LESSON ONE

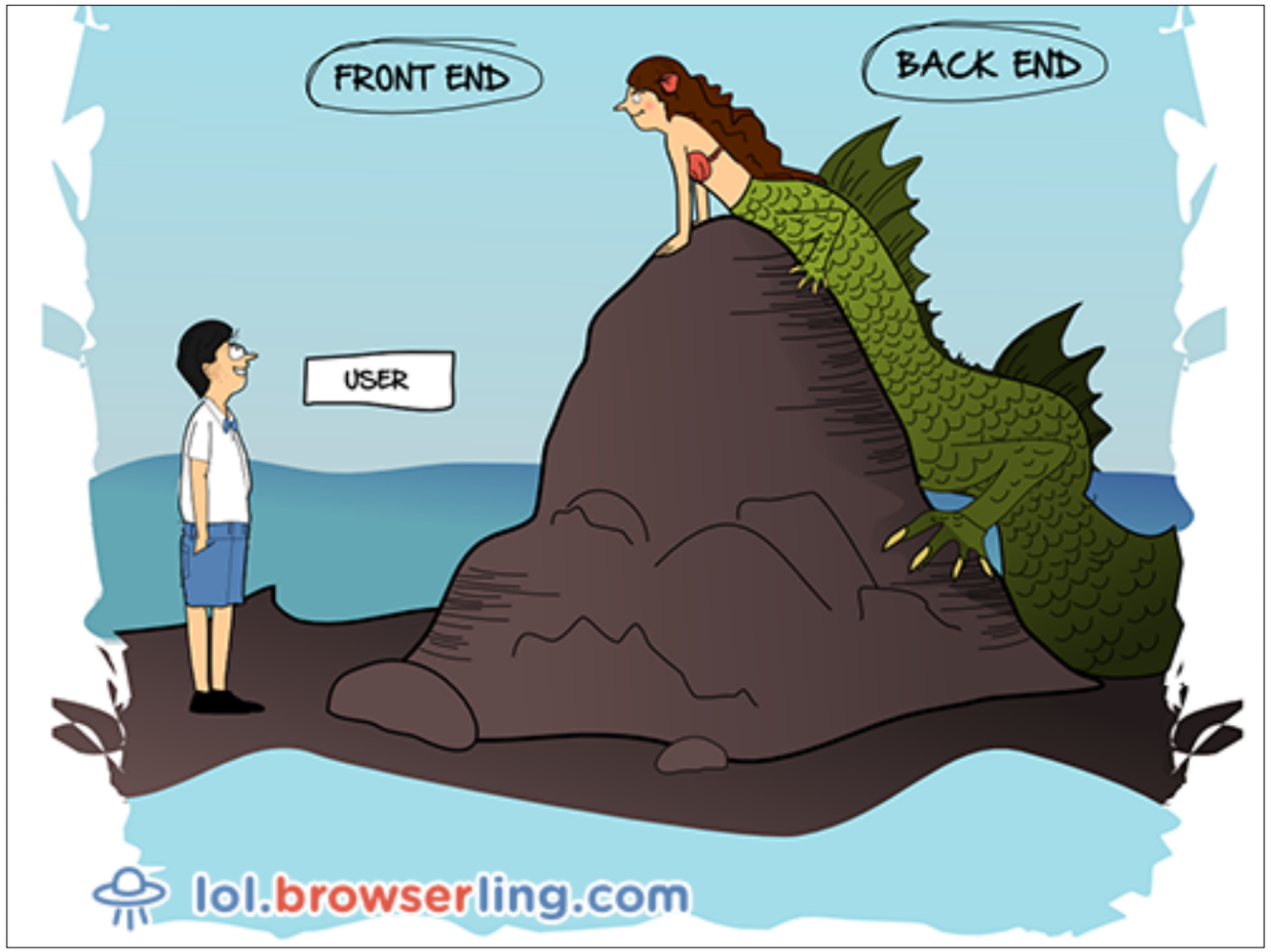
前端表演系

“

十年前端 九年UX。

— *Left*

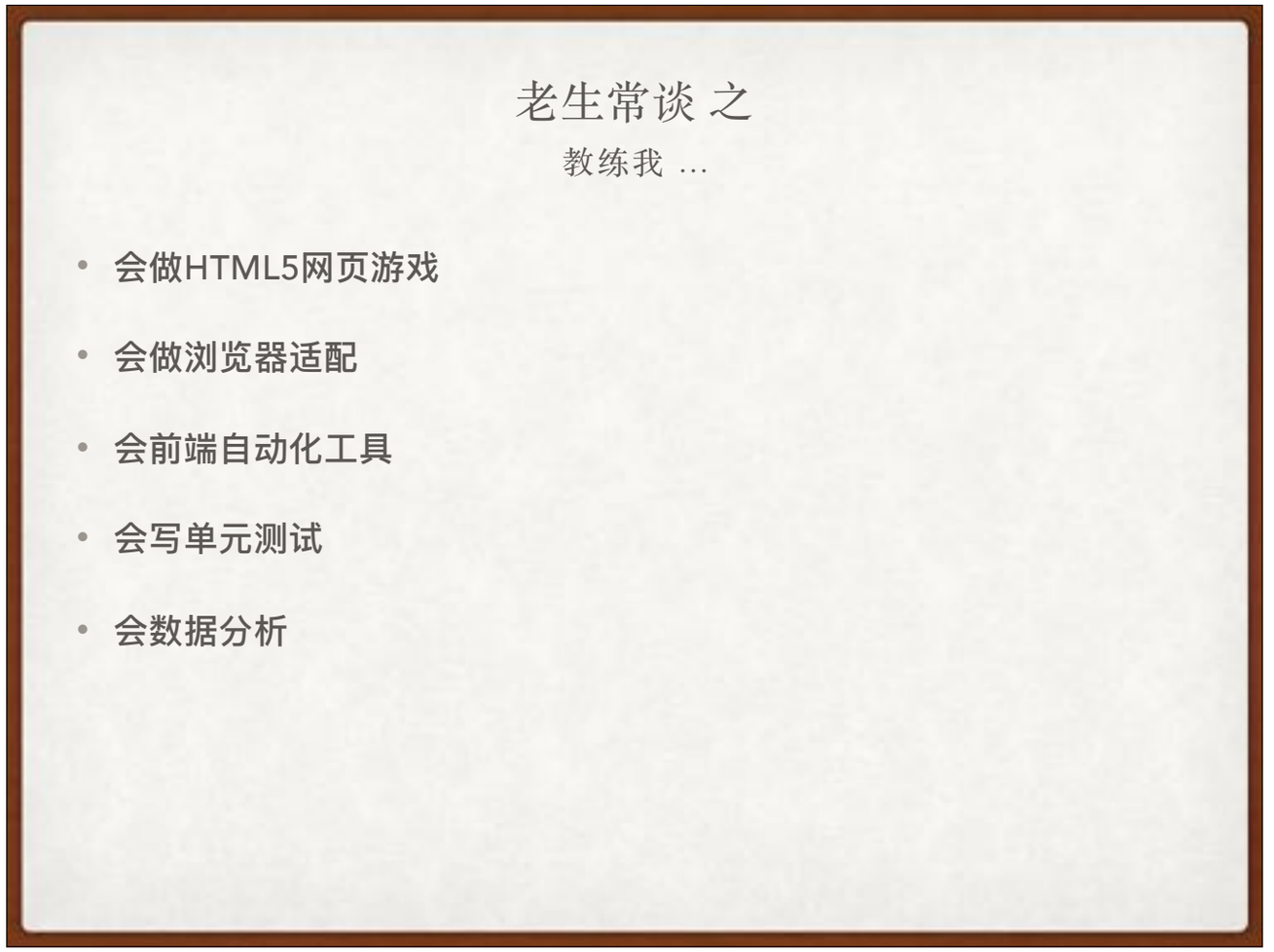
”


FRONT END
BACK END
USER
lol.browserling.com

表演系宗旨

拥抱变化
什么有趣
学什么

前端小



学习有趣的前提是要有基础
所以我默认大家都大概了解了前端是什么（在浏览器中执行的部分），做些什么工作
但是还是要来普及一下具体的细节

前端不仅仅只是做一个网页，前端是完成所有客户端用以将信息展示给用户的部分的工作。

前端在业界其实是有大前端与小前端之分。当然谁也不想自己显得特别的小，尤其是程序员大多都是男生，害怕被人说自己小。所以都统称前端。

所谓“大前端”，顾名思义应该是比“小前端”承担更多职责。传统上，Web应用可分为前端（在浏览器中执行的部分）和后端（在服务器中执行的部分）。

小前端比较趋向于传统的切图型人才，然后设计页面。也会负责一些js的编码工作。针对几十年前在前端圈还不像娱乐圈那样乱，变化也不会像这样来熊抱你的时候，那时候的前端工程师可以称之为现在的小前端。

