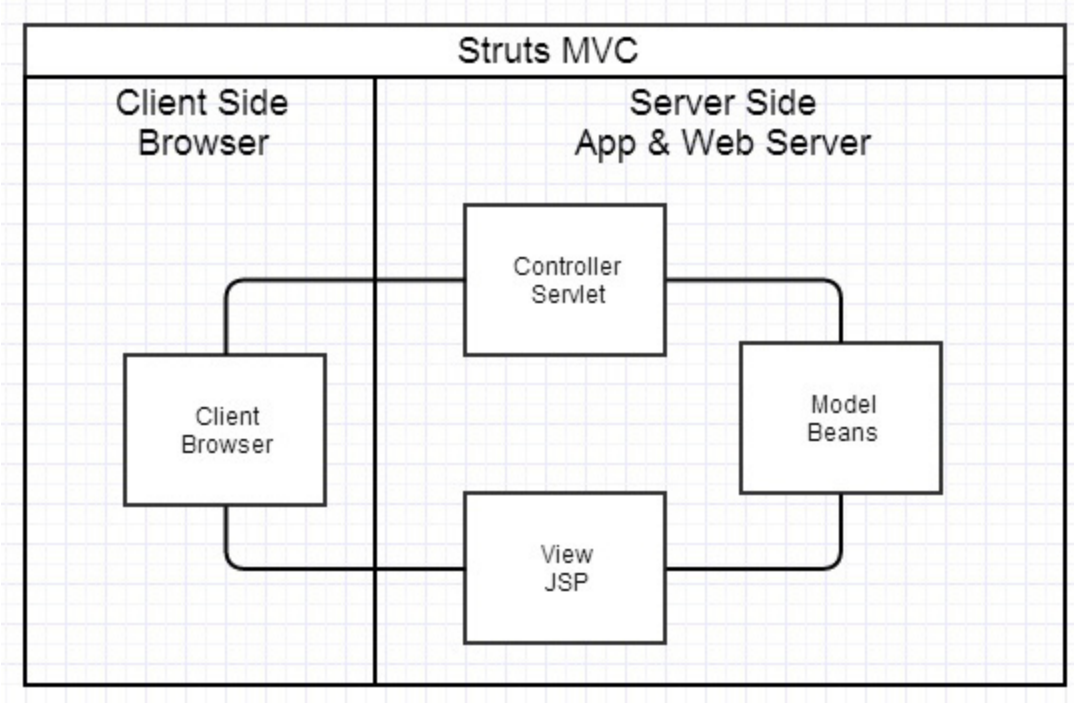
再谈Web开发架构

- 传统型
- SPA
- 基于node半栈
- 基于NodeJS全栈式开发

传统型



> 此种开发模式特别适合小公司小项目，效率高，成本小。往往1-2人就能搞定一个项目，然而问题却接踵而至

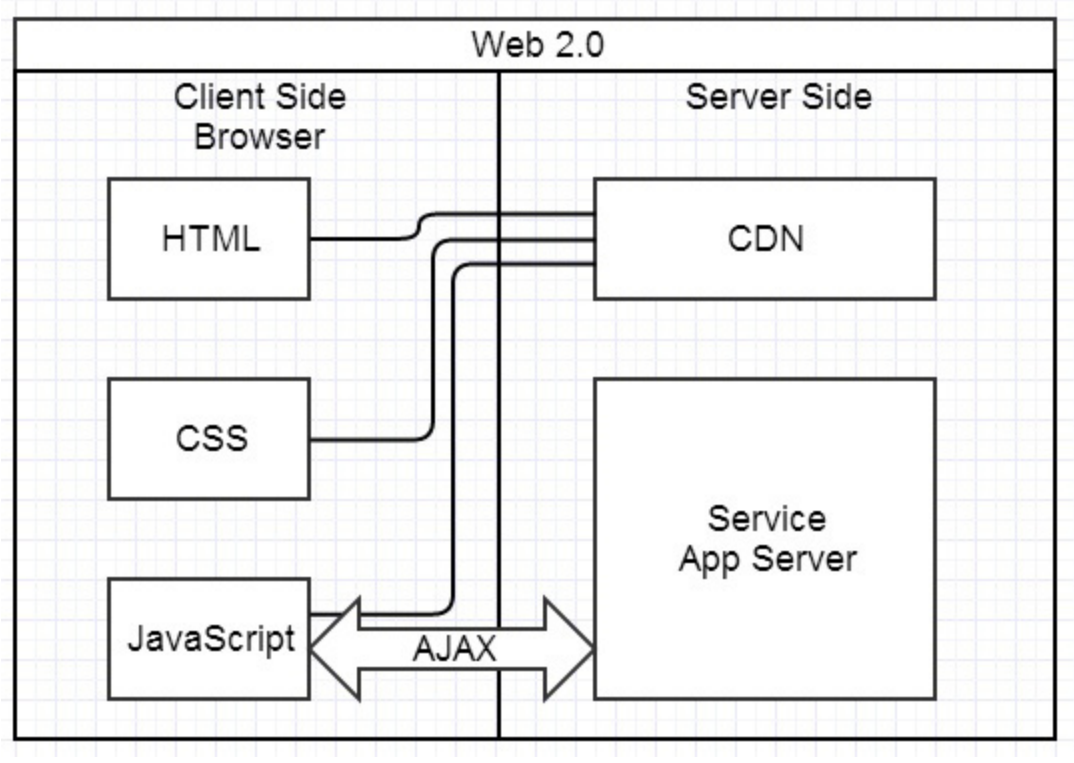
- 前后端不分工: JSP/ASP 等模板语言可以编写java/c# 等后台代码导致页面维护性越来越差。有人js写的很好，java写的很烂，每个人擅长的技术不同

