前端开发环境mock+持久化方案

缘起

前后端分离已经喊了很多年了，尽管现代前端开发都已经用上了三大框架，无论是`Angular`的前端`MVC`实践，还是`React`的`UI = f(data)`的函数式思想，或者是`Vuejs`的`.vue`文件集中管理模块的样式、逻辑与模板，其实都是在`UI`层以及其与数据层的通讯方式上下功夫，状态驱动和双向数据绑定并没有解决前端数据层对后端的依赖问题，当后端接口尚未开发和部署时，前端无法形成自己独立的数据层服务来为UI提供接近于真实业务场景的数据。

当前端应用变得日趋复杂时，架构层面自然需要演化出自己独立的数据层服务，这之中应该包含两方面：

- 与后端弱关联的页面状态管理，比如组件的显隐状态，某个数据展示组件的当前数据等，这一部分数据可以使用各个框架现有的状态管理工具来管理.
- 与后端强相关的，需要频繁使用`ajax`去接口请求的数据，这一部分数据，应该抽离成为独立的`API`服务层，由这一层统一处理数据的请求，请求参数和返回值的格式化处理等一系列工作。(对于抽象`API`层的好处，请参阅抽象API服务层的好处)

前一种更像是一种临时数据，应用（或组件）销毁关闭或卸载时即消失，而后一种数据则是需要在后端数据库进行持久化的。

我们今天讨论的就是针对这后一种数据的前端`mock`和持久化方案。

对后端弱依赖的前端开发流程

1. 根据产品原型设计和业务需求文档，与后端确定好接口文档，包含每个接口所要服务的场景、要满足的数据返回值和需要的请求参数等。
2. 前端开始根据设计稿还原产品设计，然后根据接口文档的约定使用自己的数据层服务生成符合业务需求的mock数据，并进行持久化存储，使用这些数据进行交互还原和各种增删改查的逻辑。
3. 当后端接口开始提供服务后，前端只需要修改响应的接口地址，即可开始于后端联调接口，无需再进业务逻辑的开发，仅需要重点关注后端提供的数据是否符合约定，是否符合场景需求和流程需求。

在以上流程中，仅第一步和第三步需要和后端进行一些沟通和联调工作，中间百分之八十的时间，前端可以完全不依赖后端而进行独立的业务逻辑开发和调试。

mock

`mock` 其实已经不是什么新鲜玩意了，借助于著名的 `mock.js` 库，我们几乎可以仅在前端数据层就生产出所有我们期望的数据来供 `UI` 层消费。

我们今天不讨论 `mockjs` 的语法和用法，这些网上已经有很多人讨论了。我们要讨论的是，如何让 `mock` 成为一种即插即用又方便卸载的服务，即当后端接口尚无法投入服务时，我们可以用 `mock` ，当后端接口准备就绪后，我们可以一键切换至真实接口。

首先，我们确定几点：

1. `mock` 只有在开发环境中使用
2. `mock` 分两部分：
    - 数据模拟
    - 请求拦截

基于此，我们提出以下几点需求

- 可以根据当前环境一键切换是否 `mock`
- 可以控制哪些接口需要 `mock` ，哪些接口不需要 `mock` ，只拦截那些需要 `mock` 的接口
- 拦截操作需要统一，不需要在每一个需要拦截的接口处写拦截方法

持久化

有了 `mock` ，我们还远达不到使用前端数据层模拟真实数据消费场景的目的。

因为真实的后端数据都是由持久化的，我们每次请求到的数据应该都是一样的，包括增删改查针对的应该都是同一个数据集，而不是每次都随机出不同的数据集。

所以，为了实现 `mock` 数据的前端持久化，我们引入了 `lowdb`

它是一个纯前端的轻量的数据持久化方案，同时支持 `nodejs` 端和浏览器端，其中， `nodejs` 端，使用一个 `.json` 文件作为其数据库来持久化数据，在浏览器端，则是使用 `LocalStorage` 来进行数据的持久化。

