



南京大學

研究生畢業論文
(申請工程碩士學位)

論文題目 基于 **Docker** 的彈性資源池性能測試系統

作者姓名 楊郁芬

學科、專業名稱 工程碩士（軟件工程領域）

研究方向 軟件工程

指導教師 陳振宇教授

2022 年 5 月 23 日

学 号：MF20320201

论文答辩日期：2022 年 05 月 20 日

指 导 教 师： (签字)

(English Title of Thesis)

by

Yuqin Yang

Supervised by

Professor Zhenyu Chen

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER OF ENGINEERING
in
Software Engineering



Software Institute
Nanjing University

May 23, 2022

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： 基于 Docker 的弹性资源池性能测试系统

工程硕士（软件工程领域） 专业 2020 级硕士生姓名： 杨郁芩
指导教师（姓名、职称）： 陈振宇教授

摘 要

在互联网飞速发展的当下，诞生了桌面、移动、Web 等多种多样的软件应用，这些应用均依赖于服务器提供的接口来传输数据。随着互联网用户数量的增长，软件应用需要去承受更高的流量，软件性能成为了软件开发方无法忽视的问题，性能测试结果也已经成为软件质量评估的重要指标之一。在使用单机执行性能测试时，测试机器将会受限于服务器 CPU、内存等硬件配置，无法完成大规模的性能测试。若使用多台测试机进行测试，不同机器的硬件配置不同将会导致其负载量不同，需要进行负载均衡配置。为了突破单机测试的限制、更好地开展性能测试工作，需要一个性能测试系统来对测试资源池进行配置和管理，并为用户提供测试资源池各节点在测试过程中的软硬件监测数据。

基于上述背景，本文需要实现一个打通性能测试过程、支持多机器“分布式”测试、支持资源池负载均衡配置、支持资源池开销监控的性能测试系统。为实现上述目标，本文设计并完成了基于 Docker 的弹性资源池性能测试系统。此系统包含 5 个模块，分别是接口场景定义、测试执行、测试资源管理、测试数据处理、测试报告生成。测试人员能够自定义或导入测试接口，将测试接口组合为测试场景，基于已定义的测试场景执行性能测试。在测试资源管理模块中，测试人员能够配置多个测试机，当使用配置的测试资源池执行性能测试时，系统中的节点容器管理服务将会在测试机中下载用于测试执行的 Docker 镜像，并启动 Docker 容器。测试执行时，由调度服务将请求下发至各个测试机来执行测试。测试过程中，利用 Kafka 从各个测试机中获取测试结果，将测试结果进行清洗与汇总。与此同时，测试报告生成模块从 Kafka 中读取并整合数据，将特定的指标或图表展示给用户。测试人员能够根据测试报告中不同资源池的测试表现，评估软件性能，进一步提升软件质量。

本系统实现了性能测试全场景的功能，并为测试人员提供了支持大规模流量测试的弹性资源池配置功能。目前本系统已经成功上线并投入使用，为了对接国内大力发展的信创产业，本系统适配了国产操作系统，并测试了不同操作系统中软件正确性与一致性。

关键词： 软件测试，性能测试，Docker

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Performance Test System for Scalable Testing Resources
Based on Docker
SPECIALIZATION: Software Engineering
POSTGRADUATE: Yubin Yang
MENTOR: Professor Zhenyu Chen

Abstract

Currently, with the rapid development of the Internet, varieties of software application have been born, such as desktop, mobile, and Web, which rely on the interface provided by the server to transmit data. With the growing number of Internet users, those applications need to withstand higher network traffic. Therefore, software performance has become a grim problem that developers cannot ignore, and performance test results have become one of the important indicators of software quality evaluation. However, cause by hardware configuration such as CPU and bandwidth, single test machine will be limited in large-scale traffic scenarios, and unable to complete the performance test of large-scale traffic. If multiple test machines are used for testing, the hardware configuration of different machines will lead to different loads. So we need a system to support load balancing settings for each resource node and provide software and hardware monitoring data during performance testing for each node.

Based on the background, this paper needs to implement a performance testing system, which covers the whole performance testing process, supports multi-machine "distributed" testing, supports resource pool load balancing configuration, and supports resource pool monitoring. In order to achieve the above objectives, this paper designs and completes a Docker-based scalable resource pool performance testing system. The system contains five modules: api and scenario definition module, test execution module, resource pool management module, test result processing module, test report generation module. Testers can customize or import test api and scenario, and then create performance test based on defined scenario. In resource pool management module, tester can configure multiple test machines. When using the configured resource pool

to perform performance test, the node container management service will download the Docker image for test execution in the test machine, and then start the Docker container. while performance test is executing, api request will be sent to each test machine to execute test and collect results. During this process, using Kafka to obtain test results from each test machine, those data will be cleaned and summarized by data processing module. At the same time, the test report generation module presents specific indicators or charts to user. Finally, testers can evaluate software performance based on the performance report of different resource pools, and improve software quality in the future.

This system implements the whole process of performance test that include test scenario orchestration, test load configuration, test execution, and test report display. In addition, system provides scalable resource pool configuration that support large-scale traffic testing. At present, this system has been successfully launched, in order to dock with the vigorous development of the Xinchuang' s domestic software, the system adapted to the domestic operating system, and tested the correctness and consistency of the software in different operating systems.

keywords: Software Testing, Performance Testing, Docker

目 录

目 录	v
插图清单	ix
附表清单	xi
代码清单	xii
第一章 引言	1
1.1 背景与研究意义	1
1.2 国内外研究现状与对比	3
1.3 本文的主要工作	5
1.4 本文的组织结构	6
第二章 相关技术概述	7
2.1 服务框架技术	7
2.1.1 Docker 容器技术	7
2.1.2 SpringBoot	8
2.1.3 Vue.js	9
2.1.4 ECharts 图表库	9
2.2 测试与监控技术	11
2.2.1 JMeter	11
2.2.2 Prometheus	11
2.3 数据存储与传输技术	12
2.3.1 MyBatis	12
2.3.2 Kafka	13
2.4 本章小结	14
第三章 需求分析与概要设计	15
3.1 系统整体概述	15
3.2 系统需求分析	16
3.2.1 系统涉众分析	16
3.2.2 功能性需求分析	18

3.2.3 非功能性需求分析	18
3.3 系统用例分析	19
3.3.1 系统用例图	19
3.3.2 系统用例描述	20
3.4 系统总体设计	25
3.4.1 系统架构总览	25
3.4.2 4+1 视图	27
3.4.3 多资源池性能测试流程设计	32
3.5 持久化模型设计	33
3.5.1 系统业务数据库设计	33
3.5.2 多资源池测试报告内容模型设计	35
3.6 本章小结	37
第四章 系统详细设计与实现	39
4.1 接口场景定义模块详细设计与实现	39
4.1.1 接口场景定义模块架构设计	39
4.1.2 接口定义与测试模块核心类设计	40
4.1.3 接口场景定义模块顺序图	42
4.1.4 接口场景定义模块关键代码	44
4.2 测试执行模块详细设计与实现	46
4.2.1 测试执行模块架构设计	46
4.2.2 测试执行模块核心类图	47
4.2.3 测试执行模块顺序图	49
4.2.4 测试执行模块关键代码	50
4.3 测试资源管理模块详细设计与实现	53
4.3.1 测试资源管理模块架构设计	53
4.3.2 测试资源管理模块核心类图	55
4.3.3 测试资源管理模块顺序图	57
4.3.4 测试资源管理模块关键代码	58
4.4 测试数据处理模块详细设计与实现	61
4.4.1 测试数据处理模块架构设计	61
4.4.2 测试数据处理模块核心类图	62
4.4.3 测试数据处理模块流程	64

目 录	vii
4.4.4 测试数据处理模块关键代码	65
4.5 测试报告生成模块详细设计与实现	68
4.5.1 测试报告生成模块架构设计	68
4.5.2 测试报告生成模块核心类图	70
4.5.3 测试报告生成模块顺序图	71
4.5.4 测试报告生成模块关键代码	72
4.6 系统实例展示	77
4.7 本章小结	81
第五章 系统测试	83
5.1 测试目标	83
5.2 测试环境	84
5.3 接口测试	84
5.4 功能测试	86
5.5 可移植性测试	90
5.6 负载测试	91
5.7 本章小结	93
第六章 总结与展望	95
6.1 总结	95
6.2 展望	96
致 谢	97
参考文献	99
简历与科研成果	105

插图清单

1-1	性能测试流程图.....	2
2-1	使用 Docker 搭建不同应用	7
2-2	MVVM 架构图	9
2-3	ECharts 架构图	10
2-4	ECharts Component 设计	10
2-5	prometheus 架构图	12
2-6	Kafka 架构图	13
3-1	系统流程图	16
3-2	系统用例图	20
3-3	系统整体架构图.....	25
3-4	系统逻辑视图	27
3-5	系统开发视图	29
3-6	系统进程视图	30
3-7	系统物理视图	31
3-8	多资源池性能测试流程图	32
4-1	接口场景定义模块架构图	39
4-2	接口场景定义模块类图	40
4-3	接口场景定义模块顺序图	43
4-4	测试执行模块架构图	46
4-5	测试执行模块——接口连通性测试执行类图	47
4-6	测试执行模块——弹性资源池性能测试执行类图	48
4-7	测试执行模块——接口连通性测试执行顺序图	49
4-8	测试执行模块——弹性资源池性能测试执行顺序图.....	49
4-9	测试资源池管理模块架构图.....	54
4-10	资源池管理模块类图	55

4-11 资源池管理模块——节点容器管理类图	56
4-12 资源池管理模块——节点容器管理顺序图	57
4-13 测试数据处理模块架构图	62
4-14 测试数据处理模块类图	63
4-15 测试数据处理模块顺序图	64
4-16 Metric 元数据处理流程图	65
4-17 测试报告生成模块整体架构图	68
4-18 测试报告生成模块前端组件结构图	69
4-19 测试报告生成模块类图	70
4-20 测试报告生成模块顺序图	72
4-21 测试接口定义页面截图	77
4-22 连通性测试页面截图	77
4-23 测试场景定义页面截图	78
4-24 测试资源池创建弹窗截图	78
4-25 测试资源池创建弹窗截图	79
4-26 测试报告数据概览页面截图	79
4-27 测试报告数据概览页面截图	80
4-28 测试报告请求统计页面截图	80
4-29 测试报告错误统计页面截图	80
4-30 测试报告监控详情页面截图	81
5-1 Postman 接口测试结果图	86
5-2 可移植性测试执行截图	91
5-3 查看测试报告场景负载测试监控数据图	92
5-4 本系统测试执行过程服务器监控数据图	92
5-5 JMeter 测试执行过程服务器监控数据图	92
5-6 本系统多节点测试执行过程主服务器监控数据图	93

附表清单

1-1 主流性能测试工具比较	4
3-1 系统涉众特征与期望分析结果	17
3-2 系统功能性需求分析	18
3-3 系统非功能性需求分析	19
3-4 系统用例与需求对应关系	19
3-5 定义测试接口用例描述	21
3-6 定义测试场景用例描述	22
3-7 创建测试任务用例描述	22
3-8 执行测试任务用例描述	23
3-10 查看测试报告用例描述	23
3-9 管理测试资源池用例描述	24
3-11 查看资源池报告用例描述	24
3-12 测试接口-Api 对象模型设计	33
3-13 测试场景-Scenario 对象模型设计	33
3-14 测试任务-Task 对象模型设计	34
3-15 测试资源池-ResourcePool 对象模型设计	34
3-16 测试节点-ResourceNode 对象模型设计	35
3-17 测试报告-Report 对象模型设计	35
3-18 报告结果-ReportResult 对象模型设计	36
3-19 图表数据-ChartData 结构设计	36
3-20 错误数据-ErrorData 结构设计	36
3-21 概览数据-OverviewData 结构设计	37
3-22 请求统计数据-RequestData 结构设计	37
4-1 HashTreeObj 对象结构设计	43
5-1 测试服务器硬件环境	84

5-2 测试服务器软件环境	85
5-3 接口测试用例设计与测试结果	85
5-4 定义测试接口功能测试用例	87
5-5 定义测试场景功能测试用例	87
5-6 创建测试任务功能测试用例	88
5-7 执行性能测试功能测试用例	88
5-8 管理测试资源池功能测试用例	89
5-9 查看测试报告功能测试用例	89
5-10 查看资源池报告功能测试用例	90
5-11 可移植性测试环境准备	90
5-12 可移植性测试结果	91

代码清单

4.1	Scenario Config 场景配置的 JSON 主要结构	41
4.2	创建测试场景核心代码	44
4.3	测试接口连通性核心代码	45
4.4	JMeterService 测试执行逻辑代码	50
4.5	LocalRunner 测试执行类代码	51
4.6	JMeterExecuteService 性能测试执行代码	52
4.7	测试执行线程任务代码	52
4.8	DockerEngine 容器启动类核心代码	58
4.9	ContainerService 节点容器创建代码	59
4.10	基于 Kafka 的测试结果数据消费者代码	66
4.11	测试数据处理模块核心业务代码	66
4.12	生成报告的入口方法 generateReport 代码	72
4.13	子报告——响应时间折线图代码	74
4.14	DataParse 类中 parseToChart 方法代码	75

第一章 引言

1.1 背景与研究意义

软件作为 20 世纪的一个新兴技术，从 20 世纪 60 年代编程语言出现开始，已发展了 50 余年。随着高级编程语言的不断出现与迭代，人们对软件的需求日渐增多，软件应用的复杂度也逐渐提升 [1]。当传统的软件开发方法不能继续满足软件需求、保障软件质量时，软件工程技术应运而生。软件工程是一门致力于设计、开发与使用软件的技术，也是对系统化、可量化方法进行实践与研究的学科 [2]。软件工程的基本目标是：生产出兼备正确性、可用性与合算性的软件产品 [3]。正确性指软件能够正确保质地向用户提供预期功能，可用性指软件的结构、实现与文档需要达到用户可用的程度，合算性指软件开发、运行中的开销满足用户需求，以上三个特点说明了保障软件质量是软件工程的基本目标之一，也是软件开发中不可忽视的一环。

世界范围内的软件工程领域中，从 20 世纪 70 年代起就开始了软件质量的研究，如今已经确定了许多相关的质量度量与标准，ISO9000 标准和 CMM 模型 (capability maturity model) 就是其中最具有代表性的成果。在软件成熟度模型中 [4]，将软件开发过程的成熟度等级分为初始级、可重复级、已定义级、已管理级和优化级，随着成熟度等级的提高，软件开发过程更为标准、可控，进而最终软件产品质量也将越来越高。这说明了软件产品质量很大程度上取决于软件开发与维护过程的质量，因此，如何在软件过程中度量软件质量、提升软件质量，成为了软件开发者与研究者共同关注的问题。

根据 GB/T16260.1 标准 [5]，软件质量可以用以下特性来评价：功能性、可靠性、易用性、效率、维护性、可移植性，其中可靠性描述的是在特定的一段时间与条件下，软件维持性能水平的能力。性能是软件的一种非功能特性，用于衡量软件系统在时间限制和资源分配方面的有效性 [6]，传统的性能指标包括响应时间、吞吐量和利用率。在不同的视角中软件性能的表现有所不同，在用户眼中，性能表现为软件加载时间的快慢；对管理员而言，更关注系统最多支持多少用户访问、系统最大业务处理量是多少、系统能否支持 7×24 小时的业

务访问等更高层的指标；而对开发者而言，需要关注软件在不同场景、高并发环境中的各个指标表现，以向用户提供稳定运行的软件。

Web 应用作为 21 世纪的主流应用，为其提供接口的后端 Web 服务也成为最重要的软件技术之一，也是电子商务中的一个有效解决方案 [7]。如今，除了在 Web 浏览器中调用 Web 服务，移动端、PC 端应用也能够通过中间层封装调用 Web 服务所提供的接口。随着 B/S 架构的应用日渐增多，流量也随之大规模增长，网站流量的快速增长导致一些常用网站由于过载而反应迟钝，当网站的性能缺陷达到无法忍受的水平时，将导致用户过早地终止他们在网站上的交易。对于商业网站来说用户流失意味着收入减少，最终带来严重的财务亏损 [8]。这些就是软件中隐藏的性能问题，引起这些问题的原因有系统运行环境不合理、Web 服务自身稳定性的欠缺 [9] 等。

为了对软件性能进行保障和验证，需要对软件性能展开测试 [10]，测试人员根据响应时间、吞吐量和可用性等指标来衡量软件应用性能 [11]。性能测试可以细分为压力测试与负载测试，压力测试侧重于系统或组件处理处于或超出其预期或指定工作负载限制的峰值负载的能力，负载测试侧重于系统自动处理由受控数量的并发用户或进程生成的事务请求导致的预期实际负载水平不断提高的能力 [12]。通常，性能测试由五个流程组成 [13]，如图 1-1 所示。由于软件性能的评判是建立在一定负载的基础上，聚集大量的测试员集中进行人工测试的成本开销巨大，因此，自动化性能测试成为了主流发展方向。



图 1-1: 性能测试流程图

在软件 20 世纪 80 年代中期，第一款自动化测试工具出现，至今已经诞生了 Apache JMeter、LoadRunner、PTS、Locust 等性能测试工具。这些工具大多覆盖了整个性能测试流程，从测试用例编排到测试报告生成，为用户提供了完整的性能测试服务。请求脚本录制、测试场景创建、测试参数设置、测试压力配置、测试报告生成是自动化测试工具中的核心功能。但 LoadRunner、PTS 等性能测试服务往往需要复杂的配置和大量的脚本编写 [14]，更适用于大规模的企业级专业测试工作，对专业性与前置知识有较高的要求。因此对于中小规模的测试团队来说，测试工具的易用性是需要考虑的因素之一。

当利用自动化测试工具执行性能测试任务时，测试机器将模拟用户向待测

软件发送 HTTP 请求，但发送请求需要消耗服务器 CPU 和内存，当单台机器上同时发送大规模请求时，可能会出现内存溢出甚至系统崩溃的情况。为了解决单机性能测试带来的限制，可以使用多台测试机器作为代理以分摊压力，由此向待测软件施加更大的负载。开发“分布式”的性能测试系统存在着以下几个问题：一是如何实现简单易用的“分布式”性能测试功能，以便于测试人员快速配置用于执行测试的资源池；二是资源池中的各个测试机之间如何通信，测试结果如何存储；三是测试报告中如何展示资源池中不同测试机的测试执行情况，从而给测试人员提供更全面的性能测试结果。

为解决上述问题，本文基于 Docker 设计并实现了一个弹性资源池性能测试系统，测试资源池是一至多个测试节点的组合，弹性意味着测试资源池中的节点能够被添加与删除，并对各个节点的并发数进行配置。测试人员依托弹性资源池可以进行小规模软件的性能测试，也能够通过添加测试机器进行大规模测试。

1.2 国内外研究现状与对比

1985 年，最早的自动化测试工具之一 AutoTester 问世，四年后 Mercury Interactive 公司以这款工具为基础发布了性能测试工具 LoadRunner，至今，LoadRunner 仍在不断迭代并且在主流工具中占据着一席之地。三十多年来，自动化测试从录制/回放的线性测试发展至智能化驱动测试框架，使用模块分解、关键字驱动等技术将越来越多的测试步骤自动化，以减少人工测试成本。

目前，支持 Web 服务性能测试、能够进行详尽测试编排、提供完整测试报告的主流自动化测试工具包括但不限于 JMeter、LoadRunner、Locust，本节将对这三种测试工具进行介绍与对比，分析不同工具的优劣势，下钻本系统的核心创新点。

JMeter 是 Apache 组织的利用 Java 开发的开源性能测试工具，能够支持 Web、数据库、FTP、LDAP、Web Service、JMS 等多个领域的性能测试 [8]，并且可以免费使用。使用 JMeter 可以灵活有效地模拟大量用户并发场景，准确地统计响应时间、吞吐量、错误率等性能指标，为测试人员分析系统性能瓶颈。

LoadRunner 是用于对系统的行为和性能作出预测的负载测试工具，主要由 VuGen、Controller 和 Analysis 三部分组成 [15]，VuGen 用于生成和编辑脚本，

Controller 负责控制、启动和排序测试脚本实例，Analysis 负责收集来自各种负载生成器的日志并格式化报告 [16]。与 JMeter 不同，LoadRunner 是付费的商业软件，其包括三个不同版本的产品：Cloud 提供基于 SaaS 的云端测试方案；Professional 提供基于项目性能测试工具；Enterprise 更关注企业中执行测试时的团队合作，为团队提供平台性的测试系统。

Locust 是基于 Python 的开源 Web 应用性能评估工具 [17]，其最大的特点是无需依赖复杂的 UI 界面或编写脚本代码，能够直接在 python 中引入依赖，通过编写代码来执行性能测试。当代码中定义的性能测试开始执行后，将会自动生成 Web 界面以实时显示测试进度以及最终的测试报告。利用精简灵活特性，测试人员能够对工具进行二次开发，以实现满足测试需求的工具。

表 1-1: 主流性能测试工具比较

指标	JMeter	LoadRunner	Locust
使用授权	开源免费	商用收费	开源免费
易用性	较高	低	较高
底层语言	Java	C/Java	Python
运行平台	GUI/命令行脚本	GUI	Python 脚本
测试报告	简单	完善但生成速度慢	简单
实时测试报告	不支持	不支持	不支持
并发机制	线程	进程/线程	协程
单机并发能力	低	低	较高
分布式测试配置	复杂	相对简单	较为复杂
资源监控能力	不提供	提供	不提供

上述三种测试工具的功能对比 [18][19][20] 如表 1-1所示，对比可知三种主流工具的目标用户存在着较大差异，JMeter 和 Locust 是面向个人开发者的简单、易用的测试工具，更适合测试入门者使用，而 LoadRunner 则凭借其全面的功能和高额收费，成为大型、中型企业的不二之选。在分布式测试功能上，虽然三种工具都有支持，但是均存在痛点。JMeter 多测试机器配置较为繁琐，由于只能在桌面端操作，测试团队内部难以同步测试过程与测试配置情况，并且 JMeter 不支持对于测试机器的资源监控。LoadRunner 由于其功能强大而完整，使用工具前需要高额的学习成本，并且其并不支持查看测试过程中的实时报告，当测试大规模并发时，需要等待很长时间才能够获取报告。Locust 没有提供 UI 界面，只能够使用脚本来进行测试，因此，虽然其在单机测试中拥有较

高的性能，也很难在企业的测试团队中大规模应用。

在 SUFFIAN.M.D 等人对不同性能测试工具在执行相同测试时响应时间的研究中 [20] 得出结论：不同测试工具在对同一网页进行相同压力测试时，其响应时间存在差异。造成响应时间不同的原因很多，主要原因之一是各性能测试工具的体系结构存在差异。使用 Java 开发的测试工具需要基于 JVM 才能执行，Java 堆大小的值设置将会对测试结果产生影响，合适的 Java 堆大小可以生成最佳的用户负载，而不会给客户端带来额外的负担。因此，不同测试工具执行相同测试的响应时间可以作为评判测试工具性能的指标之一，并且此指标是能够通过软件重构与数据通信来优化的。

为了对性能测试工具进行评估，清华大学的赵冲冲、白晓颖等人提出了一个性能测试工具能力评估框架 [21]。该框架从可靠性、测试能力与资源能力三方面对工具能力进行评估：可靠性包括正确的测试原理和环境仿真，目的是保证测试结果真实性；测试能力包括测试支持、测试监测与结果分析；资源管理能力描述了测试工具对接口、场景、报告等资源的管理与复用支持。满足上述三方面能力的测试工具，便能够满足测试人员乃至测试团队的大部分性能测试需求，本系统的需求设计将围绕此框架展开，以满足三个能力为最终目标。

1.3 本文的主要工作

针对上述研究现状，本文基于 Docker 容器调度框架，设计并实现了一个支持多资源池测试执行、参数配置、报告生成的弹性资源池性能测试系统，为测试人员提供灵活易用、稳定可靠、负载平衡的测试服务，为 Web 服务的可靠运行提供保障。本系统中划分出接口与场景定义、测试执行、测试资源池管理、测试报告生成、测试过程监测模块五个模块，分别是：

(1) 接口与场景定义模块：该模块为用户提供 Web 服务接口定义、参数配置、接口导入功能，并能够将多个接口组合成为测试场景，以实现测试方案制定的能力。

(2) 性能测试执行模块：其主要职责是提供总并发量、加压曲线、测试资源池等测试参数的配置，并利用 Apache JMeter 提供的命令行工具在测试机中执行性能测试，在测试过程中产生的响应信息通过 Kafka 进行传递，经过清洗处理后存储至数据库中。

(3) 测试资源池管理模块：该模块能够将多个测试机器组合成测试资源池，

测试人员能够在资源池中配置服务器 IP、端口号、SSH 等信息，并对资源池进行新增、删除、编辑操作。

(4) 测试报告生成模块：该模块通过读取测试执行过程中或完成后处理完成的数据，将数据按不同维度进行整合，最后将响应时间、吞吐量、错误率等性能指标以图标的形式展示在前端页面中。

(5) 测试过程监测模块：该模块主要依赖测试机中的 Prometheus 工具对服务器的 CPU 使用情况、内存占用情况、网络输入输出情况等指标进行收集，在测试过程中将监测数据实时展示给测试人员。

1.4 本文的组织结构

第一章为引言部分，介绍了本文的研究背景与当前国内外相关研究情况，通过结合当前互联网发展趋势分析出性能测试所存在的痛点，说明了实现弹性资源池性能测试系统的必要性，并描述了本文的主要工作与组织结构，使读者大致了解本文内容。

第二章为相关技术概述，该部分简要描述了本文所涉及到的相关软件技术，包括服务框架技术、测试监控技术和数据存储与传输技术，说明了各个工具的优势与选型原因。

第三章为需求分析与概要描述，本章根据第一章所分析的痛点问题，结合现状调研结果，提出了自己的解决方案，并将解决方案抽象为系统的功能性需求与非功能性需求。在需求的基础上对系统用例进行整合，对系统架构进行设计，最终将系统划分为接口与场景定义、测试执行、测试资源池管理、测试报告生成、测试过程监测五个模块。

第四章为系统详细设计与实现，该部分基于第三章划分出的五个模块，依赖 UML 类图、流程图、时序图、顺序图等软件工程方法，对各模块进行详细分析与设计，最后展示最终的完成截图与核心代码。

第五章为系统测试，本章根据第三、四章的需求与设计思路，选取系统的核心部分进行测试设计、执行，测试手段包括单元测试、接口测试与系统功能测试，以提高系统稳定性与可用性为目的展开测试。

第六章为总结与展望，本章总结与归纳了本文所实现的功能，说明了当前系统的进展情况，并结合专家与指导教师意见整理本系统中存在的不足之处，为后续的进一步优化迭代进行提前部署，希望将系统产品化。

第二章 相关技术概述

2.1 服务框架技术

2.1.1 Docker 容器技术

Docker 是当前最流行的容器技术之一，Docker 容器将操作系统层虚拟化，相比于虚拟机更便捷、更高效。标准化是 Docker 容器的特性之一，Docker 容器能够无视硬件环境、基础设施部署到任何服务器上。随着微服务、分布式等软件架构的兴起，Docker 的应用越来越广泛。

LXC（即 Linux Container）是 Docker 容器化的基础，LXC 能够为开发者创建一个虚拟的 Linux 软件环境，开发者能够在此环境中创建和管理系统应用，并且与其他环境隔离。Docker 使用内核级与应用程序级的 API 扩展了 LXC，这些 API 将会在独立隔离的容器环境中自动执行 CPU、内存、I/O、网络等进程 [22]，此外，Docker 使用命名空间将应用程序与底层环境隔离，从而开发者无需关心底层操作系统或硬件，只关心它运行的是 Docker 引擎即可。如图 2-1 所示，开发者能够在 Docker 容器中部署软件程序、数据库、云服务器资源等。

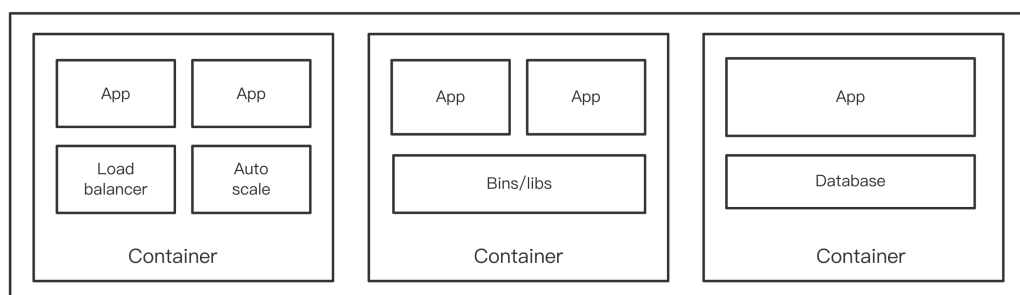


图 2-1: 使用 Docker 搭建不同应用

Docker 镜像可以看作是一个特殊的文件系统，它为容器提供运行时所需的环境资源、配置参数，是一个不包含任何动态数据的只读模版。镜像的构建是分层的，上层的镜像将作为下层镜像的基础，这使得镜像的复用、定制更为容

易。利用 Docker 镜像可存储、静态的特点，我们能够将提供测试的服务打包为 Docker 镜像，分发给不同的测试机使用。

如今微服务和分布式是大型软件程序的主流组织方式，在本性能测试系统中，Docker 是实现弹性资源池的核心技术。本系统将负责接口测试与性能测试的服务封装成 Docker 镜像，用户在 Web 系统上配置测试资源池后，系统会自动在资源服务器上通过 Docker 镜像启动 Docker 容器并执行服务。

2.1.2 SpringBoot

2014 年，Pivotal 团队正式发布了一个轻量级的开源框架——Spring Boot，其主要目标是帮助开发人员更快速地基于 Spring 来创建应用程序 [23]，由于其轻量易用的特性，Spring Boot 成为了构建 Java 应用最主流的框架。Spring Boot 具有以下特点：

(1) “开箱即用”：Spring Boot 内嵌了 Tomcat、Jetty、Undertow 等服务器供开发者选择，并为提供了许多自动化配置 [24]，启动器模块包含了 Spring-MVC、JPA、MongoDB、Spring Batch、SpringSecurity、ElasticSearch 等常用软件技术，开发者能够选择自己所需的依赖库，Spring Boot 将会负责安装并组织在框架中。

(2) “约定优于配置”：在 JDK1.5 版本推出注解功能后，Spring 使用注解来代替了大部分 XML 配置，但当注解数量增多时项目将会变得繁杂且难以管理。Spring Boot 提出了组合注解的概念，通过 AutoConfiguration 机制来解决配置的复杂性。Spring Boot 通过提供默认的文件结构、配置文件类型等约定，为开发者搭建应用降低成本。

