如何成为一名程序员：一份简短、全面且个人化的总结

罗伯特·L·里德 <read@hire.com>

2002年12月16日

献给 Hire.com 的程序员们

Contents

1 Introduction

成为一名优秀的程序员既困fi难又高尚。将一个软件项目的集体愿景真正落地，最困难的部分在于与同事和客户打交道。编写计算机程序很重要，需要极高的智力和技能。但与一名优秀程序员为了让一个软件系统同时对客户以及她部分负责的众多同事取得成功而必须完成的其他一切相比，这其实只是小孩子的把戏。计算机编程可以在课程中教授。优秀的著作《程序员修炼之道》[8]、《代码大全》[3]、《快速开发》[2]以及《极限编程解析》[4]都教授计算机编程以及如何成为一名优秀程序员这一更广泛的议题。在阅读本文之前，当然也应该阅读 Paul Graham[6] 和 Eric Raymond[7] 的文章。本文与那些优秀作品有所ff不同之处在于，它强调社会性问题，并在我看来对所需的全部技能集合进行了全面总结。

这是一件非常主观的事情。因此，这篇文章注定是个人化的，并且带有一定的观点色彩。我将自己 confine 于程序员在工作中极有可能遇到的问题。许多这样的问题及其解决方案对人类处境而言都具有高度的普遍性，因此我可能会显得有些说教。尽管如此，我仍希望这本书能有所帮助。我试图尽可能简明地总结那些我希望在二十一岁时有人向我解释过的事情。

在本书中，术语 *boss* 用来指代分派你项目的人。我将 *business*、*company* 和 *tribe* 这几个词视为同义词，只是 business 带有营利意味，company 指现代职场，而 tribe 通常指与你共享忠诚的人。

欢迎加入部落。

Part I
Beginner

2 Personal Skills

2.1 Learn to Debug

调试是成为一名程序员的基石。这个词的 first 含义是移除错误，但真正重要的含义是 *to see into the execution of a program by examining it*。一个不能 effectively 调试的程序员是盲目的。

理想主义者可能认为设计、分析、复杂性理论或其他什么更加基础，但他们并不是实际工作的程序员。实际工作的程序员并不生活在理想世界中。即使她本人完美无缺，她也被主要软件公司、GNU 之类的组织以及同事编写的代码所包围，并且必须与这些代码交互。大多数代码都不完美，文档也不完善。若没有能够洞察这些代码执行状况的能力，哪怕最轻微的波折都可能让她彻底受阻。而这种可见性往往只能通过实验来获得，也就是调试。

调试关注的是程序的运行，而不是程序本身。如果你从一家大型软件公司购买软件，通常是看不到程序代码的。但仍然会出现代码与文档不一致的地方（让整台机器崩溃是一个常见而又壮观的例子），或者文档对此只字未提。更常见的情况是，程序员制造了一个错误，检查自己写的代码，却完全不知道这个错误是如何产生的。不可避免地，这意味着她所做的某个假设并不完全正确，或者出现了她未曾预料到的某种条件。有时，盯着源代码看的“魔法技巧”会奏效；当它不起作用时，她就必须进行调试。

要深入了解程序的执行情况，必须能够运行代码并观察其某些方面。有时这些是可见的，例如屏幕上显示的内容，或两个事件之间的延迟。在许多其他情况下，则涉及一些本不打算被看到的事物，例如代码内部某些变量的状态，哪些代码行实际上被执行了，或者在复杂的数据结构中某些断言是否成立。这些隐藏的内容必须被揭示出来。

查看正在执行的程序内部的常见方法可以分为以下几类：

- 使用调试工具，
- printlining — 对程序进行临时的修fi改，通常是添加用于输出信息的代码行，而 • logging — 则通过日志的形式为程序的执行创建一个永久的窗口。

调试工具在能够正常工作时非常有用，但另外两种方法甚至更为重要。调试工具往往滞后于语言的发展，因此在任何特定时间点都可能尚不可用。由于调试工具可能以细微的方式改变程序的执行过程，在某些情况下它们并不实用。最后，还有一些类型的调试，例如针对大型数据结构检查断言，无论如何都需要编写代码。了解如何在调试工具稳定时使用它们是有益的，但能够运用另外两种方法则至关重要。

一些初学者在修改并执行代码这一意义上的调试方面，内心存在一种潜意识的恐惧。这是可以理解的—它有点像探索性手术。但初学者必须学会戳一戳代码，让它“跳起来”。他们必须学会在代码上做实验，并且明白无论他们对它做什么，都不会让事情变得更糟。如果你是这些胆怯者的老师或导师，请通过温和地向他们示范如何去做，并在必要时牵着他们的手，帮助他们克服这种恐惧。在这个脆弱的起步阶段，我们因为这种恐惧而失去了许多优秀的程序员。

2.2 How to Debug by Splitting the Problem Space

调试很有趣，因为它始于一个谜题。你以为它应该做某件事，但它却做了别的事。事情并不总是这么简单—，与实践中有时发生的情况相比，我能给出的任何例子都会显得刻意。调试需要创造力和巧思。如果说调试只有一个关键，那就是对这个谜题运用*divide and conquer*技术。

假设例如我们创建了一个程序，它应该按顺序完成大约 10 件事情。当我们运行它时却崩溃了。我们并没有把“崩溃”编程进去，所以现在我们有了一个谜团—\It crashes." 通过查看输出，我们可以看到它完成了前面的 first #7。最后三件事从输出中是看不到的，因此现在我们的谜团变小了：\It crashed on thing #8, thing #9, or thing #10"。我们能否设计一个实验来看看它究竟是在做哪一件事时崩溃的呢？当然可以。我们可以使用调试器，或者在 #8 和 #9 之后添加 printline 语句（或者你所使用的语言中的等价方式）。然后我们再运行一次，谜团就会进一步缩小，比如：\It crashed on thing #9." 我 find，始终牢记在任何时刻谜团究竟是什么，有助于保持专注。当几个人在压力之下一起处理一个问题时，事情很容易变得相当混乱。

将分而治之作为一种调试技术的关键，与其作为算法设计方法时是一样的。只要你在中间把这个谜题拆分得足够好，就不需要拆分太多次，调试也会进行得很快。但谜题的“中间”究竟在哪里呢？这正是真正的创造力和经验发挥作用的地方。对于真正的初学者来说，可能错误的空间看起来就像源代码中的一行行代码。她还没有形成后来会发展的那种视野，无法看到程序的其他维度，例如已执行代码行的空间、数据结构、内存管理、与外部代码的交互、存在风险的代码以及简单的代码。这些其他维度使得有经验的程序员能够形成一个并不完美但非常有用的心智模型，用以理解

所有可能出错的事情。拥有这种思维模型能够帮助人们有效地 find 解开谜团的核心。

一旦你均匀地细分了所有可能出错的空间，你必须尝试决定错误在哪个空间里。在简单的情况下，谜题是“导致崩溃的那一行是在这行执行之前，还是执行之后？”你只需要观察哪些行被执行，然后就完成了。在其他情况下，你对谜题的细分可能更像是“要么图中的某个指针指向了错误的节点，要么是我用来累加图中变量的算法有问题。”在这种情况下，你可能需要写一个小程序来检查图中的指针是否都正确，以便决定哪个部分的细分谜题可以被排除。

2.3 How to Debug Using a Log

日志记录是编写一个系统的实践，使其生成一系列称为日志的信息记录。打印日志的实践仅仅是生成一个简单且通常是临时的日志。通常，日志记录意味着一个永久的并且可能是可配置的日志，提供以下优势：

- 日志可能提供有关难以重现的错误的有用信息（例如那些在生产环境中发生但无法在测试环境中重现的错误）。
- 日志可以提供统计和性能信息。
- 如果可配置，日志可能允许捕获一般信息以调试一个特定问题，如果仅在临时基础上进行，可能很难为每个问题恢复。

日志中输出的信息量始终是信息与简洁性之间的折衷。信息过多会使日志变得昂贵且难以使用，信息过少则可能无法包含所需的信息。在这方面，使得输出信息量可配置非常有用。通常，日志中的每条记录都会标识其在源代码中的位置，执行该记录的线程（如果这是一个问题），执行的精确时间，通常还会包含一条额外的有用信息，例如某个变量的值、剩余内存量、数据对象的数量等。这些日志语句分布在源代码中，特别是在主要功能点和风险较大的代码周围。每条语句都可以分配一个级别，只有在系统当前配置为输出该级别时，才会输出记录。应该尽量预测可能出现问题的地方，并设计日志语句以解决这些问题。测量特定子系统性能的需求通常很容易预测，并且非常适合放入日志中，因为它允许在每个环境中收集性能数据。

如果你有一个永久日志，那么现在可以基于日志记录来进行打印插桩，而且其中一些调试语句很可能会被永久地加入到日志系统中。

2.4 How to Understand Performance Problems

理解如何了解正在运行的系统的性能是不可避免的，其原因与调试相同。即使你编写的代码在性能方面考虑得再完美，施加在其上的需求也会发生变化，所使用的硬件会变化，而它所接口的软件系统在性能方面可能是隐藏的，甚至会带来意想不到的情况。然而，在实践中，性能问题与一般的调试相比略有不ff同，也稍微更容易一些。

假设你认为一个系统或子系统太慢了。在试图让它更快之前，你必须建立一个关于它为何缓慢的心智模型。为此，你必须知道时间或其他资源真正花在了哪里。一个 profiling 工具或一份良好的日志对此非常有用。有一句著名的格言：90% 的时间会花在 10% 的代码上。不过，我还要补充输入/输出开销（I/O）在性能问题中的重要性。往往，大部分时间以某种方式花在 I/O 上。找到昂贵的 I/O 和代价高昂的 10% 代码是一个很好的 first 步骤。

计算机系统的性能有许多维度，也会消耗许多资源。要衡量的 first 个资源是墙钟时间（wall-clock time），即计算过程中流逝的总时间。然而，这可能并非全貌。有时，耗时稍长但不消耗那么多处理器秒的方案，在你实际需要应对的计算环境中会好得多。同样地，内存、网络带宽、数据库或其他服务器访问，最终可能比处理器秒要昂贵得多。

对同步的共享资源的争用可能导致死锁和饥饿。死锁是由于不当的同步或资源需求而无法继续执行的情况。饥饿是指未能对某个组件进行恰当调度。如果可以预见，最好在项目一开始就具备一种衡量这种争用的方法。即使这种争用并未发生，能够充满信fi心地断言这一点也非常有帮助。记录墙钟时间尤其有价值，因为在其他分fi析不切实际的情况下，它可以反映出在不可预测情境中出现的状况。

