HZ BOOKS
华章科技
PEARSON

深入解析高效软件构造系统的实现原理和运作机制，及其可伸缩性和性能优化

系统讲解实现和维护软件构造系统所需的理论、工具、流程、方法和技巧，以及各种常见的错误和陷阱，包含大量最佳实践

Software Build Systems
Principles and Experience

深入理解软件构造系统
原理与最佳实践

（加）Peter Smith 著
仲田 等译

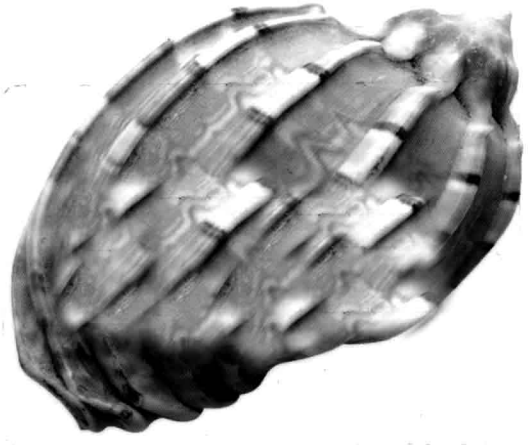
机械工业出版社
China Machine Press

华章程序员书库

Software Build Systems
Principles and Experience

深入理解软件构造系统
原理与最佳实践

（加）Peter Smith 著
仲田 等译



机械工业出版社
China Machine Press

构造系统在软件开发过程中处于核心地位，它的正确性和性能，在一定程度上决定了软件开发成果的质量和软件开发过程的效率。本书作者作为一名软件构造系统专家，总结了自己在构造系统开发和维护方面的多年经验，对软件构造系统的原理进行了深入浅出的剖析，并通过各种实际使用场景，对几种最流行的构造工具进行了对比分析，另外还讨论了构造系统的性能优化、规模提升等高级主题。

本书分为四部分。第一部分：基础知识，第1~5章分别从构造系统的高层概念、基于Make的构造系统、程序的运行时视图、文件类型与编译工具、子标的与构造变量等方面介绍构造系统的概念和相关主题。第二部分：构造工具，第6~10章结合实际场景案例，对GNU Make、Ant、SCons、CMake和Eclipse IDE这五种构造工具进行分析比较，品评优劣，帮助读者了解构造工具的当前状况，并理解每种工具的优缺点。第三部分：高级主题，第11~16章对依赖关系、元数据、软件打包与安装、构造机器、工具管理等高级主题进行讨论，帮助读者理解关于建设构造系统的许多高级主题，并了解最佳实践。第四部分：提升规模，第17~19章讨论了在大规模构造系统的环境下，如何降低复杂性，提高构造运行速度，帮助读者理解如何设计出能够适应规模增长的小型构造系统，从而对软件构造系统有更好的认识。

本书适合软件开发相关人员，包含软件开发人员、项目经理、软件构造专业人士等阅读。

本书版权登记号：图字：01-2011-2886

图书在版编目（CIP）数据

深入理解软件构造系统：原理与最佳实践/（加）史密斯（Smith，P.）著；仲田等译. —北京：机械工业出版社，2012.5

（华章专业开发者丛书）

书名原文：Software Build Systems:Principles and Experience

ISBN 978-7-111-38226-3

Ⅰ. 深… Ⅱ. ①史… ②仲… Ⅲ. 软件开发–研究 Ⅳ. TP311.52

中国版本图书馆CIP数据核字（2012）第082332号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：秦 健

北京京师印务有限公司印刷

2012年6月第 1 版第 1 次印刷

186mm×240mm • 26.75印张

标准书号：ISBN 978-7-111-38226-3

定价：89.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

对本书的赞誉

“本书对软件构造过程进行了彻底而深入的研究，其中包括在精心设计的构造过程中所做出的各种选择，相关的利弊，以及遇到的难点。我不仅要向所有的软件构造工程师推荐这本书，还要向所有的软件开发人员推荐，因为软件开发过程有效性的关键，在于具备一套精心设计的构造过程。”