本系统选用 Java 语言作为服务端开发语言，最终需要搭建一个 Java Web 应用部署至测试人员提供的测试机上。由于 Spring Boot 能够应用能够借助 maven 插件 [25] 基于开发者提供的 pom.xml 配置进行打包，开发者仅需执行 `java -jar` 即可将系统部署在不同环境中。此外，Spring Boot 内置的 `spring-boot-maven-plugin` 插件中提供了 `build-image` 命令，能够直接将项目代码构建为 Docker 镜像，极大提升系统的可移植性与灵活性。综上所述，本系统选用 Spring Boot 作为服务端框架进行开发。

2.1.3 Vue.js

Vue.js 是目前最主流的前端框架之一，其使用 MVVM 模式为开发者提供搭建一个单页 Web 应用的框架。

MVVM（Model - view - viewmodel）是一种软件架构模式，在此模式下，用户与视图进行交互，视图与 ViewModal 模块通过数据绑定、事件监听进行交互，通过 ViewModal 模块来获取 Modal 层的数据 [26]，其架构图如 2-2 所示。

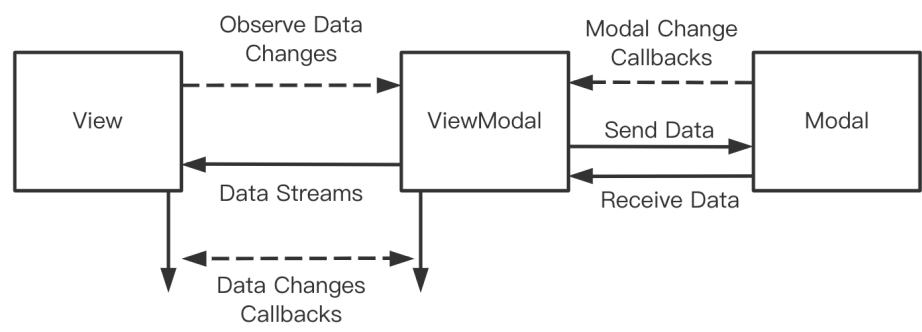


图 2-2: MVVM 架构图

Vue.js 承担了 ViewModal 模块的职责，将视图层和模型层进行分离，开发者能够在 <template> 标签中编写 HTML 代码，在 <script> 标签中编写 JavaScript 代码，通过双向数据绑定连接视图层和模型层。Vue.js 中使用发布者-订阅者模式实现了双向数据绑定，从而实现视图与数据的同步更新。

利用 Vue.js 灵活、易用、高效的特效，能够快速地搭建本系统的前端应用，并且保证了前端的性能。

2.1.4 ECharts 图表库

ECharts 是一个基于 JavaScript 的可视化图表库，图表是数据可视化的一种形式，它将大型数据集提取为可视化图形，以便轻松了解数据中的复杂关系，使得用户更直观地理解数据、更好地对数据进行分析处理。

从高层抽象来看（如图 2-3），Echarts 库可以分为 3 层 [27]:

- (1) echarts layer: 负责监听用户操作、处理数据、对数据进行编码、确定渲染布局；
- (2) zrender layer: 负责将形状和点渲染到页面上；
- (3) 基础组件: Canvas、SVG 等。

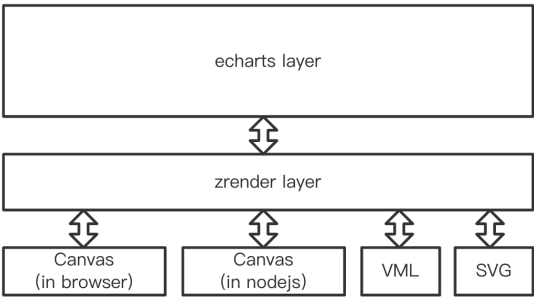


图 2-3: ECharts 架构图

从具体实现来看（如图2-4），图表中的各个“零部件”如 xAxis、legend、tooltip 等被抽象成 Component，在源代码中以组件的形式进行封装。每个组件按照 MVC 模型进行管理，如 legend 文件夹下有基本的 LegendModel(M)、LegendView(V)、LegendAction(C)。Model 文件负责管理配置项，View 文件的核心是 render 方法，Action 则负责注册、监听事件。

在本文的性能测试系统中，需要为用户以图表的形式展示性能测试结果。使用折线图来展示不同时间点的接口调用情况、压测机资源利用情况，使用饼图来展示接口成功/失败个数等。ECharts 配置简单、性能表现良好、图标库丰富，完美匹配本系统的需求。

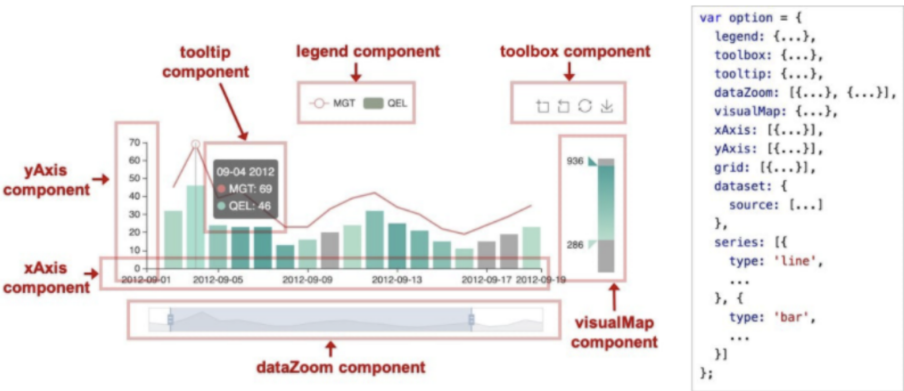


图 2-4: ECharts Component 设计

2.2 测试与监控技术

2.2.1 JMeter

JMeter 是一个开源的负载测试工具 [28]，可用于服务器、网络或对象，通过模拟重负载来测试其服务的可压缩性，或分析不同压力下的整体性能，常用于 Web 应用测试中。利用 JMeter 进行性能测试主要包括以下几个步骤 [29]：

1. 使用 JMeter 代理或 Badboy 等录制工具记录请求，生成测试脚本；
2. 设置线程组来模拟多用户并发环境，其中每个虚拟用户使用相同的测试脚本，这是性能测试的核心；
3. 利用参数化将测试数据和脚本分开，从而提高测试脚本的灵活性与测试覆盖率；
4. 设置断言，检查返回的响应是否符合预期结果，这是判断请求成功与否的方式之一；
5. 通过侦听器中的结果树获取请求和响应的详细信息，获取测试结果，测试结果中包含了平均响应时间、响应时间中位数、90% 用户响应时间、每秒处理请求数（吞吐量）等关键测试数据，这将成为测试人员衡量性能的关键指标；

JMeter 的开发者为测试人员提供了桌面应用与命令行两种使用方式，桌面应用适合直接使用 JMeter 进行测试的初级测试人员，命令行工具的可扩展性更高，开发者能够在命令行的基础上对测试执行脚本进行动态处理、或是拿到测试结果进行自定义展示。本系统的接口测试、性能模块就是基于 JMeter 脚本命令实现，站在巨人的肩膀上，保障了测试服务的稳定性。

2.2.2 Prometheus

Prometheus 是目前世界上最热门的开源监控工具之一，随着开发者对性能要求的提升，全面的服务监控是维护系统可靠性和稳定性的重要手段。为保证监控的准确性和实时性，Prometheus 存储了时序数据，即按相同时序 (相同名称和标签)，以时间维度存储的连续数据集合。

Exporter 是 Prometheus 的一类数据采集组件的总称，它负责从目标处搜集数据，并将其转化为 Prometheus 支持的格式。与传统的数据采集组件不同的是，它并不向中央服务器发送数据，而是等待中央服务器主动前来抓取。

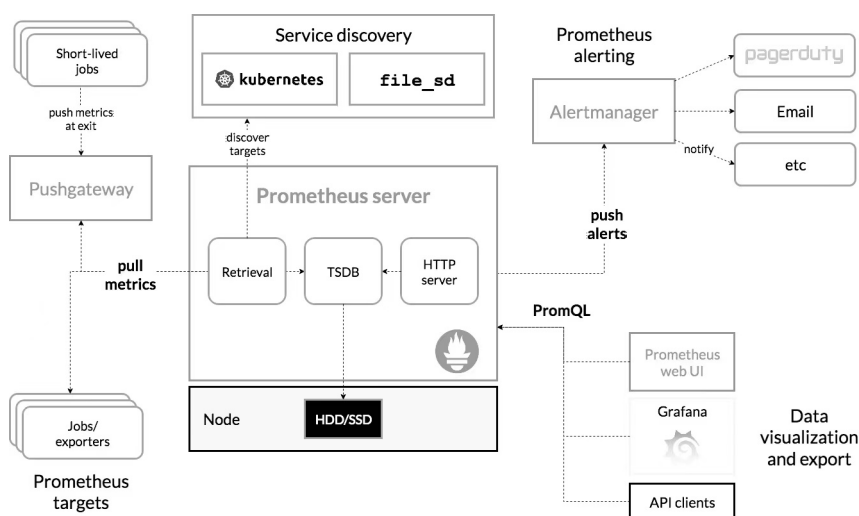


图 2-5: prometheus 架构图

Prometheus 目前支持的 Exporter 有：数据库、硬件、消息中间件、存储系统、HTTP 服务器、JMX 等。

如图 2-5 所示，Prometheus 将会按照配置的 exporter 等服务发现方式来确定监控目标，接着从目标处发起 HTTP 请求到特定的端点，最后将响应持久化至 TSDB（时序列数据库）中。开发者可以根据这些指标配置告警规则，或利用 API 将监控数据上报给其他服务。

Prometheus 为开发者提供了灵活的查询语言，通过服务发现或者静态配置来发现目标服务对象 [30]。在本系统中，需要对测试执行中的资源服务器进行监控，通过服务器的资源利用率、容量变化来观察性能测试执行过程。

2.3 数据存储与传输技术

2.3.1 MyBatis

MyBatis 是一个基于 Java 语言的持久层框架，与 Hibernate 不同，MyBatis 是一个 SQL 映射框架 [31]，它能够自动地将 SQL 执行结果填充到 Java 对象中，也能够将 Java 对象中的数据持久化到数据库中，并为开发人员提供了简单易用的 API 进行数据库交互。

在传统的持久层代码中，需要手动编写与数据库进行交互的方法，当查询复杂时还需要在代码中编写 SQL 语句。MyBatis 提供了能够自动生成 dao 代码的自动化工具，以节省持久层的开发成本。自动生成数据库交互代码的

流程如下：在 SpringBoot 项目中需要引入通用 Mapper 依赖，并在配置文件中 进行 MyBatis 的相关配置。当 Java 代码中的数据模型对象定义完成后，使用 maven 运行 mybatis-generator 插件生成对应数据对象的基础数据库交互类与 xml 文件。

本系统选用 MyBatis 编写持久层代码，部分基础数据库操作直接调用自动生产的 Mapper 文件，而复杂的操作逻辑则通过编写扩展 Mapper 实现。

2.3.2 Kafka

Kafka 来自于 Apache 的一款分布式、高吞吐量、开源的发布订阅消息系统，其广泛地应用于互联网中流式数据的处理，是大数据的主要软件技术。Kafka 系统在磁盘中进行消息持久化的复杂度为 $O(1)$ ，数据规模达到 TB 级别，支持高吞吐量的数据传输，能够满足每秒数十万条消息的处理需求 [32]。Kafka 的架构中有以下几个核心概念：

- producer: 负责记录流数据并发布消息到 broker
- broker: Kafka 集群中的代理，或用于接收消息的服务器
- consumer: 负责从 broker 获取、处理、消费数据

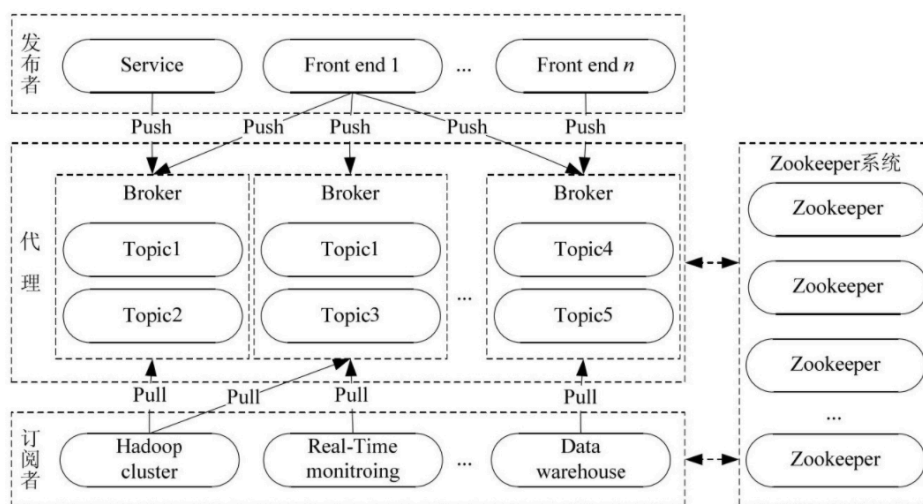


图 2-6: Kafka 架构图

Kafka 的架构是由以上三种角色构成的分布式架构，如图 2-6 所示，发布者可以向一个或多个主题发送消息，而订阅者则能够监听并拉取所需关注的主题。与传统队列将记录按特定顺序放在主机上，当多个用户向队列中发送消息

时，由于记录的传递与用户是同步的，所以可能会遇到 `consumer` 接受到的消息不一致的情况。`Kafka` 为了解决这一问题 [33]，在主题内部的分区内按序存储，主题间支持并行，这样平衡了消费者进程池上的负载，同时确定该特定分区的读取者只能是按顺序使用数据的消费者。

在本系统执行性能测试的过程中，将会源源不断地产生接口调用数据，简单地 `HTTP` 连接无法承担如此高吞吐量的数据，因此需要借助 `Kafka` 消息系统实现测试服务器与数据库的消息通信连接。`Kafka` 通过三种不同的交付语义来保证数据交付的可靠性，能够有效减少数据丢失情况 [34]。

2.4 本章小结

在本章中，从服务框架、测试工具、监控工具、数据存储与传输四方面，介绍了弹性资源池性能测试系统所使用的相关技术，并结合系统功能与架构需求，阐述了选用这些技术的原因。依赖于不同方向的专业前沿技术，本系统得以完整地实现。

第三章 需求分析与概要设计

需求是解决用户问题所需要达成的目标，也是整个系统开发的基础。本章基于第一章所分析的业务痛点，对系统进行需求分析与概要设计。首先确定系统需要解决什么问题，为了解决这些问题需要搭建一个怎样的场景与流程，这便是系统的整体框架，框架确定后结合弹性性能测试的应用场景，整合出系统的功能性需求与非功能性需求，并将其整合为用例。最后基于需求进行系统架构设计、核心流程设计与数据库模型设计。

3.1 系统整体概述

随着应用流量的激增，企业中因软件性能带来的重大事故层出不穷，大至火车抢票软件崩溃、2008 年奥运门票在线销售系统崩溃，小至日常生活中使用的微博、抖音等应用在产生重大新闻事件时产生的无限加载，始终存在的性能隐患不仅会给企业带来损失，也有可能導致政府、国家形象受损。因此，用户量越大、热度越高的应用更应该重视软件性能测试，并且测试难度也会更高。

为了实现对 Web 服务的高并发测试，人工进行大规模模拟接口调用与响应结果验证是极度困难的，因此需要借助自动化测试工具进行测试。JMeter 作为最主流的开源测试工具之一，其目标是提供一个平台无关的、高可用的开放平台 [28]。但利用 JMeter 执行测试存在一个致命的问题，当测试人员直接使用 JMeter 桌面版本进行测试时，由于其是一个由纯 Java 实现的 GUI 应用，若需要处理数以万级的并发量，配置普通的机器可能无法承担如此巨大的压力。因此我们需要为测试人员提供“分布式”的性能测试功能，将一台机器所承受的压力分摊出去，以保证接口的正常调用。

此外，因为在性能测试过程中，不同测试机的硬件配置不同将会导致 CPU 可承受的并发压力不同，当需要对多个测试机同时执行测试时，不能以平分的策略进行流量分发，而是给予软硬件配置更好的测试机更多流量，因此，测试资源池中的负载均衡配置是不可或缺的。

最后，当测试规模日渐提升时，个体测试人员将会成长为测试团队，为了

更好地管理性能测试过程、监测风险与规避风险，对各个测试机的实时监控和测试报告是必不可少的，为测试人员展示测试过程中资源池各个机器的 CPU 变化情况、内存占用情况等稳定性指标，能够更好地开展工程化的性能测试。

为了解决以上问题，本文需要实现支持弹性测试资源池的性能测试系统。基于这个目标，规划出本系统的流程，如图 3-6所示。

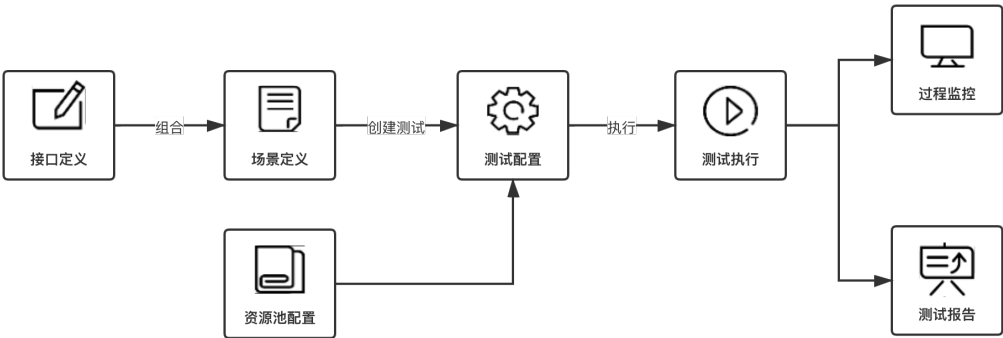


图 3-1: 系统流程图

首先，测试人员通过配置调用链接、参数、请求头等信息对待测试的接口进行定义；接着，为了制定测试方案，测试人员需要将接口组合成场景，并对顺序、前后依赖进行编排，场景定义完成后便可以创建性能测试；若测试规模较大，测试人员可以对测试资源池进行配置，增加多个测试机器来承担流量压力；随后，测试人员根据选定的场景创建性能测试任务，并进行总并发量、加压曲线、加压时间、测试资源池等信息的配置；任务创建完成后，便开始执行任务；任务执行过程中，测试人员能够实时监看当前的测试完成情况与测试机负载情况；测试任务完成后，将会生成一份测试报告，报告中包括响应时间、吞吐量、错误率、通过率等性能描述指标，测试人员能够参考这份报告，对性能进行评估。

3.2 系统需求分析

3.2.1 系统涉众分析

涉众作为系统实现后所服务或影响的个体，是软件开发中至关重要的一部分，找出真实的涉众才能够达到满足用户需求的目的。本系统作为产品化的性能测试系统，其涉众主要包括测试人员、测试管理员、测试交付方，分析结果

如表 3-1所示。

表 3-1: 系统涉众特征与期望分析结果

涉众名称	涉众特征与期望
测试人员	作为测试系统的核心使用者，测试人员具备快速上手使用系统的专业知识与专业技能，为了更好的配合其他方向的开发人员也需要具备一定的软件工程能力。测试人员期望能够直接在系统中执行完整的性能测试流程，无需切换其他工具，因此，系统的功能完备性、产品易用性和可理解性对测试人员来说至关重要。
测试管理员	作为测试团队或开发团队的管理者，测试管理员具备软件工程、软件测试、人员管理等多方向的交叉知识，作为团队的指引者把控着测试计划与结果。因此，管理员期望系统拥有管理能力，且最终能够输出清晰明了的测试报告。
测试交付方	作为测试结果的关注者，测试交付方可能是开发人员，也可能是不具备软件开发能力的测试需求提供方，他们仅负责提供测试需求，不关注测试执行的过程，最终期望得到一份数据简洁完整的测试报告。

测试人员是性能测试系统的核心使用者，其主要职责是了解待测软件的需求与使用方法，分析核心性能瓶颈点，制定出完备的测试计划并执行测试。在本系统中，测试人员可以进行待测接口的定义与编排，基于场景来创建测试，配置参数后在测试机器上自动化执行测试。

测试管理人员是一个测试团队或开发团队的管理者，其主要职责是对待测软件的核心性能测试点进行决策，指导测试人员完成测试计划的制定，并对最终的测试结果进行分析，整合成简明易懂的文字报告进行汇报。在本系统中，管理员的特殊功能并不多，他能够作为计划制定的主导者对接口与场景进行编排，也可以为接口或场景指定测试人，测试完成后能够导出测试报告进行分析。

测试交付方是软件的实际需求方，测试交付方可能是公司内部的开发团队，也可能是外部开发人员，由于企业可能并不是为自己开发软件，而是作为外包为其他企业或个人实现需求，因此他们只需要关注软件的实际运行效果和预期并发量。在本系统中，实际交付方能够查看最终的测试报告，通过报告中的错误率、通过率、接口调用情况等数据分析软件的性能情况，若性能表现较差，则需要与开发人员进一步分析瓶颈点，提升软件性能。

3.2.2 功能性需求分析

结合研究背景与涉众需求，将图 3-6 中的七个步骤拆分为 7 个核心功能性需求。接口定义流程中包含定义测试接口、测试接口连通性两个需求，场景定义流程包括定义测试场景、导入/导出测试场景两个需求，测试配置流程体现为创建性能测试需求，其余查看测试报告、管理测试资源池、查看资源池报告需求与流程图中的流程一致。功能性需求的具体描述见表 3-2。

表 3-2: 系统功能性需求分析

需求 ID	需求名称	需求描述
R1	定义测试接口	测试人员能够通过配置调用链接、请求参数、请求头等信息对测试接口进行定义。
R2	测试接口连通性	测试人员能够对已定义的接口进行测试，最终返回接口响应结果，包括状态码、响应内容、测试日志等。
R3	定义测试场景	测试人员或测试管理员能够对已定义的接口按顺序进行编排，并配置上下文参数，测试管理员能够将场景分配给某个测试人员负责。
R4	导入/导出测试场景	测试人员或测试管理员能够导入外部编写的 JSX 文件作为测试场景，也可以将系统内定义的场景导出为 JSX 文件。
R5	创建性能测试	测试人员或测试管理员能够基于已定义的场景创建性能测试任务，并对并发量、加压曲线、资源池等测试信息进行配置。
R6	执行性能测试	测试人员或测试管理员能够执行已创建的性能任务，并实时查看执行情况。
R7	查看测试报告	测试人员、测试管理员、测试交付方在测试过程中能够实时查看当前时刻的测试报告，并且在测试结束后能够查看完整的测试报告，报告包括性能总体评分、平均响应时间、平均吞吐量、错误率等性能指标情况。
R8	测试资源池管理	测试人员、测试管理员能够对测试资源池中的测试机进行配置，创建性能测试时能够指定资源池进行测试。
R9	查看资源池报告	测试人员、测试管理员能够查看资源池中各测试机器在测试执行过程中的 CPU 开销情况、内存占用情况等反映服务器负载的硬件指标。

3.2.3 非功能性需求分析

根据最新的《计算机软件需求规格说明规范》[35]，在软件需求规格中，需要明确性能需求，用以规定软件或人与软件作用的静态和动态的数量化需求。此外，规范中还描述了部分可作为非功能需求的软件属性，用于验证软件的实现情况。本文选取性能、可靠性、可用性、可维护性、可移植性、安全保密性六个指标用以描述非功能性需求，具体需求描述如表 3-3 所示。

表 3-3: 系统非功能性需求分析

名称	需求描述
性能	本系统在带宽 2M 以上的公网或稳定可用的内网环境中，接口返回时间应不超过 500ms，各接口在 4C 及以上配置的单机部署并发量达到 200，对数据库进行读写操作的响应时间不超过 500ms。
可靠性	本系统应在流量迸发或同时执行任务过多时利用一定的排队策略，保障系统的稳定运行。
可用性	本系统应保证环境、网络正常的情况下持续正确运行，对于路由错误、参数错误、网络错误等异常情况，应提前进行判断并告知用户错误原因，避免因异常情况引起程序崩溃。当服务器因非人为因素需要重启时，应能够在 15 分钟内启动服务并正常运行。
可维护性	本系统应对各个独立的模块进行隔离与抽象，将模块打包成独立的 Docker 镜像由专人维护，提供可扩展的接口以便于日后迭代，
可移植性	本系统应支持在所有可安装部署 Docker 容器的操作系统中部署与执行，包括但不限于 Linux 主流发行版与至少一种主流国产操作系统。
安全保密性	本系统应对恶意访问、修改、破坏的行为作出事前预防，如使用加密技术、域间限制技术、关键变量检查等。

3.3 系统用例分析

用例是对系统反映外界请求的描述，是通过用户使用场景来分析需求的技术，通过对系统的涉众、功能性需求、非功能性需求分析，绘制出本系统的用例图如3-2所示。

3.3.1 系统用例图

表 3-4: 系统用例与需求对应关系

用例 ID	用例名称	对应需求编号
UC1	定义测试接口	R1、R2
UC2	定义测试场景	R3、R4
UC3	创建测试任务	R5
UC4	执行测试任务	R6
UC5	管理测试资源池	R8
UC6	查看测试报告	R7
UC7	查看资源池报告	R9

作为测试人员，在系统中涉及定义测试接口、定义测试场景、创建测试任

务、执行测试任务、管理测试资源池、查看测试报告、查看资源池报告七个用例；测试管理员则只涉及定义测试场景、管理测试资源池、查看测试报告、查看资源池报告四个用例；测试交付方在系统中仅拥有查看测试报告一个场景。

用例作为不同涉众使用本系统的场景，需要覆盖前文提及的功能性需求，由于用例的粒度较粗，一个用例将会对应一个或多个功能性需求，在表3-4中展示了用例与各功能需求的对应关系。

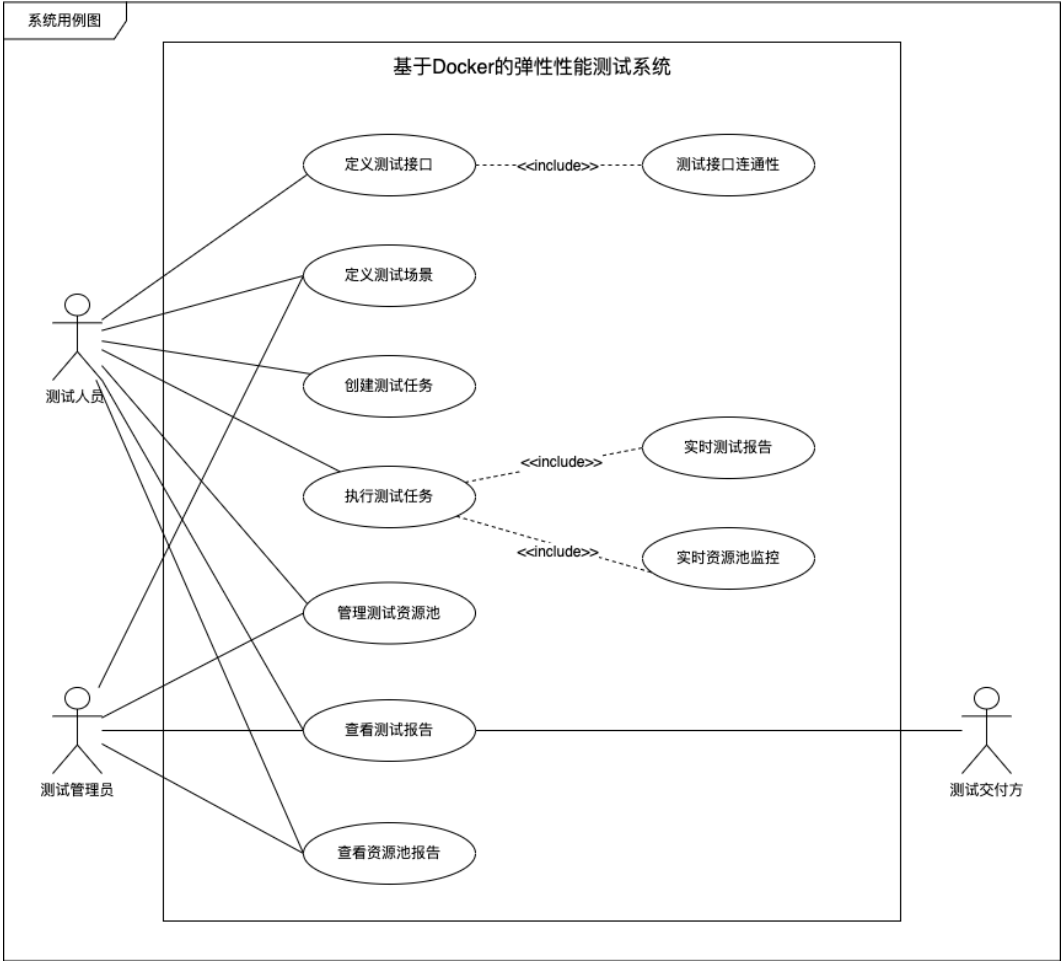


图 3-2: 系统用例图

3.3.2 系统用例描述

根据上述用例图，对本系统产生的所有与用户产生交互的用例细节展开描述，如下表3-5～3-11所示。

待测接口是性能测试得以进行的基础，测试人员需要完整填写接口链接、方法、参数、请求头等信息，并保证接口能够正常调用，定义测试接口用例的

详细描述如表 3-5所示。

表 3-5: 定义测试接口用例描述

用例编号	UC1
用例名称	定义测试接口
优先级	高
参与者	测试人员，目的是创建待测试的接口
前置条件	测试人员登录系统且进入接口定义界面
后置条件	接口创建成功
触发条件	测试人员在接口定义界面点击“+”按钮
正常流程	1. 点击“+”创建接口 2. 系统展示创建接口 Tab 界面 3. 测试人员填写接口名称、接口方法、接口 URL、接口描述等基础信息, 以及请求头信息与请求参数 4. 信息填写完成后，测试人员点击“测试”按钮以测试接口连通性 5. 系统显示接口响应内容，包括状态码、响应时间、响应体等 6. 测试人员保存接口信息 7. 系统显示保存结果
扩展流程	3a. 测试人员选择 Query、Rest、form-data 等键-值类型的参数 1. 系统展示选择框与输入框供测试人员进行填写

测试场景由多个测试接口组合而成，场景中接口的前后顺序决定着实际调用的顺序，因此测试人员需要对接口进行编排。此外，测试管理员可以指定场景的负责人，定义测试场景用例的详细描述如表 3-6所示。

完成场景定义后，便可以基于场景创建性能测试任务，测试人员需要填写任务名称、压力配置，等待后续任务执行。此外，测试人员能够导入外部的 JMX 文件作为场景以创建测试任务，此用例的详细描述如表 3-7所示。

任务创建完成后，在确认测试资源池均处于正常运行的情况下，测试人员可以启动任务执行，测试机器将会自动地按序调用场景中配置的接口，执行测试任务用例的详细描述如表 3-8所示。

测试资源池是一台或多台测试机器的集合，为了实现在大规模流量下并发测试的正常进行，需要将流量分发出去。测试人员或测试管理员能够对资源池进行创建、编辑、删除等操作，管理测试资源池用例的详细描述如表 3-9所示。

