文 Node.js + React Native 毕设：农业物联网监测系统的开发手记

javascript node.js react-native **DesGemini** 1 天前发布

毕设大概是大学四年里最坑爹之一的事情了，毕竟一旦选题不好，就很容易浪费一年的时间做一个并没有什么卵用，又不能学到什么东西的鸡肋项目。所幸，鄙人所在的硬件专业，指导老师并不懂软件，他只是想要一个农业物联网的监测系统，能提供给我的就是一个Oracle 11d数据库，带着一个物联网系统运行一年所保存的传感器数据...That's all。然后，因为他不懂软件，所以他显然以结果为导向，只要我交出一个移动客户端和一个服务端，并不会关心我在其中用了多少坑爹的新技术。

那还说什么？上！我以强烈的恶搞精神，决定采用业界最新最坑爹最有可能烂尾的技术，组成一个 Geek 大杂烩，幻想未来那个接手我工作的师兄的一脸懵逼，我露出了邪恶的笑容，一切只为了满足自己的上新欲。

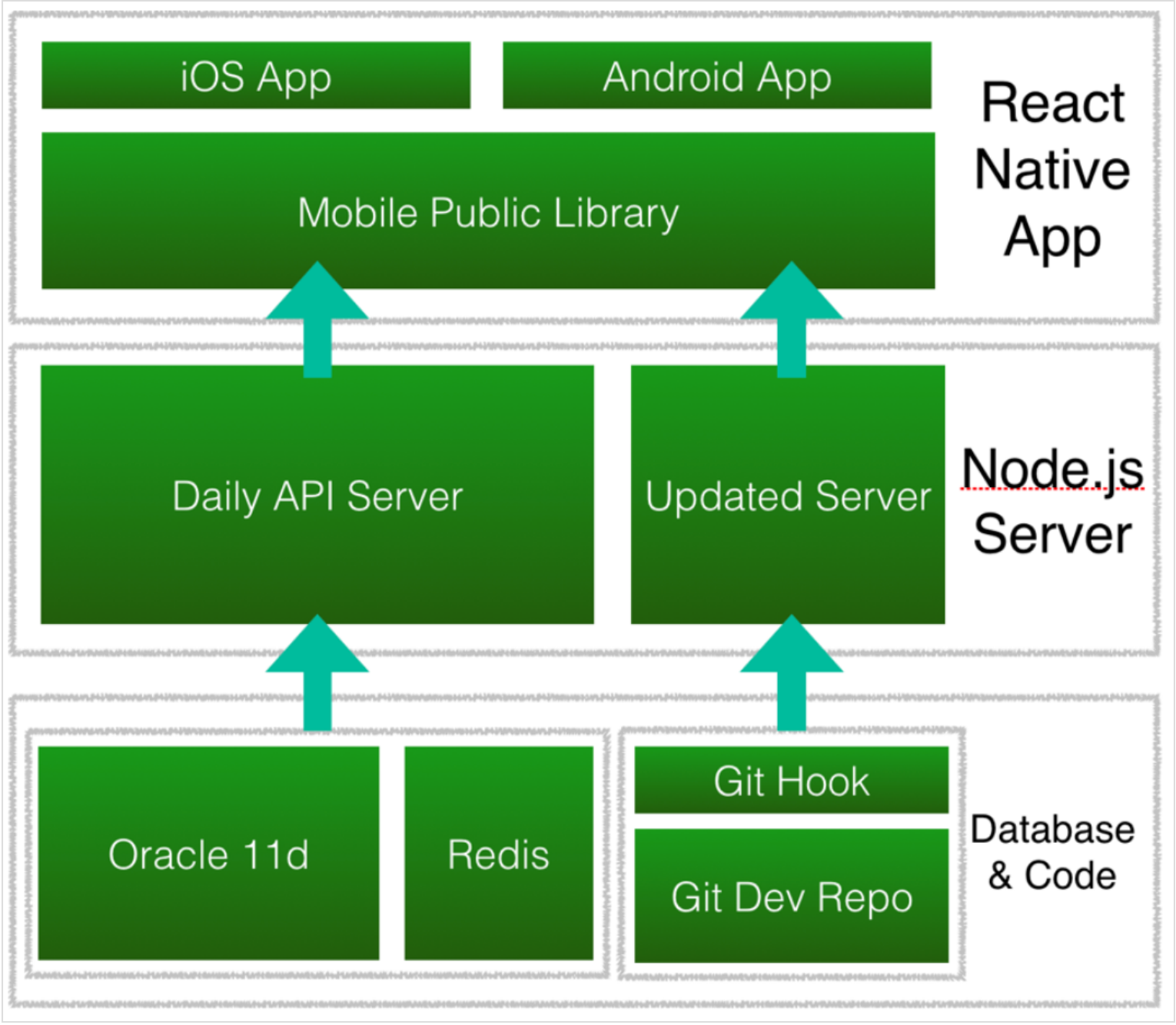
全部代码在 GPL 许可证下开源：

- 服务端代码：https://github.com/CauT/the-wall
- 客户端代码：https://github.com/CauT/Night...

