第 7 章

欲善其事，先利其器：使用工具构建软件

本章主题：

- 我们用来构建代码的工具
- 有效地使用工具
- 常见的工具类型

任何胆敢使用超乎自己力量的装置，都会身陷危险。

——J.R.R.托尔金（J.R.R. Tolkien）

要想成为一位多产的艺人，你需要有一套顺手的工具。水暖工工具箱里的东西可以帮助他完成任何任务，要不然你就不会在下次家里的水龙头漏水时去叨唠他了。

只是拥有这些工具还不够，它们的质量也很重要。差劲的工具会让人对优秀的工匠感到失望。无论你的水暖工有多能干，如果压缩阀不好，也会到处都是水。

当然，是你对这些工具的使用使你成为一名杰出的工匠。工具本身什么也做不成。在电动工具出现之前，木匠们就已经能做出精美的家具了。工具相对而言是基础的，使用工

具的技能才是创造精美物品的关键。

编程也是同样的道理。要把工作做好，你需要得到一套适当工具的支持；这应该是一套让你充满信心的工具，你知道如何使用它们，对你所遇到的工作也非常适用。要创造出非凡的代码，不仅需要有技艺精湛的编程高手，还要有好用的工具和灵活运用这些工具的能力。

这是一个重要的问题。你使用工具的方式可以看出你是否能成为一名真正多产的程序员。在极端的情况下，这些工具可以提供决定你的项目成功与否的简化操作。软件工厂那不懈的前进步伐，要求你紧紧抓住任何可以帮助你编写更好的代码，以及更快和更可靠地编写代码的工具。

其他章节会包含一些涉及某种特定工具的内容。本章我们将把软件工具作为一个整体来讨论。编程是一项没有工具就无法进行的工作。我们日复一日地使用着工具，使用编译器就像使用开罐器一样自然，没有经过太多的思考。如果它运转正常，就没有任何问题，但是当它发生了故障（或者你需要开启一个奇形怪状的罐头）时，不管开罐器有多高档，你都会被卡住。一个简单便宜但是能用的开罐器要好过一个外表华丽构造复杂但是不能用的装置。

7.1 什么是软件工具

我们用于构建软件的工具多种多样，如果用通俗的语言来讲，可以说这些工具是构建程序的程序。我们用来创建软件的所有东西都是一种形式的工具。一些工具帮助你编写代码。另一些工具帮助你编写优秀的代码。还有一些工具帮助你把刚创建出来的一团糟的代码整理出头绪。

工具的形式和大小各式各样，使用方式也有所不同。工具所处的平台和环境显然是一个因素，不过它们在以下方面还有所不同：

复杂性

有些工具是精心设计的环境，具有许多功能和不可思议的可配置性。还有一些则是为单个任务而设计的小巧的工具。这两种工具各有其优缺点：

- 功能丰富的工具很酷，特别是当你终于学会了如何用它来煮咖啡，同时还能给你送来炸面包圈的时候。但是如果过多的神奇功使它难以使用，那么它的作用就会受到影响。
- 简单的工具易于学习，其功能显而易见。你会有很多这样的工具，每个工具完成

一项任务。但是如果你把它们拼凑起来，就会发现有太多的接口点，这使得这些工具不能始终无缝地协同工作。

不同的工具有不同的应用领域，小到非常具体的任务（在文件中搜索文本字符串），大到整个项目（综合项目管理环境）。

使用频率

有些工具使用得非常频繁，离了它们我们就没法生活。而还有一些工具我们好久才会用一次，但是当你需要它们时就会发现它们价值连城。

接口

有些工具有着非常美观的图形用户接口（GUI）。另一些工具的接口则比较简单，这些工具由一个命令行接口（CLI）驱动，并且会将它们的输出定向到文件中。喜欢使用哪种工具取决于你的思维方式和你的习惯。

Windows 工具往往具有不能通过命令行操作的图形接口。标准的 UNIX 工具正好相反，这使它们更容易自动运行，也更容易使用脚本集成到较大的工具中。接口会改变你驾驭工具的方式。

集成

某些工具适合较大的工具链，常常包含在一个图形接口的集成开发环境（IDE）中。独立的命令行工具主要用作一种数据过滤器，倾向于生成纯文本的输出，其输出的格式适合作为其他工具的输入。

集成的 GUI 接口使用起来很舒服，而且集成极大地提高了你的工作效率。另一方面，设置这种接口和让你喜欢上它们一样需要很多时间，而且它们所提供的功能往往比手动的命令行工具少。但是尽管命令行接口的工具很强大，彼此分立的 UNIX 工具却各有不同的隐含接口，这使它们难以使用。

成本

有许多免费工具都很优秀[①]。然而，一般来说付出多少就得到多少。免费工具的文档

① 免费在软件世界中有两种含义：和啤酒一样免费（不用任何花费即可获得），以及和演讲一样免费（开放源码的软件，你可以查看和修改其代码）。哪种免费更重要取决于你在多大程度上是一位理想主义者。参见第 397 页的“许可”。

