目录

[1 绪论 1](#_Toc6957321)

[2 课题总体介绍 2](#_Toc6957322)

[2.1 课题介绍 2](#_Toc6957323)

[2.2 课题目的及意义 2](#_Toc6957324)

[2.3 主要研究功能 2](#_Toc6957325)

[3 前端小程序开发分析 3](#_Toc6957326)

[3.1 微信小程序开发流程 3](#_Toc6957327)

[3.2 小程序界面以及数据渲染 3](#_Toc6957328)

[3.3 小程序端请求服务端数据 6](#_Toc6957329)

[3.3.1 配置说明 6](#_Toc6957330)

[3.3.2 请求数据的具体方法 6](#_Toc6957331)

[3.3.3 小程序图片上传 7](#_Toc6957333)

[4 管理页面开发分析 9](#_Toc6957334)

[4.1 前序准备 9](#_Toc6957335)

[4.2 项目目录结构和基础配置 9](#_Toc6957336)

[4.3 登录和权限管理流程 9](#_Toc6957337)

[4.3.1 登录操作 10](#_Toc6957338)

[4.3.2 权限验证操作 10](#_Toc6957339)

[4.3.3 控制页面路由 10](#_Toc6957340)

[4.3.4 为什么不选择完全由后端来控制页面路由 10](#_Toc6957341)

[4.3.5 具体的实现 11](#_Toc6957342)

[4.4 构建vue单页 11](#_Toc6957343)

[4.4.1 页面布局 11](#_Toc6957344)

[4.4.2 页面基础结构和内容 11](#_Toc6957345)

[4.5 和服务端进行交互 12](#_Toc6957346)

[4.6 构建和发布以及运行界面 13](#_Toc6957347)

[5 服务端开发分析 16](#_Toc6957348)

[5.1 选择node.js的原因以及环境搭建 16](#_Toc6957349)

[5.2 关于koa2的项目目录结构说明 16](#_Toc6957350)

[5.3 编写REST风格API 16](#_Toc6957352)

[5.3.1 接口和前端的交互 17](#_Toc6957353)

[5.3.2 接口规范 17](#_Toc6957355)

[5.3.3 接口实现 18](#_Toc6957361)

[5.4 数据库操作 18](#_Toc6957364)

[5.4.1 为什么选择MySQL 19](#_Toc6957365)

[5.4.2 如何访问数据库 19](#_Toc6957366)

[5.5 服务端构建 20](#_Toc6957373)

[结论 21](#_Toc6957374)

[参考文献 22](#_Toc6957375)

**基于node.js和vue.js的二手交易平台的设计和实现**

**摘 要**

近几年软件开发技术的发展重心从PC端转向了移动端，让日常生活中的很多事情变得更加便捷，加上绿色低碳节能的生活态度被越来越多的人所接受，这就是我选择做这个二手交易平台的主要原因。

项目使用到的技术栈包括：node.js，vue.js以及微信小程序。选择小程序作为客户端，管理后台选用vue搭建。后端则选择了性能优越的node.js。整个项目采用了最新的前后端分离的开发模式，让前端和后端做各自擅长的事，区别于混合开发。这样可以最大程度的降低耦合。

小程序的开放生态和技术成熟度使得开发异常顺利；后台管理系统使用vue的系列组件让登陆，鉴权等基础事物变得安全可扩展；后端利用node.js的灵活轻便结合REST（the Representational State Transfer）风格接口让服务端可以安全高效的处理数据和提供复杂服务。

**【关键词】**前后端分离 微信小程序 vue node.js REST架构风格

**Architectural Styles and the Design of Network-based Software Architectures**

**Abstract**

In recent years, the development of software development technology has shifted from PC to mobile, making many things in daily life more convenient, and the attitude of green, low-carbon and energy-saving is accepted by more and more people. This is I chose to do the main reason for this second-hand trading platform.

The technology stack used in this project includes: node.js, vue.js, and WeChat Mini Program. Select the Mini Program as the client, and the management background to use vue to build. The backend chose node.js with superior performance. The entire project adopts the latest development mode of front end and back end separated, so that the front end and the back end do their own good things, which is different from hybrid development. This minimizes coupling.

The open ecology and technical maturity of the applet make the development go smoothly; the back-end management system uses the vue series of components to make the basic things such as login and authentication become secure and scalable; the back-end uses the flexible and lightweight combination of node.js (the Representational The State Transfer style interface allows the server to process data and provide complex services safely and efficiently.

**【Key words】**Representational State Transfer (REST)  Front-end and back-end separated vue node.js WeChat Mini Program

1绪论

关于小程序简介：小程序是一种全新的连接用户与服务的方式，它可以在微信内被便捷地获取和传播，同时具有出色的使用体验。微信小程序并非凭空冒出来的一个概念。当微信中的 WebView 逐渐成为移动 Web 的一个重要入口时，微信就有相关的 JS API 了。小程序的主要开发语言是 JavaScript ，小程序的开发同普通的网页开发相比有很大的相似性。对于前端开发者而言，从网页开发迁移到小程序的开发成本并不高，但是二者还是有些许区别的。

​vue.js是一套构建用户界面的渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用单文件组件和Vue生态系统支持的库开发的复杂单页应用。

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。 Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。本项目采用了后端分离的开发模式，后端的api接口依据REST风格设计，符合现代Web架构模式。

2 课题总体介绍

2.1 课题介绍

本系统分为小程序用户端和平台后台管理两个部分。小程序端的开发设计遵循用户友好原则，方便用户发布二手商品和浏览购买商品以及购买后的操作包括：评论，点赞，买家秀等；后台管理提供给平台使用，主要功能包括：人员管理，二手商品信息管理等。

2.2 课题目的及意义

时代的发展，科技的进步，新旧物品的交替日新月异，每个家庭，每个人都会有更新过的物品在闲置，如何处理这些物品也成为了人们日常关注的问题。有人会拿到当地的二手交易市场去交易，有人会因离二手市场较远或二手物品不易搬运而把它当废品卖掉甚至扔掉。这不仅不利于环保，还浪费了社会资源。

由于二手交易平台的诞生就使得二手交易不再局限于一定要去二手交易市场实行交易。由于网络的方便性，让很大部分人选择了上二手交易平台发布二手买卖信息，足不出户就可以买卖二手，极大的方便了老百姓。可以说二手交易平台极大的推动了用户买卖二手的热情，也促进了社会资源的有效利用。

2.3 主要研究功能

具体的小程序端功能实现如下：

（1）小程序属于微信生态，通过微信授权来简化登录注册流程，具有良好的用户体验。

（2）用户发布二手商品，支持文字图片上传。

（3）首页通过轮播图等形势展示热门商品。

（4）用户发布的商品信息经过后台审核后在小程序端按类型进行展示。

（5）用户可通过关键字搜索指定的商品信息。

（6）用户可以在找到合适满意的商品后获取到卖家的联系方式。平台不参与买家和买家的交易。

（7）使用平台完成交易，买家可以在平台上传买家秀。

对于后台管理页面的具体功能如下：

（1）小程序端用户信息的管理。

（2）用户发布的二手商品信息的管理。

（3）首页轮播图和热门商品信息的管理。

（4）买家上传的买家秀信息管理。

3前端小程序开发分析

3.1微信小程序开发流程

开发小程序的第一步，需要拥有一个小程序 AppID，后续的所有开发流程会基于这个 AppID 来完成。小程序的注册非常简单，只需几个操作。使用浏览器打开 https://mp.weixin.qq.com/ 点击立即注册，在打开的页面中选择小程序后，填入相关的信息，就可以完成注册了

有了AppID 还需要下载安装小程序开发者工具，这些准备工作结束之后还需要了解小程序的目录结构以及如何配置：

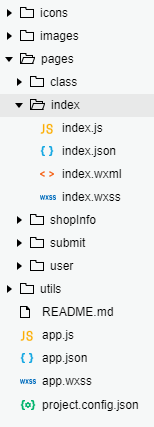


图3.1 小程序目录结构

小程序包含一个描述整体程序的 app 和多个描述各自页面的 page。一个小程序主体部分由三个文件组成，必须放在项目的根目录，其中app.js控制小程序逻辑；app.json包含小程序公共配置；app.wxss则是小程序公共样式表。一个小程序页面由四个文件组成，分别是：js 文件包含页面逻辑；wxml文件编写页面结构；json文件则是页面配置；wxss 文件包含页面样式表。

了解目录结构是所有开发的第一步，然后需要了解开发流程中最关键的一步：页面语法和API的使用。其中js文件不需要多说，就是JavaScrapt语法；wxml可以类比html；wxss则可以类比css。关于小程序的API都是微信提供的各种基础接口，主要涵盖路由，界面，网络，数据缓存，媒体，位置，转发，画布，文件，设备，Worker等。

3.2小程序界面以及数据渲染

小程序的页面渲染和vue很相似，采用数据绑定来进行条件和列表渲染，wxml 中的动态数据均来自对应 Page 的 data，其中还可以使用this.setData({})来更新数据。在wxml文件内，数据绑定使用 Mustache 语法（双大括号）将变量包起来。以首页的banners为例，在index.js文件中：

