前端开发的敏捷学习法

大纲

- 选题背景
- 敏捷学习法
- 知识迁移
- 信息获取与筛选
- 新形势下的阅读方式
- 社区化学习
- 工具&实验田
- 时间管理
- 总结

选题背景

在纠结选题时候，我一直在思考分享什么知识比较好，平时在工作中确实有许许多多经验和技术积累可以分享，但是我发现这些技术在互联网上或多或少都有相应资料，自己平时遇到各种问题也是去探寻的。于是我觉得不如聊聊面向未来的学习方法，有时候具体讲一个知识点不如告诉你如何自己在资料海洋中自我敏捷学习。

每个开发者都是从小白走过来的，这条路非常坎坷。有的人花了较少时间就达到了架构师级别并且带领小团队有序开发，有的开发者则花了5、6年时间还在基础岗折腾。抛开个人努力程度看，学习方法也是至关重要的。说到学习方法，从小到大我们学习过很多知识，可真正在工作中用到的却是很小的子集，留下的是学习知识的能力。目前行业趋势迫使大家不断跟进新技术，需要较高学习能力的人。我们研发WebIde时，团队并没有所谓的ide工程师，国内同类产品就是空白，一切只能靠我们上海5、6个人各种摸索学习在短时间内做出产品。产品很出色但是过程却是痛苦的，新问题新难点不断涌现，这时除了努力更需要的是解决问题的能力。如果你能拥有这样的能力，即使你不会某个具体技术点，那并不重要，因为你可以短时间学会，并同时夯实基础。这才是面向未来的人才。

在技术瞬息万变的今天，原来那些只靠学历、人脉、经验升职已经过时了，要想保持自己的职场竞争力，练好一件本事就够了，那就是学习敏捷力。 —美国著名企业高管教练瓦尔库。

我以前端开发为例，社区技术派别已达到了百家争鸣阶段，众多公司推出了自己的最佳实践以及相应的开源库，学校、培训班、出版社尽力的跟随。在这种情况下，很多朋友很困惑，按照传统一般的学习方法，似乎很难应对。选资料或书籍就是一个大难题，读完别人又告诉此版本已升级或者书中有很多错误，甚至工作中已经不再使用了。再者，平实工作任务很紧，没时间让你慢慢啃。还有很多知识要你同时去学。这时候很多人开始烦躁、抵触，迷茫，怀疑技术，怀疑自己。而恰恰这时候需要的是耐心，走一点，停下来，仔细思考自己的学习方法是不是需要调整一下了。
以上的思考过程，让我明白第一次 GitChat 的分享内容，以前端开发为载体和大家聊聊面向未来的社区化敏捷学习方法。必须要说明的是，这块知识量很多，本次分享只能从面上给大家介绍无法太细致。

敏捷学习法

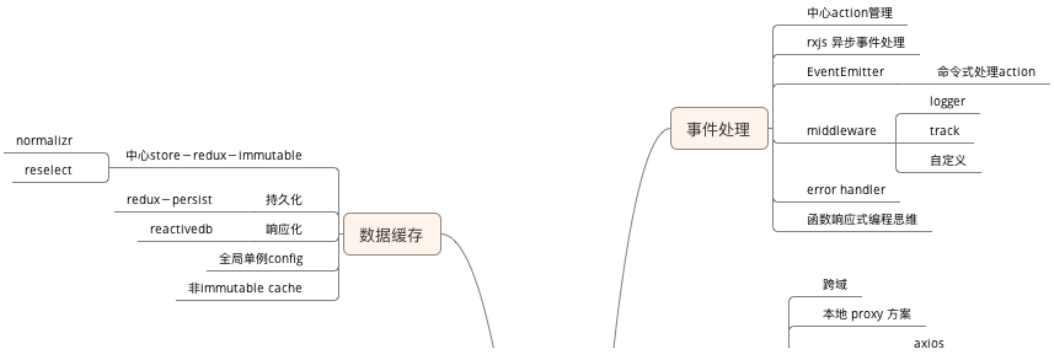
敏捷学习方法并没有专业的文献资料做背书，我只是结合自身理解和大家分享，受限于自身水平，并不一定是对的，只望给大家有参考意义。本节从基本的理论，知识树，以及两个案例带大家。

