开 发 大 师 系 列

PEARSON
Addison Wesley

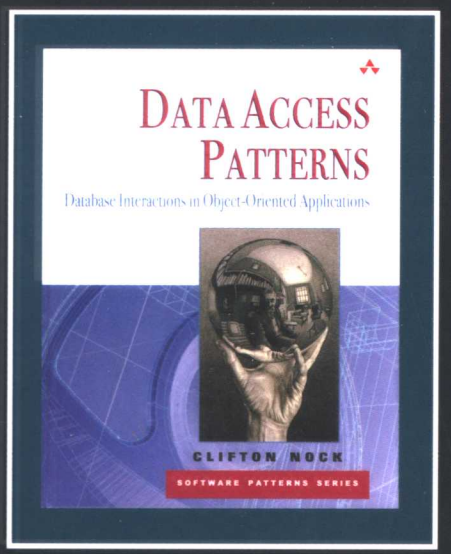
Data Access Patterns

Database Interactions in Object-Oriented Applications

数据访问模式

面向对象应用中的数据库交互

[美] Clifton Nock 著
鄢爱兰 王安鹏 等译



- 25 种改进数据访问和应用程序性能的模式
- 每一种模式均有详尽 Java/JDBC 代码和 UML 图
- 适合软件开发人员、架构师和设计师使用

Pearson Education

中国电力出版社
www.infopower.com.cn

开 发 大 师 系 列

Data Access Patterns

Database Interactions in Object-Oriented Applications

数据访问模式

面向对象应用中的数据库交互

[美] Clifton Nock 著

鄢爱兰 王安鹏 等译

中国电力出版社

www.infopower.com.cn

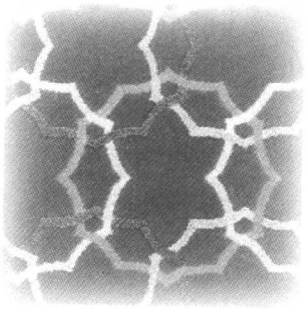
Data Access Patterns:Database Interactions in Object-Oriented Applications (ISBN 0-13-140157-2)
Clifton Nock
Copyright ©2004 Pearson Education, Inc.
Original English Language Edition Published by Addison Wesley, Inc.

Translation edition published by PEARSON EDUCATION ASIA LTD and CHINA ELECTRIC POWER PRESS, Copyright © 2004

北京市版权局著作权合同登记号 图字：01-2004-1294 号

图书在版编目（CIP）数据
数据访问模式——面向对象应用中的数据库交互 /（美）诺克著；鄢爱兰等译.
—北京：中国电力出版社，2004
（开发大师系列）
ISBN 7-5083-2195-2
Ⅰ.数... Ⅱ.①诺...②鄢... Ⅲ.数据库—程序设计 Ⅳ.TP311.13
中国版本图书馆 CIP 数据核字（2004）第 028287 号

丛 书 名：开发大师系列
书　　名：数据访问模式——面向对象应用中的数据库交互
编　　著：（美）Clifton Nock
翻　　译：鄢爱兰　王安鹏　等
责任编辑：陈维宁
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：22　　字　　数：491千字
书　　号：ISBN 7-5083-2195-2
版　　次：2004 年 6 月北京第 1 版　　2004 年 6 月第 1 次印刷
定　　价：38.00 元



译者序

设计面向对象的软件难，设计可复用的面向对象软件更难。
——摘自《设计模式：可复用面向对象软件的基础》

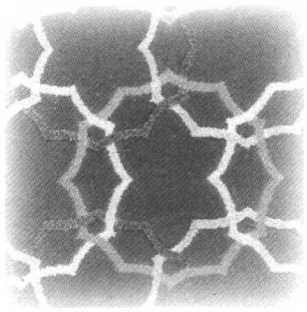
数据库是企业级应用系统的基石，即使最简单的桌面应用程序也经常要使用关系数据库支持数据持久性。数据访问代码的性能对整个系统往往有很大的影响。数据访问逻辑的复杂性以及标准的多样性，使这些代码经常成为设计中最困难的部分。即使不考虑代码复用和支持多种数据库平台，冗余和有缺陷的代码也很难避免。对此我有深刻的体会：精心设计的应用逻辑和数据访问细节纠缠成一团乱麻，调试和维护如同一场噩梦；不良的并发设计造成死锁，缓慢的数据库资源初始化也令用户喋喋不休；针对已有的数据库设计了系统，到头来用户却坚持要使用另一种产品。当我读到这本书的时候，不觉豁然开朗，原来数据访问代码应该是这样设计的！

