极客时间

# 朱赟的技术管理课

TECHNICAL MANAGEMENT

从技术到管理，让你的目标函数达到最优解

计算机博士 Airbnb 技术经理 朱赟

# TABLE OF CONTENTS

管理者在进行工作分配时，会考虑哪些问题？

硅谷人到底忙不忙？

每个工程师都应该了解的：系统拆分

技术人如何建立个人影响力?

管理者不用亲力亲为：关键是什么？

每个工程师都应该了解的：API的设计和实现

硅谷面试：那些你应该知道的事儿

项目管理中的三个技巧

每个工程师都应该了解的：中美在支付技术和大环境下的差异

不要做微观的管理者

如何处理工作中的人际关系？

编程语言漫谈

兼容并包的领导方式

如何做自己的职场规划？

小议Java语言

# 开篇词|从工程师到管理者，我的思考与实践

你好，我是朱赟，英文名Angela，目前在硅谷的Airbnb支付组担任技术经理。

我毕业于中国科技大学少年班，后来在美国莱斯大学攻读计算机硕士和博士学位，期间共发表国际论文十余篇，主要涉及的领域为程序语言设计和生物信息学的大数据分析。

在中国科学技术大学少年班，我做过一些体系结构和嵌入式系统等方面的研究工作。

在莱斯大学时，我的硕士课题是程序语言设计，那个期间我掌握了很多编译器相关的理论；博士课题是生物信息学，用到了很多机器学习相关的技术。

进入Airbnb之前，我在Square公司工作。我是那里的第一位华人女工程师，那时候主要做支付和搜索。当时，公司的软件工程师团队真的是太强大了，基本都是Google Staff级别的牛人。靠着创业公司本身锻炼机会多，再加上牛人悉心指教，在Square的两年让我在支付和搜索领域获得了极大的成长。

我的人生在入职Airbnb之前，可以用平淡和专注来形容。大部分时间，我都在不断地积累和练习。我独立地学习、做科研，或者长时间地编写代码。

虽然是女工程师，但我对编程的热爱是发自内心的。写程序可以让我自然而然地沉迷其中，我可以一整天坐在电脑前噼噼啪啪地敲击键盘编写代码，饿了随便吃一点，有时候会忘记喝水，也不会觉得困。等到从代码和逻辑中抬起头来，发现一天的时间已经不知不觉地溜走了。

编程非常容易让我进入“出神”的状态。机器完全是逻辑控制的，没什么情绪，无论运行良好，还是宕机挂掉，都会有原因，肯定不是因为它开心或者不开心。我喜欢用代码与计算机沟通。

Airbnb是我踏上管理这条路的起点。最初我是组里的技术骨干，主要负责支付与交易相关业务的架构。

但是人总要成长，总要走出舒适区，我在老板的培养和提拔下，一步步从技术转成了技术管理者。

来Airbnb之后，我遇到了两位改变了我生活足迹的人。一位是池建强老师，他让我慢慢走上分享和写作的道路，我的公众号「嘀嗒嘀嗒」就是

在那个时期创建的。分享与写作让我认识了很多人，也让很多人认识了我。

另一位是我的老板杨江明，他一点点指引我走上了带项目、带团队的道路。很多谆谆教导，极大地加速了我这两三年来在职场上的进步和成熟。

不论是在技术上，还是管理上，我要走的路还很长很长。但是，很多时候，当你充满激情、快步前行的时候，反而有更多新鲜的领悟可以分享，更切身的体会可以鼓舞同行的人。

这两三年可能是我积累和成长最快的一个阶段，我获得了很多前辈、同事和朋友的帮助，因此，我也愿意把一些技术和管理上的领悟及忠告，还有在硅谷工作的体会与见识，通过这个渠道分享给大家。

我的专栏主要会有四部分内容：

- 首先是我在技术领导方面的学习感悟。很多好的道理，来自我的上级和我自己的思考，来自我在硅谷公司的见闻和实践。 有些东西我还不能非常熟练地应用，我也还在练习，但在过去一年的刻意练习中，我发现了很多行之有效的管理方法，并会毫无保留地分享给大家。
- 其次是一些技术领域的内容。这些内容源于我的工作，我想分享给你的，不仅仅是一些技术细节。我会更着重技术在领域内的应用和前景，在工作中的实践和概括。我会给出自己的观点和做法，你也许有不同的想法，告诉我，我们都将获得成长。
- 还有一部分是关于硅谷文化，我会谈一谈这些年在硅谷的见闻和经历。比如硅谷的产品经理和工程师是怎么协作的，硅谷的面试流程和开发流程，硅谷人到底忙不忙，Code Review 怎么做，编程马拉松里的趣事和价值等。这些内容将有助于你了解美国互联网和中国的差异，文化互补，兼容并蓄。
- 最后一部分是关于技术人成长的，也记述了我遇到困难和克服困难的心路历程。如何对更多的工作说不，如何建立个人影响力，如何做自己的职业规划等等，这些内容对技术人和初级的管理者，或许都会有帮助。

我喜欢和工程师们一起工作、阅读、学习和交流。我常常会折服于工程师们的睿智、呆萌、执着、单纯和幽默。他们说着别人听不懂的笑话，做着这个时代最具创新性的工作，他们充满好奇心和创造力，并希望用技术改变生活。

我是朱赟，我是个女工程师。与我处在同一条成长道路上的人很多，大家会看到相同的景色，也会遇到相同的困难。我会把自己的思考、经验和解决过的困难第一时间分享给你。这就是我写这个专栏的初衷，让我们相互陪伴，并一起成长。


极客时间
朱赟的
技术管理课
TECHNICAL MANAGEMENT
从技术到管理，
让你的目标函数
达到最优解
朱赟
计算机博士
Airbnb 技术经理
扫码加交流群

# 职场分身术：从给答案到做引导

在学校读书的时候，偶尔作业不会做，我很可能会问同学，他很可能会把答案直接给我。但是，如果我去问老师，他很可能给我一些启发和引导。这正是我对不同帮助类型最初的感受。

进入职场之后，你我也经常会碰到需要别人帮助的情况。比如说，新入职时不知道怎么搭开发环境、不知道如何去写第一个代码块；或者是一个问题有两种处理方案，而你不知道应该选哪一个；又或者遇到了一个Bug，不知道该从哪里下手。

记得我刚开始带新人的时候，对系统的各种设计和业务逻辑已经非常熟悉了，所以每次他们有问题找我，我基本上都能快速给出答案。即使我没有现成答案，也能很快地帮他们定位问题和找到答案。

当时我没有带人经验，因此也就没有想太多。当时我是小组中最资深的一个，所以每次有人问问题，我都会下意识地紧张，生怕不知道解决方案，没办法帮到对方。而我的注意力，便大量放在了“帮他找到答案”上。

但是，每次别人问问题，你都有现成的答案，或者能快速找到答案，就会渐渐导致两个问题：第一，在你这儿容易得到答案，愿意问你问题和各种琐事的人越来越多；第二，你给的是答案，下次有类似的问题，别人还可能来找你。

这样一来，你每天都会花大把时间在“带新人”上，而其中很大一部分时间就是在回答各种随机的问题。

我变得越来越忙，看起来似乎越来越重要，有时我不在，问题就成了瓶颈，因此我本身也就成了瓶颈。当组里人慢慢多起来的时候，这种模式根本不可能扩展，而那些重要和紧急的事情，我却没有时间关注和解决。

那时老板就和我说：“你不能每次都给答案，你应该试着用引导的方式，让对方学会自己找答案。”

这话听起来简单，但做起来却很吃力。直接把答案告诉对方可能只需要5分钟，但是和对方坐下来一起梳理问题，找到解决方案，帮助他自己想明白，可能要花费半小时。

时间上一对比，我更多地选择了前者。我越忙的时候，越会用最简洁的方式直截了当地给出答案。这样就造成了恶性循环：越来越多的问题找到我，我为了省时间以最快的速度给出答案，对方并没有学会自己解决，变得越来越依赖我。

当超过一个临界点时，我实在没办法处理各种随机的问题了，之前有问必答的系统崩塌了。既然没有时间去回答所有问题了，这时我就不会给出最直接的答案了，反而会选择“拖一拖”再处理，或者给出一些想法。

比如“如果是我，我会尝试去看某某文档” 这样的建议，或者是“你觉得这个线上错误可能是哪些地方引起的呢？”“你有没有试着用排除法，先把那些不可能的因素排除掉？”很多时候用不了太久，对方就会很高兴地跑过来告诉我，问题找到了，或是知道该怎么做了。

当然，我的转变其实是个漫长的过程，但是现在回过头来看，确实有一些心得可以分享。

首先是：什么时候适合直接给答案，什么时候适合给线索，让对方自己找答案。

我觉得，如果是一个新人，他刚进入全新的领域，或者所谓的答案就是某些知识点，此时不妨直接给答案或知识点。因为这些问题，即便他全然没有线索，我们也不可能让他自己去推导出业界多年发展才形成的规则和规律。

等对方已经有一定的积累和经验后，就可以让他自己去探求解决方案了。这时候我们需要给出一些提示，或者这样说：“对于问题本身，我也没有完整的答案，只有些直觉和想法。”给他方向和建议，让对方继续寻找，这样比把问题解出来，然后再直接告诉对方一个确切的答案更有意义。

其次是：如何引导。

这里最关键的一点是，问对方正确的问题，通过问题去引导对方进行深入思考，找到解决方案。当一个好问题摆在面前的时候，人们更容易主动思考，他们会跳出自己之前设定的方案和框架，换一个角度去看待问题；或者被带到一条之前没有发现的道路入口，然后自己走下去，直到找到答案。

最后是：引导的好处。

虽然最终的结果都是对方找到了想要的答案，但如果是通过别人的引导，自己摸索出来的，那么他解决类似问题的能力后续就会提升。这样的做法，还可以调动他工作的积极性，对方会产生一种自己解决了问题的成就感。

此外，更好的事情可能会发生。获得引导帮助，对方甚至会结合他的经验，经过持续的努力找到更好的答案或者解决方案，甚至引发对系统的改进。

我们常说“授人以鱼，不如授人以渔”，但在实际工作中什么时候直接给出答案，什么时候给出方法和引导呢？希望你可以参考我今天分享的内容进行实践。欢迎你给我留言，和我一起探讨。


极客时间
朱赟的
技术管理课
TECHNICAL MANAGEMENT
从技术到管理，
让你的目标函数
达到最优解
朱赟
计算机博士
Airbnb 技术经理
扫码加交流群

# Bug引发事故，该不该追究责任？

“人非圣贤，孰能无过？”技术人员也是人，因此编程过程中难免出Bug，出了 Bug 系统就会出问题，出了问题系统就会宕机。那么，Bug引发的一连串事故，该不该追究责任，又如何去追责呢？

今天我就和你聊聊 Bug 和责任的问题。

记得有一次，一个国内的访问团来公司参观。在交流的过程中，有人问：“在你们的工作中，工程师的 Bug 或者失误引发的问题，会不会被追究责任，会不会扣工资，会不会被开除？”

当时我很诚实地按照实际情况回答说：“不会。”

这个人又继续问：“那出了事故没有任何惩罚，不会有问题吗？”当时，我围绕着员工的素质、自觉性和责任心进行了回答。后来再次思考这个问题，我越想越觉得有意思。

我在 Airbnb 负责支付和交易业务，这意味着大部分的错误都等价于真金白银。无论是从用户那少收钱，导致公司亏损，还是从用户那多收钱引起法律或者合约的纠纷，只要跟钱沾了边，都不是小事情。

俗话说 “常在河边走，哪有不湿鞋”，各种因为代码问题引起的麻烦也是屡见不鲜。那么，在Bug引发问题的情况下，怎样处理才能最大程度上保持团队的主动性、责任感和执行力呢？

我们先来假想两种极端的情况：如果每个错误都会受到惩罚，会怎样；如果所有的错误都没有任何追究和跟进，又会怎样？

假如每个错误都会受到惩罚，不难想象，以下情况一定难以避免。

1. 大家都怕闯祸，所以风险高的事没人做，或者总是那几个靠谱的“老司机”做。没有机会处理这种复杂情况的人，永远得不到锻炼，也无法积累这样的经验。
2. 如果有人搞砸了什么事情，会因为担心承担后果而推卸责任，从而尽可能掩盖错误的坏影响，不让人知道。
3. 如果别人犯了错，会觉得不关自己的事。
4. 指出别人的错误就会导致别人被追究责任，因此看到有问题也会犹豫要不要指出。

反之，如果无论发生什么错误，都不需要承担后果或进行反省，没有任何担当，那可能又会出现以下情况。

1. 同样的错误可能会一再发生。

2. 小错没有被及时制止，或者没有引起足够重视，最终导致酿成大错。
3. 做事仔细的人会觉得不公平。自己为了安全起见，每次代码改动都写很多单元测试，每个项目都反复测试和预防问题；但是别人的草草而就导致错误百出，却因为显得进度更快，反而被认为更有效率。

那么，对于工作中的错误，尤其是 Bug 导致的错误，我们应该采取什么态度和措施呢？

第一，追究责任，但不是惩罚。“知其然，并知其所以然”，搞清楚在什么场景下，什么样的 Bug 引发了什么样的错误。相关人员应该尽最大的可能去做好善后工作，并思考如何避免下次犯同样的错误。

第二，对事儿不对人。在这个追究的过程中，重点在于怎么改善流程、改进制度，来避免同样的错误，而不是指责员工不应该怎么样。如果相关人员已经那么做了，为什么这个错误仍然没有及时被发现和制止？

第三，反复问“为什么”，从根本上发现问题。错误为什么会发生？有些 Bug 可能只是显露出来的冰山一角。

举一个假设的例子，因为小王的代码改动影响了小李的代码，让小李之前实现的功能不对了。在这种情况下，我们首先要问：

1. 为什么小李代码功能不对没有立马被发现？
   答：因为小李当时的测试用例没有覆盖这种情况。
2. 为什么小李的测试例不完整？
   答：因为这个地方的测试需要 mock 一个服务的返回值，但是这个 mock 的值并不是真的服务器端的返回值，所以测也测不出来。
3. 为什么要去 mock？
   答：因为我们的测试系统框架不够完善。

这样反复问，反复想，就能找出根本上值得改进的问题，而这样的结果和受益，比惩罚犯错儿的人要好得多。

第四，员工关系的建立也很关键。我们需要培养的是大家相互信任、互帮互助，为了共同的目标努力的氛围，而不是一种不安全感。这种不安全感可能是自己不够自信，害怕犯错；也可能是对他人漫不关心，或是对其代码质量有怀疑。只有大家都相信，找出问题的根本目的是解决问题，避免问题再发生，才能建立一个不断反思、不断学习、不断进步的良性循环。

最后给你留一个思考题， 这也是现实生活中我多次听说的事故。如果你是一家公司的技术主管，团队里的一位工程师因为误操作删除了线上的用户数据，这时候你又发现，上个月数据的自动备份因为某些故障停止

了，现在你该怎么办呢？


极客时间
朱赟的
技术管理课
TECHNICAL MANAGEMENT
从技术到管理，
让你的目标函数
达到最优解
朱赟
计算机博士
Airbnb 技术经理
扫码加交流群

# 每个工程师都应该了解的：AB测试

说到 A/B 测试，不论你是工程师、数据科学家、还是产品经理，应该对这个概念都不陌生。

简单来说，A/B 测试是一种数据分析手段，它可以对产品特性、设计、市场、营销等方面进行受控实验。在实验中，数据样本被分到两个“桶”中，分别加以不同的控制和处理，然后对采集回来的信息进行对比分析。

举一个例子。

假如你想修改 UI 上一个模块的交互设计，这个模块的内容是引导用户点击“下一步”按钮，但是你不知道设计改动前后哪一种效果更佳。

于是你通过 A/B 测试，让一部分用户体验新的 UI，另一部分用户继续使用旧的 UI，再对采集回来的数据进行分析，对不同组用户在这个页面上的转化率进行比较，观察在哪一种 UI 下，用户更愿意往下走。有了数据分析，我们就可以判断新的设计是否改进了用户体验。

原理就这么简单。下面我会从自己使用 A/B 测试的经验出发，重点说一说 A/B 测试中需要注意哪些问题，观点会比较侧重于工程师视角，但是对产品经理也会有帮助。

第一点：永远不要过分相信你的直觉。 有时候，我们会觉得一个功能特征的改动是理所当然的，更新后效果肯定更好，做什么 A/B 测试，这显然是画蛇添足。

这就像一个资深的程序员修改线上代码一样：这样改，一定不会出问题。我们当然不否认这样的情况存在，但每当你开始有这样的念头时，我建议你先停下来，仔细地想一想，是不是就不那么确定了呢？

把你的想法和别的工程师、设计师、产品经理深入交流一下，看看他们会不会有不同的意见和建议。不同的角色背景也不同，考虑问题的方式也就不一样。当你不确定哪种方式更好的时候，A/B 测试就是你最好的选择。

第二点：实验样本的数量和分配很重要。 如果你的实验注定没有太多数据，也许就不要去做 A/B 测试了，小样本偏差会很大，帮不了太多的忙，除非你的测试结果出现“一边倒”的情况。

另外，请确保你在 A 组和 B 组随机分配的数据是绝对公平的。也就是说，你的分配算法不会让两个桶的数据产生额外的干扰。

比如，不要按不同时间段把用户分配到不同的组里，因为在不同时间段使用产品的用户本身就会出现一些不同的情况。区域分配也存在同样的

问题，这些都可能导致偏差。

第三点：分析的维度尽可能全面。 文章开头举的例子是说，虽然你最在乎的是用户转化率，但是功能改动可能会影响很多指标，这些指标都要尽可能地测量和分析。

比如，虽然 A 组转化率略高于 B 组，但是 A 组点击后会引发 API 调用流程的变化，结果延迟高出很多，或者出错率变高了，那么 A 依然不是更好的设计。

换句话说，A/B 测试不能只关注单一指标，测试目标虽然是转化率，但倘若高转化率的方案会导致其他风险，比如提高了出错率，也应当舍弃。

第四点：其它组的改动对 **A/B** 测试产生的影响。 当 A/B 测试成为一个广泛使用的工具后，产品很多特性的改动都会用到这个工具。这也就意味着，当你在采集数据做分析的时候，别人也在做同样的事，只不过策略和数据样本不同。

换句话说，你在跑 A/B 测试比较 A 和 B 的优劣，另一个同事在跑 A/B 测试比较 C 和 D 的优劣，结果因为实现细节的原因，A 组中大部分样本同样也是 C 组改动过的样本。这样一来，两个实验可能就会相互影响。因此，你要做足够的分析，确保实验结果考虑到了这种相关性的影响。

第五点：比较值的趋势必须是收敛的，而不是发散的。 要想比较结果有实际的统计意义，一定是每天采集数据的比较结果逐步收敛，最终趋于稳定。如果一周内 A 比较好，后面又开始波动，B 变得更好，这样来回波动的结果是没有太大参考价值的。

另外，即使比较值趋于稳定，还要确保这个稳定数据所处的阶段不在一个特殊时期。如果恰好有促销或者类似的市场活动，那么即便获得了稳定的结果，这个结果也不一定是普适的。

第六点：数据埋点。 数据的埋点和采集是 A/B 测试成功的关键。

怎么样进行埋点呢？总体来说，这其实和每个公司的代码架构有很大的关系。公司使用哪种方式触动事件、记录事件，尽可能地重用。

前端埋点一般可以采集实时数据，后端埋点可以采集实时事件，也可能是一些聚合数据。要视具体情况和应用而定。

第七点：形成一个流程，或者设计一个工具。 这一点很重要。A/B 测试作为一个工具，只有在它足够灵活、好用的情况下，才能更广泛地应用到日常的产品迭代和开发中。虽然说这个方法很简单，但是做好一套包括埋点、采集、处理和具备 UI 的工具，会让工程师事半功倍。

第八点：试图给每个结果一个合理的解释。 不用过分相信数据，也不要拿到什么分析结果都照单全收。试着去给每个结果一个合理的解释，不要觉得结果比期望值还好，就不用思考为什么结果如此完美。这可能并

不是一件好事情，实际情况是：如果解释不了，可能它就是个 Bug。

第九点：必要的时候重新设计实验。 很多实验会有不同版本，每个版本都会根据实验结果做一些改动和调整。如果发现实验设计上有漏洞，或是代码实现有问题，那就需要随时调整或者重新设计实验，重新取样、分析。实验的版本控制，会让分析和重新设置的过程更加快捷。

第十点：不同客户端分开进行实验。 Web 端、iOS、Android 尽可能分开观察。很多时候你会发现，同样的实验数据对比，在不同的客户端会有完全不同的结果。如果不分开，很可能让数据变得难以解读，或者出现“将只对移动客户端成立的结果扩展到 Web 端”，这样以偏概全的错误。

最后，我们来做一个小结。 今天我结合自己的实际工作经验，为你讲述了 A/B 测试中需要注意一些问题。

A/B 测试是一种行之有效的产品验证和功能改进方法，很多互联网公司，如Google、Facebook、Airbnb 等都有自己的 A/B 测试工具，他们会基于工具和数据验证自己的想法，持续进行功能改进、推动产品的发展。

如果你也在做 A/B 测试实验，可以对照我在文本中提到的那些问题来思考，相信你可以做出更好的测试结果。

# 如何帮助团队成员成长

以前做工程师的时候，我更多是单打独斗，只要把自己的代码写好，把负责的项目做好就可以了。那时候，我关注的更多是自身和相关的技术与业务。转型为技术经理之后，我意识到，必须停止只思考自己的状态，我需要把更多的精力放到其他人和团队上面。

一个优秀的技术管理者应该做哪些工作呢？

我想，至少需要涵盖3个重点。

1. 帮助团队成员迅速成长。 这包括通过指导、反馈、监督、交流、协调资源等方式帮助下属提升能力，迅速成长。
2. 明确地分解与布置任务。 这不仅包括分解和布置任务，还包括界定需求边界、制订计划、选拔人员和工作授权等。其中，界定需求边界，又要求与上级和下属细致沟通，确定下属需要做什么和怎么做。
3. 建立有效的合作关系。 这里指代的“有效”，指的是与下属、上级和相关部门建立坦诚交流和相互信任的合作关系。

今天我会主要聊聊第一个问题，其他留待后面的文章里继续讨论。

## 首先，不要陷入静态思维

如何帮助团队成员成长？要回答这个问题，我们不妨先换位思考一下。如果我们是下属，那么：“老板或上级的哪些举措，会促进你的成长？哪些行为，又会成为你成长的阻力？”

我们都知道，好的上级会给你机会、空间和支持，让你成功，但不好的上级却是各有各的不同。

1. 他会用固定的眼光看待你的能力。如果今天你的能力水平在一个级别上，他的思维在很长时间内都觉得，你只能做这个级别的事，而不去考虑你尚未被挖掘的潜力和创新能力。
2. 他会过于看重天分而不是你的努力。看重独立作战的能力，而忽略了你的协调作战等软技能。
3. 他会过于关注你的错误，而不是主动通过这个错误来帮助你成长。
4. 他会认为团队目标比每个人的成长更重要。
5. 他给你的反馈会偏评价或判断，而不是帮助你提升。

如果这些思维模式表现在他的行为上，就会是这样一些场景：

1. “这个问题很棘手，我觉得他肯定处理不好。”
   （你让他尝试过吗？）
2. “这个任务需要对系统比较了解，所以只有 A 才能做。”
   （别人不能学，不能通过请教别人去了解吗？）
3. “上次他就搞砸了，这种事他肯定做不了。”
   （只有一次失败还是多次？一次失败的机会还是应该给的。）
4. “我们必须严格按照期限完成这个项目，目前看来由 A 做最合适。”
   （如果还有一个 B，虽然不如 A，但这样的挑战对她来说是一次很好的锻炼机会，那是不是可以考虑呢？）
5. “你今年参与的三个项目都失败了。”（这种反馈，远不如指出某一两点对方可以改进的具体内容。比如：“我觉得每个改动前，最好做点实验，了解一下用户数据，再决定改动是不是合理。”）

好了，现在转换回你是管理者的角度。

如果你曾经说过，或者在脑海中浮现过这样的话或论调，就要警醒了：自己在一定程度上是否在用静态的思维去对待团队成员的成长。

一旦你有了这样的心态，无形中，就给团队中每个成员的成长设下了一道阻力。

# 看看大公司，他们怎么做

在硅谷有很多公司如 Google、Airbnb、Facebook等，在提升一个人到下一个级别时，通常采用这样的做法：

> 他们对每个级别都在各个方面设定了一些标准；比如，你在技术上要达到什么水平，执行能力如何，做出的项目是不是有足够的影响力，是不是能够独立地去解决各种困难，有没有对别人的成长做出贡献等等。

这样的话，那一个人是不是可以被提升，标准就是：你是不是已经在过去的半年到一年里，按照下一个级别的标准在工作。

换句话说：不是觉得你可以达到下一个级别的标准就提升你，而是你已经达到下一个级别的标准，并在这个水准上稳定地保持了一段时间，才会被提升。

在这种情况下，如果你的上级是前文所介绍的那种静态思维，没有给你机会或项目让你提升和表现自己的能力，只是给你一些与你当前能力和级别相匹配的工作，那么想升级几乎是没有可能的。

那么，作为一位管理者，你应该做哪些思考呢？

1. 怎样做能够让员工进步到更高层次？
2. 他的潜力在哪，哪些地方是可以培养和挖掘的？
3. 怎么帮助员工改进他与组内组外同事的关系，让他有机会更好地发挥他的长处？
4. 怎样尽早地发现他的错误和缺点，并帮助他认识和改进，而不是在错误变成后果后，去追究责任。
5. 怎样帮助他在不擅长的领域建立信心？
6. 怎样帮助他学会处理各种压力和矛盾？

当然了，因为不同的成员性格、背景各不相同，所以帮助其成长的侧重点也不同，但是很多时候，如果能够做到下面四点，你就有很大机率成为一个优秀的管理者。

1. 和自己对话，想想自己哪些时候、哪些方面会用静态的眼光去看待别人的能力。
2. 把自己有这种心态时的表现或内心的一些想法写出来。
3. 再遇到类似的情况，停一下，想一想是不是自己可以有所改变。
4. 诚恳地告诉组员希望帮助他成长，多交流并听取对方的想法。

最后，我说说自己的亲身经历。

我是一个技术出身的管理者。转型初期，我像大多数人一样聚焦在自身和自己的能力上。这就导致一个很大的问题：遇到稍微难一点、复杂一点的事情，我就喜欢亲自上阵，甚至所有代码都要自己看了、写了才放心。

从某种程度上说，这其实是对团队成员的不信任。刚开始我只管四五个人，这种方式还行得通，最多是自己累点儿，而且，组里的人越来越依赖我，我甚至会有一种优越感。

可是慢慢地，管的人和事越来越多，哪怕自己做事再快，24小时不休息，也不可能亲力亲为全部完成了。人们常说“浑身是铁能捻几根钉”，就是这个道理。

后来我和老板谈过很多次，但在潜意识里还是觉得：这件事儿很复杂，交给别人没有自己处理来得高效、来得快、来得稳妥。

这其实就是在用静态思维去看待动态发展的问题。我后来慢慢放手后发现，很多事情，别人可以做得比自己更好，只要你给他机会、给他资源、给他需要的支持和帮助。

一个技术管理者的成功并不在于自己代码多好、能力多强，他的成功一定建立在团队成功的基础之上。只有团队成员不断成长，这个团队才可以做成更大的事情，而你才可以在团队的基础上，站得更高、看得更

远。

如果你是一个管理者，你做过哪些举措帮助自己的团队成员成长呢，而你又得到过哪些上级的帮助呢？你可以在留言里说说。


戳此获取你的专属海报

# 当我们给别人提意见时，要注意些什么？

工作中，相信你一定有听同事展示或汇报产品、项目、报告等工作成果的经历，他们在说完之后，多半会表示有意见尽管提，但是这时候，如果你特别老实，真砸了一堆负面意见过去，换来的很可能是不欢而散。如果你的意见中还包含了很多语气助词，估计还会引发争执。

因此，即使是最好的意见、最客观的表达，如果传递方式不合理，很可能也达不到预期的效果。

提意见，显然是一门艺术。

撇开具体意见是什么不提，你在听到别人的意见时，有没有过这样一些反应。

1. 我知道这个方面我还有待加强，但你也不用老说吧。
2. 为什么要在公开场合提意见，让我下不来台。
3. 就凭你，你连 XXX 都做不好，凭什么对我指手画脚。
4. 有你说得那么严重吗。
5. 我什么时候这样了，我怎么不觉得。

现在试想一下，当你给别人提意见的时候，你的 “好意” 又会不会引起以上的心理反应？

有研究表明，大多数人对于负面的反馈和意见，心理上或多或少都有些排斥感。如果周围有个人总是给你指出毛病，你慢慢就会避免和他合作或交流。虽然这种行为可能只是你的潜意识在作怪。

如果一个人给你的反馈 80% 是正面的，另外 20% 是负面的，人的潜意识会接受这个比例，不会产生 “这个人为什么总是挑我刺儿” 的感觉。但是，一旦超过了这个比例，就很容易产生抵触感。

因此，给别人提意见，需要很小心地去处理。不论对方是你的平级、下级、还是上级。

哪怕对方是鼓励你对他“畅所欲言”，也不要产生 “你小子也有今天” 的感觉，然后毫无克制地把所有自己认为他不够好的地方一股脑儿全抛出来。

这样的效果，远远不如把意见的重点放在一个客观可改进的方面上，并通过合适的方式表达出来。

那么，具体应该注意哪些方面呢？

首先，要思考自己是不是合适提这个意见

也就是说，对方会不会产生 “就凭你” 的逆反心理。

比如，两个级别差不多的同事，一个资格稍微老些，于是常常拿自己做事的方式去 “教育” 另一个人，给他提意见；而偏偏对方并不是很佩服这个人，此时他们就可能会在一些值得商榷的问题上产生争执。

其实工作中类似事件常有发生。如果两个人技术水平明显有高低之分，大部分时候也会和睦相处，因为水平略低的一方比较容易“服气”；如果是两个各方面水平都差不多的人合作，大家都能放平自己心态，一般会相安无事；而偏偏是两者水平相差不多，而好一点的一方又自视甚高，这就容易出现问题。

所以在工作中适当放低姿态，往往更容易获得尊重。不要总是试图居高临下地提意见，尤其是你在对方心里还没有建立任何信任和威信的时候，不要贸然给别人提负面意见。

其次，提意见要讲究合适的时机

这里说的“时机”不是指提意见的场合，而是指你想“给出意见”的问题，是不是一种定式。

通常情况下，如果看到别人初次犯了一个错误，没必要上纲上线。你可以等一等，看他会不会自己改过来，或者稍微提醒一下，让对方注意到这件事，或者委婉地引导对方去关注和思考。如果你发现某个问题已经多次出现，可能就是直接点出来的时候了。

直接给出意见的时候，也要注意一些问题：

1. 地点很重要，尽量找个安静的地方，避开其他人。
2. 可能的话，稍微用几句闲聊来导入正题；
3. 提意见时，语气不应该是责备，而是描述问题；
4. 不要加入太多个人感觉、观点和情绪，避免使用一些像 “你总是”“你从来都” 的评价。

有效讨论需要心平气和，你可以客观地去说观察到或者了解到的问题，比如在什么情况下，对方做了什么，说了什么，为什么你觉得可以改进。

这种讨论不是为了说服或者被说服本身，不是为了争高下，讨论是为了增进对讨论主体的认识，引发思考，寻找可能存在问题的最优解。提意见其实是门聊天的艺术，如何处理，各人要根据具体情况以及和对方的关系来调整。

最后，多夸夸对方做得好的地方，多给予正向反馈

很多时候，正向反馈能更好地帮助对方成长。表扬的力量永远大于批

评，他会试图把事情做得更好。正向的沟通还能帮助你与对方构建良好的沟通渠道，让你们未来的合作更为顺畅和有效。

总结一下，今天主要说了一下给别人提意见的时候，应该注意的地方。

1. 大部分人对于“意见”都有排斥感，需要很小心地去处理，毫无节制的意见远不如把意见的重点放在一个客观可改进的方面更有效果。
2. 提意见时要想清楚自己是不是提这个意见的合适人选。在工作中适当放低姿态，不要总是试图居高临下地提意见，往往更容易获得尊重。
3. 提意见要讲究合适的时机，最好的方式是引导对方自己意识到自己的问题。
4. 直接提意见要找合适的场所。讨论不是为了争高下，应该保持心平气和的心态，找出问题的最优解才是最重要的，要避免使用带有情绪的用词。
5. 多夸夸对方做得好的地方，正向反馈会更好地帮助对方成长。

# 每个工程师都应该了解的：聊聊幂等

什么是幂等（Idempotency）呢？简单来说，一个操作如果多次任意执行所产生的影响，均与一次执行的影响相同，我们就称其为幂等。

这样说来，似乎很容易理解；但要知道这样的定义，其实是一个语义范畴对行为结果的定义。

如何用语法和规则去确保行为能达到这个结果，往往需要很谨慎地设计和实现。实际系统中，幂等是一个极为重要的概念，无论是在大型互联网应用还是企业级架构中，都能见到 REST API 被越来越多地采用，而正确实现幂等，往往是 API 中最难的技术点之一。

先说说为什么重要，我来举一个简单易懂的例子。

比如，你要处理一次电商网站收款或者付款的交易。当你给微信支付发送这个付款请求后，一个顺利的场景是不会有任何错误发生的，微信支付收到你的付款请求，处理所有转账，然后返回一个 HTTP 200 消息表示交易完成。

那如果发出请求后，有个请求超时，你再也没有收到关于这个请求是成功还是失败的回执，又该如何呢？

这里就有很多种可能的情况。

1. 这个请求在到达微信支付端前就已经发生超时，微信支付从来没有收到这样的请求。
2. 这个请求到达微信支付端，但是支付交易失败，这时发生超时，微信支付收到这样的请求，但没有处理成功。
3. 这个请求到达微信支付端，并且支付交易成功，这时发生超时，微信支付收到这样的请求，处理成功，但是没有回执。
4. 这个请求到达微信支付端，并且支付交易成功，并且发回回执，然而因为网络原因回执丢失，客户端超时，微信支付收到这样的请求，处理成功，发出回执，但是客户没有收到。

人们很直观的想法，也是现实中开发者最常见的做法就是：重新提交一次支付请求。但是这样做有一个潜在的问题：请求超时是上面的哪一种情况，会不会引发多次支付的可能性？

这就涉及系统中的幂等是如何实现的了。

那么幂等又该如何实现呢？

首先来看一下幂等的定义：多次执行所产生的影响均与一次执行的影响相同。简言之，你需要一个去重的机制。这往往有很多不同的实现方法，但是有两个很关键的因素。

第一个因素是幂等令牌（Idempotency Key）。客户端和服务器端通过什么方式来识别，这实际上是同一个请求或是同一个请求的多次尝试。这往往需要双方有一个既定的协议，比如账单号或者交易令牌，这种在同一个请求上具备唯一标识的元素，这种元素通常由客户端生成。

第二个因素是确保唯一性（Uniqueness Guarantee）。服务器端用什么机制去确保同一个请求一定不会被处理两次，也就是微信支付如何确保，同一笔交易不会因为客户端发送两次请求就被处理多次。