往往比较差，支持较少，并且功能集也较小。不过这并不是绝对的。有些免费工具就要比它们的商业竞争对手好得多。

对于任何类型的工具，你想付多少钱都可以，但是更高的价格并不能保证更好的质量。我曾经使用过一些非常差劲却贵得出奇的工具。高昂的价格会让人对质量产生错觉……

质量

有些工具确实不错。有些工具则确实不好。我用过一些处于临界状态的工具，我很庆幸再也没见过它们了。这些工具能够完成任务，但是很勉强，并且总是处在崩溃的边缘。但是如果不使用这些工具，我就开发不出我需要用来换工资的代码。我曾无数次冲动地想亲自重写这些工具。现在我还抱有这样的梦想。

你可以根据以上这些特性来挑选工具，并做出适当的妥协。虽然习惯于你平常使用的工具，学习并熟练使用它们很重要，但是千万不要迷信它们。大多数 Windows 用户都很看不起 UNIX 风格的程序开发，同时 UNIX 的高手们也因为 Windows 的程序员不会使用命令行而很小看他们。克服这种心态。

我鼓励你在一个大型项目中尝试在不同的编程环境下工作。这有助于让你充分理解是什么造就了好的工具链，并且帮助你从全局的角度了解软件工具。

7.2 为什么要在意工具

如果没有一套核心的软件工具，就不可能创建程序；没有编辑器或编译器你将寸步难行。还有些工具也许你没有也并无大碍，但这些工具确实很有用。为了提高你的效率、代码质量和编程技术，花一点精力关注一下你当前所使用的工具，并搞清楚这些工具都有哪些功能，将不无裨益。

如果你明白你的工具是如何工作的，以及哪个工具应该用于哪项工作，你就能更好而且更快地编写出运行良好的代码。更聪明地使用工具，会让你变为更聪明的程序员。

关键概念 **尽可能全面地了解你的常用工具。为了熟悉这些工具而投入的少许时间会让你很快就有所收获。**

让我们弄明白我们到底为什么而使用工具：工具不是替我们做我们该做的工作，而是

使我们有能力做我们的工作。软件的质量总是取决于程序员的能力。下次当你的编译器吐出一页又一页的错误消息时，想一想这句话。代码是你写的，笨蛋！

在工具的选择和使用方面，程序员们的态度差别千变万化。在这些不尽相同的态度背后，也许有着一些深层的心理原因——一些关于你是否是一位绝世天才的认知。当碰上一项新的冗长而乏味的任务时：

- 有些程序员勤勤恳恳地用人工来完成。
- 有些程序员则使用脚本语言编写一个工具，以便自动地完成工作。
- 还有些程序员会花费数小时，搜索已编写好的工具来替他们完成工作。

面对一个也许可以解决问题的工具：

- 有些程序员不断尝试这个工具，直到得到他们希望得到的结果为止。
- 还有些程序员则仔细地阅读文档，以弄明白到底能用这个工具做些什么，然后再开始使用它。

哪种方式是正确的？实际上，这要依情况而定。成为一名成熟的程序员的条件之一，就是明白不同的情况需要不同的解决方案，并将正确的工具应用到正确的工作上。每个人都是不同的，每个人的工作方式也各不相同——你的同事们也许不使用你最喜欢的工具也能达到最高效率。但是如果你看到有人在日复一日地手动将 C 语言的代码转换为汇编语言，你就会怀疑他的精神是否正常了。

切合实际地向工具投入时间和金钱。想一想你准备如何使用一个工具。仅在付出会有所回报的情况下，再去搜索或编写新的工具。不要花一个星期来编写一个每月只能为你节约一个小时的工具。但是一定要花一个星期来编写一个每天都为你节约一个小时的工具。

关键概念　用实际的眼光来看待软件工具——仅当工具能够让你的生活轻松一些的时候再使用它们。

7.3　使工具发挥作用

由于编写程序和使用工具紧密相关，为了成为一名超级程序员，你首先需要是一位超级工具使用者。这意味着什么呢？

首先，你需要清楚地了解手头有哪些工具，这很重要。在下一节中，我们会列出每个程序员在手边都应该有的一些常用工具。你不必了解市场上的每一种工具；这会造成晚宴上无聊的谈话内容。只了解现有工具的大致分类而不是每种具体的产品，才是向前进的重要一步。这会帮助你在面对某项任务时，更明白是应该去搜索一个工具，还是自己编写一个工具，或手动完成这项任务。

花一些时间，让自己了解的信息更加充分。查一查在哪里可以获得其中的某些工具——有许多专营软件工具的商店，互联上也有很多可以下载工具的网站。也许你已经安装了一些工具，但是从来没有使用过它们，甚至不知道这些工具究竟有什么用。了解你可以期望这些工具为你做些什么，这将使你准备好更好地使用工具。

关键概念　了解可用工具的种类。确保自己知道从哪里获得这些工具，即使你现在还不需要它们。

时刻准备试用新的工具，并花些时间来学习它。这是一种健康的态度。当启动一个新项目，迁移到新平台，遇到新的问题，或者发现你的旧工具已过时的时候，你可能会被迫寻找新的工具。但是不要在不得已的情况下才这样做——确保现在你使用的工具就是你所能获得的最好的。