表 3-6: 定义测试场景用例描述

用例编号	UC2
用例名称	定义测试场景
优先级	高
参与者	测试人员、测试管理员（以下统称为用户），目的是组合多个测试接口
前置条件	用户进入场景定义界面，完成相关接口定义
后置条件	场景创建成功
触发条件	用户在接口定义界面点击“创建场景”按钮
正常流程	1. 用户点击“创建场景”按钮 2. 系统展示场景创建界面 3. 用户填写场景名称、描述等基本信息，随后点击添加接口按钮 5. 系统弹出窗口，展示所有已定义接口 6. 用户选中一个或多个接口，点击“复制”按钮将接口加入至场景中 7. 系统关闭弹窗，并在“场景步骤”中展示选中的接口 8. 用户选中接口并长按进行拖拽，对接口顺序进行编排 9. 系统展示编排后的顺序 10. 用户对接口中的信息进行补全和完善，确认后点击“保存”按钮 11. 系统显示保存结果
扩展流程	3a. 当用户为测试管理员时，可以为该场景指派责任人 1. 系统展示测试人员列表 2. 用户选择担任责任人的测试人员

表 3-7: 创建测试任务用例描述

用例编号	UC3
用例名称	创建测试任务
优先级	高
参与者	测试人员，目的是根据已定义场景创建任务
前置条件	完成场景定义或存在可用 JMX 文件
后置条件	任务创建成功
触发条件	用户在场景定义界面点击“创建性能测试”按钮或在性能测试任务列表中点击“创建任务”
正常流程	1. 测试人员选中指定场景，根据场景创建测试任务 2. 系统显示任务创建界面 3. 测试人员填写测试任务名称，任务名称默认为场景名称 4. 测试人员选择测试资源池 5. 测试人员设置并发用户数、加压方式（按持续时间或迭代次数）、压测时长等压力配置参数 6. 测试人员设置超时响应时间等高级配置，点击“保存”按钮 7. 系统显示保存结果
扩展流程	1a. 测试人员根据导入的 JMX 文件创建测试任务，将 JMX 文件拖拽至上传处 1. 系统显示上传成功的文件 6a. 测试人员若需要立即执行测试任务，可以点击“保存并执行”按钮 1. 系统显示保存结果并立即执行性能测试

表 3-8: 执行测试任务用例描述

用例编号	UC4
用例名称	执行测试任务
优先级	高
参与者	测试人员，目的是执行性能测试任务
前置条件	完成测试任务创建，资源池中的测试机均正常运行
后置条件	展示任务执行结果，执行完成后生成报告
触发条件	用户在测试任务详情中点击“执行”按钮
正常流程	1. 用户进入测试任务详情页面 2. 系统展示测试任务详情 3. 用户点击“保存并执行”执行当前任务 4. 系统跳转至测试报告页面，展示当前实时的测试报告
扩展流程	1a. 用户进入已完成的测试报告详情，点击“再次执行” 1. 系统再次执行此报告对应的测试任务，并跳转至新的测试报告界面

测试报告是测试结果的可视化展现方式，测试人员、测试管理员在测试执行过程中可以实时查看当前的测试报告，而测试交付方只能够看到最终的测试报告，查看测试报告用例的详细描述如表 3-10所示。

表 3-10: 查看测试报告用例描述

用例编号	UC6
用例名称	查看测试报告
优先级	高
参与者	测试人员、测试管理员、测试交付方（以下统称为用户），目的是查看测试结果
前置条件	测试任务开始执行
后置条件	测试报告以可视化的形式展示在界面中
触发条件	用户在测试任务中点击“测试报告”
正常流程	1. 用户进入测试报告列表页面 2. 系统分页展示测试报告列表 3. 用户点击进入测试报告详情 4. 系统展示测试报告详情页面，页面中将展示压力配置、测试概览、请求统计、错误记录、日志详情信息 5. 用户点击导出报告 6. 系统将报告生成为 PDF 格式文件并自动下载
扩展流程	无

为了在测试过程中对测试机器进行监控，测试人员和测试管理员能够在测试过程中和测试完成后查看测试机器的 CPU 使用情况、内存占用情况等硬件信息，查看资源池报告用例的详细描述如表 3-11所示。

表 3-9: 管理测试资源池用例描述

用例编号	UC5
用例名称	管理测试资源池
优先级	高
参与者	测试人员、测试管理员（以下统称为用户），目的是管理用于测试的机器
前置条件	已知测试机的 IP、端口号、SSH 等信息
后置条件	配置完成后的资源池可用于测试任务创建
触发条件	用户进入测试资源池管理界面
正常流程	1. 用户进入资源池管理页面 2. 系统展示当前资源池列表 3. 用户点击创建资源池 4. 系统展示资源池创建弹窗 5. 用户填写资源池名称、描述等基础信息 6. 用户添加一个 Node 节点测试机器 7. 系统展示一条新的 Node 节点 8. 用户填写机器的 IP、端口号、最大并发数等信息 9. 用户重复步骤 6-8 至完成所有测试机的配置 10. 用户点击“确定”保存资源池 11. 系统关闭弹窗并在列表中新增一个资源池 12. 用户点击启用/禁用开关调整资源池状态 13. 系统提示“状态修改成功” 14. 用户点击删除按钮删除某个资源池 15. 系统进行二次确认并提示删除结果
扩展流程	3a. 用户点击编辑按钮修改某个资源池信息 1. 系统展示资源池修改弹框

表 3-11: 查看资源池报告用例描述

用例编号	UC7
用例名称	查看资源池报告
优先级	中
参与者	测试人员、测试管理员（以下统称为用户），目的是查看测试机器的运行情况
前置条件	测试任务开始执行
后置条件	资源池报告以可视化的形式展示在界面中
触发条件	用户在测试任务中点击“资源池报告”
正常流程	1. 用户进入测试报告详情页面 2. 系统展示测试报告详情 3. 用户点击“查看资源池报告” 4. 系统展示资源池报告界面，界面中将展示测试过程中各个测试机器的 CPU 开销情况、内存占用情况等数据 5. 用户点击导出报告 6. 系统将报告生成为 PDF 文件并自动下载
扩展流程	无

3.4 系统总体设计

3.4.1 系统架构总览

基于前三节中对于系统需求的分析，为整个系统建立前后端分离的架构，整体上分为前端平台、后端服务、存储服务三部分，如图 3-3 所示。部署软件时可以将前后端服务集成到同一个 docker 中，无需使用 Nginx 作为代理。前端和后端通过 HTTP 请求进行通信，由服务端配置 CORS 跨域来解决跨域问题，将页面渲染逻辑与业务接口逻辑进行隔离。

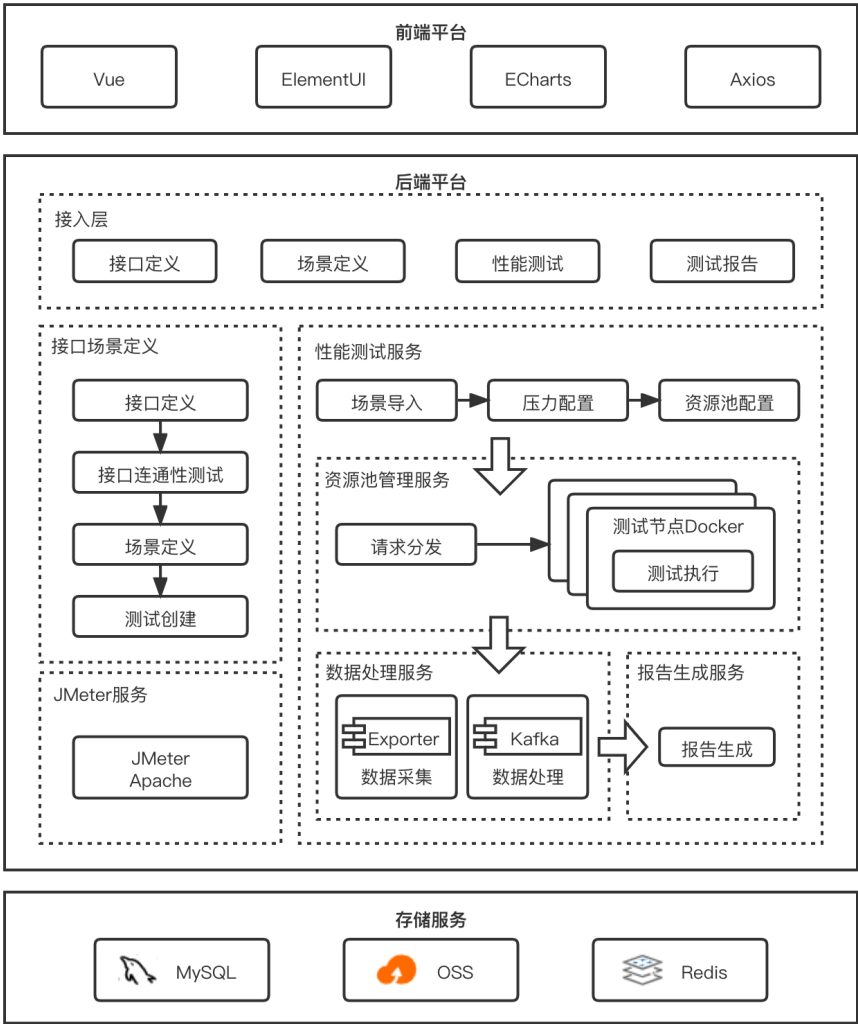


图 3-3: 系统整体架构图

前端平台是与系统的参与用户直接交互的平台，本系统的目标运行平台为

通用浏览器，因此前端技术需具备高效、兼容、灵活的特点，Vue 框架正是满足这些特点、社区高度活跃的主流前端框架之一。本系统结合 ElementUI 组件库与 ECharts 图表库实现美观的前端页面，以组件化的形式进行开发，减少重复代码。Axios 作为一款基于 Promise 的 HTTP 库，本系统使用其对后端接口进行调用，抽离出数据传输的相关代码。

后端服务除 JMeter 服务外，接口场景定义服务、性能测试服务、数据处理等其余服务均使用 Spring Boot 框架实现。后端采用 Controller-Service-Modal 的 MVC 架构组织代码，Controller 层负责定义对外提供的 API，Service 层负责处理实际实现逻辑，Modal 层负责进行数据库的增删改查操作。

接口场景定义服务为用户提供了接口定义、场景定义、接口测试功能，参考了 Postman 和 Jmeter 中的交互流程，底层依赖于 Jmeter 服务实现以接口测试。

性能测试服务为用户提供了场景配置、压力配置、性能测试等功能，性能测试执行由 JMeter 命令行工具实现，JMeter 输出的测试记录将存储在文件中。为实时查看测试执行情况、提高结果处理效率，使用 Kafka 作为消息中间件，资源池中的测试执行结果将输出到 Kafka，由数据处理服务读取 Kafka 中的结果数据并进行处理，处理完毕后将结果返回给前端界面。

资源池管理服务为性能测试提供独立节点类型的测试资源池，用于接受来自系统的性能测试任务，动态地启动 JMeter 容器执行性能测试，其底层运行环境为 Docker engine。

数据处理服务是用于处理测试结果数据的模块资源池中的性能测试完成后，会将测试结果数据通过 kafka backend listener 输出到 kafka，服务会从 kafka 的消息队列中读取测试结果数据并进行处理，处理完成后存入 Mysql 数据库中。

本系统的数据存储服务包括 MySQL 关系数据库、阿里云 OSS 存储服务、Redis 数据缓存服务。MySQL 作为主要的数据存储方式，用以存储系统中的所有业务数据，包括定义的接口、场景、测试任务、资源池等。阿里云 OSS 服务是一款安全、低成本、高可靠的存储服务，其最大的特点是无需额外编写接口即可提供给其他用户使用，系统的前后端服务、节点管理服务、Docker engine 等微服务 Docker 镜像将存储在 OSS 服务中，以便于系统的移植与部署。Redis 用于缓存系统中用户的登录态、资源池中各测试机的运行状态等数据。

3.4.2 4+1 视图

4+1 视图是描述软件架构的五个并发视图 [36]，每个视图都解决了一组特定的问题，包括逻辑视图、开发视图、进程视图、物理视图与架构描述，架构师在前四个视图中了解设计决策，并使用第五个视图来说明和验证它们。

(1) 逻辑视图

本系统的逻辑视图如图 3-4 所示，其反映了系统应向用户提供的服务，结合本系统的功能需求，利用抽象、封装和继承的原则，将系统分解成一系列的关键抽象。

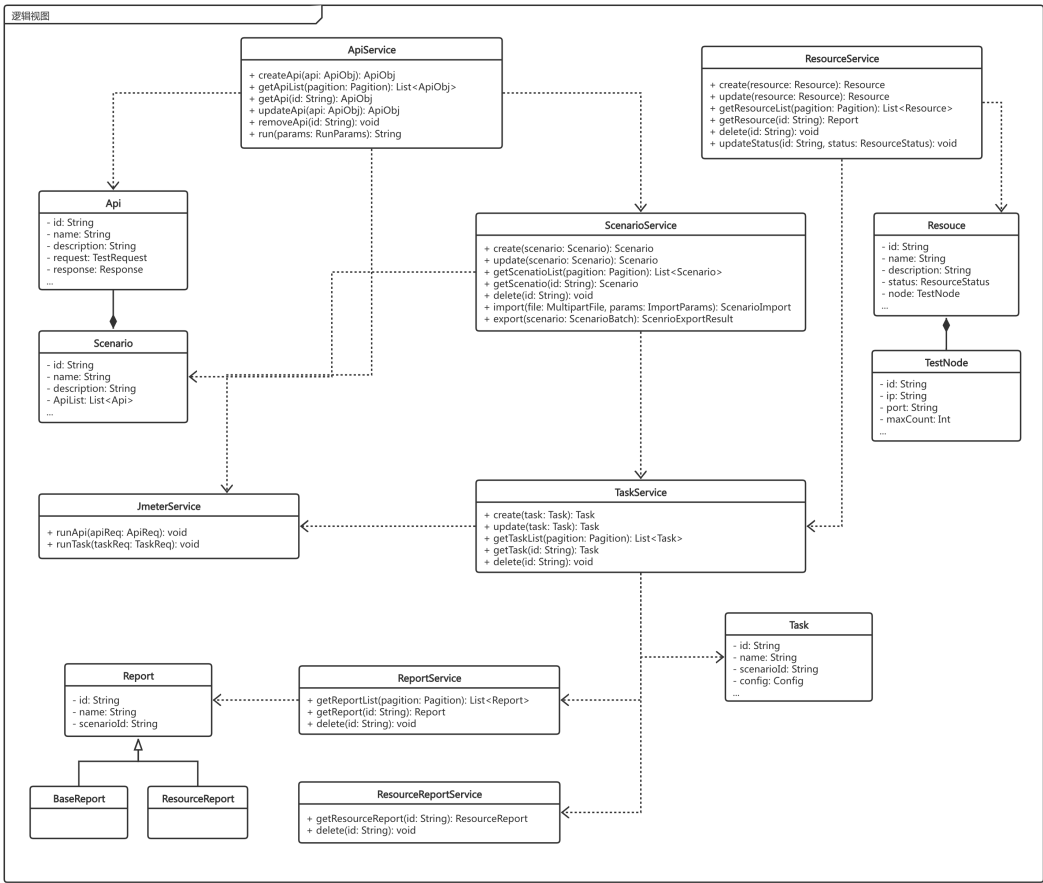


图 3-4: 系统逻辑视图

ApiService 是负责接口定义的功能模块，系统将测试接口抽象为 Api 类，ApiService 为用户提供了对 Api 类的增、删、改、查操作，除此之外，提供了测试接口连通性的 run 方法，run 方法的实现依赖于 JmeterService 的 runApi 方法。ScenarioService 是负责场景定义的功能模块，提供了查询场景列表、创建

测试场景、更新测试场景、导入/导出测试场景等功能。TaskService 是负责测试任务相关功能的模块，能够基于测试场景类 Scenario 创建测试任务，并对 Task 对象进行查询、创建、更新、删除等操作，最终的测试执行同样需要依赖于 JmeterService 中的方法。

ReportService 模块与 ResourceReportService 模块均是与测试报告相关的模块，ReportService 提供了测试过程中与完成后的测试结果展示，BaseReport 中包括了平均数据、图表数据、错误日志等数据。ResourceReportService 提供了测试资源池的监控数据，ResourceReport 中包含 CPU 开销情况、内存占用情况等字段，具体表设计见第四章中的详细设计部分。

ResourceService 作为资源池管理模块，是弹性资源池的核心。本系统将测试机抽象为 TestNode 对象，存储机器的 IP、端口号、负载配置等参数，一至多个 TestNode 可以组合为 Resource 对象。ResourceService 模块中提供了新建资源池、更新与删除资源池、查询资源池列表与详情方法。

(2) 开发视图

开发视图关注软件开发过程中的软件模块组织，需要考虑开发易用性、软件管理、代码可复用性相关等内部需求，以及编程语言和引入工具所带来的约束。本系统的开发视图如图 3-5 所示，分为前端、后端、数据存储三层架构。

前端平台层主要分为三个包，business 包为 Vue.js 编写的业务代码文件，包括前端页面、服务调用等代码；asset 包用于存放 js 工具库、外部 css 主题文件、图片、图标等静态资源；common 包是通用文件包，用于存放通用方法、通用组件、枚举常量等。

后端服务层基于 MVC 模型搭建，将业务逻辑、数据模型和界面展示分离开来。controller 包是前端与后端通信的枢纽，controller 文件夹用于存放使用 RestController 注释的接口定义文件，DTO 文件夹中定义了与前端通信的数据结构，parse 文件夹中整合了用于对请求参数或返回值进行数据处理的方法，handler 文件夹中包含对于部分需要额外处理的注解方法的拦截器。service 包用于实现各功能模块的业务逻辑，是 controller 层与 data 层的中间层，各个 service 中的方法将被 controller 层调用，并调用 data 层的 mapper 方法与数据库交互。data 包负责数据库处理，为 service 层提供数据更新与查询方法，modal 文件夹中存放与数据库表对应的系统实体对象，mapper 文件夹中是利用 MyBatis 实现的数据库操作接口。common 包中存放了枚举类、字符串常量等数据，以及用于文件、Date、Json 等数据处理的静态工具类。configure 包中定义了一系列配

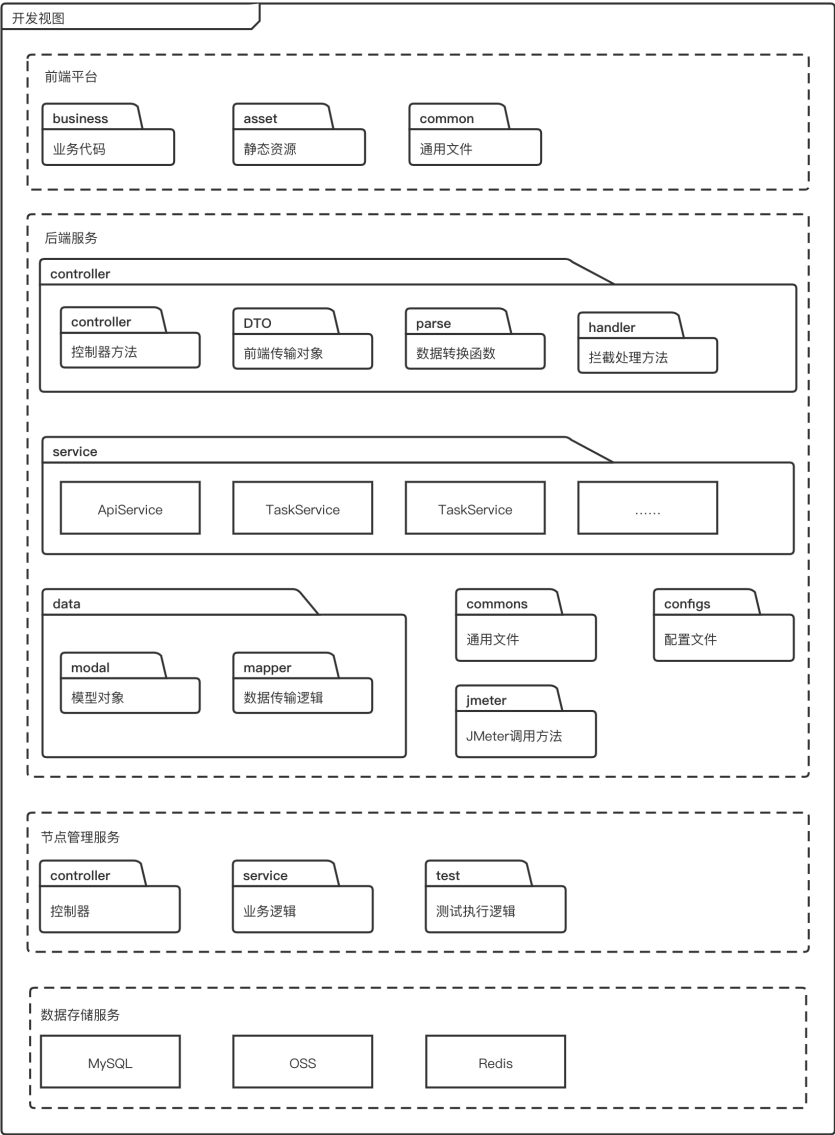


图 3-5: 系统开发视图

置文件，如 JMeterConfig、MyBatisConfig、KafkaConfig 等。jmeter 包提供了初始化 JMeter、测试参数配置等方法，为 JmeterService 提供接口。

节点管理服务层是独立于后端的微服务项目，为了提高系统可扩展性、降低耦合，因此将其从后端服务中抽离出来。由于同样基于 Spring Boot 框架开发，也包含 controller、service 包，而 test 包中是负责调用测试执行的相关方法，因逻辑较为复杂所以从 service 中独立成单独的模块。

(3) 进程视图

进程视图更关注系统中性能、可用性等非功能需求，通过描述不同抽象级

别的通信程序流程，分析系统中的并发流程与系统完整性问题。本系统的进程视图如图 3-6 所示，展示了一个性能测试执行过程中涉及的运行进程。

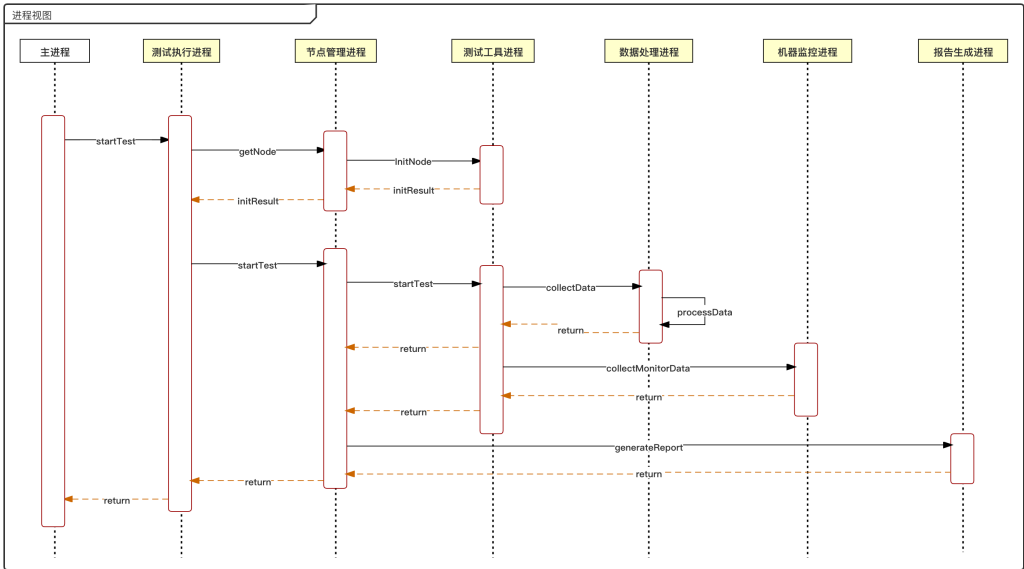


图 3-6: 系统进程视图

当测试人员在主进程中完成性能测试任务的创建后，便可以开始执行测试任务，此时主进程会将请求转发至测试执行进程，通知任务开始执行。测试执行进程接收到执行请求后，首先解析测试任务配置的资源池信息，通知节点管理进程对资源池中的测试机器进行初始化。节点管理进程便会启动一个或多个测试工具进程，在测试机器中创建并启动 **Docker** 容器，容器运行成功后将初始化结果返回至测试执行进程。随后，测试执行进程向节点管理进程发送执行测试的请求，各个测试工具进程将会调用 **JMeter** 命令开始测试，测试过程中产生的数据利用 **Kafka** 传递到数据处理进程进行清洗，测试过程中测试机的监控数据则会收集到机器监控进程中，处理完成后存储到数据库中。报告生成进程会在测试执行过程中间隔一定时间被调用，对数据进行整理为前端所需的数据结构。测试执行完成后，报告生成进程将会最后一次被执行，生成最终的测试报告。

(4) 物理视图

物理视图描述了软件架构到硬件的映射，以运维人员的视角对系统的部署逻辑与通信方式进行说明，如图 3-7 所示。

本系统中浏览器作为用户客户端，负责向主服务器请求展示页面与数据接口。由于节点管理、数据处理等高开销服务已独立成为微服务单独部署，后端

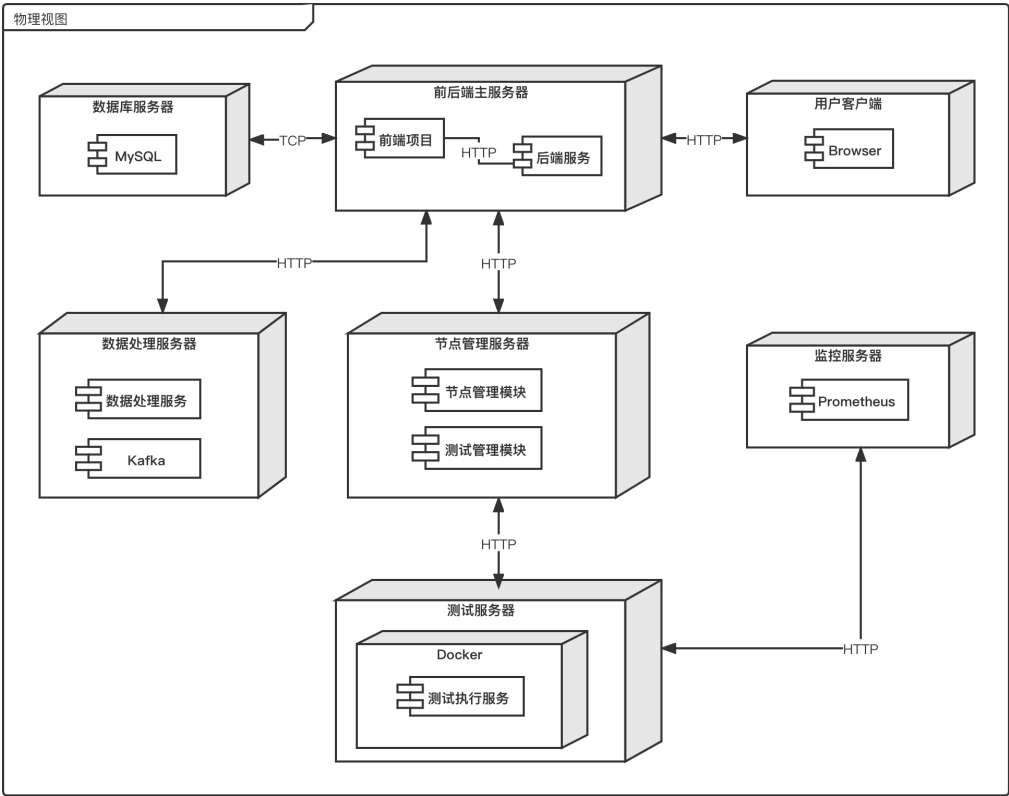


图 3-7: 系统物理视图

服务的请求压力较小，因此将前端与后端共同部署至同一个主服务中，未来能够打包成 Docker 镜像后能够快速完整地移植到新服务器中。其余各个微服务之间均通过 HTTP 请求进行相互访问，仅对外暴露 HTTP 接口，无需了解内部实现细节。此外，本系统中的数据库、Prometheus 监控组件等工具均使用开源的 Docker 镜像，在各服务器中快速启动容器，减少安装、配置带来的成本。

节点管理模块和数据处理模块均以微服务的形式进行开发部署，微服务带来的灵活性使得运维人员能够根据流量情况调整服务器个数，以实现服务多活。在数据处理服务器中，除数据处理逻辑服务外，还部署了 Kafka 流处理工具，在大数据量下提升数据传输效率。节点管理服务器将会对多个测试服务器进行管理，测试服务器中的测试执行服务以 Docker 容器的形式部署，执行测试时，测试服务器将自动基于 Docker 镜像启动执行服务。此外，各个测试节点上部署了收集监控数据的 Exporter 组件，用于监控运行中的测试服务器。

(5) 场景视图

“4+1”中的 1 是场景视图，是其余四个视图的核心，它为系统确定了用

户、边界与功能，由于已经在前三小节中进行了详尽分析，在此不做赘述。

3.4.3 多资源池性能测试流程设计

性能测试从创建、执行到结果展示的一系列流程，是本系统的核心所在，根据前文对需求与系统架构的梳理，绘制出更为详细的服务端视角性能测试流程图。如图3-8所示，主要流程包括创建测试、请求分发、数据采集、数据处理、报告整合五部分。

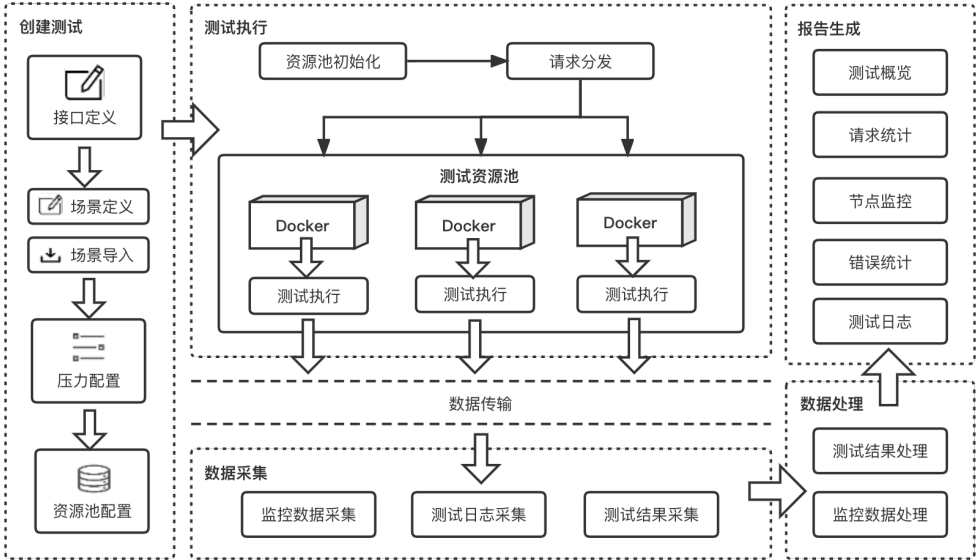


图 3-8: 多资源池性能测试流程图

性能测试的流程从创建测试开始，系统接收用户配置的测试场景、压力配置与用于执行性能测试的资源池，根据以上配置生成一个性能测试任务。任务开始执行后，首先将用户配置的资源池中所有测试节点进行初始化，随后依据一定的策略并结合用户配置的节点并发量上限，将不同并发量的请求分发至各个节点中，并开始执行性能测试。测试执行过程中，数据采集、数据处理与报告整合流程将同步进行，监控数据以 **Exporter** 的形式收集，而测试结果的处理和整合将会每隔一段时间运行一次，以保证用户能够在测试过程中获取实时数据。这便是多资源池性能测试的一个完整流程，各步骤的详细实现方法将会在第四章中给予说明。

