# 1 Why "J2EE Without EJB"

# 第 1 章 为什么要"J2EE Without EJB"

传统的 J2EE 架构方案得到的结果常常无法让人满意：过于复杂的应用程序、令人失望的性能、难以测试、开发和维护成本高昂。

事情原本不必这样的。对于绝大多数应用程序，原本应该有更好的选择。在本书中，我们将向读者介绍一种更加简单、而又不失强大的架构方案。这种方案有作者多年的 J2EE 经验作为支撑，并且使用了诸如控制反转（Inversion of Control，IoC）和 AOP 等较新的技术。它用更加轻量级、更加灵活的基础设施取代了 EJB，并因此受益良多。本书作者和其他很多人已经在很多真实应用中使用了这种架构方案，并且得到了比传统架构更理想的结果。

下面，我们就来简单看看相关的主题。在后面的章节中，我们还将深入讨论这些主题。

## 聚光灯下的 EJB

和绝大多数同行一样，当我第一次看到 EJB 时，它所许诺的美景令我激动不已。那时，我深信它就是企业中间件的不二法门。可是，时至今日，我的观点早已发生了变化，而促成这种变化的正是我本人和众多同行的亲身经验。

时移世异，自从 EJB 规范成型以来，很多事情已经变了样：

- EJB 规范中的一些部分已经过时。譬如说，J2SE 1.3 引入的动态代理（dynamic proxy）就直接对 EJB 采用的容器代码生成机制提出了置疑。拥有动态代理之后，真的还有必要为每个 EJB 实现好几个源文件吗？
- EJB 和 RMI 之间那种传统的紧密关系也开始显得有些不合时宜，这一方面是因为 web services 的迅速发展，另一方面是因为人们发现：很多时候 EJB 只需要本地接口。在大多数时候，EJB 扮演着一种重量级的对象模型，而这样的对象模型是不需要提供远程访问能力的。

- EJB 最善于实现业务对象分布的体系结构，然而这种体系结构究竟有多大程度的普遍性，如今看来是相当值得怀疑的。
- 从人们使用 EJB 的情况也可以看出 EJB 的优缺点。大多数开发者和架构师仅仅使用无状态 session bean（SLSB）和（如果需要异步调用的话）message-driven bean（MDB）。EJB 容器为支持 SLSB 而提供的服务相当简单，这也就是说，对于这些应用程序来说，整个 EJB 容器的高昂成本很难说是合理的。
- 尽管 EJB 已经存在了五年之久、并且在很多 J2EE 项目中得到应用，但是很显然，由于它的复杂性，很多开发者仍然没有真正理解它。譬如说，我所面试过的很多开发者都无法正确地说出 EJB 容器如何处理异常，自然也就更不清楚容器异常处理与事务管理之间的关系了。
- 为了解决 EJB 存在的问题，EJB 规范也在变得日益复杂。如今，EJB 规范是如此繁复冗长，几乎没有哪个开发者或者架构师有时间去通读它，更不用说是理解它了。就像应用程序一样，如果技术规范变得日益复杂，并且需要不断地加入一些权宜之计，通常就说明存在一些根本性的问题。
- EJB 是如此复杂，这也就意味着使用 EJB 的开发效率会相对较低。有不少工具力图解决这个问题，从“企业”IDE 到 XDoclet 和其他代码生成工具，不一而足。但这些工具只能把复杂性暂时掩盖起来，而且采用这些工具也会增加开发成本。
- 严格的单元测试和**测试驱动开发**（Test Driven Development，TDD）正在日益变得流行——这也是情理之中的。人们清楚地看到：大量使用 EJB 的应用程序很难测试。用**测试先行**（test first）的方法开发 EJB 应用需要做很多工作，因此，应当从根本上尽量减少应用代码对 EJB 容器的依赖。
- 对于 EJB 致力解决的中间件问题，**面向方面的程序设计**（Aspect Oriented Programming，AOP）的飞速发展为我们指出了一条更为强大——并且很可能更为简单——的解决之道。从某种意义上，AOP 可以看作 EJB 核心概念的更为广泛的应用，然而 AOP 还远不止是 EJB 潜在的替代品，它还有更多的潜力可挖。
- 在大多数时候，源代码级的元数据属性（就像.NET 中所使用的那样）可以非常优雅地取代基于 XML 的部署描述符——从 EJB 1.1 起，我们就一直在跟这些冗长的 XML 文件周旋。EJB 3.0 似乎已经开始朝着这个方向努力了，但它背负着如此沉重的历史包袱，需要走的路还很长。

经验还表明，EJB 往往招致更高的开发成本，并且提供的利益也并不像它最初所鼓吹的那么丰富。在使用 EJB 的过程中，开发者们遇到了很多棘手的问题。根据我们的经验，EJB 的失败之处主要有以下几点：

- EJB 并非降低复杂度所必须的，它倒是引入了很多新的复杂度。
- 作为一种持久化机制，entity bean 的尝试是彻底失败的。
- 与其他 J2EE 技术（例如 servlet）相比，使用 EJB 的应用程序更加缺乏不同应用服务器之间的可移植性。
- 尽管 EJB 承诺提供高度可伸缩性，然而 EJB 系统的性能常常不够理想，而且 EJB 也并不是获得可伸缩性所必须的。尽管很难得到准确的统计数据，但实际经验告诉我们：有相当一部分项目正是因为过度使用 EJB 而不得不重新进行架构设计，甚至是彻底失败。
- EJB 可能让简单的事情变得困难。譬如说，Singleton 设计模式（或者与之类似的创建型模式）就很难在 EJB 环境下实现。

所有这些问题都表明：在着手使用 EJB 之前，有必要细致分析它究竟能提供哪些价值。我希望本书能为你提供必要的工具，帮助你冷静而有效地完成这项分析。

在第 5 章里，我们将深入讨论 EJB 及其存在的问题。现在，我们先来看看 J2EE 的现状和发展趋势，以及本书能为你提供哪些帮助。

# J2EE 还剩什么

也许你要问了：“没了 EJB，J2EE 还剩什么呢？”

答案是：还有很多很多。J2EE 远不止是 EJB 而已。然而，很多 J2EE 开发者并不这样想，要是让他们看见你案头上摆着这本书，他们恐怕要颇有微词了。不过，如果平心静气地分析一下 EJB 做了什么、J2EE 又做了什么，我们就会发现：EJB 其实仅仅是 J2EE 的一小部分而已，整个 J2EE 的图景比起 EJB 要大得多、也重要得多。

从本质上来说，J2EE 就是一大堆标准化的企业级服务——例如命名和目录服务（JNDI）、为异构的事务性资源提供的标准事务管理接口（JTS 和 JTA）、连接遗留系统的标准机制（JCA）、资源池、线程管理等的集合体。J2EE 真正的威力在于这些服务，它对这些服务的标准化对于整个行业非常有益。

另一方面，EJB 使开发者能够通过一个特定的组件模型使用这些有价值的服务，它仅仅是使用这些服务的手段之一。即便没有 EJB，我们照样能够使用 JNDI、JTA、JCA、资源池……以及其他所有的 J2EE 服务。我们可以编写代码直接使用它们（这并不像乍看上去那么可怕），也可以借助久经考验的库和框架来使用它们——后者是更好的做法，因为这些库和框架不仅可以让我们摆脱使用 J2EE 服务的复杂度，而且也不会招致 EJB 所带来的复杂度。

在 EJB 容器所提供的服务之中，只有很少的几样是 EJB 独有的。而且即便对于这些 EJB 独有的服务，也有很好的替代品。譬如说：

- entity bean 是 J2EE 唯一的数据访问组件，同时也是 J2EE 最饱受非议的部分。有很多非 J2EE 的产品可以很好地取代 entity bean，例如 Hibernate 和 JDO。在某些应用中，JDBC 会是更好的选择。
- **容器管理的事务**（Container Managed Transaction，CMT）：在 J2EE 的版图中，EJB 是唯一享受声明性事务管理待遇的。这是一项极有价值的服务。不过在第 8 章和第 9 章中读者将会看到：借助 AOP，我们同样可以获得声明性事务管理的能力。CMT 是架设在 J2EE JTA 服务之上的一个薄层。要想替换应用服务器的全局事务管理机制是非常困难（而且非常不明智）的，但在这个机制上开发一个 CMT 的替代品可就容易多了。
- **业务对象的线程池缓存**（thread pooling）：如果只提供 web 界面（以及通过 servlet 引擎支持 web services 客户端）的话，我们通常不需要这项服务，因为 web 容器已经提供了线程池缓存，没必要在业务对象层再提供一次。只有在要通过 RMI/IIOP 为远程客户端应用提供支持时，我们才会需要线程池缓存，此时 EJB 方不失为一种良好而简单的技术选择。

- （与前一点相关）**业务对象的线程管理**：EJB 容器提供了线程管理的能力，开发者可以将 EJB 看作单线程组件来实现。照我的经验，这是高估了无状态服务对象——最有用的一种 EJB——的价值。说到底，EJB 尽管可以把问题掩盖在 EJB façade 之下，但它终归无法回避所有与线程相关的复杂性。实际上，正如读者将在第 12 章看到的，有比 EJB 更好的线程管理方案可供选择。

只有在提供远程调用（remoting）这件事上，EJB 是标准 J2EE 架构中唯一的实现方式。不过，正如我们将要看到的，只有在 RMI/IIOP 远程调用的领域里，EJB 才算得上一种出色的实现技术；对于 web services 远程调用，有比 EJB 更好的选择。

越来越多的人开始认识到：EJB 试图解决太多的问题，其中很多问题本不应该由它来解决的。就拿 O/R 映射来说吧，这是一个复杂的问题，EJB 则提供了一个复杂而糟糕的解决方案（entity bean）。一些核心问题在 entity bean 这里直接遭到了忽视，譬如“如何把带有继承体系的对象映射到关系数据库”——entity bean 根本就不允许继承。要是 EJB 规范的设计者们能把这些问题留给在对象持久化领域更有经验的人，而不是自己硬着头皮解决，那该有多好。

> EJB 不是 J2EE 的全部。即便使用没有 EJB 的 J2EE，我们也无须重新发明轮子——我们不必重新实现 J2EE 已经提供的服务，只是改变使用它们的方式而已。

# 站在十字路口的 J2EE

在 J2EE 的整个发展历程中，现在正是一个至关重要的时刻。从很多方面来说，J2EE 都是一个伟大的成功：它成功地在从前没有标准的地方建立了标准；它大大提升了企业级软件的开放程度；并且它得到了整个行业和开发者的广泛认可。

另一方面，我感觉 J2EE 在一些方面已经开始捉襟见肘。J2EE 应用开发的成本通常很高。J2EE 应用项目至少和从前的非 J2EE 项目一样容易失败——如果不是更容易失败的话。这样的失败率高得让人难以接受。在这样的失败率之下，软件开发几乎变成了碰运气。而在 J2EE 遭遇失败的场景中，EJB 通常都扮演着重要的角色。

J2EE 在易用性方面存在着严重的问题。正如我在前面说过的，J2EE 应用往往很复杂，而它们原本可以不必这么复杂的。对于 J2EE web 应用（例如 Sun Java Pet Store），情况尤为明显：很多 web 应用都遭遇了毫无必要的过度工程。

J2EE 仍然是一门相当年轻的技术，它不够完美也是情理之中的。现在，我们应该认真考量它究竟在哪些地方使用、在哪些地方不那么适用，然后才能扬长避短。由于 J2EE 涵盖了很多东西，这也就意味着我们需要找出 J2EE 中的哪些部分提供了最多的价值、哪些部分只是补充性的基础设施。

J2EE 社群不断地向着更简单的解决方案、更少使用 EJB 的方向发展。我的前一本书 *Expert One-on-One J2EE Design and Development*（2002 年）就是朝着这个方向迈出的坚实一步，本书则在定义和推广这些解决方案的道路上又迈出了一步。请注意，致力于推广这些解决方案的并非我孤身一人。Rickard Oberg 和 Jon Tirsen（Nanning AOP 框架的作者）等 J2EE 技术社群的先锋已经向人们揭示了基于 AOP 的解决方案是多么强大而简洁。从 *Core J2EE Patterns*

第二版的修订中我们不难看出，就连 Sun 也无法在这一波浪潮中免疫：书中已经开始提倡使用普通 Java 对象。

J2EE 和 EJB 的很多问题都源自它们“以规范为驱动”的本质。历史告诉我们：最成功的标准都是从实践中发展出来的，而不是由哪个委员会创造出来的。OMG 和 CORBA 的例子已经让我们看到了“规范先行”的危险性：OMG 成立的目标就是要创造一个分布式对象的标准，在超过 300 家企业的参与之下，它历尽千辛万苦才制订出一个复杂的规范。和大多数委员会制订的规范一样，开发者的易用性几乎从来没有被考虑过，于是我们得到了一个极度晦涩难用的编程模型。自然，这样的规范也不可能得到广泛的接受。

在某种意义上，J2EE 是现有中间件技术的发展演化，因为它所解决的很多问题正是我们在 1990 后期就已经耳熟能详的——那时 J2EE 尚处于构思阶段。譬如说，无状态 session bean 实际上照搬了一种久经实践检验、早已证明了自己价值的组件形式：带有声明性事务管理的服务对象。早在 EJB 1.0 规范问世之前，像微软事务服务器（Microsoft Transaction Servcer，MTS）之类的中间件就已经提供了此类组件。J2EE 也努力有所创新，然而它的创新常常是从制订新规范开始，而不是首先在真实的应用中检验这些技术，结果这样的创新大多以失败而告终。有状态 session bean 就是这样的一个例子：这是 EJB 引入的一种新的、尚未得到实践检验的组件形式。五年时间过去了，它仍然难堪重任：状态的复制带来了很多微妙棘手的问题，所以大多数架构师总是尽量避免使用有状态 session bean。

在我看来，J2EE 这种“以规范为驱动”的现状很快就会发生改变，这是一件好事。我并不认为 J2EE 陷入了一种“无政府”的混乱状态，但我同样无法相信开发者们会不假思索地采用 J2EE 规范中的每一项新特性、而不去考虑其他的替代品——尤其是来自开源社群的替代品。如果希望开发者遵循 J2EE 规范，那么 J2EE 规范首先必须是一个实用、易用的规范。

这里有一个关键问题：技术的最终使用者——应用开发者、负责开发项目的项目经理、以及最终使用这些应用程序的人——才是最值得关注的。然而，在应用开发第一线看来一目了然的事实，对于那些呆在委员会里制订规范的人们来说未必总是那么一目了然的。

# 前行的路

本书的主旨不是要质疑 EJB，而是找到一条前行的道路。在书中，我们将为读者提供架构原则、可用的代码，以及具有立竿见影之效的实作建议。

本书所推荐的道路始终针对一组成功项目的核心价值——我把它们称为“主旋律”。检验一个体系结构是否合理、判断一种实现选择是否合适，都要看它是否符合这一组主旋律。

## 主旋律

本书的中心旋律是：

- 简单
- 高产
- 面向对象为本

- 业务需求至上
- 重视经验过程
- 重视可测试性

下面，我们就来简单讨论一下这几项核心价值。

## 简单

这世界上的确有简单的问题，软件的架构和实现也应该尽量保持简单。正如我一直提到的，J2EE 项目常常遭受过度工程，这很大程度上是因为我们先入为主地认为 J2EE 应用需要那么复杂。但情况并非总是如此，在某些领域，J2EE 架构师们常常过高地估计了需求的复杂程度，例如：

