

上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS



论文题目 空气质量传感器管理系统设计与实现

学生姓名 孙 逢

学生学号 5120309032

指导教师 吴刚副教授

专业 软件工程专业

学院(系) 软件学院

上海交通大学 学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名: 某某

日 期: 某 年 某 月 某 日

上海交通大学 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保 密 在 _____ 年解密后适用本授权书。

本学位论文属于

不保密

(请在以上方框内打√)

学位论文作者签名: 某某

指导教师签名: 某某

日 期: 某 年 某 月 某 日

日 期: 某 年 某 月 某 日

空气质量传感器管理系统设计与实现

摘要

近几年，空气质量问题已经逐渐成为中国乃至全世界的社会焦点，但大多数地区只有政府部门会有监测站和监测点，普通民众并不能实时了解自己身边的空气质量。Clarity Movement Co. 研发出了火柴盒大小但精度可媲美光学空气质量分析仪的 PM2.5 传感器，并通过与企业和政府合作在传统硬件设备和城市建筑上搭载传感器来销售传感器。公司为之搭建了功能强大的云服务平台，可以实时地处理和推送空气质量数据，但还需要一个信息系统来操作和管理传感器。

在此背景下，本课题调研了大量 JS 框架和库基于现代主流的 WEB 开发技术为之开发了一个传感器管理系统。系统包含三个模块，分别是版本管理、Smart Home 和 Smart City 模块。版本管理模块面向公司的硬件开发人员，让他们能够方便地管理自己开发的传感器及其版本、批次、兼容性和拥有者。Smart Home 模块面向家电制造业的合作企业，让其能够使自己的家电互相联系起来。Smart City 模块面向政府合作者，让其能够直观地从不同角度查看和下载城市空气质量数据。

关键词：空气质量 传感器 信息系统 WEB 开发

Design and Implementation of Air Quality Sensor Management System

ABSTRACT

In recent years, air quality has gradually become the a social focus of Chinese and even the world. However, in most regional only governments have monitoring stations and sites and ordinary people can not get real-time air quality on their side. Clarity Movement Co. developed a matchbox-sized PM2.5 air quality sensor whose accuracy is comparable to the optical analyzer and sell the sensors by mounting them in the traditional hardware and city buildings in cooperation with businesses and governments. The company had built a powerful cloud service platform which can handle and push real-time air quality data but still needed an information system to manage and operate the sensor.

In this context, this paper investigates a large number of JS frameworks and libraries to develop a sensor management system based on modern mainstream web-development technology. The system consists of three modules, namely, Version Management, Smart Home and Smart City. Version Management module is for the company's hardware developers to easily manage the versions, batches, compatibilities and owners of the sensors. Smart Home module is for cooperative enterprises who are in appliance manufacturing to be able to link their appliances to each other. Smart City module is for government partners to visually view and download air quality of the city from different angles.

KEY WORDS: Air quality, Sensor, Information System, Web-development

目 录

插图索引	vii
表格索引	viii
第一章 绪论	1
1.1 课题背景和意义	1
1.2 项目定义	1
1.3 主要研究内容	1
1.4 论文组织结构	2
第二章 相关理论、技术和工具研究	3
2.1 设计理论	3
2.1.1 敏捷开发	3
2.1.2 Web Components	3
2.1.3 Material-design	4
2.1.4 响应式设计和 Flex 布局	4
2.1.5 RESTful API 设计和 Web Socket 数据更新	5
2.1.6 Oauth 和 JWT	5
2.1.7 Flux 架构模式	6
2.1.8 SPA 单页面应用和 server-render 服务端渲染	7
2.2 技术栈	8
2.2.1 前端: Angular VS React	8
2.2.2 后端: NodeJS 以及 Express VS Koa	11
2.2.3 数据库: MongoDB 和 RedisDB	13
2.2.4 服务器: AWS Elastic Beanstalk 云服务	13
2.3 开发工具	14
2.3.1 版本控制: Git 和 Git-flow	14
2.3.2 IDE: Webstorm VS Atom	17
2.3.3 代码生成: Yeoman 和 Yeoman Generators	17
2.3.4 构建工具: Grunt VS Gulp VS Npm	17
2.3.5 代码质量	18
2.3.6 单元测试: Karma、Mocha、Chai 和 Sinon	20
2.3.7 端对端测试: Protractor	21
2.3.8 持续集成: Solano CI	21

第三章 需求分析	22
3.1 版本管理模块	22
3.1.1 功能需求	22
3.1.2 用户用例	23
3.1.3 数据模型	23
3.1.4 数据字典	24
3.1.5 快速原型	24
3.2 Smart Home 模块	25
3.2.1 智能家居解决方案	25
3.2.2 功能需求	26
3.2.3 用户用例	26
3.2.4 设计稿	26
3.3 Smart City 模块	27
3.3.1 功能目标	27
3.3.2 用户用例	27
3.3.3 快速原型	27
3.3.4 设计稿	27
第四章 系统架构与开发实现	30
4.1 版本管理模块	30
4.1.1 API	30
4.1.2 API 路由	32
4.1.3 客户端模型定义	36
4.1.4 响应式设计	41
4.1.5 代码生成器	42
4.1.6 Git 提交记录	42
4.2 Smart Home 和 Smart City 模块	45
4.2.1 服务端渲染	45
4.2.2 前端路由器	46
4.2.3 Redux 数据流	48
4.2.4 API 请求库和 WebSocket	52
4.2.5 后端路由器和数据模型	52
4.3 主要组件详细设计和开发	53
4.3.1 业务无关的通用组件	54
4.3.2 Smart City 业务组件	55
4.3.3 Smart Home 业务组件	57

第五章 系统测试和部署	61
5.1 单元测试	61
5.2 端对端测试	62
5.3 自动部署	62
5.4 持续集成	63
第六章 总结与展望	66
6.1 工作总结	66
6.2 个人总结	66
6.3 工作展望	66
附录 A 数据字典	67
A.1 版本管理模块数据字典	67
附录 B 快速原型	71
B.1 版本管理模块快速原型	71
附录 C 设计稿	83
C.1 Smart Home 模块设计稿	83
参考文献	86
谢 辞	87

插图索引

2-1 Flux 单向数据流	6
2-2 Github angular 和 react 排名情况	9
2-3 Git-flow 工作机制示意图	15
2-4 Azwraith 分支网络	16
3-1 版本管理模块数据模型	23
3-2 版本管理模块原型样例	25
3-3 Smart Home 模块设计稿样例	26
3-4 Smart City 模块原型	28
3-5 Smart City 模块登录设计稿	29
4-1 传感器 API 目录结构	31
4-2 传感器 API 路由目录结构	33
4-3 创建用户和查看用户详情	34
4-4 修改用户和改动提示	35
4-5 传感器创建页面表单	39
4-6 传感器细节页面	40
4-7 响应式设计之前	41
4-8 代码生成器命令行交互界面	43
4-9 版本管理模块 Git 贡献记录	44
4-10 Flux 数据流	48
4-11 Redux 数据流	49
4-12 Smart City 和 Smart Home 模块 redux 目录结构	50
4-13 Echart 组件自动适应	54
4-14 loading 页面	55
4-15 AqiChart 组件	56
4-16 AqiMap 组件自动适应	56
4-17 SmartCity 组件	57
4-18 Room 组件	58
4-19 正在净化房间的 Robot 组件	59
4-20 HomePanel 组件	59
4-21 CityPanel 组件	60
5-1 Smart City 模块单元测试结果	61

5-2 版本管理模块端对端测试结果	62
5-3 版本管理模块自动部署流程	63
5-4 本系统在 SolanoCI 的持续集成页面	65

表格索引

2–1 Angular 和 React 的对比	11
2–2 Koa 和 Express 的对比	12

第一章 绪论

1.1 课题背景和意义

随着“雾霾”、“PM2.5”等话题的升温，空气质量问题逐渐成为人们关注的焦点，出门不仅要看“天气”还要看“空气”。虽然如今各式各样的空气质量网站、APP 层出不穷，有实时的也有预测的，但大多数据来源于政府环境部门监测站，少部分来自网站自己搭建的监测点。然而空气质量与天气不同之处在于，它受时间和空间的变化影响更大，对人们健康的影响也更大。因此，如何满足个人用户对周边空气质量的实时掌控成为一大难题。

Clarity Movement Co. (以下简称 Clarity) 是一家研发“世界上最小的空气质量传感器”的创业公司，目前完成研发的第二代传感器尺寸与火柴盒差不多大，便携度大增的同时，精度依旧不弱于光学空气质量分析仪。Clarity 的主要销售途径是与相关企业和政府合作，将传感器搭载在传统硬件设备和城市建筑上。Clarity 现在依旧组建了自己的云服务平台，传感器通过手机蓝牙或者 Wi-fi 上传空气质量数据，服务器处理并保存相关数据。然而，要让该产品能够为大众所用，配套的软件系统还未完备。

随着业务的发展，一方面 Clarity 自己内部需要一个信息系统来管理自己生产的传感器和传感器的版本、批次、拥有者等信息，另一方面 Clarity 的合作方往往是政府部门和传统硬件厂商如空调、汽车行业的企业，他们一般都不具备很强的软件开发能力，所以 Clarity 需要为他们定制合适的传感器管理系统来查看空气质量数据和管理传感设备。因此，本课题研究的“空气质量传感器管理系统”(以下简称本系统) 应运而生。

1.2 项目定义

本系统应当是一个全栈 JS 的 WEB 应用，具有实时性、现代化、单页面、响应式、模块化等特点，用于满足 Clarity 内部信息化需求和日益增长的业务发展的需要。

虽然总体上来讲是传感器管理系统，但需求中的确有一些互相关系不大的部分，所以需要分为互相关联但相对独立的模块。首先对于内部的传感器管理需求，主要是对版本的管理，本系统把它定义为“版本管理模块”，面向的用户是 Clarity 的硬件开发人员。其次对于合作方的需求，又可分为家庭和个人级别的“Smart Home 模块”和城市级别的“Smart City 模块”。Smart Home 模块面向想要构建智慧家庭的传统家电厂商，目前的主要目标客户是日本一家家电制造企业，最终用户则是使用家电和智能家居的个人；Smart Home 模块面向想要构建智慧城市的城市制造商或者政府，目前的主要目标客户是美国一家计划做智慧城市的政府部门，最终用户是政府工作人员。

1.3 主要研究内容

本课题的任务是设计与实现一个拥有 3 个模块的设备管理系统，为完成该任务，进行了如下研究：

1. 根据开发团队的技术储备和开发能力，结合公司的企业文化和社会经济水平，调研相关的理论、技术和工具，对于某些没有接触过或不熟悉的技术进行必要的学习和提高，如 ReactJS 就是本文作者在开发过程中自学的。保证在拥有充分的理论指导、充足的技术储备和高效的开发工具的情况下完成系统的设计与实现。
2. 详细分析 3 个模块的需求，与需求方积极沟通，深入挖掘甚至需求方自己都没想过需求，给出一系列对后续开发有帮助的文档、图表和设计；开发过程中频繁部署，及时地获得需求方的意见和反馈，及时地修复 bug 和调整需求方不满意的地方。
3. 设计并坚决执行测试方案，测试过程与开发过程同步甚至更早，单元测试和人工测试相结合，同时保障软件局部代码健壮性和整体功能的实现。

1.4 论文组织结构

本文一共分为八章，各章节介绍如下

第一章 绪论 简单阐述了本课题研究的背景和意义、项目定义、主要研究内容和本论文的组织结构。

第二章 相关理论、技术和工具研究 通过对比 WEB 设计开发的一些理论、技术和工具介绍了本系统主要使用的设计理论、技术架构和开发工具以及使用它们的原因。

第三章 需求分析 分析了本系统的功能目标、用户用例、性能要求和用户运行环境，并根据设计稿设计了快速原型和数据字典

第四章 系统设计与开发 介绍了总体上的设计思路，然后按模块介绍了设计和开发过程，最后介绍了几个主要组件的详细设计

第五章 系统最终实现效果 按不同模块和组件展示实现效果

第六章 系统测试和部署 介绍了本系统的开发、测试和运行环境以及持续集成的配置

第七章 总结与展望 总结了研究内容是否完成，思考了研究过程中的不足，并介绍了下一步的工作计划。

第二章 相关理论、技术和工具研究

软件工程是研究和应用如何以系统性的、规范化的、可定量的过程化方法去开发和维护软件^[1]，以及如何把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来的学科。它涉及到程序设计语言、数据库、软件开发工具、系统平台、标准、设计模式等方面。所以本课题首先学习了软件开发尤其是 WEB 开发的相关理论，调研、比较并选择了一些主流的 WEB 开发技术和高效的 WEB 开发工具，以期能够用快速、高效、低成本地完成系统的开发。

2.1 设计理论

本系统在开发速度、可重用性、UI/UE、响应式、实时性等方面具有较高的要求，这就意味着传统的瀑布式开发、CSS+HTML、ajax 请求等 WEB 开发理论不能达到本系统的要求。本章学习的理论可以给开发明确可行性、指明开发的方向、提高效率、减少重复造轮子，从而保证系统的顺利实现。

2.1.1 敏捷开发

敏捷开发有很多种，它们的具体名称、理念、过程、术语都不尽相同，相对于「非敏捷」，更强调程序员团队与业务专家之间的紧密协作、面对面的沟通（认为比书面的文档更有效）、频繁交付新的软件版本、紧凑而自我组织型的团队、能够很好地适应需求变化的代码编写和团队组织方法，也更注重软件开发过程中人的作用。^[2]

Clarity 是一家创业公司，业务需求变化迅速，人员结构比较精简，因此更加适合用敏捷开发。本系统的软件开发团队由 3 个人组成，每天早晨与需求方即公司 CEO 和硬件开发团队进行视频通话来总结前一天的开发进度和确定当天的开发任务，不断地在测试服务器部署新特性（feature）版本和在生产服务器部署热修复（hotfix）版本，每个特性完成后在正式发布（release）到生产服务器上，团队成员吃住同行、分工灵活，每个人都能独当一面。

2.1.2 Web Components

Web Components 是一组现在已经被 W3C 加入 HTML 和 DOM 规范的、为 Web 提供了一套标准组件模型的浏览器新特性，它们的设计理念是把“基于组件的软件工程”^[1]带到互联网的世界。Web Components 主要包括 4 个特性：

自定义元素（Custom Elements） 定义新的 HTML 元素的应用程序接口

影 DOM（Shadow DOM） 兼具封装性和可组合性的 DOM 和样式

HTML 引入（HTML Imports） 可声明的引入 HTML 文档到其他 HTML 文档的方法

HTML 模板（HTML Templates） 新的 <template>^[2] 标签，允许 HTML 文档包含惰性（延后定义或加载）的 DOM 块

^[1] 软件工程的一个分支，是一种基于重用的定义、实现和组合松散的独立组件的软件来开发系统的方法

^[2] <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template>

本系统的软件开发团队接受过良好的软件工程教育，对于重复造轮子这样的事情是坚决抵制的，所以我们对于组件的可重用性要求非常高。虽然本系统最终没有直接使用 Web Components 的官方实现 Polymer，但其中自定义元素、HTML 引入和 HTML 模板等特性都是非常有意义的，本系统中使用的 Angular 和 React 都有这些理念的原型，本系统在实现组件时也大量使用了这些概念来做到组件的可重用性。

2.1.3 Material-design

Material Design，代号为 Quantum Paper¹，是由 Google 推出的一种全新的设计语言，旨在为手机、平板、台式机等不同平台提供更加一致且广泛的外观和感觉。Material Design 总体来讲是一种隐喻式的基于纸张和墨水的设计，元素是扁平的、有阴影的，按照各自的高度浮动在背景上方，接缝和阴影让你知道哪些元素是可以触碰到的（操作的）。当你移动或改变它们的高度的时候，感觉是一张纸在被移动，符合人们对三维物体的直觉，与真实的纸不同之处在于 Material Design 的纸可以智能的伸缩和变形。

Clarity 的 CEO 本人是一名正在读本科的大学生，从创业伊始就对不管是硬件还是软件要求都特别高，在 UI 方面尤其如此，Material Design 是为数不多的能满足他的要求的设计语言。本系统中版本管理模块和 Smart City 模块都使用了 Material Design，深受设计师、程序员和用户喜爱，因为在这套设计语言的基础上，设计师省去了很多繁琐的顾虑，程序员实现时也有比较好的 UI 库，界面简洁明了，让用户直观地感受到页面上不同层次不同元素的重要性。

2.1.4 响应式设计和 Flex 布局

响应式设计是一种 WEB 设计方法，旨在构建出可以提供跨设备的（从桌面电脑到移动设备）拥有最佳的视觉和交互体验的网站，方便用户使用最少的缩放、平移和滚动操作来阅读和浏览。^[3] 为自动适应浏览器的设备环境，响应式网站使用流式的基于比例的网格布局，弹性的图片和 CSS3 media queries² 做到了以下几点：

流式网格 流式网格要求页面元素按照如百分比之类的相对单位来缩放而不是传统的如像素或点的绝对单位；

弹性图片 为防止图片显示时超出容器范围，弹性图片也按照相对单位来缩放；

Media queries Media queries 允许页面上的元素根据显示设备的某种属性来使用不同的样式，这里的属性主要是设备的宽度；

Flex 布局，全称 CSS Flex Box Layout，意思为弹性盒状模型布局，它是由 W3C 在 2009 年提出的一种新的布局方案，可以简便地、完整地、响应式地实现各种简单或复杂的页面布局，目前所有主流浏览器都支持 Flex 布局。

Clarity 作为一家创业公司没有那么多精力同时维护不同平台上的应用，因此虽然目前并不要求适配手机，但起码的屏幕宽度兼容是需要的，为将来适配更小的屏幕打好基础。本系统中三个模块都使用了 flex 布局，从而使得页面上的元素可以随着设备屏幕大小的变化（或浏览器的缩放）而自

¹<http://www.androidpolice.com/2014/06/11/exclusive-quantum-paper-and-googles-upcoming-effort-to-make-consistent-ui-simple/>

²没有正式的中文译名，直译为媒体查询，是 CSS 中 @media 规则的扩展

动伸缩。虽然我们的系统不需要适配手机，但其中版本管理模块因为页面上有比较宽的列表，在窄屏幕上显示不正常，于是采用了响应式设计来避免这个问题。

2.1.5 RESTful API 设计和 Web Socket 数据更新

RESTful 中文名叫“具象状态传输”，全称“Representational state transfer”，它是一种包含一系列同在一个分布式网络系统中协调的组件、连接器和数据元素如何按照不同的角色进行交互的设计原则（或者叫设计风格），旨在促进系统的性能、可扩展性、简单性、可修改性、可读性、便携性和可靠性。^[4,5]相比于 SOAP¹和 XML-RPC²更加简单轻量，在 URL 处理和 Payload 编码上更加简洁明了。这套设计原则并不局限于 WEB 应用程序，只要满足其约束条件的和原则的应用程序或设计就是 RESTful。WEB 应用程序最重要的 REST 原则是，客户端和服务器之间的交互在不同请求之间是无状态的，这就意味着服务器可以随时重启并且客户端并不关心连接的是哪台服务器，这十分适合云计算³的环境。

WebSocket 是 HTML5⁴中一种新的通信协议，浏览器与服务器通过一个 TCP⁵连接上的全双工通信 (full-duplex) 进行交互，只有一开始的握手需要借助 HTTP⁶请求完成。它使得浏览器可以和网站进行更多更频繁更实时的交互，从而可以摒弃从前通过轮询⁷来获取最新数据却会给服务器带来沉重负担的方法。

Clarity 如其他众多创业公司一样，选择使用云服务器来搭载自己的服务，因为其自动扩展的虚拟化资源无论是直接成本还是管理成本都比自己公司搭建服务器低。本系统搭载在 AWS 云服务器上，无状态交互是可扩展性的一大保障，频繁地增删改查设备等数据资源也需要 RESTful，于是设计了一套 RESTful 风格的 API，前端和后端分别遵循 API 接口来实现。另一方面需要让用户能够实时地看到自己的修改以及空气质量的变化，本系统使用 WebSocket 来实时地更新数据，前后端分别使用收听和发送相关主题 (topic)。

2.1.6 Oauth 和 JWT

OAuth 是 Open Authorization 的简写，它是一个被广泛用作互联网用户使用它们的微软、Google、Facebook、Twitter 等账号在不用输入它们的密码的情况下登录到第三方网站的开放授权标准。一般来讲，OAuth 提供给客户端一个代表资源拥有者访问服务器资源的“安全访问授权”。它规定了资源拥有者授权第三方访问其服务器资源而无需共享它们的凭据的过程。它是专门为超文本传输协议 (HTTP) 设计的，本质上 OAuth 允许一个授权服务器在资源拥有者批准的情况下直接把访问令牌 (Access Token) 发送到第三方客户端，然后第三方再使用访问令牌来获取资源服务器上被保护的资源。^[6]

¹ Simple Object Access protocol，简单对象访问协议

² XML（标准通用标记语言下的一个子集）远程方法调用

³ 提供动态的易扩展的虚拟化计算资源的互联网服务模式

⁴ 万维网的核心语言、标准通用标记语言下的一个应用超文本标记语言 (HTML) 的第五次重大修改

⁵ 即 Transmission Control Protocol 传输控制协议

⁶ 即 HyperText Transfer Protocol 超文本传输协议，是互联网上应用最为广泛的一种网络协议

⁷ 每隔一秒向服务器发送一个 HTTP 请求来获取最新数据

JWT 是 JSON Web Token 的简写，它是一个用来在 WEB 应用环境下在不同参与者之间传递声明 (claims) 的开放标准。这些令牌被巧妙地设计成紧凑、URL 安全且可用的，最适合使用在浏览器单点登录¹的情况下。JWT 声明被典型地用于在身份提供者和服务提供者或者任何其他业务过程中需要身份的地方之间传递认证用户的身份识别。^[7] 这些令牌本身也可以被认证和加密。

Clarity 虽然业务规模还不大，用户数量也很少，但在安全认证方面也是紧跟行业潮流，为将来接入各种社交账号登录做准备。本系统使用 OAuth 标准，其中访问令牌 (Access Token) 使用的是 JWT。用户登陆后除了会获得一个对应的访问令牌之外还有一个寿命较长的刷新令牌 (Refresh Token) 用来在用户短期离线后自动刷新访问令牌，而这一切除了一开始的登录都不需要用户重新输入密码，只有在用户长期离线刷新令牌也过期的情况下才需要再次登录。

除了对 HTTP 请求做了权限验证之外，本系统对 WebSocket 部分也做了相应的权限验证，socket 客户端连接上 Socket 服务器后，会先通过 socket 把自己的访问令牌 (Access Token) 发送给服务器，服务器验证后会发回一个通过验证或者未通过验证的消息，socket 客户端通过收听这两个消息来知道自己是否验证成功。如果 socket 客户端未通过验证，就会通知前端的原本用来刷新令牌的机制去刷新访问令牌。而每次访问令牌被改变或刷新后，socket 客户端都会重新发送请求认证的消息，socket 服务端也会通知客户端之前的认证已经失效。

2.1.7 Flux 架构模式

Flux 是 Facebook 用来构建客户端 WEB 应用的应用架构，它利用一个单向的数据流对 React²的可组合的视图组件进行了有力的补充。它更像是一种模式而不是一个框架，所以在这里本课题认为它属于理论而不是技术。

它与传统的 MVC 模型³不同，主要包括三部分：dispatcher、store 和 view，由 action 来触发状态 (state) 变化。如下图所示：所有的 action 会进入到 dispatcher 进行处理，dispatcher 会产生新的 state 用以更新 store，store 选择恰当的时机更新后通过 view 提前注册好的消息（回调）来告诉 view 更新用户界面。但其实大部分情况下 action 由用户的操作产生，因此会有从 view 产生的 action。

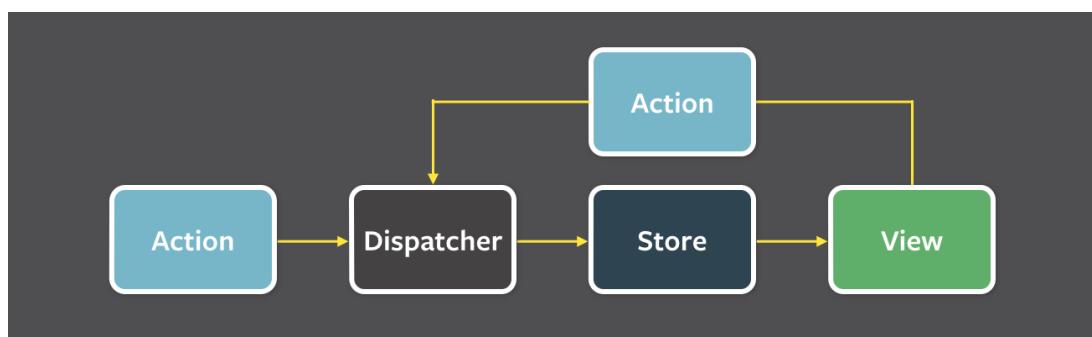


图 2-1 Flux 单向数据流
Fig 2-1 Flux Unidirectional Data Flow

¹SSO, single sign-on

²会在下一节技术架构中介绍到

³即 Model View Controller，是一种软件设计典范

单向数据流传输的 Flux 不会像 MVC 那样，一旦系统复杂了，model 和 view 之间的联系呈几何级数增长，也不会像双向绑定那样，一旦绑定层数多了，连程序员自己都不清楚状态更新是被哪里触发的。

Clarity 致力于给用户最好的体验，因此在给合作方做的 Smart Home 和 Smart City 两个模块中使用了 Flux 架构模式。用户也许会在潜意识里感觉到，他们在我们的应用上看到的状态永远是一致的，不会出现类似明明看到有未读的消息点进去却发现没有的情况。

2.1.8 SPA 单页面应用和 server-render 服务端渲染

单页面应用是一种只有一个网页的 WEB 应用，旨在给用户提供像桌面应用一样流程的用户体验¹。之所以能够做到如此，是因为所有的 HTML、JavaScript 和 CSS 文件都在最初的加载中完成，另有一些资源在用户交互的过程中动态的加载到页面中。整个页面在运行过程中不会重新加载，也不会突然打开一个新的页面，但地址栏中的地址会一直变化，直接输入这些地址也能够直接访问到 WEB 应用相应的页面。单页面应用在让用户获得流畅体验的同时，还减轻了服务端的负担，服务端更多地只是承担一个静态文件和动态资源提供者的角色。

现在中国互联网界所谓的 H5 移动应用²之所以能够流行就是得益于 SPA，因为除了一开始加载比较慢，这些应用操作起来像原生移动应用一样流畅。但正所谓成也 SPA，败也 SPA，SPA 最大的性能优点带来了它最大的性能缺点：初始加载缓慢，现在是一个节奏非常快的社会，很多人在互联网上习惯了一点链接就看到结果，缓慢加载的 SPA 给用户带来一种这个页面加载不出来的感觉，即使有一个旋转的加载动画也不能解决这个问题。

使用“单页面移动应用”的创业公司的客服听到的最多的问题就是“为什么加载不出来？”，然后客服机械地告诉他：“请您再耐心地等一会儿，不要急着关掉页面，初次加载完了，有了缓存下次访问就快了”。是的，目前解决这个问题的主要手段就是缓存，尤其是手机浏览器缓存比电脑浏览器更多更顽固，但熟悉 WEB 应用的人都知道，客户端顽固的缓存很多时候能造成比用户体验差严重得多的问题。单页面应用初始加载缓慢导致用户体验差的核心原因是不等所有的 HTML、JavaScript 和 CSS 文件加载完成，客户端无法渲染页面，用户迟迟看不到任何信息所以才会用户体验差，于是刚刚告别了服务端渲染出来单打独斗的单页面应用又在初次加载时重新拥抱了它。

使用初次加载的服务端渲染时，服务端不再是简单地、静态地提供 HTML、JavaScript 和 CSS 文件，而是先用服务器强大的计算能力替用户计算出一个能够直接被浏览器渲染的 HTML，客户端先把这个结果给用户看，然后在悄悄地、缓慢地加载全部的文件。好像刚刚获得解脱的服务端又负担起了一个艰巨的负担，其实不然，缓存再次解决了这个问题，而且服务端缓存不像客户端缓存那么多变³且不可控制。

本系统在 Smart Home 和 Smart City 模块中使用了服务端缓存

¹有些传统的非单页面应用，每点一个按钮，整个页面都会刷新一下，严重影响用户体验

²其实是一种非专业人士的误读，应该叫单页面移动应用

³客户端环境和浏览器的不同

2.2 技术栈

Clarity 的开发团队是一个精简且高效的年轻团队，我们选择了统一而不是分散的技术栈：全栈 JavaScript。从前端到后端，从业务逻辑到单元测试，全部使用 JS，这样给我们带来了不少好处：1. 提高了代码重用率，因为前后端可以共同使用一些代码逻辑 2. 提高了分工灵活度，前后端的人力可以随时支援对方 3. 提高了开发速度，因为全栈 JS 的生态圈很强大、包管理系统也非常省心省力 4. 提高了代码可维护性，因为全栈 JS 的各种代码规范标准非常强大…

下面就开始分前端、后端、数据库和服务器来介绍本系统将会使用到的技术栈：

2.2.1 前端：Angular VS React

目前 JS 的主流前端技术无外乎两个选择，Angular 和 React，如图 2-2 所示，这两个项目在 Github 上排行分别是第 5 和第 6。

2.2.1.1 Angular

AngularJS 是一个主要由 Google 维护的开源 WEB 应用框架，另外还有一个由个人和企业组成的解决 SPA¹开发过程中遇到的挑战的社区参与维护。AngularJS 旨在通过提供一个客户端 MVC 架构和 MVVM²架构的框架来简化 SPA 的开发和测试，另外它还提供了许多在丰富互联网应用（Rich Internet Application）³中被广泛使用的组件（component）。

AngularJS 首先读取预先嵌入了额外的自定义标签和自定义属性的 HTML 页面，Angular 把这些标签和属性解译为一些 directive⁴，它们可以把页面上输入和输出的部分绑定到一个标准 JavaScript 变量模型（model），模型中的值可以在代码中设置或者来自静态或动态的 JSON⁵资源。

AngularJS 在 2010 年首次发布，除了开源，它有一些创举，极大解放了前端工程师以及后端工程师的生产力：

1. 使用依赖注入（dependency injection）⁶，使得客户端也可以使用 MVC，把 WEB 应用程序的客户端和服务端解耦合了，减轻了原本后端路由控制、页面渲染的负担，使得两边都可以被重用
2. 使用双向绑定（two-way data binding）⁷，把 DOM 操作从应用程序逻辑中移除了（或者说尽量避免使用），增强了客户端程序的可测试性和性能；增强了 HTML，在 HTML 标签中插入模型变量（model），使得开发者编写 HTML 时“所见既所得”。
3. 提供了一整套构建 WEB 应用的过程：从 UI 设计，到业务逻辑，再到测试，大量的可重用组件和强大的生态圈使得开发者大量减少重复造轮子的现象