在设计模式大行其道的今天，这是一本适逢其会的好书。原书的作者依据多年从事数据库开发的经验，抽象出了 25 种常用的关系数据库访问模式，进行了详细准确的阐述。全书由导言和五部分模式组成。导言对应用程序和中间件、软件抽象、设计模式有简短而精彩的论述，并对数据访问模式的应用提供了建议。后面的每个部分都包括一组模式的详细说明和对这组模式的概述。对每个模式都体现了一种经过千锤百炼的设计的精髓，可以在许多数据访问标准和商业化产品中发现它们的遗迹。即使不将其付诸实践，您也可以从中领略到一些杰出设计思想的精妙之处。

从架构师到编码人员，包括学习数据库技术的学生，都可以从本书受益。对照传统的数据库教科书，您会发现本书的独到之处。

本书由鄢爱兰、王安鹏主译。参与本书翻译工作的还有谢君英、王延华、马孝荣、欧阳叙好，盛海燕和谢小花录入了本书的代码并进行了初排，在此一并感谢。由于译者水平有限，难免有错漏之处，欢迎批评指正。

译 者



前言

数据是构成企业基础的主要元素。会计人员要使用商业数据作出决策；生产人员和采购人员要依靠进货和订购数据调整库存；销售人员要研究客户的历史数据；执行经理则要依靠数据研究公司的管理。

企业软件使这些关键的决策者能够阅读、编写和组织数据。业务应用程序中的数据访问功能对于其质量和可用性起着举足轻重的作用。开发人员必须花费很大的精力设计有效的数据访问代码，否则整个应用程序就可能运行得很慢或者容易存在缺陷。

数据访问模式

无论在什么样的应用领域，企业软件开发人员都要解决同样的数据访问问题。以下是设计数据访问组件时遇到的一些常见问题：

- 应用程序需要使用多种数据库产品。
- 用户界面需要隐藏晦涩的数据库语义。
- 数据库资源初始化非常慢。
- 数据访问细节使应用程序难以开发与维护。
- 应用程序需要缓存频繁访问的数据。
- 多个用户需要并发访问相同的数据。

这些问题都有通用的解决方案。一些方案非常直观，已经被成千上万的开发者独立地发现。另一些则不那么明显，因为它们已经被融合在最健壮的数据访问方案中。

数据访问模式描述了解决这类共同设计问题的一般策略。模式不一定要规定具体的实现，而是要描述一种有效的设计和结构，构成解决方案的基础。

本书描述了专门用于关系数据访问的模式。到目前为止，关系数据库是现在企业软件所使用的最流行、经过最多实践检验的数据存储机制。其他持久性技术，如面向对象数据库和层次

数据库，也正在逐渐普及。这些后备的数据库存储的数据更接近运行时的对象形式，因此更容易应用传统的面向对象模式和技术。

谁应该阅读本书？

本书是为负责构建数据访问软件组件的软件架构师、设计人员和工程师编写的。此外，本书的材料也适合希望了解常见数据访问问题和解决方案的学生。

本书使用一般的数据库和面向对象概念和术语描述模式。读者应该对这两个领域有基本的了解。如果遇到不熟悉的术语，请参考本书最后的术语表。

本书所述的模式适用于多种平台、程序设计语言和数据库。每种模式的示例代码使用 Java 2 Standard Edition（J2SE）、Java 2 Enterprise Edition（J2EE）和 Java Database Connectivity（JDBC）API 编写。示例代码使用 Structured Query Language（SQL）语言表示数据库操作。如果读者对 Java 和 JDBC 有一些了解，则对于研究这些示例代码会有帮助，但这并不是必需的。对于不那么直观的代码都给出了注释和说明。

本书是如何组织的

本书是一部模式编目，详细描述了一组数据访问模式。本书根据适用性把模式组织成几个不同的部分。因为这是一本编目，所以不需要强迫自己按照顺序阅读模式的描述。如果一个模式依赖于其他模式定义的概念，则会有明确的交待。

模式使用简洁的、描述性的、熟悉的名字标识。模式名非常重要，因为你可以在交谈和撰文中使用它们。将一组互相作用的类描述成资源修饰器的实例，与反复详细描述模式中的每个成分相比更加有效。