它的所有接口和方法都是基于 `lodash` ，只要能熟练使用 `lodash` 就能狗数量地操作它的 `API` 。

mock + 持久化

接下来，我们以一个 `vue-cli3` 项目为例，来实现 `mock + 持久化` 的统一方案。

mock目录在工程中的位置和本身的目录结构

考虑到 `mock` 仅在开发环境使用，所以 `mock` 的目录应该置于 `src` 目录之外，与其并列是最好的，目录机构如下:

```
-| project
  -|src
    -| api // 抽象API服务层，统一管理数据层API部分
      -| url.js // 统一管理接口地址
      -| site.js
      -| notice.js
      -| ...
     -| store // 非持久化状态数据管理
  -|mock
    -| index.js // 统一处理请求拦截
    -| lowdb.js // 引入持久化库
    -| utils.js // 公用方法
    -| models // 分别存放各个模块的mock数据生产逻辑
      -| site.js
      -| notice.js
      -| ...
  -|...
  -|package.json

```

index.js

```
import Mock from 'mockjs'; // 引入mockjs核心库
import { param2Obj } from './utils'; // 引入工具方法
// 引入各个模块的mock数据生产逻辑
import site from './models/site';
import unit from './models/unit';
import notice from './models/notice';

// 合并所有模块
const mocks = [
    ...site,
    ...unit,
    ...notice
];

// 导出mock主程序
export function mockXHR() {
    Mock.XHR.prototype.proxy_send = Mock.XHR.prototype.send;
    Mock.XHR.prototype.send = function() {
      if (this.custom.xhr) {
        this.custom.xhr.withCredentials = this.withCredentials || false;
        if (this.responseType) {
```

```
          this.custom.xhr.responseType = this.responseType;
        }
      }
      this.proxy_send(...arguments);
    };

    // 模拟异步请求包装
    function XHR2ExpressReqWrap(respond) {
      return function(options) {
        let result = null;
        if (respond instanceof Function) {
          const { body, type, url } = options;
          result = respond({
            method: type,
            body: JSON.parse(body),
            query: param2Obj(url)
          });
        } else {
          result = respond;
        }
        return Mock.mock(result);
      };
    }
    // 统一添加拦截，  isMock是一个开关，代表当前请求是否需要开启mock
    for (const i of mocks) {
      if (i.isMock) {
        Mock.mock(new RegExp(i.url), i.type || 'get', XHR2ExpressReqWrap(i.response));
      }
    }
}

// 创建模拟的响应数据
const responseFake = (url, type, respond) => {
    return {
      url: new RegExp(`/mock${url}`),
      type: type || 'get',
      response(req, res) {
        res.json(Mock.mock(respond instanceof Function ? respond(req, res) : respond));
      }
    };
};

export default mocks.filter(route => {
  return route.isMock;
}).map(route => {
    return responseFake(route.url, route.type, route.response);
});

```

这里面主要做了这么几件事：

- 统一对所有接口进行 `mock` 拦截
- 统一模拟异步请求
- 统一模拟返回

lowdb.js

```
import low from 'lowdb';  // 引入lowdb核心库
import lodashId from 'lodash-id';  引入lodash辅助函数库
import LocalStorage from 'lowdb/adapters/LocalStorage';  // 从lowdb引入存储引擎，浏览器环境使用Localstorage
const adapter = new LocalStorage('db'); // 创建一个新的数据库
const db = low(adapter); // 初始化数据库

db._.mixin(lodashId);  // 使用lodashId的扩展数据库API，因为有些数据库操作无法使用 lodash提供的函数完成

export default db;  // 导出数据库模块
```