最常用的做法是利用数据库。比如把幂等令牌所在的数据库表的列作为唯一性索引。这样，当你试图存储两个含有同样令牌的请求时，必定有一个会报错。注意，简单的读检查并不一定成功，因为读与读之间会有竞争条件（Race Condition），因此还是有可能出错。

一个系统能正确处理和实现上面的两个要素，基本就达到了幂等的需求。那么，现实系统中常见的问题都出在哪里呢？

一是幂等令牌什么时候产生，怎样产生。这一点很重要。拿上面的例子来说，就算微信支付可以保证，每一个请求对应的支付交易一定只会被处理一次，但是这个请求的多次重复，一定要共有微信可以识别的某个标识。

假如客户端对同一笔交易多次请求，产生的幂等令牌并不相同，那么无论你其余的地方多么完美，都不可能保证 “一个操作如果具有任意多次执行，所产生的影响均与一次执行的影响相同”。

二是令牌有没有被误删的可能。这是上面问题的一个特殊情况。幂等令牌是由客户端生成的，那如果生成的令牌在被使用后（一次微信支付请求中使用了），不小心因为数据库回滚（DB Rollback）等原因被删除了，那么客户端就不知道自己其实已经发过一次请求。这就有可能生成一个新的账单，并产生全新的令牌，而服务端对此则一无所知。

三是各种竞争条件。我在前面讲过，用数据库的读检查来确保唯一性经常因为竞争而不生效，其实一个需要幂等的系统中，保证唯一性的各个环节和实现，都要考虑竞争条件（Race Condition）。

四是对请求重试的处理。这大部分是服务器端要做的工作。一个常见的方法是：区分正在处理的请求、处理成功和处理失败的请求。这样当客户端重试的时候，根据情况或者直接返回，或者再次处理。这就好像之前提到的微信支付的例子，微信支付服务需要知道每一笔交易的处理情况，只有这样，当面对再次转账请求时，才能知道应该用什么方式去处理相应的请求。

五是一个系统中需要多层幂等。这是什么意思呢？A 发送请求给 B，B 处理的一部分是要发送请求给另一个系统 C，C 在处理的过程中还可能需要发请求给另一个系统 D …… D 处理完了返回给 C，C 返回给 B，B 返回给 A。在这个链条中，如果 A、B、C、D 中任何一个系统没有正确实现幂等，也就是出现了 “幂等漏洞”，那么一次请求还是有可能被多次执行，产生区别于一次执行的影响。

今天我和你讨论了架构设计中的幂等概念。我们聊了什么是幂等，幂等的应用场景，如何实现一个幂等功能，以及幂等系统中容易出现的问题。

现在回到文章的开头，什么是幂等？一个操作如果任意多次执行所产生的影响均与一次执行的影响相同，我们就称为幂等。这是一个语义范畴上对行为结果的定义，只有当你把实现中所有的细节都做对了，你才能得到想要的效果。任何一个地方设计有漏洞，或是实现上有 Bug，系统都会出现这样或那样的问题。

# 当别人给我们提意见时，该如何应对？

在“当我们给别人提意见时，要注意些什么”这篇文章中，我与你一起关注了提意见的技巧。

文章的主要内容是人们很难在接受负面意见时保持心情愉悦，有时候甚至会引起心理上的反感。

因此，我们在给别人提意见的时候，首先，要考虑自己是不是提这个意见的最优人选；其次，要确定什么时候应该提什么样的意见；最后，要慎重考虑提意见的时候要注意的地方。

然而，在现实生活中，你接收到的所有意见，并不是都以一种愉快的方式不期而至；给你提出意见的，也并不都是你所尊敬的人。对于肯定自己的内容，每个人大概都知道如何处理。那么，当收到那些让我们觉得不太舒服的意见时，应该怎么办呢？

首先，要了解自己内心，清楚地认识到自己对于意见一般反应是什么。

1. 是不是很看重提意见的人，如果是自己不服的人，是不是根本听不进去。
2. 是不是容易把意见个人化，容易想太多，觉得对方是不是对自己不满。
3. 会不会被当时的心情、别人的语气和态度等因素左右。
4. 会不会当时经常不知道如何应对，但是回头静下心一想，又觉得这些意见不够公正，懊恼为什么自己没能及时辩驳。
5. 当时态度很强硬，甚至当场否定对方的观点。但是细想一下，觉得自己确实可能有这个问题。

很多时候，情感上的抵触是我们要克服的第一步。知道自己可能的情绪化反应，尽可能在下次听到意见的时候，下意识地要求自己去做一些改变。从某种程度上讲，这就是一种内省机制。

克服了情绪上的障碍，还有一些方面是我们需要注意的。

不论对方出于什么目的给我们提出意见，我们都不要试图从恶的一面去揣测对方的动机，而是先假设对方是善意的，并且对主动提意见的人表示感谢。

做到这一点，并不意味着我们需要对外界所有的反馈兼容并包，而是在假定对方是好意的前提下，我们可以进一步对意见进行归类和筛选。

如果是陈旧型意见，这一类意见包含了其他人已经提出的意见或自己意识到的问题。对方的意见并没有给出任何新的信息，而你已经有计划或行动在改进这个问题，不要因为对方再次提出而不满，先诚恳接收，在实际处理中忽略就是。

如果是崭新型意见，这一类意见包含了从来没有听到过的新意见，有新观点或新例证的已有问题，以及那些你以为自己已经改进而别人依然觉得有问题的地方。这时，你就需要去想一下，是因为提意见的人观点过于片面，还是时过境迁问题早就不存在，或者是自己做得真的不够。

即便你有充足的理由认为对方说的不是实际情况，但如果有不止一个人提出过意见，或者你比较看重提意见的人，这个人可能是你的上级，或者一个德高望重、谨慎靠谱的队友——那你可能就要去思考了，虽然这件事我觉得自己没有做错或不合适的地方，为什么会给别人留下“我做得不够好”的感觉呢；然后继续去思考，做什么样的改进可以让事情变得更好，可以改变别人的感知。

我之前负责过一个项目，团队大概由5、6个工程师组成，项目的内容是重写系统的一部分。项目进行了好几个月，大家都做得很认真很卖力；但是，由于项目的工作量和影响范围在开始的阶段被低估了，在项目发布的过程中，老系统遗留下来的各种坑还是引发了一些问题。

这些问题不是很大，而且很快被解决了；然而，对项目完全没有了解的人却很容易因此产生负面的判断，人们会质疑执行者是不是在认真对待工作，会提出项目是不是按照软件工程的最佳实践去做等一系列问题。

对于这些问题，我们一般都有自己的解释。但是，当很多人同时来怀疑一件事情的时候，也许这件事确实有可以改进的地方。

于是，我们让所有项目相关或感兴趣的人员把他们质疑的原因和发现的问题都列出来。收集好这些问题以后，我们约了个会议，所有人针对所有问题逐一进行讨论。

一方面，项目组的人可以反思哪些地方可以做得更好，哪些流程可以改进，哪些技术可以重构；另一方面，也可以让项目组之外的人了解到项目的各种难点和复杂度，对项目参与者的工作也多了一分认可。

经过这次会议之后，所有人不仅得到了工作上的提升，还增进了相互之间的理解与信任。

今天，我讲到了当听取别人意见的时候，我们应该注意的地方。总结一下，我认为听取意见时有几点很重要。

1. 尽量排除情绪以及偏见对我们的干扰，这一点，需要我们下意识地去克服。
2. 假定对方是善意的。尽可能以感激的心态去听取意见，尽管这一点很难，但是我们还是要尽力为之。

3. 了解意见的具体细节，把关注点放在对方的出发点，以及对方希望我们改进的地方。
4. 对意见里的信息进行分类和过滤，思考一下：哪些是误解，哪些是自己的不足，哪些是自己可以通过沟通去改变的，哪些是自己需要尝试改进的。再三确定后，如果确实不是自己的问题，也不用反感，一笑而过就好。


戳此获取你的专属海报

# 说说硅谷公司中的一对一沟通

互联网时代，人们发明了各种各样的即时交流平台。微信、钉钉、WhatsApp、Telegram、Slack等等，这些软件占据了人们大量时间，即使他们相隔大洋，各自在水一方，依然可以通过这些工具进行高效的交流。

但是，远程交流永远没法替代面对面的沟通，对方在微信里扔一个笑脸，你不会知道他是真的在微笑，还是在哭泣。你无法捕捉到对方的表情和肢体语言，也很难做最深入的沟通，这正是面对面沟通的魅力所在。

今天，我就和你聊聊硅谷公司里的一对一沟通。

在《格鲁夫给经理人的第一课》一书中，格鲁夫对一对一沟通的定义是这样的：

> 在英特尔，一对一会议通常是由经理人召集他的部属召开的，这也是维系双方从属关系最主要的方法；一对一会议主要的目的在于互通信息以及彼此学习；经过对特定事项的讨论，上司可以将其技能以及经验传授给下属，并同时建议他切入问题的方式；而下属也能对工作中碰到的问题进行汇报。

格鲁夫在写这本书的时候表示：“据我所知，其他公司很少定期举行一对一会议。”时至今日，硅谷很多公司的管理者，都开始花大量的时间与团队成员定期进行一对一沟通。也就是说，每两周会有一次到两次的一对一面谈。

经理人会和每位团队成员进行一次大约二三十分钟的跟进谈话，这个行为差不多成了硅谷所有公司的传统。当你管理的团队逐渐变大时，一对一的员工沟通会成为你每周工作中很重要的一部分。

谈话都有哪些内容呢？

谈话的内容其实没有定式。一般会谈到最近的项目进展如何，有没有什么阻力，最近的生活和家庭状况如何，有没有出游或其他特别的计划等等。

对话大体上是关于一个话题的纯粹讨论，这个话题应该是团队成员感兴趣的，这一点非常重要。因为一对一的沟通通常以下级为核心，由他来掌控谈话的内容和气氛，话题应该是他想和你沟通的内容。

相对正式的内容一般都与工作相关，包括项目进展、工作中的挑战、技术、业务、人事关系，大家甚至可以针对一个技术细节进行详细探讨和

沟通。

除此之外，还会有那些内容呢？比如，讨论一些不成熟的想法，精彩的创意，倾诉工作或生活中的焦虑，甚至抱怨。无论是工作中的、生活中的，还是情绪上的问题，都可以通过一对一的沟通进行缓解，甚至直接解决。

虽然每周不一定都有很多事值得讨论，但是保留这样一个畅通的交流渠道显然益处多多。我成为管理者之后，各种会议比以前多了很多，但这种一对一沟通的会议，对我和团队成员都非常重要。

如何让沟通更有效？

如果想让一对一沟通真的有益、有效，管理者应该学会聆听。通常说来，成为管理者，大家会更注重你的表达能力。

作为领导，你需要时时传达各种信息，比如公司决策、企业文化、项目信息、协作信息、产品信息等，你需要通过自己的表述让全组人员对某个决定或某个项目建立一致的认识，然后才能形成合力，让正确的事情持续发生。正因为如此，管理者的主动聆听能力往往会被别人甚至自己忽略。

领导者很多时候在综合层面 —— 比如经验、知识、信息、能力等 —— 都属于佼佼者，大家往往或多或少都有种“自己什么都知道”的心态，至少是什么都知道一点，所以对别人 —— 尤其是自己团队成员的一些意见 —— 不够有耐心。

因此，在领导者与员工的交流中，经常会产生一个方向上的信息流大于另一个方向的信息流这种状况。

现在换位思考一下，如果和上级进行一对一的交流，你会有什么期望呢？

我曾经共事过很多位老板，能想到的他们做得好的地方，或者我希望他们能做好的地方包括这样几个方面。

1. 找一个适合谈话的场所。场所不一定非要特别安静，但是可以轻松听到对方的话。除此之外，场所还要保证隐蔽性和安全感，周围不要有其他人能够听到对话内容。
2. 彼此百分百的专注。不会有人随时来打断，也不会有社交信息从手机上冒出来。如果轻易被打断，我会觉得自己分享的信息或感受没有价值。
3. 要有一定程度的眼神交流。有眼神和肢体语言表示对方在认真倾听和思考。
4. 不会收到批判性的反馈，哪怕是表情或者肢体语言的暗示（例如不以为然地耸肩等）。
5. 有一定程度上的交互。自己说的话，对方认真在听、在思考、有回

应。对方在不清楚的情况下会试图让你解释，或者总结他理解的意思，并请你确认。
6. 所有的对话，如果自己有保密要求，可以保证对话内容不会被传播出去。

后来，我自己成为了技术领导，在与员工一对一交流的过程中，也会注意去模仿这些好的行为，尽管有的地方我做得还是不够好，比如听到一些我已经从别的渠道知道的信息时，可能会打断谈话者。

还有一点需要注意，就是：对方想从这个谈话中获取什么，是单纯希望上级知道，还是想要我给出意见；为什么这件事看来和我无关，但是对方会主动跟我说。

不同的情况需要不同的应对方式。如果对方期待我的反馈，而我当时又没有好的想法，我可能会坦诚相告，然后和他承诺，后续我有了更多信息时会跟进这个问题。

另外，如果下属提出了很好的工作建议和创意想法，应该马上动笔记下来。一方面这些内容可能真的非常宝贵，另一方面，“记下来”这个动作其实是一种承诺，就像双方达成了一种契约，让后续可以进行跟踪和持续推进，甚至可以在下一次沟通中继续讨论这个问题。

沟通的杠杆

如果你有十个下属，每人每周沟通半个小时，你一周要花掉五个小时。这时候就会有人问了：“这五个小时是不是被浪费掉了，如果用来写代码或者讨论产品是否会有更大的产出呢？”

在这里，格鲁夫引入了杠杆率的概念。事实上，你每周在团队成员上花费的三十分钟，可能会提升他接下来一周甚至两周的工作品质。同时，通过信息分享，你也可以学习和了解到更多的东西。长远来看，这些贡献是远远超过自己那五个小时的。

对于一个技术领导来说，团队的成功，才是真正的成功。

其实今天讲的一对一沟通就是人与人之间的交流，只不过场景、角色和目标与平时闲聊有所区别，我们希望这样的交流更高效。

我想，做好一个技术领导，最重要的是你真的在乎团队里的每一个成员。只有真的在乎，才会去聆听；而一对一沟通，就是你和这个团队连接中最重要的环节之一。


戳此获取你的专属海报

# 每个工程师都应该了解的：大数据时代的算法

开了这个专栏之后，经常有读者问算法相关的问题。

> 能不能讲讲算法在工作中的运用？你个人学习算法的过程是怎样的？我对算法还是有点怕。除此之外，你认为大学是应该多花时间学应用技术还是理论知识呢？谢谢。

今天就来聊聊我自己学习算法的过程，以及算法在实际工作中的应用。

以前，我们认为大数据总是优于好算法。也就是说，只要数据量足够大，即使算法没有那么好，也会产生好的结果。

前一阵子“极客时间” App 发布了一条极客新闻：“算法比数据更重要，AlphaGo Zero 完胜旧版。”新闻的内容是谷歌人工智能团队 DeepMind 发布了新版的 AlphaGo 计算机程序，名为 AlphaGo Zero。这款软件能够从空白状态开始，不需要人类输入任何命令，便可以迅速自学围棋，并以 100 比 0 的战绩击败了上一代 AlphaGo。

AlphaGo Zero 最大的突破在于实现了“白板理论”。白板理论认为：婴儿是一块白板，可以通过后天学习和训练来提高智力。AI的先驱图灵认为，只要能用机器制造一个类似于小孩的AI，然后加以训练，就能得到一个近似成人智力，甚至超越人类智力的 AI。

自学成才的 AlphaGo Zero 正是实现了这一理论。AlphaGo 的首席研究员大卫·席尔瓦（David Silver）认为，从 AlphaGo Zero 中可以发现，算法比所谓的计算或数据量更为重要。事实上，AlphaGo Zero 使用的计算要比过去的版本少一个数量级，但是因为使用了更多原理和算法，它的性能反而更加强大。

由此可见，在大数据时代，算法的重要性日渐明晰。一个合格的程序员，必须掌握算法。

我不知道大家是怎样一步步开始精通算法和数据结构的。大二时，我第一次接触到了《数据结构》，因为从来没有过这方面的思维训练，当时的我学习这门课比较费力。那时候接触到的编程比较少，所以并没有很多实际经验让我欣赏和体味：一个好的数据结构和算法设计到底“美”在哪里。

开始学习的时候，我甚至有点死记硬背的感觉，我并不知道“如果不这样设计”，实际上会出现哪些问题。各种时间和空间复杂度对我而言，

也仅仅是一些不能融入到实际问题的数学游戏。至于“每种最坏情况、平均情况的时间空间复杂度与各种排序”，这些内容为什么那么重要，当时我想，可能因为考试会考吧。

没想到后来的时日，我又与算法重新结缘。可能是因为莱斯大学给的奖学金太高了，所以每个研究生需要无偿当五个学期的助教 。好巧不巧，我又被算法老师两次挑中当助教。所以，在命运强制下，一本《算法导论》就这样被我前前后后仔细学习了不下四遍。这样的结果是，我基本做过整本书的习题，有些还不止做了一遍。我学习算法的过程，就是反复阅读《算法导论》的过程。

那么，学习算法到底有什么用处呢？

# 首先，算法是面试的敲门砖

国内的情况我不太清楚，但就硅谷的 IT 公司而言，不但电话面试偏算法，现场面试至少有两轮都是考算法和编程的。

大一些老一些的公司，像谷歌、Facebook、领英、Dropbox等，都是直接在白板上写程序。小一些新一些的公司，如 Square、Airbnb等，都是需要现场上机写出可运行的程序。Twitter、Uber 等公司则是白板上机兼备，视情况而定。

虽说还有其它考系统设计等部分，但如果算法没有打好基础，第一关就很难过，而且算法要熟悉到能够现场短时间内写出正解，所以很多人准备面试前都需要刷题。

有一次我当面试官，电话面试另外一个人，当时是用 Codepad 共享的方式，让对方写一个可运行的正则表达式解析器。45 分钟过去了，对方并没有写出来。我就例行公事地问：“你还有什么问题想问或者想了解么？” 对方估计因为写不出程序很有挫败感，就反问：“你们平时工作难道就是天天写正则表达式的解析器么？”

一瞬间，我竟无言以对。想了想，我回复说：“不用天天写。那我再给你 15 分钟，你证明给我看你还会什么，或者有什么理由让我给你进一步面试的机会？” 对方想了一会，默默挂掉了电话。

老实说，我对目前面试中偏重算法的程度是持保留意见的。算法题答得好，并不能说明你有多牛。牛人也有因为不愿刷题而马失前蹄的时候。但是除了算法测试，显然也没有更好的方法佐证候选人的实力；然而怎样才能最优化面试流程，这也是个讨论起来没完的话题，并且每次讨论必定无果而终。

# 其次，编程时用到的更多是算法思想，而不是写具体的算法

说到实际工作中真正需要使用算法的机会，让我想一想 —— 这个范围应该在 10% 的附近游走。

有些朋友在工作中遇到算法场景多些，有的少些。更多的时候，是对业务逻辑的理解，对程序语言各种特性的熟练使用，对代码风格和模式的把握，各种同步异步的处理，包括代码测试、系统部署是否正规化等等。需要设计甚至实现一个算法的机会确实很少，即使用到，现学可能都来得及。

但是熟悉基本算法的好处在于：如果工作需要读的一段代码中包含一些基本算法思想，你会比不懂算法的人理解代码含义更快。读到一段烂代码，你知道为什么烂，烂在哪，怎么去优化。

当真的需要在程序中设计算法的时候，熟悉算法的你会给出一个更为完备的方案，对程序中出现的算法或比较复杂的时间复杂度问题你会更有敏感性。熟悉算法你还可以成为一个更优秀的面试官，可以和别的工程师聊天时候不被鄙视。

# 最后，不精通算法的工程师永远不是好工程师

当然，除了算法导论中那些已成为经典的基本算法以及算法思想（Divide-and-conquer，Dynamic programming）等，其实我们每天接触到的各种技术中，算法无处不在。

就拿人人都会接触的存储为例吧，各种不同的数据库或者键值存储的实现，就会涉及各种分片（Sharding）算法、缓存失败（Cache Invalidation）算法、 锁定（Locking）算法，包括各种容错算法（多复制的同步算法）。 虽然说平时不太会去写这些算法 —— 除非你恰恰是做数据库实现的 —— 但是真正做到了解这项技术的算法细节和实现细节，无论对于技术选型还是对自己程序的整体性能评估都是至关重要的。

举个例子，当你在系统里需要一个键值存储方案的时候，面对可供选择的各种备选方案，到底应该选择哪一种呢？

永远没有一种方案在所有方面都是最佳的。就拿 Facebook 开源的 RocksDB 来说吧。了解它历史的人都知道，RocksDB 是构建在 LevelDB 之上的，可以在多 CPU服务器上高效运行的一种键值存储。而 LevelDB 又是基于谷歌的 BigTable 数据库系统概念设计的。

早在 2004 年，谷歌开始开发 BigTable，其代码大量的依赖谷歌内部的代码库，虽然 BigTable 很牛，却因此无法开源。2011 年，谷歌的杰夫·迪恩和桑杰·格玛沃尔特开始基于 BigTable 的思想，重新开发一个开源的类似系统，并保证做到不用任何谷歌的代码库，于是就有了 LevelDB。这样一个键值存储的实现也用在了谷歌浏览器的 IndexedDB 中，对于谷歌浏览器的开源也提供了一定的支持。

我曾经在文章中提到过 CockroachDB，其实又可以看作是基于 RocksDB 之上的一个分布式实现。从另一个层面上讲，CockroachDB 又可以说是 Spanner 的一个开源实现。知道这些，就知道这些数据库或键值存储其实都同出一系。再来看看 LevelDB 底层的 SSTable 算法，就知道他们都是针对高吞吐量（high throughput），顺序读/写工作负载（sequential read/write workloads）有效的存储系统。

当然，一个系统里除了最基本的算法，很多的实现细节和系统架构都会对性能及应用有很大的影响。然而，对算法本身的理解和把握，永远是深入了解系统不可或缺的一环。

类似的例子还有很多，比如日志分析、打车软件的调度算法。

拿我比较熟悉的支付领域来说吧，比如信用卡BIN参数的压缩，从服务端到移动 App 的数据传输，为了让传输数据足够小，需要对数据进行压缩编码。

每个国家，比如中国、韩国、墨西哥信用卡前缀格式都不一样，如何尽量压缩同时又不会太复杂，以至于影响移动 App 端的代码复杂度，甚至形成 Bug 等，也需要对各种相关算法有详尽地了解，才有可能做出最优的方案。

关于算法我们来总结一下：

1. 在大数据时代，数据和算法都同等重要，甚至算法比计算能力或数据量更为重要。
2. 如何学习算法呢？读经典著作、做题，然后在实践中阅读和使用算法。
3. 算法是面试的敲门砖，可以帮助你得到一份自己喜欢的工作。
4. 写程序中用到的更多是算法思想，不是写具体的算法。
5. 不精通算法的工程师永远不会是一个优秀的工程师，只有对各种相关算法有详尽理解，才有可能做出最优的方案。

希望每个读者都成为合格甚至优秀的软件工程师，如果你在工作中遇到过有趣的算法故事，也可以在留言中告诉我。

参考外链：

https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

# 项目延期了，作为负责人该怎么办？

关于这个话题，我们先来探讨一下，为什么项目会延期。

只要你参加软件开发项目，无论是作为新人，还是作为带新人的老手，或多或少都会经历过项目延期。即使你工作很多年，成了行家里手，做得都是详细的项目计划和估算，但在项目进展的过程中，总会出现一些偏差和意外。偶尔有幸运的时刻会提前结束，延期完成，甚至是项目烂尾也是很有可能的。

为什么项目这么容易延期呢？

这是因为人们在历史长河中累积的经验失效了。

到目前为止，计算机科学只出现了短短几十年的时间，人类很难按照以前的经验对项目进行判断和预测。看到一个人在操场上慢跑，你很容易能判断出来，这事儿不难，因为你也可以去操场慢跑。看到一位百米选手跑进了10秒，你就会知道，这事你拼了老命也做不到，难易立判。

但是一到了编程的范畴，你甚至很难从敲击键盘的速度上分辨出哪个是优秀的程序员，哪个是打字速记员。是的，搞速记的人，敲键盘的技巧可能会更加纯熟一些。

同样，我们可以从物体形态的大小、结构的复杂程度上作出判断。

我们一定知道，修建一座小桥的难度和工期会远远小于一幢摩天大厦。因为物理上形态的区别和感官上的认知，总能让我们的大脑去做出正确的判断。我们很少失误，几千年来，我们的基因就是这么告诉我们的。

但是，软件没有物理的概念，它既没有体积，也没有面积，更没有速度。你写了两万行代码，你的硬盘也不会重一点，10M 的代码，也不一定会比 1M 的代码好用。

技术项目更是很难简单从表面获知其复杂程度。打开 Google 的官方网站，我们看到的是个搜索框，这个简洁的搜索界面，就是整个 Google 帝国的入口，是浮出海面的冰山一角，隐藏在下面的则是数以万计的工程师和庞大的服务器集群。

即使你是经验丰富的工程师，有时候也很难从一纸需求上确定较为准确的工期。

那一旦项目延期了，该怎么办呢？

先问自己几个问题：为什么项目延期了，你是什么时候第一次感觉到项

目可能会延期，在此之后你做了什么？

有些时候，项目延期是因为一些不可控因素，比如产品上加了特性，原有的需求变更了 —— 当然，这是经常发生的事；比如人员减少了，或者预计增加的人员没有到位；比如项目组的人员被拉去做另一个项目同时还要兼顾原有项目的任务。

有的时候，项目的延期，是因为负责人没有做好计划，或者这个计划没有很好地分享给每一个对项目进度有影响的人，也就是项目干系人。又或者，当偏离计划的事情发生时，你没有在第一时间做出好的应对措施。

对于不可控因素，我们能做的并不多，只能权衡取舍，对项目期限、人员或者需求范围做出调整。

如果是计划本身的问题，或者是因为项目成员没有很好地跟进同一个计划并保持一致，那我们应该尽最大可能避免这些问题的发生。

一旦发现项目延期了，应该有哪些具体的应对措施呢？

怎样才能做到这一点呢？下面的几个策略是我在工作中经常使用的，对你也许会有帮助。

1. 建立一定的流程。这里包括计划制定流程和计划跟进流程，也许是每周一次的同步会议，也许是一个共享的任务管理工具。不要过分相信自己随机跟进的能力，哪里出了问题就去处理问题，很多时候会陷入救火队员的困惑，手忙脚乱，有的时候还会错过最好的时机。
2. 在整个项目计划中，要有明确的优先级。知道哪些任务是非做不可的；哪些任务是别的任务所依赖，需要提前完成的等等。人们往往有先处理简单任务的惰性，如果没有搞清楚轻重缓急，那就有可能出现不太重要的杂事都做完了，而重要的事却没有太大进展的情况。
3. 制作一个共享的项目状态表，让团队成员可以一眼就看清楚项目进展，并保持该图表的更新。进度表可以作为大家对进度评估的工具，也可以是负责人判断哪里需要重点跟进的依据。确保这个计划是每个主要相关者都确认并同意的，并在同意后让他们每天都能看到这个计划的所有更新。
4. 不要漏掉任何一个人。不要觉得暂时还没有他的活儿，可以先不用跟他沟通。当一个项目的计划落实后，应该第一时间让所有相关的人了解到。临时通知会让合作者或帮忙的人没有参与感，并且很容易因为对方没有提前准备在时间上无法达成一致。
5. 提供一个有效的反馈渠道。任何人在任何时候对项目有担心或者质疑，确保他可以通过有效的途径让你第一时间知道他的担心。

如果能做到这些，每个人都会有参与感和使命感。一旦出现项目延期的情况，我们就能非常清晰地知道是哪个环节出了问题，能迅速地意识并了解这个延迟对整个项目产生的影响和后果，并在第一时间调整需求、计划、人员，重新让项目回到正确的轨道上。

如果做了所有能做的，项目还是延期了 —— 这种情况时有发生 —— 我们需要尽早和上级沟通，让他们了解为什么延迟，自己做了哪些努力，还需要哪些资源或者做出什么样的调整才有可能赶上预计期限，是不是一定会延期等等。

当你的沟通内容中有了这些详细的描述，相信上层能够体谅你和当时的项目状况，并给予你最大的帮助和支持。

互联网时代瞬息万变，不变的永远是变化，只要我们参与软件研发，总会遇到项目延期的情况。

今天我和你讨论了项目为什么会延期，如何避免项目延期，一旦发现项目延期了，应该有哪些具体的应对措施，遇到了这种情况怎么办？

建立流程、划分优先级、同步进度、实时反馈。采用了今天技术管理课中讨论的方法，你很可能会避免或挽救一次延期的灾难。如果没有，相信也可以帮你总结经验，避免同样的延迟发生。

如果你有更好的建议或想法，也欢迎给我留言，我们一起成长。

# 管理和被管理：期望值差异

在我们的职场生涯中，管理者与被管理者之间会存在各种各样的关系。我觉得其中最需要避免的情况之一，就是期望值的差异。

什么意思呢？

比如，你一直觉得自己很努力，做出的成绩也比较稳定，平时从老板那里得到的反馈也是比较积极肯定的。然而，就在效绩评估的时候，你觉得自己明明应该是优异，却只拿到一个合格；或者，你觉得你应该能升职加薪，结果这样的好事却落到了其他同事的头上。

又或者，你一直觉得你和你团队里的成员合作很愉快；然而某一天你突然得到消息，有的组员要换组，甚至跳槽；种种情况表明，原因很可能是对你有不满，或者对你的管理不满意。

不论是哪种情况，那个觉得意外的一方，或者是没有拿到好评的组员，或者是被认为不合格的管理者，都会感到委屈甚至忿忿不平：“我明明做得很好，他平时也说我做得不错，为什么事情最终会变成这样？”这种感觉就像你正在阳光和雨露中歌唱，突然来了一场瓢泼大雨，心里拔凉拔凉的。

这就是我今天想和大家讨论的话题：管理者和被管理者之间的期望值差异。

文章的重点会放在管理者身上，如果你是一个技术领导，如何去避免这样的期望值差异呢？

从我们（我个人和一些做技术领导的朋友）遇到的小样本数据来看，期望值差异很多时候都是一种很自然的存在。原因也很简单，因为每个人都会或多或少高估自己。

90%的司机都相信自己的车技水平是高于平均值的，从统计学角度来说，这个调查结果显然是存在认知偏差的，但人们就这么认为。我们常常只重视自己已知的，却忽略自己不知道的东西，我们关注自己想做的和能做的，却忽视他人的计划和环境。

我们在制定计划和展望未来的时候，会强调技能的因果作用，却忽视了运气、环境、协作和他人对自己的影响。90%甚至以上的创业公司都会死掉，但这些创业者在初期无一不认为：“我有优秀的团队、广大的市场、敏锐的产品意识和绝妙的创意，怎么可能不成功呢”？

职场中当然也是这样，每个人或多或少都会对自己的某个方面自视过高。如果这个自我评价和来自他人的评价不符，并且从来没有沟通和校准过，那么在某个特定的时候，这种问题就有可能发生，从而伤害彼此

的关系。

在平时，指出他人某些方面不足是需要慎之又慎的，谁愿意说别人不好呢。因此，很多人，尤其是在一些自驱大于管理的团队中，会选择尽可能婉转地暗示，甚至直接忽略对方的瑕疵。

那么，要怎么做才能尽可能地校准管理者和被管理者对彼此的期望值呢？

首先，最重要的一点是增加彼此的了解。知道对方是内向还是外向；平时对负反馈的接受程度如何；是偏自负还是自信心不足；是喜欢自己跟自己较劲还是爱和别人攀比等等，这些你可以通过平时合作中的接触和交流来慢慢了解。只有对这个人有了深入了解，才能在交流中采用更好的方式方法，保证自己的良好意愿被准确传达。

其次，每半年或一年进行一次关于期望值的深度对话。对话需要尽可能地了解对方的兴趣是什么；加入这个团队希望做什么，未来两三年的职场计划如何；对方做了哪些努力并正在做哪些努力；除了目前在做的工作，还想做什么，还能做什么，想承担什么样的责任。在获得了这些信息之后，进一步了解对方希望你，也就是他的上级，可以提供哪些帮助。

接下来，就是澄清和约定。明确地告诉对方，如果他想达到自己的期望值，同样需要证明自己。然后告诉他你对他的期望，包括对他当前项目的期望值，以及未来的期望值，并表示自己会尽最大可能提供机会和支持。这里需要明确的是，如果对方以前承诺的目标还没有达到，那就暂时不能把更有挑战的机会留给他。

最后，也是最重要的，有了这样的约定，还需要持续跟进。设置一些检查点，一周或两周检查一次，确保对方不会因为某个地方没有得到充分的支持而停滞不前。如果有任何客观因素导致一方之前的承诺没有兑现，双方要在第一时间保持一致的判断，并做出调整。

由于所有的约定都是在初始阶段达成一致的，在这种情况下，后续有任何正反馈或负反馈，这些反馈都会更为客观并充满正向激励，双方的交流也会变得更加容易。

最后给你留一个思考题，你是一个技术团队的领导，你团队里的一个骨干工程师突然提出想去做其他领域的工作，原因是觉得现在的任务对自己太没有挑战了，而他的工作目前非常重要，别人接手会比较困难，还有可能造成项目延期。你该怎么做呢？

请在留言中写下你的思考，我们一起成长。


戳此获取你的专属海报

# 每个工程师都应该了解的：数据库知识

数据库和编程语言一样，同样是软件工程师们的必争之地。今天我就和大家聊一些数据库相关的知识。

去年 Uber 发表了一篇文章，宣布他们从 PostgreSQL 转到 MySQL 了。文章的内容很好，同时还科普了一些数据库索引和复制的基本常识。当时，我转给了一个朋友，朋友看了之后说：“哦，他们 2013 年才发了一篇文章说他们从 MySQL 转到 PostgreSQL 。”

我找来朋友提到的那篇旧文，读过之后，大概理解了两篇转型文章背后的原因。

作为两大主流开源数据库，MySQL 和 PostgreSQL 的 “战争” 从未停止，虽然硝烟不如编程语言那么浓烈，但也是你来我往，剑影刀光。

如果去 Quora 或者 Stack Overflow 上搜索 MySQL V.S. PostgreSQL 这样的关键字，会出现一大堆帖子，大家各执一词众说纷纭，我的感觉则是：两者各有各的优势和使用场景，并不存在一种数据库对另一种压倒性优势。

对于大部分程序员来说，公司用哪个数据库，基本无需你去决定。加入一个公司的时候，除非是创业公司，或者你是 CTO、VP、总监级别的，否则大部分的技术选型早已应该尘埃落定。

尤其是数据库，一旦选择，再迁移的代价非常大。因此，除非有颠覆性的优势或者难以克服的问题，很少有公司会去费时费力做这种大的迁移。

不论是技术选型还是技术转型，其中不可忽略的因素是：你的工程师更容易驾驭哪一种技术，或者有话语权的决策者们倾向于哪一种技术。这一点其实和程序语言的选型有异曲同工之处。

