目錄

目录

How to be a Programmer 中文版

> 原文 https://github.com/braydie/HowToBeAProgrammer
>
> 译文 https://github.com/ahangchen/How-to-Be-A-Programmer-CN
>
> 现在已经生成了gitbook，适合阅读，并且可以方便地导出PDF。
>
> 如果你想要一起来改进这份翻译，可以给我提PR，不过希望翻译时尽量保持作者的原意。
>
> 文章中出现的一些词汇往往有特殊的含义，可以在4-词汇表找到注释。

引言

做一个好的程序员,困难而高尚。将一个软件工程集体愿景变为现实，最困难的地方在于与你的同事和顾客相处。编程很重要，这需要强大的智力和技能。 但在好的程序员看来，相比构建一个让客户和各种各样的同事都满意的软件系统，（纯粹的）编程真的只是小孩子的玩意。在这篇文章里，我尝试尽可能简洁地总结那些当我21岁时，希望别人告诉我的事。

这可能很主观的，所以，这篇文章注定不适用于所有人，并且有的内容有点武断。我尽量写一些程序员在ta的工作中，非常可能会遇到的事情。大部分这些问题以及它们的解决方案在人们的环境中如此普遍，以至于我(说的)可能有点唠叨。尽管如此，我还是希望这篇文章是有用的。

我们在课堂上学习编程。 那些著作: The Pragmatic Programmer [Prag99], Code Complete [CodeC93], Rapid Development [RDev96], 以及 Extreme Programming Explained [XP99] 都传授编程（知识），并阐述做一个好的程序员这个大话题。 在读这篇文章之前，或者就是现在，你当然也应该读一读Paul Graham [PGSite] 和 Eric Raymond [Hacker] 的文章。 但与那些著作不同，这篇文章强调社交问题并且总结了整套我所知的必须的技能。

在这篇文章里，boss这个词指的是任何一个交给你工程去做的人。 除了一些语境外，我会同义地使用交易，公司，集体这些词，比如，交易意味着赚钱，公司意味着现代的工作空间，集体一般是那些你一起工作的人。

欢迎来到这个群体。

目录

- 个人技能
  - 学会Debug
  - 如何分离问题debug
  - 如何移除错误
  - 如何用Log来Debug
  - 如何理解性能问题
  - 如何解决性能问题
  - 如何优化循环
  - 如何处理I/O开销
  - 如何管理内存
  - 如何处理偶现的Bug
  - 如何学习设计技能
  - 如何进行实验
- 团队技能
  - 为什么预估很重要
  - 如何预估编程时间
  - 如何搜索信息
  - 如何把人们作为信息源
  - 如何优雅地写文档
  - 如何在垃圾代码上工作
  - 如何使用源代码控制
  - 如何进行单元测试
  - 没有思路的时候，休息一下
  - 如何决定下班时间
  - 如何与不好相处的人相处

2. 进阶
   - 个人技能
     - 如何保持充满动力
     - 如何才能被广泛信任
     - 在时间和空间之间该如何权衡
     - 如何进行压力测试
     - 如何权衡简洁与抽象
     - 如何学习新技能
     - 学会打字
     - 如何进行集成测试
     - 交流语言
     - 重要的工具
     - 如何分析数据
   - 团队技能
     - 如何管理开发时间

1. 入门

- 个人技能
  - 学会 Debug
  - 如何通过分割问题 Debug
  - 如何移除一个错误
  - 如何使用日志调试
  - 如何理解性能问题
  - 如何解决性能问题
  - 如何优化循环
  - 如何处理 I/O 开销
  - 如何管理内存
  - 如何处理偶现的 Bug
  - 如何学习设计技能
  - 如何进行实验
- 团队技能
  - 为什么预估很重要
  - 如何预估编程时间
  - 如何搜索信息
  - 如何把人们作为信息源
  - 如何优雅地写文档
  - 如何在垃圾代码上工作
  - 如何使用源代码控制
  - 如何进行单元测试
  - 毫无头绪？休息一下
  - 如何决定下班时间
  - 如何与不好相处的人相处

学会 Debug

调试（Debug）是成为一个程序员的基础。调试这个词第一个含义即是移除错误，但眞实的含义是，通过检查来观察程序的运行。一个不会调试的程序员等同于瞎子。

理想主义者认为设计、分析、复杂的理论或其他东西，是更基本的东西，他们不是现实的程序员。现实的程序员不会活在理想的世界里。即使你是完美的，你也需要与在你周围的主要软件公司或组织 (比如 GNU) 的代码，和你同事写的代码打交道。其中大部分的代码以及它们的文档是不完美的。如果不能看透代码的具体执行过程，最轻微的颠簸都会把你永远地抛出去。通常这种执行过程可见性只能从实验获得，也就是，调试。

调试是一件与程序运行相关的事情，而非与程序本身相关。你从主要的软件公司购买一些产品，你通常不会看到（产品背后的）程序本身。但代码不遵循文档的情况（一个常见又特殊的例子：让你整台机器崩掉）或者文档没有提及的情况仍然会出现。更常见的是，你的程序出错了，而你检查代码的时候，却不知道这个错误是怎么发生的。不可避免的，这意味着你做的一些假设不对，或者一些你没有预料到的情况发生了。有时候，神奇的修改源代码的技巧可能会奏效。当它无效时，你就必须调试了。

为了获得一个程序执行过程的可见性，你必须能够执行代码并且从这个过程中观察到什么。有时候这是显而易见的，比如一些正在呈现在屏幕上的东西，或者两个事件之间的延迟。在许多其他的案例中，调试与一些不一定可见的东西相关，比如代码中一些变量的状态，哪一行代码正在被执行，或者一些断言是否持有了一个复杂的数据结构。这些隐藏的细节必须被显露出来。

观察一个正在执行程序的内部的方法通常可按如下分类：

- 使用调试工具（比如IDE中的Debugger）；
- Printlining(戳这里看释义) - 对程序做一个临时的修改，通常是加一些行去打印一些信息;
- 日志 - 用日志的形式为在程序的运行中创建一个永久的视窗。

当调试工具稳定可用时，它们是非常美妙的，但 Printlining 和写日志甚至是更加重要的。调试工具通常落后于编程语言的发展，所以在某些时候它们都可能是无效的。另外，调试工具可能轻微改变程序实际执行的方式。最后，调试有许多种，比如检查一个断言和一个巨大的数据结构，这需要写代码并改变程序的运行。当调试工具可用时，知道如何使用调试工具是一件好事，但学会使用其他两种方式也是至关重要的。

当调试需要修改代码的时候，一些初学者会感到害怕。这是可以理解的，这有点像探索型外科手术。但你需要学会分割代码，让它跳起来，你需要学会在它上面做实验，并且需要知道，你临时对它做的任何事情都不会使它变得更糟。如果你感受到了这份恐惧，找一位导师 - 就是因为许多人在一开始面对这种恐惧的的时候表现的太脆弱，我们因此失去了很多本可以变成优秀程序员的人。

Next 如何通过分离问题空间来 Debug

如何通过分割问题 Debug

调试是有趣的，因为一开始调试是个迷题。它不按套路出牌，你觉得它在这里应该这样，但实际上却是另一回事。很多时候还不仅是这么简单---我给出的任何例子都会被设计来与一些偶尔在现实中会发生的情况相比较。调试需要创造力与智谋。如果说调试有简单之道，那就是在这个谜题上使用分治法。

假如，你创建了一个程序，它会依次执行十件事情。当你运行它的时候，它却崩溃了。但你本来的目的并不是想让它崩溃，所以现在一个谜题扔给你了。当你查看输出时，你可以看到序列里前七件事情运行成功了。最后三件事情在输出里却看不到，所以你的谜题变小了："它是在执行第8、9、10件事的时候崩溃的"。

你是否可以设计一个实验来观察它是在哪件事情上崩溃呢？当然，你可以用一个调试器或者我们可以在第8第9件事后面加一些printlining的语句（或者你正在使用的任何语言里的等价的事情），当我们重新运行它的时候，我们的谜题会变得更小，比如"它是在做第九件事的时候崩溃的"。我发现，把谜题是怎样的一直清楚地记在心里能让我们保持注意力。当几个人在一个问题的压力下一起工作时，很容易忘记最重要的谜题是什么。

调试技术中分治的关键和算法设计里的分治是一样的。你只要从中间开始划分，就不用划分太多次，并且能快速地调试。但问题的中点在哪里？这就是真正需要创造力和经验的地方了。

对于一个真正的初学者来说，可能发生错误的地方好像在代码的每一行里都有。一开始，你看不到一些你稍后开发的时候才会看到的其它纬度，比如执行过的代码段，数据结构，内存管理，与外部代码的交互，一些有风险的代码，一些简单的代码。对于一个有经验的程序员，这些其他的维度为整个可能出错的事情展示了一个不完美但是有用的思维模型。拥有这样的思维模型能让一个人更高效地找到谜题的中点。

一旦你最终划分出了所有可能出错的地方，你必须试着判断错误躲在哪个地方。比如：这样一个谜题，哪一行未知的代码让我的程序崩溃了？你可以这样问自己，出错的代码是在我刚才执行的程序中间的那行代码的前面还是后面？通常你不会那么幸运就能知道错误在哪行代码甚至是哪个代码块。通常谜题更像这个样子的："图中的一个指针指向了错误的结点还是我的算法里变量自增的代码没有生效？"，在这种情况下你需要写一个小程序去确认图中的指针是否都是对的，来决定分治后的哪个部分可以被排除。

Next 如何移除错误

如何移除一个错误

我曾有意把检查程序执行和修复错误分割开来，但是当然，调试也意味着移除 bug。理想状况下，当你完美的发现了错误以及它的修复方法时，你会对代码有完美的理解，并且有一种顿悟(啊哈！)的感觉。但由于你的程序会经常使用其他不具有可视性的、没有一致性注释的系统（比如第三方库），所以完美是不可能的。在其他情况下，可能代码是如此的复杂以至于你的理解可能并不完美。

在修复 bug 时，你可能想要做最小的改变来修复它。你可能看到一些其他需要改进的东西，但不会同时去改进他们。请使用科学的方法去改进一个东西，并且一次只改变一个东西。修复 bug 最好的方式是能够重现 bug，然后把你的修复替换进去，重新运行你的程序，观察，直到 bug 不再出现。当然，有时候不止一行代码需要修改，但你在逻辑上仍然需要使用一个独立原子(译者注：以前人们认为原子不可再分，所以用用原子来代表不可再分的东西)的改变来修复这个 bug。

有时候，可能实际上有几个 bug，但表现出来好像是一个。这取决于你怎么定义 bug，你需要一个一个地修复它们。有时候，程序应该做什么或者原始作者想要做什么是不清晰的。在这种情况下，你必须多加练习，增加经验，评判并为代码赋予你自己的认知。决定它应该做什么,并注释或用其他方式阐述清楚，然后修改代码以遵循你赋予的含义。这是一个进阶或高级的技能，有时甚至比一开始用原始的方式创建这些代码还难，但真实的世界经常是混乱的。你必须修复一个你不能重写的系统。

Next 如何使用日志调试

如何使用日志调试

Logging（日志）是一种编写系统的方式，可以产生一系列信息记录，被称为 log。*Printlining* 只是输出简单的，通常是临时的日志。初学者一定要理解并且使用日志，因为他们对编程的理解是局限的。因为系统的复杂性,系统架构必须理解与使用日志。在理想的状态下，程序运行时产生的日志信息数量需要是可配置的。通常，日志提供了下面三个基本的优点：

- 日志可以提供一些难以重现的 bug 的有效信息，比如在产品环境中发生的、不能在测试环境重现的 bug。
- 日志可以提供统计和与性能相关的数据，比如语句间流逝过的时间。
- 可配置的情况下，日志允许我们获取普通的信息，使得我们可以在不修改或重新部署代码的情况下调试以处理具体的问题。

需要输出的日志数量总是一个简约与信息量的权衡。太多的信息会使得日志变得昂贵，并且造成滚动目盲，使得发现你想要的信息变得很困难。但信息太少的话，日志可能不包含你需要的信息。出于这个原因，让日志的输出可配置是非常有用的。通常，日志中的每个记录会标记它在源代码里的位置，执行它的线程（如果可用的话），时间精度，并且通常还有一些额外的有效信息，比如一些变量的值，剩余内存大小，数据对象的数量，等等。这些日志语句撒遍源码，但只出现在主要的功能点和一些可能出现危机的代码里。每个语句可以被赋予一个等级，并且只有在系统被配置成输出相应等级的记录的时候才输出这个等级的记录。你应该设计好日志语句来标记你预期的问题。预估测量程序表现的必要性。

如果你有一个永久的日志，printling 现在可以用日志的形式来完成，并且一些调试语句可能会永久地加入日志系统。

Next 如何理解性能问题

如何理解性能问题

学习理解运行的程序的性能问题与学习 debug 是一样不可避免的。即使你完美、精确地理解了你的代码运行时所产生的开销，你的代码也会调用其他你几乎不能控制的或者几乎不可看透的软件系统。然而，实际上，通常性能问题和调试有点不一样，而且往往要更简单些。

假如你或你的客户认为你的一个系统或子系统运行太慢了。在你把它变快之前，你必须构建一个它为什么慢的思维模型。为了做到这个，你可以使用一个图表工具或者一个好的日志，去发现时间或资源真正被花费在什么地方。有一句很有名的格言：90%的时间会花费在10%的代码上。在性能这个话题上，我想补充的是输入输出开销的重要性。通常大部分时间是以某种形式花费在 I/O 上。发现昂贵的 I/O 和昂贵的10%代码是构建思维模型的一个好的开始。

计算机系统的性能有很多个维度，很多资源会被消耗。第一种资源是“挂钟时间”，即执行程序的所有时间。记录“挂钟时间”是一件特别有价值的事情，因为它可以告诉我们一些图表工具表现不了的不可预知的情况。然而，这并不总是描绘了整幅图景。有时候有些东西只是稍微多花费了一点点时间，并且不会引爆什么问题，所以在你真实要处理的计算机环境中，多一些处理器时间可能会是更好的选择。相似的，内存，网络带宽，数据库或其他服务器访问，可能最后都比处理器时间要更加昂贵。

