J2EE 简介

J2EE 提供了一套设计、开发、汇编和部署企业应用程序的规范；提供了企业级应用程序的开发平台，提供了分布式、基于组件、松耦合、安全可靠、独立于平台的应用程序环境；提供了开发企业级应用程序的技术架构

J2EE 客户端 ——Web 客户端； Applets（客户端小应用程序 ）；Application 客户端

Web 客户端 由两部分组成：

(1)由运行在 Web层的 Web 组件生成的包含各种标记语言 (HTML 、XML 等等) 的动态 Web 页面〔①交互性、②自动更新、③因时因人而变〕 **(2)**接收从服务器传送来的页面并将它显示出来的 Web 浏览器

Web 组件

既可以是 servlet 也可以是 JSP 页面； Servlets 是一个 Java 类，它可以动态地处理请求并作出响应； JSP 页面是一个基于文本的文档，它以 servlet 的方式执行，但是它可以更方便建立静态内容 ；静态的 HTML 页面、 applet、服务器端的功能类并不被 J2EE 规范视为 Web 组件

Business 组件

由运行在业务层的 enterprise beans(EJB)处理；有三种类型的 enterprise beans(EJB):会话 beans, 实体 beans, and 消息驱动 beans

J2EE 容器

容器为 J2EE 应用程序组件提供了运行时支持；容器充当组件与支持组件的底层特定于平台的功能之间的接口；

J2EE 服务器以容器的形式为每一个组件类型提供底层服务（如事务处理、状态管理、多线程、资源池等

容器服务

1）容器是一个组件和支持组件的底层平台特定功能之间的接口， 在一个 Web 组件、enterprise bean 或者是一个应用程序客户端组件可以被执行前，它们必须被装配到一个 J2EE 应用程序中，并且部署到它们的容器 **2**）装配的过程包括为 J2EE 应用程序中的每一个组件以及 J2EE 应用程序本身指定容器的设置。容器设置定制了由 J2EE 服务器提供的底层支持，这将包括诸如安全性、事务管理、 Java 命名目录接口 (JNDI) 搜寻以及远程程序连接 **3**）J2EE 的安全性模式可以让你对一个 Web 组件或 enterprise bean 进行配置以使得只有授权用户访问系统资源 **4**）J2EE 的事务模式可以让你指定方法之间的关系以组成一个单个的事务， 这样在一个事务中的所有方法将被视为一个单一的整体 **5**）JNDI 搜寻服务为企业中的多种命名目录服务提供一个统一的接口， 这使得应用程序组件可以访问命名目录服务 **6**）J2EE 远程连接模式管理客户端和 enterprise bean 之间的底层通信。在一个 enterprise bean 被建立后，客户端在调用其中的方法时就象这个 enterprise bean 就运行在同一个虚拟机上一样 **7**）J2EE 体系结构提供了可配置的服务意味着在相同的 J2EE 应用程序中的应用程序组件根据其被部署在什么在地方在实际运行时会有所不同 **8**）容器还管理诸如一个 enterprise bean 和 servlet 的生存周期、数据库连接资源池等不能配置的服务。

J2EE 核心技术

1）Enterprise JavaBeans(EJB）技术 ——企业 Bean 包含有业务逻辑代码。代码包含实现业务逻辑的方法和字段 **2**）Java Servlet 技术— Servlet 是驻留在服务器上 Java 类，用于响应通过 HTTP 传入的请求 **3**）JSP 技术 Java 服务器页面允许程序员将 Servlet 代码写入基于文本的文档中。这些页面与 HTML 页面类似，只是它们还含有 Java 代码 **4**）JDBC API——DBC API 有助于从使用 Java 编程语言编写的方法之中执行 SQL 命令。当默认容器管理持续性被覆盖时， 或者在使用会话 bean 访问数据库时， 可在企业 bean 中使用 JDBC API5）JNDI 技术

Web 应用程序

定义： Web 应用程序是 servlets,jsp 页面 HTML 页面，类，和其他资源等的集合

缺省情形下， Web 应用程序的实例必须运行在一个 JVM（java virtual machine）中；以两种方式存在于文件系统：一是 web 归档文件， 以.war 扩展名结尾； 一是 web 归档文件展开后的目录结构； 元素构成 :Servlets、JSP 页面 、帮助类、静态文档 (HTML, images, sounds, etc.)、客户端的 Java applets, beans, and classes 等以及把上述元素组合起来的描述性的元信息

