

重视大脑的学习指南

第一阶段  
全新的  
概念  
模拟测验

# Head First

# Servlets & JSP™

通过SCWCD考试之路

(中文版)

提供200余个实际  
的模拟测验题让  
你自测



Watch it!

避开1.5考试中的  
致命陷阱



了解Ted如何利用动态  
属性增加魅力



Bryan Basham, Kathy Sierra & Bert Bates 著

荆涛 林剑 等译

O'REILLY® 中国电力出版社



全面更新以涵盖针  
对J2EE 1.5的最新版  
SCWCD考试



使用c:out向世界  
传达你的消息

掌握定制  
标记库



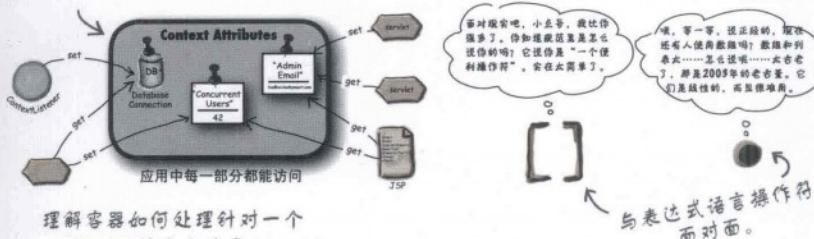
# Head First Servlets & JSP (中文版)

Java/Certification

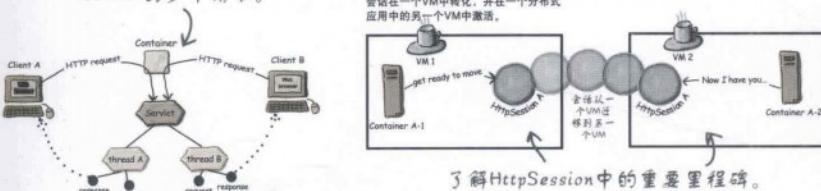
## 您将从本书学会什么?

你是不是要学最新版本J2EE 1.5参加Sun认证Web组件开发人员(SCWCD)考试?《Head First Servlets&JSP(第二版)》并没有给你一大堆需要死硬背的条条框框;它能将知识直接送入你的大脑。你会通过不寻常的方式Servlet和JSP打交道,可以学得更深入、更快捷。读完全书后,你会看到一个全新的模拟测验,这是模拟实际考试而专门设计的。

让你的大脑牢牢记住3个作用域。



理解容器如何处理针对一个Servlet的多个请求。



## 这本书为何与众不同?

我们认为,你的时间相当宝贵,不应当过多地花费在与新概念的纠缠之中。通过应用认知科学和学习理论的最新研究成果,本书可以让你投入一个需要多感官参与的学习体验,这本书采用丰富直观的形式使你的大脑真正开动起来,而不是长篇累牍地说教,让你昏昏欲睡。

“……事不宜迟, Head First 是你不二的选择。”

—Scott McNealy, Sun Microsystems公司主席、董事长兼CEO

“为了开设Servlet/JSP课程,我们买了不下10本这方面的书,但没有一本能真正满足我们的教学需求……终于最后发现了手中的这本书! Head First系列让我们教得更好。”

—Philippe Maquet,  
Loop Factory高级讲师,  
Brussels

ISBN 978-7-5083-8897-7



定价: 118.00元

O'REILLY®

[www.oreilly.com](http://www.oreilly.com)  
[www.headfirstlabs.com](http://www.headfirstlabs.com)

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

# Head First Servlets and JSP™

第二版



Bryan Basham,  
Kathy Sierra &  
Bert Bates 著  
荆涛 林剑 等译

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Taipei • Tokyo

## 图书在版编目 (CIP) 数据

Head First Servlets and JSP: 第2版: 中文版 / (美) 巴萨姆 (Basham,B.) , (美) 西拉 (Sierra,K.) , (美) 贝茨 (Bates, B.) 著; 荆涛等译.—北京: 中国电力出版社

书名原文: Head First Servlets and JSP, Second Edition

ISBN 978-7-5083-8897-7

I. H… II. ①巴…②西…③贝…④荆… III. JAVA语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字 (2009) 第087140

北京版权局著作权合同登记

图字: 01-2009-1959号

©2008 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2010. Authorized translation of the English edition, 2008 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc.出版2008。

简体中文版由中国电力出版社出版, 2010。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc.的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名/ Head First Servlets and JSP: 中文版

书 号/ ISBN 978-7-5083-8897-7

责任编辑/ 刘炽

封面设计/ Edie Freedman, Louise Barr, Steve Fehler, 张健

出版发行/ 中国电力出版社

地 址/ 北京市东城区北京站西街19号汇置通大厦 (邮政编码100005)

印 刷/ 航远印刷有限公司

开 本/ 880毫米×1230毫米 20开本 46印张 1265千字

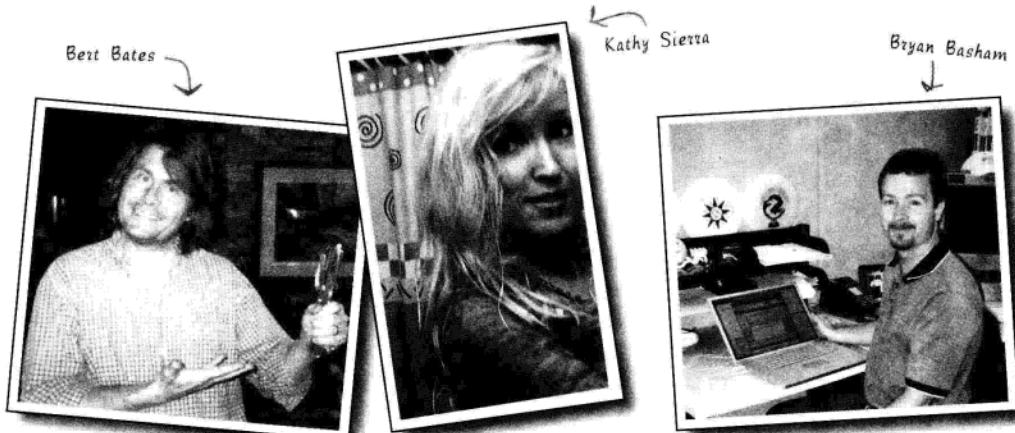
版 次/ 2010年8月第1版 2010年8月第1次印刷

印 数/ 0001~3000册

定 价/ 118.00元 (册)



# Head First系列（和这本书）的缔造者



**Bert**很早就是一位软件开发人员和架构师，不过由于在人工智能领域有近十年的经历，使他对学习理论和基于技术的培训发生了兴趣。在他软件生涯的最初十年，他在全世界游历，为诸如Radio New Zealand、Weather Channel和Arts & Entertainment Network (A & E)等众多客户提供帮助。他现在是Sun的Java证书考试开发小组中的一员，参与开发了许多证书考试，其中包括新的SCWCD考试。

Bert可以长时间地玩go游戏，无可救药地上了瘾，而且为go程序投入了很多精力。最后，还是Java语言的影响力让他终于罢手。他是一个不错的吉他手，现在正在努力学五弦琴。最近他买了一匹冰岛马，这也是他培训生涯中的一个新体验……

**Kathy**从开始设计游戏（她为 Virgin、MGM和Amblin等都编写过游戏）和开发AI应用以来，一直对学习理论很感兴趣。Head First系列的大多数格式都出自她之手，具体来说，都是她为UCLA Extension (加利福尼亚大学洛杉矶分校) 的“Entertainment Studies”研究项目讲授“New Media Interactivity”课程时完成的。最近，她成为Sun Microsystems公司的一名高级培训人员，负责教Sun的Java讲师如何讲授最新的Java技术，并参与开发了多个Sun的认证考试，其中包括SCWCD考试。与Bert Bates一道，她积极地使用Head First概念培训了成千上万的开发人员。她还是世界上最大的Java群体网站的创始人之一，即javaranch.com，这家网站赢得了2003年和2004年《软件开发》杂志生产力大奖。她的爱好包括跑步、滑雪、骑马、玩滑板，还有超自然科学。

曾经在NASA使用AI技术开发过高級自动化软件。他还曾任职于一家开发定制OO企业应用的顾问公司。目前，Bryan成为Sun课程开发小组的一员，主要关注Java和OO设计原则。他曾参与开发过Sun的许多Java课程的开发，包括JDBC、J2EE、Servlets和JSP，以及OO软（新媒体交互）课程时完成的。最近，件开发。他也是原来和最新版本SCWCD考试的首席设计者。

Bryan很热衷佛教，喜欢玩飞盘，他还是一个音乐发烧友，另外滑雪水平相当高超。

我们的E-mail地址：

[terrapin@wickedlysmart.com](mailto:terrapin@wickedlysmart.com)

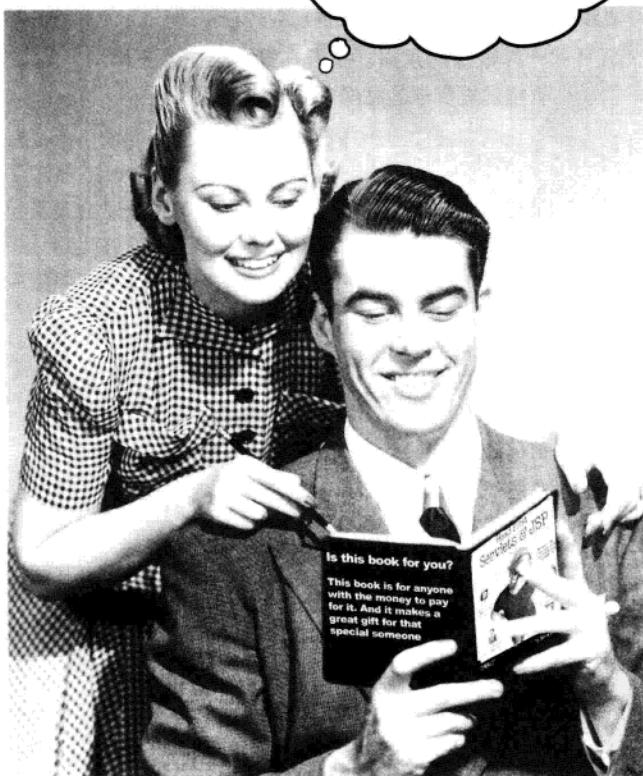
[kathy@wickedlysmart.com](mailto:kathy@wickedlysmart.com)

[bryan@wickedlysmart.com](mailto:bryan@wickedlysmart.com)

## 如何使用这本书

# 引子

真是无法相信，这样一些东西也能放在一本编程书里！



有一个问题真是听得我们耳朵都磨出茧了。这就是：“你们为什么要把这样一些东西放在一本编程书里呢？”这一节正是要回答这个问题。

## 谁能看这本书？

如果对下面的所有问题你都能肯定地回答“是”：

- ① 你知道如何用Java编程吗（不过不要求精通）？
- ② 你喜欢挑战困难吗？你是不是愿意在做中学，而不仅是纸上谈兵？你想学习并通晓servlets和JSP，把它牢牢记住吗？是不是还希望顺利通过面向Java EE 1.5的SCWCD考试？
- ③ 你是不是更喜欢一种轻松的氛围，就像在餐桌上交谈一样，而不愿意被动地听技术报告似的枯燥乏味的说教？

那么这本书正是你需要的。

## 谁暂时还不适合看这本书？

如果满足下面任何一种情况：

- ① 你是不是对Java一无所知？虽然不要求你是一位高级Java程序员，但起码有一些经验才行。如果确实一点都不了解Java，先买一本《Head First Java》看看吧，不错，就是现在，事不宜迟，看完了那本书以后，再回来看这本书吧。
- ② 你是不是一个一流的Java开发人员，正在找一本参考书？
- ③ 你本身已经是一个经验丰富的Java EE专家，需要了解一些超高级的服务器技术、特定于服务器的有关问题以及企业体系结构，另外希望得到大量复杂、健壮的实际应用代码，是这样吗？
- ④ 你是不是对新鲜事物都畏头缩尾？只喜欢简单的直条，不敢尝试把条纹和格子混在一起看看？你是不是觉得，如果把Java组件都拟人化了，这样的一本书肯定不是一本正儿八经的技术书？

那么，太遗憾了，这本书不适合你。



[来自市场的声音：只要买得起，就能看这本书]

# 我们知道你在想什么。

“这算一本正儿八经的编程书吗？”

“这些图用来做什么？”

“我真能这样学吗？”

# 我们也知道你的大脑在想什么。

你的大脑总是渴求一些新奇的东西。它一直在搜寻、审视、期待着不寻常的事情发生。大脑的构造就是如此，正是这一点才让我们不至于固步自封，能够与时俱进。

我们每天都会遇到许多按部就班的事情，这些事情很普通，对于这样一些例行的事情或者平常的东西，你的大脑又是怎么处理的呢？它的做法很简单，就是不让这些平常的东西妨碍大脑真正的工作，那么什么是大脑真正的工作呢？这就是记住那些确实重要的事情。它不会费心地去记乏味的东西；就好像大脑里有一个筛子，这个筛子会筛掉“显然不重要”的东西，如果遇到的事情枯燥乏味，这些东西就无法通过这个筛子。

那么你的大脑怎么知道到底哪些东西重要呢？打个比方，假如你某一天外出旅行，突然一只大老虎跳到你面前，此时此刻，你的大脑里会发生什么呢？

看到这只大老虎，你的神经元会“点火”，情绪爆发，释放出一些化学物质。

好了，这样你的大脑就会知道……

**这肯定很重要！可不能忘记了！**

不过，假如你正待在家里，或者坐在图书馆里。这里很安全，很温暖，肯定没有老虎。你正在刻苦学习，准备应付考试。也可能想学一些比较难的技术，你的老板认为掌握这种技术需要一周时间，最多不超过十天。

这就存在一个问题。你的大脑很想给你帮忙。它会努力地把这些显然不太重要的内容赶走，保证这些东西不去侵占本不算充足的脑力资源。这些资源最好还是用来记住确实重要的事情。比如大老虎，再比如火灾险情。如果你曾经只是身着短衣裤被大雪围困，这件事肯定不会忘却，你的大脑会记住绝不要让这种情况再发生第二次。

我们没有一种简单的办法来告诉大脑，“嘿，大脑，真是谢谢你了，不过不管这本书多没意思，也不管我对它是多么的无动于衷，但我确实希望你能帮助我把这些东西记下来。”

你的大脑想着：这真的很重要。



唉，又是800多页没意思的文字，枯燥又乏味。

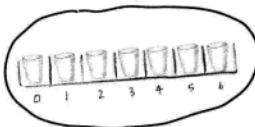
你的大脑认为，这些根本不值得去记。



## 我们认为，“Head First”的读者就是学习者。

那么，怎么来学习呢？首先，必须了解，然后要保证自己确实不会忘记。这可不是填鸭式的硬塞。根据认知科学、神经生物学和教育心理学的最新研究，学习的途径相当丰富，绝非只是通过书本上的文字。我们很清楚怎么让你的大脑兴奋起来。

### 下面是一些Head First学习原则：



**看得到。**与单纯的文字相比，图片更能让人记得住，通过图片，学习效率会更高（对于记忆和传递型的学习，甚至能有多达89%的效率提升），而且图片更容易看懂。以往总是把图片放在一页的最下面，甚至放在另外的一页上，与此不同，如果把文字放在与之相关的图片内部，或者在图片的周围写上相关文字，学习者的学习能力就能得到多至两倍的提高，从而能更好地解决有关的问题。

**采用一种针对个人的交谈式风格。**最新的研究表明，如果学习过程中采用一种第一人称的交谈方式直接向读者讲述有关内容，而不是用一种干巴巴的语调介绍，学生在学习之后的考试中成绩会提高40%。正确做法是讲故事，而不是做报告。要用通俗的语言。另外不要太严肃。

如果你面对着这样两个人，一个是你在聚会上结识的一位有意思的朋友，另一个人学究气十足，喋喋不休地对你说教，在这两个人中，你会更注意哪一个呢？

抽象方法真是没意思。  
这些方法没有方法体。

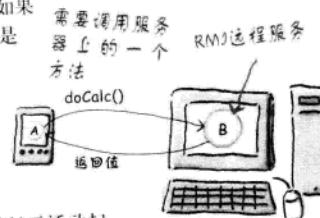


**引起读者的注意，而且要让他一直保持注意。**我们可能都有过这样的体验，“我真的想把这个学会，不过看过一页后实在是让我昏昏欲睡”。你的大脑注意的是那些不一般、有意思、有些奇怪、抢眼的、意料之外的东西。学习一项有难度的新技术并不一定枯燥。如果学习过程不乏味，你的大脑很快就能学会。

`abstract void roam();`

↑  
没有方法体！  
用一个分号结束。

**影响读者的情绪。**现在我们知道了，记亿能力很大程度上取决于所记的内容对我们的情绪有怎样的影响。如果是你关心的东西，就肯定记得住。如果你感受到了什么，这些东西就会留在你的脑海中。不过，我们所说的可不是让你感受到了什么，这些东西就会留在你的脑海中。不过，我们所说的可不是什么关于男孩与狗的伤心故事。这里所说的情绪是惊讶、好奇、觉得有趣，想知道“什么是……”，还有就是一种自豪感，如果你解决了一个难题，学会了所有人都觉得很难的东西，或者发现你了解的一些知识竟是那些自以为无所不能的傲慢家伙所不知道的，此时就会有一种自豪感油然而生。



## 元认知：有关思考的思考

如果你真的想学，而且想学得更快、更深，就应该注意你怎样才会专注起来，考虑自己是怎样思考的，并了解你的学习方法。

我们中间大多数人长这么大可能都没有上过有关元认知或学习理论的课程。我们想学习，但是很少有人教我们怎么来学习。

不过，这里可以做一个假设，如果你手上有这本书，你确实想学习如何用Java构建Web应用。另外因为你要参加考试，所以需要记住你读到的所有内容。为此必须理解这些内容。要想最大程度地掌握这本书或其他任何一本书中介绍的知识，就要让你的大脑负起责来，要求它记住这些内容。

怎么做到呢？技巧就在于要让你的大脑认为你学习的新东西确实很重要，对你的生活有很大影响。就像老虎出现在面前一样。如若不然，你将陷入旷日持久的拉锯战中，虽然你很想记住所学的新内容，但是你的大脑却会竭尽全力地把它们拒之门外。

那么究竟怎样才能让你的大脑把servlet看作是一只饥饿的老虎呢？

这有两条路，一条比较慢，很乏味。另一条路不仅更快，还更有效。慢方法就是大量地重复。你肯定知道，如果反反复复地看到同一个东西，即便再没有意思，你也能学会并记住。如果做了足够的重复，你的大脑就会说，“尽管看上去这对他来说好像不重要，不过，既然他这样一而再、再而三地看同一个东西，所以我觉得这应该是很重要的。”

更快的方法是尽一切可能让大脑活动起来，特别是开动大脑来完成不同类型的活动。如何做到这一点呢？上一页列出的学习原则正是一些主要的可取做法，而且经证实，它们确实有助于让你的大脑全力以赴。例如，研究表明，把文字放在所描述图片的中间（而不是放在这一页的别处，比如作为标题，或者放在正文中），这样会让你的大脑更多地考虑这些文字与图片之间有什么关系，而这就让更多神经元点火。让更多的神经元点火 = 你的大脑更有可能认为这些内容值得注意，而且很可能需要记下来。

交谈式风格也很有帮助，当人们意识到自己在与“别人”交谈，往往会更加关注，这是因为他们总想跟上谈话的思路，并能做出适当的发言。让人惊奇的是，大脑并不关心“交谈”的对方究竟是谁，即使你只是与一本书“交谈”，它也不会在乎！另一方面，如果写作风格很正式、干巴巴的，你的大脑就会觉得，这就像坐在一群人当中，被动地听人做报告一样，很没意思，所以不必在意对方说的是什么，甚至可以打瞌睡。

不过，图片和交谈风格还只是开始而已，能做的还有很多。

我想知道  
怎么才能骗过我  
的大脑，让它记住这  
些东西……



# 我们是这么做的：

我们用了很多图，因为你的大脑更能接受看得见的东西，而不是纯文字。对你的大脑来说，一幅图确实能胜过一千个字。如果既有文字又有图片，我们会把文字放在图片当中，因为文字处在所描述的图片中间时，大脑的工作效率更高，倘若把这些描述文字作为标题，或者“淹没”在别处的大段文字中，那么就达不到这种效果了。

我们采用了**重复手法**，会用不同方式，采用不同类型的媒体，运用多种思维手段来介绍同一个东西，目的是让有关内容更有可能储存在你的大脑中，而且能够在多个区中都有容身之地。

我们会用你意想不到的方式运用概念和图片，因为你的大脑喜欢新鲜玩意；在提供图和思想时，至少会含着一些**情绪因素**，因为如果能产生情绪反应，你的大脑就会投入更大的关注。而这会让你感觉到这些东西更有可能要被记住，其实这种感觉可能只是很幽默，让人奇怪或者比较感兴趣而已。

我们采用了一种针对个人的**交谈式风格**，因为当你的大脑认为你在参与一个交谈，而不是被动地听一场演示汇报时，它就会更加关注。即使你实际上在读一本书，也就是说在与书“交谈”，而不是真正与人交谈，但这对你的大脑来说并没有什么分别。

在这本书里，我们加入了40多个**实践活动**，因为与单纯的阅读相比，如果能实际做点什么，你的大脑会更乐于学习，更愿意去记。练习都是我们精心设计的，有一定的难度，但是确实能做出来，因为这是大多数人所希望的。

我们采用了**多种学习模式**，因为尽管你可能想循序渐进地学习，但是其他人可能希望先对整体有一个全面的认识，另外可能还有人只是想看一个示例。不过，不管你想怎么学，要是同样的内容能以多种方式来表述，这对每一个人都会有好处。

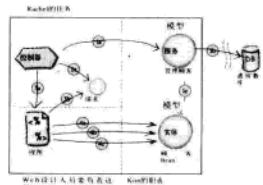
这里的内容不只是单单涉及左脑，也不只是让右脑有所动作，我们会让你的左右脑都开动起来，因为你的大脑参与得越多，你就越有可能学会并记住，而且能更长时间地保持注意力。如果只有一半大脑在工作，通常意味着另一半有机会休息，这样你就能更有效率地学习更长时间。

我们会讲故事、留练习，从多种不同的角度来看同一个问题，这是因为，如果要求大脑做一些评价和判断，它就能更深入地学习。

你会看到我们给出的一些练习，还要回答一些问题，这些问题往往不是直截了当就能做出回答，通过征服这些挑战，你就能学得更好，因为让大脑真正做点什么的话，它就更能学会并记住。想想吧，如果只是在健身馆里看着别人流汗，这对于保持你自己的体形肯定不会有帮助，正所谓临渊羡鱼，不如退而结网。不过另一方面，我们会竭尽所能不让你钻牛角尖，把劲用错了地方，而是能把功夫用在点子上。也就是说，你不会为搞定一个难懂的例子而耽搁，也不会花太多时间去弄明白一段晦涩难懂而且通篇行话的文字，我们的描述也不会太过简洁而让人无从下手。

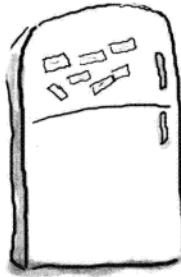
我们用了**拟人手法**。在故事中，在示例中，还有在图中，你都会看到人的出现，这是因为你本身是一个人，不错，这就是原因。如果和人打交道，相对于东西而言，你的大脑会投入更多的关注。

我们充分利用了80/20方法。我们认为，如果你真的要攻读JSP博士的话，这本书肯定不会是你唯一的JSP书。所以我们不打算面面俱到……这里只提供了你真正需要的东西。



要点





# 可以用下面的方法让你的大脑就范

好了，我们该做的已经做了，剩下的就要看你自己的了。这些提示只是个开头：听一听你的大脑是怎么说的，弄清楚对你来说哪些做法可行，哪些做法不能奏效。还可以做些新的尝试。

把这一页撕下来，贴到你的冰箱上。

## ① 慢一点。你理解的越多，需要记的就越少。

不要光是看看而已。停下来，好好想一想。书中提出问题的时候，你不要直接去翻答案。可以假想成真的有人在问你问题。你让大脑想得越深，就越有可能学会并记住。

## ② 做练习，自己记笔记。

我们给你留了练习，但是如果这些练习的解答也由我们一手包办，那和有人替你参加考试有什么分别？不要只是坐在那里看着练习发呆。**拿出笔来**，写一写，画一画。大量研究都证实，学习过程中如果能实际动动手，将改善你的学习效果。

## ③ 阅读“*There are No Dumb Questions*（这里没有傻问题）”部分。

顾名思义。这些问题可不是可有可无的旁注，它们绝对是核心内容的一部分！千万不要跳过去不看。

## ④ 上床睡觉之前不要再看别的书了，或者至少不要再看其他有难度的东西。

学习中有一部分是在你合上书之后完成的（特别是，要把学到的知识长久地记住，这往往无法在看书的过程中做到）。你的大脑也需要有自己的时间来再做一些处理。如果在这段处理时间内你又往大脑里灌输了新的知识，那么你刚才学的一些东西就会被丢掉。

## ⑤ 要喝水，而且要多喝点水。

如果能提供充足的液体，你的大脑才能有最佳的表现。如果缺水（可能在你感觉到口渴之前就已经缺水了），学习能力就会下降。

## ⑥ 说出来，大声地说出来。

说话可以刺激大脑的另一部分。如果你想看懂什么，或者想更牢固地记住它，就要大声地说出来。更好的办法是，大声地解释给别人听。这样你会学得更快，而且可能会有一些新的认识，而这是以前光看不说的时候无从发现的。

## ⑦ 听听你的大脑怎么说。

注意一下你的大脑是不是负荷太重了。如果发现自己开始浮光掠影地翻看，或者刚看的东西就忘记了，这说明你该休息一会儿了。达到某个临界点时，如果还是一味地向大脑里塞，这对加快学习速度根本没有帮助，甚至还可能影响正常的学习。

## ⑧ 要有点感觉！

你的大脑需要知道这是很重要的东西。要真正融入到书中的故事里。为书里的照片加上你自己的说明。你可能觉得一个笑话很蹩脚，不太让人满意，但这总比根本无动于衷要好。

## ⑨ 等完全看完这本书之后再做最后的模拟测验。

如果太早做这个模拟测验，你就会不清楚自己为真正的考试准备得怎么样。等你觉得准备得差不多了，再做测验。而且一定要在180分钟的时间内完成这个模拟测验，SCWCD考试的时间正好180分钟。

## 看这本书需要些什么：

要把你的大脑开动起来，另外需要一支笔，除此以外，你还需要Java、Tomcat 5和一台计算机。

其他的开发工具就不需要了，比如，不是非要有一个集成开发环境（Integrated Development Environment, IDE）。我们强烈建议，在学完这本书之前，只用一个基本编辑器就可以了，其他的工具先不要用。支持servlet/JSP的IDE会隐藏一些细节，你将无法切身体会这些细节，而这些内容又确实很重要（可能正是要考的），所以，最好完全手工地开发servlet/JSP代码，而不要借助某个工具的力量。等你真正理解了到底会发生什么情况之后，可以再转而采用某个工具，让它自动完成创建和部署servlet/JSP的一些步骤。如果你已经知道怎么使用Ant，那么读完第3章之后就可以使用Ant来帮助你完成部署。不过在你完全记住了Web的部署结构之前，还是建议你不要使用Ant。

### 得到Tomcat

- 你需要有Java SE v1.5或更高版本。
- 如果还没有Tomcat 5，可以通过以下方式得到：  
<http://tomcat.apache.org>  
在主页左侧的下载菜单中选择“Tomcat v5.5”。
- 把页面向下滚动到“Binary Distributions”，下载合适的版本。如果不知道需要哪一个版本，可以选择“Core”发行包，这正是你需要的。
- 把安装文件保存在一个临时目录。
- 安装Tomcat。
- 如果是Windows，要双击install.exe文件，然后按照安装向导的指令完成安装。
- 如果是其他系统，你想把Tomcat安装到硬盘上的哪个目录上，就把安装文件解压到那个位置。
- 要想更容易地运行书中给出的指令，请把Tomcat主目录命名为“tomcat”（或者为实际的Tomcat主目录建一个“tomcat”别名）。
- 根据不同的系统，适当地设置环境变量JAVA\_HOME和TOMCAT\_HOME。
- 你手上应该有一份规范，不过不是非得有一个规范才能通过考试。写这本书的时候，可以从以下地址获得规范：  
Servlet 2.4 (JSR #154)      <http://jcp.org/en/jsr/detail?id=154>  
JSP 2.0 (JSR #152)      <http://jcp.org/en/jsr/detail?id=152>  
JSTL 1.1 (JSR #52)      <http://jcp.org/en/jsr/detail?id=52>
- 访问JSR，点击最新版本的下载链接（Download Page）。
- 测试Tomcat，为此对于Linux/Unix/OS X，启动tomcat/bin/startup脚本（startup.sh）。在浏览器上访问<http://localhost:8080/>，就会看到Tomcat欢迎页面。



Java 2标准版1.5

Tomcat 5

考试涵盖以下规范：

- Servlets 2.4
- JSP 2.0
- JSTL 1.1

## 需要了解的最后几点：

要把这看做是一个学习过程，而不要简单地把它看成是一本参考书。我们在安排内容的时候有意做了一些删减，只要是对有关内容的学习有妨碍，我们会毫不留情地把这些部分一律删掉。另外，第一次看这本书时，要从头看起，因为书中后面的部分会假定你已经看过而且学会了前面的内容。

我们对UML有所修改。  
使用了一种更简单的“UML”。  
↓

我们用了“类”UML图（注意，可不是UML类图，而是指与UML图很相似）。

你很可能早就会UML了，不过这个考试对UML没有要求，所以在学这本书之前不必非要先学UML。你不用担心在学servlet、JSP、JSTL的同时还要学UML。

我们没有把规范中的细枝末节一网打尽。

SCWCD考试是很重细节的，当然我们也一样。但是，如果规范中的某个细节不在考试范围内，我们就不会谈到这个内容，除非它对大多数组件开发人员都很重要。开始开发Web组件(servlet和JSP)需要知道些什么？想通过考试又需要知道些什么？其实这两者85%的内容都是重叠的。我们也谈到了一些考试不要求的内容，不过会特别指出这是考试不要求的，如果只是要通过考试，你就不必去记这些内容。实际的SCWCD考试就是我们出题，所以我们知道你应该把劲往哪里使！如果考试中的某个问题问到了一个过于“吹毛求疵”的细节，但是我们认为学习这一点有些得不偿失，事倍功半，我们就会忽略这个细节，或者轻描淡写地提一下，也可能只是在一一道模拟测验题中问到。

书里的实践活动不是可有可无的。

这里的练习和实践活动可不是可有可无的装饰和摆设，它们也是这本书核心内容的一部分。其中有些练习和活动有助于记忆，有些能够帮助你理解，还有一些对于如何应用你所学的知识很有帮助。千万不要忽略任何实践活动。

我们有意安排了许多重复，这些重复非常重要。

Head First系列的书有一个与众不同的地方，这就是，我们希望你确确实实地学会，另外，希望在学完这本书之后你能记住学过了什么。尽管重复很有必要，不过大多数参考书不一定认为重复和回顾是一个重要的环节，但是在这本书里，你会看到一些概念会一而再、再而三地出现很多次。

代码例子尽可能短小精悍。

有读者告诉我们，如果查了200行代码才能找到要理解的那两行代码，这是很让人郁闷的。这本书里大多数例子往往都开门见山，作为上下文的代码会尽可能的少，这样你就能一目了然地看到哪些东西是需要你学习的。别指望这些代码很健壮，要知道这里的代码甚至是不完整的。要开发健壮而且完整的代码，这正是你学完这本书以后要做的工作。书里的示例特意写得很简单，以便于你学习，这些例子的功能往往不太完备。本书的部分代码示例可以在[www.headfirstlabs.com](http://www.headfirstlabs.com)得到。

<b>Director</b>
getMovies
getOscars()
getKevinBaconDegrees()

## 关于SCWCD考试（针对Java EE 1.5）

更新后的SCWCD考试现在叫作“面向Java平台企业版5的Sun认证Web组件开发人员考试（Sun Certified Web Component Developer for the Java Platform, Enterprise Edition 5）”（CX-310-083），不过不要被这个名字吓到。更新后的考试还是针对Java EE v1.4、servlet v2.4和JSP v2.0规范设计的。

### 我得先通过SCJP吗？

是的。不管是Web组件开发人员考试（Web Component Developer exam）、业务组件开发人员考试（Business Component Developer exam）、移动应用开发人员考试（Mobile Application Developer exam），还是Web服务开发人员考试（Web Services Developer exam），实际上所有开发人员考试都要求你首先是一个Sun认证Java程序员，也就是必须先通过SCJP（Sun认证Java程序员考试，Sun Certified Java Programmer）。

### 考试中有多少个问题？

参加考试时，你要回答69个问题。每个人回答的问题可能不一样，考试提供了很多套题目。不过，每个人拿到的题目难度都一样，而且每套题都会全面地覆盖所有内容，没有特别的侧重。在实际的考试中，针对大纲中的每个考试要求都至少会有一道题，有些考点可能还涉及不只一个问题。

### 我要在多长时间内完成考试？

考试时间是三个小时（180分钟）。对大多数人来说，这都不成问题，因为这些问题都不会很长、很复杂，不会让你太为难。大多数问题都是很简短的多项选择，所以一下子就能确定是不是知道答案。

### 有哪些题型？

与我们最后提供的模拟测验题几乎一样，只有一个比较大的差别，因为题目是多项选择，实际的考试会告诉你每道题有几个正确答案，但是模拟试题中没有明确指出这一点。另外，你在实际考试中可能会遇到这样一些题目，要求把什么东西拖放到某个位置，这种问题在模拟测验里没有出现。不过，这种拖放问题实际上就是用交互的方式来回答匹配问题，也就是一个东西和另外哪个东西匹配。

### 要正确回答多少个问题才算通过？

必须答对49个问题（70%）才算通过考试。回答完所有问题之后，不要急匆匆地就按下done按钮，可以让鼠标在这个按钮上停一会，等有足够的勇气后再点击。因为一旦按下，几乎立刻（大约6纳秒）就能知道你是否通过（当然，我们相信你一定能通过）。

### 为什么这本书的模拟试题不指出有多少个正确选择？

我们希望模拟试题比实际的考试要稍微难一点，这样你就能对自己是否已经做好准备有一个最真实的认识。一般来说，人们在做书上的模拟试题时得分总是高一些，因为这套题目可能做了不只一次，这样一来，你可能对自己的准备程度有过高的估计，我们可不希望你在估计不足的情况下草率参加考试。读者后来都反馈说他们最后的考试得分和做这套模拟题的得分相当接近。

## 考试之后我能拿到什么？

离开考试中心之前，记得要拿到你的测验报告。你在每一部分得分多少，会在这个测验报告中有一个总结，另外报告中还会指出你是否通过了考试。要留好这份报告！这是你通过认证的第一份证明。考试完再过几周，你会收到Sun教育服务机构（Sun Educational Services）寄出的一个小包裹，这里面有你真正的纸质证书，Sun给你的一封祝贺信，还有一个可爱的胸针，上写Sun Certified Web Component Developer（Sun认证Web组件开发人员）几个“小字”，这几个字实在是太小了，所以就算是你凭这个胸针谎称自己得到了其他认证，可能也能蒙混过关，没人能看出有什么分别。如果你想喝点酒庆祝自己通过了考试，别指望Sun教育服务机构会随包裹给你寄瓶酒。

## 考试费用是多少，怎么注册呢？

考试的费用不便宜，需要\$200。所以你需要这本书……，确保一次就能通过考试。你要通过Sun教育服务机构注册，向他们提供你的信用卡号。相应地，你会得到一个准考证号，用这个准考证号就可以与离你最近的考试中心（Prometric Testing Center）约一个考试时间了。

要想从网上了解有关的详细情况并购买一个准考证，可以先访问：<http://www.sun.com/training/certification/>。如果你在美国，访问这个网页就够了。如果你不在美国，可以从右边的菜单条选择你所在的国家。

## 考试软件是什么样子？

这个软件用起来极其简单，就是向你问一个问题，再由你回答。如果你还不想回答，可以先跳过去，等以后再答。如果确实回答了，但是没有把握，希望在有时间的情况下再检查一下，可以对这个问题“做个标记”。做完之后，你会看到一个界面，其中显示出哪些问题还没有回答，另外哪些问题加了标记，这样就能直接返回去再完成或检查这些题目。

在考试刚开始的时候，会给你提供一个简单的教程，告诉你怎么使用这个软件，还会让你做一个小测验（与servlet无关）练练手。学习这个教程用的时间不算在SCWCD考试时间内。等你学完这个考试软件教程，而且确实准备好了，时钟才开始计时。

## 到哪能找到一个有关这个考试的学习小组，要准备多久才能参加这个考试？

要说关于这个考试的在线讨论组，最好的莫过于本书作者维护的论坛了（当然了，这有什么奇怪的）。你可以访问[www.javaranch.com](http://www.javaranch.com)，然后浏览Big Moose Saloon（所有论坛都在这里）。这个讨论组很有用，可别错过了。在那里肯定会有回答你的问题，没准回答问题的正是我们自己。JavaRanch是互联网上最友好的Java群体，所以不管你的Java水平如何，都不用害怕，这里欢迎每一个人。如果你还需要通过SCJP考试，同样可以在这里获得帮助。

要准备多久才能参加这个考试，这要取决于你已经有多少servlet和JSP经验。如果你以前从未接触过servlet和JSP，可能需要6~12周的时间，具体要多久，这要看你每天在这上面花多少时间。如果你已经有丰富的servlet和JSP经验，往往只需要3个星期的时间就能做好准备。

# Beta版测试人员&技术审校



Dave Wood



Joe Konior

Bear Bibault



没有附照片（但同样令人敬畏）： Amit Londhe



Philippe Maquet



Johannes deJong

Andrew Monkhouse



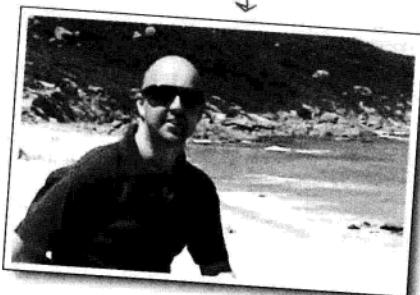
由于本书多了  
两根白头发



Jeb Cumps



Jason Menard



Sergio Ramirez



Dirk Schreckmann



Theodore Casser



Ulf Dittmer



Oliver Roell

Tollins Tchoumba →



Preetish Madalia



感谢

## 其他要批评的人：

致O'Reilly：

最要感谢的要算O'Reilly的Mike Loukides，所有策划都归功于他，是他的大力帮助才使Head First概念形成了一个系列。而且我们很庆幸，这位编辑确实对Java相当精通。另外，非常感谢Tim O'Reilly，是他在幕后推动Head First走到今天。真的很幸运，他总是能高瞻远瞩，而且极富开创意识。非常感谢你，聪明的Kyle Hart，你是当之无愧的“Head First系列之母”，是你发现了Head First正满足计算机图书领域所需，并促使了这个系列的诞生。

致勇敢无畏的技术审校：

不错，这本书比我们预想的要厚一些。不过，如果没有JavaRanch审查主管Johannes deJong，肯定会推迟很多。Johannes，你是我们的英雄。还要特别感谢Joe Konior，你对每一章的反馈篇幅都刚好合适。而且非常非常感谢Philippe Macquet，你的一丝不苟，你的高超技术，还有乐观的态度，让我们三个作者都迷恋不已，恨不能和你结婚……不过这当然有些怪异。我们特别感谢Andrew Monkhouse提供的技术反馈，而且帮助我们调整澳大利亚式的英语。Jef Cumps，你唱的“setHeader”真是不怎么样（不过倒是很有感情），但你的技术评论确实很有帮助。

Dave Wood总是喜欢打击我们，最喜欢指着前几页说，“看看，这不太Head First嘛”。另外，JavaRanch审查官Jason Menard、Dirk Schreckmann（前面的那个鱼脸）、Rob Ross、Ernest Friedman-Hill和Thomas Paul都提供了绝妙的反馈。一如平常，要特别感谢javaranch Trail的老板，Paul Wheaton。

还要特别感谢第二版的以下技术审校：Bear Bibeault、Theodore Casser、Ulf Dittmer、Preetish Madalia、Sergio Ramirez、Oliver Roell、Neeraj Singhal和Collins Tchoumba。

## 模拟测验

如果你发现一道JSP模拟题实在太难，害你苦思冥想也不得要领，不要怪我们，要怪就怪Marc Peabody！感谢Marc对我们的帮助，让参加SCWCD考试的人能做好充分的准备。Marc把大量空闲时间都用来审查JavaRanch，在这里他因鼓动程序员用普通的Java EE技术构建出人意料的mashup应用而著称。



Marc Peabody

## 还有一些要感谢的人 (注1)

### Bryan Basham的致谢

可以先来感谢我的妈妈，不过，这种写法有点老套……我对Java Web开发的知识是从以前开发的几个中型应用积累起来的，但是，要不是多年来与Sun的Java讲师通过电子邮件的争论，我的知识就不会取得现在的进步。特别地，我要感谢Steve Stelting、Victor Peters、Lisa Morris、Jean Tordella、Michael Judd、Evan Troyka以及Keith Ratliff。在我的学习道路上，帮助我成长的人很多，但是这6个人才使我成为今天的我，他们的影响给我留下了深深的烙印。

像所有写书过程一样，最后三个月最为难熬。我要感谢我的未婚妻Kathy Collina一直对我很耐心。还要感谢Karma和Kiwi（我们的猫），最后几晚一直趴在我旁边，偶尔还按几下键盘。

最后，也是最重要的，必须感谢Kathy和Bert，居然建议我一同参加这个写书的项目。Kathy Sierra真是无与伦比。她对元认知和教育设计的卓越见识，再加上她的独创性，完美地缔造了这一系列的“Head First”书。我在教育行业已经工作了5年，几乎从Kathy那里学到了所有一切……哦，别担心我妈妈，我在下一本“Head First”书里一定会用一大段来感谢她。妈妈，我爱你！

### Kathy和Bert的致谢

Bryan真会奉承人（不过Kathy倒也乐于听到这些奉承）。关于你的未婚妻，我们深表同意。但是她好像不太想念你呀，在我们汗流浃背、日以继夜地为这本书工作时，她却一整个夏天都在Ultimate开心地玩。不过，这个过程对Bryan来说肯定是一个难得的体验，你也是我们最好的合作者（注2）！你一直都那么平和，那么快乐，实在让我们佩服。

我们都非常感激Sun认证考试小组的努力工作，特别是Java证书主管Evelyn Cartagena，另外要感谢所有帮助开发Servlet和JSP规范相关JSR的人。

注1：之所以要感谢这么多人，这是因为我们发现了这样一条定律，书中致谢里提到的每个人都至少会买一本书，可能还会买好几本书，给亲戚和周围的所有人都送上一本。如果你希望我们在下一本的致谢里提到你，而且你们家族的人很多的话，可以写信给我们。

注2：澄清一下：Bryan是我们至今的唯一合作者，不过确实堪称一流。

# 目录概览

引子	xix
1 为什么使用Servlets & JSP：前言与概述	1
2 Web应用体系结构：高层概述	37
3 MVC迷你教程：MVC实战	67
4 作为Servlet：请求和响应	93
5 作为Web应用：属性和监听者	147
6 会话状态：会话管理	223
7 作为JSP：使用JSP	281
8 没有脚本的页面：无脚本的JSP	343
9 强大的定制标记：使用JSTL	439
10 JSTL也有力不能及的时候：定制标记开发	499
11 部署Web应用：Web应用部署	601
12 要保密，要安全：Web应用安全	649
13 过滤器的威力：过滤器和包装器	701
14 企业设计模式：模式和Struts	737
A 附录A：最终模拟测验	791
i 索引	865

# 详细目录



## 引子

让你的大脑来学Servlets。你想学些东西，但是你的大脑却在帮倒忙，不让你记住这些东西。你的大脑在想，“还是把空间留给更重要的事情吧，比方说要躲开哪些野兽，还有光着身子滑雪不太好吧。”那么你该如何骗过大脑，怎么让它认为要是不懂Servlets你就活不下去？

谁能看这本书	xx
我们知道你在想什么	xxi
元认知：有关思考的思考	xxiii
让你的大脑就范	xxv
看这本书需要些什么	xxvi
通过证书考试	xxviii
技术审校	xxx
致谢	xxxi

## 1

# 为什么使用Servlets & JSP

Web应用炙手可热。你知道有几个GUI应用能由全世界数百万的用户使用？作为一个Web应用开发人员，不用像对付独立应用那样，亲自处理棘手的部署问题，完全可以通过浏览器把你的应用交付给任何人。不过，你还需要Servlet和JSP助你一臂之力。因为原先普通的静态HTML页面太……，怎么说呢，只能算是1999年的老古董了。你会了解如何把Web网站变成真正的Web应用。



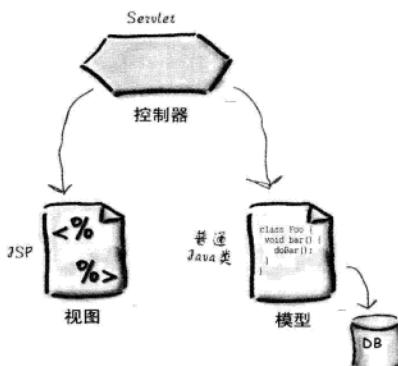
## 考试大纲

Web服务器和客户做什么？它们如何会话？	4
HTML速成指南	7
什么是HTTP协议？	10
HTTP POST请求剖析	16
使用URL定位Web页面	20
Web服务器擅长提供静态Web页面	24
Servlets揭秘：编写、部署和运行	30
HTML中引入Java，这就是JSP	34

## 2

# Web应用体系结构

Servlet也需要帮助。请求到来时，必须有人实例化servlet，或者至少要建一个新的线程处理这个请求。必须有人调用servlet的doPost()或doGet()方法。另外，必须有人把请求和响应交给servlet。还得有人管理servlet的生与死以及servlet的资源。在这一章中，我们会介绍容器，还会首次接触MVC模式。



## 考试大纲

什么是容器？	39
代码里有什么 (servlet何以成为一个servlet)	44
一个servlet可以有3个名字	46
故事：Bob构建了一个速配网站	50
模型 - 视图 - 控制器 (MVC) 概述和例子	54
一个“实际”部署描述文件 (DD)	64
J2EE如何集成这一切	65

# 3

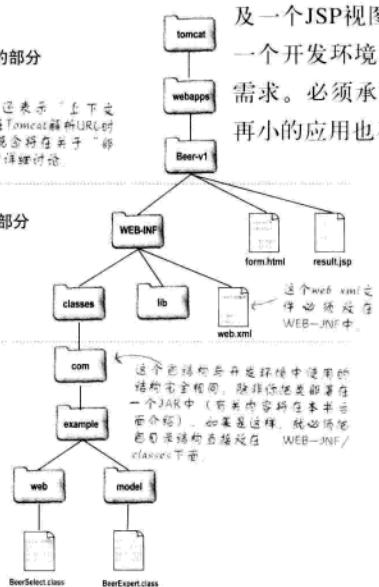
## MVC迷你教程

**创建和部署MVC Web应用。**该你动手写点东西了，要写一个HTML表单、一个servlet控制器、一个模型（普通的Java类）、一个XML部署描述文件，以及一个JSP视图。你要构建、部署和测试这个应用。不过第一步首先需要建立一个开发环境。需要建立遵循servlet和JSP规范的部署环境，并满足Tomcat的需求。必须承认，我们构建的只是一个小应用。不过……只要使用了MVC，再小的应用也不算小。

特定于Tomcat的部分

这个目录名还表示“上下文根”，这会在Tomcat解析URL时用到。这个概念将在关于“部署”的一章中详细讨论。

Servlet规范的部分



应用

BeerSelect.java  
BeerExpert.class

### 考试大纲

构建一个真正的（小）Web应用	69
创建你的开发环境	72
第一个表单页面的HTML	75
创建部署描述文件（DD）	76
创建、编译、部署和测试控制器servlet	81
设计、构建和测试模型组件	82
改进控制器，让它调用模型	83
创建和部署视图组件（JSP）	87
改进控制器servlet，让它调用JSP	88

# 4

## 作为servlet

Servlet的存在就是要为客户提供服务。servlet的任务就是得到一个客户的请求，再发回一个响应。请求可能很简单，“请给我一个欢迎页面。”也可能很复杂，“为我的购物车结账。”这个请求携带着一些重要的数据，你的servlet代码必须知道怎么找到和使用这个请求。servlet代码还要知道怎样发送响应，或者不发送……

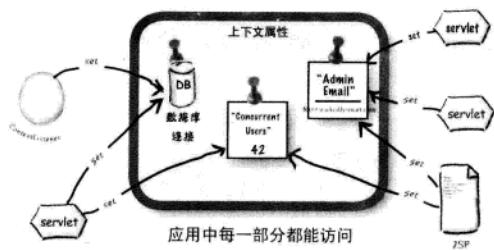


### 考试大纲

servlet受容器的控制	95
servlet初始化和线程	100
servlet真正的任务就是处理GET和POST请求	105
非幕等请求的故事	112
如何决定得到一个GET请求还是POST请求？	117
发送和使用参数	119
已经了解了请求……下面来看响应	126
可以设置响应头部，也可以增加响应头部	133
servlet重定向和请求分派器	136
复习：HttpServletResponse	140

# 5 作为Web应用

**没有servlet能独立存在。**在当前的现代Web应用中，许多组件都是在一起协作共同完成一个目标。你会有模型、控制器和视图；会用到参数和属性；还有一些辅助类。但是怎么把这些部分组织在一起呢？怎么让这些组件共享信息？如何隐藏信息？怎么让信息做到线程安全？能不能得出答案可能会决定你能不能保住工作。



## 考试大纲

148

解决之道：初始化参数和ServletConfig

150

JSP怎么得到Servlet初始化参数？

155

解决之道：上下文初始化参数

157

比较ServletConfig和ServletContext

159

她想要一个ServletContextListener

166

教程：编写一个简单的ServletContextListener

168

编译、部署和测试监听者

176

完整的故事：ServletContextListener复习

178

8个监听者：不只是针对上下文事件

180

到底什么是属性？

185

属性API和属性不好的一面

189

上下文作用域不是线程安全的！

192

问题的慢动作回放……

193

试试同步

195

会话属性是线程安全的吗？

198

SingleThreadModel

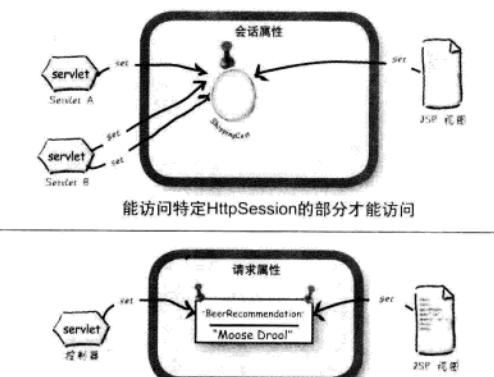
201

只有请求属性和局部变量是线程安全的

204

请求属性和请求分派

205



## 属性

196

能访问特定ServletRequest的部分才能访问

206

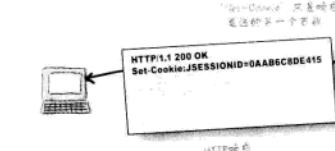


## 控制器

207

# 6 会话状态

Web服务器没有短期记忆。一旦发送了响应，Web服务器就会忘了你是谁。下一次你再做请求的时候，Web服务器不会认识你。它们不记得你曾经做过请求，也不记得它们曾经给你发出过响应。什么都记不得了。但有些时候可能需要跨多个请求保留与客户的会话状态。对于购物车，如果要求客户必须在一个请求中既做出选择又要结账，这是不合适的。



达基尔的cookie。  
重置会话ID……



HTTP响应

HTTP请求

## 考试大纲

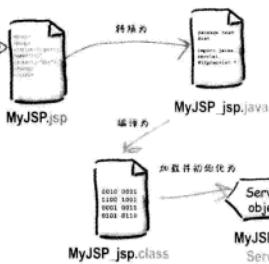
就像真正的会话（会话如何工作）	226
会话ID、cookie和会话的其他基础知识	231
URL重写：一条后路	237
如果会话过时，删除无效的会话	241
除了用于会话，还能用cookie做什么？	250
HttpSession的重要里程碑	254
不要忘了 HttpSessionBindingListener	256
会话迁移	257
Listener示例	261

# 7 作为JSP

JSP会变成servlet。这个servlet不用你来创建。容器会查看你的JSP，把它转换成Java源代码，再编译成完整的Java servlet类。但是，你必须知道，JSP中的代码转换成Java代码时到底发生了什么。你可以在JSP中写Java代码，但是这样做对吗？如果不写Java代码，那写什么呢？JSP代码怎么转换成Java代码？在这一章中，我们将看到6种不同类型的JSP元素，每种元素都有自己的用途，当然了，也有各自不同的语法。你将了解如何编写JSP，为什么编写JSP，还有在JSP中写些什么。而且你将了解在JSP中哪些不该写。



编辑



## 考试大纲

使用“out”和page指令创建一个简单的JSP	283
JSP表达式、变量和声明	288
来看看由JSP生成的servlet	296
out变量并非唯一的隐式对象……	298
JSP的生命周期和初始化	306
既然说到这里了……来看看3个指令	314
Scriptlet有害吗？答案尽在EL	317
先等等……还有动作没有讲到	323

# 8 没有脚本的页面

**扔掉脚本。**Web页面设计人员真的必须懂Java吗？服务器端Java程序员还得是图形设计人员吗？如果是你，你真的希望JSP里有大堆的Java代码吗？你会不会说这是“维护噩梦”？编写无脚本的页面不只是可能而已，有了新的JSP 2.0规范，编写无脚本的页面变得更容易，更灵活，这多亏了新的表达式语言（Expression Language，EL）。因为有JavaScript和XPATH的基础，Web设计人员能轻松自如地使用EL，一旦习惯了你也会喜欢它的。不过它也存在一些陷阱……尽管EL看上去很像Java，但它并不是Java。有时即使你使用了Java中同样的语法，EL却可能有完全不同的表现，所以一定要注意！



① Header文件 ("Header.jsp")

```
<%@ page language="java" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="javax.servlet.jsp.*" %>
```

② Contact.jsp

```
<%@ page language="java" %>
<%@ include file="Header.jsp" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="javax.servlet.jsp.*" %>

<%@ include file="Footer.jsp" %>
```

③ Footer文件 ("Footer.jsp")

```
<%@ page language="java" %>
```

注意：我们从被包含文件中去除了所有HTML和BODY标记。

注：嵌入式注释法对脚本语句（<script>和<include>）都适用。

浏览器显示效果：

① Web SERVICES  
We know how to make SOAP suck less.  
We can help! ②  
Contact us at: likewecare@wickedlysmart.com  
home page ③

考试大纲	344
如果属性是bean	345
标准动作：useBean、getProperty和setProperty	349
可以建立多态的bean引用吗？	354
解决之道：param属性	360
转换性质	363
救星：表达式语言（EL）！	368
使用点号（.）操作符来访问性质和映射值	370
[]提供更多选择（List、数组……）	372
点号操作符与[]的更多细节	376
EL隐式对象	385
EL函数，以及“null”的处理	392
可重用的模板部分-两种“包含”	402
<jsp:forward>标准动作	416
她不知道还有JSTL标记（预习）	417
标准动作和包含复习	418

# 9 强大的定制标记

有时只是EL或标准动作还不够。如果你想迭代处理一个数组中的数据，并在一个HTML表中每行显示一项，该怎么做？你很清楚，如果在一个scriptlet中使用for循环，只需两秒钟这个问题就能迎刃而解。不过你想尽量避开脚本。没问题。倘若EL和标准动作力不能及，你还可以使用定制标记。在JSP中，使用定制标记与使用标准动作同样简单。更妙的是，已经有人写了一大堆你很可能需要的定制标记，并把它们打包在JSP标准标记库（JSTL）中。在这一章中，我们将学习如何使用定制标记，下一章再介绍如何创建我们自己的定制标记。



考试大纲	440
不用脚本就能实现循环<code><c:forEach></code>	446
使用<code><c:if>和<c:choose></code>完成条件控制	451
使用<code><c:set>和<c:remove></code>标记	455
有了<code><c:import></code>，现在有3种包含内容的方法	460
定制所包含的内容	462
用<code><c:param>做同样的事情	463
<code><c:url></code>可以满足所有超链接需求	465
建立你自己的错误页面	468
<code><c:catch></code>标记。有点像try/catch……	472
如果需要一个不在JSTL中的标记怎么办？	475
注意<code><rtexprvalue></code>	480
标记体里能放什么	482
标记处理器、TLD和JSP	483
taglib <code><uri></code>只是一个名字，而不是一个位置	484
如果JSP使用了多个标记库	487

# 10 JSTL也有力不能及的时候……

有时JSTL和标准动作还不够。你需要些定制的东西，但又不想走老路去写脚本，那你可以编写自己的标记处理器。这样一来，页面设计人员就能在他们的页面中使用你的标记，所有艰苦的工作都由你的标记处理器类在后台完成。不过，构建自己的标记处理器有3种不同的方法，所以要学的还很多。在这3种方法中，有两种（简单标记和标记文件）是在JSP 2.0中新引入的，它们能让你的日子更好过。

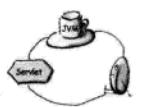


# 11 部署Web应用

Web应用终于到最后的重要时刻了。你的页面完美无瑕，代码也经过了测试和调优，最后期限已经过去了两个星期。但是所有这些东西放到哪里呢？这么多目录，这么多的规则。你怎么对目录命名？客户又怎么认为？客户实际请求的是什么？容器怎么知道到哪里查找？

**本地bean的引用** 代码示例展示了JNDI条目名  
`<jb-local-ref>`  
`<ejb-ref-name>/jb/customers/ejb-ref-name`  
`<ejb-ref-type>Stateless</ejb-ref-type>`  
`<local-home>com.wickedbyte.CustomerHome</local-home>`  
`<local-bean>com.wickedbyte.Customer</local-bean>`

这里必须是bean的部署ID的完全别名



**远程bean的引用** 代码示例展示了JNDI条目名  
`<jb-ref>`  
`<ejb-ref-name>/jb/localCustomer/ejb-ref-name`  
`<ejb-ref-type>Stateless</ejb-ref-type>`  
`<remote-home>com.wickedbyte.CustomerHome</remote-home>`  
`<connection-ref>com.wickedbyte.Customer</connection-ref>`



## 考试大纲

部署的核心任务：各个部分要放在哪里？

## WAR文件

servlet映射究竟如何工作？

在DD中配置欢迎文件

在DD中配置错误页面

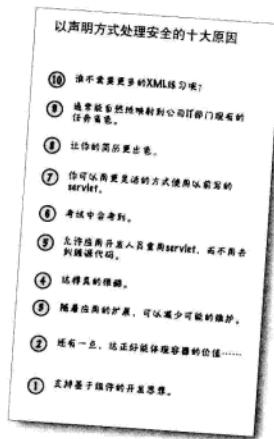
在DD中配置servlet初始化

建立一个XML兼容的JSP：JSP文档

考试大纲	500
标记文件：就像是包含，但比包含更好	502
容器在哪里查找标记文件	509
简单标记处理器	513
有体的简单标记	514
如果标记体使用了表达式呢？	519
还必须了解传统标记处理器	529
一个非常简单的传统标记处理器	531
传统标记生命周期取决于返回值	536
IterationTag允许体重复执行	537
TagSupport的默认返回值	539
DynamicAttributes接口	556
基于BodyTag，你会得到两个新方法	563
如果有些标记要一同工作呢	567
使用标记处理器的PageContext API	577

# 12 要保密, 要安全

你的Web应用危险重重。网络的每个角落都潜伏着危险。你不希望坏家伙监听网上商店的交易，不希望他们窃取信用卡号。你不希望这些坏蛋骗你的服务器，说他们就是那些拿大折扣的大客户。另外你也不希望有人（不管是好人还是坏人）偷看机密的员工数据。市场部的Jim有必要知道工程部的Lisa拿的薪水是他的3倍吗？

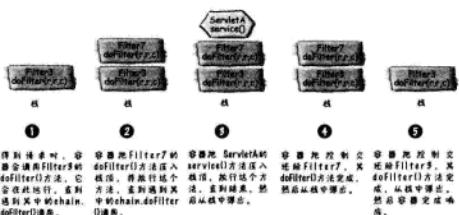


## 考试大纲

考试大纲	650
servlet安全中的4大要素	653
HTTP世界中如何认证	656
以声明方式处理安全的十大原因	659
谁来实现Web应用中的安全？	660
授权：角色和约束	662
认证：4种类型的认证	677
保护正在传输的数据的解决之道：HTTPS	682
如何以声明方式保守地实现数据机密性和完整性	684

# 13 过滤器的威力

过滤器允许你拦截请求。如果能拦截请求，那你还可以控制响应。最棒的是，servlet对此一无所知。它不知道在客户做出请求和容器调用servlet的service()方法之间已经有人介入。对你来说，这意味着什么？这说明你会有更充裕的假期！因为照往常你可能要重写某个servlet才行，但现在大可不必，只需编写和配置一个过滤器，它就能影响所有的servlet。想为应用中的每个servlet都增加用户请求跟踪吗？没问题。想要管理应用中每个servlet的输出吗？也没问题。而且，你甚至不用“接触”servlet代码。

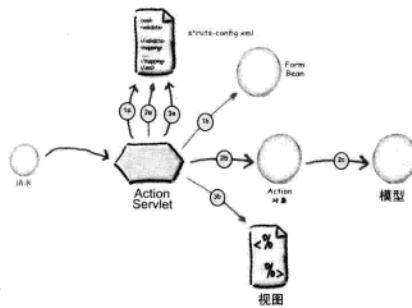


## 考试大纲

考试大纲	702
构建一个请求跟踪过滤器	707
过滤器的生命周期	708
声明过滤器，确定过滤器调用的顺序	710
用响应端过滤器压缩输出	713
包装器	719
具体的压缩过滤器代码	722
压缩包装器代码	724

# 14 企业设计模式

已经有人做过了。如果你刚开始用Java开发Web应用，那你可真是幸运。已经有数百万的开发人员在这条路上摸索了很久，你能轻松地享用到他们智慧和经验的结晶。通过使用J2EE特有的设计模式以及其他设计模式，可以大大简化你的代码，你也能更轻松。Web应用最重要的设计模式是MVC，对此甚至还有一个相当流行的框架Struts。利用这个框架，你能构建一个灵活而且可维护的servlet前端控制器。一定要充分利用别人的成果，这样你才能把宝贵的时间用在更重要的事情上……



考试大纲	738
推动模式发展的软件和硬件原因	739
软件设计原则复习	744
支持远程模型组件的模式	745
JNDI和RMI概述	747
业务委托是一个“中间人”	753
该讲到传输对象了吧？	759
业务层模式速览	761
再来看介绍过的第一个模式……MVC	762
没错！实际上这正是Struts（前端控制器）	767
针对Struts重构啤酒应用	770
模式复习	778



**最终模拟测验。**终于到它了。69个问题。语调、内容和难度都和实际考试几乎相同。这一点我们很清楚（因为考试试题就是我们开发的）。

最终模拟测验	791
答案	828



## 索引

# 为什么使用Servlet & JSP?



Web应用炙手可热。确实，GUI应用可以使用一些新奇古怪的Swing组件（widget），不过，你知道有几个GUI应用能由全世界数百万的用户使用？作为一个Web应用开发人员，不用像对付独立应用那样，亲自处理棘手的部署问题，完全可以通过浏览器把你的应用交付给任何人。不过，要想构建一个真正强大的Web应用，你还需要一些帮手，这就是Java、Servlet和JSP。因为原先普通的静态HTML页面太……怎么说呢，只能算是1999年的老古董了。如今，用户需要的网站应该是动态的、交互式的，而且应该能够由客户定制。通过后面的介绍，你会了解如何把Web网站变成真正的Web应用。

# OBJECTIVES

## Servlet & JSP概述

1.1 对于每一种HTTP方法（如GET、POST、HEAD等）：

- \* 介绍这种HTTP方法的优点
- \* 介绍这种HTTP方法的功能
- \* 列出客户（通常是一个Web浏览器）会出于哪些原因使用这个方法

大纲1.1还涉及以下内容，不过这个内容不在第1章中介绍：

- \* 明确与各个HTTP方法对应的HttpServlet方法

## 内容说明：

这一部分的要求将在另一章中做充分的讲解，所以可以把这一章看作是一个热身，为下一章做准备。换句话说，如果读完这一章时还不了解（或者没记住）大纲要求的这些内容，你也不用着急；这一章相当于一个背景介绍。如果你已经了解了这些知识，可以跳过这一章，直接去看第2章。

第1章没有针对这些要求给出模拟题，等到后面更具体地介绍了这些内容之后，才会让你做些模拟题练练手。

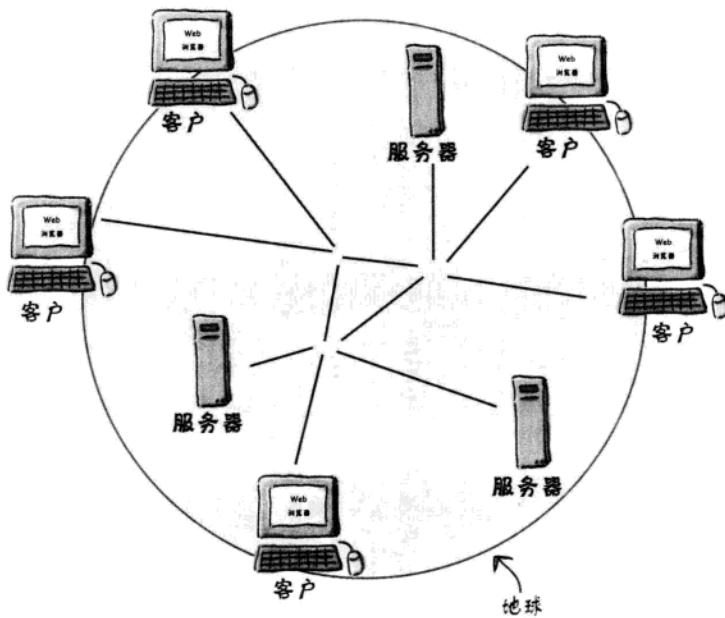
# 每个人都想有个网站

你可能想要一个超级网站。要打败竞争对手，你需要一个灵活而且可扩展的体系结构。这就需要Servlet和JSP。

在构建具体的Web应用之前，先让我们远远地看看WWW（World Wide Web，万维网）是什么。这一章最关心的就是Web客户和Web服务器怎么对话。

后面几页的内容对你来说可能并不陌生，特别是如果你已经是一个Web应用开发人员，肯定早就知道这些了，不过，通过这些介绍，我们会提到本书中将要用到的一些术语。

Web包括数以亿计的客户（使用像Mozilla或Safari之类的浏览器）和服务器（使用Apache等等Web服务器应用），这些客户和服务器之间通过有线和无线网络连接。我们的目标就是构建一个全世界客户都能访问的Web应用。直白一点，就是要插大线。



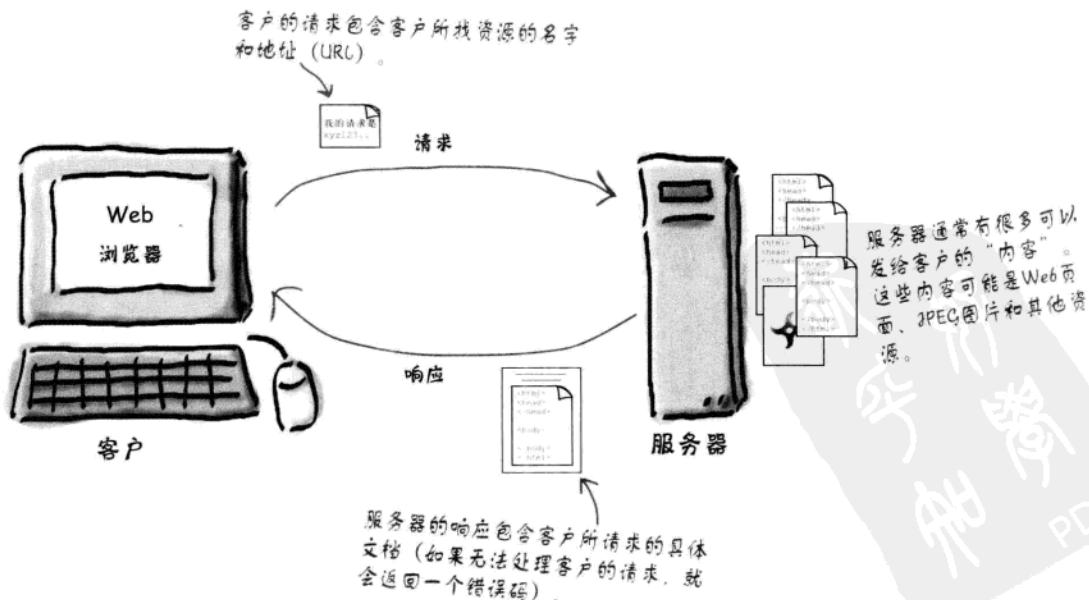
# 你的Web服务器做些什么？

Web服务器接收客户请求，然后向客户返回一些结果。

用户可以通过Web浏览器请求一个资源。Web服务器得到请求后，查找资源，然后向用户返回一个结果。有时资源可能是一个HTML页面，有时可能是一个图片，也可能是一个声音文件，甚至是一个PDF文档。具体是什么没有关系，总之都是客户请求某个东西（资源），再由服务器返回所请求的资源。

除非没有这样一个资源，或者至少资源不在服务器原来预想的位置上，此时就会显示一个“404 Not Found (404: 未找到)”错误，这个错误你肯定已经很熟悉了。如果服务器找不到你请求的东西，它就会给你这样一个响应。

谈到“服务器”时，可能是指物理主机（硬件），也可能是指Web服务器应用（软件）。在这本书里，如果要区别服务器硬件和软件，我们会特别指出所说的到底是硬件还是软件。



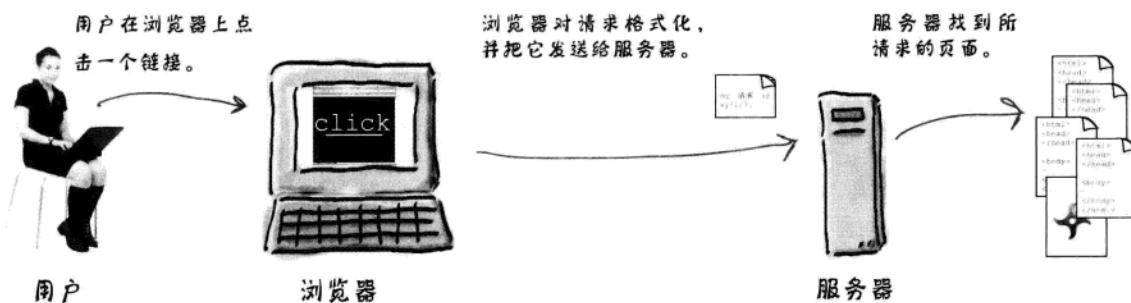
# Web客户做些什么？

Web客户允许用户请求服务器上的某个资源，并且向用户显示请求的结果。

谈到客户时，通常是指人类用户，或者浏览器应用，也可能二者都包括。

浏览器就是一个软件（比如Netscape或Mozilla），它知道怎么与服务器通信。浏览器还有一个重要的任务，这就是解释HTML代码，并把Web页面呈现给用户。

所以，从现在开始，如果我们说到“客户”，通常并不强调指的是人类用户还是浏览器应用。换句话说，我们所说的客户就是浏览器应用，它能完成用户指定的任务。



# 客户和服务器都知道 HTML和HTTP

## HTML

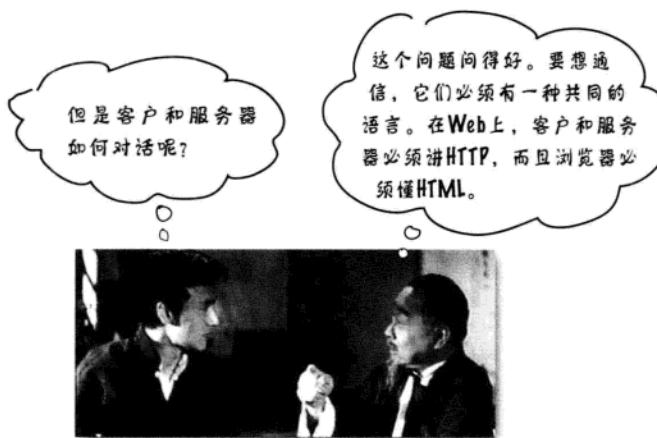
当服务器对一个请求做出回答时，通常会向浏览器发送某种类型的内容，以便浏览器显示。服务器一般会向浏览器发送一组用HTML编写的指令，HTML就是超文本标记语言（Hyper-Text Markup Language）。HTML告诉浏览器怎样把内容呈现给用户。

所有Web浏览器都知道如何处理HTML，不过，如果页面是用更新版本的HTML编写的，有时老版本的浏览器可能无法完全理解。

## HTTP

Web上客户和服务器之间的大多数会话都是使用HTTP协议完成的，HTTP协议支持简单的请求和响应会话。客户发送一个HTTP请求，服务器会用一个HTTP响应做出应答。关键是：如果你是一个Web服务器，就必须讲HTTP。

Web服务器向客户发送HTML页面时，就是使用HTTP发送的（通过后面几页的详细介绍，你会了解这是怎么做到的）。



HTML告诉浏览器  
怎样向用户显示内  
容。

HTTP是Web上客户  
和服务器之间进行  
通信所用的协议。

服务器使用HTTP向  
客户发送HTML。

# HTML速成指南

开发Web页面时，就是用HTML描述页面应该是什么样子，以及它有怎样的表现。

HTML包含数十个标记，还有成百上千个标记属性。HTML的目标是拿到一个文本文档，然后为它增加一些标记，告诉浏览器如何对这个文本格式化。下面列出了后面几章我们将要使用的一些标记。如果你想更全面地了解HTML，建议你读一读《HTML & XHTML The Definitive Guide》。

标记	描述
<!-- -->	在这里加注释
<a>	锚点——通常用来放一个超链接
<align>	对内容左对齐、右对齐、居中，或调整行距
<body>	定义文本体的边界
 	行分隔
<center>	将内容居中
<form>	定义一个表单（通常提供了输入域）
<h1>	一级标题
<head>	定义文档首部的边界
<html>	定义HTML文档的边界
<input type>	在表单中定义一个输入组件
<p>	一个新段落
<title>	HTML文档的标题

理论上讲，`<center>` 和 `<align>` 标记在HTML 4.0中已经废弃不用了。不过在我们的一些示例中还是用到了这两个标记，因为与相应的新标记相比，它们读起来更简单，而且毕竟我们不是在专门学习HTML。

你要写的东西……  
(HTML)

假设你在创建一个登录页面。最简单的HTML可能是这样的：

这是一个HTML注释

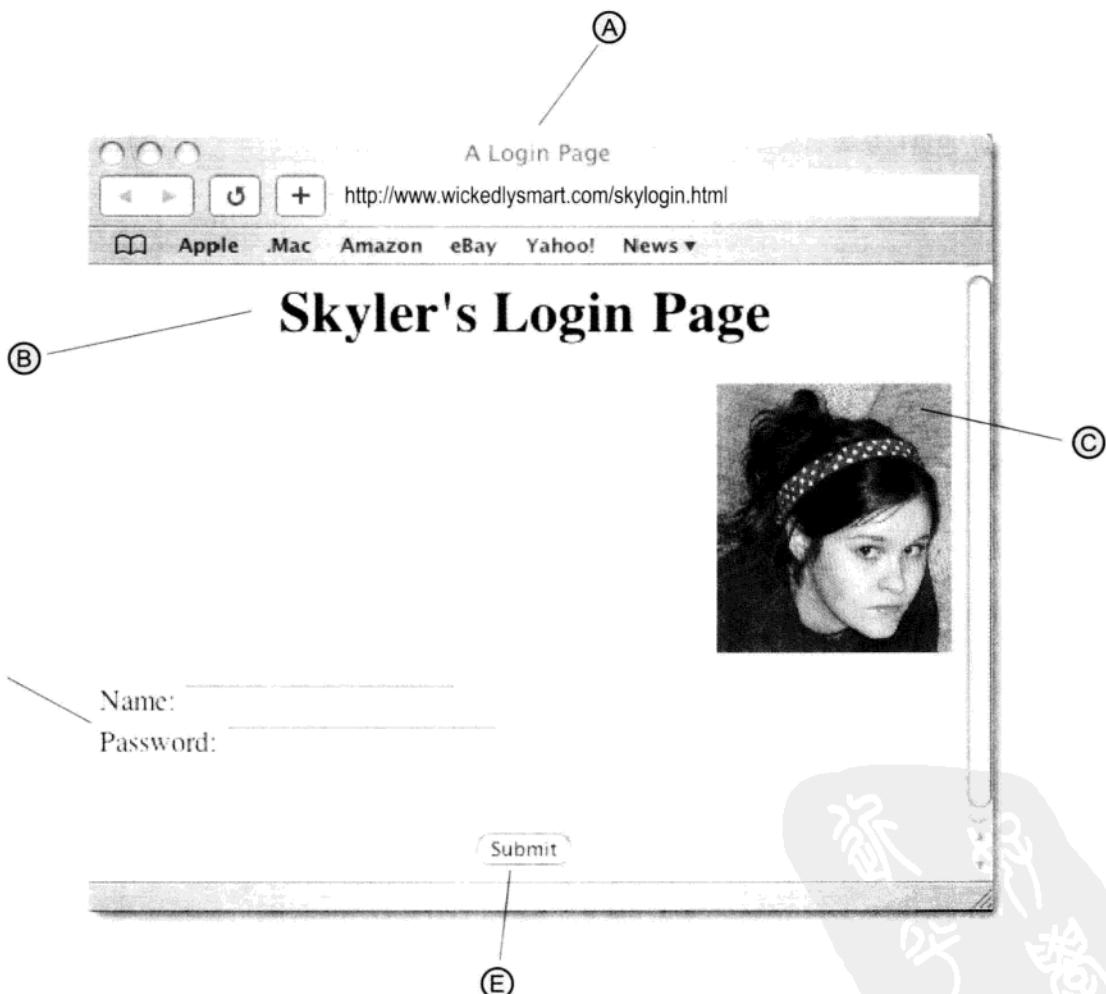
① <html>  
  <!-- Some sample HTML --> ←  
  <head>  
    <title>A Login Page</title>  
  </head>  
  <body>  
    <h1 align="center">Skyler's Login Page</h1>  
  
    <p align="right"> ←  
        
    </p>  
    要 把 请求 发送 到 这个  
    <form action="date2"> ←  
      Servlet.  
  
② Name: <input type="text" name="param1"/><br/>  
Password: <input type="text" name="param2"/><br/><br/><br/>  
  
<center>  
  <input type="SUBMIT"/>  
</center>  
</form>  
表单上的 "submit"  
(提交) 按钮。 ←  
  
③ 后面还会更多地介绍表单。  
不过简单地讲，浏览器会  
收集用户的输入，并把这  
些输入返回给服务器。  
  
<br/> 标记会导致  
换行。  
  
</body>  
</html>

 Relax 你只需要了解最基本的HTML知识。

考试中会大量出现HTML，但不会考你有关HTML的知识。不过，确实在很多问题中都能看到HTML，所以，看到简单的HTML时你至少应当能知道会发生什么。

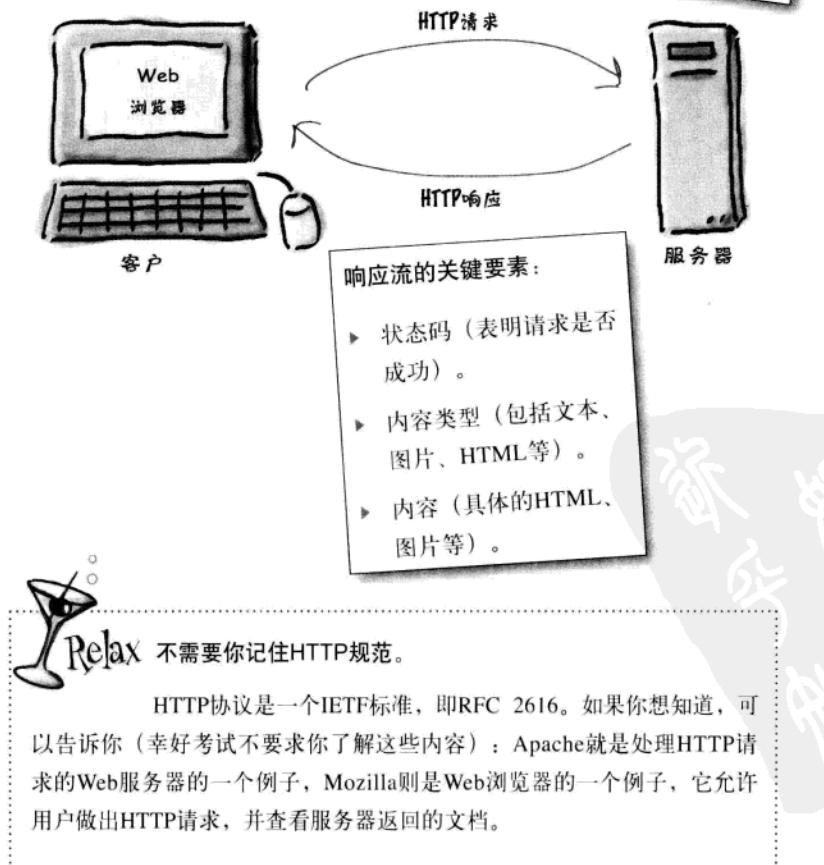
# 浏览器要创建的东西……

浏览器读取HTML代码，创建Web页面，并显示给用户。



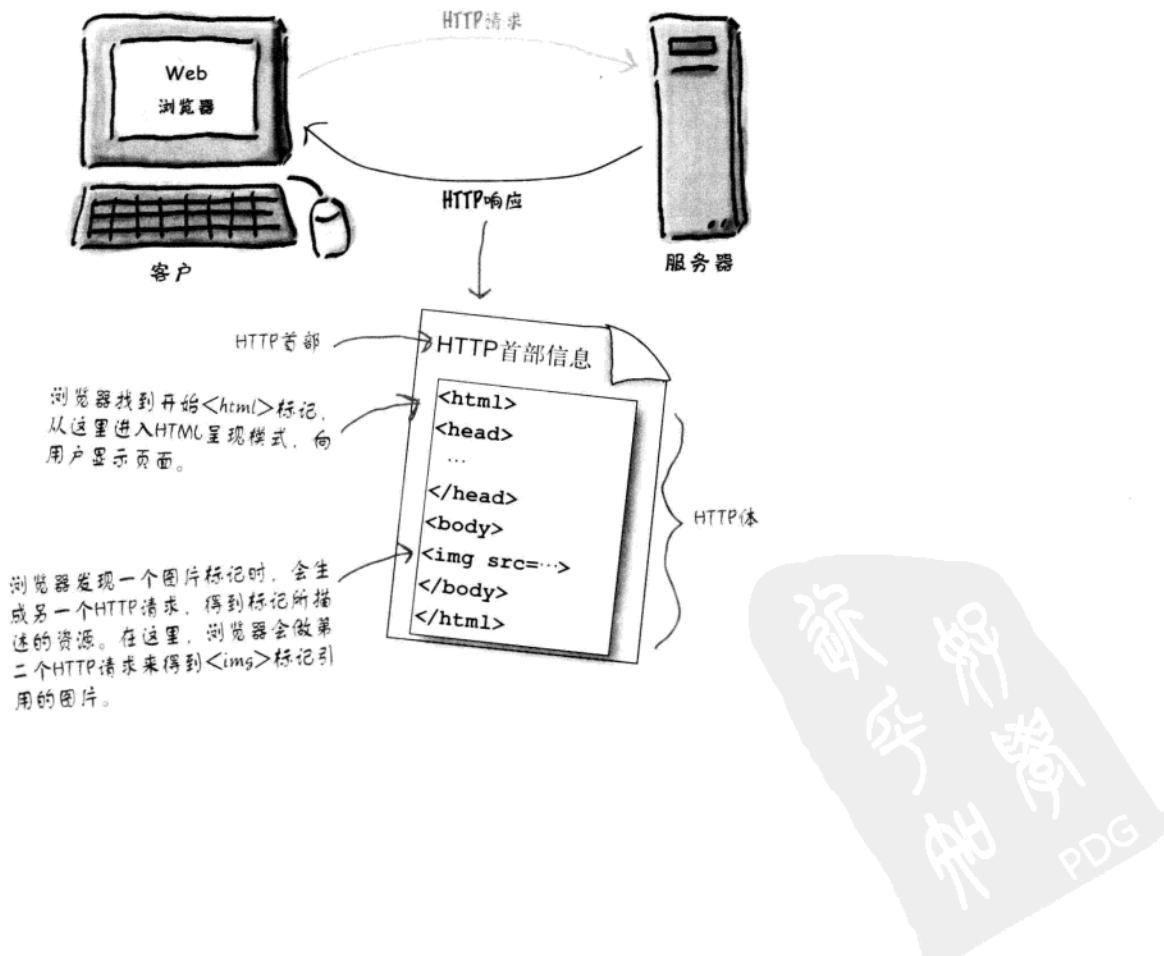
# 什么是HTTP协议？

HTTP是TCP/IP的上层协议。如果你对这些网络协议还不太熟悉，下面就提供一个非常简单的解释：TCP负责确保从一个网络节点向另一个网络节点发送的文件能作为一个完整的文件到达目的地，尽管在具体传送过程中这个文件可能会分解为小块传输。IP是一个底层协议，负责把数据块（数据包）沿路移动/路由到目的地。HTTP则是另一个网络协议，有一些Web特定的特性，不过它要依赖于TCP/IP从一处向另一处完整地传送请求和响应。HTTP会话的结构是一个简单的请求/响应序列：浏览器发出请求，服务器做出响应。



# HTML是HTTP响应的一部分

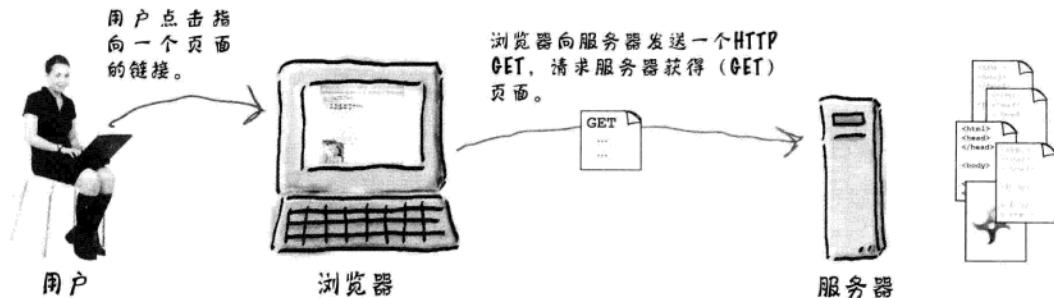
HTTP响应可以包含HTML。HTTP还会在响应中包含的内容（换句话说，就是服务器返回的东西）前面增加首部信息。HTML浏览器使用首部信息来帮助处理HTML页面。可以把HTML内容看作是粘贴到HTTP响应中的数据。



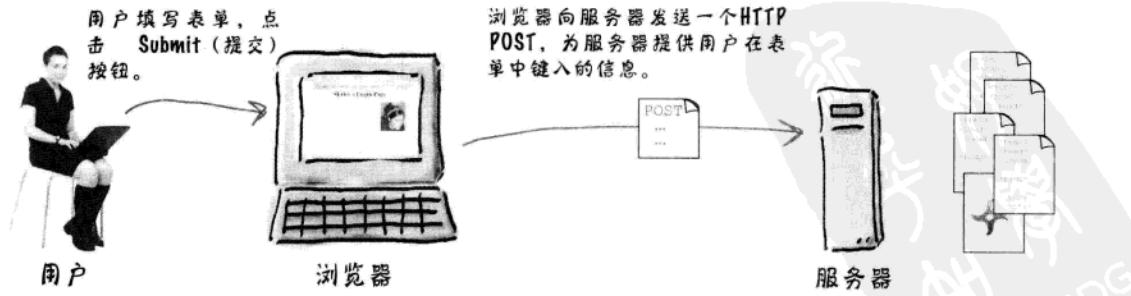
## 如果响应是这样，那么请求里又是什么呢？

首先会看到一个HTTP方法名。这不是Java方法，但是道理是一样的。方法名告诉服务器做了哪一类请求，并指出消息中余下的部分该如何格式化。  
HTTP协议有许多种方法，但最常用的当属GET和POST。

### GET



### POST



# GET是一个简单的请求， POST可以发送用户数据

GET是最简单的HTTP方法，它的主要任务就是要求服务器获得一个资源并把资源发回来。这个资源可能是一个HTML页面、一个JPEG、一个PDF文档等等。具体是什么资源没有关系，关键是，GET就是要从服务器拿些东西回来。

POST是一种更强大的请求，就像是GET++。利用POST，可以请求某个东西，与此同时向服务器发送一些表单数据（在本章的后面，我们会解释服务器对这些数据如何处理）。

*there are no  
Dumb Questions*

**问：**除了GET和POST，还有其他的HTTP方法吗？

**答：**GET和POST是大家使用最多的两个方法。不过，除此之外，确实还有其他一些很少用的方法（而且Servlets确实能够处理这些方法），具体包括HEAD、TRACE、PUT、DELETE、OPTIONS和CONNECT。

对考试来说，不要求你对这些很少用的方法知道多少，不过，这些方法也有可能在某个问题中出现。“Servlet的生与死”一章中会详细介绍需要你了解的其余HTTP方法的细节。

稍等片刻，我打赌肯定  
见过能向服务器发送一  
些参数数据的GET请求。



# 不错……确实能用HTTP GET发送一点点数据

但是你可能不想这样做。使用POST而不是GET的原因如下：

- ① GET中的总字符数是有限的（取决于服务器）。比如，如果用户在“搜索”输入框里键入了很长的一段文字，GET方法可能无法正常工作。
- ② 用GET发送的数据会追加到URL的后面，在浏览器地址栏中显示出来，所以你发送的数据会完全暴露。最好不要把口令或其他敏感数据作为GET请求的一部分发送！
- ③ 由于前面的第2点，如果你使用POST而不是GET，用户就不能对一个表单提交建立书签。取决于具体的应用，你可能希望（或不希望）用户对表单提交所得到的请求建立书签。

额外参数之前的URL。

“?”将路径和参数（额外的数据）分隔开。GET发送的数据量是有限的，而且所有数据都暴露在浏览器地址栏里，每个人都能看到。这两部分加在一起，整个串才是随请求发送的URL。

The screenshot shows a web browser displaying the JavaRanch Big Moose Saloon. The URL in the address bar is [http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get\\_topic&f=18&t=003475](http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=18&t=003475). The page title is "JavaRanch Big Moose Saloon: Books for the SCWCD 1.4 ??".

The main content area displays a forum topic titled "How Clover really learned polymorphism." with a thumbnail image featuring a moose head and a dog. Below the thumbnail are two "Post New Topic" buttons.

