

09 | MVC架构解析：视图（View）篇

2019-09-30 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 18:16 大小 14.65M



你好，我是四火。

今天我们继续学习 MVC 架构，主要内容就是 MVC 架构的第二部分——视图（View）。

概念

首先，我想问一问，什么是视图？有程序员说是界面，有程序员说是 UI（User Interface），这些都对，但是都不完整。

我认为**MVC 架构中的视图是指将数据有目的、按规则呈现出来的组件**。因此，如果返回和呈现给用户的不是图形界面，而是 XML 或 JSON 等特定格式组织呈现的数据，它依然是视图，而用 MVC 来解决的问题，也绝不只是具备图形界面的网站或者 App 上的问题。

页面聚合技术

虽然视图的定义实际更宽泛，但是我们平时讲到的视图，多半都是指“页面”。这里，就不得不提花样繁多的页面聚合技术了。

回想一下，之前我们在讲 Model 层的时候，是怎样解耦的？我们的办法就是继续分层，或是模块化；而对于 View 层来说，我们的办法则是拆分页面，分别处理，最终聚合起来。具体来说，这里提到的页面聚合，是指将展示的信息通过某种技术手段聚合起来，并形成最终的视图呈现给用户。页面聚合有这样两种典型类型。

结构聚合：指的是将一个页面中不同的区域聚合起来，这体现的是分治的思想。例如一个页面，具备页眉、导航栏、目录、正文、页脚，这些区域可能是分别生成的，但是最后需要把它们聚合在一起，再呈现给用户。

数据 - 模板聚合：指的是聚合静态的模板和动态的数据，这体现的是解耦的思想。例如有的新闻网站首页整个页面的 HTML 是静态的，用户每天看到的样子都是差不多的，但每时每刻的新闻列表却是动态的，是不断更新的。

