

# 1 前言：JEE 概述

## 1.1 什么是企业级应用

企业级应用是指那些为商业组织、大型企业而创建并部署的解决方案及应用。这些大型企业级应用的结构复杂，涉及的外部资源众多、事务密集、数据量大、用户数多，有较强的安全性考虑。

当代的企业级应用决不可能是一个个相互独立的系统。在企业中，一般都会部署多个彼此连接的、相互通过不同集成层次进行交互的企业级应用，同时这些应用又都有可能与其它企业的相关应用连接，从而构成一个结构复杂的、跨越 Intranet 和 Internet 的分布式企业应用群集。

此外，作为企业级应用，其不但要有强大的功能，还要能够满足未来业务需求的变化，易于升级和维护。

一般认为现代的企业计算解决方案除了企业的业务逻辑 (business logic) 外, 还需要提供对 8 种基本服务的支持，这些服务分别是：

### (1) 命名/目录服务 Naming and Directory Service

在企业范围内的信息共享 (包括计算机、用户、打印机、应用程序等所有资源) 过程中，命名/目录服务扮演着重要的角色，它维护着名字和实际资源之间的连接关系，以便使用者能通过名字实现对实际资源的透明访问。在一个大型的企业中可能有多种命名/目录服务并存，如何将所有的命名/目录服务在现有的计算架构完整地统一起来，以便让应用程序透明地访问所有的命名/目录服务，就成了企业计算解决方案必须解决的问题之一。

### (2) 数据访问服务 (Data Access Service)

大部分的应用都需要访问数据库。企业计算解决方案必须提供一种统一的界面对所有的数据库进行访问。

### (3) 分布式对象服务 (Distributed Object Service)

在一个分布式的环境中，构成整个应用的所有组件可能位于不同的服务器上，这些组件通常通过 CORBA 进行互联。在企业计算环境里必须提供一种统一的方法访问这些组件，以保护企业的投资。

### (4) 企业管理服务 (Enterprise Management Service)

在任何分布式环境中，管理都是一个关键的需求，需要应用程序提供远程管理的能力，以防止出现非法操作和访问以及对设备、应用程序状态的管理。

### **(5) 事务处理服务 (Transaction Processing Service)**

一些关键部门在日常工作中有大量的事务处理，事务处理的开发是一件极其复杂的工作。

### **(6) 消息服务 (Messaging Service)**

在一些企业，特别是一些对同步传输要求不高的企业，通常采用消息机制在应用程序之间进行通讯。

### **(7) 安全服务 (Security Service)**

应用程序的安全性是任何企业都关心的焦点之一，任何企业计算解决方案都不能回避。

### **(8) Web 服务 (Web Service)**

大部分企业级应用都以 Web 服务器为中心，Web 服务成为企业计算解决方案的重要组成部分之一。

总结：企业级开发有什么特点：

- ① 稳定可靠（鲁棒性）
- ② 异构性（软件或硬件的环境不同）
- ③ 可维护性
- ④ 扩展性
- ⑤ 可重用
- ⑥ 多线程
- ⑦ 复杂的事务管理

## **1.2 JEE 的概念**

目前，Java2 平台有 3 个版本，它们是适用于小型设备和智能卡的 Java2 平台 Micro 版 (Java Platform Micro Edition, JME)、适用于桌面系统的 Java2 平台标准版 (Java Platform Standard Edition, JSE)、适用于创建服务器应用程序和服务的 Java2 平台企业版 (Java Platform Enterprise Edition, JEE)。

**JEE 是一种利用 Java 2 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。**J2EE 技术的基础就是核心 Java 平台或 Java 2 平台的标准版，JEE 不仅巩固了标准版中的许多优点，例如“编写一次、随处运行”的特性、方便存取数据库的 JDBC API、CORBA 技术以及能够在 Internet 应用中保护数据的安全模式等等，同时还提供了对 EJB (Enterprise JavaBeans)、Java Servlets API、JSP (Java Server Pages) 以及 XML 技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。

JEE 体系结构提供中间层集成框架用来满足无需太多费用而又需要高可用性、高可靠性以及可

扩展性的应用的需求。通过提供统一的开发平台，JEE 降低了开发多层应用的费用和复杂性，同时提供对现有应用程序集成强有力支持，完全支持 Enterprise JavaBeans，有良好的向导支持打包和部署应用，添加目录支持，增强了安全机制，提高了性能。

**综上所述：JEE 能解决企业级开发中遇到的一系列问题。**

## 1.3 JEE 的优势(了解)

JEE 为搭建具有可伸缩性、灵活性、易维护性的商务系统提供了良好的机制：

1. 保留现存的 IT 资产：由于企业必须适应新的商业需求，利用已有的企业信息系统方面的投资，而不是重新制定全盘方案就变得很重要。这样，一个以渐进的（而不是激进的，全盘否定的）方式建立在已有系统之上的服务器端平台机制是公司所需求的。JEE 架构可以充分利用用户原有的投资，如一些公司使用的 BEA Tuxedo、IBM CICS, IBM Encina、Inprise VisiBroker 以及 Netscape Application Server。这之所以成为可能是因为 J2EE 拥有广泛的业界支持和一些重要的‘企业计算’领域供应商的参与。每一个供应商都对现有的客户提供了不用废弃已有投资，进入可移植的 JEE 领域的升级途径。由于基于 JEE 平台的产品几乎能够在任何操作系统和硬件配置上运行，现有的操作系统和硬件也能被保留使用。
2. 高效的开发：JEE 允许公司把一些通用的、很繁琐的服务端任务交给中间件供应商去完成。这样开发人员可以集中精力在如何创建商业逻辑上，相应地缩短了开发时间。高级中间件供应商提供以下这些复杂的中间件服务：
  - a) 管理服务 -- 让开发人员写更少的代码，不用关心如何管理状态，这样能够更快地完成程序开发。
  - b) 持续性服务 -- 让开发人员不用对数据访问逻辑进行编码就能编写应用程序，能生成更轻巧，与数据库无关的应用程序，这种应用程序更易于开发与维护。
  - c) 分布式共享数据对象 CACHE 服务 -- 让开发人员编制高性能的系统，极大提高整体部署的伸缩性。
3. 支持异构环境：JEE 能够开发部署在异构环境中的可移植程序。基于 JEE 的应用程序不依赖任何特定操作系统、中间件、硬件。因此设计合理的基于 JEE 的程序只需开发一次就可部署到各种平台。这在典型的异构企业计算环境中是十分关键的。JEE 标准也允许客户订购与 JEE 兼容的第三方的现成的组件，把他们部署到异构环境中，节省了由自己制订整个方案所需的费用。
4. 可伸缩性：企业必须要选择一种服务器端平台，这种平台应能提供极佳的可伸缩性去满足那些在他们系统上进行商业运作的大批新客户。基于 JEE 平台的应用程序可被部署到各种操作系统上。例如可被部署到高端 UNIX 与大型机系统，这种系统单机可支持 64 至 256 个处理器。（这是 NT 服务器所望尘莫及的）JEE 领域的供应商提供了更为广泛的负载平衡策略。能消除系统中的瓶颈，允许多台服务器集成部署。这种部署可达数千个处理器，实现可高度伸缩的系统，满足未来商业应用的需要。

5. 稳定的可用性：一个服务器端平台必须能全天候运转以满足公司客户、合作伙伴的需要。因为 INTERNET 是全球化的、无处不在的，即使在夜间按计划停机也可能造成严重损失。若是意外停机，那会有灾难性后果。JEE 部署到可靠的操作环境中，他们支持长期的可用性。一些 JEE 部署在 WINDOWS 环境中，客户也可选择健壮性能更好的操作系统如 Sun Solaris、IBM OS/390。最健壮的操作系统可达到 99.999% 的可用性或每年只需 5 分钟停机时间。这是实时性很强商业系统理想的选择。

## 1.4 JEE 的本质

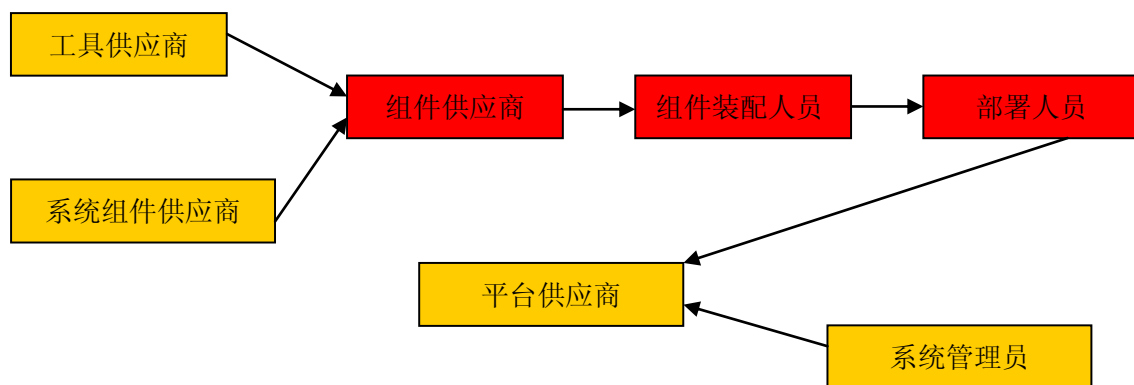
1. JEE 是一个框架集
2. JEE 是一个规范集
3. JEE 是一个技术集或 Api 集

注：

1. 能解决企业级应用开发中的问题的框架的集合；
2. 规范集就是说明规范的文档；
3. 一个框架就是一项技术，落实到代码上就是Api集；

## 1.5 JEE 中的角色划分

- 平台供应商：提供 JEE 平台，包括组件、容器、平台 APIs 等等；
- 组件供应商：提供应用程序组件，包括 HTML 页面设计人员、JSP 程序员、EJB 开发人员等等；
- 组件装配人员：组装由组件供应商提供的组件，最后形成 EAR(Enterprise Archive) 文件
- 部署人员：将装配好的组件部署到容器上；
- 系统管理员：管理和配置部署好的系统；
- 工具供应商：提供开发组件所使用的工具；
- 系统组件供应商：提供系统组件；



## 1.6 JEE 的核心 API 和组件

表现层: Servlet , Jsp , JavaBeans , Taglib , Jsf

逻辑层: Ejb (Session Bean)

数据层: JDBC , Ejb (Entity Bean)

通讯层: RMI , JNDI , IDL , IIOP

服务层: JTA , JTS , JMS , JavaMail

JEE 平台由一整套服务 (Services)、应用程序接口 (APIs) 和协议构成, 它对开发基于 Web 的多层应用提供了功能支持, 下面对 JEE 中的 13 种技术规范进行简单的描述 (这里只能进行简单的描述):

1. Java Servlet: Servlet 是一种小型的 Java 程序, 它扩展了 Web 服务器的功能。作为一种服务器端的应用, 当被请求时开始执行, 这和 CGI Perl 脚本很相似。Servlet 提供的功能大多与 JSP 类似, 不过实现的方式不同。JSP 通常是大多数 HTML 代码中嵌入少量的 Java 代码, 而 servlets 全部由 Java 写成并且生成 HTML。
2. JDBC (Java Database Connectivity): JDBC API 为访问不同的数据库提供了一种统一的途径, 象 ODBC 一样, JDBC 对开发者屏蔽了一些细节问题。JDBC API 主要用来连接数据库和直接调用 SQL 命令执行各种 SQL 语句。利用 JDBC API 可以执行一班的 SQL 语句、动态 SQL 语句以及带 IN 和 OUT 参数的存储过程。另外, JDCB 对数据库的访问也具有平台无关性。
3. JNDI (Java Name and Directory Interface): 由于 JEE 应用程序组件一般分布不同的机器上, 所以需要一种机制以便组件客户使用者查找和引用组件及资源。JNDI API 被用于执行名字和目录服务。它提供了一致的模型来存取和操作企业级的资源如 DNS 和

LDAP, 本地文件系统, 或应用服务器中的对象。

4. EJB(Enterprise JavaBean): JEE 技术之所以赢得媒体广泛重视的原因之一就是 EJB。它们提供了一个框架来开发和实施分布式商务逻辑, 由此很显著地简化了具有可伸缩性和高度复杂的企业级应用的开发。EJB 规范定义了 EJB 组件在何时如何与它们的容器进行交互作用。容器负责提供公用的服务, 例如目录服务、事务管理、安全性、资源缓冲池以及容错性。但这里值得注意的是, EJB 并不是实现 JEE 的唯一途径。正是由于 JEE 的开放性, 使得有的厂商能够以一种和 EJB 平行的方式来达到同样的目的。
5. RMI(R emote Method Invoke): 远程方法调用服务, RMI 协议调用远程对象上方法。它使用了序列化方式在客户端和服务端传递数据。RMI 是一种被 EJB 使用的更底层的协议。
6. Java IDL/CORBA: 在 Java IDL 的支持下, 开发人员可以将 Java 和 CORBA 集成在一起。他们可以创建 Java 对象并使之可在 CORBA ORB 中展开, 或者他们还可以创建 Java 类并作为和其它 ORB 一起展开的 CORBA 对象的客户。后一种方法提供了另外一种途径, 通过它 Java 可以被用于将你的新的应用和旧的系统相集成。
7. JSP(Java Server Pages): JSP 页面由 HTML 代码和嵌入其中的 Java 代码所组成。服务器在页面被客户端所请求以后对这些 Java 代码进行处理, 然后将生成的 HTML 页面返回给客户端的浏览器。
8. XML(Extensible Markup Language): XML 是一种可以用来定义其它标签语言的语言。它被用来在不同的商务过程中共享数据。XML 的发展和 Java 是相互独立的, 但是, 它和 Java 具有的相同目标正是平台独立性。通过将 Java 和 XML 的组合, 您可以得到一个完美的具有平台独立性的解决方案。
9. JMS(Java Message Service): JMS 是用于和面向消息的中间件相互通信的应用程序接口(API), 它提供创建、发送、接受、读取消息的服务。它既支持点对点的域, 也支持发布/订阅(publish/subscribe)类型的域, 并且提供对下列类型的支持: 经认可的消息传递, 事务型消息的传递, 一致性消息和具有持久性的订阅者支持。JMS 还提供了另一种方式来对新的应用与旧的后台系统相集成。
10. JTA(Java Transaction Architecture): JTA 定义了一种标准的 API, 它支持事务的开始、回滚和提交, 应用系统由此可以访问各种事务监控。
11. JTS(Java Transaction Service): JTS 是 CORBA OTS 事务监控的基本的实现。JTS 规定了事务管理器的实现方式。该事务管理器是在高层支持 Java Transaction API (JTA) 规范, 并且在较底层实现 OMG OTS specification 的 Java 映像。JTS 事务管理器为应用服务器、资源管理器、独立的应用以及通信资源管理器提供了事务服务。
12. JavaMail: JavaMail 是用于存取邮件服务器的 API, 它提供了一套邮件服务器的抽象类。不仅支持 SMTP 服务器, 也支持 IMAP 服务器。
13. JAF(JavaBeans Activation Framework): JavaMail 利用 JAF 来处理 MIME 编码的邮件附件。MIME 的字节流可以被转换成 Java 对象, 或者转换自 Java 对象。大多数应用都可以不需要直接使用 JAF。

## 2 Web 应用技术

### 2.1 概述

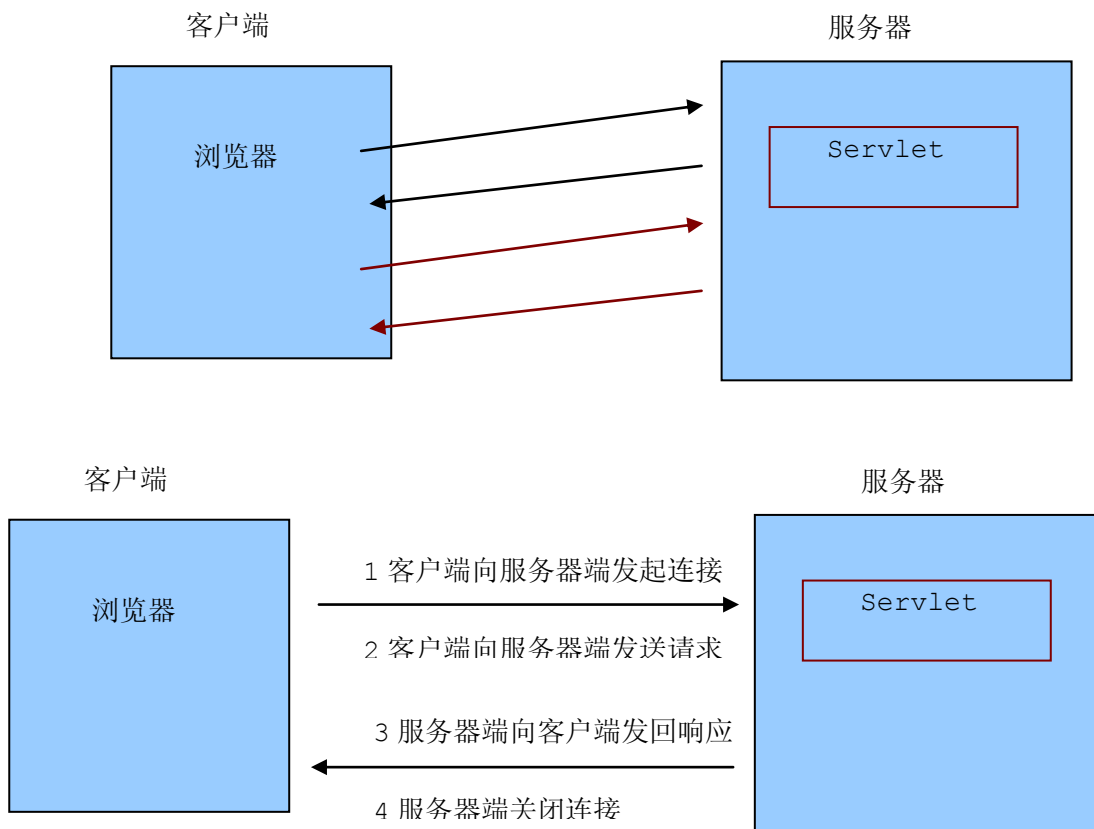
#### 2.1.1 HTTP 协议

HTTP 是 Hypertext Transfer Protocol（超文本传输协议）。从 1990 年开始就在万维网上广泛应用，是现在网络上应用最多的协议。从网络架构上讲，**Http 是应用层协议**。

**HTTP 是一个客户端和服务端请求和应答的标准（TCP）**。客户端是终端用户，通过使用 Web 浏览器，客户端就可以发起一个到服务器上指定端口（默认端口为 80）的 HTTP 请求。（我们称这个客户端）叫**用户代理（user agent）**。应答的服务器上存储着（一些）资源，比如 HTML 文件和图像。（我们称）这个应答服务器为**源服务器（origin server）**。在用户代理和服务端之间，就是通过 Http 在 Internet 上进行网络的发送和接收。

#### 2.1.2 http 协议是无连接的（无状态）

一次请求对应一个响应，请求响应完成后，谁也不认识谁，再次请求，又是一个新的请求。



### 2.1.3 IP 地址

IP(Internet Protocol) 是网络之间互连的协议，也就是为计算机网络相互连接进行通信而设计的协议。

网络中每台主机都必须有一个唯一的 IP 地址。IP 地址是一个逻辑地址，具有全球唯一性。

Ip 地址占 4 个字节（32 位），常用点分十进制的格式表示，例如：192.168.1.105  
119.57.24.4

### 2.1.4 URL

我们在浏览器的地址栏里输入的 web 地址叫做 URL (Uniform Resource Locator, 统一资源定位符)。就像每家每户都有一个门牌地址一样，每个网页也都有一个 Internet 地址。当你在浏览器的地址框中输入一个 URL 或是单击一个超级链接时，URL 就确定了要浏览的地址。浏览器通过超文本传输协议(HTTP)，将 Web 服务器上的页面代码提取出来，并翻译成漂亮的页面。

因此，在我们认识 HTTP 之前，有必要先弄清楚 URL 的组成。

例如：`http://host:8080/bookec/pages/user/index.htm`。它的含义如下：

1. `http://`：代表超文本传输协议
2. `host`：表示 Internet 主机域名或者 ip 地址
3. `8080`：端口 (port)，端口号的范围从 0 到 65535，1024 一下的端口号保留给预定义的服务。`http` 使用的是 80 端口。
4. `bookec`：webContext 区分是哪一个 web 应用
5. `pages/user/` web 应用中的路径 (path)
6. `index.htm`：是文件夹中的一个 HTML 文件 (页面)