——Kevin Bodie，Pitney Bowes公司软件开发主管

“在软件开发项目中，构造系统常常不受重视，而它实际上是软件开发项目的重要组成部分之一。本书对构造系统进行了精彩而详尽的解说，其中，仅对构造系统相关生产率所进行的讨论，其价值就足以抵得上为阅读本书所花的时间。”

——John M. Pantone，Objectech公司副总裁、IT教育者和课程研发人

“作者为我们探寻软件构造系统的世界提供了一幅深入浅出的蓝图，其中融汇了他多年的实践经验，涵盖了构造工程师常用的几乎所有类型的工具。本书体裁得当，文字精练，而且相当深入。我向在工作中用到构造系统的所有人推荐这本书。”

——Jeff Overbey，Photran公司项目副主管

“本书指导我们如何看待构造软件，本书涵盖构造软件产品所用的工具和技术，以及我们要避免的各种歧路迷途。对于构造系统新手和经验丰富的构造系统工程师来说，本书都具有足够的吸引力。”

——Monte Davidoff，Alluvial软件公司软件开发咨询师

译 者 序

软件构造是大多数开发人员熟知的、软件开发过程中不可缺少的一个工作环节。在这一过程中，软件构造系统借助各种工具，将软件从源代码形式，通过编译、链接等处理步骤，转换成最终可执行程序形式。一般开发人员只了解软件构造系统的基本概念和常规操作方法，很少有人深究软件构造系统背后的工作原理和技术细节。

本书作者是一名软件构造系统专家，他融汇自己在构造系统开发和维护方面的多年经验，对软件构造系统的原理进行了深入浅出的剖析，并通过各种实际使用场景，对几种最流行的构造工具进行了对比分析，另外还讨论了构造系统的性能优化、规模提升等高级主题。

本书分为四部分，共19章，涵盖以下内容：

- 第一部分：基础知识，介绍构造系统的概念和相关主题。
- 第二部分：构造工具，结合实际场景案例，对GNU Make、Ant、SCons、CMake和Eclipse IDE这五种构造工具进行分析比较，品评优劣。
- 第三部分：高级主题，对依赖关系、元数据、软件打包与安装、构造机器、工具管理等高级主题进行讨论。
- 第四部分：提升规模，讨论了在大规模构造系统的环境下，如何降低复杂性，提高构造运行速度。

译者来自软件开发行业，虽有多年从业经验，使用过多种开发语言，但对构造系统涉猎不深，平常只满足于普通操作使用，很少深究背后的原理。通过翻译本书，对构造系统也有了更全面、更深入的理解，深感构造系统虽不起眼，但却在软件开发过程中处于核心地位，它的正确性和性能，在一定程度上决定了软件开发成果的质量和软件开发过程的效率。

本书的翻译得到了机械工业出版社陈冀康编辑的大力支持，在此深表感谢。另外，张建炜、吴畏、汪燕、张贺、张鄂军也参与了本书的翻译工作，在此一并表示感谢。

由于译者水平有限，译文中如有错漏之处，恳请读者不吝批评指正。

前　言

你是软件开发人员吗？你想知道构造系统的工作原理吗？如果你正在阅读这本书，你会对上述两个问题作出肯定的回答。不过，许多软件开发人员却对自己编写的程序是如何编译出来的兴趣寥寥，大多数人只想简单地点击按钮，把源代码转变成可执行程序即可。如果他们需要修复缺陷，只会修改源代码，然后再次点击相同的按钮。他们的乐趣在于看到自己的程序实现所有预期的功能，对他们来说，构造系统只是隐身幕后的一件必要工具而已。

当源文件的规模稍微大一些，就需要某种形式的自动化构造系统。这种系统可以是个shell脚本，供你在每次源代码修改之后运行；也可以是个makefile，它了解源文件和目标文件之间的关系；或是个更复杂的构造框架，它可以扩展到成千上万个源文件的处理规模。

如果你曾在UNIX或Windows命令行环境中写过程序代码，那么以下代码看起来就很眼熟：

```
cc -o sorter main.c sort.c files.c tree.c merge.c
```