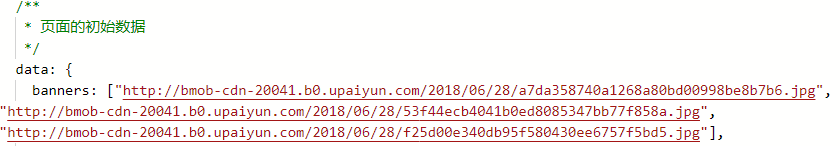


图3.2小程序index.js代码

在index.wxml文件内对应的渲染代码为：

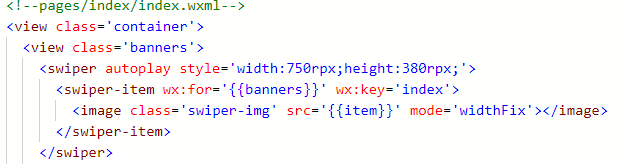


图3.3 小程序index.wxml代码

小程序在数据渲染之后就可以看到最终的运行界面如下：

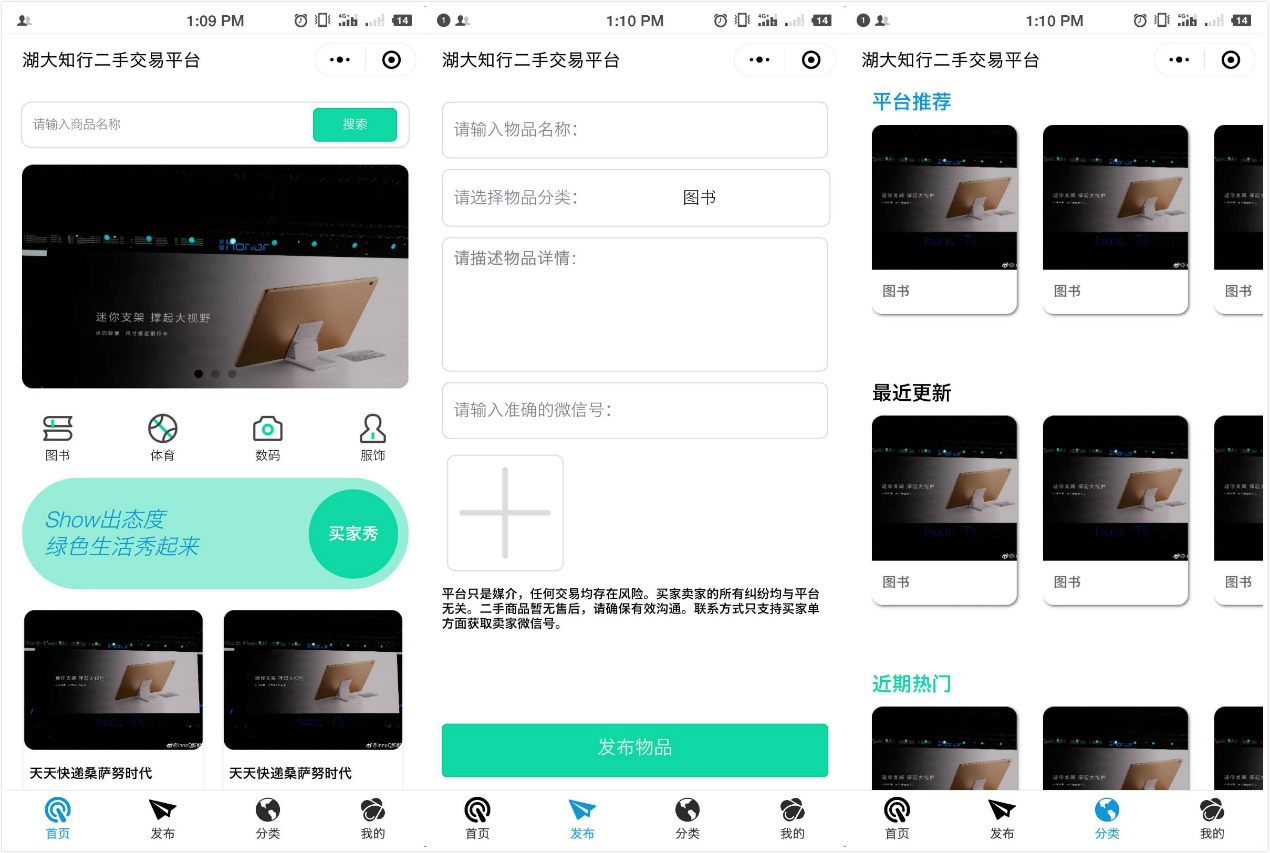


图3.4 小程序运行主页界面

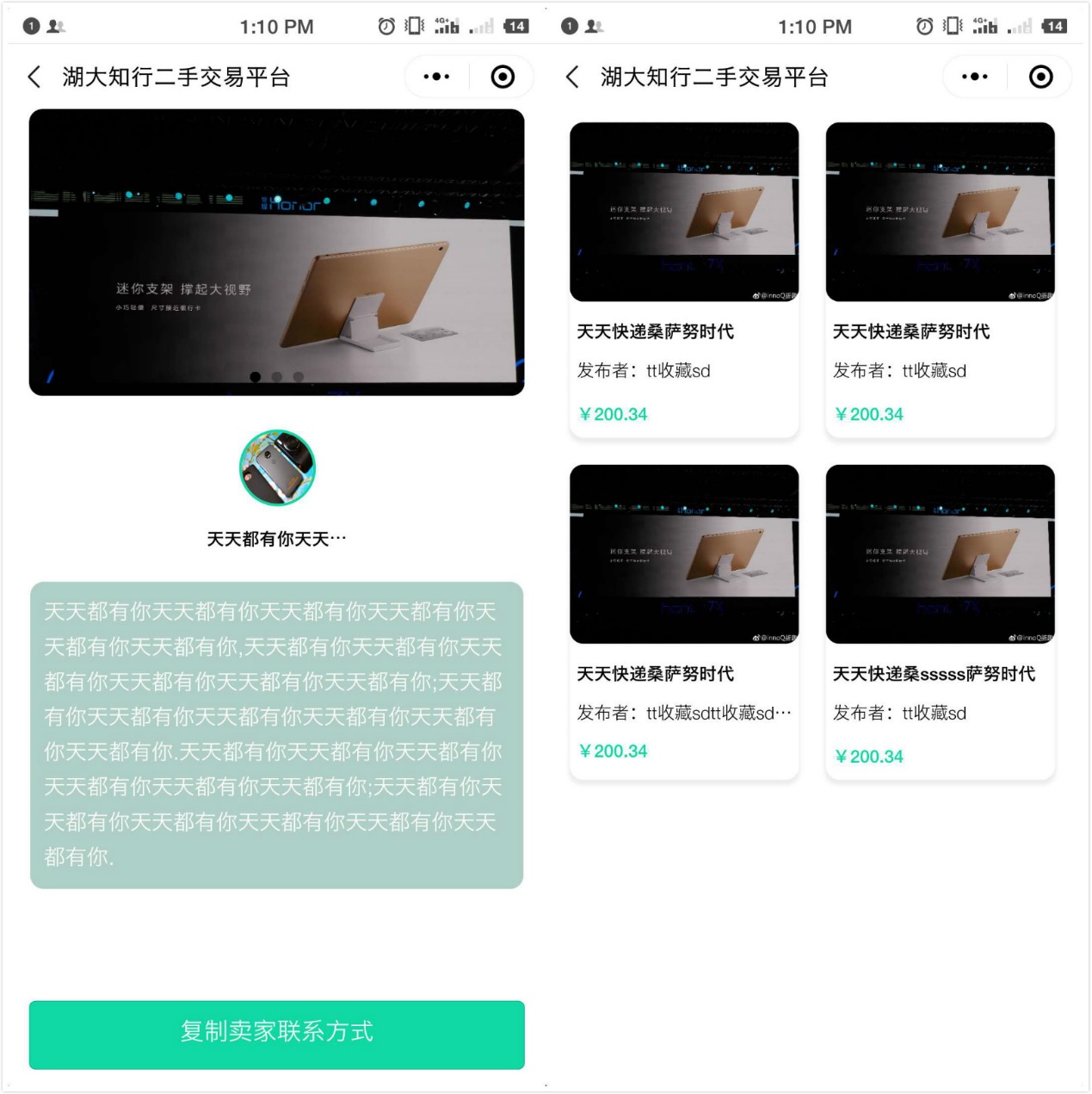


图3.5 小程序商品分类以及详情界面

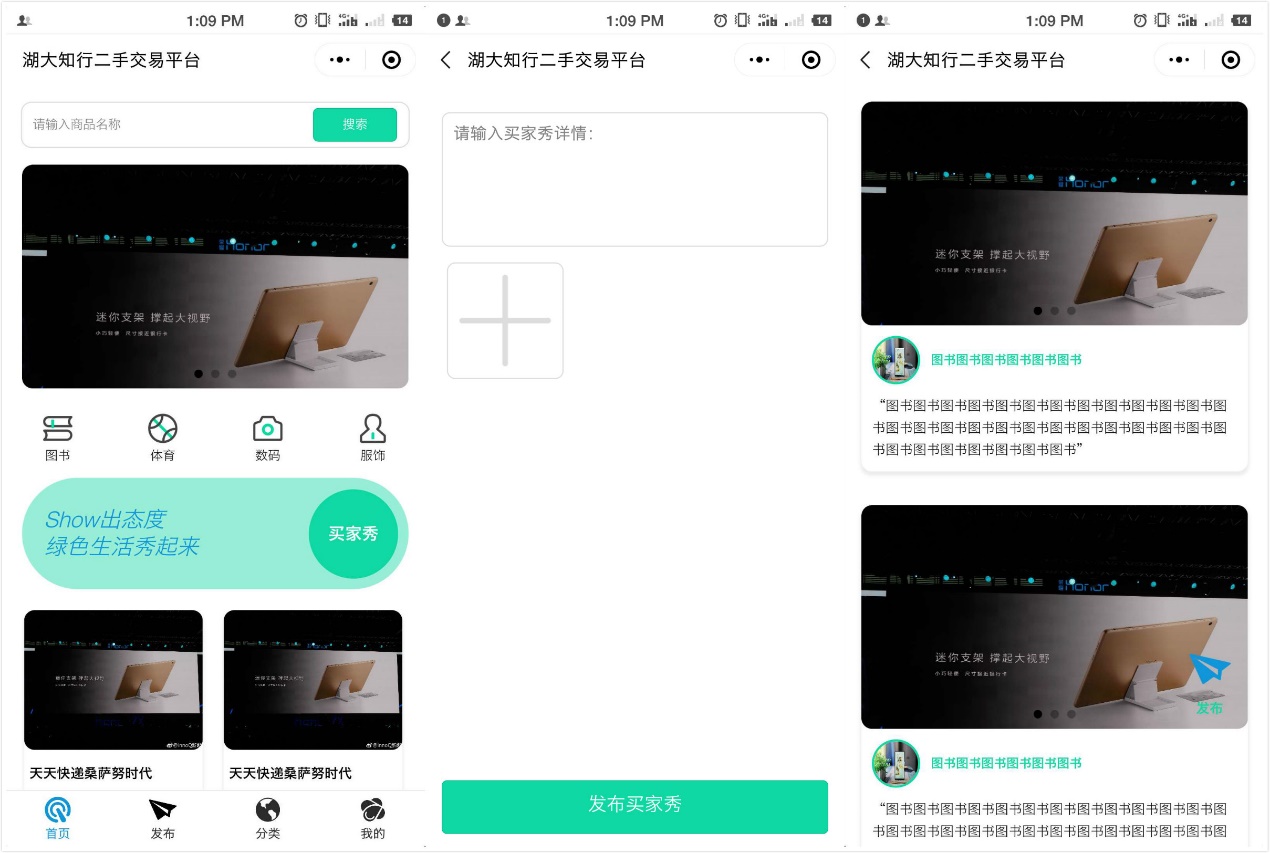


图3.6 小程序买家秀界面

3.3 小程序端请求服务端数据

小程序端不可能都是静态数据，更多的数据是需要从服务端请求的，出于安全考虑，小程序只可以跟指定的域名与进行网络通信。包括普通 HTTPS 请求（wx.request）、上传文件（wx.uploadFile）、下载文件（wx.downloadFile) 和 WebSocket 通信（wx.connectSocket）。

