第一部分 JSP 入 门

第一章 概 述

1.1 Java技术

Java是一种简单易用、完全面向对象、具有平台无关性且安全可靠的主要面向 Internet的开发工具。自从1995年正式问世以来，Java的快速发展已经让整个Web世界发生了翻天覆地的变化。随着Java Servlet的推出，Java在电子商务方面开始崭露头角，最新的 Java Server Page技术的推出，更是让Java成为基于Web的应用程序的首选开发工具。

要学习Java技术中的Java Server Page，Java基础是必不可少的，本书将在第 2章为没有Java基础的读者简单讲解Java的基础语法和Java Beans等，它们是在学习JSP之前必须掌握的Java知识。这里，先回顾一下Java的发展历程，然后讲解几个后面将要用到的重要概念。

1.1.1 Java技术的发展

Java技术是由美国 Sun公司倡导和推出的， Java技术包括Java语言和Java Media APIs、Security APIs、Management APIs、Java Applet、Java RMI 、JavaBeans、JavaOS、Java Servlet、JDBC、JNDI、Enterprise JavaBeans等，下面是Java技术的发展简述。

1990年，Sun公司James Gosling领导的小组设计了一种平台独立的语言 Oak，主要用于为各种家用电器编写程序。

1995年1月，Oak 被改名为Java，1995年5月23日，Sun公司在Sun World ‘95上正式发布Java和HotJava浏览器。

1995年8月至12月，Netscape公司、Oracle公司、Borland公司、SGI公司、Adobe公司、IBM公司、AT&T公司、Intel公司获得Java许可证。

1996年1月，Sun公司宣布成立新的业务部门——JavaSoft部，以开发、销售并支持基于 Java技术的产品，由Alan Baratz任总裁。同时推出Java开发工具包JDK(Java Development Kit) 1.0，为开发人员提供编制Java应用软件所需的工具。

1996年2月，Sun发布Java芯片系列，包括picoJava、MicroJava和UltraJava，并推出Java数据库连接JDBC (Java Database Connectivity)。

1996年3月，Sun公司推出Java WorkShop。

1996年4月，Macrosoft公司、SCO公司、苹果电脑公司(Apple)、NEC公司等获得Java许可证。Sun公司宣布苹果电脑、 HP、日立、IBM、微软、 Novell、SGI、SCO、Tamdem等公司将把Java平台嵌入到其操作系统中。

1996年5月，HP公司、Sybase公司获得Java许可证。北方电讯公司宣布把Java技术和Java微处理器应用到其下一代电话机中的计划。5月29日，Sun公司在旧金山举行第一届JavaOne世界Java开发者大会，业界人士踊跃参加。Sun公司在大会上推出一系列Java平台新技术。

1996年8月，Java WorkShop成为Sun通过互联网提供的第一个产品。

1996年9月，Addison-Wesley和Sun推出Java虚拟机规范和Java类库。

1996年10月，德州仪器等公司获得Java许可证。Sun提前完成JavaBeans规范并发布。发布第一个Java JIT(Just-In-Time)编译器，并打算在Java WorkShop和Solaris操作系统中加入JIT。10月29日，Sun发布Java企业计算技术，包括JavaStation网络计算机、65家公司发布的85个Java产品和应用、7个新的Java培训课程和Java咨询服务、基于Java的Solstice互联网邮件软件、新的Java开发者支持服务、演示HotJava Views、Java Tutor、完成Java Card API等。Sun宣布完成Java Card API规范，这是智能卡使用的第一个开放API。Java Card规范将把Java能力赋予全世界的亿万张智能卡。

1996年11月，IBM公司获得JavaOS和HotJava许可证。Novell公司获得Java WorkShop许可证。Sun和IBM宣布双方就提供Java化的商业解决方案达成一项广泛协议，IBM同意建立第一个Java检验中心。

1996年12月，Xerox等公司获得Java或JavaOS许可证。Sun发布JDK 1.1、Java商贸工具包、JavaBeans开发包及一系列Java APIs。推出一个新的Java Server产品系列，其中包括Java Web Server、Java NC Server和Java Server Toolkit。Sun发布100%纯Java计划，得到百家公司的支持。

1997年1月，SAS等公司获得Java许可证。Sun交付完善的JavaBeans开发包，这是在确定其规范后不到8个月内完成的。

1997年2月，Sun和ARM公司宣布同意使JavaOS能运行在ARM公司的RISC处理器架构上。Informix公司宣布在其Universal Server和其他数据库产品上支持JDK 1.1。Netscape公司宣布其Netscape Communicator支持所有Java化的应用软件和核心APIs。

1997年3月，HP公司获得Java WorkShop许可证，用于HP-UX操作系统。西门子AG公司等获得Java许可证。日立半导体公司、Informix公司等获得JavaOS许可证。Novell公司获得Java Studio许可证。Sun发售JavaOS 1.0操作系统，这是一种在微处理器上运行Java环境的最小、最快的方法，提供给Sun的JavaOS许可证持有者使用。Sun发售HotJava Browser 1.0，这是一种Java浏览环境，可以方便地按剪裁来编制专用的信息应用软件，如客户自助台和打上公司牌号的网络应用软件。Sun推出JDK 1.1.1。

1999年6月，Sun 发布JDK 1.3和Java Web Server 2.0。

1.1.2 JavaBeans

什么是JavaBeans？JavaBeans就是Java的可重用组件技术。ASP通过COM来扩充复杂的功能，如文件上载、发送email以及将业务处理或复杂计算分离出来成为独立可重复利用的模块。JSP通过JavaBean实现了同样的功能扩充。JSP 对于在 Web 应用中集成 JavaBean 组件提供了完善的支持。这种支持不仅能缩短开发时间（可以直接利用经测试和可信任的已有组件，避免了重复开发），也为 JSP 应用带来了更多的可伸缩性。JavaBean 组件可以用来执行复杂的计算任务，或负

责与数据库的交互以及数据提取等。

在实际的JSP开发过程中，读者将会发现，和传统的ASP或PHP页面相比，JSP页面将会是非常简洁的，由于JavaBeans开发起来简单，又可以利用Java语言的强大功能，许多动态页面处理过程实际上被封装到了JavaBeans中。

1.1.3 JDBC

JDBC是用于执行SQL语句的Java应用程序接口，由一组用Java语言编写的类与接口组成，在JSP中将使用JDBC来访问数据库。JDBC是一种规范，它让各数据库厂商为Java程序员提供标准的数据库访问类和接口，这样就使得独立于DBMS的Java应用程序的开发工具和产品成为可能。一般的Java开发工具都带有JDBC-ODBC桥驱动程序，这样，只要是能够使用ODBC访问的数据库系统，也就能够使用JDBC访问了。有趣的是，不同于ODBC是Open Database Connectivity的简称，JDBC并不是Java Database Connecivity的简称，而是SUN的注册商标，至少官方说法是这样的。

1.1.4 J2EE

电子商务和信息技术的快速发展以及对它们的需求给应用程序开发人员带来了新的压力。必须以比以前更少的金钱、更少的资源来更快地设计、开发企业应用程序。

为了降低成本，并加快企业应用程序的设计和开发，J2EE平台提供了一个基于组件的方法，来设计、开发、装配及部署企业应用程序。J2EE平台提供了多层的分布式的应用模型、组件再用、一致化的安全模型以及灵活的事务控制。您不仅可以用比以前更快的速度向市场推出创造性的客户解决方案，而且您的平台独立的、基于组件的J2EE解决方案不会被束缚在任何一个厂商的产品和API上。

1. J2EE规范定义了以下种类的组件

- 应用客户组件。
- Enterprise JavaBeans组件。
- Servlet及JavaServer Pages（JSP页面）组件（也被称作Web组件）。
- Applet。

一个多层的分布式的应用模型意味着应用逻辑被根据功能划分成组件，并且可以在同一个服务器或不同的服务器上安装组成J2EE应用的这些不同的组件。一个应用组件应被安装在什么地方，取决于该应用组件属于该多层的J2EE环境中的哪一层。这些层是客户层、Web层、业务层及企业信息系统层（EIS）等。

(1) 客户层

J2EE应用可以是基于Web的，也可以是不基于Web的。在一个基于Web的J2EE应用中，用户的浏览器在客户层中运行，并从一个Web服务器下载Web层中的静态HTML页面或由JSP或Servlet生成的动态HTML页面。在一个不基于Web的J2EE应用程序中，一个独立客户程序不运行在一个HTML页面中，而是运行在其他一些基于网络的系统（比如手持设备或汽车电话）中，Applet程序，在客户层中运行，并在不经过Web层的情况下访问Enterprise Beans。这个不

基于Web 的客户层可能也包括一个JavaBeans 类来管理用户输入，并将该输入发送到在企业层中运行的Enterprise Beans类来进行处理。根据J2EE 规范，JavaBeans 类不被视为组件。

为J2EE 平台编写的JavaBeans 类有实例变量和用于访问实例变量中的数据的“ get 和set 方法 ”。以这种方式使用的 JavaBeans 类在设计和实现上通常都是简单的，但是它们必须符合JavaBeans 规范中列出的命名和设计约定。

(2) Web 层

J2EE Web 组件可以由JSP 页面、基于Web 的Applet以及显示HTML 页面的Servlet组成。调用Servlet或者JSP 页面的HTML 页面在应用程序组装时与 Web 组件打包在一起。就像客户层一样，Web 层可能包括一个 JavaBeans 类来管理用户输入，并将输入发送到在业务层中运行的Enterprise Beans 类来进行处理。运行在客户层的 Web 组件依赖容器来支持诸如客户请求和响应及Enterprise Bean 查询等。

(3) 业务层

作为解决或满足某个特定业务领域（比如银行、零售或金融业）需要的逻辑的业务代码由运行在业务层的Enterprise Beans 来执行。一个Enterprise Bean 从客户程序处接收数据，对数据进行处理（如果需要），再将数据发送到企业信息系统层存储起来。一个 Enterprise Beans 还从存储中检索数据，并将数据送回客户程序。运行在业务层的 Enterprise Beans 依赖于容器来为诸如事务、生命期、状态管理、多线程及资源存储池等提供通常都非常复杂的系统级代码。业务层经常被称作Enterprise JavaBeans （EJB ）层。业务层和Web 层一起构成了3 层J2EE应用的中间层，而其他两层是客户层和企业信息系统层。

(4) 企业信息系统层

企业信息系统层运行企业信息系统软件，这层包括企业基础设施系统，例如企业资源计划(ERP)、大型机事务处理（mainframe transactionprocessing ）、数据库系统及其他遗留信息系统(legacy informationsystems)。J2EE 应用组件因为某种原因(例如访问数据库) 可能需要访问企业信息系统。J2EE 平台的未来版本将支持Connector 架构，该架构是将J2EE 平台连接到企业信息系统上的一个标准API。

(5) 查询服务

因为一个J2EE 应用程序的组件是单独运行的，并且往往在不同的设备上运行，因此，需要一种能让客户层和 Web 层代码查询并引用其他代码和资源的方法。客户层和 Web 层代码使用Java 命名和目录接口（JNDI ）来查询用户定义的对象（例如Enterprise Beans ）、环境条目（例如一个数据库驱动器的位置）、企业信息系统层中用于查找资源的 JDBC DataSource对象，以及消息连接。

(6) 安全和事务管理

诸如安全和事务管理这样的应用行为可以在部署时在 Web 和Enterprise Beans 组件上进行配置。这个特征将应用逻辑从可能随装配而变化的配置设定中分开了。

J2EE 安全模型允许配置一个Web 或Enterprise Beans 组件，使系统资源只能由授权的用户访问。例如，一个Web 组件可以被配置成提示输入用户名和密码。一个 Enterprise Beans 组件可以被配置成只让特定团体中的成员调用其某些方法。或者，一个 Servlet 组件可以被配置成让某个

组织中的所有人都能访问其某些方法，同时只让该组织中的某些享有特权的人访问其中一些方法。同样是该Servlet 组件，可以针对另外一个环境而被配置成让每个人都能访问其所有方法，或者仅让选定的少数人访问其所有方法。

J2EE 事务模型使得能够在部署时定义构成一个单一事务的方法之间的关系，以使一个事务中的所有方法被处理成一个单一的单元。这是我们所希望的，因为一个事务是一系列步骤，这些步骤要么全部完成，要么全部取消。例如，一个 Enterprise Beans 可能有一组方法，使我们可以通过从第一个账户借出并存入第二个账户的方式而将钱从第一个账户转移到第二个账户。我们希望全部的操作被作为一个单元对待，这样，如果在借出之后存入之前发生了故障，该借出操作被取消。事务属性是在装配期间定义在一个组件上的。这使得能将来自多个应用组件的方法归到一个事务中，这说明，我们可以轻易变更一个 J2EE 应用程序中的应用组件，并重新指定事务属性，而不必改变代码或重新编译。在设计应用组件时，要记住，尽管 Enterprise Beans 有一个可使应用组件的容器自动启动多步事务的机制，但是 Applet 和应用的客户容器可能并不支持这一点。然而，Applet 和应用客户容器总是能够调用支持这一点的一个 Enterprise Beans。还应当注意，JSP 页面和Servlet 没有被设计成是事务的，它们通常应当将事务工作交给一个 Enterprise Bean 来完成。然而，如果事务工作在一个 JSP 页面或Servlet 中是必须的，那么此种工作也应当是非常有限的。

(7) 可重用应用组件

J2EE 组件（Applet 、应用的客户、Enterprise Beans 、JSP 页面及Servlet ）都被打包成模块，并以Java Archive （JAR ）文件的形式交付。一个模块由相关的组件、相关的文件及描述如何配置组件的配置描述文件组成。例如，在组装过程中，一个 HTML 页面和Servlet 被打包进一个模块之中，该模块包含 HTML文件、Servlet 组件及相关的配置描述文件，并以一个 Web ARchive（WAR ）文件的形式交付，该WAR 文件是一个带.war 扩展名的标准JAR 文件。模块的使用使得利用相同组件中的某些组件来组装不同的 J2EE 应用程序成为可能。例如，一个 J2EE 应用程序的Web 版可能有一个Enterprise Beans 组件，还有一个JSP 页面组件。该Enterprise Beans 组件可以与一个应用客户组件结合，以生成该应用程序的非 Web 版本。这不需要进行额外的编码，只是一个装配和部署的问题。并且，可重用组件使得将应用开发和部署过程划分成由不同的角色来完成成为可能，这样，不同的人或者公司就能完成封装和部署过程的不同部分。

2. J2EE 平台定义了如下角色：

(1) J2EE 产品提供商

设计并使J2EE 平台、API 和在J2EE 规范中定义的其他特征能被其他公司或人购得的公司。

(2) 应用组件提供商

创建用于J2EE 应用程序的Web 组件、Enterprise Beans 组件、Applet 或应用客户程序的公司或个人。在装配过程中，应用组件文件、接口及类被打包进一个 JAR 文件中。

(3) 应用程序装配商

从组件提供商获得应用组件 JAR 文件，并将它们组装成一个 J2EE 应用的Enterprise Archive（EAR）文件的公司或个人，这种文件是一个带 .ear扩展名的标准文件。应用装配商提供与该应用程序相关的整体信息，并使用验证工具来检验 EAR 文件的内容是正确的。组装和部署信息存

储在一个基于文本的配置描述文件中，此种文件使用 XML 标记来标记该文本。应用装配商可以使用一个能通过交互式选择来正确添加 XML 标记的装配和配置工具来编辑该配置描述文件。

(4) 部署商

部署（depley）J2EE 应用程序的公司或个人。其职责包括设定事务控制、安全属性，并根据应用组件提供商提供的指示来标明一个 Enterprise Bean 是自己处理自身的存储，还是由一个容器来处理等。部署涉及配置和安装。在配置过程中，部署商遵循应用组件提供商提供的指示来解决外部依赖问题，定义安全设定，以及分配事务属性。在安装过程中，部署商将应用组件安装到服务器上，并生成容器特定的类和接口。

(5) 系统管理员

配置并管理运行J2EE 应用程序的计算环境和网络基础设施，并监督运行环境的人员。

(6) 工具提供商

生产被组件提供商、装配商及部署商使用的用于进行开发、组装和打包的工具的公司或个人。

(7) 设计用户界面和引擎

在为J2EE 应用程序设计用户界面和后端引擎时，需要决定让该程序是基于 Web，还是不基于Web。在做出这个决定时，我们可能希望考虑平台配置、下载速度、安全、网络流量和网络服务。

例如，包含有用户界面并且经常被大量用户访问的一个 Applet 可能需要花很长的时间才能被下载下来，这让用户沮丧。然而，如果知道该 Applet要运行在一个公司的内部网内的受控环境中，那么，在这种情况下，该 Applet 将拥有一个完全可接受的下载速度。另一个考虑是，繁重的处理应当在哪里执行。例如，如果客户程序在一个蜂窝电话或呼机中执行，服务器应当完成尽量多的计算和数据处理，而客户程序只应显示结果就可以了。然而，设计在一个强大的台式机平台上运行的大型财务分析系统则应当在客户机上完成其复杂计算。应用的客户程序和 Applet 用户界面通常都是用Swing API 创建的，该API 可从标准版Java2平台中获得。Swing API 提供了一整套GUI 组件（表格、树形结构、按钮等），这些组件可以被用来实现一种比用一个典型的 HTML 页面所能实现的更为交互的体验。 Swing 也支持HTML 文本组件，这个组件可以被用来显示来自一个服务器的响应。客户程序可以直接访问 Enterprise Beans 层或企业信息系统层。但应谨慎实现这种程序。绕过EJB 层的程序可以使用JDBC API 来访问一个关系型数据库，但应被限制于对数据库表格进行维护等管理任务上。

(8) 设计基于Web 的应用程序

基于Web 的应用程序是基于浏览器的，并且，如果它们运行在 Internet上，就可能被全世界的人访问。当设计一个基于 Web 的应用程序时，不仅需要决定用什么来处理内容和应用逻辑（HTML 、XML 、JSP 页面及Servlet），而且还应当考虑使该应用程序国际化。一个国际化的基于Web 的应用程序向用户提供了选择一种语言，然后根据该选定语言加载应用的正文的方式。对被支持的每种语言而言，应用正文都被存储在一个外部文件中，并且与另外一个文件的关键词相对应。应用代码使用这些关键词及选定的语言来加载正确的文本。国际化 API 还提供类来根据选定的语言格式化日期和金钱。一旦制订了使应用程序国际化的细节，就可以决定用什么

来实现它了。总的来说，一个基于 Web 的应用程序使用 HTML 来显示数据；用 XML 来定义数据以使其可被另一个程序读取并处理；使用 JSP 页面或 Servlet 来管理用户与业务层或存储层之间的数据流。

可以在 J2EE 平台上实现的基于 Web 的应用程序有四种。从简单到复杂排列，它们是：

- B 基本 HTML。
- B 带基本 JSP 页面或 Servlet 的 HTML。
- B 带 Java Beans 类的 JSP 页面。
- B 将应用逻辑根据功能划分成区域的高度结构化的应用。

当设计一个基于 Web 的应用程序时，需要决定用什么来建立它。如果是从建立一个简单的应用程序开始着手，并且认为以后会给该应用程序添加功能，那么，设计就应当适应今后发展的需要。

(9) 模型、视图和控制器架构

在基于组件的 J2EE 平台充分内置了灵活性的情况下，剩下的问题可能是如何组织应用程序以实现简单高效的应用程序升级和维护，以及如何让不懂程序代码的人员避开程序数据。答案就在模型、视图和控制器架构（MVC）的使用之中。MVC 这样的架构是一个描述重现的问题及其解决方案的设计范式，但每次问题重现时，解决方案都不会完全相同。

MVC 设计范式包括三种对象：模型（model）提供应用业务逻辑（Enterprise Beans 类）；视图（view）则是其在屏幕上的显示（HTML 页面、JSP 页面、Swing GUI）；控制器则是 Servlet 、JavaBeans 或 Session Beans 类，它用于管理用户与视图发生的交互。我们可以将控制器想像成处在视图和数据之间，对视图如何与模型交互进行管理。通过使视图完全独立于控制器和模型，就可以轻松替换前端客户程序。并且，通过将控制器和模型代码保持在视图之外，那些不理解这些代码的人员就不能改变他们不应改变的东西。

将控制器和模型分开就可以在不影响模型的情况下改变控制器，也可以在不影响控制器的情况下改变模型。例如，如果应用的前端是一个 HTML 页面，HTML 专家就可以更新它。如果使用一个 JSP 页面，将控制器的代码放到一个 JavaBeans 或 SessionBeans 类中，或使用动作标记（action tags），这样，JSP 页面就仅包含 JSP 代码了。

本书将在第二部分中讲解如何使用 J2EE 来建立企业级的 Web 应用。

1.1.5 EJB

EJB 就是前面说的 Enterprise JavaBeans。EJB 上层的分布式应用程序是基于对象组件模型的，低层的事务服务使用了 API 技术。EJB 技术简化了用 JAVA 语言编写的企业应用系统的开发、配置和执行。EJB 的体系结构规范由 Sun Microsystems 公司制定。

EJB 技术定义了一组可重用的组件：Enterprise Beans。可以利用这些组件像搭积木一样你的建立分布式应用程序。当你把代码写好之后，这些组件就被组合到特定的文件中去。每个文件有一个或多个 Enterprise Beans，在加上一些配置参数。最后，这些 Enterprise Beans 被配置到一个装了 EJB 容器的平台上。客户能够通过这些 Beans 的 Home 接口，定位到某个 Beans，并产生这个 Beans 的一个实例。这样，客户就能够调用 Beans 的应用方法和远程接口。

EJB服务器作为容器和低层平台的桥梁，管理着 EJB容器和函数。它向EJB容器提供了访问系统服务的能力。例如：数据库的管理和事务的管理，或者其他的 Enterprise的应用服务器。

J2EE 应用程序中的Enterprise Beans

当编写管理特定业务功能（比如追踪雇员资料或进行复杂财务计算）的 J2EE 应用程序时，请将完成这些任务的业务逻辑放置在EJB 层的Enterprise Beans 中。通过这种方式，就可以使代码集中在解决手边的业务问题，而利用 Enterprise Beans 容器来支持低层服务，比如状态管理、事务管理、线程管理、远程数据访问和安全等。将业务逻辑与低层系统逻辑分开意味着容器可以在运行时创建和管理 Enterprise Beans。按照规范编写的任何 Enterprise Beans，都可以根据其在一个特定的J2EE 应用程序中将被如何使用来对其事务管理或安全属性进行配置，并可以被部署到任何一个与规范兼容的容器中。可重用组件使不必改变和重新编译 Enterprise Beans 代码成为可能。一个Enterprise Beans 由接口和类组成。客户程序通过 Enterprise Beans的Home 和远程接口来访问Enterprise Beans 的方法。Home 接口提供了创建、删除和定位Enterprise Beans 的方法，而远程接口则提供了业务方法。在部署时，容器由这些接口来创建类，使客户能够创建、删除、定位或调用位于Enterprise Beans 上的业务方法。Enterprise Beans 类提供了业务方法、创建方法和查询方法的实现。如果 Enterprise Beans 管理它自己的持久性的话，还为其生命期方法提供了实现。有两种Enterprise Beans ：Entity Beans 和Session Beans。

一个Session Beans 代表与客户程序的一个短暂会话，而且可能执行数据库读写操作。一个Session Beans 可能会自己调用JDBC，或者它可能使用 Entity Beans来完成此种调用。在后面这种情况下，这个Session Beans 是该Entity Beans 的客户。一个Session Beans 的域包含会话状态，而且是短暂的。如果服务器或者客户程序崩溃，该 Session Beans 就丢失了。这种模式通常被用于像PL/SQL 这样的数据库程序设计语言上。

一个Entity Beans 代表一个数据库中的数据及作用于该数据的方法。在一个关系型数据库中的雇员信息表中，每一行都有一个 Beans 来代表。Entity Beans 是事务的，并且是长寿命的。只要数据留在数据库中，Entity Beans 就存在。这个模式可以被很容易地用于关系型数据库，而不仅仅限于对象数据库。

Session Beans 可以是有状态的，也可以是无状态的。一个有状态的 Session Beans 包含代表客户程序的会话状态。该会话状态是该 Session Beans实例的域值加上这些域值所引用到的所有对象。有状态Session Beans 并不代表在一个持久数据存储中的数据，但是，它可以代表客户程序访问和更新数据。无状态 Session Beans 没有用于某个特定客户程序的任何状态信息。它们通常被用于提供不保持任何特定状态的服务器端行为。无状态 sessionBeans 要求更少的系统资源。一个提供一种一般服务，或用于表示被存储的数据的一个被共享的视图的业务对象是无状态 Session Bean的一个例子。

因为Enterprise Beans 占用可观的系统资源和带宽，你可能希望将某些业务对象构造成数据访问对象或值对象。数据访问对象完成诸如代表客户程序访问数据库等工作。值对象用于代表容纳数据字段并提供简单的“ get 和set ”方法来访问这些数据的一个结构。另外，可以将程序构造成使用Enterprise Bean 在客户和EJB 层的其他部分之间承担通信的任务。

对于一个使用容器管理的持久性来访问关系型数据库的 Enterprise Beans，并不要求在Beans

的代码中使用任何JDBC 2.0 API 来进行数据库访问，因为容器完成了这些工作。然而，如果使用Beans 管理的持久性，或者要访问一个非关系型数据库的企业信息系统，那么就必须在 Beans 中提供相应的代码来完成这些工作。在 Enterprise Beans 使用Beans 管理的持久性来访问一个数据库的情况下，必须使用 JDBC 2.0 API 代码来实现该Enterprise Beans 的生命期方法，以便处理数据的加载和存储，以及运行时在系统和持久数据存储之间维持数据的一致性。

一个使用Beans 管理的持久性的Enterprise Beans，或一个需要访问企业信息系统的 Web 组件必须提供合适的代码。这些代码可能是用于进行数据库访问的 JDBC 2.0 API ；或是用于访问一个特定企业信息系统的企业信息系统 API ；或是用于抽象企业信息系统 API 的复杂性和低层细节的一个访问对象，或是用于访问企业信息系统资源的一个连接对象。

尽管Web 层使用HTTP 或HTTPS 来在各层之间传输数据，但是 EJB 层使用的是RMI-IIOP。RMI-IIOP 是一个完整的分布式计算协议，能让任何访问 Enterprise Bean 的客户层程序或Web 层程序直接访问EJB 层的服务。这些服务包括用于查找和引用 Enterprise Beans 的JNDI，发送和接收异步消息的Java Message Service （JMS ），以及用于关系型数据库访问的 JDBC。

1.1.6 Java Servlet

Java Servlet是JSP技术的基础，而且大型的 Web应用程序的开发需要 Java Servlet和JSP配合才能完成，这里简单介绍 Servlet的相关知识，Servlet的开发将在第二部分讲述。

Servlet这个名称大概源于 Applet，现在国内的翻译方式很多，本书为了避免误会，决定直接采用Servlet而不做任何翻译，读者如果愿意，可以称之为“ 小服务程序 ”。Servlet其实和传统的CGI程序和ISAPI、NSAPI等Web程序开发工具的作用是相同的，在使用 Java Servlet以后，用户不必再使用效率低下的 CGI方式，也不必使用只能在某个固定 Web服务器平台运行的 API方式来动态生成 Web页面。许多Web服务器都支持Servlet，即使不直接支持 Servlet的Web服务器也可以通过附加的应用服务器和模块来支持 Servlet。得益于Java的跨平台的特性，Servlet也是平台无关的，实际上，只要符合 Java Servlet规范，Servlet是完全平台无关且是 Web服务器无关的。由于 Java Servlet内部是以线程方式提供服务，不必对于每个请求都启动一个进程，并且利用多线程机制可以同时为多个请求服务，因此 Java Servlet效率非常高。

但Java Servlet也不是没有缺点，和传统的 CGI、ISAPI、NSAPI方式相同，Java Servlet是利用输出HTML语句来实现动态网页的，如果用 Java Servlet来开发整个网站，动态部分和静态页面的整合过程简直就是一场恶梦。这就是为什么 SUN还要推出Java Server Pages的原因。

1.2 JSP技术

前面说过，Java Servlet的最大缺点就在于没有把网站的逻辑和页面的输出分开，导致整个 Servlet代码混乱不堪。为了解决Java Servlet的这种缺点，SUN推出了Java Server Pages —— JSP。

1.2.1 JSP技术概述

按照脚本语言是服务于某一个子系统的语言这种论述， JSP应当被看作是一种脚本语言，然而，作为一种脚本语言， JSP又显得过于强大了，在JSP中几乎可以使用全部的Java类。

作为一种基于文本的、以显示为中心的开发技术，JSP提供了Java Servlet 的所有好处，并且，当与一个JavaBeans 类结合在一起时，提供了一种使内容和显示逻辑分开的简单方式。分开内容和显示逻辑的好处是，更新页面外观的人员不必懂得 Java 代码，而更新JavaBeans类的人员也不必是设计网页的行家里手，就可以用带 JavaBeans 类的JSP 页面来定义Web 模板，以建立一个由具有相似的外观的页面组成的网站。JavaBeans 类完成数据提供，这样在模板中就没有Java 代码，这意味着这些模板可以由一个 HTML 编写人员来维护。当然，也可以利用 Java Servlet来控制网站的逻辑，通过Java Servlet调用JSP文件的方式来将网站的逻辑和内容分离。本章我们后面将对这种分离网站的逻辑和内容的设计方法做一些更深入的描述。

在选择使用一个Java Servlet，还是一个JSP 页面时，要记住的是，Java Servlet 是一个程序设计工具，它最适用于不需要频繁修改的低级应用功能；而JSP页面则通过以显示为中心的描述性的方法将动态内容和逻辑结合在一起。对于使用一个 JSP 页面的简单的基于Web 的应用程序，可以使用定制标记或者 Scriptlet，而不是使用JavaBeans 类来将内容与应用逻辑结合起来。定制标记被打包到一个标记库中，并被引入到一个 JSP 页面中。Scriptlet是直接嵌入在JSP 页面中的很小的Java代码段。

一般来说，在实际的 JSP引擎中，JSP页面在执行时是编译式，而不是解释式的。解释式的动态网页开发工具如 ASP、PHP3等由于速度等原因已经满足不了当前大型电子商务应用的需要了，传统的开发技术都在向编译执行的方式改变，如 ASP→ASP+；PHP3→PHP4。而尽管JSP的规范书中并没有要求实际的 JSP引擎要使用编译式的执行方式，但估计一般是不会使用解释的方式来执行JSP页面的。通常说来，JSP页面一般是翻译为 Servlet的Java源文件，再经过Java编译器编译为Servlet的class文件。为什么要编译为 Servlet呢？据说是为了让原先的 Servlet引擎可以直接服务于JSP，而JSP引擎就仅仅需要将 JSP转译为Servlet就可以了。这里要注意的是： JSP规范书中并没有规定如何将 JSP页面转译为 Servlet，因此，不同的 JSP引擎转译的结果也是不一样的。在JSP文件转译为 Servlet以后，每次客户机（通常是用户的 Web浏览器）向服务器请求这一个 JSP文件的时候，服务器将检查自上次编译后 JSP文件是否有改变，如果没有改变，就直接执行 Servlet，而不用再重新编译，其效率是相当高的。一般来说， JSP文件的编译是在第一个用户访问到这个JSP页面时发生，而这第一个用户通常是开发人员自己，这样，正式放在服务器上让用户访问的JSP文件一般都已经有了对应的编译好的 Servlet了。许多服务器都有设置，可以使 JSP文件在第一个用户访问之前就预先编译好，这样看来，效率就更高了。后面在第 4章中，将展示一个简单的JSP文件对应的 Servlet。

在JSP规范书中，并没有明确要求 JSP中的程序代码部分（称为 Scriptlet）一定要用 Java来写，实际上，有一些JSP引擎就是采用的其他脚本语言，如：EMAC-Script、WebL等等，但实际上这几种脚本语言也是构建在Java上面，编译为Servlet来实现的。按照JSP规范书，完全和Java没有任何关系的Scriptlet也是可以的，不过，由于JSP的强大功能主要在于能和JavaBeans、Enterprise JavaBeans一起工作，所以即使是Scriptlet部分不使用Java，编译成的执行代码也应该是与Java相关的。