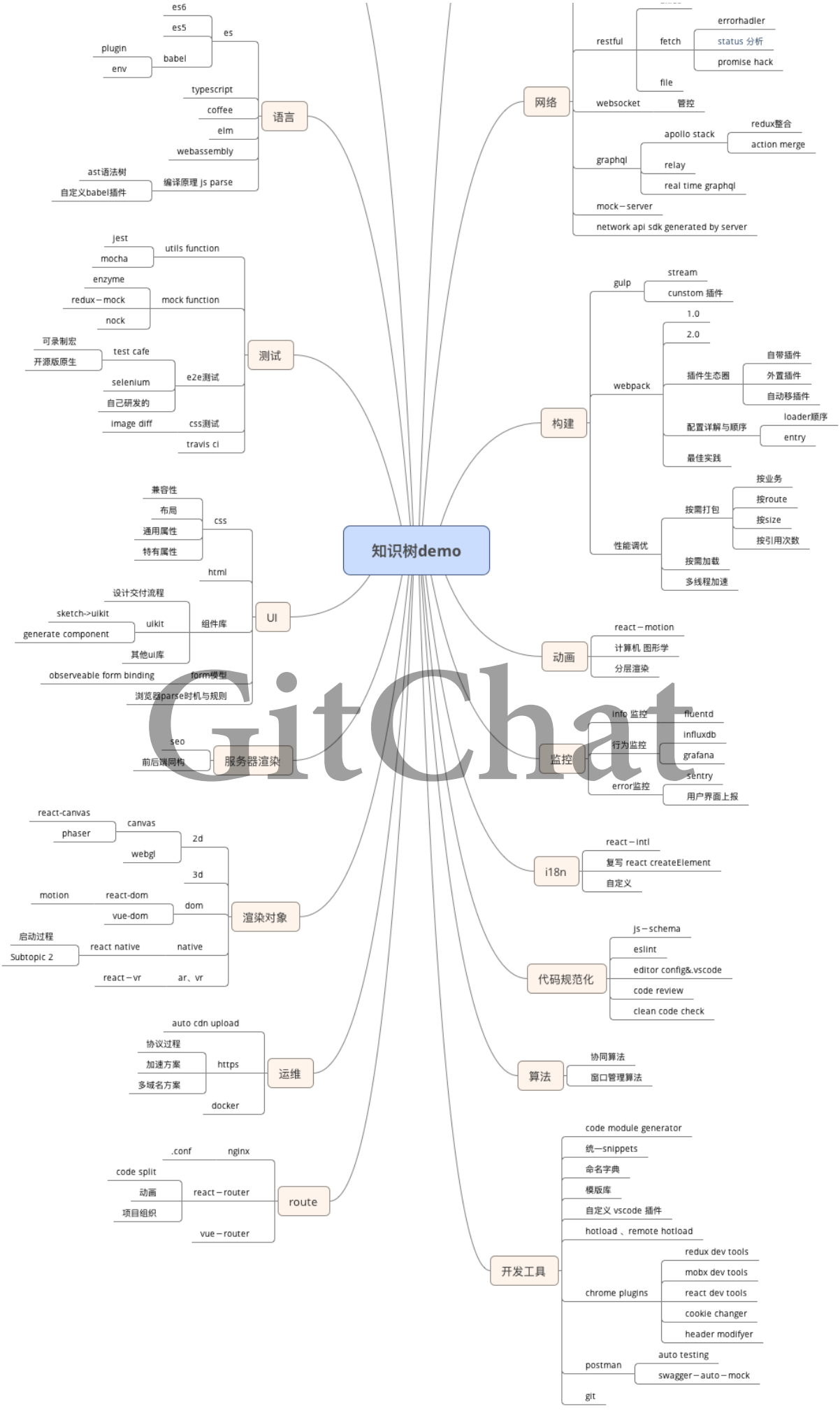
理论

敏捷学习的核心是敏捷的通过外部刺激更新自己的知识树。所谓的外部刺激主要包括被动解决各类问题以及主动研究各类课题。解决新问题时，利用自己已积累的所有相关知识判断可能的最优解决方法并拿它排序。其后对解决方案做拆解得出一些知识关键词，逐条快速通过知识获取途径得到相关资料，迅速实验田验证，过程中需要实时更新自己的知识树，后期不断完善整理知识树，成为新的积累，便于之后新的问题突破。主动研究课题时，则通过阅读各类资料迅速整理知识点，更新知识树，重建新知识与老知识的关系链，弄清楚每一个知识之间的联系。

知识树基本概念

我们把自己的知识想象成是一颗带有关系箭头的树（思维导图）。如图：




知识树demo
网络
restful
fetch
errorhadler
status 分析
promise hack
file
websocket
管控
graphql
apollo stack
redux整合
action merge
relay
real time graphql
mock－server
network api sdk generated by server
构建
gulp
stream
cunstom 插件
webpack
1.0
2.0
插件生态圈
自带插件
外置插件
自动移插件
配置详解与顺序
loader顺序
entry
最佳实践
性能调优
按需打包
按业务
按route
按size
按引用次数
按需加载
多线程加速
动画
react－motion
计算机 图形学
分层渲染
监控
info 监控
fluentd
influxdb
行为监控
grafana
sentry
error监控
用户界面上报
i18n
react－intl
复写 react createElement
自定义
代码规范化
js－schema
eslint
editor config&.vscode
code review
clean code check
算法
协同算法
窗口管理算法
开发工具
code module generator
统一snippets
命名字典
模版库
自定义 vscode 插件
hotload 、remote hotload
chrome plugins
redux dev tools
mobx dev tools
react dev tools
cookie changer
header modifyer
postman
auto testing
swagger－auto－mock
git
route
.conf
nginx
code split
react－router
动画
项目组织
vue－router
运维
auto cdn upload
https
协议过程
加速方案
多域名方案
docker
渲染对象
2d
canvas
phaser
webgl
3d
dom
react-dom
motion
vue-dom
native
react native
启动过程
Subtopic 2
ar、vr
react－vr
服务器渲染
seo
前后端同构
UI
css
兼容性
布局
通用属性
特有属性
html
组件库
设计交付流程
uikit
sketch->uikit
generate component
其他ui库
form模型
observeable form binding
浏览器parse时机与规则
测试
utils function
jest
mocha
mock function
enzyme
redux－mock
nock
e2e测试
test cafe
可录制宏
开源版原生
selenium
自己研发的
css测试
image diff
travis ci
语言
es
es6
es5
babel
plugin
env
typescript
coffee
elm
webassembly
编译原理 js parse
ast语法树
自定义babel插件

此图只是demo，每个知识结点都有自己的关联和上下文。我们可以给每个子模块添加颜色，深浅代表我们的对他的记忆程度，越实则使用越频繁，记忆也越深，越虚则越淡。甚至可能只是一个软链接在这里占位，遇到这个问题知道怎么搜索怎么查，但是不一定深刻理解。我们每天做各类项目和平时阅读交流的过程中，只要遇到新的知识，就应该先他挂到我们的知识树上，成为一颗虚树，而后继续整理，填充知识关系与知识深度使树更完善。这里所描述的知识树是一个抽象的概念，具体表现形式大家可以有自己的发挥。我一般使用思维导图，复杂的知识需要多个sheet来表达，并做好关联工作。

