Table of Contents

1.1库(lib)的特点

- 是针对特定问题的解答，具有专业性
- 不控制应用的流程
- 被动的被调用

例如：当年最流行的 ***jQuery*** 这货是用来做HTML元素选择操作、css和动画、事件绑定、ajax封装等所有网页基本业务的，其中很多设计特点，和方法名称，都被业内完全认可的，跟jQuery类似的库有很多：prototype，mootools，国内也有很多公司做了自己的类似的库。

模块化的库：最流行的当属requireJs和seaJs

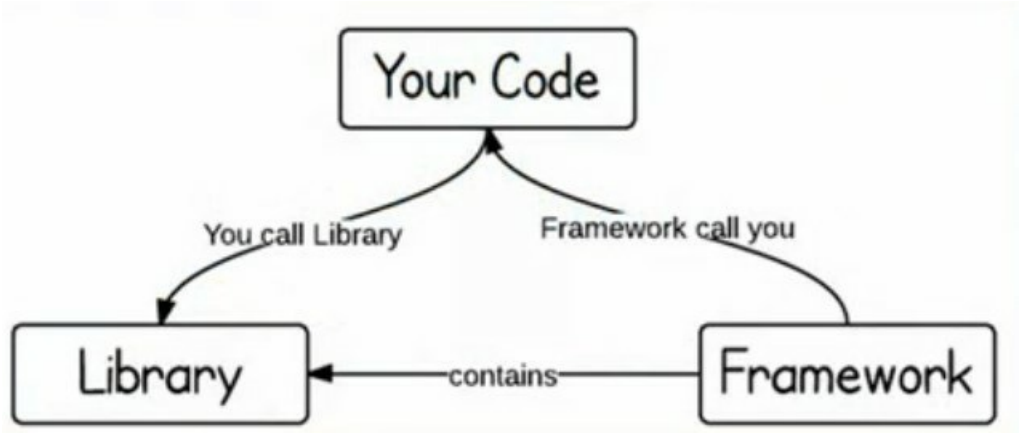
图表制作库：矢量图库highcharts，百度的canvas图库echarts等

js模板引擎:js进行复杂的HTML拼装的时候。可能需要使用js模板引擎，这样的库也是数不胜数，handlebars，Mustache，jade等

1.2框架(frameword) 的特点

- 具有控制反转**(inverse of control)**的功能
- 决定应用程序的生命周期
- 一般来说，集成了大量的库
- 决定应用程序的生命周期

由下图所示，框架会在特定的时间要求程序执行某段代码。框架决定了什么时候调用库，决定了什么时候要求代码去执行特定功能



而实际上，一个库有时也可以称之为框架，而框架里面集成的方法称之为库 框架和库的区别不由实际大小决定，而由思考角度来决定。框架和库实际上可以统称为解决方案

前端面临的问题：

前端开发中的解决方案主要用于解决以下7个方面的问题：

- **1**、**DOM**
- **2**、**Communication(**通信**)**
- **3**、**Utililty(**工具库**)**
- **4**、**Templating(**模板集成**)**
- **5**、**Component(**组件**)**
- **6**、**Routing(**路由**)**
- **7**、**Architecture(**架构**)**

为什么要使用外部解决方案：

- **1**、提高开发效率
- **2**、可靠性高（浏览器兼容，测试覆盖）
- **3**、配备优良的配套，如文档、**DEMO**及工具等
- **4**、代码设计的更合理、更优雅
- **5**、专业性高

如果问题过于简单，或者备选框架的质量和可靠性无法保证，再或者无法满足业务需求，则不应该选择外部的框架。如果团队中已经有相关的积累，就更不需要使用了

如何选择解决方案：

一般地，解决方案要实际开发中有以下3种使用方式：

- **1**、开放式：基于外部模块系统，并自由组合
- **2**、半开放式：基于一个定制的模块系统，内部外部解决方案共存
- **3**、封闭式：深度定制的模块系统，很少需要引入外部模块

DOM 解决方案

关于DOM，主要包括Selector(选择器)、Manipulation(DOM操作)、Event(事件)、Animation(动画)这四个部分

DOM相关的解决方案主要用于提供以下操作 1、提供便利的 DOM 查询、操作、移动等操作 2、提供事件绑定及事件代理支持 3、提供浏览器特性检测及 UserAgent 侦测 4、提供节点属性、样式、类名的操作 5、保证目标平台的跨浏览器支持

2.1.1 DOM 常用方案

常用的DOM解决方案有 jQuery、zepto.JS、MOOTOO.JS等 jQuery是曾经风靡一时的最流行的前端解决方案，jQuery特有的链式调用的方式简化了javascript的复杂操作，而且使人们不再需要关心兼容性，并提供了大量的实用方法 zepto是jQuery的精简版，针对移动端去除了大量jQuery的兼容代码，提供了简单的手势，部分API的实现方式不同 mootools源码清晰易懂，严格遵循Command-Query(命令-查询)的接口规范，没有诸如jQuery的两义性接口。还有一个不得不提的特点是，使用选择器获取的是DOM原生对象，而不是被包装过的对象。而它支持的诸多方法则是通过直接扩展DOM原生对象实现的，这也是它的争议所在 相比较而言，最稳妥的DOM解决方案是jQuery

2.1.2 专业领域

上面的解决方案用于解决DOM一般的通用问题。随着技术的发展，DOM的专业领域出现一些小而精致的解决方案 1、手势 Hammer.JS包括了常见手势封装（Tab、Hold、Transform、Swifp）并支持自定义扩展 2、局部滚动 iscroll.JS是移动端position:fix + overflow:scroll的救星 3、高级动画 Velocity.JS可以复杂动画序列实现，不仅局限于 DOM 4、视频播放 Video.JS类似原生 video 标签的使用方式，对低级浏览器使用 flash 播放器

