# IBM WebSphere 開發人員技術期刊：Java EE 重要最佳實務做法

等級：入門

Keys Botzum，IBM 資深技術人員
Kyle Brown，IBM 傑出工程師
Ruth Willenborg，IBM 資深技術人員，
Albert Wong，IBM I/T 架構設計師

2007 年 1 月 24 日

2004 年《IBM® WebSphere® 開發人員技術期刊》(IBM® WebSphere® Developer Technical Journal) 曾發表類似標題的文章，本文爲該文章之更新版，不僅考量到瞬息萬變的技術趨勢，更重要的是，作者發現某些做法一般應加以遵守，但卻常被忽略，因此於本文加以介紹。

由《IBM WebSphere 開發人員技術期刊》提供。

## 前言

過去將近十年來，許多文獻都已談到 Java™ Platform Enterprise Edition (Java EE) 的最佳實務做法。現在市面上也有許多書籍及數百篇（也許更多）文章，詳細探討如何撰寫 Java EE 應用程式。事實上，這方面的參考資料實在太多，其中往往夾雜矛盾的建議，以致光是參考這些混亂內容的過程，也成了採用 Java EE 的障礙。因此，爲了提供入門客戶一些簡單指引，我們編輯了這份最理想的做法清單，列出我們覺得重要有效的 Java EE 最佳實務做法。然而，儘管我們一心想詳加說明，卻無法在簡要的十大最佳實務清單中詳細解說一切。因此，爲避免遺漏重要的最佳實務，並推動 Java EE 的發展，我們列出了不可或缺的 Java EE「前十九大」最佳實務做法。

## 最佳實務做法

1. 一律使用 MVC。

2. 別再從新摸索。
3. 在每個層級應用自動化的單元測試及測試管理。
4. 依規格開發，而非應用程式伺服器。
5. 打從一開始就規劃使用 Java EE 安全性。
6. 建置自己熟悉的內容。
7. 使用 EJB 元件時一律用 Session Facade。
8. 使用 Stateless Session Bean 取代 Stateful Session Bean。
9. 使用儲存器管理交易。
10. 以 JSP 爲顯示技術的首選。
11. 使用 HttpSessions 時，儘量只儲存目前商業交易需要的狀態，不存多餘的部分。
12. 善用應用程式伺服器的功能，不需要修改程式碼。
13. 充分利用現有環境。
14. 充分利用應用程式伺服器環境提供的服務功能。
15. 充分利用 Java EE，別做表面功夫。
16. 規劃版本更新。
17. 在程式碼的所有重點處，都使用標準記載架構記錄程式狀態。
18. 完成相關作業後，一定要自行清理收尾。
19. 遵循嚴格的開發及測試程序。

## 1. 一律使用 MVC。

**明確區分商業邏輯（Java Bean 和 EJB 元件）、控制器邏輯（Servlets/Struts 動作）、顯示方式（JSP、XML/XSLT）。良好的分層方式可避免許多錯誤。**

這是成功採用 Java EE 的重要做法，當屬首要秘訣。模型視圖控制器 (Model-View-Controller, MVC) 是理想 Java EE 應用程式的設計基礎，可直接將程式設計工作分爲下列三部分：

1. 商業邏輯部分（模型：通常使用 Enterprise JavaBeans™ 或一般舊式 Java 物件來執行）。
2. 顯示使用者介面的部分（視圖）。
3. 應用程式導覽部分（控制器通常與 Java Servlets 或相關類別一同實作，如 Struts 控制器）。

市面上有許多關於 Java EE 的優秀評論，我們特別推薦有興趣的讀者參考 [Fowler] 或 [Brown] 的文章（請見「參考資料」），以詳盡深入瞭解。

如果不遵守基本的 MVC 架構，可能會導致許多問題。多數問題發生的原因，在於架構的「視圖」部分輸入太多資料。使用 JSP 標籤庫執行資料庫存取，或在 JSP 中執行應用程式流程控制，都是小型應用程式的常見做法，但這種做法卻會導致後續開發問題，因爲 JSP 會越來越難維護及除錯。

同樣地，我們也經常看到將「視圖」層結構移到商業邏輯的做法，例如，將建構視圖所用的 XML 剖析技術納入商業層，便是常見的問題。商業層應該在商業物件上運作，而不是在視圖相關的特定資料呈現上運作。

然而，單是有適當元件並不能正確進行應用程式分層。常見的情況是，應用程式同時擁有 Servlets、JSP 和 EJB 三種元件，其中多數商業邏輯是在 Servlet 層執行，或以 JSP 處理應用程式導覽。您必須嚴格執行程式碼複查及程式重構，確定商業邏輯只在「模型」層中處理、應用程式導覽僅在「控制器」層中執行，而「視圖」的作業只是將模型物件譯成適當的 HTML 及 Javascript™。

現在本項建議的重要性，應該比本文舊版發表時更加明確了。由於使用者介面技術快速變遷，而連結商業邏輯與使用者介面的做法也改變了「只是介面」的原貌，這兩種趨勢對現有系統影響至深。幾年前，Web 應用程式的使用者介面開發人員，還可以選擇 Servlets 與 JSP、Struts 或是 XML/XSL 轉換。從那時開始，Tiles 和 Faces 便廣受青睞，如今 AJAX 的支持者也越來越多。如果每次偏好的使用者介面技術有所變更，就必須重新開發應用程式的核心商業邏輯，這實在很糟糕。

