文章编号: 1000-5641(2019)04-0120-13

一种分布式可视化 Dubbo 接口测试平台

李艳丽, 张宗勇, 冯 捷, 李志辉

(平安壹钱包电子商务有限公司, 上海 200232)

摘要: 随着互联网金融的发展, 金融业务变得更加复杂, 产品功能迭代更加快速. 为了支持业务的发展, 金融应用开始进行支持 Dubbo 协议的开发. 现有的接口测试框架和工具在支持 Dubbo 协议、多人协作及测试用例维护及数据分析上都存在问题: 首先, 没有很好的工具支持 Dubbo 协议的测试, 已有的 Dubbo 测试框架无法很好地推广; 其次, 复杂业务会涉及多个应用, 目前的框架很少考虑多系统间的自动化配合; 再次, 单机版自动化测试工具或纯编码方式的自动化测试框架不利于多人协作编写自动化测试用例及脚本的维护; 最后, 数据分析一般是查看单个子系统测试用例的执行结果, 无法很好地对数据进行全局分析. 为了更好地管理大批量用例及支持多子系统版本迭代, 在已有老框架基础上, 使用分布式技术设计并实现了一个灵活的可视化的 Dubbo 接口自动化测试平台: 基于界面操作, 提供基于可视化的数据驱动及关键字驱动方式, 支持编写复杂测试用例, 并基于接口解析的方式, 自动生成测试用例. 详细表述了可视化的 Dubbo 接口测试平台的架构、用例管理及用例执行; 展示了新平台与老框架的耗时对比、用例增长对比和新平台日常自动化执行情况.

关键词: 可视化; 分布式; 自动化测试平台; 接口自动解析; 用例管理; 用例执行; 负载均衡

中图分类号: TP399 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2019.04.012

A distributed user-friendly Dubbo interface testing platform

LI Yan-li, ZHANG Zong-yong, FENG Jie, LI Zhi-hui

(*Pingan eWallet Co., Ltd, Shanghai* 200232, *China*)

Abstract: With the development of online finance, the financial business has become increasingly complex and product iterations have become more frequent. Financial applications have started to use the Dubbo protocol for programming to support this prosperous business. There are some problems, however, in supporting testing of the Dubbo protocol, multiplayer collaboration, maintenance, and data analysis with the current testing tools. First, these tools can only be used to test the Dubbo protocol and the Dubbo testing framework that exists at present is difficult to promote. Second, complex businesses involve multiple applications; the current system does not consider automated

收稿日期: 2018-08-06

第一作者: 李艳丽, 女, 硕士, 平安壹钱包资深测试, 主要研究方向为自动化测试和中间件测试.
E-mail: liyanlyee@qq.com.

通信作者: 张宗勇, 男, 平安壹钱包高级总监, 主要研究方向为互联网技术及安全.
E-mail: zzynet@163.com.

collaboration among multiple systems. Third, single-machine automatic testing tools or coded automatic testing frames are not good for multiplayer collaboration in writing automatic test cases and maintenance. Finally, data analysis is usually the execution result when reviewing single-system test cases so it cannot provide a holistic analysis of data. In order to manage a large quantity test cases and support multi-system upgrades, this paper uses distribution technology to design and implement a flexible and visible Dubbo automatic testing platform with the existing testing framework. Based on an interface application, the paper provides visible data-driven and keyword-driven methods and supports the programming of complex test cases. It can also generate the interfaces that you need to test and can generate test cases according to them automatically. This paper describes the architecture of this user-friendly Dubbo testing platform and the management of test cases and execution in a detailed way. Lastly, it shows a comparison of cost vs. time and a comparison of automated test case growth between the old framework and the daily execution results of the new platform's automation testing.

Keywords: user-friendly; distributed; automation test framework; automation interfaces analysis; test case management; test case execution; load balance

0 引　　言

随着互联网金融的发展, 金融业务变得更加复杂, 金融产品功能迭代更加快速[1]. 为了保证应用功能的质量, 金融机构都在探索如何更好地做自动化测试[2]. 根据业界标准的金字塔模型, 接口服务占 32.63%, 对比 UI (User Interface) 自动化, 接口自动化测试性价比更高. 在互联网企业中, 目前使用最广泛的为 RPC[3](Remote Procedure Call, 远程过程调用) 协议 Dubbo[4]. 对于 Dubbo 服务, 迫切需要方便的自动化测试工具来对 Dubbo 协议接口进行测试, 以提高工作效率. 目前主要通过两种方式来测试 Dubbo 接口: 第一种是在已有测试工具上做扩展, 即在扩展路径下放入需要的 jar 包, 这样测试人员可以通过编写脚本或简单界面的方式来编写自动化测试用例. 但这种方式一般是单机版, 不利于多人协作开发及自动化测试用例的整合, 且编码工作量巨大可维护性差. 第二种方式是开发专门的自动化测试框架, 使用编码的方式进行自动化, 本文测试团队目前使用的就是这种方式, 即测试人员搭建开发环境, 使用自动化框架开发自动化测试用例. 但使用这种方式, 有两个问题, 第一, 这种自动化方式很难推广普及. 因为针对不同网段的测试环境, 自动化开发环境需要同步搭建, 另外每位自动化测试人员都需要搭建对应的自动化开发环境, 对于功能测试任务繁重的测试人员, 时间上不能保证. 第二, 数据相对分散, 第一种自动化测试方式也存在同样的这个问题. 自动化测试代码编写好之后, 一般通过 jenkins 来定时执行, 并且需要通过开发新的平台来搜集 jenkins 执行数据, 测试人员需要在不同的平台查看需要的信息.

基于上述原因, 本文设计并实现了一种新的针对 Dubbo 协议的接口自动化测试平台, 平台基于 Web 界面操作, 提供基于可视化的数据驱动及关键字驱动方式, 支持编写复杂测试用例, 并基于接口解析方式, 自动生成测试用例. 通过统一的可视化方式编写和调试测试用例, 测试用户能很方便地完成测试用例的编写、执行及测试结果的查看; 失败原因分析可以对失败的原因进行分类, 让测试人员更好地发现用例编写和代码问题.

从多应用角度出发, 目前接口测试都侧重于单应用系统的自动化, 很难支持系统间的自动化. 本文新的 Dubbo 接口测试平台 (简称新平台) 通过子系统间依赖方式, 支持一个用例管

理中包含多个业务子系统接口的调用, 从而可以测试复杂业务功能链路; 工具集成模块, 可以方便引入并使用外部类.