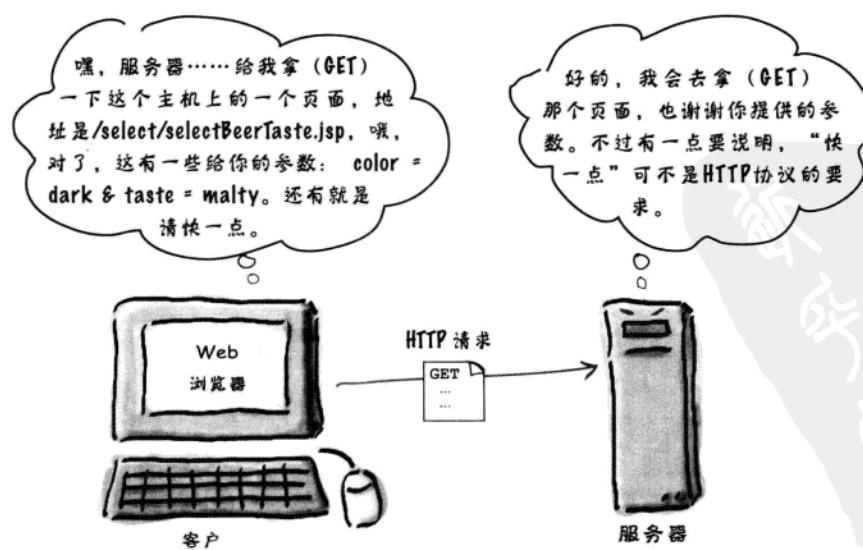
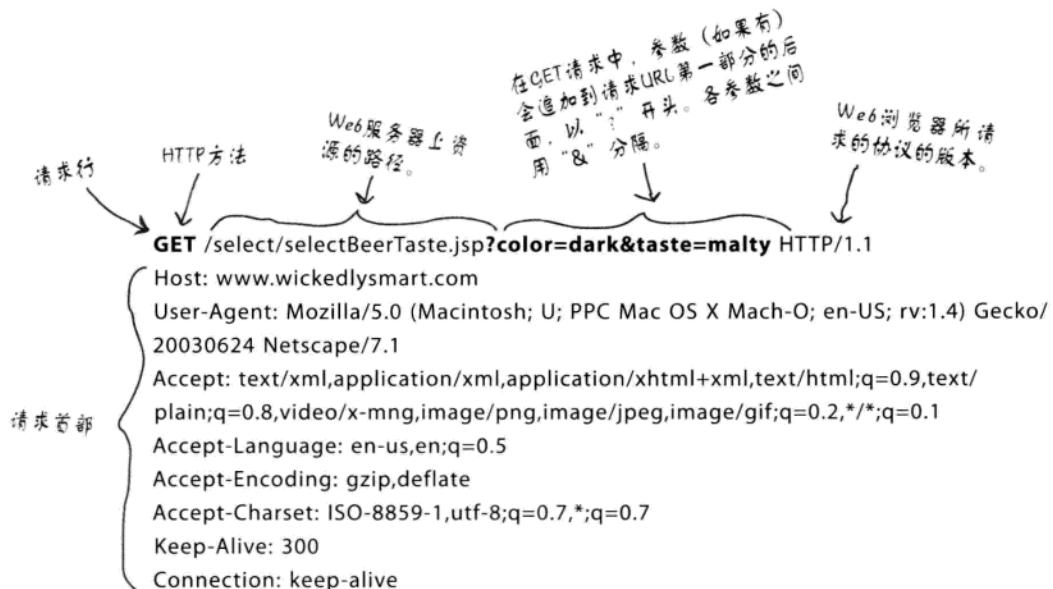
On the left, there's a sidebar with a moose icon and a link to "Hello, Kathy Sierra [log out]". On the right, there are navigation links for "My Profile | Register | Search | FAQ | Current Month | Previous | Next".

The forum post by "Bill Wilde" (greenhorn Member) is visible, dated February 05, 2004, at 09:15 AM. The post content asks for advice on books for SCWCD 1.4 preparation. A response from "Kathy Sierra" (sheriff) dated February 05, 2004, at 09:25 AM, also asks for book recommendations for the new SCWCD 1.4 exam.

A large watermark with the text "JavaRanch" and "PDG" is overlaid on the bottom right of the screenshot.

# HTTP GET 请求剖析

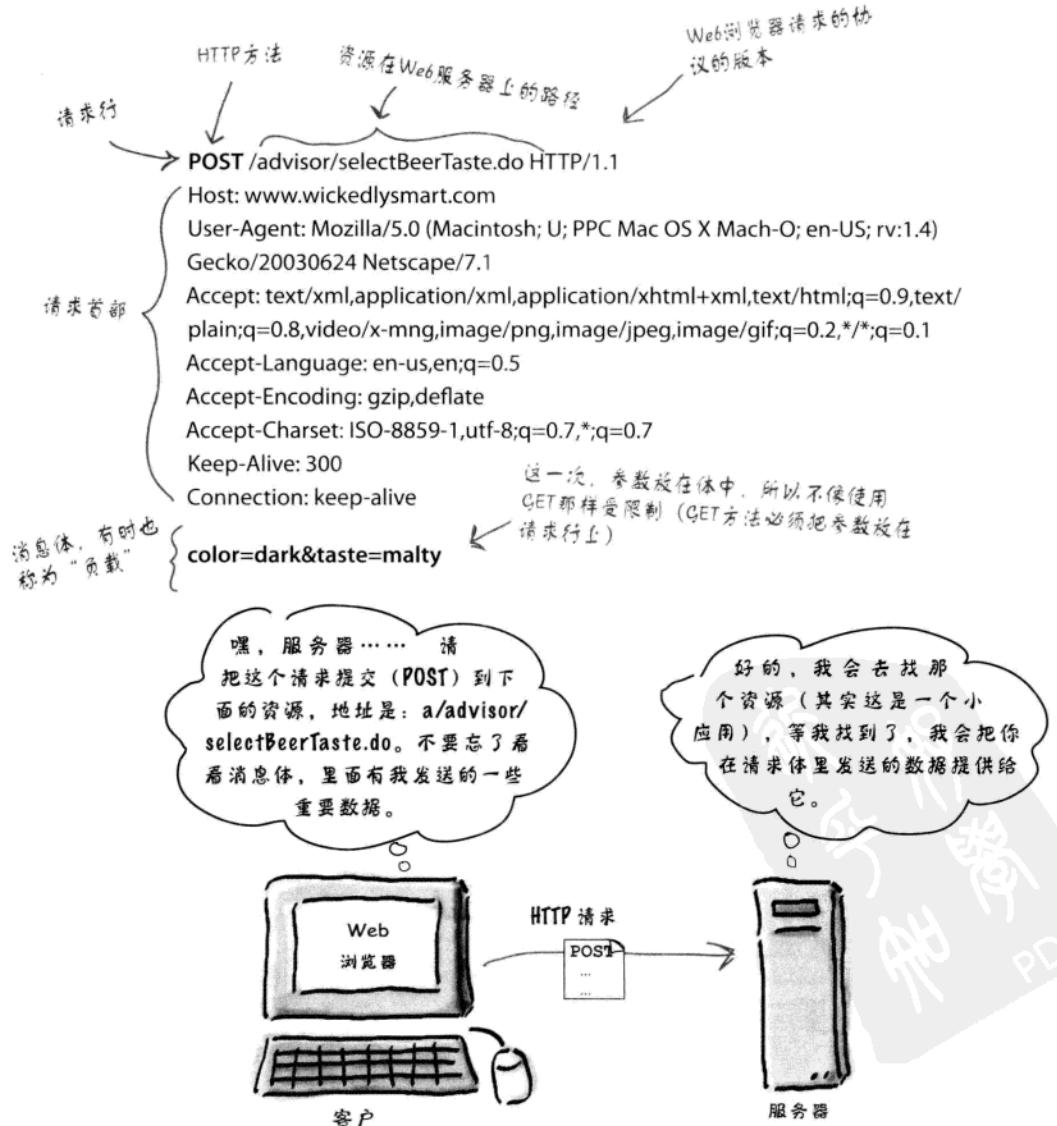
资源的路径以及增加到URL的所有参数都会包括在“请求行”中。



# HTTP POST 请求剖析

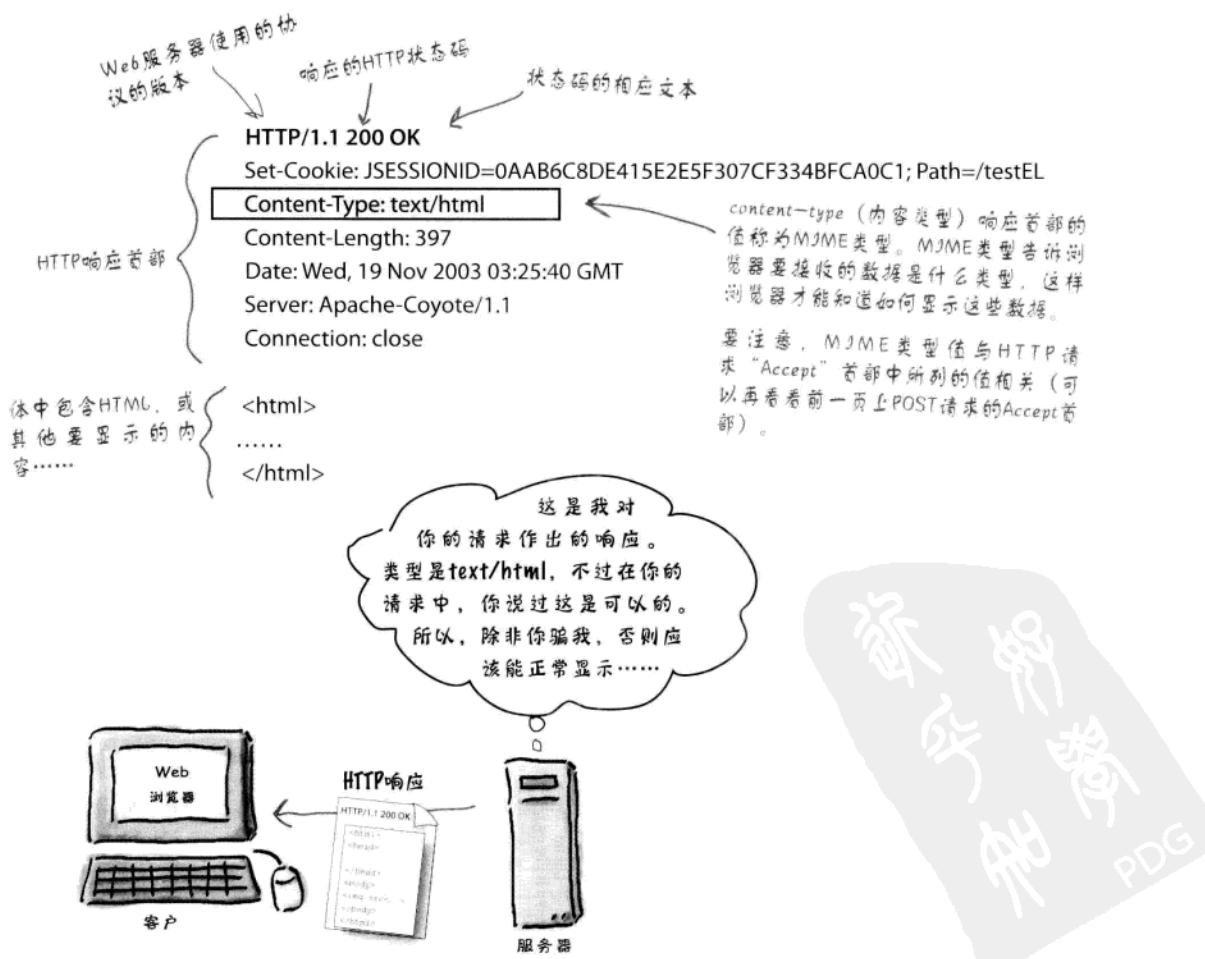
HTTP POST请求设计为：浏览器可以用它向服务器做复杂的请求。

例如，如果用户刚完成了一个很长的表单，应用可能希望把表单的所有数据都增加到一个数据库中。发回给服务器的数据称为“消息体”或“负载”，这个消息体可以非常大。

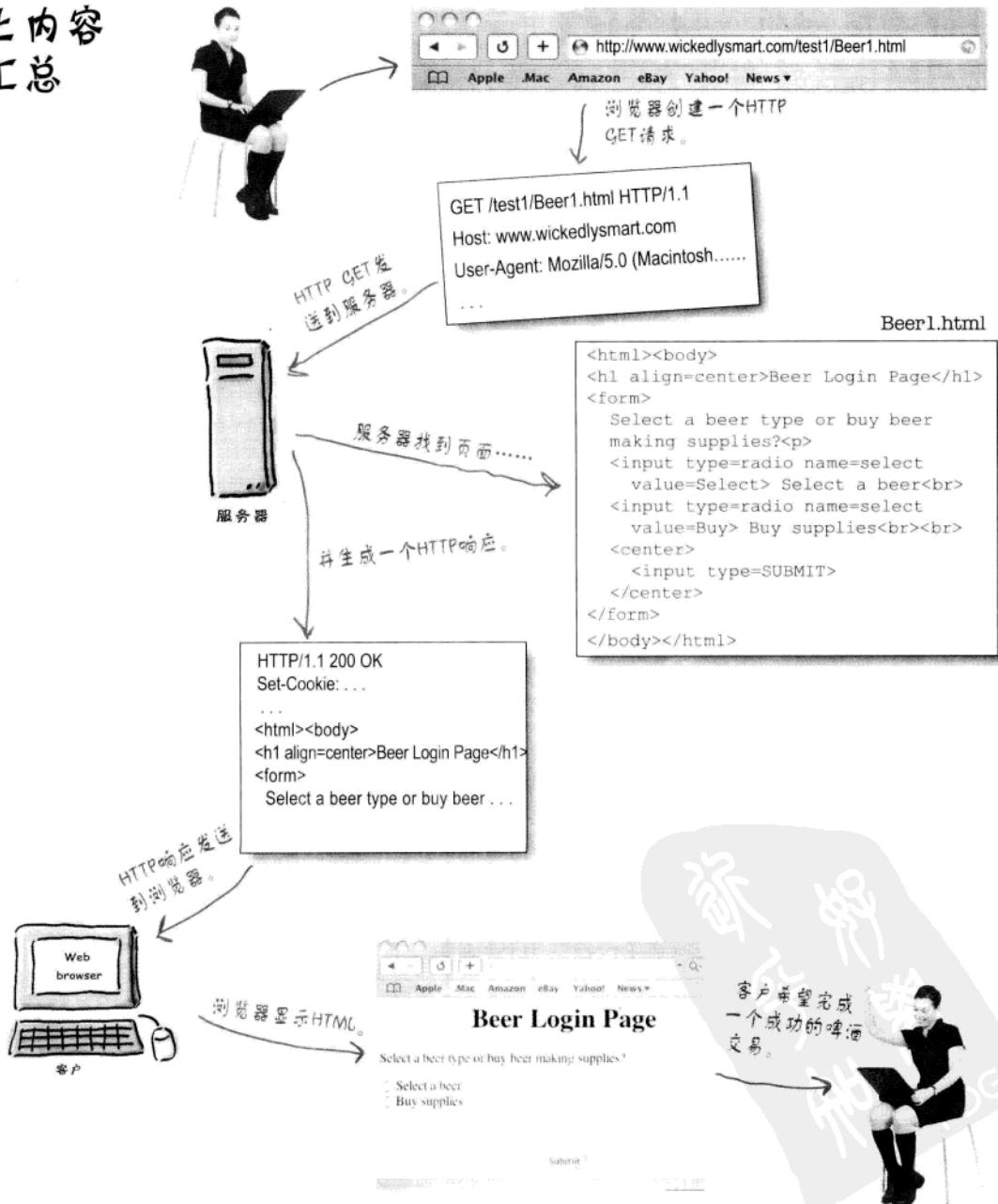


# HTTP响应剖析，到底什么是“MIME类型”？

前面已经了解了从浏览器向服务器发送的请求，下面来看看服务器做出响应时会返回什么。HTTP响应包括一个首部和一个体。首部信息告诉浏览器使用了什么协议，请求是否成功，以及体中包括何种类型的内容。体包含了让浏览器显示的具体内容（例如，HTML）。



## 以上内容的汇总





## GET还是POST?

对以下描述，看看应该选择哪个HTTP方法来实现相应功能，并根据你的判断圈出POST或GET。如果你认为两种方法都可以，就将两个方法都圈出。不过，要想好为什么做这样的选择……

- POST    GET    用户要返回一个登录名和口令。
- POST    GET    用户通过超链接请求一个新页面。
- POST    GET    聊天室用户发送一个写好的响应。
- POST    GET    用户点击“next”（下一页）按钮来查看下一页。
- POST    GET    用户在一个安全银行网站上点击“log out”（注销）按钮。
- POST    GET    用户在浏览器上点击“back”（后退）按钮。
- POST    GET    用户向服务器发送一个包括名字和地址的表单。
- POST    GET    用户要做一个单选钮选择。

# URL。无论如何，别把它拼成“Earl”就行。

如果在缩略语词典里查U，你会看到一大堆以U打头的缩略词，例如，URI、URL、URN等，一眼望不到头。现在我们重点介绍URL，也就是统一资源定位符（Uniform Resource Locatios），这是我们熟知而且热爱的。Web上的每个资源都有唯一的地址，采用的就是URL格式。





Off the path

## TCP端口只是一个数字而已

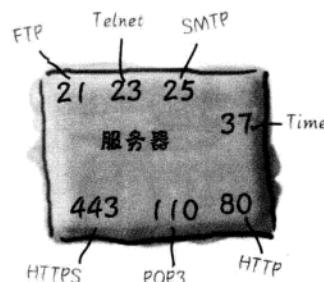
端口是一个16位数，标识服务器硬件上一个特定的软件程序。

Internet Web (HTTP) 服务器软件在端口80上运行。这是一个标准。如果你有一个Telnet服务器，它会在端口23上运行。FTP呢？端口21。POP3邮件服务器呢？端口110。Time服务器在端口37运行。可以把端口看作是唯一的标识符。端口号表示与服务器硬件上运行的一个特定软件的逻辑连接。仅此而已。别指望能在你的硬件主机里找到一个TCP端口。一方面，服务器上有65536个端口（0~65535）。另一方面，端口并不表示一个可以插入物理设备的位置。它们只是表示服务器应用的“逻辑”数而已。

如果没有端口号，服务器就没有办法知道客户想连接哪个应用。而且，由于每个应用可能有自己特有的协议，想想看，如果没有这样一些标识符该是多么麻烦。例如，如果你的Web浏览器访问的是POP3邮件服务器，而不是HTTP服务器，会怎么样呢？邮件服务器不知道如何解析一个HTTP请求！即使POP3服务器确实能解析这样一个请求，它也不知道如何发回一个HTML页面。

如果你要编写服务（服务器程序），想在一个公司网络上运行，就应该咨询系统管理员，检查一下哪些端口已经被占用。系统管理员可能会提供一些建议，例如，告诉你不能使用3000以下的端口号。

众所周知的常用服务器应用TCP端口号



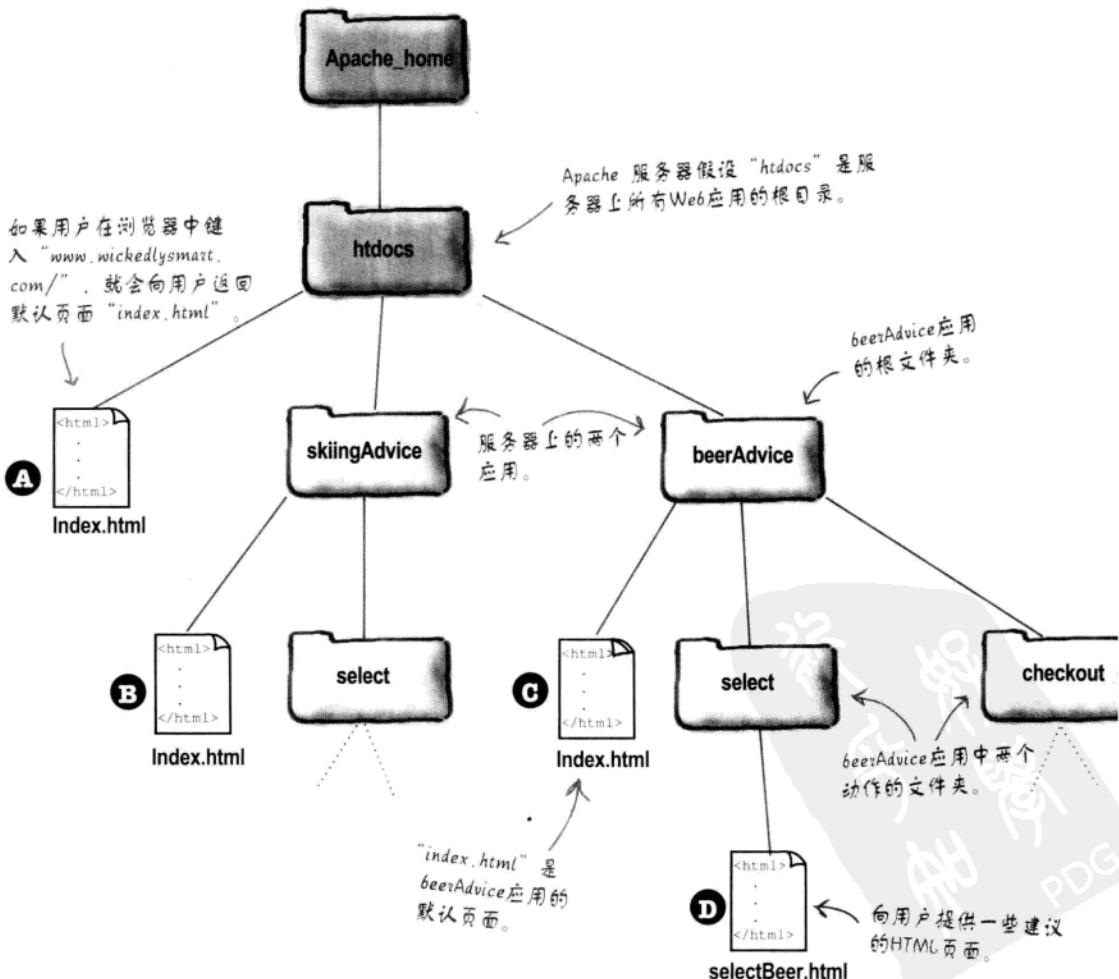
如果每个端口上使用一个服务器应用，一个服务器上最多可以运行65536个不同的服务器应用。

**必须记住：1023的TCP端口号已经保留，由一些众所周知的服务使用（包括我们最关心的“NO.1”——端口80）。你自己的定制服务器程序不要使用这些端口！**

# 一个简单的Apache网站的目录结构

后面会更多地谈到Apache和Tomcat，不过，现在先假设有一个简单的Web网站在使用Apache（这是一个非常流行的开源Web服务器，没准你已经在用了）。如果一个名为www.wickedlysmart.com的网站包含两个应用，一个应用提供滑雪建议，另一个应用提供有关啤酒的建议，这样一个网站的目录结构是怎样的呢？假设Apache应用在端口80上运行。

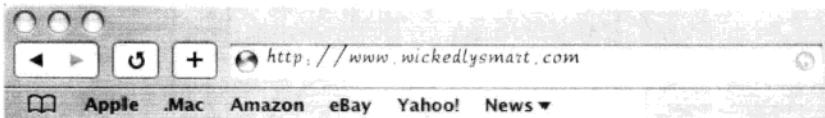
以下的.html页面都标上了一个字母（A、B、C和D），分别对应下一页上的各个练习。



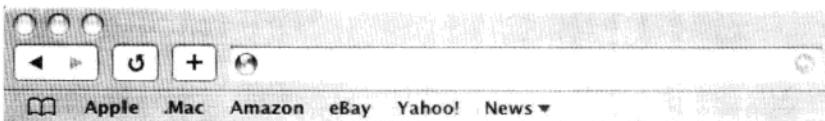


## 将URL映射到内容

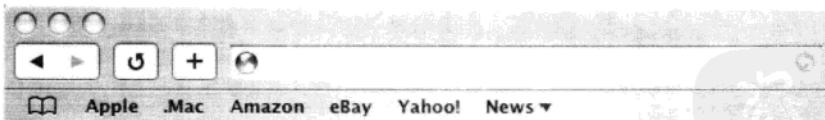
先看看前一页上的目录结构，要得到目录结构中标为A、B、C和D的4个.html页面，请在下面分别填写相应的URL。我们已经好心地填出了第一个URL（A）。这个练习中，假设Apache在端口180上运行（答案在下一页的最下面）。



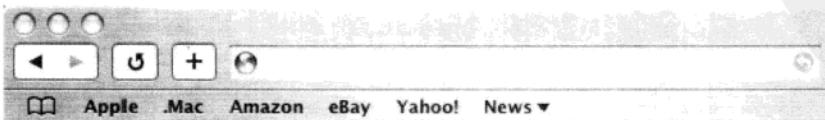
使服务器返回**A**位置上的index.html页面。



使服务器返回**B**位置上的index.html页面。



使服务器返回**C**位置上的index.html页面。



使服务器返回**D**位置上的index.html页面。

# Web服务器擅长提供静态 Web页面



但是如果你想要点别的，比如说，在页面上显示出当前时间，该怎么办？如果我需要一个有点动态的页面，怎么做呢？难道HTML中不能有变量之类的东西吗？



```
<html>
<body>
The current time is [insertTimeOnServer].
</body>
</html>
```

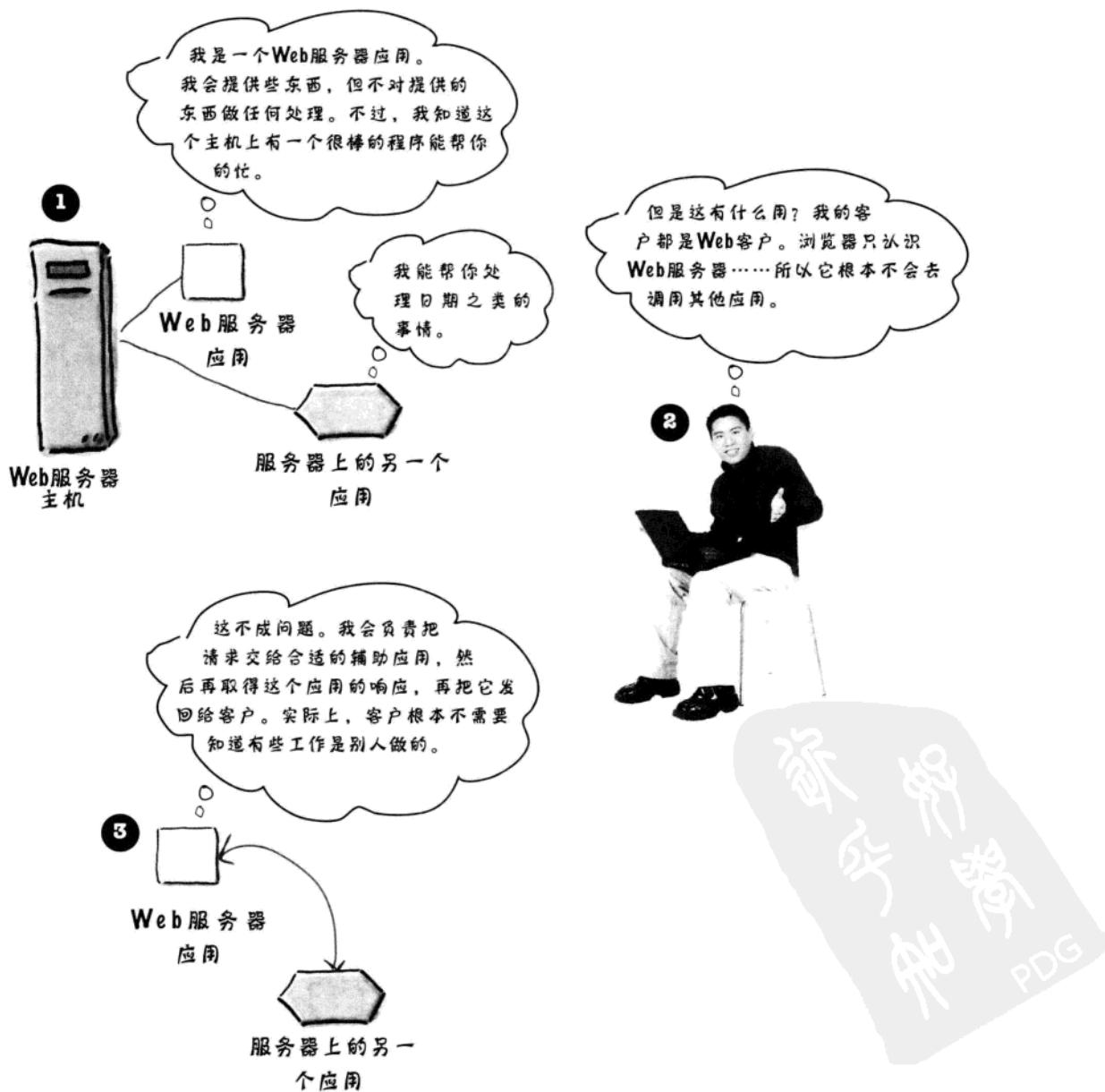
要是我想在HTML页面里加上一些能变化的东西呢：

- D- [www.wickeleyissmart.com/beerAdvice/selectSelectiveBeer.html](http://www.wickeleyissmart.com/beerAdvice/selectSelectiveBeer.html)  
 C- [www.wickeleyissmart.com/beerAdvice/](http://www.wickeleyissmart.com/beerAdvice/)  
 B- [www.wickeleyissmart.com/skiingAdvice/](http://www.wickeleyissmart.com/skiingAdvice/)

唯一的答案：



# 不过，有时不仅仅需要 Web服务器



# Web服务器自己不做的两件事

如果你需要即时（just-in-time）页面（这是动态创建的页面，在发出请求之前还不存在），而且希望能够把数据写至/保存到服务器上（也就是说，写到一个文件或数据库中），只依靠Web服务器是不够的。

## 1 动态内容

Web服务器应用只提供静态页面，但是有一个“辅助”应用可以生成非静态的即时页面，而且这个辅助应用能与Web服务器通信。动态页面可以是一个编目、Web日志，甚至只是一个随机选择显示图片的页面。

可能不希望是这样：

```
<html>
<body>
The current time
is always 4:20 PM
on the server
</body>
</html>
```

而希望是这样：

```
<html>
<body>
The current time is
[insertTimeOnServer]
on the server
</body>
</html>
```

即时页面在请求到来之前  
并不存在，创建这样一个  
HTML页面就像是搭建一  
个“空中楼阁”。

请求到来后，辅助应用

“具体写出”HTML，  
Web服务器再把这个  
HTML交回给客户。

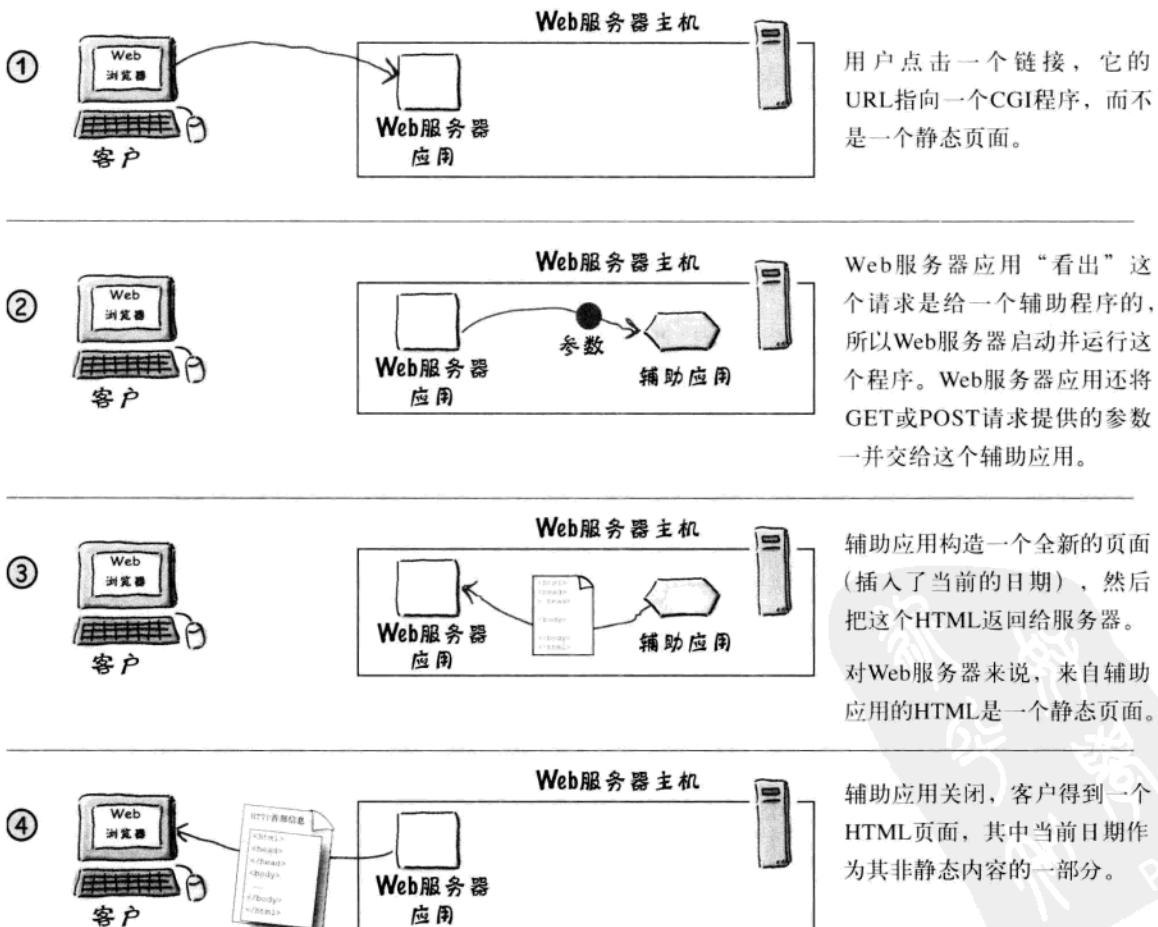
## 2 在服务器上保存数据

用户提交表单中的数据时，Web服务器看到表单数据，然后想，“嗯？是给我的吗？”要处理这个表单数据，可以把它保存到一个文件或数据库中，或者甚至用它生成响应页面，为此需要另一个应用。如果Web服务器认为这是发给一个辅助应用的请求，Web服务器会假设参数是提供给那个应用的。所以Web服务器会把参数移交给辅助应用，由这个应用为客户生成一个响应。

# 如果不按Java术语来讲，Web服务器辅助应用就是“CGI”程序

大多数CGI程序都编写成Perl脚本，不过用其他语言编写也可以，如C、Python和PHP（CGI代表公共网关接口，即Common Gateway Interface，为什么叫这个名字我们并不关心）。

如果使用CGI，下面展示了为什么一个动态Web页面能够显示当前的服务器日期。



## Servlet和CGI在Web服务器中都扮演着辅助应用的角色

请听听两位高手关于CGI与Servlet利与弊的论。

CGI



CGI比Servlet强。我们小组就用Perl写CGI脚本，因为每个人都懂Perl。

你们自己使用Java可能没什么问题，因为你们懂Java呀。但是要让我们都转而使用Java就不值得了。这没有任何好处。

你在抬杠吗？你有什么根据？

Java也好不了多少呀……你是怎么看JVM的？JVM的每个实例都是一个重量级的进程，难道不是吗？

我想你忘了一些事情吧。Web服务器现在能让一个Perl程序跨客户请求运行。所以再谈这方面的开销是没有意义的。

你说什么呢？任何CORBA兼容的应用都可以是J2EE客户。

暂停一下——我上普拉提课要迟到了。不过这不算完。以后再接着说。

Servlet



每个人都懂Perl？这一点我很怀疑。我也喜欢Perl，是我们小组里都是Java程序员，所以我们更喜欢Java。

坦率地说，老兄，要处理CGI，Java比起Perl来说可好处多多呀。

比如说，性能。使用Perl的话，对于同一个资源的一个请求，服务器都必须启动一个重量级的进程！

嗯，不错，但是要知道，Servlet会一直加载，而且果多个客户请求都指向同一个Servlet资源，这些请会作为同一个运行servlet的单独线程来处理。所以有启动JVM、加载类等等开销……

我没有忘，老兄。但并不是所有Web服务器都能做这一点。你说的只是一个特例，并不适用于所有PCGI程序。不过Servlet不同，效率总能更高。而且要忘了，Servlet可以是一个J2EE客户，而Perl CGI程序做不到。

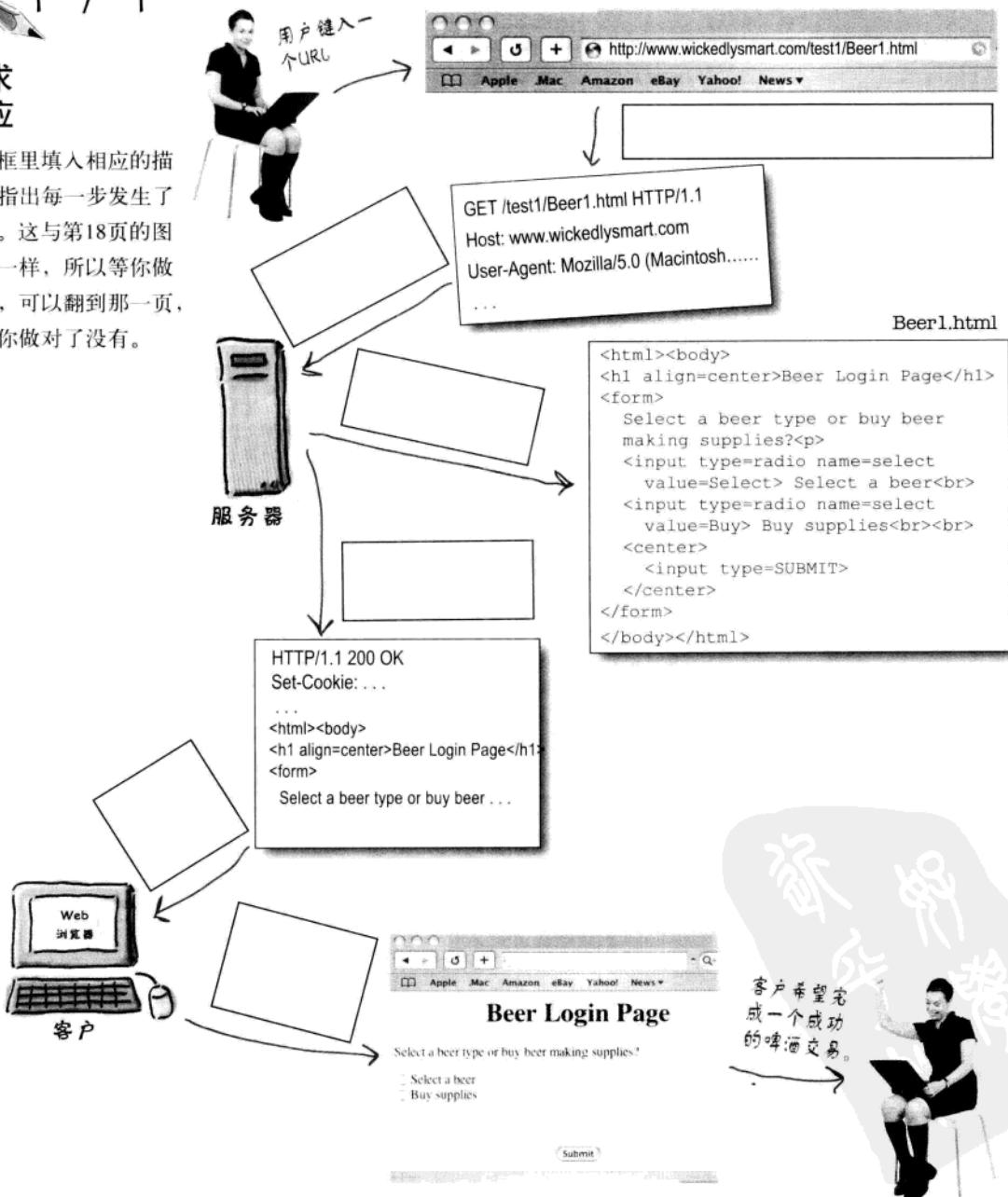
我说的可不是J2EE程序的客户，我的意思是客户本是J2EE。在J2EE Web容器中运行的Servlet可以与企bean一同参与安全和事务，而且——

未完待续...

# Sharpen your pencil

## 请求 响应

在方框里填入相应的描述，指出每一步发生了什么。这与第18页的图完全一样，所以等你做完了，可以翻到那一页，看看你做对了没有。



# Servlet揭秘（编写、部署、运行）

有些人可能还没有接触过Servlet，为了不至于让这些人感到恐惧，下面是如何编写、部署和运行Servlet的一个快速入门。看完下面的介绍后，你可能只是一知半解，甚至会发现更多不懂的问题。不要惊慌，现在还不要求你完全理解。因为有些人等不及想看看Servlet，所以这里只是为他们做一个快速演示。下一章将提供一个更全面的教程。

- ① 建立这样一个目录树（不要放在Tomcat下）。
- ② 编写一个名为Ch1Servlet.java的servlet，把它放在src目录中（为了让这个例子尽可能简单，我们没有把servlet放到一个包里，但是本书后面的所有其他servlet示例都会放入包里）。

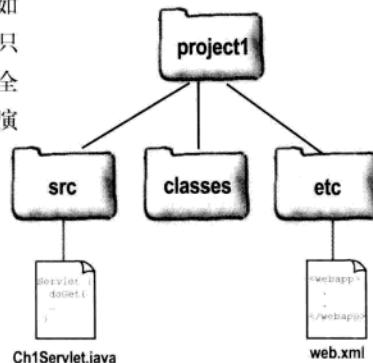
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Ch1Servlet extends HttpServlet{
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException {
        PrintWriter out = response.getWriter();
        java.util.Date today = new java.util.Date();
        out.println("<html> " +
                   "<body>" +
                   "<h1 align=center>HF's Chapter1 Servlet</h1>" +
                   "<br>" + today + "</body>" + "</html>");
    }
}
```

- ③ 创建一个部署描述文件（deployment descriptor, DD），名为web.xml，把它放在etc目录中。

```
<?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_4.xsd"
          version="2.4">
    <servlet>
        <servlet-name>Chapter1 Servlet</servlet-name>
        <servlet-class>Ch1Servlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Chapter1 Servlet</servlet-name>
        <url-pattern>/Servlet</url-pattern>
    </servlet-mapping>
</web-app>
```



标准Servlet声明（我们将用大约400页专门讲这个内容）。

嵌在Java程序中的HTML。  
看上去还不错，是不是？

重点：

- 每个Web应用都有一个部署描述文件
- 一个部署描述文件可以声明多个servlet。
- <servlet-name> 和 <servlet> 元素与相应的 <servlet-mapping> 元素绑定。
- <servlet-class> 是 Java 类。
- <url-pattern> 是客户所用的域名。

④ 在已有的Tomcat目录下建立这个目录树……

⑤ 从project1目录编译servlet……

```
{ %javac -classpath /your path/tomcat/common/lib/
  Servlet-api.jar -d classes src/Ch1Servlet.java
  (这整个是一个命令)
```

(Ch1Servlet.class文件最后会放在project1/classes目录中)。

⑥ 把Ch1Servlet.class文件复制到WEB-INF/classes，再把web.xml文件复制到WEB-INF。

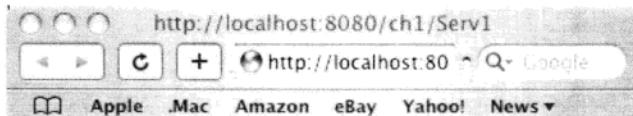
⑦ 从Tomcat目录，启动Tomcat……

%bin/startup.sh

⑧ 打开浏览器，键入

http://localhost:8080/ch1/Serv1

应该显示如下：

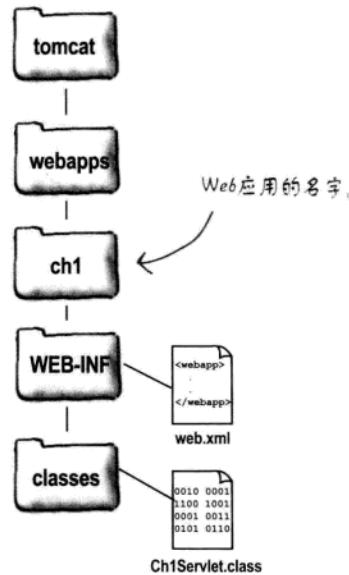


Tue April 10 16:20:01 MST 2004

← 你的日期可能与此不同.....

⑨ 现在，每次更新servlet类或部署描述文件都要关闭Tomcat：

%bin/shutdown.sh





```
out.println("<html> " +  
           "<body>" +  
           "<h1>Skyler's Login Page</h1>" +  
           "<br>" + today +  
           "</body>" +  
           "</html>");
```

在servlet中创建一个动态Web页面就是如此。必须把整个HTML打印到一个输出流（实际上是要打印的HTTP响应的一部分）。

实际上，在servlet的out.println()里格式化HTML确实不太好。

这是servlet最糟糕的方面之一（不，不是“之一”，这就是最糟糕的）。在println()中填入一些适当格式化的HTML标记，以便插入变量和方法调用，这种做法相当原始。如果HTML再复杂一点，不难想像该有多麻烦。

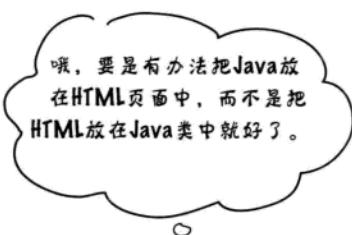
*there are no  
Dumb Questions*

**问：** 没那么糟吧……为什么不从Web页面编辑器（比如Dreamweaver）复制整个HTML页面，然后把它粘贴到println()中呢？这样我就不用再读这些代码了。

**答：** 很显然，你没有这样做过。听上去好像不错。确实，可以用一个很棒的Web页面编辑器来创建我的网页（甚至用一个简单的文本文件也比通过Java代码来生成要容易），然后把它复制粘贴到println()中，等着瞧吧！

可能会有1 378个编译错误在等着你。

要记住，String直接量里没有回车（真正的回车）。而且，既然提到了String，如果HTML中本来就有双引号该怎么办？



### 她还不认识JSP

```
<html>
<body>
<h1>Skyler's Login Page</h1>
<br>
<%= new java.util.Date() %> ←
</body>
</html>
```

skylerlogin.jsp

哇呜！看上去这是Java代码，居然放在HTML中  
间！！

JSP页面就像是一个HTML页面，只不过可以把Java和与Java有关的东西放在这个页面中。所以，这确实就像是把一个变量插入到HTML中。

# HTML中引入Java，这就是JSP

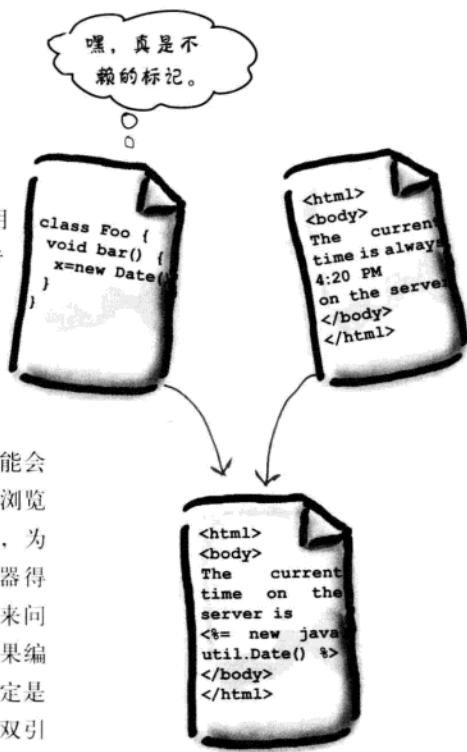
把Java放到HTML中，这样可以解决两个问题：

## 1 并不是所有HTML网页设计人员都懂Java。

应用开发人员懂Java。Web网页设计人员懂HTML。利用JSP，Java开发人员可以处理Java，HTML开发人员则可以专心处理Web页面。

## 2 把HTML以某种格式放入一个String直接量的做法确实很糟糕。

就算是稍有点复杂的HTML，放到println()的参数里时也可能会导致编译错误。要让HTML正确地格式化，不仅能在客户的浏览器上正常显示，而且还要满足有关String直接量的Java规则，为此需要做大量工作。例如，不能有回车，但是从网页编辑器得来的大多数HTML都会在源代码中包含回车。引号也会带来问题，例如，很多HMTL标记都使用引号把属性值引起来。如果编译器看到一个双引号，知道会怎么样吗？它会这样想“这肯定是String直接量的头或尾。”当然，你也可以回过头去把所有双引号都替换成相应的转义字符……但是这很容易出错。



**问：** 等等……这里还有错误！上面的第一个好处中声称“不是所有网页设计人员都懂Java……”，但是HTML页面设计人员还是必须在JSP页面里写Java代码！JSP确实让Java程序员免于编写HTML，但是对HTML设计人员并没有多大帮助。可能在JSP里写HTML比在println()中要容易一些，但是HTML开发人员还是必须懂Java才行呀。

**答：** 看上去是这样，对不对？但是，根据新的JSP规范，而且遵循最佳实践，页面设计人员应当尽可能少地在JSP中放入Java代码（甚至根本不放）。他们确实要学点东西，但是要学的只是如何放入一些调用具体Java方法的标记，而不是把具体的Java代码嵌入到页面本身。他们要学的是JSP语法，而不是Java语言。



## 要点

- HTTP代表超文本传输协议（HyperText Transfer Protocol），这是Web上使用的网络协议。HTTP运行在TCP/IP之上。
- HTTP使用一种请求/响应模型，客户做出一个HTTP请求，Web服务器返回一个HTTP响应，再由浏览器（根据响应的内容类型）确定如何进行处理。
- 如果来自服务器的响应是一个HTML页面，就会把HTML增加到HTTP响应中。
- HTTP请求包括请求URL（客户想要访问的资源）、HTTP方法（GET、POST等），以及（可选）表单参数数据（也称为“查询串”）。
- HTTP响应包括一个状态码、内容类型（也称为MIME类型），以及响应的实际内容（HTML、图像等）。
- GET请求会把表单数据追加到URL的最后。
- POST请求将表单数据包括在请求的体中。
- MIME类型告诉浏览器要接收哪一类数据，以便浏览器知道如何加以处理（呈现HTML、显示图片、播放音乐等）。
- URL代表统一资源定位符（Uniform Resource Locator）。Web上的每个资源都有自己的唯一地址，都采用这种URL格式。首先是一个协议，然后是服务器名以及一个可选的端口号，再后面通常是一个特定的路径和资源名。如果URL对应一个GET请求，那么它还可能包含一个可选的查询串。
- Web服务器擅长提供静态HTML页面，但是如果需要页面中有动态创建的数据（如当前时间），就需要某种辅助应用与服务器协作。如果不从Java术语来说，这些辅助应用（大多用Perl编写）通常称为CGI，这代表公共网关接口（Common Gateway Interface）。
- 把HTML放在println()语句中的做法很糟糕，也很容易出错，不过JSP可以解决这个问题，它允许把Java放在HTML页面中，而不是把HTML放在Java代码中。



# Web应用体系结构



Servlet也需要帮助。请求到来时，必须有人实例化servlet，或者至少要建一个新的线程处理这个请求，必须有人调用servlet的doPost()或doGet()方法。另外，对了，那些方法还有一些重要的参数——HTTP请求和HTTP响应对象，所以必须有人把请求和响应交给servlet。还得有人管理servlet的生与死以及servlet的资源。这个“人”就是Web容器。在一章中，我们会介绍Web应用在容器中怎样运行，我们还会第一次用模型—视图—控制器（Model View Controller, MVC）设计模式分析Web应用的结构。

# OBJECTIVES

## 高层Web应用体系统结构

- 1.1 对于每一种HTTP方法（如GET、POST、HEAD等），描述该方法的用途，以及该HTTP方法协议的技术特性，并列出客户（通常是一个Web浏览器）会因为哪些原因使用这种方法，明确对应这种HTTP方法的相应HttpServlet方法。
  
- 1.4 描述Servlet生命周期的作用和事件序列：(1) servlet类加载；(2) servlet实例化；(3) 调用init方法；(4) 调用service方法；(5) 调用destroy方法。
  
- 2.1 构建Web应用的文件和目录结构，可能包含：(a) 静态内容；(b) JSP页面；(c) servlet类；(d) 部署描述文件；(e) 标记库；(f) JAR文件；(g) Java类文件，并描述如何保护资源文件避免HTTP访问。
  
- 2.2 描述以下各个部署描述文件元素的作用和语义：servlet实例、servlet名、servlet类、servlet初始化参数，以及URL与命名servlet的映射。

## 内容说明：

这一部分的所有要求会在后面几章中再充分的讲解，所以可以把这一章看作是个热身，为以后做准备。换句话说，如读完这一章时还不了解（或者没记住）纲要求的这些内容，你也不用着急。

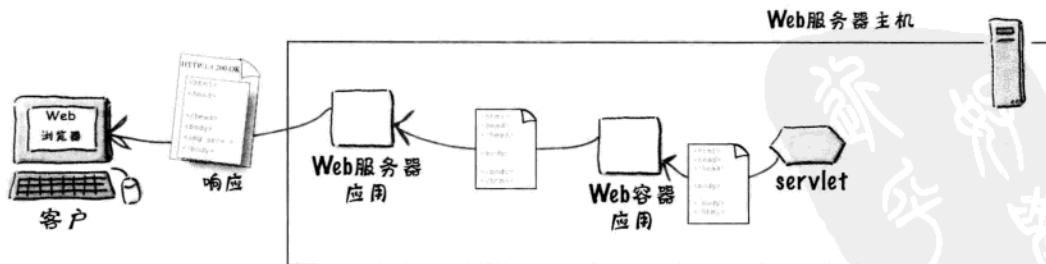
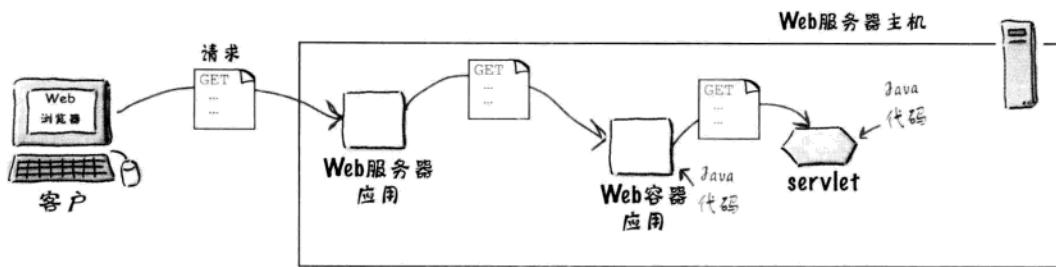
这一章没有针对这些要求给出模拟题，等到后面更具体地介绍有关内容之后，才会让你做些模拟题练习手。趁着现在还不用做题，好好享用这个简短的背景介绍吧，挺有意思的！

不过……在继续学习之前，确实需要知道这些内容。如果你已经有了一定的servlet经验，可以大致地翻翻这一章，只看看图片做做练习，然后直接看第3章。

# 什么是容器？

Servlet没有main()方法。它们受控于另一个Java应用，这个Java应用称为容器。

Tomcat就是这样一个容器。如果Web服务器应用（如Apache）得到一个指向某servlet的请求（而不是其他请求，如请求一个普通的静态HTML页面），此时服务器不是把这个请求交给servlet本身，而是交给部署该servlet的容器。要由容器向servlet提供HTTP请求和响应，而且要由容器调用servlet的方法，如doPost()或doGet()。



## 如果有Java，但是没有Servlet或容器，会怎么样？

如果你要写一个Java程序来处理到达Web服务器应用（像Apache）的动态请求，但是没有像Tomcat这样的容器会怎么样？换句话说，假设没有servlet之类的东西，你能拿到的只是核心J2SE库而已（当然，假设你能适当地配置Web服务器应用，让它能调用你的Java应用）。如果你对容器能做什么还不太清楚，也没有关系。只要想像一下，你需要对Web应用的服务器端支持，而你现有的只是普通的Java。



真正的勇士是不使用容器的。他能赤手空拳仅用J2SE写出一切。

在没有容器的前提下，列出你必须在J2SE应用中实现的功能：

\* 创建与服务器的socket连接，并为这个socket创建一个监听者。

---

---

---

---

---

JSP文件，对于，还有内存管理……  
可能的复杂逻辑；创建一个线程管理器，实现安全，对日志之类的过滤，

# 容器能提供什么？

我们知道，要由容器来管理和运行servlet，但是为什么要这样呢？由此会带来一些额外的开销，这样值得吗？

**通信支持** 利用容器提供的方法，你能轻松地让servlet与Web服务器对话。无需自己建立ServerSocket、监听端口、创建流等。容器知道自己与Web服务器之间的协议，所以你的servlet不必担心Web服务器（如Apache）和你自己的Web代码之间的API。你要考虑的只是如何在servlet中实现业务逻辑（如接受在线商店的一个订单）。

多亏有了容器，你可以更专注于自己的业务逻辑，而不用考虑为线程管理、安全性和网络通信编写代码。

你能把精力都用来建立一个超级在线商店，而把底层的服务（比如安全和JSP处理等）交由容器负责。

**生命周期管理** 容器控制着servlet的生与死。它会负责加载类、实例化和初始化servlet、调用servlet方法，并使servlet实例能够被垃圾回收。有了容器的控制，你就不用太多地考虑资源管理了。

**多线程支持** 容器会自动地为它接收的每个servlet请求创建一个新的Java线程。针对客户的请求，如果servlet已经运行完相应的HTTP服务方法，这个线程就会结束（也就是会死掉）。这并不是说不用考虑线程安全性，还是会遇到同步问题的。不过，由服务器创建和管理多个线程来处理多个请求，这样确实能让你少做很多工作。

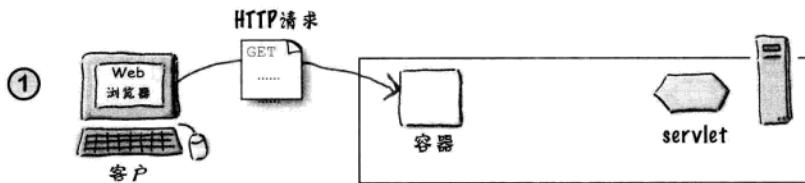
**声明方式实现安全** 利用容器，可以使用XML部署描述文件来配置（和修改）安全性，而不必将其硬编码写到servlet（或其他）类代码中。想像一下，不用去改你的Java源文件，也不用重新编译，你就能管理和修改安全性配置，这是多么吸引人呀！

**JSP支持** 你已经知道了JSP有多酷。那你知道是谁负责把JSP代码翻译成真正的Java吗？当然，正是容器。

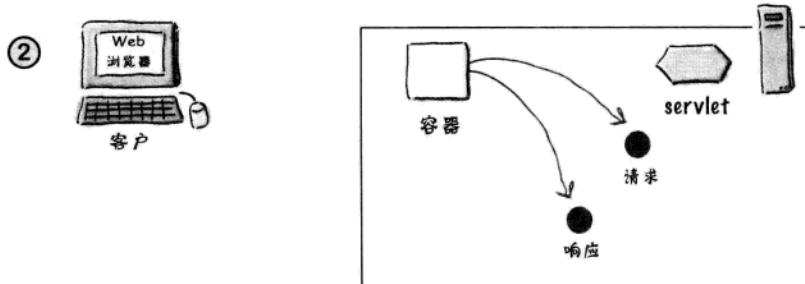


# 容器如何处理请求

后面会更详细地分析这个内容，不过这里先做一个简要的介绍。

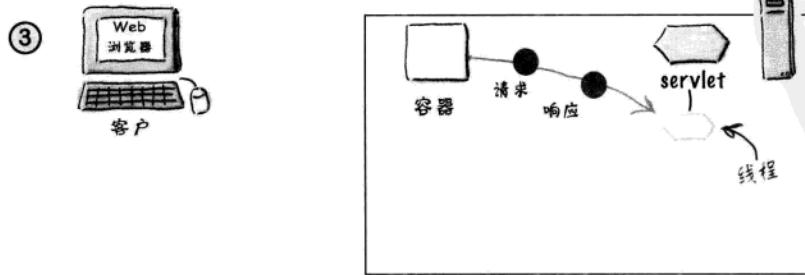


用户点击一个链接，其URL指向一个servlet而不是静态页面。



容器“看出来”这个请求要的是一个servlet，所以容器创建两个对象：

- (1) `HttpServletResponse`;
- (2) `HttpServletRequest`.



容器根据请求中的URL找到正确的servlet，为这个请求创建或分配一个线程，并把请求和响应对象传递给这个servlet线程。

④



容器调用servlet的service()方法。根据请求的不同类型，service()方法会调用doGet()或doPost()方法。

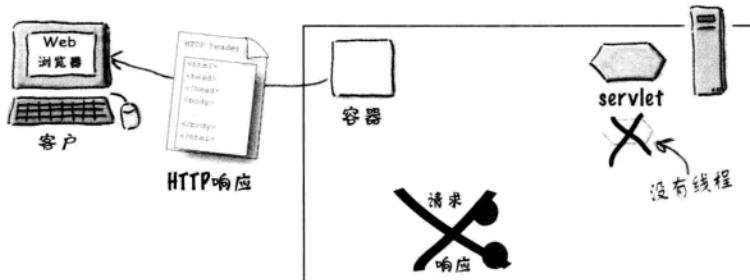
对于这个示例，假设请求是一个HTTP GET请求。

⑤



doGet()方法生成动态页面，并把这个页面“填入”响应对象。要记住，容器还有响应对象的一个引用！

⑥



线程结束，容器把响应对象转换为一个HTTP响应，把它发回给客户，然后删除请求和响应对象。

# 代码里有什么 (servlet何以成为一个servlet)

在实际中，99.9%的servlet都会覆盖  
doGet()或doPost()方法。

注意…… 没有main()方法。  
servlet的生命周期方法 (如  
doGet()) 由容器调用。

```

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

public class Ch2Servlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException {
        PrintWriter out = response.getWriter();
        java.util.Date today = new java.util.Date();
        out.println("<html> " +
                   "<body>" +
                   "<h1 style='text-align:center>" +
                   "HF's Chapter2 Servlet</h1>" +
                   "<br>" + today +
                   "</body>" +
                   "</html>");
    }
}

```

99.9999%的servlet都  
是HttpServlet。

servlet从这里拿到容器  
创建的请求和响应对象  
的引用。

在servlet从容器得到的响应对  
象中，可以拿到一个PrintWriter。  
使用这个PrintWriter能够将  
HTML文本输出到响应对象（除  
了PrintWriter以外，还可以输出  
其他内容，比如输出图片而不是  
HTML文本）。

*there are no*  
Dumb Questions

**问：** 我记得见过doGet()和doPost()，但是在前一页上你给出了一个service()方法，对不对？这个service()方法从哪来的？

**答：** 这个方法是servlet从HttpServlet继承来的，HttpServlet则是从GenericServlet继承来的，而GenericServlet自己又是从……这个类层次结构会在“作为Servlet”一章中说清楚，所以你要再耐心一点。

**问：** 你没有解释容器怎么找到正确的servlet……比如说，URL怎样与一个servlet关联？用户必须完整地键入servlet的绝对路径和类文件名吗？

**答：** 不必。这是一个很好的问题。不过，这个内容（servlet映射和URL模式）相当复杂，要讲清楚需要花些功夫，所以后面几页只做简单的了解，在本书后面（关于“部署”的一章）还会更详细地介绍。

# 你想知道容器怎样找到Servlet……

作为客户请求的一部分，URL会以某种办法映射到服务器上的一个特定servlet。URL与servlet的这种映射可以采用多种不同的方法处理，不过，这是Web应用开发人员遇到的最基本的问题之一。用户请求必须映射到一个特定的servlet，要理解而且（通常）要配置这种映射。你怎么考虑？



## 开动脑筋

### 容器怎样把servlet映射到URL？

用户在浏览器里做某个动作，比如点击一个链接，按下“Submit”（提交）按钮，输入一个URL等，可能会向你构建的一个特定servlet（或其他Web应用资源，如JSP）发送一个请求。此时会发生什么？

对于以下各种方法，分别考虑它们的优缺点。

- ① 把映射硬编码写到HTML页面中。换句话说，客户使用servlet的绝对路径和文件（类）名。

优点：

缺点：

- ② 使用容器开发商提供的工具来完成映射。

优点：

缺点：

- ③ 使用某种属性表来存储映射。

优点：

缺点：

# 一个servlet可以有3个名字

显然，servlet有一个文件路径名，比如classes/registration/SignUpServlet.class（具体类文件的路径）。开发servlet类的人要选择类名（以及包名，其中定义了部分目录结构），服务器上的位置则决定了完整的路径名。不过，部署servlet的任何人都可以给它起一个特殊的部署名。部署名只是一个秘密的内部名，不必非得与类名或文件名相同。它可以是servlet类名（registration.SignUpServlet），也可以是类文件的相对路径（classes/registration/SignUpServlet.class），不过也可以是毫不相干的一个名字（比如EnrollServlet）。

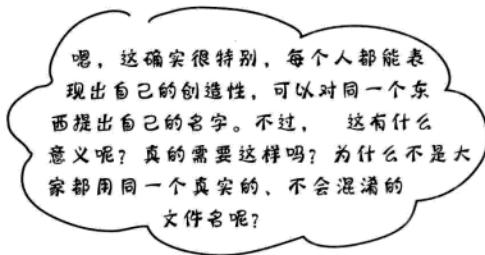
最后一点，servlet还有一个公共的URL名，这是客户所知道的名字。换句话说，这个名字要写在HTML中，这样当用户点击一个指向该servlet的链接时，就可以把这个公共URL名放在HTTP请求中发送给服务器。



客户看到HTML中对应一个servlet的URL，但是并不真正知道这个servlet名如何映射到服务器上的目录和文件。公共URL名只是一个虚构的名字，完全是为客户提供 的。

部署人员可以造一个名字，这个名字只有他自己以及实际操作环境中的其他人知道。同样地，这也是一个虚构的名字，只用于部署servlet。这个内部名不必与客户使用的公共URL名一致，也不必与servlet类的实际文件和路径名一样。

开发人员的servlet类有一个完全限定名，其中包括类名和包名。servlet类文件有一个实际的路径和文件名，这取决于服务器上目录结构所在的位置。



建立servlet名的映射，这有助于改善应用的灵活性和安全性。

想想吧。

如果你把真实的路径和文件名硬编码写到了所有使用这个servlet的JSP和其他HTML页面中，没问题。但是假设你现在想要重新组织你的应用，可能要把某些东西移到不同的目录结构下，会怎么样呢？你真的想让每一个使用servlet的人都知道（而且一直采用）同样的目录结构吗？

通过映射servlet名，而不是把真实的文件和路径名写入代码，这样就能提供很大的灵活性，使你能轻松地移动文件，而不用担心遭遇维护噩梦，否则，就需要跟踪客户代码中哪里引用了servlet文件原先的位置，并相应地进行修改，这个工作量将非常巨大。

再考虑一下安全性。你真的希望客户对你的服务器上的目录结构了如指掌吗？你希望他们绕过你的应用做事情吗？比如直接访问servlet，而不经过适当的页面或表单？因为，如果最终用户可以看到真实的路径，就可以在浏览器中键入这个路径直接访问。

# 使用部署描述文件将URL映射到servlet

将servlet部署到Web容器时，会创建一个相当简单的XML文档，这称为部署描述文件（DD），部署描述文件会告诉容器如何运行你的servlet和JSP。尽管使用部署描述文件不只是为了映射servlet名，最起码要知道，可以使用两个XML元素把URL映射到servlet，其中一个将客户知道的公共URL名映射到你自己的内部名，另一个元素把你自己的内部名映射到一个完全限定类名。

用于URL映射的两个部署描述文件元素如下：

## ① <servlet>

内部名映射到完全限定类名

## ② <servlet-mapping>

内部名映射到公共URL名



# 先等等！部署描述文件还有别的作用。

除了把URL映射到实际的servlet，还可以使用部署描述文件对Web应用的其他方面进行定制，包括安全角色、错误页面、标记库、初始配置信息等，如果这是一个完整的J2EE服务器，甚至可以声明你要访问特定的企业JavaBean。

先不要过分担心这些细节。现在的重点是，利用部署描述文件，你能采用一种声明方式修改应用，而无需修改源代码！

想想吧……这意味着，就算你不是Java程序员，也可以定制Java Web应用，而且不用中断你快乐的热带旅行，你不必中途赶回来修改代码。

there are no  
Dumb Questions

**问：** 我被搞糊涂了。看看部署描述文件，里面没有哪里指示servlet的实际路径名呀！它只是指出了类名。这仍然没有回答容器的问题：容器怎么使用这个类名来找到一个特定的servlet类文件呢？是不是别的什么地方还有一个映射、指出了某个类名会映射到哪个位置上的哪个文件，是这样吗？

**答：** 你看得很仔细。不错，我们只是在<servlet-class>元素里放了类名（完全限定类名，包括包名）。这是因为，容器会在一个特定的位置寻找你在部署描述文件的映射中指定的所有servlet。

实际上，容器使用了一组很复杂的规则来寻找匹配的servlet，它会根据客户请求提供的URL查找位于服务器某个位置上的一个具体Java类。不过这个内容到后面的一章（有关“部署”的一章）再做说明。就目前而言，关键是要记住确实可以完成这种映射。

**部署描述文件（DD）提供了一种“声明”机制来定制Web应用，而无需修改源代码！**



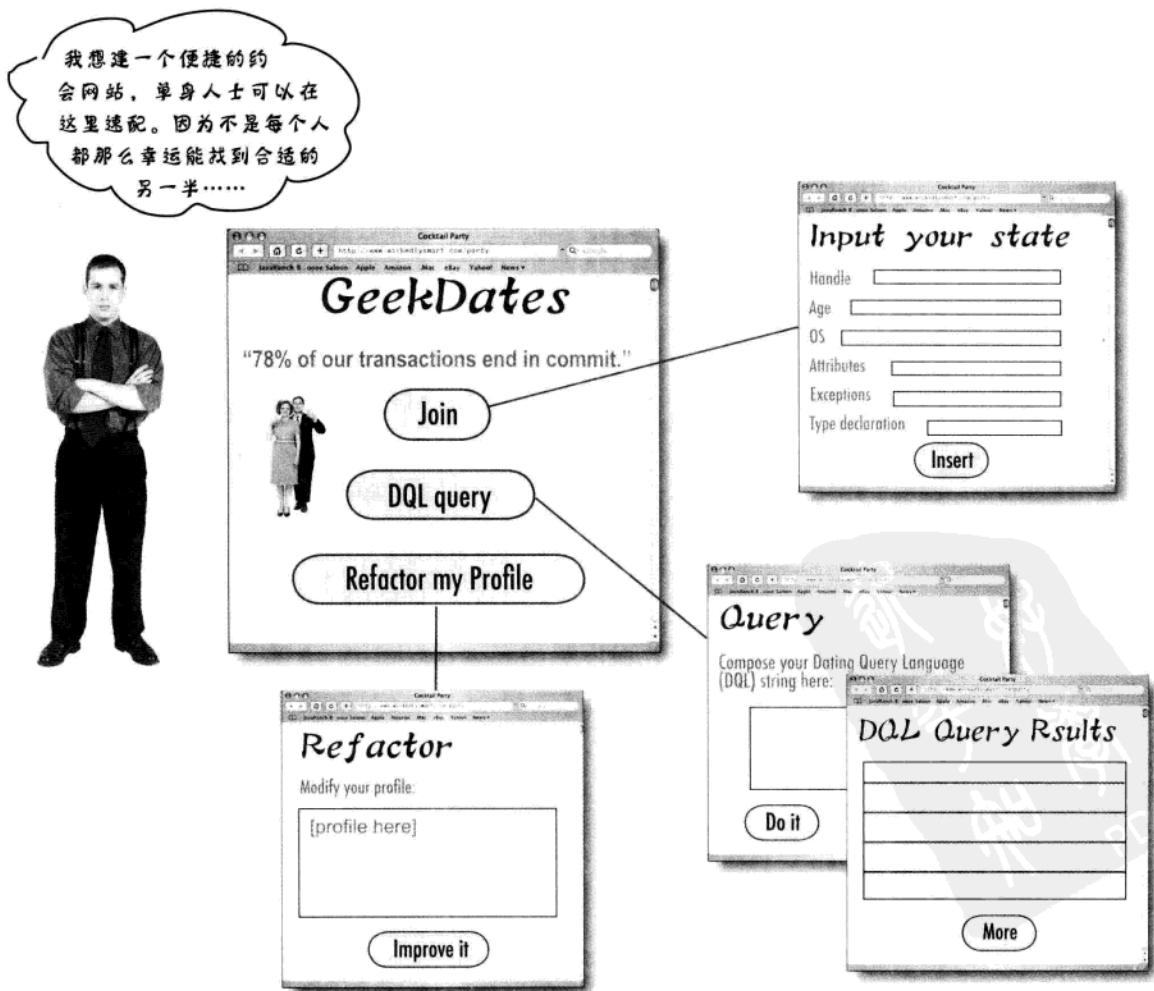
## DD的优点

- 尽量少改动已经测试过的源代码。
- 即使你手上并没有源代码，也可以对应用的功能进行调整。
- 不用重新编译和测试任何代码，也可以让你的应用适应不同的资源（如数据库）。
- 可以更容易地维护动态安全信息，如访问控制列表和安全角色。
- 非程序员也可以修改和部署你的Web应用，而你可以留出精力，做更有意思的事情，比如看看你的行装是否适合夏威夷之旅。

## 故事：Bob构建了一个速配网站

如今约会真是不容易。大家都很忙，谁有空约会呢？Bob想在.com浪潮里试试身手，认为创建一个专门为单身人士服务、帮助约会的网站很有意义，可以让他成就一番事业，摆脱现在无聊的工作。

问题是，Bob已经做了很久的软件管理人员，对当前的软件工程做法有点生疏了。不过，他知道一些时兴的术语，还懂一点Java，另外他还读过一些servlet的书，所以他很快做了一个设计，并开始编写代码……



# 他开始建一大堆servlet……针对每个页面有一个servlet

他本来考虑只建一个servlet，里面用大量if测试语句来区分不同情况，但后来决定分别创建不同的servlet，这样更能体现面向对象的精神，每个servlet只完成一个任务，比如查询页面、登录页面、搜索结果页面等。

每个servlet中包括修改或读取数据库所需的全部业务逻辑，还要把HTML输出到响应流返回给客户。

```
// import语句

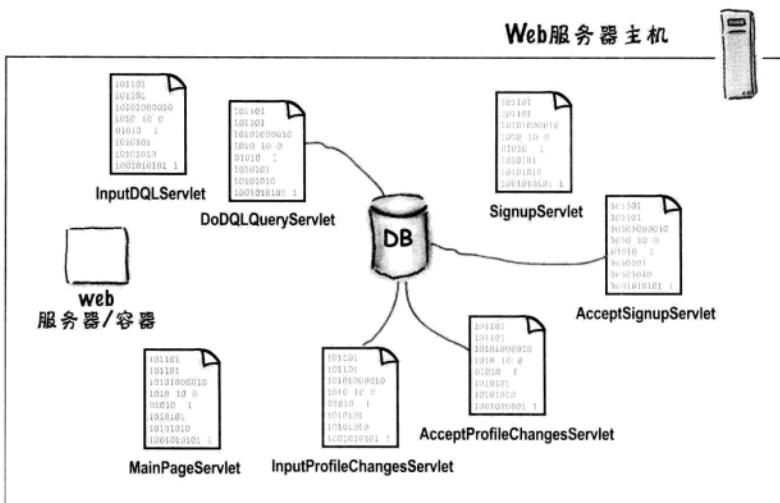
public class DatingServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
        // 业务逻辑放在这里,
        // 具体的业务逻辑取决于这个servlet要做什么
        // (写至数据库, 完成查询等)

        PrintWriter out = response.getWriter();

        // 构成动态HTML页面
        out.println("something really ugly goes here");
    }
}
```

这是一个绝妙的OO设计。我的所有servlet都只有一个任务。



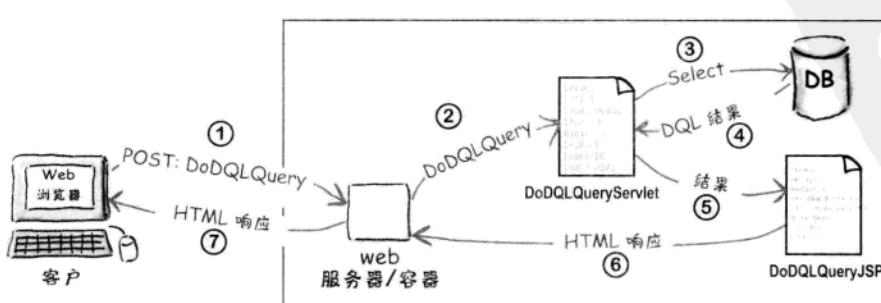
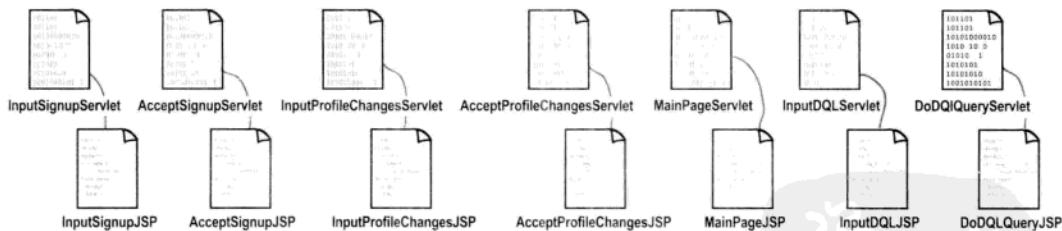
所有的业务逻辑和客户HTML页面响应都放在servlet代码中。

# 但是这很难看，所以他又增加了JSP

很快，那些输出响应的`println()`语句变得很丑陋。Bob简单了解了一下JSP，决定让每个servlet完成必要的业务逻辑（查询数据库、插入新记录或更新一个记录等），然后把请求转发给一个JSP，由JSP处理响应HTML。这样还能将业务逻辑与表示分离……因为他一直在读设计方面的书，所以很清楚关注点分离很有意义。

```
// import语句
public class DatingServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException {
        // 业务逻辑放在这里,
        // 具体的业务逻辑取决于这个servlet要做什么
        // (写至数据库, 完成查询等)

        // 把请求转发到一个特定的JSP页面,
        // 而不是把HTML打印到输出流
    }
}
```



这个JSP设计更酷了。现在servlet代码更加清晰……每个servlet只运行自己的业务逻辑，然后调用一个特定的JSP处理响应HTML，这就能把业务逻辑与表示相分离。



# 不过他的朋友问，“你不会没有使用MVC吧？”

Kim想知道，能不能从一个Swing GUI应用访问这个约会服务。Bob说，“不知道，我还没考虑这个问题呢。”然后Kim说，“没关系，这应该不成问题，因为我想你肯定用了MVC，所以应该能通过Swing GUI客户访问这个业务逻辑类。”

嗯。”Bob不置可否。

Kim又说，“你不会没有用MVC吧？”

Bob回答，“我确实把表示与业务逻辑分离了……”

Kim说，“这只是开始……让我看看……你的业务逻辑居然都放在servlet里！”

Bob突然想起他为什么采用现在这种管理方式。

不过，他决定改弦易辙，所以让Kim给他简要地讲讲MVC是什么。

**采用MVC，不仅要求业务逻辑与表示分离……实际上，业务逻辑甚至根本不知道有表示存在。**

MVC的关键是，业务逻辑要与表示分离，而且要在二者之间放上别的东西，这样业务逻辑本身就能作为一个可重用的Java类存在，它根本不用对视图有任何了解。

Bob做对了一部分，他做到了业务逻辑与表示分离，但是他的业务逻辑还是与视图有着紧密的联系。换句话说，他把业务逻辑混到了servlet里，这说明无法将业务逻辑用于其他视图（如Swing GUI，甚至无线应用）。他将业务逻辑放在servlet里，这是不对的，应该放在一个能重用的独立Java类中才对！

如果你想让一个  
Swing GUI应用访问这个约会  
服务，而且使用同样的业务  
逻辑，能做到吗？



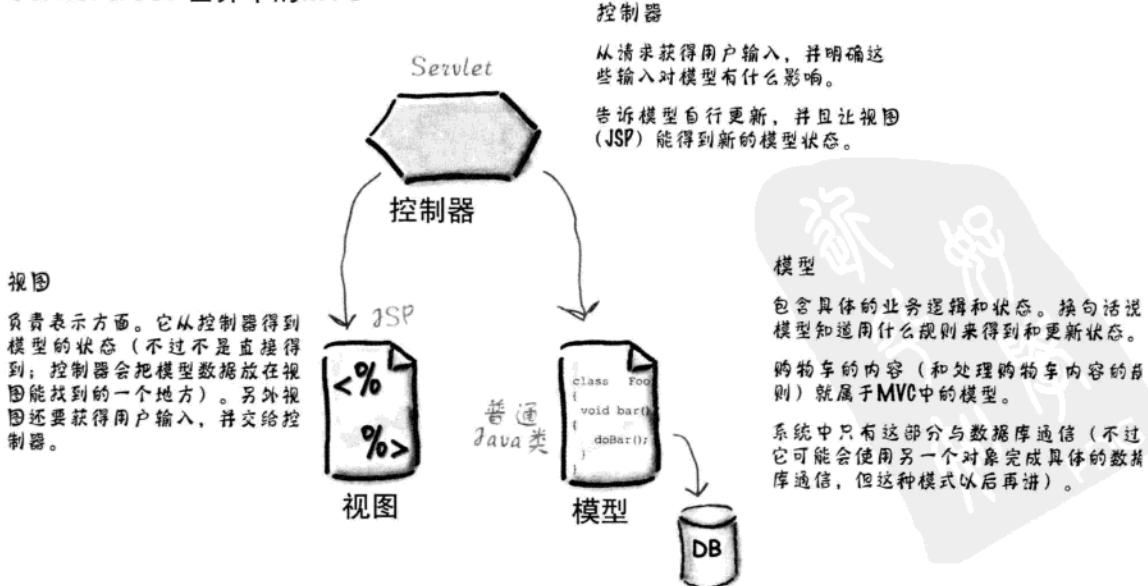
# 模型 - 视图 - 控制器(MVC)设计

## 模式可以解决这个问题

如果Bob理解了MVC设计模式，他就会知道，业务逻辑不应当放在servlet里。他还会认识到，如果把业务逻辑嵌在一个servlet中，等哪一天需要用另一种方法访问这个约会服务时，他就该犯愁了。比如说，有可能需要从一个Swing GUI应用访问这个服务。在本书的后面还会更多地谈到MVC（以及其他模式），不过你现在起码要对这种设计模式有一些认识，因为这一章最后构建的示例应用就使用了MVC。

如果你对MVC已经很熟悉，就会知道MVC并不是servlet和JSP所特有的，业务逻辑与表示要清晰地分离，不论在哪一种类型的应用中，这都是不变的真理。不过，对于Web应用来说，这一点则显得尤其重要，因为你不能保证别人只会从Web访问你的业务逻辑！相信你从事这个行业的时间应该不短了，肯定知道软件开发中有一点是铁定的，这就是：规范总在变。

### Servlet & JSP世界中的MVC

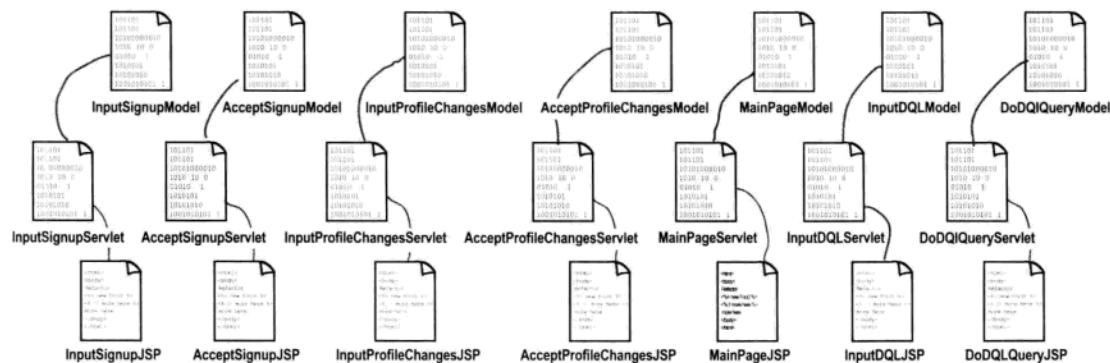


# 速配Web应用采用MVC模式

这么一来，Bob知道该怎么做了。先把业务逻辑与表示分离，再分别为每个Servlet创建一个常规的Java类……用来表示模型。

这样原先的servlet就变成了控制器，新的业务逻辑类则成为模型，JSP就是视图。

对于应用中的每个页面，现在分别有了一个servlet控制器、一个Java类模型和一个JSP视图。



还行，不过这真的是一个好设计吗？

## 不过他的朋友Kim又来看了看

Kim过来说，虽然这也算是一个MVC设计，但是太笨拙了。不错，业务逻辑已经放到模型中，servlet只是扮演着控制器的角色，在模型和视图之间协调，这样模型就能全然不知道视图的存在。这很好。但是再来看看那些小servlet。

他们到底做了些什么？既然业务逻辑已经安全地移到了模型中，servlet控制器实际上没做多少事情，而且他们做的工作对这个应用是通用的。不错，它只是更新模型，然后驱动视图来反映新的模型。

不过，最糟糕的是，每一个servlet中都重复地放着这些通用的应用逻辑！如果要做一点修改，那么每个servlet都需要修改。想想吧，这下肯定会遭遇繁琐的维护问题，想躲都躲不掉。

“嗯，这些重复的代码是让我有点烦，”Bob说，“但是我还能怎么样呢？你的意思不会是让我再把所有这些都放在一个servlet里吧？那又有什么好？”



拜托……你不是说真的  
吧，你真希望我把这些都放回到一  
个servlet里吗，那样不就违背面  
向对象精神了吗？

这个设计太蹩脚了！看看每  
个 servlet里的重复代码。如果  
采用这个设计，你就必须把同样  
的通用应用代码（比如安全性代  
码）在每一个servlet中都放一  
份。



# 有答案吗？

Bob该不该回过头来只用一个  
servlet控制器来避免重复的代码?  
这样还能算是一个好的OO设计吗? 因  
为servlet实际上在做不同的事情呀!  
Kim真的在行吗? 他说的对吗?



## 开动脑筋

我们希望你自己来考虑这些问题。

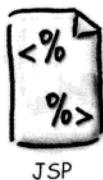
你怎么想? 你知道答案吗? 有答案吗? 你是同  
意Bob的意见, 就让servlet保持原样; 还是认  
为要把代码都放在一个servlet控制器里?  
如果只用一个控制器来处理所有事情,  
这个控制器又怎么知道该调用哪个模型  
和哪个视图呢?

这个问题的答案现在先不告诉你, 等  
到这本书的最后你才会找到现成的答  
案, 所以你先考虑一下, 别忘了就行, 就  
像你脑子里有一个后台线程一样……

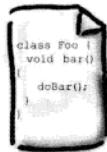




- ① 在servlet & JSP世界中使用MVC，这三个组件（JSP、Java类和servlet）分别扮演着不同的MVC角色。根据各个组件在MVC中的角色，请在“M”、“V”或“C”上画圈。对应每个组件只圈一个字母。



JSP

M  
V  
C非servlet  
Java类M  
V  
C

Servlet

M  
V  
C

- ② MVC这几个字母在MVC设计模式中表示什么？

M 代表 \_\_\_\_\_

V 代表 \_\_\_\_\_

C 代表 \_\_\_\_\_

### 要点

- 容器为Web应用提供了通信支持、生命周期管理、多线程支持、声明方式安全，以及JSP支持，这样你就能全神贯注地专心开发你自己的业务逻辑。
- 容器创建一个请求对象和一个响应回对象，servlet（或Web应用的其他部分）可以用这些对象得到有关请求的信息，并把信息发送给客户。
- 典型的servlet是一个扩展了HttpServletRequest的类，而且覆盖了一个或多个服务方法（doGet()、doPost()等），分别对应于浏览器调用的HTTP方法。
- 部署人员可以把servlet类映射到一个URL，这样客户可以使用这个URL来请求该servlet。部署名可以与实际的类文件名完全不同。



### 谁来负责?

填写下表，指出Web服务器、Web容器和servlet中谁最该负责所列的任务。有些情况下，可能同一个任务有且不只一个答案。如果想加分，还请加上简短的注释，对过程做一个描述。

任务	Web服务器	容器	Servlet
创建请求和响应对象			
调用service()方法			
开始一个新线程来处理请求			
把响应对象转换为一个HTTP响应			
了解HTTP			
把HTML增加到响应对象			
有响应对象的一个引用			
在部署描述文件中查找URL			
删除请求和响应对象			
协调生成动态内容			
管理生命周期			
名字与部署描述文件中的<servlet-class>元素匹配			



## 代码贴



冰箱上贴着一个实际运行的servlet以及它的部署描述文件（DD），但不完整。要构成完整的servlet及DD，请在右边的代码贴中找出合适的代码增加到左边不完整的代码清单中。这个servlet的URL以/Dice结尾。右边有些代码贴可能根本用不上！

### — Servlet —

```
public class _____
    extends HttpServlet {
```

```
public void doGet(
```

```
throws IOException {
```

```
String d1 = Integer.toString((int)((Math.random()*6)+1));
String d2 = Integer.toString((int)((Math.random()*6)+1));

out.println("<html> <body>" +
    "<h1 align=center>HF's Chap 2 Dice Roller</h1>" +
    "<p>" + d1 + " and " + d2 + " were rolled" +
    "</body> </html>");
}
```

### — DD —

```
<web-app ...>
```

记住，这不是一个完整的<web-app>开始标记，在这一章的最后会有一个完整的示例。不过即使没有完整的开始标记对这个练习也没有影响。

```
C2dice </servlet-name>
```

```
</web-app>
```

# 代码贴（续）……

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

&lt;/url-pattern&gt;

public void service(

C2dice

Ch2Dice

&lt;servlet-name&gt;

ServletRequest request,

PrintWriter out = response.getWriter();

HttpServletResponse response)

&lt;servlet-mapping&gt;

C2dice

ServletResponse response,

&lt;servlet-name&gt;

&lt;/servlet-class&gt;

/Dice

Ch2Dice

HttpServletRequest request,

&lt;servlet&gt;

/Dice

PrintWriter out = request.getWriter();

Ch2Dice

&lt;url-pattern&gt;

&lt;servlet-class&gt;

&lt;/servlet&gt;

&lt;/servlet-mapping&gt;



## 练习答案

任务	Web服务器	容器	Servlet
创建请求和响应回对象		在开始线程之前创建	
调用service()方法		然后service()方法 再调用doGet()或doPost()	
开始一个新线程来处理请求		开始一个servlet线程	
把响应回对象转换为一个HTTP响应		由响应回对象中的数据 生成HTTP响应流	
了解HTTP	用于与客户浏览器 对话		
把HTML增加到响应回对象			提供给客户的 动态内容。
有响应回对象的一个引用		容器把它交给servlet	用它打印响应。
在部署描述文件中查找URL		找到对应请求的适当 servlet	
删除请求和响应回对象		servlet一旦结束就 删除请求和响应回对象	
协调生成动态内容	知道如何转发到容器	知道要调用什么方法	
管理生命周期		调用服务方法（以及后 面将看到的其他方法）	
名字与部署描述文件中的 <servlet-class>元素匹配			任何公共类

## 练习答案（续）……

**Servlet**