3.3.1配置说明

配置流程：服务器域名请在 「小程序后台-设置-开发设置-服务器域名」 中进行配置，配置时需要注意：域名只支持 https (wx.request、wx.uploadFile、wx.downloadFile) 和 wss (wx.connectSocket) 协议；域名不能使用 IP 地址或 localhost；可以配置端口，如 https://myserver.com:8080，但是配置后只能向 https://myserver.com:8080 发起请求。如果向 https://myserver.com、https://myserver.com:9091 等 URL 请求则会失败。

如果不配置端口。如 https://myserver.com，那么请求的 URL 中也不能包含端口，甚至是默认的 443 端口也不可以。如果向 https://myserver.com:443 请求则会失败。

域名必须经过 ICP 备案；

出于安全考虑，api.weixin.qq.com 不能被配置为服务器域名，相关API也不能在小程序内调用。 开发者应将 AppSecret 保存到后台服务器中，通过服务器使用 getAccessToken 接口获取 access\_token，并调用相关 API；对于每个接口，分别可以配置最多 20 个域名。

3.3.2请求数据的具体方法

小程序中使用wx.request()接口来发起 HTTPS 网络请求。具体的请求参数如下：

表1.1 wx.request()请求参数

|  |  |
| --- | --- |
| 属性 | 说明 |
| url | 接口地址 |
| data | 请求的具体参数 |
| header | 设置请求header |
| method | 请求的方法，包括GET,POST,PUT, DELETE等 |
| success | 请求成功的回调函数 |

关于data 参数说明：最终发送给服务器的数据是 String 类型，如果传入的 data 不是 String 类型，会被转换成 String 。转换规则如下：对于 GET 方法的数据，会将数据转换成 query string；对于 POST 方法且 header['content-type'] 为 application/json 的数据，会对数据进行JSON 序列化；对于 POST 方法header['content-type'] 为 application/x-www-form-urlencoded 的数据，会将数据转换成 query string。首页请求二手商品的代码如下：



图3.7 小程序wx.request()代码

3.3.3小程序图片上传

小程序中上传图片需要用到wx.chooseImage()接口和wx.uploadFile()接口。chooseImage接口作用是从本地相册选择图片或使用相机拍照。uploadFile作用是把本地选中的图片上传到制定服务器。

表1.2 wx.chooseImage()请求参数

|  |  |
| --- | --- |
| 属性 | 说明 |
| count | 最多可以选择的图片张数 |
| sizeType | 所选的图片的尺寸 |
| sourceType | 选择图片的来源 |
| success | 请求成功的回调函数 |

表1.3 wx.uploadFile()请求参数

|  |  |
| --- | --- |
| 属性 | 说明 |
| url | 开发者服务器地址 |
| filePath | 要上传文件资源的路径 |
| name | 文件对应的 key，开发者在服务端可以通过这个 key 获取文件的二进制内容 |
| header | HTTP请求 Header，Header 中不能设置 Referer |
| formData | HTTP 请求中其他额外的 form data |
| success | 请求成功的回调函数 |

二手物品图片上传时的代码如下：

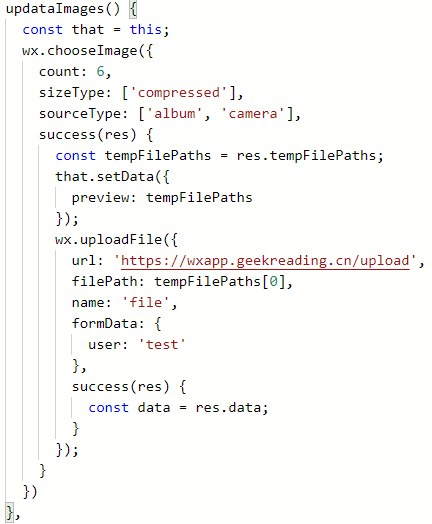


图3.8 小程序图片上传代码

4管理页面开发分析

4.1前序准备

管理页面使用vue前端框架开发，需要在本地安装node和git。本项目技术栈基于 ES2015+，vue，vuex，vue-router ，vue-cli，axios 和 element-ui，所有的请求数据都使用Mock.js进行模拟。

4.2项目目录结构和基础配置

build文件与构建相关；config文件包含配置相关；dist文件下是最终编译生成的页面内容；node\_modules文件是npm模块；src中是所有页面的源代码；static包括第三方不打包资源，例如：富文本；.babelrc文件是babel-loader 的配置文件；eslintrc.js是eslint 配置项；.gitignore是git忽略项；favicon.ico是页面的icon图标； index.html是首页的html模板；package.json是webpack打包的配置文件。src下的文件目录也在下面列出：api包含了所有请求；assets包含主题，字体等静态资源；components包含公用组件； directive包含全局指令；icons包含所以的icons；router是路由页面；store是全局store管理；syles包括全局样式；utils包含全部的公共方法；views所有页面的view都在这个文件下；App.vue入口页面；main.js入口加载组件初始化等。

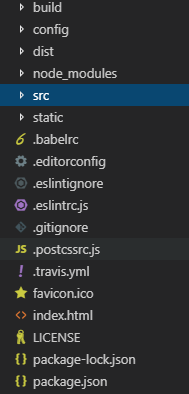
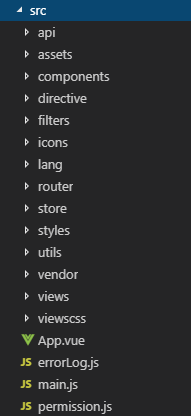
 

图4.1 管理页面目录结构

首先需要配置的是config下的dev和prod件，dev是指开发环境配置，而prod则是生产环境配置。使用module.exports来把配置参数以对象的形式暴露出来，其中包括了调用的api地址，项目名称，npm运行指令和短信接口是否启用。

4.3登录和权限管理流程

后台项目有别于其它的项目，权限验证与安全性是非常重要的，可以说是一个后台项目一开始就必须考虑和搭建的基础核心功能。本项目所要做到的是：不同的权限对应着不同的路由，同时侧边栏也需根据不同的权限，异步生成。这里先简单说一下，实现登录和权限验证的思路。

4.3.1登录操作

其中首先面对的是登录：当用户填写完账号和密码后向服务端验证是否正确，验证通过之后，服务端会返回一个token，拿到返回的token之后会将这个token存到浏览器的cookie中，保证刷新页面后能记住用户登录状态，然后前端会根据token再去拉取一个user\_info的接口来获取用户的详细信息例如：用户权限，用户名等。

4.3.2权限验证操作

再来说关于权限验证：通过token获取用户对应的role，动态根据用户的role算出其对应有权限的路由，通过router.addRoutes动态挂载这些路由。

上述所有的数据和操作都是通vuex全局管理控制的。需要补充说明的是：刷新页面后 vuex的内容也会丢失，所以需要重复上述的那些操作。

在接下来的每一次请求都是要带token来验证权限的，我的做法是在axios里进行封装，这样一来就不用每个请求手动来配置token，还可以在axios里做一些统一的异常处理，这是一劳永逸的做法。

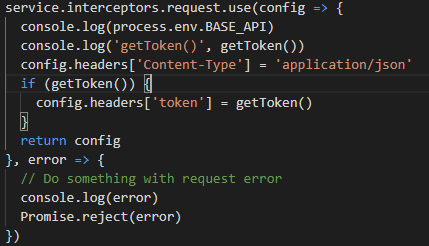


图4.2 在axios里封装token配置的代码

4.3.3控制页面路由

权限控制的主要思路在于页面路由。一般来说前端会有一份路由表，它表示了每一个路由可访问的权限。但是，要清楚这些控制都只是页面级别的，说到底前端来做权限控制不是绝对安全的，最终都需要后端来处理这些权限。

在现在的项目中，前端只是控制页面级权限，不同的权限下显示的侧边栏是完全不同的，后端则会在前端请求接口时验证这些操作。具体的验证方式为：每一个发给后台的请求，无论是GET还是POST都需要前端在请求的header里面携带用户token发给后端。之后，后端再根据该用户的token 来验证此用户是否有权限执行该操作。若没有权限则抛出一个对应的状态码，前端检测到该状态码，做出相对应的操作；如果有权限则得到数据进行其后的操作。

4.3.4为什么不选择完全由后端来控制页面路由

在很多项目中的路由表是于后端根据用户的权限动态生成的，那为什么本项目没有采取这种方式呢？原因如下：

首先，本项目计划会不断的迭代，如果前端开发一个新的页面还要让后端重新配置路由和权限，那就回到了曾经的前后端不分离时代，不符合本项目前后端分离的开发模式。

其次，在本项目中，虽然后端确实也有权限验证，但在后端的验证更多的时候是针对业务来划分的，但对于前端来说并不能按业务来划分权限。

还有一点是就在vue的2.2.0版本之前异步挂载路由是很麻烦的事。好在官方最终也给出了新的接口来优化异步挂载。

4.3.5具体的实现

首先在创建vue实例时将vue-router进行挂载，但这个时候vue-router挂载一些登录或者不用权限的公用的页面。

之后，当用户登录后，获取用role，将role和路由表每个页面的需要的权限作比较，生成最终用户可访问的路由表。

最后，调用router.addRoutes(store.getters.addRouters)添加用户可访问的路由。再使用vuex管理路由表，根据vuex中可访问的路由渲染侧边栏组件。