由下列三部分组成：

第一部分是协议（或称为服务方式） ；第二部分是存有该资源的主机 IP 地址（有时也包括端口号） ； 第三部分是主机资源的具体地址，如目录和文件名等。

第一章 **HTTP** 协议

HTTP 请求包由三个部分构成， 分别是：方法-URI-协议/版本，请求头，请求正文；HTTP 应答包由三个部分构成，分别是：协议-状态代码-描述，应答头，应答正文 ;**HTTP 1.1** 支持七种请求方法 :GET、POST、HEAD、OPTIONS、PUT、DELETE 和 TRACE 等

URI 由三部分组成： 1. 访问资源的命名机制 2. 存放资源的主机名 3. 资源自身的名称，由路径表示

URL 由下列三部分组成：

第一部分是协议（或称为服务方式） ；第二部分是存有该资源的主机 IP 地址（有时也包括端口号） ；第三部分是主机资源的具体地址，如目录和文件名等。

第一部分和第二部分之间用“ : // ”符号隔开，第二部分和第三部分用“ / ”符号隔开。第一部分和第二部分是不可缺少的，第三部分有时可以省略。语法如下： http_URL＝http://host [“ : ” port] [path]

第二章 **Servlet** 技术

Java Servlets是基于 Java技术的 Web 组件，用来扩展以请求 /响应为模型的服务器的能力，提供动态内容。

Servlet

是：使用 Java Servlet 应用程序设计接口（ API）及相关类和方法的 Java 程序。

Servlet 由容器或引擎来管理，通过请求 /响应模型与 Web客户进行交互

Servlet 的特性和优势

可移植性 (Portability)、强大的功能、安全 (提供不同层次的安全保障)、简洁（ Servlet 代码面向对象，在封装方面具有先天的优势 ）、集成（ Servlet 和服务器紧密集成，它们可以密切合作完成特定的任务）

servlet容器

servlet 由 servlet 容器管理， servlet 容器也叫 servlet 引擎，是 servlet 的运行环境

servlet 容器是 web 服务器或应用服务器的一部分，管理和维护 servlet 的整个生命周期

必须支持 http 协议，负责处理客户请求、把请求传送给适当的 servlet 并把结果返回给客户。

虽然容器的实现可能各不相同，但容器与 servlet之间的接口是由 servlet API 定义好的。

servlet接口

定义在 javax.servlet.Servlet

当在编写某个 servlet 的时必须直接或间接的实现这个接口。一般趋向于间接实现：通过从 javax.servlet.GenericServlet或 javax.servlet.http.HttpServlet派生。

在实现 **servlet**接口时必须实现它的五个方法

1）public void init(ServletConfig config) throws ServletException 一旦对 servlet 实例化后， 容器就调用此方法。 容器把一个 ServletConfig 对象传统给此方法， 这样 servlet 的实例就可以把与容器相关的配置数据保存起来供以后使用

2）Publicvoid service(ServletRequestreq,ServletResponseres)throwsServletException,IOException只有成功初始化后此方法才能被调用处理用户请求。 前一个参数提供访问初始请求数据的方法和字段， 后一个提供 servlet 构造响应的方法 **3**）public void destroy()容器可以在任何时候终止 servlet 服务 **4**）public ServletConfig getServletConfig()在 servlet 初始化时，容器传递进来一个 ServletConfig 对象并保存在 servlet 实例中，该对象允许访问两项内容：初始化参数和 ServletContext 对象 **5**）public String getServletInfo()此方法返回一个 String 对象，该对象包含 servlet 的信息

请求处理方法（ **Service)**

1）Serlvet 接口只定义了一个服务方法就是 service，而 HttpServlet 类实现了该方法并且要求调用对应的 doXXX()方法。

2）通常情况下，在开发基于 HTTP 的 servlet时只需要关心 doGet和 doPost方法，其它的方法需要开发者非常的熟悉 HTTP 编程。

3）通常情况下，实现的 servlet 都是从 HttpServlet 扩展而来。 doPut 和 doDelete 方法允许开发者支持 HTTP/1.1 的对应特性； doHead 是一个已经实现的方法，它将执行 doGet 但是仅仅向客户端返回 doGet 应该向客户端返回的头部的内容； doOptions 方法自动的返回 servlet 所直接支持的 HTTP 方法信息； doTrace方法返回 TRACE 请求中的所有头部信息。

4）对于仅支持 HTTP/1.0 的容器而言，只有 doGet, doHead 和 doPost方法能被使用，因为 HTTP/1.0 协议没有定

义 PUT, DELETE, OPTIONS,或者 TRACE 请求。

5）ervlet 规范要求， servlet 容器至少要实现 HTTP/1.0 协议规范，推荐实现 HTTP/1.1 规范，在此基础上可以实现其它的基于请求回应模式（ based request response model）的协议（例如 HTTPS）。