本书的“绪论”说明了研究和应用数据访问模式的动机，并简要介绍了每种模式。这一章还定义了以后各章描述模式细节的形式。

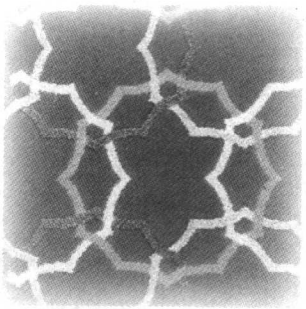
本书的其他部分就是模式编目，把每一类模式划分为一个部分：

- **第一部分，“解耦模式”** 描述了其他应用程序逻辑从解耦数据访问代码的模式，这些模式可以产生更清晰的应用程序代码，减少了仅和数据访问细节有关的修改造成缺陷的可能性。
- **第二部分，“资源模式”** 描述了有效管理数据库资源的模式。
- **第三部分，“输入/输出模式”** 描述了简化输入输出操作的模式，在以物理形式表示的关系数据和域对象表示之间使用一致的转换。
- **第四部分，“缓存模式”** 描述了实现战略性数据缓存的模式，解决数据访问优化和缓存开销之间的折衷问题。
- **第五部分，“并发模式”** 描述了实现并发策略的模式。

总结

和其他模式编目一样，本书也是不完备的。建议你调整书中的解决方案使其适应你的应用程序，并在这个过程中发现新的数据访问模式。即使你没有像本书这样正式用文档记录模式，使用和鉴别它们也是有好处的。

关于本书所述的模式，如果你有什么意见或见解，我都真诚地欢迎。你可以写信请Addison-Wesley转交给我，也可以发送电子邮件至dataaccesspatterns@awl.com。



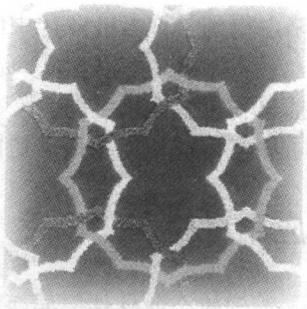
致谢

首先要感谢我的导师和朋友 Paul Monday，他鼓励我编写这本书，并在写作的过程中不断给我建议。Paul 的技术专长和干劲对于我和其他许多人来说是无价的财富。他的思想对于本书的内容和最后的成型有很大的影响。

我向 Bill Giovinazzo、Neil Harrison、Ralph Johnson、John Neidhart、Clark Scholten、John Vlissides 和 Rebecca Wirfs-Brock 表示真诚的谢意，他们为本书的技术内容提出了颇有创见的、建设性的建议。

我还要真诚地感谢我现在和以前的同事。一些人直接影响了我个人对本书所述概念的理解，一些人则改变了我的职业道路，使我能够自由地研究和学习，他们是：Axel Allgeier、John Broich、Mike Gorman、Gray Grieshaber、Maneesh Gupta、Wray Madsen、David Misheski、Bob Nelson、Lynn Nelson、Scott O'Bryan、Schuman Shao、David Wall 和 Bill Wiktor。

本书写作过程中的大多数时间，我六岁的儿子 Dylan 与我分享同一个“办公室”。他用了数量惊人的打印纸，精通了订书机装订的学问。他还自己写了几本书，使我有时间撰写这本书。Dylan、Liam、Ethan 还有 Angie，你们给了我无条件的容忍、鼓励和关心。我有幸和你们分享我的生活和工作。



绪论

基本上每个企业应用程序都有稳定的数据支持。对于用户而言，应用程序的数据常常比应用程序本身有更大的价值。许多系统的主要资源都用于实现和管理数据访问细节和逻辑。因此，详细了解数据结构和应用程序与数据结构的交互非常重要，这样就可以调整这两个方面以提高效率和可维护性。

精心设计的数据模型是有效数据访问最好的基础。规范化的数据库以及正确定义的索引可以产生更有效的查询和更新。通常，程序员和设计人员并不能自由地修改或者重新定义已有的数据模型。修改数据模型的某个方面，从一个软件版本到下一个版本，可能需要复杂的表转换过程，在升级过程中中断企业的计算环境。此外，客户可能还使用这些数据创建了自己定制的应用程序或者报表。改变数据模型将迫使客户同时更新这些系统。这可能最终阻止客户进行升级。