由于数据库是学校实验室所有，所以不能放出数据以供运行，万分抱歉~。理论上应该有一份文档，但事实上太懒，不知道什么时候会填坑~。

总体架构

OK，上图说明技术框架。



该物联网监测系统整体上可分为三层：数据库层，服务器层和客户端层。

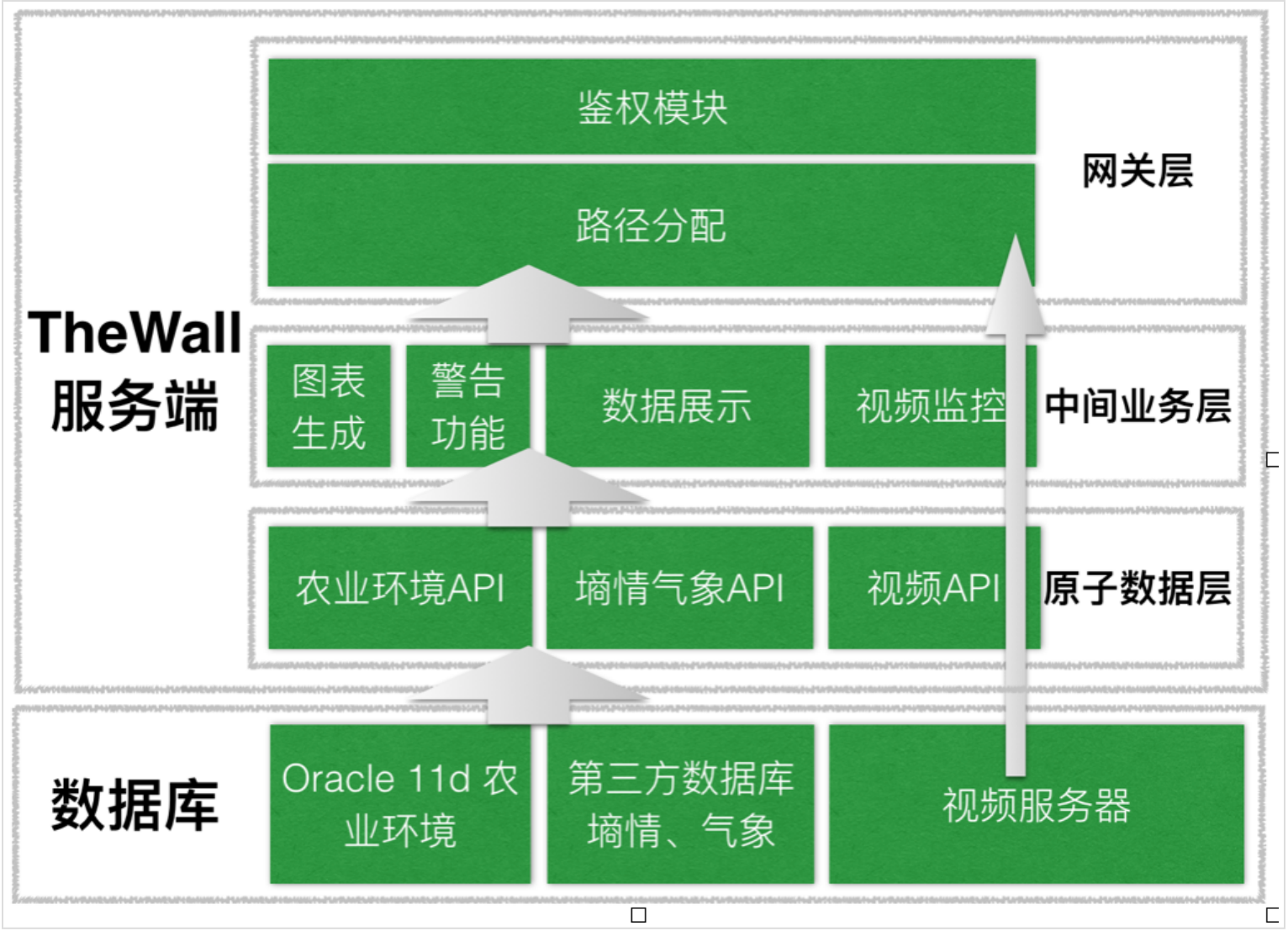
数据库和代码层

数据库层除了原有的Oracle 11d数据库以外，还额外增加了一个Redis数据库。之所以增加第二个数据库，原因为：

1. Node.js 的 Oracle 官方依赖 node-oracledb 没有ORM，也就是说，所有的对数据库的操作，都是直接执行SQL语句，简单粗暴，我担心自己孱弱的数据库功底（本行是 Android 开发）会引发锁表问题，所以通过限制只读来避开这个问题。
2. 由于该系统服务于农业企业的内部管理人员，因此其账号数量和总体数据量必然有限，因此使用 redis 这种内存型数据库，可以不必考虑非关系型数据库在容量占用上的劣势。读取速度反而较传统的 SQL 数据库有一定的优势。
3. 使用非关系型数据库比关系型数据库好玩多了（雾
4. 之所以写了右边的Git部分，是因为原本打算利用docker技术搞一个持续集成和部署的程序，实现提交代码=>自动测试=>更新服务器部署更新=>客户端自动更新 这样一整套持续交付的流程，然而最后并没有时间写。

服务器层

服务器层，采用 Node.js 的 Express 框架作为客户端的 API 后台。因为 Node.js 的单线程异步并发结构使之可以轻松实现较高的 QPS，所以非常适合 API 后端这一特点。其框架设计和主要功能如下图所示：



像网关层：鉴权模块这么装逼的说法，本质也就是`app.use(jwt({secret: config.jwt_secret}).unless({path: ['/signin']}));`一行而已。因为是直接从毕业论文里拿下来的图，毕业论文都这尿性你们懂的，所以一些故弄玄虚敬请谅解。

客户端层

