前端体系建设之上下连接

前端体系建设从前端开始专职发展就慢慢开始演进了。过去，我们习惯描绘一些细节技术作为前端体系所需要的，而没有看到背后的连接，所谓建设之道，即是建立向前向后的连接。前端在项目研发中的上流是交互视觉，下流是后端研发。这两个角色之间的连接对我们来说至关重要。

本文主题着重在团队与团队之间的体系，即上下流团队之间的连接。

与设计师的连接

对于前端的前端是交互视觉，交互视觉在我们固有的理解一定带着『不可控』的因素，即便我们定了交互视觉标准。

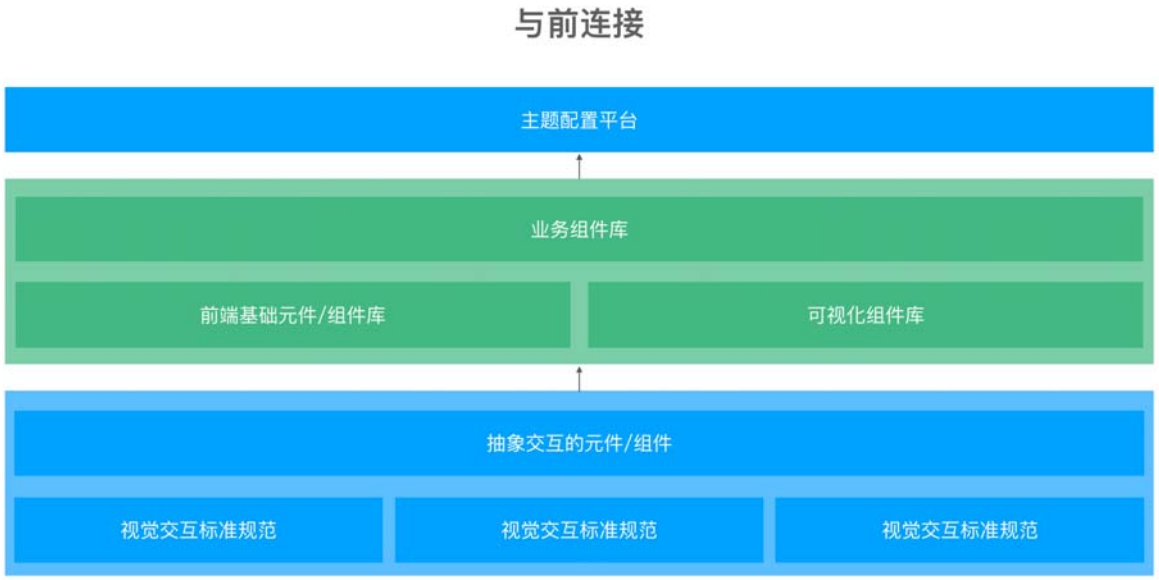
沉淀组件体系标准

我们分解系统的基本交互逻辑，一定由基本组件构成。我们完全可以固化下来沉淀成基础组件库。对于基本组件而言，我们确定一个『可变』范围，包括底色，边框色，圆角，字体等，梳理出一套『样式变量』。那么就在可视范围内，我们可以调整出多套基本交互视觉模板。

