编程哲学（一）：愚者无疑，智者多虑

作者：何幻

原文链接：https://zhuanlan.zhihu.com/p/37653499

自从事软件行业以来，我接触到了很多有想法的人，他们的观点令人震撼，让人耳目一新。
然而，这样的好想法却从来没有在大雅之堂，或者在公开交流时被提及。
有想法的人们，总是在仔细考虑，暗自琢磨。

毕竟，**愚者无疑，智者多虑。**

> The whole problem with the world is that fools and fanatics are always so certain of themselves, and wiser people so full of doubts.
> ——Bertrand Rusell

即便如此，我也想要讨论一些“编程哲学”，并不介意被认为愚蠢。
因为，我太愚蠢了，写不出我好代码。

软件需要哲学家，是的，我深以为然。

> 我们需要伟大的思想家——菲奥多尔·陀思妥耶夫斯基、大卫·休谟、亚里士多德、让·保罗·萨特、本·富兰克林、伽利略·伽利莱、伯特兰·罗素和阿尔伯特·爱因斯坦这样的人来指引我们走出软件的黑暗世纪：这个每一寸都深陷黑暗和无知的时代一定会像中世纪一样被人铭记。

即使我们用着不同的编程语言，**表达**的也是同样的想法。
这些想法，是如何组织代码，如何促使软件发展，如何对概念进行抽象，
如何沟通，如何传递我们的知识，等等。

语言本身的影响会随着深入程度而逐渐降低。

> 社会是一个圆锥，每个人都在圆锥的高上面爬。你和同等水平不同领域的人的距离就是你所处平面圆的半径。只要你的水平更高，你接触别的领域的人的距离就会更短。——趣谈:实力把我们推向圆锥顶点

