南京大學

NANJING UNIVERSITY

研 究 生 毕 业 论 文

（申请硕士专业学位）

论 文 题 目 Web 应用自动化测试系统执行服务的设计和实现

作 者 姓 名 尹子越

专 业 名 称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 刘嘉 副教授

2020 年 4 月 11 日

学　　号：MF1832225

论文答辩日期：2020 年 5 月 23 日

指 导 教 师：  （签字）

The Design and Implementation of Execution Service in Web Application Automation Testing System

By

Ziyue Yin

Supervised by

Associate Professor **Jia Liu**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Software Institute

April 2020

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目： Web 应用自动化测试系统执行服务的设计和实现

工程硕士（软件工程领域） 专业 2018 级硕士生姓名： 尹子越

指导教师（姓名、职称）： 刘嘉 副教授

摘 要

Web 应用随着互联网和前后端技术的发展，具有应用领域广、使用人数多、页面交互动态性和复杂性上升等特点。由于 Web 应用开发过程存在着频繁的迭代，对 Web 应用的自动化测试方法和工具也因此有很高的需求。目前存在的一些 Web 应用的自动化测试方法，存在测试效率不够高、无法对动态加载的应用进行测试的问题，无法应对频繁的测试需求。

本文设计和实现了一种 Web 应用自动化测试系统，用于解决复杂的动态 Web 页面的自动化测试的需求，生成测试缺陷报告和软件质量评估。测试启动所需要的配置简单。本文详细描述了该系统执行服务部分，服务将基于有限状态机的 Web 页面状态探测思想应用于测试执行中，搜索 Web 页面状态空间，生成可执行的测试路径。该服务基于微服务技术和分布式技术构建。执行服务总体分为测试接入模块、任务调度模块和执行引擎单元。其中接入服务采用 Spring Boot 构建微服务业务逻辑，实现与用户之间的交互。整个服务的分布式体现在调度模块和执行模块，依赖于 RabbitMQ 消息队列和 Docker 容器服务实现。服务通过分布式执行提高自动化测试的整体效率，通过容器减少对宿主机环境的依赖。执行引擎单元使用 Web 页面状态空间探测技术来捕获缺陷。服务添加了网站权限验证流程，以扩大应用的状态空间。

本文选取了常见的各种类型的网站，对该自动化测试服务进行了初步验证。本服务对 Web 应用在规定 1 小时的运行时间内，同时使用 3 种品牌的浏览器执行，每个应用平均可以探测 89 个 Web 应用状态，平均可以找出 83 个应用存在的缺陷。由执行服务构建的状态图生成的测试脚本执行成功率为 99.8%，脚本执行后缺陷复现率为 89.4%，缺陷分类准确率为 97.5%。同时测试服务相比人工能够提高 2.12 倍的测试效率。目前该系统已经可以为慕测平台企业版 Web 自动化测试任务提供服务，支撑 Web 自动化测试业务发展。

关键词： Web 应用测试，自动化测试，微服务，分布式系统

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of Execution Service in Web Application Automation Testing System

SPECIALIZATION: Software Engineering

POSTGRADUATE: Ziyue Yin

MENTOR: Associate Professor Jia Liu

Abstract

With the development of the Internet and front-end technology, Web applications have the characteristics of wide application fields, a large number of users, dynamic and complex page interaction, and so on. Because there are frequent iterations in the web application development process, it is a high demand for automated test methods and tools for web applications. There are currently some automated testing methods for Web applications, which have the problems of insufficient test efficiency and the inability to test dynamically loaded applications. They are unable to cope with frequent testing needs.

This thesis designs and implements a Web application automation test system, which is used to solve the needs of automated testing of complex dynamic Web pages, generate testing vulnerabilities reports and software quality assessment. The configuration required for a testing startup is simple. This thesis describes a execution service as a part of the system in detail. The service applies the idea of Web page state detection based on finite state machine to test execution, searches the web page state space, and generates executable test paths.The service is based on microservice technology and distributed technology. The execution service is generally divided into a test access module, a task scheduling module, and an execution engine unit. Among them, the access service uses Spring Boot to build microservice business logic to achieve interaction with users. The entire service is distributed in the scheduling module and the execution module, which depends on the RabbitMQ message queue and Docker container service implementation. The service improves the overall efficiency of automated testing through distributed execution and reduces the dependency on the host environment through containers. The execution engine unit uses Web page state space

detection technology to capture vulnerabilities. The service adds a website permission verification process to expand the state space of the application.

This thesis selected common types of websites and conducted preliminary verification of this automated testing service. This service executes web applications within a specified one-hour running time using three brands of browsers at the same time. Each application can detect an average of 89 web application states, and on average can find out 83 vulnerabilities on each application. The execution success rate of the test scripts generated by the state diagram constructed by the execution service is 99.8%. The vulnerabilities recurrence rate after script execution is 89.4%, and the vulnerabilities classification accuracy rate is 97.5%. At the same time, the testing service can improve the testing efficiency by 2.12 times compared to manual. At present, the system can already provide services for the Web automation testing task of the MOOCTEST platform enterprise version, and support the development of Web automation testing business.

Keywords: Web Application Testing, Automated Testing, Microservice, Distributed System

目录

表 目 录

图 目 录

第一章　引言

1.1　项目背景

从商业、工业、教育、娱乐、政府到个人生活，网络已经对我们社会各个方面都产生了重大的影响。而 B/S 架构的 Web 应用相较于传统桌面应用来说有如下优点：Web 应用在浏览器上运行，除了浏览器，用户无需其他安装成本；在 Web 应用的新版本升级后，所有用户同步升级，降低了维护和支持成本；可以从任何可以访问 Internet 的计算机上的任何浏览器进行访问应用程序和用户数据。正是这些优势使得 Web 应用及其相关技术的繁荣发展。

当前已经有非常多的 Web 应用部署在互联网中。根据 InternetLiveStats 网站[1]的统计，目前在线的网站已经有 17 亿多个，并且这个数字还在快速上升中。同时，Web 技术的发展使得 Web 应用愈发复杂。Web 1.0 最初是一个让企业向人们广播他们的信息的场所，早期的网络提供了有限的用户互动或内容贡献，并且仅允许搜索和阅读信息 [1]。Web 2.0 比其前身 Web 1.0 更具动态性和交互性，让用户既可以从网站访问内容，又可以做出贡献 [2]。Web 3.0 时代的则是应用综合用户信息和互联网上的信息并整合反馈，我们实质上可以将 Web 3.0 视为集成到大型 Web 应用程序中或为大型 Web 应用程序提供支持的语义 Web 技术 [3]。在这些技术的更迭中，Web 应用逻辑复杂程度不断上升。实际的生产中对 Web 应用使用的满意度会影响用户对该产品和对应公司的忠诚度 [4, 5]，一些微小的缺陷都可能带给公司难以预计的损失。并且基于 Web 的应用程序与其他应用程序不同，因为它们需要高可靠性和可用性，产品生命周期更短，上市时间更短 [6]。在 Web 应用的开发方面，开发团队被建议使用使用敏捷开发模型 [7]，以满足客户频繁提出的需求。整个开发过程迭代周期短，迭代次数多。无论是传统的瀑布模型流程，还是新兴的敏捷开发，软件测试是都保障软件功能、性能、可用性、兼容性、稳定性等符合客户要求的重要手段 [8]，因此对 Web 应用进行测试的必要性和重要性不言而喻。在一个软件开发生命周期中，大约 70％的时间都花在了测试上，我们有必要使用技术手段来减少测试时间，提高开发效率；另一方面，测试 Web 应用程序任务会更加繁重：兼容各种操作系统、不同的硬件和网络连接，多种品牌的浏览器和 Web 服务器以及技术和编程语言的多样化使用使 Web 应用的测试变得更加复杂 [9]。Ali 等人的文章提出了结论表明使用脚本自动测试可以提高测试范围，也降低了测试成本，并且相较于传统测试模

[1]https://www.internetlivestats.com，互联网在线数据统计网，数据基于 ITU、世界银行和联合国。

型，敏捷模型的表现更加优秀 [10]。

从用户交互技术上来说，Web 应用正在从传统的静态内容和表单交互转为动态内容加载。许多最新的技术，例如 Angular[2]、Vue.js[3]和 React[4]，基本都是使用 AJAX 技术[5]在浏览器和服务器之间进行交互，以提升 Web 应用的响应性 [11]。单页应用（SPA, Single Page Application）也被更多地在实践中使用，其中最重要的技术是异步获取数据并更新页面内容或整个组件，而无需浏览器刷新。此更新可以是部分组件更新，也可以是将组件完全替换为另一个组件 [12, 13]。这些技术使得用户界面更加具有响应性并且减少了用户体验的延迟，对 Web 应用程序的用户友好性和交互性产生积极影响 [14]。

为了检测故障，测试方法应满足以下条件：到达故障执行位置；触发错误产生不正确的中间状态；传播错误，从而使错误的中间状态传播并导致可检测的输出错误。与经典的 Web 应用程序相比，对于 AJAX 应用程序而言，满足这些到达/触发/传播条件更加困难 [15]。在单页应用中，静态分析工具在获取到目标 Web 页面后，可供分析的仅有 HTML 源代码、CSS 文件以及相应的 JavaScript 静态脚本。在没有经过动态的数据请求和动态加载时，一个单页应用可供分析的源代码可能只有短短的十几行 HTML，甚至没有任何可以体现应用功能的信息，静态分析工具也无从下手。

Selenium[6]提供了一种有效自动化 Web 测试的解决方案：使用 WebDriver 控制浏览器执行测试脚本，以便达到 Web 页面的功能测试目的。但是，测试脚本的编写免不了测试人员的人工介入。测试脚本的编写途径主要有两种：通过某些工具拦截浏览器行为“捕捉/回放”用户操作；测试人员直接参与功能流程脚本的编写。往往用户界面发生改变时已有的测试就需要重新修改，且编写脚本对测试人员设计能力又有较高要求 [16]。在当前 Web 应用快速迭代的场景下，人工测试脚本编写普遍存在效率较低、测试路径覆盖不够全面的问题。甚至由于网站开发者为了有效地吸引用户、针对用户会个性化定制页面，导致每个版本的页面之间元素差别比较大，人工编写的脚本几乎无法长时间复用，大量人力资源被浪费在重新编写测试脚本上。

如何在尽量减少人工干预的情况下，对当前内容丰富、动态交互的 Web 应用进行测试，成为了一个难题。本文针对这种情况，设计了一种自动化的 Web 应用测试系统执行服务，提高测试效率，并且适用于任意前后端技术开发的 Web

[2]https://angular.io

[3]https://cn.vuejs.org

[4]https://reactjs.org

[5]Asynchronous Javascript And XML（异步 JavaScript 和 XML）

[6]https://www.seleniumhq.org/

应用。

1.2　国内外研究现状

早在 2001 年，Ricca 和 Tonella 已经意识到由于 Web 系统的性质，Web 应用趋向于快速发展并且会导致经常性的修改 [17]。因此，他们提出了一种基于 UML 分析模型，以支持 Web 应用程序演化分析和测试用例的生成，并给出了测试工具 ReWeb。Andrews 等人提出了一种基于有限状态机和测试人员给出的约束，为大型的 Web 应用程序建模的方法 [18]。Liu 等人整合当时的 GUI 模型，提出一个通用的事件流模型，代表事件和事件交互：控制流模型表示程序中所有可能的执行路径，数据流模型表示所有的定义，提出了事件空间探索策略 [19]。在国内，牟和顾等人基于 UML 活动图实现测试用例生成，给出了活动图形式化描述及测试覆盖准则的定义 [20]。在此基础上刘和吴等人采用 UML 活动图代替有限状态机，更加方便对 Web 应用程序的业务流程进行建模 [21]。这些技术将传统的基于路径的测试生成和控制流或数据流充分性评估扩展到了 Web 应用程序领域。

通过记录和分析服务器与用户的会话过程也经常被用于 Web 应用的测试。Elbaum 和 Rothermel 通过捕获用户会话期间的数据，与白盒功能测试技术结合，用于为测试工程师提供有效的测试用例 [22]。这种方法要求用户与服务器充分交互，并且需要在服务器端记录有效的日志数据。但根据服务器的增量日志信息，也很难推断出 Web 应用异步消息之间的状态流转。有时服务器的数据在正式被注入 Web 应用页面的 DOM 中后才能表示出其真正的含义。

2013 年，由 Zou 和 Chen 等人针对采用 AJAX 动态交互的 Web 应用提出了一种动态 Web 测试的混合标准，定义了 HTML 元素覆盖和应用的混合覆盖。混合覆盖的关键思想是结合 Web 应用程序的运行时客户端和服务器端功能，基于语句覆盖和元素覆盖定义而成 [23]。一年后，Zou 和 Chen 等人结合服务器代码和浏览器结果的测试方法构建 Visual-DOM 树，提出了一种新颖的覆盖标准，即 V-DOM 覆盖，以促进对动态 Web 应用程序的有效测试 [24]。其方法对服务器端代码（例如 PHP）执行静态分析，以识别服务器代码可能释放到返回给客户端的页面的所有可能的 DOM 对象，并像真实的 DOM 树一样将它们组织在树中。在另一篇文章中，Marchetto、Alessandro 和 Tonella 等人也提出了类似的想法，通过有限状态机的形式去构建基于 AJAX 技术的 Web 应用状态图 [25]，并将其应用到测试中。

Stocco 和 Leotta 等人提出了一种“Page Object”页面对象模式用于 Web 应用程序的脚本测试 [26]。使用页面对象设计模式将测试规范与其实现分开，在

更具有可读性和业务重点的幕后隐藏了有关测试代码如何与网页交互的技术细节。页面对象充当 Web 应用程序的可编程接口：将 GUI 表示为一系列面向对象的类，将每个页面提供的功能封装到方法中。该论文设计了 APOGEN 工具用于自动生成页面对象，用于完全反映网页的结构，并具有以下功能：可执行的元素、生成导航图方法、填写和提交表格的方法，提供了一种更为完成的页面对象生成方法，大大减少测试人员的手工开发工作。但是该工具无法自动完成网页交互流程的生成，还需人工介入。

Ravi 和 Hyunsook 等人为了减少对使用 PHP 开发的 Web 应用每次迭代后进行回归测试的时间和资源，提出了使用 PHP 分析和回归测试引擎（PARTE）之类的工具来识别代码路径。他们将 PARTE 扩展到测试选择，可以检测到因为迭代开发导致的测试覆盖修改后的代码路径的现有测试的子集。为了进一步减少所需的回归测试量，使用了 PARTE 的可重用约束值信息来确定可以针对新版本重用的测试，而无需修改输入的测试值 [27]。其技术主要针对 PHP 语言，并且仍然需要部分测试路径的开发。

Selenium 以及其核心的 WebDriver 的推出使得测试人员可以通过代码来控制真实的浏览器模拟用户进行操作。Holmes 和 Kellogg 等人使用 Selenium 为基于敏捷开发的 Web 应用程序编写测试脚本用于回归测试，以保证应用程序的稳定性 [28]。Selenium 还为 Firefox[7]和 Chrome[8]浏览器提供了更加方便的 Selenium IDE，可以直接录制测试人员在界面的操作流程，并生成对应的脚本代码。与 Selenium 类似的工具还有 Quick Test Professional[9]和 Test Complete[10]。在 Selenium 的基础上也有很多工具进行了封装，用于提高测试并发化和无界面化，以及提高自动化测试框架的部署效率 [29]。这些工具完成的是 Web 应用程序测试脚本的编写和回放工作，基于编写完成的测试脚本，进行回放并给出测试结果。

Sauce Labs[11]构建了一个云测试平台，为客户提供基于 Selenium/Appium 的浏览器兼容性测试 PaaS[12]云解决方案，同时也支持 Selenium 脚本的自动化测试。国内类似的工具比如：阿里开源了 F2etest[13]，提供一个面向前端、测试、产品等岗位的多浏览器兼容性测试整体解决方案，测试人员可以直接在浏览器中访问搭建完毕的浏览器云，并在浏览器云中进行 Web 应用的测试而无需自行搭建

[7]https://www.firefox.com.cn/

[8]https://www.google.cn/intl/zh-CN/chrome/

[9]https://quicktestprofessional.wordpress.com

[10]https://smartbear.com/product/testcomplete/overview/

[11]https://www.saucelabs.com

[12]Platform as a Service，平台即服务

[13]https://www.f2etest.net

测试环境。以上工具的测试过程均需要依赖于测试人员编写的自动化测试脚本，工具提供的是基础运行环境和基于脚本运行情况生成的日志和报告。

“无限猴子定理” [30] 也被应用于界面有关的测试 [31]。开发者认为，如果对应用进行长时间的随机点击，那么很有可能发现大部分的应用问题。Google 为自家的 Android[14]的开发提供了一套 Monkey Test 的测试工具，其作用是在 Android 应用程序界面进行一系列随机的事件流，包括点击事件、滑动事件、触屏输入事件等，对开发中的 Android 应用程序界面进行一系列的压力测试，用于验证系统的可靠性。在 Web 应用测试方面也有类似的设计，gremlin.js[15]提供了一套 Monkey Test 的解决方案。该库基于 JavaScript 开发，为 Node.js[16]和浏览器提供测试方法以验证应用程序的鲁棒性，避免由于非常规的用户操作导致的 Web 应用程序崩溃。

1.3　本文主要工作

本文设计了一个 Web 自动化测试系统，解决对当前内容丰富、种类繁多的 Web 应用程序（包括经典多页应用和当下流行的单页应用）的测试，对 Web 应用进行多个维度的评估，并且从测试发起开始到测试报告生成实现基本自动化。本文的主要内容关注于该系统的自动化执行服务，用于解决人工测试和人工编写测试脚本效率不够高，传统自动化测试工具无法完成动态加载应用测试的问题。该服务对给定的任意 Web 应用，自动搜索可行的测试路径并模拟用户的操作流程，在执行过程中记录 Web 应用的状态等信息用于最终测试报告的生成。其中，主要工作有：

在经典 Web 应用测试中，有限状态机和约束条件的思想有着不错的表现。本文依赖于有限状态机的思想，使用了一种 Web 应用状态空间的探索的方法，并且经过对异步交互的支持，能够将其应用于经典多页 Web 应用和 AJAX 动态交互的 Web 应用。

对于 Web 应用的评价方式需要从多个维度上进行考虑。本文的测试执行实现对于单个种类的浏览器中测试路径的生成用于探知 Web 应用整体功能和性能上的缺陷。同时，可以通过将测试任务在不同的操作系统和浏览器环境下运行，本文实现探知不同环境下 Web 应用兼容性的缺陷。

为了减少测试人员需要干预整个测试流程的程度，本文定义了一系列测试执行的配置。这些配置在测试开始执行时输入服务的执行引擎，作为执行引擎

[14]https://developer.android.google.cn

[15]https://github.com/marmelab/gremlins.js/

[16]https://nodejs.org/en/

的启动配置，以减少测试执行过程中人工干预的程度。

本文提供了整个执行服务的完整工程设计：详细阐述从测试请求的接收到服务执行结束的系统完整流程。本文设计了一系列措施的保障整个执行服务在高并发下的可用性，同时使用一系列分布式独立测试客户端同时开展测试。在有进程间同步保障的情况下，加快测试路径的整体生成速度，保证测试执行过程的效率。

1.4　本文的组织结构

本文研究的 Web 自动化测试系统执行服务，将从软件开发的整个生命周期，即需求、设计、实现和测试环节，组织文章结构，并对其进行论述：

第一章为引言部分。主要阐明 Web 自动化测试系统和该系统执行服务的相关项目背景。简要描述了国内外对于 Web 自动化测试的部分研究和实现工具，包括一些开源工具和商业软件。在此基础上，表明了当前 Web 开发技术环境下，对 Web 应用的自动化测试面临的问题，本文实现的系统需要解决的问题和相关的研究工作。最后简要介绍本文的组织结构。

第二章为技术概述。主要介绍了 Web 自动化测试系统项目和该系统执行服务所涉及的技术，以及在本文所设计的系统中使用该技术的原因和见解。

第三章为需求分析和概要设计。根据前期的背景和调研，分析出用户对该系统的主要需求。根据用户需求，进行系统的体系结构设计。阐述了对系统的设计你思路，划分系统模块，将整个项目划分为测试接入模块、任务调度模块、执行引擎单元。使用系统架构图等方式对整个项目的体系结构进行描述。

第四章为详细设计与实现。基于第三章的需求分析和概要设计，对测试接入模块、任务调度模块、执行引擎单元等模块的详细设计思路和实现方式进行完整的阐述。给出核心流程的代码实现和组件交互方式。在最后给出对整个项目的测试结果和使用效果。

第五章为系统测试。在完成第一章的从背景分析到第四章的详细设计实现后，在第五章将对本文设计的系统根据需求进行整体的测试并给出相应的结果。测试结果将分为整体结果和典型案例进行详细叙述。

第六章为总结与展望。对在 Web 自动化测试系统和该系统执行服务开发过程和在论文撰写过程中的工作进行总结，对 Web 自动化测试系统项目方面的未来扩展方向作进一步的思考和规划。

第二章　相关技术概述

本章主要内容为介绍 Web 应用自动化测试系统执行服务的开发过程中所涉及到的部分关键的概念和技术。

2.1　服务框架

服务框架技术部分主要介绍构建 Web 应用自动化测试系统执行服务主体程序所依赖的成熟的框架技术。执行服务的总体设计理念是按照微服务架构设计，方便应对低成本扩容、弹性伸缩、适应云环境的需求。以下技术顺应微服务潮流并且非常适合用于执行服务的整体构建。

2.1.1　Spring Boot

Spring Boot 作为 Spring 家族的一员，在近年来流行的微服务架构中大放异彩。Spring Boot 的宗旨是“Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can 'just run'.” [32, 33]，为开发者们提供了解决方案来创建独立的、可用于生产的 Spring 应用程序。

Spring 框架为我们提供了良好的 IoC(Inversion of Control, 控制反转)、DI (Dependency Injection, 依赖注入) 和 AOP(Aspect Oriented Programming, 面向切面编程) 的方法。在复杂的应用程序开发中，Spring 为我们实现了资源的集中管理，实现资源的可配置，降低使用资源的双方的依赖程度，也就是耦合度。开发者仅需要专注于业务逻辑的实现。但是早期的 Spring 框架需要使用大量的 XML 文件进行严格的配置，十分繁琐，不适合于快捷开发。Spring 应用程序的启动也依赖于 Tomcat[1]之类的容器。

Spring Boot 提供了一系列通过使用注解的自动配置用于减少快速开发中繁琐的通过 XML 配置 Spring 和第三方库。由于 Spring Boot 内嵌了 Tomcat/Jetty[2]/Undertow[3]容器，开发人员在部署 Spring Boot 应用的时候无需另外部署 Web 容器。通过 FatJar 技术，将应用的 jar 与依赖的第三方 jar 文件全部打进一个包中，项目的最终产物仅仅是一个 jar 包，非常清晰明了，同时应用启动十分方便。

本文设计的系统需要对外提供 Web 服务，用户通过 HTTP 请求访问服务并启动测试流程。Spring Boot 及其第三方库具有强大的 Web 服务构建功能，并且

[1]http://tomcat.apache.org

[2]http://www.eclipse.org/jetty/

[3]http://undertow.io/

开发便捷、集成简单、运行稳定、容易维护，所以将 Spring Boot 定为本文系统的 Web 服务开发框架。

2.1.2　Feign

Feign[4]是一个声明式的 Web 服务客户端。使用 Feign 可以将原本需要通过复杂的工具类构件的 Java 内部的 HTTP 客户端变得十分简单。与 Spring 框架结合后，Feign 可以通过在接口上添加注解的方式，将接口解释为 HTTP 客户端，并从方法上的注解中读出请求的 URL、请求的方法（如 GET 请求、POST 请求等），将方法参数变为 HTTP 请求的参数。

同时，Feign 还集成了可插拔式的编码器和解码器，比如常用于将方法参数的 Java 实体类对象转为 JSON 字符串类型的 HTTP 请求体，将 HTTP 响应的 JSON 字符串类型的响应体转为方法的返回值实体类。Feign 也对请求失败的情况做了一些简单的异常处理，在 Spring Cloud[5]中与 Hystrix[6]相结合可以实现服务熔断和服务降级的功能。

在本项目的设计中，测试接入模块与测试调度模块之间的消息通讯采用 HTTP 形式，Feign 注解形式的 HTTP 客户端简化了模块内部编写 HTTP 请求方法的步骤，并且有良好的稳定性保障，与 Spring 框架也有很好的兼容性。

2.1.3　Docker

Docker 作为使用 Go 语言开发的开源应用容器引擎，也是随着微服务的流行而不断扩大自己的影响力。在 Docker 诞生以前，应用程序的部署需要在实体机上重新配置依赖环境，部署的大部分精力被花在应用环境兼容上。虚拟机虽然能够作为一个解决方案，但是虚拟机的镜像相对太大，并且启动缓慢。

Docker 使用容器技术达到轻量级的虚拟化。对于某些应用程序来说，虚拟机提供的完整的操作系统环境十分冗余，而 Docker 则是虚拟了一个小规模的环境，提供应用程序能够稳定运行的资源。因此，Docker 占用空间小，启动速度十分快捷，并且资源利用效率极高。在应用程序开发完成后，使用 Docker 将应用程序打包成一个 Docker 镜像。这个 Docker 镜像包含了应用所需要的一切程序、库、资源、配置以及一些环境变量。镜像可以在任何安装了 Docker 的宿主机上使用一条简单的命令运行而不需要更多的环境配置。正如 Docker 官网所说的“Debug your app, not your environment”[34]。

如图 2.1所示，本文设计的系统中，需要分布式地部署多个 MTWebClient 对

[4]https://github.com/OpenFeign/feign

[5]https://spring.io/projects/spring-cloud/

[6]https://github.com/Netflix/Hystrix

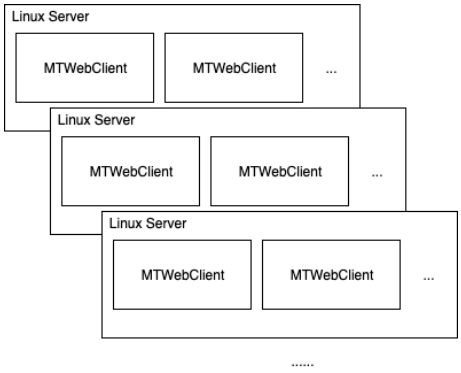


图 2.1: MTWebClient 分布式部署示例

多版本的浏览器进行路径搜索。Docker 十分契合该技术场景，为本项目减少人工干预部署环境的时间比重。

2.1.4　Celery

Celery 是一个分布式异步任务调度工具。作为一个分布式系统，Celery 由一个 Broker 中间件，多个 Worker 任务执行单元和一个 Backend 任务结果存储组成。Broker 作为任务调度的中间人，负责传递任务信息。每当应用程序调用 Celery 的时候，会向 Broker 中发送消息，而 Worker 会监听 Broker 中的消息并且获取调度的任务。Celery 本身并不会提供消息中间件的实现，这里我们使用 RabbitMQ 作为 Celery 的消息中间件实体。当 Worker 完成任务后，会将任务结果回写到 Backend 中，供任务发起者查询。

本文设计的系统中一个测试任务将会被分发到多个测试执行客户端 MTWebClient 处理，此时采用 Celery 进行任务调度，可以达成快捷、高效、稳定的调度和执行这一结果。

2.2　数据服务

Web 应用自动化测试系统执行服务需要高效的数据服务为其提供支持。由于整个执行服务存在分布式并行执行的情况，数据服务必须有效应对高并发的需求，是具有高性能读写特性的服务。

2.2.1　RabbitMQ

现代的分布式系统具有模块化体系结构，分布式的应用程序，设备或设备需要进行连接和扩展。这些应用程序也会作为大型应用程序的组件彼此连接，或者需要与用户设备和数据进行连接。其中的主要要求是克服点对点通信的限制，而且要以非同步的方式进行 [35]。

RabbitMQ 提供了一款轻量级的高性能消息队列实现。消息队列作为一个 FIFO(先进先出) 的队列，其中流转的是一条条的消息。消息队列在解决多个进程/线程间通讯方面有出色的表现，通过异步处理提高整个系统的性能，降低系统的耦合程度。

本文设计的系统会有多个测试路径生成的客户端 MTWebClient 进程同时运行，并且会有一个统一的报告生成服务接收测试过程中产生的各种日志和信息。使用 RabbitMQ 可以让我们的系统有一个规范化的消息通讯中间件，并且可以实现 MTWebClient 客户端和分析服务之间的负载均衡。RabbitMQ 同时也提供了一个友好的 Web 后台管理界面，方便观察消息队列负载、排查产生的问题。

2.2.2　Redis

Redis 是一个开源的，可基于内存亦可持久化的 key-value 数据库。Redis 不仅能保存 string(字符串) 类型的数据，还能保存 list(列表) 类型、set(集合) 类型、hash(哈希) 类型和 zset(有序集合) 类型的数据。