1.2.2 JSP的优势及与其他Web开发工具的比较

和传统的CGI相比较，JSP有相当的优势。首先，在速度上，传统的 CGI程序需要使用系统

的标准输入输出设备来实现动态网页的生成，而 JSP是直接和服务器相关联的。而且对于 CGI来说，每一个访问就需要新增加一个进程来处理，进程不断地建立和销毁对于作为 Web服务器的计算机将是不小的负担。其次， JSP是专门为Web开发而设计的，其目的是为了建立基于 Web的应用程序，包含了一整套的规范和工具。使用 JSP技术可以很方便地将一大堆 JSP页面组合成为一个Web应用程序。

和ISPAI和NSAPI相比较，JSP的开发速度要快得多，开发难度也要小得多，在编译为 Java Servlet以后，配合目前最新的JIT（Just In Time）的Java解释器，其执行速度也慢不了多少。而且，ISAPI和NSAPI这种和Web服务器过于紧密结合的技术在使用时的一旦出现错误，很容易使Web服务器崩溃，而JSP就没有这个缺点。

JSP的真正对手是ASP和PHP，还有即将问世的ASP+，在Web技术方面ASP、PHP和JSP的比较见表1-1。

注意：这里的ASP指ASP3.0，JSP指JSP规范书1.1中指出的规范，PHP指PHP4。

表1-1 ASP、JSP、PHP的比较

	ASP	JSP	PHP
Web服务器	IIS、PWS	Apache、IIS、PWS、Netscape Server、iPlanet等	Apache、IIS、PWS、Netscape Server等等
运行平台	Windows	各种UNIX（Solaris、Linux、AIX、IRIX等）、Windows、MacOS等	各种UNIX（Solaris、Linux、AIX、IRIX等）、Windows
组件技术	COM	JavaBeans、EJB	COM、JavaBeans
自定义TAG语法	无	有	无
开放性	无	多家合作，包括 SUN、IBM、 BEA Weblogic、Netscape、Oracle	自由软件
脚本语言支持	VBScript、JScript	Java、EMAC-Script、WebL等	PHP
建立大型Web应用程序	可以	可以	不宜
程序执行速度	快	极快	极快
学习难度	低	较低	低
Session管理	有	有	有
统一的数据库连接	有、ADO、ODBC	有、JDBC	无
后缀名	asp	jsp	php、php3、phps

1. Web服务器和运行平台

ASP目前仅仅被支持于Microsoft Internet Information Server（IIS）和Personal Web Server（PWS），由于IIS和PWS仅仅有Windows下的版本，故ASP目前只能在Windows平台下使用。尽管有第三方的插件号称可以在 UNIX下使用ASP，但对基于COM组件技术的ASP来说，在没有COM支持的UNIX平台下只能是一个“玩具”。

JSP仅仅是一个规范，尽管通过前面的论述可以得出 JSP一般要用Java来实现的论断，但作为跨平台的语言，Java可以在许多平台下使用。这样， JSP也就显而易见的是跨平台的了。目前的

JSP的确可以在多种Web服务器和操作系统下使用。如Apache Web Server和Microsoft IIS等。

Apache Web Server是世界上占有率最高的Web服务器产品，可以在包括SUN Solaris、IBM AIX、SGI IRIX、Linux和Windows在内的许多操作系统下运行。Apache Web Server下JSP的实现可以通过免费的Apache Jserv和GNUJSP、Jakarta-Tomcat实现，也可以使用商业的JRUN（LiveSoftware）、Weblogic（BEA）、Websphere（IBM）来实现。

Microsoft IIS本身不直接支持JSP，但可以通过JRUN、Weblogic、Websphere来实现。

还可以使用应用服务器添加JSP支持的Netscape Enterprise Server及由之发展而来的可以直接支持JSP的iPlanet Web Server等等。

PHP本身就对各种操作系统和Web服务器做了支持，PHP目前可以作为Apache的一个附加模块直接编译进入Apache中去，由于Apache支持多种操作系统，PHP相应地也就可以在各种操作系统上实现。PHP也可以CGI方式或ISAPI方式插入到IIS或PWS中去。

2. 组件技术

ASP和JSP对组件技术的支持已经很完善了，而PHP直到前不久才开始支持COM和JavaBeans。但支持也不是很完善，如果PHP不能在将来完善对组件技术的支持，在大型Web应用程序方面将很难与JSP和ASP竞争。但由于PHP技术本身的易学易用，加上众多的函数支持和开放源代码的特性，在中小型Web站点的开发上，PHP还是会占有一席之地的。

其实，JSP本身对于ASP和PHP并没有明显的优势，JSP的强大是因为其后面有强大的Java技术做支持。包括JavaBeans和J2EE技术在内的Java技术是JSP强大生命力的所在。

Microsoft最新推出的ASP+技术和ASP技术相比有了许多激动人心的进步，但是从企业级应用的角度看，JSP技术仍然有相当的优势。有理由认为，在将来的Web开发中，中小型站点将出现JSP、ASP+和PHP三分天下的局面，但是对于大型的电子商务站点，JSP及J2EE技术将成为首选。

1.3 用JSP开发Web的几种主要方式

JSP作为J2EE的一部分，既可以用于开发小型的Web站点、也可以用于开发大型的、企业级的应用程序，本节将讲述对于不同规模的Web系统，使用JSP进行开发的不同方式。

1.3.1 直接使用JSP

对于最小型的Web站点，可以直接使用JSP来构建动态网页，这种站点最为简单，所需要的仅仅是简单的留言板、动态日期等基本的功能。对于这种开发模式，一般可以将所有的动态处理部分都放置在JSP的Scriptlet中，就像一般使用PHP或ASP开发动态网页一样。

1.3.2 JSP+JavaBeans

中型站点面对的是数据库查询、用户管理和小量的商业业务逻辑。对于这种站点，不能将所有的东西全部交给JSP页面来处理。在单纯的JSP中加入JavaBeans技术将有助于这种中型网站的开发。利用JavaBeans，将很容易完成如数据库连接、用户登录与注销、商业业务逻辑封装的任务。如：将常用的数据库连接写为一个Java Beans，既方便了使用，又可以使JSP文件简单而

清晰，通过封装，还可以防止一般的开发人员直接获得数据库的控制权。

本书的第一部分将主要讲述这种开发方式，在第 2章，将简要讲述如何开发JavaBeans，没有JavaBeans基础的读者不用担心。

1.3.3 JSP+JavaBeans+Servlet

无论用ASP还是PHP开发动态网站，长期以来都有一个比较重要的问题，就是网站的逻辑关系和网站的显示页面不容易分开。常常可以看见一些夹杂着 if……then……、case select或是if {……}和大量显示用的HTML代码的ASP、PHP页面，即使是有着良好的程序写作习惯的程序员，其作品也几乎无法阅读。另一方面，动态 Web的开发人员也在抱怨，将网站美工设计的静态页面和动态程序和并的过程是一个异常痛苦的过程。

如何解决这个问题呢？在 JSP问世以后，笔者的一位朋友认为 Servlet已经完全可以被JSP代替，然而，事实是 Servlet在不再担负动态页面生成的任务以后，开始担负起决定整个网站逻辑流程的任务。在逻辑关系异常复杂的网站中，借助于 Servlet和JSP良好的交互关系和JavaBeans的协助，完全可以将网站的整个逻辑结构放在 Servlet中，而将动态页面的输出放在 JSP页面中来完成。在这种开发方式中，一个网站可以有一个或几个核心的 Servlet来处理网站的逻辑，通过调用JSP页面来完成客户端（通常是 Web浏览器）的请求。后面我们将可以看到，在 J2EE模型中，Servlet的这项功能可以被EJB取代。

1.3.4 J2EE开发模型

在J2EE开发模型中，整个系统可以分为三个主要的部分：

1. 视图

视图就是用户界面部分，在 Web应用程序中也就是HTML、XML、JSP页面。这个部分主要处理用户看到的东西，动态的 JSP部分处理了用户可以看见的动态网页，而静态的网页则由HTML、XML输出。

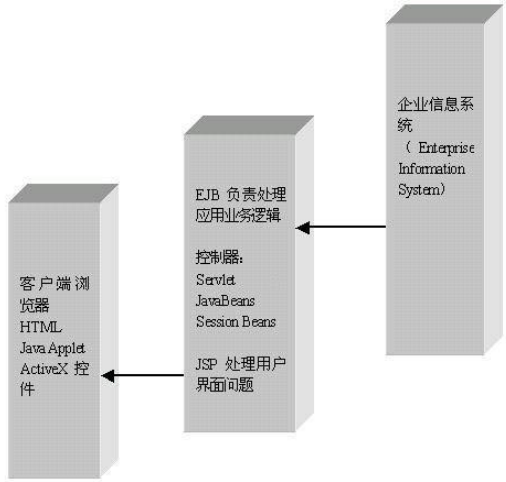

图1-1

2. 控制器。

控制器负责网站的整个逻辑。它用于管理用户与视图发生的交互。可以将控制器想像成处在视图和数据之间，对视图如何与模型交互进行管理。通过使视图完全独立于控制器和模型，就可以轻松替换前端客户程序，就是说，网页制作人员将可以独立自由地改变 Web页面而不用担心影响这个基于Web的应用程序的功能。

在J2EE中，控制器的功能一般是由 Servlet、JavaBeans、Enterprise JavaBeans中的SessionBean来担当的。

3. 模型

模型就是应用业务逻辑部分，这一部分的主要角色是 Enterprise JavaBeans，借助于EJB强大的组件技术和企业级的管理控制，开发人员可以轻松形创建出可重用的业务逻辑模块。

图1-1说明了上述三者在J2EE开发模型中的相互关系。

1.4 本书用到的软件及获取方法

1. JDK

Java Development Kit即Java开发工具包，进行JSP开发工作必须要有JDK。本书使用版本1.3，可以在http://java.sun.com找到。

同样，在SUN的主页上还可以找到一份JDK的中文文档，读者可能会觉得使用本地化的文档更方便。

2. Jakarta-Tomcat

Apache Jakarta项目组开发的基于GPL自由软件协议的JSP引擎，配合JDK就可以搭建起一个最简单的JSP试验平台。开放源代码，使用起来比较简单，缺点是不支持 J2EE技术，无法使用EJB。下载地址：http://jakarta.Apache.org。需要说明的是，原先的GNUJSP项目的开发人员现在已经完全转向了Jakarta-Tomcat项目，不建议使用ApacheJServ+GNUJSP搭建JSP平台，如果一定要使用，也可以在http://www.Apache.org找到需要的软件。

3. Apache Web Server

仅仅使用 Tomcat来搭建 JSP的实用平台是不够的，一般小型站点使用 JSP都是采用Apache+Tomcat的方式，因为作为独立的 Web Server，Tomcat的效率显然还不够，也不具有Apache的众多实用功能。Apache Web Server也可以在http://www.Apache.org下载。

4. J2SDKEE

在SUN的主页上可以找到被称为 J2SDKEE的JavaTM 2 SDK，Enterprise Edition开发工具。使用J2SDKEE可以用最小的开销获得一个J2EE的开发平台。下载地址：http://java.sun.com/j2ee。

5. IBM Websphere

要想开发基于J2EE技术的Web应用程序，最好是使用一个商业化的应用服务器（Application Server），IBM的Websphere作为应用服务器在国内使用得比较多，文档和例子也不少，学习起来比较方便。IBM公司提供免费的试用版供下载，可以在 IBM公司的主页上找到。IBM公司在国内也发布了不少光盘版的试用版，在各大 FTP站点上也有下载。

6. BEA Weblogic

作为业界领先的应用服务器提供商，BEA Weblogic应用服务器在世界上的市场占有率遥遥领先于其他的对手，不过BEA Weblogic应用服务器的使用难度大于IBM Websphere，而且文档几乎都是英文的，国内读者可能会觉得比较困难。BEA Weblogic应用服务器在BEASYS的主页上提供试用版本的下载地址为。http://www.beasys.com或http://www.Weblogic.com。

7. UltraEdit

UltraEdit并不是一个Java工具，但笔者认为它是一个比较好的JSP写作工具，在UltraEdit的主页上可以得到UltraEdit的试用版和支持Java2和JSP的语法文件地址为。http://www.ultraedit.com。

读者亦可以从其他的JSP引擎入手学习JSP，如LiveSoftware JRUN、Inprise Application Server、ServletExec等等，也有许多好的Java开发工具和文本编辑器可以使用，如：Inprise Jbuilder 3.5、IBM Virsual Age For Java、Symantec Virsual Café、EditPlus、JavaPad等等。

最后说明一下本书的代码写作规范：

类名：类名由几个单词构成时，将这几个单词紧靠在一起，且大写每一个单词的首字母。

其他：和类名相仿，但是第一个单词的首字母小写。

每个程序块统一缩进一个Tab制表符。但是对于那种长代码较多的代码段，统一取消缩进。

注释单独列为一行。

每行代码长度不超过60个字符（为了出版的需要）。

如：

```
class TestClass{
      int a;
      void TestFuncs() {
           ......
      }
}
```

第2章 预备知识

2.1 Java程序设计基础

Java是JSP的基础，要学习JSP技术，Java基础是必不可少的。本节将简要介绍Java的基本语法和概念。已经是Java编程人员的读者就不用阅读了，这里是为没有多少Java经验的读者提供一个快速入门的方法。这里对Java语言的介绍仅仅是一个基本的概况，要想深入学习JSP，必须对Java语言有深刻的理解，笔者推荐机械工业出版社翻译出版的《Java编程思想》一书，本书限于篇幅，就不多讲了。

2.1.1 Java语言规则

Java语言的基本结构像C/C++，任何用面向过程语言编写过程序的人都可以了解Java语言的大部分结构。

1. 程序结构

Java语言的源程序代码由一个或多个编译单元(compilationunit)组成，每个编译单元只能包含下列内容(空格和注释除外)：

- 程序包语句(package statement)。
- 入口语句(import statements) 。
- 类的声明(class declarations) 。
- 界面声明(interface declarations)。

每个Java的编译单元可包含多个类或界面，但是每个编译单元最多只能有一个类或者界面是公共的。Java 的源程序代码被编译后，便产生了Java字节代码。Java的字节代码由一系列不依赖于机器的指令组成，这些指令能被Java的运行系统(runtime system)有效地解释。Java的运行系统工作起来如同一台虚拟机。在当前的Java实现中，每个编译单元就是一个以.java为后缀的文件。每个编译单元有若干个类，编译后，每个类生成一个.class文件。.class文件是Java虚拟机能够识别的代码。 在引入了JAR这个概念以后，现在可以把许多Java的class文件压缩进入一个JAR文件中。新版本的Java已经可以直接读取JAR文件加以执行。

2. 注释

注释有三种类型:

// 注释一行

/* 一行或多行注释 */

/** 文档注释 **/

文档注释一般放在一个变量或函数定义之前，表示在任何自动生成文档系统中调入，提取注释生成文档的工具叫做javadoc，其中还包括一些以@开头的变量，如：@see、@version、

@param等等，具体用法参见JDK自带的工具文档。

3. 标识符

变量、函数、类和对象的名称都是标识符，程序员需要标识和使用的东西都需要标识符。在Java语言里，标识符以字符_或$开头，后面可以包含数字，标识符是大小写有区别的，没有长度限制。

有效的标识符如： gogogo brood_war Hello _and_you $bill。

声明如：

```
int a_number;
char _onechar;
float $bill。
```

以下为Java的关键字：

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

以下单词被保留使用：cast、future、generic、inner、operator、outer、rest、var。

4. 数据类型

Java使用五种基本类型:integer(整数)，floating(浮点数)，point(指针)，Boolean(布尔变量)，Character or String(字符或字符串)。此外，还有一些复合的数据类型，如数组等。

Integer 包含下面几种类型：

整数长度 (Bits)	数据类型表示
8	byte
16	short
32	int
64	long

floating 下边给出的数据表示都是浮点数的例子：

3.14159，3.123E15，4e5

浮点数长度(Bits)	数据类型表示
32	float
64	double

Boolean 下边是布尔变量的两种可能取值：

true false

Character 下边给出的都是字符的例子：

a s d f

String 下边给出的都是字符串的例子：

"gogogo,rock and roll" "JSP高级编程"

数组 可以定义任意类型的数组，如char s[]，这是字符型数组； int array []，这是整型数组；还可以定义数组的数组. intblock[][]=new int [2][3]；数组边界在运行时被检测，以避免堆栈溢出。

在Java里，数组实际上是一个对象，数组有一个成员变量： length。可以用这个成员函数来查看任意数组的长度。

在Java里创建数组，可使用两种基本方法：

1) 创建一个空数组。

```
int list[]=new int[50];
```

2) 用初始数值填充数组.

```
String names[] = { "Chenji", "Yuan", "Chun", "Yang" };
```

它相当于下面功能:

```
String names[];
names = new String[4];
names[0]=new String("Chenji");
names[1]=new String("Yuan");
names[2]=new String("Chun");
names[3]=new String("Yang");
```

在编译时不能这样创建静态数组：

int name[50]; //将产生一个编译错误

也不能用new操作去填充一个没定义大小的数组。如：

```
int name[];
for (int i=0;i<9; i++) {
            name[i] = i;
      }
```

5. 表达式

Java语言的发展中有许多是从C语言借鉴而来的，所以Java的表达式和C语言非常类似。

运算符

运算符(operator)优先级从高到低排列如下： .

```
[ ] () ++ -- !   ~ instanceof * / % + - << >> >>> < > <= >\ == ! = & ^ && || ? : = op =          ,
```

(2) 整数运算符

在整数运算时，如果操作数是long类型，则运算结果是long类型，否则为int类型，绝不会是byte，short或char型。这样，如果变量i被声明为short或byte，i+1的结果会是int。如果结果超过该类型的取值范围，则按该类型的最大值取模。单目整数运算符是：

运算符	操作
-	非
~	位补码
++	加1
--	减1

++运算符用于表示直接加 1操作。增量操作也可以用加运算符和赋值操作间接完成。 ++lvalue （左值表示lvalue+=1， ++lvalue 也表示lvalue =lvalue +1 (只要lvalue没有副作用)。--运算符用于表示减1操作。++和--运算符既可以作为前缀运算符，也可以作为后缀运算符。 双目整数运算符是：

运算符	操作
+	加
-	减
*	乘
/	除
%	取模
&	位与
|	位或
^	位异或
<<	左移
>>	右移(带符号)
>>>	添零右移

整数除法按零舍入。除法和取模遵守以下等式：(a/b) * b + (a%b) == a。整数算术运算的异常是由于除零或按零取模造成的。它将引发一个算术异常，下溢产生零，上溢导致越界。例如：加1超过整数最大值，取模后，变成最小值。一个 op=赋值运算符，和上表中的各双目整数运算符联用，构成一个表达式。整数关系运算符 <，>，<=，>=，==和!=产生boolean类型的数据。

(3) 布尔运算符

布尔(boolean)变量或表达式的组合运算可以产生新的 boolean值。 单目运算符!是布尔非。双目运算符&、|和^是逻辑AND、OR和XOR运算符，它们强制两个操作数求布尔值。为避免右侧操作数冗余求值，用户可以使用短路求值运算符&& 和||。用户可以使用==和!=，赋值运算符也可以用 &=、|=、^=。 三元条件操作符?:和C语 言中的一样。

(4) 浮点运算符

浮点运算符可以使用常规运算符的组合，如单目运算符 ++、--，双目运算符+、-、* 和/，以及赋值运算符+=，- =，*=，和/=。 此外，还有取模运算：%和%=也可以用于浮点数，例如：a%b和a-((int) (a/b)*b)的语义相同。这表示a%b的结果是除完后剩下的浮点数部分。只有单精度操作数的浮点表达式按照单精度运算求值，产生单精度结果。如果浮点表达式中含有一个或一个以上的双精度操作数，则按双精度运算，结果是双精度浮点数。

(5) 数组运算符

数组运算符形式如下：

```
<expression> [ <expression>]
```

可给出数组中某个元素的值。合法的取值范围是从 0到数组的长度减1。取值范围的检查只在运行时刻实施。

(6) 对象运算符

双目运算符instanceof 测试某个对象是否是指定类或其子类的实例。例如：

```
if (myObject instanceof MyClass) {
```

```
        MyClass anothermyObject=( MyClass) myObject;
        ...
}
```

是判定myObject**是否是**MyClass**的实例或是其子类的实例。**

(7) **强制和转换**

Java**语言和解释器限制使用强制和转换，以防止出错导致系统崩溃。整数和浮点数间可以来回强制转换，但整数不能强制转换成数组或对象。对象不能被强制为基本类型。**

6. Java**流控制**

下面几个控制结构是从C**语言借鉴的。**

(1) **分支结构**

if/else**分支结构：**

```
if (Boolean) {
            statemanets;
       }
else {
            statements;
}
```

switch**分支结构：**

```
switch(expr1) {
     case expr2:
          statements;
           break;
     case expr3:
          statements;
          break;
     default:
          statements;
          break;
     }
```

(2) **循环结构**

for**循环结构：**

```
for (init expr1;test expr2;increment expr3) {
     statements;
}
```

While**循环结构：**

```
While(Boolean) {
        statements;
}
```

Do**循环结构：**

```
do
        statements;
} while (Boolean);
```

2.1.2 Java变量和函数

Java的类包含变量和函数。数据变量可以是原始的类型，如 int、char等。成员函数是可执行的过程。例如，下面的程序：

```
public class TestClass
public TestClass () {
          i=10;
     }
     public void addI(int j) {
          i=i+j;
     }
}
```

TestClass包含一个变量i和两个成员函数，TestClass(int first)和addI(int j)。

成员函数是一个可被其他类或自己类调用的处理子程序。一个特殊的成员函数叫构造函数，这个函数名称一般与本类名称相同。它没有返回值。

在Java里定义一个类时，可定义一个或多个可选的构造函数，当创建本类的一个对象时，用某一个构造函数来初始化本对象。用前面程序例子来说明，当 TestClass类创建一个新实例时，所有成员函数和变量被创建(创建实例)。构造函数被调用。

```
TestClass testObject;
testObject = new TestClass();
```

关键词new用来创建一个类的实例，一个类用 new初始化前并不占用内存，它只是一个类型定义，当testObject对象初始化后，testObject对象里的i变量等于10。可以通过对象名来引用变量i。(有时称为实例变量) testObject.i++;// testObject实例变量加1，因为testObject有TestClass类的所有变量和成员函数，可以使用同样的语法来调用成员函数 addI：addI(10); 现在testObject.i变量等于21。

Java并不支持析构函数(C++里的定义)，因为java对象无用时，有自动清除的功能，同时 它也提供了一个自动垃圾箱的成员函数，在清除对象时被调用：

```
Protected void finalize() { close(); }
```

2.1.3 子类

子类是利用存在的对象创建一个新对象的机制，比如，如果有一个 Horse类，你可以创 建一个 Zebra子类，Zebra是Horse的一种。

```
class Zebra extends Horse { int number_OF_stripes: }
```

关键词extends来定义对象有的子类.Zebra是Horse的子类。Horse类里的所有特征都将拷贝到Zebra类里，而Zebra类里可以定义自己的成员函数和实例变量。Zebra称为Horse的派生类或继承。另外，你也许还想覆盖基类的成员函数，可用 TestClass说明，下面是一派生类 覆盖AddI功能的例子。

```
import TestClass;
public class NewClass extends TestClass {
```

```
    public void AddI(int j) {
        i=i+(j/2);
    }
}
```

当NewClass类的实例创建时，变量i初始化值为10，但调用AddI产生不同的结果。

```
NewClass newObject;
newObject=new NewClass();
newObject.AddI(10);
```

当创建一个新类时，可以标明变量和成员函数的访问层次。

public public void AnyOneCanAccess(){} public实例变量和成员函数可以由任意其他类调用。

protected protected void OnlySubClasses(){} protected实例变量和成员函数只能被其子类调用。

private private String CreditCardNumber; private实例变量和成员函数只能在本类里调 用。

friendly void MyPackageMethod(){}是缺省的，如果没有定义任何访问控制，实例变量或函数缺省定义成friendly，这意味着可以被本包里的任意对象访问，但其他包里的对象不可访问。

对于静态成员函数和变量，有时候，你创建一个类，希望这个类的所有实例都公用一个变量。就是说，所有这个类的对象都只有实例变量的同一个拷贝。这种方法的关键词为 static， 例如：

```
class Block {
      static int number=50;
}
```

所有从Block类创建的对象的number变量值都是相同的。无任在哪个对象里改变了 number的值， 所有对象的number都跟着改变。同样，可以定义static成员函数，但这个成员函数不能访问非static函数和变量。

```
class Block {
      static int number = 50;
      int localvalue;
      static void add_local(){
            localvalue++;  //没有运行
      }
      static void add_static() {
            number++;//运行
      }
}
```

2.1.4 this和super

访问一个类的实例变量时， this关键词是指向这个类本身的指针，在前面的 TestClass例子中，可以增加构造函数如下：

```
public class TestClass {
```

```
        int i;
        public TestClass() {
             i = 10;
        }
        public TestClass (int value) {
    this.i = value;
             }
        public void AddI(int j) {
        i = i + j;
             }
    }
```

这里，this指向TestClass类的指针。如果在一个子类里覆盖了父类的某个成员函数，但又想调用父类的成员函数，可以用super 关键词指向父类的成员函数。

```
    import TestClass;
    public class NewClass extends TestClass {
       public void addI (int j) {
              i = i+(j/2);
              super.addI (j);
    }
       }
```

下面程序里，i变量被构造函数设为10，然后为15，最后被父类(TestClass)设为25。

```
    NewClass newObject;
    newObject = new NewClass();
    newObject.addI(10);
```

2.1.5 类的类型

迄今为止，在类前面只用了一个public关键词，其实它有下面4种选择：

abstract。一个abstract类必须至少有一个虚拟函数，一个abstract类不能直接创建对象，必须继承子类后才能创建对象。

final一个final类声明了子类链的结尾，用final声明的类不能再派生子类。

Public。public类能被其他的类访问。在其他包里，如果想使用这个类，必须先 import，否则它只能在它定义的package里使用。

synchronicable。这个类标识表示所有类的成员函数都是同步的。

2.1.6 抽象类

面向对象的一个最大优点就是能够定义怎样使用这个类而不必真正定义好成员函数。当程序由不同的用户实现时，这是很有用的，这不需用户使用相同的成员函数名。

在java里，Graphics类中一个abstract类的例子如下：

```
public abstract class Graphics {
public abstract void drawLine(int x1, int y1, int x2,  int y2);
public abstract void drawOval(int x, int y, int width,  int height);
```

```
public abstract void drawRect(int x, int y, int width,  int height);
}
```

在Graphics类里声明了几个成员函数，但成员函数的实际代码是在另外一个地方实现的。

```
public class MyClass extends Graphics { public void drawLine (int x1, int y1, int x2,
int y2) { <画线程序代码> } }
```

当一个类包含一个abstract成员函数时，这个类必须定义为abstract类。然而并不是abstract类的所有成员函数都是abstract的。Abstract类不能有私有成员函数(它们不能被实现)，也不能有静态成员函数。

2.1.7 接口

当确定多个类的操作方式都很相像时， abstract成员函数是很有用的。但如果需要使用这个abstract成员函数，必须创建一个新类，这样有时很繁琐。接口提供了一种抽象成员函数的有利方法。一个接口包含了在另一个地方实现的成员函数的收集。成员函数在接 口里定义为public和abstract。 接口里的实例变量是public、static和final。接口和抽象的主要区别是，一个接口提供了封装成员函数协议的方法而不必强迫用户继承类。

例如：

```
public interface AudiClip {
      //Start playing the clip.
      void play();
      //Play the clip in a loop.
      void loop();
      //Stop playing the clip
      void stop();
}
```

想使用Audio Clip接口的类使用implenents关键词来提供成员函数的程序代码。

```
class MyClass implements AudioClip {
      void play(){ <实现代码> }
      void loop <实现代码> }
      void stop <实现代码> }
}
```

接口的优点是：一个接口类可以被任意多的类实现，每个类可以共享程序接口而不必关心其他类是怎样实现的。

2.1.8 包

包(Package)由一组类(class)和界面(interface)组成。它是管理大型名字空间，避免名字冲 突的工具。每一个类和界面的名字都包含在某个包中。按照一般的习惯，它的名字由“ .”号分隔的单词构成，第一个单词通常是开发这个包的组织的名称。

定义一个编译单元的包由 package语句定义。如果使用 package语句，编 译单元的第一行必须无空格，也无注释。其格式如下：

```
package packageName;
```

若编译单元无package语句，则该单元被置于一个缺省的无名的包中。

在Java语言里，提供了一个包可以使用另一个包中类和界面的定义和实现的机制。用 import关键词来标明来自其他包中的类。一个编译单元可以自动把指定的类和界面输入到它自己 的包中。在一个包中的代码可以有两种方式定义自其他包中的类和界面：在每个引用的类和界面前面给出它们所在的包的名字：

```
//前缀包名法 acme. project.FooBar
obj=new acme. project. FooBar( );
```

使用import语句引入一个类或一个界面，或包含它们的包。引入的类和界面的名字在当前的名字空间可用。引入一个包时，则该包所有的公有类和界面均可用。其形式如下：

```
// 从 acme.project 引入所有类
import acme.project.*;
```

这个语句表示acme.project中所有的公有类被引入当前包。以下语句从 acme. project包 中进入一个类Employec_List。

```
//从 acme. project而引入Employee_List
import acme.project.Employee_list;
Employee_List obj = new Employee_List( );
```

在使用一个外部类或界面时，必须要声明该类或界面所在的包，否则会产生编译错误。

import(引入)类包(class package)用import关键词调入指定package名字如路径和类名，用 *匹配符可以调入多于一个类名。

```
import java.Date; import java.awt.*;
```

如果java源文件不包含package，它放在缺省的无名package。这与源文件同目 ，类可以这样引入:

```
import MyClass;
```

Java系统包：Java语言提供了一个包含窗口工具箱、实用程序、一般 I/O、工具和网络功能的包。

各个包的用法和类详解在 JDK自带的文档中都有详细的说明，希望读者能够好好的看一看，对大部分的常用包的类都熟悉了，才能更好地掌握 Java这门技术。

在了解Java语言的概况以后，接下来看一看与 JSP技术密切相关的三种Java技术：JavaEeans，JDBC，Java Servlet。

2.2 JavaBeans

JavaBeans是什么？JavaBeasns是一个特殊的类，这个类必须符合 JavaBeans规范。JavaBeans原来是为了能够在一个可视化的集成开发环境中可视化、模块化地利用组件技术开发应用程序而设计的。不过，在 JSP中，不需要使用任何可视化的方面，但仍然需要利用 JavaBeans的属性、事件、持久化和用户化来实现模块化的功能。下面分别介绍 JavaBeans的属性、事件、持久化和用户化。

2.2.1 JavaBeans的属性

JavaBeans的属性与一般Java程序中所指的属性，或者说与所有面向对象的程序设计语言中对象的属性是一个概念，在程序中的具体体现就是类中的变量。在 JavaBeans设计中，按照属性的不同作用又细分为四类： Simple, Index, Bound与Constrained属性。

1. Simple属性

Simple 属性表示伴随有一对get/set方法（C语言的过程或函数在Java程序中称为”方法”）的变量。属性名与和该属性相关的 get/set方法名对应。例如：如果有 setX和getX方法，则暗指有一个名为”X”的属性。如果有一个方法名为 isX，则通常暗指”X”是一个布尔属性（即X的值为true或false）。例如，在下面这个程序中：

```
public class alden1 extends Canvas {
      string ourString= "Hello";
      //属性名为ourString，类型为字符串
      public alden1(){
      //alden1()是alden1的构造函数，与C++中构造函数的意义相同
            setBackground(Color.red);
            setForeground(Color.blue);
      }
      /* "set"属性*/
      public void setString(String newString) {
            ourString=newString;
}
      /* "get"属性 */
      public String getString() {
            return ourString;
      }
}
```

