# 前后端分离

## 什么是前后端分离？

前后端分离：

        前端：整个页面显示以及页面的交互逻辑。用ajax和node作为交互。其中node作为中间层。

        后端：提供api接口，利用redis保存session，与数据库交互。

流程：

        客户端（浏览器）向node请求页面交互。

        node向后端（这里用java）转发请求。java在发送请求到数据库。

        java返回结果给node。node返回页面，提供数据。

前后端分离前端：

        前端可以不依赖后端，前端自己起一个服务，如果前端页面处理好，后端的API还没有提供，那么，前端可以用mock模拟数据。

这里出现一个问题，如果没有服务，单页面是不可以ajax请求数据，会出现跨域的问题。要么浏览器做跨域。所以这里用到node。

我们利用node的express开启一个本地服务器，利用node的 http-proxy-middleware 插件将客户端发送给node的请求发给真正后服务器。

node：

        node主要是为了分层开发，前端不需要知道后端是怎么提供数据，怎么操作。后端也不需要知道node是怎么操作，前端是怎么部署。前端可以利用node自己作处理。

        node本身有着异步,非阻塞I／o。在处理并发量比较大的数据请求上有很大的优势。

## 为什么要前后端分离？

这不是一篇纯技术文章，而是一篇分享我个人在前后端分离路上收获的点点滴滴的文章，以此来为准备尝试前后端分离或者想了解前后端分离的童鞋做一个大体的讲解。

尝试与改变

如果你没有尝试过前后端分离的工作流程，那么可以先试想一下这样的流程改变：

把流程从   
PM：“我要这个功能”  
后端：“这个先找前端做个模板”  
前端：“模板做完了”  
后端：“我来对接一下，这里样式不对”  
前端：“我改完了”  
后端：“功能交付”  
PM：“春节要加这个活动”  
后端：“这个先找前端改个模板”  
前端：“模板做完了”  
后端：“我来对接一下，这里样式不对”  
前端：“我改完了”  
后端：“功能交付”

变成  
PM：“我要这个功能”  
前端：“我要接口”  
后端：“接口完成了”  
前端：“我来对接一下，功能交付”  
PM：“春节要加这个活动”  
前端：“需要增加接口”  
后端：“接口完成了”  
前端：“我来对接一下，功能交付”

由此可见，前后端分离的主要概念就是：后台只需提供API接口，前端调用AJAX实现数据呈现。

现状与分歧

作为一名前端开发人员，我们应该尝试一些新颖的技术，完善每一个细节性的问题，不断突破自我。虽然前后端分离已经算不上什么新颖的技术或思路，但是目前很多后台开发人员甚至前端开发人员都没有接触过。

据我个人的了解，如果在一个部门里，部门人员全是后台开发人员，前端的一些页面也是由后台人员完成的，那么前后端分离对于他们而言可能是一片未知的领域，项目大多是前后端强耦合的，甚至不存在前端的概念。

在不重视前端的公司或部门，不了解前后端分离这也无可厚非。在我刚进入一个全是后台开发人员的部门的时候，整个部门就我一个前端，我刚开始的主要职责就是负责项目前端页面的制作和JS功能的实现，虽然部门有前后端分离的意识，但都不知该如何去实践。在那时，部门的后台人员认为前后端分离就是后台不再需要写HTML和JS了，可以交给前端来做了，然而这只能叫做前后端分工。

以上讲述的是一种情况: 不了解前后端分离，也不知如何去实践的。下面还有一种情况：了解前后端分离，但不想去尝试的。

针对第二种情况，很多人也做过相应的解释，其实这就涉及到“前后端分离的利弊”问题。很多后台人员会认为自己所做的那一套没有问题，即便后台套用前端html也是司空见惯，一直是大势所趋，后台MVC框架也是这么推荐使用的，很合理。这时候前端开发人员在部门中的话语权往往是不够的，或者认为后台开发人员的意见永远是对的，没有主观性。

相反，也有可能是后台开发人员非常推荐前后端分离，而前端开发人员不想去实践的。这时候前端会认为后台开发人员在瞎折腾，之前前后端不分离项目做起来都很顺利，分离了反而会给自己带来额外的工作量和学习成本，而这就取决于前端的技术能力和见识了。

