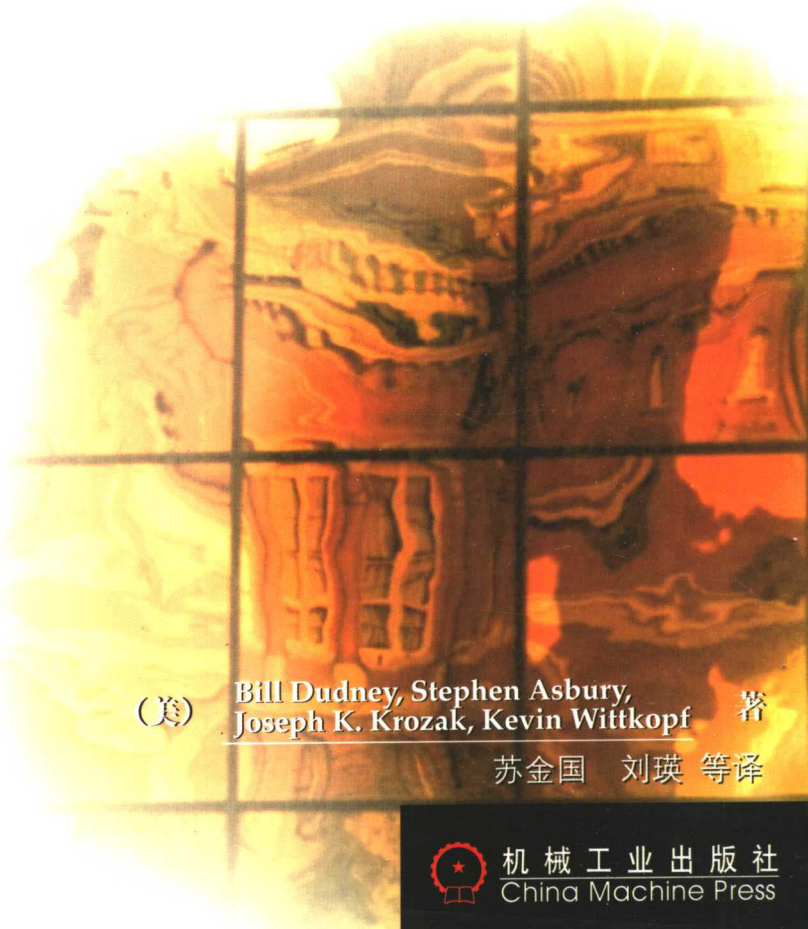
HZ BOOKS

# J2EE反模式

J2EE AntiPatterns



(美) Bill Dudney, Stephen Asbury, Joseph K. Krozak, Kevin Wittkopf 著

苏金国 刘瑛 等译

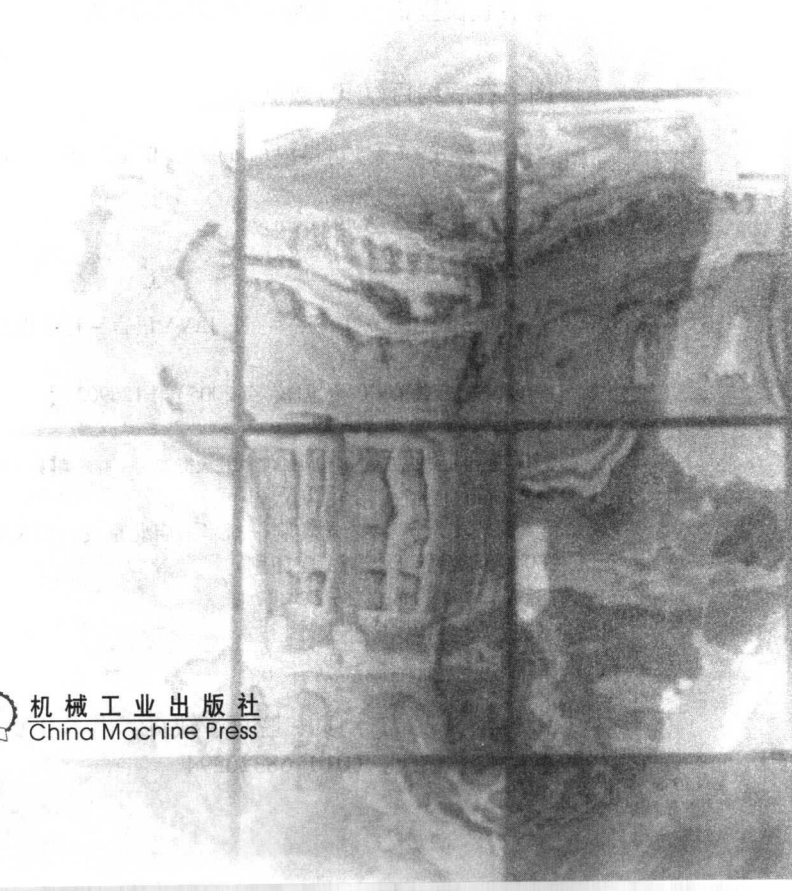
机械工业出版社
China Machine Press

# J2EE反模式

## J2EE AntiPatterns

(美) Bill Dudney, Stephen Asbury, Joseph K. Krozak, Kevin Wittkopf 著

苏金国 刘瑛 等译



机械工业出版社
China Machine Press

所谓模式，就是以一种正式模板的形式来描述好的实践做法，而反模式是采用相同的形式来描述不恰当的实践做法。本书不仅指出了许多 J2EE 开发中存在的反模式及其症状和引发的后果，而且分析了其产生的原因，并至少给出了一种重构方案，指导开发人员逐步转换代码实现，从而得到更好的设计。本书的目的不是想指导开发人员如何完成 J2EE 设计和开发，而是通过指出应用开发中常犯的错误及修复错误的方法，来指导读者成为更好的 J2EE 开发人员。

Bill Dudney, et al.: J2EE AntiPatterns (ISBN: 0-471-14615-3).
Authorized translation from the English language edition published by John Wiley & Sons, Inc.