Redis 采用 C 语言编写，与操作系统的交互方便；完全基于内存，读写时操作系统可以直接在内存中操作；核心操作由单线程完成，避免了不必要的上下文切换和多线程竞争；基于 Linux 的 epoll 实现 I/O 多路复用，保证单线程也能接受很多的连接请求。由于以上的特点，Redis 可以达到很高的读写性能。在启用了 pipeling，允许一次执行多条命令后，Redis 在 Macbook Air 11 上可以执行每秒钟 40 万次写操作、50 万次读操作 [36]。Redis 强大的性能使得其常常被用来作为缓存, 并且由于内嵌 list 类型的数据格式后，也可作为消息队列的实现。

由于使用消息队列传递大量结构型数据在工程实践中不是一种好的选择，本文设计的 MTWebClient 实时获取的测试数据将会采用 Redis 作为临时存储，而消息队列仅仅传递任务完成的信号。这样保证了在多个进程间传递大量数据的速度。同时, Redis 也提供了一种分布式锁的实现方式, 可以用于 MTWebClient 之间的同步。

2.2.3　MongoDB

MongoDB 是一款优秀的开源分布式文档数据库。作为一款非关系型数据库，其存储方式与基于数据关系模型的关系型数据库 [37] 不同。使用 MySQL 和 Oracle 这些关系型数据库，一般在设计数据结构的时候，会将一个实体类型映射到一张表，每一个实体是表里的一行数据。但是 MongoDB 作为一款文档数据库，其保存的数据结构可以直接按照数据的原有格式以 Document（文档）的形式保存在一个 Collection(集合) 中，更加灵活。

MongoDB 的文档以 key-value 键值对来表示实体的属性和对应的值，这种表示方法与常用于 HTTP 传输的 JSON 数据格式类似。因此，从 MongoDB 中取出的数据在经过一些简单的处理之后，可以直接用于 JSON 传输而不需要更多的格式转换。

本文的系统使用 MongoDB 存储自动化测试的最终结果报告，主要的原因有：1）测试报告数据格式存在多层嵌套，使用关系型数据库表示需要采用多张关联表保存实体之间的关联关系，并且在查询时关联多表容易造成慢查询；2）报告数据需要导出成 JSON 格式交由前端生成可视化报告，因此 MongoDB 在数据转换上可以节省很多步骤。

2.3　自动化测试依赖组件

Web 应用自动化测试系统执行服务构建基于有限状态机的 Web 状态空间探索模型。同时，为了应对不同技术开发的 Web 应用，必须通过真实的浏览器访问模拟用户操作来保证对 JavaScript 动态加载事件的监听。以下开源技术符合执行服务的设计需求。

2.3.1　Crawljax

Crawljax[7]作为一个能够支持爬取动态 Web 应用网页内容的开源项目，十分适合本文所设计的基于有限状态机的测试路径搜索方法。其中 Crawljax 通过使用 Selenium 的 WebDriver，控制真实的浏览器访问 Web 应用，并且模拟用户的操作后获取 Web 应用内容和状态信息。

Crawljax 在爬取网页的过程中会将整个 Web 应用的状态流转绘制成一张有向图，方便追溯爬取过的各个状态。由于没有一个幂等的返回操作可以从一个网页的状态节点（比如节点 B）退回上一个状态节点（比如节点 A），在结束节点 B 的爬取需要回退的过程中，Crawljax 都会从最初始的状态节点（Index）开始，重新执行一遍动作以便回到上一个状态节点（节点 A）。

为了减少每次回溯需要重新执行的动作，最短路径算法被应用到该路径搜索中。Crawljax 所引用的第三方 Java 图算法库 JGraphT[8]中，使用 Dijkstra 算法搜索最短路径。Dijkstra 算法计算从单个源节点到图中所有其他节点的最短路径，从而生成最短路径树。JGraphT 中的 Dijkstra 算法基于 Fibonacci 堆实现的最小优先级队列，在运行时的时间复杂度为 $O(m + nlogn)$[38, 39]。

[7]https://github.com/crawljax/crawljax

[8]https://jgrapht.org/

2.3.2　Selenium Grid

Selenium Grid 是 Selenium 的三大组件（Selenium Webdriver, Selenium IDE, Selenium Grid）之一。其作用就是提供便捷的分布式测试执行方案。当自动化测试用例达到一定数量的时候，单台机器的性能已经不足以在规定测试时间内执行完所有的测试用例，或者是有多版本的浏览器的测试场景的时候，分布式测试执行成为了一个必然的选择。

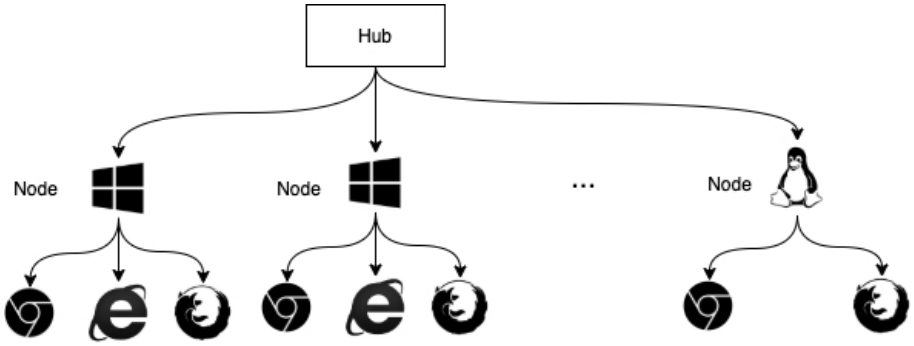


图 2.2: Selenium Grid 分布式架构图

如图 2.2所示，Selenium Grid 主要分为两个角色：Hub 和 Node。Hub 节点是一个资源管理和调度的节点。Hub 会管理各个 Node 代理节点的注册信息：每个 Node 节点的操作系统，负责的浏览器型号、版本，浏览器的可用状态等。另外，Hub 是接收远程客户端代码调用请求的入口。在接收到请求后，Hub 会将请求的命令转发给 Node 节点执行。Node 节点是真正与浏览器交互的进程。在指定好浏览器型号、版本，配置好对应的 WebDriver 后，Node 节点向 Hub 节点注册节点信息，接收 Hub 的调用命令，执行自动化脚本。

在本项目设计的系统中，Selenium Grid 提供唯一的客户端 WebDriver 调用接口，对客户端隐藏 Node 节点的浏览器调度实况，极大减少了客户端对调用 WebDriver 接口的环境配置要求。同时，通过将浏览器统一进行管理，也方便后续通过增加服务器资源，提高测试生成服务的运行效率。

2.4　本章小结

本章介绍了 Web 自动化测试系统执行服务主要涉及的一些技术和概念，阐明了这些技术在系统中所起到的重要作用。本文基于这些技术，构建起系统的主要功能，并且为系统的可用性、稳定性和可扩展性打下基础，方便后续系统的改进。

第三章　需求分析与概要设计

3.1　Web 自动化测试系统总体规划

Web 自动化测试系统实现了一个自动化测试服务，接入慕测企业版平台的业务逻辑中。在慕测企业版平台的业务逻辑模块主要包括：Web 自动化测试任务业务模块，授权业务模块。其中 Web 自动化测试任务业务模块主要负责 Web 自动化测试任务的查看、新建、发起执行、报告查看和发布众审一系列自动化测试服务相关功能。授权业务模块负责处理管理员对新建的自动化测试任务的审批操作。

自动化测试服务是 Web 自动化测试系统的核心部分。其主要分为执行服务、报告生成服务和分析服务三个部分。执行服务负责处理平台的测试任务并启动测试任务，报告生成服务获取测试数据交由分析服务分析后生成报告。

3.2　Web 自动化测试系统执行服务概述

Web 自动化测试系统执行服务属于 Web 自动化测试系统的一部分。功能覆盖范围从测试请求接收、测试任务分配到测试路径生成、获取原始数据。是整个 Web 自动化测试系统中的底层架构，也是较为重要的一个环节。

3.3　Web 自动化测试系统需求分析

3.3.1　用户群体

Web 自动化测试系统接入慕测企业版平台，负责对外提供自动化的测试服务。主要针对的是需要对自己的 Web 应用产品进行快速测试并且获取有效测试报告的群体。这些用户可能没有足够的人手对每次迭代开发的 Web 应用进行详细的测试，因此需要通过 Web 自动化测试帮他们快速了解目前系统中存在的明显的缺陷。针对教育方面，Web 自动化测试系统提供给一些教师根据报告快速对某些课程学生提交的 Web 应用进行打分的可能。整个系统还可以为那些需要对自己的 Web 应用产品质量进行快速评估的群体，提供有效的评估功能。

3.3.2　Web 自动化测试系统功能需求分析

由于 Web 自动化测试系统是提供一类特化的服务，因此系统的功能需求整体上来说十分简单，即提供一个有效的 Web 自动化测试服务，去满足用户对 Web

应用的自动化测试需要，减少用户在软件开发生命周期中花费在测试阶段的时间。在用户指定了 Web 应用的 URL[1]后，等待一段时间就可以获取对该网站的一份较为详细的测试报告。用户在指定完成 Web 应用的 URL 配置后也可以通过一些自定义配置去控制服务的执行过程。应用测试的结果除了从应用中收集和综合分析得到的应用的详细测试缺陷报告外，还具有从功能、性能、稳定性、兼容性、健壮性五个维度对整体应用质量进行评价，以及每个维度切面的钻取数据分类。

表 3.1: 功能需求列表

需求 ID	**需求名称**	**需求描述**
R1	查看 Web 测试任务	用户能够通过系统查看自己创建的所有测试任务，并且能够对测试任务进行操作。测试任务按照创建的时间进行排序。
R2	创建 Web 测试任务	用户能够通过设置测试目标和对测试的配置创建一个新的测试任务。用户可以选择从历史的测试目标中选择，也可以重新创建。
R3	执行 Web 测试任务	用户发起执行未执行过的测试任务，等待测试报告生成。系统需要告诉用户测试任务目前的状态。
R4	查看 Web 测试缺陷报告	用户在等待测试流程完成后，可以查看生成的测试缺陷报告。测试报告对 BUG 进行类型和严重等级分类，使用图表和文字充分表达测试结果。
R5	查看 Web 应用质量多维评估	用户在等待测试流程完成后，可以查看对该应用的质量五维评估，分别从功能、性能、稳定性、健壮性、兼容性方面对应用进行测评。
R6	审核 Web 测试任务	管理员需要能够对平台用户发起的测试任务进行审核，在确认测试任务合格后可以允许用户执行测试任务。

在对接平台方面，针对 Web 自动化测试任务，需要有测试任务管理的功能。用户所关注的点是围绕 Web 自动化测试任务所展开的：用户可以在平台上查看当前用户建立的所有 Web 自动化测试任务；用户需要新建测试任务的能力，在

[1]Uniform Resource Locator, 统一资源定位符

选择应用网站 URL 后，还需要对本次测试任务进行一定的参数配置以保证执行过程符合用户的需求；用户还需要触发测试任务的能力，标志着测试任务的正式开始；在测试任务结束后，用户有查看测试缺陷报告的需求，测试缺陷报告对用户提交的 Web 应用进行全方面的分析；用户同样具有查看应用质量多维评估的需求，从五个具体维度了解应用的质量情况。

为了避免平台用户使用平台执行一些不合适的测试任务，对平台造成一定损失，我们需要有对测试任务的授权功能。管理员可以在测试任务授权模块中检查对应测试任务，并给予其运行的权限或者拒绝。Web 自动化测试系统的主要功能性需求列表如表 3.1所示。

3.3.3　Web 自动化测试系统非功能需求分析

表 3.2: 非功能需求列表

非功能需求	**需求描述**
可用性	可用性主要指系统在一定时间内处于可执行规定功能状态的能力。对于本系统来说，可用性是系统在稳定的外部环境条件下能够保持 7×24 时间的持续运行。
易用性	易用性是为了保证用户能够快速上手系统，符合用户使用习惯并增加其使用的欲望。 对于本系统来说，易用性是用户通过简单的操作配置就可以达成启动自动化测试任务的目标。
可扩展性	可扩展性保证系统在面对未来修改的时候有良好的持续提升的能力。包括对硬件上的扩展和对系统功能的后续扩展。 对于本系统来说，整个系统服务采用微服务的架构保障快速扩展部署，采用消息队列进行合理的分布式协作。整个系统模块的设计高内聚低耦合，模块相互独立方便扩展。
可维护性	可维护性是系统发生故障后能够排除故障予以修复，并返回到原来正常运行状态的性质。 通过合理的日志系统记录系统运行的整体情况，及时发现系统中存在的问题并且能够快速进行修改。

对于一个成熟的应用软件来说，除了必须保证功能需求能够实现，还需要对系统的非功能需求做到严格把控。非功能需求作为隐藏在功能需求背后的需求，时刻影响着系统的行为和用户使用系统的体验。

对 Web 自动化测试系统的非功能需求评估从可用性、易用性、可扩展性、可

维护性这几个非功能需求进行阐述，如表 3.2所示。这些非功能需求会极大影响用户在使用 Web 自动化测试系统的整体感受，并且能够为系统后续迭代开发打下一定的基础。

3.3.4　Web 自动化测试系统用例

系统用例图

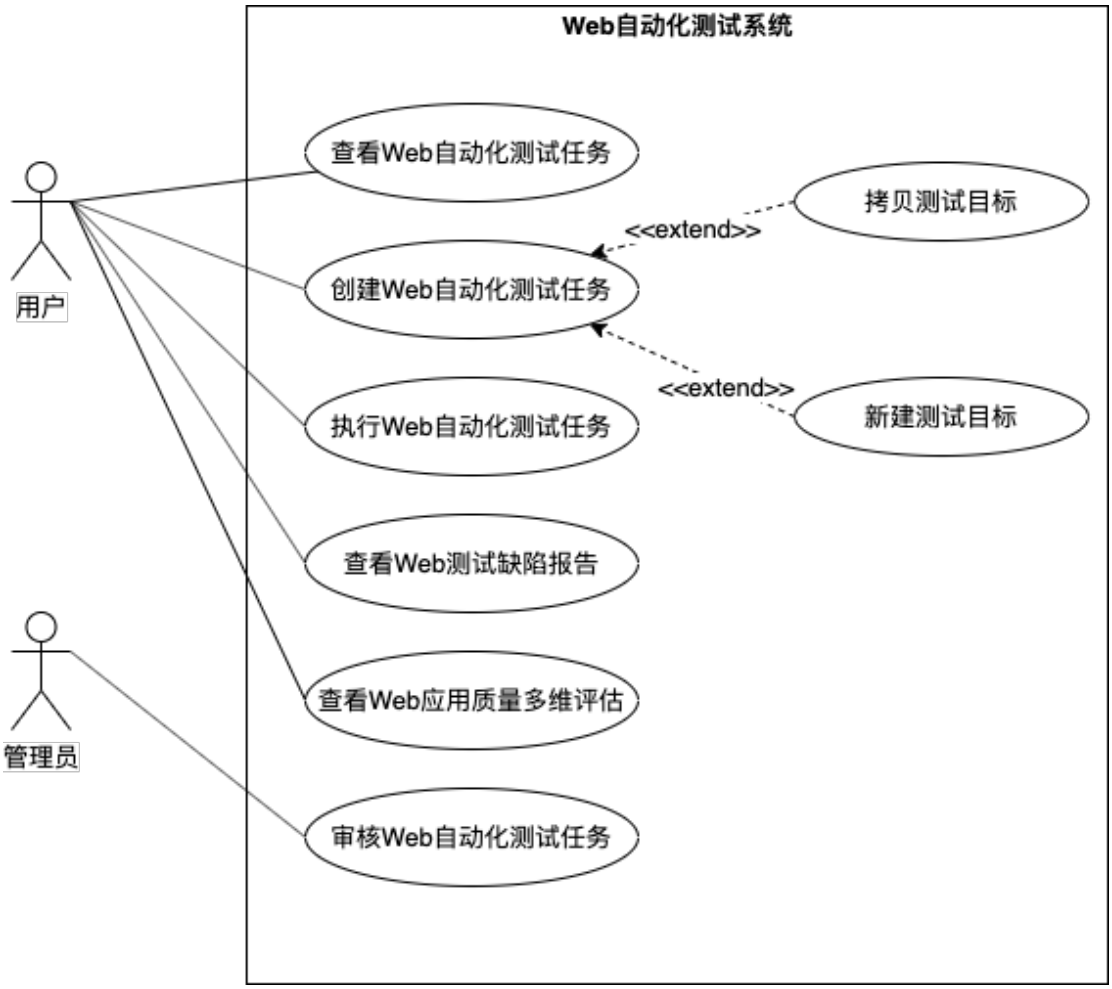


图 3.1: Web 自动化测试系统用例图

根据系统的功能需求所绘制的系统用例图如图 3.1所示。用例图共涉及 6 个用例，分别是查看 Web 自动化测试任务、创建 Web 自动化测试任务、执行 Web 自动化测试任务、查看 Web 测试缺陷报告、查看 Web 应用质量多维评估、审核 Web 测试任务。

其中，将创建 Web 自动化测试任务细化分为两种不同的创建方式，一种是输出一个全新的测试目标（Web 应用 URL），一种是从历史的测试目标中选取，

可以复制该测试目标用于创建。选择两种创建自动化测试任务的方式既方便用户开始新的测试任务，也减少了在做以往应用回归测试的时候需要进行操作。

系统用例描述

表 3.3详细描述了查看 Web 自动化测试任务的用例。用户通过这个用例，在界面上从“Web 自动化测试任务”入口进入测试任务列表，查看当前用户名下创建的所有自动化测试任务。用户可以在 Web 自动化测试任务列表中对自己发起的测试任务进行搜索，包括使用任务名称搜索和使用测试进度搜索。

表 3.3: 查看 Web 自动化测试任务

ID	UC1
名称	查看 Web 自动化测试任务
参与者	主要参与者：系统用户 目的：查看自己建立过的 Web 自动化测试任务
描述	用户在登录系统打开主页后，从侧边栏的“我的任务-Web 自动化测试任务”进入测试任务列表页面，在此页面中查看自己的历史测试任务。
前置条件	用户已经登录系统。
后置条件	用户可以在测试任务列表页面进行测试任务相关的操作。
正常流程	1. 用户登录系统后进入系统首页； 2. 用户点击左侧边栏中“我的任务-Web 自动化测试任务”； 3. 页面跳转到“我的任务/Web 自动化测试任务”页面； 4. 用户查看自己所有的自动化测试任务。
扩展流程	4a. 用户可以通过任务名称搜索 Web 自动化测试任务； 4b. 用户可以通过测试进度搜索 Web 自动化测试任务。

表 3.4详细描述了创建 Web 自动化测试任务的用例。用户通过这个用例，在界面上通过发布任务，新建或从历史创建过的测试网站中选择一个，完成测试任务的创建和发布。测试任务还需要用户进行一些初始化的配置，包括测试执行引擎所需要的测试的总状态数、需要搜索的网页深度、测试执行的最大时间等。用户可以通过自定义这些配置来生成合适自己测试需求的测试任务。另外，用户还可以自定义一些表单的填充，比如登录的用户名和密码，来保证测试路径可以通过权限验证的部分。

表 3.5详细描述了运行 Web 自动化测试任务的用例。用户通过这个用例，可以真正开始 Web 自动化任务的执行。在测试任务执行开始后，用户可以在对应

表 3.4: 创建 Web 自动化测试任务

ID	UC2
名称	创建 Web 自动化测试任务
参与者	主要参与者：系统用户目的：完成创建 Web 自动化测试
描述	用户在登录系统打开主页后，从侧边栏的“发布任务-Web 自动化测试”进入创建界面，在此界面中可以通过新建或者选择历史测试目标来创建自动化测试任务。
前置条件	用户已经登录系统。
后置条件	系统保存自动化测试任务，用户等待管理员审批。
正常流程	1. 用户登录系统后进入系统首页； 2. 用户点击左侧边栏中“发布任务-Web 自动化测试”； 3. 页面跳转到“上传 URL”页面； 4. 用户填写“扫描目标”栏目（Web 应用 URL），并确定上传； 5. 页面跳转到“配置 Web 自动化测试任务”页面； 6. 用户填写“任务名称”等配置参数后，点击“提交测试”提交任务； 7. 页面跳转到“提交完成”页面。
扩展流程	4a. 用户从“历史网址”中选择曾经运行过的测试网址； i. 用户从“历史网址”条目中点击“选取”按钮； ii. 页面跳转到“配置 Web 自动化测试任务”页面； 6a. 用户没有填写“任务名称”便点击“提交测试”按钮提交任务； i. 用户点击“提交测试”按钮； ii. 系统提示“错误-请填写项目名”。

的测试任务栏看到当前任务执行的进度。主要进度状态包括“任务审核”“任务审核不通过”“任务审核通过”“正在排队”“正在执行”“已停止”“执行完成””任务失败”。

表 3.6详细描述了查看 Web 测试缺陷报告的用例。用户通过这个用例，可以找到已经完成执行和报告生成的 Web 自动化测试任务，并且查看生成的测试缺陷报告。用户同时还可以将测试缺陷报告下载到本地，方便在线下环境浏览自己的 Web 测试缺陷报告。测试缺陷报告中包含了复现每个缺陷的测试流程和该测试流程的 Selenium 自动化测试脚本，用户可以通过下载测试脚本运行的方式，轻松复现应用缺陷。

表 3.5: 执行 Web 自动化测试任务

ID	UC3
名称	执行 Web 自动化测试任务
参与者	主要参与者：系统用户目的：真正开始启动 Web 自动化测试任务
描述	用户在登录系统打开主页后，从侧边栏的“我的任务-Web 自动化测试任务”进入测试任务列表页面，在此页面中选择通过审批的未执行的测试任务发起执行。
前置条件	用户已经登录系统，管理员审批通过测试任务。
后置条件	系统开始执行 Web 自动化测试任务，用户等待测试结束。
正常流程	1. 用户登录系统后进入系统首页； 2. 用户点击左侧边栏中“我的任务-Web 自动化测试任务”； 3. 页面跳转到“我的任务/Web 自动化测试任务”页面； 4. 用户点击审核通过的自动化测试任务的“执行”按钮； 5. 系统开始执行测试任务，测试任务处显示执行状态。
扩展流程	无

表 3.6: 查看 Web 测试缺陷报告

ID	UC4
名称	查看 Web 测试缺陷报告
参与者	主要参与者：系统用户 目的：查看执行完成的 Web 自动化测试的测试缺陷报告
描述	用户在登录系统打开主页后，从侧边栏的“我的任务-Web 自动化测试任务”进入测试任务列表页面，在此页面中查看自己的历史测试任务中已经执行完成的测试任务的缺陷报告。
前置条件	用户已经登录系统，用户发起执行的测试任务已经执行完毕。
后置条件	无
正常流程	1. 用户登录系统后进入系统首页； 2. 用户点击左侧边栏中“我的任务-Web 自动化测试任务”； 3. 页面跳转到“我的任务/Web 自动化测试任务”页面； 4. 用户选择已经完成的测试任务，点击“查看报告”； 5. 系统跳转报告页面，包括测试详情、缺陷列表和缺陷详情。
扩展流程	4a. 用户可以点击“下载报告”将报告获取到本地； 5a. 用户可以点击“用例下载”下载测试脚本。

表 3.7详细描述了查看应用质量多维评估的用例。用户通过这个用例，可以直观地对自己所提交的 Web 应用的整体质量做一个详细的了解。在对应用进行整体评定之外，还能够进入每个维度查看对应的维度分析数据，获取更加详细的应用评分信息，方便用户在测试完成后从各个维度提升自己的应用。

表 3.7: 查看 Web 应用质量多维评估

ID	UC5
名称	查看 Web 应用质量多维评估
参与者	主要参与者：系统用户 目的：查看执行完成的 Web 应用质量多维评估报告
描述	用户在登录系统打开主页后，从侧边栏的“我的任务-Web 自动化测试任务”进入测试任务列表页面，在此页面中查看自己的历史测试任务中已经执行完成的测试任务。进入该测试任务的测试报告中查看软件质量多维评估。
前置条件	用户已经登录系统，用户发起执行的测试任务已经执行完毕。
后置条件	无
正常流程	1. 用户登录系统后进入系统首页； 2. 用户点击左侧边栏中“我的任务-Web 自动化测试任务”； 3. 页面跳转到“我的任务/Web 自动化测试任务”页面； 4. 用户选择已经完成的测试任务，点击“查看报告”； 5. 系统跳转到报告页面； 6. 点击“多维评估”标签页进入质量评估评分页面； 7. 点击任意维度标签进入维度详情页面。
扩展流程	无

表 3.8详细描述了审核 Web 自动化测试任务的用例。管理员通过这个用例，可以处理系统一般用户发起的 Web 自动化测试任务申请的审核。管理员需要了解用户发起的自动化测试任务的一些信息，包括项目名称、扫描目标、开始时间等。管理员根据这些信息选择通过或者不通过用户的测试申请。

3.4　Web 自动化测试系统整体架构

根据上述从原始需求中分析出来的用例，我们可以对该系统进行整体结构上的设计，组织各个功能模块并且选择模块间的合适的通讯方式。

图 3.2展现了 Web 自动化测试系统的整体架构。Web 自动化测试系统的主体是提供 Web 自动化测试服务，同时服务需要与慕测企业版平台对接。

表 3.8: 审核 Web 自动化测试任务

ID	UC6
名称	审核 Web 自动化测试任务
参与者	主要参与者：系统用户 目的：审核系统用户提交的 Web 自动化测试任务
描述	管理员在登录系统打开主页后，从侧边栏的“管理任务-任务审核”进入任务审核页面，在此页面中查看所有申请审核任务。
前置条件	用户已经登录系统，用户具有管理员权限。
后置条件	更新自动化测试任务审核状态到数据库。
正常流程	1. 管理员登录系统后进入系统首页； 2. 管理员点击左侧边栏中“管理任务-任务审核”； 3. 页面跳转到“管理任务/任务审核”页面； 4. 管理员查看所有待审核的自动化测试任务； 5. 管理员选择一个自动化测试任务，点击“详情”按钮； 6. 页面跳转到“任务详情”页面； 7. 管理员点击“通过”按钮，提交通过申请。
扩展流程	7a. 管理员点击“拒绝”按钮，提交拒绝申请； 7b. 管理员点击“取消”按钮，退出详情页面。

测试业务模块负责处理与 Web 自动化流程相关的企业平台入口，包括用例中所展现的查看 Web 自动化测试任务、创建 Web 自动化测试任务、执行 Web 自动化测试任务、查看 Web 测试缺陷报告、查看 Web 应用质量多维评估报告。此模块集成在慕测平台中作为与服务交互的一部分。授权业务模块和用户管理业务模块是平台整个任务从发起到获取报告的流程中的一部分，负责支持平台整体业务的流转。自动化测试服务是本 Web 自动化测试系统的核心模块。自动化测试服务模块以黑盒的形式封装成一个完整的服务，接收测试目标，反馈测试缺陷报告和软件质量多维评估报告，其中的测试执行过程细节对用户隐藏，测试执行的整个运行过程无需用户介入。自动化测试服务以微服务的形式构建，以提供 HTTP 服务的形式接受测试触发请求和获取测试报告的请求。

自动化测试服务内部主要以执行服务、报告生成服务和分析服务组成完整的服务功能模块。在执行服务内部以多个测试执行客户端组成执行引擎集群。报告生成服务内部还分为测试报告模块和软件质量评估模块，报告生成服务与分析服务交互，将分析完成的数据进行结构化存储和展示。报告生成服务和执行引擎单元之间的数据流转依赖于数据服务，包括 RabbitMQ 消息队列和 Redis 缓

