J2EE 开发注意事项

目录

引言

在过去的几乎整整十年中，人们编写了很多有关 Java™ Platform, Enterprise Edition (Java EE) 最佳实践的内容。现在有十多本书籍和数以百计（可能更多）的文章，提供了关于应该如何编写 Java EE 应用程序的见解。事实上，这方面的参考资料如此之多，并且这些参考资料之间往往还存在着一些矛盾的建议，以至于在这些混杂的内容中进行学习本身也成为了采用 Java EE 的障碍。因此，为了给刚进入这个领域的客户提供一些简单的指导，我们汇编了这个最重要的最佳实践列表，其中包括我们认为最重要和最有效的 Java EE 最佳实践。遗憾的是，我们无法仅在 10 大最佳实践中描述所有需要介绍的内容。因此，为了避免遗漏关键的最佳实践和尊重 Java EE 的发展，我们的列表中包含了“19 大”关键的 Java EE 最佳实践。

1. 始终使用 MVC 框架。

将业务逻辑（Java Bean 和 EJB 组件）从控制器逻辑（Servlet/Struts 操作）和表示逻辑（JSP、XML/XSLT）中清晰地分离出来。良好的分层可以带来许多好处。

这项实践非常重要，以致没有其他最佳实践可以与其相提并论。对于良好的 Java EE 应用程序设计而言，模型-视图-控制器 (MVC) 是至关重要的。它将程序的任务简单地分为下面几个部分：

a. 负责业务逻辑的部分（模型，通常使用 Enterprise JavaBeans™ 或传统 Java 对象来实现）。
b. 负责用户接口表示的部分（视图）。
c. 负责应用程序导航的部分（控制器，通常使用 Java Servlet 或类 Struts 控制器这样相关的类来实现）。

对于 Java EE，有许多关于这个主题的优秀评论，我们特别推荐感兴趣的读者可以参考 [Fowler] 或者 [Brown]（请参见参考资料部分）的评论，以便全面和深入地了解相关内容。

如果不遵循基本的 MVC 体系结构，在开发过程中就会出现许多的问题。最常见的问题是，将过多的任务放到该体系结构的视图部分中。可能存在使用 JSP 标记来执行数据库访问，或者在 JSP 中进行应用程序的流程控制，这在小规模的应用程序中是比较常见的，但是，随着后期的开发，这样做将会带来问题，因为 JSP 逐步变得越来越难以维护和调试。

类似地，我们也经常看到将视图层构建到业务逻辑的情况。例如，一个常见的问题就是将在构建视图时使用的 XML 解析技术直接应用到业务层。业务层应该对业务对象进行操作，而不是对与视图相关的特定数据表示进行操作。

然而，仅仅使用适当的组件无法实现应用程序的正确分层。我们常常见到一些应用程序包含 Servlet、JSP 和 EJB 组件所有这三项，然而，其主要的业务逻辑却是在 Servlet 层实现的，或者应用程序导航是在 JSP 中处理的。您必须对代码进行严格的检查和重构，以确保仅在模型层中处理业务逻辑，在控制器层中进行应用程序导航，而视图应该只关心如何将模型对象呈现为合 适的 HTML 和 Javascript™。

本文中这项建议的涵义应该比原始版本中的更加清楚。用户接口技术不断地发生着变化，将业务逻辑关联于用户接口，会使得对接口的更改影响到现有的系统。几年之前,Web 应用程序用户接口开发人员可能从 Servlet 和 JSP、Struts 和 XML/XSL 转换中进行选择。在那以后，Tiles 和 Faces 非常流行，而现在，AJAX 大行其道。如果每当首选的用户接口技术发生了更改就要重新开发应用程序的核心业务逻辑，那么就糟透了。

2. 不要做重复的工作。

使用常见的、经过证实的框架，如 Apache Struts、JavaServer Faces 和 Eclipse RCP，使用经过证实的模式。

回到我们开始帮助客户使用刚出现的 Java EE 标准的时候，我们发现（和许多其他人一样），通过直接使用基础的 Servlet 和 JSP 规范构建 UI 应用程序来开发用户接口开发框架，可以极大地提高开发人员工作效率。因此，许多公司开发了他们自己的 UI 框架，这些框架可以简化接口开发的任务。

随着开放源码的框架（如 Apache Struts）的出现 [Brown]，我们相信，可以自动地和快速地转换到这些新的框架。我们认为，使用开放源码社区支持的框架非常适合于开发人员，并且这些框架很快得到了广泛认可，不仅可用于新的开发，还可以修改现有的应用程序。

但令人感到奇怪的是，事实并非如此。我们仍可以看到许多公司在维护或甚至开发新的用户接口框架，而这些框架的功能与 Struts 或者 JSF 是完全相同的。之所以会出现这种情况，有许多原因：机构惰性，“非我发明”症，不了解更改现有代码的好处、或者甚至傲慢地认为能够比开放源码开发人员的特定框架做得更好。

然而，这些原因都已经过时了，不能够成为不采用标准框架的借口。Struts 和 JSF 不仅在 Java 社区中得到了广泛认可,而且还受到 WebSphere 运行时和 Rational® 工具套件的全面支持。同样地，在富客户端领域中，Eclipse RCP（富客户端平台，Rich Client Platform）获得了广泛的认可，可用于构建独立的富客户端。尽管不

是 Java EE 标准中的一部分，但这些框架现在已成为 Java EE 社区的一部分，并且理应如此。

对于那些因为傲慢而不愿使用现成的 UI 框架的人，应该阅读 [Alur] 和 [Fowler] 中介绍的内容。这两本书详细地描述了企业 Java 应用程序中最常用的可重用模式。从类似于会话 Facade 这样简单的模式（将在后面的建议中讨论）到类似于 Fowler 持久性模式（许多开放源码的持久性框架对其进行了实现）这样比较复杂的模式，其中的内容体现了 Java 前辈们所积累的智慧。那些不能吸取教训的人必定会重蹈覆辙（如果他们非常幸运，能够在第一次失败之后获得重来一次的机会），他们不得不向哲学家 Santayana 说抱歉。

3. 在应用程序的每一层都使用自动单元测试和测试管理。

不要只是测试您的图形用户界面（GUI）。分层的测试使得调试和维护工作变得极其简单。

在过去的几年中，在方法学领域有了相当大的革新，例如新出现的被称为 Agile（如参考资料部 分中的 SCRUM [Schwaber] 和极限编程 [Beck1]）的轻量级方法现在已经得到了很普遍的应用。几乎所有的这些方法中的一个共同特征是它们都提倡使用自动的测试工具，这些工具可以帮助开发人 员用更少的时间进行回归测试，并可以帮助他们避免由于不充分的回归测试造成的错误，因此可以用来提高程序员的工作效率。实际上，还有一种被称为 Test-First Development [Beck2] 的方法，这种方法甚至提倡在开发实际的代码之前就先编写单元测试。然而，在您测试代码之前，您需要将代码分割成一些可测试的片断。一个“大泥球”是难以测试的，因为它不是只实现一个简单的易于识别的功能。如果您的每个代码片断实现多个方面的功能，将难以测试其中的每个部分以保证其正确性。

MVC 体系结构（以及 Java EE 中的 MVC 实现）的一个优点就是元素的组件化能够（实际上，相当的简单）对您的应用程序进行单元测试。因此，您可以方便地对实体 Bean、会话 Bean 以及 JSP 独立编写测试用例，而不必考虑其他代码。现在有许多用于 Java EE 测试的框架和工具，这些框架及工具使得这一过程更加简单。例如，JUnit（是一种由 junit.org 开发的开放源代码工具）和 Cactus（由 Apache 协会开发的开放源代码工具）对于测试 Java EE 组件都非常有用。[Hightower] 详细探讨了如何在 Java EE 中使用这些工具。