**本书版权登记号：图字：01-2003-7148**

**图书在版编目(CIP)数据**

J2EE 反模式/(美)达得内(Dudney, B.)等著；苏金国等译. -北京：机械工业出版社，2006.1
书名原文：J2EE AntiPatterns
ISBN 7-111-17702-9

Ⅰ.J…　Ⅱ.①达…②苏…　Ⅲ.JAVA 语言-程序设计　Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2005)第 126903 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：赵　越　　刘立卿
北京中兴印刷有限公司印刷·新华书店北京发行所发行
2006 年 1 月第 1 版第 1 次印刷
787mm×1020mm 1/16·24 印张
印数：0 001-4000 册
定价：49.00 元

# 序

心理学家说，如果一而再、再而三地重复做同一件事情，并希望会有不同的结果出现，这就说明脑子可能有问题了，可以把这定性为一种精神失常。这种情况对于软件开发过程也同样适用。遗憾的是，我们也经常重复着某些设计，却未曾意识到这些设计会招致许多大问题。在这里就可能存在着一些反模式，我们要引以为戒。所谓“反模式”(AntiPattern)，可以认为是以一种正式模板的形式来表述某种不妥当的实践做法。当然，我们也可以用同样的形式来描述好的实践做法，这就是“模式”。因此，反模式就是某种不好的实践做法，它们很常见，因此有必要把它们正式记录下来。

一项新的技术引入后，许多人可能都希望能了解技术的细节。在 J2EE 诞生的前 3 年间，这种情况尤其明显。在此期间，大批有关如何使用 J2EE 的书籍问世就是一个明证。其中许多书都是在用我们可以理解的某种语言来介绍 J2EE 规范。不过，随着 J2EE 越来越成熟，越来越多的开发人员了解了 J2EE 的细节，并且已经开始尝试应用 J2EE，有关 J2EE 设计的书籍开始出现了。这些书并不是面面俱到地教你如何使用这个技术，而是把目光更多地聚焦在如何用这种技术来进行设计。《Core J2EE Patterns》(中文版《J2EE 核心模式》已由机械工业出版社出版。——编者注)就是一例，就是采用模式的形式来关注最佳实践。

本书的用途很多。一来它可以用来确认那些不好的实践做法，你应当予以避免。二来它可以提供证据，使你确信某种做法确实是不好的实践。三来，如果参加有关设计的会议，你还可以拿书中的名词来说明设计系统时哪些是不该做的。或者，有些开发高手总认为自己的设计无与伦比(不过，除了管理层会受他们的“蒙蔽”外，所有人都很清楚并不是那么回事)，如果你对他们的自负很不以为然，也可以拿这本书来杀杀他们的锐气。

阅读本书的许多读者对 J2EE 可能已经很精通。所以，请不要把它当成一本说教“不要这么做，不要那么做”的书来读，而是应该换个角度。如果你看这本书后有新的想法，“哦，我想原先可能不该那么做”，那就对了，这才是读本书的正确观点。我相信本书的作者都很清楚这一点，因为他们不仅告诉你哪里可能做错了，还解释了如何进行修正。这正是本书的妙处所在。

书中提供了大量绝好的信息。作者用通俗的手法使我们能迅速地理解反模式的实质。正是因为它如此浅显易懂，所以你在进行设计时，完全可以把它当成一个助手放在身边。

最后还要提醒一句，一定要阅读 Web 服务反模式这一章(第 9 章)。尽管对于 Web 服务不乏争议和困惑，但阅读这一章肯定会让你长久受益。尽情地享用这本书吧。

——John Crupi，《Core J2EE Patterns》的著作者之一，Sun 公司杰出工程师

---

参加本书翻译的人员有：苏金国、刘瑛、林琪、范松峰、杨健康、张莹、易竞、程龙、卢鋆、江健、丁小峰、陈永志、牛亚峰、高强、何跃强、孙春娟、张伶。

# 前　言

如今发布的太多软件往往都是漏洞百出，运作得也很糟糕。遗憾的是，通常很难准确地找出是哪里出了问题，以及需要做哪些工作才能让情况好转。反模式正是衔系在“问题”和“解决方案”之间的“救命线”。

本书对于如何识别不好的代码以及如何采用 J2EE 进行设计提供了一些实用的建议，并介绍了可以让代码变得更优秀的方法。在书中，读者将了解到诸多 J2EE 反模式，这里不仅会给出典型编码和设计错误的形式化定义，还会提供一些存在反模式的实际代码示例。对于每一种反模式，都至少提供了一种重构方案（通常会提供多种重构方案）。对每个反模式的介绍都是一个能让你了解如何让代码变得更好的过程。

