TURING 图灵程序设计丛书

Pattern-Oriented Software Architecture Volume 3
Patterns for Resource Management

面向模式的软件架构
资源管理模式

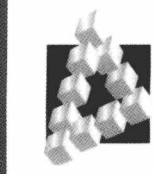
卷3

- 模式领域经典著作
- 深入剖析资源管理模式
- 高级软件开发人员必读

[德] Michael Kircher
[印] Prashant Jain 著
袁国忠 译

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

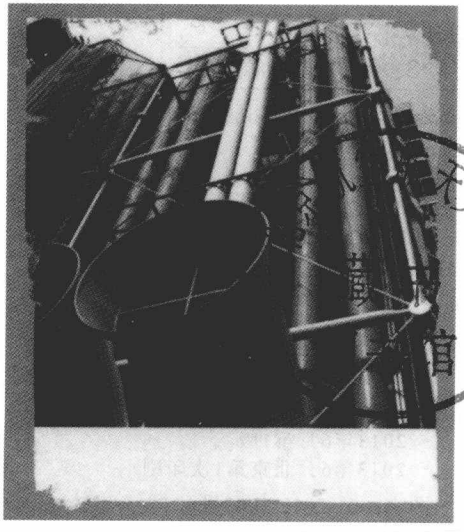


Pattern-Oriented Software Architecture Volume 3

Patterns for Resource Management

面向模式的软件架构

资源管理模式



卷3

[德] Michael Kircher
[印] Prashant Jain 著

袁国忠 译

人民邮电出版社
北京

图书在版编目（CIP）数据

面向模式的软件架构. 第3卷, 资源管理模式 / (德) 基歇尔 (Kircher, M.), (印) 耆那 (Jain, P.) 著 ; 袁国忠译. -- 北京 : 人民邮电出版社, 2013.6
(图灵程序设计丛书)
书名原文: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management
ISBN 978-7-115-31343-0

Ⅰ. ①面… Ⅱ. ①基… ②耆… ③袁… Ⅲ. ①软件设计②资源管理（电子计算机） Ⅳ. ①TP311.5②TP316

中国版本图书馆CIP数据核字(2013)第063851号

内 容 提 要

作为面向模式的软件架构系列丛书的第 3 卷，本书不仅详尽地阐述了资源管理模式，而且通过几个示例演示了如何将其付诸应用。本书包含两部分，第一部分从问题领域的角度探讨资源管理，简要地介绍了资源管理和资源管理模式，阐释了资源获取、资源生命周期和资源释放这 3 类模式。第二部分从应用领域的角度进行探讨，从案例研究的角度阐明了这些模式的实际应用。

本书适合软件架构师、设计师和开发人员阅读，对计算机专业的学生也会大有裨益。

图灵程序设计丛书

面向模式的软件架构 卷3：资源管理模式

◆ 著 [德] Michael Kircher [印] Prashant Jain
译 袁国忠
责任编辑 卢秀丽

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：12.25
字数：286千字 2013年6月第1版
印数：1-3 500册 2013年6月北京第1次印刷

著作权合同登记号 图字：01-2012-1947号

ISBN 978-7-115-31343-0

定价：49.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

献　　词

谨以此书献给 Christa、我的父母及祖父母。

——Michael Kircher

谨以此书献给 Ruchi、Aanya 及我的父母。

——Prashant Jain

Frank Buschmann 序

POSA 系列丛书的最新一卷出版了，作者署名中竟然找不到原来“五人组”（Party of Five）的身影，这是我做梦都没有想到的，但我为此感到非常自豪。我自豪，是因为新一代聪慧的软件工程师成熟了，正积极地将其经验贡献给模式界；我自豪，是因为本书表明模式的理念依旧繁荣；我自豪，是因为我们于 1996 年提出的 POSA 愿景依然在激励着模式作者以其为基础并进行改进、完善和发展。