常用通信方式

关于通信，主要包括XMLHttpRequest、Form、JSONP、Socket等 通信相关的解决方案主要用于提供以下操作

1、处理与服务器的请求与相应

2、预处理请求数据与响应数据 Error/Success 的判断封装

3、多类型请求，统一接口（XMLHttpRequest1/2、JSONP、iFrame）

4、处理浏览器兼容性

2.2.1 常用方案

除了jQuery等，其他常用的通信解决方案有Reqwest、qwest等 Reqwest支持JSONP，稳定性高，IE6+支持，CORS 跨域； Promise/A 支持 异步，跨域，有非常好用的aioxs库等； qwest代码少、支持XMLHttpRequest2、CORS 跨域、支持高级数据类型（ArrayBuffer、Blob、FormData）```

2.2.2 专业领域

对于实时性要求较高的需求可以使用websocket , socket.io，它实时性高，支持二进制数据流，智能自动回退支持，且支持多种后端语言

工具包

工具包(Utililty)的主要职责包括以下：

1、提供 JavaScript 原生不提供的功能

2、包装原生方法，使其便于使用

3、异步队列及流程控制

常用的工具包解决方案

有es5-shim、es6-shim、underscore、Lodash等；

上面提到的shim，也是经常听到的一个词，翻译过来是垫片的意思。对于es5、es6等 标准包括的一些新方法，由于浏览器兼容性不高，所以无法直接使用它们。这时，就需要在保证实现与规范一致的基础上，来扩展原型方法，这种做法就叫做shim。好处在于，实际上就是在使用javascript的语法，但不用去考虑低版本浏览器的兼容性问题 es5-shim 提供 ES3 环境下的 ES5 支持 es6-shim 提供 ES5 环境下的 ES6支持 underscore 提供兼容 IE6+ 的扩展功能函数 Lodash是underscore 的高性能版本，方法多为 runtime 编译出来的

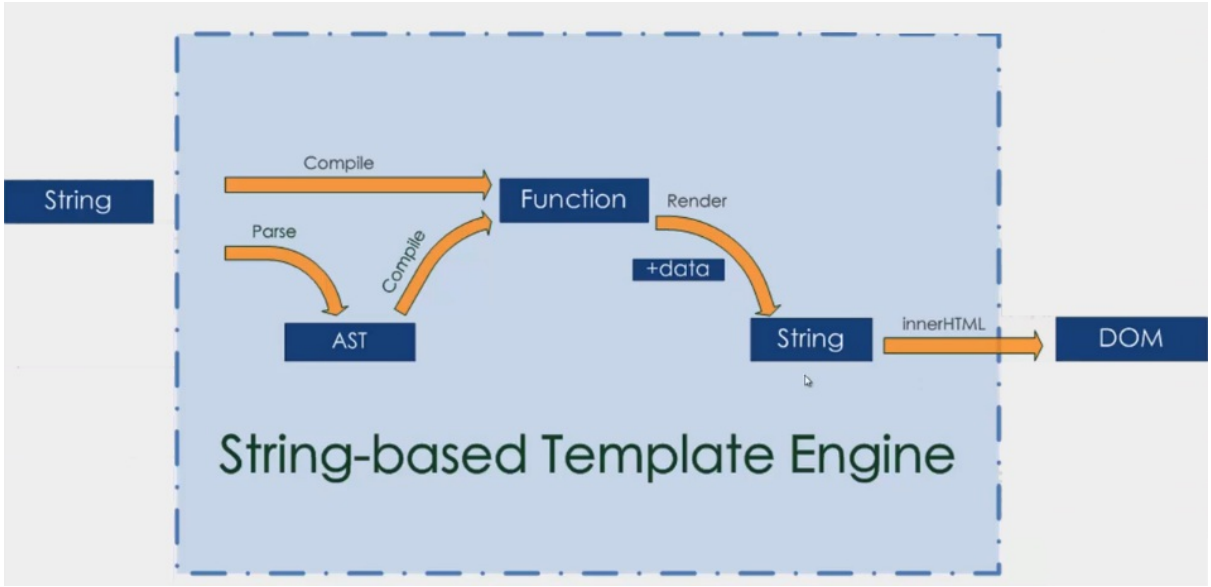
模板

模板主要包括三类：基于字符串的模板(String-based)、基于DOM的模板(DOM-based)、活动模板(Living Template)

2.4.1 基于字符串的模板(String-based)

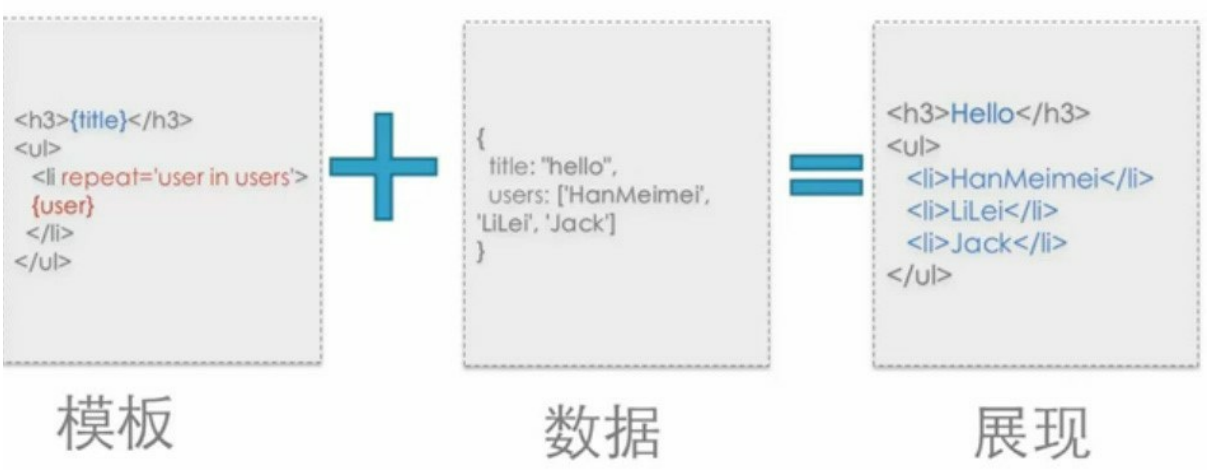
1、基于字符串的模板(String-based)，解决方案包括(dustjs、hogan.js、dot.js)