2. Indexed属性

Indexed属性表示一个数组值。使用与该属性对应的 set/get方法可取得数组中的数值。该属性也可一次设置或取得整个数组的值。例如：

```
public class alden2 extends Canvas {
     int[] dataSet={1,2,3,4,5,6};
     // dataSet是一个indexed属性
     public alden2() {
           setBackground(Color.red);
           setForeground(Color.blue);
     }
     /* 设置整个数组 */
     public void setDataSet(int[] x){
           dataSet=x;
     }
     /* 设置数组中的单个元素值 */
     public void setDataSet(int index, int x){
           dataSet[index]=x;
```

```
        }
        /* 取得整个数组值 */
        public int[] getDataSet(){
              return dataSet;
        }
        /* 取得数组中的指定元素值 */
        public int getDataSet(int x){
              return dataSet[x];
        }
}
```

3. Bound属性

Bound属性是指当该种属性的值发生变化时，要通知其他的对象。每次属性值改变时，这种属性就触发一个PropertyChange事件(在Java程序中，事件也是一个对象)。事件中封装了属性名、属性的原值、属性变化后的新值。这种事件传递到其他的 Beans，至于接收事件的Beans应做什么动作，由其自己定义。

当PushButton的background属性 与Dialog的background属性绑定时，若 PushButton的background属性发生变化，Dialog的background属性也发生同样的变化。例如：

```
public class alden3 extends Canvas{
      String ourString= "Hello";
      //ourString是一个bound属性
      private PropertyChangeSupport changes = new PropertyChangeSupport(this);
/*Java是纯面向对象的语言，如果要使用某种方法则必须指明是要使用哪个对象的方法，在下面的程序中要进行点火事件的操作，这种操作所使用的方法是在PropertyChangeSupport类中的。所以上面声明并实例化了一个changes对象，在下面将使用changes的firePropertyChange方法来点火ourString的属性改变事件。*/

public void setString(string newString){
      String oldString = ourString;
      ourString = newString;
      /* ourString的属性值已发生变化，于是接着点火属性改变事件 */
      changes.firePropertyChange("ourString",oldString,newString);
}
public String getString(){
       return ourString;
}

/** 以下代码是为开发工具所使用的。我们不能预知alden3将与哪些其他的Beans组合成为一个应用，无法预知若alden3的ourString属性发生变化时有哪些其他的组件与此变化有关，因而alden3这个Beans要预留出一些接口给开发工具，开发工具使用这些接口，把其他的JavaBeans对象与alden3挂接。*/

public void addPropertyChangeListener(PropertyChangeLisener l){
      changes.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l){
      changes.removePropertyChangeListener(l);
}
```

通过上面的代码，开发工具调用changes的addPropertyChangeListener方法把其他JavaBean注册入ourString属性的监听者队列l中，l是一个Vector数组，可存储任何Java对象。开发工具也可使用changes的removePropertyChangeListener方法，从l中注销指定的对象，使alden3的ourString属性的改变不再与这个对象有关。当然，当程序员手写代码编制程序时，也可直接调用这两个方法，把其他Java对象与alden3挂接。

4. Constrained属性

JavaBeans的Constrained属性是指当这个属性的值要发生变化时，与这个属性已建立了某种连接的其他Java对象可否决属性值的改变。Constrained属性的监听者通过抛出PropertyVetoException来阻止该属性值的改变。

例如：下面程序中的Constrained属性是PriceInCents。

```
public class JellyBean extends Canvas{
    private PropertyChangeSupport changes=new PropertyChangeSupport(this);
    private VetoableChangeSupport Vetos=new VetoableChangeSupport(this);
    /*与前述changes相同，可使用VetoableChangeSupport对象的实例Vetos中的方法，在特定条件下来阻止PriceInCents值的改变。*/
.....
  public void setPriceInCents(int newPriceInCents) throws PropertyVetoException {
/*  方法名中throws PropertyVetoException的作用是当有其他Java对象否决PriceInCents的改变时，要抛出例外。*/
    /* 先保存原来的属性值*/
    int oldPriceInCents=ourPriceInCents;
    /**点火属性改变否决事件*/
    vetos.fireVetoableChange
        ("priceInCents",new Integer(OldPriceInCents), new Integer(newPriceInCents));
/**若有其他对象否决priceInCents的改变，则程序抛出例外，不再继续执行下面的两条语句，方法结束。若无其他对象否决priceInCents的改变，则在下面的代码中把ourPriceIncents赋予新值，并点火属性改变事件*/
  ourPriceInCents=newPriceInCents;
  changes.firePropertyChange
        ("priceInCents", new Integer(oldPriceInCents),new  Integer(newPriceInCents));
  }
  /**与前述changes相同，也要为PriceInCents属性预留接口，使其他对象可注册入PriceInCents否决改变监听者队列中，或把该对象从中注销
    public void addVetoableChangeListener(VetoableChangeListener l){
        vetos.addVetoableChangeListener(l);
    }
    public void removeVetoableChangeListener(VetoableChangeListener l){
        vetos.removeVetoableChangeListener(l);
    }
.....
}
```

从上面的例子中可看到，一个Constrained属性有两种监听者：属性变化监听者和否决属性改变的监听者。否决属性改变的监听者在自己的对象代码中有相应的控制语句，在监听到有Constrained属性要发生变化时，在控制语句中判断是否应否决这个属性值的改变。

总之，某个Beans的Constrained属性值可否改变取决于其他的 Beans或者是Java对象是否允许这种改变。允许与否的条件由其他的 Beans或Java对象在自己的类中进行定义。

2.2.2 JavaBeans的事件

事件处理是JavaBeans体系结构的核心之一。通过事件处理机制，可让一些组件作为事件源，发出可被描述环境或其他组件接收的事件。这样，不同的组件就可在构造工具内组合在一起，组件之间通过事件的传递进行通信，构成一个应用。从概念上讲，事件是一种在“源对象”和“监听者对象”之间某种状态发生变化的传递机制。事件有许多不同的用途，例如在 Windows系统中常要处理的鼠标事件、窗口边界改变事件、键盘事件等。在 Java和JavaBeans中则定义了一个一般的、可扩充的事件机制，这种机制能够：

- 对事件类型和传递模型的定义和扩充提供一个公共框架，并适合于广泛的应用。
- 与Java语言和环境有较高的集成度。
- 事件能被描述环境捕获和触发。
- 能使其他构造工具采取某种技术在设计时直接控制事件、事件源和事件监听者之间的联系。
- 事件机制本身不依赖于复杂的开发工具。

特别地，还应当：

- 能够发现指定的对象类可以生成的事件。
- 能够发现指定的对象类可以观察（监听）到的事件。
- 提供一个常规的注册机制，允许动态操纵事件源与事件监听者之间的关系。
- 不需要其他的虚拟机和语言即可实现。
- 事件源与监听者之间可进行高效的事件传递。
- 能完成JavaBean事件模型与相关的其他组件体系结构事件模型的中立映射。

1. 概述

JavaBeans事件模型总体结构的主要构成： 事件从事件源到监听者的传递是通过对目标监听者对象的Java方法调用进行的。 对每个明确的事件发生，都相应地定义一个明确的 Java方法。这些方法都集中定义在事件监听者（EventListener）接口中，这个接口要继承java.util.EventListener。实现了事件监听者接口中一些或全部方法的类就是事件监听者。 伴随着事件的发生，相应的状态通常都封装在事件状态对象中，该对象必须继承自 java.util.EventObject。事件状态对象作为单参传递给应响应该事件的监听者方法中。 发出某种特定事件的事件源的标识是：遵从规定的设计格式为事件监听者定义注册方法，并接受对指定事件监听者接口实例的引用。 有时，事件监听者不能直接实现事件监听者接口，或者还有其他的额外动作时，就要在一个源与其他一个或多个监听者之间插入一个事件适配器类的实例，以建立它们之间的联系。

2. 事件状态对象

与事件发生有关的状态信息一般都封装在一个事件状态对象中，这种对象是java.util.EventObject的子类。按设计习惯，这种事件状态对象类名应以Event结尾。例如：

```
public class MouseMovedExampleEvent extends java.util.EventObject{
```

```
    protected int x, y;
    /*  创建一个鼠标移动事件MouseMovedExampleEvent */
   MouseMovedExampleEvent(java.awt.Component source, Point location) {
       super(source);
       x = location.x;
       y = location.y;
    }
    /* 获取鼠标位置*/
    public Point getLocation() {
        return new Point(x, y);
    }
}
```

3. 事件监听者接口与事件监听者

由于Java事件模型是基于方法调用的，因而需要一个定义并组织事件操纵方法的方式。JavaBeans中，事件操纵方法都被定义在继承了java.util.EventListener类的事件监听者（EventListener）接口中，按规定，EventListener接口的命名要以Listener结尾。任何一个类如果想操纵在EventListener接口中，定义的方法都必须以实现这个接口方式进行。这个类就是事件监听者。

例如：

```
/*先定义了一个鼠标移动事件对象*/
public class MouseMovedExampleEvent extends java.util.EventObject {
// 在此类中包含了与鼠标移动事件有关的状态信息
     ...
}
/*定义了鼠标移动事件的监听者接口*/
interface MouseMovedExampleListener extends java.util.EventListener {
/*在这个接口中定义了鼠标移动事件监听者所应支持的方法*/
     void mouseMoved(MouseMovedExampleEvent mme);
}
```

在接口中只定义方法名，方法的参数和返回值类型。如上面接口中的 mouseMoved方法的具体实现是在下面的ArbitraryObject类中定义的：

```
class ArbitraryObject implements MouseMovedExampleListener {
public void mouseMoved(MouseMovedExampleEvent mme) {
               ...
     }
}
```

ArbitraryObject就是MouseMovedExampleEvent事件的监听者。

4. 事件监听者的注册与注销

为了让各种可能的事件监听者把自己注册入合适的事件源中，就建立源与事件监听者间的事件流，事件源必须为事件监听者提供注册和注销的方法。在前面的 bound属性介绍中，已看到了这种使用过程，在实际中，事件监听者的注册和注销要使用标准的设计格式：

```
public void add< ListenerType>(< ListenerType> listener);
public void remove< ListenerType>(< ListenerType> listener);
```

例如：

首先定义了一个事件监听者接口：

```
public interface ModelChangedListener extends java.util.EventListener {
     void modelChanged(EventObject e);
}
```

接着定义事件源类：

```
public abstract class Model {
    private Vector listeners = new Vector();
    // 定义了一个存储事件监听者的数组
    /*上面设计格式中的< ListenerType>在此处即是下面的ModelChangedListener*/
public synchronized void addModelChangedListener(ModelChangedListener mcl){
    listeners.addElement(mcl);
}
//把监听者注册入listeners数组中
public synchronized void removeModelChangedListener(ModelChangedListener mcl){
listeners.removeElement(mcl);
//把监听者从listeners中注销
}
/*以上两个方法的前面均冠以synchronized，是因为运行在多线程环境时，可能同时有几个对象同时要进行注册和注销操作，使用synchronized来确保它们之间的同步。开发工具或程序员使用这两个方法建立源与监听者之间的事件流*/
    protected void notifyModelChanged() {
    /**事件源使用本方法通知监听者发生了modelChanged事件*/
       Vector l;
      EventObject e = new EventObject(this);
      /* 首先要把监听者拷贝到l数组中，冻结EventListeners的状态以传递事件。这样来确保在事件传递到所有监听者之前，已接收了事件的目标监听者的对应方法暂不生效。*/
      synchronized(this) {
       l = (Vector)listeners.clone();
      }
      for (int i = 0; i < l.size(); i++) {
       /* 依次通知注册在监听者队列中的每个监听者发生了modelChanged事件，并把事件状态对象e作为参数传递给监听者队列中的每个监听者*/
    ((ModelChangedListener)l.elementAt(i)).modelChanged(e);
          }
    }
    }
```

在程序中可见，事件源Model类显式地调用了接口中的modelChanged方法，实际是把事件状态对象e作为参数，传递给了监听者类中的modelChanged方法。

5. 适配类

适配类是Java事件模型中极其重要的一部分。在一些应用场合，事件从源到监听者之间的传递要通过适配类来“转发”。例如：当事件源发出一个事件，而有几个事件监听者对象都可接收该事件，但只有指定对象做出反应时，就要在事件源与事件监听者之间插入一个事件适配器类，由适配器类来指定事件应该是由哪些监听者来响应。

适配类成为了事件监听者，事件源实际是把适配类作为监听者注册入监听者队列中，而真

正的事件响应者并未在监听者队列中，事件响应者应做的动作由适配类决定。目前绝大多数的开发工具在生成代码时，事件处理都是通过适配类来进行的。

2.2.3 持久化

当JavaBeans在构造工具内被用户化，并与其他 Beans建立连接之后，它的所有状态都应当可被保存，下一次被装载进构造工具内或在运行时，就应当是上一次修改完的信息。为了能做到这一点，要把Beans的某些字段的信息保存下来，在定义 Beans时要使它实现java.io.Serializable接口。例如：

```
public class Button implements java.io.Serializable {
}
```

实现了序列化接口的 Beans中字段的信息将被自动保存。若不想保存某些字段的信息则可在这些字段前冠以transient或static关键字，transient和static变量的信息是不可被保存的。通常，一个Beans所有公开出来的属性都应当是被保存的，也可有选择地保存内部状态。 Beans开发者在修改软件时，可以添加字段，移走对其他类的引用，改变一个字段的 private/protected/public状态，这些都不影响类的存储结构关系。然而，当从类中删除一个字段，改变一个变量在类体系中的位置，把某个字段改成 transient/static，或原来是transient/static，现改为别的特性时，都将引起存储关系的变化。

JavaBeans的存储格式

JavaBeans组件被设计出来后，一般是以扩展名为 jar的Zip格式文件存储，在 jar中包含与JavaBeans有关的信息，并以 MANIFEST文件指定其中的哪些类是 JavaBeans。以jar文件存储的JavaBeans在网络中传送时极大地减少了数据的传输数量，并把 JavaBeans运行时所需要的一些资源捆绑在一起。

这里主要论述了JavaBeanss的一些内部特性及其常规设计方法，参考的是 JavaBeanss规范书。随着世界各大ISV对JavaBeanss越来越多的支持，规范在一些细节上还在不断演化，但基本框架不会再有大的变动。

2.2.4 用户化

JavaBeans开发者可以给一个Beans添加定制器（Customizer）、属性编辑器（PropertyEditor）和BeanInfo接口来描述一个Beans的内容， Beans的使用者可在构造环境中通过与 Beans附带在一起的这些信息来用户化 Beans的外观和应做的动作。一个 Beans不必都有 BeanCustomizer、PrpertyEditor和BeanInfo，根据实际情况，这些是可选的，当有些 Beans较复杂时，就要提供这些信息，以Wizard的方式使Bean的使用者能够定制一个 Beans。有些简单的Beans可能没有这些信息，则构造工具可使用自带的透视装置，透视出 Beans的内容，并把信息显示到标准的属性表或事件表中供使用者定制 Beans，前几节提到的Beans的属性、方法和事件名要以一定的格式命名，主要的作用就是供开发工具对 Beans进行透视。当然也是给程序员在手写程序中使用 Beans提供方便，使其能观其名，知其意。

1. 定制器接口

当一个Bean有了自己的定制器时，在构造工具内就可展现出自己的属性表。在定义定制器时必须要实现java.beans.Customizer接口。例如，下面是一个“按钮”Beans的定制器：

```
public class OurButtonCustomizer extends Panel implements Customizer {
    ... ...
/*当实现像OurButtonCustomizer这样的常规属性表时，一定要在其中实现addProperChangeListener和
removePropertyChangeListener,这样，构造工具可用这些功能代码为属性事件添加监听者。*/
    ... ...
    private PropertyChangeSupport changes=new PropertyChangeSupport(this);
    public void addPropertyChangeListener(PropertyChangeListener l) {
         changes.addPropertyChangeListener(l);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
         changes.removePropertyChangeListener(l);
    }
... ...
}
```

2. 属性编辑器接口

一个JavaBeans可提供PropertyEditor类，为指定的属性创建一个编辑器。这个类必须继承自java.beans.PropertyEditorSupport类。构造工具与手写代码的程序员不直接使用这个类，而是在下一小节的BeanInfo中实例化并调用这个类。例如：

```
public class MoleculeNameEditor extends java.beans.PropertyEditorSupport {
     public String[] getTags() {
          String resule[]={
              "HyaluronicAcid","Benzene","buckmisterfullerine",
              "cyclohexane","ethane","water"};
          return resule;
     }
}
```

上例中是为Tags属性创建了属性编辑器，在构造工具内，从下拉表格中选择 MoleculeName的属性应是”HyaluronicAid”或“water”。

3. BeanInfo接口

每个Bean类也可能有与之相关的 BeanInfo类，在其中描述了这个 Bean在构造工具内出现时的外观。BeanInfo中可定义属性、方法、事件，显示它们的名称，提供简单的帮助说明。

例如：

```
public class MoleculeBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
    try {
            PropertyDescriptor pd=new PropertyDescriptor("moleculeName",Molecule.class);
            /*通过pd引用了上一节的MoleculeNameEditor类,取得并返回
    moleculeName属性*/
            pd.setPropertyEditorClass(MoleculeNameEditor.class);
            PropertyDescriptor result[]={pd};
            return result;
```

```
        } catch(Exception ex) {
                System.err.println("MoleculeBeanInfo: unexpected exeption: "+ex);
                return null;
        }
      }
    }
```

2.3 Java Servlet

鉴于JSP和Java Servlet的紧密联系，笔者认为学习JSP一定要有Java Servlet的基础，当然，不需要成为Java Servlet的专家，但一定要有概念上的认识。

下面将介绍Java Servlet的基础知识，对于具体的程序开发，现在不需要读者立即掌握，本节的目的在于让读者能够对Servlet有所了解。

2.3.1 HTTP Servlet API

Java Servlet 开发工具（JSDK）提供了多个软件包，在编写 Servlet 时需要用到这些软件包。其中包括两个用于所有 Servlet 的基本软件包：javax.servlet 和 javax.servlet.http。可从sun公司的Web站点下载 Java Servlet 开发工具，现在一般使用JSDK2.0。这里主要介绍javax.servlet.http提供的HTTP Servlet应用编程接口。

1. 简述

创建一个 HTTP Servlet，需要扩展 HttpServlet 类，该类是用专门的方法来处理 HTML表格数据的 GenericServlet 的一个子类。HttpServlet 类包含 init()、destroy()、service() 等方法。其中 init() 和 destroy() 方法是继承的。

（1） init() 方法

在 Servlet 的生命期中，仅执行一次 init() 方法。它是在服务器装入 Servlet 时执行的。可以配置服务器，以在启动服务器或客户机首次访问 Servlet 时装入 Servlet。无论有多少客户机访问 Servlet，都不会重复执行 init() 。

（2）service() 方法

service() 方法是 Servlet 的核心。每当一个客户请求一个 HttpServlet 对象时，该对象的 service() 方法就要被调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。 在 HttpServlet 中已存在 service() 方法。缺省的服务功能是调用与 HTTP 请求的方法相应的 do 功能。例如， 如果 HTTP 请求方法为 GET，则缺省情况下就调用 doGet() 。Servlet 应该为 Servlet 支持的 HTTP 方法覆盖 do 功能。因为 HttpServlet.service() 方法会检查请求方法是否调用了适当的处理方法，不必覆盖 service() 方法。只需覆盖相应的 do 方法就可以了。

（3）destroy() 方法

destroy() 方法仅执行一次，即在服务器停止且卸装 Servlet 时执行该方法。典型的，将 Servlet 作为服务器进程的一部分来关闭。缺省的 destroy() 方法通常是符合要求的，但也可以覆盖它，典型的是管理服务器端资源。例如，如果 Servlet 在运行时会累计统计数据，则可以编写

一个 destroy() 方法，该方法用于在未装入 Servlet 时将统计数字保存在文件中。另一个示例是关闭数据库连接。

（4）GetServletConfig（）方法

GetServletConfig（）方法返回一个 ServletConfig 对象，该对象用来返回初始化参数和 ServletContext。ServletContext 接口提供有关 servlet 的环境信息。

（5）GetServletInfo（）方法

GetServletInfo（）方法是一个可选的方法，它提供有关 servlet 的信息，如作者、版本、版权。

当服务器调用sevlet 的Service（）、doGet（）和doPost（）这三个方法时，均需要“请求”和“响应”对象作为参数。“请求”对象提供有关请求的信息，而”响应”对象提供了一个将响应信息返回给浏览器的通信途径。javax.servlet 软件包中的相关类为ServletResponse和ServletRequest，而javax.servlet.http 软件包中的相关类为HttpServletRequest 和 HttpServletResponse。

Servlet 通过这些对象与服务器通信并最终与客户机通信。 Servlet 能通过调用“请求（Request）”对象的方法获知客户机环境、服务器环境的信息和所有由客户机提供的信息。 Servlet 可以调用“响应”对象的方法发送响应，该响应是准备发回客户机的。

2. 常用HTTP Servlet API概览

支持HTTP协议的servlet可以使用javax.servlet.http包进行开发，而javax.servlet包中的核心功能提供了 Web开发的许多类和函数，为进行 JSP的开发带来了极大的方便。比如，抽象 HttpServlet类包含对不同 HTTP请求方法和头信息的支持， HttpServletRequest和 HttpServletResponse接口允许直接与 Web服务器通信，而 HttpSession提供内置会话跟踪功能；Cookie类可以很快地设置和处理 HTTP Cookie，HttpUtils类用于处理请求字串。

(1) Cookie

Cookie类提供了读取、创建和操纵 HTTP Cookie的便捷途径，允许 servlet在客户端存储少量的数据。Cookie主要用于会话跟踪和存储少量用户配置信息数据。

Servlet用HttpServletRequest的getCookie（）方法获取 Cookie信息；HttpServletResponse的 addCookie（）方法向客户端发送新的 Cookie，因为是使用 HTTP 头设置的，所以 addCookie（）必须在任何输出发送到客户端之前调用。

虽然Java Web Server有一个 sun.servlet.util.Cookie类能完成大致相同的工作，但最初的 Servlet API 1.0 却没有 Cookie类。sun.servlet.util.Cookie类和当前 Cookie类唯一不同的是获取和创建方法是 Cookie类的静态组成部分，而不是 HttpServletRequest和HttpServletResponse的接口。

(2) HttpServlet

HttpServlet是开发 HTTP servlet框架的抽象类，，其中的 service()方法将请求分配给 HTTP的 protected service()方法。

(3) HttpServletRequest

HttpServletRequest通过扩展 ServletRequest基类，为 HTTP servlets提供附加的功能。它支持 Cookies和session跟踪及获取 HTTP头信息的功能；HttpServletRequest还能解析 HTTP的表单数据，并将其存为 servlet参数。

服务器将HttpServletRequest对象传给HttpServlet的service()方法。

(4) HttpServletResponse

HttpServletResponse扩展ServletResponse类，允许操纵HTTP协议相关数据，包括响应头和状态码。它定义了一系列常量，用于描述各种 HTTP状态码，还包含用于session跟踪操作的帮助函数。

(5) HttpSession

HttpSession接口提供了对Web访问者的认证机制。 HttpSession接口允许servlet查看和操纵会话相关信息，比如创建访问时间和会话身份识别。它还包含一些方法，用于绑定会话到特定的对象，允许“购物车”和其他的应用程序保存数据用于各连接间共享，而不必存到数据库或其他的extra-servlet资源中。

Servlet调用HttpServletRequest的getSession()方法来获得HttpSession对象，定制SESSION的行为，比如在销毁SESSION之前等待的时间，依服务器而定。

虽然任何对象都可以绑定到 SESSION，然而对一些事务繁忙的 Servlet，绑定大的对象到SESSION中将会加重服务器的负担。减轻服务器负担最常用的解决办法是，仅仅绑定用于实现java.io.Serializable接口的对象（它包含Java API核心中的所有数据类型对象）。有些服务器能将Serializable对象写入磁盘中，Unserializable对象比如java.sql.Connection,必须保留在内存中。

(6) HttpSessionBindingEvent

HttpSessionBindingListener监听对象绑定或断开绑定于会话时， HttpSessionBindingEvent被传递到HttpSessionBindingListener。

(7) HttpSessionBindingListener

当对象绑定于 HttpSession或从 HttpSession松开绑定时，通过调用 valueBound ()和valueUnbound ()来通知用于实现HttpSessionBindingListener的接口。其他情况下，这个接口可以顺序清除与session相关的资源，例如数据库连接等。

(8) HttpSessionContext

HttpSessionContext提供了访问服务器上所有活动 session的方法，这对servlet清除不活动的session，显示统计信息和其他共享信息是很有用的。 Servlet通过调用 HttpSession的getSessionContext () 方法获得HttpSessionContext对象。

(9) HttpUtils

这是一个容纳许多有用的基于 HTTP方法的容器对象，使用这些方法，可以使 Servlet开发更方便。

2.3.2 系统信息

要成功建立Web应用，必须了解它所需的运行环境，还要了解执行 servlet的服务器和发送请求的客户端的具体情况。不管应用程序运行于哪种环境，都应该确切知道应用程序所应处理的请求信息。

Servlet有许多方法可以获取这些信息。大多数情况下，每个方法都会返回不同的结果。比较CGI用于传递信息的环境变量，读者会发现 servlet方法具有以下几个优点：

- 强有力的类型检查。
- 延迟计算。
- 与服务器的更多交互。

1. 初始化参数

每个注册的servlet名称都有与之相关的特定参数， servlet程序在任何时候都可获得这些参数；它们经常使用init（）方法来为servlet设定初始或缺省值以在一定程度上定制servlet的行为。

(1) 获得初始参数

servlet用getInitParameter（）方法来获取初始参数：

public String ServletConfig.getInitParameter(String name)

这个方法返回初始参数的名称或空值 (如果不存在)。返回值总是String类型，由servlet对它进行解释。

GenericSevlet类实现ServletConfig接口，直接访问getInitParameter（）方法。

(2) 获取初始参数名

servlet调用getInitParameterNames（）方法可检验它的所有参数：

public Enumeration ServletConfig.getInitParameterNames()

这个方法返回值是字符串对象的枚举类型或空值 (如果没参数)，经常用于程序的调试。

GenericServlet类也可以使得servlets直接访问这个方法。

2. 服务器

servlet能了解服务器的大多数信息。它能知道主机名称，端口号，服务器软件及其他信息。Servlet还能将这些信息显示给客户端，以定制客户机基于特定服务器数据包的行为，甚至可以明确要运行servlet的机器的行为。

(1) 服务器相关信息

servlet通过四个方法得到服务器的信息：两个由传递给 servlet的ServletRequest对象调用，两个从ServletContext对象调用，ServletContext包含servlet的运行环境。通过使用方法getServerName（）和getServerPort（），Servlet能为具体请求分别获取服务器的名称和端口号：

```
public String ServletRequest.getServerName()
public int ServletRequest.getServerPort()
```

ServletContext的GetServerInfo（）和getAttribute（）方法提供服务器软件和属性的信息：

```
public String ServletContext.getServerInfo()
public Object ServletContext.getAttribute(String name)
```

(2) 锁定servlet到服务器

利用服务器信息可以完成许多特殊的功能，例如：写了一个servlet，而且不想让它随处运行，或许你想出售，限制那些未授权的拷贝，或者想通过一个软件证书锁定 servlet到客户机上。另一种情况可能是，开发人员为servlet编写了一个证书并且想确保它运行于防火墙后。这都不难做到，因为servlet能及时地访问到服务器的相关信息。

3 客户端

对每个请求，servlet能获取客户机、要求认证的页面及实际用户的有关信息。这些信息可用

于登录访问，收集用户个人资料或限制某些客户端的访问。

(1) 获取客户机信息

servlet可用getRemoteAddr()和getRemoteHost()分别获取客户机的IP地址和主机名称：

```
public String ServletRequest.getRemoteAddr()
public String ServletRequest.getRemoteHost()
```

以上两种方法都返回字符串对象类型。这些信息来自于连接服务器和客户端的socket端口，所以远程地址和远程主机名有可能是代理服务器的地址和主机名称。远程地址可能是“192.26.80.118”，而远程主机名可能是“dist.engr.sgi.com”。

InetAddress.getByName()方法可以将IP地址和远程主机名称转化为java.net.InetAddress对象：

```
InetAddress remoteInetAddress = InetAddress.getByName(req.getRemoteAddr());
```

(2) 限制为只允许某些地区的机器访问

由于美国政府对好的加密技术出口的限制政策，在某些 Web站点上，下载允许的软件就得特别谨慎。利用servlet的获取客户机信息的功能，可以很好地加强这种限制。这些 servlet能检查客户机，并对那些来自美国和加拿大的机器提供下载链接。

对于商业应用，可以确定让一个Servlet只对来自企业内部网的客户机服务，从而保证安全。

(3) 获取用户相关信息

如果要对Web pages作更进一步的限制，而不是根据地域限制访问，该怎么做呢？比如，发布的在线杂志，只想让已定购的用户可以访问。读者也许会说，这好办，我早都能做，不一定非得用servlet。

是的，几乎每种HTTP服务器都内嵌了这种功能，可以限制特定用户访问所有或部分网页。怎样设定限制依你所使用的服务器不同而有差异，这里我们给出它们的共同工作机理。浏览器第一次试图访问某个页面时，服务器会给浏览器一个要求身份验证的响应，浏览器收到这个响应后，会弹出一个要求输入用户名和密码的对话框。

用户输入信息后，浏览器会试图再一次访问该页，但这次请求信息中附加了用户名称和密码信息。服务器接受用户名/密码对后，就会处理此请求；如果相反，服务器不接受此用户名/密码对，浏览器的访问再一次被拒绝，用户必须重新输入。

那么，servlet是怎样处理的呢？当访问受限制的 servlet时，servlet可以调用getRemoteUser()方法，获取服务器认可的用户名称：

```
public String HttpServletRequest.getRemoteUser()
```

Servlet也可以用getAuthType（）方法获取认证类型：

```
public String HttpServletRequest.getAuthType()
```

这个方法返回所用的认证类型或可空值 (如果未加限制)，最常使用的认证类型是“ BASIC ”和“ DIGEST ”。

(4) 个性化的欢迎信息

一个简单的调用getRemoteUser()方法的servlet，可以通过称呼用户名称向他问好，并记住他上次登录的时间。但是需要注意的是，这种方法仅仅适用于授权用户。对于未授权用户，可以使用Session。

4. 请求

前面讲述了servlet如何获取服务器和客户机的相关信息，现在要学习真正重要的内容：servlet如何知道客户请求什么。

(1) 请求参数

每个对servlet的访问都可以有许多与之相关的参数，这些参数都是典型的名称/值对，用于告诉servlet在处理请求时所需的额外信息。千万注意别把这里的参数与前面提到的与servlet自身相关的参数搞混。

幸运的是，即使servlet要获取许多参数，获取每个参数的方法也都一样，即getParameter()和getParameterValues()方法：

```
public String ServletRequest.getParameter(String name)
public String[] ServletRequest.getParameterValues(String name)
```

getParameter()以字符串形式返回命名的参数，如果没指定参数则返回空值，返回值必须保证是正常的编码形式。如果此参数有多个值，那么这个值是与服务器相关的，这种情况下应该用getParameterValues()方法，这个方法以字符对象数组的形式返回相应参数的所有值，如果没指定，当然为空值。返回的每个值在数组中占一个单位长度。

除了能获取参数值之外，servlet还能用getParameterNames()获取参数名称:

```
public Enumeration ServletRequest.getParameterNames()
```

这个方法以字符串枚举类型返回参数名称，或者在没有参数时返回空值。这个方法经常用于程序的调试。

最后，servlet还能用getQueryString()方法获取请求的二进制字串：

```
public String ServletRequest.getQueryString()
```

这个方法返回请求的二进制字串(已编码的GET参数信息)，如果没有请求字串，则返回空值。这些底层数据很少用来处理表单数据。

(2) 发布许可证密钥

如果现在准备编写一个给特定主机和端口号发布KeyedServerLock许可证密钥的servlet，从servlet获取的密钥可用于解锁KeyedServerLock的servlet。那么，怎么知道servlet所要解锁的主机名和端口号呢？当然是请求参数。