尽管所有这些详述了怎样彻底地测试您的应用程序，但是我们仍然看到一些人认为只要他们测试了 GUI（可能是基于 Web 的 GUI，或者是独立的 Java 应用程序），则他们就全面地测试了整个应用程序。仅进行 GUI 测试是不够的。GUI 测试很难达到全面的测试，有以下几种原因。

1. 使用 GUI 测试很难彻底地测试到系统的每一条路径,GUI 仅仅是影响系统的一种方式。可能存在后台运算、脚本和各种各样的其他访问点，这也需要进行测试，然而，它们通常并不具有 GUI。
2. GUI 级的测试是一种非常粗粒度的测试。这种测试只是在宏观水平上测试系统的行为，这意味着一旦发现存在问题，则与此问题相关的整个子系统都要进行检查，这使得找出错误将是非常困难的事情。
3. GUI 测试通常只有在整个开发周期的后期才能很好地得到测试，这是因为只有这那个时候 GUI 才得到完整的定义。这意味着只有在后期才可能发现潜在的错误。
4. 一般的开发人员可能没有自动的 GUI 测试工具。因此，当一个开发人员对代码进行更改时，没有一种简单的方法来重新测试受到影响的子系统。这实际上不利于进行良好的测试。如果开发人员具有自动的代码级单元测试工具，开发人员就能够很容易地运行这些工具以确保所做的更改不会破坏已经存在的功能。
5. 如果添加了自动构建功能，则在自动构建过程中添加一个自动的单元测试工具是非常容易的事情。当完成这些设置以后，整个系统就可以有规律地进行重建，并且回归测试几乎不需要人的参与。

另外，我们必须强调，使用 EJB 和 Web 服务进行分布式的、基于组件的开发使得测试单个组件变得非常必要。如果没有“GUI”需要测试，您就必须进行低级（lower-level）测试。最好以这种方式开始测试，省得当您将分布式的组件或 Web 服务作为您的应用程序的一部分时，您不得不花费心思重新进行测试。

总之，通过使用自动的单元测试，能够很快地发现系统的缺陷，并且也易于发现这些缺陷，使得测试工作变得更加系统化，因此整体的质量也得以提高。

4．按照规范来进行开发，而不是按照应用服务器来进行开发。

要将规范熟记于心，如果要背离规范，需经过慎密的考虑后才可以这样做。这是因为当您背离规则的时候，您所做的事情往往并不是您应该做的事情。

当您要背离 Java EE 允许您做的事情的时候，这很容易让使您遭受不幸。我们发现有一些开发人员钻研一些 Java EE 允许之外的东西，他们认为这样做可以“稍微”改善 Java EE 的性能，而他们最终只会发现这样做会引起严重的性能问题，或者在以后的移植（从一个厂商到另一个厂商，或者是更常见的从一个版本到另一个版本）中会出现问题。实际上，这种移植问题是如此严重，以致 [Beaton] 将此原则称为移植工作的基本最佳实践。

现在有好几个地方如果不直接使用 Java EE 提供的方法肯定会产生问题。一个常见的例子就是开发人员通过使用 JAAS 模块来替代 Java EE 安全性，而不是使用内置的遵循规范的应用服务器机制来进行验证和授权。要注意不要脱离 Java EE 规范提供的验证机制。如果脱离了此规范，这将是系统存在安全漏洞以及厂商兼容性问题的主要原因。类似地，要使用 Servlet 和 EJB 规范提供的授权机制，并且如果您

要偏离这些规范的话，要确保使用规范定义的 API（例如 getCallerPrincipal()）作为实现的基础。通过这种方式，您将能够利用厂商提供的强安全性基础设施，其中，业务要求需要支持复杂的授权规则。（有关授权的更详细内容，请参见 [Ilechko]。）

其他常见的问题包括使用不遵循 Java EE 规范的持久性机制（这使得事务管理变得困难）、在 Java EE 程序中使用不适当的 J2SE 方法（例如线程或 singleton），以及使用您自己的方法解决程序到程序（program-to-program）的通信，而不是使用 Java EE 内在支持的机制（例如 JCA、JMS 或 Web 服务）。当您将一个遵循 Java EE 的服务器移植到其他的服务器上，或者移植到相同服务器的新版本上，上述的设计选择将会造成无数的问题。使用 Java EE 之外的元素，通常会导致一些细微的可移植性问题。唯一要背离规范的情况是，当一个问题在规范的范围内无法解决的时候。例如，安排执行定时的业务逻辑在 EJB2.1 出现之前是一个问题。在类似这样的情况下，我们建议当有厂商提供的解决方案时就使用厂商提供的解决方案（例如 WebSphere Application Server Enterprise 中的 Scheduler 工具），而在没有厂商提供的解决方案时就使用第三方提供的工具。当然，现在的 EJB 规范提供了基于时间的函数，所以我们鼓励使用这些标准接口。如果使用厂商提供的解决方案，应用程序的维护以及将其移植到新的规范版本将是厂商的问题，而不是您的问题。

最后，要注意不要太早地采用新技术。太过于热衷采用还没有集成到 Java EE 规范的其他部分或者还没有集成到厂商的产品中的技术常会带来灾难性的后果。支持是关键的——如果您的厂商不直接支持某种特定的技术，那么您在采用此技术时就应该非常谨慎。有些人（尤其是开发人员）过分关注于简化开发过程，忽略了依赖大量本组织之外开发的代码的长期后果，而供应商并不支持这些代码。我们发现，许多项目团队沉迷于新技术（例如最新的开放源代码框架），并很快地依赖于它，却没有考虑它对业务带来的实际代价。坦白地说，对于使用您的供应商所提供的产品之外的任何技术的决策，都应该由企业组织结构中的各个部门、业务团队和法律团队（或您的环境中的等同机构）仔细地进行评审，这与正常的产品购买决策完全相同。毕竟，我们中的大多数人是在解决业务问题，而不是推进技术的发展。

5. 从一开始就计划使用 Java EE 安全性。

启用 WebSphere 安全性。这使您的 EJB 和 URL 至少可以让所有授权用户访问。不要问为什么——照着做就是了。

在与我们合作的客户中，一开始就打算启用 WebSphere Java EE 安全性的顾客是非常少的，这一点一直让我们感到吃惊。据我们估计大约只有 50% 的顾客一开始就打算使用此特性。例如，我们曾与一些大型的金融机构（银行、代理等等）合作过，他们也没有打算启用安全性。幸运的是，这种问题在部署之前的检查时就得以解决。

不使用 Java EE 安全性是件危险的事情。假设您的应用程序需要安全性（几乎所有的应用程序都需要），敢打赌您的开发人员能够构建出自己的安全性基础设施，其比您从 Java EE 厂商那里买来的更好。这可不是个好的赌博游戏。为分布式的应用程序提供安全性是异常困难的。例如，您需要使用网络安全加密令牌控制对 EJB 的访问。以我们的经验看来，大多数自己构建的安全性基础设施是不安全的，并且有重大的缺陷，这使产品系统极其脆弱。（有关更详细的信息，请参考 [Barcia] 的第 18 章。）