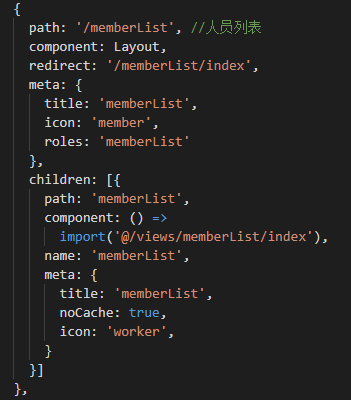


图4.3 页面路由的实现代码

4.4构建vue单页

本项目的UI选择了element-ui库，主要原因在于：有大厂背书（饿了么前端组开源项目） 。 尽管核心开发只有两到三个人，但至少不用担心哪天就不维护。

得益于vue的生态圈优异，社区活跃。这里还有一些其它的UI框架包括：ivew，个人开源框架，美观度和交互性都不错，有种介于Element和Ant之间的感觉。 vue-admin，也是一个不错的选择，vue官方的一套admin系统UI。

4.4.1页面布局

页面整体布局是一个产品最外层的框架结构，往往会包含导航、侧边栏、面包屑以及内容等。想要开始一个前端项目，先要确定基础布局。本项目大部分页面都是基于这个layout 的，除了个别页面如：login , 404, 401等页面没有使用该layout。如果你想在一个项目中有多种不同的layout也是很方便的，只要在一级路由那里选择不同的layout组件就可以了。

4.4.2页面基础结构和内容

vue单页的结构包括：style，template以及script标签。其中style控制页面样式；template控制页面结构；script控制页面逻辑和数据处理。

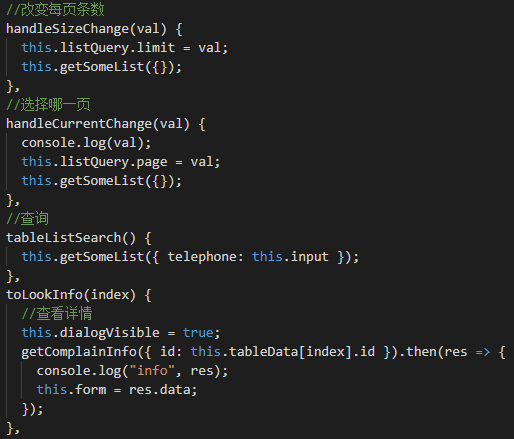


图4.4 单页中的查询代码

4.5和服务端进行交互

单页需要和服务端进行交互才可以动态获取数据。这也是前后端分离的核心。一个完整的前端从UI到服务端的交互处理流程大概是这样的：首先从UI 组件交互操作开始；触发调用统一管理的api service请求函数；然后再使用封装的request.js发送请求到服务器；之后就可以获取到服务端返回数据，本项目返回的数据统一为json格式；最终还需要前端更新 请求到的data，也就是数据；这就是一个完整的流程。

上面提及的这些请求都放在@/src/api文件内，使用来自request.js的统一封装好的请求方式。这样就可以实现全局的request拦截器，response拦截器以及统一的错误处理，统一做了超时处理，全局baseURL设置等。



图4.5 api中获取商品详情接口

4.6构建和发布以及运行界面

当项目开发完毕，只需要运行一行命令就可以打包应用：

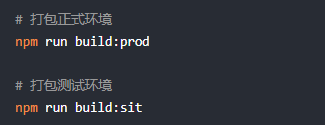


图4.6 打包命令

构建打包成功之后，会在根目录生成dist文件夹，里面就是构建打包好的文件，通常会包含有index.js ，index.css以及index.html等静态文件。到这一步就完成了整个二手交易平台项目的所有前端开发。以下是管理系统的运行界面：



图4.7 后台登录界面

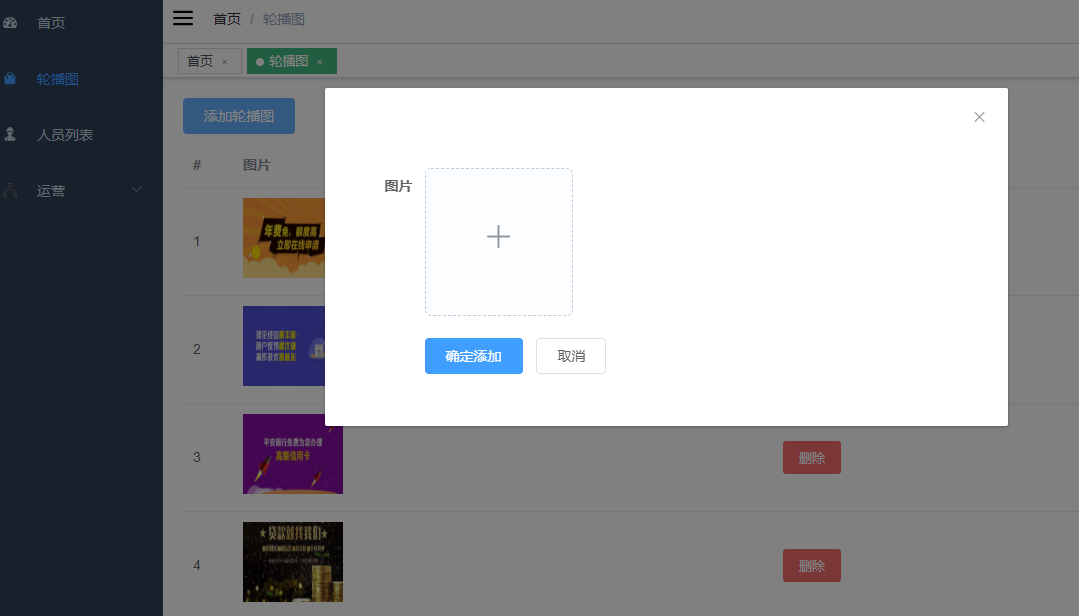


图4.8 后台首页轮播图

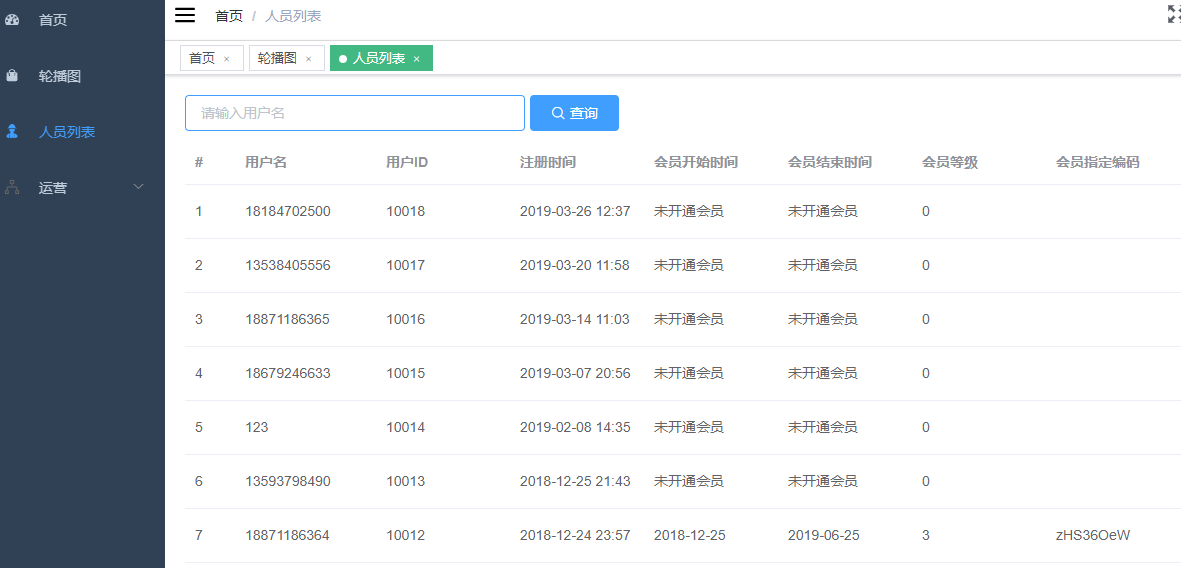


图4.9 后台人员管理



图4.10 后台二手商品管理



图4.11 后台买家秀管理



图4.12 后台物品分类管理

5服务端开发分析

5.1选择node.js的原因以及环境搭建

本项目服务端使用node.js语言进行开发，选择了比较流行的koa2框架。选择node.js的原因主要有两个：第一，本身我从事前端开发选用node.js没有重新学习语言的成本；其次在于node.js自身的性能很高，它是天生异步的，可以从性能方面节约更多的服务端资源。在众多的node.js框架中（包括：Express，Sails.js，koa，Meteor，DerbyJS，Total.js，restify等）选择koa2的原因在于相比koa第一个版本，新版支持最新的ES6和ES7的语法，异步的回调函数可以摆脱那种臃肿的地狱回调了，取而代之的是使用Promise对象来执行异步，或者使用更先进的async和await，可以轻松地把一个function变为异步模式。

关于环境搭建也非常简单。node.js依旧是遵循javascript语法，只是运行在node环境下，区别于在浏览器里运行的js。所以，首先需要安装node环境，同时会附带安装npm也就是node环境下的包管理工具。

5.2关于koa2的项目目录结构说明

后端项目的目录结果如下：

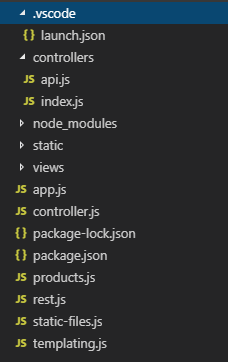


图5.1 服务端目录结构

其中包括：.vscode文件包含VScode IDE的配置文件；controllers/api.js接口文件也就是采用REST风格封装的api；controllers/index.js可以理解为MVC的控制器文件；node\_modules文件包含所有的npm包；static下是静态文件，例如页面引用的字体，css等；views下是页面文件；app.js是koa的项目入口；controller.js是扫描注册的控制器文件；package.json为项目的说明性配置文件。

5.3编写REST风格API

在本项目中最值得一提的就是前后端分离模式，其中的核心就是后端不再操作页面的DOM，而选择利用接口来返回json格式数据给前端，无论是小程序客户端还是管理页面都可以接收和发送json给服务端，因为本身json就是JavaScript的一种对象。

