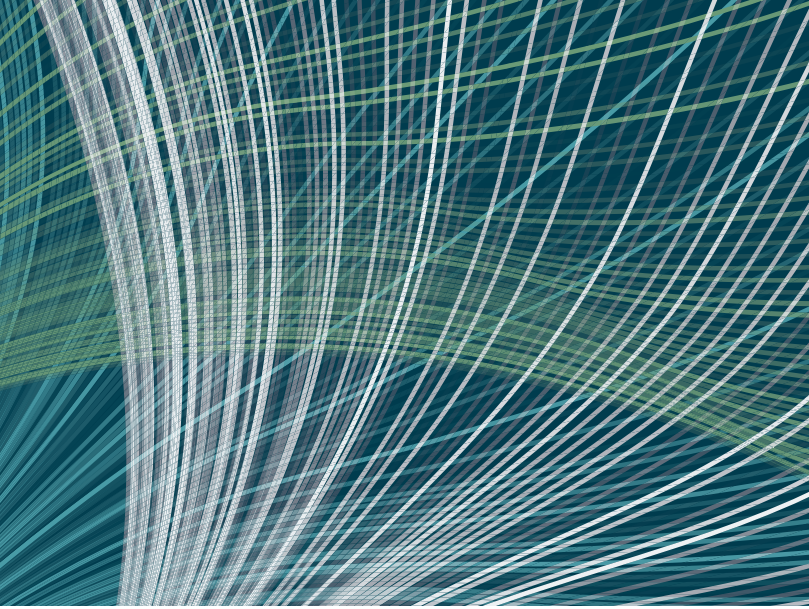


Tech Lead 之路

/thoughtworks

目录

从个人到团队

关于 Tech Lead 这个名字

Tech Lead，有各种中文对应名称。例如技术主管、技术经理、技术管理者等等。但是并没有一个词能准确且完整地诠释出 Tech Lead。

Tech Lead 是一个跨越技术和管理的角色。技术是其背景属性，但其职责又不单单是管理。带领团队冲锋陷阵，身先士卒才是 Lead 的精髓所在。主管、经理的意思却大相径庭。Tech Lead 好比战场上将帅的综合体。不但要能力过硬，可以带队冲锋陷阵。同时又要懂得调兵遣将，稳定军心，顾全大局。

从开发到 Tech Lead

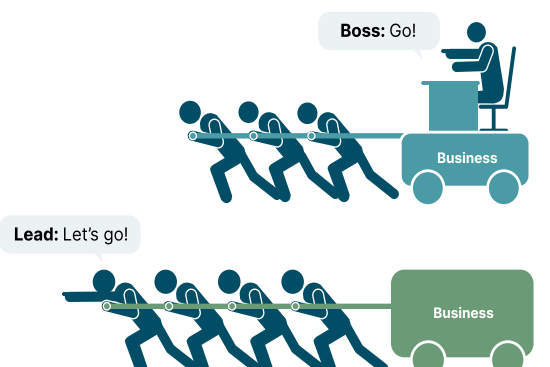
Tech Lead 的要求如此之高，但选拔 Tech Lead 的方式却有些简单。很多开发工程师都是凭借优秀的技术和积极主动的品质成为了 Tech Lead。如此选拔 Tech Lead 看起来合情合理，但细琢磨起来总觉得哪里不太对劲。类似的事情在软件行业每天都在发生。就像电影行业的 “演而优则导”。但其实好演员导出的烂片也不少。

一名 Tech Lead 新人，通常都是一腔热血投入到项目之中，准备大干一场。但很快就会发现各种状况：团队在空转、开发任务被 block、意料之外的问题、代码风格不统一、各种 bug、pipeline 飘红。总之就是团队效率低下、代码质量差。Tech Lead 恨不得高呼一声：都闪开，让我来！但就算是 10 倍程序员，也干不完 20 人的活。虽然推崇“一个队伍像一个人”，但不能真的就只有一个人。

作为一名技术工作者，在专业领域取得成绩后，逐步过渡到管理者，这合情合理。但成为 Tech Lead 后，跨越技术到管理、个人到团队的沟壑并不容易，需要艰难的转变过程。对 Tech Lead 身份的充分认知，则是转变的起点。

Tech Lead 不是 Manager

Tech Lead 需要承担一定的管理职责，但其并不是 Manager（经理），所以用技术经理或者主管来对应 Tech Lead 并不准确。Tech Lead 是 Leader。关于 Leader 和 Manager 区别的解读有很多。不过在 Tech Lead 背景下，二者有很多相似之处。但其最核心区别是，Leader 深处团队之中，带领团队一起冲锋陷阵。Manager 坐镇后方，指挥团队。



（图片引自网络）

Leader 是团队的领头羊，冲锋号。对团队的动作多为引领和指导。而 Manager 是发令施号的人，他一般不需要亲自带团队冲锋陷阵。例如上线时，Tech Lead 一定会在现场，带领团队完成一个个任务，碰到难题也会和团队一起想办法解决。Manager 可能也会出现在现场，但大多数时候只是稳定军心，关心一下进度。

Tech Lead 带领团队冲锋陷阵，同时又要肩负各种管理职责。并不是某一种我们熟悉的角色就可以完整诠释 Tech Lead 的职责。想要成为合格的 Tech Lead，首要任务是对 Tech Lead 有正确认知。

Tech Lead 角色认知

晋升为团队的技术管理者，意味着升值加薪、更多股票！但更重要地，意味着你的角色发生了变化。没错，你过去的工作方式和经验已经不好用了。过去你只需要做好自己的开发。而现在你需要带领整个团队出色地完成开发工作。是时候学习一些管理知识了，然后和你的专业知识结合起来。现在你不只是开发，还是架构师，甚至项目经理。你是这些角色的综合体，但又不可能面面俱到。有限的时间

和经验，应该把技能点点在哪里？应该做什么工作？这需要你对 Tech Lead 角色有正确的认知。

从个体到团队

作为开发可能最快乐的事情就是一个人写上几个小时的代码，沉浸在造物主的乐趣之中。我不希望，也不想和其他人有过多交流。我只关心我的程序是不是跑通了，是不是写得足够 “工匠精神”。

但是当你成为 Tech Lead 的那一天，你会和这种快乐渐行渐远。

- 过去你只关心自己程序的设计，现在你要关心系统整体设计。
- 过去你只关心自己的任务拆分，现在你要关心整个团队的任务拆分。
- 过去你只关心自己任务的依赖和优先级，现在你要关心团队的任务怎么安排更合理。
- 过去你只关心自己的代码写的是否够好，现在你要关心整个团队的代码质量。

- 过去你只关心自己如何学习技术，现在你要关心团队的技术能力成长、每个成员的个人发展。
- 过去你可以流畅讲出自己的程序实现，现在团队开发的任何内容你都要做到应答如流。

嗯？怎么看起来事情还是那些事情，只是从个人到团队了？没错，从个人到团队，意味着你要身兼项目经理、架构师、高级工程师的工作。

Tech Lead 是项目经理

从开发到 Tech Lead 的最大转变是开始承担团队管理工作。你要开始从团队管理的视角去思考问题。

- 如何提升团队的开发效率
- 每个成员如何成长
- 技术方案是否全局最优
- 现在项目进度如何
- 项目是否存在风险

诸如以上这些问题，如果成为 Tech Lead 后你还没有开始

思考，那么你的团队将十分危险。很多 Tech Lead 新人，很容易陷入一个具体问题的解决上，从而忽略团队，对项目缺少规划。这会导致团队效率低下，问题频发。最后的结果就是哪里漏水堵哪里，Tech Lead 一人堵几个最难堵的洞，剩下的洞大家一块堵。整个团队疲惫不堪，项目却做得非常糟糕。

我认为 Tech Lead 除技术之外，最重要的能力就是规划能力。要想做好规划，需要清楚知道团队的能力，细化工作任务，识别出项目的潜在风险。一句话，项目的一切尽在掌握。

Tech Lead 是架构师、高级研发

Tech Lead 的技术一定要过硬，这是成为 Tech Lead 的基本要求。可能你的某项技术并不是团队中最好的，但你一定要具备一定的技术广度。此外最重要的是快速学习的能力，能够短时间掌握新技术、解决问题。

当团队碰到技术难题，Tech Lead 肯定是冲在前面，带领团队一起攻克难关的人。注意，是带领团队，而不是 Tech

Lead 一个人解决难题。Tech Lead 需要合理利用团队能力，找到合适的人、合适的方法来解决问题。如果 Tech Lead 总是一个人钻研技术难题，这并不是正常现象。

Tech Lead 在技术层面需要关注如下三件事：

1. 为团队建立技术标准，并带团队遵守标准
2. 关注架构和方案设计，自己参与或者审查团队成员的设计
3. 带领团队解决技术难题

Tech Lead 是沟通达人

作为 Tech Lead，沟通是非常重要的技能。你不能再一个人沉浸在代码世界里了。

请你和每一位开发沟通！和项目经理沟通！和客户沟通！

一开始会有些胆怯，甚至都表达不清楚。这非常正常，沟通是每一位工程师相对欠缺的技能。没办法，谁让咱前几年把技能点都点在了技术上。沟通也有一定方法，掌握方法的同时，大胆去沟通，不怕犯错。只有不断实践，才能

跨过沟通的障碍。

Tech Lead 还是那个技术极客

Tech Lead，根基在 Tech 上。除了团队的工作之外，一定要给自己留有学习技术、编写代码的时间。

成为 Tech Lead 后，你会发现管理的事情永远做不完。这是因为管理的工作成果不如编码具像化。你会觉得自己再多思考一会，就会有更好的结果。

此种情况下，Tech Lead 如果不给自己规划好钻研技术的时间，会使自己很久不碰代码、不学习新技术。当很多 Tech Lead 意识到这个问题的时候，自己已经远离技术很久。

你虽然成为了 Tech Lead，但你骨子里还应该是那个技术极客。请继续保持对技术的热爱，不要远离代码。只有根扎的深，树才长得高，结的果实也会更多！

帮助团队成长是唯一出路

帮助团队成长是唯一出路

相比于技术路线，管理技巧难以被量化。更严重的问题在于，对于 Tech Lead 自己而言，似乎也没有动力去给领导力做进一步提升，因为一方面大部分公司对于对于团队以及 Tech Lead 的绩效考核，永远是以业务指标为导向的；另一方在公司框架的束缚和工作节奏的惯性带动下，团队工作表现也不太可能出现大的波动，似乎只要不犯错就是万事大吉。退一步说，没有胡萝卜仅仅用大棒来鞭策团队也是可以接受的，只要向上管理得当，没有人会关心你的管理工作是如何执行的。

如果不犯错是 Tech Lead 的及格线，我们还能为管理技巧加多少分？

Tech Lead 向左，Manager 向右

长时间以来，我们一直被鼓励要成为 Tech Lead 而非 manager， 关于两者差异最经典的解释是：manager 游离于团队之外发号司令，Tech Lead 身先士卒。

然而这样一句话远远不够，它没有来龙去脉：我们为什么要身先士卒而不是发号司令，两种行为产生的效应又有何不同，以及才能达到预期 Tech Lead 带来的效应。我不如再直接一些，抛开各种各样的理论，Tech Lead 究竟应该做些什么？

在 《Lean Software Development: An Agile Toolkit》一书中，作者给出了以下有关 manager 和 Tech Lead 之间的差异

Managers (cope with complexity)	**Tech Leads** (cope with change)
Plan and budget	Set direction
Organize and staff	Align people
Track and control	Enable motivation

但不同角色之间并非是完全割裂的，例如 Tech Lead 也要关注交付计划的执行状况，manager 最好也能和团队成员对齐愿景。

带着这些问题，我在《哈佛商业评论》Leadership 标签下的很多文章里搜寻答案。我认为最完整的回答《What Leaders Really Do》中已经说的很清楚了：

Management is about coping with complexity; it brings order and predictability to a situation. But that's no longer enough—to succeed, companies must be able to adapt to change. **Leadership, then, is about learning how to cope with rapid change.**

- Management involves planning and budgeting.

Leadership involves setting direction.

- Management involves organizing and staffing. **Leadership involves aligning people.**
- Management provides control and solves problems. **Leadership provides motivation.**

Management and leadership both involve deciding what needs to be done, creating networks of people to accomplish the agenda, and ensuring that the work actually gets done.