## 2. 別再從新摸索。

**使用一般的可靠架構，如 Apache Struts、JavaServer Faces 及 Eclipse RCP。使用可靠的模型。**

回想當初開始協助訓練客戶使用當時才剛興起的 Java EE 標準時，我們發現（就像其他人一樣），開發使用者介面開發架構，可大幅提升開發人員直接根據基礎 Servlet 及 JSP 規格來建置 UI 應用程式的工作效率。因此，許多企業已開發本身的 UI 架構來簡化介面開發工作。

隨著 Apache Struts 等開放原始碼架構的發展 [Brown]，這些新架構一定會立即獲得採用。我們認爲，開發人員能立即明顯感受到有開放原始碼社群支援架構的好處，而且這些好處很快就能獲得全面接受，不僅用於新的開發作業，還可修改

應用程式。

然而，令人意外的是，事實並非如此。許多企業還在維護、甚至開發功能相當於 Struts 或 JSF 的新使用者介面架構。可能原因如下：組織惰性、「不是自己發明的」排外情結、沒有意識到改變工作程式碼的好處，或甚至有點傲慢地認爲，自己的方式會比開放原始碼開發人員運用特定架構的方式「更好」。

然而，這些理由都已經過時了，不應該當作不採用標準架構的藉口。如今，Struts 及 JSF 不僅在 Java 界廣受歡迎，WebSphere 執行時期和 Rational® 工具套組也已充分支援。同樣地，在豐富型用戶端領域，Eclipse RCP (Rich Client Platform) 也成爲建置獨立式豐富型用戶端的常用工具。儘管這些架構不屬於 Java EE 標準，但現已是 Java EE 社群的一份子，應該同樣獲得採用。

那些不使用現成 UI 架構的狂妄份子，應該記取 [Alur] 及 [Fowler] 所說明的教訓。這兩本書詳細說明了 Enterprise Java 應用程式中常見的可重複使用模型。從 Session Facade 等簡單模型（稍後的建議會再探討）到較複雜的模型，如 Fowler 的永續模型（已在許多開放原始碼永續模型架構中執行），這些成果都展現了 Java 前輩所累積的智慧。借用哲學家 Santayana 的名言，無法記取教訓的人，註定要重蹈覆轍（如果他們夠幸運，在失敗之後還機會重來的話）。

## 3. 在每個層級應用自動化的單元測試及測試管理。

**千萬別只測試 GUI。分層測試可大幅簡化除錯及維護作業。**

過去數年，方法大幅改變，Agile 新輕巧型方法（如「參考資料」中的 SCRUM [Schwaber] 及 Extreme Programming [Beck1]）越來越普及。這些方法幾乎都有一個特徵：都使用自動化測試工具，協助開發人員縮短回歸測試的時間，及避免不當回歸測試所造成的錯誤，以提升程式設計師的生產力。事實上，Test-First Development [Beck2] 這種方法更進一步延伸原來的做法，在開發實際程式碼之前，先撰寫單元測試。但在測試程式碼之前，必須先分成可測試的片段。由於這個方法不會執行單一輕鬆識別功能，因此一「大堆」程式碼並不容易測試。如果程式碼的每區段都執行多項功能，就很難測試每個單元的正確性。

MVC 架構（及 MVC 的 Java EE 實作）的優點之一，就是將各元素元件化之後，即可（事實上較爲簡單）分批測試應用程式。因此，您可輕鬆撰寫測試，分別測試持續性、Session Bean，以及其餘程式碼庫之外的使用者介面部分。市面上有許多 Java EE 測試架構及工具可簡化此流程。例如，junit.org 開發的開放

原始碼工具 JUnit，Apache 聯盟的開放原始碼專案 Cactus，這兩種工具都有助於測試 Java EE 元件。[Hightower] 會詳細探討如何在 Java EE 中使用這些工具。

儘管有這麼多徹底測試應用程式的有用資訊，我們還是發現許多人認爲，只要測試 GUI（可能是 Web 型態的 GUI 或獨立式 Java 應用程式），就相當於詳細測試整個應用程式。其實，光靠 GUI 測試並不夠，其中幾個原因如下。

1. 使用 GUI 測試，很難測試到系統的每個路徑；GUI 只是影響系統的其中一種方法，可能還有背景工作、Script 及各種其他存取點需要測試，但這些項目通常沒有相關的 GUI。
2. GUI 層級的測試十分粗略；GUI 會測試系統整體的表現，也就是說如果找到問題，就要檢查整個子系統，以致要尋找所有已識別的錯誤變得十分困難。
3. GUI 測試通常必須等到開發後期才能有效執行，因爲那時才充分定義 GUI。換句話說，要等到很後期才能有系統地發現潛在錯誤。
4. 一般開發人員可能無法存取自動 GUI 測試工具，因此，在他作了一項變更後，卻沒有簡單方法可以讓他重新測試受影響的子系統，這確實會妨礙理想的測試工作。如果開發人員可存取自動化的程式碼層級單元測試，便可輕鬆執行這些測試，確保現有功能不受變更影響。
5. 如果執行自動化建置，便能輕鬆將自動化單元測試套組納入自動化建置處理。這樣一來，即可定期重建系統（通常是夜間）及進行回歸測試，不太需要手動作業。

此外，我們必須強調，若利用 EJB 及 Web Service 來進行分散式元件開發，就一定要測試個別元件。如果沒有 GUI 要測試，必須回到更低階的測試。最好是用這個方法著手測試，不要等到要將部分應用程式發佈爲分散式元件或 Web Service 時，才來煩惱得重新修改程序納入這些測試。