¹Single-page applications 单页面应用

²model-view-viewmodel

³是一种拥有很多桌面应用软件特性的 WEB 应用

⁴AngularJS 中对可重用组件的定义

⁵JavaScript Object Notation, 是一种使用可读的文本来传输以键值对表示的数据对象的开放标准格式

⁶一种使用控制反转来处理依赖管理的软件设计模式

⁷视图和逻辑中的模型变量（model）互相绑定，牵一发而动全身

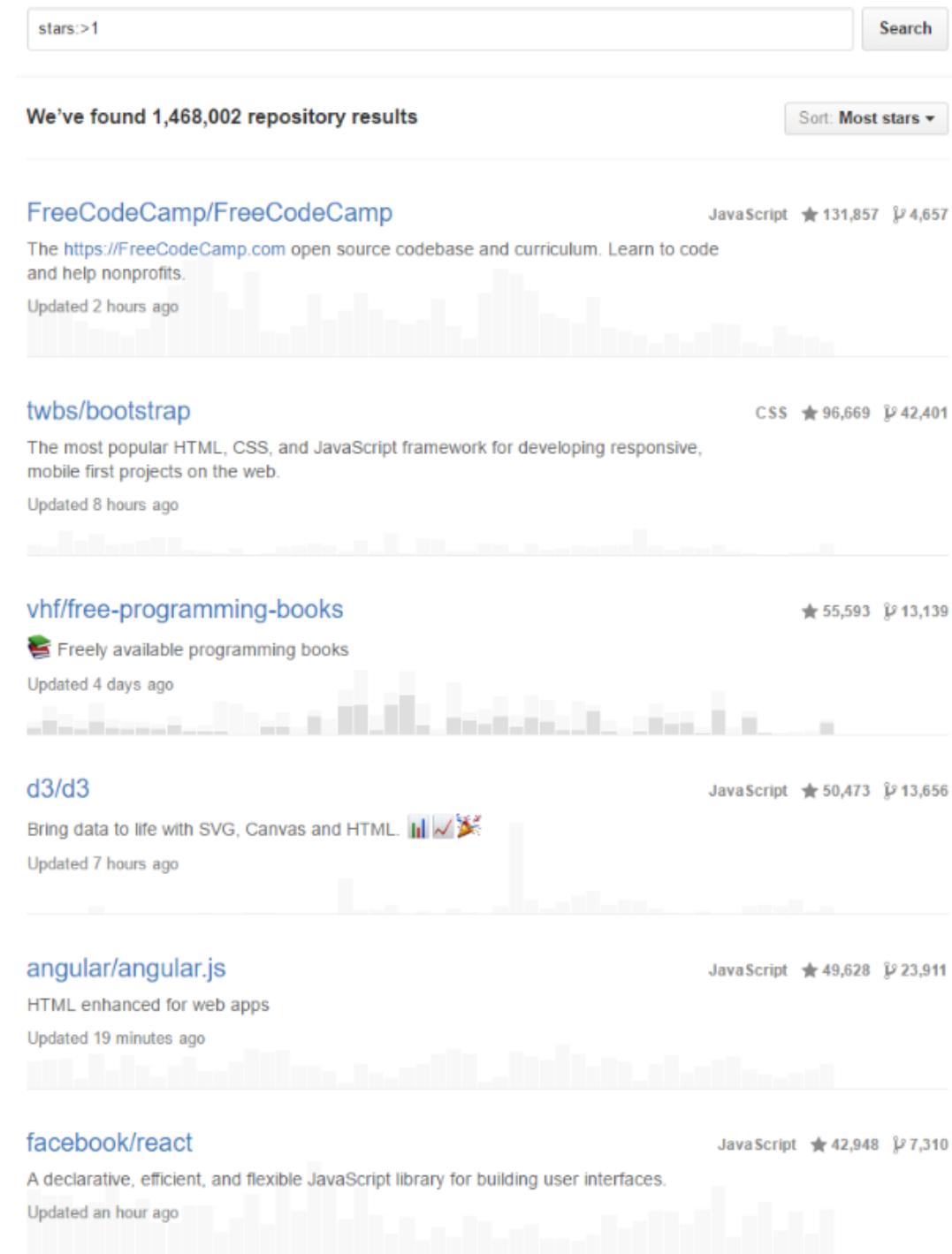


图 2–2 Github angular 和 react 排名情况
Fig 2–2 Github star ranking of angular and react

2.2.1.2 React

ReactJS 是一个用来把视图数据渲染为 HTML 的开源 JavaScript 库，它的主要维护者是 Facebook、Instagram 和一个由个人开发者和企业组成的社区。React 视图用以自定义 HTML 标签形式定义的组件 (component) 来渲染，这些组件可以也包含其他的组件。React 在现代单页面应用开发上给开发者提供了一个这样的模型：1. 子组件无法直接影响父组件的¹ 2. 当数据有所改变时，高效地更新 HTML 文档 (document) 3. 干净的组件隔离

ReactJS 是一个年轻的库，它在 2011 年被首次部署，在 2013 年美国 JS 大会 (JSConf US 2013) 上被开源，它吸收了很多 Angular 的优点，同时也发现了 Angular 的不足，总得来讲，ReactJS 有以下几点特性是它如此成功的原因：

单向数据流 (One-way data flow) 主要由前文介绍的 Flux 架构保证，每个组件会收到一些属性 (properties) 用来渲染到 HTML 标签中。与 Angular 的双向绑定不同，组件本身不可直接修改这些属性，只能通过作为属性传入的回调函数来修改，在 Flux 架构中，就是触发 Action，这种原理被叫做 “properties flow down actions flow up”。

虚拟 DOM (Virtual DOM) React 构造了一个内存中的 DOM 缓存，统一计算每次渲染之间的差异，然后高效地对浏览器中的 DOM 进行差异更新，相比于 Angular 而言，Angular 是监听每个视图模型 (model) 的变化，然后更新 DOM 中对应的部分，并没有差异更新的能力。

JSX² JSX 允许 React 很方便地在 JS 代码中渲染子组件 (subcomponents)。与 Angular 增强 HTML 不同，React 依靠 JavaScript 本身的强大，更加方便地做到了一样的功能。

不得不提的一点是 Angular 的团队同样知道这些不足，并开发了 Angular2，Angular2 在很多方面与 React 很相似，同时又保留了一些 Angular 的优点。

2.2.1.3 综合比较与选择

目前来说 Angular 是 2015 年的 JS 宠儿，而 React 则后来居上，其实 Angular2 应当能与 React 媲美，但是在项目研发时，Angular2 离可以正式使用还有一段距离，React 则已经接近正式发布。

Angular 和 React 最大的不同在于一个是 WEB 框架，一个是 JS 库。WEB 框架意味着开发一个主流的单页面 WEB 应用程序所需要的一切³Angular 都提供给你了，而 React 崇尚小而精，它只提供了最核心的部分，其他部分都由开发者自己从社区中选择或者自己开发组合起来搭建一个 WEB 应用程序。因此也就带来了 Angular 和 React 的一大不同，Angular 开发 WEB 应用程序比较方便快捷，而用 React 则有更多的选择，可以有更多的个性化定制，更加灵活多变。

如表 2-1 所示，Angular 和 React 在很多方面各有不同，有些并不能分出优劣。

本系统中版本管理模块类似传统的后台管理系统，因此选用了稳定、成熟且开发迅速的 Angular；而 Smart Home 和 Smart City 模块是给用户定制的，所以选用了灵活度更大，性能更好，页面一致性更高的 React。具体使用了哪些技术，在后面对模块的详细设计中会有所介绍。

¹所谓的数据向下流，data flows down

³狭义上的，当然不包括各种开发工具

表 2-1 Angular 和 React 的对比

Table 2-1 Comparisons between Angular and React

对比	Angular	React
灵活度	一般	灵活多变
稳定性	稳定且成熟	接近稳定
开发速度	快	一般
社区	强大	一般但快速成长
性能	一般	很高
支持 Web Component ¹	弱	强
视图核心语言	HTML	JavaScript
页面一致性 ²	一般	强

¹ React 设计之初就考虑到对 Web Component 的支持，而 Angular 的发布和 Web Component 概念提出大概是在同一时期。

² 在上一节的 Flux 介绍中提到，Angular 的双向绑定可能会导致页面出现一些不一致，而程序员自己都难以定位。

2.2.2 后端：NodeJS 以及 Express VS Koa

NodeJS 是一个用来开发服务端 WEB 应用的开源的跨平台的运行时环境，NodeJS 在运行时用 Google 的 V8 引擎¹编译 JavaScript。尽管 NodeJS 不是一个 JavaScript 框架，但它很多的基础模块都是用 JavaScript 写的，开发者也可以用 JavaScript 编写新的模块。

NodeJS 提供事件驱动的架构和异步的 I/O²处理。这些设计决策旨在优化 I/O 集中或者实时的 WEB 应用的吞吐量和可扩展性，实时 WEB 应用主要是实时通信类应用和网页游戏。

NodeJS 允许 WEB 服务端和网络工具的开发者使用 JavaScript 和一个核心功能模块（Modules）的集合，包括文件系统 I/O，网络 I/O（DNS, HTTP, TCP, TLS/SSL 和 UDP），二进制数据，加密函数、流数据³等^[8]。NodeJS 的模块使用一套设计好的 API 来减少编写服务端应用的复杂度。NodeJS 应用程序可以在 Mac OS X, 微软 Windows, NonStop⁴ 和 Unix 系统的服务器上允许。

2.2.2.1 ExpressJS

ExpressJS 是用来搭建 WEB 应用程序的 NodeJS 服务器框架，它几乎就是 NodeJS 标准的服务器框架。它最初的作者 TJ Holowaychuk 把它描述为一个由 Sinatra⁵启发的服务器，也就是说它是相对

¹ Chromium 项目为 Google Chrome 浏览器开发的一个开源的 JavaScript 引擎

² 一种允许其他进程在传输结束之前继续执行的 input/outpush 处理方式

³ 数据以一个系列的数据元素组成，分批地传输，如视频、音乐等媒体资源的传输往往用流数据的方式

⁴ 一系列专门的服务器计算机

⁵ 一个用 Ruby 写的免费开源 WEB 应用程序库，同时也是一门域专用语言（domain-specific language），特点是核心最小化，各种特性（或功能）以插件的形式来提供

最小的但用插件提供了很多功能。Express 作为后端部分与 MongoDB 数据库和 AngularJS 前端框架共同组成 MEAD stack¹。

2.2.2.2 KoaJS

KoaJS 由 Express 原班人马倾力打造，致力于把 Koa 做成一个更小、更加健壮、更富有表现力的 WEB 框架。Koa 提供了不同的 generator²，可以避免琐碎的回调函数嵌套，并极大地简化了错误处理，同时还提升了代码的可读性和健壮性，使得 WEB 应用开发变得得心应手。Koa 的内核中没有任何事先绑定的中间件（middlewares），它只提供了一个轻量级的函数库。

2.2.2.3 综合比较与选择

从设计哲学上讲，Koa 想要“修复并取代 NodeJS（fix and replace node）”，而 Express “补充了 NodeJS（augments node）”。Koa 可以被看做是 NodeJS 的 http 模块的抽象，而 Express 是一个 NodeJS 的应用框架。

表 2-2 Koa 和 Express 的对比
Table 2-2 Comparisons between Koa and Express

特性/功能	Koa	Express
中间件内核	√	√
路由		√
模板		√
发送文件		√
JSONP ¹		√
没有回调	√	
更好的错误处理	√	
不需要域	√	

¹ JSON with Padding, 是一种 WEB 开发者用来跨域的技术.

如表 2-2 所示，Koa 和 Express 相比更加专精于 HTTP 的处理，且更加优秀。

本系统中版本管理模块因为技术模型比较成熟，选择了 MEAN 捆绑包，也就是在后端部分选择了 Express。而在另外两个模块中，使用了同一个后端，且则选择了更加便于维护的 Koa。与 Koa 相结合来使用的一些模块，在后面详细设计中会具体介绍。

¹ 一个用于构建动态网站和 WEB 应用的开源 JavaScript 捆绑包，MEAN 分别指的是 MongoDB, Express, Angular 和 NodeJS

² ES6 的一种新特性，可以用一种连续的方式写异步函数

2.2.3 数据库：MongoDB 和 RedisDB

2.2.3.1 MongoDB 和 Mongoose

MongoDB 是一款免费的、开源的、跨平台的、面向文档（document-oriented）的非关系型数据库。MongoDB 避免了传统的基于表的关系型数据库的结构，转而使用一种类似 JSON 的有动态描述（schema）¹的文档（documents）来存储数据，给一些特定类型的应用带来了更简便更快速的数据集成。据月度排行网站 DB-engines.com 统计，到 2016 年 6 月，MongoDB 是世界上第四流行的数据库，并且是最流行的文档数据库。待填充（MongoDB 主要特性和功能）

Mongoose 是一个开源的 JavaScript 库，旨在为 NodeJS 提供优雅的 MongoDB 对象模型，给开发者提供了直截了当的基于描述（schema）的方法来为应用程序数据建模，包括内置的类型转换、校验、查询构建和业务逻辑 Hook。

2.2.3.2 RedisDB

Redis，这个名字是“REmote DIctionary Server”的缩写，它是一个基于内存的、开源的、支持远程访问的、具有可选的耐用性的数据结构服务器，也可以认为是一种键值对存储数据库。据月度排行网站 DB-engines.com 统计，RedisDB 是世界上第 10 流行的数据库，并且是最流行的键值对存储数据库。RedisDB 主要被广泛用于服务器端来缓存一些经常被访问的资源和一些临时的状态。

跟 MongoDB 一样，要在 NodeJS 中使用，需要一个类似 Mongoose 的库，它就是 node_redis。node_redis 是一个完整支持所有 Redis 指令的、聚焦高性能的开源 JavaScript 库。

本系统中所有模块都是使用的同一个 MongoDB 并使用 Mongoose 来对数据进行建模，其中 Smart Home 和 Smart City 的后端因为需要计算一些临时的平均值等原因使用了 Redis 来缓存这些状态。

2.2.4 服务器：AWS Elastic Beanstalk 云服务

AWS，全称 Amazon Web Services，是亚马逊旗下的一家子公司。AWS 云服务是该公司提供的一套云计算服务，这些服务组成了一个按需分配的云计算平台。AWS Elastic Beanstalk 是其中一款 PaaS（平台及服务的）云服务。它允许用户创建应用程序并把他们推送到一个可定义的 AWS 服务集合，其中包括弹性计算云²，简单存储服务³，简单通知服务⁴，云监测⁵，自动扩展服务⁶和弹性负载均衡器⁷。Elastic Beanstalk 在简单的服务器和操作系统之上又额外提供了一层抽象，实际上用户看到的是一个预装好的操作系统和平台的组合。

云服务正是大多数初创公司选择的部署服务器的方式，弹性的计算资源意味着用户量小的时候成本极低，而 AWS 在全球有 12 处地理分区，覆盖了全球绝大部分地方，所以成了 Clarity 的不二之

¹ 数据库中用某种语言来描述的数据的结构或蓝图

²Elastic Compute Cloud，简称 EC2，可以弹性地分配计算资源，AWS 最核心的服务之一

³Simple Storage Service，简称 S3，用于文件存储，也是 AWS 最核心的服务之一

⁴Simple Notification Service，简称 SNS

⁵CloudWatch，可以实时监测 EC2 用户的资源利用率

⁶autoscaling

⁷Elastic Load Balancers

选。像 Elastic Beanstalk 这样的平台及服务的云服务更是给初创公司带来的诸如自动部署、即时通知、实时监测、自动扩展、负载均衡等便利，因此 Clarity 选择了使用 Elastic Beanstalk 云服务。

2.3 开发工具

工欲善其事，必先利其器。有正确的理论指引方向和合适的技术铺平道路，还需要选择最佳的工具来加速前进。本课题调研并选择了一些最符合系统要求的工具来辅助开发，包括版本控制工具、编辑器、代码生成器、文档生成器、代码质量检测工具、编译工具、单元测试工具、持续集成工具等。

2.3.1 版本控制：Git 和 Git-flow

Git 是一个被广泛用于软件开发和其他版本控制任务的版本控制系统（Version Control System）。它是一个强调速度、数据完整性而且支持分布式、非线性工作流的分布式版本管理系统。与其他大多数分布式版本控制系统一样，不像大多数客户端、服务端系统，每一个 Git 工作目录（working directory）是一个带有全部历史信息和完全的版本追溯能力的完整仓库。

待填充（Git 主要特性和功能）

本系统一共涉及到 4 个仓库，3 个模块分别一个仓库，名字分别是前文提到过的 balanar、azwraith 和 robotic。另外还有一个比较特殊的是版本管理模块的代码生成器有一个单独的仓库 generator-material-app，这个会在后面详细设计中介绍到。

Git-flow 是提供了一系列高级仓库操作的一个 Git 扩展集或者说是一个 Git 指令库，基于 Vincent Driessen 的分支模型^[9]，该模型是一个能够帮助开发者在大型项目中同时追踪数量众多的新特性（features）、热修复（hotfixes）和发布（releases）的 git 分支和发布策略。

Git-flow 管理的项目仓库中有两个主要的分支：主分支（master branch）和开发分支（develop branch），分别对应正式生产环境¹和测试环境²；另外还有一些辅助分支：新特性分支（feature branches）、预发布分支（release branches）和热修复分支（hotfix branches），其中新特性分支对应新特性测试环境³，其余两个对应准生产环境⁴。

如图 2-3 所示，竖直向下的轴是时间轴，横轴是不同的分支，图上的圆圈圈住的点表示发生在不同时间的不同的提交（Commit），带有箭头的线表示提交的父子关系，箭头来自的提交是箭头指向的提交的父亲⁵。本课题将从时间顺序剖析这张图来介绍 Git-flow 的工作机制，假设开发人员甲负责新特性开发，开发人员乙负责修复发布和 BUG，任何人都可以维护开发分支

1. 故事从任何一次发布之后开始，主分支上带有 0.1 标签（Tag）的是最早的一个提交，“Tag0.1”标志着刚刚经过了一次正式发布；
2. 发布完成之后，开发人员先把主分支合并到开发分支，产生了一个提交，然后可能做了两个影响不大的微小调整（两个新的提交）；

¹ 正式给用户使用的环境

² 给测试人员测试和测试用户熟悉系统的环境

³ 专为新特性测试单独部署的环境，数据与测试环境相同

⁴ 专为发布之前和热修复测试单独部署的环境，数据与生产环境相同

⁵ Git 中的提交允许有多个父提交和多个子提交

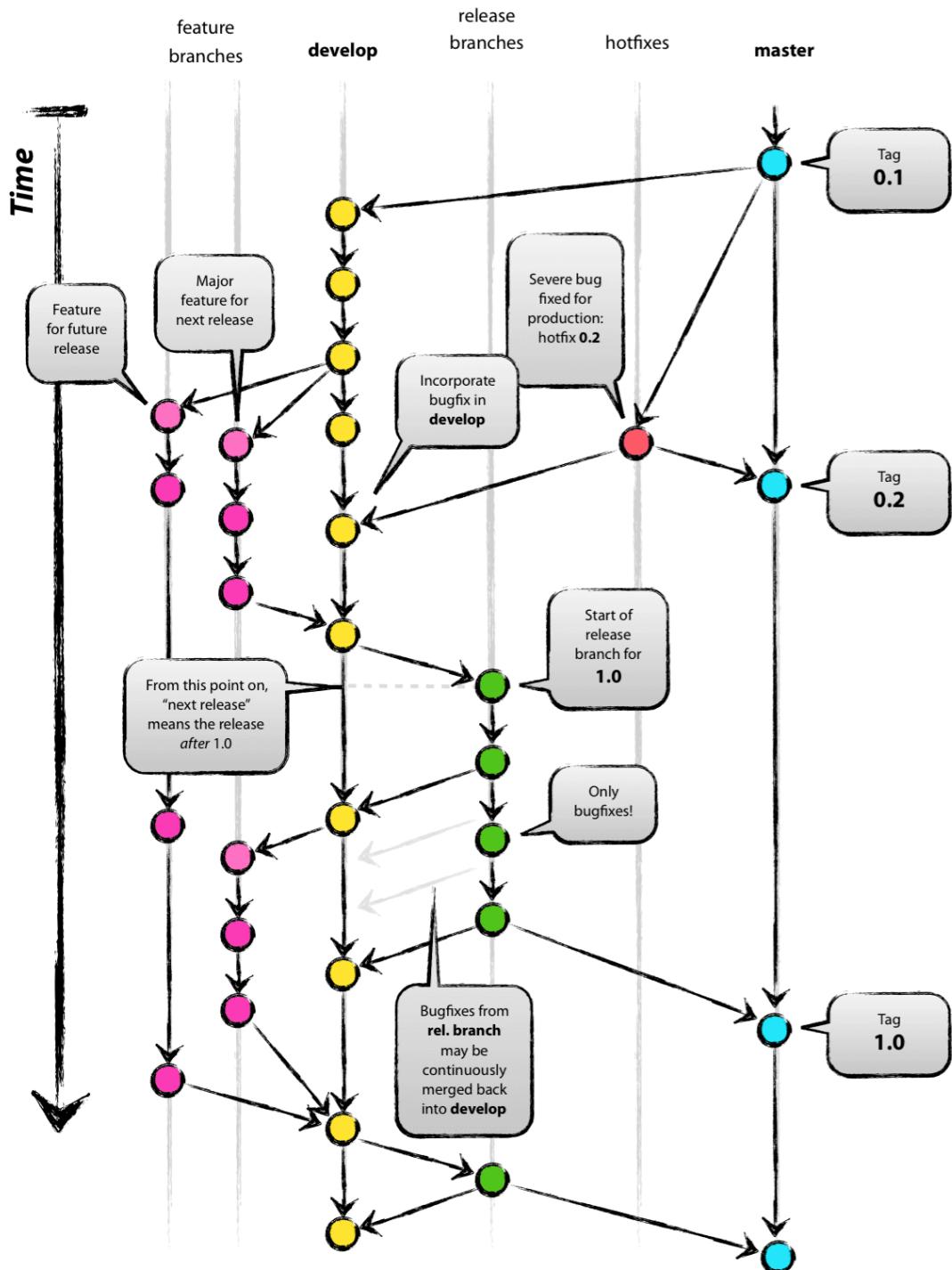


图 2-3 Git-flow 工作机制示意图
Fig 2-3 Git-flow working mechanism

3. 这时开发人员甲从开发分支分出了一个早就想做但不是很急的特性分支，我们把它叫做“未来特性” (Feature for future release)；
4. 紧接着产品经理说有一个加急特性要加上，于是开发人员 A 暂且放下“未来特性”的工作，又从开发分支分出了第二个特性分支，暂且叫做“下一个特性” (Major feature for next release)，这是一个典型的多特性并行开发的情况；
5. 与此同时生产环境可能发现了一个 bug，开发人员乙从主分支分出一个热修复分支 0.2，修复完 bug 后先合并到了主分支打上标签 0.2，再合并到了开发分支，在开发分支上产生了一个新的提交 (Incorporate bugfix in develop)；
6. 又过了一段时间，开发人员甲完成了“下一个特性”，合并到了开发分支上，回头继续开发“未来特性”；
7. 开发人员乙从 develop 分出了一个预发布分支 1.0，部署到准生产环境，经过开发人员的测试发现了一个 bug，开发人员乙修复并合并回了开发分支；
8. 这时产品经理说又有两个新特性要加上，但打算和“未来特性”一起发布，于是开发人员甲又从开发分支分出一个新特性分支并同时开发这两个新特性；
9. 开发人员乙在预发布分支 1.0 上面又修复了后来发现了几个 bug，其中有几次反复地合并回了开发分支，为了其他开发人员能更好地开发；
10. 到了正式发布的时机了，开发人员乙把预发布分支先合并到了开发分支，再合并到了主分支，并在主分支上打上标签 1.0；
11. 然后开发人员甲完成了两个新特性，开发人员乙又发布了一次，虽然图上没有，但之后应该会再把主分支合并到开发分支；

本系统开发过程中统一使用了 Git-flow 来管理分支，如图 2-4 所示，Azwraith 项目在 4 月 21 日到 26 日之间的表现就比上图更加复杂，但仓库管理过程依然方便快捷。其中黑线是主分支，紫色线是开发分支。

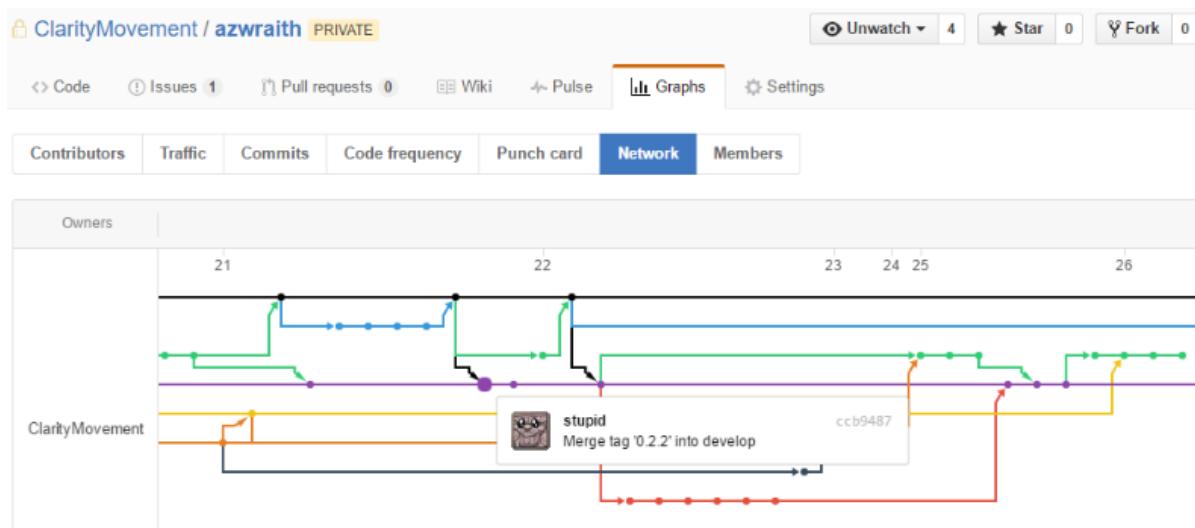


图 2-4 Azwraith 分支网络
Fig 2-4 Azwraith branch network

2.3.2 IDE: Webstorm VS Atom

Webstorm 是 JetBrains 公司开发的一系列集成开发环境 (IDE) 中面向 JavaScript 的一款收费桌面应用。JetBrains 公司在像 Java 这样的静态语言语法分析方面经验丰富，像 JavaScript 这样的非静态语言也处理的非常好。Atom 是一个由 Github 开发的一款免费的开源的文本和源代码编辑器，面向 OS X、Linux 和 Windows 用户，支持用 NodeJS 编写的插件并且内置了 Git 版本控制。Atom 也有一个强大的社区维护者免费的扩展包。本系统开发过程中，只在版本管理模块的前端使用了 Atom，其他部分均使用了 Webstorm。因为 Atom 比较轻量级并且 Atom 的社区提供的各种扩展可选择性很强，最重要的一点是 AngularJS 使用的是依赖注入，不像 ReactJS 和 NodeJS 有大量的模块互相引用，因此不需要 IDE 有很强的项目文件关系处理和跨文件变量解析能力。

2.3.3 代码生成：Yeoman 和 Yeoman Generators

Yeoman 是一个开源的客户端开发工具，包括一系列帮助开发人员搭建 WEB 应用程序的工具和框架。Yeoman 运行起来是一个命令行交互界面，工具开发人员利用它来和应用开发人员交互（或者不进行交互）获取一些信息后运行用 NodeJS 写的一些功能函数，如生成一个 WEB 应用程序的启动模板、管理依赖、运行单元测试、提供一个本地开发服务器和优化生产环境代码等。

generator-material-app¹是一个生成 Material Design 风格的 MEAN 应用脚手架（基本框架）的 Yeoman 代码生成器。

本系统在开发时，主要是版本管理模板因为是一种比较成熟的后台管理系统，使用了代码生成器来生成初始的结构和开发过程中的模块，这极大地减少了重复劳动，增加了代码可维护性，解放了开发人员的生产力。

2.3.4 构建工具：Grunt VS Gulp VS Npm

构建工具是用来构建各种自动化任务的工具，比如编译、测试等操作，有些需要一系列复杂的指令或者逻辑，那么就可以用构建工具来构建一个“自动化任务”，之后只要用这个构建工具来运行这个任务，就能完成这项操作。现在 JavaScript 世界主要使用 3 种主流的构建工具 grunt、gulp 和 npm：

grunt 使用基于声明式配置的配置文件来定义任务，自定义插件、社区强大、简单易用，不需要任何高级的概念或知识。但由于它基于配置，一旦任务数量多了或者复杂了，grunt 配置文件将会变得特别复杂而且难以控制任务执行的顺序。

gulp 使用基于流处理的 JS 代码来定义任务，设计之初就是想要取代 grunt，自定义插件、可扩展性强、只需要了解 JavaScript 就可以自定义任务，和 grunt 一样依然依赖插件和插件作者。

npm 使用 NodeJS 和 bash 脚本来定义任务，不依赖插件和插件作者，可扩展性最强，自由度最高，但需要熟悉 NodeJS 和 bash 脚本。

这三种构建工具从上到下，使用难度增大、可扩展性变强、自由度变高。本系统在选择构建工具时颇有一番斟酌，最终考虑到 npm 的难度，在大部分系统中使用了 gulp 来构建，只在 Smart City 模块的前端使用了 npm 以作试点。

¹<https://github.com/michaelkrone/generator-material-app>

2.3.5 代码质量

代码质量指的是软件的结构性质量，包含一些与功能需求无关的代码如何组织、软件是否正确生产的特性，如可读性、健壮性和可维护性。

本小节介绍了一系列提高代码质量的手段和工具：代码风格指南、代码风格检查器、Git 钩子机制（Git hooks）和代码审查（code review）。其中后者是前者的保障或备案。

2.3.5.1 代码风格指南：John Papa 和 Airbnb

风格指南（style guide）是一种编写和设计文档的标准集，不管是大众用途的还是某一个特殊领域的，比如代码编写。遵从统一的代码风格指南可以提高代码的一致性从而提高代码的可读性，一个代码风格指南一般是由一些权威的人士或者公司总结了一些最佳实践（Best practice）来编写出来的，所以能够提高代码质量，避免或减少常见错误。

本系统中使用了两种风格指南，一个是 AngularJS 的比较权威的“John Papa 的风格指南”¹（John Papa's Styleguide），另一种是 ReactJS 和 NodeJS 的“Airbnb²风格指南”³。