3.5 持久化模型设计

为了使得用户的输入与输出能够长久保存，系统需要将数据设计为持久化对象，存储在关系型数据库 MySQL 中。本节中对系统的主要实体对象，包括测试接口、测试场景、测试任务、测试资源池、测试报告等模型进行设计。

3.5.1 系统业务数据库设计

表 3-12: 测试接口-Api 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	接口 ID	同时也是该表的主键
name	varchar(255)	接口名称	简洁易懂的接口描述
method	varchar(64)	请求方法	HTTP 请求方法名
path	varchar(1000)	请求链接	HTTP 接口调用链接
description	longtext	接口描述	详细具体的接口描述
request	longtext	请求信息	以 JSON 形式存储的请求配置
response	longtext	响应信息	以 JSON 形式存储的响应配置
user_id	varchar(64)	创建人 ID	创建该接口的测试人员 ID
create_time	bigint(13)	创建时间	该接口的创建时间
update_time	bigint(13)	更新时间	该接口的更新时间

表 3-13: 测试场景-Scenario 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	场景 ID	同时也是该表的主键
name	varchar(255)	场景名称	简单描述场景的名称
description	longtext	接口描述	详细具体的场景描述
follow_people	varchar(100)	责任人	该场景的测试负责人
step_definition	longtext	场景步骤	以 JSON 形式定义的步骤信息
step_total	int(11)	步骤总数	该场景中的测试步骤数
user_id	varchar(64)	创建人 ID	创建该场景的测试人员 ID
create_time	bigint(13)	创建时间	该场景的创建时间
update_time	bigint(13)	更新时间	该场景的更新时间

Api 数据库表用于存储用户定义的接口信息，用户能够对 Api 对象进行新增、查询、更新、删除操作，该模型的各字段设计如表 3-12 所示。

Scenario 数据库表存储了用户定义的测试场景信息，用户能够将一至多个 Api 组合成为一个测试场景，该模型的各字段设计如表 3-13 所示。

Task 表中存储了性能测试任务基本信息与运行状态，用户在完成测试任务的创建后便可以开始执行测试，该模型的各字段设计如表 3-14 所示。

表 3-14: 测试任务-Task 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	任务 ID	同时也是该表的主键
name	varchar(255)	任务名称	简单描述任务的名称
description	varchar(255)	任务描述	详细描述该任务的信息
load_configuration	longtext	压力配置	以 JSON 形式定义的压力配置
adv_configuration	longtext	高级配置	以 JSON 形式定义的高级配置
status	varchar(64)	任务状态	当前任务状态
resource_pool_id	varchar(64)	测试资源池 ID	测试用资源池 ID
user_id	varchar(64)	创建人 ID	创建该任务的测试人员 ID
create_time	bigint(13)	创建时间	该任务的创建时间
update_time	bigint(13)	更新时间	该任务的更新时间

ResourcePool 数据库表负责存储用户配置的测试资源池，ResourceNode 表存储了测试机器的配置信息，一个 ResourcePool 中能够关联一至多个 ResourceNode，ResourcePool 模型的各字段设计如表 3-15 所示，ResourceNode 模型各字段设计如表 3-16 所示。

表 3-15: 测试资源池-ResourcePool 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	资源池 ID	同时也是该表的主键
name	varchar(255)	资源池名称	简要描述资源池的名称
load_configuration	longtext	压力配置	以 JSON 形式定义的压力配置
configuration	longtext	资源池配置	以 JSON 形式定义的配置信息
status	varchar(64)	资源池状态	当前资源池启用状态
user_id	varchar(64)	创建人 ID	创建该资源池的测试人员 ID
create_time	bigint(13)	创建时间	该资源池的创建时间
update_time	bigint(13)	更新时间	该资源池的更新时间

表 3-16: 测试节点-ResourceNode 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	节点 ID	同时也是该表的主键
ip_address	varchar(255)	节点 IP 地址	测试服务器的 IP 地址
port	varchar(10)	节点端口号	测试服务器的服务端口号
max_thread	int(10)	最大并发数	性能测试时可接收的最大并发量

Report 数据库用于存储测试报告信息，测试报告分为两种类型，测试结果报告与资源池监控报告，不同类型报告中 content 字段的数据结构不同，Report 表各字段设计如表 3-17所示。

表 3-17: 测试报告-Report 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	报告 ID	同时也是该表的主键
task_id	varchar(50)	测试任务 ID	生成该报告的测试任务 ID
name	varchar(255)	报告名称	默认与测试任务同名，可编辑
status	varchar(64)	报告状态	当前测试任务的状态
type	varchar(64)	报告类型	测试结果报告/资源池监控报告
user_id	varchar(64)	创建人 ID	创建该资源池的测试人员 ID
create_time	bigint(13)	创建时间	该报告的创建时间
update_time	bigint(13)	更新时间	该报告的更新时间
start_time	bigint(13)	开始执行时间	该报告开始执行的时间
end_time	bigint(13)	结束执行时间	该报告结束执行的时间
duration	int(10)	执行时长	报告执行耗时

3.5.2 多资源池测试报告内容模型设计

测试报告作为测试结果的展示形式，涉及到大量不同类型的数据，包括概览数据、图表数据、错误日志等等，这些数据以 JSON 的形式存储在 ReportResult 表中的 report_value 字段中，当查询结果时可以根据 report_id 和 report_key 获取各类型结果数据，ReportResult 表的模型设计如表 3-18所示。本小节对报告中所需要存储的内容数据进行设计，为用户提供有参考价值的测试结果。对测试报告数据进行梳理整合后，得出以下 4 种主要数据模型：图表数据、错误数据、概览数据、请求统计数据。

表 3-18: 报告结果-ReportResult 对象模型设计

字段名	类型	含义	备注
id	varchar(50)	报告结果 ID	同时也是该表的主键
report_id	varchar(50)	报告 ID	对应测试报告的 ID
report_key	varchar(64)	结果类型标识	用于标明数据类型的枚举值
report_value	longtext	结果内容	以 JSON 形式定义的结果数据

报告的图表中一个数据点的结构如表 3-19所示，多个数据点将连接成一个折线图，为用户呈现数据的变化情况。

表 3-19: 图表数据-ChartData 结构设计

字段名	数据类型	含义
xAxis	String	x 轴数据
yAxis	BigDecimal	左侧 y 轴数据
yAxis2	BigDecimal	右侧 y 轴数据
groupName	String	图表 Series 名称

错误数据主要为用户展示测试过程中产生的错误响应情况，通常情况下可以认定响应状态码不为 200 的请求是错误的，报告中需要记录这些错误，一条 ErrorData 数据对应着一种错误状态码类型，其数据结构如表 3-20所示。

表 3-20: 错误数据-ErrorData 结构设计

字段名	数据类型	含义
errorType	String	错误状态码
errorNumber	String	此状态码对应的错误数量
percentOfErrors	String	该错误在所有错误中的占比
percentOfAllRequests	String	该错误在所有请求中的占比

结果概览用于展示测试的简明结果，通过具有代表性的单个数据，使得用户能够快速知悉测试结果，反馈待测软件的性能表现，OverviewData 的数据结构如表 3-21所示。

表 3-21: 概览数据-OverviewData 结构设计

字段名	数据类型	含义
maxUsers	String	最大并发用户数量
avgTransaction	String	平均吞吐量
errors	String	请求错误率
avgResponseTime	String	平均响应时间
90ResponseTime	String	90% 响应时间
avgBandwidth	String	平均带宽速率

除总览数据外，报告中同样需要展示各请求的详细信息，请求统计数据的结构如表 3-22 所示，在测试接口的维度上对所有请求进行整合统计，向用户展示测试场景中每一个接口请求的执行、响应情况与性能表现。

表 3-22: 请求统计数据-RequestData 结构设计

字段名	数据类型	含义
label	String	请求接口链接
samples	String	接口总执行数量
ko	String	错误接口数量
error	String	错误率
avgResponseTime	String	平均响应时间
minResponseTime	String	最短响应时间
maxResponseTime	String	最长响应时间
90ResponseTime	String	90% 响应时间
95ResponseTime	String	95% 响应时间
99ResponseTime	String	99% 响应时间
transaction	String	吞吐量
received	String	网络流入速率
sent	String	网络流出速率

3.6 本章小结

本章结合研究背景与国内外现状总结出本系统需要解决的问题，从高层整体的视角进行分析，对系统的流程和应用场景进行了大致梳理。进而分析了流程中的涉众特征与期望，将待解决的问题转化成需求，整理出了本系统的功能需求和非功能需求。随后根据需求设计用户场景，利用用例图与用例描述进行

了说明，至此，本系统的需求已经明晰。需求确定后，首先结合第二章中说明的相关软件技术，对系统的整体分层结构进行了设计，进而利用 4+1 视图中的逻辑、开发、进程、物理四视图进一步地对系统架构作出详细设计。最后，对本系统中各个对象实体在数据库中的持久化数据结构进行了设计，这将成为代码开发中的底层基础。

第四章 系统详细设计与实现

4.1 接口场景定义模块详细设计与实现

4.1.1 接口场景定义模块架构设计

接口场景定义模块负责实现需求中的接口定义、接口连通性测试、场景定义功能，作为创建性能测试的前置步骤，构建可用、完整的测试场景至关重要。本模块的架构设计如图4-1所示，由于不涉及其余微服务，此架构整体上以前端视图、后端业务逻辑、存储数据库三层进行组织。

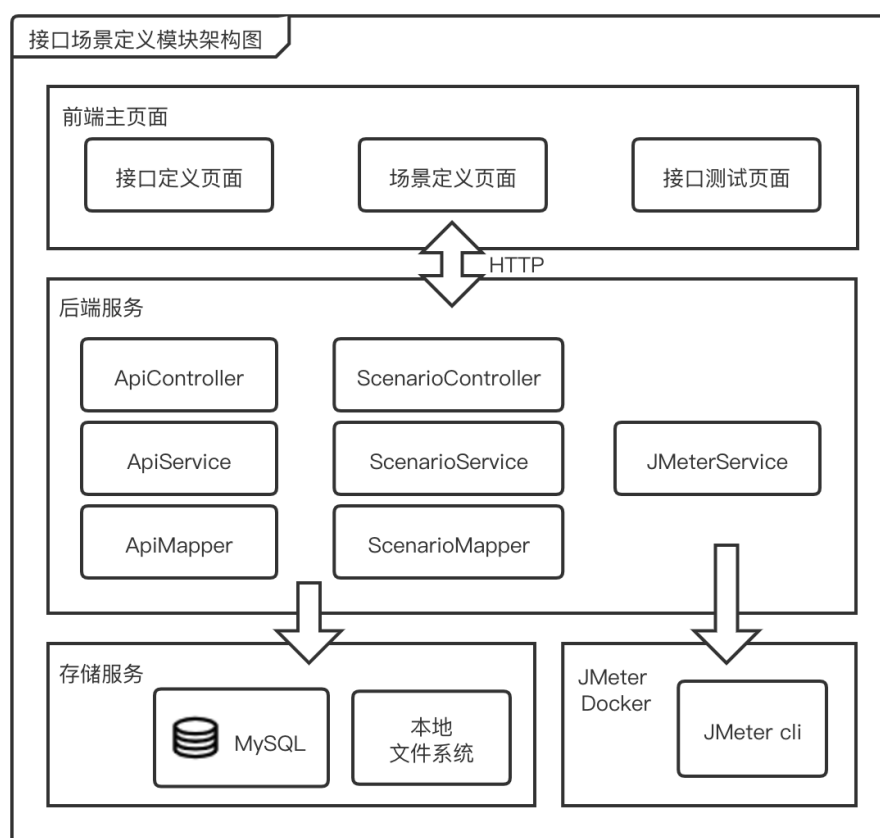


图 4-1: 接口场景定义模块架构图

在后端服务中根据功能划分为 ApiController、ScenarioController 两个控制器，分别通过 ApiService、ScenarioService 和 JMeterService 来处理业务逻辑，最后利用 MyBatis 提供的 Mapper 进行数据操作。JMeterService 是业务逻辑中较为特殊的一个，它的主要职责是启动测试机中的 JMeter Docker 容器，通过命令行的形式利用 JMeter 进行接口测试。

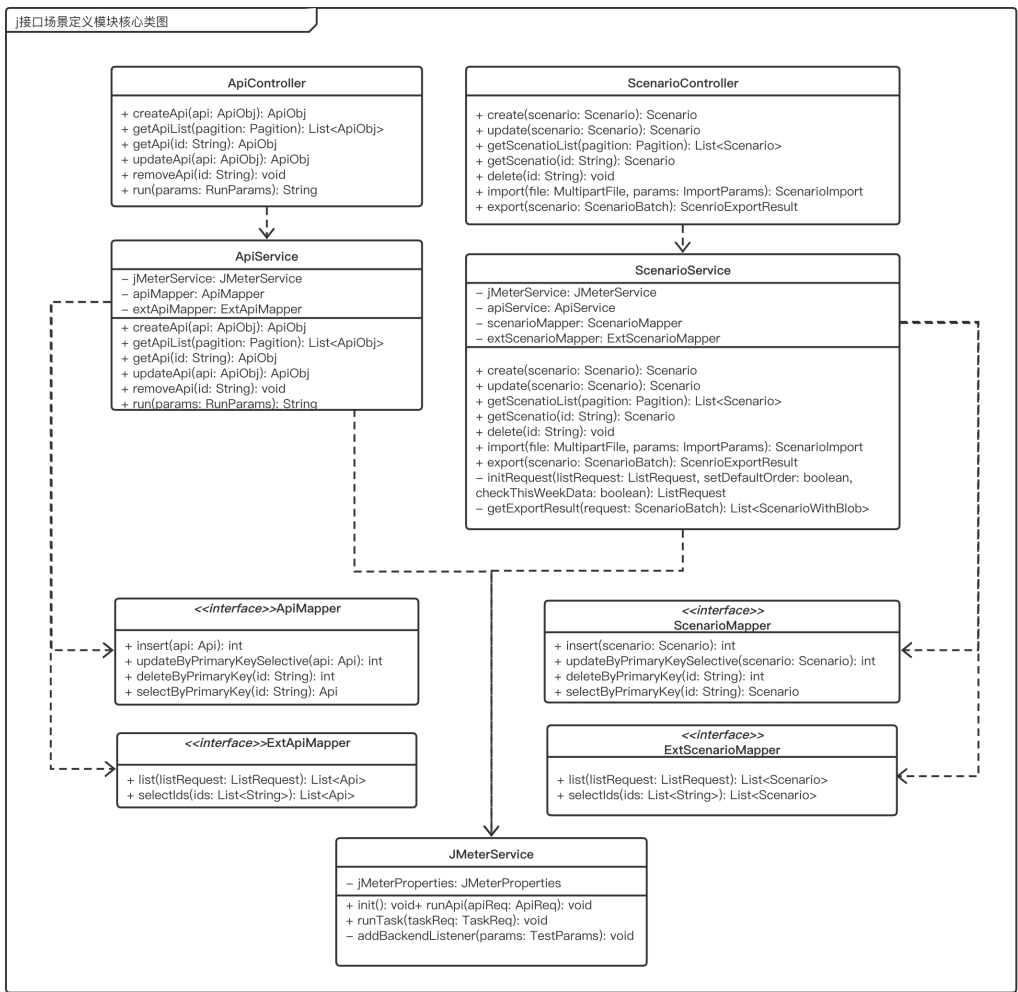


图 4-2: 接口场景定义模块类图

4.1.2 接口定义与测试模块核心类设计

本模块的核心类图设计如图 4-2 所示，图中展示了接口与场景相关对象的行为与状态。ApiController 对外提供了 Api 相关的 RESTful 接口，主要调用 ApiService 服务对实体对象进行增删改查操作，用于测试接口连通性的 run 方法

则调用 `JMeterService` 中公用的 `runApi` 来实现, `ApiMapper` 是 `MyBatis` 根据 `Api` 实体自动生成的数据库操作接口类, 而 `ExtApiMapper` 则是自定义的数据库操作类, 如根据多个筛选项进行接口搜索这些需要扩展的操作。

测试场景相关类的逻辑与 `Api` 大体相似, 在 `ScenarioController` 中除基础的增删改查操作外, 还包括了 `import` 和 `export` 操作, 用于将 `JMX` 文件导入为测试场景, 以及将场景导出成 `JMX` 文件。`import` 方法接收场景配置参数与 `MultipartFile` 文件, 利用 `jmeter` 提供的 `SaveService` 服务中的 `loadElement` 接口对文件的 `InputStream` 进行解析, 可以得到完整的测试对象, 最后将此对象转换为系统中的 `Scenario` 实体存储至数据库即可。`export` 方法的逻辑与 `import` 相反, 先将 `Scenario` 转换为 `JMX` 对象中的 `HashTree` 形式, 再调用 `SaveService` 中的 `saveTree` 方法生成文件, 具体代码实现见关键代码小节中。

在第三章中持久化设计中提到, 测试场景实体的 `step_definition` 参数以 `JSON` 的形式存储了场景中的步骤信息, 该 `JSON` 的数据结构如代码段 4.1 所示。其中 `hashTree` 字段以列表的形式存储了各个步骤中的测试接口配置信息, 由于其中仍嵌套着对象字段, 因此并未将其设计为数据实体, 采用 `JSON` 形式描述更为简洁。此 `hashTree` 结构中对象各字段及其含义如表 4-1 所示, 由于此对象能够用于表示一个测试接口信息, 因此 `Api` 实体中的 `request JSON` 字段也以此结构进行存储。

Listing 4.1: Scenario Config 场景配置的 JSON 主要结构

```
{
  "id": "b5621fb2-14c5-47de-9b16-ec39419f8073",
  "name": "登陆",
  "variables": [],
  "enableCookieShare": true,
  "hashTree": [{
    "id": "194ece82-a5b9-4e85-bfe9-86f3e7421b03",
    "name": "登陆-管理员",
    "active": false,
    "index": "1",
    "enable": true,
    "hashTree": [],
    "protocol": "HTTP",
    "method": "POST",
    "path": "www.moocetest.net/api/test/login",
```

```
"connectTimeout": "6000",
"responseTimeout": "6000",
"headers": [{
    "enable": true,
    "encode": true,
    "file": false,
    "required": true,
    "valid": false
}],
"body": {
    "binary": [],
    "json": true,
    "kV": false,
    "kvs": [],
    "oldKV": false,
    "raw": "{\n \"email\": \"test\",\n \"password\":\n  \"123456\"\n}",
    "type": "JSON",
    "valid": true,
    "xml": false
}, ...
}]
}
```

4.1.3 接口场景定义模块顺序图

接口场景定义模块的顺序图如图4-3所示，下图展示了模块中各类之间的顺序调用关系。此模块中 Api 和 Scenario 两个功能模块的逻辑相似，Controller 接收到 HTTP 请求后调用 Service 中对应的业务处理方法，Service 将会调用 Mapper 接口进行数据查询、更新操作。而接口连通性测试时，ApiController 将通过 ApiService 中的 run 方法，调用 JMeter Service 中的 runApi 方法，在 JMeter Service 中实现 JMeter 测试执行的逻辑。

表 4-1: HashTreeObj 对象结构设计

字段名	数据类型	含义
id	String	测试接口 ID
name	String	测试接口名称
active	boolean	接口是否可用
index	int	接口在测试步骤中的索引顺序值
protocol	String	接口通信协议
method	String	接口方法
path	String	接口调用链接
connectTimeout	int	请求连接超时时间
responseTimeout	int	响应接收超时时间
header	List<Object>	请求头配置列表
body	Object	请求体对象配置
rest	List<Object>	响应类型配置列表

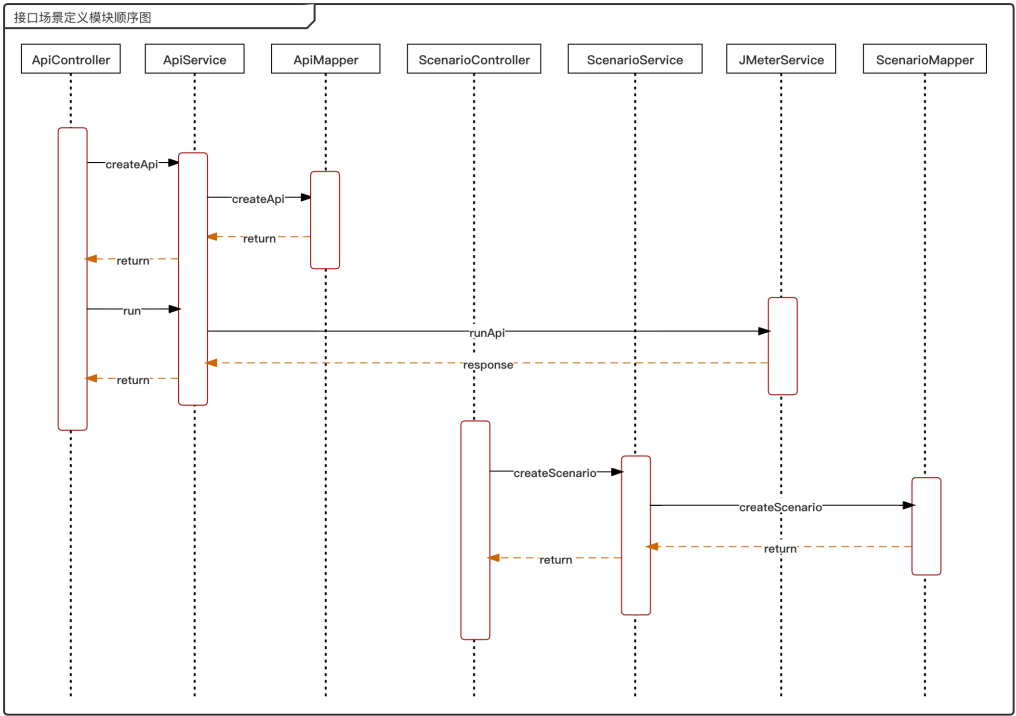


图 4-3: 接口场景定义模块顺序图

4.1.4 接口场景定义模块关键代码

代码段 4.2 中展示了创建测试场景的核心代码逻辑，代码段中的两个方法均是 `ScenarioService` 类中的方法。`create` 方法中通过调用 `buildScenario` 方法对 `Scenario` 对象进行构造，`buildScenario` 负责将 `ScenarioRequestDTO` 转换为实体存储对象，最后通过 `ScenarioMapper` 将数据插入数据库中。

Listing 4.2: 创建测试场景核心代码

```
public Scenario create(ScenarioRequest request,
    List<MultipartFile> configs) {
    // 构造Scenario对象
    final Scenario scenario = buildScenario(request);
    scenario.setCreateTime(System.currentTimeMillis());
    // 数据库新增字段
    scenarioMapper.insert(scenario);
    // 存储场景配置文件
    List<String> uploadIds = request.getUploadIds();
    FileUtils.createBodyFiles(uploadIds, configs);
    return scenario;
}

private Scenario buildScenario(ScenarioRequest request) {
    Scenario scenario = new Scenario();
    scenario.setId(request.getId());
    scenario.setName(request.getName());
    scenario.setFollowPeople(request.getFollowPeople());
    scenario.setStepTotal(request.getStepTotal());
    scenario.setUpdateTime(System.currentTimeMillis());
    scenario.setScenarioStep
        (JSON.toJSONString(request.getScenarioStep()));
    if (request.getUserId() == null) {
        scenario.setUserId(SessionUtils.getUserId());
    } else {
        scenario.setUserId(request.getUserId());
    }
    return scenario;
}
```

此模块中的另一核心代码为接口连通性测试代码，如 4.3 所示。runApi 是 ApiService 中负责处理请求的逻辑代码，在此方法中根据 Request 中传递的 ApiRequest 对象生成测试配置的 HashTree。HashTree 是 Java 中描述树状哈希表的数据对象，拥有更低成本的查询、更新速度，在系统中用于存储复杂的接口请求配置对象。

ApiRequest 类中包含 hashTree 对象，此对象的结构同样如表 4-1 所示，由于此对象可能被嵌套定义，因此 generateHashTree 方法中需要调用递归的 toHashTree 方法对 hashTree 进行展开。

Listing 4.3: 测试接口连通性核心代码

```
// ApiService.java
public String runApi(RunApiRequest request) {
    // 将HashTree JSON转换为对象
    HashTree hashTree =
        request.getApiRequest().generateHashTree();
    // 调用执行方法
    jMeterService.runApi(request.getId(), hashTree);
    return request.getId();
}

// ApiRequest.java
public HashTree generateHashTree() {
    HashTree hashTree = new ListedHashTree();
    this.toHashTree(hashTree, this.hashTree);
    return hashTree;
}

// ApiRequest.java
public void toHashTree(HashTree tree, List<ApiRequest> hashTree)
{
    if (CollectionUtils.isEmpty(hashTree)) {
        for (ApiRequest el : hashTree) {
            el.toHashTree(tree, el.hashTree);
        }
    }
}
```

4.2 测试执行模块详细设计与实现

4.2.1 测试执行模块架构设计

测试执行模块是负责测试执行、测试停止、测试结果收集存储的模块，在系统中测试场景包括接口连通性测试、弹性资源池性能测试，其架构图如 4-4 所示。

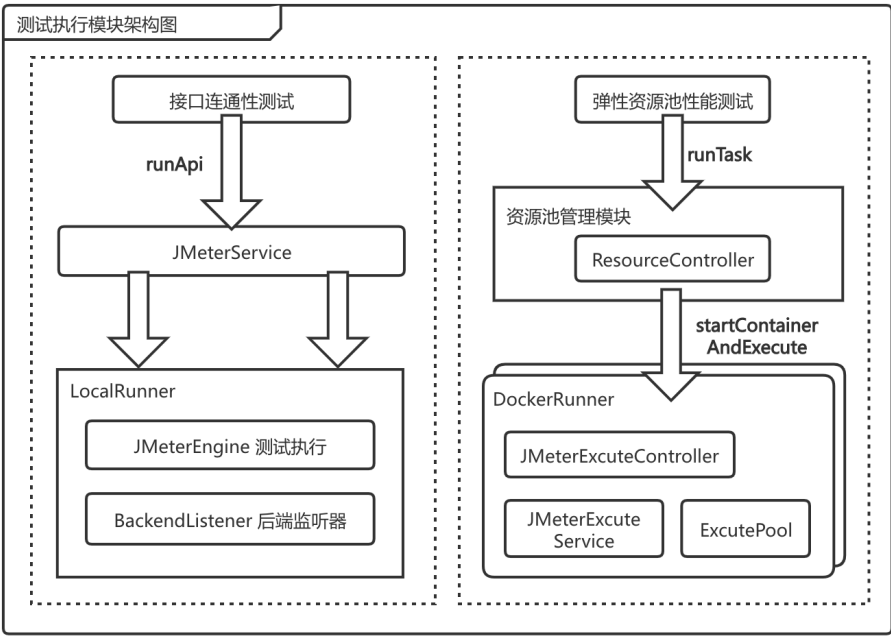


图 4-4: 测试执行模块架构图

图中左侧展示的是本机进行接口连通性测试执行的架构设计，后端服务的 Controller 接收到接口连通性测试请求后调用 JMeterService 的 runApi 方法，接收到性能测试请求后调用 runTask 方法。JMeterService 在 run 方法中创建 LocalRunner 实例，此实例主要依赖 apache.jmeter 库中提供的两个工具类来实现，其中 JMeterEngine 负责执行 hashTree 中的测试配置，BackendListener 负责异步地监听测试并获取结果。

右侧图展示的是弹性资源池场景下的性能测试，由于测试机器可能不止一个，在接收到 runTask 请求后需要调用资源池管理服务，遍历该资源池中的各个测试节点，每个测试节点对应一个 DockerEngine 类实例，由 DockerEngine 负责启动 DockerRunner 容器并调用 JMeterExcuteController 中定义的 RESTful 接

口，JMeterExecuteService 负责接收测试参数并利用 Java 提供的线程池管理类执行测试，性能测试将在资源池中的各个测试机中执行，各个测试机中的每个测试任务独占一个线程池，互不干扰。

4.2.2 测试执行模块核心类图

当进行单个测试接口测试时，由于无需对接口加压，仅需要调用接口获取响应结果，因此可以直接在后端服务机器上进行测试。如图4-2所示，执行接口测试时需要调用 JMeterService 中的 runApi 方法，此方法可以分解为三个步骤：

- (1) 初始化 JMeter 配置，对应 JMeterService 中的 init 方法。
- (2) 实例化后端监听器，设置实际监听类，对应 addBackendListener 方法。
- (3) 执行测试，调用 LocalRunner 中的 run 方法来实现。

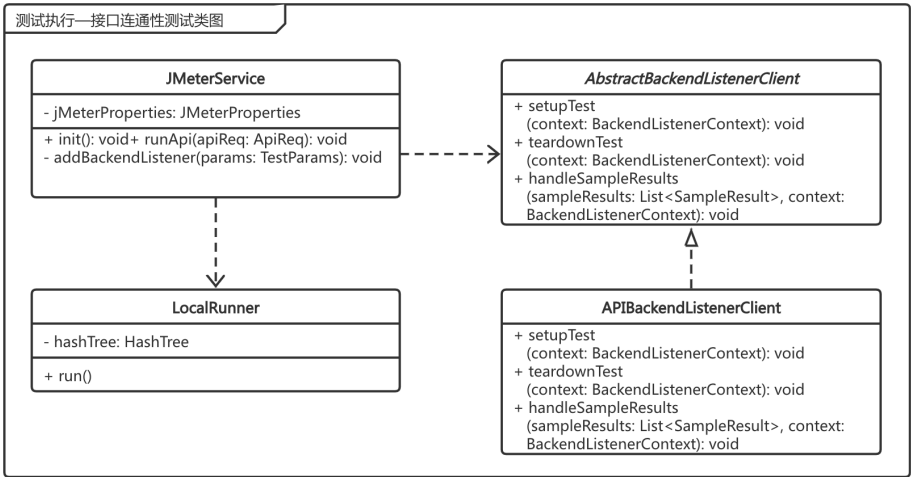


图 4-5: 测试执行模块——接口连通性测试执行类图

测试接口连通性的场景的架构图如4-5所示。在 addBackendListener 方法中，需要设置自定义的监听器对象以获取所需的测试结果数据，因此类图中的 APIBackendListenerClient 就是实现 apache.jmeter 库中提供的 AbstractBackendListenerClient 抽象类的自定义监听器。LocalRunner 是负责实际测试执行的类，当 run 方法被调用时，将会是实例化 JMeterEngine，通过调用 JMeterEngine 的 runTest 方法启动测试执行。