```

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

public class Ch2Dice extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
        PrintWriter out = response.getWriter();
        String d1 = Integer.toString((int)((Math.random()*6)+1));
        String d2 = Integer.toString((int)((Math.random()*6)+1));
        out.println("<html> <body>" +
                   "<h1 align=center>HF's Chap 2 Dice Roller</h1>" +
                   "<p>" + d1 + " and " + d2 + " were rolled" +
                   "</body> </html>");
    }
}

```

**DD**

```

<web-app ...>
    <servlet>
        <servlet-name> C2dice </servlet-name>
        <servlet-class> Ch2Dice </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name> C2dice </servlet-name>
        <url-pattern> /Dice </url-pattern>
    </servlet-mapping>
</web-app>

```

## 一个“实际”部署描述文件(DD)

如果不明白下面这些到底是什么意思，先不要着急，后面在其他章节里还会讲到（而且考试中也会考）。在这里，我们只想让你看看一个实际的web.xml部署描述文件是什么样子。这一章的其他示例都省略了开始<web-app>标记中的许多内容（你会了解到我们为什么一般不在示例中包括完整的开始<web-app>标记）。

这本书里通常会这样显示：

```
<web-app ...> ← 这个开始<web-app>标记不完整。  
  <servlet>  
    <servlet-name>Ch3 Beer</servlet-name>  
    <servlet-class>com.example.web.BeerSelect</servlet-class>  
  </servlet>  
  
  <servlet-mapping>  
    <servlet-name>Ch3 Beer</servlet-name>  
    <url-pattern>/SelectBeer.do</url-pattern>  
  </servlet-mapping>  
  
</web-app>
```

你不用记住这个开始标记。使用  
一个与servlet2.4规范兼容的容器  
时（如Tomcat 5），只需把这个  
开始标记复制下来就行了。

而实际上这应当是：

```
<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">  
  
  <servlet>  
    <servlet-name>Ch3 Beer</servlet-name>  
    <servlet-class>com.example.web.BeerSelect</servlet-class>  
  </servlet>  
  
  <servlet-mapping>  
    <servlet-name>Ch3 Beer</servlet-name>  
    <url-pattern>/SelectBeer.do</url-pattern>  
  </servlet-mapping>  
  
</web-app>
```

# J2EE如何集成这一切

Java 2企业版（Java 2 Enterprise Edition, J2EE）是一种超级规范，它结合了其他一些规范，包括Servlets 2.4规范和JSP 2.0规范，这些是对应Web容器的。另外J2EE 1.4规范还包括Enterprise JavaBean（EJB）2.1规范，这对应EJB容器。换句话说，Web容器用于Web组件（Servlet和JSP），EJB容器用于业务组件。

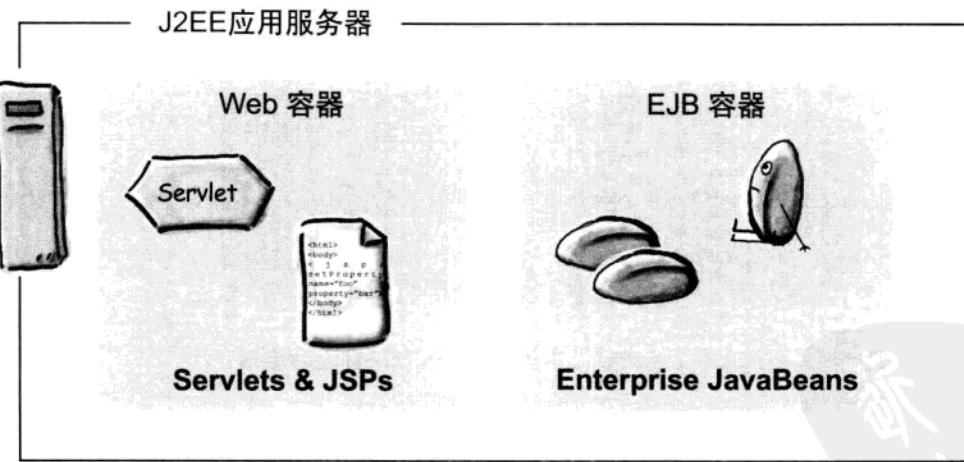
一个完全兼容的J2EE应用服务器必须有一个Web容器和一个EJB容器（以及其他一些部件，包括JNDI和JMS实现）。Tomcat只是一个Web容器！它只是与J2EE规范中有关Web容器的部分兼容。

Tomcat是一个Web容器，而不是完整的J2EE应用服务器，因为Tomcat没有EJB容器。

**J2EE应用服务器包括一个Web容器和一个EJB容器。**

**Tomcat是一个Web容器，而不是一个完整的J2EE应用服务器。**

**J2EE 1.4服务器包括Servlet 2.4规范、JSP 2.0规范，以及EJB 2.1规范。**

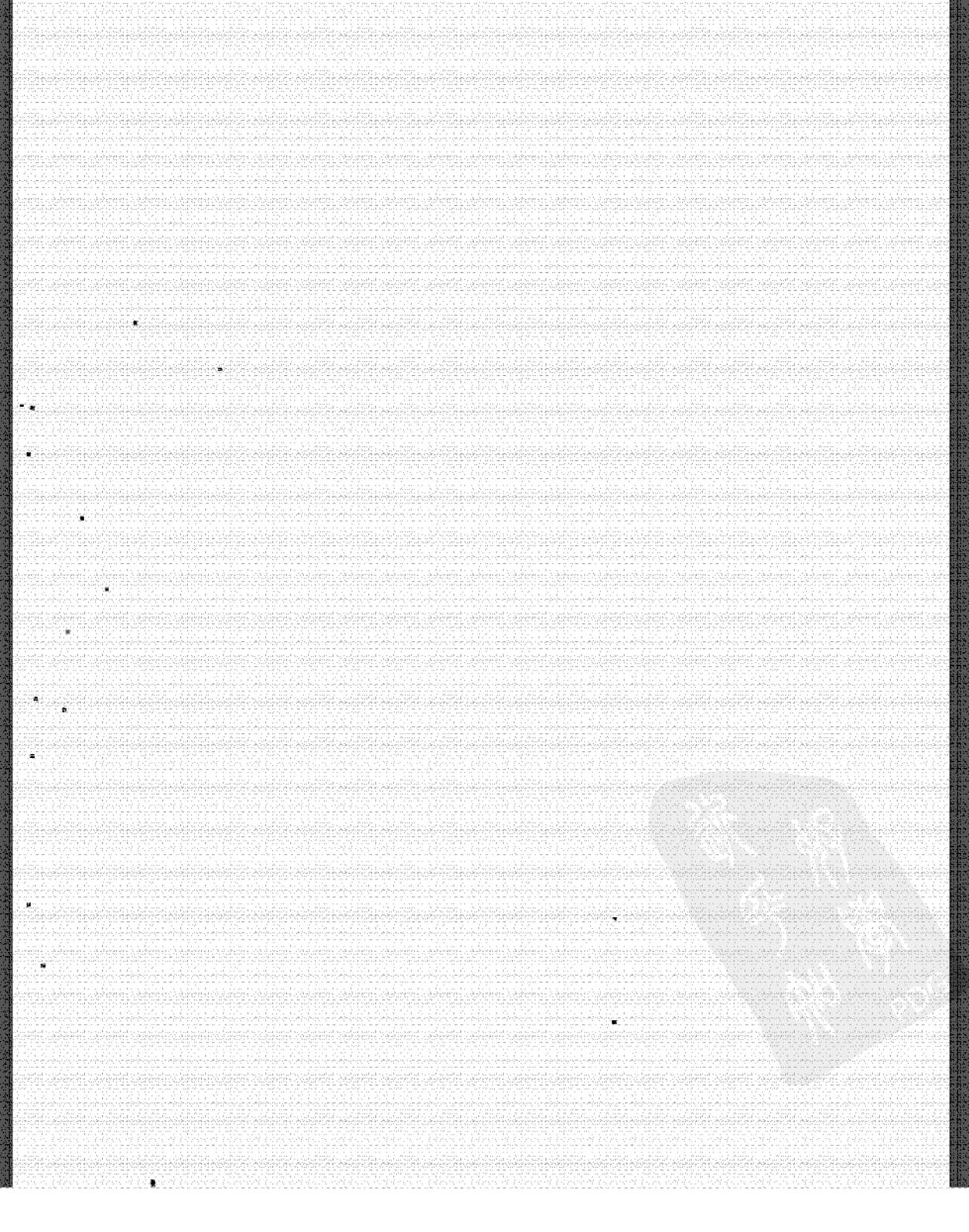


**问：** 那Tomcat是一个独立的Web容器吗……这是不是说也存在独立的EJB容器呢？

**答：** 早先，比如说2000年，确实可以找到完整的J2EE应用服务器和独立的Web容器，也能找到独立的EJB容器。不过，如今几乎所有EJB容器都作为完整J2EE服务器的一部分，但独立的web容器依然存在，如Tomcat和Resin。独立的Web容器通常配置为与一个HTTP

Web服务器（如Apache）协作，不过Tomcat容器本身就能作为一个基本的HTTP服务器。但是在HTTP服务器功能方面，Tomcat没有Apache那么健壮，所以最常见的非EJB Web应用通常会结合使用Apache和Tomcat，Apache作为HTTP Web服务器，Tomcat作为Web容器。

还有一些常用的J2EE服务器，包括BEA的WebLogic、开源的JBoss AS，以及IBM的WebSphere。



# MVC迷你教程



**创建和部署MVC Web应用。**该你动手写点东西了，要写一个HTML表单、一个servlet控制器、一个模型（普通的Java类）、一个XML部署描述文件，以及一个JSP视图。你要构建、部署和测试这个应用。不过第一步需要先建立一个开发环境，这个工程目录结构与你最后部署的应用是不同的。接下来，需要建立遵循servlet和JSP规范的部署环境，并满足Tomcat的需求。然后就要开始编写、编译、部署和运行这个应用了。必须承认，我们构建的是一个简单的应用。不过，只要使用了MVC，再小的应用也不算小，因为今天这个小应用没准明天就会火起来……

# OBJECTIVES

## Web应用部署

- 2.1 构建一个Web应用的文件和目录结构可能包含(a) 静态内容、(b) JSP页面、(c) servlet类、(d) 部署描述文件、(e) 标记库、(f) JAR文件和(g) Java类文件。说明如何保护资源文件避免通过HTTP访问。
- 2.2 描述以下各个部署描述文件元素的作用和语义：error-page、init-param、mime-mapping、servlet、servlet-class、servlet-mapping、servlet-name和welcome-file。
- 2.3 为以下各个部署描述文件元素构造正确的结构：error-page、init-param、mime-mapping、servlet、servlet-class、servlet-name和welcome-file。

## 内容说明：

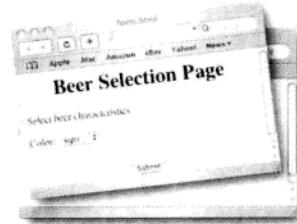
这一部分的所有要求都会在关于“部署”的一章全面介绍；这里只是做一个简单的了解。这一章是这本书中唯一一个从头到尾完整介绍的教程，所以如果你跳过这一章，等读到后面的一些章节，想测试其中的一些示例时就可能有麻烦（因为我们不会把每一个细节再完整地重复一遍）。和前两章一样，你不用费劲地去记这一章中的内容。只要照着做就可以了。

# 构建一个真正的（小）Web应用

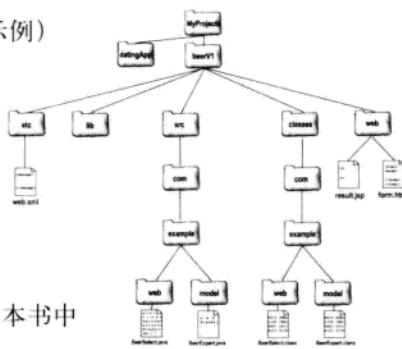
我们前面介绍过容器的角色，对部署描述文件做过一点讨论，而且简单地了解了模型 2 MVC体系结构。不过，你不能老是坐在那里看呀！临渊羡鱼，不如退而结网。现在该轮到你具体动手做点事情了。

## 我们的4大步骤：

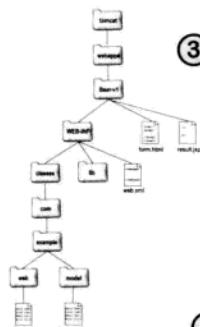
- ① 分析用户的视图（也就是浏览器要显示的东西），以及高层体系结构。



- ② 创建用于开发这个项目（以及本书中其他示例）的开发环境。



- ③ 创建用于部署这个项目（以及本书中其他示例）的部署环境。



- ④ 对Web应用的各个组件完成迭代式的开发和测试（没错，与其说这是一个步骤，不如说是一个策略）。

注意：我们建议采用迭代式的开发和测试，不过在这本书里不一定总会展示所有步骤。

## 啤酒顾问Web应用的用户视图

我们的Web应用是一个啤酒顾问（Beer Advisor）。用户访问这个应用，先问一个问题，然后会得到一条有用的啤酒建议。

form.html

Beer Selection Page

Color:

Submit

This screenshot shows a web browser window titled "form.html". The main content is titled "Beer Selection Page". It contains a single input field labeled "Color:" with a dropdown menu open, showing the option "light". Below the input field is a "Submit" button.

这个页面用HTML来写，而且会生成一个HTTP Post请求，并把用户的颜色选择作为一个参数发送。

form.html

Beer Recommendations JSP

try: Jack's Pale Ale  
try: Gout Stout

This screenshot shows a web browser window titled "form.html". The main content is titled "Beer Recommendations JSP". It displays two lines of text: "try: Jack's Pale Ale" and "try: Gout Stout".

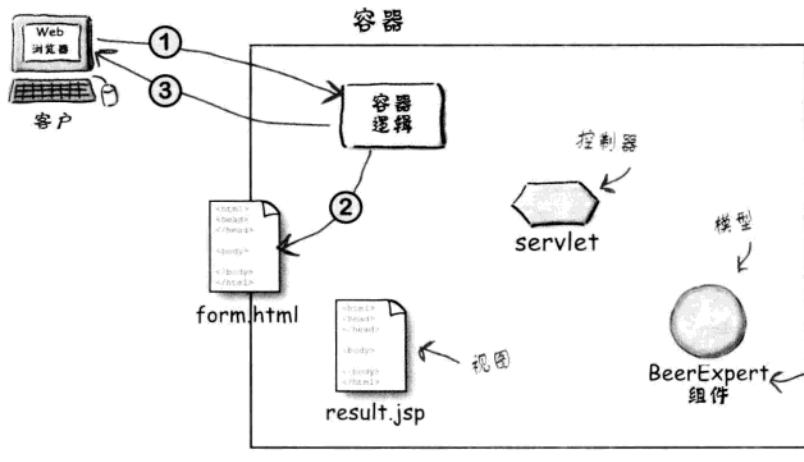
这个页面是一个JSP页面，它基于用户的选择给出建议。

**问：**为什么要写一个提供啤酒建议的Web应用？

**答：**根据广泛的市场调查，我们得出一个结论，90%的读者都很青睐啤酒。而对于另外的10%，只需把“啤酒”换成“咖啡”。

# 以下是体系结构……

尽管这是一个非常小的应用，但我们要使用一个简单的MVC体系结构来构建。这样来，等有一天它变成Web上最热门的网站时，我们就能从容地扩展这个应用了。



1. 客户请求得到form.html 页面。

2. 容器获得form.html页面。

3. 容器把这个页面返回给浏览器，用户再在浏览器上回答表单上的问题……

*只是一个POJO (普通的Java对象, Plain Old Java Object)。*

4. 浏览器把请求数据发送给容器。

5. 容器根据URL查找正确的servlet，并把请求传递给这个servlet。

6. servlet调用BeerExpert寻求帮助。

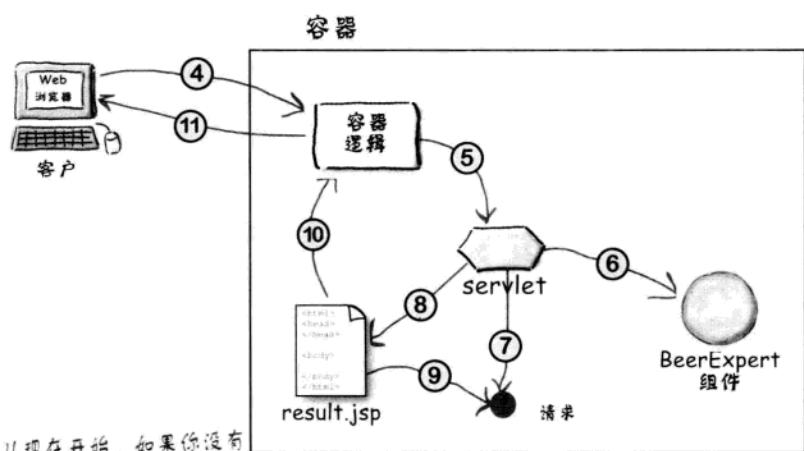
7. 这个“专家”类返回一个回答，servlet把这个回答增加到请求对象。

8. servlet把请求转发给JSP。

9. JSP从请求对象得到回答。

10. JSP为容器生成一个页面。

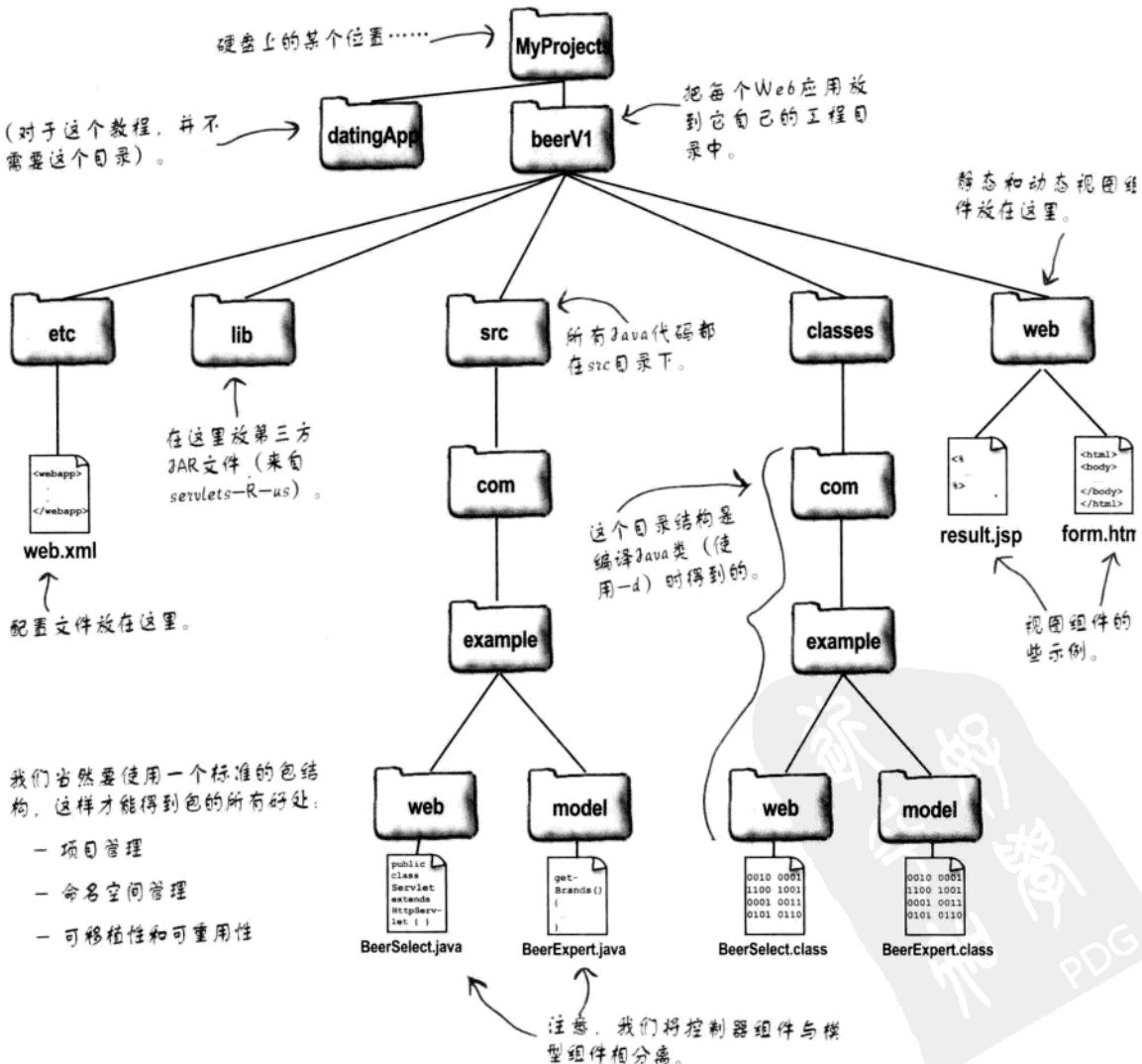
11. 容器把这个页面返回给心满意足的用户。



从现在开始，如果你没有看到Web服务器出现，请假设它已经存在。

# 创建你的开发环境

组织开发目录结构有很多办法，不过这里推荐的方法比较适用于小型和中型的项目。到部署Web应用时，要把这个目录结构适当地复制到特定容器所希望的位置上（在这个教程中，我们使用的容器是Tomcat 5）。



# 创建部署环境

要部署一个Web应用，这涉及到容器特定的规则，还涉及Servlet和JSP规范的需求（如果不在Tomcat上部署，就必须明确Web应用相对于容器的位置）。在这个例子中，不论你用的是什么容器，“Beer-v1”目录以下的所有东西都是一样的！

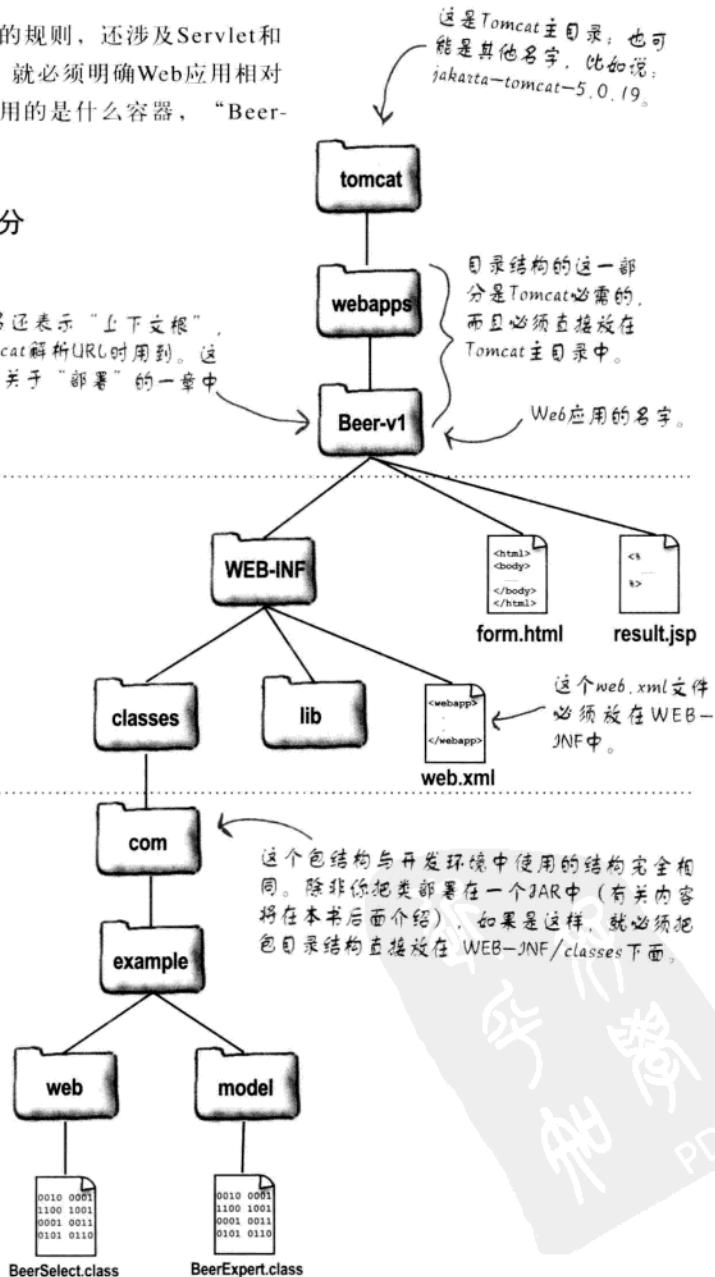
## 特定于Tomcat的部分

这个目录名还表示“上下文根”，这会在Tomcat解析URL时用到。这个概念将在关于“部署”的一章中详细讨论。

这个虚线以下的部分是Web应用，不论你使用什么容器，这一部分都是一样的。

## Servlet规范的部分

## 特定于应用的部分



## 构建应用的路线图

在这一章刚刚开始的时候，我们指出了开发Web应用的4大步骤。到目前为止，已经完成了前3步：

1. 分析Web应用的用户视图。
2. 分析体系结构。
3. 建立创建和部署应用的开发环境和部署环境。

现在该完成第4步了，下面来创建应用。

我们借用了一些流行的开发方法（部分来自极限编程、迭代式开发等），并对它们适当地加以调整来用于我们的特殊目的……

### 第4步又分为5个小步骤：

- (4a)** 构建和测试用户最初请求的HTML表单。
- (4b)** 构建控制器servlet的第一个版本，并用HTML表单测试这个控制器。  
这个版本通过HTML表单来调用，并打印出它接收到的参数。
- (4c)** 为专家/模型类构建一个测试类，然后构建并测试专家/模型类本身。
- (4d)** 把servlet升级到第2版。这个版本增加了一个功能，可以调用模型类来得到啤酒建议。
- (4e)** 构建JSP，把servlet升级到第3版本（再增加一个功能，可以把表示分派到JSP完成），然后再测试整个应用。

# 第一个表单页面的HTML

这个HTML很简单，它只包含标题文本，一个下拉列表（用户可以从  
中选择一种啤酒颜色），另外还有一个提交按钮。

```

<html><body>

<h1 align="center">Beer Selection Page</h1>

<form method="POST" <-- 为什么选择POST而不是GET?>
  action="SelectBeer.do" <-- HTML认为这就是要调用的servlet。在你的目
  Select beer characteristics<p>
  Color:
  <select name="color" size="1">
    <option value="light"> light </option>
    <option value="amber"> amber </option>
    <option value="brown"> brown </option>
    <option value="dark"> dark </option>
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form></body></html>

```

我们就是这样创建下拉菜单的，你  
可以有自己不同的选项。  
注意到 size="1" 了吗？

**问：**为什么表单提交到“SelectBeer.do”，根本没有这样一个servlet呀？在前面  
看到的目录结构中，我没有找到名为“SelectBeer.do”的文件。“.do”到底是不是扩  
展名？

**答：**SelectBeer.do是一个逻辑名，而不是一个实际的文件名。这只是我们希望客户  
使用的名字！实际上，客户根本不能直接访问servlet类文件，所以你不能直接使用类文  
件名，例如，在创建的HTML页面中，链接或动作不能包含servlet类文件的路径。

这里的技巧是，我们会使用XML部署描述文件（web.xml）把客户请求的资源  
（“SelectBeer.do”）映射到一个实际的servlet类文件，当指向“SelectBeer.do”的请求  
到达时，容器就会使用这个类文件。对目前来讲，可以把“.do”扩展名当作是逻辑名  
的一部分（而不是一个实际的文件类型）。在本书的后面，你会了解到在servlet映射中  
还可以用其他方法使用扩展名（实际扩展名或虚构/逻辑扩展名）。

# 部署和测试开始页面

要进行测试，需要把它部署到容器（Tomcat）目录结构中，启动Tomcat，在浏览器中打开页面。

## ① 在开发环境中创建HTML。

创建这个HTML文件，取名为form.html，然后保存在开发环境的/beerV1/web/目录下。

## ② 把这个文件复制到部署环境。

把form.html文件的一个副本放在tomcat/webapps/Beer-v1中（要记住，你的tomcat主目录可能是别的名字）。

## ③ 在开发环境中创建DD。

创建以下XML文档，取名为web.xml，把它保存在开发环境的/beerV1/etc/ 目录下。



form.html

你没有必要知道这是什么意  
思，只需要照着输入就行了。

```

<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">

  <servlet>
    <servlet-name>Ch3_Beer</servlet-name>
    <servlet-class>com.example.web.BeerSelect</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch3_Beer</servlet-name>
    <url-pattern>/SelectBeer.do</url-pattern>
  </servlet-mapping>
</web-app>
  
```

这是一个虚构的名字，只能  
在DD的其他部分使用。

↓  
servlet类文件的完全限定名。

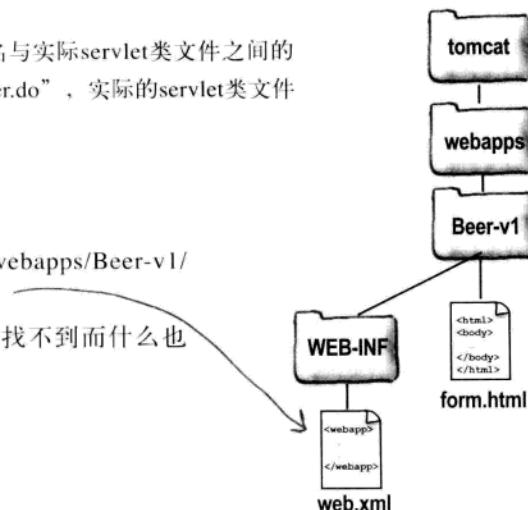
不要忘记最前面有 一个斜线。  
我们希望客户这样引用servlet。".do" 只是  
一个约定。

这个部署描述文件的主要任务是定义逻辑名与实际servlet类文件之间的映射，客户用于请求的逻辑名是“SelectBeer.do”，实际的servlet类文件是com.example.web.BeerSelect。

#### ④ 把这个文件复制到部署环境。

把web.xml文件的一个副本放在tomcat/webapps/Beer-v1/WEB-INF/。

必须把它放在这里，否则容器就会因为找不到而什么也不做，这会让你失望的。



#### ⑤ 启动Tomcat。

这本书将一直使用Tomcat，既作为Web服务器，又作为Web容器。在实际中，你可能会使用一个更健壮的Web服务器（如Apache），另外配置有一个Web容器（如Tomcat）。不过，对于这本书所要做的，将Tomcat作为Web服务器已经足够了。

要启动Tomcat，先用cd命令切换到tomcat主目录，再运行bin/startup.sh。

```

File Edit Window Help OpenSource
% cd tomcat
% bin/startup.sh
  
```

#### ⑥ 测试页面。

在浏览器中打开这个HTML页面，为此键入：

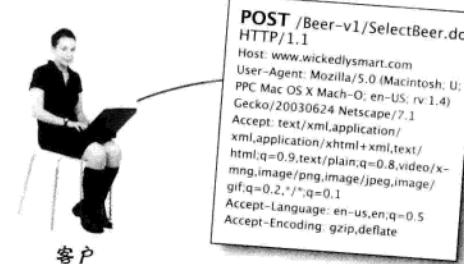
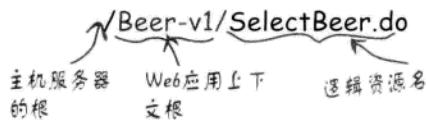
<http://localhost:8080/Beer-v1/form.html>

应该可以看到旁边显示的这个页面。



# 逻辑名映射到servlet类文件

- ① Diane填写了表单，然后点击submit（提交）。浏览器生成了以下请求URL：



- ② 容器搜索DD，找到<url-pattern>与/SelectBeer.do匹配的一个<servlet-mapping>，这里的斜线(/)表示Web应用的上下文根，SelectBeer.do就是资源的逻辑名。



- ③ 容器看到对应这个<url-pattern>的<servlet-name>是“Ch3 Beer”。但是这并不是实际servlet类文件的名字。“Ch3 Beer”是servlet名，而不是servlet类的名字！

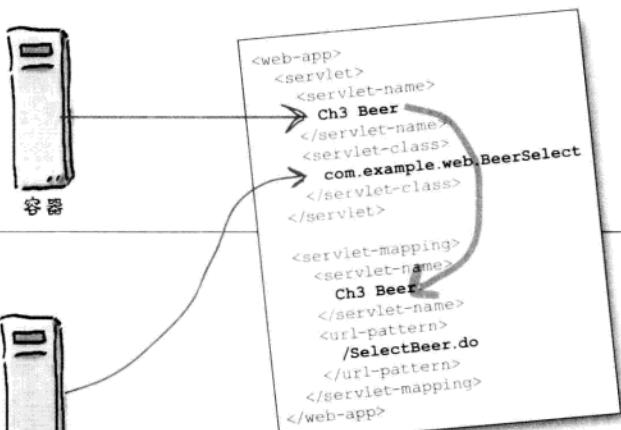
对容器来说，servlet只是在DD中<servlet>标记下命名的一个东西。servlet名只在DD中使用，以便DD的其他部分建立与该servlet的映射。

在这个HTML中，“/Beer-v1/”不是路径的一部分。在HTML中，它只是说：  
<form method="POST"  
action="/SelectBeer.do">

但浏览器为请求追加了“/Beer-v1/”，因为客户请求就来自这里。换句话说，HTML中的“SelectBeer.do”相对于其所在页面的URL。在这里，就是相对于Web应用的根：“/Beer-v1”。



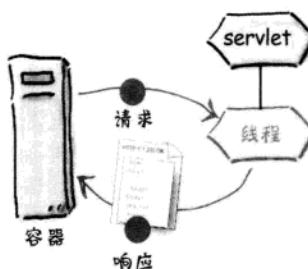
- ④ 容器查找< servlet-name>为“Ch3 Beer”的< servlet>标记。



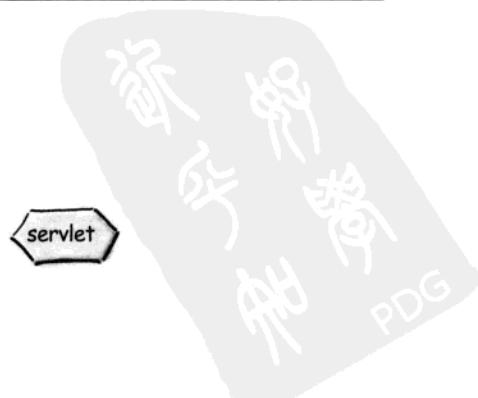
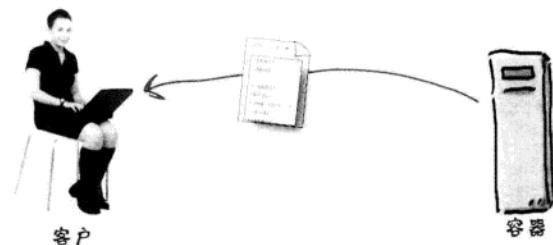
- ⑤ 根据< servlet>标记中的< servlet-class>, 容器可以知道由哪个servlet类负责处理这个请求。如果这个servlet还没有初始化, 就会加载类, 并初始化servlet。



- ⑥ 容器开始一个新线程来处理这个请求, 并把请求传递给这个线程(传递给servlet的service()方法)。



- ⑦ 容器把响应(当然是通过Web服务器)发回给客户。



# 控制器 servlet 的第1版

我们的计划是分阶段构建这个servlet，在这个过程中逐步测试各个通信链接。要记住，最后servlet要从请求接受一个参数，在模型上调用一个方法，把信息保存在JSP能找到的一个位置上，再把请求转发给JSP。但是对于第1版，只需要确保HTML页面能适当地调用servlet，而且servlet能正确地接收HTML参数就可以了。

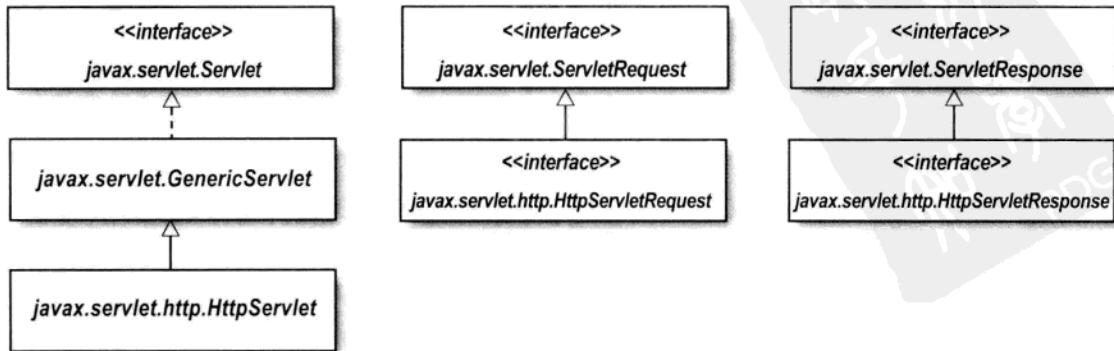
## Servlet代码

```
package com.example.web; // 确保与前面创建的开发结构和部署  
                        // 结构匹配。  
  
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.io.*;  
  
public class BeerSelect extends HttpServlet {  
  
    public void doPost(HttpServletRequest request,  
                       HttpServletResponse response)  
        throws IOException, ServletException {  
  
        response.setContentType("text/html"); // 这个方法来自  
        // HttpServletResponse接口。  
        PrintWriter out = response.getWriter();  
  
        out.println("Beer Selection Advice<br>");  
        String c = request.getParameter("color"); // 这个方法来自  
        // HttpServletRequest接口。注意这个参数与HTM  
        // <select>标记中“name”属性  
        // 的值匹配。  
        out.println("<br>Got beer color " + c);  
    }  
}
```

我们使用doPost来处理HTTP请求，因为HTML表单指出：method=POST

这里我们没有返回建议，只是把测试信息显示出来。

## 关键的API



# 编译、部署和测试控制器servlet

好了，我们已经构建、部署并测试了HTML，而且也构建和部署了DD（这里确实是把web.xml放在部署环境中，但是从理论上讲，除非重启Tomcat，否则不会部署DD）。下面该编译第1版的servlet，完成部署，并通过HTML表单进行测试。现在我们要重启Tomcat，确保它能“看到” web.xml和servlet类。

## 编译servlet

用-d标志编译servlet，把类放在开发环境中。

针对你的系统上的目录路径做适当调整！“tomcat/”后面的部分都是一样的。

```
File Edit Window Help UpdateBrain
% cd MyProjects/beerV1
% javac -classpath /Users/bert/Applications2/tomcat/common/lib/
servlet-api.jar:classes:. -d classes src/com/example/web/BeerSelect.java
```

在Windows操作系统上用分号 (;)

用-d选项告诉编译器，把.class文件放在适当包结构中的classes目录下。你的.class文件会放在/beerV1/classes/com/example/web/下。

## 部署servlet

要部署servlet，建立.class文件的一个副本，并把它移到部署结构的/Beer-v1/WEB-INF/classes/com/example/web/目录下。

## 测试servlet

1. 重启tomcat!

2. 启动浏览器，访问：

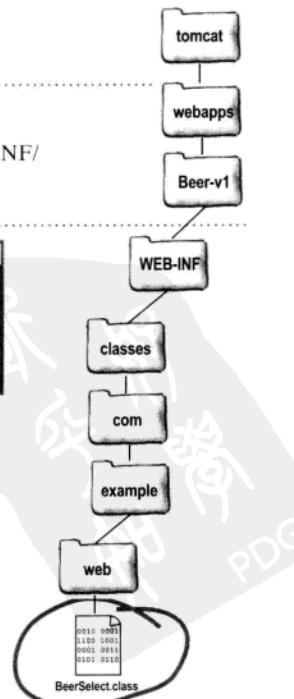
<http://localhost:8080/Beer-v1/form.html>

4. 选择一种啤酒颜色，点击“Submit”

5. 如果你的servlet能正常运行，就能在浏览器上看到servlet的响应显示为：

Beer Selection Advice

Got beer color brown



## 构建和测试模型类

在MVC中，模型是指应用的“后台”。通常是一个遗留系统，但现在想通过Web提供这个系统。大多数情况下，这只是普通的Java代码，根本不知道它可能被servlet调用。不能把模型限制为只能由一个Web应用使用，所以它应当有自己的工具包。

### 模型规范

- 包应当是com.example.model。
- 其目录结构应当是/WEB-INF/classes/com/model。
- 提供一个方法getBrands()，取一个喜欢的啤酒颜色（String）作为参数，并返回一个ArrayList，其中包含推荐的啤酒品牌（也是String）。

### 为模型构建测试类

为模型创建测试类（对，就是要在构建模型本身之前先创建测试类）。你得自己来完成，在这个教程中我们没有提供这样一个测试类。记住，刚开始测试模型时，模型还在开发环境中，与其他Java类一样，此时无需启动Tomcat也能测试。

### 构建和测试模型

模型可能非常复杂。通常会涉及与遗留数据库的连接，还要调用复杂的业务逻辑。

以下就是我们“复杂的”、基于规则的专家系统，可以用来提供啤酒建议。

```
package com.example.model;
import java.util.*;

public class BeerExpert {
    public List getBrands(String color) {
        List brands = new ArrayList();
        if (color.equals("amber")) {
            brands.add("Jack Amber");
            brands.add("Red Moose");
        } else {
            brands.add("Jail Pale Ale");
            brands.add("Gout Stout");
        }
        return(brands);
    }
}
```

注意：我们怎样利用高级的条件表达式来  
表述有关啤酒的“复杂”专家知识。

```
File Edit Window Help Skateboard
% cd beerV1
% javac -d classes src/com/example/model/BeerExpert.java
```

# 改进这个servlet，让它调用模型来得到真正的建议……

在这个第2版的servlet中，我们会改进doPost()方法，调用模型来得到建议（第3版还会从JSP提供建议）。代码改动很小，但重要的是，需要了解如何重新部署这个改进后的Web应用。可以编写代码、重新编译，再自行完成部署。也可以看看下一页，然后照着做……



## 改进servlet，第2版

先把servlet放在一边，只考虑Java。如果要完成下面的工作，需要采取哪些步骤？

1. 改进doPost()方法来调用模型。
2. 编译servlet。
3. 部署和测试更新后的Web应用。

```
public class BeerSelect extends HttpServlet {
```

## 第2版Servlet的代码

记住，模型只是普通的Java，所以我们会像调用任何其他Java方法一样，先实例化模型类，再调用它的方法！

```
package com.example.web;  
  
import com.example.model.*; ← 不要忘记导入BeerExpert所在的包。  
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.io.*;  
import java.util.*;  
  
public class BeerSelect extends HttpServlet {  
  
    public void doPost(HttpServletRequest request,  
                       HttpServletResponse response)  
        throws IOException, ServletException {  
  
        String c = request.getParameter("color"); 实例化BeerExpert类。  
        BeerExpert be = new BeerExpert(); ← 并调用getBrands().  
        List result = be.getBrands(c);  
  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        out.println("Beer Selection Advice<br>");  
  
        Iterator it = result.iterator();  
        while(it.hasNext()) {  
            out.print("<br>try: " + it.next());  
        }  
    }  
}  
  
    }  
} 打印建议（模型返回的ArrayList中的啤酒品牌项）。在最后一个版本（第3版）中，建议会从JSP打印，而不是从servlet打印。
```

# 第2版servlet的关键步骤

主要有两件事要做：重新编译servlet和部署模型类。

## 编译servlet

这里使用的编译器命令与构建第1版servlet时所用的编译器命令完全相同。

```
File Edit Window Help PlayGo
% cd beerV1
% javac -classpath /Users/bert/Applications2/tomcat/common/lib/
servlet-api.jar:classes:. -d classes src/com/example/web/BeerSelect.java
```

## 部署和测试Web应用

现在，除了servlet，我们还要部署模型。主要步骤是：

1. 把servlet .class文件的一个副本移到以下位置：

..../Beer-v1/WEB-INF/classes/com/example/web/

这会替换第1版的servlet类文件！

2. 把模型的.class文件副本移到：

..../Beer-v1/WEB-INF/classes/com/example/model/

3. 关闭并重启tomcat。

4. 通过form.html测试这个应用。

最后的浏览器输出应当如下：

Beer Selection Advice

try: Jack Amber

try: Red Moose

```
File Edit Window Help SellHigh
% cd tomcat
% bin/shutdown.sh
% bin/startup.sh
```

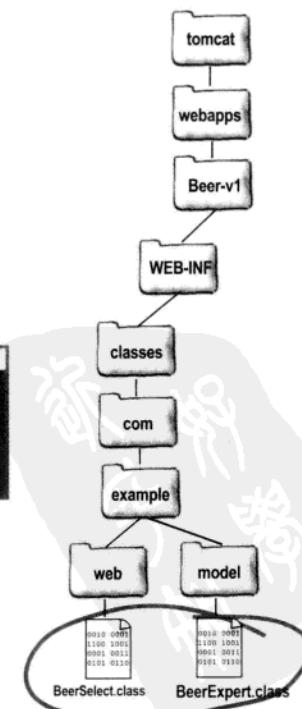
form.html  
http://localhost:8080/Beer-v1/form.html

**Beer Selection Page**

Select beer characteristics

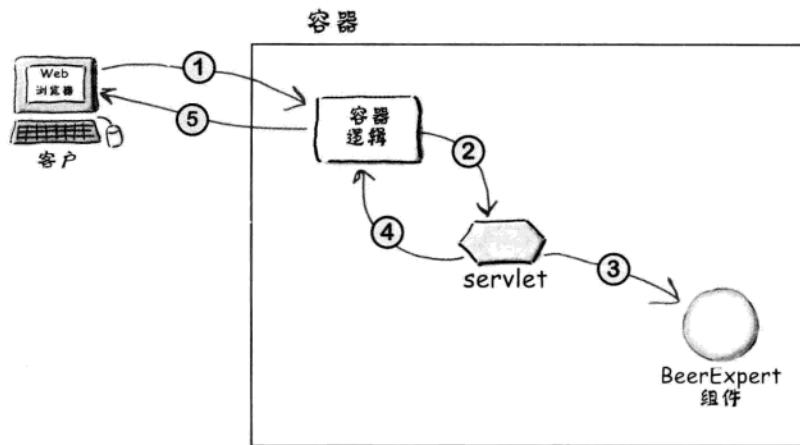
Color: light ↗

Submit ↘

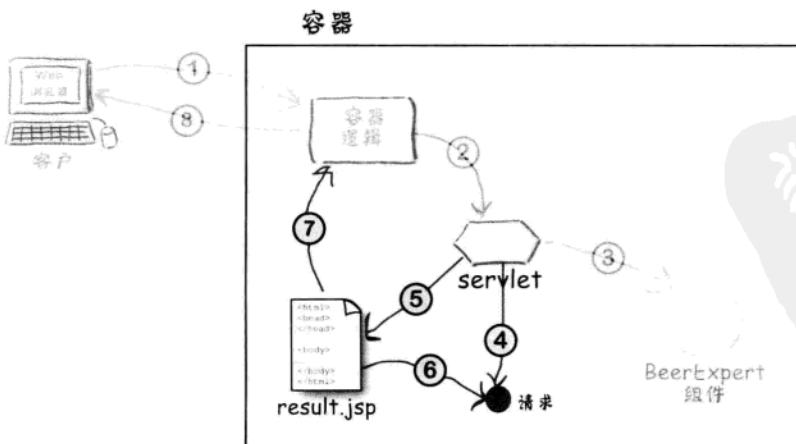


# 回顾已经完成的 MVC啤酒顾问Web应用

目前为止做的工作是……



我们希望的是……



1. 浏览器把请求数据发送给容器。
2. 容器根据URL找到正确的servlet，并把请求传递给这个servlet。
3. servlet调用BeerExpert寻求帮助。
4. servlet输出响应（打印建议）。
5. 容器把这个页面返回给心满意的用户。

1. 浏览器把请求数据发送给容器。
2. 容器根据URL找到正确的servlet，并把请求传递给这个servlet。
3. servlet调用BeerExpert寻求帮助。
4. 这个“专家”类返回一个回答，servlet把这个回答增加到请求对象。
5. servlet把请求转发给JSP。
6. JSP从请求对象得到回答。
7. JSP为容器生成一个页面。
8. 容器把这个页面返回给心满意的用户。

## 创建提供建议的JSP “视图”

不要期望太高。过几章我们才会真正谈到JSP。而且，这个JSP实际上并不太好，部分原因在于里面有一些scriptlet代码，这个问题本书后面还会专门讨论。对现在来讲，这个JSP应该很容易读懂，而且如果你想实际尝试一下，可以拿来用用看。虽然我们现在就可以从浏览器测试这个JSP，但还是先等一等，等我们修改了servlet（第3版）之后再来看它是否能正常工作。

JSP如下……

```
<%@ page import="java.util.*" %> ← 这是一个“页面指令”（我们认为  
这个名字很贴切地反映了它要做什么）。  
  
<html>  
<body>  
<h1 align="center">Beer Recommendations JSP</h1> ← 一些标准HTML（在JSP世界中称为“模板文本”）。  
<p>  
  
<%  
List styles = (List)resquest.getAttribute("styles");  
Iterator it = styles.iterator();  
while(it.hasNext()) {  
    out.print("<br>try: " + it.next());  
}  
%> ← <% %> 标记里有一些标准  
Java代码（这称为scriptlet代码）。  
</body>  
</html>
```

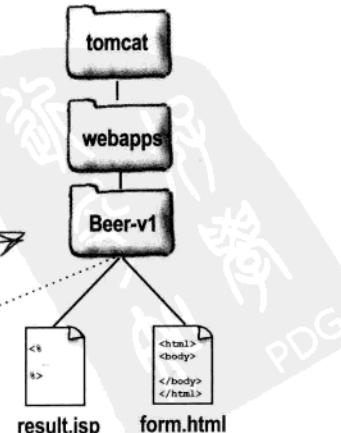
## 部署JSP

不用编译JSP（这个工作会在第一个请求到达时由容器完成）。

不过我们必须做以下工作：

1. 把它命名为“result.jsp”。
2. 保存在开发环境的/web/中。
3. 将其副本移到部署环境的/Beer-v1/中。

← 这里从请求对象得到一个属性。  
本书稍后会解释有关属性的所有  
内容，并说明我们如何得到请求  
对象……



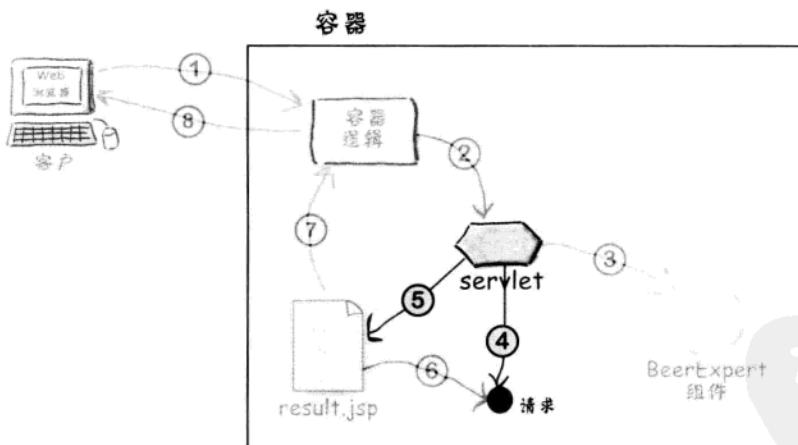
## 改进这个servlet，让它“调用”JSP（第3版）

这一步，我们要把servlet修改为“调用”JSP来生成输出（视图）。容器提供了一种称为“请求分派”的机制，允许容器管理的一个组件调用另一个组件。我们就会使用这种机制，servlet从模型得到信息，把它保存在请求对象中，然后把请求分派给JSP。

必须对这个servlet做的重要修改：

1. 把模型组件的回答增加到请求对象，以便JSP访问(第4步)。
2. 要求容器把请求转发给“result.jsp”（第5步）。

1. 浏览器把请求数据发送给容器。
2. 容器根据URL找到正确的servlet，并把请求传递给这个servlet。
3. servlet调用BeerExpert类寻求帮助。
4. 这个“专家”类返回一个回答，servlet把这个回答增加到请求对象。
5. servlet把请求发派给JSP。
6. JSP从请求对象得到回答。
7. JSP为容器生成一个页面。
8. 容器把这个页面返回给心满意足的用户。



## 第3版servlet的代码

如下修改servlet，将模型组件的回答增加到请求对象（以便JSP获取），并要求容器把请求分派给JSP。

```

package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class BeerSelect extends HttpServlet {

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

        String c = request.getParameter("color");
        BeerExpert be = new BeerExpert();
        List result = be.getBrands(c);

        // response.setContentType("text/html");
        // PrintWriter out = response.getWriter();
        // out.println("Beer Selection Advice<br>");

        request.setAttribute("styles", result);
    }

    RequestDispatcher view =
        request.getRequestDispatcher("result.jsp");
    view.forward(request, response );
}

```

既然要由JSP生成输出，就要从servlet中删除测试输出。为了让你还能看到，这里只是把它们注释掉。

为请求对象增加一个属性，供JSP使用。注意，JSP要寻找“styles”。

为JSP实例化一个请求分派器。

使用请求分派器要求容器准备好JSP，并向JSP发送请求和响应。

# 编译、部署和测试最后的应用！

在这一章中，我们构建了一个完整（尽管很小）的MVC应用，这里使用了HTML、servlet和JSP。你完全可以在你的简历里加上一笔，写上“曾开发过MVC应用”。

## 编译servlet

还是用前面的编译器命令：

```
File Edit Window Help RunntsATrap
% cd beerV1
% javac -classpath /Users/bert/Applications2/tomcat/common/lib/
servlet-api.jar:classes:. -d classes src/com/example/web/BeerSelect.java
```

## 部署和测试Web应用

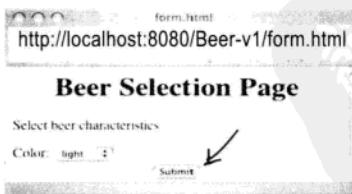
下面来重新部署servlet。

1. 把servlet的.class文件副本移到../Beer-v1/WEB-INF/classes/com/example/web/（同样地，这会替换掉前面的第2版servlet类文件）。

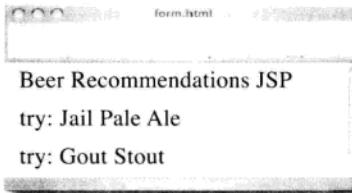
2. 关闭并重启tomcat。

```
File Edit Window Help SaveYourself
% cd tomcat
% bin/shutdown.sh
% bin/startup.sh
```

3. 通过form.html测试应用。



你应该能看到这个输出！ →



嘿，他现在可以构建MVC应用了，  
但是他还不知道怎么使用JSP表达式  
语言（也就是JSTL），不知道怎么写  
定制标记，不知道怎么使用过滤器。我还  
曾经看见他不务正业。他要学的还多  
着呢……

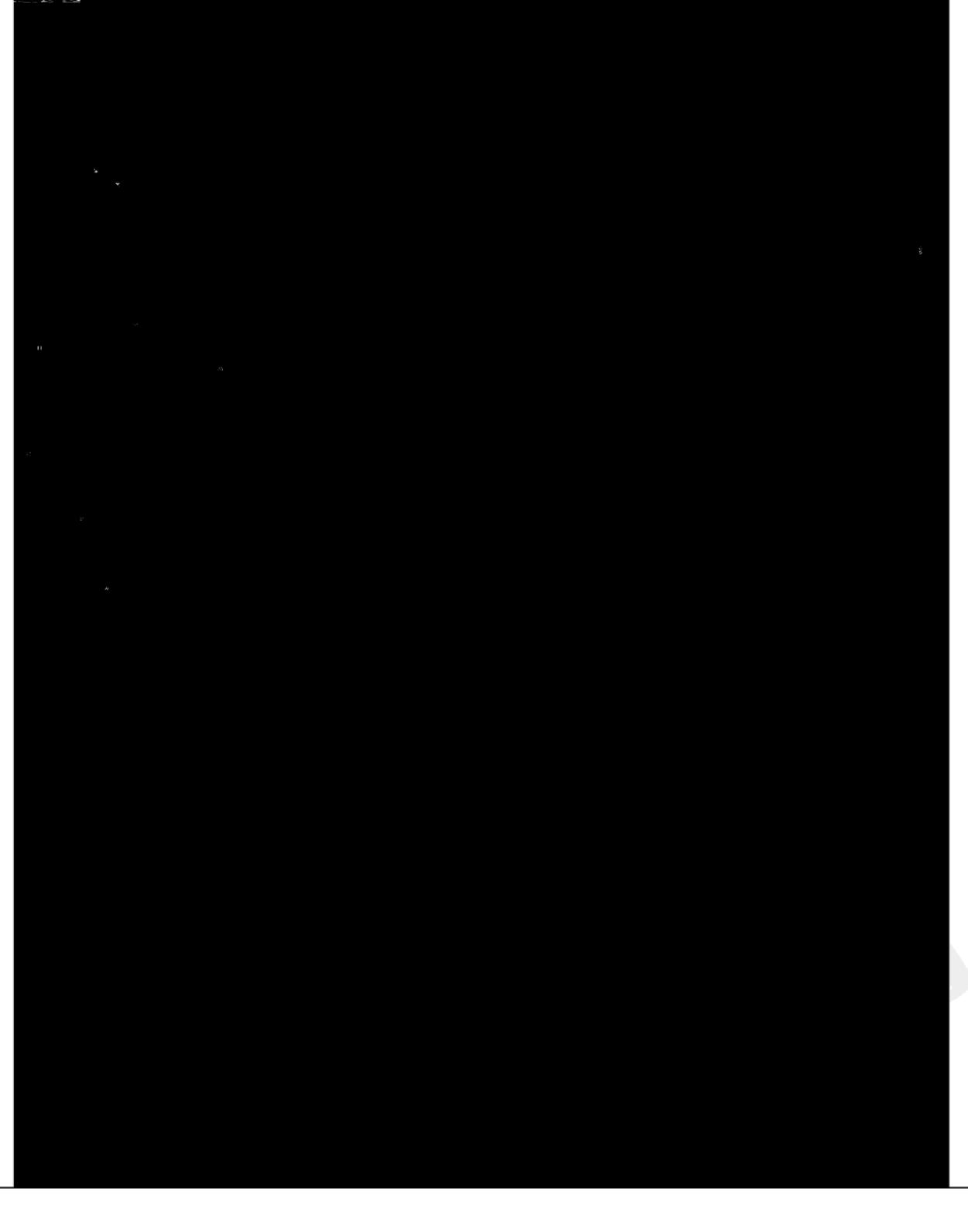


## 还有很多需要学。

好日子告一段落。这三章都只是介绍，写了一点点代码，而且充分地分析了有关HTTP请求/响应的内容。

不过，这本书还有200个模拟题在等着你呢，从下一章开始就要有模拟测验了。除非你已经很熟悉servlet开发和部署，否则先不要翻到下一页，等你真正领会了这一章的教程之后再投入下一阶段的学习吧。

我们不是在给你压力，也不是给你布陷阱……



# 作为Servlet

他居然用一个GET请求来  
更新数据库。后果肯定很严  
重……可能90天不许上苏  
珊的瑜伽课。



Servlet的存在就是要为客户提供服务。servlet的任务是得到一个客户的请求，再发回一个响应。请求可能很简单：“请给我一个欢迎页面。”也可能很复杂：“为我的购物车结账。”这个请求携带着一些重要的数据，你的servlet代码必须知道怎么找到和使用这个请求。响应也携带着一些信息，浏览器需要这些信息来显示一个页面（或下载数据），你的servlet代码必须知道怎么发送这些信息。或者不发送……你的servlet也可以把请求传递给其他人（另一个页面、servlet，或者JSP）。

# OBJECTIVES

## Servlet技术模型

- 1.1 对于每一种HTTP方法（如GET、POST、HEAD等），描述该方法的用途，以及该HTTP方法协议的技术特性，并列出客户（通常是一个Web浏览器）会因为哪些原因使用这种方法，明确对应这种HTTP方法的HttpServlet方法。
- 1.2 使用HttpServletRequest接口，编写代码从请求获取HTML表单参数，获取HTTP请求首部信息，或者从请求获取cookie。
- 1.3 使用HttpServletResponse接口，编写代码设置HTTP响应首部，设置响应的内容类型，为响应获得一个文本流，为响应获得一个二进制流，把一个HTTP请求重定向到另一个URL，或者向响应增加cookie（注1）。
- 1.4 描述servlet生命周期的作用和事件序列：(1) servlet类加载；(2)servlet实例化；(3)调用init()方法；(4)调用service()方法；(5)调用destroy()方法。

注1：在关于“会话”的一章之前，不会涉及太多有关cookie的大纲要求。

### 内容说明：

这一部分的大纲要求在本章中基本上都会介绍，只有1.3条中有关cookie的部分除外。第一章的大多数内容在前两章都已经涉及到过，在第2章我们曾经说过，

“不用急着把它记住。”

但在这一章里，你必须放慢速度，认真地学习，而且要记住这些内容。其他章节再不详细地讨论这些要求了，所以你必须在这章真正掌握。

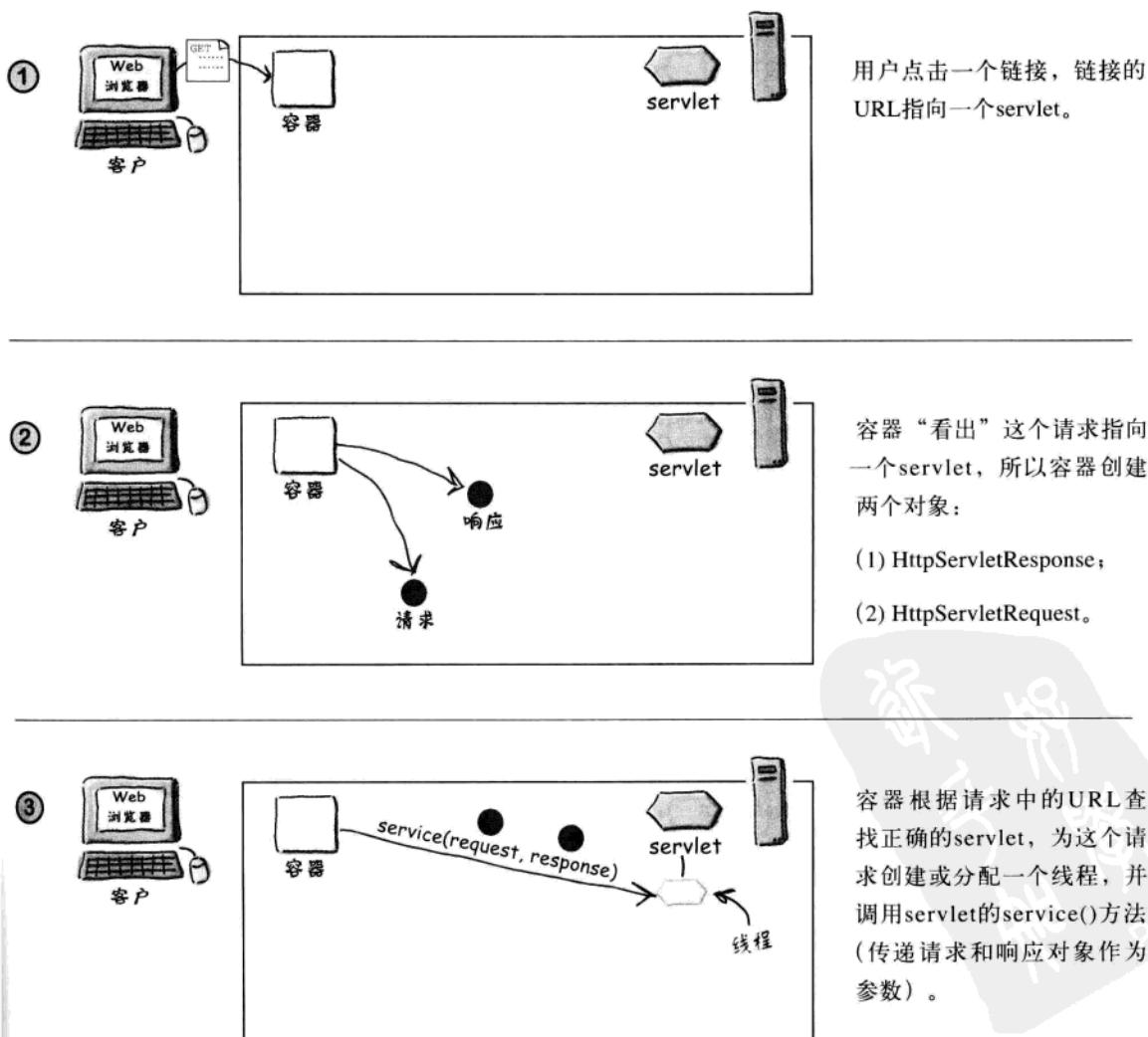
动手做做这一章的练习，查阅有关的资料，然后再完成这一章最后的模拟测验题。如果没有做到80%正确，就要回过头来再好好阅读这一章，看看哪些地方漏掉了，然后才继续看第5章。

有些模拟测验题涉及的要求已经移到第5章和第6章，因为这些问题还需要了解另外一些知识，而这些知识到第5章和第6章才会涉及到。这意味着，这一章的模拟测验题比较少，后面两章的模拟测验题会比较多，这样就不至于考你还没讲到的内容。

重要提示：前三章介绍的是背景信息，但下一页开始就不同了，你看到的每个内容要么与考试直接相关，要么本身就是考试的一部分。

# Servlet受容器的控制

在第2章，我们了解到容器全盘控制着servlet的一生，它会创建请求和响应对象、为servlet创建一个新线程或分配一个线程，另外调用servlet的service()方法，并传递请求和响应对象的引用作为参数。下面是一个简单的回顾……

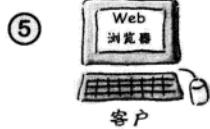


继续……



service()方法根据客户发出的HTTP方法（GET、POST等），确定要调用哪个servlet方法。

客户发出了一个HTTP GET请求，所以service()方法会调用servlet的doGet()方法，并传递请求和响应对象作为参数。



servlet使用响应对象将响应写至客户。响应通过容器传回。



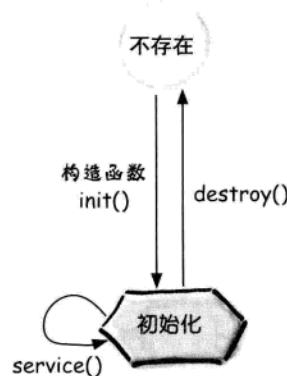
service()方法结束，所以线程要么撤销，要么返回到容器管理的一个线程池。请求和响应对象引用已经出了作用域，所以这些对象已经没有意义（可以垃圾回收）。

客户得到响应。

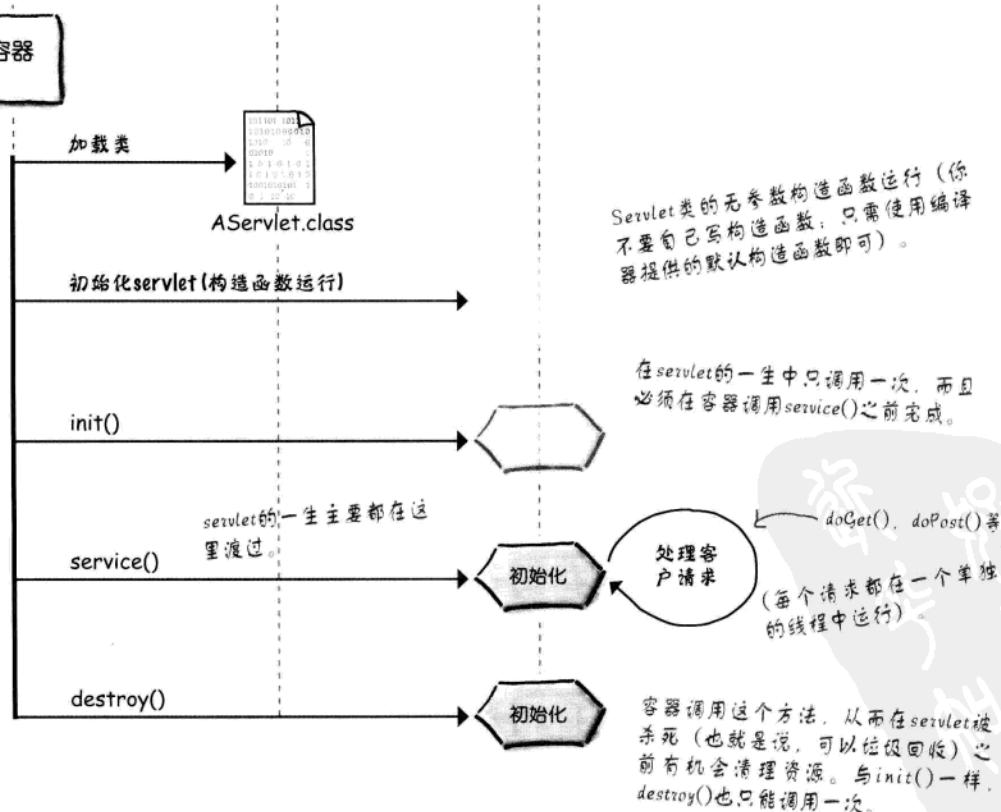
# 但是，servlet的一生还不只这些

我们直接从servlet一生的中期切入，不过还有一些问题没有回答：  
Servlet类什么时候加载？什么时候运行servlet的构造函数？Servlet对象能活多久？servlet应该什么时候初始化资源？又该什么时候清理资源？

servlet的生命周期很简单：只有一个主要的状态——初始化。如果servlet没有初始化，则要么正在初始化（运行其构造函数或init()方法）、正在撤销（运行其destroy()方法），要么就是还不存在。

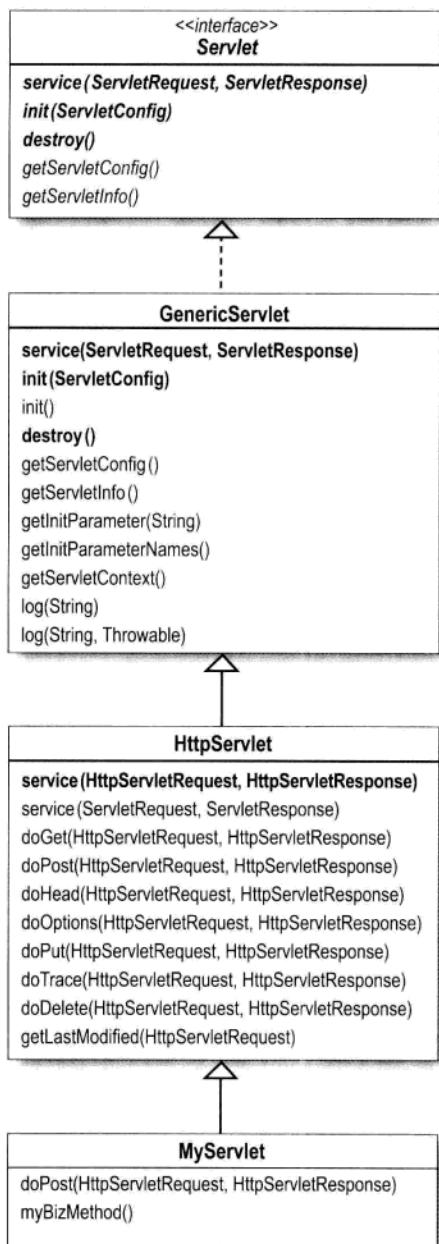


Web容器      Servlet类      Servlet对象



注意：到目前为止就完全记住了这些内容！只要对API如何工作有一个认识就行了……

# 你的servlet继承了生命周期方法



Servlet接口  
(javax.servlet.Servlet)

Servlet接口指出，所有servlet都有这5个方法（其中3个用粗体显示的方法是生命周期方法）。

GenericServlet类  
(javax.servlet.GenericServlet)

GenericServlet是一个抽象类。它实现了你需要的大部分基本servlet方法（包括Servlet接口中的方法）。你可能不会自己扩展这个类。大多数servlet的“servlet行为”都来自这个类。

HttpServlet类  
(javax.servlet.http.HttpServlet)

HttpServlet（也是一个抽象类），实现了一个service()方法来反映servlet的HTTP特性——service()方法不仅仅可以取一般的servlet请求和响应，还可以取HTTP特定的请求和响应作为参数。

MyServlet类  
(com.wickedlysmart.foo)

大多数servlet行为都由超类方法处理。你要做的只是覆盖所需的HTTP方法。

# 生命周期中的三大重要时刻

1

init()

## 何时调用

servlet实例创建后，并在servlet能为客户请求提供服务之前，容器要对servlet调用init()。

## 作用

使你在servlet处理客户请求之前有机会对其初始化。

## 是否覆盖？

有可能。

如果有初始化代码（如得到一个数据库连接，或向其他对象注册），就要覆盖servlet类中的init()方法。

2

service()

## 何时调用

第一个客户请求到来时，容器会开始一个新线程，或者从线程池分配一个线程，并调用servlet的service()方法。

## 作用

这个方法会查看请求，确定HTTP方法（GET、POST等），并在servlet上调用对应的方法，如doGet()、doPost()等。

## 是否覆盖？

不，不太可能。

不应该覆盖service()方法。你的任务是覆盖doGet()和/或doPost()方法，而由HTTPServlet中的service()实现来考虑该调用哪一个方法（doGet()、doPost()等）。

3

doGet()

和/或

doPost()

## 何时调用

service()方法根据请求的HTTP方法（GET、POST等）来调用doGet()或doPost()。

（这里只列出了doGet()和doPost()，因为你可能只会用到这两个方法）。

## 作用

要在这里写你的代码！你的Web应用想要做什么，就要由这个方法负责。

当然，也可以调用其他对象的其他方法，不过都要从这里开始。

## 是否覆盖？

至少要覆盖其中之一！

（doGet()或doPost()）。

不论覆盖哪一个，都能告诉容器你支持什么类型的请求。例如，如果没有覆盖doPost()，就是在告诉容器这个servlet不支持HTTP POST请求。

我想我明白……容器会调用servlet的init()方法，但是如果我不覆盖init()，就会运行GenericServlet中的init()。然后到来一个请求时，容器会开始或分配一个线程，调用service()方法，这个方法不用我覆盖，所以会运行HttpServlet的service()方法。HttpServlet service()方法再调用我覆盖的doGet()或doPost()。这样一来，每次运行我的doGet()或doPost()时，它都在一个单独的方法栈中。



## Servlet初始化

## 线程 A

servlet实例创建后，并在servlet能为客户请求提供服务之前，容器会在servlet实例上调用init()。

如果你有初始化代码（如得到一个数据库连接，或者向其他对象注册），就要覆盖servlet类的init()方法。否则会运行GenericServlet的init()方法。



service()方法总是在其自己的栈中调用……

## 客户请求1

## 线程 B

第一个客户请求到来时，容器会开始（或找到）一个线程，并调用servlet的service()方法。

通常不用覆盖service()方法，所以会运行HttpServlet的service()方法。service()明确请求中的HTTP方法（GET、POST等），并调用相应的doGet()或doPost()方法。HttpServlet中的doGet()和doPost()方法什么也不做，所以你必须覆盖其中之一，或者两个方法都覆盖。service()方法结束时，线程也结束（或者放回到容器管理的一个线程池中）。

## 客户请求2

## 线程 C

第二个（及所有其他）客户请求到来时，容器再创建或找到另一个线程，并调用servlet的service()方法。

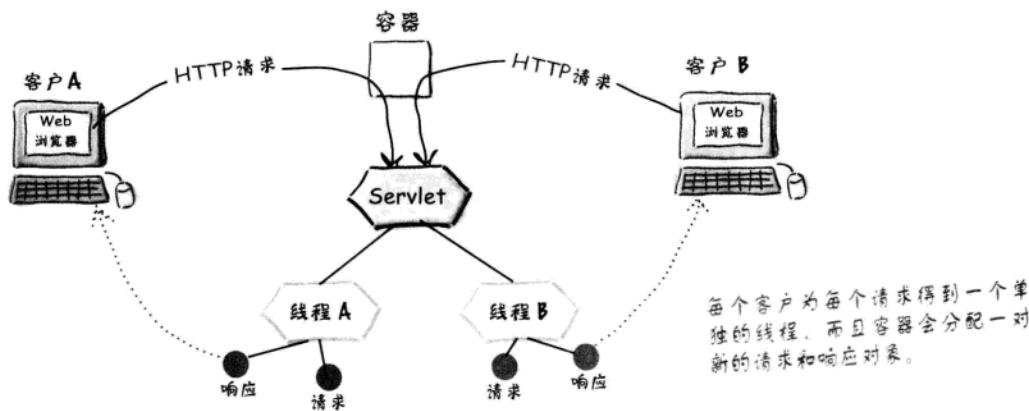
所以每次到来一个客户请求时，都会完成一个service() --> doGet()方法调用序列。在任何时刻，有多少客户请求就至少有多少可运行（runnable）的线程，这将受容器资源或容器策略/配置的限制（例如，你的容器允许你指定最多有多少个并发线程，当客户请求数超过这个上限时，有些客户就必须等待）。

# 每个请求都在一个单独的线程中运行！

你可能听过别人这么说，“servlet的每个实例……”但这种说法是错误的。任何servlet类都不会有多个实例，只有一种特殊情况例外（称为SingleThreadModel，这其实很糟糕），不过我们还不会讨论这种特殊情况。

容器运行多个线程来处理对一个servlet的多个请求。

对应每个客户请求，会生成一对新的请求和响应回对象。



## there are no Dumb Questions

**问：** 我有点不明白了……在上面的图中，你画了两个不同的客户，每个客户都有自己的线程。如果同一个客户做了多个请求会怎么样呢？是每个客户一个线程，还是每个请求一个线程？

**答：** 应该是每个请求一个线程。容器并不关心是谁做的请求，每个到来的请求都意味着一个新的线程/栈。

**问：** 如果容器使用了集群，把应用分布在多个JVM上又会怎么样呢？

**答：** 假设上面的图对应的的是一个JVM。那么每个JVM都有一个同样的图。所以，在一个分布式Web应用中，每个JVM都会有特定servlet的一个实例。不过对于每个JVM来说，仍然只有该servlet的一个实例。

**问：** 我注意到HttpServlet和GenericServlet不在同一个包里……到底有多少个servlet包？

**答：** 与servlet相关的所有一切（但除JSP以外）都要么在javax.servlet中，要么在javax.servlet.http中。而且很容易指出二者的差别……与HTTP有关的都在javax.servlet.http包中，其余的（通用servlet类和接口）则在javax.servlet包中。本书后面有专门介绍JSP相关内容的章节。

## 刚开始：加载和初始化

容器找到servlet类文件时，servlet的生命开始。这基本上都是在容器启动时发生（例如，运行Tomcat时）。容器启动时，它会寻找已经部署的Web应用，然后开始搜索servlet类文件（在有关“部署”的一章中，我们将更详细地介绍容器如何寻找servlet）。

寻找类只是第一步。

第二步是加载类，这可能在容器启动时发生，也可能在第一个客户使用时进行。你的容器可能允许你来完成类加载，也可能会在它希望的任何时刻加载类。不论你的容器是早就准备好servlet，还是在第一个客户需要时才即时地加载类，在servlet没有完全初始化之前绝不能运行servlet的service()方法。

你的servlet总是在为第一个客户请求提供服务之前得到加载和初始化。

init()总是在第一个service()调用之前完成。



### 开动脑筋

为什么有一个init()方法？换句话说，构造函数不足以初始化servlet吗？

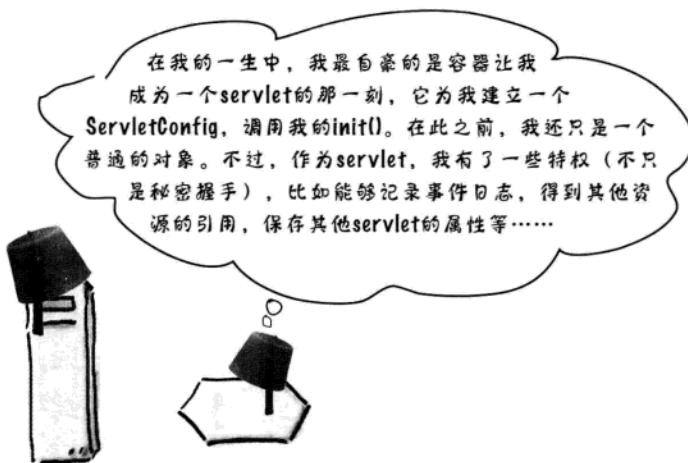
在init()方法中会放什么代码？

提示：init()方法要取一个对象引用参数。你认为init()方法的这个参数可能是什么，你要怎样使用它？

# Servlet初始化： 对象何时成为一个Servlet



`init()`在servlet一生中只运行一次，所以不要浪费！而且不要过早地做事情……在构造函数中做有关servlet的事情为时太早。



servlet从“不存在”状态迁移到“初始化”状态（这意味着已经准备好为客户请求提供服务），首先是从构造函数开始。但是构造函数只是使之成为一个对象，而不是一个servlet。要想成为一个servlet，对象必须具备一些“servlet特性”（servletness）。

对象成为一个servlet时，它会得到servlet该有的所有特权，比如能够使用ServletContext引用从容器得到信息。

## 为什么我们要关心初始化细节呢？

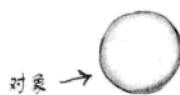
因为在调用构造函数和`init()`方法之间，servlet处在一种薛定谔 (Schroedinger's) (注2) servlet状态。你可能有一些servlet初始化代码，如得到Web应用配置信息，或查找应用另一部分的一个引用，如果在servlet的生命中太早运行这些初始化代码就会失败。不过这很简单，你只要记住一点，不要在servlet的构造函数中放任何东西！

别着急，什么都可以放在`init()`里。

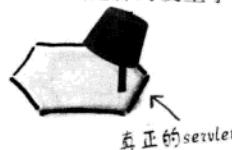
注2：如果你量子力学学得不好，可以通过Google搜索一下“Schroedinger's Cat”（提醒养宠物的人，这可不是什么宠物）。谈到薛定谔状态时，我们是指既没有完全死，也不完全活，是介于这两者之间的一个状态。

## “作为servlet”有什么好处？

servlet从这样：



变成这样时发生了什么？





**当心！**

不要把ServletConfig参数与ServletContext参数搞混了！

下一章才会明确地谈到这些，不过很多人都把它们搞混了，所以我们想先交待两句：要注意二者的区别。

先来看名字上的不同：ServletConfig中有一个“config”，这代表“配置”，涉及你为servlet配置的一些部署时值（每个servlet有一个ServletConfig）。这是servlet可能要访问的信息，但是你又不想把它们硬编码写到servlet中，如数据库名。

一旦servlet部署并运行，ServletConfig参数就不能再改变了。要想改变，必须重新部署servlet。

ServletContext叫做AppContext更合适（不过我们这个建议没有被采纳），因为每个Web应用只有一个ServletContext，而不是每个servlet都有一个。不过，这些内容会在下一章再详细说明，这里只是事先做个提示。

### ① ServletConfig 对象

- 每个servlet都有一个ServletConfig对象。
- 用于向servlet传递部署时信息（例如数据库或企业bean的查找名），而你不想把这个信息硬编码写到servlet中（servlet初始化参数）。
- 用于访问ServletContext。
- 参数在部署描述文件中配置。

### ② ServletContext

- 每个Web应用有一个ServletContext（应该叫做AppContext才对）。
- 用于访问Web应用参数（也在部署描述文件中配置）。
- 相当于一种应用公告栏，可以在这里放置消息（称为属性），应用的其他部分可以访问这些消息（下一章还会详细介绍这个内容）。
- 用于得到服务器信息，包括容器名和容器版本，以及所支持API的版本等。

# 但是servlet的真正任务是处理请求，这才是servlet存在的意义。

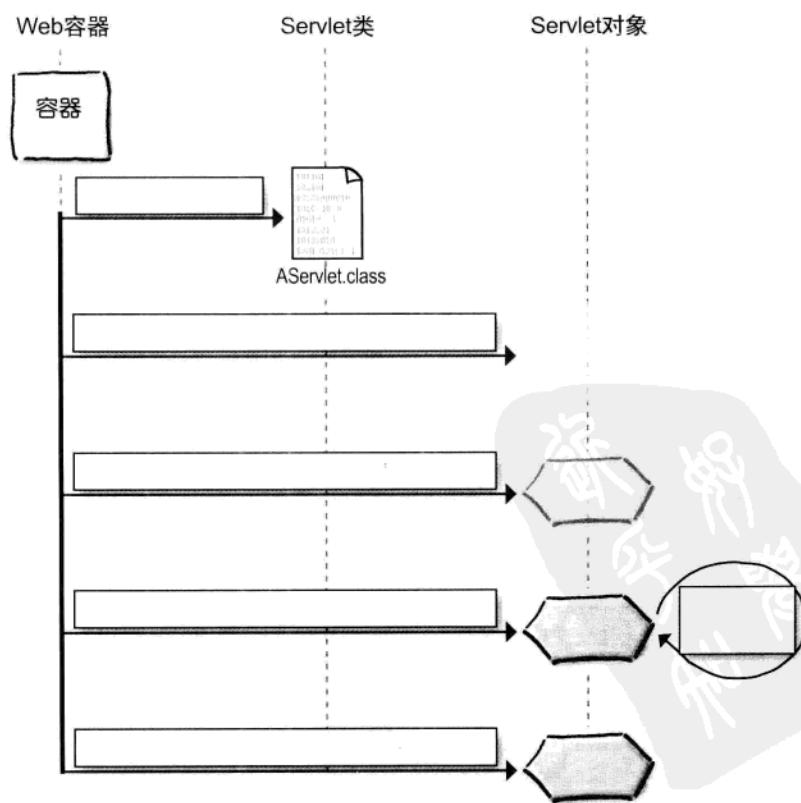
下一章我们会分析ServletConfig和ServletContext，不过对于现在来讲，我们将深入地讨论请求和响应的细节。因为ServletConfig 和ServletContext的存在只是为了支持servlet的真正任务：处理客户请求！所以在介绍上下文对象和配置对象如何帮助你完成这个任务之前，先退一步，了解一下请求和响应的基础知识。

你已经知道了，要把一个请求和响应作为参数传递给doGet()或doPost()方法，但是这些请求和响应对象能给你什么呢？这些对象怎么处理，为什么需要这些对象？

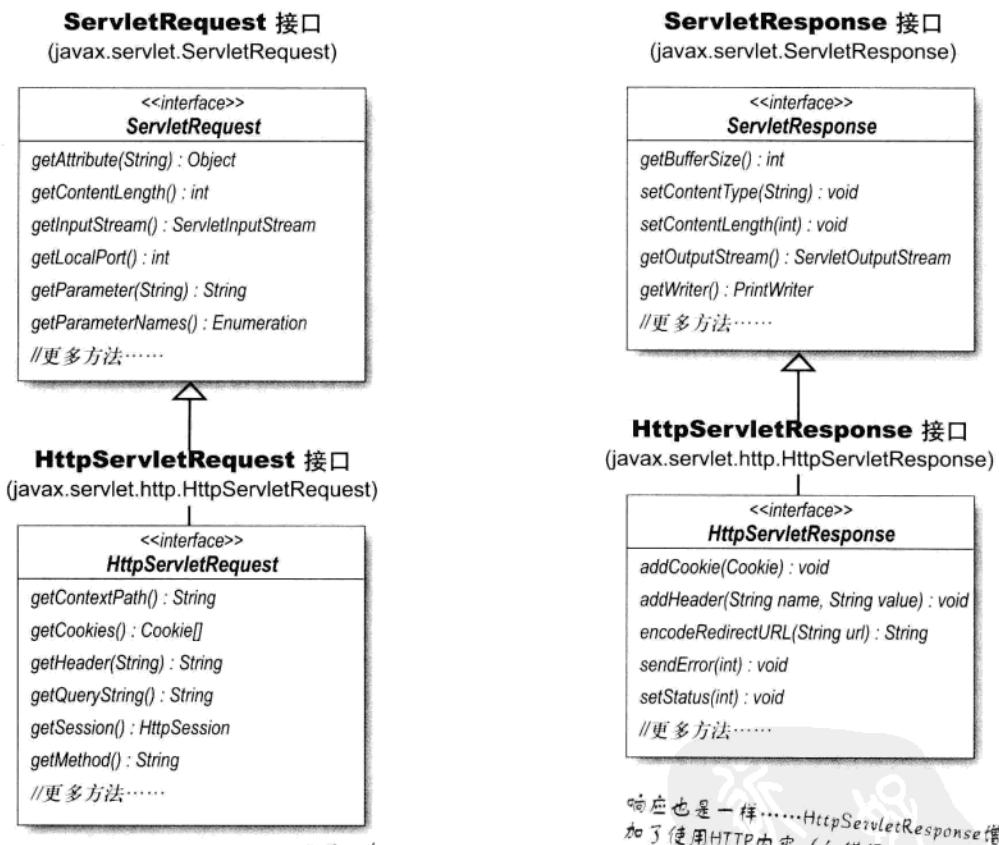


填写这个生命周期时间图中缺少的部分（空框），根据这一章前面给出的时间图，检查你的答案是否正确。

还可以加上你自己的标注，这样能帮助你记住这些细节。



# 请求和响应：这是一切一切的关键，也是service()的参数（注3）



HttpServletRequest方法与HTTP有关，如cookie、首部和会话。

HttpServletResponse接口增加了与HTTP协议相关的方法……你的servlet与客户/浏览器通信时就要使用这些方法。

响应也是一样……HttpServletResponse增加了使用HTTP内容（如错误、cookie和首部等）的有关方法。

注3： 请求和响应对象也是你写的其他HttpServlet方法的参数——doGet(), doPost()等。

there are no  
Dumb Questions

**问：**谁来实现HttpServletRequest和HttpServletResponse接口？这些类在API中吗？

**答：**第一个问题的回答是容器，第二个问题的答案是“不”。这些类不在API中，因为它们要由开发商来实现。好在这一点不用你担心。只要相信一点，调用servlet的service()方法时，肯定会向它传递两个非常好的对象的引用，这两个对象分别实现了HttpServletRequest和HttpServletResponse。完全不用你考虑实际的实现类名或类型。你要关心的只是你会得到些什么，它们拥有HttpServletRequest和HttpServletResponse的所有功能。

换句话说，你只要知道在容器交给你的对象（作为请求的一部分）上能调用哪些方法！至于它们究竟在哪个实际的类中实现，这并没有关系，你只是要按接口类型来引用请求和响应对象。

**问：**我这样看这个UML图对吗？这些接口扩展了另外一些接口吗？

**答：**对，记住，接口可以有自己的继承树。一个接口扩展另一个接口（它们也只能这样做，接口不能实现另外的接口），这说明不管是谁实现了一个接口，都必须实现该接口以及其超接口中定义的所有方法。这意味着，例如，不管是谁实现了HttpServletRequest，都必须为HttpServletRequest接口中声明的方法以及ServletRequest接口中的方法提供实现方法。

**问：**我还是搞不懂为什么要有一个GenericServlet，还要有ServletRequest和ServletResponse。如果除了HTTP servlet外没有其他的了……要这些类干什么？

**答：**可不能说“没有其他的……”。可以预见，肯定会在某个地方使用servlet技术模型时没有用HTTP协议。只不过是我们自己还没有见到，或者没有读到而已。相信我。

而且设计servlet模型时提供了灵活性，充分考虑到了有人可能想使用采用其他协议的servlet，如SMTP或者一种专用的定制协议。不过，API中只对HTTP提供了内置支持，而且几乎每个人都只使用这种内置实现。



Relax

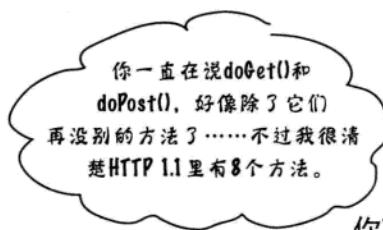
考试不要求你了解如何使用非HTTP servlet进行开发。

你不用知道怎样使用采用了其他协议（非HTTP）的servlet。不过，你应该知道类层次结构是怎样的。所以，一定要知道HttpServletRequest和HttpServletResponse分别由ServletRequest和ServletResponse扩展，而且HttpServlet的大多数实现实际上都来自于GenericServlet。

不过仅此而已。考试会认为你只是一个HttpServlet开发人员。

# HTTP请求方法确定究竟运行 doGet()还是doPost()

要记住，客户的请求总是包括一个特定的HTTP方法。如果这个HTTP方法是GET，service()方法就会调用doGet()。如果这个HTTP请求方法是POST，service()方法就会调用doPost()。



你可能不会关心GET和POST以外  
的其他HTTP方法。

对，除了GET和POST之外，确实还有其他一  
些HTTP 1.1方法，这包括HEAD、TRACE、  
OPTIONS、PUT、DELETE和CONNECT。

对于这8个方法，除了一个方法外，其余的在  
HttpServlet类中都有一个匹配的doXXX()方  
法，所以不仅是doGet()和doPost()，你还可  
以得到doOptions()、doHead()、doTrace()  
、doPut()和doDelete()。servlet API中没有处  
理doConnect()的机制，所以HttpServlet里没有  
doConnect()。

其余的这些HTTP方法可能对有些人有意义，  
比如说Web服务器开发人员，但是servlet开发  
人员除了使用GET和POST外，很少会用到其  
他方法。

对于大多数（甚至可以算是全部）servlet开发  
来说，你总会使用doGet()（针对简单请求）  
或doPost()（来接受和处理表单数据），其他的  
不用考虑。

