目录

[前言 2](#_Toc14877)

[产品篇 2](#_Toc1579)

[太关注次要的功能 2](#_Toc8317)

[不会借鉴 3](#_Toc31302)

[降低产品设计风险 5](#_Toc17951)

[盲从用户需求 6](#_Toc7332)

[技术篇 9](#_Toc23001)

[技术思维的惯性 9](#_Toc21278)

[失败的架构 10](#_Toc15115)

[过分依赖第三方框架 13](#_Toc5315)

[滥用语言特性 14](#_Toc27921)

[版本控制的常见问题 17](#_Toc9716)

[技术研发的浮躁现象 20](#_Toc8023)

[缺少自动化工具 22](#_Toc19127)

[未能预测系统的行为 23](#_Toc14062)

[花太多时间救火 25](#_Toc18548)

[反思容器化之路 25](#_Toc28110)

[运维篇 26](#_Toc6832)

[未能从事故中汲取经验 26](#_Toc32043)

[数据没有备份 29](#_Toc6787)

[管理篇 30](#_Toc13200)

[矛盾和冲突影响效率 30](#_Toc5852)

[任务依赖造成拖延 32](#_Toc7314)

[违反康威定律 35](#_Toc3027)

[多人合作的懈怠问题 37](#_Toc20314)

[违反约束理 38](#_Toc22933)

## 前言

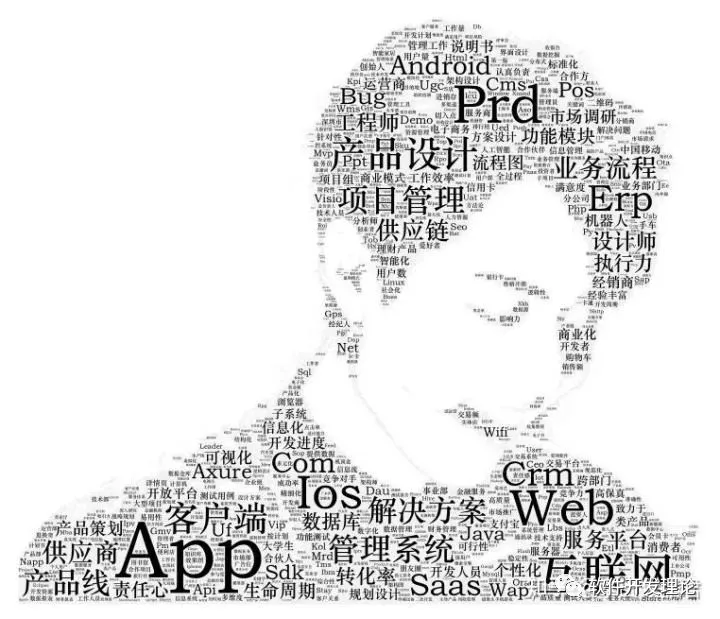
如果要学会一门本领，可以看一些相关的最佳实践，这是我们学习的案例，是我们应该照着做的。也可以看一些负面教材，这是我们不应该去做的。本书主要写的是软件开发过程中的一些常见的错误和问题。如果有啥写得不对的地方，请帮忙指出来哈,我的邮箱是：714501680@qq.com。

## 产品篇

### 太关注次要的功能

周六跟同学去聚餐。扫码下单的时候，我们发现扫码下单的小程序可以支持用户输入生日，用户生日那天过来吃饭的话会有优惠券。令我们惊奇的是，生日的输入不光支持阳历，也包括阴历。这个小功能做的非常地齐全。我们也感叹现在小程序做得水平真不错，一个细节竟然如此细心。可是当我们支付的时候，出了点小故障，一次竟然没有支付成功。在多次支付后，我们终于下单成功。我们也开始质疑这个小程序的质量问题。我们认为，这个小程序的核心功能是点菜和支付，核心功能还有待提升。选生日是次要功能，选生日做得很完善。小程序没有产品的抓好核心功能，即使次要功能做得再好也算不上好的产品。

在产品设计中，各种功能的地位和作用不是平衡的，有一些功能属于主要的功能，其他的功能只是次要的功能。在设计和开发中，应该花精力从多种任务中找到主要的任务，从而掌握工作的中心环节；当功能的主次发生变化，产品进入一个新的阶段时，要善于找出新的主要任务并及时转移工作重点。应该把产品的主要功能和次要功能当做一个有机的整体予以统筹兼顾，发挥它们之间相互促进、相互制约的作用，以推动产品的构建。



产品经理人才画像

产品经理的产出等于他直接影响的产品功能的效用之和加上他所间接影响的产品功能的效用之和。这里的效用指的是在考虑收入和成本等因素后对团队的综合价值。如果细致一些，也可以用一个公式来表达：

产品经理的产出 = 产品功能效用A \* 产品经理贡献百分数A + 产品功能效用B \* 产品经理贡献百分数 B + 产品功能效用C \* 产品经理贡献百分数 C ......

这个公式中，产品功能效用指的是某个功能的综合价值。这个效用可能是正数，可能是0，还有可能是负数。当某个功能的对产品本身很有帮助，产品本身很有用，说明这个功能也有价值。有价值的功能不一定效用为正。因为实现每一个功能都需要成本，如果实现这个功能的成本大于其价值，说明功能的效用为负。当某个功能的价值与成本相抵时，功能的效用为0。当某个功能的价值高于成本时，效用才为正。产品经理贡献百分数指的是产品经理对功能的实现所做出的贡献的百分数。一个功能的实现需要多方参与，比如产品、开发、测试、运维等等。每一个角色都对这个功能的实现有一定的贡献。而贡献的多少可以用百分数来量化。产品经理对功能的贡献百分数也可大可小，因为有的产品经理只负责产品需求的编写，而有的产品经理的活动则贯穿了软件开发的整个生命周期之中；有的产品经理只是完成了工作，有的产品经理具有天才的想法。

产品经理要增加产出，有三个方式。提高自己对每个功能的贡献百分数、提高每个功能的效用、增加功能个数。考虑到团队和产品经理的限制因素，这三个值都不可能无限地增加。但产品经理应该更多地关注效用高的功能，或者说重要的功能，少做效用低下的功能，摒弃效用为负的功能。有的产品经理能抓住重点，有的产品经理刚开始能抓住重点，可是在软件构建过程中，因为某些因素而颠倒了主次关系。比如说，某些客户比较活跃，他们常常就某些次要的需求与产品沟通。产品经理可能会产生一种错觉：客户这么重视这些需求，那我也应该最重视这些需求。这些需求描述的就是最核心的功能问题。于是乎，产品的大把时间都花在了这些需求上，主次就颠倒了。再比如，在设计和开发核心功能时，由于一些客观因素，各方面资源欠缺，交付时间非常紧急，产品明明知道这个功能很核心，但是也只能草草了事。还有些时候，各个负责人对于什么是核心功能有着不一样的见解。一旦错误的观点占了上风，主次也会颠倒。

为了抓住核心的功能，产品应该深入理解用户市场和用户需求，不光掌握需求的内容，还要掌握需求的主次关系。产品和开发应该随时就这些问题进行沟通，就重要观点达成一致，避免在次要功能上花太多资源。产品还应该认真聆听和对待用户的反馈，把每一次反馈作为升级产品的机遇。

### 不会借鉴

借鉴是产品设计的一种方法，也是产品经理必备的基本技能。但不得不说，很多产品经理，尤其是刚入门的初级产品经理不太会借鉴。

回想一下我刚开始设计产品的情景。那还是在大学，几个同学组队一起参加互联网+创新大赛。团队有产品经理、前端开发、后端开发、UI设计这四种角色。大家每周带上电脑在星巴克聚一次，讨论一下各自的疑问和进展。

项目的进展非常缓慢。我们找到了原因：每次开会，大家讨论的都是产品的问题，而不是开发的问题。项目还处在产品设计阶段，这说明产品设计得非常缓慢。大家不是对产品的设计不够理解，就是觉得产品设计不够合理。产品经理也只能参考大家的意见，对产品一再修改。

产品经理总是希望把自己的创意运用在产品上，就没有想过去借鉴。一个刚刚出道的产品经理，很难有完全自主地做产品设计的能力，因此，我们的产品设计并没有那么成功。后来一位老师给我们指点迷津，让我们多看一看其他的产品是怎么做的。我们花了很多功夫，下载了很多个手机APP，一个一个地注册、登录、使用，终于找到了门道。最后我们获了奖。如果要总结一些经验，最重要的应该是我们学会了借鉴吧。

#### 借鉴的意义

借鉴是生物界普遍存在的。动植物借鉴其他生物或非生物的行为或状态，提高了他们适应自然的能力。人是最擅长借鉴的生物，借鉴几乎存在于人的行动的方方面面之中。

绿色的叶子

描述已自动生成

叶虫模仿树叶，免于被鸟类吃掉

比如要学会写文章，就要先读别人的文章，自己照着写。要学写字，就要先临帖，然后形成自己的风格。要挣钱，也要借鉴其他人的生财之道。很多时候，学的过程，就是借鉴的过程。学校安排的老师，就是学生的借鉴的对象，所以有了”学为人师，行为世范”的说法。

软件产品的设计也离不开借鉴，它主要有这些好处：

##### 培养产品设计能力

怎么才能提高产品经理的产品设计能力呢?产品设计虽然不同于艺术，但也有类似的地方。学产品设计和学艺术的方法也有共通之处，最好的方式是先借鉴，然后从借鉴中走出来。所谓的借鉴，就是把那些好的设计放到心里。如果缺少了借鉴这一步，产品经理在做新产品设计时，只能根据生活中的一些经验去思考，很难信手拈来。借鉴得够多，无论要设计什么样的产品，脑子里面都会有类似的产品的影子。在这个基础上继续优化，就有好的产品了。

“取法乎上,仅得其中;取法乎中,仅得其下”。定最高的标准，能达到中等水平；定中等的标准，只能达到下等水平。这句话说明了高标准的重要性。在借鉴过程中，应该注意要找一些优秀的、高等的产品来借鉴，否则会事倍功半。

##### 提高产品设计效率和质量

在产品设计中，效率和质量问题是最关键的，借鉴能提高产品设计的效率和质量。凭空去想，设计的各个环节的效率都将减慢，而且想出来的点子不一定靠谱。一般来讲，那些运营很成功的产品的质量都是经过验证的，借鉴对这些现成的设计做一些修改和优化，质量也不会很差。

降低产品设计风险

自己慢慢摸索，产品可能漏洞百出，恰当的借鉴能让我们少走很多弯路。产品设计的风险主要指设计中的一些不确定性，它们可能给整个产品的构建过程带来灾难。因此，必须降低这种风险。借鉴能让我们少踩别人踩过的坑。

#### 借鉴的方法

##### 找到借鉴的目标

找到了合适的目标，借鉴就成功了一半。软件产品有不同的类型，可以囊括各种不同的元素，因此，可供借鉴的目标非常多。

最直接的办法是借鉴类似产品的设计。比如说，要设计一款电商产品，肯定绕不过淘宝、京东、拼多多这些产品。要设计炒股软件，就可以学习同花顺、广发证券易淘金等。并不是只有这些有名的产品值得借鉴，因为很多产品不是面向大众的，而是面向企业的，他们的知名度没那么大，但也是好的产品。另外，不是说一定要选最好的产品去模仿，而是要选择符合自己团队的。那些最优秀的产品，往往需要高昂的成本，有着庞大的用户体量，如果团队缺少相关的资源，或者团队有着不一样的目标，就不适合去借鉴它们。

从书籍上借鉴也是一个好办法。书里面的知识都比较体系化，比较有深度，它能让我们深入地理解某一个产品、某一个行业。在设计某一类产品的时候，借鉴一下相关书籍里的内容，会提升产品的专业水准。

一些艺术或者人造物体，例如美术、音乐、建筑、舞蹈、交通设施、城市规划等，也可以指导产品设计。软件产品和它们是大相径庭的，但如果剖析设计的难点和设计的理念，我们能看到很多相通之处。例如，城市中心的立交桥，如果设计得太简单，就不能承载多个方向的车流量，如果太复杂，又会让司机摸不着头脑。如何达到让各个方向的车辆不受路口上的红绿灯管制而快速通过的目的，又让司机能辨别好方向，安全地出行，就是立交桥的设计者们最关注的问题。软件设计也是一样，需要在简单和复杂的矛盾对立中找到一种平衡。怎么做到这一点？也许向这些艺术或者人造物体学习是一个好的办法。

还可以借鉴大自然的灵感。“天地有大美而不言,四时有明法而不议,万物有成理而不说”。 天地有最大的美，但人们无法用言语表达；一年四季有明确的规律，但人们从不议论；万物的存在和变化都遵循现成的规律，只是人们还没有发现而已。大自然有无穷无尽的妙不可言的地方，人们也可以从中总结很多规律。把大自然的美感赋予产品，把大自然运行的规律运用在产品上，或许会让产品提升到另一个境界。

##### 创造性地借鉴

借鉴和抄袭不一样。区别就是，抄袭得到的最多还是原来的设计，而借鉴得到的是一个新的设计。借鉴是一个中性词，抄袭是个贬义词，因为抄袭太无脑，也容易侵犯他人的知识产权。借鉴不当，就很容易和抄袭混为一谈。

所以，应该创造性地借鉴，而不是全盘照抄，把其他产品的设计强加于自己的产品之上。

##### 多个维度进行借鉴

如果自顶向下将整个产品设计拆解，可以得到产品框架、产品流程、产品细节三个维度。借鉴其他产品时，应该尝试从不同的维度进行思考，而不是眉毛胡子一把抓。

### 盲从用户需求

“用户希望系统能实现这个小功能，我们下周来实现它吧！”是不是听着有些耳熟？没错，这估计是开发者经常听到的来自产品经理的安排了。用户是软件的购买者和使用者，满足用户需求是团队赖以生存的根本，“客户至上”、“客户就是上帝”、“以用户为中心”这些理念在软件行业照样适用。

为用户创造价值、提供好的解决方案和服务的初衷是对的，但是很多团队中出现了盲从用户需求的现象。他们认为用户的说法都是对的，把用户表达的内容直接作为需求的内容，并且希望，所有用户需求都应该得到满足。他们认为把用户的需求叠加到一起后实现的产品形态，会真正让用户满意。