我们称 `/bookec/pages/user/index.htm` 是统一资源标识符 URI (Uniform Resource Identifier)，注意 URI 以 / 开头。

### 2.1.5 MIME 类型

MIME 类型 (Multipurpose Internet Mail Extensions) 是多用途互联网邮件扩展类型，就是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。

每个 MIME 类型由两部分组成，前面是数据的大类别，例如声音 audio、图象 image 等，后面定义具体的种类。



常见的 MIME 类型：

1. 超文本标签语言文本 .html, .html text/html
2. 普通文本 .txt text/plain
3. RTF 文本 .rtf application/rtf
4. GIF 图形 .gif image/gif
5. JPEG 图形 .jpeg, .jpg image/jpeg
6. au 声音文件 .au audio/basic
7. MIDI 音乐文件 mid, .midi audio/midi, audio/x-midi
8. RealAudio 音乐文件 .ra, .ram audio/x-pn-realaudio
9. MPEG 文件 .mpg, .mpeg video/mpeg
10. AVI 文件 .avi video/x-msvideo
11. GZIP 文件 .gz application/x-gzip
12. TAR 文件 .tar application/x-tar

### 2.1.6 web 应用的模式

**C/S 模式 (Client/Server)**，即客户机/服务器模式，又称胖客户端，或富客户端。

C/S 结构的优点是能充分发挥客户端 PC 的处理能力，客户端程序具有了一定的数据处理和数据存储能力，很多工作可以在客户端处理后再提交给服务器。对应的优点就是客户端响应速度快，应用服务器运行数据负荷较轻。

C/S 结构的缺点：高昂的维护成本且投资大，传统的 C/S 结构的软件需要针对不同的操作系统开发不同版本的软件，由于产品的更新换代十分快，代价高和低效率已经不适应工作需要。在 JAVA 这样的跨平台语言出现之后，B/S 架构更是猛烈冲击 C/S，并对其形成威胁和挑战。客户端需要安装专用的客户端软件，不利于维护。

**B/S (Browser/Server)**，即浏览器/服务器模式，又称为瘦客户端，或贫客户端。

B/S 结构是随着 Internet 技术的兴起，对 C/S 结构的一种改进。在这种结构下，软件应用的业务逻辑完全在应用服务器端实现，用户表现完全在 Web 服务器实现，客户端只需要浏览器即可进行业务处理，是一种全新的软件系统构造技术。这种结构更成为当今应用软件的首选体系结构。

## 2.2 Web 站点与 Web 应用

Web 站点由一组分层次的 HTML 文档、媒体文件及相关目录结构组成，注重的是信息的浏览。

Web 应用是一个在服务器端具有动态功能的 Web 站点，使用 HTML form 作为客户端运行代码的用户界面。

Web 应用注重的是业务功能的实现。常见的计数器、留言版、聊天室和论坛 BBS 等，都是 Web 应用程序，不过这些应用相对比较简单，而 Web 应用程序的真正核心主要是用户的业务需求和对数据库进行处理，比如管理信息系统（Management Information System, 简称 MIS）就是这种架构最典型的应用。

## 2.3 Web 动态功能的实现

### 2.3.1 CGI 程序

CGI 代表 Common Gateway Interface (通用网关界面)，它是运行在 Web 服务器上的一个程序，并由来自于用户的输入触发。

CGI 程序的优点：

可以用各种语言编写；

实现相对容易；

CGI 程序的缺点：

每个 shell (单元) 都是重量级的，每个请求都要启动一个新的进程；

不可伸缩；

CGI 处理代码（业务逻辑）与 HTML（表现逻辑）混合在一起；

语言不一定强健，不一定是面向对象的；

语言不一定是独立于平台的；

### 2.3.2 Java Servlet

- Servlet 是在服务器上执行的 Java 组件；
- Servlet 完成的任务与 CGI 程序相似，但其在不同的环境中执行；
- Servlet 完成如下工作：**处理 HTTP 请求；动态生成 HTTP 响应；**

Java Servlet 的优点：

- 性能（线程比进程更快）；
- 可伸缩；
- Java 强健且面向对象
- Java 平台独立

Java Servlet 的缺点：

- 处理代码（业务逻辑）与 HTML（表现逻辑）混合在一起；

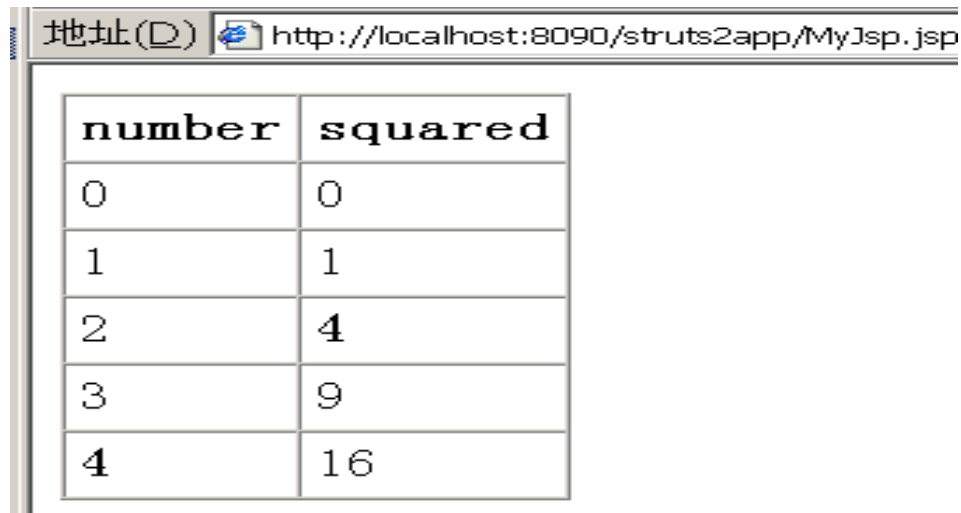
### 2.3.3 Jsp

模板页面带有嵌入代码，可完成数据和 HTML 的动态生成。

例：

```
<TABLE BORDER='1' CELLSPACING='0' CELLPADDING='5'>
<TR><TH>number</TH><TH>squared</TH></TR>
<% for ( int i=0; i<5; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
</TABLE>
```

运行效果：



| number | squared |
|--------|---------|
| 0      | 0       |
| 1      | 1       |
| 2      | 4       |
| 3      | 9       |
| 4      | 16      |

- JSP 页面被翻译成 Servlet 类并编译，在 Web 容器中作为 Servlet 对待。
- 与 ASP 和 PHP 不同，JSP 页面是需要翻译的，而不是解释的。
- JSP 页面可集中在表现逻辑，而不是业务逻辑上。
- 在使用 Java 的 Web 应用中，JSP 页面通常用作 MVC 模式中的视图部分。

JSP 的优点（具有 Servlet 所有优点）：

- 高性能、高伸缩性、平台独立性、可使用 Java 语言作为其脚本语言。

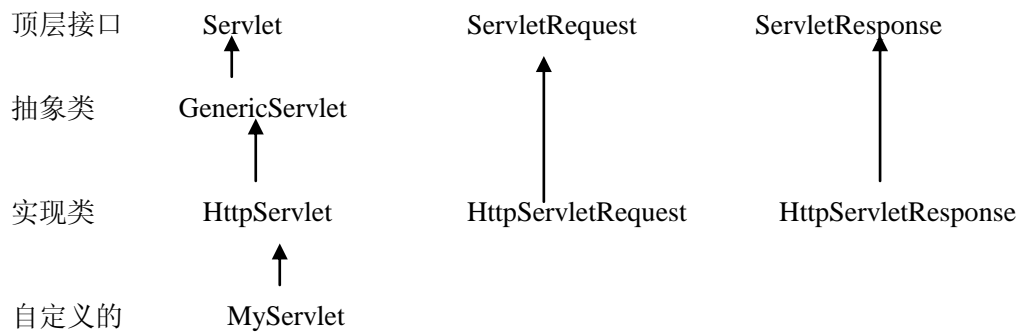
JSP 的缺点：

- 如果 JSP 页面独立使用，则 JSP 页面中完成业务和控制逻辑的脚本代码会很多

## 3 Servlet 简介

### 3.1 Servlet API

Servlet 是在服务器上执行的 Java 组件，主要功能是接收 http 请求，产生动态 http 响应。



### 3.2 servlet 的开发及运行过程

#### 3.2.1 开发及部署过程

<熟记>

(1) 构建开发环境：在 Eclipse 创建一个 java 工程，把 tomcat 下 lib 文件夹下的 servlet-api.jar 引入到 Eclipse 的工程中

(2) 开发 servlet 类：写一个类继承 HttpServlet；重写 doGet( ) doPost( )

(3) 部署：

- 安装 web 服务器，例如 Tomcat
- 在 Tomcat 的 webapps 目录下新建一个文件夹作为 web 程序的根
- 在根下新建一个名为 WEB-INF 的文件夹，里面建立一个 web.xml 的文件、一个 classes 的文件夹、一个 lib 文件夹
- 按照 servlet 的 DTD 配置 web.xml 文件

- 把编译好的 servlet 的 class 文件复制到 classes 目录下
  - lib 文件存放程序所需要的 jar 包
- (4) 启动服务器

### 3.2.2 运行过程

<熟记>

- (1) html 页面中的 form 表单发出请求到服务器中的容器（点击提交，提交到 url）
- (2) 容器接到请求找到 web.xml
- (3) 匹配对应的 Servlet 类：通过 `url-pattern` 找到 `servlet-name`，通过 `servlet-name` 找到 `servlet-class`
- (4) 容器创建 Servlet 类的实例，调用 `service` 方法（回调方法）
- (5) 执行 Servlet 类中的 `doGet` 或 `doPost` 方法

## 3.3 使用部署描述符开发 Web 应用

### 3.3.1 部署描述符 web.xml

部署描述符文件 web.xml 使用 “web app v2.3” 文档类型定义

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
  version="2.5">

  <display-name>WebApp Example</display-name>
  <description>
    This Web Application demonstrates a simple deployment descriptor.
```

```
It also demonstrates a servlet definition and servlet mapping.
</description>

<servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>com.javakc.servlet.MyServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Hello</servlet-name>
  <url-pattern>/fuq</url-pattern>
</servlet-mapping>

</web-app>
```

### 3.3.2 认识常用文件格式

|                    |                   |
|--------------------|-------------------|
| war-               | web文档压缩包          |
| ---WEB-INF         |                   |
| ----web.xml        | web应用部署文件         |
| ----lib            | 本web应用需要的jar包（可选） |
| ----classes        | 本web应用需要的类文件（可选）  |
| jar-               | java文件压缩包         |
| ---META-INF        |                   |
| ----*.MF           | java包的启动和描述文件     |
| ear-               | 企业资源包/企业应用包       |
| --META-INF         |                   |
| ---Application.xml | 配置文件              |
| --*.war            |                   |
| --*.jar            |                   |

### 3.3.3 部署环境

一个 Web 应用的部署目录结构：

webapps 是 tomcat 下的目录。

```
webapps/
  myapp/
```

```
index.html
images/
WEB-INF/
    web.xml
    classes/
    lib/
```

## 3.4 HTTP 请求

### 3.4.1 HTTP 请求格式

HTTP 请求包括三部分：**请求行** (Request Line)、**请求头部** (Headers) 和**数据体** (Body)。

- 请求行由请求方法 (method)，请求网址 Request-URI 和协议 (Protocol) 构成。
- 请求头包括多个属性，也叫请求报文。
- 数据体则可以被认为是在请求之后的文本或二进制文件。

下面这个例子显示了一个 HTTP 请求的 Header 内容，这些数据是真正以网络 HTTP 协议从 IE 浏览器传递到 Tomcat 服务器上的。

```
GET /WebTest/fuq?username=tom&age=20 HTTP/1.1
Accept:image/gif,image/x-
    xbitmap,image/jpeg,image/pjpeg,application/vnd.ms-
    powerpoint,application/vnd.ms-excel,application/msword,*.*
Accept-Language:en-us
Accept-Encoding:gzip,deflate
User-Agent:Mozilla/4.0 (compatible;MSIE 5.01;Windows NT 5.0;DigExt)
Host:www.icconcept.com:8080
Referer:http://www.yoursite.com/header.html
Connection:Keep-Alive
```

这段程序使用了 6 个 Header，还有一些 Header 没有出现。我们参考这个例子具体解释 HTTP 请求格式。

1. HTTP 请求行：请求行格式为 Method Request-URI Protocol。在上面这个例子里，“GET /icwork?search=pruduct HTTP/1.1”是请求行。
2. Accept：指浏览器或其他客户可以接受的 MIME 文件格式。Servlet 可以根据它判断并返回适当的文件格式。
3. Accept-Language：指出浏览器可以接受的语言种类，如 en 或 en-us，指英语。



4. **Accept-Encoding**: 指出浏览器可以接受的编码方式。编码方式不同于文件格式, 它是为了压缩文件并加速文件传递速度。浏览器在接收到 Web 响应之后先解码, 然后再检查文件格式。
5. **Referer**: 表明产生请求的网页 URL。如比从网页/icconcept/index.jsp 中点击一个链接到网页/icwork/search, 在向服务器发送的 GET/icwork/search 中的请求中, Referer 是 http://hostname:8080/icconcept/index.jsp。这个属性可以用来跟踪 Web 请求是从什么网站来的, 可以用它来实现防止盗链。
6. **Connection**: 用来告诉服务器是否可以维持持续的 HTTP 连接 (Persistent Connections)。HTTP/1.1 使用 Keep-Alive 为默认值, 这样, 当浏览器需要多个文件时 (比如一个 HTML 文件和相关的图形文件), 不需要每次都建立连接, 当连接超时, 服务器将连接自动断掉。
7. **Cookie**: 浏览器用这个属性向服务器发送 Cookie。Cookie 是在浏览器中寄存的小型数据体, 它可以记载和服务器相关的用户信息, 也可以用来实现会话功能。

|                                                |
|------------------------------------------------|
| 请求行(固定项) GET /icwork/? search=product HTTP/1.1 |
| 请求头: Accept(key--value)                        |
|                                                |
| Content (通过页面提交的值)                             |

### 3.4.2 ServletRequest 接口 API

定义:

```
public interface ServletRequest
```

定义一个 Servlet 引擎产生的对象, 通过这个对象, Servlet 可以获得客户端请求的数据。这个对象通过读取请求体的数据提供包括参数的名称、值和属性以及输入流的所有数据。

主要方法:

1、 **public Object getAttribute(String name);**

返回请求中指定属性的值, 如果这个属性不存在, 就返回一个空值。这个方法允许访问一些不提供给这个接口中其他方法的请求信息以及其他 Servlet 放置在这个请求对象内的数据。

2、 **public Enumeration getAttributeNames();**

返回包含在这个请求中的所有属性名的列表。

3、 **public void setAttribute(String name, Object object);**

这个方法在请求中添加一个属性，这个属性可以被其他可以访问这个请求对象的对象（例如一个嵌套的 Servlet）使用。

4、 **public String getParameter(String name);**

以一个 String 返回指定的参数的值，如果这个参数不存在返回空值。例如，在一个 HTTP Servlet 中，这个方法会返回一个指定的查询语句产生的参数的值或一个被提交的表单中的参数值。如果一个参数名对应着几个参数值，这个方法只能返回通过 getParameterValues 方法返回的数组中的第一个值。因此，如果这个参数有（或者可能有）多个值，你只能使用 getParameterValues 方法。

5、 **public Enumeration getParameterNames();**

返回所有参数名的 String 对象列表，如果没有输入参数，该方法返回一个空值。

6、 **public String[] getParameterValues(String name);**

通过一个 String 对象的数组返回指定参数的值，如果这个参数不存在，该方法返回一个空值。

7、 **public String getCharacterEncoding();**

返回请求中输入内容的字符编码类型，如果没有定义字符编码类型就返回空值。

8、 **public int getContentLength();**

请求内容的长度，如果长度未知就返回-1。

9、 **public String getContentType();**

返回请求数据体的 MIME 类型，如果类型未知返回空值。

10、 **public ServletInputStream getInputStream()  
throws IOException;**

返回一个输入流用来从请求体读取二进制数据。如果在此之前已经通过 `getReader` 方法获得了要读取的结果，这个方法会抛出一个 `IllegalStateException`。

11、 **`public BufferedReader getReader() throws IOException;`**

这个方法返回一个 `buffered reader` 用来读取请求体的实体，其编码方式依照请求数据的编码方式。如果这个请求的输入流已经被 `getInputStream` 调用获得，这个方法会抛出一个 `IllegalStateException`。

### 3.4.3 `HttpServletRequest` 接口 API

定义:

**`public interface HttpServletRequest extends ServletRequest;`**

用来处理一个对 `Servlet` 的 HTTP 格式的请求信息。

主要方法:

2. **`public Cookie[] getCookies();`**

返回一个数组，该数组包含这个请求中当前的所有 `cookie`。如果这个请求中没有 `cookie`，返回一个空数组。

3. **`public long getDateHeader(String name);`**

返回指定的请求头域的值，这个值被转换成一个反映自 1970-1-1 日 (GMT) 以来的精确到毫秒的长整数。

如果头域不能转换，抛出一个 `IllegalArgumentException`。如果这个请求头域不存在，这个方法返回 -1。

4. **`public String getHeader(String name);`**

返回一个请求头域的值。（译者注：与上一个方法不同的是，该方法返回一个字符串）

如果这个请求头域不存在，这个方法返回 -1。

#### 5. **public Enumeration getHeaderNames();**

该方法返回一个 String 对象的列表，该列表反映请求的所有头域名。

有的引擎可能不允许通过这种方法访问头域，在这种情况下，这个方法返回一个空的列表。

#### 6. **public int getIntHeader(String name);**

返回指定的请求头域的值，这个值被转换成一个整数。

如果头域不能转换，抛出一个 `IllegalArgumentException`。如果这个请求头域不存在，这个方法返回-1。

#### 7. **public String getMethod();**

返回这个请求使用的 HTTP 方法（例如：GET、POST、PUT）

#### 8. **public String getRequestedSessionId();**

返回这个请求相应的 `sessionId`。如果由于某种原因客户端提供的 `sessionId` 是无效的，这个 `sessionId` 将与在当前 session 中的 `sessionId` 不同，与此同时，将建立一个新的 session。

如果这个请求没与一个 session 关联，这个方法返回空值。

#### 9. **public String getRequestURI();**

从 HTTP 请求的第一行返回请求的 URL 中定义被请求的资源的部分。如果有一个查询字符串存在，这个查询字符串将不包括在返回值当中。例如，一个请求通过 `/catalog/books?id=1` 这样的 URL 路径访问，这个方法将返回 `/catalog/books`。这个方法的返回值包括了 Servlet 路径和路径信息。

如果这个 URL 路径中的一部分经过了 URL 编码，这个方法的返回值在返回之前必须经过解码。

#### 10. **public String getServletPath();**

这个方法返回请求 URL 反映调用 Servlet 的部分。例如，一个 Servlet 被映射到 `/catalog/summer` 这个 URL 路径，而一个请求使用了 `/catalog/summer/casual` 这样的路径。所谓的反映调用 Servlet 的部分就是指 `/catalog/summer`。

如果这个 Servlet 不是通过路径匹配来调用。这个方法将返回一个空值。

#### 11. `public HttpSession getSession();`

```
public HttpSession getSession(boolean create);
```

返回与这个请求关联的当前的有效的 session。如果调用这个方法时没带参数，那么在没有 session 与这个请求关联的情况下，将会新建一个 session。如果调用这个方法时带入了一个布尔型的参数，只有当这个参数为真时，session 才会被建立。

为了确保 session 能够被完全维持。Servlet 开发者必须在响应被提交之前调用该方法。

如果带入的参数为假，而且没有 session 与这个请求关联。这个方法会返回空值。

#### 12. `public boolean isRequestedSessionIdValid();`

这个方法检查与此请求关联的 session 当前是不是有效。如果当前请求中使用的 session 无效，它将不能通过 getSession 方法返回。

### 3.4.4 HttpServletRequest 功能示例

#### (1) 读取 HTTP 头信息

代码示例：

```
//通过request得到request的header的值
Enumeration e=request.getHeaderNames();
while(e.hasMoreElements()){
    String key=(String)e.nextElement();
    String value=(String)request.getHeader(key);
    System.out.println("key="+key+",value="+value);
}
```

header 中包含客户端的浏览器和操作系统信息，还有请求来源的信息。