在设计和开发任何应用程序时，我们认为，你应当同时兼顾到构建应用的一些正面（模式）和反面（反模式）的例子。我们都想用更好的办法去构建更好的软件，希望能从自己和别人的失败中得到教训。例如，在一个典型的 J2EE 应用开发中，通常会有一个包含多层的体系架构，数据库层保存应用程序所用的数据，Enterprise JavaBeans（EJB）层要与数据层交互，并包含业务逻辑，Web 层则提供用户界面。构建各个层时犯错误的机会是很多的。一种典型的错误就是让 Web 层直接与实体 EJB 交互。采用这种方式编写的应用程序存在很严重的性能问题。随后出现的会话外观（Session Façade）模式［Alur，Crupi，Malks］可以用来避免所写应用程序性能低下的问题。不过，原来的一些应用程序在编写时并没有使用此模式，所以还需要对这样一些应用程序进行修正。但是你大可不必把原来的代码全盘扔掉，再从头来过，第 9 章提供了一种称为外观（Façade）的重构模式，其中介绍了可以采取哪些实用步骤来修正性能差的问题。

## J2EE 中的反模式

J2EE 作为一种体系架构，不仅功能相当强大，而且非常灵活。然而与日常生活中的其他情形一样，鱼和熊掌不可兼得。在 J2EE 中，一方面我们可以在功能和灵活性方面大有收获，但另一方面又不得不放弃简单性。

J2EE 领域的覆盖面极广，从数据库访问到基于 Web 的表示可谓一应俱全。J2EE 中的许多库（或部件）本身就已经很复杂了，单是这些库就完全可以用整本书来介绍。在图 1 中可以看到 J2EE 所覆盖的一些领域。

本书中所介绍的反模式涉及 J2EE 的绝大多数概念。几乎 J2EE API 的每一部分都存在反模式，从数据库访问到使用 JavaServer Pages（JSP），无一例外。

## 反模式

所谓反模式就是重复地应用某些代码或设计，而这些代码或设计会导致不良的后果。这种不良后果可能是性能很差，代码很难维护，甚至是项目完全失败。反模式以一种详细或特定的方法来捕捉代码错误。

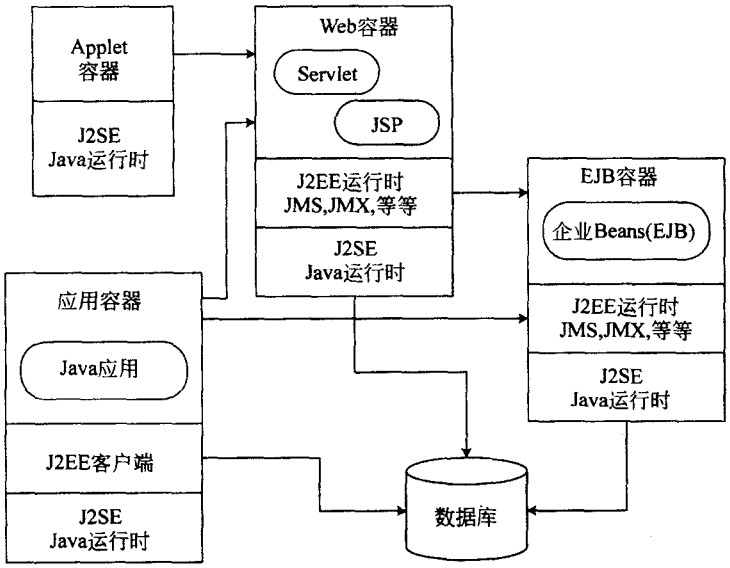



图 1　J2EE 概览

本书中的反模式将采用一种模板形式来介绍。通过这个模板，有助于确保得到一致的形式化定义，从而使反模式的学习更为容易。这个模板还提供了一种方法，由此可以从不同视角来阐述反模式的有关信息。有些人可能会从“症状及后果”发现其代码中存在的反模式，还有些人则可能根据“典型原因”找出隐藏的反模式。这个模板由两部分组成，先是一个目录(catalog)项列表，这些目录项都是些短语或名字而已，后面的详细(detail)项列表则较为复杂，其中每一项都会做更为深入的说明。这些目录项和详细项如下所列。

**目录项**

- *也称为*。在此列出与该反模式关联的一个或多个替代名。
- *常出现的领域*。这一项会指出该反模式最有可能出现在哪些领域中。
- *重构方案*。这一项会列出可以采用哪些适当的重构方案来改进代码和设计，从而消除反模式。
- *重构方案类型*。这一项会列出针对该反模式可以应用哪些类型的重构方案。
- *根本原因*。这一项会提供出现该反模式的根本原因。对这些问题做了妥善修正后，就可以很好地避开该反模式。
- *不平衡的外力因素*。这一项会说明项目中存在哪些不协调或不平衡的外力因素。一种反模式冒出时，往往是因为存在着一些外力，这些外力可能会导致一些问题，而这些问题正是该反模式的根本原因。不过，不要奢望在所有方面都达到平衡。
- *奇谈怪论*。这一项会引用一些言论，你经常会从一些开发人员或管理人员口中听到这些说法，他们的项目就饱受着反模式之苦。