分析用户需求时，很容易陷进去，造成对用户需求的盲从。在消费互联网行业或者产业互联网行业，这种现象都屡见不鲜。

有一个关于设备管理的功能。产品刚开始运行时，设备的数量很少，管理起来也很方便，后来随着业务量的增加，设备也不可避免越来越多。大家就在想办法让设备的管理更加方便。最后想到的办法是给设备分组。先按照某种标准给设备逐级分好类，然后把相同类别的设备放到同一个组内。这种分组的做法让设备更好地支持各种批量操作，例如删除，更新等等，设备的管理比之前方便许多了。但后来，用户发现少量设备不太好归类，因为它们好像既可以归类到这个组，也可以归类到那个组。用户因为问题，给产品经理提出要让每个设备支持多个分组，这就可以解决不好分组的问题了。产品没有多想，答应了用户的要求。在三周的开发和测试后，这个功能终于上线了。用户以为设备的管理更容易了，然而事实却并不如此。设备和设备组之间的关系由多对一变成了多对多，用户发现，这种关系虽然解决了少量设备不好分类的问题，却让设备和设备组之间的关系更加复杂更加凌乱了。用户不得不让产品经理把这个新加的功能去除掉。

有一个转账的软件，里面包含了很多种不同的业务，例如计算利息、计算转账数额、转账、审计等等。每个业务都有专门的用户，也叫业务员。这几年间，负责审计的业务员一直在提出大量的需求，导致团队开发者的大部分时间都用在了审计上。审计的功能很完善了，计算利息、计算转账数额、转账这些功能却没能跟上，这种不对称导致了很多审计的功能都没用被使用，软件并没有因为审计功能的增加而变得更加有价值。负责审计的业务员开始抱怨为什么其他的功能都没有跟上，却没有意识到自己提的需求太多了。

最能体现产品价值的就是得到更多的用户以及获得用户的认可。很多产品经理为了实现这些目标，严格依照用户的需求来设计产品。最后，我们看到了这些结果：

用户需求太多，团队没有充足的资源去实现这些需求。虽然不是很确定这些需求一定有价值，但团队从上到下，每个成员都急匆匆地为完成这些需求而加班加点。这些需求都是在匆忙之中实现的，因此它们的质量打了折扣。另外，团队除了去完成这些用户需求外，没有时间找其他的机会或者做其他的事情。团队的资源被用户需求霸占了，有些真正有价值的需求，团队反而没时间去实现。

用户提出的需求导致了产品的过度设计。繁多的需求，如果产品经理不加甄别地全盘接收，并转化为产品，必然导致过度设计。生活中，过度的包装设计会给社会带来很多困扰，例如资源消耗、垃圾围城等。软件中的过度设计会带来不一样的困扰。繁多的功能，会让系统的复杂度上升，开发和运维叫苦不迭。此外，功能太多也往往会导致更多的性能问题。用户希望产品功能齐全，但同时又希望产品保持简洁性。毕竟这样的产品更方便使用。过度设计会让产品变得臃肿，而不是完备。

团队会被提出这些需求的用户所“绑架”。这些用户逐渐占据用户的大部分，而其他的用户比例会减少。企业陷在这些用户群中无法自拔。产品开始变得难以定位，失去了原来设计的初衷。企业也搞不清楚核心的用户是谁。核心的用户认为产品已经不是他们所希望看到的形态了，产品不能带来更多的价值，从而减少对产品的使用，甚至流失。团队的收益变少，产品失去了发展的前景。

无论是从用户的角度，还是从公司的角度，团队都应该避免盲从用户需求。作为一个产品经理，应该做好用户需求管理。具体有以下几个方面。

#### 挖掘

充分挖掘用户需求的过程中，需要注意以下问题：用户还有那些没有说完的需求？用户的潜台词是什么？如果我自己是用户，希望产品有怎样的形态和功能?用户需求是否反映了整个市场对产品的诉求？用户的表达应该如何提炼和归纳?

#### 分类

分析用户需求就是对用户和用户需求有更加深入的理解和掌握。在分析过程中，最主要的工作是对用户需求进行分类。分类的维度有很多种。

根据需求的性质来划分。有些需求是某种服务，比如说让开发者做一次用户使用培训。还有些需求是某个系统功能。

根据需求的受众多少来划分。如果某个需求只是某一个人或者少数人特定的要求，可以将其归类为特定需求。如果某一个需求有很多人提出，可以将其归类为公共需求。

根据需求内容来划分。有些需求是功能性的，有些需求是非功能性的。在电商产品中，添加购物车、支付就属于功能性需求，而性能、可维护性、可靠性就属于非功能性需求。

根据重要程度来划分。有些需求是用来定义产品的重要需求，还有些需求仅仅属于锦上添花。可以将需求分为核心需求和次要需求。

根据对企业的价值来划分。有些用户需求对企业有价值，有些对企业没有价值。可以将需求分为有效需求和无效需求。

分类的主要作用，是让产品经理知道如何进一步处理用户需求。分类不一定很容易。要判断一个需求是功能性的还是非功能性的是非常容易的，但是要判断一个需求对企业是否有价值就很难了。这需要对市场长期的研究和把握。

#### 过滤和完善

有些需求应该过滤掉，否则团队是没有那么多资源来处理它们的。一般来说如果某个用户需求不能给公司创造价值，就应该过滤掉这个需求，不应该把这个需求转化为产品。有些需求，即使用户很少，但如果它的实现能为公司带来价值，应该保留而不是过滤。

此外，还应该完善好用户需求，这有利于将用户需求转化为产品需求，也有利于之后进行需求复盘。但为了节省时间，使用简单明了的语言即可，不用太多的修饰。

#### 转化

产品需求是写在产品需求文档中的需求，用来描述产品是什么，做什么，并指导开发者将需求转化为产品。用户需求和产品需求显然有着很大的区别，产品经理的一大工作就是将用户需求转化为产品需求。这其中包括了三个步骤。

首先，将用户需求转化为能指导开发者开发的语言，也就是产品需求。比如，用户需求是“系统提供下载图片的功能”，产品需求应该这样描述：用户可以在某页面上选定某一些图片（一次次最多选中10个图片），点击右侧导出按钮之后，会以同步的方式导出这些图片，导出的格式为zip文件。Zip文件中，有两级文件夹，这些文件夹的命名规则为XXX，图片根据所属地区放在不同文件夹内。图片的命名格式为XXX。

然后要将产品需求结构化。所谓结构化，是指将逐渐积累起来的产品需求加以归纳和整理，使之条理化、纲领化，做到纲举目张。很多产品经理用思维导图将需求组织起来，呈现给开发者，这就是一种结构化的方法。

此外，要给产品需求划分优先级。可以用P0，P1，P2这三个等级来标注需求的优先级。P0的优先级最高，是必须要在这个版本中实现的。P1表示中等，P2代表优先级最低。P1和P2都不是在本版中必须实现的需求。

最后，要用各种表现形式将产品需求呈现出来。文字是必要的，各种图表以及原型图也很重要。有了这些表现形式，开发者才能更快更准确地理解需求并投入到开发工作中。

#### 管理好用户

既要管理好需求，也要管理好用户。当用户需求得不到100%的满足时，会有一些用户感到心理上有落差感。因此，应该用一些方法去提高用户的满意度，这样才能维持长久的合作关系。首先要让用户理解，为什么他的需求不能被满足。是这个需求太小众，还是团队人数太少，还是实现有难度？总之，应该让用户觉得，不是团队不重视用户，而是这个用户需求提的不合理。

另外，还要学会管理用户的心理预期。如果用户刚开始就认为他们的需求对团队来说就是命令，一旦这些命令没有被执行，用户必然心中不愉快。因此，不能让用户的心理预期过高。

如果实在难以达成用户的愿望，要看看能否找到一个替代方案，或者折中方案，提供给用户。

## 技术篇

### 技术思维的惯性

技术思维有很多种解释，比如将整体抽象的理论原理转化为具体的技术系统的思维。而在本文中的技术思维指的是开发者在工作中用技术去解决所有问题的思维。一旦有了技术能力，我们发现自己能解决的问题的范围变得宽阔很多了，久而久之我们会有技术思维的惯性。

图片图a 一个典型的大数据技术栈

技术思维不是银弹，它能解决技术问题，却不能解决所有问题。开发人员在日常工作中要解决的问题不光有技术性的，也有非技术性的。想用技术思维去解决非技术的问题是很难的，而且往往带来不好的效果。尽管很多开发者意识到了这一点，但是技术思维的惯性却普遍存在：

需求评审中，产品经理让大家评审需求是否满足客户需求，是否合理。开发者却只去思考和提出这些功能用什么去实现，怎么去实现的问题。

项目经理让开发者提供产品用户说明书，开发者却将产品说明书写成了技术文档。项目经理和用户直呼看不懂。当用户针对文档提出问题时，开发者却觉得用户应该能独立地看懂这些文档。

开发者疲于炫技，只考虑代码能不能跑通并实现需求，不考虑代码的可读性。代码中没有必要的注释。

开发者只注意提升自己的技术水平，而忽略业务水平和沟通能力。很多开发者算法能力强，技术能力超群，却缺少业务知识的积淀。

一个简单的修改流程就可以解决的问题，非要用技术的方式解决。最后花费了时间不说，问题也变得更加复杂了。

开发者的一大特点，就是既要和机器打交道，也要和人打交道。他们写的代码，既有机器阅读，也有人去阅读。很多人在日常工作中，逐渐形成了与机器打交道的思维模式和做事习惯。这会让我们陷入一个误区，那就是单方面考虑问题，不能用联系和辩证的眼光看待问题。就像造一辆车，如果用单一的眼光去思考，就是如何让车跑得快。车跑太快了，却会引发其他很多方面的问题，比如成本、安全等。在造车的过程中，应该把所有的因素都考虑进来，权衡利弊。否则我解决的只是如何让车跑得快的问题，而不是造车的问题。

图片

图b 开发者既需要和机器打交道，也要和人打交道

软件开发的主要矛盾，是人类在生活工作中对软件日益增长的需求同系统设计和开发能力之间的矛盾。要解决这种矛盾，需要开发者提升的是系统设计和开发的能力，而不是仅仅是技术能力本身。开发者应该培养正确的思维方法，它们是软件开发活动得以成功的前提。

系统思维

系统思维就是人们运用系统观点，把对象的互相联系的各个方面及其结构和功能进行系统认识的一种思维方法。整体性原则是系统思维方式的核心。简单地说，系统思维就是把某个功能，某个需求，某个行为，某个状况当作系统的一部分来思考，不孤立地看待这些问题。

软件开发中，衡量某个产品的功能是否有价值，衡量某件事情的意义，衡量个人对团队的贡献，都要看这些因素对于系统的影响。

软件开发是一项系统工程，必须全面地、系统化地思考软件开发活动。如果只考虑软件开发的某一个阶段，不光会延误软件的开发进程，也会影响软件的质量。

抽象化思维

抽象化是指以缩减一个概念或是一个现象的资讯含量来将其广义化的过程，主要是为了只保存和特定目的有关的资讯。例如，无论什么品牌的手机，诺基亚，小米，还是苹果都可以被抽象成手机，只保留手机的属性和行为等信息。抽象化主要是为了使复杂度降低，以得到较简单的概念，好让人们能够控制其过程或以综观的角度来了解许多特定的事物。

软件开发是一门复杂度较高的工作。合理地把控住软件的复杂度，然后改造软件是开发人员的主要工作。抽象化能帮助开发人员管控复杂度。软件系统、硬件系统都很庞大，其中必然有很多开发者的脑力无法完全掌握，也没必要完全掌握的概念。如果把心思花在了这些细节上，开发者就没时间去应对要做更新的领域了。这里的破解之道，就是让开发者抽象化不需要了解细节的领域。

抽象化的方法应该贯彻在开发的方方面面中。不光在认识系统的时候需要抽象化思维，在编写代码的时候也需要抽象化思维。在定义一个类的时候，实际上就是把一类事物的公有的属性和行为提取出来，形成一个物理模型，这种研究问题的方法称为抽象。很多代码最后出现的问题，本质上就是未能很好地将概念抽象化。

图片

图c 抽象化过程

业务思维

业务思维，包括了了解行业信息、学习业务知识、培养业务素养这些内容。掌握了业务，就掌握了团队赖以存在的生命线，就知道了事情该怎么做。业务思维，就是把业务素养内化到开发者的心智中，让它去推动软件构建进程的思维。

业务和技术都是开发的关键因素。不同的行业，这两者之间的关系有着区别。在银行业，业务是技术的主导。在AI行业，技术是业务的主导。一个只懂业务但不懂技术的人在团队可能会成为业务分析师，但团队里却很少需要只懂技术不懂业务的人，除非这个人是新手。培养业务能力，需要广大的开发者长期有效地积累。

技术脱离现实的问题后，其价值等于0。技术本身没有社会价值。让技术创造社会价值，或者说简单点，让技术能解决实际问题是开发者们应该矢志不渝追求的目标。开发者应该更多思考需要做什么，而不是想要做什么。在思考该不该做，或者如何做的时候，应该从全局出发，而不是只从技术的角度出发。

### 失败的架构

成功的架构往往操作简单，运作稳定，能满足用户的需求，也能灵活地演进。有很多架构，在某些方面做的不错，可是总是有短板，不能达到成功架构的所有的标准。很多人认为在做架构的时候，只要选对了大方向，就能设计出成功的架构，却总是事与愿违。

失败的架构

一个工程包（例如war格式或者Jar格式）包含了软件所有功能的应用程序，我们称之为单体应用。这种架构模式被称之为单体应用架构。这也是传统的应用架构。单体应用可以作为一个整体，统一开发、编译和部署。单体应用在变得庞大的时候容易出现下列问题。

艰难的版本控制

单体应用中，项目的所有的代码使用同一个代码仓库进行管理。在多人参与开发的场景下，所有人将代码的改动上传到同一个仓库。每个人负责项目中的某个或者某几个功能模块，每次提交的都是同一个仓库。因此，同一个仓库将可能有很多个提交，很多个分支。这种情况下，该仓库的版本控制复杂度很高。复杂度的第一个体现是，合并代码容易出现代码冲突。第二个体现是，部署的代码不一定包含每个人最新的版本。

可维护性差

如果应用过于庞大，单体应用的可维护性会比较低。高可维护性意味着开发人员能方便地修改代码。但是如果一个项目里面的代码量过大，开发人员可能会迷失方向，如果想找到要更新的地方，会有较大的难度。代码并没有很好的分类，开发者为了找到要更新的代码，往往要跳转多次。而开发者的脑力是有限的，当跳转的次数太多时，开发者会忘记之前遍历过的代码，进而重复地遍历一遍。这样，开发者既效率低下，又没有成就感。开发者对应用和代码的兴趣会大大降低。应用会变得越来越难以运维。