由于执行性能测试的核心逻辑与单个接口连通性测试不同，因此将其抽离成独立的微服务，通过 HTTP 与后端服务通信，弹性资源池性能测试执行的类

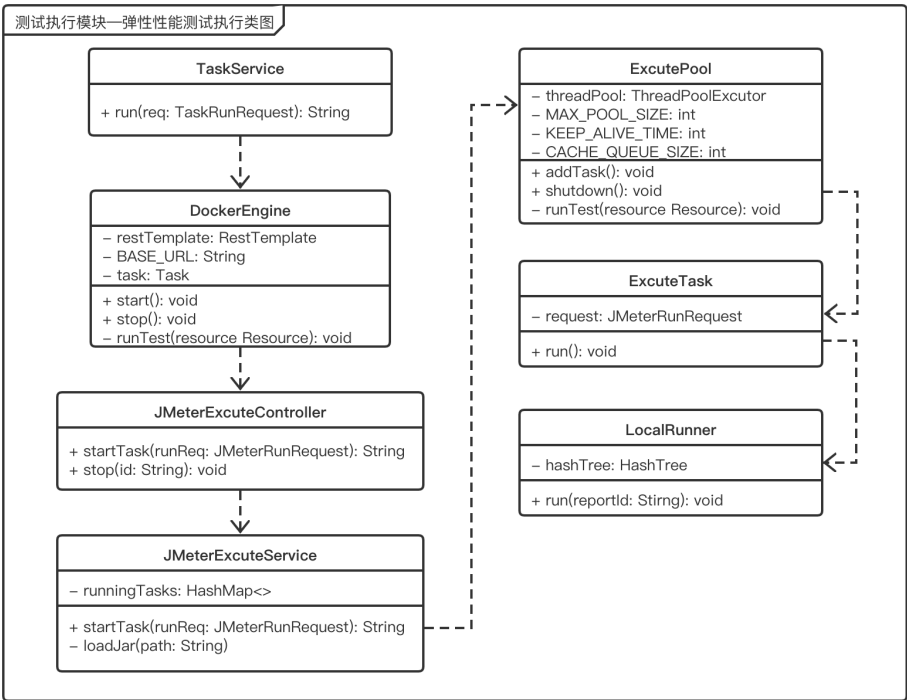


图 4-6: 测试执行模块——弹性资源池性能测试执行类图

图如图 4-6 所示。

TaskService 接收到测试执行请求后调用 run 方法，run 方法中将会以 Task 对象为参数实例化 DockerEngine 类，DockerEngine 类是用于进行 Docker 容器管理的对象，start 方法被调用后将首先请求资源池管理服务启动容器。JMeterExecuteController 定义了测试执行相关接口，包括启动、停止测试任务，JMeterExecuteService 中则负责实现接口功能。

ExecutePool 是系统中的单例类，用于管理测试执行的线程池，由于测试机中可能同时执行多个测试任务，因此需要对线程进行管理，保证测试任务在单个线程中执行，不受其他因素干扰。addTask 是 ExecutePool 中的核心方法，负责向线程池中添加任务至队列中，ExecuteTask 就是实现了 Runnable 接口的线程任务类，需要实现 Runnable 中定义的抽象 run 方法。run 方法的实现中将会构造 LocalRunner 实例，通过 apache.jmeter 提供的 StandardJMeterEngine 类调用 runTest 执行性能测试方法。

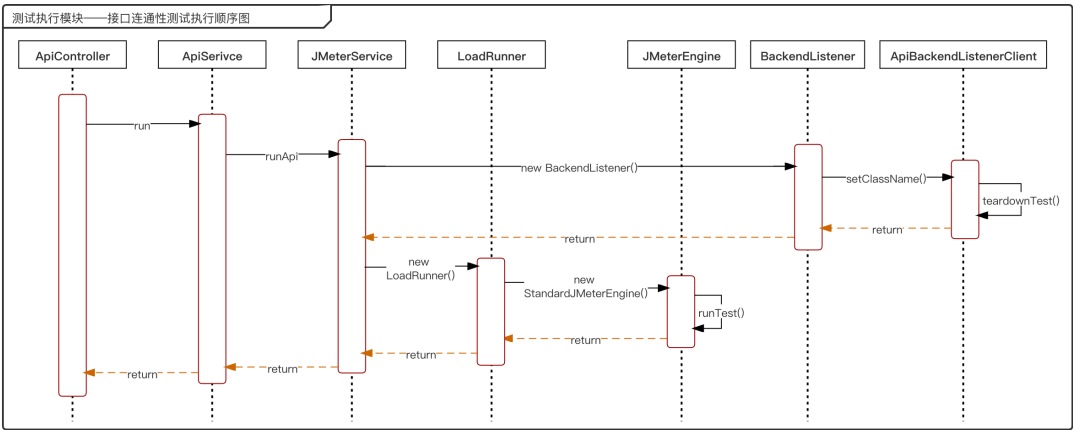


图 4-7: 测试执行模块——接口连通性测试执行顺序图

4.2.3 测试执行模块顺序图

图4-7中展示了接口连通性测试执行的顺序图，ApiController 接收到消息后，依次调用 runApi 方法至 JMeterService 中，JMeterService 在测试执行前首先初始化后端监听器 BackendListener，利用 BackendListener 的 setClassName 可以设置具体的自定义监听类 ApiBackendListenerClient，此类中实现了 teardownTest 结果处理函数。随后，将通过 new 关键词创建 LocalRunner 实例，并调用 LocalRunner 中的 run 方法，利用 StandardJMeterEngine 实例中的 runTest 方法，传递 hashTree 为测试参数执行接口测试。

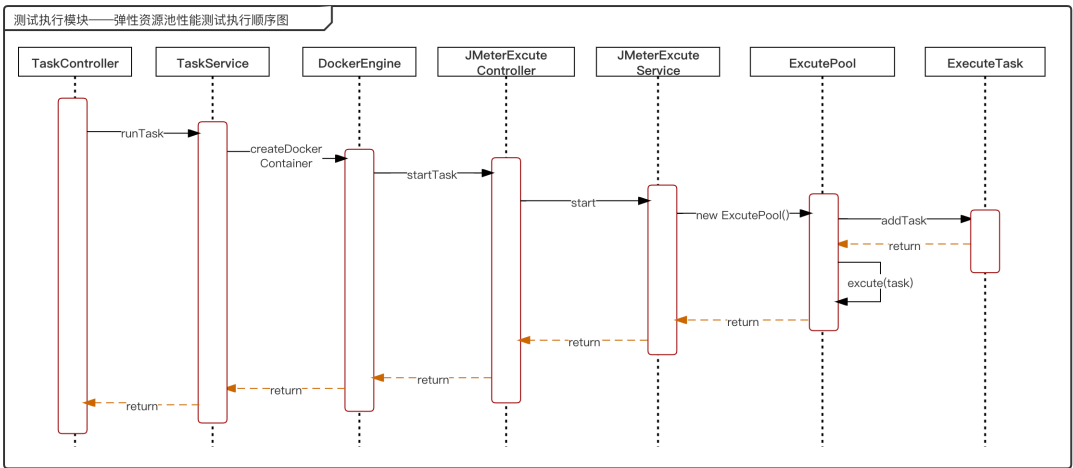


图 4-8: 测试执行模块——弹性资源池性能测试执行顺序图

图4-8中是弹性性能测试执行的顺序图，性能测试执行涉及的 Java 类

众多，但调用逻辑并不复杂。TaskController 接收到性能测试任务执行请求后，通知 TaskService 创建 DockerEngine 实例，DockerEngine 实例利用 HTTP 接口启动容器，待 Docker 容器启动并初始化完成后，利用 HTTP 接口调用 JMeterExecuteController 中定义的测试执行接口，在 JMeterExecuteService 进行下一步的业务逻辑处理。

JMeterExecuteService 的 run 方法将会向 EnginePool 线程池中插入 ExecuteTask，队列更新后调用 ThreadPoolExecutor 提供的 execute 方法执行任务，测试任务完成后调用 shutdown 方法终止线程。

4.2.4 测试执行模块关键代码

代码段 4.4 中是 JMeterService 的核心代码，在 run 方法中接收 taskId 和 InputStream 形式的接口测试配置作为参数，代码中首先利用 SaveService 提供的 loadElement 方法将 stream 解析为 Object，随后再从此对象中获取 HashTree 类型的 testPlan，testPlan 是测试执行必需的参数。

在测试执行前，需要先初始化后端监听器，addBackendListener 则是用以添加后端监听器的方法。首先，利用 new 关键词创建 BackendListener 对象，使用其内部定义的 set 方法设置监听器名称、参数、自定义监听器类。代码中的 setClassname 方法用于将 BackendListener 的具体类设置为 ApiBackendListenerClient。

Listing 4.4: JMeterService 测试执行逻辑代码

```
public void run(String taskId, InputStream stream) {
    init();
    try {
        // 解析场景配置 HashTree
        Object elements = SaveService.loadElement(stream);
        HashTree testPlan = getHashTree(elements);
        // 处理变量提取逻辑
        JMeterVars.addJSR223PostProcessor(testPlan);
        // 添加后端监听器
        addBackendListener(taskId, testPlan);
        LocalRunner runner = new LocalRunner(testPlan);
        runner.run();
    } catch (Exception e) {
```

```
        LogUtil.error(e.getMessage(), e);
    }
}

private void addBackendListener(String taskId, HashTree
    testPlan) {
    BackendListener backendListener = new BackendListener();
    Arguments arguments = new Arguments();
    arguments.addArgument(ApiBackendListenerClient.TASK_ID,
        taskId);
    backendListener.setName(taskId);
    backendListener.setArguments(arguments);
    backendListener.setClassname
        (ApiBackendListenerClient.class.getCanonicalName());
    testPlan.add(testPlan.getArray()[0], backendListener);
}
```

监听器设置完成后，传入 `testPlan` 参数创建 `LocalRunner` 实例，`LocalRunner` 类是负责测试执行的类，其代码如 4.5 所示。此类中持有一个私有对象 `jmxTree`，是 `JMeterEngine` 所需的测试配置。使用 `configure` 方法设置测试计划，最终调用 `JMeterEngine` 的 `runTest` 方法执行测试。

Listing 4.5: LocalRunner 测试执行类代码

```
public class LocalRunner {
    private final HashTree jmxTree;
    public LocalRunner(HashTree hashTree) {
        this.jmxTree = hashTree;
    }
    public void run() {
        JMeterEngine engine = new StandardJMeterEngine();
        engine.configure(jmxTree);
        try {
            engine.runTest();
        } catch (JMeterEngineException e) {
            engine.stopTest(true);
        }
    }
}
```

用于弹性资源池性能测试执行的 JMeterExecuteService 代码如 4.6 所示。startTask 方法首先加载测试所需的依赖包，使用 jar 包的形式引入到系统中。为了减少多次利用 taskId 查询测试配置所带来的开销，以 JMX 文件的形式将场景配置存储到服务器中，以便资源池中的各个测试机器读取 JMX 文件。利用 loadTree 方法从 JMX 文件解析出 hashTree 后，便调用 ExecutePool 中的 addTask 方法，向线程池中加入执行任务。

Listing 4.6: JMeterExecuteService 性能测试执行代码

```
public String startTask(JMeterRunRequest request) {
    try {
        // 加载jar工具与附件（省略）
        // 获取测试场景配置文件
        File jmxFile = new File(FileUtils.BODY_FILE_DIR + "/" +
            request.getReportId() + "_" + request.getTestId() +
            ".jmx");
        // 生成执行脚本
        HashTree testPlan = SaveService.loadTree(jmxFile);
        request.setHashTree(testPlan);
        // 向线程中添加任务
        CommonBeanFactory.getBean(ExecutePool.class).addTask(request);
    } catch (Exception e) {
        return e.getMessage();
    }
    return "SUCCESS";
}
```

ExecuteTask 是实现了 Runnable 接口的线程任务类，此类中实现了 run 方法，代码如 4.7 所示。当 ExecutePool 中的 ThreadPoolExecutor 对象调用 execute 方法时，各个任务类中的 run 方法将被依次调用。run 方法中将会正式执行性能测试任务，其核心逻辑与代码段 4.4 相似，依次进行后端监听器注册与 LocalRunner 的执行。

Listing 4.7: 测试执行线程任务代码

```
public class ExecuteTask implements Runnable {
```

```
private JmeterRunRequest request;

public ExecuteTask(JmeterRunRequest request) {
    this.request = request;
}

@Override
public void run() {
    JMeterBase.addSyncListener(request, request.getHashTree(),
        APISingleResultListener.class.getCanonicalName());
    // 测试执行
    LocalRunner runner = new
        LocalRunner(request.getHashTree());
    runner.run(runRequest.getReportId());
    if (StringUtils.isEmpty(request.getReportId())) {
        if (res == null &&
            !JmeterThreadUtils.isRunning(request.getReportId(),
                request.getTestId())) {
            // 执行结果超时
        }
    }
    LoggerUtil.info("任务: 【 " + request.getReportId() + "
        】 执行完成");
}
}
```

4.3 测试资源管理模块详细设计与实现

4.3.1 测试资源管理模块架构设计

为了在性能测试中负载大规模流量，本系统的核心解决方案是将请求分发至多个测试机器，在后端服务中，将测试机器抽象为测试节点对象，一至多个测试节点组成测试资源池对象。测试资源池模块负责实现测试资源池的创建、编辑、查看操作，此外，当把测试请求分发到各个机器中执行时需要在服务器中根据镜像启动 Docker 容器，因此节点中容器的管理也是测试资源管理模块中的核心部分。

此模块的架构图如4-9所示，后端服务主要负责处理业务逻辑，因此测试

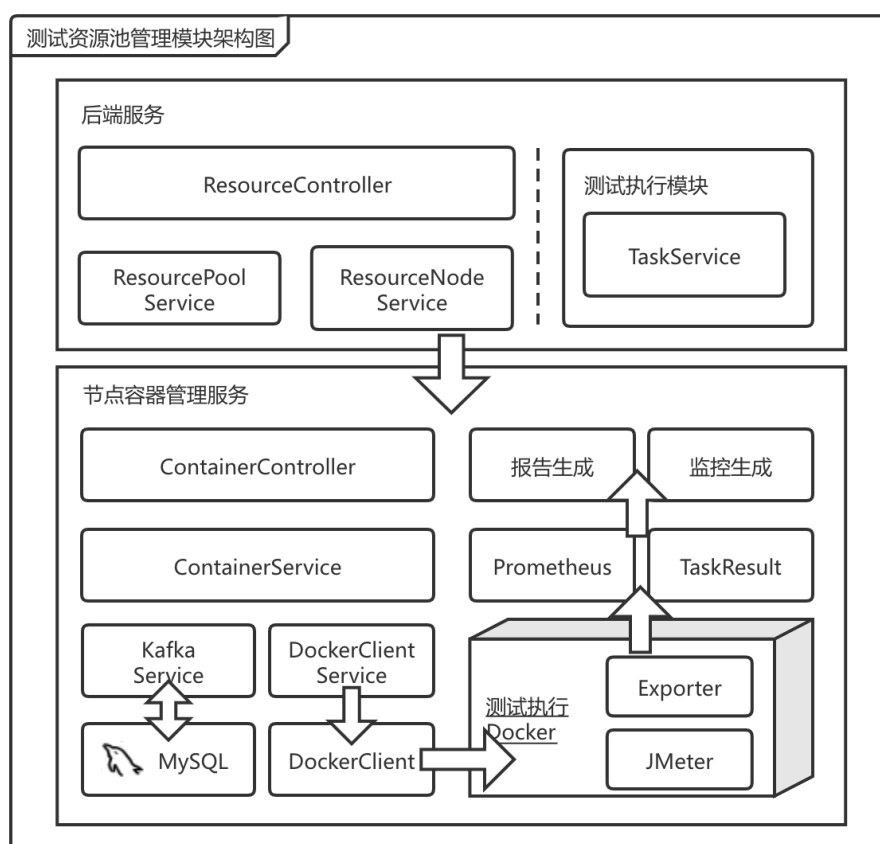


图 4-9: 测试资源池管理模块架构图

资源池对象管理的代码存放在后端服务中。**ResourceController** 中定义了提供给用户调用的资源池管理方法，通过 **ResourcePoolService** 处理测试资源池的业务逻辑，通过 **ResourceNodeService** 进行测试节点的增删改查操作。

节点容器管理服务是资源池管理模块中的核心，由于其逻辑比一般业务代码更为复杂，为降低代码复杂度，提升模块的可移植性，因此将此服务从后端抽离出来，以微服务的形式进行开发。节点容器管理服务通过 **ContainerController** 向后端服务提供启动容器、停止容器、查询容器状态等接口，并通过核心的 **DockerClientService** 类对 **Docker** 容器进行操作。

开源工具 **docker-java**[37] 是利用 **Java** 进行 **Docker** 操作的工具库，架构图中的 **DockerClient** 是 **docker-java** 中提供的 **Docker** 客户端对象，是 **Docker** 容器的抽象。一个 **DockerClient** 实例类将对应一个真实的 **Docker** 容器，**Docker** 容器中包含用于性能测试的 **JMeter** jar 包以及用于收集服务器运行情况的 **Exporter**，**Exporter** 将监控数据传递至 **Prometheus** 服务中。

4.3.2 测试资源管理模块核心类图

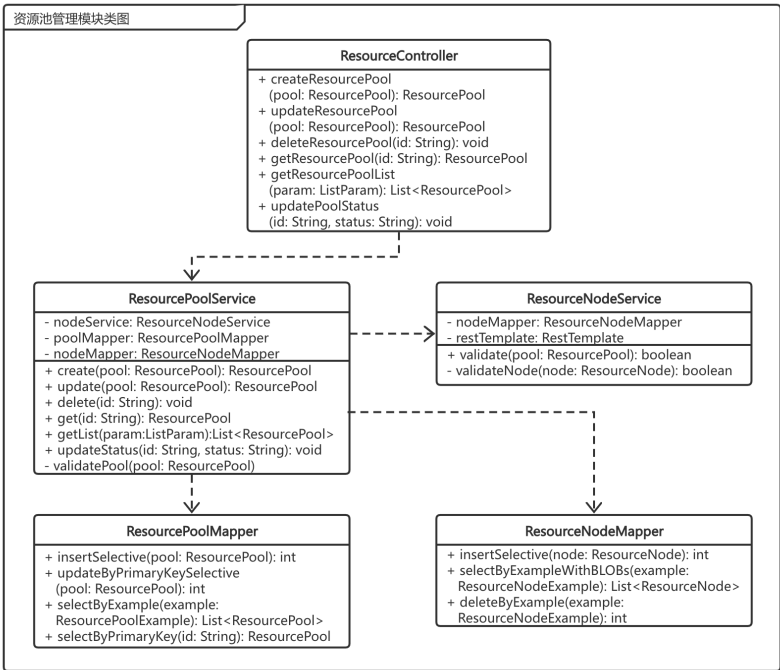


图 4-10: 资源池管理模块类图

后端服务中的资源池管理类如图 4-10 所示，ResourceController 中定义了创建、查询、更新、删除测试资源池的方法，此外还提供了更新测试资源池状态的方法，以控制测试资源池的启用或禁用状态。ResourceService 除了实现 Controller 中提供的方法外，提供了私有的 validatePool 方法，在 create 方法中进行调用，用于验证资源池可用性。ResourcePoolMapper 是测试资源池的数据库操作类，作为 ResourceService 的私有对象，提供给业务逻辑代码使用。

ResourceNodeService 是测试节点相关的业务逻辑，其提供的核心方法是 validte，即验证该测试节点是否有效。ResourceNodeMapper 则是用于测试节点增删查改的操作类，被 ResourcePoolService 直接调用，在资源池创建、更新的同时更新测试节点。

图 4-11 展示了此模块中节点容器管理的类图。ResourceController 为外部提供了三个接口，分别是：启动容器并执行测试、停止测试、获取测试状态，资源池中的各个机器将通过资源池管理模块调用这些接口来执行性能测试任务。而 ResourceService 则用于实现具体的业务逻辑，startNodeContainer 方法最

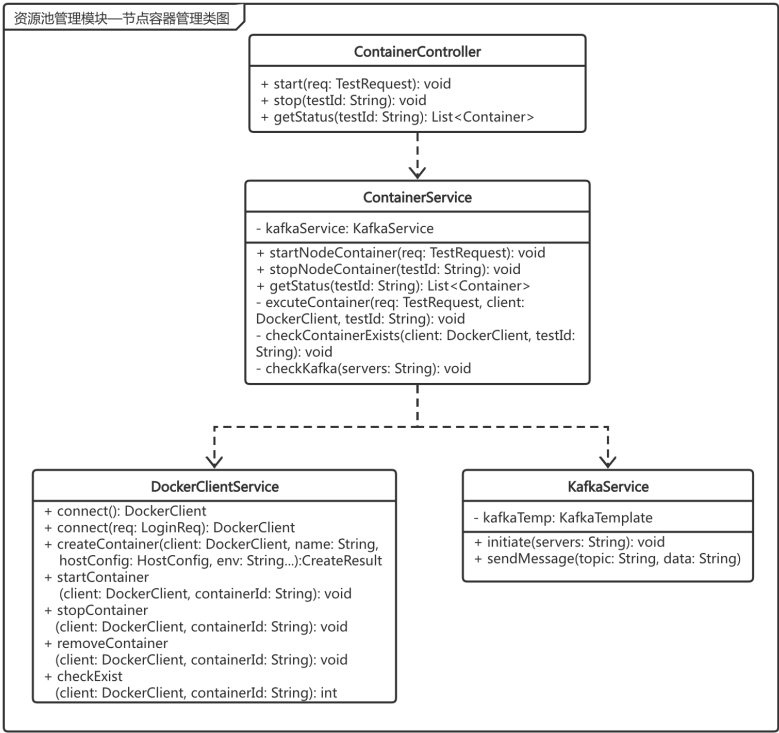


图 4-11: 资源池管理模块——节点容器管理类图

为复杂，用于启动容器并执行测试，其需要调用 `ResourceService` 类中定义的 `checkContainerExists`、`checkKafka`、`startAndExecuteContainer` 私有方法进行预检测、启动与执行操作。

`DockerClientService` 负责对 `DockerClient` 进行管理，包括了 Docker 连接、创建、执行、停止、删除等操作。值得一提的是，系统为测试人员提供了默认的资源池，即直接在后端服务器中执行测试，当测试任务配置了默认资源池时，便会调用多态方法 `connect` 中无参数的实现，通过启动默认的 Docker 来执行测试。

`KafkaService` 是负责进行消息传递的业务处理模块，向外部提供了 `init` 工具初始化、`sendMessage` 消息发送方法。当启动容器时，在容器内部初始化 Kafka 工具，当性能测试任务在容器内开始执行时，监听接收到的测试结果并利用 `sendMessage` 将消息传递至 `Consumer` 中进行处理，为保证该模块的低耦合度，将数据处理相关逻辑整合在测试报告生成模块中。

4.3.3 测试资源管理模块顺序图

由于后端服务中的资源池管理功能并不复杂，其消息通信流与其他业务功能的实现相似，即 Controller——Service——Mapper。因此顺序图中仅描述了节点容器管理功能中的消息交互，如图 4-12 所示。

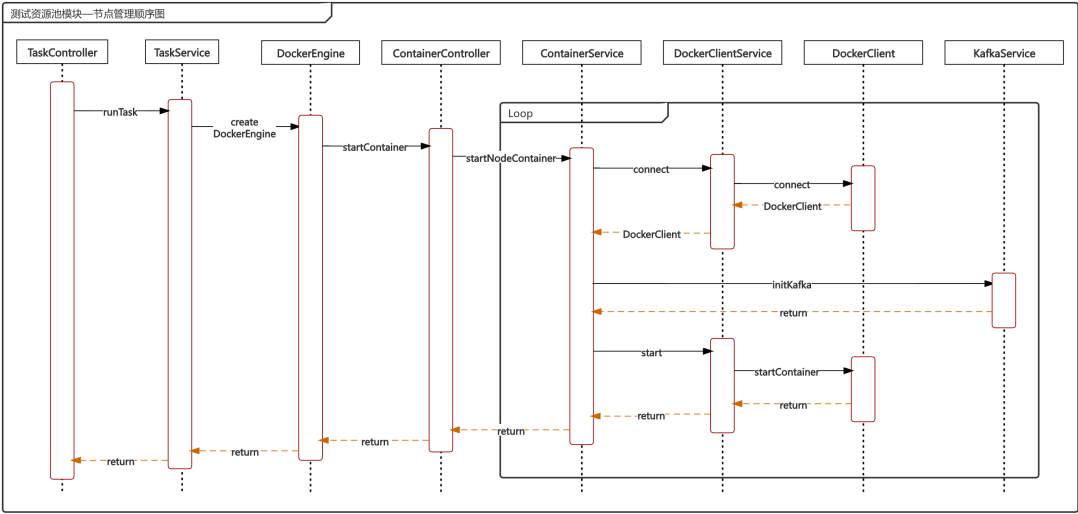


图 4-12: 资源池管理模块——节点容器管理顺序图

节点容器管理的前置条件是性能测试任务中配置了测试资源池，当测试任务开始执行时，将会首先调度节点容器管理功能。TaskService 接收到执行测试请求后，将实例化 DockerEngine 类，在 DockerEngine 类的 start 方法中调用节点容器管理服务的请求，ContainerController 通过 startContainer 方法启动各个测试节点中的 Docker 容器。

startContainer 方法中将遍历 TaskRequest 中的资源池节点对象，在循环中依次使用测试节点配置调用 ContainerService 的 startNodeContainer 方法分别启动用于测试的 Docker 容器。在 startNodeContainer 方法中，首先调用 DockerClientService 提供的连接容器的 connect 方法，此方法将返回 DockerClient 对象。在正式启动 Docker 容器前，将初始化 KafkaService 服务，在容器执行期间 Kafka 用于传递容器内部日志消息。工具初始化完成后，便调用 DockerClientService 中的 startContainer 方法启动容器。

4.3.4 测试资源管理模块关键代码

代码段 4.8 是后端服务中用于管理容器的 `DockerEngine` 类，类中 `start` 方法是用于启动所有测试节点中 `Docker` 容器的公共方法。当创建 `DockerEngine` 实例时，需要传入测试任务 `Task` 对象作为构造函数的参数，随后通过 `init` 方法设置私有变量 `restTemplate`、`nodeList` 的值。

当 `start` 方法被调用时，将遍历 `nodeList` 列表，对每一个 `node` 执行 `runTest` 方法。`runTest` 方法中首先解析 `ResourceNode` 中的 `configure` 字段，将其设置为 `startTestRequest` 的环境参数，随后使用 `node` 中的 `ip`、端口号替换 `BASE_URL` 字符串模版生成实际 URL，利用 `RestTemplate` 对象调用节点容器管理服务的 HTTP 方法，最后判断响应结果并在调用错误时抛出异常。

Listing 4.8: `DockerEngine` 容器启动类核心代码

```
public class DockerEngine{
    private static final String BASE_URL = "http://%s:%d";
    private RestTemplate restTemplate;
    private List<ResourceNode> nodeList;
    private Task task;
    public DockerEngine(Task task) {
        // init方法用于初始化restTemplate, 获取nodeList等
        this.init(task);
    }
    public void start() {
        for (int i = 0, size = this.nodeList.size(); i < size;
            i++) {
            runTest(nodeList.get(i), i);
        }
    }
    private void runTest(ResourceNode node, int resourceIndex) {
        // 配置初始化（省略）
        StartTestRequest startTestRequest = new StartTestRequest();
        startTestRequest.setImage(JMETER_IMAGE);
        startTestRequest.setEnv(env);
        // 调用节点容器管理服务启动容器
        String uri = String.format(BASE_URL +
            "/jmeter/container/start", node.getIpAddress(),
```

```
node.getPort());  
ResultHolder result = restTemplate.postForObject(uri,  
    startTestRequest, ResultHolder.class);  
if (result == null || !result.isSuccess()) {  
    throwException(result.getMessage());  
}  
}  
}
```

代码段 4.9 则是节点容器管理服务中启动容器的业务逻辑方法，`startContainer` 方法接收 `StartTestRequest` 类型的参数，首先，将 `request` 中设置的环境配置参数解析为 `Map`，随后调用 `KafkaService` 提供的初始化方法在容器中进行工具初始化。在启动 `Docker` 容器前，需要通过 `DockerClientService` 的 `connect` 方法创建 `dockerClient` 对象，与 `Docker` 容器建立连接，在检查容器有效性后调用私有的 `start` 方法，对 `DockerClient` 对象进一步处理。

Listing 4.9: ContainerService 节点容器创建代码

```
public void startContainer(StartTestRequest testRequest) {  
    Map<String, String> env = testRequest.getEnv();  
    String testId = env.get("TEST_ID");  
    String bootstrapServers = env.get("BOOTSTRAP_SERVERS");  
    // 检查kafka连通性  
    checkKafka(bootstrapServers);  
    // 初始化kafka  
    kafkaService.init(bootstrapServers);  
    DockerClient dockerClient =  
        DockerClientService.connectDocker(testRequest);  
    // 检查容器是否存在  
    checkContainerExists(dockerClient, testId);  
    // 启动测试  
    start(testRequest, dockerClient, testId);  
}  
  
private void start(TestRequest testRequest, DockerClient  
    dockerClient, String testId) {  
    // 创建 hostConfig
```

```
HostConfig hostConfig = HostConfig.newHostConfig();
hostConfig.withNetworkMode("host");
String[] envs = getEnvs(testRequest);
// 创建并启动容器
String containerId = DockerClientService
    .createContainers(dockerClient, testId,
        testRequest.getImage(), hostConfig, envs)
    .getId();
DockerClientService.startContainer(dockerClient, containerId);
LoggerUtil.info("Container create started containerId: " +
    containerId);
// 根据Docker日志记录异常
dockerClient.logContainerCmd(containerId)
    .withFollowStream(true)
    .withStdOut(true)
    .withStdErr(true)
    .withTailAll()
    .exec(new
        InvocationBuilder.AsyncResultCallback<Frame>() {
            @Override
            public void onNext(Frame item) {
                // 内存异常情况处理
                String log = new String(item.getPayload()).trim();
                String oomMessage = OOM_MESSAGE;
                if (StringUtils.contains(log, oomMessage)) {
                    // oom 退出
                    String[] contents = new String[]{reportId,
                        "none", "0", oomMessage};
                    String message = StringUtils.join(contents, "
");
                    kafkaService.sendMessage(topic, message);
                }
            }
        });
}
```

start 方法接收 StartTestRequest、DockerClient、testId 作为参数，在启动容

器前从 `StartTestRequest` 对象中解析字段构造创建容器所需的 `HostConfig` 和 `envs` 环境参数。接着调用 `DockerClientService` 封装的 `create` 和 `start` 方法在服务器中创建并启动容器，在此类关键事件节点利用 `LoggerUtils` 将日志打印出来，有利于测试人员了解测试进程，在测试出现异常时也更便于错误排查。