匀出一部分时间来磨练你使用工具的技巧——就像你会花时间来阅读技术书刊，或参加职业培训课程那样。这一点非常重要，所以根据情况投入一些时间吧。

下面是一些会使你成为工具的超级用户的简单步骤。对于你的软件开发军火库中的每一件武器……

7.3.1　了解它能做些什么

查清它的功能有哪些，即它究竟能做些什么，而不是你觉得它应该能做些什么。即使你不知道如何完全利用某个工具的全部功能（也许你不得不查阅更加深奥难解的命令行参数），了解它能做什么也是有好处的。

有没有什么这个工具不能做的特殊任务？也许它并不支持竞争对手提供的某些功能。明白这些局限性，你就知道应该在什么时候购买更好的工具了。

7.3.2　学习如何驾驭它

仅仅因为你在运行某个工具时没有出错，并不能说明它是完全按照你的要求完成工作的。你必须懂得如何正确地使用它，并确信你可以让它执行你的命令。

这个工具是如何融入到整个工具链中的？这将影响你使用它的方式。例如，通过管道相互连接（将一些较小的工具拼装为一个较大的工具）的 UNIX 工具可以用做有序的过滤器[①]。理解如何驾驭每个工具的功能，并了解它们如何相互操作，会让你的工具使用水平提高一个层次。

找出使用每种工具的最好方式——可能不是直接调用它，或在 GUI 接口中单击某个按钮。它能自动触发吗？编译器通常是通过一个构建系统而不是通过手动激活的。

① 如果你对此知之甚少，我劝你多研读一些相关的内容。UNIX 命令“man bash”是一个很好的入手点，在 man 页面搜索 pipeline。

7.3.3 了解它适合什么任务

了解每个工具是如何融入到其他可用工具的环境中的。例如，我可以在文本编辑器中创建一个按键记录宏，这样就避免了重复的操作，从而节约了我的时间。使用神奇的 sed 工具，也可以实现一些对文本的修改操作[①]。不过在这种情况下，最好还是使用按键宏——因为我已经在使用编辑器了，所以可以更快地完成这些操作。

你也许并不知道如何使用 yacc[②]，但是当你需要编写一个词法分析器时，如果你知道存在这个工具，就会节省大量的工作。

关键概念　将合适的工具用到合适的任务上。不要拿着牛刀去杀鸡。

7.3.4 检查它是否可用

每个人都有可能在某个时刻成为糟糕工具的牺牲品。你的代码不能正常运行，但是不管你如何分析那个错误行为，都找不出它的原因。绝望中，你开始随意地进行一些检查——看看是不是在刮东风，以及是不是已正确地安装了照明设备。几个小时后，你却发现有一个非常古怪的工具正在做一些意想不到的事。

编译器可能会生成有缺陷的代码。构建系统可能会使从属关系出现错误。库中可能会隐藏着 bug。在拔光自己的头发之前，你要学会如何查出明显的错误。

对于诊断你所遇到的任何问题来说，拥有你的工具的源代码非常有用，这会让你查出这个工具到底在做什么。这也许是你选择工具时的一个决定性因素。

7.3.5 找到了解更多信息的途径

你不必事事都了然于心。关键是要认识那些明白的人！

找出工具的文档在哪里。提供支持的是谁？如何获得更多的信息？查查手册、发行说明、在线资源、内部帮助文件和参考指南等资料。了解这些资料在哪里，以及如何在需要

---

① sed 是一种流编辑器命令行工具，下一节将对该工具进行详细的介绍。
② 一种词法分析生成器。不要担心——后面也会对这种工具进行介绍。

的时候获得它们。它们的联机版本是否有有用的搜索工具和好的索引？

7.3.6 查明新版本何时出现

工具似乎总是在以不可思议的速度发展着——在这个行业中，科技日新月异。有一些工具的发展要比另一些工具快得多。当作者发布一个已经更新过多次的新版本时，你也许才刚刚安装了最新的 Widgetizer 不久。

了解你所使用的工具的最新信息非常重要，这样你就不会过时，也不会一直使用着有很多缺陷并且缺乏支持的工具套件。但是更新工具时必须保持谨慎，不要盲目地追逐最新的版本。尝试新事物是有风险的！

新版本也许会有新的缺陷更高的新价格。只有当新版本提供了重大的修改并且已证明是稳定的时，再进行更新。先测试一下——对你的老代码全面地检查新工具，以确保它运行正常。

关键概念　了解你的工具的最新发展情况，但是不要随便进行升级。

7.4 哪个工具

软件开发工具多得让人吃惊。多年来，这些工具被开发出来以解决各种具体问题，这种需求往往会突然出现。如果某项任务已被完成过很多次，那么你就可以肯定有人已经为完成它而编写了工具。

你的工具套件到底包括哪些工具将取决于你的工作类型。用于嵌入式平台的工具，其功能很少能够比那些用于桌面应用程序的工具更丰富。下面我们将对一些常见的工具进行讨论。其中有一些很显而易见，而另外一些则不那么明显。

虽然我们会单独地讨论每一类工具，但是不要忘记现代的 IDE 已经将这些不同的程序整合成一个单一的、高效的接口。这个集成的接口当然很方便，但是了解每个工具是如何独立工作的也非常重要，原因如下。