资源无法隔离

单体应用被部署之后，要么是运行状态，要么是停止状态。一个内存溢出，一次CPU爆满，就能让应用停止。因为部署在一个服务中，各个功能之间是强依赖的。当处于运行状态时，不能保证所有的功能都正常运行。当处于停止状态时，可以确定的是所有功能都不能正常运行。资源没有有效地隔离，一旦系统停止，所有的资源都不可访问，这会让所有使用该应用的客户都崩溃。

微服务就是一些协同工作的小而自治的服务。有很多公司或者技术团队大力推广微服务，把所有的项目都换成了微服务架构，希望这种架构能解决使用单体应用架构的过程中面临的一系列问题。庞大的单体应用会有很复杂，微服务的出现就是因为原来单体应用架构已经无法满足当前互联网产品的技术需求。微服务的初衷是让我们能更好地认识系统，修改系统和管理系统，并且让系统同时满足成本需求和用户需求。微服务不是银弹，错误地使用微服务会让我们违背微服务的初衷。

服务太小

服务分的越小，微服务架构的优点和缺点就会越明显。如果对微服务中的“微”概念走得太远，我们很容易发现自己的开销和复杂性超过了微服务体系结构的整体收益。服务太小时，意味着服务更多。这也就意味着，很多时候，我们只需要操一份心，用微服务的时候，我们却要操几份心。服务拆分后，有些任务的工作量会减小，例如代码的版本控制。而有些任务的工作量会增加，比如打包代码。

减小的工作量和增加的工作量之差，就是微服务给我们带来的利好。差为正数，说明微服务简化了软件开发。差为负数，说明微服务复杂了软件开发。让减小的工作量和增加的工作量之差最大，就是我们的目标。显然，当服务个数太多的时候，这个差值不会是最大的，甚至可能是负数。因此，在微服务开发中，应该避免服务太小。

图片

缺少管理服务的工具

每个服务都像一个孩子，需要认真呵护。孩子的数量太多时，就容易出现管理不好的问题。微服务架构中服务数量不可避免地比较多，纯粹用人工来管理，简直是开发和运维的噩梦。因此，微服务体系中，需要引入一些服务管理工具，比如kubernetes，prometheus等。

找不到限界上下文

划分微服务，首先要找到微服务的边界，即限界上下文。如果对系统的业务或者技术理解程度不够深入，对代码不够熟悉，很容易让我们迷失方向，难以找到合适的边界来划分微服务。如果边界没有找到边界，我们就无法构筑高内聚低耦合的服务。这对于服务的修改和扩展都不利。

对架构进行评价

如果固执地认为微服务架构比单一架构好，我们会滥用微服务。微服务和单一架构的关系就像汽车和火车一样，好不好关键在于什么场景，在于我们要把系统打造成什么形态。所以，选择什么架构，要看架构是否能与那些场景良好地匹配。如果发现周围大部分人用微服务，原因之一是因为微服务能适应更多的场景。原因之二是很多人具有从众心理，盲目地认为微服务更好。为了更好地把控风险，应该时刻去评估架构是否符合需要。

微服务的最佳实践，就是尽量地扬长避短。我们应该对架构进行评价和反思，方法是要是系统目前地架构形态是否能够真正达到扬长避短。既要对每个服务进行评价，也要对系统整体进行评价。既要评价已经实现的部分，也要总结需要提升的方面。

对系统进行评价的角度有成本、可扩展性、可维护性、安全性、稳定性、高性能、稳定性等等。

没有一劳永逸的架构

大部分架构在设计之初都是经不起检验的。如果系统足够复杂，并且架构设计者对系统很陌生，架构失败的可能性会更大。架构设计的时间点都处于系统设计的早期，设计者对于系统的将来的形态并不是很清楚，对于用户的理解也不够到位，甚至市场上可能找不到类似的系统可以学习。这些主客观因素会导致设计者在作决策的时候出现失误，即使这些决策对于系统的发展至关重要。

市场和用户不断在改变，技术也在不断前进。这种运动和变化决定了架构不会是一劳永逸的。如果一个系统的架构在若干年都保持不变，原因不会是这个架构太好了，而是改善它的代价太大了。这样的系统会在受到多方的埋怨之后被弃用。

架构的演进

自然界的生物需要不断进化才能适应环境，架构也需要不断演进才能适应软发展的需要。在某个时间点，架构、代码、技术、需求这些要素互依互存，相对平衡。其他的要素会随着时间的推移而改变，打破这种状态。架构也会不断地演进，使得系统重新回到平衡的状态。

### 过分依赖第三方框架

目前，开源软件越来越多。第三方开源框架名目繁多，层出不穷，另外第三方应用平台也多如牛毛。这通常是一件好事。许多的框架不仅很好用，而且是免费的，这极大地提升了软件开发的效率。

第三方框架的作者具有较强的奉献精神。他们希望自己能用自己的软件开发能力为社区贡献自己的力量。他们希望自己的作品尽可能地普及化。当他们的软件下载量增大时，他们会成就感飙升。这种普及化带来的问题可能就是软件缺少定制化。就像做饭给一百分人吃的难度肯定比做饭给一个人吃的难度大很多，因为众口难调。所以第三方框架往往没有自己开发的软件那么适合自己的应用场景。另外，框架作者写框架，最开始是因为他们自己遇到了问题。他们想用框架解决这个问题。最后发现这个框架不光能解决自己的问题，还能解决别人的问题，于是把框架开源出来供大家使用。而实际情况可能是，每个人遇到的问题不尽相同。框架的对每个人的意义也不一样。

框架作者希望我们更多地依赖框架，这种依赖意味着难以摆脱，也意味着我们使用框架的时间更长，这能体现框架的强大生命力。但是框架毕竟不是我们自己开发的程序，对于框架的很多的权力还掌握在作者的手里，这种关系决定着使用框架具有不确定性。2021年11月24日，阿里云安全团队向Apache官方报告了Apache Log4j2远程代码执行漏洞。由于Apache Log4j2的一些功能存在递归解析功能，攻击者可直接构造恶意请求，触发远程代码执行漏洞。这个案例说明了使用第三方框架也有一定的风险。由于框架是免费的，因此框架作者除了背了框架不好的坏名声，不会为风险承担其他责任。因此这种风险是单向的。

我想到了这些方面的风险：

有些框架加大了开发的难度。为了完成开发，开发者将大量的时间花在学习框架的API上面，而不是编写业务代码上。要更新代码的时候也是如此，首先要搞清楚框架的各种用法。框架如果不能简化开发，就只能用其他的优势去弥补这一缺点。另外，有些框架的配置非常繁琐。总之，有可能框架给我们带来的好处不能与其给我们带来的精力损耗相抵。

有些框架对系统有较强的侵入性，这使得我们一旦使用了该框架之后，就再也无法摆脱这个框架了。就像生物入侵一样，一旦他们占据了地盘，就会无休止地繁殖，最后所有的地盘都被他们占据了。这种侵入性会导致我们无法继续自由地构建系统，影响系统的可扩展性。



生物入侵

框架可能有安全漏洞或者性能问题。比如我们所熟知的基于java的轻量级的ORM开发框架Hibernate。有一次，我遇到系统出了性能问题，最后发现这是Hibernate没有批量运行SQL导致的。

框架可能在未来没人维护。有些框架是一个团队整体完成的，团队会有精力去维护这个框架。还有的框架是个人完成的。他们心血来潮写了易用的框架，但是之后可能因为各种原因，没有时间去维护它了。框架就失去了保障。

有些领域框架演变和淘汰得太快。有些框架一年前可能很火热，几年之后就少有人问津了，比如前端框架jQuery，Java的structs。这既体现了这一领域的快速发展，也说明这些框架太不稳定了。当有更好的框架出现时，要用新的框架代替旧的框架，这是对系统的大手术，有可能引发不兼容的问题。这不仅耗时费力，而且风险很大。

框架的使用是不可避免的，但不能过分地依赖第三方框架。在做决策时，应该更加周全地考虑该不该用，用什么，如何用。框架可能很重，如果非用不可，可以考虑使用框架的一部分功能而不是全部功能，不要被框架绑架。

### 滥用语言特性

李文塔在《Go语言核心编程》中对编程语言的特性总结如下：

语言组织：（程序的语法格式千差万别，但表达这些语法的基本概念大同小异）

· 标识符和关键字

· 变量和常量

· 运算符

· 表达式

· 简单语句

· 控制结构

类型系统：

· 动静特性：动态语言还是静态语言

· 类型强弱：强类型还是弱类型

· 基本数据类型：包括类型及其支持的运算和操作集合

· 自定义数据类型：包括类型及其支持的运算和操作集合

抽象特性：

· 函数：是否支持函数、匿名函数、高阶函数、闭包等

· 面向对象：是否支持面向对象

· 多态：如何支持多态

· 接口：是否支持接口，以及接口实现模式

元编程特性：

· 泛型：是否支持泛型

· 反射：是否支持反射，反射的能力

运行和跨平台语言特性：

· 编译模式：是编译成可执行程序，还是编译成中间代码，还是解释器解释执行

· 运行模式：直接由OS加载执行，还是由虚拟机加载执行

· 内存管理：是否支持垃圾回收

· 并发支持：是否原生支持并发，还是库支持

· 交叉编译：是否支持交叉编译

· 跨平台支持：是否支持多个平台

语言软实力特性：

· 库：标准库和第三方库是否丰富、好用、高效

· 框架：是否有非常出众的框架

· 语言自身兼容性：语言规范是否经常变换，语言新版本向前兼容性

· 语言影响力：是否有商业公司支持，社区的活跃性，是否是著名项目

一门编程语言之所以和其他编程语言有区别，就在于其独特的编程特性。例如，Java与C/C++，它们在很多方面有着相似性，在某些方面也有着很多的区别。

Java开发语言与C语言C++的7大区别明细

|  |  |  |
| --- | --- | --- |
| **Java** | **C** | **C++** |
| 面向对象 | 面向过程 | 面向对象 |
| 运行在虚拟机上 | 直接编译成可执行文件 | 直接编译成可执行文件 |
| 效率相对低 | 效率高 | 效率高 |
| 不考虑内存管理和垃圾回收机制 | 有内存泄漏难题 | 内存泄漏 |
| 通过根类object将容器和算法分离，提高代码重用 |  | 通过模板提高代码重用 |
| 内建了丰富的数据结构：列表，集合等 |  | 用“模板”同样提供了各种数据结构 |
| 流行的开发语言 | 比较低层，和硬件打交道的地方用得比较多 | 功能很强大 |

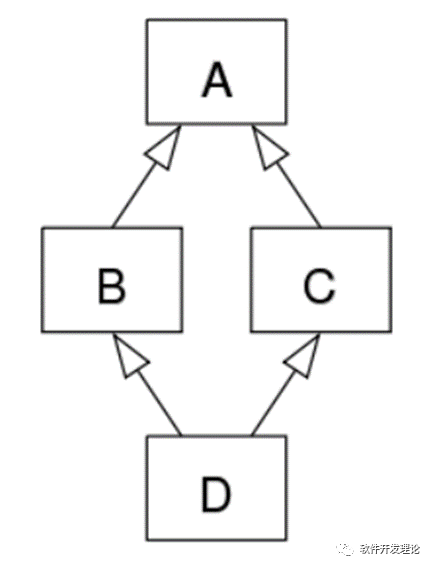
每种编程语言都是不完美的。哪怕是被称为世界上最好的语言的PHP也有一些不足，不然其他的语言存在的意义何在呢？如果说未来的某一天，哪个团队想到把当前各种编程语言的优秀特性集中到一起，设计出一种最好的语言，这恐怕是做不到的，因为很多指标是矛盾的，比如，是否支持系统自动垃圾回收和自动内存管理。C语言让程序员自己管理内存，这会让程序员把更多的精力花在内存管理上，同时也让他们对内存有了更好的把控。Java让系统自己管理内存，这给程序员减少了很多工作量，但程序员也无法更好地控制内存了。完美的语言是否应该支持系统自动垃圾回收和自动内存管理呢?这个问题很难解决。因此，完美的编程语言从理论上来说是不存在的。如果未来某个团队声称他们设计除了完美的编程语言，很容易反驳他们：既使这门语言在特性上是完美的，但是它出现得也太晚了，所以，这门语言算不上完美。

每种编程语言都有一些特性，这些特性在某些方面可能是加分项，而在另一些方面会是减分项。程序员应该识别出这些因素，避免滥用语言特性。事实上，滥用的现象屡见不鲜。

很多Java程序员喜欢使用泛型。泛型的英文是 generic，中文意思是通用的，泛型就是一种通用类型。我们一般指泛型都是指其实现方式，也就是将类型参数化。泛型适用于多种数据类型执行相同的代码。泛型和其他很多的语言特性一样，不是只有好处。是否使用泛型需要权衡很多因素。滥用泛型就会造成诸多问题，例如让代码的可读性变差、降低编译速度和执行效率等。我在工作中也用到了泛型。当时为了简化开发，我就用泛型来处理不同地区的物流信息，最后的代码运行没有问题，但是给外界暴露的API接口文档中，全都是<object>,而不是具体的类型。这下可麻烦了，用户无法知道物流信息具体的细节。最后，我把泛型都去除了才解决了问题。最近比较流行的Golang，也不支持泛型。在将来，Golang可能会支持泛型，不过Golang团队认为加入泛型并不紧急，更重要的是完善运行时机制，包括调度器、垃圾收集器等功能。

有的编程语言原生支持并发，还有的语言需要通过构建一些类或者调用库来支持并发。实际上，并发也不能滥用。并发的目标是提高程序运行的效率。但很多时候，并发对效率的提升并没有太多的帮助，如果程序的瓶颈不在没有使用并发上，而在其他方面，比如IO上，使用并发不会对提升性能。另外，并发的运用降低了程序的可读性，也提高了程序出错误的概率。以Java为例，如果没有处理好线程安全问题，或者没有管理好程序的生命周期，程序的稳定运行得不到保障，数据的安全性也得不到保护。