```
GET /select/selectBeerTaste.jsp?color=dark&taste=malty
HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U;
PPC Mac OS X; en-US; rv:1.4)
Gecko/20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml+xml;text/html;q=0.9,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

```
POST /advisor/selectBeerTaste.
do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U;
PPC Mac OS X Mach-O; en-US; rv:1.4)
Gecko/20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml+xml;text/html;q=0.9,text/plain;q=0.8,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

HTTP 请求

那好，就算它们  
对我不重要…但考试  
总会考吧。



## 确实，考试时其中一个或多个 HTTP方法可能会稍稍露面……

如果你在准备考试，应该能通过一个列表来认识所有这些方法，另外至少对每个方法做什么用有一个最简单的了解。不过，不要在这里花费太多的时间！

**在实际的servlet世界中，你只用关心GET和POST就行了。**

**在考试中，则还要稍稍考虑一下其他的HTTP方法。**

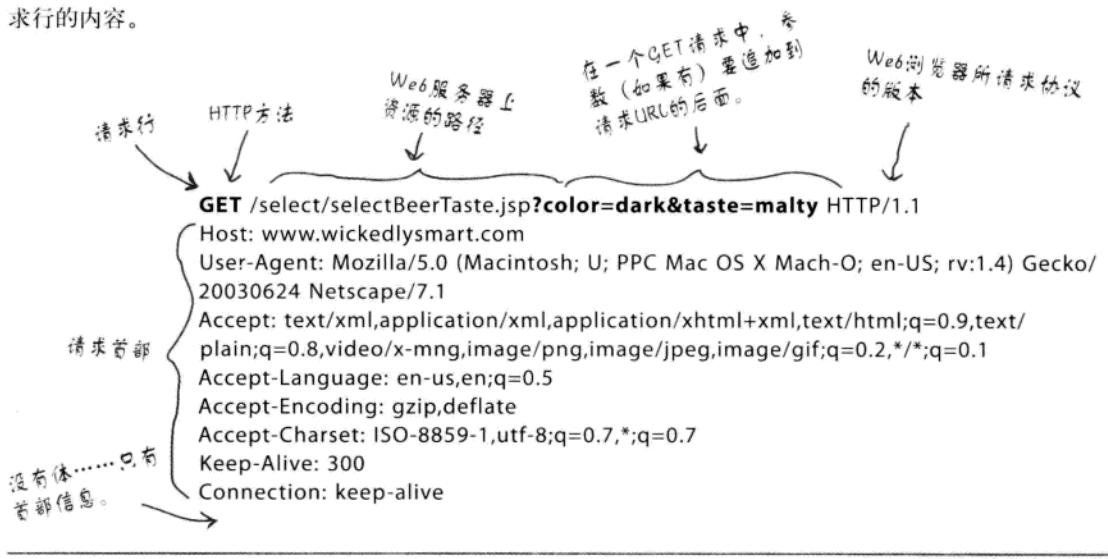
- |         |  |
|---------|--|
| GET     | 要求得到所请求URL上的一个东西（资源/文件）。   |
| POST    | 要求服务器接受附加到请求的体信息，并提供所请求URL上的一个东西。这像一个扩展的GET……也就是就，随请求还发送了额外信息的“GET”。         |
| HEAD    | 只要求得到GET返回结果的首部部分。所以这有点像GET，但是响应中没有体。它能提供所请求URL的有关信息，但是不会真正返回实际的那个东西（资源/文件）。 |
| TRACE   | 要求请求消息回送，这样客户能看到另一端上接收了什么，以便测试或排错。   |
| PUT     | 指出要把所包含的信息（体）放在请求的URL上。  |
| DELETE  | 指出删除所请求URL上的一个东西（资源/文件）。   |
| OPTIONS | 要求得到一个HTTP方法列表，所请求URL上的东西可以对这些HTTP方法做出响应。                                    |
| CONNECT | 要求连接以便建立隧道。  |

对HTTP OPTIONS请求  
的一个响应：

HTTP/1.1 200 OK  
Server: Apache-Coyote/1.1  
Date: Thu, 20 Apr 2004  
16:20:00 GMT  
Allow: OPTIONS, TRACE,  
GET, HEAD, POST  
Content-Length: 0

# GET和POST的区别

POST有一个体。这就是关键。GET和POST都能发送参数，但是利用GET的话，对参数数据有限制，参数数据只能是放在请求行的内容。

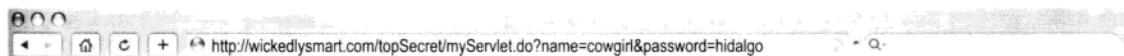




## 不，还不只是数据的大小

第一章我们谈到过GET的问题，还记得吗？

使用GET时，参数数据会显示在浏览器的输入栏（地址栏）中，就放在实际URL的后面（用一个“？”分隔）。想象一下，假如你不希望别人看到这些参数该怎么办。



所以，另一个问题就是安全性。

此外，还有一个问题，你是否需要或是否希望最终用户对请求页面建立书签。GET请求可以建立书签，POST请求则不能。在有些情况下这可能非常重要，例如，假设你有一个页面允许用户指定搜索规则。用户可能一个星期以后再回来，想再完成同样的搜索（得到同样的数据），但是此时服务器上已经有新的数据了。

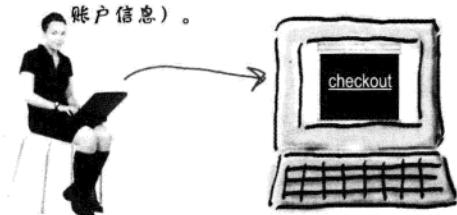
不过，除了数据大小、安全性和书签问题之外，GET和POST之间还有一个重要的差别，这就是这些方法要如何使用。GET用于得到某些东西。仅此而已，只是简单的获取。当然，你可能想使用参数来帮助确定返回的内容，但关键是，你对服务器不会做任何改变！POST则用于发送数据来进行处理。可能很简单，只是一些查询参数，用于确定要返回什么，这与GET一样。不过，使用POST时，要这样考虑：这是一个更新。要把它认为是使用POST体的数据来修改服务器上的某些东西。

这又带来了一个问题……请求是不是幂等的。如果不是，你就会遇到大麻烦，小打小闹是解决不了这个问题的。如果你对Web世界中“幂等”的含义还不太熟悉，请接着往下读……

## 非幂等请求的故事

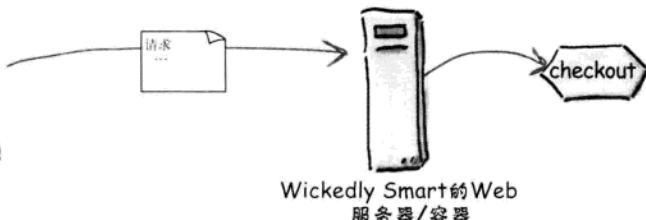
Diane有一个需求。她想从Wickedly Smart在线书店买一本《Head First Knitting》，但Diane不知道的是，这个网站还只是beta版。Diane的钱不多，她的账户里的钱只够她买一本书。她本来想直接从Amazon或O'Reilly.com网站买的，不过后来考虑到想买一本有亲笔签名的书，而这只有Wickedly Smart网站提供，所以还是决定从Wickedly Smart网站购买。这是一个会让她后悔的选择……

- ① Diane点击了CHECKOUT按钮  
(她早先已经提交了她的银行  
账户信息)。

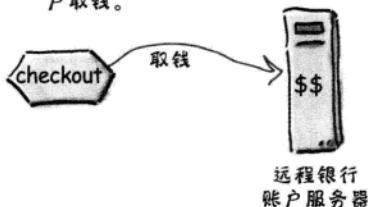


浏览器向服务器发送HTTP请求，  
提供了图书购买信息和Diane的  
客户ID号。

容器把请求发送给Checkout  
servlet进行处理。



- ② Servlet以电子方式从Diane的银行账户取钱。



- ③ Servlet更新数据库（从图书目录中  
取出书，创建一个新的发货单等）。



- ④ Servlet没有发送  
明确的响应，所以  
Diane还是看到同样的  
购物车页面，想  
着……

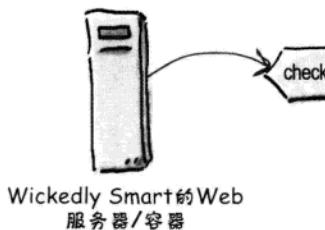


浏览器向服务器发送HTTP请求，  
提供了图书购买信息和  
Diane的客户ID号。



# 我们的故事（续）……

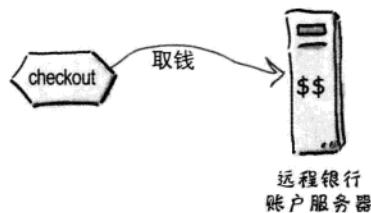
- ⑤ 容器把请求发送给Checkout servlet来处理。



- ⑥ Diane将同一本书购买了两次，但servlet并不认为这有什么问题。



- ⑦ Servlet以电子方式再一次从Diane的银行账户取钱。



- ⑧ Diane的账户接受了这次取款，但是要求她缴纳高额的透支费用。



- ⑨ 最后，Diane终于导航到Check Order Status (结账单状态) 页面，发现她为这本编织书下了两个订单……



- ⑩ 嘿，银行吗？这个Web应用程序员太蠢了（完全不是他标榜的“极聪明”），他犯了一个错误”……

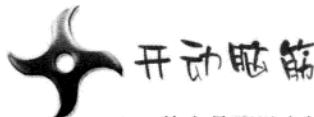




你认为哪些HTTP方法是（或者应该是）幂等的？根据你先前对这个词的理解，以及/或你刚读到的这个Diane重复买书的故事（答案在这一页最下面）。

- GET
- POST
- PUT
- HEAD

（我们特意没有写CONNECT，因为HttpServlet中不包括这个方法。）



Diane的交易哪里出问题了？

（可不只是一个问题……开发人员可能要修正好几个地方）

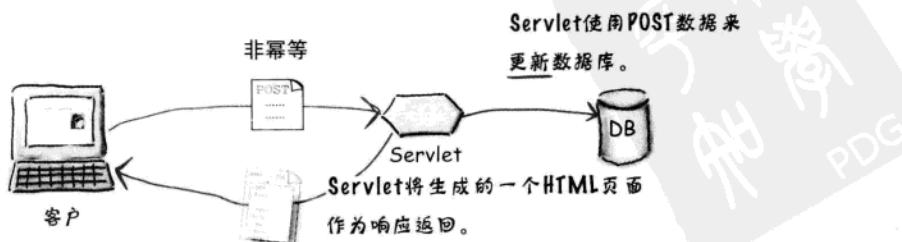
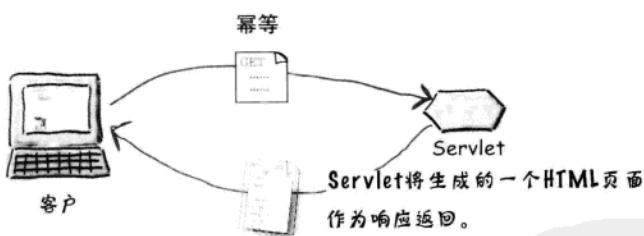
开发人员可以通过哪些方法来减少这种风险？

（提示：可能不是所有问题都能通过编程来解决）

HTTP方法中不认为是幂等的。  
HTTP方法中不认为是幂等的。POST在HTTP  
中是一个非常特殊的get()方法（但不应该这样做的）。PUT在HTTP  
中是幂等的，不过你也可以自己写一个非幂等的get()方法（但不应该这样做的）。



幂等是很好的。这说明，你可以一遍一遍反复做同一件事情，而不会有预料不到的副作用！



## POST不是幂等的

HTTP GET只是要得到东西，它不会修改服务器上的任何内容。所以，根据定义（以及根据HTTP规范），GET是幂等的。它能执行多次，而且不会产生任何不好的副作用。

POST不是幂等的，POST体中的提交数据可能用于不可逆转的事务。所以使用doPost()功能时必须特别小心！

GET是幂等的。POST不是幂等的。

你要确保Web应用逻辑可以处理Diane遭遇的这种情况，也就是POST可能做了不只一次。



GET在HTTP  
1.1中始终被认为  
是幂等的……

……不过，你在考试中也许会看到这样一些代码，它们会用一种可能导致副作用的方式使用GET参数！换句话说，根据HTTP规范GET是幂等的。但是你完全可以在你的servlet中实现一个非幂等的doGet()方法。即使你对数据的处理可能导致副作用，客户的GET请求本身是幂等的。

一定要记住，HTTP GET方法和servlet中的doGet()方法是有区别的。

注意：“幂等”一词有多种不同的用法。对于HTTP/servlet这个词表示同一个请求可以做两次，而不会对服务器产生负面影响。我们使用“幂等”并不是说同样的请求总会得到同样的响应，也不是说一个请求没有副作用。

# 怎么确定浏览器发送的是GET还是POST请求？

**GET**

简单的超链接往往意味着GET。

```
<A HREF="http://www.wickedlysmart.com/index.html/">click here</A>
```

**POST**

如果明确地说method=“POST”，那么毫无疑问这就是一个POST。

```
<form method="POST" action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

用户点击“SUBMIT”按钮时，参数会放在POST请求体中发送。在这个示例中，只有一个参数，名为“color”，参数值是用户选择的<option>啤酒颜色（light, amber, brown或dark）。

如果在<form>中没有说method=“POST”，会怎么样呢？

这一次，这里没有method=“POST”

```
<form action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

如果表单中没有method=“POST”，现在用户点击SUBMIT时参数会怎么样呢？

# POST不是默认的！

如果没有在表单中指出method=“POST”，就会默认为HTTP GET请求。这说明，浏览器会把参数放在请求首部中发送，这还是一个小问题。因为如果到来一个GET请求，这意味着倘若你的servlet中只有doPost()而没有doGet()，在运行时就会遇到大麻烦！

如果这样做：

```
HTML表单中没有
"method=POST"
↓
<form action="SelectBeer.do">
```

然后这样：

```
public class BeerSelect extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        // 代码
    }
}
```

这个servlet中没有doGet()方法。

你就会得到这样的结果：

失败！如果HTML表单使用GET而不是POST，你的servlet类中就必须有doGet()。表单的默认方法是GET。

**问：** 如果我想让一个servlet同时支持GET和POST，怎么做呢？

**答：** 如果开发人员想同时支持这两个方法，通常会把逻辑放在doGet()中，如果有必要，再建立

```
doPost()方法委托到doGet()。
public void doPost(...)
throws ...{
    doGet(request, response);
}
```

# 发送和使用单个参数

## HTML 表单

```
<form method="POST" action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

这里有一个选择框，浏览器会发送其中一个选项，并作为一个名为“color”的参数放在请求体中。例如，“color=amber”。

## HTTP POST 请求

**POST** /advisor>SelectBeer.do HTTP/1.1

Host: www.wickedlysmart.com

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624

Netscape/7.1

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,\*/\*;q=0.1

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive: 300

Connection: keep-alive

记住，浏览器会生成这个请求，所以你不用担心怎么创建，只有服务器才会看到这个请求……

color=dark

## Servlet类

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException {
  String colorParam = request.getParameter("color");
  // 更多代码……
}
```

(在这个例子中，String colorParam的值为“dark”)

↑  
这与表单中的参数名匹配。

# 发送和使用两个参数

## HTML表单

```

<form method="POST" action="SelectBeerTaste.do">
    Select beer characteristics<p>
    COLOR:
    <select name="color" size="1">
        <option>light
        <option>amber
        <option>brown
        <option>dark
    </select>
    BODY:
    <select name="body" size="1">
        <option>light
        <option>medium
        <option>heavy
    </select>
    <center>
        <input type="SUBMIT">
    </center>
</form>

```

浏览器会发送这4个选项中的一个，作为参数“color”放在请求中。

浏览器会发送这3个选项中的一个，作为参数“body”放在请求中。

## HTTP POST请求

**POST** /advisor>SelectBeerTaste.do HTTP/1.1  
Host: www.wickedlysmart.com  
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624  
Netscape/7.1  
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,\*/\*;q=0.1  
Accept-Language: en-us,en;q=0.5  
Accept-Encoding: gzip,deflate  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7  
Keep-Alive: 300  
Connection: keep-alive

**color=dark&body=heavy**

现在POST请求有了这两个参数，用一个&分隔。

## Servlet类

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
                    throws IOException, ServletException
{
    String colorParam = request.getParameter("color");
    String bodyParam = request.getParameter("body");
    // 更多代码
}

```

现在String变量colorParam的值为“dark”，bodyParam的值为“heavy”。



单个参数可以有多个值！这说明需要一个返回数组的  
getParametersValues(), 而不是返回String的getParameters()。

有些表单输入类型可以有多个值，如复选框。这说明单个参数（例如，“sizes”）可能有多个值，主要取决于用户选中了多少个复选框。在下面的表单中，用户可以选择多种啤酒规格（比如说，他对所有规格都感兴趣）：

```
<form method=POST
action="SelectBeer.do">
Select beer characteristics<p>
Can Sizes: <p>
<input type=checkbox name=sizes value="12oz"> 12 oz.<br>
<input type=checkbox name=sizes value="16oz"> 16 oz.<br>
<input type=checkbox name=sizes value="22oz"> 22 oz.<br>
<br><br>

<center>
<input type="SUBMIT">
</center>

</form>
```

在你的代码中，就要使用getParametersValues()方法返回一个数组：

```
String one = request.getParameterValues("sizes")[0];
String [] sizes = request.getParameterValues("sizes");
```

如果你想看到数组中的所有内容，不管是为了好玩，还是为了测试，就可以使用：

```
String [] sizes = request.getParameterValues("sizes");
for(int x=0; x < sizes.length ; x++) {
    out.println("<br>sizes: " + sizes[x]);
}
```

(假设“out”是从响应得到的一个PrintWriter)。

# 除了参数，我还能从请求对象得到什么？

ServletRequest和HttpServletRequest接口提供了大量可以调用的方法，不过你不用把它们都记住。对你来说，确实应该好好看看javax.servlet.ServletRequest和javax.servlet.http.HttpServletRequest的所有API，但是这里只介绍你在实际中最有可能用到的一些方法（这也是考试中最有可能考到的方法）。

在实际中，你可能很幸运地用到请求API中15%的方法（不过也许你会认为这很不幸）。如果你不清楚如何使用各个方法，或者为什么使用这些方法，也不用着急。本书后面还会更详细地介绍其中一些方法。

## 客户的平台和浏览器信息

```
String client = request.getHeader("User-Agent");
```

## 与请求相关的cookie

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

## 与客户相关的会话（session）

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

## 请求的HTTP方法

```
String theMethod = request.getMethod();
```

## 请求的输入流

```
InputStream input = request.getInputStream();
```

**ServletRequest** 接口  
(javax.servlet.ServletRequest)

<<interface>>  
**ServletRequest**

```
getAttribute(String)
getContentLength()
getInputStream()
getLocalPort()
getRemotePort()
getServerPort()
getParameter(String)
getParameterValues(String)
getParameterNames()
// 更多方法.....
```

**HttpServletRequest** 接口  
(javax.servlet.http.HttpServletRequest)

<<interface>>  
**HttpServletRequest**

```
getContextPath()
getCookies()
getHeader(String)
getIntHeader(String)
getMethod()
getQueryString()
getSession()
// 更多方法.....
```

**问：**为什么需要从请求得到一个InputStream呢？

**答：**如果是一个GET请求，那么除了首部信息以外就没有别的了。换句话说，我们不用操心请求体。不过……HTTP POST请求有所不同，它有体信息。大多数情况下，我们关心的只是用request.getParameter()抽出参数值（例如“color=dark”），但是这些值可能很大。还有可能创建一个servlet来处理计算机驱动的请求，其中请求体包含要处理的文本或二进制内容。在这种情况下，可以使用getReader或getInputStream()方法。这些流只包含HTTP请求的体而不包含首部。

**问：**getHeader()和getIntHeader()有什么区别？在我看来，首部都是String吧！既然getIntHeader()方法取一个表示首部名的String，那这个方法名里的int是什么意思？

**答：**首部有一个名（如“User-Agent”或“Host”），还有一个值（如“Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1”或“www.wickedlysmart.com”）。从首部得来的值总是String，但是对于有些首部，这个String表示的是一个数。“Content-Length”首部就会返回消息体的字节数。例如，“Max-Forwards” HTTP首部会返回一个整数，指示请求可以经过的最大路由跳数（如果你想跟踪一个请求，看看它是不是陷入一个循环，就可以使用这个首部）。

可以使用getHeader()来得到“Max-Forwards”首部的值：

```
String forwards = request.getHeader("Max-Forwards");
int forwardsNum = Integer.parseInt(forwards);
```

这样是可以的。但是如果你已经知道首部的值表示一个整数，就可以使用getIntHeader()作为便利方法，这样就不用再多做一步把String解析为一个int，而是可以直接得到int：

```
int forwardsNum = request.getIntHeader("Max-Forwards");
```



getServerPort()、getLocalPort()和getRemotePort()很容易搞混！

getServerPort()含义应该很清楚……不过，这有可能与getLocalPort()混淆。所以我们先来解释相对容易的方法：getRemotePort()。首先，你可能会问“对谁而言是远程的？”在这种情况下，由于是服务器在问，所以客户是远程的。既然客户对服务器是远程的，所以getRemotePort()是指“得到客户的端口”。也就是说要得到发出请求的客户的端口号。要记住：对于一个servlet，远程就意味着客户。

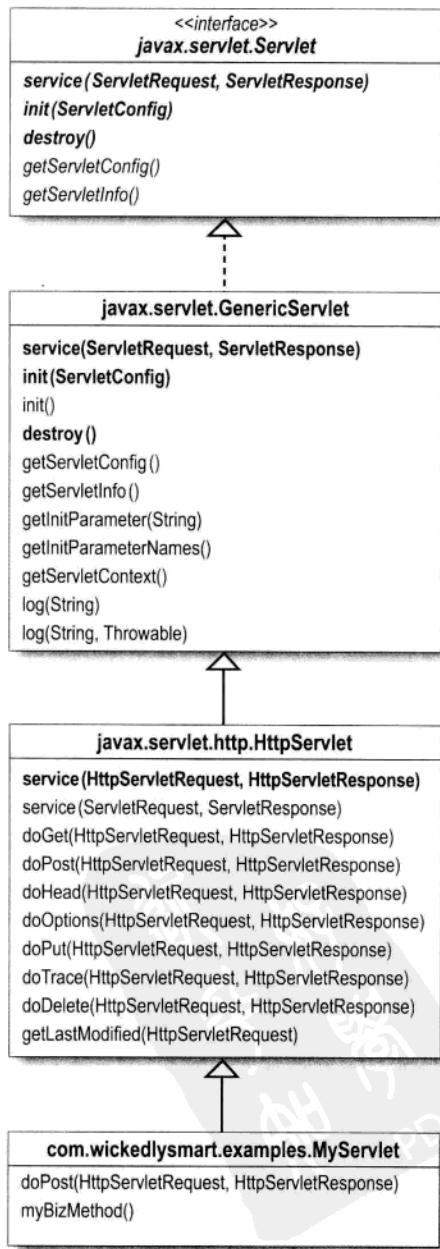
getLocalPort()和getServerPort()的差别很微妙——getServerPort()说，“请求原来发送到哪个端口？”，getLocalPort()则是说“请求最后发送到哪个端口？”。不错，二者确实有区别，因为尽管请求要发送到一个端口（服务器所监听的端口），但是服务器会为每个线程找一个不同的本地端口，这样一来，一个应用就能同时处理多个客户了。

# 复习：servlet生命周期和API

## 要点



- 容器要加载类、调用servlet的无参数构造函数，并调用servlet的init()方法，从而初始化servlet。
- init()方法（开发人员可以覆盖）在servlet一生中只调用一次，往往在servlet为客户请求提供服务之前调用。
- init()方法使servlet可以访问ServletConfig和ServletContext对象，servlet需要从这些对象得到有关servlet配置和Web应用的信息。
- 容器通过调用servlet的destroy()方法来结束servlet的生命。
- servlet一生的大多数时间都是在为某个客户请求运行service()方法。
- 对servlet的每个请求都在一个单独的线程中运行！任何特定servlet类都只有一个实例。
- 你的servlet一般都会扩展javax.servlet.http.HttpServlet，并由此继承service()方法的一个实现，它取一个HttpServletRequest和一个HttpServletResponse作为参数。
- HttpServlet扩展了javax.servlet.GenericServlet，这是一个抽象类，实现了大多数基本servlet方法。
- GenericServlet实现了Servlet接口。
- Servlet相关的类（除了与JSP有关的类）都在以下两个包中：javax.servlet 或 javax.servlet.http。
- 可以覆盖init()方法，而且必须覆盖一个服务方法（doGet()、doPost()等）。



# 复习：HTTP和HttpServletRequest

## 要点

- HttpServletRequest的doGet()和doPost()方法取一个HttpServletRequest和一个HttpServletResponse作为参数。
- service()方法根据HTTP请求的HTTP方法（GET、POST等）来确定运行doGet()还是doPost()。
- POST请求有一个体；GET请求没有，不过GET请求可以把请求参数追加到请求URL的后面（有时称为“查询串”）。
- GET请求本质上讲（根据HTTP规范）是幂等的。它们应当能多次运行而不会对服务器产生任何副作用。GET请求不应修改服务器上的任何东西。但是你也可以写一个非幂等的doGet()方法（不过这是很糟糕的做法）。
- POST本质上讲不是幂等的，所以要由你来适当地设计和编写代码，如果客户错误地把一个请求发送了两次，你也能正确地加以处理。
- 如果HTML表单没有明确地指出“method=POST”，请求就会作为一个GET请求发送，而不是POST请求。如果你的servlet中没有doGet()，这个请求就会失败。
- 可以用getParameter(“paramname”)方法从请求得到参数。返回值总是一个String。
- 如果对应一个给定的参数名有多个参数值，要使用getParameterValues(“paramname”)方法来返回一个String数组。
- 从请求对象还可以得到其他东西，包括首部、cookie、会话、查询串和输入流。



**ServletRequest 接口**  
(javax.servlet.ServletRequest)

<<interface>>	
<b>ServletRequest</b>	
getAttribute(String)	
getContentLength()	
getInputStream()	
getLocalPort()	
getRemotePort()	
getServerPort()	
getParameter(String)	
getParameterValues(String)	
getParameterNames()	
//更多方法……	

**HttpServletRequest 接口**  
(javax.servlet.http.HttpServletRequest)

<<interface>>	
<b>HttpServletRequest</b>	
getContextPath()	
getCookies()	
getHeader(String)	
getIntHeader(String)	
getMethod()	
getQueryString()	
getSession()	
//更多方法……	

## 已经了解了请求…… 下面来看看响应

响应要返回给客户。这是浏览器得到、解析并呈现给用户的东西。一般地，你会使用响应对象得到一个输出流（通常是一个Writer），并使用这个流写出HTML（或其他类型的内容），返回给客户。不过，响应对象除了I/O输出外还有其他方法，下面更详细地分析其中一些方法。

**ServletResponse** 接口  
(javax.servlet.ServletResponse)

```
<<interface>>
ServletResponse
getBufferSize()
setContentType()
getOutputStream()
getWriter()
setContentLength()
// 更多方法……
```

这些是最常用的方法。

**HttpServletResponse** 接口  
(javax.servlet.http.HttpServletResponse)

```
<<interface>>
HttpServletResponse
addCookie()
addHeader()
encodeURL()
sendError()
setStatus()
sendRedirect()
// 更多方法……
```

有时你还会用到这些方法……

大多数情况下，使用响应只是为了向客户发回数据。

会对响应调用两个方法：  
setContentType() 和 getWriter()。

在此之后，只需要完成I/O  
将HTML（或其他内容）写  
至流。

不过，你也可以使用响应设  
置其他首部、发送错误，以  
及增加cookie。

稍等……我想不能从servlet发送HTML吧，因为把它格式化输出到输出流太难看了……



## 使用响应完成I/O

对，是应该使用JSP，而不是从servlet把HTML发回到输出流。格式化HTML从而通过输出流的println()方法打印出来，这种方法确实不好。

但这并不是说绝对不能从servlet处理输出流。

为什么呢？

(1) 你的提供商可能不支持JSP。还有很多比较老的服务器和容器，它们只支持servlet而不支持JSP，所以只能使用这种方法。

(2) 出于其他原因可能没有办法使用JSP，比如你有一个很愚蠢的上司，不让你用JSP，原因只是1998年他姐夫告诉他JSP很不好。

(3) 是谁说只能在响应中返回HTML？你还可以向客户返回不是HTML的其他东西。对此可能输出流更适合。

翻到下一页来看一个例子……

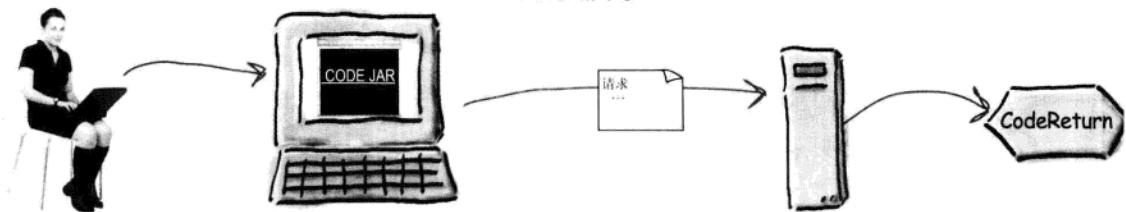
## 假设你希望为客户发送一个JAR……

假设你创建了一个下载页面，客户可以从这个页面得到JAR文件的代码。这里不是发回一个HTML页面，响应中包含的是表示JAR的字节。你要读取JAR文件的字节，然后写至响应的输出流。

- ① Diane在学一本介绍servlet和JSP的书，非常想下载这本书的JAR代码。她导航到这本书的网站，点击“code jar”链接，这指向一个名为“Code.do”的servlet。

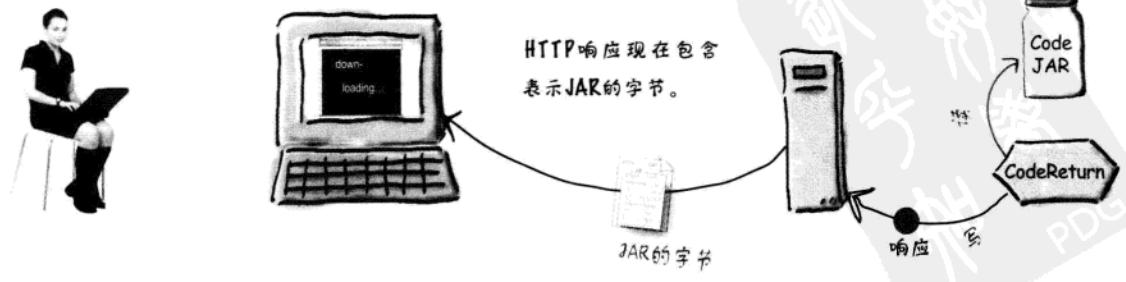
浏览器用所请求servlet的名字（“Code.do”）向服务器发送一个HTTP请求。

容器把请求发送到CodeReturn servlet（映射到DD中名“Code.do”），进行处理



- ② JAR开始下载到客户的机器上。  
Diane很高兴。

CodeReturn servlet得到JAR的字节，然后从响应得到一个输出流，并表示JAR的字节写到输出流中。



# 下载JAR的Servlet代码

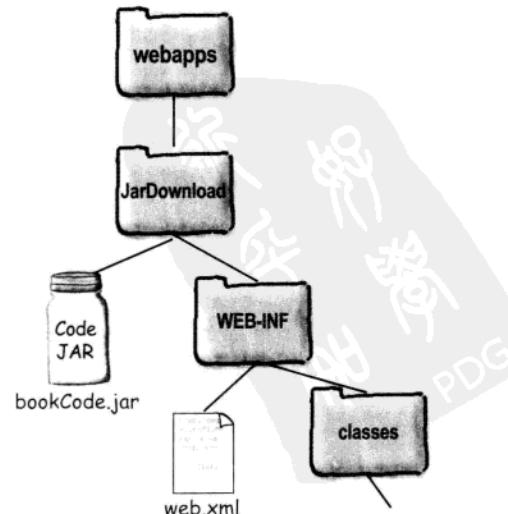
```
// a bunch of imports here

public class CodeReturn extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("application/jar");
        // 我们希望浏览器看出来这是一个
        // JAR, 而不是HTML, 所以把内容类
        // 型设置为 "application/jar"
        ServletContext ctx = getServletContext();
        InputStream is = ctx.getResourceAsStream("/bookCode.jar");
        int read = 0;
        byte[] bytes = new byte[1024];
        OutputStream os = response.getOutputStream();
        // 这表示 "为名为bookCode.jar的
        // 资源给我一个输入流"
        while ((read = is.read(bytes)) != -1) {
            os.write(bytes, 0, read);
        }
        os.flush();
        os.close();
    }
}
```

*there are no  
Dumb Questions*

**问：**“bookCode.jar” JAR文件放在哪里？换句话说，`getResourceAsStream()`方法要到哪里去找这个文件？你是怎么处理路径的？

**答：**`getResourceAsStream()`要求首先有一个斜线（“/”），这表示Web应用的根。由于Web应用名为JarDownload，因此目录结构应该就是右图中的样子。JarDownload目录在webapps目录下（所有其他Web应用的目录也是这样），然后在JarDownload目录中我们放了一个WEB-INF目录，代码JAR本身也放在JarDownload目录中。所以“bookCode.jar”位于JarDownload Web应用的根一级（别担心，在有关“部署”的一章中，我们还会非常深入地讨论有关部署目录结构的细节）。

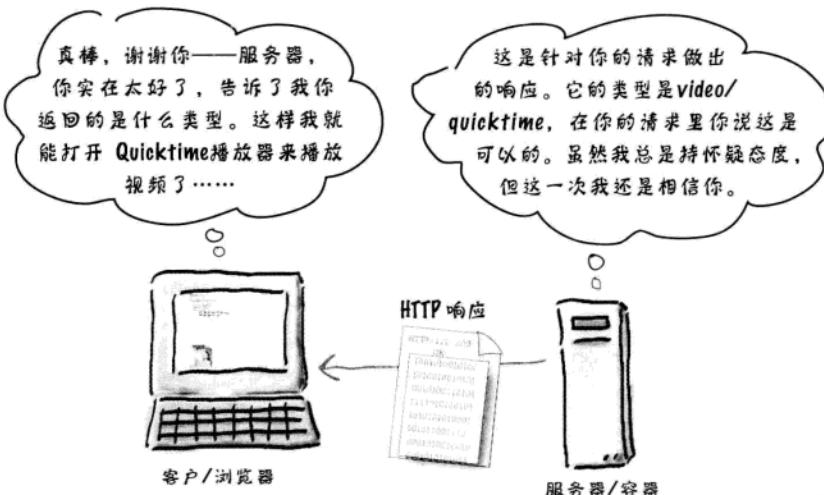


# 哇呜，内容类型是做什么的？

你可能不明白下面这一行：

```
response.setContentType("application/jar");
```

或者说，你至少应该明白这行代码。你必须告诉浏览器你要发回些什么，这样浏览器才能有正确的“举止”：可能启动一个“辅助”应用，如PDF阅读器或视频播放器；也可能向用户呈现HTML，或者把响应的字节保存为一个下载文件等。既然你想知道，可以说，谈到内容类型的时候，我们指的就是MIME类型。内容类型是HTTP响应中必须有的一个HTTP首部。



**常用MIME类型：**

- `text/html`
- `application/pdf`
- `video/quicktime`
- `application/java`
- `image/jpeg`
- `application/jar`
- `application/octet-stream`
- `application/x-zip`

## 不需要你记住一大堆的内容类型

- 你应该知道 `setContentType()` 是做什么的，应该如何使用，但是除了 `text/html` 之外，甚至连最常用的内容类型也无需记住。你只要知道有关 `setContentType()` 的一些常识……例如，写到响应输出流之后再改变内容类型没有任何好处。嗯，这说明，不能先设置一个内容类型，写些东西，再改变内容类型，再写点别的东西。不过，想想看，浏览器是怎么处理的？它一次只从响应处理一种类型的内容。



要确保一切都工作正常，有一个最佳实践（在有些情况下，这甚至要算是一个需求）：总是先调用 `setContentType()`，然后再调用获得输出流的方法 (`getWriter()` 或 `getOutputStream()`)。这样就能保证不会遭遇内容类型和输出流之间的冲突。

there are no  
Dumb Questions

**问：**为什么必须设置内容类型？服务器不能从文件的扩展名看出来吗？

**答：**如果是静态内容，大多数服务器确实能看出它的类型。例如，在Apache中，可以建立MIME类型，把一个特定的文件扩展名（.txt, .jar等）映射到一个特定的内容类型。Apache会使用这个映射来设置HTTP首部中的内容类型。不过，我们说的是servlet中发生的情况，这里根本没有文件！要由你来发回响应，容器根本不知道你在发什么。

**问：**但是最后这个例子呢？你读的是一个特定的JAR文件呀，容器看不出你在读一个JAR吗？

**答：**看不出。我们在servlet中只是读取文件的字节（只不过这个文件刚好是一个JAR文件），然后把这些字节写到输出流。我们读这些字节时容器不知道我们要做什么。它只知道我们在读某种类型的东西，而且会在响应中写完全不同的另外一些东西。

**问：**我怎么才能知道有哪些常用的内容类型？

**答：**在Google上搜索。这是说真的。新的MIME类型总在不断增加，但是在网上可以很容易地找到MIME类型列表。你也可以查看浏览器首选项，其中列出了你的浏览器配置的一些MIME类型，另外还可以检查Web服务器配置文件。同样地，不要担心考试会考这些，另外即使在实际中也不用为此太操心。

**问：**等一等……为什么你使用一个servlet发回JAR呢？让Web服务器把它作为一个资源返回不就行了吗？换句话说，为什么不让用户点击一个指向JAR的链接，而要让链接指向一个servlet呢？难道不能把服务器配置为直接返回JAR而不要经过servlet吗？

**答：**不错，这是一个很好的问题。你确实可以这样配置Web服务器，让用户点击一个HTML链接，指向服务器上的某个资源，比如说JAR文件（就像其他静态资源一样，如JPEG和文本文件），服务器只是把它放在响应中返回。

不过……我们认为，在发回输出流之前你可能还想在servlet里做点别的事情。例如，你可能想在servlet中实现一些逻辑来确定要发送哪个JAR。或者，也许你要发回的是实时创建的字节。想像一下，如果有这样一个系统，你要从用户得到输入参数，然后使用这些参数动态地生成一个声音，发送回去。原来是没有这个声音的。换句话说，声音并没有作为一个文件放在服务器上。你要建立这样一个声音，然后把它放在响应中返回。

你说的也对，如果只是发回服务器上的一个JAR，前面这个例子确实有些麻烦，但是再想想看……充分发挥你的想像力，考虑一下还有哪些原因可以说明有必要建立一个servlet。也许原因很简单，只是要在servlet中放一些代码，以便在发回JAR的同时，能在数据库中写一些有关这个特定用户的信息。也可能必须查看用户是否允许下载这个JAR，为此可能需要先从数据库读取一些信息才能决定。

# 对于输出，你有两个选择： 字符还是字节

这只是一个普通的java.io，不过ServletResponse接口只提供了两个流可供选择：ServletOutputStream用于输出字节，PrintWriter用于输出字符数据。

## » PrintWriter

例子：

```
PrintWriter writer = response.getWriter();

writer.println("some text and HTML");
```

用于：

把文本数据打印到一个字符流。尽管也可以把字符数据写至 OutputStream，但PrintWriter流专门设计用于处理字符数据。

## » OutputStream

例子：

```
ServletOutputStream out = response.getOutputStream();

out.write(aByteArray);
```

用于：

写其他的任何内容！

再说一句：PrintWriter实际上“包装”了ServletOutputStream。也就是说，PrintWriter有ServletOutputStream的一个引用，而且会把调用委托给ServletOutputStream。返回给客户的输出流只有一个，但是PrintWriter会“装饰”这个流，为它增加更高层的“字符友好”方法（即连于处理字符的方法）。



**必须记住这些方法**

考试要求你必须知道这些方法，而且这有些难度。注意，如果写至ServletOutputStream，就需要调用write()，如果写至PrintWriter，就要……使用println()！可能很自然地会想到要写(write)到一个书写器(writer)，但不能这样做。如果你用过java.io，应该很清楚这一点。但是如果你没有用过，就要记住：

println()写至PrintWriter

write()写至ServletOutputStream

一定要记住获得流或书写器的方法名，都是把返回类型的第一个词去掉：

ServletOutputStream

response.getOutputStream()

PrintWriter

response.getWriter()

看到下面这些名字，你应该知道它们是错误的：

getPrintWriter()

getResponseBody()

getStream()

getOutputWriter()

不存在这样一些方法！

# 可以设置响应首部， 也可以增加响应首部

你可能想知道这又有什么区别。先考虑一下，再来看下面这个练习。

## 画线，把方法调用与其行为匹配

```
response.setHeader("foo", "bar");
response.addHeader("foo", "bar");
response.setIntHeader("foo", 42);
```

在`HttpResponse`方法到该方法的行为之间画线。我们已经完成了最明显的一组。

为响应增加一个新首部和值，或者向一个现有的首部增加另一个值。

这是一个便利方法，用提供的整数值替换现有首部的值，或者向响应增加一个新首部和值。

如果响应中已经有同名的首部，则用这个值替换原来的值。否则，向响应增加一个新首部和值。

如果放在一起，就能很清楚地看出它们的差别。

不过如果要参加考试，你必须把它们都牢牢记住，这样等下星期二有人问你，“哪个响应方法能让我向一个现有的首部增加一个值？”你就能毫不迟疑地回答，“`addHeader`，它需要名和值这两个String作为参数”。要这么流利才行。

如果响应中还没有首部（方法的第一个参数），`setHeader()`和`addHeader()`就会增加一个首部和相应的值。二者的区别是，如果有这样一个首部，前者将设置一个值，而后者将增加一个值。在这种情况下：

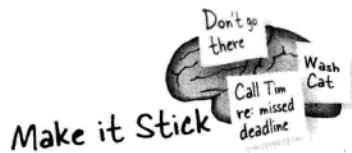
`setHeader()`会覆盖现有的值

`addHeader()`会增加另外一个值

调用`setContentType(“text/html”)`时，就是在设置一个首部，相当于：

```
setHeader(“content-type”, “text/html”);
```

那么这又有什么区别呢？没有区别……，只要你没把“`content-type`”首部敲错就行。如果你把首部名敲错了，`setHeader()`方法不会指出出来，它只是认为你要增加一个新的首部。不过，以后就会出错，因为你没有正确地设置响应的内容类型！



一个响应有首部，  
还有些负载在内部，  
没有一个首部有多值，  
此时就该用`setHeader()`。

(与`addHeader()`刚好相反，明白了吗？)

(如果你能背诵这首诗，不急不慢，可以录成mp3发给我们，第一个发来的人将得到一件特别的T恤。)

## 但是有时你可能不想自己处理响应……

可以选择让别人来为你的请求处理响应。可以把请求重定向到一个完全不同的URL，或者可以把请求分派给web应用的另一个组件（通常是一个JSP）。

### 重定向

- ① 客户向浏览器地址栏键入一个URL……



- ② 请求到达服务器/容器。



- ③ servlet决定这个请求应重定向到另一个完全不同的URL。



- ⑥ 浏览器得到响应，发现了“301”状态码，并寻找“Location”首部。

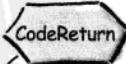


- ⑤

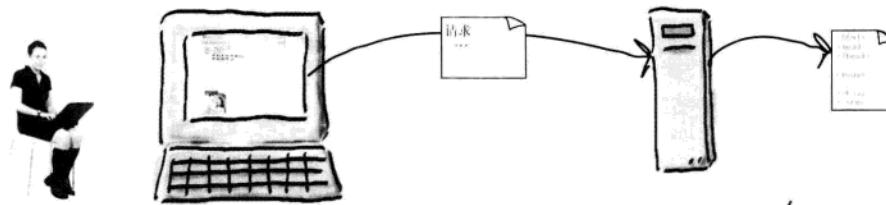
HTTP响应有一个状态码“301”，还有一个“Location”首部，这个首部值是一个URL。



- ④ servlet在响应上调用sendRedirect(`aString`)，的工作就完成了。



⑦ 浏览器使用前一个响应中“Location”首部的值（URL）建立一个新的请求。用户可能注意到浏览器地址栏上的URL改变了……



⑧ 这个请求没有什么特别的地方，尽管它是由一个重定向触发的。

⑨ 服务器得到所请求URL上的资源。这里也没有特别的地方。

怎么到这儿来了？

⑪ 浏览器显示这个新页面。这让用户很奇怪。

⑩ 这个HTTP响应与其他响应没有两样……只不过它不是来自客户键入的位置。



# Servlet重定向让浏览器完成工作

重定向使得servlet完全卸下担子。确定无法完成工作之后，servlet只是调用sendRedirect()方法：

```
if (worksForMe) {
    // handle the request
} else {
    response.sendRedirect("http://www.oreilly.com");
}
```

↑  
希望浏览器用这个URL处理请求。  
客户会看到这个URL。

## 在sendRedirect()中使用相对URLs

可以使用相对URL作为sendRedirect()的参数，而不是指定完整的“http://www……”。相对URL有两种类型：前面有斜线(“/”)和没有斜线。

假设客户原来键入的是：

`http://www.wickedlysmart.com/myApp/cool/bar.do`

请求到达名为“bar.do”的servlet时，这个servlet会基于一个相对URL来调用sendRedirect()，这个相对URL没有用斜线开头：

`sendRedirect("foo/stuff.html");`

容器会相对于原先的请求URL建立完整的URL（需要把它放在HTTP响应的“Location”首部中）：

`http://www.wickedlysmart.com/myApp/cool/foo/stuff.html`

但是如果sendRedirect()的参数确实以一个斜线开头：

`sendRedirect("/foo/stuff.html");`

容器知道原来的请求URL从myApp/cool路径开始，所以如果沒有加斜线，就会把这一部分放在“foo/stuff.html”的前面。

开始位置上的斜线意味着“相对于这个Web容器的根”。

容器就会相对于Web应用本身建立完整的URL，而不是相对于请求原来的URL。所以新的URL是：

`http://www.wickedlysmart.com/foo/stuff.html`

↑  
“foo”是一个Web应用，它不同于“myApp”Web应用。



不能在写到响应之后再  
调用sendRedirect()！

当心！

这可能很显然，但是这是一条必须遵守的定律，所以有必要特别明确。

通过查看API中的sendRedirect()，可以看到，如果你想在“响应已经提交”之后再调用这个方法，它就会抛出一个IllegalStateException异常。

IllegalStateException的意思是指，响应已经发出，这里“提交”的意思是指，响应已经发出，这说明数据已经刷新输出到流中。在实际应用中，这意味着，不能写到响应之后再调用sendRedirect()！

不过，有些爱钻牛角尖的人会告诉你，从理论上讲，也可以把数据写至流中而不刷新输出，而且之后sendRedirect()也不会抛出异常。但是这是很愚蠢的，所以我们不讨论这个问题（这里只是点到为止，不会多讲了……）。

在你的servlet中，必须要做一个决定！要么在你的servlet中，必须要做一个决定！要么调用sendRedirect()让别人处理请求，要么调用sendRedirect()来处理请求。

（顺便说一句，“一旦提交，就没有后悔药了”，这种思想也适用于设置首部、cookie、状态码、内容类型等）

sendRedirect()取一个String，而不是URL对象！

对，它要的是一个String，这个String实际上就是一个URL。但问题是，sendRedirect()不能取一个URL类型的对象作为参数。你要为它传递一个String，可以是一个完整的URL，也可以是一个相对URL。如果容器无法根据一个相对URL建立完整的URL，它就会抛出一个IllegalStateException。

关键在于，要记住下面这样是不对的：

```
sendRedirect(new URL("http://www.oreilly.com"));
```

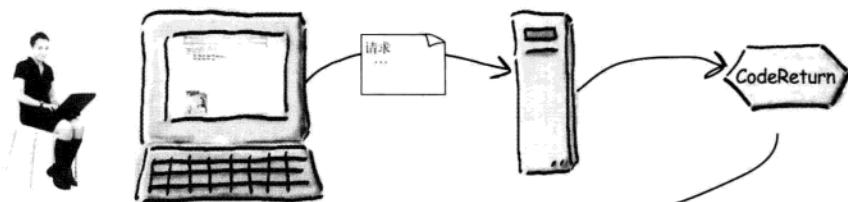
不行！看上去好像是对的，但是其实大错特错了。sendRedirect()要取一个String参数。只能是String。

# 请求分派是在服务器端做工作

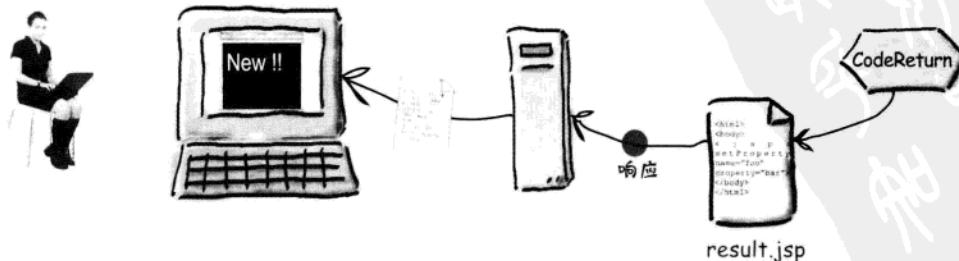
这就是重定向和请求分派之间的最大差别，重定向让客户来完成工作，而请求分派要求服务器上的某某来完成任务。所以，要记住，重定向 = 客户，请求分派 = 服务器。下一章会更多地介绍对请求分派，不过，这两页先让你对重点有一个大致的了解。

## 请求分派

- ① 用户在浏览器的地址栏键入一个servlet的URL……
- ② 请求到达服务器/容器。
- ③ servlet 决定这个请求应该交给Web应用的另一部分（在这里，就是一个JSP）。



- ⑤ 浏览器以正常方式得到响应，把它显示给用户。由于浏览器的地址栏没有变化，所以用户不知道是由JSP生成的响应。



# 重定向与请求分派

我没时间！你为什么不问问Barney呢？没准他有时间做。

重定向



servlet完成重定向时，就像让客户再给别人打电话一样。在这种情况下，客户是浏览器，而不是用户。浏览器代表用户再打一个电话。因为原来它请求的servlet说，“抱歉，你再打给这个人看看……”。

用户会在浏览器地址栏看到新的URL。

嘿，Kari，我是Dan……我想让你帮一位客户。我会把怎么联系他的详细资料转发给你，不过希望你现在就接管。

对，我知道，你自己也很忙……嗯，我很清楚试图在MVC中的角色……不，我想我找不到像你那样的JSP了……什么？我沒听清楚，你要崩溃了？……真抱歉，受不了了？……丢包……

请求分派

servlet完成请求分派时，就像要求一位同事接管一个客户的工作一样。由这个同事最后为客户做出响应，但是客户并不关心是谁的响应，只要有响应就行。

用户根本不知道是别人接管了工作，因为浏览器地址栏上的URL没有任何变化。



# 复习： HttpServletResponse

## 要点

- 使用响应向客户发回数据。
- 对响应对象（`HttpServletResponse`）调用的最常用的方法是 `setContentType()` 和 `getWriter()`。
- 要当心——很多开发人员都认为应该是 `getPrintWriter()`，但实际上得到书写器的方法是 `getWriter()`。
- 利用 `getWriter()` 方法可以完成字符 I/O，向流写入 HTML（或其他内容）。
- 还可以使用响应来设置首部、发送错误，以及增加 cookie。
- 在实际中，可能会使用 JSP 发送大多数 HTML 响应，但仍有可能使用一个响应流向客户发送二进制数据（如 JAR 文件）。
- 要得到二进制流，需要在响应上调用 `getOutputStream()`。
- `setContentType()` 方法告诉浏览器如何处理随响应到来的数据。常见的内容类型为 “text/html”、“application/pdf” 和 “image/jpeg”。
- 不用记住内容类型（也称为 MIME 类型）。
- 可以使用 `addHeader()` 或 `setHeader()` 设置响应首部。二者的区别是这个首部是否已经是响应的一部分。如果是，`setHeader()` 会替换原来的值，而 `addHeader` 会向现有的响应增加另一个值。如果响应中原来没有这个首部，`setHeader()` 和 `addHeader()` 的表现完全一样。
- 如果你不想要对一个请求做出响应，可以把请求重定向到另一个 URL。浏览器会负责把新请求发送到你提供的 URL。
- 要重定向一个请求，需要在响应上调用 `sendRedirect(aStringURL)`。
- 不能在响应已经提交之后才调用 `sendRedirect()`！换句话说，如果已经向流中写了东西，再想重定向则为时已晚。
- 请求重定向与请求分派完全是两码事。请求分派（下一章将详细介绍）在服务器端发生，而重定向在客户端进行。请求分派把请求传递给服务器上的另一个组件（通常在同一个 Web 应用中）。请求重定向只是告诉浏览器去访问另一个 URL。

**ServletResponse 接口**  
(`javax.servlet.ServletResponse`)

```
<<interface>>
ServletResponse
getBufferSize()
setContentType()
getOutputStream()
getWriter()
setContentLength()
// 更多方法……
```

**HttpServletResponse 接口**  
(`javax.servlet.http.HttpServletResponse`)

```
<<interface>>
HttpServletResponse
addCookie()
addHeader()
encodeURL()
sendError()
setStatus()
sendRedirect()
// 更多方法……
```



## 第4章 模拟测验

1 服务方法（如**doPost()**）的servlet代码如何从请求获得“User-Agent”首部的值（选出所有正确的答案）？

- A. `String userAgent =  
request.getParameter("User-Agent");`
- B. `String userAgent = request.getHeader("User-Agent");`
- C. `String userAgent =  
request.getRequestHeader("Mozilla");`
- D. `String userAgent =  
getServletContext().getInitParameter("User-Agent");`

2 可以用哪些HTTP方法向客户显示服务器正在接收的内容（选出所有正确的答案）？

- A. GET
- B. PUT
- C. TRACE
- D. RETURN
- E. OPTIONS

3 `HttpServletResponse`的哪些方法用于将一个HTTP请求重定向到另一个URL？

- A. `sendURL()`
- B. `redirectURL()`
- C. `redirectHttp()`
- D. `sendRedirect()`
- E. `getRequestDispatcher()`

---

4 哪些HTTP方法不是幂等的（选出所有正确的答案）？

- A. GET
- B. POST
- C. HEAD
- D. PUT

---

5 假设**req**是一个**HttpServletRequest**，以下哪些代码可以得到一个二进制输入流（选出所有正确的答案）？

- A. `BinaryInputStream s = req.getInputStream();`
- B. `ServletInputStream s = req.getInputStream();`
- C. `BinaryInputStream s = req.getBinaryStream();`
- D. `ServletInputStream s = req.getBinaryStream();`

---

6 如何在**HttpServletResponse**对象中设置一个名为“CONTENT-LENGTH”的首部（选出所有正确的答案）？

- A. `response.setHeader(CONTENT-LENGTH, "1024");`
- B. `response.setHeader("CONTENT-LENGTH", "1024");`
- C. `response.setStatus(1024);`
- D. `response.setHeader("CONTENT-LENGTH", 1024);`

---

7 以下哪个代码段会得到一个二进制流，用于向**HttpServletResponse**写一个图像或其他二进制类型的内容。

- A. `java.io.PrintWriter out = response.getWriter();`
- B. `ServletOutputStream out = response.getOutputStream();`
- C. `java.io.PrintWriter out =  
new PrintWriter(response.getWriter());`
- D. `ServletOutputStream out = response.getBinaryStream();`

8 servlet用哪些方法来处理来自客户的表单数据（选出所有正确的答案）？

- A. `HttpServlet.doHead()`
- B. `HttpServlet.doPost()`
- C. `HttpServlet.doForm()`
- D. `ServletRequest doGet()`
- E. `ServletRequest doPost()`
- F. `ServletRequest doForm()`

9 以下哪些方法在`HttpServletRequest`中声明，而不是在`ServletRequest`中声明（选出所有正确的答案）？

- A. `getMethod()`
- B. `getHeader()`
- C. `getCookies()`
- D. `getInputStream()`
- E. `getParameterNames()`

10 servlet开发人员在扩展`HttpServlet`时如何处理`HttpServlet`的`service()`方法（选出所有正确的答案）？

- A. 大多数情况下都应当覆盖`service()`方法。
- B. 应当从`doGet()`或`doPost()`调用`service()`方法。
- C. 应当从`init()`方法调用`service()`方法。
- D. 至少应当覆盖一个`doXXX()`方法（如`doPost()`）。



## 第4章 模拟测验答案

1 服务方法（如`doPost()`）的servlet代码如何从请求获得“User-Agent”首部的值 (APJ) (选出所有正确的答案) ?

- A. `String userAgent = request.getParameter("User-Agent");`
- B. `String userAgent = request.getHeader("User-Agent");`
- C. `String userAgent = request.getRequestHeader("Mozilla");`
- D. `String userAgent = getServletContext().getInitParameter("User-Agent");`

B是正确的。方法调用，将首部名作为一个String参数传入。

2 可以用哪些HTTP方法向客户显示服务器正在接收的内容 (选出所有正确的答案) ? (HF 4. HTTP方法)

- A. GET
- B. PUT
- C. TRACE
- D. RETURN
- E. OPTIONS

这个方法通常用于调试除错，而不用于生产环境。

3 `HttpServletResponse`的哪些方法用于将一个HTTP请求重定向到另一个 (APJ) URL?

- A. `sendURL()`
- B. `redirectURL()`
- C. `redirectHttp()`
- D. `sendRedirect()`
- E. `getRequestDispatcher()`

D是对的，在所列的方法中，这也是唯一的一个`HttpServletResponse`方法。

4

哪些HTTP方法不是幂等的（选出所有正确的答案）？

(HF 4. 幂等请求)

- A. GET  
 B. POST  
 C. HEAD  
 D. PUT

根据设计，POST要传递请求来更新服务器的状态。一般来说，同一个更新不应完成多次。

5

假设**req**是一个**HttpServletRequest**，以下哪些代码可以得到一个二进制输入流（选出所有正确的答案）？

(API)

- A. **BinaryInputStream s = req.getInputStream();**  
 B. **ServletInputStream s = req.getInputStream();**  
 C. **BinaryInputStream s = req.getBinaryStream();**  
 D. **ServletInputStream s = req.getBinaryStream();**

B是对的，它指定了正确的方法和正确的返回类型。

6

如何在**HttpServletResponse**对象中设置一个名为“CONTENT-LENGTH”的首部（选出所有正确的答案）？

(API)

- A. **response.setHeader(CONTENT-LENGTH,1024);**  
 B. **response.setHeader(CONTENT-LENGTH,1024);**  
 C. **response.setStatus(1024);**  
 D. **response.setHeader (CONTENT-LENGTH,1024);**

B是对的，它用两个String参数来设置一个HTTP首部，一个参数表示首部名；另一个表示首部值。

7

以下哪个代码段会得到一个二进制流，用于向**HttpServletResponse**写一个图像或其他二进制类型的内容。

(API)

- A. **java.io.PrintWriter out = response.getWriter();**  
 B. **ServletOutputStream out = response.getOutputStream();**  
 C. **java.io.PrintWriter out =  
           new PrintWriter(response.getWriter());**  
 D. **ServletOutputStream out = response.getBinaryStream();**

A不对，因为它使用了一个面向字符的 PrintWriter。

8 servlet用哪些方法来处理来自客户的表单数据（选出所有正确的答案）？ (APJ)

- A. `HttpServlet.doHead()`
- B. `HttpServlet.doPost()`
- C. `HttpServlet.doForm()`
- D. `ServletRequest doGet()`
- E. `ServletRequest doPost()`
- F. `ServletRequest doForm()`

C~F不对，因为这些方法根本不存在。

9 以下哪些方法在`HttpServletRequest`中声明，而不是在`ServletRequest`中声明 (APJ)  
明（选出所有正确的答案）？

- A. `getMethod()` A、B和C都与HTTP请求的某个部分有关。
- B. `getHeader()`
- C. `getCookies()`
- D. `getInputStream()`
- E. `getParameterNames()`

10 servlet开发人员在扩展`HttpServlet`时如何处理`HttpServlet`的`service()`方法 (APJ)  
（选出所有正确的答案）？

- A. 大多数情况下都应当覆盖`service()`方法。
- B. 应当从`doGet()`或`doPost()`调用`service()`方法。
- C. 应当从`init()`方法调用`service()`方法。
- D. 至少应当覆盖一个`doXXX()`方法（如`doPost()`）。 D是正确的，开发人员一般只关心`doGet()`和`doPost()`方法就行了。

## 5 属性和监听者

# 作为Web应用



没有servlet能独立存在。在当前的现代Web应用中，许多组件都是在一起协作共同完成一个目标。你会有模型、控制器和视图；会用到参数和属性；你还有一些辅助类。但是怎么把这些部分组织在一起呢？怎么让这些组件共享信息？如何隐藏信息？怎么让信息做到线程安全？能不能得出答案，这决定着你能不能活命。所以，学这一章时，一定要精神一点，多多地喝茶，但是别喝太刺激的东西。

# OBJECTIVES

## Web容器模型

内容说明：

- 3.1 对于servlet和ServletContext初始化参数：编写servlet代码访问初始化参数，创建部署描述文件元素来声明初始化参数。

除了第3.3条之外，这一部分的要求在本章都有详细介绍，第3.3条涵盖在关于“过滤器”的一章。

- 3.2 对于基本的servlet属性作用域（请求、会话和上下文）：编写servlet代码来增加、获取和删除属性；给定一种使用场景，明确属性适当的作用域；明确与各个作用域有关的多线程问题。

这一章的大多数内容在本书其他部分也有出现，但是如果你想通过考试，就要好好利用这一章来掌握和记住有关的要求。

- 3.3 描述Web容器请求处理模型的元素：过滤器、过滤器链、请求和响应包装器，以及Web资源（servlet或JSP页面）。

← 在关于“过滤器”的一章介绍。

- 3.4 描述请求、会话和Web应用的Web容器生命周期事件模型；为每个作用域生命周期创建和配置监听者类；创建和配置作用域属性监听者类；给定一个场景，明确要使用的适当的属性监听者。

- 3.5 描述RequestDispatcher机制；编写servlet代码来创建一个请求分派器；编写servlet代码转发或包含目标资源；明确容器向目标资源提供的额外的请求作用域属性。



Kim想在DD中配置他的email地址，而  
不是硬编码写到servlet类中

Kim不希望在servlet中这样做：

```
PrintWriter out = response.getWriter();
out.println("blooper@wickedlysmart.com");
```

↑  
把地址硬编码很不好！  
如果email有变化怎么办？他就必须重  
新编译……

他更想把email地址放在部署描述文件（web.xml文件）中，这样部署这个Web应用时，servlet就能以某种方式从DD“读取”他的email地址。这样一来，他就不用在servlet类中硬编码写入地址，要改变email，只需修改web.xml文件就行了，而不必去改动他的servlet源代码。

## 解决之道：初始化参数

你已经知道，可以把请求参数传递给doGet()或doPost()，不过servlet还可以有初始化参数。

在DD文件 (web.xml)中：

```
<servlet>
    <servlet-name>TestInitParam</servlet-name>
    <servlet-class>com.wickedlysmart.TestInitParam</servlet-class>
    <init-param>
        <param-name>adminEmail</param-name>
        <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>

</servlet>
```

给定一个参数名 (param-name)  
和一个参数值 (param-value)。  
很简单，只要保证它在DD的  
<servlet>元素中就行。

在servlet代码中：

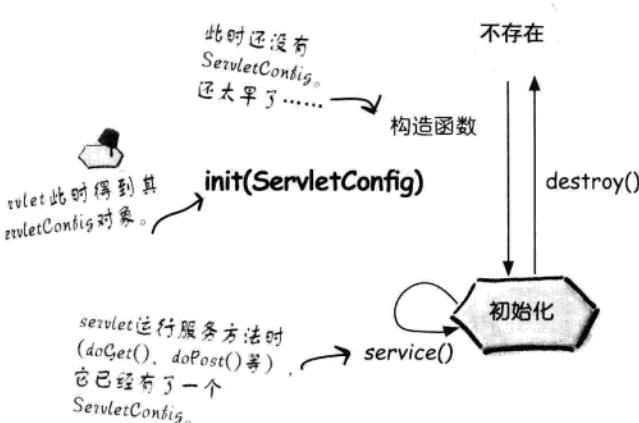
```
String adminEmail = getServletConfig().getInitParameter("adminEmail");
```

↑  
每个servlet都继承了一个  
getServletConfig()方法。

getServletConfig()方法返回一个……先等  
等……ServletConfig。getInitParameter()是  
ServletConfig的一个方法。

# 在servlet初始化之前不能使用servlet初始化参数

前面已经看到了，servlet继承了getServletConfig()，所以可以从servlet中的任何方法调用getServletConfig()来得到ServletConfig的一个引用。一旦有了一个ServletConfig引用，就可以调用getInitParameter()。不过，要记住，不能从构造函数调用这个方法！在servlet的一生中这还为时太早……容器调用init()之前，它还不能算一个完整的servlet。



容器 初始化 一个  
servlet时，会为这个  
servlet建一个唯一的  
ServletConfig。

容器 从 DD “读出”  
servlet初始化参  
数，并把这些参数交给  
ServletConfig，然后把  
ServletConfig传递给  
servlet的 init()方法。

there are no  
Dumb Questions

**问：**回顾上一章，你说过servlet要成为一个真正的、正式的servlet需要两样东西。你提到了ServletConfig，另外还提到了ServletContext。

**答：**对，没错，再过几页就会谈到ServletContext。不过，现在我们只关心ServletConfig，因为要从这里得到servlet初始化参数。

**问：**等一下！上一章你说我们可以覆盖init()方法，但对ServletConfig参数却只字未提！

**答：**我们是没有提到init()方法要取一个ServletConfig参数，因为你覆盖的那个init()方法没有这个参数。超类包括两个版本的init()，一个有ServletConfig参数，还有一个便利版本，它没有任何参数。继承的init(ServletConfig)方法会调用无参数的init()方法，所以你只需覆盖无参数的版本就足够了。

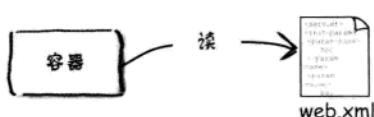
不过，并不是说不能覆盖那个有ServletConfig参数的版本，但是如果这样做，最好调用super.init(ServletConfig)!但退一步讲，为什么要覆盖init(ServletConfig)方法呢？这没什么必要吧，因为只要调用继承的getServletConfig()方法就可以得到ServletConfig了。

# servlet初始化参数只能读一次

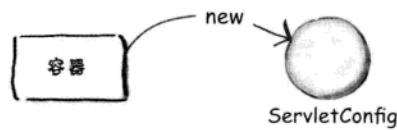
## ——就是在容器初始化servlet的时候

容器建立一个servlet时，它会读DD，并为ServletConfig创建名/值对。此后容器不会再读初始化参数！一旦参数置于ServletConfig中，就不会再读了，除非你重新部署servlet。仔细想想看。

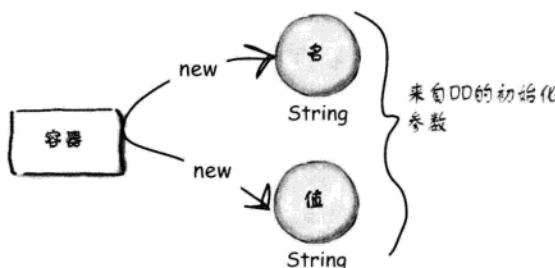
- ① 容器为这个servlet读取部署描述文件，包括servlet初始化参数<init-param>。



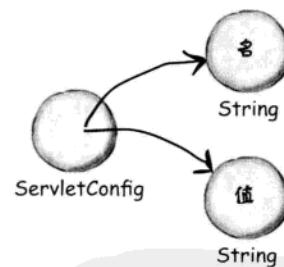
- ② 容器为这个servlet创建一个新的ServletConfig实例。



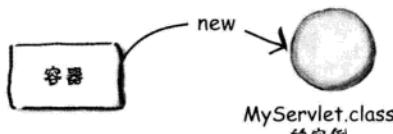
- ③ 容器为每个servlet初始化参数创建一个String名/值对。假设这里只有一个初始化参数。



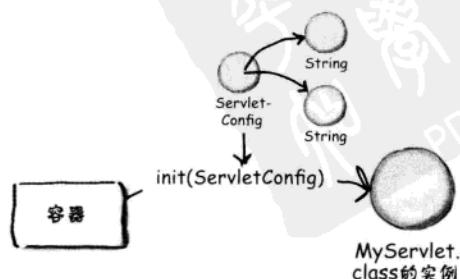
- ④ 容器向ServletConfig提供名/值初始化参数的引用。



- ⑤ 容器创建servlet类的一个新实例。



- ⑥ 容器调用servlet的init()方法，传入ServletConfig的引用。





there are no  
**Dumb Questions**

**问：**那么，Tomcat中“重新部  
署”按钮在哪呢？

**答：**Tomcat中并没有这样一个  
简单的管理工具，能通过一个按钮完  
成部署和重新部署（不过Tomcat中  
确实提供了一个管理工具）。但是请  
考虑一下，要改变servlet的初始化参  
数，哪种做法更糟糕？你可以快速  
修改web.xml文件、关闭Tomcat（bin/  
shutdown.sh），然后重启Tomcat(bin/  
startup.sh）。重启时，Tomcat就会查看  
它的webapps目录，并部署它在那里  
发现的所有应用。

**问：**告诉Tomcat关闭和启动当  
然很容易，但是如果Web应用正在运  
行怎么办？这些Web应用也必须关  
闭！

**答：**从理论上讲是这样。如果关  
闭Web应用的目的只是想重新部署一  
个servlet，这是有些草率了，特别是，  
如果你的网站业务量很大，这就很不  
合适。不过，正因如此，大多数生  
产质量的Web容器都允许你完成热部  
署，这意味着，你不必重新启动服务  
器，也不用让其他Web应用关闭。实  
际上，Tomcat就包括一个管理工具，  
允许你部署、取消部署以及重新部署

整个Web应用，而不必重启Tomcat。  
在生产环境中最好这样做。不过，对  
于测试来说，重启Tomcat更为容易。  
有关这个管理工具的信息可以参考：

[http://jakarta.apache.org/tomcat/tomcat-  
5.0-doc/manager-howto.html](http://jakarta.apache.org/tomcat/tomcat-5.0-doc/manager-howto.html)

但是在实际中，甚至热部署也存在很  
大的问题，如果只是因为初始化参  
数值有变化就让应用关闭，就算是只  
关闭一个应用，这也不是一个好的想  
法。如果初始化参数的值经常变化，  
最好让servlet方法从一个文件或数据  
库得到值，不过这种做法意味着，每  
次servlet代码运行时都会有更多的开  
销，而不是只在初始化期间有开销。

# 测试ServletConfig

ServletConfig的主要任务是提供初始化参数。它还提供了一个ServletContext，但是我们一般会以另外一种方式得到上下文，getServletName()方法很少使用。

在DD文件(web.xml)中：

```
<?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/web-app_2_4.xsd" version="2.4">
  <servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>com.example.TestInitParams</servlet-class>
    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>mainEmail</param-name>
      <param-value>blooper@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>BeerParamTests</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

在servlet类中：

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestInitParams extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test init parameters<br>");

        java.util.Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
            out.println("<br>param name = " + e.nextElement() + "<br>");
        }
        out.println("main email is " + getServletConfig().getInitParameter("mainEmail"));
        out.println("<br>");
        out.println("admin email is " + getServletConfig().getInitParameter("adminEmail"));
    }
}
```

javax.servlet.ServletConfig

<<interface>> <b>ServletConfig</b>
getInitParameter(String)
Enumeration getInitParameterNames()
getServletContext()
getServletName()

大多数人都不会用  
到这个方法。

哦，我刚发现我的应用中使用了JSP来显示页面。那么JSP能“看到”servlet初始化参数吗？

## JSP能不能得到Servlet初始化参数？

ServletConfig用于servlet配置（而不是JSPConfig）。所以，如果想让应用的其他部分使用你在DD中置于servlet初始化参数的信息，就要再做些工作。

前面的啤酒应用是怎么做的？我们使用了一个请求属性将模型信息传递给JSP……

```
// inside the doPost() method
String color = request.getParameter("color");
BeerExpert be = new BeerExpert();
List result = be.getBrands(color);
request.setAttribute("styles", result);
```

还记得吗：我们从请求得到了用户的颜色选择。

} 然后实例化，并使用模型来得到视图所需的信息。

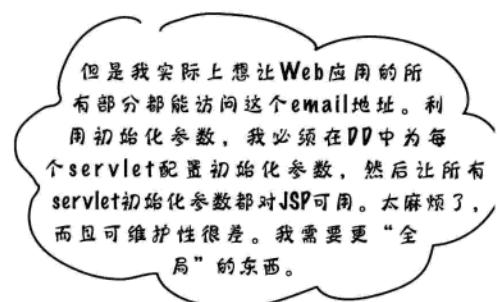
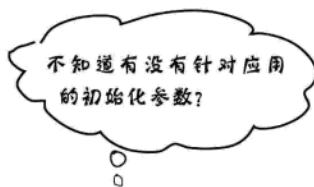
然后在请求中设置一个“属性”，接收转发请求的JSP就能够得到这个信息。

我们现在也可以这么做。请求对象允许设置属性（可以把它认为是名/值对，而值可以是任何对象），得到请求的任何其他servlet或JSP都能使用这些属性。这意味着，如果使用RequestDispatcher来转发请求，接收到转发请求的任何servlet或JSP就能使用请求对象的属性。在这一章的最后还会详细介绍RequestDispatcher，不过现在我们只关心怎么得到数据（在这里就是要得到email），并把数据提供给Web应用中需要它的其他部分，而不只是一个servlet。

## 设置请求属性是可以的…… 但是只适用于接收转发请求的 JSP

对于啤酒应用，把客户请求的模型信息保存在请求对象中是有意义的，因为下一步就是把请求转发给一个JSP，由它负责创建视图。由于这个JSP需要模型数据，而且数据只与特定的请求相关，所以一切都OK。

但是对于email地址，这种方法就没有多大的帮助了，因为我们可能需要在整个应用中使用这个地址！一种方法是让servlet读取初始化参数，然后把它们保存在一个地方，这样应用的其他部分都能使用，但是这么一来，我们必须知道，应用部署时总是先运行哪个servlet，而且只要对Web应用有改动，就会把一切都搞砸。不行，这么做是不行的。



## 解决之道：上下文初始化参数

上下文初始化参数与servlet初始化参数很类似，只不过上下文参数对整个Web应用可用，而不只是针对一个servlet。所以，这说明应用中的所有servlet和JSP都自动地能够访问上下文初始化参数，我们不必操心为每个servlet配置DD，而且如果值有变化，只需在一个地方修改就行了！

在DD文件(web.xml)中：

将< servlet >元素中  
的< init-param >元素  
去掉。

```
<context-param>
  <param-name>adminEmail</param-name>
  <param-value>clientheadererror@wickedlysmart.com</param-value>
</context-param>
```

重要提示！< context-param > 是针对整个应用的，所以并不嵌套在某个< servlet > 元素中！把< context-param > 放在< web-app > 里，但是要放在所有< servlet > 声明之外。

在servlet代码中：

```
out.println(getServletContext().getInitParameter("adminEmail"));
```

↑  
每个servlet都继承一个getServletContext()方法。JSP也能以特殊方式访问上下文。

毫不奇怪，getServletContext()方法返回一个 ServletContext 对象。它有一个 getInitParameter() 方法。

或

```
ServletContext context = getServletContext();
out.println(context.getInitParameter("adminEmail"));
```

这里把代码分为两步——得到ServletContext引用，并调用其getInitParameter()方法。

# 记住servlet初始化参数和上下文初始化参数之间的区别

下面对上下文初始化参数和servlet初始化参数之间的区别做一个简单的回顾。要特别注意，它们都称为初始化参数，不过只有servlet初始化参数才在DD配置中包含“init”一词。

## 上下文初始化参数

## Servlet初始化参数

### 部署描述文件

在<web-app>元素中，但是不在具体的<servlet>元素内。

```
<web-app .....>
  <context-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </context-param>

  <!-- other stuff indicating
       servlet declarations --&gt;
&lt;/web-app&gt;</pre>
```

注意，对于上下文初始化参数，DD中没有任何地方提到“init”，而servlet初始化参数不同。

在每个特定servlet的<servlet>元素中。

```
<servlet>
  ...
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>

  ...
</servlet>
```

### Servlet代码

```
getServletContext().getInitParameter("foo");  
同样的方法名！
```

```
getServletConfig().getInitParameter("foo")
```

### 可用性

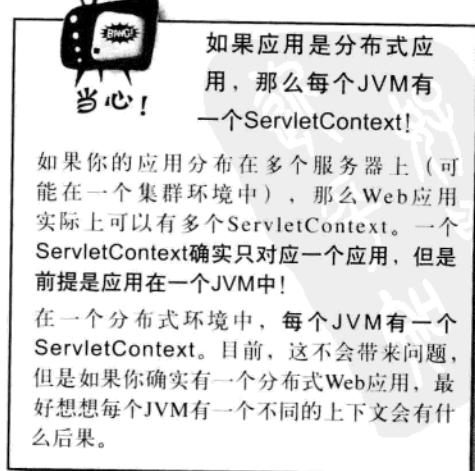
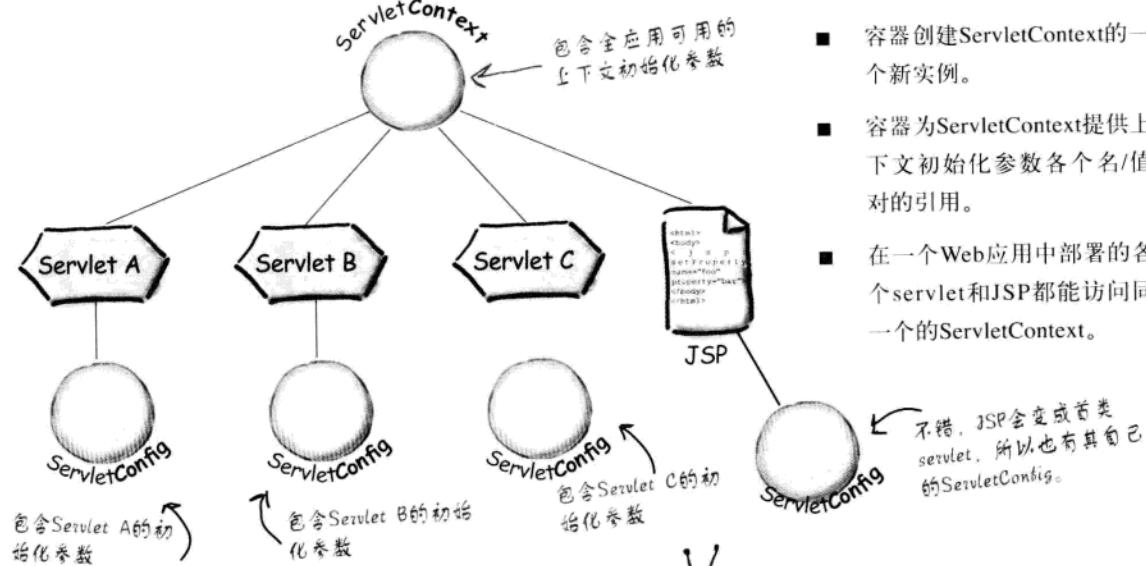
对Web应用中的所有servlet和JSP都可用。

只对配置了<init-param>的相应servlet可用。  
(不过servlet可以通过保存在一个属性中来得到更大范围的可用性)

# 每个servlet有一个ServletConfig

# 每个Web应用有一个ServletContext

整个Web应用只有一个ServletContext，而且Web应用中的所有部分都能访问它。不过，应用中的各个servlet有自己的ServletConfig。部署Web应用时，容器会建立一个ServletContext，这个上下文对Web应用中的每个Servlet和JSP（也会成为一个servlet）都可用。



## there are no Dumb Questions

**问：** 怎么会有不一致的命名机制呢？DD里的元素分别是<context-param>和<init-param>，但是在servlet代码中，为什么他们都使用getInitParameter()方法呢？

**答：** 取名时没有人征求我们的意见。如果他们曾经问过我们，我们当然会说应该是getInitParameter()和getContextParameter()才对，这样才能与DD中的XML元素匹配。或者也可以用不同的XML元素，可能是<servlet-init-param>和<context-init-param>。不过，这样一来也就没有把这些搞清楚的乐趣了。

**问：** 为什么会使用<init-param>？如果总是使用<context-param>，这样应用的其他部分就能重用这些值，而不用对每个servlet声明重复同样的XML代码，不能这样做吗？

**答：** 这要看应用中的哪些部分要看到这些值。你的应用逻辑可能要求使用一个值，而且仅限于一个servlet能得到这个值。不过，一般来说，开发人员发现，与特定于servlet的servlet初始化参数相比，应用范围的上下文初始化参数的作用更大。上下文参数最常见的用途可能就是存储数据库查找名。你希望应用的所有部分都能访问正确的查找名，而且一旦这个名有变化，只需在一处修改。

**问：** 在同一个Web应用中，如果上下文初始化参数和servlet初始化参数同名，会怎么样呢？

**答：** 新泽西的一个研究设备可能会莫名其妙地造出一个很小的黑洞，就这么一个小黑洞可能会覆盖地核，以至于把整个地球都毁掉。

不过也可能没有什么问题，因为既然你通过两个不同的对象（ServletContext或ServletConfig）来得到参数，所以没有命名空间冲突。

**问：** 如果修改XML来改变一个初始化参数的值（servlet初始化参数或上下文初始化参数），servlet或Web应用的其余部分什么时候能看到这个改变呢？

**答：** 只有当Web应用重新部署时才会看到。要记住，我们以前讲到过，servlet只会初始化一次，就是在它生命刚开始时初始化，也正是这个时候会为它提供ServletConfig和ServletContext。容器创建这两个对象时会从DD读取，并设置对象的值。

**问：** 我能不这样做，而是在运行时设置值吗？肯定会有个API能让我动态地修改这些值……

**答：** 不行，并没有这样的API。可以在ServletContext或ServletConfig里查看，你会发现一个获取方法（getInitParameter()），但是找不到设置方法。根本没有setInitParameter()之类的方法。

**问：** 这太不完整了。

**答：** 这就是初始化参数。Init（初始化）一词源自拉丁文initialization（初始化）。如果你把它们看作是纯粹的部署时常量，那就对了。实际上，这一点实在太重要了，所以我们要用粗体再写一遍。

**要把初始化参数认为是部署时常量！**

**可以在运行时得到这些初始化参数，但是不能设置。根本没有setInitParameter()。**

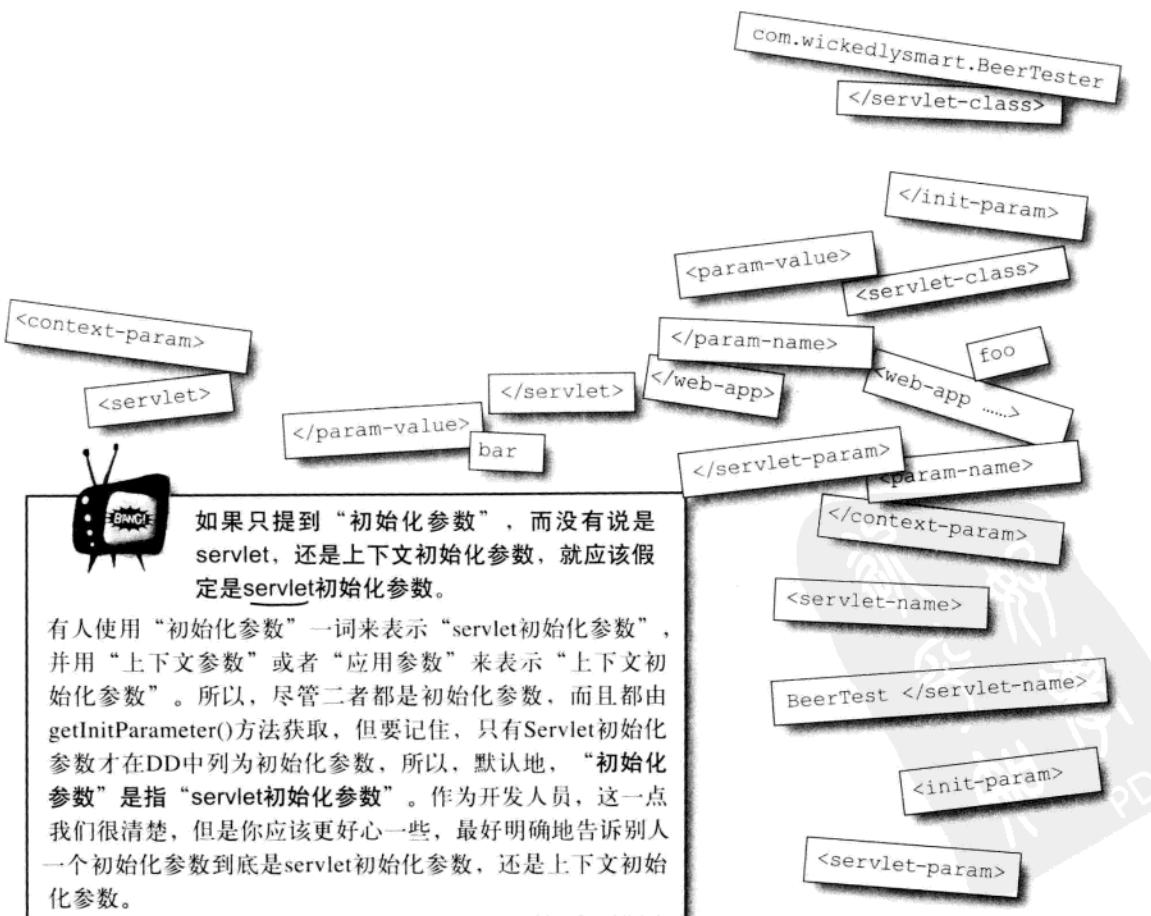


## 代码贴

重新组织下面的代码贴构成一个DD，其中声明一个与以下servlet代码匹配的参数：

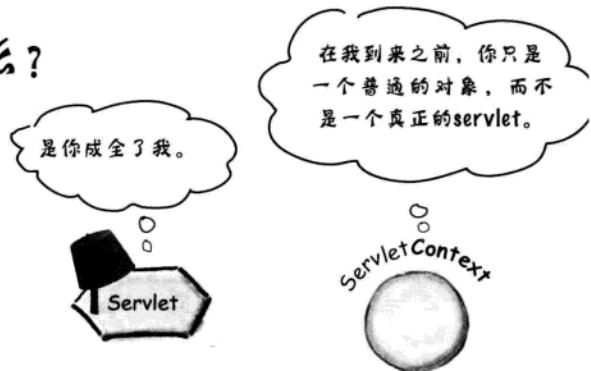
```
getServletContext().getInitParameter("foo");  
并不是所有代码贴都会用到！
```

(说明：看到<web-app>时，要记住这里为了节省篇幅做了删减。但是只有当<web-app>标记包含了所需的全部属性时，才能部署web.xml文件)



# 用ServletContext还能做什么？

ServletContext是JSP或servlet与容器及Web应用其他部分的一个连接。这里列出了一些ServletContext方法。粗体显示的是考试要求必须掌握的一些方法。



<<interface>>	
ServletContext	
<b>getInitParameter(String)</b>	
<b>getInitParameterNames()</b>	
<b>getAttribute(String)</b>	
<b>getAttributeNames()</b>	
<b>setAttribute(String, Object)</b>	
<b>removeAttribute(String)</b>	
<hr/>	
<b>getMajorVersion()</b>	
<b>getServerInfo()</b>	
<hr/>	
<b>getRealPath(String)</b>	
<b>getResourceAsStream(String)</b>	
<b>getRequestDispatcher(String)</b>	
<hr/>	
<b>log(String)</b>	
<i>// 更多方法</i>	

javax.servlet.ServletContext

得到初始化参数以及  
获取/设置属性。

得到有关服务器/容器  
的信息。

写至服务器的日志文件（特  
定于开发商）或System.out。

我们会用好几页来讨论参数与  
属性。

本章稍后会讨论  
RequestDispatcher。



可以用两种不同方式得到ServletContext……

servlet的ServletConfig对象拥有该servlet的ServletContext的一个引用。

所以如果考试时看到这样的servlet代码，请不要奇怪：

```
getServletConfig().getServletContext().getInitParameter()
```

这样做不仅是合法的，而且与下面的代码等价：

```
this.getServletContext().getInitParameter()
```

在一个servlet中，只有一种情况需要通过ServletConfig得到ServletContext，那就是你的Servlet类没有扩展HttpServlet或GenericServlet（getServletContext()方法是从GenericServlet继承的）。但是使用非HTTP servlet的可能性几乎为0。所以只需调用getServletContext()方法就可以了，不过倘若真的看到使用ServletConfig来得到上下文的代码也不用诧异。

但是，在一个非servlet的类中（例如，一个辅助/工具类），该怎么做呢？有人可能为这个类传递了一个ServletConfig，类代码必须使用getServletContext()来得到ServletContext对象的一个引用。

**问：** 一个Web应用的各个部分如何访问自己的ServletContext？

**答：** 对于servlet，你已经知道了，要调用继承的getServletContext()方法。

对于JSP，则稍有些不同，JSP有一些所谓的“隐式对象”，其中包括ServletContext。在介绍JSP的几章中，你会清楚地了解到JSP如何使用ServletContext。

**问：** 那么，你是通过上下文得到内置的日志支持？听上去很不错嘛！

**答：** 嗯，不是这样的。除非你的Web应用确实很简单，否则还有更好的办法来支持日志。最流

行、最健壮的日志机制是Log4j；有关信息可以访问Apache网站：

<http://jakarta.apache.org/log4j>

还可以使用java.util.logging的日志API（logging API），这是J2SE 1.4版本新增加的。

对于简单的小实验，使用ServletContext log()方法就很好，但是在实际的生产环境中，你很可能想选用其他方法。在O'Reilly出版的《Java Servlet & JSP 经典实例》中很好地介绍了利用Log4j以及不利用Log4j时如何实现Web应用的日志机制。

日志机制不在考试范围内，但是它很重要。幸运的是，你会发现有关的API很容易使用。

不好意思，打断你们的  
ServletContext聚会，不过，这些  
初始化参数只能是**STRING**！这就有问  
题了，如果我想用所有servlet都能使用  
的一个数据库DataSource来初始化我  
的应用，这该怎么办呢？



## 如果希望应用初始化 参数是一个数据库 DataSource呢？

上下文参数只能是String。毕竟，你不能把一个Dog对象硬塞到XML部署描述文件中（实际上，确实可以用XML表示一个串行化对象，但是在当前的Servlet规范中还没有相关的支持……没准将来会提供）。

如果你真的想让Web应用的所有部分都能访问一个共享的数据连接，该怎么做呢？当然可以把这个DataSource查找名放在一个上下文初始化参数里，这也是当前上下文参数最常见的一种用法。

不过，在此之后谁将这个String参数转换成由Web应用各部分共享的一个具体DataSource引用呢？

不能把这个代码放在servlet中，因为你选择谁作为第一个servlet来查找DataSource，并把它存储在一个属性中呢？你真的想保证总是让某个特定的servlet最先运行吗？好好考虑一下。



### 开动脑筋

怎么解决这个问题？

怎样用一个对象来初始化Web应用？假设你需要String上下文初始化参数来创建这个对象（想想数据库的例子）。

噢，如果整个Web应用有一个  
main方法就好了。能放一些在  
servlet或JSP之前运行的代  
码……



她想要的是一个监听者 (*listener*)。

她想监听一个上下文初始化事件，这样就能得到上下文初始化参数，并在应用为客户提供服务之前运行一些代码。

她需要这样一个东西，它能一直坐在那里，等着得到应用正在启动的通知。

但是，应用中哪一部分能做这个工作呢？你不希望让一个servlet来做这样的监听者，这不是servlet的任务。

在一个普通的独立Java应用中，这是没有问题的，因为你有一个main()方法！但是对于servlet，该怎么办呢？

你需要的不是servlet或JSP，而是另外某种Java对象，其唯一的用途就是初始化应用（也可能在知道应用快完蛋时还要取消初始化，清理资源等）。

# 她想要的是ServletContextListener

可以建立一个单独的类，而不是一个servlet或JSP，它能监听ServletContext一生中的两个关键事件，初始化（创建）和撤销。这个类实现了javax.servlet.ServletContextListener。

我们需要一个单独的对象，它能做到：

- 上下文初始化（应用正在得到部署）时得到通知。
  - 从ServletContext得到上下文初始化参数。
  - 使用初始化参数查找名建立一个数据库连接。
  - 把数据库连接存储为一个属性，使得Web应用的各个部分都能访问。
- 上下文撤销（应用取消部署或结束）时得到通知。
  - 关闭数据库连接。.