本卷的主题是资源管理，这个主题几乎对所有软件系统的成败都至关重要。这一点看似显而易见，但仅仅在 10 年前，很多开发人员都认为只有嵌入式系统领域才需要关心资源管理。在桌面和企业领域，关注这个主题的开发人员屈指可数。为何要关心内存等资源呢？如果内存不够用，将计算机升级就是了。必须承认，在职业生涯的早期我也曾持类似的观点：资源有的是，取之不尽，用之不竭。这样的想法真是大错特错！所幸的是，我搬起石头砸了自己的脚，很快吸取了经验教训。

当前，几乎没有开发人员对资源管理的重要性置若罔闻。随着基于组件的系统和应用程序服务器横空出世以及计算机运行的应用程序日益庞大而复杂，大家认识到妥善地管理资源对软件系统的质量影响极大。即便是在最庞大的服务器上，内存、CPU 功率、线程及连接等资源也有限，更何况人们期望这些服务器向所有用户提供高品质的服务——即便有大量用户同时访问。要解决这种矛盾，必须明确而妥善地管理服务器的资源。然而，“资源”并不限于内存和连接等低层的东西，在当今的网络计算世界，资源还包括客户端应用程序远程使用的组件和服务。多个客户端应用程序争相访问组件和服务的情况司空见惯，确保所有客户都得到满意的服务是到位的资源管理的题中应有之义。

然而，认识到资源管理的重要性是一码事，妥善地管理资源是另一码事。卓有成效地管理资源既困难又极具挑战性。如果做好了，应用程序将高效、稳定、可扩展、可预测且易于使用；如果做得不好，应用程序在最好的情况下也只能提供有限的运行质量，而在最糟的情况下可能根本不可行——就这么简单。将资源管理职责交给容器并非屡试不爽，因为很多软件系统承受不了这样的基础设施。即便可以使用容器来管理资源，你也得明白容器是如何管理资源的，这样才能打造出高品质的系统。很多使用容器的应用程序都失败了，只因开发人员对容器认识不足。

如何获取这方面的认识呢？资源管理面临哪些挑战？对于这些挑战，有何应对之策呢？在应

对策略中，该做什么、不该做什么呢？在哪里能找到这方面的资料呢？你手里捧着的 POSA 系列丛书第 3 卷就提供了这样的资料。它囊括了作者多年积累的资源管理经验和解决方案，其品质已被无数著名应用程序和中间件所证明。它以模式的方式将这些经验和解决方案记录下来，让每个软件开发人员都能够加以利用。通过阅读本书，新手可了解资源管理关注的基础问题及其解决方案，专家可交叉检查和评估替代解决方案，还可了解特定解决方案的细节。据我所知，还没有任何资源管理文献的详尽程度可与本书媲美。

正如在开头所说，我为本书感到自豪，个中缘由你阅读后就会知道。

——Frank Buschmann 于西门子技术研究院

Steve Vinoski 序

如果从事软件开发工作的时间足够长，你肯定有过所谓的“忆往昔编程岁月稠”的经历。开发人员在社交场所聚集（如共进午餐）时，经常会发生这样的事情。一切都是在不经意间开始的，某位开发人员讲到他最近遇到的一个相当棘手的问题。另一位开发人员不甘人后，马上插话，一五一十地说道他是如何征服另一个更棘手的问题的。然后，每个故事都力图比前一个更精彩，最后只剩下老鸟们谈论最古老的计算机，这种计算机只有几字节内存，编程时使用的是打孔卡片或拨钮开关。期待有那么一天，有人在“忆往昔”时力图让我相信，他最初涉足编程时，只能使用0，而没有1。

开发人员之所以能够像上面描述的那样相互攀比，是因为编程天生就要求做大量取舍：应用程序之间必须共享计算资源；内存和磁盘空间都有限；CPU每秒能处理的指令数是固定的；磁盘和设备I/O可能需要相对较长的时间；建立数据库连接和网络连接可能需要很长时间，还可能占用大量资源。在电子计算的整个历史中，硬件、操作系统、中间件和应用程序都取得了巨大进步。遗憾的是，即便如此，在很大程度上来说，编程依然是一种做出正确取舍、让整个计算解决方案尽可能卓有成效的艺术。

