不止代码

工程师必备 职业发展黄金手册

Alibaba Group 阿里巴巴集团 | 阿里技术

阿里技术

扫一扫二维码图案，关注我吧

「阿里技术」微信公众号

「阿里巴巴机器智能」公众号

目录

如何快速成长为技术大牛？阿里资深技术专家的总结亮了 1

毕业 3 年，为何技术能力相差越来越大？ 10

程序员吃的是青春饭？本质上取决于…… 15

技术变化那么快，程序员如何做到不被淘汰？ 21

加班越久故障越多，如何跳出程序员的恶性循环？ 32

如何在阿里技术面试中脱颖而出？ 39

使用开源项目的正确姿势，血和泪的总结 48

前端工程师的未来在哪里？ 57

前端 Leader 如何做好团队规划？阿里内部培训总结公开 65

一位优秀前端的自我修养 76

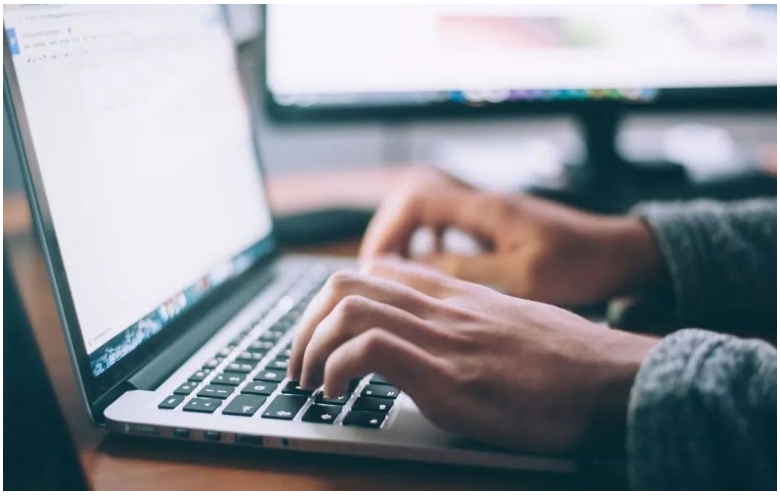
如何成为一名顶尖的阿里架构师？ 87

哪些技术好书值得一读再读？这有一份经典书单 94

阿里技术大牛最爱的“闲书”，你看过多少？ 103

如何快速成长为技术大牛？阿里资深技术专家的总结亮了

李运华



阿里妹导读：你是否有类似这样的问题——“天天写业务代码的程序员，怎么成为技术大牛，开始写技术代码?”今天，阿里资深无线开发专家李运华，系统梳理了自己的思考和理解，希望帮助更多同学少走一些弯路。

写在前面

不管是开发、测试、运维，每个技术人员心里多多少少都有一个成为技术大牛的梦，毕竟“梦想总是要有的，万一实现了呢”！正是对技术梦的追求，促使我们不断地努力和提升自己。

然而“梦想是美好的，现实却是残酷的”，很多同学在实际工作后就会发现，梦想是成为大牛，但做的事情看起来跟大牛都不沾边，例如，程序员说“天天写业务代码还加班，如何才能成为技术大牛”，测试说“每天都有执行不完的测试用例”，运维说“扛机器接网线敲 shell 命令，这不是我想要的运维人生”。

我也是一位程序员，所以我希望通过以下基于程序开发的一些例子，帮助大家解决这些困惑。大道理是相通的，测试、运维都可以借鉴。

几个典型的误区

拜大牛为师

有人认为想成为技术大牛最简单直接、快速有效的方式是“拜团队技术大牛为师”，让他们平时给你开小灶，给你分配一些有难度的任务。

我个人是反对这种方法的，主要的原因有几个：

- 大牛很忙，不太可能单独给你开小灶，更不可能每天都给你开 1 个小时的小灶；而且一个团队里面，如果大牛平时经常给你开小灶，难免会引起其他团队成员的疑惑，我个人认为如果团队里的大牛如果真正有心的话，多给团队培训是最好的。然而做过培训的都知道，准备一场培训是很耗费时间的，课件和材料至少 2 个小时（还不能是碎片时间），讲解 1 个小时，大牛们一个月做一次培训已经是很高频了。
- 因为第一个原因，所以一般要找大牛，都是带着问题去请教或者探讨。因为回答或者探讨问题无需太多的时间，更多的是靠经验和积累，这种情况下大牛们都是很乐意的，毕竟影响力是大牛的一个重要指标嘛。然而也要特别注意：如果经常问那些书本或者 google 能够很容易查到的知识，大牛们也会很不耐烦的，毕竟时间宝贵。经常有网友问我诸如“jvm 的 -Xmn 参数如何配置”这类问题，我都是直接回答“请直接去 google”，因为这样的问题实在是太多了，如果自己不去系统学习，每个都要问是非常浪费自己和别人的时间的。
- 大牛不多，不太可能每个团队都有技术大牛，只能说团队里面会有比你水平高的人，即使他每天给你开小灶，最终你也只能提升到他的水平；而如果是跨团队的技术大牛，由于工作安排和分配的原因，直接请教和辅导的机会是比较少的，单凭参加几次大牛的培训，是不太可能就成为技术大牛的。

综合上述的几个原因，我认为对于大部分人来说，要想成为技术大牛，首先还是要明白“主要靠自己”这个道理，不要期望有个像武功师傅一样的大牛手把手一步一步地教你。适当的时候可以通过请教大牛或者和大牛探讨来提升自己，但大部分时间还是自己系统性、有针对性的提升。

业务代码一样很牛逼

有人认为写业务代码一样可以很牛逼，理由是业务代码一样可以有各种技巧，例如可以使用封装和抽象使得业务代码更具可扩展性，可以通过和产品多交流以便更好的理解和实现业务，日志记录好了问题定位效率可以提升 10 倍等等。

业务代码一样有技术含量，这点是肯定的，业务代码中的技术是每个程序员的基础，但只是掌握了这些技巧，并不能成为技术大牛，就像游戏中升级打怪一样，开始打小怪，经验值很高，越到后面经验值越少，打小怪已经不能提升经验值了，这个时候就需要打一些更高级的怪，刷一些有挑战的副本了，没看到哪个游戏只要一直打小怪就能升到顶级的。成为技术大牛的路也是类似的，你要不断的提升自己的水平，然后面临更大的挑战，通过应对这些挑战从而使自己水平更上一级，然后如此往复，最终达到技术大牛甚至业界大牛的境界，写业务代码只是这个打怪升级路上的一个挑战而已，而且我认为是比较初级的一个挑战。

所以我认为：业务代码都写不好的程序员肯定无法成为技术大牛，但只把业务代码写好的程序员也还不能成为技术大牛。

上班太忙没时间自己学习

很多人认为自己没有成为技术大牛并不是自己不聪明，也不是自己不努力，而是中国的这个环境下，技术人员加班都太多了，导致自己没有额外的时间进行学习。

这个理由有一定的客观性，毕竟和欧美相比，我们的加班确实要多一些，但这个因素只是一个需要克服的问题，并不是不可逾越的鸿沟，毕竟我们身边还是有那么多的大牛也是在中国这个环境成长起来的。

我认为有几个误区导致了这种看法的形成：

1）上班做的都是重复工作，要想提升必须自己额外去学习

形成这个误区的主要原因还是在于认为“写业务代码是没有技术含量的”，而我现在上班就是写业务代码，所以我在工作中不能提升。

2）学习需要大段的连续时间

很多人以为要学习就要像学校上课一样，给你一整天时间来上课才算学习，而我们平时加班又比较多，周末累的只想睡懒觉，或者只想去看看电影打打游戏来放松，所以就没有时间学习了。

实际上的做法正好相反：首先我们应该在工作中学习和提升，因为学以致用或者有实例参考，学习的效果是最好的；其次工作后学习不需要大段时间，而是要挤出时间，利用时间碎片来学习。

正确的做法

Do more

做的更多，做的比你主管安排给你的任务更多。

我在 HW 的时候，负责一个版本的开发，这个版本的工作量大约是 2000 行左右，但是我除了做完这个功能，还将关联的功能全部掌握清楚了，代码（大约 10000 行）也全部看了一遍，做完这个版本后，我对这个版本相关的整套业务全部很熟悉了。经过一两次会议后，大家发现我对这块掌握最熟了，接下来就有趣了：产品讨论需求找我、测试有问题也找我、老大对外支撑也找我；后来，不是我负责的功能他们也找我，即使我当时不知道，我也会看代码或者找文档帮他们回答。最后我就成了我这个系统的“专家”了。虽然这个时候我还是做业务的，还是写业务代码，但是我已经对整个业务都很熟悉了。

以上只是一个简单的例子，其实就是想说：**要想有机会，首先你得从人群中冒出来，要想冒出来，你就必须做到与众不同，要做到与众不同，你就要做得更多！**

怎么做得更多呢？可以从以下几个方面着手：

1）熟悉更多业务，不管是不是你负责的；熟悉更多代码，不管是不是你写的

这样做有很多好处，举几个简单的例子：

- 需求分析的时候更加准确，能够在需求阶段就识别风险、影响、难点
- 问题处理的时候更加快速，因为相关的业务和代码都熟悉，能够快速的判断问题可能的原因并进行排查处理
- 方案设计的时候考虑更加周全，由于有对全局业务的理解，能够设计出更好的方案

2）熟悉端到端

比如说你负责 web 后台开发，但实际上用户发起一个 http 请求，要经过很多中间步骤才到你的服务器（例如浏览器缓存、DNS、nginx 等），服务器一般又会经过很多处理才到你写的那部分代码（路由、权限等）这整个流程中的很多系统或者步骤，绝大部分人是不可能去参与写代码的，但掌握了这些知识对你的综合水平有很大作用，例如方案设计、线上故障处理这些更加有含金量的技术工作都需要综合技术水平。

“系统性”、“全局性”、“综合性”这些字眼看起来比较虚，但其实都是技术大牛的必备的素质，要达到这样的境界，必须去熟悉更多系统、业务、代码。

3）自学

一般在比较成熟的团队，由于框架或者组件已经进行了大量的封装，写业务代码所用到的技术确实也比较少，但我们要明白“唯一不变的只有变化”，框架有可能要改进，组件可能要替换，或者你换了一家公司，新公司既没有组件也没有框架，要你从头开始来做。这些都是机会，也是挑战，而机会和挑战只会分配给有准备的人，所以这种情况下我们更加需要自学更多东西，因为真正等到要用的时候再来学已经没有时间了。

以 java 为例，大部分业务代码就是 if-else 加个数据库操作，但我们完全可以自己学些更多 java 的知识，例如垃圾回收，调优，网络编程等，这些可能暂时没用，但真要用的时候，不是 google 一下就可以了，这个时候谁已经掌握了相关知识和技能，机会就是谁的。

以垃圾回收为例，我自己平时就抽时间学习了这些知识，学了 1 年都没用上，但

后来用上了几次，每次都解决了卡死的大问题，而有的同学，写了几年的 java 代码，对于 stop-the-world 是什么概念都不知道，更不用说去优化了。

Do better

要知道这个世界上没有完美的东西，**你负责的系统和业务，总有不合理和可以改进的地方**，这些“不合理”和“可改进”的地方，都是更高级别的怪物，打完后能够增加更多的经验值。识别出这些地方，并且给出解决方案，然后向主管提出，一次不行两次，多提几次，只要有一次落地了，这就是你的机会。

例如：

重复代码太多，是否可以引入设计模式?

系统性能一般，可否进行优化?

目前是单机，如果做成双机是否更好?

版本开发质量不高，是否引入高效的单元测试和集成测试方案?

目前的系统太庞大，是否可以通过重构和解耦改为 3 个系统?

阿里中间件有一些系统感觉我们也可以用，是否可以引入 ?

只要你去想，其实总能发现可以改进的地方的；如果你觉得系统哪里都没有改进的地方，那就说明你的水平还不够，可以多学习相关技术，多看看业界其它优秀公司怎么做。

我 2013 年调配到九游，刚开始接手了一个简单的后台系统，每天就是配合前台做数据增删改查，看起来完全没意思，是吧? 如果只做这些确实没意思，但我们接手后做了很多事情：

- 解耦，将一个后台拆分为 2 个后台，提升可扩展性和稳定性；
- 双机，将单机改为双机系统，提高可靠性；
- 优化，将原来一个耗时 5 小时的接口优化为耗时 5 分钟

还有其它很多优化，后来我们这个组承担了更多的系统，后来这个小组 5 个人，负责了 6 个系统。

Do exercise

在做职业等级沟通的时候，发现有很多同学确实也在尝试 Do more、Do better，但在执行的过程中，几乎每个人都遇到同一个问题：**光看不用效果很差，怎么办?**

例如：

- 学习了 jvm 的垃圾回收，但是线上比较少出现 FGC 导致的卡顿问题，就算出现了，恢复业务也是第一位的，不太可能线上出现问题然后让每个同学都去练一下手，那怎么去实践这些 jvm 的知识和技能呢?
- Netty 我也看了，也了解了 Reactor 的原理，但是我不可能参与 Netty 开发，怎么去让自己真正掌握 Reactor 异步模式呢?
- 看了《高性能 MySQL》，但是线上的数据库都是 DBA 管理的，测试环境的数据库感觉又是随便配置的，我怎么去验证这些技术呢?
- 框架封装了 DAL 层，数据库的访问我们都不需要操心，我们怎么去了解分库分表实现?

诸如此类问题还有很多，我这里分享一下个人的经验，其实就是 3 个词：learning、trying、teaching！

1）Learning

这个是第一阶段，看书、google、看视频、看别人的博客都可以，但要注意一点是“系统化”，特别是一些基础性的东西，例如 JVM 原理、Java 编程、网络编程，HTTP 协议等等，这些基础技术不能只通过 google 或者博客学习，我的做法一般是先完整的看完一本书全面的了解，然后再通过 google、视频、博客去有针对性的查找一些有疑问的地方，或者一些技巧。

2）Trying

这个步骤就是解答前面提到的很多同学的疑惑的关键点，形象来说就是“自己动手丰衣足食”，也就是自己去尝试搭建一些模拟环境，自己写一些测试程序。例如：

- Jvm 垃圾回收：可以自己写一个简单的测试程序，分配内存不释放，然后调整各种 jvm 启动参数，再运行的过程中使用 jstack、jstat 等命令查看 jvm 的堆

内存分布和垃圾回收情况。这样的程序写起来很简单，简单一点的就几行，复杂一点的也就几十行。

- Reactor 原理：自己真正去尝试写一个 Reactor 模式的 Demo，不要以为这个很难，最简单的 Reactor 模式代码量（包括注释）不超过 200 行（可以参考 Doug Lee 的 PPT）。自己写完后，再去看看 netty 怎么做，一对比理解就更加深刻了。
- MySQL：既然有线上的配置可以参考，那可以直接让 DBA 将线上配置发给我们（注意去掉敏感信息），直接学习；然后自己搭建一个 MySQL 环境，用线上的配置启动；要知道很多同学用了很多年 MySQL，但是连个简单的 MySQL 环境都搭不起来。
- 框架封装了 DAL 层：可以自己用 JDBC 尝试去写一个分库分表的简单实现，然后与框架的实现进行对比，看看差异在哪里。
- 用浏览器的工具查看 HTTP 缓存实现，看看不同种类的网站，不同类型的资源，具体是如何控制缓存的；也可以自己用 Python 写一个简单的 HTTP 服务器，模拟返回各种 HTTP Headers 来观察浏览器的反应。

还有很多方法，这里就不一一列举，简单来说，就是要将学到的东西真正试试，才能理解更加深刻，印第安人有一句谚语：**I hear and I forget. I see and I remember. I do and I understand**，而且“试试”其实可以比较简单，很多时候我们都可以自己动手做。

当然，如果能够在实际工作中使用，效果会更好，毕竟实际的线上环境和业务复杂度不是我们写个模拟程序就能够模拟的，但这样的机会可遇不可求，大部分情况我们还真的只能靠自己模拟，然后等到真正业务要用的时候，能够信手拈来。

3）Teaching

一般来说，经过 Learning 和 Trying，能掌握 70% 左右，但要真正掌握，我觉得一定要做到能够跟别人讲清楚。因为在讲的时候，我们既需要将一个知识点系统化，也需要考虑各种细节，这会促使我们进一步思考和学习。同时，讲出来后看或者听的人可以有不同的理解，或者有新的补充，这相当于继续完善了整个知识技能体系。

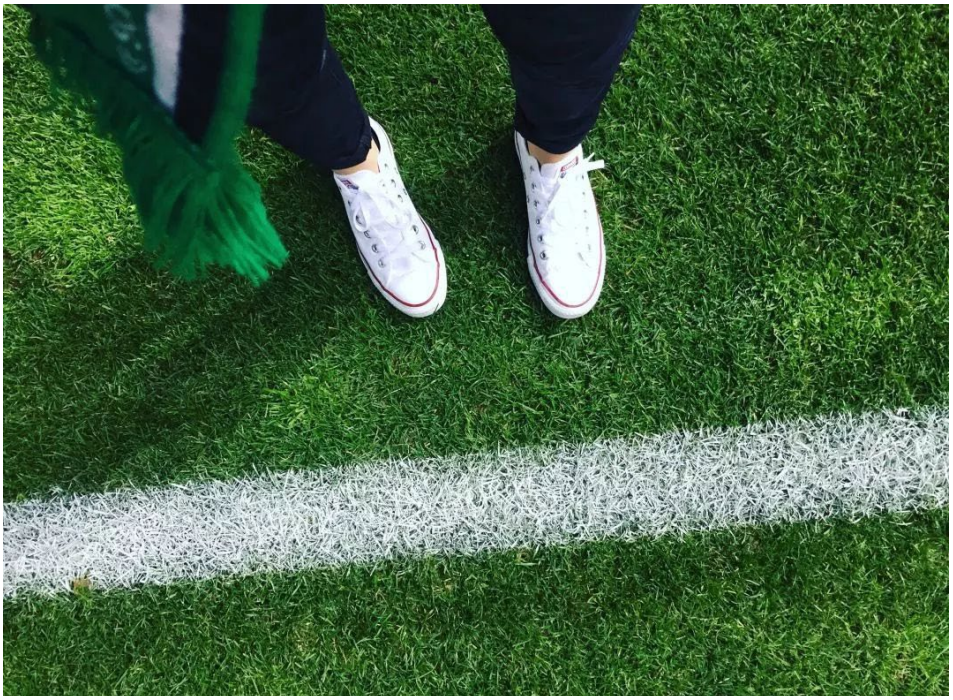
这样的例子很多，包括我自己写博客的时候经常遇到，本来我觉得自己已经掌握很全面了，但一写就发现很多点没考虑到；组内培训的时候也经常看到，有的同学写了 PPT，但是讲的时候，大家一问，或者一讨论，就会发现很多点还没有讲清楚，或者有的点其实是理解错了。写 PPT、讲 PPT、讨论 PPT，这个流程全部走一遍，基本上对一个知识点掌握就比较全面了。

后记

成为技术大牛梦想虽然很美好，但是要付出很多，不管是 Do more 还是 Do better 还是 Do exercise，都需要花费时间和精力，这个过程中可能很苦逼，也可能很枯燥，这里我想特别强调一下：前面我讲的都是一些方法论的东西，但真正起决定作用的，其实还是我们对技术的热情和兴趣！

毕业 3 年，为何技术能力相差越来越大?