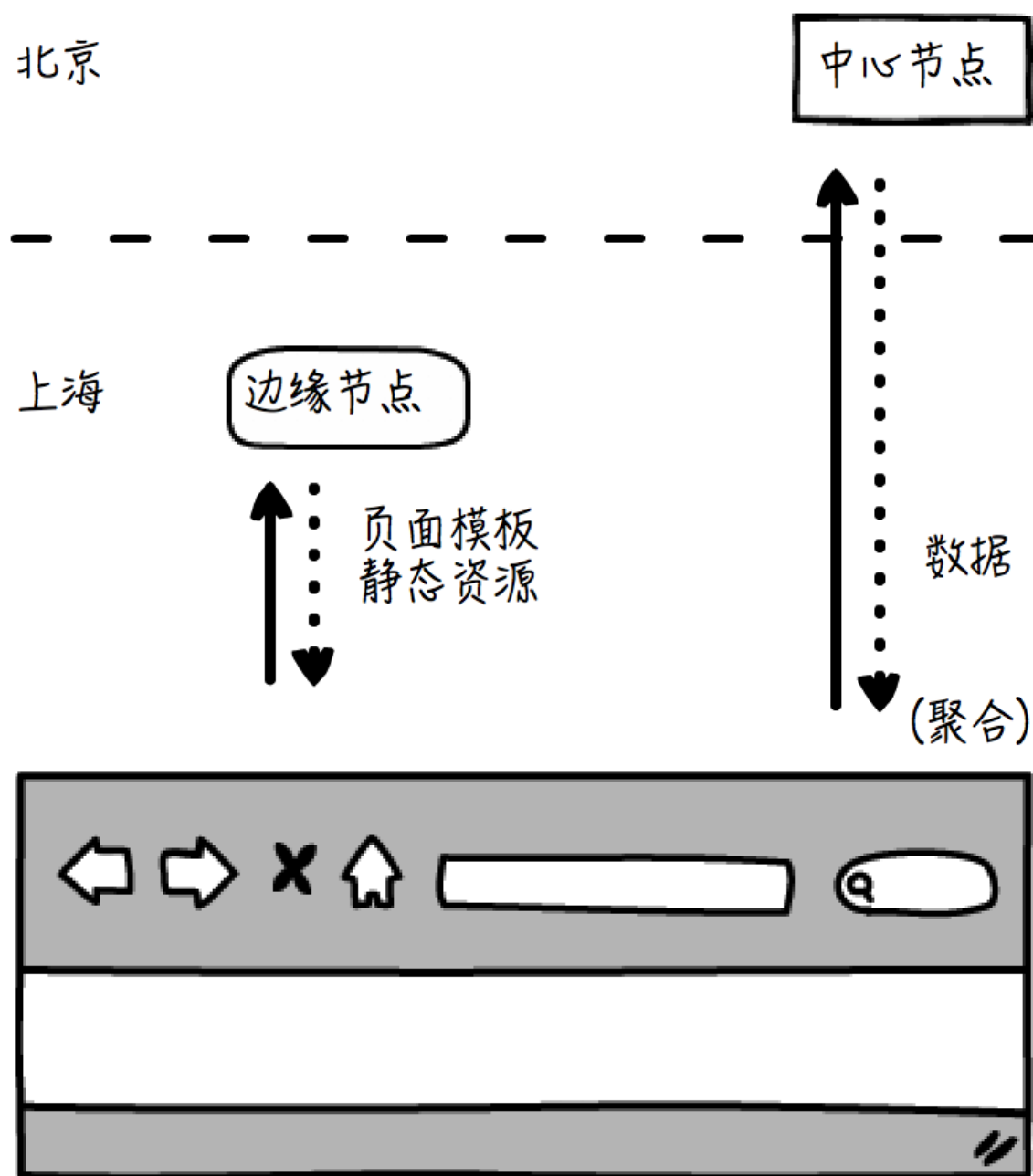
请注意这两者并不矛盾，很多网站的页面都兼具这两种聚合方式。

服务端和客户端聚合方式的比较

客户端聚合技术的出现远比服务端晚，因为和服务端聚合不同，这种聚合方式对于客户端的运算能力，客户端的 JavaScript 技术，以及浏览器的规范性都有着明确的要求。但是，客户端聚合技术却是如今更为流行的技术，其原因包括：

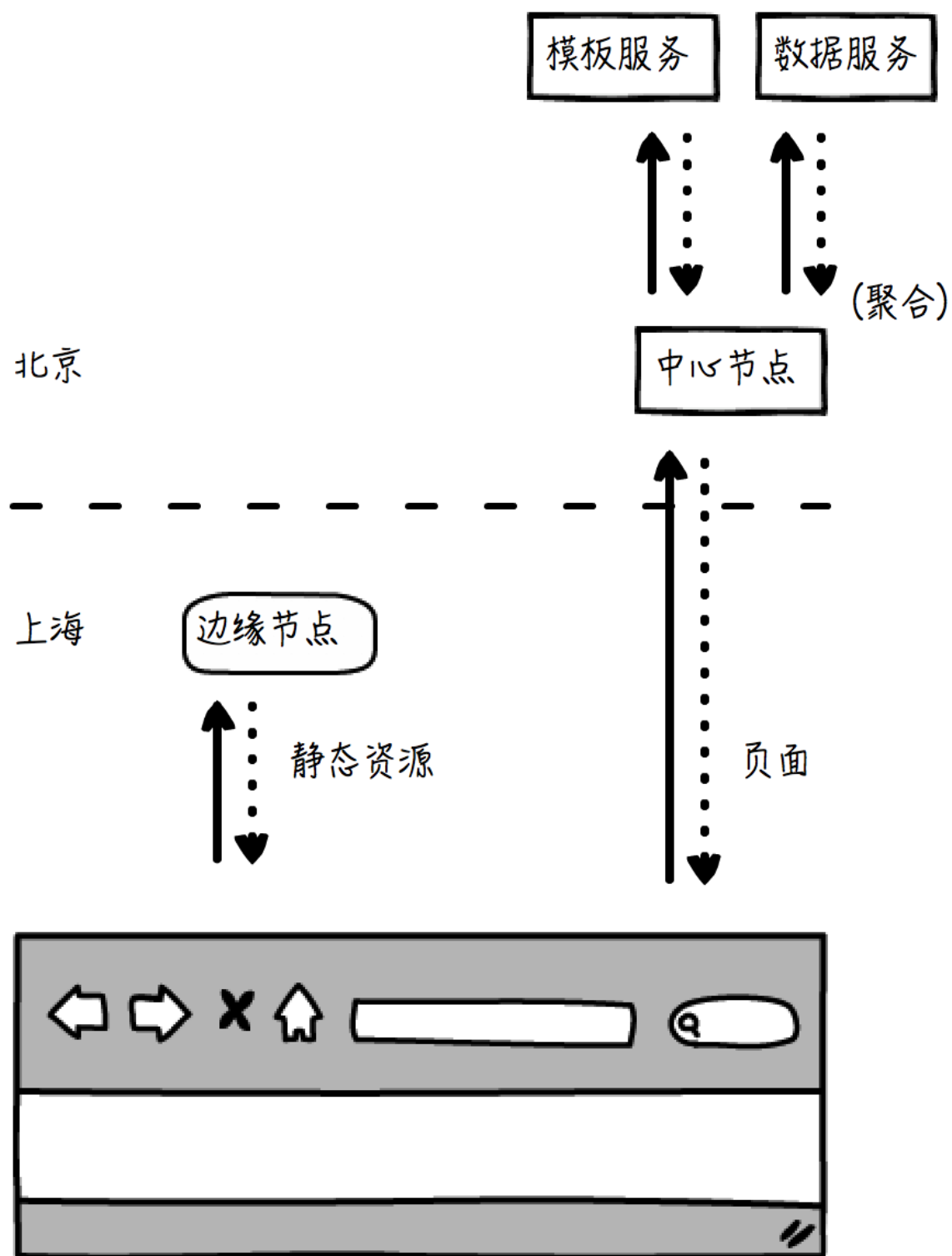
架构上，客户端聚合达成了客户端 - 服务端分离和模板 - 数据聚合这二者的统一，这往往可以简化架构，保持灵活性。

