

浙江大学

本科实验报告

课程名称: B/S 体系软件设计

姓 名: 葛浩

学 院: 计算机科学与技术学院

系: 计算机科学与技术系

专 业: 计算机科学与技术

学 号: 3180103494

指导教师: 胡晓军

2021 年 6 月 27 日

浙江大学实验报告

课程名称: B/S 体系软件设计 实验类型: 软件设计

实验项目名称: IoT 系统网站

学生姓名: 葛浩 专业: 计算机科学与技术 学号: 3180103494

同组学生姓名: 无 指导老师: 胡晓军

实验地点: 浙江大学玉泉校区 实验日期: 2021 年 6 月 27 日

项目配置指导

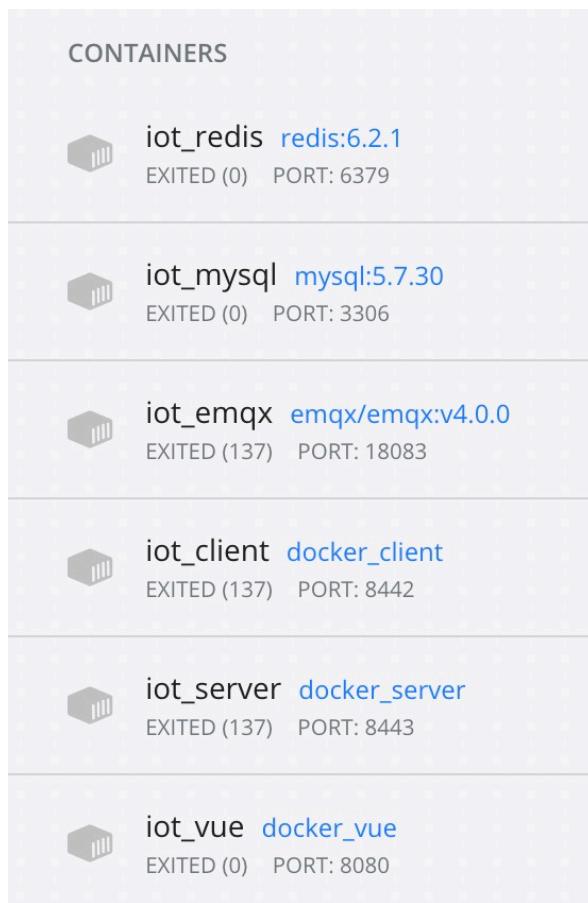
可以使用docker启动，也可以自行安装依赖软件后本地启动

使用docker启动

背景介绍

本项目前后端分离，前端主要由vue开发完成，后端主要由springboot+redis+mysql+emqx开发完成，为了方便任何一个人、任何一台机器都能够直接部署本项目，我耗费了近4天的时间，几乎踩完了docker、dockerfile、docker-compose所有的坑，最终将所有独立的服务配置成docker镜像，而使用docker-compose工具对这些docker容器进行编排，从而使得只用一条命令就能启动多个容器，全自动化完成环境部署

这里配置的docker镜像和对应的容器列举如下：



快速启动

想要启动前后端服务，你只需要 cd 进入 docker/ 目录，并输入下面这行命令：

```
docker-compose up
```

随后，所有docker镜像对应的容器都会自行启动，命令行会输出它们启动时的Log信息，大约1-2分钟之后(第一次启动可能需要更长时间)，就可以访问 <http://localhost:8080> 来登录网站了

注意事项

有几个注意点：

- 当你第一次运行上面这行命令时，由于需要在本地创建镜像，因此会耗费更多的时间，请耐心等待
- 本项目的docker映射了本机的几个端口号，启动docker之前请务必保证这些端口号不被占用(需要关闭本地运行的mysql、redis、emqx等服务，否则它们会和docker抢占端口)

```
- 8080 # 前端访问端口号
- 8443 # 后端访问端口号
- 8442 # 老师提供的iotclient的端口号
- 3306 # mysql的端口号
- 6379 # redis的端口号
- 1883 # 以下都是emqx的端口号
- 18083
- 8883
- 8083
- 8084
```

- 如果是macos系统，建议不要直接通过`brew install docker`安装docker，因为docker是系统级包，使用`brew install`安装会缺少必要的权限，详情可[参考](#)，建议的安装命令：

```
brew install --cask docker
```

直接本地启动

安装依赖软件

想要本地启动，需要本地先安装以下软件：

- emqx:v4.0.0
- redis:6.2.1
- mysql:5.7.30
- java:8
- npm:7.11.2
- node:15.14.0
- vue:2.9.6

配置并启动依赖软件

启动emqx：

```
emqx start
```

启动redis：

```
redis-server
```

启动mysql：

```
sudo /usr/local/mysql/support-files/mysql.server start
```

初始化数据库：

```
mysql -u root -p  
> source init.sql
```

启动服务端：

建议直接利用idea打开并启动iotserver工程(springboot)

启动客户端：

建议直接利用idea打开并启动iotclient工程(springboot)

配置vue：先命令行进入iot-vue目录下，执行

```
npm install
```

启动vue：同样命令行进入iot-vue目录下，执行

```
npm run dev
```

访问网站

在执行完`npm run dev`后，命令行会给出网站的预览地址，一般为`http://localhost:8080`，在浏览器中打开即可

BS: IoT设计文档

1. 背景

1.1 物联网概念

1.2 项目概述

2. 技术选型

2.1 IoT设备

2.2 数据传输协议

2.2.1 MQTT介绍

2.2.2 MQTT v.s. HTTP

2.2.3 本项目的协议架构

2.3 web后端技术

2.3.1 主流后端技术介绍

2.3.2 本项目的后端技术栈

2.4 web前端技术

2.4.1 主流前端技术介绍

2.4.2 本项目的前端技术栈

2.5 技术栈小结

3. 项目架构设计

3.1 架构设计抽象

3.2 架构设计具体实践

3.3 功能模块设计与解耦

4. 项目实现与安排

BS: IoT设计文档

任选Web开发技术实现一个物联网应用的网站

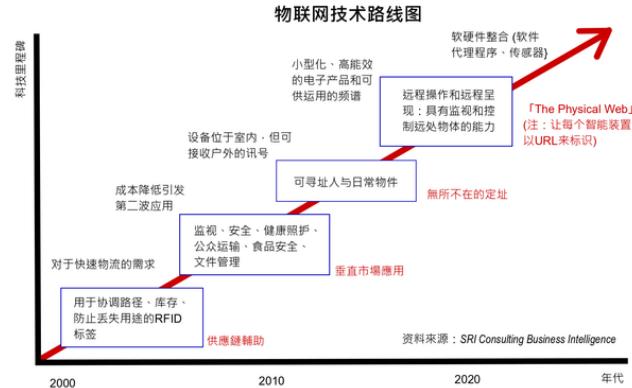
葛浩 3180103494 计算机科学与技术

1. 背景

1.1 物联网概念

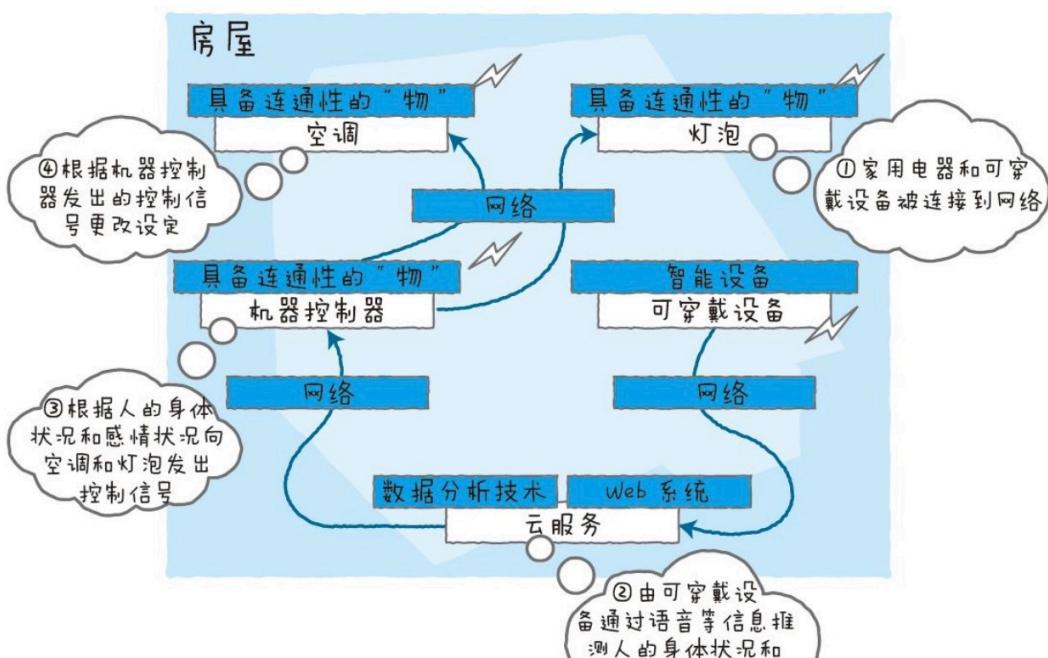
"物联网"一词的英文是Internet of Things, 即IoT, 其概念最早可追溯到1980年代初期, 全球第一台隐含物联网概念的设备为位于卡内基·梅隆大学的可乐贩卖机, 它连接到互联网, 可以在网络上检查库存, 以确认还可供应的饮料数

实际上, 物联网本身的定义上还是集中于Internet, 是一种物与物之间的连接方式, 就目前物联网的发展趋势而言, 它已经不是一个单纯的Internet, 而进一步成为一种抽象意义的系统, 或者说生态, 我们目前更多的不是在关注其网络的连接, 而是利用物联网构成了一种什么样的系统