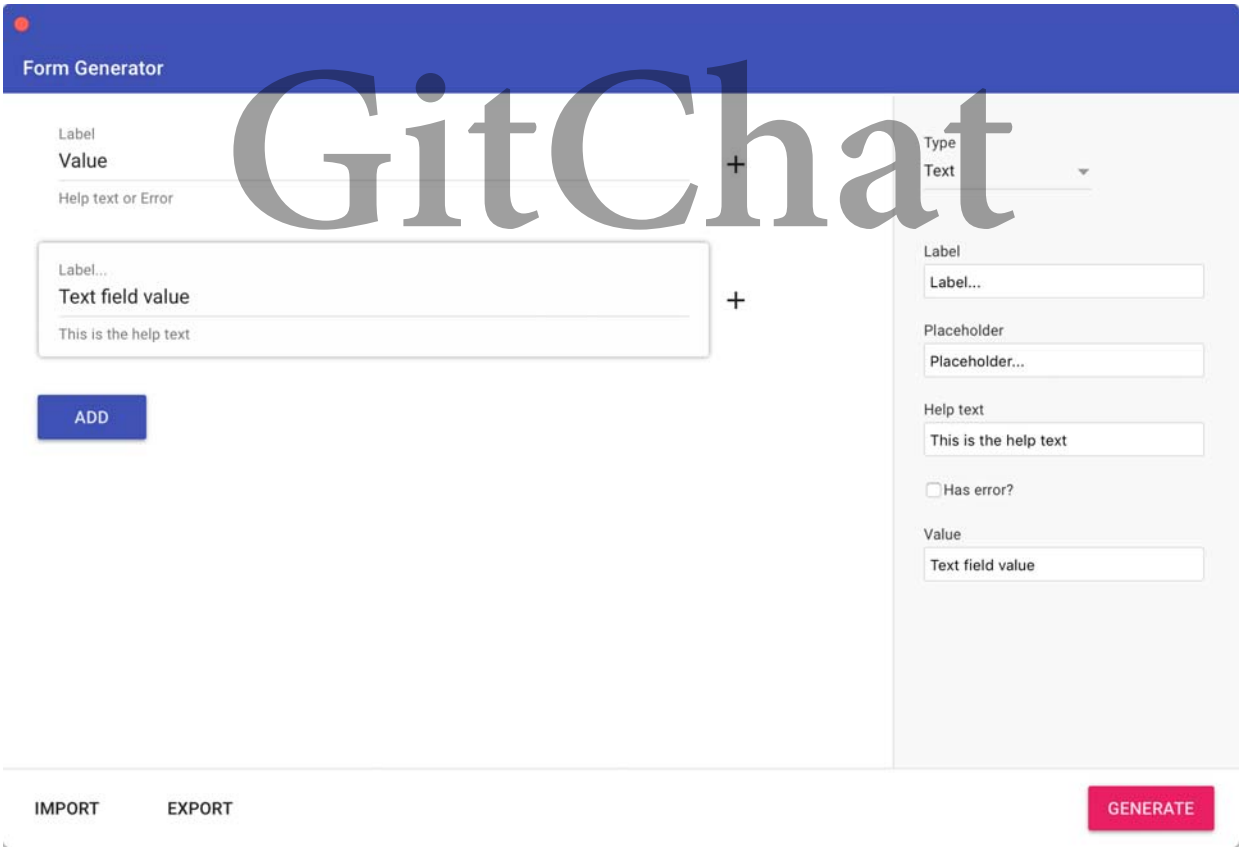
不同形态的产品用户体验心流是不同的，所形成的交互视觉会有所不同。因此，对于前端来说与之连接一定是在同一个交互视觉规范下。这个范围的界定可以通过用户体验心流一致来判断，比如操作控制类，数据展示类等。今天，设计师们一般会将产品设计理念沉淀成一套设计语言，如 antd 会针对后台应用类。

这时，我们必须要考虑背后的问题，我们怎么针对不同的交互视觉抽象组件。antd 背后的 rc-component 就高度抽象了组件行为，让上层 antd 去选择用怎样的交互。形成很好的组件体系，也很方便的让其它前端基于此新建一套设计语言。

因此，抽象交互的组件体系结合不同的视觉交互标准规范就可以形成一系列组件库，从而构成前端组件体系，而其中的样式部分经过我们的抽象可以成为主题配置的要素，最终形态是形成便捷的主题配置。当我们的业务选择生成一套组件的时候可以同时生成交互所需要的组件模板，非常方便地让视觉完成页面和应用级别的工作。当然，同个团队对应不同的视觉交互风格是极少见的情况。



material design 是很好的设计风格。它本身有一整套前端组件库，google 设计师同样也基于 material 风格开发了 sketch plugin 意在生成复杂的 material design 风格的交互控件。这样就非常方便做视觉到前端的连接。



更进一步 sketch 也许可以再生成模板代码。这是良性化学反应。

更好的相互理解

当然，在与交互视觉的连接中，前端的一部分工作像是对它们的『翻译』。但工程师本身对于设计，甚至美感是相对不擅长的领域，但我们在工作中常常遇到设计师的天马行

空，对于前端开发成本较高的情况。对于这种情况，今天很多设计师会去学习 CSS。

另外，像可视化图表其实带着非常多数据分析属性在内，很多时候包括产品、交互都很难选择，或选择对一款可视化图表在当前的业务中。

因此，对于角色本身来说，并不完全都是正向连接的关系。前端也有义务帮助设计师更好的理解页面的构建方式，可视化图表的正常使用方式等。

与后端研发的连接

前端与后连接的是后端研发，现在应用为了高效的交互体验已经极少使用同步请求，因此，我们与后端的连接主要在于异步接口定义上，还有一部分是页面初始化的页面模板渲染两个部分。

接口定义对于前后端之间是一份契约，将两个角色团队的人连接起来，高效协作。对于传统前后端交互，为了并行开发的便利，会在项目前相互对接口进行约定，然后各自开发，最后在预发环境下联调正式接口。这个过程有几个问题。