除此之外本文新平台还提供了数据分析功能. 详细的执行日志、多种类型的报表, 能方便地让测试人员查看自动化执行数据及代码覆盖率, 方便分析测试用例新增及修改状况; 报表导出功能能方便在本地查看并整合数据.

综上所述, 本文贡献点总结如下.

(1) 设计并实现一种针对 Dubbo 协议的分布式接口自动化测试平台, 通过可视化方式支持 Dubbo 协议的自动化测试, 使自动化测试更容易普及.

(2) 可视化的方式支持数据驱动、顺序测试用例上下文串联、多子系统间测试用例串联和使用外部封装的工具类.

(3) 测试用例通过平台方式集中管理, 方便迭代和维护; 支持数据分析, 包括用例执行情况及代码覆盖率, 方便测试人员根据质量报告增加测试用例和测试场景覆盖, 减少漏测.

1 相关工作

自动化测试是把以人为驱动的测试行为转化为机器执行的一种过程. 通常在测试用例设计完成并通过评审之后, 由测试人员根据测试用例中描述的过程按步骤执行测试, 并将得到的实际结果与期望的结果进行比较. 自动化测试使这个过程通过工具或框架的方式由机器自动运行, 从而在版本迭代时, 自动执行重复的手工测试, 缩短测试时间, 提高测试效率. 按照分层的自动化测试概念, 自动化测试分为UI 测试、集成接口测试、单元测试这 3 层. UI 测试是对 UI 层的功能进行测试. 集成接口测试关注的是一个函数、类 (方法) 提供的接口是否可靠. 单元测试关注的是代码的实现逻辑. 分层模型为倒金字塔形, 表示不同阶段所投入自动化测试的比例, 越往上层, 维护成本越高. 正常的产品测试行为为单元测试、接口测试及 UI 测试. 对比接口, UI 层的元素会时常会发生改变. 因而接口自动化测试是相对有效的一种自动化方式.

已有的典型的工具为 SoapUI[5]. SoapUI 主要关注 SOAP[6]、HTTP REST 的服务测试. 对于 Dubbo[4,7] 的测试, SoapUI 支持可扩展方式, 在 SoapUI 的扩展路径下放入 Dubbo 的 jar 包, 然后通过编写脚本方式发送请求. SoapUI 是单机版本, 代码导出方式为 xml 文件, 这种方式在多人协作开发及测试用例整合方面很繁琐. Jmeter[8] 可以测试接口, 但主要用于性能测试, 其编写 Dubbo 测试用例的方式与 SoapUI 相同, 通过可扩展方式支持. LoaderRunner[9] 也是通过扩展方式支持, 测试人员需要掌握函数编写脚本, 其也是用于性能测试, 但需要付费才能使用. 目前开源的测试框架大都是支持 HTTP[10] 协议的自动化测试, 通过编码方式来编写自动化测试用例. 开源自动化测试平台 Phoenix Framework[11], 提供了 HTTP 及 mobile 自动化测试方法, 针对接口测试, 请求参数及请求发送都需要编码实现, 测试人员需要搭建测试环境才能做接口测试. WTAF[12] 基于 IBM 开源框架 STAF(Software Test Automation Framework)[13] 提出了一个模型, 引入 CVS Server 维护测试用例变更并调用 STAF 的服务, 使测试用例分布式执行, 测试人员需要通过代码编写测试用例, 并通过 xml 设定用例执行规则, 通过 STAF 的接口来调用服务包编写好的脚本, 这个框架强依赖 STAF, 借鉴了脚本的分布式执行, 但并没有很好的可视化页面辅助测试人员方便地编写调试测试用例.

基于数据驱动的自动化测试框架, 其基本思想是将测试案例和测试数据分开, 对于每一个测试用例, 可以根据不同的测试数据产生不同的执行结果. 此种方式在初始建立测试数据

时耗费巨大, 同时要求测试人员具备专业的编程功底, 而且在处理复杂业务的自动化测试时, 需要编写大量的代码来保证测试用例的关联. 关键字驱动的自动化测试则扩展了数据驱动的自动化测试, 用关键字代替测试数据来控制测试案例的执行, 减少了维护案例的开销[14]. 不同格式的测试数据和结果数据都统一成 Json[15]格式, 方便处理和结果校验[16].

互联网企业在多元化业务需求下, 子系统中存在较多的共享业务, 企业一般会选择服务化架构来搭建业务子系统. 服务化的核心是 RPC, 由客户端和服务端这两部分组成, 服务提供方所提供的方法需要由服务调用放以网络的形式进行远程调用, 这个过程称为 RPC 请求; 服务提供方根据服务调用方提供的参数执行指定的服务方法, 执行完成后再将执行结果响应给服务调用方, 这是一次 RPC 调用. Dubbo 是一个分布式服务化治理框架, 是一种 RPC 框架, 提供对多种基于长连接的 NIO (New Input/Output)[17]框架抽象封装, 包括多种线程模型、序列化, 以及“请求–响应”模式的信息交换方式. Zookeeper[18]是 Apache Hadoop 的子项目, 是一个典型的分布式一致性解决方案, 是 Google Chubby[19]的开源实现.

现阶段并没有很好的工具支持基于 Dubbo 的应用系统的自动化测试. 目前本文团队使用的测试方式为自身研发的自动化测试框架 (简称老框架), 该框架通过 maven[20]管理; 测试人员通过在 pom.xml 中添加待测 jar 包并执行框架提供的命令来生成待测接口; 测试人员通过编码方式组装 Dubbo 请求并对响应结果进行验证; 在固定的文件夹 TestCase 中存放测试数据, 命名规则为 Dubbo 服务接口名称为子文件夹名称; 每个服务接口对应不同的测试方法, 测试数据存放在 excel 文件中, 文件名称根据待测试方法来命名; 测试人员需要通过编码组装接口的请求对象和返回对象, 并调用测试方法; 框架根据数据和组装的 Dubbo 请求生成 TestNG[21]的测试用例, 并执行. 框架自动生成接口类并自动生成自动化测试用例, 一定程度上减少了测试人员测试单接口用例的负担. 但对于复杂场景, 涉及多个测试方法的串联, 测试人员需要在请求中加复杂逻辑进行用例的串联, 并且分不同的分支来判断响应结果的差异, 这对测试人员的代码能力要求很高. 特别是针对不同测试环境, 由于数据的变更, 需要对请求及验证代码重新编写及调试, 则需要懂得编程的自动化测试人员编写测试用例代码, 因此很难在功能测试人员中推广和普及.

