

演进：在工作的前三年里快速成长（练习篇）

有人可以靠中彩票，然后一夜暴富；有人随随便便发几张自拍，就一不小心一夜成名。可技术成长，要一步一个脚印地练习，才能掌握某项特定技术。等到我们掌握了学习的技巧，才能用更短的时间，来掌握某项特定的技术。

而练习，也不是一天里写一万行代码，也不是重复写一百行代码，而是在一百天里，每天写下一百行代码。它需要一定的技巧，不懈的坚持，还有一些休息。因此在这篇文章里，我将分享工作几年里的练习技巧：

- 基础篇：正确的练习姿势。从程序员的基本技能：盲打，到练习使用快捷键、重构技能等，再到如何使用新的框架练习。
- 进阶篇：如何通过练习来提高。初学时，我们可以使用 Vue、React 去高仿一些项目；有经验以后，高仿应用只会让我们更累。我们便需要一些更高级的练习技巧，从引入别的框架思想，到造各式各样的轮子。
- 找到合适的时间练习。早上，慢慢进入状态；中午，适合做一些 Review；碎片时候，可以做一些知识的管理等等。
- 怎样才能持之以恒下去。分享一些制定目标的技巧，及激励自己的方式。

当然，练习有一个大前提是：**我们有充足的时间**。时间是一种很珍贵的资源，特别是对于长期加班的开发人员来说。因为**技术能力不足导致的加班**，会变成恶性循环。

如果你还没工作，那么便相当的幸运，你有相当多的时间。工作的时候，大家都忙于实现业务功能，没有时间让你提升自己。如果你已经工作了，那么你需要每天预留一些时间，才有机会去练习。每天会占用一些游戏、看电视时间，哪怕只是半个小时，一周、一个月、一年下来，帮助就很大了。

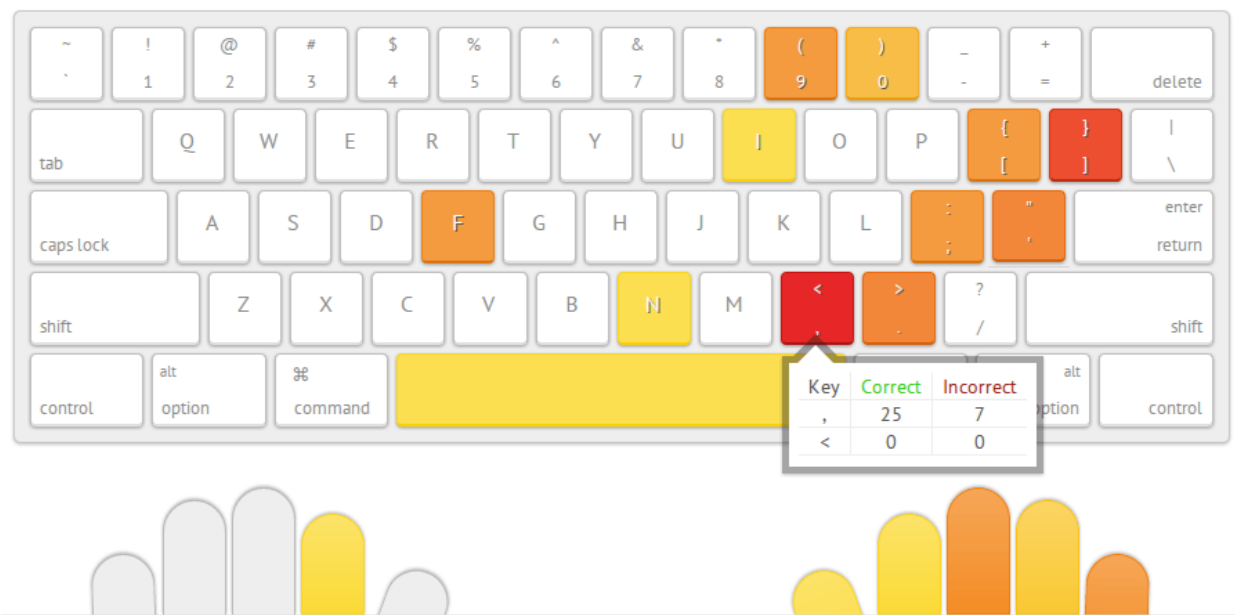
进行这些练习之前，请不要忘了根本——**能熟练地用框架、语言完成工作**。完成工作，相当于必须达到的 60 分及格要求。在胜任工作之外，提高能力到 80、90 分，追求更好的技术能力，才是正确的路线。