5.3.1接口和前端的交互

整个的前后端交互流程其实非常简单也非常直接，后台的管理系统也就是本项目的服务端可以把数据通过REST API以接口的形势暴露给客户端和管理页面，在本项目中客户端就是小程序，管理页面就是vue编写的pc页面。如下图所示：

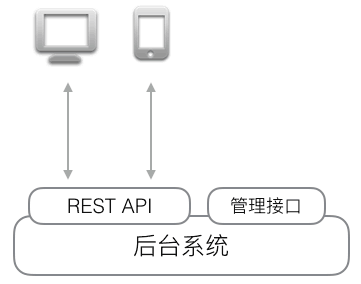


图5.2 前后端交互模式

5.3.2接口规范

由于选择使用REST风格进行接口封装，所以遵循REST的编写规范是前提，需要说明的是这些规范不是强制的，但是遵循规范可以让代码更易于理解方便扩展。

例如，商品Product就是一种资源。获取所有Product的URL如下：

GET /api/products

而获取某个指定的Product，例如，id为123的Product，其URL如下：

GET /api/products/123

新建一个Product使用POST请求，JSON数据包含在body中，URL如下：

POST /api/products

更新一个Product使用PUT请求，例如，更新id为123的Product，其URL如下：

PUT /api/products/123

删除一个Product使用DELETE请求，例如，删除id为123的Product，其URL如下：

**DELETE** /api/products/123

资源还可以按层次组织。例如，获取某个Product的所有评论，使用：

GET /api/products/123/reviews

当我们只需要获取部分数据时，可通过参数限制返回的结果集，例如，返回第2页评论，每页10项，按时间排序：

GET /api/products/123/reviews?page=2&size=10&sort=time

5.3.3接口实现

在koa中实现REST接口要做的第一步就是npm安装好需要的标准库，之后在app.js文件内引用这些库。做好准备工作就可以在api.js文件内编写api接口，时刻需要记得遵循上面的约定。

其实在上面的步骤中，我们可以看到一个重点就是app.use(bodyParser())这个语句，它给koa安装了一个解析HTTP请求body的处理函数。如果HTTP请求是JSON数据，我们就可以通过ctx.request.body直接访问解析后的JavaScript对象。

然后到api.js中我们就可以添加一个GET接口来请求数据库里的商品，代码如下：



图5.3 api.js中的GET接口代码

实际上暴露接口就是在处理函数的async，然后具体做法就是设置Content-Type和对应的设置Response Body这两项。

然后我们通过客户端来请求这个商品接口就可以得到如下的json数据：



图5.4 接口请求到的数据

5.4数据库操作

服务端的作用就是处理数据，那就绕不开数据库的设计和操作。node.js生态提供了关于操作MySQL数据库的驱动，只需要使用npm安装就可以使用。

5.4.1为什么选择MySQL

MySQL属于关系型数据库，现在有很多流行的NoSQL数据库，它们一般都宣称自己速度快，存储规模可以是MySQL的几十倍甚至上百倍，但是这些仅仅是在特定的场景下，而且拥有SQL语句的基础，上手MySQL可以节省更多的时间。

再来看在同类的关系型数据库里：Oracle，SQL Server，DB2这些都是需要付费的，再加上各自的生态有很大的局限性，相比之下的MySQL既有免费版本可选，又拥抱开源，生态非常棒。所以，选择MySQL的理由很充分了。

5.4.2如何访问数据库

在node.js里访问MySQL数据库主流方法就是利用ORM技术（Object-Relational Mapping），简单讲就是把数据库里的表结构映射在了js的对象上，这样一来就可以避免去写复杂而冗余的底层SQL语句。在本项目中就选用了Sequelize框架，正如上面提到的Sequelize就是一个基于ORM的node框架。

编码之前需要做的是编写一个MySQL的配置文件，具体内容如下：



图5.5 MySQL配置文件内容

之后创建一个Sequelize实例，具体代码如下：

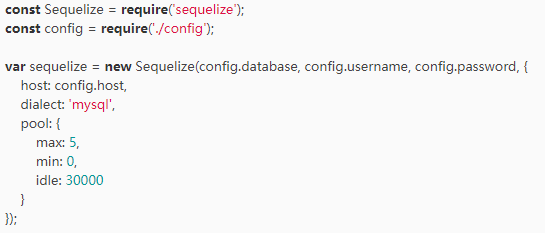


图5.6 Sequelize实例

最后还需要给Sequelize对应一个数据表映射，也可以称为定义模型：

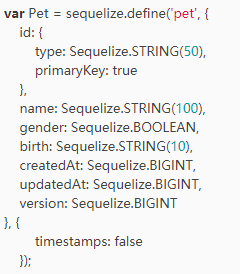


图5.7 建立映射关系

到这里为止后端就实现了数据库操作，并且这样的效率是远远超过写SQL语句的。

5.5服务端构建

服务端的作用就在于处理数据并暴露接口给小程序和后台管理页面，现在这些步骤都已经完成。最后只需要上传到云服务器在命令行里执行npm run app.js指令前端就可以请求数据。

结论

一：微信小程序的页面布局最好选择flex布局，从设备兼容性上来看flex布局无论在低版本的iOS设备还是在屏幕不规则的安卓设备上都不会出现显示问题。

二：从产品交互的角度来说，客户端的一些按钮，icon的可点击区域可以适当做大，因为在移动设备上的交互不同于pc端，可能出现无法点击的情况。

三：无论是在服务端还是在vue项目中尽量不要使用国内的npm镜像来安装模块或者包，可能会出现意想不到的麻烦。

四：后端管理页面在添加新的路由时，最后可以和新的views页面文件一一对应，有助于二次开发或者项目重构。

五：在编写REST风格的接口时要区分它与HTTP请求是不同的，主要在于REST请求的Content-Type为application/json。当然返回给客户端的也是json格式的数据。

参考文献

[1] 高性能MySQL（第3版）. 编者: (美)Baron Schwartz，(美)Peter Zaitsev，(美)Vadim Tkachenko . 译者:宁海元，周振兴，彭立勋. 电子工业出版社,2013

[2] Vue.js实战.编者：梁灏. 清华大学出版社,2017

[3] JavaScript设计模式.编者：(美)Addy Osmani. 人民邮电出版社,2013

[4] Node.js实战 第2版.编者：[英] 亚历克斯•杨（Alex Young），布拉德利•马克（Bradley Meck），[美] 麦克•坎特伦（Mike Cantelon） 著，吴海星 译. 人民邮电出版社,2018

[5] 深入理解Nginx：模块开发与架构解析（第2版）.编者：陶辉著. 机械工业出版社,2016

[6] Hands-On Microservices with Node.Js.编者：Diogo Resende & 著. 出版社Packt Publishing,2018

[7] Vue.js in Action.编者：Eric Hanchett / Benjamin Listwon.出版社Manning Publications,2018

[8] Web前端开发技术实验与实践——HTML5、CSS3、JavaScript.编者：储 久良 著 出版社: 清华大学出版社,2018

[9] 重构JavaScript.编者: [美] 埃文 伯查德 著. 中国电力出版社,2018.

[10] 算法导论（原书第3版）.编者：[美] Thomas H.Cormen，[美] Charles E.Leiserson，[美] Ronald L.Rivest，[美] Clifford Stein 著，殷建平，徐云，王刚 等 译. 机械工业出版社,2012

[11] 数据产品设计.编者：艾达 著. 电子工业出版社,2017

[12] 数据库系统：设计、实现与管理（基础篇）（原书第6版）.编者：[英] 托马斯•康诺利（Thomas Connolly），卡洛琳•贝格（Carolyn Begg） 著，宁洪 译. 机械工业出版社,2016

[13] MySQL 8 Cookbook.编者：[美] Karthik，Appigatla 著.电子工业出版社,2018

附录:

整个项目系统的将最新的web开发流程展现了出来，无论是前后端分离的设计模式，还是后端采用REST接口设计风格，都是现在的主流做法。关于系统稳定性在最初的后端技术选型就考虑到了，node.js在这方面可以说是当之无愧的语言。

关于客户端的选择，也是经过多次考虑的。选择使用微信小程序最重要的就在于可以兼顾安卓和iOS用户，不需要去开发两个系统下的客户端APP。后端管理系统的选择则是因为考虑到登陆流程和vue拥有很棒的生态。

致 谢

本次的毕业设计是在龙文佳老师的悉心的指导和帮助下完成的。在这近三个月的开发过程中，我们占用了老师的很多时间，给她添了很多麻烦。但是龙老师依旧没有厌烦，每次都很耐心地给我们讲解、纠错。可以说，老师在我们这次的毕业设计中花费了很大的心血。在她的耐心指导下，我们系统地学习了软件开发的过程，从需求分析、总体设计到详细设计，我们由一无所知到比较熟练的掌握点点滴滴都离不开老师的指教！在此，我代表我们向老师表示衷心的感谢，谢谢他这两个月的帮助与指导！

四年时间很快过去了，我即将毕业，心中有万般的不舍。四年来，在班主任及系、院领导的关心帮助下，我不仅学到了很多专业知识，也学到了很多为人处世的道理。在此，我深深地感谢四年来帮助过、关心过我的所有老师！

同时，我的各个方面的进步，也离不开全体同学在学习、生活各方面的支持与帮助。所以，我也要向他们表示由衷的感谢。

**架构风格与基于网络的软件架构设计**

原文来源：**Architectural Styles and the Design of Network-based Software Architectures**

