分类号_____ 密级_____

UDC _____ 学号_____

太原理工大学

毕业设计（论文）

论文题目 基于 SSM 国家逃犯网络在线查询系统的设计与实现

Thesis Topic Design And Implementation of Fugitive Criminal Query System Based on SSM

学 生 姓 名	刘磊
学 号	2015005973
所 在 院 系	软件学院
专 业 班 级	软件 1516 班
导师姓名职称	张玲 讲师
完 成 日 期	2019 年 6 月 3 日

2019 年 6 月 3 日

太 原 理 工 大 学

太 原 理 工 大 学

毕业设计（论文）任务书

毕业设计（论文）题目：

基于 SSM 国家逃犯网络在线查询系统的设计与实现

毕业设计（论文）要求及原始数据（资料）：

1. 综述 Web 应用开发技术现状；
2. 深入了解 Java 平台的 web 应用开发技术；
3. 熟练掌握 SSM 框架组合的使用方式；
4. 训练软件工程过程规范和文档书写能力；
5. 使用 SSM 框架组合实现逃犯在线查询系统；
6. 训练检索文献资料和利用文献资料的能力；
7. 训练撰写技术文档与学位论文的能力。

毕业设计（论文）主要内容：

1．综述现代 web 应用开发技术；

2．了解查询系统的设计和实现方式；

3．熟悉 Java 平台的 web 应用程序开发；

4．熟悉 SSM 框架的概念和使用方法；

5. 使用 SSM 框架组合实现逃犯在线查询系统；

6. 完善软件开发过程中的主要文档

7. 完成软件测试及其结果分析。

学生应交出的设计文件（论文）：

1．内容完整、层次清晰、叙述流畅、排版规范的毕业设计论文；

2．包括毕业设计论文、源程序等内容在内的毕业设计电子文档及其它相关材料。

主要参考文献（资料）：

[1] Guillaume Chau. Vue.js 2 Web Development Projects[M]. Britain: Packt, 2017.

[2] Ravi Sharma, Shipra Ravi Kumar. Strategies for Web Application Development Methodologies [R]. ICCCA, 2016 .

[3] Rick Osowski . Introduction to microservices[J/OL]. IBM Developer, 2015 .

[4] Craig Walls, 张卫滨. Spring 实战(第四版) [M]. 北京: 人民邮电出版社, 2014 .

[5] 郝佳.Spring 源码深度解析(第二版)[M].北京: 人民邮电出版社, 2019.

[6] 章仕锋,潘善亮.Docker 技术在微服务中的应用[J/OL].电子技术与软件工程,2019(04).

[7] Martin R C. Agile software development: principles, patterns, and practices[M]. Prentice Hall, 2002.

[8] 郭丞乾,蔡权伟,林璟锵,刘丽敏.单点登录协议实现的安全分析[J].信息安全究,2019,5(01) .

[9] 魏春来,付永振.基于微服务的 DevOps 研究与实现[J].网络安全和信息化,2018(11):54-56.

[10] Pivot.Building a RESTful Web Service[J/OL]. Spring.io

专业班级 软件 1516 班 学生 刘磊

要求设计（论文）工作起止日期 2019 年 3 月 18 日~2019 年 6 月 21 日

指导教师签字 日期 2019 年 3 月 15 日

教研室主任审查签字 日期

系主任批准签字 日期

基于 SSM 国家逃犯网络在线查询系统的设计与实现

摘　要

逃犯在线查询系统是为了在全国公安系统内快速共享逃犯信息、记录逃犯线索、方便追逃工作的开展而设计的。一个有效的逃犯信息管理系统可以提升逃犯在公安部门中的曝光率进而增加追逃的成功率。

目前公安系统信息化管理比较落后，系统比较老旧，与当前最新的信息管理系统差距较大，使用不方便；本文介绍了一种新设计的逃犯信息查询系统，可以提供逃犯信息管理、通缉令管理以及记录逃犯线索等功能。

本文基于 Java 平台最成熟的 web 应用开发框架组合 Spring、SpringMVC 和 MyBatis 设计并实现了一套国家逃犯网络在线查询系统，并且从应用开发到应用部署的每个过程都尽可能的使用了目前比较新的软件项目管理方式，系统具有很好的可扩展性和良好的用户体验。

关键词: SSM;SpringBoot;逃犯查询;查询系统

Design And Implementation of Fugitive Criminal Query System Based on SSM

Abstract

The fugitive online query system is designed to quickly share fugitive information, record fugitive clues, and facilitate work between the Police departments for chasing fugitives. An effective fugitive criminal information management system can increase the exposure of fugitives in the Police departments and increase the success rate of capture.

At present, the information management that Police departments use is relatively backward and inconvenient to use. There is a large gap between the system polices use and other business IMS. This paper introduces a newly designed fugitive information query system, which can provide fugitive information wanted orders management as well as record fugitive clues and more.

This article introduces the design and implementation of a fugitive online query system based on the most popular frameworks on Java platform: Spring, SpringMVC and MyBatis. There are a lot of new technologies have been uses in the process of developing the system, which make this system easy to use and available to add new functions to it.

Keywords: SSM;SpringBoot;fugitive query;query system

目　录

第 1 章　绪论

1.1. 课题背景

从 1990 年第一个网页浏览器诞生以来，web 应用开发已经有了将近 30 个年头的历史，从最初的完全静态页面，到后来支持数据持久化和动态计算与显示。后来使用 JavaEE 平台来开发 web 应用成为最广泛的方案，并且在此过程中出现了非常经典的 MVC 应用架构。但是 MVC 应用架构是单体应用时代的产物，主要采用的是服务端对数据进行处理并直接填充进最终的 html 代码中的方式；最终返回给浏览器的内容是一切都组装好的。

最近的几年 web 应用的开发出现了更大的变化，MVC 的方式正在逐渐成为过去，更新的方式是前后端完全分离。后端应用编写时不需要再考虑视图的问题，只需要通过 HTTP 协议对外提供数据接口即可；使用 JSON[1]作为数据序列化格式，web API 采用 REST[2]风格设计；最后前端应用程序使用 MVVM[3]框架来构建单页面应用(Single Page Application)，全部的数据交互都是通过发起 HTTP 异步请求的方式来完成。

1.2. 本文研究内容

1.2.1. 逃犯查询系统

对于本题目，通过多种途径搜索资料关于逃犯查询系统的信息少之又少，几乎没有相关的文献或项目。对于逃犯查询系统的相关功能，在公开资料中本文没有可借鉴资源。所以在功能方面，将仅实现基本的逃犯信息的增删查改，另外包括实现一个信息管理系统最基本的用户识别和安全控制。

对于查询系统而言，可以理解为是一个简单的信息管理系统，其最主要的功能便是查询。由于所存储的信息具有高维度的特性，在对已有信息进行检索时，应该支持较为全面且灵活的组合查询条件。有信息查询的地方少不了数据的录入，对于本课题中的逃犯查询系统而言，信息查询的主要使用者也同样是信息数据的主要来源，所以本查询系统对于基础数据的添加和更新也将提供较为全面的支持。

另外对于逃犯查询系统，属于一个数据敏感型的特殊系统，目标用户是全国公安系

[1] JSON 即 JavaScript Object Notation，一种数据序列化格式，使用 key:value 的方式来嵌套的定义数据结构
[2] 即 Representational State Transfer，详见 https://zh.wikipedia.org/wiki/表现层状态转换
[3] MVVM 即 Model-View-viewmodel，详见 https://zh.wikipedia.org/wiki/MVVM

统内的警察。全国公安系统通过本逃犯查询系统对逃犯和通缉信息进行管理，也可以进行逃犯信息的共享，以帮助公安系统及时了解全国的在逃人员信息，提升抓捕逃犯的成功概率。

1.2.2. 现代 web 应用开发技术

本论文的研究重点一方面是设计实现逃犯查询系统，更重要的一方面是在这个过程中尽可能多的使用 web 应用软件开发领域的最新技术来设计、开发、架构和运行一套 web 应用软件，并以此宏观的介绍 web 前端开发、web 后端开发以及 web 应用实施部署方案的前沿技术。这主要包括 web 前端开发技术中的 MVVM 框架--Vue、后端微服务架构、后端开发风格--RESTful API、使用 Docker 容器技术在开发和生产环境来部署各种服务和中间件；以及使用 Kubernetes 在生产环境（集群）中进行容器编排与负载均衡。

之所以分别强调前端和后端开发技术，是因为目前大部分的中大型企业应用开发几乎全都采用前后端完全分离开发的方式。前端程序已经不是单纯的设计传统 html 页面那么简单了，前端应用程序独立部署和运行，完全是一个独立的客户端程序，只不过是运行在现代浏览器中的。前端应用程序也需要处理部分业务逻辑，ECMAScript 作为浏览器脚本语言的标准，在具体实现：TypeScript 和 JavaScript 的支持下，使得客户端程序具有完备的可编程特性。诸如数据动态加载、丰富的用户交互逻辑、友好流畅的图形界面、数据校验、数据预处理以及数据排序等等任务都可以由运行在浏览器中的客户端程序来完成。图 1.1 所示为前后端分离应用架构方式。

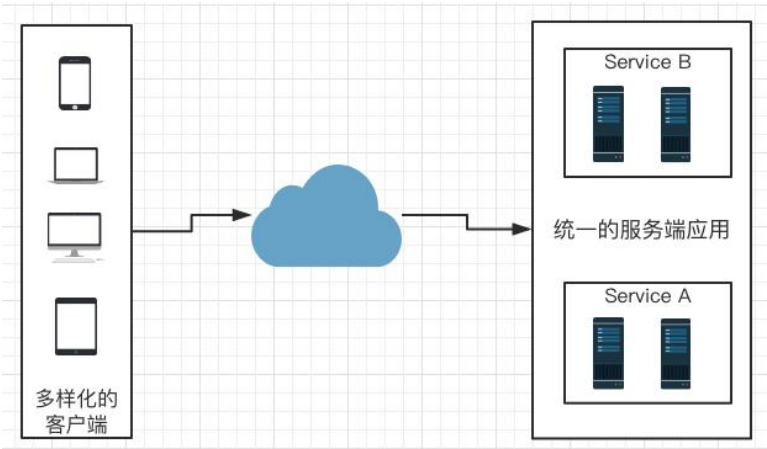


图 1.1 前后端应用分离支持的应用模式

虽然现代浏览器完全可以满足独立客户端应用程序的运行环境要求，但是要完全按

照这种标准进行开发并不容易，主要涉及到网络请求的发送、接收并通过 DOM 操作局部更新页面内容的操作，在原生 JavaScript 的环境中开发会变得特别低效。为了高效的开发浏览器中的客户端应用程序，和后端程序开发一样，出现了一系列优秀的应用程序开发框架，来帮助开发者工程化的完成前端应用开发，并且由于前后端程序开发的分离，前端开发可以独立出一个团队来单独管理，从软件工程的角度来讲这也是一种进步。

Web 应用的业务逻辑主要是在服务端进行的，在云计算时代，更多的程序其实是运行在服务端的，在软件使用过程中产生的数据也都保存在服务端。所以 web 应用开发中服务端程序的开发向来是最重要的。互联网飞速发展的 20 几年，为了将后端软件开发工程化和可团队化，出现了很多后端开发技术，其中包括微软 Windows 平台的 ASP.NET，Java 平台下 Oracle 的 JavaEE、Pivotal 的 Spring 套件等等。其中 Java 平台广泛的平台兼容性使得它占据了服务端应用开发的最广大市场。本论文所描述的“国家逃犯在线查询系统”也将基于 Java 平台上最流行的后端应用开发框架 Spring 套件来进行构建。

Spring 套件主要包括 SpringCore 组件提供的依赖注入和面向切面编程支持、SpringMVC 组件提供的高层 Servlet 封装支持以及广泛的第三方兼容组件的支持。本论文涉及的项目中使用了一款兼容 Spring 的 ORM 框架：MyBatis 来进行数据持久化层的开发，使用 Logback 进行日志记录。

前后端分离的另一个好处是，在移动互联网时代，终端设备多种多样，客户端生态也是多种多样；目前一个成熟的 2C 商业项目尤其是社交类应用，一般都会适配 Web 版、Android 版、iOS 版甚至还需要提供对微信小程序、支付宝小程序以及头条系小程序的支持。如此多元化的客户端接入，如果依然使用七八年前流行的 MVC 架构进行设计，后端应用程序是无法统一进行支持的，Web 版应用和其他平台应用就会割裂开来。

要想一套后端支持所有客户端访问，就应该有一套通用的数据交互标准，目前服务端应用开发通常是提供基于 JSON 数据序列化且使用 HTTP 协议进行通信的通用 Web API；当然这是遵循 REST 风格而使用的一种标准，另外 SOAP（Single Object Access Protocal）风格的接口开发使用 xml 作为数据交互格式，并不太适合今天的客户端软件。本系统的开发采用前者方案。HTTP 协议是互联网时代最为广泛支持的应用层网络协议，理所当然的成为大部分客户端应用通用的网络协议，虽然经常需要适配很多种客户端，但是并不需要自己开发一套全新的通讯协议。

然后是微服务架构；由于信息管理系统的数据存储、计算、主要逻辑处理、安全访问控制等等都需要在服务端进行，服务端应用变得越来越庞大，模块划分也越来越多，一个需要不断维护的系统如果依然采用传统的单体应用开发的模式，很不利于团队协作，并且所有的业务逻辑都集成在一个项目里耦合度很高，改动一处全局都需要重新编译或打包。微服务的应用架构将处理同一个方面的业务逻辑拆分成一个个独立的项目，多个微服务之间也通过约定的通信标准进行进程通信或网络通信（比如使用和客户端通信相同的 RESTful API，或者使用通信效率更高的 RPC 的方式），各个微服务独立开发和部署，互不干扰；每个微服务不限定开发语言和开发环境，仅要求可以使用约定的通讯标准进行微服务之间的数据交互。

最后是容器化应用。云计算的显著特征是使用大量的廉价机器组成大规模的计算集群，以一个低成本的方式提供比昂贵的小型机性能还强的计算和存储能力。在公有云计算平台上部署应用时通常是购买一个定量非配计算能力和RAM的完全独立的虚拟操作系统。在云计算平台上部署微服务应用一定会遇到生产环境相互不兼容的情况，一方面是开发环境与生产环境的软件差异，另一方面是不同的服务之间可能存在操作软件环境的冲突。使用传统的虚拟化技术将使用软件模拟整个操作系统，耗费了大量的计算资源，目前更加合适的解决方案是使用容器技术来隔离每个服务。

容器是一种轻量级的虚拟化技术，为每个应用提供命名空间和隔离的软件依赖环境，但是共享一个操作系统内核。如此既解决了不同的软件之间的软件环境冲突问题，业带来了非常显著的性能提升。容器技术与虚拟机对比如图 1.2 所示。

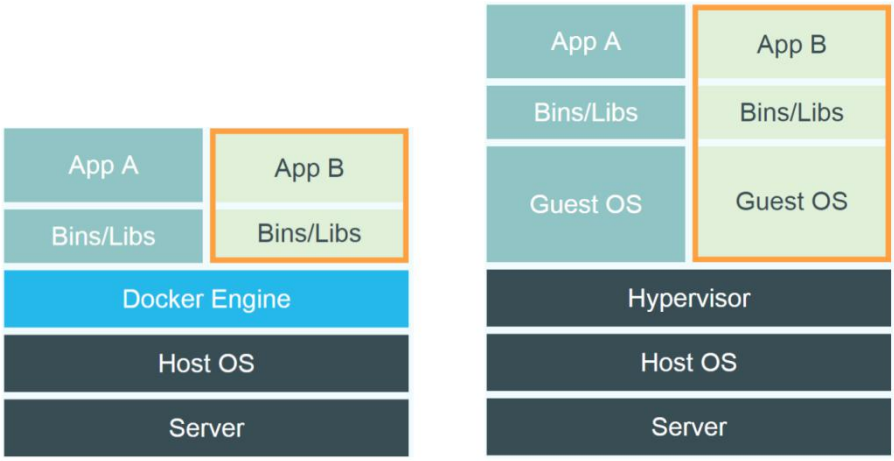


图 1.2 容器(左)对比虚拟机(右)