基于老框架, 本文提出了一种可视化的平台进行 Dubbo 自动化测试——Dubbo 接口测试平台测试平台通过版本方式管理测试用例, 基于可视化的方式支持数据驱动, 利用关键字方式支持测试接口的上下文传递, 从而支持测试复杂的业务场景. 并且通过多种的测试用例重用方式, 简化了测试用例编写, 降低了案例维护开销. 测试用例执行模块化、分布式的策略, 在测试用例快速增长时, 可以方便地对测试用例执行机器进行扩展. 同时, 平台提供可视化配置方式支持版本测试用例定时执行, 以及丰富的日志和报表数据支持用例执行分析减少了漏测.

2　测试平台

Dubbo 接口测试平台 (新平台) 架构见图 1. 如图 1 所示, 数据流 (实线) 表示测试平台的不同模块, 数据流 (虚线) 表示测试平台与外部的关联数据和子系统. Dubbo 接口测试平台核心包括 6 个部分: 子系统管理、权限管理、接口管理、测试用例管理、测试用例执行器、工具类管理. 子系统管理展示了待测的子系统的详细信息, 包括所属测试团队、对应 IP 地址、部署的 jar/war 包信息、系统负责人等相关信息. 权限管理主要对测试人员访问子系统的权限进行设置, 一个测试人员可以属于多个测试组, 一个测试组包含多个子系统, 只有加入了测试组才能对该测试组下的子系统用例进行增、删、改操作. 接口管理展示了待测子

系统的所有 Dubbo 接口列表, 在测试之前, 首先上传待测试系统的 jar/war 包, 根据 Dubbo 协议解析出对应接口列表, 包括包名、方法名、请求参数、返回参数; 除了自动解析的接口, 还可以手动添加接口, 根据接口是否发生变更, 状态可分为新增及更新, 如果在测试过程中, 接口发生了变更, 则状态由新增变更为更新. 测试用例管理和测试用例执行器主要是测试用例的增删、改查及执行. 工具类管理主要是对外部封装的工具类进行管理, 展示工具类的方法列表、参数及调用方式. 新平台的核心是输出日志和执行数据, 并在数据分析模块分析处理数据并生成报表.

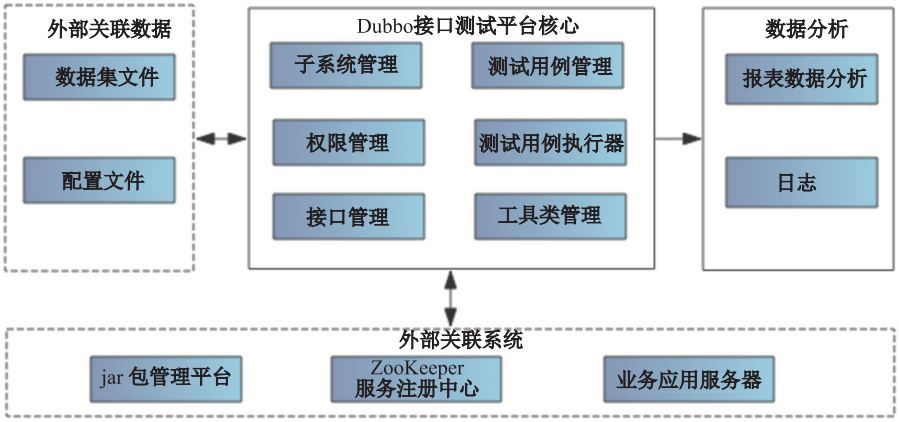

图 1 Dubbo 接口测试平台架构

Fig. 1 Architecture of the Dubbo interface testing platform

与外部数据相关的两个模块为外部关联数据和外部关联系统. 外部关联数据模块包括数据集文件和配置文件. 基于数据驱动的思想, 一个测试用例可以对应多条测试数据, 测试人员可以在 excel 编辑需要的测试数据集. 测试用例中的数据参数化支持, 除了可以使用数据集的数据, 还可以使用配置文件中的数据. 数据集和配置文件的数据都通过 Dubbo 接口测试平台核心模块进行导入和导出. 外部关联系统主要包括 jar 包管理平台、ZooKeeper 服务注册中心、业务应用服务器. jar 包管理平台通过 maven 私服搭建, 测试文件都存储在平台上, 通过配置 maven 的 GAV(GroupId, ArtifactId, Version) 信息, 新平台核心模块自动下载待测包. ZooKeeper 服务注册中心通过 ZooKeeper 统一管理业务服务. 接口测试平台核心模块通过 ZooKeeper 服务中心获取业务服务列表并发送服务调用请求至业务应用服务器, 并获取服务响应数据.

2.1 用例管理

用例管理模块的架构图如图 2 所示. 用例管理中, 测试用例以子系统分类, 分别对应不同的权限, 包括用例的新、增、改及执行等权限. 接口列表、配置信息、工具类都是子系统级. 根据待测试功能进行分类, 一个子系统的测试用例分为多个场景, 一个场景下又有多条测试用例.

在用例编写中, 一部分用例需要准备或清理相同的数据来保证用例的正确执行, 场景前置和后置脚本, 可以保证批量用例的环境准备, 避免用例编写中的重复工作.

通过初始化原子用例生成模块, 测试人员可以选择需要测试的接口列表批量生成测试用例和对应接口步骤. 随着版本迭代, 一个子系统会对应多个版本, 测试人员可以通过页面自动切换版本来查看对应的测试信息.

一个用例包含前置脚本、后置脚本、测试步骤和执行策略. 测试步骤分为接口步骤和脚本步骤. 一个接口步骤包括 6 部分: 前置脚本、请求报文、返回报文、异常信息、后置脚本和断言. 脚本步骤仅支持脚本编写. 创建接口步骤时, 选择待测试的接口, 此模块会展示对应的请求及返回的参数, 并自动根据参数类型填入默认值. 测试人员需要做的是准备测试数据, 发送请求, 并通过返回自动添加验证信息. 脚本步骤主要是在步骤级别编写脚本, 方便测试人员合理设计自动化用例. 一个测试用例需要包含多个测试步骤来完成单个功能点的测试. 测试用例的执行策略分为两种: 一种是如果一个步骤执行失败, 后续步骤停止执行; 另一种是即使前面步骤执行失败, 后续步骤也需要继续执行, 类似 TestNG 的 softAssert 操作. 此模块通过添加开关来设置步骤失败执行策略, 默认为关闭, 即失败后停止, 以避免不必要的执行耗时.