2.5 How to Fix Performance Problems

大多数软件项目都可以在代码完成日期（即所有代码完全可用的最早日期）时的基础上，通过相对较少的 effort，将速度提升到原来的十倍到一百倍。在上市时间压力下，选择一种简单而快速地完成工作的解决方案既明智又 effective，尽管它的 efficiently 低于其他一些解决方案。然而，性能是可用性的一部分，而且往往最终必须更加仔细地加以考虑。

提升一个非常复杂系统性能的关键在于充分分析它，以便find出其中的瓶颈。这些消耗了大部分资源的地方随后就可以被直接改进。去优化只占计算时间1% 的函数并没有太大意义。你应该先做一次性能分析first，以find出时间真正花在了哪里，然后再据此决定要改进什么。作为一条经验法则，在动手之前你应该仔细思考，除非你认为这会让系统或其中一个 significant 的部分至少快一倍。通常总有办法做到这一点。只有在你已经认真思考过那些能带来巨大收益的改进之后，才去处理任何性能提升不到一倍的工作。这样做的一个原因是要考虑你的改动所需要投入的测试和质量保证 effort。每一次改动都会带来测试负担，因此做少数几次大的改动要好得多。

在某件事情上实现了两倍的改进之后，你至少需要重新思考，甚至重新分析，以发现系统中下一个最粗糙的环节，并针对它下手，从而获得另一个两倍的改进。

通常，性能上的粗糙之处就是“数牛腿再除以四”而不是直接“数牛头”的典型例子。比如，我曾经犯过这样的错误：没有在我经常查询的一列上为关系型数据库系统提供合适的索引，这很可能至少让它慢了二十倍。其他例子还包括在内层循环中进行不必要的 I/O、保留已经不再需要的调试语句、不必要的内存分配，尤其是对库和其他子系统的不熟练使用，而这些库和子系统在性能方面往往文档欠缺。这类改进有时被称为“低垂的果实”，意思是可以轻松摘取，从而带来一些 benefit。

当你开始用尽那些“唾手可得的果实”时，你会怎么做？嗯，你可以去够更高的地方，或者把树砍倒。你可以继续进行小幅改进，或者认真地重新设计一个系统或子系统。（这是一个发挥你作为优秀程序员技能的绝佳机会，不仅体现在新的设计上，也体现在说服你的老板这是个好主意上。）在你为重新设计一个子系统据理力争之前，你应该先问问自己，是否能让它变得five到十倍更好。

2.6 How to Optimize Loops

有时你会遇到一些运行时间很长的循环或递归函数，它们会成为你产品中的性能瓶颈。你也许可以让这个循环稍微快一点，但不妨花几分钟想想，是否有办法将它彻底移除。换一种 different 的算法能做到吗？能否在计算其他东西的同时把它算出来？如果你不能 find 出绕开它的办法，或者你选择不这么做，那么就可以优化这个循环。这很简单；把 stuff 移到外面。最终，这不仅需要巧思，还需要理解每一种语句和表达式的开销。不过，这里有一些建议：

- 移除floating点运算。

• 内联子程序调用。• 不要不必要地分配新的内存块。• 合并常量。• 将 I/O 移入缓ff冲区。• 尽量避免做除法。• 尽量避免进行代价高昂的类型转换。• 移动指针而不是重新计算索引。

2.7 How to Deal with I/O Expense

对于很多问题而言，与与硬件设备通信的成本相比，处理器是很快的。这种成本通常简称为 I/O，它可以包括网络成本、磁盘 I/O、数据库查询、file I/O，以及其他那些往往会让一些并不靠近处理器的硬件去执行操作的开销。因此，构建一个快速的系统，往往更多是改善 I/O 的问题，而不是改进某个紧密循环中的代码，甚至不是改进某个算法。

改进 I/O 有两种非常基础的技术：缓存和表示。缓存是通过在本地存储该值的一个副本来避免 I/O（通常是避免读取某个抽象值），从而在获取该值时不需要执行 I/O。它会带来缓存一致性问题，但可能非常 effective，而且已有大量相关论述。表示则是通过更 efficiently 地表示数据来降低 I/O 成本的方法。这往往与其他需求形成张力，例如人类可读性和可移植性。

表示形式通常可以在其first实现的基础上提升两到三倍。实现这一点的技术包括：使用二进制表示而非人类可读的表示、随数据一起传输符号字典以避免对长符号进行编码，以及在极端情况下采用诸如 Huffman 编码之类的方法。我认为这通常是一种低垂的果实。

第三种有时可行的技术是通过将计算推近数据来提高引用局部性。例如，如果你从数据库读取一些数据并对其进行诸如求和之类的简单计算，尽量让数据库服务器替你完成。这在很大程度上取决于你所使用的系统类型，但总体而言，更频繁地探索构建在数据附近产生计算结果的服务器，从而只需传输少量数据，是值得的。

2.8 How to Manage Memory

内存是一种宝贵的资源，你无法 afford 让它耗尽。你通常可以暂时忽略它，但最终你将不得不决定如何管理内存。

需要超出单个子程序范围的空间通常称为 *heap allocated*。根据你使用的系统，你可能需要在该空间即将变成垃圾时显式地释放它，或者你可以依赖垃圾回收器。一个内存块在没有任何引用指向它时就是垃圾。垃圾回收器会识别垃圾并释放其空间，而程序员无需任何操作。垃圾回收是非常棒的。它减少了错误，并且便宜地增加了代码的简洁性和简短性。尽量使用它。但即使有垃圾回收，你也可能会 fi用完所有内存并堆满垃圾。一个经典的错误是使用哈希表作为缓存，却忘记从哈希表中移除引用。由于引用仍然存在，引用的对象无法被回收，但它已经没用了。这被称为内存泄漏。你应该尽早查找并 fi 修复内存泄漏。如果你有长时间运行的系统，在测试中内存可能永远不会耗尽，但最终用户会耗尽内存。

在任何系统中，创建新对象的成本适中。直接在子例程的局部变量中分配的内存不会产生额外开销，因为其释放策略可以非常简单。你应该避免不必要的对象创建和在局部子例程参数之外不必要的内存分配。

一个重要的情况是，当你能够定义一个上限，限制你在某一时刻所需对象的数量时。如果这些对象占用相同的内存空间，你可能能够分配一个单独的内存块，或者一个缓冲区，来存放它们。你所需的对象可以在这个缓冲区内按照固定的轮换模式分配和释放，因此它有时被称为环形缓冲区。这通常比堆分配更快。

有时候，您必须显式地释放分配的空间，以便它可以被重新分配，而不是依赖垃圾回收。然后，您必须对每一块分配的内存应用细致的智能，并设计一种方法，在适当的时间将其释放。此方法可能会因您创建的每种对象而有所不同。您必须确保每次执行内存分配操作时，最终都会有相应的内存释放操作。这是如此困难，以至于人们通常只是实现一种基本的垃圾回收形式，如引用计数，来为他们完成这项工作。

2.9 How to Deal with Intermittent Bugs

间歇性缺陷是那种“来自外太空的五十英尺隐形蝎子”式缺陷的近亲。这种噩梦出现得如此罕见，以至于难以观察，却又频繁到无法忽视。你无法调试，因为你无法find到它。

间歇性错误必须遵循与其他事物相同的逻辑法则。问题在于它仅在未知条件下发生。尝试记录发生时的情况，以便你可以猜测变动的真正原因。这个条件可能与数据值相关，例如“只有在输入 *Wyoming*作为值时才会发生。”如果这不是变动的根源，接下来需要怀疑的是同步不当的并发问题。

试着，试着，再试着去复现它。如果你无法复现它，就为它设下一个陷阱。

通过构建一个日志系统——必要时是一个特殊的系统——能够在问题真正发生时记录你认为需要的信息。如果这个缺陷只在生产环境中出现，这将是一个漫长的过程。你从日志中得到的每一条关于问题可能原因的线索，未必能直接给出解决方案，但会提示需要添加更多日志。新的日志往往需要很长时间才能部署到生产环境。然后你还得等待缺陷再次发生，才能获得更多信息。这个循环可能会持续相当一段时间。

2.10 How to Learn Design Skills

要学习如何设计软件，在导师进行设计时亲临现场，观察并学习他们的工作方式。然后研究那些编写良好的软件作品。之后，你可以阅读一些关于最新设计技术的书籍。

那么你必须亲自去做。从一个小项目开始。当你finally完成时，思考设计是如何失败或成功的，以及你是如何偏离最初构想的。然后转向更大的项目，最好与他人合作。设计是一种判断力，需要多年才能获得。一个聪明的程序员可以在两个月内充分掌握基础，并在此基础上不断提高。

形成你自己的风格是自然而且有益的，但要记住，设计是一门艺术，而不是一门科学。在这个主题上写书的人有既得利益，倾向于把它说得看起来 scientific。不要对特定的设计风格变得教条。

3 Team Skills

3.1 Why estimation is important

要尽快获得一个投入实际使用的可运行软件系统，需要规划开发，同时也要规划文档、部署、市场营销、销售以及 finance。没有对开发时间的可预测性，就无法 effectively 规划这些事项。

良好的估算能够带来可预测性。管理者喜欢这一点，而且他们确实应该如此。事实上，无论在理论上还是在实践中，准确预测软件开发需要多长时间都是不可能的，但这一事实常常被管理者忽视。我们一直被要求去做这件不可能的事，而我们必须诚实地面对它。然而，如果不承认这种不可能性，并在必要时加以解释，那就是不诚实的。关于估算存在着大量的误解空间，因为人们往往有一种惊人的倾向，会一厢情愿地认为下面这句话：

> 我估计，如果我真的理解了那个问题，而且在那段时间里没有人打扰我们，那么在5周内完成的可能性大约是50%。

真正的意思是：

I promise to have it all done 5 weeks from now.