继承是很多编程语言中的重要特性。继承是面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。不过继承也容易被滥用，有些人甚至觉得，继承带来的好处远远比坏处多。好的代码是高内聚低耦合的，而继承增加了类之间的耦合度，大量地使用继承会让类和类之间关系变得很复杂，如果继承的链路太长，分支太多，会让类形成网状的结构，这会让开发者摸不清头脑。另外，如果继承没有被正确地设计，容易让代码违反里氏替换原则。子类继承了父类，但父类中有些属性和方法不是子类所需要的，这会影响代码的逻辑性和精简性。还有的语言，例如C++支持多继承，不恰当地使用这一特性会增加代码的歧义，还有可能引发多继承菱形的问题。



多继承菱形问题

日常写代码中，应该避免滥用语言特性的情况。在学习某种语言的时候，不仅仅要学习某些特性，还要学习其中的设计哲学和编程思想。开发者应该看到每种语言的优势和不足，尽量去扬长避短，避免踩不必要的坑。

### 版本控制的常见问题

版本控制是指对软件开发过程中各种程序代码、配置文件及说明文档等文件变更的管理。版本控制的作用主要有两个。一是保留各个提交版本的信息，让成员可以对它们进行搜索。二是方便不同的人之间的合作。

如果版本控制做的不好，也会造成很多麻烦。在这里罗列了版本控制中出现的常见问题以及解决方案。

#### 数据丢失

版本控制加强了代码的管理，在很大程度上减少了数据丢失的可能性。有些不当的操作还是有可能让数据丢失。例如，以下操作可以永久删除未提交给git的数据：

*git rm -f - 可以删除尚未签入的文件*

*git reset --hard - 将删除尚未签入git的更改*

*git clean -f - 将删除git未跟踪的文件*

*git checkout /path/to/file - 可以将未签入的更改还原为git*

*git checkout <rev> -f - 可以覆盖未签入git的更改*

*rm -rf .git- 不要删除你的.git目录!这就是存储您所有本地历史的内容*

解决这些问题的方法是提前备份好要删除的数据，或者在执行操作的时候，保证那些将被删除的数据都是没有意义的。

代码提交了之后也可能造成数据丢失。比如，在解决代码冲突时，有可能会误删有价值的代码，而保留了没有价值的代码，这会导致一些线上问题的产生。因此，在解决冲突的时候，应该更加慎重和仔细，每一个冲突都应该考虑清楚到底删除哪些代码，保留哪些代码。此外，如果对代码还不够熟悉的话，尽量不要去改公共的代码，因为改动这些代码所造成的影响会更大。这些重要的代码应该让更有经验的人去操作。

#### 分支

分支可以跟踪并行演进，多条分支可以用来记录不同的版本演化轨迹。很多程序员不喜欢分支之间的切换，在该创建新分支的时候没有去做这件事，把所有的功能代码都放在同一个分支中。这会造成这个主分支太过于繁忙，既要管理这个功能模块，也要管理那个功能模块，既要处理这个区域的业务，也要处理那个区域的业务，所有的版本都交织在一起。无论是代码层面还是系统运行层面，都容易出现问题。

创建了多个分支后，代码同步和代码合并的频率太低会妨碍持续集成和持续部署。特定环境中的每次部署都只能是唯一的分支，这个分支应该包含所有应该部署的代码。所以，该做合并的时候要做好合并，该做同步的时候要做好同步，不要因为太忙或者麻烦忽略了这些事情。

#### 忽略标签

很多版本控制系统，例如VCS，Git都可以给某次提交打上一个标签，用来标记某个重要的节点。很多人忽略了标签的使用，因为他们认为，分支就够用了。某个分支中的代码可能一直在更新，而标签记录下来的代码版本是静止的，所以分支是不能取代标签的。在项目达到某一个里程碑时，用标签记录下当前仓库的视图很有意义，它极大地方便了以后的追溯和恢复。

#### 提交不规范

提交频率太低是第一个不规范问题。如果开发者一个月提交一次代码，

每个版本控制系统都有写提交语句的功能。很多人在编写提交语句时习惯于应付，并没有写出有价值的说明，或者说明不能代表这次提交的内容。在项目的代码量和提交量都很少时，问题不会很大，但代码量和提交量都很大时，提交语句就会显得很混乱。本质上，提交语句就是对所有的提交进行分类的一种手段，比如这次提交实现了什么功能，下次提交又实现了什么功能。在之后的追溯过程中，可以通过这些功能对提交进行过滤。如果分类不够完善，过滤和追溯也将变得困难。

应该制定一些关于提交语句的规范。代码审核应该包含对提交语句的审核。这虽然不难，却很有意义。提交语句应该足够具体，一目了然，而不是很抽象很宽泛。比如，“修复订单因为网络问题不能撤销的bug”就比“修复订单bug”要好。另外，要对提交语句进行分类，一种常见的分类方法是，按照feature、fix、docs、style、refactor、test、chore进行分类。这些分类的具体含义如下：

*feat:（用户的新功能，而不是构建脚本的新功能）*

*fix: (针对用户的错误修复，而不是对构建脚本的修复)*

*docs：（对文档的更改）*

*style:（格式化，缺少分号等；没有生产代码更改）*

*refactor：（重构生产代码，例如重命名变量）*

*test:（添加缺失的测试，重构测试；没有生产代码更改）*

*chore:（更新 grunt 任务等；没有生产代码更改）*

为了提交提交语句的规范性，可以使用钩子来限制提交的过程。Git能在特定的重要动作发生前触发自定义脚本，这个触发过程叫做钩子。有两组这样的钩子，他们分别是：服务端钩子和客户端钩子。可以在服务器端添加这样一个钩子：当用户将代码push到远程服务器（将代码push到远程服务器就是一个特定的重要动作）前，需要触发一个脚本，脚本的内容是对用户的提交语句进行正则表达式的校验。只有当提交语句满足某一范式时，代码才能push到远程服务器。

#### 仓库中只有源代码

许多年前，我经历过这样一次事故。一个新加入团队的伙伴想给ElasticSearch添加一个字段。可是不巧的是，他输入的命令有问题，把ElasticSearch的某个重要的索引给删掉了。索引的字段有七十多个，里面的内容是关键的物流数据，一下子全没了。惊慌之中，我们只得快速地将index给建立起来。但是我们手头没有初始化ElasticSearch索引的语句，后来我想到了我们的物流项目的代码仓库中有这些语句。我们赶紧把它们执行了一遍，以为系统可以成功地运行起来。

事实并没有这么顺利。系统还是报错，经过仔细排查，我们发现代码仓库中创建索引的语句是九个月以前的版本，根本就不正确，有很多字段存在缺失。我们打算找运维帮忙，因为运维那里有最新的部署文件，可是运维正在休假中……我们只能仔细地对一对所有的字段。原本几分钟就可以解决的事情，我们花了三小时才搞定，那一天我们加班到了深夜。

我们把初始化索引的语句放入了代码仓库中，进行版本控制，但我们并没有认真地对待这件事，以致于这些语句在几个月的时间里都没有更新，这跟没做版本控制没有太大的区别。假如当初把这些初始化语句看得跟代码一样重要，我们就不会碰到这么多问题了。

仓库中应该包含开发用得到的所有内容。仓库中除了源代码，还应该包含各种测试脚本、配置文件，各种文档，项目中使用的图片、流程图等等。那些在网上可以下载的工具或者框架，比如Tomcat，Maven，Spring源码不应该放到我们的仓库中，因为它们不需要我们来管理。我们只是使用它们，也不用担心以后找不到它们，因为它们有自己的官网做版本控制。

应该慎重对待仓库中的各种内容，及时更新，保证内容的正确性和权威性。这也是为了方便大家的使用的时候更加方便。

### 技术研发的浮躁现象

技术研发推动着社会和经济的发展。同时，我们在这过程中也看到一些浮躁的风气。这种风气不仅影响了开发者的开发进程，也影响了广大用户的利益。我们注意到了这些浮躁的现象：

##### 夸大宣传

技术官网是对某种技术的官方的宣传和描述。在做技术选型的时候，大家往往会根据技术的官网去分析技术的优劣，并作出选择。在学习某门技术的时候，大家也会去参考技术文档。广大开发者对技术官网的认可度是很高的，因为它是权威且专业的。

但是有些技术官网上面尽是一些对技术本身“夸张”的正面描述。这些描述就像一些减肥广告一样，甚至有过之无不及。呈现在用户面前的，全都是某某技术的优点，例如“高性能”、“高可用”、“可扩展性强”等等。用户想看到的是对技术客观的描述，而不是广告连篇，因为广告难以指导用户怎么使用技术，告诉用户该技术有什么坑。一般产品的广告，很容易让用户区分出来它就是广告，因此用户不会轻易被广告误导，但是技术官网不一样。当广告伪装成官方文档时，用户就很难区分它是广告还是客观描述了。最好的解决办法就是用户自己去测试技文档是否言过其实，不过测试是需要成本的。

有些公司生产相同类型的产品，他们都宣称自己是行业第一，这就造成了一个产品多个第一的笑话。公司也很无奈。谁记得住行业第二的产品呢?谁又愿意承认自己的产品比别人差呢？

##### 面对新事物的浮躁

2021年，在美国华盛顿的一次演讲中，刘慈欣表示：“元宇宙”是人类文明的一次内卷，最后会引导人类走向死路。元宇宙是2021年流行起来的跟技术相关的新概念，它指的是一个聚焦于社交链接的3D虚拟世界之网络，主要探讨一个持久化和去中心化的在线三维虚拟环境。

刘慈欣的观点体现了他对元宇宙存在的质疑，同时也反映了他在面对新概念和新技术时的理性态度。技术能改变世界，颠覆人类，正是因为如此，有很多人，甚至是做技术的人，面对新概念和新技术时，显得很浮躁，具体表现为，当新事物流行时，由于从众心理，也开始鼓吹，但由于缺乏研究和评估，他们对新事物的理解程度不够，这说明了这种对新事物的鼓吹是盲目的或者是缺乏根据的。

##### 难以兑现的承诺

在做技术调优时，很多开发者被问到能将性能提升多少时，他们会说一个很大的数字，例如五倍、十倍。这个回答会让老板们非常满意。但在测试中，这些数字往往难以达到，甚至远远达不到。要么是开发者对性能调优的理解还不够到位，要么是开发者仅仅想达到让领导们满意的目的，总之，这些数字承诺难以兑现。

这种情况也很常见。在项目开始之前，部门会对项目进行各种估算，例如成本、人力、交付时间。但因为各种原因，例如进行估算的人对项目的各项任务没有清晰的认识，这种估算往往是不正确的。而且之后，也没人愿意调整这个估算，这就造成了开发任务不能按时按量按质完成的问题。

##### 应付交差

应付交差的问题也很普遍。很多开发者给自己定了一个较高的目标，最后发现自己的任务完成不了了，只能应付交差，草草了事。很多创新项目，刚开始做的轰轰烈烈，越到后面越难以深入推进，最后项目成了名副其实的“烂尾楼”。

还有些开发者习惯了临时抱佛脚的工作方式。项目刚开始进行时，不紧不慢。到要验收时，赶紧加快进度。项目的质量大打折扣。

还有的项目，为了快点出成果、多出成果，绕开了真正有价值的解决方案，而选择了一些折衷方案，或者一些现成的但不怎么好的方案。许多团队为了KPI，不愿去做高风险、出成果慢的事情，而是去追求短期利益，用了更容易实现的办法。

#### 对策

##### 合理引导

无论是国家和政府层面，还是公司和团队层面，都可以引导一种踏实的风气，拒绝浮躁。工程师文化的精髓，就是乐于创新，甘于坐冷板凳，不急于求成。对良好的风气加以引导，杜绝不好的风气和习惯，带来的是长期的收获。

##### 避免过度自信

过度自信指的是，人们往往通过观察自身行为的结果来了解自己的能力，在这个过程中存在着一个自我归因偏差，即人们在回顾自已过去的成功时，会高估自己的成功，相比那些与失败有关的信息，人们更容易回忆起与成功有关的信息。简而言之，过度自信就是太过于自信了。

开发者应该克服过度自信的毛病。应该提升自己的认知水平，知道自己对哪些方面认识得很充分，对哪些领域还有不够熟悉。在做决策时，应该让自己的思考慢下来，充分考虑到事情的后果，而不是做拍脑袋的方案。如果某些工作是高风险的，例如金融转账类的开发，开发者更应该注意到在筹划任何事情时，要有最坏的打算。任何关键点都有可能出问题，任何问题都有可能造成损失，所以要用怀疑的态度看待一切。

##### 多一些评估性工作

对于新技术和新概念，开发者应该多做一些评估。只有做好了评估，才能认识到这些新事物的长处和短处，才能在开发过程中避免不必要的问题。在了解一门技术的时候，不仅要看那些宣传性的文章，也要读一些利益无关者写的文章。这些文章能让我们用冷静的视角，更加全面的审视这些技术。

对于任务和开发者自身，也需要充分的评估。当发现之前的评估有误时，应该敢于纠正，避免在未来出现更大的问题。

### 缺少自动化工具

软件产品的出现大大地简化了人们的生活和工作。很多需要线下完成的工作，在线上也可以完成了。很多软件开发者只关注给用户提供售卖项，却忽视了用什么样的工具能让自己的开发也加快。

很多开发团队还在大量地使用Excel来处理业务。其流程大概如下：首先，从数据库或者页面把数据复制到Excel，之后对数据进行一系列修改、查询、复制、删除、对比、校验等操作。最后得出结果，并将结果传到下游进行处理。

20年前，Excel是软件中的领军者。“Excel可能是有史以来最有影响力的软件，在SaaS出现之前，Excel就是每个行业都依赖的软件领域的思想自行车”。SaaS正在逐渐地去Excel化，人们把需要执行的功能写成代码，从而避免使用Excel。这种去Excel化在本质上就是自动化。大量地使用Excel，说明团队的自动化程度还不够。

有些团队每周都需要执行很多复杂而重复的手动流程。有些流程是针对数据库的，比如查找一些数据之后写入文件中。还有些是针对服务器的，比如，拉取一些表示服务器中各个服务状态的信息并发送给相关责任人。开发者很想逃避这项任务，但是在自动化实现之前，他们没有其他的选择。

自动化工具指的是软件构建过程中使用的用来替代手动流程并提高开发或者运维效率的软件。这些工具无法用来售卖，不会写到需求文档中，却是重要的生产资料，他们可以用来提高生产力，缩短软件发布的时间。但是很多团队却没有足够地重视自动化工具。有很多因素阻碍了其创建和使用。