我们想要深入钻研任何领域，都不可避免的进行哲学上的思辨，
考虑很多与具体工作无关的指导思想，建立各种知识之间的联系。

任何事情要想做到极致，就不得不进行一系列的**理性思考**，总结和反思。
而这些思考才是从业者的核心价值。

编程哲学（二）：让我们想个办法

我们在编写代码的时候，如果发现某个处理过程已经在别处写好了，
想拿过来用，就会有所考量。

不能复制过来，因为修改起来费时费力；
也不能轻易建立依赖，因为修改可能会产生意外影响。

是的，**重复会增加冗余，但是复用会增加依赖。**

为此，聪明的人们想到很多办法，
指出了“软件设计原则”，还有人提到了“代码坏味”，
代码质量被重视起来了。

扩展性，高性能，稳定性，可维护性，都是人们追求的目标。
丑陋的设计和肮脏的代码，是不可容忍的。

人们当然知道“适用”才是最好的，当然不想“过度设计”，
当然知道“模式”只是前人总结的一些可选经验。

然而，某些代码我们就是**不能容忍**。

后来，我们从这个层面跳出来了，
看到了代码要解决的**问题**，看到了“方案”和“需求”，
看到了软件怎样被人使用，看到了代码在工程中的作用。

于是，我们学会了如何根据**问题的结构**来组织代码，
根据业务的发展来促进代码的演进，
学会了推动别人来使用软件，为生态做出贡献。

然而，这并没有什么卵用。

我们仍然会遇到新状况，仍然有不能用已有经验去解决的问题，
我们可能会遇到**沟通问题**，遇到**协作问题**，遇到**工程问题**。

这不是一个人的事情，
我们不得不让大家都知道哪里出问题了，才有可能解决它。

我们发现了**人的重要性**，
我们必须营造良好的“氛围”，优秀的人们才会被吸引过来。

我们不得不主动沟通，统一方向，
不得不扩大影响力，促成某件事情落地执行。

我们总有事情要做，没有通用的解决方案，没有“**银弹**”。

很多事情与代码无关，与编程也无关，
工程师们到底应该做什么呢？

有人会说，快速写出可用的代码去实现业务目标就够了。
我承认这是应该做的，但是还有**价值**更大的事情。

这件事情就是，“**想办法**”。
有的人总是没想法，而有的人总是会想办法。

这些，经验、原则、模式、方案、工程、文化，不都是别人想出来的吗？
关键不在于做什么，而在于谁去做它，
不同的人会想出不同的办法。

所以，我认为软件工程师并不仅仅是写代码的人。
而是**发现问题，并思考如何解决**的人。
能发现多大价值的问题并解决它，工程师就能创造多大的价值。

这里提到的问题，并不局限于业务功能，也不局限于代码本身，
不局限于软件工程，更不局限于团队文化。
它们都是问题，都需要想办法。

我们想到的办法如果可以用代码解决，就实现它，
如果不能，那就用别的办法解决它。

我觉得，到了这一步，才能从容的面对丑陋的代码，
面对混乱的项目工程，面对水土不服的文化。

为什么？因为这些问题，
本来就应该是工程师们，想办法去解决的呀。
发现问题，让我们想个办法，然后解决它。

> 陈皓：我给大约 40 多家公司做过相应的技术咨询和解决过很多技术问题，绝大多数公司都是因为性能和稳定性的问题来找我的，我给这些公司解决问题的时候，基本都是这样的 Pattern：一开始，发现都是一些技术知识点的问题；然后，马上进入到系统架构方面方面的问题；当再解决架构问题的时候，我发现，已经是软件工程的问题；而软件工程问题的后面，又是公司管理上的问题；而公司管理的问题，结果又到了人的问题上；而人的问题，又到了公司文化的问题……
> 我能做的是，观察这个公司的业务形态、和相关的思维方式，以及现有的资源和相应的技术实力，帮助他们从技术到管理上缓解或改善现有的问题。
> ——这多年来我一直在钻研的技术

编程哲学（三）：是什么影响了我们的开发效率

工作量是实际工作任务或可达工作任务，
而**工作效率**，一般指工作投入与产出之比。
在进行某项任务时，工作效率是取得的成绩与所用时间、精力、金钱等的比值。
产出大于投入，就是正效率；产出小于投入，就是负效率。

软件是一个神奇的行业，
不同的工作方式，在工作效率上可能会产生 15 倍甚至 100 倍的差距。

因此延长工作时间，变成了一件不是特别重要的事情了，
人们更多考虑的是**如何在有限的时间内效率更高**。

在提高工作效率方面，每个人都有自己的办法。
“不要重复造轮子”就是其中一个，
它使我们看到了重复劳动，这在一定程度上确实提高了我们的工业水平。

然而，另外一些方面，就不是那么直观了。

我经常看到很多人在忙着写代码，却没有意识到，
我们确实有很多事情要做，但是却**未必有那么多代码要写**。
更多的代码，意味着更高的开发成本，测试成本和维护成本。

因此，当我们需要动手实现很多功能的事情，
不妨问一下自己，**为什么我们不得不写这么多东西**。

难道我们真的走在了业界的前沿，做一些发明创造吗？
这个问题的答案通常是“否”。

没有在专业性上保持谦逊

某个领域的专家，会更倾向于喜爱自己所在的领域，
认可自身领域专业性的价值，否则当初就难以成为专家了。

这是一件利弊参半的事，
专业性使得一些工作被巧妙的解决掉，也使得一些工作被解决的过于勉强。

软件也是如此，
只有极少数情况下，用户是不得不需要软件的，
虽然我们听到和感受到的都是他们的确需要。

商业软件要解决的问题，通常在于缓解当前已有的工作压力，
或者说**对现有方案做出改善**，
却很少创造出全新的解决方案，虽然我们不是这么宣传的。

因此，带着专业领域的自豪感，我们很容易绑架用户，
或者帮用户做太多只能由他们做的事情。
这会在不经意间给用户带来新的负担，还会极大的增加软件的功能范围和复杂度。

所以，我理解的专业性，并不是在专业领域给用户寻找方案，
而是**专业性的给用户寻找方案**，结果可能是用户并不需要我们做那么多事情。

没有把自己变成信息源

人们对工程师的认识可能带有成见，
认为工程师一定是内向的，不善言辞的，
因为他们觉得只有这样才会显得更专注。

然而，别人这么认为，并不代表着这样做就是好的，
仅仅代表着如果这么做会给自己带来较小的阻力。

事实上我们应该反思一下，
内向和不善言辞是不是真的有助于自己把工作做好。

沟通问题在任何行业都会存在，并不是软件行业所独有的。
缺乏沟通，人们都被动的接受信息，会降低团队的工作效率。

这件事大家都是知道的，
然而却很少有人肯站出来，主动汇报自己的工作，变成**信息源**。

人们腼腆的不分享自己的成功案例，这可能算是一种谦虚，
但是因为没有机会得到反馈，而坚持自己的错误就很难被定义为谦虚了。

软件工程师需要主动得到工作反馈，确认待解决问题的动向，
向团队汇报自己的工作内容，向显然已经知道答案的同事学习经验。

不要自己扛下所有的事情，不要自己研究。

没有吃自己的狗粮

Eating your own dog food，直译为“吃你自家的狗粮”，也称为 dogfooding，
是一句英语俚语，常用于描述公司（尤指软件公司）使用自己生产的产品这一情况。

好的工匠常常拥有自己的**工具箱**，
工程师也会思考如何利用团队的产出反哺团队自身。

我们有哪些工具是完成业务目标之外的副产品?
哪些副产品可以在后期当做产品来发布的?
我们做事情的方式是不是可以总结下来?

