****



**本 科 毕 业 设 计**

院 系 软件学院

专 业 软件工程

题 目 BugHunter 系统后台的设计与实现

年 级 2014 学 号 141250196

学生姓名 张兴楚

指导教师 房春荣 职 称 讲师

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： BugHunter 系统后台的设计与实现

软件学院 院系 软件工程 专业 14 级本科生姓名： 张兴楚

指导教师（姓名、职称）： 房春荣讲师

摘要：

\*\*项目背景(为什么要做这个项目)。

\*\*技术简介(项目开发中使用了哪些技术，有何特点)。

\*\*项目组整体完成了项目中的哪些功能。

\*\*本人在项目中承担了哪些工作。

关键词：

**南京大学本科生毕业论文（设计）英文摘要**

THESIS：

DEPARTMENT：

SPECIALIZATION:

UNDERGRADUATE:

MENTOR:

ABSTRACT：

KEY WORDS: \*\*项目，\*\*技术(关键名词)1，\*\*技术(关键名词)2，…..

# 目录

[目录 I](#_Toc513492607)

[图目录 III](#_Toc513492608)

[表目录 IV](#_Toc513492609)

[第一章引言 1](#_Toc513492610)

[1.1项目背景 1](#_Toc513492611)

[1.2群智测试工具以及现状 1](#_Toc513492612)

[1.2.1bugtags 1](#_Toc513492613)

[1.2.2班墨云测试 2](#_Toc513492614)

[1.2.3群智测试现状 3](#_Toc513492615)

[1.3项目功能概述 3](#_Toc513492616)

[第二章技术概述 4](#_Toc513492617)

[2.1后台框架Spring Boot 4](#_Toc513492618)

[2.2依赖注入IoC和面向切面编程AOP 5](#_Toc513492619)

[2.3 Spring Boot JPA 6](#_Toc513492620)

[2.4关系型数据库管理系统MySQL 7](#_Toc513492621)

[2.5AppKey和AppSecret 8](#_Toc513492622)

[2.6跨域请求 8](#_Toc513492623)

[第三章系统需求分析与概要设计 10](#_Toc513492624)

[3.1BugHunter项目整体概述 10](#_Toc513492625)

[3.2系统需求分析 10](#_Toc513492626)

[3.2.1用例图 10](#_Toc513492627)

[3.2.2用例描述 11](#_Toc513492628)

[3.2.3非功能需求描述 17](#_Toc513492629)

[3.3系统概要设计 17](#_Toc513492630)

[3.3.1E-R图 17](#_Toc513492631)

[3.3.2 Spring Boot项目bughunter工程结构 18](#_Toc513492632)

[3.3.3开发包图 19](#_Toc513492633)

[3.4本章小结 20](#_Toc513492634)

[第四章BugHunter详细设计与实现 21](#_Toc513492635)

[4.1项目分层概述 21](#_Toc513492636)

[4.1.1Controller层 21](#_Toc513492637)

[4.1.2Service层 21](#_Toc513492638)

[4.1.3Data层 21](#_Toc513492639)

[4.2 User模块详细设计 22](#_Toc513492640)

[4.2.1模块概述 22](#_Toc513492641)

[4.2.2整体结构 22](#_Toc513492642)

[4.2.3接口及代码示例 23](#_Toc513492643)

[4.2.4动态模型 27](#_Toc513492644)

[4. 3App模块详细设计 29](#_Toc513492645)

[4.3.1模块概述 29](#_Toc513492646)

[4.3.2整体结构 29](#_Toc513492647)

[4.3.3接口及代码示例 30](#_Toc513492648)

[4.3.4动态模型 34](#_Toc513492649)

[4.4 Bug模块详细设计 36](#_Toc513492650)

[4.4.1模块概述 36](#_Toc513492651)

[4.4.2整体结构 37](#_Toc513492652)

[4.4.3接口及代码示例 37](#_Toc513492653)

[4.4.4动态模型 43](#_Toc513492654)

[4.5本章小结 45](#_Toc513492655)

[第五章总结与展望 46](#_Toc513492656)

[5.1总结 46](#_Toc513492657)

[5.2展望 46](#_Toc513492658)

[参考文献 47](#_Toc513492659)

[致谢 48](#_Toc513492660)

# 图目录

[图2.1 对象关系映射示例 7](#_Toc513490137)

[图2.2 ORM三个阶段 7](#_Toc513490138)

[图2.3 CORS实现 9](#_Toc513490139)

[图3.1 用例图 11](#_Toc513490140)

[图3.2 系统E-R图 18](#_Toc513490141)

[图3.3工程结构 19](#_Toc513490142)

[图3.4开发包图 20](#_Toc513490143)

[图4.1 User模块设计 23](#_Toc513490144)

[图4.2邮件配置 25](#_Toc513490145)

[图4.3邮件发送代码 25](#_Toc513490146)

[图4.4UserRepository接口 26](#_Toc513490147)

[图4.5数据库配置 26](#_Toc513490148)

[图4.6添加用户顺序图 27](#_Toc513490149)

[图4.7获取所有用户顺序图 28](#_Toc513490150)

[图4.8激活邮件发送顺序图 28](#_Toc513490151)

[图4.9 App模块设计 29](#_Toc513490152)

[图4.10 cont函数的使用 33](#_Toc513490153)

[图4.11复合主键AppMemberKeys 34](#_Toc513490154)

[图4.12主键类的使用 34](#_Toc513490155)

[图4.13添加应用顺序图 35](#_Toc513490156)

[图4.14获取成员顺序图 35](#_Toc513490157)

[图4.15添加成员顺序图 36](#_Toc513490158)

[图4.16 Bug模块设计 37](#_Toc513490159)

[图4.17文件上传的实现 40](#_Toc513490160)

[图4.18服务器资源获取 41](#_Toc513490161)

[图4.19 自定义SQL 42](#_Toc513490162)

[图4.20提交bug顺序图 43](#_Toc513490163)

[图4.21获取图片资源顺序图 43](#_Toc513490164)

[图4.22修改bug顺序图 44](#_Toc513490165)

[图4.23获取统计数据 45](#_Toc513490166)

# 表目录

[表3.1 个人信息管理用例 12](#_Toc513486393)

[表3.2 应用添加用例 12](#_Toc513486394)

[表3.3 应用信息修改用例 13](#_Toc513486395)

[表3.4 应用删除用例 13](#_Toc513486396)

[表3.5 应用参与者添加用例 13](#_Toc513486397)

[表3.6 筛选bug信息用例 14](#_Toc513486398)

[表3.7 修改bug信息用例 14](#_Toc513486399)

[表3.8 提交bug信息用例 15](#_Toc513486400)

[表3.9 查看当前bug信息 15](#_Toc513486401)

[表3.10 应用bug统计查看用例 16](#_Toc513486402)

[表3.11 应用集中管理 16](#_Toc513486403)

[表3.12 用户集中管理 16](#_Toc513486404)

[表3.13 非功能需求 17](#_Toc513486405)

[表3.14 约束 17](#_Toc513486406)

[表4.2 UserController类接口 23](#_Toc513486407)

[表4.3 UserService接口设计 24](#_Toc513486408)

[表4.3 AppController设计 31](#_Toc513486409)

[表4.4 AppService接口设计 32](#_Toc513486410)

[表4.5 BugController设计 39](#_Toc513486411)

[表4.6 BugService接口设计 42](#_Toc513486412)

# 第一章引言

## 1.1项目背景

自2007年，苹果推出了第一代iPhone，智能手机开始真正走向市场。并凭借其人性化，功能强，速度快的特点飞速发展和普及。随之而来的是移动应用出现在我们生活的方方面面，从日常社交，新闻浏览，运动健身到购物付款，移动应用保罗万象，甚至南大宿舍洗衣机也需要U净这款app使用。现代人一机在手，行走天下，越来越离不开手机，离不开里面覆盖生活方法面面的应用。据工信部发布2017年上半年我国互联网业务运行情况报告显示国内移动应用程序市场持续活跃且移动互联网应用数量已超402万款，其中游戏达116万款。

然而在这高速发展的移动应用市场却也存在着诸多隐患，2016年CNCERT/CC捕获及通过厂商交换获得的移动互联网恶意程序样本数量为2,053,501个。而Android平台用户成为最主要的攻击对象。恶意程序针对Android平台的恶意程序共有2,053,450个，占99.9%以上，位居第一。应用安全正成为用户越来越关注的点。另外移动应用的使用场景比传统pc端应用或网页应用更加复杂，同一个移动应用的使用环境包括不同版本的不同操作系统，不同的硬件设备，不同的网络环境。

所以对于开发者而言，确保移动应用在复杂环境中有效运行无疑是一个艰巨的任务。而测试无疑是质量安全的必要保证。但移动应用环境的复杂带来了移动应用测试的复杂，对于开发者而已，保证测试设备完备，测试人员专业必然是困难的事情。面对这样的难题，一个提供大量只有多样手机用户的专业的测试平台则是帮助开发人员的法宝。这就是群智测试工具兴起的必然。

## 1.2群智测试工具以及现状

### 1.2.1bugtags

Bugtags是国内一个精干的移动开发团队为改善移动产品质量而专门打造的测试平台产品。Bugtags采用的是所见即所得的问题上报方式，得益于此有效提高了问题上报的效率和问题描述的准确度；同时Bugtags也提供了自动收集分析崩溃信息与bug生命周期管理功能。能够展示bug描述信息，设备信息，重现步骤，堆栈信息，用户自定义数据，控制台日志，网络请求，十分全面如下图1.1所示。该团队还提供了bugtags日志的专有服务。但如果是免费用户则只能是使用基本功能，得不到细致周到的服务。



图1.1 bugtags的bug信息

### 1.2.2班墨云测试

班墨云测试平台自称为是全球领先的人工智能云测试平台，其除了提供与bugtags类似的众测服务外，还为用户提供兼容性测、适配测试、稳定性测试、网络测试、功能测试、性能测试等服务。自动化测试，人工智能测试时其一大特色。该测试平台也不局限于移动应用，网站，H5，游戏开发者均可使用该平台，改善项目开发。该平台如下图1.2所示：



### 1.2.3群智测试现状

从测试平台现状来看，移动应用群智测试大多借助于平台广阔的用户群体，由用户下载装有平台开发SDK的应用，使用logcat和monkey进行测试，收集log日志和crash信息，用web页面进行展示。其中用户测试时缺乏反馈信息，大量用户重复提交相同bug，一定程度上提升了开发人员的负担。我们希望除了提供群智测试平台外，用户测试时可以实时得到平台反馈，减少bug的重复提交，提升bug描述的精确有效。

## 1.3项目功能概述

本人负责开发群智测试系统bughunter的后台服务器，该服务器应提供一下基本功能：

1.bug信息的提交。

2.bug信息的筛选反馈。

3.应用bug信息的统计分析数据。

4.用户，应用及其版本信息，bug信息的存储使用，主要是增删改查功能，其中需要记录bug的修改历史记录。

1.4论文的主要工作和组织结构

论文主要表述群智安卓测试系统BugHunter后端的设计与实现，还有其与web前端及Android测试端的交互。

本文第一章为论文的引文，第二章是bughunter所用到的技术概述，第三章描述了系统的需求分析和概要设计，第四章则表述了系统的详细设计与实现，第五章作为最后表达了本次毕业设计的总结与展望。

