前面我们介绍过，早期魔盒通过”物理隔离“的方式实现了”前后端分离“。这一部分我们来谈谈在业务发展过程中，我们在”物理分离“这条路上遇到的问题。而发现问题就要解决问题，我们再来提出一套可行的解决方案。

一、“物理分离”遇到的问题

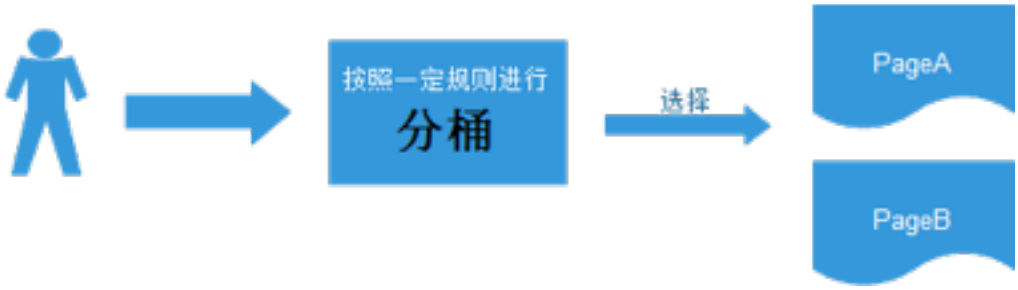
1、ABtest

（1）什么是ABtest

在互联网产品的发展过程中，ABtest是极其重要的一环。我们需要通过一个个阶段性试验来收集、分析数据，从而有可靠的结论指导下一步产品的发展。

进行ABtest实验的核心工作可以用两个词归纳”分桶，响应“。

- 分桶（对数据进行哈希）：一个用户请求发过来，我们是通过A页面响应，还是B页面响应。我们一方面要选择分桶的基础数据（如acookie），另一方面要确定使用什么样的规则进行分桶（简单取模还是别的）。
- 响应：这个很简单，我们只需要分别对来A、B页面进行实现就OK了。



（2）“物理隔离”在ABtest上遇到了什么问题

前面提到”分桶“的两个重要元素（数据和规则），在这一点上单独依靠前端是做不到的。

- 在无登陆情况下，前端没有办法种acookie：
  acookie是通过Tengine的beacon模块种下来的，” 物理分离“的方式页面完全由cdn响应，无法通过Tengine种acookie。
- ABtest分桶信息无法打通全链路：
  严格的ABtest必须将实验分桶信息打通整个服务链路（前端、服务引擎、算法），目的是保证服务的各层能够按照统一的分桶互相配合，从而保证实验的准确性。

而”物理分离“的方式，在这方面先天不足。

2、运营需求

互联网产品少不了运营同学维护的页面，从而实现多样化页面元素、建立直达用户渠道等目的。

常见的运营干预页面，如系统公告、banner、广告位等等。

（1）早期的实现----手工运营

早期的实现可谓”刀耕火种“，运营同学一旦有系统公告等需求的时候，首先通知前端同学，然后前端同学通过一次发布将文案发上去。

当然了，过一段时间还要通过一次发布下掉系统公告。

（2）遇到的问题

最近一段时间，产品运营同学提出了新的需求。希望不同的渠道、设备的用户请求，响应不一样的banner、公告。

举个简单的例子，假如最近我们在做app的推广。那么合理的一种思路是，ios设备用户推荐ios的app，而android设备推荐android app。

这是对运营平台化的诉求，但是上面提到的早期实现（人肉改页面），明显是不能满足需要的。

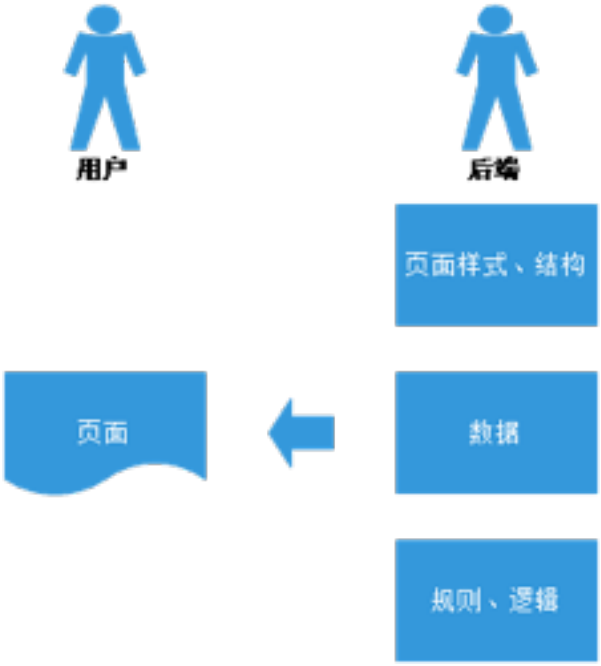
二、说说前端模式----几种前端模式的比较

1、页面是怎么来的

如下图，一个用户发来请求，最终交付给他的就是一个页面。

那么页面从何而来？

- 页面样式、结构：页面的骨骼，描述了一个页面的设计。
- 数据：是页面实际填充的内容
- 规则逻辑：一些业务、运营上的规则，比如前面提到的banner、公告、ABTest等等。