当然风格指南只是一种指南，并不能保证每个开发人员至始至终遵守，人也没有精力准确地判断每一段代码是否符合风格指南，还需要一些工具来保证代码的风格一致性。

2.3.5.2 代码风格检查器：JSCS、Jshint/ Eslint 和 Stylelint

检查代码风格的一个主要工具是各种代码风格检查器，以下列出本系统中使用的各类地代码风格检查器：

- JSCS 是一种针对 JavaScript 的语法层面的代码风格检查器，并有一定的自动修复功能，没有默认的规则要求，就是说如果不指定规则，JSCS 什么都不会检查。
- Jshint 是一种针对 JavaScript 的语法及语义层面的代码风格检查器，一般与 JSCS 混合使用，专注检查语义，有一些默认的规则要求。
- Eslint 和 Jshint 类似，有着最好的 ES6 支持，一般也与 JSCS 混用，有一些默认的规则要求。
- Stylelint 是一种针对 CSS 的代码风格检查器，没有默认的规则要求。

这些代码风格检查器都通过一个简单的配置文件.jscsrc、.jshintrc、.eslintrc、.stylelintrc，里面列有一系列的规则来自定义想要的各种代码风格。也可以通过简单的 preset 选项来使用某一个内置的代码风格，然后可以增加一些规则来修饰。如果在某一个单独的文件中想要突破规则也可以暂时地修改规则或者关闭规则。

这里以 JSCS 为例，本系统在使用“John Papa 的风格指南”时是完全自己定义，因为太长所以不在这里展示，而在使用“Airbnb 代码风格”时使用的是 preset 选项再加上一些自己的修改：

代码 2.1 Airbnb 代码风格 JSCS 配置

```
1 {  
2   "preset": "airbnb",
```

¹<https://github.com/johnpapa/angular-styleguide>

²美国一家做租房的公司，相当于租房业的 Uber

³<https://github.com/airbnb/javascript>

```
3 "esnext": true,
4 "excludeFiles": ["build/**", "node_modules/**"],
5 "validateQuoteMarks": null,
6 "requireCurlyBraces": ["else", "for", "while", "do", "try", "
    catch"]
7 }
```

虽然在这里都罗列出来介绍，但其实 Jshint 和 Eslint 是可选的，AngularJS 不需要 ES6 支持，所以在版本管理模块中使用了 Jshint，而且其他两个模块中使用了 Jshint。

有了代码质量检查器也不能保证开发人员会去使用，或者开发人员是同意使用的，但总是会有忘记和疏漏的时候。这时就需要 Git hook 来保障代码质量检查器的运行。

2.3.5.3 Git hooks 和 Pre-commit

Git hook 是 Git 提供的一种钩子（hook）机制，每当你使用 Git 做某种操作时就会触发来执行一段脚本，比如本系统中就是用了预提交钩子（**pre-commit hook**），顾名思义就是在提交之前会执行的钩子。

本系统中使用了一个开源的封装 **pre-commit**，配合 npm 脚本（scripts）只需要安装和在 **package.json** 中简单的配置就能在每次 commit 之前运行我们的代码风格检查器：

代码 2.2 package.json 中的 pre-commit 设置部分

```
1 [
2   "...": "...",
3   "scripts": {
4     "eslint": "eslint src tools",
5     "jscs": "jscs src tools --verbose",
6     "stylelint": "stylelint 'src/**/*.{scss}' --syntax scss"
7   },
8   "pre-commit": [
9     "eslint",
10    "jscs",
11    "stylelint"
12  ]
13 ]
```

然而 Git 又提供了一些方法来突破钩子机制直接提交，毕竟有可能会发生恶意篡改代码风格检查器配置或者没空修复代码风格的紧急情况，所以 Git hook 也不是最终的保障。

2.3.5.4 Code review

一台机器可以做五十个普通人的工作，但是没有哪部机器可以完成一个伟人的工作^[10]。最终的检查还是要靠人，代码审查（code review）是一种对软件源代码的系统性检查，目的是找出在开发阶段倍忽视的错误和提高整体的代码质量。代码质量经常能够发现一些常规的安全漏洞来提高软件的安全性，如格式字符串漏洞、竞争情况、内存泄漏和缓冲区溢出等。在线代码仓库如 git、svn¹等允许团队协作地进行代码审查，每当有一部分代码想要合并到主要分支，代码贡献可以发起一个合并请求（merge request）并制定团队中的一个开发人员为审查者，审查者可以在线（意味着不需要开发环境）看到每个提交或者整体的逐行差异（difference）、可以在每一行代码上面进行注释，理论上直到每一行被注释的代码都被修改后才能请求再次审查或者进行本次合并，以上过程可以重复多次直到审查者彻底满意。

本系统在开发过程中，把代码审查和前文提到的 Git flow 结合，每个新特性分支、热修复分支和预发布分支在合并到主要分支之前，都需要先发出一个合并请求（git 上叫做 pull request），然后指定团队中的一个人从客观的角度审查每一行修改，如果新增的代码或改动能让一个不够了解这个项目、这个特性的人也能看懂其意图，那么证明它的可读性和可维护性是足够高的，从而保障了主要分支代码的高质量。

2.3.6 单元测试：Karma、Mocha、Chai 和 Sinon

本系统中的单元测试使用 Karma 作为测试环境，Mocha 作为测试框架，BDD 风格的 Chai 作为测试语言，使用 Sinon 来模拟假的 JavaScript 对象和函数。

2.3.6.1 Karma

Karma 是 AngularJS 开发团队研发的一个专门用于跑测试的系统，主旨在于给开发者提供有生产力的（productive）测试环境。开发人员不需要设置复杂的配置载入，只需要写测试、写代码然后获得及时的测试反馈，这样开发人员才能够既有生产力（productive）又有创造力（creative）。

2.3.6.2 Mocha

Mocha 是一款功能全面的测试框架，既能在 NodeJS 中运行也能在浏览器中运行，能够很好地支持异步测试。Mocha 串行地执行测试程序，给出灵活切准确的报告，而且能够把未捕捉到的异常映射到正确的测试用例上面。

2.3.6.3 Chai

Chai 是一个开源的、测试驱动开发（TDD）或行为驱动开发（BDD）的断言库或者叫语言库，它运行在 NodeJS 或者浏览器中，可以轻易地与各种 JavaScript 测试框架结合。链式的 BDD 风格（包括 should 和 expect 两种风格）提供了富有语言表现力和可读性的代码：

代码 2.3 BDD 风格的 chai 代码

¹类似 git 的一个版本控制系统

```
1 foo.should.be.a('string');
2 foo.should.equal('bar');
3
4 expect(foo).to.be.a('string');
5 expect(foo).to.equal('bar');
```

而 TDD 的断言风格给人一种经典的感觉

代码 2.4 经典的断言风格

```
1 assert.typeOf(foo, 'string');
2 assert.equal(foo, 'bar');
```

2.3.6.4 Sinon

Sinon 是一个独立的、与测试框架无关的、开源的 JavaScript 库，专注于做三种用于模拟测试的类或者对象：

间谍 (spy) 包装过的类或者对象，拥有正常的行为，但每一个动作都被监听

桩 (stub) 假的类或者对象，直接模拟一些指定的行为，给出特定的、看起来正确的结果

模 (mock) 完全的假的类或对象，任何行为直接返回 0 或者空

这一套测试技术栈是目前比较主流的一种，另一种是 Karma+Jasmine，Jasmine 比较全面地提供了相当于 Mocha+Chai+Sinon 的功能，但本系统出于灵活性的考虑本系统还是没有选择 Jasmine。

2.3.7 端对端测试：Protractor

Protractor 是一个端对端测试框架，它用一个真正的浏览器运行 WEB 应用，像一个真正的用户一样与应用进行交互，它直接支持 AngularJS 应用。Protractor 使用遵从 W3C WebDriver specification¹的 WebDriverJS 来模拟用户会触发的原生事件和浏览器相关的驱动来与 WEB 应用交互。Protractor 测试直接就是异步的，会自动把测试任务一个一个连起来，开发人员不再需要手动在测试中加等待和睡眠。Protractor 支持 AngularJS 独有的查找器策略，用来测试 Angular 的页面元素时，不需要开发人员有任何额外的配置。本系统使用 Protractor 主要就是考虑到它对 AngularJS 的支持。

2.3.8 持续集成：Solano CI

Solano CI 是一个持续集成和部署解决方案。可以直接使用在线的 SaaS 云服务，也可以在私有虚拟设备上搭建主机。Solano CI 独有的时间表 (schedule-optimizing) 优化技术使用测试文件的多重输入来计算最好的运行时间，使其能够自动地、智能地并行执行自动化软件测试。开发人员只需要简单的配置就能够轻松地完成持续集成和自动部署，本系统中的持续集成使用情况会在后面的专门章节介绍。

本系统选择使用 Solano CI 主要是因为它的速度很快而且和对 AWS 云服务的支持比较好，另一个经济原因是它比其他一些更大的持续集成工具便宜。

¹一种与所有主流 WEB 浏览器兼容的、平台化、语言中立的接口标准

第三章 需求分析

本系统分为 3 个共享数据库的互相关联又相对独立的模块，需求分析的工作不是仅仅做三遍就行，而是要考虑到相互联系和共用资源。由于整体上是传感器管理系统，所以传感器（Sensor）这个资源肯定是共享的，不同的系统都得有登录和权限管理，所以用户（User）也需要共享，这两种资源需要统一在版本管理系统中管理，这样能够保证所有用户和设备资源的一致性，又能在此基础上各自独立地实现业务逻辑。不同系统上的用户可能实际含义不太相同，本系统使用用户的角色（role）属性来区分，具体角色有：

普通用户（user） Smart Home 模块和 Smart City 模块的最终用户，一个传感器会属于一个普通用户，一个普通用户可以拥有多个传感器。

管理员（admin） 版本管理模块的最终用户，也就是 Clarity 的硬件开发人员和管理人员，除了管理传感器版本等信息外还可以管理用户。

根用户（root） 超级用户，给系统维护人员或开发人员的角色，拥有最高权限，可以修改任何资源。

注 1：本系统中版本管理模块取名“balanar”，Smart Home 取名“robotic”，Smart City 名字叫“azwraith”。这 3 个名字都只做内部代号，但也许会在一些代码或截图中出现，为免误解，特此声明。其中只有 robotic（意为机器人的）跟项目内容有点联系，其余两个无实际含义。

注 2：本章节中的原型和设计稿都不一定与最终的需求吻合，因为可能有些需求修改没有必要在原型和设计稿上体现。

下面本章将详细从不同的角度对各模块进行需求分析：

3.1 版本管理模块

本模块的主要用户是硬件开发人员，原本用户使用 Excel 记录传感器版本、批次、拥有者等信息，但随着传感器的不断改进，合作方越来越多，制作给合作方的特殊版本越来越多，零件进货的批次各有不同，使用 Excel 的时间成本和精力成本越来越高。

3.1.1 功能需求

所以版本管理模块需要解决的问题和实现的功能如下：

1. 版本、批次和传感器之间的关系混乱，包括硬件版本、固件版本、软件版本、五种零件版本和五种零件批次；Balanar 要理清关联关系，用最佳的数据结构体现；
2. 版本编号格式难以维护；Balanar 要能够自动校验格式并给出错误描述；
3. 硬件版本和固件版本兼容，固件版本又和软件版本兼容；Balanar 要能够记录和判断版本兼容性；

3.1.2 用户用例

经过与需求方的多次需求会议，本课题总结出如下用户用例：

1. 每当用户（这里用户指 Clarity 的硬件开发人员）设计出新版本的零件，需要到系统中添加一个零件版本，填写一些参数。
2. 每当用户下单订购或者收到一批零件，需要到系统中添加一个零件批次或者修改到货日期，选择相应的一种零件版本。
3. 每当用户准备装配一种拥有新版本零件的传感器，需要到系统中添加一个硬件版本，选择相应的五种零件版本。
4. 每当用户准备装配一种拥有新固件的传感器，需要到系统中添加一个固件版本，并添加对应的兼容性。
5. 每当用户装配完成一批新的传感器，需要到系统中添加这些传感器，选择相应的硬件版本、固件版本和五个零件批次，同时可以指定一个拥有者（普通用户）。
6. 每当 Clarity 有新的合作伙伴，Clarity 的管理人员可以添加普通用户（角色），将来有给他们专用的传感器时，可以指定为拥有者。

3.1.3 数据模型

本课题使用 ARIS 业务流程建模软件建立了一个数据模型，如图 3-1 所示，其中灰色的 Sales 是不在本模块也不在本系统中的：

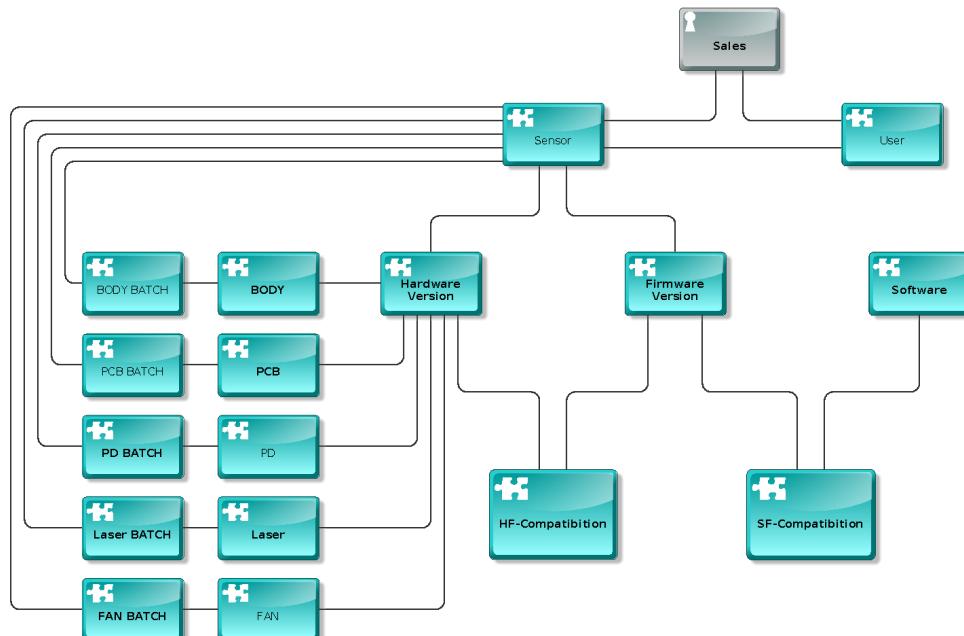


图 3-1 版本管理模块数据模型
Fig 3-1 Version Management module's data model

3.1.4 数据字典

本课题使用普通文本定义了一套数据字典，以免开发过程中混淆不同的资源名和属性名，同时对字段类型做了一些限定，这里只展示了开头部分关于传感器资源的定义，详情请看附录 A (数据字典)。

代码 3.1 版本管理模块的数据字典

```
1 Explanation:  
2 @: description or restriction of keys or attributes  
3 @fk: short for 'foreign key'  
4 @pk: short for 'primary key'  
5 @uk: short for 'unique key'  
6 @path(type): a file or directory path in cdn of specific type  
7 ...: a combination of object, e.g. {a, b, ...{c, d}, e} == {a, b, c  
     , d, e}  
8 ': {}': attributes of a table or an attribute  
9 ': type()': describe type of an attribute, including [string,  
           integer, decimal, enum]  
10 -----Tables-----  
11 Sensor: {  
12   @pk id  
13   @fk hardwareVersion  
14   @fk firmwareVersion,  
15   @fk componentBatches: {body, pcb, photodiode, laser, fan},  
16  
17   threshold: type(integer),  
18   noiseLevel,  
19   calibrations: {  
20     mass: type(decimal),  
21     number: type(decimal)  
22   },  
23   applicationType: type(enum(SMART_CITY)),  
24 }
```

3.1.5 快速原型

本系统在正式开发之前还使用一款叫做 balsamiq 模拟软件做了一个快速原型。如图 3-2 所示，这里只展示了一个页面，详情请看附录 B (快速原型)。

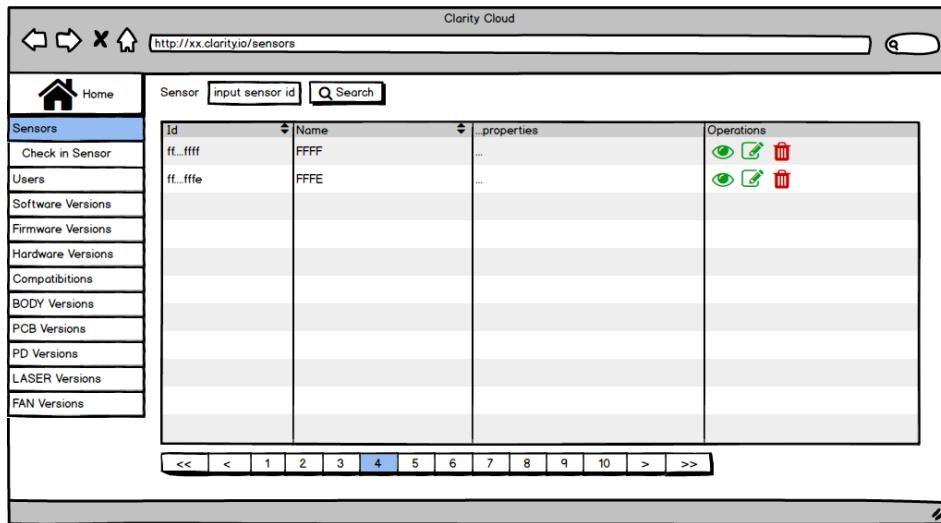


图 3-2 版本管理模块原型样例

Fig 3-2 Example of Version Management module's mockups

3.2 Smart Home 模块

本模块的当前的主要目标客户是一家日本的家电制造企业，最终用户是该企业家电产品的用户。该企业生产的产品是空调和空气净化器等家用电器，因此 Clarity 为其量身打造了智能家居（Smart Home）解决方案。

3.2.1 智能家居解决方案

首先，目标用户是购买了该企业各种家电和空气净化器的个人或家庭，用户还可以随身携带一个 Clarity 的个人空气质量传感器。普通的空气净化器只能一直开着或者在人的操控下间歇性地净化空气，而搭载空气质量传感器的家电也只能检测空气质量并报告给用户，并不能实际改善空气质量，Clarity 在两者之间引入了一个烧录了智能程序的可编程扫地机器人巧妙地弥补了两者的不足。此智能家居解决方案按以下流程工作：

1. 不同房间的空气质量传感器持续上传数据到服务器，服务器保存数据；
2. 服务器定时地计算并评估空气质量的好坏；
3. 一旦有房间空气质量低于一定水平，服务器就会给机器人下达净化指令，机器人会移动到对应的房间并开启其上搭载空气净化器；
4. 空气质量好转之后，服务器再给机器人下达停止指令，机器人会关掉空气净化器原地待命；

3.2.2 功能需求

1. 实时展示个人、家里和城市的空气质量情况和健康评价；
2. 实时展示家里不同房间的空气质量状况；
3. 实时展示机器人的位置和空气净化器的工作状态；
4. 能够绘制每个房间的历史空气质量折线图；
5. 能够让用户手动控制机器人；

3.2.3 用户用例

1. 用户随时查看功能需求中展示的所有实时信息；
2. 用户想要手动控制机器人时，可以调节到手动模式；
3. 用户可以控制机器人去任何房间并自动开始净化空气；

3.2.4 设计稿

本模块的需求方给出了一个比较详细的设计稿，如图 3-3 所示，这里只展示了一个页面，详情请看附录 C（设计稿）。

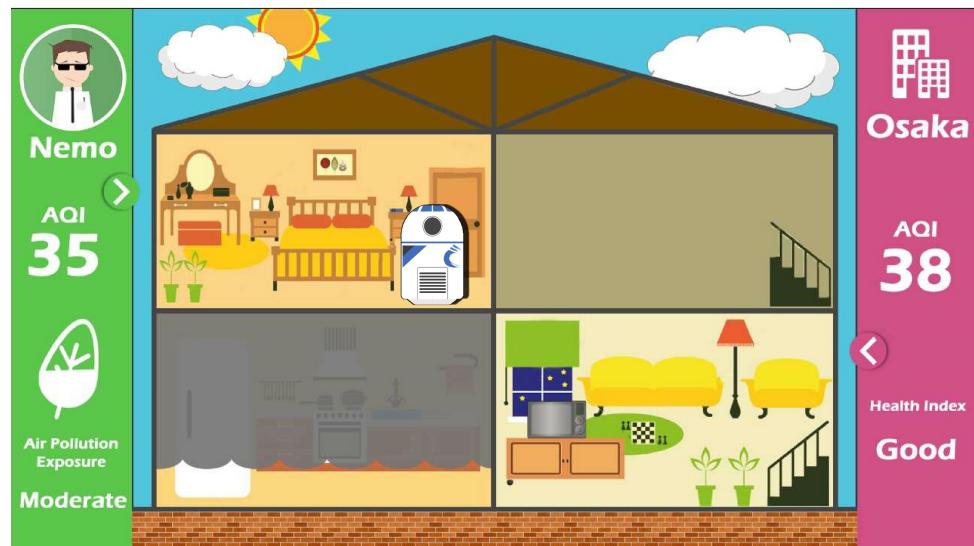


图 3-3 Smart Home 模块设计稿样例
Fig 3-3 Example of Smart Home module's design

3.3 Smart City 模块

本模块的当前主要目标客户是美国一家城市政府，最终用户是政府工作人员。该城市计划大量安装 Clarity 的传感器以监控城市空气质量，这就需要在智慧城市级别提供空气质量的数据查看和设备管理服务。

3.3.1 功能目标

本模块需要提供一下功能：

1. 能够添加和删除、显示和隐藏设备（传感器）；
2. 以地图形式直观的展示传感器位置和当前空气质量；
3. 以图表形式直观地展示某些设备的最近历史数据；
4. 以 csv 形式提供不同设备、时间跨度、时间精度和空气质量度量¹的空气质量数据；

3.3.2 用户用例

1. 市政部门安装了新的传感器或者拆除了旧的传感器，需要在此系统上添加或删除设备。
2. 政府工作人员随时查看设备列表、传感器位置和历史数据。
3. 政府工作人员想要下载某几个设备的最近历史数据，可以在设备列表上选择一些设备，在图表中选择时间精度和度量，然后在图表的工具栏中点击下载。
4. 政府工作人员想要下载某一个设备的长期历史数据，可以在一个单独的下载页面上，在表单中选择时间跨度、时间精度和度量，然后点击下载。

3.3.3 快速原型

本系统在开发之初做了一个简单的快速原型，如图 3-4 所示。

3.3.4 设计稿

但后来因为这个模块比较重要，而且不像版本管理模块不是非常注重 UI，公司又请专门的设计师给出了一个设计稿，如图 3-5 所示。

¹如 AQI 或者 pm2.5 浓度

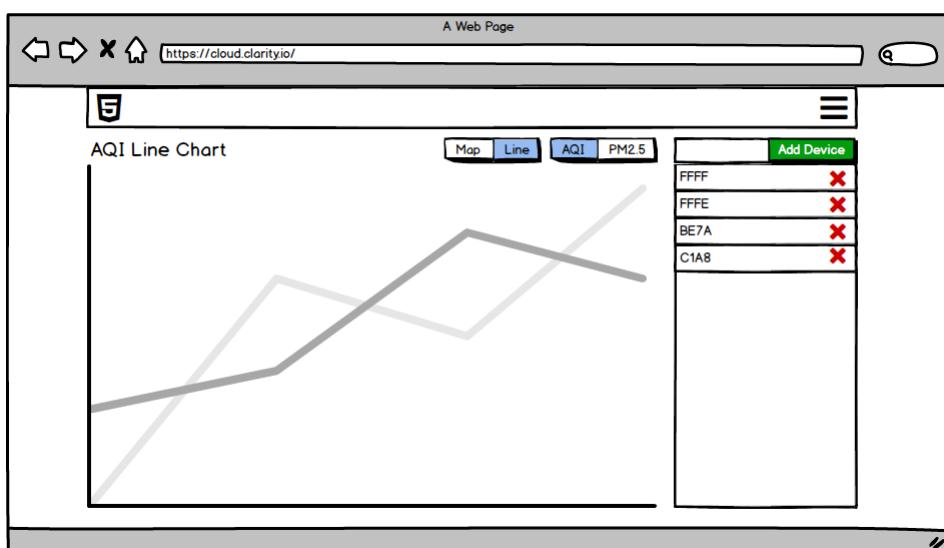
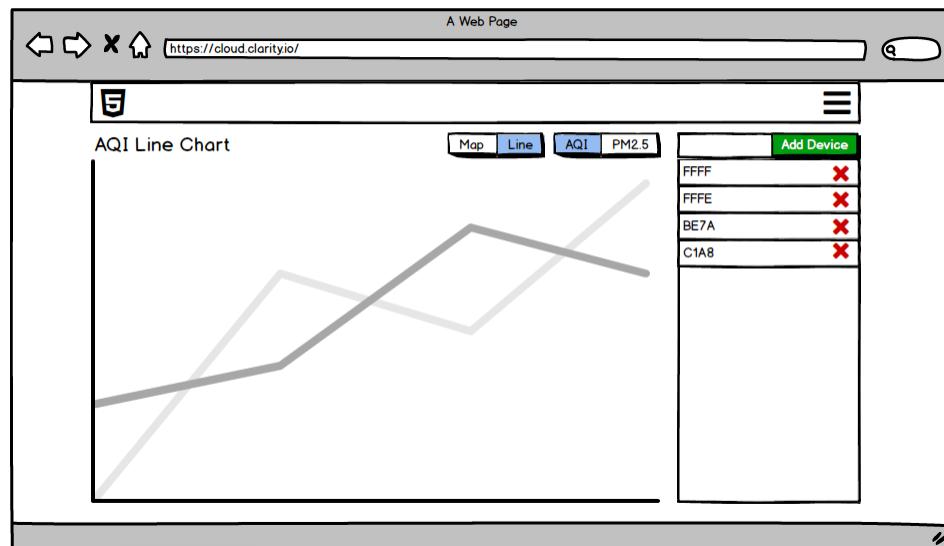


图 3-4 Smart City 模块原型
Fig 3-4 Smart City module's mockups

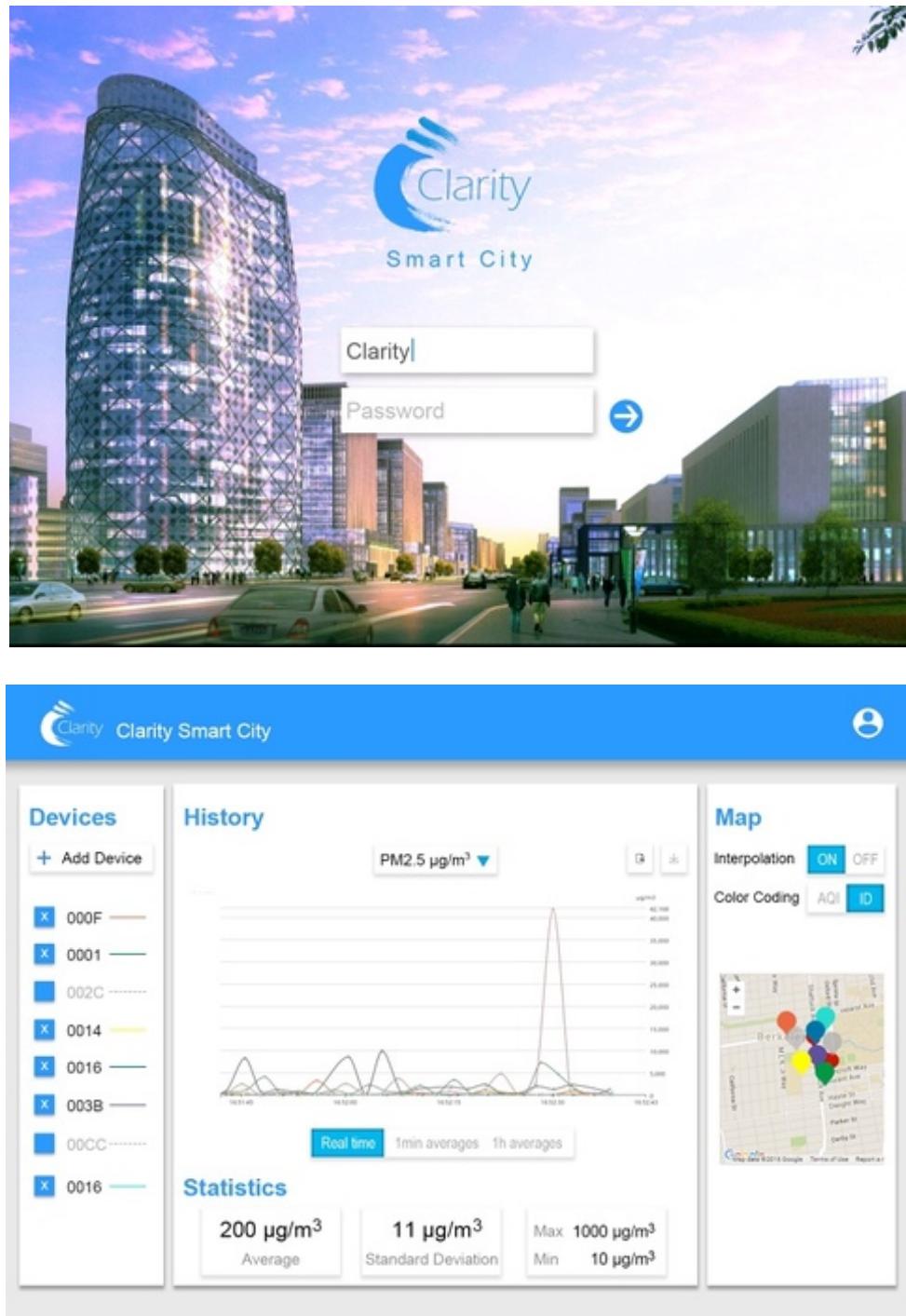


图 3-5 Smart City 模块设计稿
Fig 3-5 Designs of Smart City module

第四章 系统架构与开发实现

本系统基于敏捷开发、响应式设计、Flux 架构模式等理论指导，在简洁且充分的需求分析的基础上，使用全栈 JS 技术和各类高效的开发工具开始了系统设计与开发。三个模块使用的各类理论、技术和工具互有重叠：

