

《深入理解软件构造系统》

图书基本信息

书名：《深入理解软件构造系统》

13位ISBN编号：9787111382263

10位ISBN编号：7111382269

出版时间：2012-6-15

出版社：机械工业出版社华章公司

作者：Peter Smith

页数：406

译者：仲田

版权说明：本站所提供下载的PDF图书仅提供预览和简介以及在线试读，请支持正版图书。

更多资源请访问：www.tushu111.com

《深入理解软件构造系统》

内容概要

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。

《深入理解软件构造系统》

作者简介

Peter

Smith，资深软件开发工程师和软件构造系统专家，专注于软件生产效率的探索和研究，对各种新型软件工具的选用与开发、软件项目管理、IT基础设施项目管理、基于软件工具的流程改进，以及如何使企业的现有流程实现自动化等能帮助企业提高软件生产效率的一系列核心问题都有非常深入的认识，实践经验极为丰富。

Peter毕业于哥伦比亚大学，拥有计算机科学博士学位，研究方向是编译器和语言设计。他曾在大学任教，主要教授编译器设计、编程语言设计、软件工程和计算机网络等方面的课程。此外，他还是OOPSLA（面向对象编程、系统、语言与应用）协会的委员。

书籍目录

对本书的赞誉

译者序

前言

致谢

作者介绍

第一部分 基础知识

第1章 构造系统概述2

1.1 什么是构造系统2

1.1.1 编译型语言3

1.1.2 解释型语言3

1.1.3 Web应用4

1.1.4 单元测试5

1.1.5 静态分析5

1.1.6 文档生成6

1.2 构造系统的各个组成部分6

1.2.1 版本控制工具7

1.2.2 源树与目标树7

1.2.3 编译工具和构造工具8

1.2.4 构造机器9

1.2.5 发布打包与目标机器9

1.3 构造过程和构造描述11

1.4 如何使用构造系统12

构造管理工具12

1.5 构造系统的质量13

本章小结14

第2章 基于Make的构造系统15

2.1 Calculator示例15

2.2 创建一个简单的makefile17

2.3 对这个makefile进行简化19

2.4 额外的构造任务20

2.5 框架的运用21

本章小结23

第3章 程序的运行时视图24

3.1 可执行程序24

3.1.1 原生机器码25

3.1.2 单体系统镜像25

3.1.3 程序完全解释执行26

3.1.4 解释型字节码26

3.2 程序库28

3.2.1 静态链接28

3.2.2 动态链接29

3.3 配置文件和数据文件30

3.4 分布式程序30

本章小结31

第4章 文件类型与编译工具33

4.1 C/C++34

4.1.1 编译工具34

- 4.1.2 源文件35
- 4.1.3 汇编语言文件37
- 4.1.4 目标文件38
- 4.1.5 可执行程序40
- 4.1.6 静态程序库40
- 4.1.7 动态程序库41
- 4.1.8 C++编译42
- 4.2 Java43
 - 4.2.1 编译工具43
 - 4.2.2 源文件44
 - 4.2.3 目标文件45
 - 4.2.4 可执行程序47
 - 4.2.5 程序库48
- 4.3 C#48
 - 4.3.1 编译工具49
 - 4.3.2 源文件49
 - 4.3.3 可执行程序51
 - 4.3.4 程序库53
- 4.4 其他文件类型55
 - 4.4.1 基于UML的代码生成56
 - 4.4.2 图形图像57
 - 4.4.3 XML配置文件58
 - 4.4.4 国际化与资源绑定58
- 本章小结59
- 第5章 子标的与构造变量60
 - 5.1 针对子标的进行构造61
 - 5.2 针对软件的不同版本进行构造62
 - 5.2.1 指定构造变量63
 - 5.2.2 对代码的定制调整65
 - 5.3 针对不同的目标系统架构进行构造68
 - 5.3.1 多重编译器68
 - 5.3.2 面向指定平台的文件 / 功能69
 - 5.3.3 多个目标树69
- 本章小结71
- 第二部分 构造工具
- 现实场景75
 - 场景1：源代码放在单个目录中75
 - 场景2：源代码放在多个目录中76
 - 场景3：定义新的编译工具76
 - 场景4：针对多个变量进行构造77
 - 场景5：清除构造树77
 - 场景6：对不正确的构造结果进行调试78
- 第6章 Make79
 - 6.1 GNU Make编程语言80
 - 6.1.1 makefile规则：用来建立依赖关系图80
 - 6.1.2 makefile规则的类型81
 - 6.1.3 makefile变量82
 - 6.1.4 内置变量和规则84
 - 6.1.5 数据结构与函数85