蛰剑



阿里妹导读：毕业三年，每个人在技术能力跑道上，有了或大或小的差距。有些人永远在重复的劳动，有些人却能从中总结和解决问题。今天我们来探讨下，如何避免让战术上的勤奋掩盖战略上的懒惰，使得真正掌握好的知识点慢慢生长，连接，最终组成一张大网。

写在前面

高考的时候大家都是一样的教科书，同一个教室，同样的老师辅导，时间精力基本差不多，可是最后别人考的是清华、北大或者一本，而有些童鞋的实力只能考个三本，这是为什么?

关键问题点

为什么你的知识积累不了?

有些知识看过就忘、忘了再看，实际碰到问题还是联系不上这个知识点。这其实是知识的积累出了问题，没有深入理解好，自然就不能灵活运用，也就谈不上解决问题。大家一起看相同的高考教科书但是高考结果不一样，问题出在了理解上。每个人的理解能力不一样（智商），绝大多数人对知识的理解要靠不断地实践（做题）来巩固。

同样实践，效果不一样?

同样工作一年碰到了 10 个问题（或者说做了 10 套高考模拟试卷），但是结果不一样，那是因为在实践过程中方法不够好。或者说你对你为什么做对了，为什么做错了，没有去复盘。

假如碰到一个问题，身边的同事解决了，而我解决不了。那么我就去想这个问题他是怎么解决的，他看到这个问题后的逻辑和思考是怎么样的？有哪些知识指导了他这么逻辑推理，这些知识哪些是我也知道但是我没有想到这么去运用推理的（说明我对这个知识理解得不到位导致缺乏灵活运用）；这些知识中又有哪些是我不知道的（知识缺乏，没什么好说的，快去搜索学习下——有场景案例和目的加持，学习理解起来更快）。

等你基本把这个问题按照你同事掌握的知识和逻辑推理想明白后，需要再去琢磨一下他的逻辑推理、解题思路中有没有不对的，有没有啰嗦的地方，有没有更直接的方式（对知识更好地运用）。

我相信每个问题都这么去实践的话就不应该再抱怨。灵活运用、举一反三，同时知识也积累下来了，这种场景下积累到的知识是不会那么容易忘记的。

这就是向身边的牛人学习，同时很快超过他的办法。这就是为什么高考前你做了 10 套模拟题还不如其他人做一套的效果好。

知识 + 逻辑就基本等于你的能力，知识让你知道那个东西，逻辑让你把东西和问题联系起来。

这里的问题你可以理解成**方案、架构、设计**等。

系统化的知识哪里来?

知识之间是可以联系起来的并且像一颗大树一样自我生长，但是当你都没理解透彻，自然没法产生联系，也就不能够自我生长了。当我们讲到入门了某块知识的时候一般是指对关键问题的点理解清晰，并且能够自我生长，也就如滚雪球一样可以滚起来了。

好的逻辑又怎么来?

- 实践
- 复盘

讲个前同事的故事

我有一个前同事，所有解决不了的问题都找他。这位同学让我最佩服的是解决问题的能力，好多问题其实他也不一定擅长，但是他就是有本事通过 Help、Google 不停地验证、尝试就把一个不熟悉的问题给解决了，这是我最羡慕的能力，在后面的职业生涯中一直不停地往这个方面尝试。

应用刚启动连接到数据库的时候比较慢，但又不是慢查询

1. 这位同学的解决办法是通过 tcpdump 来分析网络通讯包，看具体卡在哪里把这个问题硬生生地给找到了。
2. 如果是专业的 DBA 可能会通过 show processlist 看具体连接在做什么？比如看到这些连接状态是 authentication 状态，然后再通过 Google 或者对这个状态的理解知道创建连接的时候 MySQL 需要反查 IP、域名，这里比较耗时，通过配置参数 skip-name-resolve 跳过去就好了。
3. 如果是 MySQL 的老司机，一上来就知道 skip-name-resolve 这个参数要改改默认值。

在我眼里这三种方式都解决了问题，最后一种最快但是纯靠积累和经验，换个问题也许就不灵了；第一种方式是最牛逼和通用，只需要最少的业务知识。

我当时跟着他从 sudo、ls 等 linux 命令开始学起。当然我不会轻易去打搅他、问他，每次碰到问题我尽量让他在我的电脑上来操作，解决后我再自己复盘，通过 history 调出他的所有操作记录，看他在我的电脑上用 Google 搜啥了，然后一个个去学习分析他每个动作，去想他为什么搜这个关键字，复盘完还有不懂的再到他面前跟他面对面地讨论他为什么要这么做，指导他这么做的知识和逻辑又是什么。

空洞的口号

很多文章都会教大家：举一反三、灵活运用、活学活用、多做多练。但是只有这些口号是没法落地的，落地的基本原则就是前面提到的，却总是被忽视了。

什么是工程效率，什么是知识效率

有些人纯看理论就能掌握好一门技能，还能举一反三，这是知识效率，这种人非常少。

大多数普通人都是看点知识，然后结合实践来强化理论，要经过反反复复才能比较好地掌握一个知识，这就是工程效率，讲究技巧、工具来达到目的。

肯定是知识效率最牛逼，但是拥有这种技能的人毕竟非常少（天生的高智商吧）。从小我们周边那种不怎么学的学霸型基本都是这类，这种学霸都还能触类旁通非常快地掌握一个新知识，非常气人。剩下的绝大部分只能拼时间 + 方法 + 总结等，也能掌握一些知识。

非常遗憾我就是工程效率型，只能羡慕那些知识效率型的学霸。但是这事又不能独立看待，有些人在某些方向上是工程效率型，有些方向就又是知识效率型（有一种知识效率型是你掌握的实在太多，也就比较容易触类旁通了，这算灰色知识效率型。）

使劲挖掘自己在知识效率型方面的能力吧，两者之间当然没有明显的界限，知识积累多了，逻辑训练好了，在别人看来你的智商就高了。

知识分两种

一种是通用知识（不是说对所有人通用，而是说在一个专业领域去到哪个公司都能通用），另外一种是跟业务公司绑定的特定知识。

通用知识没有任何疑问，碰到后要非常饥渴地扑上去掌握他们（受益终生，这还有什么疑问吗？）。对于特定知识就要看你对业务需要掌握的深度了，肯定也是需要掌握一些的，特定知识掌握得好的，一般在公司里混得也会比较好。

程序员吃的是青春饭？本质上取决于……

毗卢

阿里妹导读：你是否曾经认真思考过——毕业 3-5 年、10 年，乃至更久后，我们希望成为什么样的人？作为一名技术人，我们要如何规划自己的职业发展生涯？网上热议的“35 岁中年危机”，本质上又因什么而焦虑？今天，阿里资深技术专家毗卢，将带来自己的思考与理解，希望对大家有所启发。

毗卢：近期，我与团队同学探讨了职业发展规划的问题。有些同学表示希望后续能进一步在技术领域（或管理方向）有进一步的积累；有的同学表示希望在新的一年能具有更好的技术影响力，自己能做一些技术决定，去影响其他人，这样自己会很有成就感。

因此，我也问了一些问题：

- 你希望技术能进一步积累，那你积累的方向和期望达到的结果分别是啥?
- 你希望能有技术决策，希望有影响力，你觉得应该如何做到？是希望通过岗位任命的方式吗?
- 你觉得是否成功的标志，就是今年或明年得到晋升吗?
- 等等

大部分同学在面对这些问题时，其实是比较迷茫的，也缺少真正可度量的衡量标准。是否能在短期内获得晋升成了大部分人作为“组织是否认可、自己是否认可”的衡量标准了。

当然，这个话题仁者见仁、智者见智，这里我简单地谈谈我的看法。我以相对比较口水化的方式，将职业发展分两个阶段来进行阐述：

1）第一阶段：大学毕业 3 到 5 年

2）第二阶段：大学毕业 5 到 10 年

第一阶段：大学毕业 3 到 5 年

对于从事 Java 软件开发的技术同学，在毕业后的 3 到 5 年内主要都是以学习、积累为主。这个阶段的工作几乎每天都有惊喜，都有收获。从一开始啥都不懂的校园“新鲜人”向“职业人”转变。在这个阶段，你会学习：

- 基础的 Java 知识：你会开始看《Java 编程思想》、《Effective Java》。
- 高质量代码进阶知识：你会开始看《重构：改善既有代码的设计》、《代码大全》、《编程珠玑》。
- 常用的主流框架：比如 SSH 相关的《Spring 实战》、《Spring Boot 实战》、《Hibernate 实战（第 2 版）》。当然，这些书已经不够了，你会通过 Google、Baidu 大量地浏览在线的资源：Apache 官网、Spring 官网、Hibernate 官网。你会去 StackOverflow 问问题或找答案。
- 系统设计与算法知识：《系统分析与设计方法》、《设计模式》、《需求分析与系统设计》、《面向对象分析与设计》、《UML 用户指南》、《算法导论》
- 其他知识：比如数据库调优、缓存框架、NoSQL 数据库、日志框架等等

在这 5 年间，快速地完成这些基础知识的学习，并能在项目中快速地学以致用。不仅自身能获得比较高的成就感，而且实际的用人的单位、猎头也会非常喜欢这类熟练工。

从大部分人的实际发展轨迹看，这个阶段发展快的人和正常发展速度的人，差别还不是很大。比如，发展非常快的人，从毕业就入职阿里的 P5 到 P7（注：阿里内部职称评级），可能三年就可以做到。发展速度正常的人，可能需要 5-6 年也可以到 P7。也就是说，这个阶段正常发展速度的同学也仅仅比发展速度快的人慢 2-3 年而已。

这 2 到 3 年的差距，是可以通过有针对性的学习、重大项目的历练等完成这些知识的学习。无非是，有的同学会严格要求自己，有严格的学习计划；有的同学赶早参加了一些重点的、痛苦的项目得到了锻炼。只要是做技术的，其实迟早都会经历过，都会成长起来。

发现没有？这个阶段，我们能协调好的资源其实就是自己，更多的是一个“个人贡献者”。只要把自己管好了，学习计划执行好了，工作高质量做好了就能得到认可。

第二阶段：大学毕业 5 到 10 年

很多本科同学，特别是研究生同学。在毕业 10 年后，就已经到了 34、35 岁左右了。也是前段时间网上广泛讨论的所谓 34+ 岁现象。**其实，年龄并不是问题的真正原因。真正的原因还是在于自身“竞争力”是否符合这个年龄所应该具备的。**

到了这个年龄的人，往往已经不是“个人贡献者”了，而是“团队贡献者”。团队贡献者可能是带团队的 TL，也可能是个架构师，在技术决策上具有团队影响力和话语权。

那么，为什么这些人能管理团队或者有影响力呢?

从公司的经营视角看，一个管理团队的人，他必须为业务的成功负责。说个大白话，一个 TL 管了 N 个人，他至少要能保证大家输出所产生的价值，至少要高于这个团队的工资、奖金、五险一金、OPEX、CAPEX 等等吧。这个 TL 为了大家输出得

有价值，他是不是需要能：

- 能对所负责领域的业务特点、发展趋势、友商竞争分析有很好的洞察？能知道这个业务领域的客户是谁？他们的需求是什么？他们的痛点是什么？所以，这个 TL 应该需要学习《咨询的奥秘》、《探索需求》、《系统化思维导论》。对于技术型的 TL，还应该了解《成为技术领导者：掌握全面解决问题的方法》。
- 服务于特定领域的客户，我们需要能了解我们的客户企业架构、业务知识。要了解清楚规划的产品、服务，什么才是客户所需要的。那么，从理论上，我们是否应该学习一些 TOGAF、NGOSS、ITIL 等业务理论以及业务知识？
- 作为 TL，是否有必要能将自己对于市场的洞察转换成业务规划，并能向自己的老板（或者投资人）说清楚、讲明白？并争取到老板的同意，包括资金、人力资源等。对于，能否把事情讲明白，我们可能需要学习《金字塔原理》，并能非常清晰、有逻辑性地进行表达与沟通。当然，有些业务发展的事不一定特别有逻辑，是需要摸索、尝试，那么你是否能将一个不确定的领域说服老板并获得支持，我们又需要什么？《博弈论》、《影响力》等。
- 获得老板支持后，就需要开始带着兄弟们干活了。作为带头人，你看我们是否需要能将业务趋势、客户痛点进行业务建模好让团队的 PD、技术都能理解？在做业务进一步深入分析，可能就需要学习《领域驱动设计：软件核心复杂性应对之道》、《实现领域驱动设计》、《企业应用架构模式》、《恰如其分的软件架构》等等。
- 做完业务设计后，开始要带着团队做技术方案设计、接口设计以及编码实现等。这个过程，TL 又需要具备软件项目管理的能力。无论是《PMBOK 指南》，还是《敏捷软件开发》、《人月神话》、《程序开发心理学》，相信总归还是会有点帮助的。
- 对于一些有国际化要求的公司，还需要再学习英语吧！
- 嗯，还需要有个好的身体，还需要经常锻炼，学习科学的健身吧（说起来自己脸红）。至少我明白了一个道理，以前我都是跟自己说，等这段时间过了，闲下来去锻炼一下。其实，我发现，越是忙的时候，越需要锻炼身体！

- 另外，在这 10 年内，比较关键的是——你还经历过什么有挑战的业务、技术、产品、平台等方面的成功与失败经验？在这些经历里，你可能会遇到这些困难与挑战：团队磨合的挑战、技术方案上的争执、平台优先 or 业务优先的博弈、低落的团队氛围、个人的低谷等等。这些困难与挑战，你是退缩了？还是有成长？在带团队时，再次面临这些挑战时，这时你是否有解或者有勇气了？

发现没有？毕业 10 年后，作为一个团队贡献者，你可能需要具备这些能力，并且还远远不止。而且，更可悲的时，当毕业 10 年后，突然发现自己不具备这个能力时（比如晋升失败时发现了），这些能力 GAP 就不再是 2 到 3 年就能追得上的了。我见过一些有准备的同学，他们给自己的目标是在毕业第 7 年就要具备这些能力，他有严格的学习计划、实践计划、甚至是冒险的创业经历。当他到第 10 年这个点时，这些高阶技能很可能已经有 3 年的实践经验了。

如果我们没有做好准备，10 年后，如何和这批人竞争？这些软、硬知识，从十年这个时间刻度倒排，学习计划、实践计划的执行还是很紧张的。所以，从现在开始给自己制定一个严格的学习计划、严格执行，多实践吧！

从价值出发 – 找寻学习与工作的新思路

迷茫能引发思考，架构则塑造了视野，而价值则是我们之所以存活，之所以工作的逻辑起点。基于这样一种价值思维，对我们的学习和工作又可以有哪些改启示呢?

明确自身的业务相关主体：找出你工作的协作关系网内的业务方和客户方，这样你就可以从客户方中找到离你最近的业务价值点，从你的业务方中挖掘更多的资源。甚至你可以按这个思路顺着网络向上或向下挖掘价值链条，整合更多的上下游资源以实现更大的价值。

向前一步，为更大的价值负责：不要因为自己是开发人员就不去关注软件运维，不要因为只是测试就不关注软件开发，因为你关注的越多你越能看清全局的价值目标。如果只关注一亩三分地，那么注定这辈子只能困守在这一亩三分地里，成为一名流水线上焦虑至死的码农。试着转变思维，从架构师的角度思考价值问题，看看能否将技术贯穿到业务、到用户、到最终的价值去。之前我的朋友说过要把产品经理踢到运营位置去，把程序员踢到产品经理位置去，这样才是正确做事方式。这句话也是类似的意思，向前一步才能懂得怎么做的更好。

像架构师一样思考，用价值找寻重心：人的迷茫是因为找不到重心，而价值的意义在于引导我们思考做哪些事情才能实现价值，先做哪些事情会比后做哪些事情更能创造收益。像架构师那样全局性思考，把遇到问题进行拆分，把学习到的事物串联起来，努力构成完整的价值链条。

学会连接，构建体系：前几天看到一篇文章对今日头条的产品形态极尽批判之词，指责它的智能算法将人类封死在自己的喜好之中，将人类社会进一步碎片化。这似乎很有道理，有趣的是互联网将我们连接至广袤的世界，却也把我们封闭在独属于自己的小世界里。依旧是我的那位朋友，他说他的最大价值在于连接，将不同的人连接在一起，有趣的事情可能就会即将发生。

或许算法的天性就是顺从与迎合，但人最终想理解这个世界还是需要依靠自身的行动与不同人之间建立联系，这也是一种摆脱流水线限制的有效方式。另外，我们自身也是某种事物连接的产物，比如架构师，他是业务、技术、管理连接在一起的一种产物。所以我们应当树立自身的知识体系以吸收融合新知识，将孤立的概念连接起来，形成自身的价值链条。比如这篇文章将我从事技术开发经验、与对架构的理解以及自身过往经历结合起来，这也是一种内在的体系梳理。

作者简介：空融，网名“D 调的暖冬”。现就职蚂蚁金服，从事支付宝身份认证相关领域的技术开发。

加班越久故障越多，如何跳出程序员的恶性循环？

冠楠

阿里妹导读：如何让每一位可爱的工程师少加班、不加班？阿里巴巴技术专家张冠楠，在质量保障体系建设、持续集成领域、敏捷实践领域和研发效能领域方面具有丰富的经验和心得。今天，冠楠将用阿里研发团队的实际案例，生动说明如何用数据驱动研发效率提升。

本文是我利用云效公有云度量功能，加上敏捷部分的方法指导，实践于某事业部几十人团队沉淀的成果，希望能给大家一些借鉴意义。我会就各种具有关键表征的数据进行介绍，但是详细数据，包括具体研发团队的数据，还需要访问云效公有云度量功能页面。

注：本文图片数据来自云效度量数据功能，有兴趣的童鞋可点击文末“阅读原文”了解更多。

数据展现

先直接给大家数据，我是 4 月份开始进入这个团队的。大家重点看这个团队 3 月份的数据：

问题分析

上面几张图比较容易看出来，这个团队的明显特征是：

- 3 月份完成需求数明显上升，且团队负载较重。
- 质量不高 – 缺陷数、reopen 率以及线上发布成功率。
- 需求平均完成时长特别长。
- 突增故障。

于是我们带着数据暴露出来的这几个问题，和团队一线研发人员、PD、TL 进行沟通，分析数字背后的意义。大家很快达成一致，发现团队存在的主要问题是：

- 需求 deliver 传统瀑布模型，要 1 个半到 2 个月去完成一个特别大的需求，最后却和用户期望偏差较大，数据表征上就是之前需求数量较少，3 月份突然完成了很多而且时间很长的需求。
- 大家加班加点干活，负载较重，引入的缺陷也较多，PD 和用户不满意带来的修改又会加重工作量，如此恶性循环。
- 缺陷重视度不高，管理不规范，优先级划分不清楚，甚至残留重要缺陷，留在 bug 列表里未解决而流到了线上引发故障。

上面三点形成了恶性循环，结果就是越做越多，越多越错，越错越改，越改越多。

解决方案落地和数据运营

发现问题之后，有针对性的进行解决和落地就相对容易，我们给到团队的解决方案是：

- 需求细化：拆分成最小可交付产出，尽量避免一个需求做了 1 个多月，才去找 PD 和用户验收。
- 随时拥抱用户：迭代式产出，交付即验收，让不准确性降到最低，在错误误差最小的时候修正。
- 重点跟进质量管理和运营：透明数据，鼓励团队尽早尽快修复 bug，并有严格的上线前 bug 解决率标准。
- 尽全力保证线上发布成功率。

同时辅助于团队的决策，我们进行定期的数据运营，每周都会去统计和分析数据，包括质量和效率相关的，确保我们能在第一时间发现问题，纠正偏差。所以在 3 个多月的时间里，我重点关注了如下数据。关于这些数据的解读和分析，内容比较深