- 在开发模式方面，都是使用的敏捷开发，频繁部署新版本的同时不断地根据需求和反馈迭代开发。
- 在 UI 设计和页面布局方面，都使用了 Flex 布局和 google 的 Material design，程序员能够方便地控制页面布局，同时页面也简洁美观富有质感。
- 在 API 设计方面，各有一套 RESTful 风格的 API 让 APP 能够方便地增删改查各类资源，同时 WebSocket 的使用让 APP 能够及时地获得数据更新。
- 在权限校验方面，在 Oauth 和 JWT 的帮助下，用户能够安全、方便地访问我们的系统，将来系统也很容易加上单点认证系统或者接入其他第三方平台。
- 在版本控制和代码风格方面，都使用了 Git 和 Git flow 来控制版本和发布，使用了 JSCS、stylelint、pre-commit 和代码审查来保证代码风格的一致。
- 在测试方面，都选择了 Karma+Mocha+Chai+Sinon 的技术栈，加上 Solano CI 来持续集成和自动部署。

4.1 版本管理模块

本模块使用基于比较成熟的 MEAN 技术栈，即 MongoDB+ExpressJS+AngularJS+NodeJS。本模块比较特殊的一点在于使用了代码生成器，本文作者也是这个项目的主要代码贡献者之一。另外本模块在页面布局方面额外使用了响应式设计来使得页面能够适应所有大于平板的屏幕宽度，使用了 gulp 作为构建工具，使用 jshint 来检查语法，前后端分别使用了 Atom 和 Webstorm 来编辑源代码。

下面将主要以用户和传感器两种资源为例，按照资源从后向前传递、从整体到细节展示的顺序，从 API、API 路由、客户端模型定义和响应式设计几个方面介绍本模块的详细设计与开发，另外还特别介绍了本模块使用的代码生成器，最后从 Git 代码提交记录的角度介绍了本模块和代码生成器的关系。

4.1.1 API

本模块使用 RESTful 风格的 API，包含新建、删除、修改、列表查询和单一查询五种 API，围绕 Mongoose 的数据模型来进行增删改查等业务，同时使用 WebSocket 在保存和删除的同时发送消息给前端，让前端及时获悉改动。如图 4-1 所示，这是传感器 API 完整的目录结构，其中包含 controller（业务逻辑控制器）、model（Mongoose 数据模型）、params（查询参数）和 socket（WebSocket 事件注册）以及两个测试文件。

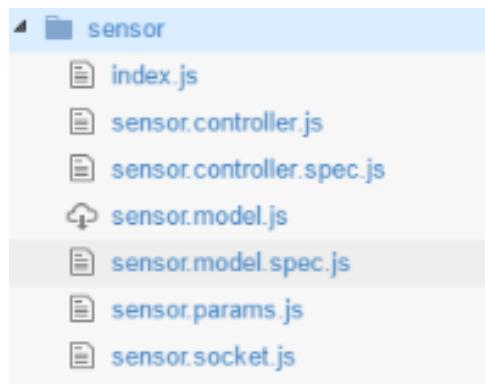


图 4-1 传感器 API 目录结构

Fig 4-1 Directory of Sensor's API

这里的业务逻辑控制器里面其实没有真正的增删改查逻辑，因为增删改查等操作是每个资源的 API 都应该有的，所以它们被泛化到了控制器的基类 CrudController 中。如代码 4.1 所示，CrudController 是一个构造函数，接收一个模型（model）作为参数，所以不同的业务逻辑控制器可以传入各自的 model 就可以使用基类的增删改查函数来注册 API 了。

代码 4.1 传感器 API 定义

```
1 /* in crud.controller.js */
2 function CrudController(model) {
3     // set the model instance to work on
4     this.model = model;
5 }
6 /* in sensor.controller.js */
7 function SensorController(router) {
8     // call super constructor
9     CrudController.call(this, Sensor, router);
10 }
11 /* in sensor/index.js */
12 // register sensor routes
13 router.route('/')
14     .get(controller.index)
15     .post(controller.create);
16
17 router.route('/' + controller.paramString)
18     .get(controller.show)
19     .delete(controller.destroy)
20     .put(controller.update)
21     .patch(controller.update);
```

如代码 4.2 所示，本模块在 socket 文件中给传感器模型的保存和删除操作添加了钩子（hook），一旦保存或删除则用 WebSocket 发送'sensor:save' 或'sensor:remove' 的消息并附上保存完或删除前的对象，用以通知前端。

代码 4.2 传感器资源的 Socket 注册函数

```

1 function registerSensorSockets(socket) {
2   Sensor.schema.post('save', function (doc) {
3     socket.emit('sensor:save', doc);
4   });
5
6   Sensor.schema.post('remove', function (doc) {
7     socket.emit('sensor:remove', doc);
8   });
9 }

```

如代码 4.3 所示，传感器的数据模型通过一个可嵌套的对象来定义。

代码 4.3 传感器的 Mongoose 数据模型

```

1 var SensorDefinition = {
2   name: {type: String, required: true},
3   hardwareVersion: {type: ObjectId, ref: 'HardwareVersion'},
4   firmwareVersion: {type: ObjectId, ref: 'FirmwareVersion'},
5   componentBatches: {
6     body: {type: ObjectId, ref: 'ComponentBatch'},
7     pcb: {type: ObjectId, ref: 'ComponentBatch'},
8     photodiode: {type: ObjectId, ref: 'ComponentBatch'},
9     laser: {type: ObjectId, ref: 'ComponentBatch'},
10    fan: {type: ObjectId, ref: 'ComponentBatch'},
11  },
12  threshold: {type: Number, required: true},
13  noiseLevel: {type: Number, required: true},
14  calibrations: {
15    mass: Number,
16    number: Number
17  },
18  applicationType: String,
19  info: String,
20  active: Boolean
21 };

```

4.1.2 API 路由

在 AngularJS 中一个页面由四份代码渲染而成，分别是 HTML 代码、样式代码、控制器代码和路由器（router）代码。在本模块中把这四份代码分别写在一个文件中，合起来的目录结构叫做一个路由（route）；把一套创建、列表、详细和编辑的页面叫做 API 路由（apiroute），因为它是一套与增删改查 API 相对应的路由的集合。一个 API 路由中除了增删改查的基本路由，还有两个抽象路由 list 和 sensor，因为它们的控制器里面几乎没有业务逻辑，HTML 代码也不包含任何实际有意义的内容，只是在路由器层面组织起这个结构，特殊的是在这里 list 路由作为一个插入点对传感器的 WebSocket 进行了监听，在列表中及时地更新被创建、修改或删除的内容。一个 API 路由的额外包含一个资源服务（resource service），提供与 API 交互的所有增删改查函数。

如图 4-2 所示，这是传感器 API 路由完整的目录结构，其中 sensor.html、sensor.scss、sensor.controller.js 和 sensor.js 文件分别对应上述四份代码，main、create、detail、edit 和 items 都分别是一

个完整的路由，`sensor.service.js` 就是用来定义传感器的资源服务的文件。在这里 `main` 路由是 `create` 路由和 `list` 路由的入口，不作过多介绍。

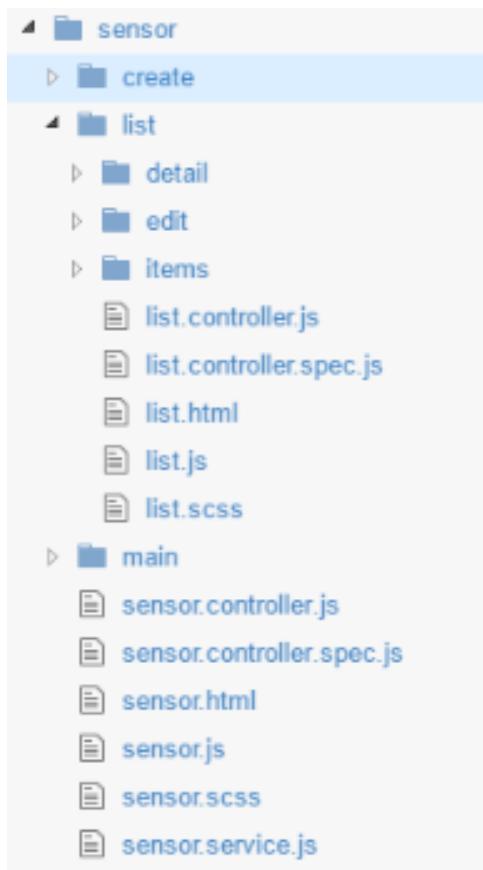
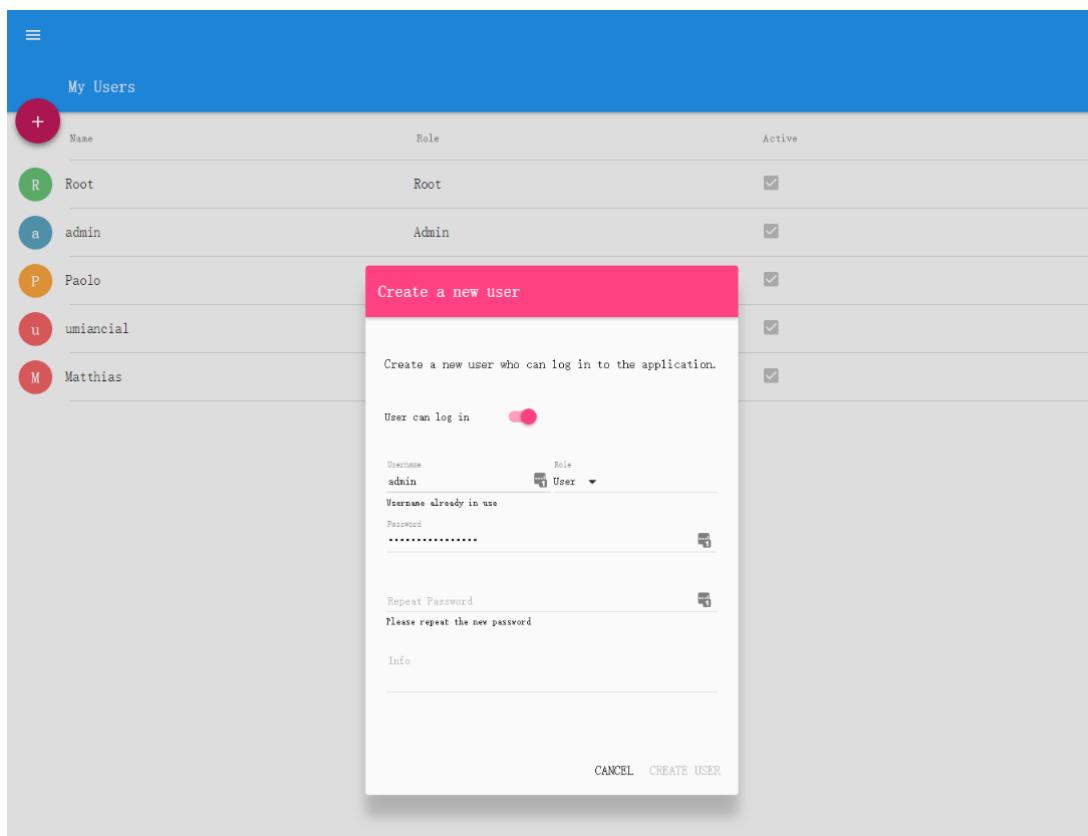


图 4-2 传感器 API 路由目录结构
Fig 4-2 Directory of Sensor's apiroute

这里以用户页面为例，展示 API 路由的全貌，如图 4-3 所示，这套设计让 `create` 页面以一个弹窗（modal）的形式浮在 `list` 上方，`detail` 页面从 `list` 右方弹出；如图 4-4 所示，一旦有修改资源，会有 `toastr`¹ 窗口提示。

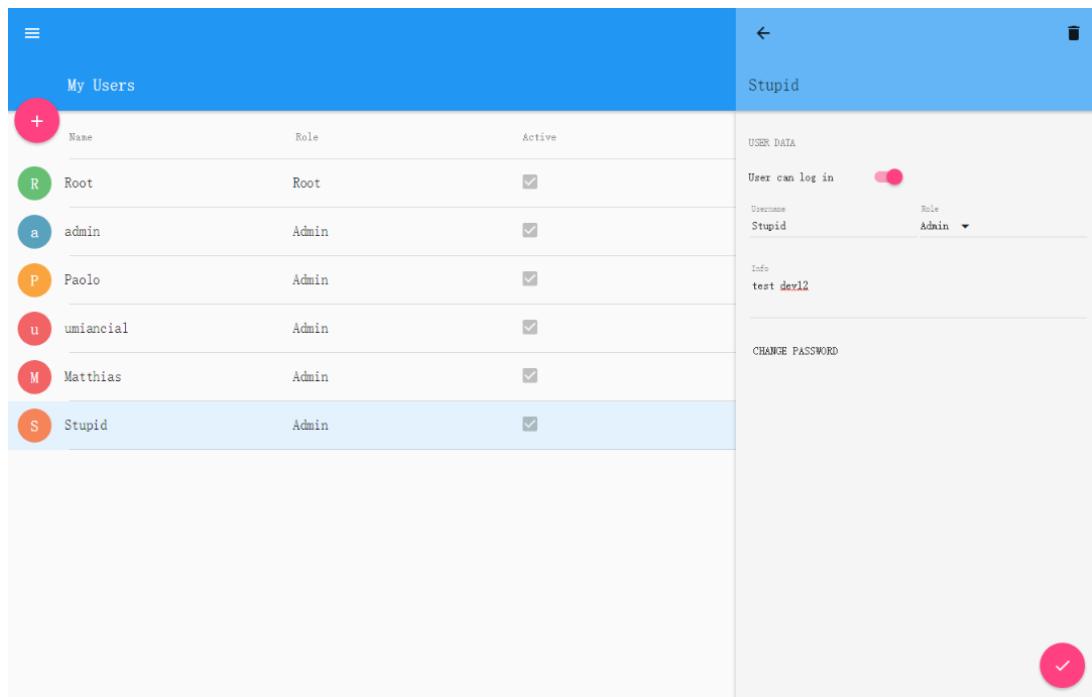
¹一个专门做窗口提示的 JavaScript 库



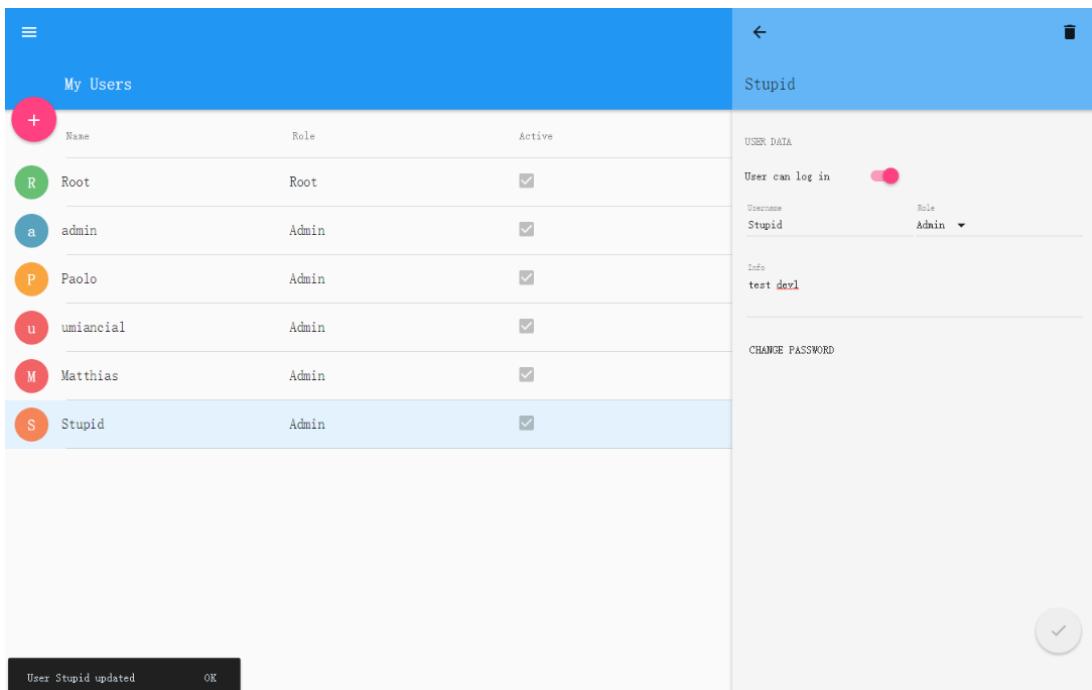
My Users			Stupid	
Name	Role	Active	USER DATA	
Root	Root	<input checked="" type="checkbox"/>	Username: Stupid	<input type="button" value="Edit"/>
admin	Admin	<input checked="" type="checkbox"/>	Role: Admin	<input type="button" value="Edit"/>
Paolo	Admin	<input checked="" type="checkbox"/>	Info: test devl	<input type="button" value="Edit"/>
umiancial	Admin	<input checked="" type="checkbox"/>	RECORD INFORMATION	
Matthias	Admin	<input checked="" type="checkbox"/>	Modified: a few seconds ago by Root	
Stupid	Admin	<input checked="" type="checkbox"/>	Created: 06. June 2010 1:12	

图 4-3 创建用户和查看用户详情

Fig 4-3 Create and detail pages of User



The screenshot shows a user management interface. On the left, a list of users is displayed with columns for Name, Role, and Active status. The user 'Stupid' is selected and highlighted in blue. On the right, a detailed view for 'Stupid' is shown, including fields for User can log in (checked), Username (Stupid), Role (Admin), and Info (test dev12). A 'CHANGE PASSWORD' section is also present. A large pink circular button with a checkmark is visible at the bottom right.



This screenshot is similar to the first one, but it includes a toastr message at the bottom left stating 'User Stupid updated' with an 'OK' button.

图 4-4 修改用户和改动提示
Fig 4-4 Edit page of User and toastr box prompted

4.1.3 客户端模型定义

受 Mongoose 的基于描述的模型 (schema-based model) 启发，本模块独创性地自定义了一种客户端模型定义 (Client Model Definition)，并开发了模型定义服务 (ModelDefinitions) 和与之配套使用的 (当然也可以不使用) 两种可重用组件 (Angular 中的 directive) 模型输入群组 (ModelInputGroup) 与模型视图群组 (ModelViewGroup) 来生成表单、列表和详情页面。

4.1.3.1 模型定义服务

模型定义服务是一个 Angular 服务 (factory)，名叫 ModelDefinitions。它是一个 JavaScript 对象函数¹，作为函数它能够把嵌套的、带有各种简写和别称的模型定义转化为一组扁平化的属性定义 (flatten PropDefinitions)，如下文中代码 4.4 里嵌套的 calibrations.mass 字段会变成一个名叫'calibrations.mass' 的属性定义，一个资源的模型定义最终表现形式就是属性定义数组。为方便使用扁平化的名字对模型进行操作，作为对象的模型定义服务还提供有深度扩展 (deepExtend)、深度读写 (deepGet 和 deepSet)、深度显示 (deepDisplay) 等辅助函数，分别是它的属性 extend、get、set 和 display。

对于模型设计的设计，首先模仿了 Mongoose 模型的类型 (type)、必要 (required)、引用 (ref) 三种选项，其中引用在前端定义中是一种资源，所以叫 resource。type 也有比较大的差异，主要是 Mongoose 中的基础类型对应地变成了前端 input 输入框中允许的类型以及特殊的选择类型，

input 支持的类型 如 text (String)、url (String)、number (Number)、date (Date)、password (String)，括号中为对应的 Mongoose 字段类型；

select 因为 JavaScript 数组是弱类型的，所以 select 可以对应 Mongoose 字段的任意类型，具体类型由选项数组中的值的类型来定；

select/resource 对应 Mongoose 中的 ObjectId 类型，和 ref 一样另有 resource 来指定资源。

客户端模型的每个字段除了有与 Mongoose 类似的类型、必要和资源三个选项外，还有一些用于定义表单输入和页面视图的选项：

desc 字符串，用于显示在输入群组和视图群组中对字段的描述，如果不填，默认为字段名的首字母大写；

displayKey 字符串，用于显示在输入群组和视图群组中字段的值，如果字段值为对象，只显示对象中的这个指定字段；

format 对象或正则表达式，用于输入群组中的正则格式校验，可以包含正则表达式 (value) 和出错信息 (error) 两个选项，也可使用默认出错信息直接缩写为正则表达式；

remoteUnique 字符串，字段唯一，值为指定的资源名称，会自动检查此字段有无其他记录含有相同值；

repeatInput 字符串，强制重复输入，值为需要重复的字段名，一般与 type=password 配合使用，用于重复输入密码；

validators 对象，用于输入群组中的各类校验，上述 required、format、remoteUnique 和 repeatInput 都属于 validators 中配置项的简写或别名 (alias)，对应 required、pattern、remote-unique 和 repeat-

¹JavaScript 中函数也是对象，可以像对象一样拥有属性

input;

displayPriority 字符串，用于响应式布局的视图群组中决定是否显示字段，具体细节在下面的**响应式设计**中介绍；

当 type 为 select 时，表单中这个字段对应的输入组件不再是普通的输入框（input box）而是下拉列表（dropdown list），上述都是一些通用的选项，还有一些只在下拉列表中起作用的选项：

options 数组，显示在下拉列表中的选项，与 displayKey 和 valueKey 配合使用可以做到显示和实际选择的值不同；

valueKey 字符串，实际值字段，如果选项是对象，选项在下拉列表中被选中时字段实际被赋予的值；

getOptions 函数，异步获取选项，只在下拉列表（dropdown list）被点开时才异步地获取选项；

resource 服务对象（service），当资源被指定时，会自动生成 getOptions 选项，并使用服务对象异步调用查询函数来获取选项数组；

params 对象，在使用 resource 自动生成 getOptions 选项时，传入查询函数的参数，用于类条件查询；

比如对应上文传感器（Sensor）的 Mongoose 数据模型，由客户端模型定义的模型如代码 4.4 所示：

代码 4.4 Sensor 的客户端模型

```

1 function SensorDefinition (ModelDefinitions, HardwareVersion, FirmwareVersion, ComponentBatch,
2   ComponentTypes) {
3   return ModelDefinitions({
4     name: {type: 'text', required: true, remoteUnique: 'Sensor', desc: 'ID'},
5     hardwareVersion: {
6       type: 'select/resource',
7       resource: HardwareVersion,
8       desc: 'Hardware Version'
9     },
10    firmwareVersion: {}, // 与上一个类似
11    componentBatches: {
12      body: {
13        type: 'select/resource',
14        resource: ComponentBatch,
15        params: {
16          type: ComponentTypes.BODY
17        },
18        desc: 'Body Batch',
19        displayPriority: 'low'
20      },
21      pcb: {}, // 与上一个类似
22      photodiode: {}, // 与上一个类似
23      laser: {}, // 与上一个类似
24      fan: {} // 与上一个类似
25    },
26    threshold: {type: 'number', required: true},
27    noiseLevel: {
28      type: 'number',
29      required: true,
30    }
31  }
32)
33
```

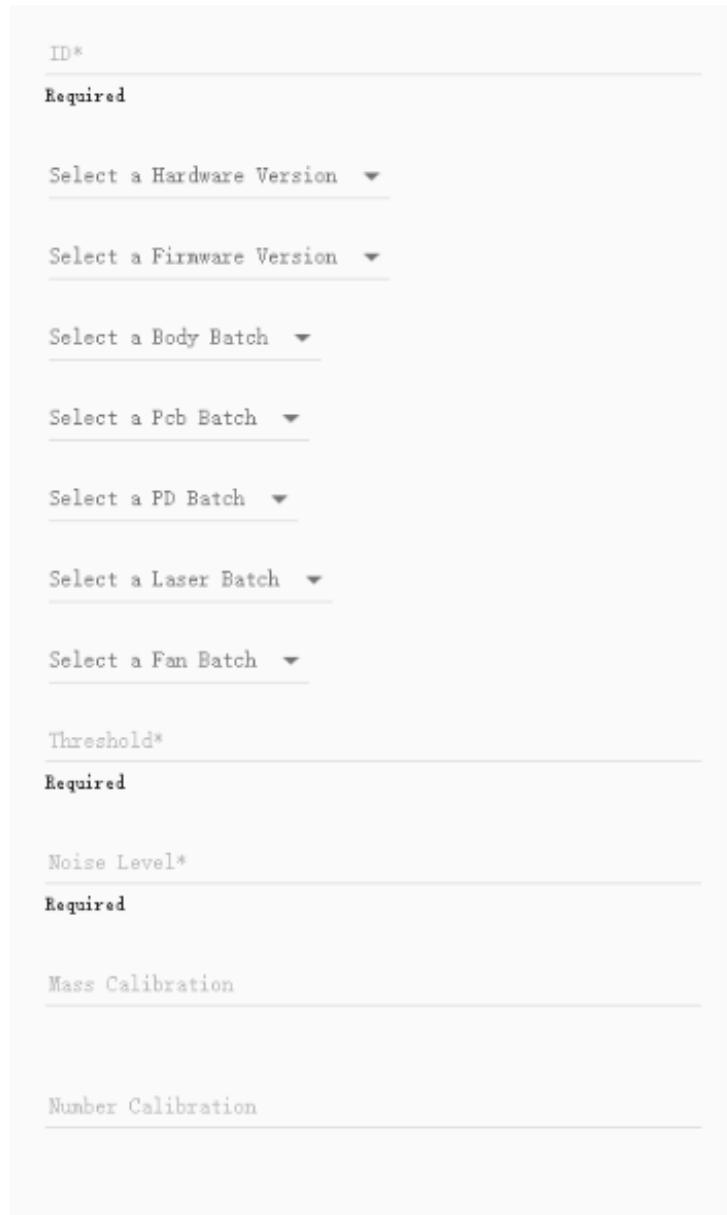
```
29     desc: 'Noise Level'
30   },
31   calibrations: {
32     mass: {type: 'number', desc: 'Mass Calibration'},
33     number: {type: 'number', desc: 'Number Calibration'}
34   }
35 });
36 }
```

4.1.3.2 模型输入群组

模型输入群组 (ModelInputGroup) 是一个本模块开发的 Angular 可重用组件 (directive)，它包含一个模型输入组件 (ModelInput)，它接收一个 Angular 模型 (ng-model，在组件内部命名为嵌套模型，nestedModel)、一组域定义 (fieldDefinitions，因为在表单中所以命名为域定义而不是属性定义) 和一个表单对象 (form)。模型输入群组会把域定义数组中的每个域定义再传递给模型输入组件。因为域定义是扁平化的，直接按照域定义很难直接管理实际需要的嵌套模型，所以模型输入群组内部额外维护了一个的扁平模型，用户在模型输入组件中输入时直接更新的是扁平模型，然后使用模型定义服务的深度赋值函数对嵌套模型进行修改。如代码 4.5 所示，只需要插入这样一段调用组件的 HTML 代码，就可以在传感器资源的创建 (create) 页面渲染出如图 4-5 所示的一组输入组件，在修改 (edit) 页面上也类似。

代码 4.5 传感器创建页面中的模型输入群组代码

```
1 <model-input-group ng-model="create.sensor"
2   form="createForm"
3   field-definitions="create.sensorDefinition">
4 </model-input-group>
```



The screenshot shows a web-based form for creating a sensor. The fields are arranged vertically:

- ID*
Required
- Select a Hardware Version ▾
- Select a Firmware Version ▾
- Select a Body Batch ▾
- Select a Pcb Batch ▾
- Select a PD Batch ▾
- Select a Laser Batch ▾
- Select a Fan Batch ▾
- Threshold*
Required
- Noise Level*
Required
- Mass Calibration
- Number Calibration

图 4-5 传感器创建页面表单
Fig 4-5 Form in create page of Sensor

4.1.3.3 模型视图群组

模型视图群组 (ModelViewGroup) 也是一个本模块开发的 Angular 可重用组件，它接收一个模型对象 (model, 这里不需要修改所以不需要用 ng-model)、一组属性定义和一个类型。还有可选项狭窄模式 (narrow-mode)，用于响应式布局。类型可以分为：

1. items-header (列表头部，只横向列出每个属性定义的描述，此类型不需要传入模型对象)
2. items-content (列表内容，使用模型定义服务的深度显示函数横向列出模型对象每个属性定义

的数值)

3. detail (详细内容, 坚向列出属性定义的描述和对应的数值)

如代码 4.6 所示, 只需要插入这样几段调用组件的 HTML 代码, 并且当前 app 打开了细节页面时 (app.inDetailState 为 true), 就可以在传感器资源的列表 (list) 和细节 (detail) 页面渲染出如图 4-6 所示的一组视图组件。

代码 4.6 传感器列表页面中的模型输入群组代码

```

1 <!-- in items.html -->
2 <model-view-group definitions="index.sensorDefinition"
3           type="items-header"
4           narrow-mode="app.inDetailState">
5 </model-view-group>
6 ...
7 <model-view-group model="sensor"
8           definitions="index.sensorDefinition"
9           type="items-content"
10          narrow-mode="app.inDetailState">
11 </model-view-group>
12 <!-- in detail.html -->
13 <model-view-group model="detail.sensor"
14           definitions="index.sensorDefinition"
15           type="detail">
16 </model-view-group>
```

Sensor List							TestSensor	
	ID	Hardware Version	Firmware Version	Threshold	Noise Level	Mass Calibration	Number Calibration	SENSOR DATA
	TestSensor	TestHardware	TestFirmware	112	122			ID TestSensor Hardware Version TestHardware Firmware Version TestFirmware Body Batch TestBodyBatch Pcb Batch PD Batch Laser Batch Fan Batch Threshold 112 Noise Level 122 Mass Calibration Number Calibration
	SENSOR2	TestHardware	TestFirmware	12.2	22.1			RECORD INFORMATION Modified 6 days ago by Root Created 25. May 2016 11:57

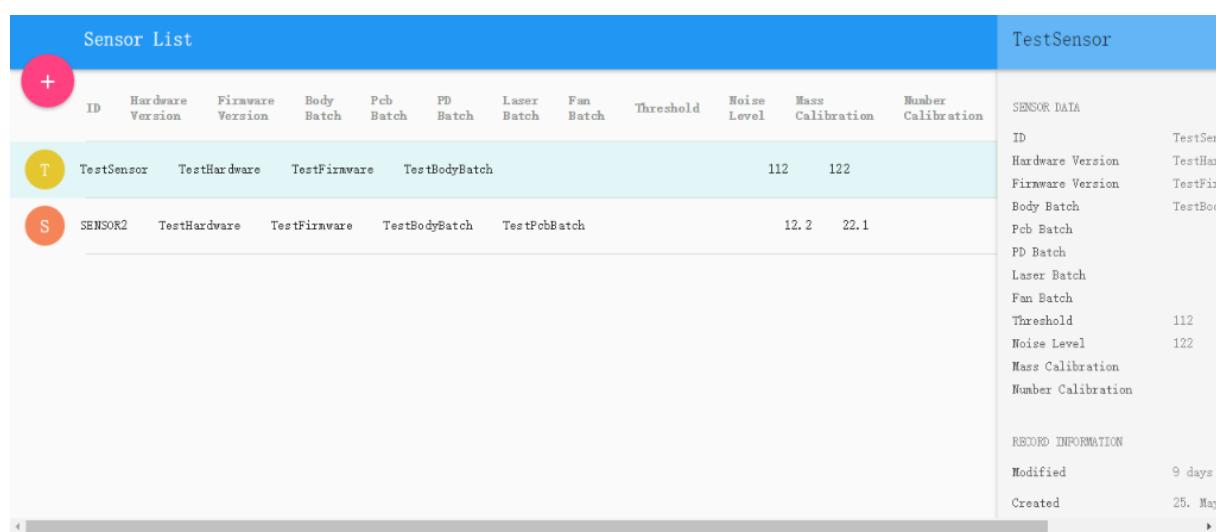
图 4-6 传感器细节页面
Fig 4-6 Detail page of Sensor