本例对5个C语言文件进行编译和链接，创建一个名为sorter的独立可执行程序。对于使用集成开发环境（Integrated Development Environment，IDE）的人来说，以上内容可能比较陌生，但它就是等同于在IDE环境中创建一个含有5个源文件的项目，然后在工具栏点击【build】按钮。

在进行过几次手工编译程序之后，你可能想把这些命令保存到一个shell脚本中，然后在每次修改代码后重新运行这个脚本。另外，你也可以从命令行历史中检索命令，然后在每次修改代码后再次执行相同的命令。

如果你基本了解Make工具，就可以创建自己的makefile，并在每次需要重新构造时输入make命令。使用Make的好处是它仅当源文件在上次编译之后发生变化时，才对程序进行重新构造。以下是用来编译sorter程序的简单makefile示例：

```
sorter: main.c sort.c files.c tree.c merge.c
        cc -o sorter main.c sort.c files.c tree.c merge.c
```

如果你熟悉Make，就会立即看出这不是个编写makefile的好方法。第一个错误是把源文件列出了两次：第一次是用于指定依赖关系，第二次是用在编译命令中。第二个错误是每次重新构造程序时，会把所有源文件都编译一次，即使它们未做任何修改。另外，这里只字未提C文件可能具有头文件依赖关系。

更好的解决办法是分解编译步骤，使每个源文件的编译和重新编译都与其他文件无关。另外应当建立依赖文件（以.d为扩展名）来跟踪头文件的使用情况。改进建议还有很多，我们与其陷入所有技术细节，不如直接来看满足全部要求的makefile最终版本：

```
SOURCES = main.c sort.c files.c tree.c merge.c

OBJECTS = $(SOURCES:.c=.o)

sorter: $(OBJECTS)
        $(CC) -o $@ $^
-include $(SOURCES:.c=.d)

%.d: %.c
        @$(CC) -MM $(CPPFLAGS) $< | sed 's#\(.*\)\.o: #\1.o
➥\1.d: #g' > $@
```

这就是全部，用最少量的makefile代码，最大限度地减少重复劳动。很简单，是不是？

不过，如果你是一名开发人员而非构造专家，你真的理解上面的例子吗？经验丰富的Make专家当然理解这些语法，而且会提出更高效的建议来实现相同的效果。但对我们这些只想有个【build】按钮的大多数人来说，第一次总要浪费大量时间才能让makefile正常奏效。

实现和维护构造系统常常是复杂的。设计得糟糕的构造系统可能会浪费大量时间，如果某个文件应当却未被重新编译的话。当规模上升到成千上万个源文件，开发人员可能会浪费半天时间来跟踪排查问题，最后发现从零开始构造（删除所有目标文件）是唯一的解决办法。看来【build】按钮涉及的工作真不少！

构造系统逐渐复杂的原因

构造系统会十分复杂并难以维护，这可能会让你感到惊讶。如今图形化用户界面是如此普及，你当然期望构造工具也会同样简单易用。遗憾的是，许多人把创建构造系统看成是一种黑色艺术。只有少数知识渊博的大师，才知道构造工具的完整语法，或依赖关系体系的具体细节。尽管基于IDE的构造工具已解决了部分问题，但它们仍无法承载大规模构造系统的复杂度。

在大多数情况下，一般软件产品的起始版本都来源于少量的源文件，这些文件被编译和链接形成程序。在这种情况下，一个简单的makefile就足以满足需要，而且这个makefile只需从用户手册的makefile模板中复制一些内容，花几个小时修改一下就可以轻松出炉。在几个月内，除了向其中增加新的源文件和库文件，没有人需要修改这个构造系统。

再过一段时间，人们开始发现这个构造过程中的问题。例如，当文件需要重新编译时却没有编译；当文件本身或其依赖的数据都没有变化时，文件却被错误地重新编译了；文件可能在一次构造过程中被编译了几遍，导致构造速度很慢。很快，“永远采用‘clean 构造’（即首先删除所有目标文件）”或“修改相关文件迫使其重新编译”等做法，就变成了人们的共识。