入，我这里只做简单的概括性介绍：

- 需求的吞吐量：团队指定时间段内完成的需求数，可大体反应出团队的产出趋势。
- 需求的平均完成时长：需求从创建到终态的平均时长，时间越多，需求交付粒度越小效率越高。
- 新增缺陷的数量 ：统计时间段内团队被新增指派的缺陷数量，结合存量缺陷以及缺陷平均解决时长，反应团队产品的质量以及对于缺陷解决的效率。
- 缺陷的平均解决时长 ：缺陷从创建到解决的平均时长，表征解决缺陷的效率。
- 线上发布的成功率：线上发布成功次数与总次数之比，越高证明产品上线质量越高。
- 缺陷的 reopen 率 ：缺陷被 reopen 的次数与缺陷数目之比，该值越高证明修复缺陷的质量越差，reopen 率是表征产品质量的一个重要指标。

结果分析和总结

大家回到上面的 6 张图以及下面的一张缺陷解决时间图，我们 3 月底进入，重点看从 4 月份开始的数据：

- 团队的负载得到了控制，需求的完成数下降了，后续 3 个月保持一个相对平稳的状态。
- 需求细化拆分后，交付的时长下降了，团队以更快的速率去和用户交付需求。
- 缺陷的数量下降，reopen 率下降，线上发布成功率上升，质量在好转。
- 缺陷的平均解决时间明显上升，团队更快的交付，更快的反馈问题，更快的解决问题。

总体而言，就是需求交付的快，得到的反馈快，修正错误 / 缺陷的成本低，缺陷也慢慢收敛，质量也随之提升，缺陷修复的也快了，这就是一个良性循环，概括总计就是：效率提高了，质量也保证了。团队的人干活也是更加努力啦！

如何进一步提升？

根据对需求数量以及平均完成时长的数据显示，团队还是有上升空间的，对于需求的交付粒度和速率上，还是略显波动，要想更快的知道我们做的是否是用户需要的，就要快速的、迭代式的交付需求，以免用户想要个车，我们给了他 4 个轮子。

所以能否彻底解决此团队需求的交付和用户期望偏差的问题，还是需要再向前走一步，需求继续细化，提升交付速率。 参见敏捷中推荐的，快速迭代，快速交付，快速得到用户反馈，只为了更快更准确。

总结

数据有魅力，研发数据也一样，我们使用它就是为了两个目的：一是保证质量；二是确保交付的速率。行走过程中深度使用了云效度量新功能，结合敏捷中部分理念，配合传统测试方式保障，来助力研发团队。

可能有的人会质疑，需要用这么冰冷冷的数字去衡量我们可爱的程序员哥哥吗？我的回答是：这不是衡量。数据只是手段，是帮助我们去诊断团队的一个切实有效的手段。学会利用它并驾驭它。因此我们只需要：

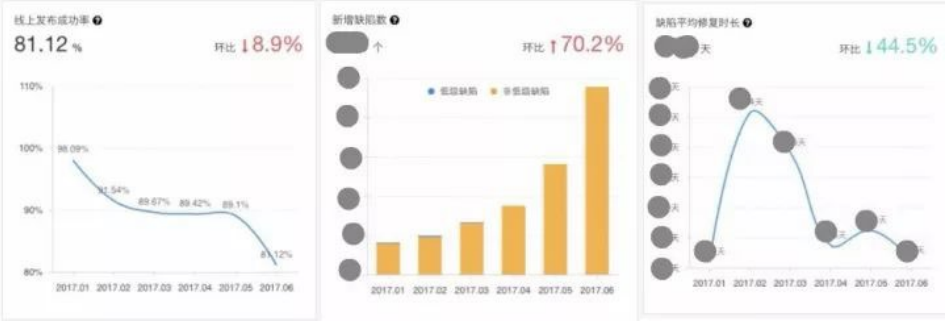
- 关注数据，读懂数据。
- 重点问题重点解决，优先解决，一段时间只关注一个或很少的几个问题。
- 相信团队的自驱能力，同时结合 TL 的管理与激励，养成良好的团队建设力。

欢迎交流讨论

研发团队每天打交道最多的就是需求、缺陷、代码、发布、应用、测试等等，这些和我们研发人员息息相关的数据，云效现在以研发大盘、团队空间、人员效能、质量分布等多种维度数据整合到了数据平台上，后续更会以定制化的方式满足研发团队对于研发数据的需求。利用好这个工具，能帮助我们清晰的了解团队的现状，暴露问题，找到改进措施，提升团队效率和产品质量。

我是一个敏捷爱好者，在深入研发团队做测试以及质量管理的时候，也是吸取和借鉴了敏捷的部分思想去落地。我的感受是：拿最切实有用比如站会、看板、快速迭代式交付需求，再加上数据辅助，都是能帮助到团队更快、更准确的交付高质量产品的手段。

最后贴几张我在度量上截的某研发团队的数据展示，这个团队是我们最近接触的团队，通过数据我们对这个团队的推测是：团队在质量上需要提升，在缺陷的管理上需要加强。首先团队缺陷的数量逐月上升，这已经是质量不好的趋势体现。

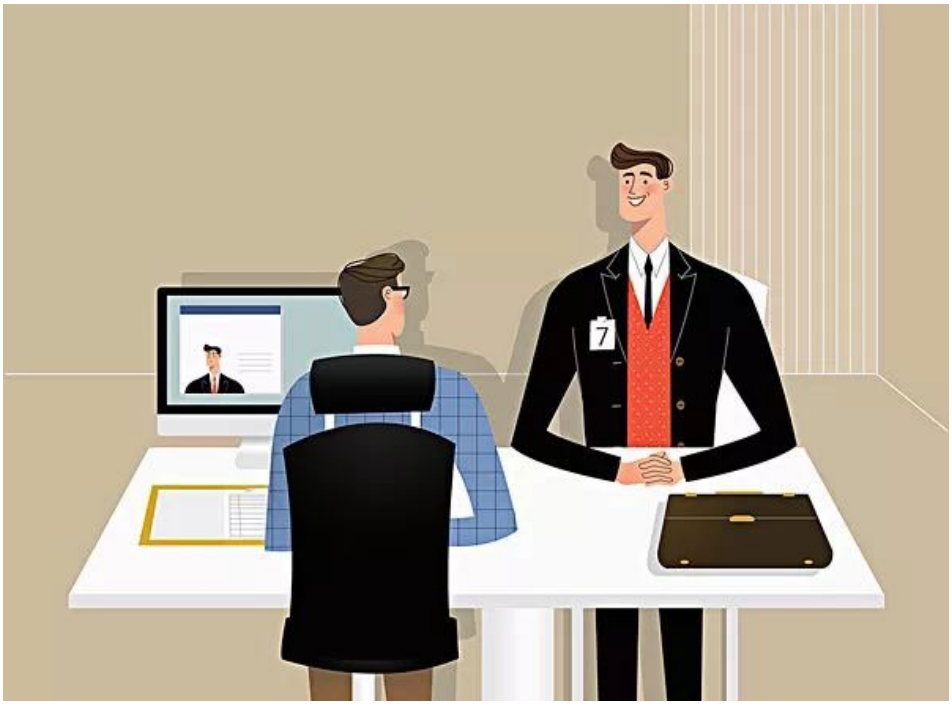


另外缺陷的解决时间也没有加快，这样会导致越来越多的缺陷流到线上去，可见团队除去 1 月份无故障，后续几个月都有故障。而且这个团队的线上发布成功率持续走低，开发对上线的代码把控程度较低。 所以，找到这些数据表征的背后原因，并且着手去解决掉，是这个团队近期最迫切的事情了。

养成良好的研发习惯，保持高效的团队协作，应该是每个研发同学持之以恒的追求。

如何在阿里技术面试中脱颖而出?

宗心



阿里妹导读：知己知彼、百战不殆，面试也是如此。只有充分了解面试官的思路，才能更好地在面试中充分展现自己。今天，阿里高级技术专家将分享自己作为面试官的心得与体会。如果你是面试者，可以借此为镜，对照发现自己的长处与不足，有针对性学习成长；如果你是面试官，相信也能通过他的讲述，有所启发。

招聘是团队管理者工作中的重要一环。本文会结合自己亲身经历以及接受的招聘培训，综合分析怎么找到我们要的人，也希望可以通过招聘这面镜子照亮自己，怎样成为一个更好的工程师。

招聘的目的

当今社会，技术已经成为影响商业成功的关键因素，工程师成为了这些公司最宝贵的财富，没有优秀的人组成团队来完成商业目标，公司根本不可能有今天的成就。所以招聘，就是选择最优秀的人。

招什么样的人？

招优秀的人显然是一个很模糊的概念，我们来度量的时候，我个人认为三个因素是最关键的：

- 技能

工作项目经验，以及解决疑难问题的能力，毕竟招来的人首先必须很好的完成工作，这是最基本的要求，注意，是很好的完成，不是仅仅完成。

- 潜力

这个概念看起来比较模糊，其实还是比较容易评价的，对计算机相关的专业的知识体系是不是完整，基础是不是扎实，平常是不是喜欢钻研，对这个世界充满好奇心，这几年走下来，沉淀的速度如何，都是判断一个人的潜力的方式，注意我们看潜力主要是基于候选人的之前的成长经历实事求是来看，**过去的优秀经历才能给未来背书**。潜力和技能的重要性一样重要，我们不能只看眼前，团队是需要不断发展和前进的，所以我们招人应该面向未来。

- 软实力

软实力这里其实包含了性格，执行力，领导力等方方面面，它代表了候选人是否能快速融入团队，拿到结果，带领团队攻城拔寨，激励和影响身边的人变得更加优秀等等，软实力一般 HR 肯定会考察，虽然技术面不会特别去关注，但是从面试的过程中可以看出候选人的沟通能力，以及性格相关的特点，也值得我们注意。

说了这么多，其实在招人上有一个对比的标杆，就是你招的人是不是**比团队中同一等级中 50% 的同学优秀**，如果你觉得没有他们优秀，那不用纠结，这个候选人不要了，团队必须不停加入更好的同学，才能变得更加强大。

面试的方法

这里结合之前的培训以及自己的真实经历，讲解面试的一些方法。

面试不要做的事

- 问一些知道性的问题比如问知不知道这个 API 干什么的，怎么调用，这个命令怎么用的，知道性的知识，google 一下或者认真看下文档就应该知道。
- 问一些特别复杂的问题比如问一个特别复杂的算法，问一个很抽象的大问题，短时间内很难给予回答。
- 问一些假设性的问题假设你参与了这个项目，你觉得哪几个地方需要优化。

之所以说这些问题不应该问，我认为主要是因为这些很难考察到面试者的真实能力，45 分钟的时间本来就很短，有些问题有可能比较偏，有些问题又过于庞大没法一下子描述特别清楚，还有一些问题缺乏上下文，让人摸不到头脑，所以尽量避免这么问问题，另外把握一个重要原则，**不要在面试中试图证明别人不如自己**，毫无意义，人无完人，总有覆盖不到的地方，按照这个规则招聘，会错过很多优秀的人才。

面试应该做的事

- 问已经发生的事情

比如面试移动开发者，面试官应该认真看下其做过的 App，具体的工作是什么，准备一些相关的问题，这里就可以看出来之前工作中的积累是什么，有多深。

- 问题解决思路

针对项目经验和一些学习的经验上面，应该问拿到问题以后解决思路是什么，在什么场景下为什么这么做，这里根据面试者的方案，分析的方法论，就可以大致了解面试者是否聪明，知识面是不是够广，遇到问题时会不会举一反三。

具体可以举个简单的例子，很多同学说自己做过架构，然后都会讲自己做了一个解耦和分层的框架，其实这类框架 iOS 很多，外部 github 上就有各种方案。在阿里内部手淘早先做的 bundle 拆分时沉淀的容器规则，天猫开源出去的 beeHive，闲鱼内部的 Xframework，抑或是服务端的 spring mvc，其实都实现了 IoC，但实现和思路上都有一些差异，到底为什么这么做，其实是有区别的，这里面就可以看出知识广度，总结和思辩能力，在关键路径上的技术判断。

又比如说，我们总在强调性能稳定性怎么做，业界也有很多方案，到底哪个方案更好呢？答案没有绝对的对错，取决于某个时间点和场景下哪个问题是最核心的突破点，而你的选择标准和落地的技术方案是不是合理（考虑成本，收益，以及后续的风险是什么）。一般来讲，我们更倾向于用系统化的思维看待一个问题，也就是说，相比根据人的经验去识别性能瓶颈，我们更希望能通过自动化，智能化，数据化的方式去解决问题。

- 少问多听

一般刚开始做面试官的同学很喜欢以问为主，但因为大家的知识体系不太一样，成长环境也不同，直接这么问起来很难就找到面试者的优点，所以尽量让应试者自己陈述，然后以学习和交流的心态针对陈述中存疑的地方再进行发问，会更容易让应试者放松，也更容易让应试者更全面的表达自己。另外，问的差不多的时候，结尾的时候可以补充一句：您觉得刚才的面试中还有哪些我没问到的，您想再补充一下的内容？末了，再问下：我的问题问完了，您有什么想要问我的吗？

知道了应该怎么做，那具体的提问方法有没有什么技巧呢？在招聘中有一个重要的 STAR 原则，可以跟大家分享。

STAR 原则

- 处境 (situation)

在什么样的环境下

- 任务 (task)

接到了什么样的任务

- 行动 (action)

然后具体怎么落地的

- 结果 (result)

拿到了什么结果

我们尽量问清楚对方在什么样的环境下接到这个任务，接到以后是做了什么事情，最后的结果是什么样子的。乍一听，感觉，这不是套路嘛，是不是知道这个原则的人，只要按照这四点编故事，就能通过面试了？当然不是，在叙述过程中，我们应该分辨出 STAR 中的真假，那下面就举一些例子。

假的 STAR

- 描述含糊不清

比如，我用这个方案解决了这个问题，效果很好，得到了大家的一致好评。注意，效果好是哪里好，有什么度量的标准？一致好评的体现是在具体 KPI 还是比如团队有个什么奖励之类的。

- 只表达态度和看法

我觉得线上稳定性非常重要，应该重点解决和持续跟进。如果只有这一句话，没有后面具体认为重要的解决方案的话，这部分的经验难以令人信服。

- 假设式描述

如果我来做这件事情，我会 1234 怎么怎么样。前面其实面试应该问的问题里面有提到，我们本身就不应该问假设性的问题，那作为面试者，假设没做过的事情，如

果只是看思路还好，但是如果说的天花乱坠，这个时候要警惕了，毕竟说和做之前的差异是很大的。对于假设的事情，面试官是没法评估具体效果的，因为它不像过去已有的项目和工作内容，是有明显结果的，如果对过去结果存疑，后续也可以背调了解具体的情况。

针对假的 STAR，我们要甄别分辨出来，引导其表达出真正的情况。

鉴别方式

- 更多的关心 What/How/Why

做了什么事情，具体做的方案 1234 几步，为什么要这么做，比如图片的优化，最早肯定什么都没有，后续加 cache，cache 策略又可以升级，包括 cache 本身的算法以及多级 cache 的实现，图片尺寸上面后来有做了什么裁切之类的，图片格式上面后续又做了优化等等。

每个阶段不太一样，关注的重点也不一样，刨根问题问一问，会了解是不是真的做过这件事情，另外有一些可能项目做得很久说很多东西忘了，这里我分享一个观点，之前看过一句话，招聘的人中有一种人是比较好的，**他总能比较清楚的记住过往**

项目当中的重点，这样的人在经验沉淀的过程中肯定更快一些，当然这样的同学肯定得归结在聪明一类的人了，当然能记住也说明他可能喜欢总结和回顾，平常的学习习惯应该也比较好。

- 细节！细节！细节！

很多关键节点的细节很重要，比如网络库的优化。如果你是一个 iOS 开发，一般都会知道 iOS 的网络协议优化常采用拦截 NSURLProtocol 的方式进行，然后针对传统的 https 协议我们会将其替换成为 spdy 协议或者 http2 协议，过程中还有一些 httpdns 等的优化。但如果你今天希望招聘一个有这部分网络优化经验的同学，怎么判断这个同学有实操的经验呢？你可以让他说细节上面的很多事情，比如说 URLProtocol 拦截 request 以后，针对不同的 case 的降级策略是什么，选择依据是什么？当时遇到了什么其他的坑没有？你自己的做法有什么缺陷？

又比如 Weex 的实现上面，整个渲染的流程到底是怎样的，渲染部分还有什么优化空间吗？或者说这个方案本身做了哪方面的优化？它的配套工程体系上的问题是什么，你遇到以后是怎么解决掉的，这些在了解大概思路后，都可以往深入再问一下细节的部分，认真研读以及修改过代码的同学，肯定是答得出来的。

其他 Tips

- 你在面试别人，别人也在选择你

面试是双向的，面试官是一个团队对外的门面，不要迟到，提问和交流要尊重面试者，让面试者感受到我们的真诚。

- 为未来招聘而不是现在

我们永远应该为未来招聘，因为招聘的人入职也是发生在未来，不能立刻就解决你眼前的问题，所以我们招聘的时候也放长远一点，招为团队未来更有好处的人。

- 面试是一面镜子

以人为镜，优秀的面试者给我们能带来新的思路和新的方法，而差强人意的面试者则提高我们看人的能力，为后续的提升招聘效率找到更正确的人打下基础。

技术人如何不断成长?

招聘，培训，人才选拔晋升，我认为评价标准和方法都应该有比较多的重合的部分，我们从刚才的面试经验中，反思下，如果现在是我们去找工作，这个市场或者团队更需要什么样的人?

