****

****

**本 科 毕 业 设 计**

**院 系** 软件学院

**专 业** 软件工程

**题 目** HappyMock集成测试框架的设计与实现

**年 级** 2011级 **学 号** 111250190

**学生姓名** 谢福恒

**指导教师** 李翔 **职 称** 讲师

**论文提交日期**

**南京大学本科生毕业论文（设计）中文摘要**

毕业论文题目： HappyMock集成测试框架的设计与实现

软件学院 院系 软件工程 专业 2011 级本科生姓名：谢福恒

指导教师（姓名、职称）： 李翔 讲师

摘要：

近几年随着SOA和REST的发展，基于HTTP协议的接口越来越流行，随之而来对基于HTTP协议的接口的集成测试和使用都不易进行或者较为繁琐，这无疑降低了公司整体的工作效率。本项目就是致力于解决基于HTTP协议接口的集成测试问题。用户通过编辑简单的用例来模拟接口的实现，以期减轻测试开发人员负担，提高工作效率。

整个项目可以划分为两个模块：用例编辑模块和DSL解析引擎。用例编辑模块前端采用Jade＋Less实现，后端使用Express框架给前端提供REST接口；DSL解析引擎采用Java开发，借助Netty开发服务端程序，大大提高了程序的响应速度；两个模块均是采用MongoDB作为数据库，最后采用Docker进行部署发布，进一步提高了工作效率。

该项目的第一版本中，我们较好的完成了用例编辑模块，提供了较为人性化的交互界面；同时DSL解析引擎模块能够支持较为常见的Request和Response关键字，并且在架构设计时，为项目的可扩展性打下坚实的基础。

本人全程参与了整个项目，从需求获取到最终部署发布，尽管遇到了各种难题，但最终依然完成了整个项目。

关键词：集成测试，HTTP，mock，DSL，框架、REST。

南京大学本科生毕业论文（设计）英文摘要

THESIS： Design and implementation of HappyMock integration testing framework

DEPARTMENT： Software Institute

SPECIALIZATION: Software Engineering

UNDERGRADUATE: 2011

MENTOR: Xiang Li

ABSTRACT：

In recent years, with the development of SOA and REST, interfaces based on HTTP protocol becomes more and more popular, followed by integration testing and use of interface based on HTTP protocol are not easy or more complicated, which will undoubtedly reduce the working efficiency of the whole company. This project is committed to addressing the problem. Using this project, Users can edit simple cases to simulate the interface, which will help tester and developers to mitigate the burden and improve efficiency.

The entire project can be divided into two modules: use cases editing module and DSL parsing engine. Use cases editing module adopted Jade && Less as the front framework, with Express as the back-end framework to provide REST interface; DSL parsing engine developed in Java and used Netty to develop server-side program, which greatly improved the response speed of the procedure; At the same time, the two modules all used MongoDB as a database, Finally, we used Docker to deploy and release the project, which further improved the work efficiency.

In the first version of the project, we completed a good case edit module, provides a more user-friendly interface; while DSL parsing engine module can support the common Request and Response keywords, What's more, In the architectural design, we laid a a solid foundation for the project scalability.

I have full participated in the entire project, from requirements capture to final deployment, despite experiencing a variety of problems, but in the end still completed the entire project.

KEY WORDS: integration testing，HTTP，mock，DSL， framework，REST。

**目 录 TOC \t "标题 1, 1,标题 2, 2,标题 3, 3"**

**图目录 PAGEREF \_Toc \h III**

**表目录 PAGEREF \_Toc1 \h IV**

**第一章 引言 PAGEREF \_Toc2 \h I**

1.1 项目背景 PAGEREF \_Toc3 \h I

1.2 国内(外)同类软件研究现状 PAGEREF \_Toc4 \h II

1.3 论文的主要工作和组织结构 PAGEREF \_Toc5 \h III

**第二章 HappyMock技术概述 PAGEREF \_Toc6 \h IV**

2.1 Spring技术 PAGEREF \_Toc7 \h IV

2.1.1 Spring简介 PAGEREF \_Toc8 \h IV

2.1.2 Spring模块 PAGEREF \_Toc9 \h V

2.2 Netty技术 PAGEREF \_Toc10 \h VI

2.3 Nodejs/Express技术 PAGEREF \_Toc11 \h VII

2.3.1 Nodejs简介 PAGEREF \_Toc12 \h VII

2.3.2 Nodejs特性 PAGEREF \_Toc13 \h VII

2.3.3 Express简介 PAGEREF \_Toc14 \h VII

2.4 Docker技术 PAGEREF \_Toc15 \h VII

2.5 Bootstrap技术 PAGEREF \_Toc16 \h VII

2.6 Less＋Jade技术 PAGEREF \_Toc17 \h VIII

2.6.1 Less简介 PAGEREF \_Toc18 \h VIII

2.6.2 Jade简介 PAGEREF \_Toc19 \h VIII

2.7 JQuery技术 PAGEREF \_Toc20 \h VIII

2.7.1 JQuery简介 PAGEREF \_Toc21 \h VIII

2.7.2 JQuery基本功能 PAGEREF \_Toc22 \h VIII

2.8 Git技术 PAGEREF \_Toc23 \h IX

2.9 本章小结 PAGEREF \_Toc24 \h IX

**第三章 HappyMock系统需求分析与概要设计 PAGEREF \_Toc25 \h X**

3.1 HappyMock项目整体概述 PAGEREF \_Toc26 \h X

3.1.1 用户特征 PAGEREF \_Toc27 \h X

3.1.2 系统功能性需求 PAGEREF \_Toc28 \h XI

3.1.3 系统非功能性需求 PAGEREF \_Toc29 \h XI

3.2 HappyMock系统的需求分析 PAGEREF \_Toc30 \h XI

3.2.1 需求列表 PAGEREF \_Toc31 \h XI

3.2.2 系统用例 PAGEREF \_Toc32 \h XIII

3.2.3 数据设计 PAGEREF \_Toc33 \h XXV

3.3 HappyMock系统的概要设计 PAGEREF \_Toc34 \h XXV

3.3.1 系统框架结构 PAGEREF \_Toc35 \h XXV

3.3.2 模块设计 PAGEREF \_Toc36 \h XXVII

3.4 本章小结 PAGEREF \_Toc37 \h XXVIII

**第四章 HappyMock项目的详细设计与实现 PAGEREF \_Toc38 \h XXIX**

4.1 HappyMock项目概述 PAGEREF \_Toc39 \h XXIX

4.1.1 用例编辑模块概述 PAGEREF \_Toc40 \h XXIX

4.1.2 DSL解析引擎模块概述 PAGEREF \_Toc41 \h XXIX

4.2 HappyMock项目的详细设计 PAGEREF \_Toc42 \h XXIX

4.2.1 数据库设计 PAGEREF \_Toc43 \h XXIX

4.2.2 类设计 PAGEREF \_Toc44 \h XXX

4.2.3 接口描述 PAGEREF \_Toc45 \h XXXIII

4.3 HappyMock项目的实现 PAGEREF \_Toc46 \h XXXV

4.3.1 创建新用例 PAGEREF \_Toc47 \h XXXV

4.3.2 DSL解析引擎模块解析接口实现 PAGEREF \_Toc48 \h XXXIX

4.4 本章小结 PAGEREF \_Toc49 \h XLIII

**第五章 总结与展望 PAGEREF \_Toc50 \h XLV**

5.1 总结 PAGEREF \_Toc51 \h XLV

5.2 展望 PAGEREF \_Toc52 \h XLV

**参考文献 PAGEREF \_Toc53 \h XLVI**

**致谢 PAGEREF \_Toc54 \h XLVII**

图目录

TOC \t "图目录4, 1"

图2.1 Spring架构图 PAGEREF \_Toc \h IV

图2.2 Netty架构图 PAGEREF \_Toc1 \h VI

图3.1 系统用例图 PAGEREF \_Toc2 \h XIII

图3.2 用户登录顺序图 PAGEREF \_Toc3 \h XV

图3.3 创建新项目顺序图 PAGEREF \_Toc4 \h XVI

图3.4 创建新用例顺序图 PAGEREF \_Toc5 \h XVIII

图3.5 编辑用例基本信息顺序图 PAGEREF \_Toc6 \h XIX

图3.6 编辑用例配置顺序图 PAGEREF \_Toc7 \h XX

图3.7 激活/取消激活顺序图 PAGEREF \_Toc8 \h XXII

图3.8 查询用例顺序图 PAGEREF \_Toc9 \h XXIII

图3.9 删除用例顺序图 PAGEREF \_Toc10 \h XXIV

图3.10 使用用例顺序图 PAGEREF \_Toc11 \h XXV

图3.11 实体关系图 PAGEREF \_Toc12 \h XXV

图3.12 编辑用例系统框架图 PAGEREF \_Toc13 \h XXVI

图3.13 DSL解析引擎框架图 PAGEREF \_Toc14 \h XXVII

图3.14 系统模块图 PAGEREF \_Toc15 \h XXVIII

图4.1 数据库表设计 PAGEREF \_Toc16 \h XXX

图4.2 用例编辑模块类图 PAGEREF \_Toc17 \h XXXI

图4.3 DSL解析引擎模块类图 PAGEREF \_Toc18 \h XXXII

图4.4 编辑新用例图 PAGEREF \_Toc19 \h XXXV

图4.5 添加用例到项目图 PAGEREF \_Toc20 \h XXXVI

图4.6 创建新用例JS代码 PAGEREF \_Toc21 \h XXXVII

图4.7 主控制器代码 PAGEREF \_Toc22 \h XXXVIII

图4.8 子控制器mockItems代码 PAGEREF \_Toc23 \h XXXVIII

图4.9 用例使用效果图 PAGEREF \_Toc24 \h XXXIX

图4.10 HappyMockEngine代码 PAGEREF \_Toc25 \h XL

图4.11 ComposeFactory 代码 PAGEREF \_Toc26 \h XLI

图4.12 BodyTextBinder 代码 PAGEREF \_Toc27 \h XLII

图4.13 UriProcessor 代码 PAGEREF \_Toc28 \h XLIII

表目录

TOC \t "表目录2, 1"

