1.思考方法与领域模型

> 珠峰前端架构师技术分享课 <https://ke.qq.com/course/272058>

为什么要讲思考方法

在市场现有的前端课程中，传授思想的非常少，这就导致大多数培训出来同学：

- *思想局限，总是按照固有的经验思考问题，缺乏头脑风暴的思维涌现。*
- *学习能力较差：基础薄弱，英语水平有限，有问题百度等不好的习惯，导致信息获取的来源和学习方式不良。*
- *大多数人拿到需求后，按照经验、惯性开始直接编程，不去思考，久而久之，我们更不愿意思考。*

算法、思想、基础比框架和技术更重要

无论前端后端，程序的本质就是数据结构加算法

在业务代码中，数据结构指的是缓存在内存中的数据变量，包括类、函数、方法、变量以及它们的继承、聚合、回调、消息通知等关系。而算法并不是你熟知的排序、查找等算法，任何解决问题的方法或步骤，都称之为算法。也就是我们数据结构的处理函数，称之为算法。

在架构设计中，根据经典的算法总结出的经验，往往被称为**编程设计思想**，比如经典的设计模式、分解参数列表的柯里化、大家mvvm框架中熟知的依赖分析和dom-diff、以及react-16使js原本只能阻塞执行的代码变成不阻塞的任务调度(fiber)等。

前端的大多数思想都是借鉴后端已有的思想，当然也有大量的前端原创，比如新版fiber的任务调度(调和)算法。

认识到问题的本质并设计出解决这个问题领域的方案

如果想设计一个框架，或者成为架构师。需要我们拿到需求，养成思考的习惯并深度思考，思考的方面包括但不局限于：会面临哪些问题，市场现有的解决方案有几种，有哪些优劣等，**这需要我们对比现有方案，设计最优的解决方案，思考远比代码重要的多。**

写代码之前，需要设计好领域模型图。

前端多为事件驱动，不适合传统的uml图，可以用模块关系图或者时序图表示即可。

然后思考模块与层级的关系是否正确，模块是否高内聚并解耦合，层级是否单向逐级调用并且没有出现抽象渗透、各个时序会不会有数据冲突等等问题。然后再开始编码。

这就需要我们必须找到适合自己的一种思考方法：

1. 思考方法

1.1 黄金圈法则 <https://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action/t

ranscript> （翻译链接 <https://open.163.com/movie/2011/7/0/A/M78065A8E_M7806OF0A.html> ）

这是一种面对问题的通用思考方法，不仅限于框架设计。

绝大多数人拿到需求，会根据经验，列出需要完成的功能点，然后制定方案并且完成工作。

经验往往禁锢我们的思想，让我们机械化的按部就班做事。尤其在忙碌的业务开发中，我们更不愿意去思考，往往会产生以下结果：

- 无趣：厌烦重复劳动的工作，觉得没有创新与挑战，想要摆脱环境。
- 模仿：别人如何实现，我就照着实现。
- 缺乏思辨：领导或者别人交代的事情、说过的话等，很少思考原因与正确性。

而世界上成功的领袖、创造者，比如乔布斯、莱特兄弟、马丁-路德-金等人的思考、行为、沟通方式，都完全一样，
但是与其他人的方式完全相反，他们不会使用经验列出所要做的事，而用被称为“黄金圈模型”的思维方式思考与行动。

其实我们每个人都知道自己做的是什么，如何工作的，但是我们很少思考为什么要做这件事，为什么要这样实现，可能面临什么问题...。

黄金圈法则2011年由西蒙·斯涅克提出，多年位于ted排行榜前几名经久不衰。

内容很简单，从what->how->why转向why->how->what的思考顺序。

1. 思考为什么要做这件事
2. 思考我们要达到什么目标和结果
3. 思考如何做这件事

1.2 头脑风暴法

- 在设计之前，使用自己的之前的经验，想出几种现有的解决方案，思考他们优劣与联系、区别，以及为什么会使用这种解法。
- 思维发散，给自己完全没有障碍的空间，不要受之前的理论束缚。
- 然后再去想具体的可能面临的问题和特殊情况，以及如何解决这些问题。