下面结合前端开发工作中实际的解决问题与主动学习两个案例让大家热热身。

敏捷解决问题案例

本节介绍自己曾经解决过的一个问题，告诉大家我的思考方式和学习方式。

问题分析

issue：解决一个rn项目开机经常直接闪退的问题。

直接静态观察代码，并看不出什么问题来，简单的debug也反映不出问题。假定我是一个RN初学者,但是kpi要求我迅速完成工作。

我通过其他编程经验告诉我，对闪退和性能的问题，需要对开机过程每一步启动过程做一个分析，打合适断点观测，看看哪一步闪退了，确定问题。而后则方案无非就是1.catch掉相应的error，不能让他影响整个app，2.解决相应业务的问题。

知识拆解

我把问题第一步监测大点拆解为以下三个问题。

- RN开启到首页渲染的关键帧（关键重要的可测结点）。
- 监测与表达。
- 监测内容。

第一个问题：先从自己的知识树去找，假设我对RN的启动顺序，只停留在“先加载需要用的资源，然后调用js逻辑”的层面。为了解决问题，必须获取这方面知识，在知识树上建立虚树。技术术语主要就是： react native start order，react-native-debug之类。从google搜索得到了一些网页信息，从stackoverflow也得到了一些相关问题。经过整理然后我得出了解决方案，断点应该打在native启动入口，js-bridge，js入口点，js初始化操作（redux store初始化）、js view mount的起点等…同时我把刚才的几篇文章link和内容做了一个速记，补充在这颗虚树中，提醒我之后可以完善它。关于获取信息的方法与筛选在之后的章节会详细讲解。

第二个问题：基本上思路就是把断点处的各种监测数据dump到同一的dashboard中。我联想到在其他编程体验中，我也许会用自带的断点工具，或者console.log把相关信息打在控制台。当然也可能会丢给data collector server统一观测，这取决于不同的业务场景。

在这个案例中，js和两个native以及中间的一些库的service并不在一个scope，很难能跨服务的跟踪，追查哪一步出了什么问题导致启动慢而容易crash。于是就用了做微服务监测的知识点，用fluentd做一个log center，它能把logger汇聚到一起展现出来。

第三个问题：回到一开始的需求.我需要解决启动慢且容易闪退的问题。所以需要检测两个主要信息点，一个是time duration，一个是health check。所以每一层断点需要做的就是给我反馈出耗时（当前时间与第一个断点时间差）、当前健康情况。

实验

迅速做实我们必须迅速做实验判断此方案的可行性，本案例中实验环境的搭建很简单。fluentd用docker版本地先布，代码库则建一个multiple worktree,在fake节点上做实验。做先测几个最重要的断点，最快速度判断此方案是否可行，并初步缩小范围。经过检验，方案可行，在native层的health情况正常，时间在允许范围内。接下来只需要测试js部分即可。

实施

经过实验，我们确定这个监测方案是可行的，接下来按照流程一步一步实施。实施过程和之前一样。实施的结论：最终我们发现，redux-persist-store 这个库耗时过长并有一定几率崩溃。

确诊问题

我们看到的监测结果类似医院的化验结果，确诊需要拿结果结合经验以及别人的实践找出原因，目的是为了寻找测试用例。根据这库功能猜可能是缓存数据没存对，导致一进来的时候就奔溃了。此时关键词描述变成了 redux-persist-store crash问题，看了一圈别人的issue之后，更加坚定这个想法。这时候需要寻找必现问题的测试用例。adb进安卓系统，发现persist store在本地存的是文件。这时候我通过对他的，增、删、改等手段，观察我们的log与app情况。发现当数据结构不一致时它必然崩溃，所以猜想被印证是对的。此时立马更新我的知识树，并加深了我对redux - persist的印象。

对症下药

这个步骤需要对症下药，此时症状能表达的关键词更加明确，1.对这个错误类型做catch，2.persist store需要调整数据缓存中间过程，不应该留有错误数据。这个问题先看看社区是怎么做的，于是继续进入资料查找模式，找到此库有custom transform的api可以让我处理incomming和outcomming的流，于是我可以在那个scope去做data check。至于check这个知识点，根据以前编程知识我们很容易想到结构验证，和数据筛选。这里继续用敏捷学习法，学到结构验证可用js-schema做，数据筛选则直接写规则filter。schema的规则根据实际数据情况定，filter规则根据做code split概念的影响，应该是有按需缓存，按使用比例缓存，按size缓存的策略。对error handler这个知识点，我们很容易想到try ，catch， filter error type这些词语，但是这里是native报错而出问题的库在js里，RN

肯定做了一些事。对着这个查询 RN 的error机制，获取到社区推荐的ErrorUtil全局对象catch fatal error的方式，于是下一步依然事实验与实践，发现起效。最后则把刚才的代码配上前面的测试用例，写成测试安心交付。同时把这个问题研究结果子树，render到自己的知识树上，并把知识相应的关联体系补上，比如以前这块知识只是redux-persist是解决store数据持久化的，为了下一次打开还原现场。这次新增对持久化的数据做transform以解决什么数据需要持久化，多少数据需要持久化，存储校验，异常处理等等。

总结

我们在解决问题的思维过程应该和CI流程一样，一步一步非常清晰。不断的迁移知识分析问题，不断拆解问题到更小的原子问题，不断对每个问题深化关键词，不断获取此关键词的信息并从信息获取更准确的关键词，不断抽象解决方案模型做实验验证可行性弄清楚实现机制，每一步则注意Single responsibility principle的应用，最后过程中的知识增长需要把diff记录进自己的知识树。