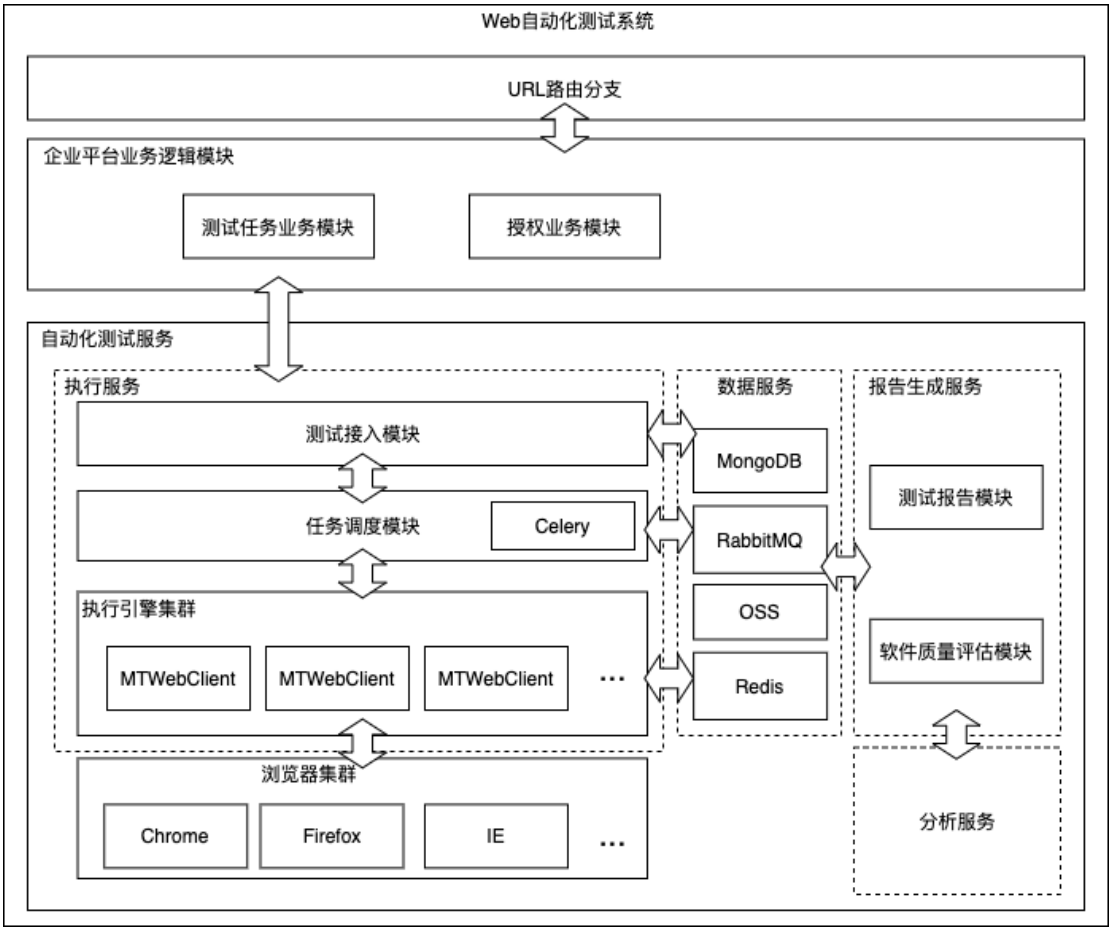


图 3.2: Web 自动化测试系统整体架构图

存。在测试报告生成完毕后，报告数据将会被保存到 MongoDB 中，一些测试过程中的截图会被保存到 OSS 中。

3.5　Web 自动化测试系统执行服务设计

3.5.1　执行服务架构设计

Web 自动化测试系统执行服务是整个自动化测试服务中重要的一环。执行服务采用基于有限状态机和约束的 Web 测试路径生成模型，结合分布式容器技术并行提升探测效率，采用真实浏览器驱动模拟用户操作流程，记录探测过程数据用于最终测试报告生成。在接收到测试任务请求后，执行服务将会启动多个测试执行客户端，每个客户端可能对应不同的浏览器，用于探测整个 Web 应用程序在不同浏览器上的测试路径。在客户端探测的过程中，将 Web 应用的相关信息，包括网络交互情况、控制台输出情况、界面截图等作为原始数据，通过数据服务传递给测试报告生成服务用于报告的分析生成。

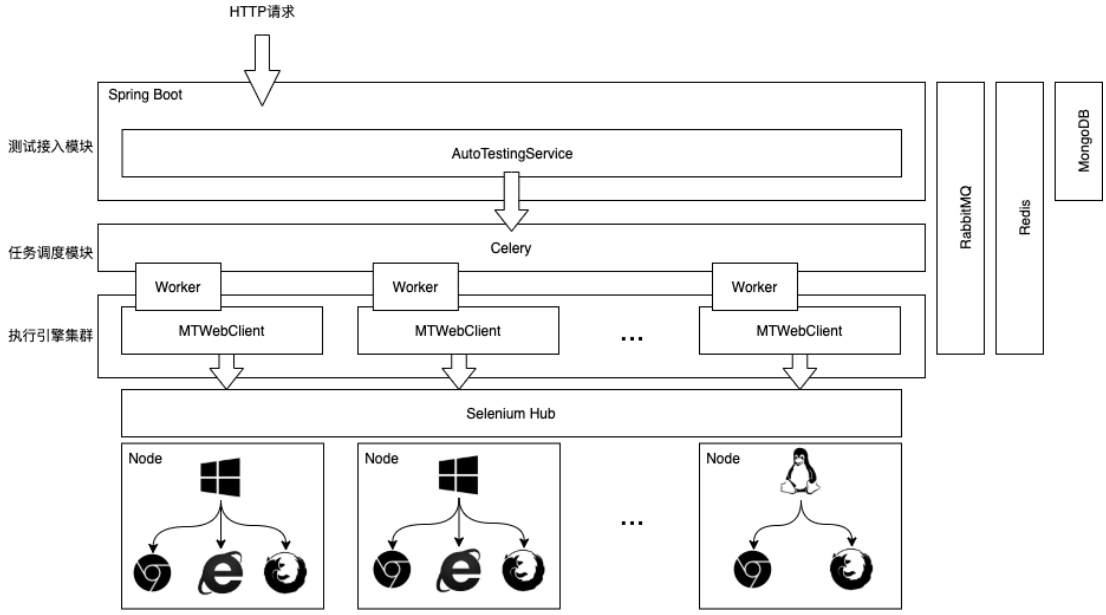


图 3.3: Web 自动化测试系统执行服务架构

图 3.3展现了执行服务的整体架构。根据模块功能内聚的设计原则，将整个执行服务分为三个模块：接入模块、调度模块和执行模块。每个模块分别承担独立的职责，对外呈现为黑盒的形式。

测试接入模块。此模块是整个执行服务的入口，是整个自动化测试流程的起点。模块使用 Spring Boot 框架提供对外 HTTP 服务接口，接收用户测试任务。根据用户的测试需求，将整体测试任务根据不同操作系统、不同的浏览器品牌、不同型号划分为更小的子任务，交由调度模块分配。同时，模块接入 MongoDB 承接测试任务数据的持久化存储。由于整个自动化测试任务执行的时间较长，测试任务结果被设计为通过异步的方式返回给用户。根据微服务的设计思想，子任务信息将以 HTTP 请求的形式发送给任务调度模块。

任务调度模块。此模块以 Celery 作为任务调度的基础框架。模块接收测试接入模块的测试子任务，并且将测试子任务分派到不同的 Worker 下。Worker 承担任务的执行、监控和结果回调的职责。模块使用 Flask 搭建 HTTP 服务接口。在接收到测试接入模块发来的测试子任务请求后，子任务内容被转为 Celery 任务配置。任务调度采用 RabbitMQ 作为中间消息服务。Worker 并发运行在分布式的系统节点中，通过监听 RabbitMQ 消息队列内容获取测试子任务配置，通过命令行启动宿主机上执行引擎单元的 Docker 容器，并监听容器的生命周期和返回值，用于反馈子任务执行状态。调度模块依赖于消息队列进行整体的任务分配和调度，因此有对分布式的天然支持。在任意物理机上，只要有 Docker 和 Celery

的支持，就具备了启动执行引擎单元的运行环境。将任务调度这个过程封装为一个独立的模块，整个调度过程对于测试接入模块和测试执行引擎单元而言都是黑盒的，方便未来测试集群的横向扩展。

执行引擎单元。此模块将会作为一个独立客户端被构建为一个 Docker 镜像，可以在任意安装有 Docker 服务的宿主机上执行。同一个任务下会存在多个执行引擎单元的 Docker 容器，并行探测 Web 应用以便提升测试效率。容器之间通过 R 通过 Redis 提供的 SETNX 方法，在运行过程中同步同一任务下测试客户端的数量、测试任务开始和结束信号。除了数据同步，Redis 还承担了与测试报告服务交换测试执行的过程数据的职责。执行引擎单元与测试报告服务交换的数据将会被作为 Web 应用功能、性能、稳定性、健壮性评估的原始数据。执行引擎单元的应用探测过程利用 Selenium Grid 构建的浏览器集群。Selenium 采用 WebDriver 控制真实浏览器的方式运行待测应用，因此保证了对使用各种技术构建的 Web 应用进行测试的普适性。

3.5.2　执行服务逻辑视图设计

逻辑视图用于支撑系统的功能需求，整个系统将会被分为一系列功能的抽象信息，也作为标识整个系统的各个不同部分的通用机制和设计元素。在这里表现为类和类提供的服务内容。由于执行服务的三个主要模块分属于不同的子系统中，采用外部通讯方式进行数据交互。因此，对模块逻辑视图设计将根据模块分别阐述。

图 3.4展示了接入模块的逻辑视图，也是的主要类的设计。接入模块以 Spring Boot 为框架构建 Web 服务，其大部分类的实例的生命周期由 Spring 管理。

AutoTestingController 类是整个自动化测试服务的入口。该类提供 HTTP 接口以供调用。作为 Spring 项目的控制器类，该类添加了 @RestController 注解用于将请求和返回值格式化以符合 JSON 格式的接口调用。通过添加 @RequestMapping 注解指明接口路由。

AutoTestingTaskService 类负责提供自动化测试任务转换服务。从 Controller 中传入的测试任务请求在这里被分解为多个符合执行引擎单元运行要求的子任务，并派发给 Celery。

MTWebTaskService 类和 MTWebTaskRepository 接口主要是负责测试任务的相关业务逻辑，包括增删改查操作。MTWebTaskRepository 接口作为数据层接口与 MongoDB 交互，其具体实现逻辑由 spring-data-mongodb 实现。

MTWebTask 类和 CeleryStartResponse 类是对业务数据模型的封装。MTWebTask 类保存了一个测试任务的相关信息；CeleryStartResponse 是映射了调用 Cel-

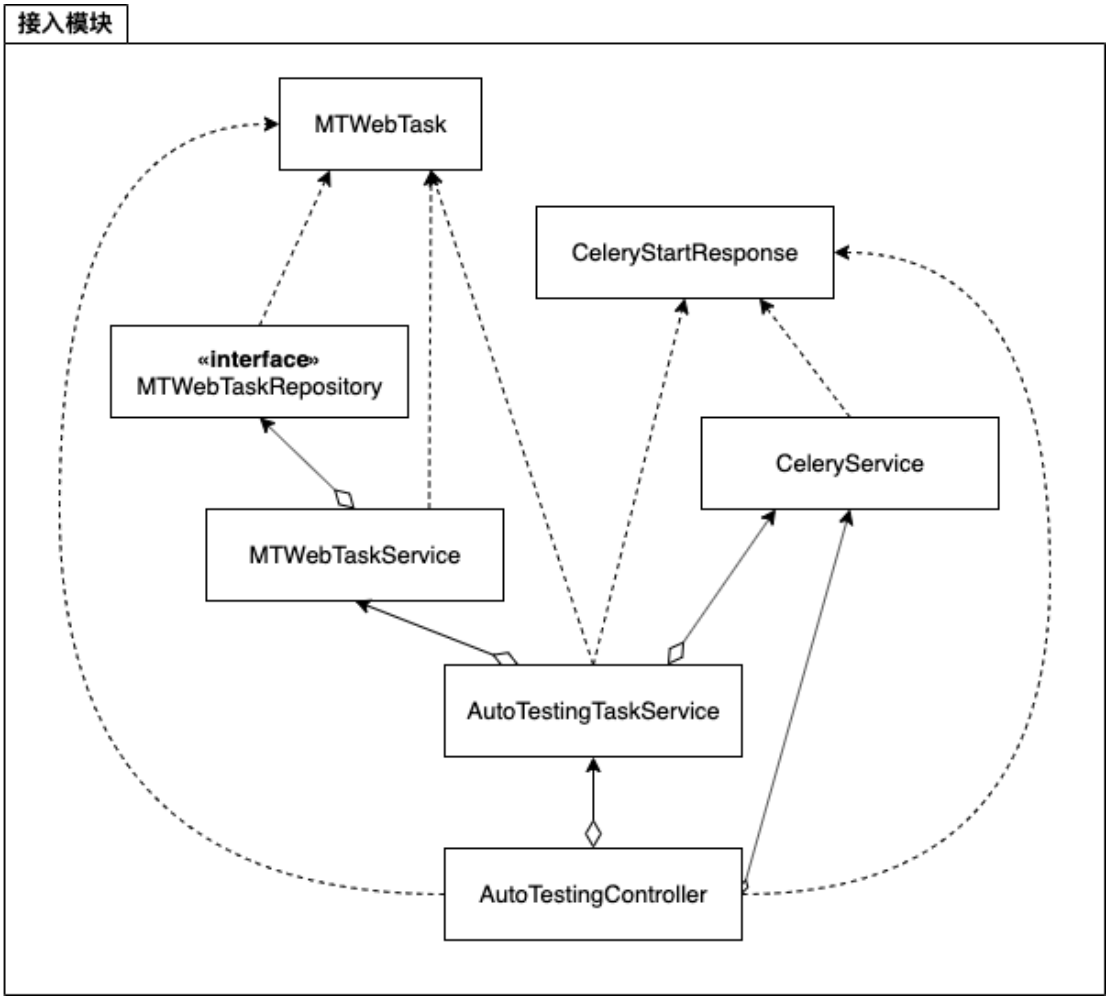


图 3.4: 接入模块逻辑视图

ery 启动接口的返回值。

调度模块依赖于开源工具 Celery 实现，在第二章中已经对 Celery 任务调度工具开源技术和在本系统中承担的作用进行了较为详细介绍。

图 3.5展示了执行模块中的一个执行引擎单元的逻辑视图，也是执行模块的系统类图。执行引擎单元会作为一个普通的 Java 程序，从命令行获取启动参数并进行测试路径生成方法的初始化和执行。

RunnerCommand 类作为解析命令行参数，装配执行所用到的类，以及启动整个测试路径搜索流程。整个客户端是使用 Google Guice[2]进行依赖注入，将根据不同的配置使用不同的实现类这个复杂的判断过程分散到各个装配类中，降低整个系统的耦合程度。

[2]https://github.com/google/guice

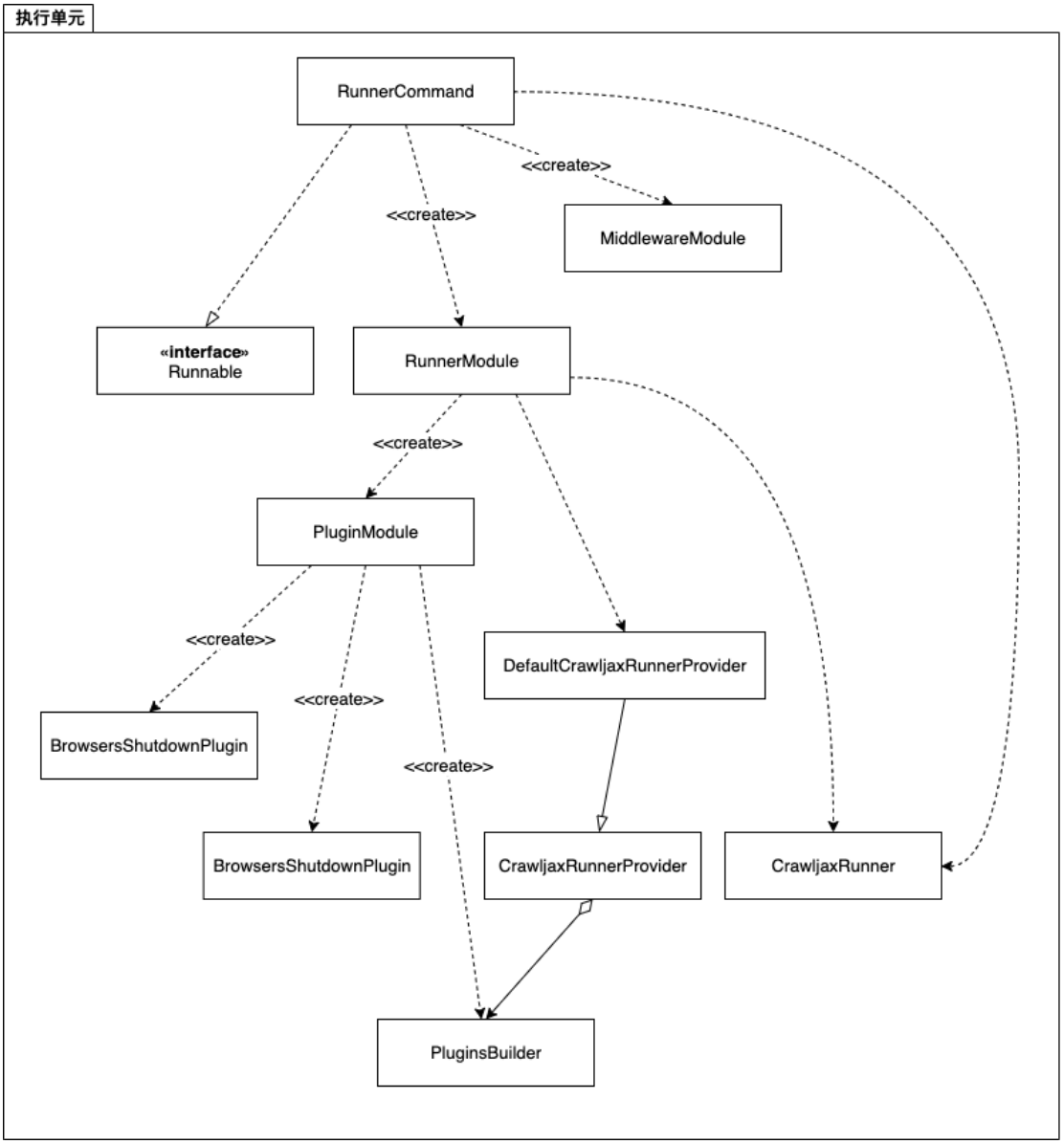


图 3.5: 执行引擎单元逻辑视图

RunnerModule、PluginModule 和 MiddlwareModule 是三个配置模块，负责根据不同配置指定抽象接口的对应实现。RunnerModule 是总装配模块，提供构建 CrawljaxRunner 类实例的工厂类 CrawljaxRunnerProvider。RunnerModule 类中同时还嵌套了 PluginModule 类的初始化和配置，用于构建执行过程中某些检查点所需要执行的监听方法。

MiddlewareModule 类负责装配与服务中间层交互的客户端，比如与 Redis 和 RabbitMQ 交互的类的实例。这些客户端类的实例在系统中作为单例的形式存在，降低线程资源消耗，避免出现资源竞争的情况。并且由于客户端类本身不存

在内部状态，因此在多线程调用的过程中是线程安全的。

SignalPlugin 类和 BrowsersShutdownPlugin 类实现了测试路径搜索过程中某些状态点的监听。这两个类将会运行在整个测试路径执行结束后。SignalPlugin 类会发送结束信号；BrowsersShutdownPlugin 类会释放所用到的所有浏览器，将其归还到浏览器资源池中，供新的任务获取使用。

CrawljaxRunner 是真正的执行器。在配置完成后，通过 CrawljaxRunner 类的 call 方法可以正式启动测试路径的搜索。

3.5.3　执行服务进程视图

进程视图是关注于整个系统内部的交互和运行特性，涉及系统的非功能需求。图 3.6展示了接入模块在整个自动化测试业务逻辑过程中类之间以及类与外部组件交互的时序图。在整个交互过程中，自动化测试执行过程是一个典型的长耗时的异步任务。为了便于用户查询测试任务的执行情况，我们通过一个字段来标明自动化测试任务的执行状态。

用户发起自动化测试请求后，由 AutoTestingController 类负责接收请求。AutoTestingController 类将请求参数解析为 AutoTestingConfigurationModel 类后交由 AutoTestingService 类进行配置解析。通过不同浏览器的配置将测试任务分为多个测试子任务，为提交测试调度模块做最后的准备。

AutoTestingService 类将子任务构建为 Celery 任务后，通过 CeleryService 类提交给 Celery 执行。与 Celery 接口的交互是同步返回的。Celery 接口将会告知 Celery 异步任务的执行 ID，此时 AutoTestingService 会记录该任务 ID 方便后期用户查询任务状态和任务结果。测试任务也会在 MongoDB 中存储为一份多层嵌套格式的文档。

用户会在发起测试的接口获取同步返回值，其中包括了测试任务的 ID。用户根据测试任务 ID 轮询测试任务结果接口。在测试报告尚未生成之前，测试任务的状态都会处于 RUNNING 状态下。测试的整个运行流程和测试报告生成完毕后，会将生成的测试报告存入 MongoDB 对应的测试任务中，并修改测试任务的状态为 FINISHED。用户可以获取到测试状态为 FINISHED 的测试任务报告。如果测试任务产生异常失败终止，用户将会获取到 FAILURE 任务状态告知。

图 3.7展示了执行模块的一个执行引擎单元在整个测试执行过程中类之间和类与外部组件交互的时序图。整个自动化执行过程被设计为一个多线程并发任务，充分利用硬件资源，将单线程中浏览器加载网页的等待时间最大限度利用起来，加快执行速度，缩短执行时间。

执行请求将会从 Celery 的 Worker 下发，通过命令行启动 MTWebClient 客户

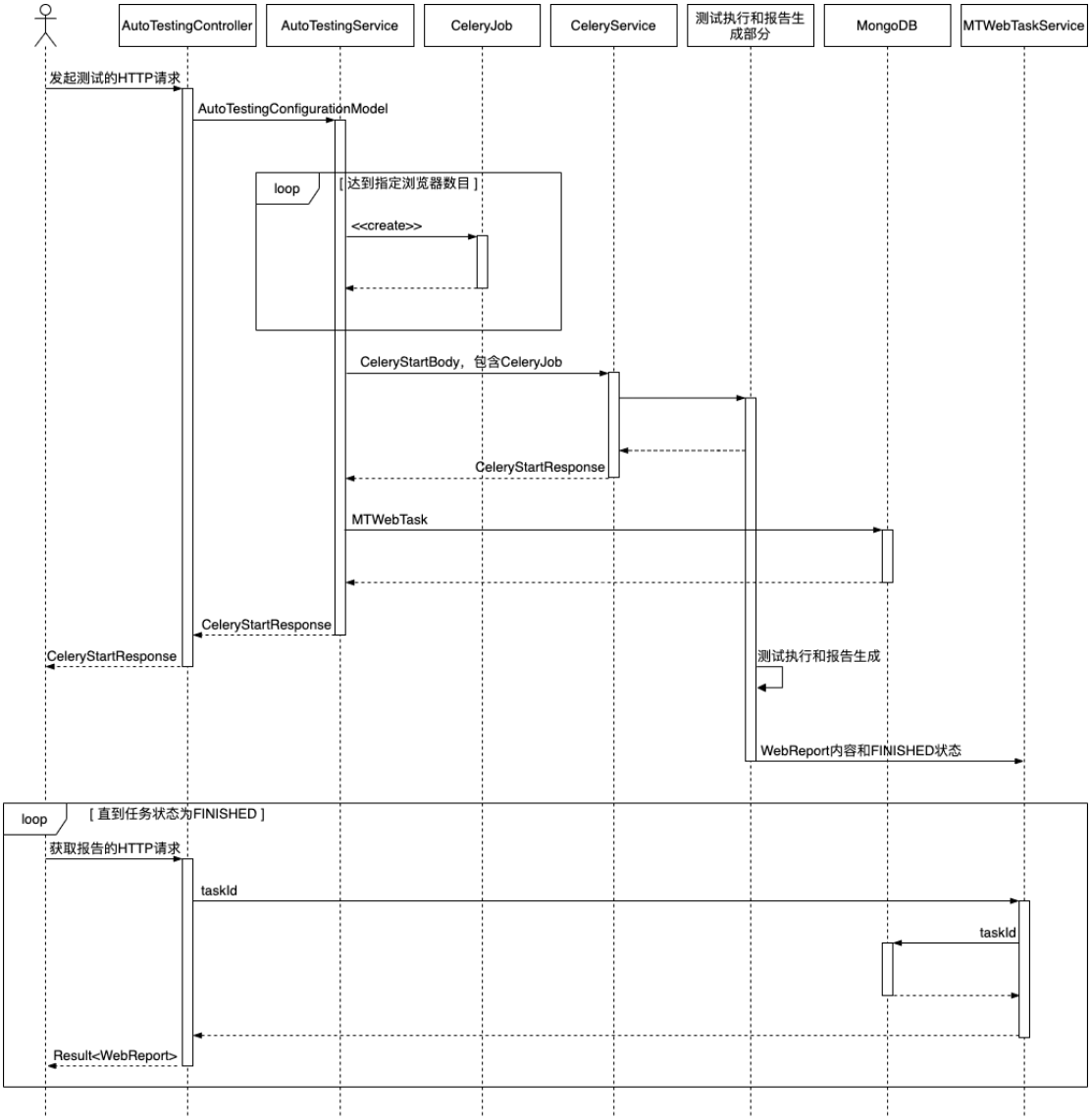


图 3.6: 接入模块时序图

端的 Docker 镜像，构建容器。容器内部是一个可执行的 jar 包，使用 *java -jar* 命令启动 Java 程序。

RunnerCommand 作为接收和解析命令行参数的类，构建模块启动配置，交由 Guice 的装配器获取中间件组件和执行器 CrawljaxRunner 的对象实例。

在实例化了 CrawljaxRunner 后，就可以启动测试执行了。CrawljaxRunner 会构建一个对整个执行过程起到调度作用的类 CrawlController。在执行正式开始之前，CrawlController 类会通过 RabbitMQ 发送任务开始信号，同时会在 Redis 内部记录生效的执行引擎单元个数。

通过 executeConsumers，CrawlController 启动了多个线程并发处理测试任务

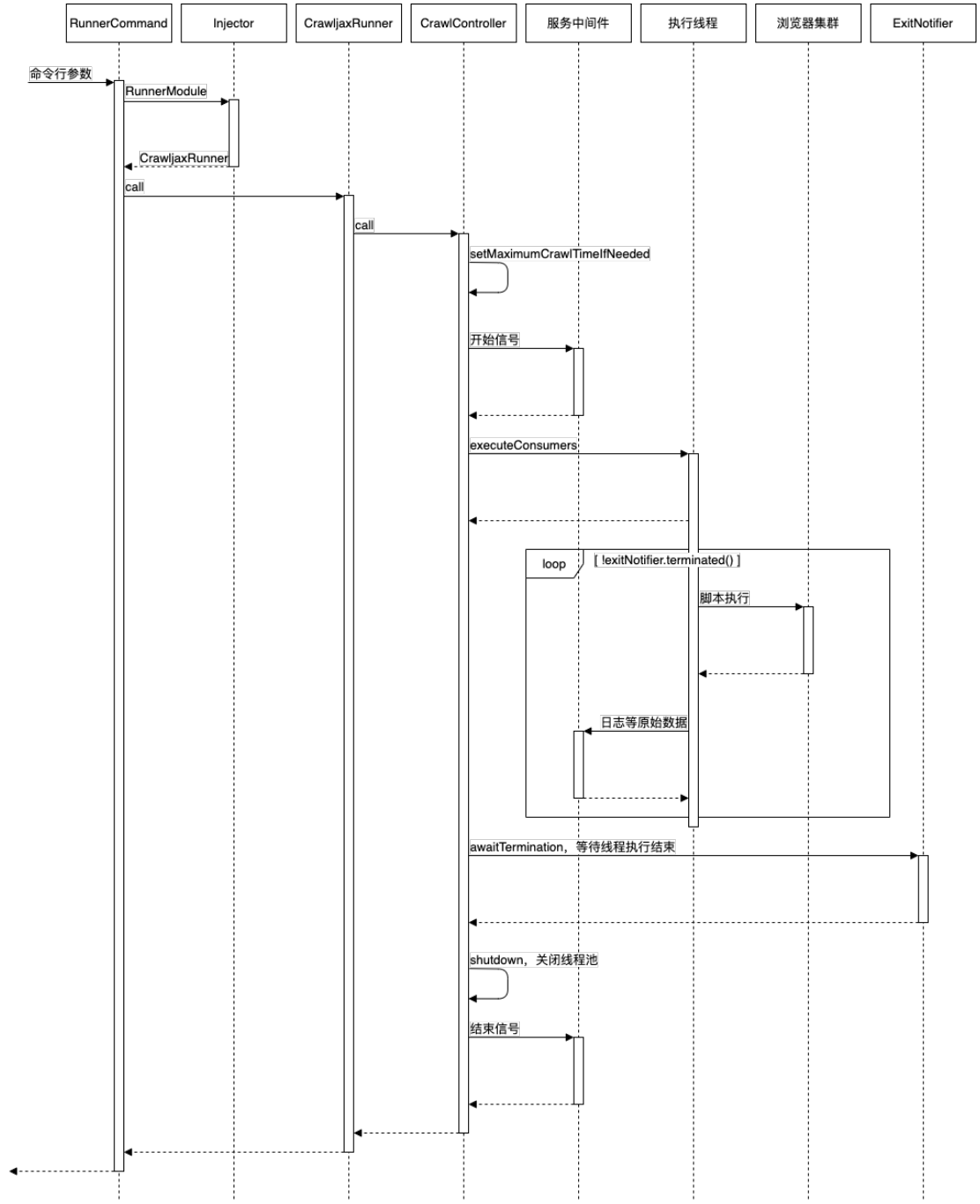


图 3.7: 执行引擎单元时序图

的执行。线程会与远程的浏览器集群交互，每个线程都会启动一个独立的浏览器进程作为自动化操作执行环境。线程每获取到一个新的状态，将会触发数据提交的事件，将从浏览器获取到的日志等原始数据存入 Redis，并且通过 RabbitMQ 通知分析程序处理。

在到达设定的结束点，比如搜索状态达到设定上限、执行时间达到设定上限、没有可用于搜索的状态时，CrawlController 类将会被唤醒并执行 shutdown 操作，关闭线程池、释放浏览器资源等。最后，会通过数据服务发送执行引擎单元的结束信号。