竞争共享的资源被同步使用，可能导致死锁和互斥。死锁是由于不恰当的同步和请求资源导致线程执行能力的丧失。互斥是对于资源访问的不恰当安排。如果这是可以预料到的，最好在你的项目开始前就采取措施来地衡量线程争抢。即使线程争抢不会发生，对于有效维护它们也是很有帮助的。

Next 如何修复性能问题

如何修复性能问题

大部分软件都可以通过付出相对较小的努力，让他们比刚发布时快上10到100倍。在市场的压力下，选择一个简单而快速的解决问题的方法是比选择其它方法更为明智而有效率的选择。然而，性能是可用性的一部分，而且通常它也需要被更仔细地考虑。

提高一个非常复杂的系统的性能的关键是，充分分析它，来发现其“瓶颈”，或者其资源耗费的地方。优化一个只占用1%执行时间的函数是没有多大意义的。一个简要的原则是，你在做任何事情之前必须仔细思考，除非你认为它能够使系统或者它的一个重要部分至少快两倍。通常会有一种方法来达到这个效果。考虑你的修改会带来的测试以及质量保证的工作需要。每个修改带来一个测试负担，所以最好这个修改能带来一点大的优化。

当你在某个方面做了一个两倍提升后，你需要至少重新考虑并且可能重新分析，去发现系统中下一个最昂贵的瓶颈，并且攻破那个瓶颈，得到下一个两倍提升。

通常，性能的瓶颈的一个例子是，数牛的数目：通过数脚的数量然后除以4，还是数头的数量。举些例子，我曾犯过的一些错误：没能在关系数据库中，为我经常查询的那一列提供适当的索引，这可能会使得它至少慢了20倍。其他例子还包括在循环里做不必要的 I/O 操作，留下不再需要的调试语句，不再需要的内存分配，还有，尤其是，不专业地使用库和其他的没有为性能充分编写过的子系统。这种提升有时候被叫做“低垂的水果”，意思是它可以被轻易地获取，然后产生巨大的好处。

你在用完这些“低垂的水果”之后，应该做些什么呢？你可以爬高一点，或者把树锯倒。你可以继续做小的改进或者你可以严肃地重构整个系统或者一个子系统。（不只是在新的设计里，在信任你的 boss 这方面，作为一个好的程序员，这是一个非常好的使用你的技能的机会）然而，在你考虑重构子系统之前，你应该问你自己，你的建议是否会让它好五倍到十倍。

Next 如何优化循环

如何优化循环

有时候你会遇到循环，或者递归函数，它们会花费很长的执行时间，可能是你的产品的瓶颈。在你尝试使循环变得快一点之前，花几分钟考虑是否有可能把它整个移除掉，有没有一个不同的算法？你可以在计算时做一些其他的事情吗？如果你不能找到一个方法去绕开它，你可以优化这个循环了。这是很简单的，move stuff out。最后，这不仅需要智慧而且需要理解每一种语句和表达式的开销。这里是一些建议：

- 删除浮点运算操作。
- 非必要时不要分配新的内存。
- 把常量都放在一起声明。
- 把 I/O 放在缓冲里做。
- 尽量不使用除法。
- 尽量不适用昂贵的类型转换。
- 移动指针而非重新计算索引。

这些操作的具体代价取决于你的具体系统。在一些系统中，编译器和硬件会为你做一些事情。但必须清楚，有效的代码比需要在特殊平台下理解的代码要好。

Next 如何处理I/O开销

如何处理I/O代价

在很多问题上，处理器的速度比硬件交流要快得多。这种代价通常是小的 I/O，可能包括网络消耗，磁盘 I/O，数据库查询，文件 I/O，还有其他与处理器不太接近的硬件使用。所以构建一个快速的系统通常是一个提高 I/O，而非在紧凑的循环里优化代码或者甚至优化算法的问题。

有两种基本的技术来优化 I/O：缓存和代表（译者注：比如用短的字符代表长的字符）。缓存是通过本地存储数据的副本,再次获取数据时就不需要再执行 I/O,以此来避免 I/O（通常避免读取一些抽象的值）。缓存的关键在于要让哪些数据是主干的，哪些数据是副本变得显而易见。主干的数据只有一份（在一个更新周期里）。缓存有这样一种危险：副本有时候不能立刻反映主干的修改。

代表是通过更高效地表示数据来让 I/O 更廉价。这通常会限制其他的要求，比如可读性和可移植性。

代表通常可以用他们第一实现中的两到三个因子来做优化。实现这点的技术包括使用二进制表示而非人类可识别的方式,传递数据的同时也传递一个符号表，这样长的符号就不需要被编码，一个极端的例子是哈弗曼编码。

另一种有时能够用来优化本地引用的技术是让计算更接近数据。例如，如果你正在从数据库读取一些数据并且在它上面执行一些简单的计算，比如求和，试着让数据库服务器去做这件事，这高度依赖于你正在工作的系统的类型，但这个方面你必须自己探索。

Next 如何管理内存

如何管理内存

内存是一种你不可以耗尽的珍贵资源。在一段时期里，你可以无视它，但最终你必须决定如何管理内存。

堆内存是在单一子程序范围外，需要持续（保留）的空间。一大块内存，在没有东西指向它的时候，是无用的，因此被称为垃圾。根据你所使用的系统的不同，你可能需要自己显式释放将要变成垃圾的内存。更多时候你可能使用一个有垃圾回收器的系统。一个垃圾回收器会自己注意到垃圾的存在并且在不需要程序员做任何事情的情况下释放它的内存空间。垃圾回收器是奇妙的：它减小了错误，然后增加了代码的简洁性。如果可以的话，使用垃圾回收器。

但是即使有了垃圾回收机制，你还是可能把所有的内存填满垃圾。一个典型的错误是把哈希表作为一个缓存，但是忘了删除对哈希表的引用。因为引用仍然存在，被引用者是不可回收但却无用的。这就叫做内存泄露。你应该尽早发现并且修复内存泄露。如果你有一个长时间运行的系统，内存可能在测试中不会被耗尽，但可能在用户那里被耗尽。

创建新对象在任何系统里都是有点昂贵的。然而，在子程序里直接为局部变量分配内存通常很便宜，因为释放它的策略很简单。你应该避免不必要的对象创建。

当你可以定义你一次需要的数量的上界的时候，一个重要的情况出现了：如果这些对象都占用相同大小的内存，你可以使用单独的一块内存，或缓存，来持有所有的这些对象。你需要的对象可以在这个缓存里以循环的方式分配和释放，所以它有时候被称为环缓存。这通常比堆内存分配更快。（译者注：这也被称为对象池。）

有时候你需要显式释放已分配的内存，所以它可以被重新分配而非依赖于垃圾回收机制。然后你必须谨慎机智地分配每一块内存，并且为它设计一种在合适的时候重新分配的方式。这种销毁的方式可能随着你创建的对象的不同而不同。你必须保证每个内存分配操作都与一个内存释放操作相匹配。（译者注：在C里面，no malloc no free，在C++里面，no new no delete）。这通常是很困难的，所以程序员通常会实现一种简单的方式或者垃圾回收机制，比如引用计数，来为它们做这件事情。

Next 如何处理偶现的 Bug

如何处理偶现的 Bugs

偶现 bug 是一种类似于外太空50足隐身蝎子的东西。这种噩梦是如此稀少以至于它很难观察，但其出现频率使得它不能被忽视。你不能调试因为你不能找到它。

尽管在8个小时后你会开始怀疑，偶现的 bug 必须像其他事情一样遵循相同的逻辑规律。但困难的是它只发生在一些未知的情形。尝试着去记录这个 bug 出现时的情景，这样你可以去推测到底是什么样的可变性。情况可能跟数据的值相关，比如“这只是在我们把*Wyoming*作为一个值输入时发生”，如果这不是可变性的根源，下一个怀疑应该是不合适的同步并发。

尝试，尝试，尝试去在一种可控的方式下重现这个 bug。如果你不能重现它，用日志系统给它设置一个圈套，来在你需要的时候，在它真的发生的时候，记录你猜想的，需要的东西。重新设计这个圈套，如果这个bug只发生在产品中，且不在你的猜想中的话，这可能是一个漫长的过程。你从日志中得到的（信息）可能不能提供解决方案，但可能给你足够的信息去优化这个日志。优化后的日志系统可能花很长时间才能被放入产品中使用。然后，你必须等待 bug 重新出现以获得更多的信息。这个循环可能会继续好几次。

我曾创建过的最愚蠢的偶现 bug 是在用一个函数式编程语言里为类工程做多线程实现的时候。我非常仔细地保证了函数式程序的并发估计， CPU 的充分使用（在这个例子里，是8个 CPU）。我却简单地忘记了去同步垃圾回收器。系统可能运行了很长一段时间，经常结束在我开始任何一个任务的时候，在任何能被注意到的事情出错之前。我很遗憾地承认在我理解我的错误之前，我甚至开始怀疑硬件了。

在工作中我们最近有这样一个偶现的 bug 让我们花了几个星期才发现。我们有一个多线程的基于 Apache™ 的 Java™web 服务器,在维护第一个页面跳转的时候，我们在四个独立线程而非页面跳转线程里，为一个小的集合执行所有的 I/O 操作。每一次跳转会产生明显的卡顿然后停止做任何有用的事情，直到几个小时后，我们的日志才让我们了解到底发生了什么。因为我们有四个线程，在一个线程内部发生这种情况并不是什么大问题，除非所有的四个线程都阻塞了。然后被这些线程排空的队列会迅速填充所有可用的内存，然后导致我们的服务器崩溃。这个 bug 花了我们一个星期去揪出这个问题，但我们仍然不知道什么导致了这个现象，不知道它什么时候会发生，甚至不知道它们阻塞的时候，线程们在干什么。

这表明了有关使用第三方软件的一些风险。我们在使用一段授权的代码，从文本中移除HTML标签。受它的起源的影响，我们把它叫做法国脱衣舞者。尽管我们有源代码（由衷感谢！），我们没有仔细研究它，直到查看我们服务器的日志的时候，我们最终意识到是“法国脱衣舞者”使邮件线程阻塞了。

这个工具在大多数时候工作得很好，除了处理一些长而不常见的文本时。在那些文本里，代码复杂度是 N 的平方或者更糟。这意味着处理时间与文本的长度的平方成正比。正式由于这些文本通常都会出现，所以我们才可以马上发现这个 bug。如果他们从来都不会出现，我们

永远都不会发现这个问题。当它发生时，我们花了几个星期去最终理解并且解决了这个问题。

Next 如何学习设计技能

如何学习设计技能

为了学习如何设计软件，你可以在导师做设计的时候，在他身边学习他的行为。然后学习精心编写过的软件片段（译者注：比如 android 系统中的谷歌官方应用源码）。在这之后，你可以读一些关于最新设计技术的书。

然后你必须自己动手了。从一个小的工程开始，当你最后完成时，考虑为什么这个设计失败了或成功了，你是怎样偏离你最初的设想的。然后继续去着手大一点的工程，在与其他人合作时会更有希望。设计是一种需要花很多年去学习的关于评判的事情。一个聪明的程序员可以在两个月内充分打好这种基础，然后从这里开始进步。

发展出你自己的风格是自然而有用的，但记住，设计是一种艺术，而不是一种技术。人们写的关于这个主题的书都有一种使得它好像是技术的既定的兴趣。不要武断对待特定的设计风格。

Next 如何进行实验

如何进行实验

已故的伟大的 Edsger Dijkstra 曾经充分解释过：计算机科学不是一门实验科学[ExpCS],并且不依赖于电子计算机。当他提出这个观点时，他指的是19世纪60年代。[Knife]

> ...危害已经出现：主题现在已经变成了“计算机科学” - 这实际上，像是把外科手术引用为“手术刀科学” - 这在人们心中深深植入了这样一个概念：计算机科学是关于机器和它们的外围设备的。

编程不应该是一门实验科学，但大多数职业程序员并没有保卫 Dijkstra 对于计算机科学的解释的荣耀。我们必须在实验的领域里工作，正如一部分，但非所有的物理学家做的那样。如果三十年后，编程可以在不进行任何实验的前提下进行，这将是计算机科学的一个巨大成就。

你需要进行的实验包括：

- 用小的例子测试系统以验证它们遵循文档，或者在没有文档时，理解它们的反应；
- 测试一些小的代码修改去验证它们是否确实修复了一个 bug；
- 由于对一个系统不完全的理解，需要在两种不同情况下测量它们的性能表现；
- 检查数据的完整性；
- 对困难的或者难以重现的 bug，收集解决方案中可能提示的统计数据。

我不认为在这篇文章里我可以讲述实验的设计，你会在实践中学习到这方面的知识。然而，我可以提供两点建议：

第一，对你的假设或者你要测试的断言要非常清楚。把假设写下来也是很有用的，尤其是如果你有点迷惑或者与其他人合作时。

第二，你会经常发现你必须设计一系列的实验，它们中的每个都基于对最后一个实验的理解。所以，你应该设计你的实验尽量去提供最多的信息。但不幸的是，这会让实验保持简单变的困难 - 你必须通过经验来提升这种权衡的能力。

Next 团队技能 - 为什么评估很重要

为什么评估很重要

为了尽快获得一个可以高效使用的工作软件系统，不仅需要为开发做计划，还需要为文档，部署，市场做计划。在一个商业工程里，这还需要销售和金融计划。没有对开发时间的预测能力，是不可能高效预测以上这些东西的。

好的估计提供了预测能力。管理者喜欢，而且应该这么做。事实是这不可能，不论是理论上还是实践上，准确预测开发软件所消耗的时间总是被管理者所忽视。我们总是被要求做那些不可能的事情，而且我们必须诚实地面对它。不论如何，不承认这个任务的不可能性也是不诚实的，必要的时候，需要解释。对于评估来说，会产生很多沟通不畅的情况，因为人们令人吃惊地倾向于一厢情愿地认为下面这句话：