簡言之，若使用自動化單元測試，可以更迅速簡單找出錯誤，有系統執行測試，因而提高整體品質。

## 4. 依規格開發，而非應用程式伺服器。

**牢記規格，若要悖離，必須經過審慎評估。有能力這麼做，不代表就應該這麼做。**

游走在 Java EE 的功能限制邊緣，往往只是自找麻煩。我們發現，許多開發人員會努力嘗試一些自認爲比現有 Java EE「好一些」的做法，最後卻導致嚴重的

效能問題或移轉問題（廠商間移轉，更常見的是版本移轉）。其實，這是移轉常發生的問題，[Beaton] 稱此原則爲移轉作業的基本最佳實務做法。

有幾個地方若沒有採用直接的方法，一定出問題。常見的例子是，開發人員使用 JAAS 模組來取代 Java EE 安全性，而不是根據符合規格的內建應用程式伺服器機制來進行鑑別及授權。請嚴防逾越 Java EE 規格所提供的鑑別機制，這可能會造成重大的安全漏洞及廠商相容性問題。同樣地，使用 Servlet 及 EJB 規格所提供的授權機制，如須逾越，請確定使用規格的 API（如 getCallerPrincipal）作爲執作準則。這樣一來，您就可以善用廠商提供的強大安全基礎架構，在企業有需要時，也可支援更複雜的授權規則。（如需授權的相關資訊，請見 [Ilechko]）。

其他常見問題包括使用未納入 Java EE 規格的持續性機制（以致交易管理困難）、使用 Java EE 程式中的不當 Java Standard Edition 機制（如執行緒作業或 Singleton），以及「自己制定的」解決方案，進行程式之間的通訊，而不是利用 Java 2 Connectors、JMS 或 Web Service 等獲得支援的機制。在從某 Java EE 相容伺服器移到另一伺服器，或移到同一伺服器的新版本時，這種設計選擇就會造成永無止境的困擾。使用 Java EE 以外的元素，往往會導致難察覺的可攜性問題，只有在無法透過規格解決明確問題時，才能放棄使用既有規格。例如，在 EJB 2.1 問世之前，排定執行定時商業邏輯是一大問題。像這種情況，我們會建議儘可能使用廠商提供的解決方案（如 WebSphere Application Server 中的 Scheduler 機制），如果沒有，則使用協力廠商的工具。當然，EJB 規格目前已具備時間型功能，所以我們建議使用標準介面。這樣一來，維護及移轉到較新的規格版本就成了廠商的問題，而不是您本身的問題。

最後，避免太早採用新技術。在新技術尚未納入其他 Java EE 規格或廠商產品之前，過度熱衷採用往往會導致棘手問題。支援十分重要，如果廠商沒有直接支援特定技術，就要審慎考慮是否該使用此技術。人們（尤其是開發人員）往往過度重視簡化開發程序，卻忽略採用大量外部所開發的程式碼（而且廠商沒有支援），長期下來可能造成的不良後果。我們已經見識過太多專案團隊執著於新技術（如最新的開放原始碼架構），很快就依賴這種技術，卻不考慮對企業的實際成本。坦白說，企業架構設計、業務及法務團隊（或貴企業的相關單位），必須仔細評估是否採用非購自廠商的任何技術，就像評估一般產品採購決策一樣。畢竟，除了少數例外情況，我們多數人的工作都是解決業務問題，而不是純粹爲了好玩才發展技術。

## 5. 打從一開始就規劃使用 Java EE 安全性。

**啓用 WebSphere 安全功能，至少限制所有已鑑別的使用者存取所有 EJB 及 URL。不要問，做就對了！**

令我們驚訝連連的是，在我們合作的客戶中，居然只有少數原本就打算啓用 WebSphere Application Server 的 Java EE 安全功能。根據我們估計，只有大約 50% 的客戶一開始就打算使用此功能。我們曾跟數家重要金融機構（銀行、券商等）合作，他們並沒有打算啓用安全功能；所幸，這種狀況通常在部署前的評估階段就得以解決。

未善用 Java EE 安全功能是危險的做法。假設應用程式需要安全功能（幾乎都要），您卻斷定開發人員建置的安全基礎架構會比向 Java EE 購買的更好，這種判斷不太牢靠。保障分散式應用程式的安全極爲困難；例如，您必須使用網路安全加密標記來控制 EJB 的存取權。根據我們的經驗，多數客戶自家開發的安全基礎架構，都有明顯的安全弱點，使正式作業系統承受極高的風險。（詳細資訊請見 [Barcia] 第十八章）。

一般企業不使用 Java EE 安全功能的原因包括：擔心效能降低、認爲 IBM Tivoli® Access Manager 及 Netegrity SiteMinder 等其他安全產品已處理此問題，或忽略 WebSphere Application Server 安全特色及功能。千萬別掉入這些陷阱。尤其，雖然 Tivoli Access Manager 等產品具有卓越的安全功能，但光靠這些並不能保障整個 Java EE 應用程式的安全。這些產品必須搭配 Java EE 應用程式伺服器使用，才能保障整個系統的安全。

另一個不使用 Java EE 安全功能的常見原因是，角色型態的模型無法提供充分的精細存取控制，以符合複雜的商業規則。雖然大多時候確實如此，但卻不能因此不用 Java EE 的安全功能，相反的，應該配合您的特定延伸規則，善用 Java EE 鑑別模型及 Java EE 的功用。如果需要複雜的商業規則來制定安全決策，請撰寫程式碼加以執行，根據現成且可靠的 Java EE 鑑別資訊（使用者 ID 和角色）來作決定（如需授權的相關資訊，請見 [Ilechko]）。

