

# Component Development for the Java Platform



# Java 平台组件开发

(美) Stuart Dabbs Hallaway 著  
韩宏志 译



清华大学出版社

# Java 平台组件开发

(美) Stuart Dabbs Halloway 著  
韩宏志 译

清华大学出版社

北 京

Simplified Chinese edition copyright © 2004 by **PEARSON EDUCATION ASIA LIMITED** and **TSINGHUA UNIVERSITY PRESS**.

Original English language title from Proprietor's edition of the Work.

Original English language title: **Component Development for the Java Platform**, by **Stuart Dabbs Halloway**, Copyright ©2002

EISBN: 0-201-75306-5

All Rights Reserved.

Published by arrangement with the original publisher, **Pearson Education, Inc.**, publishing as **Addison-Wesley**.

This edition is authorized for sale only in the **People's Republic of China** (excluding the **Special Administrative Region of Hong Kong and Macao**).

本书中文简体翻译版由培生教育出版集团授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字: 01-2004-3043

版权所有, 翻版必究。举报电话: 010-62782989 13901104297 13801310933  
本书封面贴有 **Pearson Education** (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

图书在版编目(CIP)数据

Java 平台组件开发/(美)赫莱畏(Halloway, S.D.)著; 韩宏志译. —北京: 清华大学出版社, 2004.9

书名原文: **Component Development for the Java Platform**

ISBN 7-302-08934-5

I. J… II. ①赫…②韩… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2004)第 061897 号

出版者: 清华大学出版社 地址: 北京清华大学学研大厦  
<http://www.tup.com.cn> 邮编: 100084  
社总机: 010-62770175 客户服务: 010-62776969

组稿编辑: 曹 康

文稿编辑: 王 黎

封面设计: 康 博

版式设计: 康 博

印刷者: 北京市通州大中印刷厂

装订者: 三河市李旗庄少明装订厂

发行者: 新华书店总店北京发行所

开 本: 185×260 印张: 15.75 字数: 327 千字

版 次: 2004 年 9 月第 1 版 2004 年 9 月第 1 次印刷

书 号: ISBN 7-302-08934-5/TP·6320

印 数: 1~4000 册

定 价: 32.00 元

---

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题, 请与清华大学出版社出版部联系调换。联系电话: (010)62770175-3103 或 (010)62795704

# 序

几年前，Stuart 离开了曾令自己心仪的 COM 领域，开始研究 Java。许多同事表示不解，但 Stuart 坚持己见，不为所动。当时，Stuart 的老板在 COM 领域大量投资，在 Java 领域却收效甚微。Stuart 在这样的背景下改弦易辙，实在难能可贵。

我在阅读本书后，觉得受益匪浅，不由得赞叹 Stuart 的选择。本书的出版，无论对 Stuart 本人，还是广大读者，都是一件幸事。

Stuart 对 Java 平台的理解令人耳目一新。本书将 Java 虚拟机(JVM)描述为组件化软件的基础，它并没有重点描述生成组件的语言和编译器，也未详细叙述 JVM 执行的各种服务，如无用单元收集和 JIT 编译，而是避虚就实，重点阐述了 JVM 在软件集成中的作用。

本书强调类加载器体系结构，全面分析 Java 组件特性，文字精练，脉络清晰。我曾在 COM 领域工作 8 年，近两年来，又参与其后续技术公共语言运行库(Common Language Runtime, CLR)的开发，这些工作经历使我真切地感受到，要理解任何组件技术，关键在于首先领会如何在执行时发现、初始化组件代码，并确定其作用域。在 JVM 中，类加载器负责所有这些任务。Stuart 对此作了浓墨重彩的描述。

JVM(以及 Java 平台)当前面临着巨大的竞争，Microsoft 公司以不同程度将大多数 Java 技术包含到 .NET 中，特别是 CLR 中。Sun 如何迎接挑战呢？我们将拭目以待。在比较 JVM 和 CLR 后，我认为，JVM 奉行的“少而精”原则是其力量源泉。无疑，商业压力将促使 Sun 为增强市场竞争力而新添功能，但我也衷心希望，未来的 Sun 能秉持其传统的基本设计精髓。

