前端小团队如何培养技术积累？

引言

前端是个挺特别的岗位，一方面它的技术栈更新几乎是软件开发领域中最快的，但另一方面它的不可替代性相对而言却并不算高。并且，虽然多数企业都有相对独立的前端团队，但团队多半都有不少业务负担，加上前端较高的迭代速度，一线业务同学的成长多少容易遇到些瓶颈：需求都做不完了，还有时间读源码解析吗？

其实我们都知道，个人成长和团队成长是分不开的。一方面在技术氛围较好的团队中个人的进步会更快，另一方面团队的技术积累也是由一个个成员们贡献出来的。这就引出了我们的问题：在业务团队中，有什么方法能让个人和团队在迭代中都得到更好的成长呢？

能想到最简单的答案应该就是「定期技术分享」和「不加班，给大家更多的学习时间」这样的吧。不只是前端，相信这肯定也是广大开发同学们喜闻乐见的。可惜业务压力摆在那里，如果个人学习影响了短期内的团队产出，那么老板们的脸色也许会有些微妙……从另一个角度上来说，前端同学提升技能水平所需学习的各种模式、框架、类库、工具，在开源社区都有非常详尽的文档和教程，如何提升个人能力的话题也更不是本文所能覆盖的。不过，如果一个前端团队能够通过一些技术层面上的方式，培养出由团队成员**共建的**良好技术基础的话，相信老板们和开发同学们都可以接受吧。这其实也正是本文所关注的。

积累的方向

在介绍「怎么做」之前，我们不妨考虑下「做什么」，即在什么方向上去培养技术积累呢？

全栈

我们知道 Full Stack Developer 的概念一直很火，那么按照 Full Stack 的概念，前端团队的所需的技术积累是怎样的呢？这意味着前端团队需要横向地扩展自己的能力，拉通 DB 到 HTML 的流程。技术栈覆盖面广自然是好事，可是为什么还是经常能够听见对全栈的争议呢？在很多需要快速迭代实现原型的场合，这样的能力模型是很合适的。问题在于在一个系统的开发中，多数情况下团队成员负责开发的是独立的模块。而按照信息隐藏的理念，而只有在代码模块采用定义良好的接口来封装，模块的内部结构对外部不可见，复杂度被屏蔽而非暴露在他人模块内部结构前的时候，开发的效率才是最高的，也并不是团队中所有成员都需要了解系统整体的技术细节。并且，前端本身也是对开发职

能分工的细化，在后端有成熟稳定团队的前提下，前端在团队层面按照 Full Stack 的方向积累技术，在投入和产出上未必是最优的。当然，这和许多大厂「只招全栈」的理念并不矛盾，毕竟在个人开发者层面的全栈所真正要求的并不是杂而全，而是高效地解决各类问题的学习能力，以及对技术更加全面的深入。

全生命周期

在团队层面，和 Full Stack 相对的其实有一个 Full Life Cycle Engineer 的概念。这个理念实际上鼓励的是让前端同学去更全面地参与产品的生命周期，并在生命周期的各个阶段去更好地支持业务。这个情景下从项目的原型中前端就会以提供 UI 组件的方式参与，在实现了具体需求后也会参与到测试和发布的流程中，而服务端渲染等特性也要求前端去支持起一定的运维职责。这种模式下，前端的职责并不仅仅局限在传统的客户端 HTML + CSS + JS 中，而是一系列与用户交互相关的技术合集。这个背景下，可以把前端团队理解为一个垂直的功能性的团队，技术积累上也是更多地围绕着我们所面对的业务问题去做沉淀。套用《人月神话》里外科手术式团队的概念，如果说全栈的角色更接近于全能的「外科医生」的话，那么 Full Life Cycle Engineer 则更接近于纵向深入的「代码专家」。

围绕着这个理念展开，不难发现许多技术点是在团队层面上可以去做积累的。而具体到落地，就更需要鼓励团队同学们来解决比常规业务开发更难的工(zao)程(lun)化(zi)问题了。前端轮子是公认的多，几乎所有常规开发所涉及的领域都存在非常多的现成轮子，那么是否有必要去积累团队内部的轮子，又在什么方向去做呢？这其实视团队情况不同，是非常业务驱动的。比如，维护 C 端产品的团队，可能更需要性能优化、监测告警方向的轮子，而开发 B 端中后台产品的团队，对状态管理、统一组件库一类的轮子需求度则会更高。在深入业务的过程中，几乎总能找到特定的场景能够抽取出特定的复用模块，或找到适合针对性优化的地方，这其实就相当于技术积累的起点了：**在业务驱动下开始造（甚至发明）团队内部的轮子。**