没有时间和资源开发工具

开发者会被规定在某一个时间点之前完成某项开发任务，这些任务通常都是用来服务客户的。当日常时间被排满时，开发者就没时间去思考和开发自动化工具了。产品经理在验收产品的时候，只会看开发者有没有完成需求。即使开发者创建了工具，对他个人的绩效也没有太大的帮助。这种以需求为导向的开发工作会让大家只顾完成被安排好的事情。

有些团队有一定的时间去开发工具，还是没有设计出好的作品。团队没有太重视这件事情，没有认真分析自动化相关的需求，也没有相关的测试来验证工具的价值。除了开发者的有限的时间，团队没有为这件事情提供更多的资源。工具的设计成了开发者个人的任务，而不是团队的任务。很多工具最后都成了半成品。

有的公司会组织一些创新比赛，用来鼓励公司的各种创新。很多热爱技术的开发者会在这个场合提出一些新的点子，包括一些构建工具的想法。有些想法是很有价值的，不过，由于缺乏经费和持续性鼓励，比赛中提出的工具提案最终能落地的还是很少。

另外，在没有详细指导的情况下，很多人不知道怎么去构建一个合格的工具。他们的工作主要是运维和测试，而不是编写代码。

担心工具不够靠谱

有些操作具有很高的风险，比如修改数据库、更新镜像版本、删除文件等等。开发者宁肯自己动手，也不愿意让工具去执行这些操作。因为开发者不够信任这些工具。

如果工具是使用者自己开发的，开发者对它会有较高的信心指数。反之，开发者会担心工具的使用会给系统带来副作用。有些工具的脚本可能需要修改，如果没有修改就开始运行，会得到错误的结果。如果不熟悉脚本，修改脚本对开发者来说会有难度。

对于一些高风险的操作，开发者可能要一遍一遍地检查好工具之后才敢使用。这种情况下，开发者可能更愿意自己手动执行操作，而不是使用工具。如果业务需求变化较多，很难开发出一个通用的工具。也许现阶段这个工具可以使用，但是一段时间后，这个工具就不符合业务需求了，这种变更也会阻碍开发者使用工具。

自动化工具缺失的后果

自动化的目标是快速完成任务。工具的缺失会让开发者把过多的时间花在本可以让工具完成的事项中。这既浪费开发者的精力，也影响任务的进度。开发者应该把更多的时间花在工具完成不了的事情上，否则会陷入执行繁杂事务的循环之中，久而久之，这种状况会影响开发者对技术的探索精神，也会降低他们的工作积极性。大部分开发者是想让自己的技术在工作中得到锤炼的。

团队缺少了自动化工具，会渐渐地退化为技术能力和创新能力都很弱的组织，其生命力和成长空间也会被减弱。

### 未能预测系统的行为

周一上午，我接了一杯水但是没有喝完，就随手将水放到了桌子上，打算下午继续喝。下午来看这杯水的时候，发现杯子里面没水了。我确认过没有人动过我的水，原来是水被蒸发成了水蒸气。蒸发这种现象很常见，我未预料到的是，空气竟然这么干燥，水蒸发的速度太快了。

软件系统中，出现这种预料之外的事件的概率很高。有些事件能带来惊喜，有些让系统更加糟糕。这些事件常常是系统的某些行为，开发者常常犯的问题，是未能准确预测系统的行为。这会导致系统的行为逐渐失去我们的控制。

这些行为不是被设计好的，而是突然出现的。比如，在系统单独启用A功能或者单独启用B功能的时候，系统能顺畅地运行，然而，一旦系统将A功能和B功能都启用的时候，尽管两个功能都可以使用，但系统的运转效率迅速下降，我们觉得功能A影响了功能B，也觉得功能B影响了功能A。也可以理解为，当功能A和功能B都运行的时候，系统产生了除了功能A与功能B的行为之外的第三种行为，那就是运转效率下降。

系统的奇妙之处，在于把系统的各个部分的功能进行组合之后，一些新的行为，会随着这些功能与功能之间的相互作用、用户与系统之间的相互作用而呈现。系统的整体效果，不是各个功能的线性叠加，而是系统之内的各个部件、系统之外的各种因素之间巧妙作用的结果。

这些意料之外的行为对系统会产生各种影响。有些行为对系统有利。比如，在一个讲计算机的博客中嵌入一段卖电脑的广告。当用户看完博客之后，去看电脑的广告的概率会提高很多。如果不把电脑的广告和博客放到一起，就不会有更多人看广告这个效用。还有的行为会产生负面效果，甚至引发系统故障。在人工智能相关产品的研发之中，我发现，当系统要处理更多AI模型的时候，每个模型的准确率都会降低。其中的原因是，模型变多后，系统之中某些环节产生了拥堵，这个拥堵导致模型产生的某些结果不能被正确地处理，因此准确率降低了。尽管这些模型都还在正常运行着，它们所带来的整体效果却是不好的。

预测系统的行为

开发者应该准确地预测系统可能具有的没有被设计的行为。这能让我们避免系统产生不好的行为，或者更好地处理这些行为。

预测的另外一个好处是让我们更好地规划资源。如果能够提前得知系统将会有一些负面的行为，我们就能在早期做好规划，为处理这些问题安排好时间。这对于开发复杂系统来说非常重要。很多系统延误，就是因为开发者严格按照需求来完成任务。完成需求之后，没有时间去处理这些未预料到的系统行为所带来的问题了。

有这些方案可以用来预测系统的行为。最常见的就是根据经验判断，事实上现在很多开发者也是这么做的。我们可以根据团队内部积累的解决方案，或者从团队外部学习到的经验，认识到系统未来可能发生的行为。软件行业已经不是一个崭新的行业了，无论我们从事什么领域的开发，都能很容易地从开源库中找到类似的例子。我们还可以从公司内部的其他团队那里汲取优秀的经验。此外，还有很多经验是通过耳濡目染获得的，不需要专门来学习。

根据经验预测的好处就是简单直接。但是这种方式带来的结果具有一定的抽象性，因为我们很有可能只是学了个皮毛，把对象的表面呈现的东西作为真理。毕竟不同的系统之间是有区别的，如果完全照搬，出问题的概率会比较高。

如果觉得经验不足以帮我们准确地预测系统的行为，可以尝试做实验或者做测试。在软件开发中，一般没有实验这个说法，大家都用测试来称呼一些实验性的步骤。这里测试的主要对象是一些在开发之前没有规划好的系统行为，而并非写在需求文档中的系统功能。当功能变得复杂，用户数量增加，数据达到一定量，调用频率逐渐加快的时候，更容易出现一些意料之外的系统行为。因此，在测试中，应该注意各方面的条件是否都充分。测试不能百分之百地帮我们预测出系统所有可能发生的行为，但在测试中，也可能出现一些真实场景下不能产生的行为。比如，我们在测试中模拟了一亿用户，而现实中，用户的个数却最多只有一百万个。测试中庞大的用户量所导致的系统行为，是真实场景下难以复现的，如果我们根据用户量为一亿的测试去预测系统的行为，会误导整个开发流程，也会浪费很多精力。因此，测试应该尽量模拟真实场景，而非追求极限目标。

预测系统行为不应该只出现在系统构建的某一环，而是应该伴随在整个构建过程中。在产品设计的阶段，预测有助于让产品的设计更加合理，避免踩坑。由于产品还没有开始开发，因此这时候无法进行测试。这一阶段预测的主要办法是推理和经验。在开发过程中，预测有助于让开发者找到最佳的解决方案。这一阶段，预测的主要办法会从推理与经验逐渐走向测试，越是在开发的后期，测试的效用越大。

### 花太多时间救火

如果能准确得出每个开发者每天工作的内容，就能判断整个团队的状态。我们可以根据需求和问题的来源给开发者的任务分类，例如，来自产品经理编写的需求文档的任务，来自开发团队内部的任务，来自测试的修bug的任务和来自前场运维的救火的任务。如果开发者大部分时间花在前两种任务上，说明团队处于发展状态，项目处于成长期。如果大部分时间花在后两种任务上，说明业务成长较慢，项目处于缓慢增长期或者停滞期。

很多团队希望能发展得更好更快，但是心有余而力不足，因为开发者花了太多的时间在救火工作上。这类工作种类繁多，它们有个共同的特点，就是对时间要求很紧急，否则就算不上救火工作了。开发者必须停下手头的其他工作，并把救火工作置于最高的优先级。最艰难的场景是，系统到处都是“火”，或者这里的“火”熄灭了，那里的“火”有烧起来了。系统里到处是问题，而且有些不同的问题像藤条一样互相缠绕。开发者很多时候被困于这些工作之中，难以抽出时间去做常规的开发，这影响了常规开发的进度。因为开发者担心自己在正常的时间节点内无法完成常规任务，就会不顾质量只顾速度地赶进度，因此，这也给产品引入了更多的问题。久而久之，团队就处在了一种恶性循环之中。

救火工作往往不讨开发者喜欢。首先，很多开发者认为救火工作没有常规开发重要。常规开发代码量往往比较大，这会让很多开发者心中很有成就感。另外，救火意味着很多时候开发者需要去看已经写好的代码有什么问题。看自己的代码不是很难，但是谁愿意去发现自己写的代码的问题，并承认问题呢?看别人的代码是令人头疼的，发现问题也是困难的。读代码远远没有写代码有趣，否则“读源码”就不会成为一件高级工作了。

在救火工作进行中，开发、运维和测试会互相埋怨。测试和运维一方面不想多次麻烦开发去解决这么多线上问题，一方面又会愤怒地认为开发的工作没有做好，导致了这么多问题。开发会认为很多问题都不需要提给他们，因为很多问题在开发者看来并不是问题，运维应该去解决部分问题，而不是开发者来解决所有的问题。

### 反思容器化之路

#### 换了运行环境，软件老出问题

软件能跑起来，最主要的因素有两个，一是软件本身，包括软件的设计、代码、配置等等。另外一个因素是运行环境。运行环境要跟软件的要求相匹配，如果运行环境跟不能满足软件的要求，软件要么跑不起来，要么不能正常运行。这就意味着，一个软件，可能在一个环境中能正常运行，但是在另一个环境中却不能正常运行。

Java之所以得以迅速占领市场，独领风骚多年，跟其“一次编译到处运行”的跨平台能力分不开。如果一个Java编写的软件使用了JDK 1.8的新特性，而环境中安装的JDK版本是1.6，软件就会报错。Java尚且如此，那些不支持跨平台的语言恐怕问题更多。很多软件在Linux系统能运行，在Windows中就会报错。换了运行环境，软件运行老出问题，最坏的情形下，有多少个运行环境，就需要软件有多少个不同的匹配的版本。

容器化最主要解决的就是这个问题。

#### 容器太多，操作麻烦

当容器的个数很少时，操作并不复杂。不过容器的优势似乎并没有很好地发挥出来。容器个数少，意味着服务的个数很少，也意味着系统在部署方面的复杂度并不是很高。不使用容器，问题并不大，大不了不同的运行环境用版本去部署得了。

容器变多时，手动管理很耗时，也容易出错。试想一下，假如一个分布式系统有四个节点组成，微服务的个数有30个。这也就意味着，如果把每个微服务都用容器部署，镜像和容器的个数都会超过30个，每次对镜像和容器进行操作都需要使用docker image，docker run，docker kill，docker rm这些命令。由于事前没有对这些指令分类，因此，不同的命令和容器之间很容易混淆，操作起来非常麻烦。

Docker Compose解决了这个问题。Compose 是用于定义和运行多容器 Docker 应用程序的工具。通过 Compose，用户可以使用 YML 文件来配置应用程序需要的所有服务。然后，使用一个命令，就可以从 YML 文件配置中创建并启动所有服务。

#### 系统缺少韧性和弹性

由容器组成的系统如何才能提供稳定的服务？系统出问题后如何找到解决方案？这是容器化面临的第二个问题。系统不是静止的，它无时无刻不在变化。

使用kubernetes可以为系统增加韧性和弹性。

## 运维篇

### 未能从事故中汲取经验

在软件领域，事故指的是发生的一系列意料之外的事件导致了系统的不稳定。并不是特大的灾害才能算得上事故。一个例如用户无法登录的小bug可以被叫做事故，一次造成重大金钱损失的银行系统崩溃也可以被叫做事故。

事故可能造成各种损失，例如让公司损失一笔金钱，让团队失去一次机会，让某个人受到处罚，但如果更加广义地去去思考，会发现事故是团队的一次宝贵的学习机会。

学习有两种方式。一种是正面的教材，一种是反面的教材。反面教材是经典的供人借鉴吸取失败教训的反面事例等的统称。正面的教材的逻辑往往是，事实是什么样子的，我们应该怎么样做。反面教材的逻辑往往是，事实不是什么样子的，我们不应该怎么做。学校里，老师会表扬几个做得好的同学，他们就是正面教材，也会批评一些做的不好的行为，这些行为就是反面教材。

事故就是我们宝贵的教材。我们既可以从系统达到预期效果的行为中去学习，也可以从系统没有达到预期效果的行为中去学习。当一切正常时，我们的认知会与真实情况不一致。我们会自信地认为我们对系统所做的改动都是对的，因为系统达到了我们的预期效果。而往往系统并不能达到预期的效果，只是场景还不够复杂而已。这就好比一个人平时身体没有什么状况，所以他总是觉得自己的身体很棒。他会保持以前的生活习惯，因为他认为这种习惯不会对身体造成大的伤害。突然有一天他感到有些不适，去做体检，结果发现自己已经生病很长时间了，只是之前没有症状而已。如果他不感到不适，他会一直觉得他是健康的。表面现象是他没有生病，真实情况是他生病了，如果没有症状，他的认知跟表面现象是一致的，跟真实情况是矛盾的。如果有了症状，他的认知才会跟真实情况一致，问题才有可能得到解决。

很多人或者团队没有虚心地向事故学习。从心理上，谁愿意承认自己以前的想法有问题呢？从精力上，很多开发团队忙于新需求的实现，而不是去总结经验教训。一次宝贵的学习机会就这样被浪费掉了。究其原因，还是因为事故没有充分引起大家的重视。大家不愿意费神去研究事故产生的原因和解决办法。即使了解了原因，也没有再进一步研究深层原因。想到的解决办法也只能临时解决问题，不能从根源上消除问题。

为了更好地从事故中汲取经验，团队需要专门花时间组织一些经验学习和经验分享活动。这种活动有很多种叫法，比如事后经验总结，事后经验分享会，事后回顾，我比较喜欢用事后复盘这个叫法。