—— Don Box

2001 年 9 月

California 州 Manhattan Beach

# 前 言

本书详细讨论了 Java 平台面向组件的特性，分析类加载、反射、串行化、本机交互操作及代码生成等技术。

本书对“组件”的定义独特新颖：组件是一个独立的生产和部署单元，可与其他组件结合，以组装应用程序。

对象和组件是不同的概念。对象表示问题域的实体，而组件是被安装解决方案的原子成分<sup>1</sup>。但对象与组件互为补充，正确的设计应兼顾二者。

Java 是备受开发者推崇的现代开发平台，提供了创建类和组件所需要的基础结构。Java 提供封装、继承和多态性，以支持面向对象的编程等重要功能。Java 还提供加载器和各种类型信息，以支持组件。本书将围绕组件阐述如何有效使用 Java 的组件基础结构。

Java 加载器在运行时定位、加载和连接组件。用 Java 加载器您可以做如下工作。

- 部署细粒度组件
- 根据需要动态加载组件
- 从网络上的其他计算机加载组件
- 从自定义存储库加载组件
- 创建存在于多个虚拟机的移动式代码代理
- 导入非 Java 组件的服务

加载器管理组件间的二进制边界。在分布式应用程序和多组件供应商环境中，加载器用于定位和连接所有兼容的组件。

类型信息描述一些代码单元的功能。在某些开发环境中，类型信息仅存在于源代码。而在 Java 中，类型信息不单纯是一种源代码，还是编译类的内在部分，可在运行时通过编程接口使用。因为 Java 类型信息从不被“编译掉”，所以加载器使用它在运行时验证类之间的链接。在应用程序编程中，类型信息的作用如下。

- 串行化 Java 对象的状态，以便在另一虚拟机上重建对象
- 在运行时创建动态代理，以提供可用于任何接口的通用服务
- 将数据转换为另一种表示形式，以与非 Java 组件交互操作
- 将方法调用转换成网络消息
- 在 Java 和 XML 之间转换，XML 是企业系统的新通用语言

---

<sup>1</sup> 此处，“原子”意指不可分割，而非独立的含义。实际上，大多数组件都依赖于其他组件。

- 使用应用程序专用元数据为组件添加注释

通过类型信息，可使原本需要手动编码的任务自动实现。此外，类型信息还有助于组件与未来平台的兼容。

## 读者对象

为了解 Java 应用程序的整个生存期，不仅要考虑对象，还要考虑组件。本书讨论作为组件平台的 Java 的核心功能：类加载器，反射，串行化，以及与其他平台的交互操作。本书适用于在 Java 中设计、开发或部署大量应用程序的读者。在阅读本书前，应了解 Java 语法的基本知识，并具有一些用 Java 编写面向对象程序的经验。

本书并未特别介绍高级 Java 技术，如远程方法调用(Remote Method Invocation, RMI)、企业 JavaBean(Enterprise JavaBean, EJB)、JINI、Java 服务器页(Java Server Page, JSP)、servlet 或 JavaBean。但蕴涵的主题是这些技术的重要基础。通过学习本书介绍的组件服务，可以理解这些高级技术的构建原理，从而有效地加以应用。

安全性也是组件开发和部署的一个重要方面。由于篇幅所限，本书未对此做过多介绍。要详细了解 Java 平台上的安全性，请参见[Gon99](见本书最后的参考书目)。

## 本书内容

全书分三部分。第 1 章简要介绍了组件，第 2 章到第 6 章解释 Java 平台上的加载器和类型信息。第 7 章显示这些服务的更高级使用。

第 1 章介绍面向组件的编程。不仅在编译时，而且在部署和运行时建立组件的关联。本章提出组件编程的关键问题，并将这些问题与后续章节中讨论的 Java 平台服务联系起来。在学习本书时，建议您首先阅读第 1 章，之后可以按自己安排的顺序阅读其他章节。

第 2 章讨论如何使用类加载器和解决类加载故障。类加载器用于控制代码加载，并在同一进程中创建代码间的命名空间边界。通过类加载器，可在运行时动态加载代码，甚至可以从其他计算机加载。类加载器命名空间允许单个 Java 虚拟机中有同一个类的多个版本。使用类加载器，可在不关闭虚拟机的情况下重新加载更改的类。本章将介绍如何使用类加载器，类加载器委托模型如何创建命名空间，如何解决类加载故障，以及如何有效控制引导类路径、扩展路径和类路径。