原理如下：输入一段模板字符串，通过编译之后，生成一段Function，通过Function的render或类render函数渲染输入的数据data，输出模板字符串，字符串通过innerHTML或类似的方式渲染成最后的DOM结构。这类模板的问题在于通过字符串生成DOM之后就不再变化，如果在改变输入的数据data，需要重新render，重新生成一个全新的DOM结构，性能较差。但该模板可以在服务器端运行

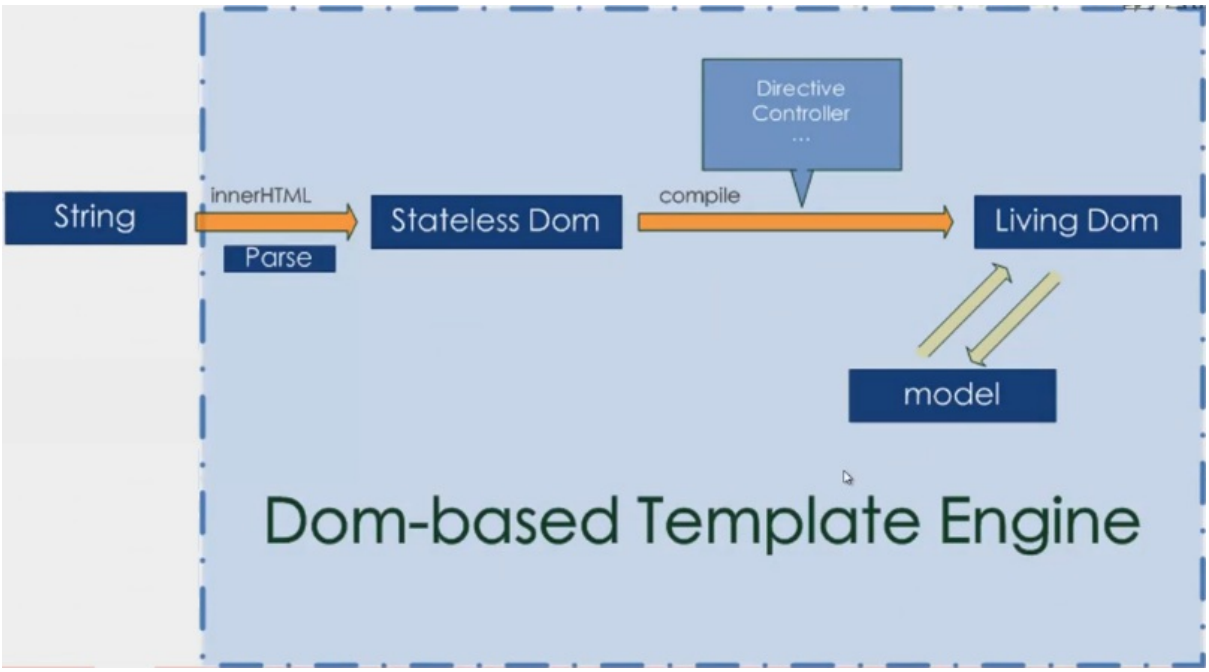


2.4.2 基于DOM的模板(DOM-based)

基于DOM的模板(DOM-based)，解决方案包括(angularjs、vuejs、knockout)

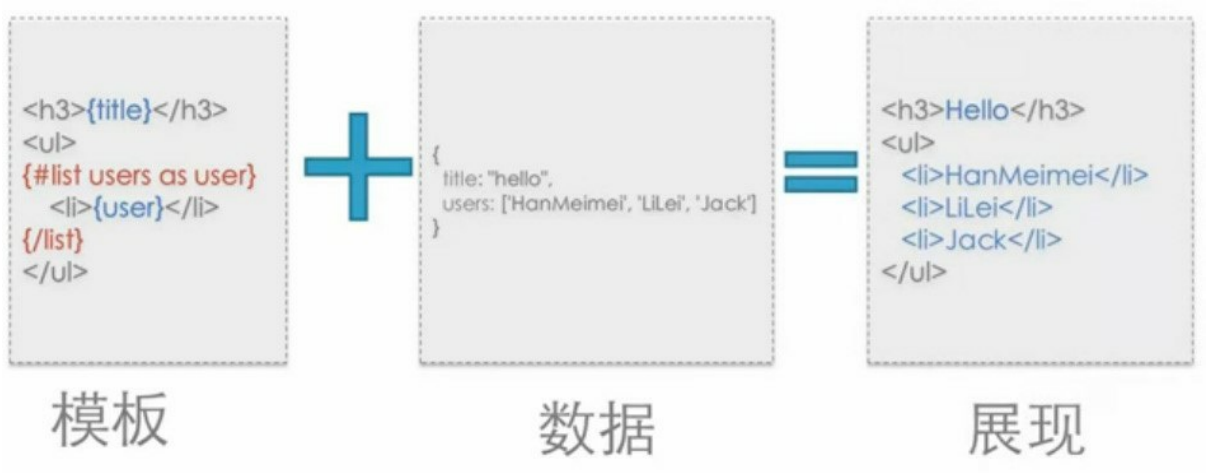


原理如下：将输入的字符串模板通过innerHTML转换为一个无状态DOM树，然后遍历该节点树，去抓取关键属性或语句，来进行相关的绑定，进而变成了有状态的DOM树，最终导致DOM树会与数据模型model进行绑定。这类模板的特点是修改数据时，会使有状态的DOM树实时更新，运行时性能更好，也会保留 DOM 中的已有事件