- **数据库的分布**。很多应用只需要使用一个数据库，这也就意味着它们不需要 JTA、两阶段提交或者 XA 事务。此时，这些高级特性都会招致不必要的性能损失，并且提高系统的复杂度。
- **多种客户端**。应用程序需要一个 web 界面，这是理所当然的，但很多 J2EE 架构师却一厢情愿地认为应用程序还必须支持远程 Swing 客户端。“J2EE 应用理应能够支持多种客户端”的想法在不少人的脑子里已经根深蒂固了。（说实话，多亏一位审稿人在审阅我以前的书稿时指出，我才意识到：需要支持多种客户端的情况其实并不常见。这也促使我更多地在本书中描述自己曾经参与过的真实项目的情况，而不是沉湎于一些不切实际的想象。）

按照 J2EE 的正统思想，我们根本不允许客户有如此简单的需求。我们这些 J2EE 架构师凭自己的学识就知道：客户的业务早晚会变得复杂起来，那时我们提前叫客户掏钱购买的复杂架构就能派上用场了。

这种想法有两个问题：首先，是否让系统变得如此复杂不应该由作为架构师和开发者的我们来决定，因为买单的人不是我们；其次，即便系统最终变得如此复杂，我们又怎么知道一开始将它们考虑进来就能节约成本呢？说不定，等到有需求的时候再修改架构还会更节约呢。实际上，很可能永远不会出现这样的需求；即便它们真的出现了，我们也可以逢山开路遇水架桥。譬如说，最终要访问我们的应用程序的远程客户端可能是在 Windows 平台上运行的 C#或者 VB .NET 程序，要为这些程序提供远程服务，基于 EJB 的远程调用体系结构就不一定是最好的选择。XP（eXtreme Programming，极限编程）的核心教义之一就是：很多时候，越是节约成本，就越能开发出高质量的软件；不要试图预先解决所有能想到的问题。

> 我们应该尽量降低架构的复杂度，只为现实的（和合理的可预见的）需求提供支持，不要试图预先把所有的问题都考虑进去。但是，在力求简单的同时，有必要多留意架构的设计质量，以保证未来能够对其进行重构，使其能够应对更加复杂的需求。对架构的重构不像重构代码那么简单，但既然我们不希望面对新的需求时被迫修改大量代码，就必须重视架构的重构。

拥有一个具有伸缩性的简单架构是至关重要的。毕竟，我们没办法裁减 EJB 这样复杂的架构来适应简单的需求。

照我的经验，使 J2EE 项目具备架构重构能力的关键在于：

- 遵循良好的 OO 设计法则，并且始终**针对接口编程、而非针对类编程**。这是经典图书 *Design Patterns* 教给我们的基本常识，可惜人们常常忽视了这一点。
- 将 EJB 之类的技术隐藏在普通 Java 对象背后。

本书所讨论的架构方法和框架使得开发者能够更容易地实践这些原则。

## 生产率

软件的生产率是一个极其重要的问题，但 J2EE 的正统思想常常忽视了这一点。

J2EE 在提升生产率方面的记录并不漂亮。J2EE 开发者常常要用大把的时间来与 API 和复杂的部署问题周旋，而这些时间本应该用来处理业务逻辑的。和使用 CORBA 的日子比起来，J2EE 的生产率有所提高，但仍然不能令人满意。在这些被浪费的时间中，有很大部分都是与 EJB 相关的。

本书所推荐的方案有相当高的生产率，这在很大程度上是因为这些方案比较简单，省却了很多不必要的负担。

## OO

当然了，我们要 OO。不过，既然 Java 是一种相当好的 OO 语言，难道 J2EE 应用不是天生就面向对象的吗？

它们应该是，但实际上很多 J2EE 应用仅仅是“EJB 应用”或者“J2EE 应用”，而不是 OO 应用。很多常见的 J2EE 实践和模式过于轻率地背弃了面向对象的原则。

> OO 的设计比具体的技术（例如 J2EE）要更加重要。我们应该尽量避免让技术上的选择（例如 J2EE）妨碍我们使用真正的 OO 设计。

让我们来看两个例子，看看很多 J2EE 是如何背弃了 OO 的原则：

- **使用带有远程接口的 EJB，以便分布业务对象**。设计一个业务对象带有远程接口、具备分布能力的应用程序，这是对 OO 的严重破坏。出于性能的考虑，带有远程接口的组件必须提供粗粒度的接口，以避免频繁的调用。另外，它们还需要以**传输对象**或者**值对象**的形式传递输入和输出参数。的确有一些应用必须提供分布式的业务对象，但大多数应用并不需要这样做，它们最好是让业务对象呆在自己应该在的地方。使用分布式对象的问题并非 EJB 一家独有的，EJB 不是第一个分布式对象技术，也不会是最后一个。这个问题之所以总是和 EJB 联系在一起，是因为组件的分布恰好是 EJB 处理得比较好的几件事之一——也许好得过头了。
- **认为持久对象不应该包含任何行为**。很长时间以来，这一直是 J2EE 开发者不容置疑的信条，包括我本人也曾经信奉过它——自从 *Expert One-on-One J2EE Design and Development* 出版之后，我的思想发生了一些转变，这就是其中的转变之一。实际上，J2EE 开发者们之所以有这样的信念，更多的是因为 entity bean 在技术上的严重缺陷，而不是因为什么所谓的设计原则。仅仅暴露 getter 和 setter（例如，通过 getter 和 setter

暴露持久化数据）的对象不是真正的对象，一个真正的对象理应把针对自己状态的行为动作封装起来。使用 entity bean 会促使开发者把这种缺陷看作一种规范的做法，因为 entity bean 中的业务逻辑很难测试，而且不可避免地与特定的持久化策略捆绑在一起。（譬如说，如果在 entity bean 中编写了大量业务逻辑，那么很显然，为了获得足够高的性能，唯一的办法就是使用 SQL 和 JDBC 访问关系型数据，这是一种典型的需要重构的做法。）在这里，更好的方案是使用 JDO 或者 Hibernate 之类的透明持久化技术，因为它们可以让持久对象成为真正的对象，在其中放置更多的业务逻辑，而不会使这些业务逻辑对持久化细节有过多的依赖。

> 一旦你发现自己正在编写一个“不是真正对象” ——也就是说，只有一些用于暴露数据的方法——的对象，你就应该想想自己为什么要这样做、是否有更好的选择。

遵循 OO 原则自有其价值。运用合理的话，OO 可以带来非常高的代码复用率和非常优雅的设计。

在本书中，我们会一直牢记 OO 的价值。我们会努力向读者展示：如何让一个 J2EE 应用成为一个真正的面向对象应用。

## 需求至上

应用架构应该由业务需求来驱动，而不是以技术为目标，这本来应该是显而易见的事。可惜，在 J2EE 这里，情况似乎并非总是如此。J2EE 开发者们常常凭空想出一些需求，例如：

- 需要同时支持多个数据库，此前我们已经讨论过这个问题了；
- 要求能够毫无成本地移植到别的应用服务器；
- 要求能够轻松地移植到别的数据库；
- 支持多种客户端。

这些都是潜在的业务需求，但它们是不是真实的需求？这就需要具体情况具体分析。使用其他技术（例如.NET）的开发者通常并不需要操心这些空想的需求，这也就是说：J2EE 开发者们仅仅因为他们的技术选择——而不是客户的需求——就耗费了更多的精力。

J2EE 让很多在其他技术中不可能做到的事情成为了可能，这是 J2EE 的一大优势，但同时也是一个潜在的危险，因为它常常会让我们忘记：尽管我们可以做到所有这些事，但做每件事都是需要成本的。如果没有一个足够好的理由，我们不应该让客户承担所有这些成本。

## 经验过程

我的妻子是一位医生。近年来，一种名叫**循证医学**（Evidence-Based Medicine，EBM）的模式对医生们的工作产生了很大的影响。按照这种模式，治疗决策需要在很大程度上参考医学研究的证据。而在此之前，医生们通常都是根据患者的状况和不同的选择在以前的

结果来决定哪种治疗方案的。

尽管 EBM 对医学实践的影响未必全都是积极的，但类似这样的经验方法无疑值得软件开发者们学习。

我们的 IT 行业很大程度上是由时尚和激情来驱动的。我几乎每天都在听到不同的人们重复着那些他们无法验证其正确性的观点（例如“EJB 应用天生就比不使用 EJB 的应用更具可伸缩性”），或者重复着一些根本没有真实证据能够提供支持的宗教信仰（例如“.NET 不是一个可靠的企业级平台”）。而这也就意味着人们常常是凭着一种固执在架构企业级系统。总之，他们知道什么是最佳方案——然而，并没有足够的证据能够证明这一点。

对于这种情况，或许最好的办法就是“问问电脑”（这个说法来自另一位技术作家 Randy Stafford）。在耗费太多的钱和时间在我们所推荐的体系结构上面之前，我们应该始终“问问电脑”对这个体系结构究竟怎么想。

在迭代式方法中，这种做法被总结为**可执行架构**（来自 RUP）、**垂直切片**（vertical slice）或者**穿刺方案**（来自 XP）。不管叫什么名字，总之是要尽快搭建一个完整的、可执行的示例应用，以尽量减少采用采用此架构的风险。在 RUP 这里，这一过程的目标是找出风险性最大的架构问题，以最大程度地降低风险；在 XP 这里，这个过程则是由核心**用户故事**（user story）来驱动的。不管怎样，这一过程必须与实际需求紧密相关。如果采用敏捷开发流程，那么我们会很自然地创建一个垂直切片，所以就没有必要单独强调这一过程了。当垂直切片建立起来之后，我们就可以在它的基础上验证架构选择是否合理。其中重要的衡量标准包括：

- **性能**。这个架构能满足非功能性的需求吗？这很有可能成为 J2EE 应用的症结。有很多项目存在棘手的性能问题，如果没有早期的垂直切片，我们常常要到最后一分钟才能发现这些问题。
- **采用的难度**。开发者在这个架构上耗费的时间和成本是否与实现的需求成比例？用它开发出的产品的复杂度是否与需求的复杂度成比例？采用这个架构的开发过程是可重复的吗？或者需要某种魔法才能获得成功？
- **可维护性**。在这个垂直切片的基础上增加新的功能困难吗？一个新手花多少时间可以理解这个应用的架构和实现、并投入有效的开发？

非常遗憾，很多项目并没有进行这样的风险缓解。更糟糕的是，很多广受推荐的 J2EE 架构实际上是由技术规范或者厂商来推动的，它们并没有在真实的生产环境中证明自己，所以它们不值得信任。

> 不仅不要相信这些技术架构，也不要相信我们或者别的任何人。请为你的应用搭建一个垂直切片，根据你的关键需求去考察它。“问问电脑”哪个架构最适合你的需求。
>
> 你的需求也许是独一无二的，但本书中介绍的架构已经在很多项目中获得了成功，并且已经证明自己能够制造出优秀的系统。所以，不妨把它作为你的第一个考察对象。

### 可测试性

在最近几年中，**测试先行的开发**（test first development）日益流行，而且通常都能收获令人满意的结果。然而，是否能够编写有效的单元测试不仅取决于开发者为此投入的时间和精力，还取决于应用程序的高层架构。在本书中，我会反复强调：应用架构理当让开发者能够轻易地编写有效的单元测试。而这正是 EJB 最大的缺陷之一：由于对 EJB 容器依赖过重，在 EJB 中编写的业务逻辑非常难测试。

难以测试的代码通常也难以修改、难以在不同环境下复用、难以重构。可测试性是**敏捷**（agile）项目的基本要素。本书所介绍的 J2EE 架构方案很适合敏捷项目使用，我们也会常常讨论关于敏捷的话题。

在第 14 章，我们还会详细讨论关于可测试性的问题。

## 轻量级框架和容器

每个应用程序都需要一些基础设施，拒绝使用 EJB 并不意味着拒绝 EJB 所采用的基础设施解决方案。我们当然不想回到 1990 年代后期、EJB 出现之前的状况：为了开发一个复杂的 Java 企业级应用，人们必须亲手实现资源池缓存、线程管理、服务定位、数据访问层等等基础设施。

本书将向读者展示：如何利用现有的框架提供这些基础设施服务。我们相信，对于成功的 J2EE 项目，这些替代 EJB 的基础设施是不可或缺的。因此，介绍轻量级容器的功能和用法就成为了本书的核心内容之一。

回顾 2003 年，这样的“轻量级”容器如同雨后春笋般层出不穷，它们提供了管理业务对象和企业级服务的能力，而不必求助于沉重的 EJB 基础设施。这也反映出一个事实：越来越多的人正在朝着更简单、更轻量的 J2EE 解决方案努力。很多这样的框架——例如 Spring、PicoContainer 和 Nanning——都来自繁荣的 Java 开源社群。在第 5 章，我会更加详细地谈论这个话题。

除了开源产品之外，也有很多商业产品——例如 Mind Electric 公司的 GLUE web services 产品——在其他领域提供了比 EJB 更为轻量级的解决方案，例如远程调用。

### Spring框架

如果没有可用的、在真实产品中经过考验的代码来展示我们介绍的架构方案，我们就不可能写出这本书。

Spring框架（`http://www.springframework.org`）是一个流行的开源产品，它的作用是为J2EE应用常见的问题提供简单、有效的解决方案。2003 年，*Expert One-on-One J2EE Design and Development*一书不同寻常地以一个完整而有发展前途的应用框架作为示例，随后，从该书的示例代码中发展出了Spring项目。Spring有一个兴旺蓬勃的开发者和用户社群，并日益变得强大而可靠，质量远远超过任何我独自开发的东西。（本书的合作者Juergen Hoeller是Spring项目的领导者之一，他为项目社群的建设作出了无法估量的贡献。）在Spring项目中，基础框架的设计甚至早于*Expert One-on-One J2EE Design and Development*的写作，它们来自我过去几个商业项目的经验。

在构思之初，我们并没有打算让 Spring 成为 EJB 的替代品，但它确实为普通 Java 对象提供了强大、可测试的功能实现（例如声明性事务管理），让很多开发者能够在很多项目中避免使用 EJB。

Spring 并不是唯一一个这样的项目。在本书中所介绍的设计，鲜有专属于 Spring 的。譬如说，我们建议使用 AOP 来解决一些常见的企业级问题，除了 Spring 之外还有好几个别的开源 AOP 框架可供选择。

大量使用具有产品级质量的开源代码，这是本书最大的独特之处。绝大多数 J2EE 书籍都会提供一些代码示例，但很可惜，这些代码的价值非常有限，因为它们所展示的都是出于教学目的而大大简化了的问题。如果读者试图在真实应用中使用这些代码，他很快就会遇到麻烦。所以，使用一个具体的框架——即便并非所有读者都将使用这个框架——来阐释作者的观点，这种做法只适合用于介绍简单得失去真实意义的解决方案，因为惟其如此，才能把整个解决方案放进书中。

Spring 为体系结构重构提供了绝佳的支持。譬如说，你可以在应用架构中加入带有本地接口的 EJB，也可以给业务对象添加 AOP 支持，而不必修改任何一行调用代码——当然，你必须遵循基本的编程原则：针对接口编程，而不是针对类编程。