表3.1 用户角色划分表 PAGEREF \_Toc \h X

表3.2 用户角色划分表 PAGEREF \_Toc1 \h XI

表3.3 功能性需求列表 PAGEREF \_Toc2 \h XII

表3.4 非功能性需求列表 PAGEREF \_Toc3 \h XII

表3.5 系统用例表 PAGEREF \_Toc4 \h XIV

表3.6 用户登录描述表 PAGEREF \_Toc5 \h XIV

表3.7 创建新项目描述表 PAGEREF \_Toc6 \h XVI

表3.8 创建新用例描述表 PAGEREF \_Toc7 \h XVII

表3.9 编辑用例基本信息描述表 PAGEREF \_Toc8 \h XIX

表3.10 编辑用例配置描述表 PAGEREF \_Toc9 \h XX

表3.11 激活/取消激活用例描述表 PAGEREF \_Toc10 \h XXI

表3.12 查询用例描述表 PAGEREF \_Toc11 \h XXII

表3.13 删除用例描述表 PAGEREF \_Toc12 \h XXIII

表3.14 使用用例描述表 PAGEREF \_Toc13 \h XXIV

表4.1 HappyMockPersistService类接口规范 PAGEREF \_Toc14 \h XXXIII

表4.2 ResponseBinder类接口规范 PAGEREF \_Toc15 \h XXXIII

表4.3 RequestProcessor类接口规范 PAGEREF \_Toc16 \h XXXIV

表4.4 Compiler类接口规范 PAGEREF \_Toc17 \h XXXIV

第一章 引言

1.1 项目背景

随着SOA[[1]](#footnote-1)（Service-oriented architecture）概念的提出以及最近几年的发展和完善，面向服务的体系架构在业界受到越来越多的重视。与此同时，Roy Fielding在2000年提出了REST[[2]](#footnote-2)的概念，Dropwizard[[3]](#footnote-3)、CXF[[4]](#footnote-4)等可以暴露REST服务的框架也层出不穷，越来越多的公司开始开发满足REST风格的接口。REST风格接口的流行随之也带来了一个问题：如何进行HTTP/HTTPS协议接口的集成测试？比较传统的解决方案可以是类似于单元测试一样写集成测试用例，但是这只适用于接口开发完毕的情况，不支持正在开发过程中或者第三方的接口；当然也可以借助一些开源框架模拟接口，比如HTTPretty[[5]](#footnote-5)、node-mocks-http[[6]](#footnote-6)等，但是这些都或多或少有一些语言上或者功能上的限制。而该测试问题在Web项目中尤其严重，Web项目区别于其他项目的本质原因就在于，它直接面对需求的提出者——客户，而这部分的需求变动也是最为活跃的。也就是因为这样，往往很多时候我们对于web的开发要求的就是——快速！这时候我们对于底层系统的集成往往束手无策？因为底层系统一般来说是公司产品的核心内容，也遵循着典型的产品开发模式，在需求变更方面也显得很谨慎而小心。面对突入其来的的需求变动，底层系统往往很难跟的上节奏，而且很多时候 Web的开发和底层系统的开发是分开的，或者分属于不同的开发部门，即使是一个方法级别的调用，就会浪费掉大家很多的沟通成本，接口的变更或者是 API的向下兼容这些都是翻来覆去争论的话题。由于大家所处的开发领域不同，工作的方式，方法的不同，甚至是编程语言的差别，这些问题最终会暴露在集成测试这里，从而影响整体项目的开发进度。

问题已经暴露出来，那该如何解决该问题呢？我感觉国人工程师郑晔开发的Moco框架给我们提供了一个很好的思路：Moco[[7]](#footnote-7)是一个简单搭建stub的框架，主要用于测试和集成。这个框架的开发灵感来自Mock框架，如Mockito和 Playframework。Moco的出现，将工程师从无聊重复的开发部署工作中解放出来。开发团队只要根据自己的需要进行相应的配置，就会很方便得到一个模拟服务器。

本文介绍的HappyMock项目的灵感正是来自于Moco框架。HappyMock项目提供了简单的图形界面，用户通过配置一些简单易学的外部DSL[[8]](#footnote-8)代码即可以很方便的获得一个模拟服务器，从而大大提高软件工程师的测试开发效率。同时借助于该项目可以搭建一个中央模拟服务器，不仅可以方便团队对单个项目进行测试开发，而且可以抽象出模版，实现测试用例的复用，从而进一步提高测试开发团队的效率。

1.2 国内(外)同类软件研究现状

集成测试，尤其是基于HTTP协议的集成测试——web service、REST 等，在我们的项目开发中被广泛应用。以前，我们每次都要往Jetty或Tomcat 等应用服务器上部署一个新的WAR或者JAR。大家都知道，开发部署一个 WAR或者JAR的过程是很枯燥的，即使在嵌入式服务器上也是如此。而且，每次我们做一点改动，整个 WAR或者JAR都要重新组装。

而随着REST的流行，针对基于HTTP协议接口的集成框架也越来越多，比如Github上面人气比较高的nock，HTTPretty、Mocky、moco、mockserver等，他们基于不同的编程语言开发，很多侧重于单个编程语言的集成测试。虽然各自解决了自己编程语言集成测试方面的问题，但它们在跨语言、跨平台以及可扩展性方面存在一定的问题。

相对而言，国人开发的moco框架一定程度上解决了上面的问题，它基于Java编程语言开发，支持两种使用方式：内嵌于Java代码或者独立运行。虽然独立运行一定程度上解决了跨语言及可扩展性方面的问题，但是却存在其他一些问题，比如不利于复用，没有提供对配置的提前校验功能，不便于操作和使用，不利于团队使用等。而本文介绍的HappyMock项目可以在一定程度弥补moco项目中存在的不足。

1.3 论文的主要工作和组织结构

本文主要介绍HappyMock集成测试系统的工作流程框架和整个项目的的需求、设计、开发等内容。整个HappyMock集成测试系统包括两个模块：用例编辑模块和DSL解析引擎模块。本人参与了整个项目中所有部分的设计开发部署工作。

第一章：引言部分，主要介绍了项目背景，基于HTTP协议接口集成测试技术的发展历史和现状，并且简述了本篇论文的主要工作和组织结构。

第二章：HappyMock系统技术概述，主要介绍了该系统设计开发过程中用到的主要技术，包括Spring框架、Nodejs/Express、Docker部署工具、git版本开发管理工具、jQuery等技术。

第三章：HappyMock系统需求分析与概要设计，主要从需求和概要设计方面介绍了HappyMock系统的整体需求和设计架构，从用户角度和高层设计角度描述了系统的结构。

第四章：HappyMock系统的详细设计与实现，主要介绍了用例编辑模块创建新用例功能和DSL解析模块解析接口的详细设计与实现，描述了系统使用的细节过程。

第五章：总结与展望，对HappyMock系统整个项目进行了总结，描述整个项目的完成情况和进度，以及自己在项目组中的工作，同时，对未来系统进一步的扩展和完善做了一定的规划。

第二章 HappyMock技术概述

2.1 Spring技术

2.1.1 Spring简介

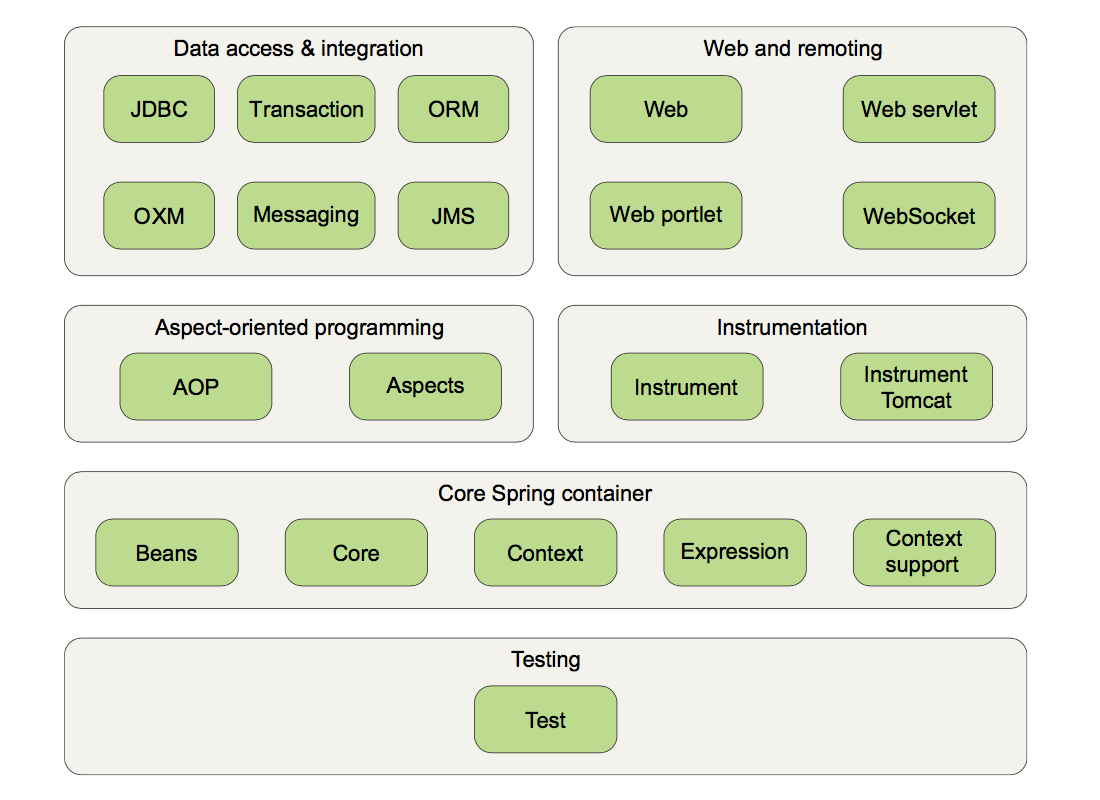
Spring是一个开源框架，最早由Rod Johnson创建。Spring是为了解决企业级应用开发的复杂性而创建的，使用Spring不仅可以让简单的JavaBean实现之前只有EJB才能完成的事情，更可以让任何Java应用都能在简单性、可测试性和松耦合等方面从Spring中受益[1]。Spring的核心是控制反转（IoC）和面向切面（AOP）,大致框架图如图2.1所示：

图2.1 Spring架构图

2.1.2 Spring模块

Spring框架是由六个不同的模块所构成的，这六个模块为开发企业级应用提供了所需的一切。但是也不必将应用建立在整个Spring框架之上，可以自由选择适合自身应用需求的√模块；当Spring不能满足需求时，完全可以考虑其他选择。事实上，Spring甚至还集成了其他第三方框架和类库，提供给开发者自由使用。

下面我们就逐一浏览Spring的六大模块：

* 核心Spring容器

容器是Spring框架最核心的部分，核心容器的主要组件是BeanFactory，它是工厂模式的实现。BeanFactory使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