Action 1:加载和实例化

1）容器负责加载和实例化一个 servlet。实例化和加载可以发生在引擎启动的时候， 也可以推迟到容器需要该 servlet 为客户请求服务的时候（加载的时机） 。

2） 容器必须先定位 servlet 类，在必要的情况下，容器使用通常的 Java 类加载工具加载该 servlet，可能是从本机文件系统，也可以是从远程文件系统甚至其它的网络服务。

3）容器加载 servlet 类以后，它会实例化该类的一个实例。 需要注意的是可能会实例化多个实例， 例如一个 servlet 类因为有不同的初始参数而有多个定义，或者 servlet 实现 SingleThreadModel 而导致容器为之生成一个实例池。

Action 2: 初始化

1） 容器必须在 servlet 能够处理客户端请求前初始化它。

2） 初始化的目的：是读取永久的配置信息，昂贵资源（例如 JDBC 连接）及其它仅需执行一次的任务。

3） 通过调用它的 init 方法并给它传递唯一的一个（每个 servlet 定义一个） ServletConfig 对象完成这个过程。

4） 该配置对象允许 servlet 访问容器的配置信息中的名称－值对（ name-value）初始化参数。同时给 servlet 提供了访问实现了 ServletContext 接口的具体对象的方法，该对象描述了 servlet 的运行环境。

Action 3:处理请求

在 servlet 被适当地初始化后，容器就可以使用它去处理请求了。

每一个请求由 ServletRequest 类型的对象代表，而 servlet 使用 ServletResponse 回应该请求。这些对象被作为 service 方法的参数传递给 servlet。

在 HTTP 请求的情况下，容器必须提供代表请求和回应的 HttpServletRequest 和 HttpServletResponse 的具体实现。

需要注意的是容器可能会创建一个 servlet 实例并将之放入等待服务的状态， 但是这个实例在它的生存期中可能根本没有处理过任何请求。

处理请求中的异常

可能抛出 ServletException 或者 UnavailableException 异常。

ServletException 表明在处理请求的过程中发生了错误， 容器应该使用合适的方法清除该请求。

UnavailableException 表明 servlet 不能对请求进行处理，可能是暂时的或永久的。

Action 4:服务结束

Servlet 规范并没有对容器将一个加载的 servlet 保存多长时间作出要求，因此一个 servlet 实例可能只在容器中存活了几毫秒，当然也可能是其它更长的任意时间（但是肯定会短于容器的生存期）

当容器决定将之移除时（原因可能是保存内存资源或者自己被关闭） ，那么它必须允许 servlet 释放它正在使用的任何资源并保存任何永久状态（这个过程通过调用 destroy 方法达到）。

容器在能够调用 destroy 方法前，它必须允许那些正在 service 方法中执行的线程执行完或者在服务器定义的一段时间内执行（这个时间段在容器调用 destroy 之前）。

一旦 destroy 方法被调用，容器就不会再向该实例发送任何请求。如果容器需要再使用该 servlet，它必须创建新的实例。 destroy 方法完成后，容器必须释放 servlet 实例以便它能够被垃圾回收。

部署描述符中 **servlet**相关部分—— **<servlet> </servlet>**元素； **<servlet-mapping> </servlet-mapping>**元素

<servlet> 元素

servlet 元素必须含有 servlet-name元素和 servlet-class元素， 或者 servlet-name元素和 jsp-file 元素。servlet-name 元素用来定义 servlet 的名称，该名称在整个应用中必须是惟一的。 servlet-class元素用来指定 servlet 的完全限定的类名称。 jsp-file 元素用来指定应用中 JSP 文件的完整路径。这个完整路径必须由 /开始。 init-param 元素是可选元素，有 param-name, param-value两个子元素。

<servlet-mapping> 元素

<servlet-mapping>元素为一个 servlet 实例提供一个 URL pattern；必须包含 <servlet-name>元素和 <url-pattern>元素；必须和在 web.xml 文件某处 <servlet>元素定义的 <servlet-name>元素一致。

Servlet过滤器

1）Servlet 过滤器是在 Java Servlet规范 2.3 中定义的，它能够对 Servlet 容器的请求和响应对象进行检查和修改。

2） 过滤器本身并不产生请求和响应对象，它只能提供过滤作用。 Servlet 过滤能够在 Servlet 被调用之前检查 Request 对象，修改 Request Header 和 Request 内容；在 Servlet 被调用之后检查 Response 对象，修改 Response Header和 Response内容。