一些不使用 Java EE 安全性的理由包括：担心性能的下降，相信其他的安全性（例如 IBM Tivoli® Access Manager 和 Netegrity SiteMinder）可以取代 Java EE 安全性，或者是不知道 WebSphere Application Server 安全特性及功能。不要陷入这些陷阱之中。尤其是，尽管像 Tivoli Access Manager 这样的产品能够提供优秀的安全特性，但是仅仅其自身不可能保护整个 Java EE 应用程序。这些产品必须与 Java EE 应用服务器联合起来才可能全面地保护您的系统。

其他一种常见的不使用 Java EE 安全性的原因是，基于角色的模型没有提供足够的粒度访问控制以满足复杂的业务规则。尽管事实是这样的，但这也不应该成为不使用 Java EE 安全性的理由。相反地，应该将 Java EE 验证及 Java EE 角色与特定的扩展规则结合起来。如果复杂的业务规则需要做出安全性决策，那就编写相应的代码，其安全性决策要基于可以直接使用的以及可靠的 Java EE 验证信息（用户 ID 和角色）。（有关授权的更详细的信息，请参见 [Ilechko]。）

6. 创建您所知道的。

反复的开发工作将使您能够逐渐地掌握所有的 Java EE 模块。要从创建小而简单的模块开始而不是从一开始就马上涉及到所有的模块。

我们必须承认 Java EE 是庞大的体系。如果一个开发团队只是开始使用 Java EE，这将很难一下子就能掌握它。在 Java EE 中有太多的概念和 API 需要掌握。在这种情况下，成功掌握 Java EE 的关键是从简单的步骤开始做起。

这种方法可以通过在您的应用程序中创建小而简单的模块来得到最好的实现。如果一个开发团队通过创建一个简单的域模型以及后端的持久性机制（也许使用的是 JDBC），并且对其进行了完整的测试，这会增强他们的自信心，于是他们会使用该域模型去掌握使用 Servlet 和 JSP 的前端开发。如果一个开发团队发现有必要使用 EJB，他们也会类似地开始在容器管理的持久性 EJB 组件之上使用简单的会话 Facade，或者使用基于 JDBC 的数据访问对象（JDBC-based Data Access Objects，DAO），而不是跳过这些去使用更加复杂的构造（例如消息驱动的 Bean 和 JMS）。

这种方法并不是什么新方法，但是很少有开发团队以这种方式来培养他们的技能。相反地，多数开发团队由于尝试马上就构建所有的模块，同时涉及 MVC 中的视图层、

模型层和控制器层，这样做的结果是他们往往会陷入进度的压力之中。他们应该考虑一些敏捷（Agile）开发方法，例如极限编程（XP），这种 开发方法采用一种增量学习及开发方法。在 XP 中有一种称为 ModelFirst[Wiki] 的过程，这个过程涉及到首先构建域模型作为一种机制来组织和实现用户场景。基本说来，您要构建域模型作为您要实现的用户场景的首要部分，然后在域模型之上 构建一个用户界面（UI）作为用户场景实现的结果。这种方法非常适合让一个开发团队一次只学到一种技术，而不是让他们同时面对很多种情况（或者让他们读很 多书），这会令他们崩溃的。

还有，对每个应用程序层重复的开发可能会包含一些适当的模式及最佳实践。如果您从应用程序的底层开始应用一些模式（如数据访问对象和会话 Facade），您就不应该在您的 JSP 和其他视图对象中使用域逻辑。

最后，当您开发一些简单的模块时，在开始的初期就可以对您的应用程序进行性能测试。如果直到应用程序开发的后期才进行性能测试的话，这往往会出现灾难性的后果，正如 [Joines] 所述。

7. 当使用 EJB 组件时，始终使用会话 Facade。

在体系结构合适的情况下，使用本地 EJB。

当使用 EJB 组件时，使用会话 Facade 是一个确认无疑的最佳实践。实际上，这个通用的实践被广泛地应用到任何分布式技术中，包括 CORBA、EJB 和 DCOM。从根本上讲，您的应用程序的分布“交叉区域”越是底层化，对小块的数据由于多次重复的网络中继造成的时间消耗就越少。要达到这个目的的方法是， 创建大粒度的 Facades 对象，这个对象包含逻辑子系统，因而可以通过一个方法调用就可以完成一些有用的业务功能。这种方法不但能够降低网络开销，而且在 EJB 内部通过为整个业务功能创建一个事务环境也可以大大地减少对数据库的访问次数。[Alur] 对这种模式进行了规范的表示,[Fowler](并且包括除 EJB 之外的情况)和 [Marinescu] 也对其进行了描述（请参见参考资料）。细心的读者会发现，这实际上正是面向服务的体系结构 (SOA) 中的核心原则之一。

EJB 本地接口（从 EJB 2.0 规范开始使用）为共存的 EJB 提供了性能优化方法。本地接口必须被您的应用程序显式地进行访问，这需要代码的改变和防止以后配置 EJB 时需要应用程序的改变。如果您确定 EJB 调用始终是本地的，那么可以充分利用本地 EJB 的优化。然而,会话 Facade 本身的实现(典型例子如无状态会话 Bean)应该设计为远程接口。通过这种方式，其他的客户端可以远程地使用 EJB 本身，而不会破坏现有的业务逻辑。因为 EJB 可以同时具有本地和远程接口，所以这是完全可以实现的。

为了性能的优化，可以将一个本地接口添加到会话 Facade。这样做利用了这样一个事实，在大多数情况下（至少在 Web 应用程序中），您的 EJB 客户端和 EJB 会共同存在于同一个 Java 虚拟机（JVM）中。另外一种情况是，如果会话 Facade 在本地被调用，可以使用 Java EE 应用服务器配置优化（configuration optimizations），例如 WebSphere 中的“No Local Copies”。然而，您必须注意到这些可供选择的方案会将交互方法从按值传递（pass-by-value）改变为按引用传递（pass-by- reference）。这可能会在您的代码中产生很微妙的错误。最好使用本地 EJB，因为对于每个 Bean 而言，其行为是可以控制的，而不会影响到整个应用服务器。

如果在您的会话 Facade 中使用远程接口（而不是本地接口），您也可以将同样的会话 Facade 在 Java EE 1.4 中以兼容的方式作为 Web 服务来配置。（这是因为 JSR 109，Java EE 1.4 中的 Web 服务部署部分，要求使用无状态会话 Bean 的远程接口作为 EJB Web 服务和 EJB 实现的接口。）这样做是值得的，因为这样做可以为您的业务逻辑增加客户端类型的数量。

8. 使用无状态会话 Bean，而不是有状态会话 Bean。

这样做可以使您的系统更经得起故障转移。使用 HttpSession 存储和用户相关的状态。

以我们的观点来看，有状态会话 Bean 的概念已经过时了。如果您仔细对其考虑一下，一个有状态会话 Bean 实际上与一个 CORBA 对象在体系结构上是完全相同的，无非就是一个对象实例绑定到一个服务器，并且依赖于服务器来管理其生命周期。如果服务器关闭了，这种对象也就不存在，那么 这个 Bean 的客户端的信息也就不存在。

Java EE 应用服务器为有状态会话 Bean 提供的故障转移能够解决一些问题，但是有状态的解决方案没有无状态的解决方案易于扩展。例如，在 WebSphere Application Server 中，对无状态会话 Bean 的请求，是通过对部署无状态会话的成员集群进行平衡加载来实现。相反地，Java EE 应用服务器不能对有状态 Bean 的请求进行平衡加载。这意味着您的集群中的服务器的加载过程会是不均衡的。此外，使用有状态会话 Bean 将会再添加一些状态到您的应用服务器上，这也是不好的做法。有状态会话 Bean 增加了系统的复杂性，并且在出现故障的情况下使问题变得复杂化。创建健壮的分布式系统的一个关键原则是尽量使用无状态行为。