- 6.1.6 理解程序流程87
- 6.1.7 进一步阅读资料90
- 6.2 现实世界的构造系统场景90
 - 6.2.1 场景1：源代码放在单个目录中90
 - 6.2.2 场景2(a)：源代码放在多个目录中92
 - 6.2.3 场景2(b)：对多个目录进行迭代式Make操作93
 - 6.2.4 场景2(c)：对多个目录进行包含式Make操作96
 - 6.2.5 场景3：定义新的编译工具101
 - 6.2.6 场景4：针对多个变量进行构造102
 - 6.2.7 场景5：清除构造树104
 - 6.2.8 场景6：对不正确的构造结果进行调试105
- 6.3 赞扬与批评107
 - 6.3.1 赞扬107
 - 6.3.2 批评108
 - 6.3.3 评价109
- 6.4 其他类似工具110
 - 6.4.1 Berkeley Make110
 - 6.4.2 NMake111
 - 6.4.3 ElectricAccelerator和Spark Build111
- 本章小结113
- 第7章 Ant115
 - 7.1 Ant编程语言116
 - 7.1.1 比“Hello World”稍多一些116
 - 7.1.2 标的的定义和使用118
 - 7.1.3 Ant的控制流119
 - 7.1.4 属性的定义120
 - 7.1.5 内置的和可选的任务122
 - 7.1.6 选择多个文件和目录125
 - 7.1.7 条件126
 - 7.1.8 扩展Ant语言127
 - 7.1.9 进一步阅读资料128
 - 7.2 现实世界的构造系统场景129
 - 7.2.1 场景1：源代码放在单个目录中129
 - 7.2.2 场景2(a)：源代码放在多个目录中130
 - 7.2.3 场景2(b)：多个目录，多个build.xml文件130
 - 7.2.4 场景3：定义新的编译工具133
 - 7.2.5 场景4：针对多个变量进行构造136
 - 7.2.6 场景5：清除构造树140
 - 7.2.7 场景6：对不正确的构造结果进行调试141
 - 7.3 赞扬与批评142
 - 7.3.1 赞扬143
 - 7.3.2 批评143
 - 7.3.3 评价144
 - 7.4 其他类似工具144
 - 7.4.1 NAnt144
 - 7.4.2 MS Build145
- 本章小结146
- 第8章 SCons147
 - 8.1 SCons编程语言148

- 8.1.1 Python编程语言148
- 8.1.2 简单编译151
- 8.1.3 管理构造环境154
- 8.1.4 程序流程和依赖关系分析157
- 8.1.5 决定何时重新编译158
- 8.1.6 扩展该语言160
- 8.1.7 其他有趣的特性162
- 8.1.8 进一步阅读资料163
- 8.2 现实世界的构造系统场景163
 - 8.2.1 场景1：源代码放在单个目录中163
 - 8.2.2 场景2(a)：源代码放在多个目录中163
 - 8.2.3 场景2(b)：多个SConstruct文件164
 - 8.2.4 场景3：定义新的编译工具165
 - 8.2.5 场景4：针对多个变量进行构造167
 - 8.2.6 场景5：清除构造树168
 - 8.2.7 场景6：对不正确的构造结果进行调试169
- 8.3 赞扬与批评171
 - 8.3.1 赞扬171
 - 8.3.2 批评172
 - 8.3.3 评价173
- 8.4 其他类似工具173
 - 8.4.1 Cons173
 - 8.4.2 Rake174
- 本章小结176
- 第9章 CMake177
 - 9.1 CMake编程语言178
 - 9.1.1 CMake语言基础178
 - 9.1.2 构造可执行程序 and 程序库179
 - 9.1.3 控制流182
 - 9.1.4 跨平台支持184
 - 9.1.5 生成原生构造系统185
 - 9.1.6 其他有趣的特性以及进一步阅读资料190
 - 9.2 现实世界的构造系统场景191
 - 9.2.1 场景1：源代码放在单个目录中191
 - 9.2.2 场景2：源代码放在多个目录中191
 - 9.2.3 场景3：定义新的编译工具192
 - 9.2.4 场景4：针对多个变量进行构造193
 - 9.2.5 场景5：清除构造树194
 - 9.2.6 场景6：对不正确的构造结果进行调试194
 - 9.3 赞扬与批评195
 - 9.3.1 赞扬195
 - 9.3.2 批评195
 - 9.3.3 评价196
 - 9.4 其他类似工具196
 - 9.4.1 Automake196
 - 9.4.2 Qmake197
 - 本章小结197
- 第10章 Eclipse199
 - 10.1 Eclipse的概念和GUI199