在完成对 `Docker` 容器的创建和启动后，还需要进行两个额外操作：一是获取 `Docker` 容器的运行日志，筛选并记录重要的执行异常；二是监听容器是否已经退出，若处于未使用状态则退出容器。

`logContainerCmd` 是 `DockerClient` 类提供的日志监听接口，在 `exec` 中创建用于异步获取容器日志的 `AsyncResultCallback` 实例，通过重写实例中的 `onNext` 方法获取日志详情。`Frame` 用于表示一个日志框架，利用 `getPayload` 获取字节流并将其转换为字符串形式，通过 `StringUtils` 提供的 `contains` 方法判断日志中是否存在内存相关异常，若存在异常则调用 `KafkaService` 提供的 `sendMessage` 方法将此日志传递给后端服务器进行持久化存储。

4.4 测试数据处理模块详细设计与实现

4.4.1 测试数据处理模块架构设计

当性能测试开始执行后，根据测试人员的配置向资源池中各个节点施加压力，为了获取实时的测试报告，需要在测试执行过程中将测试结果实时传送至报告生成模块。但由于大规模流量下的测试结果数据量巨大，若使用 `HTTP` 进行数据传输，在请求体中携带大量数据的情况下传输速率将大幅降低，影响测试人员正常查看实时测试报告。因此本系统中选用 `Kafka` 作为连通测试资源池与后端服务的消息传递工具，测试数据处理模块便是基于 `Kafka` 进行实现。

测试数据处理模块是用于收集测试数据，并对数据进行预处理的模块，其架构图如 4-13 所示。由于 `Kafka` 只需利用 `Broker` 作为消息中转站，而无需考虑生产者和消费者的实际部署载体，非常适合多机器、分布式的系统架构。在测试资源池与后端服务默认机器中，每一个能够用于测试的 `Docker` 节点中均定义了 `Producer` 类，用于向 `Kafka Broker` 里传递数据。`Kafka` 中根据功能划分了 `Data`、`Log`、`Report` 等主题，各个主题下传递相同类型的数据，通过对应类型的 `Consumer` 进行消息获取。

数据的接收和处理逻辑收敛在数据处理服务中，在服务中定义了多个用于监听并接收数据的 `Consumer`，其中 `DataConsumer` 负责接收测试结果数据，

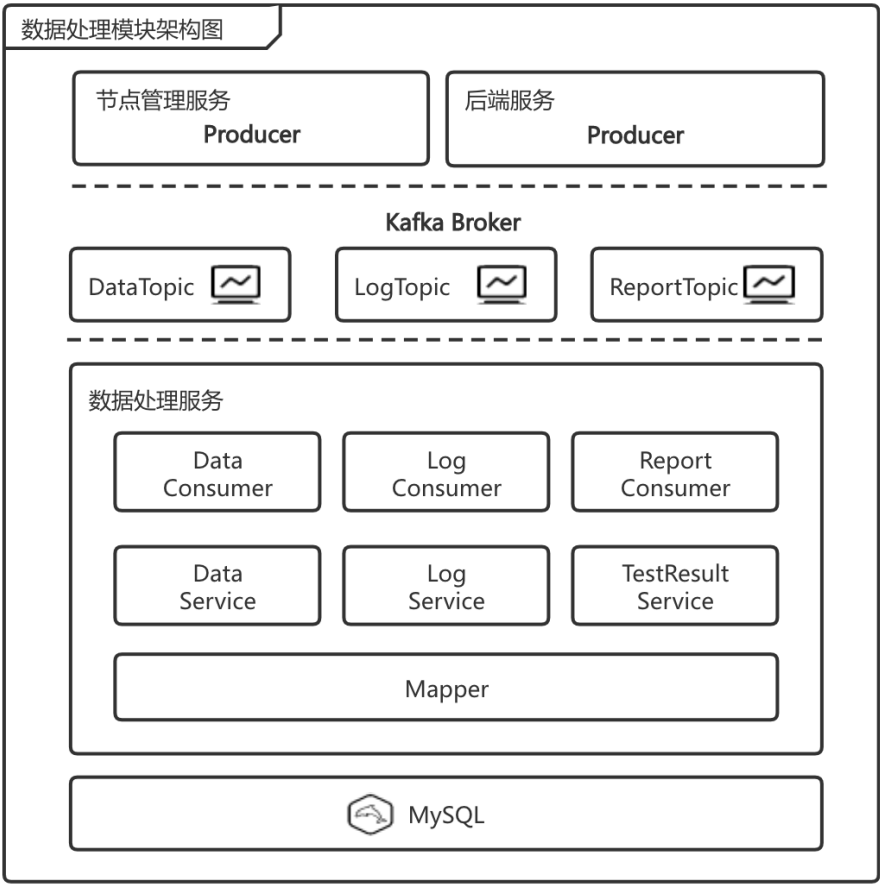


图 4-13: 测试数据处理模块架构图

LogConsumer 负责接收测试日志数据，ReportConsumer 负责接收测试报告数据。Consumer 接收到 Record 记录后，调用相应的 Service 进行业务逻辑处理，当测试结果处理完成后，使用 MyBatis 提供的 Mapper 将数据存储至数据库中，随后报告生成模块便能够直接从数据库获取报告数据并展示在前端页面中。

4.4.2 测试数据处理模块核心类图

图 4-14 是测试数据处理模块的类图，DataConsumer、LogConsumer、ReportConsumer 是该模块的核心类，类图将基于这三个类进行展开。DataConsumer 类中的 consume 方法需要被 KafkaListener 注解标记，是用于接收监听数据的方法，此方法中的参数 ConsumerRecord 用来描述对应 group 和 topic 中的数据，通过 ConsumerRecord.value() 即可获取接收到的数据，并将数据加入 MetricQueue 队列中。Metric 是描述 JMeter 测试数据的类，用于表示 Consumer

接收到的初始数据。handleQueue 将在 DataConsumer 类构造完成后被调用，用于处理 MetricQueue 队列，在新线程中获取当前 MetricQueue 队列的队头元素，将队头元素插入 MetricList 中。handleSave 方法在 handleQueue 调用完成后执行，负责再一次对 MetricQueue 和 MetricList 进行判断，保证测试结果被正确存储。

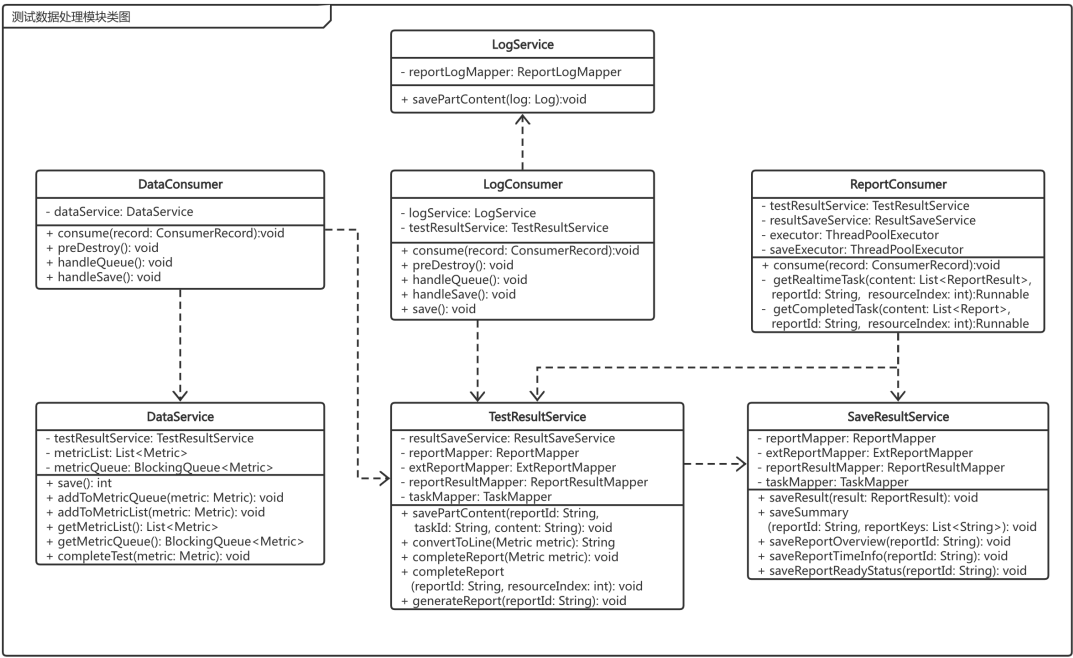


图 4-14: 测试数据处理模块类图

DataService 是负责 Metric 数据处理的服务类，被 DataConsumer 调用。在 DataService 类中维护了 MetricQueue 与 MetricList，MetricQueue 是 BlockingQueue 类型的数据队列，每当 Consumer 接收到新消息，将会调用 Service 中的 addToMetricQueue 方法将数据插入队列。MetricList 中维护了待处理的数据列表，队列中的 Metric 将被依次加入列表，当列表达到一定容量时（系统设置为 1000），将调用 save 方法使用 testResultService 的接口保存元数据，并清理 MetricList 中的已处理数据。

TestResultService 是 Metric 元数据与测试报告数据的中转站，负责从 DataService 接收测试结果元数据，并调用 generateReport 生成测试报告，报告数据生成后调用 SaveResultService 保存测试报告。savePartContent 是暴露给外部的数据存储方法，将元数据存储到数据库中，并开始进行测试报告数据的生成。报告数据处理完成后，使用 SaveResultService 将测试报告结果存储至数

数据库中。

4.4.3 测试数据处理模块流程

本模块的顺序图如4-15所示，以 Kafka Broker 为界被分为两部分，左侧是负责发送数据的 Producer，右侧是负责并处理数据的 Consumer。在测试执行模块中的 ExecuteTask 开始执行任务前，将会首先根据配置对 Kafka 进行初始化，确认与 Kafka Broker 的连通性。进而在任务执行时监听测试执行过程并调用 KafkaService 的 sendMessage 方法发送数据，KafkaService 负责创建 KafkaTemplate 并发送消息。

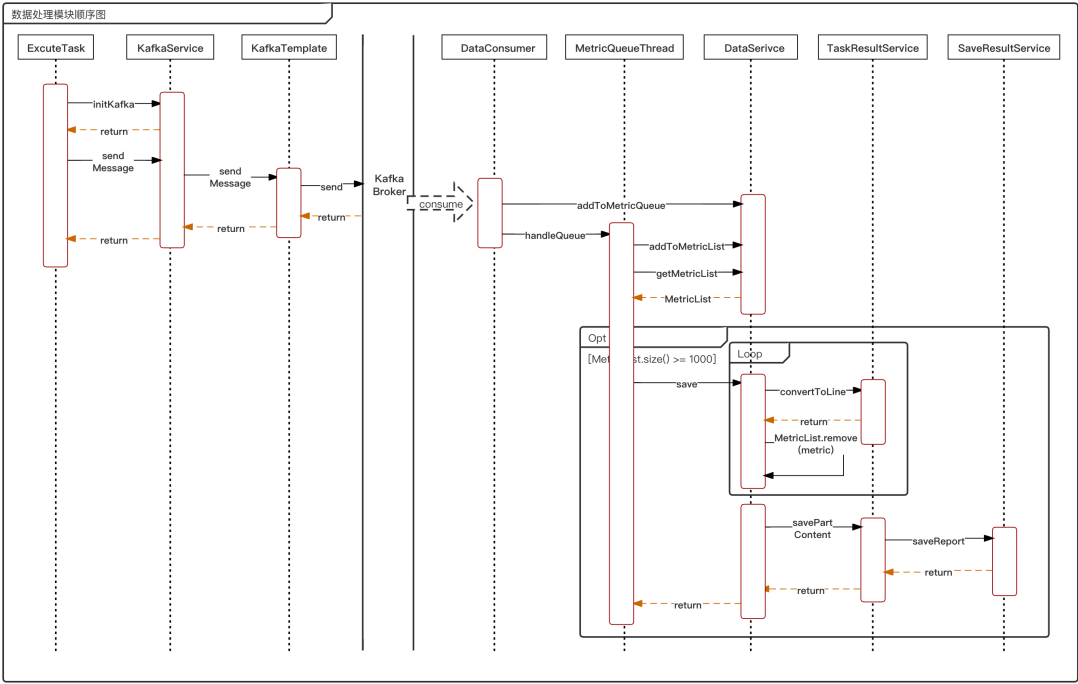


图 4-15: 测试数据处理模块顺序图

右侧部分是以 DataConsumer 为核心的测试数据接收与处理过程，DataConsumer 中维护了私有变量 isRunning，用于标记当前 Consumer 是否处于运行状态。在 DataConsumer 中通过实现 consume 方法获取到测试结果元数据 Metric，每当 consume 中接收到数据时，会将 Metric 插入到 DataService 维护的 MetricQueue 中。

随后标记了 @PostConstruct 注解的 handleQueue 方法被调用，此方法中将会启动一个新线程，在顺序图中称之为 MetricQueueThread。在 Metric-

QueueThread 线程中，取 MetricQueue 队列的第一个元素插入至 MetricList 的末尾，当 MetricList 的元素数量达到 1000 时，调用 DataService 中的 save 方法。为了梳理 MetricQueue 与 MetricList 的关系与作用，绘制了图 4-16 中的简易流程图，图中展示了 MetricQueue 和 MetricList 的逻辑流程，Metric 元素会首先加入队列而后在 MetricList 中等待处理，为了不过于频繁地处理数据，因此设置了最大队列规模为 1000，队列与列表将始终保持着元素平衡进出。

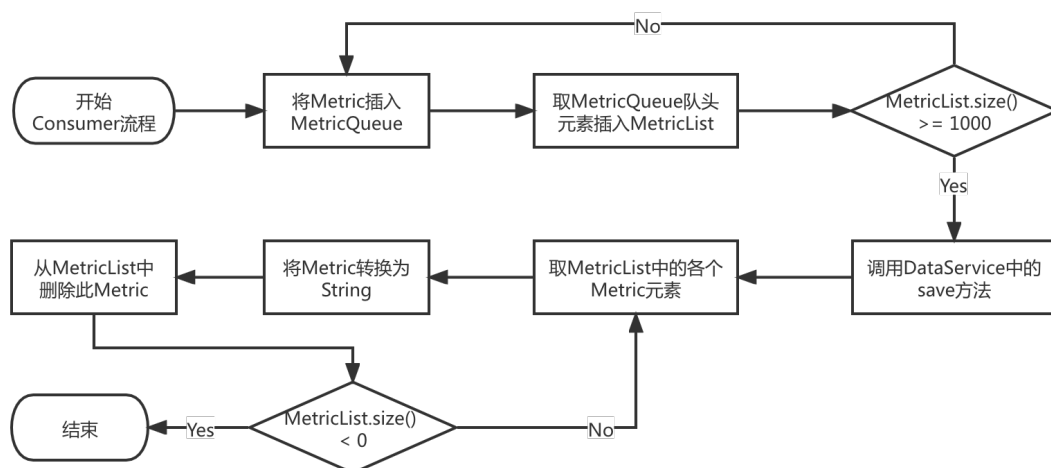


图 4-16: Metric 元数据处理流程图

在 DataService 的 save 方法中，首先对 MetricList 中的所有元数据进行遍历，通过调用 TestResultService 的 convertToList 将每一个 Metric 数据转换为 String 形式，利用 savePartContent 将字符串数据进行存储。savePartContent 方法中将执行 generateReport 方法，此部分的逻辑将在测试报告生成模块中进行详细设计，在此不做展开。

4.4.4 测试数据处理模块关键代码

代码段 4.10 为 DataConsumer 类的核心代码，consumer 是被注解的 KafkaListener 方法，用于接收配置中值为 kafka.topic 主题的数据。在 consume 方法中，首先利用 MyBatis 提供的对象处理类 ObjectMapper 将 record 值转换为 Metric 数据类，接着根据 Metric 中的 completed 值判断当前测试是否结束，若测试未结束，则调用 add 方法将当前接收到的 Metric 数据插入 MetricQueue 队列。若测试已结束则最后执行一次 DataService 中的 save 方法，保证所有元数据均被处理，随后执行 DataService 中的 completeTest 方法，标识测试已经完成，开始测

试报告生成。

Listing 4.10: 基于 Kafka 的测试结果数据消费者代码

```
public class DataConsumer {
    @Resource
    private ObjectMapper objectMapper;
    @Resource
    private DataService dataService;
    @KafkaListener(id = "metric-data", topics = "${kafka.topic}",
        groupId = "${spring.kafka.consumer.group-id}")
    public void consume(ConsumerRecord<?, String> record) throws
        Exception {
        Metric metric = objectMapper.readValue(record.value(),
            Metric.class);
        if (BooleanUtils.toBoolean(metric.getCompleted())) {
            dataService.save();
            dataService.completeTest(metric);
            return;
        }
        dataService.addToMetricQueue(metric);
    }
}
```

DataService 中维护了用于控制待处理数据的 MetricQueue 和 MetricList 变量，其代码如代码段 4.11 所示。在此服务类中，定义了最大等待队列容量 QUEUE_SIZE 为 1000。在初始化 metricList 时使用了 CopyOnWriteArrayList 类，此类是 ArrayList 的线程安全版本 [38]，用于保证此数组在生命周期中永远不会受到干扰与影响，保障内存一致性。而用于表示队列的 BlockingQueue 则是 Java 中提供的线程安全的队列访问方式——阻塞队列，当阻塞队列插入数据时，若队列已满，线程将会阻塞等待直到队列中有元素被删除。

Listing 4.11: 测试数据处理模块核心业务代码

```
public class MetricDataService {
    public static final Integer QUEUE_SIZE = 1000;
    private final List<Metric> metricList = new
        CopyOnWriteArrayList<>();
}
```

```
private final BlockingQueue<Metric> metricQueue = new
    ArrayBlockingQueue<>(QUEUE_SIZE);
@Resource
private TestResultService testResultService;

public synchronized int save() {
    int size = metricList.size();
    Map<String, List<Metric>> reportMetrics =
        metricList.stream()
            .collect(Collectors.groupingBy(Metric::getReportId));
    reportMetrics.forEach((reportId, metrics) -> {
        String testId = "";
        StringBuilder content = new StringBuilder();
        for (Metric metric : metrics) {
            content.append
                (testResultService.convertToLine(metric));
            testId = metric.getTestId();
            metricList.remove(metric);
        }
        testResultService.savePartContent(reportId, testId,
            content.toString());
    });
    return size;
}

public void addToMetricQueue(Metric metric) throws
    InterruptedException {
    metricQueue.put(metric);
}
}
```

save 方法中首先对 metricList 进行处理，利用 collect 方法将列表解析为 reportId 为 key、Metric 为 value 的 Map 结构，接着对 Map 进行遍历，即将每一个测试进程中的 Metric 通过 convertToLine 方法转化成字符串，并拼接到 content 中，最终 content 将包含一个测试进程中的所有测试结果。convertToLine 方法的逻辑是将 Metric 中的各个字段通过 getter 获取出来，并利用 StringBuilder 的 append 方法，将各个字段按照特定顺序，组合为利用“,”符号分隔的字符串。

testResultService.savePartContent 方法将测试结果 content 进行存储，并开始测试报告生成。

4.5 测试报告生成模块详细设计与实现

4.5.1 测试报告生成模块架构设计

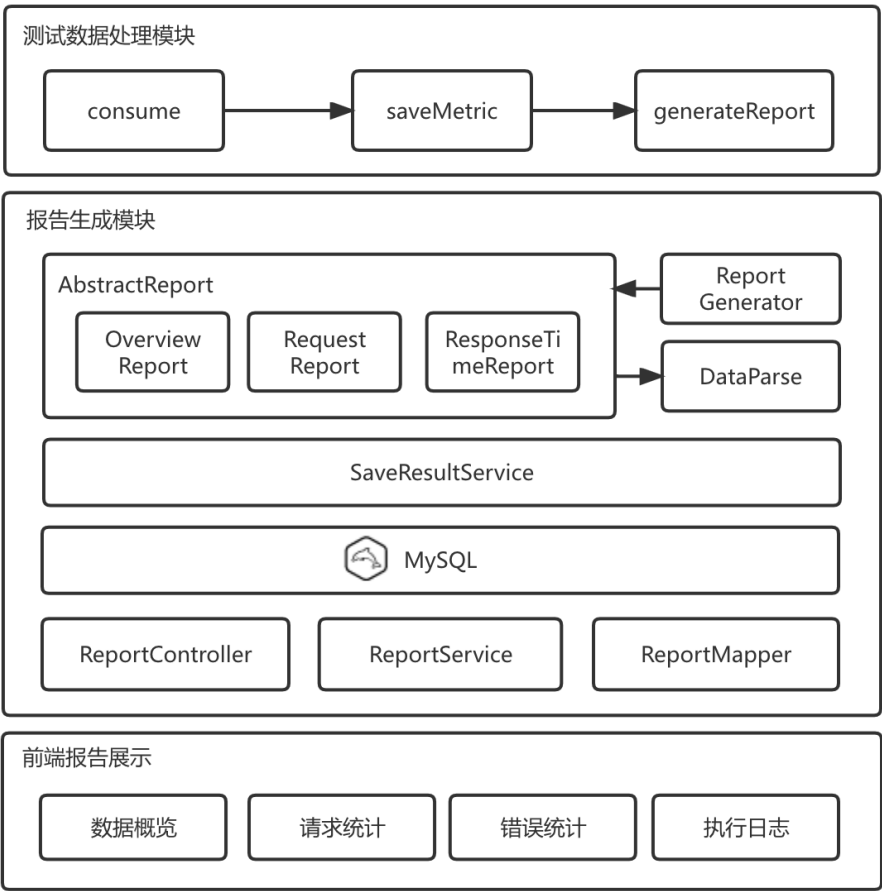


图 4-17: 测试报告生成模块整体架构图

当测试开始执行后，将通过数据处理模块获取 JMeter 中产生的测试结果数据，并将数据转换为测试报告中所需的数据格式。由于性能测试数据具有规模大、指标复杂的特点，因此本系统需要从结果数据中筛选出有意义的指标数据，并以图表、数据卡片、统计表格的形式将测试结果展现给用户，从而测试人员能够清晰地得知待测软件的性能表现。

图 4-17 中展现了测试报告生成模块的整体架构，数据处理模块是报告生成模块的前置模块，用于为报告生成提供数据，前端报告展示模块为本模块的后置模块，负责将生成后的数据展示到前端页面中。首先，在测试数据处理模块中通过 `consume` 监听并获取到测试结果元数据后，将调用 `generateReport` 方法开始报告生成流程。

在报告生成模块中，依赖 `ReportGenerator` 类来生成不同类别的报告，所有类别的报告均是 `AbstractReport` 的子类，图中的 `OverviewReport`、`RequestReport`、`RTChartReport` 仅是用于描述设计的典型报告，系统中的报告类别高达 15 个。所有类别的具体类 `Report` 均实例化完成后，将会循环调用 `DataParse` 中提供的数据转换方法，将测试结果转换为图表数据或概览数据，最后使用 `SaveResultService` 将报告数据存储至 MySQL 数据库中。

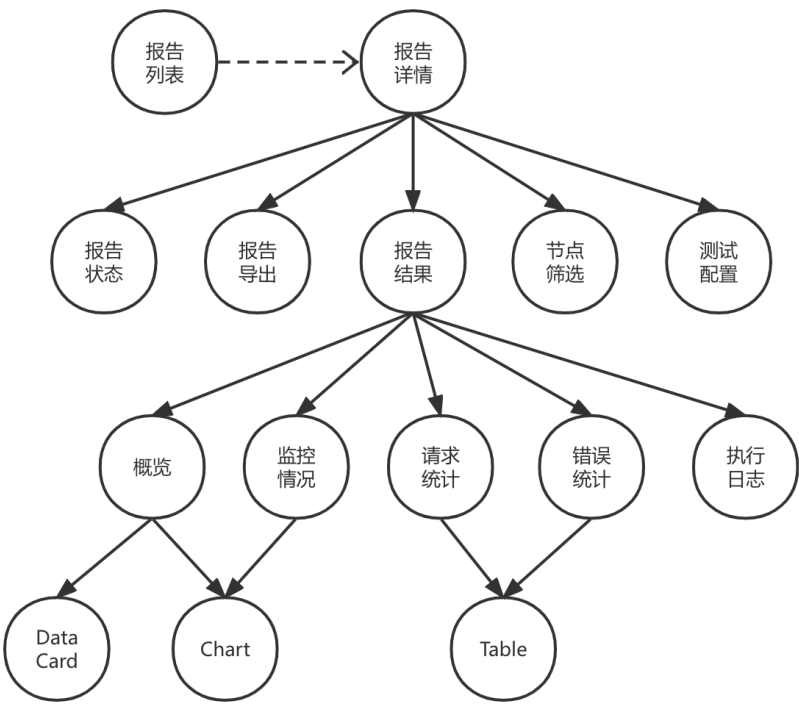


图 4-18: 测试报告生成模块前端组件结构图

为了使前端能够获取到数据，报告生成模块使用 `ReportController` 类定义接口，为测试人员提供查看、编辑报告的功能。最终在前端报告展示时将测试报告划分为四个部分，分别是数据概览、请求统计、错误统计与执行日志，以便于测试人员查看不同类别的测试结果。由于测试报告模块的前端展示较为复杂，绘制了前端组件结构图 4-18 以说明前端组件的分解与组合设计。

报告列表页面是报告详情页面的入口，负责以列表形式展示所有生成的报告。报告详情页面由报告状态、报告导出、节点筛选、报告结果、测试配置五个组件组成，其中节点筛选用于选择查看特定测试资源池中测试节点的报告，测试配置用于展示该性能测试的压力配置情况，而报告结果组件则用于展示测试结果，是测试报告的核心。报告结果组件由概览、监控情况、请求统计、错误统计、执行日志四个组件构成，概览中包括各个不同的指标数据和指标变化图表，这些数据的展示由 DataCard 和 Chart 组件负责，请求统计与错误统计则是利用 table 组件以表格的形式展示。

4.5.2 测试报告生成模块核心类图

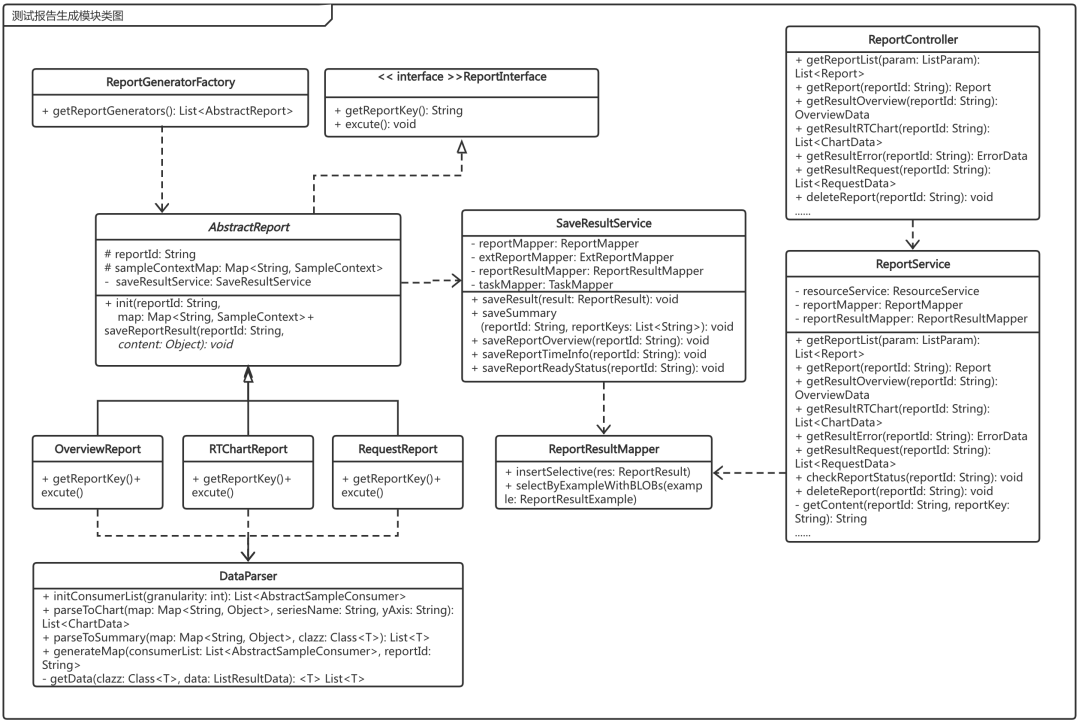


图 4-19: 测试报告生成模块类图

该模块的类设计如图4-19所示。ReportGeneratorFactory 类通过暴露 getReportGenerator 方法，返回 List<AbstractReport>，即所有类别的报告实例。AbstractReport 是所有类别报告的高层抽象，实现了 ReportInterface 接口，并且提供了 init 和 saveResult 方法。init 方法通过接收 reportId 与 sampleContentMap 对 Report 进行初始化，而 saveResult 则负责将数据构造为 ReportResult 对象并

调用 `SaveResultService` 中的 `saveResult` 方法将报告结果数据存储至数据库中。继承了 `AbstractReport` 的子类需要实现 `ReportInterface` 接口定义的两个方法，`getReportKey` 返回该报告名称，`execute` 负责获取 `sampleContextMap` 中的 `SampleContext` 并调用 `DataParse` 中的数据转换方法获取报告结果数据。`DataParse` 是用于数据转换的类，其中 `parseToChart` 负责将测试数据转换为折线图数据，`parseToSummary` 负责将测试数据转换为概览形式的平均或统计数据。

`ReportController` 是提供给前端的接口定义类，其中定义了获取报告列表、获取报告信息、删除报告，此外还有获取所有类别报告结果的接口，因数量较多并未全数展示在类图中。`ReportService` 类负责实现接口的业务逻辑，通过 `ReportMapper` 与 `ReportResultMapper` 等数据库交互类与 MySQL 数据库通信，此外，为了便于获取各个类别的报告结果，`ReportService` 中定义了私有方法 `getContent`，负责根据 `reportId` 与 `reportKey` 获取结果字符串，字符串的处理则由各个具体方法来实现。

4.5.3 测试报告生成模块顺序图

由于本模块与前端的交互时许较为简单，因此顺序图 4-20 中仅描述了测试报告生成与存储的信息交互情况，省略了 `ReportController` 到前端的交互。

首先，`TestResultService` 在获取并处理完成 `Consumer` 监听到的数据后，将会调用 `checkStatus` 方法检查报告状态，若报告已经在处理中则不会执行报告生成。若报告状态为未开始，`TestResultService` 将调用 `ReportGeneratorFactory` 提供的 `getGenerator` 方法获取生成的 `Report` 列表，随后对报告列表进行遍历，一次循环开始时首先对报告进行初始化，初始化完成后便调用各个具体类实现的 `execute` 方法对报告结果进行计算，计算过程将依赖 `DataParse` 提供的 `parse` 转换方法，当报告结果计算完成后，利用 `SaveResultService` 提供的 `saveReportResult` 方法将报告结果以 `ReportResult` 类的形式存至数据库中。循环执行完成即代表着该报告中所有类别的子报告均已成功生成，最终需要调用 `saveInfo` 方法对报告生成的结束时间、报告状态等数据进行更新，至此，系统生成了一个完整的测试报告并保存在数据库中。