比如setState怎么实现页面的更新、vue怎么如何实现界面的更新，两种api可能会遇到什么问题，当初作者为什么这么设计，有没有更好的解决方案。

想的越全越好，不要怕过度设计，一切过度设计或者对外提供不友好的api都可以通过语法糖或者类似ts的语言包装对外提供。只有没有思考到的地方和设计不全的框架，之后可能面临所有都要推翻重做。

1.3 问题导向法

- 仔细思考面临的问题的本质或者突发事件，如果已经了解问题的本质但又不可避免，从投资收益比上思考一定要解决这个问题吗？可不可以绕开。
- 解决问题会产生什么良好的结果，有几种方案能导向结果，会不会引发其他side-effect(副作用)，如果会引发其他问题，是否可以解决，当解决完所有可想到的问题，把方案推入队列。
- 对比队列中的几种方案，把每种方案的元素建立联系，对比每种方案区别，并从曾经实施过的方案使用者角度思考，他们经历过哪些失败方案以及相关解法，想一想作者为什么这么解，然后记录每种方案的耗时以及风险。
- 从中选出耗时以及风险最低的方案，思考是否可以优化，最后执行方案。

2. 领域模型

2.1 问题域

- 思考的内容：我们应该做什么，解决了哪些行业痛点
- **不要只完成需求，这样无论从市场还是从业务需求变动来说，功能都非常局限，不得不频繁修改代码。**
  - 有些人常抱怨客户提出的需求难以理解、经常变动，不合理性要求、需求分析不好做等等，渐渐也就失去动力、兴趣，为终日的编写、修改需求报告所累，大部分情况下是被客户引导着工作了。
  - 而有些人往往是站在客户的角度、走在客户的前边、引领客户工作，的确是一件很有成就和开心的事情。
- 尝试把问题抽象，然后画出dsl图，建立领域模型，形成业界统一的解决方案。
- 考虑到特殊情况、突发情况等异常流程，找到变化，封装之。

2.2 解决域

2.2.1 看到问题本质：

- 理解原理，也就是内部实现，以及作者设计的目的、想法。
- 能推断出类似问题的实现方式或最优解。

2.2.2 主动思考的实例：

使用框架的原因

- 解决现有痛点
- 统一团队开发规范
- 加快开发效率

web框架的本质是什么

- 数据结构：内部数据维护的结构，用来保存对象的数据、生命周期等
- 算法：内部程序对数据结构以及用户配置等的加工，得到结果
- 框架提供方法，内部调用。

框架和类库的区别是什么

- 框架：大多数内部函数框架主动调用，对外暴露配置
- 类库：大多数内部函数用户主动调用，对外暴露接口

ng/redux/vue这些框架解决了哪些前端痛点

- 在mvvm之前的时代，前端页面元素的展示与变化经历了静态页面、操作dom与模板三个时代。
- 操作dom性能底下，代码繁琐。模板更新时，由于没有依赖分析机制（ng叫做脏值检测，react称之为dom-diff），会全量解析template的innerHTML，非常消耗性能。
- 以react的出现为例：解决了数据难以管理（flux/redux）,无法实现局部更新（dom-diff），无法统一浏览器兼容（如事件差异）等等。而组件的复用早在ext时代就已经实现，template时代已经很完善，并不是react所开创的领域，只不过react的jsx完成了一种更好的方式。

为什么目前业界流行的框架，都是三个层次。

- 三层之间的通信最为简单。假设有4个点，中间会连6条线，如果超过4个层次，层次之间的解耦将会变得十分复杂。
- 有些层次过深，非常容易出现高层次需要获取低层次的数据，比如网络七层，用户需要在http层必须获得数字链路层的信号，会出现链路层的接口向上层层传递。这样的设计代码十分冗余，被称之为抽象渗透。

组件和类库的基本区别

- 组件不会离开框架，组件会带有框架所包含的生命周期、事件回调等等信息。而在同一个语言中，无论是函数还是类，都可以放到其他库或者框架里去使用，这种称之为类库，而组件不可以放到其他地方使用。