- 你会明白如何最好地利用 IDE 的每种可用功能。
- 你会明白你的 IDE 缺少哪些有用的功能。

大多数 IDE 都是模块化的——你可以用一个更好的组件来替代其中的某个组件，也可以将目前没有的功能添加进去。了解工具的类型有哪些，这样你会增加你的 IDE 经验。

7.4.1　源代码编辑工具

制陶工人的媒介是黏土，雕刻家的媒介是石头，而程序员的媒介是代码。这是我们所处理的最基本的东西，所以选择一个出色的工具来帮助我们编写、编辑和研究源代码非常重要。

源代码编辑器

源代码编辑器可能是你最重要的工具，甚至比编译器还要重要。编译器面对的是计算机，而编辑器面对的是你。并且进行操纵的是你。你的编程生涯大半将在这里度过，因此选择一个好的编辑器并学会使用它真的非常重要。有效地利用你的文本编辑器将极大地提高你编写代码的能力。

关键概念　代码编辑器的选择至关重要：这对你如何编写代码影响甚大。

哪一种源代码编辑器最好是一个历时已久的争论话题，在此没有必要再引发争端，不过你应该选择一个你用起来感觉很舒服并且能够完成你的任务的编辑器。不要仅仅因为某个编辑器嵌入在你的可视化 IDE 中，就把这个编辑器当作是最好的。另一方面，你也许会发现将编辑器集成在 IDE 中确实能给你带来很多实惠。在源代码编辑方面，我要求我的编辑器至少具有以下特点：

- 易于理解的语法着色（支持多种语言，因为我需要使用很多语言）；
- 简单的语法检查（例如，高亮显示不匹配的括号）；
- 良好的渐近式搜索工具（一种交互的查找方式，在你输入的同时进行搜索）；
- 键盘宏录制；
- 高度地可配置；
- 可以在我所使用的所有平台上运行。

我对编辑器的要求和选择可能与你的不一样，不过上面这些要求可以看作是一个最重要的功能的合理列表。我不介意花一些时间来学习如何更好地利用所有这些功能。如果这能让我更有效率，那么就是值得的。

根据你的工作性质，你可能会发现其他一些类型的编辑器很有用。这些编辑器包括二进制文件编辑器（通常以十六进制来显示文件的内容，常被称作十六进制编辑器），以及专门用于特定文件格式的编辑器（例如 XML 文件编辑器）。

Vim 和 Emacs 是非常著名的基于 UNIX 的编辑器，现在已经发行了适用于几乎所有平台的版本（甚至可能在你在电烤箱上都能用）。这些编辑器和那些与 IDE 绑定在一起的默认编辑器形成了鲜明的对比。

源代码处理工具

UNIX 的理念体现在大量较小的命令行工具上。GUI 环境具有与每个工具相对应的功能，但是这些功能很少有工具那么强大，而且很难协同工作。不过 GUI 版本学习起来要容易得多。

下面的一些 UNIX 命令提供了一些强大的机制，用于分析并修改源代码：

diff

比较两个文件并高亮显示两者的区别。基本的 diff 会将输出传送到控制台，但是也存在更高级的图形版本。甚至还有一些编辑器可以处理用 diff 比较过的文件，这些编辑器将文件并排显示，并随着你的输入随时更新文件的区别。有些特殊的 diff 工具可以一次比较三个文件。

sed

sed 代表流编辑器（stream editor）。它运用一种特殊的转换规则，一次阅读文件的一行。sed 可以用作全局搜索和替换工具，对条目进行重新排序，或在行中插入样式。

awk

想像一下在 steroids 上运行的 sed。awk 是另一种可以处理文本文件的样式匹配程序。它为这一任务提供了完整的编程语言，因此你可以编写十分高级的 awk 脚本，以执行相关的处理。

grep

该工具用于在文件中搜索文字的样式。这些样式由正则表达式描述。正则表达式是一种袖珍的语言，允许使用通配符和灵活的匹配标准。

find/locate

这两个工具用于在文件系统中查找文件。它们可以根据文件名、日期或许多其他的标准来查找文件。

以上这些工具只不过是冰山一角，除此之外还有很多其他的工具，例如用于计算字数

和字符数的 wc。像 sort、paste、join 和 cut 等都是非常有用的工具。

源代码浏览工具

真正的大型项目有着像城市一样复杂的代码库。即使是城市的规划人员也无法对每个街区都了如指掌。出租车司机们知道周围哪条路最好走。一般的居民对他们自己的周边地区非常熟悉。而旅客则会下了车便分不清东南西北。

你可以使用一系列的工具来帮助自己研究和理解代码、设计代码、轻松地执行搜索、浏览和交叉引用等操作。有些工具可以生成调用关系树，这样你就可以了解控制在系统中的流动方式。这些工具可能会生成一个图形化的流程图，或者与你的编辑器集成在一起，以提供自动执行、函数调用帮助等功能。这些功能对于大型的代码库或者结构良好的项目非常有价值。

LXR、Doxygen 和值得尊重的 ctags 都是很好的免费工具。

版本控制

我们在此不对源代码控制工具进行详细的论述，第 387 页的“源代码控制”将专门讨论这类工具。这里只说一句：你必须使用一个源代码控制工具，否则你就会像缺了左膀右臂一样。