往小的说，传统的internet是关于人的连接网络，主要是人与人之间传递信息，而发展到物联网之后，就不仅仅是人与人之间的连接了，更多的是人与物，物与物之间的连接关系，形形色色的“物”都可以通过internet相连

物联网本身是从连接的方式来定义的，但是目前随着物联网的发展，物联网更多的是与智能结合在一起，物联网目前应该是包含了传感器，网络连接，终端设备三个角色的具有智能的反馈型网络，下图给出了一个包含上述定义的物联网系统组成



1.2 项目概述

在上述IoT的背景下，本项目的基本目标是利用Web开发技术实现一个物联网应用的网站

做项目其实跟搭积木差不多，想要完成一个项目，首先就要分析整个流程，并选择好使用哪些积木，这些积木也就是我们所要采取的技术，而每样技术又有不同的实现方式，即技术选型(这块内容将在后面阐述)，有了积木之后，接下来的任务就是设计好整个框架，用积木去搭建填充即可

上述步骤中，除了搭建木这个具体工程之外，剩余内容都属于项目设计的部分，也就是本设计文档要重点描述的内容

因此这里我先简单分析一下整个IoT项目运作的基本流程，再进一步地阐述完成这样一个项目设计需要用到哪些技术，具体的技术选型在后面的章节阐述

IoT项目的基本流程如下：

- IoT设备通过网络将数据发送到我们的服务器

- 服务器在内存里接受数据，并将其写入数据库
- 服务器对数据进行处理，并将处理的结果返回前端显示
- 服务器还需要提供用户权限校验和注册、登录等功能

为了完成上述流程，我们需要用到的设备和技术如下：

- IoT设备和传感器
- IoT数据传输的网络协议
- 服务器
- web后端技术
- 缓存与数据库技术
- web前端技术

本设计文档主要将围绕上述技术的技术选型展开，并以web前后端技术架构为主体

2. 技术选型

2.1 IoT设备

由于本项目侧重于开发物联网应用的网站，而并非侧重于物理设备的开发，因此这里使用老师提供的iotclient工程来模拟终端发送数据

iotclient的实现原理也很简单，利用多个线程来模拟多台不同的设备device，并让它们定时地向服务器的 `tcp://ip:1883` 端口发送数据，数据的生成也很简单，根据杭州经纬度随机生成设备位置信息即可

```
Vector<WorkerThread> threadVector = new Vector<WorkerThread>();
for (int i = 0; i < devices; i++) {
    WorkerThread thread = new WorkerThread();
    thread.setDeviceId(i + 1);
    thread.setMqttServer(mqttServer);
    thread.setTopic(topic);
    thread.setClientPrefix(clientPrefix);
    threadVector.add(thread);
    thread.start();
}
```

2.2 数据传输协议

2.2.1 MQTT介绍

IoT设备和服务器必然不在同一个物理设备里，而是处于网络上的两个节点，那么它们之间就需要用特定的协议来进行通信和收发数据

通常对于B/S场景，我们会用HTTP协议来进行浏览器和服务器之间的数据交互，而对于IoT设备和服务器之间的数据交互，这里采用的是MQTT协议

什么是MQTT呢，这里基于我个人的理解给出一个简要的概括：MQTT是一个极其轻量级的发布/订阅消息传输协议，适用于网络带宽较低的场合，它通过一个代理服务器(broker)，任何一个客户端(client)都可以订阅或者发布某个主题的消息，所有订阅了该主题的客户端都会收到该消息

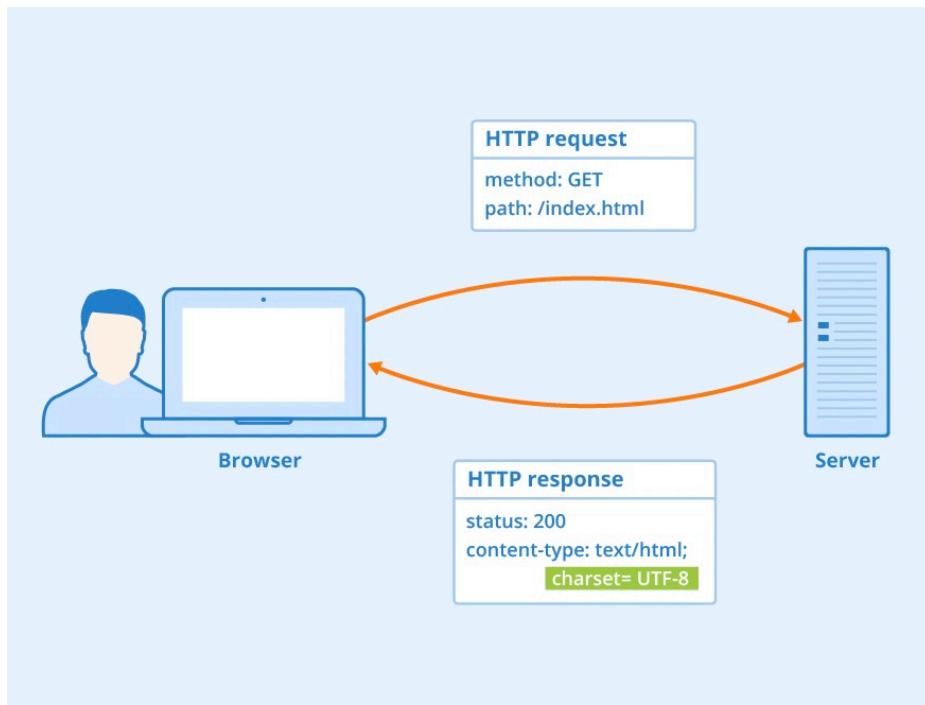
更详细地说，MQTT是基于二进制消息的发布/订阅编程模式的消息协议，最早由IBM提出的，如今已经成为OASIS规范，由于规范很简单，非常适合需要低功耗和网络带宽有限的IoT场景，其基本设计原则如下：

- 精简，不添加可有可无的功能
- 发布/订阅（Pub/Sub）模式，方便消息在传感器之间传递
- 允许用户动态创建主题，零运维成本
- 把传输量降到最低以提高传输效率
- 把低带宽、高延迟、不稳定的网络等因素考虑在内
- 支持连续的会话控制
- 理解客户端计算能力可能很低
- 提供服务质量管理
- 假设数据不可知，不强求传输数据的类型与格式，保持灵活性

2.2.2 MQTT v.s. HTTP

传统的B/S架构通常采用HTTP协议作为web传输数据的机制，那为什么不用让IoT设备以HTTP请求的形式发送数据，以HTTP响应的方式接收数据呢？实际上，HTTP是一种很重的协议，它包含许多标头和规则，并不适合受限的网络，且作为1对1的协议，服务器想要将消息传送到网络的所有设备上，不仅困难，成本也很高

此外，HTTP是一种同步协议，客户端需要等待服务器响应，Web浏览器具有这样的要求，它的代价是牺牲了可伸缩性；在IoT领域，大量设备由于不可靠或高延迟的网络使得同步通信成为问题，此时异步消息协议就会更适合IoT应用程序，即传感器发送读数，让网络确定将其传送到目标设备和服务的最佳路线和时间



MQTT协议之所以如此轻量且灵活，离不开它的一个关键特性——发布和订阅模型，与所有消息协议一样，它将数据的发布者与使用者分离

MQTT协议在网络中定义了两种实体类型：一个消息代理和一些客户端：

- 代理是一个服务器，它从客户端接收所有消息，然后将这些消息路由到相关的目标客户端
- 客户端是能够与代理交互来发送和接收消息的任何事物，客户端可以是现场的IoT传感器，或者是数据中心内处理IoT数据的应用程序，在本项目是就是后端的web程序

首选客户端连接到代理，它可以订阅代理中的任何消息“主题”，此连接可以是简单的TCP/IP连接，也可以是用于发送敏感消息的加密TLS连接；然后客户端通过将消息和主题发送给代理，发布某个主题范围内的消息；最后代理将消息转发给所有订阅该主题的客户端