3.5.4　执行服务开发视图

开发视图为整个项目的开发人员提供切实的指导。图 3.8展示了执行服务的开发视图。整个 Web 自动化测试系统使用 Maven[3]工具进行管理。Web 自动化测试系统是一个父项目。执行服务的开发所涉及到的主要有三个子项目：mt-web-client、web-auto-testing-analysis 和 web-auto-testing-common。

mt-web-client 子项目覆盖了执行引擎单元部分。web-auto-testing-analysis 子项目是一个 Spring Boot 项目，涉及测试接入模块。由于两个 Java 子项目之间存在着多个需要交互的实体数据类和一些工具类，因此 web-auto-testing-common 子项目负责作为一个通用数据实体类库和通用工具类库，被添加到另外两个子项目的依赖中去。这样避免了在 mt-web-client 子项目和 web-auto-testing-analysis 子项目中分别实现交互的实体类，也避免了实体类分别实现可能带来的修改不同步等严重问题隐患。

mt-web-client 子项目中的包是面向解析控制台命令和实际的测试执行开发。包 command 下的类负责解析命令行参数，并且启动测试。包 core 下面的内容是构建执行内容所需要的组件。包 di 下面的类与依赖注入相关，涉及测试执行的具体实现类。包 util 则是一些通用工具。

web-auto-testing-analysis 子项目是标准的 Spring Boot 项目，符合 Spring Boot 项目结构。包 controller 是 HTTP 请求的入口，而主要的测试任务业务操作和测试任务调度服务都放在包 service 中。包 model 是该子项目用到的数据实体类，包 repository 参与数据库交互。

Web 自动化测试系统采用 Maven 工具管理，能够很好组织各个项目所需要的第三方依赖，也可以明确展示各个子项目内部的依赖关系，保证项目结构对于开发者来说易于理解。Maven 项目还能够提供良好的 CI/CD(持续集成/持续交付) 特性，并且通过在 POM 文件中添加 Docker 相关的镜像构建插件，方便开发者快速打包部署。

[3]https://maven.apache.org/

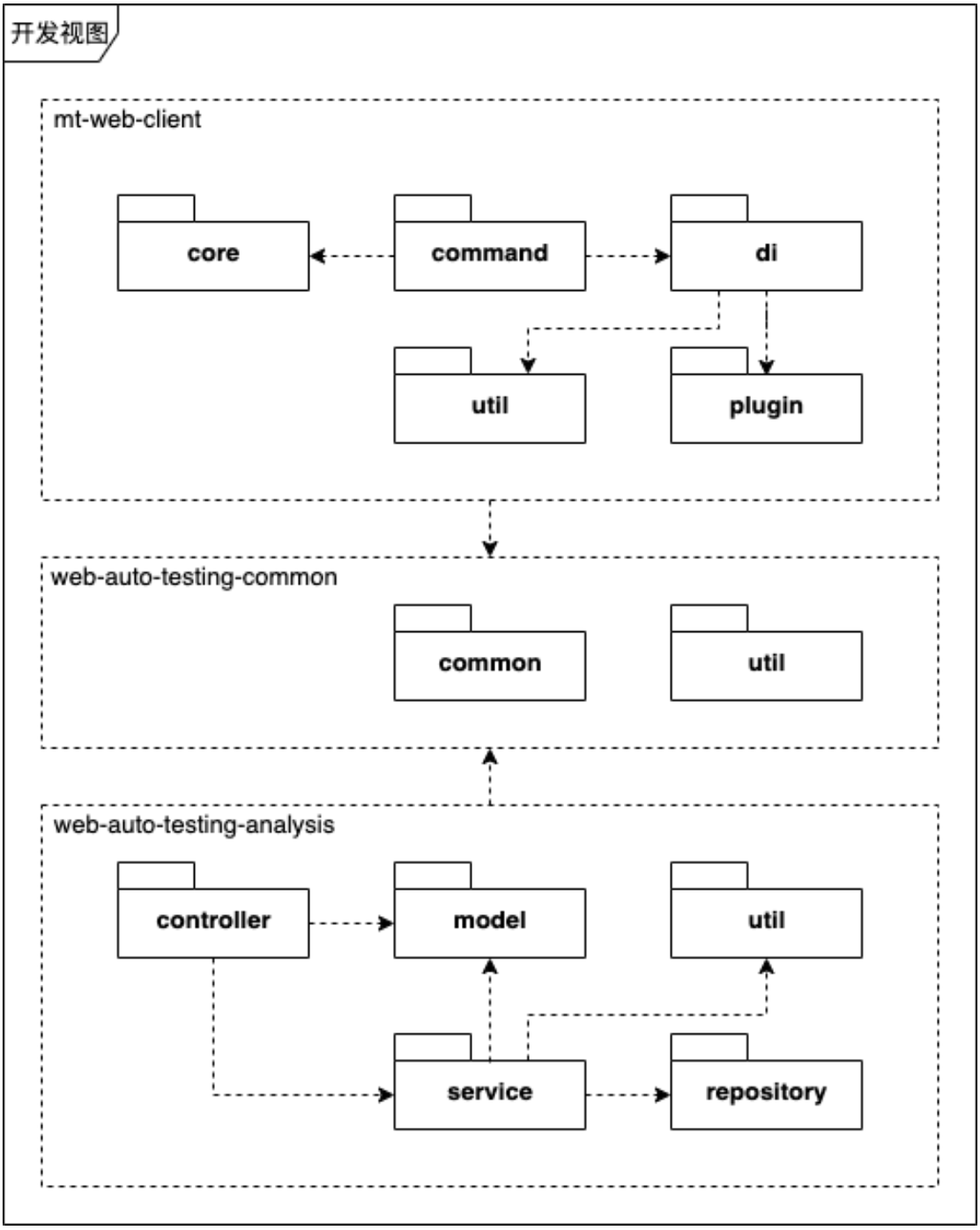


图 3.8: 执行服务开发视图

3.5.5　执行服务物理视图

物理视图通过运维人员的视角，考虑的是如何将软件映射到硬件系统中。物理视图的设计需要考虑整个系统的性能、规模、可靠性等，为系统通信、安装和

拓扑结构提供解决方案。

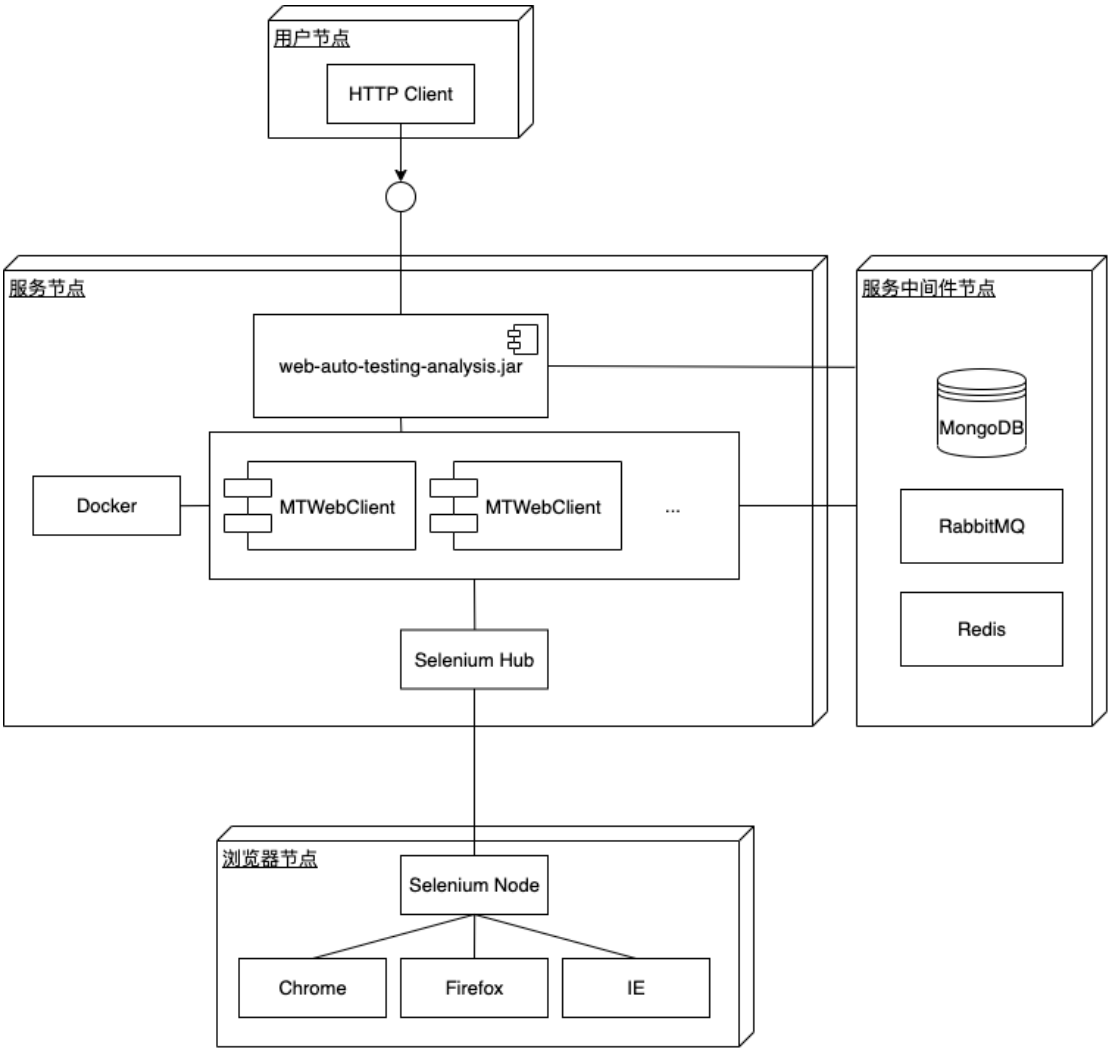


图 3.9: 执行服务物理视图

执行服务的物理视图情况如图 3.9所示。用户节点可以是任何形式的 HTTP 客户端，只要其能够发送符合接口定义的 HTTP 请求，就可以通过 API 调用自动化执行服务。服务节点所在的服务器宿主机上包含了多个系统所需的服务组件：Celery，Docker，以及使用 Spring Boot 打包而成的 Web 服务的 jar。

作为整个自动化测试服务的一部分，执行服务的接收模块被收纳进了 web-auto-testing-analysis.jar 中，该 jar 中还包含了报告生成服务相关的内容。测试服务的 jar 包采用 Maven 构建，将 Spring Boot 框架和执行服务项目所依赖的一系列第三方库、第三方工具和配置文件全部打入一个 FatJar 中，减少了服务环境依赖配置过程。服务的 jar 包内部自带 Tomcat 容器，直接可以运行 HTTP 服务。项目采用 Maven 进行管理还方便接入企业自动化集成的流程中去。

服务的顺利执行还依赖于 Celery 服务和 Docker 服务，这些服务在当前是与测试服务 web-auto-testin-analysis.jar 一并部署在同一台宿主机上的。由于 Celery 服务和 Docker 服务都有很强的可移植性，因此对于以后添加宿主机进行扩展的需求，能够很方便地进行应对：只要在新的服务器上安装完成 Celery 服务环境和 Docker 服务环境，拉取客户端运行镜像，并且可以通过局域网或者广域网 IP 地址访问到服务所使用的 RabbitMQ 消息队列，就可以横向扩展测试客户端的数量，提升整体的性能。Docker 服务会在该宿主机上启动 MTWebClient 的镜像并生成多个容器，用于执行测试。容器的运行与宿主机本地操作系统和 Java 环境无关，仅需要依赖于 Docker 服务。

数据存储依赖于 MongoDB 服务，进程间通讯依赖于 RabbitMQ，这两个服务可以单独部署，通过 IP 进行网络连接访问。

Selenium Hub 进程作为浏览器集群的入口，在目前的环境中也被部署在了同一台宿主机上，方便与 MTWebClient 的 RemoteDriver 通讯。如果有后续的性能扩展的需求，可以将 Selenium Hub 独立部署在一台服务器上，专门用于提供 Remote Driver 的连接和执行接口。浏览器节点可以在其他服务器上进行部署，这里选择了另一台服务器，操作系统为 Windows Server 作为宿主机。使用 Windows Server 的目的是使用 Windows 环境进行 IE 浏览器的兼容性测试。

3.5.6　执行服务场景视图

场景视图是从系统用户视角出发，描述用户的使用场景，在本文中对应 3.3 节中系统的用例分析和用例图。

3.5.7　执行服务服务数据库设计

服务使用 MongoDB 作为最终的持久化存储，使用 spring-data-mongodb 作为将 Java 对象存储为 MongoDB 的 Document 的持久层 ORM 库。

与执行服务相关的主要文档对象仅一例，其结构设计如图 3.10所示。其中 *id*、*celerySessionId*、*url*、*user*、*status*、*createdAt*、*updatedAt* 为普通的数据字段，*partitionIds*、*config*、*report*、*scoreModel* 为嵌套文档对象。嵌套文档能够将该字段所需要的所有数据保存在字段下面，而不是像 MySQL[4]之类的关系型数据库，需要通过外键的形式另起一张数据表保存相关数据，减少了查询时的关联操作和保存时的多表修改操作。该文档映射到的 Java 类为 MTWebTask 类。

[4]https://www.mysql.com/

```
{
    "_id": String, //  文档主键
    "celerySessionId": String, // Celery 任务主键
    "partitionsIds": [], // Celery 子任务主键列表
    "url": String, //  目标网站 URL
    "config": AutoTestingConfigurationModel, //  任务启动配置
    "user": String, //  发起用户
    "status": CeleryStatus, //  任务状态
    "report": WebReport, //  任务结果报告
    "scoreModel": ScoreModel, //  任务评分
    "createdAt": Date, //  开始时间
    "updatedAt": Date //  更新时间
}
```

图 3.10: MTWebTask 文档设计

3.5.8　本章小结

本章首先对 Web 自动化测试系统整体进行了规划和描述，分析系统的相关功能需求和非功能需求。接着，通过架构图展现 Web 自动化测试系统的高层设计，着重且细致地描述了 Web 自动化测试系统中的执行服务部分的架构设计。随后，以 4+1 视图形式详细展示了 Web 自动化系统执行服务从需求分析到开发部署的整个过程中经历的设计过程。使用逻辑视图类结构设计展示了执行服务的两个模块的概要内容；使用进程视图时序图展示了在整个执行过程中各个模块进程内部和进程间的主要交互；使用开发视图展示了开发过程中整个项目的实际结构；使用物理视图展示了整个服务在物理机上的部署情况；场景视图则是用例图的再现。本章还展示了执行服务持久化数据结构。

本章所阐述的对服务的高层的架构和流程设计也为下一章中服务模块的详细设计和实现做出良好的铺垫。

第四章　详细设计与实现

从第三章给出的 Web 自动化测试系统执行服务的架构和流程设计，本章将详细讲述服务的几个模块的具体技术使用和代码实现，在最后也会给出整个系统的实现界面。

4.1　模块综述

第三章已经简单介绍了执行服务根据功能内聚的设计原则分为三个模块：测试接入模块、任务调度模块和执行引擎单元。测试接入模块负责接收测试任务、划分测试任务、提供测试结果，详细设计主分为的以下两个子功能：测试任务的管理、启动测试执行。任务调度模块负责将测试任务分配到不同物理机上。本章将详述如何通过消息队列对任务进行分配。执行引擎单元负责最终对 Web 应用进行状态空间的探测以及测试原始数据的收集。本章将详述测试执行的复杂参数配置、基于有限状态机和约束的 Web 状态空间探测模型、结合分布式容器技术并行提升探测效率的测试执行方法。

4.2　测试接入模块

测试接入模块如前文所述，是接收用户侧发送的规定格式的 HTTP 请求，并根据请求内容执行测试任务启动、获取测试任务状态等于测试任务相关的操作。测试接入模块是与用户交互的入口，也是整个自动化测试系统的入口。

4.2.1　测试启动参数

测试启动参数是开始测试执行的整个流程所需要用户给出的配置项，启动参数定义了一系列测试执行的约束，为的是避免服务还需要用户在测试执行过程中进行干预，减轻用户的使用负担。

图 4.1是测试启动参数类的具体实现。其中，参数内容主要分为 4 个类别：执行和结束条件、浏览器配置、目标元素、登录验证流程。参数 url 是待测 Web 应用的资源地址，接下来的两个参数 maximumStates、maximumRuntime 规定了测试执行的结束条件，在到达最大状态、最大运行时间后，测试将会自动停止，避免程序无限搜索下去的情况。参数 maximumDepth 则限制了搜索的深度，减少状态机中的环路出现。参数 browsers 是用户定义的需要使用的浏览器环境，通过比较不同浏览器的运行结果，可以对 Web 应用的兼容性情况进行分析。在默

```
public class AutoTestingConfigurationModel {

    private String url;
    private int maximumStates = 0;
    private long maximumRuntime = 0;
    private int maximumDepth = 2;
    private Set<BrowserConfigurationCommand> browsers;
    private CrawlRulesModel crawlRulesModel;
    private AuthorizingActionConfigurationModel authConfigModel;
    private String user;

}
```

图 4.1: AutoTestingConfigurationModel 类具体实现

认的配置下，执行将以 <a> 标签、<button> 标签和 <input> 标签为主要的事件操作对象。由于 Web 应用可以通过 JavaScript 对任意标签添加事件，所以在启动参数中，使用 crawlRulesModel 配置允许执行器操作的 DOM 元素。有些网站需要登录后才能进入更多的页面、进行更多的操作，authConfigModel 提供了在测试执行开始前的登录操作流程。

通过 Spring 提供的 HttpMessageConverters，本文实现了 AutoTestingConfigurationModel 类与 HTTP 的 POST 请求中的请求体 JSON 格式的字符串之间的相互转化，使用 @RequestBody 注解直接映射 JSON 与 Java 实体类。

4.2.2　测试任务启动

测试任务启动功能是用户使用系统的重要功能，用户只有通过这个接口才能启动测试。测试启动接口使用 Spring 框架的 spring-web 组件提供 HTTP 接口。整个测试启动的活动图如 4.2所示。

用户通过 HTTP 请求是整个活动的开始。用户发起测试任务后，系统内部会根据用户提交的测试请求中需要执行的浏览器列表生成 Celery 子任务。构建完成 Celery 子任务后，封装成一个完整的 Celery 任务通过 HTTP 接口发送给 Celery 调度执行。由于整个测试任务的执行时间较长，Celery 接口同步返回的不是执行完成的数据，而是 Celery 执行的任务 ID。任务 ID 等任务相关信息将会被保存进 MongoDB 数据库，同时返回任务 ID 方便用户通过轮询的方式检查任务执行状态。

HTTP 服务通过 Spring 框架构建，Spring 提供了通过注解的方式构建类实例的方法。在测试业务入口 AutoTestingController 中添加了 @RestController 和 @RequestMapping(”/api/v1”) 的注解，将该类标记为一个用于真实处理路由前缀

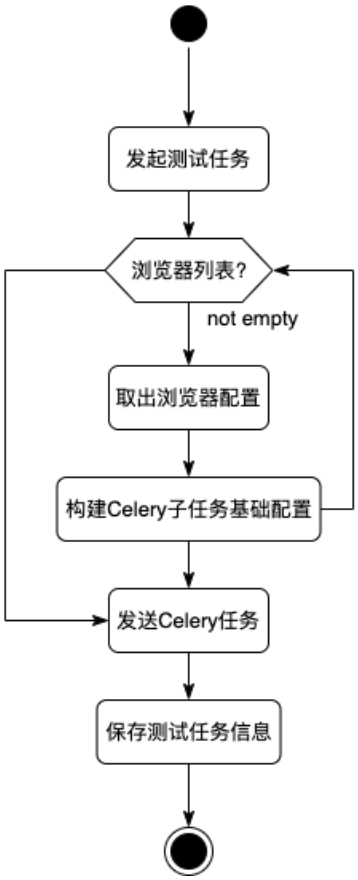


图 4.2: 测试启动活动图

为”/api/v1” 的 HTTP 请求的控制器类。在类方法 startTask 上通过 @PostMapping(”task/start”) 注解，Spring 会将接收到的 HTTP 请求中路由为”task/start” 的 POST 请求转交由该方法处理。方法的参数 configModel 添加了 @RequestBody 注解，对于 POST 方法来说，HTTP 请求中以 JSON 格式传输的 body 将会被映射成 AutoTestingConfigurationModel 类。方法返回值则会被 Spring 转换为 JSON 格式的字符串作为 HTTP Response 的返回体。

AutoTestingController 负责处理 “HTTP 请求-Java 实体类” 的转换，测试任务启动的业务逻辑会被交由 AutoTestingService 处理。在 AutoTestingService 类中，入参 AutoTestingConfigurationModel 类将会根据用户配置的浏览器型号种类分解为多个 MTWebClientConfigurationModel 类，作为客户端的测试执行的配置。接着，将测试执行配置，测试执行所需要的环境的配置（包括 Redis、RabbitMQ、Selenium Hub、OSS 的地址和权限）综合起来，构建成一个 CeleryJob 类。通过 Java 8 的 Lambda 表达式新特性，可以很简单的将 MTWebClientConfigurationModel 类 map 为 CeleryJob 类。使用 Lambda 表达不仅可以实现实用、简洁和易读的编码，

此外，新的 Lambda 表达式也可以提供性能上面的优势 [40]。

所有的 CeleryJob 实例会被汇总成一个 CeleryStartBody 类，通过 CeleryService 启动执行。在启动的这个时间点上，我们不关心整个测试执行的结果，等启动这个异步任务返回任务 ID 后，将配置信息和任务信息存入 MongoDB，任务状态设置为 STARTED。整个 AutoTestingService 的具体实现如图 4.3所示。

```
public CeleryStartResponse startTask(AutoTestingConfigurationModel configModel)
{
    Preconditions.checkNotNull(configModel.getBrowsers());
    Preconditions.checkState(!configModel.getBrowsers().isEmpty());
    CeleryStartBody body = new CeleryStartBody();
    // ...
    String taskId = UUID.randomUUID().toString();
    List<MTWebClientConfigurationModel> clientConfigModels = new LinkedList<>();
    for (BrowserConfigurationCommand browserConfigModel
            : configModel.getBrowsers()) {
        MTWebClientConfigurationModel clientConfigModel
             = new MTWebClientConfigurationModel();
        // ...
        clientConfigModels.add(clientConfigModel);
    }
    List<CeleryJob> celeryJobs = clientConfigModels
             .stream()
             .map((clientConfig) -> {
                CeleryJob job = new CeleryJob();
                // ...
                return job;
             })
             .collect(Collectors.toList());
    body.setJobs(celeryJobs);
    Preconditions.checkNotNull(body.getToolName());
    Preconditions.checkNotNull(body.getJobs());
    CeleryStartResponse celeryStartResponse = celeryService.start(body);
    MTWebTask mtWebTask = new MTWebTask();
    // ...
    this.mtWebTaskService.save(mtWebTask);
    return celeryStartResponse;
}
```

图 4.3: 测试启动模块代码

CeleryService 提供了通过 HTTP API 通知 Celery 任务启动的接口。我们使用 Feign 作为 HTTP 的客户端，与 Spring 进行集成。相比较于通过 HttpURLConnection 等原生 Java HTTP 客户端来说，使用 Feign 能够更好地与 Spring 框架融合并且调用更加简单。由于在系统中对服务依赖的调用可能不止一处，往往一个 HTTP 接口会被多处调用，一般都会针对每个 HTTP 服务自行封装一些客户端类

来包装这些依赖服务的调用。Feign 在此基础上做了进一步封装，我们只需要创建一个接口并使用注解 @FeignClient 的方式来配置它，即可完成对服务提供方的接口绑定。Spring 服务会在构建的时候向 Bean 中注入接口的实现类。Feign 的调用时序图如 4.4所示。

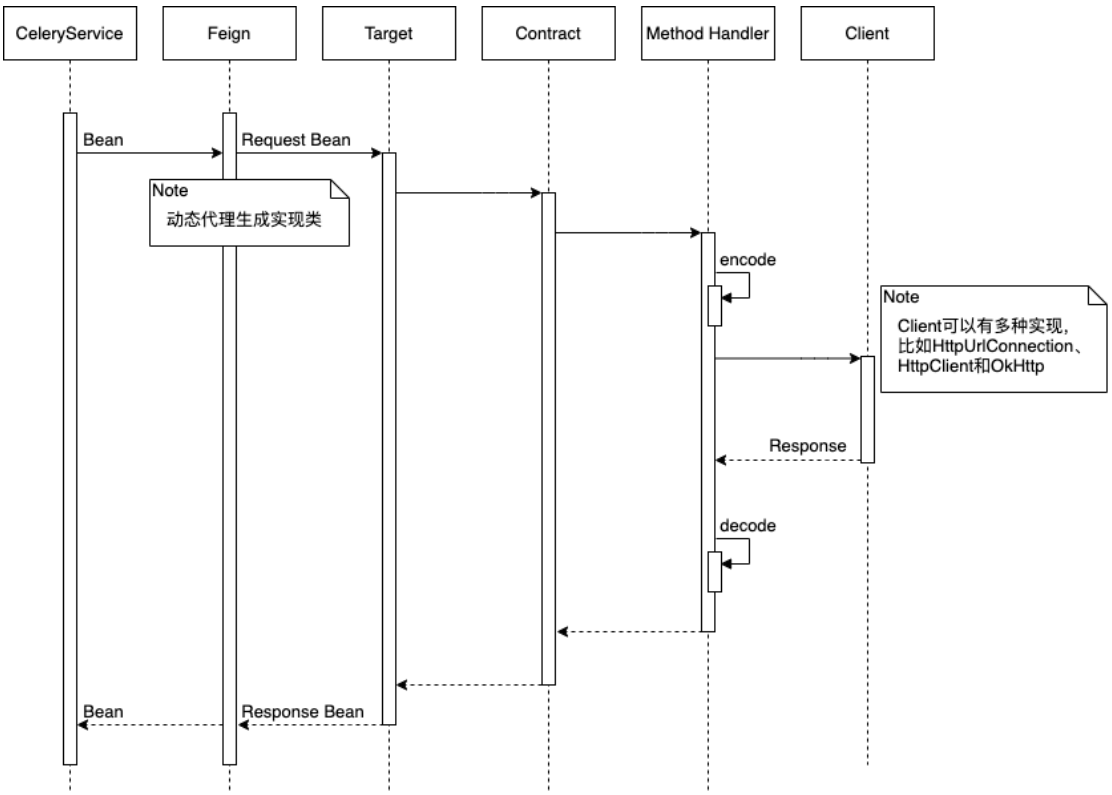


图 4.4: CeleryService 的 Feign 接口调用时序图

CeleryService 接口拥有 @Service 注解和 @FeignClient 注解，这表明了该接口由 Spring 进行生命周期的控制。测试启动方法 start，添加了 @PostMapping 注解，这与 Controller 里该注解的使用方式如出一辙。只不过在 Feign 中，服务端和客户端的角色定位刚好与 Controller 中相反，Feign 是作为 HTTP 请求的客户端，是向远程接口请求服务的一方。在 start 方法中，添加了 @RequestBody 的参数会被 JSON 格式化成 POST 请求的请求体，而方法返回值是由 HTTP Response 的返回体映射而来。图 4.5展示了 CeleryService 接口的具体实现方式。

使用 Feign 提供的注解，我们可以很方便地将构建 HTTP 请求并调用的过程，转为直接调用 Java 的接口方法，调用方不用关心整个 HTTP 请求的过程，也不用关心 Java 实体类与 JSON 字符串的转换，只用着眼于方法提供的具体功能即可。

```
@Service
@FeignClient(name = "celery", url = "${celery.base-url}")
public interface CeleryService {
   @PostMapping(value = "/celery/start",
      consumes = {MediaType.APPLICATION_JSON_VALUE})
   CeleryStartResponse start(@RequestBody final CeleryStartBody body);

   @GetMapping(value = "/celery/status/{session_id}")
   CeleryStatusSessionResponse sessionStatus(@PathVariable("session_id") final
String sessionId);

   @GetMapping(value = "/celery/status/{job_id}")
   CeleryJobResponse jobStatus(@PathVariable("job_id") final String jobId);

   @DeleteMapping("/celery/stop/{session_id}")
   CeleryStatusSessionResponse stopSession(@PathVariable("session_id") final String
sessionId);

   @DeleteMapping("/celery/stop/{job_id}")
   CeleryJobResponse stopJob(@PathVariable("job_id") final String jobId);

   @PostMapping(value = "/celery/retry",
      consumes = {MediaType.APPLICATION_JSON_VALUE})
   CeleryJobResponse retry(@RequestBody final CeleryStartBody body);

   @GetMapping("/celery/result/{job_id}")
   CeleryJobResult getResult(@PathVariable("job_id") final String jobId);
}
```

图 4.5: CeleryService 接口代码

4.2.3　测试任务管理

测试任务管理功能模块是很常见的 CRUD[1]业务功能模块。涉及的主要有 MTWebTaskService 类、MTWebTaskRepository 接口。这两个类负责处理测试任务的增、改、查询功能。MTWebTaskService 类添加了 @Service 注解，由 Spring 负责控制生命周期，通过 @Autowired 方式在别的服务中进行注入。MTWebTaskRepository 接口是 Spring 与 MongoDB 的交互组件，添加了 @Repository 注解。接口继承 MongoRepository<MTWebTask, String> 接口，通过泛型表明 MongoDB 的文档数据格式将会被映射为 Java 类 MTWebTask。采用泛型 DAO 模式，目的是数据访问代码的类型安全，同时也能提高代码的可读性和通用性 [41]。