4.1.4 响应式设计

从上文所展示的列表页面来看，如果一个模型的字段数量过多，或者字段描述过长，不可避免地会在列表的一行内显示不下，由于是 flex 布局，每个元素根据自己的内容长度都有自己的最小宽度，同时整个页面也有一个最小宽度，因此会出现两种不美观且给用户带来混淆和不便的布局现象，如图 4-7，因为某些字段的值过长，整体上列表头和内容不对齐了，编辑页面打开时，因为左边列表页面由最小宽度，所以被挤到屏幕外面去了，要通过横向滚动条才能看到全部的编辑页面。



Sensor List												
	ID	Hardware Version	Firmware Version	Body Batch	Pcb Batch	PD Batch	Laser Batch	Fan Batch	Threshold	Noise Level	Mass Calibration	Number Calibration
T	TestSensor	TestHardware	TestFirmware	TestBodyBatch					112	122		
S	SENSOR2	TestHardware	TestFirmware	TestBodyBatch	TestPcbBatch				12.2	22.1		



Sensor List												TestSensor	
	ID	Hardware Version	Firmware Version	Body Batch	Pcb Batch	PD Batch	Laser Batch	Fan Batch	Threshold	Noise Level	Mass Calibration	Number Calibration	SENSOR DATA
T	TestSensor	TestHardware	TestFirmware	TestBodyBatch					112	122			ID TestSensor
S	SENSOR2	TestHardware	TestFirmware	TestBodyBatch	TestPcbBatch				12.2	22.1			Hardware Version TestHardware

SENSOR DATA

ID	TestSensor
Hardware Version	TestHardware
Firmware Version	TestFirmware
Body Batch	TestBodyBatch
Pcb Batch	TestPcbBatch
PD Batch	
Laser Batch	
Fan Batch	
Threshold	112
Noise Level	122
Mass Calibration	
Number Calibration	

RECENT INFORMATION

Modified	9 days
Created	25. May

图 4-7 响应式设计之前

Fig 4-7 Before responsive

因此，在客户端模型定义中引入了一个响应式的设计，首先属性定义会有一个可选的显示优先级 (displayPriority) 选项，其次模型视图群组接收一个狭窄模式 (narrow-mode) 的属性。如果某个字段的显示优先级为低级 ('low')，该字段对应的列表头和内容就会被加上一个 CSS 类: hide-in-narrow，如代码 4.7 所示，外面套了一层 Media Query 让元素在屏幕宽度低于 1200 像素的时候自动隐藏；同时因为 CSS 不能自动检测出来右边的编辑或详情页面是否打开，所以主动传入当前 APP 是否在一个详情路由 (inDetailState) 作为狭窄模式，模型视图群组会通过一个叫 showProp 的函数来判断是否显示该字段，如代码 4.8 所示。最终实现的效果如上文中图 4-6 所示，一些不太重要的属性在列表中被隐藏了。

代码 4.7 CSS 类 hide-in-narrow 的代码

```
1 @media screen and (max-width: 1200px) {  
2   .hide-in-narrow {  
3     display: none;  
4   }  
5 }
```

代码 4.8 传感器列表页面中的模型输入群组代码

```
1 function showProp(propDef) {  
2   return !scope.narrowMode || propDef.displayPriority !== 'low';  
3 }
```

4.1.5 代码生成器

本模块不得不提的一点是使用了代码生成器 generator-material-app，除了一开始的基础框架是自动生成的之外，还有一系列的子生成器，用于生成各种常用类型的 Angular 模块，极大地解放了开发人员也就是论文作者的生产力，基本只需要关注核心业务逻辑的编写。上文中提到的所有关于本模块的设计都是先用子生成器生成对应的 Angular 模块，在本模块中试验测试通过后，最后都反哺到了代码生成器中，所以核心业务逻辑基本只剩下前端和后端的模型定义了，这进一步解放了开发人员的生产力，因此本文作者也成为了该代码生成器的主要代码贡献者之一。主要常用的子生成器有：

api name 用于生成一个名为 name 的资源的后端 API 框架；

apiroute name 用于生成一个名为 name 的资源前端 API 路由；

route name 用于生成一个名为 name 的路由，路由更多地是包含在 apiroute 中被生成；

controller name 用于生成一个名为 name 的控制器，控制器更多地是包含在 apiroute 中被生成；

directive name 用于生成 Angular 可重用组件 directive，如模型输入群组和模型视图群组就是先用这个生成再实现具体逻辑的；

factory name 用于生成 Angular 服务 factory，如模型定义服务就是先用这个生成再实现具体逻辑的；

service name 用于生成 Angular 服务 service，Angular 服务的一种，没有必要在此赘述；

provider name 用于生成 Angular 服务 provider，Angular 服务的一种，没有必要在此赘述；

decorator name 用于生成 Angular 修饰器，是用于修改其他 Angular 库的行为的一种 Angular 模块；

filter name 用于生成 Angular 过滤器，相当于一种转化函数，主要用在 Angular 的 HTML 代码中把一个值转换一种显示方式，比如在一些时间格式的显示，也可以直接在 JavaScript 代码中使用；

如图 4-8 所示，这是代码生成器在生成 App 和一个 apiroute 时的命令行交互界面：

4.1.6 Git 提交记录

如图 4-9 所示，这是本文作者 (git 用户名为 stupidisum) 在本模块 balanar 和代码生成器 generator-material-app 这两个项目中的贡献记录。其中 balanar 因为部署需要，包含了一些第三方库的源代码，

因此看起来行数比较多。

```

→ material-app yo
fetch
contextHeaders.js
DOMUtil.js
history.js
passport.js
TokenManager.js
auth
devices
? 'Allo Sun! What would you like to do? Material App ♥ Update Available!
runtime
Make sure you are in the directory you want to scaffold into. from clarityApi
This generator can also be run with: yo material-app A.getCurrentUser();
actions.test.js
? Existing .yo-rc configuration found, would you like to use it? No
configureStore.js
# Configuration
index.js
? Shall I scaffold out a JWT authentication (with users and stuff)? Yes
? Shall I scaffold out a demo resource with full client model definition? Yes
identical however
→ material-app yo material-app:apiroute test
? Where would you like to create this resource? client/app/test
? What will the url of your route be? /test
? What is the main API the route shall communicate with? /api/tests
? What is the url for socket to sync this resource? test= [[{type: A.SET_...
? May only authenticated users be able to access this route? Yes
? What role may access thus route, Milord? (Use arrow keys)
user
  > admin
  withAuth.js
root
  withLoading.js
  59  it('should refreshToken from clarityApi')
  60    asyncAction = A.refreshAccessToken()
  61    returnValue = {access_token: 'a'

```

图 4-8 代码生成器命令行交互界面

Fig 4-8 Cmd ui of code-generator

但从提交数量上来比较，可以看出本模块与代码生成器的开发量比大概在 5:3，基本上只要本模块中需要什么新的特性，在 balanar 中的一个资源上试验通过后，都会反过来加到代码生成器中，然后再用代码生成器重新生成其他资源对应的代码，从而减少了重复的开发工作。如果没有代码生成器，估计 balanar 中的提交数量要翻个三倍以上。

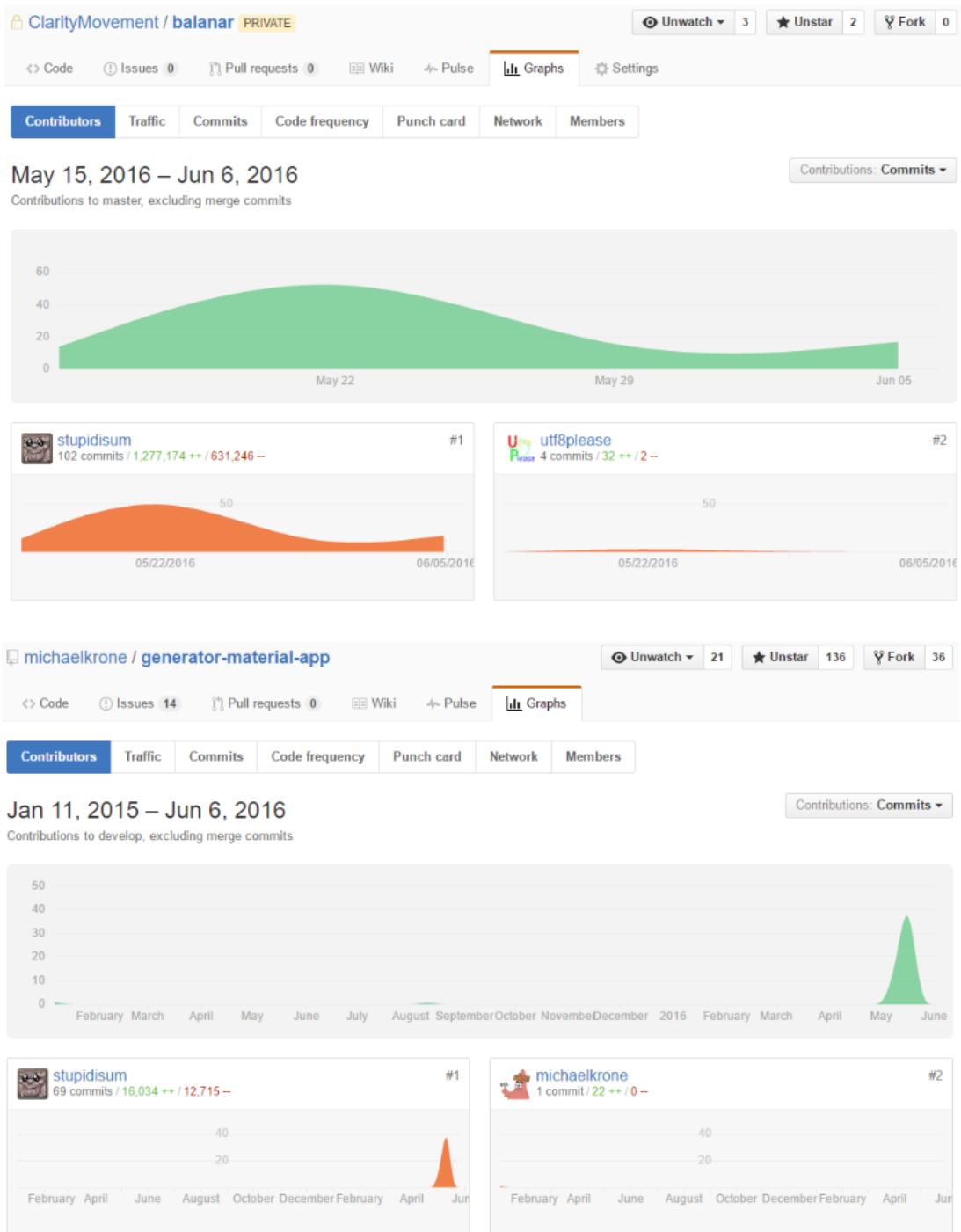


图 4-9 版本管理模块 Git 贡献记录

Fig 4-9 Contributions to balanar

4.2 Smart Home 和 Smart City 模块

Smart Home 和 Smart City 两个模块（下面以本模块代指这两个模块）架构相同，使用了现在越来越流行的灵活技术栈：React+Koa+MongoDB+RedisDB，其中 React 和 Koa 都是核心精简类型的，因此搭配有许多其他的库来完善各自的功能。所以 React 部分架构又可分为专注路由的 react-router 库、专注 flux 数据流的 redux 库和专注下一代浏览器 Fetch 标准¹的 fetch 库等，Koa 部分架构又可分为专注路由的 koa-router 库、专注权限校验的 koa-oauth-server 库、专注 MongoDB 数据资源建模的 Mongoose 库和专注 RedisDB 缓存操作的 redis 库等。本模块是用 ES6²编写的，因此使用了 Eslint 来做语义上的代码风格检查；在构建工具方面，后端使用的依然是成熟且简单的 gulp，但在前端使用了 npm 作为一个尝试。

下面将主要以权限校验为例，按照一次访问的生命周期的时间顺序，从服务端渲染、前端路由器、Redux 数据流、API 请求库和后端路由和数据模型等角度来介绍本模块的详细设计与开发。虽然从直观上看，用户是先访问前端页面再访问后端 API，但其实用户从浏览器进入一个 WEB 应用，发生的第一件事是向服务器请求 HTML 文件，服务端渲染就发生在这个时候。最终介绍了一下主要组件的详细设计与开发。

4.2.1 服务端渲染

服务端渲染，如之前理论介绍中所讲，会在服务端把应用的 HTML、JavaScript 和 CSS 渲染成用户浏览器可以直接渲染的 HTML 页面，而原本选择渲染哪些 HTML、JavaScript 和 CSS 的是前端路由器，显然再实现一个前端路由器很不明智，所以服务端渲染还是利用前端路由器来选择路由的。

目前流行的服务端渲染方式是使用 cookie，在渲染之前就获得了用户的 token³，并用这个 token 获取当前用户传入前端路由器，前端路由器在分配路由之前会验证用户是否已经登录，决定是直接渲染目标路由（当前用户想要访问的页面）还是渲染隐式跳转（implicit redirect）后的登录路由（登录页面）。出于安全性的考虑，本模块不使用 cookie，所以在渲染之前无法获得用户 token，这就意味着这个首次访问的服务端渲染不会验证用户是否已经登录，本模块设计的服务端渲染是假设用户已经登录直接渲染出附加了一层加载蒙版的目标路由给客户端，在客户端加载完成之后，也就是已经获得存在客户端本地的 token 之后，再向服务器验证是否已经登录，如果是，则去掉加载蒙版，如果不是，则显式跳转（explicit redirect）到登陆路由。

如代码 4.9 所示，使用 ReactDOM 的 renderToString 函数把路由器渲染成一个字符串，再通过一个 template 函数把渲染结果插入到 index.html 中并返回给客户端。如果是用流行的那种服务端渲染，应该还会传入用访问令牌拿到的当前用户。

代码 4.9 服务端渲染样例代码

```
1 // ... 省去了错误处理部分，和一些声明无关的代码
2 match({ routes, location: req.url }, (error, redirectLocation, renderProps) => {
3   // ...
4   data.body = ReactDOM.renderToString(
5     <RouterContext {...renderProps} />
```

¹<https://fetch.spec.whatwg.org/>

²ECMAScript 6，目前稳定的新版本的 JavaScript

³令牌，包括访问令牌和刷新令牌，在前文的 Oauth 和 JWT 理论小节有过介绍

```

6   );
7   // ...
8   // 返回 index.html
9   res.send(template(data));
10 });

```

4.2.2 前端路由器

在本模块的前端渲染中，前端路由器通过一组用户自定义的路由声明来组建一个路由树，然后通过监听地址栏的变化，把路由器选择的路由上的对应组件嵌套起来渲染到对应的 DOM 对象上。

声明路由这一功能由 `react-router` 封装地非常完美，如代码 4.10 所示，这是 Smart City 模块的路由声明。一个路由的 `component` 属性里面传入的就是一个具体的组件，其中被 `withAuth` 包裹的 `SmartCity` 组件需要用户登录才能访问。如代码 4.11 所示，前端渲染在监听到地址栏变化后，会调用 `ReactDOM` 的 `render` 函数把路由器渲染到应用容器（`appContainer`）中。如果访问的是登录页面，那么经过路由器的选择，相当于渲染了如代码 4.12 所示的一个组件树。

代码 4.10 Smart City 路由声明

```

1 <Route>
2   <Route path="/" component={App} >
3     <IndexRoute component={withAuth(SmartCity)} />
4     <Route path="sign_in" component={SignIn} />
5   </Route>
6 </Route>

```

代码 4.11 Smart City 前端渲染

```

1 history.listen(location => {
2   // ...
3   ReactDOM.render(
4     <Router history={history}>{routes}</Router>,
5     appContainer
6   );
7 });

```

代码 4.12 Smart City 登录页面实际渲染的组件树

```

1 <App>
2   <SignIn />
3 </App>

```

如代码 4.13 所示，`withAuth` 是一个 React 的高阶组件函数¹，可以把普通的组件变成有登录验证功能的组件，这段代码中跟上文中精简的示例性代码不同有很多冗余，原因是下文中会有所介绍，放在这里可以关联起来。其中 `withLoading` 是另一个高阶组件函数，其功能是简单地把一个 `loading` 画面遮在当前 Component 上方，使其不能被用户操作，在此不予以详述。`withAuth` 函数接受一个组件，

¹HOC 函数，higher-order component

返回一个新定义的组件，这是 HOC 的标准格式。返回之前用 redux 的一个 HOC 函数 connect 包装了一下，后面会详细介绍。其核心逻辑在类里面。

代码 4.13 withAuth 高阶组件函数

```

1  function withAuth(Component) {
2    const ComponentWithLoading = withLoading(Component);
3
4    class ComponentWithAuth extends React.Component {
5      state = {
6        waitingForAuth: true
7      };
8
9      componentWillMount = () => {
10        if (this.props.location.action !== 'POP') {
11          this.state.waitingForAuth = false;
12        }
13      };
14
15      componentDidMount = () => {
16        if (this.state.waitingForAuth) {
17          this.props.onGetCurrentUser();
18        } else {
19          this.checkAuth();
20        }
21      };
22
23      componentWillReceiveProps = this.checkAuth;
24
25      checkAuth() {
26        const {userSignedIn, location} = this.props;
27        if (!userSignedIn && !this.state.waitingForAuth) {
28          this.props.onPush({pathname: '/sign_in', state: {next: location}});
29        }
30      }
31
32      render = () => this.props.userSignedIn ? <Component /> : <ComponentWithLoading />;
33    }
34
35    const mapStateToProps = state => ({userSignedIn: isUserSignedIn(state)});
36    const mapDispatchToProps = dispatchMapper({push, getCurrentUser});
37
38    return connect(mapStateToProps, mapDispatchToProps)(ComponentWithAuth);
39  }

```

本模块完整的前端权限校验流程是这样的：

步骤零 路由控制，如果是需要权限的路由跳到步骤一，如果是登录路由，跳到步骤四，否则跳到步骤五。

步骤一 检查是否为首次渲染，如果是跳到步骤二，如果不是直接跳到步骤三。具体表现为检查当前访问 (location) 是不是直接进来的 (action 为'POP')，如果是应用内页面跳转的话，action 是'PUSH'。

步骤二 获取当前用户信息，完成后之后跳到步骤三。具体表现为调用了 `onGetCurrentUser` 函数，该函数是 redux 的一个 action，完成后会触发下一个周期的 `componentWillReceiveProps`，也就是 `checkAuth` 函数，至于如何触发的将在下一个节 Redux 数据流中介绍。

步骤三 检查当前用户是否拥有足够权限，如果没有跳到步骤四，如果有直接跳到步骤五。具体表现为调用了 `checkAuth` 函数，跳转到登录页面时会把当前地址存储在登录 `location` 的状态中作为登录后跳转的下一个地址 (`state: next: location`)。

步骤四 用户登录并自动跳转，跳到步骤零。具体表现为调用了如代码 4.14 所示的 `SignIn` 组件的 `onSubmit` 函数被触发，跳转到之前指定的下一个地址 (`next`)，或者缺省的主页 ('/')。

步骤五 目标路由正常显示。

代码 4.14 `SignIn` 组件中的 `onSubmit` 函数

```

1 onSubmit = model => {
2   const locationState = this.props.location.state || {next: '/'};
3   this.props.onSignIn(model)
4   .then(() => this.context.router.replace(locationState.next || '/'))
5 }

```

4.2.3 Redux 数据流

Redux 是 Flux 的一个变种，大部分行为和 Flux 相同，如图 4-10，某个地方把 Action (行为) 传入 Dispatcher (触发器)，会引起 Store (仓库) 的变化，Store 的变化又会引起 View (视图) 的变化，视图中可以反过来用 Callback (回调) 函数再把 Action 传给 Dispatcher。

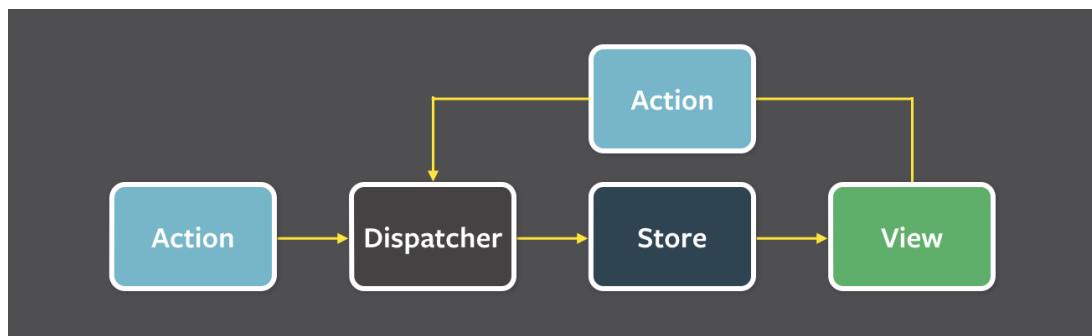


图 4-10 Flux 数据流
Fig 4-10 Flux Data Flow

如图 4-11，Redux 把 Flux 中的 Action 扩展为 Action Creators (行为构造器)、Actions 两个步骤并提供了一个 `dispatch` (触发) 函数来触发一个 Action，把 Flux 中的 Dispatcher 细分为 Middlewares (中间件) 和 Reducer (处理器或直译为压缩器) 两个步骤，Redux 把 Action 定义为一个含有两个属性的对象，分别是 `type` 和 `value`，`type` 的值就是 Action Types (行为类型)，把 Store 中实际存储的状态树叫做 State，Redux 的特点在于 1. 整个应用只有一个 Store 2. 函数式编程，Reducer 应当是纯函数，没有任何副作用 3. 状态不可修改，Reducer 中不可以修改 State 变量，而是每次都必须返回一个新的完整的 State 变量。Redux 推荐使用 State 的第一层属性来表示应用所需要的不同状态，对于第

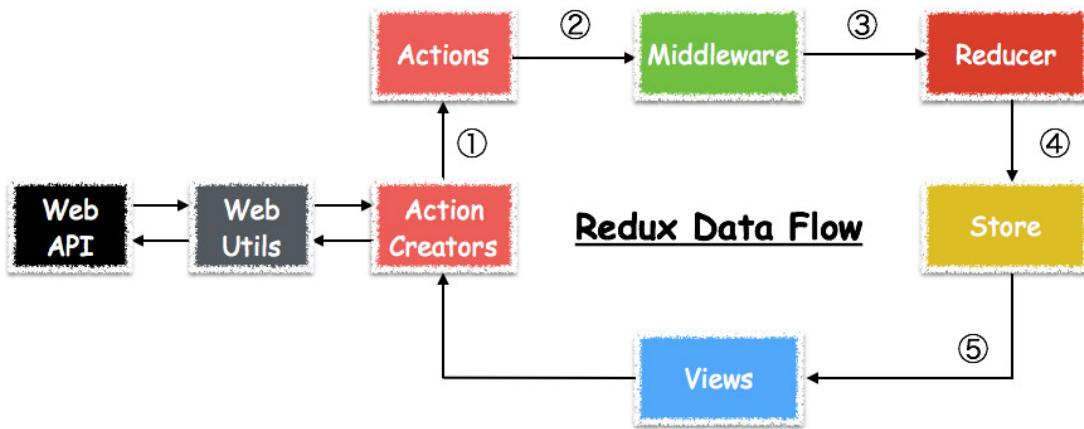


图 4-11 Redux 数据流

Fig 4-11 Redux Data Flow

一层的每个属性都定义一个 SubReducer (子处理器) 函数，最后使用它提供的 combineReducers (整合处理器) 函数组合起来成为图中的 Reducer，SubReducer 应当接收一个缺省为 initialState (初始状态) 的 previousState (旧的状态)，返回一个 nextState (新的状态)。Redux 不推荐直接手动构造一个 Action，所以一般会专门定义一种 Action Creator (行为构造器) 函数。如果构造 Action 之前需要进行一些异步操作再返回这个新的 Action，比如需要调用 Web Utils 函数来调用 Web API 拿到结果后使用结果来构造 Action，一般会专门定义一种特殊的 Action Creator 叫做 Async Action Creator (异步行为构造器)。如果一个行为不仅仅要改变某个状态，还有一些额外副作用的操作的话，Redux 推荐放在 Middleware 中，不推荐放在 Action Creator 和 Reducer 函数中，Action Creator 和 Reducer 函数都应该只负责一件事。

如果是跟 React 一起使用，想要让一个 React Component (React 组件) 得到 State 来改变视图、能够注册 callback 来 dispatch(action)，就得用到 Redux 的一个重要函数 connect (连接)，它的作用就是把 store 中的 state 和某一个 Component 联系起来。connect 的第一个参数是 mapStateToProps (把状态映射到组件属性)，虽然这里是 mapStateToProps 而不是 mapStatePartialsToProps，但 Redux 推荐的是只取需要的部分映射到 Component 的属性上，而不是把整个 State 暴露给 Component；如果当前 Component 需要的属性不是 State 树中直接存在的某个节点，那么一般会专门定义一种 Selector 函数来对 State 进行一些操作后返回 Component 需要的属性。connect 的第二个参数是 mapDispatchToProps (把触发函数映射到组件属性)，虽然这里为了保持灵活度，是 mapDispatchToProps 而不是 mapDispatchCallbackToProps，但 Redux 还是推荐使用构造一些 callback 传给 Component，而不是直接把 dispatch 直接扔给 Component。

如图 4-12 所示，分别是 azwraith 和 robotic 的 redux 目录结构，其中开发 robotic 时，本文作者对其理解还不够深入，所以结构比较粗糙。State 的第一层属性对应不同的文件夹如 auth、devices 等；SubReducer 函数分别定义在文件夹中的 reducer.js 中；一个 State 对应的 Action Types 和 Action Creators 函数（包括异步的）定义在文件夹中的 actions.js 中；Selector 函数定义在 selectors.js 中。根目录下的 actions.js 用于把所有第一层属性的 Actions Creators 组合起来方便引用，reducers.js 的作用是把所有 reducer 函数组合成一个 rootReducer，configStore.js 提供一个配置函数让客户端和服务

端第一次载入时能够构造出一个新的 Store。createReducer.js 中的 createReducer 函数和 index.js 中的 dispatchMapper 函数是本模块自定义的两个函数，用于简化代码量，增加可读性，createReducer¹用于使用简单的 Map²来构造 Reducer，dispatchMapper³用于使用简单的 Map 来构造一组 callback。

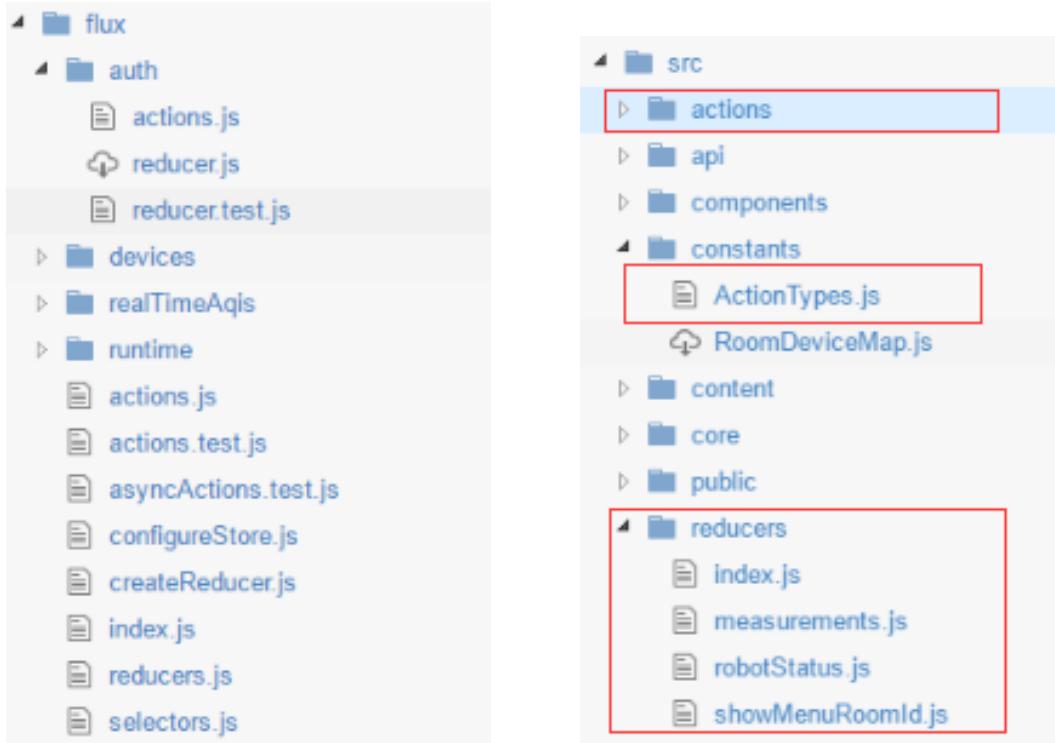


图 4-12 Smart City 和 Smart Home 模块 redux 目录结构

Fig 4-12 Redux directories of Smart Home and Smart City

如代码 4.15 所示，在这里主要以 azwraith 为例介绍首次访问一个受保护的路由（Route with auth）的过程中 Redux 的数据流是怎样的：

- 一旦有状态改变（包括初始状态），connect 函数使用 isUserSignedIn⁴把 userSignedIn 这样的信息映射给了 ComponentWithAuth，使用了 dispatchMapper 函数把 push 和 getCurrentUser 变成了 onPush 和 onGetCurrentUser 这两个 callback 函数映射给 Component；
- ComponentWithAuth 在遇到首次访问受保护的路由的情况下会先调用 onGetCurrentUser 函数，即 getCurrentUser；
- getCurrentUser 会调用 clarityApi.getCurrentUser 向服务器请求当前用户；
- 请求无论成功或失败后都会调用 setCurrentUser 这个 Action Creator 并 dispatch 它返回的 Action；

¹ 把一个初始状态和一个针对每个 Action Type 的处理函数组成的 Map，变成一个正常的 SubReducer；

² Map，在这里是一个 JavaScript 对象，以映射形式表示一组有名字的函数

³ 简单地把一个的 Action Creators 组成的 Map，变成一个接受 dispatch 为参数的并返回一个由 callback 组成的 Map 的函数，这些 callback 的名字以 on 开头加上原本 Action creators 大写首字母的名字