> 我估计，如果我确实理解了这个问题，我们在5周内有50%的可能完成任务（如果在此期间没有人干扰我们的话）。

的真实的含义是：

> 我保证从现在开始五个星期内完成任务。

这个常见的解读问题需要你与你的 boss 和客户明确地讨论（就好像把他们当做傻子那样）。重新阐述你的解释，不管对你来讲它们有多么显而易见。

Next 如何估计编程时间

如何评估编程时间

评估需要实践，也需要劳动。因为它需要花如此长的时间，以至于评估评估本身的时间可能是一个好主意，尤其是你被要求去评估一些巨大的事情。

当被要求评估一些比较大的事情的时候，该做的最可靠的事情是先停下来。大多数工程师是充满热情并且是渴望愉悦的，而停下来当然会让他们不开心。但对一个进行中的事情做评估一般是不准确且不可靠的。

停下来，使得考虑一些事情或者为任务重新定型成为可能。如果政策压力允许，这是执行评估的最准确的方式，并且它会产生确实的进度。

在没有时间做调查的时候，你首先应该非常清晰地建立评估的含义。首先重新阐述要评估的内容和你编写的评估的最后部分。在你准备编写评估的时候应该把这项任务分解为一个个更小的循序渐进的任务，并且使每个小任务需要的时间不超过一天（理想情况是每个任务的长度最多为一天）。最重要的事情是不要漏掉任何事情。例如，文档，测试，规划的时间，与其他小组交流的时间，还有度假时间，这些都是很重要的。如果你每天都要花时间和一些傻逼交流，在评估里为这件事情划一个明确的时间界限。这能让你的boss对于你将要花费的最少时间有了一个认识，并且可能给你更多的时间。

我认识一些会隐式地填充评估时间的好的程序员，但我推荐你不要这样做。填充的一个结果是你可能会耗尽别人对你的信任。例如，一个工程师可能为一个将要花费一天的工作评估为三天。这个工程师可能计划花两天去为代码写文档，或者花两天去做一些其他有用的工程。但当任务在一天内完成时，如果它在那天暴露出来的话，这是可以察觉的，并且松懈或高估的表现会出现。为你确实要做的事情做合适的剖析要好得多。如果写文档需要花两倍于编程的时间，并且评估的结果就是这样的，让这对管理者可见就能得到巨大的好处。

相反，显式填充。如果一个任务可能花一天，但如果你的方法没有生效，可能花十天 - 用某种方式在你的评估里记下这个情况，否则，至少为这个可能性，评估一个权重计算可能的时间。任何你可以识别和进行评估的风险因素应该在时间表里被体现。一个人不太可能在给定的任何星期都生病。但一个有很多工程师的大项目可能会有一些疾病时间，还有休假时间。或者，是否会有公司内部的强制培训研讨会的可能性呢？如果这可以预估，也把它算进来。当然，还有一些未知的未知，或者 unk-unk 。Unk-unk 在定义上是不能被独立评估的。你可以尝试为所有 unk-unk 创建一个全局的界线，或者用你与你的 boss 交流好的其他方式去处理它们。然而，你不能让你的 boss 忘记它们的存在。在把评估变成时间表的过程中，把它们遗忘是超级容易的。

在一个团队环境里，你应该让任务的执行者去做这种评估，而且你们应该在团队范围内对评估的结果达成一致。人与人在技术，经验，准备和信心上都有很多的不同。当一个牛逼的程序员为他自己评估了时间，然后一些弱一点的程序员被这种评估约束时，灾难就会降临。整

个团队在一个一行一行的细致的评估计划上取得的一致，阐述了团队的理解，以及允许在策略上对资源的重新分配的机会（比如，把负担从弱一点的团队成员那里移到强一点的成员那里）。

如果有不能评估的大风险，你应该无论如何都要提出来，这是你的责任，这让才能让管理者不会在这个问题上做承诺，以免在风险发生时让管理者难堪。这种情况下，任何需要的事情都有希望被执行来减小这个风险。

如果你可以说服你的公司去使用极限编程,你只需要评估相当小的事情，这也是更加有趣和有效率的。

Next 如何发现信息

如何发现信息

你所搜寻的事情的本质决定了你应该如何去寻找它。

如果你需要客观的而且容易辨认的关于具体事物的信息，例如一个软件的最新补丁版本，可以在Internet搜索，礼貌的询问很多的人，或者发起一个讨论组。不要在网上搜索任何带有观点或主观解释的东西：能够抵达真相的概率太低了。

如果你需要“一些主观的普遍知识”，人们对这些东西已有的思考历史，那就去图书馆吧。例如，想要了解数学，蘑菇或着神秘主义，就去图书馆吧。

如果你需要知道如何做一些琐碎的事情,找两三本关于这个主题的书，仔细阅读。你可以从网络上学到如何做好这些琐碎的事情，比如安装一个软件包。你甚至可以学到一些重要的东西，例如好的编程技术，但相比读一本纸质书的相关部分，你很容易花更多时间在搜索和对结果排序，以及评估结果的权威性。

如果你需要可能没有人知道的信息，例如，“这个新品牌的软件在海量数据的情况下能工作吗”，你仍然必须在网络和图书馆里搜索。在这些选项都完全竭尽后，你可能需要设计一个实验来搞清楚这个问题。

如果你需要一些考虑了某些特殊环境的观点或估值，和一个专家聊聊。例如，如果你想要知道用 Lisp 构建一个现代数据库管理系统是否是一个好主意，你应该和一个 Lisp 专家和一个数据库专家聊一聊。

如果你想要知道它具体是怎样的，比如一个还未发布的在一个特定程序上更快的算法，跟一些在这个领域工作的人聊聊。

如果你想要做一个只有你自己能做的个人决定，比如你是否应该开始某个事业，尝试把一些对这个想法有益和有害的点列出来。如果这没有什么用，做一些预测。假设你已经从各个角度研究了这个想法，并且做了所有该做的准备，在心里列举所有的后果，包括好的和坏的，但你仍可能犹豫不决。你现在应该遵循你自己内心的想法，然后让你的大脑停止思考。大多数可用的预测技术都对决定你内心一半的欲望有作用，因为它们在体现你自己完全多义和随机模式的潜意识都很有用。

Next 如何将人们作为信息源

如何把人们作为信息源

尊重其他每个人的时间，与你的时间相平衡。问别人问题比得到答案能获得更多。人们会从你的存在和倾听特定的问题从你身上学到东西。你也可以用同样的方式从别人身上学习到东西，你可能学到你正在搜寻的东西的答案。这通常比你的问题更加重要得多。

然而，这个问题的价值会减少你在上面做的事情。你毕竟使用了一个人拥有的最珍贵的商品：时间。交流的好处必须与代价相权衡。更进一步，特定的代价和好处在人与人之间都不一样。我强烈相信一个100人的管理者每个月应该花五分钟与他所在的组织的每个人谈话，大概是它们的时间的5%。但十分钟可能太多了，如果他们有1000个员工，5分钟也可能太多了。你与组织中每个人交谈花费的时间取决于他们的角色（而非他们的位置）。你应该和你的 boss 交谈而非和你 boss 的 boss 交谈，但你偶尔也可以和你 boss 的 boss 交谈啦。这可能不太舒服，但我相信你有责任每个月和你的上上级稍微聊聊，什么都行。

基本的规则是，每个与你交谈的人都能稍微受益，他们与你聊得更多，他们能获得的收益越少。你的应该给他们提供这种好处，还有得到与他们交流的好处，平衡这种好处与花费的时间。

尊重你自己的时间是很重要的。如果和一些人聊天，即使这会消耗他们的时间，结果会节省你很多的时间，那么你应该这样做，除非你认为他们的时间在这个因素上，对整个集体，比你的时间更加有价值。

一个奇怪的例子是暑期实习生。一个处于高技术含量位置的暑期实习生不能被期望去完成太多东西；他们可能会把每个人纠缠到地狱。但为什么这是被允许的呢？因为被纠缠的人从实习生身上可以接收到一些重要的东西。他们得到了一点炫耀的机会，他们可能有机会去听到一些新的思想，他们有机会可以从不同的角度去看问题。他们可能会尝试招聘这个实习生，但即使不是这样，他们也获得了很多。

如果你真诚地相信别人有一些东西可以告诉你，无论合适，应该询问他们的意见与智慧。这能让他们高兴并且你可以从他们身上学到一些东西，也可以教会他们一些东西。一个好的程序员不会经常需要销售副经理的建议，但如果你需要，你当然应该询问这个问题。我曾经被要求去倾听一些销售电话以便更好地理解我们的销售员工的工作。这不会耗费超过30分钟，但却让我通过这么小的付出就对公司的销售队伍有了深刻的印象。

Next 如何优雅地写文档

如何睿智地写文档

人生太短，不能写没人会读的废话，如果你写了废话，没人会去读。所以好一点的文档是最好的。经理不会去理解这些东西，因为不好的文档会给他们错误的安全感以至于他们不敢依赖他们的程序员。如果一些人绝对坚持你真的在写没用的文档，就告诉他们“是的”，然后安静的找一份更好的工作。

没有其他事情比精确估计 把好的文档转为放松文档要求的估计 更为有效率。真相是冷酷而艰难的：文档，就像测试，会花比开发代码多几倍的时间。

首先，写好的文档是好的写作。我建议你找一些关于写作的事情，学习，练习他们。但即使你是一个糟糕的写手或者对你需要写文档的语言掌握不好，这条黄金规则是你真正需要的：己所不欲，勿施于人。花时间去确实地思考谁会读你的文档，他们从文档中想要获得的真正的东西是什么，并且你可以如何把这些东西交给他们。如果你这样做，你将会变成一个超过平均水平的文档编写者，和一个好的程序员。

当代码可以自成文档时，与提供文档给非程序员看相反，我认识的最好的程序员们有这样一个普遍的观点：编写具有自我解释功能的代码，仅在你不能通过代码清晰解释其含义的地方，才写注释。有两个好的原因：第一，任何人需要查看代码级别的文档大多数情况下都能够并且更喜欢阅读代码。不可否认的，有经验的程序员似乎比初学者更容易做到这件事，然而，更重要的是，没有文档的话，代码和文档不会是自相矛盾的。源代码最糟糕的情况下可能是错误并且令人困惑的。没有完美编写的文档，可能说谎，这可糟糕一千倍。

负责任的程序员也不能让这件事变得更简单些。如何写自解释的代码？那意味着什么？它意味着：

- 编写知道别人会去阅读的代码(译者注：编写给人看的代码)
- 运用黄金法则
- 选择直接的解决方案，即使你可以更快地获得另一个解决方案
- 牺牲那些可能混淆代码的小的优化
- 为读者考虑，把你珍贵的时间花在让她更加容易阅读的事情上,并且
- 永远不要使用这样的函数名比如 `foo`, `bar`, 或 `doIt`!

Next 如何在糟糕的代码上工作

如何在糟糕的代码上工作

工作在别人写的糟糕的代码上是常有的事。不要把他们想得太糟，直到你用他们的鞋子走路时。他们可能被要求非常自觉地快速完成一些东西来满足时间表的压力。不管之前发生了什么，为了在不清晰的代码上工作，你必须理解它。理解它需要花费一些学习时间，你必须坚持从时间表中某些部分划出一部分时间来做这件事。为了理解它们，你必须读源代码，你可能需要在上面做一些实验。

即使是为你自己，编写文档也是一个好的时机，因为尝试为你的代码编写文档会强迫你从你可能没有考虑过的角度思考，并且完成的文档可能会有用。当你在做这个时，考虑重写部分或所有代码会消耗你什么东西。是否重写一部分代码事实上真的会节省时间？你重写代码后你会更信任它吗？在这里小心你的傲慢。如果你重写它，你处理它会更容易，但下一个必须阅读它的人是否真的更加容易？如果你重写了，测试的负担在哪里？重新测试的需要是否大于可能获得的好处？

在任何对你没有编写的代码的评估中，代码的质量会影响你对风险问题的认识以及一些未知的事情。

铭记抽象和封装是很重要的，这两个程序员最好的工具，对糟糕的代码是特别好用的。你可能不能够重新设计一大块代码，但如果你可以为它增加一定量的抽象，你不用重新在这整团迷雾上工作就可以获一些好的设计所带来的好处。特别的，你可以尝试去隔离尤其糟糕的代码，这样他们就可以被独立重构。

Next 如何使用源代码控制

如何使用源代码控制

源代码控制系统（又称版本控制系统）让你高效地管理工程。他们对一个人是很有用的，对一个团队是至关重要的。它们追踪不同版本里的所有改变，以至于所有代码都未曾丢失，其含义可以归属于改变。有了源代码控制系统，一个人可以自信地写一些而半途而废的代码和调试的代码，因为你修改的代码被仔细地与提交的、官方的即将与团队共享或发布的代码分割开。

我挺晚才开始意识到源代码控制系统的好处，但现在即使是一个人的工程，我也不能离开源代码控制系统。当你们团队在同样的代码基础上工作时，通常它们是必要的。然而，它们有另一个巨大的优点：它们鼓励我们把代码当做一个成长的有机系统。因为每个改变都会被标记为带有名字或数字的修正，一个人会开始认为软件是一种可见的一系列渐进的提升。我认为这对初学者是尤其有用的。

使用源代码控制系统的一个好的技术是一直保持在几天后提交更新。在提交后，一定程度上不活跃，不被调用的代码在几天内都不会完成，因此也不会对其他任何人产生任何问题。因提交错误的代码而降低你队友的开发速度是一个严重的错误，这往往是一种禁忌。

Next 如何进行单元测试

如何进行单元测试

单元测试，对独立的代码功能片段，由编写代码的团队进行测试，也是一种编码，而非与之不同的一些事情。设计代码的一部分就是设计它该如何被测试。你应该写一个测试计划，即使它只是一句话。有时候测试很简单："这个按钮看起来好吗？",有时候它很复杂："这个匹配算法可以精确地返回正确的匹配结果？"。