1.3. 本文结构

本文主体是一个 web 应用软件的开发文档，在软件开发过程的每个阶段对应的文档的适当位置来叙述现代 web 应用开发使用的新技术。

第一章绪论介绍项目背景和研究内容；第二章介绍逃犯查询系统开发和部署的各种软件工具和环境；第三章为逃犯查询系统的需求说明文档；第四章为逃犯查询系统的总体设计文档；第五章依据总体设计文档对系统实现进行描述，作为详细设计说明文档；第六章为系统测试文档；最后附上本论文的参考文献。

第 2 章　开发与运行环境介绍

2.1. 开发环境

在开发环境直接使用 Docker 来运行 PostgreSQL 和 Redis 等持久化和中间件工具，通过定义容器镜像文件来锁定各种软件版本；在线上环境部署应用时直接使用定义好的镜像和软件配置来创建容器实例。如此可消除由于开发环境与生产环境的软件差异带来的部署和调试困难的问题，另外也可以大大加快应用部署速度。

1) 硬件环境如表 2.1 所示

表 2.1 开发环境硬件

硬件名称	硬件信息
CPU	Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
内存	16 GB 2133 MHz LPDDR3
显示	13.3－英寸 (2560 x 1600)
	Intel Iris Plus Graphics 640 1536 MB 图形卡

2）软件环境如表 2.2 所示

表 2.2 开发环境软件

用途	软件项目	版本
操作系统	MacOS Mojave	10.14.4
容器引擎	Docker	CE 18.09.2
关系型数据库	PostgreSQL	11.2
缓存数据库	Redis	5.0.4
程序平台	Java (JDK \ JRE)	11.0.3
IDE	Intellij IDEA	Ultimate 2019.1

2.2. 开发过程辅助工具

1） 版本控制

软件开发过程通常是一个不断改进和迭代的过程，随着需求的不断完善和细化以及

系统设计的改进和重构，软件代码需要不断的满足要求而发生变化。

对于代码量很小的项目可能由单个开发者凭借对项目的熟悉即可熟知每一个变更细节，掌控全局代码；但是对于一个代码量比较大，功能模块较多的复杂软件系统而言，代码的管理变得非常困难；在代码作出修改版本进行演进之后很可能会出现 bug，这时无法追踪代码的改变会使得代码的修复变得比开发还困难。

另一个问题是，对于现代软件工程项目，通常是线上运行着一个稳定版本，而开发团队在不断研发着好多新特性，也会有不同的小团队来负责不同的特性开发；这些由不同的小组负责的新功能特性开发工作常常是同步进行的。这样就可能导致为了不同的开发任务编辑或修改了软件产品的同一处代码实现从而产生了代码冲突，此时如果没有一个强大的工具来管理代码，似乎新的几个功能根本没办法同时集成进已有系统。

首先 VCS 的最基本任务就是记录代码的修改。使用版本控制系统来追踪(Track)源代码文件，进行过一些修改后通过提交命令(commit)来提交本次修改，这样版本控制系统就会记录下这次修改的内容；随后可能发生了很多次 commit，版本控制系统都会记录下来。价值就在于对每一次 commit，随时都可以找回(checkout)查看并重新编辑之前发布的内容，每一个 commit 记录都是“时光机”的跳跃点。

VCS 的另一个重要作用就是解决线上线下多个软件版本在多个软件开发小组之间的协作问题。VCS 通过“分支”(branch)机制来管理处在不同开发进度的多个软件版本；对于每一个正在研发的新功能特性都可以建立一个分支，这些分支和主版本之间的关系就像树枝与树干的关系，各自可以有自己的版本演进(commit)；而强大之处就在与这些分支可以随意的进行合并(merge)，只要能解决不同分支之间的冲突[4](conflict)即可。产品分支模型如图 2.1 所示。

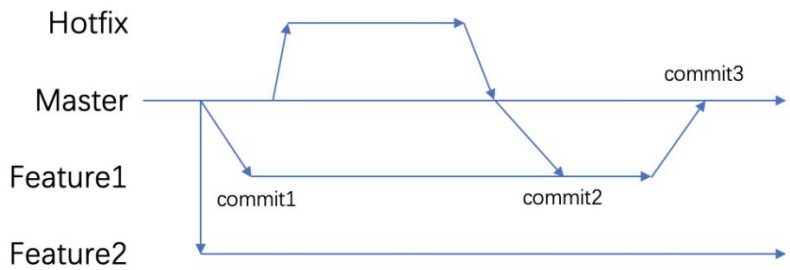


图 2.1 产品分支模型

[4] 冲突即不同的分支修改了同一个文本文件的同一行内容，VCS 不知道应该保留那一个分支的修改，此时需要人为决定。

本文中的示例项目将使用目前最流行的版本控制工具 GIT 来进行版本控制，并使用 GitHub 来作为远程 git 仓库来保存代码；项目将在本论文通过所有审核后变为开放访问权限。

本项目 GitHub 地址：

后端项目：https://github.com/linkinghack/criminalquery

前端项目：https://github.com/linkinghack/CriminalQueryApp

2） 依赖管理

在软件开发过程中时常需要调用一些第三方库来利用开源成熟解决方案解决软件开发中的某些问题，并且依赖项目之间也有相互的依赖关系。

在本文所述的逃犯查询系统中，将使用 Java 平台比较流行的 Apache Maven 来管理依赖组件。

Maven 可以用来管理 Java 软件项目开发过程中的依赖 jar 包，也可以用来进行软件构件(build)，还可以通过模块化的方式来拆分项目，并解决模块间的依赖调用；配合 IDE 的提示功能在模块间的依赖代码也可以获得完整的代码提示。Maven 对于依赖项的上层依赖和重复依赖都能自动解决。Maven 的一次自动构建结果如图 2.2 所示。

```
[INFO]
[INFO] -------------------< com.linkinghack:criminalquery >--------------------
[INFO] Building criminalquery 1.0-SNAPSHOT                               [4/4]
[INFO] --------------------------------[ pom ]---------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] criminalquery-model 0.0.1-SNAPSHOT ..... SUCCESS [  2.084 s]
[INFO] criminalquerybase 0.0.1-SNAPSHOT ....... SUCCESS [ 25.778 s]
[INFO] criminalquery-sso 0.0.1-SNAPSHOT ....... SUCCESS [ 10.367 s]
[INFO] criminalquery 1.0-SNAPSHOT ............. SUCCESS [  0.002 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  39.352 s
[INFO] Finished at: 2019-05-04T21:57:50+08:00
[INFO] ------------------------------------------------------------------------
(base) ➜ criminalquery-all git:(master) ✗
```

图 2.2 Maven 自动构建结果

2.3. 其他编程环境

1) SpringBoot

Spring 套件虽然强大，但是其繁杂冗长的配置文件让开发效率变得非常低，查文档补配置需要花费大量的时间，并且与 Spring 集成的各种第三方组件的配置通常是固定

的。为了解决配置困难问题以及适应微服务的开发，SpringBoot 提供了解决方案。

SpringBoot 通过一系列 Maven(或 Gradle)的 starter 项目来预置一些经常使用的项目配置方案，在 pom 中引入这些 starter 项目就可以直接开始开发业务相关的功能。在配置方面，SpringBoot 提供了全局配置方案，在项目 classpath 中直接编辑 application.yml 或 application.properties 文件即可对 Spring 和其他支持的第三方库进行统一配置。另外 SpringBoot 集成了内置的 Tomcat，可以将整个项目打包成可执行 jar，无需再单独安装 Servlet 容器。

2) Java 11

虽然目前用户最多的 Java 版本是 Java 8，但是自从 Oracle 宣布 Java 的半年期迭代计划以来，已经有四个主要版本更新，目前最新为 2019 年 3 月下旬发布的 Java 12。Java 11 是目前的 LTS(Long Term Support)版本，为了兼顾新特性和稳定性，故在本论文所涉及的项目中使用 Java 11（包括 JDK 和 JRE）。

3) Lombok

Lombok 是一个帮助创建 POJO 的小插件。在 Web 应用开发过程中通常需要定义许多 POJO 类，它们仅仅用来表示与数据库表对应的数据结构，除了必要的属性定义外，剩下的都是重复性的 Getter 和 Setter 方法。为了使代码更加简洁，节约由无意义的劳动浪费掉的时间，本项目使用 Lombok 来自动完成 Getter 和 Setter 的添加，只需要在 POJO 类上使用@Data 注解即可。

4) Golang

Golang 是 Google 开发的一款强类型编译型语言，其语法简单，核心库完善，已经成为现代 web 应用开发的新秀。越来越多的公司将 Golang 作为业务实现的主要语言；具有代替 JavaEE 的潜力。

在本论文所涉及的项目中使用 Golang 完成了全国行政区划数据库的数据导入工作。

5) PostgreSQL

PostgreSQL 是一个开源的关系型数据库管理系统，起源于伯克利大学的 POSTGRES 项目。PostgreSQL 支持非常丰富的数据类型，具有不错的性能表现。本系统使用 PostgreSQL 作为主要的数据持久化后端。在构建全国行政区划数据库时，行政区层级路径的存储便用到了 PostgreSQL 的数组类型。

程序与数据库建立连接使用了 SpringBoot 集成的 Hikari Pool。它是一个性能表现优异的数据库连接池工具。数据库连接池将保持几个数据库连接不断开，可以一直等待服务，在并发访问量增加时也能自动增加连接数量，动态平衡网络资源。通过设置活动连接数量来保证程序需要数据库访问时可以立即得到结果，节省了因为建立 TCP 连接而用掉的时间。

PostgreSQL 的部署，本系统使用 Docker 来部署 PostgreSQL，postgres 将作为一个服务在 docker-compose 中启动，在开发环境通过暴露 5432 端口到宿主机来支持开发测试，在生产环境将仅允许 docker 网络内部访问，保护数据库安全。

6) Redis

Redis 是一个内存数据库，使用 Key-Value 的方式保存数据，支持 String、List、Set、Sorted Set、Hash、Bit array、HyperLogLog、Stream 八种 Value 数据类型；而 Key 的数据类型则是支持使用任意的二进制序列，甚至可以用一张图片的二进制序列作为 Key。

在本系统中将 Redis 作为单点登录系统用于会话控制的工具，得益于 Redis 提供的数据有效期功能，单点登录系统在缓存 Token 和用户数据时将设置一个过期时间，此过期时间也就成为了 Token 对应的登录态的有效期；过期后 Token 访问无效，以此保护用户账号安全。

2.4. 小结

本章介绍了开发逃犯查询系统主要的软硬件环境和开发工具，以及适应于系统架构设计的软件运行环境和工具。 这些软件环境和工具不都是开发本文所述逃犯查询系统必须使用的，本文通过尽可能多的使用这些工具来展现 web 应用软件的开发思路。

第 3 章　需求分析

3.1. 可行性分析

1） 经济可行性分析

开发此系统总体来说难度不大，属于三到五人团队一个月的工作量。在部署实施方面，可以直接使用公有云计算业务提供的基础计算和存储能力；由于此系统是内部系统，服务的用户数量比较少比较稳定，不会具有突然增加的性能问题，所以计算资源的年支出费用将比较固定。

2） 技术可行性分析

由于本项目将完全重新开发，不需要考虑是否有就项目的整合问题，所以在技术选型上没有约束，选择最新的架构和开发方式。

为了降低对客户端的硬软件要求，采用 B/S 架构的基本方式，可以提供以 web 页面的方式访问系统全部功能的用户接口。

对于可扩展性的要求，可以采用后端基于最通用的 HTTP 协议提供 JSON REST API 的方式，如此可以放宽对客户端软件的支持范围。最初可以仅支持标准 web 页面访问，随后可以在后端逻辑只做很少甚至不需要改动的情况下迅速接入其他客户端类型，比如直接支持手机 App、微信小程序等等。

对于易维护性，推荐技术架构首选微服务方式，不同的模块之间耦合非常低，一个服务出现故障不至于影响整个系统。微服务架构方式配合容器化应用部署可以做到灵活的服务伸缩，如果系统用户的确不断增长也可以很快的增加计算能力。

3） 系统运行可行性

本系统是一个信息管理系统，采用 Java 平台 B/S 架构的方式开发应用。系统可以运行在任何支持 Java 环境的计算机中。

4） 管理和操作可行性

逃犯查询系统的管理和使用完全由公安机关内部进行，系统不对外开放。系统本身具有一定的管理功能，可以完成日常管理。系统后续维护可根据文档交给任何一个可信任的开发团队。

3.2. 功能需求分析

3.2.1. 系统基本流程

逃犯查询系统最基本功能便是录入逃犯相关信息，然后随时可以查询已经录入过的逃犯信息。目标用户是公安部门机关内部人员。

目标系统要求可以供全国公安部门使用，数据统一管理，保证一致性。另外要求应用部署简单，具有一定的可扩展性和较强的可维护性；同时对于客户端的硬软件要求尽量降到最低，做到随时随地可用。系统总体流程图如图 3.1 所示，总体数据流图如图 3.2 所示。

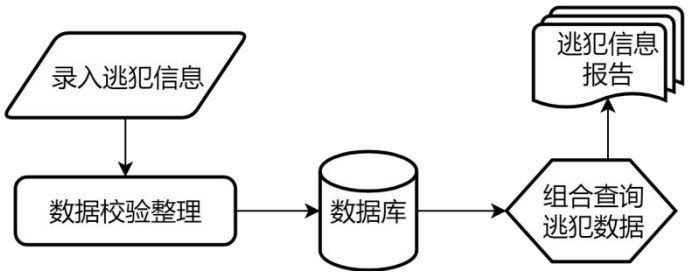


图 3.1 系统总体流程图

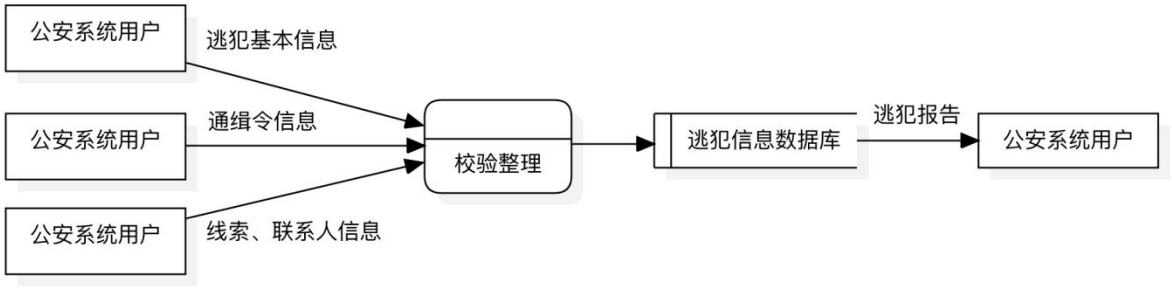


图 3.2 总体数据流图

3.2.2. 用例分析

1） 用户管理模块

用户管理模块需要完成系统用户识别和临时用户识别令牌 Token 的下发，新用户的登记，以及对系统其他模块接口提供安全访问控制。在用户管理模块对于普通用户和管理员用户有着不同的功能范围。