关于造轮子

其实造轮子对技术积累的促进，并不仅仅体现在实现轮子这件事本身。比如，即便是一个非常简单的 UI 按钮，在将它实现为可复用的代码时，所需要做出的工程化努力都可以是很深度的。比如，即便一开始只在团队内部使用，API 设计得简单没有关系，但作为一个可复用的模块，怎么样让使用者不需要读源码就能用呢？这时候我们需要最基本的文档；怎么样保证升级后 API 稳定呢？这时候单元测试就体现出了意义；发布为模块的话，样式怎样和组件一起提供呢？这时候需要的是构建流程……这些和基本业务逻辑并不直接相关的工程化内容，即便只是开发出一个简单的内部轮子时都可以渐进地去考虑和实现。并且，这些**工程化的特性也正是做技术积累的良好切入点**。如果编写的代码都是不需要复用的「一次性」代码，那么文档就是多余的；如果实现 UI 逻辑只是为了满足基本的业务需要，那么实际上是没有什么机会和必要去将单元测试落地的；如果最终打出的包始终只是面向用户而非面向开发者，那么甚至也不需要考虑 `dependency` 和 `devDependency` 的区别…工作内容限制在一个狭窄的子集内的时候，成长显然是会受限的。

当然了，以上引入的工程化特性客观上都需要时间投入，也不免会有重复造轮子浪费人力之嫌。这时如何去做权衡和取舍呢？

动机和理由

我们不妨评估一下引入新轮子的投入和产出。比如，分析一个没有积累公共组件或依赖第三方组件库的团队，是否有开源项目中未能提供的 UI 组件，这些组件是否有复用的可能性和条件呢？如果有的话，引入工程化的发布流程会带来多少时间成本，是否能够方便其他同学在后续的使用中节约回来呢？再比如，是否存在业务场景是现有的开源轮子不能完全符合需求的？如果自己动手，能够收获多少易用性或性能上的提升呢？毕竟造轮子也是工程，而工程问题总是可以评估、分析和 Tradeoff 的。

实际上，前端领域中，由于业务场景的多样性，开源轮子不能完全匹配实际需求的情况是很常见的。比如，Redux 是个用来支持复杂应用的状态管理库，在增查改删的后台应用中使用起来十分沉重；实现不需考虑兼容的简单页面时，自己封装出的简单 DOM 操作库肯定比 jQuery 或 Zepto 更加轻量；封装登录认证库时如果全部交互都通过 JSONP，那么依赖 fetch polyfill 或 axios 的成本还不如直接实现……熟悉业务后，这样的场景总是容易发现的。

但是光有动机，并不能成为决策的充分理由。一个团队所面对的挑战更绝不仅仅只是造一两个轮子就能够解决的。这时候在团队层面，讨论出一份 Todo List 通常是很有帮助的。我们可以通过讨论，整理出一个团队目前需要所需要解决的技术问题的任务列表，并按照「是否重要，是否紧急」的四象限方法梳理出任务的优先级，再去根据列表中各个任务的重要、紧急程度，来决定如何安排团队的资源投入。毕竟不论轮子的设计是否简洁、代码是否优雅，只有能够更好地解决具体问题，能够更广泛地被业务所使用，造轮子才能够获得更多的产出。

如何启动

在决定了具体方向后，我们就可以开始动手了。对于怎样实现业务逻辑，相信同学们都已经很有经验了。那么，怎样从头开始造一个轮子呢？这里面从 0 到 1 的过程和经验可能也是一些刚组建或者积累较少的小团队所较为欠缺的。

实际上，从 0 到 1 的过程，其实就相当于一个实现 Proof of Concept 的过程。一个 POC 原型**只需要实现新轮子最核心的一两个特性就足够了**，实现上并不需要考虑得大而全。并且，面向团队内部使用的轮子一般也具有较高的定制性，不需要过多地按照普适性解决方案来设计。在实现这样定制型较高的的模块时，通常能够带来设计上的简单性并更好地节约时间。这方面 SalesForce 的经验可以作为参考：在实现一个系统的时候，如果让各种流程和参数都充分地可定制，那么整体上开发所需工作量可达到简单实现的 2-3 倍。其实，在内部面向业务需求定制的组件如果追求灵活性很高的设计，那么 API 设计通常会更加复杂，从而使得每次在使用时也会相对地更加复杂（例如能够自由传入子组件的 React 组件，其复杂度和使用难度都比普通的组件要高不少），这时开发组件 + 使用组件的总体投入甚至有可能小于代码模块化的产出。更加可行的方法论是通过权衡实际业务场景，本着 KISS 原则去来保证实现上的简单性，同时轻微地在正确性、完整性和一致性上做一些让步，这样会更容易产出和维护可复用的模块。同时，许多工程化内容