因此，我们建议对大多数应用程序使用无状态会话 Bean 方法。任何在处理时需要使用的与用户相关的状态应该以参数的形式传送到 EJB 的方法中(并且通过使用一种机制如 HttpSession 来存储它）或者从持久性的后端存储（例如通过使用实体 Bean)作为 EJB 事务的一部分来进行检索。在合适的情况下，这个信息可以缓存到

内存中，但是要注意在分布式的环境中保存这种缓存所潜在的挑战性。缓存非常适合于只读数据。

总之，您要确保从一开始就要考虑到可扩展性。检查设计中的所有设想，并且考虑到当您的应用程序要在多个服务器上运行时，是否也可以正常运行。检查设计中所有 的假设，判断如果您的应用程序运行于多个服务器之上，它们是否依然成立。这个规则不但适合上述情况的应用程序代码，也适用于如 MBean 和其他管理接口的情况。

避免使用有状态性不只是对 IBM/WebSphere 的建议，这是一个基本的 Java EE 设计原则。请参见 [Jewell] 的 Tyler Jewell 对有状态 Bean 的批评，其观点和上述的观点是相同的。

9. 使用容器管理的事务。

学习一下 Java EE 中的两阶段提交事务，并且使用这种方式，而不是开发您自己的事务管理。容器在事务优化方面几乎总是比较好的。

使用容器管理的事务（CMT）提供了两个关键的优势（如果没有容器支持这几乎是不可能的）：可组合的工作单元和健壮的事务行为。

如果您的应用程序代码显式地使用了开始和结束事务（也许使用 javax.jts.UserTransaction 或者甚至是本地资源事务），而将来的要求需要组合模块（也许会是代码重构的一部分），这种情况下往往需要改变事务代码。例如，如果模块 A 开始了一个数据库事务，更新数据库，随后提交事务，并且有模块 B 做出同样的处理，请考虑一下当您在模块 C 中尝试使用上述两个模块，会出现什么情况呢？现在，模块 C 正在执行一个逻辑动作，而这个动作实际上将调用两个独立的事务。如果模块 B 在执行中失败了，而模块 A 的事务仍然能被提交。这是我们所不希望出现的行为。如果，相反地，模块 A 和模块 B 都使用 CMT 的话，模块 C 也可以开始一个 CMT（通常通过配置描述符），并且在模块 A 和模块 B 中的事务将是同一个事务的隐含部分，这样就不再需要重写复杂的代码了。

如果您的应用程序在同一个操作中需要访问多种资源，您就要使用两阶段提交事务。例如，如果从 JMS 队列中删除一个消息，并且随后更新基于这条消息的纪录，这时，要保证这两个操作都会执行或都不会执行就变得尤为重要。如果一条消息已经从队列中被删除，而 系统没有更新与此消息相关的数据库中的记录，那么这种系统是不一致的。一些严重的客户及商业纠纷源自不一致的状态。

我们时常看到一些客户应用程序试图实现他们自己的解决方案。也许会通过应用程序的代码在数据库更新失败的时候“撤销”对队列的操作。我们不提倡这样做。这种 实现要比您最初的想象复杂得多，并且还有许多其他的情况（想象一下如果应用

程序在执行此操作的过程中突然崩溃的情况）。作为替代的方式，应该使用两阶段提 交事务。如果您使用 CMT，并且在单一的 CMT 中访问两阶段提交的资源（例如 JMS 和大多数数据库），WebSphere 将会处理所有的复杂工作。它将确保整个事务被执行或者都不被执行，包括系统崩溃、数据库崩溃或其他的情况。其实现在事务日志中保存着事务状态。当应用程序 访问多种资源的时候，我们怎么强调使用 CMT 事务的必要性都不为过。如果您所访问的资源不支持两阶段提交，那么您当然就没有别的选择了，只能使用一种比较复杂的方法，但是您应该尽量避免这种情况。

10. 将 JSP 作为表示层技术的首选。

只有在需要多种表示输出类型，并且输出类型被单一的控制器及后端支持时才使用 XML/XSLT。

我们常听到一些争论说，为什么您选择 XML/XSLT 而不是 JSP 作为表示层技术，因为 JSP“允许您将模型和视图混合在一起”，而 XML/XSLT 不会有这种问题。遗憾的是，这种观点并不完全正确，或者至少不像白与黑那样分的清楚。实际上，XSL 和 XPath 是编程语言。事实上，XSL 是图灵完备的（Turing-complete），尽管它不符合大多数人定义的编程语言，因为它是基于规则的，并且不具备程序员习惯的控制工具。

问题是既然给予了这种灵活性，开发人员就会利用这种灵活性。尽管每个人都认同 JSP 使开发人员容易在视图中加入“类似模型”的行为，而实际上，在 XSL 中也有可能做出一些同样的事情。尽管从 XSL 中进行访问数据库这样的事情会非常困难，但是我们曾经见到过一些异常复杂的 XSLT 样式表执行复杂的转换，这实际上是模型代码。

然而，应该选择 JSP 作为首选的表示技术的最基本的原因是，JSP 是现在支持最广泛的、也是最被广泛理解的 Java EE 视图技术。而随着自定义标记库、JSTL 和 JSP2.0 的新特性的引入，创建 JSP 变得更加容易，并且不需要任何 Java 代码，以及可以将模型和视图清晰地分离开。在一些开发环境中（如 IBM Rational Application Developer）加入了对 JSP（包括对调试的支持）的强大支持，并且许多开发人员发现使用 JSP 进行开发要比使用 XSL 更加简单,主要是因为 JSP 是基于例程的,而不是基于规则的。尽管 Rational Application Developer 支持 XSL 的开发，但一些支持 JSP 的图形设计工具及其他特征（尤其在 JSF 这样的框架下）使得开发人员可以以所见即所得的方式进行 JSP 的开发，而使用 XSL 有时不容易做到。

然而，这并不表示您*绝不* 应该使用 XSL。在一些情况下，XSL 能够表示一组固定的数据，并且可以基于不同的样式表（请参见 [Fowler]）来以不同的方式显示这些数据的能力是显示视图的最佳解决方案。然而，这只是一种特殊的情况，而不是通用的规则。如果您只是生成 HTML 来表达每一个页面，那么在大多数情况下，XSL 是

一种不必要的技术，并且，它给您的开发人员所带来的问题远比它所能解决的问题多。

11. 当使用 HttpSession 时，尽量只将当前事务所需要的状态保存在其中，其他内容不要保存在 HttpSession 中。

启用会话持久性。

HttpSessions 对于存储应用程序状态信息是非常有用的。其 API 易于使用和理解。遗憾的是，开发人员常常遗忘了 HttpSession 的目的——用来保持临时的用户状态。它不是任意的数据缓存。我们已经见到过太多的系统为每个用户的会话放入了大量的数据(达到兆字节)。如果同时有 1000 个登录系统的用户，每个用户拥有 1MB 的会话数据，那么就需要 1G 或者更多的内存用于这些会话。保持这些 HTTP 会话数据较小。不然的话，您的应用程序的性能将会下降。一个大约比较合适的数据量应该是每个用户的会话数据在 2K-4K 之间。这不是一个硬性的规则。8K 仍然没有问题，但是显然会比 2K 时的速度要慢。一定要注意，不要使 HttpSession 变成数据堆积的场所。