其中的很多观点在它其他很多篇的内容中也有重复，比如在《Becoming a More Humane Leader》中：

Management is about managing others, about exercising executive control over people. **Leadership, on the other hand, is about seeing and hearing others, setting a direction, and then letting go of controlling what happens next.**

在《The Leader as Coach》中

companies are moving away from traditional command-and-control practices and toward something very different: a model in which managers give support and guidance rather than instructions... **The role of the manager, in short, is becoming that of a coach.**

不难发现 Leader 总是与 vision、people、direction 这些词汇关联在一起。答案也就呼之欲出了： Leader 纵然有管理的特质，但是他无法一意孤行，你必须要不断和团队在方向上达成一致，并肩前行。

帮助团队成长的理由

我不认可让团队成员放飞自我的工作方式，这类工作方式在《打造 Facebook》或者《奈飞文化手册》书中看似美好的一个隐形前提是，公司愿意肯花费巨资去寻找最优秀的人。而我相信大部分公司的现状是，想找到一个靠谱能出活的人都难上加难。

我曾经的一位 Tech Lead 送给过我一本他颇为受用的图书《一分钟经理人》，书中作者一语点破了这种现状：

作为经理人，你实际上有三种选择。第一，你雇佣成功者，这些人很难找，而且开价很高；或者，第二，找不到成功者，你还可以雇佣有成功潜力的人，通过系统的训练把他们培养为成功者；要是前两种选择你都不满……那你只剩最后一种选择了——祈求老天保佑。

看上去我们并无选择是不是？只有培养他们。

当然这里有一个前提是，公司认可人才带来的价值的，并且有意愿去招募好的人才。大部分公司并不会买这个帐，因为他们用平庸的人打造出的平庸产品就能够占领足够的市场获取丰厚利润了。

另一个自私的理由是，尽可能去培养与你旗鼓相当的人，还可能是对 TL 自己的解脱。

我曾经纠结了相当长时间的一个问题是：我应不应该为团队成员分担更多的代码？毕竟我能写得更快更好。但在和某位前司大佬促膝长谈之后，他用下列理由明确告诉我不要这么做：

1. 我之所以能写得更快和更好，恰恰是因为我已经驾

轻就熟了，付出再多的时间对我个人来说不会有更多的收益。

2. 然而同样的工作对于尚未接触这块知识点的同学来完成带来的收获可能说受益匪浅，所以交由他们完成更加适合。
3. 应当从执行者的视角转向管理者的视角转变，解决预见性的问题，解决非常规问题。

在另一本书《The Coaching Habit: Say Less, Ask More & Change the Way You Lead Forever》中开篇就提出了类似的观点：通过养成培养（coach） 他人的习惯，你能够轻松打破工作场合中关于你自己的三类恶性循环：

- 对于你个人的过度依赖：当你成为全组人员的依赖之后，你的工作量会剧增，同时也成为系统中的瓶颈，其他人自然也就失去了工作的热情和动力。
- 忙到停不下来：无论你掌握了多么高超的高效工作技巧，你解决的问题越多，有待你解决的问题也就越多，在你被各类不同优先级的工作缠身之后，你便会失去工作的重心。
- 与最重要的事情脱节：我们的目标不应该仅仅是完

> 成工作，更是要带来影响力和赋予工作意义。培养这件事能够帮助他人走出舒适区，向卓越迈进。

跳过私人理由，即使从团队健康发展的角度上说，团队内也不应该有 hero 的存在，或者你听过更常见的说法是“摇滚明星程序员”（Rockstar Programmer）。这类人群在团队中无所不能，专攻疑难杂症，救项目于水火之中，但很快你就会开始发现他会成为瓶颈（single point of failure），越来越多的环节缺少他会开展不下去。这种对他的依赖同样也会反噬他自己,他的负担会变得越来越重，导致他最终燃烧殆尽（burn out）。

最后，如果你是痴迷于“掌控感”，那我可以很明确地告诉你 membership 和 ownship 对于员工也同样重要，他们工作中的幸福感也来源于掌控感。

营造氛围

去帮助我的团队除了给他们带来成长以外，还有一项任务可能是你想不到的：为他们营造无压力的氛围。

我最近在东东枪老师的播客“宇宙牌电饭锅”一期有关上

班的节目里，听到一则让我非常有共鸣的观点，大意是工作会给人带来一种无力感。无力感不是有意谁强加给谁的，可能是从上到下，从外到内传导的结果。

我想表达的是作为 Tech Lead，你不仅有能力将 suffering 在你这一层就阻断掉，还有义务这么做。道理非常简单：suffering 的环境只可能给工作环境带来负面效应。对于团队内产生的问题，拍桌、呵斥、咒骂都无济于事。 如果你相信新自由主义自我管理的那一套的话，在《第五项修炼》或者《Question Behind the Question》书中永远在告诫你，永远从自己找问题，永远去想如何解决问题。

我说服不了自己如果团队成员能够少关心繁琐的事情，少参与一些政治斗争，哪怕花很少的成本就能让他们工作得开心一点，那为什么不呢?

知乎上一个沉重问题下的答案更深刻地解释了这个道理，这一小节其实引用他的话就足够了：

不是说因为经理在 suffering 就能合理化手下的人在 suffering。做经理的人知道这是 suffering 的工作，他们可以选择不做经理。好的经理往往都要有一定自我牺牲的

意识，帮手下把屎挡掉。一个好的经理就是应该尽力把 suffering 的事情终结在自己这个层级，不要再往下扩散，只是有时候这不现实，往下扩散无法避免。

撕开公司柔情脉脉的面纱，对团队以及 Tech Lead 的评价始终是要建立在结果为导向上的。人毫无疑问是对交付来说最基本和最有效的保障。“以人为本”喊了那么多年听到我们耳朵都起茧了，但不得不承认它依然是有效的。

在我的文章《去年做 Tech Leader 犯过最大的错》的最后我留下了一个问题：

皮克斯公司在《玩具总动员》大获成功之后，他们安排《玩具总动员》一片的创意团队专心致力于《虫虫危机》的制作，而只派遣了两名（导演经验为零的）资深动画师来掌舵《玩具总动员 2》。

而等到一年之后再来审视《玩具总动员 2》的进展时，却发现整部动画惨不忍睹。于是他们不得不做出换掉导演团队的艰难决定。

但冒险也是皮克斯不可以扔掉的一个特质，这就意味着他

们必须把钥匙交到那些不太符合传统导演标准的新人手中。那么如何避免上上面的情况再次出现?

他们开始有意识的培养新人:

为那些我们认为具备导演潜质的人才提供培训和历练的机会。我们不再奢望新人会通过潜移默化自然而然地吸收资深导演们的智慧，而是设置了一套正规的培训课程……此后,每位资深导演每周都要和自己所培训的新人进行交流，对新人的电影构思给予激励并提出建议。

技术视角

Tech Lead 的 "Tech" 指明了 Tech Lead 日常工作的核心内容和管理重心。那么作为 Tech Lead，究竟需要关心哪些 Tech 相关的内容呢？又具体需要做哪些事情呢？

指导技术解决方案

首要的当属对技术解决方案的把控和指导，这个工作感觉很像一个架构师的职责。没错，当团队在各种场景下需要给出解决方案时，技术管理者就要戴上架构师的帽子，设计架构，做技术决策。

这里可能有很多种情况，跟项目的具体情况，以及 Tech Lead 接手时项目所处阶段有关，比如：

1. 全新项目开始前：全新的业务需求，架构设计和解决方案尚未确定。
2. 项目初始阶段: 项目已经有了解决方案的大致方向，包括技术选型、系统职责、集成关系等。
3. 项目中后期：可能已经是一个遗留系统，在设计解决方案时可能有各种约束，或者坑。

这三个阶段对于 Tech Lead 的挑战是逐渐递增的，全新的项目在这里是相对简单的场景，澄清业务需求以及各参与部门或系统之间的职责和关系，结合业务及技术目标设计架构和解决方案，Tech Lead 有足够的灵活度来处理各种需求带来的问题。

而对于已经有了方案基础，或者开发了一段时间的系统（也可能是年久失修的遗留系统），在做解决方案设计时需要考虑很多内容：

- 既有架构设计的问题：特别是针对遗留系统，有哪

些问题，严重程度如何，是否有规划中的方案等。

- 既有架构设计初衷以及业务目标是否还成立，是否有新增的需求，是否能够满足未来业务演进的需求。
- 当前是否有更优的架构设计方案，是否需要做技术架构的迁移。

相较于全新的项目，对已有系统设计解决方案之前，要充分了解系统中的技术债，结合架构设计的长远目标，可用资源以及开发周期，给出相对合理的方案。通常一个完整的技术解决方案会包括下面的内容：

1. 应用架构设计
2. 部署架构设计
3. 数据架构设计
4. 技术栈选型
5. 集成方案设计
6. 专项解决方案设计
7. 跨功能需求

上面的 1~4 是系统的整体架构设计，在项目之初就需要

设计好，包括相应的架构设计原则。随着开发过程的进行，一定会对架构整体设计进行调整，推荐使用 ADR (Architecture Decision Records) 记录架构调整的决策。

更值得一提的是集成方案设计和专项解决方案设计，这两项在日常开发中的占比更大，也更容易出问题：

集成方案涉及第三方系统，限制或风险有很多，比如：

- 因第三方系统的接口技术比较落后而需要做大量的适配。
- 因第三方系统的开发周期很长，要考虑 Toggle 等灵活应对方案。
- 因双方理解不一致而导致联调失败，无法上线。

专项解决方案是针对特定需求而定制的解决方案，通常也是 Tech Lead 接手项目后为解决当前系统的一些问题而设计的解决方案，也是当前项目架构设计的一些特定关注点，比如针对历史数据的迁移方案，针对当前业务的分库分表方案，针对当前项目业务数据量定制的报表设计方案等。

最后是 跨功能需求 (Cross-Functional Requirements,

CFR)，永远不要忘记跨功能需求，跨功能需求帮忙定义了解决方案的更多细节，并规避一些安全、基础设施等风险。

更多跨功能需求管理的内容参见后续章节: 跨功能需求管理。

统一团队方向

除了架构师的职责，Tech Lead 的另一个重要的职责就像黑夜中海上的“灯塔”，把团队引领到一个统一的方向上。当团队成员产生分歧时，能够协调团队成员达成一致，并守护大家达成一致的结果。

通过一个小例子来看看团队成员意见不一致产生的问题：

> 在团队中有 2 个非常有经验的开发同学，两个同学各有所长，一个同学追求函数式编程，另一个同学更崇尚面向对象编程，并且在各自的方向都有比较深入的研究和见解。两个同学写出的代码风格完全不同，各有千秋，在某些问题上还会争个面红耳赤，逐渐演变成了两种设计风格之争，各有立场，甚至不愿意去碰对方写过的代码。

上面的场景相信在开发中并不少见，好学的同学总是能引入一些新的想法或技术到项目中，从某种角度来看这是我们推荐的，但确实会对项目既有的一些设计原则和实践带来冲击，如果放着不管，就会逐渐形成上面的情况，团队内的分歧会越来越多，甚至对团队共同承担的责任都有分歧，你的代码你改，你的设计你负责，边界感越来越强。

破坏规则很容易，但守护规则需要所有人一起努力。Tech Lead 需要有敏锐的观察力，及时发现项目中的不一致，引导大家找到问题的本质原因，建立统一的方向，并带领团队守护方向，涉及的内容包括：

- 架构设计原则
- 代码设计原则
- 团队内约定的流程和实践
- 问题处理的流程和原则

管理技术风险

再优秀的团队也无法完全规避风险，从专业的视角来看，只有足够的技术背景才能够感知技术风险，并准确评估技术风险带来的影响，Tech Lead 责无旁贷要承担这个职责。

Tech Lead 要带领团队一起识别风险，排列优先级并逐个消除风险。类似技术债管理，可以通过有效的手段可视化技术风险，帮助团队快速理解问题并引起重视，持续跟踪风险的状态，保证风险尽快得到解决。

更多技术风险管理的内容参见后续章节：风险管理

演进的视角

Tech Lead 的 "Tech" 看起来和技术更相关，但却不仅仅是技术，也是权衡的艺术。

一个团队中的决策大大小小有很多，有些是大家一起讨论做出决策，有些是 Tech Lead 依据经验进行决策，也有很多是开发人员依据经验、知识和自己的技术观来决定。大到如何设计一个解决方案，小到如何写一段代码，这些决策都是权衡的结果，是决策者基于自己认知的权衡。

作为 Tech Lead，肩负着指导项目技术解决方案和统一团队方向的重任，看问题的视角需要打开。在决策时，需要考虑决策产生的影响，比如未来是否需要返工，是否会带来运维工作量的增加，是否会对其他团队产生不良影响等

等。更需要站在更长远的视角，考虑软件架构演进的方向，考虑系统未来需要承载的业务增长量等等，也要有意识培养整个团队站在更长远的视角看问题，避免引入技术债和技术风险。

跨功能需求管理

什么是跨功能需求

在成为 Tech Lead 的初始阶段，大部分同学很容易忽视业务需求之外的一些需求，我们称之为跨功能需求 (Cross-Functional Requirements, CFR)，有时也常被称为非功能需求 (Non-Functional Requirements，NFR)。为了直观感受下什么是跨功能需求，我们来看几个交付过程中常见的场景：

> **场景一：**在一个需求上线后，线上数据量明显超出想象，上线后仅仅半年，数据库存储就无法支撑新增的业务数据，性能下降，甚至可预见在未来一个月内，数据库会因为存储满了而无法提供服务。
>
> **场景二：**需求开发上线后工作正常，但在一次市场活动中，业务迎来一个峰值，系统因无法处理高峰期的大量并发请求而宕机。

在这两个场景中，需求上线时业务都是可以正常工作的，但随着时间推移或使用场景发生变化，系统无法正常工作，表现为不可用或性能差。软件产品是一个复杂的系统，承载着帮助产品使用者完成复杂的日常工作，提升效率的使命，我们希望它能够在业务演进和架构演进的过程中，保持稳定性和可用性，做高效的助手，而非找麻烦的捣蛋鬼。

正如跨功能需求这个名字中定义的，这些需求是跨越了我们构建的所有功能需求，不只是关注在某一个业务模块，而更关注在多个业务功能协同工作时，为保证业务正常运

转，软件系统需要满足的一些特性。这些跨功能需求通常表达为“XX 性”，如安全性、可靠性、兼容性、可扩展性等。

关注跨功能需求

从上面两个场景案例可以看出，问题一旦发生，都会是整个交付团队第一优先级需要处理的问题。跨功能需求和功能需求同样重要，不容忽视。

当问题解决后，回顾和反思为什么没有人第一时间考虑这些需求时，或许会将矛头指向需求方：

> “为什么没有同时提出这样的跨功能需求？”
> “为什么没有提前说会有市场活动？”

而得到的答案往往也比较直接：

> “我们只管提业务需求，怎么实现是你们的事情。”
> “为什么你们在设计技术方案时没有考虑业务量会增长的情况？”

不管是功能需求还是跨功能需求，需求的澄清和确定都不是某一个角色可以独立完成的，而需要所有参与方共同协作，根据软件系统的现状和未来发展目标，讨论并达成一致。

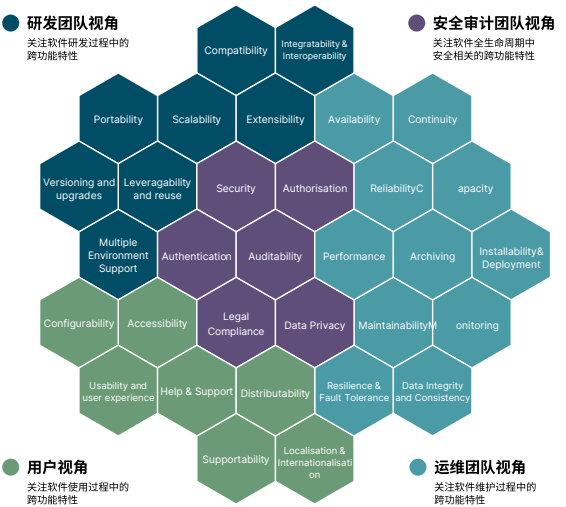
作为 Tech Lead，在需求讨论和解决方案设计阶段，要从技术侧提出相关问题，帮助团队在进入开发前澄清和确定跨功能需求。这些问题通常需要关注以下几点：