### 详细项

- *背景*。这一部分会介绍所述反模式的背景，并从个人角度提供该反模式的一些实际例子。这种反模式会对各种项目带来哪些影响？该反模式是如何出现的，或者是如何被注意到的？这些有关内容也将在这一部分加以说明。
- *一般形式*。反模式往往是以在此所述的形式暴露的。你可以在这里找到必要的信息，从而帮助你判断你的应用程序是否遭遇到反模式陷阱。这一部分还包括一些图表或代码，这样你就能对在此所讨论的概念有一个具体的认识。
- *症状及后果*。症状就是应用开发中的“痛处”，这些症状能够很好地指示出反模式的存在。后果则说明了如果对反模式置之不理，任由它对你的设计和代码“找麻烦”，会导致什么样的结果。
- *典型原因*。这一部分会介绍出现该反模式的“根本原因”的一些具体体现，还会提供一些相关实际例子的信息，这些实际的问题在过去曾导致项目出现过这种反模式。
- *已知的例外情况*。所谓例外情况是指，即使存在这种反模式，也不会对项目带来负面影响。
- *重构方案*。这一部分详细介绍了针对这个反模式如何采用有关的重构方案，从而最有效地消除应用程序中的反模式。
- *其他变种*。这一部分会重点强调一些重要例子，从中可以了解到这种反模式如何依具体情况不同而存在差异。
- *示例*。这一部分会提供一个例子，从这里我们可以了解到存在反模式的代码或设计会是什么样子。
- *相关解决方案*。这一部分包含了一些其他反模式的相关资料，这些反模式通常会与此反模式一同出现，另外此部分还可能包括其他一些有助于解决此反模式的重构方案。

## 重构方案

所谓重构方案(refactoring)是一种原则性的方法，通过改造代码的实现，从而得到更优的设计，而对外部所见的行为表现不会带来变化。

要想成功地进行重构，关键是要有一组好的单元测试。仅仅是单元测试这个主题，就可以用整本书的篇幅来介绍，而且这方面的书也确实不少。不过，我们起码应该知道，如果没有预先设定的一组单元测试，你永远也无法肯定你的重构是否成功，特别是在你的重构不影响外部所见行为表现的情况下。

本书中的重构方案也同样采用一种模板的形式来介绍。实际上，这个模板借用自 Martin Fowler 那本享有盛誉的《Refactorings》(Fowler，2000)。与介绍反模式的模板类似，重构模板提供了一种以标准化形式介绍重构的方法，通过这个模板，我们可以更容易地学习重构方案的有关内容。该模板的形式如下。

- *重构图*。每个重构图部分都包含一条说明问题的语句(一个句子或一个短语描述，指出代码或设计中存在什么问题，需要加以修正)，还包含一条有关解决方案的语句(一个

句子或一个短语描述，指出需要做什么工作来修正上述问题），另外还包含一个相关的图（一个简单的图，以图的形式说明需要做什么工作来修正上述问题）。这里的图可以是一个需要重构的代码段或 UML（统一建模语言，Unified Modeling Language）设计图，紧接着还会提供一个视图，由此我们可以了解到应用了重构之后的代码或 UML 图会是什么样子。

- *动机*。这一部分说明为什么要应用此重构方案。这里通常包含有关的描述，指出采用此重构方案将会避免或减少哪些不良后果。
- *方法步骤*。这一部分会介绍如何通过一个循序渐进的过程，将代码从当前的不好状态调整到一种较好的状态。
- *示例*。这一部分包含一个受反模式影响的代码或设计例子。针对这种不好的代码或设计，将分别应用重构的各个步骤，直到重构完成。

## 为什么写这本书

如何从头开始新建一个 J2EE 项目，在这方面有许多不错的书籍可供参考，但这些书籍对于如何让你现有的应用程序表现得更好却无济于事。本书介绍了开发 J2EE 应用时经常容易犯的错误，并且提供了相应的重构方案，以帮助你摆脱这些问题。本书的目标就是，指出应用开发中通常会犯哪些错误，并提供必要的工具来修复出了问题的地方，从而帮助你成为一个更好的 J2EE 开发人员。重构方案提供了一些实用的建议，你将由此了解到，如何将受反模式所累的设计和代码转变为一个更简洁、更可维护的应用程序。换句话说，这不是一本介绍如何来完成 J2EE 设计和开发的书，相反，本书要介绍的是在设计和开发过程中存在哪些常见的错误，以及如何进行修正。

在写这本书时我们有许多收获，如果你能像我们一样有这么多收获，或者至少了解到其中的一部分，我们的目的和希望也就达到了。在我们多年构建 J2EE 应用的体验中，曾经见过（自己也曾写过）一些很糟糕的代码，这些代码都有很不好的后果。通过捕获一些最常见的问题，你会看到你的代码将工作得更好、更快，而且也无需你像原先那么费劲。我们希望，你能从我们的错误中吸取教训，并成为一个更优秀的 J2EE 开发人员。

# 致　谢

首先要向 Christ 致以最诚挚的谢意，感谢他一直以来对我的教诲，正是他不懈地鼓励我最大限度地发挥自己的潜能，才使我能够超水平发挥。还要感谢我的好妻子 Sarah，没有她的支持和爱，我是无法支撑下去的。另外要谢谢我可爱的孩子们，是你们让我的生活像现在这样多姿多彩。Andrew、Isaac、Anna 和 Sophia，你们就是快乐的化身。我的父亲总是教育我说，只有站起身来，持之以恒地前进，不断地加油，最后才能登上山顶。谢谢您告诉我这些。

