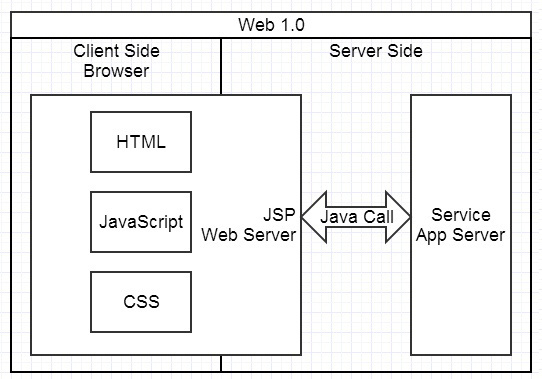
**一、简单明快的早期时代**



可称之为 Web 1.0 时代，非常适合创业型小项目，不分前后端，经常 3-5 人搞定所有开发。页面由 JSP、PHP 等工程师在服务端生成，浏览器负责展现。基本上是服务端给什么浏览器就展现什么，展现的控制在 Web Server 层。

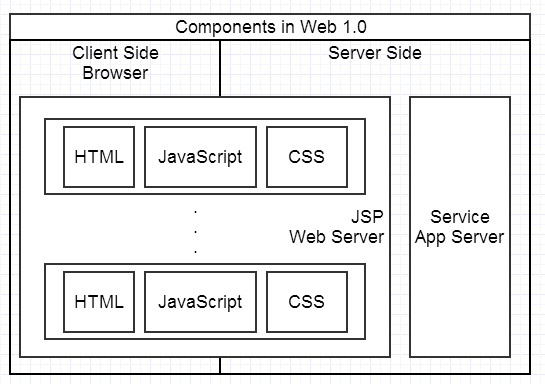
这种模式的好处是：简单明快，本地起一个 Tomcat 或 Apache 就能开发，调试什么的都还好，只要业务不太复杂。

然而业务总会变复杂，这是好事情，否则很可能就意味着创业失败了。业务的复杂会让 Service 越来越多，参与开发的人员也很可能从几个人快速扩招到几十人。在这种情况下，会遇到一些典型问题：

1、**Service 越来越多，调用关系变复杂，前端搭建本地环境不再是一件简单的事。**考虑团队协作，往往会考虑搭建集中式的开发服务器来解决。这种解决方案对编译型的后端开发来说也许还好，但对前端开发来说并不友好。天哪，我只是想调整下按钮样式，却要本地开发、代码上传、验证生效等好几个步骤。也许习惯了也还好，但开发服务器总是不那么稳定，出问题时往往需要依赖后端开发搞定。看似仅仅是前端开发难以本地化，但这对研发效率的影响其实蛮大。

2、\*\*JSP 等代码的可维护性越来越差。\*\*JSP 非常强大，可以内嵌 Java 代码。这种强大使得前后端的职责不清晰，JSP 变成了一个灰色地带。经常为了赶项目，为了各种紧急需求，会在 JSP 里揉杂大量业务代码。积攒到一定阶段时，往往会带来大量维护成本。

这个时期，为了提高可维护性，可以通过下面的方式实现前端的组件化：

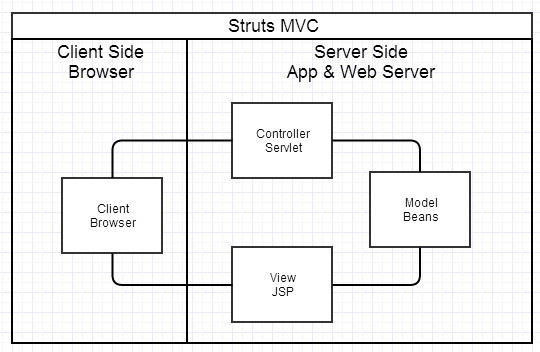


理论上，如果大家都能按照最佳实践去书写代码，那么无论是 JSP 还是 PHP，可维护性都不会差。**但可维护性更多是工程含义，有时候需要通过限制带来自由，需要某种约定，使得即便是新手也不会写出太糟糕的代码。**

**如何让前后端分工更合理高效，如何提高代码的可维护性，在 Web 开发中很重要。**下面我们继续来看，技术架构的演变如何解决这两个问题。

**二、后端为主的 MVC 时代**

为了降低复杂度，以后端为出发点，有了 Web Server 层的架构升级，比如 Structs、Spring MVC 等，这是后端的 MVC 时代。



代码可维护性得到明显好转，MVC 是个非常好的协作模式，从架构层面让开发者懂得什么代码应该写在什么地方。为了让 View 层更简单干脆，还可以选择 Velocity、Freemaker 等模板，使得模板里写不了 Java 代码。看起来是功能变弱了，但正是这种限制使得前后端分工更清晰。然而依旧并不是那么清晰，这个阶段的典型问题是：