比如说，对于模板和静态资源（如脚本、样式、图片等），可以利用 CDN（Content Delivery Network，内容分发网络）技术，从网络中距离最近的节点获取，以达到快速展示页面的目的；而动态的数据则可以从中心节点异步获取，速度会慢一点，但保证了数据的一致性。数据抵达浏览器以后，再完成聚合，静态和动态的资源可以根据实际情况分别做服务端和客户端的优化，比如浏览器适配、缓存等等。如下图：



你看上面这个例子，浏览器在上海，模板和静态资源从本地的上海节点获取，而数据异步从北京的中心节点获取。这种方式下，静态资源的访问是比较快的，而为了保证一致性，数据是从北京的中心节点获取的，这个速度会慢一些。在模板抵达浏览器以后，先展示一个等待的效果，并等待数据返回。在数据也抵达浏览器以后，则立即通过 JavaScript 进行客户端的聚合，展示聚合后的页面。

如果我们使用服务端聚合，就必须在服务端同时准备好模板和数据，聚合形成最终的页面，再返回给浏览器。整个过程涉及到的处理环节更多，架构更为复杂，而且同样为了保证一致性，数据必须放在北京节点，那么模板也就被迫从北京节点取得，聚合完成之后再返回，这样用户的等待时间会更长，用户也会看到浏览器的进度条迟迟完不成。见下图：



资源上，客户端聚合将服务器端聚合造成的计算压力，分散到了客户端。可是实际上，这不只是计算的资源，还有网络传输的资源等等。比如说，使用服务端聚合，考虑到数据是会变化的，因而聚合之后的报文无法被缓存；而客户端聚合则不然，通常只有数据是无法被缓存，模板是可以被缓存起来的。

但是，**客户端聚合也有它天然的弊端。其中最重要的一条，就是客户端聚合要求客户端具备一定的规范性和运算能力。**这在现在多数的浏览器中都不是问题，但是如果是手机浏览器，这样的问题还是很常见的，由于操作系统和浏览器版本的不同，考虑聚合逻辑的兼容性，客户端聚合通常对终端适配有更高的要求，需要更多的测试。