这是产生**技术产品**的一个有效办法，
而那些立志于只产出技术产品的团队，却往往难以存活下来，
因为他们**并不用自己的产品**。

吃自己的狗粮，让我们把一部分注意力放到了**副产品**和**历史积累**上面。
这些积累才是一个团队赖以生存的根基，
也是工作效率不可能被新团队取代的根本保障。

编程哲学（四）：把控间接性

间接性

计算机领域有句名言：
“计算机科学领域的任何问题都可以通过增加一个**间接的中间层**来解决”。

但是**过多的间接性**反而会造成不好的影响，所以人们进行了这样补充，

> ...except for the problem of too many layers of indirection.[1]

间接性指的是，为了达成某个目的，我们可以**先做另外一件事情**，
然后再绕过来解决原始的问题。

间接性在编程工作中很常见，实际在不知不觉中，我们已经使用了它。

> 计算过程是存在于计算机里的一类抽象事物，在其演化进程中，这些过程会去操作一些被称为数据的抽象事物。人们创建出一些称为**程序**的规则模式，以**指导**这类过程的进行。从作用上看，就像是我们在通过自己的写作魔力去控制计算机里的精灵似的。[2]

可见，**程序符号**不同于它们所操纵的**计算过程**，
编写程序是间接性的一种体现，
程序只是软件功能的一种**符号表示**。

我们应该选取合适的符号，用以描述目标系统的软件功能。

在数学史上，区分**符号**以及对符号的**解释**，却花费了很长时间。
人们总是**不由自主的**把符号解释为日常生活中熟悉的概念。

> 这是整个十九世纪数学的最深刻的教训之一。[3]

表达能力

我们经常处于一种**表达能力受限而不自知**的状态。

为了得到编程语言相关的种种商业好处，
经常把自己局限在**某个特定编程语言**范围之内。

我们只用这种语言去“描述”心中想做的“事情”，
结果我们能“描述”的“事情”，
就慢慢局限在了该语言善于“描述”的“事情”范围之中了。

除此之外，写出易读的代码，显然和**写作水平**有关，
好的表达方式，可以把长篇大论平白直叙的“流水账”，
改造成结构清晰发人深省的“文学作品”。

因此，我们应该多从文学作品中学习经验，
训练自己怎样把事情说清楚，
以及在每个层面上把问题展开成什么样的细节程度。

只有在这种情况下，**封装信息**和**隐藏细节**才突然有了意义。

扁平化

重复未必是有问题的，重复的描述细节才有问题。

某些代码显得无比冗余啰嗦，
我们才想到要把它们放在更为细节的层次上。

通常我们会先去构建一些粒度较大的“砖块”，
再用这些“砖块”去搭建主流程，**简化主流程的描述方式**。

然而，过多的细节**层次**也是不恰当的，它增加了我们的**描述复杂度**。
在大型项目中，这些“砖块”本身也会包含很多的细节，由更小的“砖块”组成。
阅读代码的人，必须经常**在不同的层次中上下跳跃**，才能理解我们到底想要表达什么。

这时候，识别出可复用的代码才是关键。

通过分析问题本身的**数学结构**，或者理解项目相关的**业务背景**，
我们可以看到具有逻辑完整性的模式和工具。

把它们提取出去独立放在其他的地方，
可以帮助我们减少当前项目的描述层次，使之扁平化。

参考

1.Indirection
2.计算机程序的构造与解释 - P1
3.哥德尔.艾舍尔.巴赫_集异璧之大成 - P117

编程哲学（五）：未雨绸缪

备选方案

和优秀的人合作，很容易得到理解和体谅，
因为他们对问题本来就有很多种解决方案，也都明白**方案是灵活机变**的。

因此，互相怀疑对方能力，不能理解对方的处境，绝不让步，
这些事永远都不会发生。

当我们面对困难的时候，如果我们除了仅有的一个选择之外别无它法，
这往往不是一个好的状态。
如果**没有权衡的余地**，就只能被动的接受一切，也就没有办法处理任何突发状况。