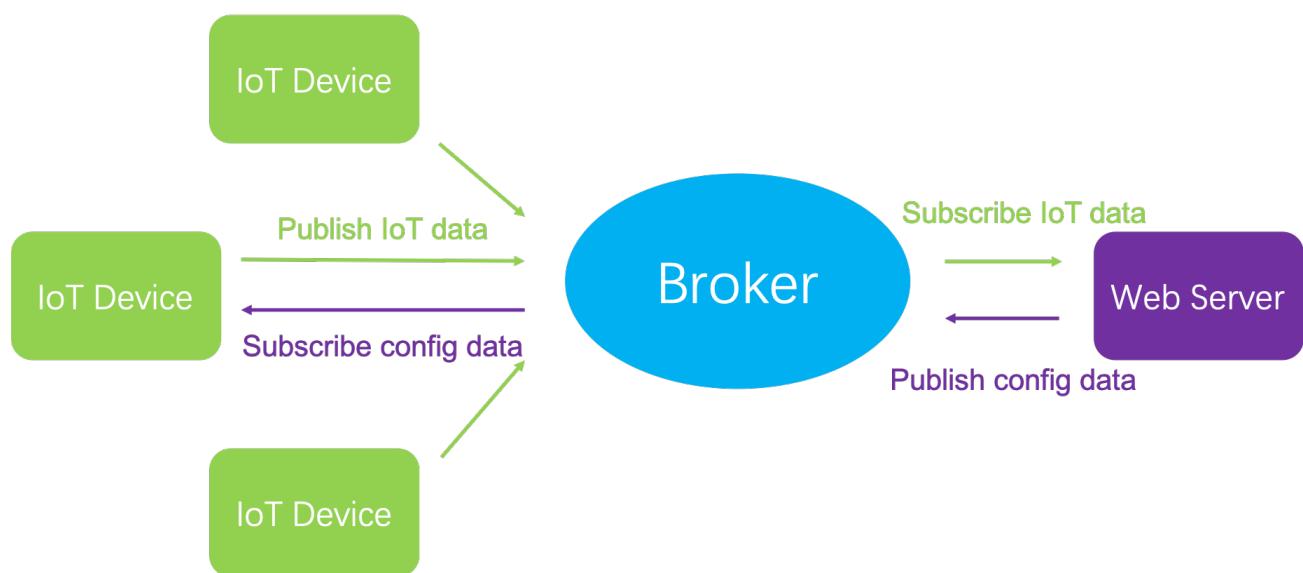
2.2.3 本项目的协议架构

为了将MQTT协议运用到本项目中，我按照自己的理解制作了下述的流程图

由于MQTT消息是按主题进行组织的，而根据文档给出的实验要求，我们在协议层面要完成两件事情：接收IoT设备发送的数据包+向IoT设备发送更改配置信息的数据包

因此，需要有两个主题：“IoT”主题和“config”主题

- 对传感器来说：
 - 将实时采集到的数据在“IoT”主题范围内发布
 - 订阅web服务端发布的“config”主题，根据后端的要求创建或修改设备信息，如设备ID、设备名称等
- 对web服务端来说：
 - 接收系统管理员的命令来调整设备的配置，并在“config”主题范围内发布
 - 订阅“IoT”主题范围内的数据，并将其保存到后端数据库、处理后发送给前端显示



2.3 web后端技术

2.3.1 主流后端技术介绍

目前的web后端技术有很多，主流的包括JAVAEE、SSM、Spring Boot等，通常业界都会使用SSM或Spring Boot来搭建自己的后端系统，比如我目前在实习的阿里集团就是用Spring Boot的升级版Pandora Boot来开发，其中包含了分布式RPC框架HSF、分布式非持久化配置中心ConfigServer、分布式缓存Tair、分库分表TDDL等等

当然，这些中间件都是作为微服务提供给各个中台、业务来开发使用的，涉及到成千上万台机器的集群，对于本项目来说，只有一台服务器提供web服务，因此无需用到上述的大部分组件

2.3.2 本项目的后端技术栈

这里我选用的后端技术栈是Spring Boot+Maven+Redis+Mybatis+MySQL的配置，接下来做逐一介绍

Springboot：是Spring家族中的一个全新的框架，它是用来简单应用程序的创建和开发过程，化繁为简，简化SSM框架的配置；比如在使用SSM框架开发的时候，我们需要配置web.xml、配置spring、配置mybatis，再将它们整合到一起，而是用Springboot就不同了，它采用了大量的默认配置来简化这些文件的配置过程

Maven：Maven是Apache软件基金会唯一维护的一款自动化构建工具，专注于服务Java平台的项目构建和依赖管理，它基于项目对象模型(POM)，可以通过一小段描述信息来管理项目的构建、报告和文档

Redis: Redis是C语言开发的一个开源的高性能键值对(key-value)的内存数据库，可以用作数据库、缓存、消息中间件等，再本项目中用于作为MySQL数据库的缓存

Mybatis: MyBatis是一个持久层(DAO)框架，本质上是JDBC的一次封装，主要作用是方便我们进行数据库的增删改查CRUD操作，它使业务代码与JDBC相解耦

MySQL: 互联网行业中最流行的数据库

2.4 web前端技术

2.4.1 主流前端技术介绍

目前主流的前端技术应该就是React和Vue框架了，由于我对前端技术不是那么了解，这里仅做简单介绍，本项目中前端的主要目的就是能让数据可视化出来，而不过分追求美观

下面给出维基百科对两种框架的介绍：

React: 是一个为数据提供渲染为HTML视图的开源JavaScript库，React视图通常采用包含以自定义HTML标记规定的其他组件的组件渲染，React为程序员提供了一种子组件不能直接影响外层组件的模型，数据改变时对HTML文档的有效更新，和现代单页应用中组件之间干净的分离

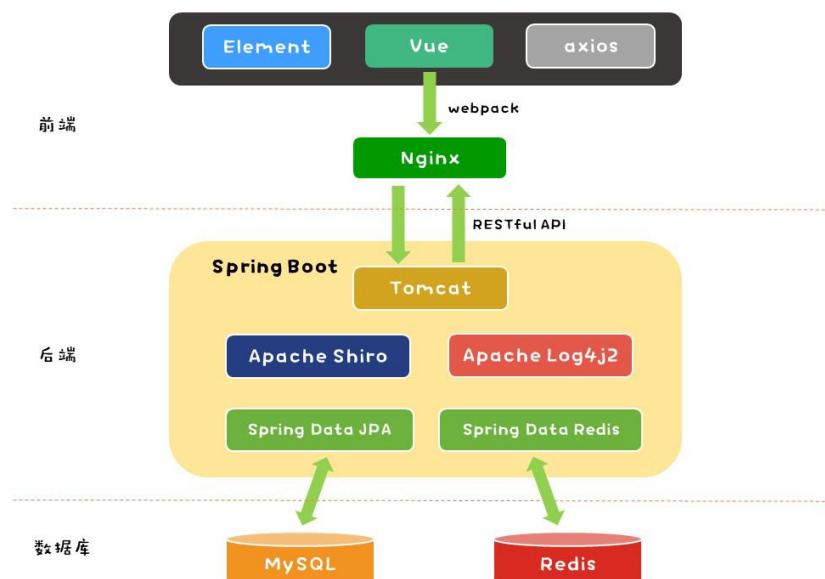
Vue.js: Vue.js是一个用于创建用户界面的开源JavaScript框架，也是一个创建单页应用的Web应用框架；它是一套用于构建用户界面的渐进式框架，与其它大型框架不同的是，Vue被设计为可以自底向上逐层应用，Vue的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合，另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue也完全能够为复杂的单页应用提供驱动

2.4.2 本项目的前端技术栈

实际上，由于Vue相比React更为简单上手，因此我选择了Vue+ElementUI+Axios作为前端的技术栈

2.5 技术栈小结

本项目使用的技术栈小结如下：后端技术栈采用Spring Boot+Maven+Redis+Mybatis+MySQL，前端技术栈采用Vue+ElementUI+Axios



3. 项目架构设计

前文已经介绍了本项目使用的技术栈，接下来我将介绍一下本项目整体架构上的设计

3.1 架构设计抽象

对于前端工程，基本架构已经由Vue本身定义好，这里不再阐述

对于后端工程，这里采用MVC设计模式，并将整体架构的设计抽象如下：

- 终端显示层：即前端渲染显示部分
- 开发API层：后端提供的服务封装成HTTP接口，开放给前端调用
- 请求处理层(controller层)：主要包含Filter过滤和Intercept拦截功能，以及web请求的转发
- service层：主要负责业务逻辑的处理，在service层之上还会有biz层和manager层，它们是对业务逻辑的进一步封装
- DAO层：数据访问层，主要通过mybatis与底层的mysql数据库进行交互
- 数据存储层：即mysql数据库