当简单的构造系统变得难以使用，makefile专家就需要反思构造系统的设计，这可能需要创建一个框架来解决所有的构造问题，并把实现细节与最终用户隔离开。例如，软件开发人员想看到的是源文件、库文件、文件中所用编译标志参数等的清单，但对依赖关系是如何管理的却毫无兴趣。例如：

```
SOURCES := main.c sort.c files.c tree.c merge.c

PROGRAM := sorter
```

```
LIBRARIES := libc libz

include framework.mk
```

我们的最终目标是得到一个正确而易用的构造系统，并将所有的复杂性隐藏在framework.mk文件中。对于只想有个【build】按钮的软件开发人员来说，这是个理想的解决方案。

这种框架式方法在一段时间内很有效果，效率也颇高，但在未来的某个时间就会开始问题频发，对于那些在几年内不断增长的成功软件产品来说尤其如此。这种适用于中小规模产品的构造系统，当产品规模增长时将不再适用。

考虑一下，如果要对购自第三方供应商的新代码模块进行集成，你会怎么做？新代码已有自己的构造系统，与你原来产品的构造框架并不相同。当开发人员修改代码时，就在新增代码和现有代码库之间建立了相互依赖关系，这就要求构造系统能理解更复杂的文件间关系。最终结果是单方或双方的构造框架需要进行重大修改，甚至可能是推翻重做。

随着时间的推移，框架也在不断增长，从而使框架的维护工作变得困难重重。在某些情况下，框架的原作者已经离开，无法进行修改，只有让不熟悉的生手做这项工作。缺乏足够构造经验的开发人员常常采用走捷径留后患的方法来构造软件，如后文所述，这些方法包括：写得很糟糕的shell脚本、大量使用符号链接，最糟糕的是源文件重复拷贝。构造过程变得机关重重，没人能够轻而易举地进行维护。

悲哀的是，许多组织并不觉得需要完善自己的构造系统。如果组织的管理者是其他领域（例如电脑游戏、通信、商业应用等）的专家，他们会将热情全部投注在创造产品上，增加产品的新功能，力图以之打动最终用户。在他们看来，构造系统只是产品生命周期中必要的一环，但完善构造系统并不是自己的责任。这一工作当然也从不出现在公司的企业目标或季度开发计划中。

如你所见，本书通篇讨论了在设计构造系统时必须考虑的大量问题。这不仅仅是拥有一个随叫随到的makfile大师之类的事，还应当让开发环境保持一种可维护的状态。把钱和时间花在清理构造系统上是值得的，这可以让软件开发团队的生产率提升很多倍。

构造系统的真正成本

如果你还未相信一套可靠的构造系统的重要性，那就想想真正的成本。也就是说，如果你没有一套良好的构造系统，将会付出多大代价？这种代价并不反映在财务报表上，而是隐藏在软件开发人员日常生产率之中。

一项行业调查表明（参见参考文献[1]），开发人员因为构造问题导致平均生产率降低12%，而有些被调查对象声称20%～30%也并非罕见。值得注意的是，这项调查关注的是少于20人的小型开发团队，他们一般不会碰到大型软件所遭遇的规模度问题。

我们先假定在你的团队中，所有软件开发人员在遇到构造系统问题时会损失10%的工时。你对此产生何种反应取决于你以前软件项目的经验。对有些人来说，10%可能是了不得的大问题，而对许多组织来说，10%只是入门级程度。

这10%的生产率损失在哪里？回想一下你的团队以前几乎肯定碰到过的一些典型问题：

- **错误的依赖关系导致编译失败**：构造系统获取的依赖关系信息有误，从而导致部分源代码无法正确地重新编译。当发生这种情况时，开发人员会投入所有时间，力图完成一次成功的构造。而构造错误信息不仅晦涩难解，而且与开发人员刚刚修改的代码范围完全无关。如果不解决此类构造问题，他们就无法继续有效开展工作。
- **错误的依赖关系导致无效的软件实体**：和上面的情况类似，错误的依赖关系导致部分代码编译失常。然而，构造系统并没有给出编译错误，但程序却生成无效的输出结果。这样一来，开发人员和测试人员看到的是代码中问题重重，而他们常常以为是自己的原因，而不是怀疑构造系统。开发人员要浪费一两天的时间来排查测试错误，最后才发现问题并非是自己修改代码造成的。而采用全新的构造树来开始编译，这些问题就会销声匿迹。
- **编译速度太慢**：大型软件系统更经常遇到这一问题，而小型软件可以在几分钟之内构造完成。如果你的软件代码库需要很多小时的编译时间，那么当开发人员在等待编译完成时，他们的时间就被完全浪费了。这个问题对于增量式构造来说尤其严重，只修改一个源文件，就可能导致 5 ～ 10 分钟的延迟，然后才能重新运行程序。