用户登录：仅限公安系统内部使用，除登录、注册之外的所有功能都与要有用户权限校验。

用户注册：为方便添加系统用户，允许公安系统用户自主申请账号，注册之后账号处于冻结状态，无法登录；等管理员核对通过后账号方可使用。用户注册时需要选择申请成为的用户角色和申请加入的部门，角色可选成为管理员或普通用户，选择部门时可选当前系统中已经添加的部门。

新用户审批：管理员对于新注册的处于冻结状态的账号可以进行激活操作，账号核对任务应该人工完成。

新用户驳回：管理员可以拒绝新用户的注册请求，驳回注册后删除此用户信息。

删除用户：对于系统内处于激活状态的用户也可进行删除操作。

用户管理模块用户用例如图 3.3 和 3.4 所示。

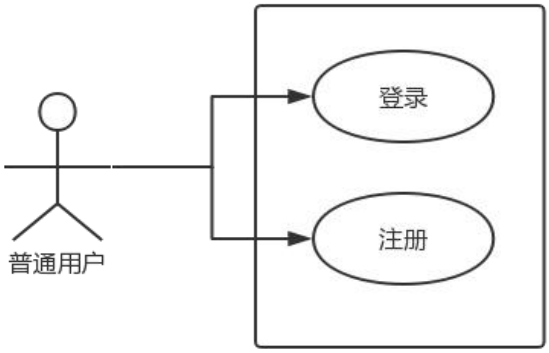


图 3.3 用户管理模块普通用户用例图

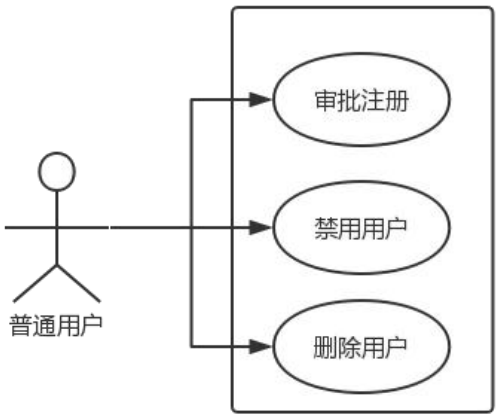


图 3.4 用户管理模块管理员用例图

2） 部门管理模块

部门管理模块需要为管理员提供对系统用户进行部门级别管理的功能，目标用户仅为管理员。部门是指公安机关的部门，例如“太原市公安局”、“小店区公安局”，每一个注册用户都属于一个部门，部门又具有层级关系。部门的层级关系来自于部门所管

辖的区域的包含关系。

添加部门：使用系统的公安部门灰度加入此系统，系统中对用户安部门进行简单管理。部门类似“公安部”、“xx 省公安厅”、“xx 市公安局”、“xx 县/区公安局”，对应部门信息中记录部门管辖区域，影响部门成员可以发布通缉令的通缉范围。

删除部门：可以对部门进行删除操作。

查看部门：管理员可以查看已有部门的列表，并显示部门拥有的成员数量。

部门管理模块用户用例如图 3.5 所示。

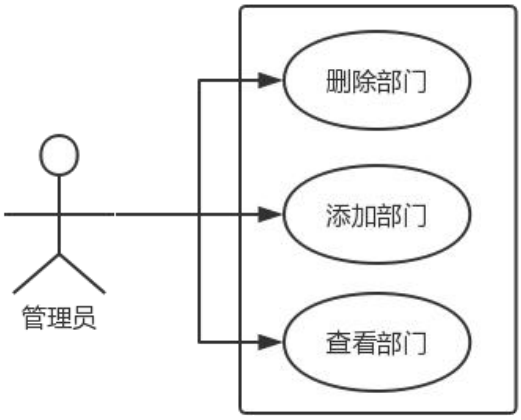


图 3.5 部门管理模块管理员用例图

3） 逃犯信息管理模块：

逃犯信息管理模块是本逃犯查询系统的最主要功能模块，使用本系统在各公安部门间发布和共享逃犯信息的基本诉求主要靠本模块来满足。逃犯信息管理模块提供丰富的逃犯信息的录入和多种逃犯信息搜索功能。

搜索逃犯信息：系统可以根据记录的多种逃犯信息进行任意组合的模糊搜索，也可以通过身份证号对系统数据库中的逃犯信息进行精确搜索。

通缉逃犯：系统可以对逃犯发起通缉，通常在第一次添加逃犯时就发起第一个通缉；随后系统中的其他用户可以对逃犯进行二次通缉。

修改通缉令状态：可以对通缉令的开关状态进行控制，影响搜索逃犯的条件。

发布广范围通缉：用户发起通缉时仅能在所在部门管辖区域内发布通缉信息，后续可以提供将通缉令发布至更大区域，由对应区域的管理员进行审核方可发布。

添加逃犯线索：系统需要具有一定的记录逃犯线索的能力，对于每一个逃犯可以添加数条线索。

添加逃犯联系人信息：对于每一个逃犯，可以添加其相关联系人信息，以供公安部门调查使用。

编辑逃犯信息：系统支持对已经加入到系统中的逃犯信息可以进行修改。

删除逃犯信息：系统管理员可以对已有的逃犯信息进行删除操作。

如图 3.6 所示为逃犯信息模块管理员用例图。

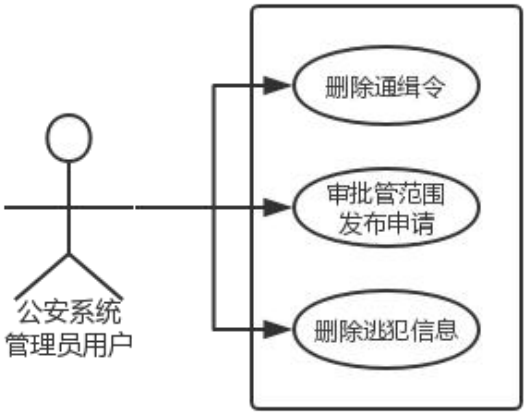


图 3.6 逃犯信息模块管理员用例图

4）总体用例模型

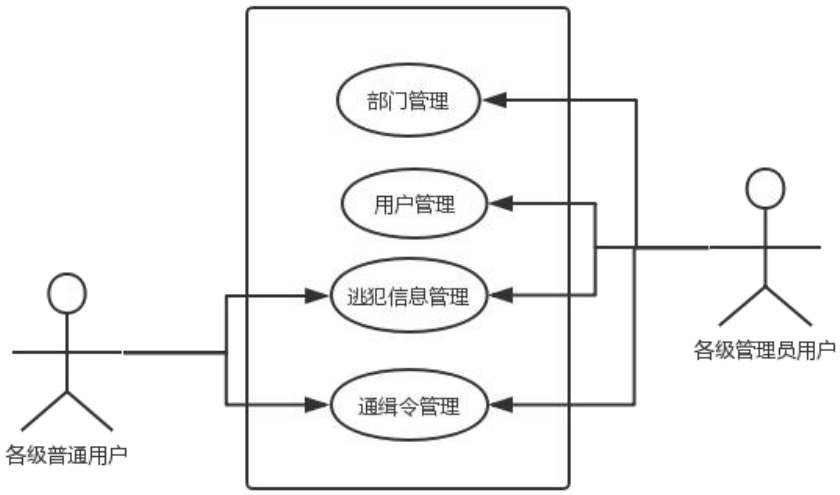


图 3.7 系统总体用例模型

3.3. 非功能需求分析

1) 易用性

系统应具有很好的易用性，在系统页面设计时应该具有足够全面和直观的功能提

醒，对于最终用户应该达到免培训即可上手使用。当系统出现故障或由于用户操作失误导致的错误，给出用户足够的提示信息，表明操作的状态。应用应该具有流畅的操作体验，在需要用户等待的场景给出相应的状态展示，页面流转要顺畅一致。

2) 实时性

系统提供逃犯信息的发布和搜索功能应该具有较强的实时性，逃犯信息发布后在最短时间内出现在目标通缉区域公安机关的关注列表中，以使公安部门及时的了解当前需要关注的嫌疑人员。

3) 安全性

本系统不论将来是否做 IP 访问限制，都应该具有保护数据安全和功能访问限制的能力。仅确定的公安系统用户可以访问系统提供的功能，面向公网开放时为了给公安用户提供随时随的访问的快捷性，但是要保证在公网复杂的网络威胁中系统的数据安全。

4) 可扩展性

系统需要具有很强的可扩展性以应对将来不断变化和增加的需求。未来需要添加新功能模块时可以很方便的与现有系统进行集成。

5) 并发性

系统面向的用户时全国范围的公安机关用户，具有一定的并发访问需求，系统应该可以支撑日常的万/秒级的访问请求，当访问量不断增多时，应该支持灵活的服务伸缩配置。在不影响系统使用的情况下增加或减少服务机器数量。

3.4. 小结

本章从多个方面对逃犯查询系统的项目开发可行性进行了分析，获取了用户最基本需求，给出了系统需要处理的数据流，叙述了逃犯查询系统的功能需求和非功能需求。本章可作为系统分析与设计阶段的参考文档，系统功能设计与开发需要满足本章所述的系统特性。

第 4 章　系统设计

4.1. 设计原则和目标

本系统的设计首先要可以满足需求分析中的功能需求，其次要在架构设计时考虑非功能需求中的易用性、可扩展性等需求。系统架构设计直接影响了未来对系统进行维护和修改时难易程度，在系统架构时考虑模块之间的高内聚、低耦合设计。

4.2. 总体架构设计

4.2.1. 架构设计

逃犯查询系统本质上是一个信息管理系统，要支持全国各地公安机关用户使用同一套系统最便捷的方式是开发为 web 应用。系统总体上拆分为两个微服务：一个用于全局用户身份校验的单点登录系统，和一个主要的逃犯信息管理系统；微服务之间具耦合程度很低，可以单独开发和部署。其中将用户校验模块单独拆出一个服务是为了满足可扩展性的需求，未来开发新的功能特性时可以直接使用已有的身份校验功能。系统总体架构如图 4.1 所示

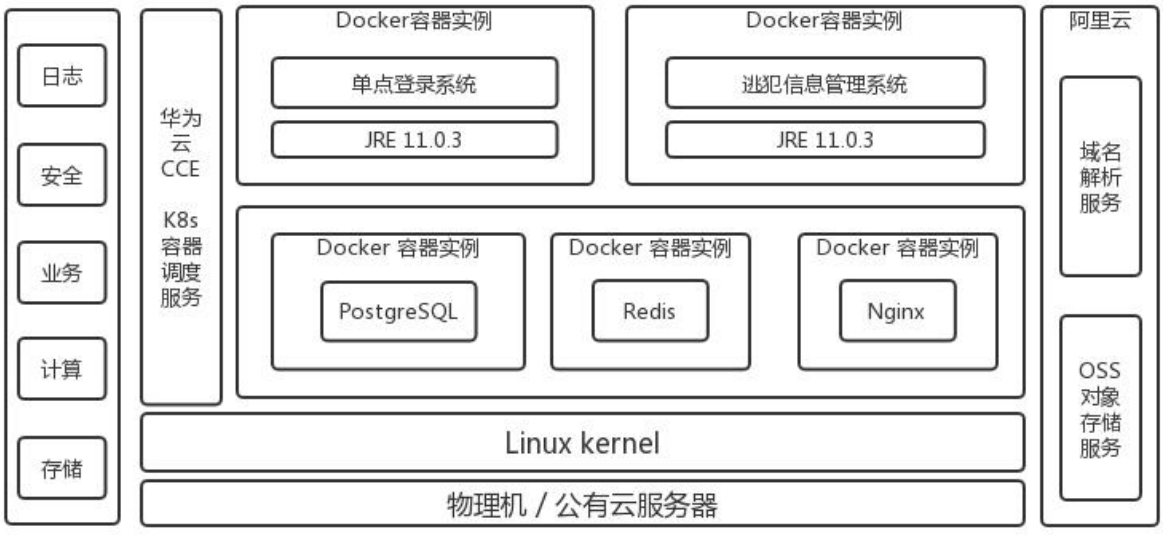


图 4.1 系统总体架构图

4.2.2. 软件技术

1） 微服务

微服务架构是软件一种架构风格，这种风格可以描述为"一种使用小型服务构建系统的架构方法，每个服务都在自己的进程中，它们通过轻量型协议进行通信"。每个服

务都是独立开发和部署，每个微服务专注一个自己擅长的小任务。[11]

微服务具有技术中立、松散耦合的特点，每个微服务的开发团队可以使用自己熟悉的编程语言，仅需确保各个服务之间可以通过约定的通信协议进行通信即可。

2） 单点登录

单点登录（Single Sign On ），用户仅需要登录一次，就可以使用系统的全部服务，即使这些服务并没有部署在同一台服务器甚至不再同一个地点。单点登录系统为系统内所有受信任的服务提供用户身份校验功能，其他服务只需要关注本身的业务逻辑，只需要很少的开发就可以完成与单点登录系统的整合。

3） REST API

REST 即 Representational State Transfer 表征状态转移，是一种软件架构风格。对于 Web 服务发开来说，RESTful Web 服务推荐关注资源状态的流转，每一个 URI 对应一个资源，充分利用 HTTP 协议的方法（GET、POST、PUT、DELETE、PATCH 等）来表示对资源的操作。每个资源对象公开一个 URI 表示自己的地址，然后在对此地址进行相应的操作。比如使用 POST 方法新建一个对象，使用 GET 方法获取一个对象，使用 PUT 或 PATCH 对资源进行更新，使用 DELETE 方法删除资源；这样就分别对应了信息管理系统中最常进行的增、查、改、删功能。组件通信示意如图 4.2 所示。

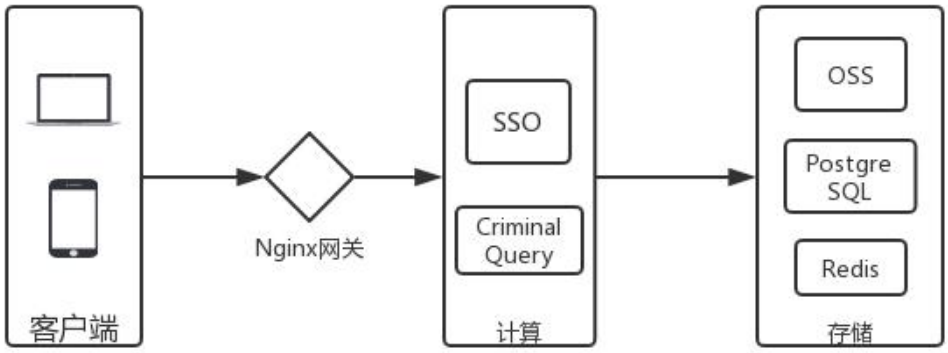


图 4.2 系统组件通信示意图

图 4.3 所示为系统活动图。

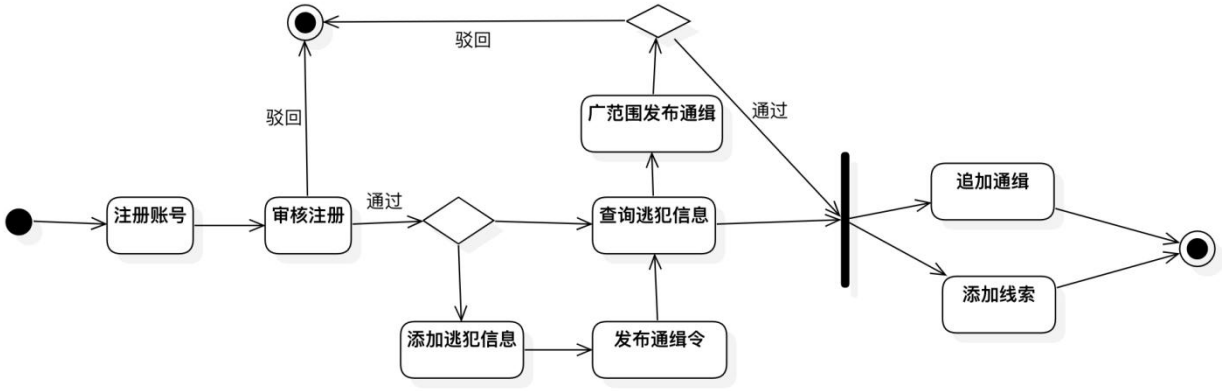


图 4.3 逃犯查询系统活动图

4.2.3. 用户角色

1) 按数据操作权限划分

普通用户：使用本系统的最普遍用户，属于公安系统内部用户，具有添加逃犯信息、添加逃犯线索、添加逃犯联系人信息等基本功能的权限，也可以编辑逃犯信息。具有查询系统内逃犯信息的权限。