编写代码也是如此，
如果我们一开始只有一种办法来实现功能，
那么这通常不是一个最优的选择。

实际上，我们应该不遗余力的寻找备选方案，未雨绸缪。

有失才有得

我听过很多人都呐喊着想做出改变，
但是真正主动承受痛苦改变成功的却没有多少人。

究其原因是，仅仅想要做出改变，还远远不够。
任何改变不可能只是带来利益，而不需要付出沉重的代价。

所以，关键不在于人们是否愿意改变自己，
也不在于人们是否能够战胜自己的习惯。
而是在于愿不愿意为不确定的事情付出显而易见的**代价**，是否有能力承担**风险**。

事情就是这样，我们会得到一些自己想要的，
却同时又会失去一些我们不想失去的。

维持任何优雅的代码特征都是需要成本的，
我们都想追求可维护性，可读性，
又要保证性能和质量，还要按时完成，这几乎是不可能的。

我们需要考虑的是，
维持这些特性的成本是否值得投入，以及性价比有多大，
是否有更重要的事情要做。

更上一层楼

当我们视野不够的时候，总是容易表现得小肚鸡肠，对一些细枝末节斤斤计较。
所以，最好先看一下较远的地方，再低下头来审视当前工作的价值。

制定目标的时候也是如此，
先想一下未来的样子，
然后再**将长期计划截断为短期计划**。

在软件行业，新瓶装旧酒的技术方案受制于各种商业因素的影响，此起彼伏层出不穷。
如果看不清发展趋势，我们当前努力打造的代码堡垒，
会被潮流的更替瞬间击垮，凭添太多改造成本。

所以，保持忙碌很重要，
也要谨防忙碌的样子，
正确的做事很重要，也要胆识**做正确的事情**。

意料之外还是意料之中

没有明确的目的，会让我们很难进行取舍。

例如，保留一个已有的问题会节省时间，
但是可能会使这个问题以后更难被解决，该如何选择？
事实上，这取决于我们到底想节省时间，还是想避免以后的麻烦。

只有明确了目的，明确了某些做法是我们有意而为之，才会避免心理上的抵触。
洁癖和**强迫症**才不会干扰我们，世界才能清静下来。

编程哲学（六）：从正确归因到个人影响力

归因问题

归因理论是社会心理学研究的理论之一，它描述了我们怎样解释人们的行为。

我们总是试图将某个人的行为或者某个结果，
归因于性格（内因）或情境（外因）。

我们无休止的分析和讨论**事情为什么发生**，
特别是当我们经历一些消极事件或者预期之外的事件的时候。

归因理论的研究者发现，人们在归因时存在一个普遍性的问题，
当我们解释他人的行为时，会低估**环境**造成的影响，而高估**个人的特质和态度**所造成的影响。这种个体在归因时低估情境因素作用的倾向，被称为**基本归因错误**。

就像当我们看一个演员出演正面或反面角色时，
尽管我们知道这些都不是真实的，
但我们却很容易固执的认为这就是那个人本质的真实反映。

人们习惯性的将自己的失误归因于环境，而将别人的失误归因于他们的内部秉性。

改变他人 vs 改变环境

想要改变他人，是人们表达对环境不适的正常反应。
然而，对每个人来说，本身也是自己所处环境的一部分。
因此，想要改变他人，也是人们缺乏行动力，没办法行动起来的正常反应。

