'chromeOptions': {
        "w3c": false,
        args: ['--enable-automation', '--disable-bundled-ppapi-flash', '--ignore-certificate-errors', '--proxy-pac-url=https://***.pac'],
        prefs: {
            'plugins.plugins_disabled': ['Adobe Flash Player']
        }
    }
});
```

Chromium 参数文档 <https://chromedriver.chromium.org/capabilities>

chromeOptions的args命令行参数列表文档 <https://peter.sh/experiments/chromium-command-line-switches/>

## 自定义浏览器扩展程序

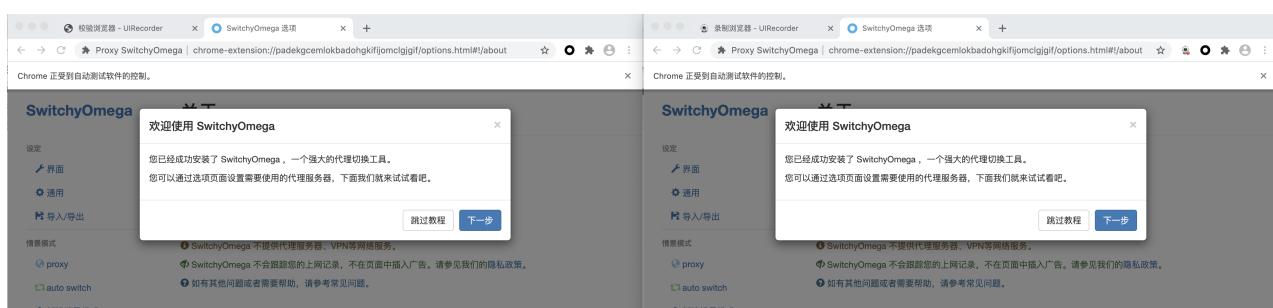
录制用例时会启动两个浏览器：录制浏览器、验证浏览器。如果我们希望启动的浏览器有插件能力。可以在uirecorder源码中修改函数newChromeBrowser，加载Chrome扩展程序的文件。以下以SwitchyOmega为例。

```
1747 const switchyOmegaCrxPath = path.resolve(__dirname, '../tool/switchyOmega2.5.21_0.crx');
1748 const switchyOmegaExtContent = fs.readFileSync(switchyOmegaCrxPath).toString('base64');
1749 if(options.isRecorder){
1750   if(options.debug){
1751     capabilities.chromeOptions = {
1752       args: ['--enable-automation', '--disable-bundled-ppapi-flash', '--load-extension=' + path.resolve(__dirname, '../chrome-extension')],
1753       prefs: {
1754         'plugins.plugins_disabled': ['Adobe Flash Player']
1755       },
1756       extensions: [switchyOmegaExtContent]
1757     };
1758   }
}
```

Chrome扩展程序的文件可以通过访问chrome://extensions打开对应扩展的详情，然后打包扩展程序获取到crx3文件。



一般扩展程序都有初始引导，每次录制脚本都会弹窗就比较麻烦，Chrome扩展程序都是开源的，可以修改源码去掉相关逻辑。



有些插件需要用户配置数据，默认是没有数据的，比如SwitchyOmega，可以把我们要的配置直接打包到扩展文件中。具体看大家的业务使用场景。可定制化能力还是很丰富的。

## 在Docker中执行用例

```
docker pull selenium/hub
docker pull selenium/node-chrome

docker run -d --name hub -p 4444:4444 selenium/hub
docker run -d -P -p 5901:5900 -p 15000:5555 --link hub:hub selenium/node-chrome

// 修改uirecorder的项目中的config.json
"webdriver": {
  "host": "127.0.0.1",
  "port": "15000",
  "browsers": "chrome"
}
```

## 理解测试用例脚本

录制会自动生成脚本，在必要的时候我们也可以自己修改脚本，比如可以单独封装一个登录操作的公共脚本。

主要用到了以下javascript库

- Mocha —— 核心框架：提供了包括 describe 和 it 的通用型测试函数和运行测试的主函数。
- Chai —— 提供很多断言支持的库。它可以用很多不同的断言。assert.should.equal、assert.should.be、assert.should.not.be等等。

```
171     describe(caseName, function(){
172
173     |     this.timeout(600000);
174     |     this.slow(1000);
175
176     |     let driver;
177 >     |     before(function(){ ...
225
226     |     module.exports();
227
228 >     |     beforeEach(function(){ ...
235
236 >     |     afterEach(async function(){ ...
263
264 >     |     after(function(){ ...
267
268     |});
```

## describe

```
describe("title",function(){...})
```

表示我们正在描述的功能是什么。用于组织 workers (it代码块)

## before/after and beforeEach/afterEach

我们可以设置 before/after 函数来在运行测试之前/之后执行。也可以使用 beforeEach/afterEach 在执行每一个 it 之前/之后执行。

比如在before中new JWebDriver 和 传入chrome配置信息，在beforeEach中计数。