- 软件产品的业务目标
- 产品利益相关者的长期愿景
- 公司内对技术合规的约束和要求

常见的跨功能需求有哪些？

跨功能需求影响着软件的整个生命周期，在项目交付过程中，可以根据软件产品的目标和特点，从以下几个视角来收集和确定跨功能需求：

- 研发团队视角，关注软件研发过程中的跨功能特性，包括软件架构设计相关的一些特性，如可扩展性、可移植性、可伸缩性、兼容性等。



- 用户视角，关注软件使用过程中的跨功能特性，关注用户体验，如设备兼容性、可访问性、可配置性等。
- 运维团队视角，关注软件维护过程中的跨功能特性，包括基础设施运营维护、数据维护、故障恢复相关的一些特性，如性能、可用性、容量、监控、熔断降级策略等。
- 安全审计团队视角，关注软件全生命周期的安全相关的跨功能特性，大部分企业有专门的安全审计部

门，会对软件产品的安全提出很多需求，如可审计性，法律合规性，数据隐私性。

以上的分类视角可以作为参考，帮助我们快速识别当前产品在哪些方面可能存在跨功能需求，但分类没有绝对的边界，在不同的软件产品研发过程中，一个跨功能需求可能影响多个参与方。

常见的跨功能需求及描述参见下表：

CFR	描述
可扩展性 Extensibility	是否需要组件化？ 是否需要提供一个插件功能，由谁来实施？
可移植性 Portability	是否有迁移到另一个数据库产品或操作系统的必要性？
可安装性和可部署性 Installability & Deployment	需要提供什么样的基础设施？ 需要什么样的安装便利性？ 需要支持持续交付吗？ 如何回滚或升级版本？
兼容性 Compatibility	需要与哪些其他系统集成？ 需要遵循哪些其他行业标准？ 是否需要考虑现有数据格式？

CFR	描述
可集成性和互操作性 Integratability & Interoperability	是否需要提供 API 或库供其他系统使用？版本管理和升级策略是什么？
可复用性 Leveragability & Reuse	是否能够复用企业现有组件 / 库，或者当前组件 / 库是否将被重用？
可伸缩性 Scalability	如何根据不断变化的用户量来提高吞吐量？ 如何对此进行测试？
版本化和升级策略 Versioning and upgrades	版本化策略是什么？ 如何跟踪内部 / 外部的版本？ 有没有向后兼容的限制？
可访问性 Accessibility	支持有特殊需要的用户（如读屏）
本地化和国际化 Localisation & Internationalisation	对静态 / 操作数据是否有多语言支持？ 日期 / 时间 / 货币？ 转换和翻译？

CFR	描述
可用性和用户体验 Usability and user experience	用户体验对这个系统有多重要，在这个过程中会如何运作? 是否有公司的用户体验准则要遵循? 我们是否需要考虑不同的设备 / 屏幕尺寸?
分布性 Distributability	人们是否能够在特定区域使用该系统，而不会因为地点或距离而受到阻碍? 它能在没有互联网接入的情况下运行吗? 如何同步信息?
帮助与支持 Help & Support	需要什么程度的用户文档? 是否需要建立一个教程，提供视频或提供一个帮助和支持团队? 是否需要为培训做计划?
可配置性 Configurability	用户或管理员可以配置功能吗? 如何进行配置管理（如文件、用户界面、API 等）?
支持性 Supportability	确定用户 / 操作支持的级别，以及将如何支持它们?

CFR	描述
归档 Archiving	归档什么信息？ 何时归档？ 如何归档？ 谁 / 如何访问这些归档信息？
可用性 Availability	是否有可用性目标要求? 需要什么架构来满足这些要求? 是否有特定的日子（如黑色星期五）需要这些? CAP（一致性、可用性、分区容忍度）。
容量 Capacity	是否有任何特定的存储要求? 是否有高峰负荷的考虑? 需要由系统处理的数据量是多少? 有多少用户将同时使用该系统?
连续性 Continuity	是否考虑灾难恢复计划?
数据完整性和一致性 Data Integrity and Consistency	是否需要数据校验、日志追踪或提供数据恢复机制?

CFR	描述
可维护性 Maintainability	最大的可容忍停机时间（RTO/ 恢复时间目标）是什么？ 是否有预定停机时间的通知要求？ 错误页面如何处理？
监控 Monitoring	是否有现有的工具和基础设施？ 应该衡量哪些业务 / 技术指标？ 如何进行监测？ 哪些类型的警报是必要的？
多环境支持 Multiple Environment Support	理想情况下需要多少 / 哪些环境？ 如何配置和管理这些环境？
性能 Performance	吞吐量 / 响应时间要求是什么？ 需要有计划地进行性能测试吗？ 是否需要考虑异步场景？
弹性和容错性 Resilience & Fault Tolerance	如果某些外部依赖失效 / 不可用，系统将如何降级？
可靠性 Reliability	不可靠的成本是什么？ 需要多少成本来保证可靠？
可审计性 Auditability	哪些操作应该被跟踪？ 必须满足哪些法律或监管要求？

CFR	描述
认证 Authentication	如何鉴别用户身份? 应该遵循什么标准或使用哪些现有的认证系统?
授权 Authorisation	哪些角色和权限是必要的，他们需要访问哪些功能或数据? 谁来维护权限，如何维护和应用权限?
法律合规性 Legal Compliance	对数据 / 系统或软件交付过程是否有法律限制? 对发布过程是否有任何限制?
数据隐私 Data Privacy	哪些数据应该被加密? 哪些数据对终端用户和操作人员可见 / 隐藏? 开发 / 测试过程中能否使用生产数据? 或者如何做生产数据脱敏处理?
安全性 Security	需要为安全审计或渗透测试制定哪些流程? 企业的安全准则是什么? 是否有 SSL 或 VPN 要求?

小结

跨功能需求涉及了软件产品的各个方面，但这些需求并不会随着业务需求一起提出。作为团队里的 Tech Lead，需要根据项目的情况，尽早地识别哪些跨功能需求在当前项目中需要特别关注。对于某些需要长期持续关注的跨功能需求，可以转化为解决方案设计中的一些规范，而针对其他非固定的关键跨功能需求可以设计一个 checklist，在每个新需求到来时进行检查。

在实现业务目标时，有些跨功能需求不是必需的，因此在预算不足的情况下，往往会被踢出去，而变成技术债。Tech Lead 需要权衡对于一些关键跨功能需求在当前实现和延后实现的收益和成本，做出相对合理的决策，也要时刻警惕因跨功能需求处理不当而引入的风险。

风险管理

什么是风险?

说到风险这个词，相信大家脑子里已经有了很多画面，比如:

- 开发过程中没有写单元测试
- 数据库没有备份
- 下个月有个关键角色要离开团队
- ……

很明显，这些场景都会给项目的开发和交付带来不良影响，这些都是风险吗？

先来看看风险的定义：

> 风险是指在某一特定环境下，在某一特定时间段内，某种损失发生的可能性。
>
> —— MBA 智库 • 百科

> 风险是相对某有机体的，指某可能发生的事件（辞源于航海者），如果发生，能阻碍有机体的发展，甚至走向衰亡，风险是指事件发生与否的不确定性。
>
> ——维基百科

风险有两个关键要素：

- 风险是可能发生的事件，具有不确定性
- 风险一旦发生，会产生损失或不良影响

回过头来再看看开头的几个场景，虽然都会产生不良的影响，但都是确定的事件，这些并不是风险（Risk），而是需要马上处理的问题（Issue）。作为 Tech Lead，要能够区分风险和问题，风险是未来有概率发生的，需要提前干预管理，而问题是现在已经或正在发生的，需要马上解决。

Tech Lead 为什么要关注风险？

既然风险并不是现在已知的问题，未来也不一定会发生，那为什么还要关注风险呢？有必要为一些概率性事件投入大量精力吗？

我们首先从统计学角度分析下风险发生的概率：

> 假设有 10 个风险，每个风险有 10% 的概率会发生，那么有一个风险实际发生的概率是：1 - 0.90^10 ≈ 65%。

也就是说，在这 10 个风险中，至少有一个会发生的概率超过了 50%。如果这些风险的后果是很严重的，项目无法

承受的，那无疑是一种灾难。显然，我们必须要采取措施，提前想好应对策略，防患于未然。

然而，风险管理是项目管理者一定会关注的，作为 Tech Lead 还需要关注风险吗？这不是重复的工作吗？如果两个角色都来管，工作职责边界好像也不清晰，不会带来问题吗？

讲一个真实发生的小故事：

> 在一个软件系统集成方案中，集成双方根据各自的需求一起讨论并确定了集成接口的契约，集成方式，调用频率以及性能要求，以及一些异常场景的处理方案。然而，集成双方并不了解对方的业务流程和数据流，很多问题都要等到集成联调测试或上线后才发现。当前项目的 Tech Lead 在跟对方闲聊技术时，无意中发现了对方的数据状态变更节点和自己这边的数据状态变更节点不完全一致，上线后一旦遇到某些特殊场景，就会 block 当前业务流程。于是发起了新一轮的方案讨论。

在跨系统集成的场景，不能想当然地认为对方系统是怎样工作的，更不能忽略未来上线的风险。

不同角色专业能力不同，能识别到的风险是不一样的，在软件开发项目中，Tech Lead 在识别技术相关风险方面天然具有优势，更是责无旁贷。

> By failing to prepare, you are preparing to fail.
>
> — Benjamin Franklin

有什么类型的风险?

从软件开发项目的各关键要素来看，交付过程中可能出现的风险大致可以分为下面几类：

设计相关的风险

说起项目交付风险，很多人第一个想到的就是需求不明确带来的风险：一句话需求，需求变更，需求冲突等。虽然需求更多是业务分析师的工作，但 Tech Lead 可以从技术的视角帮忙评估需求相关风险带来的影响，需求之间是否存在技术方案上的冲突，Tech Lead 同样要对不明确的需

求保持警惕，要有敏锐的判断力。

除了关注业务方案设计，作为 Tech Lead，更要主导和把关技术方案设计，降低未来的风险。然而，实际情况下的方案设计都是一种权衡，是对成本，对交付时间，对技术选择的权衡，并不存在绝对完美的方案，Tech Lead 需要在技术方案设计时考虑到以下风险：

- 新技术引入带来的风险
- 新系统集成风险
- 上线失败回退的风险
- 安全风险，包括开源软件相关安全风险，数据安全等

运行时相关的风险

有些项目除了关注开发阶段，同时也要负责后期的运维工作，对软件系统运行时的风险更要格外关注。如果说软件开发过程是在规定的时间内生产出合格的产品，那后期运维过程是让用户实际使用产品的过程，产品价值更多体现在后面，任何可能导致软件产品无法使用的风险都要重点关注。

运行时的软件系统为了配合业务运营需求，实际的需求和使用情况也是在动态变化的，对运行时状态的观察要考虑前瞻性，可能存在的风险有：

- 基础设施（包括 VM、网络、存储、DB 等）意外宕机
- 随业务增长导致的基础设施资源不足
- 数据泄露
- 集成系统异常

开发过程相关的风险

没有规矩不成方圆，即使在自治的敏捷开发团队中，也有一些约定俗成的规定和流程，这是开发团队能够持续保持活力和战力的基础。作为 Tech Lead，需要结合软件系统的特点，以及团队的特点，定制化调整团队的流程实践，避免缺少关键流程导致的风险。这些风险可能是：

- 进度风险
- 质量风险

团队相关风险

软件开发是一种创造性的工作，对人的依赖性很高，甚至在长期的项目上，因为项目的上下文不易快速获得，对一些关键角色的依赖性更强，这种现象无法避免。因此，在软件交付过程中，人员的更替和流动一定会给团队带来不同程度的风险：

- 人员能力不足带来的风险
- 人员流动带来的风险

如何应对风险?

那么有什么手段可以帮助我们降低风险呢?

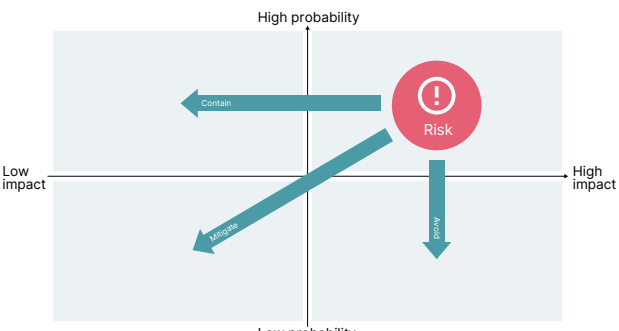
上文中讲了风险的两个关键要素：具有不确定性，会产生不良影响。因此，要降低风险，就要从这两个方面入手，要么想办法降低风险发生的概率，要么降低风险带来的影响。

上图中给出了降低发生概率和影响的几种手段：

1. **避开风险（Avoid）：**对于可能产生风险的活动，尽

量不做，以最大可能降低风险发生的概率，避免风险发生。

2. **接纳风险（Contain）：**通过计划额外的成本（时间或资源）应对可能产生的风险，在风险发生时及时降低风险产生的影响。
3. **缓和风险（Mitigate）：**分析风险，并立刻采取干预手段，减小风险发生的概率和影响。



上面几种手段看似直接简单，仔细想想，是需要非常多的经验积累的，作为 Tech Lead 要不断从过往的经历中总结每一次的事故，提炼经验。在日常交付过程中，对风险进行评估，排优先级，根据不同的风险采用不同的手段。

当然在一些特殊的情况下，我们还有另一个应对风险的选择：

4. **忽视风险（Evade）**，如果风险发生，再想办法应对，如果风险不发生，那就赚到了，这只适用于那些即便发生，影响在可接受范围的场景。

最后

风险管理从来都不是 Tech Lead 一个人的事情，应该是全团队共同承担的责任，Tech Lead 在整个软件开发过程中，也要调动全体团队成员在应对风险上的积极性，通过组织一些风险相关的头脑风暴，可视化技术债和风险，将风险管理作为全团队成员共同关注的话题。