# 第二章技术概述

## 2.1后台框架Spring Boot

从应用方面而言，框架是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;从目的方面可以说框架是可被应用开发者定制的应用骨架。

软件系统发展到今天变得非常复杂，特别是服务器端软件，涉及到的知识，内容，问题极多。而使用别人成熟的框架，框架可以帮助完成一些基础工作，开发人员只需要集中精力完成系统的业务逻辑设计。并且框架一般是成熟，稳健的，可以处理系统很多细节问题，如事物处理、安全性、数据流控制等问题。框架一般都经过广泛使用和验证，结构性好，扩展性强，而且它是不断升级的。

为了便于bughunter系统的后台开发，我们也应用了框架来开发，采用了目前流行的java web框架Spring Boot。Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。相比Spring和SpringMVC需要进行很多手动配置，不仅徒增工作量而且在跨平台部署时容易出问题，Spring-Boot遵循约定大于配置的软件设计范式，使用了特定的方式来进行配置，从而使开发人员从样板化的配置中解放出来。

依据Spring Boot的官网，其主要提供以下核心功能：

1.独立运行的Spring项目，Spring Boot可以以jar包的形式来运行，运行一个Spring Boot项目只需要通过java -jar xx.jar的命令。

2.内嵌Servlet容器，Spring Boot可以内嵌Tomcat，无需以war包的形式部署项目。

3.提供starter简化Maven配置，使用Spring或者SpringMVC需要添加大量的依赖，而这些依赖大都是固定的。Spring Boot 通过starter能够帮助我们简化Maven配置。

4.自动配置Spring。

5.准生产的应用监控。

6.无代码生成和xml配置。

关于使用Spring Boot进行开发的更详细的内容，将在第三章和第四章进行更详细的说明。

## 2.2依赖注入IoC和面向切面编程AOP

Spring的核心特性就是IoC和AOP，IoC（Inversion of Control），即“控制反转”；AOP（Aspect-Oriented Programming），即“面向切面编程”。

IoC也叫DI（Dependency Injection），即依赖注入。并非一种技术实现，而是一种设计思想。在任何一个有实际开发意义的程序项目中，会使用很多类来描述它们特有的功能，并且通过类与类之间的相互协作来完成特定的业务逻辑。这样，每个类都需要负责管理与自己有交互的类的引用和依赖，代码将会变的异常难以维护和极度的高耦合。而IoC正是用来解决这个问题，通过IoC将这些相互依赖对象的创建、协调工作交给Spring容器去处理，每个对象只需要关注其自身的业务逻辑关系。如此，获得依赖的对象的方式，进行了反转，变成了由spring容器控制对象如何获取外部资源。可以说IoC有利于实现松耦合，对象被动接收依赖类而不是自己主动寻找，实现了JNDI的反转：对象不是从容器中查找它的依赖类，而是容器在实例化对象时主动将依赖类注入给它。

AOP（Aspect Oriented Programming），面向切面编程，《Spring3.x企业应用开发实战》第6章写道：AOP是OOP的有益补充。

OOP引入封装、继承、多态等概念建立一种对象层次结构，模拟公共行为的一个集合。但OOP并不适合定义横向的关系，例如日志功能，日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系。还有如安全性、异常处理和透明的持续性，这种散布在各处的无关的代码被称为横切，在OOP设计中导致了大量代码的重复，不利于模块重用。

AOP则不同，利用称为"横切"的技术，剖解开封装的对象内部，将那些影响了多个类的公共行为封装至一个可重用模块，命名为"Aspect"，即切面，减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。使用"横切"技术，AOP将软件系统分为两个部分：核心关注点和横切关注点。业务处理主要流程是核心关注点，与之关系不大的、经常发生在核心关注点的多处的部分则是横切关注点。AOP的作用就在于此，分离系统中的各种关注点，将核心关注点和横切关注点分离开来。将业务逻辑从系统服务中分离出来，有助于内聚开发，将服务模块化，并把它们声明式地应用在需要它们的地方，使这些组件更加专注于自身的业务，不需要涉及其它系统问题。

## 2.3 Spring Boot JPA

JPA是Java Persistence API的简称，中文名Java持久层API，是JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中。JPA为java开发者提供了对象-关系的映射机制，用于管理java应用中的关系数据。如下图2.1就是对象关系映射的一个例子。

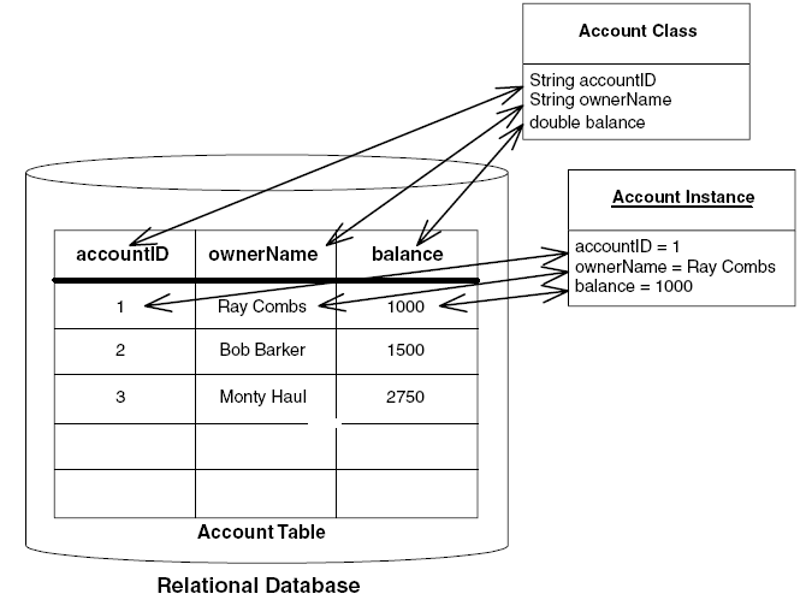


图2.1 对象关系映射示例

对象关系映射ORM通过编程的方式将对象类型转换成关系类型。主要特点是将object映射成数据库中的数据。在映射的过程中必须考虑数据，数据类型和数据之间的关系。ORM在三个阶段中将数据存入数据库中如下图2.2所示。

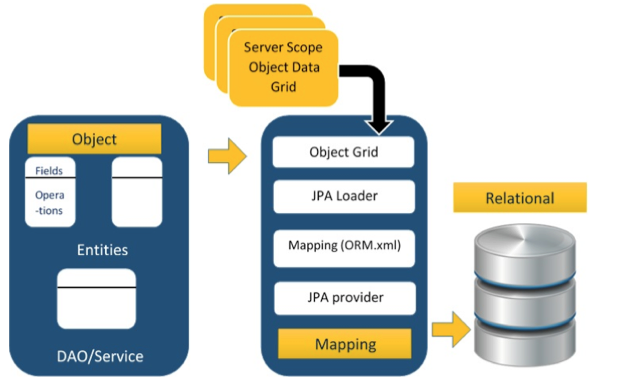


图2.2 ORM三个阶段

而在本次开发中，由于使用了Spring Boot的框架，使用JPA极其简单方便，只需要配置maven依赖，配置文件，创建实体以及数据访问接口即可。可以说本次开发得益于JPA的简单方便，查询能力，数据层开发十分快捷简练。

## 2.4关系型数据库管理系统MySQL

RDBMS，即传统的关系型数据库管理系统。典型的代表有MySQL、Oracle、SQL Server以及PostgreSQL等。

关系型数据库一般满足事务型数据库的要求ACID，即：Atomicity（原子性），Consistency（一致性），Isolation（隔离性），Durability（持久性）。除此之外，RDBMS的表设计还有几大设计范式，1NF、2NF、3NF、BCNF。

这些原则与范式强调数据必须在任何时刻都是完整的、可靠的、一致的。这种模型是很直观的、符合人们对于数据预期的一种模型，也是对于安全性来说最佳的数据模型。

考虑到关系型数据库存在锁（如MySQL存储引擎MyISAM的表锁、InnoDB的行锁），在高并发量下，锁竞争会异常激烈，导致响应时间下降的问题，还有关系型数据库对bughunter的优势并不明显，本次bughunter系统后台的开发中，思考过使用与JSON完美结合的MongoDB，或者当下很流行的Spring Boot和Key-Value数据库Redis组合，但鉴于时间紧迫，开发人员的知识储备等情况，仍选用开源关系型数据库MySQL作为数据持久化存储的解决方案。

MySQL是最流行的关系型数据库管理系统之一，在WEB应用方面，MySQL是最好的RDBMS应用软件。与其他的大型数据库例如Oracle、SQL Server等相比，MySQL有它的不足之处，但丝毫没有减少它受欢迎的程度。对于一般的个人使用者和中小型企业来说，MySQ提供的功能绰绰有余，而且MySQL是开源软件，可以大大降低成本。当下一般中小型网站的开发都选择 MySQL 作为网站数据库。

MySQL在技术社区中有大量支持，而且在课程的学习实践中，MySQL也是我们选择最多的数据层的解决方案。相信采用MySQL，能够更快更好地完成本次bughunter的后台的开发工作。

## 2.5AppKey和AppSecret

目前开放api接口平台会有AppId、AppKey、AppSecret，各有用途：

1. AppId是用来标记开发者账号的, 是用户id。

2. AppKey客户端调用API时的唯一标识，服务器通过App Key来鉴别应用的身份。调用API接口时必须传入的参数。

3.AppSecret是服务端给客户端分配的密钥，用来保证应用来源的可靠性，防止请求数据被伪造。

当下也有简化的场景如省去AppId, 默认每一个用户有且仅有一套权限配置， 所以直接将AppId= AppKey。或者省去AppId和AppKey，通常用于开放性接口的地方。

本次开发bughunter后台仅供web前端和Android端调用，所以为简便，只采用了AppKey和AppSecret，主要用于Android端请求，唯一标识App。

## 2.6跨域请求

本次开发集成测试时，发现由于前后端域名不一致，存在跨域请求问题，遂采用CORS（Cross-origin resource sharing）即跨域资源机制。CORS是一个W3C标准，允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

CORS需要浏览器和服务器同时支持。目前所有浏览器基本都支持该功能，IE浏览器则不能低于IE10。

CORS通信过程都是浏览器自动完成，不需用户参与。对于开发者来说，CORS通信与同源的AJAX通信没有区别，代码一致。浏览器发现AJAX请求跨源，就会自动添加附加的头信息，有时还会多发出一次附加请求，但用户不会有所察觉。

实现CORS通信的关键在于服务器。只要服务器实现了CORS接口，就可以跨源通信了。本次于服务其中采用Servlet中的过滤器Filter来实现CORS机制。

过滤器Filter可以实现如下功能：

1.在HttpServletRequest 到达Servlet之前，拦截HttpServletRequest。根据需要检查HttpServletRequest，也可以修改HttpServletRequest头和数据。