在实际项目中，我们往往能见到客户端聚合和服务端聚合混合使用。具体来说，Web 页面通常主要使用客户端聚合，而某些低端设备页面，甚至 Wap 页面（常用于较为低端的手机上）则主要使用服务端聚合。下面我们就来学习一些具体的聚合技术。

常见的聚合技术

1. iFrame 聚合

iFrame 是一种最为原始和简单的聚合方式，也是 CSI (Client Side Includes, 客户端包含) 的一种典型方式，现在很多门户网站的广告投放，依然在使用。具体实现，只需要在 HTML 页面中嵌入这样的标签即可：

 复制代码

```
1 <iframe src="https://..."></iframe>
```

这种方式本质上是给当前页面嵌入了一个子页面，对于浏览器来说，它们是完全独立的两个页面。其优势在于，不需要考虑跨域问题，而且如果这个子页面出了问题，往往也不会影响到父页面的展示。

不过，这种方式的缺点也非常明显，也是因为它们是两个独立的页面。比如子页面和父页面之间的交互和数据传递往往比较困难，再比如预留 iFrame 的位置也是静态的，不方便根据 iFrame 实际的内容和浏览器的窗口情况自适应并动态调整占用位置和大小，再比如对搜索引擎的优化不友好等等。

2. 模板引擎

模板引擎是最完备、最强大的解决方案，无论客户端还是服务端，都有许许多多优秀的模板引擎可供选择。比如 [Mustache](#)，它不但可以用作客户端，也可以用作服务端的聚合，这是

因为它既有 JavaScript 的库，也有后端语言，比如 Java 的库，再比如非常常用的 [Underscore.js](#)，性能非常出色。

某些前端框架，为了达到功能或性能上的最优，也会自带一套自己的模板引擎，比如 AngularJS，我在下一章讲前端的时候会介绍。

在使用模板引擎的时候，需要注意保持 View 层代码职责的清晰和纯粹，这在全栈项目开发的过程中尤为重要。负责视图，就只做展示的工作，不要放本该属于 Model 层的业务逻辑，也不要干请求转发和流程控制等 Controller 的活。回想上一讲我们学的 JSP 模板，就像 JSP Model 1 一样，功能多未必代表着模板引擎的优秀，有时候反而是留下了一个代码耦合的后门。