创建好了这两个文件，我们看看如何在各个模块的 `mock` 数据生产逻辑中使用它们，我们以其中一个模块为例：

models/notice.js

```
import Mock from 'mockjs';  // 引入mock核心库
import db from '../lowdb.js'; // 引入数据库模块
import urls from '../../src/api/url.js'; // 引入请求地址库
import { transParamsToInt } from '../utils.js';  // 引入需要个工具函数

// 自定义的MOCK数据占位符
Mock.Random.extend({
  timeKey: function() {
      var times = ['2020-08', '2020-09', '2020-07', '2020-06'];
      return this.pick(times);
  },
  status: function() {
      var status = ['0', '1'];
      return this.pick(status);
  },
  filename: function() {
    var filename = Mock.Random.ctitle();
    var fmt = ['zip', 'pdf', 'excel', 'docx'];
    return filename + '.' + this.pick(fmt);
  },
  picname: function() {
    var filename = Mock.Random.ctitle();
    var fmt = ['png', 'jpg', 'gif', 'jpeg'];
    return filename + '.' + this.pick(fmt);
  },
  noticetype: function() {
    var status = ['recive', 'send'];
    return this.pick(status);
  }
});

// 定义数据库项—相当于一张表

```

```
let noticeList = [];

// 定义数据总量
const count = 100;

// 生成图片列表MOCK数据
function getPicList() {
  const piclist = [];
  for (let i = 0; i < 4; i++) {
    piclist.push(Mock.mock({
      'url': Mock.Random.url('http'),
      'filename': '@picname',
      'filedesc': '@ctitle(5,10)'
    }));
  }
  return piclist;
}


// 生成文件列表mock数据
function getFileList() {
  const piclist = [];
  for (let i = 0; i < 4; i++) {
    piclist.push(Mock.mock({
      'url': Mock.Random.url('http'),
      'filename': '@filename',
      'filedesc': '@ctitle(5,10)'
    }));
  }
  return piclist;
}


// 如果该表不存在或者表中没有数据，则创建新数据存储数据库，否则返回已经存在的数据
if (db && db.get('notice').size().value()) {  // size()获取数据表总量
  noticeList = db.get('notice').value(); // get()...value() 从数据库获取数据
} else {
  for (let i = 0; i < count; i++) {
    noticeList.push(Mock.mock({
      'crunit': '@county(true)',  // 随机生成一个地名
      'crunitid': '@increment',  // 一个自增数
      'noticeid': '@increment',
      'cruser': '@cname',  // 中文姓名
      'timekey': '@timekey', // 生成一个自定义日期
      'crtime': '2020-08-01 00:00:00',
      'content': '@cparagraph(4)', // 中文段落
      'read': '@integer(60, 100)', // 指定生成某个范围内的整数
      'readall': '@integer(100, 180)',
      'reply': '@integer(20, 40)',
      'readstatus': '@status', // 生成一个自定义状态值
      'type': '@noticetype',  // 生成一个自定义type值
      'piclist': getPicList(), // 调用函数生成一个图片列表
      'filelist': getFileList() // 调用函数生成一个文件列表
    }));
  }
  db.set('notice', noticeList).write();  // set() 新建或重写一个表，  .write () 执行写入操作
}
```

```
91
92  // 导出每个接口的模拟逻辑
93  export default [
94    {
95      url: urls['notice/list'], //获取列表
96      type: 'get',
97      isMock: false,  // 是否开启mock，这个开关可以将mock开关控制到具体接口
98      response: config => {
99        const query = transParamsToInt(config.query, ['noticeid', 'page', 'limit']); //
    转换参数类型
100       const { page = 1, limit = 10 } = query;  // 处理默认分页参数
101       if (config.query.page) { delete query.page; }
102       if (config.query.limit) { delete query.limit; }
103       const mockList =
104         db.get('notice')  //获取表中的所有数据
105         .filter(query) // 根据传入的查询条件进行过滤
106         .sortBy((o) => o.timeKey) // 根据某个字段进行排序
107         .slice(limit * (page - 1), limit * page) // 根据分页过滤返回数据
108         .groupBy('timekey') // 根据某个字段进行聚合
109         .reduce((r, v, k) => {
110             const arr = [];
111             arr.push({
112               timekey: k,
113               list: v
114             });
115             return r.concat(arr);
116           }, [])
117         .value(); // 组装为前端需要的数据格式
118       const total =
119         db.get('notice')
120         .filter(query)
121         .size()
122         .value(); // 获取符合查询条件的数据的总数
123
124       return {  // 返回模拟数据
125         code: '200',
126         datas: mockList,
127         link: [],
128         summary_info: {
129           total: total,
130           pageSize: limit,
131           pageNum: page
132         }
133       };
134     }
135   },
136
137   {
138     url: urls['notice/detail'],
139     type: 'get',
140     isMock: false,
141     response: config => {
142         return {
143             code: '200',
144             datas: noticeList,
145             link: [],
146             summary_info: {
147                 total: 10,
```

```
                pageSize: 10,
                pageNum: 1
            }
        };
    }
  }
];