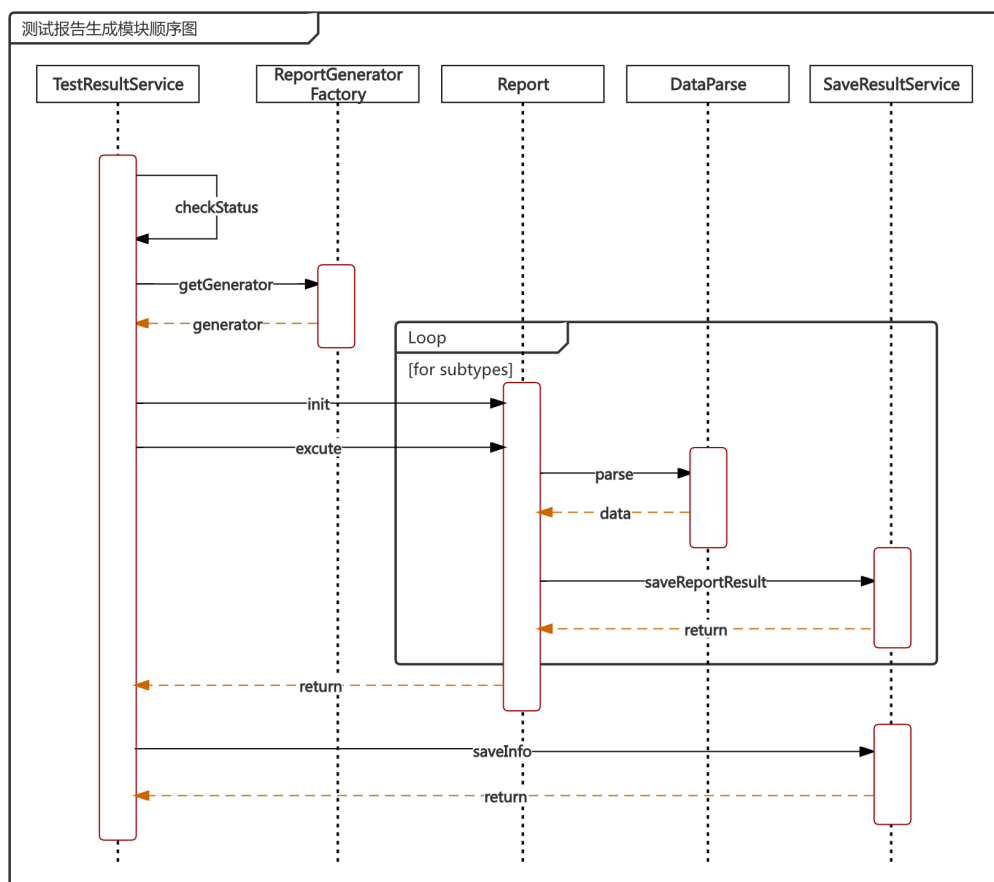


图 4-20: 测试报告生成模块顺序图

4.5.4 测试报告生成模块关键代码

本节中选取了测试报告生成模块中从 `generateReport` 到调用 `execute` 执行 `Report` 的相关代码，生成报告的入口方法 `generateReport` 的代码实现如 4.12 所示。此方法中传入 `reportId` 作为参数，首先调用 `getReportGenerators` 方法实例化所有类别的 `Report` 对象，使用其父类 `AbstractReport` 表示。由于各个 `Report` 对象的处理执行计划在线程池中执行，并且根据时序图，需要在所有 `Report` 对象处理完毕后保存部分报告信息，因此利用计数为 `reportGenerators` 元素个数的 `CountDownLatch` 对象来保证所有线程均已执行完成。随后，通过 `DataParse` 中的 `computeReport` 方法构造一个以测试结果类名为 `key`，`SampleContext` 为 `value` 的 `Map`，这个 `Map` 将会提供给 `Report` 对象使用。

Listing 4.12: 生成报告的入口方法 `generateReport` 代码

```
public void generateReport(String reportId) {
    // 生成所有类别的子报告列表
    List<AbstractReport> reportGenerators =
        ReportGeneratorFactory.getReportGenerators();
    // 线程CountDown
    CountDownLatch countDownLatch = new
        CountDownLatch(reportGenerators.size());
    // 获取sampleContextMap
    Integer granularity = getGranularity(reportId);
    Map<String, SampleContext> sampleContextMap =
        DataParse.computeReport(reportId,
            DataParse.initConsumerList(granularity));
    // 遍历处理子报告
    reportGenerators.forEach(r -> reportThreadPool.execute(() -> {
        r.init(reportId, sampleContextMap);
        try {
            r.execute();
        } catch (Exception e) {
            LogUtil.error(e);
        } finally {
            countDownLatch.countDown();
        }
    }));
    try {
        countDownLatch.await();
    } catch (InterruptedException e) {
        LogUtil.error(e);
    } finally {
        saveResultService.saveReportTimeInfo(reportId);
        saveResultService.saveReportStatus(reportId);
    }
}
```

接着便是该代码段的核心逻辑，遍历 `reportGenerators` 中的每一个具体 `Report` 类，并且将其中的处理逻辑放入线程池中执行，以避免各个 `Report` 类的执行相互影响。在线程的 `execute` 方法中，首先调用 `report` 中的 `init` 方法初始化

私有变量，该方法在 `AbstractReport` 类中进行了实现，接着在 `try-catch` 代码块中执行 `report` 的 `execute` 方法，不同类别的 `Report` 对象将拥有不同的 `execute` 实现，最后在 `finally` 代码块中调用 `CountDownLatch` 的 `countDown()` 方法，减少计数 `count` 的值，当计数值递减为 0 后，则会释放所有等待的线程。

最后的 `try-catch` 代码块主要用于实现所有线程均结束执行后的报告信息存储，`count` 计数为 0 后，`await()` 方法后的其他内容将会被执行，因此将需要执行的代码写在 `finally` 代码块中，保证是否存在异常代码均能够被执行。代码中调用 `saveResultService` 的两个方法的作用如下，`saveReportTimeInfo` 用来存储报告的开始、结束、持续时间，`saveReportStatus` 用来更新报告的状态。

代码段 4.13 中展示了继承自 `AbstractReport` 抽象类的子报告类 `RTChartReport` 代码，此类是用于可视化展示响应时间的子报告，通过构造以测试时间为 x 轴、响应时间和用户并发数为两侧 y 轴的数据折线图。子类中利用 `@Override` 注解重写接口中定义的 `getReportKey` 和 `execute` 方法，`getReportKey` 方法仅是简单地返回该类别对应枚举值的名称，而 `execute` 方法负责获取、处理、保存图表数据。

Listing 4.13: 子报告——响应时间折线图代码

```
public class RTChartReport extends AbstractReport {
    @Override
    public String getReportKey() {
        return ReportKeys.RTChart.name();
    }
    @Override
    public void execute() {
        SampleContext responseTimeMap = sampleContextMap.get(
            ResponseTimeOverTimeGraphConsumer.class.getSimpleName());
        // 数据转换
        List<ChartData> resultList =
            DataParse.parseToChart(responseTimeMap.getData(), "",
                "yAxis");
        // 保存ReportResult
        saveResult(reportId, resultList);
    }
}
```

在 `execute` 方法中，首先从 `AbstractReport` 的私有变量 `sampleContextMap` 中获取响应时间 `Consumer` 所提供的 `SampleContext`，即 `JMeter` 数据样本。随后调用 `DataParse` 提供的 `parseToChart` 方法，将样本数据转化为前端展示所需的图表格式数据 `List<ChartData>`，此数据被命名为 `resultList`。最后将 `reportId` 和 `resultList` 作为参数，将此条子报告以 `ReportResult` 的形式进行保存。

用于将样本数据转换为图表数据的方法 `parseToChart` 如代码段 4.14 所示，此方法接收的 3 个参数分别为：`SampleContext` 提供的 `Map` 类型的数据、数据集名称、y 轴名称标识。`map` 对象中每个 `key` 对应了一次接口调用结果，因此我们对 `map.keySet()` 进行遍历即可对各个接口调用数据进行处理。在循环中，首先利用 `key` 获取 `JMeter` 提供的 `MapResultData`，进而利用 `getResult` 方法获取列表形式的图表 `series` 数据。由于可能存在 y 轴左右两侧分别展示两个不同数据集的情况，因此我们需要对 `series` 数据进行遍历。

Listing 4.14: `DataParse` 类中 `parseToChart` 方法代码

```
public static List<ChartData> parseToChart(Map<String, Object>
    map, String seriesName, String yAxis) {
    List<ChartData> list = new ArrayList<>();
    for (String key : map.keySet()) {
        MapResultData mapResultData = (MapResultData) map.get(key);
        ListResultData series = (ListResultData)
            mapResultData.getResult("series");
        if (series.getSize() > 0) {
            for (int j = 0; j < series.getSize(); j++) {
                MapResultData resultData = (MapResultData)
                    series.get(j);
                ListResultData data = (ListResultData)
                    resultData.getResult("data");
                ValueResultData label = (ValueResultData)
                    resultData.getResult("label");
                if (data.getSize() > 0) {
                    for (int i = 0; i < data.getSize(); i++) {
                        ListResultData listResultData =
                            (ListResultData) data.get(i);
                        String result = listResultData.accept(new
                            JsonizerVisitor());
                    }
                }
            }
        }
    }
    return list;
}
```

```
        result = result.substring(1, result.length() - 1);
        String[] split = result.split(",");
        ChartData chart = new ChartData();
        // 1. 设置chart的xAxis为timestamp (省略)
        // 2. 设置y轴数据
        if (StringUtils.equals("yAxis2", yAxis)) {
            chart.setyAxis2(new
                BigDecimal(split[1].trim()));
            chart.setyAxis(new BigDecimal(-1));
        } else {
            chart.setyAxis(new
                BigDecimal(split[1].trim()));
            chart.setyAxis2(new BigDecimal(-1));
        }
        // 3. 设置图表名称 (省略)
        list.add(chart);
    }
}
}
}
}
return list;
}
```

每一项 *serise* 数据均以 *MapResultData* 的数据结构存在，其中的“data”字段存储了图表中各个数据点的列表，由于“data”字段的类型 *ListResultData* 无法直接利用函数转换为 *List<String>* 格式，因此利用 *ListResultData* 提供的 *accept* 方法将列表转换为 JSON 格式的 *String*，将其删除前后“[]”字符后再使用 *split* 分隔，即可得到 *String* 类型的列表对象，记为 *result*。*result* 字符串列表的各项值分别为 [时间戳字符串, y 轴数据值]，随后利用 *result* 对 *ChartData* 各项进行赋值即可构造一个完整的图表数据对象，循环最后将其加入到 *List<ChartData>* 列表中即可。

4.6 系统实例展示

图4-21是测试接口定义的页面，测试人员在完成接口基本信息、请求头、请求参数等设置后，通过点击“测试”按钮调用接口以检查接口连通性，连通性测试页面如图4-22所示。

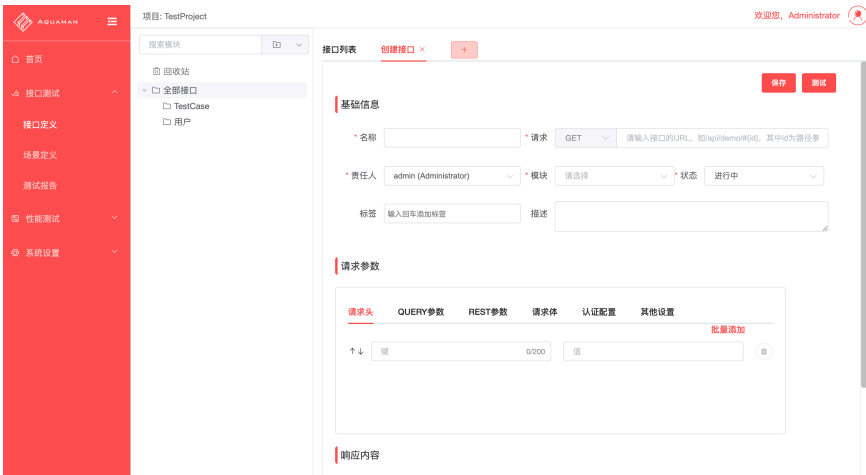


图 4-21: 测试接口定义页面截图

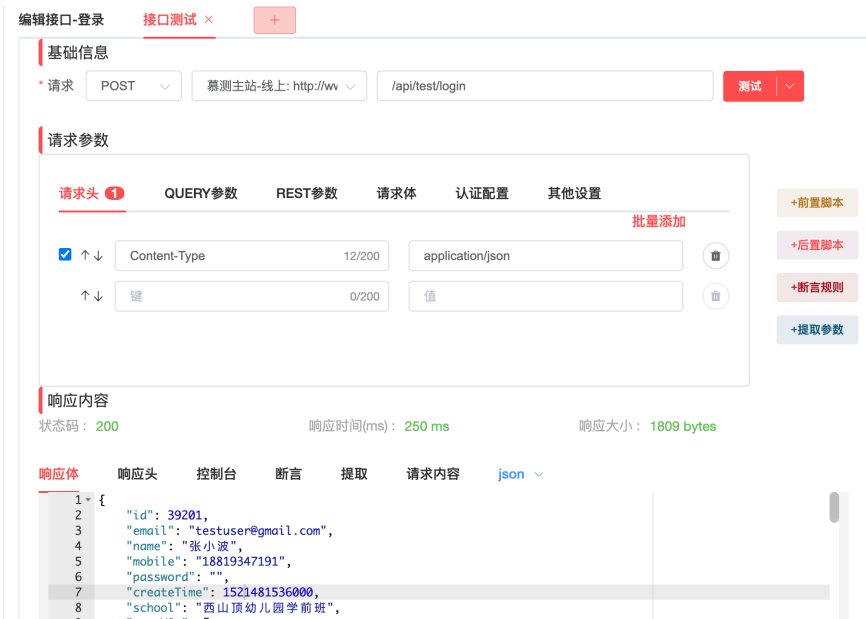


图 4-22: 连通性测试页面截图

图4-23是测试场景定义的页面，在此界面中，测试人员需要首先填写测试场景的基本信息，若是管理员用户还能够选择该场景的责任人。测试人员通过点击右下角的接口导入向场景中添加接口，并通过拖拽来修改接口调用顺序。

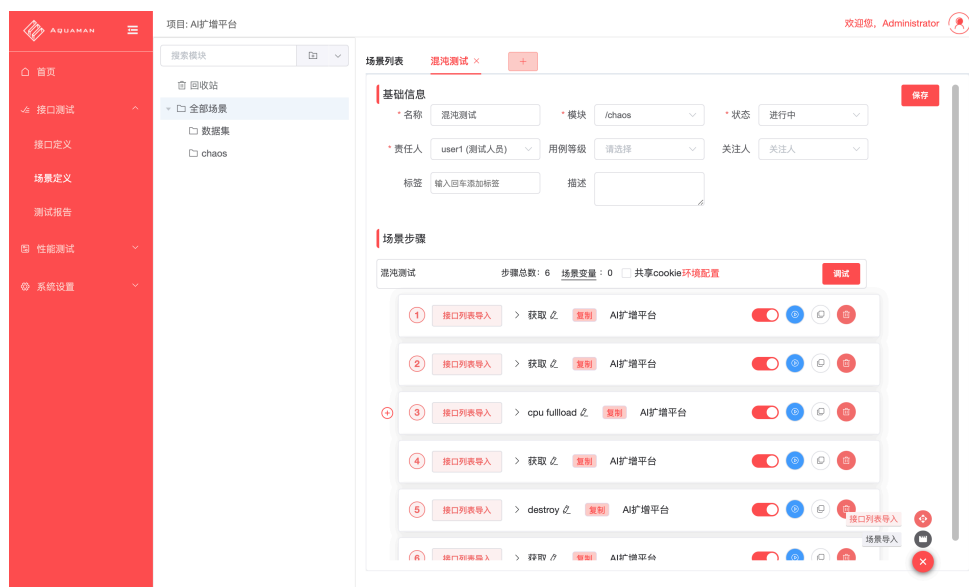


图 4-23: 测试场景定义页面截图

图 4-24是资源池管理菜单中创建测试资源池的页面，测试人员可在弹窗中设置资源池名称、测试节点等信息，在设置测试节点时可以点击”+” 按钮添加多个节点，并补充各个节点的 IP、端口号、最大并发量等信息。

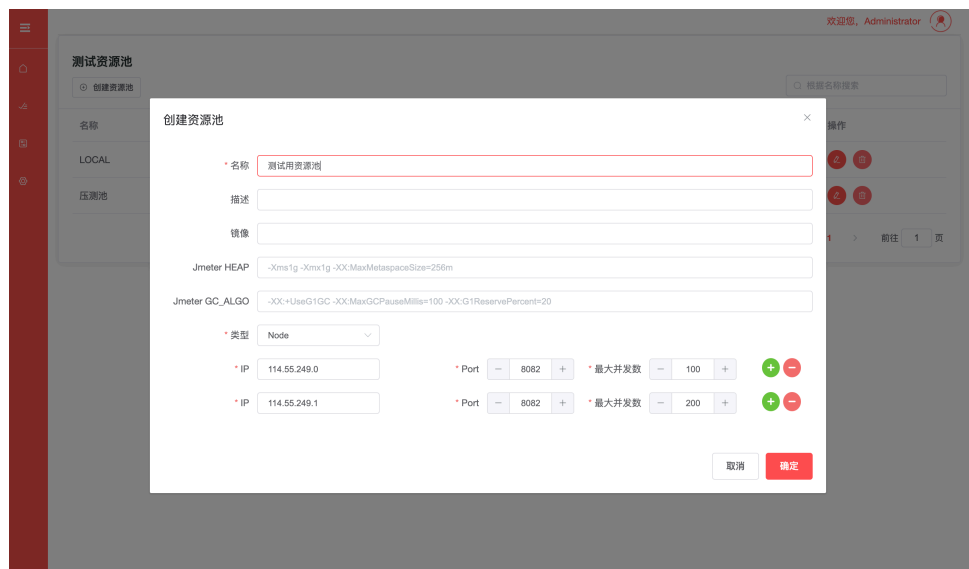


图 4-24: 测试资源池创建弹窗截图

图 4-25是创建性能测试时进行测试配置的页面，在选定测试场景或导入 JMX 文件后，进行测试加压曲线、最大并发数等压力配置。

图 4-26是性能测试报告列表，当测试开始执行时便会自动创建测试报告，

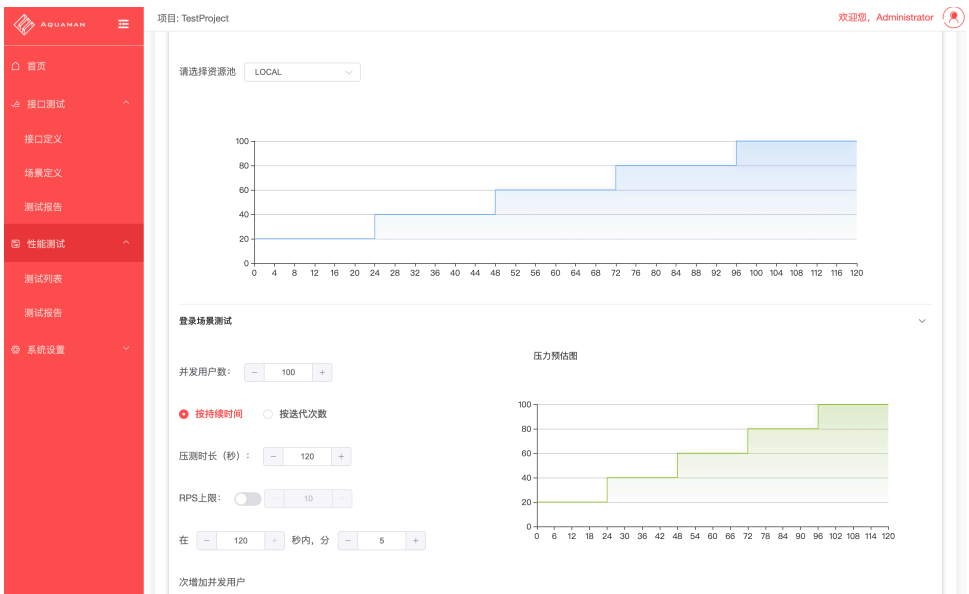


图 4-25: 测试资源池创建弹窗截图

在列表中展示了报告名称、所属测试名称、状态、创建人、创建时间等信息。

图 4-26: 测试报告数据概览页面截图

图 4-27~图 4-29是测试报告页面，上方的选择器用于修改测试报告维度，默认查看测试资源池的整体报告，用户也能够选择查看单个测试节点的测试报告。报告详情页面利用 Tab 标签分为了五部分，分别为测试配置、测试概览、请求统计、错误记录与日志详情。图 4-27的测试概览中展示了性能测试的核心数据指标，以及各个接口的并发执行情况、响应时间情况、错误情况和状态码统计随时间的变化图表。

图 4-28与 4-29中以表格的形式展示了请求统计数据与接口调用错误的统计数据，为用户展示了更为具体的测试执行情况。

图 4-30是测试报告中的监控详情界面，展示了测试执行过程中资源池节点的监控数据，测试人员能够从此子报告中获知服务器的资源占用情况。



图 4-27: 测试报告数据概览页面截图

测试概览 请求统计 错误记录 日志详情 监控详情 测试配置														
请求接口		执行情况			响应时间(ms)统计							吞吐量	网络(KB/s)	
Label	Q	Samples	失败数	失败率 (%)	Avg	Min	Max	Med	90%	95%	99%	Trans/s	Recd	Sent
默认1		964	40	4.149	1452.04	213	6196	1013.5	3203.5	4800.5	5725.8	8.019	4.489	5.104
默认2		952	28	2.941	1052.667	61	13141	652.5	2000.8	3784.35	5610.98	7.803	2.051	4.793
默认3		940	26	2.766	1032.347	67	6812	622	2212	3778.95	5589.34	7.968	2.057	4.903
默认4		933	71	7.61	1424.902	61	8367	880	3930.6	5289.7	6736.72	7.865	19.173	6.351
默认5		920	27	2.935	1025.489	52	8507	630	2200.7	3746.65	5510.9	7.9	15.674	4.852
默认6		916	94	10.262	1544.64	53	13270	941.5	5059.3	5316.75	5883.99	7.798	19.704	6.116

图 4-28: 测试报告请求统计页面截图

测试概览 请求统计 错误记录 日志详情 监控详情 测试配置			
错误统计			
错误类型	错误量	错误占比(%)	所有请求占比(%)
Non HTTP response code: java.net.SocketTimeoutException	513	100.000	6.190
错误详情			
接口名称	接口调用Samples	产生错误数	
默认1	964	40	
默认3	940	26	
默认2	952	28	
默认9	874	74	
默认8	884	32	
默认5	920	27	
默认4	933	71	
默认7	904	121	

图 4-29: 测试报告错误统计页面截图



图 4-30: 测试报告监控详情页面截图

4.7 本章小结

本章基于第三章分析的系统需求与持久化模型设计，对接口场景定义模块、测试执行模块、测试资源管理模块、测试数据处理模块、测试报告生成模块的详细设计和实现进行了介绍。通过架构图说明模块的整体结构与组件关系，通过 UML 类图说明模块中各个类的设计与调用关系，通过顺序图说明模块中各个服务和类之间的通信过程，并对模块中的核心代码实现进行了解释说明。最后在系统实例展示小节中提供了系统实际运行截图，以便于读者更直观地了解系统实现。

第五章 系统测试

5.1 测试目标

软件测试是软件研发流程中的重要部分，用于验证软件的正确性与需求实现情况 [39]，测试人员旨在软件产品正式上线前，发现尽可能多的软件缺陷并指导研发人员修复缺陷。为了对本文实现的性能测试系统进行验证，主要设计了四种类型的测试。四种类型的测试以渐进的方式开展，接口测试保证接口可用性，功能测试保证系统需求完整性，可移植性测试保证系统可移植性，负载测试保证系统高性能与可靠性。

(1) 接口测试：针对程序接口的功能完成度、可靠性、安全性进行的测试，接口测试并不在意接口内部的实现逻辑，仅关注传入特定参数时接口返回的响应状态码和响应数据。由于本系统是一个 Web 系统，对外提供的接口常常由多个方法组合实现，因此使用接口测试在测试过程中替代集成测试。SpringBoot 提供的接口能够直接使用 HTTP 调用，本文选用了简洁易用的自动化接口测试工具 Postman 来开展接口测试。

(2) 功能测试：功能测试的测试目标是软件以及其底层硬件构成的完整系统，其目的是验证最终系统是否满足了需求规格说明中定义的各项需求 [40]。与前接口测试类型不同，验收测试能够以人工测试的形式开展，通过思维导图的形式组织测试用例，分别验证需求规格中的功能需求。此外，为了使验收结果更具备真实性，可以邀请研发、测试团队之外的非开发人员参与其中，保证系统与用户实际需求的一致性。

(3) 可移植性测试：可移植性测试的目的是评估软件的移植边界，判断软件是否能够被成功地移植到不同的软硬件平台中并正确运行。由于《中国信创产业发展白皮书 (2021)》^①对国产软硬件、国产操作系统的发展规划，系统对国产软硬件的适配成为了软件开发过程中的关注点。为了对本系统进行可移植性测试，需要准备多个不同的软硬件环境，将系统部署至不同环境中并验证其功能一致性。

^①<http://www.cinic.org.cn/hy/zx/1040638.html>

(4) 负载测试：负责测试本系统的性能、可靠性两大非功能需求，通过对系统执行指定并发量的请求来测试系统在大规模调用时的性能表现。对本系统进行负载测试的主要目的是与其他主流性能测试工具在相同并发下的性能表现进行比较，以确保本系统的性能达到主流产品的同一水平。

5.2 测试环境

在对本系统进行测试前，首先对软件的部署环境进行说明，所有四类测试均在同样的硬件环境中进行。表中 5-1 展示了用于本系统各服务的云服务器环境配置，环境指标包括 CPU 数量、内存大小、带宽、实例规格。本系统的前后端均部署在主服务器中，数据处理服务也以 Docker 微服务的形式部署在主服务器中进行测试。此外，为了对测试资源池的可用性与可靠性进行测试，准备了两个测试节点服务器用于验证弹性资源池。

表 5-1: 测试服务器硬件环境

软件服务	云服务器规格	CPU 核数	内存	带宽
前后端主服务器	阿里云 ecs.g5.xlarge	4 核	16G	2M
测试节点服务 1	阿里云 ecs.c5.xlarge	4 核	8G	2M
测试节点服务 2	阿里云 ecs.c5.xlarge	4 核	8G	2M

测试服务器的软件环境如表 5-2 所示，表中展示了服务器的操作系统版本以及各个软件依赖项的版本号。所有服务器操作系统选用 64 位的 Ubuntu16.04，程序语言使用 Java8，数据库使用稳定的 MySQL5.7，用于部署各项服务的 Docker 版本为 19.03，负责测试执行的 JMeter 工具版本为 5.4.3，监控服务选用了 Prometheus 2.31.0 和 node_exporter1.3.1 来实现。

5.3 接口测试

本系统作为前后端分离的 Web 应用，HTTP 接口是连接前后端的渠道，接口也用于连通各个使用 Docker 部署在其他节点中的微服务，因此保证接口的正确性至关重要。接口测试是一种黑盒测试，测试时仅需要关注接口 URL 和请求参数，无需了解接口的实现逻辑。结合第三章的功能需求与第四章中各模

表 5-2: 测试服务器软件环境

软件依赖名称	软件版本
操作系统	Ubuntu16.04 64 位
Java-JDK	JAVA1.8.0_191
MySQL	MySQL5.7
Docker	Docker Engine 19.03
JMeter	Apache JMeter 5.4.3
Prometheus	prometheus-2.31.0
node_exporter	node_exporter-1.3.1

块 Controller 类的设计，挑选出复杂度较高的核心功能来设计接口测试用例，如表 5-3 所示。

表 5-3: 接口测试用例设计与测试结果

测试 ID	接口描述	输入	预期输出	测试结果
TEST-1	创建测试场景	正确的场景信息与接口列表	状态码 200; 返回新测试场景 ID	通过
TEST-2	创建性能测试	正确的测试任务信息	状态码 200; 返回新测试任务 ID	通过
TEST-3	创建性能测试	不完整的测试任务信息	状态码 400; 返回参数不完整提示	通过
TEST-4	执行性能测试	正确的测试任务 ID	状态码 200; 返回测试任务状态	通过
TEST-5	查看测试报告	正确的测试报告 ID	状态码 200; 返回报告详情数据	通过
TEST-6	查看测试报告	不存在的测试报告 ID	状态码 400; 返回报告不存在提示	通过
TEST-7	查看测试报告	失效的用户登录态	状态码 403; 返回需重新登录提示	通过
TEST-8	查看资源池报告	正确的资源池报告 ID	状态码 200; 返回资源池报告详情数据	通过
TEST-9	创建测试资源池	正确的资源池与节点信息	状态码 200; 返回新资源池 ID	通过
TEST-10	启动容器并执行测试	正确的 Docker 启动参数与测试任务 ID	状态码 200; 返回 Docker 状态	通过
TEST-11	启动容器并执行测试	未部署测试服务的节点 IP	状态码 404	通过

接口测试工具 Postman 模拟客户端向主服务器发送请求报文，最终通过验证服务器返回的响应来判断接口正确性，判断接口正确性的逻辑如下：

- (1) 当输入正确参数时，响应状态码为 200 且返回值正确。
- (2) 当输入错误参数时，响应状态码为 400 且返回错误提示信息。
- (3) 当用户未登录或登录态过期时，响应状态码为 403 且返回错误提示信息。

RUN SUMMARY		1	2	3	4	5	6	7	8	9	10
▶ POST	创建测试场景	10		0							
▶ POST	创建性能测试	10		0							
▶ POST	创建性能测试-缺失信息	10		0							
▶ POST	执行性能测试	10		0							
▶ GET	查看测试报告	10		0							
▶ GET	查看测试报告-不存在的报告ID	10		0							
▶ GET	查看测试报告-失效登录态	10		0							
▶ GET	查看资源池报告	10		0							
▶ POST	创建测试资源池	10		0							
▶ POST	启动测试节点容器	10		0							
▶ POST	启动测试节点容器-错误节点IP	10		0							

图 5-1: Postman 接口测试结果图

如图 5-1 所示，使用 Postman 对上述 11 个接口测试用例进行各 10 次的循环测试，其中包括 4 个 GET 方法与 7 个 POST 方法，最终所有测试输出均符合预期，且各接口 10 次循环测试输出均相同，接口测试全部通过。

5.4 功能测试

为了达到验证需求功能的目的，本节根据第三章的系统用例设计了与之相对应的 7 个功能测试用例，手动地执行系统用例中定义的各个流程，同步与预期结果进行对比，当在正确的操作流程下所有测试结果均符合预期，即可认定其功能是完整且正确的。