2.在HttpServletResponse 到达客户端之前，拦截HttpServletResponse。根据需要检查HttpServletResponse，可以修改HttpServletResponse 头和数据。

而CORS实现正是利用了功能1，本次实现如下图2.3所示：



图2.3 CORS实现

# 第三章系统需求分析与概要设计

## 3.1BugHunter项目整体概述

群智安卓测试BugHunter系统是一个面向安卓测试的移动应用群智测试平台，它的主要目标是为安卓测试经理提供bug生命周期的管理功能，为测试人员提交bug高效准确的上报功能。

BugHunter系统通过web前端安卓测试经理不仅提供app和bug的管理功能，还提供了丰富的bug信息统计图表的展示，更便于分析app的开发情况。

通过开发安卓测试SDK，应用开发人员可以方便的将我们提供的静态库添加到其app中添加bug的反馈和提交功能，之后测试人员可以很方便的使用我们提供的悬浮按钮查看当前页面存在的bug和提交当前页面bug，当app崩溃时将自动上传bug信息。

本人则负责用spring-boot+mysql为基础开发server端，为app提供bug上报途径，为web端提供app和bug的信息筛选和管理功能。

## 3.2系统需求分析

### 3.2.1用例图

为了更形象的阐述需求，将系统的功能以用例图和用例描述的方式展示如下图3.1所示：

用例图由角色，系统边界和用例组成，它可以帮助我们更好地描述系统内外交互。系统的涉众有测试人员和测试经理两类。测试人员借助系统完成测试任务上报bug。测试经理则研究系统收集到的数据，分析应用开发状况。

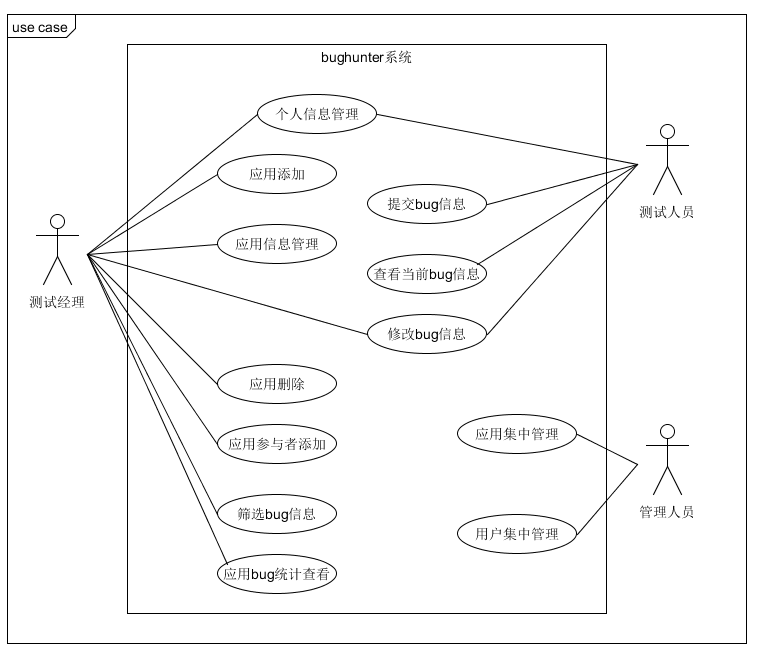


图3.1 用例图

### 3.2.2用例描述

基于以上用例图编写用例描述如下列表所示：

1）个人信息管理

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc01 |
| 名称 | 个人信息管理 |
| 参与者 | 用户 |
| 触发条件 | 用户需求管理个人信息 |
| 前置条件 | 用户已登陆成功 |
| 后置条件 | 用户信息按用户的需要做出修改 |
| 正常流程 | 1.用户进入个人信息页面  2.系统返回用户个人信息  3.用户修改个人信息，并点击确定  4.系统将用户个人信息修改情况反馈用户 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.1 个人信息管理用例

2）应用添加

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc02 |
| 名称 | 应用添加 |
| 参与者 | 测试经理 |
| 触发条件 | 测试经理添加应用到账户 |
| 前置条件 | 测试经理已经登陆成功 |
| 后置条件 | 系统将应用信息添加到数据库 |
| 正常流程 | 1.测试经理进入应用添加页面，并且输入完整应用信息，点击添加  2.系统自动生成AppKey和AppSecret，将应用信息添加到数据库，返回AppKey和AppSecret |
| 扩展流程 | 无 |
| 特殊需求 | 1.AppKey应为app主键，确保唯一性  2.AppSecret为app验证信息，需难以破解，确保安全性 |

表3.2 应用添加用例

3）应用信息管理

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc03 |
| 名称 | 应用信息管理 |
| 参与者 | 测试经理 |
| 触发条件 | 应用信息需要被查看或修改 |
| 前置条件 | 测试经理已经登陆成功 |
| 后置条件 | 系统将应用信息修改提交到数据库 |
| 正常流程 | 1.测试经理进入应用信息页面  2.系统返回应用信息  3.测试经理修改应用信息，点击确定  4.系统按需要修改应用信息，返回修改结果 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.3 应用信息修改用例

4）应用删除

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc04 |
| 名称 | 应用删除 |
| 参与者 | 测试经理 |
| 触发条件 | 应用已经被放弃，需要删除 |
| 前置条件 | 测试经理已经登陆 |
| 后置条件 | 应用被从数据库删除，包括应用所含bug |
| 正常流程 | 1.测试经理进入应用页面，点击删除  2.系统删除数据库中包含的应用及其bug信息，返回结果 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.4 应用删除用例

5）应用参与者添加

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc05 |
| 名称 | 应用参与者添加 |
| 参与者 | 测试经理 |
| 触发条件 | 有人员需要被添加应用管理权限 |
| 前置条件 | 该应用测试经理已经登陆 |
| 后置条件 | 选择的用户被添加到应用中 |
| 正常流程 | 1.测试经理进入应用成员添加页面，输入添加人员邮箱，成员类型，点击确定  2.系统将该成员添加至应用，返回结果 |
| 扩展流程 | 2a.该成员邮箱不存在  2a1.系统返回该成员不存在  2b.该测试经理无添加成员权限  2b1.系统返回权限不足信息 |
| 特殊需求 | 1.仅应用创建者有权限添加成员 |

表3.5 应用参与者添加用例

6）筛选bug信息

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc06 |
| 名称 | 筛选bug信息 |
| 参与者 | 测试经理 |
| 触发条件 | 测试经理筛选bug信息，分析应用开发状况 |
| 前置条件 | 测试经理已经登陆 |
| 后置条件 | 无 |
| 正常流程 | 1.测试经理点击进入bug展示页面  2.系统返回该应用下所有bug用于展示  3.测试经理点选筛选条件  4.系统根据筛选条件返回所需bug信息 |
| 扩展流程 | 无 |
| 特殊需求 | 1.筛选条件为app版本和bug类型，状态，优先级 |

表3.6 筛选bug信息用例

7）修改bug信息

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc07 |
| 名称 | 修改bug信息 |
| 参与者 | 测试经理或测试人员 |
| 触发条件 | bug信息不够完善，或bug状态改变需要修改信息 |
| 前置条件 | 用户已登录 |
| 后置条件 | 系统按需要修改bug信息并记录修改历史 |
| 正常流程 | 1.用户已进入bug信息页面，做出修改，点击确定  2.系统按需要修改bug信息并记录修改历史后返回结果 |
| 扩展流程 | 无 |
| 特殊需求 | 1.bug信息用户仅可以修改描述、状态、优先级、类型 |

表3.7 修改bug信息用例

8）提交bug信息

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc08 |
| 名称 | 提价bug信息 |
| 参与者 | 测试人员 |
| 触发条件 | 测试人员发现bug或应用崩溃 |
| 前置条件 | 测试人员登陆成功或测试人员仅使用而未登陆均可 |
| 后置条件 | 系统将测试人员提交的bug保存到数据库 |
| 正常流程 | 1.测试人员发现bug并且填写提交信息完毕后点击确定，或应用在使用中崩溃自动提交bug。  2.系统记录提交的bug到数据库，返回结果 |
| 扩展流程 | 2a.该系统的版本未被记录  2a1.系统记录该系统版本 |
| 特殊需求 | 1.用户仅可填写bug描述、状态、优先级、类型，其他信息由SDK自动提取 |

表3.8 提交bug信息用例

9）查看当前bug信息

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc09 |
| 名称 | 查看当前bug信息 |
| 参与者 | 测试人员 |
| 触发条件 | 测试人员进入app行的activity |
| 前置条件 | 无 |
| 后置条件 | 无 |
| 正常流程 | 1.测试人员点击应用使应用进入新的页面  2.系统返回当前页面已存在的bug信息 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.9 查看当前bug信息

10）应用bug统计查看

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc10 |
| 名称 | 应用bug统计查看 |
| 参与者 | 测试经理 |
| 触发条件 | 测试经理进行bug统计查看 |
| 前置条件 | 该应用测试经理已经登陆 |
| 后置条件 | 无 |
| 正常流程 | 1.测试经理点击进入bug统计信息展示页面  2.系统返回当前最新版本的bug统计信息  3.测试经理点选不同版本查看  4.系统返回不同版本bug统计信息 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.10 应用bug统计查看用例

11）应用集中管理

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc10 |
| 名称 | 应用集中管理 |
| 参与者 | 测试经理 |
| 触发条件 | 测试经理进行bug统计查看 |
| 前置条件 | 该应用测试经理已经登陆 |
| 后置条件 | 无 |
| 正常流程 | 1.测试人员点击应用使应用进入新的页面  2.系统返回当前页面已存在的bug信息 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.11 应用集中管理

12）用户集中管理

|  |  |
| --- | --- |
| 项目 | 内容描述 |
| ID | uc12 |
| 名称 | 用户集中管理 |
| 参与者 | 测试经理 |
| 触发条件 | 测试经理进行bug统计查看 |
| 前置条件 | 该应用测试经理已经登陆 |
| 后置条件 | 无 |
| 正常流程 | 1.测试人员点击应用使应用进入新的页面  2.系统返回当前页面已存在的bug信息 |
| 扩展流程 | 无 |
| 特殊需求 | 无 |

表3.12 用户集中管理

### 3.2.3非功能需求描述

系统的非功能需求如表3.10，要求系统具有良好的可靠性、易用性、可维护性和可移植性，同时，系统要求在较短时间内对用户操作做出响应。

|  |  |
| --- | --- |
| 需求类别 | 需求描述 |
| 效率 | 对于所有操作，系统应及时作出响应，响应时间应在0.5秒之内 |
| 可维护性 | 系统设计应尽可能合理，容易维护和扩展新模块 |
| 安全 | 系统需要验证操作是否合法，并拒绝非法操作 |