- 经验丰富，知识体系完整

经验能解决实际的问题，另外知识体系可以让你在遇到新的问题时举一反三，当然大公司和小公司要求的知识体系又不太一样，大公司更偏向一专多能的 T 型人才，小公司更喜欢全栈，所以到底要成为什么样的人，跟你的职业规划很有关系，是想在大公司成就一番事业，还是出去闯荡，那你点的技能树肯定是不一样的。到底应该怎么做，我自己的经验是，找到身边的标杆，向更优秀的同学学习，在阿里当然非常优秀的专业人才也好，架构师也好，都非常多，所以标杆应该也好找，业界当然也有很多成功的人，有了标杆，就努力向上吧。

- 保持良好的习惯，不忘总结和提升

当我还是一个菜鸟的时候，当时的老板问了我一个问题，每周写周报的时候，想想自己这一周到底收获了什么，这给我留下了很深的印象。我在想，既然我每次面试别人都问你最近有研究什么新的技术或者看到什么有趣的文章没有的，那我自己是不是能这样要求自己呢？不积跬步无以至千里，贵在坚持积累。

使用开源项目的正确姿势，血和泪的总结

李运华

阿里妹导读：开源精神是技术发展的源动力之一，受到工程师们的热烈欢迎。但是开源项目如此之多，哪一个最适合自己？如何更好利用开源项目，甚至做二次开发？ 今天，阿里资深无线开发专家李运华，总结多年与开源项目打交道的经验，讲述如何正确利用开源项目，希望对大家有所启发。

软件开发领域有一个流行的原则：DRY，Don't repeat yourself，我们翻译过来更形象通俗：不要重复造轮子。开源项目主要目的是共享，其实就是为了让大家不要重复造轮子，尤其是在互联网这样一个快速发展的领域，速度就是生命，引入开源项目，可以节省大量的人力和时间，大大加快业务的发展速度，何乐而不为呢？

然而现实往往没有那么美好，开源项目虽然节省了大量的人力和时间，但带来的问题也不少，相信绝大部分同学都踩过开源软件的坑，小的影响可能是宕机半小时，大的问题可能是丢失几十万数据，甚至灾难性的事故是全部数据都丢失。

除此以外，虽然 DRY 原则摆在那里，但实际上开源项目反而是最不遵守 DRY 原则的，重复的轮子好多，尤其是歪果仁，一看哪个开源方案不爽，自己就吭哧

吭哧搞一个差不多的：你有 MySQL，我有 PostgreSQL；你有 MongoDB，我有 Cassandra；你有 memcached，我有 redis；你有 Gson，我有 Jackson；你有 Angular，我有 React。总之放眼望去，其实相似的轮子很多！相似轮子太多，选择就是让人头疼的问题了。

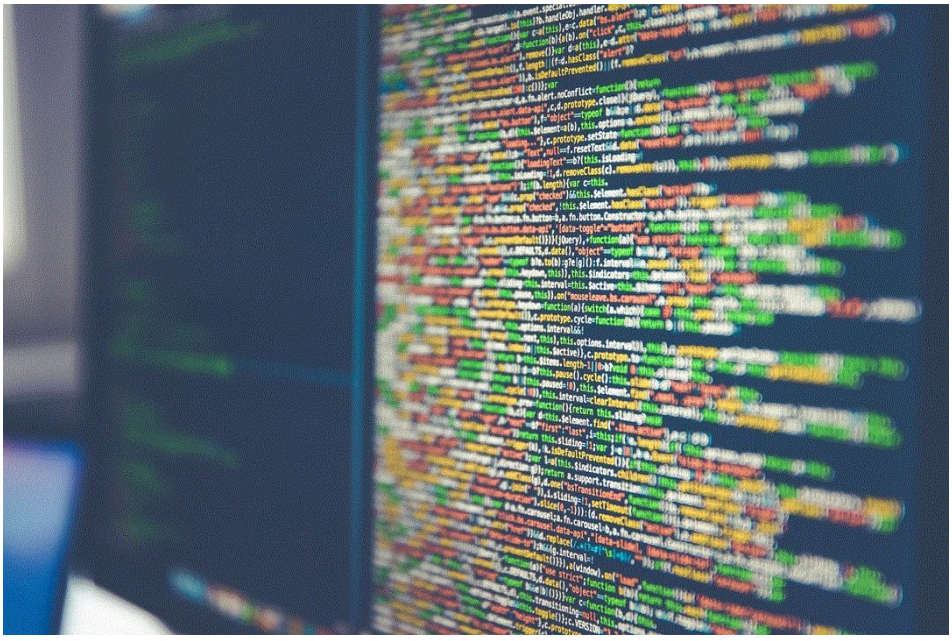
怎么办？完全不用开源项目几乎是不可能的，我们需要更加聪明的去选择和使用开源项目。形象点说：不要重复发明轮子，但要找到合适的轮子！你开的是保时捷，可别找个拖拉机的轮子。

接下来我将根据加入 UC，5 年与开源项目有关的经历，总结出一些“如何正确使用开源项目”的经验和教训。有的项目是我亲身经历，有的是我接触到的，有的是我观察的，其中部分描述细节可能并不完全准确，大家可以结合自己的经历一起探讨。

以下内容主要分 3 个部分进行描述，分别是“选”、“用”、“改”。

选：如何选择一个开源项目？

聚焦是否满足业务？



我们在选择开源项目的时候，一个头疼的问题就是相似的开源方案较多，而且后面的总是要宣称比前面的更加牛逼。我们在选择的时候有点无所适从，总是会担心选择了 A 方案而错过了 B 方案，或者反过来。这里我们的经验是聚焦于是否满足业务，而不需要过于关注开源方案是否牛逼。

案例：当时尝试一个社交类业务时，我们发现了 TT（Tokyo Tyrant）这个开源方案，觉得既能够做缓存取代 Memcached，又有持久化存储功能，可以取代 MySQL，很牛逼，很高大上，于是就在业务里面大量使用了。但后来的使用过程让人很蛋疼，主要表现为：

1. 不能完全取代 MySQL，因此有两份存储，设计的时候每次都要讨论和决策
2. 功能上看起来很高大上，但相应的 bug 也不少，而且有的 bug 是致命的，例如所有数据不可读，后来是自己研究源码写了一个工具才恢复了部分数据。
3. 功能确实牛逼，但需要花费较长时间熟悉各种细节

后来我们反思和总结，其实当时的业务 Memcached + MySQL 完全能够满足，且大家都熟悉，当时的业务完全不需要引入 TT。

简单来说：如果你的业务要求 1000 TPS，那么一个 20000 TPS 和 50000 TPS 的方案是没有区别的。有的人可能会担心我 TPS 不断上涨怎么办？其实不用担心，我们的架构会不断演进的，等到真的需要这么高的时候我们再来架构重构，记住：不要过早优化，过早优化是万恶之源 ——《UNIX 编程哲学》

聚焦是否成熟

很多新的开源项目往往都会声称自己比以前的项目更加牛逼：性能更高、功能更强、引入更多新概念。看起来都很诱人，但实际上都有意无意的隐藏了一个负面的问题：都更加不成熟！不管多牛逼的程序员写出来的项目都会有 bug，千万不要以为作者牛逼就没有 bug，Windows、Linux、MySQL 的开发者都是顶级的开发者吧，一样很多 bug。

不成熟的开源项目应用到生产环境，风险极大。轻则宕机，重则宕机后重启都恢复不了，更严重的是数据丢失都找不回了。还是以上面提到的 TT 为例：我们真的遇

到异常断电后，文件被损坏，重启也恢复不了的故障，还好当时每天做了备份，于是只能用 1 天前的数据进行恢复，但当天的数据全部丢失了。后来我们花费了大量的时间和人力去看源码，自己写工具恢复了部分数据，还好这些数据不是金融相关的数据，丢失一部分问题也不大，否则就有大麻烦了。

所以在选择开源项目的时候，尽量选择成熟的开源项目，降低风险。

可以从以下几个方面考察是否成熟：

1）版本号：一般建议除非特殊情况，否则不要选 0.X 版本的，至少选 1.X 版本的，版本号越高越好。

2）使用的公司数量：一般开源项目都会把采用了自己项目的公司列在主页上，公司越大越好，数量越多越好

3）社区活跃度：看看社区是否活跃，发帖数、回复数、问题处理速度等

聚焦运维能力

我们在选择开源项目的时候，基本上都是聚焦于技术指标，例如性能、可靠性、功能这些方案，而几乎不会去关注运维方面的能力。但如果要将方案应用到线上生产环境，运维能力是必不可少的一环，否则一旦出问题，运维、研发、测试都只能干瞪眼，求菩萨保佑了！

可以从以下几个方案去考察运维能力：

1）开源方案日志是否齐全：有的开源方案日志只有寥寥启动停止几行，出了问题根本无法排查

2）开源方案是否有命令行、管理控制台等维护工具，能够看到系统运行时的情况

3）开源方案是否有故障检测和恢复的能力，例如告警、倒换等

用：如何使用开源方案？

深入研究，仔细测试

很多人用开源项目，其实是完完全全的“拿来主义”，看了几个 Demo，把程序跑起来就开始部署到线上应用了。就好像看了一下开车指南，知道了方向盘是转向、油门是加速、刹车是减速，然后就开车上路了，其实是非常危险的。

案例：我们有团队使用了 elasticsearch，基本上是拿来就用，倒排索引是什么不太清楚，配置都是用默认值，跑起来就上线了，结果就遇到节点 ping 时间太长，剔除异常节点太慢，导致整站访问挂掉。

案例 2：很多团队最初使用 MySQL 的时候，也没有怎么研究过，经常有业务部门抱怨 MySQL 太慢了，其实经过定位，发现最关键的几个参数（例如 innodb_buffer_pool_size, sync_binlog，innodb_log_file_size 等）都没有配置或者配置错误，性能当然会慢。

可以从如下几方面进行研究和测试：

1）通读开源项目的设计文档或者白皮书，了解其设计原理

2）核对每个配置项的作用和影响，识别出关键配置项

3）进行多种场景的性能测试

4）进行压力测试，连续跑几天，观察 cpu、内存、磁盘 io 等指标波动

5）进行故障测试：kill，断电、拔网线、重启 100 次以上、倒换等

小心应用，灰度发布

假如我们做了上面的“深入研究、仔细测试”，发现没什么问题，是否就可以放心大胆的应用到线上了呢？别高兴太早，即使你的研究再深入，测试再仔细，也还是要小心为妙，因为再怎么深入的研究，再怎么仔细的测试，都只能降低风险，但不可能完全覆盖所有线上场景。

案例：还是以 TT 为例吧，其实我们在应用之前专门安排一个大牛看源码、做测试，做了大约 1 个月，但最后上线还是遇到各种问题。线上生产环境的复杂度，真的不是测试能够覆盖的，必须小心谨慎。

所以，不管研究多深入、测试多仔细、自信心多爆棚，时刻对线上要有敬畏之心，小心驶的万年船。我们的经验就是先在非核心的业务上用，然后有经验后慢慢扩展。

做好应急，以防万一

即使我们前面的工作做得非常完善和充分，也不能认为就万事大吉了，尤其是刚开始使用一个开源项目，运气不好的话就可能遇到一个之前全世界的使用者从来没遇到的 bug，导致业务都无法恢复，尤其是存储方面，一旦出现问题无法恢复可能就是致命的打击。

案例（此案例是听说的）：某个业务使用了 MongoDB，结果宕机后部分数据丢失，无法恢复，也没有其它备份，人工恢复都没办法，只能接一个用户投诉处理一个，导致 DBA 和运维从此以后都反对我们用 MongoDB，即使是尝试性的。

虽然因为一次故障就完全反对尝试是有点反应过度了，但确实故障也给我们提了一个醒：对于重要的业务或者数据，使用开源项目时，最好有另外一个比较成熟的方案做备份，尤其是数据存储。例如：如果要用 MongoDB 或者 Redis，可以用 MySQL 做备份存储。这样做虽然复杂度和成本高一些，但关键时刻能够救命！

改：如何基于开源项目做二次开发？

保持纯洁，加以包装

当我们发现开源项目有的地方不满足我们的需求的时候，自然会有一种去改改的冲动，但是怎么改是个大学问。一种方式是投入几个人从内到外全部改一遍，将其改造成完全符合我们业务需求。但这样做有几个比较严重的问题：

1）投入太大，一般来说，redis 这种级别的开源方案，真要自己改，至少要投入 2 个人，搞个 1 个月以上

2）失去了跟随原方案演进的能力：改的太多的话，即使原有开源项目继续演进，我们也无法合并了，因为差异太大。

所以我们的建议是不要改动原系统，而是要开发辅助系统：监控，报警，负载均衡，管理等。以 Redis 为例，如果我们想增加集群功能，不要去改动 Redis 本身的实现，而是增加一个 proxy 层来实现，Twitter 的 Twemproxy 就是这样做的，而 Redis 到了 3.0 后本身提供了集群功能，原有的方案简单切换到 Redis 3.0 即可。详细可参考 (http://www.cnblogs.com/gomysql/p /4413922.html)

如果实在想改到原有系统，怎么办呢？我们的建议是直接给开源项目提需求或者

bug，但弊端就是响应比较缓慢，这个就要看业务紧急程度了，如果实在太急那就只能自己改了，不过不是太急，建议做好备份或者应急手段即可。

发明你要的轮子

这点估计让很多人大跌眼镜，怎么讲了半天，最后又回到了“重复发明你要的轮子”呢?

其实选与不选开源项目，核心还是一个成本和收益的问题，并不是说选择开源项目就一定是最优的方案，最主要的问题是：没有完全适合你的轮子!

软件领域和硬件领域最大的不同就是软件领域没有绝对的工业标准，大家都很尽兴，想怎么玩怎么玩，不像硬件领域，你造一个尺寸与众不同的轮子，其它车都用不上，你的轮子工艺再高，质量再好也是白费；软件领域可以造很多相似的轮子，也基本上能到处用，例如你把缓存从 Memcached 换成 Redis，不会有太大的问题。

除此以外，开源项目为了能够大规模应用，考虑的是通用的处理方案，而不同的业务其实差异较大，通用方案并不一定完美适合具体的某个业务。比如说 Memcached，通过一致性 hash 提供集群功能，但是我们的一些业务，缓存如果有一台宕

机，整个业务可能就被拖慢了，这就要求我们提供缓存备份的功能，但 Memcached 又没有，而 Redis 当时又没有集群功能，于是我们投入 2~4 个人花了大约 2 个月时间基于 LevelDB 的原理，自己做了一套缓存框架支持存储、备份、集群的功能，后来又在这个框架的基础上增加了跨机房同步的功能，很大程度上提升了业务的可用性水平。如果完全采用开源方案，等开源方案来实现，是不可能这么快速的，甚至都有可能开源项目完全就不支持我们的需求。

所以，如果你有钱有人有时间，投入人力去重复发明完美符合自己业务特点的轮子也是很好的选择！毕竟，土豪们（BAT、Facebook、Google..... 等）很多都是这样做的，否则我们也就没有那么多好用的开源项目了：）

前端工程师的未来在哪里?

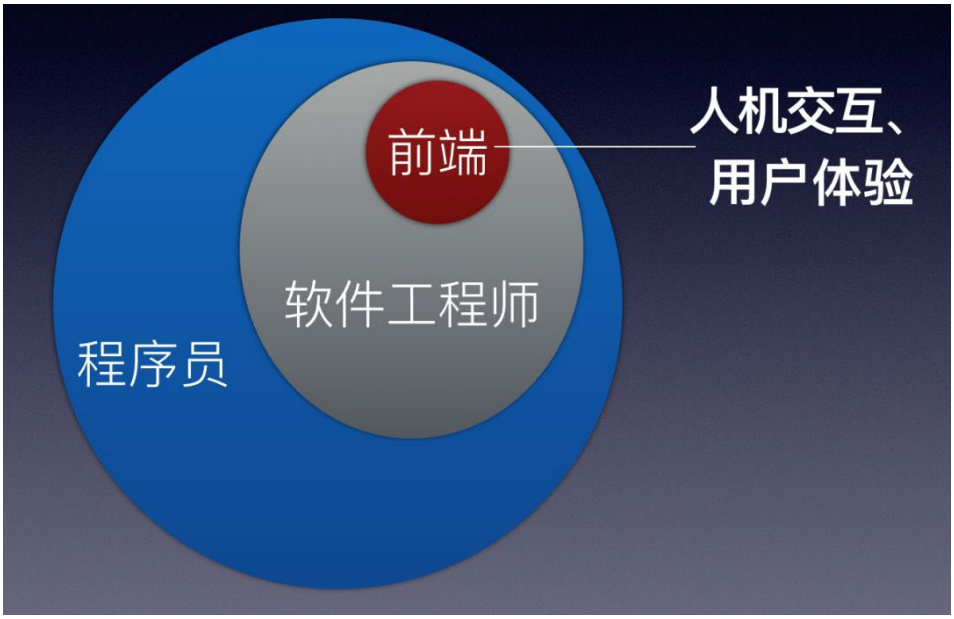
成曰

阿里妹导读：很多童鞋在上次的问卷调查里表示，希望多推出一些前端方向的内容。今天为大家分享一篇关于前端工程师职业发展的文章，抛砖引玉，期待与大家一起交流探讨。

我是成曰，目前在蚂蚁金服数据平台部任职前端工程师，从事数据中台产品的研发。目前对前端架构与质量、用户体验、TypeScript 比较感兴趣。

以下我说的都是个人观点，比较宏观粗浅，主要针对的是前端工程师本身，没有深入技术和业务细节，请谨慎参考。

职能概览



前端工程师首先是个程序员，其次也是个软件工程师，他们工作在离用户最近的地方，负责人机交互和用户体验，虽然叫“前端”，但其实他们的工作边界其实已经很宽了。

展望未来，我想前端的工作会继续分化，也会继续融合，分工是工业革命以来社会高效协作的主要推动力，以后很长一段时间应该也会维持这种形态，融合的原始推动力也是提高效率。分化和融合是不断的演化和互吸收转化的，不过核心的东西我想还是不会有太大变化。

观点

1. 继续分化（领域、行业、技术栈）
2. 继续融合（端技术、Web 全栈技术、人工智能与端技术）
3. 核心不变（计算机科学本质、软件工程思想与实践、程序员职业素养）

1. 继续分化

领域

前端领域会继续分化，例如阿里内部的前端就已经有中后台、图形、端技术、泛Node、开发者服务 5 个大方向了，每个大方向也会细分，举一些例子：

- 中后台：有云控制台、信息 & 资产管理平台、内部研发 & 项目管理平台、人工智能 & 机器学习平台、数据研发分析平台，企业内部信息平台等。具体产品如阿里云控制台、ERP、PAI、DeepInsight、阿里内外、Basecamp 等。
- 图形：有基础图形库、3D 图形、数据可视化、流程图等。具体产品如 G2、DataV、阿里云城市大脑、滴滴智能交通调度图、双十一大屏等。
- 端技术：有移动端（iOS、Android、MobileWeb、PWA、小程序）、PC 端（客户端、Web 端）、触屏电脑、各种监控大屏、智能手表手环，智能汽车 &

家居屏幕等。具体产品如淘宝支付宝的 App、PC 主站、移动 H5 站，阿里郎、VS Code、双十一大屏、UC 浏览器 UWP 版本、各种智能手表、手环、汽车、家居屏幕等。

- 泛 Node：有工具链、Web 框架、IoT、客户端（Electron、NW）等。具体产品如 DEF/Atool/F2E-Test（阿里前端开发者工具）、Egg.js、阿里云的 IoT 应用、VS Code 等。
- 开发者服务：有应用开发运维平台、组件市场等。具体产品如阿里云的应用搭建平台 Boat、Fusion-Design 组件市场，支付宝小程序开发者工具等。

行业

- 2B
  - 信息管理、财务、建筑、航天、水利、金融、制造等传统行业软件以及阿里提出的五新：新零售，新制造，新金融，新技术和新能源，新技术赋能传统行业
  - SAAS 软件及服务：如 Teambition、Trello、钉钉企业版、Basecamp、Growing.io
- 2C
  - 移动 App：如微信、微博
  - PC 工具应用：如 Google Doc
  - 产品展示类网站：如阿里云、支付宝官网

技术栈

- React（Native）
- Angular（NativeScript）
- Vue（Weex）

2. 继续融合

端技术

- 前端、客户端技术思想的融合

- 组件化（组件化搭建页面）
- 组件生命周期钩子函数（如 iOS ViewController）
- MV*（如 MVVM 设计就来源于微软客户端开发框架）

大前端的统一

- 虚拟 DOM 技术：React/ReactNative/ReactCanvas
- 各种移动设备内核和引擎的统一：WebKit/V8
- Web 技术文档的统一：Mozilla Web Docs

Web 全栈技术

- 前端、后端技术思想的融合
  - MV*（如前端的第一个 MVC 框架 Backbone.js 就来自于 Ruby on Rails 开发者）
  - AOP、依赖注入（Angular）
  - GraphQL（SQL）
  - IndexedDB（Database）

人工智能与端技术

- 人工智能、前端技术的融合
  - 端是最终触达用户的节点
  - 端数据采集 -> 后端机器学习、数据分析 -> 智能推荐呈现
- 物联网、前端技术的融合
  - 智能家居 / 汽车 / 工业设备可能是有屏幕的，同时可以基于如 JerryScript 这样的 JS 执行引擎使用 Node.js 开发联网应用