下面，让我们开始第一部分的内容吧~。

基础篇：正确的练习姿势

编程的时候，我们只是在码字——编码的过程（即思路）实际上是在脑子里完成的。娴熟的码字能力，可以帮助我们更好地编程。

小学时，自参加了五笔打字比赛之后，便开启了我的编程生涯。可当工作的时候，已经可以**熟练的完成工作**的我，仍然无法打对每一个字符。有一天，看到了一个名为 [Typing](#) 的在线代码打字练习工具。练习了一次之后，发现它会给出一些建议，便开始进行了一些编码练习。但是得到的反馈能表明，在打字这方面，仍然有一些提升的空间：



我的“自我解释”是：**今天的编程语言设计得不合理**——使用了各种字符，导致了右手在这方面的负担比较大。在那之后，我便陆续进行了一些基础的练习，并整理他们的因果关系，便有了下面的一些练习项目：

- 作为经常用电脑的人，应当掌握好打字的基本技巧，比如说采用正确的打字姿势，以及盲打技能。
- 作为一个程序员，应当“精通”使用手上各式的IDE、编辑器，熟练使用它们的快捷键。
- 作为一个专业的程序员，我们还要将重构代码、命名等高级的技巧掌握好。

这些练习，可以让我们成长为一个更专业的程序员。

语言与框架的练习

对于语言与框架的练习，算是比较简单的。于我而言，这种练习过程便是：

1. 买本相关的书籍，或者寻找份教程、官方指南。
2. 再找个合适的 Demo，熟悉基础概念，并掌握好相关基础。
3. 在 Demo 的基础上，实现一些业务功能，了解各种功能、特性。
4. 查看官方文档，查有没有漏掉了什么重要的东西。
5. 撰写博客、日志来记录这个过程。

因此，只需要找一个合适的网站、APP，作为模仿的对象，一步步往下练习即可。唯一的难点在于，第一次写 Web 应用的时候，可能会多花费更多的时间。新手期的程序员，对很多的概念都不清楚，如若能找到一个新手社区、群体，提高起来就会方便多了。

熟练使用语言或者框架，不能帮助我们成为一个『优秀』的成员。只能带领我们成为一个“胜任”的程序员，即我们可以凭借着这种练习，找到一份养家糊口的工作。

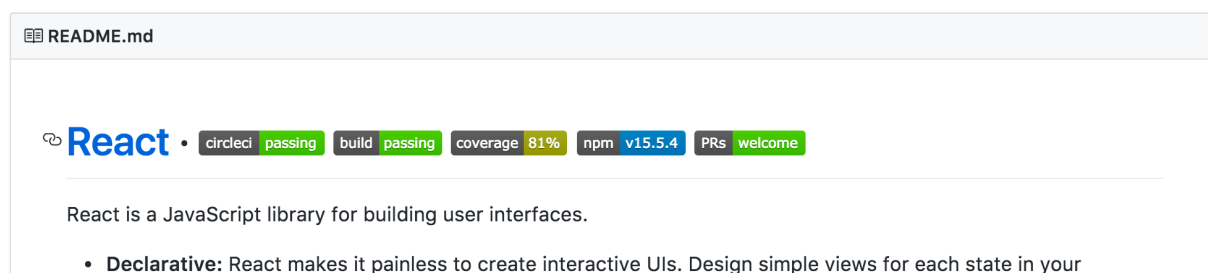
工程实践练习：模仿开源软件

工作的时候，写的都是业务代码，纯技术上的实践并不多。这意味着，**多年的工作经验，与技术能力的关系并无太大关联**。如果有一天，我们看到几年前写的代码，和今天写的代码并没有太大的区别，那么说明了：我们已然陷入了这样的一个瓶颈。

在学校写的代码，与工作写的代码，最大的区别在于：**软件工程实践**。单单凭借工作经验，那么在软件工程实践上的提高可能不会太大。受限于上线 deadline 的影响，多数项目的软件工程实践，并不能做到最好，甚至可能很差劲。如我们所见，国内的大部分公司（包括BAT）在这方面的实践也很难做全，更不用说做好。这些实践包括：

- 使用版本管理。诸如 GitHub 上的项目采用的 Git，基本已经普及。
- 使用持续集成。它可以为团队协作，提供一个可靠的帮助。
- 完整的测试用例。编写单元测试、功能测试等等。
- 代码检视。用于提高整个项目的质量。
- 等等。

而对于一个优秀的开源软件来说，为了保证好项目的质量。拥有者往往付出了很多的精力，在提高软件工程的实践上。因此，对于软件工程实践来说，最好的练习，便是模仿开源软件，并自己去创造一些轮子。以 React 为例，其在首页拥有下面的几个徽章（badge）：