表3.13 非功能需求

系统所受的约束见表3.11，系统需要在有网络的环境下运行。

|  |  |
| --- | --- |
| 约束类别 | 约束描述 |
| 开发约束 | 系统服务器要求使用spring-boot+mysql的方式开发 |
| 部署环境 | 该系统服务器端将被部署到阿里云服务器ECS上，配置为1核，2GB，系统使用CentOS 7.3 64位 |
| 网络约束 | 系统必须在有网络环境下运行 |
| 用户约束 | 测试人员可不经过登陆使用系统，但须提供AppKey和APPSecret |

表3.14 约束

## 3.3系统概要设计

### 3.3.1E-R图

根据需求给出bughunter系统E-R图如下图3.2所示：

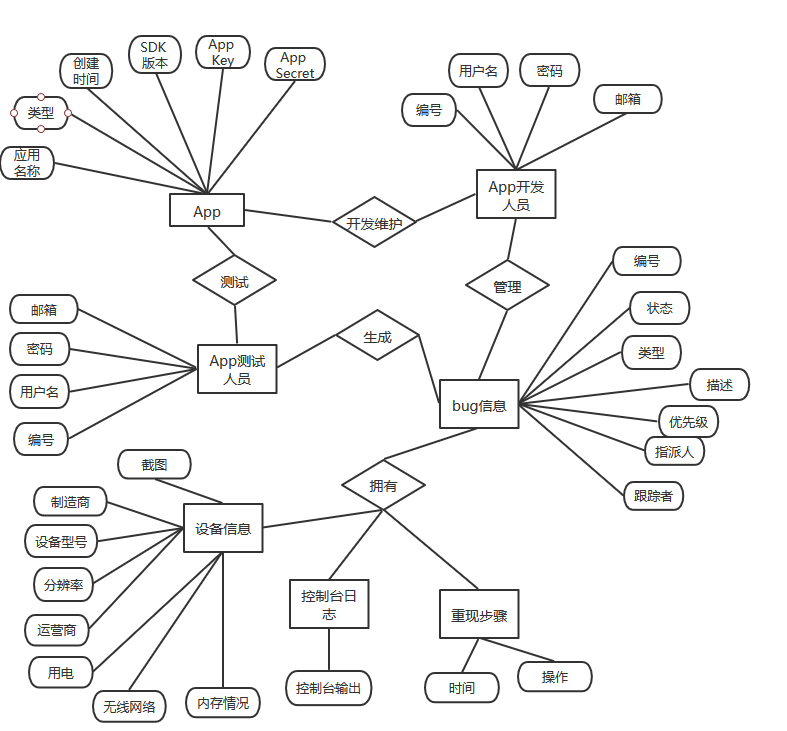


图3.2 系统E-R图

### 3.3.2 Spring Boot项目bughunter工程结构

代码层结构: 根目录bughunter.bughunterserver

1.启动类(BughunterserverApplication.java)位于bughunter.bughunterserver根目录下

2.实体类(entity)位于bughunter.bughunterserver. model.entity

3.数据访问层(Dao)位于bughunter.bughunterserver. model.repository

4.数据服务层(Service)位于bughunter.bughunterserver..service,数据服务的实现接口(serviceImpl)至于bughunter.bughunterserver.service.impl

5.前端控制器(Controller)位于bughunter.bughunterserver.controller

6.工具类(utils)位于bughunter.bughunterserver.util

7.数据传输类(vo)位于bughunter.bughunterserver.vo

资源文件的结构

1.配置文件(application.properties)位于BugHunterServer\src\main\resources文件夹下

2. maven配置文件pom.xml位于BugHunterServer根目录下

3.保存截图位于/home/bughunter/screenshot文件夹下

如下图所示：

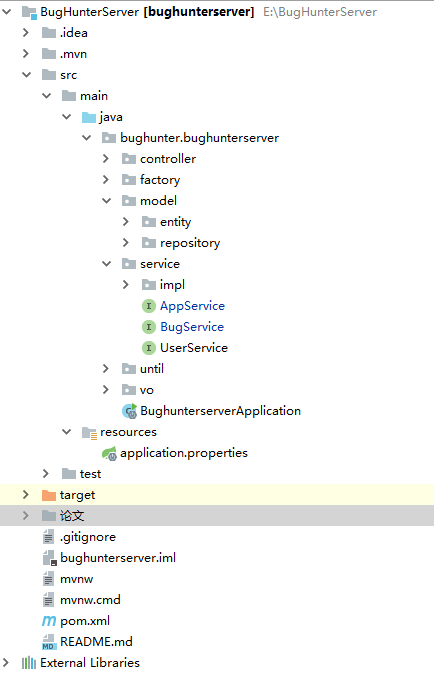


图3.3工程结构

### 3.3.3开发包图

更进一步，给出bughunter后台系统开发中的开发包图如下图所示：

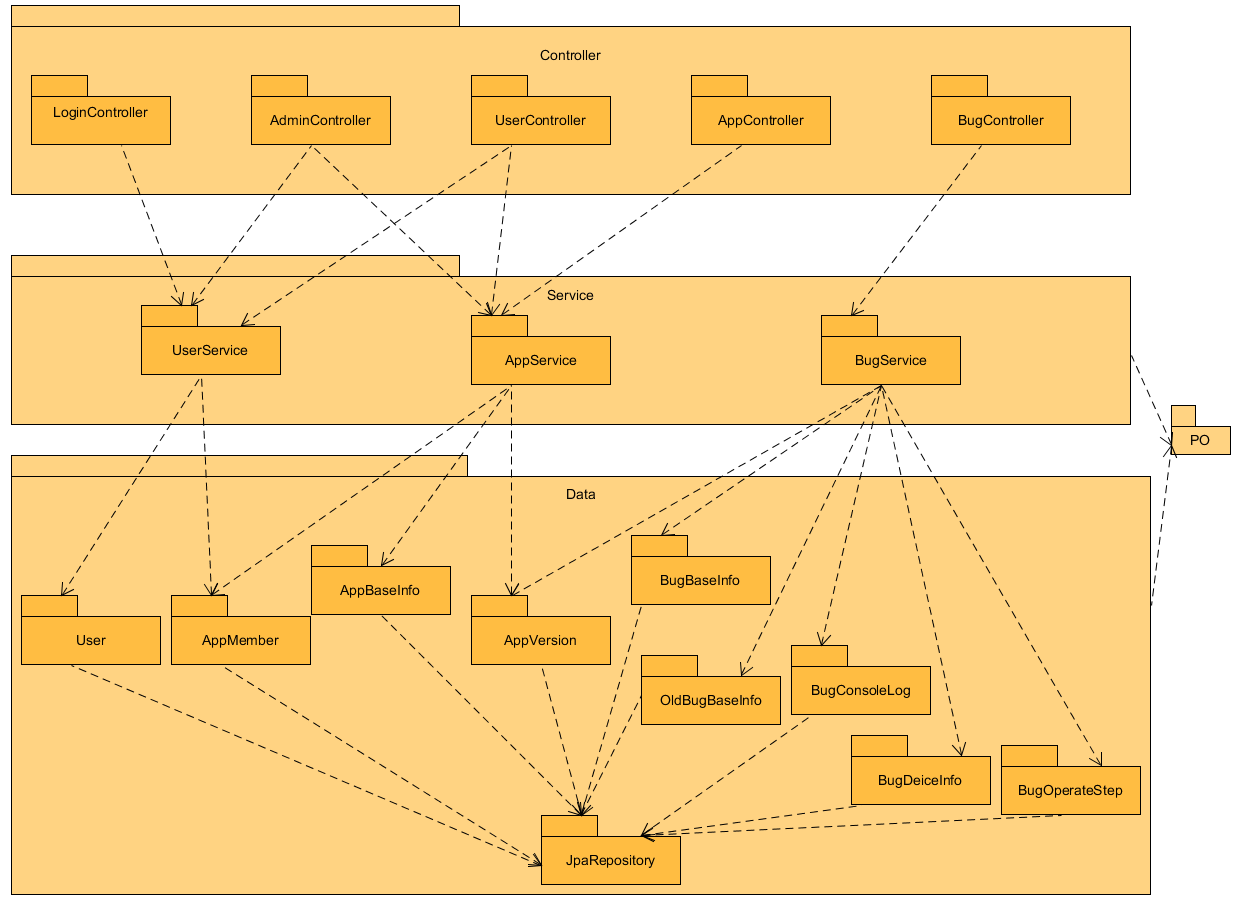


图3.4开发包图

## 3.4本章小结

本章主要介绍项目的整体概述，需求分析和概要设计，通过用例图，E-R图，项目工程架构图等介绍了项目的整体实现。也描述了项目的非功能需求。下章更详细的介绍bughunter各个子模块做准备。

# 第四章BugHunter详细设计与实现

## 4.1项目分层概述

本项目以Spring Boot框架为基础开发，主要分为Controller、Service、Data三层，User、App、Bug三个模块。下面先分别对这三层进行简要描述。各模块的描述及详细设计将在4.2等做说明。

### 4.1.1Controller层

Controller层主要用来映射url路径及接收HTTP请求，返回响应。当请求被发送至服务器后，根据路径将被映射到Controller层响应对象的响应方法中，由于我们采用JSON格式传输数据，将根据需要解析请求体中的JSON数据，后调用Service层的方法以获得请求的对象，之后将数据打包返回响应。不合法的请求将被抛弃。Controller层仅涉及路径映射和返回响应，不涉及data层及其他复杂的业务逻辑。

### 4.1.2Service层

Service层介于Controller层和data层之间，为Controller层提供所需要的服务，从data层获取需要的数据，有时需要对数据进行再加工处理，之后打包为Controller层需要的格式，返回Controller层调用方。Service层应该负责业务逻辑的处理，实际开发中，项目业务逻辑较少，使Service层更多的成为Controller层和data层之间的中介，降低系统的耦合性。

### 4.1.3Data层

Data层是bughunter项目后台开发的重中之重，Data层主要利用JPA为java开发者提供的对象-关系映射机制来管理关系数据。开发主要包括数据库配置，定义实体类，实现DAO层。其中DAO层主要利用了JpaRepository，之后会做详细说明。Data层为Service提供管理数据库数据的功能，帮助Service完成自身所提供的服务。

## 4.2 User模块详细设计

### 4.2.1模块概述

User模块主要负责提供用户信息管理服务，为用户查看、修改个人信息。也为管理员提供删除用户服务，各项服务都通过提供API，处理响应HTTP请求的方式进行。

### 4.2.2整体结构

采用分层风格，将系统分为controller层，service层，data层。每一层之间为了灵活性，添加了接口，以实现针对接口编程，隔离数据传输的职责，降低层与层之间耦合，添加了UserService接口。User模块的设计如图4.1所示：