[1]增加 (Create)、读取 (Retrieve)、更新 (Update) 和删除 (Delete)

4.3　任务调度模块

任务调度模块是将测试接入模块提供的 Celery 子任务，通过 RabbitMQ 消息队列通知分布式的 Celery Worker。Worker 接收到任务后，启动执行引擎单元的 Docker 镜像，构建容器，并监听整个执行引擎单元的生命周期。下面几个小节将介绍任务调度模块相关功能的详细实现。

4.3.1　Celery 任务接收和分配

Celery 是 Python[2]的一个分布式任务调度库。因此，在任务调度模块中，我们使用 Python 程序来接入 Celery。

由于涉及到 Java 的 Spring Boot 程序与 Python 程序的交互，即两个独立的进程交互，我们选择使用一个简单的 HTTP 服务作为 Python 程序的交互入口。Flask[3]是一个 Python 轻量级 Web 应用程序框架，通过简单的配置即可启动 HTTP 服务。

```
signatures = []
session_tool_name = request.json['toolName']
args = request.json['jobs']
queue_name = request.json['queueName']

for i in range(0, len(args)):
       if 'toolName' in args[i].keys() and args[i]['toolName'] != '':
          tool = args[i]['toolName']
       else:
          tool = session_tool_name
       signatures.append(Signature(task='clientWrapper.start', args=(tool,
                str(args[i]['params']), celeryconfig.rabbit_queue),
                queue=celeryconfig.rabbit_queue))
   task_group = group(signatures, app=app)
   group_result = task_group()
```

图 4.6: 创建 Signature 对象

通过 @flask.route('/celery/start', methods=['POST']) 注解，我们定义了一个 HTTP 的 POST 接口，在这里接收服务接入模块中所提到的 CeleryService 类发出的 HTTP 请求，将请求参数从 JSON 格式字符串解析为 Python 的字典。接着，根据子任务的个数，将子任务打包为 Signature 对象，指明 Celery Worker 客户端参数，作为通讯媒介的消息队列配置。该过程的实现方式如图 4.6所示。

通过 group 函数创建并执行一组 Celery 任务，此时任务消息将会被提交到

[2]https://www.python.org/

[3]http://flask.pocoo.org/

消息队列中，主线程中可以获取这一组任务的任务对象 group_result 列表。从任务对象列表中可以获取异步任务返回值，其中包括任务运行的 ID。将这些 ID 构建成 HTTP 方法的返回体，交由调用方处理。

4.3.2　子任务启动和结果保存

Celery Worker 所需要执行的子任务入口由 @app.task() 表明。子任务会收到前文所述的 Signature 对象中的启动参数 args。该参数指明了 tool（工具名称）和其他配置 params。

```
try:
    # prepare args
    args_dict = eval(args)
    extra_args = args_dict['extraArgs']

    self.update_state(state='PROGRESS_RUNNING')

    # start the service by the args
    image_name = tool_name + ":latest"

    # generate command
    command = "docker run --rm --name %s %s start %s" % (task_id, image_name,
extra_args)

    print(command)

    commands = command.split(' ')
    c = subprocess.Popen(commands, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    # get the result and error
    (stdout, stderr) = c.communicate()

except Exception, e:
    self.returnCode = 1
    raise Exception(e)
```

图 4.7: Celery Worker 任务启动

如图 4.7所示，在启动工具执行前，首先将 Worker 子任务的执行状态设置为 PROGRESS_STARTING，表明任务正在启动。在从启动参数中将参数字符串解析成 Python 字典对象后，任务准备阶段结束。

在任务启动阶段，程序将任务执行状态设置为 PROGRESS_STARTING。由于执行引擎单元工具是以 Docker 镜像构建的，因此 Celery Worker 所要完成的工作是启动一个控制台子进程。通过 Python 提供的 subprocess 模块，启动一个新的进程，执行 Docker 命令创建工具容器。subprocess 模块还可以连接

stdin/stdout/stderr 的管道，获取子进程的进程号。Docker 容器的整个生命周期因此可以由 Worker 管理，并获取运行过程的返回值。

由于整个执行过程的主要原始数据是通过 RabbitMQ 消息队列和 Redis 进行实时传递，并且 Docker 容器的所有中间状态将会在容器执行完成后销毁，因此 Docker 容器的标准输出和错误输出以日志输出为主。

Celery 通过 Backend 来保存异步任务结果，这里的结果即 Docker 容器的标准输出和错误输出。通过指明 Backend 保存方式，在 Docker 结束运行后，结果会被导入 Backend 所指定的存储对象中。在本文中，Backend 选择了文件形式存储，其表现形式为 result_backend 被设置为 “file:///root/celery/results” 。因为 Docker 的执行结果主要以工具运行日志为主，浏览器日志和性能监控数据不通过标准输出的形式返回给报告生成服务，所以工具运行日志以文件形式持久化是一个不错的选择。文件存储形式简单，也方便应对后续可能出现的对工具运行日志分析的需求。Backend 等 Celery 的相关配置如图 4.8所示。

```
from kombu.common import Queue

rabbit_user = 'web'
rabbit_pass = '***'
rabbit_virtual_host = '***'
rabbit_host = '115.29.203.43'
rabbit_port = 5672
rabbit_queue = 'mt_web_client_task_queue'

broker_url = 'amqp://' + rabbit_user + ':' + rabbit_pass + '@' + rabbit_host + ':'
   + str(rabbit_port) + '/' + rabbit_virtual_host
result_backend = 'file:///root/celery/results'

# add the Queue you want to listen
task_queues = (
   Queue(rabbit_queue),
)
```

图 4.8: Celery Backend 配置

4.4　测试执行模块

测试执行模块是整个执行服务的实际执行者。执行模块构建的主要产物为以可执行 jar 文件打包成的 Docker 镜像，在存储到私有的镜像仓库后，可以从任意安装有 Docker 服务的宿主机上下载镜像并构建 Docker 容器运行。测试执行模块的主要交互方式为命令行交互、RabbitMQ 消息队列交互和 Redis 交互。测

试执行模块中的执行引擎单元依赖于 Selenium Hub 提供的 Remote Driver 接口进行浏览器的操作。下面几个小节将详细描述测试执行模块的主要实现。

4.4.1　执行引擎单元构建

执行引擎单元的实体在系统中呈现为 Docker 镜像。Docker 镜像通过 Docker 服务提供的 run 命令生成 Docker 容器，并开始执行容器内部设定好的程序。

Docker 镜像由一系列的基础镜像构建，其镜像内容包括了可执行程序、配置文件、类库以及文件系统。多个可读的镜像层组成一个完整的 Docker 镜像。当容器成功构建后，需要获取某一个文件，容器会从上到下去下一层的镜像层去获取文件，直到最后一层。

本文的执行引擎单元 MTWebClient 的 Docker 镜像是基于 OpenJDK[4]的镜像搭建而成。OpenJDK 的镜像提供了 Java 8 的完整运行环境，MTWebClient 作为可执行 jar 文件以命令行启动的方式运行。

图 4.9给出的是执行引擎单元构建 Docker 镜像所需要使用到的 Dockerfile 文件。FROM 配置指出了该镜像是基于 openjdk:8-jre-alpine 原始镜像构建而成。openjdk:8-jre-alpine 摒弃了与 Java 程序正常运行无关的所有功能，保留最低限度的可执行文件和运行环境，因此整个镜像的大小得到了有效的限制。

```
FROM openjdk:8-jre-alpine

RUN adduser -D -s /bin/sh ise
RUN mkdir /workspace && chown ise:ise /workspace

WORKDIR /workspace

ADD entrypoint.sh entrypoint.sh
RUN chmod 755 entrypoint.sh && chown ise:ise entrypoint.sh
USER ise

ENTRYPOINT ["./entrypoint.sh"]

ADD mt-web-client*.jar mt-web-client.jar
```

图 4.9: Dockerfile 镜像构建

在 Dockerfile 文件中配置程序执行的用户、权限，以及程序的工作空间（文件夹）。文件 entrypoint.sh 是启动镜像时需要执行的脚本，通过 ENTRYPOINT 配置指明，一个 Dockerfile 中只能有一个 ENTRYPOINT。当一个 Docker 镜像被启

[4]http://openjdk.java.net/

动运行后，ENTRYPOINT 中所执行的进程的生命周期代表了整个 Docker 镜像所构建的容器的生命周期。进程执行结束后，Docker 容器也将被停止或者是直接销毁。

通过 ADD 配置将我们运行所需要的程序和文件拷贝进入原始的 Docker 镜像，在本项目中主要有两个文件，一个是 entrypoint.sh 启动脚本，一个是 mt-web-client.jar 即执行引擎单元打包完成的 Java 程序。

4.4.2　执行引擎单元配置

如前文所述，执行引擎单元的启动参数将以命令行参数的形式传递。因此，在程序的 main 方法中，需要对命令行参数解析并映射成程序所能使用的实体类。

使用 airline[5]实现命令行参数的解析。airline 库提供了通过注解的方式将命令映射为 Java 类。一般情况下，通过命令行启动 Java 程序的整个命令会如下所示“java -jar XXX.jar <args>”。“<args>”会从 main 方法的“String[] args”参数被程序获取到。

如图 4.10所示，通过 @Command 注解，<args> 参数中顺序最靠前的 start 命令会被 RunnerCommand 类所捕获。RunnerCommand 中的成员变量上添加了 @Option 注解，start 命令后形式如“-C ... -H ... -R ...”等的参数会被依次赋值给各个成员变量。由于整个配置参数嵌套结构较为复杂，成员变量以 String 类型先接收配置内容被格式化后的字符串，在 run 方法中通过 fastjson 还原为 Java 实体对象，供后续功能使用。

4.4.3　执行器类实例构建

为了保障执行引擎单元的核心执行器 CrawljaxRunner 按照需求设定的约束条件运行，必须在实例化 CrawljaxRunner 对象前完成约束的配置。由于配置所需要设定的项目十分繁重，在本项目中，需要配置的不仅仅有页面搜索所需要的状态数、页面深度、最大执行时间这些约束，还包括本执行引擎单元所需要使用的浏览器和浏览器所处的操作系统的配置，页面操作对象的元素约束，登录权限验证流程等。为了有效获取执行过程中浏览器的状态、性能日志和执行路径，在关键的运行节点处还需要添加自定义的插件 Plugin 用于将数据回传给报告生成服务。整个执行器的消息通讯还依赖于各个数据服务的客户端使用，包括 Redis、RabbitMQ、OSS 等客户端。

面对这些复杂的依赖关系，必须实现有效的方法以减少组件之间的耦合程

[5]https://github.com/rvesse/airline

```
@Command(name = "start", description = "Start crawljax task.")
public class RunnerCommand implements Runnable {

    private static final String COMMAND_QUEUE = "QUEUE_LIFETIME";

    private static final Logger LOG = LoggerFactory.getLogger(RunnerCommand.class);

    @Option(name = {"-C", "--clientConfig"}, description = "MTWeb Client
        Configuration(JSON)")
    private String MTWebClientConfigurationModelStr = null;

    @Option(name = {"-H", "--hubHost"}, type = OptionType.GLOBAL,
        description = "Selenium Hub Host Url")
    private String hubHost = "http://localhost:4444/wd/hub";

    @Option(name = {"-R", "--redisConfig"}, type = OptionType.GLOBAL,
       description = "Redis Configuration(JSON)")
    private String redisConfigurationModelStr = null;

    // ...

    @Override
    public void run() {
       // ...
    }
}
```

图 4.10: 命令行参数解析

度，为整个程序的可扩展性考虑。本文使用 Guice[6]实现组件之间的依赖关系注入，将配置和数据服务客户端的生命周期进行统一管理，以模块的形式明确执行引擎单元的各个组件的依赖关系。

Guice 是 Google 开发的轻量级依赖注入框架，能够对构造函数、属性和方法进行注入。Guice 以模块 module 这一概念组织代码，使得整个代码设计的产物与代码中的 module 类一一对应，能够更好地梳理模块依赖关系。这些方法避免将依赖项硬编码在程序的源代码中，而是转向运行时进行组合，在运行时建立依赖关系，然后运行时环境会根据上下文推理出这些依赖关系 [42]。

执行引擎单元的整个架构和代码设计主要分为三个部分：执行器核心页面搜索功能、数据服务连接客户端、日志等状态收集插件。因此，将执行引擎单元的主要架构依次分为以下三个模块：RunnerModule、MiddleModule、PluginModule。如图 4.11所示，其中 PluginModule 依赖于 MiddlewareModule，RunnerModule 依赖于 PluginModule 和 MiddlewareModule。接下来会分别介绍 MiddlewareModule、

[6]https://github.com/google/guice

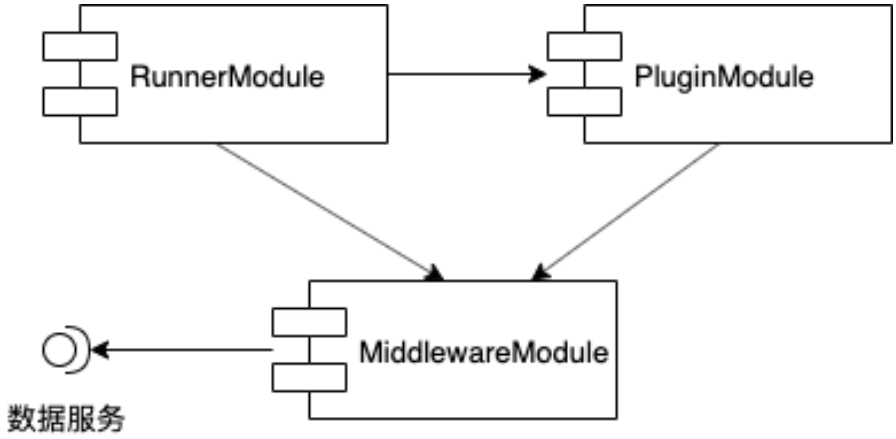


图 4.11: 执行引擎单元模块依赖

PluginModule 和 RunnerModule 的具体实现。

MiddlewareModule 类继承了 AbstractModule 类，负责处理所有与 RabbitMQ、Redis 和 OSS 数据服务所需要的连接客户端的依赖关系。连接配置由 ExtraConfiguration 类传入模块。如图 4.12所示，本项目中所使用的连接客户端主要有 JedisPool 类、Connection 类（RabbitMQ）和 OSS 类。通过 Guice 提供的 bind 方法，将这三个类的具体实例绑定为单例工具类返回的客户端实例，保证在整个程序的生命周期中，每个客户端对象的实例只有一个，避免出现多实例资源竞争的情况。单例工具类控制对象实例的创建与销毁，在整个搜索执行完毕后统一进行关闭连接、资源释放操作。

连接客户端的使用方无需了解客户端的创建过程和具体实例，只需通过 Injector 类获取模块提供的对象实例，如图 4.13所示。Guice 通过 MiddlewareModule 实例获取 Injector 实例。接下来，使用方只需要根据客户端的类对象就可以获取对象实例，而不用在使用方代码中初始化客户端或者是引用单例工具类，将客户端的创建和销毁与使用方隔离。

PluginModule 与 RunnerModule 的配置需要基于 MiddlewareModule 提供的数据服务客户端实例。在 PluginModule 的具体实现中，所有 Plugin 的实例在 PluginModule 类中创建。RunnerModule 类中添加了 PluginModule 的实例化，并作为 RunnerModule 的一个部分共同提供注入的实例对象。

RunnerModule 提供了最终的执行器 CrawljaxRunner 实例化的实现。如图 4.14所示，由于 CrawljaxRunner 的配置过于复杂，不适合在 RunnerModule 中直接进行实例化，CrawljaxRunner 的创建将交由 CrawljaxRunnerProvider 提供解决方案。

CrawljaxRunnerProvider 的角色类似于工厂类所承担的角色，而 CrawljaxRun-

```
public class MiddlewareModule extends AbstractModule {

   private final ExtraConfiguration extraConfig;

   public MiddlewareModule(ExtraConfiguration extraConfig) {
      this.extraConfig = extraConfig;
   }

   @Override
   protected void configure() {
      if (Objects.nonNull(this.extraConfig.getRedisConfig())) {
         bind(JedisPool.class)
            .toInstance(JedisUtils
               .getInstance(this.extraConfig.getRedisConfig()));
      }
      if (Objects.nonNull(this.extraConfig.getRabbitmqConfig())) {
         Connection rabbitConnection
            = RabbitmqUtils.getConnection(this.extraConfig.getRabbitmqConfig());
         bind(Connection.class).toInstance(rabbitConnection);
      }
      if (Objects.nonNull(this.extraConfig.getMongodbConfig())) {
         MongoDatabase mongoDatabase
            = MongodbUtils.getDatabase(this.extraConfig.getMongodbConfig());
         bind(MongoDatabase.class).toInstance(mongoDatabase);
      }
      if (Objects.nonNull(this.extraConfig.getOssConfig())) {
         OSS ossClient = OSSUtils.getInstance(this.extraConfig.getOssConfig());
         bind(OSS.class).toInstance(ossClient);
      }
   }
}
```

图 4.12: MiddlewareModule 具体实现

nerProvider 作为一个抽象父类，其具体实现交由 DefaultCrawljaxRunnerProvider 提供。这一套流程实现了抽象工厂模式，可以通过使用不同的工厂实现构建不同的 CrawljaxRunner 对象实例，提高整个执行引擎单元的可扩展性。

CrawljaxRunnerProvider 实现了 CrawljaxRunner 类的主要配置过程。基于获取到的配置类 MTWebClientConfigurationModel 实例的成员变量的值，CrawljaxRunnerProvider 使用 CrawljaxConfigurationBuilder 来完成执行器的配置实例生成，提供包括执行时间、搜索状态数、搜索深度深度的约束配置。

图 4.15展示了在 CrawljaxRunnerProvider 中对浏览器的选择。由于执行引擎单元是以 Docker 容器的形式运行，Docker 容器内部并没有本地可以运行的浏览器，因此必须使用 Selenium Hub 提供的 Remote Driver 远程连接的形式启动浏览器运行。使用 Selenium Hub 的同时也将对不同品牌、版本的浏览器调度功能的

```
Injector middlewareInjector
   = Guice.createInjector(new MiddlewareModule(extraConfig));

JedisPool jedisPool = Objects.nonNull(extraConfig.getRedisConfig())
   ? middlewareInjector.getInstance(JedisPool.class) : null;

Connection rabbitConnection = Objects.nonNull(extraConfig.getRabbitmqConfig())
   ? middlewareInjector.getInstance(Connection.class) : null;

MongoDatabase mongoDatabase = Objects.nonNull(extraConfig.getMongodbConfig())
   ? middlewareInjector.getInstance(MongoDatabase.class) : null;

OSS ossClient = Objects.nonNull(extraConfig.getOssConfig())
   ? middlewareInjector.getInstance(OSS.class) : null;
```

图 4.13: MiddlewareModule 的 Injector 实现

```
@Override
protected void configure() {
   install(new PluginModule(this.mtWebClientConfig, this.extraConfig,
          this.jedisPool, this.rabbitConnection,
          this.mongoDatabase, this.ossClient, this.noticeOfEnd));

   bind(MTWebClientConfigurationModel.class).toInstance(this.mtWebClientConfig);
   bind(ExtraConfiguration.class).toInstance(this.extraConfig);

   if (Objects.nonNull(this.jedisPool)) {
      bind(Jedis.class).toInstance(this.jedisPool.getResource());
   }
   if (Objects.nonNull(this.rabbitConnection)) {
      bind(Connection.class).toInstance(this.rabbitConnection);
   }
   if (Objects.nonNull(this.mongoDatabase)) {
      bind(MongoDatabase.class).toInstance(this.mongoDatabase);
   }

   bind(CrawljaxRunnerProvider.class).to(DefaultCrawljaxRunnerProvider.class);

   bind(CrawljaxRunner.class).toProvider(CrawljaxRunnerProvider.class);
}
```

图 4.14: RunnerModule 具体实现

依赖从执行引擎单元中剥离，降低了代码耦合。对浏览器的数据收集中，日志形式的浏览器页面渲染数据、JavaScript 网络交互数据需要通过开启浏览器日志和性能日志。这些配置在代码中以 LoggingPreferences 参数来设置，我们需要的两类原始数据分别为 LogType.BROWSER 和 LogType.PERFORMANCE。

```
DesiredCapabilities capabilities;
switch (browserConfigModel.getBrowserType()) {
    case CHROME:
        capabilities = DesiredCapabilities.chrome();
        break;
    case FIREFOX:
        capabilities = DesiredCapabilities.firefox();
        break;
    case IE:
        capabilities = DesiredCapabilities.internetExplorer();
        break;
    default:
        capabilities = new DesiredCapabilities();
}
LoggingPreferences preference = new LoggingPreferences();
// 开始浏览器与性能日志
preference.enable(LogType.BROWSER, Level.ALL);
preference.enable(LogType.PERFORMANCE, Level.ALL);
// preference.enable(LogType.DRIVER, Level.ALL);
capabilities.setCapability(CapabilityType.LOGGING_PREFS, preference);

capabilities.setPlatform(browserConfigModel.getPlatform());

capabilities.setBrowserName(browserConfigModel.getBrowserType().getName());
if (Objects.nonNull(browserConfigModel.getVersion())) {
    capabilities.setVersion(browserConfigModel.getVersion());
}
```

图 4.15: 开启浏览器日志记录

DefaultCrawljaxRunnerProvider 是抽象类 CrawljaxRunnerProvider 的子类，实现了 Provider<CrawljaxRunner> 接口的抽象方法 get。图 4.16是 DefaultCrawljaxRunnerProvider 类的具体实现。使用 @Singletion 注解向 Guice 表明该类的实例在整个模块的运行过程中一定为单例的形式存在。在构造方法上加入 @Inject 注解，Guice 会根据构造方法所需要的参数类型自动从 RunnerModule 模块中找到绑定的对象实例，进行自动注入。get 方法是声明了 Provider 接口所需要实现的方法。该方法是通过 Provider<CrawljaxRunner> 工厂类返回 CrawljaxRunner 对象实例的唯一方法，对应的是 RunnerModule 中的 toProvider(CrawljaxRunnerProvider.class) 语句。

在 RunnerModule 配置完成生成 CrawljaxRunner 实例所需要的一系列组件和方法后，我们就可以通过 RunnerModule 的 Injector 来自动装配 CrawljaxRunner 实例，通过 getInstance 方法获取对象引用。

```
@Singleton
public class DefaultCrawljaxRunnerProvider extends CrawljaxRunnerProvider {

   @Inject
   public DefaultCrawljaxRunnerProvider(
                                MTWebClientConfigurationModel mtWebClientConfig,
                                ExtraConfiguration extraConfig,
                                PluginsBuilder pluginsBuilder) {
      super(mtWebClientConfig, extraConfig, pluginsBuilder);
   }

   @Override
   public CrawljaxRunner get() {
      CrawljaxConfigurationBuilder builder
          = this.getCrawljaxConfigurationBuilder();
      return new CrawljaxRunner(builder.build());
   }

}
```

图 4.16: DefaultCrawljaxRunnerProvider 的具体实现

4.4.4　Web 应用测试路径

在前面的章节中已经讨论过关于当前 Web 应用测试存在的问题，主要在于对动态 Web 应用的 HTML 文件和 JavaScript 文件进行静态分析，其文件内容在静态的情况下无法展示应用的所有事件交互。因此，使用真实浏览器进行 Web 应用的动态分析目的就是为了解决异步事件交互这个需求，并且能够生成 Web 应用的测试路径。

Yunxiao Zou 等人在文章 A Hybrid Coverage Criterion for Dynamic Web Testing[23] 中提出了一种对 Web 应用状态的定义：

定义 4.1.（***Web* 应用状态图**）一个动态 *Web* 应用 A 的状态转换图 G 可以表示为一个三元组 $\langle i, V, E\rangle$，并且：

- i 是 *Web* 应用 A 加载完成后的初始化状态（通常是 *Web* 应用的入口）；
- V 是表示 *Web* 应用 A 的运行时 *DOM* 状态的一个集合；
- $E \subseteq V \times V$ 是顶点之间的一条有向边。当且仅当通过在状态 v_1 中的 *HTML* 元素 ec 上执行事件才能从状态 v_1 到达 v_2 时，边 (v_1, v_2) 才存在。

在 Web 应用状态图的基础上，从 Index 到 $v \in V$ 的任意一条路径可以被视为 Web 应用 A 中从 Index 状态到达 DOM 状态 v 的一条测试路径。路径上的每一个节点可以作为测试的检查点，路径上的每一条边可以被视为一个测试输入。

Web 应用状态图的整体结构如图 4.17所示。

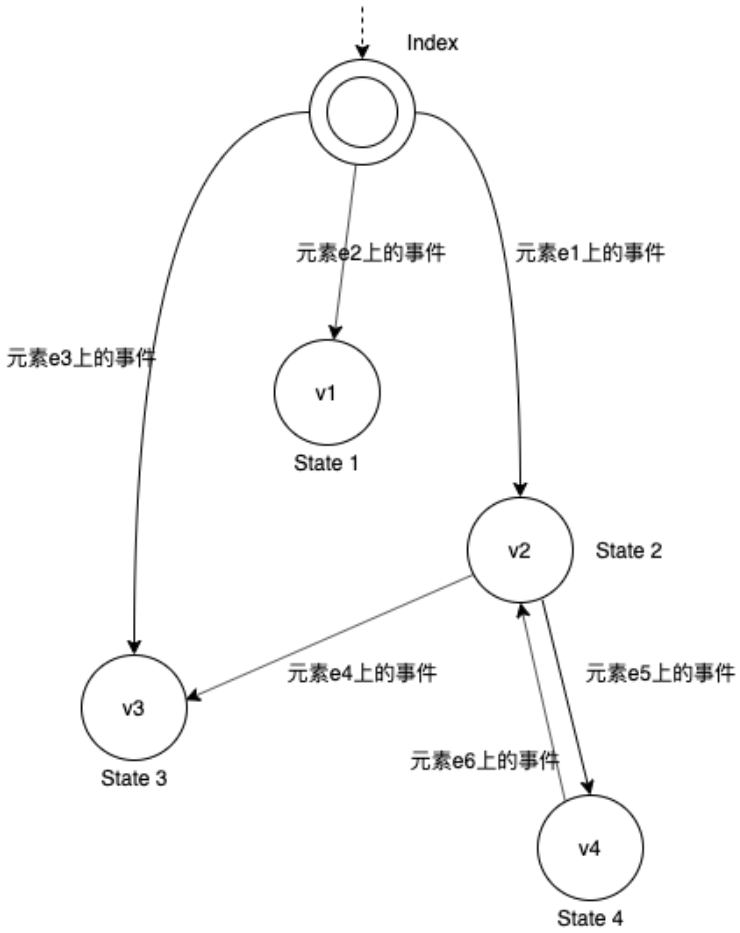


图 4.17: Web 应用状态图

HTML 上的 DOM 事件有很多中，包括点击事件、超链接事件、键盘事件、悬停事件、表单提交事件等。其中超链接事件和表单提交事件通常可以通过点击事件或者键盘事件触发。因此，本文主要关注在点击事件上和键盘输入事件上。HTML 元素的定位有多种方法，包括 xpath、name、id、tag、text、partialText 这些常见的元素选择器。由于有的时候元素有很多属性是空的，或者是重复的，这时候使用 xpath 选择器会更加方便，也更加精准，因此本文采用的定位方式主要以 xpath 定位为主。

根据 Web 应用的状态图的性质，我们可以了解到在程序运行过程中需要生成的这张状态图是一张典型的有向图。本文使用 JGraphT 库提供的 DirectedPseudograph 类作为有向图的基础实现，在这之上封装了 Web 应用状态的一系列方法。以 StateVertex 类作为图的顶点，代表三元组中的 V；以 Eventable 类作为图的边，代表三元组中的 E。

```
public interface StateFlowGraph {
   // ...
   boolean canGoTo(StateVertex source, StateVertex target);
   ImmutableList<Eventable> getShortestPath(StateVertex start, StateVertex end);
   ImmutableSet<StateVertex> getAllStates();
   ImmutableSet<Eventable> getAllEdges();
   List<List<GraphPath<StateVertex, Eventable>>> getAllPossiblePaths(
         StateVertex index);
   // ...
}


public abstract class ModifiableStateFlowGraph implements StateFlowGraph,
Serializable {
   protected final ExitNotifier exitNotifier;
   protected final StateVertexFactory vertexFactory;
   public ModifiableStateFlowGraph(ExitNotifier exitNotifier,
                              StateVertexFactory vertexFactory) {
      this.exitNotifier = exitNotifier;
      this.vertexFactory = vertexFactory;
   }
   public abstract StateVertex putIfAbsent(StateVertex stateVertex);
   public abstract StateVertex putIndex(StateVertex index);
   protected abstract StateVertex putIfAbsent(StateVertex stateVertex,
                                              boolean correctName);
   public abstract boolean addEdge(StateVertex sourceVertex,
                                   StateVertex targetVertex, Eventable clickable);
   abstract StateVertex newStateFor(String url, String dom,
                                    String strippedDom, EmbeddedBrowser browser);
}
```