因此，要像对待一个什么都不懂的人一样，明确地与接收估算的人讨论这个估算意味着什么。重申你的假设，不管在你看来它们多么显而易见。

3.2 How to Estimate Programming Time

估算需要练习。它也需要投入劳力。它需要的劳力如此之多，以至于先估算一下完成这次估算需要花多长时间，可能是个好主意，尤其是当你被要求去估算一些你认为很愚蠢的事情时。

当被要求对一件庞大的事情给出估计时，最诚实的做法是暂缓回应。大多数工程师都很热情、渴望取悦他人，而拖延肯定会让被拖着等待的一方感到不快。但当场给出的估计很可能既不准确也不诚实。

在拖延的同时，或许可以考虑实际去做或对该任务进行原型设计。如果政治压力允许，这是生成估计的最准确方式，并且能够取得真正的进展。

在无法做到这一点时，你应该首先非常清楚地界定该估算的含义。将这一含义作为你书面估算的 first 和最后一部分重新表述。通过将任务分解为逐步更小的子任务来准备书面估算，直到每个小任务不超过一天，理想情况下最多半天。最重要的是不要遗漏任何内容。例如，文档、测试、规划时间、与其他团队沟通的时间以及休假时间都非常重要。如果你每天有一部分时间要应付一些糟心的人或事，就在估算中为此单独列一项。这至少能让你的老板清楚地看到你的时间都花在了什么地方，也可能为你争取到更多时间。

我认识一些优秀的工程师会在估算中隐性地加缓冲，但我建议你不要这样做。这样做的一个结果是会在一定程度上降低信任。例如，工程师可能会把一个自己认为只需要一天的任务估算为三天。该工程师可能计划用两天来写文档，或者用两天时间去做其他有用的项目。但如果结果确实如此，就可以被察觉到这个任务实际上只用了一天完成，从而给人留下偷懒或高估工期的印象。更好的做法是如实、清晰地展示自己实际在做什么。如果文档编写花费的时间是编码的两倍，并且估算中如实反映了这一点，那么把这一情况对经理透明化将带来巨大的优势。

应当明确地加入缓冲。如果一项任务通常可能只需一天，但如果你的方法行不通则可能需要十天，那么在估算中尽量以某种方式注明这一点；如果做不到，至少按你对各种概率的估计做一个加权平均。任何你能够识别并给出估计的风险因素，都应该纳入进度计划。单个人在任意一周生病的概率并不高，但一个拥有许多工程师的大型项目必然会出现一定的病假时间；休假时间亦然。再比如，强制性的全公司培训研讨会发生的概率是多少？如果可以估计，就把它也算进去。当然，还存在未知的未知因素，

或未知未知值。根据定义，未知未知值无法单独估算。你可以尝试为所有未知未知值创建一个全局条目，或者以其他方式处理它们，并将处理方式告知你的老板。然而，你不能让老板忘记它们的存在，而且如果没有考虑未知未知值，估算很容易变成一个进度计划。

在团队环境中，你应该尽量让执行工作的人来做估算，并且应该尽量达成全团队对估算的共识。人们在技能、经验、准备度和信心方面差异很大。灾难发生在一个强大的程序员为自己做估算，然后让较弱的程序员按照这个估算来执行。如果让整个团队在逐行的基础上达成一致，这个估算不仅能够澄清团队的理解，还能提供重新分配资源的机会（例如，将负担从较弱的团队成员转移到较强的成员）。

如果存在无法评估的重大风险，你有责任以足够强硬的方式指出这一点，确保你的经理不会贸然作出承诺，从而在风险发生时陷入尴尬。希望在这种情况下，会采取一切必要措施来降低风险。

如果你能说服公司采用极限编程，你只需要估算相对较小的事情，这既更有趣，也更高效。

3.3 How to Find Information

你需要了解的内容的性质决定了你应该如何find它。

如果你需要关于具体事物的信息，这些信息是客观的且容易验证的，例如一个软件产品的最新补丁级别，可以通过在互联网上搜索或在讨论组中发帖，礼貌地询问大量人群。任何带有意见或主观解释的内容不应在互联网上搜索，因为真理与胡言乱语的比例太低。

如果你需要了解某些主观性的常识，了解人们对它的历史看法，可以去图书馆（存放书籍的实际建筑）。例如，要了解数学、蘑菇或神秘主义，就去图书馆。

如果你需要了解如何做一件并不简单的事情，那就找两三本关于该主题的书来读。你也许可以从互联网学会一些琐碎的事情，比如安装一个软件包。你甚至可以学到诸如良好编程技巧之类的重要内容，但你很容易在搜索和筛选结果、并试图判断这些结果的权威性上花费的时间，比阅读一本扎实书籍中相关部分所需的时间还要多。

如果你需要的是别人无法被期望知道的信息，例如，这个全新的软件是否能在巨型数据集上运行，你仍然必须搜索互联网和图书馆。在这些途径都被完全用尽之后，你可以设计一个实验来查明这一点，比如亲自试用。

如果你想要考虑某些独特情况的意见或价值判断，请咨询专家。　例如，如果你想知道用 LISP 构建一个现代数据库管理系统是否是个好主意，你应该咨询一位 LISP 专家和一位数据库专家。

如果你想知道某个特定应用是否很可能存在一种尚未发表的更快算法，去和在那个field工作的某个人谈谈。

如果你想做出只有你自己才能做出的个人决定，例如是否应该创业，那么可以考虑富兰克林方法或占卜。假设你已经从各个角度研究过这个想法，做完了所有功课，并在心中推演了所有后果以及利弊，但仍然举棋不定。现有的众多占卜技术对于判定你自身的半意识欲望非常有用，因为它们各自呈现出一个完整而模糊、随机的模式，你的潜意识会为其赋予意义。

3.4 How to Utilize People as Information Sources

尊重每个人的时间，并与自己的时间保持平衡。向某人提问不仅仅是为了得到答案，远比此更有意义。通过享受你的存在和听到你提出的具体问题，那个人了解了你。你也通过相同的方式了解那个人，同时可能得到你想要的答案。这通常比你的问题更为重要。

然而，随着你做得越多，这个价值会减少。毕竟，你在使用一个人最宝贵的资源——时间。沟通的好处必须与成本相权衡。此外，特定的成本和好处因人而异。我坚信，一个拥有一百人的高管应该每月花五分钟与她组织中的每个人交谈，这大约是她时间的五个百分点。但十分钟可能太长，如果他们有一千名员工，五分钟也可能太多。你与组织中每个人交谈的时间取决于他们的角色（比职位更重要）。你应该比你的老板与老板的老板交谈得更多，但你也应该和老板的老板聊一点。虽然可能不太舒服，但我相信你有责任每个月与所有上级至少聊一点，不管怎样。

基本规则是，每个人从与你稍微交谈中都会 benefits，而他们与你交谈得越多，所获得的 benefit 就越少。你的职责是向他们提供这种 benefit，同时获得与他们沟通的 benefit，并在所花时间与 benefit 之间保持平衡。

尊重你自己的时间很重要。如果与某人交谈，即使会占用他们的时间，却能为你节省大量时间，那么你就应该这样做，除非你认为按那个倍数来看，他们的时间对部落的价值比你的更高。

一个奇怪的例子就是暑期实习生。在一个高度技术化的岗位上，暑期实习生不能指望完成太多工作；他们可以指望让在场的每个人都烦透了。那么，为什么这会被容忍呢?

因为被打扰的人正在从实习生那里获得一些重要的东西。他们有机会稍微show off一下。他们也许有机会听到一些新的想法；他们有机会从一个different的视角看待事物。他们也可能在尝试招募这名实习生，但即便不成，他们也会获得大量的benefit。

你应该在你真诚地认为他们有话可说的时候，向人们请教一点他们的智慧和判断。这flatters他们，你也会学到一些东西，并教会他们一些东西。一名优秀的程序员并不常常需要销售副总裁的建议，但如果她真的需要，她一定要去征求。

3.5 How to Document Wisely

生活太短暂，不值得写没人会读的废话。如果你写废话，没人会读。所以，写一点好的文档最好。坏文档非常糟糕。经理们通常不理解这一点，因为即使是坏文档，也会给他们一种虚假的安全感，让他们觉得自己不依赖程序员。如果有人坚持让你写完全没用的文档，答应他们，然后默默开始寻找更好的工作。

没有什么比将准确的文档制作时间估算纳入估算中，更能有效地减轻文档需求的压力。事实是冷酷而无情的：文档编写，像测试一样，可能比开发代码花费更多的时间。

写好文档，first of all，是写好文字。我建议你find一些关于写作的书，学习它们，并加以练习。但即使你文笔拙劣，或对必须用来写文档的语言掌握不佳，你真正需要的也只有一条黄金法则：“己所不欲，勿施于人。”花时间认真思考谁会阅读你的文档，他们需要从中获得什么，以及你如何把这些教给他们。做到这一点，你就会成为一名高于平均水平的文档作者，也是一个优秀的程序员。

当涉及到实际记录代码本身时，与生成可以被非程序员阅读的文档不同，我所认识的最优秀的程序员都有一个共同的观点：编写自解释的代码，除了在无法使其清晰的地方，不要记录代码。这样做有两个充分的理由。首先，任何需要查看代码级文档的人在大多数情况下都能够并且更愿意直接阅读代码。诚然，这对经验丰富的程序员来说比对初学者要容易得多。然而，更重要的是，如果没有文档，代码和文档就无法失去同步。源代码最坏的情况是错误且令人困惑。而文档如果没有写得完美，可能会撒谎，这比错误更糟糕千倍。

这并没有让负责任的程序员更轻松。如何编写自解释的代码？那到底是什么意思？这意味着：

- 编写代码时知道有人必须阅读它；

• 遵循黄金法则；• 运用你可能学到的任何良好写作规则；• 选择一种直截了当的解决方案，即使你可以用另一种方案更快地应付；• 放弃那些会让代码变得晦涩的微小优化；• 为读者着想，花费一些你宝贵的时间，让她更轻松。

3.6 How to Work with Poor Code

与他人编写的低质量代码打交道是非常常见的。不过，在你真正设身处地之前，不要过分苛责他们。他们很可能是在明确的要求下，为了应对进度压力而被要求尽快把事情做完。但要处理不清晰的代码，你必须先理解它。理解它需要学习时间，而这些时间必然要从某个地方、某个计划中挤出来，你必须坚持争取。要理解它，你必须阅读源代码。你很可能还需要对它进行实验。

现在是记录的好时机，即使这只是为了自己，因为试图记录代码的行为会迫使你考虑一些可能没有考虑过的角度，最终的文档可能会很有用。在做这件事时，考虑一下重写部分或全部代码需要什么。重写其中的一部分真的能节省时间吗？如果重写了它，你能更信任它吗？这里要小心傲慢。如果你重写了代码，自己处理起来可能会更轻松，但对下一个需要阅读代码的人来说，真的会更容易吗？如果你重写了它，测试负担会怎样？重新测试的需要是否会超过可能获得的任何fi益处？

在你对并非由你编写的代码所进行的任何工作估算中，该代码的质量都应当affect你对问题以及未知的未知风险的认知。

重要的是要记住，抽象和封装——程序员最好的两种工具——尤其适用于糟糕的代码。你也许无法重新设计一大块代码，但如果你能为它加入一定程度的抽象，就可以在不重做整个烂摊子的情况下，获得良好设计的一些好fi处。尤其是，你可以尝试将那些特别糟糕的部分隔ff离开来，以便它们可以被独立地重新设计。

3.7 How to Use Source Code Control

源代码控制系统可以让你有效地ff管理项目。它们对个人非常有用，而对团队来说则是必不可少的。它们会跟踪不同ff版本中的所有更改，这样代码永远不会丢失，并且可以为这些更改赋予意义。我曾经很晚才认识到它们的好处fi，但现在即使是在单人项目中，我也离不开它。

使用源代码控制系统的一种好方法是始终保持与最新状态相差不超过几天。无法在几天内finished 的代码也会被提交，但会以一种处于非活动状态、不会被调用的方式提交，因此在完成单元测试之前不会给任何其他人造成问题。

3.8 How to Unit Test

单元测试，即由编写代码的团队对单个编码功能进行的测试，是编码的一部分，而不是与之 different 的东西。设计代码的一部分就是设计如何对其进行测试。你应该写下一个测试计划，即使只有一句话。有时测试会很简单：\按钮看起来好吗?" 有时它会很复杂：\这个匹配算法是否精确地返回了正确的匹配?"