ServletContextListener类：

```

import javax.servlet.*;
import javax.servlet.annotation.*;

public class MyServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
        //code to initialize the database connection
        //and store it as a context attribute
    }

    public void contextDestroyed(ServletContextEvent event) {
        //code to close the database connection
    }
}

```

ServletContextListener在javax.servlet包中

上下文监听者很简单：只需实现ServletContextListener。

这是要得到的两个通知。它们会提供一个ServletContextEvent。

好了，我有一个监听者类了。那现在我要做什么？把这个类放在哪里？谁来实例化这个类？怎样向事件注册？监听者如何在适当的ServletContext中设置属性？



## 开动脑筋

你觉得应该采用何种机制在特定Web应用中加入监听者？

提示：怎样告诉容器有关Web应用其他部分的信息？容器可能在哪里发现监听者？



# 教程：一个简单的ServletContextListener

下面我们一步一步地建立和运行一个ServletContextListener。这只是一个简单的测试类，以便你知道怎么联合所有部分协同工作。我们没有使用数据库连接的例子，这是因为如果使用数据库连接，就必须先建立一个数据库才行。不过不管在监听者回调方法中放什么代码，步骤都是一样的。

在这个例子中，我们要把String初始化参数转换为一个真正的对象——一个Dog。监听者的任务是得到有关狗品种（Beagle、Poodle等）的上下文初始化参数，然后使用这个String来构造一个Dog对象。监听者再把这个Dog对象存放到一个ServletContext属性中，以便servlet获取。

关键是，servlet现在能访问一个共享的应用对象（这里就是一个共享的Dog），而且不必读取上下文参数。这个共享的对象是一个Dog还是一个数据库连接并没有关系。重点是使用初始化参数来创建一个对象，让应用的所有部分都能共享这个对象。

## Dog例子：

- 监听者对象向ServletContextEvent对象请求应用ServletContext对象的一个引用。
- 监听者使用这个ServletContext引用得到“breed”的上下文初始化参数，这是一个String，表示狗的品种。
- 监听者使用这个狗品种String构造一个Dog对象。
- 监听者使用ServletContext引用在ServletContext中设置Dog属性。
- 这个Web应用的测试servlet从ServletContext得到Dog对象，并调用这个Dog的getBreed()方法。

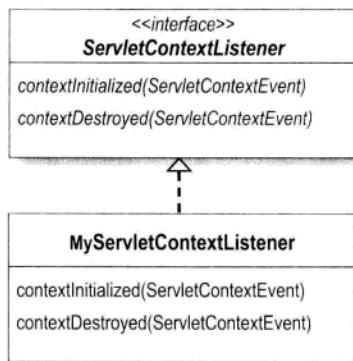


在这个例子中，我们把一个Dog放到ServletContext中。

# 建立和使用一个上下文监听者

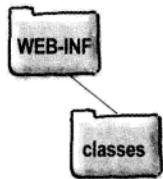
你可能还在考虑，容器怎样发现和使用监听者呢……就像告诉容器Web应用其他部分的有关情况一样，要用同样的办法配置监听者，对，就是通过web.xml部署描述文件！

## ① 创建一个监听者类。



要监听`ServletContext`事件，需要编写一个实现`ServletContextListener`的监听者类，把它放在WEB-INF/classes目录中，并在部署描述文件中放一个`<listener>`元素来告诉容器。

## ② 把类放入WEB-INF/classes。



(并不是只能放在这里……容器会在几个位置上查找类，WEB-INF/classes只是其中之一。在关于“部署”的一章将介绍其他位置)

## ③ 在web.xml部署描述文件放一个`<listener>`元素。

```

<listener>
  <listener-class>
    com.example.MyServletContextListener
  </listener-class>
</listener>
  
```

你要回答一个问题——`<listener>`元素放在部署描述文件的哪一部分：放在一个`<servlet>`元素内部，还是就放在`<web-app>`下面？仔细想想。

# 需要3个类和一个部署描述文件

对于这个上下文监听者测试示例，需要编写一些类和web.xml文件。

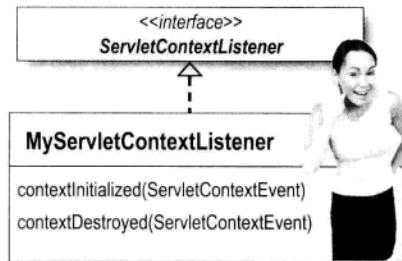
为了便于测试，我们把所有类都放在同一个包中：

com.example

## ① ServletContextListener

**MyServletContextListener.java**

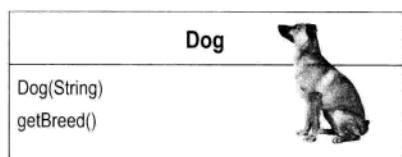
这个类实现了ServletContextListener，它得到上下文初始化参数，创建Dog，并把Dog设置为上下文属性。



## ② 属性类

**Dog.java**

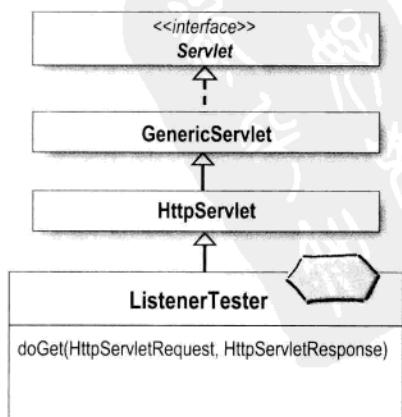
Dog类只是一个普通的Java类。它的任务是作为一个属性值，由ServletContextListener实例化，并设置在ServletContext中，以便servlet获取。



## ③ Servlet

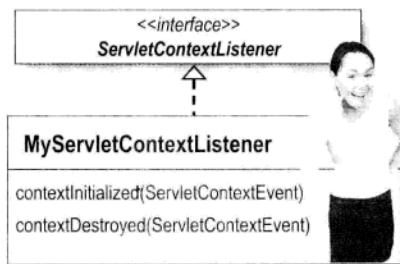
**ListenerTester.java**

这个类扩展了 HttpServlet。它的任务是验证监听者的工作，为此，这个Servlet会从上下文得到Dog属性，然后调用Dog的getBreed()，并把结果打印到响应（以便我们在浏览器上看到）。



# 编写监听者类

这与你熟悉的其他类型的监听者很类似，如Swing GUI事件处理器。要记住，我们需要得到上下文初始化参数来得出狗的品种，建立Dog对象，并把这个Dog作为属性放在上下文中。



```

package com.example;

import javax.servlet.*;

public class MyServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext(); ← 由事件得到ServletContext。
        String dogBreed = sc.getInitParameter("breed"); ← 使用上下文得到初始化参数。
        Dog d = new Dog(dogBreed); ← 建立一个新Dog。
        sc.setAttribute("dog", d); ← 使用上下文来设置一个属性（名/对象对），这个属性就是Dog，现在应用的其他部分就能得到这个属性（Dog）的值了……
    }

    public void contextDestroyed(ServletContextEvent event) {
        // nothing to do here
    }
}
  
```

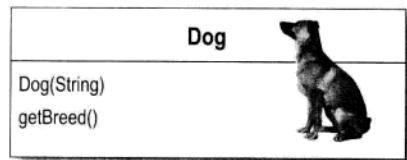
这里什么也不用做，Dog不需要清理工作……上下文如果没有了，这说明整个应用都结束了，当然也包括Dog。

## 编写属性类(Dog)

对了，我们需要一个Dog类，这个类表示读取了上下文初始化参数之后要存储在ServletContext中的对象。

```
package com.example;           这里没什么特殊的地方。  
public class Dog {           只是一个普通的Java类。  
    private String breed;  
  
    public Dog(String breed) {  
        this.breed = breed;  ↗ (使用这个上下文初始化参数作为  
    }                         Dog构造函数的参数)  
  
    public String getBreed() {  
        return breed;  
    }  
}
```

↑  
servlet会从上下文得到Dog（就是监听者  
设置为属性的那个Dog）。调用该Dog的  
getBreed()方法，并把品种打印到响应中，以  
便我们在浏览器中看到。



**问：** 我记得servlet属性必须是可串行化的(Serializable)……

**答：** 这个问题很有趣。有多种不同的属性类型，属性是否为Serializable，这一点只对会话属性有意义。而且只在一种情况下有影响，这就是应用可能分布在不只一个JVM上。我们会在关于“会话”的一章更详细地讨论这个内容。

从理论上讲，没有必要让所有属性（包括会话属性）都成为Serializable，不过，确实可以考虑让所有属性默认为Serializable，除非有充分的理由不这么做。

想想看，你真的能肯定任何人都不会在远程方法调用中把这种类型的对象用作为参数或返回值吗？你能保证使用这个类（在这里就是Dog）的应用绝对不会在分布式环境中运行吗？

所以，尽管并不一定非要让所有属性都是Serializable，但还是尽量这么做比较好。



# 编写servlet类

这个类用来测试ServletContextListener。如果一切正常，Servlet的doGet()方法第一次运行时，Dog已经作为一个属性放在ServletContext中。

```
package com.example;

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

public class ListenerTester extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

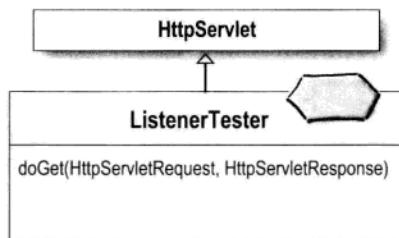
        out.println("test context attributes set by listener<br>");
        out.println("<br>");

        Dog dog = (Dog) getServletContext().getAttribute("dog");
        out.println("Dog's breed is: " + dog.getBreed());
    }
}
```



getattribute()返回的类型是Object！需要对返回结果进行强制类型转换！

但是getInitParameter()返回的是String。所以必须对getattribute()的返回结果完成强制类型转换，而getInitParameter()的返回结果可以直接赋值给一个String。所以……如果考试中有代码没有使用强制类型转换，可别被蒙住了：  
Dog d = ctx.getAttribute(“dog”); ← 不对！  
(假设 ctx是一个ServletContext)



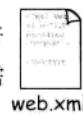
到此为止还没有特殊的地方……只是一个常规的servlet。

现在从ServletContext得到Dog。如果监听者正常，第一次调用服务方法之前，Dog就已经放在上下文中了。

如果有问题，肯定是在这里出现……如果试图调用getBreed()，但是没有Dog，就会得到一个超大的NullPointerException异常。

## 编写部署描述文件

现在我们来告诉容器：已经为这个应用建立了一个监听者，为此要使用`<listener>`元素。这个元素很简单，它只需要类名，就这么简单。



这是`web.xml`文件，放在这个Web应用的WEB-INF目录中。

```

<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">

  <servlet>
    <servlet-name>ListenerTester</servlet-name>
    <servlet-class>com.example.ListenerTester</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ListenerTester</servlet-name>
    <url-pattern>/ListenTest.do</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>breed</param-name>
    <param-value>Great Dane</param-value>
  </context-param>

  <listener>
    <listener-class>
      com.example.MyServletContextListener
    </listener-class>
  </listener>

</web-app>

```

需要应用的一个上下文初始化参数。  
监听者需要这个参数来构造Dog。

把这个类注册为监听者。重要提示：  
`<listener>`元素不放在`<servlet>`元素内。否则  
不能正常工作，因为上下文监听者对应一个  
ServletContext事件（这意味着在整个应用范围  
内）。关键是，要在初始化任何servlet之前对  
应用初始化。

## there are no Dumb Questions

**问：** 等等……你怎么告诉容器这是ServletContext事件的监听者？好像没有一个<listener-type>之类的XML元素来指出监听的事件类型呀！不过，我注意到，类名里有“ServletContextListener”，容器是通过这个知道的吗？根据命名约定吗？

**答：** 不是，这里没有命名约定。这样命名只是为了使我们编写的类更清楚一些。容器只会检查类，并注意监听者接口或多个接口（一个监听者可以实现多个监听者接口），以此明确监听什么类型的事件。

**问：** 这是不是说，在servlet API中还有其他类型的监听者？

**答：** 对，还有许多其他类型的监听者，稍后就会谈到。

# 编译和部署

下面让它真正运行起来。步骤如下：

**① 编译前面的3个类。**

它们都在同一个包中……

**② 在Tomcat中创建一个新Web应用。**

- 创建一个名为listenerTest的目录，并把它放在Tomcat webapps目录下。
- 创建一个名为WEB-INF的目录，放在listenerTest目录下。
- 把web.xml文件放在WEB-INF目录中。
- 在WEB-INF中建一个classes目录。
- 在classes下建一个与包结构一致的目录结构：目录名为com，下面包含example目录。

**③ 把3个已编译的类文件放在Tomcat下你的Web应用的目录结构中。**

listenerTest/WEB-INF/classes/com/example/Dog.class

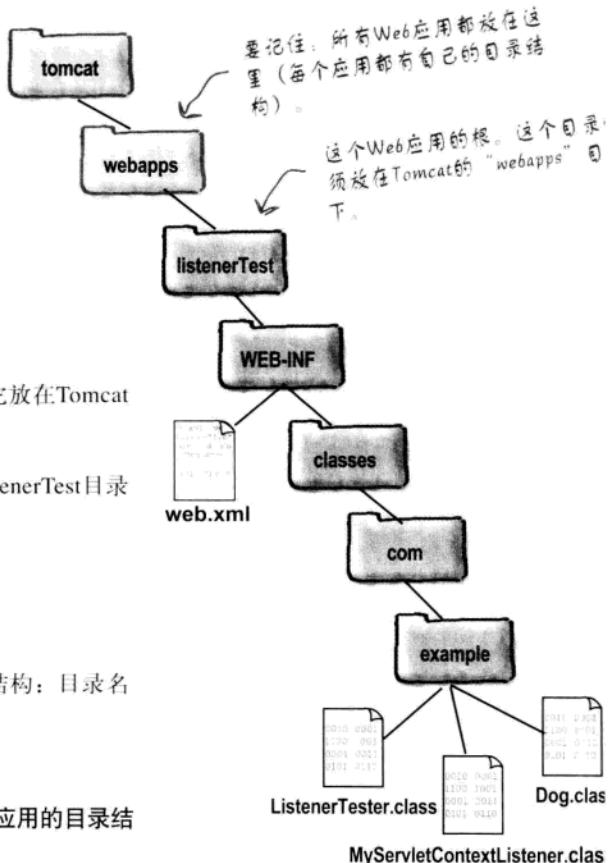
listenerTest/WEB-INF/classes/com/example/ListenerTester.class

listenerTest/WEB-INF/classes/com/example/MyServletContextListener.class

**④ 把web.xml部署描述文件放在这个Web应用的WEB-INF目录中。**

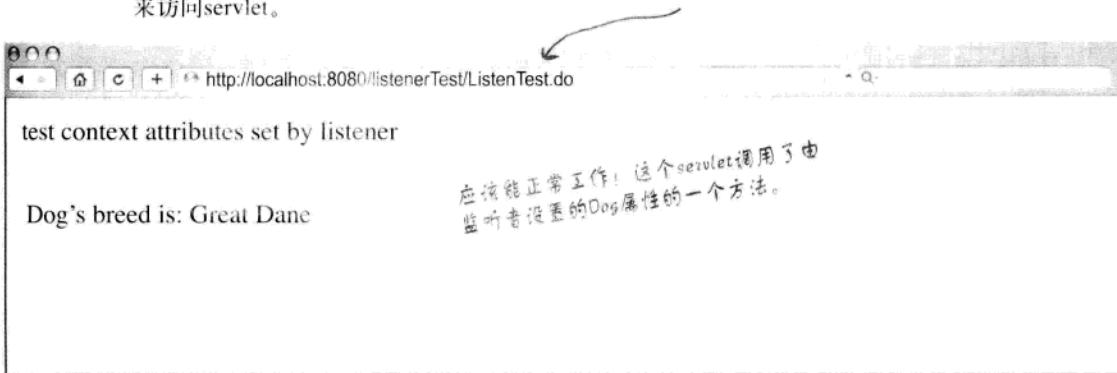
listenerTest/WEB-INF/web.xml

**⑤ 部署应用，为此关闭Tomcat后再重启。**



## 试试看

打开浏览器，直接访问servlet。我们不想那么麻烦再建一个HTML页面，所以直接键入部署描述文件中servlet映射里的URL（ListenTest.do）来访问servlet。



## 除错

如果你得到一个NullPointerException异常，说明你未能从getAttribute()得到一个Dog。检查一下setAttribute()中使用String名，一定要与getAttribute()中使用的String名匹配。

再来检查你的web.xml，确保已经注册了<listener>。

仔细查看服务器日志，看看能不能得出监听者是否真正得到了调用。

为了尽可能避免产生混淆，我们所起的名字都稍有一点差异。之所以这样命名，是因为我们希望你能注意这些名字如何使用，倘若把所有东西都起同样的名字，就很难分清这些名字对应用有什么影响。

Servlet类名： ListenerTester.class

要注意这是Listener还是Listener  
Tester还是Test。

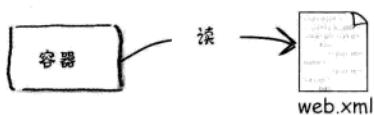
Web应用目录名： listenerTest

映射到servlet的URL模式： ListenTest.do

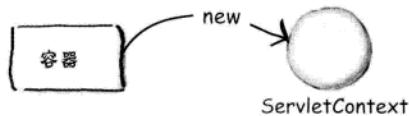
## 完整的故事……

下面是一个完整的场景，自始（应用初始化）至终（servlet运行）完全展现。你会看到，在第11步中，我们把Servlet初始化压缩为一大步。

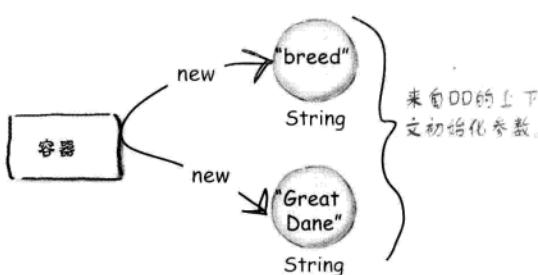
- ① 容器读这个应用的部署描述文件，包括<listener>和<context-param>元素。



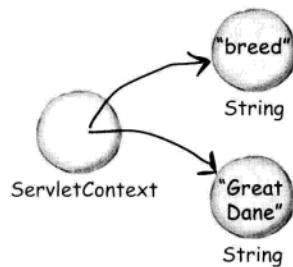
- ② 容器为这个应用创建一个新的ServletContext，应用的所有部分都会共享这个上下文。



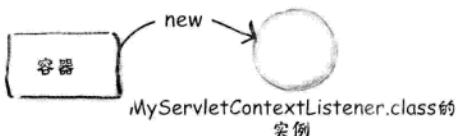
- ③ 容器为每个上下文初始化参数创建一个String名/值对。这里假设只有一个参数。



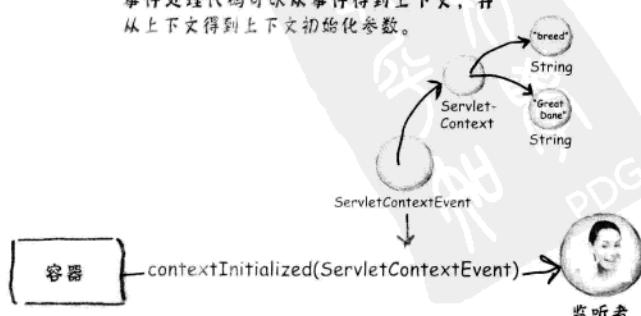
- ④ 容器将名/值参数的引用交给ServletContext。



- ⑤ 容器创建MyServletContextListener类的一个新实例。

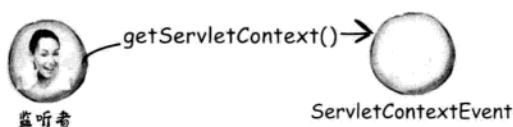


- ⑥ 容器调用监听者的contextInitialized()方法，传入一个新的ServletContextEvent。这个事件对象有ServletContext的一个引用，所以事件处理代码可以从事件得到上下文，并从上下文得到上下文初始化参数。

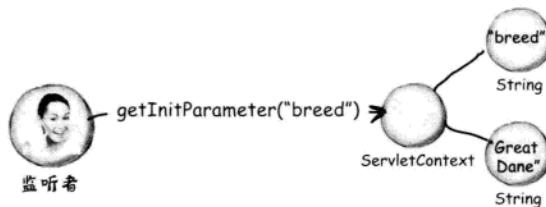


# 故事（续）……

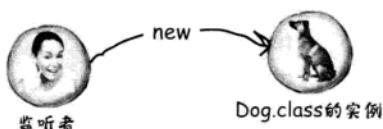
- ⑦ 监听者向ServletContextEvent请求ServletContext的一个引用。



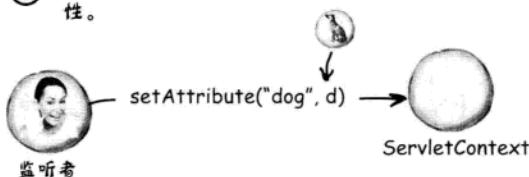
- ⑧ 监听者向ServletContext请求上下文初始化参数“breed”。



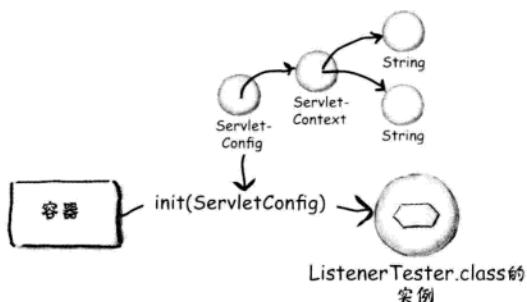
- ⑨ 监听者使用初始化参数来构造一个新的Dog对象。



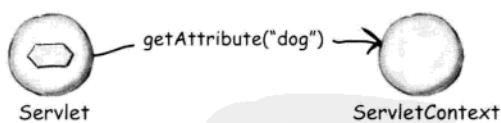
- ⑩ 监听者把Dog设置为ServletContext中的一个属性。



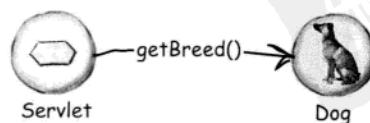
- ⑪ 容器建立一个新的Servlet（也就是说，利用初始化参数建立一个新的ServletConfig，为这个ServletConfig提供ServletContext的一个引用，然后调用Servlet的init()方法）。



- ⑫ Servlet得到一个请求，并向ServletContext请求属性“dog”。



- ⑬ Servlet在Dog上调用getBreed()（并将结果打印到HttpResponse）。



我还在想……既然属性可以通过编程设置（不同于初始化参数），我能监听属性事件吗？比如说有人增加或替换了一个Dog？



## 监听者：不只是面向上下文事件……

只要是生命周期里的重要时刻，总会有一个监听者在监听。除了上下文事件外，还可以监听与上下文属性、servlet请求和属性，以及HTTP会话和会话属性相关的事件。



Relax 你不用了解所有监听者API。

除了ServletContextListener外，实际上不需要记住每个监听者接口的每一个方法。不过……确实需要知道你能监听哪些类型的事件。

考试要求很明确：会给你一个场景（开发人员开发应用的目标），你要确定适用什么类型的监听者，或者是否有可能得到生命周期事件的通知。

注意：在下一章之前我们不会过多的讨论会话，所以如果你还不知道HTTP会话是什么，以及为什么要考虑HTTP会话，也不用太担心……



## 选择监听者

将左边的场景与支持该目标的监听者接口

(在这一页下面) 匹配, 每个接口只能用一次 (没错, 这个内容还没有介绍过。看看你能不能从名字就得出结论)。下一页有答案, 所以别偷看!



### 场景

你想知道一个Web应用上下文中是否增加、删除或替换了一个属性。

你想知道有多少个并发用户。也就是说, 你想跟踪活动的会话。

每次请求到来时你都想知道, 以便建立日志记录。

增加、删除或替换一个请求属性时你希望能够知道。

你有一个属性类 (这个类表示的对象将放在一个属性中), 而且你希望这个类型的对象绑定到一个会话或从会话删除时得到通知。

增加、删除或替换一个会话属性时你希望能够知道。

从这些监听者接口中选择。每个监听者只能用一次。

*HttpSessionAttributeListener*

*ServletRequestListener*

*HttpSessionBindingListener*

*HttpSessionListener*

*ServletContextAttributeListener*

*ServletRequestAttributeListener*

# 8个监听者

场景	监听者接口	事件类型
你想知道一个Web应用上下文中是否增加、删除或替换了一个属性。	javax.servlet.ServletContextAttributeListener <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	ServletContextAttributeEvent
你想知道有多少个并发用户。也就是说，你想跟踪活动的会话（下一章将详细介绍会话）。	javax.servlet.http.HttpSessionListener <i>sessionCreated</i> <i>sessionDestroyed</i>	HttpSessionEvent
每次请求到来时你都想知道，以便建立日志记录。	javax.servlet.ServletRequestListener <i>requestInitialized</i> <i>requestDestroyed</i>	ServletRequestEvent
增加、删除或替换一个请求属性时你希望能够知道。	javax.servlet.ServletRequestAttributeListener <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	ServletRequestAttributeEvent
你有一个属性类（这个类表示的对象将放在一个属性中），而且你希望这个类型的对象在绑定到一个会话或从会话删除时得到通知。	javax.servlet.http.HttpSessionBindingListener <i>valueBound</i> <i>valueUnbound</i>	HttpSessionBindingEvent
增加、删除或替换一个会话属性时你希望能够知道。	javax.servlet.http.HttpSessionAttributeListener <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	HttpSessionBindingEvent 注意，这里的命名不一致！对应 HttpSessionAttributeListener 的事件可能和你预想的不一样（你可能以为 HttpSessionAttributeEvent）。
你想知道是否创建或撤销了一个上下文。	javax.servlet.ServletContextListener <i>contextInitialized</i> <i>contextDestroyed</i>	ServletContextEvent
你有一个属性类，而且希望这个类型的对象在其绑定的会话迁移到另一个JVM时得到通知。	javax.servlet.http.HttpSessionActivationListener <i>sessionDidActivate</i> <i>sessionWillPassivate</i>	HttpSessionEvent 不是“ HttpSessionActivationEvent ”

# HttpSessionBindingListener

你可能不清楚一个 HttpSessionBindingListener 和一个 HttpSessionAttributeListener 之间有什么区别（也可能不是你，是与你共事的某个人）。

普通的 HttpSessionAttributeListener 类只是希望当会话中增加、删除或替换了某种类型的属性时能够知道。但是 HttpSessionBindingListener 不同，有了 HttpSessionBindingListener，属性本身才能够在增加到一个会话或者从会话删除时得到通知。

```
package com.example;

import javax.servlet.http.*;

public class Dog implements HttpSessionBindingListener {
    private String breed;

    public Dog(String breed) {
        this.breed=breed;
    }

    public String getBreed() {
        return breed;
    }

    public void valueBound(HttpSessionBindingEvent event) {
        // code to run now that I know I'm in a session
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // code to run now that I know I am no longer part of a session
    }
}
```



使用这个监听者，我能更清楚我在应用中的角色。我被放到一个会话中时（或者从会话中取出时），他们会告诉我。

这一次 Dog 属性也是一个监听者……它会监听 Dog 本身何时增加到会话或从会话删除（注意：绑定监听者并不在 DD 中注册……而会自动进行）。

这里使用“bound”（绑定）和“unbound”（解除绑定）来表示“增加到”和“删除”。

**问：** 那好，我知道怎么回事了。Dog（将增加到会话的一个属性）想知道它什么时候进入和离开会话。但我不明白这是为什么。

**答：** 如果你对实体 bean 有点了解的话……可以把这个功能看作是“一种小实体 bean”。如果你不了解实体 bean，应该马上到书店去买两本《Head First EJB》（一本自己看，另一本给一位好友，这样你们就能讨论有关的一些问题了）。

另外，还可以这样想，假设 Dog 是一个 Customer 类，每个

---

活动的实例表示一个客户的信息，包括名字、地址、订单信息等等。实际的数据存储在底层数据库中。你使用数据库信息来填充 Customer 对象的字段，但是问题是，你怎么保持数据库记录和 Customer 信息同步呢？另外什么时候让它们同步？你知道，只要有一个 Customer 对象增加到会话，就应该用数据库中相应记录的客户数据来刷新 Customer 对象的字段。所以 valueBound() 方法就像是敲钟提示“用数据库里的新数据来加载我……因为从上一次用我之后数据可能有变化”。而 valueUnbound() 则是说“用 Customer 对象字段的值来更新数据库”。



## 记住监听者

尽量填完这个表。要记住监听者接口和方法遵循一致的命名模式（基本上是这样）。

答案在这一章的最后。



属性监听者	
其他生命周期监听者	
所有属性监听者中的方法（除了绑定监听者）	
与会话相关的生命周期事件 (不包括与属性相关的事件)	
与请求相关的生命周期事件 (不包括与属性相关的事件)	
与servlet上下文相关的生命周期事件（不包括与属性相关的事件）	

## 到底什么是属性？

我们已经了解ServletContext监听者如何创建Dog对象（得到上下文初始化参数之后），以及如何将Dog作为一个属性存储（设置）到ServletContext，这样应用的其他部分就能得到它了。较早前，在啤酒教程中，我们还看到了servlet如何将对模型调用的结果作为属性保存到请求对象（通常是HttpServletRequest），以便JSP/视图能得到这个值。

属性就是一个对象，可能设置（也称为绑定）到另外3个servlet API对象中的某一个，包括ServletContext、HttpServletRequest（或ServletRequest）或者HttpSession。可以把它简单地认为是一个映射实例对象中的名/值对（名是一个String，值是一个Object）。在实际中，我们并不知道也不关心它具体如何实现，我们关心的只是属性所在的作用域。换句话说，谁能看到这个属性，以及属性能存活多久。



**属性就像是钉到公告栏上的一个对象。有人在公告栏上贴公告，以便其他人看到。**

**这里有一个关键的问题：  
谁能访问公告栏，公告栏能存在多久？换句话说，  
属性的作用域是什么？**



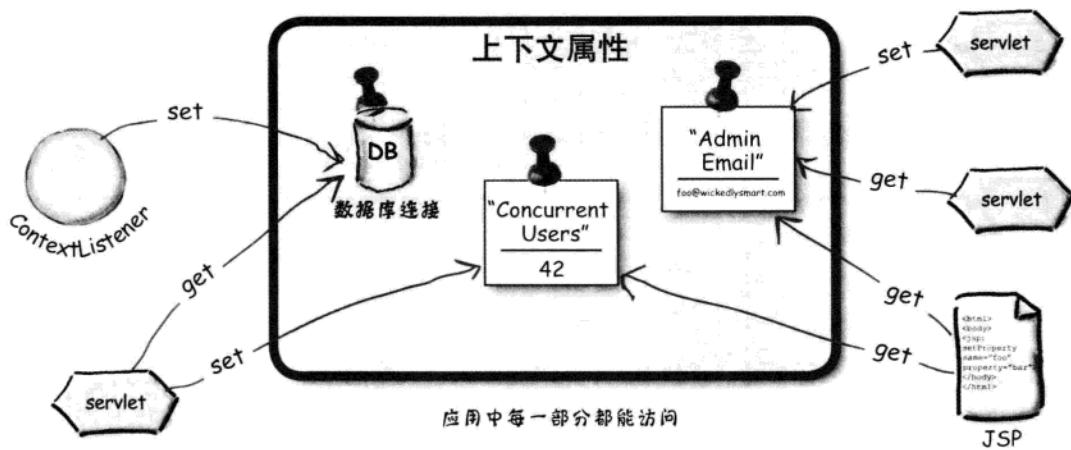
# 属性不是参数！

如果你没有接触过servlet，可能需要花点时间明确属性和参数之间的区别。要知道，利用下表，我们只需多花一点点时间就能在考试中把属性和参数问题设计得尽可能的复杂（注1）。

属性		参数
类型	应用/上下文 请求 会话	应用/上下文初始化参数 请求参数 Servlet初始化参数 没有会话参数的说法！
设置方法	<code>setAttribute(String name, Object value)</code>	不能设置应用和Servlet初始化参数，还记得吧？它们都在DD中设置（对于请求参数，可以调整查询串，但这是另一回事）。
返回类型	<code>Object</code>	<code>String</code> ← 这是一个显著的区别！
获取方法	<code>getAttribute(String name)</code>  不要忘了必须强制类型转换，因为返回类型是 <code>Object</code> 。	<code>getInitParameter(String name)</code>

注1：这是实话。如果考试很简单，很容易，就算你通过了考试也不会有成就感。也许你认为我们的想法是尽量让考试题难一些，这样要想通过考试就需要买一本学习指导书，但并不是这样，这不是我们的初衷。说实在的，我们只是在为你考虑，想让你真正学到东西。

## 三个作用域：上下文、请求和会话





## 属性作用域

尽量填写下表。考试要求一定要理解属性作用域，在实际中这也很重要，因为你必须知道对于一个给定场景最好使用哪一个作用域。后面几页将给出答案，但是先不要看！如果你要参加考试，听我们的……你要花些时间好好考虑，完全自己动脑筋来填这个表。

可访问性 (谁能看到)	作用域 (能存活多久)	适用于……
Context (上下文)		
HttpSession (会话)		
Request (请求)		

（注意：在考虑作用域时应该想到垃圾回收的影响……有些属性在应用取消部署或结束之前不能被垃圾回收。考试中不会考内存管理的内容，但是对于这个方面你应该有所了解）。

# 属性API

3个属性作用域（上下文、请求和会话）分别由ServletContext、ServletRequest和HttpSession接口处理。每个接口中对应属性的API方法完全相同。

**Object getAttribute(String name)**  
**void setAttribute(String name, Object value)**  
**void removeAttribute(String name)**  
**Enumeration.getAttributeNames()**

上下文

请求

会话

<<interface>>
ServletContext
getInitParameter(String)
getInitParameterNames()
<b>getAttribute(String)</b>
<b>setAttribute(String, Object)</b>
<b>removeAttribute(String)</b>
<b>getAttributeNames()</b>
getMajorVersion()
getServerInfo()
getRealPath(String)
getResourceAsStream(String)
getRequestDispatcher(String)
log(String)
// 还有更多方法……

<<interface>>
ServletRequest
getContentType()
getParameter(String)
<b>getAttribute(String)</b>
<b>setAttribute(String, Object)</b>
<b>removeAttribute(String)</b>
<b>getAttributeNames()</b>
// 还有更多方法……

<<interface>>
HttpSession
<b>getAttribute(String)</b>
<b>setAttribute(String, Object)</b>
<b>removeAttribute(String)</b>
<b>getAttributeNames()</b>
setMaxInactiveInterval(int)
getId()
getLastAccessedTime()
// 还有更多方法……

<<interface>>
HttpServletRequest
getContextPath()
getCookies()
getHeader(String)
getQueryString()
getSession()
// 还有更多方法……

## 属性不好的一面……

Kim想要测试属性。他设置了一个属性，然后立即得到这个属性的值，并把它显示在响应里。他的doGet()代码是这样的：

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

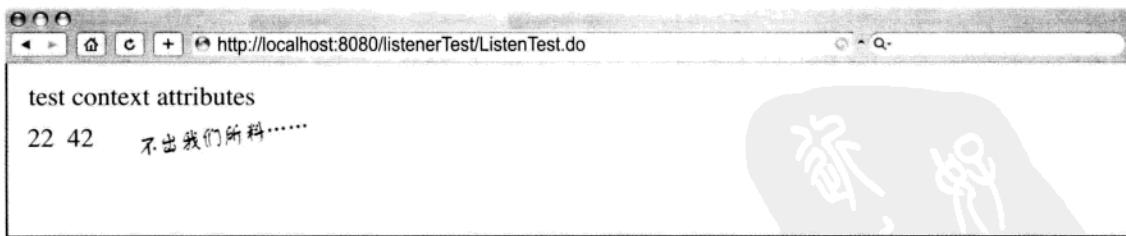
    out.println("test context attributes<br>");

    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));
}
```

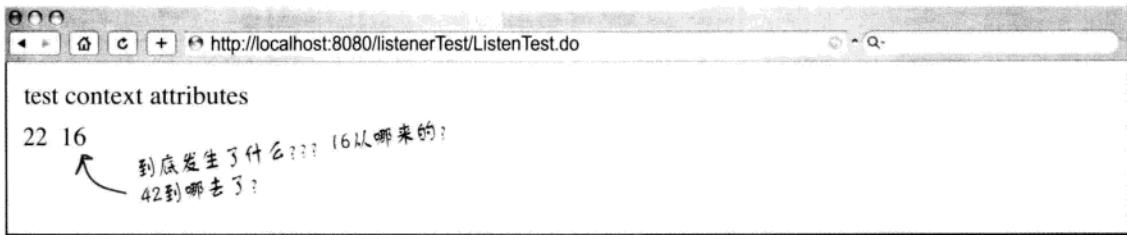
第一次运行的时候他能看到下面的结果。

这与他想像中的完全一样。



# 但是接下来出了严重的问题……

第二次运行的时候，他惊讶地发现：



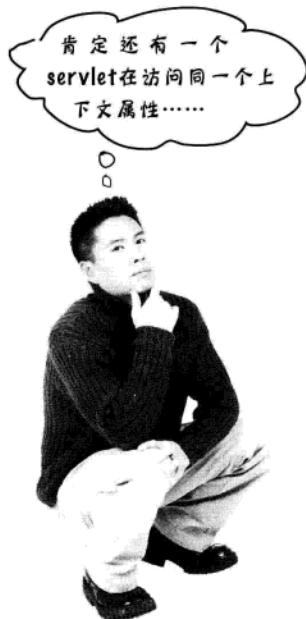
## 开动脑筋

仔细看看代码，想想看到底发生了什么。你能发现解释这个问题的蛛丝马迹吗？

要想揭开这里的秘密，你现在了解的信息可能还不够，所以给你一条线索：Kim把这个代码放在一个测试servlet中，这个servlet是一个更大的测试Web应用的一部分。换句话说，包括这个doGet()方法的servlet部署在一个更大的应用中。

现在你能发现问题吗？

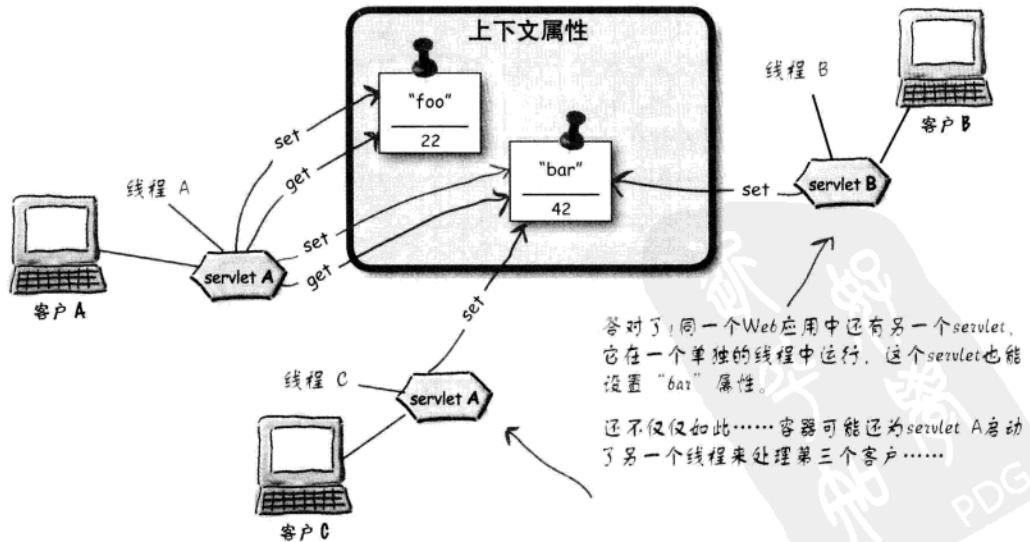
你能想出来怎么解决这个问题吗？



## 上下文作用域不是线程安全的！

这正是问题所在。

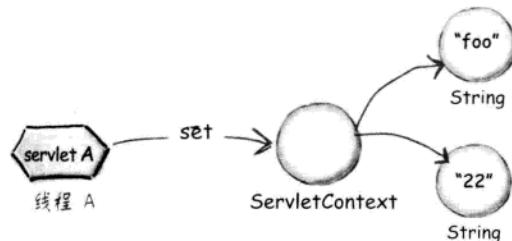
要记住，应用中的每一部分都能访问上下文属性，而这意味着可能有多个servlet。多个servlet则说明你可能有多个线程，因为请求是并发处理的，每个请求在一个单独的线程中处理。不论这些请求指向同一个servlet还是不同的servlet，都是如此。



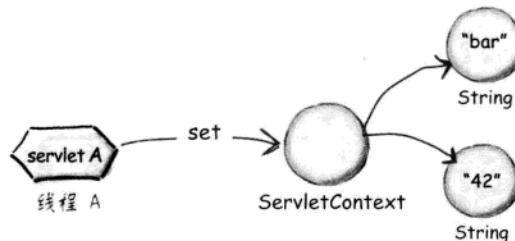
# 问题的慢动作回放……

Kim的测试servlet发生了什么，下面来详细分析。

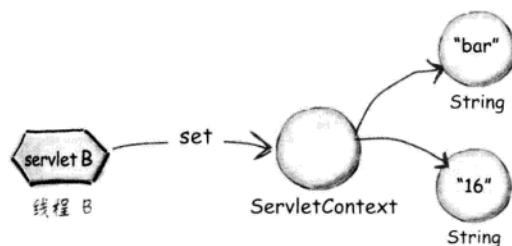
- ① Servlet A设置上下文属性“foo”的值为“22”。



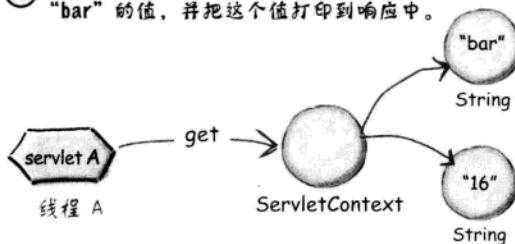
- ② Servlet A 设置上下文属性“bar”的值为“42”。



- ③ 线程B成为活动线程（线程A回到一种可运行但是未运行的状态），并设置上下文属性“bar”的值为“16”（值42丢失了）。



- ④ 线程 A重新成为活动线程，它得到“bar”的值，并把这个值打印到响应中。



```
getServletContext().setAttribute("foo", "22");
getServletContext().setAttribute("bar", "42");

out.println(getServletContext().getAttribute("foo"));
out.println(getServletContext().getAttribute("bar"));
```

在servlet A设置bar" 值和取得  
"bar" 值的间隙，另一个servlet线  
程介入，把 "bar" 设置为一个不  
同的值。  
所以等到servlet A 打印 "bar" 的值  
时，值已经改成 "16" 了。

## 怎样让上下文属性做到线程安全？

来听听其他开发人员怎么说…



我想可以同步doGet()方法，但是好像不太好。不过，我想不出还有什么好办法。



对doGet()同步意味着再也不能并发处理了。如果你让doGet()同步，这说明servlet一次只能处理一个客户！



规范指出，保护属性是你自己的事情。同步会带来很大开销，如果你本来不需要同步，又何必对你强加这些同步开销呢？当然，有些Web容器确实以某种方式实现了同步，但是不能完全保证，所以使用时最好当心。



为什么Servlet规范的开发人员不干脆同步ServletContext中的获取和设置属性方法，这不就能使属性做到线程安全了吗？

# 同步服务方法是一个非常非常糟糕的想法

那好，既然我们知道了同步服务方法会妨碍并发性，但这样确实能提供线程保护，是这样吗？看看下面的代码，这段代码是合法的，请确定这样能不能避免Kim遇到的问题，能不能防止上下文属性被另一个servlet修改……

```
public synchronized void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test context attributes<br>");

    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));
}
```

还是不行！不错，  
这是一个合法的servlet，  
但是我不认为这样能解决  
问题……

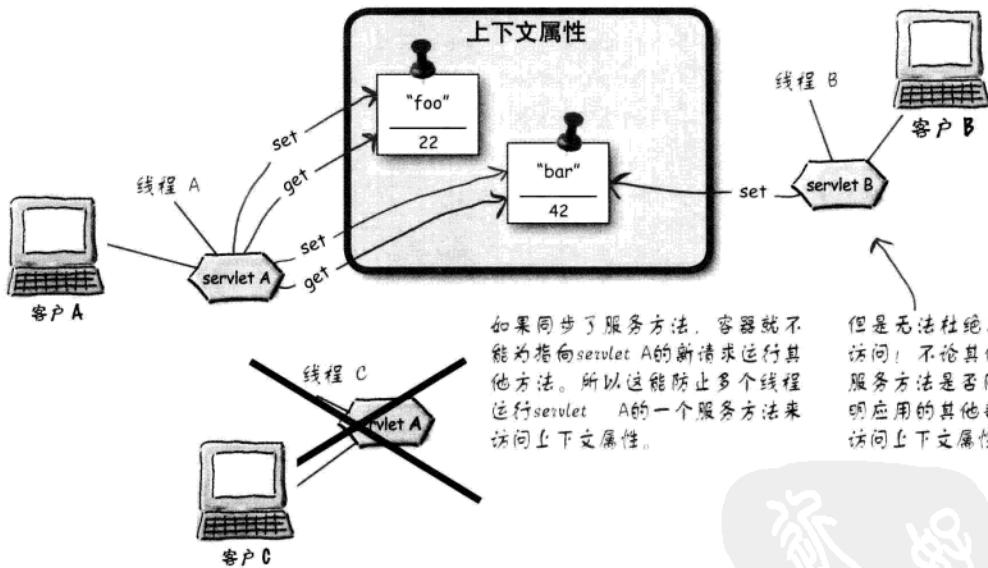


你怎么想？它能解决Kim的问题吗？如果不能确定，再回过头去看代码和图。

## 同步服务方法并不能保护上下文属性！

同步服务方法意味着servlet中一次只能运行一个线程……但是并不能阻止其他servlet或JSP访问这个属性！

同步服务方法会防止同一个servlet中的其他线程访问上下文属性，但是不能阻止另外一个不同servlet的访问。



# 不是需要对servlet加锁……而是需要对上下文加锁！

保护上下文属性的一般做法是对上下文对象本身同步。如果访问上下文的每一个人都必须先得到上下文对象的锁，就能保证一次只有一个线程可以得到或设置上下文属性。但是……这里有一个“如果”。只有当处理同一上下文属性的所有其他代码也对ServletContext同步时，这种做法才奏效。如果一段代码没有请求锁，那么该代码还是能自由地访问上下文属性。不过，你在设计Web应用时，可以要求每个人在访问属性之前必须先请求一个锁。



对于上下文属性，在Servlet上同步没有任何好处，因为应用的其他部分还是能访问上下文！



ServletContext

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test context attributes<br>");

    synchronized(getServletContext()) {
        getServletContext().setAttribute("foo", "22");
        getServletContext().setAttribute("bar", "42");

        out.println(getServletContext().getAttribute("foo"));
        out.println(getServletContext().getAttribute("bar"));
    }
}
```

既然有了上下文锁，可以认为一旦我们进入了同步块，这些上下文属性就能安全地不被其他线程访问，除非我们离开同步块……这样理解基本上是对的。这里的安全是指“同样对ServletContext同步的其他代码无法同时访问上下文属性。”

但是，为了让上下文属性做到线程安全，这已经是你能采用的最佳方法了。

现在得到了上下文本身的锁！要以这种方式来保护上下文属性状态（而不应该使用synchronized(this))。



可能会看到大量有关线程安全的代码

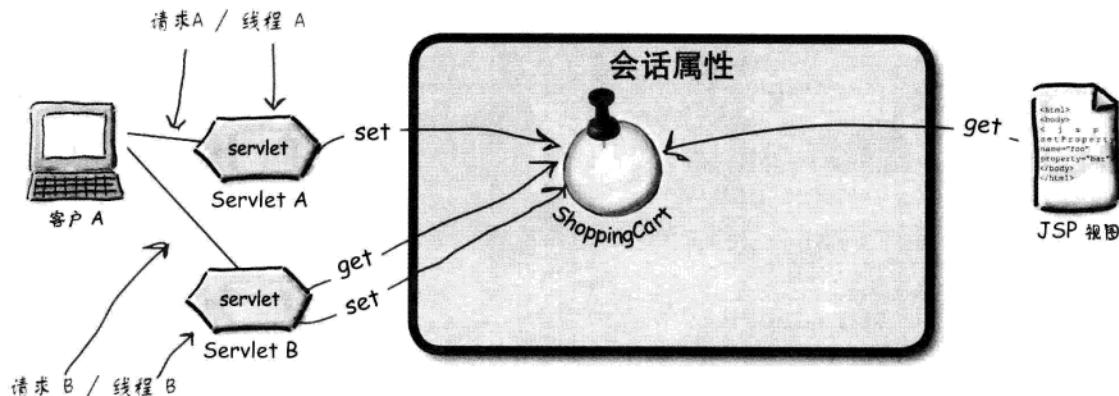
在考试中，你可能会看到很多代码展示怎样让属性做到线程安全，为此可能有不同策略。你要根据特定的目标，确定这些代码是否可行。代码本身可能是合法的（可以编译和运行），但这并不意味着它一定能解决问题。

# 会话属性是线程安全的吗？

先考虑一下。

我们还没有详细介绍HTTP会话（会在关于“会话”的一章中讨论），但是你应该已经知道会话就是一个对象，用于维护与一个客户的会话状态。对于同一个客户的多个请求，会话会跳过请求持久存储。但是我们只讨论了一个客户的情况。

如果只有一个客户，而且一个客户一次只有一个请求，这不就很自然地说明会话是线程安全的吗？换句话说，就算是涉及多个servlet，在任何给定时刻，一个特定客户也只有一个请求……所以会话中一次只有一个线程运行，是这样吗？



就算是两个servlet能在单独的线程中访问会话属性，每个线程都是一个单独的请求。所以看上去是安全的。

除非……

如果同一个客户同时有多个请求，你能想像这种场景吗？

你怎么考虑？这些会话属性能保证线程安全吗？

## 关于属性和线程安全，到底哪些是正确的？



我们知道上下文属性本质上讲并不安全，因为应用的所有部分都能访问上下文属性，可以从任何请求访问（这意味着任何线程）。

很对。再来说说会话属性。它们是安全的吗？

你还只是半瓶子醋！还没有完全掌握会话属性的本质。说话前还要三思。

你要想想容器之外的事情。要让思路开阔一些。

答对了！容器把来自第二个窗口的请求看作是来自同一个会话。

你怎么保护这些会话属性不受多个线程的破坏呢？

不错，但是对谁同步呢？

听听两位黑带高手关于保护属性状态避免多线程问题的讨论。

对，师傅。而且我知道同步服务方法不是解决办法，因为尽管它能防止一个servlet同时服务多个请求，但是不能阻止同一个Web应用中的其他servlet和JSP访问上下文。

可以这么说，师傅。它们只针对一个客户，而且一个客户不会同时做多个请求，这是自然规律。

但是，师傅，我已经考虑过了，还是想不出一个客户怎么会同时做多个请求……

对了，师傅！我明白了！客户可能打开一个新的浏览器窗口！所以容器还是为客户使用相同的会话，尽管它来自另一个浏览器实例，情况是这样吗？

这么说，会话属性不是线程安全的，而且它们也必须得到保护。我要好好想想……

哈，我知道了……只要是访问这些会话属性的代码都必须同步。就像对上下文属性的处理一样。

必须对HttpSession同步！

# 对 HttpSession 同步来保护会话属性

看看前面使用了什么技术来保护上下文属性。我们是怎么做的？

对会话属性也可以采用同样的做法，对 HttpSession 对象同步！

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test session attributes<br>");
    HttpSession session = request.getSession();

    synchronized(session) { 这一次，要对 HttpSession 对象同步来保护会话属性。
        session.setAttribute("foo", "22");
        session.setAttribute("bar", "42");

        out.println(session.getAttribute("foo"));
        out.println(session.getAttribute("bar"));
    }
}
```

*there are no  
Dumb Questions*

**问：** 是不是有些言过其实了？到底有没有可能……一个客户打开另一个浏览器窗口？

**答：** 当然有。你自己肯定就这样做过，可能是一个窗口的响应太慢，你不想一直等着，也可能你把浏览器窗口最小化了，或者把窗口误放在其他位置上，但自己没有意识到，出于诸如此类的原因就会打开另一个窗口。关键是，如果希望会话变量是线程安全的，你就不能有侥幸心理。必须知道，一个会话作用域属性很有可能一次由多个线程使用。

**问：** 同步代码的想法不太好吧，因为这会导致大量的开销，还会妨碍并发性。

**答：** 在同步代码之前，确实必须仔细考虑，因为你说的很对，同步要检查、获得和释放锁，所以确实会增加一些开销。如果你需要保护，可以使用同步，但是要注意，不论是哪种形式的锁都有一条标准：一定要在最短的时间完成目标，使持有锁的时间最短！换句话说，如果代码并不访问受保护的状态，就不要同步这些代码。要让同步块尽可能小。得到锁之后，进入同步块，得到所需的东西，赶快出来，以便释放锁，这样其他线程才能运行这段代码。

# SingleThreadModel 设计用来保护实例变量

以下是servlet规范关于SingleThreadModel（或STM）接口的说明。

确保servlet一次只处理一个请求。

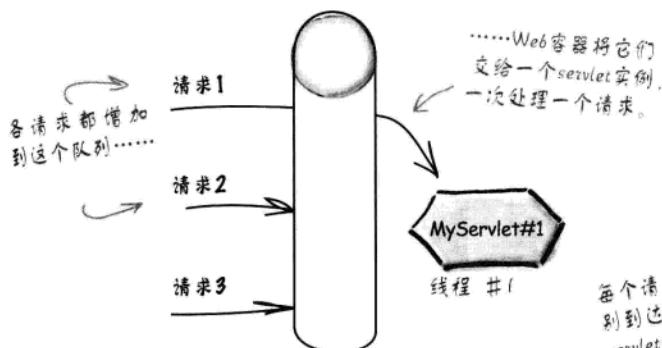
**关键** 这个接口没有任何方法。如果一个servlet实现了这个接口，就可以保证不会在该servlet的服务方法中并发执行两个线程。通过同步对servlet单个实例的访问，或者通过维护一个servlet实例池并把每个新请求分派到一个空闲的servlet，servlet容器可以保证这一点。

但是web容器如何保证一个servlet一次只能得到一个请求呢？

web容器开发商可以有一个选择。容器可以维护单个servlet，不过对所有请求排队，必须在完全处理了一个请求之后才允许处理下一个请求。或者，容器也可以创建一个servlet实例池，并发地处理各个请求，每个servlet实例对应一个请求。

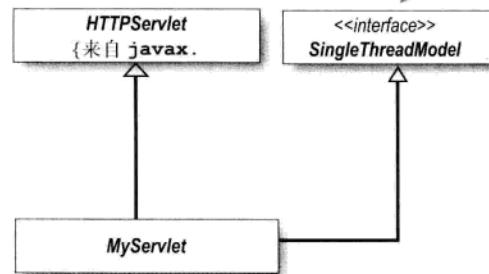
你认为哪一个STM策略比较好？

对所有请求排队



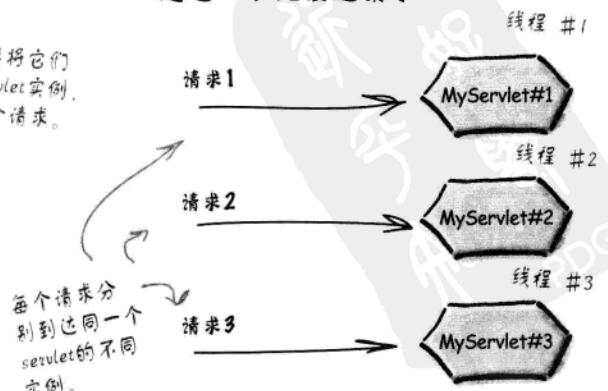
你的servlet应当扩展  
HTTPServlet.....

.....另外还要实现  
SingleThreadModel接口



由于MyServlet实现了STM，web容器可以确保这个servlet一次只能处理一个请求。

通过一个池发送请求



## 哪一种STM实现更好？

对于这个问题，必须再一次咨询我们的黑带高手。他们肯定清楚哪一个才是最棒的STM实现。下面来看他们对此展开的讨论……



让所有请求排队



通过一个池发送请求

让一个servlet的所有请求排队才最为合理。这样能清楚地实现规范编写者的本来意愿。

没错，不过这是保护servlet实例变量的唯一出路。

哈，孩子，你现在就像看到了求签饼里的运气纸条，不过你还不知道这样做的命运到底会有多糟糕……

servlet规范中定义，部署描述文件中的单个servlet声明会在运行时成为单个对象实例，不过现在如果使用STM接口，这个定义就不再成立了。你能想象出在什么情况下使用多个servlet实例会失败？

完全正确！你已经看透了servlet池深藏的秘密。“单个servlet实例”定义不再具有原来的语义。servlet已经与实际不符。

不过，师傅，这样不是会影响性能吗？如果让每一个请求都排队，不是会导致多个用户无法访问同一个servlet吗？

但是，师傅，你别忘了，容器还可以创建一个servlet实例池。这样一来，容器可以用一个servlet实例处理一个请求，而用第二个servlet实例处理另一个请求。各个请求就能得到并行处理了。

师傅，别让我猜谜语了。这种实例池策略会有什么问题吗？

嗯，如果某个实例变量要记录已经处理了多少个请求，这种情况下是不是就不能用多个servlet？否则，计数器变量会有多个不同的计数，而其中没有一个正确的……它们的总和才是正确的计数。

## there are no Dumb Questions

**问：** 怎么了？为什么servlet规范这么不明确？

**答：** 规范的编写者希望容器开发商们能有机会在性能和灵活性方面相互竞争。

**问：** 我怎样才能知道我的开发商使用哪一种策略？

**答：** 嗯，这很可能在Web容器文档的某一部分已经写明。如果没有明确写出，就应该联系你的容器开发商，具体问一问。

**问：** STM策略对我如何编写servlet代码会有什么影响？

**答：** 如果容器使用一种排队策略，“单个servlet实例”的语义就是成立的，你不需要对代码做任何改变。但是，如果容器使用一种实例池策略，一些实例变量的语义就可能改变。例如，如果有一个实例变量，其中存放一个“请求计数器”，当实例池中创建了多个servlet实例时，这个变量将无法正确地计数。在这种情况下，可以选择将这个计数器变量置为一个类变量。

**问：** 但是类变量是线程安全的吗？

**答：** 很遗憾，类变量不是线程安全的，而且STM机制对类变量没有任何帮助。没错，STM机制可以保护实例变量免受并发访问，但是通过将多个实例入池，servlet的语义已经发生改变。更有甚者，STM对其他变量或属性作用域也没有帮助。你得靠你自己……

**问：** 那使用SingleThreadModel有什么好处？

**答：** 说实在的，没有什么好处。这也正是为什么servlet API废弃STM的原因！

← 不过为了考试你还是需要了解STM。



Relax

考试中不需要你了解这些容器STM策略。  
你只要知道STM试图保护servlet的实例变量就足够了。



把不是线程安全的划上对勾（我们已经完成了第一个）。

- 上下文作用域属性
- 会话作用域属性
- 请求作用域属性
- servlet中的实例变量
- 服务方法中的局部变量
- servlet中的静态变量

# 只有请求属性和局部变量是线程安全的！

就是如此。（谈到“局部变量”时也包括方法参数）。其他的都可能由多个线程处理，除非你采取了特别的防范措施。

*there are no  
Dumb Questions*

**问：**那么，实例变量不是线程安全的吗？

**答：**没错。如果有多个客户对一个servlet做出请求，这意味着有多个线程在运行该servlet代码，而且所有线程都能访问servlet的实例变量，因此，实例变量不是线程安全的。

**问：**但是如果你实现了SingleThreadModel，实例变量就是线程安全的，对不对？

**答：**是的，对于这个servlet不会有多个线程，所以这种情况下实例变量确实是线程安全的。但是，这样一来，别人会看不起你，没有人会让你再进servlet俱乐部了。

**问：**我只是做一个假设。“如果有人傻乎乎地实现了SingleThreadModel...”不是说我会这么做。不过，再假设一下，如果我有一个朋友，他同步了服务方法，是不是也能让实例变量做到线程安全？

**答：**对。但你的那个朋友可能很笨。实现SingleThreadModel的效果和同步服务方法的效果实际上是一样的。这两种做法都会让Web应用效率非常差，而且并不能保护会话和属性状态。

**问：**但是如果你不打算使用SingleThreadModel或者同步服务方法，又怎么让实例变量做到线程安全呢？

**答：**根本做不到！看看一个写得好的servlet，你不会看到任何实例变量。或者至少看不到非final变量（如果你是一个Java程序员，应该知道即使是final变量也可以处理，除非它是不可变的）。

所以，如果需要线程安全的状态，就不要用实例变量，因为servlet的所有线程都可以处理实例变量。

**问：**那如果需要servlet的多个实例共享某些东西的时候，该用什么呢？

**答：**停！你说了“servlet的多个实例”。可能你本意不是这个，但是要说清楚，应该知道servlet只有一个实例。记住，只有一个实例，但可以有多个线程。

如果希望所有线程都能访问一个值，要确定哪个属性状态最合适，并把这个值保存在一个属性中。往往可以用下面两种办法来解决这个问题：

1) 把变量声明为服务方法中的局部变量，而不是一个实例变量。

或者

2) 在最合适的作用域中使用一个属性。

# 请求属性和请求分派

如果希望应用的其他组件接管全部或部分请求，就可以使用请求属性。举一个典型的简单例子，一个MVC应用从一个servlet控制器开始，但最后以一个JSP视图结束。控制器与模型通信，得到视图建立响应所需的数据。这些数据不应该放在上下文或会话属性中，因为它只针对这个请求，所以我们把它放在请求作用域中。

那么怎么让组件的其他部分接管这个请求呢？这就要用到一个RequestDispatcher。

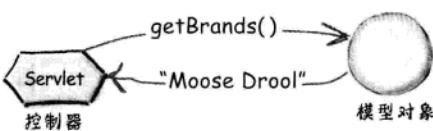
```
// code in a doGet()
BeerExpert be = new BeerExpert();
ArrayList result = be.getBrands(c);

request.setAttribute("styles", result);           ← 把模型数据放在请求作用域中。

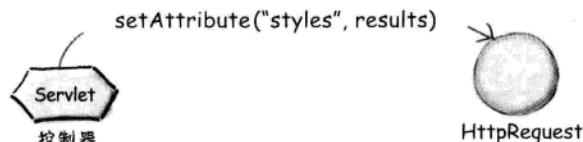
RequestDispatcher view =
    request.getRequestDispatcher("result.jsp");

view.forward(request, response);                ← 为视图JSP得到一个分派器。
                                                ← 告诉JSP接管请求，哦，对了，还要提供请求和响应对象。
```

- ① Beer servlet调用模型的getBrands()方法，返回视图需要的一些数据。



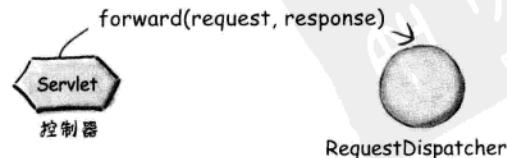
- ② servlet设置一个名为“styles”的请求属性（先把“Moose Drool”放入一个ArrayList中）。



- ③ servlet向HttpRequest要一个RequestDispatcher，并传入视图JSP的相对路径。



- ④ servlet调用RequestDispatcher的forward()，告诉JSP要接管请求（这里没有显示的是：JSP得到转发来的请求，并从请求作用域得到“styles”属性）。



# RequestDispatcher揭密

RequestDispatcher只有两个方法：forward()和include()。这两个方法都取请求和响应对象为参数（接收转发请求的组件需要这些对象来完成任务）。在这两个方法中，forward()是目前最常用的。一般不太可能从控制器servlet调用include方法，不过，在后台，JSP可能在<jsp:include>标准动作（第8章将介绍）中调用include方法。可以采用两种办法来得到RequestDispatcher：从请求得到，或者从上下文得到。不论怎么得到，都必须告诉它要把请求转发给哪个Web组件，也就是要告知接管请求的servlet或JSP。

## 从ServletRequest得到RequestDispatcher

```
RequestDispatcher view = request.getRequestDispatcher("result.jsp");
```

ServletRequest中的getRequestDispatcher()方法需要一个String路径作为参数，你要把请求转发到哪个资源，就要在这里指定这个资源的路径。如果路径最前面有一个斜线（“/”），容器就会把这个看作是“要从这个Web应用的根开始”。如果路径不是以斜线开头，则认为路径相对于原来的请求。但是别想骗过容器让它在当前Web应用之外查找。换句话说，就算是你的路径里有很多“..../..””，如果超出了当前Web应用的根，容器将无法工作。

<<interface>>
<b>RequestDispatcher</b>
<i>forward(ServletRequest, ServletResponse)</i>
<i>include(ServletRequest, ServletResponse)</i>

javax.servlet.RequestDispatcher

这是相对路径（因为最前面没有斜线（“/”））。在这种情况下，容器会在请求所“在”的逻辑位置查找“result.jsp”（相对路径和逻辑位置将在有关“部署”的一章中详细介绍）。

## 从ServletContext得到RequestDispatcher

```
RequestDispatcher view = getServletContext().getRequestDispatcher("/result.jsp");
```

与ServletRequest中的相应方法一样，这个getRequestDispatcher()方法也取一个String路径作为参数，要把请求转发给哪个资源，就要在这里指定资源的路径，但是不能指定相对于当前资源（也就是接收此请求的资源）的路径。这说明，路径必须以斜线开头！

## 在RequestDispatcher上调用forward()

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

很简单。从上下文或请求得到的RequestDispatcher知道你要把请求转发到哪个资源，其实这就是作为参数传递给getRequestDispatcher()的资源（servlet或JSP）。所以，你会说，“嘿，RequestDispatcher，请把这个请求转发给我先前告诉你的资源（在这里，就是一个JSP），就是原先我得到你的时候告诉你的那个资源。另外把请求和响应给你，因为新资源需要它们来处理请求。”

ServletContext的getRequestDispatcher()方法的参数必须使用斜线。

# 这个代码有什么问题？

你觉得有问题吗？这个RequestDispatcher代码会像你预期得那样正常工作吗？

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    response.setContentType("application/jar");
    ServletContext ctx = getServletContext();
    InputStream is = ctx.getResourceAsStream("bookCode.jar");
    int read = 0;
    byte[] bytes = new byte[1024];
    OutputStream os = response.getOutputStream();
    while ((read = is.read(bytes)) != -1) {
        os.write(bytes, 0, read);
    }
    os.flush();
    RequestDispatcher view = request.getRequestDispatcher("result.jsp");
    view.forward(request, response);
    os.close();
}
```

假设这些能正常  
工作。

你会得到一个又大又胖的IllegalStateException异常！



如果已经提交了响应，就不能再  
转发请求！

“提交响应”是指，“把响应发送给客户”。再看看代码，这里有一个严重的问题：

**os.flush();**

就是这一行会导致响应发送到客户，此时，响应已经完成，结束了。这个时候不能再转发请求，因为请求已成往事！你已经做出了响应，而且只有一次这样的机会。

所以，如果在考试时看到这样的问题，要在发送一个响应之后再转发请求，千万别被蒙骗了。容器会抛出一个IllegalStateException异常。

**问：**为什么不谈谈RequestDispatcher include()方法？

**答：**一来，考试的时候不会考。再说了，我们已经提到过，实际中一般不太会用到这个方法。但是为了满足你的好奇心，可以告诉你，include()方法会把请求发送给别人（通常是另一个servlet）来完成工作，然后再返回发送者！换句话说，include()意味着请求别人帮助处理请求，但是这不是完全的移交。只是暂时地把控制交给别人，而不是永久地让别人接管。如果使用forward()，这说明“给你了，我不会再做处理这个请求和响应的任何事情”，但是如果使用include()，则是说“我想让别人帮忙处理请求和/或响应，但是一旦它们的工作结束，我希望自己来完成请求和响应的处理（不过，在此之后，我也有可能决定再使用另一个include或forward…）”。



## 记住监听者

答案

属性监听者	<code>ServletRequestAttributeListener</code> <code>ServletContextAttributeListener</code> <code>HttpSessionAttributeListener</code>
其他生命周期监听者	<code>ServletRequestListener</code> <code>ServletContextListener</code> <code>HttpSessionListener</code> <code>HttpSessionBindingListener</code> <code>HttpSessionActivationListener</code>
所有属性监听者中的方法 (除了绑定监听者)	<code>attributeAdded()</code> <code>attributeRemoved()</code> <code>attributeReplaced()</code>
与会话相关的生命周期事件 (不包括与属性相关的事件)	会话创建时和会话撤销时 <code>sessionCreated()</code> <code>sessionDestroyed()</code>
与请求相关的生命周期事件 (不包括与属性相关的事件)	请求初始化或撤销时 <code>requestInitialized()</code> <code>requestDestroyed()</code>
与servlet上下文相关的生命周期事件 (不包括与属性相关的事件)	上下文初始化或撤销时 <code>contextInitialized()</code> <code>contextDestroyed()</code>

(注意属性监听者与这些监听者的唯一区别只是接口名中插入了“Attribute”)。

(注意，在有关“会话”的一章  
还会介绍更多其他事件)。

(注意请求和会话事件的区别，对于  
会话是`sessionCreated()`，对于请求是  
`requestInitialized()`)。



## 属性作用域

答案

练习

可访问性 (谁能看到)	作用域 (能存活多久)	适用于……
----------------	----------------	-------

Context (上下文) (不是线程安全的!)	Web应用的所有部分，包括servlet、JSP、ServletContextListener、ServletContextAttributeListener。	ServletContext的生命期，这意味着所部署应用的生命期。如果服务器或应用关闭，上下文则撤销（其属性也相应撤销）。	希望整个应用共享的资源，包括数据库连接、JNDI查找名、email地址等。
HttpSession (会话) (不是线程安全的!)	访问这个特定会话的所有servlet或JSP。要记住，会话可能从单个客户请求扩展到跨同一个客户的多个请求，这些请求可能指向不同的servlet。	会话的生命期。会话可以通过编程撤销，也可能只是因为超时而撤销（有关细节将在有关“会话管理”的一章介绍）。	与客户会话有关的资源和数据，而不只是与单个请求相关的资源。它要与客户完成一个持续的会话。购物车就是一个典型的例子。
Request (请求) (线程安全)	应用中能直接访问请求对象的所有部分。简单来说，这主要是指使用RequestDispatcher将请求转发到的JSP和Servlet，另外还有与请求相关的监听者。	请求的生命期，这说明会持续到Servlet的service()方法结束。换句话说，就是线程（栈）处理这个请求的整个生命期。	将模型信息从控制器传递到视图……或者特定于单个客户请求的任何其他数据。



# 代码贴

答案

(在DD中配置上下文参数)



&lt;web-app ...&gt;

脚 没 不 会 遗 忘

```

<servlet>
  <servlet-name>BeerTest</servlet-name>
  <servlet-class>com.wickedlysmart.BeerTester</servlet-class>
</servlet>
  
```

&lt;context-param&gt;

```

  <param-name>foo</param-name>
  <param-value>bar</param-value>
</context-param>
  
```

&lt;/web-app&gt;

这些没有用到：

</servlet-param> </init-param>

<init-param>

<servlet-param>

<init-param> 用于 servlet 初始化参数，而不是上下文初始化参数。<init-param> 只会放在<servlet>元素中。

没有<servlet-param>之类的元素。



## 第5章 模拟测验

1 使用**RequestDispatcher**时，使用哪些方法通常会导致**IllegalStateException**异常？（选出所有正确的答案）

- A. `read`
- B. `flush`
- C. `write`
- D. `getOutputStream`
- E. `getResourceAsStream`

2 关于**ServletContext**初始化参数，以下哪些说法是正确的？（选出所有正确的答案）

- A. 应当用于很少改变的数据。
- B. 应当用于经常改变的数据。
- C. 可以使用**ServletContext.getParameter(String)**访问。
- D. 可以使用**ServletContext.getInitParameter(String)**访问。
- E. 应当用于与一个特定servlet相关联的数据。
- F. 应当用于适用于整个Web应用的数据。

3 哪些类型定义了**getAttribute()**和**setAttribute()**方法？（选出所有正确的答案）

- A. HttpSession
- B. ServletRequest
- C. ServletResponse
- D. ServletContext
- E. ServletConfig
- F. SessionConfig

4 如果使用**RequestDispatcher**的**forward**或**include**方法调用一个servlet，servlet的请求对象可以用哪些方法访问容器设置的请求属性？（选出所有正确的答案）

- A. getCookies()
- B. getAttribute()
- C. getRequestPath()
- D. getRequestAttribute()
- E. getRequestDispatcher()

5 哪些调用提供了适用于整个Web应用的初始化参数的有关信息？（选出所有正确的答案）

- A. ServletConfig.getInitParameters()
- B. ServletContext.getInitParameters()
- C. ServletConfig.getInitParameterNames()
- D. ServletContext.getInitParameterNames()
- E. ServletConfig.getInitParameter(String)
- F. ServletContext.getInitParameter(String)

**6**

关于监听者，以下哪些说法是正确的？（选出所有正确的答案）

- A. 发送一个servlet响应时，**ServletResponseListener**可以用于完成一个动作。
- B. **HttpSession**超时时，**HttpSessionListener**可以用于完成一个动作。
- C. servlet上下文要关闭时，**ServletContextListener**可以用于完成一个动作。
- D. 从**ServletRequest**删除一个属性时，**ServletRequestAttributeListener**可以用于完成一个动作。
- E. 已经创建servlet上下文，而且可以为第一个请求服务时，**ServletContextAttributeListener**可以用于完成一个动作。

**7**

哪些最适合存储为会话作用域的属性？

- A. 用户输入的查询参数的一个副本。
- B. 直接返回给用户的数据库查询的结果。
- C. 系统所有Web组件使用的一个数据库连接对象。
- D. 表示刚刚登录系统的一个用户的对象。
- E. 从**ServletContext**对象获取的初始化参数的一个副本。

**8** 以下是一个**HttpServlet**中的一段代码, 这个**HttpServlet**也注册为一个**ServletRequestAttributeListener**:

```

10. public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
11.         throws IOException, ServletException {
12.     req.setAttribute("a", "b");
13.     req.setAttribute("a", "c");
14.     req.removeAttribute("a");
15. }
16. public void attributeAdded(ServletRequestAttributeEvent ev) {
17.     System.out.print(" A:" + ev.getName() + "->" + ev.getValue());
18. }
19. public void attributeRemoved(ServletRequestAttributeEvent ev) {
20.     System.out.print(" M:" + ev.getName() + "->" + ev.getValue());
21. }
22. public void attributeReplaced(ServletRequestAttributeEvent ev) {
23.     System.out.print(" P:" + ev.getName() + "->" + ev.getValue());
24. }
```

会生成怎样的日志输出?

- A. A:a->b P:a->b
- B. A:a->b M:a->c
- C. A:a->b P:a->b M:a->c
- D. A:a->b P:a->b P:a->null
- E. A:a->b M:a->b A:a->c M:a->c
- F. A:a->b M:a->b A:a->c P:a->null

**9** DD中声明一个监听者时, <**listener**>元素需要哪些子元素? (选出所有正确的答案)

- A. <**description**>
- B. <**listener-name**>
- C. <**listener-type**>
- D. <**listener-class**>
- E. <**servlet-mapping**>

---

**10**

哪些类型的对象可以存储属性？（选出所有正确的答案）

- A. **ServletConfig**
  - B. **ServletResponse**
  - C. **RequestDispatcher**
  - D. **HttpServletRequest**
  - E. **HttpSessionContext**
- 

**11**

以下哪些说法是正确的？（选出所有正确的答案）

- A. Web应用要关闭时，不能保证监听者通知的顺序。
  - B. 发生监听者关心的事件时，监听者调用的顺序是不可预计的。
  - C. 容器基于部署描述文件中的声明注册监听者。
  - D. 只有容器能够置会话无效。
- 

**12**

关于**RequestDispatcher**，以下哪些说法是正确的（假设**RequestDispatcher**不是通过**getNamedDispatcher()**得到的）？（选出所有正确的答案）

- A. **RequestDispatcher**可以用于把请求转发到另一个servlet。
  - B. **RequestDispatcher**接口中只有一个方法，即**forward()**。
  - C. 查询串中指定的用于创建**RequestDispatcher**的参数不会由**forward()**方法转发。
  - D. 接收转发请求的servlet可以通过调用**ServletRequest**的**getQueryString()**方法访问原来的查询串。
  - E. 接收转发请求的servlet可以通过调用**ServletRequest**的**getAttribute("javax.servlet.forward.query\_string")**方法来访问原来的查询串。
- 

13 对于处理servlet和线程安全，推荐以下哪种做法？

- A. 编写servlet代码来扩展**ThreadSafeServlet**。
- B. 让servlet实现**SingleThreadModel**。
- C. 对所有servlet方法调用建立日志记录。
- D. 完全使用局部变量，如果必须使用实例变量，则同步对实例变量的访问。

14 给定以下方法：

- **getCookies**
- **getContextPath**
- **getAttribute**

将以上方法与下面的类或接口匹配。注意每个方法可能会使用多次。

<b>HttpSession</b>	.....	.....	.....
<b>ServletContext</b>	.....	.....	.....
<b>HttpServletRequest</b>	.....	.....	.....

15 关于**RequestDispatcher**接口，以下哪种说法正确？（选出所有正确的答案）

- A. 在它的两个方法中，**forward()**使用更频繁。
- B. 其方法取以下参数：一个资源、一个请求和一个响应。
- C. 取决于使用哪个类的方法创建**RequestDispatcher**，指向所转发资源的路径会改变。
- D. 不论使用哪个类的方法创建**RequestDispatcher**，指向所转发资源的路径都不会改变。
- E. 如果你的servlet调用了**RequestDispatcher.forward**，它可以在调用forward之前（而不是之后）发送其自己的响应。



## 第5章 模拟测验答案

1 使用**RequestDispatcher**时，使用哪些方法通常会导致**IllegalStateException**异常？（选出所有正确的答案）。

(Servlet v2.4 167页)

- A. `read`
- B. `flush`
- C. `write`
- D. `getOutputStream`
- E. `getResourceAsStream`

响应已经提交到客户时（调用了`flush`方法），再想调用`forward`就会导致一个**IllegalStateException**异常。

2 关于**ServletContext**初始化参数，以下哪些说法是正确的？（选出所有正确的答案）。

(Servlet v2.4 31页)

- A. 应当用于很少改变的数据。
- B. 应当用于经常改变的数据。  
B是不对的，因为**ServletContext**初始化参数只在容器启动时读取。
- C. 可以使用**ServletContext.getParameter(String)**访问。  
C不对，因为不存在这个方法。
- D. 可以使用**ServletContext.getInitParameter(String)**访问。
- E. 应当用于与一个特定servlet相关的数据。  
E不对，因为每个Web应用只有一个**ServletContext**对象。
- F. 应当用于适用于整个Web应用的数据。

3 哪些类型定义了**getAttribute()**和**setAttribute()**方法？（选出所有正确的答案）。

- A. HttpSession
- B. ServletRequest
- C. ServletResponse
- D. ServletContext
- E. ServletConfig
- F. SessionConfig

(Servlet v2.4 第32, 36, 59页)

4 如果使用**RequestDispatcher**的**forward**或**include**方法调用一个servlet，  
servlet的请求对象可以用哪些方法访问容器设置的请求属性？（选出所有正确的答案）。

- A. getCookies()
- B. getAttribute()
- C. getRequestPath()
- D. getRequestAttribute()
- E. getRequestDispatcher()

B是正确的方法。利用这个方法可以访问容器设置的javax.servlet.forward.Xxx 和 javax.servlet.include.Xxxx 属性。

C和D方法根本不存在。

5 哪些调用提供了适用于整个Web应用的初始化参数的有关信息？（选出所有正确的答案）。

- A. ServletConfig.getInitParameters()
- B. ServletContext.getInitParameters()
- C. ServletConfig.getInitParameterNames()
- D. ServletContext.getInitParameterNames()
- E. ServletConfig.getInitParameter(String)
- F. ServletContext.getInitParameter(String)

(Servlet v2.4 32页)

A和B不正确，因为这些方法不存在。

C和E不正确，因为这些方法用于访问特定于servlet的初始化参数。

6

关于监听者，以下哪些说法是正确的？（选出所有正确的答案）。

(Servlet v2.4 80页)

- A. 发送一个servlet响应时，**ServletResponseListener**可以用于完成一个动作。
- B. **HttpSession**超时时，**HttpSessionListener**可以用于完成一个动作。
- C. servlet上下文要关闭时，**ServletContextListener**可以用于完成一个动作。
- D. 从**ServletRequest**删除一个属性时，**ServletRequestAttributeListener**可以用于完成一个动作。
- E. 已经创建servlet上下文，而且可以为第一个请求服务时，**ServletContextAttributeListener**可以用于完成一个动作。

A不正确，因为没有  
ServletResponseListener  
接口。

E不正确，因为此时应该用  
ServletContextListener。

7

哪些最适合存储为会话作用域的属性？

(Servlet v2.4 58页)

- A. 用户输入的查询参数的一个副本。 A不正确，因为查询参数通常直接用于完成一个操作。
- B. 直接返回给用户的数据库查询的结果。 B不对，因为这种数据通常直接返回，或者存储在请求作用域。
- C. 系统所有Web组件使用的一个数据库连接对象。 C不对，因为（由于它并非针对某个特定会话），它应当存储在上下文作用域。
- D. 表示刚刚登录系统的一个用户的对象。
- E. 从**ServletContext**对象获取的初始化参数的一个副本。 E不对，因为servlet上下文参数应当用**ServletContext**对象存储。

8 以下是一个`HttpServlet`中的一段代码, 这个`HttpServlet`也注册为一个`ServletRequestAttributeListener`:

(Servlet v2.4 199~200页)

```

10. public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
11.         throws IOException, ServletException {
12.     req.setAttribute("a", "b");
13.     req.setAttribute("a", "c");
14.     req.removeAttribute("a");
15. }
16. public void attributeAdded(ServletRequestAttributeEvent ev) {
17.     System.out.print(" A:" + ev.getName() + "->" + ev.getValue());
18. }
19. public void attributeRemoved(ServletRequestAttributeEvent ev) {
20.     System.out.print(" M:" + ev.getName() + "->" + ev.getValue());
21. }
22. public void attributeReplaced(ServletRequestAttributeEvent ev) {
23.     System.out.print(" P:" + ev.getName() + "->" + ev.getValue());
24. }

```

会生成怎样的日志输出?