3）过滤器负责过滤的 Web组件可以是 Servlet、JSP或者 HTML 文件，即动态或静态的 web 资源。

Servlet 过滤器的特点

Servlet 过滤器可以检查和修改 ServletRequest和 ServletResponse对象；可以被指定和特定的 URL 关联，只有当客户请求访问该 URL 时，才会触发过滤器； 可以被串联在一起， 形成管道效应， 协同修改请求和响应对象 （过滤器链）

Servlet 过滤器的作用

查询请求并作出相应的行动；阻塞请求 -响应对， 使其不能进一步传递；修改请求的头部和数据； 用户可以提供自定义的请求；修改响应的头部和数据；与外部资源进行交互。

过滤器编程步骤

1）建立一个实现 Filter 接口的类。

所有的 Servlet 过滤器类都必须实现 javax.servlet.Filter 接口。

这个接口含有 3 个过滤器类必须实现的方法： init(，)、doFilter(，)、destroy()

2）在 doFilter 方法中实现过滤行为。

doFilter 方法为大多数过滤器的关键部分。 每当调用一个过滤器时， 都要执行 doFilter。对于大多数过滤器来说，doFilter 执行的步骤是基于传入信息的。因此，可能要利用作为 doFilter 的第一个参数提供的 ServletRequest 这个对象常常构造为 HttpServletRequest类型，以提供对该类的更特殊方法的访问。

3）调用 FilterChain 对象的 doFilter 方法。

Filter 接口的 doFilter 方法以一个 FilterChain 对象作为它的第三个参数。在调用该对象的 doFilter 方法时，激活下一个相关的过滤器。这个过程一般持续到链中最后一个过滤器为止。在最后一个过滤器调用其 FilterChain 对象的 doFilter 方法时，激活 servlet 或页面自身。但是，链中的任意过滤器都可以通过不调用其 FilterChain 的 doFilter 方法中断这个过程。 在这样的情况下， 不再调用 JSP页面的 serlvet，并且中断此调用过程的过滤器负责将输出提供给客户机。

4）对相应的 servlet和 JSP页面注册过滤器。

部署描述符文件的 2.3 版本引入了两个用于过滤器的元素，分别是： filter 和 filter-mapping。

filter 元素向系统注册一个过滤对象

filter-mapping 元素指定该过滤对象所应用的 URL。

5）禁用激活器 servlet。

Servlet 过滤器对请求的过滤

A． Servlet 容器创建一个过滤器实例

B． 过滤器实例调用 init 方法，读取过滤器的初始化参数

C． 过滤器实例调用 doFilter 方法，根据初始化参数的值判断该请求是否合法

D． 如果该请求不合法则阻塞该请求

E． 如果该请求合法则调用 chain.doFilter 方法将该请求向后续传递

Servlet 过滤器对响应的过滤

A． 过滤器截获客户端的请求

B． 重新封装 ServletResponse，在封装后的 ServletResponse中提供用户自定义的输出流

C． 将请求向后续传递

D． Web 组件产生响应

E． 从封装后的 ServletResponse中获取用户自定义的输出流

F． 将响应内容通过用户自定义的输出流写入到缓冲流

G． 在缓冲流中修改响应的内容后清空缓冲流，输出响应内容

Servlet 过滤器使用的注意事项

A．由于 Filter、FilterConfig 、FilterChain 都是位于 javax.servlet 包下，并非 HTTP 包所特有的，所以其中所用到的请求、响应对象 ServletRequest、ServletResponse 在使用前都必须先转换成 HttpServletRequest、HttpServletResponse再进行下一步操作。

B．在 web.xml 中配置 Servlet 和 Servlet 过滤器，应该先声明过滤器元素，再声明 Servlet 元素

会话---会话和会话管理、会话追踪机制、会话编程

会话是指一个用户在客户端登录，为达到某个目的与服务器端进行多次交互，最后退出应用系统的全过程。

会话管理 是对用户会话全过程中涉及的会话状态保存与恢复、会话信息的采集与管理的一系列活动。

会话机制 是：**1**）一个 web 容器必须支持 HTTP 协议 **2**）会话 HTTP 是一种无状态的协议

3）需要一种服务器端的机制，维持会话状态信息 **4**）服务器使用一种类似于散列表的结构来保存信息。

常用的会话追踪机制有： cookies、SSL SessionsURL、重写、表单隐藏

Servlet 的会话追踪机制 是基于 Cookie 或 URL 重写技术，融合了这两种技术的优点。

当客户端允许使用 Cookie 时，内建 session对象使用 Cookie 进行会话追踪； 如果客户端禁用 Cookie，则选择使用 URL 重写

创建会话 —— HttpSession session=request.getSession(true)