“大前端”则是将传统上归于后端的服务器脚本和模板划归到前端，根据自己对前端生态圈的了解与对前端建构的理解，来负责团队的前端编码工作。

前端大

老生常谈

大前端

- 对前端生态圈以及建构的理解
- 构想从开始到长尾的UX
- 部分设计、逻辑工作从设计、后端转向前端

学习有趣的前提是要有基础
所以我默认大家都大概了解了前端是什么（在浏览器中执行的部分），做些什么工作
但是还是要来普及一下具体的细节

前端在业界其实是有大前端与小前端之分。当然谁也不想自己显得特别的小，尤其是程序员大多都是男生，害怕被人说自己小。所以都统称前端。

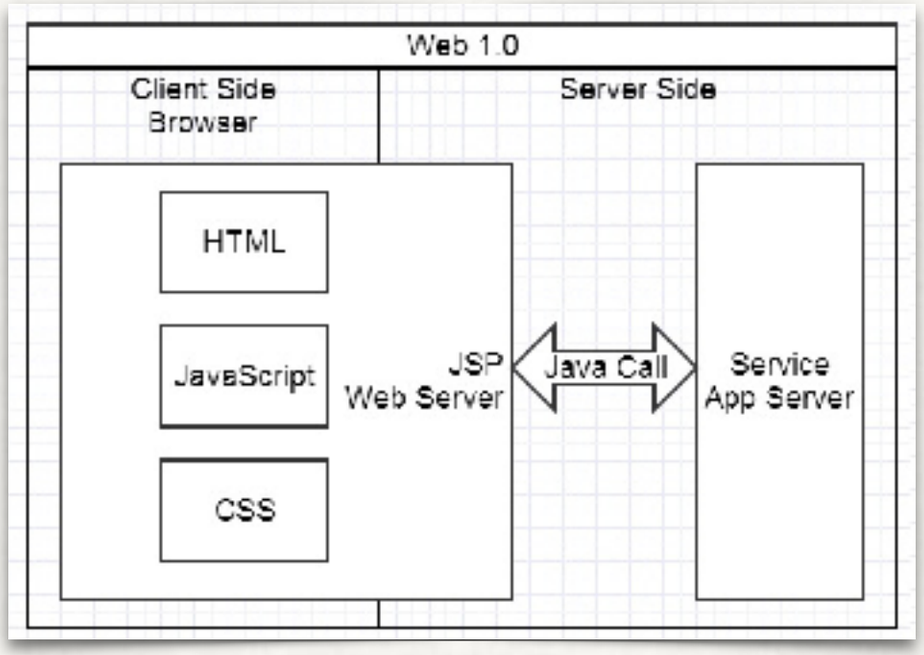
所谓“大前端”，顾名思义应该是比“小前端”承担更多职责。传统上，Web应用可分为前端（在浏览器中执行的部分）和后端（在服务器中执行的部分）。

小前端比较趋向于传统的切图型人才，然后设计页面。也会负责一些js的编码工作。针对几十年前在前端圈还不像娱乐圈那样乱，变化也不会像这样来熊抱你的时候，那时候的前端工程师可以称之为现在的小前端。

“大前端”则是将传统上归于后端的服务器脚本和模板，以及一些属于界面设计的工作划归到前端，根据自己对前端生态圈的了解与对前端建构的理解，来负责团队的前端编码工作。然后一步步走向全栈的深渊。

前后端分离
MV*

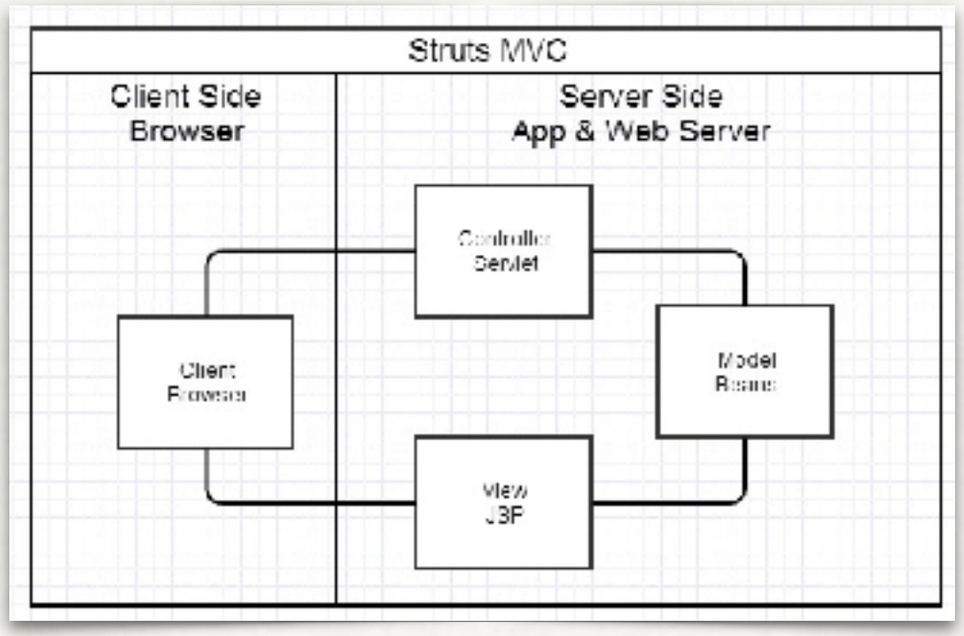
WEB 1.0 时代



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。
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。