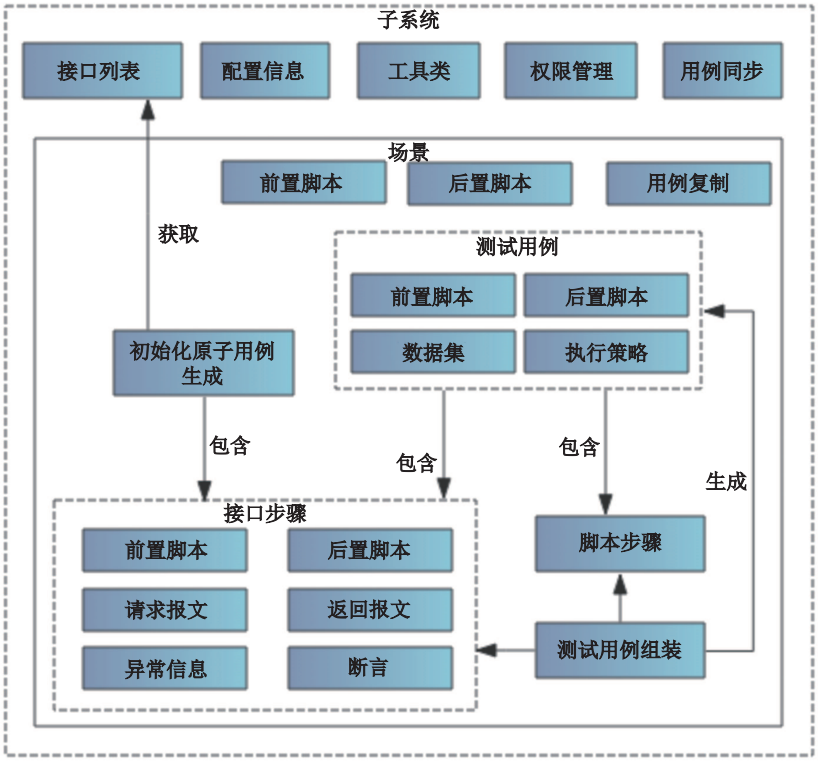

图 2　用例管理架构图

Fig. 2　Architecture of case management

新平台基于数据驱动, 主要体现在 3 个方面: 配置信息、数据集和工具类. 对于不同的测试环境, 配置项管理通过键值对方式支持变量配置, 同一个 key, 可以对应多个 value 值, 根据环境配置来区分此配置项在哪个环境测试环境生效. 可以在脚本中把配置项作为变量使用, 来编写不同场景的测试用例及验证. 对于同一个接口, 不用数据对应不同的场景及验证点, 在编写测试用例时会需要多条测试数据. 此模块提供数据集的功能, 可以通过 excel 方式批量导入测试数据, 并展示为 Json[15]格式, 可以在测试脚本中把 key 值作为变量进行使用. 新平台同时提供 Web 页面方式, 支持对 Json 数据的新、增、改, 方便测试数据的使用. 对于外部封装的工具类, 可以直接以工具类的方式上传到平台, 然后添加需要的工具类作为待测子系统的依赖子系统. 工具类中的方法可以直接在脚本中使用或者作为接口来使用.

在单版本测试过程中, 开发代码有缺陷会导致接口及代码变更. 本文设计的新平台会检测 jar/war 包变更, 自动发现接口变更, 并在接口中标记此变更. 在执行中, 可以实时看到执行日志, 方便查看用例的执行情况及测试流程数据是否正确.

在此模块中定义了一些关键字来辅助测试用例的编写. 关键字如表 1 所示.

表 1 用例关键字

Tab. 1 Keywords of cases

用例关键字	说明
$app.config(key)	获取配置项 key 的值
$app.globalConfig(key)	获取系统级配置项 key 的值
$app.request.key	获取请求报文中 key 的值
$app.content.key	获取返回报文中 key 的值
$app.addContext(key,value)	把 key 及 value 值加入上下文
$app.getContext(key)	通过 key, 把对应的 value 值从上下文中取出
dataSet[key]	通过 key, 取出数据集中对应的 value 值
$app.assert(key,value)	通过 key, 验证期望的是 value 值

通过关键字 $app.addContext(key,value) 和 $app.getContext(key), 测试人员可以方便地实现步骤间数据的传递, 从而快速编写多接口测试用例.

在编写测试用例过程中, 部分功能点的测试是多个接口的不同组合, 此模块提供了测试用例及相关步骤复制的功能. 测试用例的复制分为 3 个级别: 子系统级别、测试用例级别和测试步骤级别. 用例同步模块是指开发迭代过程中, 新的待测版本生成, 测试人员可以通过Web页面选择从哪个旧的版本中复制哪些测试用例到新的版本, 这是子系统级别的用例复制. 对于测试用例级别, 测试人员可以通过可视化方式, 复制测试用例到不同的场景, 并通过修改用例名称和新增、修改测试步骤来生成新的测试用例. 测试步骤创建后, 所有的步骤会显示在案例管理的左侧, 可以通过拖拽方式复制测试步骤来组装新的测试用例.

2.2 新平台技术架构及多系统多版本用例执行的负载均衡策略

此模块主要介绍新平台的技术架构和多子系统多版本下的负载均衡策略. 新平台技术架构如图 3 所示.

测试人员通过前端控制台界面对用例进行操作, 新平台通过 NGINX[22] 实现 Web 请求的分发, NGINX 负责 Web 层的负载均衡. Web 应用以集群形式部署在多个 tomcat 上, 主要负责可视化的编写与调试. 用例的执行是在 work 测试执行器上执行, 新平台中 work 测试执行器也是以集群方式部署. Web 服务和 work 服务都注册在 Eureka[23] 上. 针对高并发及可能出现的服务异常及子系统、版本个数及自动化测试用例的日益增加, 新平台对测试用例执行实现了自定义的负载均衡策略, 这部分包含在 Web 中的执行负载均衡模块.

由于接口测试用例数量也很多, 同一时间并发请求数也会很多, 这对缓存访问有很高的要求; 并且根据执行的负载均衡策略, 同一个应用的测试用例会分配在同一台 work 执行器上, 新平台会把测试用例执行的上下文信息存储在本地缓存服务 Ehcache[24] 上, 以提高执行速度. 而用户登陆信息及执行中使用到的 Dubbo 服务信息是存储在缓存服务器 Redis[25] 上, 并根据负载均衡策略从 work 执行器集群中选择一台进行执行, work 执行器从 ZooKeeper 注册中心选择可用的业务服务器, 发请求给业务服务并获取响应结果. 测试用例的数据保存在数据库中, 数据库使用 MySQL, 一主一从. 新平台中所有文件都保存在分布式文件服务器 NFS 上.