优化数据访问代码是一种更可行的方法，因为实现的变化只要求安装新的软件。在整个企业系统中，和数据访问有关的操作很容易成为代价最高的部分。幼稚的或者低效率的数据访问代码造成了流畅、透明的吞吐量和低响应效率之间的差异。因此在研究重要的优化时，数据访问代码是一个很好的目标。

需要考虑的数据访问策略涉及到的另一个领域是维护。分布在多个组件中的数据访问代码难以修改和维护，更不用说理解了。最好把数据访问代码封装在少量组件中，这样维护起来就会更加容易。

同样的数据访问问题往往出现在各种不同的应用领域中。财务、人力资源管理、库存以及客户跟踪软件可能有完全不同的数据模型和用户界面，但仍然会遇到一般的数据访问瓶颈，如连接开销、多余的查询和并发问题。

当相同的问题出现在不同类型的应用程序中时，开发人员也会使用通用的策略解决这些问题。比如，许多应用程序引入连接池机制以减少连接初始化的开销。随着这种方式日渐普及和推广，开发人员利用它改进了越来越多的软件产品。最终，连接池成为一般的数据访问优化策

略，得到了许多商业化中间件产品的支持，并进行了标准定义。

资源池（见第 6 章）是一个数据访问模式的例子，它是通过对多种不同的连接池实现进行抽象和重组而确定的。像本书中的多数模式一样，它适用于多种应用程序域和平台。本章说明了如何识别和应用数据访问模式，并介绍本书后面所述的数据访问模式。

应用程序和中间件

创建企业软件需要两种不同类型的专门技术。首先，开发人员必须非常熟悉应用程序域。应用程序域包括精确建模业务概念和过程的对象和逻辑。建立有效的业务应用程序需要了解涉及什么样的数据、数据之间的关系，以及什么水平的用户将使用它。显然，如果应用程序不是由领域内的专家设计的，就将非常痛苦，因为用户将不得不使用不自然的范式。

开发企业软件需要的第二类专门技术，是编写有效的、透明的中间件的能力。“中间件”一词指的是应用程序代码和系统之间的连接管道。常见的中间件包括系统资源库、预制的图形用户界面（GUI）组件和数据库驱动程序等。此外，也可以把所编写的、作为应用程序和这些库之间管道的任何代码看作中间件。编写有效的、可维护的中间件代码需要丰富的数据库资源和库知识。

这两种专门技术在一些方面存在差异。具备应用程序领域专长的开发人员一般通过生产实践获得他们的技术，而中间件专家则从专业培训和文档中学习。很难找到同时具备两方面技术的程序员。多数中间件程序员不能涉及真正能够销售的业务应用程序。相反，许多业务应用程序开发人员也不会编写有效的中间件代码。

应用程序和中间件的另一个区别是可销售性。从应用程序开发公司的角度看，应用程序能带来收益，而中间件则不能。客户根据他们实现的业务过程购买应用程序，因此领域特性是应用程序可以销售的基础。另一方面，业务客户不那么关心中间件的特性，只要足够快、足够健壮就行。因此，与中间件相比，应用程序供应商对应用程序的开发投资更多。

最后，应用程序所涉及业务的功能性都是独特的。应用程序的特性往往集中在特定于垂直行业的处理，因此，各应用领域的代码明显不同。对比之下，中间件代码对于所有应用领域通常可以是一致的，因为所有的应用程序都需要类似的中间件功能。

随着中间件开发人员从不同的应用程序中取得越来越多的经验，他们理解了那些通用的功能组件，并将它们添加到了自己的技巧箱中。某些技术几乎适用于各种应用领域。开发人员不断地改进和重用这些技术，最终抽象成可重用的代码或设计。这样就对每个项目降低了中间件的成本，提高了开发效率。随着中间件代码开发成本降低，软件公司可以增加对应用程序开发的投入，从而带来更高的收益。

软件抽象

刚入门的程序员往往用特定的方案解决具体的问题。编写的代码完全针对客户和需求的直

接要求。随着项目的推进，程序员积累了更多的经验，他们领会到如何将通用的代码设计分解到模块中。

模块是软件抽象的形式。软件抽象是把特定的解决方案推广应用到更大范围的过程。当你抽象特定的方案时，将关注方案的不同特征并将其从系统的其他组件中分离出来[Booch 1994]。焦点的集中减少了方案的复杂性，因为它削弱了和其他对象交互的重要性。软件抽象一般会形成更有效的可以重用的设计和代码。如果使用得当，抽象和重用可以显著降低整个软件开发和维护的成本。