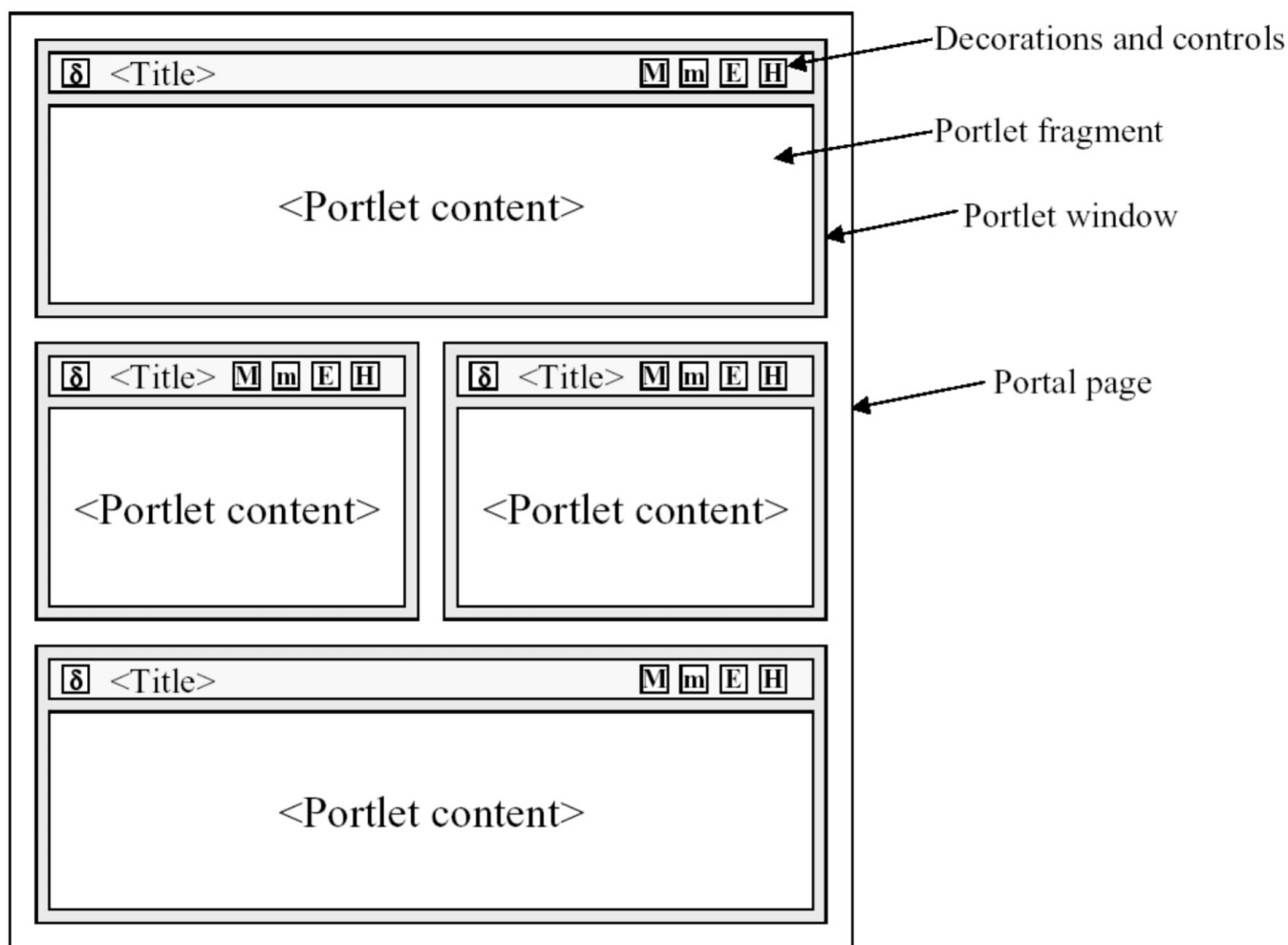
3. Portlet

Portlet 在早几年的门户应用（Portal）中很常见，它本身是一种 Web 的组件，每个 Portlet 会生成一个标记段，多个 Portlets 生成的标记段可以最终聚集并嵌入到同一个页面上，从而形成一个完整的最终页面。

技术上，Portlet 可以做到远程聚合（服务端），也可以做到本地聚合（客户端），数据来源的业务节点可以部署得非常灵活，因此在企业级应用中也非常常见。

Java 的 Portlet 规范经历了 [三个版本](#)，详细定义了 Portlet 的生命周期、原理机制、容器等等方方面面。从最终的呈现来看，网站应用 Portlet 给用户的体验就像是在操作本地计算机一样，多个窗口层叠或平铺在桌面，每个窗口都是独立的，自包含的，并且可以任意调整位置，改变布局 and 大小。

如今 Portlet 因为其实现的复杂性、自身的限制，和较陡峭的学习曲线，往往显得比较笨重，因此应用面并不是很广泛。



(上图来自 JBoss 的官方文档: [Portlet development](#), 上图代表了一个页面, 上面的每一个窗口都分别由一个 Portlet 实现)

4. SSI

还记得前面讲过的 CSI, 客户端包含吗? 与之相对的, 自然也有服务端包含——SSI (Server Side Includes)。它同样是一种非常简单的服务端聚合方式, 大多数流行的 Web 服务器都支持 SSI 的语法。

比如下面这样的一条“注释”, 从 HTML 的角度来讲, 它确实是一条普通的注释, 但是对于支持 SSI 的服务器来说, 它就是一条特殊的服务器端包含的指令:

复制代码

```
1 <!--#include file="extend.html" -->
```


模板引擎的工作机制

前面已经提及了一些常见的页面聚合技术，但是模板引擎始终是最常用的，也自然是其中的重点。下面我们就结合一个小例子来一窥模板引擎的工作机制。

还记得 [\[第 07 讲\]](#) 介绍的 JSP 工作原理吗？类似的，模板引擎把渲染的工作分为编译和执行两个环节，并且只需要编译一次，每当数据改变的时候，模板并没有变，因而反复执行就可以了。