- A. A:a->b P:a->b
- B. A:a->b M:a->c
- C. A:a->b P:a->b M:a->c
- D. A:a->b P:a->b P:a->null
- E. A:a->b M:a->b A:a->c M:a->c
- F. A:a->b M:a->b A:a->c P:a->null

这里有一个陷阱! 如果属性被替换, `getValue`方法返回的是原来的属性。

9 DD中声明一个监听者时, `<listener>`元素需要哪些子元素? (选出所有正确的答案)。

(Servlet v2.4  
10.4和13.4.9节)

- A. `<description>`
- B. `<listener-name>`
- C. `<listener-type>`
- D. `<listener-class>`
- E. `<servlet-mapping>`

`<listener-class>`子元素是`<listener>`元素唯一必要的子元素。

(API)

10

哪些类型的对象可以存储属性？（选出所有正确的答案）。

- A. **ServletConfig**      A、B和C不对，因为这些类型不存储属性。
- B. **ServletResponse**
- C. **RequestDispatcher**
- D. **HttpServletRequest**
- E. **HttpSessionContext**      -E不对，因为没有这种类型。

注意：另外两种与servlet相关而且可以存储属性的类型是 HttpSession 和 ServletContext。

11

以下哪些说法是正确的？（选出所有正确的答案）。

(Servlet v2.4 81~84页)

- A. Web应用要关闭时，不能保证监听者通知的顺序。
- B. 发生监听者关心的事件时，监听者调用的顺序是不可预计的。
- C. 容器基于部署描述文件中的声明注册监听者。
- D. 只有容器能够置会话无效。

A和B不对，因为容器会使用DD来确定注册监听者的通知顺序。

D是不正确的，因为servlet可以使用 HttpSession.invalidate() 方法置会话无效。

12

关于**RequestDispatcher**，以下哪些说法是正确的？（假设**RequestDispatcher**不是通过**getNamedDispatcher()**得到的）（选出所有正确的答案）。

(Servlet v2.4 65页)

- A. **RequestDispatcher**可以用于把请求转发到另一个servlet。
- B. **RequestDispatcher**接口中只有一个方法，即**forward()**。
- C. 查询串中指定的用于创建**RequestDispatcher**的参数不会由**forward()**方法转发。
- D. 接收转发请求的servlet可以通过调用**ServletRequest**的**getQueryString()**方法访问原来的查询串。
- E. 接收转发请求的servlet可以通过调用**ServletRequest**的**getAttribute("javax.servlet.forward.query\_string")**方法来访问原来的查询串。

B不对，因为这个接口还包含一个**include**方法。

C不对，因为在这种情况下会转发这种参数。

D不对，因为这个方法会从 RequestDispatcher 返回 URL 模式中的查询串。

13

对于处理servlet和线程安全，推荐以下哪种做法？

- A. 编写servlet代码来扩展**ThreadSafeServlet**。
- B. 让servlet实现**SingleThreadModel**。
- C. 对所有servlet方法调用建立日志记录。
- D. 完全使用局部变量，如果必须使用实例变量，则同步对实例变量的访问。

A和B不对，因为Servlet API中不在**ThreadSafeServlet**，规范2.4版本**SingleThreadModel**已经废弃，不建议使用。

14

给定以下方法：

(APJ)

- **getCookies**
- **getContextPath**
- **getAttribute**

将以上方法与下面的类或接口匹配。注意每个方法可能会使用多次。

<b>HttpSession</b>	.....	<b>getAttribute</b>	.....
<b>ServletContext</b>	.....	<b>getContextPath</b>	.....
<b>HttpServletRequest</b>	<b>getCookies</b>	<b>getAttribute</b>	<b>getContextPath</b>

目前不必把这些都记住，更重要的是要了解各个作用域中适用哪些方法。

15

关于**RequestDispatcher**接口，以下哪种说法正确？(选出所有正确的答案)

(APJ)

- A. 在它的两个方法中，**forward()**使用更频繁。
- B. 其方法取以下参数：一个资源、一个请求和一个响应。
- C. 取决于使用哪个类的方法创建**RequestDispatcher**，指向所转发资源的路径可能会改变。
- D. 不论使用哪个类的方法创建**RequestDispatcher**，指向所转发资源的路径都不会改变。
- E. 如果你的servlet调用了**RequestDispatcher.forward**，它可以在调用forward之前（而不是之后）发送其自己的响应。

B: 资源在对象创建时指定。

E: 如果你的servlet使用了一个RD，它绝对不能发送自己的响应。

## 6 会话管理

# 会话状态



Web服务器没有短期记忆。一旦发送了响应，Web服务器就会忘了你是谁。下一次你再做请求的时候，Web服务器不会认识你。换句话说，它们不记得你曾经做过请求，也不记得它们曾经给你发出过响应。什么都记不得了。有时这样也没关系。但有时可能需要跨多个请求保留与客户的会话状态。对于购物车，如果要求客户必须在一个请求中既做出选择又要结账，这是不合适的。对此，在Servlet API中可以找到一种极其简单的解决方法。

# OBJECTIVES



## 会话管理

内容说明：

- 4.1 编写servlet代码，将对象保存到一个会话对象中，以及从会话对象获取对象。
- 4.2 给定一个场景，描述访问会话对象使用的API，解释何时创建会话对象，并描述撤销会话对象使用的机制，以及何时撤销会话对象。
- 4.3 使用会话监听者，编写代码对会话的有关事件做出响应，包括向会话增加一个对象，以及会话对象从一个VM迁移到另一个VM。
- 4.4 给定一个场景，说明Web容器可以采用哪些会话管理机制，如何使用coockie来管理会话，以及如何使用URL重写管理会话，并编写servlet代码完成URL重写。

有关会话管理的4个考试要求在这一章中全面介绍（不过有些内容在上一章中已经所涉及）。这一章是学习和记住这些内容唯一机会。所以一定要珍惜。



我希望啤酒应用与客户之间能有来有往地对话……用户回答一个问题后，如果Web应用能根据上一个回答提出一个新的问题，这样多酷呀！



## Kim希望能够跨多个请求保留客户特定的状态

现在，模型中的业务逻辑只是检查请求中的参数，并返回一个响应（建议）。应用中没有谁记得在当前请求之前与这个客户之间发生过什么。

他现在的模型是：

```
public class BeerExpert {
    public ArrayList getBrands(String color) {
        ArrayList brands = new ArrayList();
        if (color.equals("amber")) {
            brands.add("Jack Amber");
            brands.add("Red Moose");
        } else {
            brands.add("Jail Pale Ale");
            brands.add("Gout Stout");
        }
        return brands;
    }
}
```

↑ 检查接收到的一个参数  
(color)，并返回最后的响应  
(满足这种颜色的一个  
品牌数组)。这个建议不  
是太聪明……

他希望的模型是：

```
public class BeerExpert {
    public NextResponse getAdvice(String answer) {
        // 处理客户的回答，为此
        // 查看客户以前的所有回答，
        // 以及对当前请求的回答，
        // 如果有足够的信息，就返回最后的建议，
        // 否则，返回另一个问题要求用户回答。
    }
}
```

↑ 假设NextResponse类封装了下一次为用户显示的东西。  
假设NextResponse类封装了一个指示信息，指出显示的是最后  
面的推荐建议还是一个新问题。

模型（业务逻辑）必须明确是否有足够的信息来给出推荐意见（换句话说，提供最后的建议），如果信息不够，就要向用户提出下一个问题。

## 就像真正的会话一样……

我们聚会上想来点更好的饮料。我打电话问问Kim……



喂，我在参加Joe的海滩聚会，我手里是……照我说是一杯不太好的红伞饮料……你再拿点饮料来，现在就拿来！



伞饮料？哦，真是糟糕。你这个电话打对了……我来问你几个问题——首先，你想来点深色的、琥珀色的，还是浅色的？



哦，我喜欢深色的……但是这帮人看上去好像不太喜欢太浓的，所以为稳妥起见，还是要点琥珀色的吧。



喂……我这里琥珀色的可不少呢……你关心价钱吗？



喂，老兄……如果我不需要钱，我干嘛“舍身”给一本计算机书当模特？我当然关心价钱！



没问题……我有一些外购的苦啤酒，可以发给你。



# 怎么跟踪客户回答？

除非Kim能跟踪客户在会话过程中都说过些什么，而不只是跟踪对当前请求做出的回答，否则他的设计就无法成功。他需要Servlet得到表示客户选择的请求参数，并且把它保存在某个地方。每次客户回答一个问题时，建议引擎就会利用客户以前的所有回答，要么提出另一个问题，要么提供最终的推荐意见。

有哪些选择呢？

## 使用一个有状态会话企业Java bean



当然，他可以这么做。可以让servlet成为一个有状态会话bean的客户端，每次请求到来时，就可以找到客户的有状态bean。为此需要解决很多小问题，不过当然了，确实可以使用一个有状态会话bean来保存会话状态。

但是，对于这个应用来说，这种办法实在是太牵强了（太大材小用了）！而且，Kim的提供商并没有一个带EJB容器的完整EE服务器。他只有Tomcat（只是一个Web容器），仅此而已。

## 使用一个数据库



这样也行。他的托管提供商确实允许访问MySQL，所以这么做是可以的。可以把客户的数据写到数据库里……但是这样对运行时性能造成的影响几乎与企业bean造成的影响不相上下，没准影响还更大。而且这种做法已经超出了他本来的需要。

## 使用一个HttpSession



你已经知道了。我们可以使用一个HttpSession对象保存跨多个请求的会话状态。换句话说，保存与该客户的整个会话期间的会话状态。

（实际上，即使Kim确实选择了其他的方法，如数据库或会话bean，也必须使用一个HttpSession，这是因为，如果客户是一个Web浏览器，Kim还需要将特定客户与一个特定的数据库主键或会话bean ID匹配，在这一章你将看到，HttpSession会负责这种身份认证）

**HttpSession对象可以保存跨同一个客户多个请求的会话状态。**

**换句话说，与一个特定客户的整个会话期间，**HttpSession**会持久存储。**

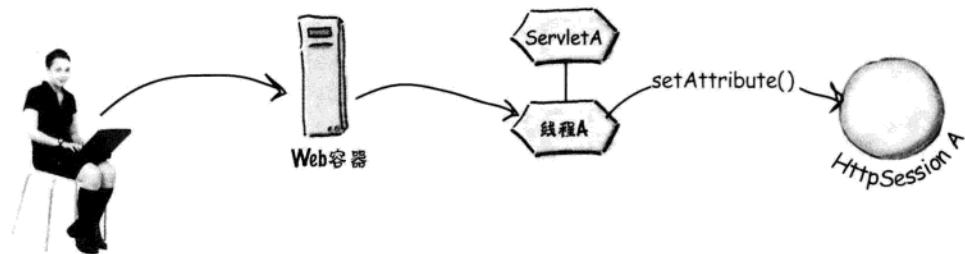
**对于会话期间客户做的所有请求，从中得到的所有信息都可以用**HttpSession**对象保存。**

# 会话如何工作

- ① Diane选择了“Dark”，并点击submit（提交）按钮。

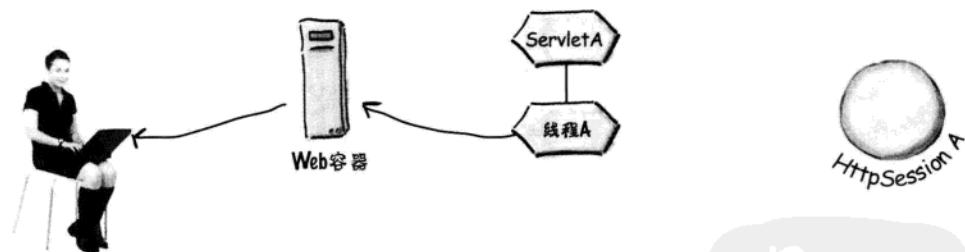
容器向BeerApp servlet的一个新线程发出请求。

BeerApp线程找到与Diane相关的会话，并把她的选择(“Dark”)作为一个属性保存在会话中。



- ②

Servlet运行其业务逻辑（包括调用模型），并返回一个响应……在这里返回了另一个问题，“价钱如何？”



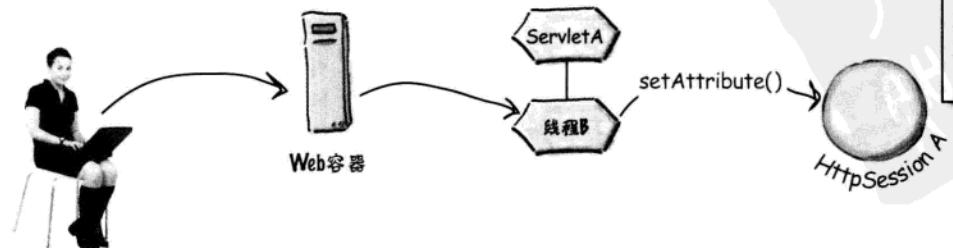
- ③

Diane先考虑页面上的新问题，选择“Expensive”，然后点击submit（提交）按钮。

容器向BeerApp servlet的一个新线程发出请求。

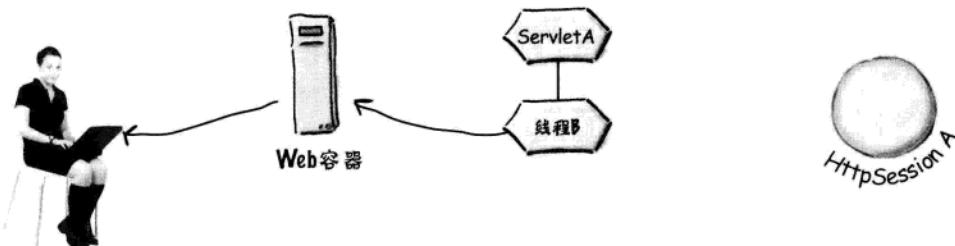
BeerApp线程找到与Diane相关的会话，并把她的选择(“Expensive”)作为一个属性保存在会话中。

相同的客户  
相同的servlet  
不同的请求  
不同的线程  
相同的会话



④

servlet运行其业务逻辑（包括调用模型），并返回一个响应……在这里又会返回另一个问题。

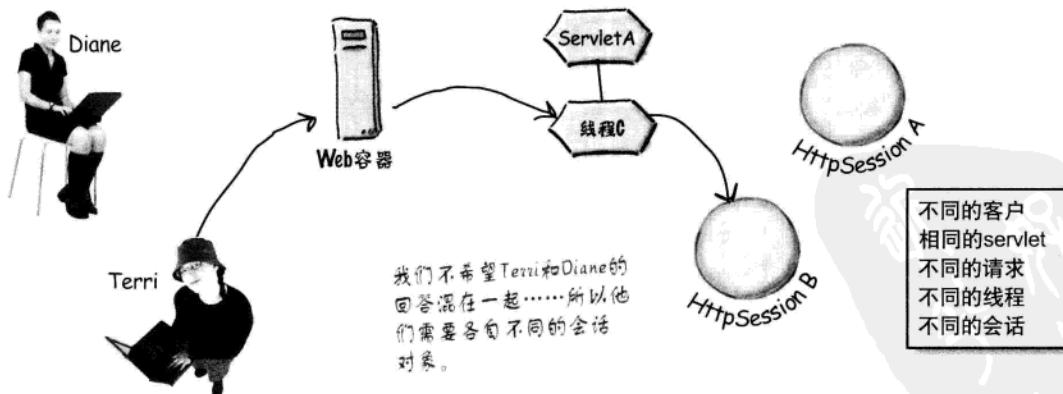


与此同时，假设又有一个客户来到啤酒网站……

⑤ Diane的会话仍是活动会话，但是此时Terri选择了“Pale”，并点击submit（提交）按钮。

容器把Terri的请求发送给BeerApp servlet的一个新线程。

BeerApp线程为Terri开始一个新的会话，并调用setAttribute()来保存她的选择（“Pale”）。

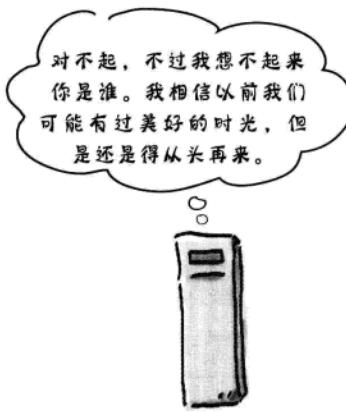


## 还有一个问题……容器怎么知道客户是谁？

HTTP协议使用的是无状态连接。客户浏览器与服务器建立连接，发出请求，得到响应，然后关闭连接。换句话说，连接只为一个请求/响应存在。

由于连接不会持久保留，所以容器认不出做第二个请求的客户与做前一个请求的客户是同一个客户。对容器而言，每个请求都来自于一个新的客户。

容器怎么认得出这是Diane, 而不是Terri?  
HTTP是无状态的，所以每个请求都是一个新连接……



*there are no Dumb Questions*

**问：**为什么容器不干脆使用客户的IP地址呢？IP地址也是请求的一部分，不是吗？

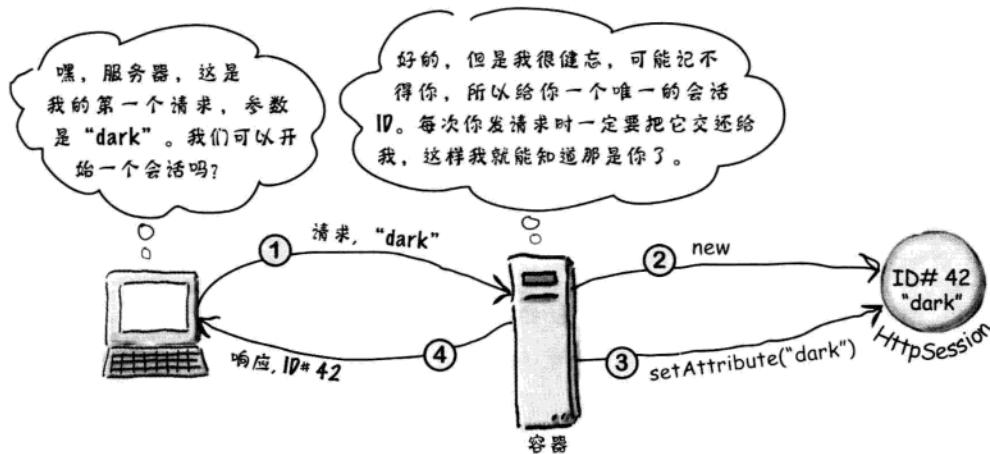
**答：**哦，容器确实可以得到请求的IP地址，但是IP地址能唯一地标识客户吗？如果你在一个局域IP网络中，就会有唯一的IP地址，但是很有可能不是外界看到的IP地址。对于服务器来说，你的IP地址是路由器的地址，所以你和这个网络中所有其他人的IP地址都是一样的！因此，靠IP地址是不行的。你会遭遇同样的问题，Jim放在他的购物车里的东西最后居然进了Pradeep的购物车，反之亦然。所以，这样不行，IP地址不是解决方法，不能唯一地标识Internet上的特定客户。

**问：**那么安全信息呢？如果用户登录，而且连接是安全的（HTTPS），容器就能准确地知道是哪一个客户，是这样吗？

**答：**对，如果用户登录，而且连接是安全的，容器就能认出客户，并把它与一个会话关联。但是，这个条件太苛刻，一般并不满足。大多数好的网站设计都指出“除非确实有必要，否则不应要求用户登录，另外除非确实有意义，否则不要使用安全连接（HTTPS）”。如果你的用户只是想浏览一下，就算是他们确实向购物车里增加商品，在用户真正决定结帐之前，你可能并不想让系统对他们进行身份认证！一方面可以减少你的开销，另一方面也能减少用户的麻烦。所以，我们需要一种合适的机制，能把客户关联到一个不需要客户安全认证的会话（你们会在……请等等……关于“安全”的一章中详细介绍！）。

## 客户需要一个唯一的会话ID

道理很简单：对客户的第一请求，容器会生成一个唯一的会话ID，并通过响应把它返回给客户。客户再在以后的每一个请求中发回这个会话ID。容器看到ID后，就会找到匹配的会话，并把这个会话与请求关联。



# 客户和容器如何交换会话ID信息？

容器必须以某种方式把会话ID作为响应的一部分交给客户，而客户必须把会话ID作为请求的一部分发回。最简单而且最常用的方式是通过cookie交换这个会话ID信息。

## Cookies



“Set-Cookie”只是响应中发送的另一个首部。

这是你的cookie，里面有会话ID……



```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=0AAB6C8DE415
Content-Type: text/html
Content-Length: 397
Date: Wed, 19 Nov 2003 03:25:40 GMT
Server: Apache-Coyote/1.1
Connection: close

<html>
...
</html>
```

HTTP响应



好，这是我的请求中的cookie。



“Cookie”只是请求中发送的另一个首部。

```
POST /select/selectBeerTaste2.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0
Cookie: JSESSIONID=0AAB6C8DE415
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,image/png,*/*;q=0.8,video/x-mng,image/jpeg,image/
gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

HTTP请求



# 最棒的是：容器几乎会做 cookie的所有工作！

你确实必须告诉容器想创建或使用一个会话，但是除此以外，生成会话ID、创建新的cookie对象、把会话ID放到cookie中、把cookie设置为响应的一部分等等工作都将由容器负责。对于后续的请求，容器会从请求中的cookie得到会话ID，将这个会话ID与一个现有的会话匹配，并把会话与当前请求关联。

## 在响应中发送一个会话cookie：

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

就这么简单。在你的服务方法中请求一个会话，余下的所有事情都会自动完成。

你不用自己建立新的 HttpSession 对象。

你不必生成唯一的会话ID。

你不用自己建立新的Cookie 对象。

你不用把会话ID与 cookie 关联。

你不用在响应中设置 Cookie（在 Set-Cookie 首部下）。

cookie 的所有工作都在后台进行。

## 从请求得到会话ID：

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

看上去是不是很熟悉？没错，这与为响应生成会话ID和cookie时所用的方法完全一样！

IF(请求包含一个会话ID cookie)

找到与该ID匹配的会话。

ELSE IF(没有会话ID cookie OR 没有与此会话ID匹配的当前会话)

创建一个新会话。

cookie 的所有工作都在后台进行。

向请求要一个会话，容器会负责余下的所有事情。你什么也不用做！  
 (这个方法不只是创建一个会话，对请求第一次调用这个方法时，会导致随后响应发送一个 cookie。现在还不能保证客户会接受这个 cookie... 不过我们先假设客户支持 cookie。)

哇！得到会话ID cookie（并把它与现有会话匹配）的方法与发送会话ID cookie 的方法完全一样。你自己不会具体看到会话ID（不过可以要求会话给你一个会话ID）。

# 我怎么知道会话是已经存在，还是刚刚创建？

这个问题问得好。请求的无参数方法getSession()会返回一个会话，而不论是否已经有一个会话。因为你总能从这个方法得到返回的一个 HttpSession 实例，所以要想知道会话是不是新创建的，只有去问会话。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test session attributes<br>");

    HttpSession session = request.getSession(); ← 不管怎么样，getSession()都会返回
    if (session.isNew()) { ← 一个会话……但是无法区别这是不是一个新会话，除非去问会话。
        out.println("If the client hasn't used this session yet, "
            + "it has not responded, so isNew() returns true.");
        out.println("This is a new session.");
    } else {
        out.println("Welcome back!");
    }
}
```

**问：**通过调用request.getSession()可以得到一个会话，但这是得到会话的唯一方法吗？能不能从ServletContext得到一个会话？

**答：**之所以可以从请求对象得到会话，这是因为……想想看……会话是由请求标识的。对请求调用getSession()时，就是在说，“我想为这个客户要一个会话……可以是与客户所发出会话ID相匹配的一个会话，也可以是一个新的会话。但是不论怎样，这个会话都只是针对与这个请求关联的客户。”

但是，确实还有另外一种办法来得到会话……从会话事件对象也能得到会话。要记住，监听者类不是servlet或JSP，它只是一个想知道发生了某些事件的类。例如，监听者可能是一个属性，想知道自己（属性对象）何时增加到一个会话，或者何时从一个会话删除。

与会话相关的监听者接口定义了事件处理方法，这些方法取一个类型为HttpSessionEvent的参数（或其子类HttpSessionBindingEvent）。HttpSessionEvent就有一个getSession()方法！

所以，如果实现了与会话相关的4种监听者接口中的任何一个接口（本章稍后将介绍），就可以通过事件处理器方法来访问会话。例如，以下代码取自一个实现 HttpSessionListener 接口的类：

```
public void sessionCreated(HttpSessionEvent event) {
    HttpSession session = event.getSession();
    // event handling code
}
```

# 如果我只想要一个已经有的会话呢？

在某些情况下，servlet可能只想使用一个原来创建的会话。例如，让结账servlet再开始一个新的会话可能就不太合适。

所以，针对这个目的，专门有一个重载的getSession(boolean)方法。如果你不想创建一个新会话，可以调用getSession(false)，调用这个方法要么得到null，要么得到一个已经有的 HttpSession。

以下的代码调用了getSession(false)，然后检查返回值是不是null。如果确实为null，代码就会输出一个消息，然后创建一个新的会话。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test sessions<br>");

    HttpSession session = request.getSession(false);
    if (session==null) { ← 现在可以测试是否已经存在一个会话（无参数getSession()绝对不会返回null）。
        out.println("no session was available");
        out.println("making one...");
        session = request.getSession(); ← 在这里可以知道，我们在建立一个新的会话。
    } else {
        out.println("there was a session!");
    }
}
```

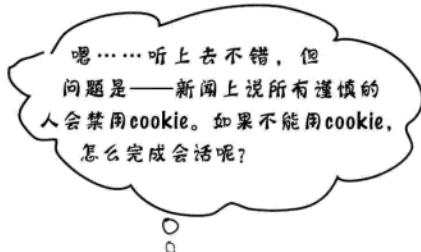
传递“false”表示，这个方法会返回一个已经有的会话。如果没有与此客户关联的会话，则返回null。

**问：** 和前一页相比，上面的代码不是很傻吗，同样是创建一个新的会话，这么做效率不是很低吗？

**答：** 你说得没错。上面的代码只是要测试两个不同版本的getSession()如何工作。在实际中，只有一种情况下可能想要使用getSession(false)，这就是你确实不想创建一个新的会话。倘若最终目的是创建一个新会话，只是在知道这是一个新会话（而不是已经存在的会话）时会做不同的响应，那么还是使用无参数的getSession()方法比较好，然后使用HttpSession isNew()方法询问会话，看它是不是一个新会话。

**问：** 这么说，getSession(true)和getSession()完全一样吗……

**答：** 你又说对了。无参数版本是一个便利方法，如果你知道反正是要一个会话，不论是新会话还是已经存在的会话，使用无参数的版本就非常合适。如果你很清楚不想要一个新会话，或者在运行时才决定是否建立一个新的会话（而且向getSession(someBoolean)方法传入了一个变量），此时有一个布尔值参数的版本就很有用。



## 就算是客户不接受cookie，你也能完成会话，但是必须稍微多做点工作……

你说所有谨慎的人都会禁用cookie，我们不同意这种说法。实际上，大多数浏览器都确实启用了cookie，而且一切都很正常。但是没有什么是万无一失的。

如果你的应用相当依赖于会话，就需要另外一种方法让客户和容器交换会话ID信息。幸运的是，容器对拒绝cookie的客户也留有一手，但是你要为此多做些工作。

如果你使用前面几页的会话代码，在请求上调用getSession()，容器就会尝试使用cookie。如果没有启用cookie，这说明客户不会加入会话。换句话说，会话的isNew()方法总会返回true。

禁用cookie的客户会忽略“Set-Cookie”响应首部

如果客户不接受cookie，你不会得到异常。没有警告，没有提示，没人告诉你想与这个客户建立会话的企图没有得逞。实际上，这只是意味着客户会忽略你用会话ID设置cookie的企图。在你的代码中，如果没有使用URL重写，这意味着getSession()总会返回一个新会话（也就是说，在这个会话上调用isNew()时总会返回“true”）。但客户不会发回一个带有会话ID cookie首部的请求。

## URL重写: 一条后路

如果客户不接受cookie，可以把URL重写作为一条后路。假设你的做法得当，URL重写就总能起作用，客户并不关心具体发生了什么，也不会采取任何行动来禁止URL重写。要记住，我们的目的是让客户和容器交换会话ID信息。要交换会话ID，来回传递cookie是最简单的方法，但是如果不能把ID放在一个cookie中，又能把它放在哪里呢？URL重写能取得置于cookie中的会话ID，并把会话ID附加到访问应用的各个URL的最后。

假设有一个Web页面，其中每个链接都有一些额外的信息（会话ID）附加到URL的最后。当用户点击这种“enhanced”（改进）链接时，到达容器的请求会在最后携带着这个额外信息，容器会取下请求URL中这个额外部分，并用它查找匹配的会话。

**URL + ;jsessionid=1234567**

**HTTP/1.1 200 OK**  
Content-Length: 397  
Date: Wed, 19 Nov 2003 03:25:40 GMT  
Server: Apache-Coyote/1.1  
Connection: close

```
<html>
<body>
<a href="http://www.wickedlysmart.com/BeerTest.do;jsessionid=0AAB6C8DE415">
click me
</a>
</body>
</html>
```

在响应发回的HTML中，把会话ID增加到所有URL的最后。

HTTP响应

会话ID放在请求URL的最后作为“额外”信息返回（不一定用分号作为分隔符，不同开发商可能采用不同的分隔符。）

**GET /BeerTest.do;jsessionid=0AAB6C8DE415**  
HTTP/1.1  
Host: www.wickedlysmart.com  
User-Agent: Mozilla/5.0  
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,image/jpeg,image/gif;q=0.2,\*/\*;q=0.1  
Accept-Language: en-us,en;q=0.5  
Accept-Encoding: gzip,deflate

HTTP请求

## 如果不能用cookie，而且只有告诉响应 要对URL编码，URL重写才能奏效

cookie不能工作时，容器就会求助于URL重写，但是只有当你额外做了一些工作，对响应中发送的所有URL完成了编码，此时URL重写才奏效。如果希望容器总是默认地先使用cookie，而URL重写只作为最后一道防线，那你大可放心。这正是容器的一般做法（不过第一次除外，这种情况稍后再做介绍）。但是，如果你没有显式地对URL编码，而且客户不接受cookie，就无法使用会话。如果确实对URL完成了编码，容器会首先尝试使用cookie来完成会话管理，只有当cookie方法失败时才会转而“投奔”URL重写。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession(); ← 得到一个会话。
    out.println("<html><body>");
    out.println("<a href=\"" + response.encodeURL("/BeerTest.do") + "\">click me</a>"); ↑
    out.println("</body></html>"); ↑
}
```

↑ 向这个URL增加额外的会话ID信息。

**问：** 等一等……容器怎么知道cookie不能正常工作呢？容器什么时候决定使用URL重写？

**答：** 如果是一个很傻的容器，它并不关心cookie是否能工作，这个傻容器每次都会尝试发送cookie，同时完成URL重写，就算是cookie能工作也不例外。但是，高级的容器会如下处理：

容器看到一个getSession()调用，而且没有从客户的请求得到会话ID，容器就知道它必须尝试与客户建立一个新的会话。此时，容器并不知道cookie是否能工作，所以向客户返回第一个响应时，它会同时尝试cookie和URL重写这两种做法。

**问：** 为什么不能先试试cookie……得不到返回的cookie时再在下一个响应中完成URL重写？

**答：** 要记住，如果容器没有从客户得到会话ID，器甚至不知道这是来自同一个客户的另一个请求。容器没有办法知道它上一次尝试过cookie，而且cookie不能正常工作。记住，容器要想确定以前是否曾经见过这个客户唯一的办法就是看客户是否发送了一个会话ID！

因此，当容器看到你调用了request.getSession()，而认识到它要与这个客户建立一个新的会话，容器就会回一个“双保险”响应，不仅针对会话ID有一个“Set-Cookie”首部，而且会向URL追加会话ID（假设使用response.encodeURL()）。

下面假设同一个客户发出了下一个请求，它把会话ID追到请求URL，但是如果客户接受cookie，这个请求还会一个会话ID cookie。servlet调用request.getSession()时，器从请求读取会话ID，找到会话，并且这样考虑：“该客户接受cookie，所以我可以忽略response.encodeURL()用。在响应中，我要发送一个cookie，因为我知道它能正常工作，而且没有必要完成任何URL重写，所以我不用心……”

## 使用sendRedirect()的URL重写

可能有这样一种情况，你想把请求重定向到另外一个URL，但是还想使用一个会话。为此有一个特殊的URL编码方法：

```
response.encodeRedirectURL("/BeerTest.do")
```

**问：** 那所有静态HTML页面呢？里面有很多[链接](#)。我怎么对这些静态页面完成URL重写？

**答：** 根本就不能对静态页面完成URL重写！使用URL重写只有一种可能，这就是作为会话一部分的所有页面都是动态生成的！显而易见，不能硬编码写会话ID，因为ID在运行时之前并不存在。所以，如果依赖于会话，就要把URL重写作为一条后路。另外，因为需要URL重写，就必须在响应HTML中动态生成URL！这意味着必须在运行时处理HTML。

不错，这里存在一个性能问题。所以你必须仔细地考虑在哪些地方会话对你的应用有意义，另外是不是必须使用会话，还是使用会话只有一点帮助。

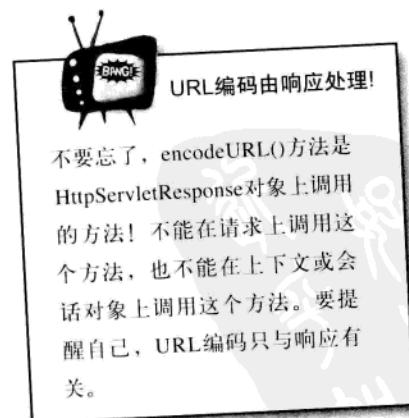
**问：** 你是说，要使用URL重写，页面就必须是动态生成的，那么这是不是意味着我可以用JSP来完成？

**答：** 对！可以在JSP中完成URL重写，甚至还有一简单的JSTL标记可以很容易地做这个工作，这就是<c:URL>，在讨论如何使用定制标记的一章中将介绍这个标记。

**问：** URL重写是不是以开发商特定的方式处理？

**答：** 没错，URL重写确实以开发商特定的方式处理。Tomcat使用分号“;”将额外的信息追加到URL。其他的开发商可能使用逗号或其他分隔符。另外，Tomcat在重写的URL中增加了“jsessionid=”，但其他开发商可能只是追加会话ID本身。关键是，不论容器使用什么分隔符，请求到来时容器都能识别。所以，当容器看到它使用的分隔符时（换句话说，就是它在URL重写时增加的分隔符），就知道在此分隔符后面的所有内容都是容器放在这里的“额外信息”。也就是说，容器知道如何识别和解析它（容器）追加到URL的额外内容。

URL重写是自动的……但是只有当你对URL完成了编码时它才奏效。必须通过响应对象的一个方法（可以是encodeURL()或encodeRedirectURL()）来运行所有URL，其他的所有事情都由容器来完成。





### 不要被请求参数“jsessionid”或“JSESSIONID”首部搞糊涂。

你不会直接用到“jsessionid”。如果看到一个“jsessionid”请求参数，说明肯定有问题。绝对不应该看到下面这样的代码：

```
String sessionId = request.getParameter("jsessionid");
```

不对!!

而且，在请求或响应中，不应该看到定制的“jsessionid”首部：

```
POST /select/selectBeerTaste.do HTTP/1.1
```

```
User-Agent: Mozilla/5.0
```

```
JSESSIONID: 0AAB6C8DE415
```

← 不能这样做！这会被认为是一个首部！

实际上，唯一可以放“jsessionid”的地方就是在cookie首部中：

```
POST /select/selectBeerTaste.do HTTP/1.1
```

这是对的，但是不要自己

```
User-Agent: Mozilla/5.0
```

这样做。

```
Cookie: JSESSIONID=0AAB6C8DE415
```

URL重写的结果

或者作为“额外信息”追加到URL的最后：

(同样不要自己)

```
POST /select/selectBeerTaste.do;jsessionid=0AAB6C8DE415
```

这样做)。



### 要点

- 在写至响应的HTML中，URL重写把会话ID增加到其中所有URL的最后。
- 会话ID作为请求URL最后的“额外”信息再通过请求返回。
- 如果客户不接受cookie，URL重写会自动发生，但是必须显式地对所有URL编码。
- 要对一个URL编码，需要调用response.encodeURL(aString)。
 

```
out.println("<a href='"
+ response.encodeURL("/BeerTest.do")
+ ">click me</a>");
```
- 没有办法对静态页面完成自动的URL重写，所以，如果你依赖于会话，就必须使用动态生成的页面。



## 删除会话

客户到来，开始一个会话，然后又改变主意，离开了这个网站。或者客户到来，开始一个会话，然后他的浏览器崩溃了。再或者客户到来，开始一个会话，然后完成一次购买交易（购物车结账）来结束会话。还有可能客户的计算机崩溃。总之有种种原因。

关键是，会话对象占用着资源。你肯定不希望会话不必要地过久保留。记住，HTTP协议没有提供任何机制让服务器知道客户是不是已经走了（如果你熟悉分布式应用，按分布式应用的说法，就是这里没有租约）（注1）。

那么容器（或你）怎么知道客户什么时候走的呢？容器怎么知道客户的浏览器崩溃了呢？容器又如何知道什么时候能安全地撤销一个会话？

↑  
(他想节省机器上的空间来玩游戏。)



## 开动脑筋

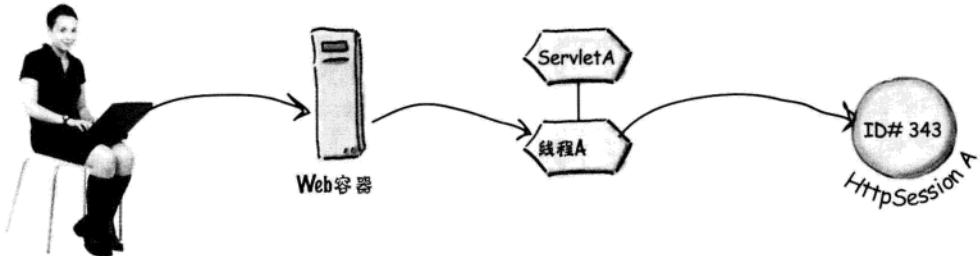
要管理会话的个数，删除不需要的会话，为此你（和容器）可以使用哪些策略？容器可以采用哪些可能的办法来通知已经不再需要某个会话？

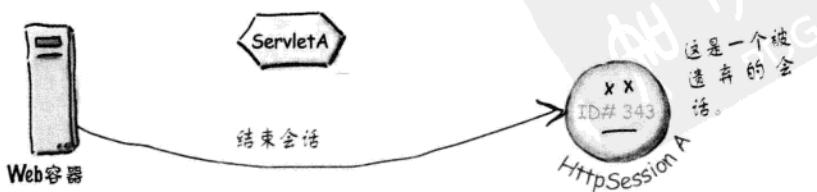
想想看，再看看后面几页的 HttpSession API 来得到一些线索。

注1：有些分布式应用使用租约，使服务器能知道客户什么时候离开。客户从服务器得到一个租约，然后必须按指定的间隔续租，告诉服务器这个客户仍存活。如果客户的租约到期，服务器就知道可以释放为这个客户保留的所有资源。

## 我们希望实现……

我们希望，如果一个会话太长时间都不活动，容器能把它识别出来，并撤销这个会话。当然，必须告诉容器“太长”的含义是什么。20分钟足够长吗？1个小时呢？一天呢？（可能我们有办法告诉容器“太长”是多长）

- ① Diane选择“Dark”，并点击submit（提交）按钮。
- 容器向BeerApp servlet的一个新线程发送请求。
- 容器建立一个新会话，ID# 343。“JSESSIONID” cookie通过响应发给Diane（这里没有显示）。
- 
- 
- ② Diane神秘地消失了。
- 容器在空闲时间完成所能做的工作（可能还有很多其他客户需要服务）。
- 为Diane建立的会话还保留着……一直在等待……它被遗弃了。

- 
- 
- ③ Diane没有回来。很长时间过去了，还是没有回来……
- 容器检查会话# 343的状态，发现已经有20分钟没有得到对该会话ID的任何请求。
- 容器说，“20分钟就太长了。她不会回来了。”然后撤销这个被遗弃的可怜会话。
- 

## HttpSession 接口

调用 `getSession()` 时，你关心的只是得到一个实现了 `HttpSession` 接口的类实例。而创建具体的实现是容器的任务。

一旦有了会话，你能用它做什么？

大多数情况下，你会使用会话来得到和设置会话作用域属性。

不过，当然还不只如此。看看你能不能自己得出这些关键方法的含义（答案在下一页上，所以先不要翻到下一页！）。



它做什么

`<<interface>>`  
`javax.servlet.http.HttpSession`

```
Object getAttribute(String)
long getCreationTime()
String getId()
long getLastAccessedTime()
int getMaxInactiveInterval()
ServletContext getServletContext()
void invalidate()
boolean isNew()
void removeAttribute(String)
void setAttribute(String, Object)
void setMaxInactiveInterval(int)
// 更多方法
```

你用它做什么

<code>getCreationTime()</code>		
<code>getLastAccessedTime()</code>		
<code>setMaxInactiveInterval()</code>		
<code>getMaxInactiveInterval()</code>		
<code>invalidate()</code>		

## 关键的 HttpSession 方法

你已经知道了与属性有关的一些方法 (`getAttribute()`、`set-Attribute()`、`removeAttribute()`)，不过下面这些关键方法在你的应用中也很可能用到（考试中也很有可能会考）。

	它做什么	你用它做什么
<code>getCreationTime()</code>	返回第一次创建会话的时间。	得出这个会话有多“老”。你可能想把某些会话的寿命限制为一个固定的时间。例如，你可能会说“一旦登录，就必须在10分钟之内完成这个表单……”
<code>getLastAccessedTime()</code>	返回容器最后一次得到包含这个会话ID的请求后过去了多长时间(毫秒数)。	得出客户最后一次访问这个会话是什么时候。可以用这个方法来确定客户是否已经离开很长时间，这样就可以向客户发出一封email，询问他们是否还回来。或者可以调用 <code>invalidate()</code> 结束会话。
<code>setMaxInactiveInterval()</code>	指定对于这个会话客户请求的最大间隔时间(秒数)。	如果已经过去了指定的时间，而客户未对这个会话做任何请求，就会导致会话被撤销。可以用这个方法减少服务器中无用的会话。
<code>getMaxInactiveInterval()</code>	返回对于这个会话客户请求的最大间隔时间(秒数)。	得出这个会话可以保持多长时间不活动但仍“存活”。可以使用这个方法来判断一个不活动的客户在会话撤销之前还有多长的“寿命”。
<code>invalidate()</code>	结束会话。当前存储在这个会话中的所有会话属性也会解除绑定（更多内容见本章后面的介绍）。	如果客户已经不活动，或者你知道会话已经结束（例如，客户完成了购物结账，或已经注销），可以用这个方法杀死（撤销）会话。会话实例本身可能由容器回收，但是这一点我们并不关心。置无效（ <code>Invalidate</code> ）意味着会话ID不再存在，而且属性会从会话对象删除。



### 开动脑筋

既然已经看到了这些方法，你能想出可以采用什么策略来撤销被遗弃的会话？

你不是说真的吧……这是不是说我得跟踪会话行为，而且必须由我来撤销无用的会话？这个工作为什么不能由容器来完成？



## 设置会话超时

好消息：你不用自己来跟踪。看到上一页的那些方法了吗？无需你使用这些方法来删除无用的（不活动的）会话。这些事情容器会为你做的。

会话有3种死法：

- ▶ 超时。
- ▶ 你在会话对象上调用invalidate()。
- ▶ 应用结束（崩溃或取消部署）。

### ① 在DD中配置会话超时

在DD中配置会话超时与在所创建的每一个会话上调用setMaxInactiveInterval()有同样的效果。

```
<web-app ...>
  <servlet>
    ...
  </servlet>
  <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</web-app>
```

“15”是指15分钟。这是说，如果客户15分钟没有对这个会话做任何请求，就杀死它。

### ② 设置一个特定会话的会话超时

如果你想改变某个特定会话实例的session-timeout值（而不影响应用中其他会话的超时时间）：

```
session.setMaxInactiveInterval(20*60);
```

只有对会话调用这个方法才有意义。

这个方法的参数以秒为单位，所以这表示如果客户20分钟没有对此会话做任何请求，就杀死它。<sup>\*</sup>

<sup>\*</sup>杀死会话，而不是客户。



DD中的超时时间  
以分钟为单位！

这里有一点很不一致，一定要当心……在DD中指定超时时，是以分钟为单位，但是在程序中设置超时时间时，单位却是秒！



## 代码贴

如果一个会话20分钟没有接收到任何请求，就应该将这个会话撤销，请通过在DD中指定并编写程序来实现这一点。我们已经在servlet中为你放了一个代码贴，你可以以此为起点，另外，并不是所有代码贴都会用到。

— DD —



— Servlet —

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

HttpSession

}



## 作为容器



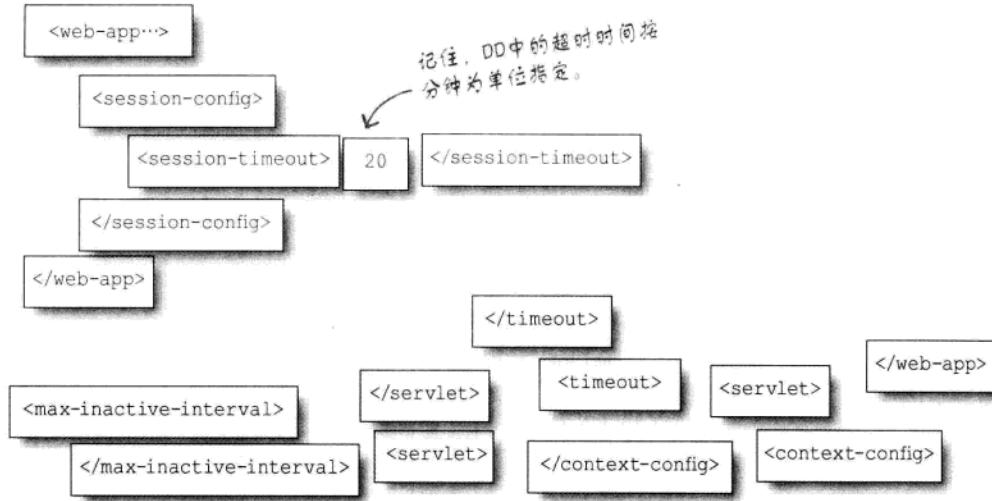
下面的两个代码清单分别是已编译  
HttpServlet的代码。你的任务是，把  
自己想成是容器，确定这些servlet被同  
一个客户调用两次时会发生什么。请说  
明同一个客户第一次和第二次  
访问servlet时会发生什么情  
况。

① public void doGet(HttpServletRequest request, HttpServletResponse response)  
throws IOException {  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    HttpSession session = request.getSession();  
    session.setAttribute("foo", "42");  
    session.setAttribute("bar", "420");  
    session.invalidate();  
    String foo = (String) session.getAttribute("foo");  
    out.println("Foo: " + foo);  
}

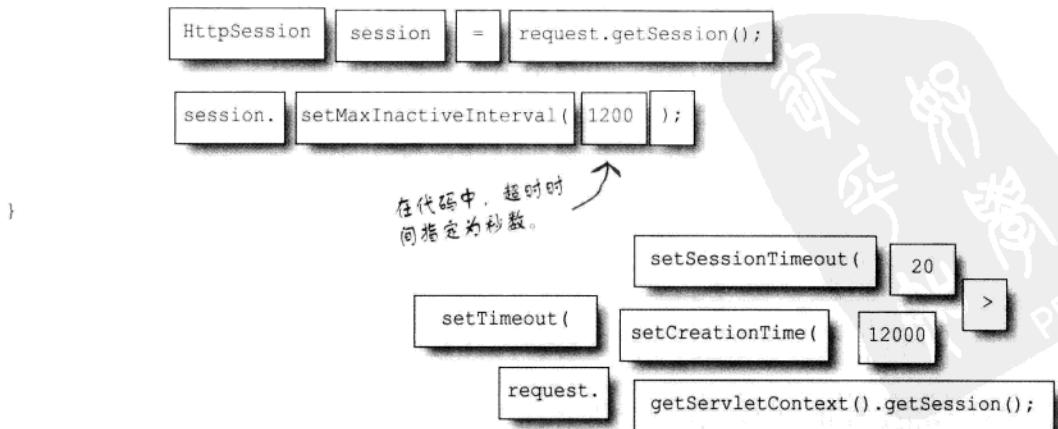
② public void doGet(HttpServletRequest request, HttpServletResponse response)  
throws IOException {  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    HttpSession session = request.getSession();  
    session.setAttribute("foo", "42");  
    session.setMaxInactiveInterval(0);  
    String foo = (String) session.getAttribute("foo");  
    if (session.isNew()) {  
        out.println("This is a new session.");  
    } else {  
        out.println("Welcome back!");  
    }  
  
    out.println("Foo: " + foo);  
}



如果一个会话20分钟没有接收到任何请求，就应该将这个会话撤销，请通过在DD中指定并编写程序来实现这一点。

**DD****Servlet**

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





## 作为容器 答案

```
① public void doGet(HttpServletRequest request, HttpServletResponse response)
                     throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setAttribute("bar", "420");
    session.invalidate(); ← 这里将会话置为无效。
    String foo = (String)session.getAttribute("foo");
    out.println("Foo: " + foo);
}
```

唉呀！在这里对会话调用  
getAttribute()就太晚了，因为  
会话已经无效了！

**结果：**会抛出一个运行时异常（IllegalStateException），因为会话已经无效，此后是无法得到属性的。

```
② public void doGet(HttpServletRequest request, HttpServletResponse response)
                     throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession();
    session.setAttribute("foo", "42");
    session.setMaxInactiveInterval(0); ← 这里让会话立即超时，因为我们指
    String foo = (String) session.getAttribute("foo");

    if (session.isNew()) { ← 对于已经置为无效的会话，不能调用isNew()。
        out.println("This is a new session.");
    } else {
        out.println("Welcome back!");
    }
    out.println("Foo: " + foo);
}
```

所以会出现以上代码同样的问题……不能在  
一个已经无效的会话上调用这个方法。

**结果：**会抛出一个运行时异常（IllegalStateException），因为会话已经无效，此后不能调用isNew()。将不活动的最大间隔时间设置为0，这说明会话会立即超时，并置为无效！

## cookie可以另作其他用处吗? cookie是不是只能用于会话?

尽管原先设计cookie是为了帮助支持会话状态，不过也可以使用定制cookie来完成其他的工作。要记住，cookie实际上就是在客户和服务器之间交换的一小段数据（一个名/值String对）。服务器把cookie发送给客户，客户做出下一个请求时再把cookie返回给服务器。

cookie的一大妙处是，用户不必介入，cookie交换是自动完成的（当然，要假设客户端支持cookie）。

默认地，cookie与会话的寿命一样长：一旦客户离开浏览器，cookie就会消失。“JSESSIONID” cookie就是如此。但是你可以让cookie活得更长一些，甚至在浏览器已经关闭后仍存活。

这样一来，即使该客户的相应会话已经到期，你的web应用仍能得到cookie信息。假设Kim希望在用户每次返回啤酒网站时显示这个用户的名字。因此第一次接收到客户的名字时会设置cookie，如果在一个请求中又得到了这个cookie，他就知道不用再询问这个客户的名字了。而且不论用户是否重启浏览器，甚至一个星期都没有访问这个网站，都没有关系！

```
HTTP/1.1 200 OK
Set-Cookie: username=TomasHirsch
Content-Type: text/html
Content-Length: 397
Date: Wed, 19 Nov 2003 03:25:40 GMT
Server: Apache-Coyote/1.1
Connection: close

<html>
...
</html>
```

服务器先发送  
第一个cookie。

```
POST /select/selectBeerTaste2.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0
Cookie: username=TomasHirsch
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/x-mng,image/png,image/
jpeg,image/gif;q=0.2,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

客户再将这个  
cookie返回。



**可以使用cookie在服务器和客户之间交换名/值String对。**

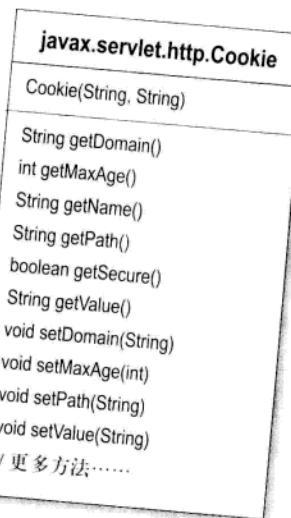
**服务器把cookie发送给客户，客户再在以后的每个请求中发回这个cookie。**

**客户的浏览器退出时，会话cookie就会消失，但是你可以告诉cookie在客户端上待得更久一些，甚至在浏览器关闭之后还持久保存。**



## 小用Servlet API使用Cookie

可以从HTTP请求和响应得到与cookie相关的首部，但最好不要这样做。对于cookie，你要做的工作都已封装在3个类的Servlet API中：HttpServletRequest、HttpServletResponse和Cookie。



### 创建一个新Cookie

```
Cookie cookie = new Cookie("username", name);
```

Cookie构造函数取一个  
名/值String对作为参数。

### 设置cookie在客户端上存活多久

```
cookie.setMaxAge(30*60);
```

*setMaxAge以秒为单位定义。这段代码是说“在客户端上存活30\*60秒”（30分钟）。如果把最大寿命设置为-1，那么浏览器退出时cookie就会消失。如果在“JSESSIONID”上调用getMaxAge()，你知道会返回什么吗？*

### 把cookie发送到客户

```
response.addCookie(cookie);
```

### 从客户请求得到cookie（或多个cookie）

```

Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getName().equals("username")) {
        String userName = cookie.getValue();
        out.println("Hello " + userName);
        break;
    }
}
    
```

没有getCookie(String)方法……你只能得到一个Cookie数组，然后必须循环处理这个数组来找到你想要的那个cookie。

## 简单的定制cookie示例

那么，假设Kim想建立一个表单，要求用户提交用户名。这个表单调用一个servlet，由servlet得到用户名请求参数，并使用用户名值在响应中设置一个cookie。

这个用户下一次请求这个Web应用中的任何servlet时，cookie会随请求返回（假设根据cookie的maxAge设置，这个cookie还存活）。Web应用中的servlet看到这个cookie时，它能把用户名放到动态生成的响应中，业务逻辑就能知道不用再向用户询问他的名字了。

针对我们描述的这种场景，下面的代码是一个简化的测试版本。

### 创建和设置cookie的Servlet

```

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

public class CookieTest extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        String name = request.getParameter("username");
        Cookie cookie = new Cookie("username", name);
        cookie.setMaxAge(30*60);
        response.addCookie(cookie);
        RequestDispatcher view = request.getRequestDispatcher("cookieresult.jsp");
        view.forward(request, response);
    }
}

```

得到表单中提交的用户名。

建立一个新cookie来存放用户名。

在客户端上存活30分钟。

将此cookie增加为“Set-Cookie”响应头部。

让JSP建立响应页。

下面这个JSP呈现以上servlet生成的视图

```
<html><body>
<a href="checkcookie.do">click here</a>
</body></html>
```

没错，这里没有多少JSP的内容，不过，就算只是这些HTML，我们也不想从servlet输出到一个JSP，但这不会改变。尽管转发到一个JSP，这时候cookie已经被设置。请求转发到JSP的时候cookie已经在响应中了……

## 定制cookie示例（续）...

得到cookie的Servlet

```

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

public class CheckCookie extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Cookie[] cookies = request.getCookies(); ← 从请求得到cookie。
        if (cookies != null) {
            for (int i = 0; i < cookies.length; i++) {
                Cookie cookie = cookies[i];
                if (cookie.getName().equals("username")) {
                    String userName = cookie.getValue(); ← 循环处理cookie数组，查找一个名
                    out.println("Hello " + userName); 为 "username" 的cookie。如果有
                    break; 这样一个cookie，得到它的值，并
                }
            }
        }
    }
}

```



法。 不用了解所有cookie方

在考试中，你不用记住Cookie类中的每一个方法，但是必须知道得到和增加Cookie的请求和响应方法。还应该知道Cookie构造函数以及getMaxAge()和setMaxAge()方法。



不要把Cookie与首部混为一谈！

向响应增加一个首部时，要把名和值String作为参数传入：

```
response.addHeader("foo", "bar");
```

但是向响应增加一个Cookie时，要传递一个Cookie对象。需要在Cookie构造函数中设置Cookie名和值。

```
Cookie cookie = new Cookie("name", name);
response.addCookie(cookie);
```

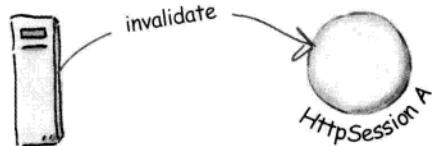
还要记住，对于首部既有setHeader()方法，又有addHeader()方法（如果已经有这个首部，addHeader会向这个现有的首部增加一个值，而setHeader会替换现有的值）。但是不存在setCookie()方法。只有一个addCookie()方法！

## HttpSession的重要里程碑

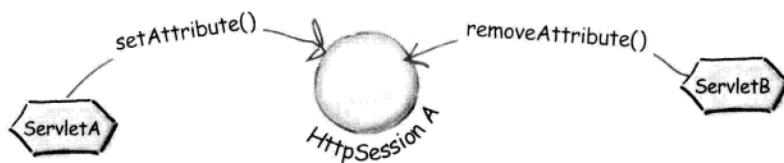
HttpSession对象一生中的重要时刻：

创建或撤销会话。

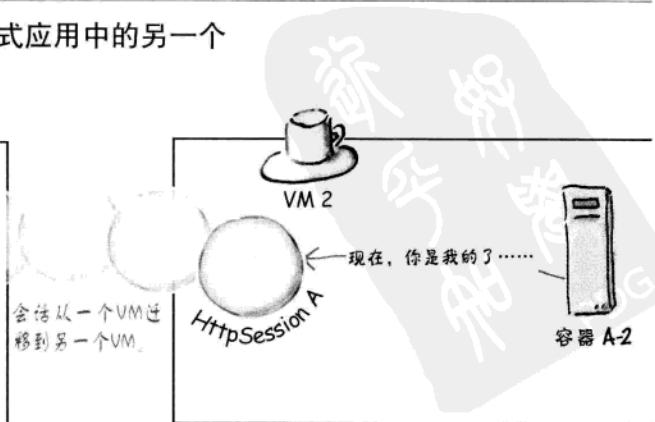
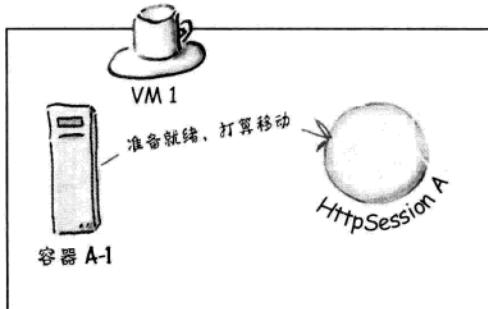
超时，或应用的某个部分对会话调用了invalidate()。



由应用的其他部分增加、删除或替换会话属性。



会话在一个VM中钝化，并在一个分布式应用中的另一个VM中激活。



# 会话生命周期事件

## 里程碑

## 事件和监听者类型

### 生命周期

#### HttpSessionEvent

#### 创建会话

容器第一次创建一个会话。此时，会话还是新的（换句话说，客户还没有用这个会话ID发送请求）。



#### 撤销会话

容器置一个会话无效（因为这个会话超时，或者应用的某个部分调用了会话的invalidate()方法）。

#### HttpSessionListener

### 属性

#### HttpSessionBindingEvent

#### 增加一个属性

应用的某个部分对会话调用setAttribute()。



#### 删除一个属性

应用的某个部分对会话调用removeAttribute()。

#### 替换一个属性

应用的某个部分对会话调用setAttribute()，而且这个属性名原来已经绑定到会话。

#### HttpSessionAttributeListener

### 迁移

#### 会话准备钝化

容器打算把会话迁移（移动）到另一个VM中。要在会话移动之前调用，这样就能让属性有机会做好迁移的准备。

#### HttpSessionEvent

#### 会话已经激活

容器已经把会话迁移（移动）到另一个VM中。要在应用的其他部分对会话调用getAttribute()之前调用，这样刚移动的属性就有机会做好准备以供访问。

#### HttpSessionActivationListener

## 不要忘了 HttpSessionBindingListener

上一页上所列的事件都是会话一生中的关键时刻。HttpSessionBindingListener则对应会话属性一生中的关键时刻。还记得在第5章，我们曾分析了如何使用这个监听者，例如，你的属性想知道自己什么时候增加到一个会话，从而与一个底层数据库同步（以及从会话删除属性时能更新数据库）。下面是对前一章的一个简单的复习：

```
package com.example;
import javax.servlet.http.*;

public class Dog implements HttpSessionBindingListener {
    private String breed;

    public Dog(String breed) {
        this.breed=breed;
    }

    public String getBreed() {
        return breed;
    }

    public void valueBound(HttpSessionBindingEvent event) {
        // 我知道我在一个会话中时要运行的代码
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // 我知道已经不在一个会话中时要运行的代码
    }
}
```



会话绑定监听者不在DD中配置！

如果一个属性类（如这里的Dog类）实现了HttpSessionBindingListener，当这个类的一个实例增加到一个会话或从会话删除时，容器就会调用事件处理回调方法（valueBound()和valueUnbound()）。这就么简单。这个工作会自动完成。但是前一页上与会话相关的其他监听者并非如此。HttpSessionListener、HttpSessionAttributeListener和HttpSessionActivationListener必须在DD中注册，因为它们与会话本身相关，而不是与会话中放置的单个属性相关。

这个监听者在javax.servlet.http包中。

这一次Dog属性同时还是一个HttpSessionBindingListener...监听Dog自己什么时候增加到会话中，或从会话删除。

"Bound"（绑定）一词是指有人把这个属性增加到一个会话。

你自己可以看出"Unbound"（解除绑定）是什么意思。



这个监听者很简单，只是让我自己知道我什么时候放入一个会话中（或者从会话中取出）。它不会告诉我有关其他会话事件的任何事情。

## 会话迁移

应该还记得，我们在前一章简单地讨论了分布式Web应用，即应用的各个部分可以复制在网络中的多个节点上。在一个集群环境中，容器可能会完成负载平衡，取得客户的请求，把请求发送到多个JVM上（这些JVM可能在相同的物理主机上，也可能在不同的物理主机上，这一点我们并不关心）。关键是，应用将位于多个位置上。

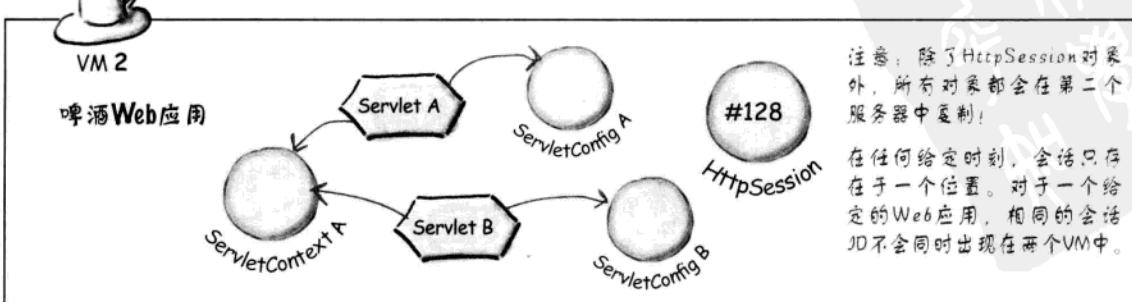
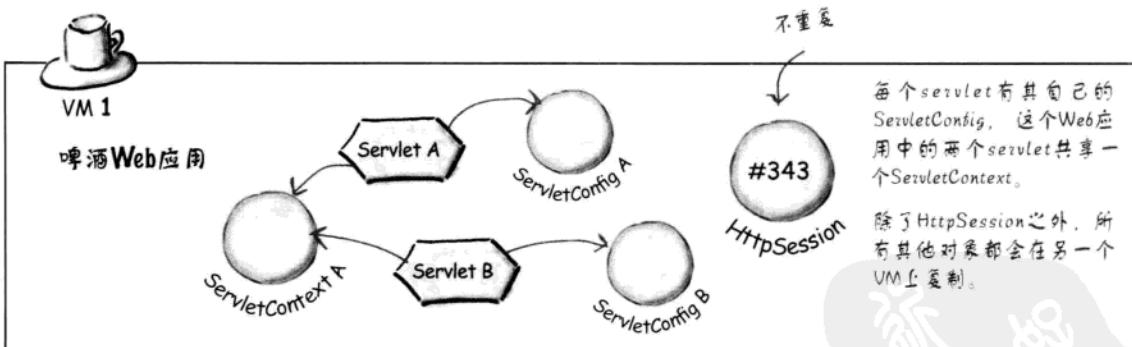
这说明，每次客户请求时，最后有可能到达同一个servlet的不同实例。换句话说，指向Servlet A的请求A可能在一个VM中完成，而指向Servlet A的请求B可能在另一个不同的VM中完成。所以，现在的问题是，ServletContext、ServletConfig和 HttpSession对象会有什么变化？

答案很简单，但意义很重大：

只有HttpSession对象（及其属性）会从一个VM移到另一个VM。

每个VM中有一个ServletContext。每个VM上的每个servlet有一个ServletConfig。但是对于每个Web应用的一个给定的会话ID，只有一个HttpSession对象，而不论应用分布在多少个VM上。

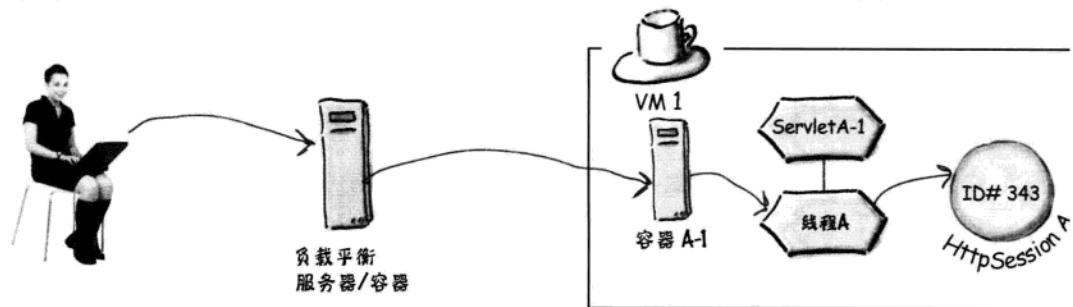
### 啤酒Web应用分布在两个VM上



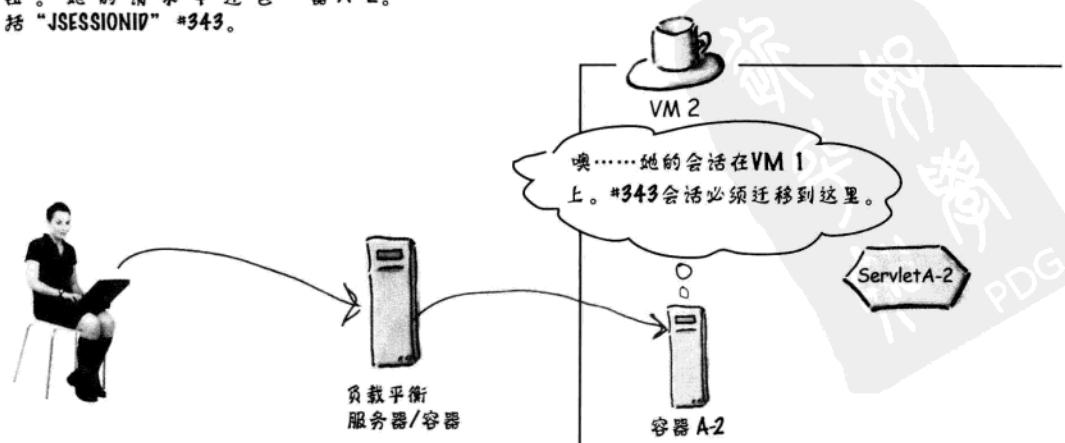
## 会话迁移实战

应用服务器开发商如何处理集群和Web应用分布，在这方面每个开发商的做法都有所不同，而且J2EE规范中并没有保证开发商必须支持分布式应用。但是，从这里给出的图，你应该能对具体的工作有一个更深层次的认识。关键是，尽管应用的其他部分会在每个节点/VM上复制，但会话对象不会复制，而只是移动。这一点是肯定的。换句话说，如果开发商不支持分布式应用，容器就必须跨VM迁移会话，这还包括迁移会话属性。

- ① Diane 选择“Pale”，并点击 submit（提交）按钮。
- 负载平衡服务器决定向 VM 1 中的容器 A-1 发送请求。



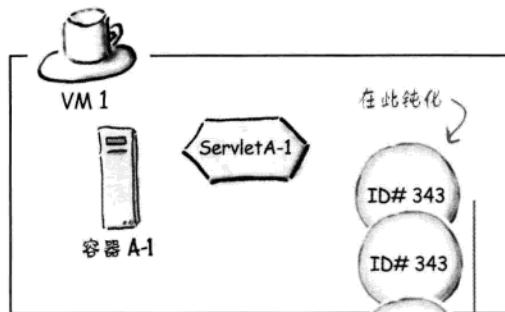
- ② Diane 选择“Bitter”，并点击 submit（提交）按钮。她的请求中还包括“JSESSIONID”#343。
- 这一次，负载平衡服务器决定把请求发送给 VM 2 中的容器 A-2。



③

会话#343从VM 1迁移到VM 2。换句话说，一旦移到VM 2，则不再存在于VM 1中。

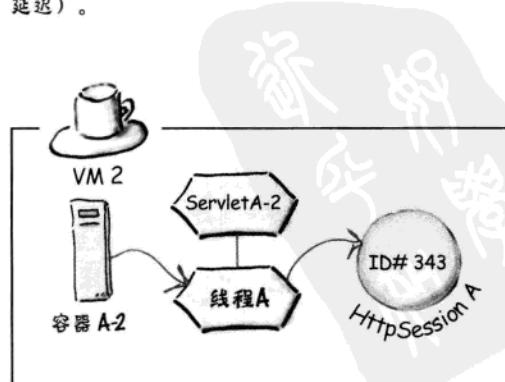
这个迁移意味着，会话在VM 1上钝化，并在VM 2上激活。



④

容器为Servlet A建立一个新线程，并把新请求与刚迁移来的会话#343关联。

Diane的新请求发送给这个线程，皆大欢喜。Diane并不知道发生了什么（只不过等待会话迁移时稍稍有点延迟）。



## HttpSessionActivationListener使属性 做好准备大搬家……

因为 HttpSession 有可能从一个 VM 迁移到另一个 VM，规范设计者认为，如果有人能告诉会话中的属性它们也可能移动，这样会好些。这样一来，属性就可以确保它们能完成这次旅程。

如果你的属性都是直接的 Serializable 对象，并不关心它们最后会放在哪里，那么可能不会用到这个监听者。实际上，我们认为 95.324% 的 Web 应用都不会使用这个监听者。但是这个监听者确实在“恭候”着，需要时随时可以使用，而且这个监听者最常见的用法是为属性提供一个机会，使它的实例变量准备好串行化。

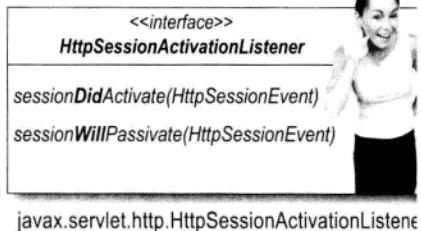
### 会话迁移和串行化

现在有点难了……

容器需要迁移 Serializable 属性（假设属性中的所有实例变量都要么是 Serializable，要么为 null）。

但是容器不一定非得使用串行化（Serialization）来迁移 HttpSession 对象！

这是什么意思？很简单：确保你的属性类类型是 Serializable，那就不用再操心了。但是如果属性类类型不是 Serializable（可能因为属性对象的某个实例变量不是 Serializable），可以让属性对象类实现 HttpSessionActivationListener，并使用激活/钝化回调方法解决这个问题。



javax.servlet.http.HttpSessionActivationListener



容器不一定非得使用串行化，所以不能保证会对一个 Serializable 属性或其某个实例变量调用 readObject() 和 writeObject()！

如果你熟悉串行化，就会知道，实现了 Serializable 的类还可以实现一个 writeObject() 方法，串行化对象时就由 VM 调用这个方法，另外还可以实现一个 readObject() 方法，这个方法在对象逆串行化时调用。 Serializable 对象可以使用这些方法来做一些事情，例如，在串行化期间把非 Serializable 的字段设置为 null (writeObject())，然后在逆串行化期间再恢复这些字段 (readObject())（如果你对串行化的细节不熟悉，也不用担心）。

但是在会话迁移中不一定会调用这些方法！所以，如果你想保存和恢复属性中的实例变量状态，可以使用 HttpSessionActivationListener，并使用两个事件回调方法 (sessionDidActivate()) 和 sessionWillPassivate()，就像使用 readObject() 和 writeObject() 一样。

## Listener例子

在后面这三页上,请注意事件对象类型,还要注意监听者同时也  
是一个属性类。

### 会话计数器

这个监听者允许你跟踪这个Web应用中活动会话的个数。

非常简单。

```
package com.example;
import javax.servlet.http.*;

public class BeerSessionCounter implements HttpSessionListener {
    static private int activeSessions;

    public static int getActiveSessions() {
        return activeSessions;
    }

    public void sessionCreated(HttpSessionEvent event) {
        activeSessions++;
    }

    public void sessionDestroyed(HttpSessionEvent event) {
        activeSessions--;
    }
}
```

这个类会像所有其他web-app类一样部署在WEB-INF/classes目录中,这样所有servlet和其他辅助类都能访问这个方法。

这些方法取一个  
HttpSessionEvent参数。

在DD中配置监听者:

```
<web-app ...>
...
<listener>
    <listener-class>
        com.example.BeerSessionCounter
    </listener-class>
</listener>
</web-app>
```

参考——如果这个应用分布在多个JVM上,这也  
不能正确工作,因为没有办法保证静态变量同步。如  
果类在多个JVM上加载,每个类中这个静态计数器变  
量都会各有自己的值。

## 监听者例子

### 属性监听者

利用这个监听者，每一次向会话增加属性、删除属性或替换属性时你都能跟踪到。

```
package com.example;
import javax.servlet.http.*;

public class BeerAttributeListener implements HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent event) {
        String name = event.getName();    }    // 利用 HttpSessionBindingEvent，你可以
        Object value = event.getValue();  }    // 得到触发此事件的属性的名和值。
                                            ↓

        System.out.println("Attribute added: " + name + ":" + value);
    }

    public void attributeRemoved(HttpSessionBindingEvent event) {
        String name = event.getName();
        Object value = event.getValue();
        System.out.println("Attribute removed: " + name + ":" + value);
    }

    public void attributeReplaced(HttpSessionBindingEvent event) {
        String name = event.getName();
        Object value = event.getValue();
        System.out.println("Attribute replaced: " + name + ":" + value);
    }
}
```

### 在DD中配置监听者

```
<web-app...>
...
<listener>
    <listener-class>
        com.example.BeerAttributeListener
    </listener-class>
</listener>
</web-app>
```

这个监听者使用了不一致的命名。  
这是一个属性 (Attribute) 监听者，  
但是取的是一个绑定 (Binding) 事  
件。

**问：**嘿，你到底要打印到哪里？Web中  
System.out会输出到哪里去？

**答：**会输出到容器选择的任何发送目标  
(可能允许你配置、也可能不允许你配置)。  
换句话说，可能是开发商特定的位置、通常是一个日志文件。Tomcat就会把输出放在tomcat  
logs/catalina.log中。要阅读你的服务器文档来  
了解容器如何处理标准输出。

## 监听者例子

属性类（监听对它有影响的事件）

这个监听者允许属性跟踪可能对属性本身很重要的事件，如增加到会话，或从会话删除，以及会话从一个VM迁移到另一个VM。

```
package com.example;
import javax.servlet.http.*;
import java.io.*;

public class Dog implements HttpSessionBindingListener,
    HttpSessionActivationListener,Serializable {
    private String breed;
    // 假设这里还有更多实例变量，包括一些
    // 非Serializable的实例变量

    // 假设这里是构造函数以及其他获取方法和设置方法

    public void valueBound(HttpSessionBindingEvent event) {
        // 我知道我在一个会话中时要运行的代码
    }

    public void valueUnbound(HttpSessionBindingEvent event) {
        // 我知道已经不在一个会话中时要运行的代码
    }

    public void sessionWillPassivate(HttpSessionEvent event) {
        // 这些代码将非Serializable字段置为某种状态,
        // 以便顺利地迁移到一个新VM
    }

    public void sessionDidActivate(HttpSessionEvent event) {
        // 这些代码用于恢复字段……取消
        // 在sessionWillPassivate()中做的动作
    }
}
```

会话绑定事件。

会话激活事件（但是注意这些方法取一个 HttpSessionEvent参数）。

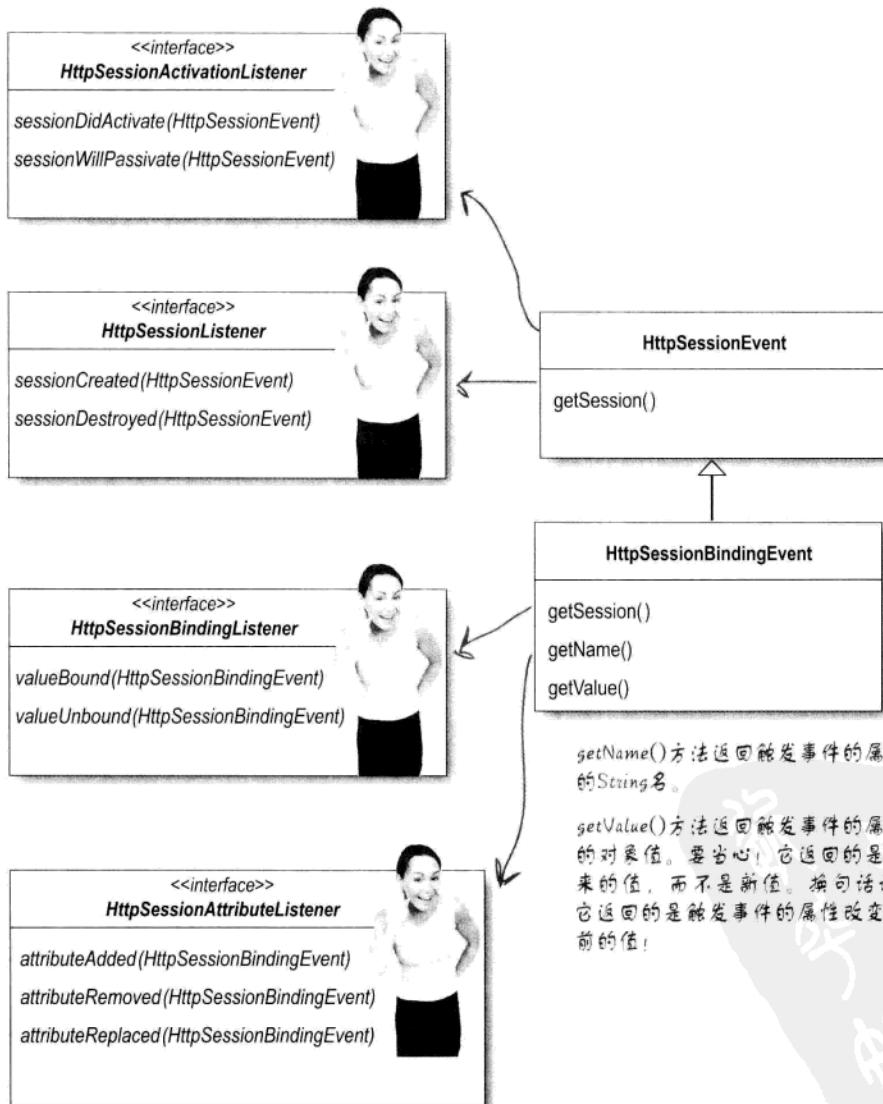
## 与会话相关的监听者

场景	监听者接口/方法	事件类型	通常由谁实现
你想知道有多少个并发用户。也就是说，你想跟踪活动的会话。	<b>HttpSessionListener</b> (javax.servlet.http)  <i>sessionCreated</i> <i>sessionDestroyed</i>	HttpSessionEvent	<input type="checkbox"/> 属性类 <del>属性类</del> 其他类
你想知道会话何时从一个VM移到另一个VM。	<b>HttpSessionActivationListener</b> (javax.servlet.http)  <i>sessionDidActivate</i> <i>sessionWillPassivate</i>	HttpSessionEvent  注意：没有特定的 <u>HttpSessionActivationEvent</u> 。	<del>属性类</del> <del>其他类</del>
有一个属性类（这个类的对象要用作为—个属性值），而且你希望此类对象绑定到会话或从会话删除时得到通知。	<b>HttpSessionBindingListener</b> (javax.servlet.http)  <i>valueBound</i> <i>valueUnbound</i>	HttpSessionBindingEvent	<del>属性类</del> <input type="checkbox"/> 其他类
你想知道会话中什么时候增加、删除或替换会话属性。	<b>HttpSessionAttributeListener</b> (javax.servlet.http)  <i>attributeAdded</i> <i>attributeRemoved</i> <i>attributeReplaced</i>	HttpSessionBindingEvent  注意：没有特定的 <u>HttpSessionAttributeEvent</u> 。	<input type="checkbox"/> 属性类 <del>属性类</del> 其他类

与会话相关的一些事件并不遵循事件命名约定！

 **HttpSessionListener**方法取 HttpSessionEvent 参数。  
**HttpSessionBindingListener**方法取 HttpSessionBindingEvent 参数。  
但是**HttpSessionAttributeListener**方法的参数是 HttpSessionBindingEvent。  
**HttpSessionActivationListener**方法的参数是 HttpSessionEvent。由于 HttpSessionEvent 和 HttpSessionBindingEvent 类能很好地满足要求，所以 API 中没有必要再增加另外两个事件类。

# 会话相关事件监听者和事件对象API概览



`getName()`方法返回触发事件的属性的String名。

`getValue()`方法返回触发事件的属性的对象值。要当心！它返回的是原来的值，而不是新值。换句话说，它返回的是触发事件的属性改变之前的值！



## 与会话相关的监听者

没错，这与两页前的表几乎完全相同，所以不要翻到前面去看。自己好好想想这些监听者，尽可能合理地猜测，并把你的猜测写下来。考试中有关会话监听者的问题至少有2个，甚至会有4个。仔细地回忆，充分运用你的常识来填写这个表。

场景	监听者接口/方法	事件类型	通常由谁实现
你想知道什么时候创建一个会话。			<input type="checkbox"/> 属性类 <input type="checkbox"/> 其他类
属性想知道什么时候迁移到一个新VM。			<input type="checkbox"/> 属性类 <input type="checkbox"/> 其他类
属性想知道什么时候在会话中被替换。			<input type="checkbox"/> 属性类 <input type="checkbox"/> 其他类
你希望在属性绑定到会话时得到通知。			<input type="checkbox"/> 属性类 <input type="checkbox"/> 其他类

注：只有两个Event（事件）对象类型。



## 第6章 模拟测验

1 给定：

```

10. public class MyServlet extends HttpServlet {
11.     public void doGet(HttpServletRequest request,
12.                         HttpServletResponse response)
13.         throws IOException, ServletException {
14.             // request.getSession().setAttribute("key", "value");
15.             // ((HttpSession)request.getSession()).setAttribute("key", "value");
16.             // (( HttpSession )request.getHttpSession()).setAttribute("key", "value");
17.         }
18. }
```

哪些行的注释可以去掉，而且不会导致编译器或运行时错误？（选出所有正确的答案）

- A. 只有第13行。
- B. 只有第14行。
- C. 只有第15行。
- D. 只有第16行。
- E. 第13行或第15行。
- F. 第14行或第16行。

2 如果客户不接受cookie，Web容器可以采用哪一种会话管理机制？（只有一个正确答案）

- A. Cookie，但不用URL重写。
- B. URL重写，但不用cookie。
- C. cookie或URL重写都可以使用。
- D. cookies和URL重写都不能使用。
- E. Cookie和URL重写必须一同使用。

---

3 关于 HttpSession 对象，以下哪些说法是正确的？（选出所有正确的答案）

- A. 会话的超时时间设置为 -1，则永远不会到期。
- B. 一旦用户关闭所有浏览器窗口，会话就会立即无效。
- C. 在 servlet 容器定义的超时时间之后，会话将无效。
- D. 可以调用 `HttpSession.invalidateSession()` 显式地置会话无效。

---

4 以下哪些类型不是 J2EE 1.4 API 中的监听者事件类型？（选出所有正确的答案）

- A. `HttpSessionEvent`
- B. `ServletRequestEvent`
- C. `HttpSessionBindingEvent`
- D. `HttpSessionAttributeEvent`
- E. `ServletContextAttributeEvent`

---

5 关于会话跟踪，以下哪些说法是正确的？（选出所有正确的答案）

- A. 服务器可以使用 URL 重写作为会话跟踪的基础。
- B. SSL 有一个内置的机制，servlet 容器可以使用这种机制来得到定义会话所用的数据。
- C. 使用 cookie 完成会话跟踪时，对于会话跟踪 cookie 的名字没有任何限制。
- D. 使用 cookie 完成会话跟踪时，会话跟踪 cookie 的名字必须是 `JSESSIONID`。
- E. 如果用户在浏览器中禁用了 cookie，容器可以使用 `javax.servlet.http.CookielessHttpSession` 对象来跟踪用户的会话。



6

给定：

```

1. import javax.servlet.http.*;
2. public class MySessionListener
3.     implements HttpSessionListener {
4.         public void sessionCreated() {
5.             System.out.println("Session Created");
6.         }
7.         public void sessionDestroyed() {
8.             System.out.println("Session Destroyed");
9.         }

```

这个类有什么错误？（选出所有正确的答案）。

- A. 第3行上的方法签名不正确。
- B. 第6行上的方法签名不正确。
- C. import语句没有导入**HttpSessionListener**接口。
- D. **HttpSessionListener**接口中不只是定义了**sessionCreated**和**sessionDestroyed**方法。

7

关于会话属性，以下哪些说法是正确的？（选出所有正确的答案）

- A. **HttpSession.getAttribute(String)**的返回类型是**Object**。
- B. **HttpSession.getAttribute(String)**的返回类型是**String**。
- C. 绑定到会话的属性可以由属于同一个ServletContext而且处理同一会话中某个请求的其他servlet访问。
- D. 在一个**HttpSession**上调用**setAttribute("keyA", "valueB")**时，如果这个会话中对应键**keyA**已经有一个值，就会导致抛出一个异常。
- E. 在一个**HttpSession**上调用**setAttribute("keyA", "valueB")**时，如果这个会话中对应键**keyA**已经有一个值，则会导致这个属性原先的值被**String valueB**替换。

---

8 哪些接口定义了**getSession()**方法? (选出所有正确的答案)

- A. **ServletRequest**
- B. **ServletResponse**
- C. **HttpServletRequest**
- D. **HttpServletResponse**

---

9 给定一个会话对象s, 以及以下代码:

```
s.setAttribute("key", value);
```

哪个监听者会得到通知? (只有一个正确答案)

- A. 仅**HttpSessionListener**
- B. 仅**HttpSessionBindingListener**
- C. 仅**HttpSessionAttributeListener**
- D. **HttpSessionListener**  
和**HttpSessionBindingListener**
- E. **HttpSessionListener**  
和**HttpSessionAttributeListener**
- F. **HttpSessionBindingListener**  
和**HttpSessionAttributeListener**
- G. 这三个监听者都会得到通知

---

10 给定**req**是一个**HttpServletRequest**, 哪个代码会在不存在会话的情况下创建一个会话? (选出所有正确的答案)

- A. **req.getSession();**
- B. **req.getSession(true);**
- C. **req.getSession(false);**
- D. **req.createSession();**
- E. **req.getNewSession();**
- F. **req.createSession(true);**
- G. **req.createSession(false);**

11

给定一个会话对象s，有两个属性，属性名分别为myAttr1和myAttr2，哪个代码段会把这两个属性从会话中删除？（选出所有正确的答案）

- A. s.removeAllValues();
- B. s.removeAttribute("myAttr1");  
s.removeAttribute("myAttr2");
- C. s.removeAllAttributes();
- D. s.getAttribute("myAttr1", UNBIND);  
s.getAttribute("myAttr2", UNBIND);
- E. s.getAttributeNames(UNBIND);

12

关于分布式环境中的HttpSession对象，以下哪些说法是正确的？（选出所有正确的答案）

- A. 会话从一个JVM移到另一个JVM时，存储在会话中的所有属性都会丢失。
- B. 会话从一个JVM移到另一个JVM时，适当注册的HttpSession-BindingListener对象会得到通知。
- C. 会话从一个JVM移到另一个JVM时，实现了HttpSessionActivationListener接口的会话属性会得到通知。
- D. 会话从一个JVM移到另一个JVM时，实现了java.io.Serializable的属性值会转移到新的JVM。

13

关于会话超时，以下哪些说法是正确的？（选出所有正确的答案）

- A. DD中的会话超时声明可以按秒为单位指定时间。
- B. DD中的会话超时声明可以按分钟为单位指定时间。
- C. 通过程序设置的会话超时声明只能按秒为单位指定时间。
- D. 通过程序设置的会话超时声明只能按分钟为单位指定时间。
- E. 通过程序设置的会话超时声明既能按秒为单位也能按分钟为单位指定时间。

---

14 以下哪个servlet代码段能从请求获取“ORA\_UID” cookie的值? (选出所有正确的答案)

- A. `String value = request.getCookie("ORA_UID");`
- B. `String value = request.getHeader("ORA_UID");`
- C. 

```
javax.servlet.http.Cookie[] cookies =
    request.getCookies();
String cName = null;
String value = null;
if (cookies != null){
    for (int i = 0; i < cookies.length; i++){
        cName = cookies[i].getName();
        if (cName != null &&
            cName.equalsIgnoreCase("ORA_UID")){
            value = cookies[i].getValue();
        }
    }
}
```
- D. 