3. 核心不变

计算机科学本质 / 软件工程思想与实践 / 程序员职业素养。

计算机科学基础：如基本的操作系统概念和计算机组成原理，算法和数据结构基础等等。

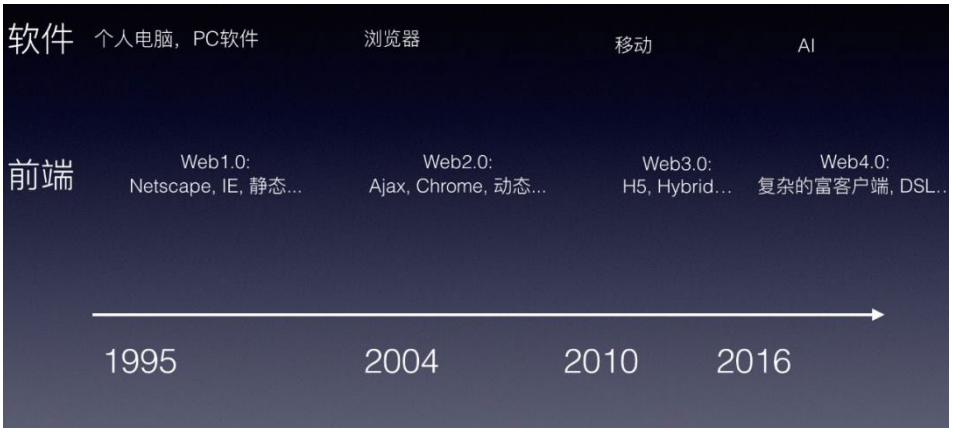
软件工程思想与实践：如软件开发模式，设计模式，架构思维，自动化思维，单

元测试集成测试，UML 等等。

程序员职业素养：如对代码整洁和可读性的追求，对软件开发的热情，对编程技艺的自我提升等等。

历史回顾

回顾过前端的演化，主要参见最底下的相关资源，下图简要回顾一下 95 年以来软件开发和前端历史：



前端的未来

那些生存空间越来越小的产业

- 小规模移动 App：移动 App 市场被一些巨头把持，小规模 App 生存空间越来越小
- PC 信息导航类网站（网址、购物、论坛、生活）：现在移动优先，而且有智能推荐，并且是强社交

无界面交互

- Web 前端能做一些事，主要是大前端的范畴
- 会话式界面（视频语音会话、语音搜索：WebRTC，开源语音库：Common Voice）
- 感官式界面（视觉：WebAR/WebVR）

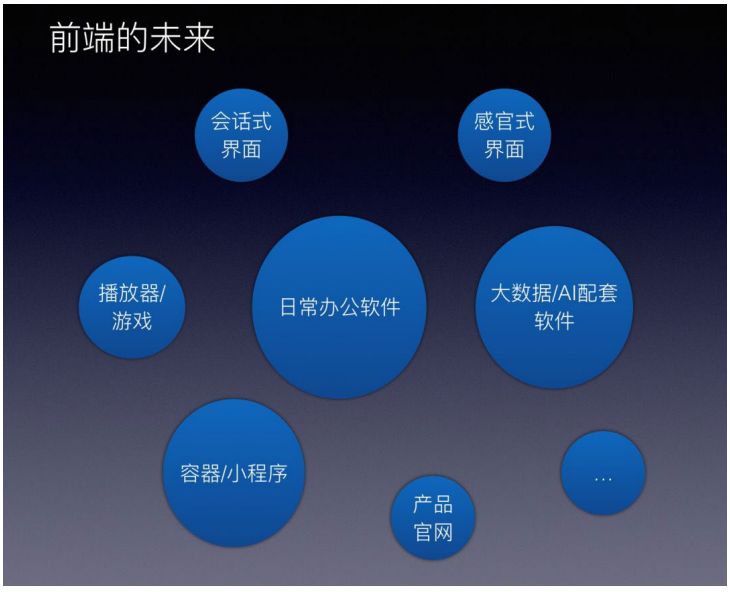
- 无界面，未来会有大量 IoT 设备（IoT 有自己的通信协议和规范，前端的工作是在用户看不见的后台）

富界面交互

大量工作需要前端来做：

- 日常办公软件（复杂应用，并且慢慢从桌面程序演化为 Web）：Office、Email、文档管理、产品设计、项目管理、代码编辑器
- 大数据 /AI 配套软件（复杂应用，并且慢慢从桌面程序演化为 Web）：需要大量的后台系统来做数据分析 / 机器学习
- 播放器 / 游戏：H5 代替 Flash，如 Web Audio、Web Video、Canvas
- 容器 /DSL/ 内核 / 小程序：支付宝 / 微信 / 钉钉容器，内核（也就是 UC、QQ 浏览器内核），及其自定义 DSL（如果将来手机的底层能力都可以上浮到小程序，很有想象空间）
- 产品信息展示类网站（炫酷应用）：各种智能设备官网、大企业官网

未来在哪里？



回顾观点

1. 继续分化（领域细分、行业细分、技术栈细分）

2. 继续融合（端技术融合、Web 全栈、人工智能与端技术的融合）

3. 核心不变（计算机科学本质、软件工程思想与实践、程序员职业素养）

一些建议

1. 关心人工智能的发展，思考 TA 在前端领域可能产生的应用场景

- 视觉稿自动生成代码
- 根据用户使用习惯自动排出最符合该用户习惯的界面
- 收集用户数据在前端实时做学习和分析，如 deeplearn.js

2. 相信前端的未来，Web 的力量

- WebKit
- V8
- Flexbox：Yoga

3. 结合公司业务特点有重点的关注前端的某些方面，毕竟技术服务于业务

后记：前端的发展超出了所有人的想象力，未来肯定是难以预测的，也没有做预测的必要，我们要做的还是踏实做好眼前的事情，“过往不恋、当下不杂、未来不迎”，与君共勉！

相关资源

- Web 开发这十年：http://www.infoq.com/cn/articles/web-development-ten-years
- GUI 应用架构十年变迁：https://segmentfault.com/a/1190000006016817
- 大话前端时代一：https://halfrost.com/vue_ios_modularization/
- 写给初学前端工程师的一封信：https://zhuanlan.zhihu.com/p/28536429
- 母鸡与前端工程师：http://www.ruanyifeng.com/blog/2016/07/hen-and-front-end-engineer.html
- 李开复人工智能预言：http://tech.sina.com.cn/it/2017-05-20/doc-ify-fkqks4361454.shtml
- 《无界面交互》：https://book.douban.com/subject/26947799/

前端 Leader 如何做好团队规划？阿里内部培训总结公开

剑平

阿里妹导读：作为一名前端团队的管理者，如何做好团队规划？老板不是前端，如何做出被认可的成绩？今天，阿里前端技术专家剑平，将结合自己的亲身经历，以及阿里内部培训课程，写下了自己的思考和理解，与大家共同分享。欢迎一起讨论交流。

前言

“行成于思，毁于随”——韩愈

在阿里从一线前端工程师到技术 TL（Team Leader）也三年有余了，最重要最难的就是做规划，你可能会遇到如下几个问题：

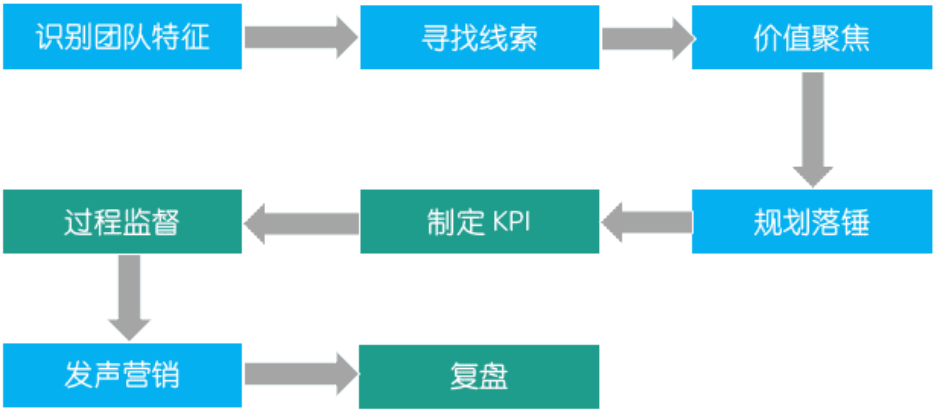
- 业务压力巨大，前端是瓶颈，如何做合适的规划？

- 如何提高规划的成功率?
- 规划的雷区是什么?
- 如何寻找规划的线索?
- 我的老板不是前端，如何做出被认可的成绩?

今年 4 月份参加了阿里集团前端委员会组织的 TL 培训（老师都是阿里的前端大牛），为期三天，收获颇丰，特别是关于前端 TL 如何做团队规划方面，整理课堂笔记时，就想结合自己这一年在阿里拍卖业务中的规划实践做下总结。

本文会引用课堂上多位阿里前端大牛老师的观点。

一般技术规划路径如下：



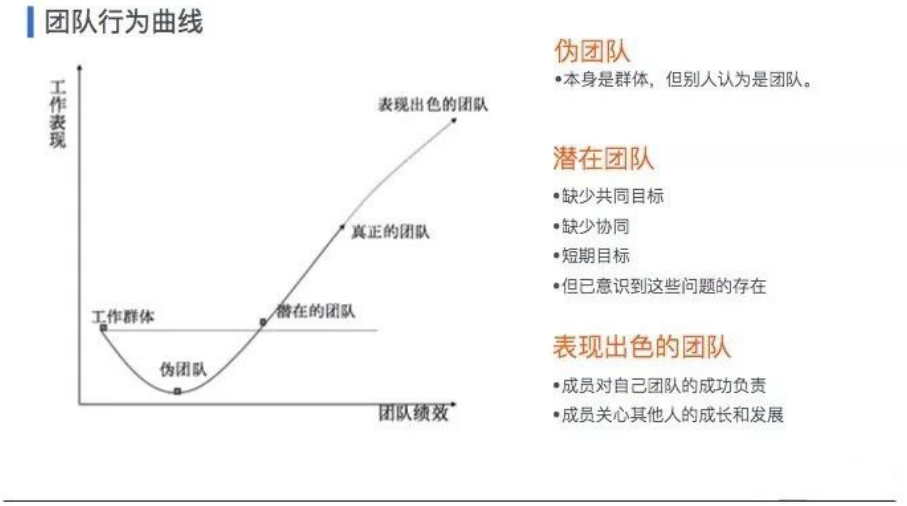
（蓝色部分为本文重点论述部分）

一、先从团队特征说起

做规划，先要定义清晰自己团队的特征与发展的阶段。因为特征不同与发展阶段不同，规划的选择就会截然不同。

1. 识别团队特征

在阿里梓骞老师关于建团队的论述中，有一块团队行为曲线的总结特别好，是团队比较核心的特征。



（上图来自梓骞老师 PPT）

我的团队特征（某财年阿里拍卖业务前端团队）：

- 背景：从淘宝技术部到垂直支撑业务
- 团队行为：处于潜在团队，共同目标不清晰、协同不够、缺乏长期目标
- 人员构成：6 人，平均层级偏低
- 业务发展情况：创业型，高速发展，局部瓶颈
- 业务重点关键字：DAU、用户体验（业务老大高频强调）
- 资源富裕度：负，人员严重不足，前端成为业务发展瓶颈
- TL 汇报对象：产品经理

当识别完团队特征，最迫切的事往往可以呼之欲出，比如招聘是第一等大事，“废话，给我找个不缺人的前端部门 ...”，莫急，本篇不讲招聘 ...

发现上个财年推进的事，印证了团队行为目标：努力将潜在团队引领成真正的团队。

如何做?

- 寻找团队共同目标
- 增强团队协同
- 明确清晰短期目标与长期目标

从团队特征中寻找规划的边界：

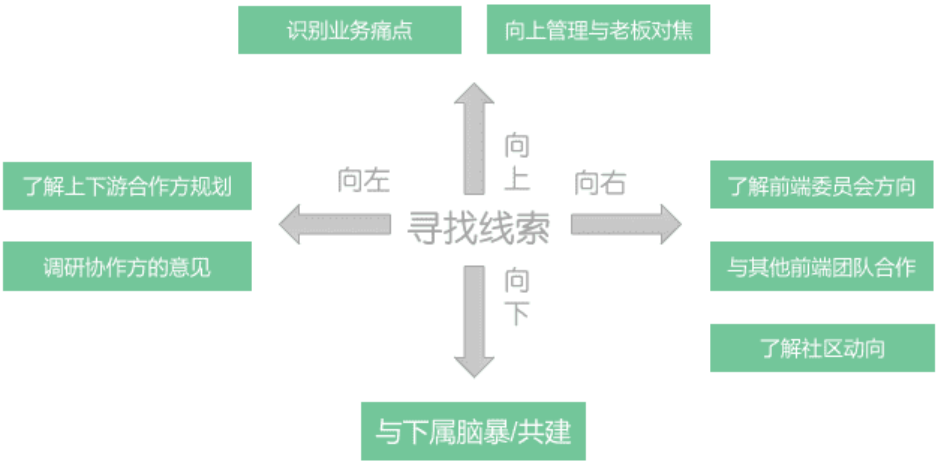
特征	要	不要
背景	紧贴业务，关联业务目标优先	无业务落地场景
团队行为	一条线索串起大家的工作	各自为战、埋头编码
人员构成	招聘、培养梯队	
业务发展情况	体系化思考、降低试错成本	单点，盲目
业务重点关键字	重点思考用户体验	向成本妥协
资源富裕度	招聘、效率	低水平重复劳动，高成本技术方案
TL 汇报对象	向上管理、多“科普”	靠猜

清晰的边界与基准、对规划内容优先级判断，对落地成功率有至关重要影响。

二、寻找规划线索

寻找规划线索是最耗时间的阶段，作为 TL 你有可能每天都在观察寻找规划的线索。

寻找线索，除了 TL 本身的思考外，需要具备外交能力，多问，如下图：



可以从四个方向去寻找链接，聊得越多，聊得越透，线索会浮现得越多。

下面针对去年比较有体感的点展开论述。

1. 向上管理

向上管理是拔赤老师比较强调的内容，如果你的老板不是前端，向上管理特别有必要，你需要消除“语言差”、做必要的前端核心概念“科普”。

一般业务 / 产品老板的关注点是：流量、转化、跳失、体量、用户体验、规模化、模式 / 产品创新等，要了解清楚现阶段老板的关注点是什么，从自己团队的维度思考试图给出到达路径，这是非常重要的规划线索。

向上管理不是有事没事找老板唠嗑，而是注意沟通的有效与质量，提问题最好带着初步的解决方案，业务 / 产品老板的时间有限，又存在“语言差”，相对复杂的内容务必准备 PPT。

2. 脑暴共创

脑暴共创是非常好的，自下而上的输入方式，而且你可以观察出下属的关注点，为后面寻找规划项目执行者提供线索。

你可以挑个风和日丽的周末，一整天与团队同学们关在一个咖啡馆或风景不错的会议厅。

因为共创会的成本相对较高，所以需要注意必须是主题式的共创，可以安排上午人员做主题式汇报，下午讨论聚焦。

另外还需要一个控场能力比较强的主持人，防止主题失焦，控制时间，共创需要的是高密度的信息。

除了与下属的脑暴共创，也可以组织与协作方的共创，参加业务方的共创。

3. 关注业务痛点

从业务痛点出发寻找的线索，确定规划后一般有充足的时间落地，是很靠谱的线索，但要注意抽象与提取，业务痛点更多是单点的现象，而我们需要做普适性的思考，这样才能充分发挥技术的价值，当然也要避免“过度设计”。

三、价值聚焦

1. 前端规划四问

1. 要解决什么核心业务问题?

2. 创造什么核心价值?

3. 为什么要做这件事？为什么是我们做?

4. 是单点，还是相对通用?

5. 以什么样的模式和方式来解决问题或创造价值?

6. 业务边界，系统边界如何取舍?

7. 问题在集团的大图位置和现状是什么样的?

8. 优势? 劣势?

9. 终局思考和实现路径是什么样的?

结合前端，我个人认为有核心四问：

1. 做成了会如何?（核心价值、终局构想）

2. 是我团队最重要的事吗?

3. 有没有更简单的方案?

4. 与业务的链接是什么?

终局构想特别重要，决定了规划内容的价值天花板，圆心老师给我们讲了案例，Pandora.js 做规划时，就要求是开源的，要服务外部应用，这个定位改变了 Pandora.js 从 0 到 1 的结构，如果一开始只是定位于为集团 node 应用服务，那么日后开源必然面临大量改造成本。

在前端团队多年，体感比较深刻有二点，前端容易将问题复杂化，喜欢用大的平台去解决小的问题，前端容易为了技术而技术，与业务“失联”。

2. 来自大牛灵魂的拷问

- 要让评审者学到点什么
- 在前端技术的横向影响
- 对业务中其他角色或业务的影响
- 对未来的判断
- 是否是重复造轮子
- 是否是“技术投机”，缺乏业务场景适用性思考

解决问题与痛点，远比构思复杂技术方案更重要，避免无差异重复造轮子或“技术投机”，着重预判规划落地后的影响力与价值论证。

3. 规划推导

规划推导分正推：从线索 -> 本质痛点或问题 -> 解决方案 -> 目标，反推：从目标 -> 解决方案 -> 本质痛点或问题 -> 线索。

二个过程都需要，你需要通过反推，去论证推导与路径的正确性，比如你的目标是提高 50 张页面 50% 的性能，推到解决方案，发现只能提高 10 张页面 10 % 的性能，就会发现解决方案是不靠谱的，需要再思考。

所有的线索都是现象，你需要去剖析现象背后本质，思考：

- 什么问题导致了出现这些现象?
- 痛点够不够痛?

解决方案的设计要思考：

- 调研是否充分，集团是否有现成方案?
- 是自己做，还是引进？还是引进后二次定制?
- 预计投入资源，投入产出比如何?