分别是：

1. Circle CI，即持续集成，诸如测试是否都通过、部署是否成功等等。
2. Travis CI，同上。
3. Coverage，代码的测试覆盖率，81%。

4. npm，当前的版本号。

5. PRs welcome，即欢迎来 Pull Request。

那么，我们在实践的时候，就可以模拟这样的项目组成，一步步往下实践：

1. 为项目添加测试框架，如 Java 里的 JUnit，Node.js 里的 Mocha 等等。
2. 添加自动化测试脚本，如 Java 里的 Gradle，Node.js 里的 Grunt、Gulp、NPM 等等。
3. 添加测试覆盖率工具。
4. 添加持续集成，如 Travi CI 或者 Circle CI。
5. 添加代码质量分析工具，如 Code Climate。
6. 制定目标，并完成。

最难的实际上是最后一步，制定一个目标并实现。它可以是测试覆盖率要达到 90% 以上，这就需要一步步的来完成。如先将目标放到 60%，再慢慢地往上提升，直到 90%，甚至 100%。在这个过程中，会不断地遇到一些挑战，如**难以测试的代码，为了编写测试而修改功能代码**等等。但是，它能确实实地帮助我们提高工程能力。

基础练习：从码字到盲打

编码的时候，如果我们心里想输入的是一个 print，结果打下的字符是 oront，那么我们就需要删了重来。又或者是小心翼翼地，边看键盘边输入一个个字符。虽说，编码只是一个打字的过程，但是很多时候，经常出现的错字会中断我们的思路。因此，**盲打应该成为程序员的基本技能**。而这里的盲打，指的并不是我们可以闭上眼睛打字聊天，而是可以完成编程工作，即能盲打下 26 个字母，及各种字符，还有各种功能键。

而在进行这一类练习的时候，**我们经常会遇到的一个障碍：度量**。即以某种方式来衡量练习的成果，我们做了很多的练习来提升自己，但是没有数据来支撑。它不像编码，我们写了几行代码，完成了一个功能，那么写下的这些代码的价值就是可以衡量的。因而练习的时候，我们可以寻找一些合适的工具，如 Typing.io、Keybr.com 这一类工具。如 Typing 使用的是真实的代码片段，它能帮我们发现真实场景下：我们容易打错哪些字，容易按错哪些键，我们的打字速度是多少等等的内容。

对于**可以衡量的**打字练习，我们可以订下**每天十几分钟的时间**，一段时间要提升到什么水平的目标。这样它便能满足**SMART原则**，就能让我们看到我们在这段时间内的提升。

当时，我拿 Typing 练习的时候，差不多练习了一个月，每天大概半小时左右。因为打字的速度比较快，所以容易出错，所以便将注意力放在减少错误上。而对于有些人来说，则是相反的，即打字速度比较慢，但是准确率比较高。而这个练习的主要目的是，能熟练地做到**盲打**，不让它影响我们的效率。



掌握了熟练开关机、键盘上的各种按键后，我们就在使用工具上做一些效率的提升。

基础练习：掌握开发工具

刚工作的时候，发现每个有经验的程序员，几乎可以不用鼠标编程。熟练的使用各种快捷键，进行代码重构、打开新页面、开启新窗口等操作。慢慢的，我觉得自己在这方面有相当大的提升空间。

这意味着，我要学习、探索开发工具的功能，也要能使用快捷键来控制。尽管在日常结对编程、代码检视、交流的时候，可以请从别人身上学习。但是理想的方式，还是应该自己去练习。