只不过这次，**我们在编译后生成的目标代码，不再是 class 文件了，而是一个 JavaScript 的函数。**因此我们可以尽量把工作放到预编译阶段去，生成函数以后，原始的模板就不再使用了，后面每次需要执行和渲染的时候直接调用这个函数传入参数就可以了。

比如这样的 [Handlebars](#) 模板，使用一个循环，要在一个表格中列出图书馆所有图书的名字和描述：

 复制代码

```
1 <table>
2   {{#each books}}
3   <tr>
4     <td>
5       {{this.name}}
6     </td>
7     <td>
8       {{this.desc}}
9     </td>
10  </tr>
11  {{/each}}
12 </table>
```

接着，模板被加载到变量 `templateContent` 里面，传递给 `Handlebars` 来进行编译，编译的结果是一个可执行的函数 `func`。编译过程完成后，就可以进行执行的过程了，`func` 接受一个图书列表的入参，输出模板执行后的结果。这两个过程如下：


 复制代码

```
1 var func = Handlebars.compile(templateContent);
2 var result = func({
3   books : [
4     { name : "A", desc : "... " },
5     { name : "B", desc : "... " }
6   ]
7 })
```



```
7 });
```

如果我们想对这个 func 一窥究竟，我们将看到类似这样的代码：

 复制代码

```
1 var buffer = "", stack1, functionType="function", escapeExpression=this.escapeExpression;
2
3 function program1(depth0,data) {
4   var buffer = "", stack1;
5   buffer += "\n  <tr>\n    <td>"
6     + escapeExpression(((stack1 = depth0.name),typeof stack1 === functionType ? stack1.:
7     + "</td>\n    <td>"
8     + escapeExpression(((stack1 = depth0.desc),typeof stack1 === functionType ? stack1.:
9     + "</td>\n  </tr>\n  ");
10  return buffer;
11 }
12
13 buffer += "\n<table>\n  ";
14 stack1 = helpers.each.call(depth0, depth0.books, {hash:{},inverse:self.noop,fn:self.prog
15 if(stack1 || stack1 === 0) { buffer += stack1; }
16 buffer += "\n</table>\n";
17 return buffer;
```

我们不需要对上面代码的每一处都了解清楚，但是可以看到一个大概的执行步骤，模板被编译后生成了一个字符串拼接的方法，即模板本身的字符串，去拼接实际传出的数据：

由于模板中定义了一个循环，因此方法 program1 在循环中被调用若干次；

对于 td 标签中间的数据，会判断是直接拼接，还是作为方法递归调用，拼接其返回值。

总结思考

今天我们学习了 MVC 架构中的视图层（View），对于五花八门的页面聚合技术，我们需要抓住其本质，和前面学习 Model 层的解耦一样，应对软件复杂性的问题，绝招别无二致，就是“拆分”。

无论是分层、分模块，还是分离静态模板和动态数据，当我们定义了不同的拆分方法，我们就把一个复杂的东西分解成组成单一、职责清晰的几个部分，分别处理以后，再聚合起来，不同的聚合方法正是由这些不同的拆分方法所决定的。