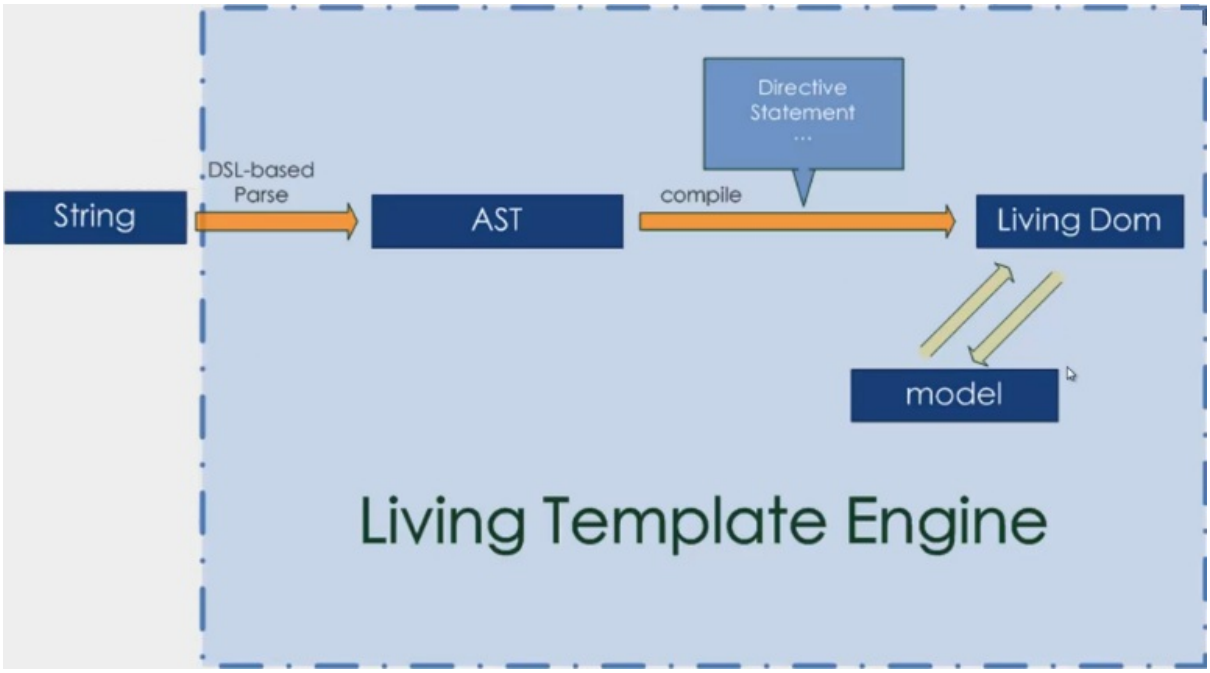


2.4.3 基于DOM的模板(DOM-based)

活动模板(Living Template)，解决方案包括(RegularJS、RactiveJS、htmlbar)



原理如下：活动模板融合了字符串模板和DOM模板的技术，模板字符串string通过自定义的解析器DSL-based Parse解析成AST(抽象语法树)，通过遍历AST，使用createElement()、setAttribute()等原生DOM方法，生成DOM树，最终导致DOM树会与数据模型model进行绑定。由于其内部完全不使用innerHTML，所以安全性较高



组件化

一、前端为什么要做组件化

在大型软件系统中，web应用的前后端已经实现了分离，而随着REST软件架构的发展，后端服务逐步倾向于微服务，简单来说就是将一个大型后端服务，拆分成多个小服务，它们分别部署，降低了开发的复杂性，而且提高了系统的可伸缩性。而前端方面，随着技术的发展，开发的复杂度也越来越高，传统开发模式总是存在着开发效率低，维护成本高等的弊端。

传统开发方式效率低以及维护成本高的主要原因在于很多时候是将一个系统做成了整块应用，而且往往随着业务的增长或者变更，系统的复杂度会呈现指数级的增长，经常出现的情况就是一个小小的改动或者一个小功能的增加可能会引起整体逻辑的修改，造成牵一发而动全身。

针对此弊端，其实业界早就有了一些探索，我们希望一个大且复杂的场景能够被分解成几个小的部分，这些小的部分彼此之间互不干扰，可以单独开发，单独维护，而且他们之间可以随意的进行组合。就拿电脑主机来说，一台整机包括CPU，主板，内存，硬盘等等，而这些部件其实都是由不同的公司进行生产的，他们彼此之间根据一套标准分别生产，最后组装在一起。当某个部件出现问题时，不需要将整台主机都进行维修，只需要将坏的部件拿下来，维修之后再将其组合上就可以了。这种化繁为简的思想在后端开发中的体现是微服务，而在前端开发中的体现就是组件化。

随着React，angular等以组件（指令等）为主的优秀前端框架的出现，前端组件化逐渐成为前端开发的迫切需求，当然这一迫切需求也逐渐成为一种主流，一种共识，它不仅提高了前端的开发效率，同时也降低了维护成本。开发者们不需要再面对一堆复杂且难阅读的代码，转而只需要关注以组件方式存在的代码片段。

那么前端组件化开发都经历了哪些阶段呢？

二、 前端组件化开发发展之路

1、交互少的静态页面时期：公共模块和CSS

这是一个很古老的时代，那时的前端页面就是一些基本的HTML标签以及JS和CSS，页面上大部分都是一些静态的文字，就在这个时期，前端JS和CSS已经出现了组件化，或许更多的应该成为模块化，即开发者把不同模块的或者公共的JS和CSS放在不同的文件中，然后在页面引入并使用，这种方式也沿用至今。

2、繁琐的早期动态页面时期：动态引入

由于静态页面不能在页面上存储数据，阅读者也不满足于基本的页面交互，更希望页面能够活起来，且能够把交互的数据存储起来，于是出现了很多服务端技术，比如ASP，JSP。这些技术的出现使得前端页面活起来了，用户可以根据自己的需求进行数据的交互。

然而这时的页面上充斥着业务逻辑，随着业务逻辑的增多，页面的内容也越来越多，越来越复杂。在这个时期前端组件化开发得到了一定的发展，开发者已经不满足于简单的将JS和CSS文件模块化，开始把一些公用的页面逻辑独立开来，然后通过页面动态引入的方式进行使用，比如公共的页面头(header)和尾(footer)以及数据库的连接(DatabaseConn.jsp)等。

3、后端为主的MVC时期：Tag标签

由于早期动态页面时期的业务逻辑都写在页面上，随着逻辑的增多，页面越来越复杂，维护起来也越来越难。于是以servlet为代表的MVC时代逐渐登上历史舞台，这时页面上的逻辑都被转入到servlet中，使得View层的表现更加简洁，也更加的易于阅读，从而达到了开发的分层。

而随着Struts以及Spring的出现，MVC的开发方式达到鼎盛时期，前端View层的展现也变得越来越简单，没有了复杂的业务逻辑，前端的组件化方式主要是taglib标签，比如jsp标签，Struts标签等，把HTML代码和业务逻辑打包成一个标签，然后使用者直接放置在想要的地方，就可以了。但这个时期，整个WEB应用的开发轻前端重后端，那些taglib标签也都是JAVA代码编写的。