对于大部分的开发工具，它们都有对应的手册、Keymap 或者 cheatsheet，即“作弊表”。如下是 IntelliJ Idea Keymap 的截图。

  DEFAULT KEYMAP			
Remember these Shortcuts			
Smart code completion	Ctrl + Shift + Space		
Search everywhere	Double Shift		
Show intention actions and quick-fixes	Alt + Enter		
Generate code	Alt + Ins		
Parameter info	Ctrl + P		
Extend selection	Ctrl + W		
Shrink selection	Ctrl + Shift + W		
Recent files popup	Ctrl + E		
Rename	Shift + F6		
General			
Open corresponding tool window	Alt + #[0-9]		
Save all	Ctrl + S		
Synchronize	Ctrl + Alt + Y		
Toggle maximizing editor	Ctrl + Shift + F12		
Inspect current file with current profile	Alt + Shift + I		
Quick switch current scheme	Ctrl + BackQuote (`)		
Open Settings dialog	Ctrl + Alt + S		
Open Project Structure dialog	Ctrl + Alt + Shift + S		
Find Action	Ctrl + Shift + A		
Debugging			
Step over / into	F8 / F7		
Smart step into / Step out	Shift + F7 / Shift + F8		
Run to cursor	Alt + F9		
Evaluate expression	Alt + F8		
Resume program	F9		
Toggle breakpoint	Ctrl + F8		
View breakpoints	Ctrl + Shift + F8		
Search / Replace			
Search everywhere	Double Shift		
Find	Ctrl + F		
Find next / previous	F3 / Shift + F3		
Replace	Ctrl + R		
Find in path	Ctrl + Shift + F		
Replace in path	Ctrl + Shift + R		
Select next occurrence	Alt + J		
Select all occurrences	Ctrl + Alt + Shift + J		
Unselect occurrence	Alt + Shift + J		
—Productivity Boosters			
Editing			
Basic code completion	Ctrl + Space		
Smart code completion	Ctrl + Shift + Space		
Complete statement	Ctrl + Shift + Enter		
Parameter info (within method call arguments)	Ctrl + P		
Quick documentation lookup	Ctrl + Q		
External Doc	Shift + F1		
Brief info	Ctrl + mouse		
Show descriptions of error at caret	Ctrl + F1		
Generate code...	Alt + Insert		
Override methods	Ctrl + O		
Implement methods	Ctrl + I		
Surround with...	Ctrl + Alt + T		
Comment / uncomment with line comment	Ctrl + /		
Comment / uncomment with block comment	Ctrl + Shift + /		
Extend selection	Ctrl + W		
Shrink selection	Ctrl + Shift + W		
Context info	Alt + Q		
Show intention actions and quick-fixes	Alt + Enter		
Reformat code	Ctrl + Alt + L		
Optimize imports	Ctrl + Alt + O		
Auto-indent line(s)	Ctrl + Alt + I		
Indent / unindent selected lines	Tab / Shift + Tab		
Cut current line to clipboard	Ctrl + X, Shift + Delete		
Copy current line to clipboard	Ctrl + C, Ctrl + Insert		
Paste from clipboard	Ctrl + V, Shift + Insert		
Paste from recent buffers...	Ctrl + Shift + V		
Duplicate current line	Ctrl + D		
Delete line at caret	Ctrl + Y		
Smart line join	Ctrl + Shift + J		
Smart line split	Ctrl + Enter		
Start new line	Shift + Enter		
Toggle case for word at caret or selected block	Ctrl + Shift + U		
Select till code block end / start	Ctrl + Shift + J / [
Delete to word end	Ctrl + Delete		
Delete to word start	Ctrl + Backspace		
Expand / collapse code block	Ctrl + NumPad+ / -		
Expand all	Ctrl + Shift + NumPad+		
Collapse all	Ctrl + Shift + NumPad-		
Close active editor tab	Ctrl + F4		
Refactoring			
Copy	F5		
Move	F6		
Safe Delete	Alt + Delete		
Rename	Shift + F6		
Refactor this	Ctrl + Alt + Shift + T		
Change Signature	Ctrl + F6		
Inline	Ctrl + Alt + N		
Extract Method	Ctrl + Alt + M		
Extract Variable	Ctrl + Alt + V		
Extract Field	Ctrl + Alt + F		
Extract Constant	Ctrl + Alt + C		
Extract Parameter	Ctrl + Alt + P		
Navigation			
Go to class	Ctrl + N		
Go to file	Ctrl + Shift + N		
Go to symbol	Ctrl + Alt + Shift + N		
Go to next / previous editor tab	Alt + Right/Left		
Go back to previous tool window	F12		
Go to editor (from tool window)	Esc		
Hide active or last active window	Shift + Esc		
Go to line	Ctrl + G		
Recent files popup	Ctrl + E		
Navigate back / forward	Ctrl + Alt + Left/Right		
Navigate to last edit location	Ctrl + Shift + Backspace		
Select current file or symbol in any view	Alt + F1		
Go to declaration	Ctrl + B, Ctrl + Click		
Go to implementation(s)	Ctrl + Alt + B		
Open quick definition lookup	Ctrl + Shift + I		
Go to type declaration	Ctrl + Shift + B		
Go to super method / super class	Ctrl + U		
Go to previous / next method	Alt + Up/Down		
Move to code block end / start	Ctrl + J/[
File structure popup	Ctrl + F12		
Type hierarchy	Ctrl + H		
Method hierarchy	Ctrl + Shift + H		
Call hierarchy	Ctrl + Alt + H		
Next / previous highlighted error	F2 / Shift + F2		
Edit source / View source	F4 / Ctrl + Enter		
Show navigation bar	Alt + Home		
Toggle bookmark	F11		
Toggle bookmark with mnemonic	Ctrl + F11		
Go to numbered bookmark	Ctrl + #[0-9]		
Show bookmarks	Shift + F11		
Compile and Run			
Make project	Ctrl + F9		
Compile selected file, package or module	Ctrl + Shift + F9		
Select configuration and run / debug	Alt + Shift + F10/F9		
Run / Debug	Shift + F10 / F9		
Run context configuration from editor	Ctrl + Shift + F10		
Usage Search			
Find usages / Find usages in file	Alt + F7 / Ctrl + F7		
Highlight usages in file	Ctrl + Shift + F7		
Show usages	Ctrl + Alt + F7		
VCS / Local History			
Commit project to VCS	Ctrl + K		
Update project from VCS	Ctrl + T		
Push commits	Ctrl + Shift + K		
'VCS' quick popup	Alt + BackQuote (`)		
Live Templates			
Surround with Live Template	Ctrl + Alt + J		
Insert Live Template	Ctrl + J		