类似 Uber 两次高调转型的事情，在我曾经工作过的 Square 公司也发生过。

Square 最早使用的是 MySQL，到了 2012 年，由于 PostgreSQL 的各种优势越来越突出 —— 比如对地理空间（Geospatial）数据和搜索的支持，当时几位资深工程师也开始大力倡导，很多新的服务就尝试性地使用 PostgreSQL。

那时候，公司的架构是 MySQL 和 PostgreSQL 并存的。对于我而言，这就有机会学习、掌握和比较两种不同的技术。

在我来看，两者各有特点，有些技术实现在 MySQL 里更方便一点，另一些则反之。无论哪种数据存储方式，总有方案可以解决问题，并没有觉得非要哪一种才行。

一个公司如果数据库从来不出问题，那一定是因为没有业务量或者流量。所有技术的选型和设计，都有它的应用场景，除去那些让人开心的案例，剩下的毫无疑问就是坑。

如何尽可能地避开这些坑，如何在出现问题的时候可以用最快的速度去修复，这些都是至关重要的因素。

大部分工程师并不是数据库专家，在 Square 公司两种数据库并存的期间，PostgreSQL 的牛人寥寥无几，但是 MySQL 却有几个专家是极为靠谱的。对我们而言 ，PostgreSQL 和 MySQL 的相对优势，都比不过出问题的时候有人救火和解惑来的重要。

另外，一个公司维系两套同类的数据库系统本身就是个负担，因为这些原因，那些使用 PostgreSQL 的服务慢慢就转成了 MySQL 。

我们支付类的技术工作需要强事务和一致性的支持，所以 NoSQL 类型的数据库用得比较少，主要使用的是 MySQL 或者 PostgreSQL。由于在工作中常常与数据库打交道，我也逐渐了解到不少相关的知识和技术，但是线上真的出了问题，我还是没有把握自己去搞定。

好在，每个公司都会有一些专攻数据库的大牛，这种专门的职位便是 DBA，有的中小创业公司没有专职 DBA，数据库便由做运维的人维护。我和几位这样的牛人私交甚好，加之平时自己处理起系统中出现的相关问题，也会常常请教他们，一来二去就知道了很多有趣的数据库知识。

对数据库专家我一向是敬佩的态度，他们的价值不可小觑。公司只要稍具规模，如果数据库这块做不好，基本也就没什么可玩的了。数据库可以说是互联网公司最宝贵的资产，这块不出问题也罢，一出问题，即是见血封喉，服务直接宕机。

# 关于数据库，最常见的问题都有哪些呢？

## 第一个是选型

因为每个公司的业务不同，数据库系统的应用场景不一样，选型也会不尽相同，但可以肯定的是，没有哪个系统一定是最好的。

比如做支付业务一定要强事务性、一致性的支持，很多社交平台更多时候需要的是高可用；有的业务写操作特别重，有的业务更重要的是读操作；有的业务可能只关心最近几天的数据，于是可以容忍老数据读写的低效，有的却要频频访问历史数据，需要读写的高效；有的业务可以通过加索引解决查询效率，有的却只能通过加缓存等等。

这就是为什么很多公司会选择多个数据库系统并存，通过不同的技术和架构给予相关的业务场景最优支持。如果一旦选型失误，便不会有频频踩坑一说，因为这基本就算直接掉进了坑里。

# 第二个是数据库相关的架构

什么意思呢？这里的架构包括数据库上层的缓存系统设计，程序语言对数据库连接的处理，代理层（Proxy Layer）的功能，以及和二进制日志（Binlog）等相关的数据管道（Data Pipeline）的搭建。当然，其中也包括了数据库系统的分区、备份等的具体设计。

很多公司早期所有的表都在一个数据库里面，因为各种连接池（Connection Pool）和吞吐量（Throughput）的限制，基本没法做扩展。能够合理地设计数据库表的分离，把数据相关的放在一起，不那么相关或甚至不相关的放在另一个数据库里。这些看起来很简单的做法，很多时候却可以很大程度上缓解可扩展的问题。

# 第三个，也是我们平时遇到最多问题的：人为错误

再好的系统，使用姿势不对也是枉然，更何况并不是所有的工程师都是数据库专家，所以人为的错误是最常出现的问题。

人为错误分成两种。 一种是操作数据库时犯的错误。另一种是程序员在程序里或者脚本里犯的错误。

操作数据库时犯的错误的概率比较低，但危害却最大。几乎所有的公司都会有类似的传奇时间，我听过的就有三种误操作的版本。

第一种是工程师无意或有意，“不小心” 删掉了数据库核心表中所有的数据。这不是段子，在 Facebook 就曾经发生过类似的事情，当事人还是我的一个朋友，好在后来恢复了，这事儿便也成了他的工程师历史上光辉的一笔。

第二种工程师在线修改表结构（ online schema change ）的时候，不小心一步误操作，结果数据库被锁（ Lock ）长达几个小时，该公司网站也就挂了好几个小时。

第三种是听国内一个大公司的朋友说的，细节还已经无法还原。只记得听说他们的两台服务器，在做主从切换的时候，拔错了一个电源插头，然后……就没有下文了。

程序员在程序里或者脚本里犯的错误就很常见了。

举个简单的例子，我们知道 Ruby on Rails 对数据库的操作基本是通过 Active Record 来完成的。Active Record 可以通过一个数据库连接池来限制每个应用到数据库的连接。

如果某一个程序或者脚本查询没有索引的数据，导致全表扫描，再加上一些网页服务器（Web Server）的并行访问，经常会有整个数据库的所有链接被占用的情况，连终止查询（Kill Query ）都没法执行，只能人肉去做一些类似重启的物理操作。还有更常见的，就是程序里的 N+1 查询问题。面对这类问题，可以使用数据库的连接查询功能，比如 left outer join 来避免 N+1 的问题。

# 最后一个是数据库访问瓶颈

只要是数据库，就有吞吐量的限制，而数据库访问瓶颈便是自然流量增长或者流量突增造成的。只要你的业务在增长，总有一天数据库访问就会达到一个上限。在这个预警到来前，你需要做各种垂直或水平扩展来提升这个上限，或者，你可以通过缓存和其他机制来对访问量进行分流，这里面可以做的工作就多了。

流量的突增一般是类似分布式拒绝服务（DDoS）或市场活动带来的，也可能是因为某个黑天鹅事件造成的，这些原因都很难预料。

如果是有计划的市场活动，就需要提前做好各种战斗准备。如果是恶意攻击，那就只能靠各种防御工程，如 IP 阻塞（IP Blocking）或者第三方的高防系统挡掉这些流量来保证数据库的正常工作。

# 研发过程中，有哪些与数据库相关的问题呢？

我们以 MySQL 为例，讲讲日常开发中应该注意的，与数据库相关的问题。

# 索引

创建索引通常是为了提高常用查询语句的性能，将某些列以特定的数据结构（常见的如 B-Tree）有序存储起来。维持这样的一个数据结构在写数据的时候会有一些系统开销 （Overhead）。但如果查询确实是高频的，那么这样的系统开销就很划算。在建表时需要考虑所有可能的高频查询，另一方面，忌讳过度地“为未来设计 ”(Design for the Future)，也就是加一堆可能根本不常用的索引，反而增加了写数据时候的成本和负担。

索引另一个常见的用途就是保证某一列或者某几列的组合是唯一的（ Unique ），这也称为唯一性索引（Unique Indexing），在写业务逻辑代码的时候会常常用到。

比如你有一个用户表（ User Table )，你想让所有用户（ User ）的电子邮件都是唯一的，这个时候用唯一索引（ Unique Indexing ）就很方便。不过唯一索引（ Unique Indexing ）和可选列（ Optional Column ）组合在一起的时候，也有很多需要注意的地方。

比如，你想对列X做唯一索引（Unique Index Column X），过了一段时间，也许有些情况下列X（Column X）并不唯一，我们便把索引改成了对列X和列Y作唯一索引（ Unique Index Column X + Column Y），但是列Y（Column Y）是 Nullable 的。

这个时候会出现什么情况呢？

你会有多条记录，有着一样的 X 值，以及 Null 的 Y 值。很意外对吧，原因就是 Null 在数据库里常常解释为 “不确定” 而不是空。

# 事务支持

还有一个比较重要的问题就是数据库的事务支持（Transactional Support）。简单说来，就是利用数据库本身提供的事务性，来封装一系列需要同时完成的动作。

比如在一段事务里面，先执行X，再执行Y (Transaction do X；Y；end)，如果 X 和 Y 都是数据库写操作，那要么两个写操作都成功，要么都失败。也就是说，对数据库的改动会统一把事务所做的修改提交到数据库（Commit），而提交（Commit）前的任何错误都会触发所有更新回滚到开始的状态（Rollback）或引发不正常进程的终止 （Abort）。

虽然正确使用事务支持（Transactional Support）会很方便，但是也常常见到过度使用让代码变得很脆弱甚至是出现 Bug 的情况。

常见的几种情况如下。

1. 事务（Transaction）中封装的代码逻辑太长太复杂，甚至调用了别的函数。很多时候，很难去推理当执行中抛出异常的话，到底哪些会回滚，哪些会产生遗留影响。
2. 事务中封装了与数据库改动无关的逻辑。
3. 事务中有不可逆的操作，例如发送电子邮件给用户，发布（Publish）到一个Job队列（Queue）等。这种情况会导致系统的不一致。比如，一个写操作被回滚了，但这条数据相关的 Job 还是被加入到队列了，就会引发错误。
4. 事务中包括了在不同数据库里面的事务，也就是分布式事务，这需要单独处理。
5. 事务中嵌套了事务，不同情况可能会有不同的结果，如果没搞清楚，就可能出现意外的行为。

更多情况就不一一列举了，但过度使用事务支持往往会让逻辑变得不必要的复杂。

# 数据库锁

数据库会出现 Race Condition，我们常常把 Race Condition 叫做竞争条件，是指多个进程或者线程并发访问和操作同一数据，且执行结果与访问发生的特定顺序有关的现象。

如何解决竞争条件（ Race Condition ）呢？常见的方法是使用各种锁机制来确保行为的可预测性和正确性。根据实际情况的不同，加锁的方式会不一样。

常见的有乐观锁定（ Optimistic Locking ）和悲观锁定（ Pessimistic Locking）。总的说来，前者在对性能要求比较高的系统里更为常见。在实际应用中，很多系统都会自己实现锁定（ Locking ）机制。

# 缓存和主从机制

为了提高性能，我们会为数据库增加缓存（Caching）和主从（Master - Slave）等机制，这有时候会引起数据的不一致性。常见的情况是，如果系统默认是在从节点（Slave）读数据，那么一些刚刚更新到主节点（Master）的数据在读的时候就有可能读不到。这个情况在使用一些数据关联（Association）的时候更容易读不到。Rails 的 Active Record 数据关联（Association），就很容易出现这一类的问题。

今天跟大家介绍了不少数据库相关的基础知识，如果你是个软件工程师，想必这些内容都已经耳熟能详，我们来总结一下。

1. 本文从 MySQL 和 PostgreSQL 的迁移和选型入手，介绍了数据库的技术特点和选型问题。在我眼里，没有更好的技术，只有更适合的技术。
2. 数据库领域会碰到哪些问题呢？我为大家介绍了数据库选择、数据库相关的架构问题、人为失误的问题，还有数据库遭遇流量瓶颈以及相关的应对方式。
3. 几乎每个工程师在编程的时候都会和数据库打交道，研发过程中我们应该注意什么问题呢？在这个章节我们讲了索引、事务支持、数据库锁、缓存和主从机制。

文中每个点都可以深入展开，独自梳理，最终形成系统的知识储备。因为篇幅所限，不能涉及所有的技术细节，文中提到的内容都是我在工作中遇到过的问题和实战经验，希望对你有帮助。

如果你有不同的想法，更好的观点，请在留言中告诉我，互通有无，一起成长。

参考链接：https://eng.uber.com/mysql-migration/


戳此获取你的专属海报

# 管理者在进行工作分配时，会考虑哪些问题？

作为技术管理者，我们的主要职责之一就是将工作进行分解和授权。那么，在这个过程中，是否有一些需要注意的事项呢？

对于不同的工作类型和工作场景而言，这个问题的答案自然也不尽相同，但总有一些共性的东西可以抽象出来，供我们参考。

今天我就和你聊一聊：管理者在进行工作分配的时候，需要注意的问题。

一般情况下，管理者会倾向于把重要、困难的工作交给自己信任的人。这种信任，包含了对他了解的程度，对其工作态度和能力水平的肯定，也就是说，你需要作出判断：接到任务的人，到底能不能全力以赴把事情做到最好。

然而，“冰冻三尺非一日之寒”，信任并不是短期内便能够建立起来的。那么，在没有建立起充分信任的情况下，我们该如何分配工作呢？

## 第一点：建立参考基线

当和一个人没有任何直接接触的时候，我们可以通过第三方评价、个人履历以及该员工做过的项目或产品来衡量他的能力。

如果我们对该员工做过的项目有所了解，或者他得到了曾经与之共事过人的大力推荐，那么我们就可以建立起一个初步的参考基线，用来评估他是否是完成当前任务的最佳人选。

## 第二点：问对问题比正确答案更重要

把任务交到员工手里之前，要和他进行充分的沟通。告诉他任务的详细情形，看他会问出什么样的问题，提出哪些想法。

这些也是评估一个人能否胜任工作的重要标准，问对问题，有时候比给出正确答案更重要。

沟通的时候，要看他的问题和想法是否会尽可能考虑所有的情况，问问题和提想法之前是会去调研，还是直接去做一些想当然的假定。在你给出反馈意见或指导性建议之后，他的反馈是什么，又会问出什么样的问

题，提出哪些想法……这些都能帮助我们进一步评估他是不是真的了解任务的状况，有没有综合去考虑任务中的问题。

## 第三点：工期估算

估算完成任务的工期是分配任务中必不可少的环节。你可以让接受任务的员工试着去估算：需要多久完成，大概什么时候完成，需要什么样的资源等。

如果他是个思维缜密的人，就会去考虑完成这个任务的所有相关问题。开发工作量是多大，会不会依赖其他人的工作，有多少沟通成本，技术难点是什么，有没有现成的方案，系统框架是什么，后期集成和测试的时间成本有多少，综合考虑后，再给出一个相对全面的时间估算。

很多时候，我们希望任务能够更快完成，所以员工给出一个短平快的工期是符合我们的预期，但如果他很多关键的因素都没有考虑到，那这就是一个过于乐观和不切实际的估算。

即使他给出了承诺，也不可能按期完成，因为对方并没有真正理解问题的复杂度。

如果一个人不能花费足够的时间去了解自己未知的部分，我们很难放心地把任务交给他独立完成。

## 第四点：执行力

执行力也是我们在分配任务的时候需要考虑的一个因素。

工作中有一些人，他们的沟通能力、计划能力都很强，但是执行力却比较差，或者没有粘滞力，执行过程中容易遇难就退、虎头蛇尾。

这样的人，不论项目初期多么细致、周到、井井有条，最后总是不能按期完成任务，或者工作成果与自己初期的规划有很大出入，甚至代码里有很多 Bug。这些人属于能说但不能把事情做好的人，难以托付大事。如果你的项目中有重要的任务，切记不要交给这样的人去做。

## 第五点：后期维护

完成一个项目并不意味着项目的结束，很多时候，项目上线了还需要一段时间的维护工作。这包括了 Bug 修复，排查用户反馈的问题，完成后续的迭代开发等等。于是，你需要去观察：一个人是不是可以自觉地维护产品，有没有责任感，会不会推卸责任，出了问题，能不能第一时间

冲到一线解决。这些都是很重要的品质，也是你判断他是否能够承担更重要任务的参考。

当一个员工的项目越做越好时，我们也会很愿意将任务交给他，并提供支持和帮助。反之，如果员工反复犯同样的错误，这样就算扶上马也会一程都送不出去的人，带起来就会非常费劲，也很难承担重要的任务。

另外，在分配工作的过程中，你还需要注意一些细节问题。

首先是如何对待职场新人

一些职场新人很有潜力，但经验不足，也许他们在项目初期有很多地方考虑不完善。这个时候，不要轻易地否定他们的工作。如果你可以耐心地花些时间悉心指导，他们可以快速地学习并且进步。

其次是如何针对不同类型的员工分配工作

管理者还会遇到各个类型的人才，有的技术强一些，有的协作好一些，有的慢性子，有的做事比较急躁。在分配任务的时候，你需要根据每个人的特点安排不同类型的任务，并提供相应的支持和帮助，扬长避短，才能发挥人员的最大效力。

比如，有的人干活比较慢，但是慢工出细活，做出来的东西出错少，即使有错也能快速定位，搞清楚原因。而有的人手快，可能早期代码错误多一些，但是能够快速迭代，不断改进，最终也能做出稳定的线上产品。

针对这种情况，我们可以把线上产品的改动交给慢而稳的那个人，把需要快速迭代开发的产品交给能快速出结果的人，最终，这两个人都可以变成项目的核心成员。

最后还要注意，大项目的工作分配

以上，我介绍了在员工独立完成某项任务的情况下，分配工作时要注意的问题。如果遇到了大项目时，你分解后的项目需要一个团队去协作完成，作为管理者的你应该怎么做呢？

这时候，你就需要指定一个团队负责人了，这个人也同样要具备分配任务的能力，很多事他不再是亲力亲为，而是采用本文提到的方法，把需要完成的工作，分配给最合适的人，团队成员通力配合，才能把项目做好。

文章的最后，我来讲一讲为什么会谈这个问题

我们在分配任务给别人的时候，别人也在分配任务给我们。我们在别人身上寻找某种品质，别人也在我们身上找寻类似的东西。

如果你希望承担更重要的任务，成为有担当的人，一方面需要提高自身的能力，发挥自己的特长，另一方面就是成为一个有态度的人。拿到一个任务，能不能全力以赴把事情做到最好。这些东西不仅仅会决定一个

项目的成败，更重要的是打磨我们自己的心性，让我们在别人眼中，成为一个可以托付重任的人。

总结一下，今天我首先分享了在没有建立起充分信任的情况下，我们该如何分配工作？这包括了建立参考基线、问对问题、估算工期、是否具备执行力、后期维护的时候能否尽职尽责等内容，之后，我还分享了如何对待职场新人，如何针对员工的特点分配不同类型的工作，如何分配大项目等问题。

你在分配任务或接受工作的时候，会考虑或提出哪些问题呢？可以在留言中告诉我，我们一起成长。感谢你的收听，再见。

# 硅谷人到底忙不忙？

经常有读者问我，硅谷的工程师忙不忙？你们有没有国内的“996”，平时加班多吗？

硅谷人到底忙不忙？这个问题其实很难一概而论。

在Airbnb，乃至整个硅谷的很多公司里，都有两类人和谐共处。一种人以工作为乐，他们晚上和周末大多数时间都会泡在公司里加班，就算偶尔在家里，他们也会学习或处理工作相关的事情，我们常常称这种人为“劳模”或“工作狂”。

还有一种人，他们在工作时会认真负责，兢兢业业，但工作时间之外，他们很少做工作相关的事情，周末基本不会在线或者回复 Email。

喜欢工作的人，视工作如生命，他们是真的喜欢工作，加班并不是因为工作忙不完，而是因为热情，他们会享受加班过程中的思维碰撞，更喜欢完成项目时带来的成就感。这些人忙起来，比简单的“ 996 ”还要辛苦得多，他们每天工作12小时是常有的事，即使自己的工作忙完了，也会主动去跟技术经理要更多的任务和挑战，或者干脆自己编程学习。

另一些人呢，他们在工作之余更愿意和家人在一起，丰富一下家庭生活，或者根据个人兴趣去攀岩、弹琴、唱歌、摄影，有的人还喜欢空闲的时候去做义工。他们热爱工作的同时，也享受生活。

硅谷的氛围是相对自由的，“打卡”或者“996”这样的公司估计很难存活下去，因为，加不加班，拼不拼命，大都是员工自己的选择。

所以，作为领导者，常常需要将“自由”和“原则”这样的理念传递出去。前段时间，我们组里有个刚毕业不久的工程师找我谈话。他是个很有上进心的工程师，热爱工作，干起活来不知道休息，之前就经常问我有没有更多的工作让他做。

他对我说：“你们（管理者）总是给大家发邮件叮嘱，要注意工作和生活的平衡，让大伙尽量不要加班，可是我晚上就是想工作啊，这样是不是不太好？”

他这个问题让我想起了自己的故事。

我属于典型的工作狂，非常喜欢工作给我带来的享受。除了特殊情况，我基本不分白天晚上，工作日还是周末，差不多把所有的时间和精力都扑在工作上。我在做工程师的时候，加班是常有的事，周末发工作邮件更是习以为常了。

刚刚转成管理者的时候，我也保持了这个习惯，后来我的老板就提醒我，作为管理者，尽可能不要在工作以外的时间发工作邮件，因为这样

很容易形成鼓励加班的错觉。

说实话，刚刚收到这个反馈的时候，我是不太能理解的，但是后来慢慢明白了，这样做确实有一些弊端：比如，我会给那些选择工作时间之外享受生活的人带来很大的压力。

那么问题来了，主动延长工作时间，努力完成更多工作，会不会有助于升职加薪呢？

答案是：我觉得在一定程度上会有帮助，但并没有直接联系。因为，在升职时，公司评估的标准不仅仅是你多么努力，管理者考虑更多的是你的项目成果，以及你在项目中表现出来的能力。

现在我们回到文章开头的问题，硅谷人到底忙不忙呢？

其实，对在硅谷奋战的工程师而言，忙不忙完全是由他们决定的，是他们自己的选择，一切工作压力并不来自升职、同事或管理层，而源自每个人的内心。

一旦勤奋工作成了自主的选择，这些人工作起来，不仅会付出更多的时间，也会有更多的责任感，在工作中还会表现出对产品或代码精益求精的态度。

在这样的氛围下，管理者要如何最大程度地帮助组员成长，充分利用这种选择和自由呢？

我觉得，这里很重要的一点是：去和每个组员交流，了解对方真正想要的是什么。

很多人会简单地认为，工作不就是为了升职加薪吗？人们常说，不想当将军的士兵不是好士兵嘛。但是你真正去和员工交流，会得到不同的答案。

我曾经试着在私下聊天时问一些同事或者组员：你在工作和生活中，最想要的是什么？

有些人会说希望得到更好的薪水，获得升职机会等等，但我还得到了一些这样的答案。

> “升不升职我到不太在乎，但是我希望做一些有影响的项目，哪怕只是参与。我想了解更多重要的系统是怎么构建出来的。”
>
> “我想尽可能尝试更多不同的东西。”
>
> “我想有更多旅行时间和机会。”
>
> “我希望能有更多的时间和我的家庭在一起。”

当你知道他们的真实想法后，就可以最大可能地按照他们想要的生活方式去分配工作和任务。

愿意全身心投入工作的，可以分配一些重要的有挑战的新项目给他；愿意尝试不同技术的，多给一些预研和前瞻性的任务；更乐意享受生活的，可以把线上稳定业务的改进和迭代交给他做。在尊重他们选择的前提下，双方保持一致的期望值。你会惊奇地发现，忙不忙，已经不重要了，重要的是，他们的工作会更出色地完成。

今天我结合了自身的成长经历，为你讲述了硅谷工程师的工作和生活方式，也介绍了硅谷人对“加班”的看法和做法。

有的人喜欢全身心投入工作，有的人享受工作和生活带来的平衡，他们都是合格甚至出色的员工。作为领导者，你应该基于员工的选择和自由尽可能去帮助他们成长。了解他们的真实想法，并提供相应的支持和工作安排，他们会回报给你更出色的工作成果。

你选择了什么样的工作和生活方式，又为此做出了哪些努力呢？可以在留言中写下你的故事，我们一起交流讨论，感谢你的收听，再见。

# 每个工程师都应该了解的：系统拆分

四年前，我加入了风头正劲的 Square 公司。两年前，我又加入了涨势甚猛的 Airbnb 公司。在我加入的时候，这两个公司都有上百名的工程师，网站和主要产品的核心功能也已齐备。

两家创业公司从 0 到 1 的创业过程我并没有亲身经历，但是两次都恰好经历了公司从 1 到 N 的扩张过程和业务拆分的过程。

今天我就和你聊聊，公司从 1 到 N 发展过程中的系统拆分问题。

# 创业初期的代码现状

在 Square 刚刚起步的时候，整个产品都是基于 Ruby on Rails 构建的，所有的产品和功能代码几乎都在一个代码库里。

等到我进入 Square 的时候，有一些服务已经从 Ruby 代码中分离出来了，形成了单独的 Java 或者 Ruby 服务，然而大部分功能还是在一大块 Ruby 代码里。

当时，几乎所有的工程师每天都在这一份基准代码（Code Base）里写程序。虽然有严格的代码审核过程和规范的开发流程，但是，不同功能的代码模块会产生交叉影响，不同工程师改动的模块会有重合或牵连，所以，系统还是会时不时出现问题。

那时候， Square 的做法是：在周五对本周所有的代码进行代码审查（Code Review），通过审查之后，把修改合并到主分支，然后再发布到生产环境。

这种做法虽然可以避免产生人为错误，但是非常不灵活，比如，每周只有周五有一次机会将改进的代码部署到线上。

可以想象一下，一百多名工程师，就算只有三分之一的人在这个代码池子里改代码，一周累积下来，已经有不少的改动了。

于是，当时 Square 有个系统管理员组（Sysops），专门负责每周五的部署。我也是工作近半年的时候，因为表现不错才被荣幸地 “选拔” 进了这个 “特别行动小组”，承担部署的重任。

那么说，每次的部署是一幅什么样的场景呢？

部署开始的时候，一正一副两位工程师正襟危坐，多个显示器同时打开，进行各种指标监控。

工程师先将在测试环境中测试无误的代码部署到若干生产机器上，进行灰度发布，这就意味着有一部分用户的访问量会调用新代码。如果监控没有发现异常的话，再进行全量发布，这周修改的代码就会被部署到几百台机器上。

一旦出现异常，监控系统就开始各种红色告警，工程师们会立刻扔下手中的可乐或者咖啡，进入备战状态，停止系统，进行数据回滚、排查问题、修复，从头开始把流程再来一遍，直到代码安全地部署到线上并能够正常运行为止。

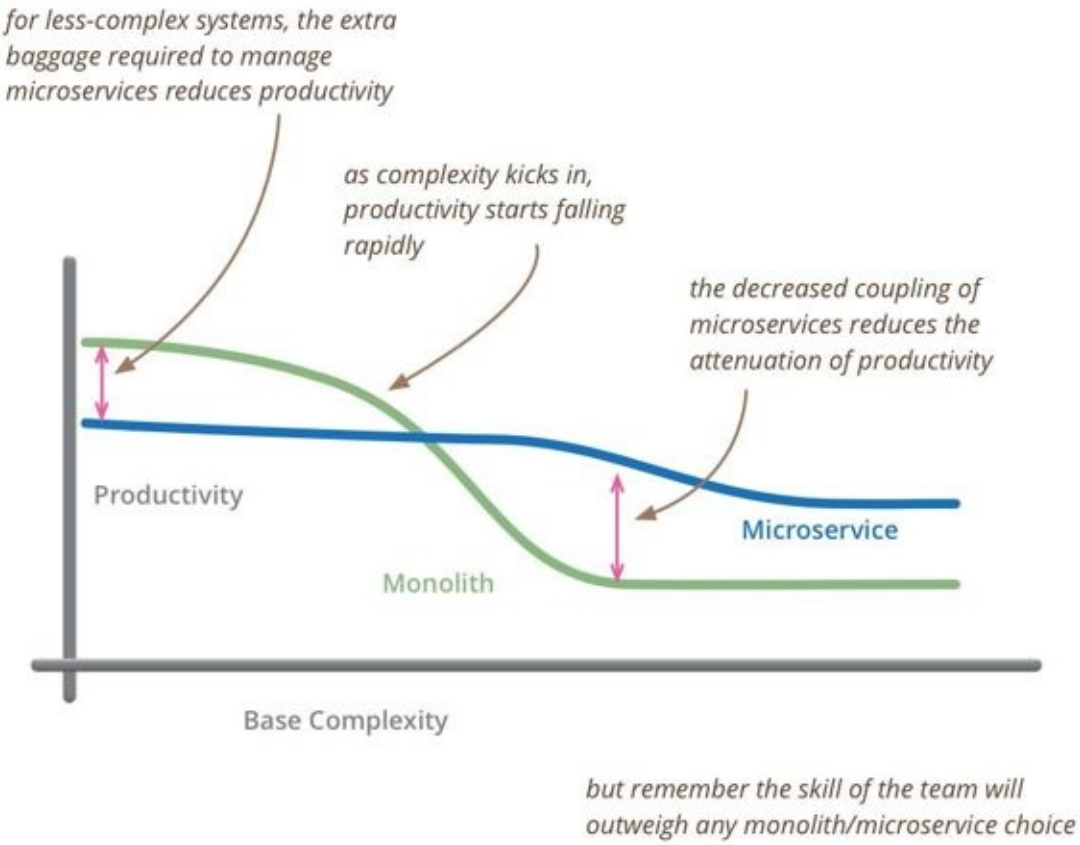
随后的两年，我们进行了细致的业务拆分，等到我离开 Square 的时候，大部分可以独立出来的服务都已经拆分出来，很多系统可以分别部署和上线，也就再没有了那种激动人心的周五上线日。

Airbnb 的情况也差不多，我刚加入的时候，代码状态甚至更原始一些。不同的是，Airbnb 没有一周只能部署一次代码的规矩，所有的工程师只要准备好了就可以做部署上线。

这样做的优点是可以快速迭代，每次部署的代码改动也很小，缺点是几乎任何时候都有人在部署代码。时时的部署也就意味着，红色告警随时可能在身边响起。

# 为什么系统需要进行业务拆分

为什么会出现这种情况呢？我在文稿中给大家放了一张图，图例很好地阐述了效率和复杂度的关系。



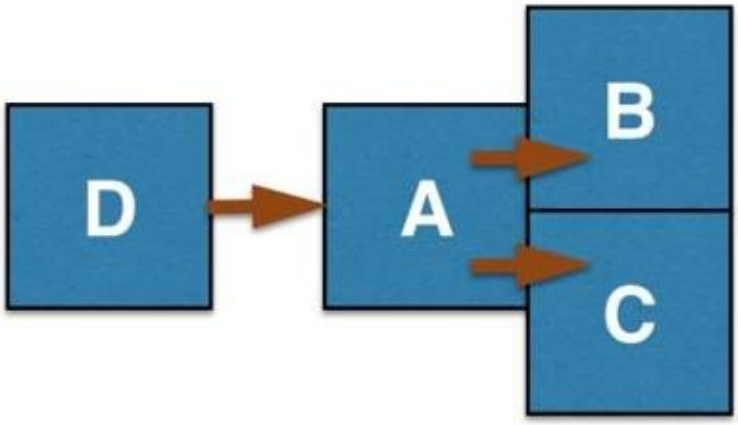

图的 X 轴代表了基本复杂度（Base Complexity），Y 轴代表了生产效率。我们可以看出，当一个公司规模很小的时候，基本复杂度相对较小，所以单一代码库（Monolith）的效率就会高。

然而，随着公司业务的扩展，访问量的增加，其基本复杂度就会逐步升高，达到某一个临界点后，微服务（Microservice）的效率就远远高于单一代码库。关于微服务，这里就不做详述了，极客时间会发布专门的微服务知识产品。

为了解决效率和复杂度的问题，无论是在 Square 还是 Airbnb，我都有一大部分时间花在了业务拆分上。下面，我就和你聊聊这几年做业务拆分的一些心得和踩过的坑。

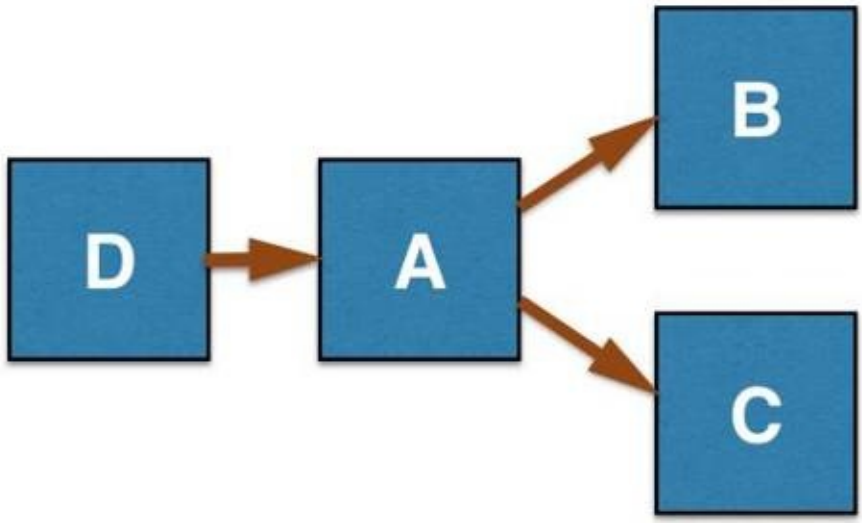
# 业务拆分并不像看起来那么简单

我们从一个例子谈起，比如你有一个功能模块，大概可以分成四部分。其中模块 A 连接一个外部模块 D，A 输出的结果，会被模块 B 和 模块 C 分别调用。




针对这样的模块，我们可以做一个集成测试（Integration Test），在模拟（mock）D 的情况下，测试 A、B、C 是不是可以正确运行。

如果有人修改了模块 A 的返回值，但忘了修改模块 B 和 C 的接口，测试就会立刻失败，不会存在因为忘了修改接口而测试通过的可能。一旦通过了集成测试，所有的改动会在一次部署中同时展现（Rollout） 或者回滚（Rollback），非常容易控制。




随着业务的发展，A、B、C 三个功能被拆分成三个独立的服务（Service），各自保存在不同的代码库，或者是同一个代码库不同的服务容器（Service Container）里。

这样的话，测试用例就不能综合测试这三部分的功能了，只能模拟相互

的请求（Request）或响应（Response），如果在开发环境下联调测试，则需要本地建立这三个服务。

根据每个公司开发环境的成熟度，这一步可能很简单，也可能耗掉你几个小时，才能让不同服务在本地正常运行，并且需要通过 RPC 相互调用。

RPC 就是远程过程调用的意思。有远程调用，就会用到本地桩和过程调用，这涉及了本地多服务的配置，过程繁复，不小心就会引入错误，测试成本也会随之增加。

如果程序员在改动的时候并没有按照正常流程进行测试，尤其是一些“很小的” 或者 “不相干的” 改动，一旦部署上线，系统就可能出现各种各样的问题。

就算一切顺利，有一天，A 修改了自己的接口，RPC 调用中请求的一个字段（Field）从 integer 变成 string 类型。

如果 A、B、C 还在一起的时候，我们在代码库里把三者的相应类型都改了就好；但是，现在 A、B、C 都是独立的服务，可以独立地部署，这事就有点麻烦了，我们很难保证 A、B、C 的部署总是完全同步。