定义测试接口的功能测试用例如表 5-4 所示，测试人员能够对待测软件的测试接口进行定义，为了验证接口可用性，测试人员能够对单个接口进行模拟调用测试。测试中需要关注接口是否创建成功，接口测试是否能够正确执行，最后展示的响应结果是否正确。

定义测试场景的功能测试用例如表 5-5 所示，测试人员可以将多个测试接口进行组合与编排，为创建性能测试任务作准备。测试该功能需要关注是否能

表 5-4: 定义测试接口功能测试用例

测试 ID	TC1
功能名称	定义测试接口
功能描述	测试人员进入接口定义页面，填写测试接口的相关信息并保存，接口填写过程中可以进行接口连通性测试
测试步骤	1. 进入接口定义页面 2. 点击“+”按钮创建接口 3. 填写接口名称、接口方法、接口 URL、接口描述等基础信息，接着填写请求头信息，最后填写 Query 形式的请求参数 4. 点击“测试”按钮以测试接口连通性 5. 点击保存接口信息
预期结果	1. 系统前端展示接口定义页面 2. 系统展示创建接口 Tab 页 3. 系统展示信息填写结果 4. 系统执行接口连通性测试，并展示接口调用响应信息 5. 系统提示保存结果
测试结果	与预期相符

够正确批量导入多个接口，最终场景能否保存成功。

表 5-5: 定义测试场景功能测试用例

测试 ID	TC2
功能名称	定义测试场景
功能描述	测试人员进入场景定义页面，填写场景基本信息，并导入接口列表对接口进行组合编排。
测试步骤	1. 用户点击“创建场景”按钮 2. 用户填写场景名称、描述等基本信息 3. 用户点击添加接口按钮 4. 用户选中一个或多个接口，点击“复制”按钮将接口加入至场景中 5. 用户选中接口并长按进行拖拽，对接口顺序进行编排 6. 用户对接口中的信息进行补全和完善，确认后点击“保存”按钮
预期结果	1. 系统展示场景定义页面 2. 系统展示信息填写结果 3. 系统弹出窗口，展示所有已定义接口 4. 系统关闭弹窗，并在“场景步骤”中展示选中的接口 5. 系统展示编排后的顺序 6. 系统显示保存结果
测试结果	与预期相符

创建测试任务的功能测试用例如表 5-6 所示，测试人员可以根据已定义的场景创建性能测试任务，当完成并发量、加压曲线等压力配置与其他高级配置后，即可保存任务等待执行测试。在测试该功能时，需要重点关注导入的测试场景是否正确，压力配置是否能够正常配置，保存任务的结果是否正确。

表 5-6: 创建测试任务功能测试用例

测试 ID	TC3
功能名称	创建测试任务
功能描述	测试人员通过指定测试场景或导入场景来创建测试任务，并对性能测试任务的压力进行配置。
测试步骤	1. 测试人员选中指定场景，根据场景创建测试任务 2. 测试人员填写测试任务名称，任务名称默认为场景名称，并选择测试资源池 3. 测试人员设置并发用户数、加压方式（按持续时间或迭代次数）、压测时长等压力配置参数 4. 测试人员点击“保存”按钮
预期结果	1. 系统展示任务创建页面 2-3. 系统展示信息填写结果 4. 系统显示保存结果
测试结果	与预期相符

执行性能测试功能测试用例如表 5-7 所示，测试人员在完成测试任务创建后即可启动测试任务的执行，并开始生成测试报告。在测试该功能时需要关注点击执行后任务状态是否改变，以及是否有新的测试报告被生成。

表 5-7: 执行性能测试功能测试用例

测试 ID	TC4
功能名称	执行性能测试
功能描述	测试人员在创建性能测试任务后，能够开始执行任务
测试步骤	1. 用户进入测试任务详情页面 2. 用户点击“保存并执行”执行当前任务
预期结果	1. 系统展示测试任务详情页面 2. 系统跳转至测试报告页面，展示当前实时的测试报告
测试结果	与预期相符

管理测试资源池功能测试用例如表 5-8 所示，测试人员能够创建测试资源池，并在资源池中添加一至多个测试节点机器，此外，能够更改资源池的可用状态以及进行编辑、删除操作。在该测试中需要关注资源池是否正确地被创建、更新状态后资源池是否可用、删除资源池后资源池是否无法对用户可见。

查看测试报告的功能测试用例如表 5-9 所示，测试人员能够实时查看已开始执行或执行完成的测试任务结果，最终结果以可视化报告的形式呈现。执行该功能测试需要关注测试报告是否与测试任务正确对应，测试报告前端页面展示是否正常，测试结果是否符合预期。

表 5-8: 管理测试资源池功能测试用例

测试 ID	TC5
功能名称	管理测试资源池
功能描述	测试人员能够对测试资源池进行查看、创建、编辑、删除操作，以实现弹性资源池的效果
测试步骤	1. 用户进入资源池管理页面 2. 用户点击创建资源池 3. 用户填写资源池名称、描述等基础信息，并添加两个 Node 节点测试机器，填写机器的 IP、端口号、最大并发数等信息 4. 用户点击“确定”保存资源池 5. 用户点击启用开关将资源池状态变为可用状态 6. 用户点击删除按钮删除某个资源池
预期结果	1. 系统展示资源池列表页面 2. 系统展示资源池创建弹窗 3. 系统展示填写结果并新增 2 条新的 Node 节点数据 4. 系统关闭弹窗并在列表中新增一个资源池 5. 系统提示“状态修改成功” 6. 系统进行二次确认并提示删除结果
测试结果	与预期相符

表 5-9: 查看测试报告功能测试用例

测试 ID	TC6
功能名称	查看测试报告
功能描述	测试人员能够查看测试执行中或已结束任务的测试报告，报告中以数据概览、图表、表格等形式展现待测软件的性能表现
测试步骤	1. 用户进入测试报告列表页面 2. 用户点击进入测试报告详情 3. 用户点击导出报告
预期结果	1. 系统分列展示测试报告列表页面 2. 系统展示测试报告详情页面，页面中将展示压力配置、测试概览、请求统计、错误记录、日志详情信息 3. 系统将报告生成为 PDF 格式文件并自动下载
测试结果	与预期相符

查看资源池报告的功能测试用例如表 5-10 所示，测试人员能够查看测试资源池在测试过程中各个测试机的硬件指标变化情况。测试该功能需要关注资源池能否正常展示，资源池报告结果是否与服务器真实状况相同。

表 5-10: 查看资源池报告功能测试用例

测试 ID	TC7
功能名称	查看资源池报告
功能描述	测试人员能够查看执行性能测试的资源池中各个测试节点的软硬件监控报告
测试步骤	1. 用户进入测试报告详情页面 2. 用户点击“查看资源池报告” 3. 用户点击导出报告
预期结果	1. 系统分列展示测试报告列表页面 2. 系统展示资源池报告界面，界面中将展示测试过程中各个测试机器的 CPU 开销情况、内存占用情况等数据 3. 系统将报告生成为 PDF 文件并自动下载
测试结果	与预期相符

5.5 可移植性测试

党的十九届中全会强调了科技创新对于当今中国的重要性，将打破桌面计算机、高端交易处理等科技领域的垄断作为科技发展的首要目标 [41]，因此国产操作系统、国产 CPU 技术正在飞速进步。近两年，麒麟、统信 UOS 作为国产操作系统的代表，已经正式发布使用，未来国产操作系统可能会迎来大规模使用。

为了对本系统的可移植性进行测试，将本系统在不同的操作系统上进行部署并验证其正确性，用于可移植性测试的环境如表 5-11所示。

表 5-11: 可移植性测试环境准备

操作系统名称	操作系统指令集
银河麒麟高级服务器操作系统（鲲鹏版）V10	arm
银河麒麟高级服务器操作系统（飞腾版）V10	arm
Centos7	x86_64

对本系统进行可移植性测试的流程如下：首先与服务器建立连接，随后在服务器中安装 Docker，下载本系统主服务器与其余服务的 Docker 镜像，随后 docker-compose 将启动系统所需的主服务、数据处理服务、Kafka、MySQL 等服务的 Docker 容器，并将端口号暴露给外部用户访问。最终用户可以直接通过由 IP+ 端口号 + 路由组成的链接访问该系统。依照上述测试流程执行的截图如 5-2所示，测试结果如表 5-12所示，测试表明本系统拥有良好的可移植性。

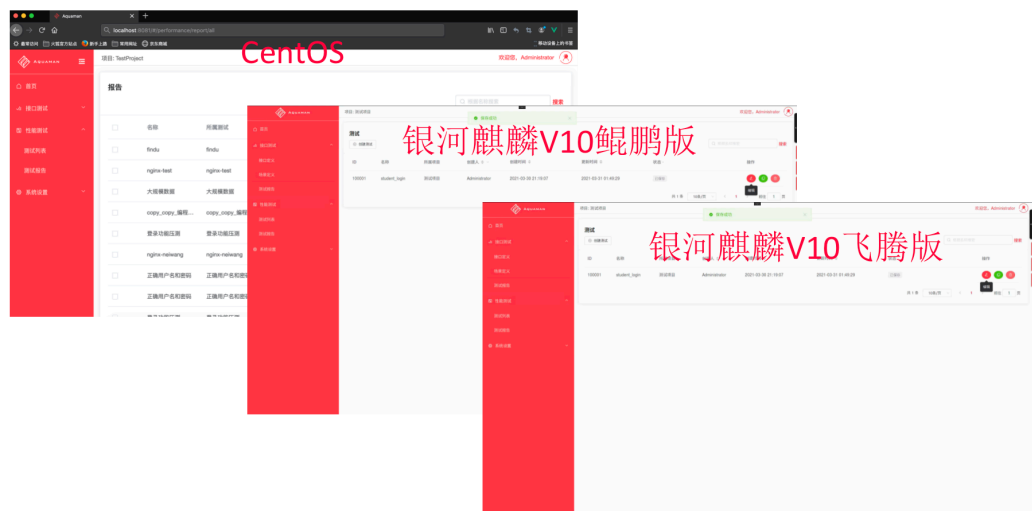


图 5-2: 可移植性测试执行截图

表 5-12: 可移植性测试结果

操作系统名称	是否成功部署	功能测试通过率
银河麒麟高级服务器操作系统（鲲鹏版）V10	是	100%
银河麒麟高级服务器操作系统（飞腾版）V10	是	100%
Centos7	是	100%

5.6 负载测试

负载测试是一种计划性的测试，测试人员将设置指定的并发数量，观察系统在执行并发请求时的功能表现与性能表现。对本系统的负载测试将分为两部分，首先使用性能测试工具阿里云 PTS 对特定场景加压以查看接口调用的负载情况，接着通过比较本系统与原生 JMeter 在多个测试执行时的硬件资源占用情况以分析其性能表现。

为了对本系统的接口负载进行测试，选用查看测试报告功能作为测试场景，设置每秒请求数量 (RPS) 为 400，测试执行 2 分钟。该场景的测试结果如图 5-3 所示，接口调用成功率为 100%，接口响应时间平均为 40ms，最高响应时间不超过 70ms，说明此查看测试报告相关接口的性能良好。

随后，为了对本系统在执行测试时的性能表现进行测试，选用性能测试工具 JMeter 与本系统使用相同测试计划、相同的压力配置分别执行性能测试，并对比执行过程中测试服务器的 CPU、内存和负载情况。在进行对比测试前，准



图 5-3: 查看测试报告场景负载测试监控数据图

备了包含 10 个步骤的测试计划，设置性能测试中并发用户数为 200、加压时间为 20 秒，以迭代地形式循环执行 5 次。

使用本系统执行测试时，服务器监控数据如图 5-4 所示，分析第一张图表可知，当测试并发量升至峰值时，CPU 的使用率也出现了峰值，为 60.27%，随着测试继续进行 CPU 使用率缓慢下降直至恢复平缓；而第二、三张图表中表明，测试过程中内存产生了小幅上升，数据与正常运行时差异不大，而系统负载有一个较为明显的峰值。

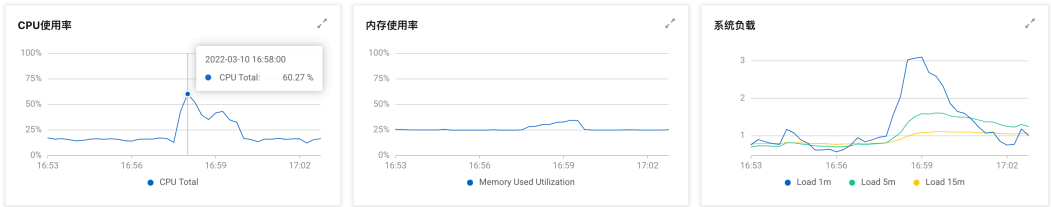


图 5-4: 本系统测试执行过程服务器监控数据图

使用 JMeter 执行测试时，监控数据如图 5-5 所示，性能测试过程中 CPU 的最高使用率为 55.53%；内存使用率仅上升 5%，波动较小；系统负载图表中显示测试过程中服务器较测试开始前更为繁忙。

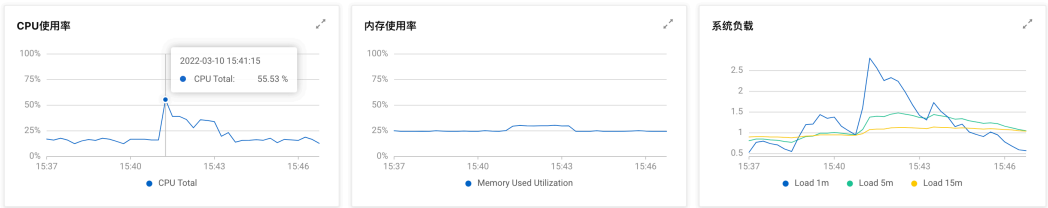


图 5-5: JMeter 测试执行过程服务器监控数据图

将使用两种工具进行性能测试的监控数据进行比较可知，内存使用率和系统负载两个指标的差异并不大，而 CPU 使用率方面，使用本系统进行测试时 CPU 使用率峰值比使用 JMeter 执行测试高近 5%。本系统 CPU 使用率相对较高的原因是除 JMeter 服务外，SpringBoot 服务、数据处理服务与报告生成逻辑均需要占用一定的 CPU，因此，比直接使用 JMeter 的开销有小幅升高在可接受范围内。

为了降低本系统测试执行产生的服务器开销，可以在弹性资源池中添加两个测试节点服务器以分摊压力。为验证此假设，启用两个 4C8G 的服务器作为测试节点，配置测试资源池中各个节点的最大并发为 100，执行测试时请求将平均分发至三个测试节点，主节点服务器的监控情况如图 5-6 所示。从第一张图表中能够看到 CPU 使用率有明显下降，因此利用资源池的弹性能力配置多个测试节点，能够对服务器的压力进行有效控制。此外，提升服务器的 CPU、内存配置也是提升测试执行性能的方法之一。

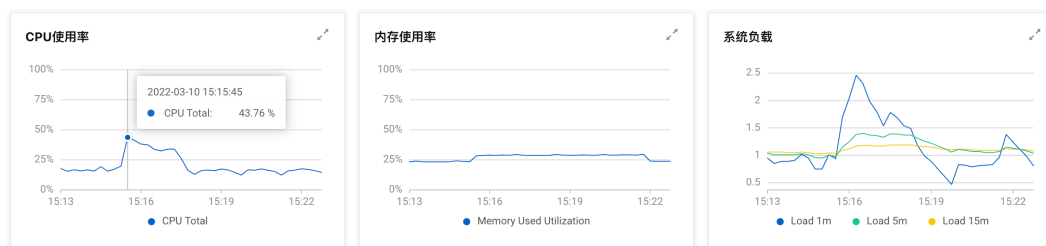


图 5-6: 本系统多节点测试执行过程主服务器监控数据图

5.7 本章小结

本章对基于第三、四章实现的基于 Docker 的弹性资源池性能测试系统进行测试，为了对系统全面的验证，设计并展开了四种类型的测试。接口测试用于验证接口可用性与正确性，功能测试用于验证系统是否满足用户需求，可移植性测试用于验证系统是否能够被移植到不同的软硬件环境，负载测试用于验证系统在一定压力下的正常运行。测试结果表明：本系统是一个高可用、功能完整、可移植、性能良好的系统。

第六章 总结与展望

6.1 总结

随着互联网用户规模的日渐扩增，软件将面临更巨大的流量冲击，因此，在软件上线前测试人员便需要对软件性能进行测试，以评估能够应对的并发用户数量。而在执行性能测试时，由于 CPU、内存等硬件配置的影响，单个机器的并发量受限，上万级并发的性能测试是很难使用单机实现的。为了解决单机性能测试受限的问题，让测试人员能够简单易用地使用多台测试机器进行性能测试，本文提出了一个基于 Docker 的弹性资源池性能测试系统。

本系统利用 Docker 高效、灵活、一致的特点，将测试节点使用的测试执行服务打包成 Docker 镜像，从而实现在测试机器上的快速下载、安装、部署。为了向测试人员提供简洁易用的测试平台，本系统以 Web 应用的形式部署，并将主服务构建为 Docker 镜像，以便未来进行私有化扩展。此外，为了打通性能测试全流程，本系统提供了测试接口定义、测试场景编排以及 JMX 文件的导入导出功能，测试人员能够在此平台中进行全过程的性能测试。最后，本系统为测试人员提供了详尽的性能测试报告，测试人员可以选择查看资源池的整体测试报告，也可以查看资源池中单个测试机的测试报告。在测试报告中展示了响应时间、吞吐量、错误率以及各个请求接口的统计信息等，并以图表的形式展示了测试执行过程中测试机器的 CPU 使用率、内存占用率等数据的变化情况。

本文的主要工作与组织结构如下：首先在第一章中对此系统的背景进行介绍，分析了国内外性能测试相关的研究进展，对性能测试工具的现状进行分析。第二章中对本系统开发所涉及的相关技术与工具进行简要介绍，阐述各个工具的选型原因。第三章整合了系统背景与研究现状中存在的问题，将解决方法整理为软件需求，确定目标用户后列举出系统需要实现的功能需求与非功能需求。需求确定后便对系统整体架构进行设计，以 4+1 视图的形式描述系统架构，并将系统划分为五个模块：接口场景定义、测试执行、节点管理、数据处理与报告生成。第四章对上述五个模块进行了详细设计，通过 UML 图和核心代码段展示了模块的实现逻辑。为了对系统进行验证，第五章设计了四种不同

类型的测试，分别对本系统进行了接口测试、功能测试、可移植性测试和负载测试，测试表明系统的功能需求和非功能需求均得以正确实现。

6.2 展望

本系统完成开发测试后，已经部署上线并成功投入使用。在使用过程中发现该系统存在着一些不足之处，并且实现过程仍有优化空间，主要包括以下几点：

(1) 测试接口扩展：本系统目前仅支持 HTTP 协议的接口，这是互联网中最主流的请求协议，但仍有其他数据传输协议被广泛用于不同类型的软件中，如 WebSocket、Dubbo、Thrift、JDBC 等。WebSocket、Dubbo、JDBC 协议的接口测试能够直接基于 JMeter 实现，而 Thrift 并不是 JMeter 支持的协议，因此需要利用 Java 对 JMeter 源码进行二次开发。通过支持更多的测试接口协议，本系统的待测软件类型能够从 Web 应用扩展到大部分类型的软件应用。

(2) 测试报告对比：当前系统提供的测试报告只存在整体资源池与单个测试节点两种维度，测试人员只能大致了解不同测试节点在执行测试时的差异。未来计划实现测试报告对比功能，可以任意选择两个及以上的测试报告进行比较，对比报告中将以表格排序形式展示不同报告概览数据的大小关系，并将各报告中的图表进行合并，使得用户能够直观量化地看到测试结果差异。

(3) 容器技术扩展：为了对执行测试的服务进行快速自动化部署，本系统选用了 Docker 容器在测试节点中执行测试，当资源池中存在过多测试节点时，对于容器状态的查询与管理将会成为测试执行中的高成本开销。未来能够利用 Kubernetes、Apache Mesos 等容器编排工具取代单一的 Docker 容器，以支持更大规模的测试场景。

致 谢

六年前，我第一次来到南京，第一次踏入南京大学的校门，写出第一行 C 代码，与小组成员合作产出第一个软件项目，在南京大学学习的六年时光如流星般转瞬即逝。非常幸运能够在保研结果确认后加入 iSE 实验室，在 iSE 实验室中得到了老师们的悉心指导，结交了志趣相投的朋友，也在毕业之际顺利完成了毕业论文，为自己的硕士生涯画上了完美的句点。

首先，我想感谢我的导师陈振宇教授，感谢他尊重我的意愿接受我进入实验室研发组，在实验室众多项目的研发过程中给予我指导，从而使我的技术能力和项目管理能力都有了质的飞跃。陈老师在毕业论文开题阶段便为我指引了正确的研究方向，编写论文过程中也及时对我进行督促与建议，他的幽默风趣时常感染着我，以乐观积极的态度学习与生活。此外，我想感谢实验室的房春荣老师、冯洋老师和赵源博士，与他们合作的过程中我也受益颇多。

其次，在 iSE 研发组学习的两年里，我想感谢一起完成项目的“战友”们，与他们共同开展研发工作的过程中，我的前端技术日渐精进，从小白逐渐成为了入门者，我的技术栈在慢慢扩展，很多课堂上学到的知识在实践中得以理解和应用。我需要特别感谢薛晓波学长，在项目中对我进行指导的同时不遗余力地给我的职业道路规划提出建议，是我研究生阶段的良师益友。我还要感谢郭超学长和王瑾同学，与他们合作完成需求的过程是非常愉悦的。最后我想感谢实验室的段梦洋学姐、李文龙学长、何云等其他同学，感谢他们陪伴我度过的快乐时光。

最后，我想感谢我的父母在读研期间始终支持关心着我，感谢我的男朋友这两年来的爱和陪伴。感谢我的舍友曹畅、袁玉婷和张贝贝，她们的存在使我每天都感到快乐和温暖，毕业后终将成为聚少离多的老友，但希望我们的情谊永存于心。

2020 到 2022，从新冠疫情、俄乌战争到东航客机失事，外公外婆也在近一年相继去世，我常常感慨着人类的渺小与无力。庆幸拥有家人、老师与挚友的陪伴与关爱，我顺利完成了人生最后的学业阶段。衷心祝愿所有人健康平安、诸事顺遂，希望逝去的人在远方给予我奔赴未来的希望。

参考文献

- [1] 王博然, 苏钢. 软件工程的历史与发展趋势 [J]. 北京工业职业技术学院学报, 2008(03): 40–42.
- [2] WEIK M H. software engineering[M/OL] // Computer Science and Communications Dictionary. Boston, MA: Springer US, 2001: 1610–1610.
https://doi.org/10.1007/1-4020-0613-6_17655.
- [3] 杨芙清. 软件工程技术发展思索 [J]. 软件学报, 2005(01): 1–7.
- [4] PAULK M, CURTIS B, CHRISSIS M B, et al. Capability maturity model for software, encyclopedia of software engineering[J]. Handbook of Systemic Autoimmune Diseases, 1993: 3–10.
- [5] ANON. GB/T16260.1, 2006: Information Technology-Software Quality Characteristics and Metrics - 1: Quality Model[S]. .
- [6] CORTELLESA V, DI MARCO A, INVERARDI P. What Is Software Performance?[M/OL] // Model-Based Software Performance Analysis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 1–7.
https://doi.org/10.1007/978-3-642-13621-4_1.
- [7] 岳昆, 王晓玲, 周傲英. Web 服务核心支撑技术: 研究综述 [J]. 软件学报, 2004(03): 428–442.
- [8] WANG J, WU J. Research on Performance Automation Testing Technology Based on JMeter[C/OL] // 2019 International Conference on Robots Intelligent System (ICRIS). 2019: 55–58.
<http://dx.doi.org/10.1109/ICRIS.2019.00023>.
- [9] 李念强, 孙传伟, 王玉泰. 分布式测试平台的应用研究 [J]. 计算机测量与控制, 2003(12): 931–933+937.

- [10] WOODSIDE M, FRANKS G, PETRIU D C. The Future of Software Performance Engineering[C/OL] // Future of Software Engineering (FOSE '07). 2007 : 171 – 187.
<http://dx.doi.org/10.1109/FOSE.2007.32>.
- [11] MENASCE D. Load testing of Web sites[J/OL]. IEEE Internet Computing, 2002, 6(4) : 70 – 74.
<http://dx.doi.org/10.1109/MIC.2002.1020328>.
- [12] YORKSTON K. The Basic Concepts of Performance Testing[M/OL] // Performance Testing: An ISTQB Certified Tester Foundation Level Specialist Certification Review. Berkeley, CA : Apress, 2021 : 1 – 67.
https://doi.org/10.1007/978-1-4842-7255-8_1.
- [13] 韩明军. 软件性能测试过程 [J]. 软件学报, 2007(11) : 41 – 43.
- [14] MENG X. Designing approach analysis on small-scale software performance testing tools[C/OL] // Proceedings of 2011 International Conference on Electronic Mechanical Engineering and Information Technology : Vol 8. 2011 : 4254 – 4257.
<http://dx.doi.org/10.1109/EMEIT.2011.6023983>.
- [15] YAN X, WEN F, FAN C, et al. Performance Testing of Open Laboratory Management System Based on LoadRunner[C/OL] // 2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control. 2011 : 164 – 167.
<http://dx.doi.org/10.1109/IMCCC.2011.50>.
- [16] MAR W. [EB]. 2015.
- [17] PRADEEP S, SHARMA Y K. A Pragmatic Evaluation of Stress and Performance Testing Technologies for Web Based Applications[C/OL] // 2019 Amity International Conference on Artificial Intelligence (AICAI). 2019 : 399 – 403.
<http://dx.doi.org/10.1109/AICAI.2019.8701327>.
- [18] VCAA. Stress, Load, Volume, Performance, Benchmark, Baseline Testing Tool Evaluation and Comparison[EB/OL]. 2005.
<http://www.vcaa.com/tools/loadtesttoolevaluationchart-023.pdf>.

- [19] JACOB A, KARTHIKEVAN A. Scrutiny on Various Approaches of Software Performance Testing Tools[C/OL] // 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA). 2018 : 509–515.
<http://dx.doi.org/10.1109/ICECA.2018.8474876>.
- [20] SUFFIAN M D M, FAHRURAZI F R. Performance testing: Analyzing differences of response time between performance testing tools[C/OL] // 2012 International Conference on Computer Information Science (ICCIS) : Vol 2. 2012 : 919–923.
<http://dx.doi.org/10.1109/ICCISci.2012.6297157>.
- [21] 赵冲冲, 白晓颖, 王钊. 性能测试工具能力评估框架研究, 2006(03):244-248+254.[J]. 计算机科学, 2006(03): 244–248.
- [22] BERNSTEIN D. Containers and Cloud: From LXC to Docker to Kubernetes[J/OL]. IEEE Cloud Computing, 2014, 1(3): 81–84.
<http://dx.doi.org/10.1109/MCC.2014.51>.
- [23] DEINUM M. Packaging[M/OL] // Spring Boot 2 Recipes: A Problem-Solution Approach. Berkeley, CA : Apress, 2018 : 307–320.
https://doi.org/10.1007/978-1-4842-3963-6_11.
- [24] PRASAD REDDY K S. Getting Started with Spring Boot[M/OL] // Beginning Spring Boot 2: Applications and Microservices with the Spring Framework. Berkeley, CA : Apress, 2017 : 21–33.
http://dx.doi.org/10.1007/978-1-4842-2931-6_2.
- [25] BOOT S. Natural population growth rate[EB/OL]. 2021.
<https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle>.
- [26] 陈涛. MVVM 设计模式及其应用研究 [J]. 计算机与数字工程, 2014(42(10)): 1982–1985.
- [27] LI D, MEI H, SHEN Y, et al. ECharts: A declarative framework for rapid construction of web-based visualization[J/OL]. Visual Informatics, 2018, 2.
<http://dx.doi.org/10.1016/j.visinf.2018.04.011>.

- [28] JMeter. The Apache Jakarta Project[EB]. .
- [29] WANG J, WU J. Research on Performance Automation Testing Technology Based on JMeter[C/OL] // 2019 International Conference on Robots Intelligent System (ICRIS). 2019 : 55 – 58.
<http://dx.doi.org/10.1109/ICRIS.2019.00023>.
- [30] CHEN L, XIAN M, LIU J. Monitoring System of OpenStack Cloud Platform Based on Prometheus[C/OL] // 2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL). 2020 : 206 – 209.
<http://dx.doi.org/10.1109/CVIDL51233.2020.0-100>.
- [31] PRASAD REDDY K S. Working with MyBatis[M/OL] // Beginning Spring Boot 2: Applications and Microservices with the Spring Framework. Berkeley, CA : Apress, 2017 : 65 – 69.
https://doi.org/10.1007/978-1-4842-2931-6_6.
- [32] 孙大为, 张广艳, 郑纬民. 大数据流式计算: 关键技术及系统实例 [J]. 软件学报, 2014(4): 839 – 862.
- [33] SHREE R, CHOUDHURY T, GUPTA S C, et al. KAFKA: The modern platform for data management and analysis in big data domain[C/OL] // 2017 2nd International Conference on Telecommunication and Networks (TEL-NET). 2017 : 1 – 5.
<http://dx.doi.org/10.1109/TEL-NET.2017.8343593>.
- [34] WU H. Research Proposal: Reliability Evaluation of the Apache Kafka Streaming System[C/OL] // 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2019 : 112 – 113.
<http://dx.doi.org/10.1109/ISSREW.2019.00055>.
- [35] GB/T 9385-2008 计算机软件需求规格说明规范 [EB/OL]. 2008.
<http://c.gb688.cn/bzgk/gb/showGb?type=online&hcno=2790825C43AD0B69E3C38C140BFFCFE6>.
- [36] KRUCHTEN P B. The 4+ 1 view model of architecture[J]. IEEE software, 1995, 12(6): 42 – 50.

-
- [37] docker-java: getting_started[EB/OL]. 2021.
https://github.com/docker-java/docker-java/blob/master/docs/getting_started.md.
- [38] docs.oracle.com javase: CopyOnWriteArrayList[EB/OL]. .
<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html>.
- [39] O'REGAN G. Software Testing Capability Maturity Model Integration[M/OL]
// Introduction to Software Quality. Cham: Springer International Publishing,
2014: 119–134.
https://doi.org/10.1007/978-3-319-06106-1_7.
- [40] 马瑞芳, 王会燃. 计算机软件测试方法的研究 [J]. 小型微型计算机系统,
2003, 12(4): 2210–2213.
- [41] 倪光南. 坚持信创科技自立自强建设网络强国和数字中国 [J]. 信息安全研究,
2021, 01(7): 2–3.

简历与科研成果

基本信息

杨郁芩，女，仫佬族，1999 年 2 月出生，广西壮族自治区南宁市人。

教育背景

2020 年 9 月 — 2022 年 6 月 南京大学软件学院

硕士

2016 年 9 月 — 2020 年 6 月 南京大学软件学院

本科