尽可能使用断言检查和测试驱动程序。这不仅能及早发现 bug，而且在后期也非常有用，还能让你消除那些原本需要费心担忧的疑惑。

极限编程的开发者们在单元测试 effectively 方面有大量著述；我所能做的不过是推荐他们的作品。

3.9 How to Stress Test

与单元测试不同，压力测试是有趣的。在 first 看来，压力测试的目的似乎是要 find 出系统在负载下是否能够工作。实际上，常见的情况是系统在有负载时确实能工作，但当负载足够大时会以某种方式失效。我把这种情况称为撞墙，或者称为 *bonking*[1]。也许会有一些例外，但几乎总会存在一堵墙。压力测试的目的在于 figure 出这堵墙在哪里，然后再 figure 出如何把这堵墙向外推得更远。

应在项目早期制定压力测试计划，因为这通常有助于明确到底期望什么。对于一次网页请求来说，两秒是惨败还是巨大的成功？five hundred 并发用户够吗？这当然取决于情况，但在设计处理该请求的系统时，必须知道答案。压力测试需要对现实进行足够真实的建模，才有用。要非常容易地同时模拟 five hundred 名人类用户使用一个系统其实并不现实，但至少可以创建 five hundred 个模拟，并尝试建模他们可能做的某些行为。

在压力测试中，先从较轻的负载开始，然后沿着某个维度（例如输入速率或输入规模）逐步给系统加负载，直到你撞墙为止。如果这道墙离你的需求太近，figure 出哪个资源是瓶颈（通常会有一个占主导的瓶颈）。是内存、处理器、I/O、网络带宽，还是数据争用？然后 figure 出你如何移动这道墙。请注意，移动这道墙，也就是提高系统能够承受的最大负载，可能并没有帮助，甚至可能会损害轻载系统的性能。通常，

[1] This term has several meanings, derived from “to hit”, but is in particular used by athletes to describe running out of blood sugar or some other basic resource that manifests as a sudden rather than gradual degradation of performance or spirt.

在高负载下的性能比在轻负载下的性能更重要。

你可能需要从多个 different 维度获得可见性，才能建立起对它的心智模型；没有任何单一技术是 sufficient 的。例如，日志通常能很好地反映系统中两个事件之间的墙上时钟时间，但如果构造不够谨慎，就无法提供对内存利用率，甚至数据结构大小的可见性。类似地，在现代系统中，许多计算机和大量软件系统可能在协同工作。尤其当你遇到“bonking”（也就是说，性能随输入规模呈非线性变化）时，这些其他软件系统可能成为瓶颈。即使只是测量所有参与机器上的处理器负载，对这些系统的可见性也会非常有帮助。

了解墙的位置不仅对于移动墙至关重要，而且还能提供可预测性，从而使业务能够以 effectively 的方式进行管理。

3.10 How To Recognize When To Break or Go Home

计算机编程是一种活动，但它也是一种文化。不幸的是，这并不是一种非常重视心理或身体健康的文化。由于文化/历史原因（例如需要在夜间使用空载的计算机工作），以及巨大的上市时间压力和程序员的稀缺，计算机程序员经常过度工作。我不认为你能相信听到的所有说法，但我认为每周六十小时很常见，而fifty几乎是最低限度。这意味着通常需要远远超过这个时间。这对优秀的程序员来说是个严重的问题，因为他们不仅要对自己负责，也要对他们的队友负责。你必须识别什么时候该回家，有时也要建议其他人回家。对于解决这个问题，不可能有任何fixed规则，正如抚养孩子也不可能有fixed规则一样，原因相同———every human being is different。

每周超过六十个小时对我来说是非同寻常的 effort，我只能在短时间内（大约一周）做到，而有时也会有人期待我这样做。我不知道要求一个人工作六十个小时是否公平；我甚至不知道四十个小时是否公平。然而我确信，工作到从多出来的那一小时里几乎得不到任何收益是愚蠢的。对我个人而言，每周超过六十个小时就是如此。我个人认为，程序员应当秉持“贵族义务”的精神，承担沉重的负担。然而，当冤大头并不是程序员的职责。可悲的事实是，程序员常常被要求充当冤大头，以便为某些人做一场表演，例如一位试图给高管留下深刻印象的经理。程序员往往会屈从于此，因为他们急于取悦他人，而且不太擅长说“不”。对此有四种防御措施：

- 尽可能与公司内的每个人沟通，以免有人就正在发生的情况误导高管，

- 学会以保守且明确的方式进行估算和排期，并让每个人都能清楚地看到计划是什么以及目前处于什么状态，
- 学会说不，并在必要时作为一个团队一起说不，
- 如果必须的话就退出。

大多数程序员都是优秀的程序员，而优秀的程序员希望完成大量工作。为此，他们必须 effectively 管理自己的时间。在对一个问题逐渐热身并深入投入时，会存在一定程度的心理惯性。许多程序员 find 他们在拥有长时间、不被打断的时间块来热身并专注时工作效果最好。然而，人们必须睡眠并履行其他职责。每个人都需要 find 一种方式，同时满足其生理节律和工作节律。每位程序员都需要采取一切必要措施来获得 efficient 的工作时段，例如预留某些日子，只参加最关键的会议。

自从有了孩子，我会尽量有时把晚上留给他们。对我最有效的节奏是：工作很长的一天，在 office 里或 office 附近睡觉（我从家到工作的通勤很长），然后第二天足够早回家，在孩子上床睡觉前陪他们。我对此并不舒服，但这是我能想出的最佳折中方案。如果你有传染病，就回家。如果你有自杀念头，你应该回家。如果你产生了超过几秒钟的杀人念头，你应该休息一下或回家。如果某人表现出严重的精神功能失常，或出现超出轻度抑郁的精神疾病迹象，你应该让他们回家。如果由于疲劳，你受到诱惑而以你平时不会的方式变得不诚实或具有欺骗性，你应该休息一下。不要用可卡因或安非他明来对抗疲劳。不要滥用 caffeine。

3.11 How to Deal with Difficult People

你可能不得不与难ffi相处的人打交道。你甚至可能就是其中之一。如果你是那种经常与同事和权威fi人物发生冲ffl突的人，你应该珍惜这所意味着的独立性，但要在不牺fi牲你的智慧或原则的前提下，提升你的人际交往能力。

这对大多数没有此类经验的程序员来说可能非常令人不安，而他们以往的人生经历教会他们的行为模式在职场中并不适用。Difficult persons 往往已经习惯于分歧，而且与他人相比，他们更少 affected 于促使妥协的社会压力。关键在于以恰当的分寸去尊重他们——比你愿意给的要多，但又不必达到他们可能想要的程度。

程序员必须作为一个团队一起工作。当出现分歧时，必须以某种方式加以解决，不能长期回避。Difficult 的人往往极其聪明，并且有非常有价值的观点可说。关键在于应当在不带有由……造成的偏见的情况下，倾听并理解 difficult 的人

人。沟通失败常常是分歧的根源，有时通过极大的耐心可以消除这种问题。尽量保持沟通冷静和友好，不要接受可能被提供的任何挑衅，以引发更大的冲突。在经过合理的理解尝试后，做出决定。

不要让恶霸强迫你做你不同意的事情。如果你是领导者，做你认为最好的决定。不要出于个人原因做决定，并且准备好解释你做决定的理由。如果你是与一个难相处的人做队友，不要让领导的决定对你产生个人影响。如果事情没有按你的方式进行，就全心全意地以另一种方式去做。

困难的人确实会改变和进步。我亲眼见过，但这非常罕见。然而，每个人都有短暂的起伏。

每个程序员，尤其是领导者面临的挑战之一，就是让难的人员保持完全投入。他们比其他人更容易回避工作并且消极抗拒。

Part II
Intermediate

4 Personal Skills

4.1 How to Stay Motivated

这是一个令人惊讶且美妙的事实：程序员往往受到创造美丽、有用或巧妙的作品的欲望驱动。这种欲望并非程序员独有，也不是普遍存在的，但它在程序员中如此强烈且普遍，以至于在大多数公司中，这使得他们与其他人区别开来。

这具有实际和重要的后果。如果程序员被要求做一些既不美观、也不实用、甚至不精巧的事情，他们的士气会很低。做丑陋、愚蠢、无聊的工作确实可以赚很多钱；但最终，乐趣将为公司带来最多的收益。

4.2 How to Tradeoff Time versus Space

你可以不上大学也成为一名优秀的程序员，但如果不了解基础的计算复杂性理论，就不可能成为一名合格的中级程序员。没有它，你或许还能凭直觉理解如何在时间与空间之间进行 tradeoff，但没有它，你将无法拥有与同事沟通所需的 firm 基础。

时间（处理器周期）和空间（内存）可以互相交换。工程学就是关于折中的，这就是一个的例子。它并不总是系统化的。然而，一般来说，通过更紧凑地编码某些东西可以节省空间，但这需要在解码时消耗更多的计算时间。通过缓存，可以节省时间，即花费空间存储某个事物的本地副本，但这需要付出维护一致性的代价。

缓存。有时通过在数据结构中维护更多信息可以节省时间。通常这会占用少量空间，但可能会大大复杂化算法。

改进空间/时间权衡ff往往会使其中一方或另一方发生戏剧性的变化。然而，在着手之前，你应该问问自己，你正在改进的是否真的是系统中最需要改进的部分。研究算法很有趣，但你不能因此忽视一个冷酷的事实：改进一个并非问题的东西不会带来任何明显的差ff别，而且还会增加测试负担。