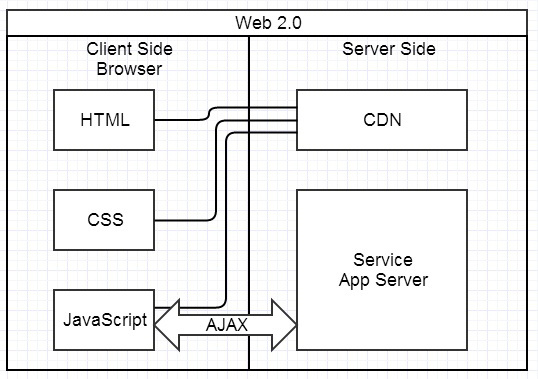
1、**前端开发重度依赖开发环境。**这种架构下，前后端协作有两种模式：一种是前端写 demo，写好后，让后端去套模板。淘宝早期包括现在依旧有大量业务线是这种模式。好处很明显，demo 可以本地开发，很高效。不足是还需要后端套模板，有可能套错，套完后还需要前端确定，来回沟通调整的成本比较大。另一种协作模式是前端负责浏览器端的所有开发和服务器端的 View 层模板开发，支付宝是这种模式。好处是 UI 相关的代码都是前端去写就好，后端不用太关注，不足就是前端开发重度绑定后端环境，环境成为影响前端开发效率的重要因素。

2、\*\*前后端职责依旧纠缠不清。\*\*Velocity 模板还是蛮强大的，变量、逻辑、宏等特性，依旧可以通过拿到的上下文变量来实现各种业务逻辑。这样，只要前端弱势一点，往往就会被后端要求在模板层写出不少业务代码。还有一个很大的灰色地带是 Controller，页面路由等功能本应该是前端最关注的，但却是由后端来实现。Controller 本身与 Model 往往也会纠缠不清，看了让人咬牙的代码经常会出现在 Controller 层。这些问题不能全归结于程序员的素养，否则 JSP 就够了。

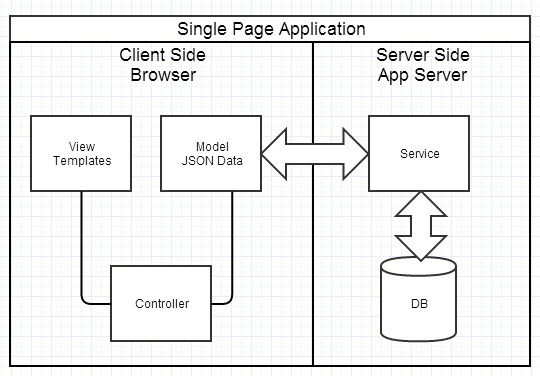
经常会有人吐槽 Java，但 Java 在工程化开发方面真的做了大量思考和架构尝试。Java 蛮符合马云的一句话：让平凡人做非凡事。

**三、Ajax 带来的 SPA 时代**

历史滚滚往前，2004 年 Gmail 像风一样的女子来到人间，很快 2005 年 Ajax 正式提出，加上 CDN 开始大量用于静态资源存储，于是出现了 JavaScript 王者归来的 SPA （Single Page Application 单页面应用）时代。



这种模式下，前后端的分工非常清晰，前后端的关键协作点是 Ajax 接口。看起来是如此美妙，但回过头来看看的话，这与 JSP 时代区别不大。复杂度从服务端的 JSP 里移到了浏览器的 JavaScript，浏览器端变得很复杂。类似 Spring MVC，这个时代开始出现浏览器端的分层架构：



对于 SPA 应用，有几个很重要的挑战：

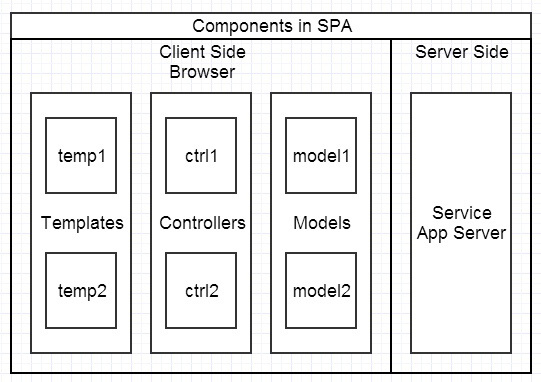
1、**前后端接口的约定。**如果后端的接口一塌糊涂，如果后端的业务模型不够稳定，那么前端开发会很痛苦。这一块在业界有 API Blueprint 等方案来约定和沉淀接口，在阿里，不少团队也有类似尝试，通过接口规则、接口平台等方式来做。有了和后端一起沉淀的接口规则，还可以用来模拟数据，使得前后端可以在约定接口后实现高效并行开发。相信这一块会越做越好。

2、\*\*前端开发的复杂度控制。\*\*SPA 应用大多以功能交互型为主，JavaScript 代码过十万行很正常。大量 JS 代码的组织，与 View 层的绑定等，都不是容易的事情。典型的解决方案是业界的 Backbone，但 Backbone 做的事还很有限，依旧存在大量空白区域需要挑战。