客户端层绝大部分是 React Native 代码，但是监控数据的图表生成这一块功能（如下图），由于 React Native 目前没有开源的成熟实现；试图通过 Native 代码来画图表，需要实现一个 Native 和 React Native 互相嵌套的架构，又面临一些可能的困难；故而最终选择了内嵌一个 html 页面，前端代码采用百度的 Echarts 框架来绘制图表。最终的结构就是大部分 React Native + 少部分 Html5 的客户端结构。

另外就是采用了 Redux 来统一应用的事件分发和 UI 数据管理了。可以说，React Native 若能留名青史，Redux 必定是不可或缺的一大原因。这一点我们后文再述。

细节详述

服务端层

服务端接口表：

API名	路径	功能说明	参数	参数说明

称		说明		
账号登陆	/v1/signin	账号登陆	username; password	账号和密码，其中密码要求为在客户端做sha256加密后的结果
账号注销	/v1/signout	账号注销	username;	账号
获取传感器种类	/v1/device/info/type_list	获取传感器种类	无	无
获取监测站编号	/v1/device/info/station_list	获取监测站编号	无	无
产生历史图表(参数方案1)	/v1/utils/ generate_graph/basic	产生历史图表	start_time; end_time; device_ids	指定开始时间和截止时间和一个传感器ID数组
产生历史图表(参数方案2)	/v1/utils/generate_graph/advance	产生历史图表	start_time; end_time; station_name; device_type	指定开始时间和截止时间;筛选传感器种类；筛选站点名称
获取当前数据	/v1/data/agri_env/current	产生历史图表	deviceType; stationName	筛选传感器种类；筛选站点名称
获取历史数据	/v1/data/agri_env/history	获取历史数据	deviceType; stationName	筛选传感器种类；筛选站点名称