> 怎么办？
> 1.规定不许在jsp页面写java代码
> 2.分工：前端写好静态html，js等 demo，本地开发调试，完事之后，交给后端套模板。

- 前后端分工：不足之处是模板可以套错，套完之后还需要跟前端沟通，沟通成本太大，另外前端开发重度依赖后端环境，前端的开发效率太差 。规定不许再jsp写java约束力太差

> 怎么办？
> 前后端分离(不是分工)，在物理层面上进行隔绝

SPA



> 此开发模式刮起了一阵旋风，好处大大的

优点

- 前后端职责很清晰

> 前端负责交互，后端负责计算和存储

- 部署独立，可以快速响应
- Vue,react,backbone,angular等mv*框架 和 webpack，gulp，fis 的出现让前端变更加工程化

缺点

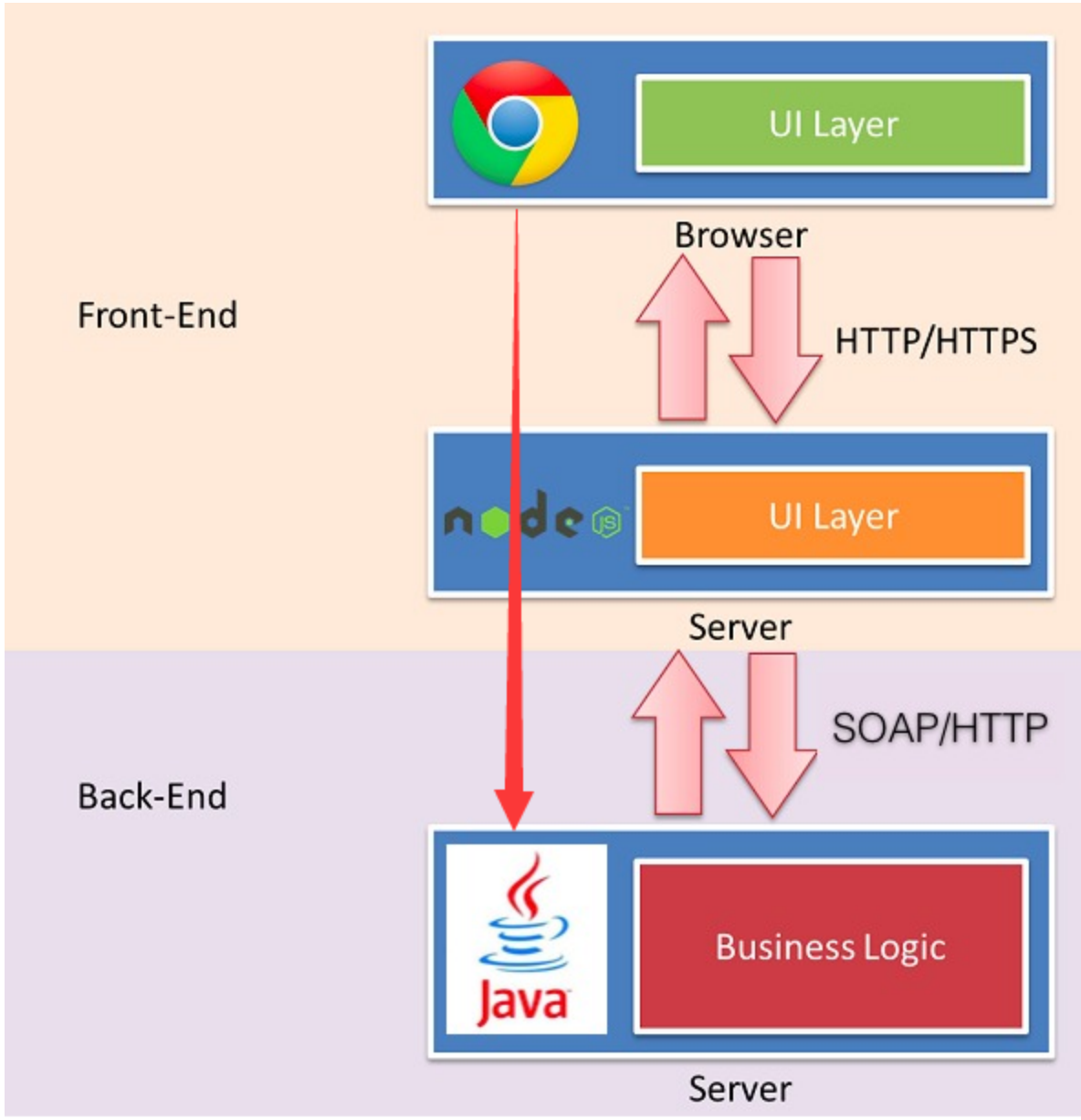
- 代码不能复用

> 后端依然需要对数据进行各种校验，校验代码写了2次

- 全异步，对SEO不利
- 搞不定多页应用
- 由于需要等待js下载完成才能开始渲染，对于移动端常常出现白屏

> 怎么办
> 我们当下的片方后台的模式：基于nodejs的半栈

基于node半栈


Front-End
UI Layer
Browser
HTTP/HTTPS
UI Layer
Server
Back-End
SOAP/HTTP
Java
Business Logic
Server

优点

- 前后端组件共享
- 服务端渲染，解决SEO,移动端首屏渲染
- 处理多页面应用得心应手

> 然而目前的方式依然存在几个问题没有解决

几个问题

- node端 用http 与java 通信未必高效，RPC方式可能更好
- 跨域

> 对于一些ajax请求往往要进行预警options请求

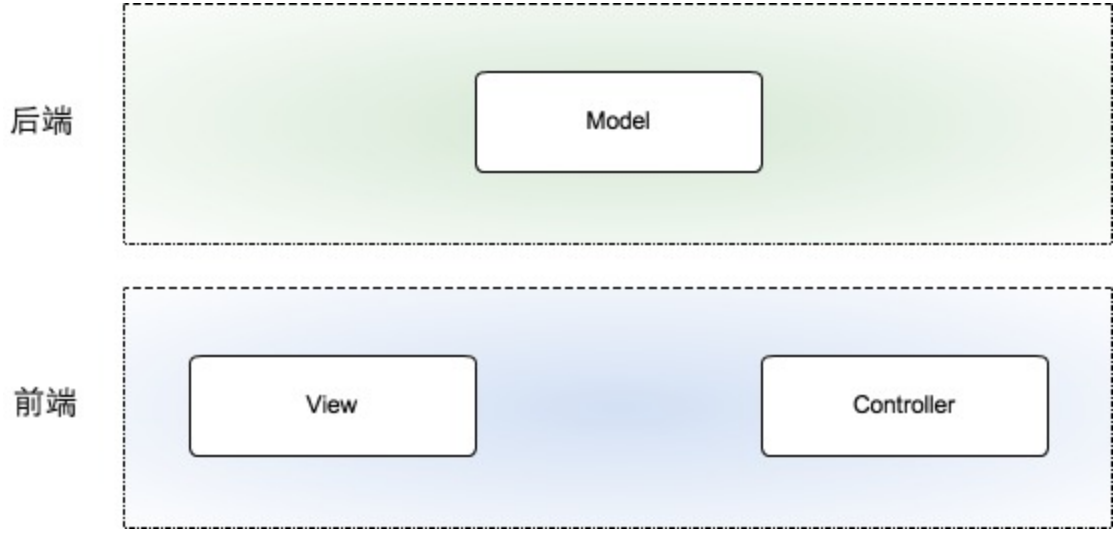
- 完全跨域情况下node端无法进行正确的渲染