译文正文：

万维网（World Wide Web）的成功，很大程度上是因为其软件架构的设计满足了 Internet 规模（Internet-scale）的分布式超媒体系统的需求。在过去的 10 年间，通过对定义 Web 架构的标准所做的一系列修改，Web 以迭代的方式不断地发展着。为了识别出 Web 需 要改善的那些方面，并且避免对其进行不想要的修改，必需要有一种现代 Web 架构的模型， 用来指导 Web 的设计、定义和部署。 软件架构的研究探索了如何以最佳的方式划分一个系统、如何标识组件、组件之间如何 通信、信息如何沟通、系统的元素如何能够独立地进化，以及上述的所有东西如何能够使用 形式化的和非形式化的符号加以描述。我的工作的动机是希望理解和评估基于网络的应用的 架构设计，通过有原则地使用架构约束，从而从架构中获得所希望的功能、性能和社会学几 方面的属性。一种架构风格是一组已命名的、协作的架构约束。 这篇论文定义了一个框架，致力于通过架构风格来理解软件架构，并且展示如何使用风 格来指导基于网络的应用的架构设计。本文使用了一个对基于网络的应用的架构风格的调查， 根据不同的风格在分布式超媒体的架构中所导致的架构属性，来对这些风格进行分类。然后 我介绍了表述性状态转移（Representational State Transfer，REST）的架构风格，并且描述了 如何使用 REST 来指导现代 Web 架构的设计和开发。 REST 强调组件交互的可伸缩性、接口的通用性、组件的独立部署、以及用来减少交互 延迟、增强安全性、封装遗留系统的中间组件（intermediary components）。我描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架构风格的约 束进行了对比。最后，我描述了从在超文本转移协议（HTTP）和统一资源标识符（URI） 的标准中应用 REST，以及从这两个标准在 Web 客户端和服务器软件的后续部署等过程中学 到的经验教训。

正如 Perry 和 Wolf 的预言，软件架构成为了 20 世纪 90 年代软件工程研究的焦点。由于 现代软件系统的复杂性，更加有必要强调组件化的系统，其实现被划分为独立的组件，这些 组件通过相互通信来执行想要完成的任务。软件架构的研究探索了如何以最佳方式划分一个 系统、如何标识组件、组件之间如何通信、信息如何沟通、组成系统的元素如何能够独立地 进化，以及上述的所有东西如何能够使用形式化的和非形式化的符号加以描述。 一个优秀的架构并非凭空想象。所有架构级的设计决策应该根据被设计系统的功能、行 为和社会学等方面的需求来作出，这是一个原则，既适用于软件架构，同样也适用于传统的 建筑架构领域。“形式追随功能”的指导方针来自从数百年失败的建筑项目中获得的经验， 但是却常常被软件从业者忽视。上面引用的那句滑稽搞笑的话来自于 Monty Python 系列讽刺 剧，这是当一个建筑师在面对设计一个城市公寓区的目标时，头脑里所抱有的荒诞想法。他 想使用所有的现代屠宰场的组成部分来完成这个设计。这也许是他所构思过的最棒的屠宰场， 但是对于预期的房客来说却谈不上舒适，他们不得不战战兢兢地行穿行在安装着旋转刀头的 走廊中。 《建筑师讽刺剧》里的夸张说法也许看似荒唐可笑，但是考虑到我们是如此频繁地看到 软件项目一开始就采用最新时髦的架构设计，到后来却发现系统的需求实际上并不需要这样 一种架构。design-by-buzzword（按照时髦的词汇来做设计）是一种常见的现象。至少在软 件行业中，很多此类行为是由于对一组特定的架构约束为什么是有用的缺乏理解。换句话说， 当选择那些优秀的软件架构来重用时，这些架构背后的推理过程（reasoning），对于设计者 来说并非是显而易见的。 这篇论文探索了在计算机科学的两个研究学科（软件和网络）边界上的连接点。软件研 究长期以来关注软件设计的分类和对于设计方法学的开发，但是极少能够客观地评估不同的 设计选择对于系统行为的影响。网络研究则恰恰相反，集中于系统之间普通的通信行为的细 节和提高特殊通信技术的性能，却常常忽略了一个事实，即改变一个应用的交互风格对于性 能产生的影响要比改变交互所使用的通信协议更大。我的工作的动机是希望理解和评估基于 网络的应用的架构设计，通过有原则地使用架构约束，从而从架构中获得所希望的功能、性 能和社会学几方面的属性。当给定一个名称时，一组协作的架构约束就成为了一种架构风格。 这篇论文的前三章定义了一个通过架构风格来理解软件架构的框架，揭示了架构风格如 何能够被用来指导基于网络的应用的架构设计。当将常见的架构风格应用于基于网络的超媒 体的架构时，将会导致一系列架构属性，根据这些架构属性来对架构风格进行调查和分类。 然后使用得到的分类来识别出一组能够改善早期万维网的架构的架构约束。 如同我们在第 4 章中所讨论的，设计 Web 的架构就必须要理解 Web 的需求。Web 是旨 在成为一个 Internet 规模的分布式超媒体系统，这意味着它的内涵远远不只仅仅是地理上的 分布。Internet 是跨越组织边界互相连接的信息网络。信息服务的提供商必须有能力应对无 法控制（anarchic）的可伸缩性的需求和软件组件的独立部署。通过将动作控制（action controls）内嵌在从远程站点获取到的信息的表述之中，分布式超媒体为访问服务提供了一 种统一的方法。因此 Web 的架构必须在如下环境中进行设计，即跨越高延迟的网络和多个 可信任的边界，以大粒度的（large-grain）数据对象进行通信。 第 5 章介绍并详细描述了为分布式超媒体系统设计的表述性状态转移（REST）的架构 风格。REST 提供了一组架构约束，当作为一个整体来应用时，强调组件交互的可伸缩性、接口的通用性、组件的独立部署、以及用来减少交互延迟、增强安全性、封装遗留系统的中 间组件。我描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它 们与其他架构风格的约束进行了对比。 如第 6 章中所展示的那样，在过去的 6 年间，REST 架构风格被用来指导现代 Web 的架 构的设计和开发。这个工作与我所创作的超文本转移协议（HTTP）和统一资源标识符 （URI）的 Internet 标准共同完成，这两个规范定义了被所有 Web 之上的组件交互所使用的 通用的接口。 就像大多数真实世界中的系统一样，并非所有已部署的 Web 架构的组件都服从其架构 设计中给出的每一个约束。REST 既被用来作为定义架构改进的一种方法，也被用来作为识 别架构不匹配（mismatch）的一种方法。当由于无知或者疏忽，一个软件实现以违反架构约 束的方式来部署时，就会发生不匹配。尽管不匹配通常无法避免，但是有可能在它们定型之 前识别出它们。在第 6 章中总结了几种在现代 Web 架构中的不匹配情况，并且对它们为何会 出现和它们如何偏离 REST 进行了分析。 概括来说，这篇论文对于 Internet 和计算机科学领域的软件研究作出了如下贡献： ● 一个通过架构风格来理解软件架构的框架，包括了一组相容的术语，用来描述软 件架构； ● 通过当某种架构风格被应用于一个分布式超媒体系统的架构时，它将导致的架构 属性，来对基于网络的应用的架构风格进行分类。 ● REST，一种新颖的分布式超媒体系统的架构风格；以及 ● 在设计和部署现代万维网的架构的过程中，应用和评估 REST 架构风格。

尽管软件架构作为一个研究领域吸引了很多人的兴趣，但是对于什么应该被纳入架构的 定义，研究者们几乎从未达成过共识。在很多情况下，这导致了在过去的研究中忽视了架构 设计的一些重要的方面。本章在检查文献中现存的定义和我自己在基于网络的应用的架构方 面的洞察力的基础上，定义了一套自相容（self-consistent）的软件架构术语。每个定义使用 方框突出显示，随后讨论该定义如何得来，或者与相关研究进行比较。

软件架构的核心是抽象原则：通过封装来隐藏系统的一些细节，从而更好地识别和支持 系统的属性[117]。一个复杂的系统包含有多层的抽象，每一层抽象都有自己的架构。架构代 表了在某个层次上系统行为的抽象，架构的元素被描述为提供给同层的其他元素的抽象接口 [9]。在每一个元素之中，也可能还存在着另一个架构，定义了子元素的系统，这个系统实 现了由父元素的抽象接口所展示的行为。这样的架构可以递归下去直到最基本的系统元素： 它们不能再被分解为抽象层次更低的元素。 除了架构的层次，软件系统通常拥有多个操作阶段，例如启动、初始化、正常处理、重 新初始化和停止。每个操作阶段都有自己的架构。例如，配置文件在启动阶段会被当作架构 的一个数据元素来处理，但是在正常处理阶段则不会当作一个架构元素，因为在这个阶段这 些信息已经分布到了系统中的各处。事实上，配置文件也有可能定义了正常处理阶段的架构。 系统架构的一个整体描述必须既能够描述各个阶段的系统架构的行为，也能够描述在各个阶 段之间的架构的迁移。 Perry 和 Wolf [105]将处理元素定义为“数据的转换”，而 Shaw 等人[118]则将组件描述 为“计算和状态的所在地”。Shaw 和 Clements [122] 进一步指出：“组件是在运行时执行某 种功能的软件单元。这样的例子有程序、对象、进程、过滤器。”这引出了软件架构 （software architecture）和通常所说的软件结构（software structure）之间的一个重要的区别： 软件架构是软件系统在运行时的抽象，而软件结构则是静态源代码的属性。尽管将源代码的 模块化结构与正在运行的系统中的行为部件对应起来是有好处的，使用相同的代码部分（例 如共享库）来实现独立的软件组件也有很多好处。我们将软件架构和源代码结构分离开来是 为了更好的关注软件运行时的特性，这些特性不依赖于一个特定的组件实现。因此，尽管架 构的设计和源代码结构的设计关系密切，它们其实是分离的设计活动。不幸的是，有些软件 架构的描述并没有明确指出这个区别。