不幸的是，**试图改变他人，其实就是互相伤害。**
有意无意的想改变对方，是矛盾的根源。

改变别人是不可能的，况且**别人也不认为自己需要作出改变。**
所以，如果对环境感到不适的时候，
应该着手从自己做起，做好想让别人去做的事情。

一旦行动起来，自身所处的局部环境就自然发生了改变。
局部环境改变了，才有可能通过环境**影响**他人。

从抱怨到合作

不是别人没有把事情做好，而是我们这些人**都没有把事情做好**。

在软件开发中，人们经常为了完成更多的功能，不得已而降低软件的可维护性和可用性，
其表现形式就是代码缺少注释和文档。

那么应该由谁来完成这些注释和文档的编写工作呢？
很显然代码的作者很难逃避这个责任，
但我想说这并不是一个人的责任，相反，整个团队应该**共同担负**这个责任。

因为不是一个人而是一个团队，交付了软件产品。
我们经常对别人能做而未做，自己也能做却不想做的事情，进行抱怨。

避免抱怨的最好方法就是，行动起来，
从自己能做的事情做起，与团队成员紧密的合作。

只有在竞争中才纠结谁对谁错，合作的时候，只看**共同的表现结果**。

参考

社会心理学（第八版）- 戴维·迈尔斯

编程哲学（七）：我写不出好代码

我们在为别人编程

编程，是一个解决问题的过程，
通过对问题本身进行分析，考虑目前可用的计算资源，
整合出一套自动化的解决问题的步骤，就是程序。

在这个过程中，我们需要发现规律，找到普适性，
以降低软件的成本，覆盖大部分场景，
这是我们每天要做的事情。

我们很少为自己编程，
我们做软件，并不是为了解决自己的问题，而是帮用户，
我们提供了一个代码库，也不是为了自己，而是因为别人要用它。

因此，仅仅创造一个解决方案，这并不够，
还得告诉别人，**该如何使用它，**
编程的艺术或许在于如何提供功能。

这是一件很容易被忽视的事情，
解决方案的创造者，默认是会使用它的，所以很难体会用户的心情。
那种灵活到几乎做什么都行，却没有一种方法可行的感受。

看似简单的事情

天才程序员写的代码库很简单，可重复使用，且功能强大。
我们写的代码库比较复杂，没有人知道该如何使用它。

这并不是一个巧合，
并不是我们刚好遇到了一件麻烦的事情，
而是**我们本来就不知道如何写出好用的代码，我们写不出来。**

我们知道漂亮的代码长什么样，知道如何使用它们，
并不意味着我们也可以写出这样的代码。
脍炙人口的文章人们都想写出来，可是却只有少数人能够办到。

这需要长期的训练以积累经验，需要仔细的斟酌，
需要**考虑如何提供一项功能**，而不仅仅是实现它，
需要把自己当做对系统一无所知，然后再教会自己。

大部分人都不擅长这一点，
毕竟考虑功能该被如何使用，与考虑功能该被如何实现，是两件不同的事情。

无微不至的软件服务

如果我们没办法写出那么好的代码，那就只能多提供一些帮助信息了。

我们不得不写上足够多的**注释**和**文档**，
解释我们为什么要这么做，以及软件的用法。
解释我们正在解决的问题，以及它存在的原因。
这些注释和文档还得与时俱进，随着代码的发展而更新。

我们觉得这些是多余的，是因为我们是代码的作者，
而用户如果没有它们，就寸步难行，
我们需要设计一条学习曲线，让用户慢慢的理解我们的意图。

商业级的软件服务需要**在编码之外做出很多额外的努力**，
除了全面的测试之外，还要有详尽的注释和丰富的文档，
我想这才是和业余项目的根本区别吧。

参考

屋中的大象

编程哲学（八）：偿还不起的技术债务

修改代码的风险

重构，就是在**不改变外部行为**的前提下，有条不紊地改善代码。
为了保障软件的外部行为，唯一的办法就是通过**测试**。
因此，重构是建立在完备的测试覆盖基础之上的。

如果我们不能保证修改后的代码还能提供相同的功能，
那么这种修改就是**错误的**，会给用户带来极大的损失。
在有风险意识的团队中，不会同意发生这样的修改。

什么是所有的功能

一个涉及几百个页面的网站，多个角色处于不同状态的用户都可以访问它，
那么它总共提供了哪些功能？

页面之间的跳转，以及同一个网页为他们展示的不同功能，都是业务逻辑的细节表现。
没有人知道“**所有的功能**”指的是什么，因为太复杂。

代码中的某个分支，看起来似乎用不到，但是可能就是有那百分之一的用户会使用它。
另外某处，为什么这里要向一个莫名的服务器发送请求，很可能必不可少。

某个类到底有没有人在使用它，我们只知道自己的依赖，很难知道谁依赖了我们。
如果不确定谁在以什么方式使用它，就**不能进行修改**。

偿还周期

哪怕我们已经有了完备的测试，如果重构所花费的时间周期太长，还是很危险，
我们不得不在这段时间内，同时应付**重构工作**和**新功能的开发**。

框架迁移就是这样的一个典型例子，如果我们打算把旧框架的功能迁移到新框架，
那么几乎所有的功能，都不得不在新框架下重新开发并测试一遍，
新需求也不得不在旧框架中完成，并且最终还得再迁移过去。