在会话中保存数据的方法 setAttribute(String s, Object o)

从会话提取原来所保存对象的方法 getAttribute(String s)

关闭会话 HttpSession.invalidate()

Session(会话**)**的常用方法

1、 getAttribute()：从 session中获取以前存储的值

2、 getAttributeNames()：返回 session中所有属性的名称

3、 setAttribute()：将键与值关联起来，存储进 session

4、 removeAttribute()：删除 session中存储的对应键的值

5、 invalidate()：删除整个 session及其存储的键值

6、 logout()：注销当前用户

7、 getId()：获取每个 session对应的唯一 ID

8、 getCreationTime()：获取 session创建的时间

9、 getLastAccessedTime(): 获取 session最后被访问的时间

10、getMaxInactiveInterval() ：在用户没有访问的情况下，会话在被自动废弃之前应该保持多长时间

URL 路径**:** Servlet－mapping 元素有子元素 servlet-name, url-pattern，目的是以指定的任何 url 形式去关联（访问）servlet。同一个 servlet指定多个不同的 url，四种不同的 url 形式：

1）以/开始/*结束的形式。按纯目录形式关联 servlet。

2）以*.postfix 形式。按具体文件类型关联 servlet。

3）单个字符串形式。精确匹配关联 servlet。

4）单个/形式。缺省 servlet（所有没精确指定资源的 url 都由该 servlet 处理）。

URL 路径解析： 通常由 servlet 容器来解析一个指向 servlet 的 URL 。解析分两步， 1.标识网络应用； 2.定位具体的 servlet。上述两个步骤都会对 URL 中的 URI（除去主机名以外的） 部分进行分段处理， 形成三个部分， Context Path, Servlet Path, Path Info。HttpServletRequest提供了三个方法－ getContextPath(),getServletPath()和 getPathInfo()分别提取不同段内容。

匹配规则

servlet容器先将整个 URI（除去 context path 之后的）和 servlet mapping进行匹配。 如果匹配成功， 则除 context path 以外的剩余部分都是 servlet path。因此， path info 部分为空。

以/为分界符倒着往前去和 servlet mapping匹配。如匹配成功，匹配的部分就是 servlet path，剩余部分是 path info。

如果 URI 最后是某种文件扩展名， 则 servlet 容器去和 servlet mapping匹配。如成功，则将整个 URI（除去 context path 之后的）视为 servlet path，而 path info 为空。

如果始终没找到相匹配的 servlet mapping，则将请求发往默认 servlet。如果不存在默认 servlet，则 servlet 容器发送错误消息，指示 servlet 没找到 .

第三章 JSP

JSP 是一种建立在 Servlet 规范提供的功能之上的动态网页技术 ,在网页文件中通过 JSP 页面元素嵌入可在服务器端执行的 java 代码。

JSP 的优点 ----将内容与表示分离，强调可重用组件，简化页面开发－ Web 设计人员和 Web 程序员使用 Web 开发工具开发 JSP 页面

JSP 与 **ASP** 之比较

相似性 ：两者都提供在 HTML 代码中混合某种程序代码、由语言引擎解释执行程序代码的能力。在 ASP 或 JSP 环境下， HTML 代码主要负责描述信息的显示样式，而程序代码则用来描述处理逻辑。普通的 HTML 页面只依赖于 Web服务器，而 ASP 和 JSP页面需要附加的语言引擎分析和执行程序代码。程序代码的执行结果被重新嵌入到 HTML 代码中，然后一起发送给浏览器。 ASP 和 JSP都是面向 Web服务器的技术，客户端浏览器不需要任何附加的软件支持。

差异性 :ASP 的编程语言是 VBScript 之类的脚本语言， JSP使用的是 Java, 这是两者最明显的区别。 此外， ASP 与 JSP 还有一个更为本质的区别： 两种语言引擎用完全不同的方式处理页面中嵌入的程序代码。 在 ASP 下，VBScript 代码被 ASP 引擎解释执行；在 JSP下，代码被编译成 Servlet 并由 Java虚拟机执行，这种编译操作仅在对 JSP 页面的第一次请求时发生。