假设我们接到一个要求，为了提高购物车应用程序的性能而缓存账户信息。我们从一个空白的缓存对象开始，当应用程序引用新的账户时向其中增加一条记录。信息已经放在缓存中，从而提高了以后账户引用的速度。

可以在以下任何一个层次上处理这个问题：

1. 硬编码（hardwiring）是一种生硬的手段，只能解决特定的问题。修改应用程序中需要账户信息的每个细节，首先检查缓存，然后如果需要的话，再查询数据库。

2. 通过模块化（modularizing）来“一次解决，多次利用”。创建通用的子例程，检查缓存并根据需要查询数据库。应用程序代码一旦需要账户信息就可以调用这个子例程。

3. 广义化（generalizing）意味着让模块解决比特定需求所限定的更广泛的问题。对于这个例子，可以定义一般的处理任何表和缓存的缓存查找子例程，而不仅仅是账户数据。

4. 模式（pattern）解决跨越多种应用领域和编程环境的设计问题。即时缓存（第 16 章）描述了一类相似的缓存问题的解决方案。账户信息缓存是要求缓存的一个具体的例子。

上述列表构成了软件抽象的演化过程。每个抽象级别都潜在地比上一级具有更高的可重用性。但是，抽象的开发需要更广泛地理解问题领域，预测在将来的迭代过程中可能出现的其他问题和需求。

设计模式

设计模式提供了通用的、可重用的设计，在设计层次上解决问题。设计模式不定义代码，也不针对给定的编程领域。相反，它为一类相似的问题提供了经过验证和测试的解决方案。设计模式也提供了通用的术语，可以使你的设计更易于编档和理解[Gamma 1995]。

模式描述了专家从多个特殊解决方案中抽象出来的技术。识别和理解模式带来了两个最基本的好处。首先，向经验较少的设计者介绍有效的设计策略，使他们不必通过尝试和经历错误重新发现这些模式。其次，为设计思想附上了通用的名称，更易于在交谈、设计讨论和文档中使用。

设计模式不是从理论上的例子中得来的。有经验的面向对象设计人员认识到，某些对象结构和交互比其他的更容易维护和重用。就像建筑和土木工程师定义适用于多种层次的大的建筑概念一样，软件设计师搜集了一组适用于不同领域的模式。

许多经验丰富的设计者积极倡导对模式进行标识和编档。设计模式分类把相关的模式组合

到单独的参考类集中。最有影响的设计模式分类书籍是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著的《Design Patterns: Elements of Reusable Object-Oriented Software》[Gamma 1995]。其中标识并描述了解决广泛的、面向对象的设计问题的模式。还有其他一些更专门的针对特定领域的分类，如企业应用程序体系结构[Fowler 2002]、J2EE 应用程序[Alur 2001, Marinescu 2002]、业务软件[Carey 2000]、嵌入式系统[Pont 2001]和测试[Binder 1999]。

数据访问模式

正如设计模式用文档记录一般的设计问题解决方案一样，数据访问模式在数据访问领域也起着类似的作用。本书中所述的数据访问模式描述了某些解决方案的一般抽象，你可以直接在应用程序中应用。一些数据访问模式具有很广泛的适用性，许多商业化产品已经用隐含的方式实现了它们。资源池（第 6 章）和对象/关系映射（第 3 章）是其中的两个例子。已经实现了这类模式的应用程序服务器或者管道平台可以使你不必再自行建立同样的基础结构。

其他的数据访问模式不那么普遍，但仍然适用于多种应用程序领域。比如，数据缓存是一种常见的优化机制，但是在实现时必须要小心，以免和你竭力避免使用的物理数据库操作相比造成更多的开销。对此，你必须考虑具体的使用模式和所缓存数据的特点。正是这些特点使得一种更通用的产品可能要花费大量的时间来进行调整。