当然，这也是我个人认为的前后端分离所存在的一些现状和分歧所在。

场景与要求

对于前后端分离的应用场景，不是所有的场景都适合，但是大多数项目都能够通过前后端分离来实现。

由于我主要从事企业级后台应用的前端开发工作，个人认为对于后台应用的开发来说，前后端分离带来的利是远大于弊的。

大多数后台应用我们都可以做成SPA应用（单页应用），而单页应用最主要的特点就是局部刷新，这通过前端控制路由调用AJAX，后台提供接口便可以实现，而且这样的方式用户体验更加友好，网页加载更加快速，开发和维护成本也降低了不少，效率明显提升。

同样的，在展示类网站和移动APP页面中前后端分离也同样试用。前后端不分离的情况下，服务端要单独针对Web端做处理，返回完整HTML，这样势必增加服务端的复杂度，可维护性差，而web端需要加载完整的HTML，一定程度上影响网页性能，这对于移动端性能为王的地方非常的不友好。

随着前端技术的发展和迭代，前端MVC框架应运而生，利用目前主流的前端框架，如React、Vue、Angular等我们可以轻松的构建起一个无需服务器端渲染就可以展示的网站，同时这类框架都提供了前端路由功能，后台可以不再控制路由的跳转，将原本属于前端的业务逻辑全部丢给前端，这样前后端分离可以说是最为彻底。下面是一段前端控制路由的代码：

[IMG_256](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

'use strict'

export default function (router) {

router.map({

'/': {

component: function (resolve) {

require(['./PC.vue'], resolve)

}

},

'/m/:params': {

component: function (resolve) {

require(['./Mobile.vue'], resolve)

}

},

'/p': {

component: function (resolve) {

require(['./PC.vue'], resolve)

},

subRoutes: {

'/process/:username': {

component: function (resolve) {

require(['./components/Process.vue'], resolve)

}

}

}

}

})

}

[IMG_257](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

前后端分离的实现对技术人员尤其是前端人员的要求会上升一个层次，前端的工作不只是切页面写模板或是处理一些简单的js逻辑，前端需要处理服务器返回的各种数据格式，还需要掌握一系列的数据处理逻辑、MVC思想和各种主流框架。

优势与意义

对于前后端分离的意义我们也可以看做是前端渲染的意义，我主要总结了下面四点：

1.彻底解放前端

前端不再需要向后台提供模板或是后台在前端html中嵌入后台代码，如：

[IMG_258](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

<!--服务器端渲染 -->

<select>

<option value=''>--请选择所属业务--</option>

{% for p in p\_list %}

<option value="{{ p }}">{{ p }}</option>

{% endfor %}</select>

[IMG_259](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

这是前后端耦合的，可读性差。

[IMG_260](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

<!--前端渲染 -->

<template>

<select id="rander">

<option value=''>--请选择所属业务--</option>

<option v-for="list in lists" :value="list" v-text="list"></option>

</select>

</template>

<script>

export default {

data: {

return {

lists: ['选项一', '选项二', '选项三', '选项四']

}

}，

ready: function () {

this.$http({

url: '/demo/',

method: 'POST',

})

.then(function (response) {

this.lists = response.data.lists // 获取服务器端数据并渲染 })

}

}</script>

[IMG_261](https://www.cnblogs.com/luozhihao/p/javascript:void(0);)

上面是前端渲染的一段代码，前端通过AJAX调用后台接口，数据逻辑放在前端，由前端维护。

2.提高工作效率，分工更加明确

前后端分离的工作流程可以使前端只关注前端的事，后台只关心后台的活，两者开发可以同时进行，在后台还没有时间提供接口的时候，前端可以先将数据写死或者调用本地的json文件即可，页面的增加和路由的修改也不必再去麻烦后台，开发更加灵活。

3.局部性能提升

通过前端路由的配置，我们可以实现页面的按需加载，无需一开始加载首页便加载网站的所有的资源，服务器也不再需要解析前端页面，在页面交互及用户体验上有所提升。

4.降低维护成本

通过目前主流的前端MVC框架，我们可以非常快速的定位及发现问题的所在，客户端的问题不再需要后台人员参与及调试，代码重构及可维护性增强。

心得与体会

一路走来，项目一个接着一个，从一开始的后台控制路由、后台渲染页面到现在的前端控制路由、前端渲染数据，工作流程和方式都发生了很大的变化。每当遇到下面情形的时候，我都会为前后端分离带来的优势而感慨一番：