数据模型

由于前端对于数据的理解不是基于底层数据，而是基于界面元素。尽管前后端对于接口的约定在项目一开始就会约定，但后端对于最终数据的来源存在不确定性。字段及格式在总是会有不少调整，对于项目来说带来的很多风险。

对于通用业务，前后端倾向于均抽象一套实体模型。那么前后端只要维护这一套数据模型就可以实现无调整即可上线。这时配合 GraphQL 感觉会有『彻底』的前后端分离，更好的是非常适合来做数据适配。

单独考虑数据适配层，对于前后端都可以来做这一层，需要做一些取舍。考虑到了发布成本，数据、后端，前端发布的成本中前端一定是最小的。一旦数据或后端出问题，前端可以快速适配作应急处理。因此，在现实当中，常常由前端来做一层的适配。

对于个性化业务，只能 case by case 来约定接口。这时，我们需要一个 Mock 接口的服务来作模拟接口，实现对线上接口的代理。通过平台与本地工具无缝切换，同步这个过程。这么做为了最小化联调的时间。Easy Mock 平台就是一个不错的开源选择。

跨端适配

由于不同的终端产品与研发的视角均不同，研发对于数据的理解也会不同。例如在无线端数据要求更精简，而 Web 上一般没有这样的要求。

那么我们想到对于底层数据都是相似的，但上层应用接口却有差异。那么，我们就让原来面向应用的接口下沉形成的微服务，提高抽象性与稳定性。而在它的前端引入一层

BfF（服务于前端的后端）层。这种模式不会为所有的客户端创建通用的接口，我们可以拥有多个 BfF，对应于 Web、移动客户端等。

这一层的好处在于后端主要的关注点下沉到微服务架构，如果其中一个服务要迁移，那么其中一个 BfF 就可以调用新服务，其它的保持不变。进一步提高了系统的解耦。

当然，这种模式的问题在于谁来维护 BfF 层。如果由后端来维护这一层那么选用一种轻量级的语言会加快研发的效率，比如 PHP。如果由前端来维护这一层，那么 Node 就非常适合。对于接口服务来说主要是高 IO 场景，正与 Node 优势相匹配。这一选择取决于团队人员的配比。

模板管理

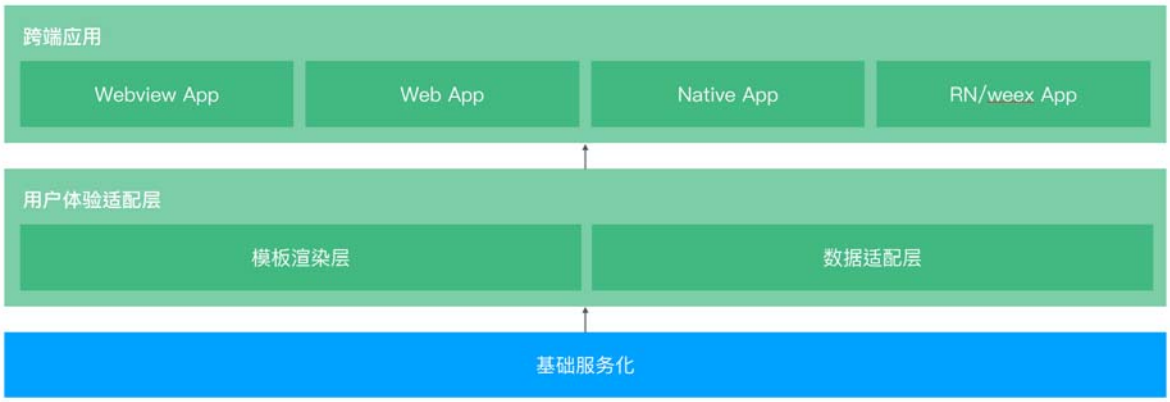
在过去接口服务还不多的情况，前后端主要的连接就在于模板。

对于后端来说会把系统变量直接写在模板中，这对于前端维护来说就带来了很多困难。对于前端来说，我们的脚本与样式的配置都在模板当中，常常因为发新版需要改地址。

对于耦合这么严重的这部分。对于今天前端来说，已经慢慢找到自己的方法。就是通过 Node 来管理这个模板，将模板做成通用服务，后端会把系统变量发送给服务，服务来完成后续的模板生成的工作。