早期的页面服务，只包含上面提到的数据、样式两点就足够了。原因是早期的页面产品已经把规则信息拆分到了前面两者当中。

就像前面提到的案例，每次修改公告都通过重新发布页面的方式来完成。

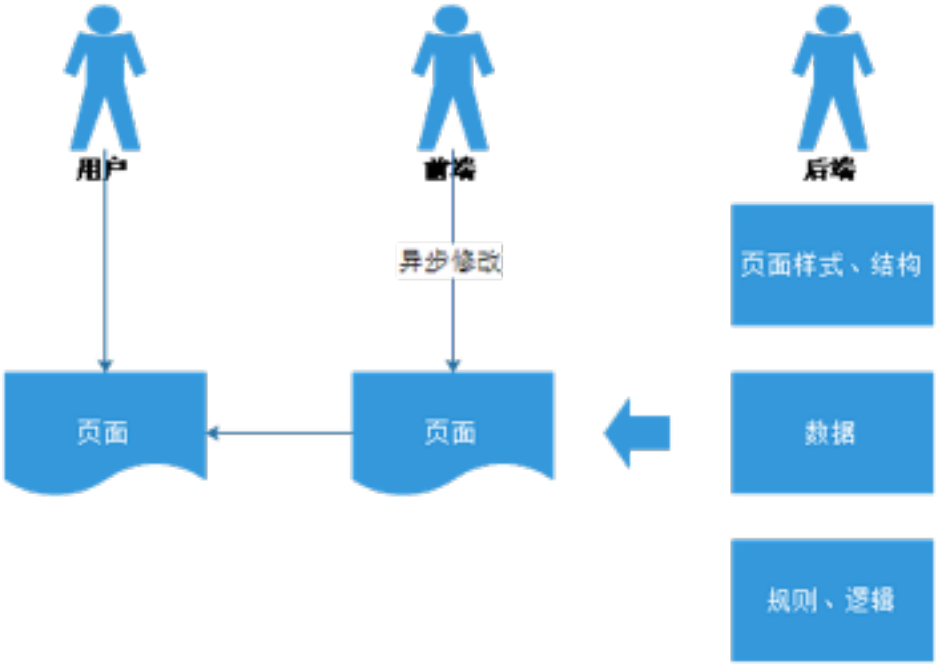
2、web mvc 的实现方式

传统mvc方式，对上述问题提供了解决方案。

我这里不做详细说明（可以看图），只说问题：

后端同学同学除了提供数据，还要关注页面如何渲染（烦躁）；api服务化、测试自动化难以展开。

前端同学只能在原有页面的基础上做异步修改，没有更多的空间发挥能力。



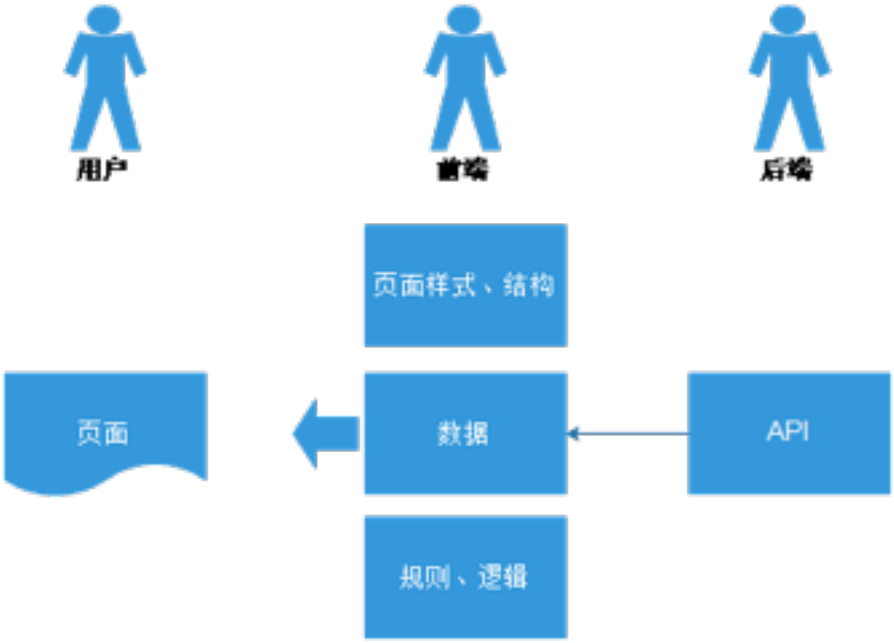
3、物理隔离

mvc模式确实造福了一代人，因为它对”样式、结构“和”数据“进行了解耦的尝试，而且也有成型的框架技术。

而物理隔离是”前后端分离“的一次大胆尝试，我们通过物理上的隔离，彻底将前后端工作分开（见下图）。

- 后端只提供api
- 页面的组织完全由前端完成

前端同学确实比mvc时代更加自由，而后端同学也可以考虑api服务化、自动化测试。



那么问题在哪里?

- 前端---自由也有代价：前面我们分析过，在ABtest、运营需求的实现上，前端表示力不从心。
- 后端---伪服务化：实际上后端提供的是接口，但并不是服务。服务本身应该是相对稳定、具备明确输入输出的，在这一点上现有的api接口难以保证。因为产品迭代过程中，UI调整是十分频繁的。而数据api又和UI十分耦合，这就造成了”前端一调整UI，后端就要跟着发布“这样的局面。

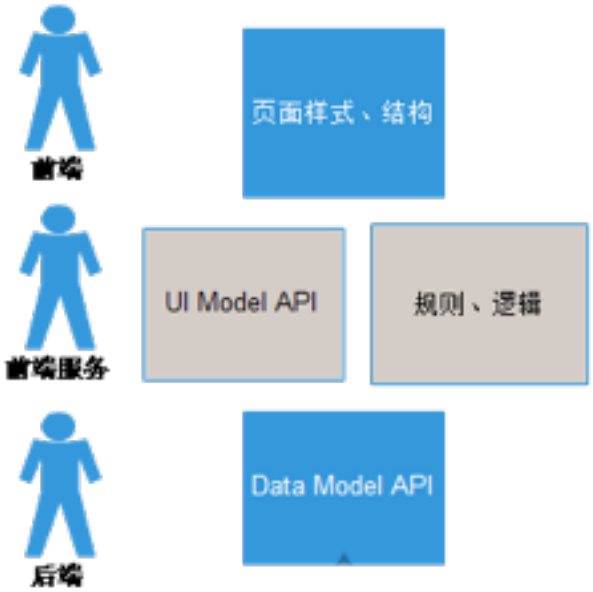
三、到底为什么要“前后端分离”的---要解决什么样的问题?

针对”前后端分离“，我们做了一些尝试，比如”物理隔离“。我们解决了一些问题，也带来了一些新问题。