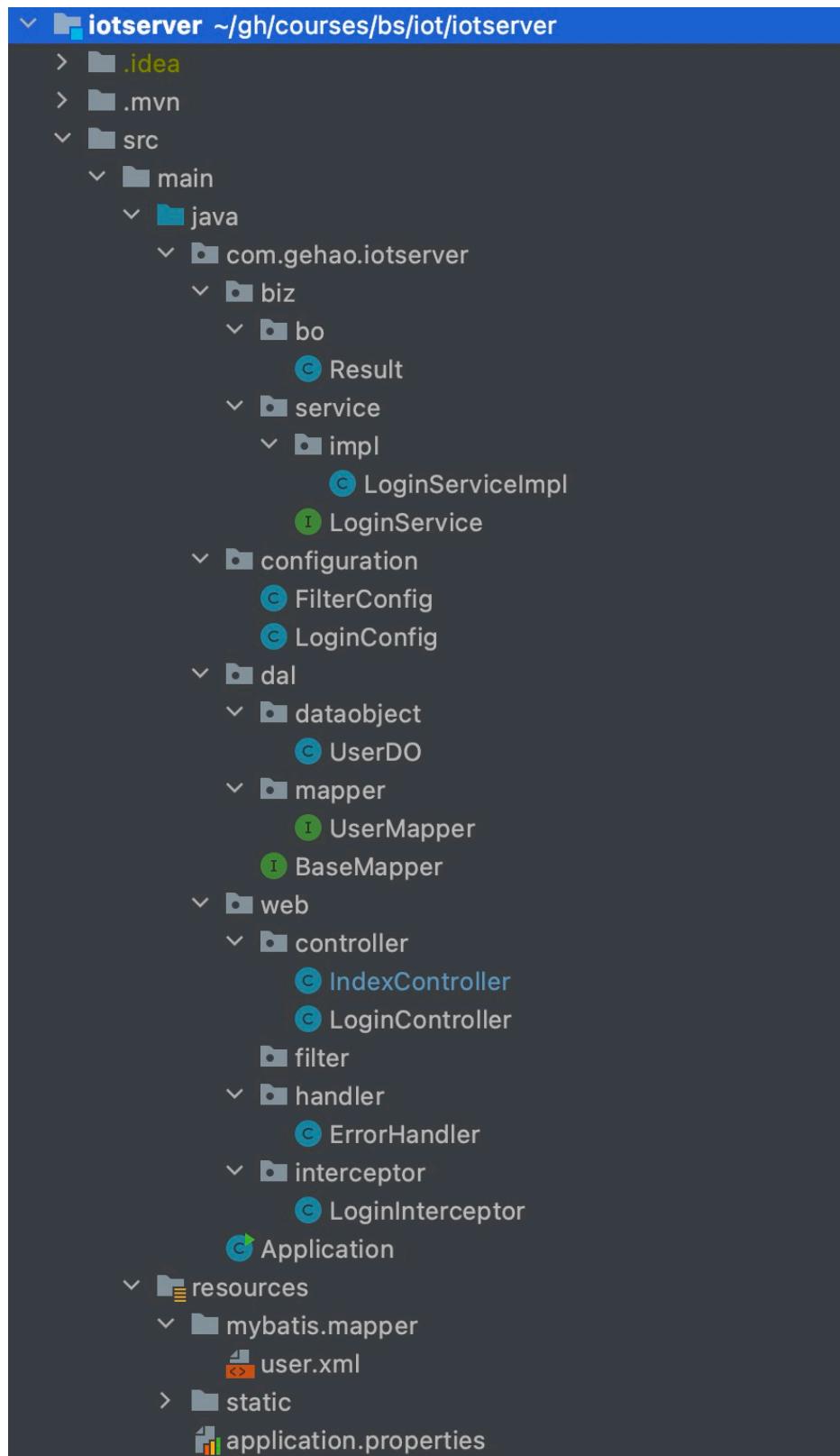
3.2 架构设计具体实践

根据上述架构抽象思想的原则，我将iotserver的架构具体设计如下：

首先根目录是 `com.gehao.iotserver`，在根目录下主要分为以下几个一级子目录：

- biz层：这一层对应着业务逻辑的实现，包括业务对象BO和业务代码service
 - bo层：业务对象的定义，它封装了某个业务相关的所有属性和基本操作
 - service层：业务逻辑代码的实现，包括interface接口以及impl目录下的具体实现
- configuration层：存放的是自定义的 `@configuration` 注解，主要用于将自定义的filter或interceptor作为JAVA Bean进入到Spring Boot的生命周期里
- dal层：存放的是数据访问相关的代码
 - dataobject层：数据对象DO的定义
 - mapper层：mybatis函数，它会和resources/mybatis/mapper下的xml文件一一对应
- web层：存放的是web处理相关的函数
 - filter、interceptor层：对过滤器、拦截器的定义
 - handler层：对一些错误error的处理
 - controller层：针对web访问请求的处理，它要做的仅仅是把对应的请求下发给具体的service处理，并将处理结果返回给web前端

注：由于本工程在刚开始搭建，因此仅实现了用户登录功能，但已经完成了项目架构上从0到1的实现，接下来的代码只需要在这个架构下添加补充内容即可



3.3 功能模块设计与解耦

这里主要对将要实现的几个功能模块进行解耦和设计，具体的实现逻辑还是需要等写代码的时候落实，过早设计反而会起到反作用

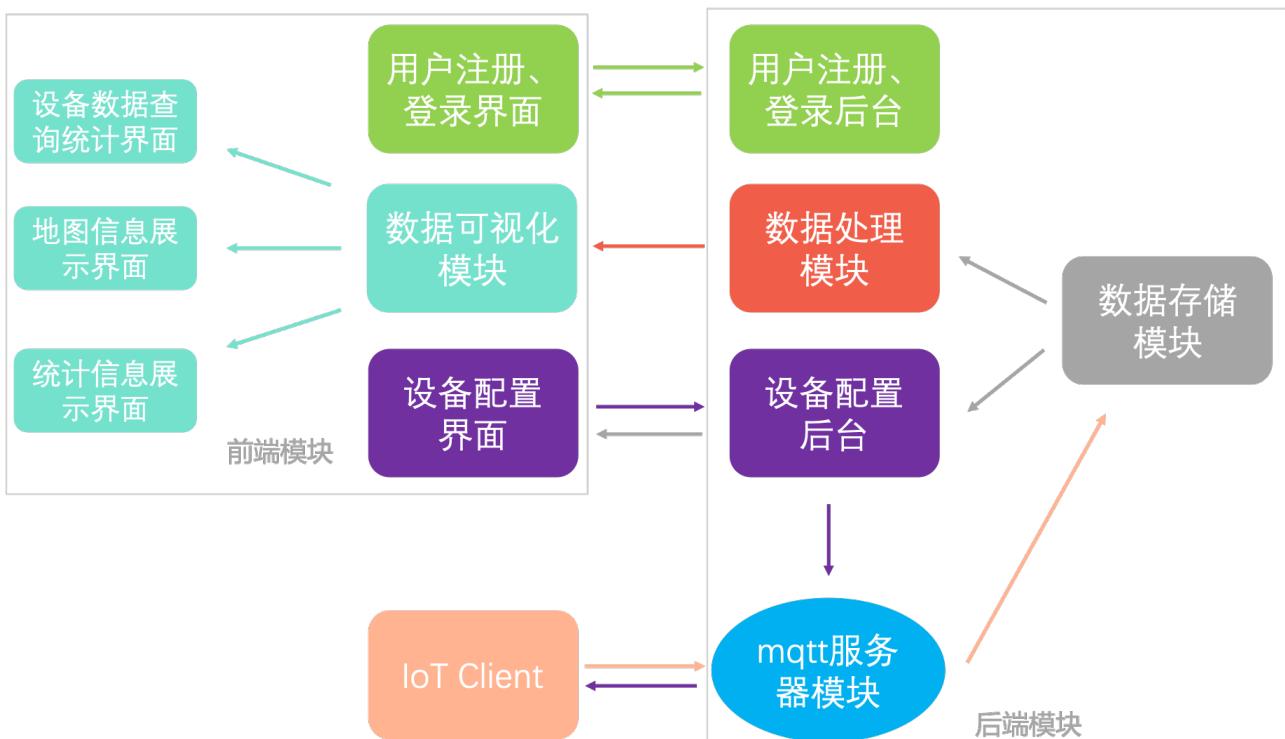
在实验需求文档里提出了一些具体的功能要求：

- 搭建一个mqtt服务器，能够接收指定的物联网终端模拟器发送的数据
- 实现用户注册、登录功能，用户注册时需要填写必要的信息并验证，如用户名、密码要求在6字节以上，email的格式验证，并保证用户名和email在系统中唯一，用户登录后可以进行以下操作

- 提供设备配置界面，可以创建或修改设备信息，包含必要信息，如设备ID、设备名称等
- 提供设备上报数据的查询统计界面
- 提供地图界面展示设备信息，区分正常和告警信息，并可以展示历史轨迹
- 首页提供统计信息（设备总量、在线总量、接收的数据量等），以图表方式展示（柱状体、折线图等）

根据上述需求，我将功能解耦成下述几个模块，下图给出了我对整个系统的整体设计，其中同颜色的箭头表示相同数据源：

- **数据存储模块**：主要存储mqtt服务器模块传过来的数据，并交给数据处理模块和设备配置模块使用
- **mqtt服务器模块**：这一模块的思路已经在前面分析过了，只需要在服务器上跑一个broker代理，接下来相当于是在web服务器上再运行一个mqtt客户端，这个客户端订阅iotclient发送的数据，并将数据存储到数据存储模块中，此外它还接收用户给出的设备配置指令，并将修改信息发布给iot client
- **数据处理模块**：主要对数据存储模块中保持的iotclient数据进行统计和处理，并将处理结果返回给前端可视化模块使用
- **设备配置模块**：后台会从数据存储模块中取出iotclient发布的数据，并交给前端界面进行显示，同时也接收用户在前端给出的设备配置修改指令并转发给mqtt服务器模块
- **用户注册、登录功能模块**：这一模块其实我已经实现了，在过滤器/拦截器上设计相应的验证代码即可，其中还需要对数据库的user表进行查询
- **前端可视化模块**：主要负责显示渲染三个界面：
 - 设备数据查询统计界面
 - 地图信息展示界面
 - 统计信息展示界面



4. 项目实现与安排

目前项目已经完成了从0到1的搭建过程，并完成了用户注册登录模块(包括前端和后端)，接下来将争取在一个月的时间内完成整个工程的任务，后续的安排如下：

- 先完成mqtt服务器模块，然后进一步完成数据存储模块，此时能够从iotclient中拿到数据并存入数据库中
- 然后再完成数据处理模块，这一模块与前面的强关联性不大，只需要从数据库读取数据并进行加工处理即可
- 然后开始搭建前端工程，先搭建数据可视化模块的三个界面，使得数据处理模块的结果能够渲染显示
- 最后再搭建设备配置模块，这一模块需要和mqtt服务器模块联合调试，并且需要修改iotclient的部分代码，使得设备修改能够在具体的IoT device(这里是模拟的线程)上真正生效

开发及使用手册

设计文档是前期对整体开发方向的把握，在具体实践过程中必然会有新的调整，本文档主要就是介绍具体开发过程和期间所选用的技术栈，顺便讲讲开发(自学)过程中遇到的问题和体会

前后端技术栈

技术栈概览

本项目采用的是前后端分离的设计方案，在大方向上以vue和springboot为主体：

- 前端：vue+vuex+ehcharts+element ui
- 后端：springboot+redis+mybatis+mysql+emqx
- 部署：docker+dockerfile+docker-compose