项目一开始制作前端页面的时候，我不再需要后台给我配置服务器环境了

项目的前端文件可以在需要调用后台接口的时候丢进服务器就好了，完全不需要事先放进去

增加一个项目页面需要配置路由的时候不再需要让后台同事给我加了，自己前端搞定

前端文件里不再掺杂后台的代码逻辑了，看起来舒服多了

页面跳转比之前更加流畅了，局部渲染局部加载非常快速

页面模板可以重复使用了，前端组件化开发提高了开发效率

等等。面对快速发展的前端，我们应该去适应其带来的工作方式和流程的改变，目前的前后端分离的工作方式必然是今后的趋势所在，作为一个前端开发人员，我们应当承担这个普及前端新知识和改变现状的职责。

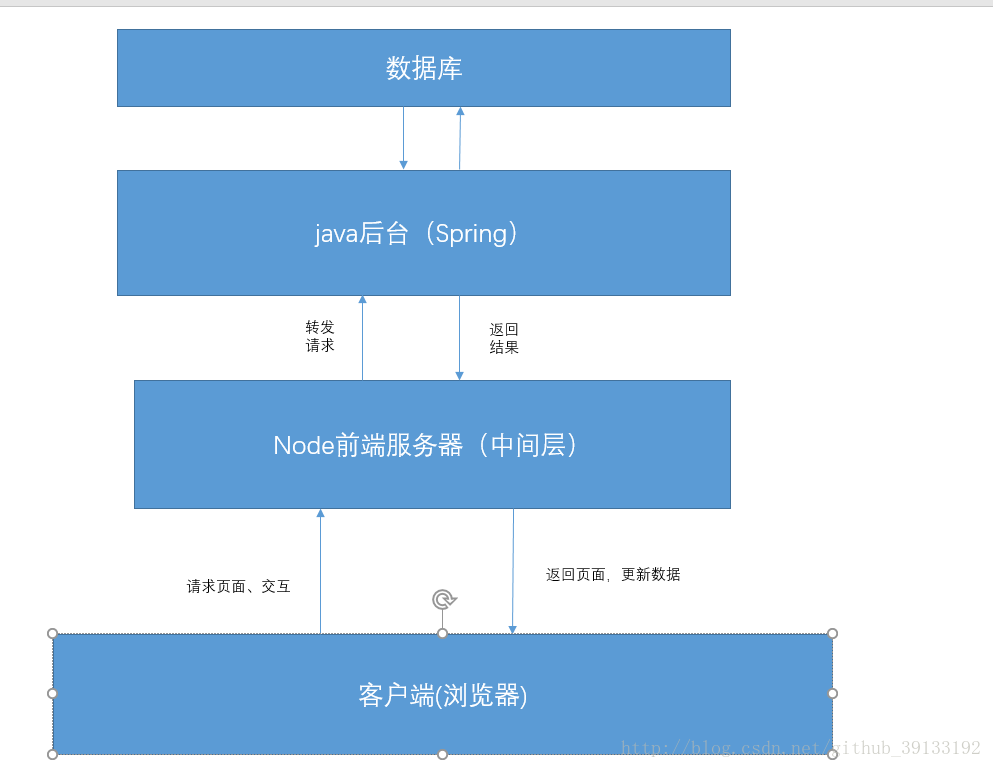
## 如何做到前后端分离

(以下的内容都是基于我们的电影购票网站来讨论的)  
前端的技术框架是: vue全家桶+nodejs+express(实现的是单页面(SPA)应用)  
首先，先分清楚前后端的工作

前端的工作：实现整一个前端页面以及交互逻辑，以及利用ajax与nodejs服务器（中间层)交互

后端的工作：提供API接口，利用redis来管理session,与数据库交互

我们项目的整一个架构如下:



接下来进入正题，如何实现前后端分离

一般来说，要实现前后端分离，前端就需要开启一个本地的服务器来运行自己的前端代码，以此来模拟真实的线上环境，并且，也是为了更好的开发。因为你在实际开发中，你不可能要求每一个前端都去搭建一个java(php)环境，并且在java环境下开发，这对于前端来说，学习成本太高了。但如果本地没有开启服务器的话，不仅无法模拟线上的环境，而且还面临到了跨域的问题，因为你如果写静态的html页面，直接在文件目录下打开的话，你是无法发出ajax请求的(浏览器跨域的限制),因此，你需要在本地运行一个服务器，可是又不想搭建陌生而庞大的java环境，怎么办法呢？nodejs正好解决了这个问题。在我们项目中，我们利用nodejs的express框架来开启一个本地的服务器，然后利用nodejs的一个http-proxy-middleware插件将客户端发往nodejs的请求转发给真正的服务器，让nodejs作为一个中间层。这样，前端就可以无忧无虑的开发了

由于前后端分离后，前端和后台同时开发时，就可能遇到前端已经开发好一个页面了，可是却等待后台API接口的情况。比如说A是负责前端，B是负责后台，A可能用了一周做好了基本的结构，并且需要API接口联调后，才能继续开发，而此时B却还没有实现好所需要的接口，这种情况，怎么办呢？在我们这个项目里，我们是通过了mock来提供一些假数据，我们先规定好了API接口，设计出了一套API文档，然后我们就可以通过API文档，利用mock(http://mockjs.com)来返回一些假数据，这样就可以模拟发送API到接受响应的整一个过程，因此前端也不需要依赖于后端开发了，可以独立开发，等到后台的API全部设计完之后，就可以比较快速的联调。

为什么要引入nodejs作为中间层

前面的我发的项目结构图中，已经表明，在这个项目里，我们将nodejs作为中间层，那么，为什么我们要特地引入nodejs呢？直接用java做不就行了吗？

我觉得引入nodejs主要是为了分层开发，职责划分，nodejs作为前端服务器，由前端开发人员负责，前端开发人员不需要知道java后台是如何实现的，也不需要知道API接口是如何实现的，我们只需要关心我们前端的开发工作，并且管理好nodejs前端服务器，而后台开发人员也不需要考虑如何前端是如何部署的，他只需要做好自己擅长的部分，提供好API接口就可以；

nodejs本身有着独特的异步、非阻塞I/O的特点，这也就意味着他特别适合I/O密集型操作，在处理并发量比较大的请求上能力比较强，因此，利用它来充当前端服务器，向客户端提供静态文件以及响应客户端的请求，我觉得这是一个很不错的选择。

前端服务器如何部署

nodejs前端服务器的职责

作为静态文件服务器，当用户访问网站的时候，将index.html以及其引入的js、css、fonts以及图片返回给用户

负责将客户端发来的ajax请求转发给后台服务器

其实前端服务器的部署工作是算比较简单的，具体有以下两个点:

将开发完的前端代码，利用webpack打包成静态压缩文件

在服务器上，利用pm2负载均衡器来执行以下的代码来开启服务器:



# Bootstrap 网格系统

本章节我们将讲解 Bootstrap 的网格系统（Grid System）。

Bootstrap 提供了一套响应式、移动设备优先的流式网格系统，随着屏幕或视口（viewport）尺寸的增加，系统会自动分为最多12列。

## 什么是网格（Grid）？

摘自维基百科：

IMG_256在平面设计中，网格是一种由一系列用于组织内容的相交的直线（垂直的、水平的）组成的结构（通常是二维的）。它广泛应用于打印设计中的设计布局和内容结构。在网页设计中，它是一种用于快速创建一致的布局和有效地使用 HTML 和 CSS 的方法。

简单地说，网页设计中的网格用于组织内容，让网站易于浏览，并降低用户端的负载。

## 什么是 Bootstrap 网格系统（Grid System）？

Bootstrap 官方文档中有关网格系统的描述：

IMG_257Bootstrap 包含了一个响应式的、移动设备优先的、不固定的网格系统，可以随着设备或视口大小的增加而适当地扩展到 12 列。它包含了用于简单的布局选项的预定义类，也包含了用于生成更多语义布局的功能强大的混合类。

让我们来理解一下上面的语句。Bootstrap 3 是移动设备优先的，在这个意义上，Bootstrap 代码从小屏幕设备（比如移动设备、平板电脑）开始，然后扩展到大屏幕设备（比如笔记本电脑、台式电脑）上的组件和网格。