  你可能会想，在等待编译完成时，开发人员可以做其他工作，这样就更有效率。但情况并不总是这样。开发人员有许多种“等待”活动，例如阅读最新头条新闻、更新社交网站内容、再喝一杯咖啡，或者起身与朋友聊天。但即使开发人员可以在构造过程中进行此类多任务，也会由于在不同任务环境之间切换而带来生产率损耗。开发人员的注意力可能被分散，而完全想不起来他们刚刚在做的事情。
- **花时间更新构造描述文件**：如果软件构造框架艰深难懂，开发人员就可能需要咨询专家才能进行修改。例如，如果想把新的源文件类型或新的编译工具加入进来，他们必须先与构造专家咨询讨论。有时要等上几天，构造专家才有空提供帮助；然后，构造专家可能要花一两个星期研究才能完成这项工作。

如果你现在相信了10%的生产率损耗是个实际的数字，那么由此带来的财务成本是多少？对此进行评估的最好办法，是计算出你的公司工资总额支出的10%是多少。如果你是义务编写软件（这在开源世界中很普遍），那么这种评估方法就无效了，但其成本数额同样是所有人关注的主题。

假定你有10名软件工程师，每人每年的工资总额是7.5万美元。这个额度是偏高还是偏低因城市而异，因此最好根据你自己的实际情况进行评估。会计对此估算结果可能会提出质疑，因为还要考虑额外成本，例如员工医疗福利、电费、房租、停车费，以及开发人员享受的其他专项福利。因此，我们暂且假定每年花在每个开发人员身上的成本是15万美元。

由此得到开发人员解决构造问题的总成本是：

$$10\% \times \text{每年15万美元} \times 10\text{名开发人员} = \text{每年15万美元}$$

这相当于一名全职开发人员一年不干活的成本！假定每年的工作日是250天，那么你的公司仅仅由于构造问题，每天就要付出600美元的代价！

如果你是一位软件经理，你会想什么办法来提高收益？是继续为团队因为构造问题所浪费的时间每天支出600美元，还是付出一两个月的600美元 / 天的代价来雇佣一名构造专家，彻底解决此类问题？你自己的组织要选择何种方式，绝对值得认真考虑。请别忘了，公司的利润来自两个方面：要么通过加大产品销售而增加收入，要么通过在起始环节削减产品的生产成本。

本书的重点

你花时间阅读本书的原因有二：

- **为了理解构造系统背后的基本原理**：本书对构造系统的特性和应用场景进行了全面彻底的研究，让你可以理解构造工具的工作方式。
- **为了获得构造系统的更多经验**：本书融汇了作者运用多种不同工具在创建和维护构造系统方面的多年经验，通过阅读本书，你就可以避免以前构造系统开发人员所犯的初级错误。

有了以上知识，你就可以做出明智的选择，采用何种构造工具，如何建设可靠的构造系统，以及如何预见可能影响生产率的陷阱和错误。从此，软件的构造过程将变得更快、更容易和更可靠。

还要提醒一下，本书不包括以下内容：

- **本书不是操作手册**：本书不提供任何特定构造工具或技术的操作手册内容，仅介绍少量示例片断（例如第 2 章）。流行的构造工具都有各自专门的网站和书籍，为你讲解每项语法和语义的细节。请参阅这些书籍，了解每种构造工具的更详细信息。
- **本书不展示构造系统的全套功能**：尽管本书包含许多示例，展示如何使用每种构造工具及很多支持工具，但并不从头至尾演示如何创建一个全功能的构造系统。同上，你应当参阅每种构造工具的文档，来查阅完整的创建示例。