感谢 Jack Greenfield 鼓励我换个角度想问题。在你的督促之下，我的思维变得更加活跃，谢谢你，Jack。还要感谢 John Crupi 和 Bill Brown，你们为这本书的内容提供了极有价值的反馈。由于你们的努力，才使这本书更出色。最后，我要感谢 Eileen Bien Calabro，能把我那乱七八糟的草稿变成纯正英语的书稿绝不是一件容易的事，由于你的贡献我才得以奉上这样一本比原先强得多的书。我在写这本书的时候学到了许多东西，希望你也能有同样的收获。

——BD

我要感谢以前以及现在一直与我一起，共同为 AntiPatterns 这本书做出努力的人们，不管你们是不是清楚你们的贡献有多大，在此都要致以感谢。我要特别感谢我的妻子 Cheryl；结婚 12 年以来，对你，我最好的朋友和妻子，感谢之情无以言表。

——SA

我要感谢我亲爱的妻子 Jennifer，她无尽的耐心、爱与支持永远是我的后盾。感谢上帝赐给我美丽的女儿，Kristina，我的小甜心，只要你想做，就去做，没有做不成的！还要感谢我的好兄弟 Raymond，你的智慧总是能指引我前进。最后，还要感谢我的母亲为我付出的爱、鼓励和无私的奉献。

——JKK

我想把这本书谨献给我一生中的最爱。首先，要献给我的妻子和最好的朋友，Catherine，以此感谢在我写这本书时她给予我的不变的爱、支持和鼓励。没有你，我什么也做不了！这本书还要献给我的两个出色的女儿，你们总是精力充沛，不断地感染着我。

我要感谢这本书的所有合作者们，感谢大家艰苦的劳动和所做出的卓越贡献。在此要特别感谢 Bill Dudney 邀请我参与这本书的编写(说实话，我有些不请自来!)，还要感谢他费心尽力地协调和审阅我们完成的书稿，以保证全书的内容统一。

我还要感谢 Wiley 技术出版公司的所有工作人员(特别是 Eileen Bien Calabro)，是大家艰苦的努力、无私的指导和无尽的耐心，才使我有勇气投入写作。

——KW

# 目　录

# 第1章　分布与扩展

**本章内容包括：**

本地化数据
误解数据需求
误算带宽需求
超负荷运转的网络中心
手持利斧乱砍一气的人
提前规划
选择适当的数据体系架构
划分数据和工作
为将来扩展做出规划(企业规模的面向对象)
规划实际的网络需求
使用特殊化网络
务求谨慎
丢掉有问题的硬件

无论企业解决方案的规模是大还是小，网络都是其核心所在。J2EE 也不例外，所有 J2EE 解决方案都是围绕着网络构建的。这说明，将一个应用移到 J2EE 世界中时，J2EE 开发人员将会遇到新的问题，并带来新的复杂性。

开发人员从原先构建小规模的企业解决方案转向构建大型企业解决方案时，不仅需要做更多的架构选择，面对更多的复杂性和后果，还需要解决一些在小规模领域中根本就不存在的新问题。为了在 J2EE 分布式网络世界中取得成功，开发人员要做到的第一步就是了解他们所处的新环境，还要明白这个环境与他们所熟悉的那个非网络化的小环境有什么不同。

对于分布式计算及其相应的 J2EE 编程存在许多常见的误解，以下 8 个有关分布式计算的谬论对此做了最好的总结。我最早是在 2000 年的 JavaOne 大会上听 James Gosling 谈到了这些谬论，据 Gosling 说，他是从 Peter Deutsch 那里了解到的。尽管知道这些说法的人不算多，不过，这些谬论确实很好地总结出了分布式应用开发人员所要面对的诸多危险。因此，对于所有 J2EE 开发人员来说，理解这 8 个谬论都是至关重要的。

**谬论 1：网络是可靠的。**在分布式计算中，人们往往很轻易做出一个假设，认为网络能一直“死心踏地”地为我们工作。但事实根本不是这样。网络确实变得越来越可靠了，但是仍有可能遭遇不可预知的电源故障和物理(硬件)方面的危险，因此网络不可能提供百分之百的可靠性。你也不能期望网络是百分之百可靠的。

**谬论 2：延迟为 0。**延迟(latency)是消息通过网络传输所花费的时间。我们大家都知道，这个值绝非为 0。尽管这种延迟本身对你所能传送的数据量没有太大影响，但却会显著地影响发送和接收数据。因此，延迟可能会对你的应用程序产生一些有趣的异常作用。例如，在某些网络技术中，如果一个程序 A 向程序 B 发送两个消息，这两个消息有可能不按发出时的顺序到达。更常见的情况是，不同的消息可能经由分支路径传输，其中经过较短路径的传输反而耗时更长(译者注：如从 A 到 B 再到 C 的消息反而比从 A 直接到 C 的消息更快到达，这一章后面还会说明这个问题)。

**谬论 3：带宽是无限的。**尽管网络速度已经有大幅提高，但带宽仍然是有限的。大多数家

庭用户的带宽是 1.5 Mbps，大多数企业网络的带宽为 100 Mbps。对于那些完成关键任务的应用程序，这些限制尤其重要，因为随着时间增长，它将影响网络传送的数据总量。带宽的问题在于，很难找出带宽都用到哪里去了。网络协议存在一些开销，诸如 JMS（Java Message Service，Java 消息服务）等消息实现也有自己的开销。另外，企业网络中还可能存在一些背景噪声，即因电子邮件、Web 浏览和文件共享等服务所引入的开销。