- 10.1.1 创建项目200
- 10.1.2 构造项目206
- 10.1.3 运行项目210
- 10.1.4 使用内部项目模型212
- 10.1.5 其他构造特性213
- 10.1.6 进一步阅读资料214
- 10.2 现实世界的构造系统场景215
 - 10.2.1 场景1：源代码放在单个目录中215
 - 10.2.2 场景2：源代码放在多个目录中216
 - 10.2.3 场景3：定义新的编译工具217
 - 10.2.4 场景4：针对多个变量进行构造217
 - 10.2.5 场景5：清除构造树220
 - 10.2.6 场景6：对不正确的构造结果进行调试220
- 10.3 赞扬与批评221
 - 10.3.1 赞扬221
 - 10.3.2 批评221
 - 10.3.3 评价222
- 10.4 其他类似工具222
- 本章小结224
- 第三部分 高级主题
- 第11章 依赖关系226
 - 11.1 依赖关系图227
 - 11.1.1 增量式编译228
 - 11.1.2 完全、增量式和子标的构造228
 - 11.2 依赖关系错误导致的问题229
 - 11.2.1 问题：依赖关系缺失导致运行时错误229
 - 11.2.2 问题：依赖关系缺失导致编译错误230
 - 11.2.3 问题：多余的依赖关系导致大量不必要的重新构造231
 - 11.2.4 问题：多余的依赖关系导致依赖关系分析失败231
 - 11.2.5 问题：循环依赖关系232
 - 11.2.6 问题：以隐式队列顺序替代依赖关系232
 - 11.2.7 问题：Clean标的什么也清除不了233
 - 11.3 步骤一：计算依赖关系图233
 - 11.3.1 获取确切的依赖关系234
 - 11.3.2 把依赖关系图缓存起来236
 - 11.3.3 对缓存的依赖关系图进行更新237
 - 11.4 步骤二：判断哪些文件已过期239
 - 11.4.1 基于时间戳的方法240
 - 11.4.2 基于校验和的方法241
 - 11.4.3 标志参数比较242
 - 11.4.4 其他高级方法243
 - 11.5 步骤三：为编译步骤排定队列顺序243
- 本章小结246
- 第12章 运用元数据进行构造247
 - 12.1 调试支持247
 - 12.2 性能分析支持249
 - 12.3 代码覆盖分析支持250
 - 12.4 源代码文档化251
 - 12.5 单元测试253