当然，首先应该阅读本书，这样你就能理解每种构造工具的优缺点，从而可以判断出你自己的构造系统应该使用哪些特性。

本书并未限定于某种开发环境或编程语言，而是从多种不同角度来提供示例并介绍概念：

- **C/C++ 语言构造**：这可能是最经典的构造类型了，其构造风格起源于 20 世纪 70 年代，到现在一直变化不大。近年来它面临的挑战是文件数量的增长，以及如何处理已用于典型软件产品的第三方程序库。
- **Java 语言构造**：Java 语言在 20 世纪 90 年代后期逐渐流行，对构造系统的设计产生了相当深远的影响。举个例子，Java 的源文件所保存的目录层次结构必须要与软件包的结构相匹配。
- **C# 语言构造**：C 语言、C++ 和 Java 都是平台无关的编程语言，因此可以用于任何操作系统（例如 Linux、Solaris、Mac OS X 和 Windows），而 C# 语言的构造环境则更倾向于微软公司的做事方式。

除了覆盖多种编程语言，本书还讨论了开发大型软件产品的两种不同方式：

- **单体构造**：此种方式是在一次构造过程中，整个代码库是从源代码编译成一个可执行程

序。这是小型程序的常用构造方式，但它的可伸缩性不好，因为采用此种方式来构造大型软件，会产生巨大的源树，消耗极长的编译时间。

- **组件构造**：与单体构造相反，本方式把源代码划分为多个层级，分别单独编译，最后把各个预构造的组件合并成最终的可执行程序。

最后提醒一下，本书讲述的内容并不遵循“Make是C/C++开发所选择的主要工具”、“所有Java和C#软件应当在IDE中进行构造”等常规假定。

预期读者

本书主要面向软件相关开发人员，包含以下几类读者：

- **开发人员**：如果你是一名有多年源代码开发经验但只有少量构造系统经验的开发人员，可以通过本书学习如何建设和维护构造系统，还可以学习各种用来描述构造过程的工具。
- 项目经理：通过本书，你可以学会较高层次的概念和技巧，而无须深入了解太多复杂细节。运用这些知识，你就可以评估团队所做的工作，并向他们提出恰当的“指引方向式”问题。
- **构造专家**：即使你有建设构造系统的多年经验，也可以从本书学到一些新东西。本书不仅展示了你可能从未用过的现代构造工具，而且关于大型构造系统可伸缩性和性能方面的讨论，会让你在以后编写构造框架时三思而后行。

本书的组织结构

本书主要分为四个部分，每个部分讨论构造系统的角度略有差异。根据自身经验和所关心的细节层次的不同，你可以选择重点阅读本书的某几部分。新手开发人员应当重点阅读第一部分和第二部分，有经验的读者可以跳过第一部分，重点阅读第二、三、四部分。

第一部分：基础知识

第一部分面向没有太多构造经验的软件开发人员，对构造系统进行简要介绍。即使有经验的读者也应快速浏览这些章节，以确保对基本概念有个完整的了解。例如，C/C++开发人员可以学到关于C#语言的新知识。

第1章介绍了构造系统的高层次概念，例如源树和目标树、构造工具、编译工具等。第2章为从未接触过Make的读者提供了如何编写makefile的快速指南。第3章描述了程序在计算机中运行的结构，以便讲解构造系统要生成哪些东西。第4章对构造过程使用的各种输入输出文件类型进行了详细讲解，并用C/C++、Java和C#语言进行举例。第5章描述了构造变量背后的基本思想，而构造变量本身则在后续章节有更详细的介绍。

阅读完第一部分，你会对构造系统设计的相关基本概念有充分理解。

第二部分：构造工具

本书第二部分对5种构造工具进行了研究比较。在现有各种工具中筛选这5种工具时，既要考虑到它们的流行程度，也要考虑到它们各自体现了软件构造的特定方法。每个章节首先介绍构造工具的语法，然后描述该工具的主要应用场景。为了提供有意义的比较结果，所有章节都使用了同一套标准示例。

第6章讨论了GNU Make工具，它是C/C++开发的最常用工具。第7章研究了Ant工具，它是Java编译的事实标准。第8章研究了新近出现的SCons工具，它采用Python语言来描述构造过程。第9章展示了CMake工具，它根据构造过程的高层次描述，生成一套原生构造系统（例如基于Make的系统）。第10章描述了Eclipse IDE中与构造相关的特性。