**谬论 4：网络是安全的。**无论是管理人员还是用户，都要面对安全这个难题。有关的挑战包括认证、授权、保密和数据保护。如果你对安全感兴趣，可以先从 Bruce Schneier 所著的《Secrets and Lies》一书（Schneier 2002）入门。然后再与一些有经验的专家一同共事，从他们那里汲取经验。要记住，网络安全不可能通过仅仅隐藏自己的算法和技术来实现。只有当解决方案是经过同行评审过（peer reviewed）的时候，才能真正做到安全。

**谬论 5：拓扑结构不会改变。**拓扑结构（topology）是计算机之间的物理互连关系。硬件故障、操纵开关以及手提电脑的位置变动有可能会使网络中减少一些计算机，或者会改变经过网络的路径，这都可能使网络拓扑有所改变。新的无线技术更是允许人们从世界各地来访问你的网络。这就使得应用程序的开发已经成为一个复杂的行当，因为无线连接可能相当慢，用户可能会走开一会再回来，而且他们可能存在着安全问题。你需要考虑为每一种客户都提供一种定制的接口。这些接口可以根据需求减少或增加功能，从而满足客户不同带宽的限制。你还必须考虑到有些客户会一直连接或一直断开，这也会改变你的数据体系架构。

**谬论 6：只有一个管理员。**大型公司总有多位系统管理员。同一个问题可能会采用多种方式来解决，而且在软件更新过程中也可能存在着时间和版本上的差别。一定要在设计阶段尽可能地对管理和维护多做规划。

**谬论 7：传输代价为 0。**在这个世界上，如果只传输数据就要花钱的话，开发人员肯定会非常注意诸如服务质量和速度价格比之类的问题。尽管大多数网络不必为传输的每一位数据付费（或者至少目前还不需要付费），但数据的传输确实存在着代价。购买更大型的硬件和备份网络的成本是昂贵的。如果某个解决方案可以以更少的成本提供同样的功能，那么不要犹豫，放手去做吧。不要认为只有时间才是做出选择的惟一衡量标准；要花多少钱，这同样可以决定你的取舍。实业型的人都很了解这一点，但是我们很容易陷入一种拿钱来解决问题的境况，而不是依靠优秀的设计来摆脱困境。另一方面，如果你确实有钱，而且这样能很快地解决问题，也不妨为之。

**谬论 8：网络是同构的。**网络是多种技术的汇集。分布式应用必须处理形形色色的标准。单从这些标准的缩写来看，就像是一盆杂烩汤，如 TCP/IP（传输控制协议/Internet 协议，Transmission Control Protocol/Internet Protocol）、HTTP（超文本传输协议，HyperText Transport Protocol）、SOAP（简单对象访问协议，Simple Object Access Protocol）、CORBA（通用对象请求代理体系结构，Common Object Request Broker Architecture）、RMI（远程方法调用，Remote Method Invocation）以及其他诸多协议。

可以想见，上述 8 个简单的误解可能会导致出现大量问题，也会带来本章所要讨论的许多反模式。我们希望，你能够找出和避免本书介绍的反模式，从而克服这些问题，建立真正的分

布式、可扩展的解决方案。

许多开发人员在用 J2EE 构建分布式、可扩展的解决方案时，会出现一些问题和错误，以下反模式就主要强调这些问题。这些反模式通常由上述对分布式计算的误解而滋生，不过也可能是因为其他一些原因造成的，如要求尽快交付市场的时间限制等。这些反模式讨论了一些常见的误解，这些误解往往是由管理人员和分析人员等非技术渠道引入的，由于这些误解的误导，开发人员在为其解决方案选择基本架构时会做出不太理想的选择。

这里介绍的各种反模式所代表的体系架构方面的问题不仅会影响解决方案中较小的组成元素，也可能会影响 J2EE 应用的整个网络。这些绝非单个开发人员就可操控的代码问题，而是涉及到体系架构和设计，要求整个团队在开发过程中从始至终都要时刻放在心上。

**本地化数据(Localizing Data)**。本地化数据过程是指把数据放在某个位置上，而且这个位置只能由大型解决方案中的某个元素来访问。例如，将数据放在一个特定 servlet 的静态变量中，这就会导致其他机器上的 servlet 无法访问此数据。当解决方案由小型部署发展为较大规模的部署时，往往就会出现这种反模式。

**误解数据需求(Misunderstanding Data Requirement)**。规划不当或功能过度扩张会导致这种不好的情况，你可能会在网络上传送过多或过少的数据。

**误算带宽需求(Miscalculating Bandwidth Requirement)**。如果没有按实际情况计算带宽需求，最终的解决方案肯定会出现重大问题，而且主要与糟糕的性能有关。有时是在一个解决方案中错误地计算了带宽。有时是在一个网络应用了多个 J2EE 方案而导致出现这种情况。

**超负荷运转的网络中心(Overworked Hub)**。网络中心(hub)是 J2EE 应用的集合点。这些网络中心可以是数据库服务器、JMS 服务器、EJB 服务器以及其他应用，在其中驻留或实现了你所需的特定功能。如果一个网络中心超负荷运转，它的速度就会变慢，还有可能完全瘫痪。