所有应用程序都需要管理某种形式的资源，如内存、网络连接、数据库连接或线程，但是，有的应用程序在编写和调优后，能妥善管理资源，有的却不能够，二者之间有着显著差异。对只是偶尔短暂运行的简单应用程序（如基本命令行工具或配置GUI）来说，忽视资源管理不是什么问题。但如果开发的应用程序将长时间运行，且必须健壮、高性能、可扩展（如Web服务器、应用程序服务器和通知系统），那么不重视资源管理必将导致失败。

本书介绍资源管理模式。一般而言，资源需要获取、管理和释放，本书介绍了专门针对这3个方面的模式。这3个主要方面可能对运行的应用程序的整体性能、规模、可扩展性和使用寿命有深远影响。例如，很多应用程序都从堆中获取内存，对Web服务器或应用程序服务器等应用程序来说，在处理请求的代码中每次从堆中分配内存时，都将进一步降低服务器的性能和可扩展性，因为这需要调用堆管理器，还需获取和释放互斥锁以防堆被并行访问，而这些操作的开销不菲。在这样的应用程序中，可使用Partial Acquisition模式，在执行请求处理代码前提早获取尽可能多的资源。还可使用Caching或Pooling模式，将资源留下来供下次请求使用，而不是先将资源释放，以后再重新获取。通过正确地结合使用多个资源管理模式，可极大地改善应用程序的性

能和可扩展性，其成效之显著，即便是经验丰富的开发人员也会感到惊讶。

本书承袭了 POSA 系列丛书的传统，强调实用性的解决方案。介绍每个模式时，都详细描述了实现问题，并详尽地列出了模式的已知应用，这是我特别喜欢的一点。另外，本书用两章的篇幅提供了详尽的案例研究，这些案例不仅演示了如何在实际应用程序中使用模式，还说明了这些模式之间的关系。归根结底，模式描述了在实际应用程序中行之有效的方法，且模式作者将模式与其来源和影响有效地紧密联系在了一起，从而确保它们始终与现实世界紧密相连。

一般而言，软件模式可帮助我们决定在架构和设计中做出哪些取舍以及在什么地方取舍。编程工作毕竟要靠人来做，而模式可提高抽象及沟通的程度，有助于团队实现架构和设计的社会化。在漫长的职业生涯中，我曾任中间件架构师、设计师和实现人员，不时地成功使用了本书介绍的各个模式。遗憾的是，在我这样做时，还没有人把这些方法总结成模式，这意味着我与同事需要设计我们自己的变种、实现它们、通过测量确定取舍、根据测量结果进行改进和调整，这花费了大量的时间。现在，Michael 和 Prashant 以模式语言的方式清晰而全面地记录了这些方法，你可以更轻松地与同事进行探讨，进而明白它们有何优点、最适合用于什么样的情形，以及实现它们时需要考虑哪些作用力。

学习了这些资源管理模式的知识，掌握了优秀的性能测量工具，你就能快速而轻松地将性能和可扩展性平平的应用程序变成个中翘楚，连自己都大吃一惊。具备这些技能后，在下次“忆往昔”时，你的故事定将给老鸟们留下深刻印象。

——IONA 科技公司产品创新部首席工程师 Steve Vinoski

关于本书

本书介绍软件系统中的资源管理模式。在试图给软件系统提供高效而有力的资源管理方式时，软件架构师和开发人员会遇到一些常见的问题，模式给出了这些问题的解决方案。在所有软件的执行过程中，高效地管理资源都至关重要。从移动设备中的嵌入式软件到大型企业服务器中的软件，为让系统正确而卓有成效地运行，高效地管理内存、线程、文件和网络连接等资源都很重要。

面向模式的软件架构（Pattern-Oriented Software Architecture，POSA）系列丛书第 1 卷[POSA1]介绍了各种通用的软件设计和架构模式，范围很广。第 2 卷[POSA2]缩小了范围，重点介绍开发复杂的并发和联网软件系统和应用程序时涉及的基本模式。这一卷将从模式的角度出发，阐述在系统中高效地管理资源的技巧。