源代码生成

有许多工具可以自动生成源代码。这些工具中有一些非常好，也有一些让我感到害怕。

一个例子是 yacc，这是一种 LALR(1)[①]分析生成器。你定义输入的语法规则，然后使用 yacc 生成可以分析匹配这些规则的形式良好的程序。yacc 生成一个 C 代码分析器，其中的钩子可以用来在分析条目时添加功能。bison 是一个与之类似的工具。

还有一类帮助你设计用户接口、生成耗费时间的后端代码的代码生成工具。这些工具主要用于复杂的 GUI 工具套件，如 MFC。如果一个库需要一个工具来做这件烦琐的工作，那么首先就意味着这个库太复杂了（或者已经损坏）。小心对待！

对于那些会写出大量你必须在日后进行修改和校正的框架代码的向导，你也必须小心对待。在对这些生成的代码动手之前，你必须真正理解它们，否则你会被自己对它们的认识不足击倒。如果你在修改了任何已生成的代码之后重新运行向导工具，你的所有手工编辑都将会被悄无声息地覆盖。天哪！

① 一种对相当复杂的语法进行分析的方式。

你甚至可以编写自己的脚本，来生成重复的代码段。有时这表明你的代码本来可以设计得更好。而有时这是正确的技术方法。过去我曾经编写过为我自动生成代码的 Perl 脚本。因为这个代码生成器是我亲手编写的，所以我信任它所生成的代码。另一位程序员可能会以不信任的眼光来看待这些代码，就像对待其他的代码向导一样。

源代码美化工具

这些工具用于通过创建一致的“最小共通特性”版面，来统一源代码的格式。老实说我觉得它们的弊大于利——在使用这些工具进行修改之后，一些重要和有用的格式可能会遭到破坏。

7.4.2　代码构建工具

我们并不喜欢整天盯着漂亮的源代码看。真正的乐趣在于让它做点什么。我们常常这样做，以至于理所当然地使用下面这些工具，并且假定它们的运行没有问题，而根本不去想后台究竟是怎么一回事。

编译器

除了源代码编辑器之外，编译器是第二个最常用的软件工具。它将你的源代码转换成可执行程序，这样你就可以看看你的程序运行时会出现哪些问题了。因为这个工具几乎天天都要使用，所以你是否能够正确地驾驭它就显得格外重要。你是否真的知道它所具有的所有选项和功能呢？许多公司都拥有一个特殊的 buildmaster 工具，以确保构建工具被正确地使用，但是这并不能成为你不了解你的编译器的借口。

- 你是否明白要优化到什么样的程度，以及这对生成的代码有什么样的影响？这很重要——这将与其他一些事情一起，决定代码在调试器中运行时会出现多少意外，甚至还决定了你会使哪些编译器 bug 出现！
- 你是否在进行编译时打开了所有的警告？不打开所有的警告是没有道理的（除非你正在维护已充满了警告的遗留代码）。警告强调了潜在的错误，如果没有出现任何警告，你就会对你的代码抱有更大的信心。
- 编译器是否默认遵循标准？C++ ISO 标准是（ISO 98），1999 C 标准是（ISO 99），Java 语言是由（Gosling et al. 00）定义的，而 C#语言是由 ISO 标准（ISO 05）定义的。编译器是否有任何非标准的扩展？如果有，你是否知道这些扩展是什么，以及如何避免它们？
- 编译器生成的代码是否符合正确的 CPU 指令集？你也许生成了大量兼容 386 的代

码，但是你却只准备在最新的 Intel whiz-bang 芯片上运行它们。让你的编译器尽可能生成最合适的代码。

我需要一个工具……

你需要执行一项任务。这是一项枯燥乏味的任务。它不断地重复出现。这种类型的任务最好由计算机来做，这样可以减少错误，避免冗长烦琐的人工操作，并且速度也更快。电脑就是因为这个原因而被发明的！你如何才能知道是否有某个工具可以替你做这项工作呢？

- 如果上面的列表中提到了这项任务，那么你就会已经知道有一个可用的工具。
- 如果上面的列表中没有这项任务，但是你可以肯定你不是第一个碰到这个问题的人，那么很可能存在某种可以帮助你的工具。简单地在网络上搜索一下，你会对找到的工具感到吃惊。
- 如果你的问题是独一无二的，那么你可能不得不亲自编写一个程序。更多信息请参见第 126 页的“自己动手，丰衣足食”。

在寻找工具的时候，你应该尽可能多地听取别人的意见。

- 问问团队中的其他人是否有什么经验。
- 搜索网络，读一读相关的新闻组。
- 向工具供应商咨询。

面对许多可供选择的工具，你需要根据我们在第一节中提到的标准进行明智的选择。为了做出决定，你必须确定你的需求。工具是否免费重要吗？或者是否你现在就可以得到它更重要一些？工具是否应该让团队中的每个人都可以轻松地使用？你是否要经常使用它——这与购置工具的费用相比是否合算？

“交叉编译器”（cross compiler）的目标是一个与进行开发的计算机不同的平台。这种工具主要用于编写嵌入式软件（毕竟在洗衣机上运行 Visual C++是相当困难的）。