无论任何可能的时候，使用断言检查以及测试驱动。这不仅能尽早发现 bug，而且在之后也很有用，让你在其他方面担心的谜题得到解决。

极限编程开发者广泛高效地编写单元测试，除了推荐他们的作品，我不能做更好的事情了。

Next 毫无头绪？休息一下

毫无头绪？，休息一下

没有思路时，休息一下。我有时候没有思路时会冥思15分钟，当我回来看问题时，它就神奇地解开了。更大尺度上，一个晚上的睡眠能做到一样的事情，临时切换到其他活动上可能也会有效。

Next 如何决定下班时间

如何识别下班时间

计算机编程是一种活动也是一种文化。不幸的事实是它不是一种看重身心健康的文化。从文化/历史缘由看（例如，在机器空载的晚上工作的需要），还有因为超过市场时间的压力和程序员的缺乏，计算机程序员传统上总是过度工作。我不认为你可以相信你听到的所有故事，但我认为一周工作60小时是常见的，50小时更多的像一个最小值。这意味着实际总是比需要的时间花费得更多。这对一个好的，不仅为他们自己负责而且为他们的同事负责的程序员来说是一个严重的问题。你需要识别什么时候下班，有时候还要建议其他人回家的时间。解决这个问题的固定规则不存在，抚养一个孩子的固定规则也是，出于同样的原因---每个人都是不同的。

一周超过60个小时工作对我来说是非常辛苦的，我可以申请挺短的一段时间（大概是一周），有时候在我的预料中。我不知道对一个人来说一周工作超过60小时是否公平，我甚至不知道40小时是否是公平的。然而，我确定，如果你努力工作，却在你额外工作的时间里获得了很少东西，这是很愚蠢的。对我个人来说，我认为做一个懦夫不是一个程序员该做的事。遗憾的是，事实上，程序员经常被要求做一个懦夫来为一些人表演，例如一个经理想要给总经理留下深刻印象。程序员经常对此屈服，因为他们希望开心，并且不善拒绝，与此相反的有四道防护墙：

- 尽可能与公司里的任何人交流，这样没人可以误导总经理正在发生的事情；
- 学习明确而防御性地评估和规划，让每个人看到时间表的内容以及它的立场；
- 学会拒绝，在必要时作为一个团队拒绝，并且
- 如果必须的话，退出团队

大多数程序员是好的程序员，好的程序员想要做很多东西。为了做到这点，他们需要高效管理他们的时间。从一个问题中兴奋和深陷其中都有一定量的心理惯性。许多程序员发现他们在长久不被打扰的一段时间里能够保持兴奋和集中注意力，这让他们能最好地工作。然而，人们必须睡觉，并且有其他的责任。每个人需要找到一种方式去满足他们的生物节奏和工作节奏。每个程序员需要做任何必须的事情来提供高效的工作周期，比如只参加的某些最关键的会议，以此保留一定的时间。

因为我有孩子，我尝试和他们在晚上相处。我自己最好的工作节奏是工作很长的一天，在办公室或办公室附近睡觉（从家到工作我需要很长的转换时间）然后足够早地回家，在我的孩子们睡觉前与他们相处。我觉得这并不舒服，但这是我可以工作的最好的妥协。如果你得了传染病，回家。如果你有自杀的想法，回家。如果你有超过几秒的凶杀的想法，回家。如果有人有严重的心理障碍或者超出心情低落的心理疾病的标志，把他送回家。如果你由于疲劳变得与平时不同地在某种程度上趋于不诚实或失望，休息一下。不要使用药物缓解疲劳。不要滥用咖啡因。

Next 如何与不好相处的人相处

如何与不好相处的人相处

你可能必须和不好相处的人相处。甚至可能你本身就是一个不好相处的人。如果你是那种与同事和权威人物有许多矛盾的人，你应该珍惜这种独立所暗示的东西，但需要在不牺牲你的智力或原则的前提下提高你的人际交往能力。

在这方面没有什么经验,或者先前生活的行为模式在工作场合的经验不能适用的一些程序员,对这种事情会非常困扰。不好相处的人经常习惯于拒绝，并且与他人相比，他们更不容易受社交压力所影响。关键是合适地尊重他们，而非你可能想做的事，但不要充分地满足他们想要的(译者注：他们想要的往往是过分的)。

程序员必须作为一个团队一起工作。当分歧出现时，它必须用某种方式解决，它不能被长时间挂起。不好相处的人通常是极度聪明的，并且有一些很有用的意见可以发表。不带对这个人的偏见，倾听并理解不好相处的人是至关重要的。失败的交流通常是分歧的基础，但它有时候可以被巨大的耐心移除。尝试冷静诚恳地保持交流，并且不接受任何可能产生更大矛盾的引诱。在一个合理的尝试理解的周期后，再做决定。

不要让一个恶霸强迫你做你所不同意的事情。如果你是老大，做你认为最好的事情。不要为任何个人因素做出决定，并时刻准备好为你的决定做出解释。如果你是一个有着不好相处的同事的团队成员，不要让老大的决定有任何个人影响。如果没有按你的想法发展，全身心地按（已成事实的）另一种方法去做。

不好相处的人能够改变与进步。我曾亲眼目睹这种情况，但这很稀少。然而，每个人都有暂时的高兴与失落情绪。

每个程序员但尤其是领导都会面临这样一个挑战：让不好相处的人保持完全的忙碌。他们比别人更倾向于枯燥的工作，并且更能被动地忍受。

Next 进阶技能

2. 进阶

如何保持活力

创建美丽，有用，聪明的东西的欲望能高度调动程序员的积极性。这是奇妙而令人惊奇的。这种欲望对程序员既不特殊也不普遍，但在程序员中，它是如此强烈而普遍以至于它把程序员与其他角色的人们分割开来。

这有一个现实而重要的推论。如果当程序员被要求做一些既不美丽，也没有用，也不漂亮的事情，他们会斗志低落。虽然可以通过做丑陋的，愚蠢的，无聊的东西赚很多的钱，但最后，乐趣才会为公司赚最多的钱。

很明显，有一些完全由动机技术组织起来的工业适用这里的情况。这些我可以识别的特定的编程中的事情有：

- 为工作使用最好的语言
- 寻找机会去使用新技术，新语言，新科技
- 尝试在每个工程里学习或教授一些东西，即使很小

最后，可能的话，估量个人激励的东西对你工作的影响。例如，修复 bug 时，数一数我完全不感兴趣的 bug 的数目，因为这和仍然存在的 bug 数目是独立的，并且这也是影响我对公司的顾客的增值的最小的可能方式。把每个 bug 和一个高兴的顾客关联起来，是对我个人的激励。

Next 如何被广泛信任

如何被广泛信任

值得信任，才能被信任。你也应该让别人了解你。如果没人了解你，没人会为你投票。跟你亲近的人一起，比如队友，这应该不是一个问题。对你部门或团队以外的人，你通过责任和博知建立信任。有时有人会滥用信任，并要求无理由的赞同。不要害怕，解释因这种赞同会让你必须放弃什么。

不要不懂装懂。与队友以外的人一起时，你必须清除地区分“当下在我脑子里不懂的东西”以及“我曾经没有认识到的东西”。

Next 如何在时间和空间权衡

如何在时间与空间权衡

没有上过大学的话，你也可以成为一个好的程序员，但你不知道基本的计算复杂度理论的话，你不可能成为一个好的进阶程序员。你不需要知道‘O’的定义，但我个人认为你应该理解‘常量时间’，‘nlogn’,'n^2'的区别。你可能可以不靠这方面的知识，凭直觉知道如何在时间和空间之间权衡，但没有这种知识，你将不会有一个和你同事交流的稳固基础。

在设计或理解算法的过程中，算法花费的时间有时候是一个以输入量为自变量的函数。当这种情况发生时，如果运行时间与输入量的对数的 n 倍成正比，我们可以说一个算法的最坏/期望/最好情况运行时间是'nlogn'，这个定义和阐述的方式也可以被应用在数据结构占用的空间上。

对我来时候，计算复杂度理论是美妙的，并且与物理学一样意义深远，并且可能还有很长的路要走！

时间（处理器周期）和空间（内存）可以相互交易。工程是关于妥协的，这就是一个好的例子。它并不总是有条理的的，然而，编码一些东西时更加紧凑可以节省空间，但要以解码时花费更多的处理时间为代价。你可以通过缓存节省时间，也就是，花费空间去存储某些东西的一个本地副本，但要以维持缓存的一致性为代价。你偶尔可以通过把更多信息放在一个数据结构里来节省时间。这通常只会有较小的空间占用，但可能会使算法复杂化。

提高时间空间转换经常把它们中的一个或另一个戏剧性地改变。然而，在你开始做这个工作前，你应该问你自己，你将要优化的是否是最需要优化的？研究算法是有趣的，但你不能让这遮蔽了你的双眼让你看不到这样一个冷酷的事实：优化一些不是问题的问题将不会带来任何明显的区别，但却会造成测试的负担。

现代计算机内存越来越便宜，因为不像处理器时间，你在达到边界前你不能看见它，但这种失败是灾难性的。使用内存也有隐藏的代价，比如你影响了其他需要被保留的程序，以及你分配和释放内存的时间。在你想要花更多空间去换取速度之前，请仔细考虑这一点。

Next 如何进行压力测试

如何进行压力测试

压力测试很有趣，一开始好像压测的目的是找出系统在负载下能不能工作。现实中，系统在负载下确实能工作，但在负载足够重的某些情况下不能工作。我把这叫做碰壁或撞响[1]。可能会有例外，但大多数情况下会有这么一堵“墙”。压测的目的是为了指出墙在哪里，然后弄清楚怎么把墙移得更远些。

压测计划需要在工程的早期就规划好，因为它经常有助于弄清楚到底什么是被期望的。两秒的网页请求是一个悲伤的失败还是一个了不起的成功？500个并发用户是否足够？这，当然，视情况而定，但一个人在设计系统时就应该知道满足需求的答案。压测需要足够好地为现实建模，使之足够有用。非常容易地模拟500个不稳定并且不可预测的人并行使用系统不是真的可能的，但我们可以至少创造500个模拟（用户），然后尝试模拟他们可能做的部分事情。

在压测中，从轻负载开始，然后为系统在一些维度上增加复杂 - 比如输入频率和输入规模 - 直到你抵达那堵墙。如果墙太近了以至于不能满足你的需要，弄明白哪个资源是瓶颈（这通常是那个主要的资源）。它是内存？处理器？I/O？网络带宽？还是数据连接？然后弄明白你可以怎么移动那堵墙。记录下移动墙的那个要素，也就是增加了系统可以处理的负载的那个要素，它可能不能真正在低负载系统下产生危害。但通常重负载下的表现比轻负载下更重要。

你必须能够观察几个不同维度，以此来为之构建一个思维模型；单一的技术是不够的。例如，日志经常是给出系统中两个事件间的挂钟时间的好主意。但除非仔细构建，日志不会给出内存使用的可见性甚至是数据结构的大小。相似的，在现代系统里，大量电脑和许多软件系统是合作的。特别是在你碰到那堵墙时（也就是，表现与输入不成线性比例时），这些软件系统可能成为瓶颈。对这些系统的透视力，甚至仅仅对所有参与工作的机器的处理器做测量，都可能是非常有帮助的。

意识到墙在哪里的关键不仅是移动这堵墙，而且也是提供对其的预测能力。这样公司可以得到更高效的管理。

---

[1] "撞响"

Next 如何在简洁与抽象间平衡

如何在简洁与抽象间平衡

抽象是编程的关键。你应该仔细选择你需要抽象的程度。充满活力的初学者经常创建许多没有什么用的抽象。一个标识是，你是否创建了这样一个类,不包含任何代码并且没有真的做什么事情,除了抽象一些东西。这种抽象是可以理解的，但代码的简洁性的价值必须与代码的抽象价值相权衡。有时候，我们可以看到一种热情的理想主义者犯的错误：在工程的一开始，定义了一大堆的看起来抽象得很美的类，然后他会推测说它们可以处理每一个可能出现的情况。随着项目推进及琐事掺杂进来，这些代码本身变得混乱了。函数体比他们本来该有的样子还要长。空的类是一种写文档的负担，在压力之下，它们会被忽略。如果让花在抽象上的精力去保持其简短，最后的结果应该会更好。这是一种推测编程的形式。我强烈推荐 PAUL gRAHAM[PGSite] 的这篇文章 'Succinctness is Power' by Paul Graham。

有这样一种关于信息封装和面向对象编程的有用技能，但有时候它们被带远了。这些技术让一个人抽象地编码并预计变数。然而，我个人认为，你不应该写太多推测性的代码。例如，在一个对象里用增量器和访问器隐藏一个整数变量是一种可接受的风格，这样变量本身就没有暴露，仅仅暴露了很少的关于它的接口。这确实允许了变量的实现的改变不影响调用代码，并且可能对一个必须提供一个稳定 API 的库编写者是合适的。我不认为这种好处会超过其冗长的代价，特别是当我的团队拥有调用代码并因此可以把调用器重构为比原来的更容易时。四到五行多余的代码会是这种推测性好处的沉重代价。

可移植性也有类似的问题。代码是否应当可移植到不同的电脑，编译器，软件系统或平台？还是简单地传输？我认为，不可移植，短而简单传输的代码比长而可移植的代码要好。把不可移植代码限制在特定的领域是一个想对轻松而且无疑是好的主意。比如一个使用了特定 DBMS 的数据库查询的类。

Next 如何学习新技能

如何学习新技能

学习新技能，尤其是非技术类，是最大的一种乐趣。大多数公司会更加有斗志如果它们明白这对程序员来说是多大的激励。

人类通过做来学。读书和上课是有用的。但你对一个从不写程序的程序员会有任何敬意吗？学习任何技能，你应该把自己放在一个可以练习技能的宽容的位置。学习一个新的编程语言时，在你必须做一个大工程前，试着用它做一个小的工程。学习管理软件项目时，先试着管理一个小的工程。