模式分类

本书是一本关于数据访问模式的编目书籍。每一章都完整地描述一个模式，并将相关的模式组织到不同的部分。这只是一种组织性的分类，很少考虑到应用程序以及模式自身的交互。

因为这是一本编目书籍，因此本书可以不按固定的顺序阅读。模式是彼此相关的，这些相关模式列在每个模式的“相关模式和技术”部分。但是这种联系并不是必然的。

下面简要地介绍分类的组织和各种模式。

解耦模式

解耦模式描述把数据访问组件从应用程序其他部分分离出来的策略。这种解耦增加了选择和设置底层数据模型、为系统改变整体数据访问策略的灵活性。

解耦模式的另一个重要方面是向系统其他部分公开的数据访问抽象。这种抽象必须具有充分的通用性以便公开适当层次的数据访问能力，但同时又有足够的广泛性以便能够接入替换的数据源和算法。第一部分“解耦模式”中的所有模式都是为了实现这些目标。

- 数据访问器（第 1 章）——在单个组件中封装物理数据访问细节，只公开逻辑操作。应用程序代码保留关于底层数据模型的知识，但和数据访问职能剥离开。
- 主动域对象（第 2 章）——在相关的域对象实现中封装数据模型和数据访问细节。主

动域对象使应用程序代码不必直接与数据库进行任何交互。

- 对象/关系映射（第 3 章）——在单个组件中封装域对象和关系数据之间的映射。对象/关系映射同时把应用程序代码和域对象从底层数据模型和数据访问细节中解耦出来。
- 层（第 4 章）——把处理数据访问问题的正交特性叠放在递增的抽象层次中。

资源模式

资源模式描述管理关系数据库访问中所涉及对象的策略。现今大量的关系数据库访问代码都采用了标准的调用级接口，如 Open Database Connectivity（ODBC）、Object Linking and Embedding Database（OLE DB）和 JDBC。接口标准在近几年已经成熟，多数都保留了那些众所周知的概念，像数据库连接、语句句柄和操作处理。第二部分“资源模式”中的模式解决了在使用调用级接口访问关系数据时经常会遇到的性能和语义问题。

- 资源修饰器（第 5 章）——在尽量减少对应用程序代码干扰的前提下，向已有资源动态附加其他的行为。资源修饰器允许不通过子类化或者改变实现来扩展资源的功能。
- 资源池（第 6 章）——回收资源以最小化资源初始化的开销。资源池在允许应用程序代码自由分配资源的同时有效地管理资源。
- 资源定时器（第 7 章）——自动释放不活动的资源。资源定时器缓和了无限制分配资源的应用程序或者用户的影响。
- 资源描述器（第 8 章）——把依赖于平台或数据源的行为隔离在单个组件中。资源描述器把针对平台关于特定数据库资源的特性作为一般的逻辑操作公开，使数据访问代码的主体保持对物理环境的独立性。
- 重试器（第 9 章）——自动重试在某种确定条件下预计会失败的操作。这种模式增强了数据访问操作的容错性。

输入/输出模式

第三部分“输入输出模式”，描述了通过在物理形式的关系数据和其域对象的表示之间使用一致的转换，简化数据输入和输出操作的模式。

- 选择工厂（第 10 章）——根据身份对象属性生成查询选择。
- 域对象工厂（第 11 章）——根据查询结果数据组装域对象。
- 更新工厂（第 12 章）——根据变化的域对象属性生成更新操作。
- 域对象装配器（第 13 章）——使用统一的工厂框架组装、维持和删除域对象。
- 分页迭代器（第 14 章）——有效迭代表示查询结果的域对象集合。

缓存模式

缓存模式解决的是通过在缓存区存储公用数据来减少数据访问操作次数的方法。这些模式通常缓存的是域对象而不是物理数据。这是一个重要的设计观点，因为它根据调用者的要求优

化缓存的形式。第四部分“缓存模式”中的模式，提供了通用的、有效的、可扩展的缓存结构。

- 缓存访问器（第 15 章）——从数据模型和数据访问细节中解耦缓存逻辑。
- 即时缓存（第 16 章）——延迟到应用程序请求数据时组装缓存。即时缓存适合频繁读取但又无法预测的数据。
- 装填缓存（第 17 章）——用预测的数据集显式地装填缓存区。装填缓存适用于频繁读取而又可以预测的数据。
- 缓存查找序列（第 18 章）——在缓存区中插入快捷项以优化将来查找所需要的操作个数。
- 缓存收集器（第 19 章）——清除缓存区中不能够再带来任何性能上的好处的项。
- 缓存复制器（第 20 章）——在多个缓存之间复制操作。
- 缓存统计（第 21 章）——使用一致的结构和统一的表示记录并公布缓存和缓冲池的统计信息。