现代计算机上的内存看起来很便宜，因为不像处理器时间那样，你在撞墙之前看不到它被使用。但一旦你撞上，就会撞得很惨。使用内存还有其他隐性成本，比如你对必须常驻的其他程序的 effect，以及分配和释放它所需的时间。在你为了获得速度而牺牲空间之前，请仔细考虑这一点。

4.3 How to Balance Brevity and Abstraction

抽象是编程的关键。应当谨慎地选择所需要的抽象程度。初学者在热情之下，常常创建比实际有用得多的抽象。一个迹象是：你创建了一些几乎不包含任何代码、也几乎不做任何事情、只是用来抽象某些东西的类。其吸引力是可以理解的，但代码简洁性的价值必须与抽象性的价值相权衡。偶尔可以看到热情的理想主义者犯下的一种温和错误：在项目开始时，定义了大量类，这些类在抽象层面上极其优美，似乎能够覆盖所有可能情况。随着项目推进、疲劳感出现，代码本身却变得凌乱。函数体变得比应有的更长。那些空类成了需要文档说明的负担，在压力之下被忽视。如果把花在抽象上的精力用来保持事物简短而简单，最终结果会更好。我强烈推荐 Paul Graham 的文章《\简洁即力量"》[6]。

围绕诸如信息隐藏和面向对象编程等有用技术，存在着某种教条主义，有时会被推得过头。这些技术使人能够以抽象方式编码并预期变化。然而，我个人认为，不应该产生太多臆测性的代码。例如，一种被广泛接受的风格是，将对象上的一个整型变量隐藏在修改器和访问器之后，这样变量本身不会被暴露，只有与之交互的小接口。这确实允许在不affecting调用代码的情况下改变该变量的实现，对于必须发布非常稳定 API 的库作者来说，这或许是合适的。但我并不认为，当我的团队拥有调用代码、因此既可以像重写被调用方一样轻松地重写调用方时，这种做法的benefit足以抵消其冗长性的成本。多出四到five行代码，是为这种臆测性的benefit付出的沉重代价。

可移植性也带来类似的问题。代码是否应该能在 different 计算机、编译器、软件系统或其他环境之间保持可移植，还是只要容易移植即可？我认为，一段不可移植但短小且易于移植的代码，比一段冗长的

可移植的那一种。将不可移植的代码限fi制在指定区域相对容易，而且无疑是个好主意，例如用于执行针对特fi定 DBMS 的数据库查询的类。

4.4 How to Learn New Skills

学习新技能，尤其是非技术性的技能，是最大的乐趣。大多数公司如果明白这对程序员的激励有多大，士气都会更好。

人类在实践中学习。阅读书籍和上课是有用的。但是你会尊敬一个从未写过程序的程序员吗？要学习任何技能，你必须把自己置于一个宽容的环境中，在那里你可以练习那项技能。在学习一门新的编程语言时，在不得不做大型项目之前，试着先做一个小项目。在学习管理软件项目时，先试着管理一个小的first。

好的导师不能替代你亲自动手，但比一本书要好得多。你能 offer 什么给一位潜在的导师，以交换他们的知识？至少，你应该 offer 努力学习，这样他们的时间就不会被浪费。

尽量争取让老板批准你接受正式培训，但要明白，这通常并不比把同样的时间用来单纯地练习你想学的新技能更好。然而，在我们这个并不完美的世界里，申请培训要比申请“玩耍时间”容易得多，尽管很多正式培训不过是在课堂上睡觉、等着晚宴而已。

如果你在带领他人，就要理解他们如何学习，并通过给他们分配规模合适、能够锻炼他们想学习的技能的项目来帮助他们学习。别忘了，对程序员来说，最重要的技能并非技术技能。给你的团队一个机会去尝试和练习勇气、诚实与沟通。

4.5 How to do Integration Testing

集成测试是对已经完成单元测试的各个组件进行集成的测试。集成的成本很高，而这种成本会在测试中体现出来。你必须在你的估算和进度安排中为此预留时间。

理想情况下，你应该组织项目，使其不在末尾出现一个必须进行集成的阶段。更好的做法是在项目过程中，随着各项工作完成而逐步进行集成。若不可避免，应当仔细进行估算。

5 Team Skills

5.1 How to Manage Development Time

为管理开发时间，保持一份简洁且最新的项目计划。项目计划是一种估算、一份进度表，以及一组用于标记进展的里程碑，

为估算中的每项任务分配你团队的时间或你自己的时间。它还应包括你必须记住要做的其他事情，例如与质量保证人员会面、准备文档或订购设备。如果你在一个团队中，项目计划应当是一份经过一致同意的协议，无论是在开始时还是在推进过程中。

项目计划的存在是为了帮助做出决策，而不是展示你的组织能力。如果项目计划过长或未及时更新，它将无法在决策中发挥作用。实际上，这些决策与具体的个人有关。计划和你的判断力能帮助你决定是否应将任务从一个人转移到另一个人。里程碑标志着你的进展。如果你使用花哨的项目规划工具，不要被诱惑为项目提前做出庞大的设计，而应利用它保持简洁和时效性。

如果你错过了一个里程碑，你应当立即采取行动，例如通知你的老板，该项目的计划完成时间已按同等幅度推迟。最初的估算和进度安排本来就不可能完美；这会造成一种错觉，让你以为可以在项目后期把错过的天数补回来。你也许可以。但同样有可能你低估了那一部分，正如你高估了它一样。因此，无论你是否愿意，项目的计划完成时间已经发生了延误。

确保你的计划中包括用于以下事项的时间：

- 内部团队会议，
- 演示，• 文档，• 定期安排的活动，• 集成测试，• 与外部人员打交道，• 疾病，• 休假，• 现有产品的维护，以及 • 开发环境的维护。

项目计划可以作为一种方式，让外部人员或你的老板了解你或你的团队正在做什么。因此，它应该简短并保持最新。

5.2 How to Manage Third Party Software Risks

一个项目往往依赖于由其无法控制的组织所开发的软件。第三方软件存在重大风险，必须被所有相关方充分认识。

永远、永远不要把任何希望寄托在 *vapor* 上。*Vapor* 指的是任何被承诺但尚未可用的所谓软件。这是走向倒闭的最稳妥途径。对软件公司承诺在某个日期发布具备某项功能的某个产品保持怀疑并不明智；更明智的是彻底忽略它，忘记你曾经听说过。绝不要把它写进你公司使用的任何文档中。

如果第三方软件不是虚拟的，它仍然存在风险，但至少这是一个可以解决的风险。如果你考虑使用第三方软件，应该尽早投入精力进行评估。人们可能不喜欢听到评估三个产品的适用性需要两周或两个月的时间。但这必须尽早完成。如果没有适当的评估，无法准确估算集成的成本。

理解现有第三方软件是否适合某一特定用途是一种高度依赖隐性知识的事情。这非常主观，通常掌握在专家手中。如果你能find到那些专家，就能节省大量时间。项目往往会如此彻底地依赖某个第三方软件系统，以至于一旦集成失败，项目就会失败。应在进度计划中以书面形式清楚地表达这类风险。尽量准备一个应急方案，例如可以使用的替代系统，或者如果风险无法及早消除，就具备自行实现该功能的能力。绝不要让进度计划建立在空中楼阁之上。

5.3 How to Manage Consultants

使用顾问，但不要完全依赖他们。他们是了不起的人，值得高度尊重。他们通常比公司员工程序员更了解具体的fi技术甚至编程技巧。最好的使用方式是将他们作为内部教育者，通过示范来教学。

然而，他们通常无法像正式员工那样在同样的意义上成为团队的一部分，哪怕只是因为你可能没有足够的时间去了解他们的优势和劣势。他们的financial承诺要低得多。他们更容易流动。如果公司经营得好，他们可能获得的回报更少。有些会很优秀，有些一般，也有些很糟，但通常你在选择顾问时不会像选择员工那样谨慎，因此你会遇到更多不合格的。

如果他们要编写代码，你必须在过程中仔细审查。你不能在项目结束时面临一大段未经审查的代码的风险。这对所有团队成员来说都是真实的，但你通常对与你更亲近的团队成员有更多的了解。

5.4 How to Communicate the Right Amount

仔细考虑一次会议的成本。它等于会议时长乘以参与人数。会议有时是必要的，但规模越小通常越好。小型会议的沟通质量更高，而且总体上浪费的时间更少。如果你正在开会而有人感到无聊，这应当是一个信号，表明也许你应该组织规模更小的会议。

应尽一切可能鼓励非正式沟通。与同事共进午餐时完成的有用工作，比任何其他时间都多。遗憾的是，更多公司并未认识到这一事实并加以支持。

5.5 How to Disagree Honestly and Get Away with It

分歧是做出良好决策的绝佳机会，但需要谨慎处理。希望在做出决定之前，你已经充分表达了自己的想法，并且得到了倾听。在这种情况下，就无需再多说什么了，你需要决定即使不同意该决定，是否仍然会支持它。如果你能够在不同意的情况下依然支持这个决定，就明确表达出来。这表明你的价值所在：你是独立的，而不是唯唯诺诺的人；同时你尊重决策，并且是一个具有团队精神的成员。

有时，在决策者未能充分受fi益于你的意见的情况下，会做出一个你不同意的决定。此时你应当基于对公司或部落的利fi益来评估是否需要提出这个问题。如果在你看来这是一个小错误，可能不值得重新考虑；如果在你看来这是一个重大错误，那么你当然必须提出论证。

通常这不是个问题。在一些高压情境下，结合某些人格类型，这可能会导致事情被当作针对个人。例如，一些非常优秀的程序员即使有充分理由认为某个决定是错误的，也缺乏挑战该决定所需的 confidence。在最糟糕的情况下，决策者本身缺乏安全感，并将其视为对她权威的个人挑战。在这种情况下，最好记住人们往往会用大脑中爬行动物的那一部分作出反应。你应该私下提出你的论点，并尝试说明新的知识如何改变了做出该决定所依据的基础。

无论该决定是否被推翻，你都必须记住，或许永远不会有人有资格说“我早就说过！”，因为另一种决定并未被充分探索。

6 Judgment

6.1 How to Tradeoff Quality Against Development Time

软件开发始终是在项目能做什么与把项目完成之间的权衡。但你可能会被要求为了加快项目的部署，在质量上做出 tradeoff，而这个项目会像制造一个被设计成会报废的烤面包机是 offensive 一样，offends 你的工程直觉。例如，你可能会被要求去做一些糟糕的软件工程实践，这将导致大量的维护问题。