一个好的导师不是你做事情的替代品，而是比一本书更好的存在。你可以提供什么给一个潜在的导师，作为他的知识的交换？至少，你应该努力学习这样他们的时间才不会被浪费。

试着让你的 boss 给你正规的训练，但必须知道，这通常并不会比把相同量的时间花在用你想学的技能来简单玩耍要好上多少。然而，要求训练比在我们不完美世界里的玩耍时间要容易得多，尽管大量正规训练只是在课程上睡觉，等着晚餐聚会。

如果你领导团队，需要知道他们是怎么学习的，并且通过给他们安排适量的和他们感兴趣的技能的工程来锻炼他们。不要忘记程序员最重要的技能不是技术。让你的团队成员有一个机会去玩，锻炼勇气，诚实，以及交流。

Next 学会打字

学会打字

学会盲打。这是一个进阶技能，因为写代码是如此困难以至于你的打字速度是不太相关的，并且不能削减写代码花费的时间，不管你打字有多好。但是，到了你是一个进阶程序员的时候，你可能花费很多时间在用自然语言给你的同事或他人写东西上。这是对你的责任感是一种有趣的测试，学习这样的东西需要专注的时间，但不怎么有趣。有这样一个传说，当 Michael Tiemann 在 MCC 的时候，人们会站在他的门外面倾听他击键的声音，这种声音是如此的急促以至于难以分辨。

Next 如何做集成测试

如何做集成测试

集成测试是对已经进行单元测试的各个部分的一种整合测试。集成是昂贵的，并且它出现在测试中。你必须把这个考虑到你的预计和时间表里。

理想情况下，你应该这样组织一个项目，使得最后没有一个阶段是必须通过显式集成来进行的。这比在项目过程中，随着事情完成逐渐集成事情要好得多。如果这是不可避免的，请仔细评估。

Next 交流语言

交流语言

在语法系统里，有一些正式定义的，非编程语言但是交流语言的语言，它们为促进交流而非标准而特别设计。2003年，最重要的这种语言有： UML, XML, SQL。你应该熟悉这些东西，这样你就可以很好地交流并且决定什么时候去使用它们。

UML 是一个丰富的用图表描述设计的正式系统。它的美丽之处在于它既虚拟又正式，在作者和观众都了解 UML 的前提下，可以容纳大量的信息。你需要了解它，因为设计有时候就是用这种方式交流的。有一些非常有用的工具可以让制作 UML 图看起来非常专业。在很多情况下，UML 太正式了，我自己会使用更简单的箱子与箭头的风格来设计图标。但我非常确定 UML 对你来说至少跟学习拉丁语一样有用（译者注：国外拉丁语使用很广泛）。

XML 是设计新标准的标准。这不是一个数据间交换的问题的解决方案，尽管你有时候会看到它在这种情况下出现。更进一步，它是一种受欢迎的对大部分数据交换的无聊部分的自动化，也就是，把表现结构化为线性序列，还有将其转回一个结构。它提供了一些漂亮的类型和正确性检查，尽管，又一次，实践中你可能需要的只是其中的一部分。

SQL 是一种非常有力而丰富的数据查询和操作语言，而非一种编程语言。它有许多种类，典型地依赖于产品，但这没有标准核心那么重要。SQL 是关系数据库的巧舌弗兰卡。你可能可以也可能不可以在任何领域从对关系数据库的理解中受益，但你必须对它们和 SQL 的语法和含义有基本的理解。

Next 重型工具

重型工具

随着我们的科技文化的进步，软件技术从不可想象，到研究，到新的产品，到标准化产品，到广泛可用和廉价产品。这些重型工具可以拉动很大的负载，但可能是进阶的，并且需要花大量投资去理解。进阶程序员必须知道如何管理它们以及它们什么时候应该被使用或考虑。

现在在我看来，一些最好的重型工具是：

- 关系数据库；
- 全文搜索引擎；
- 数学库；
- OpenGL;
- XML 解析器；
- 电子表格。

Next 如何分析数据

如何分析数据

当你检查一个商业活动并且发现了把它转换为软件应用程序的需求时，数据分析是软件开发早期的一个过程。这是一个官方的定义，当你，一个程序员，应该集中注意力在写别人设计的东西的代码时，这可能会让你相信数据分析是一种更应该归入系统分析的行为。如果我们严格遵循软件工程范式，这可能是正确的。有经验的程序员会成为设计者，最尖锐的设计者变成商业分析师，因此被冠名去思考所有数据需要，并且给你充分定义的任务去执行。这不完全是对的，因为数据是每种编程活动的核心。不管你在你的程序里做什么，你不是在移动数据就是在修改数据。商业分析师分析的是更大尺度上的需要，软件设计者更加压榨这个比例以至于，当问题在你的桌上落地时，好像你需要做的所有事情是应用聪明的算法，开始移动已经存在的数据。

不是这样的。

不管你开始观察它的是哪个阶段，数据是一个良好设计的应用程序主要考虑的因素，如果你仔细观察一个数据分析师是怎么从客户请求中获取需求的，你会意识到，数据扮演了一个基本的角色。分析师创建了所谓的数据流表，所有的数据源被标记出来，信息的流动被塑造出来。清晰定义了什么数据应该是系统的一部分，设计师将会用数据关系，数据交换协议，文件格式的形式塑造数据源，这样任务就准备好传递给程序员了。然而，这个过程还没结束，因为你（程序员）在这个周密的数据提取过程后，需要分析数据以用最好的可能方式表现任务。你的任务的底线是 Niklaus Wirth，多种语言之父，的金句："算法+数据结构=程序"。这永远不是一个独立的自嗨的算法。每个算法都至少被设计去做一些至少与一段数据相关的事情。

因此，由于算法不会在真空中滚动轮子，你需要分析其他人已经为你标记好的数据和必须写入代码的必要的数据。 一个小例子会使得事情更清楚。实现一个图书馆的搜索程序时，通过你的说明书，用户用类型/作者标题/出版社/出版年份/页数来选择书本。你的程序的中级目标是提供一个合法的 SQL 语句去搜索后端数据库。基于这些需要，你有几个选择：按顺序检查每个控制条件，使用一个 switch 语句，或者几个 if 语句；用一个数据控制数组，把它们与一个事件驱动引擎相连。

如果你的需求也包括提高查询性能，通过确认每个项在一个特殊顺序里，你可能考虑使用组件树去构建你的 SQL 语句。正如你可以看到的，算法的选择依赖于你决定使用或将要创建的数据。这样的决定产生高效算法和糟糕算法间的区别。 然而,效率不是唯一要考虑的因素。你可能在你的代码里使用一打命名变量，让它变得尽可能高效。但这样一段代码可能不能容易地维护。可能为你的变量选择一种合适的容器可以保持相同的速度，此外，在的你同事明年看代码的时候，让他们能够更好地理解代码。更多的，选择一个良好设计的数据结构可能允许他们在不重写代码的前提下，拓展你的代码的功能。长久看来，你对数据的选择决定了你结束代码的工作后，它能工作多久。

让我给你看另一个例子，只是一些思想粮食，让我们假设你的任务是找到字典里超过三位的同字异构词（一个异构词必须在同样的字典里有另一个词）。如果你把这当做一个计算任务，你将会结束于无尽的，尝试找出每个单词的所有组合，然后拿它跟列表里的所有其他单词比较，这样一个无尽的努力中。然而，如果你分析了手头的数据，你会意识到，每个单词可能被一个包含这个词本身以及用它的字母作为 ID 的排序数组的记录所代表，这个蛮力算法可能需要运行几天，而小的那个算法只是一件几秒的事。下次面对一个棘手的问题时，记住这个例子。

Next 团队技能 - 如何管理开发时间

如何管理开发时间

管理开发时间，需要维护一个简明且实时更新的计划。一个工程计划是一个估计，一个时间表，一系列取得进步的里程碑，还有对你的团队或者你的时间在每个任务的估计和安排。这也应该包括你需要记得去做的其他事，比如与质量保障人员见面，准备文档，或者订购设备。如果你在一个团队里，工程计划会是一个共同承认的协议，不论是在开始，还是进行的过程中。

工程计划存在的意义是帮助做出决定，而非展示你是如何组织的。如果一个工程计划太长或者不是最新的，它对做出决定将是无用的。现实中，这些决定通常是关于独立的个人的。计划和你的判断让你决定你是否应当把任务从一个人身上移到另一个人身上。里程碑标识了你的进展。如果你有一个奇妙的工程规划工具，不要被为工程创建一个表面巨大设计（Big Design Up Front）所迷惑，但可以用它保持清晰和实时性。

如果你没有一个里程碑，你应该采取即时的行动，比如通知你的 boss 工程已经滑过的部分中进度的完成。这种估计和时间表可能不会在开始时很完美，这会产生这样一种幻觉，你能够填补工程的上一个部分中错过的日志。你可以。但这很可能是因为你低估了那个部分或者高估了一部分。所以工程进度的完成已经滑过了，不管你是否喜欢。

确保你的计划包括了：内部团队会议，写代码，文档，规划周期活动，集成测试，处理外部关系，疾病，休假，已有工程维护，还有开发环境维护。工程计划可以作为一种为局外人或你的 boss 准备的关于你或你的团队正在做的事情的视图。因为如此，所以它应该是短且及时更新的。

Next 如何管理第三方软件危机

如何管理第三方软件危机

一个工程通常依赖于其不能控制的组织所生产的软件，第三方软件危机是每个相关的人都必须意识到的。

永远也不要把希望放在蒸汽上面。蒸汽是任何所谓的尚未可用然而声称可用的软件。这是最确定的一种破产的方式。仅仅怀疑一个软件公司在某个日期对于某个软件产品的某个特性的承诺是不明智的。更明智的做法是完全忽略它，并且忘记你曾听说过这种事。不要在你的公司使用的任何文档里写下这些东西。

如果一个第三方软件不是蒸汽，它仍然是有风险的，但至少它是一个可以处理的蒸汽。如果你正在考虑使用第三方软件， 你应该早点投入一点精力去评估它。人们可能没听说过，评估三个产品的适合性要花两个星期还是两个月，但这必须尽可能及早做。如果没有合适的评估，集成的代价就不能被准确计算。

理解已有的为某个特殊目的的第三方软件的适用性是非常见仁见智的东西。这是非常客观的，并且通常住在专家心里。如果你发现了那些专家，你可以节省很多时间。很多时候，一个工程会如此完全地依赖于第三方软件，以至于如果集成失败了，工程就失败了。像时间表里写的那样清晰地表达了危机。如果危机不能被尽早消除，试着订一个为意外准备的计划，比如可用的第二方案，或者自己写下功能点的能力。永远不要让时间表依赖于蒸汽。

Next 如何管理咨询师

如何管理咨询师

使用咨询师，但不要依赖他们。他们是神奇的人，非常值得尊敬。因为他们看过许多不同的工程，他们通常比你知道更多具体技术，甚至是编程技术。最好的使用他们的方式是像家教那样用例子教学。

然而，他们通常不能像正常员工那样用相同的感觉融入团队，可能仅仅是因为你没有足够的时间去学习他们的优点和缺点。他们的工资更低。他们更容易离开。如果公司做得好，他们可能得到的更少。有些可能是好的，有些可能与平均水平一致，有些可能挺糟糕，但希望你对咨询师的选择不会像你对雇员的选择那样仔细，这样你会获得更多不好的咨询师。

如果咨询师要写代码，你必须在你使用它们前仔细 review。有着大段带风险而没有被 review 的代码，会让你完成不了工程。事实上这对所有的团队成员都是成立的，但你通常有更多与你接近的团队成员的知识。

Next 如何适量交流

如何适量交流

仔细考虑会议的代价：这花费了随参与者数量倍增的时间。会议有时候是必要的，但越小越好。小会议的交流质量更好，过度浪费的时间更少。如果一个人在会议感到厌烦，把这当做会议应该更小的标识。

非正式交流值得做任何事情去鼓励。更多有用的沟通工作在同事间的午饭可以进行，而非其他的时间。许多公司没有意识到或者不支持这一点，这是一种遗憾。

Next 如何直言异议以及如何避免

如何直言异议以及如何避免

异议是一个做出好决定的绝佳机会，但这需要被谨慎处理。你可能会觉得你充分的表达了你的想法，并且在决定做出前，你的意见已经被听取。这种情况下，没有什么可以再说的，你应该决定你是否要支持这个决定，即使你不同意它。如果你可以在自己不同意的情况下，支持这个决定，就这样说实话。这展示了你是多么有价值，因为你是独立的，不是一个唯唯诺诺之人，同时是一个尊重决定的团队成员。

有时候一个你不同意的决定，会在决策者没有充分听取你的观点前做出。你应该在公司和集体的基础上评估是否应该提出这个话题。如果在你看来这只是一个小错误，这可能不值得重新考虑。如果在你看来这是一个大错，你当然必须提出异议。

通常，这不是一个问题。在一些充满压力的环境下，在一些个人因素下，这会导致事情个人化。例如，一些非常牛逼的程序员缺乏在有好的理由认为一件东西是错的情况下去挑战决议的信心。在最糟的情况下，决策者是不可靠的，并会把这变成一个对权威的挑战。最好记住，这种情况下，人们会用他们大脑中爬虫动物的部分来做出反应。你应该私下提出你的争议，然后尝试展示新的知识是如何改变决议做出的基础的。

不管决议是否被推翻，你必须记住你永远不能说出“我的话撂这了，我早就这样告诉你了”这样的话，因为这个决定已经得到了充分探讨。

Next 判断 - 如何在开发质量和开发时间间权衡

如何在开发质量与开发时间权衡

软件开发总是在工程该做什么与完成工程间妥协。但你可能被要求以牺牲你的工程适用性或商业适用性的方式，去交换工程的开发速度。例如，你可能被要求做一些糟糕的软件工程实践，但这将会导致大量维护问题。

如果这发生了，你的首要任务是通知你的团队，然后清楚地解释降低质量的代价。在这之后，你对这个问题的理解会比你的 boss 的理解还要更清晰。明白将会失去什么以及将要得到什么，以及在这次失去的东西，能在下一轮中得到什么。在这个过程中，由一个好工程提供的可见性应该会很有用。如果用质量换时间影响了质量保证工作，(向你的 boss 和质量保证人员)指出这个问题。如果用质量换时间会导致在之后的质量保证周期中出现更多的 bug，指出来。