UserController为HTTP请求映射提供响应。

User是做为用户管理的实体，数据库持久化对象被添加到设计模型中的。UserVO作为返回数据而存在。

UserRepository则作为DAO服务。

ResultMessageFactory为产生符合设计响应格式的返回信息的工厂类。

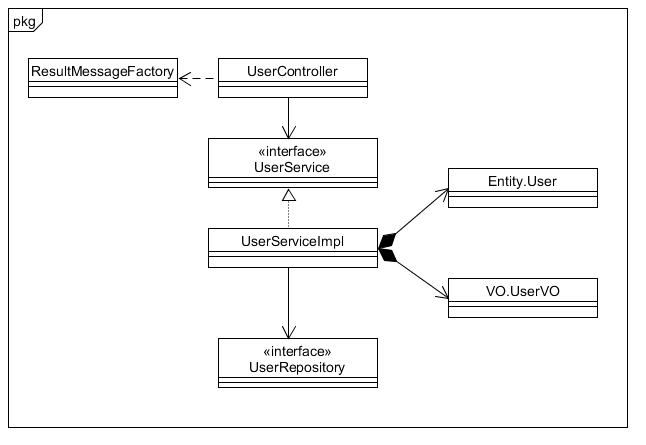


图4.1 User模块设计

### 4.2.3接口及代码示例

1.UserController接口设计

如下表所示：

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | @RequestMapping(value = "/add", method = RequestMethod.POST)  public @ResponseBody ResultMessage addUser(  @RequestBody String jsonStr)； |
| 说明 | 增加用户 |
| 语法 | @RequestMapping(value = "/{email}/sendActiveEmail",  method = RequestMethod.POST)  public @ResponseBody ResultMessage sendActiveEmail(  @PathVariable String email)； |
| 说明 | 发送激活邮件 |
| 语法 | @RequestMapping(value = "/{id}/modify", method = RequestMethod.POST)  public @ResponseBody ResultMessage modifyUser(  @PathVariable int id, @RequestBody String jsonStr)； |
| 说明 | 修改用户信息 |
| 语法 | @RequestMapping(value = "/{id}/delete", method = RequestMethod.POST)  public @ResponseBody ResultMessage deleteUser(@PathVariable int id)； |
| 说明 | 删除用户，仅管理员可用 |
| 语法 | @RequestMapping(value = "/{id}/get", method = RequestMethod.GET)  public @ResponseBody ResultMessage getUser(@PathVariable int id)； |
| 说明 | 获得用户信息 |

表4.2 UserController类接口

说明（1）：注解

由于采用了Spring Boot框架开发，而其的最大优势就是用注解取代繁琐的配置，所以只需要很简单的一些注解就可以很方便的将HTTP请求映射到具体方法。下面简要介绍一下这些注解。

@Controller：表明类为控制器，接受请求并处理。

@RequestMapping：处理地址映射的注解，可用于类、方法。

@PathVariable：即表明参数为获取的HTTP请求路径中的参数。

@RequestBody：即请求体映射到参数。

@ResponseBody：表明方法返回结果直接写入HTTP响应正文。

@Autowired：依赖注入，Spring会自动上下文找到合适的bean来注入。

2. UserService类接口设计

如下表所示：

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | UserVO testLogin(String email, String pwd); |
| 说明 | 用户登录测试，返回包括用户Id和用户类型 |
| 语法 | Boolean deleteUser(int id); |
| 说明 | 删除用户 |
| 语法 | int addUser(User user); |
| 说明 | 增加用户，返回Id |
| 语法 | Boolean modifyUser(int id, JSONObject jsonObject); |
| 说明 | 修改用户信息 |
| 语法 | UserVO findUser(int id); |
| 说明 | 根据Id查找用户 |
| 语法 | List<UserVO> findAllUsers(); |
| 说明 | 返回所有用户信息 |
| 语法 | String sendActiveEmail(String email); |
| 说明 | 发送激活邮件 |
| 语法 | UserVO findByEmail(String email); |
| 说明 | 根据email查找用户信息 |

表4.3 UserService接口设计

UserService的实现十分简单，基本依靠调用Data层方法作简要逻辑处理即可。其中发送邮件较为特殊，下面就在Spring Boot框架中实现邮件发送，及验证码生成做简要说明。

Spring Boot中实现邮件发送，首先在application.properties配置文件中添加如下配置信息，使用的是网易163邮箱。值得一提的是所配置的邮箱必须授权登录第三方邮件客户端。password字段为授权码，而不是登陆用的密码。

|  |
| --- |
| #设置邮箱主机  spring.mail.host=smtp.163.com  #设置用户名  spring.mail.username=\*\*\*\*\*\*\*\*\*\*\*\*@163.com  #设置密码  spring.mail.password=\*\*\*\*\*\*\*\*\*\*\*\*  #设置端口  spring.mail.port=465  #设置需要认证  spring.mail.properties.mail.smtp.auth=true  #将纯文本连接升级为加密连接（SSL）。  spring.mail.properties.mail.smtp.ssl.enable=true  spring.mail.properties.mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory  #超时设置  spring.mail.properties.mail.smtp.timeout=2500 |

图4.2邮件配置

完成配置后，就可以使用依赖注入，得到配置好的JavaMailSender对象，并且用来发送邮件，如下图4.5就是一个简单例子，其中RandonNumberUtils为编写的验证码生成类，getRandonString为静态方法，参数是验证码的位数。该方法主要使用随机数生成，再转化为0-9，a-z，A-Z的方法生成验证码。

|  |
| --- |
| @Autowired  private JavaMailSender mailSender;  public String sendActiveEmail(String email) {  SimpleMailMessage message = new SimpleMailMessage();  //设置发送者  message.setFrom(Constants.SEND\_EMAIL\_FROM);.  //设置接收者  message.setTo(email);  //设置邮件主题.  message.setSubject("BugHunter验证码（邮件主题）");  //获取6位验证码  String vc = RandonNumberUtils.getRandonString(6);  //邮件内容  message.setText("您的验证码：" + vc);  //发送邮件  mailSender.send(message);  return vc;  } |

图4.3邮件发送代码

3.UserRepository接口和数据库配置

说明（1）：JpaRepository的使用

UserRepository接口如下图所示：

|  |
| --- |
| @Repository  public interface UserRepository extends JpaRepository<User, Integer>{  //根据邮件查找用户  User findFirstUserByEmail(String email);  } |

图4.4UserRepository接口

Repository接口为Spring Data一个核心接口，提供最基本的数据访问功能，不提供任何方法，开发者需在自定义的接口中声明需要的方法。

而JpaRepository子接口则扩展了一些功能，实现了一组 JPA 规范相关方法。实际开发中数据访问采用了JpaRepository。其中需要标注实体类，及主键对象类型如上图的User，Integer。原因是其支持接口规范方法名查询。如上图findFirstUserByEmail方法。在接口中定义的查询方法只要符合它的命名规则，就可以不用写实现，方便了开发。JpaRepository支持的关键字十分广泛，如：find，delete，like，In，Contain，And，Or等，基本可以覆盖单表CRUD操作。本次系统实现中更复杂的使用将在接下来的模块中说明。

说明（2）：配置数据库连接信息

Spring Boot中只需要在application.properties配置文件中添加如下图所示配置信息即可。

|  |
| --- |
| spring.jpa.hibernate.ddl-auto=update  #spring.datasource.url=jdbc:mysql://101.132.181.10:3306/bughunter  spring.datasource.url=jdbc:mysql://localhost:3306/bughunter  spring.datasource.username=root  spring.datasource.password=mysql  spring.datasource.driver-class-name=com.mysql.jdbc.Driver |

图4.5数据库配置

图中spring.jpa.hibernate.ddl-auto有多种值可供选择，开发中只是用了create和update两种属性。属性值为create，每次加载Hibernate会删除数据库里的表，重新根据实体类生成新表。为update则知乎根据实体类更新表结构，且不会删除表内存在的数据。

### 4.2.4动态模型

以下是一些bughunter后台系统的运行动态模型，这里只列举典型的几个例子。

1.添加用户顺序图

图 4.8表明了bughunter后台系统中，当用户注册完成，增加用户信息之后，用户业务逻辑处理的相关对象之间的协作。逻辑十分清晰。关于用户的CRUD操作基本类似。

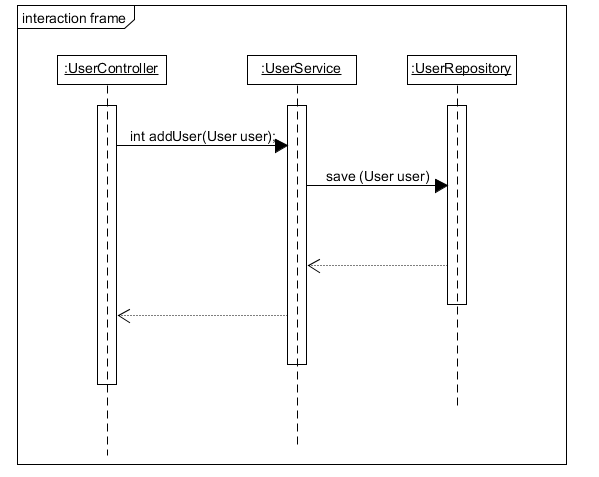


图4.6添加用户顺序图

2.获取用户顺序图

如下图4.9所示为后台获得所有用户信息，相关对象之间的协作。可以看到返回作为相应信息的是UserVO，而不是查询得到的实体直接返回。

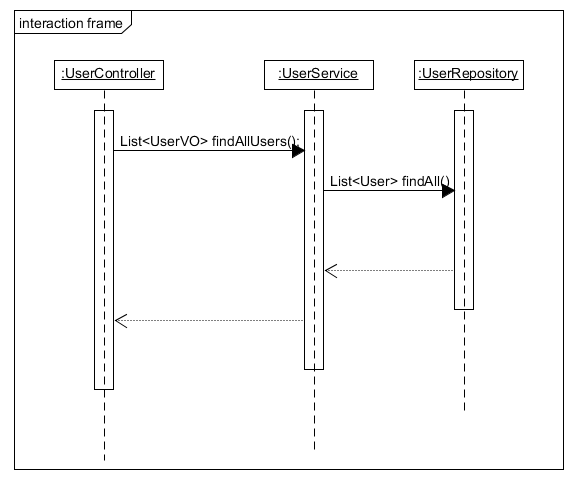


图4.7获取所有用户顺序图

3.激活邮件发送顺序图

如下图4.10所示为后台发送激活邮件，相关对象之间的协作。需要生成验证码，再发送邮件。

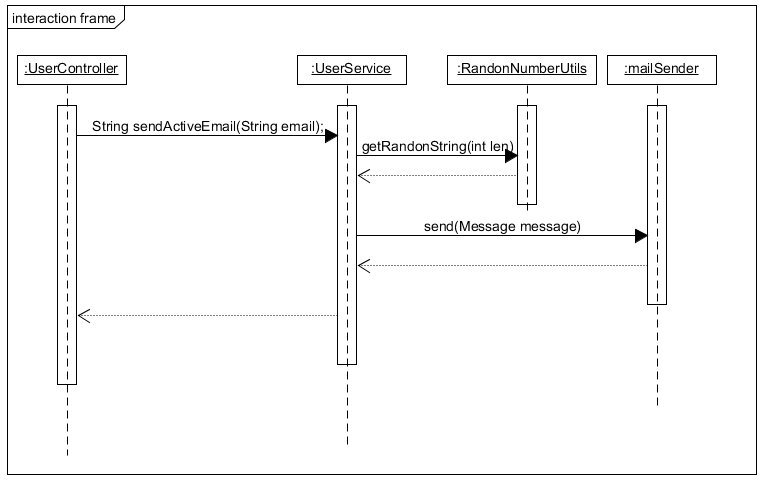


图4.8激活邮件发送顺序图

## 4. 3App模块详细设计

### 4.3.1模块概述

App模块主要负责为应用相关提供服务，提供应用信息、版本管理，应用成员管理服务，也为管理员提供全应用查看服务。各项服务与User模块一样都通过提供API，处理响应HTTP请求的方式完成任务。