第 3 章介绍 Java 类型信息。Java 以二进制类格式保存类型信息。这意味着，即使在编译 Java 程序后，仍可访问字段名、字段类型和方法签名。在运行时可通过反射访问类型信息，可使用类型信息来构建能为其他任何对象添加功能的通用服务。本章将讨论动态调用、动态代理、包反射和自定义属性的用法；还将讨论反射性能。

第 4 章讨论 Java 串行化如何使用反射。串行化是一个很好的通用服务示例。不必预先了解类格式的任何知识，串行化可以跨越时间和空间，将代码和状态从一个虚拟机移至另一个虚拟机。学习本章您可以了解到：串行化格式如何嵌入其类型信息样式，如何自定义表示形式；如何扩展默认串行化，如何用自定义外部化代码完全替换，如何调整串行化，以随代码的演变来处理类的多个版本；如何验证反串行化到应用程序的对象，以及如何用指令注解串行化对象，以查找正确的类加载器。

第 5 章返回到类加载器主题，讨论如何实现自定义类加载器。尽管标准类加载器在大多数应用程序中占据主导地位，但自定义类加载器的作用亦不容忽视。通过自定义类加载器，可在加载类时转换类代码。这些转换包括解密，添加性能监视指令，或甚至在运行时新建类。本章将介绍如何将自定义类加载器绑定到 Java 安全体系结构，如何编写自定义类加载器，如何编写可自定义加载类(以及其他任何类型资源)方式的协议处理程序。

第 6 章介绍 Java 本机接口(Java Native Interface, JNI)。JNI 是控制 Java 代码与在其他环境中编写的组件之间边界的基本方式。Java 和本机编程样式存在较大差异，类加载、类型信息、资源管理、错误处理和数组存储的方式等都存在显著区别。Java 提供一组低级别工具，将 Java 对象显示给平台本地代码，和将平台本地代码显示给 Java 对象。本章介绍如何使用 JNI 应用程序编程接口(API)实现 Java 和本机编程样式之间的转换。JNI 存在诸多不足，为此，本书编排了附录 A 以介绍更高级的方法。

第 7 章讨论使用 Java 元数据自动创建源代码或字节码。生成代码是一种高性能重用策略，因为您只用生成在运行时需要的准确代码路径。本章首先介绍 JSP 和 EJB，将此作为自动生成代码的已有应用程序示例，然后介绍在自己的程序中生成代码的一些观点。

附录 A 返回到交互操作主题。在第 7 章介绍的代码生成技术的基础上，附录 A 介绍如何在 Java 和另一组件平台 Win32/COM 之间创建交互操作层。本章以 Jawin 开放源库为例，阐释如何为 Win32 对象生成 Java 占位程序，或由 Java 占位程序生成 Win32 对象。

## 示例代码、Web 站点和反馈

除非特别声明，本书所有示例代码都是开放源代码，可从 Web 站点 <http://staff.develop.com/halloway/compsvcs> 下载。

除非特别声明，书中代码都在 Java 2 SDK 1.3 版本中经过编译和测试。大多数代码也可用于 SDK 1.2、1.3 和 1.4 版；若出现特例，本书将列出一个到适当 SDK 版本的特定引用。

欢迎您对本书提出评论、更正和反馈意见，请将电子邮件发送到 [stu@develop.com](mailto:stu@develop.com)。

# 目 录

第 1 章 从对象到组件	1
第 2 章 类加载器体系结构	8
2.1 组装应用程序	8
2.2 类加载器结构的目标	11
2.2.1 透明性	11
2.2.2 可扩展性	11
2.2.3 功能丰富	12
2.2.4 可配置性	12
2.2.5 处理命名和版本冲突	12
2.2.6 安全性	13
2.3 显式和隐式类加载	13
2.3.1 用 URLClassLoader 显式加载	13
2.3.2 隐式类加载	14
2.3.3 引用类型与已引用类	15
2.3.4 ClassLoader.loadClass 与 Class.forName	15
2.3.5 加载非类资源	16
2.4 类加载器规则	17
2.4.1 一致性规则	17
2.4.2 委托规则	18
2.4.3 可见性规则	19
2.4.4 委托用作命名空间	19
2.4.5 非 Singleton 模式的静态字段	20
2.4.6 隐式加载隐藏了大多数细节	21
2.5 热部署	21
2.6 卸载类	25
2.7 引导类路径、扩展路径和类路径	26
2.7.1 类路径	27
2.7.2 扩展路径	28
2.7.3 引导类路径	29