* Spring的AOP

Spring对面向切面编程提供了丰富的支持，AOP可以帮助应用对象解耦。借助于AOP，可以将遍布应用的关注点（例如事务和安全）从它们所应用的对象中解耦出来。

* 数据访问与集成

包含有JDBC、ORM、OXM、JMS和Transaction模块。其中JDBC模块提供了一个JDBC抽象层，它可以消除冗长的JDBC编码和解析数据库厂商特有的错误代码；ORM模块提供了对许多流行ORM框架的支持，包括Hibernate，JDO等。

* Web和远程调用

包含了Web、Web-Servlet、Web-Struts、Web-Porlet模块。Spring不仅集成了多种流行的MVC框架，而且自带了一个强大的MVC框架—SpringMVC, 两者均有助于应用提升Web层技术的松散耦合。与此同时，该模块还提供了构建与其他应用交互的几种远程调用的选择。Spring远程调用服务集成了RMI，Hessian，Burlap，JAX-WS，且自带了一个远程调用框架：HTTP invoker。

* 监测器

Instrumentation模块提供了为JVM添加代理的支持，使得可以在特定的应用服务器上使用。特别的，它为Tomcat提供了类似于classloader加载转换文件的织入代理。

* 测试

提供对使用JUnit和TestNG来测试Spring组件的支持，它提供一致的ApplicationContexts并缓存这些上下文，它还能提供一些mock对象，使得你可以独立的测试代码。

2.2 Netty技术

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty极大地简化了且加速了网络编程，比如TCP和UDP 的socket服务开发。

Netty的高性能和简单易用并不意味着编写的程序的性能和可维护性会受到影响。Netty的设计得益于实现协议如FTP、SMTP、HTTP、WebSocket、SPDY以及各种二进制和基于文本的传统协议中获得的经验。借助于异步I/O，丰富的缓冲数据结构，基于拦截链模式的事件模型以及适用快速开发的高级组件，Netty成功的提供了易于开发、高性能和高稳定性，以及较强的扩展性[2]。

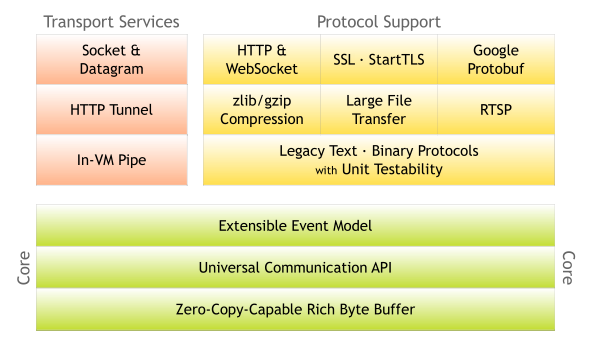
图2.2表示的是Netty框架的组成：

图2.2 Netty架构图

2.3 Nodejs/Express技术

2.3.1 Nodejs简介

Node.js是一个基于Chrome JavaScript运行时的平台，用于方便地搭建响应速度快、易于扩展的网络应用。Node.js 使用事件驱动，非阻塞I/O模型而使得应用轻量和高效，非常适合在分布式设备上运行数据密集型的实时应用[3]。

2.3.2 Nodejs特性

Nodejs作为一个新兴的后台语言，特别适合高并发，I/O密集型应用。它具有如下特性：

* 事件驱动
* 非阻塞I/O
* 单线程、单进程
* 依赖于Chrome V8引擎进行代码解释
* 轻量、可伸缩，适于实时数据交互应用

2.3.3 Express简介

Express 是一个简洁、灵活的 node.js Web 应用开发框架, 它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用[4]。

2.4 Docker技术

Docker是一个让开发者和系统管理员开发、发布、运行应用的平台。Docker可以让我们快速将组件组装成应用和在发布代码的时候避免冲突。Docker可以使我们的代码尽可能快的获得测试和部署到生成环境[5]。

简单的说Docker是一个构建在LXC（Linux Container）之上的,基于进程容器(Process container)的轻量级VM解决方案。

2.5 Bootstrap技术

Bootstrap是一个用于构建响应式网站的前端框架。它将HTML、CSS和JavaScript组合起来，为构建稳定的网站提供了基础设施，比如一系列丰富的Web组件和JQuery插件，提高了开发效率[6]。

2.6 Less＋Jade技术

2.6.1 Less简介

Less CSS是一个使用广泛的CSS预处理器，Less扩展了CSS的动态行为，比如说，设置变量（Variables）、混合书写模式（mixins）、操作（operations）和功能（functions）等等，大大简化了CSS的编写，并且降低了CSS的维护成本[7]。

2.6.2 Jade简介

Jade是一个高效的HTML模版引擎，是Haml的JavaScript实现，在服务端（NodeJS）及客户端均有支持[8]。具有如下特性：

* 生成HTML
* 支持动态代码
* 支持代码复用

2.7 JQuery技术

2.7.1 JQuery简介

jQuery是一个由美国人John Resig于2006年创建的一个开源项目，目前已发展成为集JavaScript、CSS、DOM、Ajax于一体的强大框架体系，核心理念是write less,do more(写得更少,做得更多)[9]。

2.7.2 JQuery基本功能

* 访问和操作DOM元素
* 控制页面样式
* 处理页面事件
* 在页面中运用大量插件
* 与Ajax技术的完美结合

2.8 Git技术

Git是用于Linux内核开发的版本控制工具。与CVS、Subversion一类的集中式版本控制工具不同，它采用了分布式版本库的作法，不需要服务器端软件，就可以运作版本控制，使得源代码的发布和交流极其方便。Git的速度很快，这对于诸如Linux内核这样的大项目来说自然很重要。Git最为出色的是它的合并追踪（merge tracing）能力。

2.9 本章小结

本章主要介绍了HappyMock项目开发过程中使用到的主要技术和工具，借助上面的技术和工具，我们实现了一个简单的集成测试框架，并且使项目处于安全有效的管理下。

第三章 HappyMock系统需求分析与概要设计

3.1 HappyMock项目整体概述

HappyMock系统是一个用于对HTTP/HTTPS协议接口进行集成测试的框架，一方面系统需要提供给用户一个编辑用例，进行交互的界面，该页面既要简单易懂，又需要对输入的外部DSL进行必要的验证，以降低错误率，提高工作效率；另一方面需要实现一个外部DSL的解析引擎，该引擎需要根据用户请求的内容和预先编辑的用例返回对应的结果，以期达到mock基于HTTP/HTTPS协议接口的目的。上述的两部分协调工作，共同搭建起企业集成测试的整体架构，提高企业以及个人的生产效率。

3.1.1 用户特征

系统的用户群体主要是软件行业从业者，尤其是测试工程师，针对不同的使用场景，用户角色可以细分为：用例编辑人员和用例使用人员。当然部分情况下，同一个人可以同时可以扮演这两个角色。图3.1介绍两种角色的具体特征。

|  |  |  |
| --- | --- | --- |
| 涉众 | 特征 | 主要目标 |
| 用例编辑人员 | 熟练使用计算机，熟练掌握若干计算机脚本语言或者其他编程语言，熟悉HTTP协议的具体内容。 | 希望界面尽量简洁清晰，可以快速编辑测试用例，当用例含有语法等错误时及时给出提示，同时希望可以复用已有测试用例。 |
| 用例使用人员 | 熟练使用计算机，熟练掌握若干计算机编程语言，熟悉HTTP协议的具体内容。 | 希望可以mock出需要的大部分接口，而且当切换到正式接口时，尽量降低修改的代码量。 |

表3.1 用户角色划分表

3.1.2 系统功能性需求

根据用户的角色和职责，HappyMock系统的主要需求如表3.2所示：

|  |  |
| --- | --- |
| 用户角色 | 用户需求 |
| 用例编辑人员 | 新建项目 |
| 新建/修改/删除查询用例 |
| 激活/取消激活用例 |
| 用户登录 |
| 用例使用人员 | 根据请求获取配置内容 |

表3.2 用户角色划分表

3.1.3 系统非功能性需求

* 可用性

系统可用性达到90%；

系统发生故障时能够在半小时恢复。

* 可维护性/可扩展性

增加对更多HTTP协议中关键字的支持，可以在一周内完成。

* 易用性

提供简单易于使用的界面，操作过程中提供必要的提示。

* 安全性

用例编辑人员只能看到自己的用例，不能获取修改删除他人数据。

3.2 HappyMock系统的需求分析

3.2.1 需求列表

系统的功能性需求列表如表3.3所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 需求编号 | 需求名称 | 需求内容 | 优先级 |
| R1 | 用户登录 | 用户通过Url直接登录，存在两种可能：   1. 已经存在该用户，显示用户自己的用例列表； 2. 用户不存在，创建一个新用户纪录 | 中 |
| R2 | 管理项目 | 1. 创建一个新项目 | 中 |
| R3 | 管理用例 | 1. 创建一个新用例； 2. 修改已有用例内容； 3. 删除已有用例； 4. 激活用例； 5. 取消激活用例； 6. 查询用例。 | 高 |
| R4 | 使用用例 | 1. 使用用例 | 高 |

表3.3 功能性需求列表

系统的非功能性需求列表如表3.4所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 需求编号 | 需求名称 | 需求内容 | 优先级 |
| R5 | 可用性 | 1. 系统可用性达到90%； 2. 发生故障时可以在半小时恢； | 高 |
| R6 | 可维护性/可扩展性 | 1. 增加新关键字，可以快速完成功能； 2. 代码可读性较好，出现问题可以快速解决。 | 中 |
| R7 | 可用性 | 1. 界面简单易操作； 2. 提供必要提示。 | 中 |
| R8 | 安全性 | 1. 用户不能查看修改删除别人用例。 | 低 |