本书详尽地阐述了资源管理模式，并通过几个示例演示了如何将其付诸应用。与 POSA 前两卷一样，本书向读者指明了模式的实现方式。同时，详细地介绍了资源管理，并通过两个案例研究，演示了如何将资源管理模式应用于两个不同的领域。虽然本书在介绍模式时，提供的示例是使用 Java 和 C++编写的，但这些模式不依赖于任何实现技术，如.NET、Java 或 C++。本书根据资源管理的不同方面对这些模式进行了分组，它们涵盖了资源的整个生命周期：资源获取、资源生命周期和资源释放。

本书介绍的模式涵盖了资源管理领域的很多方面。几年前我们凭借打造众多不同的软件系统获得的经验，开始记录这些模式。这些模式大多都在重要的模式会议上发表过或研讨过，但我们认为，需要将这些模式组织成一种模式语言，并使其适用于多个不同的领域。

资源管理的范畴很广。系统设计师和开发人员面临的挑战是，随着新技术的出现，需要管理的资源在不断变化。我们预期，随着时间的推移，还会有人发现并记录新的资源管理模式。为完善资源管理的模式语言，还需做哪些工作呢？本书第 9 章对此进行了探讨。

目标读者

本书是为所有软件架构师、设计师和开发人员编写的。他们可利用本书介绍的模式，应对每个典型的软件系统都将面临的资源管理挑战。

本书对计算机科学专业的学生也很有帮助，因为通过阅读本书，他们将对可用的资源管理最佳实践有个大致的认识。

组织结构

本书包含两部分。第一部分简要介绍了资源管理和资源管理模式，这部分包含 3 章，分别介绍 3 类模式：*资源获取*、*资源生命周期*及*资源释放*，它们分别对应典型资源生命周期的 3 个阶段。第二部分通过两个案例研究将这些模式付诸应用。

本书的第一部分从问题领域的角度探讨资源管理，而第二部分从应用领域的角度探讨。本书介绍的模式都不是孤立的，事实上，在探讨资源管理模式的过程中，我们引用了大量其他的相关模式。对于每个这样的模式，“引用的模式”都对其做了简要介绍。

本书包含大量的模式应用示例。介绍每个模式时，列举的都是该模式的使用示例。但在每个案例研究中，都通过一个特定领域的示例阐述了如何结合使用所有模式：首先介绍特定领域面临的问题，再结合使用各种模式来解决这些问题。使用这种方法既能证明模式的适用范围广泛，又能将模式之间的关系展现出来。

第 1 章正式介绍了软件系统中的资源管理，并界定了其范畴。该章描述了在软件系统中高效地管理资源极具挑战性的原因，还简要地介绍了模式以及如何使用模式来应对资源管理方面的挑战。

第 2 章描述了如何使用模式消解影响资源获取的作用力。使用资源前必须先获取，但资源获取不应降低系统的性能，也不应导致出现任何瓶颈。对于大型资源或只有部分可用的资源，必须调整资源获取策略。

第 3 章讨论了如何使用模式消解影响资源生命周期的作用力。不同资源的生命周期可能有天壤之别，例如，有些资源用得多而频繁，而有些资源可能只使用一次。对于不再需要的资源，可将其释放，但判断资源的释放时机并非易事。显式地控制资源的生命周期既繁琐又容易出错，为解决这种问题，需要使用自动管理技巧。另外，在诸如分布式系统等架构中，多项资源必须携手合作，以实现更大的共同目标。鉴于每项资源都可能由不同的控制线程管理，要让多项资源进行协作及整合，必须进行协调。

第 4 章分析了如何使用模式有效并且高效地释放资源。对于不再需要的资源，应将其归还给资源环境，以优化系统性能，并让其他用户能够获取这些资源。然而，如果资源用户需要再次使用已释放的资源，就得重新获取，这将影响性能。挑战在于寻找恰当的平衡，确定释放资源的最佳时机。另外，显式地执行资源管理操作（如释放资源）很繁琐。如何最大限度地减轻资源管理负担，同时确保高效率和高可扩展性呢？