敏捷主动学习案例

本节结合敏捷学习介绍主动学习redux的案例，由于这里篇幅会太长，只能浓缩介绍关键部分。如果让你快速学会redux并着手重构曾经的项目，我们会怎么做？把redux书和相关背景都读一遍恐怕是来不及的，由于知识是有背景和上下文的，再讲这个案例时我只能假定学习者已经深入理解react，初步理解flux，此时学习redux。

带着目标阅读

阅读是必不可少的，至少我要知道此技术提出什么理念解决什么问题。就如上一段所说，我们不可能从头读而是带着知识经验结合目标阅读。学习新知识首先读的一定是官方文档的getting start，以及文档中的demo，了解清楚这个知识属于哪一个领域，解决什么问题，背后的理念大概是什么。带着这几个问题读一遍官方的介绍，切记带着目标读，让资料跟着你走，而不是一上来就看怎么install，或者深入细节。显然redux介绍一上来就说了提供一个可预测的全局state管理数据。解决的问题依然是data-view-action,只不过每个过程总线化管理。接着往下看一点案例结合我们的经验不难理解redux是侵入了react本身的props，用外层的store接管了this.state，setState方法变成了需要触发被dispatch bind过的action，而后reducer接收而后改变了state而后被connect的订阅者听到影响props改变下一版数据的试图。如果你对react本身很熟悉，这些在初步阅读redux官方文档以及demo的过程就可以体会到。通过知识对比则更清晰，就是把react自己的setState - 》this.state - 》view，父亲state通过props单向往下传的模式，变成了 setState 的过程在外面通过action - reducer完成，state变了则connect订阅者会往受体注入props。这些信息在初步阅读中结合react的知识迁移就能大概体会到。最后把心得体会更新到自己的知识树上，哪怕他只是我的联想，未来会被更正。

实践学习

计算机是工科，实践和理论需要并行，互相加深理解。目前对redux初步有点概念了，但是具体怎么实践并没头绪，此时要做的就是玩开源项目并自己做实验。一般选择官方推

荐的学习项目，把项目拖下来后做一个分类。比如有的就只是todolist，为了说明数据流的机制，有的增加了异步事件处理，有的增加了全局数据缓存feature。记住不要纠结demo的功能有多low，关键是提取技术点，然后结合自己的知识树互相对比学习。项目的学习顺序一定要从越简单开始往难走，越简单越可以单元化的看问题，当你掌握了之后，以这个为基础再去看比他技术点多一点的项目。学习项目中最重要的是，学习写法并分析他的用意，遇到不懂的，立马查补。比如对es6的一些知识不够熟练，看到某个写法有点懵。此时应该立马记下来，看看能不能大概查阅他的意义跳过先看，如果确实是一个重要知识点无法跳过，则必须弄清楚后继续。这里强调实验的重要性，学习别人项目绝不能只看代码，而是自我去做功能变异与代码变异。功能变异是指增减改同类功能。比如他做了一个加法器，那我用同样的方法在下面做一个乘法器，走一遍新feature的流程，不会就debug、diff…。代码变异则更关注实现细节，比如你可以换个写法实现某个方法为达到同样的测试结果。

快速深化学习

实践学习基本上是在别人的项目上做实验，是需要忍过去的。因为你还没有开始完成我们这节的目标亲自去做工作。当你确实看了好几个项目，并做过好些实验了，我们可以开始做快速深化学习。这个过程是一个循环，参考理论和别人的项目来写自己的程序。遇到不懂立马中断，回去做实验查资料更新知识树，深化完继续前进。比较痛苦的是，往往一开始你的项目进度非常缓慢，查资料和做实验的实践耗了很久，但是我的经验告诉我，知识的持续翻叠和联想对比很快能让你走出困境。

夯实基础，缓慢递增

这时候你工作的项目已经能应付，确实采用了一种社区推荐的方案重构了你的项目，配上了redux，也踩过一些坑，大概体会到它带来的好处与麻烦的地方。此时，此时最容易放松但是千万不能放松，因为你的理论和判断能力还是很弱，换个业务逻辑也许又无从下手。但是，经过这过程，你对整个知识骨干已经有理解了，对知识关键词也很清晰。这时需要对知识做拓展，和其他知识做关联，补充之前快速学习忽略掉的一些细节知识。这个过程是缓慢的，随着时间越长经验越足积累越深。

知识迁移

我们把学习知识比做一个叠纸游戏，有的人在不断学习的过程中，只是往上叠纸，那么学习10个知识点也就10层。有的人则是把知识混合，相当于对折了10次，最后哪个人的作品高度高呢？很明显，能把纸做对折的人是压倒性优势。很多知识他是有内在关联的，当我今天听说一个新知识，首先要对这个知识做归类，一般就是属于哪块的知识，解决什么问题，与现有的知识关系是什么，然后产生新的疑问和联想，然后再去循环研究。接下来从解决问题和主动学习两方面讲解一下。

解决问题

在解决问题时，很多思路并不是这个新知识告诉我的，而是我在其他地方的编程经验告诉我的，于是我很自然的联想到在当前的场景下，是否这个方法是否也会有实现，然后

把未知问题做转换。

以前在开发前端时，我会在启动开发项目时做这样的事：

```
``APP=v1 PACKAGE_SERVER=... yarn run dev
```

其实就是启动时给当前的语句配置环境变量，这样我能在里面能抓到process.env，然后通过 webpack DefinePlugin，传递到js里，存放在全局单例 config 中，供其他 module 引用。但是我们思考一下，如果这里的环境变量非常多，那么处理起来就会很难受，必须借助 dash 或者一些 snippets 工具记录。于是迁移一下在写 ruby 时用到的解决方案，叫 dotenv 他会解析 .env 文件里的环境，在加载时，他自动把 env 灌进去，对里面代码实现无侵入，好处则是我可以把这个. env 文件gitignore 掉，然后在本地开发时简单编辑文件切换不同的api等编译时环境变量。于是自然而然去搜索 dotenv in javascript的实现， 果然js社区造就有人干了事。于是立马就引过来用解决问题。 这样的例子还很多，比如我在做前端 route 切换动画的时候，我会想的是以前写 ios 前端时，他有一个切换算法很不错，于是对着 apple 文档对着用js实现一遍。比如我学习 react ，看到一堆生命周期，就觉得非常眼熟，只要你学习过安卓或者ios，生命周期就是最核心最基础的东西，这些知识点都是相通的，可以大把节省你的学习时间。比如在实践 java的 Spring 框架时，我也会和js对比着想，了解了动态语言和java这类的区别，比如对权限的表达，对类型的深刻理解等等。