如果她仍然坚持，你应该把劣质部分隔离到特殊的你可以在下一个开发周期计划重写或优化的组件中。向你的团队解释这个问题，这样他们可以为此做些计划。

忍者程序员在 Slashdot 写下了这样的格言：

> 记住，一个好的设计会被糟糕的代码实现弹回。如果好的接口和抽象在代码中到处存在，最后的重写会更加痛苦。如果写难以修复的清晰代码很困难，考虑是什么与核心设计冲突的东西导致了这个问题。

Next 如何管理软件依赖

如何管理软件系统依赖

现代软件系统趋向于依赖大量的非直接可控的组件。通过协同与重用，这增加了生产效率。然而，每个组件会带来一些问题：

- 你该如何修复组件中的 bug？
- 组件限制你使用特殊的硬件或软件系统了吗？
- 如果组件完全坏掉了，你该做什么？

某些程度上解耦组件，让它独立可以被移除，总是最好的。如果组件被证明完全不可用，你可能能够使用不同的组件，但你可能必须自己写一个组件。解耦不是可移植性，但这让移植变得简单，这大多数时候是好事。

拥有组件源代码可以把风险降到1/4.有了源代码，你可以更容易地评估它，调试它，找到避免踩坑的方法，并且使得修复更容易。如果你进行修复，你必须把修复的内容提交给组件的拥有者，并且让修改合并到官方发布版中，否则你将不适地必须维护一个非官方版本。

Next 如何判断软件是否太不成熟了

如何判断软件是否太不成熟了

使用其他人写的软件是一种最有效率的构建一个坚实的系统的方法之一。这本不该被排斥，但与此相关的风险必须被检验。最大的一种风险在于，它通过使用变成一个可用产品成熟前的 bug 周期和与软件相关的故障时期。在你考虑将软件系统集成前，不论是你自己写的还是第三方的，考虑它是否足够成熟以使用是非常重要的。这里有十个你应该自问的相关问题：

1. 它是蒸汽吗？（那肯定是不成熟的）
2. 有可用的懂这个软件的人吗？
3. 你是第一使用者吗？
4. 有持续使用的强烈动机吗？
5. 有维护负担吗？
6. 没有当前的维护者的话，它还能用吗？
7. 有至少和它的一半那样好的经验丰富的其他可用途径？
8. 你的团队或公司了解它吗？
9. 你的团队或公司对它满意吗？
10. 即使它不好，你可以雇人在它上面工作吗？

对这些标准的一点考虑论证了良好构建的自由软件和开源软件在减小企业家风险上的巨大价值

Next 如何做一个购买或构建决定

如何做购买还是构建的决定

一个尝试用软件完成一些任务的企业级公司或工程必须不断做所谓的 *buy vs. build* 的决定。这个问题的不幸在两个方面：似乎忽视了不必被购买的开源软件和自由软件。更重要的是，这可能应该被称作获取与集成 *vs.* 购买与集成决定，因为集成的代价需要被考虑。这需要商业上，管理上，工程理解上的大量结合。

- 你的需要与它的设计意图有多接近？
- 对于你购买的软件，你想要怎样的可移植性？
- 评估集成的代价是什么？
- 集成的代价是什么？
- 购买会增加还是减少长期维护代价？
- 构建会把你放在一个你不想要的商业位置吗？

在你构建一些大到足够成为另一整个商品的基础的东西前请三思。这样的想法通常是乐观积极的将会对你的团队做出许多贡献的人提出来的。如果他们的想法很引人注目，你可能会想要改变你的商业计划，但不要在没有周全考虑前就投资一个比你自己的商业还大的解决方案。

在考虑了这些问题后，你可能应当准备两个工程计划草案，一个给购买，一个给构建。这会强迫你考虑集成代价。你也应当考虑两种措施的长期维护代价。为了评估集成代价，你必须在购买软件前对它做一个彻底的评估。如果你不能评估好它，你可以假设购买它会有一个不可预料的风险，你应该以此决定是否购买特定的产品。如果考虑后有几个购买决定，需要花一些精力去评估每个决定。

Next 如何专业地成长

如何专业地成长

承担超过你的权力的责任。扮演你想要扮演的角色。对那些对更大组织的成功做出过贡献以及对你个人提供过帮助的人表示感谢与欣赏。

如果你想成为团队的领导，去激励与团结。如果你想成为一个经理，担起规划的责任。你通常可以在和领导或经理在一起时，舒服地完成这些事情，因为这使得他们可以抽空去承担更大的责任。如果这太多了以至于你不能尝试，一次只做一点点。

评估你自己。如果你想要变成一个好的程序员，询问一些你欣赏的人你怎样才能变成他们那样。你也可以问你的 boss，他可以告诉你的东西会少一些，但对你的事业会有更大的影响。

计划学习新技能的方式，包括琐碎的技术类型，比如学习一个新的软件系统，和困难的社交类型，像漂亮的写作，把它们集成到你的工作中。

Next 如何评估面试者

如何评估面试者

评估可能的员工，却没有得到它应得的能量。一个糟糕的雇佣，就像糟糕的婚姻，是非常糟糕的。每个人首要的一部分精力应该投入到招聘上，尽管这很少发生。

有不同的面试风格。有的是折磨人的，设计用来把候选人放在巨大压力下。这是为了这样一个有用的目的：在压力下折射出性格缺陷和弱点。候选人对待面试官不会比对待他们自己更诚实，而且，人的自欺能力是令人惊奇的。

你应当，最少，对候选人进行两个小时的与口头考核等价的技术技能考核。实践后，你会能够快速了解他们知道什么，快速收缩他们不知道的来标明边界。面试者会尊重这件事情。我有几次听面试者说面试的质量是他们选择公司的一个动机。聪明人会因他们的技能而被雇佣，而非他们之前工作过的地方或他们上了哪个学校或者一些无关紧要的特征。

做这些事情，你也应当评估他们的学习能力，这比他们所知道的要更加重要得多。你也应当留心那些难以相处的人所散发出的火药味。你可能能够在面试后通过比较笔记来识别这一点，但在面试的热烈环境中这很难分辨。人们交流的能力以及与人合作的能力比在最新的编程语言上领先更为重要。

一个读者有“在家”测验面试的经验。这有一个优点是揭露了一些面试者能良好地自我表现但不能写代码 - 这样的人是很多的。我个人没有尝试过这种技术，但这听起来挺合适的。

最后，面试也是一个销售的过程。你应该把你的公司或工程销售给候选人。然而，你是在与程序员谈话，所以不要尝试改变事实。从坏的事物开始讲起，最后以好的事物作为强有力的结束。

Next 怎么决定什么时候使用奇妙的计算机科学

如何决定什么时候使用奇妙的计算机科学

有这样一些，例如算法，数据结构，数学，还有其他极客范的大多数程序员知道但很少使用的东西。实践中，这种奇妙的东西太复杂了，通常是不需要的。例如，当你花费大多数时间在低效的数据库调用上时，提高算法是没有什么用的。不幸的大量编程由让系统相互交流以及使用非常简单的数据结构去构建漂亮的用户界面组成。

高科技什么时候是合适的科技？你什么时候应当打开一本书去找一些东西而非一个毫秒级算法？做这些有时候是有用的，但这需要被小心评估。

对于潜在的计算机技术三个最重要的考虑是：

- 它是否充分封装所以其他低级系统风险和复杂度过量增加以及维护代价很低？
- 好处是否是令人惊奇的（例如，成熟系统的两倍或新系统的十倍？）
- 你能够高效测试和评估它吗？

如果一个充分独立算法使用了些许奇妙的可以减少硬件消耗或增加整个系统的两倍性能表现的算法，不考虑它可能是有罪的。争论这样一个方法的一个关键是，证明风险确实是相当的低，因为目标技术可能被充分研究过了，唯一的话题是集成的风险。在这里一个程序员的经验和评估能够真的协同奇妙的算法让集成变得容易。

Next 如何与非工程师交谈

如何与非工程师交谈

工程师和程序员（尤其是）通常被主流文化认为与他人不同。这意味着其他人与我们不同。与非工程师交流时，这是值得记在心里的，你应该时刻去理解观众。

非工程师聪明，但在创造技术类的东西不像我们那样踏实。我们制造东西。他们销售，处理，统计，管理，但他们在制造上不是专家。他们不像工程师那样擅长团队合作（毫无疑问会有例外。）他们的社交技能在非团队环境里通常像工程师那样，甚至比工程师要好，但他们的工作不总要求他们像我们那样进行亲密，珍贵的交流，以及细致的任务划分。

非工程师可能太渴望以至于不能被取悦和与你亲近，他们可能在不是真的对你满意的时候却对你说“是”，或者是因为他们有点怕你，然后不会对你说实话。

非工程师可以理解技术的东西，但他们不会做那件甚至对我们来讲都很困难的事情 - 技术评审。他们确实理解技术是如何工作的，但他们不能理解为什么一个特定的方法需要花三个月而另一种方法要花三天。（毕竟，程序员对这种估计也感到事多得可怕。）这相当于一个巨大的和他们协作的机会。

与你的团队交谈时，你会不假思索地使用某种程度上的简略表达方式，一种简单的语言更有效率，因为你通常对技术或者特别是你的产品会有许多的共享经验。对于那些没有这些共享经验的人不使用简略表达方式是需要作出一些努力的，特别是你团队内部的人员也在场的时候。这些简略的词汇会让你与那些没有分享到相关经验的人之间构建出一道墙，甚至更糟的是，浪费着他们的时间。

与你的团队一起，基本的假设和目标不需要经常重申，大多数谈话集中于细节。与外人一起，就是另一回事了。他们可能不理解你认为理所当然的东西。由于你把这当做理所当然，并且没有重申它们，使得你们的谈话陷入这样一种情况：你可能认为你们相互理解，但事实上有一个巨大的误解。你应当假设你会有错误的交流，并且仔细观察去找出这样的误解。试着总结或将你说的东西分点，来确保他们能够理解。如果你有机会经常与他们见面，花一点时间询问你是否在有效地交流，以及你可以怎样把它做得更好。如果交流有问题，在对他们失望前，寻找方法去提高你自己的实践。

我喜欢与非工程师工作。这提供了绝大的机会来学习与传授。你可以经常由实例得到关于你的交流的阐述的指引。工程师被训练于从混乱中梳理秩序，从困惑中得到解释，非工程师会因此喜欢我们。因为我们有技术评审，并且通常可以理解商业话题，我们经常可以发现一个简单的方法来解决问题。

很多时候非工程师会出于好意以及一种正确做事情的欲望去规划他们认为会让我们更容易的事情，事实上存在一个仅能通过借助你的技术评审协助外人的观点看到的好得多的方法。我个人喜欢极限编程因为它处理了这种低效，通过与快速评估相结合，使得发现成本与好处最佳结合的方法更加容易。

Next 高级技能

3. 高级

- 技术评审
  - 如何从不可能中找到困难的地方
  - 如何使用嵌入式语言
  - 选择语言
- 明智地妥协
  - 如何与时间压力斗争
  - 如何理解用户
  - 如何得到晋升
- 服务你们的团队
  - 如何发展你们的才能
  - 如何选择工作的内容
  - 如何从你的队友学到最多东西
  - 如何划分问题
  - 如何处理无聊的任务
  - 如何获得对工程的支持
  - 如何发展一个系统
  - 如何有效地交流
  - 如何告诉别人他们不像知道的东西
  - 如何处理管理神话
  - 如何处理组织混乱

如何从不可能中找到困难的部分

解决困难，识别不可能是我们的工作。大多数职业程序员认为，如果有些问题不能从一个简单系统发展而来，或者不能评估，那它就是不可能实现的。然而，根据这个定义，研究本身就是不可能的。大量的工作是困难的，但不必然是不可能的。

这种区别是滑稽的，因为你可能经常被要求做一些事实上不可能的事情，不论是从科学观点还是从软件工程观点。然后你的工作就变成了帮助老板找到一个合理的，仅仅是困难而非不可能的解决方案，去满足他们大部分的需要。当一个解决方案可以被自信地规划且风险可以预料时，它只是困难而已。

砍掉模糊的需求是不可能的，比如“构建一个系统为任何人计算最受欢迎的发型和颜色”。如果需求可以做得更加细致，它就经常会变成仅仅是困难，比如“构建一个系统去计算某个人的发型和颜色，允许他们预览与做出改变，让顾客在原始风格的基础上满意度变大，这样我们就可以赚很多钱”。如果没有清晰的成功的定义，你就不会成功。

Next 如何使用嵌入型语言

如何使用嵌入型语言

把一种编程语言嵌入到一个系统对程序员来讲有着几乎与性一样的魔力。这是一种最具有创造力的可以表现的行为。这使得系统惊人地强大。这也允许你锻炼大多数创造性和有生命力的能力，把系统变成你的朋友。

世界上最好的文本编辑器都有嵌入性语言。这可以被用于预计的观众可以掌握的语言的范围，语言的使用可以变为可选的，正如文本编辑器里那样，这样在一开始可以使用它，而没有其他人必须使用它。

我和许多其他的程序员曾坠入创造特殊目的的嵌入型语言的困境里。我曾经历过两次。已经存在了许多为嵌入型语言设计的语言，在创造一个新的语言前，你应该三思。

使用嵌入型语言前，真实的需要自问的问题是：这种工作与我的观众的文化是一致还是相悖？如果你的目标观众都是非程序员，这会有帮助吗？如果你的目标观众都是非程序员，他们会更喜欢API吗？他会是什么语言？程序员不会想要学习一种新的使用范围很窄的语言，但如果这与他们的文化混在一起了，他们将不会花太多时间去学习它。创造一种新的语言是一种快乐。但我们不应该让这遮蔽了观察用户的双眼。除非你有一些真正原始的需求与想法，为什么不使用一个已存在的语言呢？这样你就可以利用好用户对它已有的这种熟悉了。