一个常见的问题是使用 HttpSession 缓存一些很容易再创建的信息，如果有必要的话。由于会话是持久性的，进行不必要的序列化以及写入数据是一种很奢侈的决定。相反地，应该使用内存中的哈希表 来缓存数据，并且在会话中保存一个对此数据进行引用的键。这样，如果不能成功登录到另外的应用服务器的话，就可以重新创建数据。（有关更详细的信息，请参见 [Brown2]。）

当谈及会话持久性时，不要忘记要启用这项功能。如果您没有启用会话持久性，或者服务器因为某种原因停止了（服务器故障或正常的维护），则所有此应用服务器的 当前用户的会话将会丢失。这是件令人非常扫兴的事情。用户不得不重新登录，并且重新做一些他们曾经已经做过的事情。相反地，如果启用了会话持久 性，WebSphere 会自动将用户（以及他们的会话）移到另外一个应用服务器上去。用户甚至不知道发生了这样的事情。我们曾经见到过一些产品系统，因为本地代码中存在令人难以 忍受的错误（不是 IBM 的代码！）而经常崩溃，在这种情况下，上述功能仍然可以运行良好。

12. 充分利用应用服务器中不需要修改代码的特性。

使用某些特性（如 WebSphere Application Server 缓存和 Prepared Statement 缓存）可以极大地提高性能，并且使得开销最小。

前面的最佳实践 4 清楚地描述了这样一种案例，即关于为什么应该谨慎的使用可能修改代码的应用服务器特定的特性。它使得难以实现可移植性，并且可能给版本的迁移带来困难。然 而，特别是在 WebSphere Application Server 中，有一套应用服务器特定的特性，您可以并且应该充分地利用它们，因为它们不会修改您的代码。您应该按照规范来编写代码，但如果您了解这些特性以及如何正 确地使用它们，那么您就可以利用它们显著地改善性能。

作为这个最佳实践的一个示例，在 WebSphere Application Server 中，您应该开启动态缓存，并且使用 Servlet 缓存。系统性能可以得到很大的提高，而开销是最小的，并且不影响编程模型。通过缓存来提高性能的好处是众所周知的事情。遗憾的是，当前的 Java EE 规范没有包括一种用于 Servlet/JSP 缓存的机制。然而，WebSphere 提供了对页面以及片断缓存的支持，这种支持是通过其动态缓存功能来实现的，并且不需要对应用程序作出任何改变。其缓存的策略是声明性的，而且其配置是通过 XML 配置描述符来实现的。因此，您的应用程序不会受到影响，并保持与 Java EE 规范的兼容性和移植性，同时还从 WebSphere 的 Servlet 及 JSP 的缓存机制中得到性能的优化。

从 Servet 及 JSP 的动态缓存机制得到的性能的提高是显而易见的，这取决于应用程序的特性。Cox 和 Martin [Cox] 指出，对一个现有的 RDF（资源描述格式）站点摘要 (RSS) Servlet 使用动态缓存时，其性能可以提高 10%。请注意这个实验只涉及到一个简单的 Servlet，这个性能的增长量可能并不能反映一个复杂的应用程序。

为了更多地提高性能，将 WebSphere Servlet/JSP 结果缓存与 WebSphere 插件 ESI Fragment 处理器、IBM HTTP Server Fast Response Cache Accelerator (FRCA) 和 Edge Server 缓存功能集成在一起。对于繁重的基于读取的工作负荷，通过使用这些功能可以得到许多额外的好处。（请参见参考资料的 [Willenborg] 和 [Bakalova] 中描述的性能的提高。）

作为该原则的另一个示例（我们常常发现客户不使用它，仅仅是因为他们根本不知道它的存在），在编写 JDBC 代码时可以利用 WebSphere Prepared Statement Cache。在缺省情况下，当您在 WebSphere Application Server 中使用 JDBC PreparedStatement 时，它将对该语句进行一次编译，然后将其放到缓存中以便再次使用，不仅可以在创建 PreparedStatement 的同一方法中重用，还可以跨程序重用，只要其中使用了相同的 SQL 代码或者另一个 PreparedStatement。省去重新编译的步骤可以极大降低调用 JDBC 驱动程序的次数，并且提高应用程序的性能。要利用这个特性，您只需编写相应的 JDBC 代码以使用 PreparedStatements，而不需要进行任何其他工作。在编写代码时，使用 PreparedStatement 代替常规的 JDBC

Statement 类（它使用了纯的动态 SQL），您就可以实现性能的增强，而不会损失任何可移植性。

13. 充分利用现有的环境。

提供一个 Java EE EAR 和可配置的安装脚本，而不是黑盒二进制安装程序。

在大多数的实际场景中，大量的 WebSphere Application Server 用户在相同的共享单元中运行多个应用程序。这意味着，如果您提供一个需要安装的应用程序，那么它必须能够合理地安装到现有的基础设施中。这意味两个方面： 首先，您必须限制关于环境的假设的数目，并且因为您无法预料到所有的情况，所以您的安装过程必须是可见的。这里所说的可见是指，不应该提供二进制可执行文 件形式的安装程序。执行安装任务的管理员需要清楚安装过程对他们的单元所进行的操作。为了实现这种方式，您应该提供一个 EAR 文件（或者一组 EAR 文件）以及相关的文档和安装脚本。这些脚本应该具有可读性，以便安装程序能够知道它们需要执行的操作，并对脚本的内容进行验证以确保不会执行任何危险的操 作。在有些情况下，脚本并不合适，用户可能需要使用一些曾用过的其他方法来安装 EAR，这表示您必须记录安装程序所完成的工作！

14. 充分利用应用服务器环境所提供的服务质量。

设计可使用 WebSphere Application Server Network Deployment 集群的应用程序。

我们已经介绍了利用 WebSphere Application Server 安全和事务支持的重要性。还有一个更重要的、常常被我们忽视的问题，即集群。需要将应用程序设计为能够运行于集群的环境。大多数实际的环境需要通过集群来 实现可扩展性和可靠性。无法进行集群的应用程序很快会导致灾难的出现。

与集群紧密相关的是支持 WebSphere Application Server Network Deployment。如果您正在构建一个应用程序并打算将它卖给其他人，请确保您的应用程序可以运行于 WebSphere Application Server Network Deployment，而不仅仅是单个服务器版本。

15. 利用 Java EE，不要欺骗。

致力于构建真正利用 Java EE 功能的 Java EE 应用程序。

有件非常烦人的事情我们曾多次遇到过，某个应用程序声称可以运行于 WebSphere 中，但它并不是一个真正的 WebSphere 应用程序。我们曾见过几个这样的示例，其中有一小段代码（可能是一个 Servlet）位于 WebSphere Application Server 中，而其余所有的应用程序逻辑实际上位于单独的进程中，例如一个以 Java、C、C++ 或其他语言（没有使用 Java EE）编写的守护进程负责完成实际的工作。这并不是一个真正的 WebSphere Application Server 应用程序。对于这样的应用程序，WebSphere Application Server 所提供的几乎所有的服务质量都不可用。对于那些认为这是 WebSphere Application Server 应用程序的人来说，他们会突然的醒悟过来，原来并非如此。

16. 安排进行版本更新。

更改是在所难免的。安排新的发行版和修复程序更新，以便您的客户能够获得最新的版本。

WebSphere Application Server 在不断地发展，所以 IBM 定期地给出 WebSphere Application Server 的修复程序，这是很正常的，并且 IBM 还定期地发布新的主要版本。您需要为此做好安排。这会影响到两类开发组织：内部开发人员和第三方应用程序供应商。基本的问题是相同的，但对两者的影响则有 所不同。