发布之路

在各种场合，经常听到新晋的 Tech Lead 抱怨大部分时间都淹没在各种事情的海洋里，每天都很忙碌，需要参加很多会议，有各种问题需要处理，早晨的日历上还有很多时间可以安排，转眼间就被一堆突发的事情占满了：

> CI 流水线挂了，需要有人去修复。
> 技术方案漏了个边界场景，产品无法上线。
> 生产环境的基础设施有问题，没法部署新的代码。
> 积压了好多卡等待测试，测试资源不足，按时上线有风险。
> ……

作为 Tech Lead，总感觉一切都不太受控制，每天都做着似乎跟技术无关的事情，逐渐从一个编码工程师向 PPT 工程师转变，没有成就感不说，每天被突发的问题包围，一团乱麻，理不出个头绪。回想自己作为开发的时候，只需要专心于编码，其他事情都有人在处理，为什么现在大家都很忙，但仍然还有好多事情没有人在处理。

在这种情况下，相信会有人说 Tech Lead 的时间管理没有做好，导致很多事情没有时间处理。时间管理确实是摆在 Tech Lead 面前的一座大山，然而，我认为 Tech Lead 首先需要对团队中出现的问题有一个整体的认知，厘清问题根源所在，排列优先级，让问题得到解决，才能给自己腾出更多的时间来。

Tech Lead 需要一个抓手，一个能够厘清团队现状的抓手，可视化通常是一个非常有效的手段。在软件开发这个上下文中，将软件开发整个生命周期中的关键环节尽可能详细地可视化出来，列出每个环节存在的问题，不仅能够帮助 Tech Lead 掌握当前项目的技术、流程全貌，也能帮助 Tech Lead 分析出团队在整个过程中的瓶颈点。

什么是 Path to Production

Path to Production（发布之路）就是这样一种可视化手段，它是 Value-Stream Mapping（VSM，价值流图）在软件开发上的应用。Path to Production 将软件产品开发类比为传统制造工厂中的流水线，并将这条流水线上的必要信息可视化出来，通过梳理软件开发生命周期中每个阶段的内容，找到瓶颈点。

为了有效识别问题，Path to Production 通常需要可视化如下信息：

- Process 流程：从产品需求定义到编码开发，到验收测试，最后发布到生产环境给用户使用，尽可能详细地列出软件开发中的每个关键步骤。
- People 参与者：哪些关键角色触发 / 参与到这个步骤中。
- Tool 工具：该步骤具体用到了什么工具。
- Artifact 产出物：该步骤有哪些约定的产出物。
- Duration 时间：该步骤实际花费了多长时间，是否存在优化空间。

Process 流程	需求分析阶段	开发阶段	测试阶段	发布到生产环境
People 参与者	PO/产品经理	产品经理/QA/开发	QA	运维/开发/QA
Tool 工具	看板	开发 IDE、Git CI/CD 工具	Git、CI/CD工具 测试工具	CI/CD 工具 基础设施工具
Artifact 产出物	Backlog、用户故事	代码、测试用例 单元测试	自动化测试、 测试报告	发布说明
Duration 时间	4 周	1 周	3 天	4 小时

上图是 Path to Production 的一个简单例子，通常实际的图会远比这个复杂，应该尽可能把关键的步骤都列出来，更详细地展开，才能更直观看到问题。

Path to Production 要点

Path to Production 实施通常需要一个 Workshop，Tech Lead 组织团队中关键角色代表（如果人数不多，团队成员可以都参与进来），集体协作产出如上图的 Path to Production。

过程中可以通过下面的要点对 Path to Production 进行完善:

1. 流程需要包含所有步骤，尽量详细，不确定的步骤可以进一步找到对应的干系人进行确认，明确每个步骤存在的价值。
2. 要明确所有的参与者对自己的职责是否清楚，是否有沟通计划保证各角色之间的信息传递是高效的。
3. 工具的使用是否符合规范，是否有 license 和权限的问题，工具是否高效，有没有更好的工具。
4. 产出物是否有清晰定义，是否和关键干系人有时间上的约定，实际操作中是否按预期产出，如果没有产出，是否能够重复生成。
5. 有哪些浪费时间的步骤，是否有工作积压或等待的情况发生，是否有更好的办法，瓶颈在哪里。

在一个自治的敏捷软件开发团队中，保证开发过程顺利且高效从来都不是某一个人的职责，而是全团队所有人共同的责任。Tech Lead 责无旁贷需要引领团队构建这样的团队文化，产出 Path to Production 是一个比较好的方式，一方面帮助团队理解整个产品从需求到部署到生产的整个过程，另一方面也能让所有人通过这个过程理解团队中存在的问题，激发团队成员的自主性和责任感。

组建团队

当你成为 Tech lead，意味着你不再只是团队中的普通一员。现在你是开发团队的 Leader，换句话讲，你有一支你自己的团队。你需要费尽心思组建和打造你的团队！

组建搭配合理的团队

作为 Tech Lead，你要先有一支队伍。

当你还是一个 Tech Lead 菜鸟，可能更多是听取项目经理的想法来组建开发团队。但未来你需要从专业角度给项目经理建议，组建更为合理的开发团队。那么团队组建的时候，

Tech Lead 应该注意什么呢?

组建一支有战斗力的开发队伍是打胜仗的基础。就像一支 NBA 球队，首先需要有一名核心球员，搭配 1 到 2 个明星球员，再加上几名角色球员。这才是搭配合理的球队。

近几年 NBA 流行三核心，几乎是争冠的标配。但问题是要么明星球员降薪，要么交奢侈税。同时球星的个人数据也会下滑。

软件开发团队是一样的，大量配比高级研发，通常对研发效率会有较大帮助。但带来的问题就是超出预算，每位高级研发发挥的空间有限，无法满足个人成长需求。

工作经验、能力搭配

团队一般是菱形结构，中间多，两头少。打个比方，一个 8 人的团队，一般组成是 1 名 Tech Lead，1 名高级研发，4 名初级研发，2 名毕业生。

这样的搭配，一能满足项目的成本要求，二能满足不同经验成员的发展诉求。经验不足的团队成员也可以在经验丰富成员的带领下快速成长。

项目经验搭配

如果在遗留系统做项目，人员搭配还需要考虑系统经验。系统经验和能力没有关系，考察的是对遗留系统的了解程度。如果项目组所有成员都对此系统不熟悉，那么即使都是高级研发，风险也会非常高。

遗留系统的坑就在那等着你，可你就是看不到。这种亏吃得太多了。

另外由于团队对流程不熟悉，导致该准备的材料没准备，或者“突然”多出一些工作，影响了项目交付甚至上线。这种事情非常常见。

8 人的团队，如果 Tech Lead 非常熟悉遗留系统，可以对其他成员不做要求。但是如果 Tech Lead 不熟悉，至少还需要 1-2 名熟悉系统的成员。和 Tech Lead 一起制定方案，并帮助团队成员熟悉系统。

技能搭配

团队的一个重要优势是个体的多样性。技能是一个重要的

方面。招聘全栈工程师可以大大提升团队效率，但很难找到这么多的全栈程序员。团队合理的技能搭配可以打造一个全栈团队。

每个技能至少有两名成员熟悉，避免造成单点依赖。

这样的团队没有明显的短板，风险也是最小的。

不同办公点搭配

如果是异地办公，在搭配团队时要考虑不同工作地点的因素。尽量在不同的工作地点都有经验丰富的研发。一是让异地团队更有凝聚力，二是能加速初级程序员的成长。

打造成员多样性的团队

尽管每个人的背景、性格、喜好不尽相同，但如果团队招人的标准过于死板，会造成团队成员的同质化。我们打造团队的时候需要刻意考虑团队成员的多样性。请注意这里的多样性并不是指能力的多样性，例如 3 个后端，2 个前端的多样性。这里的多样性指成员的个性、背景、思考问题的方式。团队成员多样性有如下好处。

思考问题的角度不同

多样性意味着思考问题的角度不同。例如面对生产上出现的严重 bug，有人想的是立刻回退代码，有人想的是通过 log 尝试定位问题。定位问题时，方法也不一样，有的人反过来从 DB 入手，有的人从请求入口入手。思考问题的角度不同，引入解决问题的方式也不同。这样的团队可以从多个角度更快解决问题。

思维的碰撞

讨论问题时，团队成员的多样性有益于激发更多的灵感。讨论问题就是为了集思广益，那么团队的多样性，会产生更多新奇的点子，从而引发更多的讨论。讨论肯定会产生争执。有的管理者惧怕团队起争执，力求团队和谐。其实大可不必，争执是产生新点子的重要方式。人在争执过程中，会极大激发大脑的潜能，经常会冒出自己都觉得不可思议的想法。

团队成员性格互补

每个人都有自己鲜明的性格。你的性格有时是成事的关键，

有时却又是成事的绊脚石。作为个体，改变自己性格非常困难。作为团队，则幸运得多。通过招聘性格各不相同的成员，就可以获得一支性格全面的队伍。

团队成员不同性格互为补充，会让团队行事更为顺利。例如有的成员做事果断，但是容易犯错。有的成员做事思考全面，但是犹犹豫豫。这样的组合既能让事情及时推进下去，又能够兼顾到方方面面。

团队深处复杂环境之中，需要多样性的成员组成全面的队伍。否则外界任何变化，都有可能击溃一支曾经运转良好的团队。

成员有更大的发挥空间

团队成员的多样性给了成员更大的发挥空间。避免大家在同一个方向上发力，导致每个人发挥的空间有限。多样性让每个人感兴趣或者擅长的方向不一样，每人都能找到自己的独立发挥空间。有的人更具管理思维，那么可以辅助 Tech Lead 做管理工作；有的人做事细心周到，可以负责系统上线；有的人擅长沟通，可以做 UAT 支持。每个成员都可以在自己擅长的方向上发展，另外又可以带动其他不

擅长的同事，获得发展他人的机会。

建设同一技术愿景的团队

技术愿景指 Tech Lead 和团队一起设定的长期技术目标。技术愿景是团队的技术追求方向。团队可以围绕技术愿景，设定一系列的技术标准、流程和团队纪律。

制定技术愿景有如下几个作用。

保持团队纪律性

技术愿景指明了团队的技术追求方向，避免团队成员各自为战。对于一个团队，纪律性尤其重要。没有纪律，不能称之为团队，只是聚在一起的一群人而已。

如果由 Tech Lead 直接设立团队的纪律，会造成团队成员主动性不高，遵守不到位的情况。人们对纪律都有本能的抵触情绪。但每个人又都有追求技术卓越的心。通过追求技术卓越的想法，达成一致的技术愿景。围绕团队认可的技术愿景制定团队纪律，在推广和执行上就会容易得多。

指导团队做出连贯决策

团队在项目上的活动是长期的。通过技术愿景，可以确保团队长期的技术活动是连贯的。而不是不同时间的决策方向各不相同，甚至相背。有的时候在做决策的那一刻并不能看出方案的优劣，这往往让我们陷于难以决策的境地。通过设定技术愿景，做决策的时候能够给我们指导。大多数时候，决策的一致性胜于某一次更优一点的方案。

指导团队做出有利于长期收益的决策

人们会本能性选择收益更快的选项，哪怕是饮鸩止渴。缺少了技术愿景的指导，小到一行代码怎么写，大到一个方案怎么做，都很可能只关注了短期收益。

往往迫于交付的压力，我们对技术的追求步步退让。最终团队技术债债台高筑，系统加速腐化。面对这样的系统，需求实现越来越困难，但你的工作量评估却不能随意增加。否则会得到业务部门的质疑：类似的需求去年只需要 20 人天，今年怎么需要 30 人天？这导致交付的压力越来越大，进而放弃更多技术追求。陷入恶性循环后，系统会快速变成一个修修补补、摇摇欲坠、无人敢碰的遗留系统。

从写下系统的第一行代码起，项目团队就在和“熵增”做抗争。技术愿景指导团队做出有利长期收益的决策，避免只顾及眼前利益，导致系统加速走向无序泥沼。

总结

团队由一群身怀绝技又各不相同的人所组成，大家有共同的目标，并愿意为之而努力。打造你的全明星团队，和团队一起设立愿景，这是你团队走向成功的基石！

团队的不同阶段

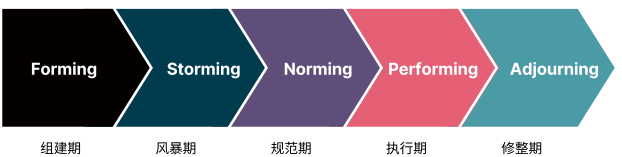
现在你已经成功组建了你的团队。看起来人员搭配合理、个个精兵良将、人人充满干劲，貌似只差一声枪响，就可以在赛场上呼风唤雨、连创佳绩！

但当真的投入到交付中，却发现完全不是这么回事。团队成员各自为战、缺乏信任、效率低下、犹豫不前。糟了，刚起步就遭受当头一棒。

别慌，其实这是团队发展的必经之路。

早在 1965 年，Bruce Tuckman 发表了一篇 16 页的论文

《小型团队的发展序列》（Developmental Sequence in Small Groups），把团队发展分成了四个阶段。这四个阶段不可逾越。在 1977 年他又加入了第五个阶段休整期。这就是著名的 Tuckman Model。



组建期

组建期的团队，成员们刚刚集合，彼此之间还比较陌生。此时团队成员间缺少安全感，目标感也很弱，每个人倾向于各自为战。

作为 Leader，这个时期应该通过组织活动来带动气氛，让成员尽快相互熟悉起来。可以刻意让成员配合工作，增加默契和信任。此外可以通过 team building 的形式让成员尽快熟悉，打破个体间的壁垒。

这个阶段和团队谈技术愿景可能有些空洞。可以先尝试

设定一些具体的团队目标。比如准时完成一个迭代所有 Story 的开发。甚至可以再小一点，比如团队保持测试覆盖率 80% 以上。这样可以让团队成员意识到大家是一个团体，一起努力才能达成团队的目标。

激荡期

团队成员慢慢熟络起来，但是还远远称不上有多高的默契度。

碰到问题的时候，大家由原来的不说话，变成积极表达己见，甚至争吵起来。这个阶段应当让团队成员充分表达自己的想法，然后形成团队规范。

组建期只是有了一个队伍，而激荡期才是让队伍过渡为团队。

我曾经在一个项目上亲身经历过印象深刻的激荡期。Code review 经常拖堂到 2 个小时。好多问题无法达成一致，长时间陷入讨论和争执。当时我的感觉糟透了。但现在想想，这其实是团队发展的必经阶段。