Next 选择语言

选择语言

喜欢程序员这份工作的独立的程序员可以为任务选择最好的语言。大多数职业程序员控制不了他们将要使用的语言。通常，这个话题会由执行行政决议而非技术决议的boss说出，他们缺少勇气去提升新型工具，即使他们知道，大多数时候使用最新的知识，最少被接受的工具是最好的。另一些情况下，这种团体中真实的好处，以及拓展一个更大的社区的好处，排除了个人的选择。通常管理者由能够雇用在给定的语言有些经验的程序员驱动。毫无疑问他们服务于他们所追求的，以成为工程或公司的最佳乐趣，而且他们会以此被尊敬。并且，我个人相信，这种最浪费而且错误的常见行为，你很有可能遇到。

但是，当然，事物永远不会是一维的。即使一种核心语言妥协了，超出了你的控制范围，通常的情况是，工具和其他程序可以且应该由另一种不同的语言来编写。如果一种语言需要作为嵌入型的（你通常都要考虑它！），语言的选择很大程度上会依赖于使用者的文化。一个人应该利用好这个问题来为你的公司或工程服务，为任务使用最好的语言，这样可以让工作变得有趣

一门编程语言，如果学习它比学习自然语言还要难，那它真的应该被称为符号。对初学者和一些门外汉来说“学习一门新语言”像是一个令人生畏的任务，但在你掌握了三种语言后，这只是一个熟悉可用的库的简单问题。一个程序员趋向于思考一个由三四门语言组成的一个大杂烩系统，但我认为这样一个系统在很多情况下要比单一语言系统要强大，理由有以下几个方面：

- 不同符号编写的部分间，必要的松耦合存在（虽然可能没有干净的接口）
- 通过独立重写每个组件，你可以轻松地写出一个新的语言平台
- 只有一门语言可能对一个庞大的系统不太适合 - 你的多个模块由多种语言组成，能让你为任务找到正确的工具。

这些效应可能有些只是心理上的，但心理上的东西也很重要。最后要说的是，语言暴政的代价超过了它能提供的所有好处。

Next 明智地妥协 - 如何与时间压力斗争

如何与时间压力做斗争

发布压力是快速推出好产品的压力。这是好的，因为它反映了市场事实，并且在某个意义上是健康的。时间压力是迫使一个产品更快地推出的压力，这是浪费的，不健康的，并且太普遍了。

时间压力的存在是有原因的。给程序员任务的人们没有完全尊重我们的强烈的工作道德以及作为一个程序员的乐趣。可能是因为他们对我们所做的事情，他们相信，要求更快会让我们更加努力工作，使得工程更快完成。这可能确实是对的，但效果很小，损害很大。另外，他们看不到生产软件眞实需要的东西。看不到到，也不能够自己创造，他们能做的唯一的事情是看着发布的压力，然后烦程序员。

与时间压力斗争的方法是简单地把它当做发布压力，实现的方法是让可用劳力与产品间的关系变得透明。提供一个诚实，细致，大部分可理解的对所有相关劳力的估计，是一种最好的实现方式。允许做出好的管理决定以权衡可能的功能也是一个附加的好处。

必须清楚解释的关键是，预算是一种几乎不可压缩的液体。就像你不能把水放进充满的瓶子里，你不能往充满的时间中填入更多任务。某种意义上，程序员永远不会拒绝，但更喜欢说“得到你想要的东西，你会失去什么？”，做出清晰的预算的效果将会是增加对程序员的尊敬。这也是其他职业任意所表现的。程序员的努力工作会被看到。很明显，设置一个不现实的时间表对每个人都是痛苦的。程序员不能被欺骗。要求他们做一些不现实的东西是对他们的不尊重和不道德。极限编程放大了这个问题，并且围绕它构造了一个过程，我希望每个读者能足够幸运去使用它。

Next 如何理解用户

如何理解用户

理解用户以及帮助你的boss理解用户是你的责任。因为用户没有像你一样密切地与你的产品的制造产生联系，他们的表现有点不同：

- 用户通常会做出简短的判断
- 用户有他们自己的工作，他们主要会思考你的产品中小的改进，而非大的改进
- 用户看不到你的产品的整个用户画像

你的责任是找出他们真实需要的东西，而非他们说他们需要的东西，然而，更好的是，在你开始前，建议他们，让他们同意你想做的，就是他们真实想要的东西，但他们可能没有这种愿景。你对你自己的主意的信心是要看情况的。你必须同时与自大和错误的谦逊做斗争去找出什么是人们真实想要的。这两种人，或者同一个人身上两种思维模式，一同和谐工作会给出最好的机会来给出正确的愿景。

你在用户身上花费的时间更多，你就能够更好地理解什么能够真正地成功。你应当尝试在你的用户上尽可能测试你的想法，如果可能的话，你甚至应当和他们一起吃饭。

Guy Kawasaki [Rules] 强调过在倾听之外，观察你的用户的重要性。

我相信，合伙人和咨询师让客户说出他们内心真正想要的东西有巨大的困难。如果你想成为一个咨询师，建议你基于用户清晰的头脑以及他们的钱包来选择客户。

Next 如何得到晋升

如何获得晋升

想要被提升为某种角色，先做那个角色该做的事情。

为了提升到某个位置，找到那个位置期望做的事情，然后去做。

想要得到薪酬的提升，带着信息去协商。

如果你觉得你值得得到提升，与你的boss聊一聊。清楚地问他们你需要做什么才能获得提升，然后努力去做。这听起来很老套，但大多数时候你对你需要做的事情的追求与你boss的想法是不同的。这可能会让你的boss在某些程度上有些失落。

大多数程序员可能在某些形式上对他们的相对能力有夸张的感觉 --- 毕竟，我们不可能都在前10%里！然而，我也见过一些非常不得志的人。人不能期望每个人的评价在什么时候都完美与现实相同，但我认为人们通常在一定程度上是公平的，有这样一个警告：如果别人看不到你的工作，你就得不到欣赏。有时候，因为偶然或个人习惯，有些人可能得不到太多关注。在家努力工作或者与你的团队和boss地理隔离的话，这会变得特别困难。

Next 服务你的团队 - 如何发展才能

如何发展才能

Nietschze夸大了他所说的：

> 杀不死我们的，只会让我们更强大。

你最大的责任是对你的团队负责。你应该非常了解他们中的每个人。你应该激励你的团队，但不要让他们过劳。你通常应该告诉他们他们被激励的方式。如果他们觉得划算，他们会被很好的激励。每个工程中，或者在每个其他的工程里，试着同时用他们建议的以及你认为对他们好的方式去激励他们。激励他们的方法不是给他们更多工作，而是给他们一个新的技能或在团队里扮演一个新的角色。

你应该允许人们（包括你自己）偶尔失败，并且应该为一些失败预留一些时间。如果从未有失败，冒险也就没有意义。如果没有偶然的失败，说明其实你没有足够努力。当一个人失败了，你应该尽可能温柔地对待他，不该把他们当做成功了那样子。

为了让每个团队成员被充分激励，问清楚他们中的每个人，如果他们没有动力的话，他们需要什么才能被充分激励。你可能需要让他们保持不满意的状态，但你需要知道每个人需要的是什么。

你不该因为这样的原因放弃，或者让一些人松懈：他们士气低落或者不满因此故意没有承担分担到的责任。你必须试着让他们充分被激励并且有效率。只要你有耐心，坚持这样做。当你的耐心耗尽时，就解雇他们吧。你不能允许故意不司其职的员工留在团队里，因为这对团队不公平。

通过在公众场合这样说，让你团队中的强大成员清楚地知道他们是强大的。表扬应当公开，批评应当私密。

团队中的强大成员会自然地比弱的成员有更多困难的任务。这是完美而自然的，没人为因此困扰，因为每个人都努力工作。

一个在工资中没有反馈出来的奇怪的事实是，好的程序员比十个糟糕的程序员要有效率得多。这导致了一种奇怪的现象。通常，如果你们的弱程序员不挡道的话，你能跑的更快。如果你这样做了，事实上你在短期能取得更多进度。然而， 你的交易会失去一些重要的好处，叫做对弱小成员的训练，对集体知识的传递，失去强大程序员后的恢复能力。强大的程序员对这种现象应该温和些，并且从各种角度去考虑这个问题。

你经常能够给强大的团队成员有挑战的，但细致描绘的任务。

Next 如何选择工作的内容

如何选择工作的内容

你需要在你个人的需要和团队的需要间权衡，选择需要做工程中的哪个部分。你应该做你最擅长的东西，但是也要试着去找一种方式来激励自己，不是通过承担更多的工作而是通过练习新的技能。领导才能和交流能力比技术能力更重要。如果你非常强大，承担最困难或最有风险的任务，在工程中尽可能早地完成这部分，以此减少风险。

Next 如何最大化利用你的队友

如何让你的队友的价值最大化

为了让你的队友的价值最大化，发展好的团队精神，试着保持每个人的个人挑战与渴望。

为了发展团队精神，文化衫与聚会是有益的，但不如对个人的尊重。如果每个人尊重其他的每个人，没有会想要让其他任何人失望。团队精神产生于人们为团队做出牺牲，优先思考团队的利益而非自己利益的时候。作为一个领导者，在这个方面，没有付出就没有收获。

团队领导力的一个关键是促进团结，这样每个人都会听你的。有时候这意味着允许你的队友犯错。也就是，基于这种团结，如果对项目没有太大的损害，你必须允许你团队的一部分成员用他们自己的方式做事，即使你有很大的信心认为这是一件错事。当这种情况确实发生时，不要同意他们的观点，简单公开地反对之，然后接受这种团结。不要让人觉得你受伤了，或者认为你是被迫的，简单地陈述你不同意，但认为团队的团结是更加重要的。这经常会导致他们反悔。如果他们真的反悔了，不要坚持他们一开始的计划。

如果在你们从所有合适的角度讨论了这个话题后，有个人会反对，简单地告诉他们，你必须做一个决定，并且这就是你的决定。如果有方法去评估你的决定是否是错的，或者它稍后是否是错的，尽可能快速切换，并感激那个对的人。

询问你的团队，包括集体与个人，这样一个问题：他们认为什么能创造团队精神以及创造一个高效的团队。

经常表扬，但不要浪费。尤其是表扬那些反对你且确实值得表扬的人。公开表扬，私下批评。但有这样一种例外：有时候进步或者纠正一个错误但却没有注意到错误的根源，是不能被表扬的，这种进步应该私下表扬。

Next 如何划分问题

如何划分问题

接手一个软件工程并把它分为可以由个人实现的任务是很有趣的。这事应该及早进行。有时候经理可能会认为不考虑个人的项目能够起作用。这是不可能的，因为每个人的生产力是如此广泛地不同。对某个组件有特殊知识的人也经常改变，并且可以对工作效果有一个数量级的影响。

正如一个作曲家认为乐器的音色会其重要作用，或者运动队教练对每个运动员的体能的考虑那样，有经验的团队领导，通常不能够把工程依据团队成员需要承担的角色那样划分成一个个的任务。这是好的团队不容易解散的一个原因。

因此有这样一种危险：人们在锻炼自己的能力时会感到无聊，并且不会提高他们的不擅长的方向或者学习新的技能。然而，如果不被过度使用的话，精通是一个非常有用的生产工具。

Next 如何处理无聊的任务

如何处理无聊的任务

有时候避免对公司或工程的成功至关重要却很无聊的任务是不可能的。这些任务可能真的会降低那些必须执行它们的人的斗志。最好的处理方法是使用或者发扬Larry Wall的程序员懒惰美德。试着找一些方法让计算机去做这个任务，或者帮助你的队友去做这个。用一个程序花一个星期去完成要手动去用一个星期完成的任务能让你懂得更多，并且有时候这是可重用的。

如果所有其他的途径都不能工作，为那些必须做这个无聊任务的人道歉，但无论什么情况，不要让他们去单独完成它。至少安排一个两人团队去做这个事情，并增强健康的团队协作来完成这个任务。

Next 如何为一个工程获取支持

如何为工程获取支持

要给工程获取支持，需要创建并交流一个能够证明这个组织整体的真正价值的愿景。试着让其他人分享他们对你的创造的愿景的观点。这给他们一个理由去支持你并给予你他们的智慧。独立地为你的工程补充关键的支持者。不论在什么可能的地方，展示，但不告诉。如果可能的话，构建一个原型或者一个模型来证明你的主意。一个原型总是有力的，但在软件中，它比任何书面的描述都要高级得多。

Next 如何发展一个系统

如何发展一个系统

树的种子包含了成长的思想，但不完全实现成长体的形式与力量。胚胎会成长。它会变大。它看起来更像成长体，并越来越有用。最终它孕育果实。最后，它死亡并且它的躯体喂养了其他的有机体。

对待软件我们也应当有这样的荣耀。一架桥不是这样的，永远不会有一架婴儿桥，但只是有一座未完成的桥。桥比软件要简单得多。

认识到软件的成长是有益的，因为这允许我们在有一个完美的思维图景前取得有用的进步。我们可以从用户那里获得反馈，并用之纠正这种成长。修剪掉疲软的四肢才能健康。

程序员必须设计一个完成的可分发可使用的系统。但高级程序员需要做的更多。你必须设计一个终于完结系统的成长路线。你的工作是，得到一个想法的萌芽，然后把它尽可能顺利地变成一个有用的人工制品。

因此，你必须模拟最终的结果，用一种工程团队可以为之雀跃的方式去交流。但你也必须和他们交流一条非跳跃式的路径，从他们所知到他们想要成为的地方去。在整个过程中，这棵树必须活着，它不能在什么时候死去，然后又复活过来。

这条路径会是螺旋发展的。里程碑之间永远不会太远，这对于在路上取得进步是有用的。在极端的商业环境里，如果里程碑可以实现并尽早赚钱是最好的。，即使他们离良好设计的端点还有很远。程序员的一个工作就是通过理智的选择用里程碑表示的成长路径来平衡即时的报酬与将来的报酬。