首先考虑修复程序。IBM 定期发布建议更新， 以修复产品中已发现的错误。尽管不太可能始终运行于最新的级别，但请注意，不要隔得太久。那么究竟“隔多久”是可以接受的呢？对于这个问题没有什么正确的 答案,但是您应该安排好对几个月内的发行版进行修复级别更新。是的，这表示一年要更新好几次。内部开发人员可以忽略某些修复级别，一次仅支持一个修复级 别，以降低测试成本。应用程序供应商则没有这么幸运。如果您是应用程序供应商，那么您同时需要支持多种修复级别，以便您的客户能够将您的软件与其他软件一 同运行。如果您仅支持一种修复级别，那么很可能无法找到同时兼容于多种产品的修复级别。实际上对于供应商而言，最好的方法是使用支持“向上兼容修复程序” 的模型。IBM 使用了这种方法来支持所集成的来自其他供应商的产品（如 Oracle®、Solaris™ 等等）。有关更详细的信息，请参考我们的支持策略。

下面再考虑一下主要版本更新。IBM 定期地发布新的主要发行版，其中对我们的产品进行了主要的功能更新。我们暂时继续支持旧的主要发行版，但不会太久。这意味着您必须安排从一个主要发行版转 到另一个主要发行版。这是不可避免的，并且应该在您的成本模型中加以考虑。如果您是供应商，这意味着您必须经常地对您的

产品进行更新，以支持新的 WebSphere Application Server 版本，否则您的客户将停滞于不受支持的 IBM 产品，我们曾多次碰到过这种情况！如果您正从供应商处购买产品，我们鼓励您要留心您的供应商，以确保他们承诺支持 IBM 产品新的版本。停滞于不受支持的软件是一种非常危险的情况。

17. 在代码中所有关键的地方，使用标准的日志框架记录程序的状态。

这包括异常处理程序。使用像 JDK 1.4 Logging 或 Log4J 这样的日志框架。

有些时候，日志记录是最乏味的工作，降低了编程的价值，但是这样做可以减少调试的时间，并尽快地完成相应的任务。根据一般的经验，在每个过渡的地方，需要进 行日志记录。当您将参数从一个方法传递到另一个方法，或从一个类传递到另一个类，需要进行日志记录。在对一个对象进行某种转换时，需要进行日志记录。在碰 到不解之处时，需要进行日志记录。

在决定了进行日志记录之后，需要选择一种合适的框架。实际上有许多选择，但是我们偏爱 JDK 1.4 Trace API，因为它们已全面地集成到了 WebSphere Application Server 跟踪子系统中，并且是基于标准的。

18. 在完成相应的任务后，请始终进行清理。

如果您从池中获取了一个对象，请始终确保将其返回到池中。

无论运行于开发、测试或生产环境中，我们发现 Java EE 应用程序最常见的错误之一是内存泄漏。绝大部分情况是因为开发人员忘了关闭连接（大多数情况下是 JDBC 连接）或将对象返回到池中。对于任何需要显式关闭的或需要返回到池中的对象，请确保进行了这样的操作。不要编写出这样糟糕的代码。

19. 在开发和测试过程中遵循严格的程序。

这包括采用和遵循软件开发方法学。

大型系统的开发是非常困难的，所以应该十分谨慎。但是，我们常常发现一些团队疏于管理、或者不能全心全意地遵循相关的开发方法（这些方法可能不适用于他们正 在进行的开发类型）、或者他们并没有很好地理解这一点。最为糟糕的可能是尝试每个月更换不同的开发方法，在单个项目的生命周期中，一个团队从 RUP 改变为 XP，以及一些其他敏捷方法。

总之，对于大多数团队而言，只要团队成员能够很好地理解、严格地执 行、并根据特定的技术本质和使用该方法的团队进行适当的调整，那么几乎任何一种方法都是有效的。对于那些尚未采用任何方法、或者那些不能够完全地利用所选 方法的团队，我们建议他们参考一些优秀的著作，如 [Jacobson]、[Beck1] 或 [Cockburn]。另一个有价值的信息来源是最近公布的用于 Eclipse Process Framework [Eclipse] 的 OpenUP 插件。对于这个已经介绍过的主题，我们不想做过多的重复，建议读者参考 [Hambrick] 和 [Beaton2]。

20. 开发的最佳实践 （重点）

20.1 开发的整体考虑

- 整体考虑
  - 开发应遵循标准的软件工程方法论
  - 开发应遵循 J2EE 规范
  - 开发尽量使用成熟的框架以及开发模式
  - 变量名、类名、对象名、包名应该遵循命名规范

20.2 对象的构造的最佳实践

- 尽量避免在被经常调用的代码中创建对象
- 对于集合类(collection),应尽量初始化它的大小
  - JVM 会自动指定缺省大小
  - 如果超过，JVM 会重新创建一个，释放掉原来的对象，加大 JVM 负担
- 当一个类的多个实例在其本地的变量里访问一个特定的对象时，最好将这个变量设计为静态(static)的，而不是每个实例中变量里都存放那个对象的引用尽量重用对象的引用，而不是 new
- 注意释放容器对象中所保存的指向别的对象的引用
- 尽量使用 primitive 数据类型
- 当只是访问一个类的某个方法时,不要创建该类的对象,而是将该方法设计成一个 static 的方法。
- 尽量简化类的继承关系和设计简单的构造函数
- 创建简单数据类型的数组要比初始化一个这样的数组快，创建一个复杂类型的数组要比克隆一个这样的数组快

20.3 String& StringBuffer 开发的最佳实践

- 事实：String 对象是不可改变的
- 如果字符串在程序中可能被改变，比如增加、接或删除字符，就应使用 StringBuffer。创建具有初始大小的 StringBuffer 对象，尽量重用该对象，而不使用“+”操作
- 分析字符串中的字符时，就不要使用 String 或 StringBuffer，而是使用字符数组，特别是在循环中分析字符时更应该如此
- 尽量少用 StringTokenizer,它的方法的性能比较差

20.4 输入输出（Input/Output）

- 小块小块的读写数据会非常慢，因此，尽量大块的读写数据
- 使用 BufferedInputStream 和 BufferedOutputStream 来批处理数据以提高性能
- 对象的序列化(serialization)非常影响 I/O 的性能，尽量少用
- 对不需序列化的类的域使用 transient 关键字，以减少序列化的数据量

20.5 循环（Loop）

- 循环常量，在循环中它的值不会改变，因此，它的值应该在循环外先计算出来。
- 本地变量(Local Variable)，在方法中使用本地变量比使用对象的属性消耗较少的资源。在循环中却不一样，因为循环中的代码要反复地被运行，因此，尽量少地在循环中创建对象和变量。
- 尽早结束循环，如果循环体在满足一定条件就可以结束，就应尽快结束。

20.6 集合类（Collections）

- Collection 这是集合类的基本接口，它为一组对象提供了一些简单的方法，
- List 具有可以控制的顺序，但并没有定义或限制按什么排序
- Set 不能包含重复的元素
- Map 将一个键(Key)影射到一个值(Value),不允许有重复的键
- Vector 和 ArrayList
  - Vector 的方法都是同步的(Synchronized),是线程安全的(thread-safe)ArrayList 的方法不是，由于线程的同步必然要影响性能，因此,ArrayList 的性能比 Vector 好.当 Vector 或 ArrayList 中的元素超过它的初始大小时，Vector 会将它的容量翻倍，而 ArrayList 只增加 50%的大小，这样，ArrayList 就有利于节约内存空间
- Hashtable 和 HashMap
  - 类似 Vector 和 ArrayList，比如 Hashtable 的方法是同步的，而 HashMap 的不是