有经验的读者知道，我们为接口做个向后兼容（Backward Compatibility）就好了，只要：

1. 先改 A 的接口，让它接受 integer 也接受 string，如果请求是 integer，先做一下转换，然后发布这个改动；
2. 修改 B 和 C 的接口，响应从 integer 变成 string 类型，发布这个改动；
3. 等到 A、B、C 的新代码都稳定了，再修改 A 的接口，只接受 string 类型的参数，发布这个变化，我们就完成了所有接口的改动。

这样就没问题了么？并没有这么简单。

因为 A 还有其他代码，所以在上面的第二步之后，你有可能发现 A 的代码有一个问题，需要将线上的代码回滚到之前的某个版本。这时候 B 和 C 的接口已经是 string 类型了，而 A 只接受 integer，然后，线上就是频繁请求报错。

当然，这里举的是一个简单的例子，我们可以通过延长第一步的兼容时间来避免出现类似问题，但是，实际工作中的改动不会是这么简单的依赖关系，或者没有约束关系，所以，服务之间无缝修改接口，是一个需要非常小心的问题。

# 业务拆分时的注意事项

系统拆分后的痛远远不止于此。就我自己的经历，大概有下面的这些感受。

## 测试会变得异常复杂

因为模块被独立出来之后，并没有办法很方便地写出集成测试用例。一个做法是模拟出所有接口的请求和响应，但实际上大部分时候根本没法测试跨服务的改动，这种做法多少有点自欺欺人的味道。

另一个方法就是在本地配置好所有的服务，用真实的服务响应来测试。但是撇开本地设置多服务的复杂度，保证本地服务一直是最新代码，同样也是一件麻烦的事。

尤其是同步开发的工程师变多以后，可能你正在测的服务没有问题，但是在你做测试的同时，已经有同事对你刚刚测试的服务做改动推送到了主分支上。

测试的复杂度，几乎是软件工程中的万恶之源。当每个小改动都让测试变得耗时耗力时，就难保没有偷工减料的员工，大家揣着 “我的改动应该没问题” 的侥幸心理，不去做完整测试，就把自己的代码合并进主分支。

尤其是大部分这么做的改动都没有问题，时间一长，侥幸心理一再滋长，人们直接合并代码的胆子也越来越大，终于有一天会把生产系统彻底搞挂。

针对这个问题，我和在 Google 工作的朋友交流过。Google 或者 Facebook 这样的大公司里，整个系统做得相当成熟，测试环境做得非常完美。

每个服务都对应设置了在线的测试服务，写集成测试极其方便，或者把服务做成开箱即用，工程师可以一次性地建立所有的本地服务进行联调和测试，但是，对于大部分创业公司来说，很难达到这个水准。

## 与接口相关的改动需要大量协调

这一点也很容易理解。比如我们要把一部分代码从一个服务迁移到另一个服务，或者修改 API 的协议，那么，所有不同服务的维护者都需要在代码里增加向前或者向后的兼容性，对代码进行保护。

同时代码的上线顺序和修改顺序也息息相关，我们需要做一张检查列表（Checklist），考虑各种可能性，精确地按照顺序执行。一旦发生代码回滚，可能又要重来一遍。

这个过程就会涉及方方面面的人、事和代码修改，过程十分繁复。

## 报错的处理

因为程序不在一起了，当异常发生的时候，我们就得不到完整的异常堆栈信息（Exception Stack），只能追踪到某个服务的接口处，于是 Debug 变得很难。你还需要去另一个服务的日志里去找，看看那个时间点从你这里发出去的请求到底发生了些什么事，然后才能进一步定位问题。

好的程序员在写服务的时候知道要把异常信息封装后层层传播出去，并最终暴露到接口的 4XX 响应里，这样，调用方就可以在堆栈信息里看到具体的出错信息。

如果有的程序员没有这么做，就很容易出现 “无语问苍天” 的感觉，比如你线上的服务出了问题，到日志系统 Kibana 里一看，只有下面这行错误信息，你是不是会很崩溃？

Error! HTTP 400 response from http://another-microservice.com/update

## 日志的完整性

系统拆分了，日志系统也会分离，不仅系统调试变得困难，一些基于日志产生的事件流（Event Stream）机制，也会变得难以处理。这意味着，想要真正从日志里获取完整有用的信息，就需要将不同服务的日志一起取出来进行分析和处理。

这种需求并不是所有的应用都需要，因为我们是做支付的，经常需要一个事务的完整审计线索（Audit trail），也就是一条告诉我们 “每个相关的变化是谁做的，什么时候做的（who did what and when for every change involved）” 的特殊日志。

这件事以前处理起来非常复杂，现在倒是有了比较标准的解决方案，就是一个共享的消息总线（Message Bus），比如 Kafka，有了日志就分门别类的扔到消息总线里处理，然后再进行分析。

## 超时设置

为了保证用户体验，我们常常在系统里做一些超时设置（Timeout），比如一个请求从终端设备发过来，我们希望用户最多等待 5 秒，超过 5 秒就会放弃请求并返回相应的结果通知用户。系统拆分之后，我们可以做一个全局的超时设置，让所有的服务都使用这个全局变量。

这一切看起来很美好，但稍不注意就会出现这样那样的问题。由于服务都是独立开发的，如果某一个服务的实现没有使用 5 秒的全局变量，我们就不知道这个服务到底超时多久才会返回结果，或者是否有超时的设置。

另外，根据某些服务的性质不同，我们希望尽可能地给出最合理的延时设置。还有些请求会经历多次跨服务的调用，一旦同时出现超时，就会进行叠加，超时设置就完全不可控了。

为了避免这些情况出现，就需要增加流程和规范，并且在进行系统拆分的时候进行宏观的设计和考虑。系统拆分会为我们带来灵活性，同样也会增加其他成本。

# 关于代码自由

记得以前看到过一句话，当每个人都有绝对自由的时候，这个世界就没有自由可言了。拆分之后每个服务的实现都可以自主选择自己的语言，自己的数据存储方式，自己的代码风格。

短期来说，这种做法可以让程序员的效率极大地提高，但是在同一个公司里，当各种各样的服务变成一场技术秀的时候，不论是维护还是稳定性都会受到极大的挑战。于是，这时便会有人扮演清道夫的角色，开始搞服务的标准化。

另外，独立服务的开发周期相对较短，往往一两个工程师几周时间就可以写出一个新的服务，这样系统里会出现数不清的服务，有的服务由于人员离职等原因没人维护了，有的服务被重写了，有的服务要退休了，为了管理这些服务，我们还需要一个服务编排和管理系统。

系统拆分之路漫漫，吾等将上下而求索。

# 如何去判断系统是不是到了必须进行拆分和服务化的临界点

写到这里可能有读者会问，这篇文章中你介绍了大代码库的弊端，也写了很多系统拆分和服务化需要注意的问题，那么，你到底想告诉我什么呢？

做为一个亲身经历过两种架构的工程师，我想说的是：系统拆分并不是做一道单选题，在进行系统拆分和服务化之前，我们需要综合考虑各种因素，找到平衡点。

1. 你的业务量是否足够大，逻辑是否足够复杂以至于必须进行系统拆分。水平扩展是不是已经不起作用了？代码的相互影响、部署时间过长真的是系统的切肤之痛么？如果答案都是肯定的，那么你就应该进行系统拆分了。
2. 对于服务化的架构，你的开发人员多少经验，能否正确驾驭而不是让本文中提到的问题成为拦路虎么？
3. 系统拆分是一个“从一到多容易，从多到一困难”的过程，这个过程几乎是不可逆的。一旦你三分天下，想再一统江山就没那么容易了。所以在做拆分计划的时候，一定要慎之又慎。

系统拆分是一个实践性很强的工作，并无一定之规，只有亲自参与了这个过程，才会有更深入的体会；在这个过程中，你的架构能力也会产生一个质的跃迁。

文章的最后，我来总结一下今天分享的内容。

今天的文章较长，涉及的内容也比较多：

第一点，我谈到了创业公司初期，代码的构建状况以及遇到的相关问题；

第二点，我解释了为什么随着业务的发展，我们会进行系统拆分；

第三点，我提醒了业务拆分并不像看起来那么简单，我们需要时刻去注意细节；

第四点，我分析了在进行系统拆分和服务化的过程中，需要注意哪些问题；

第五点，我讲解了如何去判断一个系统是不是到了必须进行拆分和服务化的临界点。

希望这些内容对走在创业路上的技术人有所帮助。

你有系统拆分的经验和故事么，可以在留言中告诉我，我们一起讨论，也欢迎你把这篇文章转发给你工作中的伙伴，我们一起成长。再见。


戳此获取你的专属海报

# 技术人如何建立个人影响力?

程序员虽然经常和管理层或其他部门协作，但更多的时候，还是和技术人员内部的分工协作。虽然好的领导者会让每个人的职责范围尽量清晰，但时不时也会有下面这样的事情发生。

程序员小罗为项目做了很多调研工作，也有一些很好的想法。同事小唐和小罗一起合作，也互相讨论问题。几次讨论过后，小唐就写了个设计文档，署了自己的名字，并发送给大家审核。小罗一看，发现很多内容都是自己之前和小唐讨论时说的，心里就觉得很不舒服。

小王和小李，一个在公司时间久一点，属于老员工，另一个级别稍微高一点，但是加入公司时间不久，属于资深的新人。在一些项目的问题上，两人各执己见，只要有不同意见，就很难说服对方。老员工小王觉得：“你根本不了解我们的系统。” 资深新人小李觉得：“这是编程中的重要思想，怎么你就不明白呢？”

这些其实都和个人的影响力有关。第一个例子，我们会觉得别人抢了自己的功劳，自己的成绩被忽略了，失去了该有的存在感。第二个例子是因为我们工作中和别人协作的难易程度，与我们在别人眼中的可信赖度息息相关，包括我们如何用自己的观点去影响和说服别人。

我个人理解的影响力，应该包括以下几个方面。

1. 存在感：你的意见别人会认真听。
2. 说服力：当别人和你意见不同的时候，能有效地让对方真的明白你的出发点，虽然不一定百分之百地采纳，但会认可你的观点。
3. 谈判力：双方共同完成一个项目，让职责划分尽可能公平，对双方都有利。
4. 协调力：多方合作的时候，你能起到桥梁的作用，促进多方更好地沟通。
5. 鼓舞力：也就是我们平时说的灌鸡汤、打鸡血的能力。

那么作为技术人，怎样在这些方面提升自己的能力，从而建立个人影响力呢？

我先说存在感。如何让别人能够认真听取你的意见呢？换位思考一下，如果别人发现，每次你给出的意见都会让事情变得顺利，或者避开了一些可能会栽进去的坑，那他一定会变得更尊重你的看法。

我的建议是：不要在一些可有可无的事情上纠缠，非让别人尊重你的意见。对一件事情不确定的话，要说清楚哪些是自己有把握的，哪些只是自己的推测。如果一个观点来自别人，点明这是谁说的，一是对别人的

尊重，二是对方如果需要跟进了解，那么，他知道该去找谁。总而言之，有三分话就不要说五分话，有七分话就不要说十分话。

再说一下说服力。说服别人之前，要尽可能保证自己观点清晰，条理清楚。最重要的是，你需要站在对方的立场去了解，为什么他会不认同你的观点。很多时候，我们只是没有完全理解对方的想法，结果造成了无谓的争论；所以，增加说服力的重点是，找到对方不认可的那些地方，反复讨论，达成一致，而不是去争执自己正确的观点有多么“正确”。

谈判力取决于你的知识、智慧、经验和直觉。要对某一件事具备精确的判断力，你必须去搞清楚事情的来龙去脉，也就是事件的经过和背景。除此之外，你还需要知道，哪些是自己的底线，哪些是对方的利益，通过最大化双方共同利益建立合作，并尽可能最小化双方冲突的地方。

在谈判的过程中，你可以让步，但是要让对方明白，这是自己的一个让步，希望对方理解，并在一些自己无法让步的方面提供支持。当然，谈判力是一种不能速成的软技能，但是作为技术人，保持你的逻辑清晰，懂得适度取舍，就是一个好的起点。

我们再来说说协调力。培养多方协调的能力，需要从平时做起。工作中你需要和经常一起合作的人保持良好关系和顺畅沟通，学会成为团队里“让事情发生的人”，搞清楚各组人员之间的相关性，项目进度之间的依赖性等。

对于资源的变更保持敏感度，面对项目计划的变更、人员和岗位的调整，要能适时合理地传达，确保合作计划尽可能地随之变化，各方面都有相应的协同。不要在任何一方还没有了解变化的情况下做出决定，如果必须做决定，也要使用对方能够接受的方式提前沟通。

最后说说鼓舞人的能力。如果你是一个技术领导者，就应该培养自己“振臂一呼，应者云集”的能力，这是技术领导力的体现之一。

那么，如何去鼓舞人呢？当众演说能力当然很重要，不过我在这里要强调的是对个人的鼓舞。

比如，你可以去了解对方担心的地方，并适时加以帮助。预先获知对方的长处，在他取得阶段性胜利的时候，或通过自己的努力克服困难的时候，找机会明确地对他的长处给予肯定，并对他的付出表示感谢、认可或者赞赏。如果有意见，多提建设性的意见，让对方在相对愉快的心境下做出改变。

最后，我来做个小结。

建立个人的影响力并不是为了凸显你自己有多重要，或者让别人更加认可你，而是通过影响力把事情做成。

在一个技术团队里，影响力很多时候与你的贡献紧密相关。如果你的存在能够让别人的工作更好地推进，让别人更容易取得成就，久而久之，

伙伴们就会很自然地认可你，信服你，有问题的时候，也会想到要咨询你的意见。帮助别人成功，帮助别人成就梦想，你就会具备积极正向的影响力。

你对技术人如何建立个人影响力有哪些好的意见或者建议，欢迎留言，我们共同成长。


戳此获取你的专属海报

# 管理者不用亲力亲为：关键是什么？

大部分被提拔成技术领导的工程师都拥有一定的领导力，最重要的是，他们都具备出色的业务能力；而这些技术能力强悍的工程师刚刚走上管理岗位时，最爱做的事情就是亲力亲为，看谁干活都不放心，恨不得自己把所有的事情都做了。

对此，我深有体会，因为我是一个典型的由技术骨干提拔上来的管理者。

成为技术管理者之前，我在组里是研发组长（Tech Lead）的角色。我不仅参与所有的设计和讨论，甚至很多的核心代码都是自己写；组里其他人的代码，我也会亲自过目。无论多忙，我都会参与所有的代码评审，尽力做到对每一个改动都心中有数。

随着业务规模的扩大，我带的项目组人数越来越多，事事亲为也变得越来越吃力。晚上和周末的时间差不多全部用来加班了，超长的工作时间变成常态。

即使这样，我也总有忙不完的事。好友池建强老师常常说，你得学着带人和授权，把事情分出轻重缓急，把有些事分出去让别人做。可那个时候的我，总觉得每件事情都很关键，什么都不愿放手，甚至我老板也常常说我，应该学会把任务授权给给别人。

我在这方面做得一直不好。一方面，我确实没有掌握到分配任务的技巧，好几次任务分出去都出了小问题，最终还要自己介入；另一方面，我那个时候的状态也不对，整个人陷到所有的细节中无法自拔，没办法做到退一步海阔天空，只能紧紧盯住眼前的事，无法看得更高更远，甚至常常在度假的时候也要处理工作上的事情。

对个人而言，我自己忙得疲惫不堪，对团队而言，一旦我这里停下来，就成了所有人的瓶颈。

这个问题在我转成管理者之后不久，变得更加突出。一方面，我多了很多管理相关的工作，这些任务会占用我很多的时间；另一方面，团队迅速成长，项目和人都变得更多了，想要事事亲力亲为再也不可能了。终于有一天，事情突破了临界点，也就是说，无论我怎么忙，都没办法延续之前的管理方式了。

当一个人熟悉的“老系统”彻底坍塌的时候，人们才会改变，去尝试构建一套新系统；但是，新系统从来不是一蹴而就的，我花费了好大的力气才从“亲力亲为”转变为授权和任务分配模式。

为什么这么困难呢？原因很多，我觉得最重要的一点可能和工程师的工作习惯有关系：一个技术方案如果自己不参与，就会担心事情不能做好。

这里面其实有两个误区：

1. 事情能不能做好和完全按照你自己的方式做好，是两码事，别人有别人的工作方式，也能把事情做好。
2. 介入和不介入并不是非黑即白，用什么方式介入，在哪些地方介入，才是关键。

既然出发点是把事情做好，那如果我们把任务分配出去，别人也一样能把事情做好，如果这样去思考，是不是就更容易放手了呢。

这样的话，我们就会把关注点放在如何帮助别人把事情做好上面。比如，接受任务的人需要哪些支持和帮助才能很好的完成任务呢，我们应该在什么样的时间点去提供这样的帮助。

作为一个管理者，我们在授权和分配任务的时候应该注意这些方面。

第一，让对方明确目标，知道最终想要达到的结果是什么，对这个任务完成的期望值是怎样的。比如，在什么时间内完成什么样的任务，你对这个任务成功的定义是什么样，如果有取舍，哪些是重要的，哪些是次要的，哪些是可以妥协的。如果这些标准没有说清楚，很可能会出现认知偏差，比如对方觉得做得很好，但是你认为有些很重要的东西并没有被照顾到。

当然，这里面一个重要的前提是：对方同意并承诺完成这个目标。

第二，制定一个计划，并保持跟进。跟进不是指导，不是让你去频繁介入别人的任务，告诉对方下面你该干什么了；而是需要你在对方对某个环节有疑问的时候，能够随时提供帮助。这种帮助可能是解决方案，也可能是技术方向。当对方对下一步该怎么做没有头绪的时候，你可以与其探讨，给出建议或者线索，确保没有大的障碍阻止他取得关键性进展。

第三，给出反馈，尤其是正面的反馈。当对方做得很好的时侯，你需要及时地给予肯定。对方处于平台期没有突破的时候，只要他还在努力，就应该给予适当的鼓励。反馈要尽可能的客观，不要在小细节上有太多意见。如果在方向或者优先级等问题上偏离了你的预期，要及时交流和纠正，了解对方的想法，找到问题的原因，让项目走上正轨。

那么，我们回到今天的主题：管理者不用亲力亲为，关键是什么？

我认为，关键就在于学会授权和任务分配。这个过程需要注意两个重点：第一，我们要有效地把任务分配出去，第二，我们要保证分配出去的任务能够被圆满地完成。

任务无论是自己做还是交给别人做，让事情更快更好地完成永远是第一位的。很多优秀的工程师最初都是独行侠，他们单枪匹马完成了很多丰功伟绩，也更习惯自己独立工作，但一旦他们发现团队协作可以做出更大的成就时，就会从亲力亲为转变为授权模式，帮助别人成功，团队才会获得更大的成功。

回顾一下，今天，我和你讨论了为什么工程师出身的技术领导更容易亲力亲为，如何从亲力亲为模式转变为授权和分配任务模式，以及我们在进行任务分配的时候，应该注意哪些方面。

最后给你留一个思考题。如果你在分配任务的过程中，对方不认可你设定的目标，或者不同意你对项目成功的定义，你该怎么做呢？可以在留言中告诉我，我们一起讨论。

# 每个工程师都应该了解的：API的设计和实现

在一个初创公司成长的过程中，作为工程师的你也许常常会遇到下面这样的情况。

有一天，你看到一个段代码或一个算法，觉得这些代码不大经得起推敲；于是你用 git blame 命令去寻找代码的主人；结果发现，原来作者是如今早就不写代码的 CTO 或 VP。

之后，在一个偶然的机会里，你和他讲起这件事，他会自豪地告诉你：“哦，那时候我们必须在一天之内做出这个产品特性。当时也就我一个程序员吧，一天的时间，这是当时能做出最好的方案了。” 说完，他便陷入了对美好时光的怀念里。

你也可能听说过这样的故事。

有一天你的 CTO 突发奇想，行云流水地提交了一段代码；大家一看很激动啊，很多人跑去观摩大神的代码，结果觉得问题多多，于是在PR（Pull Request）上提了一堆评论。

CTO 一看有点傻眼了：“几十条评论……现在代码要这么写啊，好麻烦。”于是他就和一位工程师说：“你把评论里的问题解决下，合并（Merge）到主分支吧”， 然后就开开心心地该干嘛干嘛去了。

这两个小故事是想说明一个道理：一个公司早期的代码会因为各种历史原因不是那么完美，但是，在特定的时间点，这就是当时最优的方案。

随着公司的发展，成品功能不断叠加，代码架构不断优化，系统会经历一些从简到繁，然后再由繁到简的迭代过程，代码的改动也会相当巨大，也许有一天，你会几乎不认识自己当初的作品了。

API 的设计和实现更是如此。在我们的工作中，很少能见到 API 的设计和实现从最开始就完美无瑕疵。一套成熟的 API，很多时候都是需要通过不断演化迭代出来的。今天我就和你聊聊 API 的设计和实现。

首先第一点，我们先从 API 的签名（Signature）说起。

# API 的签名（Signature）

API 的签名，或者叫协议，就是指 API 请求（Request） 和响应（Response ）支持哪些格式和什么样的参数。

首先，做过 API 的人都知道，一个上线使用的 API 再想改它的签名，会因为兼容性的问题痛苦不堪。因此，API 签名的设计初期，一定要经过反复推敲，尽量避免上线后的改动。

除了一些基本的 RESTful 原则外，签名的定义很多时候是对业务逻辑的抽象过程。一个系统的业务逻辑可能错综复杂，因此 API 设计的时候，就应该做到用最简洁直观的格式去支持所有的需求。

这往往是 API 设计中相对立的两面，我们需要找到平衡。有时候为了支持某一个功能，似乎不得不增加一个很违反设计的接口；而有时候我们为了保证 API 绝对规范，又不得不放弃对某一些功能的直接支持，这些功能就只能通过迭代调用或客户端预处理的方式来实现。

这种设计上的取舍，通常会列出所有可行的方案，从简单的设计到繁杂的设计；然后通过分析各种使用实例的频率和使用某种设计时的复杂度，从实际的系统需求入手，尽可能让常用的功能得到最简单直接的支持；还要一定程度上 “牺牲” 一些极少用到的功能，反复考虑系统使用场景，尽可能获得一个合理的折衷方案。

# API 设计原则

在这个折衷的过程中，我们需要始终保证满足这些基本原则。

1. 保证 API 100% RESTful。RESTful 的核心是： everything is a “resource”，所有的行为（ Action ）和接口，都应该是相应 Resource 上的增删改查（CRUD）操作。如果脱离这种设计模式，一定要再三考虑是不是必要？有没有其他方案可以避免破坏 RESTful 风格。
2. 在请求和响应中，应该尽可能地保持参数的结构化。如果是一个哈希（hash），就传一个哈希（不要传 hash.to_string）。API 的序列化和反序列化机制（ Serialization / Deserialization ）会将其自动序列化成字符串。多语言之间的 API，比如 Ruby、Java、C# 之间的调用，通常都是在序列化和反序列化机制中完成不同语言间类型的转换。
3. 认证（Authentication） 和安全（ Security ）的考虑。 安全的考虑始终应该放在首位，保证对特定的用户永远只暴露相关的接口和权限。可以使用证书和白名单，也可以通过用户登陆的证书（ Credentials ）生成的验证票据（ Token ），或者 Session / Cookie 等方式来处理。此外，所有的 API 层的日志（ Logging ），要保证不记录任何敏感的信息。
4. API 本身应该是客户端无关的。也就是说，一个 API 对请求的处理尽可能避免对客户端是 移动端还是网页端的考虑。客户端相关的响应格式，不应该在 API 中实现。所有的客户端无关的计算和处理，要尽可能在服务器（ Server） 端统一处理，以提高性能和一致性。
5. 尽可能让 API 是幂等（Idempotent）的。关于幂等，可以参考我之前写的“聊聊幂等“一文。这里面有几个不同层次的含义。举例说明：同一个请求发一遍和发两遍是不是能够保证结果相同？请求失败后重发和第一次发是不是能保证相同结果？当然，要不要做成幂等，具体的实现还要看具体的应用场景。

# 使用好 **API** 框架

每个语言都已经提供了很好的 API 框架，你需要在设计前先多了解这些框架。如果你是个小团队，资源没那么充分，选一个合适的框架入手，适当调整，比从零开始造轮子要好得多。等公司长大了，由于各自业务逻辑的特殊需求，最终都会定制一套自己的 API 实现方案。

评估一个 API 框架，可以从以下几个方面考虑：

1. 对访问权限的统一控制
2. 自动测试的支持
3. 对请求和响应的格式，以及序列化和反序列化（ Serialization 和 Deserialization ）的支持
4. 对日志和日志过滤（Logging 和 Logging Filtering ） 的支持
5. 对自动文档生成的支持
6. 对架构以及性能的影响

# 设计中的平衡

API 设计中存在很多对立的因素，比如简洁还是繁复，兼容性和效率，为现在设计还是为未来打算等等。根据自己的工作实践，我给出以下观点供你参考：

**1** 自由总是相对的

就好像在一个群体里，如果没有规则，完全行为自由，就会出现各种问题。小群体还好，而对于一个大群体，有人就会被别人的”自由“误伤。

写软件也是一样。一个小的创业公司里，API 怎么设计，代码怎么写，几个人一协商，达成共识，并不需要那么多的条条框框，也照样行的通。

公司越大，代码协作的人越多，个人的自由就会在设计和实现中产生问题，并导致最终的冲突。所以，很多大公司会制定一些 API 的最佳实践，强制要求设计和实现中必须按照某种模式来做。

有些规则虽有道理，但也不是说不这样不行，所以在很多时候，因为这样的规则，我们的 API 设计中会有很多限制，这在表面上似乎给设计带来无谓的难度，但是仔细考量，从规范代码和设计一致性的角度而言，还是有很大好处的。

**2** 为当前设计，还是为未来设计？

API 设计里很常见的一个情况是：一个目前并没有人使用的系统功能，它的存在只是因为有人提出：“这种情况我们以后应该要支持。” 前文中我曾讲过，由于 API 上线后再改很困难，所以在设计初期就要尽可能地考虑未来的发展；但是这些 “可能” 的应用场景因为需求的细节和使用频度都不明确，最容易造成系统的过度设计（ Over-design ）。

我记得有一个 API 设计的经典原则，概括一下就是：要考虑未来的场景，在设计时留有余地，但永远只实现当前产品真正要用的功能。

**3** 可维护性和效率（ **Maintainability v.s. Efficiency**）

设计和实现里常常会有一些封装和抽象的概念。某些特殊情况下，封装再分拆的过程可能会在一定程度上影响 API 的响应速度，或者代码质量的优化和性能的优化上有冲突。这个很难一概而论，具体的做法要看代码是否在关键路径上，或者这段代码是不是需要多人协作等等。最终的选择就要具体问题具体分析了。

**4** 是否采用**AOP**

AOP 本身就是一个极具争议的话题。概括说来，AOP 的理念是从主关

注点中分离出横切关注点。

分离关注点使得解决特定领域问题的代码从业务逻辑中独立出来，业务逻辑的代码中不再含有针对特定领域问题代码的调用，业务逻辑同特定领域问题的关系通过侧面来封装、维护，这样原本分散在在整个应用程序中的变动就可以很好地管理起来。

因为 API 的设计和实现中有很多通用的关注点，如日志（Logging）、解析（Parsing）、监控（Monitoring）等等，所以 API 成了 AOP 一个很自然的应用领域。

使用 AOP 的 API 设计继承了 AOP 的优势，如：代码的重用性，规整性，以及程序员可以集中关注于系统的核心业务逻辑等；但也会自然而然地继承了 AOP 固有问题，例如代码的剖析（Profiling）和调试（Debugging）困难增加，对程序员的相关经验有更多要求，相互协作的要求也增强了，比如改变某一个功能可能会影响到其它的功能，等等。

是否选择使用 AOP，和你的需求场景，人员技能和设计复杂度息息相关，需要技术决策者根据具体环境做出判断。

今天我从两个小故事入手，和你讨论了 API 的设计和原则，内容分为四个部分：API 的签名、API 的设计原则、使用现有编程语言的 API 框架、如何在 API 设计中取得平衡。

API 设计是现代软件系统中不可或缺的一个环节，不同的系统需求和不同编程语言下，API 的设计都大不相同，但总有一些原则和注意事项是可以提取出来的，今天我和你讨论的就是这些通用的原则，希望对你的实际工作有帮助。

最后，给你留一道思考题，API 的签名（Signature）设计是语言无关的，那你在设计中会引入更多的语言还是更少的语言去实现不同的 API 呢，优点和缺点各是什么？期待你的回复，我们一起进步。下期再见。


戳此获取你的专属海报

# 硅谷面试：那些你应该知道的事儿

一天，一位朋友偶然问我：“听说 Airbnb 的面试一共有七八轮，这是不是真的？”

我回答这是真的，而且告诉他在北京的招人流程也基本是一样的。朋友却觉得没有必要，认为一轮就能面出应试者的水平，弄那么复杂的面试是在浪费工程师的时间。

对于他的看法，我不完全赞同，也不完全反对。其实硅谷的公司大部分都会有多轮的面试，这主要包括了算法编程、系统设计、工作经验等。

首先，我就来说一说，为什么要有那么多轮的面试。

算法编程主要考察编程的硬功夫。现在很多公司都要求现场上机，程序能运行，测试例能通过；还有一些公司使用白板考算法，于是才有了 Homebrew 的作者马克斯·霍威尔（ Max Howell ）去 Google 面试时无法在白板上翻转二叉树的故事。

系统设计主要是给候选人一个系统设计的场景，根据自己的能力和经验去架构整个系统。系统设计面试通常没有唯一解，侧重的是候选人对不同架构和解决方案的熟悉程度，对方案的优缺点的深入了解。

工作经验面试主要是与候选人聊他做过的项目，深入了解和挖掘候选人的技能树。类似 Airbnb 这样的公司还会有关于价值观和团队协作能力的测试。

这样看来，一轮面试是远远不够的，因为有很多工程师程序写得又快又好，但是在系统设计方面会表现的很差，对一些基本的设计理念和方法知之甚少。

还有一些人，具备基本的系统知识，但想问题不全面，不能自己考虑各种场景的设计需求和异常处理，需要提示才能发现系统的漏洞。

又或者，有的人沟通能力很差，无法准确地表述出问题和答案，如果是平时的工作交流，沟通能力欠缺会给团队合作带来很大的阻力。当然，也有一些人恰恰相反，谈起设计和理论来头头是道，口若悬河，但是一写代码，几乎步步维艰。出了错，也不知道如何有效地调试（Debug）。

也有人说，可以知人善任。特别能写程序的，就让他多写代码，特别能说的，就让他负责沟通相关的工作，这样不就皆大欢喜了吗。

其实，这样做是行不通的。一来，硅谷优秀的工程师大部分能够独当一面，他们能把工作做好，确实需要用到方方面面的技能，这也是效绩评估中考察工程师的基本要求；二来，好的公司和岗位根本不愁招不到

人。你说自己代码写得特别好，但还会有很多写代码能力跟你不相上下，但是系统设计比你牛很多，所以你被拒的可能性还是很大。

写到这估计还会有人有疑问：即使这样，那一轮代码面试，一轮设计面试也够了啊，为什么每项需要两轮甚至三轮面试呢？

我的经验是：因为不同题目考察的知识点和解决问题能力的侧重点不同，两轮可以更全面地进行评估；更重要的是，面试官也是人，难免有一些主观看法，进行两轮面试，两个人的反馈意见就可以相互有个参考，比如应试者是不是两轮都出现同样的问题或者失误，对同一领域的不同问题是不是都有正确的见解等。

有的时候，两轮面试官意见相悖，加试一轮的情况也并不少见。此外，有的时候我们因为面试人手不足，也会不停地增加一些新的面试官。这些面试官经验不足，如果他们的反馈成为唯一依据，多少会产生一些冤案错案，进行交叉面试，可以尽可能避免这种情况的发生。

再说说刷题的问题。如果你觉得写程序手生，或者一些基本概念需要复习，刷题可能是有帮助的；但是大多数情况下，除非你是跨领域面试，否则每天的工作就是系统设计和讨论，编写代码，功底不至于完全丢下，如果纯粹靠刷题通过面试，后面工作可能还是会比较吃力。

当然，努力刷题对通过面试肯定是很有帮助的，毕竟大部分公司的题库都比较有限，有些题用了很多次被泄露出去，你来面试前做一遍肯定会占便宜。

说到了题库，我有一点对应试者的小建议。很多应试者在面试结束后，喜欢在网上分享题目。抛开道德层面，这种行为确实是极其“大公无私”的；但如果你拿了 Offer 去泄题，后面不如你的人可能会成为你的同事；而如果你没拿到 Offer 去泄题，后面不如你的人可能因此拿到你想要却拿不到的 Offer。

除此之外，面试官并不像想象的那么好骗，应试者是不是见过题目，面试官其实很容易看出来。有时候我们在反馈面试意见的时候，如果觉得有些题你之前做过但是没说，也会如实反馈。这种意见很可能会影响你能不能拿到 Offer。另外，即使是同样的题，是不是真的懂，有经验的面试官也很容易问出来。

那么，作为应试者，面试官的经验会对你造成什么影响呢？总的说来，越是经验不足的面试官，他可能更容易把思路局限在一个自己很熟的领域，如果你凑巧和他的思路一致，就有可能拿到很好的反馈。如果你提出了一个完全不同的解题思路，面试官反而不能十分理解，面试成绩上有偏差也是可能的。

面试中还有一个潜规则，就是很多题都有难度等级，也就是分梯度的。你解出了第一部分，才会看到第二部分的问题，答出了第二部分，才有机会看到第三部分。

如果一开始你就比较慢，或者解题比较挣扎，可能面试官就会直接放弃你，虽然他在当时没有明确拒绝，但心中已经给了你 NO，所以剩下的时间，即使你觉得自己把第一部分答得很完美了，然而真相是你根本不知道还有二和三，也就不知道为什么觉得自己把题解出来了，最后却拿不到 Offer。

最后说说怎么才能拿到面试机会。

很多人会觉得找人内推拿到面试的机会一定更大，其实并不是这样。别的公司不好说，但是像很多比较热门的科技公司，员工都会收到很多并不熟悉的人的内推请求。比如我自己，平均一周要内推两到三个人。

公司知道这种情况很多，所以内推系统都会设计一个选项“你是否和被推荐者共事过”，如果是，你能不能替他的能力担保？如果没有工作交集或不能担保，即使内推了，你的简历也会得到和普通简历一样的待遇，唯一的优势可能是简历不会丢，处理速度稍微快一点而已。HR 筛选简历的所有条件，并不会因为内推而降低标准。

所以平时具备扎实的基本功和出色的业绩，简历上有闪光点，才是拿到面试机会的关键。真的找人推荐，最好找那些共事过的能为你能力背书的人。

今天，我和你聊了聊硅谷面试的那些事儿，比如硅谷为什么会有那么多轮的面试，面试前要不要刷题，面试过程中与面试官的交流，面试题的难度分级和内推的作用。如何拿到一个面试机会并通过面试呢？其实没什么捷径，好的工作经历、扎实的基本功和一份特点突出的简历，就是工程师成功通过面试的金钥匙。

你有什么有趣的面试故事，也在留言中说说，我们一起分享。我们下次再见。

戳此获取你的专属海报

# 项目管理中的三个技巧

我在“管理者不用亲力亲为：关键是什么”一文中，介绍了授权和任务分配的重要性。那篇文章的重点有两点：第一我们要有效地把任务分配出去，第二我们要保证分配出去的任务能够被圆满完成。