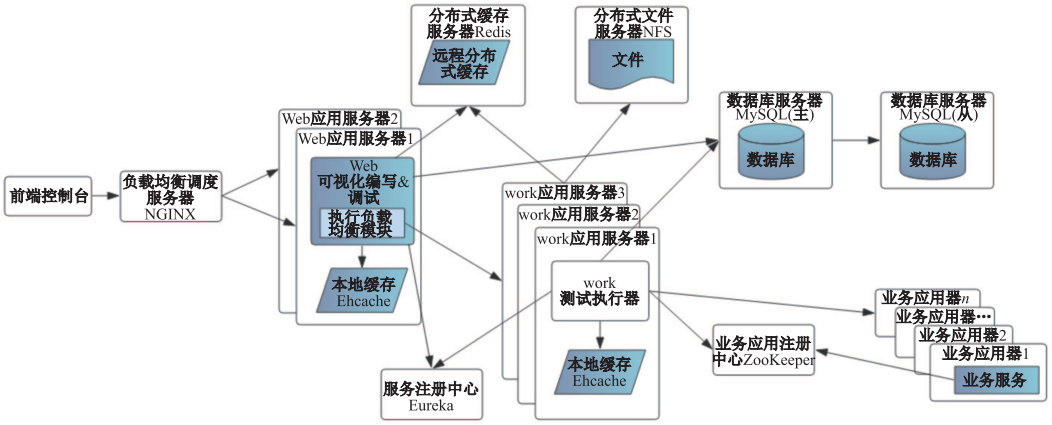

图 3　Dubbo 接口测试平台技术架构图

Fig. 3　The technical architecture of the Dubbo interface testing platform

执行负载均衡模块中的负载均衡策略基于 spring cloud round Ribbon[26], 自定义了负载均衡规则 HashRule, 此类继承自 AbstractLoadBalancerRule. RequestContext 类定义了子系统运行的线程上下文, 可以获取当前子系统的标识性信息 appId. 选取执行 work 的规则算法伪代码见算法 1.

算法 1　负载均衡规则算法

输入: 负载均衡对象 ILoadBalancer lb
输出: 选取的执行 server

```
1: hashKey ← 从线程上下文中获取子系统对应版本的唯一标志 appId
2: while true
3:       do servers.clear();//servers 保存负载均衡需要选取的 work(server)
4:       if server = null 且count++ < 10 then //try ten times
5:             reachableServers ← lb.getReachableServers();//reachableServers's
6: type is List which saves the normal work instances
7:             allServers ← lb.getAllServers();//通过负载均衡对象获取所有 servers
8:             for server1 in allServers
9:                   do o ← 从缓存中获取不满足要求的 server 列表
10:                      If o=null then
11:                            servers.add(server1);//把满足要求的 server1 存入 servers
12:                  end
13:             upCount ← reachableServers.size();//可以使用的 servers 大小
14:             serverCount ← servers.size();
15:             if upCount !=0 且serverCount != 0 then
16:                  servers.sort();//根据 appId 对 servers 进行排序
17:                  index ← Integer.valueOf(HashKey) % serverCount;//选择 server
18:                  server ← servers.get(index);
19:             if server = null then
20:                  Thread.yield();
21:             elseif
22:                  再次从缓存中获取不满足要求的 servers 列表!=null
23:                        Continue;
24:                  if server.isAlive() 且 server.isReadyToServe() then // 判断 server 可用
25:                        return server;
26:                  else server ← null;
27:             else continue;
28:             else return null;
29: end
```

负载均衡算法为从线程上下文中获取待测子系统对应版本的 appId, 找到正常工作, 并且内存、磁盘空间及 CPU 都没有超负荷运行的 work 实例列表个数, 对 appId 做取模操作, 得到分发的机器索引, 并拿到对应的 work 实例. 再次判断 work 实例没有在缓存非不宜使用列表中, 并且服务是正常的, 则返回此 work 实例. hashKey 为从线程上下文中获取的子系统的 appId, allServers、reachableServers 为从 Eureka 中读取到的所有的及可用的 work 服务实例. 根据各 work 机器信息判断哪些机器信息指标信息超负载, 加入缓存列表中. servers 保存所有可用的且满足机器资源要求的 work 服务实例. 针对单个 appId 选取的 work 实例保存在 server 中. 在获取 server 的过程中, 尝试 10 次, 如果还是没有找到对应的 work 实例, 则返回 null.

2.3 用例执行

此模块包括获取执行上下文及执行用例. 执行维度主要分为 4 种方式: 子系统级别、场景级别、用例级别和步骤级别. 用例执行架构如图 4 所示.