避免过度指责

每当发生事故的时候，团队里最常发生的事情就是为自己开脱并指责其他人。指责有时候的确可以让自己逃避一些惩罚。当一方有错的时候，管理者会认为是这一方干的活儿不够好，导致了事故。因此这一方会有很多责任。当每个参与方都有过错的时候，管理人员会意识到，也许是这个问题本身比较复杂，出点事故在所难免。他就不会针对某一个人去发脾气了。每个人都不想承认自己是那一个导致事故的人，也不希望自己在领导和其他人眼中是一个容易引发事故的人。在这种情况下，为自己开脱和互相指责在所难免。

指责往往给人之间带来不必要的矛盾，有些时候矛盾得以消除，有时候这种矛盾会进一步恶化成下一个事故。人们往往希望自己能做到对事不对人，在与别人沟通的时候，也说自己对事不对人，而真正做到这一点却非常不容易。有时候，稍有不注意就会带来引发矛盾，被指责的人这种指责是过分的，即使他犯了错，他也不应该受到这样的指责。而指责他人的人会觉得这只是让每个人明确自己的责任，就是说事，跟人没有关系。

指责容易让人产生心理阴影。在指责他人的时候，很容易就给他人贴上了一些负面标签。这些标签会影响他人在团队的合作与发展。那些被指责的人，会认为其他人对他有偏见，甚至产生自卑心理。久而久之，他与其他团队成员的关系可能发生改变，从友好合作关系变成了针锋相对的关系。而且他可能产生一种报复心理。

指责不利于下一次事故的排查、解决和复盘。人们害怕这种指责，所以会逐渐逃避责任。有些时候，他们的某些误操作造成了系统的事故。然而他们却隐瞒真相，不愿去承认他们有过什么误操作，即使这种澄清有利于事故的解决。团队要花费大量的精力去找原因，最后还是一头雾水。如果每个人的误操作都是匿名的，比如都用同一个账号对系统进行更新，那么真相可能会永远被隐瞒。如果误操作不是匿名的，人们也不愿意一开始就坦率地承认。

图片

系统事故

团队应该塑造一种过度指责的文化。在这个文化中，每个人都不怕坦诚地承认错误而受到过度指责。这对团队的良性发展很有帮助。

事后复盘

事后复盘并不是开个事后会议那么简单直接。为了让事故更充分地发挥其反面教材的价值，我们需要注意让事故的相关人员都能有所触动，并且要把复盘的结果记录下来以便于以后追。如果一次事后复盘能让团队的整体氛围变得更好，让成员的工作习惯得以改善，复盘的意义就超越了事故本身。

事故的相关责任人

技术人员

技术人员包括开发者和运维人员。如果产生了事故，直接引起事故和能直接解决问题的都是技术人员。运维人员为系统提供类赖以运行的基础设施和资源，开发者为系统提供代码和配置。问题一般就出在这两个领域。当然，也有可能是产品设计的逻辑有问题进而导致了事故，不过即使是产品的问题，产品的更新也得依赖技术人员。

项目经理

项目经理是整个项目的进度管理者。事故多多少少会影响进度，项目经理的关注能推动事故尽可能快地解决，进而让整个产品的进度正常。此外，项目经理会时时刻刻关注有关系统的所有事件，事故是其中之一。事故有助于项目经理认识到系统的长度和短处，认识到各方面工作人员的工作质量。这些信息能帮助项目经理之后做决策。比如哪里需要添加或者更换人手，哪里需要增加服务器资源，哪里需要修改功能。

产品经理

产品经理是产品的生产者和消费者之间的桥梁。他们既要负责产品的质量验收，又要负责把产品售卖出去。这种双向的压力让他们不得不认真思考阻碍产品走向成功的要素。事故的发生意味着产品的稳定性有问题，这会对产品在消费者心中的形象造成重大负面影响。产品经理即希望事故能尽快解决，也希望事故的概率能降低。他既希望系统能从技术角度加以改善，也会从产品的思路思考如何规避事故。

事故的复盘过程

整理事故的发生过程

事故的发生过程对于处理事故的作用，就像案件的发生过程对破案一样重要。事故的发生过程主要包括这些方面：事故是自动发生的还是人为操作？什么事件直接导致了这一次事故？事故的深层原因是什么？事故发生后系统有没有自动恢复正常？事故造成的损失包括哪些方面？

实施和跟踪事后行动

事后复盘的直接结果就是事后行动指南，它告诉我们谁需要在什么时间做什么事情。行动指南应该更加清晰而具体，不应该含糊其辞。此外，不光要去实施行动，还要跟踪行动的落实情况。

记录事后复盘

有条理地记录事后复盘能更好地发挥事故的作用。记录能让团队在以后的工作中少犯类似的错误，也能让那些没有直接与事故相关联的人员了解发生了什么。技术人员和非技术人员都有可能去查看事后记录，因此，记录应该尽量做到让大家都能看得懂。记录是对整个事故的描述，不是技术文档，但可以包含技术文档。记录可以包括事故概况、详细描述、事后行动这些方面。

### 数据没有备份

最近收到这则新闻：某公司收到系统监控警报，获悉微系统服务出现故障，生产环境和数据遭受到严重故障。目前，系统仍未恢复，系统预计恢复时间在五天之后……事故使得该公司损失了百万客户，预计事故对公司造成的直接经济损失为1亿。除此之外，客户质疑、损失赔偿、客户流失、竞争对手挑战，这些都是该公司接下来要面临的问题。

大部分公司，都或多或少地经历过类似的事故。有极少数事故是相关者故意造成的恶意事故。“删库跑路”是软件开发界流行的段子，不过如果这种事情真的发生，后果可能很严重，对公司造成的影响也不会小。

大多数事故是开发或者运维失误造成的。想象一下这个场景：客户刚刚通过系统完成了几笔交易，不过他之后想合并这几笔交易，但是前端没有办法完成交易的合并操作。客户要求开发者从后台修改一些数据，以达到交易的合并效果。开发者手起刀落，立即写好了SQL，让运维在生产环境跑了一遍。检查的时候发现，与该客户相关的所有交易数据都修改了，包括三个月之前的历史数据。开发者发现，SQL的问题是没有加过滤器，修改的范围太广。由于数据没有备份，开发者无法恢复历史数据。不久之后，系统就失去了该客户。

如果这个开发者做好数据备份，事情也不会这么糟糕。备份指的是将数据或者系统的状态以某种方式保存下来。数据备份泛指将数据以某种方式加以保留，以便在系统遭受破坏或其他特定情况下，重新加以利用的一个过程。它最大的好处是提高系统的高可用性和可恢复性。无论是硬件设备受到破坏，还是软件或者数据遭到人为的破坏，合理的备份都能减小事故带来的负面影响。如果数据很重要，数据备份就很重要。当我们尝试修改或者删除数据的时候，要做好数据会丢失的准备。备份应该成为必要的流程，而不是一个可选的操作。

**数据备份的三种策略**

数据备份有三种不同的策略，完全备份、增量备份和差异备份。它们的区别主要在于备份的数据到底是全部数据还是修改的数据。

**完全备份**

备份全部选中的文件，并不依赖文件的存档属性来确定备份那些文件。在备份过程中，任何现有的标记都被清除，每个文件都被标记为已备份。换言之，清除存档属性。完全备份就是指对某一个时间点上的所有数据或应用进行的一个完全拷贝。

**差异备份**

差异备份是指在一次全备份后到进行差异备份的这段时间内，对那些增加或者修改文件的备份。在进行恢复时，我们只需对第一次全备份和最后一次差异备份进行恢复。

#### ****增量备份****

备份自上一次备份（包含完全备份、差异备份、增量备份）之后有变化的数据。增量备份是指在一次全备份或上一次增量备份后，以后每次的备份只需备份与前一次相比增加或者被修改的文件。这就意味着，第一次增量备份的对象是进行全备后所产生的增加和修改的文件，第二次增量备份的对象是进行第一次增量备份后所产生的增加和修改的文件，如此类推。

这三种备份策略各有优劣。下表对比了三者的空间使用、备份速度和恢复速度。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 完全备份 | 差异备份 | 增量备份 |
| 空间使用 | 最多 | 介于两者之间 | 最快 |
| 备份速度 | 最慢 | 介于两者之间 | 最少 |
| 恢复速度 | 最快 | 介于两者之间 | 最慢 |

## 管理篇

### 矛盾和冲突影响效率

前年有这样一个新闻，某互联网公司的产品经理提了个需求，要求APP开发人员可以做到根据用户的手机壳来改变软件主题颜色，导致开发人员和产品经理大打出手。这个新闻引起了网友对APP开发的广泛关注。

先不说故事的具体情节，也不探讨APP能否根据手机壳颜色来改变主题色这一需求的可行性。它反映的是软件开发中的矛盾和冲突，这正是本文要探讨的内容。各行各业的同事之间都会有矛盾与冲突，给它们分类有助于矛盾和冲突的分析和解决。软件开发中的矛盾和冲突主要由不同团队、不同角色的不同立场和沟通差异引起，在这里主要将它们分成了这几大类型。

矛盾和冲突的类型和消极影响

决策问题

一个团队往往不止一个负责人，团队的技术主管、产品主管、项目主管、技术团队二把手、技术主管的老板以及团队内外的其他成员都有可能直接或者间接地参与团队的管理事务中。从团队整体的角度上讲，管理者们应当齐心协力共同完成好团队的目标，但是实际上，每个管理者都会有自己对产品、项目和系统的认识。这些认识的不同有可能造成分歧和矛盾。在实践中，不同管理者之间的矛盾往往是组织缺乏效率的一大原因。这些矛盾主要造成这些问题：

1、某一方说了算，虽然保证了决策的统一性，但其他管理方没有在决策中发挥作用。

2、大家有不同的观点，没有形成统一的决策。

对于产品设计和需求分析来说，产品主管是核心负责人。对于开发和技术问题来说，技术主管是核心负责人。对于项目进度管理来说，项目主管是核心负责人。在这些活动的进行中，非核心负责人也会以参谋的形式参与各种决策。

核心负责人对负责的任务具备专业的知识，他们对任务是最熟悉的，也是最有发言权的，他们常常希望自己能够对该任务具有绝对的掌控权。对于他们来说，由于要对任务的结果负主要责任，因此当参谋要与他们一起制定决策时，他们会认为参谋的建议是多余的，甚至干预了他们的工作。在这种情况下，参谋的一些有见解的建议不会被采纳。这会导致参谋的价值不能正常体现，也会让决策没有那么准确。参谋会认为自己的能力没有发挥出来，也会觉得核心负责人一个人说了算的做法有点过度自信。更为严重的是，参谋会认为自己没有被充分尊重。这些因素会让他们之间相互的不满情绪更加高涨，甚至爆发冲突。这也会影响他们之后的合作，影响团队整体的效率。

责任问题

软件在运行中出了严重的系统问题，是后端的问题还是运维的问题？产品销售额没有达到预期，是开发的责任，还是产品的责任或者商务的责任？那些重要的难以解决的技术问题或者业务问题，应该谁去弄清楚并攻克？谁应该去承担团队中吃亏不讨好的脏活儿累活儿？

这些问题都能划到同一个大类之中，那就是责任问题。人们都是趋利避害的，他们希望自己在团队中有重要的位置，有决策权，能获得更多的资源，但是愿意承担不利后果的人却很少。这就会造成工作中常见的甩锅现象，从而引发矛盾和冲突。这种推卸责任的做法，会在团队中造成沟通壁垒，也会削弱大家的责任意识，不利于大家把事情完成好。

矛盾和冲突的积极影响

彼得德鲁克认为，在做重要的决策之前引发争论非常重要，因为争论使得决策者不会成为组织的俘虏，争论也能为决策者提供不同的替代方案。争论也能刺激人的想象力，提高企业决策质量。德鲁克提到了冲突的积极影响。认识到冲突的积极影响有利于我们正确地处理冲突，避开其消极作用，发挥其正面价值。

矛盾和冲突的积极影响常常被忽略，然而在组织中，它的影响并非总是消极的、破坏的，有时候它是积极的，有益的。由于人和人之间的存在着一系列差异，矛盾和冲突不可避免地会存在于组织之中，它们是差异的体现。冲突不仅是合理的，也有着积极的一面。没有矛盾和冲突，很多组织的问题就无法暴露出来或者被妥善解决；组织如果一直处在一团和气的氛围中，会缺少创新和变革的动力，停滞不前，缺少竞争力。

矛盾和冲突的程度对组织的影响呈现倒U型的效应。没有冲突或者冲突程度太低时，团队会出现无视变化、缺乏新意的情况。有了适度的冲突，团队能做到自我批评与自我发展，这有利于创新。当冲突太多时，就会造成混乱和分裂，组织就不像一个完整的组织了。

处理矛盾和冲突

找到重要的矛盾和冲突

矛盾和冲突无处不在，每个人的精力都是有限的，因此，不应该插手每一个问题。应该先找出重要的矛盾和冲突，并把关注点放在这些重点上。如果某些矛盾和冲突的处理能解决重大争端、提高大家的认识、刺激组织的创新、促进团队凝聚力，它们就是重要的。

找出深层原因

矛盾和冲突既和事情本身有关，也和人有关。同一件事，如果发生在不同的人身上，严重程度也会不同。既要从事情本身找原因，也要从冲突双方的人身上找原因。仔细分析是哪些人卷入了冲突？双方有什么样的观点和立场？冲突是否影响了双方的利益或者在组织中的地位？双方有着什么样的性格和人格特质？造成冲突的原因是只有一个还是有多个？之前是否有类似的矛盾或者冲突发生？

例如产品经理和开发者发生了冲突，很可能由于各种原因，他们之前也发生过不愉快的事情。发生冲突的表面原因，只是一个导火索。找到深层原因后，事情才能得到根本解决。

选择解决办法

针对不同的类型的矛盾和冲突，应该采取不同的解决办法。解决的目标是大家能最大可能性地发挥自己的价值，能愉快地沟通与合作，团队也能发挥最大的效能。

解决的办法有妥协、迁就、回避、强制、合作等。如果双方地情绪非常激烈，需要先回避让双方恢复平静。如果维护双方的关系最重要，需要采用迁就的方法。如果双方各执己见僵持不下，妥协是最好的办法。如果认为事情必须要得到正确且快速地处理时，可以采用强制的办法先解决问题，之后再慢慢做安抚工作。如果有折衷的办法，双方可以开诚布公地谈好要求，走向合作。总之，要看到底是事件本身重要，还是双方的关系重要。上策是既将事情本身处理好，也照顾到大家的情绪，妥善地处理好关系。