## 移动设备优先策略

* **内容**
  + 决定什么是最重要的。
* **布局**
  + 优先设计更小的宽度。
  + 基础的 CSS 是移动设备优先，媒体查询是针对于平板电脑、台式电脑。
* **渐进增强**
  + 随着屏幕大小的增加而添加元素。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | | | | 4 | | | | 4 | | | |
| 4 | | | | 8 | | | | | | | |
| 6 | | | | | | 6 | | | | | |
| 12 | | | | | | | | | | | |

响应式网格系统随着屏幕或视口（viewport）尺寸的增加，系统会自动分为最多12列。

## Bootstrap 网格系统（Grid System）的工作原理

网格系统通过一系列包含内容的行和列来创建页面布局。下面列出了 Bootstrap 网格系统是如何工作的：

* 行必须放置在 **.container** class 内，以便获得适当的对齐（alignment）和内边距（padding）。
* 使用行来创建列的水平组。
* 内容应该放置在列内，且唯有列可以是行的直接子元素。
* 预定义的网格类，比如 **.row** 和 **.col-xs-4**，可用于快速创建网格布局。LESS 混合类可用于更多语义布局。
* 列通过内边距（padding）来创建列内容之间的间隙。该内边距是通过 **.rows** 上的外边距（margin）取负，表示第一列和最后一列的行偏移。
* 网格系统是通过指定您想要横跨的十二个可用的列来创建的。例如，要创建三个相等的列，则使用三个 **.col-xs-4**。

## 媒体查询

媒体查询是非常别致的"有条件的 CSS 规则"。它只适用于一些基于某些规定条件的 CSS。如果满足那些条件，则应用相应的样式。

Bootstrap 中的媒体查询允许您基于视口大小移动、显示并隐藏内容。下面的媒体查询在 LESS 文件中使用，用来创建 Bootstrap 网格系统中的关键的分界点阈值。

/\* 超小设备（手机，小于 768px） \*//\* Bootstrap 中默认情况下没有媒体查询 \*/

/\* 小型设备（平板电脑，768px 起） \*/@media (min-width: @screen-sm-min) { ... }

/\* 中型设备（台式电脑，992px 起） \*/@media (min-width: @screen-md-min) { ... }

/\* 大型设备（大台式电脑，1200px 起） \*/@media (min-width: @screen-lg-min) { ... }

我们有时候也会在媒体查询代码中包含 **max-width**，从而将 CSS 的影响限制在更小范围的屏幕大小之内。

@media (max-width: @screen-xs-max) { ... }@media (min-width: @screen-sm-min) and (max-width: @screen-sm-max) { ... }@media (min-width: @screen-md-min) and (max-width: @screen-md-max) { ... }@media (min-width: @screen-lg-min) { ... }

媒体查询有两个部分，先是一个设备规范，然后是一个大小规则。在上面的案例中，设置了下列的规则：

让我们来看下面这行代码：

@media (min-width: @screen-sm-min) and (max-width: @screen-sm-max) { ... }

对于所有带有 *min-width: @screen-sm-min* 的设备，如果屏幕的宽度小于 *@screen-sm-max*，则会进行一些处理。

## 网格选项

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **超小设备手机（<768px）** | **小型设备平板电脑（≥768px）** | **中型设备台式电脑（≥992px）** | **大型设备台式电脑（≥1200px）** |
| 网格行为 | 一直是水平的 | 以折叠开始，断点以上是水平的 | 以折叠开始，断点以上是水平的 | 以折叠开始，断点以上是水平的 |
| 最大容器宽度 | None (auto) | 750px | 970px | 1170px |
| Class 前缀 | **.col-xs-** | **.col-sm-** | **.col-md-** | **.col-lg-** |
| 列数量和 | 12 | 12 | 12 | 12 |
| 最大列宽 | Auto | 60px | 78px | 95px |
| 间隙宽度 | 30px （一个列的每边分别 15px） | 30px （一个列的每边分别 15px） | 30px （一个列的每边分别 15px） | 30px （一个列的每边分别 15px） |
| 可嵌套 | Yes | Yes | Yes | Yes |
| 偏移量 | Yes | Yes | Yes | Yes |
| 列排序 | Yes | Yes | Yes | Yes |

下表总结了 Bootstrap 网格系统如何跨多个设备工作：

### 基本的网格结构

下面是 Bootstrap 网格的基本结构：

<div class="container">

<div class="row">

<div class="col-\*-\*"></div>

<div class="col-\*-\*"></div>

</div>

<div class="row">...</div></div><div class="container">....