4、前端Ajax时期：JS大行其道

由于MVC时期的轻前端重后端的思想，前端页面主要以表格的形式展现，如果想要一些很炫的效果，实现起来就比较复杂了，往往要写一大堆的代码，而且很难阅读。AJAX作为早已经出现的技术在这个时候越来越受到开发者的青睐，于是出现了很多的JS框架，比如JQuery-UI，easy-UI，miniUI以及大名鼎鼎的ExtJS。

这些JS框架的出现使得前端组件化的开发到达了一个新的高度，利用封装Dom，AJAX以及页面交互的方式，一个个的很炫的组件出现了，开发者可以随意的将这些组件应用的页面中，开发变得简单的同时页面也变得越来越好看。由于这些交互都由JS来完成，运行在浏览器端，也大大的减少了服务端的压力，同时也提高了性能。

5、前端MV*时期：自定义组件

随着时间的推移，开发者发现，如果想要修改这些（ExtJS，miniUI）JS框架中的组件是非常困难的，因此开发者希望能够很容易的自定义一些组件。这时以Angular，React为代表的可以自定义组件的JS框架出现了。这些框架的出现不仅可以让开发者自定义组件，而且可以让开发者将已经存在的组件进行封装。

不仅如此，由于有了npm以及bower这些包管理库，开发者可以很容易的将自己开发的组件publish到这些库中，在使用时只要把他们下载下来（比如npm install）就可以直接使用了。比如：

以上的组件化基本以HTML和JS为主，那么CSS怎么做组件化呢？

6、CSS组件化：less和sass

前面讲了CSS的模块化基本上是将实现某一模块Dom样式的CSS放在不同的文件中，显然随着WEB应用的发展，开发者已经很不满足于这种简单的模块化了。 其实关于CSS的组件化，业界也早就已经有了很多探索，比如less，sass等。那么为什么CSS也要组件化呢？

我们知道CSS是一种扁平的结构，一个Dom可能对应着一个CSS样式，而这些CSS样式很有可能出现公共的部分，那么提取这些公共的部分也就实现了CSS的组件化，在诸如less和sass出现之前，开发者都是把公共的CSS样式写成一个个公共class，但是这样之后CSS文件的阅读性就变得困难了，当然也不容易修改。而less和sass出现之后，使得CSS的编程可以定义变量，可以实现继承，CSS内容的结构也变得更加清晰，提高了CSS文件的阅读性，更容易让人理解，修改起来也变得简单。

三、前端组件化的**4**个原则

前面讲了组件化开发的发展过程，那么我们该怎么做组件化呢？我认为组件应该遵守以下几个原则：

- 标准性

任何一个组件都应该遵守一套标准，可以使得不同区域的开发人员据此标准开发出一套标准统一的组件。

- 组合性

组件之前应该是可以组合的。我们知道前端页面的展示都是一些HTML DOM的组合，而组件在最终形态上也可以理解为一个个的HTML片段。那么组成一个完整的界面展示，肯定是要依赖不同组件之间的组合，嵌套以及通信。

- 重用性

任何一个组件应该都是一个可以独立的个体，可以使其应用在不同的场景中。

- 可维护性

任何一个组件应该都具有一套自己的完整的稳定的功能，仅包含自身的，与其它组件无关的逻辑，使其更加的容易理解，使其更加的容易理解，同时大大减少发生bug的几率。

四、总结与实践

当然组件化开发也并不是这么简单就能实践的。

对开发者的能力有一定的要求，比如传统开发方式可能只要求开发者懂HTML，JS，CSS这些就可以了，而组件化开发方式下还可能要求开发者掌握less，sass，或者ES6等的语法，以及webpack，glup等的前端打包以及构建工具。不过另一方面，哪个开发者不希望自己能掌握更多的本领呢？

技术选型，当前前端组件化框架可以说是百家齐放，这就要求技术经理或者架构师具有超前的前瞻性，根据项目需求以及框架的未来发展进行分析选型。

以我们公司目前在做的项目《普元数字化企业云平台》为例，整个前端项目由上海和西安两地的同事共4个人合作开发，在开发之初就确立了要采用一套统一的标准以减少异地开发的沟通成本，以及维护成本，显然组件化开发可以很好的满足此要求。

基于此，我们的前端使用的是Facebook的React技术，React的核心是使用组件定义界面的表现，它突出的就是开发组件化。 如下图所示，界面上的任何表现都对应着组件。组件之间很好的遵守了组件化开发的几个原则，不同区域的同事开发出的组件都如同同一个人写的，大大降低了异地的沟通成本和维护成本，以及提升了开发效率。

前端路由

路由历史 什么是路由？ 路由是根据不同的 url 地址展示不同的内容或页面

早期的路由都是后端直接根据 url 来 reload 页面实现的，即后端控制路由。

后来页面越来越复杂，服务器压力越来越大，随着 ajax（异步刷新技术） 的出现，页面实现非 reload 就能刷新数据，让前端也可以控制 url 自行管理，前端路由由此而生。

单页面应用的实现，就是因为前端路由。

前端路由实现 1.Pjax（PushState + Ajax） 原理：利用ajax请求替代了a标签的默认跳转，然后利用html5中的API修改了url

API： history.pushState 和 history.replaceState

两个 API 都会操作浏览器的历史记录，而不会引起页面的刷新，pushState会增加一条新的历史记录，而replaceState则会替换当前的历史记录。

例子：

```
window.history.pushState(null, null, "name/blue");
//url: https://www.baidu.com/name/blue

window.history.pushState(null, null, "/name/orange");
//url: https://www.baidu.com/name/orange
```

2.Hjax（Hash + Ajax） 原理：url 中常会出现 #，一可以表示锚点（如回到顶部按钮的原理），二是路由里的锚点（hash）。Web 服务并不会解析 hash，也就是说 # 后的内容 Web 服务都会自动忽略，但是 JavaScript 是可以通过 window.location.hash 读取到的，读取到路径加以解析之后就可以响应不同路径的逻辑处理。

hashchange 事件(监听 hash 变化触发的事件)，当用 window.location 处理哈希的改变时不会重新渲染页面，而是当作新页面加到历史记录中，这样我们跳转页面就可以在 hashchange 事件中注册 ajax 从而改变页面内容。