服务端程序的编写过程中，往往涉及到了大量的异步操作，如数据库读取，网络请求，JSON解析等等。而这些异步操作，又往往会因为具体的业务场景的要求，而需要保持一定的执行顺序。此外，还需要保证代码的可读性，显然此时一味嵌套回调函数，只会使我们陷入代码几乎不可读的回调地狱（Callback Hell）中。最后，由于JavaScript单线程的执行环境的特性，我们还需要避免指定不必要的执行顺序，以免降低了程序的运行性能。因此，我在项目中使用Promise模式来处理多异步的逻辑过程。如下代码所示：

```
    var device_ids;
    var queryPromises = [];

    secureCheck(req, res);

    device_ids = req.query.device_ids.toString().split(';');

    for(let i=0; i<device_ids.length; i++) {
      queryPromises.push(createQuerySingleDeviceDataPromise(
        req, res, device_ids[i], req.query.start_time, req.query.end_time));
    };

    Promise.all(queryPromises)
    .then(function(filtereds) {
      renderGraph(req, res, filtereds);
    }).catch(function(err) {
      res.status(500).json({
        status: 'error',
        message: err.message
      });
    })
  } catch(err) {
    res.status(500).json({
      status: 'error',
      message: err.message
    });
  }
});
```

这是生成指定N个传感器在一段时间内的折线图的逻辑。显然，剖析业务可知，我们需要在数据库中查询N次传感器，获得N个值对象数组，然后才能去用N组数据渲染出图表的HTML页面。 可以看到，外部核心的Promise控制的流程只集中于下面的几行之中：`Promise.all(queryPromises()).then(renderGraph()).catch()`。即，只有获取完N个传感器的数值之后，才会去渲染图表的HTML页面，但是这N个传感器的获取过程却又是并发进行的，由Promise.all()来实现的，合理地利用了有限的机器性能资源。

然而，推入queryPromises数组中的每个Promise对象，又构成了自己的一条Promise逻辑链，只有这些子Promise逻辑链被处理完了，才可以说整个all()函数都被执行完了。子Promise逻辑链大致地可以总结为以下形式：

```
function() {
    return new Promise().then().catch();
}
```

其中的难点在于：

1. 合理地切分整套业务逻辑到不同的then()函数中，且一个then()中只能有一个异步过程。
2. 函数体内的异步过程所产生的新的Promise逻辑链必须被通过return的方式挂载到父函数的Promise逻辑链中，否则即可能形成一个有先有后的控制流程。
3. catch()函数必须要做好捕捉和输出错误的处理，否则代码编写过程中的错误即不可能被发现，异步编程的整个过程也就无从继续下去了。
4. 值得一提的是Promise模式的引入。Node.js 自身不带有Promise，可以引入标准的ECMAScript的Promise实现，然而其功能较为简陋，对于各种API的实现过于匮乏，因此最后选择了bluebird库来引入Promise模式的语言支持。

由此我们可以看到，没有无缘无故的高性能。Node.js 的高并发的优良表现，是用异步编程的高复杂度换来的。当然，你也可以选择不要编程复杂度，即不采用 Promise，Asnyc 等等异步编程模式，任由代码沦入回调地狱之中，那么这时候的代价就是维护复杂度了。其中取舍，见仁见智。

客户端层

客户端主要功能如下表所示：

模块名称		模块功能说明
数据展示	农业环境信息	• 用于展示各个传感器的实时数据，下拉可以刷新数据，默认2分钟刷新一次。 • 数据可以选择监测站分组和传感器种类分组两种展示方式。 • 传感器数据标注单位，要求传感器数据准确无误。 • 给出每种传感器的安全范围。
	土壤墒情	
	气象信息	
	田间气象	

图表生成		• 给出指定参数，生成对应图表，同时给出每种传感器的安全范围。
		• 参数列表：
		  ○ 起始时间和终止时间
		  ○ 5个以内的传感器的编号
		  ○ 是否需要每个传感器这段时间的平均值
		• 图像生成模块供所有数据展示使用