- ArrayList 和 LinkedList
  - ArrayList 的内部实现是基于内部数组 Object[]
  - LinkedList 的内部实现是基于一组连接的记录，更像链表结构
  - 当操作是在一列数据的后面添加数据而不是在前面或中间，并且需要随机地访问其中的元素时，使用 ArrayList 会提供比较好的性能
  - 当操作是在一列数据的前面或中间添加或删除数据，并且按照顺序访问其中的元素时，就应该使用 LinkedList 了
  - 如果两种情形交替出现，可以考虑使用 List 这样的通用接口，而不用关心具体的实现，由实现去保证性能

20.7. 方法（Method）

- Java 编译器会把一些代码转为代码嵌入，提高性能
- Final、static、private

20.8 同步（Synchronized）

- 使用同步方法比使用非同步方法的性能要低
- 应尽量少使用同步方法
- 同步方法的代码本身就不应该再同步了

20.9 EJB 部分

- 为 EJB 实现本地接口
- 始终通过会话 Bean 访问实体 Bean
- 尽量缓存对 EJB Home 的访问
- 当使用 EJB 组件时，始终使用会话 Facades
- 尽量使用无状态会话 bean，而不是有状态会话 bean
- 尽量使用容器管理的事务

20.10 Servlet 开发的最佳实践

- 让应用以 Servlet 为中心，而不是用 JSP 为中心

绝大多数采用以 servlet 为中心，以 JSP 为中心可以用于 2-3 页的小应用典型地情况下 JSP 中有太多的 Java 代码，应该避免 JSP 中有太多的 Java 代码。在业务处理之后最好让 servlet 决定使用哪一个 JSP，可以使用 JSP 重定向，但是容易引起混乱

- init()做尽可能多的工作
  - 只在初始化是执行一次

- 尽可能少地使用 Synchronize
  - 确保不要 synchronize 整个类
- 不要用任何实例变量
  - 被所有在线的用户使用
- 不要使用 SingleThreadModel
  - 虽然是 thread-safe，但性能太差…
- 对于非 Java 群体：不要在 cookies 中存对象
- 将业务逻辑放在 Servlet 之外，如放在传统 java 类
- HttpServlet 的子类应该仅做 servlet 份内的工作

. 管理 request、response 及 HttpSession 对象

. 仅传送普通的 Java 类，不是 servlet 相关的类（例如 request, response 或 session），少用或不用共享类变量

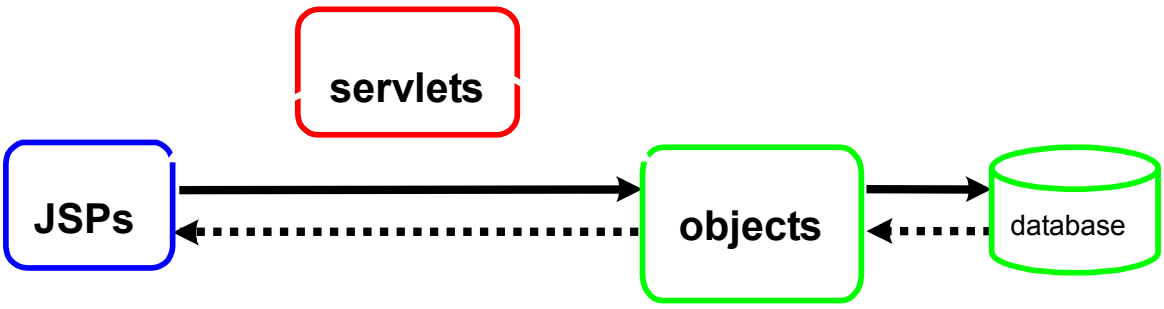
. 比 servlet 容易开发、测试和重用

- 将业务逻辑写在传统的 Java 类

. 仅传送普通的 Java 类，不是 servlet 相关的类（例如 request, response 或 session）

. 少用或不用共享类变量

. 比 servlet 容易开发、测试和重用



20.11 JSP 开发的最佳实践

- JSP 部分

- 保持尽量少的 Java 代码
  - JSP 中的 Java 代码很难维护和测试
  - 绝对不要将业务逻辑放在 JSP 中
  - 编写 helper classes 从数据生成 HTML
    - One of the few times putting hardcoding HTML in Java is 'okay'
  - 理想情况下只使用 "<%= ... %>" tags…
- 在每一个 JSP 只包含用于显示数据的简单对象
  - 有时称之为 "view bean", 但不用是一个真的 JavaBean
  - 限定 JSP 只做显示工作, 抵制在其中编写的商业逻辑的诱惑
- 如果在页面中你要共享组件和项目, 使用如下表格
  - E.g. 对于菜单, 标题栏, 页脚, 等等
  - 可以使用 HTML 或者 JSP include 指令
  - 可以包括静态 HTML, 或者其它 JSP
- 不要忘记在整个 JSP 中使用 try/catch
  - 如果在 JSP 中抛出一个异常, 它不能被 servlet 捕获
  - 作为替代, 将所有有异议的代码放在头部, 将这一部分封装在 try/catch 中

20. 12 数据库访问部分

- 数据库访问部分
  - 规格化（Normalization）数据库结构
  - 针对常用的 SQL 操作建立索引，删除多余的索引
  - 尽量使用 Prepared Statements
  - 合理运用 PreparedStatement 和连接池
  - 考虑批量执行 SQL 命令
  - 考虑使用数据库存储过程
  - 及时关闭不用的 Statement、ResultSet、Connection 等对象（但不是在 finalize 方法内）
  - 一定要在 Finally 中 close 数据库连接
  - 事务尽量短，平，快
  - 检查 JDBC driver 的版本。
  - 尽量数据源连接池 Use Connection pool
  - 控制事务 Control transaction
  - 选择好最优的事务隔离级别 Choose optimal isolation level

TRANSACTION_READ_UNCOMMITED 提供为大多数并发的事务型应用提供最好的并发性能

  - 一旦结束就要关闭连接
  - 不要传递连接
  - 优化 Statment

下面是使用 Statement 接口来提高性能的一些方法：

。 选择正确的 Statement 的接口

JDBC 提供三种 Statement 类型

Statement: 如果没有输入输出参数使用静态的 sql statement （性能差）

PreparedStatement:如果有输入参数用动态的 sql statement. （性能好）

CallableStatement:如果有输入输出参数用动态的 sql statement. （性能最好，但可能过多依赖数据库的存储过程，不好移植）

- 做批量的 update

。 用 Statement 来批量 retrieval

由于可以发送多个 sql,减少了 jdbc 调用的次数从而提高了性能。

例子：

```
statement.addBatch( "sql query1");
statement.addBatch(" sql query2");
statement.addBatch(" sql query3");
statement.executeBatch();
```

使用 Statement.setFetchSize(30)来限制返回的结果集，

。 结束后就关闭 Statement.

一结束后关闭 ResultSet.