## 6. 建置自己熟悉的內容。

**疊代式開發可讓您逐步掌握 Java EE 的所有組件。請從小型垂直組塊開始建置應用程式，而不是一開始就執行所有部分。**

我們必須承認，Java EE 確實很龐大，如果開發團隊直接從 Java EE 著手，很難一次學會所有部分，有太多概念及 API 須要掌握。在這種情況下，以受管制的小步驟開始採用 Java EE 才是成功關鍵。

最好的執行方法是從小的垂直組塊開始建置應用程式。一旦團隊透過建置簡單的領域模型 (domain model) 及後端持續性機制（可能使用 JDBC）建立起自信心，並且徹底測試該模型後，即可透過使用該領域模型的 Servlet 及 JSP 掌握前端開發。如果開發團隊認爲需要 EJB，也可以在儲存器管理的持續性 EJB 或 JDBC 型 DAO（資料存取物件）之上，先從簡單的 Session Facade 著手，再進行更複雜的建構，如訊息導向的 Bean 及 JMS。

這並不是創新做法，但卻少有團隊會實際以這個方法培養技術。相反的，多數團隊想要一次建置所有元件，因而陷入時間壓力；他們會同時著手建置 MVC 的「視圖」層、「模型」層及「控制器」層。事實上，應該考慮使用一些新的 Agile 部署方法，如 Extreme Programming (XP)，以養成這種循序漸進的學習和開發方式。有一種常用於 XP 的程序稱爲 ModelFirst [Wiki]，即先建置領域模型，作爲組織及執行使用者陳述 (user story) 的機制。基本上，先建置領域模型，作爲第一組使用者陳述的一部分，然後在執行後續使用者陳述後，再按照領域模型建置 UI。這個方法很適合讓團隊循序漸進學習，而不是讓他們去上一堆同步課程（或閱讀大量書籍），以致難以吸收。

此外，各應用程式層均以疊代式開發，可養成使用適當模型及最佳實務做法的習慣。如果從應用程式的較低層開始，並運用「資料存取物件」及 Session Facade 等模型，應該不會在 JSP 及其他「視圖」物件中使用領域邏輯。

最後，如果從小的垂直組塊著手開發，比較容易儘早開始測試應用程式效能。如同 [Joines] 所言，等到應用程式開發週期結束時再來測試效能，註定會出問題。

## 7. 使用 EJB 元件時一律用 Session Facade。

**在架構設計適用時，儘量使用本端 EJB。**

使用 Session Facade 是 EJB 其中一種完善的最佳使用實務做法，其實，這種一般做法已普遍用於任何分散式技術，包括 CORBA、EJB 及 DCOM。基本上，應用程式的「橫跨」分佈層面越低，多個重複網路中繼站（用於小量資料）所浪費的時間就越少。要達成這個目標，須建立超大塊的 Facade 物件，以包含邏輯

子系統，並以單一方法呼叫來執行有用的商業功能。這不僅可以降低網路間接成本，還可爲整個商業功能建立單一交易環境定義，大幅降低 EJB 中的資料庫呼叫數目。（詳細說明請見 [Brown]。[Alur] 說明此模型的標準表示法，但 [Fowler]（包括說明 EJB 之外的情況）及 [Marinescu] 也有相關說明。請見「參考資料」）。細心的讀者會發現，這其實是服務導向架構 (SOA) 的核心原則之一。

屬於 EJB 2.0 規格的 EJB 本端介面，可針對同位置的 EJB 提供效能最佳化。您的應用程式必須明確呼叫本端介面，需要變更程式碼，同時避免之後未經應用程式變更即發佈 EJB。如果確定 EJB 呼叫一定是在本端，則請善用最佳化本端 EJB 的功能。不過，Session Facade 本身的實作（通常是 Stateless Session Bean）應專用於遠端介面。如此一來，其他用戶端就能在遠端使用 EJB，而不會造成現有商業邏輯重大毀損。由於 EJB 可同時擁有本端及遠端介面，所以相當實用。

若要最佳化效能，可將本端介面加入 Session Facade，如此一來，多數時候，至少在 Web 應用程式中，EJB 用戶端及 EJB 會位於相同的 JVM 中。此外，如果 Session Facade 在本端呼叫但使用的是遠端介面，則可使用 Java EE 應用程式伺服器配置最佳化，如 WebSphere "No Local Copies"。然而，您必須瞭解的是，這些替代方案會將互動的語法從依值傳遞 (pass-by-value)，改爲依參照傳遞 (pass-by-reference)，導致難察覺的程式碼錯誤。最好是使用本端 EJB，因爲其行爲是依 Bean 來控制，不會影響整個應用程式伺服器。

如果您針對 Session Facade 使用遠端介面（而不是本端介面），或許也能以符合 Java EE 1.4 的方式，將相同的 Session Facade 發佈爲 Web Service。（這是因爲 JSR 109，即 Java EE 1.4 的 Web Service 部署部分，規定必須使用 Stateless Session Bean 的遠端介面作爲 EJB Web Service 與 EJB 實作之間的介面）。最好是按照此方式執行，如此才能增加商業邏輯的用戶端類型數目。