```
javax.servlet.http.Cookie[] cookies =
    request.getCookies();
if (cookies.length > 0){
    String value = cookies[0].getValue();
}
```

---

15 以下哪些方法可以用来要求容器在会话将要超时时通知你的应用? (选出所有正确的答案)

- A. `HttpSessionListener.sessionDestroyed`
- B. `HttpSessionBindingListener.valueBound`
- C. `HttpSessionBindingListener.valueUnbound`
- D. `HttpSessionBindingEvent.sessionDestroyed`
- E. `HttpSessionAttributeListener.attributeRemoved`
- F. `HttpSessionActivationListener.sessionWillPassivate`

16

在servlet中如何使用HttpServletResponse对象为客户增加一个cookie?

- A. <context-param>
 

```
<param-name>myCookie</param-name>
<param-value>cookieValue</param-value>
```
- B. response.addCookie("myCookie", "cookieValue");
- C. javax.servlet.http.Cookie newCook =
 

```
new javax.servlet.http.Cookie("myCookie", "cookieValue");
//...set other Cookie properties
response.addCookie(newCook);
```
- D. javax.servlet.http.Cookie[] cookies = request.getCookies();
 

```
String cname = null;
if (cookies != null){
    for (int i = 0; i < cookies.length; i++){
        cname = cookies[i].getName();
        if (cname != null &&
            cname.equalsIgnoreCase("myCookie")){
            out.println( cname + ":" + cookies[i].getValue());
        }
    }
}
```

7

给定:

```
13. public class ServletX extends HttpServlet {
14.     public void doGet(HttpServletRequest req, HttpServletResponse resp)
15.         throws IOException, ServletException {
16.     HttpSession sess = new HttpSession(req);
17.     sess.setAttribute(attr1, value);
18.     sess.invalidate();
19.     String s = sess.getAttribute(attr1);
20. }
21. }
```

会得到什么结果? (选出所有正确的答案)

- A. 编译失败。
- B. s的值为null。
- C. s的值为"value"。
- D. 抛出一个IOException异常。
- E. 抛出一个ServletException异常。
- F. 抛出一个IllegalStateException异常。



## 第6章 模拟测验答案

1 给定：

```

10. public class MyServlet extends HttpServlet {
11.     public void doGet(HttpServletRequest request,
12.                         HttpServletResponse response)
13.         throws IOException, ServletException {
14.     // request.getSession().setAttribute("key", "value");
15.     // ((HttpSession)request.getSession()).setAttribute("key", "value");
16.     // (( HttpSession )request.getHttpSession()).setAttribute("key", "value");
17. }
18. }
```

(Servlet规范59页)

哪些行的注释可以去掉，而且不会导致编译器或运行时错误？（选出所有正确的答案）。

- A. 只有第13行。
- B. 只有第14行。
- C. 只有第15行。
- D. 只有第16行。
- E. 第13行或第15行。
- F. 第14行或第16行。

E是对的，因为第13行和第15行都是正确的方法调用。没有必要强制转换为HttpSession，不过这样确实可以反映正确的类型，所以这是合法的。

2

如果客户不接受cookie，Web容器可以采用哪一种会话管理机制？（只有一个正 确答案） (Servlet v2.4 57页)

- A. Cookie，但不用URL重写。
- B. URL重写，但不用cookie。
- C. cookie或URL重写都可以使用。
- D. cookies和URL重写都不能使用。
- E. Cookie和URL重写必须一同使用。

B是对的，因为不能使用cookie，但是URL重写不依赖于是否启用cookie。

3

关于**HttpSession**对象，以下哪些说法是正确的？（选出所有正确的答案）

(Servlet v2.4 59页)

- A. 会话的超时时间设置为-1，则永远不会到期。
- B. 一旦用户关闭所有浏览器窗口，会话就会立即无效。
- C. 在servlet容器定义的超时时间之后，会话将无效。
- D. 可以调用**HttpSession.invalidateSession()**显式地置会话无效。

B是不正确的，因为HTTP协议中没有明确的终止信号。

D是不对的，因为此时应当调用的方法是**invalidate()**。

4

以下哪些类型不是J2EE 1.4 API中的监听者事件类型？（选出所有正确的答案）。

(API)

- A. **HttpSessionEvent**
- B. **ServletRequestEvent**
- C. **HttpSessionBindingEvent**
- D. **HttpSessionAttributeEvent**
- E. **ServletContextAttributeEvent**

**HttpSessionBindingEvent**用于**HttpSessionBindingListener**和**HttpSessionAttributeListener**。

5

关于会话跟踪，以下哪些说法是正确的？（选出所有正确的答案）

(Servlet v2.4 57页)

- A. 服务器可以使用URL重写作为会话跟踪的基础。
- B. SSL有一个内置的机制，servlet容器可以使用这种机制来得到定义会话所用的数据。
- C. 使用cookie完成会话跟踪时，对于会话跟踪cookie的名字没有任何限制。
- D. 使用cookie完成会话跟踪时，会话跟踪cookie的名字必须是 JSESSIONID。
- E. 如果用户在浏览器中禁用了cookie，容器可以使用**javax.servlet.http.CookielessHttpSession**对象来跟踪用户的会话。

C不对，因为规范指出会话跟踪cookie必须是 JSESSIONID。

E不对，因为没有这样一个类。

6

给定：

(Servlet规范276页)

```

1. import javax.servlet.http.*;
2. public class MySessionListener
3.     implements HttpSessionListener {
4.     public void sessionCreated() {
5.         System.out.println("Session Created");
6.     }
7.     public void sessionDestroyed() {
8.         System.out.println("Session Destroyed");
9.     }

```

这个类有什么错误？（选出所有正确的答案）。

应该选择A和B，因为这两个方法应当有一个HttpSessionEvent参数。

- A. 第3行上的方法签名不正确。  
 B. 第6行上的方法签名不正确。  
 C. import语句没有导入**HttpSessionListener**接口。  
 D. **HttpSessionListener**接口中不只是定义了**sessionCreated**和**sessionDestroyed**方法。

不应选择C，因为监听者就定义在导入的包中。

不应选择D，因为这个接口中只有这两个方法。

7

关于会话属性，以下哪些说法是正确的？（选出所有正确的答案）

(Servlet规范59页)

- A. **HttpSession.getAttribute(String)**的返回类型是**Object**。  
 B. **HttpSession.getAttribute(String)**的返回类型是**String**。  
 C. 绑定到会话的属性可以由属于同一个**ServletContext**而且处理同一会话中某个请求的其他servlet访问。  
 D. 在一个**HttpSession**上调用**setAttribute("keyA","valueB")**时，如果这个会话中对应键**keyA**已经有一个值，就会导致抛出一个异常。  
 E. 在一个**HttpSession**上调用**setAttribute("keyA","valueB")**时，如果这个会话中对应键**keyA**已经有一个值，则会导致这个属性原先的值被**String valueB**替换。

B是不对的，因为返回类型是**Object**。

D是不对的，因为这个作用只会替换现有的值。

8

哪些接口定义了`getSession()`方法？（选出所有正确的答案）

(Servlet规范243页)

- A. `ServletRequest`
- B. `ServletResponse`
- C. `HttpServletRequest`
- D. `HttpServletResponse`

9

给定一个会话对象s，以及以下代码：

(Servlet规范80页)

```
s.setAttribute("key", value);
```

哪个监听者会得到通知？（只有一个正确答案）

- A. 仅`HttpSessionListener`
- B. 仅`HttpSessionBindingListener`
- C. 仅`HttpSessionAttributeListener`
- D. `HttpSessionListener`  
和`HttpSessionBindingListener`
- E. `HttpSessionListener`  
和`HttpSessionAttributeListener`
- F. `HttpSessionBindingListener`  
和`HttpSessionAttributeListener`
- G. 这三个监听者都会得到通知

F是对的，因为只要增加一个属性，`HttpSessionAttributeListener`就会得到通知，另外如果值对象实现了`HttpSessionBindingListener`接口，这个值对象也会得到通知。

10

给定`req`是一个`HttpServletRequest`，哪个代码会在不存在会话的情况下

(API)

下创建一个会话？（选出所有正确的答案）

- A. `req.getSession();`
- B. `req.getSession(true);`
- C. `req.getSession(false);`
- D. `req.createSession();`
- E. `req.getNewSession();`
- F. `req.createSession(true);`
- G. `req.createSession(false);`

如果不存在会话，A和B都会创建一个新的会话。如果会话不存在，`getSession(false)`会返回一个null。

**11** 给定一个会话对象s，有两个属性，属性名分别为myAttr1和myAttr2，哪个代码段会把这两个属性从会话中删除？（选出所有正确的答案）。 (APJ)

- A. s.removeAllValues();
- B. s.removeAttribute("myAttr1");  
s.removeAttribute("myAttr2");
- C. s.removeAllAttributes();
- D. s.getAttribute("myAttr1", UNBIND);  
s.getAttribute("myAttr2", UNBIND);
- E. s.getAttributeNames(UNBIND);

B是正确的，removeAttribute()是从会话对象删除属性的唯一途径，而且它一次只能删除一个属性。

**12** 关于分布式环境中的 HttpSession 对象，以下哪些说法是正确的？（选出所有正确的答案）。 (Servlet规范60页)

- A. 会话从一个JVM移到另一个JVM时，存储在会话中的所有属性都会丢失。
- B. 会话从一个JVM移到另一个JVM时，适当注册的 HttpSession-BindingListener 对象会得到通知。
- C. 会话从一个JVM移到另一个JVM时，实现了 HttpSessionActivationListener 接口的会话属性会得到通知。
- D. 会话从一个JVM移到另一个JVM时，实现了 java.io.Serializable 的属性值会转移到新的JVM。

A是不对的，因为可串行化属性可以转移。

B不对，因为属性仍绑定至会话。

**13** 关于会话超时，以下哪些说法是正确的？（选出所有正确的答案） (APJ)

- A. DD中的会话超时声明可以按秒为单位指定时间。
- B. DD中的会话超时声明可以按分钟为单位指定时间。
- C. 通过程序设置的会话超时声明只能按秒为单位指定时间。
- D. 通过程序设置的会话超时声明只能按分钟为单位指定时间。
- E. 通过程序设置的会话超时声明既能按秒为单位也能按分钟为单位指定时间。

在DD中使用<session-timeout>只能指定分钟数；使用HttpSession.setMaxInactiveInterval()，则只能秒数。

14

以下哪个servlet代码段能从请求获取“ORA\_UID” cookie的值? (选出所有正确的答案) (APJ)

- A. String value = request.getCookie("ORA\_UID"); A中的方法不存在。
- B. String value = request.getHeader("ORA\_UID");
- C. javax.servlet.http.Cookie[] cookies =  

```
request.getCookies();
String cName = null;
String value = null;
if (cookies != null){
    for (int i = 0; i < cookies.length; i++){
        cName = cookies[i].getName();
        if (cName != null &&
            cName.equalsIgnoreCase("ORA_UID")){
            value = cookies[i].getValue();
        }
    }
}
```

C使用request.getCookies()得到一个Cookie数组，然后检查是否有指定名的Cookie。
- D. javax.servlet.http.Cookie[] cookies =  

```
request.getCookies();
if (cookies.length > 0){
    String value = cookies[0].getValue();
}
```

D只是查看数组中的第一个Cookie。

15

以下哪些方法可以用来要求容器在会话将要超时时通知你的应用? (选出所有正确的答案) (APJ)

- A. HttpSessionListener.sessionDestroyed
  - B. HttpSessionBindingListener.valueBound
  - C. HttpSessionBindingListener.valueUnbound
  - D. HttpSessionBindingEvent.sessionDestroyed
  - E. HttpSessionAttributeListener.attributeRemoved
  - F. HttpSessionActivationListener.sessionWillPassivate
- C这是一种迂回方法，不过如果有  
一个属性类，确实可以采用这种方  
法得到超时通知。

D不存在这个方法。

E删除一个属性与会话超  
时没有紧密联系。

F会话钝化不同于会话超时。

16

在servlet中如何使用HttpServletResponse对象为客户端增加一个cookie?

(APJ)

 A. <context-param>

```
<param-name>myCookie</param-name>
<param-value>cookieValue</param-value>
/<context-param>
```

 B. response.addCookie("myCookie","cookieValue");

C. javax.servlet.http.Cookie newCook =  
 new javax.servlet.http.Cookie("myCookie","cookieValue");  
 //...set other Cookie properties  
 response.addCookie(newCook);

D. javax.servlet.http.Cookie[] cookies = request.getCookies();  
 String cname = null;  
 if (cookies != null){  
 for (int i = 0; i < cookies.length; i++){  
 cname = cookies[i].getName();  
 if (cname != null &&  
 cname.equalsIgnoreCase("myCookie")){  
 out.println( cname + ":" + cookies[i].getValue());  
 }
 }
 }

B不对，因为addCookie方法取一个  
Cookie对象作为参数，而不是String。

D不对，因为这是获取cookie的servlet代码，  
而不是创建一个cookie。

17

给定：

(APJ)

```
13. public class ServletX extends HttpServlet {
14.   public void doGet(HttpServletRequest req, HttpServletResponse resp)
15.     throws IOException, ServletException {
16.   HttpSession sess = new HttpSession(req);
17.   sess.setAttribute(attr1, value);
18.   sess.invalidate();
19.   String s = sess.getAttribute(attr1);
20. }
21. }
```

会得到什么结果？（选出所有正确的答案）

 A. 编译失败。

A第16行有问题。要使用req.getSession()来得到一个实现  
了HttpSession的对象。

 B. s的值为null。 C. s的值为"value"。 D. 抛出一个IOException异常。 E. 抛出一个ServletException异常。 F. 抛出一个IllegalStateException异常。

# 作为JSP



JSP变成servlet。这个servlet不用你来创建。容器会查看你的JSP，把它转换成Java源代码，再编译成完整的Java servlet类。但是，我们必须知道，JSP中的代码转换成Java代码时到底发生了什么。你可以在JSP中写Java代码，但是应该这样做吗？如果不写Java代码，那写什么呢？JSP代码怎么转换成Java代码？在这一章中，我们将看到6种不同类型的JSP元素，每种元素都有自己的用途，当然了，也有各自不同的语法。你将了解如何编写JSP，为什么编写JSP，还有在JSP中写些什么。也许更重要的是，你将了解在JSP中哪些不该写。

# OBJECTIVES

---

## JSP技术模型

- 6.1** 识别或描述以下元素，或为以下元素编写JSP代码：(a) 模板文本，(b) 脚本元素 (注释、指令、声明、scriptlet和表达式)，(c) 标准动作和定制动作，以及(d)表达式语言元素。
  
- 6.2** 编写使用以下指令的JSP代码：(a)page（带有属性import、session、contentType和isELIgnored），(b)include以及(c)taglib。
  
- 6.3** 编写使用了正确语法的JSP文档（基于XML的文档）。
  
- 6.4** 描述页面生命周期的作用和事件序列：(1)JSP页面转换，(2)JSP页面编译，(3)加载类，(4)创建实例，(5)调用jspInit方法，(6)调用\_jspService方法以及(7)调用jspDestroy方法。
  
- 6.5** 给定一个设计目标，使用适当的隐式对象编写JSP代码：(a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext以及(i) exception。
  
- 6.6** 配置部署描述文件来声明一个或多个标记库，禁用计算（表达式）语言，以及禁用脚本语言。
  
- 6.7** 给定一个特定的设计目标，要求将一个JSP片段包含在另一个页面中，编写JSP代码使用最合适的包含机制 (include指令或jsp:include 标准动作)。

### 内容说明：

大部分要求都将在这一章介绍，不过(c) 标准动作和定制动作以及(d) 表达式语言元素将在后面的几章再详细说明。

page 指令将在这一章介绍，但 include和taglib指令到后面的几章再介绍。

不在这一章介绍；请参考有关“部署”的一章。

都在这一章介绍（提示：一旦了解了这一章介绍的基础知识，就会发现这是实际考试中最简单的题目了）。

都在这一章介绍，不过根据前面两章的介绍，你可能已经知道了其中大多数隐式对象的含义。

除了声明标记库的内容外，其他内容都在这里介绍。声明标记库将在“使用JSTL”一章中再做说明。

不在这一章介绍，请参考下一章（“使用脚本的JSP”）。

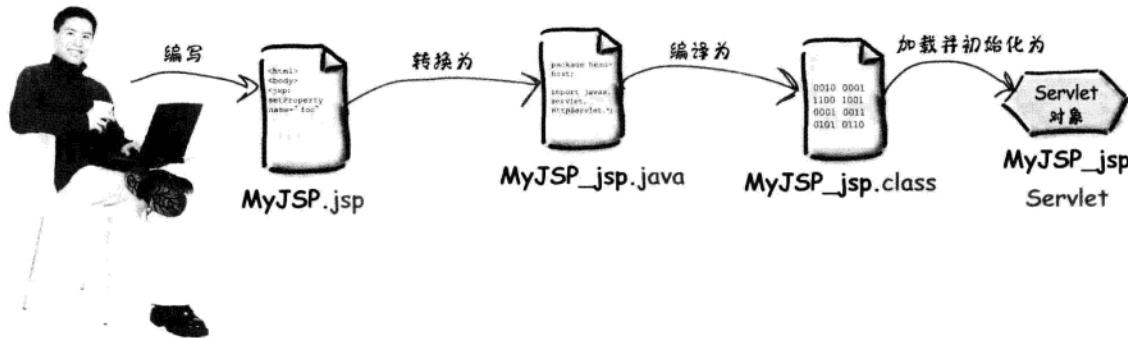
# 最后，JSP变成一个servlet

你的JSP最终会变成一个完整的servlet在Web应用中运行。它与其他的servlet非常相似，只不过这个servlet类会由容器为你编写。

容器拿到你在JSP中写的代码，把这些代码转换为一个servlet类源文件(.java)，然后再把这个源文件编译为Java servlet类。在此之后，这就是名符其实的servlet了，而且这个servlet的运行并无特别之处，就好像它的代码都是你自己编写和编译的一样。换句话说，容器会加载这个servlet类，实例化并初始化，为每个请求建立一个单独的线程，并调用servlet的service()方法。

这一章最重要的一点其实很简单：你的JSP代码在最后的servlet类中扮演什么角色？

换句话说，JSP中的元素最后会放在所生成servlet相应源代码中的什么位置？



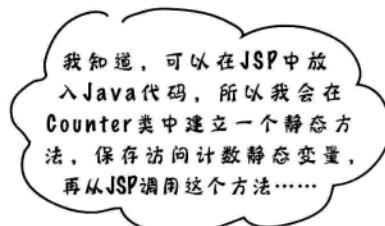
在这一章中我们要回答下列问题：

- ▶ JSP文件中的各个部分最后会放在servlet源代码中的什么位置？
- ▶ 能访问JSP页面的“servlet性”吗？例如，JSP中有ServletConfig或ServletContext的概念吗？
- ▶ JSP中可以放哪些类型的元素？
- ▶ JSP中的不同元素有什么语法？
- ▶ JSP的生命周期是怎样的？能介入到JSP生命周期中吗？
- ▶ JSP中的不同元素在最后的servlet中如何交互？

# 建立一个JSP显示被访问了多少次

Pauline想在她的Web应用中使用JSP，她实在是厌倦了，不想把HTML写到servlet的PrintWriter println()中。

她决定学JSP，建立一个简单的动态页面，打印出请求这个页面的次数。她了解到，在JSP中可以使用scriptlet放入常规的Java代码，所谓scriptlet，就是放在`<% ... %>`标记中的Java代码。



BasicCounter.jsp

```
<html>
<body>
The page count is:
<%
    out.println(Counter.getCount());
%>
</body>
</html>
```

“out”对象是隐式的。介于`<% 和 %>`之间的所有代码就是scriptlet，它们只是普通的java代码。

Counter.java

```
package foo;

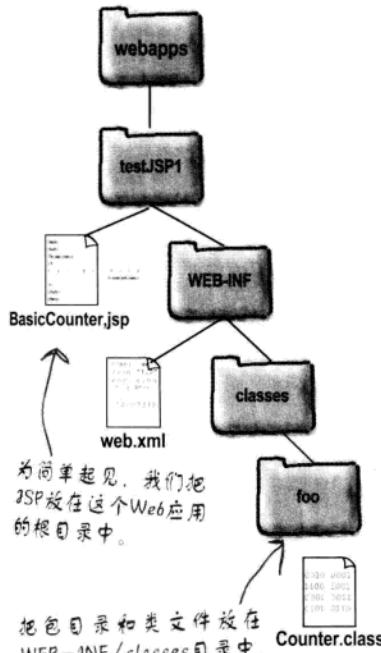
public class Counter {
    private static int count;
    public static synchronized int getCount() {
        count++;
        return count;
    }
}
```

普通的Java辅助类。

# 她完成了部署和测试

部署和测试非常简单。唯一要注意的是应当确保JSP能访问Counter类，实际上这也很容易，只要保证Counter类在Web应用的WEB-INF/classes目录中就行了。她在浏览器中直接访问这个JSP，地址是：<http://localhost:8080/testJSP1/BasicCounter.jsp>

她想看到这样的结果：



为简单起见，我们把  
JSP放在这个Web应用  
的根目录中。

把目录和类文件放在  
WEB-INF/classes目录中。  
这个Web应用的其他部分  
就能看到它了。



她实际得到的结果是：



## JSP不认识Counter类

Counter类在foo包中，但是JSP无法了解这一点。其他Java代码也同样有这个问题，你应该知道这样一条原则：要么导入包，要么在代码中使用完全限定类名。



Counter.java

```
package foo;  
  
public class Counter {  
    private static int count;  
    public static int getCount() {  
        count++;  
        return count;  
    }  
}
```

JSP 代码原先是：

```
<% out.println(Counter.getCount()); %>
```

JSP 代码应当是：

```
<% out.println(foo.Counter.getCount()); %>
```

现在就可以了。

但是你可以在  
JSP中放import语  
句……只需一个指令  
就行了。



## 使用page指令导入包

指令为你提供了一条途径，可以在页面转换时向容器提供特殊的指示。

指令有3种：page、include和taglib。include和taglib 指令将在后面几章介绍，现在我们只考虑page指令，因为这个指令允许你导入包。

要导入单个包：

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is:
<%
    out.println(Counter.getCount());
<%
</body>
</html>
```

← 这是一个带有import属性的page指令。

(注意指令的最后没有分号)。



要导入多个包：

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

↑  
使用逗号来分隔多个包。整个包列表要用引号引起！

注意到打印计数器的Java代码与page指令有什么差别吗？

Java代码放在带百分号的尖括号中间：<%和%>。而指令会为元素开始记号再增加一个字符：@!

如果看到以<%@开始的JSP代码，应该知道这是一个指令（本书后面还会更详细地介绍page指令）。

## 不过Kim又提到了“表达式”

你可能觉得已经很妥当了，不过Kim注意到scriptlet里有一个out.println()语句。这可是一个JSP呀，老兄！为什么要引入JSP，部分原因就是为了避免println()！也正是因为这个原因才有了JSP表达式元素，表达式元素会自动打印放在标记之间的内容。



### Scriptlet代码:

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is:
<% out.println(Counter.getCount()); %>
</body>
</html>
```

### 表达式代码

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is now:
<%= Counter.getCount() %>
</body>
</html>
```

表达式更简短一些，我们不需要显式地进行打印……

注意到scriptlet代码的标记与表达式的标记有什么不同吗？scriptlet代码介于带百分号的尖括号中间：`<%和%>`。而表达式会为元素的开始记号再增加一个字符：一个等号(`=`)。

到目前为止，我们已经看到了3种不同类型的JSP元素：

Scriptlet:           `<%    %>`

指令:              `<%@   %>`

表达式:            `<%=   %>`

喂！如果你想告诉我们怎么改进代码，至少应该把Java语法写对吧……表达式的最后居然没有分号！

分号到哪儿去了？



<%= Counter.getCount() %>

## 表达式会成为out.print()的参数



换句话说，容器拿到你在<%= 和%>之间键入的所有内容，并把它作为参数传递给打印语句，打印到隐式响应PrintWriter out。

容器看到这个表达式时：

<%= Counter.getCount() %>

会把它转换为：

out.print(Counter.getCount());

如果在表达式中放了分号：

<%= Counter.getCount(); %>

可能很糟糕。这表示：

out.print(Counter.getCount());

哎呀！这样就无法编译了。

**不要在表达式的最后加上分号！**

<%= neverPutASemicolonInHere %>

<%= becauseThisIsAnArgumentToWrite() %>

## there are no Dumb Questions

**问：** 好吧，如果你想使用表达式，而不是在scriptlet中放out.println()，为什么还有隐式“out”呢？

**答：** 你可能不会在JSP页面中直接使用隐式out变量，但是有可能会把它传递给“别人”……应用中的其他对象，它们不能直接访问输出流来得到响应。

**问：** 在表达式中，如果方法没有返回任何结果会发生什么情况？

**答：** 会得到一个错误！不能、绝对不能把一个返回类型为void的方法用作为表达式。容器很聪明，它知道，如果方法的返回类型为void，就打印不出任何东西！

**问：** 为什么import指令前面有“page”？为什么是<%@ page import...%>，而不是直接的<%@ import...%>？

**答：** 这个问题问得好！JSP规范中并没有一大堆各种各样的指令，而只有3个JSP指令而已，但是这些指令可以有属性。你所说的“import指令”实际上是“page指令的import属性”。

**问：** page指令的其他属性呢？

**答：** 要记住，page指令的目的是为容器提供一些信息，容器把你的JSP转换为servlet时需要用到这些信息。我们关心的属性（除了import之外）还有session、contentType和isELIgnored（这一章后面还会再来谈这些属性）。



确定以下哪些表达式是不合法的，并说明原因。我们还没有把这些例子的所有相关内容都讲到，所以请根据你对表达式工作原理的了解，做出最合理的猜测（本章后面有答案，所以你一定要现在完成这个练习）。

合法吗？（检查是否合法，如果不合法请解释原因）

- <%= 27 %>
- <%= ((Math.random() + 5)\*2); %>
- <%= "27" %>
- <%= Math.random() %>
- <%= String s = "foo" %>
- <%= new String[3] %>
- <%= 42\*20 %>
- <%= 5 > 3 %>
- <%= false %>
- <%= new Counter() %>

# Kim放了最后一个炸弹……

你甚至根本不需要  
Counter类……所有工作都  
能在JSP中完成。



嗯……我知道JSP会转换成  
一个servlet，所以也许我可以  
在scriptlet中声明一个count变量，  
它就能转换成servlet中的一个变  
量。这样可以吗？



她想这样做：

```
<html>
<body>
<% int count=0; %>
The page count is now:
<%= ++count %>
</body>
</html>
```

这样能编译吗？

能正常工作吗？

PDG

## 在scriptlet中声明一个变量

变量声明是合法的，但是不会像Pauline希望的那样工作。

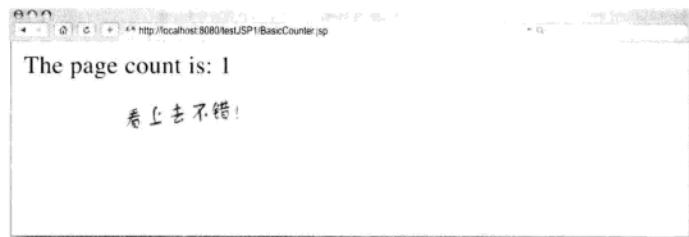
她想这样试一试：

```
<html>
<body>
<scriptlet> <% int count=0; %>      ← 声明count变量。
    The page count is now:
    <%= ++count %>
</body>
</html>
```

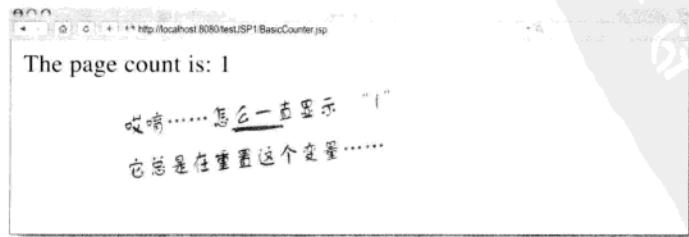
不需要导入任何东西，所以去掉了page指令。

表达式 → 让count变量增1，并打印值。

第一次点击页面时得到的结果：



第2次、第3次…以后每一次点击页面时会得到：



# JSP代码到底发生了什么？

你写的是JSP，但它会变成一个servlet。要想知道到底发生了什么，唯一的方法是查看容器对你的JSP代码做了什么。换句话说，容器怎么把JSP转换为一个servlet？

一旦了解了不同JSP元素分别放在servlet类文件中的位置，就很容易知道如何构建JSP。

这一页上的servlet代码并不是容器生成的实际代码，我们对它作了简化，只保留了基本的部分。容器生成的servlet文件会……更丑陋一些。实际生成的servlet源代码读起来会稍微困难一点，不过后面几页会分析真正的生成代码。至于现在，我们只关心JSP代码最后会放在servlet类中的哪个位置。

这个JSP:

会变成这个servlet:

```
public class basicCounter_jsp extends SomeSpecialHttpServlet {

    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.io.IOException,
                           ServletException {

        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body> ");
        int count=0;
        out.write("The page count is now: ");
        out.print( ++count );
        out.write("</body></html> ");

    }
}
```

所有scriptlet和表达式代码都放在服务方法中。  
这说明scriptlet中声明的变量总是局部变量！

注意：如果你想看看Tomcat生成的servlet代码，可以查看 yourTomcatHomeDir/work/Catalina/yourServerName/yourWebAppName/org/apache/jsp（根据你的系统和具体Web应用，如下划线的名字可能要有所调整）。

不会是这样吧：肯定有另外一种JSP元素来声明实例变量而不是局部变量……

## 需要另一个JSP元素……

在scriptlet中声明count变量意味着，每次运行服务方法时这个变量都会重新初始化。这说明对于每个请求它都会重置为0。我们需要以某种方式让count成为一个实例变量。

到目前为止，我们已经看到了指令、scriptlet和表达式。指令用于向容器提供特殊的指示，scriptlet就是普通的Java代码，会原样放在所生成servlet的服务方法中，另外表达式的结果总会成为print()方法的参数。

不过除此以外，还有一个称为声明的JSP元素。

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

↑  
百分号（%）后面要  
放一个感叹号（!）。  
↑  
这不是表达式，这里  
需要有分号！

JSP声明用于声明所生成servlet类的成员。这说明变量和方法都可以声明！换句话说，<%!和%>标记之间的所有内容都会增加到类中，而且置于服务方法之外。这意味着完全可以声明静态变量和方法。

# JSP 声明

JSP声明总是在类中定义，而且置于服务方法（或任何其他方法）之外。非常简单，声明就是要声明静态及实例变量和方法（不错，从理论上讲，你还可以定义其他成员，包括内部类，但是99.9999%的情况下都是用来声明方法和变量）。下面的代码可以解决Pauline的问题：现在每次客户请求这个页面时，计数器会不断增1。

## 变量声明

这个JSP:

会变成这个servlet:

```

public class basicCounter.jsp extends SomeSpecialHttpServlet {

    int count=0;

    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.io.IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");
        out.write("The page count is now:");
        out.print( ++count );
        out.write("</body></html>"); ← 这一次，会让一个实例变量
                                            (而不是局部变量) 增1。
    }
}

<html><body>
<%! int count=0; %>
The page count is now:
<%= ++count %>
</body></html>

```

## 方法声明

这个JSP:

会变成这个servlet:

```

public class basicCounter.jsp extends SomeSpecialHttpServlet {

    int doubleCount() { ← 这个方法与你在JSP中键入的
        count = count*2; 方法完全相同。
        return count;
    }

    int count=1; ← 由于是Java，所以超前引用没有问题（先在方法中使用变
                    章，然后才声明这个变量）。

    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.io.IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");
        out.write("The page count is now:");
        out.print( doubleCount() );
        out.write("</body></html>"); ←
    }
}

```

## 来看一个实际生成的servlet

容器会根据JSP创建servlet，我们已经看到了这样一个超简化的servlet。在开发过程，没有必要查看容器生成的代码，但是可以利用这些代码帮助学习。一旦了解了容器如何处理JSP的各个不同元素，就不用再查看容器生成的.java源文件了。而且有些开发商不允许你查看生成的Java源代码，只保留了编译后的.class文件。

如果发现部分API你不认识，也不用害怕。大多数类和接口类型都是开发商特定的实现，无需你操心。

### 容器如何处理JSP

- ▶ 查看指令，得到转换时可能需要的信息。
- ▶ 创建一个HttpServlet子类。

对于Tomcat 5，所生成的servlet会扩展：

`org.apache.jasper.runtime.HttpJspBase`

- ▶ 如果一个page指令有import属性，它会在类文件的最上面（package语句下面）写import语句。

对于Tomcat 5，package语句（你不用操心）是：

`package org.apache.jsp;`

- ▶ 如果有声明，容器将这些声明写到类文件中，通常放在类声明下面，并在服务方法前面。Tomcat 5声明了自己一个静态变量和一个实例方法。
- ▶ 建立服务方法。服务方法的具体方法名是`_jspService()`。`_jspService()`由servlet超类被覆盖的`service()`方法调用，要接收`HttpServletRequest`和`HttpServletResponse`参数在建立这个方法时，容器会声明并初始化所有隐式对象（在下一页你会看到更多隐式对象）。
- ▶ 将普通的HTML（称为模板文本）、scriptlet和表达式放到服务方法中，完成格式化，并写至`PrintWriter`响应输出。



Relax 关于生成的类，考试中基本上不会涉及。

我们显示了生成的代码，以便你了解JSP如何转换为servlet代码。但是你不必知道一个特定的开发商是怎么做的，也不用了解生成的代码到底是什么样子，这些细节都不用知道。你只要知道每一类元素的行为（scriptlet、指令、声明等）就足够了，也就是这个元素在生成的servlet中如何工作。例如，你要知道scriptlet可以使用隐式对象，而且要了解隐式对象的Servlet API类型。但是无需知道用什么代码才能使这些对象可用。

关于生成的代码，还需要知道3个JSP生命周期方法：`jspInit()`、`jspDestroy`和`jspService()`（这一章后面将详细介绍）。

## Tomcat 5生成的类

```

package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

如果 page 指令有 import (属性), 就会显示在这里 (这个 JSP 没有 import)。
}

public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    int count=0;
    private static java.util.Vector _jspx_dependants;

    public java.util.List getDependants() {
        return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();      现在初始化隐式对象。
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;
            out.write("\r<html>\r<body>\r");
            out.write("\rThe page count is now: \r");
            out.print( ++count );
            out.write("\r</body>\r</html>\r");
        } catch (Throwable t) {
            if (!(t instanceof SkipPageException)){
                out = _jspx_out;
                if (out != null && out.getBufferSize() != 0)
                    out.clearBuffer();
                if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
            }
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
        }
    }
}

```

容器把你的声明 (<%! %> 标记中的内容) 和它自己的声明放在类声明下面。

容器声明了一大堆自己的局部变量，包括表示“隐式对象”的变量。你的代码可能需要这些隐式对象，如“out”和“request”。

运行和输出JSP中的HTML、scriptlet和表达式代码。

当然，可能会出问题……

## out变量不是唯一的隐式对象……

容器把JSP转换为servlet时，服务方法最前面有一堆隐式对象声明和赋值。

利用隐式对象，编写JSP时应该知道你的代码将成为servlet的一部分。换句话说，可以充分利用“servlet性”，尽管并不是你自己直接编写servlet类。

再回头想想第4章、第5章和第6章，你使用过哪些重要的对象？你的servlet怎么得到servlet初始化参数？你的servlet怎么得到上下文初始化参数？你的servlet怎么得到会话？你的servlet怎么得到客户在表单中提交的参数？

出于这样一些原因（原因还不只是这些），你的JSP可能需要使用servlet能用的一些信息。所有隐式对象都会映射到Servlet/JSP API中的某个东西。例如，request隐式对象就是容器传递给服务方法的HttpServletRequest对象的一个引用。

API	隐式对象	
JspWriter	out	其中哪些表示请求、会话和应用的属性作用域？（没错，确实再明显不过了）。不过页面级作用域：“页面级作用域”，在又有了第4个作用域：“pageContext”。而且页面作用域属性都存储在pageContext中。
HttpServletRequest	request	
HttpServletResponse	response	
HttpSession	session	
ServletContext	application	只有指定的“错误页面”才能用这个隐式对象（本书后面还会介绍）。
ServletConfig	config	
Throwable	exception	
PageContext	pageContext	← PageContext封装了其他隐式对象，所以如果向某些辅助对象提供了一个PageContext引用，这些辅助对象就可以使用这个PageContext引用得到其他隐式对象的引用以及所有作用域的属性。
Object	page	

问：

JspWriter与从HttpServletResponse得到的PrintWriter之间有什么不同？

答：

JspWriter不在PrintWriter的类层次体系中，所以不能用来取代PrintWriter。不过，它的大多数打印方法都与PrintWriter相同，只不过增加了一些缓冲功能。

# 作为容器



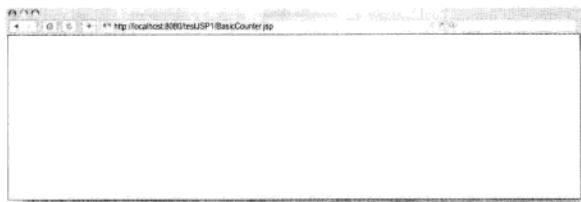
练习



以下代码清单分别取自一个JSP。你的任务是明确容器把JSP转换为一个servlet时会发生什么情况。容器能把你的JSP转换成合法、可编译的servlet代码吗？如果不能，请说明为什么不能。如果能，那么当客户访问JSP时会发生什么情况？

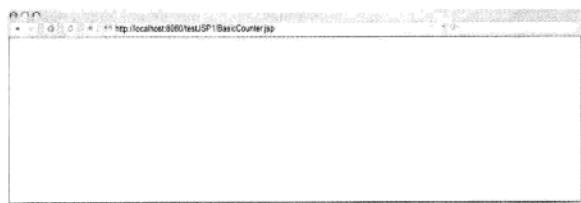
①

```
<html><body>
Test scriptlets...
<% int y=5+x; %>
<% int x=2; %>
</body></html>
```



②

```
<%@ page import="java.util.*" %>
<html><body>
Test scriptlets...
<% ArrayList list = new ArrayList();
list.add(new String("foo"));
%>
<%= list.get(0) %>
</body></html>
```



③

```
<html><body>
Test scriptlets...
<%! int x = 42; %>
<% int x = 22; %>
<%= x %>
</body></html>
```





## 模拟测验代码贴

仔细研究以下场景（以及这一页上的所有内容），再把代码帖放在JSP上，建立一个能生成正确结果的合法文件。一个代码帖不能使用多次，而且不一定每个代码帖都会用到。这个练习假设有一个servlet（你不需要看到这个servlet），它会取得初始请求，将一个属性绑定到请求作用域，并转发到你创建的JSP。

（注意：我们称之为“模拟测验代码贴”而不是“代码贴”，这是因为考试中会有大量诸如此类的拖放题目）

### 设计目标

创建一个能生成以下结果的JSP：

这三个名字来自一个名为“names”的ArrayList请求属性。你要从请求对象得到这个属性。假设有一个servlet得到这个请求，并在请求作用域设置一个属性。

### HTML表单

```
<html><body>
<form method="POST"
      action="HobbyPage.do">
  Choose a hobby:<p>

  <select name="hobby" size="1">
    <option>horse skiing
    <option>extreme knitting
    <option>alpine scuba
    <option>speed dating
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form>
</body></html>
```

这会转向一个servlet，它先  
设置请求属性，再把请求转  
发到你写的JSP。

文本“extreme knitting”来自一个表单请求参数。你要从JSP得到这个参数。会有一个servlet先得到请求（再把请求转发给你的JSP），不过这对于JSP中如何得到参数没有影响。

### 重要提示和线索

- ▶ 请求属性的类型是 `java.util.ArrayList`。
- ▶ 对应`HttpServletRequest`对象的隐式变量名为`request`，可以在`scriptlet`或表达式中使用这个隐式变量，但是不能在指令或声明中使用。在servlet中能用请求对象做什么，在JSP中也同样可以。
- ▶ JSP的servlet方法可以处理请求参数，这是因为，要记住你的代码会放在一个servlet的服务方法中。不用操心请求中使用的是哪个HTTP方法（GET或POST）。

我们已经帮你放了几行代码。你在这个JSP里放的代码必须与这里已有的代码一致。完成了这个JSP后，它应当能编译，而且能生成上一页所示的结果（必须假设已经有一个实际工作的servlet会首先得到请求，设置请求属性“names”，并把请求转发到这个JSP）。

# 停！

这绝不是一个可有可无的练习。  
这同样是JSP语法课的一部分！

The friends who share your hobby of

are: <br>

```
<% Iterator it = al.iterator();
```

这些代码不一定都会用到！

<br>

```
<% } %>
```

```
<html><body>
```

it.hasNext()

</body></html>

request.

while

%>

{ %>

%>

%>

%>

%>

%>

%>

%>

%>

import

(it.hasNext())

out.

getattribute("hobby")

=

<%=

session.

<%=

request.

<%=

al

getAttribute("names")

<%=

<%!

getParameter("hobby")

ArrayList

request.

page

<%

;

import java.util.\*;

<%>

session.

<%>

import

java.util.\*

<%>

getattribute("hobby")

=

session

<%>

al

<%

PDG



## 作为容器

### 答案

#2很直接，这是合法的。#1存在一个基本的Java语言问题（在声明局部变量之前先行使用），#3也展示了一个基本的Java语言问题，如果使用了同名的实例变量和局部变量会发生什么情况。所以……如果把JSP代码转换成servlet Java代码，应该不难看出结果。一旦JSP代码转换到一个servlet中，它就只是Java而已。

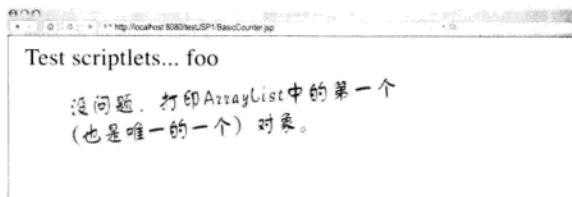
① <html><body>  
Test scriptlets...  
<% int y=5+x; %>  
<% int x=2; %>  
</body></html>

这样不能编译！

这就好像写以下方法：

```
void foo() {  
    int y = 5 + x; //你想在定义局部变量 'x' 之前先行使用  
    int x = 2; //这个变量。Java语言不允许这样做，容器不会重新摆放scriptlet代码的顺序。  
}
```

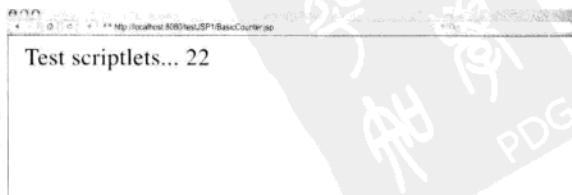
② <%@ page import="java.util.\*" %>  
<html><body>  
Test scriptlets...  
<% ArrayList list = new ArrayList();  
 list.add(new String("foo"));  
%>  
<%= list.get(0) %>  
</body></html>



③ <html><body>  
Test scriptlets...  
<%! int x = 42; %>  
<% int x = 22; %>  
<%= x %>

这个scriptlet声明了一个局部变量“x”（它隐藏了实例变量x）。所以如果你想打印实例变量x（42），而不是局部变量x（22），就要把表达式改为：

<%= this.x %>





## 代码贴答案

如果你的答案看起来与以下有些差别，但你仍坚信它应该能正常工作——可以试试看！你必须建立一个servlet得到表单请求，设置一个属性，并把请求转发（分派）到JSP。

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

因为有ArrayList和Iterator，所以需要带import属性的page指令。

The friends who share your hobby of

```
<%= request.getParameter("hobby") %>
```

are: <br>

```
<% ArrayList al = (ArrayList) request.getAttribute("names"); ; %>
```

<% Iterator it = al.iterator(); %> 从这里开始一个scriptlet……

```
while (it.hasNext()) { %> 到这里结束。
```

```
<%= it.next() %> 使用一个表达式。
```

<br> 结束while循环块！

<% } %> (如果你忘了，将无法编译)。

```
</body></html>
```



## 注释…

对，可以在JSP中放上注释。如果你是一个Java程序员，HTML经验很少，可能会毫不迟疑地键入类似下面的“注释”：

```
// this is a comment
```

但是如果这样做，除非它在一个scriptlet或声明标记中，否则最后就会作为响应的一部分显示给客户。换句话说，对于容器，这两个斜线也是模板文本，就像“Hello”或“Email is:”一样。

在JSP中可以放两种不同类型的注释：

### ► <!-- HTML 注释 -->

容器把它直接传递给客户，在客户端，浏览器会把它解释为注释。

### ► <%-- JSP 注释 --%>

这些注释是为页面开发人员提供的，就像是Java源文件中的Java注释一样，转换页面时会把这些注释去掉。如果你键入了一个JSP，想放些注释来说明你要做什么，就要像Java源文件中加注释一样地使用JSP注释。如果你希望注释作为HTML响应的一部分交给客户（不过浏览器会把它们隐藏起来，不让客户看到），就要使用HTML注释。

## 合法的和不合法的表达式

合法吗？

<%= 27 %>

所有基本类型直接量都是合法的。

<%= ((Math.random() + 5)\*2); %>

不对！这里不能有分号。

<%= "27" %>

String直接量是合法的。

<%= Math.random() %>

对，这个方法返回一个double。

<%= String s = "foo" %>

不对！这里不能有变量声明。

<%= new String[3] %>

对，因为new String数组是一个对象，而任何对象都可以发送给println()语句。

<% = 42\*20 %>

不对！算术表达式是对的，但是在%和=之间有一个空格。不能是<% =，必须是<%=。

<%= 5 > 3 %>

对，这会计算为一个布尔值，所以会打印‘true’。

<%= false %>

我们已经说过，基本类型的直接量都是合法的。

<%= new Counter() %>

没问题，这就类似于String[]……它会打印对象(Counter)的toString()方法的结果。

# 所生成servlet的API

容器根据你的JSP生成一个类，这个类实现了HttpJspPage接口。关于由容器生成的servlet的API，你只要了解这个接口就行了。例如，你不必知道Tomcat生成的servlet要扩展：

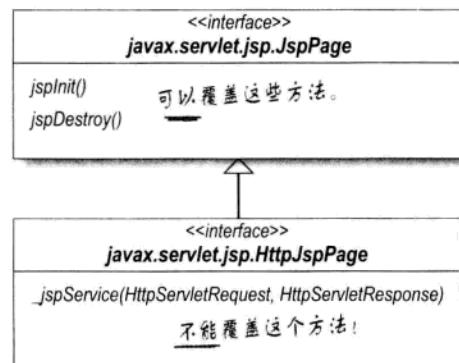
`org.apache.jasper.runtime.HttpJspBase`

你要知道的只是3个关键方法：

## ► `jspInit()`

这个方法由`init()`方法调用。

可以覆盖这个方法（你知道怎么覆盖吗）？



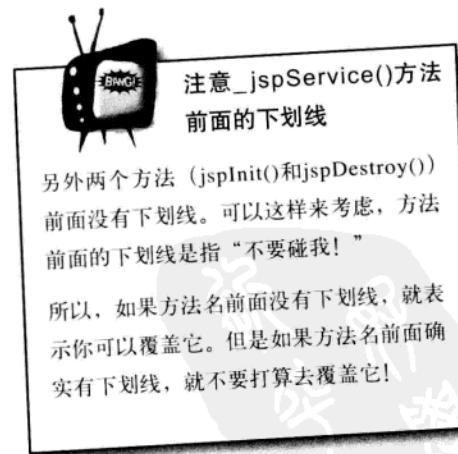
## ► `jspDestroy()`

这个方法由servlet的`destroy()`方法调用。这个方法也可以覆盖。

## ► `_jspService()`

这个方法由servlet的`service()`方法调用，这说明，对于每个请求，它会在一个单独的线程中运行。容器将Request和Response对象传递给这个方法。

不能覆盖`_jspService()`！对于这个方法，你自己什么也做不了（不过你在JSP中编写的代码会放在其中），要由容器开发商来取得你的JSP代码，并生成使用这些JSP代码的`_jspService()`方法。



# JSP的生命周期

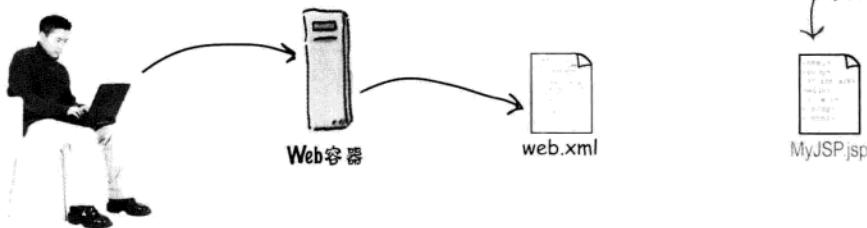
你编写了.jsp文件。

这个JSP要变成servlet，容器会为这个servlet编写.java文件。

- ① Kim 写了一个.jsp文件，并部署为Web应用的一部分。

容器“读取”这个应用的web.xml (DD)，但是对.jsp文件不做任何处理（直到得到第一个请求）。

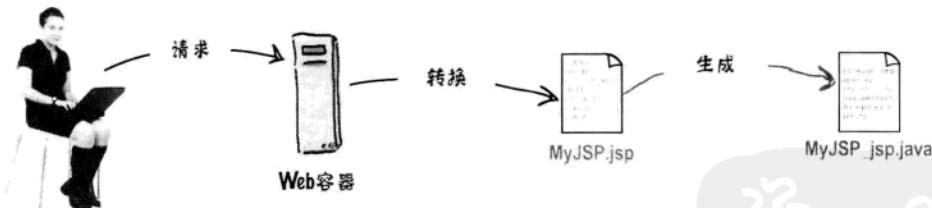
它一直待在服务器上……  
等待着客户发出请求。



- ② 客户点击一个链接，请求这个.jsp。

容器尝试将.jsp转换（翻译）为一个servlet类的.java源代码。

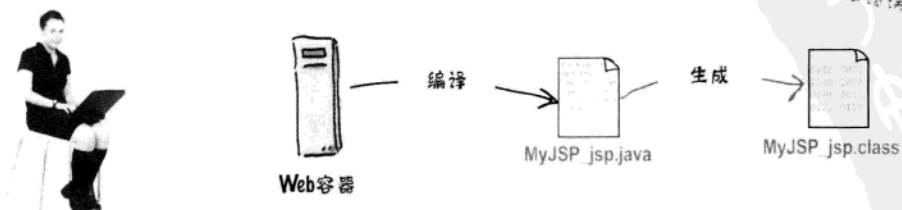
会在这个阶段发现  
JSP语法错误。



- ③

容器尝试把这个servlet .java源文件编译为一个.class文件。

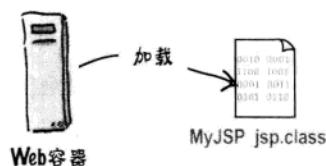
会在这里捕获到Java语  
言/语法错误。



# JSP生命周期（续）……

④

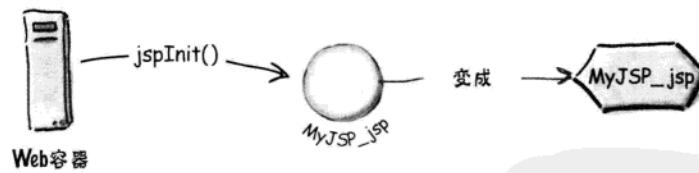
容器加载新生成的servlet类。



⑤

容器实例化servlet，并导致servlet的`jspInit()`方法运行。

对象现在成为一个真正的servlet，准备就绪，可以接受客户请求了。

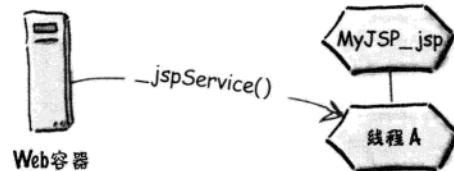


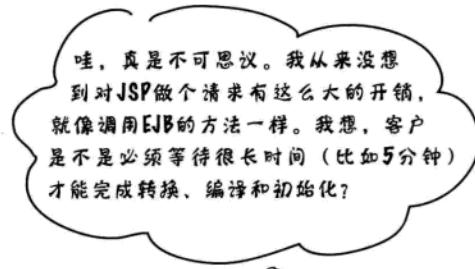
⑥

容器创建一个新线程来处理这个客户的请求，servlet的`_jspService()`方法运行。

此后发生的所有事情都只是普通的servlet请求处理。

最终servlet向客户发回一个响应  
(或者把请求转发到另一个Web应用组件)。





## 转换和编译只发生一次

如果一个Web应用包含JSP，部署这个应用时，在JSP生命周期中，整个转换和编译步骤只发生一次。JSP一旦得到转换和编译，就像其他servlet一样了。而且类似于其他的servlet，一旦这个servlet得到加载和初始化，请求时只会发生一件事，就是创建或分配一个线程来运行服务方法。所以前两页上的图只是针对第一个请求。



**问：** OK，这说明只有第一个请求JSP的客户会等得长一些。但是肯定有办法能把服务器配置为完成预转换和编译……对不对？

**答：** 尽管只是第一个客户需要等待，但大多数容器开发商确实提供了这样一种方法，可以让整个转换/编译工作提前进行，这样甚至第一个请求也能像所有其他servlet请求一样很快地得到处理。

但是要当心，这取决于具体的开发商，而且不能保证。JSP规范（JSP 11.4.2）中确实提到了一种推荐的JSP预编译协议。向JSP做出请求时可以追加一个查询串“?jsp\_compile”、容器可能（如果支持）会立即完成转换/编译，而不是等到第一个请求真正到来时才进行转换和编译。

如果JSP会转换成一个servlet，我在想能不能配置servlet初始化参数……说到这里，我还想知道能不能覆盖servlet的init()方法……



### Sharpen your pencil

考虑下面这些问题。如果需要，可以返回去看看前面几页（和前面几章），但是在完成这个练习之前先别看下一页。

不错，可以从JSP得到servlet初始化参数，问题是：

1) 如何在代码中获取这些初始化参数？（给你一个重要的提示：仔细想想在“正常”的servlet中你是如何获取的。你通常会从哪个对象来获取servlet初始化参数？这个对象在JSP代码中也能用吗？）

2) 如何配置servlet初始化参数，在哪里配置？

3) 假设你想覆盖init()方法……该怎么做呢？能不能用其他办法得到同样的结果？

# 初始化JSP

可以在JSP中完成servlet初始化工作，但是这与常规servlet中的做法稍有不同。

## 配置servlet初始化参数

要为JSP配置servlet初始化参数，这与为常规servlet配置初始化参数基本上是一样的。唯一的区别是必须在<servlet>标记中增加一个<jsp-file>元素。

```
<web-app ...>
  <servlet>
    <servlet-name>MyTestInit</servlet-name>
    <jsp-file>/TestInit.jsp</jsp-file> ← 只有这一行与常规的servlet不同。它实际上
    <init-param> 在说，“把这个<servlet>标记中的所有配
      <param-name>email</param-name> 置应用到由这个JSP页面生成的servlet...”
      <param-value>iKickedbutt@wickedlysmart.com</param-value>
    </init-param>
  </servlet> ← 为一个JSP定义servlet时，还必须为
  <servlet-mapping> JSP页面定义一个servlet映射。
    <servlet-name>MyTestInit</servlet-name>
    <url-pattern>/TestInit.jsp</url-pattern>
  </servlet-mapping>
</web-app>
```

## 覆盖`jsplInit()`

是的，真是很简单。如果你实现了`jsplInit()`方法，在JSP页面成为servlet的最开始，容器会调用这个方法。这个方法由servlet的`init()`方法调用，所以在运行`jsplInit()`方法时，已经有一个`ServletConfig`和`ServletContext`可供servlet使用。这说明，可以在`jsplInit()`方法中调用`getServletConfig()`和`getServletContext()`。

这个例子使用`jsplInit()`方法获取一个servlet初始化参数（已在DD中配置），并使用这个参数值来设置一个应用作用域属性。

```
<%! ← 使用一个声明来覆盖
  jsplInit()方法。%>
  public void jsplInit() {
    ServletConfig sConfig = getServletConfig();
    String emailAddr = sConfig.getInitParameter("email"); ← 因为这是在一个servlet中，所以可以
    // 使用繼承的getServletConfig()方法！
    ServletContext ctx = getServletContext(); ← 这与在正常servlet中
    ctx.setAttribute("mail", emailAddr); ← 的做法完全一样。
    // 得到ServletContext的一个引用。
    // 并设置一个应用作用域属性。
  }
}
```

# JSP中的属性

在上一页的例子中可以看到，这个JSP使用了一个方法声明（覆盖`jspInit()`）来设置一个应用作用域属性。但是大多数情况下，都会使用4个隐式对象之一来得到和设置对应JSP中4个属性作用域的属性。

对，是4个。要记住，除了标准的servlet请求、会话和应用（上下文）作用域外，JSP还增加了第4个作用域，即页面作用域，可以从`pageContext`对象得到。

通常你不需要（也不关心）页面作用域，除非你在开发定制标记，所以等到介绍定制标记那一章时我们再来谈页面作用域。

## servlet中

## JSP中 (使用隐式对象)

---

应用	<code>getServletContext().setAttribute("foo", barObj);</code>	<code>application.setAttribute("foo", barObj);</code>
请求	<code>request.setAttribute("foo", barObj);</code>	<code>request.setAttribute("foo", barObj);</code>
会话	<code>request.getSession().setAttribute("foo", barObj);</code>	<code>session.setAttribute("foo", barObj);</code>
页面	不适用！	<code>pageContext.setAttribute("foo", barObj);</code>

但这还不是全部！在JSP中，还有一种办法来设置和获取任意作用域的属性，可以只使用`pageContext`隐式对象。翻到下一页，看看这是怎么做到的…



没有“上下文”作用域之类的东西…尽管应用作用域中的属性绑定到`ServletContext`对象。

你可能会被这个命名约定蒙骗，认为存储在`ServletContext`中的属性是……上下文作用域属性。但是没有这样的东西。要记住，看到“上下文”时，就要想到“应用”。不过，得到应用作用域属性时，servlet和JSP的命名存在着不一致，在servlet中是：

```
getServletContext().getAttribute("foo")
```

而在JSP中则是：

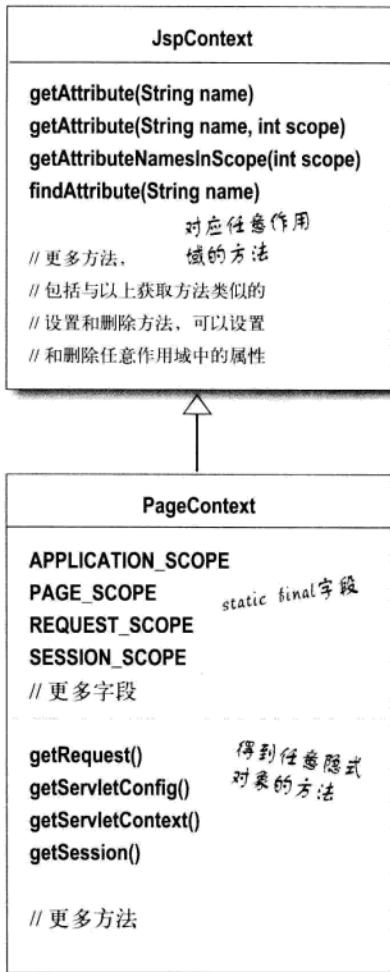
```
application.getAttribute("foo")
```



## 使用PageContext得到属性

可以使用PageContext引用来得到任意作用域的属性，包括从页面作用域得到绑定到PageContext的属性。

访问其他作用域的方法要取一个int参数，这个参数用来指示是哪一个作用域。尽管属性存取方法放在JspContext中，但作用域常量在PageContext类中。



# 使用pageContext获得和设置属性的示例

## 设置一个页面作用域属性

```
<% Float one = new Float(42.5); %>
<% pageContext.setAttribute("foo", one); %>
```

## 获得一个页面作用域属性

```
<%= pageContext.getAttribute("foo") %>
```

## 使用pageContext 设置一个会话作用域属性

```
<% Float two = new Float(22.4); %>
<% pageContext.setAttribute("foo", two, PageContext.SESSION_SCOPE); %>
```

## 使用pageContext 获得一个会话作用域属性

```
<%= pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
```

(这等价于: <%= session.getAttribute("foo") %> )

## 使用pageContext 获得一个应用作用域属性

```
Email is:
<%= pageContext.getAttribute("mail", PageContext.APPLICATION_SCOPE) %>
```

在JSP中, 以上代码等价于:

```
Email is:
<%= application.getAttribute("mail") %>
```

## 使用pageContext , 即使不知道作用域也可以查找一个属性

```
<%= pageContext.findAttribute("foo") %> 在哪里查找?
```

findAttribute()方法在哪里查找属性? 它首先在页面上下文中查找, 所以如果页面上下文作用域有一个“foo”属性, 那么在PageContext上调用findAttribute(String name), 就相当于在PageContext上调用getAttribute(String name)。但是如果页面作用域没有这样一个“foo”属性, 这个方法就会开始在其他作用域查找, 先从最严格的作用域查起, 逐步转向不那么严格的作用域, 换句话说, 先在请求作用域查找, 再查找会话作用域, 最后查找应用作用域。先找到哪一个就算“赢”, 如果在一个作用域中找到指定名字的属性, 就不会再在其他作用域中查找。

pageContext  
getAttribute(String)对  
应页面作用域

在pageContext上有两个重载的  
getAttribute()方法:  
其中单参数版本取一个String参数, 另  
一个重载方法是分别有一个String和一个  
int的两参数版本。单参数版本就像所有  
其他作用域上的相应方法一样, 用于得  
到绑定到pageContext对象的属性。但是  
两参数版本可以得到4个作用域中任意一  
个作用域的属性。

# 既然说到这里了……下面再来 谈谈3个指令

我们已经了解了可以用哪个指令把import语句放到从JSP生成的servlet类中。这就是带有一个import属性的page指令，page指令是3种指令中的一个，import属性是page指令的13个属性之一。现在我们来简单地介绍另外的指令，不过在后面的章节中还会详细介绍有关的内容，另外，还有一些内容由于极少使用，所以在这本书中可能根本不会详细谈到。

## ① page指令

```
<%@ page import="foo.*" session="false" %>
```

定义页面特定的属性，如字符编码、页面响应的内容类型，以及这个页面是否要有隐式的会话对象。page指令可以使用至多13个不同属性（如import属性），不过考试中只会考其中的4个属性。

## ② taglib指令

```
<%@ taglib tagdir="/WEB-INF/tags/cool" prefix="cool" %>
```

定义JSP可以使用的标记库。我们还没有讲到如何使用定制标记和标准动作，所以现在谈这个指令不太合适。先了解一下就行了……后面很快就会用两章专门介绍标记库。

## ③ include指令

```
<%@ include file="wickedHeader.html" %>
```

定义在转换时增加到当前页面的文本和代码。从而允许你建立可重用的块（如标准页面标题或导航栏），这些可重用的块能增加到各个页面上，而不必在每个JSP中重复写所有这些代码。

**问：** 我被搞糊涂了……这一页的标题说，“既然说到这里了……”，但是我没看出来指令与pageContext和属性有什么关系呀？

**答：** 它们之间确实没有什么关系。我们只是想把这两个不相关的主题更好地衔接起来。希望没有人会注意这个小细节，不过即便你注意到了……也权当没看见，好吗？

# page指令的属性

JSP 2.0规范中，page指令有13个属性，考试只涉及这13个属性中的4个。你不用把所有属性都记下来；只要了解你能做什么就可以了（后面的几章会介绍isELIgnored以及两个与错误相关的属性）。

## 考试时可能会考

---

### **import**

定义Java import语句，所定义的import语句会增加到生成的servlet类中。生成的servlet类中会自动（默认地）加上以下import语句：java.lang（当然了）、javax.servlet、javax.servlet.http和javax.servlet.jsp。

### **isThreadSafe**

定义生成的servlet是否需要实现SingleThreadModel，不过你知道，这是一个很糟糕的想法。默认值是... “true”，这表示“我的应用是线程安全的，所以我不需要实现SingleThreadModel，因为我知道实现SingleThreadModel从本质上讲是很不好的”。只有需要把这个属性值设置为false时才有必要指定这个属性，这说明你希望生成的servlet使用SingleThreadModel，不过绝对不要这么做。

### **contentType**

定义JSP响应的MIME内容（和可选的字符编码）。你应该知道默认值是什么。

### **isELIgnored**

定义转换这个页面时是否忽略EL表达式。我们还没有谈到EL；这是下一章的内容。对现在来说，只要知道可以在页面中选择忽略EL语法就行了，告诉容器忽略EL语法有两种办法，其中一种办法就是设置这个属性。

### **isErrorHandler**

定义当前页面是否是另一个JSP的错误页面。默认值是“false”，但是如果这个属性值为true，页面就能访问隐式的exception对象（这是令人讨厌的Throwable的一个引用）。如果这个属性值为false，这个JSP就不能使用隐式的exception对象。

### **errorCode**

定义一个资源的URL，如果有未捕获到的Throwable，就会发送到这个资源。如果这个属性指定了一个JSP，该JSP的page指令中就会有isErrorHandler=“true”属性。

## 考试时不会考

---

### **language**

定义scriptlet、表达式和声明中使用的脚本语言。现在，可取值只有一个，就是“java”，但是还是留有这个属性是为了未雨绸缪，就像那些规范开发者一样，考虑到了将来可能会使用其他的语言。

### **extends**

JSP会变成一个servlet类，这个属性则定义了这个类的超类。一般不会使用这个属性，除非你真的很清楚自己在做什么，它会覆盖容器提供的类层次体系。

### **session**

定义页面是否有一个隐式的session对象。默认值为“true”。

### **buffer**

定义隐式out对象（JspWriter的引用）如何处理缓存。

### **autoFlush**

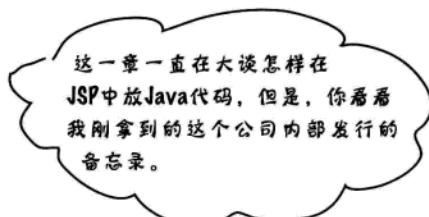
定义缓存的输出是否自动刷新输出。默认值是“true”。

### **info**

定义放到转换后页面中的串，这样就能使用所生成servlet继承的getServletInfo()方法来得到这个信息。

### **pageEncoding**

定义JSP的字符编码。默认为“ISO-8859-1”（除非contentType属性已经定义了一个字符编码，或者页面使用XML文档语法）。



CTO下发的备忘录，办公室之间传阅。

---

#### 紧急

如果发现有人在JSP代码中使用scriptlet、表达式或声明，立即停职，扣发薪水，直到能确定这个程序员真正称职或者能维护另外哪个笨家伙的代码为止。

实际上，如果能明确是哪个程序员的责任，公司还会进一步追究责任，让他的上级主管离职。

---

Rick Forester  
首席技术官

---

“记住：团队是大家的。”

“写代码时，要把将来维护这个代码的人(guy) \*想成是一个杀人狂，而且他知道你住在哪里。”

# Scriptlet是有害的吗？

这是真的吗？把这些Java代码放在JSP中会有问题吗？要知道，这不正是JSP的特点吗？这样就能在一个JSP中（实际上就是HTML页面）写Java代码，而不用在Java类中写HTML，不对吗？

有些人认为（没错，理论上讲很多人都这样认为，包括JSP和Servlet规范小组），把所有这些Java代码放在JSP中是一个不好的实践。

为什么？假设聘请你去构建一个大型的网站。你的小组包括少数后台Java程序员，还有一大堆“Web设计人员”，这些人是一些专攻图形图像的艺术家和页面创作人员，他们会用Dreamweaver和Photoshop来建立相当漂亮的Web网页。这些人不是程序员（除非还有人认为写HTML也算是“编写代码”）。



两个问题要问你——为什么你  
让我们学脚本呢？还有，如果不写脚  
本，有什么别的办法吗？如果不能在  
JSP中放scriptlet、声明和表达式，除  
了HTML外，究竟(f\*\*\*\*)还能放什  
么？



## 原先别无选择。

也就是说，已经有大量这样的JSP文件，页面中每一处都充斥着Java代码，这些Java代码嵌在scriptlet、表达式和声明标记中。到处都是脚本，对此过去没有人能做任何改变。所以，这说明你必须知道如何阅读和理解这些元素，必须知道怎么维护用这些脚本编写的页面（除非你有机会大幅重构应用的JSP）。

私下里讲，我们认为脚本还是有些意义的，要想很快地测试服务器上的某个功能，没有什么能强过简短的几行Java代码。但是，大多数情况下，对于生产环境的实际页面你可能不会使用这种办法。

考试中之所以还会考这个内容，这是因为它的替代做法还相当“年轻”，所以如今的大多数页面还是“老一套”。在一段时间内，你还是要和这种方法打交道！等到某一天，不用Java的新技术为大众所接受时，这一章的大纲要求可能就会去掉，让我们拭目以待Java完全从JSP中消失的那一天。

不过目前这一天还没有到来。

(致父母和老师：这个问题泡泡里有一个4字母的词，第一个字母是“f”，后面跟着3个星号，千万别误解。这只是一个单词而已，我们觉得它太搞笑了，要是放在这里可能会分散读者的注意力，所以采用了这种写法。这个词不是脏话，只是太好玩了。)

噢，要是JSP中有办法使用简单的标记来运行Java方法，而且不用把具体的Java代码放在页面中，那就多好。



## 一切答案尽在EL中。

或者严格地讲，所有答案几乎都是EL。对于把具体Java放在JSP中的做法，往往有两大抱怨，起码针对这两个问题来说，EL是不折不扣的答案：

- 1) 不应该要求Web页面设计人员必须懂Java。
- 2) JSP中的Java代码很难修改和维护。

EL代表“表达式语言”（Expression Language），从JSP 2.0规范开始，它已经正式成为规范的一部分。原先能用scriptlet和表达式完成的事情，都能用EL完成，而且EL往往更为简单。

当然，你现在可能会想，“但是，如果我的JSP要使用定制的方法，不使用Java的话，我怎么声明和编写这些方法呢？”

哈哈……编写具体的功能（方法代码），这可不是EL的用途。EL的用途是提供一种更简单的方法来调用Java代码，但是代码本身放在别的地方。这说明，这些代码可能在一个普通的常规Java类中，也许是一个JavaBean，一个有静态方法的类，或者是某个所谓的标记处理器（Tag Handler）。换句话说，按照现今的最佳实践，不会在JSP中编写方法代码。要把Java方法写在别的地方，再用EL来调用。

## EL一瞥

下一章通篇都将讨论EL，所以这里不会太详细地说明。之所以在这里提到EL，只是因为它是JSP中的另外一类元素（而且有自己的语法）。另外，这一章的考试要求中提到，你要知道可以在JSP中放哪些东西。

以下EL表达式：

```
Please contact: ${applicationScope.mail}
```

EL表达式的形式是：

`${something}`

换句话说，表达式总是括在大括号中，而且前面有一个美元符(\$)作前缀。

与这个Java表达式是等价的：

```
Please contact: <%= application.getAttribute("mail") %>
```

*there are no  
Dumb Questions*

**问：** 还不错，不过我看不出EL和Java表达式之间有天大的差别。当然，EL要短一些，但是你觉得为了这么一点点好处就要用一个全新的脚本语言和JSP编码方法，这样值得吗？

**答：** 你还没看到EL真正的好处呢！下一章还会更深入地介绍。到时候你就会了解二者的差别了。不过，你要记住，对于Java程序员来说，从开发角度看EL不一定是个显著的进步。实际上，在Java程序员看来，这只是意味着“拜托，我已经懂Java了，可是又要学一个新东西（它有自己的语法，自己的一套方法）……”

不过，不能片面地只想着你自己。对于非Java程序员来说，EL更好学，更容易掌握。就算是Java程序员，也会发现维护一个无脚本的页面更为容易。

不过，确实要学一些东西。不会让Web页面设计人员完全袖手旁观，但你很快就会看到，Web设计人员使用EL更为直观、也更加自然。对于现在，在这一章中，只要看到EL时能认出来就行了。现在不用操心判断表达式本身是否合法，我们现在只关心你能不能把JSP页面中的EL表达式挑出来。



## 使用<scripting-invalid>

非常简单——在DD中放一个<scripting-invalid>标记，就可以让JSP禁用脚本元素（scriptlet、Java表达式或声明）：

```
<web-app ...>
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>
      true
    </scripting-invalid>
  </jsp-property-group>
</jsp-config>
...
</web-app>
```

当心——你可能看到过其他书或文章中有禁用脚本的page指令。在2.0规范草案中，还有这样一个page指令属性：

这样不行！JSP规范里不再有`isScriptingEnabled`属性了！ → `<%@ page isScriptingEnabled=" false" %>`

但是最后的规范中已经把这个属性删掉了！

现在，要禁用脚本，唯一的办法就是通过<scripting-invalid> DD标记。

## 可以选择忽略EL

对，我们都知道，EL是个好东西，可以解决我们的大问题。但是，有时候你可能想禁用EL，为什么呢？

回头想想刚在Java 1.4中增加assert关键字的那个时候。突然之间，原来非保留、完全合法的标识符“assert”对编译器有特殊含义了。所以，如果你以前有一个名为assert的变量，就该发愁了。除非J2SE 1.4默认地禁用断言。如果知道你正在写（或重新编译）的代码没有使用assert作为标识符，就可以启用断言。

对于禁用EL也是一样的，如果你的JSP中有模板文本（普通的HTML或文本），其中刚好包括了类似于EL的东西（\${something}），你要告诉容器忽略这些看似为EL的内容，而应当把它们作为不处理的文本对待，否则你就会遇到大麻烦。不过，EL与断言有一个根本的区别：

### 默认情况下EL是启用的！

如果你希望忽略JSP中看似EL的东西，必须显式地指出，可以通过page指令，  
也可以通过一个DD元素指定。

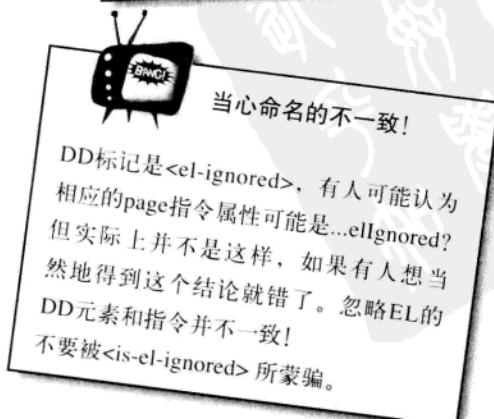
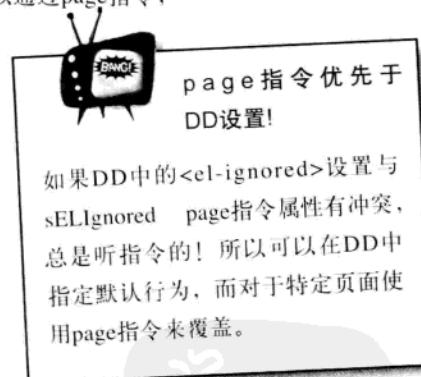
### 在DD中放置<el-ignored>元素

```
<web-app ...>
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>
      true
    </el-ignored>
  </jsp-property-group>
</jsp-config>
...
</web-app>
```

### 使用isELIgnored page指令属性

```
<%@ page isELIgnored="true" %>
```

page指令属性以“is”开头，但  
DD标记不是！



# 不过请等等……还有一个JSP元素 没见过：动作

到目前为止，你已经看到了JSP中可能出现的5种不同类型的元素：scriptlet、指令、声明、Java表达式和EL表达式。

但是我们还没有见过动作。有两种不同类型的动作：标准动作和……不标准的动作。

## 标准动作：

```
<jsp:include page="wickedFooter.jsp" />
```

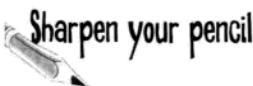
到现在来说，不要担心这些动作干什么，也不用想它们怎么做，只要在JSP中看到这样的语法能认出这是一个动作就行了。后面我们还会详细介绍。

## 其他动作：

```
<c:set var="rate" value="32" />
```

不过这有些误导，因为有些动作并不认为是标准动作，但它们仍是标准库的一部分。换句话说，后面你会了解到，有些非标准（大纲中称之为“定制”）动作是……标准的（位于标准库中），但并不认为这些动作是“标准动作”。对，可以这么说，它们是标准化的非标准定制动作。你搞清楚了吗？

一章讲到“使用标记”时，我们就有更多的知识储备，能用更多的术语更详细地讨论这个内容，所以先不用担心。对于现在，我们关心的只是看到JSP中有动作时能把它认出来！



看看动作的语法，把它与其他类型JSP元素的语法做个比较。再回答下面的问题：

) 动作元素与scriptlet有什么区别？

) 看到一个动作时，怎么才能认出来？





## 计算矩阵

想想出现以下设置（或设置组合）时会发生什么。下一页上就有答案，所以先别看后面，现在就完成这个练习。

### ① EL计算

如果一个设置会导致EL表达式计算，就在“计算”一栏中打个对勾，如果EL将像其他模板文本一样对待，就在“忽略”一栏中打勾。当然，同一行上不可能有两个对勾。

DD配置&lt;el-ignored&gt;

page指令

isELIgnored

计算

忽略

未指定	未指定		
false	未指定		
true	未指定		
false	false		
false	true		
true	false		

### ② 脚本合法性

如果一个设置会导致脚本表达式计算，就在“计算”一栏中打个对勾，如果脚本将导致一个转换错误，就在“错误”一栏中打勾。

DD配置  
<scripting-invalid>

计算

错误

未指定		
true		
false		



## JSP元素贴

把JSP元素与其标签匹配，将JSP代码段放在适当的方框里（方框上的标签表示了元素类型）。记住，在实际考试中也有类似的拖放问题，所以这个练习一定要做！

JSP元素类型



JSP代码段

把它们拖到匹配的标签上。

```
<% Float one = new Float(42.5); %>
```

```
<%! int y = 3; %>
```

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

```
<jsp:include page="foo.html" />
```

```
<%= pageContext.getAttribute("foo") %>
```

```
email: ${applicationScope.mail}
```



## JSP元素贴（剧终）

你已经知道了JSP元素的名字，但是你记得它们在生成的servlet中放在哪里吗？你当然应该记得。但是在学习下一章的新内容之前，还是先做个练习巩固一下。

（把元素放在适当的方框里，把这些JSP元素想成是servlet类文件中相应的生成代码，看看它们应该放在哪个位置。注意，元素贴本身并不表示生成的具体代码）

```
public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    ...
    ...
    ...
}

}

```

这些代码贴的顺序关系不大。

<%= request.getAttribute("foo") %>

email: \${applicationScope.mail}

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

<% Float one = new Float(42.5); %>

<% int y = 3; %>



## 计算矩阵答案

## ① EL计算

DD配置 <el-ignored>	page指令 isELIgnored	计算	忽略
未指定	未指定	✓	
false	未指定	✓	
true	未指定		✓
false	false	✓	
false	true		✓
true	false	✓	

## ② 脚本合法性

DD配置 <scripting-invalid>	计算	错误
未指定	✓	
true		✓
false	✓	



# JSP元素贴

答案

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

**指令**

```
<%! int y = 3; %>
```

**声明**

```
email: ${applicationScope.mail}
```

**EL 表达式**

```
<% Float one = new Float(42.5); %>
```

**scriptlet**

```
<%= pageContext.getAttribute("foo") %>
```

**表达式**

```
<jsp:include page="foo.html" />
```

**动作**

“表达式”本身指“脚本表达式”，而不是“EL表达式”。

当然，对于JSP元素，“表达式”一词有多个含义。如果看到“表达式”或“脚本表达式”，指的是一样的，都是使用Java语言语法的表达式：

```
<%= foo.getName() %>
```

只有当描述或标签中特别出现了“EL”时，“表达式”才指EL表达式！所以，应当认为“表达式”默认是指“脚本/Java表达式”，而不是EL表达式。



## JSP元素贴 (剧终)

答案

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

带import属性的page指令会转变成一个Java  
import语句。

```
public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase  
    implements org.apache.jasper.runtime.JspSourceDependent {
```

```
<%! int y = 3; %>
```

声明用于声明成员，所以要放在类中，而  
且在所有方法之外。

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)  
    throws java.io.IOException, ServletException {
```

...

```
<%= request.getAttribute("foo") %>
```

表达式在服务方法中变为  
print()语句。

```
<% Float one = new Float(42.5); %>
```

Scriptlet放在服务方法中。

```
email: ${applicationScope.mail}
```

EL表达式放在服务方法中。

}

}

**注意：**记住JSP代码不会真的像这样放在servlet中……它们都会转换为Java语言代码。这个练习只是想让你知道这些元素会放在生成类的那一部分，但是我们并没有显示这些元素具体会转换成什么代码。例如，声明 <%! int y = 3; %> 会变成 int y = 3;

(注意：这3个代码贴的顺序没有影响。)



## 第7章 模拟测验

1 给定以下DD元素：

```
47. <jsp-property-group>
48.   <url-pattern>*.jsp</url-pattern>
49.   <el-ignored>true</el-ignored>
50. </jsp-property-group>
```

这个元素有什么作用？（选出所有正确的答案）

- A. 有指定扩展名映射的所有文件在JSP容器看来都是良构的XML文件。
- B. 有指定扩展名映射的所有文件应该包含由JSP容器计算的表达式语言代码。
- C. 默认地，有指定扩展名映射的所有文件不应包含由JSP容器计算的表达式语言代码。
- D. 没有任何作用，容器不理解这个标记。
- E. 尽管这个标记合法，但它是冗余的，因为这是容器的默认行为。

2 哪些指令指定HTTP响应的类型是“image/svg”？（选出所有正确的答案）

- A. <%@ page type="image/svg" %>
- B. <%@ page mimeType="image/svg" %>
- C. <%@ page language="image/svg" %>
- D. <%@ page contentType="image/svg" %>
- E. <%@ page pageEncoding="image/svg" %>

3 给定以下JSP：

```
1. <%@ page import="java.util.*" %>
2. <html><body> The people who like
3. <%= request.getParameter("hobby") %>
4. are: <br>
5. <% ArrayList al = (ArrayList) request.getAttribute("names"); %>
6. <% Iterator it = al.iterator();
7.     while (it.hasNext()) { %>
8.         <%= it.next() %>
9.     <br>
10.    <% } %>
11. </body></html>
```

这个JSP中使用了哪些类型的代码？（选出所有正确的答案）

- A. EL
- B. 指令
- C. 表达式
- D. 模板文本
- E. scriptlet

4 关于`jspInit()`，以下哪些说法是正确的？（选出所有正确的答案）

- A. 能访问`ServletConfig`。
- B. 能访问`ServletContext`。
- C. 只能调用一次。
- D. 可以覆盖。

---

5 **jspInit()**方法可以使用哪些类型的对象？（选出所有正确的答案）

- A. **ServletConfig**
- B. **ServletContext**
- C. **JspServletConfig**
- D. **JspServletContext**
- E. **HttpServletRequest**
- F. **HttpServletResponse**

---

6 给定：

```
<%@ page isELIgnored="true" %>
```

它有什么作用？（选出所有正确的答案）

- A. 什么作用也没有，没有定义这个**page**指令。
- B. 如果有这个指令，JSP容器不会计算Web应用中所有JSP中的表达式语言代码。
- C. JSP容器会把包含这个指令的JSP看作是良构的XML文件。
- D. 如果JSP中包含这个指令，JSP容器不应计算这个JSP中的表达式语言代码。
- E. 只有当DD声明了一个`<el-ignored>true</el-ignored>`元素，而且其URL模式包含当前JSP时，这个**page**指令才会禁用EL计算。

---

7 有关JSP，以下哪些说法是正确的？（只有一个正确选择）

- A. 只有**jspInit()**可以覆盖。
- B. 只有**jspDestroy()**可以覆盖。
- C. 只有 **\_jspService()** 可以覆盖。
- D. **jspInit()**和**jspDestroy()**都可以覆盖。
- E. **jspInit()**、**jspDestroy()**和**\_jspService()**都可以覆盖。

**8**

哪个JSP生命周期步骤顺序不对?

- A. 把JSP转换为servlet。
- B. 编译servlet源代码。
- C. 调用`_jspService()`
- D. 实例化servlet类。
- E. 调用`jspInit()`
- F. 调用`jspDestroy()`

**9**

哪些是合法的JSP隐式变量? (选出所有正确的答案)

- A. `stream`
- B. `context`
- C. `exception`
- D. `listener`
- E. `application`

**10**

给定一个请求，它有两个参数，一个名为“first”，表示用户的名，另一个名为“last”，表示用户的姓。

哪个JSP scriptlet代码可以输出这些参数值?

- A. `<% out.println(request.getParameter("first")); out.println(request.getParameter("last")); %>`
- B. `<% out.println(application.getInitParameter("first")); out.println(application.getInitParameter("last")); %>`
- C. `<% println(request.getParameter("first")); println(request.getParameter("last")); %>`
- D. `<% println(application.getInitParameter("first")); println(application.getInitParameter("last")); %>`

**11** 给定：

```

11. Hello ${user.name}!
12. Your number is <c:out value="${user.phone}" />.
13. Your address is <jsp:getProperty name="user" property="addr" />
14. <% if (user.isValid()) {%>You are valid!<% } %>

```

哪些说法是正确的？（选出所有正确的答案）。

- A. 第11和12行有EL元素的例子（其他行没有）。
- B. 14行是scriptlet代码的例子。
- C. 这个例子中不包含模板文本。
- D. 第12和13行包含JSP标准动作的例子。
- E. 第11行展示了EL的一个非法使用。
- F. 这个例子中的4行代码在JSP页面中都是合法的。

**12**

哪个JSP表达式标记会打印名为“javax.sql.DataSource”的上下文初始化参数？

- A. <%= application.getAttribute("javax.sql.DataSource") %>
- B. <%= application.getInitParameter("javax.sql.DataSource") %>
- C. <%= request.getParameter("javax.sql.DataSource") %>
- D. <%= contextParam.get("javax.sql.DataSource") %>

**13**

关于禁用脚本元素，以下哪些说法是正确的？（选出所有正确的答案）。

- A. 不能通过DD禁用脚本。
- B. 只能在应用级禁用脚本。
- C. 可以使用page指令isScriptingEnabled属性通过程序禁用脚本。
- D. 可以使用<scripting-invalid>元素通过DD禁用脚本。

14

按以下顺序，JSP隐式对象application, out, request, response, session相应的Java类型是什么？

- A. java.lang.Throwable  
java.lang.Object  
java.util.Map  
java.util.Set  
java.util.List
- B. javax.servlet.ServletConfig  
java.lang.Throwable  
java.lang.Object  
javax.servlet.jsp.PageContext  
java.util.Map
- C. javax.servlet.ServletContext  
javax.servlet.jsp.JspWriter  
javax.servlet.ServletRequest  
javax.servlet.ServletResponse  
javax.servlet.http.HttpSession
- D. javax.servlet.ServletContext  
java.io.PrintWriter  
javax.servlet.ServletConfig  
java.lang.Exception  
javax.servlet.RequestDispatcher

15

以下哪个例子展示了JSP中用于导入一个类的语法？

- A. <% page import="java.util.Date" %>
- B. <%@ page import="java.util.Date" @%>
- C. <%@ page import="java.util.Date" %>
- D. <% import java.util.Date; %>
- E. <%@ import file="java.util.Date" %>

16

给定JSP：

```
1. <%@ page isELIgnored="true" %>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
   prefix="c" %>
3. <c:set var="awesomeBand" value="LIMOZEEN"/>
4. ${awesomeBand}
```

会得到什么输出？

- A. \${awesomeBand}
- B. LIMOZEEN
- C. 无输出。
- D. 会抛出一个异常，因为所有taglib 指令都必须放在page指令前面。



## 第7章 模拟测验答案

(JSP v2.0 ( -87页)

1 给定以下DD元素：

```

47. <jsp-property-group>
48.   <url-pattern>*.jsp</url-pattern>
49.   <el-ignored>true</el-ignored>
50. </jsp-property-group>
```

这个元素有什么作用？（选出所有正确的答案）。

- A. 有指定扩展名映射的所有文件在JSP容器看来都是良构的XML文件。
- B. 有指定扩展名映射的所有文件应该包含由JSP容器计算的表达式语言代码。
- C. 默认地，有指定扩展名映射的所有文件不应包含由JSP容器计算的表达式语言代码。
- D. 没有任何作用，容器不理解这个标记。
- E. 尽管这个标记合法，但它是冗余的，因为这是容器的默认行为。

C对，这个元素禁止JSP  
2.0 容器计算EL表达式，  
但默认情况下容器确实  
会计算EL。

2 哪些指令指定HTTP响应的类型是“image/svg”？（选出所有正确的答案）

(JSP v2.0 1.10.1节)

- A. <%@ page type="image/svg" %>
- B. <%@ page mimeType="image/svg" %>
- C. <%@ page language="image/svg" %>
- D. <%@ page contentType="image/svg" %>
- E. <%@ page pageEncoding="image/svg" %>

D展示了这个指令的正  
确语法。

3

给定以下JSP：

```

1. <%@ page import="java.util.*" %>
2. <html><body> The people who like
3. <%= request.getParameter("hobby") %>
4. are: <br>
5. <% ArrayList al = (ArrayList) request.getAttribute("names"); %>
6. <% Iterator it = al.iterator(); %>
7.     while (it.hasNext()) { %>
8.         <%= it.next() %>
9.     <br>
10. <% } %>
11. </body></html>
```

这个JSP中使用了哪些类型的代码？（选出所有正确的答案）。

- A. EL  
 B. 指令  
 C. 表达式  
 D. 模板文本  
 E. scriptlet

这个JSP中没有EL。第1行有一个指令。  
 第3行和第8行有表达式，整个JSP里都有模板文本（如第2行），当然也有脚本元素。

4

关于`jspInit()`，以下哪些说法是正确的？（选出所有正确的答案）。 (JSP v2.0 第2.1节)

- A. 能访问`ServletConfig`。  
 B. 能访问`ServletContext`。  
 C. 只能调用一次。  
 D. 可以覆盖。



5

**jspInit()**方法可以使用哪些类型的对象？（选出所有正确的答案）。

(JSP v2.0 11.2.1节)

- A. **ServletConfig**
- B. **ServletContext**
- C. **JspServletConfig**
- D. **JspServletContext**
- E. **HttpServletRequest**
- F. **HttpServletResponse**

JSP会变成普通的servlet，因此它们能访问普通的**ServletConfig**和**ServletContext**对象……不过在它的一生中现在还为时过早，还能访问请求和响应。

6

给定：

(JSP v2.0 1-49页)

&lt;%@ page isELIgnored="true" %&gt;

它有什么作用？（选出所有正确的答案）。

- A. 什么作用也没有，没有定义这个**page**指令。
- B. 如果有这个指令，JSP容器不会计算Web应用中所有JSP中的表达式语言代码。  
B不对，因为这个指令只会影响包含这个指令的JSP。
- C. JSP容器会把包含这个指令的JSP看作是良构的XML文件。
- D. 如果JSP中包含这个指令，JSP容器不应计算这个JSP中的表达式语言代码。
- E. 只有当DD声明了一个<**el-ignored**>true</**el-ignored**>元素，而且其URL模式包含当前JSP时，这个page指令才会禁用EL计算。

7

有关JSP，以下哪些说法是正确的？（只有一个正确选择）。

(JSP v2.0 11.1节)

- A. 只有**jspInit()**可以覆盖。
- B. 只有**jspDestroy()**可以覆盖。
- C. 只有 **jspService()** 可以覆盖。
- D. **jspInit()**和**jspDestroy()**都可以覆盖。
- E. **jspInit()**、**jspDestroy()**和**jspService()**都可以覆盖。

要记住，下划线是一个线索，它提示这个方法是不能覆盖的。

8

哪个JSP生命周期步骤顺序不对?

(JSP v2.0 (1.)节)

- A. 把JSP转换为servlet。
- B. 编译servlet源代码。
- C. 调用`_jspService()` `_jspService`方法不能在`jspInit`之前调用。
- D. 实例化servlet类。
- E. 调用`jspInit()`
- F. 调用`jspDestroy()`

9

哪些是合法的JSP隐式变量? (选出所有正确的答案)。

(JSP v2.0 (1.8.3节))

- A. `stream` A, B和D不是容器为JSP创建的隐式对象。
- B. `context`
- C. `exception`
- D. `listener`
- E. `application`

10

给定一个请求, 它有两个参数, 一个名为“first”, 表示用户的名; 另一个名为“last”, 表示用户的姓。

(JSP v2.0 (1-41页))

哪个JSP scriptlet代码可以输出这些参数值?

- A. `<% out.println(request.getParameter("first"));  
out.println(request.getParameter("last")); %>`
- B. `<% out.println(application.getInitParameter("first"));  
out.println(application.getInitParameter("last")); %>`
- C. `<% println(request.getParameter("first"));  
println(request.getParameter("last")); %>`
- D. `<% println(application.getInitParameter("first"));  
println(application.getInitParameter("last")); %>`

A 使用了“out” 隐式对象及其`println()`方法。C和D中少了“out” 隐式对象。

11

给定：

(JSP v2.0 1~10页)

```

11. Hello ${user.name}!
12. Your number is <c:out value="${user.phone}" />.
13. Your address is <jsp:getProperty name="user" property="addr" />
14. <% if (user.isValid()) { %>You are valid!<% } %>

```

哪些说法是正确的？（选出所有正确的答案）。

- A. 第11和12行有EL元素的例子（其他行没有）。
- B. 14行是scriptlet代码的例子。
- C. 这个例子中不包含模板文本。
- D. 第12和13行包含JSP标准动作的例子。
- E. 第11行展示了EL的一个非法使用。
- F. 这个例子中的4行代码在JSP页面中都是合法的。

C不对，因为这4行都包含了模板文本。

D不对，因为第12行没有包含JSP标准动作。

E是不对的，因为第11行中的EL是合法的。

12

哪个JSP表达式标记会打印名为“javax.sql.DataSource”的上下文初始化参数？

(JSP v2.0 1~41页)

- A. <%= application.getAttribute("javax.sql.DataSource") %>
- B. <%= application.getInitParameter("javax.sql.DataSource") %>
- C. <%= request.getParameter("javax.sql.DataSource") %>
- D. <%= contextParam.get("javax.sql.DataSource") %>

B展示了application隐式对象的正确用法。

13

关于禁用脚本元素，以下哪些说法是正确的？（选出所有正确的答案）。

(JSP v2.0 3.3.3节)

- A. 不能通过DD禁用脚本。
- B. 只能在应用级禁用脚本。
- C. 可以使用page指令`isScriptingEnabled`属性通过程序禁用脚本。
- D. 可以使用`<scripting-invalid>`元素通过DD禁用脚本。

只能通过DD禁用脚本元素  
`<jsp-property-group>`元素  
 允许禁用所选JSP中的脚本  
 只需指定要禁用的脚本的  
 URL模式。

14

按以下顺序, JSP隐式对象application, out, request, response, session相应的Java类型是什么? (JSP v2.0 | ~41页)

- A. java.lang.Throwable