第 5 章展示了应用资源管理的指导方针，即提供了将资源管理模式语言卓有成效地应用于特定领域的方法。

第 6 章演示了如何打造自组网应用程序，并使用前面介绍的模式满足其资源管理需求。

第 7 章将所有模式整合成一种模式语言，并使用该模式语言满足一个电信领域案例的需求。

第 8 章作者 Frank Buschmann 承袭了以前的传统，重新审视了 POSA 前一卷就“模式将走向何方”做出的预测，进一步分析了模式的现状，并对模式的未来做出了预测。

第 9 章是本书的最后一章，对资源管理领域未来可能出现的研究动向进行了分析。

“引用的模式”简要地介绍了本书引用的所有模式。“表示法”列举了本书使用的所有表示法。

要阅读补充材料，请访问本书的配套网站 http://www.posa3.org。该网站包含本书介绍的模式的所有源代码以及对这些模式本身的更新。该网站还包含随着时间的推移在资源管理模式语言中新增的模式。

如果读者有任何看法、建设性意见或改进建议，请给我们发电子邮件，邮箱为 authors@posa3.org。

导读

本书适合从头到尾按章阅读，但如果读者目标明确，也可按自己选择的顺序阅读。在第 2 种情况下，下述提示可帮助读者确定阅读重点和阅读顺序。

- 要了解如何实际使用各个模式以及如何结合使用多个模式，可首先阅读每个模式的“问题”和“解决方案”部分，再阅读第 6 章和第 7 章的案例研究。
- 要大致了解资源管理模式语言的广泛适用范围，可阅读第 2~4 章的摘要和每个模式的“已知应用”部分。
- 要了解资源管理模式语言与现有模式成果（尤其是涉及资源管理领域的现有模式成果）之间的关系，请参阅 1.5 节。

在 1.4 节，我们列出了资源管理模式语言中的所有模式及其消解的作用力，并通过一个模式图说明了这些模式之间的关系。

致谢

很荣幸有机会感谢众多对本书编写工作给予支持的人员。首先，要感谢我们的牧羊人[①]Charles

① 牧羊人（shepherd）负责对提交给会议的模式进行初审并提出修改建议。——译者注

Weir 及本书的审稿人 Cor Baars、Frank Buschmann、Fabio Kon、Karl Pröse、Christa Schwanninger、Michael Stal、Christoph Stückjuergen、Bill Willis 和 Egon Wuchner。特别感谢硅谷模式小组（Silicon Valley Patterns Group）作出的杰出贡献，他们是 Trace Bialik、Jeffrey Miller、Russ Rufer 和 Wayne Vucenic。事实证明，利用 Wiki 通过互联网进行协作，以获取硅谷模式小组的反馈是卓有成效的。

感谢我们的 EuroPLoP 会议和 PLoP 牧羊人详尽地审阅了各个模式，他们是 Pascal Costanza、Ed Fernandez、Alejandra Garrido、Bob Hanmer、Kevlin Henney、Irfan Pyarali、Terry Terunobu、John Vlissides 和 Uwe Zdun。还要感谢本书各章的审稿人，他们是 Roland Grimminger、Kevlin Henney、Michael Klug、Douglas C. Schmidt、Peter Sommerlad 和 Markus Völter。Kevlin 还提供了如下支持：提供多章的引文，对示意图中 UML 的用法进行审阅，就如何利用版面空白提出建议。

还要感谢 Kirthika Parameswaran 与我们携手合作，一起使用 C++实现了类似于 Jini[Sun04c] 的框架 JinACE[KiJa04]。这项工作激励我们更深入地挖掘了自组网概念，并最终发现了它背后的模式。感谢她通过长长的电子邮件与我们交流，进行着一次又一次的脑力激荡。

感谢 Douglas C. Schmidt 鼓励我们从模式语言的角度审视我们对 JinACE 所做的研究。随后浮现出的模式激励着我们研究资源管理模式语言。

另外，感谢西门子技术研究院模式小组的全体成员，他们是 Martin Botzler、Frank Buschmann、Michael Stal、Karl Pröse、Christa Schwanninger、Dietmar Schütz 和 Egon Wuchner。