作为管理者，我们平时在项目管理的过程中，更侧重的是要保证团队成员能够按照你的期望值完成任务。今天的这篇文章里，我会进一步展开讲一些项目管理的技巧。这些技巧一部分来自我个人的思考和实践，另一部分得益于我的老板的悉心指导和启发，真实可行并且有效，希望对你的日常管理工作也有帮助。

# 第一个技巧，我们在做项目计划的时候，要对多个项目进行细分重组

怎么理解呢？我们从做这件事的目的说起。在给组里多个人分配项目的时候，我们往往需要考虑的因素包括以下的内容。

1. 先评估能力，再分配任务，每个人的能力要和任务的难度匹配。
2. 每个人任务完成所需时间要尽量平等，也就是要达到一种负载平衡。
3. 每个人得到的任务里，挑战有意思的工作和脏活累活的比例要大致相等。
4. 每个人任务里有足够的挑战，能够帮助其成长，又不至于太难而让其望而生畏并产生挫败感。
5. 不同人的任务之间如果有依赖性，在分配任务时要安排合理的顺序，确保不会有人被别的人或事阻塞（Block）。
6. 每个人的任务里都应该有一个主题，就好像故事有一条主线。这样，成员会觉得自己参与了一个比较完整的任务，进而产生成就感，而不是感觉做了一堆杂活。

达到这些目的的手段，我们姑且就称其为**“**细分重组**”**。这个过程又包括两个阶段。

第一个阶段。你需要把所有要做的事，细分成一个个的小任务，每个任务的大小、完成需要的时间都大致差不多。如果有比较大的任务块，就尽可能地切分成几个小块。这需要管理者对项目本身的重点和任务细节有很好的把握。

第二个阶段。把这些大小均匀的任务块，按照上面提到的因素，分装到几个虚构的 “箱子” 里，然后分配给团队成员。这就像个打包装箱的过程，尤其需要注意的是，每个箱子一定都有一个主题，也就是说，如果你想给这个箱子起个名字，你一定能找到那个名字，并很好地概括其中的内容。最后，保证每个箱子在内容、重量等各方面都比较均衡。

完成了这个工作之后，后续项目的每一步，作为管理者的你都能做到心中有数。同时还能避免后期执行中一些可能的弊端，例如，有的人工作繁重疲累不堪，有的人则早早完成了自己任务，缺乏挑战。这种任务划分的方式还会让每个人更有成就感和责任感，因为他们完成的是一整个故事。

# 第二个技巧，工期估算

一般情况下，技术管理者都会对每个任务的完成时间有自己的判断，但最终还是要和接受任务的员工沟通清楚，并尊重对方的意见，确保双方能就任务时间线达成一致。有了承诺，工作的目标性也会更强一些，毕竟，截止日期才是最好的效率工具。

这一点非常重要。如果不是双方达成一致的协议，或者不是双方都认可合理的时间估算，一旦后期出现不能按期完成任务的时刻，就很容易出现一些令人不愉快的交流。

同时需要注意的是，很多工程师在做时间预算的时候，会过于乐观。我见过的大部分工程师都乐观、积极、自信，他们沉浸在代码的世界里，试图把软件做到最好，往往却会忽略时间因素。当核对工期的时候，他们会根据自己的经验给出一个非常乐观的期限。

和普通人一样，工程师们也会高估自己编程能力和对复杂逻辑的处理能力。甚至，有时候工程师给出的工期是自己负责的那部分程序编写完成的时间；然而，一个功能的完成，包含编译、单元测试、提交代码、集成测试、功能测试、性能测试和上线。如果不是特别留意，这些细节往往会泯灭在项目进度的时间表里，无法体现。

即使是一个很有估算经验的工程师，在新项目中也可能会遇到各种各样新的问题，你会惊奇地发现，上一个项目中的方法在新产品中失灵了。另外，开发中遇到的技术瓶颈或难以解决的 Bug，也会耗费程序员大量的精力和时间，这时候我们能做的事情只有等待，给他们时间去披荆斩棘，直到问题解决。

所以，在这个阶段，往往需要技术领导给出参考性的意见和建议；除此之外，你最好留出一些缓冲的时间，因为实际工作中总会有一些不可预见的情况发生。

# 第三个技巧，也是很重要的一点，实时跟踪，并准备好 B 计划

技术领导者要做好两手准备，比如，团队中有一部分人突然表现失常了怎么办？项目由于其他原因被阻塞了怎么办？

这时候我们需要做好以下两点。

1. 我们在 “细分重组” 中把工作分成了小块，在完成过程中，我们还需要设立各种里程碑。其中，有一些长期的大里程碑，也有一些为期一周到两周的小里程碑。这些里程碑就像你上高速行驶之前给自己定的目标：几点前要到某个服务区，几点前要到某个城市等等。有了这些里程碑，管理者就可以通过它们进行实时跟踪，了解项目的进度，看看项目这辆汽车是不是还正常地行驶在高速路上，是不是抛锚了，是不是没油了，等等。
2. 一旦出现延迟，管理者要和任务的负责人一起分析原因，询问对方能否追上进度，会不会对整个项目的进程有重大影响。如果问题不严重，可以暂时不做调整，继续跟进。如果影响比较大，就需要启动 B 计划了，比如调整执行的人员、提供额外的资源、分析执行的方法、调动其他组支援，甚至你需要重新考虑项目进度。

今天，我和你讨论了平时在项目管理过程中的一些实践技巧，总结一下：管理者首先要对大的项目进行细分重组，“打包装箱”之后再分配下去；其次，在工期估算方面，管理者要和任务的负责人达成一致，并且要注意到，工程师们在进行时间预算的时候都是比较乐观的，最好为项目预留缓冲的时间；最后，要为项目设置大大小小的里程碑，并实时跟进，一旦项目出问题，就要启动 B 计划。

通常情况下，如果这三点做得比较好，我们的 B 计划就不会用上，这也是我们期待的最好结果。

你在项目执行过程中还有那些经验和技巧呢？可以在留言中告诉我，我们一起进步。

感谢您的收听，我们下期再见。


戳此获取你的专属海报

# 每个工程师都应该了解的：中美在支付技术和大环境下的差异

支付是很多互联网行业和产品不可或缺的一部分，我在支付领域工作了很多年，也积累了不少经验和教训，今天，我就来谈一个看起来比较宏大的话题：“中美在支付技术和大环境下的差异”，与你共同探讨一下支付领域的技术和生态，以及支付在美国和中国的差异性。希望这篇文章可以对你有一定的启发。

# 概况

当我们谈支付系统的时候，一般会这样定义：一笔钱从甲方转移到乙方。这笔转移可能是商家和顾客间的货款交易，也可能是个人对个人的转账，支付系统就是让这笔交易或转账发生的系统。这中间可能是银行间的转账，也可能是一个虚拟的闭环系统，比如在支付宝、PayPal、甚至比特币系统中的价值转移。

用到支付的地方很多，比如传统的柜台、店家销售、电子商务和远程商务、账单支付、个人对个人（P2P）、公司对公司（B2B ）、薪资发放等，这些都要使用到不同的支付方式。

支付可以由收款方发起，这时候我们需要知道付款方的账号信息，比如在商店里刷信用卡；也可以是由付款方发起，这时候我们需要知道收款方的账号信息，比如公司直接往员工银行账户转账。

支付系统在处理各种交易中，不仅仅是处理交易，完成钱或者价值的转移；更多的时候，它也需要其他功能的支持，这包括了：制定法律法规、确定规范和程序、处理异常 ，还有收费的标准、一笔交易应该多久保证能完成等等。

支付方式不同，这些标准也会有所不同。比如银行卡的交易手续费通常比较高，自动交换中心（Automated Clearing House - ACH）交易完成的时间比较长，等等。

# 市场占有

在美国，传统的支付方式会包括：

1. 各种卡：如借记卡、信用卡、预付（Prepaid）卡
2. 自动交换中心 ACH：这是一种美国的转账系统，手续费远远低于各种卡交易。
3. 支票
4. 现金
5. 电汇（Wire Transfer）
6. PayPal 等通过绑定银行卡的快速支付或手机支付

除了现金支付无法追踪，以上的各种支付方式都可以进行数据追踪和分析。从近些年交易笔数来看，近一半是借记卡交易，约 20% 是信用卡交易，而余下的交易由 ACH 和支票平分秋色。从交易总额度来看，最多的为 ACH，占 60% 以上，其次为支票，约占 25%，而信用卡、借记卡因为手续费比较高，总交易额度只各占约 3%。

这和中国的情况很不一样。早些年卡的交易笔数在中国要低得多。自从支付宝、微信支付成为主流支付方式后，绑定卡的闭环系统交易数量大大增加。

不论是银行系统，还是各种新型的支付方式，都会希望尽可能地占有市场份额。这其中主要有三个原因。第一，交易手续费会是一个很大的盈利方式；第二，资金的流动会形成一个资金池，有用来投资的可能；第三，当在支付生态系统中占有了很大的用户份额后，就更有机会基于这个用户群体进行下一步的产品开发和推广，比如 Square、亚马逊支付（Amazon Pay）就属于这一类。

在市场份额的竞争中，支付技术带来的处理速度、安全性、使用的便捷性、国际化等因素，都是竞争力的主要体现。

# 钱包系统

钱包系统是指用手机作为载体，通过一个 App 实现虚拟 “钱包”。比如 PayPal，支付宝、Square Cash、星巴克钱包、谷歌钱包、苹果支付（Apple Pay） 等等。

钱包实现中常用的技术有：

1. 近场通讯（ NFC）: 一种手机芯片，可以和支持射频识别（ RFID ）的终端进行交互。苹果支付、安卓支付 、三星智付（ Apple Pay、Android Pay、Samsung Pay ）都使用了 NFC 技术。
2. 射频识别（RFID）: 一种磁卡和终端交互的无线通信技术。
3. 二维码（ QR Code）: 一种用于存储数据的机器可读的标识。
4. 磁性感应（Magnetic Induction）: 模拟读词条时磁场的一种无线信号。
5. 安全芯片（ Secure Element）: 手机中可安全存储数据的一种芯片。
6. 主机卡模拟 （HCE）: 安卓手机中使用软件对智能卡进行虚拟而精确的呈现。
7. 客户验证数据（ Customer Verification Numbers CVNs)：关联用户银行卡的一个由磁卡或芯片卡产生，能被银行系统识别的一个数字。
8. 令牌化技术 （Tokenization ）：和卡号关联的一个标识，降低把卡号公开的风险。

关于技术细节，由于时间的关系，我这里就不多做详述了。

除了这种和银行卡绑定的钱包，还有一种是 App 内预充值的钱包，比如星巴克的钱包。这种钱包只能应用于某个特定的场景，星巴克钱包就只能在星巴克连锁店里使用。星巴克的这个移动应用实际上更像是一个数字礼品卡，可以关联银行卡方便充值和支付，每次到柜台通过扫描二维码发起交易，星巴克的 POS 交易中，有约 20% 的支付是通过这种手机钱包来完成的。

# PayPal、支付宝和微信支付

## 1 PayPal

PayPal 也叫贝宝，主要在美国使用。用户使用电子邮件来唯一标识自己的身份，通过关联信用卡或借记卡在用户之间进行资金转移，在各大电商网站使用广泛。手续费用较高，支持多个币种间的转换。总的来说对付款方保护比较多，对收款方限制比较多。其主要产品和技术包括了 PayPal Pro 在内的十余种产品（ PayPal Payments Standard / Advanced / Pro、Payflow Gateway、Express Checkout、Invoicing、Send Money、Mass Payments、X.Commerce Platform等）。

此外，基于 PayPal 的 Venmo，PayPal Here，PayPal Beacon，PayPal One Touch 等产品也有不少用户。PayPal 交易中，关联信用卡的交易约占 60%，借记卡约占 25%，使用钱包额度的约占 15%。

PayPal 使用灵活，有很好的风控系统，支付确认速度比较快，并且还支持多币种和全球范围的大数据等，这些优点让它在美国支付领域逐渐积累了很大的用户群，用户的数目超过两亿。它最早主要是在易贝（eBay）上使用，现在商家已达百万级，支付交易数上亿级，支付总额度达千亿级，虽然 PayPal主要应用在电商，但也发展了很多的新型市场。

PayPal 的集成和开发也很成熟。比如 Braintree 就有 PayPal 的 SDK，为很多电商和共享经济平台使用 PayPal 提供了的支持。每天约有百万级以上的 API 调用。

专家预计，在未来至少十年内，PayPal 仍然可以保持 15% 以上的增长率。

## 2 支付宝

支付宝是对应于 PayPal 在中国使用很广泛的支付方式。对于国内读者来说，很多人应该比我更了解支付宝，所以，在这里我只简单地做一下介绍。

支付宝的创立最初是为了解决阿里巴巴淘宝网的交易安全问题，并首先使用了 “第三方担保交易模式”，让买卖双方的资金和商品都能获得系统性的保护。2004年，支付宝独立成为公司，逐渐向更多的合作方提供支付服务，并发展成为中国最大的第三方支付平台，目前隶属于蚂蚁金服。

2016年，支付宝有了 4.5 亿实名用户，71% 的支付笔数发生在移动端。2016 天猫双 11 全球狂欢节总交易额超 1207 亿，其移动端占比82%，支付宝的支付峰值达到12万笔/秒。这些都是极其庞大的支付数据。

到了今天，支付宝已经发展成为以支付业务为核心的生态环境，形成了一个提供生活服务的平台。支付宝这么强大，是不是已经无敌于天下了呢？

如果没有微信支付的话，支付宝确实可以独步江湖，只是“既生瑜，何生亮”，两大支付不得不平分秋色。下面我就来聊聊微信支付。

**3** 微信支付

微信最初是个即时通信工具，后来发展成一个庞大的社交平台，有了几亿的用户基数之后，做支付就变得势在必行。2014年，微信几乎是以必须进入的态势推出了支付功能。2015年春节，微信支付联合商家以“摇红包”的形式送出了5亿金额的现金红包，一举奠定了微信在支付领域的地位。

微信依靠简洁安全的支付方式，庞大的日活用户，节制的商业态度，很快成为支付宝最大的竞争对手。2016年支付宝和微信的支付总额达 3 万亿美元，微信的支付总额达到了 1.2 万亿美元，基本上形成了两家独大的垄断地位。今年的数据还没出来，我估计微信支付会进一步缩小与支付宝的差距。

支付宝和微信支付我个人都使用过，整体来说，这两款产品对海外用户的支持上远不如国内用户那么方便，希望未来能够更好地优化海外用户的支付体验。

# Square的产品与技术

Square 是杰克·多尔西（Jack Dorsey）和他的朋友在 2009 年共同创办的。

Square 系列包括 Square Cash 在内的多项产品（Square Register、Square Stand、Square Capital、Square Wallet（已停止）、Square Order（已停止）、Square Store、Square Capital、Caviar（订餐服务）等）。我在 2013-2015 年间作为软件工程师曾就职 Square，参与了包括 Square Wallet、Square Market、Square Store 等多个项目的工作。

Square 在技术上有自己的风格，这包括了使用音频输出口接入磁卡和芯片卡的读卡器，生成方便快捷的电子账单，指纹签名等多种创新。

Square 凭借扎实的支付技术和各种对中小商家交易费用的优化，非常受欢迎。EMV 卡新技术出来的时候，Square 也在第一时间研究了读卡器并对其加以支持，因此在移动读卡器领域一直属于领先地位。

Square 读卡器是一种新型的移动端 POS 系统。用一个小小的硬件即插即用到手机或者平板的音频输出口上，就可以像传统收银台一样支持银行卡交易。这种模式在美国中小商家曾风行一时，但是在中国市场反馈并不好，比如类似产品拉卡拉。我想这其中的原因，大概是国内直接携带并使用信用卡的人群并不是支付主流。

Square Cash 是一款个人对个人（P2P）或者个人对公司（P2B）的产品，通过每个用户或者商家定义自己的 “$Cashtags” 就可以轻松转账。因为在其技术实现中使用了一些技巧，所以它的手续费远远低于类似产品。

Square Stand 是一个可以和 iPad 集成的读卡器，并有相关的配件如 Cash Drawer，Printer 等。Square 所有交易中，iPad 上占约 60%，因此 Square Stand 一直是其主流产品。它的设计特别精美，和苹果的风格很像，也很搭配，但是制作成本很高。Squre 为了推广这一款明星产品，长期以低于成本价的价格销售。不过只要后期交易手续费足够高，这个本钱还是可以赚回来的。

虽然很多人认为 Square 是一家支付公司，但是其实杰克·多尔西一直把自己定位为一家贸易（Commerce）公司。Square 很注重数据和数据分析，风控系统也做得很好。

# 支付技术的发展趋势

1 可穿戴设备的支付。

随着软硬件技术的发展，开始出现可穿戴设备的支付方式。可穿戴支付使用的技术包括无线 BLE 及智能手机和 Beacons。比如微软智能手环（Microsoft Band）、PayPal Pebble、苹果智能手表 （ Apple Watch）、谷歌眼镜 （Google Glass） 等，都可以一定程度上绑定手机完成支付。

2 支付正变得越来越快。

所有的支付方式不论在验证，交易处理，还是记账、收据等环节都在让交易更省时更方便。最明显的就是 ACH 速度更迭越来越快。

3 虚拟货币的兴起。

如比特币、数据区块链（ Block Chain ）记账法等。这些新型货币更多地用于一些货币不稳定的国家、跨国贸易和非法交易等。用于日常生活还会有多挑战，也会有很长的路要走。

关于比特币，可以参考我早期的一篇文章进行阅读：

http://mp.weixin.qq.com/s/npIosR1C5dfBB5wAAOGxUQ

# 总结

线上支付系统是一门巨大的生意，支付领域的战争正在全球如火如荼地展开，这个领域内的方方面面绝不是我一篇文章能写完的。

今天，我主要是从一个支付从业者的角度，和你聊了支付领域的技术和生态，包括支付的概况、支付的市场，接下来我讲了讲钱包系统相关的技术点，举了 PayPal、支付宝和微信支付的例子，并简单介绍了 Square 的技术和产品介绍，最后简单展望了支付技术的发展趋势。

在现代社会，几乎所有互联网相关的业务模式都和支付有关，支付已经成为很多产品本身的一部分。这篇文章会帮助你了解一些支付的技术和生态，希望对你有所帮助。如果你们在研发产品的过程中遇到了支付相关的问题，也可以在留言中反馈，我们一起讨论解决，下期再见。

# 不要做微观的管理者

今天我和你聊一聊管理的粒度问题，在进入正题之前，我先来说说自己的故事。

第一个故事发生在我离开 Square 之前不久。那时候正值 Square 准备 IPO 的前期，公司频繁重组，很多人的顶头上司在短时间内可能更换了好几个，我也不例外，最后的那个老板，我和他的共事时间一共也就一两个月。他的主要背景是前端，而我一直做服务器端和系统架构相关的工作。成为我的老板的时候，他刚刚转为管理者。

不知道是新官上任三把火，还是他的管理风格使然，总之，做项目的时候，他恨不得手把手地指导我们，频繁地跟进和检查，很多并不算重要的技术决定，他也会给出自己的想法，而不是让组员做决定。

我个人比较偏好发挥主观能动性，这种做法让我在工作中束手束脚，做事方式也遭到质疑。当时我算是比较有经验的工程师了，加上性子有些执拗，也不喜欢去争论或沟通，工作氛围的变化让我有些疲惫，恰好 Airbnb 又在那时抛出了橄榄枝，种种因素集合在一起，促使我去面试。

面试的过程很愉快，我也很喜欢 Airbnb 的愿景和工作氛围，所以就顺势跳槽了。虽然说老板不好不是跳槽的根本原因，却让离开 Square 这个决定变得更加轻松。

虽然后来很多人问我为什么离开 Square，我都没有提这件事。不过，听说那位管理者的团队带着带着就没了，公司又让他做回了工程师，这是后话；其实他技术不差，做事也靠谱，但是管理方面，可能真的需要一些改变。

因为这件事，加上我本身对独立自主看得比较重，在走上管理岗位的时候，我会更偏向于给组员足够的自由度。对于我来说，你只需要跟我说需要做什么、为什么做、什么时候做完就可以了，我会有自己的计划和安排，也会对自己的进度、优先级和质量负责。

直到有一天，我和组内的一位职场新人聊天，他说，希望我能给他更多的指导或帮助，在完成任务的过程中得到我更多具体的支持。这算是另一个故事，不再细说。

但是，也在那个时候，我意识到管理既不能过度关注和掌控过多的细节，也不能简单地给了目标就放任自流。经过一段时间的思考、观察、阅读、交流和总结，我发现以下几个方面的度是需要管理者重点把控的。

# 一、因人而异

我们共事的同事，大部分都是很优秀的人。他们和你一样进入这个公司，每个人都经历了差不多的考察和审核，每个人的能力都是毋庸置疑的，但每个人的优点和缺点，擅长的领域，往往各不相同。

有些人在全局规划、协调资源方面有着更强的能力，有些人则注重细节，可以把定义好的任务完美完成。有的人善于思考，有的人擅长交流，有的人专注执行…… 我们要做的，是最大程度调动并发挥其长处，并帮助他在短板的方面获得更快地成长。所谓用人用其长，就是这一个道理，说起来容易，要真正做到，还需要不断的调整和练习。

另外，每个人都不是一成不变的。再有能力的一个人，刚进一个公司、一个组，接手一个项目，都需要时间和资源去学习相关的背景知识。如何给他们充分的资源、信息和指导，又同时给他们足够的空间，往往不能一概而论。

如果是工作多年的职业工程师，尽管各个公司的一些流程和工作方式不太一样，你只要给出指导性的建议和准则，他们就能很快调整适应。如果是刚刚从学校毕业的新人，那么从各种工具的使用、开发流程的熟悉、代码的规范化，都需要更多的指导。这样的人往往需要安排一个有经验的工程师去带，才能获得最快的成长速度。

# 二、因事而异

先问问自己，这个任务在整个项目中是不是很重要，是不是很紧急。如果是，并且每一步的完成有很强的时间限制，在 “怎么做”的方面会因为各种要求或者限制并没有太多的发挥余地，那么，这时候，更多的介入是你更好的选择，不过在介入之前，你需要让对方理解为什么需要频繁沟通。

如果单个任务在整个项目中有一定试错空间，或者不在时间线的关键路径上，这时，你不妨试着放手让组员尝试独立完成。这样，管理者这样才能够鼓励创新，并可以增强组员的工作积极性。因为组员全权负责任务，即使出了错，他也会更有责任感和足够的经验去改进。

也许有人觉得，我的整个项目时间都特别紧，任何一个点都不容出错，那么你就应该去思考，如何规划整个项目时间和人员的安排，尽可能去创造出一个可以让员工发挥的空间。如果什么也不能放手，除了对员工积极性有负面的影响，你的时间和精力，也将进入一个无法扩展的工作轨道。

## 三、跟进的粒度

关于跟进的粒度，最极端的两种做法是：只设立目标，然后完全放手，这样的做法是等你意识到事情已经脱离轨道的时候，往往很难拉回正轨或者补救；另外一种是每个细节都按照你的想法去推进，这就无法让员工发挥自己的能力。作为管理者，我们需要做的是在两种极端的中间点找到一定的平衡。

根据前面提到的两个原则，根据不同的人和事，我们应该做到下面几点：

**1** 制定目标，确保传达。管理者应该非常清楚地说明你想要的最终结果是什么样子，并明确地告诉员工：他的责任是达成什么目标，你会通过结果来衡量他们。

**2** 多给指导，少亲手做。管理者无需事必躬亲，也不要监视每一个细节。你只需要花一些精力确保你的意图已经被明确传递了，管理者的存在不是分担实际工作，而是一种资源，更是背后支持力量。

**3** 设定频率，保持跟进。根据对方的经验和任务的紧急程度，设定一个常规跟进的频率。每次跟进了解工作进展，是否有拦路石需要帮忙清除。除此之外，给对方足够的决定权。

**4** 交流难点，给出建议。当进度受阻的时候，你需要与员工交流，搞清楚问题所在，是能力问题、资源不够、还是时间紧张。根据交流成果，你可以给出一些建设性的建议和意见，帮助其提高能力，但不要直接上手帮忙完成任务。

## 四、交流的重要性

在这门管理课中，我曾经多次提到交流的重要性，其实，交流的重要强调多少次也并不为过。人和人之间的交流太容易产生信息丢失，很多时候我们都以为双方对交流的信息达成了一致，结果确认的时候发现完全不是一回事。

我在前面讲的三个原则，其中最关键的就是让双方完全明白对方的期望值。你可以很坦诚地告诉组员，如果他觉得你介入太多或者太少，可以随时告诉你。另外，你提出来的哪些是要求、那些是建议，也要明确地告诉对方。要求是那些必须要按时完成的，没什么回旋余地的工作，建议就是有空间的，对方可以在思考后自己决定怎么做的任务。

总结一下今天的内容，我从自己的故事谈起，与你分享了管理的粒度问题，在管理的过程中，既不要事必躬亲，也别做甩手掌柜。管理者需要重点把控四个方面，根据具体的人和事采取不同的管理方式，适当调整

跟进的粒度，并时时交流，确保双方信息一致。

最后提醒你一点，每个人都在成长。一个月前，你的组员在某件事情上喜欢你的建议和意见，一个月后，也许你这样的建议和意见就可以减少了。用动态的眼光去看待每个人的能力，调整你的跟进粒度和授权范围，确保任务在正常完成的同时，每个人的能力和创造性都得到发挥。

你是个微观管理者吗？可以在留言中告诉我你的做法。感谢你的收听，我们下期再见。

极客时间祝你元旦快乐，新的一年诸事顺利。


戳此获取你的专属海报

# 如何处理工作中的人际关系？

当我们初入职场的时候，职能级别可能比较低，大部分工作会比较独立，即使不是特别善于处理各种人际关系，问题也不算大，一来用不着你来协调资源，独立工作会占主要部分，很多时候都处于闷头写代码的状态；二来，组里会有一些老员工，可以帮助你做各种沟通工作。

但是，若想职场进阶，一直往前走，到了某个阶段，比如需要你去做更多的协调和沟通工作而不是仅仅写代码的时候，如何处理人际关系就成为了一项必不可少的软技能，重要性不比技术上的硬技能低。

今天我和你聊聊如何处理工作中的人际关系。

在文章的开始，我先来说几个情景，你可以参考一下自身在工作中有没有遇到过这样的问题。

> J 总是很耐心，你什么时候去问他问题，他都会细心解答。
>
> H 人倒不错，不过什么东西都说不清楚，问他东西特别费劲。
>
> A 总是特别有主意，你很难说服他。在审查项目或代码的时候，有时候他给出的建议非常好，有时候就会比较主观，如果不按照他说的做吧，就得花时间说服他，那就会比较烦。
>
> W 喜欢和大家讨论问题，无论是公开场合还是私下交流，不过讨论完之后他就会写文档或者发 Email，把大家讨论的想法或主意说成是自己的，有点喜欢抢功劳的样子。

以上的情景告诉我们，在工作中，我们总会对一个人产生感性的判断。比如：这个人很好相处，这个人很难合作等等。

这一点其实无可厚非，但是在工作中，我们需要尽量摆脱自己的感性判断，尽量平和地对待别人，处理好工作的人际关系。

生活中亲疏有别，有些人不愿意理也就懒得交流，但工作中的很多交流是不可避免的，所以常常需要比生活中的关系更谨慎对待。如果在工作中和某人闹僵了，后续的合作就会变得非常困难。

总之，人际关系是一个特别复杂的话题，因人而异，更因情形而异，那么如何处理人际关系呢，在这里，我有几点建议，内容可能更偏向于个人感悟，仅供你的参考。

首先，对于自己的上下级，保持开放的心态和愿意沟通的态度十分重要。

自己的上下级是职场中最无法避免的人际关系，因为各人性格不同，可

能你和有些人更容易在看法和决策上达成一致，而有些人则经常不能很好地理解对方意图，需要多次沟通，采用对应的沟通方式。上下级之间的利益大部分时候是绑在一起的，如果处理不好，对彼此的工作都有很大的影响不说，很多时候还会发生不愉快的事情。

其次，在交往过程中，尽可能对别人的分享、工作、交流持一种积极、友善和鼓励的态度。

我们常常把处理人际关系的态度分成三类。第一类是给予者，这一类人，不怎么计较得失，总是尽最大可能帮助和支持别人。第二类人秉持对等关系，谁对他好，他就对谁好。自己帮了别人一次，或者别人帮了自己，都是一个人情，都会记到内心的小本本上，那是一个无形的账本。第三类人是索取者，这种人只会与对自己有帮助的人接触，做的事也都是从自身利益出发。

当提到这三种人时，我们脑海里可能会浮现出一些人物形象并对号入座，甚至还会想到现实中这样的例子。对于自身来说，大部分时候我们会认为自己是第一类人，也可能是第二类，或者是一和二之间的某个中间态，总之，绝对不会认为自己是第三类人。

这其实说明了人们的潜意识里都认可第一类是最积极向上的处世之道，然而在现实中，因为能力、对方态度、利益关系等各种因素，我们能表现出来的”不计较得失”就可能有很大的局限性。

如果我们每个人都向“给予者”的方向努力，那么便更容易建立一个正能量满满的工作环境，与别人的交流也会更加顺畅。

第三，加入一些有利于自己成长的社交圈子。比如对特定技术有共同兴趣的社区，对摄影、读书有共同爱好的小组等。这些圈子可能很大很正式，比如一些商业组织，也可以是几个有共同困难或者目标的同事组成的微信群，平时进行一些非常随意的常规讨论。

第四，适当地寻求帮助。很多时候主动向另一个人表示你需要他的帮助，其实会很巧妙地增加两个人的亲密度。当然，不要问不值得问的问题，注意是不是会耽误对方太多的时间，自己的态度也尽可能地开放、谦逊。

反之，如果别人向你寻求帮助，尽可能耐心地协助对方解决问题，如果很忙或者时机不对，就告诉对方，并问是不是可以换个时间。如果对方比你资历浅，或者问的问题太简单，也一定不要摆架子或者显得很不耐烦。

第五，对于别人的意见要尽可能认真对待。即使没有任何实际行动，也让对方明白你的想法，为什么你没有采纳他的意见。自己给别人的意见也要诚恳，如果是个人看法，不要强加于人，尤其不要拿一些理论上有争议的看法强迫别人接受，比如哪个设计模式更好。如果是真实案例里并且会影响项目进展的对错，尽可能摆事实举例子让对方明白你的想法

和出发点，本着把事情做好的原则去交流沟通。

总结一下，今天，我与你分享了人际关系在我们的工作和生活中的重要性。良好的人际关系可以让我们在工作中如鱼得水，反之则步履维艰。如何在工作中构建好的人际关系呢？我的经验是保持开放的形态、努力成为一个给予者、加入一些社交圈子、适当寻求帮助、认真对待别人的意见。

你有什么好的做法呢，也可以在留言中告诉我，我们一起成长。感谢你的收听，我们下次再见。

# 编程语言漫谈

编程语言是一个已经被谈到耳朵发烫的话题，很多工程师都聊过。似乎无论怎么写，要么落入老生常谈的俗套，要么就是一堆理论上正确，但是对学习和理解编程语言并无多大益处的内容。

我跟池老师说：这篇可说的内容太多，反而不知道从何说起。池老师建议我：“任思维流动，想到哪写到哪。比如你可以点评一下自己最擅长的语言特性，比如新人学编程如何做到触类旁通，最后一通百通。”回想一下，这两个思路我似乎都用过了，以前写过点评 Ruby 和 Java 的文章，也写过自己的编程之路，今天说点平时不说的内容，可能不讨喜，但至少可以提供一些思考。

我在莱斯大学读硕士期间，课题就是程序语言和编译器设计。翻看当时用到的教科书和论文，需要我学习和掌握的内容大概都是这样的：

1. 一些关于图灵机和状态机的计算理论
2. 类型和类型系统，类型系统的证明和推断
3. 函数中的递归、迭代及其实现原理
4. 关于 Lambda 的演算和模型
5. 命令式程序语言、函数式程序语言、逻辑式程序语言、以及面向对象程序语言的本质区别
6. 语法器、词法器、编译器、解释器的原理及实现

所以，很多人刚开始接触编程语言，第一反应是怎么用、好不好用。而我看到一门语言，第一反应是这门语言是怎么实现的，类型系统是什么，计算能力的边界在哪等等。最早接触的几种语言，比如 C、OCaml、Schema、Python、Java 等，都被我拿着“手术刀”解剖过。那个时候，我想到最多的成语是庖丁解牛和洞若观火。

我的硕士毕业设计是用 OCaml 去实现一个机器人仿真和控制的语言，有点像 Matlab 里的机械模拟库。研究的目的，说白了就是不断用 0/1 这样的离散数学和语言，去逼近物理世界中很常见的连续函数（如导数和积分），而挑战的就是有限的计算资源和时间。

后来到了博士阶段，课题变成了生物信息学，那个时候常常要做的是生物学中的大数据处理和建模。由于业界很多数据科学相关的库都是 Python 实现的，于是我使用到了大量的 Python 编程。

不过在处理海量数据的时候，Python 的性能就成为一个很大的瓶颈。经常遇到的情况是，一个脚本或者一个函数库，其复杂度在一个临界点后是指数增长的，稍微大一点的数据量，动不动就要跑几天，或者干脆跑

不出来。

于是我又开始学习并使用一种叫做 Cython 的语言。这种语言在语法方面算是 Python 和 C 的混合体，其编译器可以将 Cython 代码转化为 C 并编译成性能很好的可执行代码。那时候除了建模需要的数据清洗和数据模型的训练，另一个挑战就是写出高性能的代码。

博士毕业后，我没有选择留在学术界，而是进入了互联网公司工作。

Square 和 Airbnb 都是以 Ruby 和 Java 为主要编程语言的公司。这个时候，语言上面临的挑战已经不再是其计算能力或者性能，而是如何在工程中用适当的语言搭建出一个方便协作、性能过得去、可读性好、模块化好、可重用、易扩展的代码库。

很多时候工程师们争论的问题，不再是对和错、是与否的问题，而是每个人的观点应用到相关的场景中的时候带来的优劣比较。也就是说，是不是把合适的技术用到了正确的场景中。平衡是我们在这个阶段要着重考虑的，这种平衡有时候是时间复杂度，有时候是空间复杂度。

最近几年我主要使用 Ruby 和 Java 编程，这两门语言的优缺点就不在这里说了，网上有很多类似的观点。关于 Java，后面我会写一篇“Java开发的中常见问题”，下面我会来简单聊聊自己对软件工程和编程语言的一些看法，有的是点评别人的观点，有的是我自己的观点，分享给你，希望可以给你带来一些启发。

**1** 初学者不要纠结**“**先学哪种语言**”**，这种时间花的很不值得，还不如随便挑一个语言，跳进去游几圈试试。对于工程师来说，学习第一门编程语言只是万里长征的第一步，只要你还在这个领域，就不可能只学一种语言，只会一种语言的工程师根本就不能称之为工程师。