```
//防止盗链
String s=request.getHeader("referer");
if("http://www.javakc.com".equals(s)){
    System.out.println("请求是从java快车的页面链接来的");
}
```

referer 两个常用的功能

1、防止盗连，比如我是个下载软件的网站，在下载页面上先用 referer 来判断上一页面是不是自己的网站，如果不是，说明有人盗连了你的下载地址。

2、电子商务网站的安全，在提交信用卡等重要信息的页面用 referer 来判断是不是从自己的网站提交的请求，如果不是，可能是黑客用自己写的一个表单，来提交的请求，为了能跳过上一页里的 javascript 的验证等目的。

使用 referer 的注意事项：

如果我是直接在浏览器地址栏里输入 url 打开的页面，取 referer 返回是 null，也就是说 referer 只有从页面点击链接来到这页的才会有内容。

## (2) 取得 cookies

代码示例：

```
//通过request取得Cookie
Cookie[] cookies = request.getCookies();
for ( int i=0; i < cookies.length; i++ ) {
    String key =cookies[i].getName();
    String value = cookies[i].getValue();
    System.out.println("cook"+i+":key="+key+",value="+value);
}
```

## (3) 取得路径信息

代码示例：

```
String s1=request.getContextPath();
String s2=request.getServletPath();
String s3=request.getRequestURI();
StringBuffer sb=request.getRequestURL();
```

输出结果：

```
s1=/servleittest
s2=/hello
s3=/hello
sb=http://localhost:8090/servleittest/hello
```

## (4) 标识 HTTP 会话

代码示例：

```
HttpSession session=request.getSession();
```

## 3.5 HTTP 响应

### 3.5.1 HTTP 响应格式

与请求格式类似，Http 响应也是由三部分组成：状态行、头信息、响应内容。

|                                                                    |
|--------------------------------------------------------------------|
| HTTP/1.1 200 OK                                                    |
| Head 头信息                                                           |
| Content (HTML 代码)<br><br><HTML><br><br><HEAD></HEAD><br><br><BODY> |

状态行的 200 表示状态码，状态码由三位数字组成，表示请求处理的结果。

- 1xx: 表示临时响应并需要请求者继续执行操作
- 2xx: 表示成功处理了请求
- 3xx: 要完成请求，需要进一步操作。通常这些状态码用来重定向
- 4xx: 表示请求可能出错，妨碍了服务器的处理
- 5xx: 表示服务器在处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错

OK 表示状态描述，只是一个文本的描述。

### 3.5.2 ServletResponse 接口 API

定义：

```
public interface ServletResponse
```

定义一个 Servlet 引擎产生的对象，通过这个对象，Servlet 对客户端的请求作出响应。这

个响应应该是一个 MIME 实体，可能是一个 HTML 页、图象数据或其他 MIME 的格式。

主要方法：

1、 **public ServletOutputStream getOutputStream()  
throws IOException;**

返回一个记录二进制的响应数据的输出流。

如果这个响应对象已经调用 `getWriter`，将会抛出 `IllegalStateException`。

2、 **public PrintWriter getWriter throws IOException;**

这个方法返回一个 `PrintWriter` 对象用来记录格式化的响应实体。如果要反映使用的字符编码，必须修改响应的 MIME 类型。在调用这个方法之前，必须设定响应的 `content` 类型。

如果没有提供这样的编码类型，会抛出一个 `UnsupportedEncodingException`，如果这个响应对象已调用 `getOutputStream`，会抛出一个 `IllegalStateException`。

3、 **public void setContentLength(int length);**

设置响应的内容的长度，这个方法会覆盖以前对内容长度的设定。

为了保证成功地设定响应头的内容长度，在响应被提交到输出流之前必须调用这个方法。

4、 **public void setContentType(String type);**

这个方法用来设定响应的 `content` 类型。这个类型以后可能会在另外的一些情况下被隐式地修改，这里所说的另外的情况可能当服务器发现有必要的情况下对 MIME 的字符设置。默认是 `text/html`

为了保证成功地设定响应头的 `content` 类型，在响应被提交到输出流之前必须调用这个方法。

### 3.5.3 HttpServletResponse 接口 API

定义：

```
public interface HttpServletResponse extends ServletResponse
```

描述一个返回到客户端的 HTTP 回应。这个接口允许 `Servlet` 程序员利用 HTTP 协议规定的头



信息。

主要方法：

**1、 `public void addCookie(Cookie cookie);`**

在响应中增加一个指定的 cookie。可多次调用该方法以定义多个 cookie。为了设置适当的头域，该方法应该在响应被提交之前调用。

**2、 `public boolean containsHeader(String name);`**

检查是否设置了指定的响应头。

**3、 `public String encodeRedirectURL(String url);`**

对 `sendRedirect` 方法使用的指定 URL 进行编码。如果不需要编码，就直接返回这个 URL。之所以提供这个附加的编码方法，是因为在 `redirect` 的情况下，决定是否对 URL 进行编码的规则和一般情况有所不同。所给的 URL 必须是一个绝对 URL。相对 URL 不能被接收，会抛出一个 `IllegalArgumentException`。

所有提供给 `sendRedirect` 方法的 URL 都应通过这个方法运行，这样才能确保会话跟踪能够在所有浏览器中正常运行。

**4、 `public void sendError(int statusCode) throws IOException;`**

```
public void sendError(int statusCode, String message)  
  
throws IOException;
```

用给定的状态码发给客户端一个错误响应。如果提供了一个 `message` 参数，这将作为响应体的一部分被发出，否则，服务器会返回错误代码所对应的标准信息。

调用这个方法后，响应立即被提交。在调用这个方法后，`Servlet` 不会再有更多的输出。

**5、 `public void sendRedirect(String location)`**

**`throws IOException;`**

使用给定的路径，给客户端发出一个临时转向的响应（`SC_MOVED_TEMPORARILY`）。给定的路径必须是任意的绝对 URL。相对 URL 将不能被接收，会抛出一个 `IllegalArgumentException`。

这个方法必须在响应被提交之前调用。调用这个方法后，响应立即被提交。在调用这个方法后，Servlet 不会再有更多的输出。

#### 6、 `public void setDateHeader(String name, long date);`

用一个给定的名称和日期值设置响应头，这里的日期值应该是反映自 1970-1-1 日（GMT）以来的精确到毫秒的长整数。如果响应头已经被设置，新的值将覆盖当前的值。

#### 7、 `public void setHeader(String name, String value);`

用一个给定的名称和域设置响应头。如果响应头已经被设置，新的值将覆盖当前的值。

#### 8、 `public void setIntHeader(String name, int value);`

用一个给定的名称和整形值设置响应头。如果响应头已经被设置，新的值将覆盖当前的值。

### 3.5.4 HttpServletResponse 功能示例

- (1) 设置 HTTP 头标
- (2) 设置 cookie
- (3) 设定响应的 content 类型
- (4) 输出返回数据

代码示例：

```
//设置HTTP头标
// 5 秒后自动刷新并转到http://.....
response.setHeader("Refresh", "5;URL=http://www.javakc.com");
//设置cookie
Cookie c=new Cookie("cookieKey","cookieValue");
response.addCookie(c);
//设定响应的content类型
response.setContentType("text/html");
response.setContentType("application/msword;charset=GB2312");
//输出返回数据
PrintWriter out =response.getWriter();
out.print("<font color='red'>Hello success!</font>");
out.print("<br>");
out.print("userId="+userId+",userName="+userName);
out.close();
```

## 3.6 HttpServlet

### 3.6.1 GenericServlet 抽象类 API

定义:

```
public abstract class GenericServlet  
  
    implements Servlet, ServletConfig, Serializable;
```

这个类的存在使得编写 Servlet 更加方便。它提供了一个简单的方案，这个方案用来执行有关 Servlet 生命周期的方法以及在初始化时对 ServletConfig 对象和 ServletContext 对象进行说明。

主要方法:

1、 **public void init() throws ServletException;**

```
public void init(ServletConfig config)  
  
    throws ServletException;
```

init(ServletConfig config) 方法是一个对这个 Servlet 的生命周期进行初始化的简便的途径。

init() 方法是用来让你对 GenericServlet 类进行扩充的，使用这个方法时，你不需要存储 config 对象，也不需要调用 super.init(config)。

init(ServletConfig config) 方法会存储 config 对象然后调用 init()。如果你重载了这个方法，你必须调用 super.init(config)，这样 GenericServlet 类的其他方法才能正常工作。

2、 **public abstract void service(ServletRequest request,  
ServletResponse response)  
throws ServletException, IOException;**

这是一个抽象的方法，当你扩展这个类时，为了执行网络请求，你必须执行它。

### 3、**public void destroy();**

在这里 destroy 方法不做任何其他的工作。

### 4、**public String getInitParameter(String name);**

这是一个简便的途径，它将会调用 ServletConfig 对象的同名的方法。

### 5、**public Enumeration getInitParameterNames();**

这是一个简便的途径，它将会调用 ServletConfig 对象的同名的方法。

### 6、**public ServletConfig getServletConfig();**

返回一个通过这个类的 init 方法产生的 ServletConfig 对象的说明。

### 7、**public ServletContext getServletContext();**

这是一个简便的途径，它将会调用 ServletConfig 对象的同名的方法。

### 8、**public String getServletInfo();**

返回一个反映 Servlet 版本的 String。

### 9、**public void log(String msg);**

**public void log(String msg, Throwable cause);**

通过 Servlet content 对象将 Servlet 的类名和给定的信息写入 log 文件中。

## 3.6.2 HttpServlet 类 API

定义：

```
public class HttpServlet extends GenericServlet
                                   implements Serializable
```

它是 `GenericServlet` 类的扩充，提供了一个处理 HTTP 协议的框架。

在这个类中的 `service` 方法支持例如 GET、POST 这样的标准的 HTTP 方法。这一支持过程是通过分配他们到适当的方法（例如 `doGet`、`doPost`）来实现的。

主要方法：

```
1、  protected void service(HttpServletRequest request,
                                HttpServletResponse response)
                                throws ServletException, IOException;

    public void service(ServletRequest request,
                        ServletResponse response)
                        throws ServletException, IOException;
```

这是一个 `Servlet` 的 HTTP-specific 方案，它分配请求到这个类的支持这个请求的其他方法。

当你开发 `Servlet` 时，在多数情况下你不必重写这个方法。

```
2、  protected void doGet(HttpServletRequest request,
                            HttpServletResponse response)
                            throws ServletException, IOException;
```

被这个类的 `service` 方法调用，用来处理一个 HTTP GET 操作。这个操作允许客户端简单地从一个 HTTP 服务器“获得”资源。对这个方法的重写将自动地支持 HEAD 方法。

GET 操作应该是安全而且没有负面影响的。这个操作也应该可以安全地重复。

这一方法的默认执行结果是返回一个 HTTP BAD\_REQUEST 错误。

```
3、  protected void doPost(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException;
```

被这个类的 `service` 方法调用，用来处理一个 HTTP POST 操作。这个操作包含请求体的数据，Servlet 应该按照他行事。

这个操作可能有负面影响。例如更新存储的数据或在线购物。

这一方法的默认执行结果是返回一个 HTTP BAD\_REQUEST 错误。当你要处理 POST 操作时，你必须在 `HttpServlet` 的子类中重写这一方法。

```
4、    protected void doPut(HttpServletRequest request,  
  
        HttpServletResponse response)  
  
        throws ServletException, IOException;
```

被这个类的 `service` 方法调用，用来处理一个 HTTP PUT 操作。这个操作类似于通过 FTP 发送文件。

这个操作可能有负面影响。例如更新存储的数据或在线购物。

这一方法的默认执行结果是返回一个 HTTP BAD\_REQUEST 错误。当你要处理 PUT 操作时，你必须在 `HttpServlet` 的子类中重写这一方法。

```
5、    protected void doTrace(HttpServletRequest request,  
  
        HttpServletResponse response)  
  
        throws ServletException, IOException;
```

被这个类的 `service` 方法调用，用来处理一个 HTTP TRACE 操作。这个操作的默认执行结果是产生一个响应，这个响应包含一个反映 trace 请求中发送的所有头域的信息。

当你开发 Servlet 时，在多数情况下你需要重写这个方法。

```
6、    protected void doDelete(HttpServletRequest request,  
  
        HttpServletResponse response)  
  
        throws ServletException, IOException;
```

被这个类的 `service` 方法调用，用来处理一个 HTTP DELETE 操作。这个操作允许客户端请求从服务器上删除 URL。这一操作可能有负面影响，对此用户就负起责任。

这一方法的默认执行结果是返回一个 HTTP BAD\_REQUEST 错误。当你要处理 DELETE 请求时，

你必须重写这一方法。

```
7、 protected void doHead(HttpServletRequest request,  
  
                           HttpServletResponse response)  
  
    throws ServletException, IOException;
```

被这个类的 service 方法调用，用来处理一个 HTTP HEAD 操作。默认的情况是，这个操作会按照一个无条件的 GET 方法来执行，该操作不向客户端返回任何数据，而仅仅是返回包含内容长度的头信息。

与 GET 操作一样，这个操作应该是安全而且没有负面影响的。这个操作也应该可以安全地重复。

这个方法的默认执行结果是自动处理 HTTP HEAD 操作，这个方法不需要被一个子类执行。

```
8、 protected void doOptions(HttpServletRequest request,  
  
                             HttpServletResponse response)  
  
    throws ServletException, IOException;
```

被这个类的 service 方法调用，用来处理一个 HTTP OPTION 操作。这个操作自动地决定支持哪一种 HTTP 方法。例如，一个 Servlet 写了一个 HttpServlet 的子类并重载了 doGet 方法，doOption 会返回下面的头：

```
Allow: GET,HEAD,TRACE,OPTIONS
```

你一般不需要重写这个方法。

```
9、 protected long getLastModified(  
  
                           HttpServletRequest request);
```

返回这个请求实体的最后修改时间。为了支持 GET 操作，你必须重写这一方法，以精确地反映最后修改的时间。这将有助于浏览器和代理服务器减少装载服务器和网络资源，从而更加有效地工作。返回的数值是自 1970-1-1 日（GMT）以来的毫秒数。

默认的执行结果是返回一个负数，这标志着最后修改时间未知，它也不能被一个有条件的 GET 操作使用。

### 3.6.3 构建自己的 Servlet 类

可通过扩展 `HttpServlet` 类，并覆盖 `doGet` 方法和 `doPost` 方法来构建自己的 `HttpServlet`。

```
package com.javakc.test1servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet{

    public void doGet(HttpServletRequest request,HttpServletResponse
response) throws IOException{
        this.doPost(request, response);
    }

    public void doPost(HttpServletRequest request,HttpServletResponse
response) throws IOException{

        //1,接受,得到request里面的值
        String userId=request.getParameter("userId");
        String userName=request.getParameter("userName");
        //2,处理,接到这些数据后要干什么
        System.out.println("userId="+userId+",userName="+userName);
        //3,响应,把数据返回给客户端
        response.setContentType("text/html");
        PrintWriter out =response.getWriter();
        out.print("<font color='red'>Hello success!</font>");
        out.print("<br>");
        out.print("userId="+userId+",userName="+userName);
        out.close();
    }
}
```



## 4 使用 HTML Form 的 Servlet

### 4.1 HTML Form

FORM 标签

ACTION- 指定 form 信息的目的地（相关的 URL）

METHOD - 指定 HTTP 方法（GET 或 POST）

语法：

```
<FORM ACTION='servlet-URL' METHOD='{GET|POST}'>
```

```
    {HTML form tags and other HTML content}
```

```
</FORM>
```

FORM 标签例：

```
<BODY BGCOLOR='white'>

<B>Say Hello Form</B>

<FORM ACTION='/servleittest/user' METHOD='POST'>

    Name: <INPUT TYPE='text' NAME='name'> <BR>

    <BR>

    <INPUT TYPE='submit'>

</FORM>
```

&lt;/BODY&gt;

## 4.2 HTML Form 组件

| Form                  | Element Tag                                                                                      | Description                           |
|-----------------------|--------------------------------------------------------------------------------------------------|---------------------------------------|
| Textfield             | <code>&lt;input type = "text" name = "txt"&gt;</code>                                            | 输入单行文本                                |
| Submit button         | <code>&lt;input type = "submit"&gt;</code>                                                       | 提交 form                               |
| Reset button          | <code>&lt;input type = "reset"&gt;</code>                                                        | 复位 form 中的域                           |
| Checkbox              | <code>&lt;input type = 'checkbox' ...&gt;</code>                                                 | 多选                                    |
| Radio button          | <code>&lt;input type = 'radio' ...&gt;</code>                                                    | 单选                                    |
| Password              | <code>&lt;input type = 'password' ...&gt;</code>                                                 | 输入单行文本, 但输入文本不可见                      |
| Hidden                | <code>&lt;input type = 'hidden' ...&gt;</code>                                                   | 静态数据域, 不显示在浏览器的 HTML form 中, 但发送到服务器端 |
| Select drop-down list | <code>&lt;select&gt;</code><br><code>&lt;option...&gt;</code> ..<br><code>&lt;/select&gt;</code> | 从列表中选择一项或多项                           |
| Text area             | <code>&lt;TEXTAREA ...&gt; ... &lt;/TEXTAREA&gt;</code>                                          | 输入一段文本                                |

### 1、Input 标签

INPUT 标签有三个主要属性:

- TYPE - GUI 组件的类型
- NAME - form 参数名
- VALUE - 组件的缺省值

#### ➤ 文本域:

```
<INPUT TYPE='text' NAME='name' VALUE='default value' SIZE='20'>
```

#### ➤ 提交按钮:

```
<INPUT TYPE='submit'> <BR>
```

```
<INPUT TYPE='submit' VALUE='Register'> <BR>
```

```
<INPUT TYPE='submit' NAME='operation' VALUE='Send Mail'> <BR>
```

➤ 复位按钮:

```
<INPUT TYPE='reset'>
```

➤ 复选框:

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='apple'> I like apples <BR>
```

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='orange'> I like oranges <BR>
```

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='banana'> I like bananas <BR>
```

➤ 单选框:

```
<INPUT TYPE='radio' NAME='gender' VALUE='F'> Female <BR>
```

```
<INPUT TYPE='radio' NAME='gender' VALUE='M'> Male <BR>
```

➤ 口令域:

```
<INPUT TYPE='password' NAME='psword' VALUE='secret' MAXSIZE='16'>
```

➤ 隐藏域:

```
<INPUT TYPE='hidden' NAME='action' VALUE='SelectLeague'>
```

隐藏域不在浏览器中显示，但其数据包含在 HTTP 请求的 form 数据中

## 2、Select 标签

➤ 单选下拉列表:

```
<SELECT NAME='favoriteArtist'>
```

```
<OPTION VALUE='Genesis'> Genesis  
  
<OPTION VALUE='PinkFloyd' SELECTED> Pink Floyd  
  
<OPTION VALUE='KingCrimson'> King Crimson  
  
</SELECT>
```

➤ 多选下拉列表:

```
<SELECT NAME='sports' MULTIPLE>  
  
<OPTION VALUE='soccer' SELECTED> Soccer  
  
<OPTION VALUE='tennis'> Tennis  
  
<OPTION VALUE='ultimate' SELECTED> Ultimate Frisbee  
  
</SELECT>
```

### 3、Textarea 标签

➤ 文本域标签:

```
<TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'>  
  
This module covers Web application basics: how browsers and Web  
  
servers were developed and what they do. It also...  
  
</TEXTAREA>
```

## 4.3 HTTP 请求方法

### 4.3.1 GET 方法

Form 数据包含在 HTTP 请求的 URL 中:

```
GET /servlet/sl314.web.FormBasedHello?name=Bryan HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,
*/*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

### 4.3.2 POST 方法

Form 数据包含在 HTTP 请求体中

HTTP POST 方法只可在 form 中被激活

```
POST /register HTTP/1.0
Referer: http://localhost:8080/model1/formEchoServer.html
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,
*/*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 129

season=Spring&year=2001&name=Bryan+Basham&address=4747+Bogus+Drive&city=Bo
ulder&province=Colorado&postalCode=80303&division=Amateur
```

### 4.3.3 GET 与 POST 的选择

HTTP GET 方法用于:

- 请求对服务器没有负面影响
- Form 数据量小
- 数据的内容应在 URL 中可见, 明文传输, 安全度低

HTTP POST 方法用于:

- 请求的处理过程改变服务器的状态，如在数据库中存储数据
- Form 数据量大
- 数据的内容应在 URL 中不可见（如，口令），安全度高

## 5 Servlet 生命周期

### 5.1 Servlet 生命周期概述

生命周期是指 servlet 实例在 web 容器中从：首次创建调用 init 方法开始初始化期，经过 service 方法运行期，一直到 destroy 方法销毁期。

结束 servlet 实例的生命周期由 web 容器来管理。

#### 1、Servlet 接口

➤ 方法：