渐渐大家就会发现，各语言的设计思想就是在不断借鉴与融合，对语言和框架的设计我个人始终它觉得是一个哲学问题。大家细心观察，就会发现很多开源项目的title后都有一个``(Inspired by 。。。)`,这表明它也是受到了其他技术的启发，在自己的项目中利用这些思想，并向原项目致敬。

主动学习

比如我今天学习垃圾回收(gc)这个知识。对这个知识结点的新增是在学c的时候就完成，后来学java垃圾回收机制时增加了知识子节点，并和c做对比。最明显的就是c里对象所占的内存在程序结束运行之前一直被占用，在明确释放之前不能分配给其它对象，java则当没有对象引用指向原先分配给某个对象的内存时，该内存便成为垃圾，这个垃圾能被自动清理。那下一个问题就是jvm怎么做自动清理的，自然也就有变量生命周期、清理调度机制的问题。下一个问题就是了解清楚后，我要怎么去节省gc开销，比如就会有对象不用时最好显式置为Null啊等实践。这些学习的问题同样会带到js上，我也会担心js啊。后来知道js也是属于自动回收，就自然也有变量生命周期一说。于是就会去思考什么情况变量算不用了，得出函数的执行结束，那怎么算函数执行结束，闭包呢？一路往下想。JS垃圾回收的调用者是浏览器，目前机制主要是按照固定的时间间隔周期性的执行，但是ie早版本就不是这样，用的引用计数法，那么当存在循环依赖就会出现内存泄漏，需要你手动置null。在知识不断联想和迁移过程中，我就会去思考在c、java、ruby、python、go分别是怎么做的，有什么共同点，我要注意什么。这次分享不对这块深入展开了。

说到这里大家应该深有感触，当我把知识交织在一起，就完成了这个叠纸游戏中的对折，而不只是往上加一层，能迅速了解一个方面的知识，而不是一个点。

信息获取与筛选

与以往的教科书或视频的学习方式不同，前端开发从开源项目源头迅速传播过来，版本也以天的单位在更新，社区非常年轻，不断涌现新的issue与新解决方案。这些知识被整理归纳成册或者教学视频是严重滞后的，我们需要更主动的获取知识并自我归纳整理，本节重点介绍信息的获取与筛选。

信息获取

对开发而言，信息获取的载体主要包括：1. 社区。 2. 博客&文章。 3. 文献资料、书籍。 4. 社交。

github(社区典型)

非常感谢github给我们营造了一个很好的技术中心，自然在这里也是检索信息最直接的途径。我们可以利用github的项目搜索，代码搜索等功能，对自己感兴趣的项目进行检索，观察别人是如何实践如何解决问题的。天下代码一大抄，看你会不会抄会不会学。github上也有丰富的社区资源，会不会寻找到小伙伴一起学习，至关重要。这会在在社区化学习一节中深入讲解。

google

google的重要性似乎大家都明白，但是能用好google的人却很少，会不会找到合适的关键词描述你的问题就是一个难点。在之前讲过的案例中，关键词是随着我对问题的深入，不断抽象成更贴切的描述原子问题的检索词。甚至我从信息结果中看到有价值的信息灵感术语拼接作为关键词再度搜索，这将会是你综合能力的体现。平时在训练时，可以你可以利用迅速它帮助别人，解决各种朋友的问题，你立马就是大神，因为其实身边能用的好google，并能迅速总结概括的人真的很少。

stack overflow &medium &博客

问答社区以及精品资料推送站往往也是重要信息渠道，社区提供很多机会让开发者参与问答。尤其在写代码的时候， stack overflow和dash一样必备，可以让我迅速了解其他小伙伴问的同类问题，自己也可以提问。而medium、优秀博客则成为主动学习的重要读物。

书籍和文献

书一定是必不可少的，我依然会去泡图书馆、下载电子书。但是读书是有方法的，不一定有时间让你从头看到尾，需要有选择性的阅读，边读边整理。比如我在学函数式编程的时候，我确实已经看明白了社区的案例，也已玩过很多demo并自己实验，但是我对函数式思维基础知识和架构的理论知识还不太理解，此时我需要去看一些相关书籍，比如函数式编程思维一书，边看边整理笔记，优化我的知识树，让我弄清楚它。具体怎么读怎么整理会在后面章节介绍。

社交

社交大家想的最多的就是娱乐聊天，其实这是我几年来最看重的学习方式之一。一个技术之所以能发展起来，对这些技术的贡献者和社区的活跃分子是重重之重，他们经常在关注的，讨论的，往往是这个技术的核心，毕竟他们会比我们更拿捏得准方向。因此我们需要去follow一些人，参与一些他们技术讨论聊天室，比如slack、gitter、discord之类，在聊天室里多问多听即可迅速成长。平时follow的重点则是湾区，硅谷的一些朋友，一些知名大公司开源组织，一些书的作者，一些技术的分享达人，一些技术的重要贡献者，往往这些人也会被整理在awesome+x 系列中的who to follow里。关注的另一好处在于当我遇到难以解决的问题，我需要向社区求指点。这比我问大学老师或者高价找培训班答疑等靠谱多了，社区有很多小伙伴都会很愿意和你分享他们的知识。