管理员用户：管理员用户在每个部门中都可以存在，主要职责是审核新用户和广范围发布通缉的请求。除具有普通用户的所有权限外，还具有删除通缉令、删除逃犯信息、删除用户、新建部门、删除部门等高级权限。

2) 按用户关系划分

地市级公安机关用户：这个概念是从管理范围来讲，所属部门的管辖的行政区域为地市级公安单位。此级别的用户可以在本区域内发布通缉令，可以查看通缉范围大于等于本区域的所有通缉令以及相关逃犯信息。

省公安厅级用户：此类用户所属部门的管辖区域为省级，管理员具有审核下级所属部门发起的广范围通缉请求的权限。

公安部级用户：此类用户数量应该很少，此级别用户的所属部门“公安部”为系统内置，系统应内置一个此级别的管理员用户，相当于超级管理员。具有操作系统内所有数据和所有功能的权限。

4.3. 模块及功能

系统总体分为两个独立的微服务：单点登录服务、逃犯查询服务。单点登录服务完成用户登录、登出和 Token 校验的功能。 逃犯查询系统作为系统主要要提供服务的模块，主要包括实现逃犯信息的增删查改、公安部门的增删查改、用户管理功能（注册、审批注册、删除）。

4.3.1. 单点登录模块

单点登录作为系统中最重要模块之一，采用微服务的方式设计，主要提供用户单点登录功能和内部服务 API 保护功能。以下是本模块功能列表：

1） 登录：

客户端要访问系统的功能，首先要与单点登录服务进行身份校验，校验方式使用传统的用户名和密码方式。用户名和密码校验通过后生成一个**全局唯一且不可枚举**的加密的 Token，此 Token 作为当前登录用户在当前客户端上的令牌，用于随后进行 API 访问权限校验。登录成功后向客户端返回用户信息和 Token，随后客户端访问系统内其他服务有权限控制的 API 时需要携带此 Token 信息。登录流程如图 4.4 所示

2） Token 校验：

系统内的其他服务如果需要对用户的操作进行鉴权，首先应该读取用户请求中包含的 Token 信息，然后向单点登录系统询问此 Token 是否有效；此时单点登录系统对 Token 进行校验，若 Token 对应了一个用户登录状态，则向其他服务返回此 Token 对应的用户信息，若 Token 无效则通知其他系统此 Token 无效；是否继续提供服务由各微服务自己决定。Token 校验过程如图 4.5 所示。

3） 登出：

提供一个登出接口，由用户主动使 Token 失效以保证账号安全。

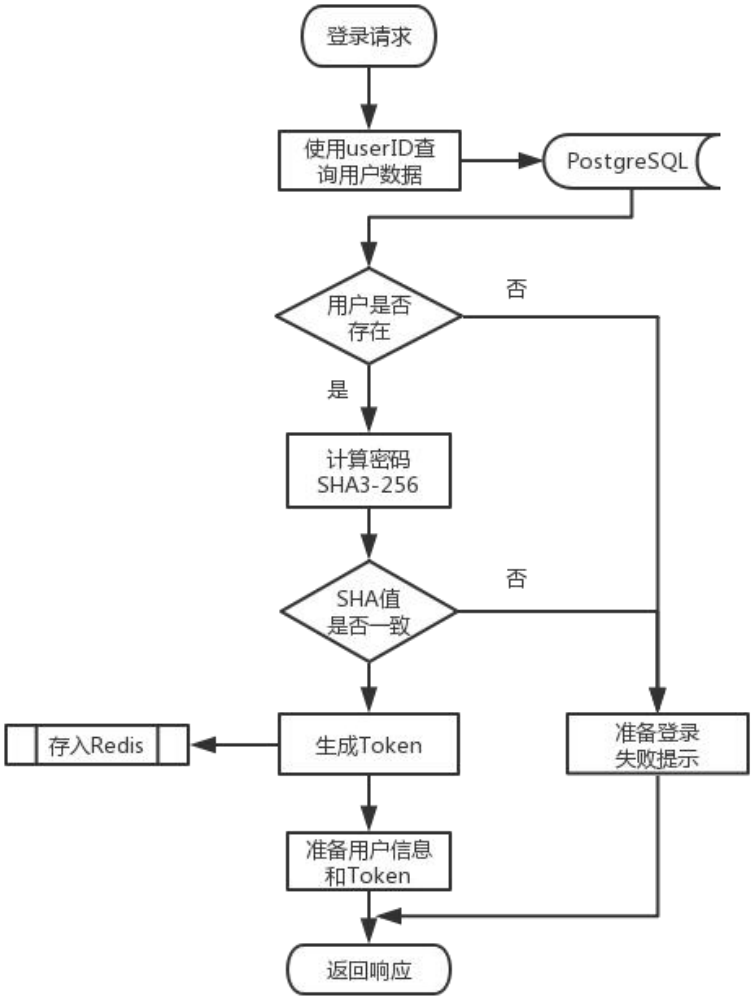


图 4.4 登录过程流程图

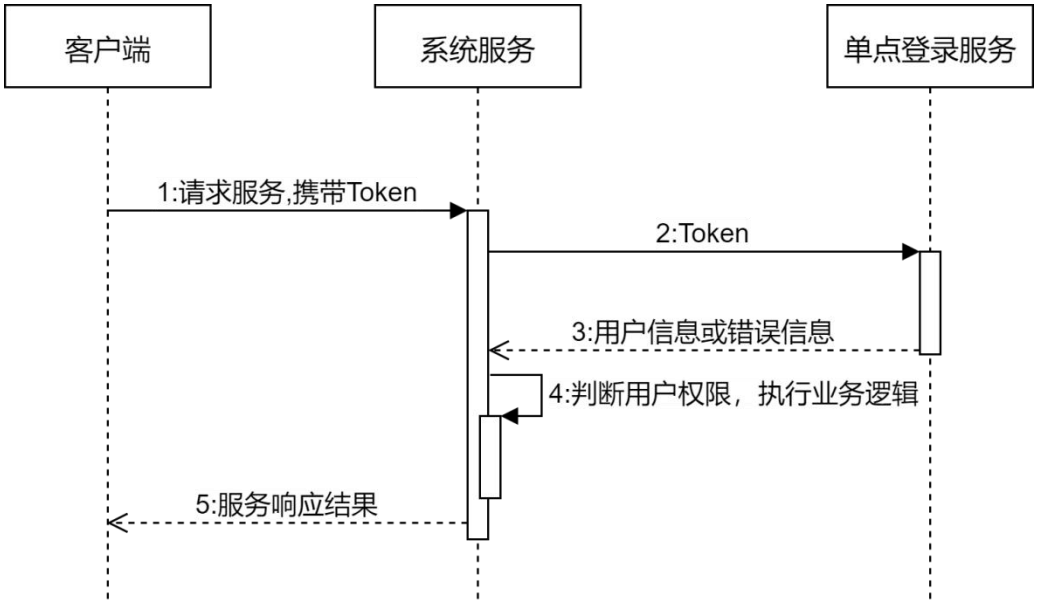


图 4.5 Token 校验时序图

4.3.2. 用户管理模块：

此模块在 model 层定义 User 类，对应数据库中的用户表；在 Service 层定义 UserService 类，其中方法包括 getAllUsers()获取用户列表、activateUser()激活用户、deleteUser()删除用户；在 Controller 层定义 UserController 类并在其中注册对应功能的请求处理方法，在 Controller 中调用 Service 层具体的业务处理方法来完成接口功能。如下是本模块功能列表：

1） 用户注册：

不需要权限校验，不需要登录态。用户填写登录用户名、密码、校验密码、邮箱、电话、真实姓名、申请所属部门、申请角色（管理员/普通用户）等信息，发起创建用户请求。用户管理模块校验用户登录名的唯一性，通过后将用户数据插入用户表，账号状态置为未激活，等待具有权限的管理员进行下一步操作（驳回或通过）。

2） 列出用户列表：

需要管理员权限。用于在用户管理页面查看所有可用的账户。为应对用户账号较多的时候管理功能执行困难，增加一个搜索框，可以按照用户姓名或用户登录名进行模糊搜索。

3） 删除用户：

需要管理员权限。列出用户列表之后可以对账号进行删除操作，敏感操作需要管理员权限。

4.3.3. 部门管理模块：

部门管理模块需要在 model 层定义 Department 类对应数据库中的部门表；在 Service 层定义 DepartmentService 类处理业务逻辑，其中包括 addDepartment()方法用于创建一个部门，deleteDepartment()方法用于删除部门，allDepartments()方法用于获取所有部门列表；另外提供一个 allDepartmentsForTree()方法用于获取树形结构的部门列表供级联选择器使用。在 Controller 层定义 DepartmentController 类来处理用户请求，并调用 Service 层方法完成业务功能。以下是本模块的功能列表：

1） 添加新部门：

仅限管理员访问。用户输入部门名称，选择部门管辖区域，设置上级部门后提交创建请求。收到请求后首先向 SSO 验证 Token，通过后判断用户是否管理员。最后向数据库加入新部门数据并返回结果。

2） 查看部门列表：

系统列出已有部门的列表，并显示每个部门的成员数量。

3） 删除部门：

部门管理的基本功能，仅限管理员访问。鉴权过程同添加过程一样。

4.3.4. 逃犯信息管理模块：

此模块在 model 层定义逃犯类 Criminal、 通缉令类 WantedOrder 和线索类 Clue；分别对应数据库在那个相应的数据表。在 Service 层定义 CriminalService 类，其中方法包括 searchCriminals()组合条件模糊搜索逃犯、getCriminalDetail()获取一个具体逃犯的详细信息、createCriminal()添加逃犯信息、createWantedOrder()添加通缉令等。在 Controller 控制器层定义与 Service 层相对应的方法，并在控制器层处理数据校验问题。本模块的功能如下：

1） 发起通缉/追加通缉：

在通缉逃犯前需要对逃犯基本信息进行录入，随后可以对录入的逃犯发起通缉；对已添加进系统的逃犯支持进行追加通缉操作，逃犯与通缉令是一对多关系。

2） 修改通缉令状态：

通缉令具有“正在通缉”和“停止通缉”两个状态，修改通缉令状态功能应该可以切换某条通缉令的状态。

3） 搜索通缉令：

逃犯查询系统最基础功能，支持使用逃犯基本信息中的大部分条目以及通缉令中的大部分条目任意进行组合搜索条件进行组合搜索。默认无条件，显示所有结果，支持分页查询。搜索的结果用于展示一个包含逃犯概要信息的表格，若要查看某个逃犯的详细信息（比如线索、联系人、通缉令等），应使用另一个专用的接口。

4） 添加线索：

对于已经存在系统中的逃犯增加线索信息，线索支持上传图片信息和文本描述信息。上传图片后由前端处理图片文件 fileID，随后正式提交线索记录时一并将 fileID 提交并由后端记录在数据库。推荐使用公有云计算服务商提供的对象存储服务。

5） 添加联系人信息：

需要登录态，不需要管理员权限。对已存在系统中的逃犯添加相关联系人信息，逃犯于联系人信息为一对多关系，可以添加多条。

6） 获取逃犯详细信息：

需要登录态，不需要管理员权限。用于获取某个逃犯的所有相关信息，以展示一个关于单个逃犯的聚合页面。返回逃犯的基本信息、此逃犯的所有通缉令信息、此逃犯的所有联系人信息、此逃犯的所有线索信息。后端程序通过 SQL 连接查询或多次查询准备所有信息一次性返回。

4.4. 数据库设计

4.4.1. 数据库设计思路：

部门管理模块中，每个部门都对应一个管辖的行政区域，每个行政区域可对应多个部门，部门与行政区域是多对一关系。

用户管理模块中，用户与部门之间是一对多关系；部门又有上下级的分别，所以部门与自身是一对多关系，同时每个部门管辖一个行政区域，与行政区域是一对一关系。

逃犯信息模块中，逃犯基本信息语附加线索、相关联系人均是一对多关系，但是允许对应 0 条线索或联系人。逃犯与通缉令为一对多关系，以实现多个公安部门对同一个逃犯发起通缉。