(3) 路径信息

除参数信息外，HTTP请求还能包括“附加路径信息”或“虚拟路径”等。通常，附加路径信息是用于指明servlet要用到的文件在服务器上的路径，一般用HTTP请求的URL形式表示，可能像这样：

http：//server：port/servlet/ViewFile/index.html

这将激活ViewFile servlet，同时传递“index.html”作为附加路径信息。Servlet可以访问这个路径信息，还能将字串“index.html”转化为文件index.html的真实路径。什么是“/index.html”的真实路径呢？它是指当客户直接请求“/index.html”文件时，服务器返回的完整文件系统路径。可能是document_root/index.html，当然服务器也可能用别名将它改变了。

除了用明确的URL形式指定外，附加信息也可写成HTML表单中ACTION参数的形式：

```
<FORM METHOD=GET
      ACTION="/servlet/Dictionary/dict/definitions.txt">
word to look up: <INPUT TYPE=TEXT NAME="word"><P>
<INPUT TYPE=SUBMIT><P>
</FORM>
```

表单激活Dictionary servlet处理请求任务，同时传递附加路径信息“ dict/definitions.txt ”。servlet会用单词definitions查找definitions.txt文件，如果客户请求“ /dict/definitions.txt ”文件，同时在server_root/publlic_html/dict/definitions.txt也存在，客户将会看到相同的文件。

1) 获取路径信息。

servlet可用getPathInfo()方法获取附加路径信息：

```
public String HttpServletRequest.getPathInfo()
```

这个方法返回与请求相关的附加路径信息，或者在没给定时，返回空值。 Servlet通常需要知道给定文件的真实文件系统路径，这样就有了 getPathTranslated()方法：

```
public String HttpServletRequest.getPathTranslated()
```

这个方法返回已经转化为真实文件路径的附加路径信息，或者在没有附加路径信息时返回空值。返回的路径未必要指向已存在的文件和目录，已转化的路径可能是：“ C：\JavaWebServer1.1.1 \public_html\dict\definitions.txt ”。

2) 特别的路径转换。

有时，servlet需要在附加路径信息中没有的路径，就得用 getRealPath()方法来完成此项任务：

```
public String ServletRequest.getRealPath(String path)
```

这个方法返回任何给定“ 虚拟路径 ”的真实路径，或者返回空值。如果给定路径是“ / ”，这个方法返回服务器文档的根目录；如果给定路径是getPathInfo()，返回的路径与getPathTranslated()方法的返回值相同。Generic servlets和HTTP servlets都可使用这个方法，CGI中没有与之相关的函数。

3) 获取MIME类型。

servlet知道文件路径后，还往往需要知道文件类型，可使用getMimeType()方法来做这项工作：

```
public String ServletContext.getMimeType(String file)
```

这个方法返回指定文件的MIME类型，或者在不知道的情况下返回空值。有时，在文件不存在时也返回“ text/plain ”。常见的文件类型有：“ text/html ”，“ text/plain ”，“ image/gif ”和“ image/jpeg ”。

下面这个语句代码可获取附加路径信息的 MIME类型：

```
String type = getServletContext().getMimeType(req.getPathTranslated())
```

(4) 服务文件

许多应用服务器，例如 WebLogi，利用servlet来处理每个请求，这不仅是 servlet处理能力上的优势，而且，这为服务器的模块化设计带来了极大的便利。比如，所有的文件都由 com.sun.server.http.FileServlet servlet提供服务，此servlet在file下注册，并负责处理“ / ”别名（是请求的缺省文件）。

(5) 决定被请求的内容

servlet能用几种方法获取客户请求的确切文件。毕竟，最根部的 servlet总假定为直接请求的目标，在一个很长的servlet链中，每个servlet只能有一个连接。

没有方法用于直接返回客户用于请求的原始 URL，javax.servlet.http.HttpUtils类的getRequestURL()方法能完成类似的工作：

```
public static StringBuffer HttpUtils.getRequestURL(HttpServletRequest req)
```

这个方法基于HttpServletRequest对象的变量信息，重构请求的 URL，它返回StringBuffer类型，包含构架（像 HTTP）、服务器名、端口号和额外路径信息。重构后的 URL 应该与客户请求的URL非常相像，它们之间的差别非常细微（比如空格形式在客户端用 %20表示，而在服务器端是"+"）。由于这个方法返回 StringBuffer类型，所以URL能被有效地修改（比如，附加些查询参数）。这个方法经常用于创建重定向消息和报告错误。

大多数情况下，servlet并不真正需要请求的URL，而需要的是URI，它是由方法getRequestURI()返回的:

```
public String HttpServletRequest.getRequestURI()
```

这个方法返回统一资源标示符（ URI），对正常的HTTP servlet来说，一个URI可以看成是URL减去构架、主机名、端口号和请求字串，但包含一些额外路径信息。

(6) 请求机理

除了知道请求的内容外， servlet还有一种方法用于获取如何请求的信息。 GetScheme()方法用于返回请求的构架:

```
public String ServletRequest.getScheme()
```

例如“ http ”， “ https ”,和“ ftp ”，还有Java特有的“ jdbc ”和“ rmi ”。虽然CGI也有包含构架的变量SERVER_URL，但没有与getScheme()相应的函数。对HTTP servlet来说，这个方法表明请求是通过使用SSL安全连接（用“ htttps ”表示），还是非安全连接（用“ http ”表示）。

getProtocol()方法返回用于请求的协议和版本号 :

```
public String ServletRequest.getProtocol()
```

协议和版本号用斜线隔开。如果不能决定协议类型，则返回空值。对 HTTP servlet来说，协议通常是“ vHTTP/1.0v ”或“ vHTTP/1.1 ”, HTTP servlet能用协议版本决定客户端是否可使用HTTP 1.1的新特性。

要获取请求所用的方法， servlet调用getMethod()：

```
public String HttpServletRequest.getMethod()
```

这个函数返回用于请求的 HTTP方法，包括“ GET ”,“ POST ”和“ HEAD ”，HttpServlet的实现函数service()用这个方法分派请求任务。

(7) 请求头

HTTP请求和响应有许多与 HTTP 头相关的内容。这些头提供了一些与请求（或响应）有关的额外信息。 HTTP 1.0协议提供了十几种头， HTTP 1.1则更多。对头的完整讨论超出了本书的范围，这里描述servlet最常访问的头。

servlet在执行请求时很少需要读 HTTP 头，许多与请求相关的头信息都由服务器处理了。这

里举一个服务器如何限制对某些文档的访问的例子，服务器使用 HTTP头，而servlet并不了解其中细节。当服务器接到访问受限页面的请求时，它会检查带有适当认证信息头的请求，这个头应该包含合法的用户名和密码，如果没有，服务器发出包含 WWW-Authenticate的头，告诉浏览器对资源的访问被拒绝，如果客户发送的请求包含正确的认证头，服务器会授权访问并允许激活的servlet通过getRemoteUser()方法获取用户名称。

1) 访问头值。

HTTP头值可以通过 HttpServletRequest对象访问。使用 getHeader()、getDataHeader()或getIntHeader()方法，它们分别返回String类型、long 类型和int类型数据：

```
public  String HttpServletRequest.getHeader(String name)
public  long HttpServletRequest.getDateHeader(String name)
public int HttpServletRequest.getIntHeader(String name)
```

getHeader()方法以String类型返回指定头的值，如果头未作为请求的组成部分，则返回空值，所有类型的头都可以用这种方法获取。

getDateHeader()返回long类型的值，表示日期，如果头未作为请求的组成部分，则返回- 1。当被调用的头值不能转化为 Date时，此方法会抛出一个 IllegalArgumentException。这个方法在处理last-Modifed和If-Modified-Since的头时非常有用。

GetIntHeader()返回头的整型值或- 1（如果未作为请求的组成部分被发送），当被调用的头不能转化为整型时，这个方法抛出 NumberFormatException异常。

如果能使用 getHeaderNames()访问，servlet还能获得所有的头名称：

```
public Enumeration HttpServletRequest.getHeaderNames()
```

这个方法以String对象的枚举类型返回所有头的名称。如果没有头，则返回空的枚举类型。

2) servlet链中的头信息。

servlet链加了一个有趣的处理 servlet头信息的环，不像其他的 servlets，处于链中或链末的servlet不是从客户请求中读取头信息值，而是从前一个 servlet的响应信息中读取头信息值。

这种处理方法的强有力和灵活性来自于这样一个事实： servlet能智能地处理前一个servlet的输出，不仅在内容方面，而且在头信息值方面。比如，它能向响应信息中加些额外的头信息，或改变已有的头信息。它甚至还能禁止头信息。

但是这种强大的功能是要负责任的：除非 servlet明确地读取前一个servlet的响应头信息，并作为它自己的响应信息的一部分加以发送，否则此头信息将不被发送，客户端就看不到此信息。正常的链中servlet总是会传递前一个servlet的头，除非有特殊原因而需要处理别的什么。

(8) 输入流

每个由servlet处理的请求都有一个与之相关的输入流，就像 servlet将相关的response对象写到PrintWriter或OutputStream中一样，servlet也能从Reader或InputStream中读取与请求相关的信息。从输入流中读取的数据可以为任何数据类型和任意长度，输入流有三个作用：

1) 在servlet链中，把前一个servlet的响应信息向下传递；

2) 将与POST请求相关的内容传递给 HTTP servlet；

3) 把客户端发来的二进制信息传递给 non-HTTP servlet。

要读取输入流中的字符数据，应该使用 getReader()方法，并返回BufferdReader的类对象：

```
public BufferdReader ServletRequest.getReader() throws IOException
```

使用BufferedReader作为返回的数据类型的优点是，它能在各种字符集间正确地转换。如果getInputStream()在同样的请求之前被调用，这个方法会抛出 IllegalStateException异常，如果输入字符不被支持或是未知字符，则抛出 UnsupportedEncodingException异常。

要从输入流中读取二进制数据，就得使用 getInputStream()方法，并返回ServletInputStream类型：

```
public ServletInputStream ServletRequest.getInputStream()  throws  IOException
```

ServletInputStream是InputStream的直接子类，可以当作正常的 InputStream来处理，具有有效的一次一行地读取数据的能力。如果在同一个请求之前调用 getReader()，这个方法会抛出IllegalStateException异常。一旦有了ServletInputStream，就可以用readLine()进行行读取：

```
public int ServletInputStream.readLine(byte b[], int off, int len)
   throws IOException
```

这个方法从输入流中将 bytes读入字节数组 b中，开始处由 off给出，遇到‘ \n ’时或读够len字节时结束读取。结束符“ \n ”也读入缓存。这个方法返回已读取的字节数，或到达输入流的末尾时返回-1。

Servlet使用getContentType()和getContentLength()分别获取经过输入流发送的内容类型和数据长度：

```
public String ServletRequest.getContentType()
public int ServletRequest.getContentLength()
```

getContentType()方法返回经过输入流的发送内容类型，或返回空值（如果类型不明，比如没有数据）；getContentLength()返回按字节计算的长度，如果类型不明则返回- 1。

1) 用输入流构建servlet链。

链中的servlet从前一个servlet中通过输入流获得响应信息。

2) 输入流处理POST请求。

当servlet处理POST请求时，使用输入流来访问 POST数据的情况是极少见的。典型地，POST数据只不过是参数信息， servlet可以很方便地用getParameter()方法获取它。 Servlet可以检查输入流的类型来识别POST请求的类型，如果是application/x-www-form-urlencoded类型，数据可由getParameter()方法或相似的方法获取。

Servlet应在调用getParameter()方法之前调用getContentLength()，以免被拒绝访问。恶意客户可能在发送POST请求时，发送不合理的巨大的数据，企图在 servlet调用getParameter()方法时，把服务器的速度降到最慢。 Servlet可以用getContentLength()来验证数据长度的合理性，或限定小于4k作为预防措施。

3) 用输入流接收文件。

servlet可以使用输入流来接收上传的文件，在讲解之前，需提醒很重要的一点是：上传文件是试验性的，并且不是所有的浏览器都支持。 Netscape Navigator 从3.0，微软从Internet Exploer 4.0才开始支持文件上传。

简单地说，任何数量的文件和参数都可以在单个 POST请求中作为表单数据被传送， POST请求格式与标准 application/x-www-form-urlencoded的表单格式不同，所以有必要将类型设为multipart/form-data。

文件上载的客户端程序的编写相当简单，下面这个 HTML将发送一个表单，询问用户名称和要上载的文件。注意ENCTYPE属性和FILE输入类型的使用。

```
<FORM ACTION = "/servlet/UploadTest" ENCTYPE="multipart/form-data"
METHOD=POST>
What is your name? <INPUT TYPE=TEXT NAME=submitter> <BR>
Which file do you want to upload? <INPUT TYPE=FILE NAME=file> <BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

具体如何实现文件上传，请见本书第 11章。

(9) 额外属性

有时servlet需要了解请求的其他信息，并且这些信息用前面提到的方法无法获取，这时就得使出最后一招，即 getAttribute（）方法。还记得 ServletContent有个getAttribute（）方法是如何返回特定服务器的属性的吗？ ServletRequest也有一个getAttribute（）方法：

```
public object ServletRequest.getAttribute(String name)
```

这个方法返回指定请求的服务器属性，如果服务器不支持指定请求的属性则返回空值。这个方法允许服务器为servlet提供有关请求的定制信息。例如， Java Web Server有三个可获得的属性：javax.net.ssl.ciphersuite、java.net.ssl.peer_certificates和javax.net.ssl.session。运行在Java Web Server上的servlet可以窥探客户 SSL连接的这些属性。

2.3.3 传送HTML信息

本节首先讲解从servlet如何返回一个标准的 HTML response，然后讲解如何建立客户端的持久连接以降低返回 response的开销。最后还要探讨一下在处理 HTML和HTTP时的一些额外的东西，包括用支持类来对象化 HTML输出，返回错误和其他状态码，发送定制的头信息，重定向请求，使用客户牵引，客户掉线检测和向服务器日志中写数据等。

1. response的结构

HTTP servlet能为客户返回三种信息：一个状态码、任意数量的 HTTP头和应答信息。状态码是个整数，就像你想像的那样，它描述了应答状态。状态码能表明成功和失败，或告诉客户机采取下一步动作完成请求过程。数字状态码通常伴有“ 原因词 ”，以人们容易看懂的方式来描述状态。通常状态码在后台工作并由浏览器软件解释。有时，尤其是当出错时，浏览器向用户显示状态码。最常见的状态码可能要数“ 404 Not Found ”，当服务器不能定位URL时，它就会发出这个状态码。

响应体是响应信息的主要内容，对 HTML页来说，响应体就是HTML本身。对图片来说，响应体是构成图片的字节。响应体可以是任何类型和任意长度；客户端通过解释响应信息中的HTTP头知道该接收什么。

普通的servlet比HTTP servlet简单——它只向用户返回应响体。然而，对 GenericServlet子类

来说，用API将一个响应体分成许多精细的部分是可能的，好像返回多条目的感觉。事实上，这就是HTTP servlet要做的工作。在底层，服务器将响应以字节流的形式发送给客户端，任何设置状态码或头的方法都是在这个基础上的抽象。

明白这点很重要，因为即使 servlet程序员不必了解 HTTP协议的细节，协议确实会影像servlet调用方法的顺序。特别是， HTTP协议规定状态码和头必须在响应体之前发送。因此，servlet要注意发送任何响应体之前总是先设置状态码和头。有些服务器，包括 Java Web Server，内部缓存了一些servlet响应体（通常大约是4K）——这允许在即使servlet写了一个较短的响应体之后，可以有一定自由度地设置状态码和头。然而，这些行为都是服务器相关的，一名明智的servlet编程人员，应该忘掉这一切。

2. 发送标准的响应信息

ServletResponse的setContentType()方法可以将响应的内容设置为指定的 MIME类型。

```
public void ServletResponse.setContentType(String type)
```

在HTTP servlet中，这个方法用于设置Content-Type HTTP头。

GetWriter()方法返回一个PrintWriter对象，用于写基于字符的响应数据：

```
public PrintWriter ServletResponse.getWriter()  throws IOException
```

这个writer根据内容类型中给定的字符集进行字符编码，如果没指定字符集（这是常事），writer将用ISO-8859-1进行编码，ISO-88598-1主要是用于西欧文字。字符集在后面的章节中会有所提及，所以现在你只要记住在得到 PrintWriter之前，总是先设置内容的类型。如果 getOutputStream()已被这个响应调用，这个方法将抛出 IllegalStateException异常；如果输出流的编码形式不被支持或不明，则抛出 UnsupportedEncodingException异常。

除了用PrintWriter返回响应外，servlet还能用java.io.OutputStream的特殊子类来写二进制数据，这就是ServletOutputStream类，它在java.servlet中定义。可以用 getOutputStream()方法得到一个ServletOutputStream对象：

```
public ServletOutputStream ServletResponse.getOutputStream()  throws  IOException
```

这个方法返回一个ServletOutputStream类对象，用于写二进制（一次一个字节）响应数据。不进行任何编码。如果已为这个响应调用了getWriter()，这个方法将返回一个IllegalStateException异常。

ServletOutputStream很像标准的Java PrintStream类，在Servlet API v1.0中，这个类用于所有的servlet的输出，既可是文本类型，又可是二进制类型。在 Servlet API v2.0中，它仅用于处理二进制的输出。由于是 OutputStream的直接子类，它拥有 OutputStream的write()、flush()和close()方法。为解决这个问题，它加了自己的 print()、println()方法，用于写大多数的 Java原始类型的数据。 ServletOutputStream接口与PrintStream接口的不同是： ServletOutputStream的print（）和println（）方法不能明确地直接显示 Object或char[]类型的数据。

3 使用持续连接

持续连接（有时又称作＂ keep-alive ＂连接）能优化 servlet向客户返回内容的方式。要明白它的优化机理，必须首先弄懂 HTTP连接是如何工作的。下面将对这里面的基本概念做一些浅显的解释。

当客户，比如浏览器，想从服务器请求一个 Web文档时，首先会与服务器建立一个 socket的

连接。通过这个连接，客户端可以发出请求和接收服务器响应。客户端发送空行表示完成请求，反过来，服务器通过关闭socket连接表明响应已完成。

就这么简单。但是如果收到的网页中含有 <IMG>标签或<APPLET>标签，要求客户机从服务器接收更多的内容，该怎么办呢？那么又得另起一个 socket，如果一个网页中包含10个图片和一个由25个类组成的applet，就需要36个连接来传送此页（当然，现在可以使用 JAR文件打包这些类而减少连接数目）。难怪有人戏称WWW为Word Wide Wait！。

一个较好的方法是同一个socket连接接收更多的网页内容，有时也把这种技术称作持续连接。持续连接的关键是客户端与服务器必须在服务器的响应结束处与客户端开始下一个连接的地方达成一致。它们可能用一个空行作为记号，但如果响应信息本身包含一个空行又怎么办呢？持续连接的工作方式是，服务器在响应体中设置 Content-Length头，告诉客户端这个响应体有多大。客户端就知道接收完这个响应体之后，才控制下一个 socket连接。

大多数服务器都能处理它所服务文件的 Content-Length头，但对不同的servlet，处理方式不一样。Servlet可以使用setContentLength()方法来处理动态内容的持续连接：

```
public void ServletResponse.setContentLength(int len)
```

这个方法返回服务器响应内容的长度，在 HTTP servlet中，这个方法设定 HTTP Content-Length头。注意这个方法的使用是可选的，然而，如果使用它， servlet将会充分利用持续连接的优点，客户端也能在下载时精确显示过程控制。

调用setContentLength（）方法时，有两点要注意： servlet必须在发送响应信息之前调用此方法，给定的长度必须精确。哪怕只有一个字节的误差，都可能出问题。这听起来似乎很难做到，实际上，servlet用ByteArrayOutputStream类型来缓存输出。

servlet不是将响应信息写到由 getWriter()返回的 PrintWriter，而是写到建立在ByteArrayOutputStream基础上的PrintWriter中，这个数组会随servlet的输出而增长。当servlet准备退出时，它能设置内容的长度为缓存尺寸，并将内容发送到客户端缓存。注意，字节是基于ServletOutputStream字节对象发送的。做这么点简单的修改，servlet就可能利用持续连接的优点。

持续连接是要付出代价的，这一点很重要。缓存所有的输出和成批发送数据需要更多的内存，还可能延迟客户端开始接收数据的时间点。对响应较短的 servlet来说，可以接受；但对响应较长的servlet，考虑内存开销和延迟的代价，可能还不如建立几个连接。

还要注意一点，并不是所有的 servlet都支持持续连接。就是说，设定 servlet响应的内容长度是好的选择，这个长度信息被支持持续连接的服务器使用，而被其他的服务器忽略。

4. 生成HTML

HTML生成包为servlet提供了一系列的类，这些类抽象了 HTML的许多细节，尤其是HTML的标签。抽象的程度取决于所使用的包：有些只有极少的 HTML标签，留下一些基本的细节（比如打开和关闭HTML标签）给编程人员。利用这类包类似于手工写 HTML，在这里就不讨论了。另一类包是很好地抽象了 HTML的具体内容，同时把HTML作为Java对象的集合。一个网页可以看作是一个对象，它可以包含其他 HTML对象（比如列表和表格），还能包含更多的HTML对象（比如列表项和表格单元）。这种面向对象的方法可以极大地简化 HTML的生成工作，并且使得servlet更容易编写、维护，有时则更高效。

生成Hello World

下例是一个普通的Hello World Servlet，**作为一个最简单的**Servlet，**相信读者通过前面的学习可以理解它。**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**Hello World! Servlet Class */
public class HelloWorld extends HttpServlet {

    /**doGet方法的重载，用于处理客户端的GET请求*/
     public void doGet(HttpServletRequest request, HttpServletResponse response)
     throws IOException, ServletException {
          response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         out.println("<html>");
         out.println("<body>");
         out.println("<head>");
         out.println("<title>Hello World!</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<h1>Hello World!</h1>");
         out.println("</body>");
         out.println("</html>");
     }
}
```

5. 状态码

目前我们的例子还没有设置 HTTP**响应的状态码。如果**servlet**没有特别指定状态码，服务器会将其设为缺省值**200 “OK”**状态码。这在成功返回标准响应时非常便利。然而，利用状态码，**servlet**可以对响应信息做更多的工作，比如，它可以重定向请求和报告错误等。**

最常使用的状态码在HttpServletResponse**类中被定义为助记常量（**public final static int**）。表**2-1**列出了其中的很少几个，全部的状态码可以在第** 6**章中查到。**

表2-1 HTTP**状态码**

助记常量	码 值	默认消息	意 义
SC_OK	200	OK	客户请求成功，服务器的响应信息包含请求的数据。这是缺省的状态码
SC_NO_CONTENT	204	No Content	请求成功，但没有新的响应体返回。接收到这个状态码的浏览器应该保留它们当前 r的文档视图。当servlet从表单中收集数据，希望浏览器保留表单，同时避免 "Document contains no data"错误信息时，这个状态码很有用

（续）

助记常量	码 值	默认消息	意 义
SC_MOVED_PERMANENTLY	301	Moved Perma-nently	被访问的资源被永久的到一个新的位置，在将来的请求中应使用新的URL。新的URL由Location头给出，大多数的浏览器会自动访问新的路径
SC_MOVED _TEMPORARILY	302	Moved Tempo- rarily	被访问的资源被暂时移到一个新的位置，在将来的请求中仍使用新的URL。新的URL由Location头给出，大多数的浏览器会自动访问新的路径
SC_ UNAUTHORIZED	401	Unauthoried	请求没有正确的地认证。用于WWW-Authenticate和Authentication连接
SC_NOT_FOUND	404	Not Found	被请求的资源没找到或不可访问
SC_INTERNAL_ SERVER_ERROR	500	Internal Server Error	服务器内部错误，妨碍了请求过程的正常进行
SC_NOT_ IMPLEMENTED	501	Not Implemented	服务器不支持用于完成请求的函数
SC_SERVICE_UNAVAILABLE	503	Service Unavail-able	服务器暂时不可访问，将来可恢复。如果服务器知道何时可被访问，它会提供Retry-After头

设定状态码

servlet可用SetStatus()方法设定响应状态码：

```
public void HttpServletResponse.setStatus(int sc)
public void HttpServletResponse.setStatus(int sc, String sm)
```

这两个方法都可将 HTTP的状态码设为指定的值，这个状态码可为一个数字，或为在HttpServletResponse中定义的SC_XXX码。带一个参数的方法，原因部分被设置为缺省的消息；带两个参数的方法，原因部分可指定不同的消息。记住， setStatus （）方法应在servlet返回任何响应体之前调用。

如果servlet在处理请求时将状态码设置报错，则它调用 sendError （）方法，而不是sendStatus （）方法：

```
public void HttpServletResponse.setError(int sc)
public void HttpServletResponse.setError(int sc, String sm)
```

servlet对sendError （）和sendStatus的处理过程不同。当带两个参数的方法被调用时，状态码的消息参数可被替代的原因取代，或者它被直接用于响应体中，这取决于服务器的执行。

6. HTTP头

Servlet能设定HTTP头，提供与响应相关的额外信息。表 2-2列出了一些servlet最常使用的HTPP头。

表2-2 HTTP响应头

Cache-Control	指定任何缓存系统对文档的操作。最常用的值有：no-cache（表明此文档不需缓存），no-store（表明此文档不必缓存，甚至不要保存在代理服务器中，通常是敏感内容），max-age=seconds（表明文档到过时的时间长度）。这个头从HTTP1.1开始引入
Pragma	HTTP1.0中的等价头是Cache-control，并且只有一个可能的值no-cache
Connection	用于表明服务器是否愿意为客户维持持续连接。如果愿意，它的值设为 keep-alive；否则设为close。大多数的服务器代表它们的servlet处理这个头，当servlet设置Content-Length头时，服务器就自动将Connection的值设为keep-alive
Retry-After	指定服务器可再次处理请求的时间，与 SC_SERVICE_UNAVAILABLE状态码一起使用。它的值要么是代表时间(秒)的整数，要么是代表实际时间的日期子串。
Expires	指定何时文档可能改变，或它的信息将变成非法。它也意味着，在这个时间之前文档不可能变化
Location	指定文档的新路径，通常与 SC_CREATED, SC_MOVED_PREMANENTLY和SC_MOVED_TEMPORARILY等状态码一起使用。它的值必须是完整的 URL（包含"http：//"）
WWW-Authenticate	指定由访问请求URL 的客户要求的认证体系和认证范围。与 SC_UNATHORIZED状态码一起使用
Content-Encoding	指定用于编码响应体的体系。可能的值有： gzip（或x-gzip）、compress（或x-compress）。多编码体系应按应用于数据的使用顺序用逗号隔开

(1) 设定HTTP头

HttpServletResponse类提供了一系列的方法，帮助 servlet设置HTTP响应头信息。可使用setHeader（）方法设置头值：

```
public void HttpServletResponse.setHeader(String name , String value)
```

这个方法将指定头的值设置为String。这个头名称是不变的，对所有的方法都一样。如果头的值已经设定，新值将覆盖前一值。头的所有类型都可通过这个方法来设定。

如果需要为头值定义一个日期戳，可使用setDateHeader（）方法：

```
public void HttpServletResponse.setDateHeader(String name , long date)
```

这个方法把指定名称的头值设定为特定的日期和时间，并接收日期的 long类型值（表示自GMT 1970，1，1，0：0：0起的毫秒数）。如果这个头已经设置过，新值将覆盖旧值。

最后，可以用setIntHeader（）为头指定一个整型值：

```
public void HttpServletResponse.setIntHeader(String name , int value)
```

这个方法将头的值设定为整型，如果已经设置过，则新值覆盖旧值。

containsHeader（）方法提供了一种检查头是否存在的途径：

```
public boolean HttpServletResponse.containsHeader(String name)
```

如果头已被设置，这个方法返回true，否则返回false。

另外，HTML3.2规范中，有另一种设置头值的方法：在 HTML页中使用内嵌的<META HTTP-EQUIV>标签。

```
<META HTTP-EQUIV="name" CONTENT="value">
```

这个标签必须作为HTML页的<HEAD>组成部分发送。这个技术对servlet没什么用，它主要

用于静态文档，它们不必访问自己的头信息。

(2) 重定向请求

servlet使用状态码和头信息的另一个用途是重定向请求。这是通过给客户端发送另一个 URL 指示来实现的。重定向通常用在：当文档移动时（给客户端发送新的路径），负载平衡（一个URL文档可能分布在几个不同的机器上），简单的随机文档（随机地选择目标路径）。

真正的重定向发生在以下两行：

```
res.setStatus(res.SC_MOVED_TEMPORARILY);
res.setHeader("Location", site);
```

第一行设置状态码，表明需要重定向，第二行给出新的位置。为确保它们正常工作，必须在发送任何输出之前调用这些方法。记住， HTTP协议在发送内容体之前发送状态码和头信息。新站点的位置必须以绝对的URL给出（例如，http：//server：port/path/file.html），否则，客户就难以找到。

用sendRedirect（）方法能很方便地将这两行简化为一行：

```
public void HttpServletResponse.sendRedirect(String location)  throws
IOException
```

这个方法重定向响应信息到新的位置，自动地设置状态码和Location的头信息，这两行就变成：

```
res.sendRedirect(site);
```

(3) 客户牵引

客户牵引与重定向类似，主要的区别在于：浏览器一直显示第一页的内容，并且在接收和显示第二页之前等待一个指定的时间间隔。它之所以叫客户牵引，是因为由客户端拖曳下一页的内容。

这有什么用呢？首先，可以在第二页下载之前，由第一页的内容告诉客户请求的内容已经移动；其次，网页可按顺序接收，为制作慢速页面动画提供了可能。

客户牵引信息是使用 Refresh HTTP 头发送给客户端的。这个头值指明在牵引下一页之前显示当前页的秒数，也可选择地包含一个 URL子串，从中下载文件。如果没给定 URL，则使用相同的URL。下面这个语句告诉客户在显示当前内容 3s后，重新下载同一个servlet：

```
setHeader ("Refresh", "3") ;
```

这里还有一个告诉客户 3s后显示Netscape主页的语句：

```
setHeader ("Refresh", "3；URL=http：//home.netscape.com") ;
```

7. 错误处理

servlet有时会出错，没关系， servlet必须得处理错误，有预料中的，也有始料未及的。错误一旦发生，必须关心两件事：

1) 将对服务器的破坏降到最低。

2) 正确的通知客户端。

因为servlet是用Java编写的，所以它对服务器的破坏程度大大降低。服务器可以安全地嵌入servlet（甚至嵌入它的进程中），就像Web浏览器能安全地嵌入下载的 applets一样。这种安全性是建立在Java的安全体系基础上的，包括保护内存使用，异常处理，安全管理机制等。 Java的内

存保护保证servlet不可能偶尔（或故意地）访问服务器的内部系统； Java的异常处理使服务器能捕获由servlet引起的每一个异常，即使servlet偶尔被零除，或调用空对象的方法，服务器仍能够继续工作；Java的安全管理机制为服务器提供了一种将信任的 servlet放于一个沙箱中的途径，限制和防止它们故意破坏的能力。

应该谨慎的是，处于安全管理器沙箱之外运行的 servlet具有造成服务器破坏的能力， servlet可能覆盖服务器的文件空间或甚至调用 System.exit()，应该相信，被信任的servlet一般不会导致服务器破坏，也极少会调用 System.exit()。

正确地向客户描述出现的问题，不仅仅是 Java语言的事，还有许多需要考虑的因素：

1) 怎样告诉客户端？

servlet是给客户端发送普通的状态码错误页，还是发送错误的字面描述，抑或发送错误堆栈的详细内容？

2) 怎样记录问题？

是保存到文件，写到服务器日志中，发送到客户端还是忽略？

3) 怎样恢复？

Servlet是否继续处理下一个请求？或servlet崩溃，这意味着必须重新下载。

这些问题的答案取决于servlet和它的用途。怎样处理错误就由开发人员自己决定，但应确保servlet在被请求时的可靠性和健壮性。接下来看看 servlet错误处理机制的全貌，可以有选择地用它们来实现错误处理策略。

(1) 状态码

servlet最简单的报错方式是用sendError（）方法发送恰当的400系列或500系列状态码。例如，当servlet被请求一个不存在的文件时，它可以返回 SC_NOT_FOUND；如果请求的工作超出了它的能力范围，它可以返回 SC_NOT_IMPLEMENTED。当发生内部异常时，它可以返回 SC_INTERNAL_SERVER_ERROR。