上面列出了其可用的快捷键，及其相应的用途。因而只需要打印好，放在眼睛能看到的地方，就能有效的改善。除了打印成纸质资料，他们还可以有不同的形式，如鼠标垫、杯子的形式。需要的时候，便可以一眼看到；平常多看到几次，也能多多少少的多记住一些。

需要注意的是：对于开发工具而言，没有必要掌握所有的快捷键，而是只掌握常用的功能。我曾陷入了一个误区，练习使用快捷键的时候，边练习一些重构的技巧，同时也花费了时间在练习一些『屠龙之术』上——一些非常少用的功能，除了炫耀，也没有什么用。时间一久，我便忘了很多的快捷键。

再举些例子：如 Vim，对我而言，一般用于服务器维护及 Git 修改。因此，主要使用的功能便是：快速地改几个字符、更新配置，保存并退出。如 Chrome 浏览器，在日常使用时，配合下 Vim 插件，便不需要鼠标。在进行前端开发的时候，便需要使用鼠标来调试。

对于大部分的工具来说，我们只需要一个 CheatSheet。复杂的工具，如 Vim、Emacs，则需要有一本更专业的书。它们是高度可定制的，这也意味着我们需要一步步的定制这些工具，寻找合适的插件，自定义快捷键，又或者是使用别人的配置。

而，要衡量快捷键使用方面的提升，目前还没有看到有效的度量工具。如果有的话，那么就是编码的时候，使用鼠标的频率。因此，在某些特定的时候，可以通过禁用鼠标来提升自己在这方面的能力。

进阶篇：如何通过练习来提高

尽管我在上面指出，学习新框架的最好姿势是：基于现有的业务来学习。即从工作中学习，从做中学。但是，如果一直**只使用**新的框架来重写旧的业务，那么你的成长就会趋近于 0。第一次，使用新框架时收获可能颇丰；第二次，收获的东西就更少了；第三次，你可能就学不到东西。

因此，在业余的练习时间里，不要一直练习新的框架，不要再拿 Vue、React Native 去高仿一些应用。**当且仅当，你所处的项目正在使用新的框架**，这种练习才是有意义的。

经过上面的练习，我们提高了我们的工作效率。同时，在别人的眼里，我们更像是一个专业的程序员。在这之上，我们还需要提高顶层的能力。下面介绍的是，我尝试过的一些，比较有效果的提升方法：

- 阅读开源软件与重构代码。
- 造自己的轮子来重写应用。
- 结合设计模式。
- 引入其它领域的思想。

总的来说，收获还是蛮多的，特别是造轮子，能有更大的提升。与其他的练习稍有不同的是，因为涉及到代码设计，这里的练习有些难以衡量。这时候，我们应该是保持着**练习的心态**，并意识到我们是在做这方面的练习。

阅读开源软件与重构代码

如果在工作环境中，没有代码写得比较好的人，那么我们就只能从开源代码中去学习。笔者之前写过一篇《如何以“正确的姿势”阅读开源软件代码》的文章，文中我建议的阅读开源软件代码的方式是：

- clone 某个项目的代码到本地。
- 查看这个项目的 release 列表。
- 找到一个看得懂的 release 版本，如1.0或者更早的版本。
- 读懂上一个版本的代码。
- 向后阅读大版本的源码。
- 读最新的源码。

可只读这些代码，不能让我们显著的提高水平，我们应该结合『重构』这个技能来练习。从我的练习经验看来，**对于重构的练习是最有意思的**。我们可以见证，一段不好的代码在我们的调教之下，焕发出新的光彩。当我们重构一段坏味道的代码，对比重构前后的代码，便会发现自己竟然有这样神奇的能力。

如果找不到合适的练习项目，可以到 GitHub 上找一些 **star 多**，但是**没有测试、缺少 CI** 等的项目练习，这样的项目在 GitHub 上也是蛮多的。

有一次，我在寻找一个迷你的 Markdown 解析器，看到 GitHub 有一个精巧的实现。它有 100+ 的 star，但是没有测试、四百行的代码里，有一个方法有三百多行等等的坏味道。于是，便花了几天的时间，边思考边重构这个项目。这样对编码的提升比较大，因为工作的时候，完成任务是第一优先级，然后才是质量。