目标的设计要思考：

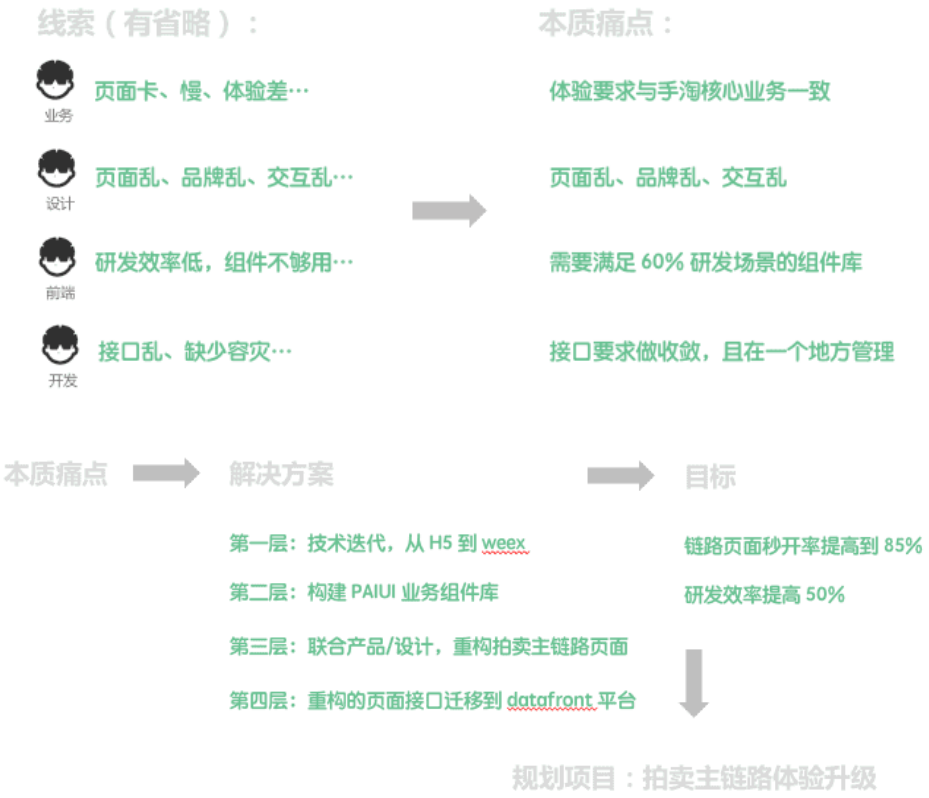
- 能够体现规划价值
- 可量化，可衡量，有影响
- 可达到

一般写规划时候，我们是先写目标，再写解决方案，而在实际推导规划时，一般是有个大概的解决方案，然后预判可能达到的目标。

举个简单的推导例子（过程做了简化）：

线索大部分单点的，如果依照单一线索，做出来的规划也是单一价值，就会发现价值不够大，比如上图，如果只是为了解决设计师的痛点，只做品牌的优化，就会发现价值点很小，而只是解决前端研发效率的问题，又如何深化前端的价值。

所以最后决定打包在一块，项目名是业务有体感的体验升级，目标是业务有体感的跳失率，将前端技术体系的建设包在了里面，同时也解决其他协作方的痛点。



4. 控制力

如果一件事件你也能做，别人也能做，且比你做的好，那么就要想想要不要做这件事。

控制力还体现在边界梳理，能跟其他岗位或其他前端团队合作是非常好的，但一定要理清楚边界，权责清晰才能有效促进规划成功。

四、规划落锤

价值聚焦完后，已经识别出最有价值的事，但未必是团队当下最重要的事，所以有规划落锤阶段，有如下几件事：

- 规划内容按照重要性排序
- 产出规划 PPT
- 产出关键里程牌时间点
- 排兵布阵、资源调度

李牧老师说，“TL 的核心素质是判断力与前瞻性”，这个阶段就很考验着二个能力。

1. 长期规划

价值聚焦完发现可做的事很多，如何办呢？这是好事，就可以试图做下长期规划，可以是三年规划、二年规划，并不是说非得一年建设完所有体系，罗马不是一天建成的，画张三年大图，给自己以指引。

明确团队技术体系的演进方向，穷尽所有高价值的事，每个季度复盘调整这张大图，让团队有共同的目标。

拔赤老师建议“技术规划以一年为最小单位，每季度做详细复盘，跟的勤，就不怕跟丢”。

2. 勇气与吸引力法则

有时你认为最有价值也是团队最重要的事会受到其他人的挑战，比如有人坚持认为现在资源紧张，不应该额外投入资源去做这件事。

这时就是很考验人的勇气的时候，选择接受，那么这件事就从你的规划移除，人员轻松了，产品满意了，但技术体系、体验没发生变化；选择坚持，那么人员工作强度变很大，协作方说不定会投诉。

你肯定也有面临这种选择的时候，无关对错，但我们需要有勇气面对挑战，做对的事，不要怂。

吸引力法则（你关注什么，就会将什么吸引进你的生活）告诉我们，有勇气去要求，笃定你的判断，有策略的执行，周围自然会发生你所希望的变化。所以关注于对的事，别被困难吓倒。

还有个策略，跟你的老板或上游来个“对赌”吧？

一定要发声营销

技术团队的 PR 意识相对淡薄，而前端团队在业务的影响力又相对较弱，所以特别需要 PR。

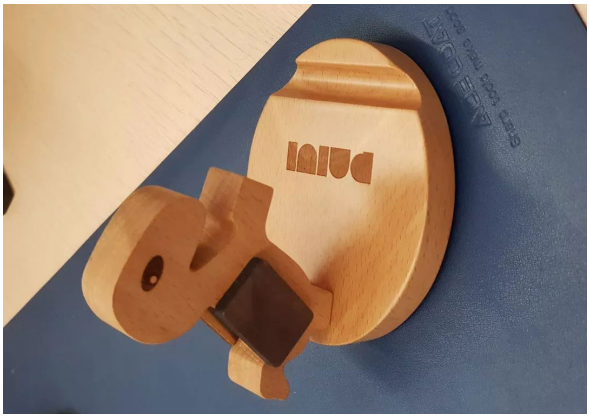
发战报是最有效的营销方式，但需要注意个注意点，如果战报对象是业务方、协作方，不懂前端技术，战报需要包装，让他们能看的懂。

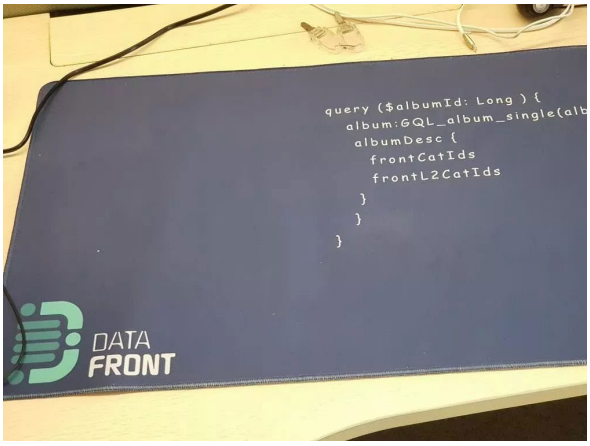
别写做了什么技术方案之类，别人看不懂，也没时间看。

技术产品宣传：

技术产品宣传分社区与公司内，办法很多，发放宣传礼品是个讨喜的办法。

去年自费做的二个宣传品：





一位优秀前端的自我修养

寒泉

今天给大家分享的主题是前端的自我成长，这是一个关于成长的话题。

很多人都有这样的感觉：听了很多技术圈子的分享，有的有深度，有的循循善诱，深入浅出，但是呢，几年下来，到底哪些用上了，哪些对自己真的有帮助了？反而有些模糊。

2015 年我在不同的场合分享了很多内容：有移动端的性能、有适配、有 Web vs Native，也有 hybrid，但是其实我一直比较担心，真正有深度的内容，其实面向的是比较小众的群体，比如说 Hybrid，其实它在大部分公司里面，是只能用现成的。

所以我这一次尝试分享一个我认为可以帮助到所有前端的话题，关于前端的成长，如果说这个分享的内容，听众里面有那么几十个人拿到 BAT 的 offer，或者升职加薪，那么我觉得我就认为我取得了成功。

前端其实是个特别苦逼的职业，因为前端技术一直革命的特别快，新技术、新技巧在不断地被发明出来。之前我有一个朋友，他讲说他对自己的认知是了解前端、熟悉前端、精通前端、熟悉前端、不懂前端。为什么呢，他说当他觉得自己对前端所有的东西觉得无所不知，无所不能的时候，忽然看到了一段代码，他完全无法理解，于是整个世界就崩塌了，从此再也不敢说自己会前端。

我就跟他说，这里，缺少的是一种正确的方法，你觉得无所不知、无所不能的标准是什么，是工作中很久没遇到解决不了的问题么？他说还真是这样。我就又问他，那你系统学过前端么？他想了想，还真没学过，大学里不开这个课。的确如此，到目前为止，还没有任何一个大学会教前端，倒是有些培训班，会讲网页开发三剑客。

我这里讲的内容，希望带给大家的，就是该如何学习前端，实现自身成长。

不包治✚前端

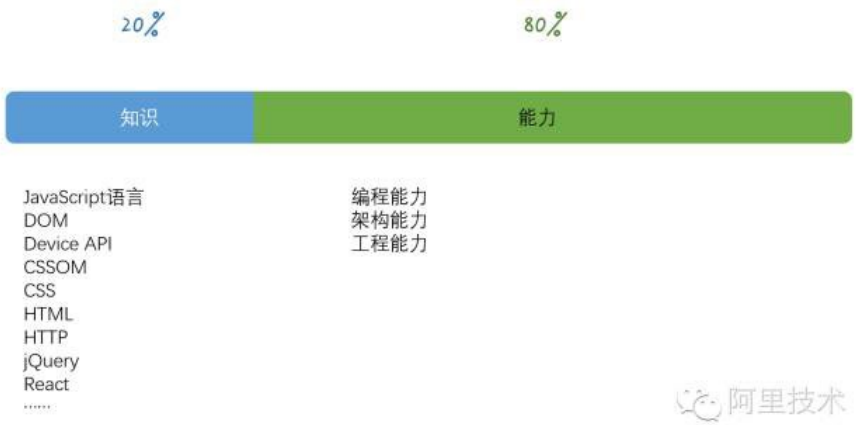
阿里技术

关于成长，首先我得发一个免责声明，不是我对我讲的内容没有信心，而是成长是自己的事，英文有句话，在外企工作的人会经常听到，叫做：

You are the owner of your career.

你是你职业发展的责任人。这句话潜台词是，你（不是你老板，也不是你爸妈，也不是你女朋友）是你职业发展的责任人。

这句话我在我职业生涯的起点听说，一直指导我的职业发展，甚至在我带团队，培养团队的时候，也是中心的指导思想，之前我带的团队的同学，他们有不少人也在带团队，其实他们也在实践这句话，所以我这里，也把这句话、把这个道理分享给给大家。



我们讲前端成长，**我认为，主要在两个方面，一部分是“能力”，一部分是“知识”。**我个人的观点，能力占百分之八十，知识占百分之二十。

从这个图上，大家可以看到，其实我们认为变化快的东西，最新出来的 Angu-

lar、React、ES2015，其实都在知识里面，知识又分成两部分，一部分我把它叫做标准，它是相对而言比较稳定的，很少会出现一个标准被推翻的事情。另一部分则是技术，像是 jQ、React 这些框架啦，像是 MVC、FLUX 这些架构的东西，这些东西是由各个公司主导的，变化就非常快，你看 Grunt 发展了没多久，Gulp 就来挑战他了，然后又有 browserify、webpack 这些东西。

而我认为占重点的能力，则是非常稳定的，我认为能力是三大块：编程能力、架构能力、工程能力。

编程能力，就是用代码解决问题的能力，你编程能力越强，就能解决越复杂的问题，细分又有调试、算法、数据结构、OS 原理等这些的支撑，你才能解决各种麻烦的问题。

架构能力，则是解决代码规模的问题，当一个系统足够复杂，你会写每一块，能解决每一个问题，不等于你能搞定整个系统，这就需要架构能力，架构能力包含了一些意识，比如解耦、接口隔离，也包含认识业务建立抽象模型，也有一些常见的模式，比如经典的 MVC，还有设计层面，面向对象、设计模式等等。

最后工程能力，则是解决协作的问题，当系统规模更大，光靠一个人，是没办法完成的，如何保证几个高手互相能够配合好？如何保证项目里面水平最差的人不拖后腿？这个工程化建设，往往会跨越多个业务，以汇报关系上的团队为单位来做。包括前后端解耦，模块化，质量保证，代码风格，等等。

其实不难看出来，这三项，其实是有顺序的，低等级、小团队，编程能力一项就能应付，越资深的前端，越大的公司和团队，越是需要后面的技能，但是这里我要强调一点，其实资深前端，大团队，对能力的需求，是既要还要——不是说资深的前端，编程能力就可以变差。

社区总会有一些声音，对工程能力，对架构能力持有一种抵触的态度，觉得比较虚，觉得不需要。实际上以某些人所在的岗位来说，也没错，毕竟公司、团队的状态确实可能用不到，但是以个人成长的角度来看，就是大错特错。

下面我们来具体讲讲，关于知识的学习。

对知识，我一直有个观点，叫做宁缺毋滥，这个图片上写了一句好前端才分对错，是的，其实很多人，他学习东西的时候就喜欢挑，挑简单的学，书选择最”深入浅出”的，在这种心态下，没有任何一丝学好的可能性，

所以我对知识学习的目标，理解为亮点，一曰准确，二曰全面。当年学习一部分知识，如果你能做到这两点，那你将来在业务上做技术决策的时候，你面对面试官技术问题的时候，信心跟你只看过皮毛是完全不一样的。

怎么做到这两点呢？我想路子肯定有很多，而我的答案，我这里要分享的，是“建立自己的知识体系”。

如何建立自己的知识体系呢？我个人总结的经验，是下面几个步骤：

第一步，寻找线索。

你要了解一个知识，比如我想学 Web 平台的 API 了，当然可以先找一本书，看看别人都写了什么，但是我不喜欢这么干。

我大学里，学前端的东西，为了找个 id 和 name 的区别，曾经要借十几本书来，对比着看，那个时候，是真的没人告诉我，什么书比较好。所以我对别人总结好的知识，第一反应是质疑，不信。

所以我比较推荐，找一些比较准确的，你可以确定它真的足够全面的资料当作线索。对 Web 平台的 API，我就用反射：

```
for(var p in window) console.log(p)
external
chrome
document
Pointman
KISSY
define
TB
TBC
g_config
T
goldlog
JSTracker2
tmsInit
_ERROR_LINKS_HTTPS_Num
ali_analytics
_ap
g_tb_aplus_loaded
g_aplus_pv_id
lib
goldminer
goldlog_queue
g_tb_aplus_launch
_img_0.29511889841075756
```

阿里技术

浏览器里给出来的这个属性列表是不会骗人的，用这个东西作为线索，我就很有信心。

同样可能比较适合做的资料，还有一些标准文档的附录，和源代码里的结构定义。

第二步，是建立联系。

比如说，看下面几个 DOM 属性：

childNodes	children
parentNode	parentElement
nextSibling	nextElementSibling
previousSibling	previousElementSibling

阿里技术

这里，左边一列是操作 Node 的，右边一列是操作 Element 的，它就存在一定的对应关系。

一般来说，我们找对应关系的方式有以下几个依据：

- 美感
- 完备性
- 操作同一组数据

特别提一下，操作同一组数据，正是面向对象的核心概念，对前端而言，有点不一样的是，所有的 API，根都是 window，所以，其实大部分的 API，可以依据面向对象的数据和操作的观点进行划分。

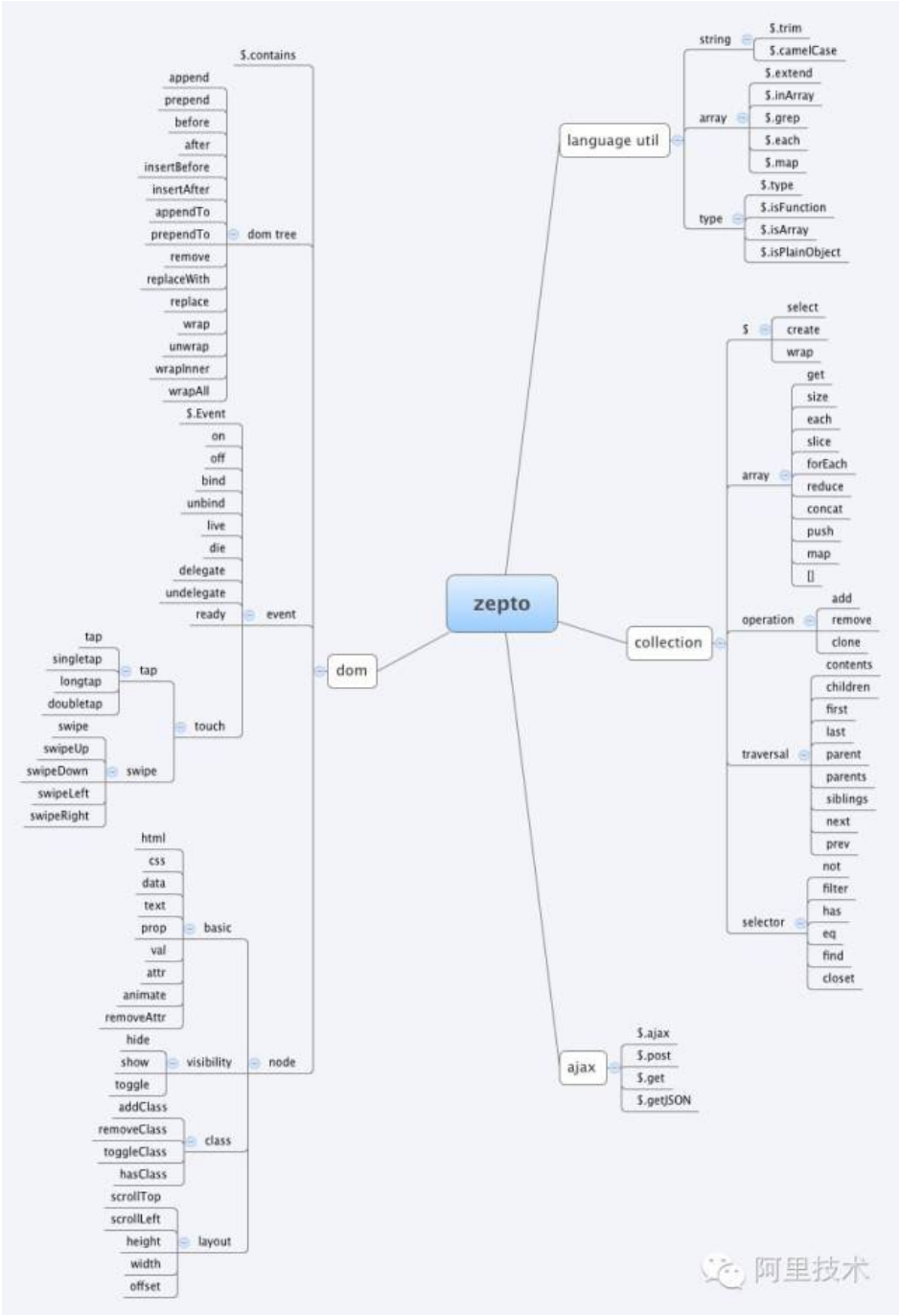
第三步，是分类。

这里我给出一个实际一些的例子，下图是我对 zepto（移动简化版 jQuery），的 API 分类建立联系以后，我们依据知识之间的联系，进行分类，就可以得到一张图谱，在这个图里面，你就可以非常清楚地知道，哪些知识，是非常重要的，哪些，其实是可以互相替代的。

而一旦有你之前没见过的东西，你又能通过把它放到图谱里，来快速理解它，或者找出一些很好的替代方案。

比如说面试的时候，如果面试官问你 bind 和 unbind 怎么用，你还不会，这时候，如果你心里有这张图，你就不至于一脸懵了，你可以说，虽然我不知道 bind 和 unbind，但是我知道 live 和 die 啊，我又知道 on 和 off 啊。

这张图里我们就可以看出，collection 里面的东西，多半没什么用，而节点操作里，肯定就都很有用。

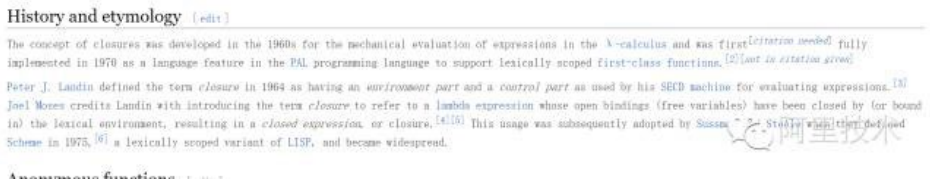

zepto
dom
$.contains
append
prepend
before
after
insertBefore
insertAfter
appendTo
prependTo
remove
replaceWith
replace
wrap
unwrap
wrapInner
wrapAll
dom tree
$.Event
on
off
bind
unbind
live
die
delegate
undelegate
ready
event
tap
singletap
longtap
doubletap
tap
swipe
swipeUp
swipeDown
swipeLeft
swipeRight
swipe
touch
html
css
data
text
prop
val
attr
animate
removeAttr
basic
hide
show
toggle
visibility
addClass
removeClass
toggleClass
hasClass
class
scrollTop
scrollLeft
height
width
offset
layout
node
language util
string
$.trim
$.camelCase
array
$.extend
$.inArray
$.grep
$.each
$.map
type
$.type
$.isFunction
$.isArray
$.isPlainObject
collection
$
select
create
wrap
array
get
size
each
slice
forEach
reduce
concat
push
map
[]
operation
add
remove
clone
traversal
contents
children
first
last
parent
parents
siblings
next
prev
selector
not
filter
has
eq
find
closet
ajax
$.ajax
$.post
$.get
$.getJSON
阿里技术

第四步，是追本溯源。

当我对一个知识体系的全貌有了概念以后，占了全面两个字，接下来需要确认它的准确性。很多知识，在社区，会有很多的争议，该相信谁呢，这是个问题。而我的答案，就是追本溯源，去找它最初的讨论和定义。

有一个真实的案例，就是闭包这个概念，曾经我们很多人的理解都是错的，把闭包和 scope 的概念给混淆起来，认为闭包是函数的执行环境上下文，但是有一个叫做 hax 的（很多人应该都认识他，哈哈），他就对此提出了质疑，认为闭包就是函数。于是我就去查证闭包的概念。

大家都知道，wiki 其实是不准确的，但是其中有一段，基本不会太有问题，就是历史。下图是 closure 这个词条的历史部分：



History and etymology [edit]

The concept of closures was developed in the 1960s for the mechanical evaluation of expressions in the λ-calculus and was first [citation needed] fully implemented in 1970 as a language feature in the PAL programming language to support lexically scoped first-class functions.[2] [not in citation given]

Peter J. Landin defined the term *closure* in 1964 as having an *environment part* and a *control part* as used by his SECD machine for evaluating expressions.[3] Joel Moses credits Landin with introducing the term *closure* to refer to a lambda expression whose open bindings (free variables) have been closed by (or bound in) the lexical environment, resulting in a *closed expression*, or closure.[4][5] This usage was subsequently adopted by Sussman and Steele when they defined Scheme in 1975,[6] a lexically scoped variant of LISP, and became widespread.

