Web UI 自动化测试技术选型分析

事实上对于 UI **自动化测试**来说，许多所谓框架之间并没有太多差别，也从来不是影响整套**测试用例**是否健壮的关键性因素。相比之下，如何提高测试用例稳定性以及出现错误时 debug 的便捷性才是让 UI 自动化测试方案落地的重要细节。

那么为什么我们还需要讨论**技术**选型呢？首先我们来看看技术选型包含哪些部分。

通常 UI 自动化测试的技术方案分为控制（控制客户端）、执行（运行通过特定 API 编写的测试用例）、结果上报这几个主要组成部分，在过去各类框架往往喜欢在执行和结果上报两个部分提供差异化的 API 来提高开发效率，但这很难对我们开头提到的两个重要细节起到本质上的帮助。随着 **Web** 技术的不断演进，Web UI 自动化测试中的控制部分也终于有了更进一步的发展，而且这一部分正是解决用例稳定性、提升 debug 能力的核心所在。

接下来就对比一下目前可选的几种控制方案的优缺点。

selenium + webdriver

selenium 的方案最为传统，也是目前最常见的**浏览器**控制方法。selenium 通常需要和 webdriver 配合使用，selenium 通过 webdriver 控制浏览器，再对上层执行层暴露 API 或 sdk。同时 selenium 也提供 standalone **server** 的方案，允许执行层通过调用标准 restful API 控制浏览器，在这种模式下对执行层的编程语言和运行时都没有任何限制，这也是 selenium 生态繁荣的重要原因。

优点

selenium 的 API 封装遵循 W3C 提供的 webdriver 标准，因此 selenium 对各大主流浏览器的支持都不错，如果测试场景对浏览器兼容性有较高的要求，需要在多种浏览器中执行测试用例，selenium 仍是首选。同时由于 selenium 已经发展多年，各种解决方

案也更为完善。例如并行方案 selenium grid，可以支持多节点的用例负载均衡；还有在 CI 场景下官方维护的各种 docker image 等。

不足

selenium 是一套较重的方案，选择 selenium 意味着依赖 webdriver、**java**，如果执行层的编程语言不是 java，那么额外引入这些无疑是较为痛苦的。另一方面，selenium 为主的方案一般会形成一个“测试用例 -> 测试框架 -> selenium -> webdriver -> 浏览器”的复杂控制链，链上每多一环就意味着 debug 的复杂度上升一级，这会让测试用例的编写成本显著上升。

chrome devtools protocol

chrome devtools protocol（以下简称 CDP）可以看作 chrome（或其他使用 Blink 内核的浏览器）开放的远程控制协议。通过 CDP 我们可以控制 DOM、Debugger 和网络请求等浏览器内部领域，从而实现测试的目的。

相比之下 CDP 不依赖 webdriver 这样的二进制文件，也不绑定特定的编程语言，理论上可以直接在测试用例文件中和 CDP 通信，控制浏览器。但是实际上已经有很多成熟的 CDP 封装，大部分情况下我们使用它们而不是直接和 CDP 交互。

对 CDP 的封装可以大致分为两类，一类是只对网络通信部分封装，而不涉及启动浏览器、管理浏览器进程等工作。例如 chrome-remote-interface 就属于这类，只提供了良好的 JS API 封装，如果需要满足自动化测试的需求还需要搭配 chrome-launcher 这样的库对浏览器进程进行可编程的管理。

另一类则是一体化的方案，最佳实践是 google 自己维护的 puppeteer 项目。puppeteer 不仅负责管理浏览器和将 CDP 封装成 high level 的 API，甚至还默认下载一个指定版本的 chromium 并使用，以避免本地浏览器版本不确定带来的稳定性隐患。

优点

作为控制部分，CDP 的方案往往控制链较短，并且对编程语言没有限制，相对而言会更加稳定，并且也带来更好的执行速度。

chrome 在 59 版本开始引入了 headless 模式，这对 CI 流程来说有很大的帮助。过去如果需要在 linux 服务器上运行 Web UI 测试，通常都需要引入 xvfb 模拟屏幕，headless 模式则可以省去。

除此之外，CDP 相比 webdriver 标准控制能力更强，例如可以对网络层进行监听，从而可以轻松实现“所有网络请求完成后开始交互”这样的功能。

不足

CDP 方案只能兼容 Blink 内核的浏览器，因此对浏览器兼容性有要求的测试不应该选用。不论是 CDP 还是更进一步的 puppeteer，目标都是实现一个通用的自动化工具，而不是聚焦于测试。因此其内部没有集成测试相关的功能，例如断言、用例编排、结果上报、执行负载均衡等等，需要继续引入第三方库或自行实现，这作为一个测试框架而言不够理想。