# 我们还应该使用 EJB 吗

EJB 仍然有它的位置。本书介绍了一种比 EJB 更简单、生产率更高的架构方案，它适用于很多 J2EE 应用。但是，我们不会声称这种方案是 J2EE 的万灵药。

> 在 *Expert One-on-One J2EE Design and Development* 一书中，我不止一次提到过“帕累托法则”。这个法则也常常被称为“80/20（或者 90/10）法则”，也就是说：花比较少（10%—20%）的力气就可以解决大部分（80%—90%）的问题，而要解决剩下的少部分问题则需要多得多的努力。在软件架构这里，这个法则告诉我们：架构的价值在于为常见的问题找到好的解决方案，而不是一心想要解决更复杂也更罕见的问题。

EJB 的问题就在于它违背了“帕累托法则”：为了满足少数情况下的特殊要求，它给大多数使用者强加了不必要的复杂性。比如说，也许只有 10%的应用需要分布式的业务对象，然而 EJB 的基础结构却与对象分布紧密相关。EJB 2.1 以及此前的 entity bean 被设计为与数据存储机制无关，但绝大多数 J2EE 应用都使用关系数据库，因此它们能够从 entity bean 的存储机制无关性中得到的利益微乎其微。（“各种数据存储介质之间的可移植性”在理论上很有价值，但在实际应用中就未必如此了。而且，尽管声称提供了这种可移植性，但在对象数据库这里，entity bean 的表现也无法令人满意。访问对象数据库的最佳途径仍然是使用它们自己提供的 API，或者 JDO 之类的解决方案。）

对于那些真正需要对象分布的应用，EJB 仍然是上佳之选——尤其是当它们完全用 Java 实现、或者用 IIOP 作为通信协议时。不过，这种应用比人们通常想象的要罕见得多。

对于需要大量使用异步消息的应用，EJB也是不错的解决方案，因为用message driven bean处理异步消息非常有效——而且相当简单。[1]

[1] 译者注：时至今日，EJB在处理异步消息方面的优势也受到了来自轻量级阵营的挑战。Apache组织属下的Jakarta Commons Messenger是一个轻量级的、基于POJO的JMS消息框架，使用这个框架可以——如MDB那般——轻松地发送和消费JMS消息，而它所需的运行环境仅仅是J2EE web容器。目前Commons Messenger项目仍处于沙箱（sandbox）阶段，但它（或者类似的其他产品）进入真实应用仅仅是个时间问题。读者可以在H`http://jakarta.apache.org/commons/sandbox/messenger/`H找到这个项目的相关信息。

EJB 能够真正为项目提升价值的典型例子或许就是金融中间件应用。金融应用中的处理过程常常需要耗费大量的时间和计算能力，比起业务处理本身的开销，远程调用的开销常常倒是可以忽略不计的了。而且，金融中间件也通常是面向消息的，很适合使用 MDB。我相信，这样的应用就是那真正复杂的 10%。

当然，也许还有强大的政治因素（而非技术因素）促使人们使用EJB，这已经不在本书的讨论范围之内了。照我的经验，赢得政治斗争通常比赢得技术斗争要困难得多。在这方面，你需要的老师是是马基雅维利[2]，而不是我。

> 我相信 EJB 是一种正在逐渐衰退的技术，不出三年它就会进入弥留阶段，尽管 EJB 3.0 竭力改进也于事无补。但本书关注的是帮助你马上着手搭建企业级应用，所以如果 EJB 能够很好地应对你眼下的需求，我就会建议你使用 EJB——但只是当下暂且使用而已。

# 小结

在本章中，我们大致浏览了一下本书后面章节将要详细讨论的各个主题。

从 J2EE 诞生之初，EJB 就一直被视为 J2EE 平台的核心，但我们认为这是一个误解。EJB 有它的位置，但对于大多数应用来说，不使用 EJB 会是更好的选择。J2EE 远不止是 EJB，EJB 只是使用 J2EE 服务的途径之一。因此，拒绝 EJB 决不意味着放弃 J2EE。

我们相信轻量级容器（例如 Spring 框架）能够更好地组织应用代码、更好地使用 J2EE 提供的服务。在本书中，我们将详细介绍这种架构方案。

我们相信业务需求和基本的 OO 常识——而非具体的实现技术——应该是项目和架构的驱动力。

在下一章中，我们将更详细地讨论本章所指出的核心价值，随后再进入关于应用架构和具体技术的讨论。

[2] 译者注：马基雅维利（Niccolo Machiavelli，1469~1527）是文艺复兴时期的意大利政治思想家、历史学家和军事理论家。他的政治科学名著《君主论》以不受感情和伦理约束的冷静客观态度指出了君主的治国之道，因此常常被认为是“为达目的不择手段”的政治诡道——也即所谓“马基雅维利主义”——的代表。

# 15 Performance and Scalability

# 第 15 章 性能与可伸缩性

性能与可伸缩性常常决定企业应用的成败。

遇到性能问题的 J2EE 应用的比例高得令人吃惊：往往发现问题的时候已经太晚了，无法用经济的方式得到解决，导致全面失败的风险。

并不是说 J2EE 注定就是缓慢的，而是因为很多导致系统缓慢的架构模式被不正常地广泛应用，而且太多的 J2EE 架构在设计时就有性能盲区。尽管有很多证据显示，性能方面的风险应该在项目周期中尽早解决，但是很多人仍然相信：性能问题可以在所有的功能都完善之后加以解决。他们常用的办法有：

- 代码优化，或者
- 更多、更快的硬件——这常常被视为万能灵药

这两条路都是歧路。代码优化能起的作用只不过相当于在泰坦尼克号上搬动一张躺椅而已。通常决定性能的是体系结构而非具体实现，优化代码很少能够从根本上改善性能。（然而，代码优化却是减轻后期维护压力和减少潜在 bug 的良好手段。）就算加上更多的硬件能解决吞吐量问题——常常还不能解决——这也会浪费很多金钱，因为应用程序占用的硬件资源比本该使用的更多。

对性能问题的视而不见常常来自这样一种信仰：“纯粹”的体系结构比高性能的程序更加重要。这是胡说：买单的人关心程序是否能满足他们的期望，远甚于关心程序是否遵循一张——本身就画满问号的——蓝图。

> 无法达到性能和吞吐量要求的程序是失败的。

在本章中，我们会讨论如何才能避免性能问题，检查各种可供选择的体系结构的性能，在进入全面开发之前确认性能特点。我们会关注：

- **设置清晰的性能及吞吐量目标的重要性。**
- **设计高性能 J2EE 应用时，最基础的架构方面的考量：是否使用 EJB。**核心架构对性能产生的影响通常远大于实现细节，也就是说：对架构的性能内涵的理解和测试，比局部的代码优化更重要。
- **我在本书中提出的中间件方案——围绕 IoC 和 AOP 来构建系统——与 EJB 方案比较，哪个性能更好。**
- **如何对 J2EE 应用进行性能调优。**
- **以“循证”方式进行性能分析的重要性。**因为性能非常重要，我们应该奉行“实际数据决定一切”的原则，而非受控于臆测与偏见。

本章大部分的关注将围绕于 web 应用，但是大多数的讨论对所有 J2EE 应用都是有效的。

Expert One-on-One J2EE Design and Development 的第 15 章长篇讨论了性能测试与优化，包括关于缓存和代码优化技巧的讨论。我不会再重复那些讨论。如果需要了解低层优化技巧，请参阅那本书和本章后面列出的其他参考资料。在本章中，我会更关注核心的架构问题，以及“J2EE without EJB”方案的性能内涵。

# 定义

有必要对“**性能**”（performance）、“**吞吐量**”（throughput）和“**可伸缩性**”（scalability）三个概念进行区分。我在这里使用的定义和 *Expert One-on-One J2EE Design and Development* 以及 Martin Fowler 的 *Patterns of Enterprise Application Architecture*（Addison-Wesley，2002，pp.7－9）是一致的（实际上与后者稍微有点差别）。

应用程序的**性能**是进行典型操作所需的时间。性能常常用“响应时间”的标准来衡量。性能经常是重要的业务需求之一：比如“程序必须能在 30 秒之内接受一条新的净值交易并发出确认信息。”性能也对用户体验产生重大影响，特别是在竞争性环境中的 web 应用操作。如果用户觉得程序很慢，他们会感到挫败，可能不再继续使用它。

**响应时间**（response time）是应用程序处理一个请求——比如从用户的浏览器得到的一条 HTTP 请求——所需的时间。一般，我们对平均响应时间感兴趣，在负载增大时响应时间的一贯性也很重要。提高负载后若响应时间曲线出现锯齿，往往说明性能乏善可陈，还有潜在的不稳定。

**延迟时间**（latency）是从应用程序得到反馈所需的最少时间，不管程序是否需要做很多工作才能得到这个反馈。远程方法调用具有很长的延迟：不管被调用的方法是否成功，都有一个固定的最小开销。

**吞吐量**是程序或者组件在一段给定时间内所能进行工作的总和。对 web 应用来说，常常用每秒点击率来衡量；对事务处理应用来说，则是每秒能完成的事务数。

**可伸缩性**指应用程序如何应对增长的流量。通常，更大的并发负载是由更大规模的用户群产生的。说到可伸缩性的时候，我们通常指**向上伸缩**（scaling up），以便应对更大的负载。可伸缩性经常等价于**水平可伸缩性**（horizontal scalability）：向上伸缩到服务器集群来提高吞吐量。我们也可以通过把应用转移到更强的服务器上来提高吞吐量。后者要简单得多，但显然并不能让应用更坚固，也只能得到有限的提高。另一种选择是**垂直伸缩**（vertical scaling）：在每台服务器上运行多份服务。这对 WebSphere 这样的应用服务器可能获得更好的结果，因此 IBM 提倡这么做。关于这项技巧的讨论，请参阅 www-306.ibm.com/software/webservers/appserv/doc/v40/ae/infocenter/was/07010303.html。“垂直伸缩”这个术语有时候被 Fowler 用来指“为单台服务器增加更多的计算能力”，比如添加额外的 CPU 或者内存。

性能和可伸缩性有时候在现实中是对立的。能在单台服务器上高性能运行的应用，却可能无法被部署到集群中：比如，为了获得高性能，针对每个用户在 session 中维护大量的数据；而在集群环境下，这些数据无法被高效地复制。

然而，必须注意到，性能低下的应用同样不会具有很好的可伸缩性。如果应用程序在单台服务器上浪费资源，就意味着即便在集群中运行，也只不过是浪费更多的资源。

我们应该把**负载测试**（load test）和**压力测试**（stress test）区分开。负载测试的目标是给系统以期望的负载量；而压力测试的目标则是在超过期望能力时确定系统行为。比如说，超过某个负载量之后，是简单地拒绝更多连接，还是变得不稳定？良好的负载和压力测试工具让我们可以测量响应时间（平均时间和分布）和延迟时间，以及系统吞吐量。

# 设置清晰的目标

对性能和可伸缩性设置清晰的目标是很基本的要求。目标必须比“应该快速运行”和“具有高可伸缩性”这样的用语更加清晰。一些关键性的问题是：

- **吞吐量和响应时间的目标是多少？**没有清晰的目标——目标应该是由业务而非应用程序架构来决定——就无法辨别性能是否令人满意，也不知道从性能角度而言，我们是否在体系架构和实现上作出了正确的选择。太多的项目缺乏清晰的目标了。
- **哪些操作必须很快，哪些是可以慢一些的**？并非所有的用例都是等价的。从性能角度来说，有一些用例具有更高的优先级。
- **在什么样的硬件和软件（应用服务器、数据库）条件下可以达到这些目标？**如果不具备这些信息，无法确认是否成功。

- **如果需要更多的开发和维护工作量，让程序运行得更快是否值得？**特别是在进行代码优化的时候，牺牲可维护性有时候仅能换取些微的性能提升。因此如果我们已经能够达到性能要求，再做更多的优化努力可能反而降低生产率。只有一小部分程序——通常是作为基础架构的程序——才需要拼命压榨每一丝性能潜力。
- **应用程序需要伸缩到支持多大的负荷？**和响应时间一样重要的是，应该支持多少并发用户（或并发事务）？
- **程序是否需要在集群环境中运行？**如果程序目前不需要集群，日后是否会运行于集群环境中？

通常我们做不到让每件事都全速运行，必须在不同的用例之间作一些厚此薄彼的倾斜。比如说，金融交易系统必须要能对当前情况作出风险分析，还要能重现历史场景。这两项需求可能对性能有所损害。系统可能会以“更便于以后随机访问”的结构来储存数据；也可能提升当前数据的优先级，而把历史数据以审计记录的方式保存。

对性能目标有实际的估计是很重要的。在.com 泡沫时期，每个刚起步的小公司都幻想着会成为下一个 Amazon.com。避免结构僵化无法伸缩是很重要的，但是把力气浪费在不切实际的空想规模上同样是不明智的。要保证采用的结构对投入的代价有所回报，能达到对应的可伸缩性和性能指标。

# 体系结构的选择：影响性能和可伸缩性的关键因素

对 J2EE 应用而言，所选择的基本架构对性能有本质性的影响。假若整体架构充斥着不必要的数据库访问，假若在 Java 对象和 XML 文档之间进行过度的转换，或者假若过多地进行远程调用（这是三个最常见的 J2EE 性能问题的根源），那么你针对某个方法进行单独优化不会带来多大性能提升。

从性能和可伸缩的角度来说，最重要的架构方面的选择是：

- 应用是否是分布式的?
- 如果需要在集群环境中运行，采用何种方式进行集群?
- 访问持久化数据的方式?