信息的筛选

在上述那么多信息获取方式中，我们已经得到巨量的信息。这可不是好事，信息越多，筛选信息就成为了最重要也是最难的事。我们的思路是寻找搜索项然后做类sql的思维过程。然而这些搜索项的建立是比较难的，他会随着我对一部分信息的理解程度变化而变化。这里我简单讲自己的做法，我会带着不同的目标看待每次的筛选过程，比如我需要有代码example，我需要时间是比较新的，发布在比较权威的资料站，大家有讨论…这些表层硬指标可以简单做一层filter，第二次filter则要从我对知识的理解而定，甚至从某些信息上发现这此搜索方向就有问题，那就立马切换。当然如果我确实找到了我要的资料集，接下来要做的就是map，把每一个子资料中我要的部分highlight，加入我自己的想法，同时去掉无关因素，最后则是reduce，把它汇聚成我要的知识集。

新形势下的阅读方式

这节让我们深入思考面向未来的阅读方式

边读边整理

我们需要有逻辑有目标的去读书，简而言之就是资料跟着你走，而不是你跟着资料走。举个例子 我在读函数式编程思维一书的时候，我很清楚我的目标就是了解函数式编程的核心思想，核心概念，以及各种写法，及解决的问题。于是我在读书的时候也会跟着自己的主线选择要看的内容，当然有时内容也会让我增加预定的主线节点。看技术书，其实和信息的筛选挺像的，不仅仅要读完，更要写完。利用margin note这样的工具，我边读边做摘抄，写自己的评论，做思维导图。例如：

当然，如果是技术书就会有很多实践类内容，更需要做实验，比如书上只是用java语言为例讲解了curry的概念，那么我一定会在边上把js的实现，python的实现补上。

同时多本书一起读

同时读几本书在以前看来是很难，然而现在对同类书籍用的非常多，比如学习函数式编程的时候，我就同时开了函数式编程思维，js函数式编程，java函数式编程。然后我对这部分采用敏捷学习法，其实就是对这些我要学的知识树不断做merge，并关注diff和互相的关系。比如，在java函数式编程里提到了范畴学和组合，但是在那本书只是提及并没有深化，于是我去找函数式编程思维和js函数式里面对这块做了解释，那么我就会整理在一起。这样的案例还很多，比如同样搞语法躺，我在学习es6语法时看到了arrow function，我立马去找coffee script，因为在coffee里有胖箭头和瘦箭头的区别，主要就是bind不bindthis的区别，而es只有胖箭头，当我在一个闭包里要创建非bind this的方法还得去用老方法，岂不是很蠢。

社区化学习

之前我有提到过社区的重要性，本节更深入的讲解一下。

你会用github么

由于我在Coding工作，Coding本身就是国内的github，可能对这块尤其重视。社区并不仅仅是被动解决问题，更是主动学习的好地方。

github项目生命周期

简单的说，一个新的feature，会经历issue的提出与大家的discuss过程，然后会有被分配者或者其它开发者向约定分支提pr，并指向这个issue，然后大家都可以对pr去review，对其中的代码做疑问。当然有权限的 core 开发者则可以确定是否合并。合并完后看放在哪个release发布。作为管理者，则会通过label排好issue的优先级，并建立好milestone，然后利用kanban管理进度。一旦有人提了pr需要去审阅他的功能，代码风格，代码文档，代码测试，并关注一下ci反馈的结果，此时也会在下面和大家做更深入的讨论。最后当一个milestone包含的issue都被kill掉了，就会新建release分支，然后在上面做一些bug的fix，改版本号，写changelog，最后发布到master。