因此，对于我们练习来说，我们只需要：

- 找一个不错的开源库。
- 阅读其中的代码。
- 找到代码中设计不好的地方。
- 对**自己认为设计**得不好的代码重构。
- 结合《重构》一书，来改进设计。

需要注意的是：不同的人对于代码设计，有着不同的观点。因此，在这时如果只是因为代码的设计不好，而不是代码里有各种坏味道（code smell），那么，就不应该去给别人的代码提 Pull Request。

造自己的轮子来重写应用

与阅读代码、重构相比，造一个自己的轮子，来实现同样的功能，便是一个更不错的选择。在 Web 开发领域，大部分的开发框架本身都是『通用型』的框架。即它拥有相当多的功能，其中有很多的功能都不会用到。如你使用 jQuery 的时候，你可能只会使用到其中的 Ajax、Event 等功能，那么你就可以写一个新的框架，兼容这两个接口。

练习时间充裕的时候，便可以自己动手去做一个。上面说到的阅读框架代码，是一种好的方法。除此无论是前端还是后端，我们都可以找到从零创建框架的资料，来帮助我们理解框架的组成。

通过阅读诸如 Python 里的 Flask、Ruby 里的 Sinatra 等轻量级的框架，我们就能理解一个框架所需要的元素，并模仿他们做出一个新的系统。这些框架的关注点是：处理 HTTP 请求的 CGI、与数据库交互的 ORM、控制逻辑的 Controller 层、返回 HTML 的 View 层等等。除了相关的框架，我们还能在 GitHub 上看到很多人模仿这些框架。做一个这样的后台框架，搭建自己的博客，那就能理解好这一系列的逻辑了。

对于前端来说，也是类似的，诸如 Building React From Scratch，可以让我们在 250 行理解 React 的原理，并做出一个类似的框架。除了 MVC，还有模块化设计、数据请求等等的内容。在两三年前，《JavaScript 框架设计》就是这方面一个不错的选择。

我曾经造过一个名为 Lettuce 的前端框架，它的主要目的就是用于：学习前端框架的设计，便在自己的多个业余项目上使用这个框架。而在前端领域，定制自己的 UI 框架、CSS 框架也是一个很不错的选择。再用到自己的博客上，再写上『自豪地采用 xx 框架』，岂不是更加的自豪？

在底层领域，又有各式各样的《自制操作系统》、《自制编程语言》、《自己动手设计物联网》等等的书籍，它们都能让我们从底层理解一个系统的组成。除此，还有各种各样的剖析类书籍，可以让我们理解底层机制的同时，也能让我们制作出一个框架。

最后，我们只需要能不改写或少数改写代码，将我们的应用运行在上面，便是成功的一个仿造的轮子了。

结合设计模式

设计模式，不同的人有不同的看法。在我看来，一个优秀的程序是要能『看懂』的。即不一定要精通，但要能识别出来，它是一种设计模式。当我们看到了一次又一次的相似设计时，应该猜想到，其背后应该是一种设计模式。如在前端开发框架中的『双向绑定』，它实际上就是发布-订阅模式，又或者称观察者模式的一种实现。

在笔者看来，模式就是一种高级的语言。当别人一说『工厂模式』，多数人瞬间就明白了，不犹得会发出：原来如此，这一类的感叹。认识了一些模式后，一遇到一些特定的场景，我们就能一下子套用这种模式。

可只凭阅读 GoF 的《设计模式》一书，又或者《Head First 设计模式》、《重构与模式》等设计模式书籍，我们所学的知识便是有限的。我们要做的是：

- 先熟悉书本上的示例代码，来对不同的设计模式有一个大的了解。
- 识别日常代码中的设计模式。
- 练习这些设计模式，并掌握常见的设计模式。
- 尝试在日常的代码中，套用设计模式。

- 重构现有的代码到设计模式。

要对设计模式进行练习，不是一件容易的事。并且很多时候，容易模棱两可的情况，即适合使用 A 模式，又适合使用 B 模式。这是因为我们是在为设计而设计，因此会尽可能的贴近现有情况。

引入其它领域的思想

不同的领域里，都有自己领域的优秀思想。如我们熟知的设计模式，便是受建筑领域的《建筑的永恒之道》中描述的 253 个 建筑模式的启发。又如今天流行的精益思想，最早是来自汽车制造业，可它对软件行来说，有着令人受益匪浅的启发。好的框架、软件是会相互学习，如 iPhone 与 Android，都在不断地借鉴——通知中心，但是又在那之上做一些改进。

又比如，今天的前端框架里，很多思想都是从后端“借鉴”过来的。如 Angular 中采用的依赖注入，便是深受 Java 语言的影响。近一点来说，Redux，框架最初是用在 React 上，但是它已经被推广到了 React 和 Vue.js 上。

因此，当我们发现一个新的优秀思想产生时，便可以尝试引入到自己的领域里。又或者我们所处的领域，正遇到一些难题，答案可能就在别的领域里。可在这方面的练习，往往都是一些创新性的练习。多数时候，我们的探索可能没有结果，但是它往往能对自己有更大的启发。