让我们一一讨论这些问题。前两个问题有着密切的关系。

## 对象分布、集群和农场

企业应用通常不会只在一台服务器上运行。水平伸缩（在多台对等服务器上运行）的主要动机是通过联合更多的 CPU 性能来增加吞吐量；另一个（可能更重要的）目的是为了让程序运行得更稳定，因为单机失效无关大局。

水平伸缩是件复杂的事情，有必要区分出不同类型的水平伸缩，特别是：

- **对象分布**，在这种类型中，具有远程接口的组件是主要的分布单元。应用程序构筑在分布式组件的外围。
- **针对部署的集群**，这种类型中应用程序的多份部署是分布的主要单元。当一台服务器接收到请求，它只调用其内部的组件来完成处理。这是我提倡的方式，特别是对于 web 应用而言。

我们还会看到，根据“希望在集群内复制的内容”不同，这两种类型各自还会有不同的集群风格。

### 定义

**对象分布**（object distribution）是 J2EE 的经典做法：通过针对组件（远程 EJB）而非部署来获得水平伸缩。这种模型的目标是，通过把 web 和业务对象（远程 EJB）分布到集群中的不同机器中，来获得负载平衡和可伸缩性。在这种模型中，所有的业务调用都通过 RMI/IIOP、或是 XML 和 web services 来进行。

**针对部署的集群**（deployment clustering）是把所有的组件都部署到集群中的每个节点中去。在 web 应用中，这会导致每个节点都包含“从 web 层直到数据库访问”的完整 J2EE 栈。这种模型下，在请求到达节点之前就已经进行了路由分配，而不是在单独的组件调用时进行分配。Martin Fowler 在他的 *Patterns of Enterprise Application Architecture* 一书的第 89 页强烈建议使用这种方式，并列举出它从编程模型直到性能的优点。这种风格的集群中，当控制权从一台单独的转发服务器中转到具体处理的服务器后，所有的调用都是本地的（通过引用调用）。

不管哪一种情况下，EIS 层的资源（比如数据库）都是在独立的服务器上。虽然这会牺牲部分性能，但可以获得管理上的优势。既然我们终归无法 J2EE 中间层的同一进程中运行企业级数据库，因此无论如何都无法避免进程间通讯的性能损失。

不管是对象分布还是对部署分布，很多情况下我们都需要在集群的各个节点间通讯。比如，各个节点需要复制数据缓存。稍后我会简单说明这个问题。

集群的风格也是多种多样的，比如一种特别的风格是**农场**（farm）。在一个农场中，整个应用运行在多台服务器上，但是每台服务器都不知道其他服务器的存在，除了共享资源（比如数据库）外也不需要进行通讯。也就是说，在服务器之间不进行状态复制。

## 分布式对象带来的麻烦

对象分布的关键问题是，它获得可伸缩性的方式是以大幅度牺牲性能为代价的，结果苦心积虑获得的可伸缩性大半被用来解决系统本身的性能低下了。

在第 11 章中提到过，真正的远程方法调用起来很慢。（不包括那些对位于本机的远程 EJB 进行的伪远程调用。它们速度很快的原因是因为实际上进行的并非远程调用，而是本地方法调用。）因为网络传输以及序列化、反序列化的开销，每次远程调用的代价都比本地调用高出好几个数量级。因此尽管从理论上来说我们可以拥有任意数量的远程业务对象，但比起“在集群的每台服务器上部署同一个应用”的做法来，前者需要更多的硬件才能达到同样的吞吐量。

有时候远程调用不是通过 RMI，而是通过 XML 或者 web services 协议来进行。这还是同样缓慢，而且带来了更多的复杂性。（远程 EJB 已经让对象分布变得尽可能不带来麻烦了。）

对付远程调用的高昂代价，传统的 J2EE 方法是通过批处理来尽量减少分布式调用的次数：比如，使用传输对象（transfer object）一次性传送大量的数据。每次传输的数据多，进行远程调用的次数就少，比“大量的远程调用而每次只传送很少的数据”要快，因为进行远程调用时，在底层发生的基础设施开销往往要比实际上传输数据本身的开销更大。然而，如果可能的话，如果完全取消分布式边界，性能会提高得更多。使用传输对象也会带来困难，它要求我们精确地知道每个远程客户端需要多少数据、该在持久化对象的关系图中追溯多深。

因为对象分布可以让承担大量负载的组件分布到多台服务器，所以它轻率地承诺可以从根本上解决可伸缩性问题。然而，从理论上来说，对象的分布只应该针对瓶颈部分，但实际上把整个应用都部署到多台服务器上会更好，因为这样可以减少远程调用开销，也能让目标组件获得足够的 CPU 能力。

根据我的经验，分布对象在性能上真正起到正面作用，这样的案例非常罕见。在典型的成功案例中，必定会存在一些非常消耗 CPU 时间的操作，以至于和这些操作相比远程调用的代价可以忽略不计。比如说，有些财务应用中的风险计算会需要极多的 CPU 能力，大量服务器组成的专用集群能够很好的应付它的需求。对于这种案例来说，远程 EJB 是一种良好的实现策略。但是这种案例本身就非常罕见。而且，对于需要很长时间才能完成的操作来说，通过消息队列（message queue）进行异步分布可能是比远程方法调用更好的选择。

## 集群的挑战

就算我们决定采用集群，下面的主要挑战就是如何运行一个集群。这里最大的难题包括：

- **路由。**发送到集群的请求如何被路由到具体的服务器？
- **复制。**有哪些状态（或者其他资源）需要在集群中复制共享？Session 状态是最重要的复制需求之一，另一个需求是确保在集群中使用的共享对象缓存保持一致。

第一个问题不难解决。实际上，路由问题甚至不必由软件来解决。但是，复制是集群中的主要问题和最大的限制。

实施水平伸缩时，你必须考虑如下的限制：

- **性能增长不是线性的。**在三台服务器上运行程序带来的吞吐量通常不可能是在单台服务器上运行的三倍，除非你采用农场模式。如果使用对象分布，从伪远程调用（在同一台机器中）转换到真实的远程调用会显著降低性能，可能还会增加 CPU 负担，导致降低吞吐量；如果采用集群分布，结果会好些，但也不可能做到性能的线性增长，因为需要 session 复制，可能还会需要访问数据库这类共享的资源。维护数据缓存在单台服务器上很简单，但是要维护共享数据的缓存则要复杂得多，会带来显著的运行时开销。
- **在一个集群中能运行服务器的数量通常也有极限**，因为通常有管理开销。比如说，在集群中的服务器越多，复制带来的开销就越大。虽然我们可以通过使用数据库来共享状态，从而降低复制开销，但是对小型集群来说这样性能更差。说到底，除非我们的 EIS 层资源有无穷的可伸缩性，否则我们在增加中间层的时候仍然会受其拖累。

> 高可伸缩性的应用首先必须是一个高效率的程序。因为水平伸缩很难得到线性回报，每单位的硬件若能得到更高的吞吐量，整个应用就更具可伸缩性。

一个经常会被忽略的重要问题是，EJB 常常并非是集群环境下需要考虑的最重要问题。下表中列出了当应用程序面对集群和复制问题时各个层面的问题：

| 结构层 | 问题 | 在哪里解决 | 讨论 |
|---|---|---|---|
| *Web*层 | 请求如何被路由到 web 容器？ | 硬件路由器<br>Apache 等 web 服务器的插件<br>Web 容器，例如 WebLogic 的 `HttpClusterServlet` | 路由可能按照不同的算法进行，比如轮换、随机和负载估计。<br>假若不考虑状态复制，路由本身倒是相对简单。状态复制给“选择哪个服务器来处理请求”增加了限制。 |

续表

| 结构层 | 问题 | 在哪里解决 | 讨论 |
| --- | --- | --- | --- |
| | 如何复制 HTTP session 状态，来提供错误恢复能力？ | 由 Web 容器处理。可能采取的策略有：<br>➢ 在集群内广播改变了的数据<br>➢ 把状态保存在数据库<br>把状态备份到一台独立的“备用”服务器 | 不同的方案在可靠性、性能和可伸缩性方面有不同的取舍。比如，把状态备份到数据库是一个可靠的方案，也可以支持很大的集群规模（假设数据库能够处理这么大的负载），但是会对性能造成严重的负面影响。 |
| | 对频繁显示的数据是否有缓存，来避免对业务层的重复操作？ | 通常由 web 层的代码提供缓存来处理 | “经典”的分布式 J2EE 应用通常包含 web 层缓存来尽量减少对远程 EJB 业务对象的调用。假若我们取消远程 EJB，这就变得不那么重要了，因为调用业务对象不再会大量消耗性能。 |
| 业务对象层 | 复制有状态的业务对象的状态 | EJB 容器会管理有状态 session bean 的实例 | 若能用 HTTP session 对象的形式来管理状态会更好。因为 web 层需要保持有状态 session bean 的引用，而如果我们又在业务层复制状态，通常我们就需要处理两个（而不是一个）复制问题。假若使用本地业务对象，就不会有“从 web 层为它们传递状态”的性能损耗；而对于远程 session bean 来说，使用有状态的业务对象会带来负面性能影响。 |
| | 把请求路由到无状态业务对象实例 | EJB 容器把调用路由到远程 EJB | **只在分布式架构中需要。** |

续表

| 结构层 | 问题 | 在哪里解决 | 讨论 |
| --- | --- | --- | --- |
| *O/R mapping* 层 | 在节点间维护一致的缓存 | O/R mapping 工具，比如 TopLink、Hibernate 或者 JDO 实现。<br>若使用 CMP entity bean，则由 EJB 容器负责解决。 | 要维护一个针对事务的分布式缓存是个复杂的问题。<br>O/R mapping 产品通常把问题交给第三方产品，比如 Tangosol Coherence 去解决。<br>CMP entity bean 的实现通常不比上面的方案更成熟。比如，在每次对集群中的实体操作前必须调用 `ejbLoad()`：这是巨大的管理开销 |
| 数据库 | 水平伸缩，横跨多台物理服务器的数据库仍然可以作为一个逻辑服务器提供服务 | RDBMS 厂商。Oracle 9i RAC 就是这样的产品之一。 | 数据库的这种伸缩能力是非常重要的，否则数据库可能会变成整个应用伸缩的瓶颈。 |

从上面这些“集群和复制带来的难题”中，你可以看出什么？对于不使用 entity bean、运行在单一节点的应用，EJB 容器无法解决上述的任何难题。因此，我们可以得出一个重要的结论：EJB 容器对可伸缩性没有任何贡献——既没有正面的，也没有反面的。

实际上，复制的难题全然没有涉及到业务对象层。最重要的复制服务是由以下部分提供的：

- **web 容器**，复制任何 session 状态。
- **O/R mapping 层**，它通过复制数据来确保在数据库之前有一致而连贯的缓存。
- **数据库**，本身就拥有跨越多台物理服务器的可伸缩性。

在这样的应用中，使用 EJB 但是不采用 entity bean（这也是目前 EJB 社群公认的最佳实践），我们可以使用 web 容器来管理 HTTP session 状态，使用与 JDO 结合的本地 session bean 来管理持久化。在这样的结构中，最重要的复制在两个地方发生：

- web 容器之间交换 session 状态。
- 在各个服务器节点的 JDO 二级（`PersistenceManagerFactory` 级别）缓存之间。这

种复制常常由专用的第三方缓存产品提供，比如 SolarMetric Kodo JDO 常常与 Tangosol Coherence 一起提供高效率的事务性缓存。

在这一体系结构中，本地 SLSB 间没有复制或者路由。因此 EJB 容器不在集群管理中，关键的分布式数据缓存并不是由应用服务器处理的，而是由持久化技术提供的。

通过减少在 HTTP session 对象中保存的数据，以及使用足够细化的 session 对象（而非一个庞大的对象），可以帮助提高 web 容器更好地进行状态复制。因为这样可以只复制那些被更改过了的对象，而不是每当 session 状态有所改变，就序列化整个庞大的对象。

无状态业务对象不需要复制，并且只有在为远程客户提供服务时才需要路由。其他情况下，路由选择早在业务对象被调用之前就已经完成了。

数据访问常常是水平伸缩的关键。集群的成功往往是由下面几件事情决定的：

如果有 O/R mapping 层，它在分布式环境中缓存数据的效率如何？这是大多数 entity bean 实现表现差劲的地方。

- 数据库是否能承担整体的负载？有效的数据缓存能够保护数据库，但是我们也应该记得数据库自身往往也包含有高效的缓存，特别是高端数据库——比如 Oracle 9i RAC——本身就可以通过集群部署获得高度的可伸缩性。

有些应用自己维护分布式数据缓存很难获得好处：比如，假若数据是写多过于读的，这种情况下可能直接使用 JDBC 操作数据库更合适。

> 在集群的 J2EE 应用中，解决复制问题的常常并不是 EJB 容器，而是 web 容器和数据访问层。被广泛信奉的“EJB 是集群所必需的”这一信念，其实是个谬误。

希望通过牺牲性能而获得更好的伸缩性——通过更多的远程调用——是危险的。不仅仅是我们无法获得那些失去的性能，我们还需要更大数量的服务器才能获得同样性能，因为每台服务器都在浪费资源，用于为网络中传送的数据进行打包和解包。也就是说，我们可能达到水平伸缩的极限。

> 不保存服务器端状态的应用，其可伸缩性是最佳的。比如在 web 应用中，如果我们只需要保存很少一点用户状态，我们可以把它保存在 cookie 中，就不再需要任何形式的 HTTP session 状态复制了。这样的应用是高度可伸缩的，也是极为稳定的。硬件路由设备可以高效地在服务器之间平衡负载，或者替换任何发现出问题的服务器。

## 数据访问

数据访问的方式对性能有重大影响。假设使用关系数据库，拥有下列特性是很重要的：

- 高效的数据库结构，能够快速执行常见的查询
- 数据库更新的次数要尽能少
- 每次更新都要高效，牵扯的内容越少越好
- 高效的数据缓存，如果应用程序的数据访问有责任进行缓存的话。

从可管理性和性能两方面来考虑的话，理想的数据访问是：使用一个自然而高效的RDBMS 数据结构，以及一个自然而高效的对象模型。也就是说如果采用 O/R mapping，我们就需要一个成熟的解决方案，它应该能够映射对象继承关系、能够利用目标数据库的优化能力、还能在必要的时候访问存储过程。

基于以上原因，entity bean 很少能够具有成熟的 O/R mapping 方案同样高的性能。（虽然只是一个令人遗憾的方案，但 entity bean 通常还是被用作 O/R mapping。）BMP entity bean 通常天生低效，因为存在“**n+1 查找问题**”（n+1 finder problem）。CMP entity bean 会好一点，但是性能往往也不够高：

- EJB QL 不像 SQL 等查询语言一样富有表现力。有些在 SQL 和 Hibernate HQL 这样的类 SQL 语言中能够高效进行的操作，在 EJB QL 中就无法高效进行。
- 如果需要将多个持久化对象关联起来，entity bean 的处理效率比基于 POJO 的 O/R mapping 要低得多。
- entity bean 模型必须使用事务，而这可能并不是必须的。
- 和最好的 O/R mapping 技术相比，entity bean 实现并不能做到同样高效的分布式缓存。

通过 Hibernate 或者优秀的 JDO 实现所能做到的、真实的 O/R mapping 常常很有价值，是获得一个良好的领域模型的最佳方法。然而也要记住，O/R mapping 并非到处都合适。

“ORM 到处都适合”的观念，代表的是 J2EE 架构师对关系数据库广泛的不信任。在 *Expert One-on-One J2EE Design and Development* 一书和其他场合，我不断在质疑这一点。不可理解，为何这种不信任在 J2EE 社群中占据如此重要的地位？关系数据库对有些事情处理得非常好，特别是在针对集合进行的操作上。还有，存储过程有时候能够减少对数据库的过多使用，提供高效的关系操作。

> 关系数据库不会因为 **J2EE** 社群不怎么喜欢它们就消失。早在 **J2EE** 出现之前很久，这些数据库就已经在那里，而且在可预见的将来它们还会在那里，用户已经在 **RDBMS** 上投入了大量的资金。

当然，如果可能的话，我们不希望把业务逻辑也放到数据库中。但是持久化逻辑是另一回事，关于它是否应该属于数据库的一部分还存在很大争议。比如说，如果用存储过程

来更新数据库表、从而将表的细节隐藏起来，我们就可能通过改进设计来把我们的 Java 代码和数据库结构分离开来。只需要保证存储过程具有同样的参数，我们就可以自由改变底层的数据库结构，或者迁移到另一个数据库。只要在存储过程中不进行业务操作——譬如说，不需要生成主键，不需要针对一笔新数据更新好几张表，也不需要根据条件或者角色不同来选择进行操作——这种方法就可以继续使用。

一般来说，假若我们发现某个操作需要通过 O/R mapping 对很多记录进行枚举的迭代操作，比如计算统计值，那么最好不要通过 O/R mapping 来进行这个操作，尽可能将它放到数据库内部去。不过，如果我们需要在客户端显示这些记录，就应该产生代表它们的 Java 对象实例。

## 其他体系结构方面的问题

还有一些结构性的问题可能影响性能。在实际应用中，我最常看到的三个最重要的问题分别是：XML 处理、表现层技术、以及如何选择应用服务器和其他产品的专有（proprietary）特性。