架构

所有的架构**(architecture)**都是一个目的，就是解耦。解耦有很多方式，可以通过事件、分层等 市面上，有很多架构模式，包括MVC、MVVM、MV*等 架构的职责主要包括以下：

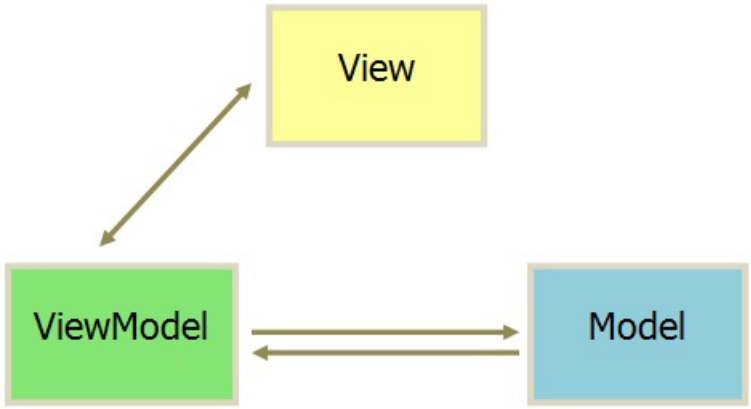
- 1、提供一种范式帮助（强制）开发者进行模块解耦
- 2、视图与模型分离
- 3、容易进行单元测试
- 4、容易实现应用扩展

以MVVM为例，如下图所示。它包括Model(数据层或模型层)、View(视图层)、ViewModel(控制层)

Model(数据层或模型层)表示数据实体，它们用于记录应用程序的数据

View(视图层)用于展示界面，界面是数据定制的反映，它包含样式结构定义以及VM享有的声明式数据以及数据绑定

ViewModel(控制层)是View与Model的粘合，它通过绑定事件与View交互并可以调用Service处理数据持久化，也可以通过数据绑定将Model的变动反映到View中



它们的关系是：各部分之间的通信，都是双向的；View 与 Model 不发生联系，都通过 ViewModel 传递；View 非常薄，不部署任何业务逻辑，称为”被动视图”（Passive View），即没有任何主动性，而ViewModel非常厚，所有逻辑都部署在那里

【SPA】 要特点注意的是，MV *!==* *SPA(*单页系统*)* *SPA*应用程序的逻辑比较复杂，需要一种模式来进行解耦，但并不一定是*MV*模式

核心系统

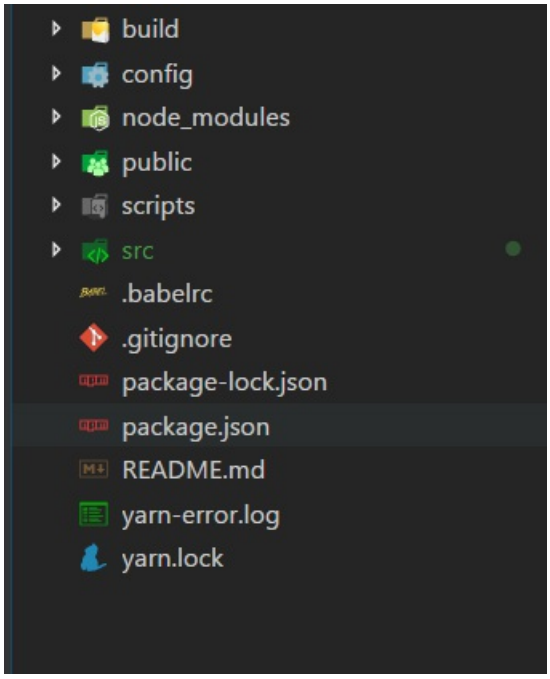
一、环境配置与启动 环境配置：

```
node@6.9.5、npm@5.3.0、vue@^2.4.2、vue-router@^2.2.0、vuex@^2.2.1
```

环境搭建 安装Node 1)、下载node-v6.9.5-x64.msi双击安装(可安装到除c盘外的其它盘) 2)、cmd命令中输入node-v，出现版本号则安装成功 3)、由于国内网络原因安装node之后，请自行安装cnpm 地址：http://npm.taobao.org/

启动 1.从gitlab上获取到和聚宝的git地址 http://git@gitlab.polyhome.net:react/insu-core-sys.git git clone 地址。 2.将代码down下来后 在项目根目录中执行npm i 或者 cnpm i 3.在项目根目录中执行 npm run dev 来启动项目 如果出现端口冲突的错误，你可以在 /config/index.js 文件中找到启动端口。

二、项目结构说明



主要功能：

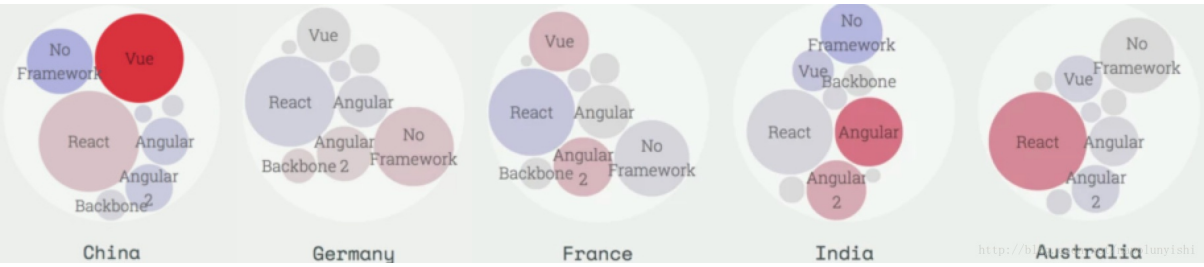
主要分为以下功能模块： 组织管理:用户管理，机构管理，组织人员 系统管理:字典管理，系统参数 权限管理:客户端授权，角色管理，菜单管理 消息推送:APP历史消息，消息推送管理，消息模板管理 系统监控:访问日志，服务器监控，在线用户，ELK状态，服务监控，zipkin状态，定时任务监控，缓存监控 客户列表:客户列表 渠道管理:渠道管理