JSP 页面构成 —— 元素（脚本元素、指令元素、动作元素） 、 模板数据 、 注释

脚本元素 （声明 +脚本段 +表达式）

声明:用于定义在其它脚本元素中可以使用的变量、方法或类（以 <%! 开始，以 %> 结束）

脚本段 :<% Java代码块 %> 在一个脚本段定义的变量为局部变量

表达式 :java 语言中完整的表达式 <%= Java表达式 %>

指令元素 （page指令 include 指令 taglib 指令）

语法 <%@ 指令名称 属性 1="属性值 1" 属性 2="属性值 2" ， 属性 n="属性值 n"%>

page 指令用于设置 JSP页面的属性 language、 import、 isErrorPage、 errorPage、 buffer

include 指令 用于在 jsp 页面中静态包含一个文件， 该文件可以是 jsp 页面, HTML 页面, 文本文件。使用了 include 指令的 jsp 页面在转换时， jsp 容器会在其中插入所包含文件的文本或代码。 include 指令语法 <%@ include file = ”文件名” %>

taglib 指令 的作用是将标签库描述符文件引入到该 JSP页面中，并设置前缀，而去利用标签 的 前 缀 去 使 用 标签库表述文件中的标签，语法 <%@ taglib uri = ”标签库表述符文件” prefix = ”前缀名” %>

动作元素 （<jsp:useBean>、 <jsp:setProperty>、<jsp:getProperty>、<jsp:forward>、<jsp:include>）

<jsp:usebean> 动作 用于实例化 JavaBean, 或者定位一个已经存在的 JavaBean实例，并把实例的引用赋给一个变量

<jsp:setProperty> 动作和<jsp:useBean>一起使用，使用 Bean 中的 setXXX() 方法设置 JavaBean的简单属性和索引属性。 name实例名、 property 被设置的属性的名字、 param 指定请求对象中参数的名字、 value 赋给 Bean 属性的值。在 <jsp:setProperty>元素中，不能同时出现 param和 value 属性

<jsp:getProperty> 动作 用来访问一个 Bean的属性，并把属性的值转化成一个 String，然后发送到输出流中。如果属性是一个对象，将调用该对象的 toString()方法

<jsp:forward> 动作 ：允许在运行时将当前的请求转发给一个静态的资源 （jsp 页面或者 Servlet），请求被转向到的资源必须位于同 jsp 发送请求相同的上下文环境中

<jsp:include> 动作 用于在当前页面中包含静态和动态的资源 ,一旦被包含的页面执行完毕 ,请求处理将在调用页面中继续执行 .<jsp:include page="OtherPage.jsp" flush="true"/>

<jsp:include> 动作 是动态包含、 include 指令是静态包含

注释

JSP 注释 ：＜%-- comment --%＞ ， 也称为 “ 隐藏注释 ” 。JSP 引擎将忽略它。标记内的所有 JSP脚本元素、指令和动作都将不起作用。

HTML 注释：＜!-- comment --＞，也称为“输出的注释”，直接出现在结果 HTML 文档中。标记内的所有 JSP 脚本元素、指令和动作正常执行。

JSP 执行过程

转译：从 JSP 元素中提取数据 编译：为 JSP 生成一个 Servlet 执行阶段：Servlet 容器加载转换后的 Servlet 类

JSP 隐式对象（Web 容器加载的一组类）

输入／输出对象：控制页面的输入和输出、访问与所有请求和响应有关的数据，包括 request、response 和 out

作用域通信对象（session、application、pageContext）：对象的作用域包括 page 范围、reques 范围、session 范围、application 范围

Servlet 对象（page、config）提供了访问 Servlet 信息的方法和变量

错误对象：exception 处理 JSP 页面中的错误、访问执行 JSP 的过程中引发的异常

第四章 **JavaBean** 在 **JSP** 中的应用

JavaBean 是用 JAVA 语言描述的软件组件模型，遵循某些特定约定的 Java 类。分为可视化组件和非可视化组件，没有 GUI 表现形式，可以实现代码的可重用性，可移植性强。

编程规范：**1)** 需要特定的命名空间（默认包是不允许的） **2)** 必须具备一个无参数（空）的构造函数 **3)** 不应该有 public 的类属性（字段）希望遵循这项准则，使用存取方法而非允许对类的属性直接访问 **4)** 应该通过 getXxx 和 setXxx 方法来访问类的属性。如果类有 getTitle 方法，返回 String，表明类有一个名为 title 的 String 属性，布尔型的属性使用 isXxx，而非 getXxx。

无参数构造函数的实现：<jsp:useBean id="beanName" class="package.Class" />

为何应用存取方法 ,而非 public 字段 ?**1)** 可以对值加以约束 **2)**可以在不改变接口的情况下修改内部的表达 **3)**可以执行任意的边界效应