表3.4 非功能性需求列表

3.2.2 系统用例

系统用例如图3.1所示。

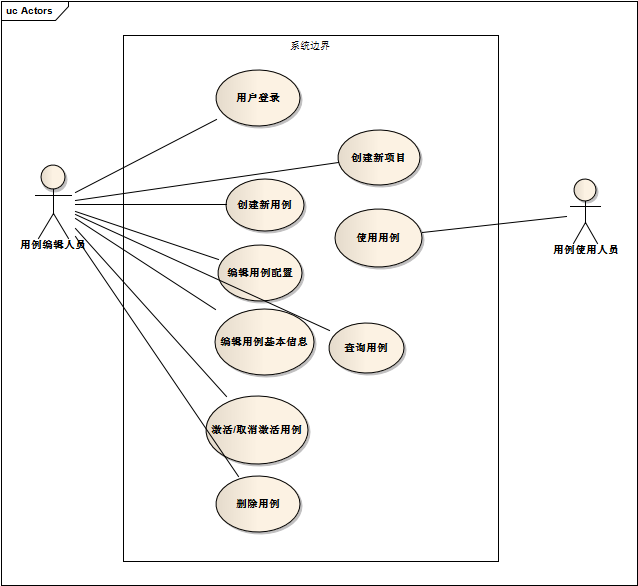


图3.1 系统用例图

系统用例表如表3.5所示。

|  |  |  |
| --- | --- | --- |
| 用例编号 | 用例名称 | 对应需求 |
| C1 | 用户登录 | R1 |
| C2 | 创建新项目 | R2 |
| C3 | 创建新用例 | R3 |
| C4 | 编辑用例基本信息 | R3 |
| C5 | 编辑用例配置 | R3 |
| C6 | 激活/取消激活用例 | R3 |
| C7 | 查询用例 | R3 |
| C8 | 删除用例 | R3 |
| C9 | 使用用例 | R4 |

表3.5 系统用例表

下面对系统用例进行详细描述。

**用例1 用户登录**

|  |  |
| --- | --- |
| 用例编号 | C1 |
| 用例名称 | 用户登录 |
| 用例描述 | 用户通过直接在Url中输入用户名登录，暂时不需要密码 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 系统启动 |
| 后置条件 | 系统显示登录用户的用例列表 |
| 优先级 | 中 |
| 正常过程 | 1. 用户在url中输入自己的用户名登录； 2. 系统显示登录用户的用例列表。 |
| 扩展流程 | 2a. 登录的用户本来不存在，显示一个空的用例列表。 |
| 特殊需求 | 无 |

表3.6 用户登录描述表

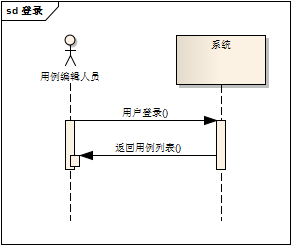
系统顺序图如图3.2所示。

图3.2 用户登录顺序图

**用例2 创建新用例**

|  |  |
| --- | --- |
| 用例编号 | C2 |
| 用例名称 | 创建新项目 |
| 用例描述 | 用户创建新的项目（分类） |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 用户已经成功登录 |
| 后置条件 | 系统显示创建新项目的结果 |
| 优先级 | 高 |
| 正常过程 | 1. 用户选择一个领域（Domain）； 2. 用户选择新建项目； 3. 用户输入项目名称； 4. 用户提交保存； 5. 系统给出新建结果，同时更新项目列表。 |
| 扩展流程 | 4a. 用户输入为空，提交按钮不能点击；  5a. 如果用户输入的项目名称已经存在，则系统给出创建失败提示； |
| 特殊需求 | 无 |

表3.7 创建新项目描述表

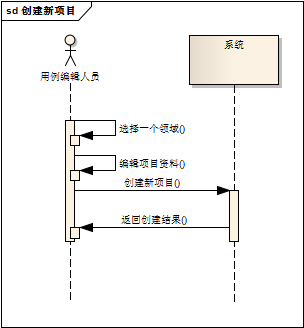
系统顺序图如图3.3所示。

图3.3 创建新项目顺序图

**用例3 创建新用例**

|  |  |
| --- | --- |
| 用例编号 | C3 |
| 用例名称 | 创建新用例 |
| 用例描述 | 用户创建新的用例 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 已经选定一个领域（Domain） |
| 后置条件 | 系统给出创建结果提示，同时更新用例列表 |
| 优先级 | 高 |
| 正常过程 | 1. 用户选择一个领域（Domain）； 2. 用户填写用例配置； 3. 用户选择添加到项目中； 4. 用户填写用例基本信息； 5. 用户提交保存； 6. 系统给出创建结果提示，同时更新用例列表 |
| 扩展流程 | 2a. 用例配置为空，不能点击添加到项目中按钮；  3a. 会对用例内容进行验证，如果不符合DSL语法将会给出提示，不能添加到项目中；  4a. 填写用户项目信息时，有两个选择：  4b. 从已有项目中选择；  4c. 创建一个新项目  5a. 用例必填基本信息如果为空，提交保存按钮不能点击； |
| 特殊需求 | 由于用例配置内容是Json格式数据，所以希望文本编辑器支持Json格式数据，且具有一些简单的补全功能。 |

表3.8 创建新用例描述表

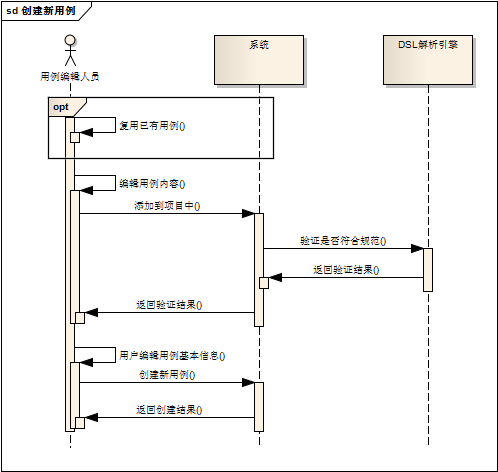
系统顺序图如图3.4所示。

图3.4 创建新用例顺序图

**用例4 编辑用例基本信息**

|  |  |
| --- | --- |
| 用例编号 | C4 |
| 用例名称 | 编辑用例基本信息 |
| 用例描述 | 用户可以编辑用例名称，描述等基本信息 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 用户选择编辑用例基本信息 |
| 前置条件 | 用例已经存在 |
| 后置条件 | 系统显示编辑结果，同时显示编辑后的用例信息 |
| 优先级 | 低 |
| 正常过程 | 1. 用户选择编辑某个用例基本信息； 2. 用户修改用例基本信息； 3. 用户提交保存； 4. 系统给出编辑结果；同时显示编辑后的用例信息 |
| 扩展流程 | 3a. 必填信息如果为空，不能点击提交按钮；  4a. 如果用例名称已经存在，提示编辑失败； |
| 特殊需求 | 无 |

表3.9 编辑用例基本信息描述表

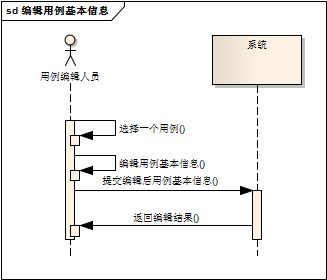
系统顺序图如图3.5所示。

图3.5 编辑用例基本信息顺序图

**用例5 编辑用例配置**

|  |  |
| --- | --- |
| 用例编号 | C5 |
| 用例名称 | 编辑用例配置 |
| 用例描述 | 用户编辑用例配置内容 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 用例为未激活状态 |
| 后置条件 | 系统显示编辑结果，同时更新编辑后的用例配置内容 |
| 优先级 | 高 |
| 正常过程 | 1. 用户选择特定用例； 2. 用户修改配置内容； 3. 保存； |
| 扩展流程 | 3a. 用例状态如果为未激活，则保存按钮不能点击  3b. 用例配置内容不符合DSL语法，系统给出提示，编辑失败 |
| 特殊需求 | 文本编辑器支持Json格式数据，且具有一些简单的补全功能。 |

表3.10 编辑用例配置描述表

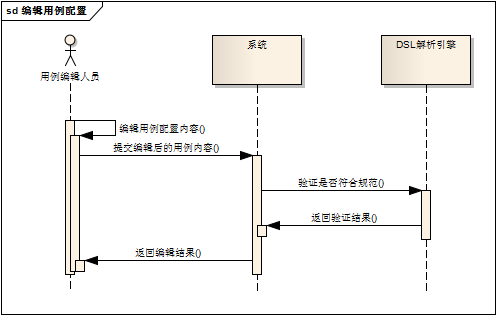
系统顺序图如图3.6所示。

图3.6 编辑用例配置顺序图

**用例6 激活/取消激活用例**

|  |  |
| --- | --- |
| 用例编号 | C6 |
| 用例名称 | 激活/取消激活用例 |
| 用例描述 | 用户可以通过激活/取消激活用例来控制用例的上下线，只有激活状态的用例才能使用 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 用例已经存在 |
| 后置条件 | 系统给出操作之后的效果 |
| 优先级 | 中 |
| 正常过程 | 1. 用户选择特定用例； 2. 用户选择激活或者取消激活操作； 3. 系统更新操作成功之后的效果； |
| 扩展流程 | 无 |
| 特殊需求 | 1. 激活和取消激活按钮不可以同时可以点击； 2. 查看用例详情和用例列表的时候都可以执行激活和取消激活操作； |

表3.11 激活/取消激活用例描述表

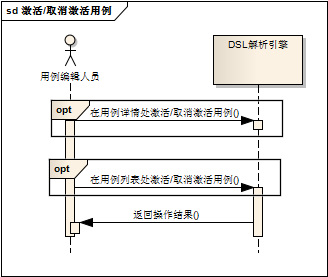
系统顺序图如图3.7所示。

图3.7 激活/取消激活顺序图

**用例7 查询用例**

|  |  |
| --- | --- |
| 用例编号 | C7 |
| 用例名称 | 查询用例 |
| 用例描述 | 用户可以通过关键字查询特定用例 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 以及选择某个领域（Domain） |
| 后置条件 | 系统显示查询结果 |
| 优先级 | 中 |
| 正常过程 | 1. 用户输入查询的关键字 2. 用户选择查询 3. 系统显示查询结果 |
| 扩展流程 | 3a. 如果关键字为空，则默认显示所有用例；  3b. 如果查询结果为空，系统给出提示； |
| 特殊需求 | 1. 支持回车提交查询请求； |

表3.12 查询用例描述表

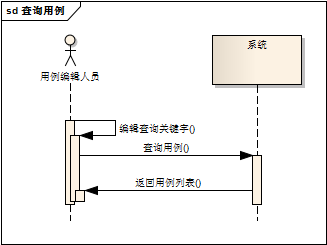
系统顺序图如图3.8所示。

图3.8 查询用例顺序图

**用例8 删除用例**

|  |  |
| --- | --- |
| 用例编号 | C8 |
| 用例名称 | 删除用例 |
| 用例描述 | 用户可以删除已经存在的用例 |
| 参与者 | 用例编辑人员 |
| 触发条件 | 无 |
| 前置条件 | 用例已经存在 |
| 后置条件 | 系统提示删除结果，同时更新删除操作之后的用例列表 |
| 优先级 | 低 |
| 正常过程 | 1. 用户选择特定用例； 2. 用户选择删除； 3. 系统再次确认是否删除； 4. 用户确认删除； 5. 系统给出删除结果，同时更新用例列表； |
| 扩展流程 | 5a. 用例如果为激活状态，给出不能删除提示； |
| 特殊需求 | 无 |