### 4.3.2整体结构

类似的，采用分层风格，分为controller层，service层，data层。每一层之间为了灵活性，添加了如AppService的接口，以实现针对接口编程，隔离数据传输的职责，降低层与层之间耦合。不同于User模块关系简单，App模块涉及操作多个实体，所以DAO较多，App模块的设计如下图所示：

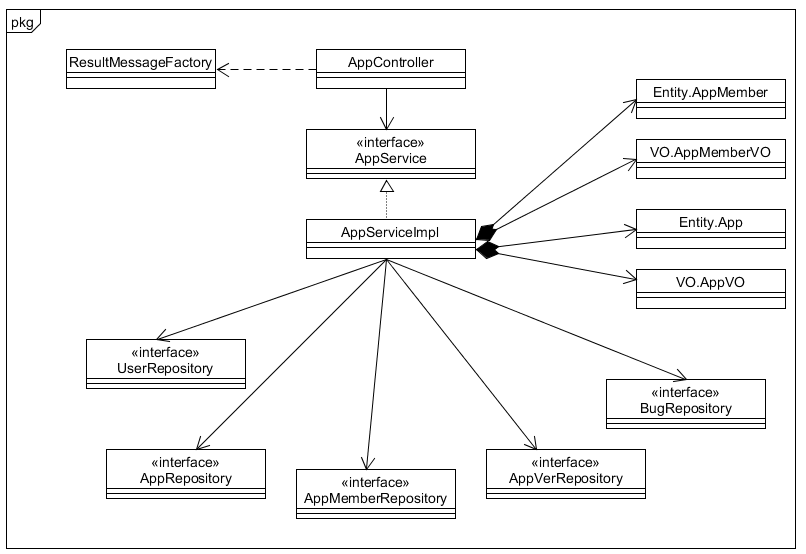


图4.9 App模块设计

与User模块相似Controller提供HTTP路径映射，返回响应，Entity为关系实体，VO为用于返回的打包信息，Repository作为DAO提供数据库操作服务。

### 4.3.3接口及代码示例

1.AppController类接口

如下表所示：

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | @RequestMapping(value = "/add", method = RequestMethod.POST)  public @ResponseBody ResultMessage addApp(  @RequestBody String jsonStr)； |
| 说明 | 添加APP应用 |
| 语法 | @RequestMapping(value = "/{appKey}/modify",  method = RequestMethod.POST)  public @ResponseBody ResultMessage modifyApp(  @PathVariable String appKey, @RequestBody String jsonStr)； |
| 说明 | 修改给定appKey指向的app的信息 |
| 语法 | @RequestMapping(value = "/{appKey}/delete",  method = RequestMethod.POST)  public @ResponseBody ResultMessage deleteApp(  @PathVariable String appKey)； |
| 说明 | 在数据库中删除给定的appKey指向的app信息 |
| 语法 | @RequestMapping(value = "/{appKey}/get", method = RequestMethod.GET)  public @ResponseBody ResultMessage getApp(  @PathVariable String appKey)； |
| 说明 | 得到指定的app信息 |
| 语法 | @RequestMapping(value = "/{appKey}/getMembers",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getMembers(  @PathVariable String appKey)； |
| 说明 | 得到指定app的成员信息 |
| 语法 | @RequestMapping(value = "/addMember", method = RequestMethod.POST)  public @ResponseBody ResultMessage addMember(  @RequestBody String jsonStr)； |
| 说明 | 为指定app添加用户 |
| 语法 | @RequestMapping(value = "/{appKey}/{uId}/deleteMember",  method = RequestMethod.GET)  public @ResponseBody ResultMessage deleteMember(  @PathVariable String appKey, @PathVariable int uId)； |
| 说明 | 删除appKey指向的app的uId指定成员 |
| 语法 | @RequestMapping(value = "/{developerId}/getAll",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getAllApps(  @PathVariable int developerId)； |
| 说明 | 得到给定用户参与的所有app信息 |
| 语法 | @RequestMapping(value = "/{appKey}/getVersions",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getVersions(  @PathVariable String appKey)； |
| 说明 | 得到appKey所指app的所有版本信息 |

表4.3 AppController设计

相关的说明在User模块已经列出，在此不再重复。

2.AppService接口设计

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | Boolean deleteApp(String appKey); |
| 说明 | 删除给定appkey的app |
| 语法 | String addApp(AppBaseInfo appBaseInfo, int uId); |
| 说明 | 添加给定信息的app，创建者的id为uId |
| 语法 | Boolean modifyApp(String appKey, JSONObject jsonObject); |
| 说明 | 修改app信息，修改信息在jsonObject中 |
| 语法 | AppBaseInfoVO findApp(String appKey); |
| 说明 | 获取appKey指向的app信息 |
| 语法 | List<AppBaseInfoVO> findAllAppsByUserId(int uId); |
| 说明 | 获取uId用户参与的所有app列表 |
| 语法 | List<AppBaseInfoVO> findAllApps(); |
| 说明 | 获取所有app，为管理员所调用 |
| 语法 | Boolean deleteMember(String appkey, int uId); |
| 说明 | 删除给定app的给定成员 |
| 语法 | Boolean addMember(AppMember appMember); |
| 说明 | 为app增加成员 |
| 语法 | List<String> findAppVersionByAppKey(String appKey); |
| 说明 | 获取app的所有版本信息 |
| 语法 | List<AppMemberVO> findAllMemberByAppKey(String appKey); |
| 说明 | 获取app的所有成员信息 |

表4.4 AppService接口设计

如上表所示，为AppService接口设计。

说明（1）：UUID

当添加app时，系统会自动生成appKey和appSecret，作为Android端提交bug的应用账号密码信息。为保证复杂性和单一性，采用了UUID来方便快捷的生成32位的appKey、appSecret。

UUID，通用唯一识别码，它保证对在同一时空中的所有机器均是唯一的。用到了以太网卡地址、纳秒级时间、芯片ID码和许多可能的数字。

本次实现为UUID.randomUUID().toString().replaceAll("-", "")。

3. Repository接口

说明（1）：count函数

JpaRepository及数据库连接配置已经在User模块做出说明。而App模块出现了一个新的使用，count函数的使用，JpaRepository十分强大，关键词如find、delete、count定义的方法只要符合它的命名规则，就不需要实现，而count的使用如下图所示，其中和find类似，鉴于表中不存在相同的行，All关键词可以省略。

|  |
| --- |
| @Repository  public interface AppMemberRepository extends JpaRepository<AppMember, AppMemberKeys> {  //统计用户指定身份的应用数量  Integer countAllByUserIdAndType(int userId, String type);  }  @Repository  public interface AppVerRepository extends JpaRepository<AppVersion, AppVersionKeys> {  //统计app的版本数  Integer countAllByAppKey(String appKey);  } |

图4.10 cont函数的使用

说明（2）：主键类的创建和使用

另外一个新出现的问题是JpaRepository必须使用主键类作为基础功能实现的依据，而AppMember和AppVersion为多个属性值组成的复合主键。所以要想使用JpaRepository，还必须依据Spring Boot的复合主键创建规则，编写主键类。

编写AppMember的复合主键类如下图所示，可以看到主键类的编写类似于实体类，但是不包含实体注释。除此以外，主键类还必须实现序列化，实现自定义hashCode()，equals()方法，如此一个主键类就完成了。

|  |
| --- |
| import java.io.Serializable;  public class AppMemberKeys implements Serializable {  private String appKey;  private Integer userId;  public AppMemberKeys(){}  public AppMemberKeys(String appKey, Integer userId) {  this.appKey = appKey;  this.userId = userId;  }  @Override  public int hashCode() {  //自定义hash值  ……  }  @Override  public boolean equals(Object obj) {  //自定义比较方法  ……  }  public String getAppKey() {  return appKey;  }  public void setAppKey(String appKey) {  this.appKey = appKey;  }  public Integer getUserId() {  return userId;  }  public void setUserId(Integer userId) {  this.userId = userId;  }  } |

图4.11复合主键AppMemberKeys

完成一个主键类的创建之后，还必须如下图所示，将主键类通过@IdClass予以配置。

|  |
| --- |
| @Entity(name = "app\_member")  @IdClass(AppMemberKeys.class)  public class AppMember { …… } |

图4.12主键类的使用

### 4.3.4动态模型

以下是一些bughunter后台系统App模块的运行动态模型，这里只列举典型且与之前User模块有较大区别的几个例子。

1.添加应用顺序图

如下图所示为增加应用的顺序图，需要首先依靠AppRepository增加应用信息，之后还必须在应用成员表添加创建者信息。值得一提的是JpaRepository插入删除操作无返回信息，所以实现中以返回的异常判断操作的错误类型。因为操作为用户发起，所以并没有验证该用户是否存在。

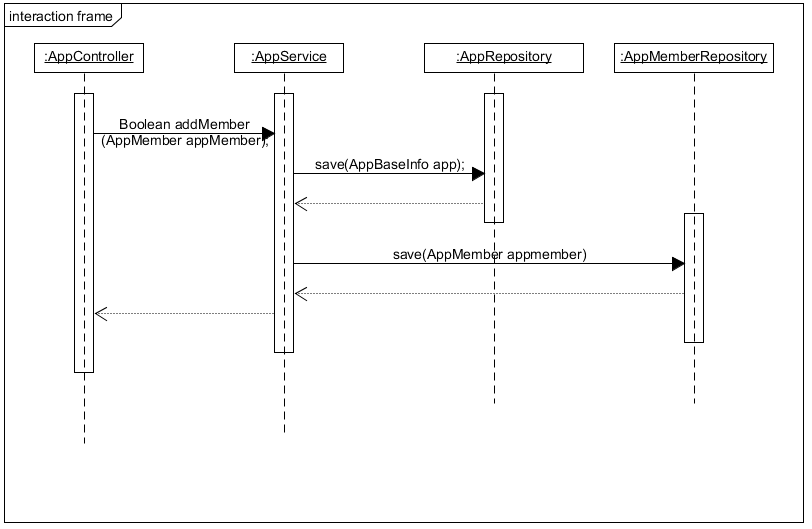


图4.13添加应用顺序图

2.获取应用成员顺序图

如下图所示为后台获取给定app的所有成员信息，相关对象之间的协作。不同于之前的操作，这里出现了多次数据库读取。因为JpaRepository使用还不够熟练，这里采用了多次读取的方法，即先从AppMemberRepositroy读取成员表，在从UserRepository中读取成员更详细的信息。有待改进。