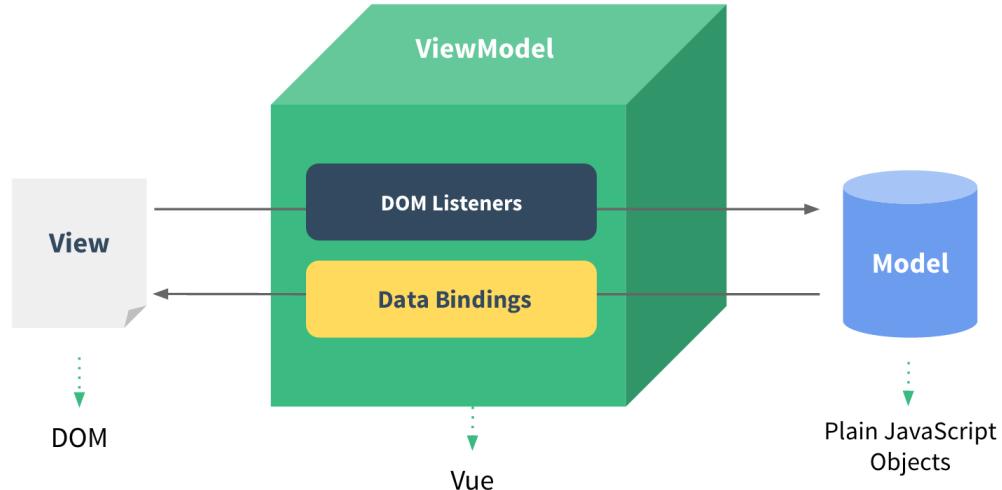
前端技术栈

[Vue](#)是当前非常流行的JS **MVVM**库，其核心思想是**数据驱动+组件化**

所谓的MVVM就是Model-View-ViewModel，其中Model对应着数据，View则对应着DOM视图元素，它们之间通过ViewModel进行绑定

此外，Vue是单页面应用，每一个页面元素都可以看做是component，组件可以单独作为一个页面，也可以作为页面的一部分，编写Vue代码本质上就是编写一个个不同的component

```
<template>视图元素(组件)</template>
<script>数据操作</script>
<style>样式</style>
```



[Echarts](#)是一款基于JavaScript的数据可视化图表库，由百度团队开源，提供非常丰富的图表可视化工具，包括柱状图、折线图、饼图等，此外，它还和百度地图api兼容，虽然官方文档没有详细的说明，但还是提供了在地图上进行数据点和路径绘制的可行途径

[Element UI](#)则是一套基于Vue的桌面端组件库，它封装了按钮、单选框、多选框、表格、表单、抽屉、导航等基本组件，可以方便快速地搭建出一个界面，当然，缺点是组件不够丰富、定制化程度低，样式简陋，如果要写出满意的页面，需要额外花费许多经历去调整样式和结构

后端技术栈

后端所采用的技术相对来说比较熟悉，内嵌Tomcat的SpringBoot作为服务器主体，Redis用于配置缓存，MySQL用于存储数据，MyBatis提供以面向对象的方式访问数据库，而Emqx则用于对mqtt服务的监听和转发，具体不再做过多说明

环境部署

可以看到，前面这些工具，如果要在一台新的电脑上部署，尽管springboot相关的依赖都可以打包成jar包，但nginx、java、redis、mysql、emqx这几项独立服务都必须要进行重新下载和配置

每次部署都要重新配置环境无疑是一件很繁琐的事情，并且在不同操作系统(windows、macos、linux)下还会遇到各类奇怪的问题，因此我选择使用docker的方式，将以上这几项服务各自打包成镜像，并使用docker-compose工具对这些镜像进行编排和统一管理，只需要一个命令就可以自动化启动各服务对应的容器，并且保证容器之间的网络通信

前端开发

背景介绍

前端开发对我来说是工作量最大的一个环境，因为之前没有接触过相关技术，几乎要从头开始学起，幸运的是有Vue这样易于入门的前端框架，即使对html、css、javascript只是一知半解，也能够借助组件化+数据驱动的基本思想快速搭建出一套UI界面出来

根据要求，前端可以解耦成以下几个基本模块：

- 登录注册模块
- IoT数据可视化界面
- 设备数据查询界面
- 设备信息配置界面

登录注册模块

登录界面如下：如果已有账号，则直接登录即可，否则点击链接进行注册



注册模块如下所示：用户注册主要由用户名、密码、确认密码、邮箱这四部分组成



得益于vue的数据绑定特性，我们可以实时获取用户在输入框填写的内容，并根据既定规则进行合法性的提示：

- 用户名长度必须大于3，且在数据库中唯一
- 密码长度必须大于6
- 两次输入的密码必须匹配
- 邮箱必须满足特定格式(正则匹配)，且在数据库中唯一



IoT数据可视化界面

这一模块是整个系统的重点所在，服务端会接收来自iot客户端的数据，并存储于数据库中，而前端需要定期向服务端请求数据，并实时可视化显示一些统计信息

这里我利用Echarts工具实现了5个可视化界面，分别是：

- 地图界面：借助百度地图api，默认实时显示iot设备的位置，正常信息用绿色标注，报警信息用红色标注；如果想要展示某个设备的历史轨迹，可以直接鼠标点击地图上的某个iot设备，界面就会自动切换到轨迹
- 各设备接收的数据量：用于分别统计各个iot设备目前接收的数据量
- 接收的数据总量：用于实时统计当前服务端总共接收的数据量
- 设备在线数量：用于实时统计当前时刻在线的设备数量
- 各设备最新的value值：用于实时统计各设备最新的value值
- 各设备value值变化趋势：用于统计历史一段时间内各设备value值的变化趋势



注：如果点击地图上的某个设备点，则地图会切换为显示该设备历史一段时间内的轨迹，用数字标记时间点，同样也是用绿色和红色区分该时刻的设备状态



此外，点击每个图表右上方的按钮，可以将其放大至全屏，再次点击恢复原状

设备数据查询界面

设备数据查询界面，默认分页展示所有历史数据

- 点击下方的分页栏可以跳转至指定页数
- 可以用关键字筛选数据
- 点击各列标题的上箭头或下箭头可以实现根据这一列属性值对数据进行正序或倒序排序

设备数据查询

设备ID	上报信息	设备数据	告警状态	经度	维度	上报时间
1 device0003	Device Data 2021/06/27 00:23:11	13	0	120.109321343 89878	30.2128337144 8517	1624724591873
2 device0003	Device Data 2021/06/27 00:23:10	85	1	120.131922852 99301	30.427857470 512393	1624724590868
3 device0002	Device Data 2021/06/27 00:23:08	98	1	119.998839402 1988	30.2512040615 0818	1624724588649
4 device0005	Device Data 2021/06/27 00:23:07	92	1	120.26258555 650712	30.495320677 757263	1624724587689
5 device0004	Device Data 2021/06/27 00:23:06	89	1	119.978261852 26441	30.2120512962 34133	1624724586649
6 device0002	Device Data 2021/06/27 00:23:06	62	0	120.427581751 3466	30.3187129735 94668	1624724586643

共 848 条 10条/页 < 1 2 3 4 5 6 … 85 > 前往 1 页

设备信息配置界面

设备配置界面主要是对设备信息的配置：

- 设备名称默认为设备ID
- 点击编辑按钮，可以对设备ID或设备名称进行修改，点击保存后生效
- 点击删除按钮，可以删除该行设备信息
- 点击新增按钮，可以增加一行设备信息

设备配置界面

设备ID	设备名称	操作		
1 device0001	device0001	<button>编辑</button>	<button>删除</button>	<button>保存</button>
2 device0002	device0002	<button>编辑</button>	<button>删除</button>	<button>保存</button>
3 device0003	device0003	<button>编辑</button>	<button>删除</button>	<button>保存</button>
4 device0004	device0004	<button>编辑</button>	<button>删除</button>	<button>保存</button>
5 device0005	device0005	<button>编辑</button>	<button>删除</button>	<button>保存</button>