## 8. 使用 Stateless Session Bean 取代 Stateful Session Bean。

**這可以讓系統發生故障時更容易修復回來。請使用 HttpSession 來儲存使用者特定狀態。**

我們認爲，Stateful Session Beans 已經是過時的觀念。如果仔細思考，您會發現 Stateful Session Beans 在架構設計上，其實跟 CORBA 物件一樣，是與單一伺服器聯結的單一物件實例，使用期限視該伺服器而定。一旦伺服器故障，物件值也跟著喪失，而該 Bean 的所有用戶端也跟著遭殃。

支援 Stateful Session Bean 故障的 Java EE 應用程式伺服器可暫時解決一些問題，但 Stateful 的解決方案並不像 Stateless 解決方案具擴充性。例如，在 WebSphere Application Server，有部署 Stateless Session Bean 的叢集，其所有成員都會平均分攤 Stateless Session Bean 的要求負載，但 Java EE 應用程式伺服器則無法將要求負載平均分攤至 Stateful Bean，換句話說，叢集各伺服器的負載量可能會不平均。此外，使用 Stateful Session Beans 會導致應用程式伺服器處於不利狀態。Stateful Session Beans 會增加系統複雜性，導致故障情況更複雜。龐大分散式系統的主要原則之一，是儘可能使用 Stateless 行爲。

因此，我們建議針對多數應用程式採用 Stateless Session Bean 方法。處理過程中所需的使用者特定狀態，都要以參數形式傳入 EJB 方法（並透過 HttpSession 等機制儲存在 EJB 之外），或從持續的後端儲存庫擷取，作爲 EJB 交易的一部分（例如，藉由使用 Entity Bean）。必要時，此資訊也可以快取到記憶體中，但須瞭解在分散式環境中維持快取一致性的潛在困難。快取最適用於唯讀資料。

通常，必須確定一開始就有周延的擴充計畫。請檢查所有設計假設，確定如果應用程式要在多台伺服器上執行，這些假設仍能成立。此規則不僅適用於上述應用程式碼，也適用於 MBeans 及其他管理介面。

避免使用有狀態的階段作業，不僅是基於 IBM 工具套組可能有限制，而作出的 IBM/WebSphere 建議，也是基本的 Java EE 設計原則。Tyler Jewell 對於 Stateful Bean 的批評，呼應了上述說法，請見 [Jewell]。

## 9. 使用儲存器管理交易。

**瞭解兩階段確定交易在 Java EE 的運作方式，並加以採用，而非自行開發交易管理。儲存器幾乎必定比交易最佳化有效。**

使用儲存器管理交易 (container-managed transaction, CMT) 具有兩大優點，若沒有儲存器支援，幾乎不可能享受這種好處：可組成的工作單元，以及強大的交易行爲。

若您的應用程式碼會明確地開始並結束交易（可能使用 javax.jts.UserTransaction 或原生的資源交易），後續組成模組的要求（可能是重構的一部分）往往需要改變交易程式碼。比方說，若模組 A 開始某資料庫交易、更新資料庫，然後確定交易，而模組 B 也執行一樣的作業，那麼從模組 C 使用兩者時，結果會如何？

現在，模組 C 正在執行單一邏輯動作，實際上會造成兩個獨立的交易發生。如果模組 B 在作業中無法執行，模組 A 的工作仍會繼續，這不是我們想要的行爲。反之，如果模組 A 和模組 B 都使用 CMT，模組 C 也可以開始 CMT（通常是透過部署描述子的隱性作業），而模組 A 及 B 的工作都是相同工作單位的隱性部分，不需要複雜的重做。

如果應用程式在相同作業中需要存取多個資源，便需要兩階段確定(two-phase commit)交易。比方說，如果從 JMS 佇列移除訊息，然後更新資料庫中該訊息的相關記錄，這兩個作業是否都發生或都沒有發生就很重要。如果從佇列中移除了訊息，但系統卻沒有更新資料庫，這個系統就不一致，這種不一致狀態會對客戶及業務造成嚴重影響。

有時候，我們會看到用戶端應用程式會嘗試執行本身的解決方案。如果資料庫更新失敗，應用程式碼可能會嘗試「取消」佇列作業。我們建議不要這麼做。實作比您原本想的還要複雜許多，而且有很多意外狀況（試想如果應用程式在執行期間損毀，後果會如何）。所以，請使用兩階段確定交易。若您使用 CMT 並在單一 CMT 中存取兩階段的提交資源（如 JMS 及多數資料庫），WebSphere Application Server 會處理棘手的工作，確定交易是否完全執行，包括確認系統損毀、資料庫當機等故障狀況。該實作會在交易記錄中維持交易狀態。我們極力建議，若應用程式要存取多種資源，就要採用 CMT 交易。若要存取的資源無法進行兩階段確定，當然就必須使用較複雜的方法，但應儘量避免這種狀況。

## 10. 以 JSP 爲顯示技術的首選。

**唯有在多種顯示輸出類型必須由單一控制器及後端支援時，才使用 XML/XSLT。**

我們常聽到一種論點，您之所以選用 XML 及 XSLT 作爲顯示技術，而不是 JSP，是因爲 JSP「會讓您過度混合模型和視圖」，但 XML/XSLT 在某種程度上則不會有這個問題。可惜事實並非如此，至少不是百分之百如此。事實上，XSL 和 XPath 是程式設計語言。儘管 XSL 以規則爲基礎，而且沒有程式設計人員慣用的所有控制功能，因此可能不符合多數人對程式設計語言的定義，但 XSL 確實是旋轉完成 (Turing-complete) 語言。