⁴ 一个 Selector 函数，从 state 中筛选出返回用户是否已经登录这样的信息

5. 如果有注册 Middleware， dispatch 会先把 dispatch 函数本身和这个 Action 传给 socketIoMiddleware¹这样的 Middleware；
6. socketIoMiddleware 中又会调用 dispatch (代码中为 next)，重复上一步骤直到没有 Middleware 了；
7. dispatch 会把当前状态和这个 Action 交给 Reducer，Reducer 又会把它们传递给 auth 对应的用 createReducer 构造的 SubReducer；
8. 最终 store 会使用 Reducer 返回的结果更新自己的 State，又回到第 1 步，只不过这次会通过权限验证或者跳转到登陆路由；

代码 4.15 Smart City 模块权限校验 Redux 数据流相关代码

```

1  /* 在withAuth.js 中 */
2  function withAuth(Component) {
3      ...
4      const mapStateToProps = state => ({userSignedIn: isUserSignedIn(state)});
5      const mapDispatchToProps = dispatchMapper({push, getCurrentUser});
6
7      return connect(mapStateToProps, mapDispatchToProps)(ComponentWithAuth);
8  }
9
10 /* 在auth/actions.js中 */
11 // clarityApi 将在下一小节前端 API 库中介绍;
12 import * as clarityApi from '../../../../../clarityApi';
13
14 // Action Types
15 export const SET_CURRENT_USER = 'SET_CURRENT_USER';
16 // Action Creators
17 export const setCurrentUser = value => ({type: SET_CURRENT_USER, value});
18 // Async Action Creators
19 export function getCurrentUser() {
20     return dispatch => clarityApi.getCurrentUser()
21         .then(user => dispatch(setCurrentUser(user)))
22         .catch(() => dispatch(setCurrentUser({})));
23 }
24
25 /* 在另外的../../../data/socket.js中 */
26 export const socketIoMiddleware = () => next => action => {
27     const res = next(action);
28     if (!socket) return res;
29     switch (action.type) {
30         case A.SET_CURRENT_USER: {
31             if (socket.connected) socket.disconnect();
32             socket.connect();
33             break;
34         }
35     }
36     return res;
37 };

```

¹socketIoMiddleware 是一个监听权限验证 Action 来重连 socket 的中间件，socket.io 部分的权限验证在前文的理论中已经介绍过，这里只是实现了一下。

```

38  /* 在auth/reducer.js中 */
39  export default createReducer({}, {
40    [A.SET_CURRENT_USER]: (auth, {value}) => ({...auth, currentUser: value}),
41  });
42

```

4.2.4 API 请求库和 WebSocket

因为 `fetch` 是一个比较底层的库，本模块在 `fetch` 库的基础上封装了一层 `clarityFetch` 的 WEB API 库。为了方便前端在 Component 和 Redux 中调用 API，本模块在 `clarityFetch` 基础上又封装了一个叫 `clarityApi` 的 WEB Utils 库，有像 `getCurrentUser`、`getRoboticStatus`（是在 Smart Home 模块中使用的）这样的简单函数。但本模块使用了 Oauth 做权限校验，所以 `clarityFetc` 需要使用 Redux 中才有的一些信息和函数，如访问令牌等，由于 Redux 部分已经依赖了 `clarityApi`，间接依赖了 `clarityFetch`，所以这里引入了一个 `TokenManager` 来防止循环依赖，Redux 部分在 `configStore` 时把获取这类信息的函数以回调函数的形式传入 `TokenManager`，`clarityFetch` 再调用 `TokenManager` 来获得 Redux 中才有的信息和函数，但其实 `clarityFetch` 并不知情这些信息来自于 Redux。

`clarityFetch` 主要有以下这些特性：

1. 默认选项的配置，比如说默认接收类型（Accept）`application/json` 格式
2. 允许 `url` 为对象从而可以传递参数
3. 使用快捷的 `post`、`delete` 等方法省去指定请求类型
4. 在每个请求的 `header` 中附加当前用户的访问令牌，访问令牌由自定义的 `TokenManager` 的 `getAccessToken` 方法获得
5. 在发请求（request）之前如果请求类型（Content-Type）是 `application/json`，对请求的 `body` 做了 JSON 序列化；
6. 在接收到回应（response）之后，增加了统一的错误处理，如果错误原因是访问令牌不合法，会使用 `TokenManager` 的 `refreshToken` 方法尝试刷新一下访问令牌，然后再用新的访问令牌重发本次请求，期间一旦发生错误就像普通的错误请求一样抛出原本的错误异常；
7. 在接收到成功的回应之后，如果接收类型（Accept）是 `application/json`，对回应的 `body` 做了 JSON 反序列化；

4.2.5 后端路由器和数据模型

后端路由器使用了 `koa-router` 和 `koa-oauth-server` 两个库，如代码 4.16 所示，在控制器的构造函数中加入一行代码就可以让在其后注册的 API 需要权限验证，而之前的不需要。

代码 4.16 Smart City 和 Smart Home 模块后端权限控制相关代码

```

1  /* 在routes.js中 */
2  loadRoutes(api, '/v2/users', require('../user/controller').default);
3
4  /* 在user/controller.js中 */
5  export default (router) => {
6    router.post('/', create);

```

```
7  router.use(convert(oauth.authenticate()));
8  router.get('/current', getCurrent);
9 }
10 async getCurrent(ctx) {
11   ctx.body = ctx.state.oauth.token.user;
12 }
13 async create(ctx) {
14   await User.register({
15     email: ctx.request.body.email,
16     password: ctx.request.body.password,
17   });
18 }
19
20 /* 在oauth.js中 */
21 import oauth from '../models/oauth';
22 export default new OAuthServer({
23   model: oauth,
24 });
25
26 /* 在models/oauth中 */
27
28 // 此 oauth 模型中的属性函数都是是 OAuthServer 要求实现的
29 export default const oauth = {
30   getAccessToken: ...,
31   //中间略去 8 个属性函数
32   revokeAccessToken: ...,
33 }
```

如代码 4.17 所示，这里使用的 User 模型也和版本管理模块一样用的是 Mongoose。

代码 4.17 Smart City 和 Smart Home 模块后端 User 模型

```
1 /* 在models/User.js中 */
2 const User = mongoose.Schema({
3   email: {type: String, unique: true, required: true},
4   password: {type: String, required: true},
5   ...
6 });
7
8 User.static('register', async function(userInfo) {
9   const user = new this(userInfo);
10  await user.validate();
11  await user.hashPassword();
12  return await user.save();
13});
```

4.3 主要组件详细设计和开发

本小节介绍本模块的主要组件，具体分为业务无关的通用组件，Smart City 业务组件和 Smart Home 业务组件，整体上是从小到大的顺序，后面的组件实际使用或包含了前面的组件。

4.3.1 业务无关的通用组件

4.3.1.1 Echart 组件

本模块使用百度开发的 Echarts 图表库来绘制图表，虽然国外有更好的 highcharts 和 D3 等图表库，但它们一个收费，一个过于高级，处理本模块简单的图表绘制需求不够经济适用。为了在 React 中更好地使用 Echarts，本模块封装了一个业务无关的通用 Echart 组件。

Echarts 中绘图主要是需要一个固定高度和宽度的元素作为容器，然后靠一个配置项配置图表中的各类元素，通过传入回调函数来控制来自定义的交互，Echarts 会以事件的形式触发这些回调函数，所以本组件接收 style、option 和 onEvents 三个参数，分别用来让调用者可以控制 Echart 容器的宽高、配置项和回调。同时为了让 Echarts 图表能够自动适应屏幕宽度的变化，在 React 生命周期的 componentDidUpdate 函数中如果检测出容器高度或宽度变了，会重新调用调用 Echarts 的 init 函数。本组件在两个模块中共用，下文的 AqiChart 组件和 HomePanel 组件都使用了 Echart 组件来绘制折线图。实际效果如图 4-13 所示，手动改变高度和宽度后，Echart 组件会自动适应。

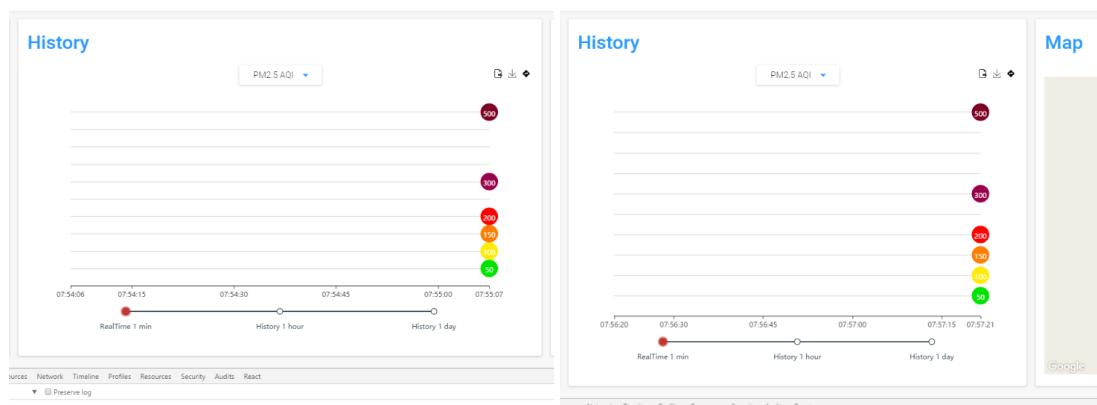


图 4-13 Echart 组件自动适应
Fig 4-13 Auto-sizing of Echart Component

4.3.1.2 HorizontalAccordion 组件

因为 Smart Home 模块独特的设计风格，所以设计了 HorizontalAccordion（水平手风琴）这样一个能够在水平方向像手风琴一样展开任意一片的通用组件。它的具体行为就是把自己包含的所有子组件 (children components) 都当做在手风琴叶片，任意一个叶片被点击，它就会让那个叶片展开，其他叶片收起，同时这个过程为了不给用户造成突兀的感觉增加了动画效果。。具体效果在这里先不演示了。

4.3.1.3 Loading 组件

Loading 组件就是上文在权限检验中提到的浮在普通页面上的 loading 页面，通过一个 withLoading 的高阶组件函数悬浮在一个等待验证授权的组件上方，实现效果如图 4-14。

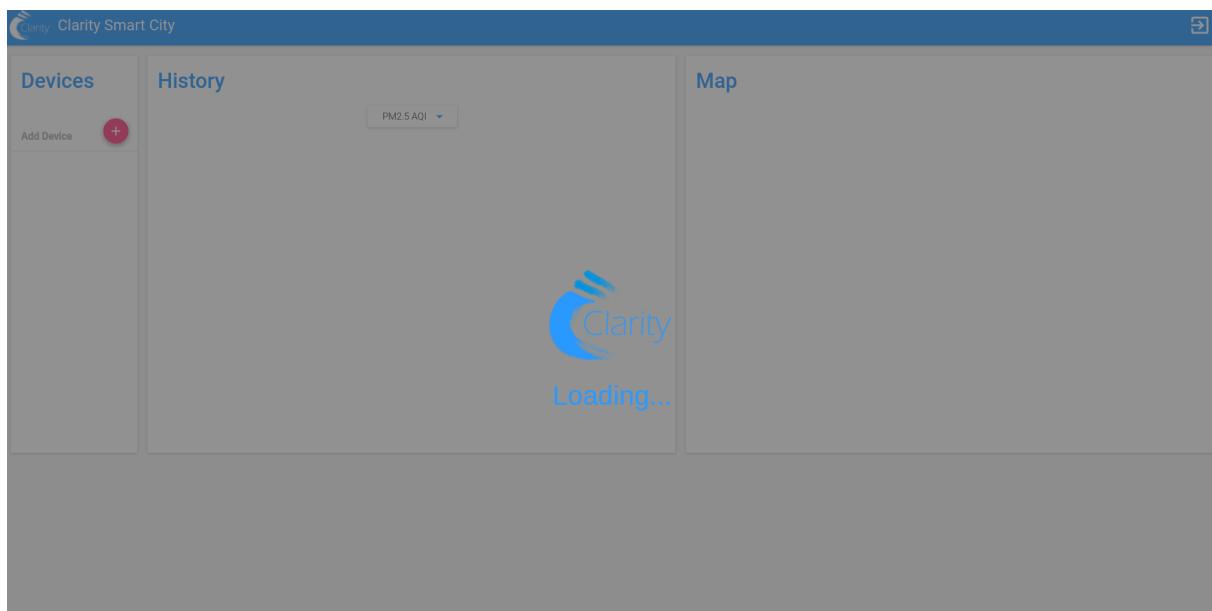


图 4-14 loading 页面
Fig 4-14 Loading Modal

4.3.2 Smart City 业务组件

4.3.2.1 AqiChart 组件

Smart City 模块需要把用户指定的设备的实时、每分钟和每小时空气质量数据用折线图的形式展示出来，同时折线图上的数据应当能够立即下载，因此本模块中定义了一个专门绘制历史空气质量折线图的 AqiChart 组件。因为实时数据是通过 Socket.io 更新的，AqiChart 自己维护不太合适，所以它通过 Redux 数据流获取当前实时数据，它自己内部用计时器每小时和每分钟各获取一次相应时间间隔的历史数据。但 AqiChart 还需要外部决定显示哪些设备以及显示的度量类型，同时还需要外部传入给 Echart 用的样式，所以它接收 devices、measurementType 和 style 三个参数。在渲染时，AqiChart 动态地生成 “RealTime 1 min”（实时一分钟）、History 1 hour（历史一小时每分钟）和 History 1 day（历史一天每小时）的三个子配置项以及用于在 Echart 中选择三者之一的 timeline（时间线）配置项，另外还生成 toolbox（工具栏，包括导出到 csv、保存为图片和高级下载三个功能）、title（标题）、legend（图例）、xAxis（x 轴）、yAxis（y 轴）等一系列配置项，最后组装到一起作为 option 参数，并且把需要监听的事件对应的回调函数构成 onEvents 参数以及外部传入的 style 一起传递给 Echart 组件。实现效果如图 4-15 所示

4.3.2.2 AqiMap 组件

Smart City 模块需要绘制地图来实时展示设备的位置和当前空气质量，所以本模块封装了一个 AqiMap 组件，引用了一个叫 google-map-react 的开源库中的 GoogleMap 组件，使用 Google 地图来绘制地图。另外自定义了一个 SensorMarker 组件来在地图上用气泡标识传感器，鼠标悬浮在气泡上



图 4-15 AqiChart 组件
Fig 4-15 AqiChart Component

会显示传感器详细信息和实时数据，在实时数据的 Redux 数据流中加上了对传感器位置的更新，于是就做到了实时更新地图上设备的位置。按理说，这个 GoogleMap 组件应该像 Echart 组件那样自己适应容器大小，但是它没有实现这个功能，所以本组件又自己实现了自适应容器大小以及自动根据传感器位置找到合适中心点和缩放比例（Google 地图定位需要的两个核心参数）的功能。如图 4-16 所示，原本中心在美国的地图，在增加了一个位置在非洲的设备后，自动变成了中心在大西洋，同时显示了两个设备。

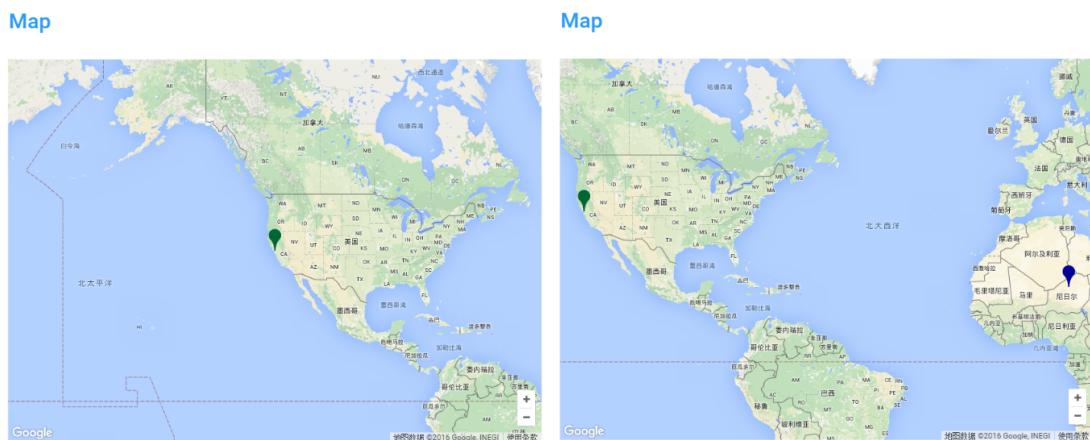


图 4-16 AqiMap 组件自动适应
Fig 4-16 Auto-sizing of AqiMap Component

4.3.2.3 SmartCity 组件

SmartCity 模块最为重要的一个路由组件就是 SmartCity 组件，SmartCity 布局上分三个部分，左边是设备列表，有一个最小宽度，右边的宽度平分给一个 AqiChart 组件和一个 AqiMap 组件。设备列表中可以添加、删除、隐藏和显示设备，另外还用实线和虚线作为折线图的图例，用线的颜色表

示不同的设备。这里之所以叫设备是因为有一个概念上的转移，最终用户并不需要了解这是传感器，只需要知道他拥有的智能设备能够检测空气质量就行。SmartCity 组件在设备数组上附加设备列表中的选中状态，然后传递给 AqiChart 和 AqiMap 来决定隐藏还是显示设备，选中状态不上传到服务器而是保存在本地。实现效果如图 4-17 所示。

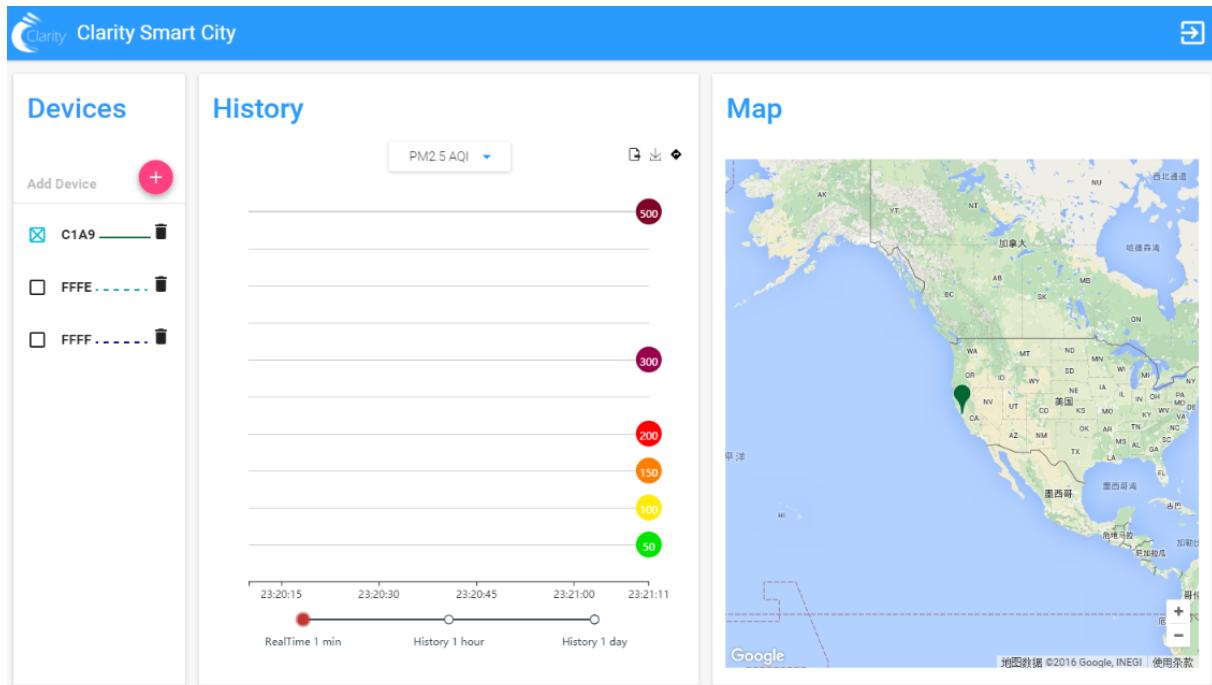


图 4-17 SmartCity 组件
Fig 4-17 SmartCity Component

4.3.2.4 DataCenter 组件

数据是 Clarity 的一大财富，所以 SmartCity 模块专门开发了一个 DataCenter 路由组件用于专门做数据方面的事情，不过目前主要的功能就是高级下载。虽然 AqiChart 组件中的导出到 csv 功能，能够让用户“所见既所下”，但是受制于图表在时间跨度上难以让用户自己选择，所以另外开发了高级下载功能，让用户能够选择任意的时间跨度，当然还是选择设备、时间精度和空气质量度量的，只不过都是以表单的形式，在这里就不用图片展示了。

4.3.3 Smart Home 业务组件

Smart Home 中主要有三个业务组件 PersonPanel、HomePanel 和 CityPanel，它们被放在一个水平手风琴中作为叶片，因为叶片有两种宽度，所以这些组件内部都有展开和收起两种模式。其中 PersonPanel 由于是需求优先级最低的一个，所以开发进度被搁置了，这里不予详细介绍。另外为了实现上文中所提到的智能家居解决方案，本模块还另外设计了 Room 和 Robot 两个组件。

4.3.3.1 Room 组件

Room 组件背景是动画风格的各种房间，点击后会出现一个菜单。如图 4-18 所示，需求要求直观的显示每个房间的空气质量情况，所以 Room 组件会在房间上面加一层窗帘状的灰雾，遮住的范围越大、透明度越低，表示房间空气质量越差。需求要求用户能够手动地控制机器人去某个房间，所以 Room 的菜单上面有个机器人图标的按钮，点击就是下令来净化当前房间。需求要求用户可以查看每个房间的实时空气质量，所以 Room 的菜单上还有一个列表图标的按钮，点击后弹出一个弹窗，其中包含一个用 Echart 组件绘制的实时数据折线图。

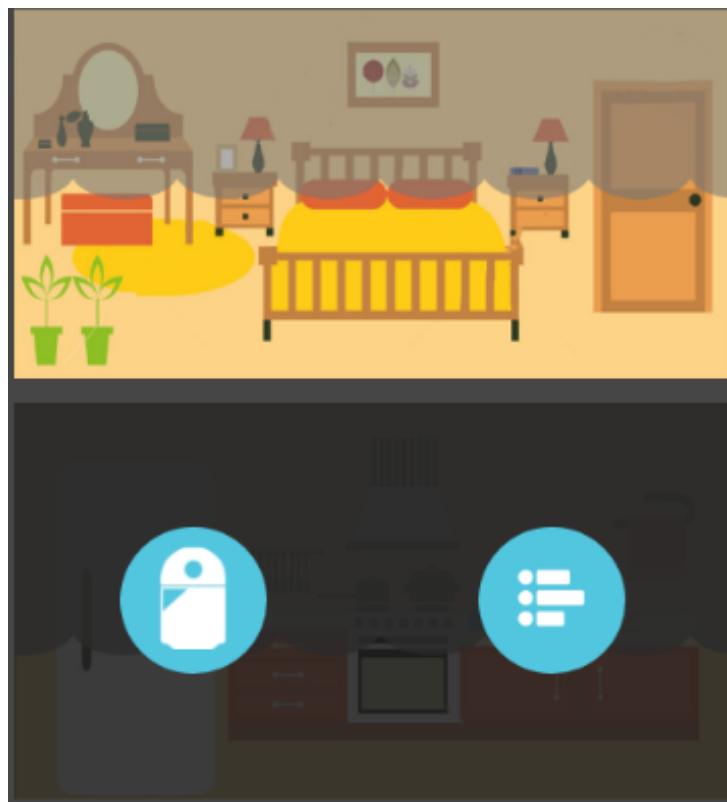


图 4-18 Room 组件
Fig 4-18 Room Component

4.3.3.2 Robot 组件

Robot 组件需要传入它能去的所有房间的位置信息，它自己有一个服务器实时推送的状态（实体机器人的状态），状态中包含机器人上次访问的房间、目的地房间、是否接通电源、空气净化器是否打开、是否处于手动模式、是否处于出错状态等信息，并根据它们来决定 Robot 的工作状态是否要切换到正在工作或者出错状态，分别会让 Robot 头部发绿光和红光。如果上次访问的房间和目的地房间不同，则 Robot 会向目的地房间移动（动画效果），由于不能得知机器人的移动进度，所以这个动画效果会越来越慢且不会到达指定位置，直到收到新的上次访问房间或者目的地房间。如图 4-19 所示，机器人正在净化一个空气状况不太良好的房间。



图 4-19 正在净化房间的 Robot 组件
Fig 4-19 Robot Component purifying a room

4.3.3.3 HomePanel 组件

HomePanel 组件是实现智能家居解决方案的主要业务组件，如图 4-20 所示，HomePanel 背景是一个动画风格的蓝天白云二层小房子，有四个房间，其中三个各有一个 Room 组件，还有一个空房间放了一个选择手动还是自动模式的开关，右下角的房间里有一个 Robot 组件。如图 4-21 所示，在狭窄模式下，HomePanel 会显示四个房间综合的空气质量和机器人的工作状态。



图 4-20 HomePanel 组件
Fig 4-20 HomePanel Component

4.3.3.4 CityPanel 组件

CityPanel 通过用户的浏览器定位用户的位置，然后显示其周围的空气质量热力图和当前城市的空气质量及预告，这里的空气质量热力图也有一个自定义的组件 AirMap，与上面的 AqiMap 一样都使用了 google-map-react 库的 GooleMap 组件，不过地图上实际绘制的内容不同。如图 4-21 所示，这是当初项目初步完成在日本大阪给目标客户演示时的截图。如图 4-20 所示，在狭窄模式下 CityPanel 会显示只显示空气质量预告。

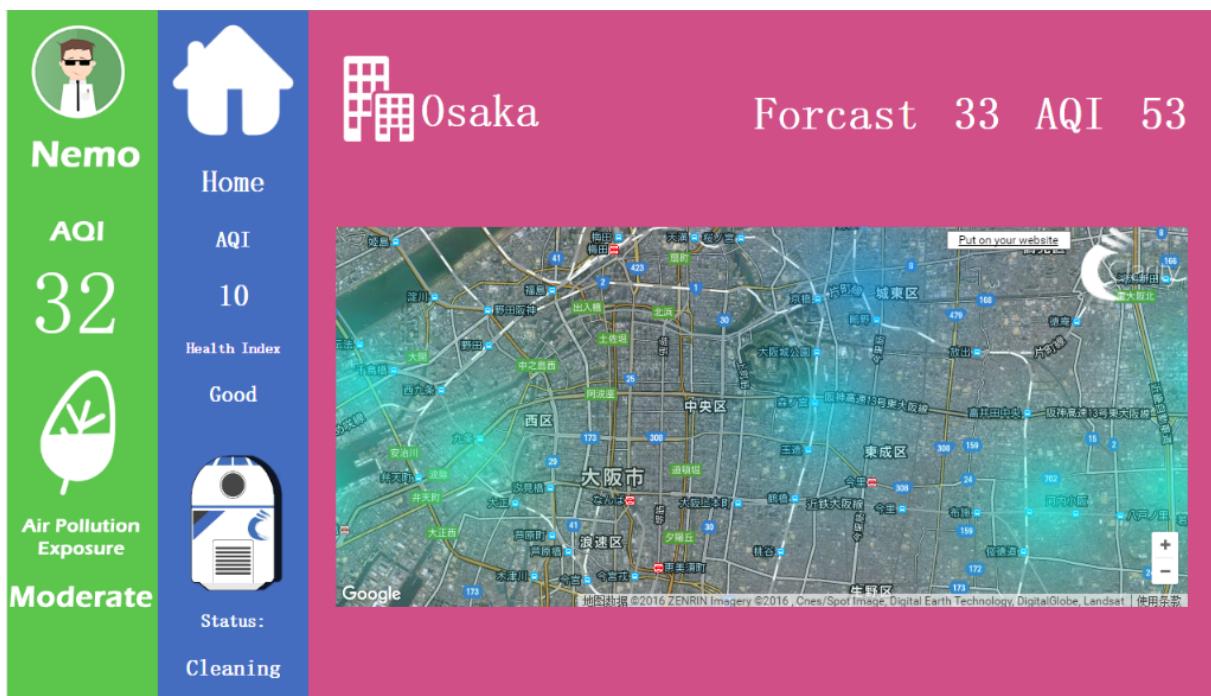


图 4-21 CityPanel 组件
Fig 4-21 CityPanel Component

第五章 系统测试和部署

本系统对重要的业务逻辑做了单元测试，对整个应用做了端对端测试，然后使用 Solano CI 做了持续集成，对于生产分支（也就是主分支，master branch），使用了 AWS 提供的 code pipeline 服务进行了持续集成。

5.1 单元测试

这里以 Smart City 模块的 redux 部分为例，即第四章中图4-12所示的 Redux 文件夹，这一部分是典型的重要业务逻辑。测试主要针对 Action Creators 和 SubReducers 函数，在 actions.test.js 中主要是确保 Action Creators 会生成正确 Action 和 Async Action Creators 会生成接收 dispatch 为参数的通用测试，asyncActions.test.js 中是对 Async Actions Creators 的专项测试，各个文件夹中对于每个 SubReducers 也有对应的测试，确保 SubReducers 不会修改旧状态 (previousState) 而且返回正确的 newState (nextState)。

如代码5.1所示，第一个测试检查了所有 Action Types 变量的值都等于自己的变量名，第二个测试检查了所有 Action Types 都有一个对于的普通 Action Creator 会返回这个类型的 Action。如图5-1所示，这是 Smart City 模块的单元测试结果。

代码 5.1 单元测试样例代码

```
1 describe('redux actionCreators', () => {
2   // ...
3   it('each action type\'s value is its name', () => {
4     expect(normalActions).to.deep.equal(expectedNormalActionsKeys);
5     actionTypes.forEach(actionTypeName => {
6       expect(A[actionTypeName]).to.equal(actionTypeName);
7     });
8   });
9
10  it('each ActionType has a corresponding actionCreator returns this type of action', () => {
11    expect(actionTypes.length === normalActions.length).to.be.true;
12    actionTypes.forEach(actionTypeName => {
13      const actionCreatorName = _.camelCase(A[actionTypeName]);
14      expect(normalActions).to.include(actionCreatorName);
15      expect(A[actionCreatorName]().type).to.equal(actionTypeName);
16    });
17  });
18});
```