表3.13 删除用例描述表

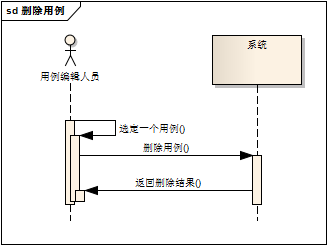
系统顺序图如图3.9所示。

图3.9 删除用例顺序图

**用例9 使用用例**

|  |  |
| --- | --- |
| 用例编号 | C9 |
| 用例名称 | 使用用例 |
| 用例描述 | 用户使用用例进行集成测试或者开发 |
| 参与者 | 用例使用人员 |
| 触发条件 | 无 |
| 前置条件 | 系统开启，用例存在且为激活状态 |
| 后置条件 | 系统显示用户请求的结果 |
| 优先级 | 高 |
| 正常过程 | 1. 用户编辑请求内容； 2. 系统根据用户请求内容和用例编辑人员的配置内容返回响应； |
| 扩展流程 | 2a. 如果用户编辑请求不符合规范，则给出错误提示； |
| 特殊需求 | 1. 返回速度<1s; |

表3.14 使用用例描述表

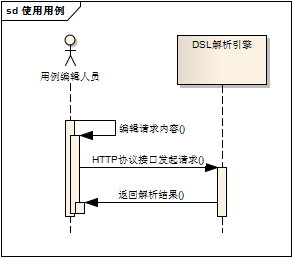
系统顺序图如图3.10所示。

图3.10 使用用例顺序图

3.2.3 数据设计

根据上述用例，系统存在的实体对象有：领域、项目、用例、用例编辑人员。实体关系如图3.11所示。

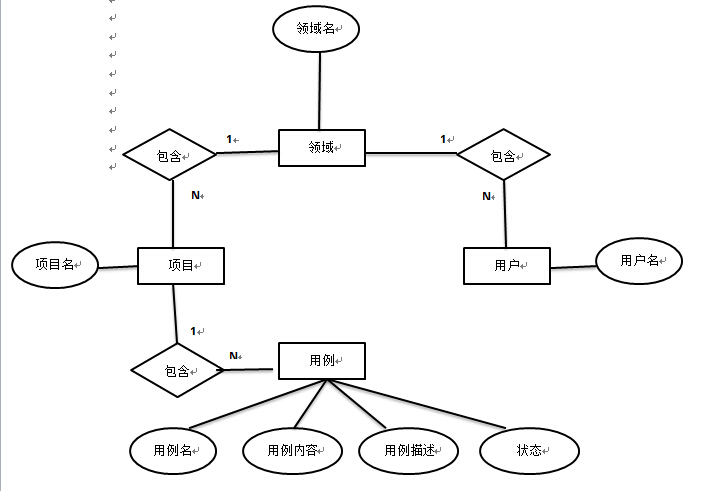
****

图3.11 实体关系图

3.3 HappyMock系统的概要设计

3.3.1 系统框架结构

整个HappyMock系统包含两个部分：用户编辑用例系统和DSL解析引擎。

用户编辑用例系统采用Express框架开发，整个结构不需要过多设计，大致结构如图3.12所示。系统前后端分离，通过Ajax调用后端接口；后端使用Express提供REST接口，所有控制器由一个主控制器控制，并将不同的请求路由到不同的子控制器。

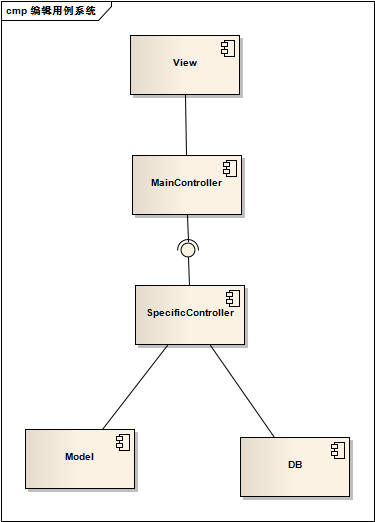


图3.12 编辑用例系统框架图

DSL解析引擎可以看作是简单的分层架构，总共可以分为：数据持久层、解析层、数据转换层，如图3.13所示。Handler模块负责转换用户的输入输出数据，HappyMockEngine从Handler模块拿到用户输入数据，然后根据用户输入从持久层获取满足条件的数据，然后转交给ComposeFactory进行实际的DSL解析，最后解析出MociSetting对象，再经过Handlers模块转化之后输出给用户。

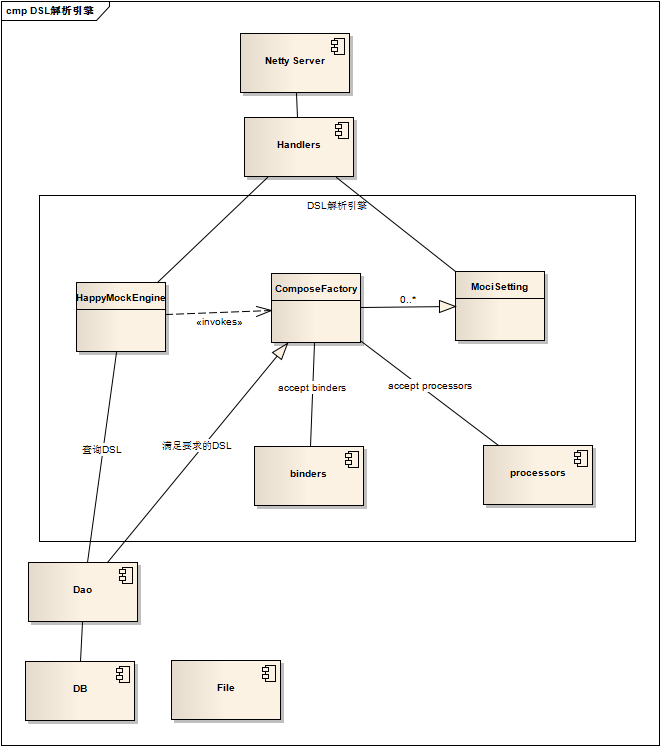


图3.13 DSL解析引擎框架图

3.3.2 模块设计

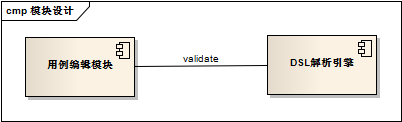
本系统主要包含两个系统，也就大致可以分为两大模块：用例编辑模块和DSL解析引擎模块，如图3.14所示。其中用例编辑模块需要使用DSL解析引擎的模块提供的验证用例内容是否符合DSL语法的接口。

图3.14 系统模块图

3.4 本章小结

本章主要介绍了HappyMock项目的需求分析和概要设计。根据项目的背景，得出了系统功能点和用户特征，再从用例图、用例描述、实体关系图等多个方面进行用例分析，得出分析模型；最后再根据需求分析，划分出具体模块，明确模块职责和之间的协作关系。

第四章 HappyMock项目的详细设计与实现

4.1 HappyMock项目概述

4.1.1 用例编辑模块概述

用例编辑模块主要是给用例编辑人员提供一个编辑用例的入口，用例编辑人员通过该模块提供的界面完成对用例的增删改查等操作，同时提供一些必要的输入验证功能。

4.1.2 DSL解析引擎模块概述

DSL解析引擎是该系统的核心环节，主要提供两个功能：验证用例内容是否符合DSL语法，根据用户输入和已有用例返回解析结果。

4.2 HappyMock项目的详细设计

4.2.1 数据库设计

由上一章实体关系图3.11我们可以发现，整个系统虽然有两个子系统，但是仅包含Domain、User、 Resource和MockItem四个实体。根据实体关系图，我们的数据表设计如图4.1所示。其中黄色表示主键。

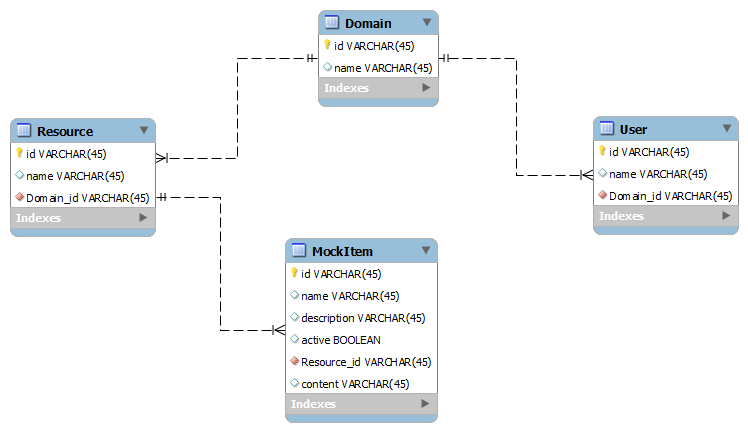
****

图4.1 数据库表设计

4.2.2 类设计

整个项目的详细设计可以划分为两个模块：用例编辑模块和DSL解析引擎。

用例编辑模块的类设计如图4.2所示。由于本模块采用Express框架开发，所以整个架构均按照该框架约定，不需要过多额外设计。另外由于我们是借助Express封装REST接口，所以在控制器中的方法也不好在类图中体现出来。整个模块一共抽象出User、Domain、Resource和MockItem四个实体类，控制器层使用实体类直接操作数据库，然后由App这个主控制器将用户请求路由给下面的子控制器。