激荡期是团队形成规范的必经之路。作为 Tech Lead 应当鼓励成员进行讨论，并得出结论。这个过程中需要注意如

下几点：

1. 制定时间限制，不要无休止地纠缠在某个问题上，消耗团队太多精力。
2. 控制讨论的气氛，让成员们专注在问题本身，避免过分冲突。
3. 争执不下时需要 Tech Lead 带领团队得出结论。很多问题并没有 100% 的对错，重要的是讨论的过程而不是结果。
4. 需要对团队中强势的 “意见领袖” 稍加控制，尽量让团队所有成员都能参与讨论，发表自己的看法。避免每次都是 “声音大” 的一方占据优势，而让其他成员心里不爽。

这个阶段我们需要团队成员激烈 “争吵”，但也要适当控制时间和情绪，否则会扩大负面影响。激荡期不可跳过，但我们可以帮助团队安稳、快速地度过这个阶段。

规范期

度过激荡期后，团队进入规范期。经过激荡期的磨合，团

队成员在大多数分歧上已经达成了一致，姑且不谈达成一致的结论是否合理，但总归大家有了共同的规范。团队不需要再花费大量精力在讨论和争执上。此时的团队会更加团结。大家在这个时期会逐步建立起团队的目标。

作为团队的 Tech Lead，在这个阶段需要考虑如下事情：

制定团队目标

此时团队成员已经比较熟悉，对团队的信任感也在增强。可以为团队制定更为宏大的目标。

制定团队的技术愿景，统一团队的标准和决策。让团队越来越像一个人在工作，而不是一团散沙。

制定和完善规范

你的团队在激荡期已经产出了一些规范，但这远远不够。这个时期我们要趁热打铁，逐一制定出指导团队行为的规范。比如技术使用规范、代码提交规范、项目活动的流程及会议等等。

执行期

团队成员已经相互熟识，规范也逐渐丰富。此时团队进入了高效产出时期。团队已经可以在达成共识的框架中开展工作，有条不紊，井井有序。

这个时候 Tech Lead 是不是就高枕无忧，可以松一口气了？答案当然是否定的。

在这个时期，Tech Lead 要考虑下面几件事情。

迭代规范

经过规范期，团队已经有了很多规范。这些规范支撑团队进入高效的执行期。但相信作为 Tech Lead，永远不会觉得当前已经做得足够棒了。

事实也是如此，团队制定出的规范和流程一定存在改进的空间。想一想已有的这些规范中，有多少是迫于时间线不得不作出的选择？又有多少是因为某些团队成员嘴上功夫了得而被采纳？又有多少是凭空想象而得出的结论？这些规范可以支撑团队运转，团队效率看起来也不错，但一定还可以进一步完善。

在执行期，我们有了大量规范执行的事例。哪些规范是合理的，哪些有问题，现在都可以用事实说话。我们应该通过事实来纠正、改善、补充规范。通过不断迭代，让规范更合理。

此外，内外部环境的变化会导致运行良好的规范出现问题。这也是我们要不断重新审视已有规范的原因。拥抱变化，找到不合理的规范，在执行期要持续关注。给予团队信任。

这个时期的团队成员已经渐入佳境，个个都摩拳擦掌想要一展实力。

根据社交 HRT 原则（谦虚、尊重、信任），Tech Lead 应该充分给予团队成员信任。信任的建立是需要过程的。盲目信任不能称之为信任。这只是偷懒，或者称之为撒手不管。

只有每个团队成员都充分发挥自己的主动性，争相成为某项事情的 Leader，团队才会朝气蓬勃快速前进。如果所有事情都是 Tech Lead 冲在最前头，那么 Tech Lead 的能力和精力将会成为团队的天花板。

打造团队文化

所谓团队文化，其实就是团队的性格。团队文化发源于团队的创始人，形成于初始团队。后面有新成员加入时会潜移默化被团队文化所影响。新成员会带来一些小的改变，但是不会对文化构成大的冲击。除非团队更换 Tech Lead。

新的团队成员要么融入团队文化，要么就会离开团队。团队自身不允许有和团队文化不合的成员存在。

如果团队文化过于孱弱，也有可能被强势的新成员所影响。这种变化可能是好的，但对团队来说有很大的破坏性和不确定性。团队接受新文化的成本非常高，会产生不和谐的声音。这里建议 Tech Lead 应该强化团队文化，并守护团队文化。

即使团队文化需要发生变化，那么也应该是在 Tech Lead 的主导下，以团队能接受的方式发生变化。

从某种意义上说，团队文化其实是 Tech Lead 自身性格的体现。

休整期

天下没有不散的宴席。作为一个团队，终将会解散。可能是团队目标已经达成或者组织需要变革。

当项目完成时很可能意味着团队的解散。大家可能离开，加入新的团队。也可能团队还在，但要做大范围的人员调整。当然也可能很幸运，团队直接承接下一个项目。

在这个时期，团队成员面临对未来的不确定的恐慌以及对分离的不舍，效率会有所下降。Tech Lead 在这个阶段应该主动疏导大家情绪，开诚布公地沟通团队未来的变化以及大家可能的去向。避免谣言和小道消息在团队蔓延。作为 Tech Lead ，应当帮助大家总结在项目上的收获，以及给出未来发展的建议。

总结

在团队发展的不同阶段，团队效率和氛围有着明显的不同。作为 Tech Lead，应当随着时间推移分析团队目前所处的阶段，有针对性地采取行动，提升团队效率，促进团队成长。

培养团队

当你成为 Tech Lead 后，将面临各种各样带人的问题。比如不好意思给团队成员安排工作；团队成员工作漏洞百出；团队成员能力停滞不前；团队成员缺乏主动性；团队成员个性强，很难带。如何带团队、如何带人，对于技术出身的 Tech Lead 是一个全新的领域，面对问题很容易陷入一筹莫展的境地。本文聊一聊 Tech Lead 如何带人。

带人的心理关

昨天还是在一起结对写代码、一起吐槽前 Tech Lead 和项目经理的好兄弟，今天你却要开始给他分配任务还要教他

做事情。这是不是有点尴尬?

太熟了，有点下不去手？大多数新任 Tech Lead 可能都有这个问题。心里会觉得曾经一起干活的兄弟，现在却要给他安排工作、追问进度，不好意思开口。其实大可不必。这是你作为 TL 的工作职责，没有什么不好意思的。大家各司其职而已。

你表现得更为优秀、经验更加丰富，时机也已成熟。你光明正大地晋升为 Tech Lead，行使自己的职责，有什么可担心的?

你要能够把控自己的团队，那么必须迈过心里这道坎。碍于以往的关系会导致工作无法正常推进。

其实你在意的这些，对方可能并不在意。也有可能对方不适应你身份的转变,甚至对你的工作安排有所抵触和抗拒。但记住，这不是你的问题。作为团队的 Tech Lead，不可能由你去适应他。而是他需要融入到你负责的团队之中。否则只能是他离开这个团队。

自信点，你现在是团队的 Tech Lead，做你应该做的事情，

不要有私人关系和感情上的顾虑。

构建自己的影响力

作为新组建团队的 Tech Lead，你跑到某位团队成员身边说：你好，我经验很丰富，技术也不错，我来教你做事情？

对方肯定一脸懵逼：你想干嘛？

上面的例子有些夸张，但作为团队的新 Tech Lead，你还未在团队中建立起足够影响力，很多事情不能操之过急。

利用好团队组建期和风暴期

我在《Tech Lead 如何应对团队发展不同阶段》一文中介绍过团队在组建期以及风暴期的状态，以及 Tech Lead 应该关注的事项。对于团队而言，此时是磨合阶段，各成员间逐步建立起信任。对于 Tech Lead 而言，这也是构建自己影响力的最佳时期。

风暴期会有很多激烈的讨论。身为 Tech Lead，需要在讨

论时能够输出有价值并且令人信服的观点。这也是 Tech Lead 快速赢得团队信任的方法之一。对于含糊不清的技术问题，Tech Lead 需要立刻付之行动。在快速学习、实践后，给团队讲解清楚。这会让团队成员对你心生敬佩。

在此阶段，展现出你对技术卓越的追求非常重要，并且一定要有理有据坚持自己的观点，而不是左右摇摆。如果只是坚持自己的观点，那么很容易做到。但是做到有理有据让人信服，则非常不容易。首先需要你对问题有充分的调研，另外也需要一定的口才。如果不做足准备，只是以 Tech Lead 的身份强推自己的观点，只会适得其反。

亮出你的代码

大家都是开发，亮出你的代码比夸夸其谈一个小时让人信服得多。

可以通过如下亮代码的方式提升你的影响力。

1. 完成一个功能，给团队打样，展现出你优秀的编程思想和代码风格。
2. 遇到技术问题，主动承担调研解决，并给团队分享。

3. 开发底层代码。
4. 开发技术难度最大的需求。
5. 和团队成员结对开发。

Tech Lead 切忌远离代码。尤其在团队组建期，哪怕加班也要开发代码，否则你建立起的“影响力”如同在沙滩上盖楼。

Tech Lead 的工作繁杂，写代码的时间非常有限，需要有选择性地写代码。关于 Tech Lead 应该写哪些代码，在另一篇文章《Tech Lead 如何应对编码时间下降》中有更为详细的分析，这里就不再赘述。

分析团队

如果是你亲手组建的队伍，你对团队的情况当然是了如指掌。但也有可能是你接手一支队伍。那么你首先要了解这支队伍。团队成员的技术栈是什么，代码能力如何，是否有本项目的经验等等。具体要考虑的几个方面和组建团队考虑的一样，可以参考我之前写的《Tech Lead 如何组建你的全明星团队》

当你对团队有了充分了解后，需要结合项目短期和中长期的需要，对每一位团队成员进行分析，找到其擅长的工作和需要提升的地方。分配工作时，需要考虑如何充分发挥每个人的特长以及给予其一定的提升空间。

分析完团队成员后，需要和每位成员对齐期望和他个人的诉求。工作中尽量为其提供机会，帮助其成长。

如何分配工作

有了目标和计划，最终还是需要在项目中得到锻炼和成长。下面我们看看如何合理地分配工作，帮助团队成员不断成长。

Tech Lead 分配工作时需要平衡两个方面：

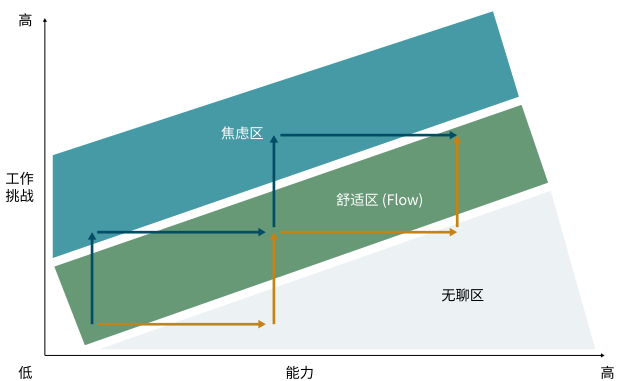
1. 项目的质量和进度
2. 个人成长需求

如果只考虑项目的质量和进度，会导致团队成员一直做自己擅长的工作。但久而久之，团队成员会失去工作的热情。最终结果可能团队变得懈怠，也可能有团队成员觉得无聊

离开团队。

但如果只考虑个人成长需求，安排的都是非常有挑战性的工作，那么出问题的概率必然大幅上升。面临的后果可能是项目失败，或者团队成员屡屡受挫，自信心受到沉重打击。

我们以工作挑战性高低以及人员能力高低为横纵坐标，绘制下图。



舒适区是团队成员工作效率最高，心态也最好的区域。团队成员沿着 "flow" 上升，能力和承担工作的难度同步提升。

这是我们想要达到的理想状态。

我们假设团队成员目前处于舒适区中。此时给团队成员分配的工作需要略带挑战，使其跨出舒适区，处于轻微焦虑的状态。这可以促使其在可以承受的压力下，努力提升自己的能力，再回到舒适区中。此时他已经沿着 flow 向上移动，无论是能力还是可以胜任的工作，都上了一个台阶。重复这个过程，他的能力将得到不断提升。

另一种情况是团队成员能力提升后，工作难度却没有变化。团队成员进入无聊区。此时 Tech Lead 需要尽快提升他的工作难度，让其回到舒适区。否则他对工作会产生懈怠，能力也会止步不前。

如果团队成员的工作挑战性太强，则会造成拔苗助长，功未练成，精神、肉体已练废。

给团队成员分配略带挑战性的工作，观察团队成员的状态，根据能力的提升，适时调整工作难度。这些工作需要 Tech Lead 长期关注。

何时“教”别人做事

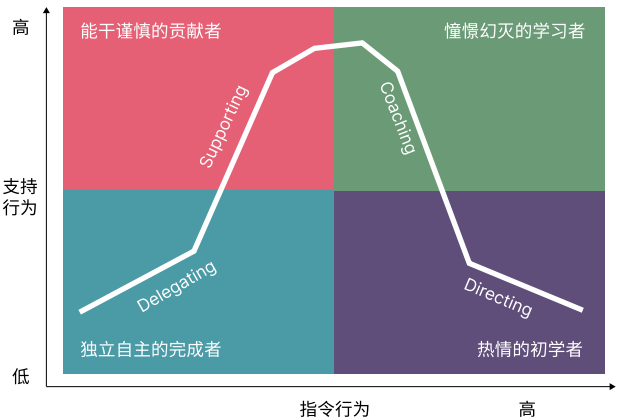
Tech Lead 作为团队中综合能力数一数二的存在，承担着培养团队成员的重任。

培养的方式有很多种。对于初级开发，需要将他要做的事情掰开了揉碎了，告诉他如何一步步去完成。对于有一定经验的开发则会给他更大的自由度。在过程中， Tech Lead 可以给予帮助，发现问题时及时纠正。

大部分 Tech Lead 心中都有因材施教的概念，但是并不很清楚如何去实践。有一个非常好的理论模型能够帮助我们解决这个问题。

情景领导力 -Situational Leadership， 由 Situational Leadership 和 Ken Blanchard 在 1969 年提出，他们认为应当根据团队成员的成熟度不同,选择正确的领导风格，才能获得领导的成功。