2.8	调试类加载结构	30
2.8.1	改编应用程序	31
2.8.2	使用-verbose:class 标志	31
2.8.3	改编内核 API	32
2.9	倒置问题和上下文类加载器	35
2.10	小结	39
2.11	资源	40
<b>第 3 章</b>	<b>类型信息和反射</b>	<b>41</b>
3.1	二进制类格式	42
3.1.1	二进制兼容性	42
3.1.2	二进制类元数据	45
3.1.3	从二进制类到反射	47
3.2	反射	47
3.2.1	反射字段	49
3.2.2	get 和 getDeclared 的区别	49
3.2.3	运行时出现的类型错误	51
3.2.4	反射方法	51
3.3	反射调用	52
3.3.1	反射启动器	53
3.3.2	包装基本类型	54
3.3.3	避开语言访问规则	55
3.3.4	反射调用引发的异常	59
3.4	动态代理	60
3.4.1	用委托代替实现继承	60
3.4.2	动态代理使委托通用化	61
3.4.3	实现 InvocationHandler	62
3.4.4	实现转发处理程序	63
3.4.5	将 InvocationHandler 用作通用服务	64
3.4.6	处理 InvocationHandler 中的异常	65
3.4.7	客户程序或服务器都可安装代理	65
3.4.8	动态代理的优点	66
3.5	反射性能	67
3.6	包反射	69
3.6.1	设置包元数据	69
3.6.2	访问包元数据	70

3.6.3	封装包 .....	70
3.6.4	版本控制机制的弱点 .....	71
3.7	自定义元数据 .....	71
3.8	小结 .....	74
3.9	资源 .....	74
<b>第 4 章</b>	<b>串行化处理机制 .....</b>	<b>75</b>
4.1	串行化处理和元数据 .....	75
4.2	串行化基础知识 .....	76
4.2.1	串行化忽略的一些字段 .....	78
4.2.2	串行化与类构造函数 .....	79
4.3	使用 readObject 和 writeObject .....	80
4.4	将流与类匹配 .....	81
4.4.1	serialVersionUID .....	82
4.4.2	重写默认 SUID .....	83
4.4.3	兼容的和 not 兼容的更改 .....	84
4.5	显式管理可串化字段 .....	85
4.5.1	ObjectInputStream.GetField 应用提示 .....	86
4.5.2	编写器对格式的处理 .....	86
4.5.3	重写类元数据 .....	88
4.5.4	性能问题 .....	88
4.5.5	自定义类描述符 .....	88
4.6	停用元数据 .....	89
4.6.1	在 defaultWriteObject 后写入自定义数据 .....	89
4.6.2	可外部化 .....	90
4.6.3	用 writeObject 只写入原始数据：错误做法 .....	91
4.7	对象图 .....	93
4.7.1	用 Transient 关键字缩减图大小 .....	94
4.7.2	保存标识 .....	94
4.7.3	通过 reset 关键字来利用无用单元收集器 .....	95
4.8	对象替换 .....	96
4.8.1	流控制替换 .....	96
4.8.2	类控制替换 .....	99
4.8.3	替换的排序规则 .....	100
4.8.4	控制图排序 .....	104

4.9	查找类代码	106
4.9.1	RMI 中的注解	107
4.9.2	RMI MarshalledObject	108
4.10	小结	109
4.11	资源	110
<b>第 5 章</b>	<b>自定义类加载器</b>	<b>111</b>
5.1	Java 2 安全性	112
5.2	自定义类加载器	115
5.2.1	Java 2 之前的自定义类加载器	115
5.2.2	SDK 1.2 后的类加载	116
5.2.3	转换类加载器	117
5.3	协议处理程序	122
5.3.1	实现处理程序	123
5.3.2	安装自定义处理程序	125
5.3.3	加载器和处理程序之间的选择	127
5.4	为需要的加载器传递安全性	128
5.5	读取自定义元数据	129
5.5.1	版本属性示例	130
5.5.2	可串行类用作属性	130
5.5.3	在类加载时读取属性	133
5.5.4	调试支持	138
5.6	小结	138
5.7	资源	139
<b>第 6 章</b>	<b>交互操作 1: JNI</b>	<b>140</b>
6.1	交互操作的原因	140
6.2	本机代码的危险	141
6.3	查找和加载本机代码	142
6.3.1	名称映射	143
6.3.2	类型映射	143
6.3.3	重载名	145
6.3.4	加载本机库	146
6.3.5	类加载器和 JNI	147
6.3.6	加载本机库的常见错误	149
6.3.7	本机加载释疑	152