感谢 John Wiley & Sons 出版社与我们对口联络的人员提供的大力支持，他们是 Gaynor Redvers-Mutton、Juliet Booker 和 Jonathan Shipley。感谢文字编辑 Steve Rickaby 出色地润色了本书内容，与他合作令人愉快。

最后，要特别感谢 Frank Buschmann。他不仅审阅了全书并撰写了第 8 章，还给予我们灵感和鼓励，让本书最终得以付梓。

目　录

第1章 绪 论

设计傻瓜都会用的产品时，大家常犯的一种错误是低估彻头彻尾的傻瓜的智商。[①]

——Douglas Adams（1952—2001）

资源是一种实体，其供应有限，因此存在请求方和机制，其中请求方为资源用户，需要使用该实体来执行特定功能，而机制为资源提供方，应请求提供该实体。在软件系统中，资源可以是内存、同步原语、文件句柄、网络连接、安全令牌、数据库会话、本地服务和分布式服务等。资源可以是应用服务器组件[VSW02]等重量级对象，也可以是文件句柄等精致的轻量级对象。

有时候，要确定资源是什么是件极具挑战性的工作。例如，在编程环境中，图像（如JPEG或GIF文件）常被称为资源，但实际上并没有定义获取和释放图像的语义。相反，这种资源由组成图像的数据构成，因此更准确的做法是，将图像占用的内存视为资源，在加载图像时获取该资源，并在不再需要图像时释放它。

对资源进行分类的方式有很多，其中最简单的方式是，将资源分为可重用的和不可重用的。通常从资源提供方那里获取可重用的资源，使用完毕后将其释放；这种资源被释放后，可被再次获取和使用。内存就是一种可重用的资源，由操作系统分配，使用完毕后归还给操作系统。其他可重用的资源包括文件句柄和线程。可重用的资源是最重要的资源种类，因为资源提供方拥有的资源通常有限，所以重用未被消耗掉的资源合乎常理。相反，不可重用的资源会被消耗掉，因此获取这样的资源后，要么不释放，要么隐式地释放。计算网格中的处理时间[Grid04]就是一种不可重用的资源，处理时间被获取并使用后就消失了，无法归还。

还可根据访问或使用方式对资源进行分类。获取后，资源要么可供多名用户同时使用，要么只能供一位用户使用。可供多名用户同时访问的资源包括服务、队列和数据库。如果可供多名用户同时访问的资源包含可修改的状态，则需要对资源访问进行同步。相反，如果可供多名用户同时访问的资源没有状态，则不需要同步。J2EE EJB[Sun04b]应用程序服务器的无状态会话bean是一种不需要同步的资源，而网络通信套接字是一种需要同步的资源。可供多名用户同时访问的资

① 该句引用自*Mostly Harmless*（Tor Books出版社，1993）。

源不要求每位用户都显式地获取它，相反，资源用户可共享资源引用，因此有一名用户获取资源引用后，其他用户就可直接使用它。

相反，如果资源只能供单个用户使用，则被称为独占式资源。一个独占式资源的例子是服务的处理时间。可将处理时间视为不可重用的独占式资源，由服务（即资源提供方）提供。资源用户可从服务那里获取处理时间，而服务本身可被视为另一项资源。获取服务意味着获取处理时间。

独占式资源可以是可重用的，也可以是不可重用的。然而，如果资源是不可重用的，则肯定是独占式的，因为它只能供一位资源用户使用。另外，如果独占式资源是可重用的，则只能依次重用它，即在不同的时间使用。

将资源分成不同类别常常不那么简单，甚至毫无意义。例如，与服务的处理时间一样，CPU处理时间也可以被认为是宝贵资源。你可能认为它是一种不可重用的独占式资源，由线程（资源用户）获取，但实际上，CPU处理时间不受任何应用程序的控制，而由操作系统控制。从应用程序的角度看，CPU处理时间由操作系统分配给线程，因此线程并不需要将它作为一种资源进行获取。

资源常常依赖于其他资源。例如，文件句柄是一种表示文件的资源，而文件本身也可被视为一种资源。另一个例子是组件提供的服务，它由线程和内存资源组成。这种资源依存关系可使用包含多个资源提供方的图形表示。