图 4.18: StateFlowGraph 接口和 ModifiableStateFlowGraph 抽象类声明

StateFlowGraph 定义了 Web 应用状态图接口，其中给出了在搜索前端状态时所需要实现的方法。如图 4.18所示，canGoTo 定义了一个方法用于检查任意两个状态之间的连通性。getShortestPath 定义了一个方法获取两个状态之间的最短路径，以减少从一个状态转移到另一个状态所需要的操作步骤。

本文使用 ModifiableStateFlowGraph 抽象类声明了 StateFlowGraph 接口，添加了一些在整个测试执行过程中所需要用到的成员变量，并且额外抽象出一些方法。ExitNotifier 类负责管理测试执行的终止条件，StateVertexFactory 类是一个工厂类，负责图的顶点类 StateVertex 的装配。putIfAbsent、pubIndex 方法用于向有向图中添加顶点，addEdge 方法用于向有向图中添加边。

InMemoryStateFlowGraph 是 Crawljax 类库中定义的 StateFlowGraph 接口的实现类。本文中将 StateFlowGraph 接口扩展为 ModifiableStateFlowGraph 抽象父类，将与 Web 应用状态图有关的操作作为抽象方法放入 ModifiableStateFlow-

```
@Singleton
@SuppressWarnings("serial")
public class InMemoryStateFlowGraph extends ModifiableStateFlowGraph {

   private final AbstractBaseGraph<StateVertex, Eventable> sfg;
   private final Lock readLock;
   private final Lock writeLock;
   private final AtomicInteger stateCounter = new AtomicInteger();
   private final AtomicInteger nextStateNameCounter = new AtomicInteger();
   private final Map<Integer, StateVertex> stateById;

   @Inject
   public InMemoryStateFlowGraph(ExitNotifier exitNotifier,
                                 StateVertexFactory vertexFactory) {
      super(exitNotifier, vertexFactory);
      sfg = new DirectedPseudograph<>(Eventable.class);
      stateById = Collections.synchronizedMap(new HashMap<>());
      LOG.debug("Initialized the state-flow graph");
      ReadWriteLock lock = new ReentrantReadWriteLock();
      readLock = lock.readLock();
      writeLock = lock.writeLock();
   }

   // ...
}
```

图 4.19: InMemoryStateFlowGraph 成员变量

Graph 类中。InMemoryStateFlowGraph 类中代表真实的有向图的成员变量为 sfg，其类型为 AbstractBaseGraph<StateVertex, Eventable>。其中 StateVertex 代表顶点，Eventable 代表边。由于在整个测试执行过程中会有多个线程对 Web 应用状态图的操作，所以需要使用 readLock 和 writeLock 作为读写锁，保证读写 sfg 的操作是线程安全的。并且相较于使用重量级 synchronized 方法，Java 提供的轻量级并发工具在效率上有十分优异的表现 [43]。

图 4.20展现了 InMemoryStateFlowGraph 中 putIfAbsent 方法的具体实现。该方法实现了向 Web 应用状态图中添加一个状态顶点。在进入方法后，需要使用写锁锁住资源，保证线程安全。接着比较添加的顶点与现有顶点是否有存在相同的，如果存在就直接返回，如果不存在就添加进有向图中，增加状态计数，并返回 null。随着 Web 页面搜索的推进，新的状态会不断出现，此时需要与图中已有的状态进行比较，检查有向图中是否已经存在该状态。

InMemoryStateFlowGraph 作为 ModifiableStateFlowGraph 的一个子类，实现的是在内存中保存 Web 应用状态图。对于后续方便系统分布式集群扩展，本文还实现了另一个子类 RedisStateFlowGraph。RedisStateFlowGraph 将 Web 应用状

```
protected StateVertex putIfAbsent(StateVertex stateVertex, boolean correctName) {
   writeLock.lock();
   try {
      boolean clone = this.hasClone(stateVertex);
      if (!clone) {
         setNearDuplicate(stateVertex);

         boolean added = sfg.addVertex(stateVertex);
         if (!added)
            LOG.info("Vertex should be added !!" + stateVertex);
         stateById.put(stateVertex.getId(), stateVertex);
         int count = stateCounter.incrementAndGet();
         exitNotifier.incrementNumberOfStates();
         LOG.info("Number of states in the graph is now {}", count);
         return null;
      } else {
         // Graph already contained the vertex
         LOG.debug("Graph already contains vertex {}", stateVertex);
         return this.getStateInGraph(stateVertex);
      }
   } finally {
      writeLock.unlock();
   }
}
```

图 4.20: 添加状态的方法具体实现

态图信息的具体存储提升到 Redis 缓存中，并且使用 Redis 的原子操作实现读写上的 lock 和 unlock 方法，避免多个执行引擎单元之间读写冲突。RedisStateFlowGraph 的成员变量基本与图 4.19中的成员变量一致，将 AbstractBaseGraph、Lock 和 AtomicInteger 的具体实现由 Java 原生多线程并发替换为使用 Redis 原子操作实现的多进程、多线程并发。

4.4.5　测试执行过程

整个测试执行过程即是从 Index 状态（Web 页面初始状态）出发，不断推进 Web 应用状态图的构建完善，最终达成在约束条件下对 Web 应用进行测试的功能需求，尽可能多地覆盖 Web 应用的状态空间。测试执行过程不仅涉及到 Web 应用状态图的完善，还涉及到测试执行生命周期的管理、Web 应用 DOM 元素解析、事件候选元素处理、关键状态监听等。

如算法 1所示，测试执行流程开始，会先设置好 Browser（浏览器控制对象）、Crawler（执行对象）、StateMachine（状态机）、UnfiredCandidateActions（待执行动作）、Plugins（插件）、Context（测试环境上下文）和 ExitNotifier（结束条件）。在执行启动前首先触发 Plugins 的 preCrawling 事件，此时会启动一些测试

执行前的方法 runPreCrawlingPlugins，包括向分析服务发送任务启动信号、执行引擎单元使用 Redis 同步统计启动数量等。测试执行正式启动进入 Crawl(Index) 方法，首先根据用户给出的 Web 应用地址打开站点入口，并获取初始状态 Index，解析 Index 内容并将首页的待执行对象解析完毕、状态机中加入初始状态。接下来是一个循环，终止条件由 ExitNotifier 控制。其中终止条件包括：状态机中的状态数达到设定值、执行时间达到设定值、Web 应用程序所有状态被搜索完成、执行中出现严重错误异常退出。

一次循环中所要执行的任务的页面状态顶点会从一个阻塞队列中获取。每一个任务执行的循环过程中，存在一个重要的挑战：如何进行状态回溯以及判断两个状态是否相同。由于 DOM 的动态更改未记录在浏览器历史记录中，因此 AJAX 技术不允许正确使用浏览器的“后退”和“前进”功能，从而使状态机内的导航更加困难。由于任意的状态之间不能直接跳转而至，而当前浏览器所处的状态与任务状态不一定是同一个状态。在这种情况下，我们假定每次从初始状态 Index 进行同样的操作流程后，到达的状态都相同。这样，只要目标状态存在于 Web 应用状态图中，且存在有一条路径可以从 Index 状态到达，就可以从 Index 状态通过 Dijkstra 算法 [39] 寻找一条最短操作路径，通过浏览器重新执行一遍该路径，到达我们需要的目标状态，并且我们认为执行完成到达的状态与之前每一次该操作路径的执行结果都相同。在目前算法中如何判断两个状态是否相同，采用的方法较为简单：比较两个状态下的 DOM 内容是否完全一致。我们期望在页面上经过的每一种操作都能够对当前的页面内容进行改变，比较 DOM 的字符串格式内容是在时间效率和比较精度上都能够接受的一种方法。

在到达目标任务状态后，uca.pollActionOrNull 通过状态 ID 从一个线程安全的 Map 中获取当前状态的待触发事件阻塞队列，获取并移除队列第一条事件。使用执行对象的 fireEvent 方法操纵浏览器触发事件，得到一个新的页面状态。接着，根据比较状态机中已经存在的状态判断当前状态是否为新状态。如果状态不存在于状态机中，向状态机的有向图中添加新状态，并将触发事件作为 ps 前一状态和 ns 当前状态的一条有向边添加进状态机中。循环从阻塞队列中获取待触发事件，并根据这一流程不断地使用真实浏览器尝试触发事件，直到产生新状态后退出循环，完成本轮事件触发任务。

在新状态产生后，会触发 Plugins 的 OnNewState 事件，向函数 runOnNewStatePlugins 中传入新状态的引用。事件 onNewState 的主要内容包括通过数据服务向报告生成服务发送原始的浏览器页面数据、浏览器性能日志、浏览器控制台日志、页面截图、测试路径等。这些数据会作为报告服务的基础数据进行整合、比对、分析，生成应用的缺陷数据和功能、性能、稳定性、兼容性、健壮性

Algorithm 1: 测试执行

```
1  input: URL, Config
2  Procedure Main()
3  begin
4      global browser ← initEmbeddedBrowser(URL, Config)
5      global crawler ← initCrawler(Config)
6      global sm ← initStateMachine(Config)
7      global uca ← initUnfiredCandidateActions(Config)
8      global p ← initPlugins(Config)
9      global context ← initContext(Config)
10     global en ← initExitNotifier(Config)
11     p.runPreCrawlingPlugins(context)
12     Crawl(Index)
13     while not en.terminated do
14         s ← uca.awaitNewTask()
15         Crawl(s)
16 Procedure Crawl(State ps)
17 begin
18     crawler.reachFromHome(ps)
19     action ← uca.pollActionOrNull(ps)
20     while action != null do
21         event ← initEventable(action.element, action.eventType)
22         crawler.fireEvent(event)
23         dom ← browser.getDom()
24         if sm.newState(dom) then
25             ns ← sm.addState(dom)
26             p.runOnNewStatePlugins(context, ns)
27             sm.addEdge(ps, sm, event)
28             sm.changeToState(ns)
29             uca.addCandidateElements(ns)
30             action ← null
31         else
32             action ← uca.pollActionOrNull(sm.getCurrentState())
```

切面钻取数据。

在一次循环的最后，会根据用户配置的约束条件，获取新状态下的 DOM 中待交互事件。该交互事件会作为新状态的元素与新状态一起存入 UnfiredCandidateActions 中等待后续循环中获取。

```
@Singleton
@ThreadSafe
public class DefaultExitNotifier extends ExitNotifier {
    private final CountDownLatch latch = new CountDownLatch(1);
    private final AtomicInteger states = new AtomicInteger();
    private final AtomicBoolean isEnded = new AtomicBoolean(false);

    public DefaultExitNotifier(int maxStates) {
        super(maxStates);
    }

    @Override
    public ExitStatus awaitTermination() throws InterruptedException {
        latch.await();
        return reason;
    }

    @Override
    public int incrementNumberOfStates() {
        int count = states.incrementAndGet();
        if (count >= maxStates) {
            reason = ExitStatus.MAX_STATES;
            latch.countDown();
            isEnded.set(true);
        }
        return count;
    }

    // ...

    @Override
    public boolean terminated() {
        return isEnded.get();
    }
}
```

图 4.21: DefaultExitNotifier 类部分实现

图 4.21展示了控制执行结束的 ExitNotifier 类的一个子类 DefaultExitNotifier 的部分实现。整个执行引擎单元运行过程中会启动多线程处理，DefaultExitNotifier 作为一个单例线程安全的类担起结束执行线程的职责。latch 成员变量是 CountDownLatch 类型，作用是使给定数量的线程等待其他线程执行完毕后再执行。在执行引擎单元中，前者线程指的是主线程，后者线程是 Crawling 工作线

程，所以设置的等待线程数量为 1。方法 awaitTermination 由主线程调用，主线程会被阻塞在 latch.wait() 语句处，等待某一个约束条件被执行到。以最大状态数这一约束条件为例，工作线程在每获取到一个新的 Web 应用状态后，会调用 incrementNumberOfStates 方法，令成员变量 states 执行一个自增的原子操作 incrementAndGet，并获取新的值。如果状态 states 数量已经超过设置的 maxStates，将会调用 latch 的 countDown 方法，唤醒主线程，并将原子布尔值 isEnded 置位 false。此时前文循环的条件表达式值为 false，循环结束，工作线程终止。

在正常流程的三种结束条件（最大时间、最大状态、状态空间全部搜索完成）中，前两个结束条件均与 ExitNotifier 相关。DefaultExitNotifier 中，最大时间是使用一个独立线程的 Thread.sleep(maximumCrawlTime) 操作，在 maximumCrawlTime 时间后唤醒线程并启动任务关闭流程；最大状态是使用一个原子 Integer 值的自增操作统计当前状态图中状态个数，达到预设最大状态 maxStates 后启动任务关闭流程。

在执行引擎单元遇到运行时错误导致不能继续执行下去的时候，ExitNotifier 的 signalCrawlExhausted 方法会被调用，启动任务关闭流程，回报异常退出。

4.4.6　登录验证逻辑

对于大多数 Web 应用来说，必须登录后才能接触到更多的功能，否则在没有权限的情况下执行一些操作或者访问某些页面会被重定向到登录页面，状态空间被大大限制。登录验证逻辑是测试流程外的一个附加流程，在浏览器开始测试执行前，通过登录验证流程登录 Web 应用系统，将权限信息带入测试执行的流程中，避免被权限拦截。

```
public interface AuthorizingAction {

    void preAuthorizing(EmbeddedBrowser browser, FormHandler formHandler);

    void authorizing(EmbeddedBrowser browser, FormHandler formHandler);

}
```

图 4.22: AuthorizingAction 抽象接口声明

图 4.22是本文定义的权限验证逻辑抽象接口。该接口主要定义了两个抽象方法：preAuthorizing 和 authorizing 方法。方法 preAuthorizing 是在进行登录操作前所需要进行的一系列操作，方法 authorizing 则是具体的验证逻辑。两个方法的传入参数均为 EmbeddedBrowser 和 FormHandler，其代表浏览器的驱动和表单

操作的处理器。使用抽象接口定义权限验证逻辑，限定操作步骤，为系统后续添加新的验证逻辑流程提供便利，提高系统的可扩展性。目前系统实现了使用用户名和密码登录 Web 应用的流程。考虑到现在越来越多的 Web 应用已经采用更加安全的方式，包括扫码登录、手机短信登录，该接口在未来能够有多种具体实现的可能。

```
public class UsernamePasswordAuthorizingAction implements AuthorizingAction {

   // ...
   private URI authUrl;
   private List<Eventable> preAuthorizingActions;
   private List<Eventable> authorizingActions;

   // ...

   @Override
   public void authorizing(EmbeddedBrowser browser, FormHandler formHandler) {
      preAuthorizing(browser, formHandler);
      for (Eventable eventable : authorizingActions) {
         formHandler.handleFormElements(eventable.getRelatedFormInputs());
         try {
            browser.fireEventAndWait(eventable);
         } catch (ElementNotVisibleException | NoSuchElementException e) {
            LOG.info("Auth element not visible");
            Thread.currentThread().interrupt();
         } catch (InterruptedException e) {
            LOG.debug("Interrupted during fire event");
            Thread.currentThread().interrupt();
         }
      }
   }
}
```

图 4.23: UsernamePasswordAuthorizingAction 类部分实现

图 4.23展示了 AuthorizingAction 接口的一个实现类 UsernamePasswordAuthorizingAction。通过用户配置定义传入登录页面的 URL，以及在登录前和登录是所需要的一系列操作的列表。在方法 authorizing 中，首先会执行 preAuthorizing 方法，处理好登录操作前的环境。接着，通过 formHandler 将用户配置的用户名、密码以及其他可能需要的信息填入对应的 DOM 元素中，并通过 browser 触发登录事件。测试执行的过程中可能存在多个线程同时对网站进行搜索，而每个线程会打开不同的浏览器。独立的浏览器进程之间用户数据无法交互，因此线程之间的权限验证是无法通用的。在每个线程的浏览器启动后，权限验证逻辑会都作为测试前的流程执行。

4.4.7　执行引擎单元并行执行、同步与数据交互

在测试接入模块中已经设计了将用户提交的测试任务依照操作系统、浏览器品牌和版本划分为更多子任务并交由调度模块进行任务分配。每个执行引擎单元所承担的为一个测试子任务。通过将不同操作系统、浏览器品牌和版本的测试子任务分配到不同的物理机进程上，整个测试任务的执行过程呈现为完全并行的情况。采用并行充分提高执行服务在兼容性任务执行上的效率。另一方面，在执行引擎单元内部采用多线程并发的方式，将 Web 应用状态导航图的生成通过多线程机制加速。由于通过 WebDriver 控制浏览器访问应用的每个操作步骤从触发事件到页面加载完成存在相当明显的时间间隔，多线程并行能够交错利用不同线程产生的时间间隔，有效提升单个执行引擎单元的测试效率。

在整个测试运行过程中，多个执行引擎单元可能隶属于同一个测试任务下，并行执行引擎单元需要通过数据服务与报告生成服务所处的进程进行交互，所以本文设计了执行引擎单元之间的同步机制以及数据交互机制。

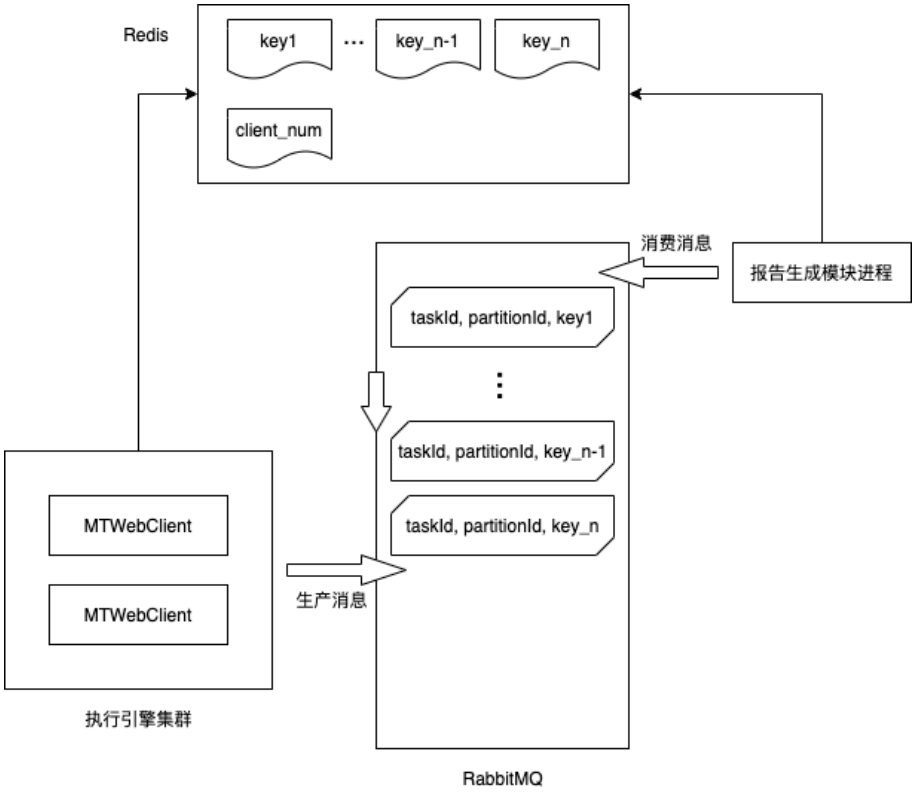


图 4.24: 执行引擎单元同步与数据交互示意图

如图 4.24所示，执行引擎单元的之间的同步和执行引擎单元与报告生成服务之间的数据交互依赖于数据服务 RabbitMQ 和 Redis。在测试执行的过程中，

会产生一系列的原始测试日志数据供报告生成服务分析处理。由于原始数据内容较多，不适合用 RabbitMQ 直接传输，因此 Redis 承担临时中间存储的角色，RabbitMQ 向报告生成服务进程传递数据存储在 Redis 中的 key。执行引擎单元在执行过程中依次向消息队列管道中放入触发消息，包含任务相关的 taskId 和执行引擎单元相关的 partitionId，以及本次原始数据所处的状态 key，报告生成服务会根据这些信息取出本次分析需要的原始数据请求分析服务进行分析，并与 Redis 中相同 taskId 下的历史分析报告不断进行合并处理。

执行引擎单元间依靠 Redis 进行数据同步，其中主要需要同步的参数为：当前时间点一个任务（同一个 taskId）下存活的执行引擎单元数目。该值起到监控 taskId 任务的执行引擎单元生命周期的作用。

```
@Override
public void preCrawling(CrawljaxConfiguration config) throws RuntimeException {
   LOG.info("Start signal sending...");
   String clientNumKey = taskId + "_CLIENT_NUM";

   if (this.jedis.setnx(clientNumKey, "1") == 1) {
      try {
         TaskCommand taskCommand = new TaskCommand();
         taskCommand.setTaskId(taskId);
         taskCommand.setStatus(TaskCommand.TaskStatus.START);

         Channel channel = rabbitConnection.createChannel();
         channel.queueDeclare(COMMAND_QUEUE, true, false, false, null);
         channel.basicPublish("", COMMAND_QUEUE, null,
             JSON.toJSONString(taskCommand).getBytes());
         channel.close();
      } catch (IOException | TimeoutException e) {
         LOG.error(e.getMessage(), e);
      }
   } else {
      this.jedis.incr(clientNumKey);
   }
   LOG.info("Client count: " + this.jedis.get(clientNumKey));
}
```

图 4.25: SignalPlugin 类的 preCrawling 方法具体实现

图 4.25展示了 SignalPlugin 类中的 preCrawling 方法，该方法所实现的功能是在任务开始时的同步。Redis 提供了 SETNX 方法 [44]，该方法实现了当某个 key 不存在的时候，Redis 才会设置该 key。由于 Redis 是单进程单线程模型，所以可以使用 SETNX 方法进行并发中的同步。

在每个执行引擎单元的任务开始之前，首先尝试设置 Redis 中与 taskId 有

关的 clientNumKey 代表的字符串指向的缓存为 1，表示目前有一个执行引擎单元正在执行任务编号为 taskId 的测试任务。当执行引擎单元成功执行 SETNX 操作，说明该执行引擎单元是第一个开始 taskId 任务的，此时打开 RabbitMQ 的一个 Channel，向着通信队列 COMMAND_QUEUE 中发送执行引擎单元 START 的消息，同时也是整个测试任务的唯一开始消息，报告生成服务截获 START 消息会初始化测试任务的测试报告内容。START 消息在一个任务 ID 下有且仅能有一个。

如果执行引擎单元未能成功执行 SETNX 操作，说明 Redis 缓存中已经存在测试任务的运行信息了，此时执单元无需再次通知报告生成服务任务启动，并且使用一个原子自增操作更新 clientNumKey 指向的缓存中的值，代表一个新的执行引擎单元开始执行同一个任务。缓存 clientNumKey 中的值代表了当前执行 taskId 任务的执行引擎单元个数。

SignalPlugin 类中的 postCrawling 方法与 preCrawling 方法刚好相反，代表的是任务结束时的同步。通过 SETNX 原子操作递减缓存中的值，实现执行引擎单元逐步运行结束退出任务的功能。当执行引擎单元的个数减为 0 时，最后一个执行引擎单元就会发送任务结束信息，告知报告生成服务，进行最后的数据整合和报告生成。

4.4.8　Redis 分布式锁实现

Redis 的 SETNX 方法在本文设计的服务中同样用来实现不同进程之间的分布式锁。执行引擎单元在读写临界区数据的时候，使用该实现保证不同引擎单元进程、进程中的线程之间的读写互斥。具体实现如图 4.26所示。

RedisLock 类为每一个请求 lock 方法的线程赋予了独一无二的标识 token，并且在请求加锁的时候加上超时时间作为值的后缀。在一个线程使用 SETNX 方法请求加锁资源后，如果 Redis 中没有 lockKey 值的缓存，则说明没有线程加锁。此时允许该线程加锁并设成超时时间。反之，如果 lockKey 已经存在，则比较 lockKey 的缓存值是否以当前线程 token 为前缀。如果 token 符合当前线程 token 值，则更新锁的持续时间，获取锁成功；反之获取锁失败。在调用 lock 方法处使用 while 循环自旋。

在解锁的时候，获取 lockKey 所在的值。当且仅当 lockKey 的缓存值前缀与当前线程的 token 值一致的时候，删除 lockKey 缓存内容，释放锁资源。

```
public class RedisLock {
   // . . .
   private boolean lock(String token) {
      String timeToken = String.valueOf(System.currentTimeMillis() + this.expire);
      String value = token + DELIMITER + timeToken;
      value = value.trim();
      if (redisTemplate.opsForValue().setIfAbsent(this.lockKey, value)) {
         redisTemplate.expire(this.lockKey, this.expire, TimeUnit.MILLISECONDS);
         return true;
      }
      String currentValue = redisTemplate.opsForValue().get(this.lockKey);
      if (!StringUtils.isEmpty(currentValue)) {
         currentValue = currentValue.trim();
         String[] splits = currentValue.split(DELIMITER);
         if (token.equals(splits[0])) {
            redisTemplate.opsForValue().getAndSet(this.lockKey, currentValue +
DELIMITER + timeToken);
            redisTemplate.expire(this.lockKey, this.expire,
TimeUnit.MILLISECONDS);
            return true;
         }
      }
      return false;
   }
   // . . .
}
```

图 4.26: RedisLock 分布式锁具体实现

4.5　系统界面展示

图 4.27展示了 Web 应用自动化测试的任务列表界面。列表展示了任务的“检测时间”“任务名”“测试网址”“测试服务”“测试进度”“操作”栏目。为了避免一页展示过多的任务条目，方便用户快速查看任务的简要信息，并且减少单次访问的数据加载时间，列表采用分页操作，每页展示 10 条任务内容。用户通过“操作”栏目，可以进行“查看报告”“下载报告”“发布众审”等操作。未开始的任务会显示“执行”按钮供用户点击发起测试执行。

图 4.28展示了 Web 应用自动化测试任务的目标创建界面。用户在此界面填写自动化服务所要扫描的 Web 应用目标。可以通过输入框填写，也可以选择历史网址。在进入配置界面以后，会有多种约束条件需要用户在测试启动开始之前进行填写：限制测试最大时长、限制最大状态数、限制搜索深度、测试执行过程中所需要的一些用户信息、应用可能需要的登录流程等。

图 4.29是测试执行完成后所展示的自动化测试报告应用测试基本概况界面。该界面主要展示了完成的测试任务的基本信息：Web 应用 URL、测试提交时间、

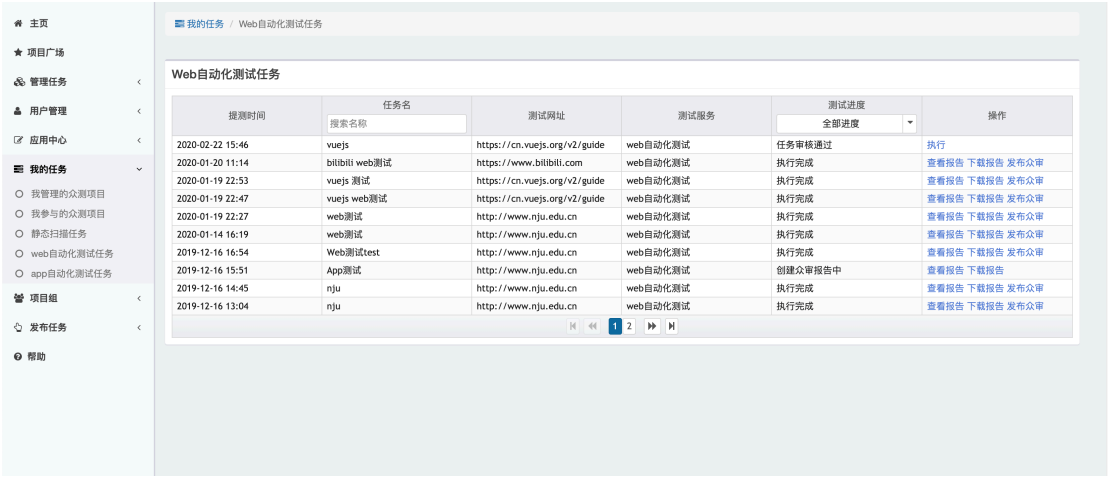


图 4.27: Web 应用自动化测试任务界面

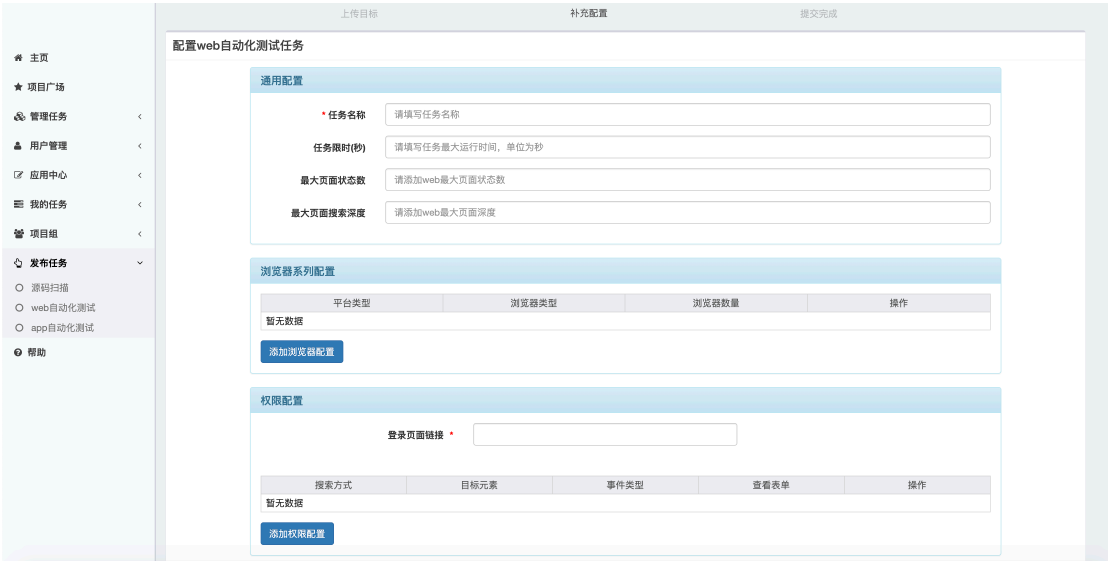


图 4.28: Web 应用自动化测试任务创建界面

测试运行耗时、测试总状态数、测试中找到的缺陷数。在页面下方是整个测试的一个宏观图表分析，分别展示了应用的缺陷在不同平台的比重，应用的缺陷的不同类型的比重和应用的缺陷不同严重程度比重，用户可以直观了解到应用的测试结果情况。饼图中每一个扇形都被添加了点击事件设计，用户可以直接通过点击饼图中的扇形跳转到缺陷列表 Tab 页面下，并且缺陷列表内容经过了点击事件所激发的分类项目的筛选。