- 12.6 静态分析256
- 12.7 向构造系统加入元数据257
- 本章小结258
- 第13章 软件打包与安装259
 - 13.1 归档文件260
 - 13.1.1 用于打包的脚本260
 - 13.1.2 其他归档文件格式262
 - 13.1.3 对打包脚本的改进263
 - 13.2 包管理工具265
 - 13.2.1 RPM包管理工具格式265
 - 13.2.2 rpm build过程266
 - 13.2.3 RPM规格文件示例267
 - 13.2.4 根据规格文件创建RPM文件272
 - 13.2.5 安装RPM示例274
 - 13.3 定制式GUI安装工具275
 - 13.3.1 Nullsoft Scriptable Install System (NSIS) 276
 - 13.3.2 安装工具脚本277
 - 13.3.3 定义安装页面280
 - 13.3.4 许可授权页面281
 - 13.3.5 选择安装目录282
 - 13.3.6 主要组件282
 - 13.3.7 可选组件283
 - 13.3.8 定制页面285
 - 13.3.9 安装页面和卸载程序286
- 本章小结288
- 第14章 版本管理289
 - 14.1 对哪些东西进行版本控制290
 - 14.1.1 构造描述文件290
 - 14.1.2 对工具的引用292
 - 14.1.3 大型二进制文件296
 - 14.1.4 对源树的配置296
 - 14.2 哪些东西不应当放到源树中297
 - 14.2.1 生成的文件被保存到源树中297
 - 14.2.2 生成的文件被纳入到版本控制中299
 - 14.2.3 构造管理脚本299
 - 14.3 版本编号300
 - 14.3.1 版本编号体系300
 - 14.3.2 协调并更新版本号301
 - 14.3.3 版本号的保存与检索302
- 本章小结303
- 第15章 构造机器305
 - 15.1 原生编译与跨平台编译306
 - 15.1.1 原生编译306
 - 15.1.2 跨平台编译306
 - 15.1.3 异构环境307
 - 15.2 集中式开发环境307
 - 15.2.1 构造机器为何有差异308
 - 15.2.2 管理多个构造机器310
 - 15.3 开源开发环境312

- 15.4 GNU Autoconf315
 - 15.4.1 高层次工作流315
 - 15.4.2 Autoconf示例317
 - 15.4.3 运行autoheader和autoconf319
 - 15.4.4 在构造机器上运行configure脚本320
 - 15.4.5 使用配置信息322
- 本章小结323
- 第16章 工具管理324
 - 16.1 工具管理的规则324
 - 16.1.1 规则1：做笔记324
 - 16.1.2 规则2：对源代码进行版本控制325
 - 16.1.3 规则3：定期升级工具326
 - 16.1.4 规则4：对工具的二进制文件进行版本控制327
 - 16.1.5 对规则的破坏329
 - 16.2 编写自己的编译工具329
- 用Lex和Yacc编写定制工具330
- 本章小结332
- 第四部分 提升规模
- 第17章 降低最终用户面对的复杂性334
 - 17.1 构造框架334
 - 17.1.1 面向开发人员的构造描述335
 - 17.1.2 面向框架的构造描述336
 - 17.1.3 惯例优先于配置336
 - 17.1.4 构造工具示例：Maven337
 - 17.2 避免支持多个构造变量的原因338
 - 17.2.1 需要测试更多的构造变量338
 - 17.2.2 代码会变得混乱339
 - 17.2.3 构造时间会增多340
 - 17.2.4 需要更多磁盘空间340
 - 17.3 降低复杂性的各种技术方法340
 - 17.3.1 使用现代构造工具340
 - 17.3.2 自动检测依赖关系341
 - 17.3.3 把生成的文件放在源树之外341
 - 17.3.4 确保正确清除构造树341
 - 17.3.5 碰到第一个错误即中止构造342
 - 17.3.6 提供有意义的错误信息343
 - 17.3.7 校验输入参数343
 - 17.3.8 不要把构造脚本搞得过分复杂344
 - 17.3.9 避免使用晦涩的语言特性344
 - 17.3.10 不要用环境变量控制构造过程345
 - 17.3.11 确保构造形成的发布版与调试版保持相似345
 - 17.3.12 准确显示正在执行的命令346
 - 17.3.13 把对工具的引用纳入版本控制347
 - 17.3.14 把构造指令纳入版本控制347
 - 17.3.15 自动检测编译标志参数的变化347
 - 17.3.16 不要在构造系统中调用版本控制工具347
 - 17.3.17 尽量频繁地进行持续集成348
 - 17.3.18 统一使用一种构造机器348
 - 17.3.19 统一使用一种编译器348