并发模式

并发模式解决当多个用户发出涉及相同数据的并发数据库操作时怎么做的问题。多数数据库都包括锁定机制以帮助解决这类问题，但是定制的、应用程序级的解决方案可以针对特定的应用程序语义进行调整并向最终用户提供更好的反馈信息。第五部分“并发模式”中的模式，描述了以保持数据完整性作为主要动机，用于健壮的并发数据访问的策略。

- 事务（第 22 章）——以原子性、一致性、隔离性和持续性的方式执行并发工作单元。几乎所有数据库平台都支持本地的事务。
- 乐观锁定（第 23 章）——维护版本信息以防止不当的数据库更新。乐观的锁定在更新数据库前使用应用程序语义验证工作副本的版本。
- 悲观锁定（第 24 章）——向数据附加锁定信息以防止不当的数据库更新。与类似的本地事务支持相比，明确的悲观锁定常常能够提供更好的应用程序诊断信息。
- 补偿事务（第 25 章）——定义显式的校正操作，回滚不属于本地数据库事务的工作单元。

模式的形式

本书中的每个模式章节都采用统一的形式。这种形式用于规范每个模式分析的不同方面。每一章都分成以下各节，每一节描述一个特定的模式特征。

模式名

模式的名称作为最容易识别的标识符。你可以在交谈和设计文档中引用模式名说明所采用的模式，而不必更多地详细解释。本书约定使用简短、无歧义的名词短语命名模式。作为一本快速参考工具书籍，每当在其他章节提到一个模式时，后面都有相应的章节号放在括号中，比

如：乐观锁定（第 23 章）。

简述

“简述”部分是一个句子或者很短的段落，简洁地说明该数据访问模式所提供的解决方案。可以利用模式的简述快速判断一个模式和具体设计问题的相关性。

背景

工程师并不是从零开始识别出模式的。模式的首创者通过多次解决同一问题，最终得到了一个通用的解决方案。模式的背景介绍的是该模式所解决的问题的特点或问题的类型。这一节也包括解决这些问题的例子。

适用性

模式的适用性是一个条件列表，表明何时将该模式集成到设计中最有价值。不要把这些陈述看成是必要条件，这只是一般的原则。

结构

模式的静态结构使用一个或多个统一建模语言（Unified Modeling Language，UML）类图加以说明。图中的接口、类和关系表示模式在一般意义上的基本概念。绝对不要认为必须完全使用这一节定义的实体名称和关系。这里有意使用一般的名称是为了给后续的讨论打下基础。预计在应用于具体的问题时，你会自己定义这些实体。

对某些模式的结构性实体而言，以下术语组成了一个小小的命名约定：

- 客户（Client）——使用模式实现的应用程序或者调用者。
- 数据（Data）——存储在数据库中的物理形式的数据。
- 域对象（Domain Object）——针对域的数据的对象表示。
- 访问器（Accessor）——在模式中起封装数据访问作用的对象。
- 基（Base）——接口的一般的、基本的实现。
- 具体（(Concrete）——接口的特定的、具体的实现。
- 工厂（Factory）——主要职能是创建或决定实现实例的对象。

“结构”一节也描述了模式背景中每个参与实体的作用。

交互

这一节描述模式参与者之间的交互。这些交互基本上都是定义模式如何解决一个问题。有些情况下，交互使用一个或多个 UML 顺序图加以说明。顺序图说明参与者在履行各自的职责时彼此之间的操作流。其他情况下，如果文字比顺序图更简洁，这一节就只使用文字描述。

效果

对模式的所做结论，描述模式对整个系统或应用程序的正面影响和负面影响。这一节包括一个或多个以下几种类型的结论：

- 优点——模式的优点是它对整个系统的正面影响。通常首先列举优点，同时伴随着对模式的描述，并说明优先使用该模式的原因。
- 缺点——模式的缺点描述它对整个系统的消极作用。并不是所有的缺点都会出现在模式的每个应用中，但是在应用模式时要记住这些情况。
- 折衷——模式的折衷描述对平衡互斥的优点和不利因素的考虑。