框架选择

前端**MVC**框架

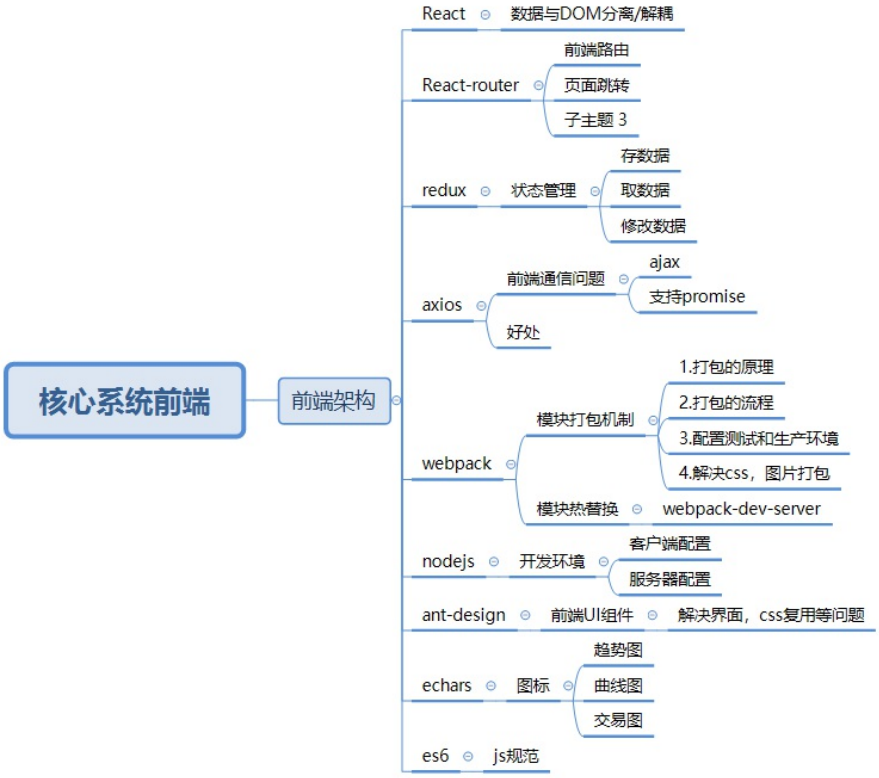
通常前端技术选型最核心的决策点就是所谓的前端MVC框架，网上充斥着这类框架的对比文章，但是孰优孰劣仍然是萝卜白菜，各有所爱，说一句正确的废话就是：框架的选择要因地制宜，根据应用场景和团队实际情况来定。

目前热度最高的三大前端框架是Angular（2.0+）、React和Vue.js。三者都采用了组件化的思想，同一component的html、js、css组织在一起，组件化思想对早期绝对的“关注点分离”则文件分离思想是一定程度的矫正，利于组件的单独开发和复用，减轻了html、js，特别是css的全局污染问题。 Angular是由Google推出的基于TypeScript的MVVM框架，视图和模型双向绑定，通过指令增强模板的表达能力，同时可以自定义组件化的指令，支持依赖注入、注解。由于策略上的问题，Angular2.0不兼容AngularJS 1.0，因此，近两年Angular的活跃度有所下降，然而其内置完备的特性和工具让拥护者们认为它是一个企业级的JS框架。 React由Facebook主推，最显著的特点是一切以JS为中心，HTML和CSS都由JS代码生成，为此，还产生了一种新的语法：JSX，这跟Angular把JS以扩展标签的形式放到HTML中是完全相反的做法。React实现了Virtual DOM，在DOM频繁变化的场景下，性能有不错的表现。另外，React本身主要关注于视图层，并不是一个大而全的框架，由于React认为MVC架构在大规模应用场景下会导致状态的混乱，因此创新地提出了单向数据流（Unidirectional data flow）的概念，出现了状态管理的模式或框架，如Flux/Redux。 Vue.js是三者之中最年轻、最轻量的一个框架，由华裔程序员尤雨溪发起，也是主要关注视图部分，既支持双向绑定也可以单向绑定。Vue.js的模板语法有点类似于Angular，提供了一些内置标签。Vue.js支持单文件组件，也就是将html、js、css写到一个后缀为.vue的文件中，也支持Virtual DOM，性能表现在三者中最好。Vue.js可以使用Redux做状态管理，但也提供了自己的Vuex。 值得一提的是，上面三个主流框架都有对应的移动端方案，比如Angular对应的Ionic，React的React Native以及Vue对应的Weex（阿里开发）。 数据和视图的不同交互方式催生出几个不同的概念：MVC、MVP、MVVM以及单向数据流。 下图是2017年不同国家对几大框架使用情况的调查反馈。 调查反馈。



技术选型

本系统主要采用**react**技术栈全家桶： React,react-redux,react-router,axios,Ant-Design, 利用Node的NPM解决项目依赖包管理问题，webpack解决模块化，构建等问题, Babel 转码器将最新的ES6转为大多数浏览器兼容的ES5.



3.3.1 React

最重要作用：数据与**DOM**分离/解耦，负责**Dom**的渲染工作，

React是Facrbook内部的一个JavaScript类库，已于1年开源，可用于创建Web用户交互界面。它引入了一种新的方式来处理浏览器DOM。那些需要手动更新DOM、费力地记录每一个状态的日子一去不复返了——这种老舅的方式既不具备扩展性，又很难加入新的功能，就算可以，也是有着冒着很大的风险。React使用很新颖的方式解决了这些问题。你只需要声明地定义各个时间点的用户界面，而无序关系在数据变化时，需要更新哪一部分DOM。在任何时间点，React都能以最小的DOM修改来更新整个应用程序。

我们只需要关注数据的变化，数据变化React通过一系列处理能自动更新Dom，这就是它的神奇之处。

React本质上只关心两件事：

1.更新DOM；

2.响应事件。

React不处理Ajax、路由和数据存储，也不规定数据组织的方式。它不是一个Model-View-Controller框架。如果非要问它是什么，他就是MVC里的“V”。React的精简允许你将它集成到各种各样的系统中 。