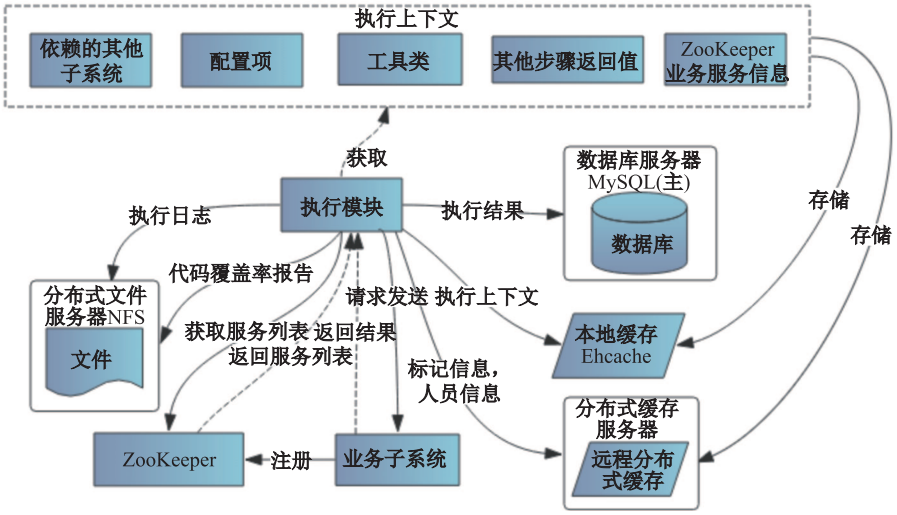

图 4 用例执行架构图

Fig. 4 Architecture of test case execution

在执行用例之前, 需要获取执行的准备信息, 即执行上下文信息, 包括依赖的其他子系统、配置项信息、使用的工具类中的方法、其他步骤的返回值及 ZooKeeper 中注册的 Dubbo 服务信息. 针对 Dubbo 请求, 需要从 ZooKeeper 中获取注册的服务信息, 有了这些信息, 用例才能执行. 由于自动化测试用例数量很多, 同一时间并发请求数也会很多, 则执行上下文会存储在本地缓存服务 Ehcache 上, 执行中用到的标记信息及测试人员登陆信息存储在缓存服务器 Redis 上. 用例执行会发请求给业务子系统, 然后根据执行情况生成对应的日志, 执行日志及生成的代码覆盖率报告存储在 NFS 服务器上. 对于执行结果, 包括返回报文和执行成功率及代码覆盖率数据会存储在数据库中. 可以通过手动触发或通过定时调度方式来执行单个子系统单个版本的所有测试用例, 并生成日志报告及执行报告和代码覆盖率报告. 用例执行报告包括测试用例总数、测试步骤数、测试用例成功数、成功率、失败数、失败原因分析和失败用例详情. 可以通过页面方式点击方式来查看失败的测试用例及失败详情, 并可定位到失败的步骤. 测试用例级别的执行会同时执行测试用例下的所有步骤, 并生成日志. 测试步骤级别的执行支持单个步骤的执行, 并支持步骤间上下文的串联. 在调试测试用例过程中, 可以仅执行单个步骤来查看结果, 而不用执行所有的步骤, 极大地节省了测试用例的调试时间.

在执行过程中, 可以对测试步骤、测试用例和测试场景进行标记禁用, 有此标记的测试用

例在各级别的执行中都会被略过, 并且在报表中不会被统计.

覆盖率报告展示了自动化测试用例对于开发代码的覆盖情况. 从报告中, 可以看到哪些代码被覆盖, 哪些没有覆盖, 并可以通过 Dubbo 服务接口关键字搜索到相关的类文件来查看相关接口的覆盖情况, 从而帮助测试人员更好地设计自动化测试用例, 避免漏测.

2.4　用例编写示例

编写示例为两个系统之间的测试串联, 主要涉及两个子系统 A、B 和工具类 tool-util. 如果 A 系统需要调用 B 系统的 Dubbo 接口 bizOrderAiFacade 的方法 bizOrder, 用于落业务单操作并使用工具类的方法打印自定义日志, 则 A 系统需要添加 B 系统和工具类 tool-util 作为依赖. 由于开发迭代, 一个子系统或工具类会对应多个版本, 需要选择选择依赖的子系统及对应的版本进行添加, 依赖关系可视化页面展示如图 5 所示.

图 5　依赖系统信息

Fig. 5　Information on Dependent Systems

编写测试用例. 测试步骤为落业务单、支付、交易通知. 第一步, 创建接口步骤, 选择接口方法 bizOrderAiFacade 的方法 bizOrder, 并使用变量的形式给参数赋值; 第二步, 选择 A 系统的接口 TransAiFacade 中方法 payOrder 做支付操作; 第三步, 选择接口 MessageNotifyAiFacade 的方法 payNotify 做交易通知. 在每个步骤编写完之后, 可以执行单个步骤或整个测试用例并获取执行结果. 界面展示如图 6 所示.

图 6　包含 3 个步骤的测试用例

Fig. 6　Test case with three test steps

单个测试用例包括用例名称、测试用例属于哪个场景、测试用例步骤数、用例优先级、权重、执行时间的执行、修改、删除、用例前置后置、数据集及日志查看. 图 6 的最左侧展示的是已有的步骤. 编写测试用例可以通过拖拽的方式来复制测试步骤.

以第二个步骤 payOrder: TransAiFacade (方法名: 接口名) 查看单个步骤, 页面展示如图 7 所示.

测试步骤包括前置脚本 (BeforeMethod)、后置脚本 (AfterMethod), 脚本里可以使用工具类的方法, 比如日志打印. 接口请求参数及返回数据. 请求参数的赋值可以参数化: idempotentFlag 通过配置项的方式赋值; paymentId 通过上下文的方式赋值. 断言可以通过 3 种方式编写: 添加断言 (添加单条验证点)、批量添加断言、脚本方式验证. 点击运行, 获取测试用例执行结果.