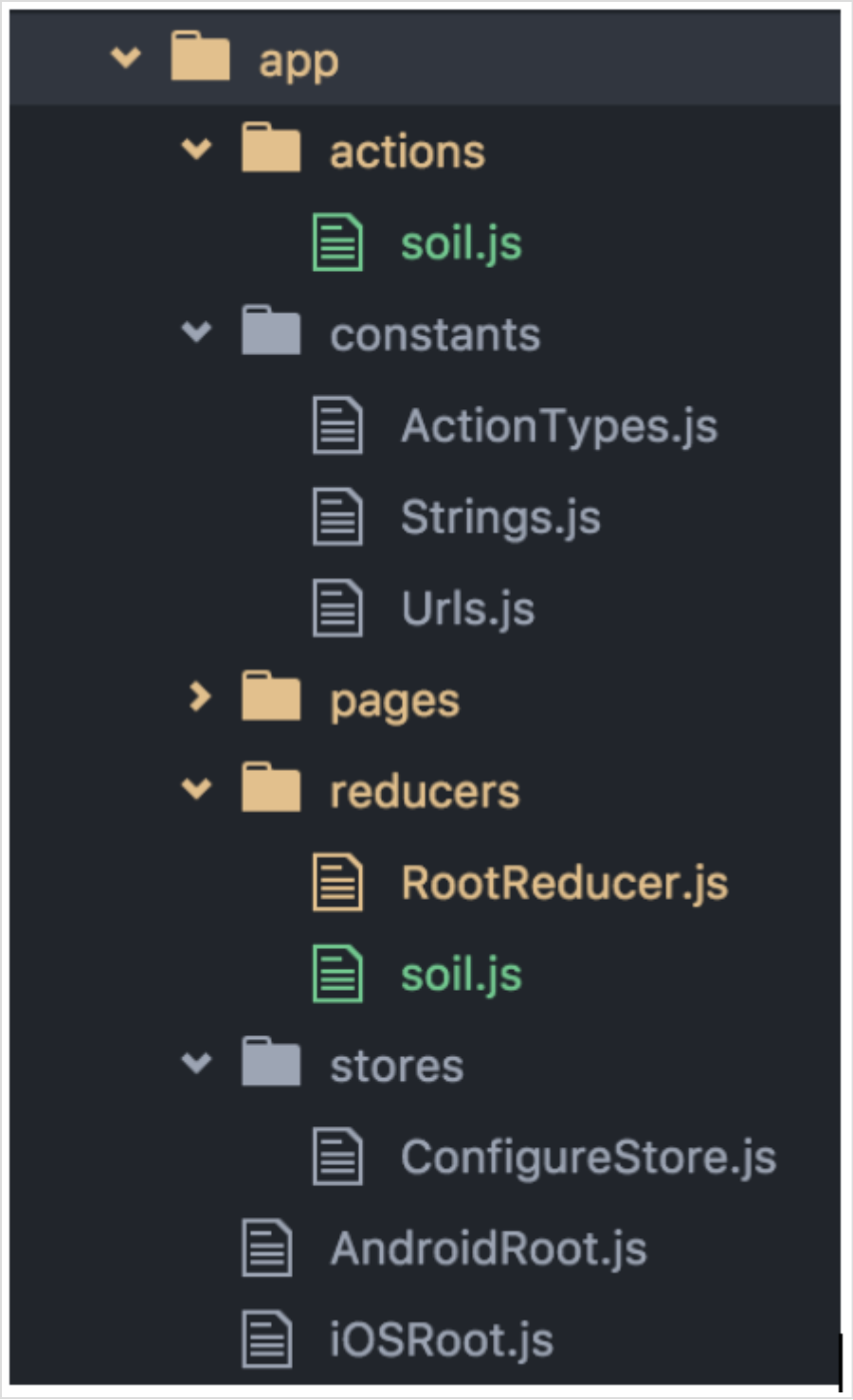
权限控制		• 与服务器验证账号密码，保证物联网数据的安全性。 • 服务器的对所有API请求都要求带有session ID，防止非法请求。 • 对同一IP限定每小时API调用次数上限为120次。超过则拒绝请求。 • 账号的添加必须在数据库上操作。 • 安全手势锁功能。一旦切换出App，则必须通过手势验证才能进入。 • 超时踢出功能。服务器的session ID有效期为1天，超过则需要重新验证账号密码。
告警功能		• 可设置若干项传感器数据的警戒值。 • 可设置警告方式为，则使手机响铃或震动，并推送警告消息到下拉窗。

接下来简单介绍下几个主要页面。可以发现 iOS 明显比 Android 要来的漂亮，因为只对 iOS 做了视觉上的细调，直接迁移到 Android 上，就会由于屏幕显示的色差问题，显得非常粗糙。所以，对于跨平台的 React Native App 来说，做两套色值配置文件，以供两个平台使用，还是很有必要的。

上图即是土壤墒情底栏的当前数据页面，分别在Android和iOS上的显示效果，默认展示所有当前的传感器的数值，可以通过选择传感器种类或监测站编号进行筛选，两个条件可以分别设置，选定后再点击查找，即向服务器发起请求，得到数据后刷新页面。由于React Native 的组件化设计，刷新将只刷新下侧的DashBoard部分，且，若有上次已经渲染过的MonitorView，则会复用他们，不再重复渲染，从而实现了降低CPU占用的性能优化。MonitorView，即每一个传感器的展示小方块，自上至下依次展示了传感器种类，传感器编号，当前的传感器数值以及该传感器显示数值的单位。MonitorView和Dashboard均被抽象为一个一般化，可复用的组件，使之能够被利用在气象信息、病虫害监测之中，提升了开发效率，降低了代码的重复率。