通过使用状态码，servlet为服务器提供了一个给响应信息特殊处理的机会。例如，有些服务器如Java Web Server，可以用与服务器相关的错误信息取代 servlet的响应信息；如果错误应由servlet向客户提供解释，它可以用 setStatus（）设置状态码，同时发出相应的响应体，这个响应体可以是文本格式，图像格式和其他恰当的类型。

在发送响应体之前，servlet应尽量捕获和处理任何类型的错误。你可能还记得（因为我们反复提到），HTTP规定状态码和HTTP头必须在响应体之前发送。一旦发出响应信息，哪怕只发出一个字符，都来不及更改状态码和 HTTP头信息。为避免这种“太迟”的尴尬局面，可以使用ByteArrayOutputStream缓存或HTML生成器包。

(2) 日志

servlet能将它们的行为和错误通过使用 log（）方法写入日志文件中：

```
public void ServletContext.log(String msg)
public void ServletContext.log(Exception e, String msg)
```

单个参数的方法是将给定的消息写入 servlet日志中，这个日志通常是事件日志文件。带两个参数的版本是将给定的消息和异常堆栈信息写入 servlet日志中。这两种方法的输出格式和日志文

件的路径都是服务器相关的。

GenericServlet类也提供了log（）方法：

```
public void GenericServlet.log(String msg)
```

这是ServletContext方法的另一个版本，移入GenericServlet类中是为了使用方便，这个方法允许servlet像下面这样简单调用，写入servlet日志之中：

```
log (msg) ;
```

然而，GenericServlet没提供两个参数的log（）版本，这可能是个疏忽，会在将来的版本中增加。现在，servlet可以通过调用下列函数执行类似的功能：

```
getServletContext().log(e , msg);
```

log()方法提供了跟踪servlet行为的途径，可用于帮助程序调试。它也提供了保存 servlet遇到的任何错的误详尽描述方法，这个描述可以与发送给客户端的一样，也可以更为详尽。

(3) 报告错误

除了能为服务器管理员记录错误和异常外，开发过程中显示问题的详细描述也是很方便的。遗憾的是，异常不能以String形式返回它的堆栈跟踪信息，它只能显示堆栈跟踪信息到PrintStream或PrintWriter中。要以String类型收集堆栈跟踪信息，就必须绕些弯子。必须让异常显示到特殊的PrintWriter中，这个PrintWriter是建立在ByteArrayOutputStream基础上的，它能捕获输出并转化为String类型。com.oreilly.servlet.ServletUtils类有个getStackTraceString()方法可完成这种工作：

```
public static String getStackTraceAsString(Exception  e)  {
  ByteArrayOutputStream bytes = new ByteArrayOutputStream();
  PrintWriter writer = new PrintWriter(bytes, true);
  e.printStackTrace(writer);
  return bytes.toString();
}
```

下面是提供包括IOException堆栈跟踪信息的ViewFile:

```
//Return the file
try {
   ServletUtils.returnFile(file ,out );
}
catch (FileNotFoundException e) {
  log("Could not find file: " + e.getMessage());
  res.sendError(res.SC_NOT_FOUND);
}
catch (IOException e) {
  getServletContext().log(e, "Problem sending file");
  res.sendError(res.SC_INTERNAL_SERVER_ERROR,
             ServletUtils.getStackTraceAsString(e));
}
```

(4) 异常处理

前面曾经提到，抛出的异常如果没被servlet捕获，就将被服务器捕获。服务器如何处理异常是因服务器而异的：它也许给客户端发送消息和堆栈跟踪信息；它或许自动地记录异常；它甚

至也许在servlet上调用destroy（）方法并重新下载。

为某特定的服务器设计和开发的 servlet可以优化服务器的行为，而设计为跨多种服务器的servlet做不到，如果这种servlet需要特殊的异常处理，就得考虑相应的服务器因素。

有些异常类型是 servlet必须捕捉的， servlet仅能将IOException、ServletException或RuntimeException的子类异常传给它的服务器。原因与方法的特性有关， servlet的service（）方法在它的throws语句中声明抛出IOException和ServletException异常，因此它不捕获编译过程中的异常。RuntimeException是个特殊类型的异常，它不需要在 throws语句中被声明，一个常见的例子是NullPointException。

init（）方法和destroy（）方法也有自己的特征， init方法声明仅抛出ServletException异常，而destroy（）方法声明不抛出异常。

ServletException是java.lang.Exception的子类，此类是servlet相关的，并在javax.servlet.package包中被定义。这个异常表明常规servlet问题，它与java.lang.Exception具有相同的构造函数：一个不带参数，一个仅带一个消息字串参数。捕捉这种异常的服务器可能用任何合适的方法处理它。

javax.servlet包定义了一个ServletException子类UnavailableException，这个异常表明servlet不可访问，或是临时的，或是永久的。

UnavailableException有两个构造函数：

```
java.servlet.UnavailableException(Servlet servlet , String  msg)
java.servlet.UnavailableException(int Seconds , Servlet servlet , String  msg)
```

双参数的构造函数创建一个新的异常，表明指定的 servlet永久性地不可访问，并且由 msg给出解释；三参数的构造函数创建一个新的异常，表明指定的 servlet暂时不可访问，并由msg给出解释，不可访问的时间长度由seconds给出，这仅仅是个估计时间。

2.4 SQL语言

JSP的数据库方面所依赖的是JDBC，而JDBC的强大在于：JDBC可以使Java成为一种能同不均匀的数据库环境打交道的强大工具，这种不均匀的数据库环境尽管的确差别很大，但是无论是哪一种关系数据库，从Oracle到DB2到Sybase再到MS SQL Server，有一点是相同的，那就是SQL语言-结构化查询语言。

尽管各个不同的数据库厂商对 SQL做了各自的扩展，如：Oracle的PL-SQL、Microsoft SQL Server的Transact-SQL、还有SQL语言鼻祖IBM的DB2 SQL，每一个RDBMS厂商都宣称自己的扩展是最优秀的，然而，这些不同的SQL仍然有共同点，他们都基于ANSI SQL 92。

SQL不是一门特别复杂的语言，不过如果想要学好 SQL，特别是各个不同厂商特有的 SQL，仍然需要特别的努力，仅仅讲述 SQL中最基本的语句，本书在第一部分的例子程序中也只会用到最基本的SQL语句，在第二部分的例子中由于将会使用存储过程，所以会使用一些扩展的 SQL语言，这些扩展将在需要时再进行讲解。

2.4.1 SQL子类型

SQL语言的子类型包括：数据处理语言（ DML）、数据定义语言（DDL）和数据控制语言

(DCL)。

数据处理语言DML完成在数据库中确定、修改、添加、删除某一数据的值的任务，下面是在Java和JDBC中常用到的一些数据处理SQL语句：

SELECT 在数据库中依据某一种规则查询数据。

INSERT 向数据库中添加一行数据。

DELETE 删除数据库中的某一行数据。

UPDATE 改变数据库中已有记录。

数据定义语言DDL完成定义数据库的结构，包括数据库本身、数据表、目录、视图等数据库元素：

CREATE 在数据库中建立一个元素。

DROP 在数据库中删除一个元素。

数据控制语言DCL完成管理数据库中数据的存储权限的任务，下面是在 Java和JDBC中常用到的一些数据控制SQL语句：

GRANT 设置某一用户或用户组可以某种形式访问数据库中的某一元素。

REVOKE 去掉某一用户或用户组可以某种形式访问数据库中的某一元素的权利。

2.4.2 SQL语言的具体命令和使用

下面的范例使用了 Microsoft SQL Server 7.0自带的样本数据库，需要说明的是，尽管Microsoft SQL Server在Java的支持上比其他的RDBMS如Oracle、DB2要差，但Microsoft SQL Server比较容易使用，本书在第一部分的例子基本上都是使用 Microsoft SQL Server 7.0建立的；另一方面，第一部分的例子没有使用到 SQL Server特有的SQL语句的情况，所以读者可把这些例子移植到Oracle、DB2或是其他任何支持SQL的数据库系统上。

1. SELECT 语句

SELECT 无疑是SQL语句中最常用的语句，一个SELECT语句可以十分简单，也可以十分复杂，下面先从最简单的开始：

在Query Analyzer中选择数据库为Northwind，然后输入：

```
SELECT * FROM  customers
```

执行它，则可以看见如图2-1所示的结果：

这条SELECT语句的含义从字面上就很好理解，即：从 customers数据表中检索出全部数据，“*”表示全部的列。

如果把“*”换为“ CustomerID ”，则结果将会变为：

```
customerID
----------
ALFKI
ANATR
ANTON
.....
```

注意：为了节省篇幅，下面的例子执行结果将不再使用图形表示，读者应当可以看懂。

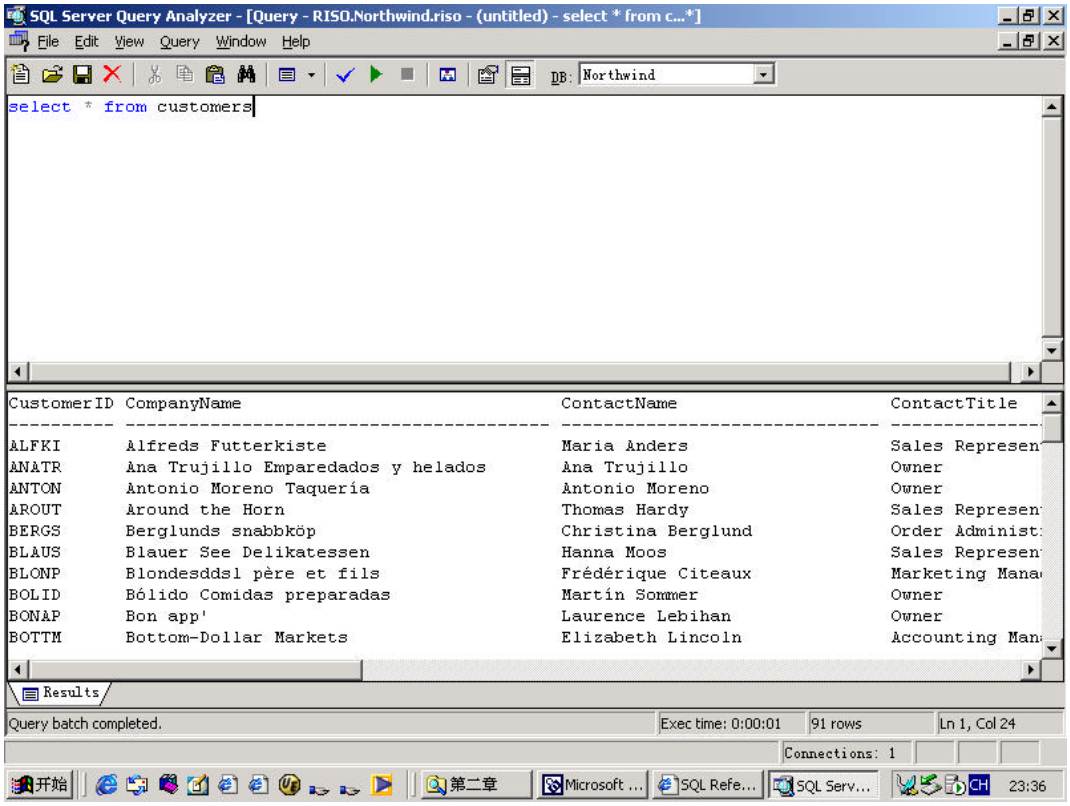

图2-1

“＊”中的内容也可以包含多项，其中用“，”隔开即可，如：

```
"SELECT CustomerID,CompanyName from customers"。
```

(1) **使用别名**

数据表中某一列的名称应该是有意义的，但不幸的是，这仅仅是对某一些人而言，常常有这种情况：某一位数据库建立者创建的数据库中包含的列名对他自己来说是有明确意义的，但对另外一些人来说却是不知所云。

解决办法就是在查询的时候为数据表的某一列建立一个别名，下面举例说明：

```
select phone as "电话", fax as "传真"from customers
```

结果如下：

电话	传真
030-0074321	030-0076545
(5) 555-4729	(5) 555-3745
(5) 555-3932	NULL

.....

这样，原先对于不懂英文的人来说不知所云的“ phone ”和“ fax ”现在变成了简单易懂的

“电话”和“传真”。

(2) 在查询输出中加入文本

尽管上面加上别名之后的输出结果让人容易理解，但仍然不是太明确，在查询输出中加入文本的方法将可以输出完整的句子。

```
select CompanyName as "公司名称","公司的电话是",phone as"电话"from customers
```

结果是：

公司名称		电话
Alfreds Futterkiste	公司的电话是	030-0074321
Ana Trujillo Emparedados y helados	公司的电话是	(5) 555-4729
Antonio Moreno Taquer í a	公司的电话是	(5) 555-3932

.....

在JDBC和JSP中，在查询中加入文本有助于直接输出可以用在网页中且包含 HTML代码的查询结果。

(3) ORDER BY 子句

ORDER BY子句的作用是将输出结果按照某一列按升序或降序排列，其中，升序排列的附加命令是ASC，而降序排列的附加命令是DESC，缺省为升序排列。

```
SELECT CompanyName,phone from customers ORDER BY phone
```

结果如下：

CompanyName	phone
Maison Dewey	(02) 201 24 67
Supr ê mes d é lices	(071) 23 67 22 20
Rancho grande	(1) 123-5555

而如果是：

```
SELECT CompanyName,phone from customers ORDER BY phone DESC
```

则结果是：

CompanyName	phone
Wartian Herkku	981-443655
Bon app'	91.24.45.40
Wilman Kala	90-224 8858

(4) WHERE短语

WHERE是一个有条件的选择数据的短语，它指定只返回那些和 WHERE短语重指定的条件一致的数据。

WHERE短语的条件可以包含关系运算、布尔运算、 LIKE、IN、BETWEEN等等，甚至可以包含其他的SELECT语句的查询结果。下面分别介绍：

1) **关系运算。**SQL**语言的关系运算包括：**“ = ”、“ > ”、“ < ”、“ >= ”、“ <= ”、“ <> ”。**从这些符号本身就应该可以理解其意义，下面是一个实例：**

```
SELECT CompanyName,City from customers where City="London"
```

目的是找出客户中所有所在地为伦敦的公司。结果如下

CompanyName	City
Around the Horn	London
B's Beverages	London
Consolidated Holdings	London
Eastern Connection	London
North/South	London
Seven Seas Imports	London

2) **布尔运算。**SQL**语言的布尔运算包括“** AND **”、“** OR **”、“** NOT **”，即“与”、“或”、“非”三种运算。**

例子如下，目的是找出订单中所有和代号“ VINET **”的公司相关并且由**2**号雇员处理的订单。**

```
SELECT OrderID,CustomerID,EmployeeID FROM orders
      WHERE CustomerID="VINET"AND EmployeeID=2
```

结果如下：

OrderID	CustomerID	EmployeeID
10295	VINET	2
10737	VINET	2

(5) LIKE**运算**

LIKE**运算的用途是在那些文本类型的数据中找出某一特定的字符串，加上通配符的使用，只需学会使用**LIKE**运算就可以构造一个简单的搜索引擎了。**

在LIKE**运算中包含如下两个通配符：**

% **代表多个字符**

_ **代表一个字符**

例子如下：

第一个例子在客户数据表中查找所有名称中含有“ Hungry **”的公司：**

```
SELECT CompanyName,CustomerID FROM customers
     WHERE CompanyName LIKE "%Hungry%"
```

结果如下：

CompanyName	CustomerID
Hungry Coyote Import Store	HUNGC
Hungry Owl All-Night Grocers	HUNGO

第二个例子是在订单数据表中查询所有的订单号以“ 1024 ”开头的，且一共为五位的订单。

```
SELECT OrderID,CustomerID FROM orders WHERE OrderID LIKE "1024_"
```

结果如下：

OrderID	CustomerID
10249	TOMSP
10248	VINET

(6) IN运算

IN运算通过一个预先定义好的值表来限定所用值的范围，当所给参数和表中的值匹配时才认为是“ 真 ”。

例如，在订单数据表中查询所有代号为 VINET和TOMSP的客户：

```
SELECT CustomerID,OrderID,ShipName FROM orders
        WHERE CustomerID IN("VINET","TOMSP")
```

结果如下：

CustomerID	OrderID	ShipName
TOMSP	10249	Toms Spezialit?ten
TOMSP	10438	Toms Spezialit?ten
TOMSP	10446	Toms Spezialit?ten
TOMSP	10548	Toms Spezialit?ten
TOMSP	10608	Toms Spezialit?ten
TOMSP	10967	Toms Spezialit?ten
VINET	10248	Vins et alcools Chevalier
VINET	10274	Vins et alcools Chevalier
VINET	10295	Vins et alcools Chevalier
VINET	10737	Vins et alcools Chevalier
VINET	10739	Vins et alcools Chevalier

(7) BETWEEN运算

和IN运算一样，BETWEEN运算也是限定所用值的范围，当所给参数和预设的值匹配时才认为是“ 真 ”。不过BETWEEN运算所限定的方式不是给出一个值表，而是给出一个最大值和最小值。当数据表中的值在这个最大和最小值之间（包括最大值和最小值）时认为是“ 真 ”。

例如：

要找出订单数据表中所有订单号在 10249和10254之间的订单：

```
SELECT CustomerID,OrderID,ShipName FROM orders
    WHERE OrderID BETWEEN 10249 AND 10254
```

结果如下：

CustomerID	OrderID	ShipName
TOMSP	10249	Toms Spezialit?ten
HANAR	10250	Hanari Carnes
VICTE	10251	Victuailles en stock
SUPRD	10252	Suprêmes délices
HANAR	10253	Hanari Carnes
CHOPS	10254	Chop-suey Chinese

也许有的读者需要得到在最大值和最小值之间，但并不包括最大值和最小值的数据，那么可以这样做：

```
SELECT CustomerID,OrderID,ShipName FROM orders
       WHERE OrderID BETWEEN 10249 AND 10254
       AND NOT OrderID IN (10249,10254)
```

这样结果就变成了：

CustomerID	OrderID	ShipName
HANAR	10250	Hanari Carnes
VICTE	10251	Victuailles en stock
SUPRD	10252	Suprêmes délices
HANAR	10253	Hanari Carnes

(8) **使用函数**

尽管大部分关系数据库系统（RDBMS**）都扩充了可以在**SQL**中使用的函数，许多数据库系统还允许用户自己扩充函数，但下面的几个函数总是可以使用的：**

AVG **返回某一组中的值除以该组中值的个数的和。**

COUNT **返回一组行或值中行或值的个数。**

MAX **返回一组值中的最大值。**

MIN **返回一组值中的最小值。**

下面是实际的例子：

求出所有订单的数量总和：

```
SELECT COUNT(Freight) FROM orders
```

求出所有订单的运费平均值：

```
SELECT AVG(Freight) FROM orders
```

求出所有订单的运费最大值

```
SELECT MAX(Freight) FROM orders
```

求出所有订单的运费最小值

```
SELECT MIN(Freight) FROM orders
```

(9) **子查询**

子查询的概念在于将一个查询的结果作为另一个查询的条件，举例如下：

在订单数据表中的客户公司是使用公司代号来表示的，如果需要查询运费在 500以上的公司的名称和电话，就需要使用子查询这个概念：

```
SELECT CompanyName,phone FROM customers
   WHERE CustomerID IN (
     SELECT CustomerID from orders
        WHERE freight>500
   )
```

结果得到了需要的数据：

CompanyName	phone
Ernst Handel	7675-3425
Great Lakes Food Market	(503) 555-7555
Hungry Owl All-Night Grocers	2967 542
Queen Cozinha	(11) 555-1189
QUICK-Stop	0372-035188
Rattlesnake Canyon Grocery	(505) 555-5939
Save-a-lot Markets	(208) 555-8097
White Clover Markets	(206) 555-4112

2. 使用数据修改命令

SQL语言中数据的修改命令包括：

INSERT 建立记录。

DELETE 删除记录。

UPDATE 修改记录。

(1) INSERT语句

INSERT语句在使用时有两种不同的格式。需要注意的是， INSERT语句假定需要插入数据的数据表已经用CREATE语句或其他工具建立。

第一种用法是不列出数据表的各个列名，而按照数据表建立时的顺序将数据列出：

```
INSERT INTO Customers VALUES
    ("AAAAA","AAAAA Co.Ltd.","riso liao","Owner","Peking University","Bei Jing","Bei
Jing","HaiDian","100871","86-10-62754178","86-10-62763126")
```

第二种用法是在数据表的后面按照后面数据需要插入的列的顺序列出数据表中各个列的名称：

```
INSERT INTO Customers(
     CustomerID,CompanyName,ContactName,ContactTitle,Address,City,Region,PostalCode,
Country,Phone,Fax
     )
     VALUES
     ("AAAAA","AAAAA Co.Ltd.","riso liao","Owner","Peking University","Bei Jing","Bei
Jing","HaiDian","100871","86-10-62754178","86-10-62763126")
```

上面两条语句的作用都一样，不过在实际使用中建议使用第二种方法，因为第二种方法可以让数据和数据要插入的列一一对应，而且还有利于插入空值，例如：现在需要加入到记录中的这个公司数据相当不完整，只有公司名称和代号，当采用第一种方法时，需要这样书写 SQL 语句：

```
INSERT INTO Customers VALUES
    ("AAAAA","AAAAA Co.Ltd.",NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL)
```

而第二种方法只需要书写如下语句即可：

```
INSERT INTO Customers(
   CustomerID,CompanyName)
   VALUES
   ("AAAAA","AAAAA Co.Ltd.")
```

(2) DELETE语句

DELETE语句的使用是相当简单的，具体就是：

```
DELETE FROM 表名 条件
```

其中条件不是必需的，当没有条件时，就意味着删除表中的所有记录。

例如，语句DELETE FROM customers WHERE CustomerID= ”AAAAA ”将删除刚才建立的记录；而语句 DELETE FROM customers将删除customers数据表中的所有记录。

(3) UPDATE语句

UPDATE语句的作用是将数据库中某一条记录的某一个记录域更新，语句格式如下：

```
UPDATE 数据表 SET 列名=新数据 条件
```

和DELETE语句一样，这里的条件也可以是没有的，如果没有条件，那么数据表中的每一条记录都将被更新。

现在来试验一下UPDATE语句：

首先看一看数据本来的样子：

```
SELECT CustomerID,CompanyName FROM customers
```

可以看到第一行记录为：

```
CustomerID          CompanyName
-----------------   -------------------------
ALFKI               Alfreds Futterkiste
```

现在执行一个UPDATE：

```
UPDATE customers SET CompanyName=’AAAAA’
   WHERE CustomerID=’ALFKI’
```

再执行一下SELECT CustomerID,CompanyName FROM customers

发现结果变成了：

```
CustomerID          CompanyName
---------------     ----------------------
ALFKI               AAAAA
```

这样，数据库中的一条语句就被更新了。

2.5 JDBC

本节将在上节讲述的SQL语言的基础上介绍JDBC，JDBC使得在Java程序中可以轻松地操纵数据库：从企业级的Oracle、Sybase、DB2到最简单的Access、MySQL。在JSP中，就是利用JDBC来访问数据库的。

2.5.1 什么是 JDBC

JDBC 是一种用于执行 SQL 语句的 Java API，它由一组用 Java 编程语言编写的类和接口组成。JDBC 为工具/数据库开发人员提供了一个标准的 API，使他们能够用纯 Java API 来编写数据库应用程序。

有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。换言之，有了 JDBC API，就不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写一个程序，为访问 Informix 数据库又写另一个程序，等等。只需用 JDBC API 写一个程序就够了，它可向相应的数据库发送 SQL 语句。而且，使用 Java 编程语言编写的应用程序，无须去忧虑要为不同的平台编写不同的应用程序。将 Java 和 JDBC 结合起来将使程序员只需写一遍程序就可让它在任何平台上运行。

Java 具有坚固、安全、易于使用、易于理解和可从网络上自动下载等特性，是编写数据库应用程序的杰出语言。所需要的只是 Java 应用程序与各种不同数据库之间进行对话的方法。而 JDBC 正是作为此种用途的机制。

JDBC 扩展了 Java 的功能。例如，用 Java 和 JDBC API 可以发布含有 applet 的网页，而该 applet 使用的信息可能来自远程数据库。企业也可以用 JDBC 通过 Intranet 将所有职员连到一个或多个内部数据库中（即使这些职员所用的计算机有 Windows、 Macintosh 和 UNIX 等各种不同的操作系统）。随着越来越多的程序员开始使用 Java 编程语言，对从 Java 中便捷地访问数据库的要求也在日益增加。

MIS 管理员们都喜欢 Java 和 JDBC 的结合，因为它使信息传播变得容易和经济。企业可继续使用它们安装好的数据库，并能便捷地存取信息，即使这些信息是存储在不同数据库管理系统上。新程序的开发期很短。安装和版本控制将大为简化。程序员可只编写一遍应用程序或只更新一次，然后将它放到服务器上，随后任何人就都可得到最新版本的应用程序。对于商务上的销售信息服务， Java 和 JDBC 可为外部客户提供获取信息的更新更好方法。

1. JDBC 的用途

简单地说，JDBC 可做三件事：

- 与数据库建立连接。
- 发送 SQL 语句。
- 处理结果。

下列代码段给出了以上三步的基本示例：

```
Connection con = DriverManager.getConnection ("jdbc:odbc:wombat", "login",
```

```
"password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
      int x = rs.getInt("a");
      String s = rs.getString("b");
      float f = rs.getFloat("c");
}
```

2. JDBC 是一种低级 API，是高级 API 的基础

JDBC 是个”低级”接口，就是说，它用于直接调用 SQL 命令。在这方面它的功能极佳，并比其他的数据库连接 API更易于使用，但它同时也被设计为一种基础接口，在它之上可以建立高级接口和工具。高级接口是”对用户友好的”接口，它使用的是一种更易理解和更为方便的 API，这种 API 在幕后被转换为诸如 JDBC 这样的低级接口。在编写本文时，正在开发两种基于 JDBC 的高级 API：

一种是用于 Java 的嵌入式 SQL。至少已经有一个提供者计划编写它。DBMS 实现 SQL：一种专门设计来与数据库联合使用的语言。JDBC 要求 SQL 语句必须作为 String 传给 Java 方法。相反，嵌入式 SQL 预处理器允许程序员将 SQL 语句直接与 Java 混在一起使用。例如，可在 SQL 语句中使用 Java 变量，用以接受或提供 SQL 值。然后，嵌入式 SQL 预处理器将通过 JDBC 调用把这种 Java/SQL 的混合物转换为 Java。

另一种关系数据库表到 Java 类的直接映射。JavaSoft 和其他提供者都声称要实现该 API。在这种”对象/关系”映射中，表中的每行对应于类的一个实例，而每列的值对应于该实例的一个属性。于是，程序员可直接对 Java 对象进行操作；存取数据所需的 SQL 调用将在”掩盖下”自动生成。此外还可提供更复杂的映射，例如将多个表中的行结合进一个 Java 类中。

随着人们对 JDBC 的兴趣日益浓厚，越来越多的开发人员一直在使用基于 JDBC 的工具，以使程序的编写更加容易。程序员也一直在编写力图使最终用户对数据库的访问变得更为简单的应用程序。例如，应用程序可提供一个选择数据库任务的菜单。任务被选定后，应用程序将给出提示及空白供填写执行选定任务所需的信息。所需信息输入后，应用程序将自动调用所需的 SQL 命令。在这样一种程序的协助下，即使用户根本不懂 SQL 的语法，也可以执行数据库任务。

3. JDBC 与 ODBC 和其他 API 的比较

目前，ODBC（开放式数据库连接）API 可能是使用最广的、用于访问关系数

据库的编程接口。它能在几乎所有平台上连接几乎所有的数据库。为什么 Java 不使用 ODBC？

对这个问题的回答是：Java 可以使用 ODBC，但最好是在 JDBC 的帮助下以 JDBC-ODBC 桥的形式使用，这一点稍后再讲解。现在的问题已变成：”为什么需要 JDBC”？回答如下：

ODBC 不适合直接在 Java 中使用，因为它使用 C 语言接口。从 Java 调用本地 C 代码在安全性、实现性、坚固性和程序的自动移植性方面都有许多缺点。

从 ODBC C API 到 Java API 的字面翻译是不可取的。例如，Java 没有指针，而 ODBC 却对指针用得很广泛（包括很容易出错的指针 “void *”）。你可以将 JDBC 想像成被转换为面向对

象接口的 ODBC，而面向对象的接口对 Java 程序员来说较易于接收。

ODBC 很难学。它把简单和高级功能混在一起，而且即使对于简单的查询，其选项也极为复杂。相反，JDBC 尽量保证简单功能的简便性，而同时在必要时允许使用高级功能。 启用”纯 Java “ 机制需要像 JDBC 这样的 Java API。如果使用 ODBC，就必须手工地将 ODBC 驱动程序管理器和驱动程序安装在每台客户机上。如果完全用 Java 编写 JDBC 驱动程序，则 JDBC 代码在所有 Java 平台上（从网络计算机到大型机）都可以自动安装、移植并保证安全性。

总之，JDBC API 对于基本的 SQL 抽象和概念是一种自然的 Java 接口。它建立在 ODBC 上，而不是从零开始。因此，熟悉 ODBC 的程序员将发现 JDBC 很容易使用。JDBC 保留了 ODBC 的基本设计特征；事实上，两种接口都基于 X/Open SQL CLI（调用级接口）。它们之间最大的区别在于：JDBC 以 Java 风格与优点为基础并进行优化，因此更加易于使用。

最近，Microsoft 又引进了 ODBC 之外的新 API： RDO、 ADO 和 OLE DB。这些设计在许多方面与 JDBC 是相同的，即它们都是面向对象的数据库接口且基于可在 ODBC 上实现的类。但在这些接口中，未看见有特别的功能使我们要转而选择它们来替代 ODBC，尤其是在 ODBC 驱动程序已建立起较为完善的市场的情况下。它们最多也就是在 ODBC 上加了一种装饰而已。这并不是说 JDBC 不需要从其最初的版本再发展了；然而，我们觉得大部份的新功能应归入诸如前一节中所述的对象/关系映射和嵌入式 SQL 这样的高级 API。

4. 两层模型和三层模型

JDBC API 既支持数据库访问的两层模型，同时也支持三层模型。

在两层模型中，Java applet 或应用程序将直接与数据库进行对话。这将需要一个 JDBC 驱动程序来与所访问的特定数据库管理系统进行通信。用户的 SQL 语句被送往数据库中，而其结果将被送回给用户。数据库可以位于另一台计算机上，用户通过网络连接到上面。这就叫做客户机/服务器配置，其中用户的计算机为客户机，提供数据库的计算机为服务器。网络可以是 Intranet（它可将公司职员连接起来），也可以是 Internet。

在三层模型中，命令先是被发送到服务的”中间层“，然后由它将 SQL 语句发送给数据库。数据库对 SQL 语句进行处理并将结果送回到中间层，中间层再将结果送回给用户。 MIS 管理员们都发现三层模型很吸引人，因为可用中间层来控制对公司数据的访问和可作的更新的种类。中间层的另一个好处是，用户可以利用易于使用的高级 API，而中间层将把它转换为相应的低级调用。而且，许多情况下三层结构可提供一些性能上的好处。

到目前为止，中间层通常都用 C 或 C++ 这类语言来编写，这些语言执行速度较快。然而，随着最优化编译器（它把 Java 字节代码转换为高效的特定于机器的代码）的引入，用 Java 来实现中间层将变得越来越实际。这将是一个很大的进步，它使人们可以充分利用 Java 的诸多优点（如坚固、多线程和安全等特征）。JDBC 对于从 Java 的中间层来访问数据库非常重要。

5. SQL 的一致性

结构化查询语言 (SQL) 是访问关系数据库的标准语言。其困难之处在于：虽然大多数的 DBMS（数据库管理系统）对其基本功能都使用了标准形式的 SQL，但它们却不符合最近为更高级的功能定义的标准 SQL 语法或语义。例如，并非所有的数据库都支持存储程序或外部连接，那些支持这一功能的数据库又相互不一致。人们希望 SQL 中真正标准的那部份能够进行扩展以