图 5-1 Smart City 模块单元测试结果
Fig 5-1 Unit-test results of Smart City

5.2 端对端测试

这里以版本管理模块的端对端测试为例，如代码5.2，如图5-2

代码 5.2 端对端测试样例代码

图 5-2 版本管理模块端对端测试结果

Fig 5-2 E2E-test results of Version Management

5.3 自动部署

AWS 的 CodePipeline 提供的自动部署一般分为三个步骤，分别是从在线版本管理系统获取新版本的代码，编译代码，然后运行编译结果，在本系统中在线版本管理系统是 Github，编译代码部分交给持续集成，最后用 Node 运行持续集成提交上来的结果。这里以版本管理模块的自动部署为例，如图5-3所示这是在 AWS 上配置了三个步骤的自动部署流程。



图 5-3 版本管理模块自动部署流程
Fig 5-3 Code pipeline of Version Management

5.4 持续集成

SolanoCI 简单的持续集成只需要一个 `solano.yml` 配置文件放在项目根目录下面就行了，本系统因为要和 AWS 的自动部署配合，所以需要在持续集成时进行编译，所以额外使用了一些脚本在 `solano.yml` 中调用。这里以 Smart City 模块的持续集成配置为例，介绍本系统的持续集成情况，该模块因为使用了 ES6 需要用 NodeJS5 来编译，而 AWS 上面只提供了 NodeJS4 的运行环境，所以持续集成时涉及到多个 Node 版本的切换，使用了一个叫 `nvm` 的库来切换 Node 版本。如代码5.3所示，持续集成时分 2 个任务 (`tests`)，分别确保进行编译和测试的完成，另有一些钩子 (`hooks`) 分布在持续集成的生命周期中：

pre-setup 在启动持续集成之前下载安装 `nvm` 并且提前安装好要用的 Node4 和 Node5，最后用

Node5 安装了执行任务所需要的依赖

tests 各个任务会在 pre-setup 之后 post-worker 之前执行。

post-worker 在每个任务完成后，执行用于发布 (-release) 的编译 (npm run build)，并在编译的结果中，使用 Node4 来安装 (npm install) 生产环境 (-production) 的依赖，然后把 build 文件夹打包并复制到一个特殊的文件夹，SolanoCI 之后会把这个文件夹中的内容作为持续集成中这一步的结果上传到服务器上，这样本模块的开发人员就可以下载到持续集成时产生的编译结果，出现错误时方便调试。

post-build 在所有任务完成之后，用 build 文件夹代替了当前文件夹，因为 AWS 最后会把这个目录作为编译结果。

代码 5.3 Smart City 模块持续集成配置

```
1 hooks:
2   pre_setup: bash ./tools/solano/pre-setup.sh
3   post_worker: bash ./tools/solano/post-worker.sh
4   post_build: bash ./tools/solano/post-build.sh
5 test_pattern: 'none'
6 nodejs: false
7 iojs: false
8 tests:
9   - source ./tools/solano/load-nvm.sh && nvm use 5 && npm run build
10  - source ./tools/solano/load-nvm.sh && nvm use 5 && npm test
11
12 # pre-setup.sh
13 curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh | bash
14 source ./tools/solano/load-nvm.sh
15 nvm install 4
16 npm install -g npm@3
17 nvm install 5
18
19 npm install
20
21 # post-worker.sh
22 source ./tools/solano/load-nvm.sh
23 nvm use 5
24
25 npm run build -- --release
26
27 nvm use 4
28 (cd build && npm install --production)
29 (cd build && zip -qr ../build.zip .)
30 cp build.zip $HOME/results/TDDIUM_SESSION_ID/session/
31
32 # post-build.sh
33 mv build ../
34 dir=$(pwd -P)cd..rm -rf -- "$dir"
35 mv build "$dir"cd"$dir"
```

一个 SolanoCI 的项目会自动监控对应项目的所有分支，不过有自动部署的 master 分支会由 AWS 的 Code Pipeline 来触发持续集成，所以在 SolanoCI 的配置页面中去掉了 master 分支。如图5-4所示，

eb 开头的持续集成时由 AWS 的 Elastic Beanstalk 触发的，dashboard 对应的就是 azwraith 项目的生产环境，普通分支的持续集成时 SolanoCI 自己触发的。

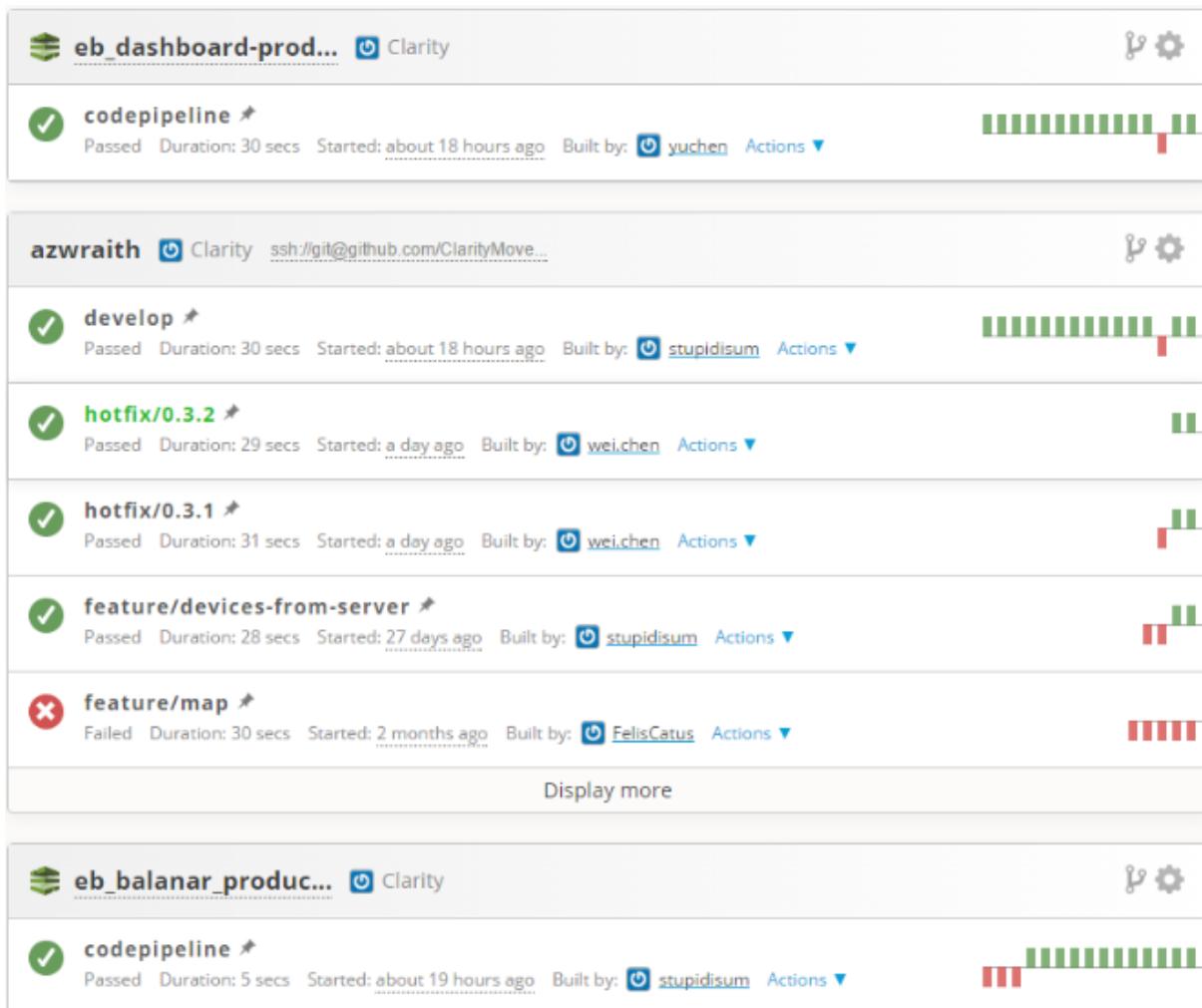


图 5-4 本系统在 SolanoCI 的持续集成页面

Fig 5-4 Dashboard of SolanoCI

第六章 总结与展望

6.1 工作总结

本系统基本完成了所有需求，并且三个模块全部正式上线正在运行。

Smart Home 模块是最先上线的，在 Clarity 和目标客户（那家日本家电企业）洽谈时，让对方实际看到了 Clarity 软件开发能力和云服务平台的强大从而达成了合作意向。在有这个系统之前，别人看不见摸不着，并不知道 Clarity 的云服务平台有什么作用。

智慧城市是 Clarity 当前的业务重点，美国的那个城市政府是 Clarity 非常重要的一一个合作伙伴，在 Clarity 对智慧城市这个需求的摸索过程中，Smart City 模块经过了多次大的需求变动，最终还是第二个上线的，提供了很强大的数据可视化功能和下载功能。

版本管理模块的需求优先级在最初的规划中是最低的，所以它是最后一个开始开发的系统，但正所谓“磨刀不误砍柴工”，虽然代码生成器的开发额外占用了一些精力，但其开发周期反而是最短的。正在接收用户的反馈并且不断地改进之中。

6.2 个人总结

本文作者在一边开发一边学习 ReactJS 和 Redux 的过程中，不断地更新对软件开发和这两项技术的认知，不断地发现别人的更好的或最佳的实践（Best Practice），不仅给了我个人技术的成长，更是让我对这个时代的软件行业和 JavaScript 的世界更加热爱。

版本管理模块给了我一个巨大的惊喜，那就是虽然在给公司做私人项目，但还是有机会为开源社区贡献代码，并且那就是工作的一部分，让我也成为了开源世界的一个贡献者。

6.3 工作展望

从 Clarity 当前的业务重点来看，目前 Smart Home 模块的需求基本被搁置了。

主力需求是 Smart City 模块，这个模块已经迎来第三次大的需求和界面变动，是本团队下一步的工作重点。

版本管理模块还有很多让用户更加方便地增删改查的需求，如批量添加、复制添加、自动生成 ID、上传文件等，也会逐步作为新特性添加到系统中。

在开源工作方面也有一些计划，比如持续改善 generator-material-app 这个代码生成器以及把客户端模型定义单独拿出来做一个开源项目，把这一生产力利器分享给世界。

附录 A 数据字典

A.1 版本管理模块数据字典

代码 A.1 版本管理模块的数据字典

```
1 Explanation:  
2 @: description or restriction of keys or attributes  
3 @fk: short for 'foreign key'  
4 @pk: short for 'primary key'  
5 @uk: short for 'unique key'  
6 @path(type): a file or directory path in cdn of specific type  
7 ...: a combination of object, e.g. {a, b, ...{c, d}, e} == {a, b, c  
     , d, e}  
8 ': {}': attributes of a table or an attribute  
9 ': type()': describe type of an attribute, including [string,  
         integer, decimal, enum]  
10  
11 PS: If the descriptor part is too complex, the examples below maybe  
     helpful.  
12  
13 -----Tables-----  
14 Sensor: {  
15     @pk id  
16     @fk hardwareVersion  
17     @fk firmwareVersion,  
18     @fk componentBatches: {body, pcb, photodiode, laser, fan},  
19  
20     threshold: type(integer),  
21     noiseLevel,  
22     calibrations: {  
23         mass: type(decimal),  
24         number: type(decimal)  
25     },  
26     applicationType: type(enum(SMART_CITY)),  
27 }
```

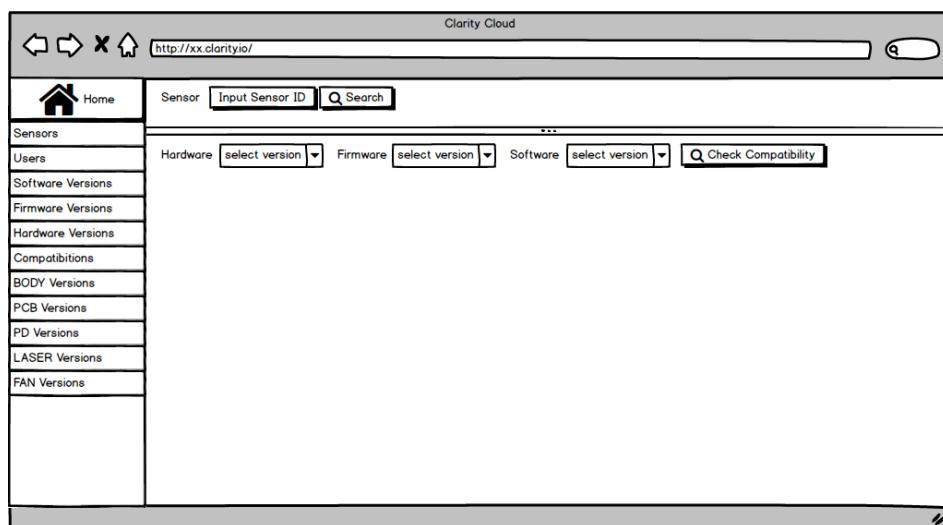
```
28
29 User: {
30     @pk id
31     @uk email
32     password
33     company
34     contacts
35     mobile
36     address
37 }
38
39 HardwareVersion: {
40     no: concat(bodyId, pcbId, photodiodeId, laserId, fanId)
41     @fk componentVersions: {body, pcb, photodiode, laser, fan}
42     @path(zip) testResult
43     @path(excel) calibrationMatrix
44
45     comment
46 }
47
48 FirmwareVersion: {
49     @pk id: type(string), format(/ddd\.\d/, e.g. 001.1)
50     outputQuantities: {
51         massConc: type(string),
52         numberConc: type(string),
53     }
54     @path(hex)
55     bluetoothChip, wifiChip
56     @path(zip) project
57
58     comment
59 }
60
61 ComponentVersion {
62     @pk no,
63     @pk type: type(enum(BODY, PCB, PHOTODIODE, LASER, FAN))
64
65     ...parameters
```

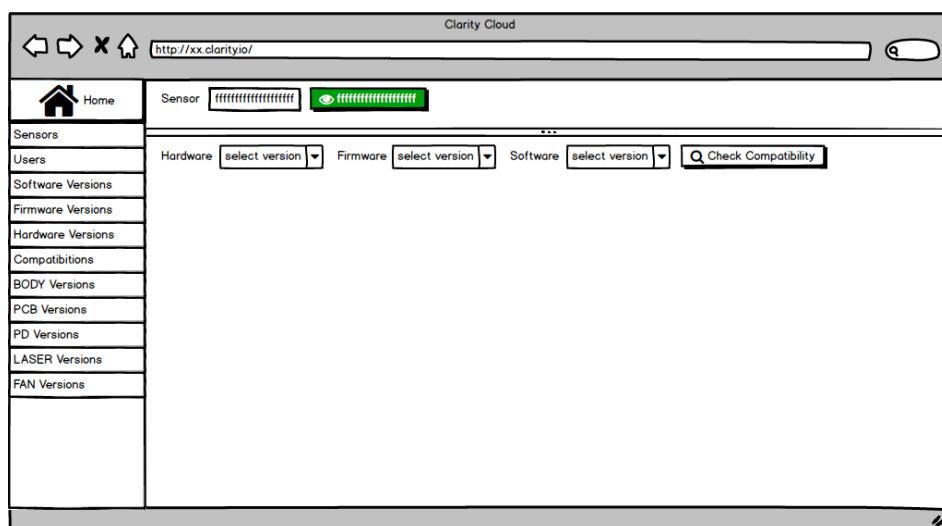
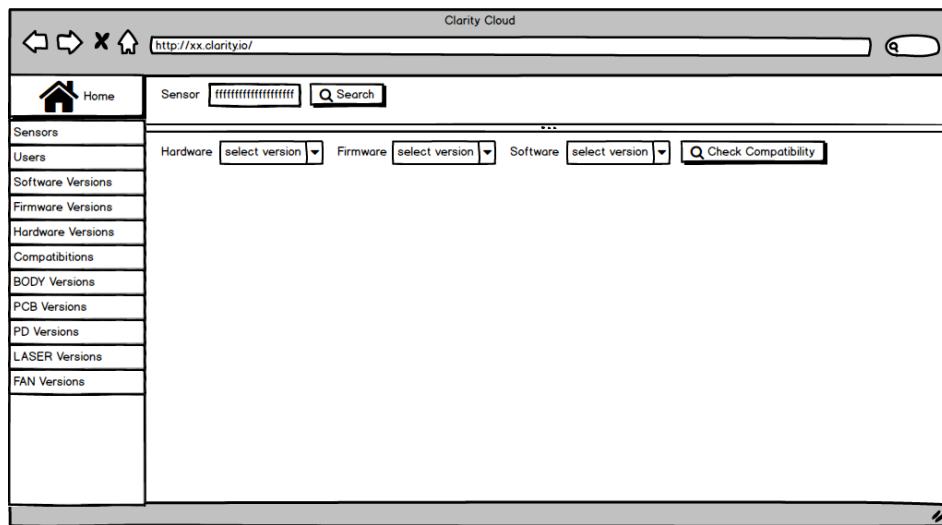
```
66     ...attachments
67     snapshot
68     comment
69 }
70
71 ComponentBatch {
72     @pk no
73     type: type(enum(BODY, PCB, PHOTODIODE, LASER, FAN))
74     @fk
75     componentVersion
76     supplier
77     orderDate
78     deleveryDate
79     quantity
80     totalCost
81     unitPrice
82     comment
83 }
84
85 BodyVersion: ComponentVersion {
86     no: /[A-Z]\d\d/,
87     dimensions: {length, width, height},
88     material: 'Material',
89     productionTechnique: 'Production Technique',
90     cadFile: {name: 'CAD File', type: 'cad'},
91 }
92
93 PcbVersion: ComponentVersion {
94     no: /[A-Z]\d\d/,
95     dimensions: {length, width},
96     pcbFile: {name: 'PCB File', type: 'pcb'},
97     schematicsAndAssembly: {name: 'Schematics and Assembly', type: 'pdf'},
98     bom: {name: 'BOM File', type: 'excel'}
99 }
100
101 PhotodiodeVersion: ComponentVersion {
102     no: /[A-Z]/,
```

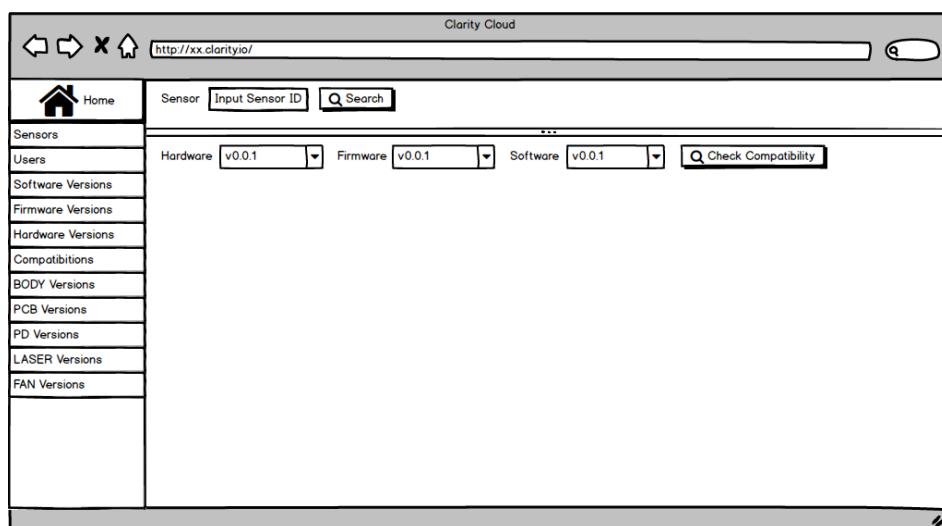
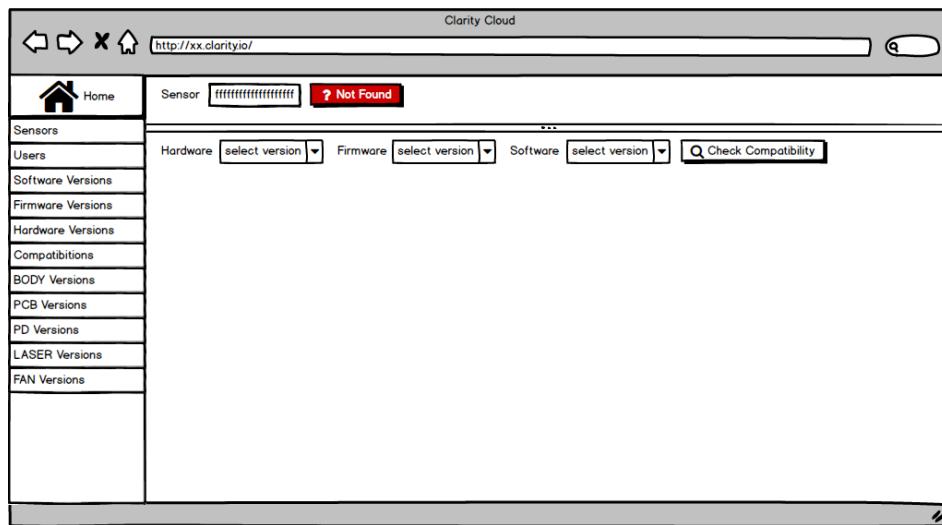
```
103     dimensions: {length, width, height}
104     peakDetectionWavelength: 'Peak Detection Wavelength',
105     manufacturerSite: 'Manufacturer Site',
106     datasheet: {name: 'Datasheet', type: 'pdf'}
107 }
108
109 LaserVersion: ComponentVersion {
110     no: /[A-Z]/,
111     dimensions: {diameter, length},
112     consumption: {current, voltage},
113     emissionWavelength: 'Emission Wavelength',
114     power: 'Power',
115     lensMaterial: 'Lens Material',
116     focalLength: 'Focal Length',
117     lifetime: 'Lifetime',
118     manufacturerSite: 'Manufacturer Site',
119     datasheet: {name: 'Datasheet', type: 'pdf'},
120 }
121
122 FanVersion: ComponentVersion {
123     noFormat: /[A-Z]/,
124     dimensions: {length, width},
125     airflowAt0PressureDrop: 'Airflow at zero Pressure Drop',
126     lifetime: 'Lifetime',
127     manufacturerSite: 'Manufacturer Site',
128     datasheet: {name: 'Datasheet', type: 'pdf'},
129 }
```

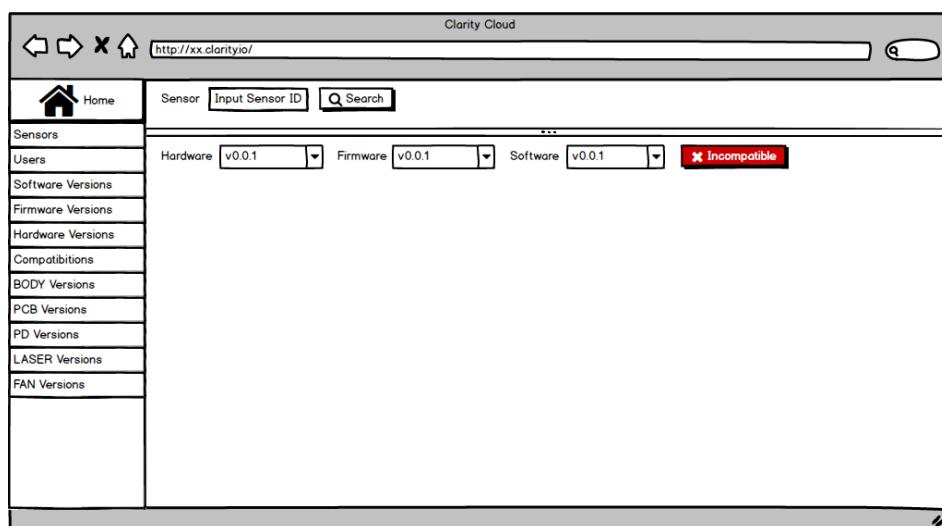
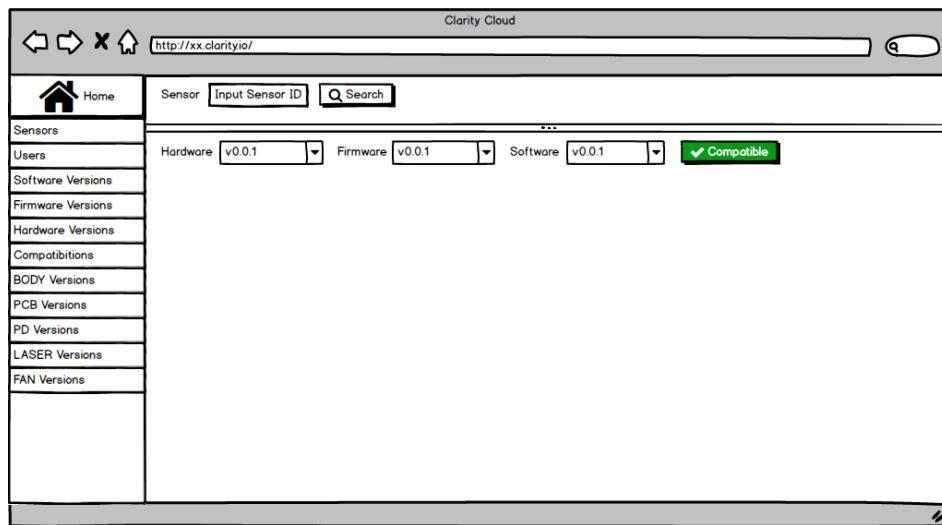
附录 B 快速原型

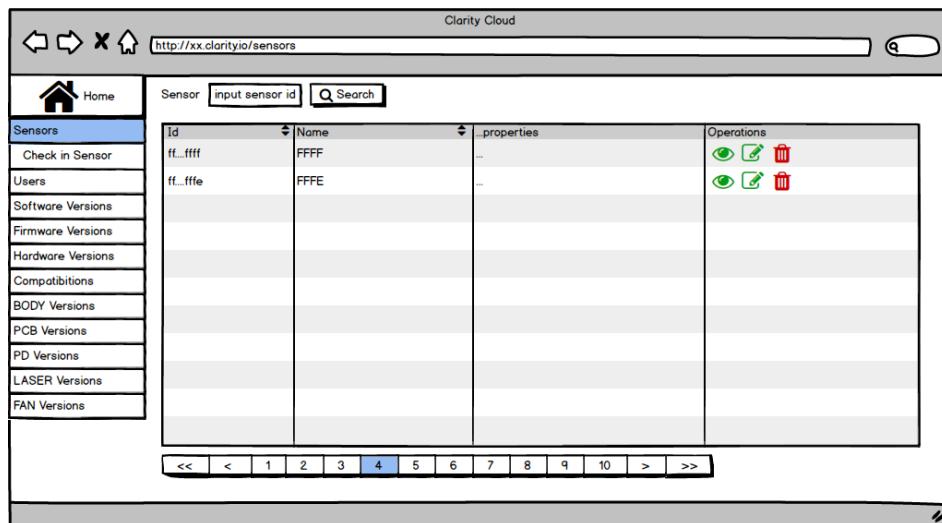
B.1 版本管理模块快速原型



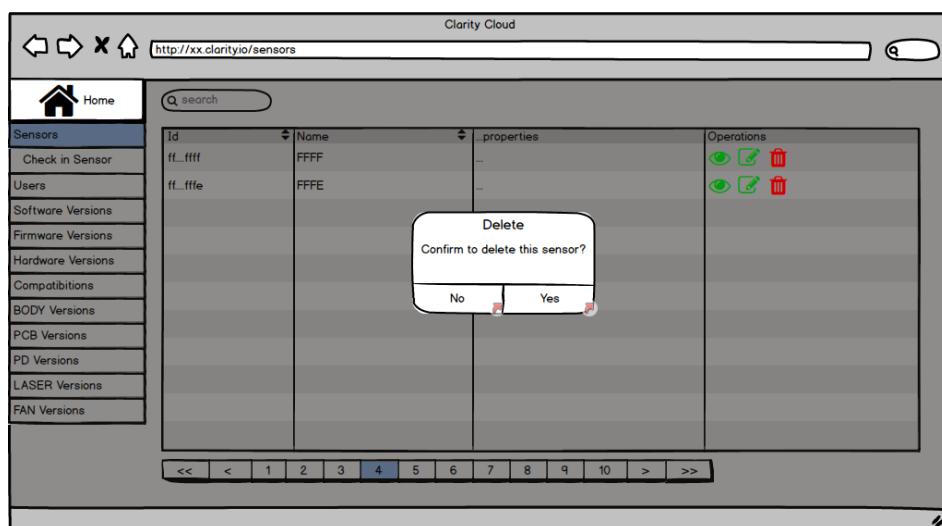








The screenshot shows a web-based management interface for sensors. The top navigation bar includes a back button, forward button, close button, and a search bar with the URL <http://xx.clarityio/sensors>. On the left, a sidebar menu lists various categories: Home, Sensors, Check in Sensor, Users, Software Versions, Firmware Versions, Hardware Versions, Competitions, BODY Versions, PCB Versions, PD Versions, LASER Versions, and FAN Versions. The main content area is titled "Clarity Cloud" and displays a table of sensor data. The table has columns for Id, Name, properties, and Operations. Two rows are visible: one with Id f...ffff and Name FFFF, and another with Id f...fffe and Name FFFE. Each row has three operations icons: a green eye, a green edit/pencil icon, and a red trash bin. Below the table is a navigation bar with page numbers from << to >>. The overall interface is clean and modern.



This screenshot shows the same interface as above, but with a modal dialog box centered over the sensor list. The dialog is titled "Delete" and contains the message "Confirm to delete this sensor?". It has two buttons at the bottom: "No" and "Yes". The background of the main content area is dimmed to indicate that interaction with the underlying table is disabled while the dialog is open. The rest of the interface, including the sidebar and navigation, remains visible.