情景领导力模型如下。



根据团队成员的不同发展阶段分为四个象限，分别为热情的初学者、憧憬幻灭的学习者、谨慎能干的初学者、独立自主的完成者。

热情的初学者

回想一下团队中最近刚加入的毕业生。虽然他能力确实有限，但是对技术抱有极高的热情，如饥似渴。他们会为技术问题苦恼，也会因为攻克一个问题而兴奋不已。

对于热情的初学者，Tech Lead 更多采用指定工作的方式。将团队成员的工作进行分解，并告诉其这么拆解的原因。

受限于认知能力，此时他可能并不能完全理解如此拆解的原因，但并不重要。他关注的是拆解后的每个具体任务如何去实现。这就像我们学习羽毛球，都需要从一个个基本动作练起。

完成任务的过程中，初学者的关注点在于如何正确处理 DB 中的数据、如何写一个 API、如何写测试等等。不要觉得这些任务都很基础。将项目中涉及的各种技术正确运用起来，完成一个完整的功能，对于初学者来说并不是一件容易的事情。

Tech Lead 对于热情的初学者，应尽量将任务拆分到可以直接执行的粒度，然后按照时间表去检查初学者的进度。

憧憬幻灭的学习者

可以肯定地说，这是最为痛苦的一个阶段。一腔热血的初学者，很快就开始四处碰壁，最终灰心丧气，情绪低落。

还记得我的孩子第一次去练习跳绳时的情景。下楼前他信心满满，觉得很快就能一展身手。但练了半小时后，还是不能连贯跳过三次，导致情绪崩溃，哇哇大哭。

工作中我们学习一项新技能，往往也是如此。

初学者面对各种各样的技术栈，问题不停冒出。一开始还能勉强招架，但随着学习的深入，慢慢就会感觉到力不从心。在熬了几个通宵，头发也掉了一大把后，面对依旧堆积如山的问题，初学者的憧憬将走向幻灭。

在这个时期，作为 Tech Lead 需要在情绪上疏导团队成员。来自 Tech Lead 的鼓励非常重要。在精神上给予团队成员支撑，让其坚定地走下去。

只有鼓励当然是不够的。Tech Lead 需要帮助其制定合理的学习成长计划，给出学习的指导和建议，帮助他尽快度过此阶段。

如果团队成员认为当前工作难度过大，导致自己过于焦虑，也可以适当降低他的工作难度。

能干谨慎的贡献者

这个阶段的团队成员已经具备独自完成一些工作的能力。但是缺乏一些自信，而且在一些细节上还需要 Tech Lead 指点和帮助。

此时，Tech Lead 应该鼓励他独自制订方案，然后倾听他的方案。如果觉得方案有可以改进的地方，需要采用引导的方式帮助其完善方案。

在方案的执行阶段，Tech Lead 需要向其提供所需的资源，给予必要的支持。持续跟踪方案的执行，确保能够成功实施。

按上述方式，团队成员成功实施几次方案后，无论能力还是自信心都会有较大的提升，从而顺利度过这个阶段。

独立自主的完成者

此阶段的团队成员已经能够独立自主完成任务。Tech Lead 此时应该给予团队成员充分的信任，让其主导方案设计以及时间计划。

Tech Lead 需要和团队成员充分沟通任务的重要性以及他是否有意愿和信心去完成。如果他的能力和信心都没有问题，那么可以放手让其独立完成，Tech Lead 只需要关注最终结果。当然，为了避免执行上的偏差，过程中还是需要按照时间表关注工作进展。在这个阶段，时间表应由团队成员来主导创建。

情景领导力小结

上述四个阶段，并不是每位团队成员都会经历，并且每个阶段的时间长度也会因人而异。同一个人，对于不同的工作，会重新经历这几个阶段。

作为 Tech Lead，需要分析当前工作对于特定的团队成员，处于什么阶段。根据不同阶段的特点，针对性地帮助其完成工作。

正确使用委托

对于独立自主的完成者，Tech Lead 应该多采取委托行为。在这里，我想再多聊两句委托。

委托在《现代汉语词典中》的解释为 “把事情托付给别人或别的机构”。比如当事人委托律师出庭。再比如房屋买卖中委托家人过户。我认为委托有两个特点，一是被委托人有足够的专业技能，二是对被委托人有足够的信任。

我常听到 Tech Lead 说：“xxx，我忙不过来了，这个事情委托你做一下吧。”

似乎委托是解放 Tech Lead 带宽的救命稻草。但在 “委托”之后，时常会出现被委托人执行得不尽如人意。甚至最后搞砸了，还得 Tech Lead 来收场。这样不但未能解放 Tech Lead 的带宽，而且被委托人也会信心受挫。

在委托前，需要先问自己一个问题：对于被委托的事情，受托人已经达到 “独立自主的完成者” 阶段了吗？

如果受托人从来没有做过这项工作，你却很信任地 “委托”给他，并且过程中也没有参与，那么往往结果会不尽如人意。这种“委托”，只能称之为偷懒，并不是真正意义上的委托。

信任的产生，需要一个过程。对于陌生的人、陌生的事都是如此。在信任建立起来之前，要小心行事。切忌盲目信任，滥用委托。

根据被委托人的能力和其对委托工作的熟悉程度，Tech Lead 需要做到事先沟通、事中检查、事后复盘。这是对工作负责、对项目负责、也是对被委托人负责。不要因为工作繁忙而不分场景地使用 “委托”。

总结

Tech Lead 跨过带人的心理关以及构建起自己的影响力都不是太大的问题。带人的难点在于因材施教。这需要 Tech Lead 分析每位团队成员的情况，根据情景领导力模型，找到其所处的阶段。根据每个阶段的特点来指导团队成员工作。

多角色协作

Tech Lead 作为软件开发团队的技术负责人，对外是团队技术能力的展现窗口，需要将团队的技术能力呈现给客户或业务团队；对内他需要和各个角色紧密协作，给非技术角色以技术角度的建议及支持。

软件开发团队底层支撑是技术，团队一半以上成员是开发。项目经理、产品经理、QA 等角色对技术了解有限，大多数开发又缺少全局技术视角，并且沟通能力相对薄弱。开发成员和非开发成员间会形成天然的沟通壁垒。此种情况之下，Tech Lead 是串联起各个角色的不二人选。

Tech Lead 技术和沟通能力自然没得说，此外 Tech Lead 对技术的全局观、从客户得到的第一手信息，都支持他能够代表开发团队做出更合理的建议和决策。

软件开发团队中的角色

不同项目模式下，团队构成略有区别，但大体角色类似。软件开发团队一般有如下角色：

1. 项目经理
2. 产品经理
3. 用户体验设计师
4. 测试工程师
5. 开发工程师

其中产品经理、用户体验设计师都和需求相关，类似的角色还有业务分析师、UI 设计师等等。

除了开发工程师，这些角色中和 Tech Lead 合作最为紧密的是项目经理，其次是产品经理，然后是测试工程师。下面我们来重点看看 Tech Lead 和以上三种角色的协作。

和项目经理的协作

Intel 的 CEO 基辛格曾说过，每家科技公司都应该有一位技术 CEO。同样的道理，软件团队的项目经理如果懂技术，对团队益处颇多。但是懂技术的项目经理可遇而不可求，大多数项目经理是非技术出身。不过没有关系，项目经理通过和 Tech Lead 紧密协作，便可以让团队拥有一位技术出身的 “项目经理”。

Tech Lead 和项目经理的协作非常多。我们一个一个来看。

团队组建

项目组成立时，一般最先被确定的是项目经理和 Tech Lead。他们肩负着搭建团队的责任。团队需要什么技能的成员、如何搭配，需要 Tech Lead 有自己的想法，能够给项目经理专业的建议。尤其是开发团队的组成，Tech Lead 更有发言权，有时候需要 Tech Lead 更加强势，争取到关键成员。

开发计划制定

制定开发计划毫无疑问是项目经理的职责。但是合理的计划一定需要 Tech Lead 参与制定。

Tech Lead 需要从开发角度将需求做粗粒度的拆分。然后对每个需求给出粗略的估算，理清需求间的依赖关系。这些工作项目经理无法独立完成，但却是制定开发计划的必要输入。

风险把控

Tech Lead 应该是项目中对技术风险最敏感的角色。碰到了什么技术难题、技术方案哪里不够完善、外部的变化影响有多大，Tech Lead 一定是最先接触并识别到风险。

很多时候，Tech Lead 已经嗅到了风险，但项目经理还觉得一片向好。此时需要 Tech Lead 尽快将风险传递给项目经理。项目经理才能在第一时间尽快做出安排。

我们不要惧怕风险，更不要试图掩盖风险。将风险尽早暴露出来，制定应对策略，是识别到风险时唯一正确的选择。

进度跟进

Tech Lead 也需要关心项目的进度。你可能会觉得这是项目经理应该关心的事情。我们换个角度讲，Tech Lead 需要关心开发团队的效能。让开发团队高效工作是 Tech Lead 最为重要的职责之一。

当团队的进度不理想时，项目经理的手段可能只有右手加班、左手加人。简单粗暴，效果还不一定好。Tech Lead 可以从更深层次入手，分析是团队成员能力不足，还是搭配不当，或者有阻断开发的难题需要解决。Tech Lead 此时会采取更有效的办法解决技术相关的问题，提升开发团队的研发效率。比较常见的方式有调整任务的优先级、集中攻克难题、组织团队学习等等。

项目经理协作小结

Tech Lead 作为开发团队的技术管理者，必然需要和项目经理紧密配合。Tech Lead 这个角色本身也携带非常多的项目经理属性，千万不要对技术之外的问题不闻不问。不夸张地说， Tech Lead 就是半个项目经理。Tech Lead 来补全项目经理缺失的技术版图。项目管理的各个角度，只要需要技术判断的地方，都应该由 Tech Lead 和项目经理协作完成。

和产品经理的协作

产品经理提出的所有需求，最终都需要开发团队来实现。Tech Lead 代表了整个开发团队，自然少不了和产品经理

的协作。

需求的合理性

开发一定抱怨过自己开发的是个什么烂需求，这东西做出来怎么可能有人用。产品经理固然专业，但是也有自己认知的局限。产品经理提出的需求或者设计并非一定合理。Tech Lead 会比其他开发人员更早接触到产品经理的需求设计，需要在早期识别出不合理的需求。此时回头的成本最低，也避免了后面产品经理和开发的摩擦。另外由于 Tech Lead 也会参与需求的前期讨论，所以和产品经理的信息更对等，更容易沟通、达成一致。

需求拆分粒度

在敏捷开发的实践中，需求会被产品经理转化为一张张 Story 卡。控制每张 Story 卡的合理大小是非常优秀的实践。如果 Story 卡过大，需要很多开发一起合作。自己完成的工作不能及时进入测试，需要等待全部开发工作完成。开发创造的价值不能及时向下流动。这导致很多敏捷实践都无法推行。

产品经理对每张卡的大小虽然有自己的判断，但是由于对

技术不够了解，往往会不够准确。如果等到迭代计划会议估点时才发现 Story 过大已经太晚了，无法回头。

Tech Lead 需要在产品经理刚开始写卡的时候就关注 Story 的拆分力度，如果发现 Story 有过大的情况，需要立即反馈给产品经理，并阐明理由和原因。在这个沟通的过程中，产品经理也慢慢会培养出从技术角度判断 Story 大小的感觉，从而拆解出颗粒度适中的 Story。

给产品经理技术加持

产品经理在做产品设计的时候，需要考虑技术上实现的复杂度，一般会选取性价比更高的设计方案。Tech Lead 需要给出专业的建议。这些建议会在很大程度上影响产品设计。工作中我经常会听到产品经理这么说：

“这两种设计我还以为开发难度差别很大，如果没什么区别，当然用第二种用户体验更好的设计方式。”

“我以为这个效果很容易实现，没想到有这么大的工作量。那我们换一种方式。”

此外，Tech Lead 可以结合技术给产品经理更好的设计建

议。甚至可以 “卖弄” 一下最新的技术，拓宽产品经理的设计思路，没准和业务结合起来，能产生更大的价值。这个场景也经常发生，比如我们也会听到产品经理这样说：

“这个效果太棒了，我从来没想过可以这样实现！就用你这种方式来实现！”

“居然有技术可以做到这样，我回去研究下这个功能是否可以设计得更好。”

“这个新技术太厉害了，和业务有很多可以结合的地方！在产品和设计方面能做很多事情。我要仔细想一想。”

业务知识和技术知识的流动

产品离不开技术知识，技术也离不开业务知识。技术知识掌握在开发团队中，业务知识掌握在产品经理手中。Tech Lead 可以通过和产品经理的紧密协作，让不同知识在不同角色间流动。

Tech Lead 可以向产品经理输入技术知识，培养产品经理的技术感觉。另外，我一直深信只有深入了解业务的开发

才是优秀的开发。作为开发团队对外的桥梁，Tech Lead 需要创造更多机会，引入产品经理为开发团队赋能，培养开发团队的业务知识能力。

和 QA 的协作

优化开发效率

软件开发并不是开发完成就做完了。还需要经过 QA 的测试，整个需求才真正完成开发。在敏捷团队中，QA 和开发的工作结合非常紧密。因此，Tech Lead 对团队效率的优化一定要考虑 QA 的工作，否则很可能只优化了局部。例如可以通过测试左移，让质量问题更早暴露在开发阶段，提升质量的同时，让 QA 有精力做更高价值的工作。

质量数据

Tech Lead 需要关注 QA 提供的各种质量数据，如 Bug 率、Bug 分析等。这些数据能够帮助 Tech Lead 找到团队当前的问题。可能是团队的能力有短板，也可能业务不熟悉，还有可能是团队有阶段性的松懈。这些问题往往都隐藏在数据后面。根据数据我们找到蛛丝马迹，然后深入分析予以解决。

给予 QA 支持

QA 在测试过程中会遇到各种各样的技术问题或者环境问题。Tech Lead 需要找到合适的开发人员协助解决问题，尽量保证 QA 能在一个稳定的环境中进行测试，这是 QA 保持高效的前提。在支持 QA 的过程中，应该为 QA 赋予基础的技术能力，让其有能力自己定位和解决基础的技术问题。