包括越来越多的功能。但同时 JDBC API 又必须支持现有的 SQL。

JDBC API 解决这个问题的一种方法是允许将任何查询字符串一直传到所涉及的 DBMS 驱动程序上。这意味着应用程序可以使用任意多的 SQL 功能，但它必须冒这样的风险：有可能在某些 DBMS 上出错。事实上，应用程序查询甚至不一定是 SQL，或者说它可以是为特定的 DBMS 设计的 SQL 的专用派生物（例如，文档或图像查询）。

JDBC 处理 SQL 一致性问题的第二种方法是提供 ODBC 风格的转义子句。

转义语法为几个常见的 SQL 分歧提供了一种标准的 JDBC 语法。例如，对日期文字和已存储过程的调用都有转义语法。

对于复杂的应用程序，JDBC 用第三种方法来处理 SQL 的一致性问题。它利用 Database MetaData 接口来提供关于 DBMS 的描述性信息，从而使应用程序能适应每个 DBMS 的要求和功能。

由于 JDBC API 将用做开发高级数据库访问工具和 API 的基础，因此它还必须注意其所有上层建筑的一致性。“符合 JDBC 标准”代表用户可依赖的 JDBC 功能的标准级别。要使用这一说明，驱动程序至少必须支持 ANSI SQL-2 Entry Level（ANSI SQL-2 代表美国国家标准局 1992 年所采用的标准。Entry Level 代表 SQL 功能的特定清单）。驱动程序开发人员可用 JDBC API 所带的测试工具包来确定他们的驱动程序是否符合这些标准。

“符合 JDBC 标准”表示提供者的 JDBC 实现已经通过了 JavaSoft 提供的一致性测试。这些一致性测试将检查 JDBC API 中定义的所有类和方法是否都存在，并尽可能地检查程序是否具有 SQL Entry Level 功能。当然，这些测试并不完全，而且 JavaSoft 目前也无意对各提供者的实现进行标级。但这种一致性定义的确可对 JDBC 实现提供一定的可信度。随着越来越多的数据库提供者、连接提供者、Internet 提供者和应用程序编程员对 JDBC API 的接受，JDBC 也正迅速成为 Java 数据库访问的标准。

2.5.2 JDBC 产品

在编写本文时，已经有许多可用的 JDBC产品问世。当然，本节中的信息将很快成为过时信息。因此，有关最新的信息，请查阅 JDBC 的网站，可通过从以下 URL 开始浏览找到：

http://java.sun.com/products/jdbc

1. JavaSoft 框架

JavaSoft 提供三种 JDBC 产品组件，它们是 Java 开发工具包 (JDK) 的组成部分：

JDBC 驱动程序管理器、JDBC 驱动程序测试工具包和 JDBC-ODBC 桥。

JDBC 驱动程序管理器是 JDBC 体系结构的支柱。它实际上很小，也很简单；其主要作用是把 Java 应用程序连接到正确的 JDBC 驱动程序上，然后退出。

JDBC 驱动程序测试工具包为使 JDBC 驱动程序运行你的程序提供一定的可信度。只有通过 JDBC 驱动程序测试包的驱动程序才被认为是符合 JDBC 标准TM 的。

JDBC-ODBC 桥使 ODBC 驱动程序可被用作 JDBC 驱动程序。它的实现为 JDBC 的快速发展提供了一条途径，其长远目标是提供一种访问某些不常见的 DBMS（如果对这些不常见的 DBMS 未实现 JDBC）的方法。

2. JDBC 驱动程序的类型

目前所知的 JDBC 驱动程序可分为以下四个种类：

1) JDBC-ODBC 桥加 ODBC 驱动程序：JavaSoft 桥产品利用 ODBC 驱动程序提供 JDBC 访问。注意，必须将 ODBC 二进制代码（许多情况下还包括数据库客户机代码）加载到使用该驱动程序的每个客户机上。因此，这种类型的驱动程序最适合于企业网（这种网络上客户机的安装不是主要问题），或者是用 Java 编写的三层结构的应用程序服务器代码。

2) 本地 API - 部分用 Java 来编写的驱动程序： 这种类型的驱动程序把客户机 API 上的 JDBC 调用转换为 Oracle、 Sybase、Informix、DB2 或其他 DBMS 的调用。注意，像桥驱动程序一样，这种类型的驱动程序要求将某些二进制代码加载到每台客户机上。

3) JDBC 网络纯 Java 驱动程序：这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，之后这种协议又被某个服务器转换为一种 DBMS 协议。这种网络服务器中间件能够将它的纯 Java 客户机连接到多种不同的数据库上。所用的具体协议取决于提供者。通常，这是最为灵活的 JDBC 驱动程序。有可能所有这种解决方案的提供者都提供适合于 Intranet 用的产品。为了使这些产品也支持 Internet 访问，它们必须处理 Web 所提出的安全性、通过防火墙的访问等方面的额外要求。几家提供者正将 JDBC 驱动程序加到他们现有的数据库中间件产品中。

4) 本地协议纯 Java 驱动程序：这种类型的驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。这将允许从客户机机器上直接调用 DBMS 服务器，是 Intranet 访问的一个很实用的解决方法。由于许多这样的协议都是专用的，因此数据库提供者自己将是主要来源，有几家提供者已经开发出了这样的驱动程序。

最后，我们预计第 3、4 类驱动程序将成为从 JDBC 访问数据库的首选方法。第 1、2 类驱动程序在直接的纯 Java 驱动程序还没有上市前将会作为过渡方案来使用。对第 1、2 类驱动程序可能会有一些变种（下表中未列出），这些变种要求有连接器，但通常这些是更加不可取的解决方案。第 3、4 类驱动程序提供了 Java 的所有优点，包括自动安装（例如，通过使用 JDBC 驱动程序的 applet applet来下载该驱动程序）。

表2-3显示了这 4 种类型的驱动程序及其属性：

表2-3

驱动程序种类	纯 Java	网络协议
JDBC-OCBC 桥	非	直接
基于本地 API 的	非	直接
JDBC 网络的	是	要求连接器
基于本地协议的	是	直接

3. JDBC 驱动程序的获取

在编写本文时，已经有许多JDBC驱动程序存在，如果使用的是Weblogic或是Websphere，产品本身就带有不少数据库系统的驱动程序。许多重要的商业数据库系统也自带了适合自己的JDBC驱动程序，如：Oracle、Sybase、IBM DB2。要获取关于驱动程序的最新信息，请查阅 JDBC 的网站，其网址为： http:// java.sun.com/products/jdbc。

2.5.3 连接概述

Connection 对象代表与数据库的连接。连接过程包括所执行的 SQL 语句和在该连接上所返回的结果。一个应用程序可与单个数据库有一个或多个连接，或者可与许多数据库有连接。

1. 打开连接

与数据库建立连接的标准方法是调用 DriverManager.getConnection ()方法。该方法接受含有某个 URL 的字符串。DriverManager 类（即所谓的 JDBC 管理层）将尝试找到可与那个 URL 所代表的数据库进行连接的驱动程序。DriverManager 类存有已注册的 Driver 类的清单。当调用方法 getConnection ()时，它将检查清单中的每个驱动程序，直到找到可与 URL 中指定的数据库进行连接的驱动程序为止。Driver 的方法 connect 使用这个 URL 来建立实际的连接。

用户可绕过 JDBC 管理层直接调用 Driver 方法。这在以下的特殊情况下将很有用：当两个驱动器可同时连接到数据库中，而用户需要明确地选用其中特定的驱动器时。但一般情况下，让 DriverManager 类处理打开连接将更为简单。

下述代码显示如何打开一个与位于 URL “jdbc:odbc:wombat” 的数据库的连接。所用的用户标识符为 “oboy”，口令为 “12Java”：

```
String url = "jdbc:odbc:wombat";
Connection con = DriverManager.getConnection(url, "oboy", "12Java");
```

2. 一般用法的 URL

由于 URL 常引起混淆，所以先对一般 URL 作简单说明，然后再讨论 JDBC URL。

URL（统一资源定位符）提供在 Internet 上定位资源所需的信息。可将它想像为一个地址。

URL 的第一部份指定了访问信息所用的协议，后面总是跟着冒号。常用的协议有 “ftp”（代表”文件传输协议”）和 “http” （代表”超文本传输协议”）。如果协议是 “file”，表示资源是在某个本地文件系统上而非在 Internet 上（下例用于表示我们所描述的部分；它并非 URL 的组成部分）。

ftp://javasoft.com/docs/JDK-1_apidocs.zip

http://java.sun.com/products/jdk/CurrentRelease

file:/home/haroldw/docs/books/tutorial/summary.html

URL 的其余部份（冒号后面的）给出了数据资源所处位置的有关信息。如果协议是 file，则 URL 的其余部份是文件的路径。对于 ftp 和 http 协议，URL 的其余部份标识了主机并可选地给出某个更详尽的地址路径。例如，以下是 JavaSoft 主页的 URL。该 URL 只标识了主机：

http://java.sun.com

从该主页开始浏览，就可以进到许多其他的网页中，其中之一就是 JDBC 主页。JDBC 主页的 URL 更为具体，它看起来类似：

http://java.sun.com/products/jdbc

3. JDBC URL

JDBC URL 提供了一种标识数据库的方法，可以使相应的驱动程序能识别该数据库并与之建立连接。实际上，驱动程序编程员将决定用什么 JDBC URL 来标识特定的驱动程序。用户不必关心如何来形成 JDBC URL；他们只需使用与所用的驱动程序一起提供的 URL 即可。JDBC

的作用是提供某些约定，驱动程序编程员在构造他们的 JDBC URL 时应该遵循这些约定。

由于 JDBC URL 要与各种不同的驱动程序一起使用，因此这些约定应非常灵活。

首先，它们应允许不同的驱动程序使用不同的方案来命名数据库。例如， odbc 子协议允许（但并不是要求）URL 含有属性值。

第二，JDBC URL 应允许驱动程序编程员将一切所需的信息编入其中。这样就可以让要与给定数据库对话的 applet 打开数据库连接，而无需要求用户去做任何系统管理工作。

第三， JDBC URL 应允许某种程度的间接性。也就是说，JDBC URL 可指向逻辑主机或数据库名，而这种逻辑主机或数据库名将由网络命名系统动态地转换为实际的名称。这可以使系统管理员不必将特定主机声明为 JDBC 名称的一部分。网络命名服务（例如 DNS、NIS 和 DCE ）有多种,而对于使用哪种命名服务并无限制。

JDBC URL 的标准语法如下所示。它由三部分组成，各部分间用冒号分隔：

```
jdbc:< 子协议 >:< 子名称 >
```

JDBC URL 的三个部分可分解如下：

jdbc — 协议。JDBC URL 中的协议总是 jdbc。

〈子协议〉 — 驱动程序名或数据库连接机制（这种机制可由一个或多个驱动程序支持）的名称。子协议名的典型示例是 “ odbc ”，该名称是为用于指定 ODBC 风格的数据资源名称的 URL 专门保留的。例如，为了通过 JDBC-ODBC 桥来访问某个数据库，可以如下所示 URL：

```
URL： jdbc:odbc:fred
```

本例中，子协议为 “ odbc ”，子名称 “ fred ” 是本地ODBC 数据资源。

如果要用网络命名服务（这样 JDBC URL 中的数据库名称不必是实际名称），则命名服务可以作为子协议。例如，可用如下所示的 URL ：

```
jdbc:dcenaming:accounts-payable
```

本例中，该 URL 指定了本地 DCE 命名服务应该将数据库名称 “ accounts-payable ” 解析为更为具体的可用于连接真实数据库的名称。

〈子名称〉 — 一种标识数据库的方法。子名称可以依不同的子协议而变化。它还可以有子名称的子名称（含有驱动程序编程员所选的任何内部语法）。使用子名称的目的是为定位数据库提供足够的信息。前例中，因为 ODBC 将提供其余部份的信息，因此用 "fred" 就已足够。然而，位于远程服务器上的数据库需要更多的信息。例如，如果数据库是通过 Internet 来访问的，则在 JDBC URL 中应将网络地址作为子名称的一部份包括进去，且必须遵循如下所示的标准 URL 命名约定：

```
//主机名:端口/子协议
```

假设 dbnet 是个用于将某个主机连接到 Internet 上的协议，则 JDBC URL 类似：

```
jdbc:dbnet://"wombat:356/fred"
```

4. odbc 子协议

子协议 odbc 是一种特殊情况。它是为用于指定 ODBC 风格的数据资源名称的 URL 而保留的，并具有下列特性：允许在子名称（数据资源名称）后面指定任意多个属性值。 odbc 子协议

的完整语法为：

```
jdbc:odbc:< 数据资源名称 >[;< 属性名 >=< 属性值 >]*
```

因此，以下都是合法的 jdbc:odbc 名称：

```
jdbc:odbc:qeor7
jdbc:odbc:wombat
jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:qeora;UID=kgh;PWD=fooey
```

5. 注册子协议

驱动程序编程员可保留某个名称以将之用作 JDBC URL 的子协议名。当 DriverManager 类将此名称加到已注册的驱动程序清单中时，为之保留该名称的驱动程序应能识别该名称并与它所标识的数据库建立连接。例如，odbc 是为 JDBC- ODBC 桥而保留的。假设有个 Miracle 公司，它可能会将 “ miracle ” 注册为连接到其 Miracle DBMS 上的 JDBC驱动程序的子协议，从而使其他人都无法使用这个名称。

JavaSoft 目前作为非正式代理负责注册 JDBC 子协议名称。要注册某个子协议名称，请发送电子邮件到下述地址：

```
jdbc@wombat.eng.sun.com
```

6. 发送 SQL 语句

连接一旦建立，就可用来向它所涉及的数据库传送 SQL 语句。JDBC 对可被发送的 SQL 语句类型不加任何限制。这就提供了很大的灵活性，即允许使用特定的数据库语句甚至于非 SQL 语句。然而，它要求用户自己负责确保所涉及的数据库可以处理所发送的 SQL 语句，否则将自食其果。例如，如果某个应用程序试图向不支持存储程序的 DBMS 发送存储程序调用，就会失败并将抛出异常。JDBC 要求驱动程序应至少能提供 ANSI SQL-2 Entry Level 功能才可算是 “ 符合 JDBC 标准 ” 的。这意味着用户至少可信赖这一标准级别的功能。

JDBC 提供了三个类，用于向数据库发送 SQL 语句。Connection 接口中的三个方法可用于创建这些类的实例。下面列出这些类及其创建方法：

Statement 由方法 createStatement 所创建。Statement 对象用于发送简单的 SQL 语句。

PreparedStatement 由方法 prepareStatement 所创建。PreparedStatement 对象用于发送带有一个或多个输入参数（ IN 参数）的 SQL 语句。PreparedStatement 拥有一组方法，用于设置 IN 参数的值。执行语句时，这些 IN 参数将被送到数据库中。 PreparedStatement 的实例扩展了 Statement ，因此它们都包括了 Statement 的方法。PreparedStatement 对象有可能比 Statement 对象的效率更高，因为它已被预编译过并存放在那里以供将来使用。

CallableStatement 由方法 prepareCall 所创建。CallableStatement 对象用于执行 SQL 存储程序 — 一组可通过名称来调用（就像函数的调用那样）的 SQL 语句。CallableStatement 对象从 PreparedStatement 中继承了用于处理 IN 参数的方法，而且还增加了用于处理 OUT 参数和 INOUT 参数的方法。

以下所提供的方法可以快速决定应用哪个 Connection 方法来创建不同类型的 SQL 语句。

createStatement 方法用于。

简单的 SQL 语句（不带参数）。

prepareStatement 方法用于：

带一个或多个 IN 参数的 SQL 语句。

经常被执行的简单 SQL 语句。

prepareCall 方法用于：

调用已存储过程。

7. 事务

事务由一个或多个这样的语句组成：这些语句已被执行、完成并被提交或还原。当调用方法 commit 或 rollback 时，当前事务即告结束，另一个事务随即开始。

缺省情况下，新连接将处于自动提交模式。也就是说，当执行完语句后，将自动对那个语句调用 commit 方法。这种情况下，由于每个语句都是被单独提交的，因此一个事务只由一个语句组成。如果禁用自动提交模式，事务将要等到 commit 或 rollback 方法被显式调用时才结束，因此它将包括上一次调用 commit 或 rollback 方法以来所有执行过的语句。对于第二种情况，事务中的所有语句将作为组来提交或还原。

方法 commit 使 SQL 语句对数据库所做的任何更改成为永久性的，它还将释放事务持有的全部锁。而方法 rollback 将丢弃那些更改。

有时用户在另一个更改生效前不想让此更改生效。这可通过禁用自动提交并将两个更新组合在一个事务中来达到。如果两个更新都是成功的，则调用 commit 方法，从而使两个更新结果成为永久性的；如果其中之一或两个更新都失败了，则调用 rollback 方法，以将值恢复为进行更新之前的值。

大多数 JDBC 驱动程序都支持事务。事实上，符合 JDBC 的驱动程序必须支持事务。DatabaseMetaData 给出的信息描述 DBMS 所提供的事务支持水平。

8. 事务隔离级别

如果 DBMS 支持事务处理，它必须有某种途径来管理两个事务同时对一个数据库进行操作时可能发生的冲突。用户可指定事务隔离级别，以指明 DBMS 应该花多大精力来解决潜在冲突。例如，当事务更改了某个值而第二个事务却在该更改被提交或还原前读取该值时怎么办？ 假设第一个事务被还原后，第二个事务所读取的更改值将是无效的，那么是否可允许这种冲突？ JDBC 用户可用以下代码来指示 DBMS 允许在值被提交前读取该值（”dirty 读取”），其中 con 是当前连接：

```
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

事务隔离级别越高，为避免冲突所花的精力也就越多。 Connection 接口定义了五级，其中最低级别指定了根本就不支持事务，而最高级别则指定当事务在对某个数据库进行操作时，任何其他事务不得对那个事务正在读取的数据进行任何更改。通常，隔离级别越高，应用程序执行的速度也就越慢（用于锁定的资源耗费增加了，而用户间的并发操作减少了）。在决定采用什么隔离级别时，开发人员必须在性能需求和数据一致性需求之间进行权衡。当然，实际所能支持的级别取决于所涉及的 DBMS 的功能。

当创建 Connection 对象时，其事务隔离级别取决于驱动程序，但通常是所涉及的数据库的

缺省值。用户可通过调用 setIsolationLevel 方法来更改事务隔离级别。新的级别将在该连接过程的剩余时间内生效。要想只改变一个事务的事务隔离级别，必须在该事务开始之前进行设置，并在该事务结束后进行复位。不提倡在事务的中途对事务隔离级别进行更改，因为这将立即触发 commit 方法的调用，使在此之前所作的任何更改成为永久性的。

2.5.4 DriverManager概述

DriverManager 类是 JDBC 的管理层，作用于用户和驱动程序之间。它跟踪可用的驱动程序，并在数据库和相应驱动程序之间建立连接。另外， DriverManager 类也处理诸如驱动程序登录时间限制及登录和跟踪消息的显示等事务。

对于简单的应用程序，一般程序员需要在此类中直接使用的唯一方法是 DriverManager.getConnection。正如名称所示，该方法将建立与数据库的连接。JDBC 允许用户调用 DriverManager 的方法 getDriver、getDrivers 和 registerDriver 及 Driver 的方法 connect。但多数情况下，让 DriverManager 类管理建立连接的细节为上策。

1. 跟踪可用驱动程序

DriverManager 类包含一列 Driver 类，它们已通过调用方法 DriverManager.registerDriver 对自己进行了注册。所有 Driver 类都必须包含有一个静态部分。它创建该类的实例，然后在加载该实例时对 DriverManager 类进行注册。这样，用户正常情况下将不会直接调用 DriverManager.registerDriver；而是在加载驱动程序时由驱动程序自动调用。加载 Driver 类，然后自动在 DriverManager 中注册的方式有两种：

1) 通过调用方法 Class.forName。这将显式地加载驱动程序类。由于这与外部设置无关，因此推荐使用这种加载驱动程序的方法。以下代码加载类 acme.db.Driver：

```
Class.forName("acme.db.Driver");
```

如果将 acme.db.Driver 编写为加载时创建实例，并调用以该实例为参数的 DriverManager.registerDriver（本该如此），则它在 DriverManager 的驱动程序列表中，并可用于创建连接。

2) 通过将驱动程序添加到 java.lang.System 的属性 jdbc.drivers 中。这是一个由 DriverManager 类加载的驱动程序类名的列表，由冒号分隔。初始化 DriverManager 类时，它搜索系统属性 jdbc.drivers，如果用户已输入了一个或多个驱动程序，则 DriverManager 类将试图加载它们。以下代码说明程序员如何在 ~/.hotjava/properties 中输入三个驱动程序类（启动时，HotJava 将把它加载到系统属性列表中）：

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.test.ourDriver;
```

对 DriverManager 方法的第一次调用将自动加载这些驱动程序类。

注意：加载驱动程序的第二种方法需要持久的预设环境。如果对这一点不能保证，则调用方法 Class.forName 显式地加载每个驱动程序就显得更为安全。这也是引入特定驱动程序的方法，因为一旦 DriverManager 类被初始化，它将不再检查 jdbc.drivers 属性列表。

在以上两种情况中，新加载的 Driver 类都要通过调用 DriverManager.registerDriver 类进行自我注册。如上所述，加载类时将自动执行这一过程。

由于安全方面的原因， JDBC 管理层将跟踪哪个类加载器提供哪个驱动程序。这样，当

DriverManager 类打开连接时，它仅使用本地文件系统或与发出连接请求的代码相同的类加载器提供的驱动程序。

2. 建立连接

加载 Driver 类并在 DriverManager 类中注册后，它们即可用来与数据库建立连接。当调用 DriverManager.getConnection 方法发出连接请求时，DriverManager 将检查每个驱动程序，查看它是否可以建立连接。

有时可能有多个 JDBC 驱动程序可以与给定的 URL 连接。例如，与给定远程数据库连接时，可以使用 JDBC-ODBC 桥驱动程序、JDBC 到通用网络协议驱动程序或数据库厂商提供的驱动程序。在这种情况下，测试驱动程序的顺序至关重要，因为 DriverManager 将使用它所找到的第一个可以成功连接到给定 URL 的驱动程序。

首先 DriverManager 试图按注册的顺序使用每个驱动程序（jdbc.drivers 中列出的驱动程序总是先注册）。它将跳过代码不可信任的驱动程序，除非加载它们的源与试图打开连接的代码的源相同。

它通过轮流在每个驱动程序上调用方法 Driver.connect，并向它们传递用户开始传递给方法 DriverManager.getConnection 的 URL 来对驱动程序进行测试，然后连接第一个认出该 URL 的驱动程序。

这种方法初看起来效率不高，但由于不可能同时加载数十个驱动程序，因此每次连接实际只需几个过程调用和字符串比较。

以下代码是通常情况下用驱动程序（例如 JDBC-ODBC 桥驱动程序）建立连接所需所有步骤的示例：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //加载驱动程序
String url = "jdbc:odbc:fred";
DriverManager.getConnection(url, "userID", "passwd");
```

2.5.5 一个简单的例子

下面是一个简单的例子，在这个例子中，将利用 JDK自带的JDBC-ODBC桥驱动程序查询一个Microsoft SQL Server 7.0 自带的例子数据库，并将得到的结果在屏幕上显示出来。

1. 建立ODBC数据源

在Windows系统的控制面版中，选择“数据源（ODBC）”，如果使用Windows 2000，那么将在“管理工具”中选择。在系统DSN中，选择“添加”，如图2-2所示。

然后，建立一个名为Northwind的数据源，并且设定数据源为需要使用的 SQL Server，这里假设为本地SQL Server数据源，如果读者的数据源不在本地，请自行修改，如图 2-3所示。

然后，在接下来几步中设定缺省数据库为Northwind，然后点击“完成”，建立ODBC数据源，如图2-4所示。

2. 程序代码

在建立数据源以后，就可以开始程序设计工作了，下面的程序建立在一个 Java Application 基础上，使用AWT组件来显示数据库查询的结果，具体的程序代码比较简单，读者应该能看懂。

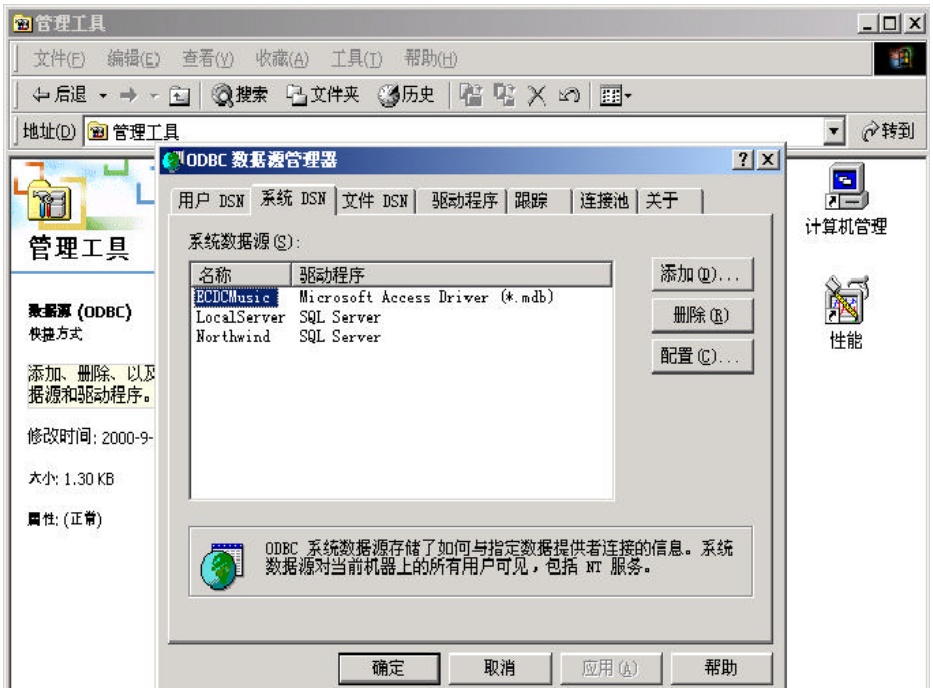

图2-2

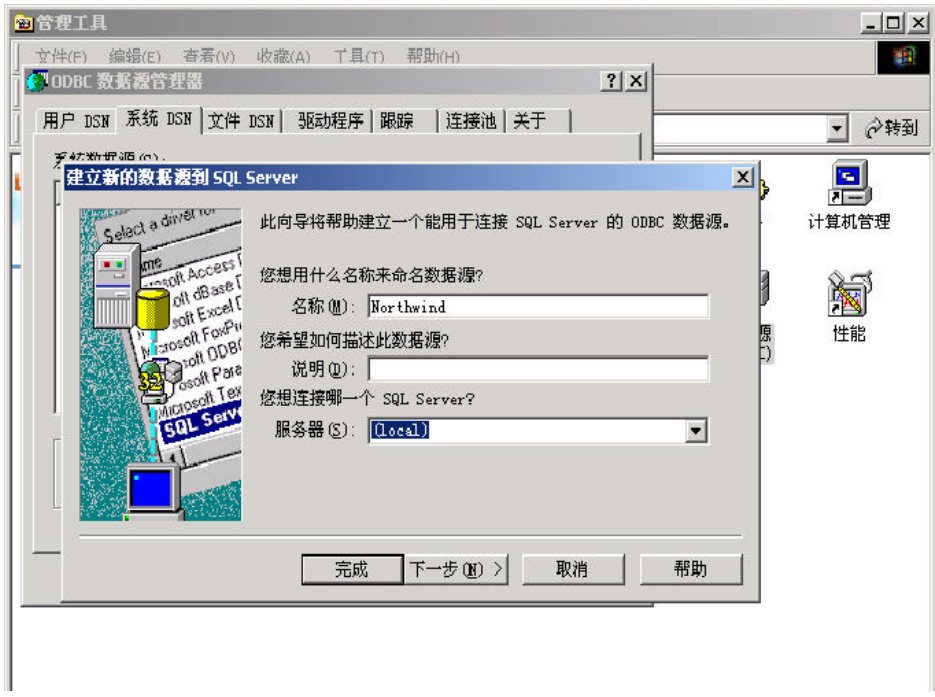

图2-3

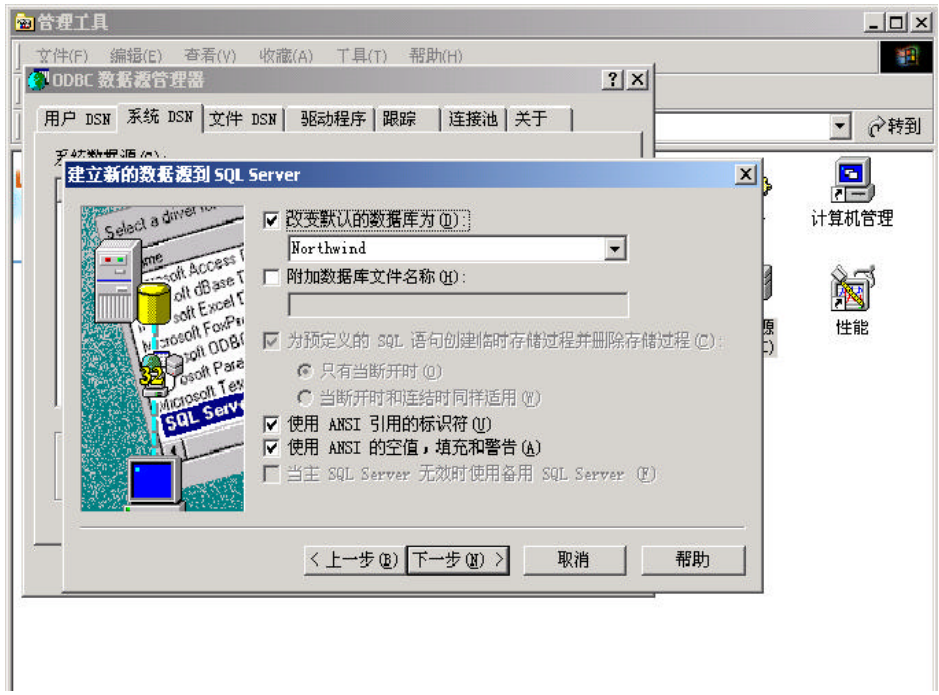

图2-4

```
import java.awt.*;

//在使用JDBC之前，必须引入JAVA的SQL包
import java.sql.*;

class JDBCTest extends Frame {

      TextArea myTextArea;

      public JDBCTest () {

           //设定程序的显示界面
           super("一个简单的JDBC范例");
           setLayout(new FlowLayout());
           myTextArea = new TextArea(30,80);
           add(myTextArea);
           resize(500,500);
           show();
           myTextArea.appendText("数据库查询中，请等待......\n");

      }

      void displayResults(ResultSet results) throws SQLException {
```

```
    //首先得到查询结果的信息
    ResultSetMetaData      resultsMetaData = results.getMetaData();

    int     cols = resultsMetaData.getColumnCount();
    //将等待信息清除

    myTextArea.setText("");

    //显示结果
    while(results.next()) {
         for(int i=1;i<=cols;i++) {
                if(i>1)
                     myTextArea.appendText("\t");
                try{
                     myTextArea.appendText(results.getString(i));
                }

                //捕获空值时产生的异常
                catch(NullPointerException e){
                }
         }
         myTextArea.appendText("\n");
    }
}

public boolean handleEvent(Event  evt) {

    if (evt.id == Event.WINDOW_DESTROY) {
         System.exit(0);
         return true;
    }
    return super.handleEvent(evt);
}

public static void main(String argv[]) throws SQLException,Exception {

    //设定查询字串
    String  queryString = "select * from Customers";

    JDBCTest       myJDBCTest = new JDBCTest();

    //加载驱动程序
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    //建立连接
Connection myConn =
    DriverManager.getConnection("jdbc:odbc:Northwind","riso","riso");
```

```
        Statement       myStmt = myonn.createStatement();

        //执行查询
        ResultSet       myResults = myStmt.executeQuery(queryString);

        myJDBCTest.displayResults(myResults);