- 17.3.20 避免遗留#ifdefs的垃圾代码348
- 17.3.21 使用有意义的符号名349
- 17.3.22 删除垃圾代码349
- 17.3.23 不要复制源文件350
- 17.3.24 使用统一的构造系统350
- 17.4 对构造系统进行规划充分、人力充足的改进351
- 本章小结352
- 第18章 管理构造规模353
 - 18.1 单体模型构造存在的问题354
 - 18.2 组件式软件355
 - 18.2.1 使用组件的好处357
 - 18.2.2 组件到底是什么358
 - 18.2.3 把多个组件集成到单个产品中361
 - 18.3 人员和过程管理364
 - 18.3.1 开发团队的结构365
 - 18.3.2 组件版本队列管理367
 - 18.3.3 管理组件缓存368
 - 18.3.4 协调软件新特性的开发370
 - 18.4 Apache Ivy372
- 本章小结373
- 第19章 更快的构造375
 - 19.1 度量构造系统性能375
 - 19.1.1 启动阶段的性能度量375
 - 19.1.2 编译阶段的性能度量382
 - 19.1.3 性能度量工具386
 - 19.1.4 修正问题：改进性能388
 - 19.2 构造减免：消除不必要的重新构造389
 - 19.2.1 目标文件缓存389
 - 19.2.2 智能依赖关系391
 - 19.2.3 构造减免的其他技术方法395
 - 19.3 并行396
 - 19.3.1 构造集群 / 云396
 - 19.3.2 并行构造工具397
 - 19.3.3 对可伸缩性的限制398
 - 19.4 减少磁盘使用398
- 本章小结400
- 参考文献401

版权页：插图：14.2.2生成的文件被纳入到版本控制中如果目标文件或自动生成的源文件已经被提交到版本控制系统，这很可能是个错误。开发人员可能因为没看清楚，误将生成的文件当做源代码提交到版本控制系统中。如上文所述，如果生成的文件被不正确地与源代码保存在同一目录，开发人员就很容易犯这种错误。把生成的文件检入到版本控制系统，产生的一个副作用是：当第一个开发人员将这些文件提交后，所有其他开发人员都可能错误地提交这些文件。由于生成的文件是由构造系统自动写入的，当有人执行构造时，这些问题文件总会被修改。版本控制系统会注意到文件已修改，并准备再次将它提交到版本控制仓库中。如果开发人员不够细心，他们就会一次又一次地提交这些文件。为了尽快发现这一问题，可以考虑以只读模式检出文件（有些版本控制工具默认要求采用此模式）。当重新构造源树时，构造系统就无法写入这些生成的文件，从而导致构造失败。开发人员就会明白，这些文件是被误提交的，因此他们可以先从版本控制系统中删除这些文件，再继续进行构造。在某些情况下，把生成的文件提交到版本控制系统确实有意义，尽管这种想法一般是有问题的。例如，可以对构造树中不常变化的部分（例如第三程序库）进行预编译，从而加速构造过程。通过预编译程序库并将结果提交到版本控制系统，就可以避免让每个开发人员都自行编译这些程序库。构造程序库必须使用特殊构造标的，从而使构造系统不会默认重新创建程序库，因此不会被标为“已修改”（进而不会被再次提交到版本控制系统）。某些版本控制工具对这种想法作了进一步扩展，它们可以自动缓存生成的文件，以节省开发人员的时间，不必重新构造这些文件。第19章将详细讨论这种机制。14.2.3构造管理脚本不应当纳入版本控制系统的最后一种场景，是当脚本或工具相对于产品源代码，更依赖于外部环境（例如构造机器或磁盘）时。把这种性质的工具提交到版本控制系统，会增加修正工具缺陷的维护工作量。例如，某脚本建议开发人员，当前何种构造机器是最快的，或何种文件系统有最大磁盘空间，那么该脚本不应提交到版本控制系统。软件的版本1和版本2没必要分别使用该脚本的不同版本。事实上，当修正该脚本的缺陷时，可能导致源代码的每个分支版本都必须修改。这当然是我们不希望看到的。因此，该脚本不应当纳入版本控制，而是应当保存为常规磁盘文件，例如/TOOLS/BIN/DISK-ADVISOR。对该脚本的任何修改（例如增加新的构造机器信息或新磁盘信息），可以单独进行。这同一个脚本可以用于所有代码分支，每个分支不需要分别使用不同的脚本。而且，该脚本只关注自身所处的当前构造环境，并不关心过去怎样。如果该脚本的某些行为依赖于代码的特定分支版本，那么还可以把该脚本所需的配置信息保存在版本控制系统，但把脚本主体仍放在/TOOLS/BIN目录。例如，如果DISK-ADVISOR脚本需要知道构造过程创建了哪些输出目录，则可以创建一个配置文件来列出这些信息。