找到合适的时间练习

每天能有半小时、一小时甚至更长时间的稳定练习，比三天打鱼两天晒网的效果要好得多。清理出一些固定的时间，用于为自己腾出时间来提升自己。既然，你都有时间到这篇文章，那么你应该属于能腾出时间的人。

如果不能的话，那么我们也可以尝试去挤出一些时间，如从上下班去寻找空间。即使是同一公司，不同的人都有不同的上下班时间，所花费在路上的时间也有所不同。有的人，需要在几环外坐个一个多小时的地铁，再转公交才能到公司；有的人，只需要出门左转，走个十分钟就到公司了。因为在路上花费的时间不同，也在一定程度上影响了学习、练习等等的的时间。

因此，如果可能的话，应该减少花费在上班路上的时间，才能避免继续陷入这样一个恶性循环：**租不起近的房子，花费大量的时间在路上，没有时间提升技能。**

早上

早上的练习，是一种慢慢进入一天工作状态的感觉。一旦上班时间到来的时候，就已经进入工作姿态了——对于“资本家”来说，可谓好事一件。早晨刚醒来，总会想起昨天项目做到哪一步，便更容易反思哪里做得有问题。

如笔者已经习惯了，每天七点起床、洗漱，随后写会代码，再去上班。有时候，可以有一个半小时的练习时间，有时候会有半个小时，将这些时间浪费在梦里总是有些可惜。同时，之前为了能成功地上公交，便提前半个小时到公司，写一些开源软件的代码。毕竟，作为一家非产品公司，你无法和别人解释说，我们做了些什么、取得了哪些成就。

在很多地方，这是一个很好的策略：**错开高峰期上下班，路上就不容易堵车**。所花在路上的时间就缩短了，那么我们就有时间来练习了。

需要注意的是：**练习的时候不要关注时间，而是关注怎么于提高**。关键点在于：让每天进步一点。

中午

吃完饭后，因为米饭血糖指数高的缘故，容易犯困。对于北方的同学来说，因为主食不是米饭，所以这就不算是一个问题了。这个时候，身体会妨碍我们进行一些练习。可如果你的午休时间比较长，那么也可以做一些练习，再去休息片刻。

碎片时间

对着屏幕写代码，时间一久，集中力就会开始涣散，便应该休息会儿。刷刷资讯、朋友圈，又或者收集各种资料，开放我们的视野。接收各种新的知识，来扩大自己的视野，以便于自己了解整个市场的水平。

常见的方式有：

- 阅读个人博客、微信公众号。
- 维护自己的 Awesome 列表——寻找自己觉得好的开源项目。
- IT 新闻、技术文章聚合网站——我很不喜欢聚合网站，大部分的聚合站点的行为无异于文章抄袭。
- GitHub Trending。

将这些内容存储到 Evernote、WunderList、OneNote 等各式各样的云笔记里，然后**定期清理、定期清理、定期清理**。收集只是一种方式——没有啥用的方式，**因此建议先读完一遍**，再去收藏这样的文章。多数时候，我们会发现自己收藏了很多的内容、买了很多的书，但是却没有时间去读。

晚上

经历了漫长的加班，回到住的地方，可能就会想休息了。如果白天没时间练习，晚上也不能抽出时间练习，长期以来，一年的工作经验就要变成五年来用了。

晚上练习的同时，我们应该注意：在睡觉前 30~60 分钟停止编码，否则上床的时候，脑子里可能还是这些代码，就容易失眠。万一灵感一来，那就还要爬起来继续写。这个时候，可以**阅读**一些相关或者无关的书籍、资料。在阅读的过程中，尽管我们已经不在思考内容。但是潜意识里还在思考中，**这时就能很容易就会遇到一些灵感**。

最后，休息的时候，尽情睡觉吧~。

怎样才能持之以恒下去

在上文里，我们只谈论一些方法和技巧，可是它们并没有什么用。每个人都知道所谓的『一万小时理论』，但是真正要坚持下来，并没有我们想象中的那么容易。

我们需要从娱乐时间里抽到一部分，原本舒适的玩游戏、睡觉、刷微博时间，现在要成为另外一种『痛苦』？可是，既然这些“无聊的事情”我们都能上瘾，那么我们是不是没有找到合适的路？

设定目标与 SMART 原则

按上文中的分法，练习可以分为：**日常固定时间的练习**，及针对**某一特定主题的练习**等多种类型。当我们开始练习某一个具体的技术、框架、模式时，最好能制定一个简单的练习计划，如每几天练习某一内容、多少天内用某一个框架实现什么功能。