SPA 让前端看到了一丝绿色，但依旧是在荒漠中行走。

**四、前端为主的 MV\* 时代**

为了降低前端开发复杂度，除了 Backbone，还有大量框架涌现，比如 EmberJS、KnockoutJS、AngularJS 等等。这些框架总的原则是先按类型分层，比如 Templates、Controllers、Models，然后再在层内做切分，如下图：



好处很明显：

1、**前后端职责很清晰。**前端工作在浏览器端，后端工作在服务端。清晰的分工，可以让开发并行，测试数据的模拟不难，前端可以本地开发。后端则可以专注于业务逻辑的处理，输出 RESTful 等接口。

2、**前端开发的复杂度可控。**前端代码很重，但合理的分层，让前端代码能各司其职。这一块蛮有意思的，简单如模板特性的选择，就有很多很多讲究。并非越强大越好，限制什么，留下哪些自由，代码应该如何组织，所有这一切设计，得花一本的厚度去说明。

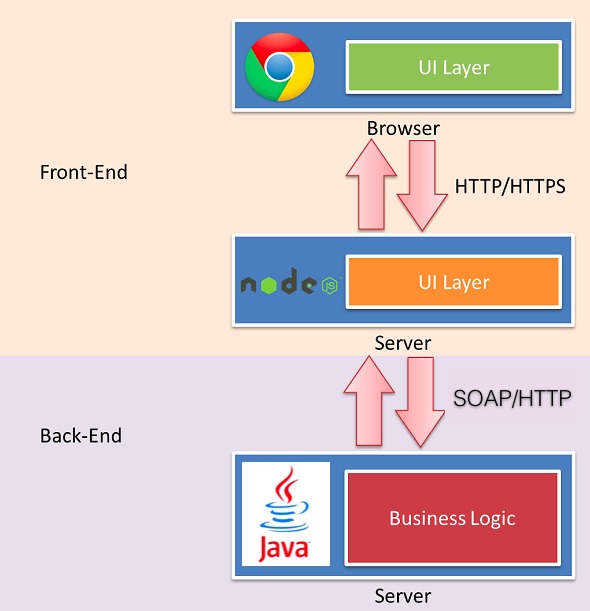
3、**部署相对独立**，产品体验可以快速改进。

但依旧有不足之处：

1、代码不能复用。比如后端依旧需要对数据做各种校验，校验逻辑无法复用浏览器端的代码。如果可以复用，那么后端的数据校验可以相对简单化。  
2、全异步，对 SEO 不利。往往还需要服务端做同步渲染的降级方案。  
3、性能并非最佳，特别是移动互联网环境下。  
4、SPA 不能满足所有需求，依旧存在大量多页面应用。URL Design 需要后端配合，前端无法完全掌控。

**五、Node 带来的全栈时代**

前端为主的 MV\* 模式解决了很多很多问题，但如上所述，依旧存在不少不足之处。随着 Node.js 的兴起，JavaScript 开始有能力运行在服务端。这意味着可以有一种新的研发模式：



在这种研发模式下，前后端的职责很清晰。对前端来说，两个 UI 层各司其职：

1、Front-end UI layer 处理浏览器层的展现逻辑。通过 CSS 渲染样式，通过 JavaScript 添加交互功能，HTML 的生成也可以放在这层，具体看应用场景。

2、Back-end UI layer 处理路由、模板、数据获取、cookie 等。通过路由，前端终于可以自主把控 URL Design，这样无论是单页面应用还是多页面应用，前端都可以自由调控。后端也终于可以摆脱对展现的强关注，转而可以专心于业务逻辑层的开发。

通过 Node，Web Server 层也是 JavaScript 代码，这意味着部分代码可前后复用，需要 SEO 的场景可以在服务端同步渲染，由于异步请求太多导致的性能问题也可以通过服务端来缓解。前一种模式的不足，通过这种模式几乎都能完美解决掉。

与 JSP 模式相比，全栈模式看起来是一种回归，也的确是一种向原始开发模式的回归，不过是一种螺旋上升式的回归。

基于 Node 的全栈模式，依旧面临很多挑战：

1、需要前端对服务端编程有更进一步的认识。比如 network/tcp、PE 等知识的掌握。  
2、Node 层与 Java 层的高效通信。Node 模式下，都在服务器端，RESTful HTTP 通信未必高效，通过 SOAP 等方式通信更高效。一切需要在验证中前行。  
3、对部署、运维层面的熟练了解，需要更多知识点和实操经验。  
4、大量历史遗留问题如何过渡。这可能是最大最大的阻力。

**六、小结**

回顾历史总是让人感慨，展望未来则让人兴奋。上面讲到的研发模式，除了最后一种还在探索期，其他各种在各大公司都已有大量实践。几点小结：

1、模式没有好坏高下之分，只有合不合适。  
2、Ajax 给前端开发带来了一次质的飞跃，Node 很可能是第二次。  
3、SoC（关注度分离） 是一条伟大的原则。上面种种模式，都是让前后端的职责更清晰，分工更合理高效。  
4、还有个原则，让合适的人做合适的事。比如 Web Server 层的 UI Layer 开发，前端是更合适的人选。