3.3.2 React渲染机制

React分为react-dom和react的原因是React-Native的出现，它可以实现跨平台实现相同的组件。 react包包含了React.createElement、.createClass、 .Component、 .PropTypes、.Children以及其他的描述元素和组件的类。 react-dom包包含了ReactDOM.render、.unmountComponentAtNode和.findDOMNode等。下面看一个创建组件的实例

```
var React = require('react');
var ReactDOM = require('react-dom');
var MyComponent = React.createClass({
 render: function() {
 return <div>Hello World</div>;
}
});
ReactDOM.render(<MyComponent />, node);
```

ReactDOM.render是**React**的最基本方法用于将模板转为**HTML**语言，并插入指定的**DOM**节点。它可以将一个**React**元素呈现在指定的**DOM container**中，并返回对组件的引用对象。

3.3.3 React 与 JavaScript 渲染模式

每次状态改变时，使用JavaScript重新渲染整个页面会非常慢，这应该归咎于读取和更新DOM的性能问题。React运用一个虚拟的DOM实现了一个非常强大的渲染系统，在React中对DOM只更新不读取。

这些函数读入当前的状态，将其转换为目标页面上的一个虚拟表现。只要React被告知状态有变化，他就会重新运行这些函数，计算出页面的一个新的虚拟表现，接着自动把结果转换成必要的DOM更新来反映新的表现。

这种方式看上去应该比通常的JavaScript方案——按需要更新每一个元素——要慢，但是React确实是这么做的：它使用了非常高效的算法，计算出虚拟页面当前版本和新版间的差异，基于这些差异对DOM进行必要的最少更新。React赢就赢在了最小化了重绘，并且避免了不必要的DOM操作，这两点都是公认的性能瓶颈。

模块热替换(Hot Module Replacement)

HMR是webpack最令人兴奋的特性之一，当你对代码进行修改并保存后，webpack 将对代码重新打包，并将新的模块发送到浏览器端，浏览器通过新的模块替换老的模块，这样在不刷新浏览器的前提下就能够对应用进行更新

其实实现HMR的插件有很多，webpack-dev-server只是其中的一个，当然也是优秀的一个，它能很好的与webpack配合。另外，webpack-dev-server只是用于开发环境的。

全局安装：npm install webpack-dev-server --g (全局安装以后才可以直接在命令行使用webpack-dev-server)

本地安装：npm install webpack-dev-server --save-dev 在webpack的配置文件里添加webpack-dev-server的配置：

```
module.exports = {
        devServer: {
        contentBase: path.resolve(__dirname, 'build'),
    },
}
```

webpack-dev-server实现热更新(HMR)

热更新可以做到在不刷新浏览器的前提下刷新页面，热更新的好处是：

- 保持刷新前的应用状态(这一点在react里是做不到的，具体原因看下面)
- 不浪费时间在等待不必要更新的组件被更新上面
- 调整CSS样式的速度更快

采用非Node模式，添加hot: true，并且一定要指定output.publicPath，建议devServer.publicPath和output.publicPath一样。

webpack.config.js

```
const publicPath = '/';
const buildPath = 'build';

output: {
        path: path.resolve(__dirname, buildPath), //打包文件的输出路径
        filename: 'bundle.js', //打包文件名
        publicPath: publicPath, //重要!
    },
    devtool: 'inline-source-map',
    devServer: {
        publicPath: publicPath,
        contentBase: path.resolve(__dirname, buildPath),
        inline: true,
        hot: true,
    },
```

React-router

React Router 是一个基于 React 之上的强大路由库，它可以让你向应用中快速地添加视图和数据流，同时保持页面与 URL 间的同步。

路由匹配原理

路由拥有三个属性来决定是否“匹配“一个 URL：

- 嵌套关系 和
- 它的 路径语法
- 它的 优先级

嵌套关系 `React Router` 使用路由嵌套的概念来让你定义 `view` 的嵌套集合，当一个给定的 URL 被调用时，整个集合中（命中的部分）都会被渲染。嵌套路由被描述成一种树形结构。`React Router` 会深度优先遍历整个路由配置来寻找一个与给定的 URL 相匹配的路由。

例子：

```
<Router>
  <Route path="/" component={App}>
    <Route path="accounts" component={Accounts}/>
    <Route path="statements" component={Statements}/>
  </Route>
</Router>
```

动态路由

对于大型应用来说，一个首当其冲的问题就是所需加载的 JavaScript 的大小。程序应当只加载当前渲染页所需的 JavaScript。有些开发者将这种方式称之为“代码分拆” —— 将所有的代码分拆成多个小包，在用户浏览过程中按需加载。

对于底层细节的修改不应该需要它上面每一层级都进行修改。举个例子，为一个照片浏览页添加一个路径不应该影响到首页加载的 JavaScript 的大小。也不能因为多个团队共用一个大型的路由配置文件而造成合并时的冲突。

路由是个非常适于做代码分拆的地方：它的责任就是配置好每个 view。

React Router 里的路径匹配以及组件加载都是异步完成的，不仅允许你延迟加载组件，并且可以延迟加载路由配置。在首次加载包中你只需要有一个路径定义，路由会自动解析剩下的路径。

Route 可以定义 getChildRoutes，getIndexRoute 和 getComponents 这几个函数。它们都是异步执行，并且只有在需要时才被调用。我们将这种方式称之为 “逐渐匹配”。 React Router 会逐渐的匹配 URL 并只加载该 URL 对应页面所需的路径配置和组件。

如果配合 webpack 这类的代码分拆工具使用的话，一个原本繁琐的构架就会变得更简洁明了。

```
const CourseRoute = {
  path: 'course/:courseId',

  getChildRoutes(location, callback) {
    require.ensure([], function (require) {
      callback(null, [
        require('./routes/Announcements'),
        require('./routes/Assignments'),
        require('./routes/Grades'),
      ])
    })
  },

  getIndexRoute(location, callback) {
    require.ensure([], function (require) {
      callback(null, require('./components/Index'))
    })
  },

  getComponents(location, callback) {
    require.ensure([], function (require) {
      callback(null, require('./components/Course'))
    })
  }
}
```