学习

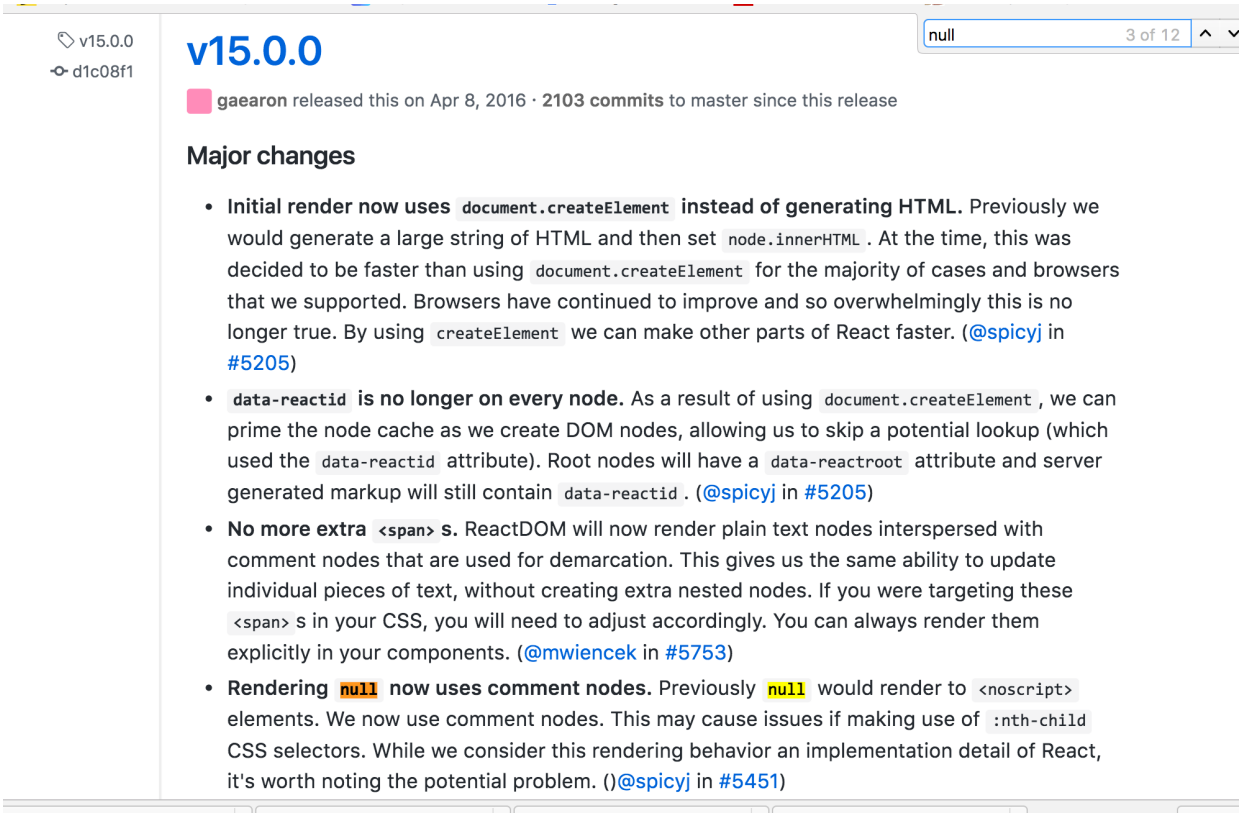
这些周期中，我们主要参与的是：

- 对issue的深入讨论
- 对项目的直接代码贡献
- 代码的review

review不仅仅只是纠错，更是一种学习。我们可以了解别人的写法用意和实现方式，我经常在社区review别人的代码，很多时候只是提问 + 学习，效果绝对比自己读源码好得多，毕竟这里有大家一起讨论。

issue则是让我知道大家会遇到哪些问题，现在是怎么解决的，往往能学到很多知识。每个issue的建立初衷一般都是为了得到pr改善这个问题，当然很多问题不一定需要pr就被别人解决了或者被关联的issue一起解决。

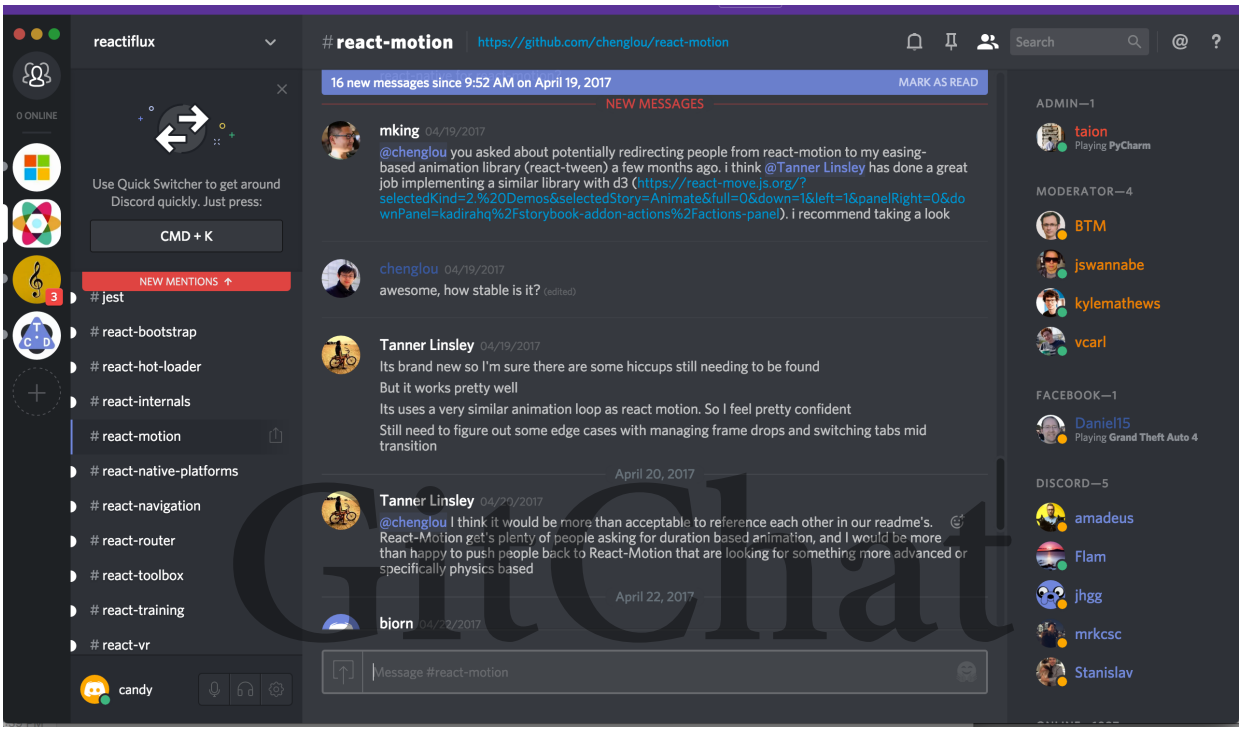
举个例子，比如我在学习react的createElement(null)在dom中的表现，当然我可以做实验，我也会去react的github搜一下release log，于是我发现在15.0.0的版本中提到。



于是我追溯进去看了一下pr和大家对此的讨论
Use comment nodes for empty components，通过它我又知道了这个issue的历史，曾经chenglou同学在解决React.EmptyComponent (NullComponent?) 这个issue是通过返回\

你会聊天么

社区除了github阵营外，团队有比较知名的slack，gitter，以及discord甚至reddit等讨论区。个人则是twitter和facebook以及自己的博客居多。我们以facebook社区官方的discord社区为例讲解。往往我们经常在用的技术都有自己的官方讨论区。