如果发生这种情况，你的fi首要责任是通知你的团队，并清楚地解释质量下降所带来的成本。毕竟，你对这一点的理解应该远胜于你老板的理解。要明确说明失去了什么、得到了什么，以及在什么成本下、将在下一个周期中如何收复失去的阵地。在这一点上，一个良好的项目计划所提供的可见性应该会有所帮助。如果质量权衡ff影ff响了质量保证的投ff入，请指出这一点（既要告诉你的老板，也要告诉质量保证人员）。如果质量权衡ff将导致在质量保证周期结束后报告更多缺陷，也请指出这一点。

如果她仍然坚持，你应该尝试将这些低劣之处隔离到具体的组件中，以便你可以在下一个周期中计划重写或改进它们。向你的团队解释这一点，以便他们能够据此进行规划。

6.2 How to Manage Software System Dependence

现代软件系统往往依赖于大量并非由你直接控制的组件。这通过协同效应和复用提高了生产力。然而，每个组件也会带来一些问题：

- 你将如何在组件中fix 缺陷？ • 该组件是否会将你限制在特定的硬件或软件系统上？ • 如果组件完全失效，你将怎么办？

以某种方式对组件进行封装总是最好的，这样它就被隔离开来，并且可以被替换。如果该组件证明完全不可用，你也许能够得到一个不ff同的，但你也可能不得不自己编写一个。封装并不等同于可移植性，但它能让移植更容易，这几乎同样好。

拥有某个组件的源代码可以将风险降低到原来的四分之一。有了它，你可以更容易地评估、更容易地调试、find 更容易地制定变通方案，并且更容易地进行 fixes。如果你做了 fixes，你应该把它们交给该组件的所有者，并尽量让他们将其纳入一个 official 发布版本；否则你将不得不不太舒服地维护一个 unofficial 版本。

6.3 How to Decide If Software Is Too Immature

使用他人编写的软件是快速构建稳固系统的最 effective 方式之一。这不应被劝阻，但必须审视与之相关的风险。最大的风险之一是在软件通过使用而成熟为可用产品之前，常常伴随的一段漏洞频出、几乎无法运行的时期。在考虑与某个软件系统进行集成或以某种方式对其产生依赖时，无论是内部创建还是由第三方提供，都非常重要的一点是要考虑它是否真的足够成熟可以使用。以下是你应该就此问自己的几个问题:

- 它是蒸汽ware吗？（承诺非常不成熟。）
- 是否有关于该软件的、可获取的知识体系？
- 你是第 first 个用户吗？
- 是否有继续下去的强烈激励？
- 是否有过维护方面的 effort？
- 如果当前维护者流失，它还能存活吗？
- 是否存在一个至少有它一半水准的成熟替代方案？
- 你的社群或公司是否了解它？
- 它对你的社群或公司是否有吸引力？
- 即使它很糟，你也能雇到人来为它工作吗？

6.4 How to Make a Buy vs. Build Decision

一家试图通过软件来实现目标的创业型公司，必须不断地做出“购买还是自建”的决策。这需要将商业、管理和工程方面的敏锐洞察力高度结合。或许更准确地说，这应被称为“购买并集成”与“构建并集成”的决策，因为必须考虑集成成本。

- 你的需求与它的设计用途有多匹配？
- 你购买的东西中有多少是你真正需要的?

- 评估集成的成本是多少?

- 集成的成本是多少？
- 购买会增加还是降低长期维护成本？
- 自行构建是否会让你陷入一个你并不想处于的业务处境?

在构建一个规模大到足以作为另一个完整业务基础的东西之前，你应该三思而后行。这样的想法往往由聪明而乐观的人提出，他们通常能为你的团队做出大量贡献。如果他们的想法确实引人注目，你或许会考虑调整你的商业计划；但在没有经过深思熟虑之前，不要投入一个规模超过你自身业务的解决方案。

在考虑了这些问题之后，你或许应该准备两个项目计划草案：一个用于自建，一个用于购买。这样会迫使你考虑集成成本。你还应当考虑两种方案的长期维护成本。要估算集成成本，你必须在购买前对软件进行彻底评估。如果你无法评估它，那么购买它就会承担不合理的风险，你应当决定不购买该产品。如果同时在考虑多个采购方案，则必须投入精力对每个方案进行评估。

6.5 How to Grow Professionally

承担超出你职权范围的责任。扮演你渴望的角色。对人们为更大组织的成功所作出的贡献，以及那些在个人层面帮助过你的事情，表达感激之情。

如果你想成为团队领导者，就推动共识的形成。如果你想成为管理者，就对日程安排负责。通常你可以在与领导者或管理者共事时从容地做到这一点，因为这会让她腾出精力去承担更大的责任。如果这对你来说尝试起来太过勉强，那就一点一点来。

评估你自己。如果你想成为一名更好的程序员，向你钦佩的人请教如何才能变得像她一样。你也可以询问你的老板，他可能知道得更少，但对你的职业生涯会产生更大的影响。

通过将其融入你的工作中，规划学习新技能的方法，无论是学习新软件系统这类琐碎的技术技能，还是像写作能力这类困难的社会性技能。

6.6 How to Evaluate Interviewees

对潜在员工的评估并未投入应有的精力。一次糟糕的雇用，就像一段糟糕的婚姻一样，是可怕的。每个人的精力中都应该有相当fi大的一部分用于招聘，但这种情况却很少发生。

有不ff同的面试风格。有些是折磨人的，旨在让候选人承受巨大的压力。这可能具有非常重要的价值，因为它可能在压力之下揭示性格fi缺陷和弱点。候选人对面试官的坦诚程度并不比对自己更高，而人类自我欺骗的能力令人惊讶。

你至少应该给候选人进行相当于两小时的技术技能口头考试。通过练习，你将能够快速覆盖他们所知道的内容，并在他们不知道的地方迅速收回，以划定边界。面试者会尊重这一点。我曾多次亲耳听到他们说，那是他们的动机之一，为了

选择一家公司。优秀的人希望因他们的技能而被雇用，而不是因为他们上一次在哪里工作、毕业于哪所学校，或其他一些无关紧要的特征。

在这样做的同时，你还应该评估他们的学习能力，这比他们已经知道什么要重要得多。你还应该留意由 difficult 的人 given off 的“whiff of brimstone”。事后你常常能识别出来，但在面试的紧张当下却很难察觉。人们沟通与协作的能力，比是否掌握最新的编程语言更重要。

最后，面试也是一个推销的过程。你应该试图将你的公司推销给候选人。然而，你面对的是程序员，所以常规的销售技巧不起作用。不要试图美化事实。先从糟糕的部分 off 开始，然后用好的部分 finish strong stuff 结束。

6.7 How to Know When to Apply Fancy Computer Science

关于算法、数据结构、数学以及其他让人惊叹的stuff，有一整套知识体系，大多数程序员都了解，但很少真正使用。在实践中，这些精彩的stuff过于复杂，而且通常并非必要。比如，当你大部分时间都花在进行inefficient 数据库调用时，改进算法并没有意义。不幸的是，相当一部分编程工作只是让系统彼此通信，并使用非常简单的数据结构来构建一个漂亮的用户界面。

高技术在什么时候才是合适的技术？什么时候应该翻书，去获取一些不同于司空见惯算法的东西？有时这样做确实有用，但应当经过谨慎评估。

对于潜在的计算机科学技术，三个最重要的考虑因素是：

- 它是否被良好封装，从而对其他系统的风险较低，并且整体复杂性和维护成本的增加也较低？ • 该fi收益是否令人惊讶（例如，在成熟系统中达到两倍，或在新系统中达到十倍）？

- 你将能够测试并评估它effectively吗？

如果一个良好隔离、采用稍微花哨的算法，能够在整个系统范围内将硬件成本降低或将性能提升一倍，那么不去考虑它几乎是不可原谅的。为这种方法辩护的关键之一，是证明其风险实际上相当低，因为所提出的技术很可能已经被充分研究，唯一的问题只是集成风险。在这里，程序员的经验和判断可以与这种花哨的技术真正形成协同，使集成变得容易。

通过精心设计封装方案以降低风险，你应该能够做出一个具有信fi心的估计，从而使该提案的成本和收fi益得到恰当评判。

6.8 How to Talk to Non-Engineers

工程师和程序员特别被大众文化认定为与其他人不同。这意味着其他人也与我们不同。在与他们交流时，值得牢记这一点。应始终理解受众。

非工程师很聪明，但不像我们一样扎实地创造技术性事物。我们制造东西。他们销售东西、处理东西、计算东西和管理东西，但他们不是制造事物的专家。

他们在团队合作方面不如工程师（当然也有例外）。在非团队环境中，他们的社交技能通常与工程师相当，甚至更好，但他们的工作并不总是要求他们像我们一样进行那种紧密、精确的沟通和仔细的任务分配。他们的团队更像是小组。

非工程师可能过于急于取悦你，他们可能会被你吓到。就像我们一样，他们可能会为了取悦你或因为有些害怕你而答应，却并不真心这样做，之后也不会为自己的话负责。

非程序员可以理解技术内容，但他们没有技术判断力。他们理解技术是如何工作的，但无法理解为何某种方法需要三个月，而另一种方法只需三天。这为与他们协同合作提供了极大的机会。

当与团队交谈时，你会不自觉地使用一种简写语言，这种语言非常ff有效，因为你们在技术和你的产品方面有很多共同的经验。然而，这对外部人员来说并不奏效，尽管通过练习他们会逐渐适应。你必须与他们慢慢来。

与团队合作时，基本假设和目标无需频繁重述，大多数对话都集中在细节上。而与外部人员交流时，情况则恰恰相反。他们可能不理解你认为理所当然的事情。由于你认为这些是理所当然的并且没有重复，它可能导致你与外部人员的对话结束时，你以为自己理解对方，但实际上存在很大的误解。你应该假设自己会被误解，并仔细观察以fi发现他们的误解。尝试让他们总结或转述你所说的内容，以确保他们理解。

我喜欢与非工程师一起工作。这提供了很好的学习和教学机会。在沟通的清晰度方面，你可以通过以身作则来领导。工程师被训练来从混乱中带来秩序，从困惑中带来清晰，非工程师正是欣赏我们这一点。因为我们具有技术判断力，并且通常能理解商业问题，我们往往能 fi 找到一个简单的解决方案。非工程师常常提出他们认为能让我们更轻松的解决方案，出于善意和做正确事情的愿望，但实际上存在一个更好的解决方案，他们因为缺乏技术判断力而看不见这个更好的解决方案。