编译器是大型工具链上的一部分，工具链中还包括链接器、汇编器、调试器、分析器以及其他对象文件的操纵器等。

gcc、Microsoft 的 Visual C++和 Borland 的 C++ builder 等都是一些流行的编译器。

链接器

链接器与编译器紧密相联。它获取编译器所生成的所有中间对象文件，并将其聚合成一个单独的可执行代码整体。C 和 C++的链接器与编译器结合得如此紧密，以至于有时是同一个可执行程序来完成这两个任务。对于 Java 和 C#语言而言，链接器是与运行时环境结合在一起的。

当使用你的链接器时，确保你知道：

- 它是否会剥出二进制？也就是说，它是否会移除像变量名和函数名那样的调试符号？调试器可以使用这些调试符号，以显示有用的诊断信息，但是这些符号也可能会使可执行程序变得非常大，从而减慢了加载速度。
- 它是否会删除重复的代码段？
- 你是否可以让链接器生成库对象，而不是可执行程序呢？你对库能够进行哪些控制——你是否可以静态或动态地加载库？

构建环境

完整的构建环境不止是一个编译器和链接器。我们所使用的构建工具是 UNIX 的 make 程序或者 IDE 中的构建部分。这些工具将自动完成编译过程。许多开放源码的 UNIX 项目使用 autoconf 和 automake 工具来简化构建过程。

学会如何最有效地利用你的集成构建环境，但是不要因此而了解如何使用每个单独的构建工具。我们将在第 10 章更详细地讨论这些主题。

测试工具链

注意，测试工具是代码构建工具，而不是调试工具！适当的测试对于制作可靠的、高质量的软件是至关重要的。测试常常被忽略——也许是因为看上去测试的工作量太大，从而把精力从编写代码的重要任务中分散了。这样做对于开发优秀的软件是最大的威胁之一。除非你能证明代码可以正常地运行，否则你就无法得到可靠的代码，而证明的唯一方法是一边编写代码一边进行测试。

有一些工具可以帮助你自动执行单元测试，它们提供了一个框架，你可以在这个框架中放入你的测试代码。这些工具可以很容易地集成到你的构建系统中，这样测试就成为了代码构建过程的一个核心部分。

与自动进行单元测试的工具一样，有一些工具可以生成测试数据和创建测试用例。还有一些工具可以模拟目标平台，也许还具有模拟特定错误情形（如内存不足、负载过高等）的能力。

7.4.3 调试和调查工具

这些工具描述了运行代码的特征，有助于找到问题所在——既包括我们已看到的错误，也包括等待时机爆发的潜在灾难。我们将在第 169 页的“除蜂剂、驱虫剂、捕蝇纸……”中对这些工具进行详细的讨论。

调试器

拥有一个高质量的调试器并了解如何使用它，可以为你节省大量用于追查意外行为的开发时间。调试器使你可以对程序中的执行路径进行分析，中断程序的执行，调查变量的值，设置断点，以及通常将运行的代码分割成不同的部分。它远不止像在程序中放置许多 printf 日志语句那么简单！

gdb 是 GNU 的开放源码调试器，它已经被移植到几乎所有可以想到的平台上。ddd 是 gdb 的一个功能丰富的图形接口。所有 IDE 和工具链都有其自己的调试器。

分析器

如果你的代码运行的速度太慢，那么就要用到这个工具。分析器用于分析代码各部分的运行时间并找出瓶颈所在。使用分析器可以为切合实际的优化找到优化对象。有了要优

化的对象，你就不会再去浪费精力来加速几乎不会执行的代码。

代码校验器

代码校验器分为静态的和动态的两类。前者以一种与编译器类似的方式整理代码，检查你的源文件，以确定可能存在问题的区域以及对语言的错误使用。lint 是一个广为人知的例子，它对 C 语言中一系列常见的编码错误执行静态的检查。静态校验器的大部分功能已内置在现代编译器中，不过仍然有一些独立的工具可以用来执行额外的检查。

动态校验器在代码被编译时对代码进行修改和插装，然后在运行时执行检查。内存分配/边界检查器就是很好的例子——这两个工具确保了所有动态分配的内存都被正确地释放，并且对数组的访问没有越界[①]。这些工具可以节省大量为查找不明显的 bug 而花费的时间。它们在大多数情况下都比调试器更有用，因为它们更像是一种预防机制，而不是单纯的补救：它们将在代码缺陷有机会破坏你的程序之前找到它。

度量工具

这些工具用于执行代码检查，它们通常的形式为静态分析器（尽管动态度量工具确实存在）。它们会生成关于代码质量的统计评估。虽然统计数字很容易产生误导作用，但是这些工具可以有效地指出最脆弱的代码区域。这些信息可以帮助你挑出具体的目标来进行代码审查。

度量数据通常是以函数为基础来收集的。最基本的度量数据是代码的行数，其次就是注释与代码的比率。这两个数据对于你来说并不是特别有用，不过除此之外还有许多更令人感兴趣的度量数据。圈复杂度（cyclomatic complexity）是代码复杂性的度量数据，它考虑了决定点和潜在控制流的数量。较高的圈复杂度预示着难以理解的代码，这些代码很可能比较脆弱或存在缺陷。