阅读完第二部分，你会了解构造工具的当前状况，并理解每种工具的优缺点。

第三部分：高级主题

第三部分讨论更高级的构造系统概念，例如依赖关系分析、软件打包与安装、版本管理，以及构造机器和编译工具的管理。这些章节假定你有较复杂软件项目的工作经验，可以理解这里讨论的内容。

第11章详细探讨了各种依赖关系检查技术，用于判断哪些文件是否必须重新编译。第12章展示了构造系统如何生成元数据来帮助进行调试、性能分析和源代码文档编写。第13章提供了相关示例，展示如何进行软件打包，使之达到可安装到目标机器的状态。第14章研究了与构造系统有关的版本控制方面的问题。第15章提供了关于构造机器管理的最佳经验做法（构造机器是指软件在其上进行编译的机器）。第16章对编译工具进行了类似讨论。

阅读完第三部分，你会理解关于建设构造系统的许多高级主题，并了解部分最佳经验做法。

第四部分：提升规模

本书的最后一部分讨论了在构造系统设计时如何适应大型软件产品。随着软件产品的规模不断增长，它会面临可伸缩性问题，例如复杂度增长、磁盘空间占用急剧增多、构造时间也变长了。所有这些问题都会降低软件开发的生产率。

第17章提出了降低最终用户能感觉到的构造系统复杂度的几种方法。第18章描述了怎样把大型软件产品划分为多个组件，让开发更有效率。最后，第19章讨论了对软件构造时间进行度量和改进的技术方法。

阅读完第四部分，关于如何设计能够适应规模增长的小型构造系统，你就会有更好的认识。

小结

建设高质量构造系统不是件容易的事，但如果做不到的话，将导致软件团队产生重大问

题。如果源代码应当重新编译却没有进行，你的团队成员将会面临更长的构造时间或无规律的构造失败，他们还可能把时间浪费在对无效软件实体进行调试排错上。因此，投入一定时间来确保构造系统正常工作是非常值得的。

使用质量低劣的构造系统的真正成本，可以用金钱来衡量。对于一般的软件组织，他们会发现开发人员把10%的时间浪费在解决构造问题上，这也能推算出每年浪费多少钱。

本书讲解了很多构造系统概念，介绍了一系列常用的构造工具，提供了部分最佳实践做法，并讨论了关于建设与维护大型构造系统方面的问题。

致谢

如果没有我妻子Grace的鼎力支持，本书不可能完成。许多个夜晚和周末，我都躲在我的“安乐窝”里，敲着键盘，神游物外。Grace理解我写这本书的重要意义（它列在我的遗愿清单上），她的耐心和支持让本书得以面世。还要感谢Stan（我们的马尔济斯比熊犬），它终于体会到坐在地板上通常比坐在我的笔记本电脑或键盘上更舒服。

感谢我的父母，Sally和Smithy，允许我在他们餐厅的桌上编写几个章节的内容。我还要感谢他们多年来纠正我的拼写和语法错误，让我更轻松地写出本书这么多的内容。

我要感谢接纳本书予以出版的Pearson Education团队提供的支持，感谢Raina Chrobak、Chris Zahn和Chris Guzikowski在本书的写作和编辑过程中给予的指导，也感谢初稿评审人从实践者或构造系统专家的角度提出的反馈意见，他们是Monte Davidoff、Jeffrey Overbey、J. T. Conklin、Kevin Bodie、Brad Appleton、John Pantone和Usman Muzaffar。

然后，我还要感谢Kevin Cheek和Bob McLaren，以及爱立信公司团队中的其他人，他们允许我重新协商我的长期合同，让我有足够的时间来写书。还要感谢许多朋友和同事，他们贡献出自己全部的构造系统经验，我希望他们的每条经验都已体现在本书中。

最后，必须感谢为构造工具的设计和制造作出贡献的每个人。大多数软件项目都使用某种构造工具，这让构造系统成为软件技术的关键环节之一。而创造这些工具的人们，并不总是得到应有的尊重。

目　录