6.4	从 C++调用 Java	152
6.4.1	最小化往返次数	154
6.4.2	性能比较	156
6.4.3	JNI 和反射调用的区别	157
6.5	JNI 中的错误处理	159
6.5.1	本机代码中的故障	159
6.5.2	处理 C++异常	159
6.5.3	从本机代码处理异常	160
6.5.4	从本机代码抛出 Java 异常	163
6.6	资源管理	163
6.6.1	与无用单元收集器的交互	163
6.6.2	管理本机资源	169
6.6.3	管理数组	170
6.6.4	管理字符串	174
6.7	小结	176
6.8	资源	176
<b>第 7 章</b>	<b>生成式编程</b>	<b>177</b>
7.1	使用生成式代码的原因	177
7.1.1	建立变性模型的面向对象的方法	178
7.1.2	按绑定时间进行分析	180
7.1.3	分离绑定时间和规范	180
7.1.4	选择规范语言	181
7.1.5	重用性的高需求	182
7.1.6	小型域分析很危险	182
7.2	用 Java 生成代码的原因	182
7.2.1	类型信息用作自由规范文档	182
7.2.2	类加载支持灵活绑定模式	183
7.2.3	易于生成 Java 源文件	183
7.2.4	易于生成 Java 二进制类	183
7.2.5	代码生成功能有利于提高性能	183
7.2.6	代码生成的责任级别	184
7.3	绑定时间和模式的分类	184
7.4	RMI 中的代码生成	186
7.5	JSP 中的代码生成	187

7.6	EJB 中的代码生成	189
7.6.1	部署描述符	191
7.6.2	替补实现	193
7.7	生成强类型集合	194
7.8	生成自定义串行化代码	197
7.9	小结	201
7.10	资源	203
<b>第 8 章</b>	<b>展望</b>	<b>204</b>
8.1	当前状况	204
8.2	发展方向	205
8.3	资源	205
<b>附录 A</b>	<b>交互操作 2: 连接 Java 和 Win32/COM</b>	<b>206</b>
A.1	综述	206
A.2	半透明占位程序	207
A.3	平台差异	209
A.4	组件对象模型	210
A.4.1	COM 加载器	210
A.4.2	COM 类型信息	211
A.4.3	COM 对象生存期	212
A.4.4	COM 类型发现	213
A.4.5	COM 错误处理	213
A.4.6	COM 线程亲缘性	214
A.4.7	COM 安全性	216
A.5	Win32 动态链接库	216
A.5.1	DLL 加载器	216
A.5.2	DLL 类型信息	217
A.5.3	DLL 对象生存期	218
A.5.4	DLL 类型发现	218
A.5.5	DLL 错误报告	218
A.5.6	DLL 线程亲缘性	218
A.5.7	DLL 安全性	219
A.6	编组体系结构	219
A.6.1	共享占位程序	219
A.6.2	通用占位程序	223
A.6.3	指令字符串	223

A.7 生成占位程序 .....	225
A.7.1 生成共享占位程序 .....	225
A.7.2 生成接口占位程序 .....	228
A.8 小结 .....	230
参考书目 .....	231

# C H A P T E R

# 1

## 从对象到组件

优秀的 Java 程序兼具面向对象和面向组件的特性。本章首先描述对象观点和组件观点之间的区别，然后利用一个面向对象的程序设计，并将这个程序设计修改为面向组件的特性，以此来演示二者之间的区别。

我们以典型的“联系人管理系统”问题域为例。此问题域的一个要点是根据各种不同标准查询联系人信息的功能。程序清单 1-1 显示了 `Contact` 和 `ContactFinder` 接口的部分清单。

程序清单 1-1 `Contact` 和 `ContactFinder` 接口

```
package contacts;
public interface Contact {
    public String getLastName();
    public void setLastName(String ln);
    public String getFirstName();
    public void setFirstName(String fn);
    public String getSSN();
    public void setSSN();
    //etc. for other standard fields
}

//contacts/ContactFinder.java
package contacts;
public interface ContactFinder {
```

```

    Contact[] findByLastName(String ln);
    Contact[] findByFirstName(String fn);
    Contact[] findBySSN(String ssn);
    //other more exotic search methods...
}