这带来的另一个好处是我们在做 SSR 时只需要将逻辑在这一层当中完成即可。

与后连接（数据传递标准化）



那么，在理想情况下，我们会抽象一层『用户体验适配层』来做两个工作，模板渲染与数据适配。这也是今天前端工作外沿的一个表现之一。对于后端来说，这一层相当轻，对于前端来说，这一层对于我们的连接显得更加顺畅。

全链路稳定性保障

产品的数据化运营，我们都耳熟能详，一般体现在运营效率的突破，商业模式的创新，客户价值的提升等。

那么再来看商业价值的背后是产品的稳定性，从产品角度稳定性链路是贯穿在交互行为的链路中。其中，一系列的交互行为引发了一系列的前后端的请求，后端与数据库的请求。一般我们称之为端到端的稳定性。

端到端的监控，除了我们必须要制定好详细的规范以外，重点看的是每条链路上的性能，并能够在最短的时间内知道哪里的链路出问题了。做法比较简单，前端发送的每一个请求都会带上一个 traceId，这样就可以马上跟踪到是后端的哪一个类，并关联到是哪些 SQL 的调用，哪些关联表位于哪些机房中。

正因为有这样的全链路保障，我们不同角色之间相互独立，又相互连接在一起。成为产品保障的基石。

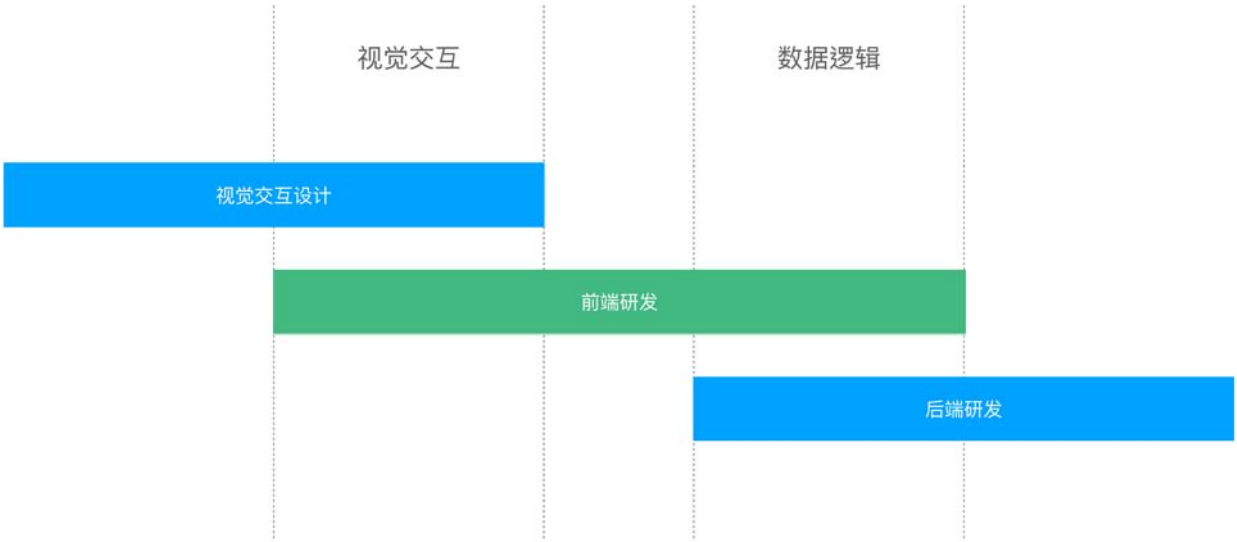
更好的相互理解

诚如设计师，与后端研发合作同样涉及到如何帮助后端更好的理解前端。

前端与后端的理解差异往往来自于数据接口上。后端是服务器环境，前端是浏览器环境，往往两者处于的立场不同，不能感同深受。前端考虑的是用户体验增强，因此，为了在数据传输大小与次数上都有自己的考虑，这时候与后端的工作就有关联。

因此，现代前端也更倾向于将抹平这一层差异的工作放到『中间层』来做。为了做更极致的前后端解耦与性能优化。

总结



软件工程中，每一个角色都有其上下流，与上下流之间处理好关系是工程体系中重要的工作。而其中我们要做到最好的解耦，勿必要：

1. 理解上下流角色的关注点是什么，帮助他们理解我们所处角色的需要。

2. 做好规范化设计与标准，减少沟通成本。

3. 一切以角色人员结构作最优配置，并不断升级配置。

最后，说到被今天在前端圈被神话的全栈工程师，我认为并没有脱离前端的范畴，只是为了更好的为前端角色来工作的延伸角色。在角色结构配置不断升级的情况下，每一个角色可能都是全栈的，因为了解痛点，所以必须延伸。