先设计一个小的目标，并且能在短期实现。当发现自己可以轻松地坚持下来时，再慢慢的扩大目标，直至我们能做得更好。可是，设定一个练习目标也不是一件简单的事，它也有很多考量的地方。

毕业的时候，在公司接受了针对于毕业生的培训，期间学习到了一个用于制定任务、目标的 SMART 原则：

- **具体的 (Specific)**。即我们要有一个**明确**的目标，如在一周内用 Django 写一个博客系统，而不是用 Django 写个东西。
- **可度量的 (Measurable)**。即衡量是否达成目标，我们只需要能创建、查看、删除博客，那么我们就算完成了这样的任务。它可以用来不断地突破自己。
- **可实现性 (Attainable)**。即这个目标一定是可以实现的，不能实现的目标没有啥意义。与些同时，练习初期定下的目标不能困难。
- **相关性 (Relevant)**。即目标与其他目标的关联情况，如我们练习 Django 是为了提高 Django 或者后台的技能。如果我们的大目标是提高前端技能，那么这个目标对于当前的意义并不是太大。
- **时限 (Time-based)**。即时间限制，如上面提到的一周内用 Django 写一个博客系统的期限。

经常能在微信朋友圈看到，朋友的 100 天英语阅读计划，这样的目标就是合理的——可实现的、具体的、有时间限制、可度量的。

如果我们想每天固定时间进行练习，那么我们应该做一个短暂的尝试，如七天，再慢慢的不断扩大时间目标，二十一天、两个月，随后再扩大到一个更大的目标。

坚持与激励自己

我们可以使用 GitHub 上的 Contributions 来激励自己，每一天的痕迹都很明显，甚至是可以拉拢一些小伙伴，与我们一起参加类似的活动。GitHub 本身具有社交属性，可以让我们看到别人做了什么，做了多久。

由于 GitHub 的服务器在国外，访问的时候可能会受限于网络。国内的开源中国的码云和 Coding 也有类似的活跃度，建议访问 GitHub 有问题的读者，可以使用这些服务。

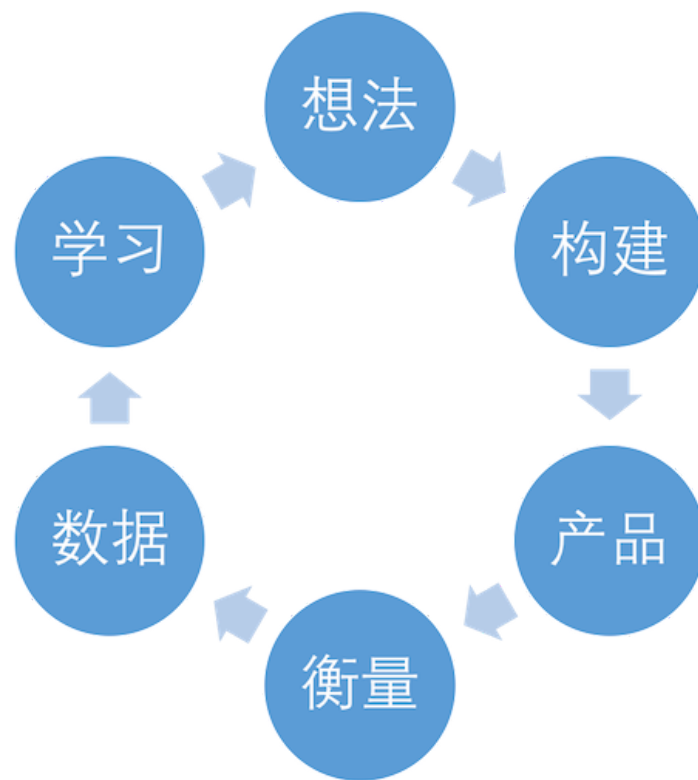
上文中提到的朋友圈 100 天英语阅读计划，也是相似的，它可以让别人监督你是否完成——前提是，有人一起和你做相同的事，因此**可以找个人和你一起练习，相互监督**。

刚开始练习的时候，练习的内容基本上很充实。时间一长，可能会陷入一些瓶颈：要么找不到合适的练习内容，要么觉得练习过于乏味。因此这个时候，可以切换不同类型的练习项目——如，做一些自己觉得有意思的小项目练习。又或者，当我们完成一个目标时，给自己一些奖励，以此来鼓舞自己。

总结

练习完之后，还有一种很好的提高方式，就是输出、总结。整理自己练习过程中学到的知识，将之与我们需要的技能做对比，我们就会发现：在哪些地方还需要提高。我们就能制作出下一次练习的目标，不断地反复，以此来提高自己。

GitChat



经常做总结，除了看到自己提高的地方，还能让阅读文章的人，鼓励你更好的前进。

那么，现在让我们创建一个项目，更新一次 README，开始练习吧！

GitChat