的落地难点实际也是工具引入上从 0 到 1 的一次性投入，例如在从 0 到 1 跑通第一个测试用例后，添加和完善用例的难度就小得多了。并且，实现这些特性的过程中通常也能够接触到更新和更完善的工具链，将内部轮子中小规模运用并验证的的新特性再去逐步推广到业务项目中，也是实践中可行的方式。

维护与管理

然而，光有一两个由核心成员业余开发出的类库，并不一定能够促进团队整体的技术积累。在一般的模式下，可复用代码的开发通常会交给团队中经验相对丰富的同学，而这些同学本身很可能还有更重要的业务问题去解决，这时候即便有了一次性的产出，后续的维护和迭代也是一个问题。为此，我们探索的模式更倾向于**基于 Monorepo 的共建**。什么是 Monorepo 呢？这是一个管理机构代码的方式。它摒弃了原先一个模块对应一个 Repo 的组织方式，而是将所有的模块都放在一个 Repo 中来管理，这也是目前 Babel / React / Angular / 等流行项目所应用的开发方式。早期的团队项目中往往会将各个模块仓库拆分管理，形成 Multirepo 的模式，这时对于各自独立的业务项目没有关系的，但对于复用的模块而言，这种模式主要会存在这样的一些问题：

- Repo 与 Module 需要拆分、迁移或整合时维护上的困难，尤其在涉及交接时。
- Issue 缺乏集中的管理。
- 模块间依赖造成版本发布时，对各个仓库手工维护 link / lint / test 时的繁琐操作。
- 新同学接手开发或维护时的较高复杂度。

而 Monorepo 的方式，则是将团队内积累的可复用模块整合至同一个仓库中来维护，这样在团队同学有维护和改进的需求时，跑通环境而后 up and go 的流程会得到很大程度上的简化。以组件库为例，不同组件之间可能会存在相互依赖，而 Monorepo 就在依赖配置上有更好的支持，也易于整合提供更完善的构建工具，简化发布的流程。这时，共建的模式就从「个别同学作为上游承担所有需求」，转化为「提供一个完善了构建和测试的基本框架来实现共建」。团队成员可以根据业务情况抽取出公共组件整合至组件库内，这个开发新组件的过程相对于自己另起炉灶会更加简化，并且更重要的是在这个环节中，一般性业务开发中经常缺失的单元测试和 Review 是真正有机会去实施的，相对于对业务代码的 Review 经常由于进度问题而难以实施，在**可复用组件的层面去做 Review 是更加有意义的**。同时，内部的组件库对 bug 的响应也能够更加迅速，依据 SemVer 高效发布 patch 版本也能够更好地支持业务。在组件库使用范围为团队内部的前提下，组件库外部依赖过多造成业务方版本更新时出现问题的可能性也会小得多。

除开用于基本业务开发的组件库，Monorepo 里还能够根据实际的业务情景，装下许多应对团队其它需求的内容。比如，对于经常需要开发临时性小项目的团队，整合了内部服务的 boilerplate 脚手架就是一个很好的可复用模板。不过这里较为特殊的一点是，脚手架的配置一般需要暴露，且可以假设使用者熟悉相关的配置细节，在 Webpack 等工具的使用方式几乎已成为通用领域知识的前提下，对成熟工具进行二次封装的内部工具在学习成本和灵活性上比较容易遇到问题。这里我们一方面可以保持组件库的小而美，另一方面则可以结合 repo 中的其他复用模块，精简脚手架中的代码到其它模块中。一般在实际业务场景中，脚手架所整合、接入的一些内部服务很容易在不同业务中暴露出一些问题，这时候使用脚手架的团队成员也很容易通过 PR 的形式将 bugfix 同步回原仓库，这个流程的参与门槛也并不高，但不能小看维护者增长对项目长期稳定性的帮助。作为一个

小技巧，我们不妨去除掉每个源码文件中自动生成的 `@author` 注释，代之以在根目录中放置开源项目中常见的 `CONTRIBUTOR / AUTHORS` 文件或 `package.json` 中的 authors 字段，这里的关注点就从「背锅」转化为了「参与」。