策略

这一节描述实现模式的实用技术。它分析实现细节中不那么明显的一些方面，为克服常见的问题提供建议。

示例代码

“示例代码”一节的例子通常是“背景”一节所举例子的实现。这一部分并没有完整的应用程序，而是提供一些代码块，集中说明模式的关键特性。

本书中的所有例子都使用了 Java，其中多数是用 JDBC 和 SQL 编写的。JDBC 是 Java 代码的标准关系数据访问接口，SQL 是最通用的表示关系数据库操作的语言。本书中的模式当然并不仅限于 Java、JDBC 和 SQL。但是这些技术被人们所熟知，使用它们可以使例子简洁。我尽力保持示例代码编写得简单清晰，读者不需要理解 Java 的语法细节和语义也能流畅阅读示例代码。

示例代码有意省略了必要但意义不大的代码细节，如 import 语句和连接的统一资源定位符（URL）。这种省略是为了简化起见，使读者把精力集中到和模式有关的实现方面。

相关模式和技术

每个数据访问模式都在某些方面和其他模式、标准或产品有关。这一节交叉引用这种关系。模式之间的关系包括：

- 使用——如果一个模式建立在另一个模式的基础上，则称它使用另一个模式。比如，域对象装配器（第 13 章）依赖于选择工厂（第 10 章）、域对象工厂（第 11 章）和更新工厂（第 12 章）的实现。少数情况下，使用关系会扩展到不同的层次，这一节只说明直接的联系。
- 实例化——如果一个模式细化另一个模式的结构，则称其为另一个模式的实例。比如，资源修饰器（第 5 章）是[Gamma 1995]中所述 Decorator 模式的实例，因为它定义了后者的一个应用，专门适用于数据库资源。
- 替代——如果模式的解决方案可以互换并得到类似的效果，则称彼此可以替代。要求

缓存（第 16 章）和装填缓存（第 17 章）是替代的例子，适用于同一类缓存问题。

- 协作——如果模式能够一起应用形成全面的解决方案，则称模式可以协作。比如，缓存收集器（第 19 章）可以和缓存访问器（第 15 章）协作实现健壮的缓存解决方案。

应用数据访问模式

阅读和理解模式可以开阔你的思路，在你自己的软件设计中找到使用模式的地方。应用数据访问模式需要一些技巧。最重要的技巧是在遇到适用的问题时能够确定适当的模式。有时候，问题明显地和模式的描述完全相符。其他情况下则需要一定的创造性，只有这样才会注意到模式必须以某种方式具体化，而不是直接套用书中所述的例子。几乎同样重要的是，要能够判定在什么时候所选择的模式不适用于给定的设计问题。使用不适当的模式可能会困扰你的设计，迫使你建立笨拙的结构而危及你的设计目标。

实践和积极地尝试与抛弃解决方案是获得这些技巧的最佳途径。以下是应用模式时要考虑的一些带有普遍性的问题：

- 熟悉模式——阅读模式的描述，理解每个模式提供了什么。尽管仅有这些信息还不能应用一个模式，但是，对模式的基本了解有助于在适用的设计和优化问题出现时，提醒你更详尽地研究某个模式。
- 在设计和代码的文档中引用模式——编纂模式文档的一大好处是能够定义可重用的术语。文档化的模式命名了一般的、经过证实的思路。在设计和编码的文档中引用使用到的所有模式，可以帮助读者在更广泛的层次上理解你的设计，这样你就不用详细解释你的设计了。
- 根据应用领域改变实体名称——把模式中所述的类、接口和操作的名称看成是占位符而不是最终使用的名称。当应用模式时，用符合应用领域的名称代替这些实体名。
- 随意对待模式——模式描述了广泛适用的抽象设计思路。如果模式能够解决某个问题，但不能直接适用于现有的设计，那么就修改模式。即使使用模式的一部分，或者修改模式的结构或交互仍然能为你的应用程序带来好处，那么也值得将它写入文档。最重要的是保持设计的清晰性，而不是严格地遵循一个模式。
- 不要限制你的设计——模式并非软件设计的万能药。模式编目收集并记录了相关的模式，但绝对不能囊括所有的解决方案。如果对你的设计有利就使用模式，但不要强迫你的设计完全使用模式。
- 创建自己的模式——当把模式实际应用于设计和优化问题时，也可以开始识别你自己的抽象解决方案。花点时间以你选择的形式把这些方案作为模式记录下来，并在你的设计文档中引用它们。你可以在一个地方描述你的模式，然后在其他地方引用它。因为只在一个地方对模式进行描述，所以你可能更愿意花费时间详细解释其中的概念。