        //关闭所有打开的资源
        myResults.close();
        myStmt.close();
        myConn.close();
        }
    }
```

上面的程序对于了解Java语言的读者应该是不难理解的，程序的作用就是：首先建立一个数据库连接，然后执行一个查询，最后将所得的结果显示在屏幕上。程序的执行结果如图2-5所示。

注意：

1) Class.forName()函数指定了加载的驱动程序。

2) getConnection()函数的三个参数中，第二个参数和第三个参数分别表明登录用的用户名和密码。

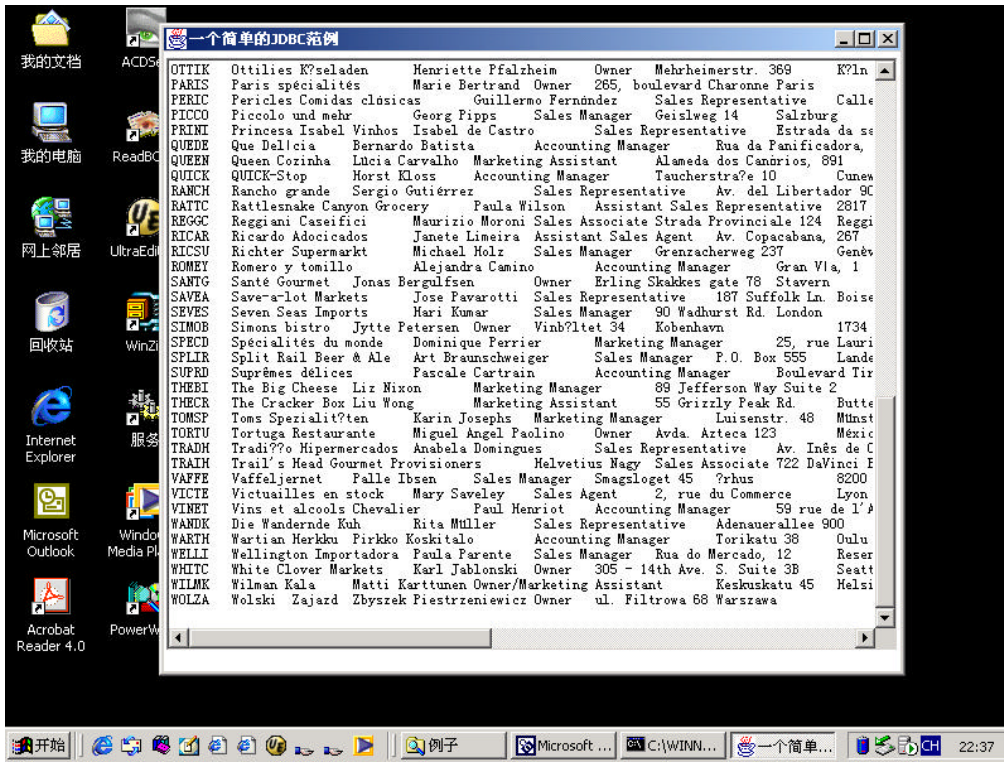

图2-5

第3章　JSP开发平台的建立：Tomcat

自从JSP发布以后，推出了各式各样的JSP引擎。作为世界上用得最多的Web服务器软件——Apache的Apache Group也在进行JSP的实用研究。最初的软件产品是在Apache的Java Servlet引擎即ApacheJServ的基础上实现的GNUJSP，一直到GNUJSP1.0，基本上实现了JSP 1.0标准。另外还出现了一个被称为GSP的产品，是作为GNU体系的一个服务器端的Script语言实现的。

GNUJSP基本上是一个ApacheJServ的附属，它主要是利用Servlet将JSP源文件翻译为一个Servlet的Java语言源文件，然后经Java编译器编译后作为Servlet执行，这样做的好处前面已经说明了。

在完成GNUJSP1.0的开发以后，开发组的成员开始考虑在SUN的JSWDK基础上开发一个可以直接提供Web服务的JSP服务器，当然同时也支持Servlet。这样，Jakarta-Tomcat就诞生了。

作为一个开放源代码的软件，Jakarta-Tomcat有着自己独特的优势：

首先，它容易得到。事实上，任何人都可以从互联网上自由地下载这个软件。无论从http://jakarta.Apache.org还是从其他网站。

其次，对于开发人员，特别是Java开发人员，Tomcat提供了全部的源代码，包括Servlet引擎、JSP引擎、HTTP服务器……，无论是对哪一方面感兴趣的程序员，都可以从这些由世界顶尖的程序员书写的代码中获得收益。

最后，由于源代码的开放及世界上许多程序员的卓有成效的工作，Tomcat已经可以和大部分的主流服务器一起工作，而且是以相当高的效率一起工作。如：以模块的形式被载入Apache，以ISAPI的形式被载入IIS或PWS，以NSAPI的形式被载入Netscape Enterprise Server……。

接下来，读者可以看到：

- 如何安装Tomcat，让它发挥作用。
- 如何让Tomcat和Apache、IIS等一起工作。
- 如何配置Tomcat，让它符合自己的要求。

下面，先来建立一个试验用的JSP页面，读者可以先将以下的代码存为HelloWorld.jsp。

```
<HTML>
    <HEAD>
       <TITLE>JSP测试页面---HelloWorld!</TITLE>
    </HEAD>
    <BODY>
<%
    out.println("<h1>Hello World!<br>世界，你好！</h1>");
%>
    </BODY>
</HTML>
```

3.1　Tomcat的安装和直接使用

在Apache的jakarta项目的主页上，可以看到有Tomcat的超连接，在这里可以找到各种版本

的下载区域，包括当前的发布（Release）版本、开发中的各种版本，其中又分为 Win32版本和Linux版本，其实对于完全由 Java写成的Tomcat，Win32版本和Linux版本没有多大区别，比如Linux版本，在Solaris下也没有问题。这里，主要以Win32版本作为示例。

注意：在安装使用Tomcat之前，先安装JDK，最好是Sun的JDK1.2.2或JDK1.3。

首先，下载jakarta-tomcat.zip包，解压缩到一个目录下，如：“c:\tomcat”。这时，会得到如下的目录结构：

```
tomcat
    ---jakarta-tomcat
        |---bin          Tomcat执行脚本目录
        |---conf         Tomcat配置文件
        |---doc          Tomcat文档
        |---lib          Tomcat运行需要的库文件（JARS）
        |---logs         Tomcat执行时的LOG文件
        |---src          Tomcat的源代码
        |---webapps      Tomcat的主要Web发布目录
|       ---work          Tomcat的工作目录，
    Tomcat将翻译JSP文件到的Java文件和class文件放在这里
```

在Bin目录下，有一个名为startup.bat的脚本文件，执行这个脚本文件，就可以启动 Tomcat服务器，不过，在启动服务器之前，还需要进行一些设置。

首先，设置环境变量。

Win9x在autoexec.bat里用set 语句来设定环境变量，如： set TOMCAT_HOME = c:\tomcat。

在winnt/win2000里可以选择“我的电脑”，右键点出菜单，选择属性，弹出对话框“系统特性”，选择“高级”选项页，然后点按钮“环境变量”，可以编辑系统的环境变量。

- TOMCAT_HOME值：c:\tomcat (用TOMCAT_HOME指示tomcat根目录。
- JAVA_HOME值：c:\java\jdk(用JAVA_HOME指示jdk1.3安装目录)。
- CLASSPATH值：c:\java\jdk\lib\tools.jar。

实际上，对于 CLASSPATH也可以直接打开 tomcat.bat文件，在中间可以找到好几行 set CLASSPATH.....，将自己希望加入的库文件加入到其中即可。

另外，对于JDK1.3，在中文系统上安装之后，系统注册表会有问题，请用 regedit打开注册表查javasoft，位置为hkey_local_machine -> software -> javasoft -> ，找到“Java 运行时环境”把它导出到文件 temp.reg....，然后用notepad编辑它，把“Java 运行时环境”替换成“Java Runtime Environment”，然后导入。

同样，最好也把javasoft注册表项中的“Java 插件”另外复制一份为“Java Plug-in”。

接下来就可以执行TOMCAT_HOME\bin\startup.bat，测试一下Tomcat是否运行正常。

运行Web浏览器，如 Netscape Navigator或Internet Explorer。在浏览器的地址栏中键入：http://127.0.0.1:8080。如果看到Tomcat的信息，那么就说明 Tomcat已经安装成功了。然后测试Tomcat的JSP引擎是否正常工作，即将前面建立的 HelloWorld.jsp文件拷贝到TOMCAT_HOME\w

ebapps\examples\jsp目录下，然后在浏览器的地址栏中键入： http://127.0.0.1:8080/examples/jsp/HelloWorld.jsp，这时候应该可以看到如图 3-1所示的画面：

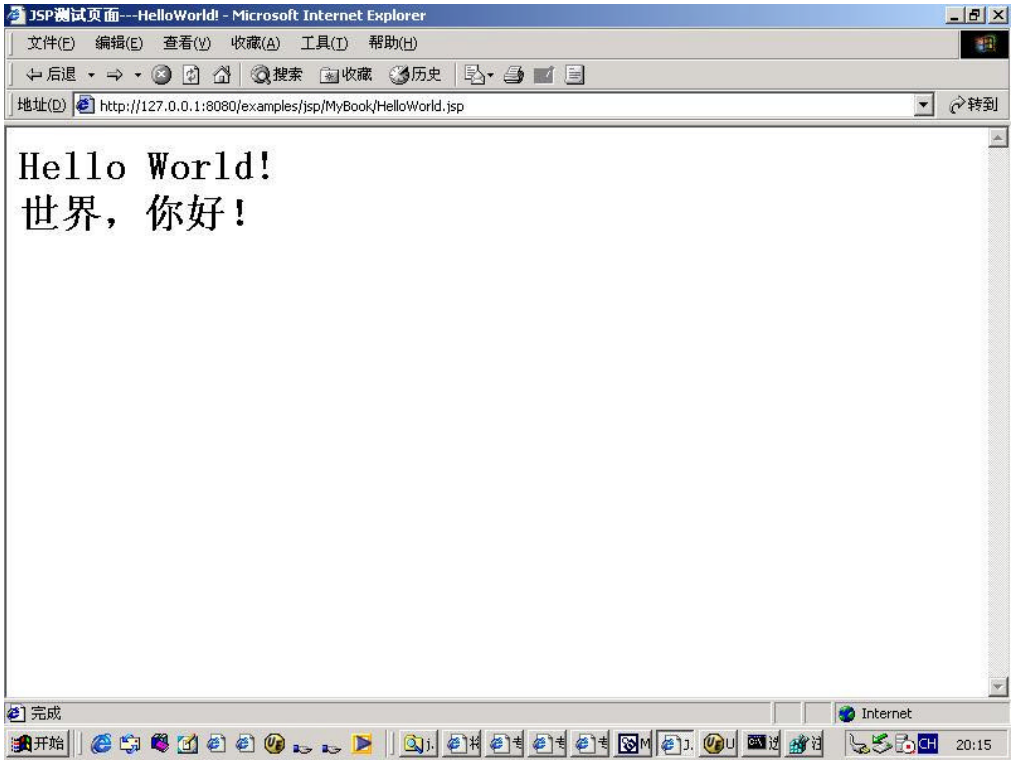

图 3-1

在启动Tomcat的过程中，可能会遇到一些问题，这里就常见问题进行一些说明。

1) 启动Tomcat失败。出现这种情况时，可能有两种现象：

第一种：执行startup.bat以后没有Java窗口出现。

第二种：有Java窗口出现，但是接着自行退出。

对于第一种情况，很可能是 TOMCAT_HOME环境变量设置有问题，打开 startup.bat文件，观察脚本的写法，一般可以发现问题。

对于第二种情况，可能是当前系统中已经有一个服务器占用了 8080端口，这时需要把原先的服务器关闭，或者利用后面讲的 Tomcat的配置方法修改 Tomcat的服务端口。也可能是由于CLASSPATH设置有误，这时需要检查CLASSPATH是否设置正确。

2) 启动Tomcat成功，可以看到首页，但是不能执行 JSP脚本。

这种情况一般是由于 CLASSPATH设置有误或JAVA_HOME设置有误，经过试验发现，当使用Windows自带的java.exe（c:\winnt\system32或c:\windows\system）时可能会出现这种问题。

3.2 Tomcat和Apache的配合

作为 Apache的一个子项目 jakarta-tomcat当然要对 Apache提供强有力的支持，在下载的

Tomcat压缩包解开后，在TOMCAGT_HOME\conf目录下有一个tomcat-Apache.conf文件，这个文件并不是Tomcat自己的配置文件，而是提供给Apache用来使Tomcat能够和Apache一起工作的。实际上，这个文件是在Tomcat的运行过程中自动生成的。

但是，光有这个文件还不能使Apache和Tomcat一起工作，还需要一个Apache的动态载入库文件ApacheModuleJServ.dll，这个文件也可以在网站http://jakarta.Apache.org得到，需要说明的是，对于Linux版本的tomcat，需要的是mod_jserv.so文件，ws50为后缀的文件是Linux下的动态连链库文件。

首先，要得到Apache HTTP服务器。Apache是一个免费而且提供源代码的HTTP服务器，由于Apache强大的性能和用户可以利用源代码构造自己的HTTP服务器的特性，Apache及其衍生出来的产品已经成为世界上应用最多的HTTP服务器，甚至连著名的IBM公司为Websphere应用服务器提供的IBM HTTP Server也是由Apache改造而来的。

在http://jakarta.Apache.org可以得到Apache服务器的最新版本，本书写作的时候，最新的发行版已经到了Apache1.3.12，而Apache2.0a6也已经提供用于测试了。

Windows下的Apache版本是一个安装文件，可以轻松地安装在计算机上。而如果在Linux下使用Apache，那么最好使用源代码包自己进行编译，需要注意的是，编译时需使用选项enable-module=so。

双击Apache_1_3_12_Win32.exe文件进行安装，缺省安装目录为C:\Program Files\Apache Group\Apache，可以修改为自己喜欢的目录。

如果需要修改Apache服务器工作的端口号以及HTML发布目录或者其他Apache的参数，那么可以修改Apache安装目录\conf\httpd.conf，一般可以修改HTML发布目录为自己喜欢的目录。至于端口号，当计算机上还运行有其他Web服务器时可以修改之，一般Windows9x的机器上如果装有PWS就需要修改，而WindowsNT和Windows 2000的机器上如果装有IIS，也需要修改。对于没有连接到网络的机器，有时需要设置一下ServerName。Apache服务器的具体配置请见相关书籍，这里就不讲了。

另外最好将Apache作为一个服务安装在运行WindowsNT和Windows 2000的电脑上。这只需要执行开始→程序→Apache Web Server→Install Apache as a service即可。

打开浏览器，在地址栏中键入http://127.0.0.1:Apache运行的端口号，如果能够见到Apache的欢迎页面，或者是一大堆文件让你选择，就可以认为Apache服务器已经开始工作了。

Apache HTTP服务器配置成功以后，就可以着手让Tomcat和Apache一起工作。首先，将得到的ApacheModuleJServ.dll文件拷贝到Apache安装目录下的modules子目录下，Linux的用户将mod_jserv.so文件拷贝到Apache安装目录的libexec目录下，然后将Apache安装目录下的httpd.conf文件用文本编辑器打开，在最后面加入下面的指令：

INCLUDE Tomcat_Home\conf\tomcat.conf ——对于Windows用户。

或INCLUDE Tomcat_Home/conf/tomcat.conf ——对于Linu用户。

上面的Tomcat_Home指的是Tomcat的安装目录。

最后，在httpd.conf文件中加上一行:LoadModule jserv_module modules/ApacheModuleJServ.dll。

对于Linux下的用户，一般不需要手动加上LoadModule jserv_module libexec/mod_jserv.so这一行，tomcat-Apache.conf文件已经缺省加上了，如果没加，自行加上即可。

一切就绪以后，重新启动 Apache服务器和 Tomcat，在浏览器的地址栏中键入： http://127.0.0.1:Apache运行的端口号 /examples/jsp/，如果能够看到 Tomcat的JSP示例列表，就说明Tomcat已经和Apache一起工作了。

3.3 Tomcat和IIS的配合

Windows平台下最常用的Web服务器无疑是IIS（包括PWS），对于IIS，Tomcat也提供了配合工作的方法，使用这种方法，可以为本来不具有 Java Servlet和JSP功能的IIS增加处理JSP和Java Servlet的功能。

为了使Tomcat和IIS一起工作，首先要得到isapi_redirect.dll，这是一个IIS的插件（Plug-in），可以从http://jakarta.Apache.org/直接下载编译好的版本，也可以自己使用 Visual C++编译得到。得到以后，放到一个自己喜欢的目录，例如 c:\tomcat\Jakarta-tomcat\bin\iis\i386\ 目录下。

另外，在使IIS和Tomcat配合的过程中，还需要用到另外两个 Tomcat的配置文件，一个是workers.properties，这个文件定义了Tomcat的工作进程使用的主机和端口。在Tomcat的conf目录中有一个示范性的workers.properties文件。另一个是uriworkermap.properties，这个文件是映射URL目录和Tomcat工作进程的。同样，在Tomcat的conf目录中有一个示范性的uriworkermap.properties文件。

首先，配置isapi_redirect.dll。

1) 在系统注册表中建立一个新的键值： HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Jakarta Isapi Redirector\1.0。

2) 添加一个名为extension_uri的字符串值为/jakarta/isapi_redirect.dll。

3) 添加一个名为log_file的字符串值为c:\tomcat\Jakarta-tomcat\logs\isapi.log。

4) 添加一个名为log_level的字符串值为debug、inform、error、emerg中的一个。

5) 添加一个名为worker_file的字符串值为

6) c:\tomcat\jakarta-tomcat\conf\workers.properties。

7) 添加一个名为worker_mount_file的字符串值为

8) c:\tomcat\jakarta-tomcat\conf\ uriworkermap.properties。

然后，打开IIS的管理控制台，在需要使用Tomcat提供附加的JSP和Java Servlet服务的Web站点中添加一个虚拟目录。注意，一定要使用“ jakarta ”作为虚拟目录的名称，这个虚拟目录的实际物理位置应当是包含isapi_redirect.dll文件的目录，这里假设为c:\tomcat\Jakarta-tomcat\bin\iis\i386。在设定虚拟目录时注意要设此虚拟目录为可执行。如果是在PWS中，一样处理。

接着，在IIS的控制台中为此Web站点添加一个ISAPI过滤器（在此Web站点上点击鼠标右键，选择属性）。名称随意，但过滤器要设定为 isapi_redirect.dll这个文件。如果使用的是 PWS就比较麻烦了。需要使用注册表编辑器，在键 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3SVC\Parameters中，有一个名为Filter Dlls的键值，在这个键值中添加 isapi_redirect.dll，注意要包含完整的路径。

最后，重新启动IIS或PWS，最好是能够重新启动计算机。

启动Tomcat以后，打开浏览器，在地址栏中键入 http://127.0.0.1/examples/，如果能够看到jsp和servlet两个目录，就说明利用 isapi_redirect.dll所作的重定向已经成功，可以执行一下 jsp目录下的例子做试验。

3.4 Tomcat的配置和常见问题

Tomcat为用户提供了一系列的配置文件来帮助用户配置自己的 Tomcat，和Apache HTTP不同，Tomcat的配置文件主要是基于 XML的；如server.xml、web.xml.....，只有workers.properties和uriworkermap.properties等少数几个文件是传统的配置文件。本节将详细讨论 Tomcat的主要配置文件以及如何利用这些配置文件解决常见问题。

3.4.1 Tomcat的主配置文件：server.xml

观察server.xml，可以发现其中有如表0一些元素。

表3-1

元 素	描 述
Server	Server元素是server.xml文件的最高级别的元素， Server元素描述一个Tomcat服务器，一般来说用户不用关心这个元素。一个 Server元素一般会包括Logger和ContextManager两个元素
Logger	Logger元素定义了一个日志对象，一个日志对象包含有如下属性： 1) name。表示这个日志对象的名称。 2) path。表示这个日志对象包含的日志内容要输出到哪一个日志文件。 3) verbosityLevel。表示这个日志文件记录的日志的级别。 一般来说，Logger对象是对 Java Servlet、JSP和Tomcat运行期事件的记录
ContextManager	ContextManager定义了一组ContextInterceptors（ContextManage的事件监听器），RequestInterceptors（ContextManage的事件监听器），Contexts（Web应用程序的上下文目录）和它们的 Connectors（连接器）的结构和配置。 ContextManager包含如下一些属性： 1) debug。记录日志记录调试信息的等级。 2) home。webapps/、conf/、logs/和所有Context的根目录信息。这个属性的作用是从一个不同于 TOMCAT_HOME的目录启动Tomcat。 3) workDir。Tomcat工作目录。
ContextInterceptor 和RequestInterceptor	两者都是监听ContextManager的特定事件的拦截器。ContextInterceptor监听Tomcat的启动和结束事件信息。而RequestInterceptor监听用户对服务器发出的请求信息。一般用户无需关心这些拦截器，对于开发人员，需要了解这就是全局性的操作得以实现的方法
Connector	Connector（连接器）对象描述了一个到用户的连接，不管是直接由Tomcat到用户的浏览器还是通过一个 Web服务器。 Tomcat的工作进程和由不同的用户建立的连接传来的读 /写信息和请求 /答复信息都是由连接器对象管理的。对连接器对象的配置中应当包含管理类、TCP/IP端口等内容

（续）

元　素	描　述
Context	每一个Context都描述了一个Tomcat的Web应用程序的目录。这个对象包含以下属性： 1) docBase。这是Context的目录。可以是绝对目录也可以是基于ContextManage的根目录的相对目录。 2) path。这是Context在Web服务时的虚拟目录位置和目录名。 3) debug。日志记录的调试信息记录等级。 4) reloadable。这是为了方便Servlet的开发人员而设置的，当这个属性开关打开的时候，Tomcat将检查Servlet是否被更新而决定是否自动重新载入它

1. 加入自己的日志文件

添加Logger对象就可以加入自己的日志文件，添加工作相当简单，只需要将作为示例的Logger对象复制一份，然后修改一下前面介绍的几个属性就可以了。在设定了 Logger以后，就可以在自己的Servlet中使用ServletContext.log()方法来建立自己的日志文件。

2. 设定新的JSP目录

设立新的JSP工作目录是比较简单的，只需要添加一个 Context对象就可以了。如，要在c:\jsp目录下开发JSP项目，并且让用户可以使用/mybook/虚拟目录访问，则：

```
<Context path="/mybook" docBase="c:\jsp" debug="0" reloadable="true" >
 </Context>
```

一般来说，这样就可以直接执行 JSP文件了，如果进一步想要在这下面建立 Web应用程序，那么还需要进一步的配置，具体方法在后面论述。

3.4.2 Windows下代码保护的问题

在Windows下使用Tomcat时有一个问题需要注意，可以做一个试验，启动 Tomcat后，在浏览器的地址栏中键入：http://127.0.0.1:8080/examples/jsp/ HelloWorld.JSP（注意后缀要大写）。就会发现奇怪的现象，浏览器的窗口中什么都没有，查看 HTML源文件就会发现，这个JSP文件的源代码被Tomcat完全输出到了浏览器！如果是这样，岂不是服务器端的任何源代码都会被暴露在互联网上。

实际上，解决方法很简单，把各种后缀的组合全部写到 Tomcat_Home\conf\web.xml里就可以了，这样tomcat会将不同后缀名的jsp分开对待，就不会泄露代码了。

```
<servlet-mapping>
    <servlet-name>
         jsp
    </servlet-name>
    <url-pattern>
         *.jsp
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
```

```
    <servlet-name>
        Jsp
    </servlet-name>
    <url-pattern>
        *.Jsp
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        JSp
    </servlet-name>
    <url-pattern>
        *.JSp
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        JsP
    </servlet-name>
    <url-pattern>
        *.JsP
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        JSP
    </servlet-name>
    <url-pattern>
        *.JSP
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        jSp
    </servlet-name>
    <url-pattern>
        *.jSp
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>
        jSP
    </servlet-name>
    <url-pattern>
        *.jSP
    </url-pattern>
</servlet-mapping>
<servlet-mapping>
```

```
    <servlet-name>
        jsP
    </servlet-name>
    <url-pattern>
        *.jsP
</url-pattern>
</servlet-mapping>
```

3.4.3 Apache、IIS和Tomcat协作时工作目录的添加

1. Apache

由于Jakarta-Tomcat项目是Apache的一个子项目，所以向Tomcat-Apache协作的Web服务器添加工作目录时只需要修改 Tomcat-Apache.conf文件就可以了。也许读者会觉得奇怪， Tomcat-Apache.conf文件不是在Tomcat启动时自动生成的吗？的确如此，但是 Tomcat自动生成的Tomcat-Apache.conf文件仅仅是Tomcat提供的一个缺省配置文件而已，如果需要，可以修改，然后存放在另外的目录中，或是更名，再在 httpd.conf文件中将这个新的文件包含进来就可以了。

为什么Tomcat不自动修改Tomcat-Apache.conf文件以适应工作目录添加的需要呢？ Tomcat的确修改了Tomcat-Apache.conf文件，但是修改的结果显然是不正确的。如，前面在 server.xml中添加了工作目录/mybook-->c:\jsp后，Tomcat修改Tomcat-Apache.conf文件，添加了这么几行：

```
Alias /mybook C:\tomcat\jakarta-tomcat\webapps\mybook
<Directory "C:\tomcat\jakarta-tomcat\webapps\mybook">
    Options Indexes FollowSymLinks
</Directory>
ApJServMount /mybook/servlet /mybook
<Location /mybook/Web-INF/ >
    AllowOverride Non
    deny from all
</Location>
```

这显然是有问题的，尽管虚拟目录是 /mybook，但是实际的目录并不是 C:\tomcat\jakarta-tomcat\webapps\mybook。查看Tomcat的源代码C:\tomcat\jakarta-tomcat\src\org\Apache\tomcat\task\ApacheConfig.java文件可以发现， Tomcat在生成Tomcat-Apache.conf文件的时候，简单地在虚拟目录前面加上原先Tomcat的缺省webapp目录作为新的工作目录：

```
pw.println("Alias " + path + " " +
    FileUtil.patch(tomcatHome + "/webapps" + path));
pw.println("<Directory \"" +
   FileUtil.patch(tomcatHome + "/webapps" + path) +
   "\">");
pw.println("    Options Indexes FollowSymLinks");
pw.println("</Directory>");
```

那么如何解决这个问题呢？修改 Tomcat的源代码也可以，不过，对于一般的用户，如前所述直接修改Tomcat-Apache.conf文件更现实一些。修改的方法举例如下：

```
Alias /mybook C:\jsp
```

```
<Directory "C:\jsp">
    Options Indexes FollowSymLinks
</Directory>
ApJServMount /mybook/servlet /mybook
<Location /mybook/Web-INF/ >
    AllowOverride None
    deny from all
</Location>
```

也就是简单地将原先错误的实际目录 C:\tomcat\jakarta-tomcat\webapps\mybook修改为c:\jsp就可以了。

修改完文件后，一定要存为另一个文件并修改 httpd.conf文件将这个文件包含进来，否则，重新启动 Tomcat后，这个正确的 Tomcat-Apache.conf文件会被 Tomcat重新生成的 Tomcat-Apache.conf文件覆盖掉。最后，重启 Apache和Tomcat就可以了。

2. IIS

与Apache和Tomcat几乎无缝的配合不一样， IIS和Tomcat的配合多少有些复杂，需要向ISAPI Redirect添加新的内容。不过还好， Tomcat的uniworkermap.conf文件将这个过程简单化了。

需要做的事情共分两步：

1) 向Tomcat中添加一个工作目录。前面已经讲述了如何实现，这里依然使用 /mybook-->c:\jsp这个例子。

2) 向ISAPI Redirect添加工作目录。使用文本编辑器打开文件 uniworkermap.conf，添加一行：

```
/mybook/*=ajp12
```

然后重新启动IIS和Tomcat就可以了。

3.4.4 设定Tomcat作为Windows的服务而启动

手工启动Tomcat显然不是一个合适的使用 Tomcat作为Web服务的方法，在 Linux下可以通过修改启动脚本自动启动 Tomcat，在Windows下则可以设定Tomcat作为Windows的服务而启动。

Tomcat作为Windows NT/2000的一个服务是需要借助工具的

1) 下载工具，这里作为例子的是 gservany ----将NT下的一般应用程序作为服务运行的工具。下载网址为 http://www.advok.com/gservany.html。将 zip文件解压缩，将 gservany.exe放入winnt\system32目录下，(以防以后被误删)。

2) 在NT的Command（命令行模式）下输入： gservany -i tomcat “ C:\jakarta-tomcat\bin ” “ startup.bat ”“ C:\jakarta-tomcat\bin ” “ shutdown.bat ”。

3) 启动service管理器，会看到tomcat service被装上，加些注释说明这个 service实际干什么，再修改启动类型为 “ 自动 ”。然后再启动它。

这样，就成功地将Tomcat作为Service安装在NT下了。

其实，将NT下的应用程序作为服务安装在 NT中的工具还有很多，任何一种都应该可以将

Tomcat加入到NT的服务中。

3.4.5 在Tomcat中建立新的Web应用程序

JSP主要是为建立Web网站而开发的技术，这种技术由Web应用程序的一整套Web文件（jsp，servlet，html，jpg，gif，class.....）所组成。Tomcat为Web应用程序的建立提供了一系列的帮助，下面分步骤描述。

1. 应用程序的目录和结构

按照Tomcat的规范，从/example例子目录来看，Tomcat的Web应用程序应该由如表3-2所示目录组成的。

表3-2

*.html, *.jsp, etc.	这里可以有许多目录，由用户的网站结构而定，实现的功能应该是网站的界面，也就是用户主要的可见部分。除了HTML文件、JSP文件外，还有js（JavaScript）文件和css（样式表）文件以及其他多媒体文件等等
Web-INF/web.xml	这是一个Web应用程序的描述文件。这个文件是一个XML文件，描述了Servlet和这个Web应用程序的其他组件信息，此外还包括一些初始化信息和安全约束等等
Web-INF/classes/	这个目录及其下的子目录应该包括这个Web应用程序的所有Servlet文件，以及没有被压缩打入JAR包的其他class文件和相关资源。注意，在这个目录下的Java类应该按照其所属的包组织目录
Web-INF/lib/	这个目录下包含了所有压缩到JAR文件中的类文件和相关文件。比如：第三方提供的Java库文件、JDBC驱动程序等等.....

2. web.xml文件

web.xml文件包含了描述整个Web应用程序的信息。下面以一个web.xml文件为例，讲解里面的各个对象。

```
web.xml：
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<!-
Web应用程序的主要描述
-->
<display-name>My Web Application</display-name>
<description>
    在这里加入Web应用程序的描述信息
</description>
<!-
下面定义了Web应用程序的初始化参数，
在JSP或Servlet文件中使用下面的语句
来得到初始化参数
    String value =
         getServletContext().getInitParameter("name");
这里可以定义任意多的初始化参数
-->
```

```
<context-param>
  <param-name>webmaster</param-name>
  <param-value>myaddress@mycompany.com</param-value>
  <description>
    这里包含了初始化参数的描述
  </description>
</context-param>
<!-
下面的定义描述了组成这个Web应用程序的
Servlet，还包含初始化参数。在Tomcat中，也
可以将放在Web-INF/classes中的Servlet直接以
servlet/Servlet名访问，但是一般来说，不推荐这
样使用。而且这样的使用方法还会导致Servlet的
相关资源组织的复杂性。所以一般来说推荐将所
有的Servlet在这里定义出来。初始化参数可以在
Servlet中一这种语句的到：
    String value =
         getServletConfig().getInitParameter("name");
-->
<servlet>
  <servlet-name>controller</servlet-name>
  <description>
    这里加入这个Servlet的描述
      </description>
  <servlet-class>com.mycompany.mypackage.ControllerServlet</servlet-class>
  <init-param>
    <param-name>listOrders</paramName>
    <param-value>com.mycompany.myactions.ListOrdersAction</param-value>
  </init-param>
  <init-param>
    <param-name>saveCustomer</paramName>
    <param-value>com.mycompany.myactions.SaveCustomerAction</param-value>
  </init-param>
  <!-
 服务器启动后这个Servlet加载的时间
  -->
  <load-on-startup>5</load-on-startup>
</servlet>
<servlet>
  <servlet-name>graph</servlet-name>
  <description>
    这个Servlet的描述
  </description>
</servlet>
<!-
Servlet映射对应了一个特殊的URI请求
到一个特殊的Servlet的关系
-->
```