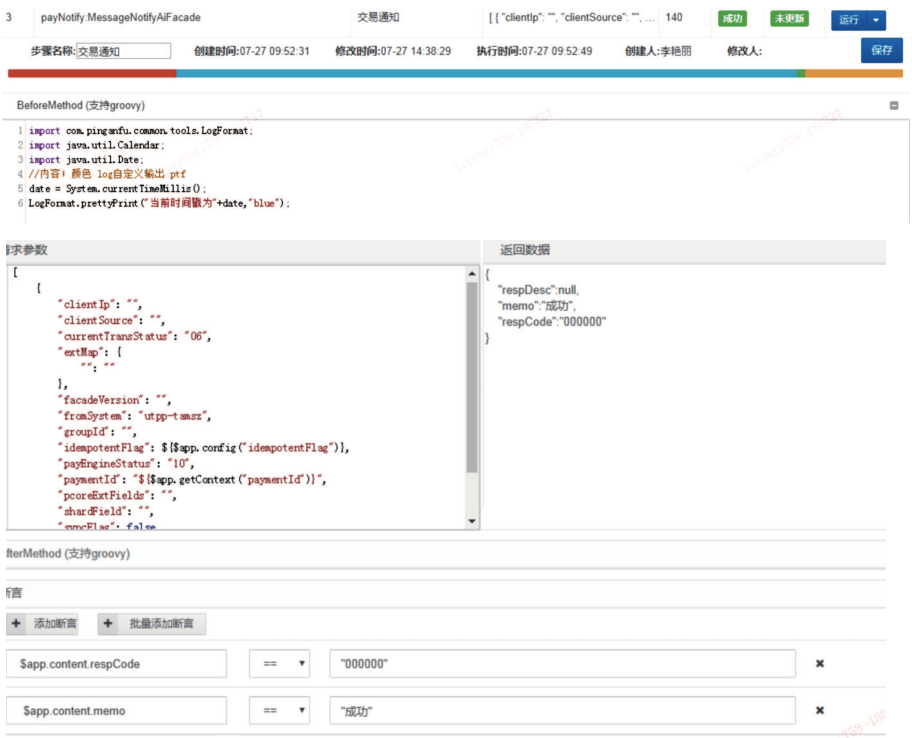

图 7 测试步骤示例

Fig. 7 Example of One Test Step

3 实 验

3.1 实验环境

实验环境共有 9 台 RedHat 6.5 的测试机器: 1 台部署 NGINX 和 Redis; 2 台部署 Web; 3 台部署测试用例执行器; 1 台部署 MySQL 主库; 1 台部署备库; 1 台部署 NFS. 3 台测试用例执行器配置为 8 个主频为 2.4 GHz 的 Intel Skylake CPU 芯片, 内存为 16 GB, 磁盘容量为 80 GB; 2 台 Web 配置为 2 个主频为 2.4 GHz 的 Intel Broadwell CPU 芯片, 内存为 4 GB, 磁盘容量为 80 GB; NGINX 和 NFS 机器配置分别为 2 个主频为 2.4 GHz 的 Intel Broadwell CPU 芯片, 内存为 8 GB, 磁盘容量为 80 GB; 1 台 MySQL 主库为 16 个主频为 2.4 GHz 的 CPU E5-2695 芯片, 内存为 32 GB, 磁盘容量为 250 GB; 老框架运行环境为 Windows 7, 配置为 1 个主频为 2.5 GHz 的 Intel i3-3120M CPU, 内存为 8 GB, 磁盘容量为 200 GB.

3.2 新平台与老框架耗时对比数据

表 2 为新平台与老框架的耗时数据, 使用第 2.4 节中的用例编写示例. 从表 2 的数据可以看出, 平台化后, 测试人员不用花费时间搭建开发环境来编写测试用例, 投入到自动化测试的人数越多, 节省的时间越多. 由于新平台会解析接口参数, 并且能自动生成单接口测试用例, 测试人员只需要对请求参数赋值, 并把赋值参数化以适配多环境, 大大节省了单步骤测试用例准备及多环境适配的时间.

由于老框架同一个接口方法的请求参数代码只有 1 份, 在处理复杂场景时需要处理多种情况的串联, 耗时较多. 新平台对于多环境适配, 需要在配置文件中, 对于相同的键, 新增 value 值, 相对耗时较少. 对于测试结果, 新平台提供了丰富的日志及执行策略, 失败原因分析可以快速定位问题, 相比原来的查看 jenkins 批量执行日志, 单个子系统平均几百个用例, 节省的时间是按小

时计算的. 总地来说, 可视化的新平台在用例编写、执行及结果分析上节省了5倍时间. 用例数越多, 参与自动化的人员越多, 节省发的时间越多.

表 2　新旧耗时对比数据

Tab. 2　Cost vs. time comparison between the old and new framework

	老框架	新平台
准备测试环境	4 h/人	0
准备测试用例 (单步骤)	3 min/个	1 min/个
执行测试用例 (单步骤)	1 min/个	0.3 s/个
准备测试用例 (复杂场景)	20 min/个	3 min/个
多环境适配 (单步骤)	4 min/个	1.3 min/个
多环境适配 (复杂场景)	20 min/个	3 min/个
分析测试结果	2 h/子系统	1 h/子系统
汇总	6.8 h	1.2 h

3.3　新平台与老框架自动化接口测试用例增长对比数据

图 8 展示了新平台与老框架按月统计 (1 月份—6 月份) 的测试用例增长数据, 每个月月底统计当月的测试用例增长情况. 从数据可以看出, 老框架的测试用例数据为以百位增长, 新平台的自动化用例则以千位增长. 每个月的数据增长有变化, 主要是由功能测试人员的时间决定, 如果功能测试任务较重, 则花费在自动化上的时间会减少. 新平台门槛相对较低, 测试人员在功能测试的同时, 可以基于新平台编写自动化用例进行测试.

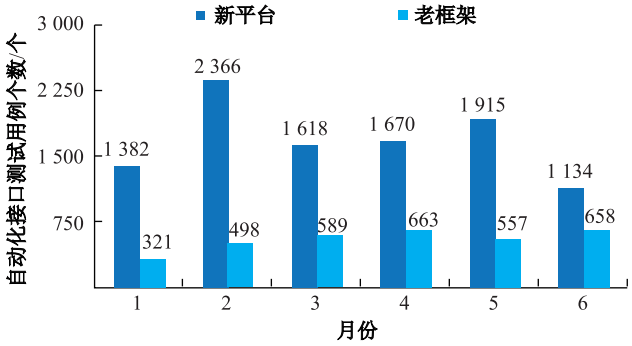

图 8　新旧自动化接口测试用例增长对比数据

Fig. 8　Automated interfaces test case growth comparison between the old and new framework

3.4　子系统执行数据

目前新平台子系统个数有 247 个, 执行的用例数为 16 271 个, 总体成功率为 79.2%, 平均行覆盖率 30%. 日常自动化数据选取当日测试用例总数最高的 10 个子系统, 用例执行数据如表 3 所示.

执行耗时包含了接口实际调用时间 (请求发送到业务子系统到请求返回), 访问数据库时间及脚本中设定的 Sleep 时间. 批量执行时间都控制在 30 min 左右.

对于平台的扩展性, 根据执行的负载均衡算法, work 机器实例增加, 算法会动态获取可用的 work 实例进行 Hash, 从而达到弹性扩容的目的.

表 3 用例日常执行数据

Tab. 3 Execution records of daily cases

子系统-版本	TC 总数	TC 成功率/%	步骤总数	行覆盖率/%	执行耗时
1-20180510	1 267	64.40	6852	46.22	81 min
2-20180510	773	96.64	1 744	39.89	8 min35 s
3-20180510	773	66.24	1 422	11.36	8 min20 s
4-20180510	700	76.88	2 122	22.32	37 min16 s
5-20180510	550	84.24	563	31.95	7 min42 s
6-20180510	504	93.45	523	35.24	1 min43 s
7-20180510	451	94.92	696	45.22	6 min58 s
8-20180510	419	71.26	432	25.16	7 min17 s
9-20180510	378	88.62	2 203	27.53	23 min49 s
10-20180510	336	97.24	420	47.90	2 min29 s

4 结论与展望

为了更好地测试互联网金融使用Dubbo协议的应用, 快速编写并管理大批量用例及更好地支持多子系统版本迭代和测试用例数据分析, 本文使用分布式技术设计并实现了一个灵活的Dubbo接口自动化测试平台. 首先介绍了测试平台的架构; 然后着重介绍了用例管理和用例执行, 并介绍了多子系统多版本用例的并发执行及执行中使用的负载均衡策略, 最后介绍了日常自动化执行情况、用例及步骤耗时, 并以一个具体例子说明测了试用例的编写. 虽然新平台降低了自动化门槛, 大大节省了用例编写时间, 但批量执行单个子系统的测试用例时还需要考虑并发执行, 进一步减少执行时间.