采用一边倒的方式解决，往往会走向另一个极端。

那么我们是否可以实现一个方案，能够把问题集合的大部分加以解决。

首先，我们看看“前后端分离”的问题集合。



1、前端---专注业务相关的素材渲染:

专注页面渲染，

这里值得一提的是我们把“替换banner、修改公告”等类似的工作抽离出来（放到了前端服务这一层），why?

虽然banner、公告本身确实也是页面的一部分，但是我们认为这部分工作并不能算作前端工作。

前端只需关注“业务素材”的渲染。

2、后端---稳定、安全：

在业务启动之初，我们经历了一段“怎么快怎么来”的发展过程。

随着业务发展壮大，“安全、稳定、可持续集成”成为我们必须要思考的问题。只有这样，才能保证业务持续成长。

否则，我们就会每天疲于奔命似的人肉修改、测试，最终可能导致做死。

下面几点，共同支撑了“安全、稳定、可持续集成”的目标：

（1）服务化：

将稳定的数据服务沉淀到后端，专注于提供数据服务（输入、输出明确）。

（2）和UI解耦：

UI调整是产品迭代中比较频繁的一部分，为了尽可能的减少UI调整对后台服务的牵连，后台服务本身就要做到和UI解耦。

当然，这并不代表后台就可以一劳永逸不再修改，一些重大的UI修改还是有可能对后台服务提出新的挑战（当然这就是业务发展了）。

（3）自动化测试：

通过（1）（2）两点，就使得自动化测试成为可能。

测试人员可以对后台服务进行针对性的自动化测试，进一步提升系统的稳定性。

3、前端服务（灰色部分）：

我把这里叫做灰色部分，是因为无论是mvc模式还是物理隔离，有一部分工作总是像“烫手山芋”一样，在前后端同学之间传来传去。

一句话“他看上去是前端的事，也像后端的事”。

时下十分流行的方案，是在前端、后端之间，新增一个前端服务层，我们对它的定位是这样的：敏捷响应UI变化，提供新的接口给前端；处理同步页面需求（如ABtest、平台化运营）