Perry 和 Wolf [105] 对软件架构的范围和知识基础进行了全面的检查，他们提出了一个 模型，将软件架构定义为一组架构元素，这些元素具有通过一组基本原理（rationale）来描 述的特殊形式。架构元素包括处理、数据、连接元素。形式则由元素的属性和元素之间的关系（即元素之上的约束）来定义。这些基本原理通过捕获选择架构风格、选择元素和形式的 动机 ，为架构提供了底层的基础。 我的软件架构定义是在 Perry 和 Wolf [105]的模型基础之上的一个更加详尽的版本，但 是不包括基本原理部分。尽管基本原理是软件架构研究中很重要的一个方面，尤其是在架构 的描述方面，但是将它包括在软件架构的定义中，将会暗示设计文档是运行时系统的一部分。 是否包括基本原理能够影响一个架构的开发，但是架构一旦建成，它将脱离其所基于的原理 而独立存在。反射型的系统[80]能够根据过去的性能改变今后的行为，但是这样做是用一个 更低层次的架构替换另一个更低层次的架构，而不是在那些架构中包含基本原理。 用一个类比来说明，想象一下当一个大楼的蓝图和设计计划图被烧毁了将会发生什么事 情？大楼会瞬间倒塌么？不会，因为支撑着楼顶的墙体仍然保持完好无损。按照设计，一个 架构拥有一组属性，允许该架构满足甚至超出系统的需求。忽视这些属性，在将来的修改中 可能会违反架构的约束，就好像用一面大型窗户取代承重墙会破坏大楼结构的稳定性一样。 所以，我们的软件架构定义中没有包括基本原理，而是包括了架构的属性。基本原理说明了 这些属性，缺少基本原理可能会导致架构随时间的推移逐渐退化，但是基本原理本身并不是 架构的一部分。 Perry 和 Wolf [105]的模型中的一个关键特征是不同类型的元素之间的区别。处理元素 （processing elements）是执行数据转换的元素，数据元素（data elements）是包含被使用和 被转换的信息的元素，连接元素（connecting elements）是将架构的不同部分结合在一起的 粘合剂。我将使用更加流行的术语：组件（components）和连接器（connectors）来分别表 示处理元素和连接元素。 Garlan 和 Shaw [53]将系统的架构描述为一些计算组件和这些组件之间的交互（连接 器）。这一模型是对于 Shaw 等人的模型[118]的扩展：一个软件系统的架构按照一些组件和 这些组件之间的交互来定义。除了指出了系统的结构和拓扑以外，架构还显示出了想要获得 的系统需求和构建系统的元素之间的对应关系。更加详细的定义可以在 Shaw 和 Garlan [121] 的文章中找到。 Shaw 等人的模型的令人惊讶之处，是他们将软件架构的描述当作成是架构本身，而不 是将软件的架构定义为存在于软件之中。在这个过程中，软件架构被简化为通常在大多数非 形式化的架构图表中能够看到的东西：方框（组件）和直线（连接器）。数据元素和其他很 多真实软件架构的动态方面都被忽略了。这样的一个模型是不足以描述基于网络的软件架构 的，因为对于基于网络的应用而言，数据元素在系统中的位置和移动常常是系统行为唯一至 关重要的决定因素。

组件是软件架构中最容易被识别出来的方面。在 Perry 和 Wolf [105]的定义中，处理元 素被定义为提供对于数据元素的转换的组件。Garlan 和 Shaw [53]将组件简单描述为执行计 算的元素。我们的定义试图更加精确地将组件和连接器之中的软件（software within connectors）区分开来。 组件是软件指令和内部状态的一个抽象单元，通过其接口提供对于数据的转换。转换的 例子包括从二级存储将数据加载到内存、执行一些运算、转换为另外一种格式、使用其他数 据来封装等等。每个组件的行为是架构的一部分，能够被其他组件观察到（observed）或看 到（discerned） [9]。换句话说，组件应该由它为其他组件提供的接口和服务来定义，而不是由它在接口之后的实现来定义。Parnas [101]将此定义为其他架构元素能够对该组件作出的 一组假设。

Perry 和 Wolf [105] 将连接元素模糊地描述为将架构的不同部分结合在一起的粘合剂。 Shaw 和 Clements [122]提供了一个更加精确的定义：连接器是是对于组件之间的通讯、协调 或者合作进行仲裁的一种抽象机制。连接器的例子包括共享的表述、远程过程调用、消息传 递协议和数据流。 也许理解连接器的最佳方式是将它们与组件加以对比。连接器通过将数据元素从它的一 个接口转移（transferring）到另一个接口而不改变数据，来支持组件之间的通信。在其内部， 一个连接器可以包含一个由组件组成的子系统，为了转移的目的对数据进行某种转换、执行 转移、然后做相反的转换并交付与原始数据相同的结果。然而，架构所捕获到的外部行为的 抽象可以忽略这些细节。与之相反，从外部的角度观察，组件可以（尽管并非总是）对数据 进行转换。

上面已经提到，是否有数据元素是 Perry 和 Wolf [105]所提出的模型与大多数其他软件 架构研究所提出的模型[1, 5, 9, 53, 56, 117-122, 128]之间的最大区别。Boasson [24] 批评当前 的软件架构研究过于强调组件的结构和架构开发工具，他建议应该把注意力更多地放在以数 据为中心的架构建模上。Jackson [67]也有相似的观点。 数据是组件通过一个连接器接收或发送的信息元素。数据的例子包括字节序列、消息、 编码过的参数、以及序列化过的对象，但是不包括那些永久驻留或隐藏在组件中的信息。从 架构的角度来说，一个“文件”其实是一种转换，文件系统组件从它的接口接收到的一个“ 文件名”数据，将该数据转换为记录在（隐藏的）内部存储系统中的字节序列。组件也能够 生成数据，例如一个时钟或传感器的软件封装。 数据元素在基于网络的应用的架构中是一个无法避免的天性，这往往决定了一个特定的 架构风格是否是合适的。在对移动代码设计范例（mobile code design paradigms）的比较中 [50]尤其明显，在这个场景中你必须要在两种风格中做出选择：是直接与组件进行交互；还 是将组件转换为一个数据元素，通过网络转移，然后转换回一个能够在本地与之交互的组件。 不在架构层面上考虑数据元素，是完全不可能评估这样的一个架构的。

Abowd 等人[1]将架构的描述定义为根据三个基本的语义类来对系统进行描述：组件— —计算的所在地；连接器——定义组件之间的交互；配置——相互交互的组件和连接器的集 合。可以使用多种与特定风格相关的形象化符号来可视化地展示这些概念，便于描述合法的 计算和交互、以及想要得到的系统约束。

严格来说，你可能会认为一个配置等价于一组组件交互之上的特定约束。例如，Perry 和 Wolf [105] 在他们的架构形式关系（architectural form ralationships）的定义中包括了拓扑。 然而，将主动的拓扑与更加通用的约束分离开，使得一个架构师更容易将主动的配置与所有 合法配置可能影响的领域区分开。Medvidovic 和 Taylor [86]给出了在架构描述语言中额外用 来对配置进行区分的基本原理。

软件架构的架构属性集合包括了对组件、连接器和数据的选择和排列所导致的所有属性。 架构属性的例子包括了可以由系统获得的功能属性和非功能属性，例如：进化的相对容易程 度、组件的可重用性、效率、动态扩展能力；这些常常被称作品质属性（quality attributes [9]）。 属性是由架构中的一组约束所导致的。约束往往是由在架构元素的某个方面应用软件工 程原则[58]来驱动的。例如，统一管道和过滤器（uniform pipe-and-filter）风格通过在其组件 接口之上应用通用性（generality）原则——强迫组件实现单一的接口类型，从应用中获得了 组件的可重用性和可配置性的品质。因此，架构约束是由通用性原则所驱动的“统一组件接 口”，目的是获得两个想要得到的品质，当在架构中实现了这种风格时，这两个品质将成为 可重用和可配置组件的架构属性。 架构设计的目标是创建一个包含一组架构属性的架构，这些架构属性形成了系统需求的 一个超集。不同架构属性的相对重要性取决于想要得到的系统本身的特性。2.3 节检查了那 些对于基于网络的应用的架构特别重要的属性。

原文正文：The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia system. The Web has been iteratively developed over the past ten years through a series of modifications to the standards that define its architecture. In order to identify those aspects of the Web that needed improvement and avoid undesirable modifications, a model for the modern Web architecture was needed to guide its design, definition, and deployment. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations. My work is motivated by the desire to understand and evaluate the architectural design of networkbased application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. An architectural style is a named, coordinated set of architectural constraints. This dissertation defines a framework for understanding software architecture via architectural styles and demonstrates how styles can be used to guide the architectural design of network-based application software. A survey of architectural styles for network-based applications is used to classify styles according to the architectural properties they induce on an architecture for distributed hypermedia. I then introduce the Representational State Transfer (REST) architectural style and describe how REST has been used to guide the design and development of the architecture for the modern Web. REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. Finally, I describe the lessons learned from applying REST to the design of the Hypertext Transfer Protocol and Uniform Resource Identifier standards, and from their subsequent deployment in Web client and server software.

As predicted by Perry and Wolf [105], software architecture has been a focal point for software engineering research in the 1990s. The complexity of modern software systems have necessitated a greater emphasis on componentized systems, where the implementation is partitioned into independent components that communicate to perform a desired task. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations. A good architecture is not created in a vacuum. All design decisions at the architectural level should be made within the context of the functional, behavioral, and social requirements of the system being designed, which is a principle that applies equally to both software architecture and the traditional field of building architecture. The guideline that “form follows function” comes from hundreds of years of experience with failed building projects, but is often ignored by software practitioners. The funny bit within the Monty Python sketch, cited above, is the absurd notion that an architect, when faced with the goal of designing an urban block of flats (apartments), would present a building design with all the components of a modern slaughterhouse. It might very well be the best slaughterhouse design ever conceived, but that would be of little comfort to the prospective tenants as they are whisked along hallways containing rotating knives. The hyperbole of The Architects Sketch may seem ridiculous, but consider how often we see software projects begin with adoption of the latest fad in architectural design, and only later discover whether or not the system requirements call for such an architecture. Design-by-buzzword is a common occurrence. At least some of this behavior within the software industry is due to a lack of understanding of why a given set of architectural constraints is useful. In other words, the reasoning behind good software architectures is not apparent to designers when those architectures are selected for reuse. This dissertation explores a junction on the frontiers of two research disciplines in computer science: software and networking. Software research has long been concerned with the categorization of software designs and the development of design methodologies, but has rarely been able to objectively evaluate the impact of various design choices on system behavior. Networking research, in contrast, is focused on the details of generic communication behavior between systems and improving the performance of particular communication techniques, often ignoring the fact that changing the interaction style of an application can have more impact on performance than the communication protocols used for that interaction. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. When given a name, a coordinated set of architectural constraints becomes an architectural style. The first three chapters of this dissertation define a framework for understanding software architecture via architectural styles, revealing how styles can be used to guide the architectural design of network-based application software. Common architectural styles are surveyed and classified according to the architectural properties they induce when applied to an architecture for network-based hypermedia. This classification is used to identify a set of architectural constraints that could be used to improve the architecture of the early World Wide Web. Architecting the Web requires an understanding of its requirements, as we shall discuss in Chapter 4. The Web is intended to be an Internet-scale distributed hypermedia system, which means considerably more than just geographical dispersion. The Internet is about interconnecting information networks across organizational boundaries. Suppliers of information services must be able to cope with the demands of anarchic scalability and the independent deployment of software components. Distributed hypermedia provides a uniform means of accessing services through the embedding of action controls within the presentation of information retrieved from remote sites. An architecture for the Web must therefore be designed with the context of communicating large-grain data objects across high-latency networks and multiple trust boundaries. Chapter 5 introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles.