高利贷

我们很容易低估重构的成本，
假设框架迁移需要　个“人日”的工作量，团队中有　人，需要　天才能把事情做完，
则　不等于　。

因为这　天中会有新功能要开发，这些新功能需要　人日的工作量，
于是，人们必须加班，假设人们比原来努力　倍，
则，　，因此　。

如果以前已经在加班了，那么我只能说，真是**太不幸了**。

如果框架迁移需要 100 人日，有 5 个人来完成它，他们都用 1.5 倍的努力进行工作，
则事实上需要 40 天才能完成，而不是 20 天，居然比原来估算的时间多了一倍。

这 40 天中，每人每天必须想办法比原来多做一半的事情，
我们知道，就算加班其实也很难达到这个目标。

这就是技术负债的利滚利效应，也是著名的牛顿问题。
偿还周期越长，所需偿还的债务总量就越多。

试错的代价

重构其实很难进行下去，即使进行下去了也做的很不彻底，
项目中混杂了各个时代的代码遗骸，战场从来没有干净过。

一开始就寄希望于用重构来逐渐解决问题，可能是有害的，
代码中会留下很多做到一半的事情，开发者必须小心谨慎的理解所有技术细节。

快速试错能反映出这种侥幸心理，虽然快速试错的目的是为了降低最终出错的代价，
但是**实际上很难承认自己的确是错了**，人们会想尽办法弥补它，
于是，等我们看到失败时，再退出已经来不及了。

快速试错通常是一个借口，掩饰自己还没有想清楚它。

参考

重构：改善既有代码的设计
测试驱动开发
程序员的职业素养
人月神话

编程哲学（九）：让技能被使用

随着信息化社会的发展，人们接触到的**信息呈爆炸式的增长**，
我们获取知识的途径也越来越多了，网络中的知识更是取之不尽用之不竭。

我们每天都要学习，主动的或者被动的，但是人们的记忆力总是有限的。
人们常说，“我的脑袋实在装不下了”。
就好像我们记住了新知识，旧知识就不得不被忘掉一样。

难道不是吗？我们一直在学习新技术，
结果导致大脑超负荷运转，忘记了以前熟练掌握的内容，
自己非但没有进步，反而退步了。

其实不然，
因为学习不是遗忘旧知识的原因，**不再使用那些旧知识了**才是。

不要怕忘记

心理学家**赫尔曼·艾宾浩斯**通过实验，得到了一条记忆曲线，称为遗忘曲线。
它表明了记忆的保持与时长之间的关系。

我们看到，即使不学习，旧知识也会被慢慢忘记的，
并不是新知识“排挤”出了旧知识，而是**旧知识被自然而然的忘记了**。

要想保持对旧知识的记忆，唯一的办法就是**复习**。
我们需要经常性的回顾那些已经掌握了的，但是目前暂时不用的知识。
以免在用到的时候，它已经变得不可用了。

一个演奏家，应该在**平时**进行多种训练，从而保证演出的品质，
而不是把舞台当做训练场，用以巩固演奏技术。

程序员也是这样，日常工作并不会保证自己不忘记已经学会的编程技能，
经常进行一些恢复性的训练同样也是必要的。

不要懒于实践

有很多技能，我们学了很久了，但总是不能掌握它。
这是什么原因呢？

答案可能是，缺乏实践。
我以前看过很多篇文章提到过，“动手去实践”，都不以为然，
现在发现，“动手去实践”才是掌握一项新技能的秘诀。

新技术只有被不断的使用，我们才能学会**如何用它解决问题**。
盯着它看，最终也只是**知道它可以解决问题**。

“知道可以解决问题”属于“了解”，而“能用它解决问题”就是“掌握”了，
如果你发现始终掌握不了一门新技术，很可能是因为你没有真正去使用过它。

要想掌握一门新技术，就要想办法去**用它**。
编译原理，操作系统，计算机图形学以及数据库，人们戏称为“程序员的四大浪漫”，
他们很难被掌握，是因为普通程序员总是缺乏直接的使用场景。

我们不会经常性的去写一个编译器，或者实现一个操作系统，
只是看别人是怎么做的，就很难掌握它，
自己没有踩过的坑，就没有切身的体会，以后也无从避免再次进入陷阱。

因此，要想真的掌握一门技术，就得去使用它，
创造场景，**弄脏双手**，义无反顾的去使用它。