增加

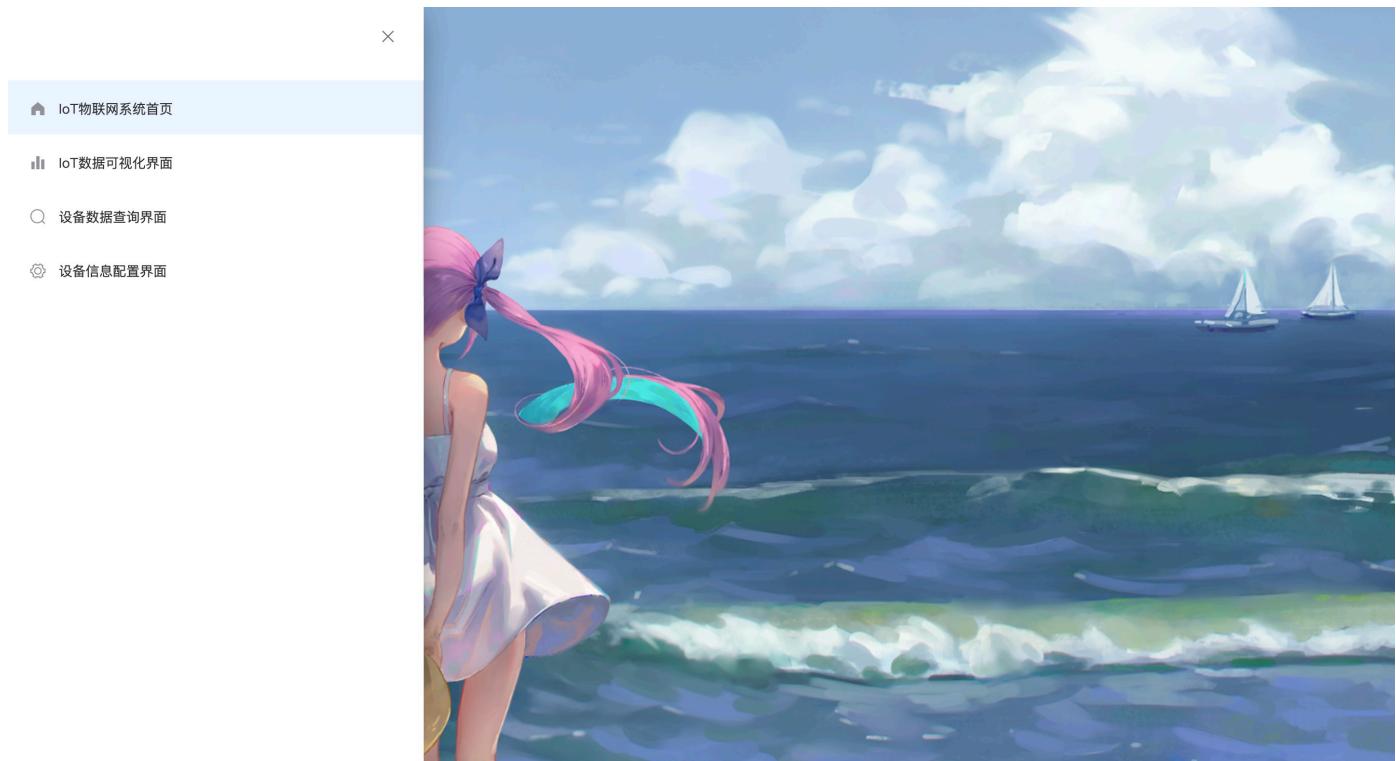
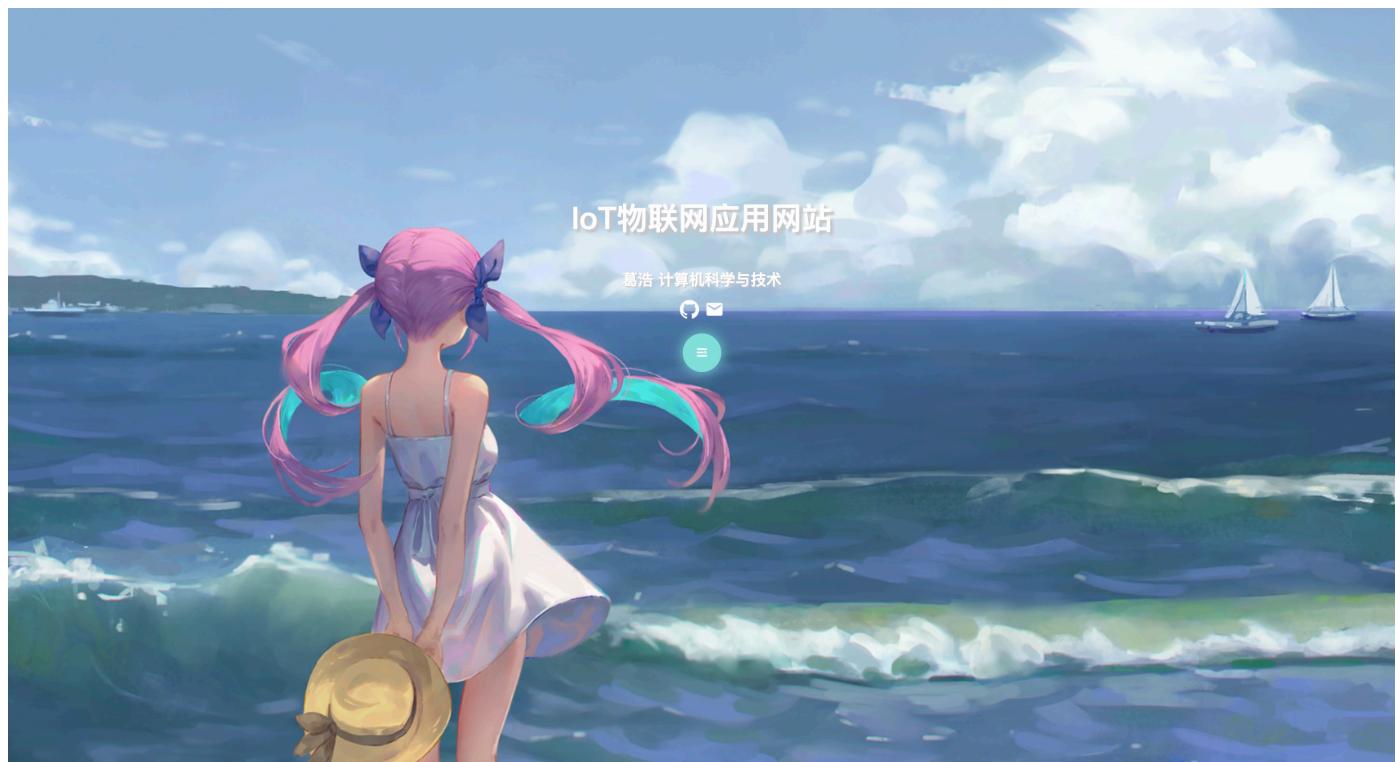
共 5 条 10条/页 < 1 > 前往 1 页

导航栏和主页

导航栏的实现着实花费了我一番功夫，由于我为整个系统布置了背景图，如果在上方或者左侧添加一个导航栏，显然会非常突兀，在可视化界面这个问题会更加明显

而响应式导航栏的设计理念非常符合当前的系统设计，我将所有的导航信息都隐藏起来，平时只以左上角的蓝青色按钮的形式存在，当用户想要切换页面时，点击该按钮，就可以弹出抽屉式的导航栏

此时，主页的设计也就变得简洁有效了：标题、个人信息、github和邮箱链接，以及导航栏按钮



后端开发

基本内容

由于我在阿里实习期间就是用SpringBoot相关技术做后端开发，因此这一模块相对来说比较简单，基本的设计想法是，通过Controller、Service、DAO进行工程架构，其中Controller层以restful api的形式向前端提供接口，Service层进行具体业务逻辑的处理，DAO层则定义数据对象和数据库访问

具体内容代码和注释写得很清晰了，基本都是CURD，没有太多需要介绍的地方

缓存设计

这里有一个比较有趣的idea是利用redis做缓存设计，因为我在前端设置了定时器，定期往后端请求数据，而后端的数据又是实时更新的，如果同一时刻有很多用户同时访问该网站，那么缓存就会起到减轻数据库压力的作用

缓存本质上就是设计key和value的形式，这里我将其统一为字符串形式，其中key本身为字符串，而value可能是各种类型的对象，这里利用fastjson将其转化为字符串，并在读取缓存时又将其重新恢复为对象

为了避免用户给出的key过长，我在RedisService中对key进行了又一层封装，进行md5加密

```
@Override  
public Boolean put(String key, String value) {  
    // cache key md5加密，避免key长度过大  
    key = DigestUtils.md5DigestAsHex(key.getBytes());  
    redisTemplate.opsForValue().set(key, value, expireTime, TimeUnit.SECONDS);  
    return true;  
}
```

一个基本的缓存使用例子如下：

```
List<IotMessageDO> result = null;  
if (redisService.exists(key)) {  
    result = JSON.parseObject(redisService.get(key),  
        new TypeReference<List<IotMessageDO>>(){});  
} else {  
    List<IotMessageDO> dataByKeyword = iotMessageMapper.getByKeyword(keyword);  
    redisService.put(key, JSON.toJSONString(dataByKeyword), expireTime);  
    result = dataByKeyword;  
}
```

由于数据是实时刷新的，因此必须要为缓存设置过期时间expireTime，避免前端读取到过时的数据，值得一提的是，利用缓存过期时间这一特性，我实现了监听设备在线数量的功能

每次读取到mqtt的消息都会往数据库里插入一条记录，此时我会顺便往缓存里也存入这条记录，key为该iot设备的ID，并设置缓存过期时间为3s，以expireTime为间隔，如果期间没有再次收集到同一iot设备的数据来更新缓存，那么对应的key就会失效，此时就可以判定为该设备不在线

开发体会

开发期间遇到了许多问题，通过搜索引擎和查看源码自我分析也逐一将其解决了，这个过程是漫长而痛苦的，但也是收获满满的，这里将我之前遇到的问题罗列出来

跨域问题

在Vue中使用axios访问后端服务时，浏览器会遇到CORS跨域问题

跨域问题的出现是因为浏览器的同源策略，即必须保证两个页面具有相同的协议(protocol: http、https)，主机号(host)和端口号(port)，由于前后端服务的端口号不同，因此前端访问后端服务时就会遇上CORS跨域问题

解决方案：使用proxy代理，Vue其实是利用webpack-dev-server起一个本地服务器来加载前端代码，地址是localhost:8080，为了规避跨域问题，webpack-dev-server提供了proxy功能(底层是http-proxy-middleware)，使前端代码向本地服务器localhost:8080提交请求，然后由本地服务器去请求真正的后端服务器(Node没有跨域限制)到config/index.js文件中，找到proxyTable，加上：

```
proxyTable: {
  '/api': {
    target: 'http://localhost:8443', // 后端地址
    changeOrigin: true, // 允许跨域
    pathRewrite: {
      '^/api': '' // 重写路径
    }
  }
},
```

此时前端请求url中的所有 /api 都会被替换成 /，由于在 main.js 里把 axios 的 baseURL 设置成了 /api，因此调用 this.\$http.get("datainfo") 发起http请求时，前端的请求url其实是 /api/datainfo，然后根据我们的 proxyTable，会把这个 /api 替换成空，即url变成了 /datainfo，然后加上target地址，变成了 http://localhost:8443/datainfo，此时在后端springboot就要配置为 @GetMapping("datainfo")，而不是 @GetMapping("api/datainfo")