Clarity Cloud

<http://xx.clarityio/sensors/create>

Check in Sensor

Name	<input type="text"/>	+
Hardware Version	<input type="text" value="please select"/>	+
Firmware Version	<input type="text" value="please select"/>	+
Body Batch	<input type="text" value="please select"/>	+
PCB Batch	<input type="text" value="please select"/>	+
PD Batch	<input type="text" value="please select"/>	+
LASER Batch	<input type="text" value="please select"/>	+
FAN Batch	<input type="text" value="please select"/>	+

...other properties

Home

Sensors

Check in Sensor

Users

Software Versions

Firmware Versions

Hardware Versions

Competitions

BODY Versions

PCB Versions

PD Versions

LASER Versions

FAN Versions

Clarity Cloud

<http://xx.clarityio/sensors/id>

View Sensor xx

editable

ID	<input type="text" value="fffffffffffffe"/>
Name	<input type="text" value="test sensor"/>
Hardware Version	<input type="text" value="v11"/>
Firmware Version	<input type="text" value="v11"/>
Body Batch	<input type="text" value="b16-03-21-01"/>
PCB Batch	<input type="text" value="b16-03-21-01"/>
PD Batch	<input type="text" value="b16-03-21-01"/>
LASER Batch	<input type="text" value="b16-03-21-01"/>
FAN Batch	<input type="text" value="b16-03-21-01"/>

...other properties

Home

Sensors

Check in Sensor

Users

Software Versions

Firmware Versions

Hardware Versions

Competitions

BODY Versions

PCB Versions

PD Versions

LASER Versions

FAN Versions

Clarity Cloud

http://xx.clarityio/sensors/id/edit

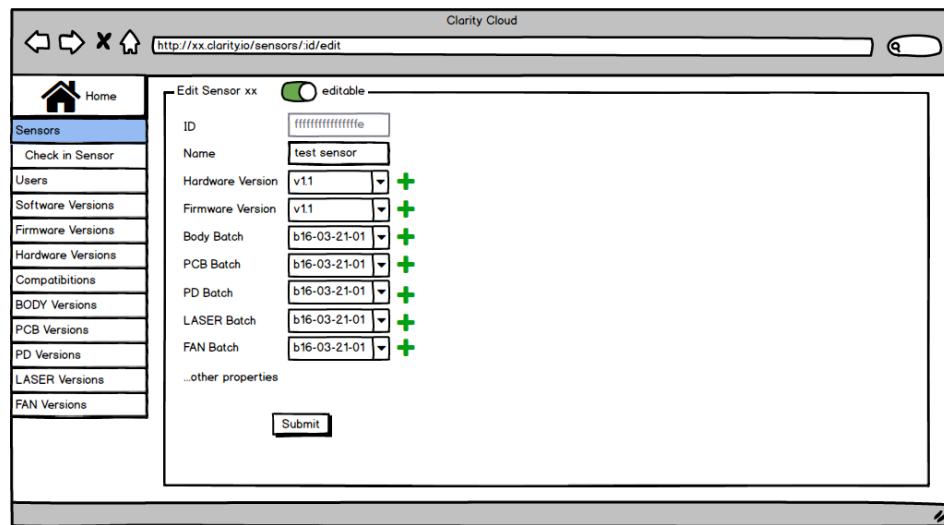
Edit Sensor xx

editable

ID	fffffffffffffe
Name	test sensor
Hardware Version	v11
Firmware Version	v11
Body Batch	b16-03-21-01
PCB Batch	b16-03-21-01
PD Batch	b16-03-21-01
LASER Batch	b16-03-21-01
FAN Batch	b16-03-21-01

...other properties

Submit



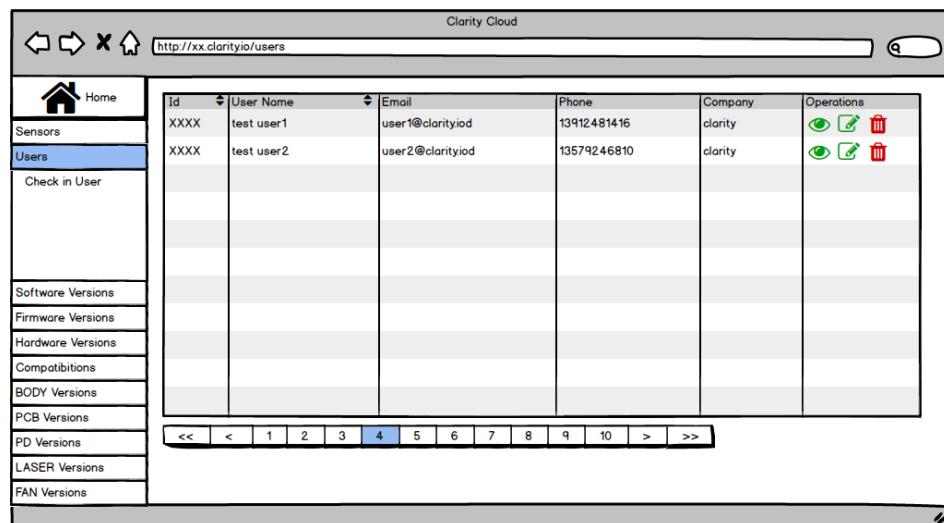
Clarity Cloud

http://xx.clarityio/users

Users

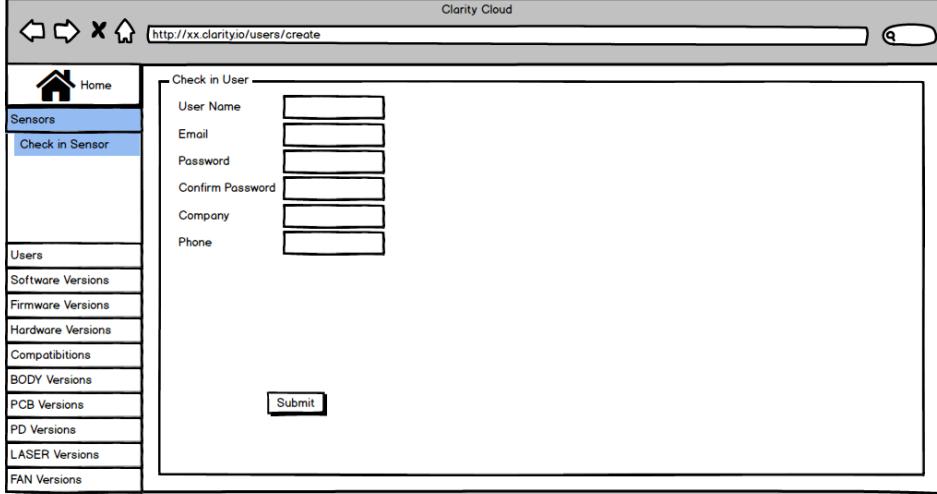
ID	User Name	Email	Phone	Company	Operations
XXXX	test user1	user1@clarityiod	13912481416	clarity	 
XXXX	test user2	user2@clarityiod	13579246810	clarity	 

<< < 1 2 3 4 5 6 7 8 9 10 > >>



Clarity Cloud

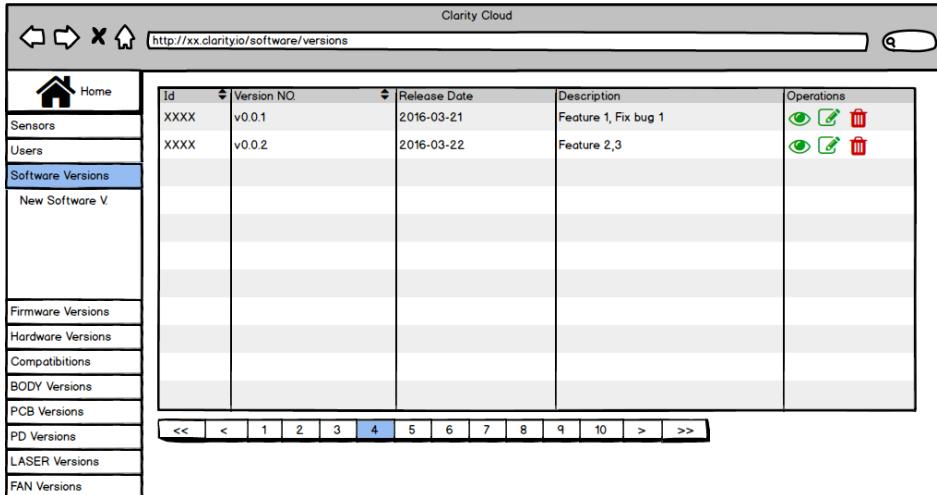
<http://xx.clarityio/users/create>



The form is titled "Check in User". It contains fields for User Name, Email, Password, Confirm Password, Company, and Phone. A "Submit" button is located at the bottom right.

Clarity Cloud

<http://xx.clarityio/software/versions>

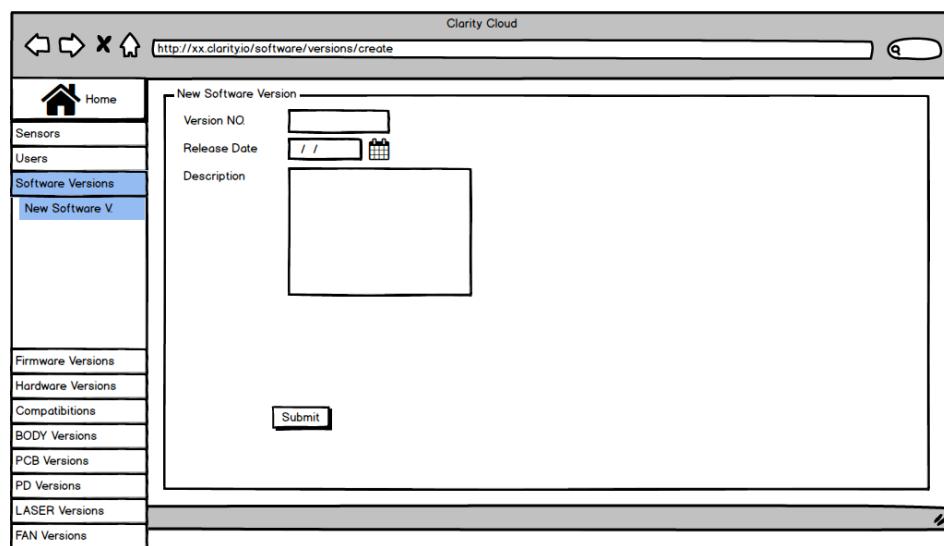


A table showing software versions. The columns are Id, Version NO, Release Date, Description, and Operations. Two rows are present:

Id	Version NO	Release Date	Description	Operations
XXXX	v0.0.1	2016-03-21	Feature 1, Fix bug 1	
XXXX	v0.0.2	2016-03-22	Feature 2,3	

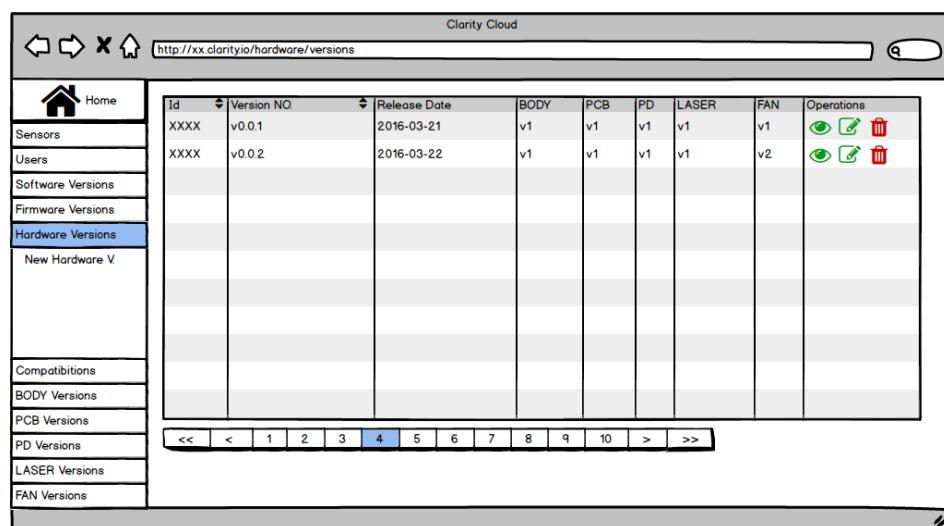
Clarity Cloud

<http://xx.clarityio/software/versions/create>



Clarity Cloud

<http://xx.clarityio/hardware/versions>



Id	Version NO.	Release Date	BODY	PCB	PD	LASER	FAN	Operations
XXXX	v0.0.1	2016-03-21	v1	v1	v1	v1	v1	
XXXX	v0.0.2	2016-03-22	v1	v1	v1	v1	v2	

Clarity Cloud

<http://xx.clarityio/hardware/versions/create>

	Home
	Sensors
	Users
	Software Versions
	Firmware Versions
	Hardware Versions
	New Hardware V
	Compatibilities
	BODY Versions
	PCB Versions
	PD Versions
	LASER Versions
	FAN Versions

New Hardware Version

Version NO.

Release Date /

Body Version please select

PCB Version please select

PD Version please select

LASER Version please select

FAN Version please select

Clarity Cloud

<http://xx.clarityio/compatibilities>

	Home
	Sensors
	Users
	Software Versions
	Firmware Versions
	Hardware Versions
	Compatibilities
	BODY Versions
	PCB Versions
	PD Versions
	LASER Versions
	FAN Versions

Hardware Firmware Software

Compatibility ID | Hardware VNO | Firmware VNO | Operations

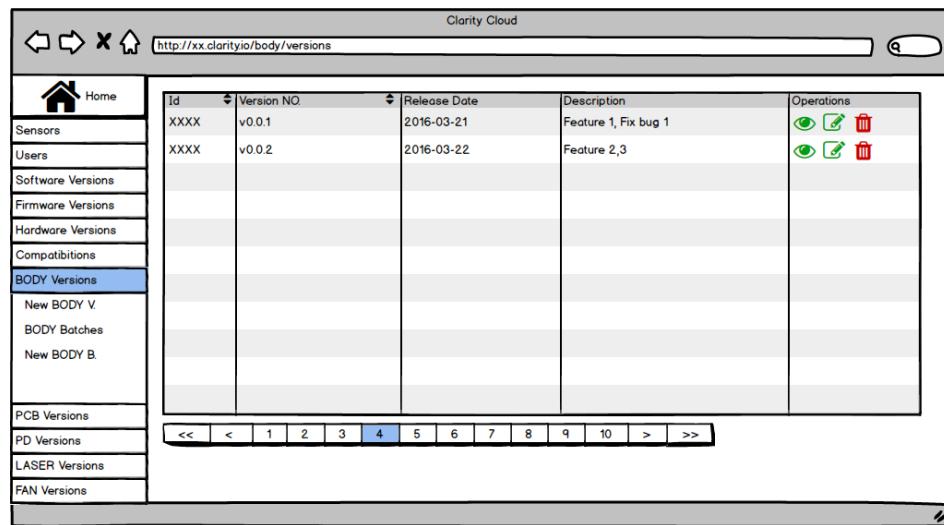
1	v0.0.1	v0.0.1	<input type="button" value="Delete"/>
2	v0.0.1	v0.0.2	<input type="button" value="Delete"/>
3	v0.0.2	v0.0.2	<input type="button" value="Delete"/>
New	<input type="text"/> please select <input type="button" value="▼"/>	<input type="text"/> please select <input type="button" value="▼"/>	<input type="button" value="Save"/> <input type="button" value="New version +"/>
		v0.0.1 v0.0.2 new version	<input type="button" value="Save"/> <input type="button" value="New version +"/>

Software & Firmware Compatibilities

Compatibility ID Software VNO Firmware VNO Operations			
1	v0.0.1	v0.0.1	<input type="button" value="Delete"/>
2	v0.0.1	v0.0.2	<input type="button" value="Delete"/>
3	v0.0.2	v0.0.2	<input type="button" value="Delete"/>
New	<input type="text"/> please select <input type="button" value="▼"/>	<input type="text"/> please select <input type="button" value="▼"/>	<input type="button" value="Save"/> <input type="button" value="New version +"/>
	v0.0.1 v0.0.2 new version	<input type="button" value="Save"/> <input type="button" value="New version +"/>	

Clarity Cloud

<http://xx.clarityio/body/versions>

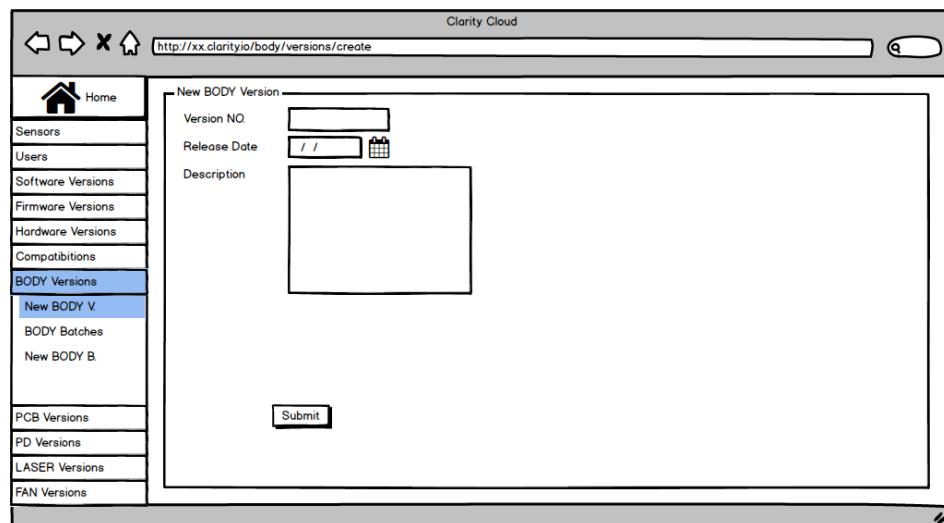


Id	Version NO	Release Date	Description	Operations
XXXX	v0.0.1	2016-03-21	Feature 1, Fix bug 1	
XXXX	v0.0.2	2016-03-22	Feature 2,3	

<< < 1 2 3 4 5 6 7 8 9 10 > >>

Clarity Cloud

<http://xx.clarityio/body/versions/create>



New BODY Version

Version NO.

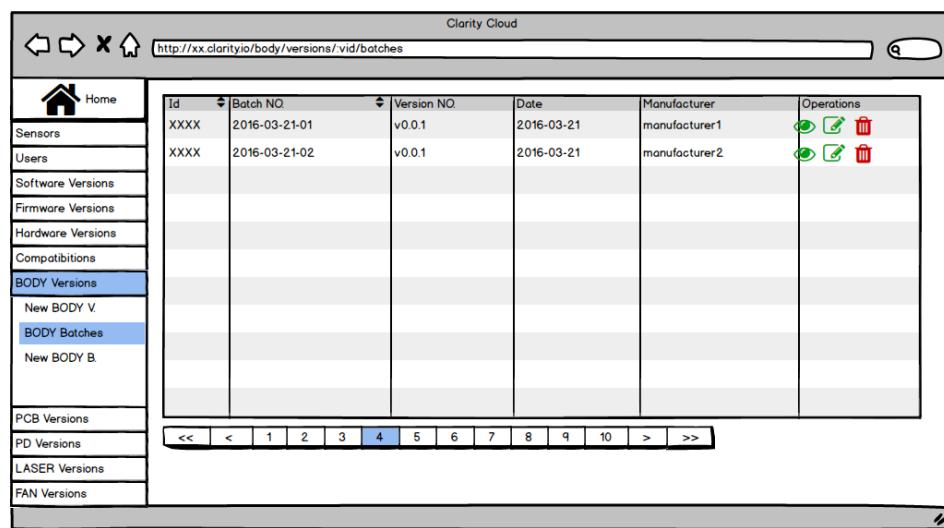
Release Date / /

Description

Submit

Clarity Cloud

<http://xx.clarityio/body/versions/vid/batches>

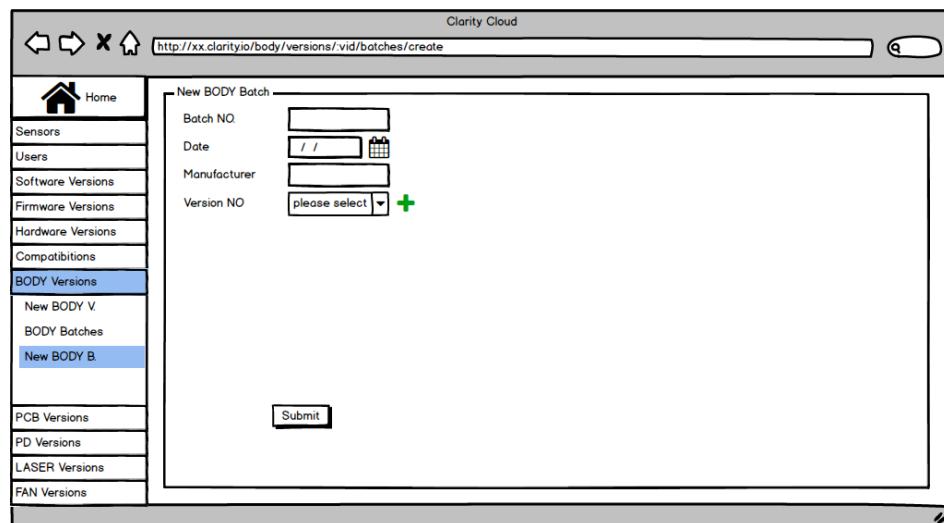


Id	Batch NO.	Version NO.	Date	Manufacturer	Operations
XXXX	2016-03-21-01	v0.0.1	2016-03-21	manufacturer1	
XXXX	2016-03-21-02	v0.0.1	2016-03-21	manufacturer2	

<< < 1 2 3 4 5 6 7 8 9 10 > >>

Clarity Cloud

<http://xx.clarityio/body/versions/vid/batches/create>



New BODY Batch

Batch NO.

Date / /

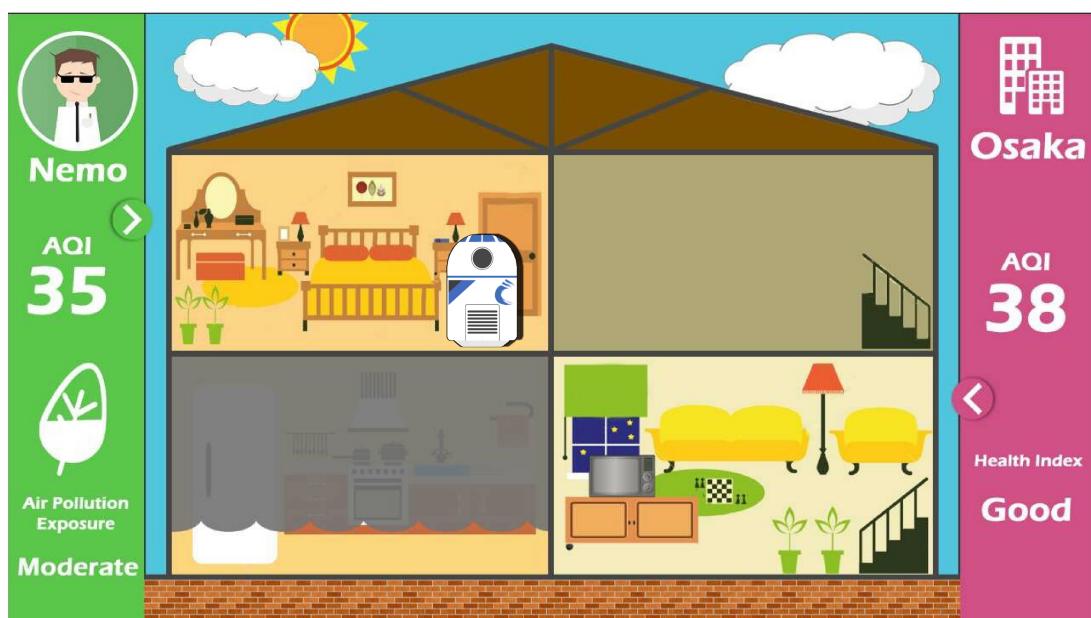
Manufacturer

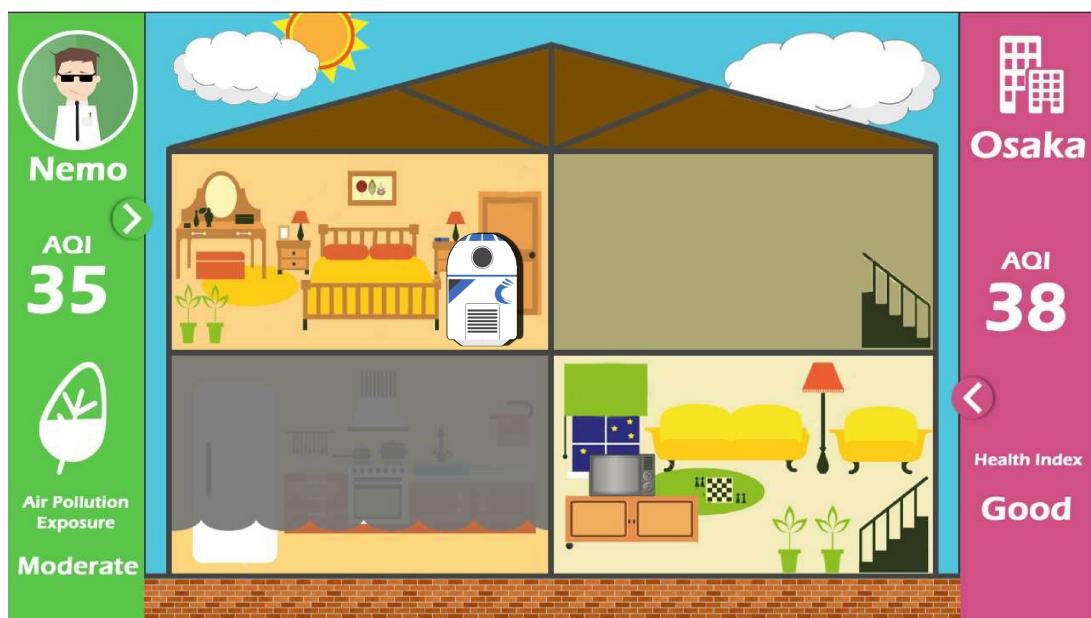
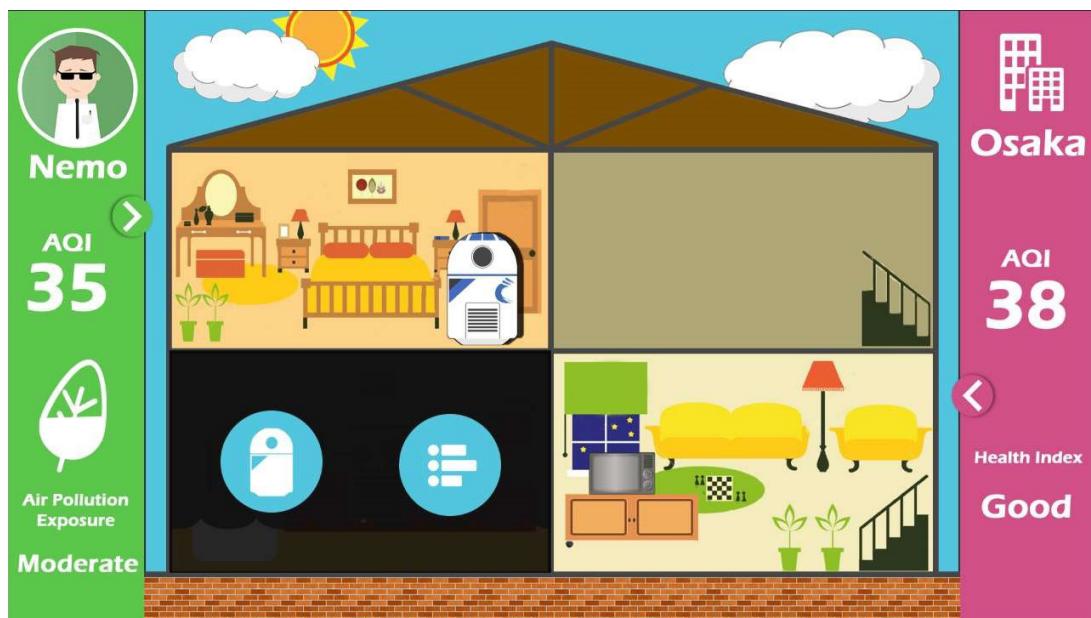
Version NO.

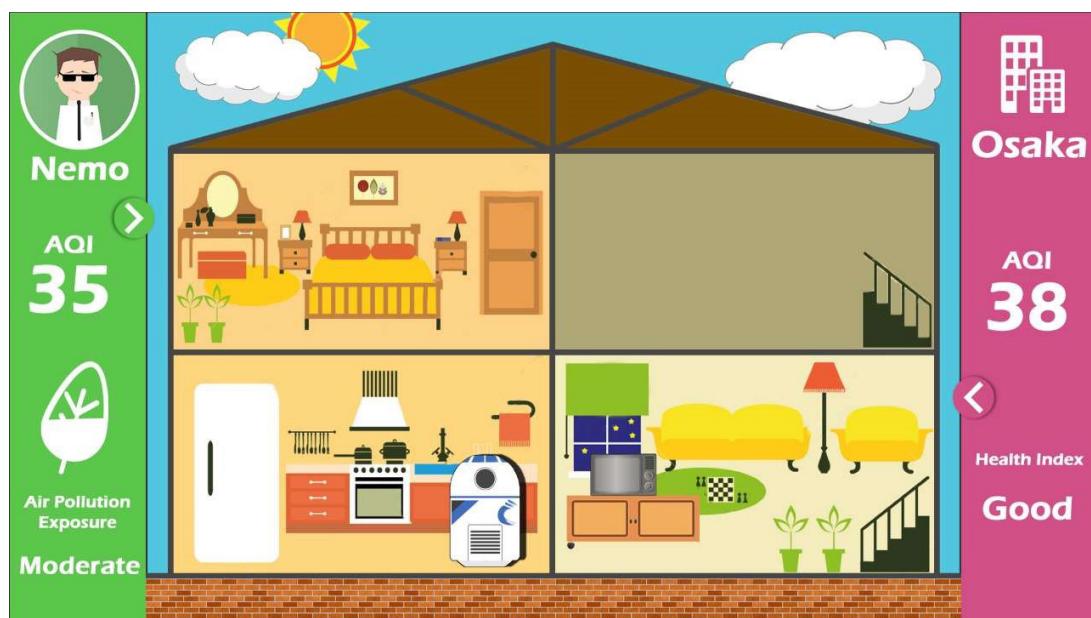
Submit

附录 C 设计稿

C.1 Smart Home 模块设计稿







参考文献

- [1] J. Radatz, A. Geraci and F. Katki. “*IEEE standard glossary of software engineering terminology*”. *IEEE Std, 1990, 610121990(121990)*: 30.
- [2] K. Beck *et al.* “*What is Agile Software Development*”. **2013**.
- [3] E. Marcotte. *Responsive web design*. Editions Eyrolles, **2013**.
- [4] R. T. Fielding and R. N. Taylor. “*Principled design of the modern Web architecture*”. *ACM Transactions on Internet Technology (TOIT)*, **2002**, 2(2): 115–150.
- [5] R. T. Fielding. *Architectural styles and the design of network-based software architectures* [phdthesis], **2000**.
- [6] D. Hardt. “*RFC6749-The OAuth 2.0 Authorization Framework. Oct. 2012*”. <https://tools.ietf.org/html/rfc6749>.
- [7] J. Bradley, N. Sakimura and M. Jones. “*JSON Web Token (JWT)*”. **2015**.
- [8] P. Teixeira. *Professional Node.js: Building Javascript based scalable software*. John Wiley & Sons, **2012**.
- [9] V. Driessen. “*A successful Git branching model*”. URL <http://nvie.com/posts/a-successful-git-branching-model>, **2010**.
- [10] E. Hubbard. “*One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.*” *Delhi Journal of Ophthalmology*, **2015**: 247.

谢 辞

感谢悉心指导论文设计和撰写的吴刚老师！
感谢提携我加入公司并在工作中给出指导和帮助的施宇晨同学！
感谢在工作中通力合作的傅浩南同学！