从这段历史里，我找到了一个名字，Peter J Landin，他是提出者，那么，我就去看看他到底是怎么说的，于是我去 google 学术搜索，找他的文章

果然找到了，于是我们看看原始的文件

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to *E* of a λ-expression *X* is represented by the closure denoted by

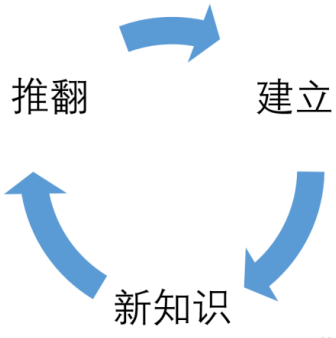
阿里技术

这个定义，对应到我们今天 JS 里的闭包，是稍微有点区别的，但是它毫无疑问，是包含了两个部分环境部分和控制（代码）部分，所以其实，闭包就是对应着 JS 的函数，而之前，普遍的观点是认为闭包只包含环境。

所以这个追溯的过程，能够帮我们真正搞清楚对错。

除了 wiki-google 学术搜索的组合，还有一些邮件列表和 github 提交历史，也是非常适合去查证一些概念和技术的历史的。

最后说，我讲的这个建立知识体系的过程，是不断接受新知识，挑战、质疑原有的体系，推翻再重建，每一次循环，你的知识体系都变得更加坚固，更加强大。



阿里技术

下面分享的一部分，是关于能力培养。

能力培养其实重要性很高，但是其实说起来，内容却很少。只有两点：教材、训练。

对知识学习，我是主张建立自己的体系，不要去相信书，但是对能力培养，我的观点就刚好相反，我觉得能力的体系，恰恰是难以自己建立的，需要教材去指导。这是由两者的复杂程度和变化速度决定的。

想培养能力，就要找经典的教材来学习，像算法导论，The C++ Programming Language 这些经典，几十年都没有过时。

注意这里我用了教材，而不是书。

教材和书最大的区别，就是有没有习题。

在我看来，内容再难的书可以一星期读两本，但是教材一定不行，教材一定得花几个月的时间，一边读一边做习题。

于是谈到训练。

其实有个事实是，工作以后，只有极少数人仍然能够做到训练，比如我自己的编程能力，我自觉工作 7、8 年，几乎没有过进步。

训练应该是系统的（需要教材）、主动的，这两个特点不可或缺，有人会觉得，我真的工作很辛苦，每天都要加班，但是其实，任何被动的痛苦，都没法给人带来进步，你的痛苦倒是可能给老板带来更多收入。

如果面临困境，可以选择系统训练来提升自己，但是对大部分人来说，可能更乐

于选择一个一个变通的办法：养成习惯，让工作变得更有挑战。

这个事情其实有不少理论，比较有名的是 Noel Tichy 提出的心理舒适区、学习区和恐慌区。选择一份对自己来说具有挑战性的工作，正面解决问题。

技术圈里流行一个笑话，说的是一个人，工作了三年，却只有一年的经验，因为后面两年都在重复第一年的工作。

所以我们要做的事，就是永远不重复劳动，当你觉得现在的工作，越来越舒适，越来越缺少风险的时候，就应该引起警惕了。

而虽然训练是个很困难的事情，其实大家也不必过于担忧，虽然到处都是“一万小时训练”的言论，现在各大公司的招聘门槛，在我看来应该都卡在几百小时训练的程度。所以我想说，一万小时太久，只争朝夕。希望看到大家成为更好的前端，做更好的自己。

以上是我分享的所有内容。

如何成为一名顶尖的阿里架构师?

无叶

在技术圈，架构师一方面是已经被说烂的职务，另一方面也是让人困扰的职位，行业发展到现在似乎人人都是架构师，各种架构图绚丽多彩漫天飞舞，同时永远有人在抱怨架构太烂、坑太多。

那么到底什么是架构师？如果有一天把你丢到架构师的位置上你会怎么做？ 做什么呢？今天，阿里国际技术事业部的无叶，与大家坐一起，聊一聊。

一、两种架构师

工作五年以上的童鞋，或多或少都会有这样的经历：在小团队或者项目中承担非明确的架构师职责，我们做项目或者产品的关键设计和实施；负责产品基础设施；引入新的理念，框架；解决团队中的复杂问题；在团队成员中享有较高的声誉；被老板

认为是团队的关键人物。

如果有一天我们决定（或者其他原因）去做一个专职架构师，那么这两者会有什么区别呢？是否只是之前的方式的延续就足够？

我把第一种状态称之为“兼职架构师”，因为处于这种状态下的同学大部分的时候担当开发人员、PM 的角色，只有在小部分时间承担了架构师的部分角色。做的绝大部分事情是自己可控的，自己做架构自己做实施或者在小团队中推行。而后一种“专职架构师”则面临的是：他们不负责具体的业务系统，而又对所有的系统负责，他们也很少直接负责项目，但是职责却要求他们必须对项目要有提前把控，他们面对的是更大的团队，更大的问题域。

当然每一个人对是否应该存在“专职架构师或团队”都有自己的想法，从阿里的历史来看单独的架构团队也是分分合合。在这里不去探讨，我们关心的是如果有，可以怎么做。

二、专职架构师的职责

首先要弄清楚的是专职架构师的职责到底是什么?

微软对架构师有一个分类：企业架构师 EA(Enterprise Architect)、基础结构架构师 IA(Infrastructure Architect)、特定技术架构 TSA(Technology-Specific Architect) 和解决方案架构师 SA (Solution Architect)。这个分类是按照架构师专注的领域不同而划分。

在阿里除了 EA 之外的领域大家可能会同时涉及到，只是不同的时期偏重点不一样。比如前面说的“兼职架构师”可能偏重 SA？专职架构师偏向 IA+TSA。另外一个角度专职架构师更多考虑问题域和相应的系统架构，而“兼职架构师”更多的是产品的系统架构，具体来说我认为专职架构师重要的职责如下：

职责一：全局的技术规划

架构师第一个最重要的职责是技术规划，架构师最重要的产出是架构，架构就是蓝图，就是阿里常说的一张图。画蓝图就是做“全局的技术规划”，这张图上有什么？ 没有什么？什么时候有？什么时候没有？当你尝试去画图的时候一连串的问题拷打着你。对于一个架构师的心力和体力都是很大的考验。只有这张图非常清晰明确才能指引整个团队在同一个时间向同一个方向前进。

为了这张图你必须和业务紧密沟通，你必须有对应的技术深度和广度，在选型上有自己的方法论，敢于做出判断和决策。

另外一个重点是全局。全局我的理解是全面 + 格局，全面就是你的技术规划包含各个方面的，在所有的领域都有明确的指引，所以这张图本质是一系列的图的集合；格局上不要只关注短期利益，更多关注长期利益。不止关注团队利益，更多从公司角度出发，只有这样长期才能为团队带来更多的成长。

职责二：统一的方法 & 规范 & 机制

架构师第二个重要的职责，我们不仅仅要提供蓝图，还要提供配套的方法论 & 规范 & 机制来保障有序进行。蓝图确保整个团队在同一个时间向同一个方向前进。规范确保前进是有序的。为了有序，你必须拆解你的图，纵向、横向、功能内聚等等纬度拆解到权责清晰对等。这是一项相对复杂且繁琐的过程。

职责三：完备的基础构建

除了蓝图确保整个团队在同一个时间向同一个方向前进、规范确保前进的有序的、我们还需要提供强大的武器库，基础构建的完备程度决定你的团队装备是小米 + 步枪，还是飞机 + 大炮。完备的基础构建是否全部作为实际架构的职责，可以因情况而定，比如是否有实体的架构组。但是架构对此应当负责。

职责四：落地的规划才是架构

如果规划不能落地就是传说中的 PPT 架构师，我甚至觉得这是对专职架构师最大的挑战，前面的几个职责更加偏向硬实力，而这一个更多的是软实力的体现。专

职的架构师如果不去关注落地的话慢慢就会架空，变成 PPT 架构师，那差不多就 game over 了。

三、专职架构师的权利

正如前面说到对架构师最大的挑战是落地层面，实际上“完备的基础构建”已经涉及到落地层面的事情，但是和完备的基础构建不同的是整体架构的落地涉及到方方面面，面临是更多影响因素：和业务的关系、组织结构、权责定义等等。

所以有人从“架构师的权利和职责”的角度出发推论谁合适做架构师。得出的结论是一个组织的领导者。因为只有他才能调动、协调组织。也有人认为架构师既不能完全负责技术团队，也不能完全游离在技术团队之外。因为负责容易屁股决定脑袋，游离就只能靠个人声望值吃饭了。

如正架构分类中 EA 的存在，很多领导者也确实身体力行的践行架构师的职责，然而精力终有限。实际上更多是平衡的过程。当然最高境界是影响力。

四、专职架构师的考核

针对前面的职责怎么考核？或者怎么设定自己目标？虽然说在不同的团队阶段，不同外在环境，不同的权责情况下不一样，但是在结果导向的背景下落地肯定是架构师重要的考核指标之一。

考核一：全局的技术规划

相比其他几项这一项是最重要又最难评价的，技术规划的好坏、全面性、前瞻性都是定性的描述，如何指引我们做出一个理性的评价呢？回归到本质上“技术规划”只是一个指路灯，团队中每一个人能不能看到“指路灯”就到达目的地是指路灯价值的体现。所以无论是唯价值论还是唯口碑论衡量的其实是同一个东西。

考核二：统一的方法 & 规范 & 机制

这一项的考核就相对容易多了，无论是业界还是每一个架构师本身都有自己的一套方法，所以只需关注这些东西对应的产出。

考核三：完备的基础构建

我认为在大公司，大部分重量级的基础构建已经是非常完备，对于架构师来说更难的不是从 0 到 1，而是克制、边界和从 1 到 2 的过程。对于架构师也好、技术团队也好“从 0 到 1”总是充满了吸引力，加上技术人的特征，大公司技术史上永远不缺少重复的轮子，创建这些轮子成就了一代一代的同学，拆除这些轮子再成就了一代的同学，所以克制尤为重要；有了克制跨团队的合作就尤为重要，对应的有两个点一是清晰边界，二是共建。

考核四：落地的规划才是架构

虽然说落地是非常不控的事情，但是考核却很容易的：做到就是做到、没有就是没有、质量好就是质量好，标准非常清晰。过程中只需要紧跟拆解的事情结合实际的组织和业务情况做出决策。

五、实施的一些想法

对现阶段团队的情况来说，我认为第一是建立“架构语言”，有了语言才有沟通协作的基础，所谓的“架构语言”并不是什么新的东西，而是产品的业务架构，用例和领域模型；研发的应用架构，组件和时序图；运维的部署架构等等。

第二是建立“认同体”，无论是通过技术能力、知识传递、领域组织等各种方式逐渐形成“认同体”，且在其中形成架构体系对应的人员体系。

第三永远做服务者，架构师对应的客户是团队的每一个成员，必须始终保持客户第一的心态。架构师存在的目的是成就研发团队每一个同学，我们提供必要的平台、服务和空间，然后彼此成就。

最后借用一句话：从无到有的是架构；从表到里的是抽象；从粗到细的是设计。大家对架构师有哪些看法。

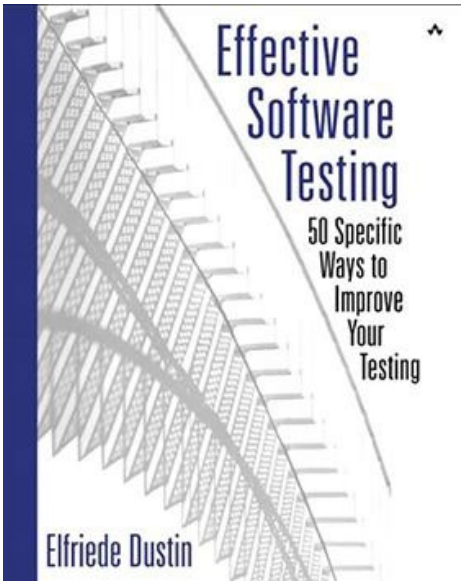
哪些技术好书值得一读再读？这有一份经典书单

阿里技术

喜爱读书，就等于把生活中寂寞无聊的时光换成巨大享受的时刻。有了书，各个领域的智慧，几乎触手可及。我们能有幸站在前辈、巨人的肩膀上，看更远的风景。

4 月 23 日世界读书日，阿里九位技术大牛为你推荐好书，与你一起共同成长、探索未来。

推荐书籍：《Effective Software Testing》

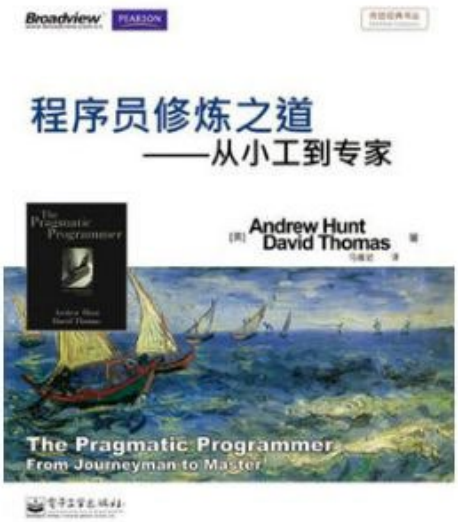


推荐者：霜波（资深测试开发专家）

推荐理由：和其他传统软件测试书籍相比较，对自动化和持续集成的方案研究比较深入，能直面自动化和持续基础现阶段的一些问题，将软件测试的周期提前到需求，设计和开发的阶段，估计产品和开发一起加入测试的工作。同时对于质量的管理有一些自己的见解和实践经验，推荐给所有技术同学一起共享。

阿里妹：天猫双 11 大队长推荐的好书，相信会为你打开一个新世界。

推荐书:《程序员修炼之道 – 从小工到专家》



推荐者：叔同（资深技术专家）

推荐理由：这是一本阐述方法论的书，关于程序员的自我修养，解决问题的方式、态度和哲学，是向高级程序员和专家进阶的思想启蒙书。从基本原则到编程风格，从思维方式到职业规划，内容覆盖广泛，兼具思想性和实用性，非常开拓视野提升格局。行文简单易懂，运用和实践却是不易，值得一读再读。

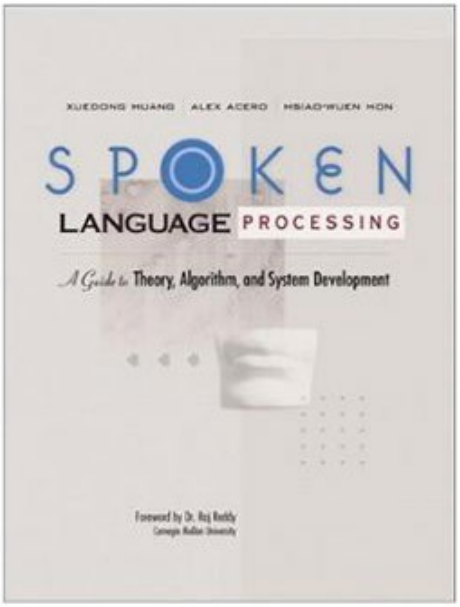
推荐书籍:《设计模式之禅》

推荐者:孤尽(阿里巴巴代码中心)

推荐理由:对于设计模式,它能够指导我们编写出可维护性好、可扩展性强的代码,对于设计模式的理解层次,我分成五个等级,以金庸小说人物为例:第一级是杨铁心,即只知道所有设计模式的概念和定义;第二级是丘处机,能够写出相关设计模式的 demo;第三级是梅超风,能够在现实中找出各个设计模式的原型;第四级是郭靖,能够在系统中抽象出来设计模式,并且合适地使用,有效隔离变化点。第五级是扫地僧,完全忘记设计模式,但写出来都是设计模式。《设计模式之禅》是一个非常好的入门,至少武功能够达到郭靖层面,讲解各个模式比较浅显易懂,促进大家在软件设计能力上的进步。

阿里妹:提到孤尽,很多人都会想起《阿里巴巴 JAVA 开发手册》。在阿里技术公众号回复“手册”,即可下载哦。

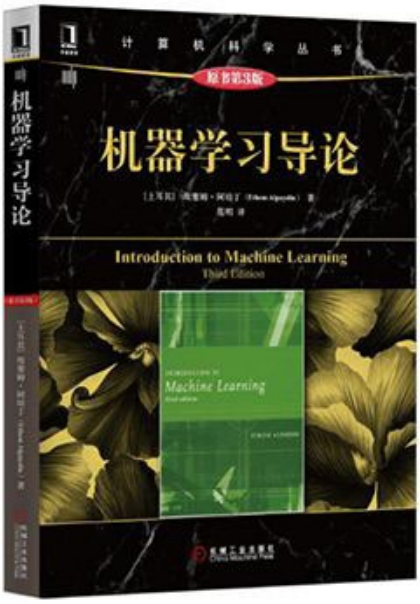
推荐书籍:《Spoken Language Processing: A Guide to Theory, Algorithm and System Development》



推荐者:智捷(资深算法专家)

推荐理由:“当今的知识世界是一个扁平的世界,很多人工智能算法已经通过 open source 的工具和 opensource 的数据库,使得大家可以轻易的获得并复现出结果。在这些“新知”之外,今天咱们要推荐一本老书,即由黄学东(微软 Technical Fellow,语音及语言 AI 技术负责人)、Acero(Apple Siri 高级总监)和洪小文(微软亚洲研究院院长)在多年前合著的语音和语言入门级专著。这本书深入浅出,将基础理论、语音识别、语音合成、语义理解和对话系统等进行了系统性的介绍,是了解口语对话系统最基础模块的一条捷径。通过这本书的‘面’上的引导,我们可以发现感兴趣的‘点’,并从这些点上更深入的进行研究和实践的工作。”

推荐书籍:《机器学习导论》

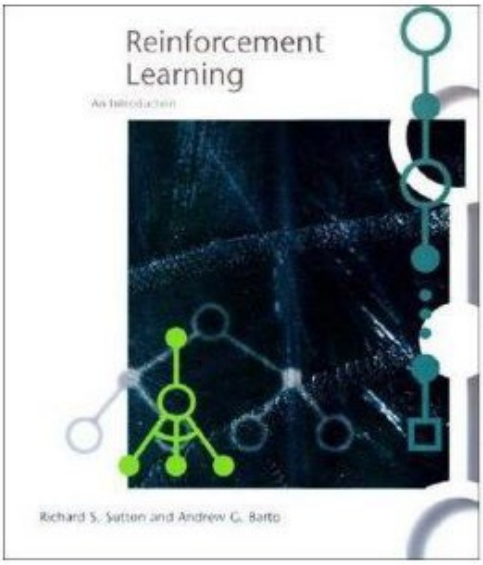


推荐者:粤谦(资深技术专家)

推荐理由:这是一本很好的机器学习入门级教程,非常适用于高年级的本科生、研究生等同学学习机器学习领域的知识。这本书基本上涵盖了机器学习的相关知识,从无监督学习、参数方法、非参数方法、线性判别式、决策树、概率图模型、贝叶斯估计,到多层感知器、SVM 和核机器、组合学习、强化学习等,都有较为全面的介绍。对算法原理阐述的比较清晰,也提供的相关的伪代码做深入的研究,并附带课程作业,非常适合机器学习的爱好者在熟悉理论基础的同时,可以进一步了解算法的原理并加以实践。

阿里妹:机器都开始学习了,何况我们呢~?