**2** 如果你不能用一种编程语言的基本特性写出好代码，那换成另外一种语言也无济于事，你会写出同样差的代码。比如，你的 Java 代码写得很糟糕，那么换成 Go、Ruby，你的代码也会一样糟糕，甚至更差。

所以，基本掌握了一门语言的功能和语法特性之后，要去做实践和练习，能写生产代码了，再回过头来去看编程语言的本质，了解这门编程语言的设计原理，能力边界和高级功能，这样有助于你更快更好地掌握其他编程语言。

**3** 很多人觉得不要用脚本语言入门，我觉得不一定，尤其现在就着人工智能浪潮搞机器学习的人，用 **Python** 入门就很好。另外，脚本语言在面试中绝对占优势。平时工作中我对 Ruby、Python、C++、和 Java 的熟练程度差不多，但是面试中使用 Ruby 或者 Python 答题，写代码的时间估计是那两者的一半。

如果让我推荐一门脚本语言，那就是 Python，关于 Python的历史和语言特性，可以参考池老师之前写过的“人生苦短，我用 Python”一文。

https://mp.weixin.qq.com/s/sSl2PHiuQWmNuQMgoL4qcw

**4** 后端工程师要熟练掌握一门前端语言，前端工程师也要熟练掌握一门后端语言。倒不是为了提倡全栈或多个能力储备，而是两者的编程思维模式很不一样。知己知彼，在架构设计和解决具体问题时，才会有更精确的判断。

另外，现在大前端的概念也比较流行，也就是大前端工程师能够同时掌握 Web 编程语言、iOS 和 Android 编程语言，原生技术（iOS 和 Android）和 Web 的配合会越来越紧密。

**5 SQL** 是一门非常非常重要并且应该熟练掌握的语言（虽然它不能被称为程序语言）。我在这里用了两个非常，因为很多工程师有些过于轻视 SQL 了，并为此付出了惨重的代价。

如果你平时的编程工作涉及到业务功能，而不是纯粹的技术架构，一定会使用到数据库。SQL 就是数据的语言，通过它，你可以和数据建立连接和沟通。

如果你的数据访问模式写得很差，轻则代码性能一塌糊涂，重则引发 Bug，而涉及数据的问题，Bug 等级都比较高，后果可能很严重。

关于SQL，可以参考我之前的专栏文章“每个工程师都应该了解的：数据库知识”。

**6** 无论使用什么语言，工程师都应该能够基于这种语言搭建测试框架，写好测试代码和写业务代码一样重要，甚至更重要。工作后你会发现，可能有时候我们只花五分钟写了一个程序，而为其写一个差不多能够覆盖所有功能路径的测试用例集却花了一个小时。

关于测试，可以参考我之前的专栏文章“每个工程师都应该了解的：A/B 测试”。

**7** 最后的，也是最重要的是：在任何时候都要用并发的、分布式的思维去看待你的程序。因为竞争条件或者并发中的不确定因素（比如调用顺序）导致的 Bug，仅仅理解语言的基本特性，根本不能解释。

每种语言都有自己的并发编程模式（比如 Go 的 Goroutine，Java 的 ForkJoinPool，Swift 的 Swift Grand Central Dispatch 等）。学习每一种语言，都应该去深入了解它的并发模型，在这个多核的时代，不懂并发的程序员不可能是个好工程师。

今天我以漫谈的形式和你聊了聊上学和工作过程中学习和使用编程语言的经历，然后我讨论了一些对软件工程和编程语言的看法，算是经验之谈，希望对想要学习或正在使用编程语言的你有所帮助。

编程语言，你看它是山，它就是山；看它是水，它就是水。你可以把它当做一门简单的编程语言，有语法，有特性，也有优缺点，但这样的语言也可以复杂到去实现和解释各种计算模型和理论。一门编程语言到底

能做什么，完全和工程师怎么去用，在什么场景中用息息相关。

最后留个讨论题，什么是真正的全栈工程师，他们需要掌握更多的编程语言吗？期待你的回复，我们下期再见。


戳此获取你的专属海报

# 兼容并包的领导方式

你可能听说过，很多硅谷公司都有个不成文的规定，叫作“多样化（Diversity）”，这是什么意思呢？

简单来说，就是要求在公司招人的时候，尽可能保持员工组成的多样化。

我先来谈谈，为什么要保持员工组成的多样化。

第一个原因是，互联网时代的市场组成是多样化的。很多欧美的产品在中国市场的销售额是公司营收的主要组成部分，国内的很多产品也都开始具备国际化意识，逐步介入欧美中东等地区的市场。

并且，硅谷地区的人口结构和美国整体是不同的，多次殖民和移民的历史，让硅谷的人口本身就呈现多样化，也因此硅谷公司的国际化视角从创建之初就有了。

在思维方式上，我们希望在构建产品的时候就开始兼顾市场的多样化，比如考虑设计、本地化、语言、使用习惯等等，而多样化的员工可以更好地帮助实现产品多样化的功能特性。

第二个原因是，产品的用户也是多种多样的。每个客户会希望产品具有个性化特性，也就是有“这个特性感觉上是为我做的，更适合我使用”的需求。因此，要保持产品的竞争力，就需要了解更多客户的不同需求。

用 Airbnb 的用户群体举例，有的人希望找到比宾馆便宜的落脚点，有的人希望体验一下别人的豪宅，有的人则更想拥有和当地人一样的体验，还有一些人可能仅仅是喜欢尝试新鲜事物。

产品或者平台怎样才能让每个用户都体验到想要的个性化呢？这就要求设计者必须意识到用户的多样化和需求的差异化，也就从根本上要求了组成设计者的员工是多样化的。

那么，如何才能保持员工的多样化呢？

首先，多样化可以体现在员工的国籍、人种和性别这样的外部区别上，比如：白种人、黄种人、黑人都要保持一定的比例；性别上也要求保持平衡，也就是比较健康的男女员工数量。还有一些公司会要求有一定比例的同性恋、不同类型的宗教徒等等。

其次，员工的多样化也可以体现在团队内部的差异上，互联网时代需要多种多样的人才，人才库的组成也应该是多元化的。一个团队里，需要具备开创性思维、敢冲敢拚的人，也需要一些能够深思熟虑，以及能够把事情落地和执行的人。

最后，员工的多样化也体现在团队的创新思维上。一个公司，如果只有一个 Idea，一个产品，那么这个公司早晚会被时代的大潮抛弃。比尔·盖茨曾经说过：“组织要么创新，要么死亡”，就是这个道理。

因此，团队内部需要有创新者，或者是“异见人士”的存在，他们的思路和见解并不总是和产品的主要设计者保持一致；这种情况下，团队很难“想到一块去”，也就保持了团队创新的活力。

有一些不同的声音，对于“哪些重要、哪些该做”有些不同的看法，哪怕这些想法在初期会被认为风险太高、太离谱，但是在某种程度上，它们会让团队的活力更持久。

多样化的员工组成保证了多样化的企业文化，所以在很多时候，这也要求领导者具备兼容并包的思维。

“包容性领导的六个特征 ”（The six signature traits of inclusive leadership） 一文中提出了一种新的领导方式，文中认为领导者应该具备以下六种技能或者特征。

**1** 坚定的承诺。领导者需要从内心深处认为多样化和融合是一个正确的人生观和价值观。保持开放接纳的心态，平等对待每一种人和文化，是对企业有益的。

**2** 谦卑的勇气。领导者不要有盲目的自我优越感，而应该客观地看待自己的优点和缺点。对待他人都要保持一种谦虚的态度，无论他们是何种教育背景、人种、性别。

**3** 正确的认知。领导者需要知道自己和企业有盲点、有偏见，知道问题在哪，知道解决问题的方向。

**4** 开放的心态。领导者需要真诚地渴望了解不同人的看法，可以接受别人的一些价值观，并且知道所有的事物都有一定程度的不确定性。

**5** 高情商。领导者能够和不同职位不同背景的人愉快合作和交流。哪怕不是百分百的认可，也可以很专业地处理。

**6** 合作的意愿。给每个人施展的机会和空间，提供公平和安全感，哪怕这意味着额外的代价和麻烦。

硅谷很多公司，把这六种特质作为公司文化最重要的组成部分。他们在战略上投入，明确表达自己的立场。比如，他们会把这样一些开放包容相关的标语印到文化衫、咖啡杯上等等。在招聘过程中，一方面表示自己支持多样化，另一方面也会考察应聘者是否具备同样的观念。

另外，在整个公司运行的过程里，是否具备兼容并包的领导力是对领导者的重要考察点之一。公司还会定期做一些培训，让大家都理解为什么多样化这么重要，领导者应该怎么做。

以前这个话题对国内很多公司可能没那么重要，因为大部分公司都在开

垦国内市场，但是现在国际化的业务需求越来越迫切了，人才的多样化和领导力的包容性应该是国内公司急需解决的问题。如果你希望公司和团队变得更优秀，更有创新力，更有活力，引入多样化的人才，也许是你需要考虑的。

总结一下，今天，我从硅谷公司的“多样化”规定讲起，首先谈及了为什么要保持员工的多样化，市场多种组成和用户的多样需求促成了员工的多样化，然后分享了如何保持人才库多样化，最后从人才的多样化谈到了领导者应当具备兼容并包的思维，只有人才的多样化和领导的包容性并存，团队才会越来越好，越来越有活力。

你的公司崇尚“多样化”吗？你可以给我留言，我们一起成长，感谢你的收听，我们下期再见。

参考资料：“The six signature traits of inclusive leadership”
https://www2.deloitte.com/insights/us/en/topics/talent/six-signature-traits-of-inclusive-leadership.html


戳此获取你的专属海报

# 如何做自己的职场规划？

写了专栏之后，经常会收到下面这样的留言。

> 安姐你好！我在国内的公司工作了将近一年，感觉公司里很多东西以业务为主，而团队的成员结构新人居多，最后做出来的东西感觉业务逻辑满屏皆是，业务代码很重，重复性的东西也多，加班还厉害。国外会有什么不一样的地方吗，会不会加班少、做的技术又很棒？
>
> 安姐，作为一个入职近一年的职场新人，我发现最近几个月一直在做重复性的工作，自己的技术水平和其他能力并没有太大进步。请问怎样才能争取到机会扩大自己的工作范围，承担更多责任，做一些更新鲜更有挑战性的工作呢？

不知道国内大环境对职场新人的友好程度如何，转成管理后，我发现自己的效绩评估里有一条很重要，就是帮助组内的人做职场规划，以及帮他们实现进阶。我最近也一直在反思自己职场历程，所以今天就和你聊聊这个话题：如何做自己的职场规划？

做一个职场规划时，你作为当事人，自己要先想清楚很多问题，然后再和你的领导者交流沟通，寻求他的支持和帮助。

那么，下面的问题是你首先应该考虑的。

1 你的个人价值观是什么，最在意什么？换句话说，什么样情境或状态会让你有幸福感或者自信心。比如，在你的成长路上哪些是你在意的，是独立解决问题的能力，还是挑战别人做不到的东西；是受欢迎程度，和所有人良好的关系，还是更在意自由、健康的家庭生活。

2 你的长期愿景是什么，五年甚至十年后，你希望自己成为什么样的人？

3 为了达到目标，你还需要哪些技能或者经验？你可以在短期内发展什么技能让你走得更远？想达成你的梦想职业，或在工作中取得成功，你还需要做些什么，需要哪些必备的技术技能？哪些技能对你来说不是必须的，但是会有很大好处？

4 你的优势和长处是什么？是合作性、独立思考、行动快速，还是有良好的产品思维。你现在的日常工作能否让你展示你的长处，又是如何展示的，你觉得你比别人在哪些方面做得好，能不能举出些具体的例子？

当你想清楚以上问题的时候，大概就知道自己想要什么了。如果你是一个管理者，也可以试着用这些问题去启发你的团队成员。

思考完自己的问题，接下来的，也是最重要的问题就是：你需要你的领导者提供什么样支持？

例如：

1. 需要一个能证明自己的项目；
2. 需要一个能带自己的老员工，这位员工可能是你的榜样（Role Model），也可能是为你提供日常指导的人；
3. 需要更多练手的机会；
4. 需要专注培养自己的某一项技能；
5. 需要让自己接触更多的业务或者架构相关的讨论；
6. 需要参加一个系统的培训。

在这些问题或要求提出之前，你应该考虑到领导者的资源和整体项目进度，也就是说，领导者在提供这些支持之前，很可能会有一些限制或者要求。

比如，你想要一个能证明自己的项目，那么在你承担重任之前，可能需要完成另外一些不那么重要的项目，甚至是脏活累活来证明自己。这种情况，沟通的关键就是契约，你需要如何证明自己，才能去做自己喜欢的项目，组里是不是有人明显比你胜任很多？

比如，你需要一个能够带着自己往前走的老员工，那你就要知道，对新人的指导会花费老员工很多时间和精力，当你这样要求的时候，是不是可以提供一些力所能及的回馈呢？比如主动帮助老员工做一些他手上的杂活，也可以把老员工传授给你的经验写成文档，这样，你被指导了一次，却留下了以后新人都能使用的文档资源。老员工会更愿意去带这样的新人。

比如，你想接触更多业务和架构相关的讨论，那你是否按时完成了自己的本职工作？如果你自己很勤奋，虽然参与了各种讨论，但自己的工作完成得又快又好，甚至利用自己的休息或娱乐时间做了一部分工作，这样领导者才不会有顾虑。如果你作为新人，该做的工作没做，整天参加各种讨论开眼界，这让领导者怎么管理团队里的其他人呢？

在这个过程中，作为员工，不要提过于脱离实际的要求。作为管理者，要明确地让对方了解不让他承担这个项目的理由是什么，风险在哪里，让对方知道自己的不足，并提供机会让他先提升这方面的技能。

你和你的领导者都应该明确，所有的支持和帮助对你的长期目标和短期发展都是有益的。不要提一些似是而非的需求，也不要人云亦云，如果这些要求不会让你离你的目标更近，那就是无用功。

得到领导者的帮助和支持后，剩下的就是让这一切变得可执行和可追踪。你可以和你的领导者一起为你的“成功”或“进步”定义一些可测量的标准，制定可执行的行动计划，然后记录你的发展，按时和你的领导者

进行一对一沟通，讨论你的进步，反省做得不够和不好的地方。需要的话，你可以适时调整你的计划，完成自我发展。

如何做自己的职场规划？我们来总结一下，你需要完成以下四个步骤：

1. 知道自己想要什么，知道现在的你和理想中的你差距在哪里；
2. 和领导者沟通，得到一些有前提或者回馈的支持和帮助；
3. 设定目标，制定一个你和你的领导者都同意的计划和期限，确保计划会让你和目标更接近；
4. 让计划变得可执行和可追踪，按部就班的完成和跟进，同时根据执行情况调整不合理或不完善的地方，持续改进。

你是如何做自己的职场规划呢？可以在留言里告诉我。感谢你的收听，我们下期再见。


戳此获取你的专属海报

# 小议Java语言

今天我和你聊聊 Java 语言，这也是我使用最久最熟悉的编程语言之一。

读博士的时候我做过两个领域。刚刚进入莱斯大学的时候进的是程序语言设计组。组里当时有两个教授，分别做两个领域。

一个是瓦利德·塔哈（Walid Taha），主要研究领域是类型系统和嵌入式系统的程序语言设计。另一个是罗伯特·卡特赖特（Robert Cartwright），他喜欢大家叫他 Corky，主要研究的是 Java 的理论和实现。

Corky 教授的研究项目中，有一个叫做 DrJava 的项目。DrJava 是一个轻量级的 Java 开发环境。主要是给在校学生用的，它提供了一个简单直观的交互界面，可以实现类似 Shell 的交互式 Java 代码执行方式。DrJava 在美国大学里有很多人使用，目前已经超过两百万的下载量。

Corky 是个典型的美国教授，与生俱来的花白卷发，清癯的脸庞上一双眼睛炯炯有神，高挺的鼻梁。他常年都是和学生一样的打扮：T恤牛仔裤加运动鞋。出入的时候，背上总是背着一个蓝灰色的电脑包。

当他不谈学术的时候，你会觉得他有着与年龄不相称的单纯和童真。他会和一群学生一起吃系里免费的披萨和点心，也会和其他教授一起站在系里走廊上聊天和哈哈大笑；然而一旦你和他讨论起 Java，他就变得滔滔不绝，整个人散发出特别的魅力。他对 Java 的理解十分深入，我每次和他对话都颇有收益。

虽然我的导师是瓦利德（Walid），但同在一个语言组，平时的研讨班都在一起，我也就有了很多的机会和 Corky 一起讨论各种程序语言的特性和实现。也就是在那个时候，我对 Java 语言有了比较多深层次的了解。

# 我和 **Java** 语言的开发者

我的硕士论文是独立实现的一个程序设计语言，包括它的解释编译和用户界面。这个语言主要用于机器人系统的嵌入式编程和仿真，曾经在一家石油公司的井下控制系统开发中被使用。不过因为我导师的离开和种种其他原因，我博士生涯的后三年转了另一个导师做生物信息学的数据分析和建模。

因为有程序设计语言的研究经验，博士毕业找工作的时候，也投了份简历在 Oracle 的 Java 语言开发组。也因为有这样相应的背景，我很顺利地拿到了Java核心类库 （Java Core Library） 开发小组的 Onsite 面试机会。

我去面试的时候应该是 2012 年底，当时面试的那个小组一共好像只有七八个人的样子。Oracle 的面试大部分是白板和聊天，和现在比较主流的面试，上机做题并无 Bug 运行的体验很不相同。我介绍了自己的硕士毕业设计，然后就谈起 Java 新的库或版本可能会增加哪些支持。

2012 年底的时候，Scala 和 Clojure 刚刚火起来不太久，Java 还没有对 Lambda 的支持。而当时整个Java团队也正在考虑这件事。话题牵扯到了 Lambda 的实现，正好是我非常熟悉的话题，因为我的导师瓦利德（ Walid ）主要的研究领域就是函数式语言，而对 Lambda 的实现，也是函数式编程语言的核心。

具体的讨论细节我已经不记得了，不过有两点感触颇深：一是他们对于选择哪些函数进核心库（ Core Libarary ）非常谨慎。Java 早期是很轻量级的，后来的版本功能越来越强大，但是语言本身也越来越沉重，这也是很多人喜欢 Scala 的原因。

二是实现函数库的语言开发者对每个函数的精度和运行时间的要求到了令人发指的程度，听说他们有时候读无数的论文，看无数的实现，作大量的比较，就只是为了敲定到底应该在最终的函数中使用哪一种实现方式。

比如浮点数是有舍入误差（ Rounding Error ）的，那么一个数值计算中先算哪一步、后算哪一步带来结果都可能是不同的；而实现中的考虑，往往为了小数点后面十几位以后的一个 1，组里也要反复斟酌很久。

# 为什么到了 **Java 8** 才有 **Lambda**？

很多人抱怨 Java 滞重，语法升级缓慢，过度封装，尤其是对函数式编程支持的不友好等等，其实都和这种谨慎有关。为什么 Lambda 的概念到 Java 8 才有了实现，之前的 Java 版本，包括很多其他语言都没有真正的 Lambda 实现呢？这其实和程序设计语言里的基本概念有关系。

假如我有一个 Lambda 表达式，用伪代码来写，可以写成：

```
def f(x)
    def g()
        return x
    end
    return g
end
```

这个 Lambda 表达式可以看到 f(10) = 10，f(20) = 20。

在一个没有 Lambda 支持，或者嵌套式函数定义支持的语言中 —— 比如 C 语言，这个可能会实现成：

```
typedef int (*fp_t)() ;

int g () {
    return x ;
}

fp_t f(int x) {
    return g ;
}
```

但是问题就在于，g 函数中的 x 是没有定义的，程序不可能编译运行。解决这个问题，我们可以引入一个全局的 x 变量，在对函数 f 进行定义的时候，给这个全局 x 赋值。但是由于 C语言不能每次运行时定义一个新的函数，因此，如果赋值：

```
a = f(10)
b = f(20)
```

那么，虽然我们希望得到 a=10，b=20，但是上面的实现只能给我们 a=20，b=20。

由此可以看出，仅仅的一个匿名函数（ Anonymous Function），或者函数指针，是不足以正确地实现 Lambda 的；而正确实现 Lambda，或者说允许把 Lambda 表示的函数作为一个像其他类型的值一样作为参数来传递，语言必须要有对 Lambda 的函数表达，以及一个用来在各层中传递参数值的“参数定义环境”两者同时的实现。这也就是函数语言中的闭包

的概念。

换句话说，实现 Lambda 可以作为一个普通类型一样的值来存储和传递，我们需要一个闭包（Closure），而闭包（Closure）可以看成：

闭包= Lambda 表达式 + 记录所有函数局部变量值在每一层 Lambda 中的赋值的一个环境。

实现闭包大体有两种方式。一种叫做“自底向上”的闭包转变，也称为 Flat Closure。它从函数定义的最里层，将每一层的局部函数变量值拷贝到次里层。每一层的变量可能重名，而这就需要变量名解析的技术，对变量按层重命名。这样逐层拷贝，最后形成一个 Lambda 对应的单层的变量赋值环境。

另一种叫做“自顶向下”的闭包转变，也称为共享闭包（Shared Closure）。它从函数定义的最外层，将每一层的局部函数变量赋值用类似指针的方式传播共享到里层的 Lambda。这种实现的好处是避免重命名和拷贝，但是实现赋值环境的共享其实是很棘手的。

总而言之，Lambda 在语言中的实现是复杂并且昂贵的。不仅容易出错，还会给语言的垃圾收集（GC）带来新的挑战。它也让语言的类型系统的所有证明和推导变得复杂无比。虽然现在主流的语言都提供了 Lambda 的实现，但用起来还是有一定限制也需要一些谨慎的。

比如， C 语言仍然不支持嵌套式的函数定义。C++11 增加了对闭包的支持，但是因为语言本身没有垃圾收集的原因，使用起来需要异常谨慎，很容易引起悬挂引用（Dangling References）。

比如，Ruby函数不能直接作为参数传递，而是通过 Method 或者 Proc 来使用。且函数的嵌套定义并没有很好的对作用域进行嵌套；而 Java 8，虽然有了对Lambda 的支持，但是 Java 类型系统（Type System）并没有对函数类型有任何的支持。换句话说，Java 8 中其实并没有对函数类型的类型系统的实现，这就意味着一些 Lambda 相关的类型错误，在编译时间可能无法被发现。

看完了这些你就会知道，一门编程语言的变革是多么艰难和复杂。好在 Java 9 已经发布了，Java 语言有了更高和更新的起点。

# Java 开发中的常见问题

很多新人入门会要求我推荐编程语言，Java 属于我推荐的语言之一，因为 Java 标准、规范，是面向对象编程的代表，Class、Object、Interface、Abstract、Public、Private、Override 等关键词显式清晰，一旦使用就不会混淆，在学习其他编程语言的时候还可以参考互通。

另外，由于 Java 的流行和开放性，围绕 Java 语言形成了最为广泛的开发平台，不仅有 Spring 这样开源生态社区，在 Java 平台之上还衍生出了很多轻量级的编程语言，比如 Scala、Groovy、Clojure 、Kotlin，这些语言都可以运行在 JVM 之上，形成了非常有生命力的生态环境。

那么在用 Java 开发的过程中需要注意哪些问题呢？

这里的问题并不是指哪些常见的“Java编程最常犯的错误”，比如：

1. 如何正确地将数组转化为列表；
2. 如何判断一个数组是否包含一个值；
3. 如何在一个循环中删除一个列表中的元素；
4. 什么时候用哈希表（HashTable），什么时候用哈希映射（HashMap）；
5. 不要在集合中使用原始类型；
6. 如何正确设定访问级别；
7. ArrayList 与 LinkedList 的对比；
8. 可变与不可变；
9. JUnit 中@Before 和@After行为的顺序；
10. SPI 中参数如何设定；
11. API 不要返回空值。

这里想说的是，从架构和规范的角度，如果你选择了使用 Java 进行开发后，应该注意的那些问题。

## 1 不要重复造轮子，也不要搬太多不同型号的轮子来用

Java 语言有非常成熟的技术社区，在 Java 的生态里有很多相关的库、插件和工具，大部分是成熟技术。比如DW（Dropwizard），基于 Java 生态中稳定的库构建了一个轻量级的框架，可以支持系统中的配置、日志、序列化等操作，还能构建基于 RESTful 的微服务架构。

如果一个公司内有不同背景的 Java 工程师，有时他们会根据之前公司的

工作经验，选取自己熟悉的库和工具，这时就有可能会引起系统中（比如序列化）使用了两种“轮子”。

如何确保大家使用统一的库和工具呢？有很多途径，比如指定编程规划，或是使用工具完成某个服务的初始化，自动配置公司常用的库，或者通过统一的代码审核，让一些资深程序员把关等等。

## 2 深入了解依赖注入，选择一个好框架

稍具规模的 Java 代码库，都会有复杂的类和对象之间的相关性和依赖性。确保所有工程师理解并使用一个统一的依赖注入框架（比如 Guice）会让很多代码的风格保持一致，避免类的初始化过于复杂。

(注：Guice 是谷歌推出的一个轻量级依赖注入框架，帮助开发者解决 Java 项目中的依赖注入问题。如果你使用过 Spring 的话，会了解到依赖注入是个非常方便的功能。不过，假如你只想在项目中使用依赖注入，那么引入 Spring 就显得太重了，这时候可以考虑采用 Guice。)

## 3 所有项目使用统一的文件夹结构

在构建项目的时候，工程师们争论最多的问题之一，就是代码文件怎么组织。库放哪里，工具类Utils 文件夹放哪里，哪些做成单独的 Service 文件夹，哪些是 Client文件夹，分别又有哪些类。这个需要早期的工程师或者资深工程师统一协调，保证代码库的整体结构自顶而下都结构清晰，不同项目里的文件结构也尽可能遵循同样的组织方式。

（注：utils、service 和 client 都是常用的代码文件夹名称。）

## 4 规范所有项目的 API，保持统一的风格和模式

API 的接口定义是不是使用 了 IDL（Interface Description Language），服务器（Service） 之间的交互协议是不是一致的，比如基于 HTTPS，内部功能模块和数据 Schema 之间的分层和转化是否有统一的标准。如果在开发过程中，不同服务器或者模块采取了不同的接口模式，让集成变得异常复杂。

## 5 规范开发环境，提供合适的工具

可能每个人都经历过这样的场景，用自己的编辑器打开代码库的代码之后，做一下自动格式化，或者新增一些代码，结果缩进、空行、括号等

等，突然都不一样了。在协同开发的时候，经常会出现多人修改同一批代码的状况，如果频繁遇到代码不规范会非常麻烦，并且会出现很多无效格式的提交（Commit）。

要保证所有人的代码格式设置一致，才能避免这样的问题。定义一些标准的格式文件，做成编辑器插件，要求每个开发者使用统一的格式定义，就能避免上述的问题。

总结一下，今天我和你谈了我上学时是如何接触和使用 Java 语言的，讲述了 Java 语言开发者的故事，探讨了为什么 Java 很晚才支持 Lambda 这样的语言特性，最后介绍了在实际开发过程中应该注意的一些问题，希望对你有帮助。

如果你还不知道应该学习哪一门编程语言，那么就从 Java 开始吧。感谢你的收听，我们下期再见。

# 如何激发团队人员的责任心

英文中有两个词很有意思：**Accountability** 和 **Responsibility**，大体都可以翻译成“责任”，但是细细品味，却在用法上有所不同。Accountability 有点儿问责、责任制的意思，是一个由外自内的原则和约定，让某一个人对某件事负责任的意思；Responsibility 虽然有类似的意思，但更多时候是一种道德和义务上的责任心、责任感的意思。

在公司和团队里，这两种责任都不少见。不过 Accountability 不是自发的，而是通过交涉、讨论、约定，最终决定由谁来负责，并要求责任人同意负责这件事。比如保质保量按时完成某个项目中的某一个模块，是产品经理的责任，哪些需求使用哪些技术实现，是某个工程师的责任，等等。

Responsbility 是责任心和责任感，是一种更强烈的自发的责任，员工从内心深处觉得觉得自己对某件事、某种集体利益和荣誉具有不可推卸的责任。公司里有一些没有具体岗位划分和衡量标准的工作，比如处理不是自己责任范畴的 Bug、帮助别人答疑解惑等，具备 Responsbility 的人会主动去承担这些工作。

从某种程度上来说，责任心更多是一种抽象的概念，很难去培养，即使可以培养，也是个漫长的过程。除了员工自身的素质，公司或团队的规模和文化、公司的发展前景、员工的信息、平均水准和表现等等，都是影响责任心培养的元素；但是，责任心在团队里的作用却很大，有责任心的团队，战斗力和生命力都是异常强大的。

作为团队管理者，如何去激发团队人员的责任心呢？可以从以下三个方面考虑：

1. 明确责任制，尽可能通过规则来明确和规范与责任心、责任感相关的事情；
2. 让责任制变得有效，而不是形同虚设。每个人真正对自己的那块业务负起责任来；
3. 尽可能地让团队成员充满归属感，进而激发他们的责任心。

首先说说明确责任制，怎么把一些没有明确职责范围的事变成职责呢？有一些不同的方法可以尝试。比如，适当的放权，让团队人员不是负责执行一些事情，而是对某一块业务具备完全的决定权，也就是说，让他们去主导一些事情。这样员工会认为自己对项目有完整的所有权，进而具备责任心。

在硅谷的很多公司，工程师们不仅仅写代码，他们会参与产品和设计的

讨论，负责自己模块的架构设计，编写代码实现，然后测试上线，其中任何一个环节都是连续的，并不是工程师们写完代码扔给测试就不管了。最后，他们还要负责产品的改进和 Bug 修复。

当然，由于人员流动、项目组重组的原因，这种全权负责的项目制度还是会留下灰色地带。比如，前面一个人项目刚做完甚至做了一半就离职了，或者被抽调到其他组了，那后期的 Bug 和改进要怎么做呢？

如果项目剩下的工作还很多，可以找一个工程师接手并负起全部责任。如果剩下的工作不多了，而且很零散，可以通过一种守门员的机制，组里的人每周轮流值班，负责这一周与该项目相关的各种杂活，比如处理用户反馈，修复 Bug 等。

另外，还要有适当的奖励机制，对于主动承担责任的员工表示认可和感激，适当的物质激励也是可以的。他们自发地做了一些对团队有益处的事情，承担了本不属于自己的责任，理应得到其他人的尊重。

其次，让责任制变得有效而不是形同虚设。关键点是让责任人意识到承诺了就要努力做到，如果承诺的事没做，需要承担后果，而不是没人在意。

比如 Bug 引发了事故，大家只是去修复 Bug，但没人去跟进 Bug 是怎么产生的，造成了什么影响，后续怎么预防，那 Bug 就会越来越多。如果项目延期，大家只是继续延后项目完成的时间线，而不是去分析为什么会延期，如何赶上进度，是否需要外部资源的支持，那期限就会变得可有可无。

有效的责任制，在开始的时候就要让所有人明确责任与权利，而不是最后追究责任或推卸责任。

在这个基础上，根据每个人的不同情况，在执行过程中适度跟进。发现问题的时候，及时指出来，但这时需要注意的是，管理者要用关心的口吻，而不是追究的态度，让对方了解到问题出在哪里。不要因为做好人而什么都不说，那样只会让小问题扩散成大问题。多花时间，让对方自己认识到问题所在，而不是把你的主观感觉强加于人，用引导的方式，会更好地激发团队的责任心。

不要变成一个微观管理者，也不要成为一个纯粹的规则执行者。那样对团队人员的责任感、上进心和积极性等，都有害无益。

最后谈谈归属感。归属感是指一个人对某样事物、组织的从属感觉，是一种主观的个人感受。

比如一个对公司有归属感的人，会对公司产生一种“家”的感觉，觉得自己是公司的一分子，会非常在意公司发生的一切，并希望公司发展得更好，自己也会有更多的空间；相反，如果员工对公司没有归属感，始终会认为自己是这个公司的过客，总有一天会离开的，自然也谈不上什么

责任心，能够做一天和尚撞一天钟就不错了

如何增加员工的归属感呢？首先要在有利公司发展的基础上建立独特的企业文化，创新、公开透明、积极向上，这些因素可以留住更优秀的员工。

需要强调的是，公司不是温情脉脉的家，公司是一艘大船，有了方向，大家合力划桨，才能到达理想的彼岸。想要优秀的人产生归属感，仅仅靠丰厚的薪酬待遇和舒适的工作环境是不够的，他们还需要远大的目标和坚定的信念，只有真正伟大的创见，才能让这些优秀的人与公司一起往前走。

除此之外，管理者还应该以身作则，让员工看到自己的努力，对公司目标的追求，对企业文化的践行。真诚对人，能够从员工角度考虑问题，对好的行为认可并加以鼓励，同时做一些仪式感比较强的团队活动和建设，都是增加员工归属感的好方式。

总结一下，今天我和你讨论了两个有意思的英文：Accountability 和 Responsibility，并说明了这两个词的区别。那如何去激发团队人员的责任心呢？建立明确的责任制，让责任制变得有效而不是形同虚设，最后一点，让员工产生归属感，给他们足够明确和远大的目标，建立独特的企业文化，让优秀的员工与公司一同前行。

你在激发员工责任心方面有什么心得呢？可以在留言中告诉我。感谢你的收听，我们下期再见。

# 说说硅谷互联网公司的开发流程

今天，我和你聊一聊硅谷互联网公司的开发流程。之前我的很多文章里就或多或少涉及过这一方面的内容，最近我又全程参与并负责了两个大项目，对流程有了更深一步的理解，今天就在专栏里系统地和你探讨一下。

总的说来，我们的开发流程包括这么几个阶段：

1. OKR 的设立；
2. 主项目及其子项目的确立；
3. 每个子项目的生命周期；
4. 主项目的生命周期；
5. 收尾、维护、复盘。

# 第一点，**OKR** 的设立

所有项目的起始，都应该从 Roadmap 做起。硅谷公司的 OKR（Objectives and Key Results）一般都是自顶而下的。也就是说，先有整个公司的 OKR，然后有每个部门的 OKR，继而是大组的 OKR，再到小组的 OKR，确保整个公司有一致的目标。在这个过程里面，技术驱动反映在哪些方面呢：

首先，确定 Roadmap 的过程中，我们会采用调查（Survey）模式，确保工程师的声音可以准确地触达管理层。比如：工程师们觉得基础架构比较薄弱，公司就会加大这一块的支持力度。如果大家觉得开发环境很低效，就会把这个因素也放到 OKR 的考虑。硅谷的公司一般会分为产品组和系统架构组。总的说来，系统架构组的 OKR 里，工程师的声音会很大。

其次，项目怎么做，怎么规划，一般是由工程师来决定。OKR 只确立目标，是不是要构建新的服务，是不是要沿用现有的架构，如何进行技术选型等等，这些不是 OKR 的组成部分。

最后，估算 OKR 里的目标工期的时候，我们会除去一些用来做技术创新和支持的时间，比如编程马拉松，开源支持等的事务。谷歌的员工会给自己留 20% 的自由项目时间，这些都是时间缓冲区。

（注：OKR 是企业进行目标管理的一个简单有效的系统，能够将目标管理自上而下贯穿到基层。具体概念可以参考 http://wiki.mbalib.com/wiki/OKR。）