Over the past six years, the REST architectural style has been used to guide the design and development of the architecture for the modern Web, as presented in Chapter 6. This work was done in conjunction with my authoring of the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI), the two specifications that define the generic interface used by all component interactions on the Web. Like most real-world systems, not all components of the deployed Web architecture obey every constraint present in its architectural design. REST has been used both as a means to define architectural improvements and to identify architectural mismatches. Mismatches occur when, due to ignorance or oversight, a software implementation is deployed that violates the architectural constraints. While mismatches cannot be avoided in general, it is possible to identify them before they become standardized. Several mismatches within the modern Web architecture are summarized in Chapter 6, along with analyses of why they arose and how they deviate from REST. In summary, this dissertation makes the following contributions to software research within the field of Information and Computer Science: • a framework for understanding software architecture through architectural styles, including a consistent set of terminology for describing software architecture; • a classification of architectural styles for network-based application software by the architectural properties they would induce when applied to the architecture for a distributed hypermedia system; • REST, a novel architectural style for distributed hypermedia systems; and, • application and evaluation of the REST architectural style in the design and deployment of the architecture for the modern World Wide Web.

In spite of the interest in software architecture as a field of research, there is little agreement among researchers as to what exactly should be included in the definition of architecture. In many cases, this has led to important aspects of architectural design being overlooked by past research. This chapter defines a self-consistent terminology for software architecture based on an examination of existing definitions within the literature and my own insight with respect to network-based application architectures. Each definition, highlighted within a box for ease of reference, is followed by a discussion of how it is derived from, or compares to, related research.

At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties [117]. A complex system will contain many levels of abstraction, each with its own architecture. An architecture represents an abstraction of system behavior at that level, such that architectural elements are delineated by the abstract interfaces they provide to other elements at that level [9]. Within each element may be found another architecture, defining the system of sub-elements that implement the behavior represented by the parent element’s abstract interface. This recursion of architectures continues down to the most basic system elements: those that cannot be decomposed into less abstract elements. In addition to levels of architecture, a software system will often have multiple operational phases, such as start-up, initialization, normal processing, re-initialization, and shutdown. Each operational phase has its own architecture. For example, a configuration file will be treated as a data element during the start-up phase, but won’t be considered an architectural element during normal processing, since at that point the information it contained will have already been distributed throughout the system. It may, in fact, have defined the normal processing architecture. An overall description of a system architecture must be capable of describing not only the operational behavior of the system’s architecture during each phase, but also the architecture of transitions between phases. Perry and Wolf [105] define processing elements as “transformers of data,” while Shaw et al. [118] describe components as “the locus of computation and state.” This is further clarified in Shaw and Clements [122]: “A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters.” This raises an important distinction between software architecture and what is typically referred to as software structure: the former is an abstraction of the run-time behavior of a software system, whereas the latter is a property of the static software source code. Although there are advantages to having the modular structure of the source code match the decomposition of behavior within a running system, there are also advantages to having independent software components be implemented using parts of the same code (e.g., shared libraries). We separate the view of software architecture from that of the source code in order to focus on the software’s run-time characteristics independent of a given component’s implementation. Therefore, architectural design and source code structural design, though closely related, are separate design activities. Unfortunately, some descriptions of software architecture fail to make this distinction (e.g., [9]).

A comprehensive examination of the scope and intellectual basis for software architecture can be found in Perry and Wolf [105]. They present a model that defines a software architecture as a set of architectural elements that have a particular form, explicated by a set of rationale. Architectural elements include processing, data, and connecting elements. Form is defined by the properties of the elements and the relationships among the elements — that is, the constraints on the elements. The rationale provides the underlying basis for the architecture by capturing the motivation for the choice of architectural style, the choice of elements, and the form. My definitions for software architecture are an elaborated version of those within the Perry and Wolf [105] model, except that I exclude rationale. Although rationale is an important aspect of software architecture research and of architectural description in particular, including it within the definition of software architecture would imply that design documentation is part of the run-time system. The presence or absence of rationale can influence the evolution of an architecture, but, once constituted, the architecture is independent of its reasons for being. Reflective systems [80] can use the characteristics of past performance to change future behavior, but in doing so they are replacing one lowerlevel architecture with another lower-level architecture, rather than encompassing rationale within those architectures. As an illustration, consider what happens to a building if its blueprints and design plans are burned. Does the building immediately collapse? No, since the properties by which the walls sustain the weight of the roof remain intact. An architecture has, by design, a set of properties that allow it to meet or exceed the system requirements. Ignorance of those properties may lead to later changes which violate the architecture, just as the replacement of a load-bearing wall with a large window frame may violate the structural stability of a building. Thus, instead of rationale, our definition of software architecture includes architectural properties. Rationale explicates those properties, and lack of rationale may result in gradual decay or degradation of the architecture over time, but the rationale itself is not part of the architecture. A key feature of the model in Perry and Wolf [105] is the distinction of the various element types. Processing elements are those that perform transformations on data, data elements are those that contain the information that is used and transformed, and connecting elements are the glue that holds the different pieces of the architecture together. I use the more prevalent terms of components and connectors to refer to processing and connecting elements, respectively. Garlan and Shaw [53] describe an architecture of a system as a collection of computational components together with a description of the interactions between these components—the connectors. This model is expanded upon in Shaw et al. [118]: The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. Further elaboration of this definition can be found in Shaw and Garlan [121]. What is surprising about the Shaw et al. [118] model is that, rather than defining the software’s architecture as existing within the software, it is defining a description of the software’s architecture as if that were the architecture. In the process, software architecture as a whole is reduced to what is commonly found in most informal architecture diagrams: boxes (components) and lines (connectors). Data elements, along with many of the dynamic aspects of real software architectures, are ignored. Such a model is incapable of adequately describing network-based software architectures, since the nature, location, and movement of data elements within the system is often the single most significant determinant of system behavior.

Components are the most easily recognized aspect of software architecture. Perry and Wolf’s [105] processing elements are defined as those components that supply the transformation on the data elements. Garlan and Shaw [53] describe components simply as the elements that perform computation. Our definition attempts to be more precise in making the distinction between components and the software within connectors. A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface. Example transformations include loading into memory from secondary storage, performing some calculation, translating to a different format, encapsulation with other data, etc. The behavior of each component is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another component [9]. In other words, a component is defined by its interface and the services it provides to other components, rather than by its implementation behind the interface. Parnas [101] would define this as the set of assumptions that other architectural elements can make about the component.

Perry and Wolf [105] describe connecting elements vaguely as the glue that holds the various pieces of the architecture together. A more precise definition is provided by Shaw and Clements [122]: A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. Examples include shared representations, remote procedure calls, message-passing protocols, and data streams. Perhaps the best way to think about connectors is to contrast them with components. Connectors enable communication between components by transferring data elements from one interface to another without changing the data. Internally, a connector may consist of a subsystem of components that transform the data for transfer, perform the transfer, and then reverse the transformation for delivery. However, the external behavioral abstraction captured by the architecture ignores those details. In contrast, a component may, but not always will, transform data from the external perspective.

As noted above, the presence of data elements is the most significant distinction between the model of software architecture defined by Perry and Wolf [105] and the model used by much of the research labelled software architecture [1, 5, 9, 53, 56, 117-122, 128]. Boasson [24] criticizes current software architecture research for its emphasis on component structures and architecture development tools, suggesting that more focus should be placed on data-centric architectural modeling. Similar comments are made by Jackson [67]. A datum is an element of information that is transferred from a component, or received by a component, via a connector. Examples include byte-sequences, messages, marshalled parameters, and serialized objects, but do not include information that is permanently resident or hidden within a component. From the architectural perspective, a “file” is a transformation that a file system component might make from a “file name” datum received on its interface to a sequence of bytes recorded within an internally hidden storage system. Components can also generate data, as in the case of a software encapsulation of a clock or sensor. The nature of the data elements within a network-based application architecture will often determine whether or not a given architectural style is appropriate. This is particularly evident in the comparison of mobile code design paradigms [50], where the choice must be made between interacting with a component directly or transforming the component into a data element, transferring it across a network, and then transforming it back to a component that can be interacted with locally. It is impossible to evaluate such an architecture without considering data elements at the architectural level.

Abowd et al. [1] define architectural description as supporting the description of systems in terms of three basic syntactic classes: components, which are the locus of computation; connectors, which define the interactions between components; and configurations, which are collections of interacting components and connectors. Various style-specific concrete notations may be used to represent these visually, facilitate the description of legal computations and interactions, and constrain the set of desirable systems. Strictly speaking, one might think of a configuration as being equivalent to a set of specific constraints on component interaction. For example, Perry and Wolf [105] include topology in their definition of architectural form relationships. However, separating the active topology from more general constraints allows an architect to more easily distinguish the active configuration from the potential domain of all legitimate configurations. Additional rationale for distinguishing configurations within architectural description languages is presented in Medvidovic and Taylor [86].

The set of architectural properties of a software architecture includes all properties that derive from the selection and arrangement of components, connectors, and data within the system. Examples include both the functional properties achieved by the system and non functional properties, such as relative ease of evolution, reusability of components, efficiency, and dynamic extensibility, often referred to as quality attributes [9]. Properties are induced by the set of constraints within an architecture. Constraints are often motivated by the application of a software engineering principle [58] to an aspect of the architectural elements. For example, the uniform pipe-and-filter style obtains the qualities of reusability of components and configurability of the application by applying generality to its component interfaces — constraining the components to a single interface type. Hence, the architectural constraint is “uniform component interface,” motivated by the generality principle, in order to obtain two desirable qualities that will become the architectural properties of reusable and configurable components when that style is instantiated within an architecture. The goal of architectural design is to create an architecture with a set of architectural properties that form a superset of the system requirements. The relative importance of the various architectural properties depends on the nature of the intended system. Section 2.3 examines the properties that are of particular interest to network-based application architectures.