实践经验

我们的团队组建时间并不长，主要的业务场景是开发和维护较多的 PC 端业务系统，这里分享一点技术积累的产出来抛砖引玉吧��

脚手架

由于一般而言团队内部基本上都会希望能使用统一的编码规范和技术栈，这时候脚手架其实是一个很不错的切入点。脚手架也是我们最早实现、业务使用最广泛、同时有最多同学参与维护的轮子。看看国内前端社区铺天盖地的「xx + yy + zz 项目模板」就知道，实现一个脚手架的难度并不高，但就提升开发效率，节约团队同学时间而言，它确实有在团队内部推广的价值。在开始阶段的脚手架其实就是一个 Koa + React / Redux 的 Webpack 模板，随着业务复杂度增加，脚手架中除了整合各种接入的服务外，还出现了越来越多的参数配置。在这个过程中它开始逐渐变得沉重、不易上手。为了改善这个问题，我们除了按照「如何配置以满足常见需求」的方式完善了文档外，还进行了登录服务、代理服务的依赖化，将基础性的业务需求以 Koa 中间件的形式抽离以实现更好的复用。

我们对脚手架本身的定位更偏向于项目起始的模板，而不是一套对 Webpack 的封装。实际上许多对 Grunt / Gulp / Webpack 等工具做二次封装定制出后的新工具，虽然在一定程度上能够更快地开发出原型，但在需要团队其它成员定制、维护时，团队成员对主流工具的使用经验难以应用到这些封装后的工具上，尤其在内部文档不够详尽的时候，这些定制的工具往往更加难以推广。因此，我们在维护脚手架时，更加偏向于将 Webpack 的配置清晰地暴露给使用者，不在这里掺杂额外的 Magic。目前，团队过半同学脚手架已经加入了脚手架的 AUTHORS 名单内，我们希望它能够作为一个长期的积累，得到持续的维护。

组件库

React 组件库是另一个已有一定产出的团队积累项目。组件库的开发除了整合默认 UED 交互，满足业务需求外，更多地成为了一个新技术的试验田。组件库中落地了单元测试、 Publish Hook 等质量保障的特性，也更多地尝试了新工具，更是首先引入了 Monorepo 的管理方式，贡献组件的流程也实现了自动化。目前虽然组件规模有限，但我们同样以提升团队技术积累的愿景在持续维护。

当然了，团队中并不是所有同学都有时间和精力能够贡献新组件，维护组件库也不是强制性的要求。但对于没有参与实际组件开发的同学，也并非没有参与的方式。例如，每位同学可以在自己负责的业务中使用组件库中的组件，反馈问题、提出 Issue，或者也可以交流、Review 组件的 API 设计。开源项目对 Issue 的响应一般在天级，而组内的扁平交

流能够将组件 bugfix 更新的速度提高到小时级，对于业务迭代较快的团队，具备内部组件库的高效响应能力是很有帮助的。

Node 服务

脚手架和组件库都是面向客户端 JavaScript 的技术，而在前端同学需要对所负责的体验做更高程度的优化时，我们的能力范围也需要拓展到 Node 服务端。虽然对于 Node 层的存在曾经有过一定的争议，但实践证明 Node 在性能上对于 Web 服务这样主要瓶颈在于 IO 的服务是足够的，而维护性上「回调地狱」也早已在语言机制的改进中得到了解决。目前对 Node 层的使用，主要目的而非深入到后端所专注的数据接口中去替代后端，而是为了提高开发效率，解决前端服务部署与渲染的问题，提升前端在整个应用生命周期中的服务能力。在这种场景下，前端对 Node 的使用其实是很轻量的，但作为 Web 服务，Node 端同样需要接入许多较为基础的服务，如性能、日志、监控、告警等。和浏览器端一言不合造轮子的方式不同的是，这类内容多数已有成熟的服务，在 Node 端所需实现的主要也是一些接入服务所需的代码。同样地，这些内容也很适合在 Monorepo 中提供可供快速验证的 demo，或整合到团队的 boilerplate 项目中，提供从本地调试到线上部署的支持。这时共建的模式同样适用，并且在团队业务覆盖面较广时，许多服务的接入更可能首先来自于业务同学，而后再整合至 Monorepo 中。这种模式也是很合适的。

总结

从前后端分离到 Node 出现，再到各类框架全家桶兴起，以及现在的服务端渲染和 PWA，近年来前端的技术发展确实相当快速，可谓「唯一不变的是变化本身」。本文中想要探索的其实也是前端团队在技术的演进中积累技术沉淀的若干一些小技巧和较为一般性的规律，希望抛砖引玉能对有耐心读到这里的前端同学们有所帮助 :-)