# 第二点，主项目及其子项目的确立

一旦确立了 OKR，下一步就是确立主项目和子项目了。主项目是主要的技术或商业产品，一般由产品经理、技术经理和一些技术骨干经过产品需求和技术讨论之后，确定要做什么（Scope），不做什么（Non Scope）和大的里程碑（Milestone）；后面我会在“工程师、产品经理、数据工程师是如何一起工作的”一文中更详细地介绍不同角色之间的合作细节。

一旦主项目确定了，就需要安排不同的人做不同的模块，也就是子项目。一般团队协作有两种方式：一种是每个人负责一个子项目，从始至终；另一种是大家先一起完成基本框架，然后逐个需求、逐个模块推进，最终一起完成整个项目。

下面，我来谈谈两种协作方式在实践中的优缺点对比。

第一种协作方法：每人完成一个子项目。

> 优点：责任清晰，每个人都知道自己的职责，工程师们也有更多的拥有感，他们可以独立负责产品的设计、实现、测试和维护，工作贯穿整个项目过程。
>
> 缺点：如果负责某个子项目的工程师设计或者实现能力不足，由于比较独立，这个子项目很容易成为路障或者瓶颈，工程师之间也缺乏互相学习的机会。
>
> 另外，因为是按人并行推进项目，需要根据每个人设置里程碑，管理的时候，技术管理者需要常常跟进每个人的进度，管理代价更高。代码审核往往也只是有限的几个人参与。

第二种协作方法：所有人一起逐次完成每个模块或需求。

> 优点：工程师之间合作最大化，可以彼此协调、彼此学习、在对方有事的时候相互补位。项目管理有明确的统一的里程碑，每个工程师都有机会接触更多的工作，每个人的代码可以有更多人参与审核。
>
> 缺点：每个工程师的责任不是那么明显，很容易出现能者多劳、勤者多劳的现象。一些新人总是做一些执行或打杂的事，得不到锻炼。

这两种模式我都曾亲身经历过，感觉两者各有利弊。现实中可以根据情况组合使用。比如，两到三个人合作负责一个模块，也可以在每人一个模块的基础上，将小模块组合成大模块。然后每个大模块有个技术负责

人（Tech Lead），对一些能力不足的工程师给予指导和支持等。

# 第三点，每个子项目的生命周期

子项目一旦确认，它的生命周期就融入到工程师们的日常工作中，内容如下。

1 开发初期的设计文档。一般使用可以共享的谷歌文档（Google Docs），Quip 等。不同的人可以编辑或者评论、阅读。一般设计文档会先由组内工程师和产品经理审核，然后到大组评审（包括Legal，Compliance，Finance 等等）。

如果涉及公司的整体架构，还需要发给全公司审核。参与审核的人员是所有的工程师。很多人会有选择的参与一些设计的审核，通常技术骨干会预留时间审核所有的技术设计文档。设计文档不仅包括怎么实现，还有选型的理由、考虑的因素、支持和不支持的属性、时间线等等。

2 设计测试实验，这是可选的，如果针对某个产品需求我们想知道用户的反馈，就需要数据工程师参与设计实验，也就是 A/B 测试。实验中的数据埋点也会在下一步的实现中完成。

3 一旦设计文档锁定，就可以开始实现了。不论是单人负责还是多人合作，实现都是按照多次代码提交（Pull Requests）来迭代的。每次代码提交要写清除代码改动的摘要和测试。并通知不同的工程师审核。

4 所有的实现都要加入监控、日志、预警代码。

5 所有实现都是隐藏在一个开关后。当代码都就位后，就开始灰度发布。通常是先发布给几个开发人员测试，然后到项目组，然后到其他员工（Google 称之为 Dog Food，因为他们可以大量使用自己的产品），最后按照百分比推给用户。

推送的过程中会结合 A/B 测试，只有测试结果显示对用户体验、公司主要的指标（ Metrics ）没有明显的负面影响，才会正式发布给所有用户使用。

6 对一些需要重构的关键产品链路，有时候也会使用双重写入（Dual Write），就是新特性和旧特性都写入数据库，并通过不同方式比较两个实现的结果。只有验证结果一致时，才会将交易（Traffic ）从旧实现切换到新实现。

7 最后是一些扫尾工作，包括移除用来做 A/B 测试和灰度发布的代码开关等，有时候还会有一些次要需求的实现。

# 第四点，主项目的生命周期

主项目的生命周期根据子项目的实现方式会有所不同，但有一些特点是共有的。

1. 项目开始都有一个整体设计文档，界定所有子项目的范围和相关性、时间线等。
2. 在所有子项目进行的过程中，有时候会发现一些共同需要的架构或者服务，可以单独提取成公共服务或库，比如一个调度服务，或者一个幂等实现等等。
3. 给相关人员做进度报告，包括主项目的里程碑。
4. 由于子项目完成时间可能不一样，需要进行人员的重新配置。
5. 在开发过程中不断更新文档。
6. 因为不确定的需求变动，会取消或者生成新的子项目。
7. 有时候，也会因为公司的方向变化或战略调整，对主项目做比较大的变更，同时对应调整相关的子项目。
8. 在项目开始和结束的时候，需要做好对外的交流和沟通。一来确保自己的项目改动不会影响到其他组的项目，二来让将来会依赖这个项目的产品组了解相关信息，确定计划。

# 第五点，收尾、维护、复盘

整个项目结束后，一般都会做一些代码清理和文档的更新和整理，有时还需要写新的用户手册或 Wiki 等。一些基本的错误和异常处理要写到运维手册（Oncall Playbook）里，便于以后运维的人知道怎么处理一些已知的问题。

每个项目结束都会进行复盘，总结整个项目的教训和经验。有时候还需要在组内做些演讲，让更多的人了解这个项目。

总结一下，今天我主要和你探讨了软件产品和服务的开发流程，一般硅谷里稍具规模的互联网公司都会遵循类似的流程，他们就是通过这样的流程开发出了创新性的产品。

这些流程包含什么内容呢：首先在公司内部确认 OKR，然后确定主项目和子项目，开始进行产品实现，也就是完成主项目和子项目的生命周期，最后进行项目的收尾、维护和复盘，一个大项目就开发完成了。

你的项目开发流程有什么异同吗？可以在留言中告诉我，大家一起进步。感谢你的收听，我们下期再见。

# 编程马拉松

今天，我和你聊聊“编程马拉松 Hackathon”的故事。

Hackathon是一个合成词，取自Hack和Marathon，意为“编程马拉松”，又叫“黑客节”或“编程节”，它起源于Sun公司在1999年的 JavaOne 大会的一次活动。

在那次活动中，约翰·盖奇 （John Gage）向参会人员发出了一个挑战：用Java为新款的Palm V编写一个程序。这个程序可以让Palm V用户通过红外线端口和其他Palm V的用户通信，此外，Palm V用户还能够通过该程序在网上注册他们的手机。这个活动第一次引入了编程马拉松的概念，并沿用至今。

现在，湾区比较知名的创业公司都有个不成文的传统，通常一年会有几次编程马拉松，国内的一些互联网公司和技术社区也举办过类似的活动，就是有一周的三五天内，大家不工作而是去“Hack”一些自己感兴趣的小项目 。

编程马拉松期间，全公司理论上是不允许有任何工作相关的会议，所以你每天照常上班，但上班的内容是自己找一群小伙伴，大家一起做一个和工作不直接相关的项目，最后一天则展示自己的成果。

编程马拉松与平时工作的不同之处就在于，它让你去探索一些平时工作中无暇顾及的创意和想法，在很短的时间内推进和完成一个项目。

不只是说一说，而是去通过设计、交互和程序去展示你的想法。编程马拉松的另外一个好处是，可以让你在团队工作中和一些平时少有合作的同事建立很特殊的感情和纽带。

有一次编程马拉松接近尾声，我晚上开车回家，天色渐渐暗下来，看着路上忽亮忽暗的一排排车灯，以及路边不多的高楼上的广告牌灯光，听着一些略显过时的美国流行歌曲，突然间，以前 “Hacking”的一幕幕涌上了心头。很多老朋友、新朋友的笑容就清晰地闪现在我的眼前。

# 第一个故事：**Hack**的项目成了正式产品的原型

我第一次参加编程马拉松还是在Square，那时我加入Square Market组可能刚刚两三个月。Square Market组里有三个整天腻在一起的男生，关系特别好，一起吃饭，一起喝茶，晚上经常一起联网打游戏，周末也经常到一起玩纸牌。

他们三个有一个人是个在美国出生的华人，中文能听懂，但是说得不好，所以不太经常说。他早年在谷歌中国待过，是谷歌早期的牛人之一。另一个是香港人，来美国十几年了，普通话也不那么流利，但是前端后端都十分精通。

还有一个是个地地道道的美国人，但娶了个韩国妹子，所以会说一些韩文。他早年也是谷歌亚洲区的，系统层面的活儿做得特别熟。他们三个都自己创过业，Square的工作对他们来说也是得心应手。

我的第一次编程马拉松就是和他们三个一起完成的。为什么我能有幸和这亲密无间的三剑客一起做项目呢？

主要是因为Square的每个新人，都会有一个类似于“辅导员”的老员工带着新人熟悉系统和技术，而他们其中有一个人就是我的“辅导员”。

我是以应届毕业生的背景进入 Square 的，很多应用层面的技术都一窍不通，所以Rails、Search、JavaScript 等好多技术，都是我的“辅导员”手把手带我入门的。

大概是我一直比较刻苦的缘故，他们一直对我还是很友善的；虽然我很弱，他们三个做项目，还是把我也加进去了。

我的第一个编程马拉松项目，就是和他们三个一起用 Elastic Search 重写 Square Search 的后端。那一周我们四个在公司做什么都很自然地在一起，收获之一是我们的项目做得很好，后来Square Search后端的原型就是基于我们的编程马拉松的项目。

收获就是交了三个好朋友，后来在Square我和他们也走得近很多。甚至离开Square，他们还时不时地给我发消息，偶尔也来找我吃饭。

# 第二个故事：技术，不一定是在代码里

我经历的第二个编程马拉松还是在Square。当时我有个想法，就是做一个类似于公司内部的 eBay，大家可以卖、送和交换一些家里不再用的旧物品。和我同组的其他人都是妹子，因为对这个想法感兴趣的都是女生，其他人有做产品经理的，有做 HR 的，还有做设计的，只有我一个人是程序员。当时我对于 Square Market 已经很熟了，代码量也不多，所以写代码基本上我一个人就能搞定。

一个叫 M 的女生给我印象很深，她是那种你看她一眼，就会轻易联想到 Fashion 的风格。她的头发经常是不同的造型，口红也是每天不一样，都是各种极其鲜艳的色彩，衣服也是很独特的设计。总之，从上到下你都能看得出她精心而夸张的装饰，但是总体又给人非常和谐的感受，就像杂志里的模特一样。

她在这个项目里，是帮大家选定的商品拍照。这是个有难度又很繁琐的事情，可是她真的很厉害。这么说吧，同样一对耳环，我拍出来的效果可能是大家觉得白送都不要的地摊货，可她把背景、灯光、陪衬一搭配，那对耳环你说是 Tiffney 卖的都不为过，高大上立显，这可能真的需要对设计和审美有着很独特的眼光。

那段在一起 Hack 的日子里，让我对公司的非程序员的职位有了很大的改观，也更深刻地体会到，技术不一定是在可见的代码里，可能更多的是，让一些无形的产品增值。这个项目我也接触了很多非码工的妹子，交了几个很要好的女性朋友。

# 第三个故事：和大牛结对编程是怎样的一种体验

我经历的第三个编程马拉松就是在 Airbnb 了。与Square的一整周不一样，Airbnb的编程马拉松只有三天；而且这一次其实我本身没打算做什么，因为全公司都去 Hack 了，那周得有一个人负责运维，处理线上突发问题，或是协调一些产品代码的改动。那次正好轮到了我运维。

JM 知道我没有打算做什么，就拉着我一起，做了一个 Airbnb 内部的员工静态定位系统。简单来说，这个系统就是在公司的内部地图上可以定位到找到某个人座位在哪，公司大一点的时候，平时大家需要找一个人的时候经常问别人，但很多人都说不清。

说到这里，就需要交代一下JM 是谁？他是我当年在 Square 一起奋斗过的小伙伴，不过我们在工作上没有直接的交集。这个人在支付领域六七年之后，完全是个架构师级别的大牛了。

当初我拿了 Airbnb 的 Offer，第二天意外发现他恰好也同“轨迹”跳槽，离职入职的时间和我只差两个星期。这样，我们就是 Airbnb 唯二的前 Square 中国人，更恰好的是，我们都进了支付组。住得又很近，我和他因为这层特殊的关系，在 Airbnb 就格外亲密些。

JM读书的时候就参加过微软搞的编程竞赛，在世界排得上名次的，所以在简历上比我是高出不止一截。平时他也喜欢随大流称我一声“安姐”，和大家各种调侃玩笑，也从没给过我技术差距上的压力。

在简历上看见一个人牛，和在实际工作中体验一个人牛，是完全不同的感受。当时一起做办公室地图，我和他负责研究明白谷歌地图的一个 API 怎么用，怎么把办公室地图集成进去，以试用 API 提供的一些旋转，放大，移动等功能。

JM做了大部分的工作，我只帮忙写了程序，做了球面坐标和平面坐标的转换以及边界的转换等。我的工作量比他的少，而且总有人给我发信息协调运维相关的事，所以我很久才把函数写出来。他拉过我打草稿的纸，看了两眼，想了几秒，就说：“这里不对”，然后动手三下五除二就把我的代码改了。

本来我还觉得不太可能，结果证明人家就是把你一个隐藏很深的 Bug 给干掉了，这一点真的十分厉害。两个人一起写代码，如果水平差不多，会互有收益；但是如果水平差得太多了，基本看着那个会写的写，自己跟着学学就好了。

于是我就乐得清闲，坐在旁边看他敲敲代码，自己干干杂活，偶尔端个

茶递个水剥个橘子。累的时候陪着聊个天，俨然一副“鼓励师”的架势。他也不介意，反正活都干了，干得挺麻利。所以我的第三个编程马拉松就这样轻松过关。

（在这里郑重申明的一点是：平时的工作中，我是非常认真的，这次马拉松是特殊情况，由于杂务缠身才稍显散漫，最后也贡献了一些技术代码。）

# 第四个故事：即使面对技术大牛，也可以表达意见

最近的一次编程马拉松，又是一个跟以往完全不一样的体验。这次一起做的是 J。J 不是我们支付组的，是 PI 组的，这个组是搭建和管理公司整体系统架构的一个组。

J在 Airbnb 是什么一个声望呢？除非你完全不碰系统，纯做前端或者移动，否则你列举的 Airbnb 牛人 J 必占其一。不管是系统设计审核，还是网站受攻击时的处理，还是正规化一些已有的软件，还是我司的开源等等，他都做得很好。

J完完全全走技术路线，和人员管理一点不搭边，因此他可以有更多的时间放在技术上。而且他很拼，真的很拼。几乎每次我加班晚睡，他都还在线工作。我发给他们组的所有技术讨论相关的邮件，他都在很短时间内回复并详细解答，而且解答都很中肯。总之，是我眼中纯技术的真大牛。

可能是因为我之前问过他编程马拉松打算做什么，所以他有了想法以后，就问我要不要一起做。他想做的什么呢？他有一个以前的环境警报 LED 灯，就是一个由电板控制的LED 设备，可以通过一些串行端口的编程，来让它在不同的情况下显示不同的颜色。

比如，如果 Airbnb 的网站挂了，就显示红色并不停闪动；如果现在某一个服务挂了，就显示橘色；如果某些页面的错误率很高，就显示紫色；一切正常就是绿色并且完全不闪动。介于红色和绿色之间的颜色以一个中间的频率闪动等等。

项目进行很顺利，除了技术细节没什么可写的。不过这次和牛人合作，我又有了一个不同的体会：就是不要害怕自己可能不如别人，就不敢出声或者不敢写代码。

刚开始我还比较小心，都是比较有把握的建议或意见才说出来。慢慢我发现，其实只要是自己真实的想法，说出来，有的时候可能别人真的就没有想到。

也许是个错误的意见，别人就可以告诉你为什么不对。以后再有类似的问题，就会有更多的了解和体会。所以，后面的两天，我们真的就是一边讨论一边做，在这个过程学了很多，更开心的是，觉得自己在这个项目中也是有真实贡献的。

经历了很多次编程马拉松，也就接触了很多类似 J 这样的极客，你看到这些技术大牛们在网路上从容不迫、侃侃而谈，是因为他们对技术和产

品近乎执着的关心，因为他们日积月累了很多实战经验；更因为他们在背后付出了超出常人的努力。很庆幸自己身边有着很多很多这样的值得我学习和尊敬的极客们。

最后总结一下我对编程马拉松的感受：

1. 编程马拉松是一项能够激发创意的活动，很多活动中的产品最终都成了成功产品的原型。
2. 根据公司的实际情况举办编程马拉松，掌握好节奏。
3. 编程马拉松不仅可以激发创意，还可以让你和团队的小伙伴建立特殊的感情和纽带，增加团队的凝聚力。
4. 参加编程马拉松尽可能选择一些可以扩展你知识边界的项目，开阔视野。
5. 与牛人结对编程，要敢于说出自己的真实想法，无论对错，都会有不同的收获。

最后，终于有机会说出我自己一直很喜欢的一句话：**Geek is the new sexy!** 用中文来说就是：会写代码的人真的就是很帅！

你经历过什么样的编程马拉松呢？可以在留言里说说。

# 工程师、产品经理、数据工程师是如何一起工作的？

做为一名工程师，免不了与产品经理打交道，如果公司大一些，数据量多一些，还会有数据工程师这个角色。今天会和你主要聊一聊在工作中，产品经理和数据工程师在哪些方面对我们工程师的帮助最大，以及我从他们身上都学到了些什么。

# 先来说说产品经理

我工作过的两个公司在早期的时候，很多服务器相关的研发组都是没有产品经理的。这种事在国内公司里比较少见，因为国内大部分产品都是由产品经理来驱动的，但在硅谷很多早期的公司里，这并不算太奇怪，毕竟很多产品经理的职责是工程师兼有的。

随着公司的发展和壮大，每个小组都有了固定的产品经理角色，我接触的产品经理也就慢慢多了起来。

在这个从无到有的过程中，对我帮助最大的是什么呢？我觉得应该是对项目边界和进度把控。

第一，当一个公司足够大的时候，虽然自顶而下管理，很多大的目标和方向在公司内保持了一致性，但是还有很多东西是需要讨论和确定的。

比如：产品和项目的边界如何确定，应该由哪个组做；如果两个组的工作范围有一定的重合性，如何确保两个组的工作是互补而不是重复甚至矛盾的；如果两个组要做的东西有一定的依赖性，在时间安排上如何保证被依赖的部分可以提前完成。

这样一些跨组协调，虽然技术管理者在一定程度上会有很大的话语权，也会参与所有的讨论和决策，但是有产品经理的帮助，他们可以更多把精力放在技术相关的问题上。

第二，不论你是在哪个组，做出来的产品一定是给人用的，使用的人就是你的用户。他们可能就是公司外部客户，也可能是公司内部另一个组的工程师。

有客户就会有需求，当有不同方面的人给你提出不同需求的时候，如何去合理地设定优先级，如何去和那些需求没有被满足的组沟通，如何有技巧地挡掉一些不那么重要的需求等等。这些都需要产品经理做大量的工作，让工程师把时间和精力更加专注地放在最重要最紧急的项目上。

第三，很多产品经理也会和技术管理者一起，兼任项目管理的职责。小到帮助安排各种定期不定期的会议，负责会议记录；大到帮助技术管理者一起制定项目的进度表，定期进行工程进度总结汇报等。

那么在这个过程中，技术经理或者技术管理者又给产品经理提供哪些帮助呢？

这包括：对某一个项目或者子项目的技术难度的工作量进行评估；给出多个可能的技术方案，包括长期方案和短期方案，以及每个方案的利弊；对组员能力的评估，知道谁可能去做什么，组里现在是不是有人有经验或者能力，去负责某个项目，攻克技术点等等。这样产品经理可以

更好地设计产品特征，设定需求边界。

从产品经理身上，我学了很多东西，这包括了：各种沟通能力，会议、邮件、一对一面谈、处理和制定优先级的能力、如何回绝不合理的需求，还有一些项目管理和追踪的技巧。

# 再来说说数据工程师

关于数据工程师的文章很多，我在自己的公众号和之前的专栏文章（每个工程师都应该了解的：A/B 测试）都略有涉及。这里着重讲讲他们在工程协作中什么时候和工程师的交互最多，以及在合作中有哪些地方让我印象深刻。

硅谷很多项目在产品开发初期就会有数据工程师的参与。

包括如何设计数据实验，了解产品或者项目是不是达到预期的效果；哪些老的数据和指标（ Metrics ）是需要监控的，以保证不会影响其他的产品或者项目；哪些地方可以通过数据模型给出最优方案 —— 这些地方通常是可以使用机器学习的部分。

当然，在很多专门的机器学习应用领域，数据工程师的重要性更加不言而喻。

在产品开发后期，或者产品发布初期，数据工程师会帮助调整各种监控系统或者预警系统，确保当系统出现异常，或者某些用户行为不在预料之中的时候，工程师们能第一时间了解情况。

与软件工程师相比，数据工程师有着不同的技术背景，他们的很多能力是和工程师互补的，但是有一件事，是我们可以去学习并为我所用的，那就是对数据的敏感性。

记得有一次，生产线上的一个 Bug，导致整个产品线有一个转化率变得格外差。因为问题比较严重，好几个工程师都去调试并定位错误，但是都没能找到问题所在。

结果他们的数据工程师通过观察不同指标下数据变化的时间线，在一些数据变化的异常曲线里找到了问题发生的关键时间点；再去找那个时间所有的改动，包括代码改动和灰度发布控制开关的变化，从而找出了问题。

另一个案例是听朋友说的，他也做支付，产品的一个代码改动，没有设置合适的异常，也就是该抛异常的地方没有把异常扔出来，结果数据下溢导致一笔金额数字错误。因为是很少见的场景，所以测试的时候也没有发现，但是数据工程师在数据图中的异常报警发现了这个问题，减少了一笔不必要的损失。

见微知著，其实我描述的案例，仅仅体现了数据工程师的一小部分价值。在大数据和人工智能的时代，数据工程师的作用正变得越来越不可替代。

今天主要讲了一些我和产品经理、数据工程师一起工作的故事，这其中

包括了我自己的感受。

做为一个软件工程师，有机会和他们一起工作，从他们视角去看待问题，可以拓展自己的知识领域和眼界，一方面可以把产品做得更好，另一方面，让自己成为一个既有产品思维，又有数据思维的工程师，会让你在这个时代具备更强的竞争力。

关于产品经理的主题，你如果有兴趣了解，可以去订阅隔壁邱岳老师的“产品手记”。

无论是你工程师、产品经理还是数据工程师，请在留言里写下你的故事，我们一起成长。感谢你的收听，我们下期再见。


戳此获取你的专属海报

# 技术人的犯错成本

今天和你分享一下我和周围一些朋友在职场曾经犯过的错误，供你参考和反思。基于表述的方便，我会采用第一人称或我的同事或朋友的方式讲述，读者们无需对号入座。

# 错误一：所有关于公司公关层面的事都不是小事

尤其是公司规模大到一定程度，这种事情更应该小心处理。

很多技术人都喜欢分享，但是分享的过程中要避开公司公关层面内容，更多去分享自己的技术和成长体验。比如我的公众号“嘀嗒嘀嗒”里，很多读者会留言：“安姐，可以说说 Airbnb 的某某方面怎么做的。”

这其中有很多内容可以写，但更多的内容是相对敏感的，不合适在个人作者的公众号里披露出来。所以我很少在公众号写公司的事情，毕竟“嘀嗒嘀嗒”并不是 Airbnb 的官方宣传工具，而是个人创作平台。以前我曾经出过类似的问题，一不小心，就会惹出不小的麻烦。

我曾在一篇文章中提及了 Airbnb 使用的一个第三方服务，表述方式其实都算不上负面评价，只是言辞中指出了这个公司的一些产品的瑕疵，结果该公司的公关直接联系了我们公司的公关及我们组的负责人，说这样的文章对他们公司的产品有负面影响。

最终这件事的结果是我把那篇文章删除了事，但是若细究起来，我也有被辞退的可能性。一个人具备了一定的影响力，所有公开的言论都可能会被认为，你代表了公司的观点。从那以后，我就很少去写公司相关的具体事件了，就算有涉及的也会泛泛带过。

后来我开始负责 Airbnb 的官方博客，每一篇博文的发布都是要经过公关审核的，当用户量形成规模之后，每一篇正式的稿件，如果措辞模棱两可，都可能引起误读。

作为技术人，如果你在任何场合有公开演讲或者文字输出，都要谨记自己可能被默认是公司的代表。

# 错误二：人事变动的宣布一定要谨慎

有一次，某公司空降了一位技术经理，不过按照公司惯例，一线的技术管理岗位都是需要做大概半年的工程才能正式转岗成为管理者。

所以，虽然大家都知道：只要半年内没有大的问题，他就会成为技术经理；但是，这个任命并不是立即宣布的。结果一位不明就里的同事给全组发邮件，欢迎这位新的技术经理，事情就弄得比较尴尬，管理层也觉得流程上出了问题。

类似的事还见过几次，比如，老板还没宣布谁离组离职，某同事就在公开的邮件中就提及此事等。或者在一些群组里，把未公开的、很多人其实并不知道的人事变动透露出来，显得自己消息灵通。在职场上来说，这些都是很不专业的举动。

这么做的代价有可能会让事情变得很糟糕，你也会失去领导和伙伴的信任。

# 错误三：关于技术或业务问题，如果你不确定，就不要随便给答案

我有一位朋友，他的能力不算差，本人很喜欢交流和分享，就是有时候爱出点风头。因为这样的个性，他比较喜欢在自己公司的各种群里回答别人的提问。

虽然说大部分时间里他的答案都是准确的，但确实也有几次，他给出的答案是错误的，或者不完全对，不过他并没有使用不确定的语气来表示他给出的信息可能不准确。

喜爱分享和帮助别人自然没什么错，但是对自己不那么确定的答案，最好用“我觉得”“可能是”，或者“我也可能是错的，你最好验证一下”这样的措辞，否则同样会失去信任。

即使你帮助了很多人，但还是会有人觉得你不那么靠谱。长此以往，即使你的答案是确信无疑的正确，大家对你的信任也会打个折扣。

一旦涉及技术和业务，都是可丁可卯的事情，很多时候对就是对，错就是错，没有中间态。技术人处理这些问题一定要严谨，千万不要丢失自己技术上的信用分。

# 错误四：技术失误导致错误的产品决定

在这个技术越来越重要的时代，我们常说工程师手握几千万甚至上亿的数据资产，如果对应到支付或交易，那就是真金白银。

有人和工程师炫耀说：“我炒股每天上下几十万浮动”，工程师们就会笑笑回复：我一行代码就是千万资金，这倒也所言非虚；所以涉及交易、产品决策的技术和数据问题，都要慎之又慎，反复考量犯错的概率和成本。

举个例子，某公司在做一个大的产品决策前，用 A/B 测试来测试不同功能对用户、公司利润、数据增长的影响，最后，所有的验证结果都一边倒地偏向新特性 B。

那么，公司就投入资源和人力全力研发 B 特性，可是等到 B 特性上线后半年，工程师才发现数据实验设计中有一个参数错误， B 并不是最优选择，甚至都不是一个好的选择。

这个失误导致公司白白损失了一大笔研发费用，还有上千万的产品收入，最后，公司还需要花费很大的力气把 B 特性下线。

事实上，有时候技术人犯的错误会有很严重的后果，损失可能是一辈子的工资都赔不起的。

比如我曾经看到的一个案例，一个游戏公司的工程师手抖把公司的用户数据库删掉了，而恰恰这个数据库的自动备份由于某种原因在一个月前停掉了，最后这个公司真的就倒闭了。从删库到跑路，并不是传说中的故事。

这一期，我和你聊了聊技术人可能会犯的各式各样的错误，那么说，把错误拎出来单独谈，目的是什么呢？

我想，错误告诉我们两个重要的事情就是：第一要去试错；第二要从错误中成长。

在工作中，很多公司都会鼓励试错，鼓励多尝试，但某些时候，工程师们的错误成本是很大的。这些错误可能会引起比较大的麻烦或损失，也可能会让我们失去信用和机会。

这并不是说我们要裹足不前，什么都不做，而是去认清我们犯错的成本，知道我们可能会犯什么错误，避免去犯致命的错误。

在下一篇文章，我会介绍一种关于错误的分类法，以及技术人如何从错误中成长，如何避免犯没有价值的错误。

你在工作中有什么样的错误或教训呢，可以在留言里告诉我，我们一起

从错误中成长。感谢你的收听，我们下期再见。


戳此获取你的专属海报

# 硅谷人如何做CodeReview

今天技术管理课的主题是： Code Review，也就是我们常说的代码评审。Code Review 主要是在软件开发的过程中，对源代码进行同级评审，其目的是找出并修正软件开发过程中出现的错误，保证软件质量，提高开发者自身水平。

和国内的工程师聊天，发现国内公司做代码评审的比例并不算高，这可能和各公司的工程师文化有关系。不过硅谷稍具规模的公司，代码评审的流程都是比较规范的，模式也差不多。

# 首先是两个概念

在进入正题之前，先介绍两个概念，一个是 Commit，一个 PR。硅谷大部分公司都使用 GitHub 企业版管理自己的代码仓库，GitHub 里的 Commit 和 PR 的概念在代码审核中非常重要。

**1 Commit**。就是 GitHub 上的一次“ Commit ”行为，这是可以单独保存的源代码的最小改动单位。

**2 PR**。也就是 Pull Request，是一次代码提交请求。一个 PR 可以包含一次 Commit，也可以是多个。提交请求后 GitHub 会在相关页面上显示这次提交请求的代码和原代码的所有不同之处，这就是本次 PR 的所有改动。

请求提交后，其他工程师可以在 PR 的页面上提出意见和建议，也可以针对某一些代码的改动进行讨论，也可以给整体评价。代码的作者也可以回复这些意见和建议，或者按照建议进行改动，新的改动将为本次 PR 中提交新 Commit（也可以覆盖之前的 Commit）。

关于 GitHub 和 Pull Request，池老师最近在他的公众号里写了一篇“ GitHub 为编程世界带来了什么改变 ”，这篇文章中有比较详细的描述，你可以参考学习。

# 其次，我来谈谈代码合并规则

一般情况下，所有的 PR 都必须有至少一个人认可，才能进行合并。如果改动的内容涉及多个项目，则需要每个项目都有相关人员认可才能合并。还有一些特别关键的代码，比如支付相关的，通常也会需要支付组的人确认才行。

在代码合并之前，进行 Code Reivew 的工程师们会在 GitHub 的相关页上给出各种评论，页面是共享的，这些信息大家都能看到。

有些评论是询问，代码的作者直接回复或解释就行，有些是指出代码的问题，代码作者可能会据此改动，也可能不会同意，那就需要回复评论，阐述观点，你来我往。有时候一个实现细节，讨论的主题可以多达十几条或几十条，最终需要达成一致才能进行合并。

# 再次，帮助别人成长，而不是帮他写代码

以前有朋友会说：“看到代码写得太差，觉得来回评论效率太低，干脆自己冲上去搞定”。曾经我也有过类似的想法，不过我慢慢意识到这并不是一个合适的想法。

首先，从对方的角度来说，代码写不好，可能是对业务不熟悉，对编程语言不熟悉，也可能是对公司代码的整体架构不熟悉。如果你帮他“写”，而不是耐心指出哪里有问题，那么下一次他可能还是不知道怎么做。这样不仅无益于别人成长，有的时候甚至会让别人有挫败感。

并且，帮别人写代码的方式可扩展性很差。即使 Code Review 会花掉十倍于你自己写的时间和精力，但它会让人明白代码应该怎么写，从长远来看，这其实是在一定程度上“复制”你的生产力。

你不能什么都自己写，尤其是开始带项目、带新人以后。每天 Review 五个人的代码和写五个人的代码，长期而言哪个更合算呢？答案显然是前者。

写代码是一个学习过程，怎么做一个好的代码审核人更是一个学习和成长的过程。自己绕过一个坑不难，难的是看到别人那么走，远远地你就能告诉他那里有个坑，而他在你的指导下，以后也会帮忙指出别人的类似问题。

我在这一点上最近感触尤为深刻。随着团队越来越大，新人也越来越多，有一段时间我每天工作的一半时间都在 Review 别人的代码，写评论。

这样做的初期确实比较辛苦，不过效果也随之慢慢显现，大部分时间工程师们已经可以进行相互 Reivew 代码了，于是我可以腾出很多时间来做别的工作，代码质量也得到了保障。

# 提交代码的类型

在进行 Code Review 之前，要搞清楚提交的代码到底是干嘛的，然后进行针对性的审核。我们一般把提交的代码分成四类。

1. Bug 修复。一般公司都有独立的 Bug 追踪和管理系统，每个 Bug 都有一个票据。代码提交的 PR，一般和票据是关联的。
2. 代码优化。比如文件的移动和拆分，部分函数的重构等。
3. 系统迁移。包括代码库拆分，用另一种语言重写等。
4. 新系统和新功能。新功能在实现之前都需要进行设计审核，最终版本的设计文档会包括数据库的 Schema、API 的签名（ Signature ）、代码的流程和模块等内容；相关代码的提交，也就是 PR，一般是和设计文档挂钩的。

了解了提交代码的作用，审核就会更有针对性和效率，也更容易从作者的角度阅读代码。

# 最后说一下 Code Review 的注意事项

从代码提交者的角度，在代码审核中需要注意哪些问题呢？

第一，为什么要进行 PR ？原因一定要在提交的时候写得非常清楚，才能帮助审核者理解这个改动是不是合理。上面说的四种提交代码的类型，具体是哪一种，应该写到 PR 的小结中，写得越详细越好。

这在以后需要进行回溯或追踪系统变化时，也是很有益的。如果改的是前端代码，最好贴一个改动前和改动后的截屏，让改动效果一目了然。

第二，除非是极其明显的单词拼写问题，尽量不要引入不是这个 PR 目的的改动。PR 要尽可能保持目标的单一性。每次遇到有人把一些代码结构的优化合并到功能相关的改动时，我都有一种肝火上升的感觉。

这种行为不仅会增加审核者的困难，降低效率，还会掩盖一些简单的错误。并且，如果因为功能的修改导致线上出了问题，一般需要退回到之前的版本，也就是反转PR ，这时候，针对优化相关的改动也就必须被反转。总之是弊远远大于利。

第三，找谁审核？除了本组的人外，有时候代码还会和其他项目组的代码相关，需要找该组的成员审核，这时具体找谁呢？

一般有两个机制来解决这个问题。一是在 GitHub 里 @ 一个组，比如 Payment 组，Risk 组等等，这些组会通知组里所有的人，相关的人看到了就回去审核；二是有一些组的代码，不希望其他组的人在自己不知道的情况下进行改动，就会设置规则，如果有人动了这些代码，也会通知到整个组。