重點是由於有這種彈性，開發人員就會善加利用。雖然眾人都同意，JSP 有利於開發人員在視圖中執行「類似模型」的行爲，但事實上，用 XSL 也可以做到某

些相同動作。儘管從 XSL 呼叫資料庫十分困難（若可能的話），但我們也看過有些異常複雜的 XSLT 樣式表執行困難的轉換，最後還是可以產生模型程式碼。

不過，您應該以 JSP 作爲顯示技術首選，最基本的原因是它能有效支援及掌握可用的 Java EE 檢視技術。由於自訂標籤庫、JSTL 及 JSP 2.0 功能的出現，建置 JSP 也變得越來越容易，不需要任何 Java 程式碼，而且可明確區分模型及視圖。有些開發環境已內建重要的 JSP 支援（包括除錯支援），如 IBM Rational Application Developer，而且許多開發人員覺得用 JSP 開發比用 XSL 容易，主要是因爲 JSP 以程序爲基礎，而非規則。雖然 Rational Application Developer 也支援 XSL 開發，但支援 JSP 的圖形配置工具及其他功能（尤其是在像 JSF 的架構環境下），可方便開發人員以 WYSIWYG 的方式工作，XSL 便難以做到這點。

當然，我們不是說您絕不能使用 XSL，在某些情況下，使用 XSL 來呈現視圖是最好的方法，例如，XSL 能表示一組固定資料，然後根據不同樣式表以多種方式呈現（請見 [Fowler]）。不過，這種需求通常是例外狀況，而不是通用規則。如果您只是要產生一個 HTML 來呈現每一頁，XSL 就顯得太過複雜，對開發人員來說，所造成的問題會比解決的還多。

## 11. 使用 HttpSessions 時，儘量只儲存目前商業交易需要的狀態，不存多餘的部分。

**啓用階段作業持續性。**

HttpSessions 很適合用來儲存應用程式狀態的相關資訊。API 便於使用，也容易理解。可惜的是，開發人員往往忽略 HttpSession 的用意，那就是維持暫時的使用者狀態，但這並不是任意資料快取。我們發現，許多系統會將大量資料 (MB) 放入每個使用者的階段作業。如果有 1000 名使用者登入，每人有 1 MB HTTP 階段作業，那麼光是階段作業所使用的記憶體就有 1 GB 或以上。儘量維持小量 HTTP 階段作業，如果不這麼做，應用程式的效能就會降低，最好的經驗法則是低於 2K-4K，但這並非硬性規定，8K 也可以，但顯然會比 2K 慢。請留意這一點，避免 HttpSession 變成垃圾場，專門囤積「可能」會用到的資料。

有個常見問題，便是使用 HttpSessions 來快取容易重建的資訊（如有必要）。由於階段作業是持續的，所以強迫執行不必要的序列化及寫入資料，是十分浪費資源的決策，應該使用記憶體雜湊表來快取資料，在階段作業中只需存放資料的索

引即可。這樣一來，如果使用者自動切換到另一應用程式伺服器，即可重建該資料。（詳細資訊請見 [Brown2]）。

談到階段作業持續性，千萬別忘了啓用這個功能。如果未啓用階段作業持續性，一旦伺服器因爲某些原因停機（伺服器故障或一般維護），當時在該應用程式伺服器上的使用者就會喪失其階段作業，這是很不愉快的經驗。他們必須重新登入，再做一次先前的工作。但如果啓用了階段作業持續性，WebSphere 便會明確將使用者（及其階段作業）自動移到另一台應用程式伺服器，使用者甚至不會察覺到移動的過程。這項功能效果良好，我們確實看過正式作業系統因爲原生程式碼有嚴重錯誤（並非 IBM 的程式碼！）而經常當機，還能提供適當的服務。

## 12. 善用應用程式伺服器的功能，不需要修改程式碼。

**使用 WebSphere Application Server 快取及備妥陳述式 (Prepared Statement) 快取記憶體等功能，即可大幅提升效能並降低間接成本。**

上述第 4 項最佳實務做法已清楚說明，運用特定應用程式伺服器功能來修改程式碼時，爲何要特別謹慎。因爲這麼做會影響可攜性，可能會使版本移轉變得更爲麻煩。不過，您可以、也應該充分利用一組特定應用程式伺服器的功能（尤其是 WebSphere Application Server），因爲這些功能不會變更您的程式碼。您的程式碼應寫入規格中，但如果您瞭解這些功能且知道如何正確使用，則可大幅提升效能。

比方說，在 WebSphere Application Server，您應該啓用動態快取及使用 Servlet 快取，這不僅大幅提升效能，將接間成本降到最低，程式設計模型也不受影響。用快取來改善效能的好處十分明確。很可惜的，目前的 Java EE 規格並沒有納入 Servlet/JSP 快取機制。不過，WebSphere Application Server 可透過動態快取功能來支援頁面及片段快取，不需變更應用程式。快取原則已明確界定，而配置則透過 XML 部署描述子來設定。因此，您的應用程式不會受到影響，仍符合 Java EE 規格且具可攜性，同時享有 WebSphere Servlet 及 JSP 快取所提供的最佳效能。

透過 Servlet 及 JSP 的動態快取功能可能大幅提升效能，視應用程式特性而定。Cox 及 Martin [Cox] 說明，在現有 RDF (Resource Description Format) 站台摘要 (RSS) Servlet 使用動態快取功能，效能好處可高達 10 倍。請注意，這個實驗是使用簡單的 Servlet，較複雜的應用程式組合的改善幅度未必會相同。