图 4.30是 Web 自动化测试报告中一个缺陷的详细内容页面。本页面主要给出 Bug 基本信息以及 Bug 上下文内容。Bug 基本信息主要分为 Bug 类别、所属页面路由、产生 Bug 的执行引擎单元运行的浏览器和浏览器所在的操作系统环境。

图 4.29: Web 应用自动化测试报告概况界面

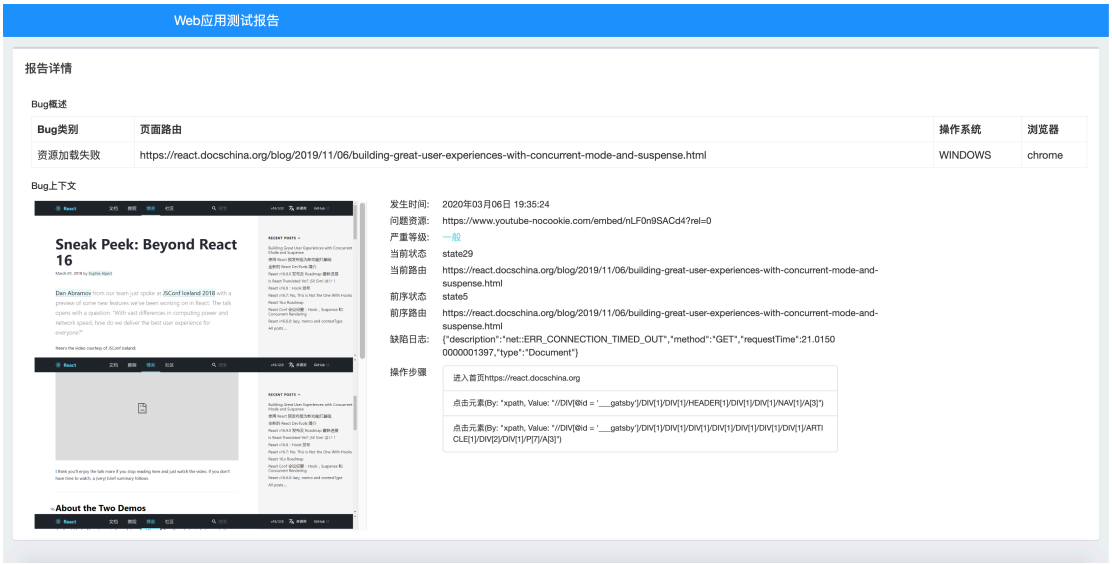
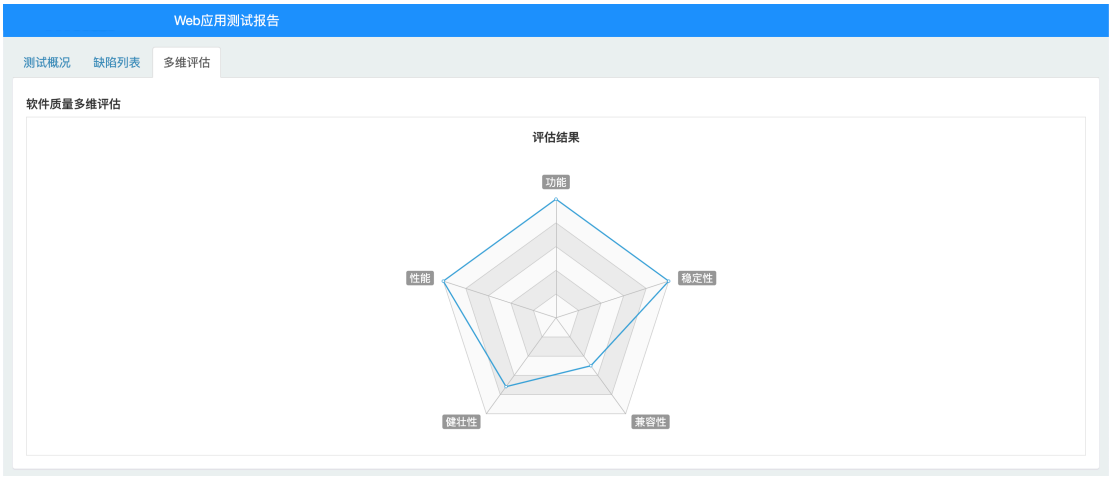
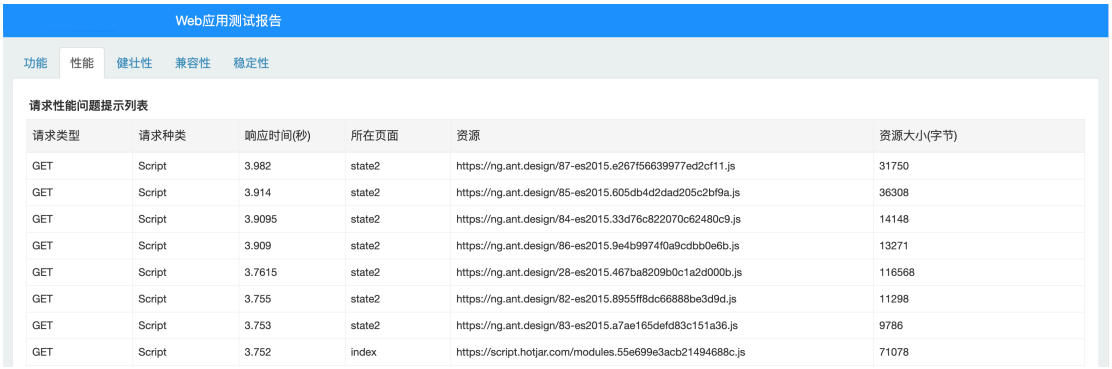


图 4.30: Web 应用测试报告缺陷详情界面

Bug 上下文则详细的描述了 Bug 在什么样的测试路径下产生的，基础内容包括 Bug 发生的时间、产生 Bug 的问题资源（比如容易产生运行时错误的 JavaScript 脚本、容易产生资源加载问题的 CSS 或图片等静态资源）、严重等级、当前状态、Bug 当前路由。为了帮助用户能够复现 Bug，报告给出了 Bug 的一些上下文环境，包括 Bug 产生页面的前序路由，产生 Bug 的所需要的操作步骤。系统还提供了方法，用户可以通过点击操作步骤的相关下载链接，下载对应的 Java 版本 Selenium 自动化脚本文件。在上述详细信息的左侧还给出了 Bug 上下文所在的页面的截图，方便用户快速理解是出问题的页面及当时所处的页面情况。



(a) Web 应用质量多维评估



请求类型	请求种类	响应时间(秒)	所在页面	资源	资源大小(字节)
GET	Script	3.982	state2	https://ng.ant.design/87-es2015.e267f56639977ed2cf11.js	31750
GET	Script	3.914	state2	https://ng.ant.design/85-es2015.605db4d2dad205c2bf9a.js	36308
GET	Script	3.9095	state2	https://ng.ant.design/84-es2015.33d76c822070c62480c9.js	14148
GET	Script	3.909	state2	https://ng.ant.design/86-es2015.9e4b9974f0a9cdbb0e6b.js	13271
GET	Script	3.7615	state2	https://ng.ant.design/28-es2015.467ba8209b0c1a2d000b.js	116568
GET	Script	3.755	state2	https://ng.ant.design/82-es2015.8955ff8dc66888be3d9d.js	11298
GET	Script	3.753	state2	https://ng.ant.design/83-es2015.a7ae165defd83c151a36.js	9786
GET	Script	3.752	index	https://script.hotjar.com/modules.55e699e3acb21494688c.js	71078

(b) Web 应用质量维度切面

图 4.31: Web 应用质量多维评估界面（性能切面）

图 4.31是 Web 自动化测试报告中应用质量评估的详细界面。本页面主要以一个雷达图来展示 Web 应用的五个维度的质量：功能、性能、稳定性、健壮性、兼容性。从雷达图的每项标签上可以跳转到一个新的页面，用于展示从执行服务收集到的原始测试数据中钻取到的各维度数据切面。比如功能切面被展现为执行服务在规定的运行时间和运行状态数下所搜索到的 Web 应用状态导航图，性能切面则是关注浏览器请求交互时间和页面加载时间，健壮性切面关注应用在执行过程中产生的缺陷数量，兼容性关注不同平台运行时应用状态导航图的差异，稳定性关注 Web 应用执行过程中是否会崩溃。通过这五个维度的雷达图和具体维度切面，用户可以全面了解报告中的应用质量，指导用户完善应用。图中选择了性能切面作为切面页面的展示。性能切面展示了请求性能问题提示列表。列表内容是整个运行过程中 Web 应用在浏览器中与后台的资源请求交互时间超过一定阈值的请求数据，按照请求时间长度倒序排列。每一条请求给出了

请求类型、请求的资源种类、响应时间、所在页面、请求资源和资源大小这几项数据，为用户直观展示资源加载问题。

4.6　本章小结

本章的主要内容是是详细介绍了 Web 自动化测试系统执行服务的各个模块的具体的内部设计和代码实现。主要介绍了测试接入模块的测试启动参数、测试任务启动、测试任务管理，任务调度模块的任务接收分配、子任务启动和结果保存，执行引擎单元的执行引擎单元构建、配置读取、执行器类实例构建、状态转换、执行流程、登录验证、数据同步等系统模块内容，并且给出了较为详细的代码实现。最后，本章给出了系统的可视化界面作为一个对本系统直观的展示。

第五章　系统测试与实验分析

本章前几节已经给出了执行服务的详细设计和具体实现，完整的服务已经部署在慕测公司内部的云服务器上。为了保证执行服务的设计和实现能够满足需求分析时给出的功能需求和非功能需求，本节将依据需求对 Web 自动化测试系统执行服务展开测试。

表 5.1: 测试环境

参数	**参数详情**
机型	阿里云 ECS
数量	2 台（1 台 Linux 服务器，1 台 Windows 服务器）
CPU	Linux: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz Windows: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
内存	Linux: 16GB Windows: 16GB
带宽	Linux: 10M Windows: 10M
操作系统	Linux: Ubuntu 16.04.6 LTS Windows: Windows Server 2016
JVM 版本	Linux: OpenJDK 64-Bit Server VM (build 25.232-b09, mixed mode) Windows Java HotSpot(TM) 64-Bit Server VM (build 25.231-b11, mixed mode)
Redis 版本	3.0.6
RabbitMQ 版本	3.8.2
MongoDB 版本	4.2.2
Seleinum 版本	3.141.59

5.1　测试环境

根据第三章所展示的部署图，本服务中测试接入模块部分和执行引擎单元部分将会被部署到 Linux 服务器上。服务中间件 Celery 和 Docker 也将会运行在 Linux 操作系统环境下。浏览器集群中，Selenium Hub 会被设置在 Linux 服务器

上，而 Selenium Node 由于是提供不同环境下的浏览器测试服务，需要部署在不同的操作系统环境下，目前使用一台 Windows Server 部署。整个系统的测试环境将以阿里云提供的轻量级云服务器为基础搭建测试工作。

表 5.1是系统的测试环境。服务器需要开放以下端口以保证外网的正常访问：Linux 服务器开放 8080 端口（Spring Boot 搭建的测试服务的端口）、6379 端口（Redis 访问端口）、5672 端口（RabbitMQ 访问端口）、27017 端口（MongoDB 访问端口）、5000 端口（Celery 服务端口）、14444 端口（Selenium Hub 服务端口）。

5.2　执行服务功能测试

由于 Web 自动化测试系统执行服务所提供的功能非常具体，因此对整个系统的测试主要集中在测试执行功能上，设计不同的请求测试用例，检查整个测试流程的可用性和稳定性。对执行服务功能进行测试的测试用例如表 5.2所示，主要针对测试执行这一流程中不同的任务通用配置条件下的执行情况进行功能测试，保证执行流程符合用户提供的配置约束，测试时间、测试状态数、测试深度符合用户要求。

表 5.2: 启动执行服务功能测试用例

ID	TUS1
名称	启动执行服务
用例描述	用户通过设置测试目标和通用配置启动测试任务
测试功能	执行服务的正常使用流程
输入/步骤	用户填写待测应用 URL，在通用配置处根据用户自身需要配置测试任务的最大执行时间、最大搜索状态、最大搜索深度，点击提交测试开始执行。
预期结果	1. “我的任务-Web 自动化测试任务”页面在开始执行后，所处任务状态变为“正在执行”； 2. 任务结束后，状态转为“执行完成”； 3. “操作”栏展示“查看报告”“下载报告”“发布众审”按钮； 4. 用户可以通过“查看报告”按钮查看测试已经完成的对应任务报告。

在 Web 自动化执行的过程中，没有 Web 应用的权限会导致整个状态空间范围变小。因此，用户能否通过执行服务给出的权限流程配置功能扩展自动化测试执行的状态空间，是 Web 自动化测试系统的能否适应更多的 Web 应用的一条

重要途径。表 5.3展示了启动权限验证流程的测试用例，让用户配置 Web 应用的登录流程，完成对服务执行前的权限验证操作的功能测试，保障前置操作的有效性和完整性。

表 5.3: 启动权限验证流程测试用例

ID	TUS2
名称	启动权限验证流程
用例描述	用户通过设置 Web 应用登录操作流程完成测试流程中的登录操作
测试功能	执行服务的权限验证流程
输入/步骤	用户填写登录所需的操作流程，包括需要填写的用户名输入框、密码输入框、登录提交按钮，完成登录操作配置，并开始测试。
预期结果	1. 执行服务在正式启动测试前，通过用户给出的流程登录待测 Web 应用，在此之后启动测试； 2. 确定可以测试需要登录后权限的页面。

5.3　执行服务性能测试

5.3.1　执行引擎单元性能测试

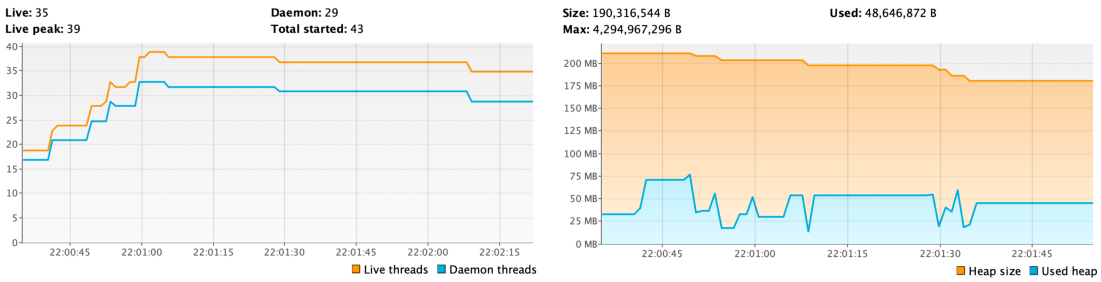


图 5.1: 执行引擎单元客户端性能

每个执行引擎单元的性能影响到执行服务的整体运行效率和后续扩展服务所需要的服务器性能参数。因此，对执行引擎单元进行合理的压力测试可以了解单个执行引擎单元能够承载的任务规模。

本部分使用 VisualVM[1]监视单个执行引擎单元在整个运行过程中的内存使用情况和线程使用情况。测试内容选取 https://ng.ant.design/docs/introduce/zh 网

[1]visualvm.github.io

站，在规定的 1 个小时执行时间内，可以达到 73 个搜索状态，属于在测试网站中较高的一个数值。由于状态空间根据算法将会被存储在虚拟机堆中，因此能够给内存一个较大的存储压力，方便我们查看内存峰值。线程方面将浏览器启动个数设置为 10 个（单个执行引擎单元使用时平均浏览器启动个数在 2 5 个），将带来一个较大的线程压力。

图 5.1展示了通过 VisualVM 监控的压力测试的部分时间段内的内存、线程使用图。在整个执行过程中，堆内存大小基本能够稳定在 200MB 左右，线程稳定在 35 个左右，不存在内存泄漏的风险。

5.3.2　测试接入服务性能测试

测试接入服务是一组 HTTP 接口。在实际的业务场景下，由于整个测试执行到结果获取是一个异步的过程，因此在接口的设计中，测试结果获取接口将会被慕测平台企业版主站进行定时轮询，是一个存在高频访问可能性的接口。

采用 JMeter 对测试结果获取接口进行性能测试。首先在 JMeter 测试计划下创建新线程组，其中每个线程都可以看做一个虚拟用户。设置线程组的 Number of Threads 为 80 个，Ramp-Up Period 为 2 秒，这意味着 JMeter 将在 2 秒内启动 80 个线程，模拟 80 个用户同时访问目标接口。在线程组中添加对应的测试接口“/api/v1/task/result/${id}”，使用配置文件对 id 的实际值进行初始化。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
115.29.203.43:...	320	34	27	35	23	262	0.00%	25.4/sec	
TOTAL	320	34	27	35	23	262	0.00%	25.4/sec	

图 5.2: JMeter 测试结果

图 5.2展示了 JMeter 测试的结果。使用 80 个线程对接口进行访问，一共执行了四次，共 320 条访问时间数据，平均访问时间为 34 毫秒，中位数为 27 毫秒，接口响应时间完全在可以接受的范围内。平时在业务场景下基本不存在如此高的并发访问情况，说明当前的接口设计完全能够承载起业务访问的需求。

5.4　实验结果

本文选择市面上常见的各类网站对本文所开发的项目进行自动化测试的实验，以验证 Web 自动化测试系统能够有效进行 Web 自动化测试。实验网站涵盖了当前的 Web 应用的几种技术类型：纯多页应用、多页应用结合 AJAX 动态加载、纯单页应用。对网站的采用的技术分类依据主要依靠 Chrome 提供的开发者工具中 Network 监控网络访问情况。多页应用会在状态改变后，首先通过网

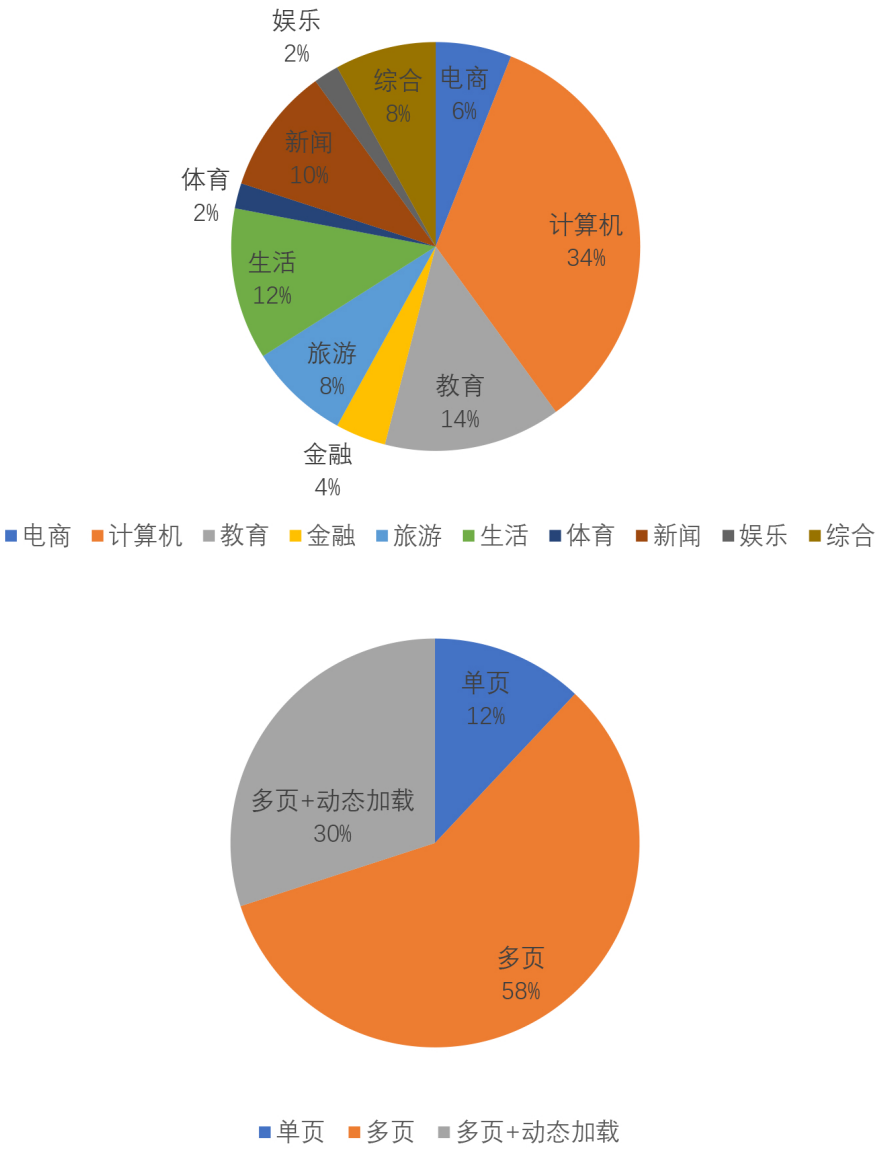


图 5.3: 测试应用类型饼图

络请求一个 HTML 文件作为目标页面，之后再请求 CSS 和 JavaScript 内容，并进行页面渲染，因此我们能够在 Network 中找到每一个状态的 HTML 文件。单页应用只有在第一次访问的时候会请求 HTML 文件，后续的状态改变完全通过 JavaScript 脚本控制页面渲染刷新，因此在 Network 监控筛选 DOM 后我们仅能找到一个请求的 HTML 文件。多页应用结合动态加载是两者的综合，在此类应用中，既存在多个 HTML 文件请求，也有在某个页面上通过 JavaScript 动态刷

新的内容。图 5.3展示了实验的各类网站的主要占比。

在整个网站的实验过程中，我们发现目前的网站技术选型中，一般的新闻网站和一些门户网站由于网站性质以及一些历史延续的原因，主要是以多页应用的形式为主；而新兴的电商、旅游应用网站，比如京东、淘宝、携程、途牛等，由于每个商品都需要一个页面，所以网站主要还是以多页应用为主，但是在商品搜索的页面会通过 AJAX 动态加载的方式避免了刷新页面。

5.4.1　总体情况

本文的自动化测试工具在常见的 50 个网站中的自动化实验结果如下：自动化测试服务限制运行 3600 秒，最大搜索深度为 3（从 Index 状态到任意一个状态的最短距离不超过 3 个节点），在以上约束条件下，平均搜索状态数为 83 个，平均每个应用找到的缺陷数为 81 个。

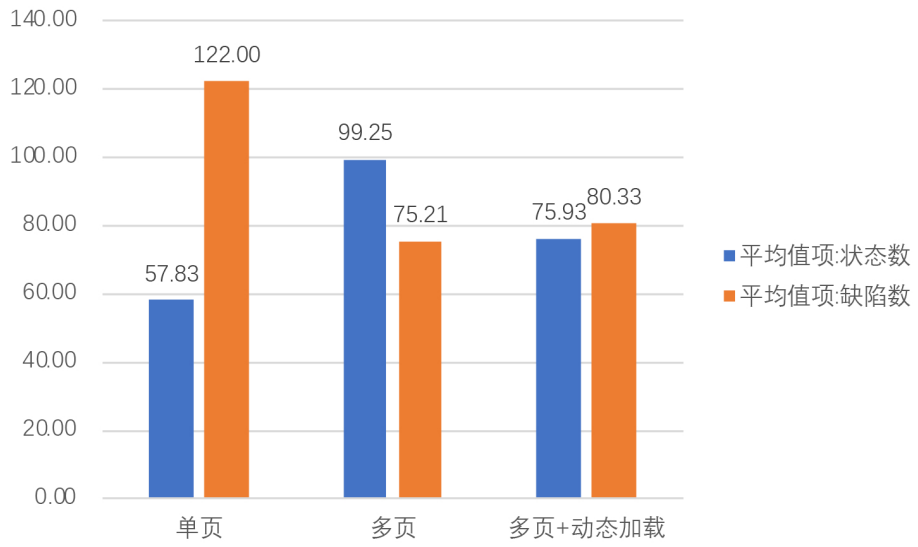


图 5.4: 应用测试状态数和缺陷数总览

图 5.4展示了自动化测试服务在不同技术类别的 Web 应用上的整体测试结果。在规定的一个小时内，工具在单页应用上平均搜索了 57.83 个状态，平均找到 122 个缺陷；在多页应用上平均搜索了 95.52 个状态，平均找到 70.11 个缺陷；在多页应用结合动态加载的应用上，平均搜索了 70.57 个状态，平均找到 86.07 个缺陷。

图 5.5从测试用例复现情况、生成的测试脚本执行情况、缺陷分类准确性三个方面对自动化测试服务的结果进行了评估。整个评估共分析了 632 个状态，涉

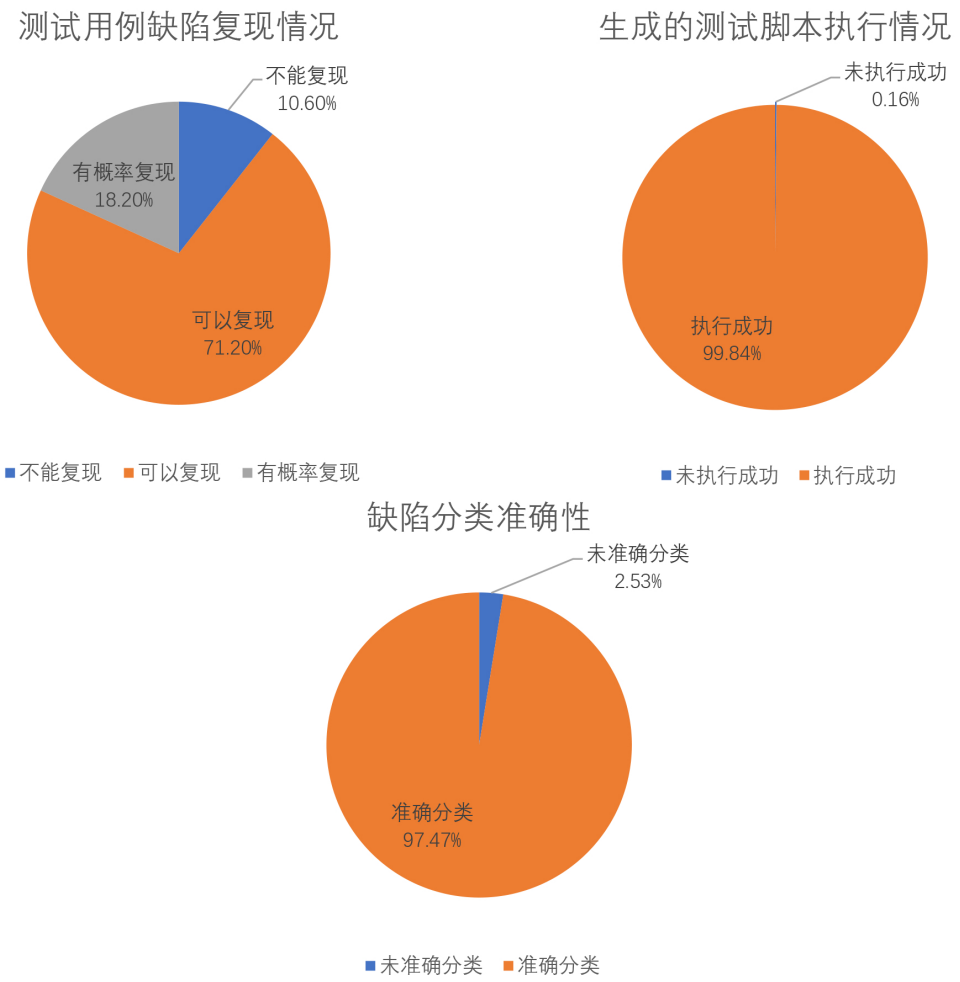


图 5.5: 应用测试结果评估

及报告数占整体的 10%，涉及的网站分类包括计算机、生活、教育行业，涉及的技术类别包括多页应用、多页应用结合动态加载的应用。

测试用例复现情况中，使用自动化生成的测试用例可以完全复现缺陷的比重为 71.20%，有概率复现的情况占比 18.20%。对于某些资源加载问题或是性能问题，无法确保每次使用测试用例的时候都能够将其复现。缺陷产生的原因与应用所处网络状态和应用服务器压力有关，但确实存在风险，因此将其归类为“有概率复现”。可以复现的缺陷与有概率复现的缺陷总占比接近 90%，同时，自动化测试服务生成的测试脚本执行成功率接近 100%。因此自动化测试服务能够助力于测试人员进行有效便捷的 Web 应用自动化测试流程，而为自动化测试服务提供基础支撑（包括从 Web 应用有限状态机转为自动化测试脚本）的执行服务也能够被证明其有效性。