往后我们还会学习别的解耦技术，到那时请你回想我们本章到目前为止学过的这些有关“拆分”的方法，再放到一起比较，我相信你会有更多感悟。

在今天的选修课堂和扩展阅读之前，我先来提两个问题：

你在工作中是否使用过什么模板引擎，能说说当初在做技术选型时为什么选择了它吗？

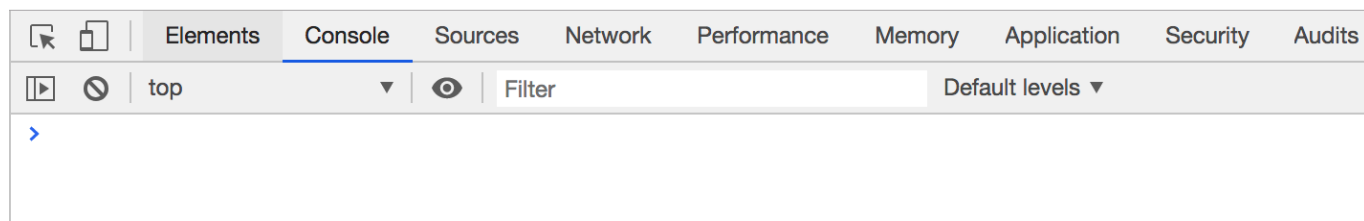
有朋友说，服务端聚合已经过时了，现在的网站都应该在客户端完成聚合的，你同意这个观点吗？

好，对于上面的问题，以及今天的内容，你有什么想法？欢迎在留言区和我讨论。


选修课堂：动手使用 HTML 5 的模板标签

文中我介绍了一些模板技术，还有一点你要知道，HTML 5 引入了模板标签，自此之后，我们终于可以不依赖于任何第三方库，在原生 HTML 中直接使用模板了。下面就让我们来动手实践一下吧。

打开 Chrome 的开发者工具，选择 Console 标签：



然后，让我们先来检验一下你的浏览器是否支持 HTML 5 模板，即 `template` 标签。请执行：

 复制代码

```
1 'content' in document.createElement('template')
```

你应该能看到 `“true”`，这就意味着你的浏览器是支持的。这是因为，`content` 是 HTML 5 的 `template` 标签特有的属性，用于放置原模板本身的内容。

接着，请在硬盘上创建一个 HTML 文件 `template.html`，写入如下内容：

```
1 <!doctype html>
2
3 <html>
4   <div>1</div>
5   <div>3</div>
6   <template id="t">
7     <div>2</div>
8   </template>
9 </html>
```

使用 Chrome 打开它，你应该只能看到分别显示为 “1” 和 “3” 的两行，而 template 标签内的内容，由于是模板的关系，被浏览器自动忽略了。

我们再打开 Chrome 的开发者工具，选择 Console 标签，这次敲入这样两行命令：

```
1 rendered = document.importNode(document.getElementById("t").content, true);
2 document.getElementsByTagName("div")[0].append(rendered);
```

这表示找到 id 为 “t” 的模板节点，根据其中的内容来创建一个节点，接着把这个节点安插到第一个 div 的标签后面。

这时，你应该看能看到三行，分别为 “1” 、 “2” 和 “3” 。

扩展阅读

【基础】今天的内容我们正式和几个 HTML 的标签见面了，如果你对 HTML 也不太熟悉的话，请一定学习一下，作为前端基础的三驾马车之一（另两驾是 CSS 和 JavaScript），我们以后会经常和它们见面的。首推 MDN 的教程，有一篇比较短的 [HTML 基础](#)，也有更为[详尽的展开](#)。

【基础】文中也涉及到了一点点 JavaScript 的基础知识，如果你对于 JavaScript 还不了解，那么我推荐你阅读 MDN 的 [JavaScript 教程](#)中的快速入门部分。

对于文中模板引擎以外的三种聚合方式，我也给你找了可选的阅读材料，你可以跟随自己的兴趣选择：对于 iFrame 聚合，[iframe, or not, that is the question](#) 这篇文章介绍了

和使用 script 标签来写入页面内容比起来，使用 iFrame 的优劣；对于 Portlet 聚合，请参阅 [Java Portlet Specification](#) 词条，你将看到 Portlet 规范从 1.0 到 3.0 的改进；对于 SSI，请参阅维基百科的[服务器端内嵌](#)词条，上面介绍了一些它常用的指令。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | MVC架构解析：模型（Model）篇

精选留言 (2)

写留言



张理查rootv

2019-09-30

喜欢这种带扩展阅读的方式，节省了很多查找优质信息的时间。



Kada

2019-09-30

从反爬虫的角度，

放在客户端聚合，势必要分多个请求。不想暴露的数据api就得引入加解密。

对于服务端聚合的页面，比较好控制呈现的数据维度和量，爬虫也只能老实的解析dom...
展开 ∨