Scope（作用域）属性的取值

scope="page"默认值。在处理当前请求的过程中，bean 对象都应该存放在 PageContext 对象中。让同一 servlet 中的其他方法可以访问该 bean。

scope="request"处理当前请求的过程中，bean 对象应存储在 ServletRequest 对象中，可以通过 getAttribute 访问到它。

scope="session" bean 会被存储在与当前请求关联的 HttpSession 中，和普通的会话对象一样，在常规 servlet 中可以使用 getAttribute 和 setAttribute 访问到它们。

scope="application" bean 将存储在 ServletContext 中（通过 application 变量或调用 getServletContext()来访问）。ServletContext 由同一 Web 应用中的所有 servlet 共享（或服务器上的所有 servlet——在没有另外定义 Web 应用的情况下）。

例：使用 **StringBean** 的 **JSP** 页面**(**代码**)**

```
<jsp:useBean id="stringBean" class=" cn.edu.njust.StringBean" />
<LI>Initial value (from jsp:getProperty):
    <I><jsp:getProperty name="stringBean" property="message" /></I>
<LI>Initial value (from JSP expression):
    <I><%= stringBean.getMessage() %></I>
<LI><jsp:setProperty name="stringBean"  property="message" value="Best string bean: Fortex" />
    Value after setting property with jsp:setProperty:
    <I><jsp:getProperty name="stringBean" property="message" /></I>
<LI><% stringBean.setMessage("My favorite: Kentucky Wonder"); %>
    Value after setting property with scriptlet:
    <I><%= stringBean.getMessage() %></I>
```

第五章 **JDBC** 概述

JDBC 本身是个商标名而不是一个缩写字“Java Database Connectivity”是一种用于执行 SQL 语句的 Java API，

由一组类和接口组成。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。采用 JDBC，不需要针对各种关系数据库编写不同的数据库程序

JDBC 体系结构 ——JDBC API、驱动程序管理器（ Driver Manager）、数据库驱动

JDBC 数据库驱动程序类型

1）JDBC-ODBC Bridge 桥接器驱动程序之一。特色是必须在使用者端的计算机上事先安装好 ODBC 驱动程序，然后通过 JDBC-ODBC 的调用方法，进而通过 ODBC 来存取数据库；

Application--->JDBC-ODBC ；Bridge---->JDBC-ODBC ；Library--->ODBC Driver-->Database

适用于快速的原型系统，没有提供 JDBC 驱动的数据库如 Access

2）JDBC-Native API Bridge 桥接器驱动程序之一。这类驱动程序也必须先在使用者计算机上先安装好特定的驱动程序（类似 ODBC），然后通过 JDBC-Native API 桥接器的转换，把 Java API 调用转换成特定驱动程序的调用方法，进而存取数据库。利用开发商提供的本地库直接与数据库通信。 Application--->JDBC Driver---->Native Database library---->Database

比 A 类性能略好。

3）JDBC-middleware 好处：省去了在使用者计算机上安装任何驱动程序的麻烦，只需在服务器端安装好 middleware，而 middleware 会负责所有存取数据库必要的转换。Application--->Jdbc Driver----->java middleware--->JDBC Driver---->Database 具有最大的灵活性，通常由那些非数据库厂商提供。

4）Pure JDBC driver

这类型的驱动程序是最成熟的 JDBC 驱动程序，不但无需在使用者计算机上安装任何额外的驱动程序，也不需要在服务器端安装任何中介程序(middleware)，所有存取数据库的操作，都直接由驱动程序来完成。Application--->Jdbc driver----->database engine--->database 最高的性能，通过自己的本地协议直接与数据库引擎通信，具备在 Internet 装配的能力。

JDBC API 实现了四个基本的功能：建立与数据的连接、执行 SQL 语句和处理执行结果、关闭数据库的连接。

Driver 接口（**JDBC** 驱动程序必须实现 ）

URL 语法 jdbc:<subprotocol>:<subname> 。<subprotocol>子协议用来标识一种特定种类的数据库连接机制 <subname> 的内容依赖于子协议 ,推荐的语法是：//hostname:port/subsubname

DriverManager 类（负责管理 **JDBC** 驱动程序）

使用 JDBC 驱动程序之前，必须先将驱动程序加载并向 DriverManager 注册后才可以使用，同时提供方法来建立与数据库的连接。通过指定的 URL 查找合适的驱动

DriverManager 关键方法 :registerDriver ;getConnection(String url) ;getConnection(String url, java.util.Properties prop