无论如何，为整个技术架构的每个部分进行性能评测是非常重要的。也就是说，必须把系统耗费的全部时间细化，分别衡量每部分占用的时间。本章的后面会讨论采样评测。

### XML 的使用

XML 现在几乎变成 Internet 的通用语了。在用于把不同技术实现的系统松散连接起来时——特别是 web service 成为事实标准后——XML 是富有价值的技术。但是，在 J2EE 应用内部采用 XML 进行通讯很不合适，虽然这种做法现在像感冒一样流行。

Java 对象和 XML 之间的相互转换代价是非常高昂的。XML 数据绑定能降低代价，但是不能消除它。

> 目前流行的 J2EE 体系结构中，“在应用程序中大量使用 XML”被认为是一种好的做法，但这种做法直接损害了系统性能。

### 表现层技术

表现层技术（以及我们使用这些技术的方式）对性能有重大影响。比如说渲染一个 JSP，可能比从数据库访问获取必须的数据花费的时间还长。一些不良习惯（比如过度使用自定义标签）也会降低性能。

使用 XSLT 有一些其他的好处，但是 XSLT 转换对性能而言也是昂贵的。如果它被用来产生表现层的结果，会导致显著的性能下降。XSLT 引擎的性能参差不齐，因此选择好的引擎至关重要。

因为表现层技术及其使用会显著影响性能，对表现层单独进行评测会很有用处。

### 可移植性与性能

有时，可移植性的目标与性能互相冲突。特别是当无法用标准的技术应对所有需求时，这个问题就显得更加突出。架构师往往以“纯净”的理由拒绝使用产品或厂商专有的扩展，但这可能是以性能为代价的。如果你采用的技术平台对某种特殊操作提供了专有的、非标准的高效方法，你就应该考虑这个操作是否有性能上的要求、是否应该采用这种非标准的技术。同时，你应该尽可能遵循良好的 OO 实践，把这些不可移植的部分隐藏在便于移植的接口后面。

# 不同实现的选择

在说到“架构”这个词的时候，我们指的是一些大问题，比如是否需要分布？是否采用 O/R mapping?完成这些选择之后，我们还要针对具体实现进行选择。

- 对于本地业务对象，采用本地 EJB 还是采用 AOP 注入操作代码后的 POJO?
- 使用 EJB 容器还是 IoC 容器？
- 使用 Hibernate 还是 JDO 实现？

本节中，我会从性能的角度来思考这些选择，也会说明为何这些选择通常不像“根本性的架构选择”那样重要。我将重点介绍：摆脱 EJB 对性能造成怎样的影响。

随后，我将带领读者考虑代码优化的问题，以及如何以尽量少的工作量和代价获得尽量大的性能提升。

## 摆脱 EJB 服务设施对性能的影响

抛弃 EJB 而采用我们在本书中所提倡的体系结构，对于性能和可伸缩性会大有帮助。这样的观点是否让你吃惊了？每当听到别人批评 EJB 在其它方面的弱点（例如复杂度）时，EJB 的鼓吹者们总是会反驳说：如果容忍那些缺陷，EJB 具有比其他竞争对手更好的可伸缩性和吞吐量潜力。难道这不是事实吗？

在本节中，我会进行一些质疑，看看到底真相如何。

请考虑下列情况：我们有一个服务接口，可以选择采用本地无状态 session bean，这样就可以利用 CMT 等 EJB 企业服务；也可以选择 POJO，它能够利用具有 AOP 能力的轻型容器（例如 Spring）提供的企业服务。作为参考比较，稍后我也会考虑真正的远程 EJB 调用的性能结果。

> 虽然声明式服务比较引人注目，但这里主要关注业务对象调用的性能问题。声明式服务的性能开销——不管是对于 **EJB** 还是其他技术——只有在大量细粒度对象（例如一个巨大结果集中返回的持久化对象）的情况下才会成为一个问题。稍后我也会讨论整个应用程序的性能潜力，并通过 **web** 层行为来客观衡量它。

## 基准评测方法

下面的讨论基于一次基准测试，被测目标是一个访问关系数据库的 web 应用。被测应用会进行下列操作，来模拟典型 web 应用常见的几种操作。

- 处理请求，这个请求需要对业务对象进行反复的调用，而业务对象则会迅速返回结果，以模拟“返回缓存数据”的情况。这一操作不需要事务。
- 建立订单、更新库存，这需要更新两张数据库表。这是通过一个 PL/SQL 存储过程进行的。
- 用户发起的订单查询。

由于这些情况不能从缓存中获得好处——基准测试也不是为了比较缓存实现——因此业务对象会借助 DAO 访问数据库，后者直接使用 JDBC。应用程序会从事先拟定的范围中随机选取用户和物品 ID 来进行“建立订单”和“订单查询”的操作。

被选择进行测试的实现包括：

- 远程部署的远程 SLSB，使用 CMT 进行事务管理
- 本地部署的 SLSB，使用 CMT——对于大多数 web 应用，这是推荐的 EJB 架构
- 全局共享单一实例的 POJO，通过 Spring AOP 引入事务功能：这是我们对大多数情况推荐的架构
- 通过 Spring AOP 基础设施进行池管理的 POJO，也用 AOP 引入事务功能
- 全局共享单一实例的 POJO，通过自编程实现事务管理，没有任何的声明式服务

所有的 POJO 都部署到一个 web 容器中，该 web 容器与 EJB 容器在同一个应用服务器中运行。所有的情况下，都使用同样的 JSP 来生成内容。底层的事务基础设施都是应用服务器的 JTA 实现。不同的实现都从同样的容器 `DataSource` 获取数据库联接。数据库每次运行后都清空。

每种情况的业务对象中的代码几乎都是同样的。 EJB 扩展了 POJO 实现，并实现 session bean 生命周期必须的方法，具体实现采用了 Decorator 模式，本章稍后会讨论这个模式。为了尽量减少 JNDI 查询的次数，用于获得 EJB 引用的服务定位器实现缓存了 `EJBHome` 对象。

不管是 EJB 还是 AOP，唯一需要的声明式中间件服务只有事务管理服务。在“模拟返回缓存数据”的情况下，无须事务管理。

负载测试模拟具有长延时的重负载，同时测试稳定性和吞吐量。下面将要列出的结果是采用一个流行的开源应用服务器测试得出的。我们也使用一个高端的商业产品进行了测

试，各项数据都有了相当显著的提高，但同样显示本地 EJB 不如 Spring 方案——但差距已经缩小了很多。

下面的基准测试运行于 Sun JDK 版本 1.4.2_03，操作系统是 RedHat Enterprise Linux 3.0。应用服务器设置使用 HotSpot Server JVM，其垃圾收集和堆栈设置已经预先设置成为服务器应用优化。应用服务器运行于双 Xeon 2.8GHz 处理器的机器，配置 2GB 内存和两个 SCSI 硬盘，并且使用 RAID 0 来优化性能。因为基准测试包括远程 SLSB，一台类似的机器被作为客户端，两台机器通过 GB 速度的电缆相连来消除网络开销。数据库（MySQL 4.0.16）也为测试配置了合适的缓存和缓冲区大小。所有的测试执行时都使用干净的数据库，之前服务器会被重启，等待一段预热时间。测试使用 ApacheBench 进行。

我们也在不同的系统上执行测试，也换过不同的数据库（把MySQL换成Oracle），操作系统则是从Red Hat换成Windows 2000/XP，结果大同小异。如果对这次基准测试有更多的兴趣，可以点击Spring网站（`www.springframework.org`）上的“Performance”链接，下载源代码和详细的结果。

感谢 **Alef Arendsen** 提供了生产级别的硬件并且采集大多数性能结果。

## 声明式中间件模型和编程式中间件模型

在讨论线程模型之前，首先我们看看事务管理的方法。

### 使用 CMT 的远程 SLSB

当 EJB 真正以远程方式运行时，除非业务操作本身就很慢（比如，需要插入到数据库），否则远程调用的总体开销对吞吐量和响应时间有巨大的负面影响。

### 使用 CMT 的本地 SLSB

本地 SLSB 的性能会好很多，吞吐量的提高引人注目。与执行业务对象所需要的代价相比较，在同一进程中执行 EJB 调用的代价就很低了。

### 用 AOP 进行事务管理

Spring AOP 方式的性能比本地 EJB 又有了显著提高——当然，就更不用说比远程 EJB 好得多了。Spring AOP 框架实现声明式服务的代价就算与本地 EJB 调用比也小得多。区别主要体现在调用“不使用声明式服务的方法”时，此时 Spring AOP 实现的开销仅是开源 EJB 容器的零头，也比商业 EJB 容器稍微少一点点。这是一个很有趣的结果，因为商业 EJB 容器使用 Java 代码生成和预编译来避免使用任何反射操作，而 Spring AOP 使用了动态代理或者 CGLIB 代理。在 Java 1.4 JVM 中，反射不再是代价高昂的操作了。

不出所料，假若被拦截的业务方法进行了昂贵的操作（比如数据库插入），**AOP** 和 **EJB** 调用的开销之间的差别就变得不那么明显。此时，不管 **AOP** 还是 **EJB** 基础设施的开销都被执行数据库操作的时间所掩盖了。然而，拦截的性能越高，就意味着越多的 **CPU** 资源可以用于 **I/O** 操作，因此 **J2EE** 服务器所占据的时间也变少了，所以仍然会得到显著的回报。也可以在类似 **Tomcat** 这样的 **web** 容器中运行这个基准测试，此

时便不使用 **JTA**，而是使用 **Spring** 为 **JDBC** 提供的 ***PlatformTransactionManager*** 和第三方连接池（例如 **Commons DBCP**）。假若有不止一个需要事务的数据源，这种方法就不能奏效，但这仍然是一个有用的选择，因为这样在任何 **web** 容器上都可以允许声明式事务管理，不需要改变编程模型。（和 **EJB** 不同，**Spring AOP** 既可以向高端伸展，也可以向低端收缩。）此时的性能有一部分取决于你选择的连接池的效率，但结果仍然可以证实：**Spring** 拦截的性能比 **EJB** 的性能更高。

### 编程实现事务管理

这里我们使用一个事务**装饰器**（decorator）子类。比如，我们覆盖业务 facade 上的 `placeOrder()`方法，在之前开始事务，在之后提交或者回滚事务，并且仍然调用超类的实现来完成必要的业务逻辑。对“返回缓存住的结果”的模拟操作来说，就不需要提供任何事务装饰器。

我们之前说过，JTA 是一个难用的 API，我们使用 Spring 事务 API 作为 JTA 之上的一个薄中间层。我们的代码是这样的：

```
public void placeOrder(long userid, Order order) throws NoSuchUserException,
NoSuchItemException, InsufficientStockException {
  TransactionStatus txStatus = txManager.getTransaction(new
DefaultTransactionDefinition());
  try {
        super.placeOrder(userid, order);
        // Leave txStatus alone
  }
  catch (DataAccessException ex) {
        txStatus.setRollbackOnly();
        throw ex;
  }
  catch (NoSuchUserException ex) {
        txStatus.setRollbackOnly();
        throw ex;
  }
  catch (NoSuchItemException ex) {
        txStatus.setRollbackOnly();
        throw ex;
  }
  catch (InsufficientStockException ex) {
        txStatus.setRollbackOnly();
        throw ex;
  } finally {
        // Close transaction
        // May have been marked for rollback
        txManager.commit(txStatus);
  }
}
```

注意我们使用了 Spring `TransactionStatus` 对象的 `setRollbackOnly()`方法，在捕获到意外的时候进行回滚。

我们使用了简化的抽象 API，因此以上的代码看上去没那么复杂。如果某个简单的应用只有一两个需要事务的方法，这个方法比设置 EJB 或者 Spring AOP 以便使用声明式事务管理还要来得简便。当然，如果很多方法中都需要这样的代码，编写这种代码就令人痛苦，犯错误的机会也更多。

> 如果用到了装饰器模式，通常暗示着可能应该采用 AOP 方法。

这次基准测试在底层使用 Spring `JtaTransactionManager` 实现。这是一个基于 JTA 的很薄的层。因此在所有的测试中，这和使用基本的事务设施没什么不同。

和 **Spring AOP** 的方法一样，**Spring** 的编程式事务管理方法也可以使用非 JTA 的底层事务（比如 **JDBC**），事务机制的转换不需要修改程序代码。

我们可能认为，自己编程会比声明式方式更快，但实际上这里只有 2%到 10%的性能提高。这个结果说明，相对于花费在自己编写事务管理的力气而言，回报实在太少。付出了程序开发和维护上的代价，获得这很少的性能提升，实在太不值得了。

> 相比而言，自己编程管理事务是一种复杂的编程模型，除非有必要的理由才能这么做。不仅仅是因为编写的代码越多就越容易有潜在的错误，而且我们的代码还会变得难于测试，因为为了测试我们的业务逻辑，必须提供事务基础设施的测试替代（stub）或者模仿（mock）对象。如果我们使用 Spring 对事务的抽象，能减少一些痛苦，因为它把 JTA 这样难于配置和替代的底层事务设施隔离开来。但是，更好的做法是：根本不编写、也就无需测试事务代码！

## 线程模型

现在来看看我在第 12 章中建议的线程模型。SLSB 实例池和下面的替代方案比较，结果会如何呢？

- 全局共享的、可以多线程执行的对象（不是像 SLSB 实例池那样，而是用一个对象应对所有的客户端）
- 对 POJO 提供透明的实例池，而不使用 EJB

下面的讨论会和前面谈到的声明式服务模型结合起来。

对非 EJB 的选择，我主要关注 Spring，因为它令程序员从业务对象的生命周期管理中脱离出来——同样，除了 Spring IoC 容器之外，这里也有 AOP 的功劳。

稍后我会把这个基准测试和事务管理的基准测试放在一起讨论。

### 共享的多线程可执行对象

第 12 章中我提出过一个论断：对于大多数无状态服务对象，“全局共享唯一实例”是最合适的线程模型。

正常情况下，由于这样的对象没有“既可读又可写”的状态，因此无须同步。如前所述，即使多线程执行的对象需要一些同步，只要同步代码能快速运行，性能也不会受到显著影响。

因为基准测试的目的是进行仿真测试，不准备包含需要同步的情况，因此我们不包括同步。

在所有的测试中，共享的多线程执行对象都比并置的 EJB 对象池具有更好的性能，不管是在开源应用服务器还是在商用服务器上。

### Spring 实例池

下面看基于 Spring 的实例池。这需要使用 Spring AOP 和 `org.springframework.aop.TargetSource` 接口的实现——在这里是基于 Commons Pool 1.1 的实现。`org.springframework.aop.target.CommonsPoolTargetSource` 类作为 Spring 的一部分发布。这提供了类似 EJB 的编程模型，业务对象在编写时可以假设为是单线程的。我们不需要修改目前的任何代码，因为业务对象中没有线程问题。我们把池的大小设置为和在 EJB 测试中 SLSB 的池大小一致。

在我们的测试中，假若 web 容器的配置是合理的，Spring 实例池要比单个共享的实例的性能差。如果 web 容器允许非常多的连接（这应该算是一种不恰当的配置），业务对象的实例池会提高性能。