4.4.2. ER 图

1） 逃犯基本信息实体如图 4.6 所示

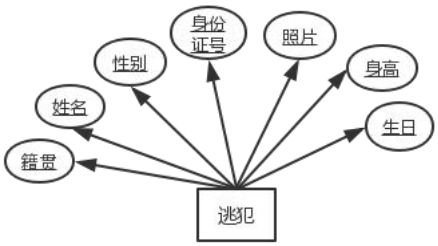


图 4.6 逃犯基本信息 ER 图

2） 逃犯联系人实体如图 4.7 所示

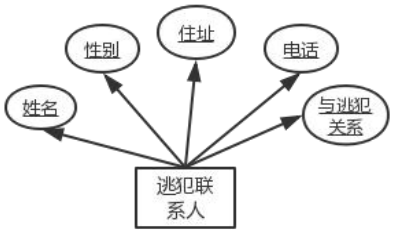


图 4.7 逃犯联系人信息 ER 图

3） 通缉令实体如图 4.8 所示

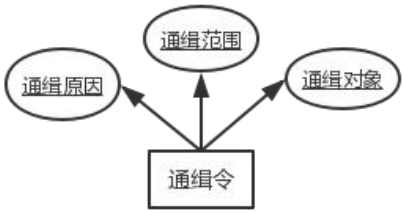


图 4.8 通缉令 ER 图

4） 逃犯线索实体如图 4.9 所示

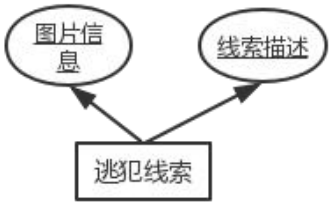


图 4.9 逃犯线索 ER 图

5） 系统用户实体如图 4.10 所示

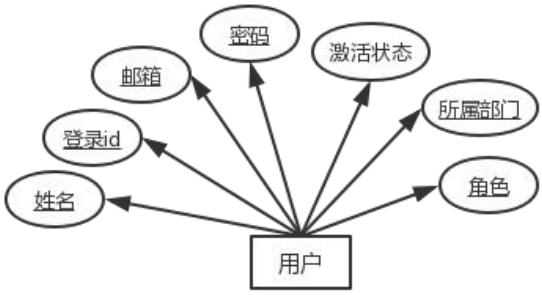


图 4.10 系统用户 ER 图

6） 行政区划实体如图 4.11 所示

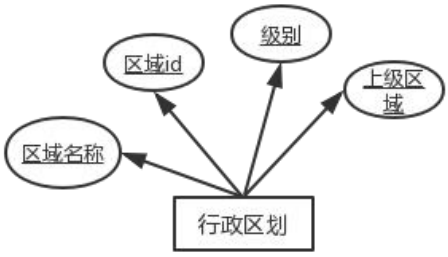


图 4.11 行政区划 ER 图

7） 部门实体如图 4.12 所示

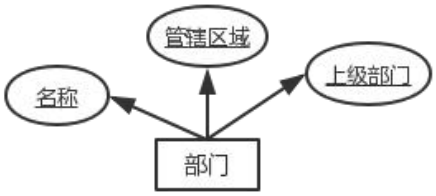


图 4.12 部门 ER 图

8） 总体实体关系如图 4.13 和图 4.14 所示

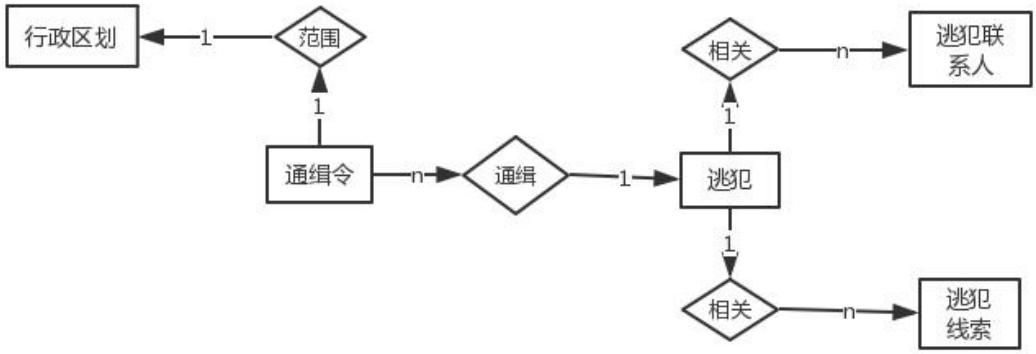


图 4.13 总体 ER 图-逃犯相关

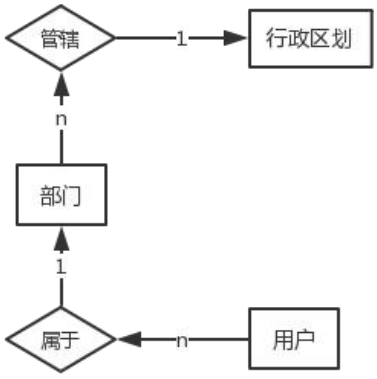


图 4.14 总体 ER 图-用户相关

4.4.3. 逻辑结构设计

本系统使用的数据库逻辑结构设计如下：

1.行政区划 (区域 ID, 区域名称, 区域级别, 上级区域 ID, 区域层级路径)。

2.部门 (部门 ID, 部门名称, 管辖区域 ID, 上级部门 ID)

管辖区域 ID、上级部门 ID 为外键。

3.用户 (用户 ID, 用户登录 ID, 密码, 邮箱, 电话, 姓名, 所属部门 ID, 角色, 激活状态)

其中用户 ID 是自增主键，用户登录 ID 具有唯一索引，所属部门 ID 是外键。

4.逃犯 (逃犯 ID, 逃犯姓名, 性别, 生日, 籍贯, 身高, 身份证号, 照片)。

5.逃犯联系人 (联系人 ID, 姓名, 性别, 电话, 住址, 与逃犯关系)。

6.逃犯线索 (线索 ID, 相关照片, 线索描述)。

7.通缉令(通缉令 ID, 逃犯 ID, 通缉原因, 通缉区域 ID, 通缉级别)

其中逃犯 ID 和通缉区域 ID 为外键。

4.4.4. 物理结构设计

数据库物理结构设计确定具体的数据表和属性名称以及属性数据类型，以下是本系统中使用的数据库表设计。行政区划表如表 4.1 所示，部门表如表 4.2 所示，用户表如表 4.3 所示，逃犯表如表 4.4 所示，逃犯联系人表如表 4.5 所示，逃犯线索表如表 4.6 所示，通缉令表如表 4.7 所示。

表 4.1 行政区划表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，区域 ID
I_SUPER_ID	INTEGER	NOT NULL, DEFAULT 0	上级区域 ID
IA_PATH_ROOT	INT ARRAY	NOT NULL	层级 ID 数组
CH_NAME	VARCHAR	NOT NULL	区域名称
I_LEVEL	INTEGER	NOT NULL	区划层级

表 4.2 部门表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，部门 ID
CH_DEPARTMENT_NAME	VARCHAR	NOT NULL	部门名称
I_SUPERVISOR_ID	INTEGER	NOT NULL, DEFAULT 0	上级部门 ID
I_DISTRICT_ID	INTEGER	NOT NULL	管辖区域 ID

表 4.3 用户表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，用户 ID
CH_USER_ID	VARCHAR(64)	UNIQUE, NOT NULL	用户登录 ID
CH_PASSWORD_SHA256	VARCHAR(128)	NOT NULL	用户密码哈希值

CH_REALNAME	VARCHAR	NOT NULL	用户姓名
I_ROLE	INTEGER	NOT NULL	用户角色:0-普通用户，1-管理员
I_DEPARTMENT_ID	INTEGER	NOT NULL	所属部门 ID
CH_EMAIL	VARCHAR	NOT NULL	用户邮箱
CH_PHONE	VARCHAR	NOT NULL	用户电话
B_ACTIVATED	BOOLEAN	NOT NULL, DEFAULT 0	账号激活状态

表 4.4 逃犯表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，逃犯 ID
CH_NAME	VARCHAR	NOT NULL	逃犯姓名
I_SEX	INTEGER	NOT NULL	逃犯性别, 0-女, 1-男
I__HEIGHT_CM	INTEGER		逃犯身高
D_BIRTHDAY	DATE		逃犯生日, 计算年龄
CH_BORN_PLACE	VARCHAR		逃犯籍贯
CH_IDCARD_ID	VARCHAR	UNIQUE	逃犯身份证号
CH_PORTRAIT_FILEID	VARCHAR		逃犯照片文件 ID
CH_OTHER_FEATURES	TEXT		其他特征

表 4.5 逃犯联系人表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，联系人 ID
I_CRIMINAL_ID	INTEGER	NOT NULL	关联逃犯 ID
CH_NAME	VARCHAR	NOT NULL	联系人姓名
CH_RELATION	VARCHAR	NOT NULL	与逃犯关系
CH_PHONE	VARCHAR		联系电话
CH_ADDRESS	VARCHAR		住址

表 4.6 逃犯线索表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，线索 ID
I_CRIMINAL_ID	INTEGER	NOT NULL	关联逃犯 ID
CH_FILEIDS	VARCHAR		相关照片文件 ID, 多个文件逗号分隔
CH_DESCRIPTION	VARCHAR	NOT NULL	线索描述

表 4.7 通缉令表

字段名	数据类型	约束	备注
I_ID	INTEGER	PK, AUTO INCREMENT	自增主键，通缉令 ID
I_CRIMINAL_ID	INTEGER	NOT NULL	关联逃犯 ID
I_ARREST_LEVEL	INTEGER	NOT NULL	通缉级别

CH_ARREST_REASON	VARCHAR	NOT NULL	通缉原因
I_ARREST_STATUS	INTEGER	NOT NULL	通缉状态
I_DISTRICT_ID	INTEGER	NOT NULL	通缉范围行政区域 ID

4.5. 小结

本章系统概要设计文档指定了系统的具体数据存储标准，确定了各模块需要实现的每个功能，同时规定了应用程序的数据交互方式和协议并给出了应用部署的总体结构图。在后续系统实现阶段可参考本文档给出的功能定义来开发对应每个功能操作的 API 和页面，在系统部署阶段参考本文档来完成预想的系统部署。

第 5 章　系统实现

5.1. 概述

逃犯查询系统详细设计说明书将详细描述概要设计中指定的所有已确定设计方案的具体实现；本说明书时本系统实际开发人员编写的主要开发文档。

逃犯查询系统详细设计读者对象为所以与本系统开发工作直接相关的人员（包括所有开发小组成员、系统实施与部署人员、未来对本系统进行修改和维护的人员）以及需要关注本系统的开发与实施过程的所有人员。

本文档将按照与软件架构方式相似的结构进行组织;基本单位是每一个功能的实现方案（包括功能/目标、输入输出标准、接口、关键算法描述、程序逻辑、关键数据库操作），描述的顺序先给出后端实现细节，最后给出前端实现细节。

5.2. 开发环境和架构配置

5.2.1. 接口规范约定

除特别说明外，系统所有接口返回数据都是 JSON 格式，且具有统一的结构样式，包括状态、返回数据、提示信息三个数据域。下文中若无特别说明，返回数据都是本模版的 data 数据域的内容。

全局响应数据示例

```
{
    "status": 200,
    "data": {} | [],
    "msg": "成功"
}
```

状态码定义

200：接口工作正常

400：用户原因导致错误

500：服务端异常导致错误

5.2.2. 后端技术说明

后端系统使用 Maven 做项目模块和依赖关系管理，首先将系统内的数据模型全部提出到一个单独的包中，独立成为一个 Maven 模块，所有其他服务都引用此模块，以此来保证各种 POJO 在各微服务和模块之间都是一致的。

另外将单点登录系统和逃犯查询系统两个微服务作为 Maven 模块，分别开发；两个微服务都使用 SpringBoot 进行开发，最终每个微服务构建之后都打包为一个可以直接运行的 jar 程序，可以部署在不同的服务器上分别运行。

1) SpringMVC 开发 RESTful API 说明

SpringMVC 最经典的用法是遵循 Model-View-Controller 的方式来在后端将数据直接渲染净 html 模版引擎，开发 RESTful API 的最明显差别就是不需要在后端进行 html 页面的数据渲染。SpringMVC 提供了几个注解来直接将 Controller 中的方法返回值转换为响应体，而不是用返回值代表视图名称。

3） SpringBoot 配置

使用 SpringBoot 来管理 SSM 框架后大部分的配置都集中在 application.yml 中，其中包括数据库配置、服务端口配置、MyBatis 映射文件地址配置等等。以下是逃犯查询系统的配置样例。

```
server:
  port: 8082   # 服务端口
spring:
  datasource: #  数据源配置
    url: jdbc:postgresql://postgres:5432/postgres
    username: postgres
    password: "*6Z,<[9A97Mp;6W"
    driver-class-name: org.postgresql.Driver
    hikari: #  数据库连接池配置
      minimum-idle: 2
mybatis:   # mybatis 配置
  mapper-locations:
    - classpath*:mapper/*Mapper.xml
  config-location:
    - "classpath*:mybatis-config.xml"
```

4） Nginx 配置

系统后端包括了两个服务，分别运行在 8081 和 8082 端口上，所以使用 Nginx 作反向代理，将浏览器对同一个域名不同 URL 的访问分配到不同的端口上，使 Nginx 充当一个 API 网关的角色；同时通过对 Nginx 配置 SSL 来实现数据的加密传输。

```
server {
    server_name grad.linkinghack.com; #  监听的主机域名
    location /{   # 根 URL 返回编译好的前端资源
        root /criminalquery/web;
```

```
        index index.html index.htm;
    }

    location /sso {   # 跳转至单点登录服务
        proxy_pass http://sso:8081/;
    }

    location /api {   # 跳转至逃犯查询服务
        proxy_pass http://base:8082/;
    }
 # 开启 SSL 配置
    listen 443 ssl;
    ssl_certificate /criminalquery/ssl/grad.linkinghack.com.pem;
    ssl_certificate_key /criminalquery/ssl/grad.linkinghack.com.key;
    ssl_session_timeout 5m;
    ssl_ciphers
ECDHE-RSA-AES128-GCM-SHA256:ECDHE:ECDH:AES:HIGH:!NULL:!aNULL:!MD5:!ADH:!RC4;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_prefer_server_ciphers on;
}

# 针对访问 80 端口的非安全请求进行跳转
server {
    if ($host = grad.linkinghack.com) {
        return 301 https://$host$request_uri;
    }
    listen 80;
    server_name grad.linkinghack.com;
    return 404;
}
```

5.2.3. 前端技术说明

1） 前端 MVVM 框架--Vue

前端页面不是由后端程序组装渲染后返回到浏览器，而是最初访问系统是一并加载完成。本系统使用 Vue 框架来管理前端程序。

2） Vue-Router 页面路由

使用 Vue 框架后不会出现全页面重新渲染的情况，但是为用户提供的功能依然是分开几个页面的，前端页面的跳转需要一个路由引擎，本系统使用 Vue-Router 解决单页 Vue 应用的页面跳转问题。

3） 异步请求

在本系统中使用开源第三方 HTTP Client 工具 Axios 发起异步请求并处理响应数据。所有的用户操作都是如此完成与后端程序通信。

4） 页面组件

本项目使用阿里巴巴开源的 Ant Design 中的包括按钮、表格、网格等各种 UI 组件来构造基本用户页面。

5） 前端程序主体结构

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>criminalquery-app</title>
  </head>
  <body>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

在此页面结构中，具有 id=“app”属性的 标签就是前端应用程序要渲染的位置，由此也可看出单页应用几乎所有内容都是动态生成的。

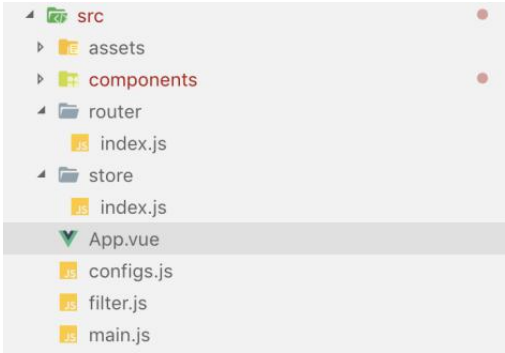


图 5.1 前端项目结构

图中 components 目录下存放的就是 Vue 组件，页面上的各种元素都可以单独开发成一个小组件，以重复使用。

使用全局唯一的一个 Vue 对象将对应到页面上的一个 id 为“app”的 div 元素，所有使用 Vue 框架开发的 UI 组件和业务逻辑都在程序运行起来之后动态生成。前端的多

个页面、各种 UI 元素都作为根 Vue 对象的组件来动态加载。

```
new Vue({
  el: '#app',
  router,
  store,
  components: { App },
  template: '<App/>',
})
```

图 5.2 全局唯一 Vue 对象实例

5.2.4. 系统类结构

系统功能开发类设计自底向上划分为 DAO 层、Service 层和 Controller 层，通过依赖注入将下层实现类提供给上层使用。

DAO（即 Data Access Object）层完成数据持久化相关的任务，包括对数据库进行增、删、改、查操作。

业务逻辑主要是在 Service 层完成的，通过 DAO 提供的数据工具以及需要使用的其他第三方工具库来完成指定的功能，在这一层的数据输入来自 Controller 层，输出又返回到 Controller 层。

Controller 层主要进行 http 请求的处理和请求参数数据校验，在 Controller 层验证数据基本的长度、格式等属性后将数据传递到 Service 层完成相应功能，最后将 Service 的处理结果再返回到客户端；在 SpringMVC 中 URL 绑定也在这一层完成。

5.3. 功能模块实现

5.3.1. 单点登录模块

单点登陆模块要完成用户登录和 Token 校验两大主要功能，另外提供登出功能主动使 Token 失效。

登录功能的实现包括前端提供一个登录页面和相关的表单，后端提供登录接口；在登录过程中通过用户登录 ID 查找用户信息，若找到则检查用户激活状态，激活的用户才允许登录；然后对请求中的密码计算 SHA3-256 哈希值，最后与数据库中保存的密码哈希值做比对。登录通过后计算一个全局唯一的 Token 作为会话凭证返回给浏览器，由前端程序负责保存，并在以后的请求中携带 Token。

单点登录模块为其他模块提供用户身份校验服务，在登录阶段后端程序将 Token 和用户信息保存在 Redis 缓存数据库中，在用户身份校验时以客户端携带的 Token 为键