自动化测试服务分析出的缺陷经过人工重新审阅，从提供的缺陷日志判断自动化测试服务对缺陷的分类是否准确。其中准确分类占比为 97.47%，证明了自动化测试服务对缺陷分类的有效性。

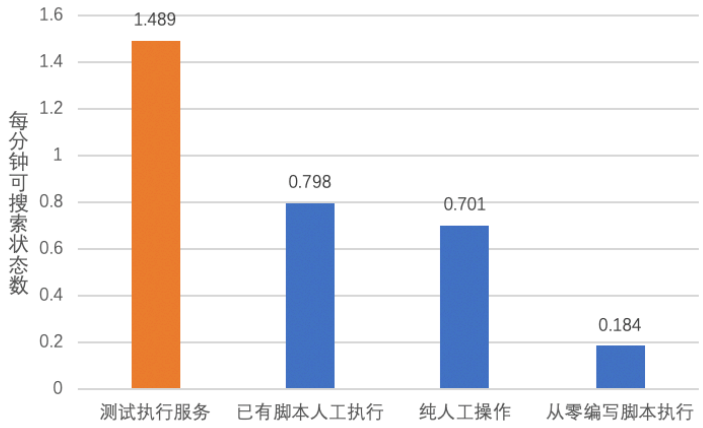


图 5.6: 执行服务性能对比

对所有待测 Web 应用发起的自动化测试任务所限定的运行时间均为 3600 秒即 1 个小时。图 5.6根据自动化测试任务的结果分析执行服务对测试效率提升的有效性。如图所示，根据所有实验内容，执行服务每分钟可以搜索的状态数（DOM 数）平均为 1.489 个，而对相同应用相同状态采用纯人工手动进行测试并查看控制台和网络交互内容每分钟可以搜索的状态数为 0.701 个。因此，执行服务相较于人工测试的效率有一倍左右的提升。另一方面，如果采用从头编写 Web 应用的 Selenium 测试脚本，每分钟可以搜索的状态数为 0.184 个。这是因为人工从网页获取需要操作元素的定位方式，并且编写为 Selenium 脚本所需要完成的内容还是相当复杂的，与执行服务自动搜索并解析有着相当的差距。对于已有自动化脚本的情况来说，整体操作时间会大大缩短。本次实验中自动化运行脚本的情况下，还需要一些人工介入分析最终状态的控制台和网络交互情况。如果把这一部分的内容也尽量通过自动化的方式执行和判断，每分钟能够搜索的状态数将会逐渐逼近执行服务所达成的性能指标。

对 Web 自动化测试服务的测试结果说明，本文所设计的 Web 自动化测试服务无论是传统的多页应用还是通过动态加载的 AJAX 应用，都能够很好地进行支持，并且能够高效搜集、整理、分析 Web 应用的缺陷供测试人员使用。

5.4.2　工具对比

Link Checker[2]是 W3C 的一款传统的 Web 应用自动化测试工具，负责检查 Web 应用内部的断链死链情况。其中对整个应用的搜索过程为“解析页面链接-递归访问链接内容”。其搜索过程与本文的执行服务有相似之处。本文使用 Link Checker 对同样的 50 个网站进行测试，其测试报告的 DOM 数结果如图 5.7所示。该结果与本文系统的测试结果 5.4相比较，可以发现本文的执行服务所支持的自动化测试可以找出更多的 Web 应用状态，能够对应用进行更全面地测试。影响 Link Checker 的测试结果过少的原因主要有：页面可供扩展搜索的链接过少，待测网站禁止非浏览器形式的访问这两种。本文使用的 Web 页面状态模型有限状态机关注用户操作事件而非应用链接，采用真实浏览器访问应用避免网站屏蔽，可以有效解决以上两个问题。

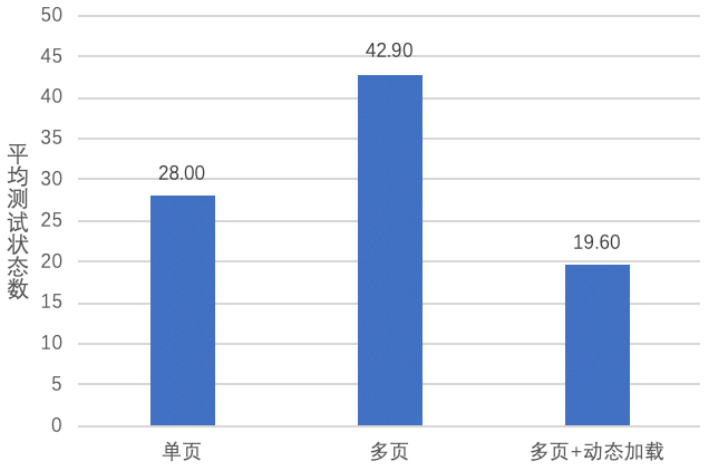


图 5.7: Link Checker 测试结果

从测试结果的细节上来看，在面对一些采用 JavaScript 动态加载构建新颖的单页应用的时候，Link Checker 的表现欠佳。如表 5.4所示，对于表格中所列出的单页应用来说，Link Checker 最多只能访问到给出的 URL 链接的首页内容。而单页应用的首页内容如果没有经过浏览器运行 JavaScript 脚本的过程，其具体内容组件是不会显示在返回的 HTML 文件中的。因此如 Link Checker 这种传统的自动化测试工具无法访问更多的页面，也就无法对 Web 应用进行有效的测试。

本文的执行服务采用真实浏览器运行，能够等待 JavaScript 重绘页面内容，达到与用户操作浏览器同一效果。在面对不同技术所开发的 Web 应用来说，展

[2]https://validator.w3.org/checklink

表 5.4: 单页应用下执行服务与 Link Checker 探测 DOM 数对比

应用 URL	执行服务	Link Checker
https://juejin.im/timeline	102	0
https://angular.cn/	36	1
https://ng.ant.design/docs/introduce/zh	73	1
https://angular.io	136	1
https://ng-alain.com/docs/getting-started/zh	128	1
http://reactjs.luckybird.me/	104	1

现给用户的内容的最终表现形式应该是一致的。因此执行服务不仅仅关注于应用页面 DOM 内容中的 <a> 元素链接跳转，同时也关注诸如 <button>、<input type=”button”>、<input type=”submit”> 的事件交互内容。更进一步，执行服务为测试用户提供方法，允许用户添加除了默认上述元素的其他元素作为交互元素。这样可以将交互元素扩展到所有 HTML 标签中去。

5.4.3　典型案例

本文选择两个典型应用的测试结果进行分析。每个案例执行所使用的约束条件如下所示：

- 运行时间 3600 秒；
- 最大探测深度为 3；
- 操作元素为 <a>、<button>、<input type=”button”>、<input type=”submit”>；
- 没有设置登录验证流程。

图 5.8展示了一个典型的多页应用，某旅游门户网站的测试整体概况。在 3600 秒的约束时间内共搜索状态 59 个，找到缺陷数 341 个。其主要问题集中在资源加载失败和 404 资源未找到上。这个结果在这种旅游类的门户网站上是可以预见的。以该 Web 应用的首页为例，使用 Chrome 开发者工具的 Network 菜单栏查看：首页 Index 状态在 3.19 秒内发送了共 158 条请求，获取的总数据大小约共 5.5MB。其中，该 Web 应用的资源加载问题主要体现在 JavaScript 脚本文件、CSS 样式文件和图片文件上。图 5.9是其中的一份文件 toTop2.js 的 “ERR_ABORTED” 加载问题，该文件推测实现的是网站向下滑动后通过一个按钮回归顶部的功能，此功能因为脚本未能加载导致无法使用。

图 5.8: 某旅游门户网站测试概况

图 5.9: 某旅游门户网站缺陷详情

该旅游门户 Web 应用的 JavaScript 脚本文件、CSS 文件采用的是多个小文件频繁请求，仅 JavaScript 文件的请求数量就达到了 33 条。频繁的请求会使得服务器频繁开启 HTTP 连接，大量耗费资源。用户的响应时间 80%-90% 时间花在 HTML 文档所引用的所有组件（图片，JavaScript，CSS 等）的 HTTP 请求上，HTML 文档所用时间只有 10%-20%。频繁的请求也可能导致某些文件加载出错。本文的自动化测试服务完成了通过对浏览器的各种请求交互的拦截和日志获取，自动化获取到该应用网站文件加载失败的错误信息，并进行整合划分到某一状

态和某一路由下这一功能，帮助测试人员快速了解当前 Web 应用可能存在的资源加载缺陷，并可以快速定位解决。

图 5.10: NG-ZORRO 应用测试概况

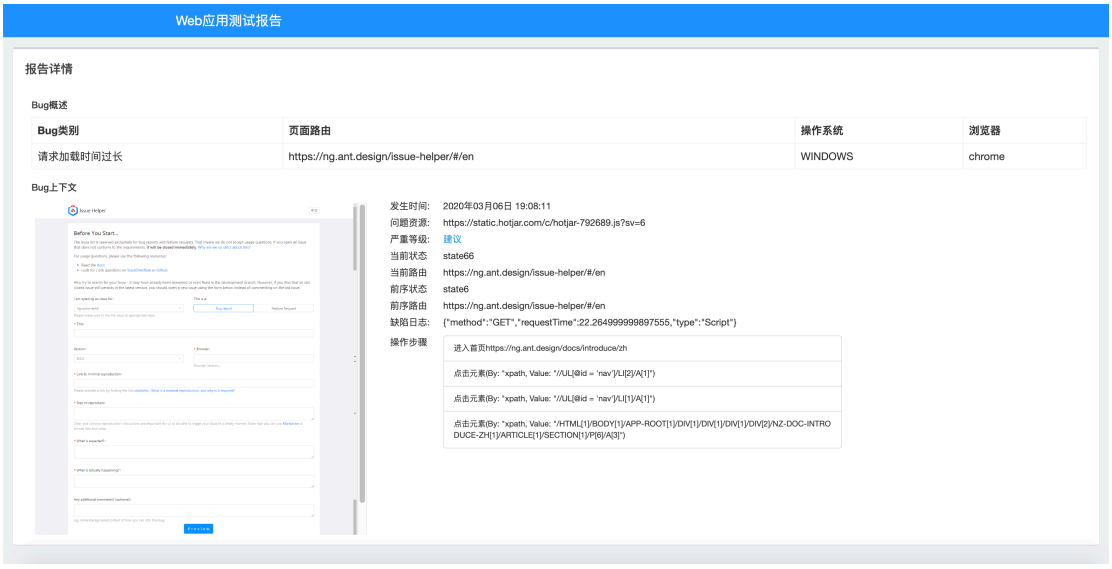


图 5.11: NG-ZORRO 应用缺陷详情

图 5.10展示了一个单页应用 NG-ZORRO 的测试概况。该 Web 应用是 Ant Design[3]在 Angular 框架下实现的教程网站。该 Web 应用基本是一个单页应用，路由切换使用的 JavaScript 脚本提供的功能，而非通过 HTTP 请求重新访问后端

[3]https://ant.design/

加载新的 HTML 文件。由于 NG-ZORRO 网站使用的技术比较先进，并且维护网站的公司有着雄厚的技术实力，自动化测试工具在搜索了 84 个状态后仅找到了 10 个缺陷，并且属于风险程度较低的缺陷。

由于单页应用常常因为采取前端单独打包部署的形式，通过 webpack[4]等工具打包过后的前端资源，会将前端项目中使用到的多份 JavaScript 文件将合并并压缩成一份较大的 JavaScript 文件，CSS 文件也同理。相比较于上文所提到的多页应用，NG-ZORRO 项目在 22 秒内发送了 82 个 HTTP 请求，下载数据资源大小为 5.8MB。其中 JavaScript 脚本文件为 11 份，CSS 文件仅 1 份。

图 5.11展示了本文所完成的自动化测试服务在该应用网站中扫描出的缺陷详情。由于 NG-ZORRO 应用中使用了某些国外的资源，这些资源的加载速度十分缓慢。因为单页应用依赖于 JavaScript 进行事件监听和 DOM 双向绑定，所以应用中 JavaScript 的加载影响着整个页面的展示以及使用。该页面的脚本加载时间达到了 22 秒多，对于该应用的使用用户来说，可以明显感知到使用时的迟滞和操作无响应的情况，影响使用感受。本文的自动化服务模拟真实用户在浏览器中的操作，并且能够通过浏览器日志掌握整个页面的资源加载时间，并提出合理的警示信息。

5.5　本章小结

本章主要阐述了对本文设计的 Web 自动化测试系统执行服务的测试和实验。其主要内容包括执行服务的功能能够完成了设计初的功能需求；执行引擎单元和测试接入服务的性能测试满足要求。接下来介绍了本文的自动化工具在常见类型的网站上运行的自动化测试结果，与现有开源工具 Link Checker 的自动化测试结果比较。最后，着重介绍了一个多页应用和一个单页应用经过自动化测试后分析出的存在缺陷，验证了 Web 自动化测试服务的功能。

[4]https://www.webpackjs.com/

第六章　总结与展望

6.1　总结

为了解决动态 Web 应用的自动化测试问题，本文设计和实现了一种 Web 应用自动化测试系统，以保障在动态 Web 应用下的自动化测试的效率和覆盖率，同时简化测试配置。本文首先介绍了 Web 应用以及其相关技术的发展带来的问题，Web 应用自动化测试的重要性，Web 应用自动化测试领域的相关研究和产品动态，进一步介绍了本文所设计的系统所需要的相关前沿技术，包括 Spring Boot 微服务技术、Docker 容器技术、Redis 高速缓存、RabbitMQ 消息队列以及 Selenium 自动化测试工具。

随后，本文重点分析了本文所需要产出的系统的功能需求和非功能需求，根据功能需求和非功能需求生成相应的用例图和用例描述。根据用例相关内容，本文构建出整个 Web 自动化测试系统的整体架构图，并对 Web 自动化测试系统的执行服务部分的架构图进行详细阐述。由系统整体架构图和执行服务架构，本文将执行服务拆分为三个主要系统模块：测试接入模块、服务调度模块、测试执行模块。其中测试执行模块是整个执行服务的重点，也是实际执行者。而测试接入模块和服务调度模块将共同完成用户发起的测试任务下发和测试任务获取的功能。

接着，本文针对上述划分的三个模块进行设计要点和代码实现的详细描述。测试接入模块主要描述了自动化测试所提供的微服务与用户交互的部分：如何接收测试任务请求并将请求下发，如何通过微服务获取测试任务的详细信息和数据。服务调度模块主要描述了在接收到测试接入模块发送的任务请求后，通过消息队列进行子任务的自动分配，并且启动执行引擎单元开始异步任务，监控子任务的生命周期，获取任务结束后的进程返回值。测试执行模块主要描述了测试执行的复杂环境配置，执行器实例的构建，Web 应用状态图的定义和基于定义设计的测试执行过程，基础的用户名密码登录流程，分布式环境下各个执行引擎单元的同步机制和数据传输。本文展示了整个 Web 自动化测试系统的可视化用户界面。

最后，本文对系统进行的功能测试和性能测试。通过一系列实验和实验数据，证明了本文设计的 Web 自动化测试系统执行服务应对复杂 Web 应用测试的有效性，能够帮助测试人员进行高效的 Web 应用自动化测试，节省测试人员的时间，提高工作效率。

本文构建的 Web 自动化测试系统执行服务，通过采用真实浏览器模拟用户操作的方式，搜索交互元素，构建 Web 应用状态图。采用多执行引擎单元进程的方式考察不同品牌的浏览器的兼容性，采用单个执行引擎单元多线程的方式提升整体的测试执行效率。与传统的静态内容分析相比，可以有效对动态交互的 Web 应用进行自动化测试，并能够根据运行日志输出客观的评分。

6.2　进一步工作展望

本文设计的 Web 自动化测试系统，为慕测企业版平台打下了 Web 应用自动化测试的基础，明确了整个 Web 应用自动化测试的流程。在本文的基础上，还有以下方面具有进一步提升的空间：

一方面，本文设计的主要登录验证逻辑目前还是依赖于用户提供的登录操作步骤，自动化程度欠缺。同样目前测试搜索过程中对表单的填充简单地以随机值填充，或是依赖于用户给定目标元素的输入值。未来可以考虑引入图像识别技术和文本分析技术，针对页面特定的文本框，可以自动识别当前文本框所需要的输入的语义，并生成相应的测试数据。同时，图像识别技术还可以检查 Web 应用在程序功能之外的 Web 应用问题，比如界面布局等会对用户使用感受产生影响的问题。

另一方面，本文设计的 Web 应用测试目前停留在对前端代码和数据交互的测试上。对于整个 Web 应用来说，前端只是 Web 应用的一部分。如果能够获取到 Web 应用的后端日志，并综合前后端测试内容进行自动化分析，将会得到更优的报告质量。在未来可以考虑引入 Web 应用后端日志数据，并入测试流程，增强测试分析。

参考文献

[1] S. Aghaei, M. A. Nematbakhsh, H. K. Farsani, Evolution of the world wide web: From web 1.0 to web 4.0, International Journal of Web & Semantic Technology 3 (1) (2012) 1–10.

[2] S. Murugesan, Understanding web 2.0, IT Professional 9 (4) (2007) 34–41.
URL https://doi.org/10.1109/MITP.2007.78

[3] J. Hendler, Web 3.0 emerging, IEEE Computer 42 (1) (2009) 111–113.
URL https://doi.org/10.1109/MC.2009.30

[4] R. E. Anderson, S. S. Srinivasan, E-satisfaction and e-loyalty: A contingency framework, Psychology & marketing 20 (2) (2003) 123–138.

[5] A. Ahmad, O. Rahman, M. N. Khan, Exploring the role of website quality and hedonism in the formation of e-satisfaction and e-loyalty, Journal of Research in Interactive Marketing 11 (3) (2017) 246–267.

[6] P. Sfetsos, L. Angelis, I. Stamelos, P. Raptis, Integrating user-centered design practices into agile web development: A case study, in: N. G. Bourbakis, G. A. Tsihrintzis, M. Virvou, D. Kavraki (Eds.), 7th International Conference on Information, Intelligence, Systems & Applications, IISA 2016, Chalkidiki, Greece, July 13-15, 2016, IEEE, 2016, pp. 1–6.
URL https://doi.org/10.1109/IISA.2016.7785424

[7] N. G. Maleki, R. Ramsin, Agile web development methodologies: A survey and evaluation, in: R. Y. Lee (Ed.), Software Engineering Research, Management and Applications [selected papers from the 15th International Conference on Software Engineering Research, Management and Applications, SERA 2017, London, UK, 2017], Vol. 722 of Studies in Computational Intelligence, Springer, 2017, pp. 1–25.
URL https://doi.org/10.1007/978-3-319-61388-8_1

[8] S. Malik, C. Nigam, A comparative study of different types of models in software development life cycle, Int Res J Eng Technol.

[9] K. Ali, X. Xiaoling, A reliable and an efficient web testing system, International Journal of Software Engineering & Applications (IJSEA) 10 (1).

[10] S. Dhir, D. Kumar, Automation software testing on web-based application, in: Software Engineering, Springer, 2019, pp. 691–698.

[11] L. D. Paulson, Building rich web applications with ajax, IEEE Computer 38 (10) (2005) 14–17.
URL https://doi.org/10.1109/MC.2005.330

[12] S. Mousavi, Maintainability evaluation of single page application frameworks: Angular2 vs. react (2017).

[13] E. Saks, Javascript frameworks: Angular vs react vs vue.

[14] A. Mesbah, A. van Deursen, A component- and push-based architectural style for ajax applications, J. Syst. Softw. 81 (12) (2008) 2194–2209.
URL https://doi.org/10.1016/j.jss.2008.04.005

[15] A. Mesbah, A. van Deursen, Invariant-based automatic testing of AJAX user interfaces, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 210–220.
URL https://doi.org/10.1109/ICSE.2009.5070522

[16] 黄侨, 葛世伦, 开源 web 自动化测试框架的改进研究, 科学技术与工程 12 (15) (2012) 3630–3635.

[17] F. Ricca, P. Tonella, Analysis and testing of web applications, in: H. A. Müller, M. J. Harrold, W. Schäfer (Eds.), Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada, IEEE Computer Society, 2001, pp. 25–34.
URL https://doi.org/10.1109/ICSE.2001.919078

[18] A. A. Andrews, J. Offutt, R. T. Alexander, Testing web applications by modeling with fsms, Software and Systems Modeling 4 (3) (2005) 326–345.
URL https://doi.org/10.1007/s10270-004-0077-7

[19] A. M. Memon, An event-flow model of gui-based applications for testing, Softw. Test., Verif. Reliab. 17 (3) (2007) 137–157.
URL https://doi.org/10.1002/stvr.364

[20] 牟凯, 顾明, 基于 uml 活动图的测试用例自动生成方法研究, 计算机应用 (04) (2006) 844–846.

[21] 刘龙霞, 吴军华, 基于 uml 活动图的 web 应用测试用例生成, 江南大学学报 (自然科学版) 10 (3) (2011) 283–288.

[22] S. G. Elbaum, G. Rothermel, S. Karre, M. F. II, Leveraging user-session data to support web application testing, IEEE Trans. Software Eng. 31 (3) (2005) 187–202.
URL https://doi.org/10.1109/TSE.2005.36

[23] Y. Zou, C. Fang, Z. Chen, X. Zhang, Z. Zhao, A hybrid coverage criterion for dynamicweb testing (S), in: The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013, Knowledge Systems Institute Graduate School, 2013, pp. 210–213.

[24] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, Z. Gao, Virtual DOM coverage for effective testing of dynamic web applications, in: C. S. Pasareanu, D. Marinov (Eds.), International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, ACM, 2014, pp. 60–70.
URL https://doi.org/10.1145/2610384.2610399

[25] A. Marchetto, P. Tonella, F. Ricca, State-based testing of ajax web applications, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, IEEE Computer Society, 2008, pp. 121–130.
URL https://doi.org/10.1109/ICST.2008.22

[26] A. Stocco, M. Leotta, F. Ricca, P. Tonella, APOGEN: automatic page object generator for web testing, Software Quality Journal 25 (3) (2017) 1007–1039.
URL https://doi.org/10.1007/s11219-016-9331-9

[27] R. Eda, H. Do, An efficient regression testing approach for PHP web applications using test selection and reusable constraints, Software Quality Journal 27 (4) (2019) 1383–1417.
URL https://doi.org/10.1007/s11219-019-09449-2

[28] A. Holmes, M. Kellogg, Automating functional tests using selenium, in: J. Chao, M. Cohn, F. Maurer, H. Sharp, J. Shore (Eds.), AGILE 2006 Conference (AGILE

2006), 23-28 July 2006, Minneapolis, Minnesota, USA, IEEE Computer Society, 2006, pp. 270–275.
URL https://doi.org/10.1109/AGILE.2006.19

[29] 刘伟, 郭秋月, 胡志刚, 基于 selenium 的 web 自动化测试框架优化及应用研究, 电子测试 (20) (2018) 51–53.

[30] J. Anderson, A million monkeys and shakespeare, Significance 8 (4) (2011) 190–192.

[31] T. Wetzlmaier, R. Ramler, W. Putschögl, A framework for monkey GUI testing, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016, IEEE Computer Society, 2016, pp. 416–423.
URL https://doi.org/10.1109/ICST.2016.51

[32] P. Software, Spring, https://spring.io.

[33] D. Rajput, Mastering Spring Boot 2.0: Build modern, cloud-native, and distributed systems using Spring Boot, Packt Publishing Ltd, 2018.

[34] D. Inc., Docker, https://www.docker.com/.

[35] M. Rostanski, K. Grochla, A. Seman, Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq, in: M. Ganzha, L. A. Maciaszek, M. Paprzycki (Eds.), Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014, Vol. 2 of Annals of Computer Science and Information Systems, 2014, pp. 879–884.
URL https://doi.org/10.15439/2014F48

[36] redislabs, How fast is redis?, https://redis.io/topics/benchmarks/.

[37] E. F. Codd, A relational model of data for large shared data banks (reprint), in: M. Broy, E. Denert (Eds.), Software Pioneers, Springer Berlin Heidelberg, 2002, pp. 263–294.
URL https://doi.org/10.1007/978-3-642-59412-0_16

[38] J. Hershberger, M. Maxel, S. Suri, Finding the k shortest simple paths: A new algorithm and its implementation, ACM Trans. Algorithms 3 (4) (2007) 45.
URL https://doi.org/10.1145/1290672.1290682

[39] D. Michail, J. Kinable, B. Naveh, J. V. Sichi, Jgrapht - A java library for graph data structures and algorithms, CoRR abs/1904.08355.
URL http://arxiv.org/abs/1904.08355

[40] A. Ward, D. Deugo, Performance of lambda expressions in java 8, in: Proceedings of the International Conference on Software Engineering Research and Practice (SERP), The Steering Committee of The World Congress in Computer Science, Computer ⋯, 2015, p. 119.

[41] 孟晨, 赵春亮, 张建国, et al., 泛型 dao 模式在 java web 开发中的应用, 计算机应用与软件 29 (01) (2012) 175–177+210.

[42] S. M. A. Shah, J. Dietrich, C. McCartin, On the automation of dependency-breaking refactorings in java, in: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013, IEEE Computer Society, 2013, pp. 160–169.
URL https://doi.org/10.1109/ICSM.2013.27

[43] D. Lea, The java.util.concurrent synchronizer framework, Sci. Comput. Program. 58 (3) (2005) 293–309.
URL https://doi.org/10.1016/j.scico.2005.03.007

[44] redislabs, Setnx key value, https://redis.io/commands/setnx/.

简历与科研成果

基本情况 尹子越，男，汉族，1996 年 6 月出生，江苏省扬州市人。

教育背景

2018.9 ~ 2020.6 南京大学软件学院 硕士

2014.9 ~ 2018.6 南京大学软件学院 本科

这里是读研期间的成果

1. Zhang X, **Yin Z**, Feng Y, et al. NeuralVis: Visualizing and Interpreting Deep Learning Models[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 1106-1109.

致　　谢

时间过得飞快，转眼研究生两年生活已经接近尾声了。在本文末，我向所有帮助过我的老师、同学等表示诚挚的谢意。

首先感谢的是我的导师刘嘉老师和陈振宇老师。在整个论文和项目的完成过程中，两位老师的悉心教诲，帮助我明确论文方向，给予我实验室丰富的资源，为我展现实验室未来方向的架构蓝图。在论文写作、内容编排纠错方面两位老师都给予了指导性建议。

其次我要感谢 iSE 实验室与我一起支持本文所涉及的系统的同学，张晨剑和周赛同学。我依然记得我们三个人在设计和开发系统的过程中相互帮助，相互支持前进的情景。同时，感谢实验室的平台组支持的同学们。在本文的系统接入慕测企业版平台中给予了我们极大的帮助。

我还要感谢研究生和本科与我共度了很长大学生活的舍友们和同学们。是你们与我一路同行，充实了研究生和大学的时光。此外，我还要感谢我的家人，是你们作为我上大学时的坚强后盾。

最后，我要感谢南京大学软件学院的培养。从本科到硕士，是软件学院在这一段时间内为我打开了计算机世界的大门，充实了我的人生。同时也要向百忙之中参与论文评审的各位老师表示由衷的感谢！

致谢基金：

国家自然科学基金项目：基于可理解信息融合的人机协同移动应用测试研究（61802171）, 2019-2021

中央高校基本科研业务费专项资金资助项目：基于群智协同的众包测试技术（14380021），2020-2020

南京大学技术创新基金项目：基于 AI 中台的移动应用测试技术研究与实现（14913413），2020-2020

版权与原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

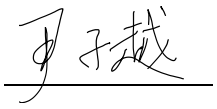
作者签名：

日期：2020 年 5 月 24 日

附件二

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名： 

2020 年 5 月 23 日

论文题名	Web 应用自动化测试系统执行服务的设计和实现				
研究生学号	MF1832225	所在院系	软件学院	学位年度	2020
论文级别	□硕士 √硕士专业学位 □博士 □博士专业学位 （请在方框内画钩）				
作者 Email	mf1832225@smail.nju.edu.cn				
导师姓名	刘嘉				

论文涉密情况：

√ 不保密

□ 保密，保密期（_____年_____月_____日 至 _____年_____月_____日）

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。