### 任务依赖造成拖延

#### 独立任务、串行任务、并行任务和互补任务

产品的最终实现需要团队协作完成若干项任务，这些任务之间有着不同的关系。有些任务没有明显的关系，有些任务之间的关系却很紧密。我把任务之间的关系大致分为四种形态。

有的任务的完成和另一项任务的完成没有明显的关系。他们相互独立，并且他们也不会依赖于同一个任务。这两个任务是互相独立的任务。图a是独立任务示意图。有的任务的完成依赖于另一个任务的完成或者部分完成，这两个任务是串行的。在串行关系下，有的任务需要另一个任务全部完成才能开始进行，更多情况下，后面的任务只需要前面任务的部分信息就可以开始，但在前面任务完成之前不能结束。图b是串行任务示意图。有的任务之间没有依赖关系，他们依赖于相同的任务，但彼此独立，这两个任务之间是并行关系。图c是并行任务示意图。还有的任务，需要依赖于两个或者多个任务的完成或者部分完成。这些任务都是下一个共同任务的必要条件，他们一起组成了互补任务。图d是互补任务示意图。

图片

图a 独立任务

图片

图b 串行任务

图片

图c 并行任务

图片

图d 互补任务

真实场景下的任务数量要比这些示意图中多不少。而且，任务之间的关系更加复杂。在极端条件下，任何两个任务之间都可能形成依赖关系，并且依赖关系可以是相互的。如果有n个任务，并且每个任务都依赖于其他的任务，这n个任务会组成一个具有n\*（n-1）的关系网络。在大部分情况下，n个任务之间依赖关系的数量在0到n\*（n-1）之间。

图片

图e n个任务之间最多可能形成n\*（n-1）个依赖

任务的依赖关系造成了拖延

要开发完产品，需要完成每一个任务，并且处理好任务之间的相互关系。完成某一项开发任务，可能有诸多阻碍：比如对业务的理解不够清楚，某个技术点太复杂没有办法解决，系统有瓶颈需要加资源，上游没有发数据给我们，环境还没搭建好，某个员工突然离职……

问题可能多种多样，如果进行概括，大概可以把问题分为两类。一类是我们自己的问题，一类是其他人的问题。自己能解决的问题往往都是好解决的，难解决的是要依赖其他团队或者成员的问题。团队中的成员或者不同团队所执行的任务往往是互相依赖的。如果被依赖的成员或者团队没有完成任务，依赖的成员或者团队也是无法完成任务的。这就像软件中的依赖关系一样。如果被调用的方法存在缺陷，调用的方法自然不能输出正确的结果。

团队或者成员的相互依赖性是他们相互作用的变量之一，并且这种依赖性往往与任务的依赖性直接挂钩。即使任务之间没有依赖性，团队或者成员之间也可能有依赖关系，但任务的依赖性使得团队或者成员之间的依赖性更强。这种依赖性不光存在于软件开发领域，也存在于其他领域。

软件开发中的依赖关系有很多种不同的类型，但大多是对信息和数据的依赖。比如，开发者依赖于产品经理阐述需求，前端程序员依赖于后端程序员提供接口，后端反依赖于前端进行联调和效果展示。还有的依赖是审计性质的，比如要发布新的版本之前，开发团队需要产品发布团队的审批。

无论依赖是什么类型的，它都造成了一定程度的拖延。开发者会认为，既然被依赖方没有完成好任务，依赖方也无法完成任务。当他人询问进度的时候，开发者也有了开脱的办法。不管开发者能不能在最终时间点前完成任务，拖延一定发生了。如果把这些拖延的时间用来做其他更有意义的事情，比如提高系统的质量和性能，对团队是大有裨益的。

克服依赖问题造成的拖延

让任务的依赖关系变得清晰

不要高估开发者对依赖关系的理解程度。一个新手，可能只能发现任务中的一些明显的依赖关系，而不能发现一些潜在而重要的依赖关系。这会让他们在碰到要依赖的环节时摸不着头脑。他们不知道该找谁解决依赖问题，进而无法顺利推进工作。

在开发的预研环节，主管应该让任务的依赖关系变得清晰，让所有的开发者明确自己的任务边界，清楚自己的任务中需要依赖他人任务的部分，并且和被依赖方达成沟通。被依赖方应该向依赖方承诺一个完成被依赖任务的时间点。被依赖方应该明确任务中被依赖方强依赖的环节，而不是胡子眉毛一把抓。依赖方应该明确任务中不依赖其他方的环节，并将其优先级调高，除非客观条件不允许。

在开发中破解任务依赖的困境

为了提高整体效率，开发者应该主动要求被依赖的团队或者成员尽快解决问题，而不是被动地等待。如果被依赖方没有及时解决问题，可能是因为资源不足，也可能是没有意识到问题的重要性，不论是什么原因，都应该让被依赖方明确这个问题会造成依赖方的等待。如果被依赖方没有明确的回应，可以将问题升级。

如果充当被依赖方的角色，开发者应该明确，优先级不光是对自己而言的，而是对整体而言的。不能想当然地先去做自己认为最重要的事情，而是整个项目最重要的事情。如果资源不足，应该申请增加资源。总之，应该让整个团队运转起来，而不是让团队看起来像发生了死锁问题，不再运转。

管理者在开发流程把控中，应该善于破解任务依赖的困境。一旦发现开发者的工作进展放缓了，不光要思考开发者本身的任务的问题，还要思考被依赖方的任务的问题。应该主动地协调好他们的工作，让他们加强对所做任务的依赖关系的认识，并调整优先级。既要帮助依赖方解决问题，也要帮助被依赖方的解决问题。

改进流程中的把关

在开发中会有一些审批流程，有时候这些来自其他团队或者小组的审批流程会真的增加价值，比如降低风险，提高质量。还有些情况下并非如此。过多的把关不一定能解决问题，反而让开发者在工作中把心思花在这些流程上而不是真正的产品上。

应该改进流程中的把关。去除不必要的把关，让开发者专注在产品的质量上。自动化某些流程，让审批变得更加容易。

### 违反康威定律

康威定律 (康威法则 , Conway's Law) 是马尔文·康威1967年提出的："设计系统的架构受制于产生这些设计的组织的沟通结构。"即系统设计本质上反映了企业的组织机构。系统各个模块间的接口也反映了企业各个部门之间的信息流动和合作方式。

James O. Coplien与Neil B. Harrison在《敏捷软件开发的组织模式》中写道: “如果团队、部门、子部门等的组织结构没有紧密反映产品的必要组成或产品组成的关系，那么项目将会遇到麻烦。”换言之，什么样的团队架构和团队组成，就有什么样的产品。很多部门想构建低耦合的系统，最后发现系统的模块化却很差。他们除了需要去调整系统架构和设计细节之外，也要思考团队是否违反了康威定律。康威定律反映的是团队和工作内容之间的映射，它从宏观上说的是团队规模和结构的问题，微观上，说的是任务分配的问题。

团队规模

生物学家斯蒂芬.杰.古尔德曾在书里讲过一个故事：两个女孩在游乐场聊天，其中，一个女孩问：“如果蜘蛛能变得和大象一样大，还能到处爬，那岂不是很可怕吗？”另一个女孩回答说：“才不会。如果蜘蛛有大象那么大，那它就会变得跟大象一样笨重了。” 古尔德认为第二个女孩的说法是对的，因为生物的大小决定了其形态和行为。如果蜘蛛变得跟大象一样大，其形态和行为也会变得跟大象一样。

开发团队也是如此，当规模增加到一定程度的时候，团队就会变得复杂和不灵活，其战斗力就会被削弱。软件开发中有很多这样案例。一个五人小组正在开发转账功能，由于这个功能比较复杂，他们的开发工作量都比较大，每天都很忙碌。后来团队中增加了两个新人，七人一起完成这个功能。尽管大家的忙碌程度并没有降低，项目的进度却没有提高。这种情况很普遍，其原因是团队规模的增加会导致一些新的问题，例如沟通的成本提升、资源的竞争等等。这些问题会给团队的效率提高的绊脚石。

在这里，把边际成本定义为，团队规模扩大或者减小一个1个单位所引起的团队成本的变动。在相关范围之内，扩大或者减小1个单位的团队规模就是单位规模的变动成本。把边际收益定义为团队规模扩大或者减小一个1个单位所引起的团队收益的变动。在相关范围之内，扩大或者减小1个单位的团队规模就是单位规模的变动收益。

假设每个人的工作效率都相同，这时，规模的边际收益会一直不变。但是边际成本会随着规模的扩大而上升。在规模达到一定的水平时，边际收益和边际成本相等。此时团队的人数是合理的，因为此时团队总收益最大。当团队规模超过这个临界值时，团队总收益会逐渐减少。继续增加团队规模，团队总收益会小于0。这也是收益和支出相等的状态。

图片

图1 边际成本曲线和边际收益曲线

图片

图2 团队总收益曲线

理想状况下，团队的规模应该是图1中边际收益和边际成本相等时的规模。当人数小于这个规模时，团队应该继续扩大。然而在实际开发中，每个人的效率不可能完全一样。有些优秀的开发者，其完成的工作量可能是一个普通开发者的数倍。另外，边际收益和边际成本不光跟人有关，也跟要完成的任务本身有关。任务的属性和复杂度很大程度上决定着开发者的效率。因此，衡量团队的合理规模很困难。总的来说，相同任务的开发人数不宜太多。

团队结构

团队结构应该跟系统的架构保持一致。把团队分成不同的小团队，每个小团队都对自己模块的整个生命周期负责。这样能让团队自然地自治内聚，也能让成员成为自己所属模块的专家。

团队也不宜分得太小。否则每个人都成了螺丝钉，没办法承担更大的任务。而且，这容易让团队的工作安排变得不灵活。

团队结构的设计应该让团队成员之间的沟通变得更加容易，而不是更难。如果一个部门有很多个平行团队，团队之间工作上的依赖比较多，但是团队之间的成员互相不熟悉，沟通太难，部门的整体效率就会大打折扣。应该有意识地去破除这种沟通壁垒。团队之间应该有专门做沟通和协调的负责人去推动各种事项。

任务分配

开发主管会根据任务的工作量和时间紧迫性、现有开发者人数和经费这些因素来思考某个任务究竟给多少人做、给谁做。任务分配具有风险，需要慎重考虑。分配的最终目标，是让软件在规定时间内能按质按量完成。

有哪些分配方式容易是违背规则的呢？

任务的工作量和分配的资源不匹配。工作量大，但分配的资源不足会造成工作无法按时交付的问题。工作量小，分配过多资源，会造成资源浪费。

任务分配没有充分发挥开发人员的长处，或者不能充分发挥积极性。每个任务里面包含了多个板块，每个开发人员都有自己擅长和感兴趣的领域。如果开发人员的特长和任务本身不匹配，就会降低开发人员的开发热情，进而对整个开发造成负面影响。有的开发者水平一般，如果把难度很大的活儿交给他恐怕很难按时完成任务。对于那些喜欢啃硬骨头的开发者，让他们尝试一些难度大的任务是很有必要的。

任务分配不利于开发者们的协调与配合。比如，两个同事之间有一些工作之外的矛盾，如果让他们一起完成一件事情，恐怕不是最好的选择。因为他们两人之间的关系会影响他们的合作。如果合作不能有效地进行，就难以提高他们的效能。

第四，没有关注任务的重要性和优先级。对于那些非常重要的任务，应该重点关注，找靠谱的人去干。因为如果这里出了问题，坏的影响会更大，风险也更高。

怎么分配任务才能降低复杂度呢?分配任务是一个重要的决策。在做决策之前，主管应该充分认识任务本身，充分了解开发者。了解好双方之后再做决策，才能提高决策的成功率。决策失误的概率就会小很多。

主管应该全面掌握各个任务，并划清什么是主要任务，什么是次要任务。还要清楚什么是紧急任务，什么是非紧急任务。在有些时候，主要任务和紧急任务是同一件事情。有些时候，主要任务和紧急任务是两码事。就像一道席，有的是开胃菜，有的是主菜。开胃菜要先上餐桌，优先级更高。不过主菜才是这道席的重点和亮点。主管应该掌握其中的技术难点、业务难点，掌握开发人员主要会把时间花在哪个环节。

主管应该充分了解每个开发人员的长处和短处，了解他们的开发习惯和兴趣点。效率和意愿是两个重要的因素，前者决定开发者能不能干，后者决定他想不想干。只有在既能干又想干某件事的时候，开发者才能做好某个任务。此外，除了了解每个开发者，主管还应该知道，开发者之间是否能顺利地合作。很多类型的合作都是不成功的，不顺利的，这不光体现在做不同任务的团队之间的合作中，也体现在做相同任务的同事的合作之中。如果合作不是那么有效，就会出现人多反而不如人少的局面。这不利于复杂度的控制。

### 多人合作的懈怠问题

在开会的时候，台下的众人一起鼓掌，如果每个人都用力鼓掌的话，会掌声雷动。但是很多场合下，掌声没有那么响亮。因为人多的场合下，我们很难区分掌声是谁发出的，很难知道谁用力了，谁没用力。这是一种社会懈怠的心理作用：在群体效应下，个人很有可能会减小努力。

人多未必力量大。在软件开发领域，这种情况也屡有发生。如果一个开发任务，需要两人或者多人完成，任务被分担了，责任也被分担了，每个人受到他人观察和他人评价的顾忌就减小了。在这种社会作用下，人内心趋向于更加懈怠而不是更加积极。这样每个人的贡献也就更少了。这里有两种情况。第一种是，所有的人的责任意识都大幅度降低了。就像鼓掌一样，他人不太会关注每个个体是否用力鼓掌了。在这种情景下，除非某个人的责任感特别强，每个人都可能会有所倦怠。还有种情况是，团队中的一部分人被认定为这个任务的负责人，或者他们自发地成为了任务的实际负责人。这些人往往在团队中能力突出，比如他们在团队有着更长时间的工作经验，或者他们的工作成熟度更胜一筹。另一部分人能力弱一些，他们是团队里的跟随者。这种情况下，负责任的人往往比不负责任的人干的活儿多很多，收获也更多。