下表对资源分类进行了总结，并为每种资源提供了一个例子。

	可　重　用	不可重用
独占式	内存	处理时间
非独占式	只读式对象	—

1.1　资源管理概述

在软件系统中，资源管理指的是资源用户对资源可用性进行控制的过程。资源用户可以是任何获取、访问或释放资源的实体。资源管理涵盖下面几个方面：确保资源在需要时可用；确保资源的生命周期是确定的；确保资源及时得到释放，以免影响使用资源的系统的响应速度。

管理资源很难，要有效地管理资源更难。软件的非功能性需求（如性能、可扩展性、灵活性、稳定性、安全性和服务质量）常常严重依赖于有效的资源管理。这些非功能性需求是影响软件设计和实现方式的作用力（force）[①]。虽然开发系统时可分别考虑这些作用力，但需要在多个作用

① 软件模式是从建筑学借鉴而来的，同时也借鉴了一些相关的术语。在软件领域，作用力指的是解决问题时需要考虑的各个方面，包括需求、约束条件以及解决方案的特征等。——译者注

力之间找到平衡，这极具挑战性。要在这些作用力之间找到平衡，需要解决多个问题。例如，确保系统性能比确保它灵活且易于维护更重要吗？类似地，系统响应时间的可预测性比可扩展性更重要吗？初次访问服务所需的时间比平均访问时间更重要吗？

如果要同时考虑多个作用力，将面临极大的挑战，因为解决一个问题的同时，通常必须在系统的其他方面做出妥协。例如，灵活性的改善常常以降低系统性能为代价。同样，优化特殊用例（如初次访问服务）常常会增加复杂度，并增加处理普通用例的延迟。要解决彼此冲突的作用力，将面临巨大挑战，这与有效的资源管理关系紧密。要解决上述大部分作用力，都将涉及资源的获取、访问和管理方式。

设计高效管理资源的系统时，需要考虑的主要作用力如下。

- 性能　性能至关重要的系统必须具备众多特征，其中包括延迟低和吞吐量高。每项操作通常都涉及众多资源，因此避免不必要的资源获取、释放和访问至关重要，因为这些都会带来处理开销和延迟。
- 可扩展性　复杂的大型系统通常有很多需要多次访问资源的资源用户。在很多情况下，设计系统时都考虑了用例，如确定预期的用户数量。随着时间的推移，常常会增加新的需求，进而需要扩展这些用例。例如，新需求可能要求支持更多的用户和更高的传输容量，同时最大限度地降低它们对系统性能的影响。如果系统能够满足这些需求，就是可扩展的。为实现这个目标，管理资源及其生命周期的方式至关重要。除同步开销外，获取和释放资源占用的CPU周期最多[SMFG01]。除向上扩展外，还必须考虑向下扩展，这点很重要。向下扩展指的是为大型系统设计和开发的软件必须适用于较小的系统，使用较少的资源就能运行。
- 可预测性　可预测性指的是系统的行为符合预期。在有实时需求的系统中，每项操作的最长响应时间都必须是可预测的。要实现可预测性，系统必须审慎管理其资源。进行优化时，不能为改善性能或可扩展性而牺牲服务的可预测性。
- 灵活性　系统的一种常见需求是易于配置和定制。这意味着可在编译、初始化或运行阶段修改系统的配置，真正灵活的系统给用户留下了这种自由。因此，在资源管理方面，获取和释放资源的机制以及控制资源生命周期的机制必须灵活，同时满足性能、可靠性和可扩展性方面的需求。
- 稳定性　软件系统的一项重要需求是，频繁的资源获取和释放不应导致系统不稳定。分配和管理资源的方式应确保资源得到有效利用，并避免因资源不足导致系统不稳的情况发生。如果多项资源需要彼此交互或作为一个整体被某项操作使用，则必须避免因其中一项或多项资源出现故障而导致系统处于不一致的状态。
- 一致性　所有的资源获取、访问和释放都必须确保软件系统处于一致的状态。具体地说，必须妥善管理资源之间的相互依存关系，确保系统的一致性。