反汇编程序

这种工具深入到可执行文件中，使你可以检查程序的机器代码。调试器确实包含对这种功能的支持，但是高级的反汇编程序可以在不存在任何符号的情况下尝试重新构造代码，生成以高级语言表示的对二进制程序文件的重新解释。

缺陷跟踪

一个优秀的缺陷跟踪系统提供一个共享的数据库，其中包含在你的系统中找到的 bug

① 更有社会责任感的语言（例如 Java）在其语言设计中避免了这种问题。

的跟踪记录。它使你的同事可以报告缺陷，对缺陷进行查询、分配或注释，并最终将缺陷标记为已修正。它是确保产品质量的一种关键工具——你需要对缺陷进行系统的管理，否则它们会从你的指间滑走，而你会发布存在问题的产品。在回顾项目的历史时，得到并储存这些信息也非常有用。

7.4.4 语言支持工具

为了使用高级语言来编写代码，你需要许多支持。语言的实现提供了你编写代码所需要的所有功能，这要比你在机器代码的沼泽中跋涉轻松得多。

编程语言

语言本身就是一个工具。一些语言提供了其他语言所没有的功能。一些你可以在程序的源代码上运行的单独工具也许能弥补这些差别。例如，C 语言中饱受非议的预处理程序可以变得非常有用，而对于其他语言则存在文本处理包。通配代码工具（如 C++的模板）和前置/后置条件检查是其他几种类似的语言工具，这些工具也非常有用。

在手边有一些可供选择的语言非常有价值。了解它们之间的区别，它们分别适合什么任务，以及它们的弱点是什么。然后你就可以为任何给定的任务选择最佳的语言。

关键概念　掌握几种语言，每个语言都会教给你一种解决问题的不同方式。将语言作为工具来考虑，为每项任务选择最合适的语言。

运行时和解释程序

大多数语言没有必需的运行时支持就无法使用。解释型语言依赖它们的解释程序（或虚拟机），但是直接编译的语言仍然依赖它们的支持库。这些库通常都与语言本身紧密地交织在一起，所以二者无法分开。

正如你可以选择一种不同的编译器一样，你也可以选择一种具有不同特点的语言运行时。

Java 的 JVM（Java 虚拟机）是一种常见的语言解释程序。C++标准库支持 C++语言，它为某些核心的语言功能提供了默认的处理程序。类似地，C#语言依赖.NET 环境的运行时支持。

组件和库

没错，它们也是工具！对软件组件的重用和找出可实现你所需功能的库，可以避免重复进行相同的工作。像任何其他软件工具一样，一个好的库也可以提高你的工作效率。

这些库的应用范围多种多样，一些是用于整个操作系统的庞大的抽象层，而另一些可能只完成一些简单的任务，例如提供一个功能简单的 date 类。他们照管着自己的细节，并且隐藏了复杂的具体实现，因此你不需要为其费心。你不再需要花时间来编写、测试和调试你自己的版本。

目前所有的语言都在一定程度上提供了库支持。C++ STL 就是强大的可扩展库的一个绝佳的例子。Java 语言和.NET 环境附带了多到你无法一一举出的标准库。此外还存在许许多多的第三方库，这些库既有商业性的，也有免费的。

7.4.5　其他工具

故事并没有在这里结束。你还会遇到大量的其他工具。第 143 页的“另请参见”指出了其他一些我们将讨论软件工具的地方。

下面是其他一些有趣的工具种类。

文档工具

良好的文档非常宝贵，它是设计良好的代码的关键部分。有很多种工具可以帮助你来编写文档，既包括在源代码本身中编写，也包括单独地编写（我在第 68 页的“实用的自文档化方法”中介绍了一些这样的工具）。绝不要低估优秀的文字处理软件的重要性。

文档不仅需要编写，也要被阅读。好的在线帮助系统（由高质量的电子书架支持）非常重要。

自己动手，丰衣足食

当你无法为一项任务找到工具，而手工完成它看起来又是不可能时，会发生什么？使用“自行设计的”工具没什么错。真的，如果这项任务会重复地出现，短暂的工具开发从长期来看可以为你节约大量的时间。

有些任务天生就比其他任务更适合用工具来完成。确保你所做的事切实可行，并且核算你所付出的努力是否划算。

以下是一些创建工具的常见方式。

- 以一种新的方式来组合现有的工具（通常使用 UNIX 的管道机制），也许需要编写一些连接代码。你可以将复杂的命令行“咒语”放在 Shell 脚本（或 Windows 的批处理文件）中，这样就不再需要每次都输入它们。
- 使用脚本语言。很多较小的自制工具都是以某种形式的脚本语言（通常是 Perl）编写的。脚本语言很好掌握，并且也足够强大，可以提供你编写工具所需要的支持。
- 从零开始创建完善的程序。只有在它是一种你将会不断反复地使用的重要工具时，你才需要这样做。否则，投入的精力可能就不合算。

在编写工具时，考虑以下事项：

- 使用者——工具需要有多精美？有一些未处理的毛边是否可以接受？如果只是你和另一位程序员使用它，那么这样可以对付。但是如果有一天其他新手需要使用它，也许你就应该把它包装得更精美一些。
- 你是否可以扩展一个现有的工具（整理它的命令，或者为它创建一个插件）？