《深入理解软件构造系统》

媒体关注与评论

《深入理解软件构造系统:原理与最佳实践》深入彻底地解析了软件的构造过程，包括软件构造过程中需要做出的各种选择、可能遇到的各种困难，以及原理与最佳实践。我不仅要向所有的软件构造工程师推荐这《深入理解软件构造系统:原理与最佳实践》，还要向所有的软件开发人员推荐，因为软件开发过程中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公司软件开发总裁

《深入理解软件构造系统:原理与最佳实践》向我们清晰地展示了软件构造的原理与细节，内容涵盖构造软件产品需要用到的所有工具和技术，以及要避免的各种错误。无论是构造系统新手，还是经验丰富的构造系统工程师，《深入理解软件构造系统:原理与最佳实践》对他们来说都具有足够的吸引力。——Monte Davidoff Alluvial软件公司软件开发咨询师

《深入理解软件构造系统》

编辑推荐

《深入理解软件构造系统:原理与最佳实践》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。《深入理解软件构造系统:原理与最佳实践》深入解析了高效构造系统背后的核心原理，并调查研究了系统的特性和使用场景。《深入理解软件构造系统:原理与最佳实践》是作者多年创建并维护各种构造系统的经验结晶，能帮助我们在选择工具和方法时做出依据充分的决策，并避开常见的陷阱和错误。《深入理解软件构造系统:原理与最佳实践》还提供了丰富的实用示例，以及在JAVA、C、C++和C#等多种环境中总结的经验教训。《深入理解软件构造系统:原理与最佳实践》主要内容：系统讲解构造系统的基础知识，包括源树、构造工具以及编译工具。比较5种领先的构造工具：GNU Make、Ant、SCons、CMake和Eclipse IDE的集成构造特性。确保准确进行依赖关系检查，高效进行增量式构造。使用元数据帮助进行调试、性能分析和为源代码编制文档。打包软件，以备在目标机器上安装。包含管理复杂的版本控制系统、构造器和编译工具的最佳实践。功能欠缺的构造系统可能会对开发人员的生产效率产生巨大的影响。错误的依赖关系、中断的编译错误、失效的软件实体、缓慢的编译速度，以及费时费力的手工处理，这些都是被人诟病的构造系统存在的问题。在本书中，软件生产效率专家Peter Smith向我们展示了如何实现能够解决以上所有问题的构造系统，使我们以更快的速度和更低的成本交付可靠的软件产品。如果你是一名开发人员，本书将向你展示在构造系统的建设和维护过程中涉及的各种问题，使之最符合团队的需要；如果你是一名管理人员，你会学习如何对团队的构造系统进行评估和效能改进；如果你是一名软件构造专家，无论面临多么严苛的要求，通过学习本书，你都能很好地优化构造系统的性能和可伸缩性。

《深入理解软件构造系统》

精彩短评

- 1、全面而理论的讲述了软件构造系统(build system)中的理论与常用的构建工具， Cpp 中的CMake,基础的Makefile等。
- 2、前端同学如果了解build tool的水有多深可以看看这本，虽然由于作者的关系，广度还是有限...
- 3、介绍了软件系统设计过程中所需要用到的一系列工具
- 4、仔细读了一下，内容没有超出我理解的build工具，空洞无物
- 5、书还可以，但是有错误
- 6、很常用但又想不到会去专门研究的方面
- 7、花几个小时翻了一下，感觉很泛泛。很全很浅
- 8、贵！
- 9、以前很少关注底层的东东，同事推荐买的，粗略看了一遍，很有收获。基础还是要打牢靠了，才能往上盖高楼
- 10、开发一个烂的软件构造也许成本很低，但其维护的成本比开发的费用高的多。软件系统的构造很重要！

《深入理解软件构造系统》

版权说明

本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问:www.tushu111.com