在开源应用服务器上，Spring 池的性能比 EJB 实例池有非常明显的优势。在高端商业服务器中，Spring 池的性能还有一些差距，说明服务器的 EJB 池实现极为高效，击败了 Spring 的池实现所使用的 Commons Pool 1.1。（假若能够使用其它的池产品来重新运行 Spring 基准测试，应该会很有趣。）不管怎样，看上去 EJB 容器的实例池并非魔术，Spring 可以对 POJO 达到同样的效果。Spring 的池性能会有一些优势，除非 EJB 容器特别高效。

因为 Spring IoC 容器提供的解耦能力，在共享实例和实例池模型间切换的时候，客户端代码和业务实现代码都不需要任何改变。

## 结果总结

开源应用服务器和 Spring 方案的结果对比如图 15-1 和图 15-2 所示，我对“响应时间”数据做过一些处理以便于图形展示。左边的两个结果来自远程 EJB 和本地 EJB，之后两个来自 Spring 共享实例和实例池方案，在最右边是没有使用 EJB 和 Spring 声明式服务、而是自己编程管理事务的结果。

图 15-1 显示了“业务操作瞬时完成”时的性能，这种情况是在模拟返回缓存中的值。

这测试了业务 facade 中部分操作无需事务、而另一些则必须具有声明式事务管理的情形（因此，可能对所有的方法都需要某种形式的代理）。吞吐量和响应时间的差别在远程访问和最慢的本地访问之间大约是 10 倍。当然，需要两台服务器才能获得远程访问的结果，因此它需要的硬件资源两倍于其它的所有测试。

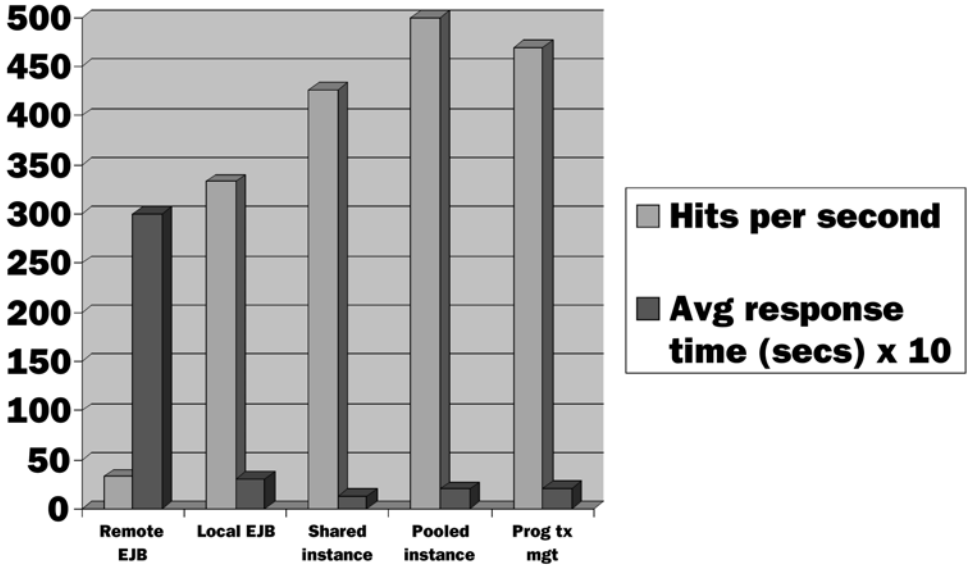



图 15-1

测试显示 **Spring AOP** 实例池获得了最佳的性能。在多个环境下运行测试，我们后来发现如果能优化 **web** 容器的线程池大小，就算在最重的负荷下，共享实例也能胜过实例池。然而，因为我们不能在前面介绍过的硬件配置条件下重新运行所有的测试，我们这里采用初始的测试结果。

图 15-2 显示了如果业务操作本身代价高昂时的性能。此时我们使用 JDBC 查询数据库（没有缓存），获得平均 50 个对象的结果集，此时远程和本地调用的差别就小多了。创建订单的结果对比情况非常相似。硬件和软件配置与前面的测试一模一样。

对于典型应用，远程调用的真实代价应该在这两者之间。远程业务对象的有些方法会反应迅速，远程调用的代价就显得很高；对于慢速的操作，远程调用的代价则似乎不是问题所在。假若机器之间使用慢速网络相连，远程调用的代价会变得非常高。（以上测试中的连接带宽是 1GB。）在所有的测试中，远程调用的 CPU 占用率都比本地调用高，表示虽然测试结果是通过两台性能很高的机器得出的，远程调用还是比本地调用消耗了更多的服务器能力，这也许能对“对于成为性能瓶颈的业务操作，远程调用可以充分利用 CPU 计算能力”这一论断提供一个反证。

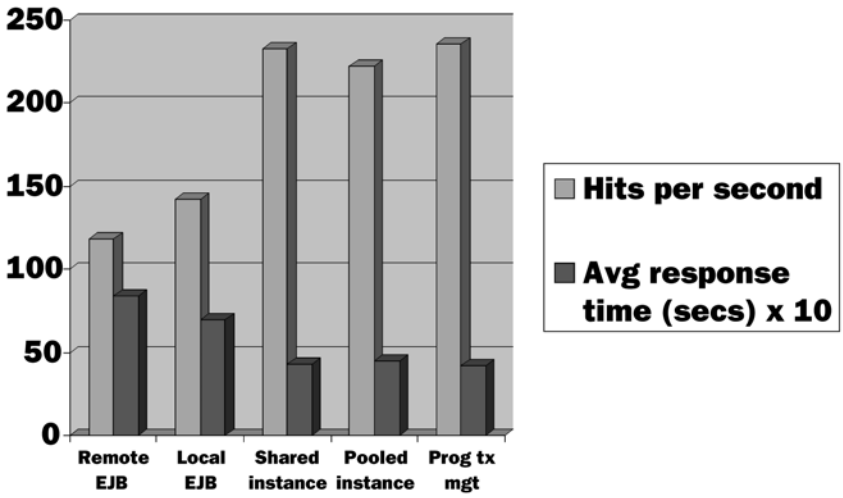



图 15-2

在所有的测试中，对比使用同样的 JTA 事务管理的本地 EJB 调用，在吞吐量和响应时间两个方面，Spring 方案都清楚地表明了其优越性。

可以大致总结如下：

- “调用 EJB 很慢”这个假设是错误的。本地 EJB 和并置运行的远程 EJB 性能都不错。在现代 EJB 容器中，在本地使用 EJB 并不会造成很大的开销。
- 远程 EJB 调用有很高的性能代价——对于大多数 web 应用，这个代价都太高了。
- 非 EJB 的方案能比本地 EJB 获得更好的性能，还能用更灵活的方式达到同样功能。除非你能用得起最高端的应用服务器，没有任何测试可以证明本地 EJB 比提供同样的声明式服务的 Spring AOP 效率更高，实际上 EJB 的性能看上去明显要低。Spring AOP 方法能够提供声明式事务管理，而开销比本地 EJB CMT 小得多。
- 不采用声明式服务、而是自己编程管理事务，无法得到合理的性能回报。在我们的测试中，这只获得了不到 10%的性能提升——不足以抵消付出的复杂性和潜在的错误的代价。编程获得事务要比声明式配置麻烦得多。
- 假若 web 容器的线程池大小得当，实例池比全局共享的单个实例性能要差。
- 虽然我们建议使用多线程运作的共享对象作为大多数业务对象的首选模型，假若你希望使用单线程模型，也未必使用 EJB。在我们的测试中，Spring 的池比开源应用服务器的 SLSB 实例池的性能好得多，也足以和高端商业服务器相提并论。
- 在 Spring 和 EJB 的声明式服务之间的性能比较，部分取决于你使用的应用服务器的品质。尽管在所有的基准测试中 Spring 的性能都比 EJB 来得好，如果转到不那么高效的应用服务器中，它的优势将更加明显。

> 没有证据表明对于声明式事务或单线程模型来说，本地 EJB 的性能更好。别的方案也可以让我们给 POJO 加上同样的行为，而且更简单、更灵活，性能也至少与 EJB 相当（甚至更好）。

选择 EJB 也不是一个稳定的方案。在我们的测试中，当操作需要耗费很长时间时，Spring 实现在各种线程模型中负载并发会话的能力与 EJB 相当，并且占用的 CPU 更少。最引人注目的是，它的响应时间更短、更具一致性，说明它可能比 EJB 更加稳定。（当然，必须说明：EJB 和 Spring 方案都很稳定，即便负载量很大也是如此。）

# 缓存和代码优化

现在，让我们简要考虑一下其他的实现问题，主要是与代码优化相关的问题。

## 代码优化，以及为何要避免它

代码优化不仅仅是提升性能的错误方法：实际上，在大多数情况下，它都是有害的。

- 它很困难：代码优化会需要很多工作量
- 它可能降低可维护性，引入 bug
- 它常常是不必要的

> 杰出的计算机科学家高德纳（Donald Knuth）写道：“我们应该忘记大约 97%的小优化，不成熟的优化是万恶之源”（着重号为本书作者所加）。这真是明智的建议。

我们应该对优化采取保守的态度，小心对待它。优化——不管是架构性的还是实现性的——都只应该针对执行缓慢的操作来考虑。虽然这可能是一个显而易见的观点，但我常常见到开发者在一些代码上殚精竭虑，仅仅因为他们知道这一小段代码还可以运行得更快——虽然这些代码在任何现实场景中都算得上“足够快”。应该考虑从架构和实现中加以优化的、最明显的慢速操作的代表包括：

- 过多的数据库访问
- 远程方法调用

导致最大性能损失的通常是整体架构，而不是糟糕的实现代码。

当然这不是说我们在编写程序的时候应该把性能因素扔在一边。有些开发者不管处理什么问题都是用最慢的方法，但是根据我的经验，有能力的开发者想出的最简单直接的办法通常会是最优的。通常会有更快、但稍微绕点弯子（因此比较难于维护）的解决方法，但是获得的回报通常不足以抵消其复杂性。

> 需要优化的是设计，而非代码。

## 缓存

缓存是一种介于架构和代码级别优化之间的重要优化。

虽然合理使用缓存能获得很大的性能提升，但缓存是复杂的，很容易引发错误。如果没有强烈的需要，最好不要实现任何缓存。*Expert One-on-One J2EE Design and Development* 的第 15 章详细讨论了缓存的优缺点，包括缓存和采样的例子。这里不会再次重复，只关注一些缓存的基本方针。

- 只应该在对业务需求有明确认识的前提下才实施缓存。是否能容忍部分数据过期？如果出现竞争，有什么隐藏的含义？
- 数据读超过写的程度越多，缓存就越有价值。如果对被缓存的数据进行大量写操作，缓存会变得难以实施，获得的性能提升也会大幅下降。
- 在结构的每一层都可以进行缓存。最好避免重复缓存，要考虑整个应用结构，而非只关注某个特定层。从 EIS 层开始，可以进行缓存的地方有：
  - 数据库
  - O/R mapping 层
  - 业务对象
  - web 层对象
  - 过滤器与 JSP 标签缓存
  - 通过 HTTP 头控制的浏览器 web 缓存
- 假若不采用分布式架构，数据访问或者业务层缓存能获得更大的好处。在传统的分布式 J2EE 应用中，数据访问缓存与 web 层距离太远了。
- 简单场景的缓存往往能带来很大的效益，比如对只读数据的缓存。
- 如果在应用程序中需要考虑复杂的并发问题，可以考虑采用第三方缓存产品，或者至少使用一个并发类库来协助处理线程问题。
- 对业务层来说，AOP 是一个优秀的缓存选择。

> 很多 J2EE 开发者（包括我在内）都有一种近乎直觉的假设：即便一次方法调用命中了被缓存的中间层业务对象，仍然会造成高昂的开销。这可能是因为使用分布式 EJB 的经验留下的后遗症：对分布式 EJB 进行的调用都是远程方法调用，因此方法调用本身开销也很大。如果使用并置的架构，特别是如果你发现自己正在用复杂的设计避免调用中间层，就应该问问自己：是不是这种后遗症又开始发作了。

当然，你仍然需要控制数据访问，因为这的确很慢。但是，此时进行缓存的最佳位置可能是中间层业务对象，而非 web 层。

不妨把上面列表中的最后一点（即 AOP）稍做展开讨论。广为人知的缓存方法之一就是使用 Decorator 设计模式：实现一个缓存装饰器（decorator）类，它与目标类实现同样的接口，但是会进行一些数据缓存，从而减少对目标调用的次数。（这也可以被认为是缓存代理。）

如果使用 Decorator 模式，我们可以编写这样一个实现，将所有方法委托给原来的对象，并对某些方法进行缓存。我们也可以继承原对象，并覆盖一些方法来提供缓存。这两种方法都不够优雅。使用第一种方式，即使只需要处理某几个方法，也需要对所有方法编写代码，造成很多重复的代码。继承的方式能避免这个问题，但是无法为同一接口、不同实现类型的目标对象提供缓存，也就是说，缓存被紧密绑定到一个具体的类。

AOP 用来对付这种横切注入非常理想。我们可以使用缓存**注入代码**（advice）来把缓存**方面**（aspect）的代码从客户代码中完全分离出来。通过**切点**（pointcut），AOP 给了我们有力的手段来指定哪些方法是缓存的目标。我们也可以考虑采用元数据属性（metadata attribute）来标注可以缓存的值。（比如，指出 `getHeadline()`的方法可以在 10 分钟之内保持缓存。）

这种通用的缓存甚至可以在集群环境中提供一致性。可以切换不同的缓存实现，而无需改变应用程序。

这种缓存可以对任何业务对象都有效，不会丢失任何强类型检查。

在 Spring 和其他基于拦截器的 AOP 框架中，这可能体现为**缓存拦截器**（caching interceptor）的形式。

# 潜在代码优化

通过下列技巧，可以让危险变得最小，而代码优化获得的回报最大：

- **只进行必要的优化。**使用基准测试和采样来确定代码优化的关注点。
- **关注容易进行的优化。**常常通过简单的优化就可以得到很好的回报。应该避免采用复杂的优化，以免得不偿失，还降低可维护性。
- **有一个详尽的回归测试套件，才是安全的。**如果详尽的测试套件无需更改就能通过，你就可以知道：你的代码做到了原来所有的事情，而且更快，这真是太痛快了！

有很多优化技巧，我们不可能在这里罗列他们。我们会选取一些简单易用见效快的优化。请参见本章后面的“资源”部分，那里有很多有用的参考资料。

## 从算法开始

在考虑底层代码细节之前，先检查运行缓慢的方法所使用的算法。是否有冗余？是否有被忽略的缓存的机会？对此问题是否有更好的算法？

## 选用正确的集合类型

有时候，选择正确的集合类型可以大幅度提高性能。选择正确的数据结构——比如，在合适的时候选择哈希表而不是列表——应该成为你的直觉。但是，同样数据结构的不同实现可能也有很大的性能差异。比如，如果你需要读入大量元素，预先获取一个大的 `ArrayList` 可能比使用 `LinkedList` 快得多。在 `ArrayList` 中进行随机访问也比 `LinkedList` 快。

看一下 `java.util` 包中它们的不同实现就知道为什么会这样了。有时候看看标准类库的源代码也是大有裨益的。

### 避免不必要的字符串操作

Java 中的字符串相加是很慢的。字符串是不可变的，也就是说不可以被修改其中的内容，每次需要相加的时候都会创建一个新对象。通常使用 `StringBuffer` 比 String 效率高。