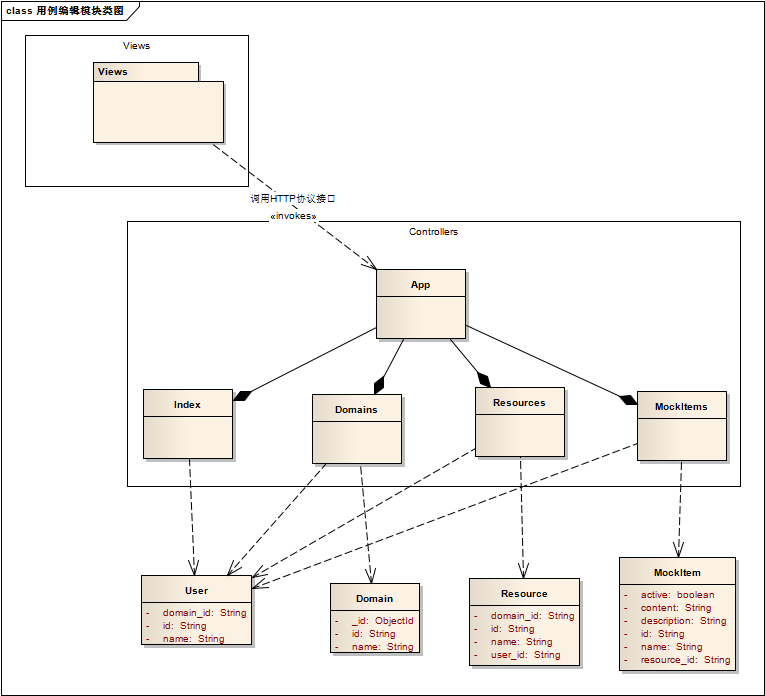
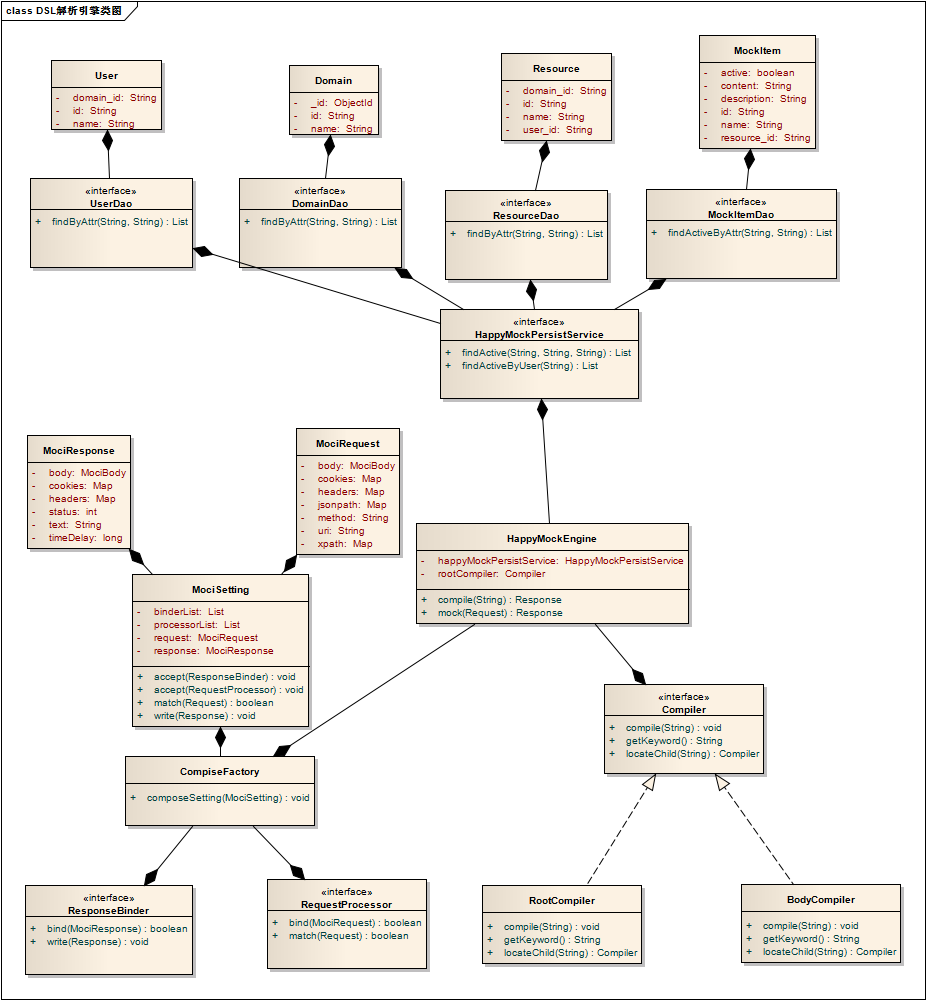


图4.2 用例编辑模块类图

DSL解析引擎模块的类设计如图4.3所示。该模块可以大致分为数据持久层、解析层、数据转换层。这里主要展现了数据持久层和最核心的解析层，其中数据持久层也可以抽象出User、Domain、Resource和MockItem四个实体类，和用例编辑模块一致。数据持久层对外提供HappyMockPersistService接口，更加利于程序的扩展，目前就分别实现了对MongoDB和文件数据的读取支持。而该项目最核心的解析层将用户HTTP请求的Request和Response分别抽象成MociRequest和MociResponse对象，整个完整的HTTP请求抽象成包含MociRequest和MociResponse两个成员变量的MociSetting对象。整个解析层的入口类是HappyMockEngine，该类主要实现两个接口：mock和compile，分别对应解析和验证两个功能。解析功能主要依赖Compiler接口，解析功能主要依赖ComposeFactory类和RequestProcessor、ResponseBinder两个接口。这里面向接口编程，非常利于程序的扩展，每增加一个HTTP关键字，我们可以直接实现Compiler、RequestProcessor和ResponseBinder接口，而不需要修改原有代码。

另外这里省略了数据转换层的类图，数据转换层主要是将netty中的FullHttpRequest、DefaultFullHttpResponse和我们自己定义的Request、Response对象转换。由于功能比较简单，所以这里就不画出类图了。

|  |
| --- |
|  |

图4.3 DSL解析引擎模块类图

4.2.3 接口描述

上一节4.2.2只是给出了两个模块的类图，为了更好的理解我们整个系统的类设计，下面我们描述一下DSL解析引擎模块用到的一些重要接口。

如表4.1所示，该接口是数据持久层提供给解析层的接口规范。

|  |  |  |
| --- | --- | --- |
| 方法 | 语法 | public List<MockItem> findActive(String userName, String domainName, String resourceName) |
| findActive | 功能 | 根据用户名、领域名和项目名查找激活状态的用例 |
| 前置条件 | userName、domainName和resourceName不为空 |
| 后置条件 | 返回一个用例列表，如果不存在满足条件的用例，返回一个空列表 |

表4.1 HappyMockPersistService类接口规范

表4.2是ResponseBinder类接口的规范，主要用来绑定用户返回数据。

|  |  |  |
| --- | --- | --- |
| 方法 | 语法 | public void write(Response response) |
| write | 功能 | 将数据写入Response |
| 前置条件 | Response不为空 |
| 后置条件 | 数据写入Response |
| 方法 | 语法 | public boolean bind(MociResponse response) |
| bind | 功能 | 绑定MociResponse数据 |
| 前置条件 | MociResponse不为空 |
| 后置条件 | 完成绑定MociResponse数据 |

表4.2 ResponseBinder类接口规范

表4.3是RequestProcessor的接口规范。主要用来匹配用户请求。

|  |  |  |
| --- | --- | --- |
| 方法 | 语法 | public boolean match(Request httpRequest) |
| match | 功能 | 判断是否匹配Request |
| 前置条件 | Request不为空 |
| 后置条件 | 返回匹配结果，不匹配返回false，否则返回true。 |
| 方法 | 语法 | public boolean bind(MociRequest mociRequest) |
| bind | 功能 | 绑定MociRequest数据 |
| 前置条件 | MociRequest不为空 |
| 后置条件 | 完成绑定MociRequest数据 |

表4.3 RequestProcessor类接口规范

表4.4是Compiler的接口规范。主要用来验证用例内容是否符合DSL语法规范。

|  |  |  |
| --- | --- | --- |
| 方法 | 语法 | public void compile(String dsl) throws CompileException |
| compile | 功能 | 编译给定的DSL |
| 前置条件 | 无 |
| 后置条件 | 如果不符合DSL语法，会抛出异常；否则正常运行 |
| 方法 | 语法 | public String getKeyword() |
| getKeyword | 功能 | 获取关键字 |
| 前置条件 | 无 |
| 后置条件 | 返回关键字 |
| 方法 | 语法 | public Compiler locateChild(String keyword) |
| locateChild | 功能 | 查找子Compiler |
| 前置条件 | keyword不为空 |
| 后置条件 | 返回子Compiler |

表4.4 Compiler类接口规范

4.3 HappyMock项目的实现

由于整个项目由两个模块组成，不可能逐一介绍所有用例的实现，这里就介绍一下用例编辑模块创建新用例和DSL解析引擎模块的解析接口的实现。

4.3.1 创建新用例

该功能大致流程是：用户选择重新编写或是复用一个已有用例，然后将编辑好的用例添加到一个项目（resource）中。图4.4和4.5是该功能的前端页面截图，因为用例具体内容是Json格式数据，为了减少用户输入的错误以及提供一些简单的补全功能，我们使用了开源项目jsoneditor[11]。