推荐书籍:《Reinforcement Learning: An Introduction》



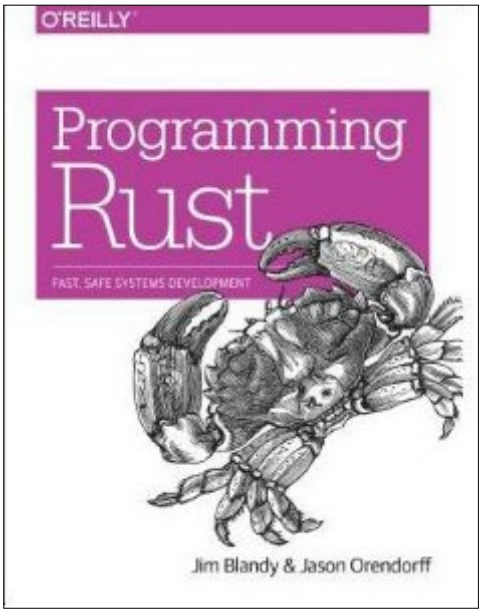
推荐者:仁重(资深算法专家)

推荐理由:本书是强化学习领域的最经典书籍,它既是初学者打好强化学习基础的必读著作,也是强化学习研究者们需要温故而知新的强化学习宝典。该书的作者 Richard S. Suttion 和 Andrew G. Barto 是强化学习顶尖学者的代表,在此领域深耕超过 30 年。这本书详细地介绍了强化学习发展历程、经典方法以及现实应用。该书第一版于 1998 年发表,第二版于最近撰写完成。第二版保留了第一版的整体结构,对一些细节问题进行了更深入的剖析(比如:通过策略梯度的推导说明了经典的 Tabular Actor-Critic 方法的由来),同时也加入最近十几年强化学习领域的重要进展。

同时也推荐我们阿里自己的作品《强化学习在阿里的技术演进与业务创新》,本书从多个实例讲述强化学习如何在工业界应用。强化学习已经在游戏中获得了巨大的成功,但在实际工业界中,大家都还处于初步的尝试阶段,本书给大家提供一些思路,我们是怎么把强化学习应用在实际业务中,以及会遇到什么样的问题,怎么去解决,希望可以给大家一些帮助。

阿里妹：关注阿里技术公众号，回复“强化学习”，即可下载《强化学习在阿里的技术演进与业务创新》（一般人我不告诉他）~

推荐书籍：《Programming Rust》

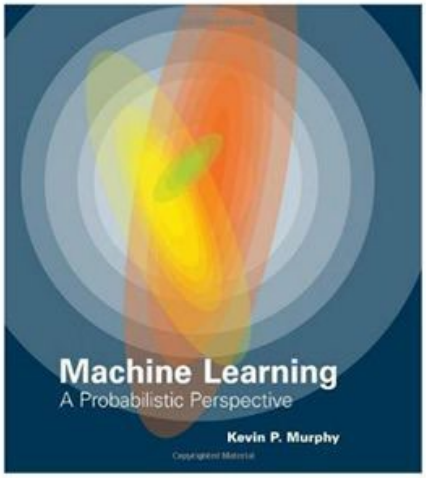


推荐者：布民（资深技术专家）

推荐理由：随着互联网与移动应用的普及，大规模分布式系统正变得越来越重要。系统构建优势往往可以直接对应到商业竞争优势。与此形成对比的是，系统编程——也就是在资源受限情况下，实现安全、稳定和高性能（并发）系统——并不为大部分程序员所熟知。Rust 是一门新兴的系统编程语言，它为安全性和并发而设计，提供高层的抽象，同时有 C/C++ 的性能。希望大家通过阅读“Programming Rust”，不仅能欣赏到系统编程的美，也能帮助普及和推广系统化（编程）思维。

阿里妹：这本书特别适合有经验的开发者（特别是 C++）阅读，最好能够边做项目边学习，理解会更加透彻。

推荐书籍:《Machine Learning: A Probabilistic Perspective》



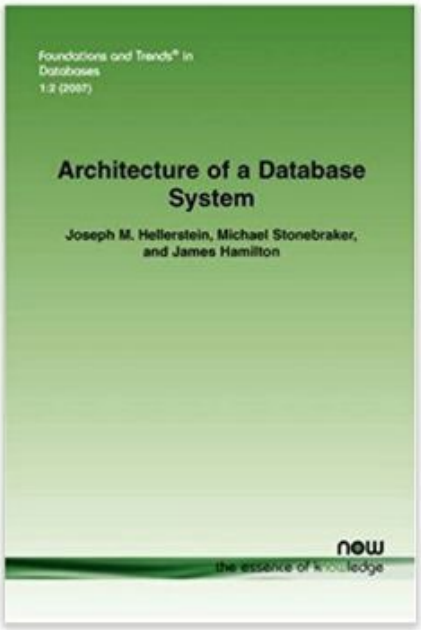
推荐者：鸿侠（资深算法专家）

推荐理由：当今网络化的电子数据洪水泛滥，大数据公司的每一个项目都需要自动化的数据分析方法。机器学习提供了相应的解决方案，不仅可以自动检测数据中的模式，也可以使用学习到的模式来预测未覆盖到的数据。推荐的这本书使用统一的概率方法为机器学习领域提供了一个全面和独立的介绍。

本书的深度和广度覆盖都很好，涵盖了概率，优化和线性代数等必要的背景材料，并详尽的涵盖了机器学习的最新发展，包括条件随机场，L1 正则化和深度学习等比较流行的方向，并且提供了相应算法的伪代码。所有主题都用彩色图像进行了丰富的说明，并从生物学，文本处理，计算机视觉和机器人等应用领域中绘制了实例。

阿里妹：ML 领域经典教材，能够帮你建立起对该领域的整体认知。理解 80% 以上内容的童鞋，请不要犹豫速砸简历来～

推荐书籍:《Architecture of a Database System》



推荐者:圭多(资深技术专家)

推荐理由:此书是数据库图灵奖获得者 Stonebraker 老爷子在 2007 年完成的,全书不长(119 页),但极具功底。老爷子通过此书,向大家剖析了一个成熟数据库系统的整体架构,以及数据库的各个核心模块,包括:SQL 与优化器、内存和存储管理、事务和并发控制等的设计原则和实现方式,是全面了解数据库系统的第一选择。

阿里妹:想要了解数据库整体架构、内部运行机制,看这本就对了。

阿里技术大牛最爱的“闲书”，你看过多少?

阿里技术

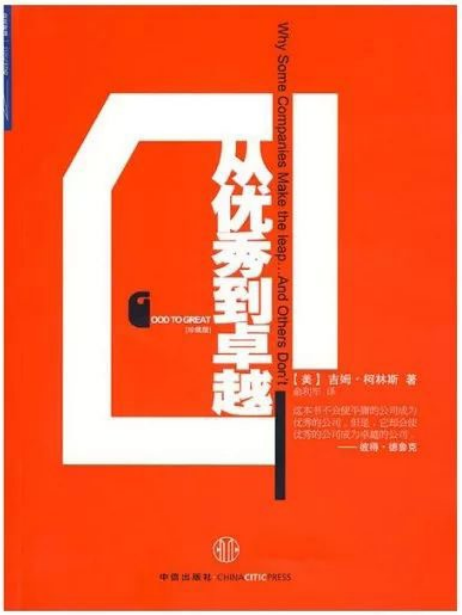
在忙碌的写代码、修 bug 生活里，你有多久没有闲下来，读读“闲书”，取悦自己了呢?

正如梁文道所说，“读一些无用的书，做一些无用的事，花一些无用的时间，都是为了在一切已知之外，保留一个超越自己的机会，人生中一些很了不起的变化，就是来这种时刻。”

今天，阿里的数位技术大牛，带来了最爱的“闲书”，无关技术，却值得一读，期待与你共赏。

工作很忙，效率很重要。以下书籍或许能帮助你提高时间利用率，突破事业瓶颈，打开另一番天地。

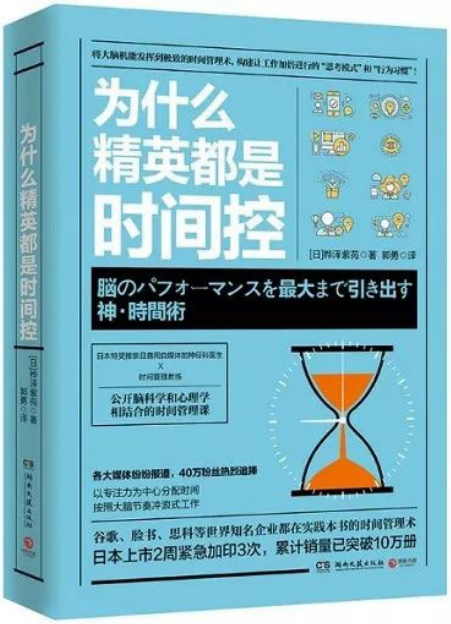
推荐书籍:《从优秀到卓越》



推荐者：索尼（研发效能事业部研究员）

推荐理由：优秀或许不难，但是做到卓越，除了能力外更重要的是意志和胸怀，乐观且皮实，聪明而自省，这些说起来都是大家都懂的道理，但是真正在面对工作、生活中的每一个选择时，做起来却太难太难。本书系统性描述一个能带领团队从优秀到卓越的第五级经理人的特质，通过他我们可以具象建立面向未来的组织画像，为自身的成长建立标杆。

推荐书籍：《为什么精英都是时间控》



推荐者：林轩（系统软件事业部资深技术专家）

推荐理由：在阿里，人人都聪明；在阿里，人人都努力！那么，在一群又聪明又努力的人当中，大家拼的是什么？效率！谁能把 24 小时用出 48 小时的效率？你应该如何分配一天的时间？什么时间应该高速工作？什么时间又应该安心休息？当专注力下降的时候，如何一键修复？每天高强度的工作，你是不是经常会感觉疲倦和体力不支？

推荐《为什么精英都是时间控》。拒绝鸡汤，拒绝晦涩的工具书语言，作者用简

单的语言阐述了有效的时间管理方式。和作家以及商界人士写的时间管理书籍不同，作者是医学博士出身，现在也是日本的神经科名医。所以读这本书，一不小心就收获了许多医学术语，交感神经，生长激素，血清素……知其然知其所以然，理解了人体的奥秘，你就可以更好的掌控自己的精力，double 你的效率！加油吧。

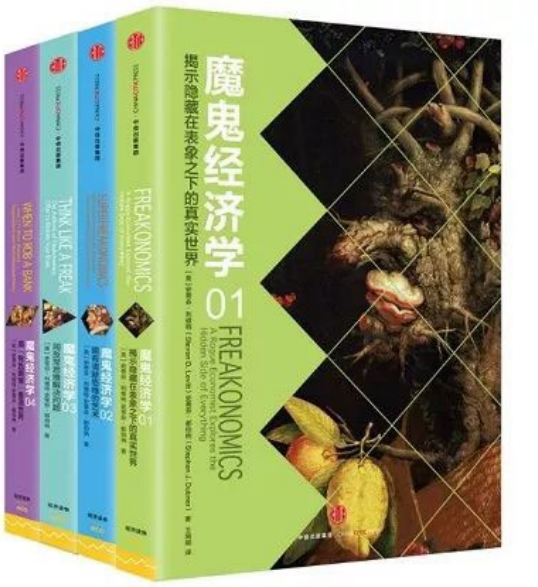
推荐书籍:《创新者的窘境》

推荐者：大舞（计算平台事业部资深运维专家）

推荐理由：从另外一个角度去审视企业的创新，当一个企业价值网络形成后，对整个组织不同阶层的员工都会产生一定的影响，特别是非高层员工他们会依据自己的理解来决定资源的分配，如何在企业中保持破坏性创新的体系是一个值得思考的问题。

“经济基础决定上层建筑”，下面推荐几本经济学读物，不信你不想看 ~

推荐书籍:《魔鬼经济学》

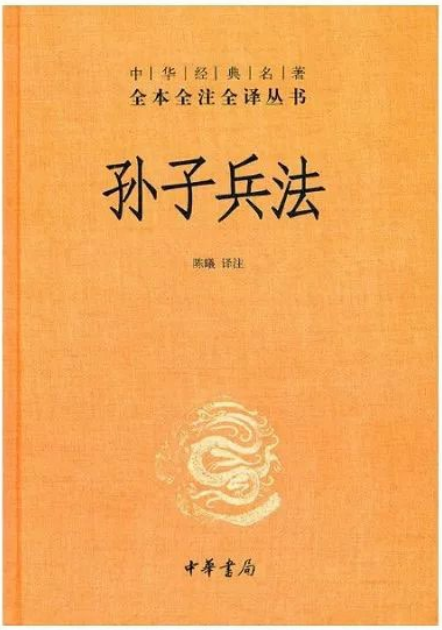


推荐者：祁越（信息平台事业部资深技术专家）

推荐理由：给你一个非常不同的视角和思考方式看待世界和身边的人。读这本书的感觉仿佛置身真实的案件侦破亦或是看到一个女孩缓缓解开头上的面纱，会让你很爽；例如书中“犯罪率升高是由于电视的普及”这类看似匪夷所思的观点被作者用新颖的逻辑分析进行了推理，对提高我们在日常中提升思考深度很有帮助。

看完经济学，军事岂能错过。大丈夫心怀天下，书中便有“天下”。

推荐书籍:《孙子兵法》

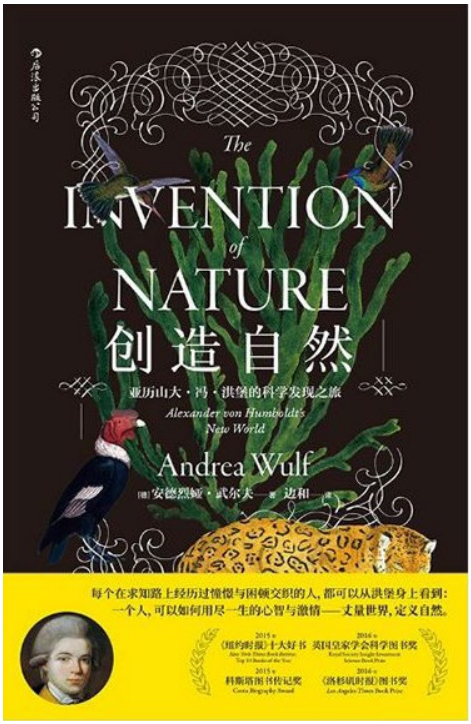


推荐者：沈询（中间件事业部资深技术专家）

推荐理由：这本书在讲的是战略，而且解释的最通透，最精炼，如果只推荐一本书，我觉得还是这本吧，从道，天，地，将，法的角度来观察战略与兵势，是古今中外通用的道理，这本书属于读一本书可以代表一百本书的典范。

“世界这么大，我要去看看”，如果身体没办法走向远方，让“心”出走看看也是好的。看看他人走过的路，经历过的人生，似乎也得到了不同的人生体验。

推荐书籍:《创造自然》

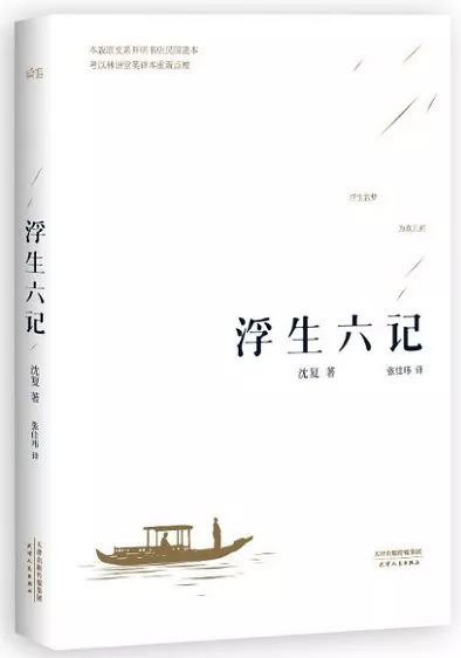


推荐者：褚霸（阿里云事业群研究员）

推荐理由：本书块头不小，但是非常精彩，我是以难抑制的激动心情捧在手里一口气读完的。作者以自然为线索展现了洪堡探险和对科学追求的传奇的一生，同时穿插着欧洲和拉美的历史和人文关键事件。今天我们习以为常的等温线、植被带这些概念以及大自然本身就是个有机体的观点就是洪堡在拉美冒着生命危险在一座座山头穿越攀爬、无数流血的脚印中得出的真知。

对科学的严谨，善于记录、测量、总结，辩证看世界的眼光，孜孜不倦探索自然的规律，利用自然的力量的行动，影响了一代代人，包括达尔文等人。在今天变化快速的数字化的世界，洪堡探索世界的方法看起来依然非常有效，推荐阅读！

推荐书籍:《浮生六记》



推荐者：樵隐（机器智能事业部资深技术专家）

推荐理由：我读《浮生六记》，原是冲着“中国文人心目中最完美的女人”去的。读完之后，对芸的观感一般，却对作者由衷倾佩，可见我不是文人，而是典型的码农一枚。作为典型性技术人员，一直觉得自己缺乏发现生活中美的能力，也缺乏在坎坷愁绪中安之若素的泰然。

读了浮生六记，为作者对生活的认真所感动，一碗藏粥，一方印章，一块卤瓜，一餐野营，一盆插花，均含无限乐趣，于是开始学习喝茶听曲，世界果然多彩了起来，于是变成为习惯，跟着生活也变得没那么焦虑了，对比起坎坷记愁的生命脆弱，一切也许都可以“佛系”处之。读《浮生六记》引发的人生思考，对于我的生活观产生了巨大影响，因此推荐给跟我一样“自认为没有生活情趣”的程序员们。

阿里技术

扫一扫二维码图案，关注我吧

「阿里技术」微信公众号

「阿里巴巴机器智能」公众号