## 不必要的log

请看下面的语句：

```
for (int i = 0; i < myArray.length; i++) {
  foo.doSomething(myArray[i]);
  logger.debug( “Foo object [ “ + foo + “] processing [ “+ myArray[i] + “]” );
}
```

这段代码中写 log 的部分看起来没什么问题，结果耗时却比数组操作还多。为什么呢？

因为字符串操作很慢，而且常常没有在编程的时候考虑效率问题（可能会显示一些很少使用的对象状态，而这些状态值需要额外的计算才能得到），而且 `toString()`调用通常是很慢的。因此上面的代码中，在循环中不加保护地使用 log 输出并不合适。

因此，至少我们需要加一点保护：

```
for (int i = 0; i < myArray.length; i++) {
foo.doSomething(myArray[i]);
  if (logger.isDebugEnabled()) {
        logger.debug(“Foo object [“ + foo + “] processing [“+ myArray[i] + “]”);
  }
}
```

这样好多了，因为“检查一个特定的 log 级别是否打开”要比字符串处理快得多。但是在这种情况下，值得去想一下：这里的 log 有什么用处吗？是否真的有人会打开 log 察看这么长的输出？而且，在这个循环中输出那么多信息，就会淹没其他的调试记录，让它们难以使用。

如果你大量使用调试 log，就算对 log 类库的 `isDebugEnabled()`这样的方法调用可能都是昂贵的。（这些方法很快，但是仍然会消耗一些资源。）这时，应该把方法调用结果放在一个 `boolean` 变量中。

## 有争议的优化

上面的优化方法，从代码质量的角度来说都是完全的保守方法。但是，在特殊情况下，也可以使用一些激进的代码优化方法，比如：

- **控制继承树的深度。**虽然太深的继承树可能暗示设计很糟糕，不过我们通常不想让性能因素影响面向对象设计。但是，在对性能要求苛刻的场合，消除继承也能明显提高性能。在很特殊的情况下，可以考虑是否可以容忍靠拷贝粘贴部分代码，而不是把它们重构到一个基类中去。
- **直接操作对象字段，而非通过方法访问。**通过让 AOP 代理直接访问 `org.springframework.aop.framework.AdvisedSupport` 代理管理器类的字段、而不是通过方法调用访问，我降低了 Spring AOP 大约 20%的开销。这需要把 `AdvisedSupport` 中的一些变量的访问级从 `private` 变成 `protected`（通常为了加强类继承树中的封装，默认访问级都是 `private`）。这种情况下，获得这样的性能提升是值得的。而在一般情况下，不应该这么做，不应该把类之间联系得如此之紧。
- **用局部变量来代替实例变量。**局部变量在堆栈中分配，访问起来快得多。如果某个实例变量被反复访问，把它复制到局部变量中能提高性能。

只有在你绝对以性能优先的时候以上这些技巧才是有价值的。因为 Spring AOP 框架是很多应用程序的基础设施，因此它的确对性能要求苛刻；然而，很少有应用程序的代码需要这样做。只有在有足够的证据时，你才能考虑使用这些具有侵入性的技巧来优化特定部分的代码。

> 一般来说，要避免出现会让代码变得复杂、或者增加维护工作量的优化。

## 改进测试的机会

有时候，进行一项优化可能破坏你或者你的同事编写某个测试套件时的假设，比如可能会把“每次创建一个新对象”改为“始终重用同一个对象”。可能会带来原先的测试未曾覆盖的失败情况，比如对象状态可能不一致。这些情况就提醒你应该**在进行优化之前**编写更多的测试，将这些失败情况检查出来。无论优化是否成功，新加的测试都可以对防止未来的 bug 有所帮助。

> 好的开发者总是不断寻找让测试变得更严格的机会。有时候在进行优化之前强化测试套件是很重要的。

# 调优和部署

通常，在深入优化程序代码之前，最好能检查一下部署的调优选项。调优部署的工作量要小得多，也不会带来复杂的维护问题。有时候它能提供非常惊人的效果，同时提高程序稳定性。

## JVM

首先，为你的应用服务器和应用程序选择合适的 JVM。我已经发现 BEA JRockit 有很高的性能，对长时间运行的服务器端应用，大概会比标准的 Sun JRE 快 2 到 4 倍。但是，你应该运行你自己的基准程序，因为不同的 JVM 有不同的长短之处，没有一个适合所有的应用。

下一步，正确地配置 JVM。查阅它的文档，以及你的应用服务器提供的文档，确保你针对应用服务器和应用程序正确地配置了那些重要的设置，包括：

- 初始堆尺寸与最大堆尺寸
- 垃圾收集选项
- 线程选项

下面的连接有一些示例：

- `http://edocs.bea.com/wljrockit/docs81/tuning/`：“调优WebLogic JRockit 8.1 JVM
- `http://publib7b.boulder.ibm.com/wasinfo1/en/info/aes/ae/urun_rconfproc_jvm.html`：“Java虚拟机设置：WebSphere应用服务器”

在项目生命周期的前期就要进行基准测试，选择合适的 JVM，进行合适的设置。

## 应用服务器

首先，针对你的性能需求选择合适的应用服务器。不同应用服务器的性能大相径庭，在同样硬件上的性能可能差别很大，也就是说：在应用服务器这里多花一点钱，也许就可以在硬件这里少花一点钱。对于典型的 web 应用，不同应用服务器的性能可能相差 3~4 倍。

> 假若你必须选用不是那么高效的服务器，轻型容器方案就更为吸引人了。把低效的 **EJB** 容器换成 **Spring AOP**，你可以避免服务器中的很多低效的代码，同时 **JTA** 实现与容器数据源等重要特性仍然得到保留。

第二步，针对你的应用特点定制应用服务器配置。应用服务器设置对性能和吞吐量有重要的影响，包括：

- **数据库连接池大小**。这对吞吐量有至关重要的影响。连接池太小会导致线程因等待连接释放而阻塞；而太大的连接池在集群环境中可能带来问题，数据库可能耗光连接监听器。

- **线程池大小**。线程池太小会浪费 CPU 能力，太大的线程池可能反而降低性能。
- **使用 SLSB 时的实例池大小**。它的影响和线程池相似。不要把它设置得比线程池更大。
- **使用 SLSB 时的事务描述符**。除非业务对象的所有方法都是需要是事务的，不要对所有方法都使用同样的事务声明。确定每个方法都有合适的事务属性：如果不需要事务就是 none。
- **HTTP session 复制选项**。根据你的应用程序选择是把 session 保存到数据库还是保持在内存中，仔细检查你的应用服务器提供的选项。
- **JTA 选项**。假若只使用一个数据库，必须确保你没有使用开销巨大的 XA 事务。
- **log 设置**。应用服务器和应用程序一样都会产生 log 输出，这可能造成巨大的开销，记住让你的服务器只产生必要的信息 log。

应用服务器厂商通常会提供关于可用选项的良好文档。

## 框架配置

我们前面说过，持久化技术对性能和可伸缩性有巨大影响，因此是否正确配置它们极为重要，特别是使用缓存（和分布式缓存）的时候。

如果你使用 Spring 之类的框架来取代中间件的一些职责，而不是采用 EJB 容器，你需要确认你正确配置了它。它的原理和 EJB 是相似的，但是具体的设置有所不同。

比如说，如果你使用前面讨论过的 Spring 池，和决定无状态 session bean 的池大小类似，你需要设置合理的池大小。Spring AOP 框架中的不同选项也可能影响性能，不过默认配置在大多数主流应用中都运作良好。重要的 Spring 设置还包括事务 AOP 代理的事务设置。和 SLSB 一样，不要为那些无须事务的方法创建不必要的事务，这很重要。

对于所有参数，都要有能方便重复运行的基准测试来检查配置变更的影响。理想的基准测试是长时间的负载测试，在测试新配置的性能的同时检查代码是否稳定。

## 数据库配置

数据库是否正确配置极为重要。数据库服务器是复杂的产品，DBA 对于项目成功至关重要。每种数据库有不同的设置，重要的参数有：

- 线程模型
- 监听器池

- 索引
- 表结构（Oracle 等数据库为不同的使用模式提供很多不同的表结构）
- 查询优化选项

数据库产品通常会提供采样工具，有助于指出哪些地方可以提高性能。

# 一种循证的性能策略

性能问题**可以**并且**应该**以科学的方式来解决。然而，现实生活中却常常不是这样的。我不断看到这样的问题：

- 没有尽早收集性能的客观证据
- 根据没有证据的猜测来进行性能优化

第一个问题会浪费工作量和时间——比如，一个需要验证的结构没有及早得到验证，到后来才发现它有性能问题。前一章中，我讨论过在项目周期中及早创建**垂直切片**（vertical slice）或者**可执行架构**（executable architecture）的重要性。在性能问题上，它们能够帮助验证性能指标，这是很重要的。现代方法学——即便在实践上有所不同，例如 RUP 与 XP——都强调这类架构验证的重要性。不幸的是，实际生活中这还是做得太不够了。

假若没有及早发现并避免性能问题，等到后来才来解决可能就太晚了；但事情常常还没这么简单：整件事情可能会演变成一场政治辩论，你得花费更多努力才能找到问题所在，而解决问题的难度就更大了。（我看见过好几个项目在出现问题时将其掩盖起来，而不是集中精力解决问题。）

> 在项目早期，要做垂直切片试验，来验证备选架构的性能，确认它能够满足需求。

因为性能的重要性、以及优化的代价和风险，不应该在没有证据基础的情况下做出性能判断，因为那有可能导致浪费时间在优化那些根本不造成任何问题的代码上。

> 在没有实证的时候，千万不要做出重要的决策。

让我们看看有哪些方法可以用来收集必须的证据，来指导我们的调优过程。

## 基准测试

在项目的早期，要进行基准测试来确认能达到性能目的。我经常看到架构师把基准测试步骤推迟到项目有显著进展之后，因为他们之前没办法得到生产级别的硬件，这不能成为理由：在开发者的桌面计算机上，已经可以获得很多有关性能的信息了。能够运行现代IDE 的机器，只要能够快速连接数据库之类的资源，就有能力为服务器端应用提供很高的吞吐量。

基准测试本质上用于减轻风险。假若能发现危险的瓶颈，就发挥出了它们的用处。假若一台 P4 桌面计算机只能得到每秒两次事务（2TPS）的性能，那么就不难得出这样的结论：再多的硬件也无法满足 100TPS 这一要求。

> 在开发者的桌面计算机上运行起来慢得令人担心的应用程序，在生产硬件上也会很慢。

基准测试必须仔细规划，因为有很多的变数需要控制（之前就讨论过一些），比如：

- JVM 版本
- 应用服务器配置（假若需要在部署环境中运行基准测试）
- log 级别
- 网络配置
- 数据库配置
- 负载测试工具

可以用于基准测试的工具有很多。但是，讨论不同的工具已经超出了本章的内容。

理想情况是，很多测试可以在容器外针对应用程序代码运行。但是，针对现实的发布配置进行基准测试也是很重要的。通常发布基准测试应该先运行，然后根据分段基准测试的结果来确定哪些部分可以改善。

对于有 web 界面的并置应用，在容器内运行基准测试非常容易，因为有很多 web 负载测试工具可供选择，比如有：

- Microsoft Web Application Stress Tool（WAS，`www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/intranet/downloads/webstres.asp`）：有些陈旧了，但是易于使用，也是免费的负载测试工具。
- Apache JMeter（`http://jakarta.apache.org/jmeter/`）：纯Java测试工具，可以用于测试纯Java类。配置起来相当复杂。

- The Grinder（`http://grinder.sourceforge.net/`）：另一个Java测试工具，也可以用来测试纯Java类。配置起来相当复杂。
- ApacheBench（`http://perl.apache.org/docs/1.0/guide/performance.html#ApacheBench`）：这个工具在任何Apache web服务器发行包中都已经包括。这是一个很简单的命令行测试工具，可以把服务器推至极限。没有提供复杂的脚本功能。

性能测试要像功能测试那样可以重复，相关的方法已经有人整理成文。注意别犯常见的错误，比如：

- 不要轻信单次测试的结果。结果必须可以重复。
- 小心别让负载测试软件扭曲了测试结果。
- 注意别让负载测试软件与运行被测试应用程序的应用服务器竞争 CPU 时间。对于 web 应用来说，最好在其他的一台或多台机器上运行负载测试，以消除任何此类影响。

假如能对程序的不同层面——比如展示层、业务对象（访问测试数据库或者替换的DAO）和数据库查询——分别进行基准测试，这会很有帮助，这能让我们清楚地看到：应该在哪些部分集中进行采样测试。

> 对于令人担忧的基准测试结果，随后的两节——“采样”和“诊断”——介绍了后续应该采取的步骤。假若你的应用已经能够满足性能要求，就不要浪费时间来采样和优化了。当然，进行优化可以让它更快，而且可以享受挑战自我的满足快感。（我个人很喜欢采样和性能优化。幸运的是，因为的很多时间是在编写基础代码，我比大多数应用程序开发者更经常享受这种乐趣。）但是，没有理由的话，不用让它变得更快。
>
> 可以进行一次快速的采样来看看是否有明显的瓶颈，但是不要在性能上浪费太多努力。总是有好多其他事情等着你做呢。

# 采样（Profiling）

如果我们发现了问题，如何逐步逼近它呢？

我强烈建议使用采样器，它可以帮助你明确分析每个方法使用了多少时间。

## 为什么要使用采样器?

在你发现性能问题后，为什么需要采样器呢?直接检查代码不能够发现瓶颈所在吗？

当然，对于一些常见的问题，可能可以预见到问题所在。比如：

- 我们知道，字符串操作很慢。
- 我们知道没有 `isInfoEnabled()`保护的 log 语句假若调用了速度缓慢的 `toString()` 方法、或者简单的字符串相加，也会变得很慢。

但是，我们并不知道是否这些“慢速”的代码是否真的造成了问题，我们也不知道解决这些问题是否值得。对 `toString()`值进行缓存可能造成问题，除非我们确信要这么做，不然这不是个好主意。

坚实的证据是无可替代的。采样可以指出**哪一段慢速代码真的在惹祸**。比如，我对性能采样有很多经验，对于特定方法我可能可以精确的给出各个片断执行时间的预测。但是我不能精确预测整个应用中需要照管的数千个方法。计算机处理这些任务比人可好多了。采样器可以轻松做到这件事。

采样器必定也可以指出哪些慢速的代码**可以忽略不管**。对于性能问题，很多开发者都抱持某些信念，这些信念或多或少有有一定的事实根据，比如：

- 反射很慢
- 对象创建很慢

我曾经看到过：在缺乏有效证据的情况下，开发者就把编写得非常漂亮的反射代码删除了，而用更复杂和晦涩的代码取而代之；或者实现复杂的池机制，以避免创建对象。这些假定很有可能是错误的。（在现代的 JVM 中，相比于实际运作的方法，反射已经相当快了。对象创建也很少再造成问题。）

因此，对应用程序代码采样是很重要的，不管是否在部署环境里都是如此。

> 采样要基于现实的应用场景。不要人为地制造那种会大量触发慢速代码、但不大可能在现实中出现的场景。

一开始，我通常在应用服务器内运行基准测试；而当我知道需要调整哪段程序的时候，会创建很多在外部运行的采样过程（在 IDE 里面）。通常我可以从调用堆栈中猜测出现问题的区域，通过与此关联的基准测试来确认它。假若这不可行，我就会在应用服务器中对应用程序采样。