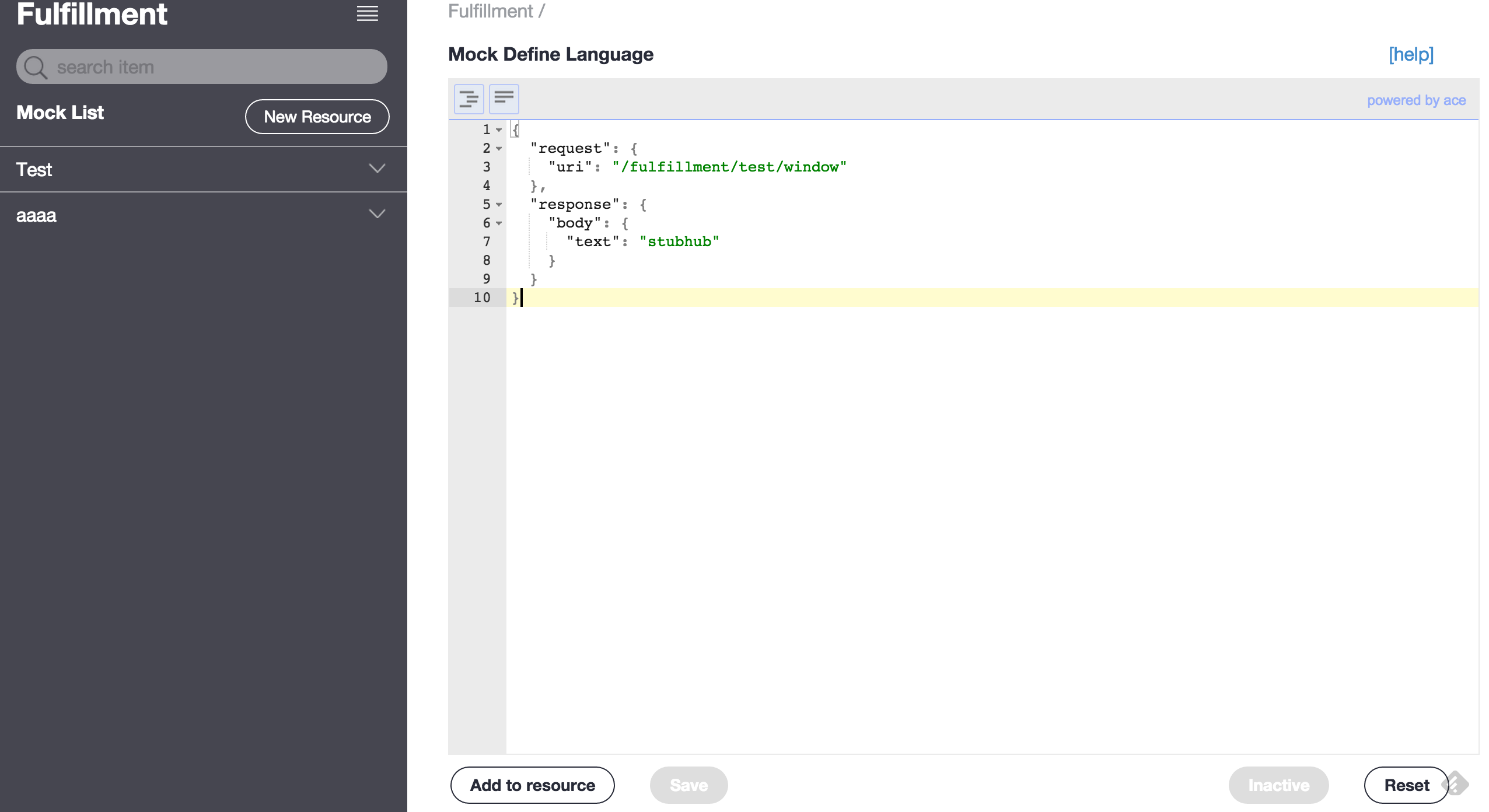


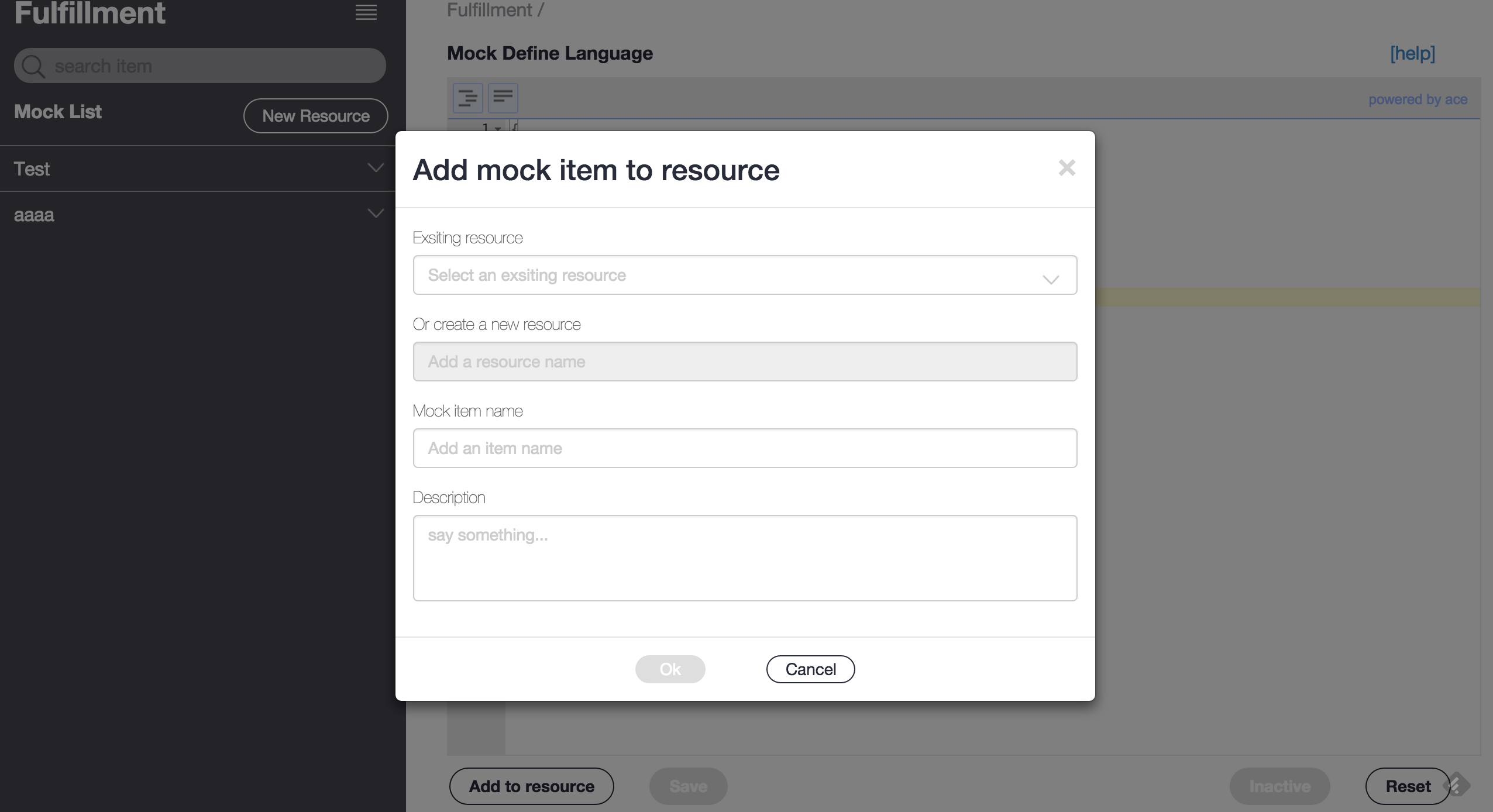
图4.4 编辑新用例图

图4.5 添加用例到项目图

View层的Js代码如图4.6所示，当用户编辑好之后，经过一些必要的JS提前校验，会将用户编辑的数据通过Express框架提供的REST接口提交到服务端，然后创建一个新的用例条目。View层大量采用Ajax技术，可以给用户提供良好的使用体验。

而服务端REST接口实现代码如图4.7和4.8所示。图4.7是服务端主控制器，当前端请求时，主控制器将请求路由到具体的字控制器，子控制器具体实现了REST接口，如图4.8所示，就是对用例新增接口的实现。

|  |
| --- |
| (function(window, $, undefined){  var mockService = {  attachEvents: function(){  $('#createItemModel')  .on('click', '.btn-right', function() { //create a new item  //get data from inputBox && validate if the itemName is existing  var isExist = false;  if(isExist) {  var $alert = $createItemModel.find('.alert');  $alert.text('Item Name already exists!');  $alert.removeClass('hidden').addClass('alert-danger');  setTimeout(function() {  $alert.addClass('hidden').removeClass('alert-danger');  }, 2000);  $name.val('');  $(this).addClass('disabled');  } else {  mockService.createNewItem(global.domain.id, name, content, resource, description);  }  })  },    createNewItem: function createNewItem(domain, name, content, resource,description) {  $.ajax({  url: '/mock/items',  type: 'post',  dataType : "json",  data: {name: name, domain: domain, resource: resource, description: description, content: content},  success: function(data) {  //do something  }  });  },  init: function(){  this.attachEvents();  },  };  mockService.init();  }(window || this, jQuery, undefined)); |

图4.6 创建新用例JS代码

|  |
| --- |
| var mockItems = require('./routes/mockItems');  var app = express();  //config the router  app.use('/', routes);  app.use('/mock/items', mockItems);  app.use('/mock/domains', domains);  app.use('/mock/resources', resources); |

图4.7 主控制器代码

图4.8 子控制器mockItems代码

|  |
| --- |
| var router = express.Router();  var MockItem = require('../models/mockItem');  var Resource = require('../models/resource');  router.route('/')  .post(function(req, res) { //crate a new item.  var mockItem = new MockItem();  mockItem.name = req.body.name;  mockItem.content = req.body.content;  mockItem.description = req.body.description;  var domainName = req.body.domain;  var resourceName = req.body.resource;  Resource.findOne({name: resourceName, domain\_id: domainName}, function(err, resource) {  mockItem.resource\_id = resource.\_id;  mockItem.save(function(err) {  if(err) {  res.send(err);  }  res.json(mockItem);  });  });  }); |

4.3.2 DSL解析引擎模块解析接口实现

正如上文4.2章节中所介绍的，DSL解析引擎的解析接口是该系统最核心的部分。如图4.9所示，我们使用4.3.1节中用到的例子发起请求，然后系统返回给我们了之前配置的stubhub。

该实例的实现代码如图4.10、4.11、4.12和4.13所示。DSL解析引擎会将用户输入的FullHttpRequest转换成系统中定义的Request，然后ComposeFactory根据用户的Request组装MociSetting，最后选出最匹配的MociSetting，将结果返回给用户。

其中组装MociSetting主要是在ComposeFactory类中实现。该类会遍历所有的RequestProcessor和ResponseBinder，然后组装所有满足的RequestProcessor和ResponseBinder。

图4.12和4.13分别给出了ResponseBinder和RequestProcessor的一个实现，每当有新的关键字需要扩展，可以通过实现两个接口就可以轻松实现。