无论是哪种情况，团队都没有把每个角色的效能充分发挥出来。这种复杂度似乎只跟人有关，而跟软件本身无关。这是错误的看法。因为软件开发和鼓掌不太一样。鼓掌不需要什么培训，软件开发确是一个复杂的工作。那些干活儿多的开发人员所能学到的业务和技术会比干活儿少的开发人员多，他们在以后的开发和运维中，也会更加熟练。他们承担的任务更重，而且这会形成一种循环，直到某种变化打破这种状态。甚至有这种状况，当某个开发者的能力或者工作积极性远低于其他团队成员时，他会经常性地处于掉队的状态，他对团队所做的贡献可能微乎其微。

多人合作容易出现成员懈怠。个人担责和众人担责是两个不同的情形，完全根除懈怠问题，让每个人都百分百地努力是很难达到的。我们可以采取以下的某些方法。比如，将一个大任务拆分地更细，让团队中的每个人都明确自己的任务，而不是含糊地分配任务，并且每个人的分工对于其他人都是可见的。其他人在评价某个任务完成得好不好时，会直接奖惩每一个人，而不是这个团队的责任人。干活儿多的人，应该更具有团队意识，不能只想着自己把所有的活儿完成掉，而是让大家一起完成好任务。这对于大家来说，也是一种培训。另外，要让每个成员都感受到任务的挑战性、吸引力、重要性，也要让每个成员感受到自己在团队中的重要性。任务的难度对成员来说应该适度。如果某个任务，对熟练的成员来说不费吹灰之力，对不熟练的成员来说要花费几十倍的精力，让不熟练的成员完成这项任务，不会激发他的成就欲望，因为他即使花那么多的时间，其他同事也只需一下子就能完成。因此，选人也很重要。在我看来，如果两个成员的工作熟练度相差太大的话，他们不太适合平行地合作，完成同一个任务。熟练度的差别，最后会体现为工作成效的差别。把他们安排在一起，最后的结果往往是，熟练的同事觉得他也许一个人工作比合作还来的快。这对团队，对个人都没有好处。

很多文章指出，要把工作做好，需要有ownership。但这些文章并没有指出，怎么才能更具有ownership。要成为有ownership的人，既需要自己有基础和积极性，也需要特定的场景。比如某个中等熟练度的开发者，如果让他跟一个熟练度一般的成员一起开发的话，他必须自己负责整个事情，否则任务无法完成，这时候他往往很具有ownership。如果，他跟一个熟练度非常高的成员一起合作完成更有挑战性的任务，他可能就更加依赖那位更熟练的成员了。因此，要想解决多人合作的懈怠问题，需要研究所有成员的熟练度和心理因素，也要研究成员之间的关系，还要研究任务本身。

### 违反约束理论

约束理论 (Theory of Constraints) 是一种管理哲学，专注于链条中最薄弱的环，以改善系统的性能，它是企业识别并消除在实现目标过程中存在的制约因素（即约束）的管理理念和原则。约束理论在软件开发领域照样适用。很多团队效率低下，不是开发者的技术不够好，也不是系统架构不合理，而是有些短板限制了团队达到更高的目标。这里从流程、团队成员、部署环境三个方面阐述那些制约团队效率的因素，并提出解决方案。

#### 软件开发中的约束条件

##### 流程

A团队负责物流系统的开发和运维。该系统每个两个月有一次版本发布，今天是周五，下一个发布日期还有一周。这次发布的内容比较多，业务上包括了一些新功能和一些老功能的优化，既有大量代码变更，又有数据库层面的更新。这两个月来，A团队一直在为这次发布奋力拼搏，当测试人员宣布该版本没有解决的bug的数量少于10个（少于10个说明bug数量不多，系统达到了发布的条件）的时候，大家都松了一口气。接下来就是准备各种文档，向公司的软件发布委员会提变更请求的时间了。只要在下周三内，变更请求被审批通过，版本在下周五就可以顺利地发布。否则，发布就得延期，公司和部门都不希望A团队发生这样的事。

当天，A团队提交了变更文档。软件发布委员会要认真地检查一次变更的文档，大约需要二十分钟，如果加上一些沟通时间，整个审批大概要半个小时。A团队之所以当天就提交请求，是因为担心审批太慢。事实上，发布委员会向来如此。他们总是有干不完的活儿，一个请求体过来，在他们那里至少得花好几天才能走完审批流程。

A团队和其他往常一样，开始了几天的等待。这几天，A团队也做了一些其他事情，例如编写用户手册、规划下一个版本，但是他们的心还是困在变更请求的审批上。毕竟这才是重中之重。周二，团队经理找到了发布委员会中负责物理系统审批的人，希望他尽快处理。得到的答复是，今天实在没空，已经尽力了。周三，团队越来越焦虑，经理直接通过电话找到变更委员会的主管。主管发了一封邮件，督促审批者尽快处理，在过了五个小时后，也就是晚上九点多，A团队终于看到变成了变更请求的状态变成了“已审批”。

终于松了一口气！但是两个月之后，A团队将再一次陷入变更请求等待审批的焦虑中。半小时就可以搞定的事情，A团队要花好几天的时间等待，相当于半小时的几百倍。而这还不是最坏的情况。如果变更文档有问题，审核次数将会增加，等待时间也会加倍。A团队还遇到过最坏的情况：在截止时间之前，审批没有通过，版本发布不能顺利进行。

类似的故事，不光发生在A团队中，也发生在其他团队中。一般来说，只要有流程存在，就会有走完流程的等待时间。

##### 团队成员

软件开发的前期，很多开发者会发现产品需求文档中有不少不合理、不清晰、不完整的地方。当开发受到需求不到位的阻碍时，开发者会去找产品经理咨询，但是产品经理还有很多事情要做，有些问题他们自己还不是很清楚，因此他们也无力去澄清所有的问题。开发者要么基于不到位的需求开发出不正确的系统，要么降低开发的速度。这种情况下，产品经理或者需求问题成了制约开发者效率的约束条件。

有些团队中，个别开发者对系统的熟悉程度远远高于其他人，甚至系统的某些模块，只有他一个人知道内幕细节。这会造成如下后果：当他在上班的时候，如果有关于这些模块的相关问题，大家总是想到他，因为只有他能找到问题的解决方案。由于问题太多了，他每天忙到精疲力竭。当他休假了，他还得继续帮忙大家解决问题，毕竟能者多劳。由于找他的人太多，他开始力不从心起来，这导致他要很久才能回答或者解决别人的问题，哪怕这个问题并不难。很多问题都排在长长的等待队列中，大家的时间都被耽误了。显然，这个开发者应付不了这么多的问题，这种状况也成了一大瓶颈。

还有的系统只有一个运维工程师提供支持。一旦团队中出现运维方面的问题，他就要忙碌了。这些问题导致开发、测试都没办法干活，属于优先级高的问题，但他手头上有一堆优先级高的问题，只能一个一个来。其他人等待他回复消息的平均时间是二十分钟，等待他处理问题的平均时间是两个小时。运维成了团队的短板。

##### 项目部署环境

项目部署环境是一些满足项目部署的各种需求的服务器。项目部署环境一般可分为三种：生产环境，测试环境，开发环境。开发环境是程序员用来做开发的环境，具有改动快，稳定性低，配置随意的特点。测试环境是测试人员用来测试的环境，是开发环境到生产环境的过度环境。生产环境是指正式提供对外服务的环境，该环境中的数据都是用户在使用过程中产生的，而不是开发者或者测试人员在测试环节加入的。生产环境是最重要的环境。

开发人员写完代码后，会把代码提交到环境去测试。环境有时候会像开发者的电脑内存一样，成为开发效率的瓶颈。和电脑内存相比，环境往往是多人同时在使用的，因此情况往往更加复杂。环境问题容易成为增加系统复杂度的因素，不光是因为在大规模软件的开发中，服务器的数量太多，还因为环境的使用者太多，造成了环境的更新过快、资源不足、数据紊乱等问题。

环境数量的不合理是第一个问题。环境的数量应该根据开发人员的需要随时变动，而不是一直固定在某个值。如果环境太多，对于开发人员是好事，但是这会造成服务器资源的浪费。如果环境太少，在人多的情况下，很容易造成冲突。因为一个服务器往往只能部署一套应用。如果第一个开发者在这个环境部署了他的版本，第二个开发者如果也想使用该环境进行测试，就只能等第一个开发者用完该环境。否则，第二个开发者就会让自己的版本提前覆盖上一个版本。这不光会带来不满和抱怨，也会带来效率低下的问题。

资源的配置也是一个问题。曾经经历了不少服务器资源不足而导致的线上问题。服务器的资源需要成本，人力成本也得考虑。为了解决这样一个问题，需要耗费各方人员的时间，如果把这些时间用钱来衡量，或许比资源的费用更多。所以，在事先就能充分预测到软件运行的资源需求，并为环境配置好相应的资源是很重要的。这避免了事后补锅。

环境的权限管理是否合理也应该得到重点关注。权限管理的目标是安全和效率并重。权限管理得越严格，环境会越安全，不过效率会降低。权限管理太松，任何角色能随意地对环境做出改动，这会造成安全隐患。权限管理需要有充分的考量，既不影响开发者开发，也不影响程序的正常运行和测试和用户的使用。权限管理还应该让系统问题的解决变得容易而不是复杂。开发人员应该能方便地拿到各种环境的日志。如果不能做到这一点的话，一旦某个环境的应用出了问题，开发人员要花费更多的时间去获取日志并找到问题。开发人员的主要精力应该在分析问题上，而不是在获取环境权限、获取日志上。为了防止开发人员对环境做出更改影响系统的正常运行，不能给与每个开发者所有的权限。应该根据需要，根据开发者的级别，给与不同的权限。这样才能同时保障安全和效率。

环境还可能带来其他问题。很多软件开发方面的突破都是围绕环境而展开的。不论用什么环境，不让环境成为开发的短板，让环境充分为系统服务，进而为用户服务，这才是最主要的。

TOC约束理论的9个原则

*原则1：迫求物流的平衡，而不是生产能力的平衡*

*TOC主张在企业内部追求物流的平衡。它认为生产力的平衡实际是做不到的。因为波动是绝对的，市场每时每刻都在变化，生产能力的稳定只是相对的。所以必须接受市场波动这个现实，并在这种前提下追求物流平衡。所谓物流平衡就是使各个工序都与瓶颈机床同步，以求生产周期最短、在制品最少。*

*原则2:非瓶颈资源的利用程度是由系统的约束决定的*

*系统的约束就是瓶颈，因为系统的产出是由所能经过瓶颈的产品最决定的，即瓶颈制约着产销率。而非瓶颈资源的充分利用不仪不能提高产销率，而且会使库存和运行费增加。非瓶颈资源的利用程度是由瓶颈资源来决定的。非瓶颈资源的使用率一般不应该达到100%.*

*原则3:资源的利用(Utilization)和活力(Activation)*

*按传统的观点，一般是将资源能够利用的能力加以充分利用，所以利用和活力是同义的。按OPT的观点，两者有着重要的区别:因为做可以做出的最大工作(应该做的，即利用)与根据需要能够最大程度做出的工作(能够做的，即活力)之间是明显不同的。所以对系统中非瓶颈资源的安排使用，应基于系统的约束。*

*原则4:瓶颈上一个小时的损失则是整个系统的一个小时的损失*

*一般来说，生产时间包括调整准各时间和加工时间。但在瓶颈资源与非瓶颈资源上的调整准备时间的意义是不同的。因为瓶颈控制了产销率，在瓶颈上中断一个小时，是没有附加的生产能力来补充的。而如果在瓶颈资源上节省一个小时的调整准备时间，则将能增加一个小时的加工时间，相应地，整个系统增加了一个小时的产出。所以，瓶颈必需保持100%的利用，尽量增大其产出。为此，对瓶颈还应采取特别的保护措施，不使其因管理不善而中断或停工。*

*原则5:非瓶颈资源节省的一个小时无益于增加系统的产销率*

*因为非瓶颈资源上除了生产时间(加工时间和调整准备时间)之外，还有闲置时间。节约一个小时的生产时间，将增加一个小时的闲置时间，并不能增加系统的产销率。*

*原则6:瓶颈控制着库存和产销率*

*如果瓶颈存在于企业内部，表明企业的生产能力不足，相应的产销率也受到限制;而如果当企业所有的资源都能维持高于市场需求的能力，那么，则市场需求就成了瓶颈。这时，即使企业能多生产，但由于市场承受能力不足，产销率也不能增加。同时，由于瓶颈控制了产销率，所以企业的非瓶颈应与瓶颈同步，它们的库存水平只要能维持瓶颈上的物流连续稳定即可，过多的库存只是浪费，这样，瓶颈也就相应地控制了库存。*

*原则7:运输批量可以不等于加工批量*

*TOC采用了一种独特的动态批量系统，它把在制品库存分为两种不同的批量形式，即:*

*①运输批量，是指工序间运送一批零件的数最；*

*②加工批量，指经过一次调整准各所加工的同种零件的数量，可以是一个或几个运输批量之和。*

*根据TOC的观点，为了使瓶颈上的产销率达到最大，瓶颈上的加工批量必须大。但另一方面，在制品库存不应因此增加，所以运输批量应该小，即意味着非瓶颈上的加工批量要小，这样就可以减少库存费用和加工费用。*

*原则8:批量大小应是可变的，而不是固定的*

*在TOC中，运输批量是从在制品的角度来考虑的，而加工批量则是从资源类型的角度来考虑的。同一种工件在瓶颈资源和非瓶颈资源七加工时可以采用不同的加工批量，在不同的工序间传送时可以采用不同的运输批量，其大小根据实际需要动态决定.*

*原则9:编排作业计划时考虑系统资源约束*

*提前期是作业计划的结果，而不是预定值。MRP II制定作业计划的方法一般包括以下几个步骤:*

*①确定批量；*

*②计算提前期；*

*③安排优先权，据此安排作业计划；*

*④根据能力限制调整作业计划，再重复前三个步骤。*

#### 约束处理

*TOC的五大核心步骤*

*TOC有一套思考的方法和持续改善的程序，称为五大核心步骤(Five Focusing Steps)，这五大核心步骤是：*

*第一步，找出系统中存在哪些约束。*

*第二步，寻找突破（Exploit）这些约束的办法。*

*第三步，使企业的所有其他活动服从于第二步中提出的各种措施。*

*第四步，具体实施第二步中提出的措施，使第一步中找出的约束环节不再是企业的约束。*

*第五步，回到步骤1，别让惰性成为约束，持续不断地改善。*