```

那么，在各个模块的模拟逻辑中，主要就是做了两件事：

- 生成模拟数据
- 对数据进行持久化操作（包括写入和读取）

在模块 `mock` 逻辑中，我们引入了一个 `url` 模块, 是对所有接口的地址进行统一管理的地方

url.js

```
const urls = {

    // 通知相关接口映射
    'notice/list': '/workorder/findNotice', // 我发出的通知
    'notice/detail': '/workorder/findNoticeDetail', // 通知详情
    //......

};

export default urls;

```

在这个文件里，我们统一管理所有的接口后缀，这里是一个映射表，映射了我们的API服务层调用的地址和真实的地址之间的一一对应关系，这样可以应对后端接口地址的不确定性，当后端变更接口地址后，我们只需要在这里统一修改，而不需要单独去修改 `API` 层的每一处调用，同时， `API` 层与MOCK层都统一引入这一份地址库，可以保持请求的一致性。

api/notice.js

```
import urls from './url.js';

/**
 * [export 获取通知列表]
 *
 * @param   {[type]}  query  [query description]
 *
 * @return  {[type]}         [return description]
 */
export function fetchNoticeList(query) {
```

```
    return axios({
        url: urls['notice/list'], // 调用地址库的键，可以应对真实接口地址的变化，保持API层的稳定性
        method: 'get',
        params: query
    });
}

/**
 * [export 获取通知详情]
 *
 * @param   {[type]}  query  [query description]
 *
 * @return  {[type]}         [return description]
 */
export function fetchNoticeDetail(query) {
    return axios({
        url: urls['notice/detail'],
        method: 'get',
        params: query
    });
}
```

这里面，我们使用了 `axios`，实际上还可以改进，因为这个 `axios()` 其实也是我们对 `axios` 库的封装，这里用了这个名字，如果以后换其它请求库了，还用这个名字就不太好了，所以应该统一成与库无关的 `request()` 是比较好的实践：

```
export function fetchNoticeList(query) {
    return request({
        url: urls['notice/list'], // 调用地址库的键，可以应对真实接口地址的变化，保持API层的稳定性
        method: 'get',
        params: query
    });
}
```

调用API服务的组件 NoticeList.vue

```
<script>
import { fetchNoticeList } from '@/api/notice.js';

export default {
  //...
  methods: {
    getNotice(query) {
            const params = query || {};
            params.type = this.type === 'recive' ? 0 : 1;
            fetchNoticeList(params).then(data => {
                if (data.code === '200') {
```

```
                    this.datas = data.datas;
                    this.page = this.transPage(data.summary);
                }
            });
        },
    }
}
</script>
```