但注意，webpack-dev-server是面向开发阶段发挥作用的，因此将 npm run build 生成的代码部署到服务器之后，是没办法再使用proxyTable进行转发的，通常需要自己配置Nginx代理进行请求的转发

```
location /api/ {
  # 代理服务器
  # proxy_pass http://20.89.46.55:8443/;
  proxy_pass http://server:8443/;
  proxy_set_header Host $proxy_host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

绘制地图轨迹

Echarts是很好的绘制图表的工具，由于我们需要绘制出设备在地图上的信息，因此简单使用官网介绍的矢量地图数据是没有办法完成这个需求的，因为矢量数据仅精确到浙江省，而设备的地理位置是在杭州市

后来了解到Echarts可以兼容百度地图API，尽管相关的文档介绍很少，但起码有了解决思路

在vue根目录下的 `main.js` 中引入echart，挂载到vue的原型对象上，全局只需要用 `this.$echarts` 即可使用

```
import echarts from 'echarts'  
Vue.prototype.$echarts = echarts
```

然后引入百度地图api

```
import 'echarts/extension/bmap/bmap'
```

遇到的问题是，Echarts绘制图表对setOption本身默认是做merge，不会删除原先已经绘制的旧点，如果我要在地图上绘制点，并实时刷新数据，就会出现很多历史数据点扎堆叠在一起的情况

为了解决这个问题，有两种思路：

- 删除整个ehcharts实例对象，并进行重新绘制：重载资源太浪费了

```
// 重置echarts，性能低，但目前官方并没有提供更好的方案  
resetEcharts() {  
    // console.log(this.chartInstance.getOption())  
    this.chartInstance.clear()  
    this.chartInstance.setOption({bmap:{}})  
  
    this.bmap = this.chartInstance.getModel().getComponent('bmap').getBMap()  
    this.bmap.addControl(new BMap.ScaleControl({anchor: BMAP_ANCHOR_BOTTOM_LEFT}))  
    // 地图左下角显示比例尺  
    this.bmap.setMapStyleV2({styleId: '8394c7f10d2cb459727e8c39ba0a3650'})  
}
```

- setOption时删除旧的部分组件，做属性的替换而不是合并：<https://echarts.apache.org/zh/api.html#echartsInstance.setOption>，参考replaceMerge参数

```
this.chartInstance.setOption(dataOption, {replaceMerge: 'series'})
```

当时我觉得这个参数好啊，完美解决了我目前的问题，但使用了之后发现并没有预期的结果，为此我以为是出了什么bug，还给Echarts提了[issue](#)

最后发现是我使用Echarts4.9.0版本不支持replaceMerge特性，需要升级到echarts5，然而升级到5之后，又和原先的代码有很多兼容性的冲突，故不得不放弃

- 最终没有特别好的解决方案，只能是按照merge的特点，把series的数据进行重新赋值重载(只要新的series数组和之前的长度相等、且属性完整，就会做等量替换)，尽管多写了一些不必要的多余代码，但起码实现了功能

响应式导航栏的设计

一开始设计导航栏，直接贴在网页上方或者左侧显得非常突兀，此外，如果将每一个导航项都直接详细的列出来，适配移动端的话，就会因为移动端屏幕宽度的限制，导致在电脑端完美显示的布局在移动端可能就乱掉了

因此我想采用响应式导航栏的设计方式，用一个按钮来代替原来的导航栏，通过点击这个按钮来显示或隐藏具体的导航信息，可惜的是Element UI并没有提供响应式导航栏的组件，因此只能自己手写一个了

幸运的是，Element UI提供了抽屉组件`drawer`，我们只需要将菜单组件`menu`嵌入抽屉组件之中，基础的导航栏就写好了，当然这一部分要默认隐藏

然后再设计一个显式的按钮，只有当按钮按下时，带有菜单的抽屉才会从左侧弹出，点击右侧任意空白空间，导航栏又会被隐藏，至此，响应式导航栏就设计完毕了

```
<template>
<div>
  <el-button
    type="menu"
    icon="el-icon-s-unfold"
    @click="drawer = true"
    circle
  ></el-button>
  <el-drawer :visible.sync="drawer" direction="ltr" class="drawer">
    <el-menu
      :default-active="activeIndex"
      mode="vertical"
      router
      @select="handleSelect"
      @open="handleOpen"
      @close="handleClose"
    >
      <el-menu-item index="/index">
        <i class="el-icon-s-home"></i>
        <span slot="title">IoT物联网系统首页</span>
      </el-menu-item>
      <el-menu-item index="/screen">
        <i class="el-icon-s-data"></i>
        <span slot="title">IoT数据可视化界面</span>
      </el-menu-item>
      <el-menu-item index="/query">
        <i class="el-icon-search"></i>
        <span slot="title">设备数据查询界面</span>
      </el-menu-item>
      <el-menu-item index="/config">
        <i class="el-icon-setting"></i>
        <span slot="title">设备信息配置界面</span>
      </el-menu-item>
    </el-menu>
  </el-drawer>
</div>
```

```
</template>
```

剩余的部分就是对Echarts和Element UI的使用了

Docker相关的配置

写完代码之后，想要部署到服务器，一个想法就是，实在不想每次都重新安装配置mysql、redis、emqx、nginx、java等软件了，因此就尝试使用docker的方式，将其整体打包成一个镜像组，能够统一编排和部署

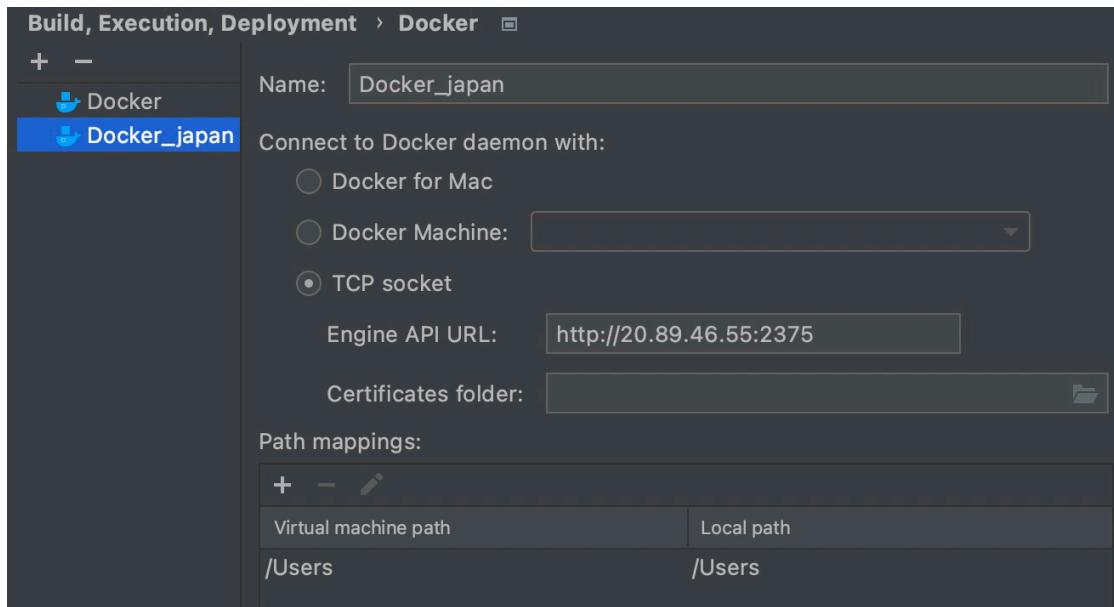
这里我有一台自己的服务器：20.89.46.55，想要在本机直接对远程服务器上的docker进行配置和启动，因此在idea上安装了docker插件

服务端配置docker，开启远程安全访问：

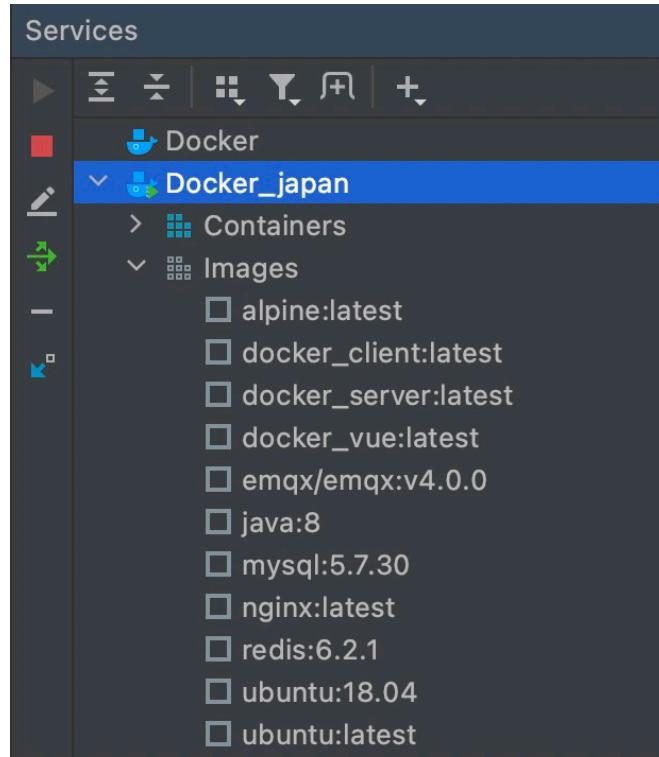
```
sudo install docker.io
sudo vim /lib/systemd/system/docker.service
# 修改: ExecStart=/usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:2375