Inject script

Inject script 的方式是指在浏览器打开的 Web 应用内注入测试引擎、测试用例等脚本，将测试用例执行在被测试应用的运行时中。

Cypress 和 Testcafe 这两个测试框架是这类控制方式的实践者，它们认为过去许多依托于 selenium 构建的测试框架的核心问题在于都是从外部控制浏览器和 Web 应用，执行命令或者获取信息都需要通过网络请求进行交互，因此交互的信息需要进行序列化。这不仅限制了交互的内容，还对 debug 带来了极大的不便，同时网络请求带来的开销也让测试变得更加缓慢。

与之相反的是 inject script 选择从内部控制浏览器，测试用例代码将和被测试的 Web 应用运行在同一个浏览器运行时中，可以理解为注入的脚本即为测试客户端，与后端建立通信，所有的操作指令都是通过 Javascipt 实现并执行，本质上只是函数的调用，客户端和后端之间的通信仅用于测试结果的收集，不包含具体的指令执行。

值得一提的是 selenium 1 中最早采用的也是类似 inject script 的实现，但是由于当时各个浏览器中的 Javascript 运行时缺乏统一的标准，执行结果不一致，因此产生了很多兼容性问题，selenium 才不得不通过标准更规范、并且有厂商支持的 webdriver 来进行控制。但时至今日，Javascript 规范已经非常成熟，主流浏览器的 Javascript 内核也都严格遵循规范予以实现，inject script 方案最大的弱点也就得以克服。

Cypress 和 Testcafe 的定位是完整的测试框架，并且只聚焦于完成 UI 自动化测试这一个任务，内部包含执行框架、断言、请求 mock、结果上报等所有测试所需功能，不需要再配置其它的依赖项。

由于控制流程的改进，双方在测试稳定性、运行速度方面都有很大的改善，可以被视作是下一代 UI 测试框架，但是两者也因为目标场景略有不同而在一些方面各有优劣，需要根据实际使用场景进行选择。

优点

用例执行的稳定性在两个框架中都作为最高优先级加以解决。许多 UI 测试的不稳定性来源于不能很好的处理界面中的异步情况，而 Cypress 和 Testcafe 在指令的 API 设计中就默认所有指令均有异步情况，会自动完成 wait 的处理。

此外同一运行时的设计在测试一些现代前端框架构建的应用时有巨大的优势，例如直接测试框架数据状态管理（如 redux）的正确性等等，支持编写一些更偏向**白盒测试**的用例。

双方也都有官方支持的 CI 集成方案，保证在 linux 服务器环境下也能够快速的搭建起运行环境。

不同之处在于 Testcafe 原本是商业产品，有着成熟的客户群体和解决方案，在转向开源之后依然能够发挥其经验丰富的优势，在一些实现上更注重易用性。因此会封装更高级的指令，例如拖拽、文件上传等等，并且所有指令均用原生 Javascript 实现，所以对浏览器的兼容性很好。还对测试用例提供了预处理器，所以用户可以选择用最新的 Javascript 特性编写用例或者使用 Typescript 这样的超集语言。

高级功能方面 testcafe 支持通过录制操作生成脚本，对于一些非开发人员来说提供了一定的便利性。

相比之下 Cypress 的目标不仅仅是做 selenium 的替代品，更希望成为革命性的测试框架，因此做了更多大刀阔斧的改进。

以上多次提到的 debug 问题是 Cypress 的最大优势，Cypress 实现了 DOM 快照、操作回放、指令可视化、网络请求监控等功能让用例执行失败的原因一目了然。

高级功能方面 Cypress 原生支持截屏、录屏等功能，进一步加强 headless 模式或 CI 模式下的 debug 能力。

Cypress 另一大高级功能是对应用中的网络请求实现监控、mock 等操作，让一些原本通过 UI 比较难以实现的断言可以转为对网络请求进行判断。

不足

双方的不足更多是和对方相比较之下产生的。

Testcafe 对高级功能的支持有所不足，例如录屏和 DOM 快照等高效的 debug 方案还未实现。

Cypress 部分指令和高级功能普通 Javascript 无法实现，需要借助 chrome extension API，因此目前并不能兼容所有的浏览器。并且由于特殊的控制实现方式，Cypress 不能支持跨浏览器、跨 tab 的测试需求，还要求被测试 Web 应用是同源的。

此外 Cypress 还有一些临时性的不足，例如用例执行时的负载均衡等高级功能的支持还不完整，一些原生操作例如文件上传也还需要通过 Javascript 模拟，不过这些临时问题都在 roadmap 上排期解决。