项目管理

管理和工作协调工具使你可以报告和跟踪工作的进度、管理缺陷和监视团队的绩效等。根据管理工具的应用范围，低级别的程序员也许不需要使用它。但是非常特殊的系统也许会成为项目活动的中心，这样所有用户就都要参与其中。

7.5　总结

给我们适当的工具，我们就会完成任何工作。

——温斯顿·邱吉尔爵士（Sir Winston Churchill）

工具使软件开发成为可能。优秀的工具大大地简化了软件开发。

对你所使用的工具集做一个评估。你是否真的知道如何正确地使用它们？你是否缺少一些本来应该拥有的工具？你最大限度地发挥了你的工具的作用了吗？

工具只会和它的使用者一样好。俗话说“差劲的工匠往往只会抱怨他的工具不好”，这其中有许多真理。不论使用多少工具，差劲的程序员只会创造出差劲的代码。实际上，工具甚至会帮助他们编出极其糟糕的代码。对你的工具箱抱有一种职业的、负责的态度，将使你成为一名更好的程序员。

优秀的程序员……

- 愿意一次学会如何使用一种适当的工具，而不愿不断地重复执行一项单调乏味的工作
- 了解不同的工具链模型，并且用起来每个都感觉很舒服
- 使用工具以使他们的生活更简单，但是不会成为工具的奴隶
- 将他们所使用的所有东西都看作是工具，即一种可替换的用具
- 很有效率，因为他们对工具的使用是第二天性

糟糕的程序员……

- 了解某些工具的使用方法，并且从这些工具的角度来看待所有问题
- 害怕花时间来学习新的工具
- 从一开始使用某种开发环境，到现在一直近乎虔诚地使用它，从不尝试改变，甚至不尝试了解其他的开发环境
- 当遇到一个颇有价值的新工具时，并不将其添加到他们的工具箱中

7.6 另请参见

第 10 章：代码构建

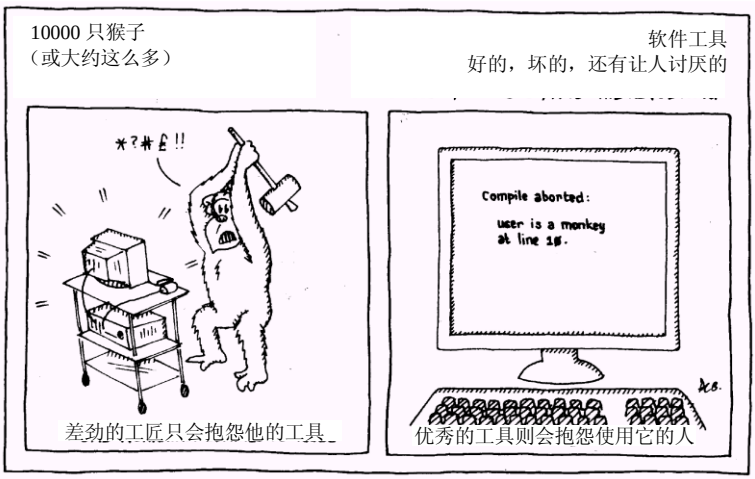
软件的构建过程是由工具来推动的。想像一下徒手编译代码的情况吧！

第 13 章：崇尚设计

本章中的一节将对具体的设计工具进行讨论。

第 18 章：安全措施

本章专门讨论版本控制工具的使用。



7.7 思考

关于这些问题的详细讨论，可以在第 491 页的“答案和讨论”部分找到。

7.7.1 深入思考

1. 是开发团队中的所有人都使用相同的 IDE 重要，还是每个人都选择一种适合他或她的 IDE 重要？如果不同的人使用不同的工具，会有什么要求？

2. 任何程序员都应该拥有的最小工具集，需要包含哪些工具？

3. 命令行工具和基于 GUI 的工具，哪个更强大？

4. 有没有不是程序的构建工具？

5. 对于一个工具来说，最重要的是什么？

   a. 互操作性

   b. 灵活性

   c. 专用化

   d. 强大

   e. 易于使用和学习

7.7.2　结合自己

1. 你的工具箱中一般有哪些工具？你每天都使用的工具有哪些？你每周使用几次的工具有哪些？你只是偶尔才会用到的工具有哪些？

   a. 你对这些工具的熟悉程度如何？

   b. 你是否最大限度地发挥了每个工具的作用？

   c. 你是如何学会使用它们的？你是否花过一些时间来提高你使用这些工具的技巧？

   d. 这些是否是你可以使用的最好的工具？

2. 你的工具是最新的吗？如果它们的版本不是最新的，会有关系吗？

3. 你是喜欢集成的工具集（如可视的开发环境），还是喜欢分立的工具链？这两种方式各有什么优点？你对这两种工作方式分别有多少经验？

4. 你是只接受默认设置的丹尼尔，还是会不停地尝试不同配置的汤姆？你是会接受你编辑器中的默认设置，还是会详尽地定制它？哪一种方式更好？

5. 你如何决定软件工具的预算？你如何知道一个工具是否物有所值？