我们通常所看见的软件架构图，都是组件在最顶层的，为什么react组件涉及了所有的层次。

- 层次与功能是不同的两个内容，数据层、逻辑控制层、展现层是层次，而事件、事物、组件是具体的功能。
- 层次应该屏蔽下层细节，再对上层输出。只能单向调用，没有交叉依赖。比如写组件时，不关心真实dom元素，只关心虚拟dom。
- 我们之前设计的事件、视图处理包括dom diff，都是框架的组件，也就是功能，并不是层次。 组件并不等于层次，组件会跨越到所有的层次，组件与所有内容都要打交道，如事件、视图、数据
- 层次势必会出现(抽象泄露)[https://zh.wikipedia.org/wiki/%E6%8A%BD%E8%B1%A1%E6%B3%84%E6%BC%8F <https://zh.wikipedia.org/wiki/%E6%8A%BD%E8%B1%A1%E6%B3%84%E6%BC%8F>]，比如我们虽然不关心dom，可有时候就需要拿到渲染出来的dom元素。这需要底层对外暴抽象细节，也就是查询dom的方法。

diff机制

- 经过判断和操作两个步骤。
- 判断原理
  - 如果有key，根据key查找，没有key，根据index查找。
  - 查找后判断组件type：
    - 如果是原生，判断tagName
    - 如果是react组件，比较displayName
- 操作原理
  - 如果新老一致，执行更新操作
  - 如果新老不一致，且新的有，老的没有，则插入
  - 如果老的有，新的没有，则删除
  - 如果新老都有，类型不同，则先删除老节点在插入新节点
  - 如果顺序不同，则按照老的顺序遍历，当老的顺序遍历完，所有新节点剩余节点全部重新插入。比如abcdef改成了bcfdea，则bcf在老节点都能找到，而f已经是老节点最后一个节点，所以之后的节点dea都是重新插入。

设计框架，优先考虑什么

- 先从用户语义出发，满足用户需求，优先其他一切内容
- 势必要牺牲性能或者代码可读性、接口规范化等一些指标

fragment这种把框架提供的功能自己又封装了一遍，算不算抽象泄露？

用户而不是把fragment当做组件，而是一个语法糖，并没有组件与生命周期等完整的内容，不算抽象泄露。

为什么框架需要设计可扩展性？

为了满足用户需求，框架会妥协一些功能或者语法糖(软约定)，这让框架越来越难以维护。所以设计框架之前，就应该考虑把会变化的部分做成可扩展。

为什么dom-diff算法再快，也没有vue的模板快。

模板可以做静态编译（依赖分析），在渲染之前就知道有哪些数据变化会影响界面。
而react也可以分析render函数，把变量渲染出来的，给个标记。数据变化时候，分析标记的数据，只更新标记数据变化的组件

为什么vue2.0引入了jsx？

模板解析最终解析成virtual dom，能屏蔽最终的实现问题。
jsx能更好的表达用户的循环、判断语法，能更好的表达用户的语义。

真实dom对比和虚拟dom对比与用两个虚拟dom对比的区别

当存在数组时，虚拟dom和真实dom层级结构不同。所以不能用虚拟dom tree和真实dom tree对比。

react的原生组件和复合组件嵌套问题

方式一：react组件维度划分类，属性保存原生组件以及结构。
方式二：react和原生组件维度划分类，children包含下一个组件。

- 方式1：内存占用大，接口统一，粒度精细，递归一致，为了更精确的控制，能实现其他的需求。
- 方式2：内存占用小，组件对比不可打断，不适合fiber。

设计框架，需求、性能、代码量应该按照什么顺序考虑

优先考虑用户需求，然后是性能，最后是代码量多少

setState放到数组里面处理

为了实现统一修改的能力，一定要把setState放到统一数组里处理。
事件，开始时候，setState，加入batchUpdate队列，结束时候，统一执行state更新

Fiber

大型耗时的计算时候浏览器会卡顿，无法响应用户的操作

什么是大型计算

比如页面的渲染，动画的渲染，dom-diff等

哪些过程可以被打断？

dom-diff可以被打断，这一次的状态和上一次的状态无冲突，可以被打断

任务调度，如何保证低优先级的内容不会永远被卡顿

每个任务设置过期时间，当低优先级的任务快过期了优先执行，fiber采用了过期时间

实现fiber引擎

用过期时间实现队列，任务小的片段，主动return掉，看看有没有需要响应用户操作的地方，如果有，就打断，执行用户操作。