Part III
Advanced

7 Technological Judgment

7.1 How to Tell the Hard From the Impossible

我们的工作是完成艰难之事并辨识不可能之事。¿从大多数在职程序员的角度来看，所谓的研究是不可能的，因为它无法被预测、估算和排期。大量的*mere work*是困难的，但不一定是不可能的。

这种区分并非儿戏，因为你很可能会被要求去做在实践中几乎不可能完成的事情，无论是从scientific的角度还是从软件工程的角度。于是，这就成了你的工作，去帮助创业者find一个合理的解决方案，该方案是*merely hard*，并且能实现她大部分的诉求。当一个方案可以被confidently地排期并且风险已被理解时，它只是困难而已。

不可能满足一个模糊的需求，例如 \Build 一个系统，为任何人计算最具吸引力的发型和颜色。" 如果需求可以变得更加明确，往往就会变成只是困难一些，例如 \Build 一个系统，为某个人计算一种有吸引力的发型和颜色，让他们能够预览并进行修改，并且基于最初造型的客户满意度高到让我们赚很多钱。" 如果没有对成功的清晰 definition，你就不会成功。

7.2 How to Utilize Embedded Languages

将一种编程语言嵌入到系统中，对程序员而言具有一种情色般的吸引力。它使系统变得异常强大，使她得以施展自己最具创造性、最具普罗米修斯精神的技能，并让系统成为她的朋友。

我和许多其他程序员都曾陷入过创建特殊用途嵌入式语言的陷阱。我自己就陷进去过两次。问题在于，一个可编程的系统只有当你是程序员时才显得美妙。

它无疑 offer 着巨大的力量。世界上最好的文本编辑器都内置了语言。其使用程度取决于目标受众能否掌握该语言。当然，语言的使用可以像文本编辑器中那样设为可选，这样入门者可以使用，而其他人不必。

在嵌入一种语言之前，真正需要问自己的问题是：这是否与我的受众文化相契合，还是相冲突？如果你的目标受众完全是非程序员，这将如何提供帮助？如果你的目标受众完全是程序员，他们是否更倾向于应用程序员接口？那会是哪种语言？程序员并不想学习一个

仅仅为了好玩而去学习一种在其文化之外的新语言；但如果它与他们的文化相契合，他们就不必花太多时间去学习。创造一门新语言是一种乐趣。但我们不应让这遮蔽了用户的需求。除非你确实有一些真正原创的需求和想法，否则为什么不使用某种现有的语言，从而利用用户对该语言已经具备的熟悉度呢?

7.2.1 Choosing Languages

孤独的程序员（一个黑客）可以为任务选择最合适的语言。大多数工作中的程序员对他们将使用的语言几乎没有控制权。通常，这个问题是由尖头发型的老板决定的，他们做的是政治决定，而不是技术决定，而且缺乏勇气去推广一个非传统的工具，即使他们知道，通常凭借fi手的经验，那个不太被接受的工具才是最好的。在其他情况下，团队内部，甚至在某种程度上与更大社区的团结所带来的真实利益，排除了个人的选择。

8 Compromising Wisely

8.1 How to Fight Schedule Pressure

市场推出压力是指快速交付好产品的压力。它之所以是好的，是因为它反映了一个财务现实，并且在一定程度上是健康的。进度压力是指强迫比实际交付速度更快交付的压力，它是浪费的、不健康的，并且过于常见。

排期压力存在于多个原因。任务安排人员并没有充分理解我们作为程序员拥有多么强的职业道德，以及成为程序员有多么有趣。也许是因为他们将自己的行为投射到我们身上，认为要求我们提前完成任务会促使我们更努力地完成工作。这个观点或许在某种程度上是正确的，但效果微乎其微，且带来的负面影响却非常大。此外，他们无法了解开发软件的真实需求。由于无法看到这一过程，也无法亲自完成它，他们唯一能做的就是看到市场投放压力，并因此对程序员发火。

应对fi时间表压力的关键是简单地将其转化为市场时间压力。做到这一点的方法是使可用劳动力与产品之间的关系变得可见。生成一个诚实、详细且最重要的是易于理解的所有劳动力的估算是实现这一目标的最佳方式。它还有一个附加的好处，即能够做出关于可能的功能权衡的良好管理决策。

估算必须清楚表明的关键洞见是：劳动是一种几乎不可压缩的fl体。你无法在一段时间里塞进更多的劳动，就像你无法把超过容器体积的水装进容器一样。从某种意义上说，程序员的工作并不是说“不”，而是要说：“为了得到你想要的那个东西，你愿意放弃什么？”产生这种估算的 effect 将会是

提高对程序员的尊重。这正是其他专业人士的行为方式。程序员的辛勤工作将会被看见。制定不切实际的进度安排也会让所有人显而易见，而且令人痛苦。程序员不会被蒙骗。要求他们去做不切实际的事情既不尊重人，也会打击士气。极限编程 aplifies 这一点，并围绕它构建了一套流程；我希望每一位读者都能幸运地使用它。

8.2 How to Understand the User

你的职责是理解用户，并帮助你的老板理解用户。因为用户不像你那样深入参与产品的创建，她的行为会有些不ff同：

- 用户通常发表简短的言论。
- 用户有自己的本职工作；她主要会想到的是对你产品的小改进，而不是大的改进。• 用户不可能对你产品的所有用户拥有全局视野。

你的职责是给他们真正想要的东西，而不是他们说自己想要的东西。然而，在开始之前，最好先向他们提出你的方案，并让他们同意你的方案正是他们真正想要的，但他们可能没有这样的远见。你对自己在这方面想法的 confidence 应该有所变化。在判断客户真正想要什么时，你必须同时防范傲慢和虚假的谦逊。程序员受过设计和创造的训练。市场研究人员受过训练去 figure out 人们想要什么。这两类人，或者同一个人身上的两种思维方式，和谐地协同工作，最有可能形成正确的愿景。

你与用户相处的时间越多，就越能理解或判断什么才会真正成功。你应该尽可能多地向他们验证你的想法。如果可以的话，你应该和他们一起吃喝。

8.3 How to Get a Promotion

想晋升到某个角色，先扮演那个角色first。

要晋升到一个头衔fi，弄清楚该头衔的期望并做到。

要想获得加薪，应在掌握充分信息的情况下进行谈判。

如果你觉得自己早就该得到晋升了，就和你的老板谈谈。明确地问她你需要做些什么才能获得晋升，并尽力去做。这听起来很老套，但你对自己需要做什么的看法和你老板的看法往往会 differ 很大。而且这样在某些方面也会让你的老板给出明确说法。

9 Serving Your Team

9.1 How to Develop Talent

尼采说这句话时言过其实:

> 什么不摧毁我，便使我更强大。

你最大的责任是对你的团队。你应该了解他们每一个人。你应该挑战你的团队，但不要让他们不堪重负。你应该经常与他们讨论他们被挑战的方式。如果他们认同这一点，他们会有很高的积极性。在每个项目或每隔一个项目，尝试以他们建议的方式和你认为对他们有利的方式来挑战他们。挑战他们不是通过给他们更多的工作，而是通过给他们一个新技能，或者更好的是，给他们一个新的角色来扮演。

你应该允许人们偶尔失败，并在你的日程中为一些失败做好计划。如果从未发生过失败，就无法感受到冒险的意义。如果没有偶尔的失败，那就说明你没有足够努力。当某人失败时，你应该尽量温柔地对待她，但不要把它当作她成功了。

尽量让每个团队成员认同并保持良好的积极性。如果他们没有动力，明确询问他们需要什么才能保持积极。你可能不得不让他们感到不满意fi，但你应该知道每个人的需求。

你不能放弃一个故意因为士气低落或不满而没有尽自己责任的人，不能任由他们懈怠。你必须尽力让他们重新充满动力和生产力。只要你有耐心，就坚持下去。当你的耐心耗尽时，fi他们。你不能允许一个故意低于自己能力水平工作的人留在团队中，因为这对团队不公平。

向团队中强大的成员明确表示你认为他们很强，方法是公开表扬他们。表扬应该公开，批评应私下进行。

团队中的强成员自然会承担比弱成员更多的困难任务。这是完全自然的，只要每个人都努力工作，没人会因此感到困扰。

一个奇怪的事实是，良好的程序员比十个差的程序员更有生产力，这一点在薪水中并没有反映出来。这就创造了一种奇怪的局面。通常情况下，如果你的弱程序员能让开，你会发现自己可以走得更快。如果你这样做，实际上短期内你会取得更多进展。然而，你的团队将失去一些重要的好处，比如训练弱者、传播部落知识，以及在失去强者时的恢复能力。强者在这方面必须保持温和，并从各个角度考虑这个问题。

您可以经常给更强的团队成员分配具有挑战性但精心划定的任务。

9.2 How to Choose What to Work On

你在选择项目的工作内容时，会在个人需求与团队需求之间取得平衡。你应该做自己最擅长的事，但也应尝试通过运用一项新技能来让自己有所突破，而不是仅仅承担更多工作。领导力和沟通能力比技术能力更为重要。如果你非常强大，应承担最困难或风险最高的任务，并在项目尽可能早的阶段完成，以最大限度地降低风险。

9.3 How to Get the Most From Your Teammates

为了充分发挥队友的最大潜力，要培养良好的团队精神，并尽量让每个个体在个人层面上既受到挑战又保持投入。

为了培养团队精神，像印有标志的服装和聚会这样的老套东ff西是有用的，但不如个人之间的尊重来得好。如果每个人都尊重他人，就没有人会想让任何人失望。当人们为团队作出牺fi牲，并且从团队的整体利益而不是个人利益来思考时，团队精神就会形成。作为一名领导者，你不能要求别人付出超过你自己所付出的。

团队领导力的关键之一是促进共识，让每个人都有参与感和认同感。这有时意味着允许你的队友犯错。也就是说，如果不会对项目造成太大的伤害，即使你非常有信心认为这是错误的做法，只要团队在某种做法上达成了共识，你也必须允许团队中的一些人按他们的方式去做。你不必每次都这样做，但应该偶尔这样做。当这种情况发生时，不要表示同意，只需公开表达不同意见并接受共识。不要表现出受伤的样子，或者像是被迫接受一样，只需说明你不同意，但认为团队的共识更重要。这往往会促使他们反悔。如果他们真的反悔了，也不要坚持让他们继续执行最初的计划。

如果在你从所有适当的方面讨论过问题之后，仍有个别人无法达成共识，那就直接表明你必须作出决定，而这就是你的决定。如果有办法判断你的决定是否会是错误的，或者后来证明是错误的，就尽可能快地调整，并承认那些判断正确的人。

询问你的团队，以集体和个人两种方式，了解他们认为哪些因素能营造团队精神，并促成一支 effective 的团队。

多给予赞扬，而不要过度赞美。尤其是在值得称赞时，要赞扬那些与你意见相左的人。公开表扬，私下批评；但有一个例外：有时，对成长或纠正错误的表扬若公开进行，反而会尴尬地把注意力引回最初的错误，因此这种成长应当私下表扬。

9.4 How to Divide Problems Up

把一个软件项目拆分成将由个人完成的任务是件有趣的事。这应该尽早完成。有时管理者会

人们往往喜欢认为，在不考虑将要执行工作的个人的情况下也可以做出估算。但当我们知道个人生产率 differs 相差一个数量级，而同一人在 different 任务上的生产率 differs 也相差一个数量级时，这怎么可能呢?