我会尽量减少在应用服务器中运行采样的次数。虽然在 IDE 内部运行应用服务器、或者通过特殊的 JVM 配置来启动一个可以远程 debug 的应用服务器都是可以做到的，但这种做法有几个很重要的缺点：

- 和基础设施一起启动 JVM 会把整个现场都变得非常的慢。也就是说你的应用服务器启动的时间可能是以分钟计，而非以秒计。
- 很难把“杂音”过滤出去，比如应用服务器自己的代码。

> 尽量减少代码对 EJB 容器或者应用服务器的依赖的好处之一就是，你可以从容器外运行采样。比如，遵循本书建议的结构指南，你可以在 IDE 中对绝大多数应用代码进行采样，而不需要启动应用服务器。
>
> 对于调试来说，也能获得同样的好处。能够在应用服务器之外进行调试会变得容易许多。当然，有时候我们需要知道代码和 J2EE 服务器是如何交互的，但是更多的是要追踪那些肯定位于应用程序代码内部的错误。这时就和单元测试一样，我们也希望避免任何外部关系。

记住，因为采样器本身的消耗，也会拖慢性能曲线。因此要经常重新运行不包含采样的初始基准测试，来验证是否有性能提高。

几乎可以肯定，通过采样逐次得到的性能回报会逐步减少。通常你可以很轻松地摘下那些“较低的果子”，比如说去掉所有不必要的字符串操作等等。做完这些以后，你可能会发现自己已经取得了不小的成果，看到基准测试的切实提高会让你兴奋不已。但是，你会发现下面需要进行越来越多的努力，得到的性能提升却越来越少。你也会发现：为了得到这样很少的提高，你开始试图采用一些对可维护性会产生伤害的优化方法。我是唯一一个感受到这种诱惑的程序员吗？我深表怀疑。采样和代码覆盖分析一样，假若使用得当都是非常有用的工具，但是千万不要让它诱惑你，毁了你的正常工作方式。

> 采样对定位问题来说是无价之宝。它也能指出设计的哪一部分需要修改。

采样也是一种观察调用栈的有趣途径。这可能有助于理解代码的动态行为，也是观察代码静态结构的有益补充。你可能发现简单的代码错误，比如，不小心调用了一个方法两次，而应该使用第一次调用返回的对象引用。

因此，就算我没有遇到性能问题，我也喜欢进行采样——虽然这种情况下我不会花很多时间在上面。

## 采样实战

下面的笔记出自于我自己对 Spring 和其他项目使用采样工具的实际经验。在进行采样时，我依照下面的步骤：

1. 运行一次有实际意义的，典型的基准测试。
2. 使用采样器再运行同样的基准测试，可能减少负载量或操作数，以便平衡采样器对性能带来的影响。
3. 按照采样的结果修改代码，每次修改后重新运行完整的单元测试套件。

4. 去掉采样，经常重复第 2 步和第 3 步来检查是否性能得到了提升。第 1 步就不需要经常重做了。

一些小提示：

- 确保你的采样测试运行了足够多的操作，避免测试启动带来开销造成结果失真。
- 配置好你的采样器，把类库排除在外，以免淹没你自己应用程序的信息。

我通常使用 Eclipse Profiler 插件（`http://eclipsecolorer.sourceforge.net/index_profiler.html`）进行采样。对Eclipse用户来说，这个工具容易配置，用起来也很直观。这个采样器可以运行任何Java应用程序（也就是说，有一个`main`入口点），并且完全支持从采样结果跳到对应的源代码。

图 15-3 和图 15-4 展示了对 Spring AOP 框架的一百万次调用的采样结果——这足够消除启动对结果带来的影响。我展开了线程调用树，来显示所有占用超过 5%运行时间的方法。这个线程调用树同时也指出了运行时的结构。

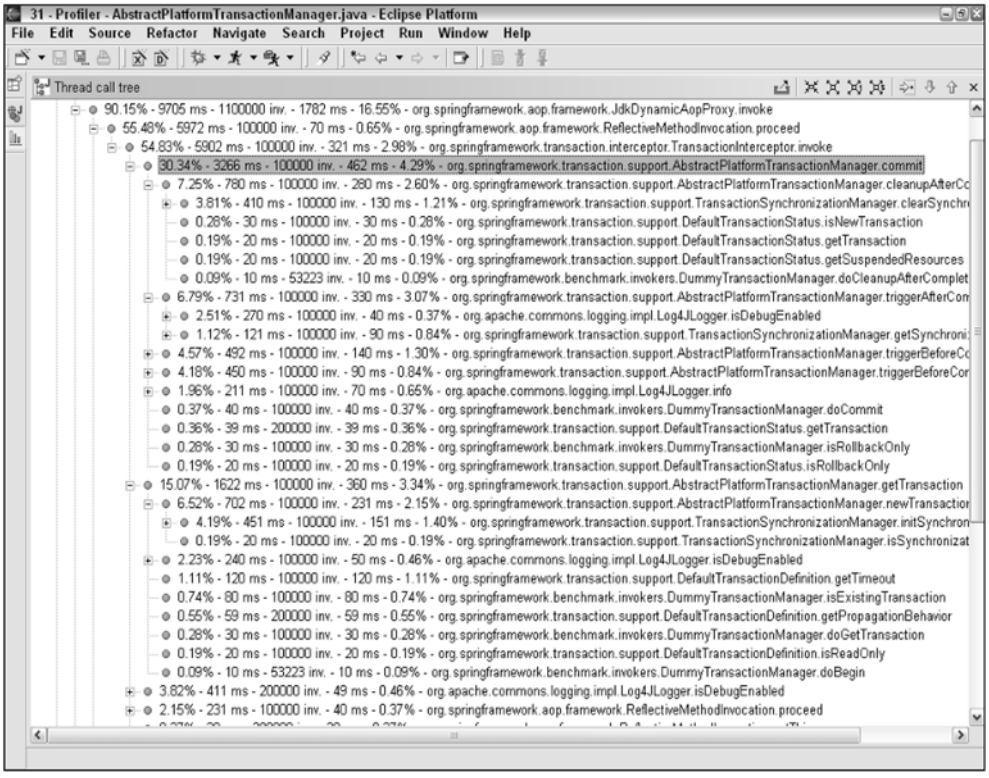
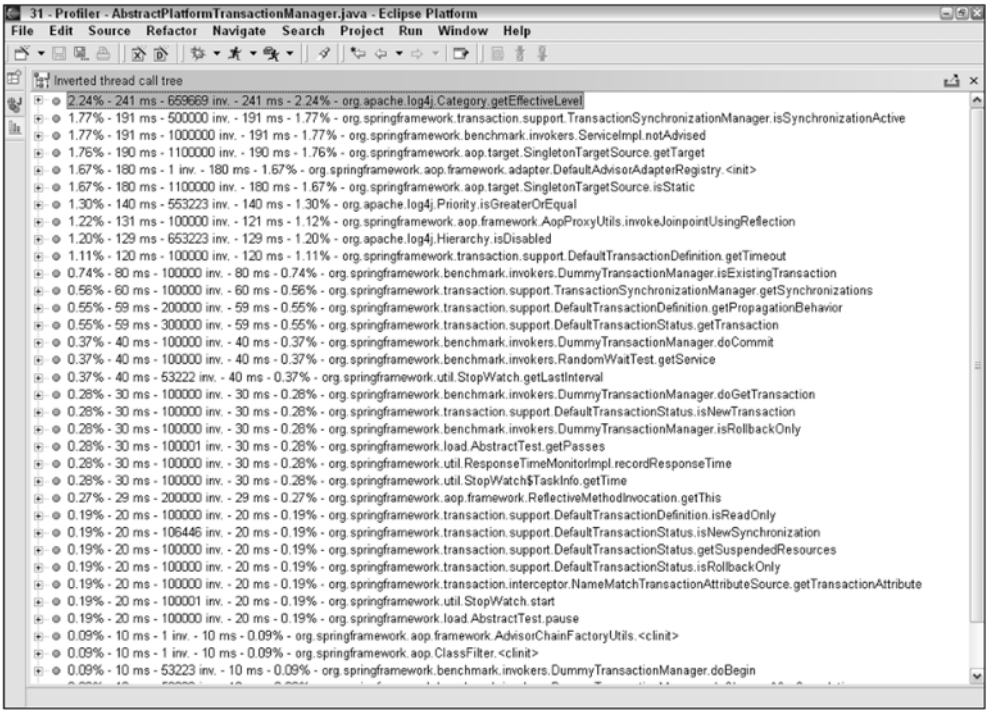



图 15-3




（图 15-4）

因为采样器在 IDE 内运行，所以可以从调用树直接跳转到源代码元素去，也可以用图形化的方式察看“热点”方法。

另一个有用的视图是“反转调用树”（图 15-4）：展示调用链末端的方法，并按照它们的执行时间排序。这对于观察哪些方法值得优化很有帮助。

以前我用过 Sitraka JProbe，作为商业产品它很优秀，但同时也很昂贵。我发现 Eclipse Profiler 对于大多数要求都可以满足了：特别是我喜欢在容器外采样。但是，商业产品通常对在应用服务器中采样提供了更好的支持，也能生成更详细的图表。

## 诊断

除了采样，我们也可以进行一些诊断，特别是对于部署环境。

好的采样工具可以让我们察看每个类创建对象的个数。从 JVM 也可以获取这种信息。对部署后的环境这会特别有用。假如你发现问题是因为过多的对象分配和垃圾收集造成的，检查一下你的 JVM 的文档，看看有什么标志你可以设置（比如`-verbosegc`），通过这些启动标志，你可以得到垃圾收集进程的信息记录。

通过应用服务器的控制台和其他监视工具，你也能在部署后的环境中观察连接池以及正在进行的事务的行为。

诊断对于生产系统特别重要，因为性能情况可能和负载测试的时候有所不同。要能够在不对性能或者稳定性造成负面影响的前提下进行行为分析，这是很重要的（因此不能使用采样）。

## 资源

对代码优化和其他性能问题来说，我建议继续阅读下面的参考资料：

- *Java Performance Tuning*，Jack Shirazi 著(O'Reilly 出版)。这本书提供对 Java 性能问题的精彩讨论。虽然这本书主要关注底层代码优化技巧，没有详细覆盖 J2EE，仍然值得任何对编写高性能代码有兴趣的开发者阅读。
- Shirazi的web站点：`www.javaperformancetuning.com`。这个站点有很多有用的链接，也比上面的书多了很多有关J2EE的内容。
- *Expert One-on-One J2EE Design and Development*，第 15 章（“性能测试与应用调优”），本人所著。这本书讨论了对 J2EE 有用的代码优化技巧，包括采样和负载测试工具，以及——假若你必须使用分布式结构的话——提高序列化性能的技巧。
- *J2EE Performance Testing with BEAWebLogic Server*，Peter Zadrozny、Philip Aston 和 Ted Osborne 所著，Expert Press 2002 年出版。这本书对很多基于实证的案例进行了精彩讨论，虽然有点过时了，仍然值得一看。

## 总结

基础架构的选择很大程度上决定了企业应用的性能特征。没有比“分布式架构还是并置架构”更基本、更重要的抉择了。我们的基准测试显示，在真实的 web 应用场景中，远程 EJB 调用比本地 EJB 调用慢整整一个数量级。

> 如果你关注性能，除非商业上需要，不要采用分布式架构。“依靠分布式架构获得可伸缩性”的说法是非常值得怀疑的。

通过并置部署所有应用组件，我们可以避免远程调用的高昂代价。假若必要，我们可以对整个应用部署进行集群。

除了远程调用的巨大开销之外，只要使用得当，EJB 中没有任何部分是天生缓慢的。

现代的 EJB 相当高效，对本地 EJB 调用没有很大的开销——不过毫无疑问，在开发、部署和测试方面，EJB 带来的成本很高。因此，假若使用 EJB，我们不必担心调用本地 EJB 的性能代价，也不应该认为本地 EJB 会增加开销而减少 EJB 的使用。

但是，从性能角度而言，EJB 背后没有什么魔术。在本章中，我针对典型的 web 应用的场景，比较了本地 EJB 和 Spring AOP 架构的基准测试成绩，后者也提供了声明式事务管理和——可选的——线程管理。这二者在使用底层 J2EE 服务上是等价的，因为 Spring 部署也配置为使用 JTA 作为底层事务基础设施。

Spring 方案比 EJB 方案的性能更高。我在所有的 EJB 容器中都得到了同样的结果，包括高端的商业产品。随 EJB 容器不同，性能的差异程度也有所不同，在某个不是特别高效的 EJB 容器中表现得特别明显——那是一个流行的开源产品。说得更具体点，Spring 方案比 EJB 方案提供了更短的响应时间，反应也更一致。

这是个重要的结果。性能是一个非常重要的考虑因素。我介绍的轻量级 Spring 解决方案在几乎任何重要的方面都比本地 EJB 更好——开发效率；成熟的 IoC 方案；容易测试，与 TDD 兼容；避免 OOP 的那些不好的效果；更加灵活的声明式事务管理；真正的 AOP 框架。现在，我们又亲眼看到 EJB 方案不能提供更好——甚至哪怕只是与 Spring 方案相同——的性能和可伸缩性，这就连选择 EJB 的最后一个论点都驳斥了。

> EJB 的声明式服务(比如 CMT)和为 POJO 提供同样服务的高效的 AOP 实现相比，没有效率或者可伸缩性上的优点。我们已经在比较本地 EJB 方案和 Spring AOP 方案时得出了这一结论。

数据访问方法对性能和可伸缩性有重要影响。这里两个重要的选择是：是否适合采用 O/R mapping，以及如何在分布式环境中获得一致的数据缓存。O/R mapping 对于基于集合的关系操作不合适。在适合采用 O/R mapping 的场合，我们建议采用专业的 O/R mapping 产品，比如 Kodo JDO 或者 Hibernate，或许应该和一个成熟的分布式缓存产品一起使用。

> 不用硬性要求使用 O/R mapping。假若你使用 O/R mapping，就应该使用一个成熟和灵活的 O/R mapping 工具。

总而言之，要记住简单通常有助于带来良好的性能——通常也有助于按时交付、并获得良好的可管理性。使用EJB常常带来了不必要的复杂性。杰出的J2EE作家和咨询师Bruce Tate 在接受 JavaPerformanceTuning.com 的一次采访时表明了这一点（`www.javaperformancetuning.com/news/interview036.shtml`）。Tate重申了简单对性能带来的功效。记者问他：“你所见过的获得最大性能提升的项目中，采用了什么变化？”他的回答是：“我们抛弃了EJB，用简单的POJO方案来代替。”

因为性能和可伸缩性对项目的成败有重要影响，所以必须根据坚实的证据来做出性能

方面的决策。我的建议是：

- 在项目的早期，对垂直切片进行基准测试
- 在开发周期中，要反复进行性能测试，检查是否有性能衰退
- 使用采样工具来检查瓶颈，确保性能优化都是有的放矢

我也介绍了一些代码级别的优化技巧，它们可以获得性能提升。但是要警告你，这些技巧也许会损害代码质量和可维护性，并且只能得到很少的性能提升，你必须做出权衡。

更明显的性能提升很可能并非来自代码优化。如果你能选择最适合需求的应用服务器，同时确保服务器、JVM 和其他核心软件（比如数据库）都被配置在最佳状态，你得到的性能提升将胜过任何代码优化。