高级程序员对软件，团队，个人的成长有集体责任。

一个读者，Rob Hafernik，在这一节中写下了这样的评论：

> 我认为你过低强调了这里的重要性。不仅是系统，还有算法，用户界面，数据模型，等等。当你工作在一个大的系统中，必须有即时目标相关的可测量的进步时，这些也是至关重要的。没有什么比抵达终点却发现一切都不能工作更加恐怖（看看最近的投票新闻系统的崩溃吧）。我甚至想进一步把这当做自然的法则：没有庞大，复杂的系统可以由碎片实现，这只能由一个简单的系统循序渐进成长为一个复杂的系统。

对此，我们只能回答，要有光！

Next 如何良好地交流

如何有效地沟通

为了良好地沟通，你必须认识到它的困难。它本身就是一种技能。与你交流的人本身是有瑕疵的,这一事实使得沟通变得更加困难。他们不会努力去理解你。他们不善言辞。他们经常过度工作或者无聊，至少，有时候只关注他们自己的工作而非你要发表的长篇大论。上课，练习写作，公共演讲，聆听，这些东西的一个好处是，如果你擅长它们，你可以更容易看到问题所在以及解决方法。

程序员是一种社会动物，他们的生存依赖于与团队的交流。高级程序员是一种社会动物，他们的满意依赖于与团队外的人的交流。

程序员从混沌中带来秩序，一种实现这一目标的有趣方法是从外部的一个提议开始。这能用稻草人或白纸模式或者口头的方式来完成。这种领导对于让团队置身于辩论中有极大的好处。这也把你暴露到批评，或者，拒绝与否定中。高级程序员必须准备好接受这些，因为他有特殊的能力，也因此有特殊的责任。非程序员出身的企业家需要程序员在某些方面提供领导。程序员是思想与现实之间的一部分桥梁。

我没有很好地掌握沟通的技巧，但我正在尝试的是一种四叉路径：在我有了一些有序的主意并且充分准备好后，我试着口头表达，交给人们一张白纸（可能是真实的纸，也可能是电子的）来给他们展示一个demo，然后耐心地重复这个过程。很多次我想过，我们在这种困难的沟通里还是不够耐心。如果你的想法没有马上被接受，你不应该丧气。如果你在准备中投入了能量，没有人因此会看低你。

Next 如何告诉人们他们不想听的东西

如何告诉人们他们不想听的东西

你会经常需要告诉人们一些让他们不舒服的事情。记住，你必须为某种原因才这样做。即使没有什么可以用来解决这个问题，你也该尽早告诉他们，这样他们才能充分警觉。

向别人指出一个问题的最好方法是同时提供一个解决方案。其次的方法是呼吁他们寻求帮助。如果你可能会不被信任，你应该为你的主张寻求支持。

一种你必须说的最不舒服且普遍的事情是“时间不够”。尽责的程序员讨厌这样说，但必须尽早说。没有什么比deadline抵达，不得不推迟进度更加糟糕，即使唯一的行动是通知每个人。更好的做法是，作为一个团队整体来做这件事，如果物理上做不到，至少是精神上这样做。你会想要得到你的团队对你的观点以及你为之所做的事情的支持，团队必须与你共同面对这样的后果。

Next 如何处理管理神话

如何处理管理神话

神话这个词有时候意味着虚构。但这有着更深层的内涵。它也意味着一些宗教内容来解释宇宙和人类与之的关系。管理者倾向于忘记他们作为一个程序员时学到的东西，并且相信某种传说。试着让他们相信这种传说是错的,正如让一个虔诚的宗教信徒从他们的信仰中醒悟过来一样粗鲁而失败。因此，你应该认可这些信仰：

- 文档越多越好。（他们需要文档，但他们不会想要你在这些东西上花时间。）
- 程序员是平等的。（程序员可以按重要程度分类。）
- 分配更多资源给迟来的项目可以让它加速。（与新人的交流的代价大多数时候很繁重并且无用。）
- 程序员的效率可以用一些简单的标准尺度来度量，比如代码行数（如果简洁才是力量，那么代码行数是坏的，而非好的。）

如果有机会，你可以试着解释这些东西，但如果你没有成功，不要觉得悲伤，不要好斗地反对这些神话以致损害了你的声望。每个这样的神话增强了管理者关于他们有一些对正在进行的事情的实际控制的想法。真相是，如果管理者是好的，他们会帮助你，如果他们是坏的，他们会妨碍你。

Next 如何处理组织混乱

如何处理组织混乱

经常会有短暂的组织混乱，比如解雇，收购,IPO，新雇佣，等等。对每个人来说这都是令人不安的，但可能对于那些将自尊建立在能力而非位置上的程序员来讲，这种不安不会那么严重。组织混乱对程序员来讲是锻炼他们的魔力的好机会。因为这是一个集体秘密，在最后我会有所保留。如果你不是一个程序员，就不要再读下去了。

> 工程师有能力创造与维持。

非工程师可以安排人们，但，在典型的软件公司，如果没有程序员的话，他们不能创造与维持任何东西，正如工程师通常不能有效地销售产品或者管理商业。这种力量对于大多数与临时组织混乱相关的问题是一种抵抗。如果你有这种力量，你应当完全忽视这种混乱并当做什么都没发生那样坚持下去。你可能，当然，被解雇，但如果这发生了，你可能因为这种力量得到新的工作。更普遍的，一些紧张的没有这种魔力的人会进入你的空间并告诉你做一些蠢事。如果你真的确信这是一件蠢事，最好的做法是微笑，点头，直到他们走开，然后继续做你认为对公司最好的事情。

如果你是一个领导者，告诉你的员工做一样的事情，告诉他们忽视其他任何人告诉他们的东西。这种行为的过程对你个人是最好的，对你的公司或工程也是最好的。

Next 词汇表

词汇表

这是这篇文章里用到的一些短语的词汇表。它们不一定是人们熟悉的标准含义，Eric S. Raymond曾经编译过一份信息量巨大的词汇表[HackerDict],如果你能理解其中的一些片段，阅读这个词汇表将是惊喜而愉悦的。

unk-unk : unknown-unknown的简写。指的是一些暂时不能被概念化的问题，它们会偷走项目的时间并且阻塞时间表。

boss : 给你任务的人或实体，有些地方可能泛指公众。

printlining : 在严格的临时机制上，在程序中插入一些语句，为调试输出一些程序执行过程中的信息。

logging : 实践中编写程序的一种方式，使得它能够产生可设置的输出以描述它的执行过程。

分治：一种自上而下设计的技术，更重要的是，一种调试的技术，划分问题或谜题为小的问题或谜题。

vapour : 幻觉，而且通常是对还不能出售的软件虚假的承诺，往往不会物质化为任何固定的东西。

boss : 给你设定任务的人，有些时候，也指用户。

tribe : 与你一同为相同目标奋斗的人们。

低垂的水果：轻易能达到的巨大提升。

主办人 :项目的发起人

垃圾：不再需要被放在内存中的对象

商业：一群为财富聚合在一起的人

公司：一群为财富聚合在一起的人

集体：一群与你共享文化亲缘与忠诚的人。

滚动目盲：一种由于有效信息被太多无效信息掩盖导致你不能发现它的效应

挂钟：由挂钟测量的现实中真实的时间，与CPU时间相对。

瓶颈：系统性能最重要的限制/一个可以限制性能的界限。

主线：一个独特的信息块，所有缓存副本都从它继承而来，作为这份数据的官方版本。

分配的堆：一份内存在这样的情况下可以被称为分配了堆：当释放它的机制已经完成时。

垃圾：已经被分配但不再有有效意义的内存。

GC：一个回收垃圾的系统。

内存泄露：无意持有的一系列对象的引用，它们避免了垃圾回收（或者垃圾回收器或内存管理系统中的bug！）导致程序随时间逐渐增加了它的内存占用。

极限编程：一种强调与客户交流以及自动化测试的编程风格。

碰壁：因为耗尽了某种特定的资源导致性能突然大幅度地降级

投机编程：在知道一个东西有用前就把它做出来。

信息隐藏(封装)：通过使用尽可能少暴露信息的接口来让事情保持独立解耦的一种设计原则。

面向对象编程：一种强调在对象内部管理状态的编程风格。

交流语言：一种优先为标准化而非执行设计的语言。

箱子与箭头：一种宽松，非正式的，由箱子和箭头组合而成表达关系的图表制作风格，这与正式的图表方法论，比如UML，相对。

通用语：一种语言是如此受欢迎以至于它成了它的领域中实际上的标准，例如法语一度成为国际外交的手段。

buy vs. build：用来形容购买软件还是自己编写软件这样的选择。

合并工作：需要很少创造力并产生很少风险的工作，合并工作可以被很容易地评估。

编程符号：编程语言的同义词，强调编程语言的数学本质以及它们与自然语言相比的简单之处。

稻草人：一种用来作为技术讨论起点的文档。稻草人也可以引申出火柴人，罐头人，木头人，铁人，等等。

白纸：一种信息文档，通常用来解释或将产品或思想卖给观众而非程序员。

Next 书籍/网站

附录 A - 书目/网站目录

书目

[Rules00] Guy Kawasaki, Michelle Moreno, and Gary Kawasaki. 2000. HarperBusiness. Rules for Revolutionaries: The Capitalist Manifesto for Creating and Marketing New Products and Services.

[RDev96] Steve McConnell. 1996. Microsoft Press. Redmond, Wash. Rapid Development: Taming Wild Software Schedules.

[CodeC93] Steve McConnell. 1993. Microsoft Press. Redmond, Wash. Code Complete.

[XP99] Kent Beck. 1999. 0201616416. Addison-Wesley. Extreme Programming Explained: Embrace Change.

[PlanXP00] Kent Beck and Martin Fowler. 2000. 0201710919. Addison-Wesley. Planning Extreme Programming.

[Prag99] Andrew Hunt, David Thomas, and Ward Cunningham. 1999. 020161622X. Addison-Wesley. The Pragmatic Programmer: From Journeyman to Master.

[Stronger] Friedrich Nietzsche. 1889. Twilight of the Idols, "Maxims and Arrows", section 8..

网站

[PGSite] Paul Graham. 2002. Articles on his website: http://www.paulgraham.com/articles.html. All of them, but especially "Beating the Averages".

[Hacker] Eric S. Raymond. 2003. How to Become a Hacker. http://www.catb.org/~esr/faqs/hacker-howto.html.

[HackDict] Eric S. Raymond. 2003. The New Hacker Dictionary. http://catb.org/esr/jargon/jargon.html.

[ExpCS] Edsger W. Dijkstra. 1986. How Experimental is Computing Science?. http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD988a.PDF.

[Knife] Edsger W. Dijkstra. 1984. On a Cultural Gap. http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD913.PDF.

Next History

附录 B - 历史

迁移到Github

这篇文章已经在github上作为一个仓库创建了，这样它可以很容易地被分享、更新、提高。它是从这里复制过来的。http://samizdat.mines.edu/howto/HowToBeAProgrammer.htm by Braydie Grove。 2016年1月迁移到github。

希望反馈或扩展。

请将你对这篇文章的任何评论发给我，我会考虑所有的建议，大部分都会对这篇文章有所帮助。

这篇文章处于GNU免费文档授权下。这个授权不是专门为文章而设计的。文章通常有连贯的令人信服的服务于一个中心的论据。我希望这篇文章尽量短而易读。

我希望它是说明性的，尽管不是一本教科书，它被划分成许多小节，这样新的章节可以被自由地添加进去。有了这样的倾向，你可以用你觉得合适的方式来扩展这篇文章，且服从这个授权的规定。

可能认为这个文档值得扩展有点自大，但希望生生不息。我会很高兴看到你用以下的方式扩展它：

对每个章节增加一些阅读理解，

增加更多章节，

翻译为其他语言，即使只是一小部分一小部分地翻译，或者

在文字间留下批评或评论，

用不同形式构建的能力：比如palm格式或更好的HTML格式。

如果你向我传达了你的工作，我会考虑把它包括在我的子版本里，遵循这个许可证的规定。你也可以在我的了解之外制作你自己的版本，正如这个协议所说的。

Thank you.

Robert L. Read

原始版本

这个文档的原始版本由Robert L. Read 在2000年制作，并且以电子形式在2002年首发于Samizdat Press(http://Samizdat.mines.edu) 。被Hire.com的程序员所使用。

在这篇文章2003年被Slashdot刊载后，大概有75个人给我发过邮件提过建议与错误修改。我感激他们中的所有人。可能有很多重复，但这些人不是提出来最主要的建议就是第一个找到了我的bug：Morgan McGuire, David Mason, Tom Moertel, Ninja Programmer (145252) at Slashdot, Ben Vierck, Rob Hafernik, Mark Howe, Pieter Pareit, Brian Grayson, Zed A. Shaw, Steve Benz, Maksim Ioffe, Andrew Wu, David Jeschke, 以及Tom Corcoran。

最后，我想感谢Christina Vallery,他的编辑和校对巨大地提高了第二份草稿，还有Wayne Allen，他鼓励我开始了这件事情。

原始作者的简介

Robert L. Read 生活在德克萨斯，奥斯汀, 有一个妻子和两个孩子，他现在是Hire.com的首席工程师。他在那里工作了四年。在这之前他建立了4R科技，为造纸工业生产基于扫描的图像分析质量控制工具。

Rob在1995年在德州大学获得数据库理论方向的计算机博士学位。1987年他在Rice大学获得计算机科学学士学位，在16岁时，他就是一个带薪程序员了。

Next License

Contributions

这个仓库目标是成为一个社区驱动的工程，你的加入会极大地促进这个向导的质量。

我可以做什么贡献？

有很多方式为"How to be a Programmer"做贡献

- 新章节的思想
- 提升已有的章节
- 识别排版错误或其他章节中的问题
- 为章节提供额外的资源链接
- 一般的用于提升工程的建议

贡献者

Github在这里会维护一个所有贡献者的列表contributors

校正与迁移到GitHub

Braydie Grove已经同意作为主编。

Braydie把原始的文档转成了Markdown的形式并创建了这个仓库。