在 Redis 中获取用户信息，能获取到说明 Token 有效即身份校验通过，反之亦然。以下代码展示单点登录模块中提供的 Token 校验逻辑。

```
    public User auth(String token) {
        UserStatus userStatus = userStatusFromJson(redis.opsForValue().get(token));
        if (userStatus == null) { // token 无效或登录过期
            return null;
        }
        // 刷新 redis 可用时间
        if (userStatus.getRemember())
            redis.opsForValue().set(token,userStatusToJson(userStatus),
Duration.ofMinutes(rememberTime));
        else
            redis.opsForValue().set(token,userStatusToJson(userStatus), Duration.ofMinutes(loginTime));
        return userStatus.getUser();
}
```

以下代码展示了其他服务在 SpringMVC 的拦截器中通过访问单点登录模块进行用户身份校验的方式。

```
// 向 SSO 单点登录系统询问 Token 是否有效
            User user = restTemplate.getForObject("https://grad.linkinghack.com/sso/auth/{token}",
User.class, token);
            logger.info("User from sso {}", user);
            if (user == null) {
                logger.info("API request rejected because of unknown token: token={}", token);
                response.setStatus(405);
                return false;
            }
            request.setAttribute("user", user);
                return true;
```

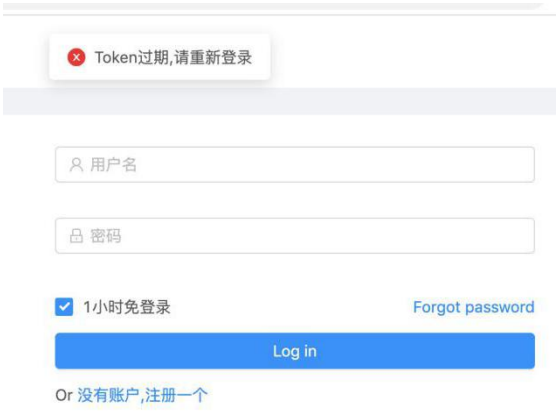


图 5.3 登录页面局部和 Token 过期提示

5.3.2. 用户管理模块

用户管理模块要完成用户注册、查看用户列表、删除用户、和审核用户注册四个功能；主要对用户表进行增删该查。模块提供注册页面、查看用户列表页面和审批用户注册页面。

用户注册时首先验证用户登录 ID 的唯一性，通过后计算密码的 SHA3-256 值后插入数据库用户表，如此可保护用户密码的安全；在插入新用户记录时默认将用户激活状态置为 0-未激活，此时新注册用户是无法登录系统的。

审核用户注册功能提供给管理员核对新加入系统的用户身份的机会，由管理员确认用户信息后选择通过或驳回注册；审核通过时将此用户的激活状态置为 1-激活。

删除用户功能提供在用户列表页面上，管理员可以对激活状态的用户进行删除操作，删除用户时传递给后端待删除用户的 id。

以下是用户注册的核心代码：

```
public UniversalResponse register(User user) {
    user.setPassword(DigestUtils.sha3_256Hex(user.getPassword())); // hash password
    try {
        int lineAffected = userMapper.createUser(user);
        if (lineAffected == 1)
            return UniversalResponse.Ok("注册成功,在审批通过前您无法登录。");
        else {
            throw new RegisterFailedException("插入新用户行数为 0");
        }
    } catch (RegisterFailedException e) {
        return UniversalResponse.ServerFail(e.getMessage());
    } catch (Exception e) {
        return UniversalResponse.ServerFail("未知错误");
    }
}
```

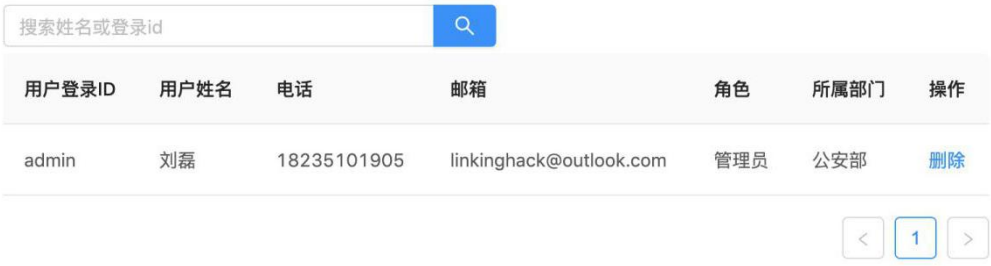


图 5.4 查看用户列表页面

图 5.5 审批用户注册页面

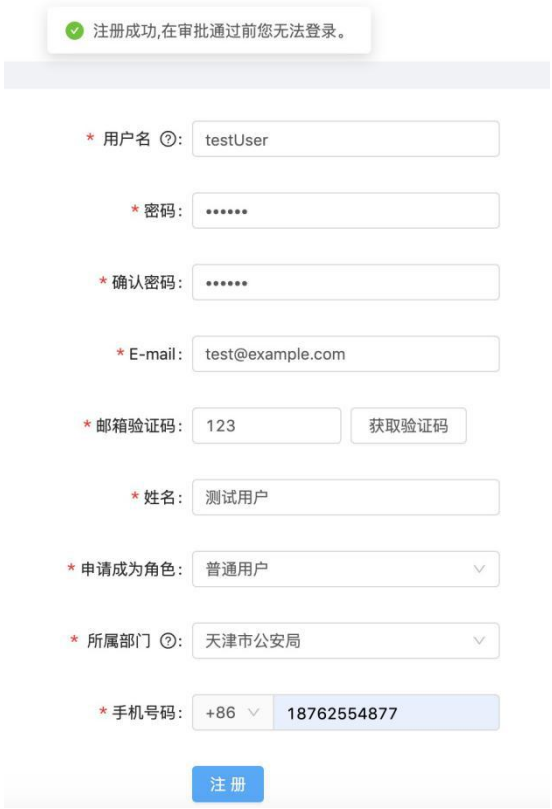


图 5.6 用户注册页面和注册提示

5.3.3. 部门管理模块

部门管理模块提供查看部门列表、增加部门、删除部门三个功能，完成对部门表的增删查操作。

模块提供一个部门管理页面，和一个新建部门的模态框，在不刷新页面的前提下完成部门新建操作；在部门管理页面列出所有部门支持分页，另外提供一个行政区域选择器来筛选需要显示的部门。

在新建部门页面中同样提供一个行政区域选择器，用于设置新部门的管辖区域；此行政区划选择器作为一个 Vue 组件可以进行复用。

以下是按管辖区域显示部门列表的关键 SQL 代码，通过连接行政区划表和部门表，在页面上显示部门的管辖区域名称。

```
<select id="departmentsOfDistrict" resultMap="department" >
    select de.i_id as did, ch_department_name, de.i_supervisor_id as i_supervisor_id, de.i_level as i_level,i_district_id, di.ch_name as district_name
    from b_departments as de
        left join b_districts as di on de.i_district_id = di.i_id
    where  #{districtID} =  ANY(di.ia_path_root)
    or de.i_district_id = #{districtID}
    order by de.i_id
</select>
```

图 5.7 部门管理页面-按区域过滤部门

以下是删除部门操作的核心代码，其中包括了对管理员角色的判断：

```
public UniversalResponse deleteDepartment(Integer departmentID, HttpServletRequest request) {
    User user = (User) request.getAttribute("user");
    Department department = departmentMapper.getDepartmentByID(departmentID);
    if (user != null && user.getRole().equals(Constants.UserRoleManeger) ) {
        int rows = departmentMapper.deleteDepartment(departmentID);
        if (rows == 1){
            logger.warn("用户{}{} 成功删除部门数据: {}", user.getUserID(), user.getRealName(), department);
            return UniversalResponse.Ok("删除成功");
        }
        else {
            logger.warn("用户{}{} 尝试删除部门数据失败: {}", user.getUserID(), user.getRealName(), department);
            return UniversalResponse.UserFail("删除失败");
        }
    }else
        return UniversalResponse.UserFail("没有权限");
}
```

图 5.8 部门选择器

图 5.9 新建部门页面

5.3.4. 逃犯信息管理模块

逃犯信息管理模块主要提供逃犯信息录入、发起通缉、搜索逃犯以及添加逃犯线索等功能，涉及到逃犯表、线索表和通缉令表的增删查改。

前端提供一个发起通缉页面，在页面中展示一个发起通缉流程；第一步添加逃犯信息，第二步是填写通缉信息并发起通缉，最后可以给出一个信息总览。

另外对于最重要的搜索功能，提供一个逃犯查询页面，分为“组合搜索”和“身份证精确搜索”两个选项卡，分别完成多条件组合模糊查询和精确查询的功能。

提供逃犯信息详情页面，显示有关某一个逃犯的具体信息，包括逃犯基本个人信息、逃犯照片、线索信息和通缉令信息等。在访问此页面是首先获取到的是逃犯表中的基本逃犯信息，照片、通缉令、线索等信息采用异步加载的方式，避免了首次载入页面等待时间过长。

对于逃犯搜索功能，为支持多条件任意组合进行搜索，使用 MyBatis 的动态 SQL 功能来完成自动 SQL 拼接，针对非空的搜索参数添加相应的匹配规则，空条件将自动跳过，不会出现在最终 SQL 里。

对于添加逃犯信息和发布通缉令功能，分别提供了对应的表单来提交数据，表单的默认行为被拦截，背后发起网络请求依然是异步的，提交数据后不会导致页面的整体刷新。在添加逃犯信息和添加线索信息时允许上传图片，此操作将先完成图片上传，成功后获取到文件 id 填入表单，最后提交时照片这一项记录的仅仅是图片文件 id。此处文件 id 表示的时对象存储服务中的 ObjectID，用于唯一标识一个存储桶中的文件；关于对象存储服务在下文“其他底层功能”中介绍。

以下是组合搜索使用的 MyBatis 查询代码，其中<if>标签和 test 属性一起使用完成了动态 SQL 的自动拼接，在代码中的大于小于号分别使用了&gt 和&lt 进行表示，是为了避免在 xml 中左右尖括号和标签出现冲突。（代码中省略了一些查询条件）

```
<select id="searchCriminals" resultMap="criminalInfo"
parameterType="com.linkinghack.criminalquerymodel.transfer_model.SearchCriminalRequest">
        select bcri.i_id as i_id, ch_name, i_sex, i_height_cm, d_birthday,( CURRENT_DATE - d_birthday)/365 as age ,
        ch_born_place, ch_idcard_id, ch_other_features, ch_portrait_fileid, ch_edu_background, ch_job, ch_workfor,
        ch_phone, ch_address, bcri.ts_created_at as ts_created_at
        from b_criminals as bcri
        left join b_wanted_orders as bwant on bcri.i_id = bwant.i_criminal_id
        <where>
            <if test="name != null">ch_name like #{name}</if>
            <if test="sex != null">AND i_sex = #{sex}</if>
            <if test="heightStart != null">AND i_height_cm &gt;= #{heightStart}</if>
            <if test="heightEnd != null">AND i_height_cm &lt;= #{heightEnd}</if>
            <if test="ageStart!=null">AND ( CURRENT_DATE - d_birthday)/365 &gt;= #{ageStart}</if>
            <if test="ageEnd!=null">AND ( CURRENT_DATE - d_birthday)/365 &lt;= #{ageEnd}</if>
        </where>
        group by bcri.i_id
        order by bcri.ts_created_at desc
        <if test="pageSize != null">limit #{pageSize}</if>
        <if test="offset != null">offset #{offset}</if>
</select>
```

图 5.10 逃犯查询页面

图 5.11 发起通缉页面-第一步添加逃犯信息

图 5.12 逃犯详情页面-逃犯信息

图 5.13 逃犯详情页面-通缉令信息

5.3.5. 其他底层功能

1） 对象存储服务

本系统中支持上传图片的功能均使用了公有云提供的对象存储服务来保存图片数

据。在将文件上传至对象存储服务之前本系统后端会自动生成一个唯一的文件 ID 作为 ObjectID 来命名文件，随后将此文件 ID 返回到客户端以便完成最终的表单提交操作。在系统中记录的文件 ID 可以通过公有云提供的 SDK 获取到临时 URL，当前端页面需要显示上传的图片时即可获取到一个临时的 URL，这样既实现了图片的异步加载，又保证了图片数据的安全。

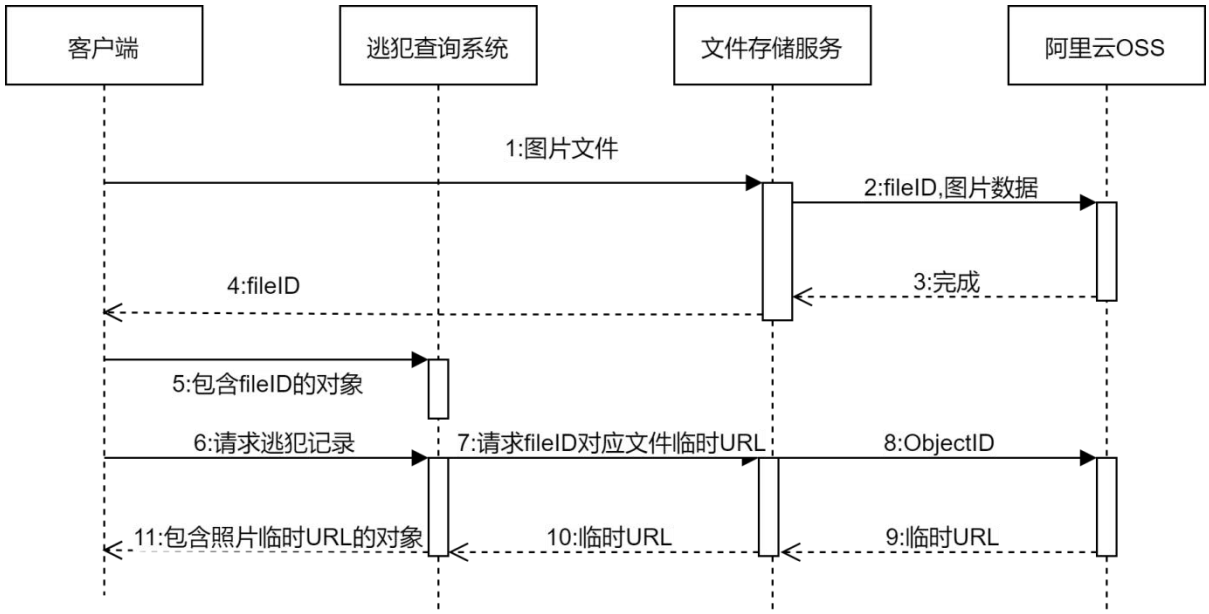


图 5.14 文件存储服务时序图

2） 构建全国行政区划数据库

本系统中多处使用到了行政区划选择器，为了方便系统根据行政区划进行一些列的信息管理，在开发系统过程中搭建了自有的一套行政区划数据，保存在行政区划表中。这些数据来自于互联网，最初得到的是一个 excel 文件，后续通过使用 golang 编写的数据导入工具将数据导入本系统数据库中供部门管理、逃犯查询等模块使用。