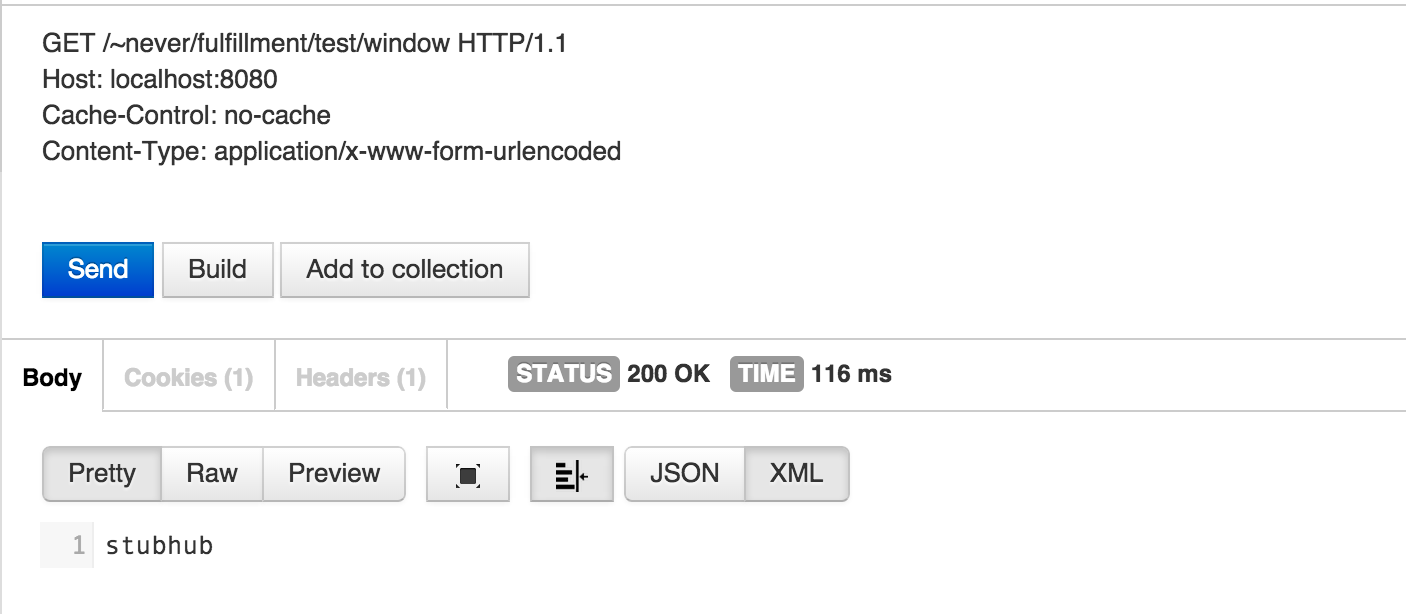


图4.9 用例使用效果图

|  |
| --- |
| public Response mock(Request request) throws Exception {  LOG.info("start mock request at url={} under utx={} ", request.getUrl(), request.getUsername());  Response response = new Response();  //load all mock specs under the user ctx  List<MockItem> mockItemList = happyMockPersistService.find(request.getUsername());  List<MociSetting> mociSettings = Lists.transform(mockItemList, new Function<MockItem, MociSetting>() {  @Override  public MociSetting apply(MockItem input) {  MociSetting mociSetting = JSON.parseObject(input.getContent(), MociSetting.class);  ComposeFactory.getInstance().composeSetting(mociSetting);  return mociSetting;  }  });  //find the best match mock specs  Optional<MociSetting> bestmatch = Optional.absent();  for (MociSetting mociSetting : mociSettings) {  if (mociSetting.match(request)) {  LOG.debug("find the matched DLS=" + mociSetting.getRequest().toString());  if (!bestmatch.isPresent() || bestmatch.get().matched() < mociSetting.matched()) {  bestmatch = of(mociSetting);//reassign the best match  }  }  }  //write mock specs to response  if (bestmatch.isPresent()) {  response.setRequest(request);  bestmatch.get().write(response);  //time delay is in millseconds unit  if (response.getTimeDelay() != null) {  Thread.sleep(response.getTimeDelay());  }  } else {  LOG.debug("find no matched DLS,return http code 404");  response.setCode(404);//does not find matched settings  }  return response;  } |

图4.10 HappyMockEngine代码

|  |
| --- |
| public class ComposeFactory {  private ComposeFactory() {  }  private static class ComposeFactoryHolder{  private static final ComposeFactory FACTORY\_INSTANCE=new ComposeFactory();  }  public static ComposeFactory getInstance(){  return ComposeFactoryHolder.FACTORY\_INSTANCE;  }  public void composeSetting(MociSetting mockiSetting){  //bind processor  for(RequestProcessor requestProcessor: RequestProcessorFactory.getInstance().getProcessList()){  mockiSetting.accept(requestProcessor);  }  //bind binder  for(ResponseBinder responseBinder: ResponseBinderFactory.getInstance().getBinderList()){  mockiSetting.accept(responseBinder);  }  }  } |

图4.11 ComposeFactory 代码

|  |
| --- |
| public class BodyTextBinder implements ResponseBinder {  private static final Logger LOG = LoggerFactory.getLogger(BodyTextBinder.class);  private String text;  @Override  public void write(Response response) {  checkNotNull(text);  response.setText(response.buildDynamicContent(text));  }  @Override  public boolean bind(MociResponse response) {  if(null!=response.getBody()&&!Strings.isNullOrEmpty(response.getBody().getText())){  this.text=response.getBody().getText();  return true;  }  return false;  }  } |

图4.12 BodyTextBinder 代码

|  |
| --- |
| public class UriProcessor implements RequestProcessor {  private String uri;  private AntPathMatcher antPathMatcher;  public UriProcessor() {  this.antPathMatcher = new AntPathMatcher();  }  @Override  public boolean match(Request httpRequest) {  if(Strings.isNullOrEmpty(httpRequest.getUrl())){  return false;  }  if(antPathMatcher.isPattern(this.getUri())){  return antPathMatcher.match(this.getUri().toLowerCase(), httpRequest.getUrl().toLowerCase());  }else{//match based on plain text  return this.getUri().equals(httpRequest.getUrl());  }  }  @Override  public boolean bind(MociRequest mociRequest) {  if (mociRequest.getUri() != null) {  this.setUri(mociRequest.getUri());  return true;  }  return false;  }  public String getUri() {  return uri;  }  public void setUri(String uri) {  this.uri = uri;  }  } |

图4.13 UriProcessor 代码

4.4 本章小结

本章节主要介绍了HappyMock项目的详细设计与实现。详细设计主要包括数据库设计，类图设计以及一些关键类的接口规范。而在具体实现模块，由于篇幅原因，不能逐一介绍，所以只介绍了其中必要重要的部分：创建新用例和DSL解析引擎解析接口的具体实现。通过这两个小功能的介绍，我们对整个系统的具体实现和技术细节有了大致的把握。

第五章 总结与展望

5.1 总结

总体来说，HappyMock项目版本一完成度较好。目前为止，本系统已经实现Request对URI、HTTP Method、Cookie、Header、JsonPath、XPath以及Text、Xml、Json三种不同格式的body的支持；也实现了Response对Status、Text、Header、Cookie以及Text、Xml、Json三种不同格式的body的支持。同时用例编辑模块页面借助jsoneditor等开源项目以及较多细致的页面处理，实现了页面简单易操作的目标，基本可以满足公司对基于HTTP协议接口集成测试的需求。

整个实习过程中，在两个导师的指导下，我从整个项目搭建开始，一步一个脚印，遇到了很多技术问题，但是最终依然是完成了整个项目。整个项目过程中，我学习到了很多东西，开始接触使用git、nodejs、mongodb等一系列新技术，开始编写测试用例，而DSL解析引擎最终在导师的指导下实现，更是让我收获良多。整个项目的顺利完成为我步入软件行业打下了坚实的基础。

5.2 展望

虽然整个系统已经可以正常运转，但是我们依然可以发现很多可以继续提高改善的地方。

首先从DSL解析引擎模块考虑，我们依然可以添加对更多关键字的支持，而目前该系统的设计也非常适合扩展，可以非常容易实现该需求；除了依靠目前通过统一编辑用例来使用该系统外，我们还可以考虑提供一套接口，直接在代码中配置对应的用例，而不需要再在页面中配置。

其次对于用例编辑模块，我们可以添加用户权限的控制，控制一部分用户的读写权利，这样更加适合在公司内部测试开发团队的使用；另一方面虽然目前页面交互方面已经做得比较合理，但是仍然有很多可以继续改进的地方，比如可以定制jsoneditor，添加对本系统中使用到的关键字的补全功能等。

产品之路，任重而道远。为了做出更好的产品，我们会继续努力，争取在下个版本将该系统做得更好。

参考文献

[1] Craig Walls,无,Spring in Action(fourth edition),Greenwich Connecticut:Manning Publications,November 2014,29~56页。

[2] Netty project, http://netty.io/。

[3] NodeJS Home, https://nodejs.org/。

[4] Express—web application framework for Node.js, http://www.expressjs.com.cn/。

[5] Docker - Build, Ship, and Run Any App, Anywhere, https://www.docker.com/。

[6] Jake Spurlock,李松峰,Bootstrap用户手册：设计响应式网站(第一版),北京:人民邮电出版社,2013.9,14~25页。

[7] Less.js: Getting started, http://lesscss.org/。

[8] Github — jadejs/jade, https://github.com/jadejs/jade#jade1。

[9] 陶国荣,无,jQuery权威指南(第一版),北京:机械工业出版社,2011.1,20~21页。

[10]维基百科-Git 词条,http://zh.wikipedia.org/wiki/Git。  
[11] Github — jsoneditor, https://github.com/josdejong/jsoneditor。

致谢

整个HappyMock项目是我在eBay实习期间完成的项目，在结束长达6个月的实习之前，虽然整个开发过程中我需要学习很多新技术，也遇到了很多技术难题，但是最终在实习导师的帮助下，以及凭借自己的专研，都将其逐一解决。

首先在这里，我需要感谢我的毕业设计指导老师—李翔老师。李老师非常认真负责，从毕业设计开题直至论文完成的整个过程，一直关注着我毕业论文的完成情况，大到论文整体篇章结构，小到格式规范，李老师的每一个建议都使我不断完善进步，最终完成了本篇论文。

其次要感谢我在eBay实习的导师Jason和Jack，正是他们为我在技术上指点迷津，我才能不断前进，最终完成了整个项目。从他们身上，我看到了编程技术上差距，这不断激励着我学无止境，技术之路，任重而道远。

最后的话我需要感谢我的朋友和同学，他们陪我走过了大学美好的四年，他们是我不断前进和进步的动力和源泉。

祝福所有关心和帮助过的人都能天天开心，一切安好！

1. 关于SOA的详细信息，参见：<http://en.wikipedia.org/wiki/Service-oriented_architecture> [↑](#footnote-ref-1)
2. 关于REST的详细信息，参见： <http://zh.wikipedia.org/wiki/REST> [↑](#footnote-ref-2)
3. 关于Dropwizard的详细信息，参见：<http://www.dropwizard.io/> [↑](#footnote-ref-3)
4. 关于CXF的详细信息，参见：<https://github.com/apache/cxf> [↑](#footnote-ref-4)
5. 关于HTTPretty的详细信息，参见：<https://github.com/gabrielfalcao/HTTPretty> [↑](#footnote-ref-5)
6. 关于node-mocks-http的详细信息，参见：<https://github.com/howardabrams/node-mocks-http> [↑](#footnote-ref-6)
7. 关于moco的详细信息，参见：<https://github.com/dreamhead/moco> [↑](#footnote-ref-7)
8. 关于DSL的详细信息，参见：<http://en.wikipedia.org/wiki/Domain-specific_language> [↑](#footnote-ref-8)