main.js

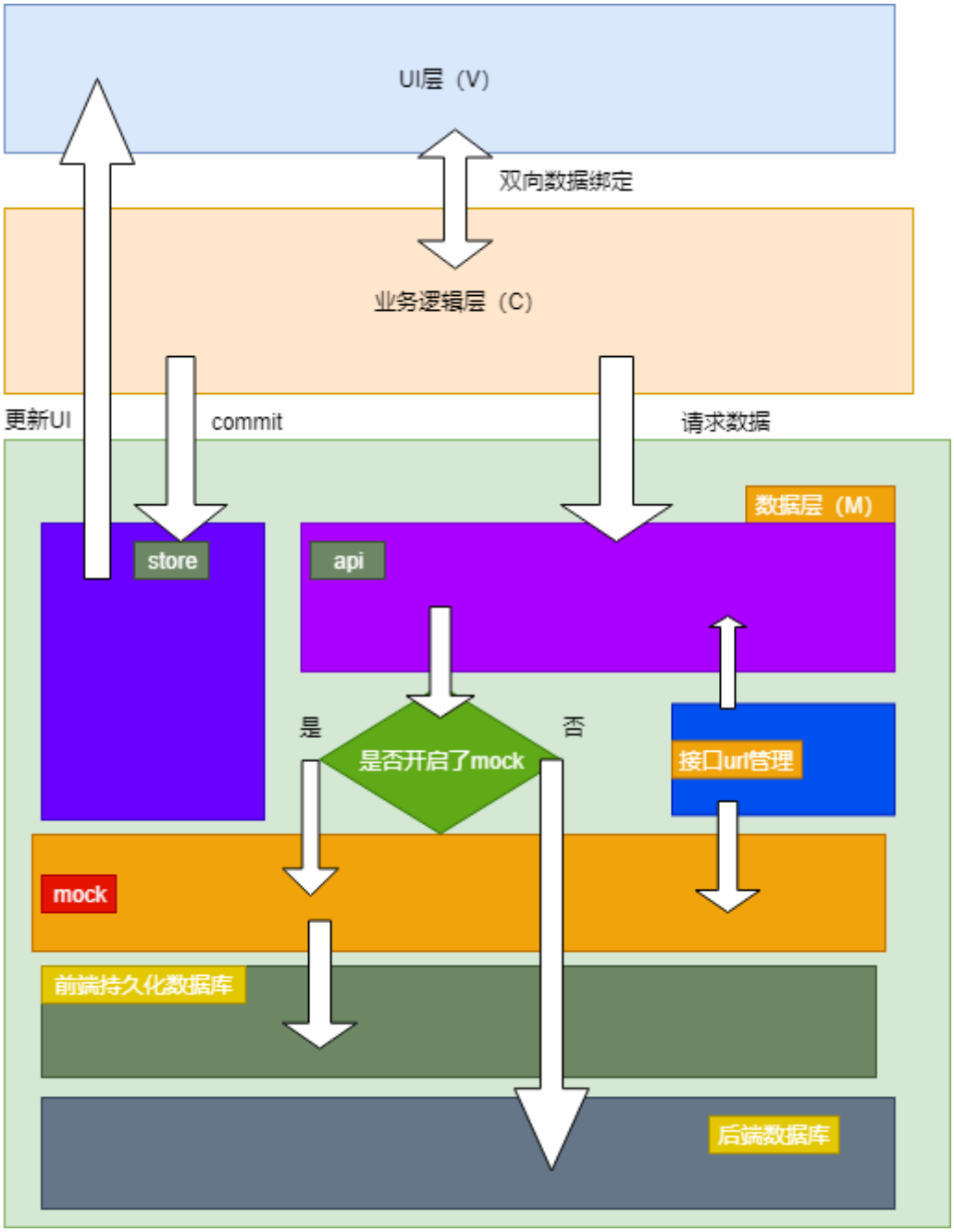
最后，在 `main.js` 中加入一键 `mock` 的开关：

```
import { mockXHR } from '../mock';
if (process.env.NODE_ENV !== 'production') {
  mockXHR();
}
```

这样的话， `main.js` 会根据当前环境自动决定是否开启MOCK模块。

总结

经过这么一系列的设计，整个应用的数据流向结构如下：



理论上来说，如果遵照这套流程，那在后端严格按照接口约定写接口返回数据的情况下，前端独立开发完成后可独立测试，等后端接口单元测试全部通过后，我们只需要修改`url.js`中的请求地址后，就可以得到一个完全符合业务需求的基本可用的系统了。