> 1.前端域名： www.a.com, 后端域名:www.b.com.
> 2.首次访问(用户未登录)www.a.com ， node端渲染出正确的未登录页面
> 3.用户在页面上访问www.b.com 的api进行登录，此api会设置cookie，此时前端js渲染出登录的页面
> 4.此时用户如果刷新页面，node端将不能给出的已登录页面
> 因为浏览器不会把www.b.com 域下的cookie发送给 www.a.com 导致node端拿不到该用户的登录状态。

- 后端的接口设计问题

> 我们现在的api接口，基本上都是按照页面的展现逻辑来提供的，有时候为了提高效率后端会为前端处理一些展现的逻辑，比如：live.mtime.com 的那一个大大的json，很显然后端人员在开发这个接口时候 是想到了页面，他们开发时甚至要看着原型图，这就意味着后端还涉及到了View层的工作，这与我们上面讲的前端负责交互，后端负责计算和存储 有着不小的冲突，会导致他们的逻辑不在单纯，各个子系统会互相耦合。一般来讲这种耦合应该发生在前端。
> 一句话来说，这样的接口设计会让后端的关注点分离。

> 怎么办
> 后端不在直接提供接口给ajax 直接调用，完全由node端进行代理。cookie 由node进行处理。node 端成为android，ios，web等view层 api 的服务提供者



> 由node端来组装各种model，让耦合发生在node端。把本该就前端控制的部分交由前端掌控

困难

- 基础设施建设，比如session共享,logger等通用模块
- 需要前端对服务端编程有更进一步的认识
- 最佳开发实践
- 安全，性能
- 部署，运维要熟练了解
- 巨大的历史遗留问题怎么办？

> but 关注度分离是一条伟大的原则。所以目标是明确的，道路是曲折的，随着流程的打通和通用解决方案的积 和 更多的项目实践，上面的问题都会解决