优化 SQL 查询

在 sql 中只查询你想要的

不好的例子

```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("select * from employee where name=RR");
```

性能好的例子

```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("select salary from employee where name=RR");
```

- 缓存 read-only and read-mostly 的数据

20.13 HttpSession 访问部分

- 确保放在 session 中的每一个对象要实现 Serializable (or Externalizable)
  - 不是必须的，但在分布式应用中是一个好的方法
- 避免在 session 中存放大对象
- 手工使 session 无效
  - Can only do this if application has explicit ‘log out’ function
- 如果在 JSP 中不需要 session，禁用它

<%page session=“false”%>

20.14 其它访问部分

- JSP 中保持尽量少的 Java 代码
- 尽可能减少 HTTP 数据传输的总量和频度
- 尽量使用局部变量
- 不要重复初始化变量
- 只有在必要时才运用线程安全的类
- 在 servlet、EJB 中不要使用多线程
- 尽量使用日志记录框架类如 log4j 输出跟踪信息，而不是使用简单的 System.out.println()
- 从正式发行的软件中删除调试信息

20.15 防止 OutOfMemory 的最佳实践

1. 对所有 sql 的返回的结果集一定要做限制，不要用 select *之类的
2. 对于 hashtable 上放的对象要显示地 remove
3. 不要在单例模式中用 hashtable.
4. 如果一个对象不再引用的话，要释放它
5. 释放内存

内存泄漏导致应用程序的大小随时间不断增长，该问题在 Java 应用程序中尤为突出，因为消耗完堆空间后将触发 JVM 的垃圾收集。使用上文讨论的系统监视工具或性能监视工具监视应用程序的内存使用情况，并检查代码以确保在处理完分配的对象后总是能够释放它们。一些内存泄漏问题常常不易发觉，比如 Hashtable 和 Vector 之类的 Java Collections 类中的内存泄漏，这些类在删除了所有其他引用后仍然会保持对对象的引用。

- Avoid creating objects in a loop.

20.16 防止数据源满的最佳实践

1. 检查 jdbc driver 的版本
2. 尽量使用 TYPE 4 的数据源
2. 防止连接池满
3. 在 finally 在关闭数据源的连接
4. 一旦结束就及时关闭 statement,resultset,connection
5. 在 servlet 的初始方法中 lookup 数据源的 jndi
6. 尽量避免在循环体中包括 sql 或者数据源的连接
7. 在同一个方法中获得和关闭连接
8. 不要在不同事务中用相同的连接
9. 用 prepareStatement，不要只用 Statement.,变量参数是?号来传递
10. 运用 CallableStatement 来调用存储
11. 关闭 close Resultset,Statement,Connection
12. 用 finally block and handle Exception

例子：

- 问题一：某些 jdbc 语句过于庞大，导致该语句的执行时间过长而锁住了 jdbc 连接等资源不被释放。
- 解答：建议检查 sql 语句的效率，把一个语句中执行的复杂操作优化成效率高的 sql 语句来执行。

原来：

select sqlb, currenthj, currentstate, sum(jjbs0), sum(jjbs1), sum(jjbs2), sum(tbbs1), tbdwbh from Todolist where sqlb='101' and **(currenthj='001' or currenthj='006' or currenthj='011' or .....))** and (clr='st001' or clr is null) and (tbdwbh like '44%') group by sqlb,currenthj,currentstate,tbdwbh order by sqlb,currenthj,currentstate,tbdwbh

改为：

select sqlb, currenthj, currentstate, sum(jjbs0), sum(jjbs1), sum(jjbs2), sum(tbbs1), tbdwbh from Todolist where sqlb='101' and **(currenthj in ('001','006','011','016','021','031I1','031I2','031I3','031J1','031J2','031J3','031K1','031K2','031K3','031M1','031M2','031M3','041','050','052','056','061','062','063','066','071','076','078','081','086','087','088','089','090','091','092','096','101','106','108','111'))** and (clr='st001' or clr is null) and (tbdwbh like '44%') group by sqlb,currenthj,currentstate,tbdwbh order by sqlb,currenthj,currentstate,tbdwbh

- 问题二：要珍惜 jdbc 连接资源
- 解答：jdbc 连接是非常宝贵的资源，当要用到的时候才去获取，而不要占住了一个连接却不使用它，也不要使用完了不释放它，这样会使应用程序的效率很低。总结来说，就是在初始化的时候对 datasource 作一次 jndi lookup 的动作，然后接下来当要使用 jdbc 连接的时候通过 getConnection 获取连接，在使用完毕后通过 conn.close()关闭，使之返回到连接池中被其他代码使用。

20.17 减少显式垃圾收集的次数

如今的 JVM 在何时需要执行垃圾收集以及收回哪些对象方面表现得十分智能。同时，垃圾收集是一项 JVM 操作，对性能会产生很大的影响。如果需要执行显式的垃圾收集来降低应用程序内存使用，那么尝试在低负载情况下或在非繁忙时刻实现垃圾收集。

20.18 尽可能缓存对象

- 可以被缓存并可以被所有用户共享
  - InitialContext object
    - JNDI

- EJB Home interfaces
  - 所有用户都一样
- DataSource

20.19. 开发的一些例子分析

- 无意识的集合对象保留

```
Vector v=new Vector(10);
for (int i=1; i<100; i++) {
    Object o=new Object();
    v.add(o);
    o=null;
}
```

问题：上面用的 Vector 中的临时对象 Object 被释放了吗？
回答：没有！

- 显示的赋 Null 值起作用吗？
  - 定义：赋空变量是指简单地将 null 值显式地赋值给这个变量，相对于让该变量的引用失去其作用域
  - 最佳实践
    - 正确地设置变量的作用域，而不要显式地赋空它们
    - 显式赋空变量一般应该没有影响
    -
    - 但是，在一些场合下会对性能产生巨大的负面影响
      - 例如，迭代的或者递归的赋空集合内的元素使得这些集合中的对象能够满足垃圾收集的条件
      - 实际上是增加了系统的开销而不是帮助垃圾收集
  - 所以前面的例子中，怎么样才是相对比较好的办法让 JVM 把 Vector 里面的申请的 Object 回收？
    - 正解：

```
For (int i=1; i<100; i++) {
v.remove(o);
}
```

  - 局部作用域
    - 一个例子

```
public static String scopingExample(String name) {
   StringBuffer sb = new StringBuffer();
   sb.append( “hello ”).append(name);
   return sb.toString();
}
```

- 问题：当方法执行完后，String Buffer 能被回收吗？

- 静态作用域
  - 一个例子

```
static StringBuffer sb = new StringBuffer();
public static String scopingExample(String name) {
   sb = new StringBuffer();
   sb.append("hello ").append(name);
   sb.append(", nice to see you!");
   return sb.toString();
}
```

- 问题：当方法执行完后， String Buffer 能被回收吗？

- 例子一：当该方法执行时，运行时栈保留了一个对 StringBuffer 对象的引用，这个对象是在程序的第一行产生的。在这个方法的整个执行期间，栈保存的这个对象引用将会防止该对象被当作垃圾。当这个方法执行完毕，变量 sb 也就失去了它的作用域，相应地运行时栈就会删除对该 StringBuffer 对象的引用。于是不再有对该 StringBuffer 对象的引用，现在它就可以被当作垃圾收集了。栈删除引用的操作就等于在该方法结束时将 null 值赋给变量 sb
- 例子二：现在 sb 是一个静态变量，所以只要它所在的类还装载在 Java 虚拟机中，它也将一直存在。该方法执行一次，一个新的 StringBuffer 将被创建并且被 sb 变量引用。在这种情况下，sb 变量以前引用的 StringBuffer 对象将会死亡，成为垃圾收集的对象。也就是说，这个死亡的 StringBuffer 对象被程序保留的时间比它实际需要保留的时间长得多－如果再也没有对该 scopingExample 方法的调用，它将会永远保留下去

由于在应用中在一个循环体中写了大量的静态的 sql，导致 pacage 用完
解决办法
1. 不在循环体中过多执行 sql
2. 避免使用 raw sql,而应该是 prepare statement，或者是 call statement

由于应用没有关闭结果集导致了 805 的问题