总结

如果把软件开发团队比作一台机器，技术除了是核心动力外，还是这台机器的润滑剂。也许部分零件没有润滑剂也能运转，但一定磕磕绊绊，问题频发。想让机器运转顺畅，那么就要让润滑剂出现在该出现的地方。而 Tech Lead 就是为机器注入顺滑剂的人。

干系人管理

回想最初作为 Tech Lead 的情景，说到 干系人 这个词，总感觉是很遥远的，Tech Lead 管的都是技术上的事，干系人管理与我何干。然而，如果观察你身边出色的 Tech Lead，就会发现他 / 她和干系人之间的信任关系非同一般，对干系人也有很强的影响力。

为什么要做干系人管理

为了说明干系人管理的必要性，先看几个干系人管理做得不好可能带来的问题：

1. 被动响应干系人的需求，缺少必要沟通，结果总是驴唇不对马嘴

Tech Lead 讨论起技术来头头是道，甚至发现一个新的技术框架，恨不得晚上加班也要尝个鲜。然而，面对一些和项目干系人的沟通，就没有这么多的热情了，通常比较被动，对方要什么，就给什么，甚至不过问为什么。

项目的关键干系人往往掌握着资源，他们对项目的各种约束非常清楚，会决定什么是项目的成功。当缺少需求背后的这些信息时，开发团队给到干系人的响应往往都是不理想的，在得到一些负面反馈后会持续产生反抗情绪：

> “他 / 她怎么总是找事，不把事情一次说清楚，这么折腾人有意思吗？”

2. 干系人“吝啬”共享信息，一旦有信息都是紧急任务

一些干系人掌握着项目的核心信息，他们并不会将所有信息都共享给开发团队。一方面有很多信息只是在讨论阶段，还没有形成决策，传递给团队会影响团队，另一方面有些

信息某种程度上比较敏感，并不适合让太多的人知道，因此干系人在共享信息方面会显得非常“吝啬”。

作为团队的 Tech Lead，在没有任何上下文的时候，直接接收到一个决策信息，而且通常都是非常紧急的任务，如：

> “明天一早要出一版方案，和 XXX 团队讨论可行性。”
>
> “需求有些调整，本周内要完成开发工作量评估，看看是否能够在规定时间内完成项目开发。”

3. 多个干系人意见不统一，夹在中间，左右为难

在一个项目中，通常不会只有一个干系人，多个干系人因部门不同，角色不同，专业背景不同，对项目的期望和目标不同，不同的干系人会引导项目朝着不同的方向发展。

这些方向有些可能是相互冲突的，这通常都是因为干系人之间缺少必要的沟通，不了解对方的目标。作为开发团队，从不同干系人侧接收到有冲突的需求，不可能也无法给出

一个满足各方需求的方案，左右为难。

4. 优先级突然变化，措手不及

很多情况下，干系人的决策是受各种因素限制的，如整体目标，战略方向，预算，时间等等，已经确定的决策被推翻或降低优先级是经常发生的事情，也有时候个别决策并不成熟，或只是临时方案，已经开发到一半的功能要回退，或者需要采取某些手段规避对当前业务功能的影响。

对于有强迫症的 Tech Lead，这无疑像是在一幅构图精妙，笔法精湛的绘画作品上填上几笔打破设计的线条，让人看着那么难受，简直不能再叫做作品。加上项目时间紧迫，突变的优先级给开发团队带来工作量的增加，各种随意的更改，没有章法，没有规则，令人沮丧。

5. 与干系人沟通经常起冲突

这种情况并不少见，上面几种情况处理不好，沟通走到情绪化的极端，往往会直接导致每次沟通的气氛非常紧张，甚至剑拔弩张，针锋相对，不欢而散。非但问题没有解决，双方还会开始抵触和对方合作，导致很多事情无法推进。

上面几个问题都非常典型，相信大家在日常工作中还遇到

过更多奇葩的问题。但这些难道都是项目的干系人在故意找茬刁难开发团队吗？当然不是，如果他 / 她不是竞争对手派来的卧底，怎么会希望自己的项目失败呢。虽然大家上下文不同，站在不同的视角，但都是为了项目的成功，只是在不同的信息面前的反应不同，因此在某些事情上大家的意见看上去是有冲突的。作为 Tech Lead，掌握着软件开发过程中最核心的技术和方案，需要找到一些方法和这些干系人合作，共同把项目引导到正确的方向上。

如何管理干系人

回顾在项目上跟干系人的冲突，都是因为在同一件事情上大家的意见和想法不一致。因此，要管理干系人就要搞清楚哪些人对项目有着至关重要的影响，理解他们的目标和想法，并想办法让大家站在统一战线上，并肩作战。

识别干系人

项目上通常有很多的参与者，任何直接或间接参与到项目中并可能影响到项目的人都是干系人。

在开发团队看来，其他的人要么是需求方，要么是监督方，

或是发号施令，或是在某个时间站出来“找茬”的。由于冲突的存在，很难让开发团队将所有参与者都放在同盟者的行列。而作为团队的 Tech Lead，要意识到项目的成功一定是经过权衡找到满足所有干系人需求的方案，按时保质保量成功交付才算成功。

因此，首先要做的就是识别参与到项目中的所有干系人，并进行分类和排序：

- 关键干系人，对项目有巨大影响力和决策权，通常是公司高管或投资者。
- 主要干系人，直接受项目影响，并通过项目的成功受益。
- 次要干系人，间接受项目影响，比如项目的支持团队。

理解干系人

并非所有干系人都对项目有同等的兴趣，因此需要跟干系人建立良好的沟通渠道，理解不同干系人对项目的期望，特别是关键干系人和主要干系人。

虽然干系人在主动分享信息时会比较“吝啬”，但如果我

们尝试主动建立沟通（最好是能够面对面地直接沟通），理解干系人真实的诉求和目标，对方也会因为被理解和被关注而更愿意分享他们的见解和意图。

特别是在某些事情上持反对意见的干系人，更要深入理解他们的动机，为什么会提出反对意见，有哪些限制因素，是否有些额外的诉求之前没有考虑到。只有解决了这些干系人反对背后的问题，才能找到双赢的解决方案。

当然，并不是干系人背后所有的诉求都能解决，比如上文提到的“沟通走到情绪化的极端”，见面就起冲突的干系人。在这种情况下，要尝试去寻找能够影响这个干系人的其他关键干系人或主要干系人，往往可能是这个干系人的直属领导，通过加强这些关键干系人或主要干系人的沟通来找到解决方案。

构建专业影响力

作为 Tech Lead，影响干系人的另一个武器就是技术。依靠对项目技术架构的深入理解以及业务发展方向的把握，Tech Lead 可以在项目的技术演进方向、解决方案设计、交付风险等方面给出非常有价值的建议，帮助干系人把项

目引往更加合理的方向，帮助干系人获得成功，这非常有利于构建和干系人之间的信任关系。

Tech Lead 可以通过技术分享，建立和干系人的定期技术沟通，定制项目技术规范，构建风险管理机制等方式，加强和干系人之间的信息同步，一方面更好的理解干系人的诉求，另一方面也能有更多的机会进行技术输出，打造技术氛围，构建在干系人一侧的技术专业影响力。

保持专业客观，解决真正的问题

然而，理解干系人并不意味着要附和所有干系人的想法，将所有人的需求汇聚起来实现一个四不像，这无疑是饮鸩止渴。

在软件产品开发上，并不是所有人都是专家，在和干系人沟通过程中，也要客观地通过一些事实证据来说明实际情况，提供更专业的解决方案建议，对于一些有悖于设计原则或演进方向的内容，和干系人共同讨论，找出一个合理的解决方案，从根本上解决问题。这非常有助于在关键干系人面前建立专业影响力。

持续构建开放合作的氛围

与干系人的沟通应该是持续的，有效的沟通渠道可以帮助开发团队及时反馈目前遇到的问题，也给干系人一个分享想法的渠道，促进信息的流动，积极构建开放合作的氛围。

像上文中的比喻一样，软件产品开发就像很多人共同完成一幅绘画作品，需要各方彼此理解，相互信任，统一目标，这是一门艺术。作为 Tech Lead，要发挥自己的专业优势，努力构建和干系人之间的信任关系，推动项目成功。

理解流程

请允许我在这里以一种低姿态来讨论流程。流程当然可以被聊得高级且深刻，就像在《创新者的窘境》一书中谈论的那样，把流程与公司的文化还有价值，以及创新能力联系在一起。但我们不如来优先解决眼下日常工作中的流程问题。

如果我问你，你多大程度上喜欢当前工作内的流程，你一定会觉得我疯了。流程能给人无感，就已经是相当正面的评价了，大部分时候我们吐槽的是它怎么能设计得这么反人类。

我的建议是，当我们决定公允地对它进行评价前，我们考虑的问题不应该是有了它会怎么样，而是没有它会怎么样。

我们从一个会贯穿整篇文章的例子开始：在我的团队中，我们需要以邮件的形式频繁和国外客户沟通，在开发人员回复客户的邮件之前，我们曾经有过类似这样的规定：

1. 邮件请遵循团队提供的邮件模版；
2. 发送前找有经验的同事评审（review）邮件内容

流程是对经验的继承

我先单说模版。

光是听到这个流程我想你们当中已经有人不爽了，这不是在侮辱人的沟通能力嘛——其实大部分开发人员的沟通能力并非理想中的优秀，再加上非母语环境，这种不理想会被放大。最后传达给客户的感受就是不专业的低效沟通。这里的优秀，一部分指的是单纯的沟通技巧，即把一件事有条理且简单扼要地解释清楚；另一方面是指在项目的上下文内读懂问题背后的问题。后者更重要。

例如客户邮件询问某个页面他为什么无法访问，开发人员在没有经验的情况下会下意识从程序设计角度去解释：该用户缺少某种类型的角色身份，邮件中引用的角色可能还直接复制自代码里的驼峰变量。这对于非技术背景的业务人员太不友好了，并且这里并没有（完整）回答客户的（隐藏）问题。

我们不妨看看根据模版还需要填充什么内容：

- 它是不是一个 bug
- 如果是，它的影响面多大，它的原因是什么，短期方案怎么解决，长期方案怎么解决，需要多少天开发
- 如果不是，我们是在什么时期开发的这个需求，它的背景是什么，我们如何配置才能让这个用户访问页面

模版的魅力便是在此，比如它能够让新人快速上手，减少犯错的空间。

流程是对经验的背叛

相比之下，第二个步骤争议颇多：每一封邮件需要在发送前找人评审，相信在多次的沟通练习之后，大部分人都可以驾轻就熟。

在这个例子的开头我其实撒了一个谎，这整套流程在我们的团队中并非规定，而是「指导意见」（guideline）。

决定将一套经验是否上升到流程高度，在我看来只有一点：没有它，我们将付出多大的额外成本。这里成本由两部分组成：重复的学习成本和试错成本，即效率和风险。

试想我们的经验如果只是口口相传的话，每个加入团队的新人都需要老人言传身教一遍，即使在付出了这些时间之后，他可能依然找不到对的人或者把步骤遗忘，那不如把经验固化成流程。这便是为了节约学习成本。

至于风险，在上面的例子中，如果不巧新人完全没有遵守建议却自由发挥的话，实话实说代价还是可以承受的，即低风险。我们会收到客户抱怨我们不专业的反馈，但不至于威胁到我们和客户的合作关系。可在某些领域中，安全问题哪怕只发生一次的概率也是不允许的。

流程是效率对风险的妥协

理想情况下我们期望流程带来的是高效率和低风险的双收益，但实际情况是风险降低了，效率也同样降低了。

想象一下把上面的例子转化成流程多半是这么落地的：在发送邮件前我们需要在系统内录入各种表单然后提交，接着等待各位大佬的审批回复，有可能邮件被再次打回需要重新修改。来来回回一天之内能把邮件发出去就万幸了。诸如此类的流程还暗示了另一件事：流程是对上负责而不是对下负责的。这会带来权力与义务倒置的问题：一线执行者需要完成大部分工作却因为流程束缚住了手脚。

流程负面效应最极端的情况是带来“平庸的恶”。即当流程在被精细切割为不同的环节之后，流程中的每个人都只对自己所处的环节负责，而没有人再对整个流程的结果负责。

这在事不利己的流程中极为常见。比如你的团队负责维护一个面向全公司的流水线部署系统。此时另一个部门的同事希望在流水线中集成一个插件用于生成测试覆盖率报告。假设他从今天上午八点提出一个咨询的工单，如果流程并

没有限定具体响应时间的话，即使你明天八点再回复他虽算不上合情合理但也不为过。即使过了这关，后续的权限申请、审批、调试可想而知遥遥无期。官僚的“魅力”便在于此。

这便是大家讨厌流程的原因之一：我们看不到真金白银的收益，赤裸裸的事倍功半却是有目共睹的。悄悄修正一下，更准确地说收益后置得太远。如果你把流程想象成为安全带会好理解很多。问题是你让我骑个自行车也要求强制系安全带就过分了。

之所以出现这样的情况，是因为风险多半是流程的出发点，同时在流程设计是前置的，也就是我们并无经验但是却又想防止问题的发生，于是保守是第一选择。本来开头的例子是一种极少的理想情况，也就是说我们先在工作中积攒了相当的经验，然后这些经验看上去可靠可行，再选择把他们固化下来成为流程。

用魔法对抗魔法

为了解决这个问题，从我的经验看有两个方向。

首先流程不应该是一成不变的，它应该定期被评审被反馈。最主要的原因当然是我们希望那些以“保险起见”名义的工作量在反复验证无效之后被移除。但评审的原因并非只有收缩这一个方向，例如当团队中有相当数量的新人加入而产生风险时，扩大化也可以是选项之一。流程的状态始终应该是处于适配中。这有一些难度，因为正如上文所说流程其实是对上负责的，撼动它最好的方式其实是从上至下，或者团队内部有通畅的沟通机制让执行者的声音能够被听到。

第二个建议听上去可能是反流程的：流程应该有退出机制（opt-out），即在关键的时候我可以不遵循流程，当然这不应该是常态。但我越来越感受到在处理一些突发事件上或者当流程手册里没有涵盖当前情况时，去遵循你的经验随机应变说不定才是最好的方案。