以下展示了数据导入工具的核心代码：

```
func InitPostgresDB(xlFile *xlsx.File) (districts []interface{}) {
    // Prepare PostgreSQL client.
    connStr := `dbname=postgres user=postgres password=*6Z,<[9A97Mp;6W sslmode=disable`
    pgdb, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal("Connot connect to PostgreSQL. err = %v", err)
    }
    txn, err := pgdb.Begin()
    if err != nil { log.Fatal(err) }
```

```
    stmt, err := txn.Prepare(pq.CopyIn("b_districts", "i_id", "i_supervisor_id", "ia_path_root", "i_level",
"ch_name", "ch_province_name", "ch_city_name", "ch_county_name"))

    // 读取 xlsx 并存入数据库
    for _, sheet := range xlFile.Sheets {
        for i, row := range sheet.Rows {
            if i == 0 { // 跳过表头
                continue
            }
            cells := row.Cells
            id, _ := strconv.ParseInt(cells[0].String(), 10, 64)
            super, _ := strconv.ParseInt(cells[1].String(), 10, 64)
            level, _ := strconv.ParseInt(cells[3].String(), 10, 64)
            path := []int64{}
            for _, v := range strings.Split(cells[2].String(), ",") {
                value, _ := strconv.ParseInt(v, 10, 64)
                path = append(path, value)
            }
            log.Println("Row data parsed: ", tmpDistrict)
            districts = append(districts, &tmpDistrict)
            _, err := stmt.Exec(tmpDistrict.ID, tmpDistrict.SupervisorID, fmt.Sprintf("{%s}",
cells[2].String()), tmpDistrict.Level, tmpDistrict.Name, tmpDistrict.Province, tmpDistrict.City,
tmpDistrict.County)
            if err != nil {
                log.Println("One row failed. err = %v", err)
            }
        }
    }
    return
}
```

图 5.15 行政区划选择器

5.4. 小结

本章系统详细设计文档是对系统需求分析和系统总体设计文档中描述的功能的具体实现方案。在软件工程中书写本文档是描述每一个功能的实现技术方案和某些关键部分的具体实现代码；详细设计文档在具体实现代码前应该首先完成功能实现的设计表达，通过团队以及负责人 review 之后才进行最后的开发实施；在开发完成之后将开发过程中进行的关键操作记录在本文档中。

本详细设计文档的作用一方面是开发过程中质量保证的重要一环，另一方面是作为未来系统维护和扩展开发时的参考手册。

第 6 章　系统测试

6.1. 测试简介

软件测试是软件工程中重要的一环，是保证软件运行的正确性、完整性、安全性和保证软件质量的过程;软件测试将在规定的条件下对程序进行操作,以发现程序的错误,衡量软件的质量，并对其能否满足设计要求进行评估。

逃犯查询系统的开发涉及到多个模块、多个微服务以及分离的客户端和服务端程序。这些不同层级的组件之间是否可以进行有效的数据通行和协作，能否完成需求文档所要求的功能，系统各个功能运行后产生的数据结果是否正确、一致等等问题都需要在本阶段进行测试。

6.2. 测试任务及目标

系统测试过程应该对逃犯查询系统的基本功能进行测试,找到功能实现的漏洞并进行修复；对系统的非功能需求方面所提出的要求进行测试验证。

对逃犯查询系统的测试任务主要分为 REST API 测试和页面逻辑测试两部分。目标是检验 API 是否可以正常工作，返回数据是否正确，以及页面显示是否与数据匹配，是否可以正常操作并完成业务逻辑。本文档给出部分关键功能的测试流程。

6.3. 测试技术方案

6.3.1. 概述

对于本系统功能的测试主要是对后端微服务的 RESTful API 的接口测试，在测试过程中主要使用黑盒测试的方式，根据功能需求说明和接口设计说明，对输入输出数据进行核对；使用的主要工具是 Postman。

对于本系统前端页面逻辑的测试通过测试用例中描述的内容进行操作,查看是否符合测试用例描述，检验各种功能是否正常。

6.3.2. 使用 Postman 测试 RESTful API

Postman 是一个 RESTful API 测试工具，主要作用是定制化 HTTP 请求，模拟客户端的请求操作，并记录 http 请求和响应的所有细节，可以精确的控制请求方法、请求头、请求体等内容。

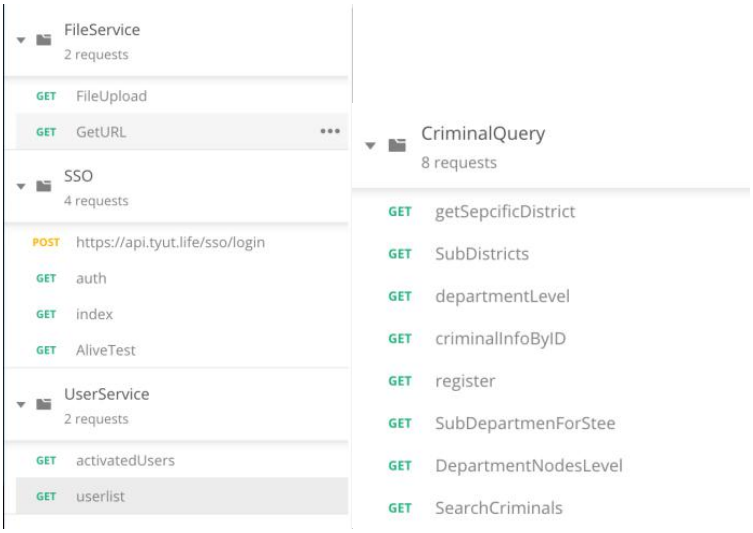


图 6.1 Postman 测试用例列表

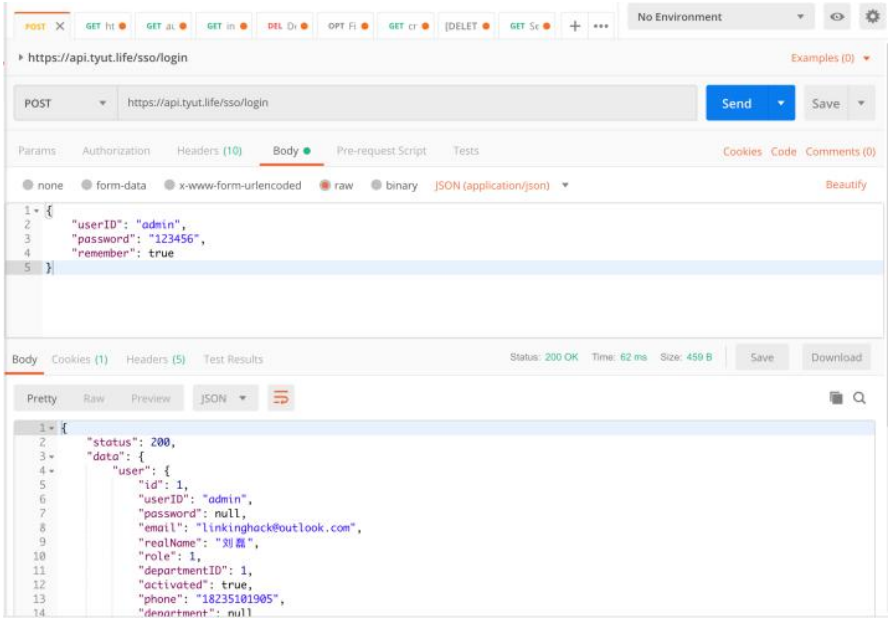


图 6.2 Postman 测试示例

6.4. 测试用例

- 测试 1：

  名称：注册

  目的：用户注册功能是否可用，用户登录 ID 是否保证唯一性，用户注册后是否直接置为冻结状态，用户密码存入数据库前是否已加密。

  内容：使用全新的 ID 尝试进行用户注册，随后尝试登录，查看是否禁止登录；使

用一个已知存在的用户登录 ID 尝试进行注册操作，查看是否拒绝注册；通过查验数据库来查看用户密码是否加密。

结果：系统可以保证用户登录 ID 的唯一性，可以保证新用户注册后需要走审批流程才可开通使用的特性，系统在保存密码进数据库前已对密码进行强加密。

- 测试 2:

  名称：登录

目的：测试使用正确的用户名和密码是否可以登录系统，测试使用错误的用户名或密码是否可以被阻止使用系统，测试正常登录后是否可以访问单点登录系统之外的其他服务。

内容：使用系统中处于激活状态的用户名和明文密码尝试登录系统，查看是否可以正常进入内容页面，从而验证密码加密校验是否正常工作；使用随意输入的错误用户名和密码尝试登录，随后直接尝试访问非公开页面检查是否可阻挡未授权的访问。

结果：使用正确的用户名和明文密码可以进入系统，正常登录后可以访问其他微服务提供的功能；使用错误的用户凭证无法进入系统，无法访问除登录和注册页面之外的任何页面。

- 测试 3:

  名称：添加新逃犯基本信息

内容：测试添加逃犯基本信息功能是否正常，测试是否可以上传逃犯主识别图像，测试提交信息后是否可以正确记录所填写信息。

结果：执行添加逃犯信息流程后，检查相应数据库表，发现逃犯主识别图像的文件 ID 正确保存，检查对象存储服务存储桶文件记录可以发现对应的文件记录；其他字段保存正常。

- 测试 4:

  名称：发起通缉

目的：测试添加逃犯基本信息后第二步页面中直接对其发起通缉功能是否正常，测试通缉范围选择器是否正常加载出行政区划数据。

内容：执行标准添加逃犯基本信息流程，进入发起通缉页面选择通缉区域并填写通缉原因后提交；直接查看数据库表中记录，查看通缉区域 ID 属性是否保存，通缉原因属性是否保存正确；检查通缉区域 ID 在行政区划数据表中记录是否对应目标通缉区域。

结果：发起通缉功能可以正常工作，完成发起通缉操作后，检查数据库表相应记录

数据均正确；通缉范围的区域 ID 可以正确对应行政区划数据表中的相关区域。

问题：发起通缉第一步创建新的逃犯信息时需要上传逃犯识别照片，此属性不属于必填项，所以最终提交创建不会校验是否包含 fileID 数据。如果用户点击提交按钮时文件上传还未完成就有可能丢掉主识别图像数据。

改进：用户选择上传逃犯识别图像后暂时将提交按钮置为不可用，等待上传结果返回后再恢复按钮状态。

6.5. 小结

本章文档介绍了逃犯查询系统的部分测试方案，给出了已测试部分功能的测试结果和意见。在测试过程中发现了开发中忽略掉或者没有注意到的隐藏的问题，通过黑盒测试模拟用户使用过程中发现了一些设计缺陷和运行不正确的问题。

通过完成软件测试，检查了系统软件运行的正确性和可靠性，通过修复测试发现的问题，确保了软件交付的质量。

第 7 章　总结

本文通过编写一个逃犯查询系统的需求分析文档、系统概要设计文档、系统详细设计文档以及软件测试文档，完成了基于 SSM 框架的逃犯在线查询系统的设计；通过实施本文所述的设计方案，完成编码工作，实现了本文所述逃犯查询系统的大部分功能。

在设计本逃犯查询系统的思路上采用了目前软件开发实际工作中较新的微服务的设计方案，并且采用前后端完全分离开发的方法，后端仅关注 REST API 的开发，满足了系统可扩展的要求；前端技术使用了目前 GitHub 上 Star 数量最多的开源项目 Vue，配合 Axios 作为 HTTP Client 与后端服务进行交互。最后使用 Docker 容器来启动项目实现了微服务的快速部署。

本文中所述逃犯系统仅实现了最基本的 API 权限控制，后续系统应该还需要一套非常完善的可以支持多角色的精确到 API 级别对各角色任意组合的权限系统；由于权限系统的开发本身也是一个研究方向，本文所述系统没有过多的去涉及；若后续有机会可以继续研究。另一个局限性是本系统还应该加入一个邮件系统，来对用户注册审核结果、通缉令发布结果、新线索、本区域新逃犯等等事件的发生给相关用户发送邮件通知，提高效率；本次开发中没有实现此功能。

参考文献

[1]LinuxFoundation.KubernetesBasics.https://kubernetes.io/docs/tutorials/kubernetes-basics/, 2019-02-22.

[2]Guillaume Chau. Vue.js 2 Web Development Projects. Britain: Packt, 2017.

[3]Rick Osowski . Introduction to microservices. IBM Developer, 2015 .

[4]Craig Walls, 张卫滨. Spring 实战(第四版) . 北京: 人民邮电出版社, 2014 .

[5]郝佳.Spring 源码深度解析(第二版).北京: 人民邮电出版社, 2019.

[6]章仕锋,潘善亮.Docker 技术在微服务中的应用.电子技术与软件工程,2019(04).

[8]郭丞乾,蔡权伟,林璟锵,刘丽敏.单点登录协议实现的安全分析.信息安全究,2019,5(01) .

[9]魏春来,付永振.基于微服务的 DevOps 研究与实现.网络安全和信息化,2018(11):54-56.

[10]Pivot.Building a RESTful Web Service. Spring.io, 2018

[11]IBM. 微服务架构 .https://www.ibm.com/developerworks/cn/java/j-cn-java-and-microservice-1st/index.html, 2017-01-20

致　谢

首先特别感谢张玲老师和李钢老师在我的毕业设计进行过程中给予耐心指导和帮助；在毕设项目需求分析阶段由于题目比较罕见，可搜集的到的资料非常少，两位老师百忙之中抽出时间一同帮我思考系统的内容，这些是我完成此论文和项目的关键。

其次感谢东软睿道的刘玉霞老师和张艺娇老师在我完成毕业设计过程中提供的指导和帮助。

最后感谢云账户技术（天津）有限公司在我实习和完成毕设过程中为我提供了强大的硬件设备支持、办公环境支持和技术指导。

外文原文

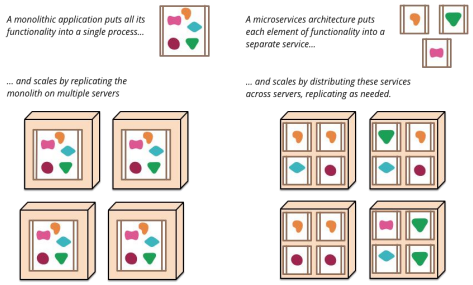
A Definition of Microservice

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

To start explaining the microservice style it's useful to compare it to the monolithic style: a monolithic application built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a *monolith* - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application.

Monolithic applications can be successful, but increasingly people are feeling frustrations with them - especially as more applications are being deployed to the cloud . Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource.

Figure.1 Monoliths and Microservices



These frustrations have led to the microservice architectural style: building applications as suites of services. As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams .

We do not claim that the microservice style is novel or innovative, its roots go back at least to the design principles of Unix. But we do think that not enough people consider a microservice architecture and that many software developments would be better off if they used it.

Characteristics of a Microservice Architecture

Componentization via Services

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define **libraries** as components that are linked into a program and called using in-memory function calls, while **services** are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. (This is a different concept to that of a service object in many OO programs.)

One main reason for using services as components (rather than libraries) is that services are independently deployable. If you have an application that consists of a multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the

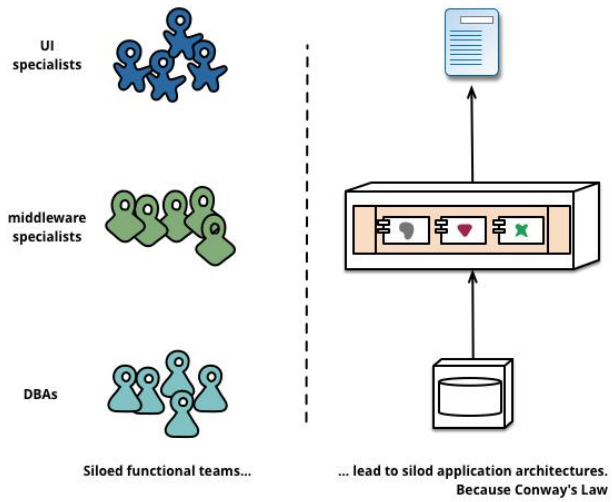
aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

Another consequence of using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining an explicit Published Interface. Often it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms.

Organized around Business Capabilities

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law in action.

Figure2 Conway's Law in action



The microservice approach to division is different, splitting up into services organized around **business capability**. Such services take a broad-stack implementation of software for that business area, including user-interface, persistant storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.