（1）UI API（敏捷响应UI变化）：

根据UI需要，从后端服务获取数据（Data Model），再根据UI生成视图数据（View Model）。

（2）平台化运营、ABTest：

一改“人肉运营”的方式，通过前端服务实现运营、ABTest平台化。

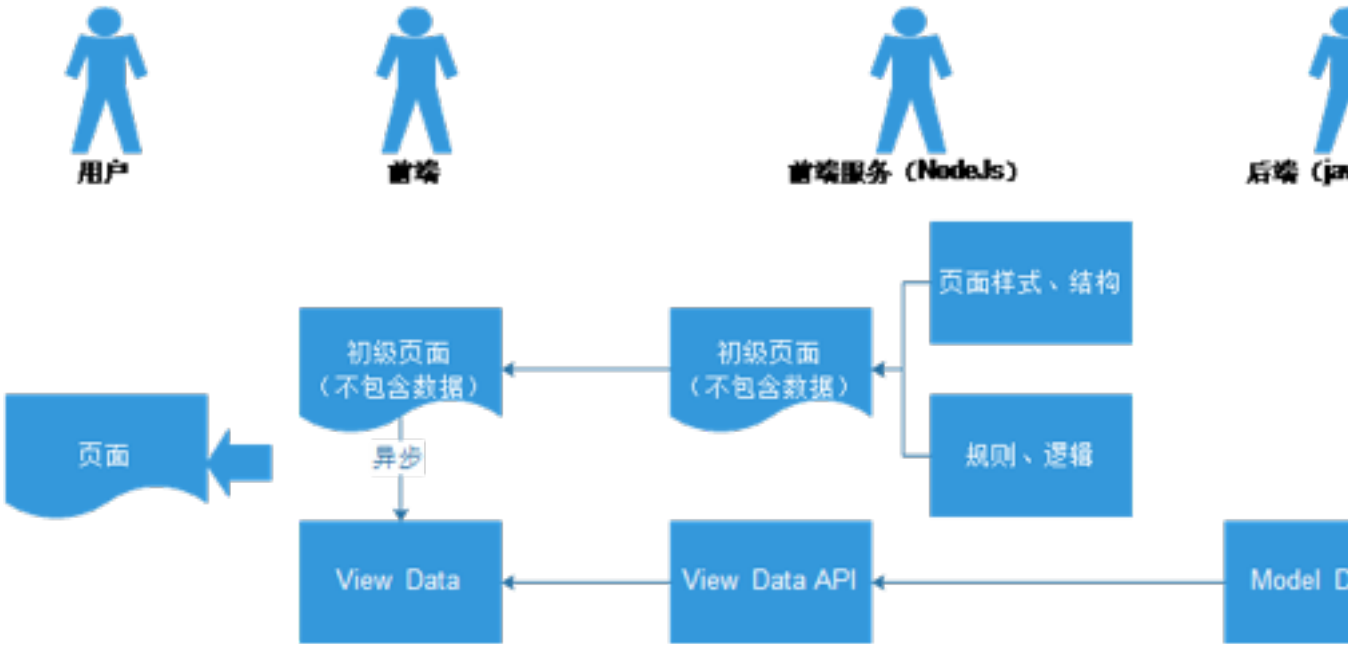
（3）页面打点、监控：

通过前段服务，我们有了同步页面。基于pv的统计、基于页面的监控报警成为可能。

四、前后端分离方案（前端、前端服务、后端）

基于上一部分中描述的问题集合，本部分对问题集合提出解决方案。

引入了新的前端服务层，和前段、后端一起解决上述问题集（如下图）。



1、前端----专注

2、前端服务----敏捷、薄薄的一层

（1）敏捷：适应UI频繁调整的需要

（2）薄薄的一层：强调薄，是因为我们必须防止这一层在迭代过程中越来越重。我们保证这一层十分轻量，将服务数据下沉到后端服务。否则，后端就无法实现服务化。

PS：在这一层的实现语言选择上，我们选择了NodeJs。我不得不承认，这是当下比较酷炫吊炸天的技术。

但我们做出这样选择的真正原因，是看好NodeJs在“敏捷”这一点的优势----“一经修改，立即生效”。这和我们的需要一拍即合。

3、后端----服务化