再次回到最初的例子，在我提到的邮件模板中我们需要告诉客户 bug 的根本原因是什么。但如果寻找 bug 原因的人力耗费巨大，我们不建议他们这么做；又或者它根本是第三方云服务商的问题超出我们的能力之外，那如何解释完全就依赖个人能力了，这超出了模板范围之外。模板能

够表达得显得捉襟见肘，这个时候模板反而成为了一种束缚。

我想说的是，如果流程算是一种“必要的恶（Necessary evil）”的话，那自由则是另一种对抗这种恶所必须存在的“必要的恶”。

我想起来哈佛商业评论的一篇谈苹果公司如何创新的文章《How Apple Is Organized for Innovation》，通篇阅读下来给我最大的感受即是对于人而非制度的尊重。例如公司内部的行业领域专家会为所有的技术决策负责；产品团队可以独自决策而不必遭受财务压力的约束。

我想说的自由的价值也是如此。

作为 Tech Lead

上面所说关于流程的各类迭代，都离不开人在其中付诸努力。在我看来最合适的人选是团队中的 Tech Lead。

为什么是 Tech Lead？简单粗暴的原因莫过于他需要为团队的交付负责，并且流程又是围绕交付来建立的。实际点

说流程的制定离不开 Tech Lead 的判断，这些判断又离不开对于项目上下文、交付价值的理解。这些都是 Tech Lead 角色自带的天然属性。

当团队成员感受到流程带来的挫败感时，这种挫败感是真实的，但不意味着流程就必须得到修改。本文开头的例子是其中一种情况，更多时候流程可能是一种多方妥协的结果，而团队成员只站在了他所处的环节上考虑利弊。流程解决的终究是群体而非个体的问题，我们必须要承认在推进的过程中短暂的不理解是被允许的。

在我看来迭代流程的关键问题不是如何去做（how），而是何时去做（when）。关于如何去做，即使是在我们没有经验的情况下试错也是可行的。但如果 Tech Lead 没有任何渠道来帮助他嗅到流程中的问题，这才是最为危险。

昂贵的质量

"To err is human"

在过去相当长一段时间内，我都在一个负责项目维护的团队内工作。团队的特殊之处在于，我们从来不开发新功能，而是负责解决每天上报的线上问题。这些 bug 无奇不有，从无法打开页面到数据奇怪丢失，麻木早已经替代焦虑成为了我们面对 bug 时的主要情绪。

但我时不时的抱怨依然是：为什么 bug 总是在发生。

缺陷早已在丰田生产系统 (Toyota Production System)

中被标注为浪费之一。没有人希望看到 bug，我们不想，客户更加不想。但我们似乎都不愿承认的一个事实是，bug 是代码的副产品而已。如果我们选择接受编码不过是人思维活动的一种形式，与思考无异，那我们也就必须接纳，人性的缺陷在代码中自然也不会缺席，恰如硬币的正反两面。

反过来说，如果你对 bug 采取的是零容忍的态度，甚至不惜把此写入 KPI 中，它也未必会带来正面效应，因为自此开始，没有人会愿意重构，没有人会愿意引入新的技术方案，道理非常简单：改动越多风险越大——这是某年发生在我所属团队的一次亲身经历。

所以我们面临的并非 bug 去或者留的选项，而是多与少的问题。

摆正质量

在 MoSCoW 方法论的框架下，我们通常可以将功能的优先级划分为四类：Must Have, Should Have, Could Have 以及 Won’tHave，如果把质量也作为功能的一个维度落入这四个区间之一的话，它一定不会在 Must Have 这个

范围内。因为人们既不会因为没有 bug 而选择长时间地使用一款应用，也不会因为存在 bug 而成为转投它竞争对手的唯一理由。我们必须承认这样一个事实：质量永远也不是商业的核心竞争力，这也暗示着：

1. 有些功能缺陷是可以被容忍的
2. 质量被分配得到的资源永远是有限的。

前者并非我们的一厢情愿，在互联网产品的高性价比和快速迭代的商业逻辑下，用户对产品质量的预期已经被规训到一个非常「理想」的状态。

而后者更为关键：我们应该如何最大化利用有限的资源去提升质量？如果你所在的部门也有机会组建一支类似于本文开头的团队，那不妨考虑一下这个建议：用资源换取更多的人员加入这支团队怎么样？

原谅我用一个粗俗的比喻来解释为什么这么做行不通：我们换来的只是打扫的速度，对制造垃圾的人产生不了任何影响，效果甚至会适得其反：考虑到总有人为他们收拾残局，我们的善后工作做得越好，他们越是会肆无忌惮。

但这是当下大部分公司的现状：如果线上问题激增，是不是 QA 工作不到位？我们似乎倾向把编码和测试的界限划分得一清二楚，从人员到职责到工序都是如此。而质量问题从编码中来，却想从测试中寻找解决之道，这与刻舟求剑无异。

铺垫了如此之多，我想表达的观点依然是老生常谈：质量内建，以及最近几年我们常常提倡的测试左移。至于什么是质量内建和测试左移，并不在这篇文章的范围内，你在网上可以找到大量的专业文章来介绍他们。

质量的成本

现在我们必须回答一个核心问题：谁该对质量负责？最官方的答案是每个角色，我们可以列举出产品经理没有全面地对功能做验收，QA 没有把控好质量关等等。但假定此刻我们必须指定一人将他五花大绑起来祭天，为的是有人需要为上一个让人焦头烂额的 bug 负责，程序员绝对是不二之选。

但程序员死也不会瞑目的。

如果你是团队经理，你发现上个月线上问题数量增加了一倍，于是你冲到你的工程师团队工位前，怒不可遏地冲他们吼道：我希望这个月的线上问题不超过个位数！你觉得他们能做到吗？

我想说的是，质量不是「希望」的结果，它是付出的收获。关键在于你愿意用什么去交换。

提升质量的诀窍一点也不神秘。口口相传的各类业内实践便是最好的灵丹妙药，比如重构、代码评审、结对编程、流水线集成等等。我们不妨就以单元测试为例，看看我们需要付出多大的成本

首先，时间便是一笔可观的支出。以我所在的项目为例，我们为前端 React 组件编写单元测试的时间，几乎与开发功能的时间相同。注意这还是在没有追求覆盖所有的边界用例，以及没有追求 100% 的测试覆盖率的情况下。另外，当我们编写的代码导致之前编写的关联测试无法通过流水线时，去查找失败的原因以及修正这些错误也是隐形时间。

其次，在我看来最难以逾越的障碍在于对工程文化的重塑。

对下要强调保证程序员去写、关注、修复的纪律；对上要争取理解和空间来辅助这些实践，单拎出来任何一件事推动起来都不简单。工程实践天然具有一种反商业活动的特性，它很难被量化、难以一针见效。没有人敢拍着胸脯说在 xx 天之内或者当测试覆盖率至少达到 xx 水平之后，bug 数量会降至 xx。我们都不否认它会变得更好。讽刺的是实践带来的「负面」效应是立竿见影的：工程团队的交付速度变慢了。

测试的部分好处还来自未来，比如它能增强我们重构代码和变更架构时的信心，防止旧功能无意被新功能破坏。但我依然无法保证你的收益会何时何地有几倍于付出的到来。

于是现状变成了一方面可见的迭代速度变慢，另一方面成本的付出看不到收益，质量就自然值得被牺牲。

在提升质量的过程中，我们解决的不是个体问题而是工业问题，细想这是很难的：一个团队中成员的能力不同背景不同，但我们却要设法让它们的产出质量处于某个水平之上。

还债

听上去我们只能义无反顾去相信（take a leap of faith）某些实践能够帮助我们提升质量，且付出的收益是不确定的。但换一个角度想，我们只是在做一些本该做好的事情而已，用户的宽容让质量变得可有可无。

我不否认有时候快比好更重要，只不过当有一天质量变成我们无法再忽视的问题时，别不知所措地想不起来质量是在哪里搞丢的。

重塑影响力

我们在这里不会谈社交媒体上的影响力，这需要动用种类繁多的运营和技术手段。本篇文章里要解决的问题非常简单：如何才能让他人跟随你的建议行动。

影响力无关权力

上级对下级的影响力天然成立吗？不尽然。虽然权力阶梯存在，但并不意味着权力绝对有效。纵然强制力可以保证事情得以发生，执行效果却并不掌握在 leader 手中。我们都明白如果团队成员既不认可也不理解接下来的愿景，他们会做的只是取悦般的服从而已：我不会花心思把它做得

更好，它的成功也与我无关。

我们每个人一定见过某种类型的角色，虽然他们不带特殊头衔，但他们的意见总会被认可，他们的建议常常被采纳，甚至你会发现当同事们遇到麻烦时，也是第一时间想找他们寻求帮助——这便是影响力的魔法。相反，我相信在我们每个人的职业生涯中，都曾经厌恶或阳奉阴违数不清的经理以及领导。我曾经阅读过一本非常好谈领导力的图书：You Don't Need a Title to Be a Leader，这个标题在这里稍作修改依然成立：You don't need a title to have influence.

我们在影响谁

在讨论如何影响「他们」之前，我们首先要搞清楚「他们」是怎样一类人群——不过是一群程序员不是吗？——这恰恰是他们的特殊之处。与传统文职人员或者蓝领工人不尽相同，虽然软件制造如今看来是与劳动密集型产业无异，“极客精神” (geek) 的特质依然保留在他们的骨子里。

如果你在 Google 中搜索 “how to manage geeks”，有一篇经久不衰的文章会出现在搜索结果前列：Opinion:

The unspoken truth about managing geeks。这篇发布于 2009 年文章里观察到的现象以及得出的结论现在看来依然不过时。采用文章中的原话一言以蔽之就是：

> IT pros will prefer a jerk who is always right over a nice person who is always wrong.

与其他行业的人员相比不同之处在于，程序员服从的并非某个头衔或者某套流程，这里服从的本质上是对他人发自内心的尊重，而尊重来源于对一个事实的判断：你可否把事情做对。这个前提是如此地重要，以至于让工作中的其他内容都显得黯然失色。正如我在援引文字中所说的那样，哪怕你是个混蛋也无所谓。在该前提下程序员的种种行为用传统的职场观念来衡量起来是匪夷所思的，例如程序员之间的争吵通常围绕的是更好的解决方案是什么，而不是谁来做以及如何能干得更少；上级的建议也不会是必须遵循的金科玉律，负责执行的程序员们会按照自己的想法实施，不幸的是程序员自己的想法永远都是对的。

与需要巧言令色讨好其他职位的角色不同，赢得程序员尊重的标准答案仅此而已。如果你意识不到这条潜规则的存在，便会在和程序员打交道的过程中步履维艰。

当我们认识到这个事实之后，便有了塑造影响力的方向。

塑造影响力

把事情做对无诀窍可言，对业务理解要正确以及用代码表达要到位。然而这听上去像是人人都应该做到及格线，我们又该如何借此来扩大影响力呢？

我的建议是「专精」：你的答案要比他人更接近正确才行。

刚刚我们在回避一个问题：「正确」的定义是什么。当我们在聊「正确」的时候，我们其实聊的是实实在在的东西，代码的正确性包含它的复杂度、可维护性以及执行效率等等。并且它们是绝对的，在解决同一个问题时如果你的代码行数更少、运行时间更快、可读性更好，那胜出便当之无愧。

如果上面的叙述还过于抽象的话，不妨看这么一个例子：如果有人建议在你的前端项目中引入端到端测试，你会如何考虑？「正确」要解决的第一重问题是要不要做；第二重问题是我应该如何去做。

在解决第一重问题时，我们需要知道：

- 端到端测试是什么？
- 项目中有什么问题是必须使用端到端测试来解决的？
- 端到端测试是唯一的方案吗？单元测试是否也能达到同样的效果？
- 端到端测试是否适用于我们的项目？

这四个问题看起来平平无奇，但每一个问题的背后都涵盖巨大的信息量。例如最后一个问题当在考量端到端测试在项目中的可行性时，我们既需要对这项技术的横向（与其他测试技术之间的差异）和纵向（目前的技术生态和业内实践）知识都有所了解，也需要掌握前项目的现状，因为维护成本高昂的端到端测试并不适用于快节奏的交付。

如果提出建议的人只是偶然间在某篇文章中读到了「端到

端测试」这么一个时髦的技术词汇（buzzword）而抛出来讨论，恰巧你又有实践端到端测试的经验，并且不认为当下是一个引入端到端测试恰当时机，那么你便可以有理有据的反驳他，影响力于是在此彰显。

从这个影响力落地的例子不难看出达成「专精」并无他法，它等同于你在某个领域内知识积累的厚度。而如何达成这个目标呢？我们其实又回到了一个纯粹又古老的问题：如何让自己从新手成长为专家？相信每个人都有自己的答案。

让声音被听见

去他的“酒香不怕巷子深”——如果你现在已经「身怀绝技」，请务必要让别人看见。没有对话，施加影响力也就无从谈起。那如何被看见呢？可能你正在苦恼每天的工作按部就班，团队内的技术趋于稳定，没有机会给到我。

我的建议是为自己创造机会。

你要相信缺陷是永远存在的，我们每个人都在编码过程中经历过妥协，过去的妥协便是未来的改进之处。作为每天接触代码的一线人员，识别并修复它们是最好的机会。你

不妨选择一些擅长的领域先把自己投资进去：之所以称之为「投资」是因为你可能需要动用到工作之外的时间去整理它们、寻找解决方案以及准备材料说服大家。

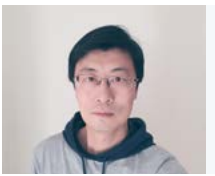
并非每一次提议都会得到支持，你需要把拒绝作为常态去接受。这种拒绝大部分时候不是来自对于你建议的否定，而是没有足够的资源把你的提议落地。但是没有关系，在和团队分享的过程中正确性已经得到了大家的肯定，影响已经发生。反观整个过程也是你技术成长的痕迹。

如果你宁愿等待机会，则需要考虑这样一个问题，当机会来临时你可否抓住它？不要看到机会之后再去完善我的知识体系，而是应该不断地丰富知识体系去静候每个机会的降临。

作者简介

李光毅

咨询师，
全栈开发工程师



李一鸣

咨询师，架构师

麻广广

咨询师，架构师

Thoughtworks 洞见

Thoughtworks 程序员新声