[参 考 文 献]

[1] 席涛, 郑贤强. 大数据时代互联网产品的迭代创新设计方法研究 [J]. 包装工程, 2016, 37(8): 1-4.

[2] 周永红, 张彦祥. 金融软件的自动化测试探索与创新之路 [J]. 中国金融电脑, 2018(1): 64-68.

[3] TAY B H, ANANDA A L. A survey of remote procedure calls [J]. ACM SIGOPS Operating Systems Review, 1990, 24(3): 68-79.

[4] APACHE SOFTWARE FOUNDATION. Apache Dubbo [OB/OL]. [2018-06-20]. http://Dubbo.apache.org/.

[5] SoapUI. Available [EB/OL]. (2017-12-04)[2018-07-01]. https:// www.soapui.org/.

[6] SOAP. WWW, Service Architecture, Soap [EB/OL]. (2016-07-17)[2018-07-01]. https://www.service-architecture.com/articles/web-services/soap.html.

[7] 杨超. 基于分布式服务框架 Dubbo 的集群式服务器的研究与实现 [D]. 北京: 北京邮电大学, 2017.

[8] Apache JMeter^{TM}. The Apache Software Foundation [EB/OL]. (2016-07-17) [2018-07-01]. http://jmeter.apache.org/index.html.

[9] MICRO FOCUS. LoadRunner Load Testing Software [EB/OL]. (2017-08-30)[2018-07-01]. https://www.microfocus.com/en-us/products/loadrunner-load-testing/overview.

[10] NIELSEN H F, GETTYS J, BAIRD-SMITH A, et al. Network performance effects of HTTP/1.1 [J]. ACM Sigcomm Computer Communication Review, 1997, 27(4): 155-166.

[11] MENG F Y. Phoenix Framewor [EB/OL]. (2016-07-17)[2018-07-01]. http://www.cewan.la/.

[12] SHANG Y, ZHANG X L, FENG Y B, et al. Research and application of automated testing tool based on STAF [C]// Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering. Paris: Atlantis Press, 2013: 2336-2339.

[13] SOURCE FORGE. Software Testing Automation Framework (STAF) [EB/OL]. (2016-12-31)[2018-07-01]. http://staf.sourceforge.net/index.php.

[14] HE Z H, ZHANG X, ZHU X Y. Design and implementation of automation testing framework based on keyword driven [J]. Applied Mechanics and Materials, 2014, 602/603/604/605: 2142-2146.

(下转第 143 页)

[31] 张思, 宁国辉, 杨铮铮, 等. 复合填料土壤渗滤系统处理农村生活污水的效果 [J]. 环境工程学报, 2014(11): 4625-4630.
[32] CHEN C, ZHANG R, WANG L, et al. Removal of nitrogen from wastewater with perennial ryegrass/artificial aquatic mats biofilm combined system [J]. Journal of Environmental Sciences, 2013, 25(4): 670-676.
[33] CHEN Y, ZHAO Z, PENG Y, et al. Performance of a full-scale modified anaerobic/anoxic/oxic process: High-throughput sequence analysis of its microbial structures and their community functions [J]. Bioresource Technology, 2016, 220: 225-232.
[34] ZHONG F, WU J, DAI Y, et al. Bacterial community analysis by PCR-DGGE and 454-pyrosequencing of horizontal subsurface flow constructed wetlands with front aeration [J]. Applied Microbiology and Biotechnology, 2015, 99(3): 1499-1512.
[35] 孙振丽, 宣引明, 张皓, 等. 南美白对虾养殖环境及其肠道细菌多样性分析 [J]. 中国水产科学, 2016(3): 594-605.
[36] 周明明, 李晓雁, 陈悦, 等. 鞘氨醇单胞菌 TP-3 原生质体制备与再生的研究 [J]. 食品工业科技, 2015(22): 184-188.
[37] 宗炯, 朱雪竹, 凌婉婷, 等. 多环芳烃污染对丛枝菌根真菌生物学性状的影响 [J]. 农业环境科学学报, 2014(2): 305-312.
[38] HE Y, ZHOU G, ZHAO Y. Nitrification with high nitrite accumulation for the treatment of “Old” landfill leachates [J]. Environmental Engineering Science, 2007, 24(8): 1084-1094.

(责任编辑: 张　晶)

(上接第 132 页)

[15] ECMA INTERNATIONAL. The JSON Data Interchange Syntax[EB/OL]. (2016-12-31)[2018-07-01]. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.
[16] JOY J, SINGH D P. A generic framework design to enhance capabilities of an enterprise test automation framework [C]// International Conference on Applied and Theoretical Computing and Communication Technology. IEEE, 2016: 207-212.
[17] GRIFFIN L, RYAN K, DE LEASTAR E, et al. Scaling instant messaging communiction services: A comparison of blocking and non-blocking techniques [C]// 2011 IEEE Symposium on Computers and Communications. IEEE, 2011: 550-557.
[18] JUNQUEIRA F, REED B. ZooKeeper: Distributed Process Coordination [M]. [S.l]: O’Reilly Media, Inc. 2013.
[19] BURROWS M. The Chubby lock service for loosely-coupled distributed systems [C]// OSDI’06: 7th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, 2006: 335-350.
[20] MAVEN. The Apache Software Foundation [EB/OL]. [2018-07-01]. https://maven.apache.org/what-is-maven.html.
[21] TestNG. The Apache Software Foundation [EB/OL]. (2017-12-19)[2018-07-01]. http://testng.org/doc/documentation-main.html.
[22] REESE W. Nginx: The high-performance web server and reverse proxy [J]. Linux Journal, 2008, 2008(173): 2.
[23] GITHUB. Netflix, Eureka [EB/OL]. [2018-07-01]. https://github.com/Netflix/eureka.
[24] EHCACHE. Software AG USA [EB/OL]. (2017-05-25)[2018-07-01]. http://www.ehcache.org/about/index.html.
[25] REDIS. Redis Labs [EB/OL]. (2017-07-00)[2018-07-01]. https://redis.io/topics/introduction.
[26] Netflix, Ribbon [EB/OL]. (2018-04-28)[2018-07-01]. https://github.com/Netflix/ribbon.

(责任编辑: 李　艺)