**手持利斧乱砍一气的人(The Man with the Axe)**。失败是人生中不可避免的一部分。对失败做出规划对于构建一个健壮的 J2EE 应用程序也是一个很重要的部分。

## 1.1 反模式：本地化数据

*也称为*：简单数据模型，数据位于节点上

*常出现的领域*：应用，系统

*重构方案*：为将来扩展做出规划，选择适当的数据体系架构

*重构方案类型*：过程，角色，体系架构

*根本原因*：草率，功能扩张

*不平衡的外力因素*：要求尽快交付市场，由个人/角色承担职责，功能规划不当

*奇谈怪论*："我们的数据备份副本太多了。""我需要这个 bean 里的数据，可是数据却在那个 bean 里。"

### 背景

对于一个简单的 J2EE 解决方案，通常最容易的实现就是将有关数据与处理该数据的代码放在一个位置上。作为一种极端做法，这可能意味着要把数据放在静态变量、本地文件或实体 bean 中。一旦数据被本地化，再想消除本地化就不那么容易了。这会使你的企业解决方案有“先天疾病”，在扩展时会存在固有的限制。

### 一般形式

“本地化数据”反模式随处可见，只要企业解决方案中的一个节点存储它自己的数据，就意味存在着本地化数据。将数据置于本地存储，这并不一定是件坏事。不过，当本地存储的数据需要在别处用到时，就会出现问题。例如，可以想象一下，你在构建一个用来接受用户订单的 Web 网站。开始时你的客户群很小，所以你采用一组 servlet 编写解决方案，这些 servlet 将顾客数据存储在文件中。这个设计看上去可能如图 1-1 所示。

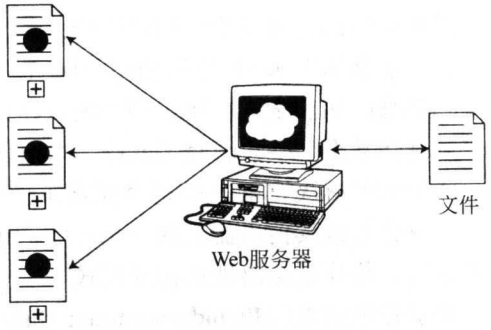



图 1-1 早期的顾客订单网站

现在，假设你的客户群规模有所增大，或者你可能向外部客户开放了一个原先属于内部的 Web 应用。无论是哪一种情况，与原先的数据流量相比，现在的流量已经不可同日而语，也许已经不是你的 Web 服务器所能处理的了。所以，你可能会采用最显然的做法，那就是购买更多的 Web 服务器。

这么一来，就会有问题了，如图 1-2 所示。你的所有顾客数据都存储在文件中，这些文件都放在第一个 Web 服务器上。尽管其他服务器想得到这些数据，但是又无能为力；这些数据被本地化存储在第一个 Web 服务器上。

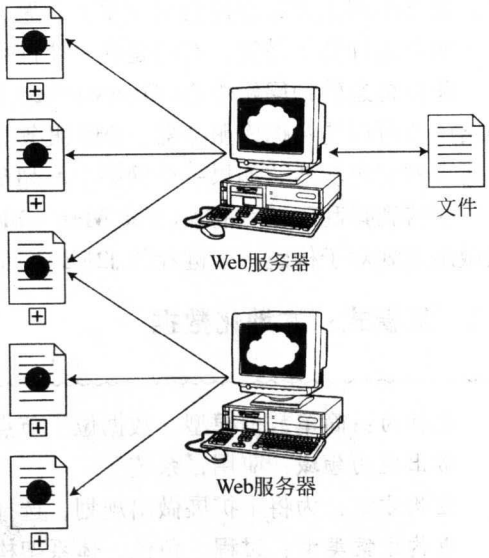



图 1-2 所有数据都是本地的(第一个服务器有数据，第二个服务器没有)

### 症状及后果

本地数据很容易发现。你可能会把数据放在内存中，或者放在一台特定机器的文件中。其结果都是一样的。你将无法得到需要的数据。

- 将数据存储在静态变量中。对于大多数 J2EE 技术来说，如 Enterprise JavaBeans(EJB)和 servlet，这都是一种不好的做法，并有相关的文挡指出要着力避免，但是在使用 JMS、RMI 或其他通信协议的定制应用中，这种情况却相当常见。
- 将数据存储在本地系统的文件中。尽管 EJB 不打算使用库文件，但 servlet 肯定会使用，

其他定制应用也存在这个问题。

- 数据只对一台机器可用。如果数据存储在实体 bean 中，那么就只能在服务器内本地使用。
- 将数据存储在一个单例对象中。这类似于使用静态变量，只不过实际上是把数据隐藏在单例对象后面了。同时，单例对象本身也可能存储在一个静态变量中，这也能够说明肯定存在问题。
- 可能违反了文档包限制。例如，你明确地表示不打算在 EJB 中使用文件应用编程接口（application programming interface，API），但服务器可能不允许你这么做。这样当你对服务器升级时可能会造成你的解决方案无法正常运作。