爲了取得其他效能好處，WebSphere Application Server Servlet/JSP 結果快取，已跟 WebSphere 外掛程式 ESI Fragment 處理器、IBM HTTP Server Fast Response Cache Accelerator (FRCA) 及 Edge Server 快取功能加以整合。若是讀取工作負載繁重，善用這些功能也可以取得其他顯著效益。（有關效能好處，請見「參考資料」中 [Willenborg] 及 [Bakalova] 之說明）。

再舉一個原則範例（我們發現，客戶只是因爲不知道有這個原則，所以才不使用），撰寫 JDBC 程式碼時，請善用 WebSphere 備妥陳述式快取記憶體。依預設，您在 WebSphere Application Server 中使用 JDBC PreparedStatement 時，它就會編譯陳述一次，然後放入快取以便重複使用，不僅可以在建立 PreparedStatement 的相同方法中重複使用，還可跨程式使用，只要程式中同一個或另一個 PreparedStatement 使用相同 SQL 程式碼即可。省下這個重新編譯的步驟，便能明顯減少 JDBC 驅動程式的呼叫次數，進而改善應用程式效能。不必執行特殊動作來使用這項功能，只要撰寫 JDBC 程式碼來使用 PreparedStatements 即可。透過撰寫程式碼來使用 PreparedStatement，而不是一般 JDBC Statement 類別（純粹使用動態 SQL），便可善用這個效能加強功能，同時又不必犧牲可攜性。

## 13. 充分利用現有環境。

**提供 Java EE EAR 及可配置的安裝 Script，而非黑箱二進位安裝程式。**

在多數情況下，大量 WebSphere Application Server 使用者會在相同的共用 Cell 中執行多個應用程式。這表示，如果您提供一個要安裝的應用程式，就必須能適當地安裝到現有基礎架構。這代表兩件事：第一，必須限制環境的假設數目，同時由於不可能預測每個變數，所以安裝程序必須明白清楚。也就是說，不能提供二進位執行檔的安裝程式。執行安裝的管理者必須瞭解安裝程序對其 Cell 所執行的動作。若要實施這種方法，應該提供 EAR 檔案（或一組 EAR 檔案），以及文件和安裝 Script。這些 Script 必須清晰易讀，以便安裝人員瞭解 Script 所執行的動作，並確認 Script 不會執行任何危險動作。在 Script 不適用的情況下，使用者可能要使用曾經用過的其他程序來安裝您的 EAR，即表示您必須說明安裝程式的執行步驟！

## 14. 充分利用應用程式伺服器環境提供的服務功能。

**使用 WebSphere Application Server Network Deployment 來設計可形成叢集的**

**應用程式。**

我們已提過善用 WebSphere Application Server 安全及交易支援的重要性。另一個經常被忽略的重點是叢集作業。應用程式的設計必須能在叢集環境中執行。多數實際環境都需要叢集作業，以取得延伸性及穩定性。無法叢集作業的應用程式，很快就會引起一連串問題。

與叢集作業密切相關的是支援 WebSphere Application Server Network Deployment。若要建置應用程式做爲商品，請確定您的應用程式可在 WebSphere Application Server Network Deployment 上執行，而且不只是單一伺服器版本。

## 15. 充分利用 Java EE，別做表面功夫。

**致力建置實際的 Java EE 應用程式，真正發揮 Java EE 功能。**

我們見過最惱人的一件事，就是明明聲稱「可在 WebSphere 中執行」的應用程式，實際上卻不是 WebSphere 應用程式。很多時候是只有一小部分程式碼（可能是 Servlet）在 WebSphere Application Server，而其餘的應用程式邏輯實際上是在不同的程序中；例如以 Java、C、C++ 等語言撰寫（但不是用 Java EE）的常駐程式處理程序，才是執行作業的程式。那不是真正的 WebSphere Application Server 應用程式。這種應用程式幾乎無法使用 WebSphere Application Server 提供的所有服務功能。若以爲這是 WebSphere Application Server 應用程式，可能會猛然發覺事實並非如此。

## 16. 規劃版本更新。

**改變是必然的，請規劃新版本及修正程式更新，以便客戶維持最新狀態。**

WebSphere Application Server 不斷發展，所以可想而知，IBM 會定期發佈 WebSphere Application Server 修正程式及新的重要版本。您必須做好更新規劃。這會影響兩種開發組織：內部開發人員及協力廠商應用程式的供應商；他們面臨的基本問題都一樣，但個別受影響的程度卻不同。

首先，以修正程式爲例，IBM 會定期發佈建議更新項目，以修正產品中的已知錯誤。雖然不太可能總是維持在最新狀態，但切記不要差太多。那「差」多少是可以接受的呢？這問題並沒有標準答案，但您應該在發佈後的幾個月內，計畫支

援修正程式等級，也就是說，在一年內要升級正式作業系統好幾次。內部開發人員可任意跳過某些修正等級，一次支援一個修正等級以降低測試成本，但應用程式供應商卻不能如此。如果您是應用程式供應商，就必須同時支援多個修正等級，以便客戶搭配其他軟體來執行您的軟體。如果您只支援一個修正等級，基本上就不太可能找到多個產品都相容的修正等級。實際上，供應商最好的做法是使用支援「向上相容修正程式」模式。這是 IBM 用來支援我們所整合的其他廠商產品（如 Oracle®、Solaris™ 等）的方法。如需相關資訊，請見我們的支援原則。