缺点：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。

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 高级模板，使得模板里写不了 后端语言代码。

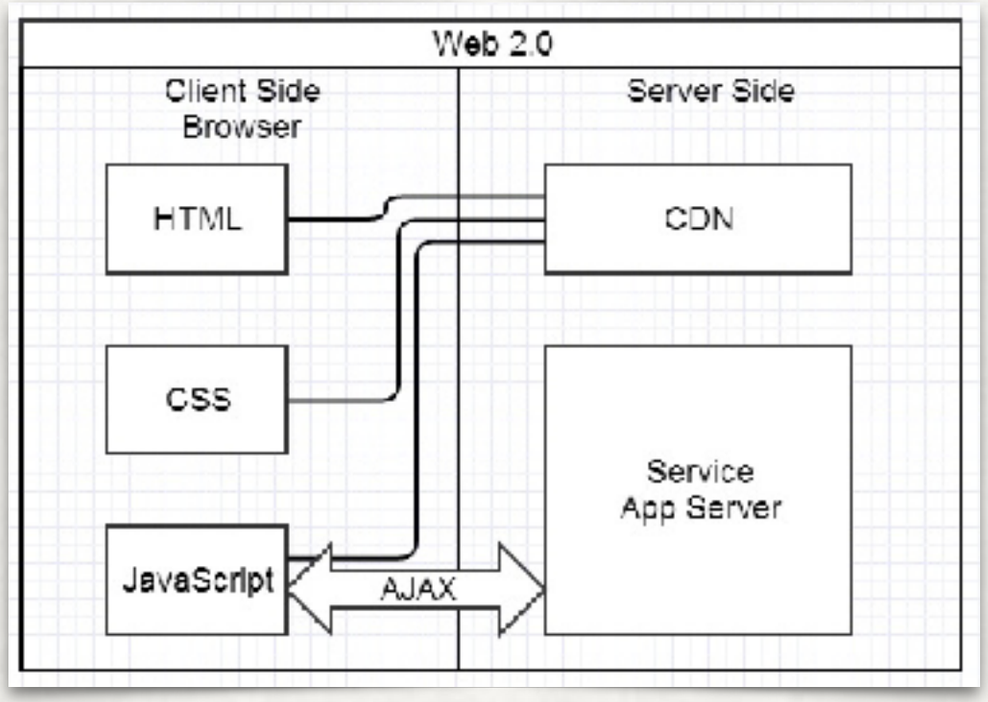
1、前端开发重度依赖开发环境。这种架构下，前后端协作有两种模式：一种是前端写 demo，写好后，让后端去套模板。淘宝早期包括现在依旧有大量业务线是这种模式。好处很明显，demo 可以本地开发，很高效。不足是还需要后端套模板，有可能套错，套完后还需要前端确定，来回沟通调整的成本比较大。可惜这种想法很好，但是一旦付诸实现就会遇到不少问题。首先后端开发者依赖于前端的 Demo，只有看到 HTML 文件他们才可以开始实现 View 层。而前端又依赖于后端开发者完成整体的开发，才能通过网络访问来检查最终的效果，否则他们无法获取真实的数据。
更糟糕的是，一旦需求发生变动，上述流程还需要重新走一遍，前后端的交流依旧无法避免。概况来说就是前后端对接成本太高。
举个例子，在开发 App 的时候，你的同事给你发来一段代码，其中是一个本地写死的视图，然后告诉你:“这个按钮的文字要从数据库，那个图片的内容要通过网络请求获取，你把代码改造一下吧。”。于是你花了半天时间改好了代码，PM 跑来告诉你按钮文字写死就好，但是背景颜色要从数据库读取，另外，再加一个按钮吧。WTF?
显然这种开发流程效率极低，难以接受。

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。 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

AJAX带来的SPA时代

SINGLE PAGE APPLICATION



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 mvc 时代区别不大。复杂度从服务端mvc的 view以及controller的一部分 里移到了浏览器的 JavaScript，浏览器端变得很复杂。

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。

优点：
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。

都有缺点怎么办?

ISOMORPHIC SSR

HTML & CSS & JS

“

TALK IS CHEAP
SHOW ME SOME FUN

— *Left*

”

常用的
HTML元素

三种语言结合

条件联动

不论鼠标指针穿过被选元素或其子元素，都会触发 mouseover 事件。对应mouseout
只有在鼠标指针穿过被选元素时，才会触发 mouseenter 事件。对应mouseleave

mouseenter子元素不会反复触发事件，否则在IE中经常有闪烁情况发生。

为什么兄弟选择器html里面没有换行?

CSS其实不呆板

最后讲一点点设计

PPT高手