```
    <servlet-mapping>
      <servlet-name>controller</servlet-name>
      <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>graph</servlet-name>
      <url-pattern>/graph</url-pattern>
    </servlet-mapping>
    <!-
    设定缺省的Session过期时间
    -->
    <session-config>
      <session-timeout>30</session-timeout>    <!-- 30 minutes -->
    </session-config>
  </web-app>
```

3. 将应用程序打包为WAR文件

WAR文件是包装Web应用程序的一种方法，使用 WAR文件，既方便了管理各种相关文件，又减小了整个应用程序的体积。

下面先来看一看将Web应用程序打包为WAR文件的语法：

```
packager -webArchive[-classpath servletorjspbean/classes [ -
   classFiles package/MyClass1.class: package/MyClass2.class ] ]
   <content-root> [-contentFiles login.jsp:index.html:images/me.gif]
   web.xml myWebApp.war
```

下面是一个简单的应用示例，将 myWebPage.xml的配置和myWebPageDir/下的文件打入包myWebPage.war中：

```
packager -webArchive myWebPageDir myWebPage.xml myWebPage.war
```

使用 -contentFiles 标志可以添加个别的目录文件

```
packager -webArchive myWebPageDir -contentFiles Hello.jsp
    myWebPage.xml myWebPage.war

packager -webArchive myWebPageDir -contentFiles Hello.jsp:Hello.html
    myWebPage.xml myWebPage.war
```

假定Servlet文件在classes/package/Servlet1.class，指定Servlet和JSP文件：

```
packager -webArchive -classpath classes myWebPageDir -contentFiles
    Hello.jsp myWebPage.xml myWebPage.war.
```

下面示例如何仅仅包含 package/Servlet1.class 和 packageB/Servlet.class 两个文件到WAR文件中：

```
packager -webArchive -classpath classes -classFiles package/
    Servlet1.class:packageB/Servlet.class myWebPageDir -contentFiles
    Hello.jsp myWebPage.xml myWebPage.war
```

最后，需要说明的是，每个 .war文件前面的.xml文件就是前面讲过的 web.xml文件。生成的WAR文件可以直接放在包含Web应用程序的目录下使用。

第4章 JSP的语法和语义

本章将详细介绍JSP的语法和语义（JSP1.1）。如果读者接触过ASP或PHP，将会发现JSP的语法稍显复杂；另一方面，如果读者有Java语言程序设计的经验，就会觉得JSP相当简单，其实，作为J2EE的成员，JSP本来就已经成为Java的一部分了。

在JSP中，主要包含以下内容：

指令。指令提供该页的全局信息，例如，重要的状态，错误处理，是否是session的一部分等。

声明。页面范围的变量和方法的声明。

脚本。嵌入页面内java代码。

表达式。把表达式变成string型以便于包含在页面的输出中。

下面将一一介绍。

4.1 通用的语法规则

JSP的页面是由许多的“元素”组成的，本节描述的语法规则对于这些“元素”都是成立的，所以称之为“通用”的语法规则，也就是这些元素共有的特性。

4.1.1 元素的语法规则

大部分的JSP元素都基于“XML”的语法，这些基于“XML”语法的JSP元素一般来说都有一个包含有元素名称的起始标志，可能还包含有属性设置，一些可选项，和一个结束标志。当然，JSP页面的起始标记和结束标记一定要在同一个页面中，有的元素也仅仅有一个包含属性设置的标志，举例如下：

```
<mytag attr1="attribute value" ...>
   body
</mytag>

<mytab attr1="attribute value" .../>
```

读者会发现，前面讲述的Tomcat的配置文件server.xml和web.xml中已经使用了这种语法形式。

脚本元素则使用的是如ASP般的语法：

```
<%.............%>
```

实际上，每一个JSP页面都应该可以转换为等价的XML页面，在下一章中将详细介绍作为XML的JSP。

JSP元素的属性也和XML中定义的属性遵从同样的原则，JSP页面的属性值一定要使用引号括起来，双引号（”）和单引号（’）都可以使用。另外，作为替代，也可以使用 '

和&quto;来表示双引号和单引号。

4.1.2 JSP中的相对路径

在JSP中，可以使用相对路径来代替绝对路径，在JSP的语法规范中称之为“URI”，感兴趣的读者可以在RFC2396中找到比较完整的描述，这里举几个例子来说明相对路径的概念：

```
"jspRelativeTest.jsp"
"/ jspRelativeTest.jsp"
"../ jspRelativeTest.jsp"
```

在这三行代码中，都假设是在c:\tomcat\Jakarta-tomcat\webapps\examples\jsp\test.jsp中使用上述相对路径。

对于第一行代码来说，显然文件jspRelativeTest.jsp的位置应当为：c:\tomcat\Jakarta-tomcat\webapps\examples\jsp\jspRelativeTest.jsp，这是比较容易理解的。但是对于第二行代码就要注意了，在JSP中，当相对路径以“/”开头的时候，不是相对于网站的根目录，而是相对于包含这个JSP文件的Web应用程序的根目录，也就应当是：c:\tomcat\Jakarta-tomcat\webapps\examples\jspRelativeTest.jsp。如果读者对Web应用程序的概念还不清楚，请参见前几章中的相关章节。第三行代码对于熟悉UNIX或DOS命令行方式的读者应该不陌生，这种情况下，文件jspRelativeTest.jsp的位置应当为：c:\tomcat\Jakarta-tomcat\webapps\examples\jsp\jspRelativeTest.jsp。

4.2 注释

一般来说，可以认为在JSP页面中包含有两种不同类型的注释：一种是JSP本身的，用于描述JSP程序代码，另一种是JSP生成的页面的，也就是HTML（或WML...）的注释，用于描述JSP页面执行后的结果页面的功能。HTML页面的注释这里就不说了，下面是JSP本身的注释语法的例子：

```
<%-- 这是一个JSP的注释 --%>
```

实际上，由于在JSP的“脚本代码”部分中，也就是后面会提到的“Scriptlet”直接使用“<% ..... %>”包含起来的部分中可以使用Java的语法，所以下面形式的注释也就理所当然的可以使用了：

```
<% /*这是一个Scriptlet中的注释 */ %>
<% /**这也是一个Scriptlet中的注释，可以用javadoc从生成的Java文件中提取出注释来 */ %>
```

4.3 指令

前面已经讲过指令在JSP中的地位，指令一般来说有如下的形式：

```
<%@ directive {attr="value"} %>
```

指令的这种语法形式尽管简单明了，但并不是符合XML的，第5章将讲述指令的XML语法。

4.3.1 “page”指令

page指令描述了和页面相关的指示信息。在一个JSP页面中，page指令可以出现多次，但是

每一种属性却只能出现一次，重复的属性设置将覆盖掉先前的设置。

page指令的基本语法格式如下：

```
<%@ page page_directive_attr_list %>
page_directive_attr_list ::= {language=" scriptingLanguage"}
                             { extends=" className"}
                             { import=" importList"}
                             { session="true|false" }
                             { buffer="none| sizekb" }
                             { autoFlush="true| false" }
                             { isThreadSafe="true|false" }
                             { info=" info_text" }
                             { errorPage=" error_url" }
                             { isErrorPage="true|false" }
                             { contentType="ctinfo" }
```

下表4-1是对这些属性的解释：

表 4-1

属性名和可选值	说 明
language = “ java ”	language 变量告诉server在文件中将采用哪种语言，在 JSP当前的规范（JSP1.1）中，java是JSP唯一支持的语法
extends = “ package.class ”	extends 变量定义了由JSP页面产生的servlet的父类，一般来说，这个属性不会用到，但是当需要实现一些特殊功能时，也是比较方便的
import = “ package.*,package.class ” 例如： <%@ page import= ” java.util.* ” %>	import 变量和任何一个 java程序的第一部分一样。同样，它总是被放在JSP文件的顶部。import变量的值是一系列用逗号分开的列表，表明想要引入的包和类 注意： java.lang.* javax.servlet.* javax.servlet.jsp.* javax.servlet.http.* 已经缺省地被JSP引擎引入了
session = “ true|false ”	session 变量的缺省值是true，表示当前页面中将有一个缺省的名为“ session ”的对象来表示当前会话。“ session ”对象的类型是： javax.servlet.http.HttpSession
buffer = “ none|8kb|sizekb ”	决定输出流（out对象）是否需要缓冲，缺省值是 8kb。与autoFlush一起使用
autoFlush = “ true|false ”	确定是否自动刷新输出缓冲，如果设成 true，则当输出缓冲区满的时候，刷新缓冲区而不是抛出一个异常
isThreadSafe = “ true|false ”	缺省值是true，如果多个客户请求发向JSP引擎时，可以一次被处理。JSP程序员要处理同步时共享的状态，以保证同步时确实是安全的。 如果isThreadSafe被设成false，则采用单线程模式控制客户端访问该页 这没有使你脱离异常分枝，然而，在 server可能的情况下，通过它的判断，创建该页的多个实例运行以处理多个装入的客户请求。而且，这无法保证同一个客户端的连续请求定位到同一个 JSP页面的实例上，在多个页面请求之间的共享资源和状态必须同步

（续）

属性名和可选值	说　明
info = “ text ”	页面信息通过页面的 Servlet.getServletInfo()来获得页面可以被访问的内容的类型
ErrorPage = “ pathToErrorpage ”	给出一个JSP文件的相对路径，这个 JSP文件用于处理没被处理的例外。这个JSP文件要把isErrorPage设成true。 需要注意的是：JSP是通过使用 ServletRequest对象的setAttribute()方法将名为javax.servlet.jsp.jspException的对象存储起来实现的。 另外，当 AutoFlush设为true的时候，在 JspWriter中的数据刷新到 ServletResponse中以后，任何从JSP文件到错误处理文件的未捕获异常将无法被正常发送
isErrorPage = “ true|false ”	标志一个页面为错误处理页面。设置为 true时，在这个JSP页面中的缺省对象 exception将被定义，其值将被设定为呼叫此页面的 JSP页面的错误对象。缺省为 false
ContentType = “ text/html; charset = ISO - 8859 -1 ”	设置JSP文件和最终文件的 mime类型和字符集的类型。这一项必须在文件的前部，任何一个其他字符在文件中出现之前。缺省为： ContentType = “ text/html; charset = ISO - 8859 -1 ”

4.3.2 Include指令

include指令的作用是包含另一个文件，其语法相当简单：

```
<%@ include file="......" %>
```

在这个指令中应该使用前面讲述的 JSP的相对路径表示法。需要说明的是， JSP还有另外一种包含其他文件的方法：

```
<jsp:include page="" />
```

表4-2比较了两者的异同：

表 4-2

语　法	状　态	对　象	描　述
<%@ include file= ” ..... ” %>	编译时包含	静态	JSP引擎将对所包含的文件进行语法分析
<jsp:include page= ”” />	运行时包含	静态和动态	JSP引擎将不对所包含的文件进行语法分析

4.3.3 taglib指令

taglib指令用于指示这个JSP页面所使用的标签库，标签库的具体用法属于 JSP比较高级的内容，这里就先不讨论了，先讲述一下基础语法：

```
<%@ taglib uri=" tagLibraryURI" prefix=" tagPrefix" %>
```

表4-3是对各个属性的解释。

表 4-3

属 性	说 明
uri	描述这个标签库位置的URI，可以是相对路径或绝对路径
tagPrefix <myPrefix:myTag>	定义了一个指示使用此标签库的前缀，例如将 tagPrefix设为myPrefix时，可以使用下面的语句来使用此标签库中的 myTag标签：
	下面这些前缀已经保留： jsp:, jspx:, java:, javax:, servlet:, sun:, 和 sunw: 目前，空的tagPrefix将被忽视

4.4 内置对象

为开发的方便，JSP中内置了一些对象，不需要预先声明就可以在脚本代码和表达式中随意使用，前面已经接触到的session和exception就是两个内置对象，表4-4详细讲述JSP中的这些内置对象：

表 4-4

对 象	类 型	描 述	作 用 域
request	javax.servlet.ServletRequest的子类	客户端的请求，通常是 HttpServlet-Request的子类，如果客户的请求中有参数，则该对象就有一个参数列表	request（用户请求期）
response	javax.servlet.ServletResponse的子类	JSP 页面的响应，是 HttpServletRes-ponse的子类	page（页面执行期）
pageContext	javax.servlet.jsp.PageContext	页面的属性和需要通过标准 API来访问的相关对象（本质上是构成服务器环境来让JSP运行的一些对象），以便JSP引擎来编译页面。但是，不同 server对这些属性和对象的实现方式不同 解决方案是JSP引擎编译用factory 类返回的服务器的 PageContext 类的实现方法。PageContext类和request、response 对象以及page指令的一些属性（errorpage ,session,buffer,autoflush）同时被初始化，同时提供request请求的相关的对象	page（页面执行期）
session	javax.servlet.http.HttpSession	HTTP session 是与request联合的对象	session（会话期）
application	javax.servlet.ServletContext	servlet的环境通过调用getServletConfig（）.getContext（）方法获得	application（整个Web应用程序运行期）
out	javax.servlet.jsp.JspWriter	代表输出流的对象	page（页面执行期）
config	javax.servlet.ServletConfig	页面的ServletConfig对象	page（页面执行期）
page	java.lang.Object	指向页面自身的方式（在 java代码中多以this替代）	Page（页面执行期）
exception	java.lang.Throwable	没有被 Throwable捕获的错误。传向了errorpage的URI	page（页面执行期）

4.5 脚本元素

在JSP中，主要的程序部分就是脚本元素，其中包括三个部分：声明（ Declaration)、表达式（Expression）和代码（Scriptlet)。从功能上讲，声明用于声明一个或多个变量，表达式将是一个完整的语言表达式，而代码部分将是一些程序片断。

三个脚本元素的基本语法都是以一个“ <% ”开头，而以一个“ %> ”结尾的。

声明的例子：

```
<%! this is a declaration %>
```

代码的例子：

```
<% this is a scriptlet %>
```

表达式的例子：

```
<%= this is an expression %>
```

脚本元素也具有相应的XML兼容语法，将在第6章介绍。

4.5.1 声明

JSP中的声明用于声明一个或多个变量和方法，并不输出任何的文本到 out输出流去。在声明元素中声明的变量和方法将在JSP页面初始化时初始化。

语法为：

```
<%! declaration(s) %>
```

举例如下：

```
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

实际上，声明变量和方法的语句完全可以放在 Scriptlet中，两者有什么不一样呢？放在<%!.....%>中的声明语句在编译为 Servlet的时候将作为类的属性而存在，而放在 Scriptlet中的声明将在类的方法内部被声明。

4.5.2 表达式

JSP中的表达式可以被看作一种简单的输出形式，需要注意的是，表达式一定要有一个可以输出的值才行。

语法为：

```
<%= expression %>
```

举例如下：

```
<%= (new java.util.Date()).toLocaleString() %>
```

4.5.3 脚本代码

所谓脚本代码，就是 Scriptlet，也就是JSP中的代码部分，在这个部分中可以使用几乎任何

Java的语法。

语法为：

```
<% scriptlet %>
```

举例如下：

```
<%
   if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) {
%>
   Good Morning
<%
   } else {
%>
   Good Afternoon
<%
   }
 %>
```

4.6 动作

动作可以影响输出的文本流，使用、编辑、建立对象。在 JSP中，有一些基本的动作，用户也可以添加自己的动作，这需要使用标签库的知识。 JSP中的动作是完全基于 XML的，下面就来看看JSP由哪些标准的动作以及具有哪些属性。

4.6.1 id和scope属性

id属性和scope属性是每一个JSP动作都具有的属性，其中 id表示一个动作的名称，而 scope则表示一个动作的作用域。

scope作用域的取值如表4-5所示。

表 4-5

作用域取值	有 效 范 围
page	由javax.servlet.jsp.PageContext得到在用户请求此页面的过程中有效
request	由ServletRequest.getAttribute(name)得到在用户的整个请求过程中有效
session	由HttpSession.getValue(name)得到在用户的整个会话期内有效
application	由ServletContext. getAttribute(name)得到在Web应用程序执行期间有效

4.6.2 标准动作

JSP规范书中规定了一些标准的动作，凡是符合 JSP规范的JSP引擎都应当实现这些标准的动作，下面将一一介绍JSP1.1中规定的标准动作。

1. <jsp:useBean>

<jsp:useBean>大概是JSP中最重要的一个动作，使用这个动作， JSP可以动态使用JavaBeans组件来扩充JSP的功能，由于JavaBeans在开发上及<jsp:useBean>在使用上的简单明了，使得 JSP

的开发过程和以往其他动态网页开发工具有了本质上的区别。尽管 ASP等动态网页技术也可以使用组件技术，但是由于 ActiveX控件编写上的复杂和使用上的不方便，实际的开发工作中组件技术使用得并不多。

<jsp:useBean>的语法如下：

```
<jsp:useBean id=" name" scope="page|request|session|application"
    typeSpec />
typeSpec   ::= class=" className" |
                  class=" className" type=" typeName" |
                  type=" typeName" class=" className" |
                  beanName=" beanName" type=" typeName" |
                  type=" typeName" beanName=" beanName" |
                  type=" typeName"
```

如果在<jsp:useBean>中需要加入其他的元素，那么使用下面的语法：

```
<jsp:useBean id=" name" scope="page|request|session|application"
     typeSpec >
     body
</jsp:useBean>
```

这里有几个语法的例子：

```
<jsp:useBean id="connection" class="com.myco.myapp.Connection" />
<jsp:useBean id="connection" class="com.myco.myapp.Connection">
     <jsp:setProperty name="connection" property="timeout" value="33">
/jsp:useBean>
```

在下面的这个例子中，这个JavaBeans对象具有会话期作用域，并且在当前会话中已经存在了。

```
<jsp:useBean id="wombat" type="my.WombatType" scope="session"/>
```

如果这个对象不存在的话，将抛出一个 ClassCastException异常。

2. <jsp:setProperty>

<jsp:setProperty>动作用于向一个 JavaBean的属性赋值，需要注意的是，在这个动作中将会使用到的name属性的值将是一个前面已经使用 <jsp:useBean>动作引入的JavaBean的名字。

表4-6说明了在使用<jsp:setProperty>时的类型转换，不过在客户端请求时使用<jsp:setProperty>设定JavaBean的属性可以使用任何类型，JSP文件的执行中也不会自动地进行类型转换。

表 4-6

属性类型	由String类型转换所使用的方法
boolean Boolean	java.lang.Boolean.valueOf(String)
byte Byte	java.lang.Byte.valueOf(String)
int Integer	java.lang.Integer.valueOf(String)
char Character	java.lang.Character.valueOf(String)

（续）

属性类型	由String类型转换所使用的方法
double	java.lang.Double.valueOf(String)
Double	
float	java.lang.Float.valueOf(String)
Float	
long	java.lang.Long.valueOf(String)
Long	

<jsp:setPropety>的语法如下：

```
<jsp:setProperty name=" beanName" prop_expr />
prop_expr ::=  property="*" |
               property=" propertyName"|
               property=" propertyName" param=" parameterName"|
               property=" propertyName" value=" propertyValue"
               propertyValue ::= string
```

表4-7是属性及其解释

表 4-7

属性名	描述
property	此属性表明了需要设定值的JavaBrean属性的名称。 这里有一个很有意思的特殊的property设定：当一个property设定为“*”时，JSP解释器将把系统ServletRequest对象中的参数一个一个的列举出来，检查这个 JavaBean的属性是否和ServketRequest对象中的参数有相同的名称。如果有，就自动将 ServletRequest对象中参数值传递给相应的JavaBean属性
param	这个属性表明了在由系统的Request向JavaBean传递参数时具体采用哪一个Request。具体到Web页面，也就是哪一个Form
value	这个属性表明了需要设定给 JavaBean属性的值，可以是直接赋值，也可以是 ServletRequest对象的一个参数名

下面就<jsp:setProperty>动作举几个例子：

将ServletRequest对象request中的参数全部输入到名为 request的JavaBean中：

```
<jsp:setProperty name="request" property="*" />
```

将ServletRequest对象user中的参数usernme输入到名为 user的JavaBean中：

```
<jsp:setProperty name="user" property="user" param="username" />
```

将值“ i+1 ”计算出来后输入到名为 results的JavaBean的属性row中：

```
<jsp:setProperty name="results" property="row" value="<%= i+1 %>" />
```

3. <jsp:getProperty>

<jsp:getProperty>动作用于从一个 JavaBean中得到某个属性的值，无论原先这个属性是什么类型的，都将被转换为一个 String类型的值。

语法如下：

```
<jsp:getProperty name=" name" property=" propertyName" />
```

例如：

```
<jsp:getProperty name="user" property="name" />
```

4. <jsp:include>

<jsp:include>用于引入一个静态或动态的页面到一个JSP文件中，这动作仅仅和JspWrite对象发生关系。

<jsp:include>动作可以包含一个或几个<jsp:param>子动作用于向要引入的页面传递参数。

语法如下：

```
<jsp:include page=" urlSpec" flush="true"/>
```

或

```
<jsp:include page=" urlSpec" flush="true">
     { <jsp:param .... /> }
</jsp:include>
```

属性flush设定是否自动刷新缓冲区，实际上，在当前的JSP版本（1.1）中，flush设为false是没有任何意义的。

下面是实例：

```
<jsp:include page="/templates/copyright.html"/>
```

5. <jsp:forward>

<jsp:forward>用于引导客户端的请求到另一个页面或者是另一个Servlet去。

<jsp:forward>动作可以包含一个或几个<jsp:param>子动作，用于向要引导进入的页面传递参数。

需要注意，当<jsp:forward>动作发生的时候，如果已经有文本被写入输出流而且页面没有设置缓冲，那么将抛出一个IllegalStateException的异常。

下面是<jsp:forward>的语法：

```
<jsp:forward page=" relativeURLspec" />
```

或

```
<jsp:forward page=" urlSpec">
     { <jsp:param .... /> }*
</jsp:forward>
```

举例如下：

```
<%
   String whereTo = "/templates/"+someValue;
%>
<jsp:forward page=’<%= whereTo %>’ />
```

6. <jsp:param>

<jsp:param>实际上提供了名称与值的一种一一对应关系，在<jsp:include>、<jsp:forward>和<jsp:plugin>中常常作为子动作使用。

语法为：

```
<jsp:param name=" name" value=" value" />
```

7. <jsp:plugin>

<jsp:plugin>动作为Web开发人员提供了一种在JSP文件中嵌入客户端运行的Java程序（如：Applet、JavaBean）的方法。在JSP处理这个动作的时候，将根据客户端浏览器的不同，JSP在执行以后将分别输出为OBJECT或EMBED这两个不同的HTML之素。

下面是<jsp:plugin>的语法：

```
<jsp:plugintype="bean|applet"
      code=" objectCode"
      codebase=" objectCodebase"
      { align=" alignment" }
      { archive=" archiveList" }
      { height=" height" }
      { hspace=" hspace" }
      { jreversion=" jreversion" }
      { name=" componentName" }
      { vspace=" vspace" }
      { width=" width" }
      { nspluginurl=" url" }
      { iepluginurl=" url" } >
      { <jsp:params>
      { <jsp:param name=" paramName" value=" paramValue" /> }+
      </jsp:params> }
      { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

表4-8是对<jsp:plugin>动作的子动作和属性的详细说明：

表 4-8

子动作或属性	说 明
<jsp:param>	设定Java Applet或JavaBean执行所需要的参数
<jsp:fallback>	设定当浏览器不支持此项PlugIn时候应当显示的内容
type	指示这个对象是一个Java Applet还是一个JavaBean
jreversion	指示这个插件对象执行所需要的JRE版本，缺省情况为“1.1"
nspluginurl	指示对于Netscape Navigator的JRE插件的下载地址（URL）
iepluginurl	指示对于Internet Explorer的JRE插件的下载地址（URL）
code、codebase、align、archive、height、hspace、name、vspace、title、width	和HTML中的意义一致

第5章　作为XML的JSP

本章将介绍如何使用标准的 XML语法来书写一个 JSP页面，上一章介绍的非 XML语法在本章中都可以找到对应的 XML形式的语法格式。注意， JSP在将来的发展中无疑将越来越强调XML的语法格式，读者可以从最新的 JSP语法规范----JavaServer Pages(tm) Specification Version 1.2 - public draft 1 (PD1)，也就是公开测试版)中发现这一变化趋势。

5.1　为什么要使用XML相容的语法形式

为什么要使用 XML相容语法来构架 JSP呢？当然不是因为 XML是一个时髦的事物，而是作为XML文档的JSP将会得到许多的好处。

例如，一个标准的XML相容的JSP语法将有助于JSP的开发。

JSP文件的XML语法使得JSP文件的内容很容易被组织和管理。

可以使用XML的开发和分析工具来开发和分析 JSP，仅仅需要更换DTD文件就可以升级到最新版本的JSP。

XML格式统一的语法更容易学习和使用。

5.2　关于文本类型的语法

5.2.1　jsp:root元素

作为XML文档的JSP文件，有一个 <jsp:root>元素作为其根元素，这个根元素同时也是标签库（taglib）设置命名域的地方。在 <jsp:root>中使用xmlns属性来设置当前使用的 JSP版本（通过DTD文件）。

例如：

```
<jsp:root
      xmlns:jsp="http://java.sun.com/products/jsp/dtd/jsp_1_0.dtd">
      remainder of transformed JSP page
</jsp:root>
```

5.2.2　公共标识符

作为XML文档的JSP文件建议使用如下形式的文档类型声明：

```
<! DOCTYPE root
PUBLIC"-//Sun Microsystems Inc.//DTD JavaServer Pages Version 1.1//EN"
"http://java.sun.com/products/jsp/dtd/jspcore_1_0.dtd">
```

5.3 指令

一般的JSP指令具有如下的形式：

```
<%@ directive { attr="value" }* %>
```

可以翻译为如下的XML语法形式：

```
<jsp:directive.directive { attr="value" }* />
```

5.3.1 page指令

作为XML文档的page指令：

```
<jsp:directive.page page_directive_attr_list />
```

举例如下

一般的“ page ”指令：

```
<%@ page info="my latest JSP Example V1.1" %>
```

相应的XML相容语法：

```
<jsp:directive.page info="my latest JSP Example V1.1" />
```

5.3.2 include指令

作为XML文档的include指令：

```
<jsp:directive.include file=" relativeURLspec" flush="true|false" />
```

举例如下

一般的include指令：

```
<%@ include file="copyright.hmtl" %>
```

相应的XML相容语法：

```
<jsp:directive.include file="htmldocs/logo.html" />
```

5.3.1 taglib指令

taglib指令如前所述，是被包含在<jsp:root>元素中，以xmlns属性的形式被定义的。

5.3.1 page指示

作为XML文档的page指为：

```
<jsp:directive.page page_directive_attr_list />
```

举例如下。

一般的page指为：

```
<%@ page info="my latest JSP Example V1.1" %>
```

相应的XML相容语法：

```
<jsp:directive.page info="my latest JSP Example V1.1" />
```

5.3.2 include指示

作为XML**文档的**include**指为：**

```
<jsp:directive.include file=" relativeURLspec" flush="true|false" />
```

举例如下。

一般的include**指令：**

```
<%@ include file="copyright.hmtl" %>
```

相应的XML**相容语法：**

```
<jsp:directive.include file="htmldocs/logo.html" />
```

5.3.3 taglib指令

taglib**指令如前所述，是被包含在**<jsp:root>**元素中，以**xmlns**属性的形式被定义的。**

5.4 脚本元素

JSP**的脚本元素包含声明、脚本代码、表达式三个部分，下面分别讲述其相应的**XML**语法形式。**

5.4.1 声明

声明的XML**形式语法为：**

```
<jsp:declaration> declaration goes here </jsp:declaration>
```

举例如下：

```
<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

相应的XML**语句为：**

```
<jsp:declaration> <![CDATA[ public String f(int i) { if (i<3) return("..."); } ]]>
</jsp:declaration>
```

下面是相关的DTD**文件片断：**

```
<!ELEMENT jsp:declaration (#PCDATA) >
```

5.4.2 脚本代码

声明的XML**形式语法为：**

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

下面是相关的DTD**文件片断：**

```
<!ELEMENT jsp:scriptlet (#PCDATA) >
```

5.4.3 表达式

声明的XML**形式语法为：**

```
<jsp:expression> expression goes here </jsp:expression>
```

举例如下：

```
<%= str + i + " " + date%>
```

相应的XML语句为：

```
<jsp:expression>
    String str ="aasasda";
    int i = 5;
    java.util.Date date = new Date();
</jsp:expression>
```

下面是相关的DTD文件片断：

```
<!ELEMENT jsp:expression (#PCDATA) >
```

5.5 如何将一个普通的JSP文件转换为一个XML文档

使用下面的方法可以将一个普通的JSP文件转换为一个XML文件：

首先，在文本中加入<jsp:root>**，在这个元素中使命名前缀“** jsp **”成为文本中标准元素的前缀。**

然后，将所有的“ <% **”按照前面介绍的方法转换为** XML**的相容语法形式。**

接着，将“ taglib **”指示转化为使用**<jsp:root>**中的“** xmlns **”属性来指示。**

最后，为每一个非JSP**元素的部分建立一个**CDATA**元素。**

利用表5-1**可以快速地将一个**JSP**文件转化成一个**XML**文件。**

表 5-1

JSP 元 素	XML相应元素
<%@ page ... %>	<jsp:directive.page ... />
<%@ taglib ... %>	**使用**<jsp:root>**设定**
<%@ include ... %>	<jsp:directive.include .../>
<%! ... %>	<jsp:scriptlet> .... </jsp:scriptlet>
<% ... %>	<jsp:scriptlet> .... </jsp:scriptlet>
<%= .... %>	<jsp:expression> .... </jsp:expression>

5.6 JSP1.1的DTD文件

这里是JSP1.1**的**DTD**文件，从中可以对**JSP**的语法有一个完整的理解。**

```
<!ENTITY % jsp.body "
(#PCDATA
|jsp:directive.page
|jsp:directive.include
|jsp:scriptlet
|jsp:declaration
|jsp:expression
|jsp:include
```

```
|jsp:forward
|jsp:useBean
|jsp:setProperty
|jsp:getProperty
|jsp:plugin
|jsp:fallback
|jsp:params
|jsp:param)*
">
<!ELEMENT jsp:useBean %jsp.body;>
<!ATTLIST jsp:useBean
id ID #REQUIRED
class CDATA#REQUIRED
scope (page|session|request|application) "page">
<!ELEMENT jsp:setProperty EMPTY>
<!ATTLIST jsp:setProperty
name IDREF#REQUIRED
propertyCDATA#REQUIRED
value CDATA#IMPLIED
param CDATA#IMPLIED>
<!ELEMENT jsp:getProperty EMPTY>
<!ATTLIST jsp:getProperty
name IREF #REQUIRED
propertyCDATA#REQUIRED>
<!ELEMENTjsp:includeEMPTY>
<!ATTLISTjsp:include
flush (true|false)"false"
page CDATA#REQUIRED>
<!ELEMENT jsp:forward EMPTY>
<!ATTLISTjsp:forward
page CDATA#REQUIRED>
<!ELEMENT jsp:scriptlet (#PCDATA)>
<!ELEMENT jsp:declaration (#PCDATA)>
<!ELEMENT jsp:expression (#PCDATA)>
<!ELEMENT jsp:directive.page EMPTY>
<!ATTLIST jsp:directive.page
languageCDATA"java"
extendsCDATA#IMPLIED
contentTypeCDATA"text/html; ISO-8859-1"
import CDATA#IMPLIED
session(true|false)"true"
buffer CDATA"8kb"
autoFlush(true|false)"true"
isThreadSafe(true|false)"true"
info CDATA#IMPLIED
errorPageCDATA#IMPLIED
isErrorPage(true|false)"false">
```

```
<!ELEMENT jsp:directive.include EMPTY>
<!ATTLIST jsp:directive.include
file CDATA #REQUIRED>
<!ELEMENT jsp:root %jsp.body;>
<!ATTLIST jsp:root
xmlns:jspCDATA#FIXED "http://java.sun.com/products/jsp/dtd/
jsp_1_0.dtd">
```