```
init()
```

```
destroy()
```

```
service(ServletRequest req, ServletResponse res )
```

➤ Web 容器管理 servlet 实例的生命周期，用户不能调用这些方法。

这种由容器来调用的方法，叫回调方法，特点是由容器决定什么时候调用

➤ 可提供这些方法的实现，来操纵 servlet 实例及资源

#### 2、init 方法

➤ init 方法在 servlet 实例首次创建时由 Web 容器调用

➤ Servlet 规范确保 init 方法完成之前该 servlet 不会处理任何请求

➤ 需覆盖 init 方法的情况：

- 创建或打开任何与 servlet 相关的资源
- 初始化 servlet 的状态（数据）

#### 3、service 方法

➤ service 方法由 Web 容器调用，处理用户请求

- HttpServlet 类实现了 service 方法，根据 HTTP 请求方法（GET、POST 等），将请求分发到 doGet、doPost 等方法

#### 4、destroy 方法

- destroy 方法在 servlet 实例被销毁时由 Web 容器调用
- Servlet 规范确保在 destroy 方法调用之前所有请求的处理均完成
- 需要覆盖 destroy 方法的情况：
  - 释放任何在 init 方法中打开的与 servlet 相关的资源
  - 存储 servlet 的状态

## 5.2 Servlet API

### 1、Servlet 接口：

```
✎ init(config:ServletConfig)
✎ service(request,response)
✎ destroy()
```

### 2、ServletConfig 接口：

```
✎ getInitParameter(name:String) : String
✎ getInitParameterNames() : Enumeration
✎ getServletName() : String
```

### 3、GenericServlet 抽象类：

```
✎ init(config:ServletConfig)
✎ init()
✎ service(request,response)
✎ destroy()
✎ getInitParameter(name:String) : String
✎ getInitParameterNames() : Enumeration
✎ getServletName() : String
```



#### 4、HttpServlet 类:

- ☞ init()
- ☞ doPost(request, response)
- ☞ doGet(request, response)
- ☞ ...

### 5.3 初始化参数

#### (2) 初始化参数的部署描述符:

```
<servlet>
  <servlet-name>EnglishHello</servlet-name>
  <servlet-class>examples.web.FirstServlet</servlet-class>
  <init-param>
    <param-name>greetingText</param-name>
    <param-value>Hello</param-value>
  </init-param>
</servlet>

<servlet>
  <servlet-name>FrenchHello</servlet-name>
  <servlet-class>examples.web.secondServlet</servlet-class>
  <init-param>
    <param-name>greetingText2</param-name>
    <param-value>Bonjour</param-value>
  </init-param>
</servlet>
```

初始化参数存储在与 Servlet 实例相关的 ServletConfig 对象中

#### (3) Servlet 访问初始化参数:

```
public class FirstServlet extends HttpServlet {

    private String greetingText;

    public void init() {
        greetingText = getInitParameter("greetingText");
        System.out.println(">> greetingText = '" + greetingText + "'");
    }
}
```

(4) 使用初始化参数:

```
// Generate the HTML response
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>Hello Servlet</TITLE>");
out.println("</HEAD>");
out.println("<BODY BGCOLOR='white'>");
out.println("<B>" + greetingText + ", " + name + "</B>");
out.println("</BODY>");
out.println("</HTML>");
```

## 6 使用 ServletContext

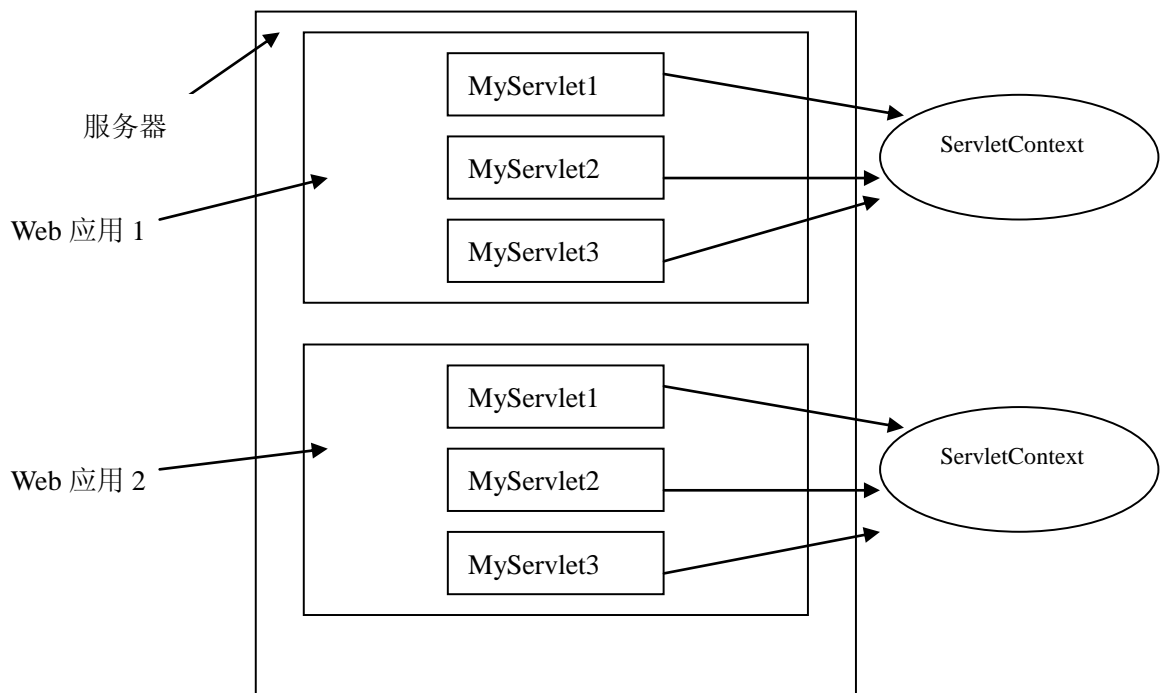
### 6.1 ServletContext

➤ Web 应用是一组自包含的静态和动态资源：

- HTML 页面
- 媒体文件
- 数据和资源文件
- servlet（和 JSP 页面）
- 其它辅助的 Java 类和对象

➤ Web 应用部署描述符用于指定一个 Web 应用使用的结构和服务

➤ ServletContext 对象是 Web 应用的运行时表示，约等于整个 web 应用，可通过其实现 Web 应用中的资源共享。



## 6.2 ServletContext API

定义:

```
public interface ServletContext
```

- 定义了一个 Servlet 的环境对象，Servlet 通过 `getServletContext` 方法访问 Servlet 上下文对象。

Servlet 上下文是提供给 Servlet 访问容器的类。

- servlet 上下文对象提供:
  - 只读访问上下文初始化参数
  - 读写访问应用级属性
  - 只读访问应用级文件资源
  - 写访问应用级日志文件

主要方法:

```
public Object getAttribute(String name);
```

返回 Servlet 环境对象中指定的属性对象。如果该属性对象不存在，返回空值。这个方法允许访问有关这个 Servlet 引擎的在该接口的其他方法中尚未提供的附加信息。

```
public Enumeration getAttributeNames();
```

返回一个 Servlet 环境对象中可用的属性名的列表。

```
public String getInitParameter(String name);
```

返回指定名字的 ServletContext 的初始化参数。

```
public void setAttribute(String name, Object o);
```

给予 Servlet 环境对象中你所指定的对象一个名称。

```
public void removeAttribute(String name);
```

从指定的 Servlet 环境对象中删除一个属性。

```
public ServletContext getContext(String uripath);
```

返回一个 Servlet 环境对象，这个对象包括了特定 URI 路径的 Servlets 和资源，如果该路

径不存在，则返回一个空值。URI 路径格式是/dir/dir/filename.ext。

为了安全，如果通过这个方法访问一个受限制的 Servlet 的环境对象，会返回一个空值。

```
public int getMajorVersion();
```

返回 Servlet 引擎支持的 Servlet API 的主版本号。例如对于 2.1 版，这个方法会返回一个整数 2。

```
public String getRealPath(String path);
```

一个符合 URL 路径格式的指定的虚拟路径的格式是：/dir/dir/filename.ext，必须以“/”开头。用这个方法，可以返回与一个符合该格式的虚拟路径相对应的真实路径的 String。这个真实路径的格式应该适合于运行这个 Servlet 引擎的计算机（包括其相应的路径解析器）。

```
public URL getResource(String uripath);
```

返回一个 URL 对象，该对象反映位于给定的 URL 地址（格式：/dir/dir/filename.ext）的 Servlet 环境对象已知的资源。无论 URLStreamHandlers 对于访问给定的环境是不是必须的，Servlet 引擎都必须执行。如果给定的路径的 Servlet 环境没有已知的资源，该方法会返回一个空值。

这个方法和 java.lang.Class 的 getResource 方法不完全相同。java.lang.Class 的 getResource 方法通过装载类来寻找资源。而这个方法允许服务器产生环境变量给任何资源的任何 Servlet，而不必依赖于装载类、特定区域等等。

```
public RequestDispatcher getRequestDispatcher(  
    String uripath);
```

如果这个指定的路径下能够找到活动的资源（例如一个 Servlet，JSP 页面，CGI 等等）就返回一个特定 URL 的 RequestDispatcher 对象，否则，就返回一个空值，Servlet 引擎负责用一个 request dispatcher 对象封装目标路径。这个 request dispatcher 对象可以用来完成请求的传送。

```
public void log(String msg);
```

```
public void log(String msg, Throwable t);
```

```
public void log(Exception exception, String msg); // 此用法将被取消
```

写指定的信息到一个 Servlet 环境对象的 log 文件中。被写入的 log 文件由 Servlet 引擎指定，但是通常这是一个事件 log。当这个方法被一个异常调用时，log 中将包括堆栈跟踪。

## 6.3 ServletContext 功能

### 6.3.1 只读访问上下文初始化参数

用于上下文初始化参数的部署描述符：

```
<web-app>
  <display-name>Servlet Context Example</display-name>
  <description>
    This Web Application demonstrates application-scoped variables
  </description>
  <context-param>
    <param-name>catalogFileName</param-name>
    <param-value>/WEB-INF/catalog.txt</param-value>
  </context-param>
</web-app>
```

在代码中访问上下文初始化参数：

```
ServletContext context = this.getServletContext();
String catalogFileName = context.getInitParameter("catalogFileName");
```

### 6.3.2 只读访问访问文件资源

- ServletContext 对象提供只读访问文件资源的功能
- getResourceAsStream 方法：返回 InputStream 对象

在代码中访问资源：

```
InputStream is=context.getResourceAsStream("a.txt");
DataInputStream dis=new DataInputStream(is);
String s;
while((s=dis.readLine())!=null){
    System.out.println(s);
}
```

### 6.3.3 写 Web 应用日志文件

- ServletContext 对象只提供对日志文件的写访问：
  - log(String) 方法将一个消息写到日志文件中
  - log(String, Throwable) 方法将一个消息和异常或错误的栈跟踪写到日志文件中
- Web 容器必须支持每个 Web 应用使用一个独立的日志文件
- 在代码中写日志文件：

```
context.log("The ProductList has been initialized.");
```

### 6.3.4 访问共享的运行时属性

- ServletContext 对象提供对所有 servlet 共享属性的读写访问：

getAttribute 方法

setAttribute 方法

- 在代码中设置应用范围属性：

```
context.setAttribute("catalog", catalog);
```

- 在代码中获取属性：

```
ProductList catalog = (ProductList) context.getAttribute("catalog");
```

### 6.3.5 定位资源

## 6.4 Web 应用的生命周期

- Web 容器启动时，初始化每个 Web 应用
- Web 容器关闭时，销毁每个 Web 应用
- 可以创建“监听器”对象触发这些事件

Web 应用生命周期监听器 API

可以创建 `ServletContextListener` 接口的实现监听 Web 应用生命周期事件

- `ServletContextListener` 接口通常用于初始化 web 应用的状态、共享资源

- `ServletContextEvent`:

```
getServletContext() : ServletContext
```

- `ServletContextListener` (回调方法)

```
contextInitialized(event)
```

```
contextDestroyed(event)
```



## 7 会话管理

### 7.1 概述

在计算机中，尤其是在网络应用中，Session 称为“会话”。

Session 直接翻译成中文比较困难，一般都译成时域。在计算机专业术语中，Session 是指一个终端用户与交互系统进行通信的时间间隔，通常指从注册进入系统到注销退出系统之间所经过的时间。

具体到 Web 中的 Session 指的就是用户在浏览某个 Web 应用时，从进入 Web 应用到浏览器关闭所经过的这段时间，也就是用户浏览这个 Web 应用所花费的时间。因此从上述的定义中我们可以看到，Session 可以理解为一个特定的时间概念。

需要注意的是，一个 Session 的概念需要包括特定的客户端，特定的服务器端以及不中断的操作时间。A 用户和 C 服务器建立连接时所处的 Session 同 B 用户和 C 服务器中建立连接时所处的 Session 是两个不同的 Session。

为什么要引入 Session 的概念呢？

HTTP 协议本身是无状态的，这与 HTTP 协议本来的目的是相符的，客户端只需要简单的向服务器请求下载某些文件，无论是客户端还是服务器都没有必要纪录彼此过去的行为，每一次请求与应答都是独立的，好比一个顾客和一个自动售货机或者一个普通的（非会员制）大卖场之间的关系一样。

然而实际的需求又需要一种状态，比如，当用户第一次登录的时候，用户的请求带着用户和密码发送到服务器端，服务器验证通过。当用户进行下一步操作后，第二次请求带着数据再次来到服务器端，因为 HTTP 协议是无状态的，HTTP 服务器会忘记以前的请求，没有凭证说明用户已经登录过，因此需要用户再次提供用户名和密码。这样操作起来就很麻烦。所以需要一种机制或范围，记录用户的状态（数据），当用户再次发送请求时，可以到这个范围内的到用户的信息，而且这个范围是安全的，只能由创建它的用户访问，其他的用户是不能访问它的。

Web 容器提供了一种机制，**存储特定用户的会话信息，为每个用户保存一个“会话对象”**。总结一下：**在服务器端用来存储特定客户端的状态（数据）的机制叫会话，即 session。**

这里用一个形象的比喻来解释 session 的工作方式。假设 Web Server 是一个商场的存包处，HTTP Request 是一个顾客，第一次来到存包处，管理员把顾客的物品存放在某一个柜子里面（这个柜子就相当于 Session），然后把一个号码牌交给这个顾客，作为取包凭证（这个号码牌就是 SessionID）。顾客（HTTP Request）下一次来的时候，就要把号码牌（Session ID）交给存包处（Web Server）的管理员。管理员根据号码牌（SessionID）找到相应的柜子（Session），根据顾客（HTTP Request）的请求，Web Server 可以取出、更换、添加柜子（Session）中的物品，Web Server 也可以让顾客（HTTP Request）的号码牌和号码牌对应的柜子（Session）

失效。顾客（HTTP Request）的忘性很大，管理员在顾客回去的时候（HTTP Response）都要重新提醒顾客记住自己的号码牌（Session ID）。这样，顾客（HTTP Request）下次来的时候，就又带着号码牌回来了。

我们可以看到，SessionID 实际上是在客户端和服务端之间通过 HTTP Request 和 HTTP Response 传来传去的。SessionID 就是 Web Server 识别 Session 的主要依据。

## 7.2 会话 API

定义：

```
public interface HttpSession
```

这个接口被 Servlet 引擎用来实现在 HTTP 客户端和 HTTP 会话两者的关联。这种关联可能在多外连接和请求中持续一段给定的时间。session 用来在无状态的 HTTP 协议下越过多个请求页面来维持状态和识别用户。

一个 session 可以通过 cookie 或重写 URL 来维持。

主要方法：

- 1、 **public Object getAttribute(String name);**
- 2、 **public Enumeration getAttributeNames();**
- 3、 **public void setAttribute(String name, Object object);**
- 4、 **removeAttribute (String name) ;**

取消给定名字的对象在 session 上的绑定。如果未找到给定名字的绑定的对象，这个方法什么也不做。这时会调用 HttpSessionBindingListener 接口的 valueUnbound 方法。

当 session 无效后再调用这个方法会抛出一个 IllegalStateException。

- 5、 **public void invalidate();**

这个方法会终止这个 session。所有绑定在这个 session 上的数据都会被清除。并通过 HttpSessionBindingListener 接口的 valueUnbound 方法发出通告。

- 6、 **public boolean isNew();**

返回一个布尔值以判断这个 session 是不是新的。如果一个 session 已经被服务器建立但是

还没有收到相应的客户端的请求，这个 session 将被认为是新的。这意味着，这个客户端还没有加入会话或没有被会话公认。在他发出下一个请求时还不能返回适当的 session 认证信息。

7、 **public int setMaxInactiveInterval(int interval);**

设置一个秒数，这个秒数表示客户端在不发出请求时，session 被 Servlet 引擎维持的最长时间。

8、 **public HttpSessionContext getSessionContext();**

返回 session 在其中得以保持的环境变量。这个方法和其他所有 HttpSessionContext 的方法一样被取消了。

9、 **public String getId();**

返回分配给这个 session 的标识符。一个 HTTP session 的标识符是一个由服务器来建立和维持的唯一的字符串。

10、 **public long getCreationTime();**

返回建立 session 的时间，这个时间表示为自 1970-1-1 日（GMT）以来的毫秒数。

11、 **public long getLastAccessedTime();**

返回客户端最后一次发出与这个 session 有关的请求的时间，如果这个 session 是新建立的，返回-1。这个时间表示为自 1970-1-1 日（GMT）以来的毫秒数。

12、 **public int getMaxInactiveInterval();**

返回一个秒数，这个秒数表示客户端在不发出请求时，session 被 Servlet 引擎维持的最长时间。在这个时间之后，Servlet 引擎可能被 Servlet 引擎终止。如果这个 session 不会被终止，这个方法返回-1。

当 session 无效后再调用这个方法会抛出一个 IllegalStateException。

13、 **public Object getValue(String name);**

返回一个以给定的名字绑定到 session 上的对象。如果不存在这样的绑定，返回空值。

当 session 无效后再调用这个方法会抛出一个 IllegalStateException。

14、 **public String[] getValueNames();**

以一个数组返回绑定到 session 上的所有数据的名称。

当 session 无效后再调用这个方法会抛出一个 `IllegalStateException`。

15、 **`public void putValue(String name, Object value);`**

以给定的名字，绑定给定的对象到 session 中。已存在的同名的绑定会被重置。这时会调用 `HttpSessionBindingListener` 接口的 `valueBound` 方法。

当 session 无效后再调用这个方法会抛出一个 `IllegalStateException`。

## 7.3 会话对象的使用

### 1、检索会话对象

```
HttpSession session = request.getSession();
```

### 2、存储会话属性

```
session.setAttribute("league", league);
```

Session 的典型应用是存放用户的 Login 信息，如用户名，密码，权限角色等信息，应用程序（如 Email 服务、网上银行等系统）根据这些信息进行身份验证和权限验证。

### 3、访问会话属性

```
HttpSession session = request.getSession();
```

```
League league = (League) session.getAttribute("league");
```

```
Player player = (Player) session.getAttribute("player");
```

### 4、销毁会话

#### 1. 可使用部署描述符控制所有会话的生命周期：

```
<session-config>
```

```
<session-timeout>10</session-timeout>
```

```
</session-config>
```

## 2. 控制特定会话对象的生命周期—HttpSession 接口:

```
invalidate( )

getCreationTime( ) :long

getLastAccessedTime() :long

getMaxInactiveInterval() :int

setMaxInactiveInterval(int)
```

会话对象由同一个应用中的多个 servlet 共享

会话对象不能被同一个 Web 容器中的多个 Web 应用共享

使用 invalidate 方法销毁一个会话，可能引起其它 servlet 冲突

## 7.4 使用 Cookie 的会话管理

### 1、HTTP 允许一个 Web 服务器在客户机器上存储信息:

- Cookie 在 Web 服务器的响应中发出
- Cookie 存储在客户的计算机上，Session 存储在服务器端。
- Cookie 的内容主要包括：名字，值，过期时间，路径和域。
- 域可以指定某一个域，比如 http://www.javakc.com，路径就是跟在域名后面的 URL 路径，比如/或者/foo 等等，路径与域合在一起就构成了 Cookie 的作用范围。
- 如果不设置过期时间，则表示这个 Cookie 的生命期为浏览器会话期间，只要关闭浏览器窗口，Cookie 就消失了。
- 生命期为浏览器会话期的 Cookie 被称为会话 Cookie。会话 Cookie 一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规范规定的。如果设置了过期时间，浏览器就会把 Cookie 保存到硬盘上，关闭后再次打开浏览器，这些 Cookie 仍然有效直到超过设定的过期时间。

- 该域（及路径）的所有 Cookie 在每一个请求中均发送给 Web 服务器。
- Cookie 具有生命周期，且在生命周期的最后由客户浏览器清除。

## 2、Cookie API

- HttpServletResponse 接口：

```
addCookie(Cookie)
```

- HttpServletRequest 接口：

```
getCookies() : Cookie[]
```

- Cookie 类：

定义：

```
public class Cookie implements Cloneable
```

这个类描述了一个 cookie，有关 cookie 的定义你可以参照 Netscape Communications Corporation 的说明，也可以参照 RFC 2109。

构造函数：

```
public Cookie(String name, String value);
```

用一个 name-value 对定义一个 cookie。这个 name 必须能被 HTTP/1.1 所接受。

以字符\$开头的 name 被 RFC 2109 保留。给定的 name 如果不能被 HTTP/1.1 所接受，该方法抛出一个 IllegalArgumentException。

## 3、使用 Cookies

- 通过将 Cookies 加到响应对象中，可将其存储在客户计算机上：

```
String name = request.getParameter("firstName");

Cookie c = new Cookie("yourname", name);

c.setMaxAge(24*60*60);

response.addCookie(c);
```

- 可从请求对象中检索 Cookies:

```
Cookie[] arr=request.getCookies();

for(Cookie c:arr){

    System.out.println("c.key="+c.getName()+" ,value="+c.getValue());

}
```

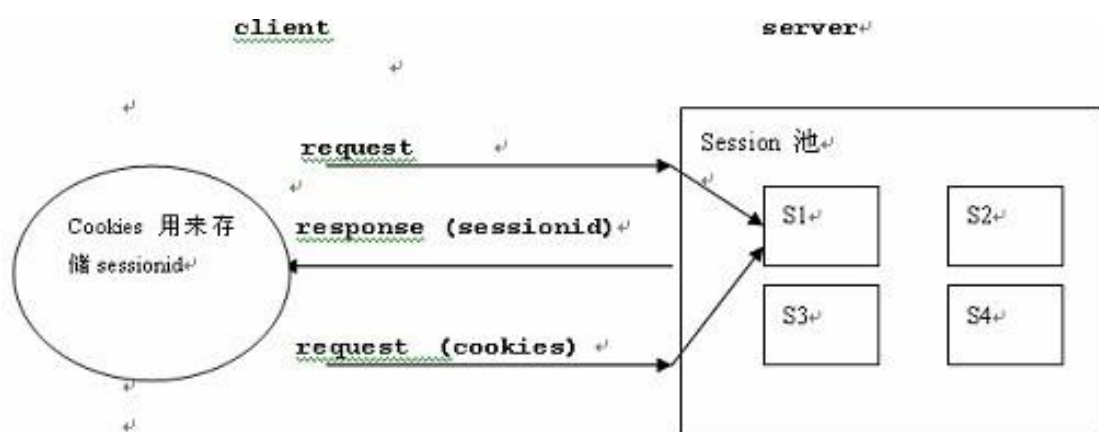
#### 4、Cookie 的会话管理

❖ 过程:

- Web 容器将一个 JSESSIONID Cookie 发送给客户
- JSESSIONID Cookie 在随后的所有请求中发送给服务器

❖ Cookie 机制是缺省的 HttpSession 策略

- 在 servlet 中利用此会话策略编码没有特别之处
- 某些用户会关闭浏览器的 Cookie



当客户端第一次请求时，服务器创建一个 Session 与 Request 绑定，用响应对象 Response 来返回 SessionId 放到客户端的 Cookie 中存储下来，下次在发送请求时，直接根据 SessionId 来检索服务器的会话（每次请求都会将所有的 Cookie 带到服务器端）

## 8 JSP 简介

### 8.1 JSP 概述

- JSP 页面允许在标准 HTML 页面中包含 Java 代码
- JSP 技术的目标是支持表现和业务逻辑的分离
  - ❖ Web 设计人员可以独立设计和修改页面<前台>
  - ❖ Java 平台的程序员可以独立编写业务逻辑代码<后台>

HelloServlet代码:

```
private static final String DEFAULT_NAME = "World";
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) throws
IOException{
    String name = request.getParameter("name");
    if ( (name == null) || (name.length() == 0) ) {
        name = DEFAULT_NAME;
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Hello Servlet</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY BGCOLOR='white'>");
    out.println("<B>Hello, " + name + "</B>");
    out.println("</BODY>");
    out.println("</HTML>");
    out.close();
}
```

与HelloServlet等价的hello.jsp页面:

```
<%! private static final String DEFAULT_NAME = "World"; %>
<HTML>
<HEAD>
<TITLE>Hello JavaServer Page</TITLE>
</HEAD>
<%-- Determine the specified name (or use default) --%>
<%
    String name = request.getParameter("name");
```



```
        if ( (name == null) || (name.length() == 0) ) {  
            name = DEFAULT_NAME;  
        }  
%>  
<BODY BGCOLOR='white'>  
<B>Hello, <%= name %></B>  
</BODY>  
</HTML>
```



## 8.2 JSP 页面的处理过程

### 第一步:

- ❖ 请求进入 Web 容器，如果第一次访问 jsp 页面，JSP Parser 将 JSP 页面翻译成 Servlet 代码；如果不是第一次访问执行第四步。

### 第二步:

- ❖ 编译 Servlet 代码，并将编译过的类文件装入 Web 容器（JVM）环境

### 第三步:

- ❖ Web 容器为 JSP 页面创建一个 Servlet 类实例，并执行 jspInit 方法

### 第四步:

- ❖ Web 容器为该 JSP 页面调用 Servlet 实例的 \_jspService 方法；将结果发送给用户

### ❖ 开发及部署 JSP 页面

#### 开发期间将 JSP 文件放在 web 目录下

#### 部署时将其拷贝到如下层次中:

```
webapps/  
  project/  
    WEB-INF/  
      web.xml  
      classes/  
      lib/  
    index.html  
    hello.jsp
```

```
date.jsp
```

## 8.3 JSP 脚本元素

JSP 脚本元素

❖ 脚本元素`<% %>`由 JSP 引擎处理；所有其它文本、脚本以外的元素均被作为响应的一部分

```
<HTML>
<%-- scripting element --%>
</HTML>
```

JSP 脚本元素分类

❖ 五类脚本元素：

- ☞ 注释：           `<%-- comment --%>`
- ☞ 指令标签：       `<%@ directive %>`
- ☞ 声明标签：       `<%! Decl %>`
- ☞ 脚本标签：       `<% code %>`
- ☞ 表达式标签：     `<%=expr %>`

### 1、注释

☞ JSP 页面注释

`<%--` JSP 注释只在 JSP 代码中可见，不显示在 servlet 代码或响应中。

`--%>`

☞ Java 注释

`<%`

`/*` Java 注释显示在 servlet 代码中，不显示在响应中

\* /

%>

## 2、指令标签

🔗 目的：指令标签影响 JSP 页面的翻译阶段

🔗 语法：

```
<%@ DirectiveName [ attr=" value"]* %>
```

🔗 例：

```
<%@page import="java.util.*" %>
```

```
<%@ page session="false" %>
```

```
<%@ include file="incl/copyright.html" %>
```

```
<%@ taglib prefix="c" uri="http://java.sum.com/jsp/jstl/core"%>
```

## 3、声明标签

🔗 目的：声明标签允许 JSP 页面开发人员包含类级声明

🔗 语法：

```
<%! JavaClassDeclaration %>
```

🔗 例：

```
<%! public static final String DEFAULT_NAME = "World"; %>
```

```
<%! public String getName(HttpServletRequest request) {
```

```
    return request.getParameter("name");
```

```
}
```

```
%>
```

```
<%! int counter = 0; %>
```

#### 4、脚本标签

🔗 目的：脚本标签允许 JSP 页面开发人员在 `_jspService` 方法中包含任意的 Java 代码

🔗 语法：

```
<% JavaCode %>
```

🔗 例：

```
<% int i = 0; %>
```

```
<% if ( i > 10 ) { %>
```

```
    I is a big number.
```

```
<% } else { %>
```

```
    I is a small number
```

```
<% } %>
```

#### 5、表达式标签

🔗 目的：表达式标签封装 Java 运行时的表达式，其值被送至 HTTP 响应流

🔗 语法：

```
<%= JavaExpression %>
```

🔗 例：

```
<B>Ten is <%= (2 * 5) %></B>
```

Thank you, <I><%= name %></I>, for registering for the soccer league.

The current day and time is: <%= new java.util.Date( ) %>

### 将 jsp 翻译成 Servlet 的过程:

❖ 指令标签 根据指令内容直接翻译到 Servlet 相应位置

❖ 声明标签

定义的变量翻译成 Servlet 的属性;

声明的方法翻译成 Servlet 的方法;

定义的类翻译成 Servlet 的内部类;

❖ 脚本标签 依照先后顺序, 翻译到\_jspService 方法里

❖ 表达式标签 封装 java 运行时表达式, 其值直接到相应位置。

❖ 注释 不翻译到 Servlet 中。

除上述脚本元素以外所有代码, 按照先后顺序全部视为文本, 翻译到\_jspService 方法里, out.print 直接输出。

## 8.4 Jsp 页面中的隐含对象

在 jsp 中存在 9 个隐含变量, 这些对象不经过显示声明直接使用, 也不需要专门的代码创建其实例, 在程序设计中可以直接使用这些对象。

❖ \_jspService 方法中预定义的变量:

➤ request 与请求相关的 HttpServletRequest 对象

➤ response 与送回浏览器的响应相关的 HttpServletResponse 对象

➤ out 与响应的输出流相关的 JspWriter 对象

➤ session 与给定用户请求会话相关的 HttpSession 对象, 该变量只在 JSP 页面参与一个 HTTP 会话时有意义

- application      用于 Web 应用的 ServletContext 对象
- config            与该 JSP 页面的 servlet 相关的 ServletConfig 对象配置
- pageContext      该对象封装了一个 JSP 页面请求的环境
- page              该变量与 Java 编程语言中的 this 变量等价
- exception        由其它 JSP 页面抛出的 Throwable 对象，该变量只在“JSP 错误页面”中可用

### **request - javax.servlet.http.HttpServletRequest**

request 对象包含所有请求的信息，如请求的来源、标头、cookies 和请求相关的参数值等。

❖ Object `getAttribute( String name ) ;`

返回由 name 指定的属性值，该属性不存在时返回 null。

❖ Enumeration `getAttributeNames() ;`

返回 request 对象的所有属性名称的集合。

❖ Cookie[] `getCookies() ;`

返回客户端所有的 Cookie 的数组。

❖ String `getHeader( String name ) ;`

返回指定名称的 HTTP 头的信息。

❖ Enumeration `getHeaderNames() ;`

返回所有 HTTP 头的名称的集合

❖ String `getParameter( String name ) ;`

获取客户端发送给服务器端的参数值

❖ String[] `getParameterValues( String name ) ;`

获得请求中指定参数的所有值。

**response - javax.servlet.http.HttpServletResponse**

response 对象主要将 JSP 容器处理后的结果传回到客户端。

A) void addCookie( Cookie cookie ) ;

添加一个 Cookie 对象，保存客户端信息。

B) void addHeader( String name, String value ) ;

添加一个 HTTP 头，覆盖同名的旧 HTTP 头。

C) void sendRedirect( String location ) ;

把响应发送到另外一个位置进行处理，重新定向的页面可以是任意的 URL

D) void setContentType( String type ) ;

设置响应的类型。

E) void setBufferSize( int size ) ;

设置以 kb 为单位的缓冲区大小。

**out - javax.servlet.jsp.jspWriter**

out 对象用于把结果输出到网页上。

❖ void print( String s ) ;

将指定类型的数据输出到 Http 流，不换行。

❖ void println( String s ) ;

将指定类型的数据输出到 Http 流，并输出一个换行符。

❖ void clear() ;

清除输出缓冲区的内容，但是不输出到客户端。

❖ void clear() ;

清除输出缓冲区的内容，但是不输出到客户端。

❖ void newLine() ;

输出一个换行字符。

❖ boolean isAutoFlush() ;

是否自动刷新缓冲区。

### **session - javax.servlet.http.HttpSession**

session 对象表示目前个别用户的会话状态，用来识别每个用户。

❖ void setAttribute( String name, String value ) ;

设置指定名称的 session 属性值。

❖ Object getAttribute( String name ) ;

获取与指定名字相关联的 session 属性值。

❖ Enumeration getAttributeNames() ;

取得 session 内所有属性的集合。

❖ String getId() ;

取得 session 标识。

❖ int getMaxInactiveInterval( int interval ) ;

返回总时间，以秒为单位，表示 session 的有效时间(session 不活动时间)。-1 为永不过期。

❖ void removeAttribute( String name ) ;

移除指定名称的 session 属性。

### **pageContext - javax.servlet.jsp.PageContext**

pageContext 对象存储本 JSP 页面相关信息，如属性、内建对象等。

1.void setAttribute( String name, Object value, int scope ) ;

void setAttribute( String name, Object value ) ;

在指定的共享范围内设置属性。

2.Object getAttribute( String name, int scope ) ;

Object getAttribute( String name ) ;

取得指定共享范围内以 name 为名字的属性值。

3.HttpSession getSession() ;



取得页面的 session 对象。

```
4. ServletRequest getRequest() ;
```

取得页面的 request 对象。

```
5. ServletResponse getResponse() ;
```

取得页面的 response 对象。

### **application - javax.servlet.ServletContext**

application 用于 Web 应用的 ServletContext 对象。

❖ `String getInitParameter( String name ) ;`

返回由 name 指定的 application 属性的初始值。

❖ `Object getAttribute( String name ) ;`

返回由 name 指定的 application 属性。

❖ `Enumeration getAttributes() ;`

返回所有的 application 属性。

❖ `void setAttribute( String name, Object value ) ;`

设定指定的 application 属性的值。

❖ `InputStream getResourceAsStream( String path ) ;`

返回一个由 path 指定位置的资源的 InputStream 对象实例。

❖ `void log( String msg ) ;`

把指定的信息写入 servlet log 文件。

### **config - javax.servlet.ServletConfig**

config 对象用来存放 Servlet 初始的数据结构。

❖ `String getInitParameter( String name ) ;`

返回名称为 name 的促使参数的值。

❖ `Enumeration getInitParameters() ;`

返回这个 JSP 所有的促使参数的名称集合。

- ❖ `ServletContext getContext()` ;

返回执行者的 `servlet` 上下文。

- ❖ `String getServletName()` ;

返回 `servlet` 的名称。

### **page - javax.servlet.jsp.HttpJspPage**

`page` 对象代表 JSP 对象本身，或者说代表编译后的 `servlet` 对象，可以用来取用它的方法和属性。

### **exception - java.lang.Throwable**

`isErrorPage="true"`后，才可以在本页面使用 `exception` 对象。

- ❖ `String getMessage()`

取得错误提示信息。

- ❖ `void printStackTrace()` ;

`void printStackTrace( printStream s ) ;`

`void printStackTrace( printWriter s ) ;`

打印出 `Throwable` 及其 `call stack trace` 信息。

## **8.5 page 指令**

➤ `page` 指令用于修改 JSP 页面的整体翻译

- ❖ 例如，可以声明从 JSP 页面生成 `servlet` 代码所需要的 `Date` 类：

```
<%@ page import="java.util.Date" %>
```

➤ 可以使用多个页面指令，但任一给定属性只能声明一次

➤ 可以将 `page` 指令放在 JSP 文件中的任何地方

➤ `page` 指令定义一组与页面相关的属性，在翻译期间与 Web 容器通信

- ❖ `language` 定义页面中使用的脚本语言。“`java`”是当前定义的唯一值，且是缺省值

- ❖ `extends` 定义了由 JSP 页面生成的 `servlet` 类的父类名（完全类名）。不要使用该属性
- ❖ `import` 定义了一组 `servlet` 类定义必须导入的类和包，值是一个由逗号分隔的完全类名或包的列表。例如：

```
import="java.sql.Date,java.util.*,java.text.*"
```

- ❖ `session` 定义 JSP 页面是否参与 HTTP 会话，值可以为 `true`（缺省）或 `false`
- ❖ `buffer` 定义用于输出流（`JspWriter` 对象）的缓冲区大小，值可以为 `none` 或 `Nkb`，缺省为 `8KB` 或更大。例如：

```
buffer="8kb"或 buffer="none"
```

- ❖ `autoFlush` 定义缓冲满时，缓冲输出是否自动完成 (`flush`) 或抛出异常。缺省值为 `true`。
- ❖ `isThreadSafe` 允许 JSP 开发人员声明 JSP 页面是否线程安全
- ❖ `info` 定义一个有关 JSP 页面的信息串
- ❖ `errorPage` 指明由另一个 JSP 页面处理所有该 JSP 页面运行时抛出的异常。例如，

```
errorPage="error.jsp" （相对于当前层次）
```

```
errorPage="/error/formErrors.jsp" （相对于 context root）
```

- ❖ `isErrorPage` 定义 JSP 页面为其它 JSP 页面 `errorPage` 属性的目标，值为 `true` 或 `false`（缺省）。所有“是错误页面”的 JSP 页面自动具有访问 `exception` 隐含变量的权限
- ❖ `contentType` 定义输出流的 MIME 类型，缺省为 `text/html`。
- ❖ `charset` 设置字符集
- ❖ `pageEncoding` 定义输出流的字符编码，缺省为 `ISO-8859-1`

## 8.6 JSP 异常处理

- 不能将 JSP 页面包装到一个 try-catch 块中
- JSP 规范提供了一种机制，可在主页面中指定错误页面。
- Web 容器捕获异常，并将其转发到错误处理页面

声明错误页面

throws\_error.jsp 声明一个错误页面：

```
<%@ page session="false" errorPage="error/ExceptionPage.jsp"%>

<!-- This page will cause an "divide by zero" exeception --%>

<HTML>

<HEAD>

<TITLE>Demonstrate Error Pages</TITLE>

</HEAD>

<BODY BGCOLOR='white'>

<OL>

<%   for ( int i=10; i > -10; i-- ) {   %>

<LI> <%= 100/i %>

<%   }   %>

</OL>

</BODY>

</HTML>
```

编写错误页面：

ExceptionPage.jsp 是一个错误页面

```
<%@ page session="false" isErrorPage="true"
import="java.io.PrintWriter" %>

<% String expTypeFullName

        = exception.getClass().getName();

        String expTypeName =

                expTypeFullName.substring(

                        expTypeFullName.lastIndexOf(".") + 1);

        String request_uri = (String)

                request.getAttribute("javax.servlet.error.request_uri");

%>

<HTML>

<HEAD>

        <TITLE>JSP Exception Page</TITLE>

</HEAD>

</HTML>
```

## 8.7 调试 JSP 页面

开发 JSP 页面时的三种基本错误:

- ❖ 翻译期间 (解析错误)
- ❖ 编译期间 (servlet 代码中的错误)
- ❖ 运行期间 (逻辑错误)

## 9 使用 Model 1 架构开发 Web 应用 (JSP+JavaBeans)

### 9.1 使用 Model 1 架构

- Model 1 架构使用 JSP 页面处理 Web 应用的视图和控制部分
- 该架构中不使用 Servlet
- 例：JSP 页面充当 form 和处理者
  - ❖ 对页面的首次调用使用 GET 请求
  - ❖ HTML form 使用 POST 请求激活相同的页面

### 9.2 JavaBeans 组件

- JavaBeans 组件是一个 Java 类，特征如下：
  - ❖ 无 public 实例变量
  - ❖ 使用 get 和 set 方法定义属性
  - ❖ 一个无参构造方法

### 9.3 JSP 标准动作

- JSP 页面中使用类似于 XML 的标签表示运行时的动作
- 语法：