```

有了这些接口，自然会联想到用于访问联系人信息的各种客户应用程序。程序清单 1-2 给出一个简单的控制台客户程序 `ListByLastName`，该程序根据指定的姓列出所有联系人。如果 `Contact` 和 `ContactFinder` 接口已正确捕获了问题域，则认为此程序是一个完全的 OO(object-oriented, 面向对象)型设计。注意，程序的接口和实现是完全分离的。程序 `ListByLastName` 中的变量均为接口类型，如 `Contact` 和 `ContactFinder`。如果未来的版本需要新的代码实现，更改一行代码即可。

#### 程序清单 1-2 ListByLastName

```

package contacts.client;

import contacts.*;
import contacts.impl.*;

public class ListByLastName {
    public static void main(String [] args) {
        if (args.length != 1) {
            System.out.println("Usage: ListByLastName lastName");
            System.exit(-1);
        }
        ContactFinder cf = new SimpleContactFinder();
        Contact[] cts = cf.findByLastName(args[0]);
        System.out.println("Contacts named " + args[0] + ":");
        for (int n=0; n<cts.length; n++) {
            System.out.println(cts[n]);
        }
    }
}

```

通过继承机制可以容易地扩展这种面向对象设计。设想一下，“联系人管理系统”的每个购买者都要将一些客户数据项添加到 `Contact` 基本概念，只需扩展 `Contact` 接口(为每个客户创建不同的子接口)即可实现。程序清单 1-3 是一个示例扩展，用以所需查询人员的文凭。

#### 程序清单 1-3 DiplomaticContact

```

package contacts.diplomatic;

import contacts.*;

public interface DiplomaticContact extends Contact {
    public float getSpyProbability();
}

```



```

public void setSpyProbability(float newProb);
public Contact[] getKnownAssociates();
//etc.
}

```

程序清单 1-3 体现了出色的联系人管理设计方法——在保留对特定实现的改进和增强能力的同时，创建问题域模型。这是一个非凡的成就，面向对象语言(如 Java)的成功源于对实现这些目标的支持。但遗憾的是，当前的设计并未着手于解决组件部署问题。

组件(Component)是一个独立的开发和部署单元，组件与组件组装在一起构成应用程序。对象和组件的概念之间有一些重叠。对象是类的实例，面向对象的设计亦可称为面向类的设计。组件一般只是一个经过编译的类(compiled class)，或一组经过编译的类<sup>1</sup>。您可能会问，对象和组件最重要的“产品”都是类，那么对象方法和组件方法的显著区别在哪里呢？答案是：对象方法强调设计和开发，而组件方法强调部署。

面向对象的设计方法强调系统中各个实体间开发时(development-time)的关系，面向组件的设计方法将这些关系扩展到应用程序生存期的其他阶段，特别是开发阶段和部署阶段。面向对象的方法引出下列问题：

- (1) 程序的设计是否捕获了问题域的相关部分？
- (2) 接口和类易于扩展和修改吗？

面向组件的设计方法引出下列问题：

- (1) 客户程序如何在运行时查找实现类？
- (2) 在运行时，若有多个版本的实现类，会出现什么情况？
- (3) 组件如何定位和加载所需的配置信息？
- (4) 如果进程或容器需要临时关闭，将出现什么情况？可透明地保存和恢复未完成的工作吗？组件实例可以从一个容器移到另一容器吗？
- (5) 在一个产品系列中，对其中一个产品的开发和维护对其他产品有何影响？
- (6) 基于代码的组件与客户的特殊任务无关吗？
- (7) 若旧组件几乎完全适用于新系统，是否可在不触及源代码的情况下，以未预期的新方式扩展旧组件吗？
- (8) 若系统的一部分必须在不同的软件平台实现，并与其余部分无缝地交互操作，将出现什么情况？

上面的问题看起来很重要。那么，与对象相比，组件为什么并不引人注目呢？

任何一组工具都会特别支持一些解决方案，而“冷落”另一些解决方案。开发人员友好的计算机环境会妨碍对于部署问题的分析。在任何时刻，开发人员的计算机都对应一个复杂的不断演变的系统状态。事实证明，通过编译器和其他开发工具

<sup>1</sup> 组件也可能是其他一些独立部署单元：文本文件、图形图像、数据文件或脚本。