比如上图中看到react-motion的大神chenglou正在回答大家的使用问题，并且我们仔细看左侧的列表，会发现react经常遇到的问题都有相应的解决方案。于是还等什么，赶快敏捷学习更新你的知识树吧，聪明的人搜搜聊天记录就能学到不少！当然这里不得不提，聊天是有礼仪的。我看到gitter上一些国人开发者有素质低下的沟通表现表示很气愤，这里的技巧有机会再给大家详细分享。

你知道follow么

有的同学很迷茫我每天该看什么，结果就看知乎的心灵鸡汤，或者是脉脉里无聊的工资的讨论。真正需要看的并没看，只是捧着技术厕所读物在那边反复咀嚼，还觉得是金子。那么我们该看什么？信息推送站比较知名的是medium，hacknoon等，个人部分则follow那些大神的twitter就好。这里rss阅读器就非常重要的，我习惯用reeder、chrome的话用panda，用他们订阅一些感兴趣的feed，包括medium，egghead，掘金，以及一些follow的朋友的博客。

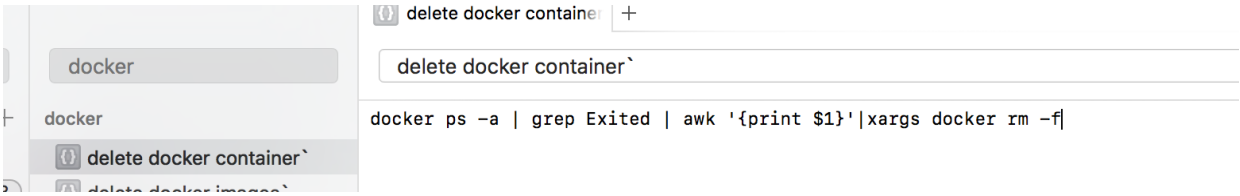
工具&实验田

工具

对工具的依赖其实很多同行有自己的看法，我是一个比较务实的人，我觉得工具就是为了提升效率的，多用也没什么问题。必备开发辅助工具比如dash，帮我快速的查询api以及使用snippet。snippetsLab，对经常用的代码片段做一个notes管理；alfred workflow之类，帮我做一些自动化的操作；kaleidoscope做一个文件diff工具，marginnote，xmind，freshcard，lucidchat等学习绘图工具。还有chrome上的一堆插件等等。

比如，我经常会用 kaleidoscope 做diff，之前我在逆向一个编辑软件的工程文件格式时，我就会去操作这个软件，然后对每一个版本的snapshot做diff，以此观测该软件的算法，我在分析ci报错时，我也会拿成功和不成功的两份 log 做diff，判断出现问题的地方。

比如，我经常会用dash的snippets做一些shell命令的快捷操作



上图中，我可以简单在shell 里打 ``delete docker container`,就可以给我反馈这一长串删无效container的方法。

这篇分享并不是工具篇就不再深入，具体还有哪些可以去看awesome-mac，我只是让大家知道工具在敏捷学习中的意义。

实验

当我脑海中有一个技术思维或者从资料中获得一些信息，我并不是很确定他是否正确，我需要快速的隔离问题并得到答案。于是我会利用工具建立自己试验田，实验环境是隔离于工程并mock完需要的上下文或者把工程mirror后作为上下文变成一个独立容器，或者利用online的实验田，例如codepen之类, 在上面搭配测试写实验。例如当我在做移除react-auth-wrapper地址栏的redirect参数时。我把问题归结于hack掉他的push配置，写一个util function，接受一个 url 参数: `www.coding.net?aa=1&redirect=xxx…` 然后丢掉redirect的信息并返回。这些就可以隔离在实验田做完，写完测试，然后把正确结果打patch丢回自己分支。

时间管理

敏捷学习的关键词是敏捷，要求在短时间内掌握知识，解决各类问题，同时不断加深知识点的深度和维度。这其中就需要对时间做管理。如果我看一个资料太陷入细节，一层一层往里追溯，最终的结果就是在那一个问题耗时太长，工作任务完不成了。于是除了聪明的学习外，我们还要管理自己、约束自己的学习。比较推荐的做法是实用omini focus，学习一下GTD时间管理法。对要做的知识学习事件做一个整理，写进 inbox ，并

对每一个事项做一些配置，包括轻重缓急，需要用到的资源、上下文，需要协作与否等。

总结

新形势下我们需要更敏捷的学习,利用一切资源快速更新自己的知识树，为了达到这点我们就必须要学会知识迁移能力，用对折的方式交织性学习而不是孤立的往上铺纸。新形势下，我们需要利用好新媒体、新搜索工具以及社区，在得到大量信息后要学会筛选和整理。为了更深入的理解原理，我们依然需要读书，只不过读书应该带着目的读，边读边整理，建议多本书对比着读，对比着整理。当我们已经整理了很多知识，要积极参与社区的活动，去和他人分享自己的实践，同时也从他人那边获取有价值的信息。得到各类信息后，需要在实践中学习和检验，我们时不时需要在实验田中做实验，检验某一个方案是否可行，把可行的有测试保障的代码很自信的搬回我们的仓库。这一系列过程，我们都需要敏捷，必须对每个学习过程做好时间管控，不能深陷其中耗时太久。

关于语言能力，由于当下技术大多是从开源英文社区直接传递过来，以上所有的步骤都离不开英语，所以对语言的基本阅读能力是敏捷学习的前提条件。

关于技术选型，我并不只学js，其他的编程语言，都在学习和实践，这样才能更好的做知识迁移，遇到新知识才会有联想有感觉。前端有太多的坑，需要一直跟进社区，一直保持重构，以及造各种适合自己的实践，而别人的实践很多情况只适合她的业务场景并不具备通用意义。关键还是这面向未来的敏捷学习能力让你任何时候立于不败之地。