```
<jsp: action [ attr=" value"]* />
```

- ❖ 例：

```
<jsp:useBean id="beanName" class="BeanClass" scope="request" />
```

```
<jsp:setProperty name="beanName" property="prop1" value="val" />
```

```
<jsp:getProperty name="beanName" property="prop1" />
```

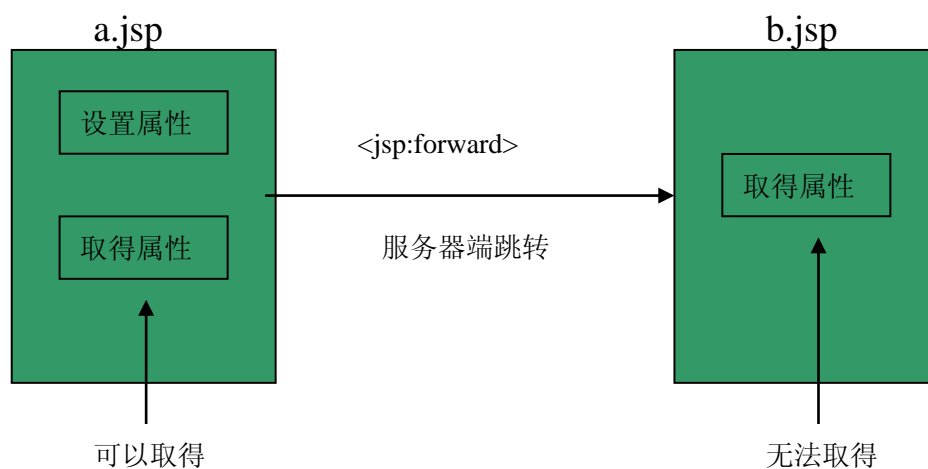
- 标准动作减少了 JSP 页面中的脚本元素
- 标准动作标签名总以 jsp 前缀开始

- Bean 与范围

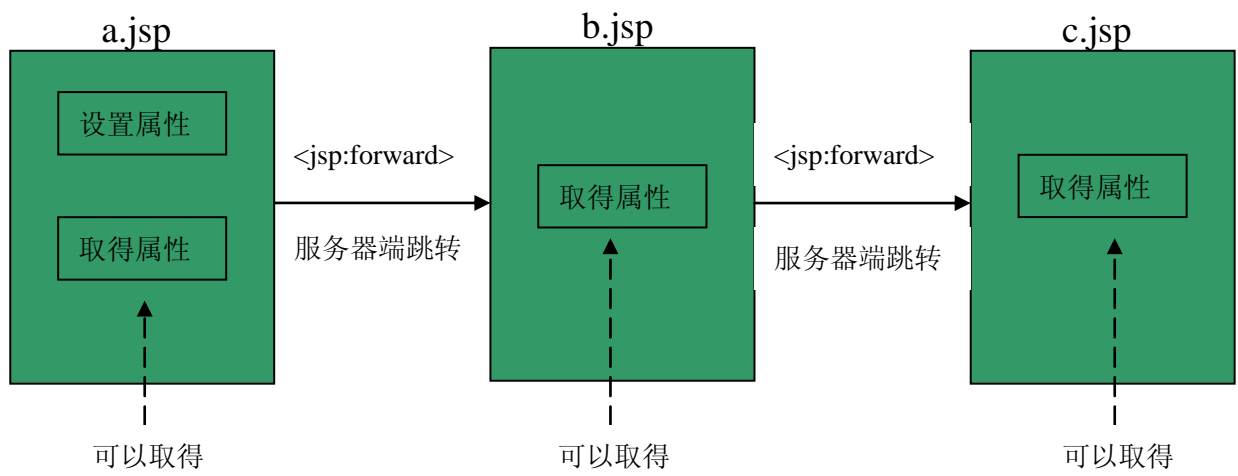
■ Bean 的四种存在范围：

- ◆ 页面范围 (page)：在一次页面范围中，创建并被存储在 PageContext 对象中
- ◆ 请求范围 (request)：在一次服务器请求范围中，创建并被存储在 ServletRequest 对象中
- ◆ 会话范围 (session)：在一次会话范围中，创建并被存储在 HttpSession 对象中
- ◆ 应用程序范围(application)：在整个 web 应用范围中，存储在 ServletContext 对象中

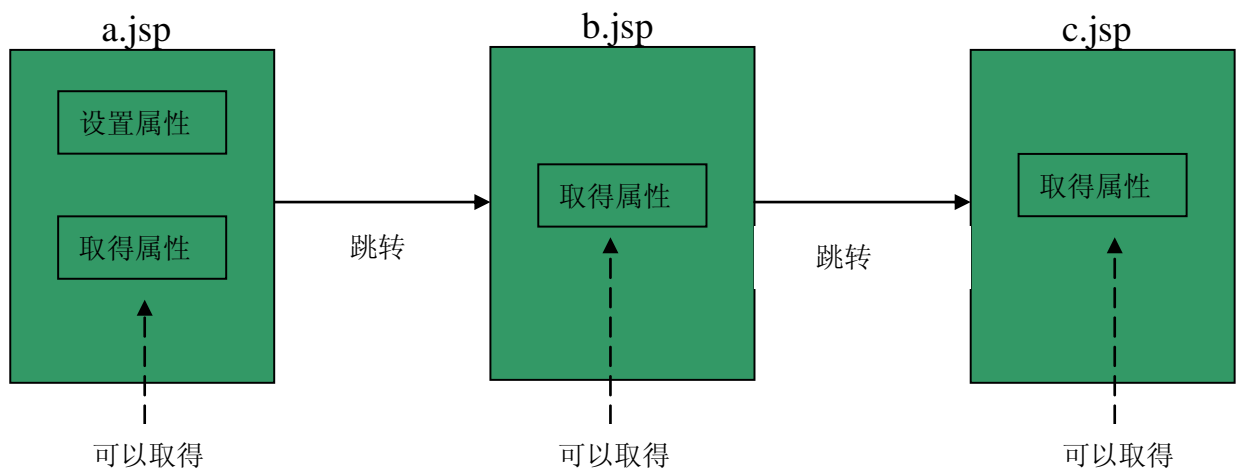
页面范围：



请求范围：

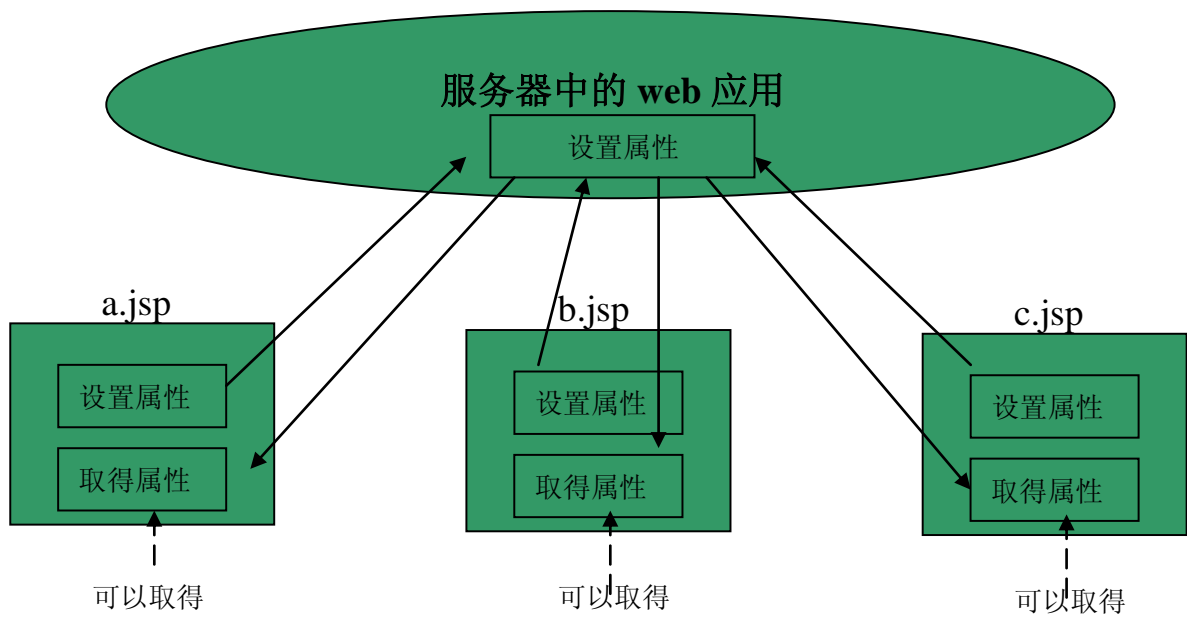


会话范围：



应用程序范围：





注：设置属性是有内存开销的，在 application 范围内设置过多的属性，或每一个 session 保存过多的对象，web 应用的性能会降低。

如果能用 request 就不要使用 session，能使用 session 就不适用 application。

使用 java 代码也可以在这四种属性范围中设置属性。

#### ➤ jsp:useBean 动作处理过程

jsp:useBean →

使用 id 声明变量 →

试图在指定的范围内查找对象 →

→ [没找到] →

创建一个类的实例 →

执行 useBean 标签体初始化对象

→ [找到] →

将对象转换为类指定的类型

➤ jsp:forward 动作

使用脚本代码处理请求时，可用 jsp:forward 动作产生一个不同的视图

```
<%
```

```
if ( request.getMethod().equals("POST") ) {
```

```
    guestBookSvc.processRequest();
```

```
    if ( guestBookSvc.wasSuccessful() ) {
```

```
%>
```

```
        <jsp:forward page="guestBookThankYou.jsp" />
```

```
<% } else { %>
```

```
        <jsp:forward page="guestBookError.jsp" />
```

```
<% } %>
```

## 10 使用 MVC 模式开发 Web 应用

### 10.1 Web 应用的活动

- 初始化所有共享数据和资源
- 服务于每个 HTTP 请求：
  - 验证 HTML form 数据（如果有）
  - 如果数据验证检查不通过，发送错误页面
  - 处理可存储持久信息的数据
  - 如果处理失败，发送一个错误页面
  - 如果处理成功，发送一个响应页面

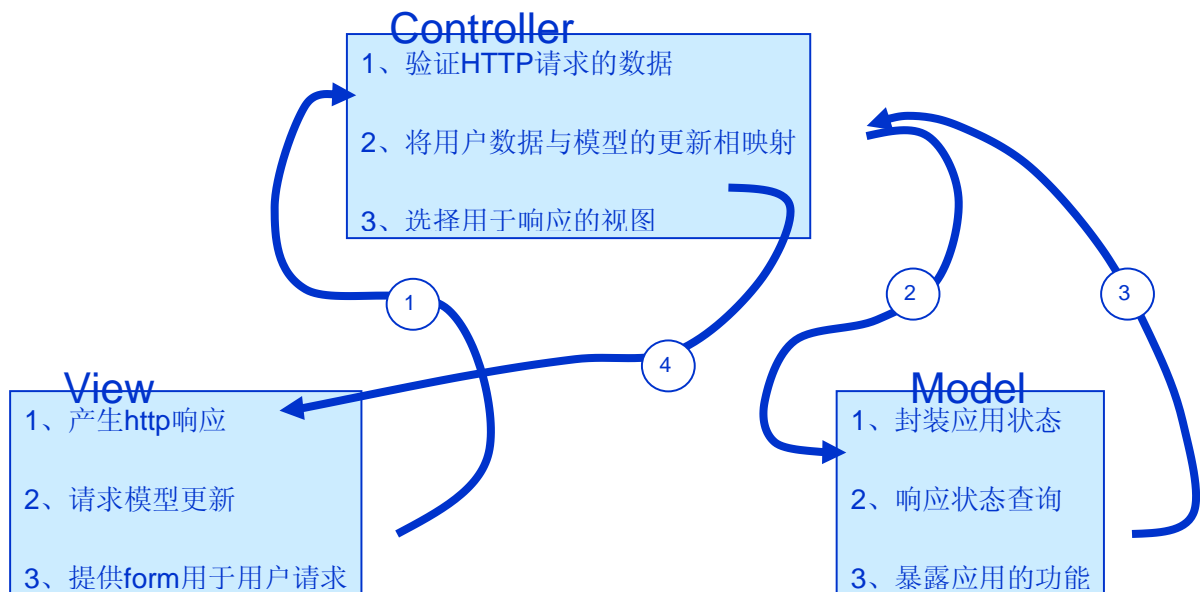
### 10.2 Web 应用的 MVC

- ❖ MVC 是一个框架型设计模式
  - ❖ 本身没有实际的代码（功能）
  - ❖ 它只是说明页面、数据处理如何摆放。
- 
- Model
    - ❖ 封装应用状态（封装应用数据）
    - ❖ 响应状态查询（对数据进行增删改查）
    - ❖ 暴露应用的功能（暴露接口<public>）
  - Controller
    - ❖ 验证 HTTP 请求的数据（收集组织数据）
    - ❖ 将用户数据与模型的更新相映射（调用逻辑层）

- ❖ 选择用于响应的视图（选择下一个界面）

➤ View

- ❖ 产生 HTML 响应（展示数据）
- ❖ 请求模型的更新（人机交互）
- ❖ 提供 HTML form 用于用户请求（收集参数，调用逻辑层 api）



**MVC 的优点:**

低耦合性：视图层和业务层分离

高重用性和可适用性

较低的生命周期成本

快速的部署

可维护性

有利于软件工程化管理

提高软件的健壮性

**MVC 的缺点:**

工作量大，增加工作的复杂性，MVC 不适合小型甚至中等规模的应用程序

## 10.3 使用 Model 2 架构开发 Web 应用程序

Model 2 架构使用 MVC 模式，JSP 页面充当视图，Servlet 充当控制器

Model 2 架构开发要求

➤ Servlet 控制器：

- 验证 HTML form 数据
- 调用模型中的业务服务
- 存储请求（或会话）范围内的域对象
- 选择下一个用户的视图

➤ JSP 页面视图：

- 使用用户界面（在 HTML 中）
- 访问域对象

➤ Model

### 请求分发器

☞ 上下文对象中的分发器：

```
ServletContext context = getServletContext();

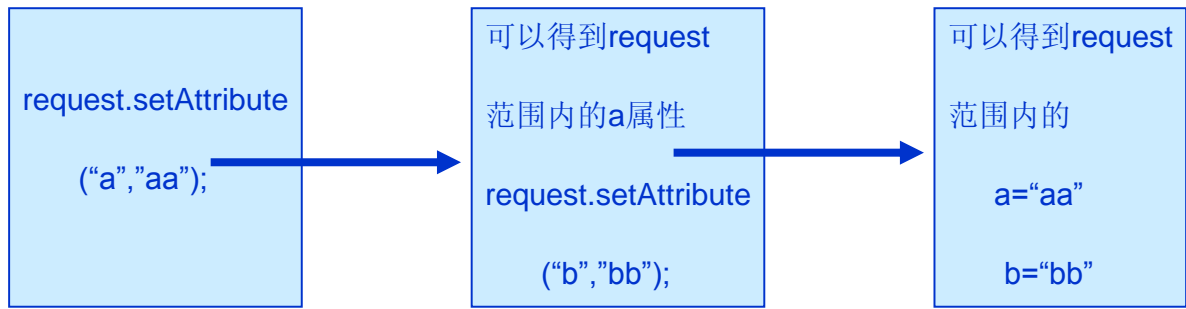
RequestDispatcher servlet = context.getRequestDispatcher("MyServlet");

servlet.forward(request, response);
```

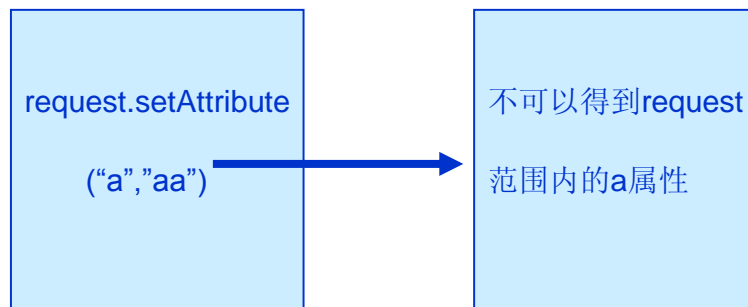
☞ 请求对象中的分发器：

```
RequestDispatcher view =
request.getRequestDispatcher("tools/nails.jsp");

view.forward(request, response);
```



分发器: `request.getRequestDispatcher("url")`



重定向: `response.sendRedirect("url")`

# 11 Web 应用的并发问题

## 11.1 Servlet 的并发

Servlet 并发问题

- 多个同类线程运行，可共享同一个 Servlet 实例
- 共享的数据和资源未合理同步，可能引起数据冲突

并发管理例：

- 将客户数据存储到一个普通文件中
- 通过同步文件写出对象，可消除并发问题

```
synchronized (customerDataWriter) {  
  
    //customerDataWriter.write(name, 0, name.length());  
  
}
```

## 11.2 属性范围

1、Web 应用中有六种属性范围：

- 🔗 局部变量（页面范围）
- 🔗 实例变量
- 🔗 类变量
- 🔗 请求属性（请求范围）
- 🔗 会话属性（会话范围）
- 🔗 上下文属性（应用范围）

2、局部变量

- ☞ 线程安全？ 是
- ☞ 使用：任何对 Servlet 方法的局部处理
- ☞ 局部变量对每个请求线程是唯一的

### 3、实例变量

- ☞ 线程安全？ 否
- ☞ 使用：
  - ❖ 存储的数据涉及给定 Servlet 的多个请求
  - ❖ 存储初始化参数（通常只读）
- ☞ 实例变量在给定 servlet 实例的每个请求间共享

### 4、类变量

- ☞ 线程安全？ 否
- ☞ 使用：存储跨多个 servlet 定义的数据
- ☞ 类变量在每个 servlet 实例的所有请求间共享

### 5、请求范围

- ☞ 线程安全？ 是
- ☞ 使用：将数据从控制器移动到用于表现的视图
- ☞ API：ServletRequest 接口
  - ❖ `getAttributeNames() :Enumeration`
  - ❖ `getAttribute(name:String) :Object`
  - ❖ `setAttribute(name:String, value:Object)`
  - ❖ `removeAttribute(name:String)`

### 6、会话范围

- ☞ 线程安全？ 否



☞ 用户可能有多个浏览器活动，访问同一个 Web 应用

☞ 使用：存储一个 Web 会话中通用的数据

☞ API: HttpSession 接口

❖ `getAttributeNames() :Enumeration`

❖ `getAttribute(name:String) :Object`

❖ `setAttribute(name:String, value:Object)`

❖ `removeAttribute(name:String)`

## 7、应用范围

☞ 线程安全？否

☞ 使用：Web 应用中所有 servlet 共用的资源

☞ API: ServletContext 接口

❖ `getAttributeNames() :Enumeration`

❖ `getAttribute(name:String) :Object`

❖ `setAttribute(name:String, value:Object)`

❖ `removeAttribute(name:String)`

## 11.3 SingleThreadModel 接口

### 1、SingleThreadModel 特点：

☞ 实现 SingleThreadModel (STM) 接口，某一时刻只有一个请求线程执行 service 方法

☞ SingleThreadModel 接口没有要实现的方法

☞ 可使用 STM 作为信号通知 Web 容器 Servlet 类必须特殊处理

### 2、Web 容器实现 STM 的方法

- ☞ 将所有请求排队，一次向 Servlet 实例传送一个请求
- ☞ 创建一个无限大的 Servlet 实例池，其中的每个实例一次处理一个请求
- ☞ 创建固定大小的 Servlet 实例池，其中的每个实例一次处理一个请求。如果有更多的请求到达，超过了池的大小，则某些请求会推迟到其它请求完成后再做

### 3、STM 和并发管理

- ❖ 使用 STM 的 Servlet 可使用多个 servlet 实例，且所有的实例将共享常规的资源
- ❖ Servlet 使用 STM 的主要缺点：
  - ❖ STM 机制的实现是与厂商相关的
  - ❖ 使用“一次一个”方法时，STM 的使用可能会极大地降低性能
  - ❖ 使用“Servlet 池”方法时，STM 的使用不能控制对静态变量的访问
  - ❖ 没有一个方法解决了会话和应用范围属性的并发问题
- ❖ 基于以上原因，不应使用 STM

## 11.4 并发管理指南

- 尽可能只使用局部和请求属性
- 使用 synchronized 语法控制并发
- ❖ 使用常规资源
- ❖ 访问和更新频繁变化的属性

```
HttpSession session = request.getSession();

    synchronized (session) {

        session.setAttribute("count", new Integer(count+1));