最后，也是最重要的，一定确保所有的改动都是测试过的，无一例外。

从代码审核者的角度，又需要注意哪些问题呢？

审核的粒度要多细？是不是每次审核都要花很多时间？当然，如果时间足够，自然是看得越细越好。如果特别忙的时候，可以做一些筛选。

比如，你可以看一下算法或者编程思路，然后加一个评论 “算法部分看来没有问题”；也可以只看你关心的部分，然后加评论 “支付部分没问题”，或者 “API 部分没问题”。还可以再 @ 一些你觉得可以对其他部分加评论的人。

另外，如果是新人的代码，尽可能地在代码风格、性能等各方面都加以审查。如果是一个老员工，这些方面可以给予更多信任。

具体哪些地方需要审核呢？总结一下。

**1** 代码格式方面。很多公司都有编程语言风格指南（Coding Style Guideline），这是大家的约定俗成，避免公司的代码风格不一致，也避免了一些“要不要把闭括号另起一行”的无谓争论。老员工除非不小心，通常大家都不会弄错；新员工在这方面不太熟悉，就有可能出问题。这一类问题是比较容易指出的。

**2** 代码可读性方面。比如函数不要太长，太长就进行拆分。所有的变量名要能说明它的用意和类型（比如 hosting_address_hash，一看就知道是房东地址，而且是个哈希类型）。

不要有嵌套太多层的条件语句或者循环语句。不要有一个太长的布尔类型（Boolean）判断语句。如果一个函数别人需要看你的长篇注释才能明白，那这个函数就一定有重构的空间。另外，如果不可避免有一些注释，则一定要保证注释准确且与代码完全一致。

**3** 业务边界和逻辑死角问题。你可以帮助代码作者想想，他有没有漏掉任何业务边界和逻辑死角问题。很多时候这是业务逻辑相关的，尤其需要资深一点的工程师帮助指出需要处理的所有情况。

**4** 错误处理（**Error Handling**）。这是最常见的问题，也是代码审核最容易帮别人指出来的问题。

我在文稿中举出了一个例子，一段简单到不能再简单的代码就至少有三个潜在的问题。这些都是需要审核者注意的。

```
def update_user_name(params)
    user = User.find(params[:user_id])
    user.name = params[:new_name]
    user.save!
end
```

1. params 里面需要 validate 是不是有 user_id 和 new_name 这两个 key
2. 能不能找到这个 user_id 对应的 user
3. save 的时候会不会有 DB level 的 exception，应该怎么处理

**5** 确保测试用例覆盖到了所有的功能路径。严格来说，每段代码都应该有测试用例。如果开发者能够预见到其他人的代码改动会引发自己的代码问题，一定要增加额外的测试用例防止这种情况的发生。

**6** 代码质量和规范。遵循公司制定的编程规范，比如，如果有重复的代码段，就应该提取出来公用，不要在代码里随意设常数，所有的常数都应该文件顶部统一定义，哪些变量应该是私有的，哪些应该是公有的，等等。

**7** 代码架构。包括代码文件的组织方式，函数是不是抽象到 lib 或者 helper 文件里；是不是应该使用继承类；是不是和整个代码库的风格一

致；API 的定义是不是 RESTful 的等等。

# 公司层面的支持

从公司层面应该有哪些措施帮助员工有效地进行代码审核呢？

1. 统一的代码提交和审核流程与工具，并确保大家使用同样的工具，遵循相同的流程。
2. 鼓励员工帮助别人审核代码，甚至可以做到效绩评估中。
3. 制定统一的编程规范和代码风格，尤其是在有争议的地方，这样可以解决一些因为程序员偏好带来的矛盾。

代码审核和编程一样，都是日常工作，不要情绪化。我曾经做过一件事，一个外组同事的代码，别人已经认可合并了，可是我觉得有问题，于是反转了流程，在评论里写了原因和建议的改法；当时心里还觉得会不会得罪人。可是后来发现同事反而很客气地接受并道谢了。

另外，评论里经常会出现不同意见，其实是两个人在编程习惯和约定俗成上没有达成共识。如果在不违背公司风格指南的情况下，没必要一定让对方和你有一样的习惯。

如果你真的觉得这样做更好，可以委婉地提出来，比如 “我个人是更偏向于A类型的编程风格，不过这不是一个硬性规定。” 或者 “嗯，改成A类型的编程风格如何，不过这不是强制的。”

今天我和你讨论了 Code Review，主要内容包括 Code Review 中的重要概念，代码合并的规则，帮助别人评审代码而不是写代码，提交代码的类型和做 Code Review 时的注意事项。

事实上工程师们在编程的时候很难保证万无一失，多几双眼睛一起帮你看一遍，就可以很大程度地减少代码里的错误。另外，相互审核的过程中还能从彼此那里学到很多编程的小技巧和好习惯。细想来，很多 Java 和 Ruby 的特别好用的技巧，我都是在做代码审查的过程中学到的。

你的团队有没有做 Code Review 呢？有好的经验可以在留言里告诉我。感谢你的收听，我们下期再见。


戳此获取你的专属海报

# 如何从错误中成长？

在上一篇文章“技术人的犯错成本”里，我和你聊了技术人可能会犯的各式各样的错误，也举了很多例子，说明了技术人犯错的成本。在竞争激烈的互联网时代，试错当然是好事，但了解错误成本，避免不应该犯的错误，最大可能地从错误中成长，才是我们应有的态度。

我之前看过一篇文章，其中把错误分成四类。

第一类：伸展错误（**The stretch mistakes**）

伸展错误是：当你尝试去做能力之外挑战的时候，因为自身能力或其他条件的束缚，做得不够好而犯的错。这种错误通常是因为我们主动尝试导致的，再小心也难以避免。不过，在此过程中，你获得学习的机会成本很高，一旦经历过一次，就可能有长足的进步。

比如在创业公司，大家有个创新性的想法需要去验证和实现，这时候谁都没经验，也没有前人的产品可以借鉴，也没有很懂的人给你指导，那怎么办？只能摸索前行，遇到问题，解决问题，犯错，然后纠正错误，所谓逢山开道，遇水搭桥，就是这个道理。当你抵达彼岸的时候，你会发现自己已经升级了，进入了另一个境界。

第二类：无知错误（**The aha-moment mistakes**）

无知错误是指：当你发现自己为什么错了的时候，你会发出“噢，原来是这样”的感慨。

这种错误一般是因为你不知道或忘记考虑某些特殊情况导致的错误，或者是你做了错误的假设。

初级程序员很容易犯这样的错误，比如忘记处理异常，没有考虑某些数值的边界值，没有进行安全校验等等，还有人因为没有仔细阅读产品文档，不知道产品设计已经改变了，也没有进行对应的调整。出现这种错误之前，你主观上根本没有意识到犯错的风险，还以为自己一直是对的，以至于出错后会有那样的反应。

这种错误一般不会反复出现，但要尽可能避免在不同类型的事情上犯同类型的错误。

第三类：粗心错误（**The sloppy mistakes**）

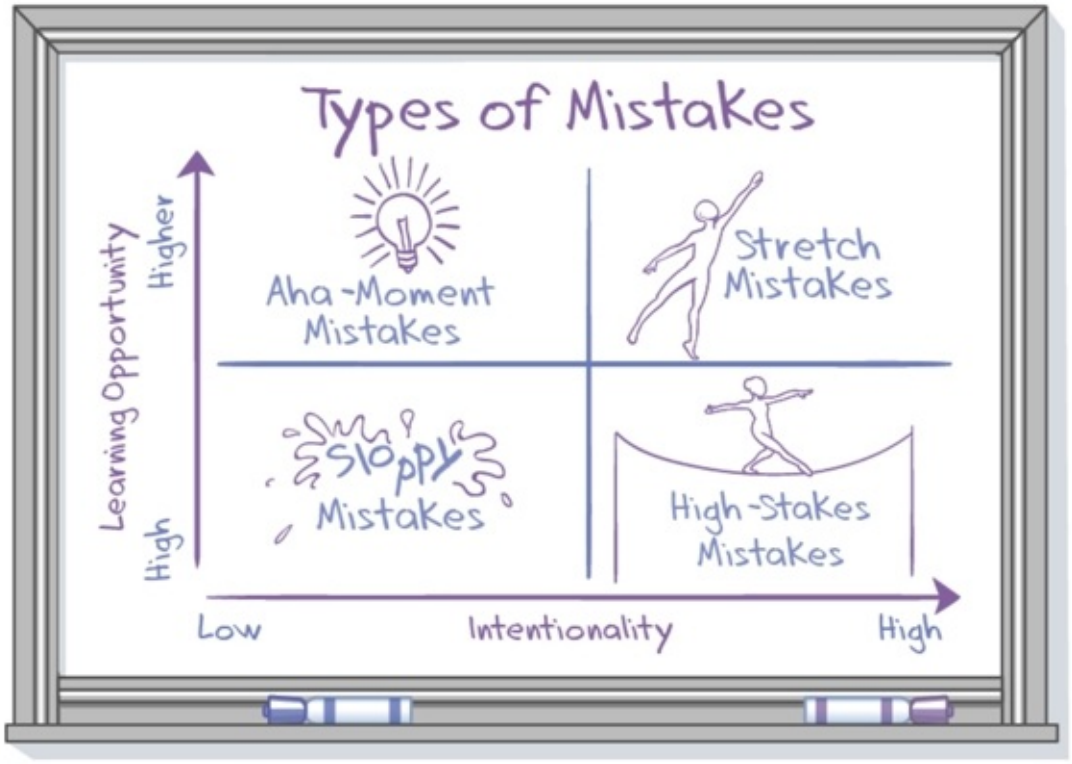
直译过来就是由于粗心大意导致的错误，很容易理解。这种错误与“无知错误”不同，是你明明知道怎么回事，但是因为不小心或者忘记了导致的错误。如果你是个粗心大意的人并且没有有意识地纠正自己，这种错误可能一犯再犯。

第四类：高风险错误（**The high-stakes mistakes**）

主动去做事情，但风险很高，是否会犯错不受自己的控制。比如你面临一个重要的选择，但在结果出来之前，你之前掌握的所有信息都无法告诉你哪个选择是绝对正确的，你只能去做自己认为是大概率的选择。

这种错误与第一类不同，经历过一次，下次遇到高风险的事件，依然可能犯错误，并且不知道会犯什么样的错。

以上我们介绍了四种错误类型，我在文章中放了一张图片，如图所示：第一类和第四类更具意识性，也就是主动去做事情，做事之前你就知道有可能出问题；第二类和第三类则属于下意识的错误，更加不可控。




从另一个角度来看，第一类和第二类错误都会让你学习到更多有价值的、可重用的信息，再有类似情况基本上不会犯重复错误；而第三类和第四类每次犯错后并不能学到太多可以复用的信息，以后有类似情况很可能再犯同样的错误。

了解了这些错误和错误分类，我们就可以有计划有意识地去应对这些错误，尽可能不犯那些对我们成长没有意义的错。当我们面临一个犯错和成长并存的机会时，我们也要知道如何去学习并避免犯同样的错误。

有哪些措施或者方法可以系统性的让我们避免那些无效错误，并从错误中成长呢？

**1** 为了避免伸展错误，尽可能地提供一些培训机制。比如对新上任经理的培训，组织一些特定领域的学习班，系统地帮助大家积累岗位可能需要的技能和信息。让一个人担任某个全新的职务时，考虑导师机制，工

作中有人结对给予实时指导，等等。

**2** 为了避免无知错误，要做好信息的透明和共享。比如完善的文档和快捷的查询机制，任何技术或产品讨论以及达成的共识，要尽可能用邮件抄送到所有相关的人。大的改动可以举行一些会议确保所有人都得到最新的信息。一些常犯的错误或者容易被漏掉的信息可以整理成 Checklist 或手册等形式，让大家更方便地获取到有效信息。

**3** 为了避免粗心错误，设置一定的复盘机制。如果是粗心大意造成的错误，要再三总结，反复问为什么，帮助加深记忆。另外还可以通过一些流程来确保低级的错误不会发生，比如细化开发流程，设置检查点，不断迭代和反馈，慢慢形成全面考虑问题的习惯。在不同的事情上，多给自己做一些 Checklist，做事的过程中对照进行，确保不会因为一时大意而有所疏漏。

**4** 为了避免高风险错误，所有的决定尽可能都有一个备用方案。也就是除了 Plan A，还要有 Plan B。这样当事情没有按照预期去发展的时候，我们为错误付出的代价也会更少。

总结一下，今天我和你分享了“技术人如何从错误中成长”的话题，首先，我谈到了工作中常见的四种错误类型：伸展错误、无知错误、粗心错误以及高风险错误。前两种错误可以帮助我们迭代成长，后两种错误代价较高，我们要尽量避免一犯再犯。

其次，我讲到了如何去避免以上的错误，针对四种错误类型，要分别采取不同的方式对待。因地制宜、对症下药，让错误成为丰富的经验，为你的成长提供充足的养分。

你在工作中犯过错么，在这个过程中你又获得了什么的成长。欢迎在下面留言，我们一起成长，感谢你的收听，我们下期再见。

戳此获取你的专属海报

# 理解并建立自己的工作弹性

在设计软件系统的时候，我们常常会提到弹性架构设计。一个好的软件系统就像是一个具备弹性的有机生命体，能够动态地改变自己的特性或功能，调配资源，去适应外界的变幻并满足不同的需求。

具备弹性的架构才是有生命力的，比如系统压力和容量都是重要的变化维度，如果一个系统能够根据系统压力和容量的增减而动态地调整自己的处理能力，我们说这样的架构就是弹性架构。

今天我要和你探讨的不是技术上的弹性，而是工作的弹性、心态上的弹性。其实，从某种角度而言，道理是相通的。

我们经常会从朋友圈或各种媒体上看到渲染焦虑情绪的文章：三十而立了，焦虑；每天要学的内容太多，学会的太少，焦虑；AI 时代来了，到处是机器学习，你还在自己学习，焦虑；开始一份新工作，但是似乎离梦想更远了，焦虑；诗和远方到底在哪呢，只看到了眼前的苟且，焦虑。

有些人可能只是过过嘴瘾，宣泄一下情绪，但有更多的人确实感受到了方方面面的压力，并且真的开始焦虑起来。比如你平时的工作本来就很紧张，项目一个接着一个，到处是里程碑和 Deadline，老板又很严厉，然后你就会变得敏感、紧张和焦虑。

为什么同样的事情，同样的要求，组里其他的同事看起来都游刃有余呢？真实的情况可能是，别人也很焦虑，只是表面上看不出来而已。

事实上，不安、轻度的焦虑，几乎会伴随人的一生。那些有抱负、有上进心并取得过成功的人，同样会有各种坏情绪，只是他们自身的系统弹性比较好，可以很快地消化和处理这些情绪，而轻度的焦虑和失控状态，有时候更容易激发人的创造力，并保持机体的活力。

前几天和池老师、隔壁专栏作者二爷讨论过一个行为，英文叫做 Rumination，中文直译过来叫做反刍，在这里想要表达的意思就是：反复思量，钻牛角尖，用消极的态度不断地去思考自己的压力、逆境，以至于走不出来，最后形成了焦虑的情绪。

那要如何走出来呢，如何让自己在压力下可以最大程度地成长，而不是因为负面情绪止步不前呢？其实就是构建自身的弹性能力。

首先要意识到，什么是 **Rumination**，什么是正确面对自己的压力的方式。

对于已经发生的事，思考和复盘能够让我们总结教训吸取经验，但是不断地反刍自己的错误，则会让我们后悔并陷入不断自责的陷阱；而对于

还没有发生的事情，思考能让我们对未来有更好的计划和规划，但是一根筋琢磨自己的不足，只会让我们觉得自己离理想越来越远，从而产生焦虑情绪并裹足不前。

其次，意识到反刍的危害，当我们有压力的时候，就要让自己走出思维的误区，打造自己的弹性能力。

比如，我们可以用纸和笔列出能够做的改进，然后进行定期 Review，看改进是否有效果。把注意力放在那些自己能控制的地方，不要过多地因为自己无法控制的因素而责怪自己。知道每个人都有弱点，有局限性，不要过高地估计自己的能力。

有些人平时睡眠很少，大部分时间用来学习和工作，但是偶尔也会有完全不想干正事的时候，那么给自己放个假，逛逛街，睡个懒觉，或者在家瞎折腾消磨时间，都是很正常的事。

每个人都有乏的时候，偶尔放纵自己，也是弹性的一种表现，没什么可自责的，不要觉得浪费了时间。文武之道，一张一弛，紧张久了，松上一扣，然后再层层递进，学到的东西必然扎实。

哪怕真的做了什么愚蠢的事情，好好总结为什么会这样，下次不犯同样的错误就行了。最多自嘲一下，谁不会偶尔笨一下呢？遇到坑就赶紧翻过去，后悔并没有太多的帮助。

如果担心自己能力不够，不能把一件事做好，那不如把注意力放在改进和提高上，找到自己的差距在哪里，哪些地方是自己要持续提升的。把注意力放在当下一个个可以实现的小目标上。只有宏伟的目标却没有一个可执行的方案，必然引发焦虑。

再次，身边要有一些积极向上的人，与他们做朋友，没有的话就去努力结识这样的人。看一些积极向上的文章，保持良好的心态。

当自己陷入迷茫的时候，有个人可以倾诉或者咨询。有的时候，你觉得自己特别惨，事实上别人都曾经经历过。知道了事实的真相，就好像过一条长长的黑暗隧道，你知道出口在哪里，也知道一定能够走出去，就不会太迷茫。

另外，要确认自己在做一件有意义的事情。这很重要，因为你一直在向自己希望的方向前行。如果你觉得自己正在做的事完全没有意义，不论你多努力都没有帮助。这个时候，不如停下来，仔细想想自己该往哪走。

最后，想要构建出自己的弹性能力，还要具备以下两个心态。

1 不论你多努力，也不是所有的付出都会有回报，或者有等比的回报。

有一次，组里的小朋友跟我聊天，说出一个困惑：我想做得更好，所以我付出的努力比别人多得多，做的事也多得多。我觉得自己做得也很好，为什么似乎很多好事都没有轮到我呢？

我是这么回答她的。

很多努力，不一定有立竿见影的回报。当我们做一些超出自己职责范围事的时候，应该是自发的，而不是一定要什么回报。否则，做了之后一直纠结回报，反而不利于成长，也不能持久。

我们的努力就好像一道线，有的人做的工作还在线下面，却因为运气、时机、项目、公司等因素收到了好得多的结果；而有些人，付出和努力已经超过了那条线，却久久看不到成果。这个时候需要的是耐心，因为你已经攒了足够的信用分，一旦机会到了，就能顺理成章拿到你应得的东西。

2 不只是你，每个人都有自己搞不定的时候。英文有句话很有意思：Fake it，until you make it。意思是：假装就好了，一直假装最后就成了真的。不要去考虑别人做成了什么，或是有什么技能。把自信放在自己身上，按照自己的步伐和速度，每天进步一点点。

别人比你强，也许也是装的呢？想学什么，就去学，想做什么，就去做。工作上也是，公正地看待自己的能力，不断给自己设定比自身能力高一点的目标，然后努力去做，你会发现自己的弹性能力远远超乎你的想象。

今天，我和你聊了个人在工作和生活中的弹性能力。什么是弹性架构，什么是人的弹性能力。

人为什么会焦虑呢？可能是自身的弹性能力不够，事实上人的弹性能力也是可以锻炼和提升的。那如何打造自己的弹性能力呢？要正确思考而不是钻牛角尖，要找到需要改进的地方持续改进，偶尔放纵，不担心犯错，从错误中成长，结识乐观的朋友，一直做自己认为有价值的事情。

最后还要具备好的心态，知道付出不一定有回报，每个人都有自己搞不定的事情。日拱一卒，不期速成。

感谢你的收听，我们下期再见。


戳此获取你的专属海报

# 如何对更多的工作说“不”

生活中，我们常常会讲拒绝的艺术，一个人如果不懂拒绝，永远说“ Yes”，一定难以远行。越来越多的“ Yes ”，会让你背上沉重的包袱，无效的社交、超出能力范围的帮助、不合理的要求等等，都会耗尽你的时间和资源。

大部分人都会说“不”，但要注意拒绝的艺术，维护好关系和礼仪。今天我主要和你聊聊如何说“不”，再具体一点，如何对更多的工作说不。

在回答这个问题之前，有个更重要的问题是：什么时候我们应该说“不”？

工作中很多时候我们是很容易说“不”的，比如一个需求评估了之后，发现无论如何在期限内都没法完成，或者技术实现上超越了自己的能力。这时候说“不”并不困难，一来你知道自己有分身也做不出来，另外需求方也能够理解这种悬殊的要求对你来说是不可能完成的任务。

难的是什么呢？自己觉得加加班，努力跳起来够一够，或许能搞定呢。或者需求方认为这就是你的事，你应该搞定：“这个不难啊，怎么会做不了呢？”碰到这种情况，无论是对自己说“不”，还是对别人说“不”，都困难得多。

踮起脚尖做事，当然是好事。读库的创始人六哥曾经在访谈和文章中屡次提到“踮起脚尖阅读”这个概念，他说：

> 人在青年时期，思维如火山活跃，激情如大雨滂沱，没有成见的束缚，没有物质的负担，没有世俗的压力，想不怀疑人生、批判社会、自己给自己找别扭、思考终极问题都难。在这么好的年龄，就应该读一些踮起脚尖才够得着的书。

做事是一个道理。但如果踮脚尖太久，亦或够得太高，容易失去平衡甚至摔倒，那么我们的弹性到底有多大呢？踮起脚尖能够到的事，值不值得自己做一些牺牲去达成呢？

要对更多的工作说“不”，我觉得需要搞清楚两件事：

1. 分清事情的轻重缓急，有些紧急的事情和优先级高的事情很难拒绝；
2. 正确评估自己的能力和时间资源。

项目要上线了，你的代码实现是最后一个关键路径，这时候你说：“不，我还有些编程指南没有写完”，估计产品经理伙同业务方已经

拿着板砖就来找你了。

如果你写的代码引发了线上 Bug，每一分每一秒都可能导致用户流失或者资金损失，这时候你说：“不，等我睡一觉明天再说”。即使你能睡得着，明天可能也要去找 HR 领盒饭了。

对于这种很紧急的事务，我们需要在短期内放下其他事情，哪怕再忙也要说 “Yes”，尽自己最大努力去处理。我刚来 Airbnb 的时候，经历过一次公司账户审核，需要搭建数据环境，测试并且要处理所有不匹配的数据。

那时候支付组一共就十几个人，能用的都投入到这个项目上了，如果不能按期完成，Airbnb 有被停止支付的风险。那一次项目大概持续了二十多天，没有周末，晚上大概睡四五个小时，所有的时间和精力都放在了这个项目上。没有人说“不”。

最后项目按期完成，公司没有遭受任何损失，项目组的人额外得到一周假期作为补偿；但是对我们而言，更大的收获是建立了自己的信用度，公司和同事会觉得这个项目的人就是靠谱。

所以，事从紧急，在攸关大局的事情上，尽可能避免说“不”。

那什么时候可以说“不”呢？

1 正确评估需求后，如果事情没有那么紧急，或者这件事别人也能完成，并且你手头正好有更重要的事情在做，告诉对方，这个需求现在做不了，需要完成当前任务之后才能考虑。

从对方的角度看，如果这个需求对他很紧急，他会去协调其他的资源完成，而不是非你不可。这种需求你如果在分身无术的情况下答应了，却没有立即处理，会让别人觉得你言而无信。

2 如果某件事的处理已经远远超出了你的能力，甚至职责范畴，不要只是为了别人高兴而充当滥好人，揽下来又做不了，或者达不到对方的预期，不仅会耽误人家的事，还会丢失自己的信用分。这时候要学会说“不”。

3 如果某件事你可以做，但是因为各种原因，自己觉得有些难处，又不愿意解释就勉强答应下来，最后的结果可能是事情做了，却做得心不甘情不愿，完成度也不够。这时候也可以考虑说“不”。

与生活中相比，在工作中说“不”肯定更理性，但是也不要强硬拒绝，让对方觉得你是因为不愿帮忙才拒绝的。

无论是自己当前正在处理更重要的事，或者是觉得自己不是最佳人员，亦或是因为其他原因不能承担某个连续的任务，都要开门见山，把自己的想法和事实清晰地表达出来，不要用借口，承认自己能力有限并不是什么丢人的事，继续努力就好了。

把说“不”的理由解释清楚，并请对方谅解，同时可以酌情表示自己愿意帮忙的意愿，并询问有没有其他的协助方式，等等。

总结一下，今天我主要和你探讨了和如何在工作中说“不”，尤其是如何对更多的工作说“不”。想搞清楚这一点，我们必须区分事情的轻重缓急，知道哪些境况里可以说“不”，哪些不能。踮起脚尖做事对我们扩展自己的弹性是有帮助的，但是在时间和能力都有偏差的时候，记得要去说“不”。

答应一件事情之前，要想清楚是不是同时有更重要的事，能否并行，以确保不会有时间或其他冲突。答应一件事情之后，尽可能地兑现自己的承诺，如果出现计划外的情况或者难处，及时沟通，多替对方着想。同时也要记住，在工作上，对自己力不能及的事情，提早说“不”，无论对人对己，都是负责任的表现。

你有过对更多工作说“不”的经历吗，在下面写下你的故事，我们一起成长。


戳此获取你的专属海报

# 尾声：成长不是顿悟，而是练习

七七八八的，专栏的事就快要落下帷幕了。

每个技术人成长到一个阶段，都会去思考：下一步到底要怎么走？是要在一个自己比较陌生的领域开辟一块新的天地，还是在自己比较擅长的领域继续精进？要不要转管理、转产品，要不要去搞机器学习、搞区块链，哪个领域在风口浪尖，就投身到哪个领域么？

你会听到太多不同的声音，每个声音都似乎那么有道理，比如，要勇于尝试、要勇于走出自己的舒适区；又比如，如果你连自己现在的事情做不好，换一件事情也很难做好。会有人告诉你纯做技术很容易就到了自身的天花板，也会有人告诉你转了管理就丢失了你原有的技术优势和核心竞争力。

我迷茫过么？迷茫过。

记得有部电影里（《清洁》）里曾这样说过：“事事顺心的时候，勇气来得也容易；但是当生活变得艰难，勇气就弥足珍贵了。”

同样的是选择。你的人生一帆风顺的时候，你很容易相信你自己的选择；但是当人生一次次给你亮红灯的时候，甚至告诉你此路不通的时候，你就会慢慢后悔自己的选择，怀疑自己的选择，甚至再有机会到来的时候，你会彷徨纠结，总是对自己做出的选择唏嘘不已，也许山那边风景更好呢。

那要怎么走出迷茫面对选择呢？

答案就是不停地成长，在自己选择错误的时候，也知道自己在这条路上是不是比起之前的自己更成熟、更有见识了。只要在不停成长，就不怕犯错误，就知道自己还有机会。反之，哪怕一路坦途，事事顺心，但是自己没有什么成长，反而更容易陷入迷茫。

那要如何不断提高自己呢？

中国有句古话，叫做：“闻道有先后。”这句话在互联网时代，已经没有什么意义了。所有的道理，尤其是好的道理，大家几乎可以在网上同时间看到。所有的知识，大部分都是你想要接触就能接触到的。

这就像修炼武功，古时候每个人能看到的武功心法都非常有限，你入了哪一门，就练哪一门的功夫。而现在，所有的武功心法，不管多牛的，内功还是外功，都可以在网上找到，那每个人的区别是什么呢？

答案无非是两个，第一是找到那些合适你的道，尤其是合适你当前的道。这个道，既是道路，也是道理。第二就是练习。

我身边的人，包括我公众号与专栏的读者，论技术、论能力、论经历、论职位，比我优秀的人比比皆是。相反，因为我的普通，很多人可以从我的经历中看到他们自己，从我的感悟里读到技术人的共鸣。那些我挣扎过的，奋斗过的，失败过的，迷惑过的，也是他们曾经经历过、体味过的。

所以那些对我有益处的道理、经验和教训，我会分享出来，也许会帮助那些同样困惑过和迷茫过的人，让他们在寻找自己的道路上多一些启发；而且这种随着我的成长展开的分享，更具有阶段性。

即使是同一个人，在人生不同阶段使其获益的感悟也很不相同。有时候读者留言，觉得我讲的对他很有启发，那是因为我碰巧走在了他的前面。偶尔也会有人觉得“你的文章说得太浅显了”，我也不会介怀，因为他可能碰巧走在了我的前面。

去年的文章，也许我今年就写不出了。因为站在当前，突然会觉得那个时候的很多东西似乎有了更多的理所当然。那我现在写的，也许随着我的成长，未来也会觉得不尽然。

我写的是不是我都能做得很好呢？有些是，有些不是。有些是走过之后回望的总结，有些其实是当前我认可的，我坚持的，我正在练习的。

就好像黄蓉练不了降龙十八掌，郭靖练不了逍遥游，有些内容哪怕我觉得应该是对的选择，对的理论，最后可能因为性格、资质、甚至生活、境遇，发现却是走不通的路。没有练成，但也还是成长。

回到专栏这件事。选择在极客时间开启我的第一个专栏，我知道我承诺的时间和精力的投入。会有什么样的回报和订阅量，虽然说会有期望，但是却不是我决定的主要因素。

对我自己而言，专栏的每一篇文章，都是我和极客时间参与者的心血，更是我人生阶段（技术转管理阶段）的一份记录。

我把所有的感想写下来，是一种沉淀，因为时过境迁，以后的我可能写不出此时此刻这样的文字。

对我的读者而言，我之所以分享，是因为它记录的是我的成长，里面的道理，都不是什么标新立异的新道理，甚至不一定是绝对的正确；但它有我的故事，是我现阶段认为正确的，是我正在坚持的。

它记录的，不是我的顿悟，而是我的练习。希望给你带来的，也不是顿悟，而是在你成长的道路上练习的时候的一种参考，一种互勉。

最后，感谢所有订阅了这个专栏的读者和支持我的朋友。池老师说，给我修改文章的时候感觉像是和我的对话。

我平时几乎很少会和别人说这么多话，写专栏的时候，我也尝试着像跟一位朋友聊天一样去记录和书写。如果你有耐心看下来，哪怕只是其中的几篇，那就是我和你的对话。

一路很长，感谢有你！


极客时间
朱赟的
技术管理课
TECHNICAL MANAGEMENT
从技术到管理，
让你的目标函数
达到最优解
朱赟
计算机博士
Airbnb 技术经理
扫码加交流群

# 新书|《跃迁：从技术到管理的硅谷路径》

你好，我是朱赟，好久不见。

在专栏结束的第77天后，我出了一本书：《跃迁：从技术到管理的硅谷路径》。

专栏是这本书的起点，也是书中的一部分。可以说，是专栏促成了这本书的诞生。

2016年1月13日，我注册了自己的公众号“嘀嗒嘀嗒”，陆陆续续地开始写一些内容，不知不觉，产出了不少文章，但是从未对它们做过系统地整理与集合。

直至开了这个专栏后，我发现，随着自己输出内容的增多，知识开始形成了自己的体系，它们奠基、建立、增长、日渐扩充，以至于有足够的内容滋长成了这本书。

书一共有五个部分，除了专栏有的技术管理、技术实践、硅谷文化与个人成长的四个部分，还额外多了一部分杂谈的内容，除此之外，上述的每个部分还在专栏文章的基础上多了一些增益的文章内容。

这本不起眼的书，其实凝聚了很多机缘巧合和刻意努力。除了我，还有“极客时间”以及“博文视点”的编辑们，更有无数我认识或者不认识的朋友们共同努力着，为它的成型出谋划策、尽心竭力。

书籍完成后，我邀请了好朋友池建强老师，也是“极客时间”App的创立者，为这本书写了序言。

我们一起来听听，他想对这本书说的话。

---

> 2015 年冬，我受邀参加极客邦科技的技术开放日活动，在硅谷拜访 Airbnb 的时候，第一次见到了本书的作者朱赟。现在回望，那可能是本书的源点。
>
> 回国后我和朱赟在技术、产品和写作方面有过多次沟通，并邀请她做过两次机器学习的线上分享。
>
> 我在她的文字中看到了独有的灵气和特别的价值。朱赟在读博士期间受到了严格的学术训练，具备优秀的学术研究能力，毕业后在硅谷著名创业公司 Square 和 Airbnb 的经历，让她迅速把计算机科学和软件工程结合起来，获得了丰富的工程经验和强大的研发能力。

2016 年初我给她的建议是：你可以把很多类似的内容 —— 技术、成长、硅谷生活等一一分享出来，也许可以帮助到更多同样在成长的人。

于是朱赟在工作之余，开通了公众号“嘀嗒嘀嗒”，从此写作一发不可收拾。她撰写了大量的科技、硅谷文化、个人成长、求职、求学等领域的相关文章。

结合自己的心路历程和独特的经历，以轻盈细腻的文笔，鲜明别致的观点，用顶尖女工程师的视角为读者呈现了硅谷的科技生态，以及她对互联网世界的理解，迅速吸引了大量的读者。

2017年，我开始构思 IT 知识服务产品极客时间，在产品开发过程中，我邀请朱赟来开一个技术管理课的专栏。

那时候她已经告别了单枪匹马的年代，开始带一个不算小的技术团队。产品临近发布，朱赟也基本完成了从程序员到技术领导的转变。

虽然她仍然在编写大量的代码，但也在做十几人的团队管理。如何做到这一点呢？于是有了“朱赟的技术管理课”，她在专栏中讲述了自己成长过程中的技术管理、技术实践、硅谷文化和个人成长的的内容。

这本书是专栏的延续和丰富，朱赟在专栏的基础上增加了更多技术实践和管理相关的内容。

准确地说，这本书历经了五个月的打磨，朱赟在“很多个工作之余家人入睡的夜晚，静坐桌前，一个字一个字开始把自己的所思所想，通过文字记录下来”，然后交给编辑打磨润色，最终完成了本书。

我们常说写文章要留有余地，要给读者想象和发挥的空间，亢龙有悔，引而不发，这一点朱赟做得特别好。读她的文章常常会有意犹未尽的感觉，文末会引发大量有价值的思考和评论。

她认为，好的评论和交流是连接作者和读者的纽带，会给双方都带来更多的成长空间。

读朱赟的文字，你感觉不到摩擦和抵抗，或者说书中没有教条和滞重，对于一些艰深的技术和让人困惑的管理难题，她总能用轻灵的文字呈现出来，所谓举重若轻，深入浅出，莫不如是。作者回避了教科书式的讲述和教条的说明，而是如同打磨玉石一样，把璞玉变为宝石，读来无比顺畅，却又给人启发和深思。

目前朱赟依旧在硅谷追求自己的技术理想，她仍然行走在成长的路上，孜孜以求科技和创意之源。这本书则记录了她成长的一个阶段，她的身边则是和她一起成长的读者。祝愿在未来的日子里，朱

赟为这个世界带来更好的技术作品和文字。

落花无言，人淡如菊，书之岁华，其曰可读。

这本书值得所有热爱技术和互联网的人阅读，也推荐给你。

---

新书正在专栏里首发，你可以在App内的极客商城中购买书籍。4月20日首发当天优惠价49元，购买新书和专栏套装还能享受89元的惊喜价。