Products not Projects

Most application development efforts that we see use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime. A common inspiration for this is Amazon's notion of "you build, you run it" where a development team takes full responsibility for the software in production. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

The product mentality, ties in with the linkage to business capabilities. Rather than looking at the software as a set of functionality to be completed, there is an on-going relationship where the question is how can software assist its users to enhance the business capability.

There's no reason why this same approach can't be taken with monolithic applications, but the smaller granularity of services can make it easier to create the personal relationships between service developers and their users.

Smart endpoints and dumb pipes

When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.

The microservice community favours an alternative approach: *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as

WS-Choreography or BPEL or orchestration by a central tool.

The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging.

Decentralized Governance

One of the consequences of centralised governance is the tendency to standardise on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer. We prefer using the right tool for the job and while monolithic applications can take advantage of different languages to a certain extent, it isn't that common.

Splitting the monolith's components out into services we have a choice when building each of them. You want to use Node.js to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine. You want to swap in a different flavour of database that better suits the read behaviour of one component? We have the technology to rebuild him.

Teams building microservices prefer a different approach to standards too. Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing. These tools are usually harvested from implementations and shared with a wider group, sometimes, but not exclusively using an internal open source model. Now that git and github have become the de facto version control system of choice, open source practices are becoming more and more common in-house .

Decentralized Data Management

Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics.

This issue is common between applications, but can also occur within applications,

particular when that application is divided into separate components. A useful way of thinking about this is the Domain-Driven Design notion of Bounded Context. DDD divides a complex domain up into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify, and as we describe in the section on business capabilities, reinforce the separations.

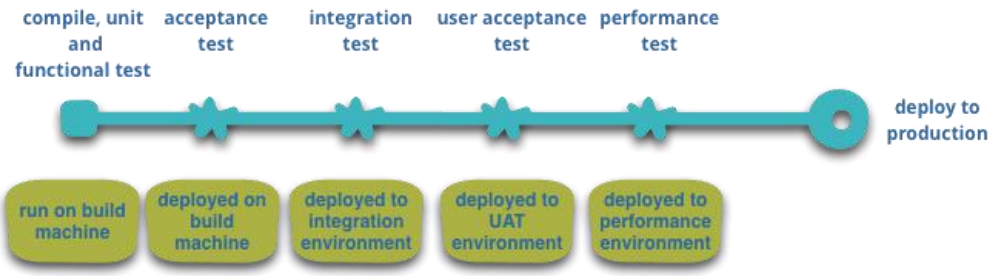
As well as decentralizing decisions about conceptual models, microservices also decentralize data storage decisions. While monolithic applications prefer a single logical database for persistant data, enterprises often prefer a single database across a range of applications - many of these decisions driven through vendor's commercial models around licensing. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence. You can use polyglot persistence in a monolith, but it appears more frequently with microservices.

Infrastructure Automation

Infrastructure automation techniques have evolved enormously over the last few years - the evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices.

Many of the products or systems being build with microservices are being built by teams with extensive experience of Continuous Delivery and it's precursor, Continuous Integration. Teams building software this way make extensive use of infrastructure automation techniques. This is illustrated in the build pipeline shown below.

Figure3 basic build pipeline



Design for failure

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

This is particularly important to a microservices architecture because the microservice preference towards choreography and event collaboration leads to emergent behavior. While many pundits praise the value of serendipitous emergence, the truth is that emergent behavior can sometimes be a bad thing. Monitoring is vital to spot bad emergent behavior quickly so it can be fixed..

Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples we often encounter in the wild.

Evolutionary Design

Microservice practitioners, usually have come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change. Change control doesn't necessarily mean change reduction - with the right attitudes and tools you can make frequent, fast, and well-controlled changes to software.

中文翻译

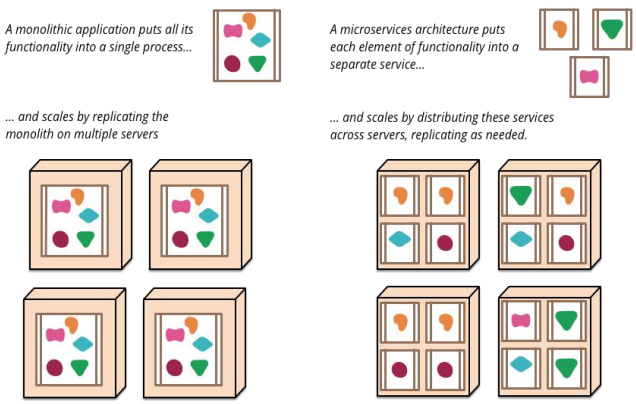
微服务的定义

在过去的几年中，“微服务体系结构”一词已经出现，用来描述将软件应用程序设计为独立可部署服务套件的一种特殊方式。虽然没有对这种体系结构风格的精确定义，但是围绕业务能力、自动化部署、端点中的智能以及语言和数据的分散控制，组织有一些共同的特征。

要开始解释微服务风格，将其与单片式风格进行比较是很有用的：单片式应用程序作为单个单元构建。企业应用程序通常由三个主要部分组成：客户端用户界面（由用户计算机上浏览器中运行的 HTML 页面和 JavaScript 组成）、数据库（由插入到公共数据库管理系统（通常是关系数据库管理系统）中的许多表组成）和服务器端应用程序。服务器端应用程序将处理 HTTP 请求，执行域逻辑，从数据库中检索和更新数据，并选择和填充要发送到浏览器的 HTML 视图。这个服务器端应用程序是一个单块单逻辑可执行文件。对系统的任何更改都涉及到构建和部署服务器端应用程序的新版本。

单片应用程序可能会成功，但越来越多的人对它们感到失望，尤其是随着越来越多的应用程序部署到云端。变更周期被捆绑在一起——对应用程序的一小部分所做的变更，需要重新构建和部署整个整体。随着时间的推移，很难保持一个好的模块化结构，这使得很难保持只影响该模块中一个模块的更改。扩展需要扩展整个应用程序，而不是部分需要更大的资源。

图 1：单体应用和微服务



这些挫折导致了微服务体系结构风格：将应用程序构建为服务套件。除了服务可以

独立部署和扩展之外，每个服务还提供了一个坚实的模块边界，甚至允许用不同的编程语言编写不同的服务。它们也可以由不同的团队管理。

我们并不认为微服务风格是新颖或创新的，它的根源至少可以追溯到 Unix 的设计原则。但我们确实认为，没有足够的人考虑使用微服务架构，而且如果他们使用它，许多软件开发会更好。

微服务体系结构的特点

1 通过服务实现组件化

微服务体系结构将使用库，但它们将自己的软件组件化的主要方式是分解成服务。我们将库定义为链接到程序中并使用内存函数调用调用的组件，而服务是与诸如 Web 服务请求或远程过程调用等机制通信的进程外组件。（这与许多 OO 程序中服务对象的概念不同。）

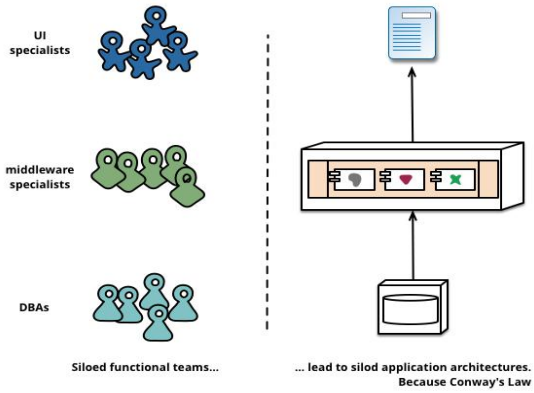
将服务用作组件（而不是库）的一个主要原因是服务可以独立部署。如果您的应用程序在一个进程中由多个库组成，那么对任何单个组件的更改都会导致必须重新部署整个应用程序。但是，如果该应用程序被分解为多个服务，那么您可以期望对单个服务的许多更改只需要重新部署该服务。这并不是绝对的，一些变化会改变服务接口，导致一些协调，但是一个好的微服务架构的目的是通过服务契约中的一致服务边界和演化机制来最小化这些变化。

使用服务作为组件的另一个结果是更显式的组件接口。大多数语言没有良好的机制来定义显式发布的接口。通常，只有文档和规程可以防止客户机破坏组件的封装，从而导致组件之间的耦合过紧。服务通过使用显式的远程调用机制更容易避免这种情况。

2 围绕业务能力组织

当要将一个大型应用程序拆分为多个部分时，管理层通常将重点放在技术层上，从而领导 UI 团队、服务器端逻辑团队和数据库团队。当团队沿着这些线分离时，即使是简单的更改也可能导致跨团队项目需要时间和预算批准。一个聪明的团队将围绕这一点进行优化，并针对两个缺点中较小的一个进行充实——只需将逻辑强制到他们可以访问的任何应用程序中。换句话说，逻辑无处不在。这是康威定律的一个实例。

图 2: 康威定律



微服务的划分方法是不同的，分为围绕业务能力组织的服务。这类服务需要为该业务领域实现大量的软件，包括用户界面、持久性存储和任何外部协作。因此，团队是跨职能的，包括开发所需的全部技能：用户体验、数据库和项目管理。

3 产品不是项目

我们看到的大多数应用程序开发工作都使用一个项目模型：其目标是交付一些软件，然后将其视为已完成。完成后，软件将移交给维护组织，建立它的项目团队将被解散。

微服务的支持者倾向于避免这种模式，他们更倾向于一个团队应该在其整个生命周期内拥有一个产品的概念。对此，一个常见的灵感来自于亚马逊的“你建立，你运行”的概念，即开发团队对生产中的软件负全责。这使开发人员能够与软件在生产中的行为保持日常联系，并增加与用户的联系，因为他们必须承担至少一部分支持负担。

产品心态，与企业能力的联系紧密。与其将软件视为一组要完成的功能，还存在一种持续的关系，问题是软件如何帮助其用户增强业务能力。

对于整体应用程序来说，没有理由不能采用相同的方法，但是服务的粒度越小，就越容易在服务开发人员和他们的用户之间创建个人关系。

4 智能端点和哑管道

在构建不同流程之间的通信结构时，我们看到了许多产品和方法，它们强调在通信机制本身中引入重要的智能。企业服务总线（EnterpriseServiceBus，ESB）就是一个很好的例子，ESB 产品通常包括用于消息路由、编排、转换和应用业务规则的复杂工具。

微服务社区支持另一种方法：智能端点和哑管道。从微服务构建的应用程序的目标是尽可能的分离和内聚——它们拥有自己的域逻辑，在传统的 Unix 意义上更像过滤器

——接收请求，适当地应用逻辑并产生响应。这些都是使用简单的 restish 协议进行编排的，而不是使用中心工具进行复杂的协议（如 WS-Choreography 或 BPEL 或编排）。

最常用的两种协议是 HTTP 请求响应和资源 API 和轻量级消息传递。

5 分散治理

集中治理的后果之一是趋向于在单一技术平台上实现标准化。经验表明，这种方法是紧缩的-不是每个问题都是钉子，不是每个解决方案都是锤子。我们更喜欢为工作使用正确的工具，虽然单片应用程序在某种程度上可以利用不同的语言，但这并不常见。

将整块体的组件分割成服务，我们在构建每个组件时都有选择。您想使用 node.js 来支持一个简单的报告页面吗？去争取它。C++用于一个特别接近实时的组件？好的。您想换一种更适合一个组件的读取行为的不同风格的数据库吗？我们有技术来重建他。

构建微服务的团队也喜欢使用不同的标准方法。与其使用一套写在纸上的已定义标准，他们更喜欢使用产生有用工具的想法，其他开发人员可以使用这些工具来解决他们所面临的类似问题。这些工具通常是从实现中获得的，并与更广泛的组共享，有时，但不是仅使用内部开放源代码模型。现在 Git 和 Github 已经成为事实上的版本控制系统的选择，开源实践在内部变得越来越普遍。

6 分散式数据管理

数据管理的分散以多种不同的方式呈现。在最抽象的层面上，它意味着世界的概念模型在系统之间会有所不同。这是跨大型企业集成时的常见问题，客户的销售视图与支持视图不同。在“销售”视图中称为“客户”的某些内容可能根本不会出现在“支持”视图中。那些有可能具有不同的属性和（更糟的）具有细微不同语义的公共属性。

此问题在应用程序之间很常见，但也可能发生在应用程序中，特别是当应用程序被划分为单独的组件时。一种有用的思考方法是有界上下文的领域驱动设计概念。DDD 将一个复杂域划分为多个有界上下文，并映射出它们之间的关系。这个过程对于整体和微服务架构都很有用，但是服务和上下文边界之间有一种自然的关联，这有助于澄清，正如我们在业务能力部分中描述的那样，加强了分离。

除了分散概念模型的决策之外，微服务还分散数据存储决策。虽然单片应用程序更喜欢单一逻辑数据库来存储持久性数据，但企业通常更喜欢跨一系列应用程序的单一数据库——其中许多决策都是由供应商的商业模型围绕许可做出的。微服务更喜欢让每个服务管理自己的数据库，要么是同一数据库技术的不同实例，要么是完全不同的数据库

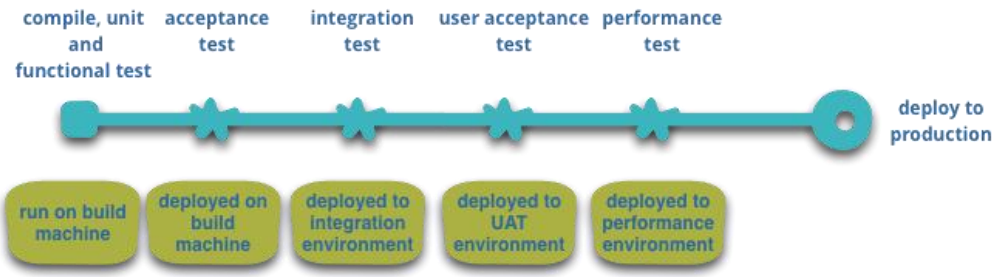
系统——一种称为 polyglot 持久性的方法。您可以在一个整体中使用 polyglot 持久性，但它在微服务中出现的频率更高。

7 基础设施自动化

在过去的几年中，基础设施自动化技术已经有了巨大的发展——尤其是云和 AWS 的发展降低了构建、部署和操作微服务的操作复杂性。

许多使用微服务构建的产品或系统都是由具有广泛连续交付经验的团队构建的，并且是它的先驱、持续集成。以这种方式构建软件的团队广泛使用基础设施自动化技术。这在下面的构建管道中进行了说明。

图 3：基本构建管道



8 失效设计

由于服务可以在任何时候发生故障，因此能够快速检测故障并在可能的情况下自动恢复服务是很重要的。微服务应用程序非常重视应用程序的实时监控，检查体系结构元素（数据库每秒收到多少请求）和业务相关指标（例如每分钟收到多少订单）。语义监控可以提供出错的预警系统，从而触发开发团队进行跟踪和调查。

这对于微服务体系结构尤其重要，因为微服务对编排和事件协作的偏好导致了紧急行为。虽然许多专家称赞意外发生的价值，但事实是，紧急行为有时是件坏事。监控对于快速发现不良紧急行为至关重要，因此可以对其进行修复。

微服务团队希望看到每个服务的复杂监控和日志记录设置，例如显示上/下状态的仪表盘以及各种与操作和业务相关的指标。有关断路器状态、电流吞吐量和延迟的详细信息是我们在野外经常遇到的其他示例。

9 进化设计

微服务从业者通常来自于一个进化的设计背景，他们将服务分解看作是一个更进一

步的工具，使应用程序开发人员能够控制他们的应用程序中的更改，而不会减慢更改的速度。变更控制并不一定意味着减少变更——有了正确的态度和工具，您就可以频繁、快速和良好地控制软件变更。