    }

HttpSession session = request.getSession();
```

```
synchronized (session) {  
  
    countObj = (Integer) session.getAttribute("count");  
  
}  
  
count = countObj.intValue();
```

❖ 尽可能减少同步块和同步方法的使用

☞ 不要同步整个 doGet 或 doPost 方法

❖ 使用正确设置了线程安全的资源类

## 12 Web 应用的异常处理

### 12.1 概述

#### 12.1.1 HTTP 错误

一个 HTTP 响应可使用 400-500 范围内的状态代码指明一个服务器错误，常见错误如下：

|     |                                    |
|-----|------------------------------------|
| 400 | Bad Request//错误请求                  |
| 401 | Unauthorized//未被认可                 |
| 404 | Not Found//没有创建                    |
| 405 | Method Not Allowed//方法不允许          |
| 415 | Unsupported Media Type//不支持类型      |
| 500 | Internal Server Error//服务器内部错误     |
| 501 | Not Implemented//没有实现              |
| 503 | Service Unavailable 没有获得服务（服务没有到达） |

缺省情况下，Web 浏览器会给用户显示一些信息。该信息通常由浏览器而不是 Web 服务器生成一般的 HTTP 错误页面。

#### 12.1.2 Servlet 异常

Servlet 可抛出一个 `ServletException`，指明异常的发生

```
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException {
    int x = 0;
    int y = 0;
```

```
try {  
    int z = x / y;  
} catch (ArithmeticException ae) {  
    throw new ServletException(ae);  
}  
}
```

Web 容器捕获到这些异常，并发送一个带有状态码 500 的 HTTP 响应，该响应有异常的跟踪信息。

错误和运行时异常也由 Web 容器直接处理。

## 12.2 定制错误页面的方式

声明方式：

使用部署描述符为特定情况（HTTP 错误或 Java 异常）声明错误页面，并由 Web 容器将处理转发到这些页面

编程方式：

直接在 servlet 代码中处理 Java 异常，并将 HTTP 请求转发到所选定的错误页面

## 12.3 声明 HTTP 错误页面

使用 `error-page` 元素声明一个给定 HTTP 状态码的处理器：

```
<error-page>  
  
    <error-code>404</error-code>  
  
    <location>/error/404.html</location>  
  
</error-page>
```

可以声明任意数量的错误页面，但一个给定的状态码只能对应一个页面。

➤ 使用 `exception-type` 元素声明给定 Java 异常的处理器

```
<error-page>
```

```
<exception-type>

    java.lang.ArithmeticException

</exception-type>

<location>/error/ExceptionPage</location>

</error-page>
```

可以声明任意数量的错误页面，但一个给定的异常类型只对应一个页面。

不能使用父类捕获多种异常。

## 12.4 容器处理异常

- 一个 servlet 可抛出 ServletException 指明有异常产生
- 所有检查到的异常必须被捕获，并封装在 ServletException 中
- 编写错误处理 Servlet
  - 应同时覆盖 doGet 和 doPost 方法
- 两个预定义的请求属性：
  - javax.servlet.error.exception：包含原始（被包装）的异常
  - javax.servlet.error.request\_uri：包含原始客户请求的 URI

## 12.5 编程处理异常

- 可编写自己的 servlet 代码捕获所有异常并自己处理，而不是让容器处理
- 使用此技术只可以处理 Java 异常，而非 HTTP 异常
- 步骤：
  - 对可能产生异常的所有代码使用一个 try-catch 块
  - 在 catch 块中使用 RequestDispatcher 将异常转发到一个 servlet 异常处理器
  - （可选）可以设置特定的请求属性
- 声明 servlet 异常处理器，但不提供 URL 映射

```
<servlet>

<servlet-name>ExceptionHandler</servlet-name>

<servlet-class>sl314.web.ExceptionDisplay</servlet-class>

</servlet>
```

➤ 使用 servlet 名获得 RequestDispatcher

```
ServletContext context = getServletContext();

RequestDispatcher errorPage=

                                context.getNamedDispatcher("ExceptionHandler");

request.setAttribute("javax.servlet.error.exception", e);

request.setAttribute("javax.servlet.error.request_uri",

request.getRequestURI());

errorPage.forward(request, response);
```

## 12.6 异常处理方式的选择

### 12.6.1 声明方式处理异常的优缺点:

优点:

- 容易实现

缺点:

- 必须为每个异常处理 servlet 创建 URL 映射
- 不能做多对一映射
- 过于通用

## 12.6.2 编程方式处理异常的优缺点

优点：

- 使处理器代码与控制器更紧密
- 使异常处理更明确（显式处理）
- 处理器可根据环境定制

缺点：

- 需要实现更多的代码



## 13 JSP 重用模板

### 13.1 JSP 代码片段的处理

在主 JSP 页面中包含表现代码片段：

➤ include 指令

➤ jsp:include 标准动作

➤ 使用 include 指令

🔗 目的：将一个代码片段在转换期间（翻译期间）包含在主 JSP 页面的文本中

🔗 语法：

```
<%@ include file=" fragmentURL" %>
```

🔗 例：

```
<!-- START of side-bar -->
```

```
<TD BGCOLOR='#CCCCFF' WIDTH='160' ALIGN='left'>
```

```
<%@ include file="/incl/side-bar.jsp" %>
```

```
</TD>
```

```
<!-- END of side-bar -->
```

(1) 使用 jsp:include 标准动作

🔗 目的：在运行时将一个代码片段包含到 HTTP 响应文本中

🔗 语法：

```
<jsp:include page="fragmentURL" />
```

🔗 例：

```
<!-- START of banner -->
```

```
<jsp:include page="/incl/banner.jsp" />
```

```
<!-- END of banner -->
```

(2) 使用 jsp:param 标准动作

🔗 jsp:include 动作可以使用 jsp:param 动态地接收指定的参数:

```
<!-- START of banner -->

<jsp:include page="/incl/banner.jsp">

<jsp:param name="subTitle" value="Thank You!"/>

</jsp:include>

<!-- END of banner -->
```

- ❖ Include 指令（编译时包括）又叫静态引用，约等于同一个界面，可以共享数据。
- ❖ Include 动作（运行时包括）又叫动态引用，完全是两个界面，不能共享数据。

## 14 过滤器

Filter 技术是 servlet 2.3 新增加的功能。它使用户可以改变一个 request 和修改一个 response。Filter 不是一个 servlet, 它不能产生一个 response, 它能够在 request 到达 servlet 之前预处理 request, 也可以在离开 servlet 时处理 response。一个 filter 包括:

1. 在 servlet 被调用之前截获;
2. 在 servlet 被调用之前检查 servletrequest;
3. 根据需要修改 request 头和 request 数据;
4. 根据需要修改 response 头和 response 数据;
5. 在 servlet 被调用之后截获。

你能够配置一个 filter 到一个或多个 servlet; 单个 servlet 或多个 servlet 能够被多个 filter 使用。几个实用的 filter 包括: 用户辨认 filter, 日志 filter, 审核 filter, 加密 filter, 符号 filter, 能改变 xml 内容的 XSLT filter 等。

一个 filter 必须实现 javax.servlet.Filter 接口并定义三个方法:

1. void setFilterConfig(FilterConfig config) //设置 filter 的配置对象;
2. FilterConfig getFilterConfig() //返回 filter 的配置对象;
3. void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) //执行 filter 的工作。

服务器每次只调用一次 **setFilterConfig** 方法准备 filter 的处理; 调用 **doFilter** 方法多次以处理不同的请求。FilterConfig 接口有方法可以找到 filter 名字及初始化参数信息。服务器可以设置 FilterConfig 为空来指明 filter 已经终结。

每一个 filter 从 doFilter() 方法中得到当前的 request 及 response。在这个方法里, 可以进行任何的针对 request 及 response 的操作。(包括收集数据, 包装数据等)。filter 调用 chain.doFilter() 方法把控制权交给下一个 filter。一个 filter 在 doFilter() 方法中结束。如果一个 filter 想停止 request 处理而获得对 response 的完全的控制, 那它可以不调用下一个 filter。

一个 filter 可以包装 request 或 response 以改变几个方法和提供用户定制的属性。Api2.3 提供了 HttpServletRequestWrapper 和 HttpServletResponseWrapper 来实现, 它们能分派最初的 request 和 response。如果要改变一个方法的特性, 必须继承 wrapper 和重写方法。下面是一段简单的日志 filter 用来记录所有 request 的持续时间。

```
public class LogFilter implements Filter {
    FilterConfig config;

    public void setFilterConfig(FilterConfig config) {
        this.config = config;
    }

    public FilterConfig getFilterConfig() {
        return config;
    }
}
```

```
public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain) {
    ServletContext context = getFilterConfig().getServletContext();
    long bef = System.currentTimeMillis();
    chain.doFilter(req, res); // no chain parameter needed here
    long aft = System.currentTimeMillis();
    context.log("Request to " + req.getRequestURI() + ": " + (aft - bef));
}
}
```

当 server 调用 setFilterConfig(), filter 保存 config 信息。在 doFilter() 方法中通过 config 信息得到 servletContext。如果要运行这个 filter, 必须去配置到 web.xml 中。:

```
<filter>
    <filter-name>log</filter-name>
    <filter-class>
        LogFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>log</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

当每次请求一个 request 时(如 index.jsp), 先到 LogFilter 中去并调用 doFilter() 方法, 然后才到各自的 servlet 中去。

下面是一个用 filter 实现转换字符编码和控制用户访问权限的代码:

```
package com.logcd.filter;
// 转换字符编码
public class EncodingFilter implements Filter {

    private String encoding = null;

    public void destroy() {
        encoding = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        String encoding = getEncoding();
        if (encoding == null) {
            encoding = "gb2312";
        }
    }
}
```

```
request.setCharacterEncoding(encoding); // 在请求里设置上指定的编码
chain.doFilter(request, response);

}

public void init(FilterConfig filterConfig) throws ServletException {
    this.encoding = filterConfig.getInitParameter("encoding");
}

private String getEncoding() {
    return this.encoding;
}
}
```

```
package com.logcd.filter;
// 控制用户访问权限
public class SecurityFilter implements Filter {
    public void doFilter(ServletRequest request,
        ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        HttpSession session = req.getSession();
        if (session.getAttribute("username") != null) { // 登录后才能访问
            chain.doFilter(request, response);
        } else {
            res.sendRedirect("../failure.jsp");
        }
    }
}
```

配置文件:

```
<filter>
    <filter-name>EncodingFilter</filter-name>
    <filter-class>com.logcd.filter.EncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>gb2312</param-value>
    </init-param>
</filter>
```

```
<filter>
  <filter-name>SecurityFilter</filter-name>
  <filter-class>com.logcd.filter.SecurityFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>EncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>SecurityFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

#### 代码分析:

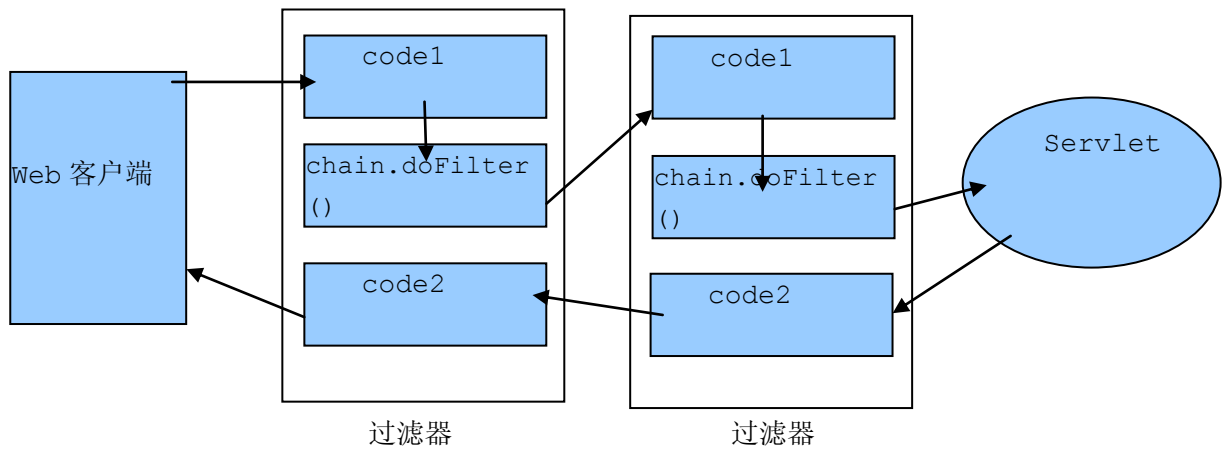
两个过滤器, EncodingFilter 负责设置编码, SecurityFilter 负责控制权限, 服务器会按照 web.xml 中过滤器定义的先后循序组装成一条链, 然后一次执行其中的 doFilter() 方法。执行的顺序就如下图所示, 执行第一个过滤器的 chain.doFilter() 之前的代码, 第二个过滤器的 chain.doFilter() 之前的代码, 请求的资源, 第二个过滤器的 chain.doFilter() 之后的代码, 第一个过滤器的 chain.doFilter() 之后的代码, 最后返回响应。

执行的代码顺序是:

1. 执行 EncodingFilter.doFilter() 中 chain.doFilter() 之前的部分:  
request.setCharacterEncoding("gb2312");
2. 执行 SecurityFilter.doFilter() 中 chain.doFilter() 之前的部分: 判断用户是否已登录。
3. 如果用户已登录, 则访问请求的资源: /admin/index.jsp。
4. 如果用户未登录, 则页面重定向到: /failure.jsp。
5. 执行 SecurityFilter.doFilter() 中 chain.doFilter() 之后的部分: 这里没有代码。
6. 执行 EncodingFilter.doFilter() 中 chain.doFilter() 之后的部分: 这里也没有代码。

过滤链的好处是, 执行过程中任何时候都可以打断, 只要不执行 chain.doFilter() 就不会

再执行后面的过滤器和请求的内容。而在实际使用时，就要特别注意过滤链的执行顺序问题，像 `EncodingFilter` 就一定要放在所有 `Filter` 之前，这样才能确保在使用请求中的数据前设置正确的编码。



# 15 jsp 标准标签库

## 15.1 标签库概述

- 标签库是一个 Web 组件，包括：
  - 一个标签库描述符文件（tld 文件）
  - 所有相关的标签处理器类（jar 文件）
- JSP 页面中使用的标签处理器可以访问在 JSP 页面中访问的对象
  - 如请求和会话范围的属性
- ❖ 标签遵循 XML 标签规则
- ❖ 需要在 JSP 页面和 Web 应用程序的部署描述文件（web.xml）中声明标签库
- ❖ 在 JSP 页面中可使用自定义的空标签
- ❖ 在 JSP 页面中使用标签，可有条件地执行 HTML 响应的某部分
- ❖ 在 JSP 页面中使用标签，可迭代执行 HTML

## 15.2 标签语法规则

- 标签使用 XML 语法：
  - ❖ 标准标签（包含标签体）：

```
<prefix: name { attribute={" value" | ' value' }}* >  
  
    body  
  
</prefix: name>
```

- ❖ 空标签：

```
<prefix: name { attribute={" value" | ' value' }}* />
```



- ❖ 标签名、属性及其前缀都是大小写敏感的
- ❖ 标签嵌套规则:

```
<tag1>
```

```
    <tag2>
```

```
    </tag2>
```

```
</tag1>
```

### 15.3 EL 语言

- EL 语言最初作为 JSP 标准标签库 (JSTL) 1.0 的一部分开发, JSP2.0 将 EL 语言正式嵌入 JSP 的标准语法。该语言的功能在于简化 JSP 的语法, 不需要使用 JavaScript 或者 java 表达式, 即可方便地从 JSP 页面访问或输出数据。

EL 不能用在只支持 JSP1.2 或更早的服务器中。

- EL 语言的语法:

- ❖ 所有 EL 语句都是“\${表达式}”, 通过使用“.”运算符和“[]”运算符来存取数据。
- ❖ \${couse.name} //表示 couse 变量的 name 属性值
- ❖ \${couse["name"]} //含义同上
- ❖ \${2 eq (4/2)} //即表达式 2==(4/2)
- ❖ \${param.un} //即表达式 request.getParameter("un")
- ❖ \${param.un= 'admin'} //即 request.getParameter("un")== "admin"
- ❖ \${sessionScope.un} //即 session.getAttribute("un")
- ❖ \${header["ua"]} //即 request.getHeader("ua")

- 使用注意

- ❖ 在 tomcat 下如果使用 jee5.0, 会导致 EL 语言的失效, 这个时候需要在 jsp 页面上方加上: `<%@page isELIgnored="false"%>`
- ❖ isELIgnored 表示是否禁用 el 表达式, 一定要设置成 false, 也就是不禁用

## 15.4 jsp 标准标签库 (JSTL)

➤ JSTL 包括 5 大类标准的已制定的标签库。

- ❖ 核心标签库
- ❖ XML 处理标签库
- ❖ I18N 格式标签库
- ❖ SQL 标签库
- ❖ 函数标签库

➤ JSTL 核心标签库

- ❖ 要使用 JSTL 中的核心标签库, 必须在页首包含下列指令:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

- 利用 JSTL 和 EL 语言开发 web 程序, 取代传统直接在页面上嵌入 java 程序的做法, 以提高程序的可读性、维护性和方便性。

### 1、显示/设定值

- ❖ `<c:out value="${param.submitFlag}"/>`

其中 value 属性是页面输出的表达式

- ❖ `<c:set value="value" var="varName" scope="request"/>`

将 value 值或标签体内容设定给一个 var 变量

## 2、流程控制-条件标签

❖ `<c:if test="${param.submitFlag=='add'}" var="sf" scope="session">`

.....

`</c:if>`

test 属性是判断的条件表达式;

var 属性为保存条件表达式结果的变量;

scope 属性是变量 var 的访问范围

## 3、循环标签

❖ `<c:forEach var="um" items="${sessionScope.listCol}" >`

.....

`</c:forEach>`

该标签用来迭代一个集合元素, 其中 items 属性代表被迭代的集合对象, var 属性代表每次迭代取出的成员变量, 还可以用 begin, end 和 step 属性来指定起点, 终点和步长。

### 15.4.1 使用标签库

在部署描述符中使用 taglib 元素声明 Web 应用程序使用一个标签库

`<taglib>`

`<taglib-uri>http://java.sun.com/jsp/jstl/core</taglib-uri>`

`<taglib-location>/WEB-INF/c.tld</taglib-location>`

`</taglib>`

## 使用 JSP 标签的步骤，以及每步详细的工作

使用自定义标记前，要拿到 .tld 和 .jar 两类文件

- 1、把 .jar 文件放到 WEB-INF 的 lib 里面。
- 2、把 c.tld 文件放到 WEB-INF 根目录下。
- 3、在 web.xml 中配置。
- 4、在页面中引用。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

uri 必须与 web.xml 中的 uri 对应

prefix 是前缀，用来区分是哪一个 taglib

使用标记

格式：<prefix:tag 名称 属性>

## 15.5 开发自定义标签

### 15.5.1 创建标签的处理类

☞ 扩展 TagSupport 类

☞ 为每个标记属性提供一个私有的实例变量；为所有在标记定义中非强制的属性提供一个显式的缺省值

☞ 为每个标记属性提供 setter、getter 方法进行访问

☞ 覆盖 doStartTag 方法：容器在遇到开始标签时会调用这个方法，该方法返回一个 int 值。

Tag.SKIP\_BODY：表示标签之间的内容被忽略，直接调用 doEndTag() 方法。

Tag.EVAL\_BODY\_INCLUDE：表示标签之间的内容被正常执行，标签主体内容并将结果包括在响应中。

☞ 覆盖 doAfterBody 方法：重复执行标签体内容的方法，该方法返回一个 int 值。

Tag.SKIP\_BODY：表示不用处理标签体，直接调用 doEndTag() 方法

Tag.EVAL\_BODY\_AGAIN：重复执行标签体内容。

☞ 覆盖 doEndTag 方法：容器在遇到结束标签时会调用这个方法，该方法返回一个 int 值。

Tag.SKIP\_PAGE：表示立即停止执行 JSP 网页，网页上未处理的静态内容和 JSP 程序均

被忽略，任何已有的输出内容立即返回到浏览器上。

Tag.EVAL\_PAGE:表示按正常的流程继续执行自定义标签以后的 JSP 网页。

🔗 覆盖 release 方法:容器通过这个方法释放本标签处理对象所占用的系统资源

标签处理类 **MyTag.java**:

```
package tag;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public class MyTag extends TagSupport{
    private String name;
    private String value;

    public void setName(String name) {
        this.name = name;
    }
    public void setValue(String value) {
        this.value = value;
    }

    public int doStartTag() throws JspException {

        //if(this.name!=null && this.name.equals("add")){
        String str=this.pageContext.getRequest().getParameter(name);
        System.out.println("str===="+str);
        if(str!=null && str.equals(value)){
            System.out.println("11111====");
            return this.EVAL_BODY_AGAIN;
        }else{
            System.out.println("22222====");
            return this.SKIP_BODY;
        }
    }
    public int doEndTag() throws JspException {
        return this.EVAL_PAGE;
    }
}
```

部署描述符 **web.xml**:

```
<taglib>
    <taglib-uri>http://javakc.com</taglib-uri>
    <taglib-location>/WEB-INF/myTag.tld</taglib-location>
</taglib>
```

**标记库描述文件 myTag.tld:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.2</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>javakc taglib</shortname>
  <uri>http://javakc.com</uri>
  <tag>
    <name>equals</name>
    <tagclass>tag.MyTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>name</name>
      <required>true</required>
      <!-- run time exception value-->
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>value</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

**JSP 页面 create.jsp:**

```
<!--在request范围内查找名字为submitFlag的参数，再与value的值进行比较-->
<!--如果相同就执行标签体内的跳转-->
<fk:equals name="submitFlag" value="add">
  <jsp:forward page="/user2"/>
</fk:equals>
```

## 16 web.xml 配置详解

```
<?xml version="1.0" encoding="utf-8"?>
<!-- 声明XML的版本, 编码格式 -->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
<!--指明schema的来源, 为http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd. -->

  <description>Web应用的描述</discription>

  <display-name>Web应用的名称</display-name>

  <icon>
  <!--icon 用来指定web站台中小图标和大图标的路径 -->

    <small-icon>/路径/smallicon.gif</small-icon>
    <!--small-icon元素应指向web站台中某个小图标的路径, 大小为 16 X 16 pixel -->
    <!--但是图象文件必须为GIF或JPEG格式, 扩展名必须为:gif或jpg -->

    <large-icon>/路径/largeicon-jpg</large-icon>
    <!--large-icon元素应指向web站台中某个大图表路径, 大小为 32 X 32
    pixel, 但是图象文件必须为GIF或JPEG的格式, 扩展名必须为:gif 或jpg-->
  </icon>

  <distributable></distributable>
  <!--distributable元素为空标签, 它的存在与否可以指定站台是否可分布式处理.
  如果web.xml中出现这个元素, 则代表此Web应用多个JSP Container之间分散执行-->

  <context-param>
  <!--context-param元素用来设定web应用的环境参数(context),
  它包含两个子元素:param-name和param-value-->
    <param-name>参数名称</param-name>
    <!-- 设定Context名称 -->
    <param-value>值</param-value>
    <!-- 设定Context名称的值-->
  </context-param>

  <!-- filter过滤器的相关设定-->
  <filter>
    <!-- Filter的名称 -->
    <filter-name>Filter的名称</filter-name>
    <!-- 定义Filter类的类名称 -->
    <filter-class>
```

```
        coreservlet.javaworld.CH11.SetCharacterEncodingFilter
    </filter-class>
    <!--Filter初始化参数-->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>GB2312</param-value>
    </init-param>
</filter>

<!-- 定义Filter所对应的URL -->
<filter-mapping>
    <filter-name>Filter的名称</filter-name>
    <!-- Filter所对应的URL -->
    <url-pattern>/Filter/Hello</url-pattern>
</filter-mapping>

<!--定义web应用的监听器 -->
<listener>
    <listener-class>
        coreservlet.javaworld.CH11.ContenxtListener
    </listener-class>
</listener>

<servlet>
    <servlet-name>Servlet的名称</servlet-name>
    <servlet-class>Servlet类路径</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Servlet的名称</servlet-name>
    <url-pattern>Servlet URL</url-pattern>
</servlet-mapping>

<!-- 定义web应用中的session参数-->
<session-cofig>
<!-- 定义这个web站台所有session的有效期限. 单位为分钟 -->
    <session-timeout>分钟</session-timeout>
</session-config>

<!-- 定义某一个扩展名和某一MIME Type做对映 -->
<mime-mapping>
    <extension>扩展名名称</extension>
    <mime-type>MIME格式</mime-type>
</mime-mapping>
<mime-mapping>
    <extension>doc</extension>
    <mime-type>application/vnd.ms-word</mime-type>
</mime-mapping>
<mime-mapping>
    <extension>xls</extension>
```



```
<mime-type>application/vnd.ms-excel</mime-type>
</mime-mapping>
<mime-mapping>
  <extension>ppt</extension>
  <mime-type>application/vnd.ms-powerpoint</mime-type>
</mime-mapping>

<welcome-file-list>
  <welcome-file>用来指定首页文件名称</welcome-file>
</welcome-file-list>

<!-- 将错误代码 (ErrorCode) 或异常 (Exception) 的种类对应 到web站台资源路径. -->
<error-page>
  <error-code>错误代码</error-code>
  <exception-type>
    Exception (完整名称的Java异常类型)
  </exception-type>
  <location>/路径</location>
</error-page>

<error-page>
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/except.jsp</location>
</error-page>

<!-- 设定JSP网页用到的Tag Library路径 -->
<taglib>
  <!--定义TLD文件的URI,JSP网页的taglib指令可以由这个URI存取到TLD文件-->
  <taglib-uri>URI</taglib-uri>
  <!--TLD文件对应的存放位置 -->
  <taglib-location>
    /WEB-INF/lib/xxx.tld
  </taglib-location>
</taglib>

<!-- 定义利用JNDI取得web应用可利用资源 -->
<resource-ref>
  <description>资源说明</description>
  <rec-ref-name>jdbc/sample_db (资源名称)</rec-ref-name>
  <res-type>javax.sql.DataSource (资源种类)</res-type>
  <!--资源由Application或Container来许可-->
  <res-auth>Application|Container</res-auth>
  <!-- 资源是否可以共享. 默认值为Shareable-->
  <res-sharing-scope>
```

```
        Shareable|Unshareable
    </res-sharing-scope>
</resource-ref>
</web-app>
```

# 17 MyEclipse

## 17.1 快捷键

### 17.1.1 CTRL

Ctrl+I 快速修复

Ctrl+D: 删除当前行

Ctrl+Q 定位到最后编辑的地方

Ctrl+L 定位在某行

Ctrl+O 快速显示 OutLine

Ctrl+T 快速显示当前类的继承结构

Ctrl+W 关闭当前 Editor

Ctrl+K 快速定位到下一个

Ctrl+E 快速显示当前 Editor 的下拉列表

Ctrl+J 正向增量查找(按下 Ctrl+J 后, 你所输入的每个字母编辑器都提供快速匹配定位到某个单词, 如果没有, 则在 status line 中显示没有找到了,)

Ctrl+Z 返回到修改前的状态

Ctrl+Y 与上面的操作相反

Ctrl+/ 注释当前行, 再按则取消注释

Ctrl+D 删除当前行。

Ctrl+Q 跳到最后一行的编辑处

Ctrl+M 切换窗口的大小

Ctrl+I 格式化激活的元素 Format Active Elements。

Ctrl+F6 切换到下一个 Editor

Ctrl+F7 切换到下一个 Perspective

Ctrl+F8 切换到下一个 View

### 17.1.2 CTRL+SHIFT

Ctrl+Shift+E 显示管理当前打开的所有的 View 的管理器 (可以选择关闭, 激活等操作)

Ctrl+Shift+/ 自动注释代码

Ctrl+Shift+\ 自动取消已经注释的代码

Ctrl+Shift+O 自动引导类包

Ctrl+Shift+J 反向增量查找 (和上条相同, 只不过是后往前查)

Ctrl+Shift+F4 关闭所有打开的 Editor

Ctrl+Shift+X 把当前选中的文本全部变为小写

Ctrl+Shift+Y 把当前选中的文本全部变为大写

Ctrl+Shift+F 格式化当前代码

Ctrl+Shift+M (先把光标放在需导入包的类名上) 作用是加 Import 语句

Ctrl+Shift+P 定位到匹配的匹配符 (譬如{ }) (从前面定位后面时, 光标要在匹配符里面, 后面到前面, 则反之)

Ctrl+Shift+F 格式化文件 Format Document。

Ctrl+Shift+O 作用是缺少的 Import 语句被加入, 多余的 Import 语句被删除。

Ctrl+Shift+S 保存所有未保存的文件。

Ctrl+Shift+/ 在代码窗口中是这种/\*~\*/注释, 在 JSP 文件窗口中是 <!--~-->。

Shift+Ctrl+Enter 在当前行插入空行 (原理同上条)

### 17.1.3 ALT

Alt+/ 代码助手完成一些代码的插入, 自动显示提示信息

Alt+↓ 当前行和下面一行交互位置 (特别实用, 可以省去先剪切, 再粘贴了)

Alt+↑ 当前行和上面一行交互位置(同上)

Alt+← 前一个编辑的页面

Alt+→ 下一个编辑的页面(当然是针对上面那条来说了)

Alt+Enter 显示当前选择资源(工程, or 文件 or 文件)的属性

#### 17.1.4 ALT+CTRL

Alt+CTRL+↓ 复制当前行到下一行(复制增加)

Alt+CTRL+↑ 复制当前行到上一行(复制增加)

#### 17.1.5 ALT+SHIFT

Alt+Shift+R 重命名

Alt+Shift+M 抽取方法

Alt+Shift+C 修改函数结构(比较实用,有N个函数调用了这个方法,修改一次搞定)

Alt+Shift+L 抽取本地变量

Alt+Shift+F 把 Class 中的 local 变量变为 field 变量

Alt+Shift+I 合并变量

Alt+Shift+V 移动函数和变量

Alt+Shift+Z 重构的后悔药(Undo) Shift+Enter 在当前行的下一行插入空行(这时鼠标可以在当前行的任一位置,不一定是最后)

Alt+Shift+O(或点击工具栏中的 Toggle Mark Occurrences 按钮) 当点击某个标记时可使本页面中其他地方的此标记黄色凸显,并且窗口的右边框会出现白色的方块,点击此方块会跳到此标记处。

下面的快捷键是重构里面常用的,本人就自己喜欢且常用的整理一下(注:一般重构的快捷键都是 Alt+Shift 开头的了)

### 17.1.6 快捷键 F

F2 当鼠标放在一个标记处出现 Tooltip 时候按 F2 则把鼠标移开时 Tooltip 还会显示即 Show Tooltip Description。

F3 跳到声明或定义的地方。

F5 单步调试进入函数内部。

F6 单步调试不进入函数内部，如果装了金山词霸 2006 则要把“取词开关”的快捷键改成其他的。

F7 由函数内部返回到调用处。

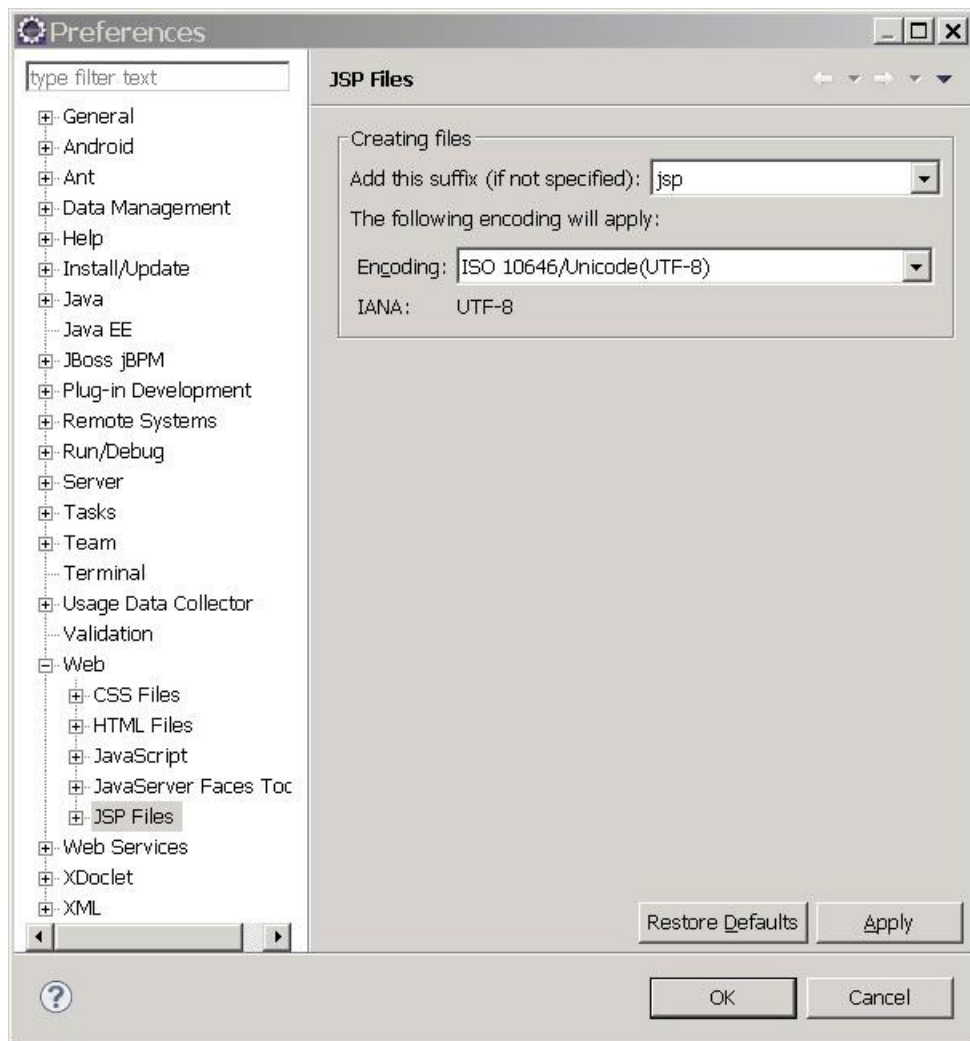
F8 一直执行到下一个断点。

## 17.2 设置

### 17.2.1 设置 jsp 模板的编码方式

由于 eclipse 中 jsp 模板的默认编码方式是 ISO-8859-1, 是不支持中文的, 所以在做 web 工程的时候, 每新建一个 jsp 页面, 都要修改 pageEncoding 为 UTF-8 或 GBK, 这样太麻烦了。

解决方法: Windows->Preference->Web->JSP Files, 解决方法如图所示:



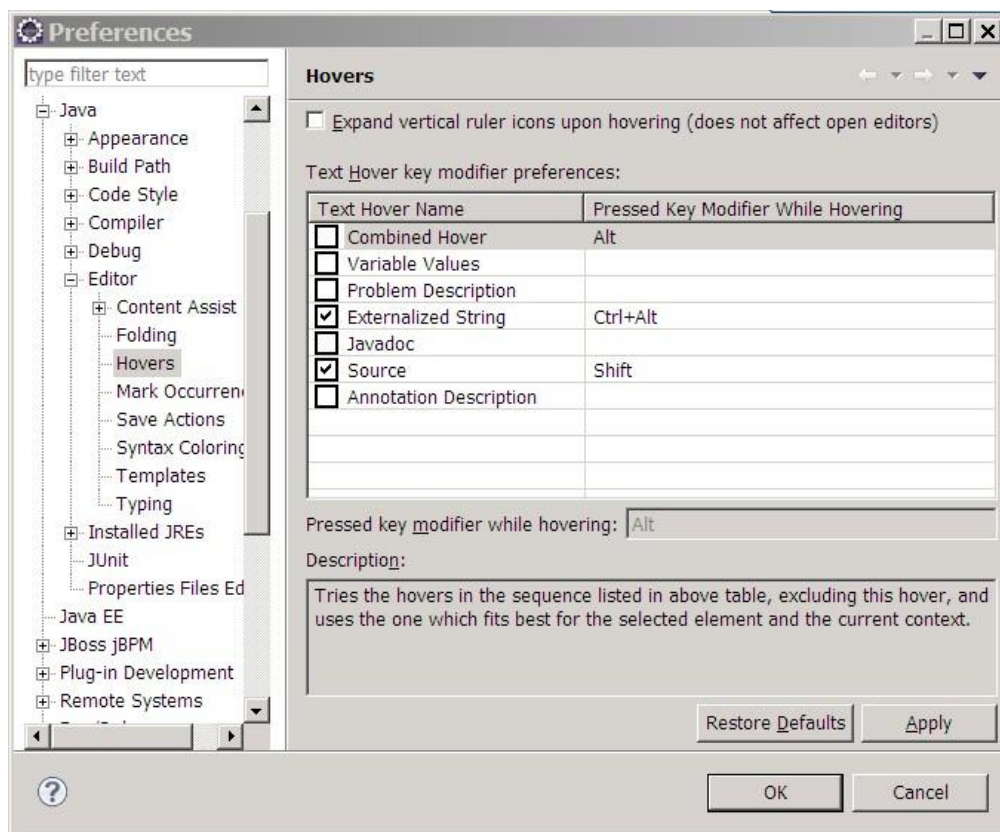
创建 Jsp 页面后, 代码如下:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
</body>
</html>
```

### 17.2.2 关闭鼠标悬停提示

Eclipse 的鼠标悬停提示功能（主要是变量类型声明和 Doc 帮助提示），突然弹出的窗口妨碍视线还影响思路，关闭方法如下：Window->Preferences->Java->Editor->Hovers 将 [Combined Hover]取消 Eclipse 中的快捷键。



取消 Combined Hover 的选项。

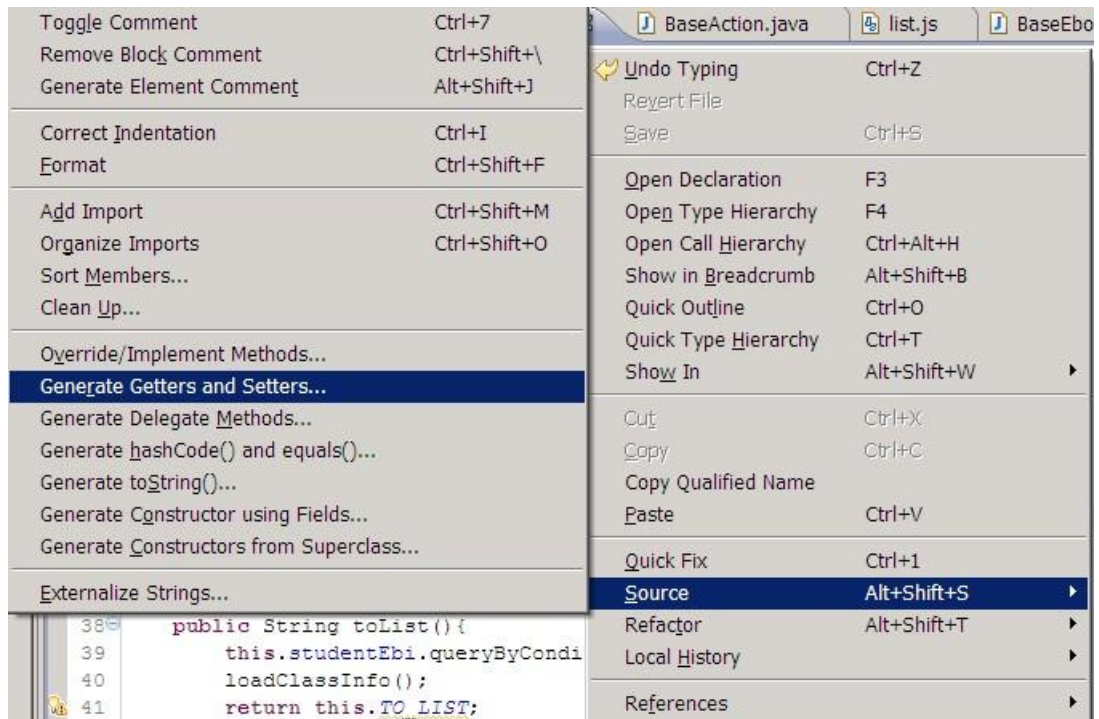
也可以为 Pressed key modifier while hovering 设置提示键 Alt，当按住 Alt 键时才会有悬停提示。

### 17.2.3 设置生成代码的位置

在自动生成 setter、getter 方法（或其他方法）时，通过 Insertion point 选项，设



置生成代码的位置。在代码中点击右键：



然后设置 Insertion point 选项：