# 加载配置+重启docker
sudo systemctl daemon-reload
sudo systemctl restart docker
# 验证
curl http://127.0.0.1:2375
```

idea安装docker插件，配置tcp，显示连接成功：



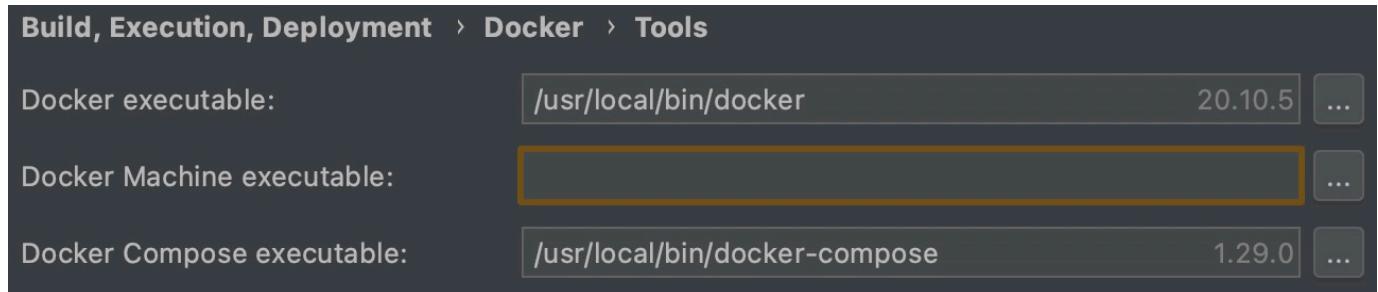
下方弹出services窗口，选中Docker_japan并连接，这个时候就可以显示我的远程机器上的docker镜像和容器了，由于之前我已经启动过docker-compose，因此机器上已经存在跟本项目相关的所有镜像了：



注意，想要启动远程机器的docker，本地机器得能够支持docker命令，因此本地先安装docker：

```
brew install --cask docker
```

然后在Docker插件->Tools栏下配置路径：



此时就可以编写docker-compose相关的yml文件对所需第三方软件镜像进行编排管理了：

```
version: "3"

services:
  server:
    build: ./server # Dockerfile文件路径，注意是目录名，而不是文件名
    container_name: iot_server # 容器名
    restart: always
    depends_on:
      - emqx
      - mysql
      - redis
      - client
    ports:
      - "8443:8443"
```

```

vue:
  build: ./vue
  container_name: iot_vue
  restart: always
  depends_on:
    - server
  ports:
    - "8080:8080"

client:
  build: ./client
  container_name: iot_client
  restart: always
  depends_on:
    - emqx
  ports:
    - "8442:8442"

emqx:
  image: emqx/emqx:v4.0.0
  container_name: iot_emqx
  restart: always
  ports:
    - "1883:1883"
    - "18083:18083"
    - "8883:8883"
    - "8083:8083"
    - "8084:8084"

mysql:
  image: mysql:5.7.30 # mysql镜像版本
  container_name: iot_mysql # 容器名
  restart: always
  volumes:
    - ./mysql/init:/docker-entrypoint-initdb.d/ # 初始化建库建表操作，不知道为啥没有生效
    - ./mysql/data:/var/lib/mysql # 将MySQL容器中的/var/lib/mysql目录挂载到宿主机的目
录./mysql/data上
  environment:
    MYSQL_ROOT_PASSWORD: "gehao"
    MYSQL_USER: "gehao"
    MYSQL_PASS: "gehao"
  ports:
    - "3306:3306"

redis:
  image: redis:6.2.1 # redis镜像版本
  container_name: iot_redis # 容器名
  restart: always

```

```

volumes:
  - ./redis/data:/data # 将Redis容器中的/data目录挂载到宿主机的目录./redis/data上
command: redis-server
ports:
  - "6379:6379"

```

接下来就可以利用idea在本地直接运行远程的docker-compose了，它会先查看机器上是否已经有目标镜像了，如果没有就build相关镜像，否则就直接利用镜像生成对应的容器启动运行，`depends on`参数可以规定这些容器启动的先后顺序，保证依赖的服务能够先后启动

结果一看，springboot容器连接mysql总是失败，通过查看docker状态发现它一直处于restarting状态，根本没能运行起来

```

gehao@sakura:~$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS              NAMES
a3b7e31e8d05        mysql:5.7.30      "docker-entrypoint.s..."   9 minutes ago     Restarting (1)
17 seconds ago          iot_mysql

```

后来看了docker-compose的官方文档才发现，这些被统一启动的机器默认在一个子网下，每个容器可以被看做是一个独立的IP地址，然而之前我在springboot的application.properties文件中，依旧使用localhost:port来连接服务的，显然这没法适用于容器之间的网络连接和通信

实际上，docker-compose会把yml文件中给每个镜像/容器的label作为其对应的IP地址，因此我们只需要用server、client、vue、emqx、mysql、redis这些label来代替配置项中的localhost即可完成网络通信

```

spring.datasource.url=jdbc:mysql://mysql:3306/iot?characterEncoding=UTF-
&serverTimezone=GMT%2B8&useSSL=false&autoReconnect=true&allowPublicKeyRetrieval=true
spring.redis.host=redis
iot.broker=tcp://emqx:1883

```

接下来mysql的启动正常了，但数据库的初始化失败了，本地的init文件夹没有被正确映射到远程容器的`/docker-entrypoint-initdb.d`文件夹(该文件夹下的所有sh和sql文件都会在初始化时执行)，因此我写的init.sql没有被执行，先尝试ssh登上远程机器，进入mysql容器，手动建库建表，此时springboot连接mysql终于成功了

springboot连接redis的过程非常顺利

但springboot一直没能连上emqx，当时我以为是emqx本身的问题，于是又给emqx提交了[issue](#)，在emqx开发人员的帮助下，我定位了自己的问题：当emqx的broker还没启动，springboot的docker就启动开始监听了，导致没法连接上emqx：这是docker自身的问题，前面所说的depends on参数只会控制docker容器之间的启动顺序，却不会监听容器内部服务的启动，也就是说，可能emqx的容器启动了，但emqx本身还没启动，但这个时候springboot容器已经启动开始连接emqx了，所以发生了错误

docker-compose的这个问题被很多很多人吐槽...

解决方案也还是有的，在springboot的Dockfile中使用[wait-for-it](#)脚本，只有监听到emqx:1883端口启动服务之后，springboot所属的server容器才开始启动，在server的Dockerfile修改ENTRYPOINT：

```
ENTRYPOINT [ "/wait-for-it.sh", "emqx:1883", "--timeout=60", "--", "java", "-jar",
"/iotserver.jar" ]
```

server镜像运行的结果：等待34s后emqx启动，springboot才开始运行

```
wait-for-it.sh: waiting 60 seconds for emqx:1883
wait-for-it.sh: emqx:1883 is available after 34 seconds

.
/\ \ / ____' - _ _ - ( )_ __ _ _ _ \ \ \ \
( ( )\ __ | ' _ | ' _ \ / _` | \ \ \ \
\ \ \ _ ) | ( ) | | | | | ( _| | ) ) )
' | ( _ | . _ | ( _ | _ \ , | / / /
=====|_|=====|__/_=/__/_/_/
:: Spring Boot ::          (v2.1.1.RELEASE)
```

根据上述手段，对mysql和redis也进行监听，由于ENTRYPOINT只能放一条命令，因此我们需要把wait-for-it的这段逻辑单独写到一个server-run.sh里

```
#!/usr/bin/env bash
# 运行iot server的脚本，需要等待mysql, redis, emqx的服务起来之后再启动jar包
/wait-for-it.sh mysql:3306 &&
/wait-for-it.sh redis:6379 &&
/wait-for-it.sh emqx:1883 --timeout=60 &&
java -jar /iotserver.jar
```

然后修改Dockerfile即可

```
FROM java:8
MAINTAINER gehao<sakura.gehao@gmail.com>
EXPOSE 8443
COPY ./target/iotserver-0.0.1-SNAPSHOT.jar /iotserver.jar
COPY ./wait-for-it.sh /wait-for-it.sh
COPY ./server-run.sh /server-run.sh
RUN chmod +x /server-run.sh # 加权限
ENTRYPOINT [ "./server-run.sh" ]
```

运行结果：

```
wait-for-it.sh: waiting 15 seconds for mysql:3306
wait-for-it.sh: mysql:3306 is available after 1 seconds
wait-for-it.sh: waiting 15 seconds for redis:6379
wait-for-it.sh: redis:6379 is available after 0 seconds
wait-for-it.sh: waiting 60 seconds for emqx:1883
wait-for-it.sh: emqx:1883 is available after 28 seconds

.
/\ \ / ____' - _ _ - ( )_ __ _ _ _ \ \ \ \
```

```
( )\__| '_| '_| | '_` | \ \\ \\
 \|_)(_|_)| | | || (_|_ ) ) )
 ' |__| .__|_|_|_|_|_\_, | / / /
 =====|_|=====|_|/_=/\_/_/
 :: Spring Boot ::           (v2.1.1.RELEASE)
```

还剩下最后一个问题：之前提到的mysql容器目录无法挂载，这个问题我排查了三四天，一直没有进展，一切显示都很正常，可目录就是没法挂载，直到后来我直接ssh登上远程服务器，本地运行了`docker-compose up -d`之后，发现竟然挂载成功了，最终我发现了问题所在：

因为我是用idea的docker插件，在本地机器起远程的docker容器，按照docker-compose所说，就相当于是把本地的目录挂载到远程机器的docker容器内的目录下，显然这是不合理的，远程主机的kernel它也没法访问本地的文件系统啊😭，因此挂载结果始终只有目录却没有文件，最终只要在远程主机上直接跑docker compose就好了，数据库表啥的都能够自动建好了

最后再把老师给的iotclient打包成jar包，把vue相关的文件也打好，一起组织进docker-compose里，所有的环境、包依赖都统一编排好了，用户只需要在本地安装好docker，就可以通过`docker-compose up`这条命令一键部署所有的服务了