就像作曲家通常会考虑演奏乐器的音色一样，经验丰富的团队领导通常无法将项目的任务分配与将要分配的团队成员分开。这也是高效团队不应被拆分的原因之一。

在这方面存在一定的风险，因为人们在依靠优势不断提升的同时，可能会感到厌倦，并且不会改善自己的弱点或学习新技能。然而，专业化是一种非常有用的生产力工具，但如果过度使用，则会适得其反。

9.5 How to Gather Support for a Project

为了为一个项目争取支持，创建并传达一个展示对整个组织具有实际价值的愿景。尽量让他人参与到你的愿景创作中。这不仅给了他们支持你的理由，还能从他们的想法中获得好处。逐个招募项目的关键支持者。在可能的情况下，展示而非仅仅说明。如果可能，构建一个原型或模型来展示你的想法。原型始终具有强大的说服力，尤其在软件领域，它远胜于任何书面描述。

9.6 How to Grow a System

一棵树的种子包含着成年体的思想，但尚未完全实现成年体的形态和潜力。胚胎生长，它变得更大。它看起来更像成年体，且拥有更多的用途。最终，它结果。随后它死亡，且其躯体滋养其他生物。

软件就是那样；我们有把它那样对待的奢侈。桥梁不是那样；从来不存在婴儿桥，只有一座 unfinished 的桥。

把软件看作是不断发展的，这是一个很好的思维方式，因为它让我们在没有完美的心智图像之前，就能取得有益的进展。我们可以从用户那里获得反馈，并利用这些反馈来纠正发展方向。修剪off 弱枝是有益的。

程序员必须设计一个可以交付和使用的finished系统。但高级程序员必须做得更多。她必须设计一条以finished系统为终点的成长路径。她的工作是从一个想法的雏形出发，构建一条路径，将这个小小的想法尽可能顺利地转化为一个有用的产物。

为此，她必须将最终结果清晰地想象出来，并以一种能让工程团队产生热情的方式传达出来。但她也必须向他们说明一条路径，这条路径应当从他们目前所在的位置出发，通向他们想要到达的地方，中间不能有大的跨越。整棵树在整个过程中都必须保持活着；它不可能在某个阶段死去，然后再被复活。

这种方法体现在螺旋式开发中。通过设置彼此间隔不太远的里程碑来标记沿途的进展。在高度竞争的

在商业环境中，如果能够尽早发布里程碑并赚取收益，即使它们距离一个精心设计的终点还很远，那是最好的。程序员的工作之一是通过明智地选择以里程碑为表达的增长路径，平衡即时收益ff与未来收益ff。

高级程序员承担着发展软件、团队和个人的三重责任。

9.7 How to Communicate Well

要进行良好的沟通，你必须认识到这有多么困难。它本身就是一项技能。而且，由于你需要与之沟通的人是有flawed的，这让事情变得更加困难。他们并不努力去理解你。他们说话不好，写作也不好。他们往往要么工作过度，要么感到无聊，至少在一定程度上更关注自己的工作，而不是你可能正在处理的更宏大的问题。参加课程并练习写作、公开演讲和倾听的一个优势在于，如果你在这些方面变得擅长，你就能更容易地看出问题出在哪里以及如何加以纠正。

程序员是一种社会性动物，她的生存依赖于与团队的沟通。高级程序员是一种社会性动物，她的满足感依赖于与团队以外的人进行沟通。

程序员从混乱中带来秩序。一种有趣的方式是发起某种形式的提案，通常是在团队之外进行。这可以通过*strawman*或白皮书格式，或者仅仅是口头方式进行。这种领导方式具有设定辩论条款的巨大优势。它也使人暴露于批评，甚至更糟的是，拒绝和忽视。高级程序员必须准备好接受这些，因为她拥有独特的力量，也因此承担着独特的责任。那些不是程序员的企业家需要程序员在某些方面提供领导力。程序员是连接思想与现实之间的桥梁的一部分，而这部分建立在现实之上。

9.8 How to Tell People Things They Don't Want to Hear

你将经常不得不告诉人们一些让他们感到不舒服的事情。记住，你这样做是有原因的。即使问题无法解决，你也尽早告知他们，以便他们能充分了解情况。

告诉某人问题的最佳方式是同时提出解决方案。第二好的方式是请求他们帮助解决问题，这通常会让他们感到fl自信。如果有可能不被相信，你应该为你的主张收集一些支持。

最令人不愉快且常见的事情之一就是你必须说，“进度表必须推迟。”那些对进度表有高度责任感的程序员不愿意说这个，但必须尽早说出来。当一个里程碑推迟时，最糟糕的事情就是不立即采取行动，即使唯一的行动是通知每个人。在做这件事时，最好是作为一个团队来做，至少在精神上是这样，如果不是身体上。你会希望团队对这两方面提供意见。

你所处的位置以及可以采取的应对措施，而团队将不得不与你一起面对后果。

9.9 How to Deal with Managerial Myths

“神话”一词有时意味着fiction。但它有更深层的内涵。它还指一种具有宗教significance的故事，用来解释宇宙以及人类与宇宙之间的关系。管理者往往会忘记他们作为程序员时学到的东西，并相信某些神话。试图说服他们这些神话是错误的，就像试图让一位虔诚的宗教信徒放弃其信仰一样，既粗鲁又不会成功。基于这个原因，你应该认识到这些神话的真实面目：

- 更多的文档总是更好。（他们想要，但又不想让你为此花任何时间。）
- 程序员可以被视为等同。（程序员之间的差异可能相差一个数量级。）
- 可以向项目中增加资源以加快进度。（与新增人员进行沟通的成本往往弊大于利。）
- 可以可靠地估算软件开发。（这在理论上都不可能。）

这些迷思中的每一个都强化了管理者的想法：他们对正在发生的事情拥有某种实际的控制力。事实是，优秀的管理者起到的是促进作用，而糟糕的管理者则会造成阻碍。

9.10 How to Deal with Organizational Chaos

组织中常常会出现极度混乱的时期。这对每个人来说都令人不安，但对那种将个人自尊建立在自身能力而非职位之上的程序员来说，也许要稍微没那么不安一些。组织性的混乱是程序员施展其魔法力量的绝佳机会。我把这一点留到最后说，因为它是一个深藏的部落秘辛。如果你不是程序员，请现在就停止阅读。

> 工程师拥有创造与维系的力量。

非工程师可以对人发号施令，但在一家典型的软件公司里，他们自身什么也创造不了，所拥有的权力也只是工程师赋予的。没有工程师，他们既无法创造任何东西，也无法维持任何东西。这种力量几乎可以让你免疫与组织混乱相关的所有问题。当你拥有它时，你应该完全无视混乱，像什么都没发生一样继续前行。你当然也可能会被fired，但如果真发生了，凭借这种魔力你很容易找到一份新工作。更常见的是，某个压力山大、没有这种魔力的人会走进你的隔间，告诉你去做一些愚蠢的事情。最好的做法是微笑点头，等他们走开后，再继续做你知道对公司最有利的事。

这一行动方针对你个人而言是最好的，也对你所供职的公司最有利。如果你是一名领导者，告诉你的团队成员也这样做，并告诉他们忽略除你之外任何人对他们的指示，包括你自己的上级。

9.11 Request for Feedback

请把你对这篇文章的任何意见发给我。这是一项尚在进行中的作品，尚未经过任何形式的编辑、校对或审阅。

谢谢。

罗伯特·L·里德 <read@hire.com>

A Glossary

- Unk-unk — (是 unknown-unknown) 的俚语，指那些目前甚至无法被概念化的问题，它们会从项目中偷走时间并破坏进度安排。

• 赢家通吃式 — 如果竞争的回报更多地基于竞争者相对成功的排序，而不是他们实际成功的程度，那么这种竞争就是赢家通吃式的。比赛是赢家通吃式的。两个建房者可能彼此竞争建造最好的房子，但如果失败者仍然会因她建造的房子而获得回报，那么他们的竞争就不是赢家通吃式的。 • Printlining — 在程序中严格临时性地插入语句，以输出有关程序执行的信息，用于调试的做法。 • Logging — 将程序编写成能够生成描述其执行过程的可配置输出日志的做法。 • 分而治之 — 一种自顶向下设计的方法，更重要的是一种调试技术，即将问题或疑难分解为逐步更小的问题或疑难。 • Vapor — 虚幻且常具有欺骗性的承诺，指那些尚未出售、且往往永远不会成形为任何实物的软件。 • Boss { 给你设定任务的人。在某些情况下，用户就是老板。 • Tribe — 与你共享对共同目标忠诚的人们。• 低垂的果实 — 以很小代价就能获得的巨大改进。

- 垃圾 — 对象是指不再需要但仍占用内存的对象。
- 企业家 — 项目的发起者。

References

[1] Kawasiki, Guy, Moreno, Michelle, 和 Kawasaki, Gary. 2000. 《革命者法则：资本家宣言，创造和营销新产品与服务》. HarperBusiness.

[2] McConnell, Steve. 1996. 《快速开发：驯服疯狂的软件进度》。雷德蒙德，华盛顿：微软出版社。

[3] 麦康奈尔，史蒂夫。1993。《代码大全》。美国华盛顿州雷德蒙德：微软出版社。

[4] 肯特·贝克。《解析极限编程：拥抱变化》。

[5] Beck, Kent 和 Fowler, Martin. 极限编程规划。

[6] Graham, Paul. 2002. 他的文章集： http://www.paulgraham.com/articles.html。所有文章，但特别是《击败平均水平》。

[7] 雷蒙德，埃里克·史蒂文。2002年。《如何成为黑客》： http://www.tuxedo.org/esr/faqs/hacker-howto.html

[8] Hunt, Andrew, Thomas, David, 和 Cunningham, Ward. 实用程序员：从学徒到大师。