**典型原因**

如果解决方案是随时间日益累积而来，而非从头构建，通常就会导致“本地化数据”反模式。

- 解决方案是随时间发展而来的。通常，构建企业解决方案的第一个版本时，往往是在一个较小的环境中建立原型并加以实现，而并非针对最终要部署这个方案的更大环境。因此，本地化数据在开发环境中并不是问题。
- 本地化数据易于访问。使用本地化数据往往是最容易的解决方案。如果没有一个企业级建构人员的监管，单个的开发人员很可能会在解决方案上取捷径。长远来看这有可能会带来负面影响，但开发人员未曾考虑这一点。
- 缺少经验。如果开发人员刚刚接触企业解决方案和大规模部署，很有可能以前从未遇到过本地化数据的问题。他们可能认为自己已经封装了数据，这样就很好地应用了面向对象技术，却没有意识到，这里还需要更大规模领域中的一些面向对象概念。

**已知的例外情况**

本地化数据并不一定总是错误的。有时，基于本地文件系统，你所构建的解决方案也能很好地工作，单服务器 Web 网站就是如此。或者是使用内存中的静态变量缓存，你的应用程序也能很好运行。根本的问题在于，是否有必要扩展数据的访问范围。如果数据只用于一个本地缓存，甚至是一个共享位置上的本地数据缓存，那么完全可以保留本地数据，这不会带来问题。但是如果数据将成为一个应用程序的一部分，而且有可能扩展，那就得对你的解决方案重新考虑了。

**重构方案**

一旦存在着本地化数据，惟一要做的就是消除这些本地化数据。这意味着要选择一种新的数据体系架构，并实现此架构。应用于这种反模式的特定重构方案包括：“为将来扩展做出规划”和“选择适当的数据体系架构”，这两种重构方案都将在这一章中介绍。

为了从根本上避免这个问题，应该从一开始就考虑你的解决方案如何针对整个企业进行扩展。你只需考虑这样一个问题“我的设计能够扩展到多大规模?”，如果答案是“可以扩展到足够大，而且如果还需要更大，我有足够的财力重新编写”，倘若如此，你的解决方案就已经够

好的了。

**其他变种**

你可能决定为应用程序增加其他层，以此来改变数据体系架构。要为解决方案增加新的层次，具体形式可能是数据库、servlet、JMS 服务器和其他可以将功能分解到多个组件的中间件。增加层实际上就是从功能角度改变数据体系架构。通过改变数据体系架构，就有可能改善解决方案的可扩展性。

**示例**

可以改变你的数据体系架构，以此来修正“本地化数据”反模式。作为一个示例，再来看前面将一个 Web 应用移到多 Web 服务器安装环境的例子。起先，所有顾客数据都在一个 Web 服务器上，并存储在文件中。尽管可以复制这些文件，但是如图 1-3 所示，如果一个服务器与另一个服务器处理同样的顾客，最终数据就有可能出现不一致。而在尚未对数据文件进行同步操作的情况下，两个位于不同服务器上的 servlet 先对顾客数据完成了互斥的操作，则有可能完全破坏你的数据。

你可能认识到确实存在着同步问题，于是决定共享这些文件，并完成文件加锁。采用这种设计，如图 1-4 所示，每个 Web 服务器都通过文件共享处理相同的文件。为了避免冲突，可以对文件加锁，通过这种机制来保护文件不会同时受到多方的访问。

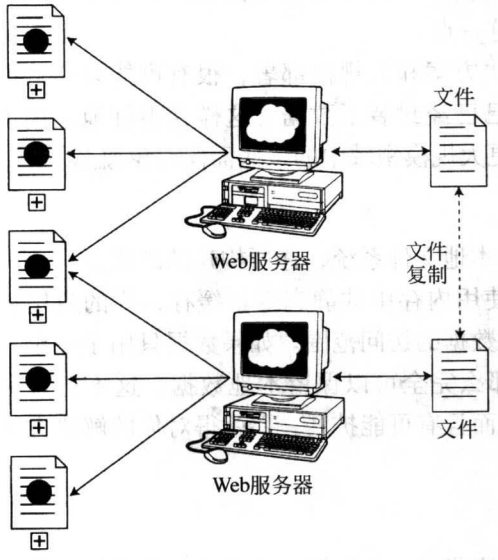



图 1-3 同步文件

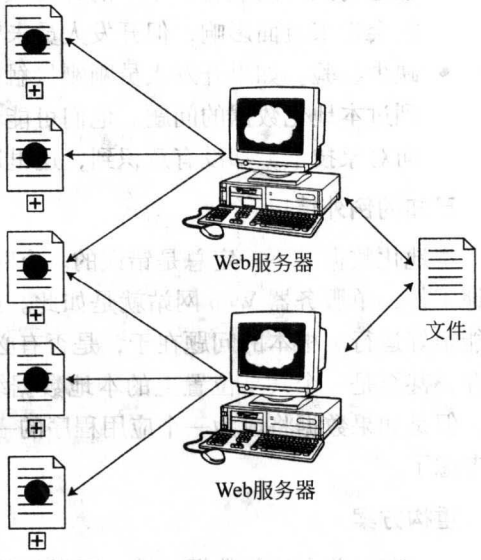



图 1-4 共享文件

最后这一步展示了改变数据体系架构的基本思想：你要把数据从单一的一个服务器移至一个共享位置。如果你对关系数据库不陌生，可能会注意到，文件共享和加锁的思想正好与使用关系数据库的目的类似。因此，解决方案中下一步就是转向一个正式的数据库，如图 1-5 所示。数据库不仅会引入加锁机制，而且还能为解决方案增加事务支持。