其次，考量主要版本升級。IBM 會定期發佈新的產品版本，其中隨附重要的功能升級。我們會繼續支援較舊的重要版本，但不會永遠支援，也就是說，您必須儘早準備將重要版本更新到另一版本。這是無可避免的，而且必須納入成本模式考量。如果您是供應商，這表示您必須持續將產品升級到新版的 WebSphere Application Server，否則您的客戶就會陷入無法支援 IBM 產品的困境。據我們所知，這種情況已經發生不止一次！若您要購買供應商的產品，我們建議您先審慎評估，確定該廠商一定會支援新版的 IBM 產品。若陷入無法支援 IBM 產品的困境，就十分危險。

## 17. 在程式碼的所有重點處，都使用標準記載架構記錄程式狀態。

**這包括異常狀況處理程式。請使用如 JDK 1.4 或 Log4J 等記載架構。**

記載有時是程式設計過程中最麻煩且容易疏忽的部分，但這可避免花時間不斷除錯，準時下班回家。根據一般經驗法則，每個轉變點都要記載，若要從某方法傳參數到另一方法，或彼此傳遞，則須加以記載；在某物件上執行轉換作業時，也須記載；發生疑問時，也要記載。

一旦決定記載，請選擇適當的架構。市面上有許多理想的選擇，但我們偏好 JDK 1.4 追蹤 API，因其已充分整合到 WebSphere Application Server 追蹤子系統，而且符合標準。

## 18. 完成相關作業後，一定要自行清理收尾。

**如果從儲存區取得一個物件，一定要放回原處。**

根據我們的觀察，無論在開發、測試或正式作業系統中執行，Java EE 應用程式

相關的常見錯誤，就是記憶體洩漏，十之八九都是因爲開發人員忘記結束連線（多數是 JDBC）或將物件放回儲存區。請務必確實依規定結束或放回儲存區的物件，千萬別成爲造成不當程式碼出現的害群之馬。

## 19. 遵循嚴格的開發及測試程序。

**這包括採納及遵守軟體開發方法。**

大規模系統開發十分困難，所以應當認真看待。然而，很多時候，我們卻發現團隊馬虎執行原則，或不認真遵守開發方法（這些方法也許不適合其開發類型），或不太解瞭這一點。也許，最糟的是嘗試「每個月換一種開發方法」，即團隊會在一個專案的生命週期中，由 RUP 換成 XP，然後再換到其他 Agile 方法。

簡言之，對多數團隊來說，只要團隊成員有效掌握、嚴格遵守並謹慎調整這些方法，因應特定技術特質及使用該方法的團隊，幾乎所有方法都會奏效。對於尚未採用方法或未充分利用所選方法的團隊，我們建議其參閱下列經典著作，如 [Jacobson]、[Beck1] 或 [Cockburn]。另一個有用的資訊來源，是最近發表的 Eclipse Process Framework [Eclipse] 之 OpenUp 外掛程式。在此就不再贅述，建議讀者參閱 [Hambrick] 及 [Beaton2] （請見「參考資料」）。

## 結論

在本摘要中，我們已介紹了有助於管理 Java EE 開發工作的核心模型及最佳實務做法。儘管未說明落實這些模型所必須的所有細節，但希望您已獲得足夠的指示及方向，可決定下一步要怎麼走。

## 謝詞

感謝率先提出這些模型及最佳實務做法的前輩（以及下列各位），此外也感謝 John Martinek、Paul Ilechko、Bill Hines、Dave Artus 和 Roland Barcia 協助審閱本文。

## 作者簡介

**Keys Botzum** 是 IBM Software Services for WebSphere 的資深技術人員，在大規模分散式系統設計方面有超過十年的經驗，專精安全功能。他使用過各種分散式技術，包括 Sun RPC、DCE、CORBA、AFS 及 DFS，最近則是鑽研 J2EE 及相關技術。他擁有史丹佛大學電腦科學碩士學位及卡內基美隆大學應用數學暨電腦科學學士學位。Botzum 曾發表許多有關 WebSphere 和 WebSphere 安全性的文章。有關 Keys Botzum 的其他文章及簡報，請見 http://www.keysbotzum.com 及 IBM developerWorks WebSphere。他也著有《IBM WebSphere: Deployment and Advanced Configuration》（與 Bill Hines 合著）。

**Kyle Brown** 是 IBM Software Services for WebSphere 的傑出工程師。他針對物件導向主題和 J2EE 技術，提供「財星」五百大企業客戶諮詢服務、教育訓練及指導。他也與人合著《Java Programming with IBM WebSphere》、《WebSphere AEs 4.0 Workbook for Enterprise Java Beans, 3rd Edition》及《The Design Patterns Smalltalk Companion》等書，更常受邀出席 Enterprise Java、物件導向設計及設計模型等主題之相關會議發表演說。

**Ruth Willenborg** 是 IBM WebSphere Technology Institute 資深技術人員，專攻虛擬化技術。在接受此職務前，Ruth 曾任 WebSphere Performance 團隊經理，負責 WebSphere Application Server 效能分析、效能評比及效能工具開發。她在 IBM 軟體開發方面有超過二十年的經驗，也與人合著《Performance Analysis for Java Web Sites》（Addison-Wesley，2002 年）。

**Albert Wong** 是 IBM Retail On Demand Emerging Business Opportunities (EBO) 的 IT 架構設計師，這是內部創投組織，旨在增加 IBM 在零售業的商業與技術解決方案。由於 EBO 的性質相當多變，他的技術也橫跨 I/T 商業與技術開發各層面，從技術預售、解決方案設計與執行、產品開發，到生態系統啓用功能，均包括在內。