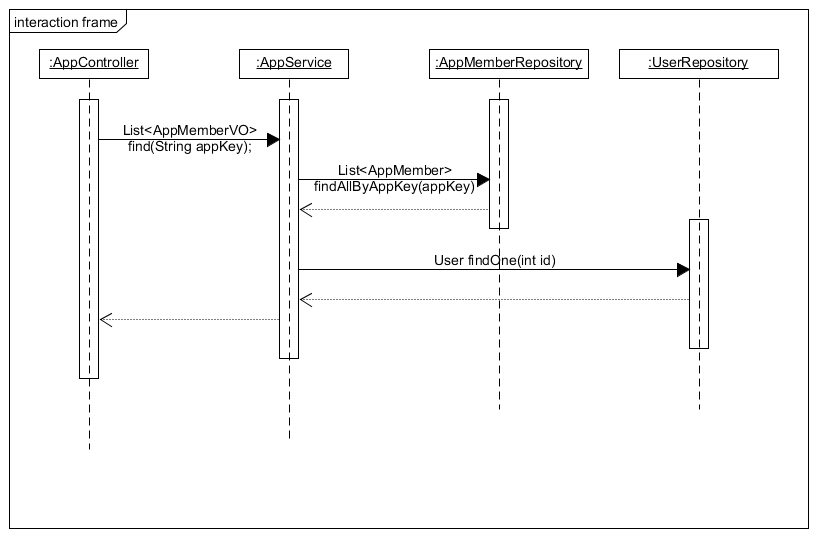


图4.14获取成员顺序图

3.增加应用成员顺序图

如下图所示为增加应用成员的顺序图，需要首先依靠UserRepository验证用户是否存在，鉴于增加成员反复是在应用存在的条件下调用，故不验证该应用是否存在。

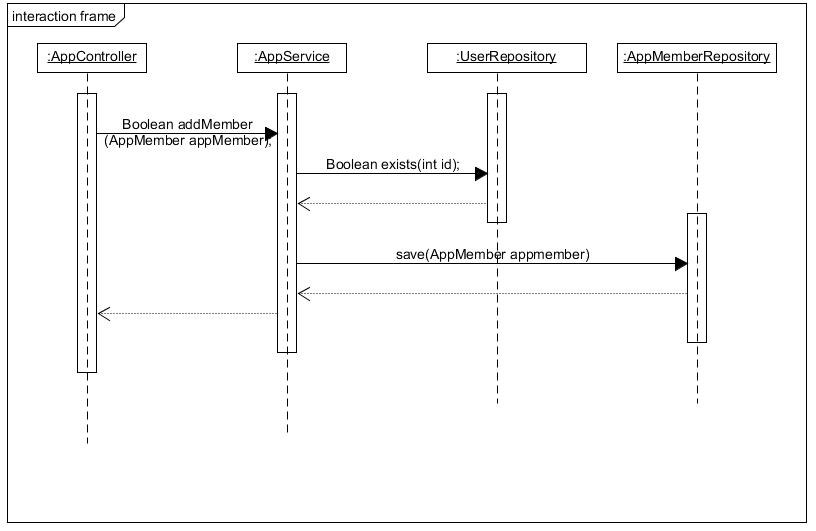


图4.15添加成员顺序图

## 4.4 Bug模块详细设计

### 4.4.1模块概述

Bug模块主要负责为应用Bug相关提供服务，提供应用bug信息、修改历史管理管理服务。各项服务与User模块一样都通过提供API，处理响应HTTP请求的方式完成任务。Bug模块是bughunter系统的重中之重，是系统的最主要功能点，Bug模块也是功能最多，最复杂的模块。该模块的实现的好与差将直接影响到bughunter系统的使用体验。

### 4.4.2整体结构

与User模块和App模块相同，采用分层风格，分为controller层，service层，data层。每一层之间为了灵活性，添加了BugService的接口，以实现针对接口编程。尽管Bug模块最为复杂，其实现与其他模块并没有本质的区别。Bug模块的设计如下图所示，一些说明类似于User模块和App模块，在此不再重复。这里的Bug实际分成BugBaseInfo、BugConsoleLog、BugDeviceInfo、BugOperateStep分别为bug基本信息、控制台输出、设备信息、操作步骤。为图示的简洁明了，予以省略。

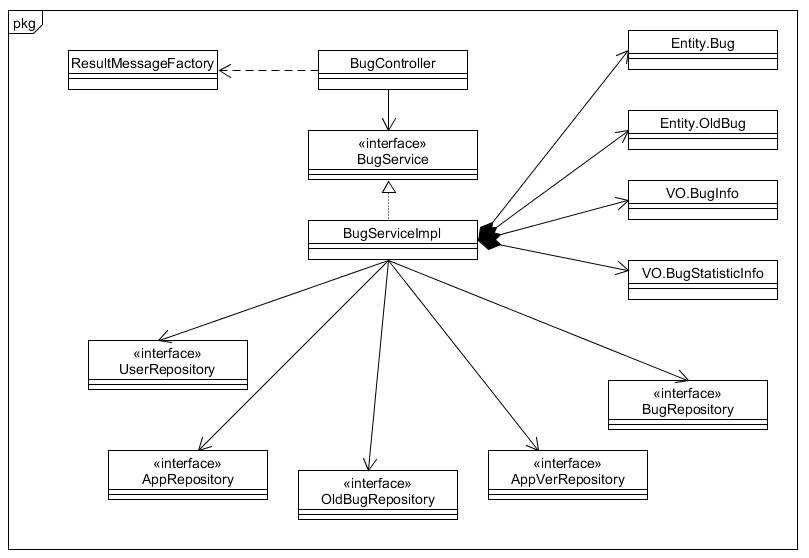


图4.16 Bug模块设计

### 4.4.3接口及代码示例

1.BugController类的接口规范

如下表所示：

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | @RequestMapping(value = "/{appKey}/{current}/getCurrentActivityBug",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getCurrentActivityBug(  @PathVariable String appKey, @PathVariable String current)； |
| 说明 | Android端获取当前页面已存在的bug信息 |
| 语法 | @RequestMapping(value = "/{appKey}/getAll", method = RequestMethod.GET)  public @ResponseBodyResultMessage getAllBugByAppKey(  @PathVariable String appKey)； |
| 说明 | 通过AppKey获取该应用存在的所有bug |
| 语法 | @RequestMapping(value = "/{userId}/getAllSubmit",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getAllBugByUserId(  @PathVariable int userId)； |
| 说明 | 通过用户id获取该用户所提交的所有bug |
| 语法 | @RequestMapping(value = "/{appKey}/{appVersion}/getAllByAppVersion",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getAllByAppVersion(  @PathVariable String appKey, @PathVariable String appVersion)； |
| 说明 | 获取应用的所有版本号 |
| 语法 | @RequestMapping(value = "/submit", method = RequestMethod.POST)  public @ResponseBody  ResultMessage submitBug(@RequestParam(name = "bug") String jsonStr,  @RequestParam(name = "screenshot", required = false) MultipartFile file)； |
| 说明 | 提交Bug，供Android和Web端共同使用 |
| 语法 | @RequestMapping(value="/{appKey}/{bugId}/get",method= RequestMethod.GET)  public @ResponseBody ResultMessage getBugById(  @PathVariable String appKey, @PathVariable String bugId)； |
| 说明 | 通过bug的id获取bug信息 |
| 语法 | @RequestMapping(method = RequestMethod.GET,  value = "/screenshot/{filename:.+}")  public @ResponseBody  ResponseEntity<?> getScreenshot(@PathVariable String filename)； |
| 说明 | 通过bug信息中存在的图片地址来获取bug截图 |
| 语法 | @RequestMapping(value = "/{appKey}/getAfterScreen",  method = RequestMethod.POST)  public @ResponseBody ResultMessage getAllBugByScreen(  @PathVariable String appKey, @RequestBody String jsonStr)； |
| 说明 | 通过类型，紧急程度，状态三项筛选bug |
| 语法 | @RequestMapping(value = "/{appKey}/{bugId}/oldGet",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getOldBugById(  @PathVariable String appKey, @PathVariable String bugId)； |
| 说明 | 获取bug的历史修改信息 |
| 语法 | @RequestMapping(value = "/{appKey}/{bugId}/modify",  method = RequestMethod.POST)  public @ResponseBody ResultMessage modifyBug(  @PathVariable String appKey, @PathVariable String bugId,  @RequestBody String jsonStr)； |
| 说明 | 修改bug信息，并且保留修改历史 |
| 语法 | @RequestMapping(value = "/{appKey}/{bugId}/delete",  method = RequestMethod.POST)  public @ResponseBody ResultMessage deleteBug(  @PathVariable String appKey, @PathVariable String bugId,  @RequestBody String jsonStr)； |
| 说明 | 删除一个bug |
| 语法 | @RequestMapping(value = "/{appKey}/{appVersion}/getStatisticInfo",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getStatisticInfo(  @PathVariable String appKey, @PathVariable String appVersion)； |
| 说明 | 获取应用一个版本的bug简单统计信息 |
| 语法 | @RequestMapping(value = "/{appKey}/getSimpleStatistic",  method = RequestMethod.GET)  public @ResponseBody ResultMessage getSimpleStatistic(  @PathVariable String appKey)； |
| 说明 | 获取应用所有版本的简单统计信息 |

表4.5 BugController设计

已在User模块和App模块说明的内容不再赘述。下面说明Bug模块独有的两项任务实现。

说明（1）：文件上传

上传Bug的方法代码如下图所示，其中包含了文件上传。

|  |
| --- |
| @RequestMapping(value = "/submit", method = RequestMethod.POST)  public @ResponseBody  ResultMessage submitBug(@RequestParam(name = "bug") String jsonStr,  @RequestParam(name = "screenshot", required = false) MultipartFile file) {  String screenshotName = Constants.SCREENSHOT\_NO\_EXIST;  if (file != null && !file.isEmpty()) {  try {  File logoSaveFile = new File(Constants.SCREENSHOT\_BASE\_URL);  if (!logoSaveFile.exists()) {  logoSaveFile.mkdirs();  }  screenshotName = file.getOriginalFilename();  String suffix = file.getOriginalFilename().substring  (file.getOriginalFilename().lastIndexOf("."));  String screenshotFileName =  Constants.SCREENSHOT\_BASE\_URL + File.separator + screenshotName;  File screenshotFile = new File(screenshotFileName);  file.transferTo(screenshotFile);  } catch (FileNotFoundException e) {  e.printStackTrace();  return new ResultMessage(1, e.getMessage());  } catch (IOException e) {  e.printStackTrace();  return new ResultMessage(1, e.getMessage());  }  }  JSONObject jsonObject = new JSONObject(jsonStr);  BugInfo bugInfo = new BugInfo(jsonObject, screenshotName);  BugInfoKeys bugInfoKey = bugService.addBug(bugInfo);  return ResultMessageFactory.getResultMessage(bugInfoKey);  } |