上图是土壤墒情界面的历史数据界面，分别在Android和iOS上的展示效果，默认不会显示数据，直到输入了传感器种类和监测站编号，选择了年月日时间后，再点击查找，才会得到结果并显示出来。该界面并非如同当前数据界面一样，Android和iOS代码完全共用。原因在于选择月日和选择时间的控件，Android和iOS系统有各自的控件，它们也被封装为React Native中不同的控件，因此，两条绿色的选择时间的按钮，被封装为HistoricalDateSelectPad，分别放在componentIOS和componentAndroid文件夹中。界面下侧的数据监测板，即代码中的Dashboard，是复用当前数据中的Dashboard。

上图是土壤墒情界面的图表生成界面，分别在Android和iOS上的展示效果。时间选择界面，查找按钮，选择框，均可复用前两个界面的代码，因此无需多提。值得说的是，生成的折线图，事实上是通过内嵌的WebView来显示一个网页的。图表网页的生成，则依靠的百度Echarts 第三方库，然后服务端提供了一个预先写好的前端模板，Express框架填入需要的数据，最后下发到移动客户端上，渲染生成图表。图表支持了多曲线的删减，手指选取查看具体数据点，放大缩小等功能。



上图则是实际项目应用中的Redux相关文件的结构。stores中存放全局数据store相关的实现。

actions中则存放根据模块切割开的各类action生成函数集合。在 Redux 中，改变 State 只能通过 action。并且，每一个 action 都必须是 Javascript Plain Object。事实上，创建 action 对象很少用这种每次直接声明对象的方式，更多地是通过一个创建函数。这个函数被称为Action Creator。

reducers中存放许多reducer的实现，其中RootReducer是根文件，它负责把其他reducer拼接为一整个reducer，而reducer就是根据 action 的语义来完成 State 变更的函数。Reducer 的执行是同步的。在给定 initState 以及一系列的 actions，无论在什么时间，重复执行多少次 Reducer，都应该得到相同的 newState。

性能测试

服务端

测试工具：OS X Activity Monitor（http_load）

性能指标	测试结果
内存占用	平均：43.3MB；峰值：125.1MB
CPU占用	平均：24.7%；峰值：81.9%
QPS	峰值237.675次/秒

客户端

iOS

测试工具：Xcode 7.3

性能指标	测试结果
内存占用	143.7MB
CPU占用	平均3.5%；峰值41%
安装包大小	5.2MB

Android

测试工具：Android Studio 1.2.0

性能指标	测试结果
内存占用	34.59MB
CPU占用	平均6.90%；峰值25.6%
安装包大小	7.6MB

代码量相关

性能指标	测试结果
客户端代码量	2176行
服务端代码量	1119行
Android独有代码量	506行
iOS独有代码量	446行
跨平台代码复用率	76.75%

简单总结

React Native 尽管在开发上具有这样那样的坑，但是因其天生的跨平台，和优于 Html5的移动性能表现，使得他在写一些不太复杂的 App 的时候，开发速度非常快，自带两倍 buff。

1 天前发布　更多

5 推荐　收藏

你可能感兴趣的文章

react-native:环境搭建　6 收藏，1.3k 浏览

react-native windows 环境搭配　4 收藏，786 浏览

react-native 初体验 - 使用 javascript 来写 iOS app　14 收藏，1.4k 浏览

讨论区

放出一个数据库的空库结构吧

欧阳大海 · 1 天前

和我去年的本科毕设差不多~ 我去年也是用node+react native做的。

dcy0701 · 6 分钟前

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题

提交评论

评论支持部分 Markdown 语法：`**bold**` `_italic_` `[link](http://example.com)` `> 引用` `` `code` `` `- 列表`。
同时，被你 @ 的用户也会收到通知

本文隶属于专栏

编程沉思录

予我风雷，以翱以翔

DesGemini
作者

关注专栏

系列文章

React Native 蛮荒开发生存指南 56 收藏， 3.1k 浏览

相关收藏夹

换一组

nodejs
4 个条目 | 1 人关注

node
6 个条目 | 0 人关注

音视频技术
4 个条目 | 1 人关注

分享扩散：