使用 **DriverManager** 的例子

```
String className,url,uid,pwd;
className = “ com.microsoft.jdbc.sqlserver.SQLServerDriver”;
url =
“ jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=testJDC;User=user;Password=password; // the
hard-code URL
 uid = ” sa“ ;
pwd = “ sa”;
Class.forName(className); // load the driver
Connection cn = DriverManager.getConnection(url);
//Connection cn = DriverManager.getConnection(url,uid,pwd);
```

DataSource接口

DataSource 对象表示一个数据源，并提供了该数据源的连接；使用 JNDI API，指定逻辑名来访问 DataSource 对象，把逻辑名映射到相应的数据源

使用 **DataSource**的例子

```
String jndi = "jdbc/jpetstore1";
```

```
Context ctx = (Context) new      InitialContext().lookup("java:comp/env");
DataSource ds = (DataSource) ctx.lookup(jndi);
Connection cn = ds.getConnection();
```

Connection类

Connection 对象代表与数据库的连接。 建立到底层数据源的连接有两种方式 :DriverManager、DataSource; 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。

Statement类

Statement类提供的方法，可以利用标准的 SQL 命令，对数据库直接新增、删除或修改操作
实现这种功能的类有三个： Statement、PreparedStatement、CallableStatement

PreparedStatement类

PreparedStatement是 Statement 的子类，在创建 PreparedStatement时，会提供一个 SQL 语句 。
区别在于： PreparedStatement类对象会将传入的 SQL 命令事先编好等待使用，当有单一的 SQL 指令比多次执行时，用 PreparedStatement类会比 Statement类有效率

CallableStatement 类：Statement 的子类为所有的 DBMS 提供了一种以标准形式调用储存过程的方法。储存过程储存在数据库中。 这种调用是用一种换码语法来写的， 有两种形式： 一种形式带结果参， 另一种形式不带结果参数。结果参数是一种输出 (OUT) 参数，是已储存过程的返回值。两种形式都可带有数量可变的输入（ IN 参数）、输出（OUT 参数）或输入和输出（ INOUT 参数）的参数。问号将用作参数的占位符。

ResultSet 类 ：负责存储查询数据库的结果。并提供一系列的方法对数据库进行新增、删除和修改操作。也负责维护一个记录光标（ Cursor），记录光标指向数据表中的某个记录，通过适当的移动记录光标，可以随心所欲的存取数据库，加强程序的效率。

光标移动 ：光标指向 ResultSet对象的当前行；当 ResultSet对象被首次创建时 ，光标位于第一行之前 .如下方式用于操作光标 :next(), **previous(),first(), last(), beforeFirst(), afterLast(), relative(int rows), absolute(int row)**

一个完整的例子

```
package cn.edu.njust;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.DatabaseMetaData;

public class DataConn
{
      public DataConn() {  }
          public static void main(String[] args) {
    try{
              //加载驱动程序，下面的代码为加载 JDBD-ODBC 驱动程序
          Class.forName(“ oracle.jdbc.driver.OracleDriver ”);
            //建立连接，用适当的驱动程序连接到 DBMS
       String url=“ jdbc:oracle:thin:@192.168.4.45:1521:oemrep”;
       String user = “ ums ”;
       String password = “ rongji ”;
        //用 url 创建连接
       Connection con=DriverManager.getConnection(url,user,password);
//获取数据库的信息
     DatabaseMetaData dbMetaData = con.getMetaData();
//返回一个 String 类对象，代表数据库的 URL
```

```
  System.out.println("URL:"+dbMetaData.getURL()+";");
   //返回连接当前数据库管理系统的用户名。
       System.out.println("UserName:"+dbMetaData.getUserName()+";");
   //返回一个 boolean值，指示数据库是否只允许读操作。
       System.out.println("isReadOnly:"+dbMetaData.isReadOnly()+";");
   //返回数据库的产品名称。
   System.out.println("DatabaseProductName:"+dbMetaData.getDatabaseProductName()+";");
   //返回数据库的版本号。
System.out.println("DatabaseProductVersion:"+dbMetaData.getDatabaseProductVersion()+";");
   //返回驱动驱动程序的名称。
    System.out.println("DriverName:"+dbMetaData.getDriverName()+";");
  //返回驱动程序的版本号。
    System.out.println("DriverVersion:"+dbMetaData.getDriverVersion());
   //关闭连接
    con.close();
}
catch (Exception e)
{
     //输出异常信息
     System.err.println("SQLException :"+e.getMessage());
     e.printStackTrace();

}
        }
}
```