图4.17文件上传的实现

该实现利用了multipart/form-data类型请求上传图片，使用Java提供的MultipartFile类接收和处理图片文件。可以看到图片保存到服务器的某一文件夹下，将文件路径保存到数据库。防止图片文件过大，存储到数据库影响数据库的性能。

说明（2）：服务器图片获取

服务器图片获取方法代码如下图所示：

|  |
| --- |
| @Autowired  public BugController(ResourceLoader resourceLoader) {  this.resourceLoader = resourceLoader;  }  @RequestMapping(method = RequestMethod.GET, value = "/screenshot/{filename:.+}")  public @ResponseBody  Resource getScreenshot(@PathVariable String filename) {  try {  return resourceLoader.getResource(  "file:" + Paths.get(Constants.SCREENSHOT\_BASE\_URL, filename).toString());  } catch (Exception e) {  return null;  }  } |

图4.18服务器资源获取

ResourceLoader接口在Spring中定义资源的加载，ResourceLoader提供 classpath下单资源文件的载入，其将加载classpath下所有资源，并返回第一个匹配的资源。鉴于图片名是唯一的，所以使用该方法是可以保证可用性的。

2. BugService接口设计

如下表所示:

|  |  |
| --- | --- |
| 提供的服务 | |
| 语法 | Boolean deleteBug(BugInfoKeys bugInfoKeys); |
| 说明 | 删除bug，返回结果 |
| 语法 | BugInfoKeys addBug(BugInfo bugInfo); |
| 说明 | 增加bug，返回主键信息 |
| 语法 | Boolean modifyBug(BugInfoKeys bugInfoKeys, JSONObject jsonObject); |
| 说明 | 修改bug信息，并保留历史记录 |
| 语法 | BugInfoVO findWholeBug(BugInfoKeys bugInfoKeys); |
| 说明 | 获取bug整体信息，包括基本信息，控制台输出，设备信息，操作步骤 |
| 语法 | List<BugBaseInfoVO> findAllBugByAppId(String appKey); |
| 说明 | 根据AppKey获取所有bug基本信息 |
| 语法 | List<BugBaseInfoVO> findAllBugByScreen(String appKey,  JSONObject jsonObject); |
| 说明 | 根据筛选信息返回bug列表 |
| 语法 | public List<BugInfoVO> findCurrentBugs(String appKey, String current); |
| 说明 | 根据应用和当前界面信息，返回已存在的bug信息 |
| 语法 | List<OldBugBaseInfoVO> findOldBug(String appKey, String bugId); |
| 说明 | 获取bug的历史修改记录 |
| 语法 | List<BugBaseInfoVO> findAllBugByAppKeyAndVersion(  String appKey, String appVersion); |
| 说明 | 根据应用和版本号获取所有bug |
| 语法 | BugStatisticInfo getStatisticInfo(String appKey, String appVersion); |
| 说明 | 根据应用和版本号获取统计信息 |
| 语法 | List<BugBaseInfoVO> findAllByUserId(int userId); |
| 说明 | 获取用户提交的所有bug |
| 语法 | Integer[] getSimpleStatistic(String appKey); |
| 说明 | 获取应用所有版本不同状态，紧急程度，类型的简单统计信息 |

表4.6 BugService接口设计

3.JpaRepository接口自定义SQL语句

在bug模块中简单的JpaRepository应用已经不能满足需要，即使可以实现，也是对效率的极大浪费。所以需要使用自定义的SQL语句。而贯彻Spring Boot对开发友好的一贯风格，JpaRepository中对自定义SQL语句的使用时十分简单的，如下图所示：

|  |
| --- |
| @Query(  value = "select distinct user\_id from app\_bug\_info b where b.app\_key=?1 and b.app\_version=?2",  nativeQuery = true)  List findUserByByAppKeyAndAppVersion(String appKey, String appVersion);  @Query(  value = "select distinct b.system\_version,b.device\_brand,b.providers\_name,b.system\_model from bug\_device\_info b join app\_bug\_info a using(app\_key,bug\_id) where a.app\_key=?1 and a.app\_version=?2 and a.priority=?3", nativeQuery = true)  List findDeviceByByAppKeyAndAppVersion(String appKey, String appVersion, String priority); |

图4.19 自定义SQL

只需要一个简单的注解并在注解中写入自定义SQL语句，就可以实现了。而多表间的SQL语句实现如图所示也没有什么区别。实现中方法名可以任写，参数值和返回值必须对应SQL语句。即使方法名与JpaRepository查询规范一致，Spring Boot也会依据<jpa:repositories>提供的query-lookup-strategy 属性值，判断是按照查询规范还是自定义SQL语句实现方法。

### 4.4.4动态模型

以下是一些bughunter后台系统Bug模块的运行动态模型，这里只列举典型且与之前User、App模块有较大区别的几个例子。

1.提交Bug顺序图

如下图所示，实际的过程没有图示的那么简单，BugController实际还完成了图片文件的本地存储，并且Bug的信息实际是分成了四张表在数据库中存储的。

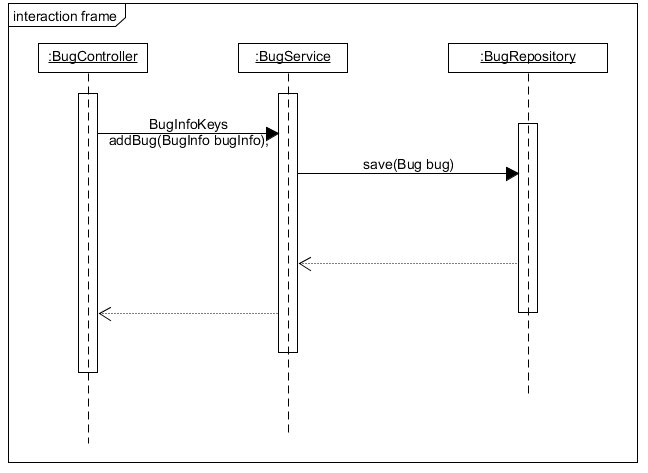


图4.20提交bug顺序图

2.获取服务器图片资源

如下图所示，该情况特殊在resourceLoader是Spring Boot提供的依赖注入的对象。

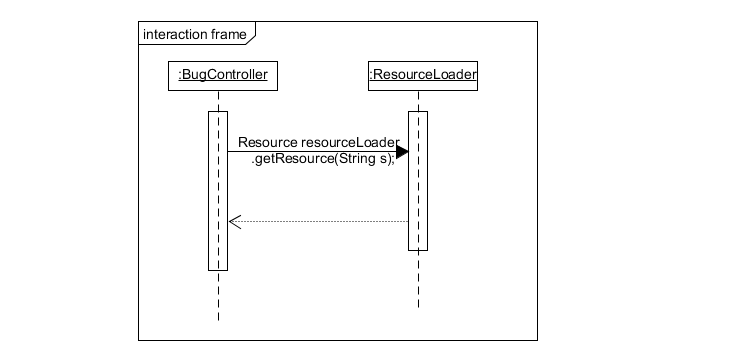


图4.21获取图片资源顺序图

3.修改bug顺序图

如下图所示，考虑到项目需求，修改bug之前需要先保存其修改前信息，以作为bug修改的历史记录。

![C:\Users\st0001\AppData\Roaming\Tencent\Users\2649258169\TIM\WinTemp\RichOle\GIK@~~0WLU]Q%Q5Q](_TXHE.png](data:image/png;base64,)

图4.22修改bug顺序图

4.获取统计数据顺序图

如下图所示，可以看到为了获取应用某一版本的bug统计数据，多次调用数据库查询，十分繁琐。实际应当有更好的实现，如在数据库中设置索引等。

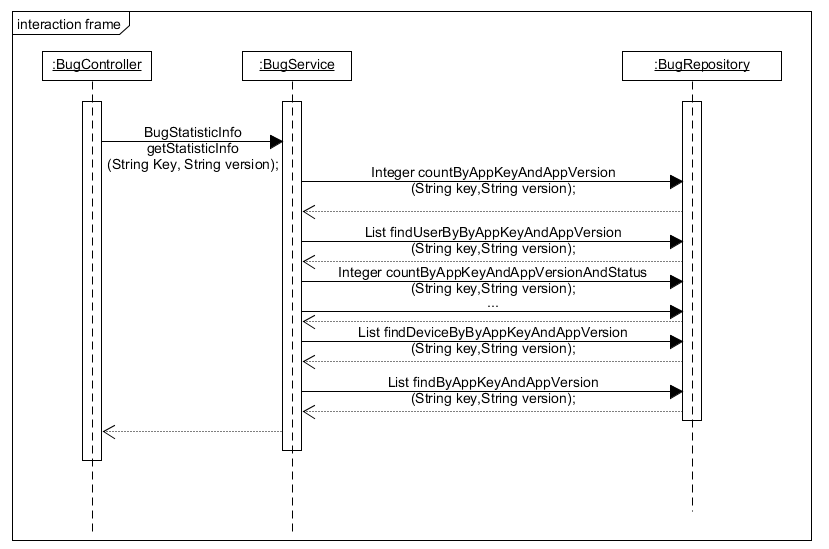


图4.23获取统计数据

## 4.5本章小结

本章主要通过介绍项目分层结构和各模块的详细设计，详细的阐述了系统的设计细节。并通过展现关键代码，帮助理解重要功能的实现。通过项目运行顺序图的形式，更加直观具体的表现项目的运行。

# 第五章总结与展望

## 5.1总结

## 5.2展望

# 参考文献

# 致谢

在论文最后，我想向所有在毕业设计阶段提供过帮助的相关者表达最真挚的感谢。

首先，我要感谢我的指导老师房春荣老师，在我本科的最后阶段毕业设计给予的帮助和指导。项目选题阶段因为对安卓的兴趣选择了房老师的课题，完成选题后，房老师积极指导我们完成毕业设计，鉴于我考研的实际情况，房老师鼓励了我和毛越同学，并且推迟了项目进度。在项目进行中，每周六房老师都会召开例会，倾听我们汇报项目进度，给我们予以项目方向上的指导。

其次，还要感谢组员石婉容同学，毛越同学，郑西宁同学。石婉容同学负责Web前端的开发，提出了很多有助于项目改进的意见。毛越和郑西宁同学负责完成Android端SDK的开发。其中毛越同学也参与了考研，感谢他在考研上的一些帮助。正是这些组员的存在，才能完成本次毕业设计，并且我个人的各项能力也得到了提升。

还要感谢我的三个舍友，在毕业设计阶段是你们陪伴我最多，给我带来欢乐和鼓舞，每日的互相吹捧都是前进的动力。

最后要感谢南京大学软件学院对我四年来的培养，感谢黄蕾辅导员老师在生活和学习上的监督与关照，感谢软件学院所有老师授道解惑。