```
java.lang.Object
java.util.Map
java.util.Set
java.util.List

 B. javax.servlet.ServletConfig
java.lang.Throwable
java.lang.Object
javax.servlet.jsp.PageContext
java.util.Map
```

- C. javax.servlet.ServletContext
javax.servlet.jsp.JspWriter
javax.servlet.ServletRequest
javax.servlet.ServletResponse
javax.servlet.http.HttpSession

C正确地显示了各个隐式对象的Java类型。

- D. javax.servlet.ServletContext
java.io.PrintWriter
javax.servlet.ServletConfig
java.lang.Exception
javax.servlet.RequestDispatcher

15

以下哪个例子展示了JSP中用于导入一个类的语法?

(JSP v2.0 | ~44页)

- A. <% page import="java.util.Date" %>
 B. <%@ page import="java.util.Date" @%>
 C. <%@ page import="java.util.Date" %>
 D. <% import java.util.Date; %>
 E. <%@ import file="java.util.Date" %>

A和D是非法的, 因为只有Java语句能包含在<% ... %>标记中。

C是唯一正确的例子, 它显示了正确的语法。

E是非法的, 因为没有import指令

16

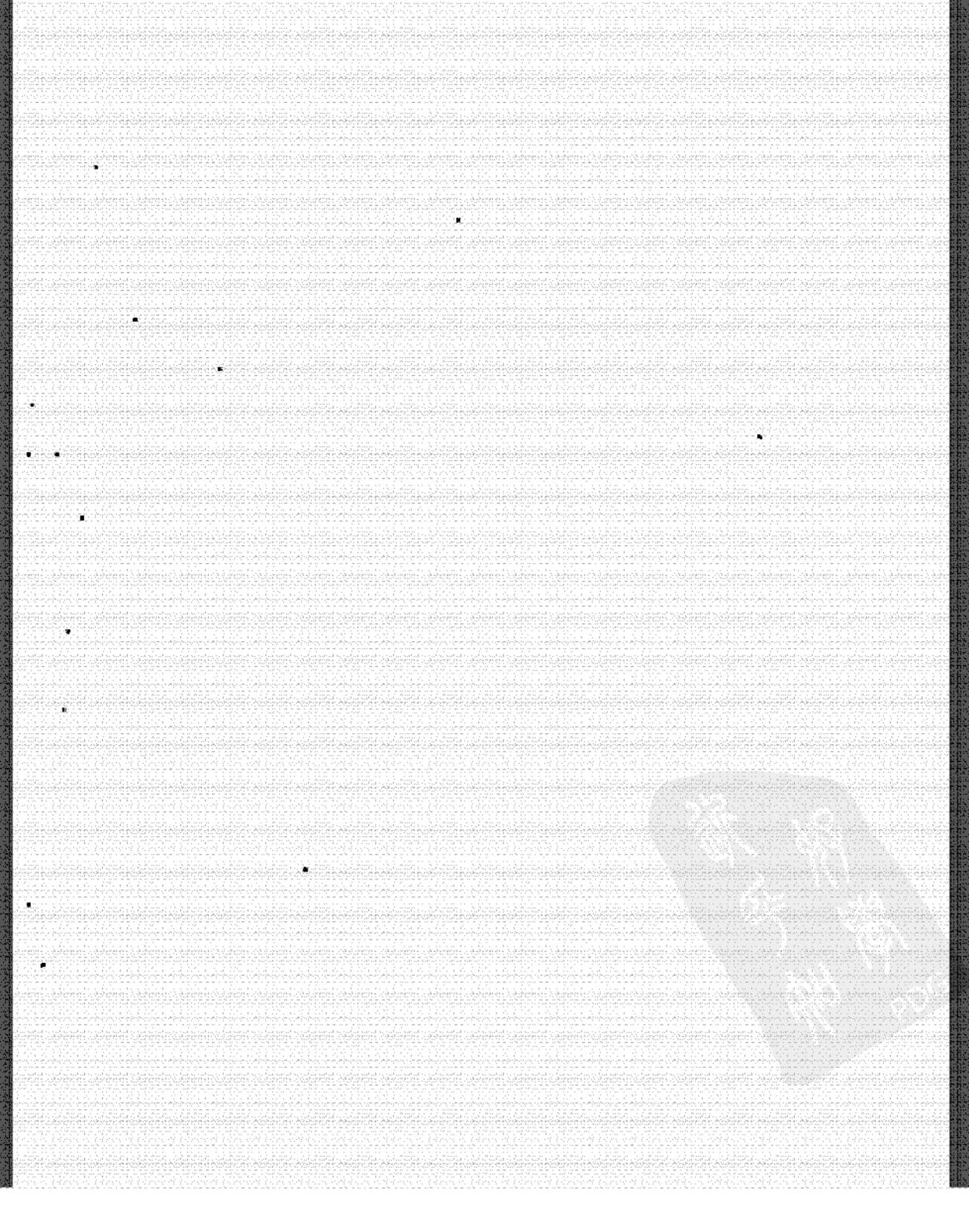
给定JSP:

(JSP v2.0 | 10.1节)

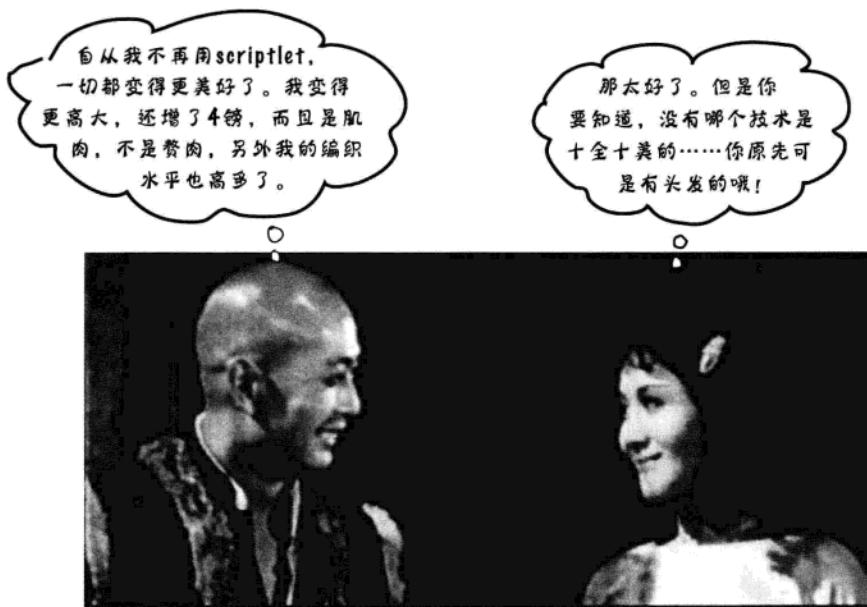
```
1. <%@ page isELIgnored="true" %>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <c:set var="awesomeBand" value="LIMOZEEN"/>
4. ${awesomeBand}
```

会得到什么输出?

- A. \${awesomeBand}      A. 由于忽略EL表达式, 会原样传递。
 B. LIMOZEEN
 C. 无输出。
 D. 会抛出一个异常, 因为所有taglib 指令都必须放在page指令前面。



# 没有脚本的页面



**扔掉脚本。**Web页面设计人员真的必须懂Java吗？这公平吗？服务器端Java程序员还得是图形设计人员吗？如果是你，你真的希望JSP里有大堆的Java代码吗？你会不会说这是“维护噩梦”？编写无脚本的页面不只是可能而已，有了新的JSP 2.0规范，编写无脚本的页面变得更容易，更灵活，这多亏了新的表达式语言(Expression Language, EL)。因为有JavaScript和XPATH的基础，Web设计人员能轻松自如地使用EL，一旦你习惯也会喜欢它的。不过它也存在一些陷阱……尽管EL看上去很像Java，但它并不是Java。有时即使你使用了Java中同样的语法，EL却可能有完全不同的表现，所以一定要注意！

# OBJECTIVES

## 使用表达式语言（EL）和标准动作构建 JSP页面

### 内容说明：

这部分的要求将在本章全面介绍。这一节很长，要多花些时间；有很多麻烦的小节需要学习。

- 7.1 使用EL中的顶级变量编写一个代码片段。这包括以下隐式变量：pageScope, requestScope, sessionScope和applicationScope; param和paramValues; header和headerValues; cookies; 以及initParam。
- 7.2 使用以下EL操作符编写一个代码片段：性质访问操作符(.) 和集合访问操作符([ ])。
- 7.3 使用以下EL操作符编写一个代码片段：算术操作符、关系操作符和逻辑操作符。
- 7.4 对于EL函数：使用EL函数编写一个代码片段；明确或创建用于声明EL函数的TLD文件结构；明确或创建一个代码示例来定义EL函数。
- 8.1 给定一个设计目标，使用以下标准动作创建一个代码片段：jsp:useBean（有以下属性：‘id’、‘scope’、‘type’ 和 ‘class’），jsp:getProperty和jsp:setProperty（包括所有属性组合）。
- 8.2 给定一个设计目标，使用以下标准动作创建一个代码片段：jsp:include, jsp:forward和jsp:param。
- 6.7 给定一个特定的设计目标，要求将一个JSP片段包含在另一个页面中，编写JSP代码使用最合适的包含机制（include指令或<jsp:include>标准动作）。

在这一章中，我们会介绍两种包含机制：考试大纲8.2的<jsp:include>标准动作和纲6.7的include页面指令（第6部分的多数要求都已经在有关JSP的前一章中涵盖）。

# 我们的MVC应用取决于属性

还记得在前面的MVC啤酒应用中，Servlet控制器先与模型（有业务逻辑的Java类）会话，再在请求作用域设置一个属性，然后才转发到JSP视图。

JSP必须从请求作用域得到这个属性，并用它生成一个响应，发回给客户。下面来快速浏览属性怎么从控制器到达视图（只考虑servlet与模型的会话），这里有所简化：

## Servlet (控制器) 代码

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

    String name = request.getParameter("userName");
    request.setAttribute("name", name); ← 使用表单中的请求参数来
                                     设置一个请求作用域属性。
                                     JSP将使用这个属性。

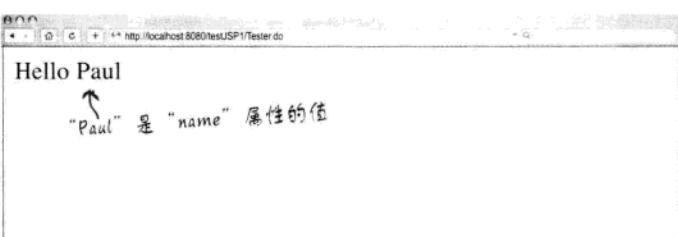
    RequestDispatcher view = request.getRequestDispatcher("/result.jsp");
    view.forward(request, response);
}
```

↑ 把请求转发到视图。

## JSP (视图) 代码

```
<html><body>
Hello
<%= request.getAttribute("name") %>
</body></html>
```

使用一个脚本表达式得到属性，并把它打印到响应。  
(记住：脚本表达式总是`out.write()`方法的参数。)



## 但是，如果属性不是一个String，而是一个Person实例，会怎么样呢？

而且不仅仅是一个普通的Person，这个Person还有一个“name”性质。这里“性质”（property）一词不按企业JavaBean（注1）的方式来理解（而只按一般的JavaBean来考虑）。Person类有一对getName()和setName()方法，根据JavaBean规范，这就说明Person有一个名为“name”的性质。不要忘了，“name”性质中第一个字母要改为小写，即应当为“n”。换句话说，要去掉方法名中的前缀“get”和“set”，再把余下的第一个字母变成小写，这样才是性质名。所以getName/setName方法对的相应性质是name。

### Servlet代码

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

    foo.Person p = new foo.Person();
    p.setName("Evan");
    request.setAttribute("person", p);

    RequestDispatcher view = request.getRequestDispatcher("result.jsp");
    view.forward(request, response);
}
```

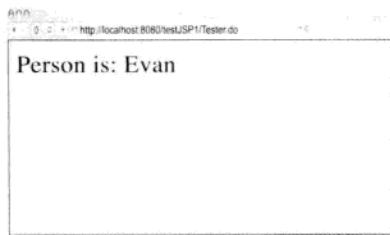
### JSP 代码

```
<html><body>

    Person is: <%= request.getAttribute("person") %>

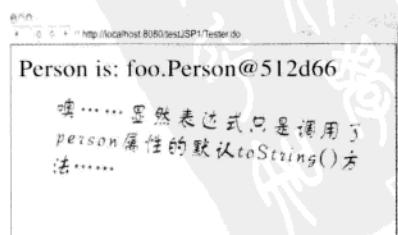
</body></html>
```

我们希望得到的结果：



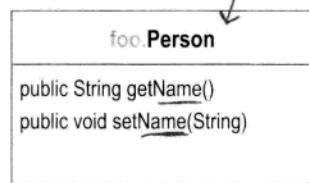
getAttribute()返回什么：

我们真正得到的结果：



注1：后面几页会谈到JavaBean，不过，对于现在，只要知道它是包含获取方法和设置方法的一个普通Java类，而且其获取方法和设置方法遵循一个命名约定，知道这些就足够了。

一个简单的JavaBean



可以从这对获取方法/设置方法了解到，Person有一个名为“name”的性质（注意“name”中“n”是小写）。

# 我们需要更多的代码来得到Person的name

把getAttribute()的结果发送给打印/书写语句，得到的并不是我们真正想要的东西，它只是运行了Person对象的toString()方法，而且因为Person类没有覆盖其继承的Object.toString()，所以，你已经知道这会得到什么结果。而我们实际上只想打印Person的name。

## JSP代码

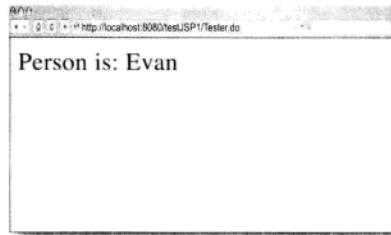
```
<html><body>
<% foo.Person p = (foo.Person) request.getAttribute("person"); %>
Person is: <%= p.getName() %>
</body></html>
```

↑  
打印 getName() 的  
结果。

或者使用一个表达式

```
<html><body>
Person is:
<%= ((foo.Person) request.getAttribute("person")).getName() %>
</body></html>
```

我们会得到：



不过，还记得那个备忘录吗……

可以把它用一句话来总结：“使用脚本则死。”

所以我们需要另外一种方法。

## Person是一个JavaBean，所以我们要使用与bean相关的标准动作

通过使用几个标准动作，就可以消除这个JSP中的所有脚本代码（要记住：脚本代码包括声明、scriptlet和表达式），而且仍能打印出person属性的name性质值。不要忘记，name不是属性（attribute），person对象才是属性。name只是从Person的getName()方法返回的一个性质。

### 不使用标准动作（使用脚本）

```
<html><body>

<% foo.Person p = (foo.Person) request.getAttribute("person"); %>
Person is: <%= p.getName() %>

</body></html>
```

前面用的就是这种办法。

### 使用标准动作（不使用脚本）

```
<html><body>

<jsp:useBean id="person" class="foo.Person" scope="request" />
Person created by servlet: <jsp:getProperty name="person" property="name" />
</body></html>
```

这里没有Java代码！没有脚本，只有两个标准动作标记而已。

## 分析<jsp:useBean>和<jsp:getProperty>

我们真正想要的正是<jsp:getProperty>的功能，因为我们只想显示person的“name”性质值。但是容器怎么知道“person”是什么含义呢？如果JSP中只有<jsp:getProperty>标记，这就像使用了一个未声明的“person”变量。容器通常并不知道你在说什么，除非你先在页面中放一个<jsp:useBean>。<jsp:useBean>可以用来声明和初始化你在<jsp:getProperty>中使用的具体bean对象。

### 用<jsp:useBean>声明和初始化一个bean属性

```
<jsp:useBean id="person" class="foo.Person" scope="request" />
```

标识标准动作  
声明bean对象的标识符。这对应于Web servlet代码中所用的名：  
W.下 servlet 代码中所用的名：  
request.setAttribute("person", p);

声明bean对象的类类型 (当然是完全限制名)。  
标识这个bean对象的属性作用域。

### 用<jsp:getProperty>得到bean属性的性质值

```
<jsp:getProperty name="person" property="name" />
```

标识标准动作  
标识具体的bean对象。这与<jsp:useBean>标记的"id"值匹配。  
注意：这个 "name" 性质与标记中 name="person" 部分中的 "name" 没有任何关系。这个性质之所以称为 "name"，只是因为Person类就是这样定义的。

## <jsp:useBean>还能创建一个bean!

如果<jsp:useBean>找不到一个名为“person”的属性对象，它就会建一个！request.getSession()（或getSession(true)）也是这样做的，首先查找已有的，但是如果找不到，就会创建一个。

来看生成的servlet的代码，可以看到发生了什么，这里有一个if测试！它根据标记中id和scope的值查找一个bean，如果找不到，就会为class中指定的类建立一个实例，把这个对象赋给id变量，然后把它设置为指定作用域（标记中定义的scope）中的一个属性。

这个标记

```
<jsp:useBean id="person" class="foo.Person" scope="request" />
```

会变成 \_jspService()方法中的以下代码

```

    根据id的值声明一个变量，从而允许JSP中的其他部分（包括其他bean标记）引用这个变量。
foo.Person person = null; ↗

    尝试在标记中定义的作用域中得到属性，并把结果赋给id变量。
synchronized (request) { ↗

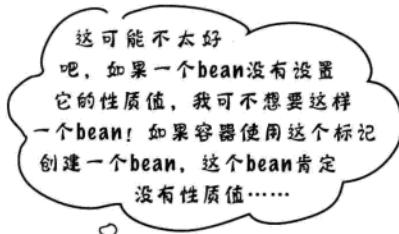
    person = (foo.Person) _jspx_page_context.getAttribute("person", PageContext.REQUEST_SCOPE); ↗

    不过，如果这个作用域中没有这样一个属性……
if (person == null){ ↗

    person = new foo.Person(); ↗
        建立一个实例，并把它赋给id变量。
    _jspx_page_context.setAttribute("person", person, PageContext.REQUEST_SCOPE);
}

    最后，把这个新对象设置为所定义作用域中的一个属性。

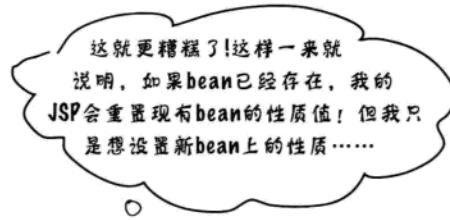
```



## 可以使用`<jsp:setProperty>`

不过，你已经知道了，有get就肯定有set。`<jsp:setProperty>`标记是第3个（也是最后一个）bean标准动作。使用很简单：

```
<jsp:useBean id="person" class="foo.Person" scope="request" />
<jsp:setProperty name="person" property="name" value="Fred" />
```



## <jsp:useBean>可以有体!

如果把设置方法代码(<jsp:setProperty>)放在<jsp:useBean>的体中，这样就能有条件地设置性质！换句话说，只有创建新bean时才会设置性质值。如果发现对于指定的scope和id已经一个bean，就不会运行<jsp:useBean>标记的体，所以你的JSP代码不会重置这个性质。

利用<jsp:useBean>体，可以有条件地运行代码……只有找不到bean属性，而且创建了一个新bean时才会运行体中的代码。

<jsp:useBean id="person" class="foo.Person" scope="page">  
<jsp:setProperty name="person" property="name" value="Fred" />  
</jsp:useBean >  
↑  
最后结束这个标记。开始和结束标记  
之间的所有内容都是体。

没有斜线！  
这是标记的体

<jsp:useBean>体中的代码会有条件地运行，只有  
找不到bean而且创建一个新bean时才会运行。

**问：**为什么不直接指定bean构造函数的参数？为什么绕那么一个大弯子来设置值呢？

**答：**这个问题的答案很简单：bean没有带参数的构造函数！没错，作为Java类，构造函数可以有参数，但是如果一个对象要作为bean，根据bean法则，只能调用bean的一个无参数的公共构造函数。只能如此。实际上，如果bean类中没有一个无参数的公共构造函数，就会全盘失败。

**问：**到底什么是bean法则？

**答：**就是遵循“古老”JavaBeans规范的法则。我们说的是JavaBean，而不是企业JavaBean（Enterprise JavaBean，EJB），这两个东西完全不相干（要搞清楚）。普通的非企业JavaBean规范定义了一个类怎么才能算是JavaBean。尽管这个规范确实很复杂，不过，结合JSP和servlet使用bean时，你只要知道以下规则就行了（我们只列出了与使用servlet和JSP相关的规则）：

- 1) 必须有一个无参数的公共构造函数。

2) 必须按命名约定来命名公共的获取方法和设置方法。首先是“get”（或者如果是一个布尔性质，获取方法前缀则是“is”）和“set”，后面是同一个词（getFoo/setFoo()）。要得到性质名，先去掉“get”和“set”，把余下部分的第一个字母变成小写。

3) 设置方法的参数类型和获取方法的返回类型必须一样。这定义了性质的类型。

int getFoo() void setFoo(int foo)

4) 性质名和类型是由获取方法和设置方法得出，而不得自于类中的一个成员。例如，如果你有一个私有的i foo变量，这并不意味着存在这样一个性质。变量可以你喜欢的任何名字。“foo”这个性质名是从方法得的。换句话说，如果说有一个性质，只是因为你有一对取方法和设置方法。具体如何实现由你决定。

5) 结合JSP使用时，性质类型必须是String，或者是一个本类型。如果不是这样，尽管也许是一个合法的bean，是如此一来，你可能还得使用脚本，而不能完全摆脱脚本使用标准动作。

# <jsp:useBean>有体时生成的servlet

很简单。容器把额外的性质设置代码都放在if测试块中。

## 有<jsp:useBean>体的相应\_jspService()代码

```

foo.Person person = null;           ← 声明引用变量。
person = (foo.Person) _jspx_page_context.getAttribute("person", PageContext.PAGE_SCOPE);
if (person == null){               ← 如果没找到，则创建
    person = new foo.Person();     ← 一个新实例。
    _jspx_page_context.setAttribute("person", person, PageContext.PAGE_SCOPE);
}

org.apache.jasper.runtime.JspRuntimeLibrary.introspecthelper(
    _jspx_page_context.findAttribute("person"), "name", "Fred", null, null, false);
}

```

你可能以为代码应当是：

```
person.setName("Fred");
```

以上代码的作用正是如此。只不过它使用了一个通用的性质设置方法，这个方法取属性、性质以及性质值为参数。最终结果还是一样的：最后它会在Person对象上调用setName()。

(要记住，你不需要知道Tomcat生成的实现代码是什么……你要的只是最终结果)。

查找有标记中指定的名  
和作用域的现有属性。

把这个新的bean对象绑定  
到指定的作用域。

这一部分是新的。只有当useBean有  
体时才会有这一部分。



# 可以建立多态的bean引用吗？

写<jsp:useBean>时，class属性确定了新对象的类（如果会创建新对象）。它还确定了所生成servlet中使用的引用变量的类型。

JSP中现在是这样：

```
<jsp:useBean id="person" class="foo.Person" scope="page" />
```

生成的servlet

```
foo.Person person = null;
// 得到person属性的代码
if (person == null){
    person = new foo.Person();
...
}
```

标记中的class属性既表示引用类型，又表示对象类型。

但是……如果希望引用类型不同于具体的对象类型该怎么办？我们要把Person类修改为抽象类，再建立一个具体的子类Employee。假设希望引用类型是Person，而新对象的类型是Employee。

```
package foo;

public abstract class Person {
    private String name;

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

    public String getName() {
        return name;
    }
}
```

```
package foo;

public class Employee extends Person {
    private int empID;

    public void setEmpID(int empID) {
        this.empID = empID;
    }

    public int getEmpID() {
        return empID;
    }
}
```

# 为<jsp:useBean>增加一个type属性

抽象类 →

如果只对Person类做了修改，倘若没有发现所找的属性，就会有麻烦了：

原来的JSP

```
<jsp:useBean id="person" class="foo.Person" scope="page"/>
```

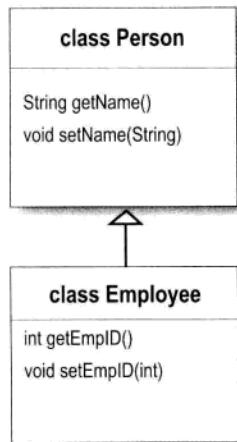
有这样的结果

```
java.lang.InstantiationException: foo.Person
```

因为容器想要运行：

```
new foo.Person();
```

Person现在是抽象类！显然，你无法建立抽象类的实例。但是容器还是会根据标记中的class属性徒劳地尝试。



要让引用变量类型为Person，而对象是Employee类的一个实例。为此，可以向标记增加一个type属性。

增加了type属性的新JSP

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee" scope="page">
```

生成的servlet

```
foo.Person person = null;
// 得到person属性的代码
if (person == null){
    person = new foo.Employee();
...
}
```

现在引用类型是抽象类Person，对象类型是具体的子类Employee。

type可以是class类型、抽象类型或者是一个接口，只要能用作为bean对象class类型的声明引用类型，都可以指定为type。当然，不能违反Java的类型规则。如果class类型无法赋给引用类型(type)，就有麻烦了。所以，这意味着class必须是type的一个子类或具体实现。

## 使用type, 但没有class

如果声明了一个type，但是没有class会怎么样？

type是抽象类或是具体类有区别吗？

JSP

```
<jsp:useBean id="person" type="foo.Person" scope="page"/>
```

没有class, 只有type

如果“page”作用域中已经存在person属性

就能正常工作。

如果“page”作用域中不存在person属性，结果如下：

不能工作！！

```
java.lang.InstantiationException: bean person not found within scope
```

如果使用了type，但没有class，bean必须已经存在。

如果使用了type（或没有type），class不能是抽象类，而且必须有一个无参数的公共构造函数。

**问：** 在你的例子里，“foo.Person”是一个抽象类型，它当然不能实例化。但如果把type改成“foo.Employee”，会怎么样呢？这个type会同时用作为引用类型和对象类型吗？

**答：** 不行！还是无法工作。如果容器发现这个bean不存在、而且只有type属性而没有class属性，它就会明白你只给了它需要的一半，有引用类型，而没有对象类型。换句话说，你没有告诉它要建立哪个类的新实例！

没有这样一个退路：“如果找不到对象，可以继续、把type同时用作为引用类型和对象类型”。这是不行的，它的做法并非如此。

关键是：如果使用了type而没有class，最好保证已经将bean存储为一个属性，要放在标记中指定的scope中，而且有标记中指定的id。

## scope属性默认为“page”

如果在<jsp:useBean>或<jsp:getProperty>标记中没有指定作用域，容器会使用默认作用域“page”。

以下JSP：

```
<jsp:useBean id="person" class="foo.Employee" scope="page"/>
```

等价于：

```
<jsp:useBean id="person" class="foo.Employee"/>
```



**不要把type与class弄混了！**

查看下面的代码：

```
<jsp:useBean id="person" type="foo.Employee" class="foo.Person"/>
```

应该能看出来，这是无法工作的！你会得到一个庞大的异常：

```
org.apache.jasper.JasperException: Unable to compile class for JSP
foo.Person is abstract; cannot be instantiated
```

```
Person = new foo.Person();
```

一定要记住

**type == 引用类型**

**class == 对象类型**

或者换种说法：

**type 是你声明的类型（可以是抽象类）**

**class 是你要实例化的类（必须是具体类）**

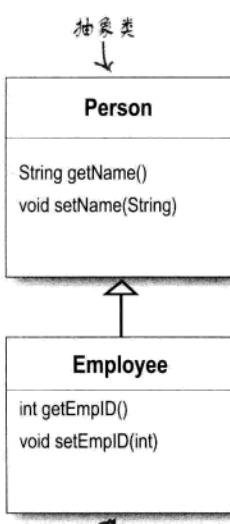
**type x = new class()**

现在你可能会这样想：“那好，class总是一个类，而type不一定是类，type还可以是一个接口。既然前者肯定是类，所以当然他们用“class”来表示，另外既然后者还可以是接口，所以用“type”表示”。你说的没错。但是你可能还会考虑到，“规范中不是所有东西都有最直观、最明显的名字，所以我还是小心为妙。”有时，一个东西的名字与它的实际含义恰好相反，比如有关安全的<auth-constraint>。但是不管怎么说，在这里class就是类，type就是……类型。



请看这个标准动作：

```
<jsp:useBean id="person" type="foo.Employee" scope="request" >
    <jsp:setProperty name="person" property="name" value="Fred" />
</jsp:useBean >
Name is: <jsp:getProperty name="person" property="name" />
```



现在假设有一个servlet会做一些工作，然后把请求转发到有以上代码的JSP。

针对两个不同的servlet代码示例，得出以上JSP代码分别有什么结果（答案在这一章的最后）。

① 如果servlet代码如下，会怎么样？

```
foo.Person p = new foo.Employee();
p.setName("Evan");
request.setAttribute("person", p);
```

具体类

(这两个类都在包  
"foo" 中)。

② 如果servlet代码如下，会怎么样？

```
foo.Person p = new foo.Person();
p.setName("Evan");
request.setAttribute("person", p);
```



## 直接从请求到JSP，没有经过servlet……

假设我们的表单是这样的：

```
<html><body>
<form action="TestBean.jsp">
    name: <input type="text" name="userName">
    ID#: <input type="text" name="userID">
    <input type="submit">
</form>
</body></html>
```

↑ 请求直接到达JSP。

我们知道，结合使用标准动作和脚本就可以做到这一点：

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee"/>
<% person.setName(request.getParameter("userName")); %>
```

甚至可以在标准动作内部使用脚本做到：

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
<jsp:setProperty name="person" property="name"
    value="<% request.getParameter("userName") %>" />
</jsp:useBean>
```

是的，你看到的没错，确实是把一个表达式放在`<jsp:setProperty>`标记中（`<jsp:setProperty>`刚好在`<jsp:useBean>`标记的体中）。必须承认，这看上去确实不好看。

## 解决之道： param 属性

非常简单。可以直接向bean发送一个请求参数，不用脚本，只用param属性就可以。

利用 param 属性，可以把bean的性质值设置为一个请求参数的值。只需指定请求参数！

在TestBean.jsp中

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
    <jsp:setProperty name="person" property="name" param="userName" />
</jsp:useBean>
```

```
<html><body>

<form action="TestBean.jsp">
    name: <input type="text" name="userName">
    ID#: <input type="text" name="userID">
    <input type="submit">
</form>

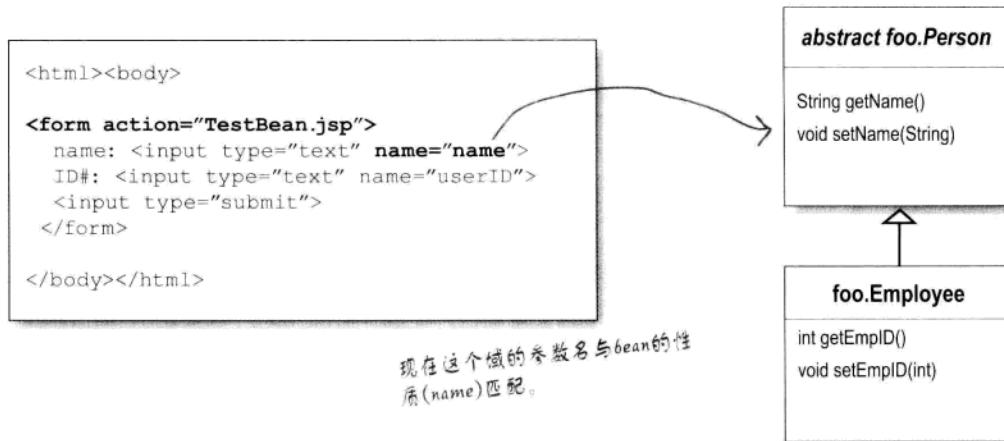
</body></html>
```

param值“userName”来自  
表单输入域的name属性。

## 请等等！还能更好……

你要做的只是确保表单输入域名（即请求参数名）与bean中的性质名相同。如果是这样，你就不必在<jsp:setProperty>标记中指定param属性了。如果指定了property，但是没有指定value或param，就是在告诉容器，要从有匹配名的请求参数得到值。

如果修改HTML，让输入域名与性质名匹配：



可以这样做：

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
  <jsp:setProperty name="person" property="name" />
</jsp:useBean>
```

↑  
没有指定任何值！

如果请求参数名与bean性质名匹配，就不需要在<jsp:setProperty>标记中为该性质指定值。

## 如果愿意，还能更好……

如果让所有请求参数名都与bean性质名匹配，看看会怎么样。person bean（这是foo.Employee的一个实例）具体有两个性质：name和empID。

如果再来修改HTML

```
<html><body>
<form action="TestBean.jsp">
    name: <input type="text" name="name">
    ID#: <input type="text" name="empID">
    <input type="submit">
</form>
</body></html>
```

现在，两个参数都与bean中的性质名匹配。

**abstract foo.Person**

```
String getName()
void setName(String)
```

**foo.Employee**

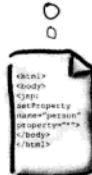
```
int getEmpID()
void setEmpID(int)
```

可以这样做：

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
    <jsp:setProperty name="person" property="" />
</jsp:useBean>
```

这多酷呀！！

我希望你迭代处理所有请求参数，找到与这个bean的性质名相匹配的所有参数，再把所有匹配性质的值设置为相应请求参数的值……



JSP

噢，你真够可以的……所有工作都让我来做。我得查看bean类的获取方法和设置方法来得出bean性质，还要与参数名匹配……



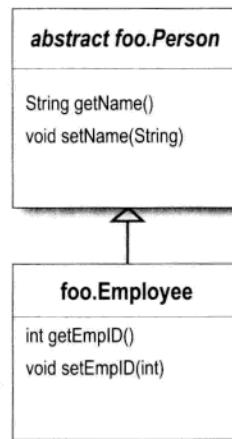
容器

# Bean标记会自动转换基本类型的性质

如果你以前已经熟悉JavaBean，就不会感到奇怪。Java-Bean的性质可以是任何东西，但是如果是String或基本类型，就能自动完成类型强制转换。

这样很好，你不用自己来完成解析和转换。

如果把type指定为Employee（而不是Person）



```

<html><body>

<jsp:useBean id="person" type="foo.Employee" class="foo.Employee" >
  <jsp:setProperty name="person" property="" />
</jsp:useBean>

Person is: <jsp:getProperty name="person" property="name" />
ID is: <jsp:getProperty name="person" property="empID" />

</body></html>
  
```

现在生成的servlet中则有：  
 Employee person = new Employee();  
 而不是：  
 Person person = new Employee();

这是可以的：

<jsp:setProperty> 动作取String请求参数，将其转换为一个int，再把这个int传给该性质的相应设置方法。



there are no  
Dumb Questions

**问：** OK，我想容器代码可能调用了`Integer.parseInt("343")`，那么，如果用户键入的东西不能解析为int，会不会得到一个`NumberFormatException`异常？比如说，如果用户在员工ID域中键入了“three”会怎么样呢？

**答：** 问得好。不错，如果`empID`性质的相应请求参数不能解析为int，肯定会出错。你要验证这个域的内容，确保其中只包含数字字符。可以把表单数据先发给一个servlet，而不是直接发给JSP。但是如果你从表单直接提交到JSP，而且不想用脚本，可以在HTML表单中使用JavaScript先检查这个域，然后再发送请求。如果你对JavaScript不熟悉（当然这与Java基本上没有任何关系），这只是一个简单的脚本语言，要在客户端处理。换句话说，它会由浏览器处理。在Google上搜索一下“JavaScript validate input field”（JavaScript验证输入域），就能很快找到一些示例脚本，可以使用这些脚本禁止用户输入不合法的内容，如只能向输入域中输入数字。

**问：** 如果bean性质不一定是String或基本类型，不用脚本的话，怎样设置这样一个性质呢？标记的`value`属性总是String，对吗？

**答：** 完全可以创建一个支持bean的特殊类，称作定制属性编辑器，不过这可能需要多做很多工作。它取得String值，并得出如何对其解析从而能用来设置一个更复杂的类型。不过，这是JavaBeans规范的一部分，不属于JSP规范。另外`<jsp:setProperty>`标记中的`value`属性是一个表达式，而不是一个String直接量，如果这个表达式计算为一个对象，它与bean性质的类型兼容，也许也可以工作。例如，如果传入一个表达式，计算为一个Dog，就会调用Person bean的`setDog(Dog)`方法。但是想想看，这意味着Dog对象必须已经存在。另外，最好不要在JSP中构造新的东西！如果不使用脚本，即使是构造和设置稍有些复杂的数据类型，也可能很困难（这些内容考试中都不会考）。



如果你使用脚本，就不会自动完成String到基本类型的转换！即使表达式在`<jsp:setProperty>`标记中也会失败。

如果使用`<jsp:setProperty>`标准动作标记，并将其`property`设置为通配符；或者只有性质名而没有`value`或`param`属性（这说明，该性质名与请求参数名匹配）；或者使用一个`param`属性来指示请求参数，要把这个请求参数值赋给bean的性质；再或者键入了一个直接量值，在这些情况下，就会自动从String转换为int。这些例子都会自动地完成转换：

```
<jsp:setProperty name="person" property="*" />
<jsp:setProperty name="person" property="empID" />
<jsp:setProperty name="person" property="empID" value="343" />
<jsp:setProperty name="person" property="empID" param="empID" />
```

但是……如果你使用脚本，就不会完成自动转换：

```
<jsp:setProperty name="person" property="empID" value="<% request.getParameter("empID") %>" />
```

这些都会自动转换！

这不会转换！



### bean标准动作标记对非程序员更为自然。

再说一次, 使用标记而不是脚本, 这对Web页面设计人员的意义比对你(Java程序员)更重大。不过即使是Java程序员也能发现, 标记比硬编码的脚本元素更容易维护。利用这些与bean相关的标记, 设计人员只需要基本的标识信息(属性名、作用域和性质名)。当然, 他们确实要知道完全限定类名, 但是对Web页面设计人员而言, 这只是一个名字, 只不过里面有点号(.)而已。Web设计人员不需要了解底层的知识, 他们可以把bean想成是包含字段的记录。你要把记录(类和标识符)和字段(性质)告诉设计人员。

不过, bean标准动作还不是尽善尽美的。

正是因为这个原因, 所以关于无脚本页面的故事还没有完。请继续读下去……

# 但是如果性质不是String或基本类型呢？

如果属性本身就是一个String，前面已经知道了，打印这个属性相当简单。接下来我们建立了另外一种属性，这是一个非String的对象（一个Person bean实例）。但是我们不想打印这个属性本身（person），而只想打印属性的一个性质（在这个例子中，就是person的name和empID）。这是可以的，因为标准动作能处理String和基本类型的性质。所以，我们知道，只要属性的所有性质都是String或基本类型，标准动作就可以处理任何类型的属性。

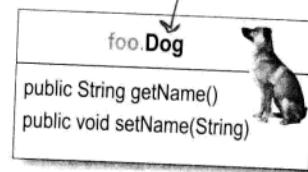
但是如果不是这样呢？如果bean有一个性质不是String或基本类型，会怎么样呢？如果性质也是Object类型呢？这个Object类型又有自己的性质呢？

如果我们真正想打印的是那个性质的性质，该怎么办？

Person有一个String“name”性质。

Person有一个Dog“dog”性质。

Dog有一个String“name”性质。



## 如果想打印Person的dog的name呢？

### Servlet代码

```

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

    foo.Person p = new foo.Person();
    p.setName("Evan");

    foo.Dog dog = new foo.Dog();
    dog.setName("Spike");
    p.setDog(dog);

    request.setAttribute("person", p);

    RequestDispatcher view = request.getRequestDispatcher("result.jsp");
    view.forward(request, response);
}

```

PDG

这一次建立了一个Dog，指定了名字，并在Person上调用setDog()。

既然Person的“dog”性质有一个Dog值，我们把Person（只是Person）设置为一个请求属性。

# 显示性质的性质

我们知道，要显示性质的性质，这可以用脚本来做到，但是能不能用bean标准动作完成呢？如果把“dog”作为<jsp:getProperty>标记中的一个性质会怎么样呢？

## 不用标准动作（使用脚本）

```
<html><body>
<%= ((foo.Person) request.getAttribute("person")).getDog().getName() %>
</body></html>
```

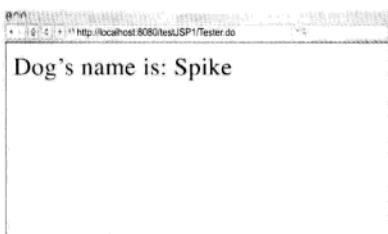
这能很好地工作……  
但是必须使用脚本。

## 使用标准动作（不使用脚本）

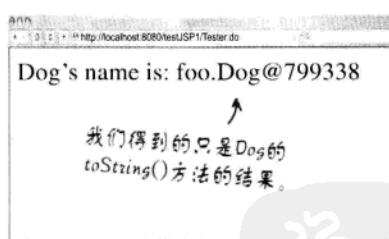
```
<html><body>
<jsp:useBean id="person" class="foo.Person" scope="request" />
Dog's name is: <jsp:getProperty name="person" property="dog" />
</body></html>
```

但是“dog”的值是  
什么？

我们希望得到的结果：



我们实际得到的结果：



不能这么说：property=“dog.name”

对于前面给定的servlet代码，任何bean标准动作的组合都不能正常工作，因为Dog不是一个属性！Dog是属性的一个性质，所以可以显示Dog，但是不能访问Person属性的Dog性质的name性质。

利用<jsp:getProperty>，只能访问bean属性的性质。它不能访问嵌套性质，你想要的是性质的性质，而不是属性的性质。

## 救星：表达式语言（EL）！

是的，救星出现了，它来得正是时候。JSP表达式语言（EL）已经增加到JSP 2.0规范中，这样我们就不用受脚本的专制了。

看看我们的JSP现在多简单……

没有脚本的JSP代码，使用EL

```
<html><body>  
Dog's name is: ${person.dog.name}  
</body></html>
```

使用EL，打印嵌套性质变得非常容易……换句话说，可以很轻松地打印性质的性质！

就这么简单！我们甚至没有声明person是什么意思……它自己已经知道。

用这个代码：

```
${person.dog.name}
```

替换以下代码：

```
<%= ((foo.Person) request.getAttribute("person")).getDog().getName() %>
```



Relax

不需要了解EL的一切。

考试不要求你是一个全面的EL专家。

你一般要用到的，或者说可能要考到的，都会在后面几页谈到。所以，如果想全面学习EL规范的所有内容，你会受不了的。到时候你就会明白为什么我们不要求你全盘了解了。

# JSP表达式语言 (EL) 剖析

表达式语言的语法相当简单，变化也不太大。难的是，有些EL看上去很像Java，但表现却不同。稍后谈到[]操作符的时候你就会了解到。所以你可能会发现，有些在Java中能用的，到了EL中却不行，反之亦然。所以不要想着把Java语言语法规则硬往EL上套，明确这一点就行了。在后面几页中，可以把EL想成是不使用Java来访问Java对象的一种方法。

EL表达式总是放在大括号里，而且前面有一个美元符前缀。

`${person.name}`

表达式中第一个命名变量可以是一个  
隐式对象，也可以是一个属性。

`${firstThing.secondThing}`

EL隐式对象

或

属性

{  
pageScope  
requestScope  
sessionScope  
applicationScope}

页面作用域中的属性  
请求作用域中的属性  
会话作用域中的属性  
应用作用域中的属性

所有这些都  
是映射对象

param  
paramValues  
header  
headerValues

cookie  
initParam

pageContext

在所有映射对象中，只有  
pageContext不是映射。这是  
pageContext对象的实际引用！  
(pageContext是一个JavaBean)。

注意：EL隐式对象与JSP脚本中可用的映射  
对象不同，只有pageContext除外。

如果EL表达式中第一项是一个属  
性，可以是存储在任意一个可  
用作用域中的属性。

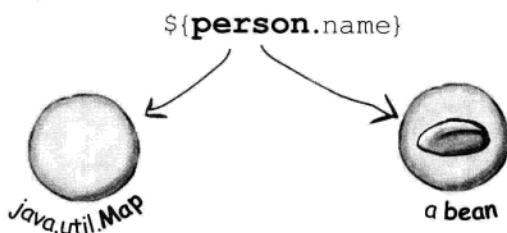
PDG

(Java提示：映射 (map)  
是一个保存键/值对  
的集合，如Hashtable和  
HashMap)。

## 使用点号(.)操作符访问性质和映射值

第一个变量可以是一个隐式对象，也可以是一个属性，点号右边可以是一个映射键（如果第一个变量是映射），也可以是一个bean性质（如果第一个变量是JavaBean属性）。

- ① 如果表达式中变量后面有一个点号，点号左边的变量必须是一个Map或一个bean。



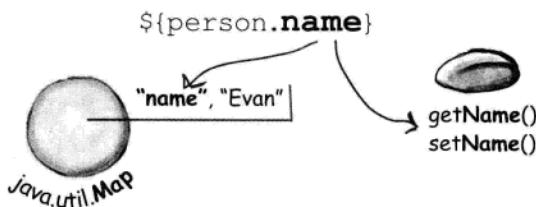
点号左边的变量要么是一个Map（有键），要么是一个bean（有性质）。

不论变量是一个隐式对象还是一个属性，都是如此。

pageContext隐式对象是一个bean，它有`getAttribute()`方法，所有其他隐式对象都是Map。

如果对象是一个bean，但是指定的性质不存在，会抛出一个异常。

- ② 点号右边必须是一个Map键或一个bean性质。



- ③ 点号右边必须遵循Java有关标识符的命名规则。

`${person.name}`

- \* 必须以字母、\_或\$开头。
- \* 第一个字母后面可以有数字。
- \* 不能是Java关键字。

## [ ]就像是更好的点号

只有当点号右边是左边变量的一个bean性质或映射键时，点号操作符才能正常工作。仅此而已，但[ ]操作符就强大多了，而且更加灵活……

以下代码：

```
 ${person["name"]}
```

与下面的代码等价：

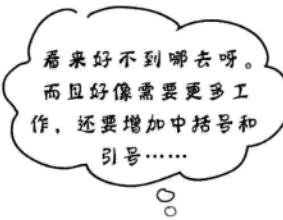
```
 ${person.name}
```

简单的点号操作符之所以能工作，这是因为person是一个bean，而且name是person的一个性质。

但是如果person 是一个数组呢？

或者如果person 是一个List呢？

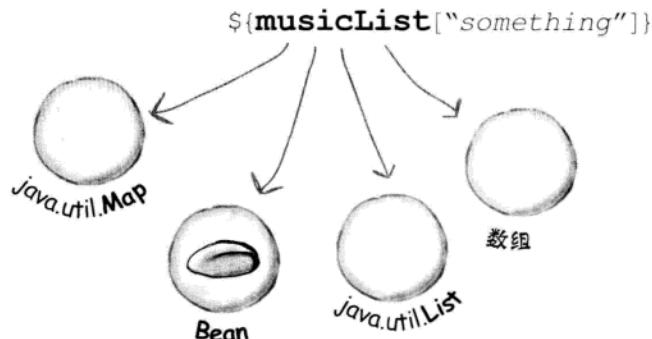
再或者，name不遵循正常的Java命名规则怎么办？



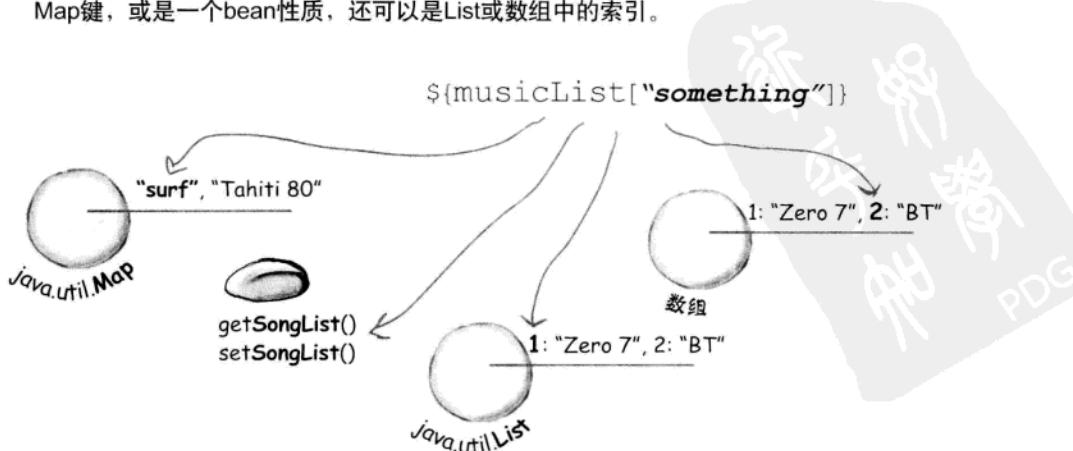
## [ ]让你有更多选择……

使用点号操作符时，左边只能是一个Map或一个bean，右边必须遵循Java关于标识符的命名规则。但是，使用[ ]时，左边还可以是一个List或数组（可以是任何类型的数组）。这也说明，右边可以是一个数，或者是可以解析为一个数，也可以是不遵循Java命名规则的标识符。例如，Map键可以是一个包含点号的String(“com.foo.trouble”）。

- ① 如果表达式中变量后有一个中括号[]，左边的变量则有更多选择，可以是Map、bean、List或是数组。



- ② 如果中括号里是一个String直接量（即用引号引起的串），这可以是一个Map键，或是一个bean性质，还可以是List或数组中的索引。



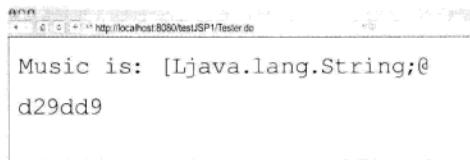
# 对数组使用[]操作符

Servlet中

```
String[] favoriteMusic = {"Zero 7", "Tahiti 80", "BT", "Frou Frou"};
request.setAttribute("musicList", favoriteMusic);
```

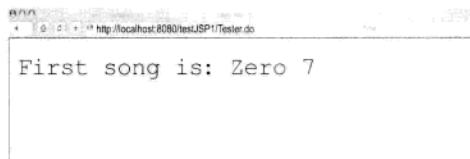
JSP中

Music is: \${musicList}



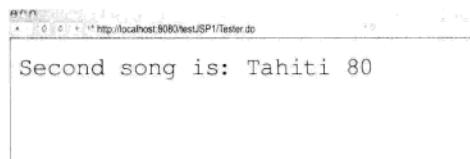
可以在数组上调用  
toString()。

First song is: \${musicList[0]}



你开玩笑吧？怎么有两种  
不同的做法，一种索引不加  
引号，另一种索引加引号……我  
不相信数组索引要加引号，这  
样不对吧，老兄……

Second song is: \${musicList["1"]}



怎么回事？



# 数组和List中的String索引会强制转换为int

访问数组的EL与访问List的EL是一样的。

大家要记住，这不是Java。在EL中，`[]`操作符并不是数组访问操作符。不是这样的，它只是叫做`[]`操作符（我们敢保证，你可以在规范里查查看，它根本没有名字！只是一个符号`[]`，仅此而已）。如果实在要给它一个名字，就应该是“数组/List/Map/bean性质访问操作符”。

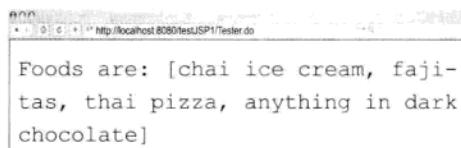
*Servlet中：*

```
java.util.ArrayList favoriteFood = new java.util.ArrayList();
favoriteFood.add("chai ice cream");
favoriteFood.add("fajitas");
favoriteFood.add("thai pizza");
favoriteFood.add("anything in dark chocolate");
request.setAttribute("favoriteFood", favoriteFood);
```

*JSP中：*

Foods are: \${favoriteFood}

虽然，ArrayList已经很好地覆盖了toString()。



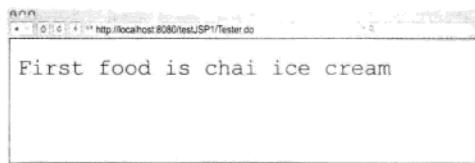
如果中括号左边是一个数组或List，而且索引是一个String直接量，那么这个索引会强制转换为int。

下面这样不行：

`${favoriteFood["one"]}`

因为“one”不能转换为一个int。如果索引无法强制转换，你就会得到一个错误。

First food is \${favoriteFood[0]}

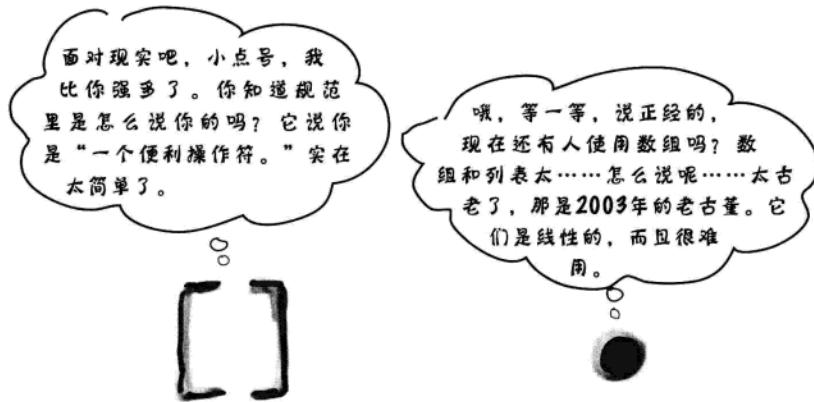


这是对的

Second food is \${favoriteFood["1"]}



真是奇怪，不过权且如此吧……如果非得这样，我也只好入乡随俗了。



## 对于bean和Map，这两个操作符都可以用

JavaBean和Map可以使用[]操作符，也可以使用方便的点号操作符。可以把映射键想成是bean中的性质名。

指定键或性质的名时，可以得到键或性质的值。

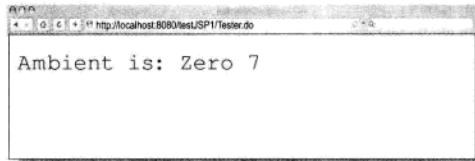
Servlet中：

```
java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti 80");
musicMap.put("DJ", "BT");
musicMap.put("Indie", "Travis");
request.setAttribute("musicMap", musicMap);
```

} 建立一个Map，在其中放一些String键和对象，然后把这个Map设置为一个请求属性。

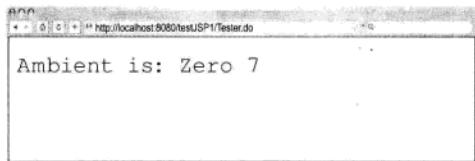
JSP中：

Ambient is: \${musicMap.Ambient}



这两个表达式都使用Ambient作为Map的键（因为musicMap是一个Map!）。

Ambient is: \${musicMap["Ambient"]}



# 如果不是String直接量，就会计算

如果中括号里没有引号，容器就会计算中括号中的内容，搜索与该名字绑定的属性，并替换为这个属性的值（如果有一个同名的隐式对象，那么总是使用隐式对象）。

Music is: \${musicMap[Ambient]}

Ambient没有加引号，这样是不行的！因为没有名为“Ambient”的绑定属性，结果会返回为null.....

查找一个名为“Ambient”的属性。

使用这个属性的值作为Map的键，或者返回null。

Servlet中：

```
java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti 80");
musicMap.put("DJ", "BT");
musicMap.put("Indie", "Frou Frou");

request.setAttribute("musicMap", musicMap);

request.setAttribute("Genre", "Ambient");
```

在JSP中这是可以的：

Music is \${musicMap[Genre]} 计算为 Music is \${musicMap["Ambient"]}

由于有一个名为“Genre”的请求属性，它的值为“Ambient”，而且“Ambient”是musicMap的一个键。

在JSP中下面这样是不行的：(给定以上servlet代码)

Music is \${musicMap["Genre"]} 不变 Music is \${musicMap["Genre"]}

因为musicMap中没有名为“Genre”的键。由于加了引号，容器不会进行计算，而只认为这是一个直接量键名。

这是合法的EL表达式，但是它所做的与我们预想的不一样。

# 在中括号里可以使用嵌套表达式

EL中都是表达式。可以任意嵌套表达式，深度不限。换句话说，可以把一个复杂的表达式放在另一个复杂表达式中，而后者可以再放在一个表达式中……如此继续。表达式会从最内层中括号开始计算。

这一部分对你来说相当直观，因为这与在小括号里嵌套Java代码没有什么不同。麻烦的是，要当心加不加引号。

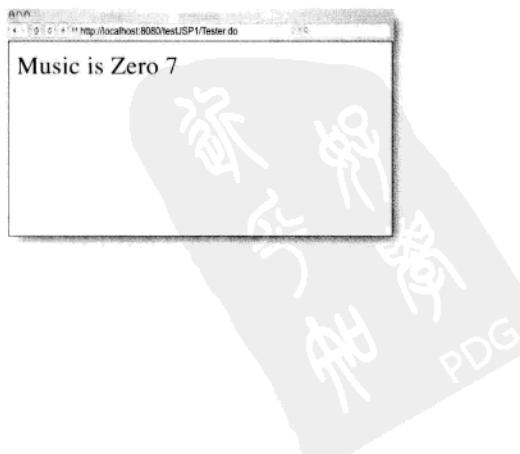
Servlet中：

```
java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti 80");
musicMap.put("DJ", "BT");
musicMap.put("Indie", "Frou Frou");
request.setAttribute("musicMap", musicMap);

String[] musicTypes = {"Ambient", "Surf", "DJ", "Indie"};
request.setAttribute("MusicType", musicTypes);
```

JSP中这样是可以的：

```
Music is ${musicMap[MusicType[0]]}
          ↓
          变成
          ↓
Music is ${musicMap["Ambient"]}
          ↓
          变成
          ↓
Music is Zero 7
```



# `\${foo.1}`是不行的

bean和Map可以使用点号操作符，但条件是你在点号后面键入的东西必须是一个合法的Java标识符。

这个EL

`${musicMap.Ambient}` 可以

与以下EL等价

`${musicMap["Ambient"]}` 可以

如果不能用作为Java代码中的变量名，就不能把它放在点号后面。

但是以下EL

`${musicList[1]}`

不能转换成：

`${musicList.1}` 不行！绝对不行！

## Sharpen your pencil

会打印什么结果？

给定以下servlet代码，明确会打印什么结果（或者，如果有错误，可以只写“error”，这你应该知道）。答案在下一页的最后。

```
java.util.ArrayList nums = new java.util.ArrayList();
nums.add("1");
nums.add("2");
nums.add("3");
request.setAttribute("numbers", nums);
String[] favoriteMusic = {"Zero 7", "Tahiti 80", "BT", "Frou Frou"};
request.setAttribute("musicList", favoriteMusic);
```

①  `${musicList[numbers[0]]}`

②  `${musicList[numbers[0]+1]}`

(后面几页还会讨论  
更多EL操作符)

③  `${musicList[numbers["2"]]}`

④  `${musicList[numbers[numbers[1]]]}`

## 代码贴



如果考试的时候看到这样的题目，请不要奇怪（不过，在实际考试中，题目看上去可能……更丑陋）。

仔细研究这一页上的3个类，以及下一页上的servlet代码，然后组织代码贴，建立适当的EL从而能生成浏览器中显示的响应（翻一页就能看到答案，但是你要做完这个练习之后再看答案，特别是如果你想参加考试，一定要好好做这个练习）。

### foo.Toy



```
package foo;
public class Toy {
    private String name;
    public void setName(String name) {
        this.name=name;
    }
    public String getName() {
        return name;
    }
}
```

### foo.Person



```
package foo;
public class Person {
    private Dog dog;
    private String name;
    public void setDog(Dog dog) {
        this.dog=dog;
    }
    public Dog getDog() {
        return dog;
    }
    public void setName(String name) {
        this.name=name;
    }
    public String getName() {
        return name;
    }
}
```

### foo.Dog



```
package foo;
public class Dog {
    private String name;
    private Toy[] toys;
    public void setName(String name) {
        this.name=name;
    }
    public String getName() {
        return name;
    }
    public void setToys(Toy[] toys) {
        this.toys=toys;
    }
    public Toy[] getToys() {
        return toys;
    }
}
```

前一章大练习的答案：1) Tom 2) Bill 3) Fred 4) Bob

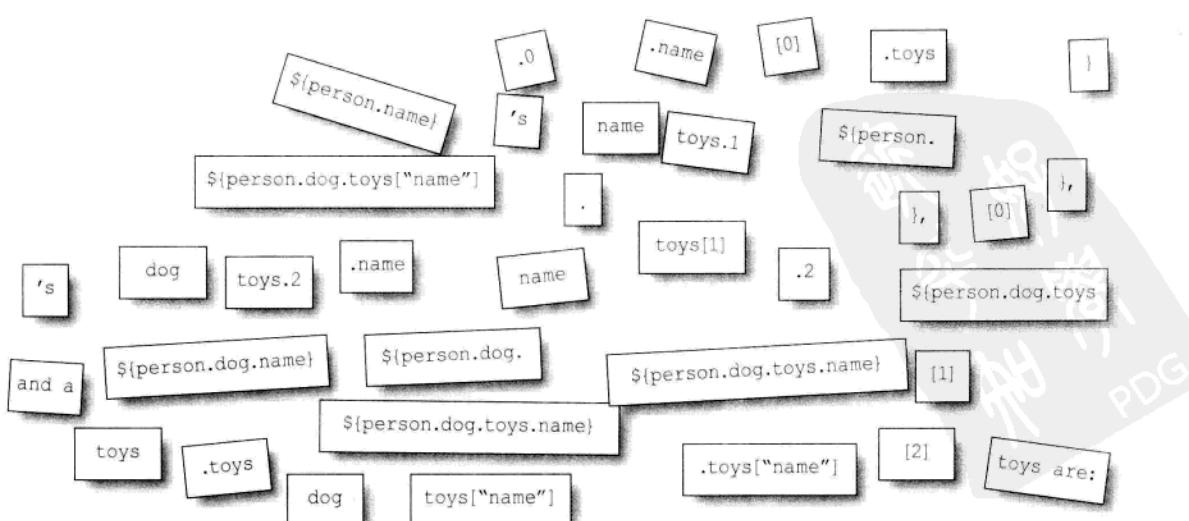
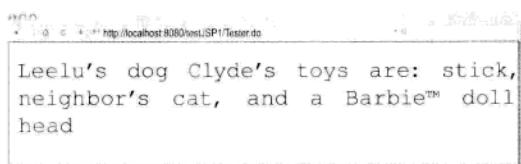
## Servlet代码

```

foo.Person p = new foo.Person();
p.setName("Leelu");
foo.Dog d = new foo.Dog();
d.setName("Clyde");
foo.Toy t1 = new foo.Toy();
t1.setName("stick");
foo.Toy t2 = new foo.Toy();
t2.setName("neighbor's cat");
foo.Toy t3 = new foo.Toy();
t3.setName("Barbie™ doll head");
d.setToys(new foo.Toy[]{t1, t2, t3});
p.setDog(d);
request.setAttribute("person", p);

```

建立能生成以下输出的EL:





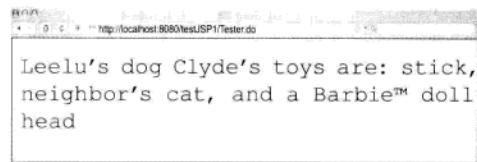
## 代码贴答案

这不是生成这个输出的唯一方法，但是如果只能使用这组代码贴的话，就只有这么一个途径。附加练习：编写一个稍有不同的EL表达式（不考虑这里的代码贴），但要求打印同样的结果。

### Servlet代码

```
foo.Person p = new foo.Employee();
p.setName("Leelu");
foo.Dog d = new foo.Dog();
d.setName("Clyde");
foo.Toy t1 = new foo.Toy();
t1.setName("stick");
foo.Toy t2 = new foo.Toy();
t2.setName("neighbor's cat");
foo.Toy t3 = new foo.Toy();
t3.setName("Barbie™ doll head");
d.setToys(new foo.Toy[]{t1, t2, t3});
p.setDog(d);
request.setAttribute("person", p);
```

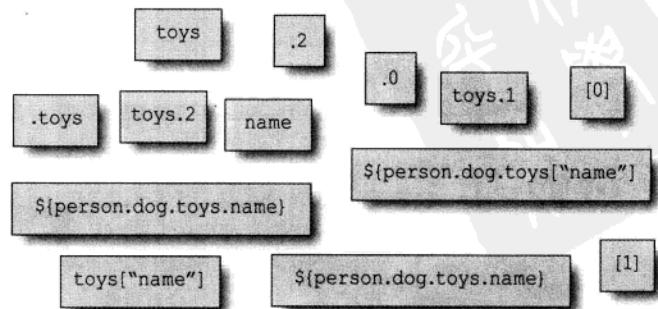
### 建立能生成以下输出的EL:



`${person.name}'s dog ${person.dog.name}'s toys are: ${person.dog.toys[0].name},`

`${person.dog.toys[1].name}, and a ${person.dog.toys[2].name} }`

`${person.name}'s dog ${person.dog.name}'s toys are: ${person.dog.toys[0].name},  
 ${person.dog.toys[1].name}, and a ${person.dog.toys[2].name}`

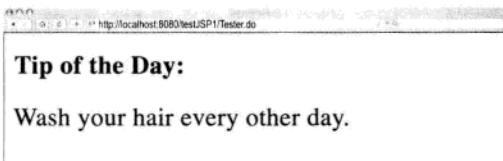


## 5分钟揭秘



### 内容丢失案件

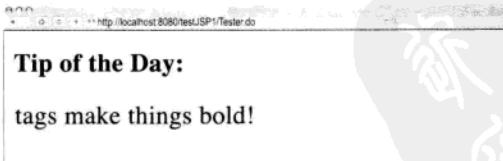
Documents-R-Us创建了一个文档管理系统，主要用来创建桌面应用的教程。这个应用中有一个功能允许内容开发人员创建“每日提示”内容块，每日提示内容存储在请求作用域属性currentTip中。例如，如果今天的提示是“Wash your hair every other day”，屏幕上就会显示类似下面的提示框：



这个提示框的JSP代码如下：

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br /> <br />
    ${pageContent.currentTip}
</div>
```

一个新客户想使用这个系统创建一个教程，但是看起来无法正确地显示提示。例如，提示“**<b></b> tags make things bold!**”会显示如下：



“怎么回事？”这家客户的首席JSP开发人员Tawny很不满。“提示最前面的内容到哪里去了？为什么没有显示bold标记？”她立即向Documents-R-Us发出一个bug报告。

你是怎么考虑的？bold标记发送到输出流了吗？为什么没有显示出来呢？

## EL会显示原始文本，包括HTML

看看实际生成的HTML，你就能明白奥秘  
何在了……

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  ${pageContent.currentTip}
</div>
```

这样看来，提示的“**<b></b>**”部分确实发送到了输出流，但是web浏览器只是把它呈现为原始HTML——将页面上的一个空格加粗。

所以，用户当然看不到屏幕上显示“**<b></b>**”标记。

生成的HTML  
↓

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <b></b><b></b> tags make things bold!
</div>
```

### 对于JSP表达式标记也是如此……

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <%= pageContent.getCurrentTip() %>
</div>
```

不论这计算为什么结果，都会作为标准HTML，所以会呈现为HTML标记，而不是作为文本显示。

### ……对于jsp:getProperty标准动作也不例外

这里是一样的。*<i></i>*和*<b></b>*类的文本在浏览器中会作为HTML标记呈现，而不是作为纯文本显示。

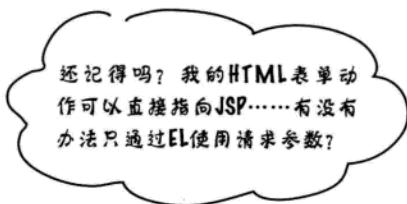
```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <jsp:getProperty name='pageContent' property='currentTip' />
</div>
```



### 开动脑筋

那好，这说明提示串确实已经发送到输出流，但是Documents-R-Us希望把提示中的HTML特殊符号转换为可以正确显示的格式。所以我们希望发出“&lt;”，以便用户在浏览器中看到实际的<字符，另外希望发出“&gt;”来产生>字符。

怎么做到这一点呢？



## EL隐式对象

记住，EL有一些隐式对象。但是这与JSP隐式对象不同（只有一个例外，即pageContext）。下面简单地列出了这些EL隐式对象，后面几页还会更详细地讨论。你会注意到，除了一个隐式对象外（又是pageContext），其他隐式对象都是简单的Map（名/值对）。

### **pageScope**

**requestScope** 作用域属性的Map。

### **sessionScope**

### **applicationScope**

**param** 请求参数的Map。

### **paramValues**

**header** 请求首部的Map。

### **headerValues**

**cookie** 哎呀……这个有些难……它是不是……  
cookie的Map?

**initParam** 上下文初始化参数（不是Servlet初始化参数！）  
的Map。

### **pageContext**

唯一一个不是Map的隐式对象。这个才是真正  
引用，它是pageContext对象的一个实际引用，可  
以把pageContext对象想成是一个bean。请查看  
PageContext获取方法的API。

## EL中的请求参数

很简单。如果你知道对应特定的参数名只有一个参数，就可以用param隐式对象。如果对应一个给定参数名有多个参数值，就要使用paramValues。

### HTML表单：

```
<form action="TestBean.jsp">
    Name: <input type="text" name="name">
    ID#: <input type="text" name="empID">
    First food: <input type="text" name="food">
    Second food: <input type="text" name="food">
    <input type="submit">
</form>
```

“name” 和 “empID” 都只有一个值。但是如果用户点击提交按钮之前填写了两个食物值，“food” 参数就有两个值……

记住，param只是参数名和值的一个Map。点号右边是表单输入域中指定的名。

### JSP中：

```
Request param name is: ${param.name}
```

```
Request param empID is: ${param.empID}
```

```
Request param food is: ${param.food}
```

```
First food request param: ${paramValues.food[0]}
Second food request param: ${paramValues.food[1]}
```

就算是“food”参数可能有多个值，仍然可以使用param隐式对象，但是这样一来就只能得到第一个值。

```
Request param name: ${paramValues.name[0]}
```

在客户的浏览器上（客户填写了表单，并点击submit按钮）

Name: Fluffy	ID#: 423
First food: Sushi	
Second food: Macaroni & Cheese	<input type="submit"/>

### 响应

```
Request param name is: Fluffy
Request param empID is: 423
Request param food is: Sushi
First food request param: Sushi
Second food request param: Macaroni & Cheese
Request param name: Fluffy
```

# 如果想从请求得到更多信息呢？

假如说，你想得到服务器主机信息（由请求的“host”首部提供），该怎么办？

如果查看HttpServletRequest API，可以看到一个getHeader(String)方法。我们知道，如果把“host”传递给getHeader()方法，就会得到：“localhost:8080”（因为Web服务器就在这里）。

## 得到“host”首部

我们知道，可以用脚本来完成

```
Host is: <%= request.getHeader("host") %>
```

但是利用EL，可以用header隐式对象做到

```
Host is: ${header["host"]}  
Host is: ${header.host}
```

header隐式对象保存了所有首部的一个Map。使用任何一个访问操作符，传入首部名，就会打印该首部的值。（注意：对于有多值的首部，另外还有一个headerValues隐式对象。它的工作与paramValues很相似。）

## 得到HTTP请求方法

噢，有点难了……HttpServletRequest API中有一个方法getMethod()，它可以返回请求方法GET、POST等。但是使用EL的话怎么得到请求方法呢？

我们知道，使用脚本可以完成：

```
Method is: <%= request.getMethod() %>
```

但使用EL的话，这是不行的：

```
Method is: ${request.method}
```

不行！绝对不行！没有request隐式对象！

不行！绝对不行！确实有一个requestScope，但是它不是请求对象本身。

这样也不行：

```
Method is: ${requestScope.method}
```

能看出来该怎么办吗？

提示：看看其他隐式对象。

## requestScope不是请求对象

隐式的requestScope只是请求作用域属性的一个Map，而不是request对象本身！你想要的（HTTP方法）是request对象的一个性质，并不是请求作用域上的一个属性。换句话说，你希望在request对象上调用一个获取方法来得到你想要的东西（如果把request对象看作是一个bean）。

但是并没有一个request隐式对象，只有requestScope！该怎么办呢？

你需要点别的……

使用requestScope会得到请求属性，而不是request性质。要得到request性质，需要通过pageCon-

使用pageContext来得到其他的一切……

Method is: \${pageContext.request.method}

pageContext有一个request性质

request有一个method性质



不要把Map作用域对象与属性绑定的对象混淆了。

如果这样想就很简单了，例如，applicationScope是ServletContext的一个引用，因为应用作用域属性就绑定到这个对象。但是，就像requestScope和请求对象一样，应用作用域属性的Map只是属性的一个映射（Map），仅此而已。不能把它当成一个Servlet上下文，所以不要指望能从applicationScope隐式对象得到ServletContext性质！

既然EL会在所有4个作用域里查找，为什么还要使用某个特定的作用域隐式对象呢？我能想到的原因只有命名冲突，不过，我想知道是不是还有其他原因……



不对！这当然是合法的，但是容器会把“foo”当成是某个作用域中的属性，而且它有一个“person”性质。但是容器永远也找不到这样一个“foo”属性。

太完美了！使用requestScope对象，我们就有办法把属性名放在引号里了。

## 作用域隐式对象能救你

如果你要做的只是打印一个person的名字，你并不关心这个person在哪个作用域里（或者，就算是你确实关心，但你知道在所有4个作用域中只有一个person），那么只需使用：

```
 ${person.name}
```

或者，如果你担心可能存在命名冲突，可以明确地指出想要哪个作用域里的person：

```
 ${requestScope.person.name}
```

但是，在属性前面加上隐式作用域还有没有其他原因呢？不光是控制……作用域？

想想下面这种情况：如果一个没有用引号起来的名字放在中括号（[]）中，这说明这个名字必须遵循Java命名规则，是这样吗？在这里我们很幸运，因为person正好是一个合法的Java变量名。之所以说它合法，原因是有人在某个地方曾说过：

```
 request.setAttribute("person", p);
```

## 但是属性名是一个String！

String并不遵循Java变量命名规则！

这说明，可能有人这样说：

```
 request.setAttribute("foo.person", p);
```

这么一来，你就有麻烦了，因为下面这样写是行不通的：

```
 ${foo.person.name}
```

不过多亏有作用域对象，使用作用域对象的话，你就能使用[]操作符，这样就算是不遵循Java命名规则的String名也能顺利地使用。

```
 ${requestScope["foo.person"].name}
```

## 得到Cookie和初始化参数

除了cookie和初始化参数的相应隐式对象外，其他隐式对象我们都已经了解了，所以现在来介绍这个内容。不错，所有隐式对象都可能在考试中出现。

### 打印“**userName**” Cookie的值

我们知道，这可以使用脚本来完成

```
<% Cookie[] cookies = request.getCookies();  
  
for (int i = 0; i < cookies.length; i++) {  
    if ((cookies[i].getName()).equals("userName")) {  
        out.println(cookies[i].getValue());  
    }  
} %>
```

这有点麻烦，因为request对象没有  
getCookie(cookieName)方法！我们必须  
得到整个Cookie数组，然后自行对它  
迭代处理。

但是使用EL的话，可以用Cookie隐式对象做到：

```
${cookie.userName.value}
```

容易多了。只需提供名字，就会从Cookie名/值对的  
Map返回适当的值。

### 打印一个上下文初始化参数的值

必须在DD中配置这个参数

要记住，上下文（应用范围）参数就要这样配置。这与servlet初始化参数不同。

```
<context-param>  
    <param-name>mainEmail</param-name>  
    <param-value>likewecare@wickedlysmart.com</param-value>  
</context-param>
```

我们知道，这可以使用脚本完成

```
email is: <%= application.getInitParameter("mainEmail") %>
```

使用EL的话，就容易多了：

```
email is: ${initParam.mainEmail}
```



EL initParam并不对应使用  
<init-param>配置的参数！

这里有一点很容易混淆：servlet初始化参数是用<init-param>配置的，而上下文参数使用<context-param>配置，但是EL隐式对象“initParam”对应的却是上下文参数！要是制订规范的人咨询我们的意见，我们可能会建议这样来命名这个变量：“contextParam”……但是他们没有问过我们。

EL是很不错……但是有时我要的是功能，而不只是属性或性质值。如果有办法让EL表达式调用一个Java方法来返回一个值……那就多好。



## 她不知道EL函数

如果还需要另外的帮助，比如说，想调用一个Java方法，但是你不想写脚本，就可以使用EL函数。这是一种很容易的方法，你可以编写简单的EL表达式，由它调用你写的一个普通Java类的静态方法。这个方法返回的结果会用在表达式中。要完成这些配置确实需要多做一些工作，但是有了函数，你就有了更多的……功能。

## 假设你希望有一个掷骰子的JSP

你认为有一个基于Web的掷骰子服务可能很酷。这样的话，不需要满地找骰子，看看桌子底下、沙发垫里有没有，用户只需访问你的Web页面，点击一个虚拟骰子，就大功告成了！它会自动开始掷骰子！（当然，你可能不知道，在Google上搜索一下，可能会找到有4,420家网站在做这个事情。）

### ① 编写有一个公共静态方法的Java类。

这只是一个普通的Java类。这个方法必须是公共的，而且是一个静态方法。它可以有参数，返回类型一般都不是void（但并没有严格要求）。毕竟，这个方法存在的意义就是从JSP调用并得到一些东西，可以把返回的结果用作为表达式的一部分，或者是把结果打印出来。

把这个类文件放在WEB-INF/classes目录结构中（对应适当的包目录结构，这与其他类是一样的）。

### ② 编写一个标记库描述文件（TLD）。

对于一个EL函数，TLD提供了定义函数的Java类与调用函数的JSP之间的一个映射。这样一来，函数名和具体的方法名可以是不同的。例如，你使用的类可以有一个着实很傻气的方法名，但你想为使用EL的页面设计人员提供一个更明显更直观的名字。没问题，TLD会这样说，“这是一个Java类，这是函数的方法签名（包括返回类型），而这是我们将在EL表达式里用的名字”。换句话说，EL中使用的名字不必与具体的方法名相同。这个映射就在TLD里定义。

把TLD文件放在WEB-INF目录中。指定扩展名为.tld。（TLD还可以放在其他位置；后面两章中将讨论这个问题。）

### ③ 在JSP中放一个taglib指令。

taglib指令告诉容器，“我想使用这个TLD，另外，在JSP中，使用这个TLD中的一个函数时，我想用这个名字作为前缀……”。换句话说，这样就能定义命名空间。你可以使用多个TLD中的函数，就算是函数同名也不碍事。taglib指令就好像给你的函数提供了一个完全限定名。调用函数时要同时指定函数名和TLD前缀。前缀可以是你喜欢的任何名字。

### ④ 使用EL调用函数。

这一部分很容易。只需要按\${prefix:name()}的形式从表达式调用函数就行了。

## 函数类、TLD和JSP

函数方法必须是公共、静态的方法。

### 有函数的类

```
package foo;

public class DiceRoller {
    public static int rollDice() {
        return (int) ((Math.random() * 6) + 1);
    }
}
```

### 标记库描述文件(TLD)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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-
  jsp.taglibrary_2_0.xsd" version="2.0">
<tlib-version>1.2</tlib-version>
<uri>DiceFunctions</uri>
<function>
    <name>rollIt</name>
    <function-class>foo.DiceRoller</function-class>
    <function-signature>
        int rollDice()
    </function-signature>
</function>
</taglib>
```

不要担心<taglib...>标记里的东西。

后面两章还会更多地介绍TLD。

taglib指令中的uri告诉容器TLD的名字（不一定是文件名！），容器需要知道TLD名，这样它才能知道JSP调用EL函数时它该调用哪个方法。

### JSP

```
<%@ taglib prefix="mine" uri="DiceFunctions"%>
<html><body>
    ${mine:rollIt()}
</body></html>
```

函数名rollIt()来自TLD中的<name>，而不是来自具体的Java类。

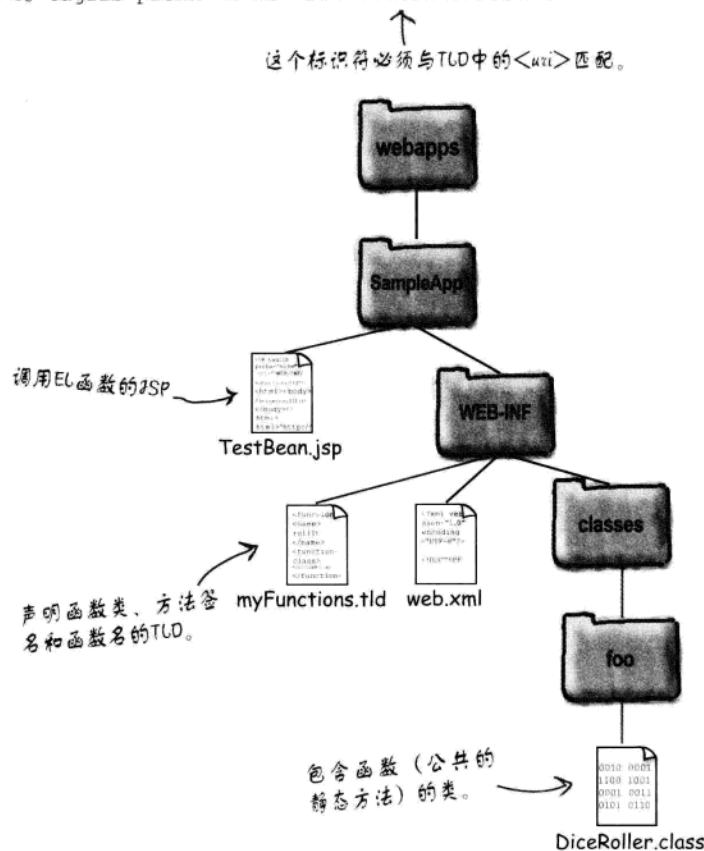
前缀“mine”只是在这个页面中使用的一个“昵称”，这样就能区别不同的TLD（万一你确实有多个TLD）。

## 部署有静态函数的应用

这里只有一个新东西，“myFunctions.tld”文件。它必须放在WEB-INF目录或者某个子目录中（除非部署在一个JAR文件中，不过这一点本书后面再做介绍）。在此，因为这个应用非常简单，所以我们把DD（web.xml）和TLD（myFunctions.tld）都直接放在WEB-INF目录下，不过你也可以把它们组织到子目录中。

关键是包含静态函数的类必须对应用可用，所以……现在你应该知道，把它放在WEB-INF/classes目录下就可以，另外要记住，在JSP里的taglib指令中，我们指定了一个URI，它与TLD中声明的URI匹配。对现在来说，可以把URI想得很简单，可以是你为TLD取的一个名字。它就是一个名字。在后面介绍如何使用定制标记的一章中，我们将更详细地介绍TLD和URI的细节。

```
<%@ taglib prefix="mine" uri="DiceFunctions"%>
```



包含函数（公共的静态方法）的类应当像servlet、bean和监听者类一样，必须对Web应用可用。这说明，要放在WEB-INF/classes中的某个位置……

TLD文件可以放在WEB-INF下的某个位置上，并确保JSP中taglib指令包含的uri属性与TLD中的<uri>元素匹配。

# there are no Dumb Questions

**问：** 常规的scriptlet表达式必须返回点东西。如果有这样一个表达式：`<%=foo.getFoo()%>`，`getFoo()`的返回类型就不能是`void`（至少应该是你先前声明过的某个东西）。所以，我想知道，它是不是与EL函数完全一样？

**答：** 不是！ 它和EL函数可不一样。尽管这很出乎我们的意料。这样来想想，如果你在调用一个EL函数，它不返回任何东西，调用它的目的只是想得到它的副作用！不过，之所以要使用EL，部分原因就是想减少JSP中的逻辑（JSP只作为视图！）。如果调用EL函数只是想得到它的副作用，这可不是一个好的想法。

**问：** 容器怎么发现TLD？URI并没有与TLD的路径名或文件名匹配，容器居然还能发现TLD，这难道不是奇迹吗？

**答：** 这是一个很好的问题，我们希望每个人都能想到这一点。不错，你说得对，我们确实没有告诉容器实际的TLD文件在哪里。部署时，容器会搜索WEB-INF和它的子目录（或者在JAR文件中搜索WEB-INF/lib），查找所有的.tld文件。如果找到，就读取URI，并创建一个映射，指出：“有这个URI的TLD就是这个位置上的这个文件……”下一章我们还会再做说。

**问：** EL函数可以有参数吗？

**答：** 当然可以有。只要记住一点，在JSP中要为每个参数指定完全限定类名（除非是一个基本类型）。例如，如果函数取一个map参数，则应当是：

```
function-signature>
    int rollDice(java.util.Map)
function-signature>
```

以这样来调用：

```
line:rollDice(aMapAttribute)}
```



方法名和函数名不一样！

当心！

要记住类、TLD和JSP之间的关系。最重要的是，记住方法名不一定与函数名匹配。EL中调用函数时，所用的函数名必须与TLD中`<function>`声明的`<name>`元素匹配。`<function-signature>`元素的作用就是告诉容器：JSP使用`<name>`时具体要调用哪个函数。

除了类声明本身之外，类名只出现在`<function-class>`元素中。

哦，对了……你注意到了吗？`<function>`标记中所有元素几乎都有`<function>`这个词，只有`<name>`标记除外！所以，别写成这样，这就错了：

```
<function>          ↘不对!
    <function-name>rollIt</function-name>      ↗
                                                <function-class>

```

```
        foo.DiceRoller</function-class>
    <function-signature>
        int rollDice()
    </function-signature>
```

```
</function>
```

对应函数名的正确标记应当是`<name>`！

```
<function>          ↘对了!
    <name>rollIt</name>      ↗
                                                <function-class>

```

```
        foo.DiceRoller</function-class>
    <function-signature>
        int rollDice()
    </function-signature>
</function>
```

## 还有另外一些EL操作符……

你可能不会（也不应该）在EL中完成计算和实现逻辑。要记住，JSP是视图，视图的任务就是呈现响应，而不是做重大决策，也不是做大规模处理。如果你真的需要实现功能，这一般是控制器和模型的任务。如果功能不太多，则可以利用定制标记（包括JSTL标记）和EL函数。

不过……对于一些小问题，有时一个小小的算术运算或者一个简单的布尔测试可能更方便。考虑到这一点，下面来看一些最有用的EL算术、关系和逻辑操作符。

### 算术操作符 (5)

加法: +

减法: -

乘法: \*

除法: / 和 div

取模: % 和 mod

顺便说一句……在EL中是可以除0的。  
你将得到INFINITY，而不是错误。

但是不能对0使用取模操作符，否则会  
得到一个异常。

### 逻辑操作符 (3)

与 (AND) : && 和 and

或 (OR) : || 和 or

非 (NOT) : ! 和 not

### 关系操作符 (6)

等于: == 和 eq

不等于: != 和 ne

小于: < 和 lt

大于: > 和 gt

小于等于: <= 和 le

大于等于: >= 和 ge

不要使用EL保留字作为标识符！

当心！

在这一页上你已经看到其中11个保留字了，它们是在关系、逻辑和一些算术操作符的相应“文字表示”。不过，除此以外还有另外一些：

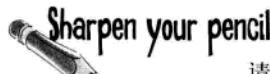
true 这是一个布尔直接量

false 这是另外一个布尔直接量

null 它就表示……null

instanceof (这是为将来预留的保留字)

empty 这个操作符可以查看是否为null或为空。  
(例如，当A为null或为空时，\${empty A})就返回true (本章稍后会介绍一个具体的例子)。



请看以下servlet代码，得出后面的各个EL表达式分别打印什么。有些地方必须猜，因为我们还没有讲到有关的全部规则。这个练习会帮助你了解EL会有怎样的表现。提示：EL很灵活，而且很宽容。另一个提示：实际上这一页最下面反着印有这9个答案，但它们没有按顺序放。不过，如果你真的需要帮助，至少可以在这9个答案基础上，使用消去法看看每个答案分别对应哪一个EL表达式。

### 给定以下servlet代码：

```
String num = "2";
request.setAttribute("num", num);
Integer i = new Integer(3);
request.setAttribute("integer", i);
java.util.ArrayList list = new java.util.ArrayList();
list.add("true");
list.add("false");
list.add("2");
list.add("10");
request.setAttribute("list", list);  
假设Dog bean类可用。
```

### 下面的EL表达式分别打印什么？

\_\_\_\_\_ \${num > 3}

\_\_\_\_\_ \${integer le 12}

\_\_\_\_\_ \${requestScope["integer"] ne 4 and 6 le num || false}

\_\_\_\_\_ \${list[0] || list["1"] and true}

\_\_\_\_\_ \${num > integer}

\_\_\_\_\_ \${num == integer-1}

\_\_\_\_\_ <jsp:useBean class="foo.Dog" id="myDog" >
 <jsp:setProperty name="myDog" property="name" value="\${list[1]}" />
</jsp:useBean>

\_\_\_\_\_ \${myDog.name and true}

\_\_\_\_\_ \${42 div 0}



false false false false true true  
true infinity



给定以下servlet代码：

```
String num = "2";
request.setAttribute("num", num);
Integer i = new Integer(3);
request.setAttribute("integer", i);
java.util.ArrayList list = new java.util.ArrayList();
list.add("true");
list.add("false");
list.add("2");
list.add("10");
request.setAttribute("list", list);
```

下面的EL表达式分别打印什么？

<i>false</i>	<code> \${num &gt; 3}</code>	找到“num”属性，将其值“2”强 制转换为一个int。
<i>true</i>	<code> \${integer le 12}</code>	更棒！Integer值会转换为其基本类型值。 再做比较。
<i>false</i>	<code> \${requestScope["integer"] ne 4 and 6 le num    false}</code>	
<i>true</i>	<code> \${list[0]    list["1"] and true}</code>	
<i>false</i>	<code> \${num &gt; integer}</code>	注意别用 == (应该用==)。
<i>true</i>	<code> \${num == integer-1}</code>	在EL中没有==。

看看如果不使用括号，  
能不能得出优先次序  
的相关规则。这很明  
显（从左到右），要  
参加考试，你就必须  
掌握优先级。

```
<jsp:useBean class="foo.Dog" id="myDog" >
    <jsp:setProperty name="myDog" property="name" value="${list[1]}" />
</jsp:useBean>

false       ${myDog.name and true}
Infinity    ${42 div 0}
```

对，可以在标记  
内部使用EL！

# EL能妥善地处理null值

EL的开发人员提出了一个重要的设计决策，要妥善地处理null值而不抛出异常。为什么呢？因为他们认为“给用户显示一个不完全的页面比显示一个错误页面更好一些。”

假设没有一个名为“foo”的属性，但确实有一个名为“bar”的属性，而且这个“bar”没有名为“foo”的性质或键。

## EL

### 打印出

```
${foo}  
${foo[bar]}  
${bar[foo]}  
${foo.bar}
```

\${7 + foo}	7	在算术表达式里。 EL把未知(unknown)变量看作是“0”。
\${7 / foo}	Infinity	
\${7 - foo}	7	
\${7 % foo}	抛出异常	

\${7 < foo}	false	
\${7 == foo}	false	在逻辑表达式中。EL把未知(unknown)变量看作是“false”。
\${foo == foo}	true	
\${7 != foo}	true	
\${true and foo}	false	
\${true or foo}	true	
\${not foo}	true	

**EL能友好地处理null。它能处理unknown或null值，即使找不到表达式中指定名的属性/性质/键，也会显示页面。**

**在算术表达式中，EL把null值看作是“0”。**

**在逻辑表达式中，EL把null值看作是“false”。**

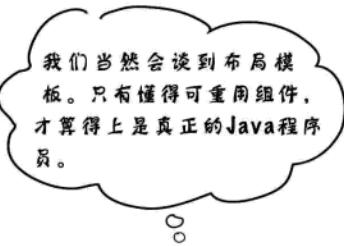
# JSP表达式语言(EL)复习

## 要点

- EL表达式总是用大括号括起，而且前面有一个美元符(\$)前缀：\${expression}。
- 表达式中第一个命名变量要么是一个隐式对象，要么是某个作用域（页面作用域、请求作用域、会话作用域或应用作用域）中的一个属性。
- 点号操作符允许你使用一个Map键或一个bean性质名来访问值，例如，使用\${foo.bar}可以得到bar的值，在此bar是Map foo的Map键名，或者bar是bean foo的一个性质。放在点号操作符右边的东西必须遵循Java的标识符命名规则！（换句话说，必须以一个字母、下划线或美元符开头，第一个字符后面可以有数字，但是不能有其他字符。）
- 点号右边只能放合法的Java标识符。例如，\${foo.1}就不允许。
- [] 操作符比点号功能更强大，因为利用[]可以访问数组和List，可以把包含命名变量的表达式放在中括号里，而且可以做任意层次的嵌套，只要你受得了。
- 例如，如果musicList是一个ArrayList，可以用\${musicList[0]}或\${musicList["0"]}来访问列表中的第一个值。EL并不关心列表索引加不加引号。
- 如果中括号里的内容没有用引号引起来，容器就会进行计算。如果确实放在引号里，而且不是一个数组或List的索引，容器就会把它看作是性质或键的直接量名。
- 除了一个EL隐式对象（PageContext）外，其他EL隐式对象都是Map。从这些Map隐式对象可以得到任意4个作用域中的属性、请求参数值、首部值、cookie值和上下文初始化参数。非映射的隐式对象pageContext，它是PageContext对象的一个引用。
- 不要把隐式EL作用域对象（属性的Map）与属性所绑定的对象混为一谈。换句话说，不要把requestScope隐式对象与具体的JSP隐式对象request混淆。访问request对象只有一条路，这就是通过pageContext隐式对象来访问。（不过，想从请求得到的一些东西通过其他EL隐式对象也可以得到，包括param/paramValues、header/headerValues和cookie。）
- EL函数允许你调用一个普通Java类中的公共静态方法。函数名不一定与具体的方法名匹配！例如，\${foo:rollIt()}并不意味着包含函数的类中肯定有一个名为rollIt()的方法。
- 使用一个TLD（标记库描述文件）将函数名（例如 rollIt()）映射到一个具体的静态方法。使用<function>元素声明一个函数，包括函数的<name>(rollIt())，完全限定类<function-class>，以及<function-signature>，其中包括返回类型以及方法名和参数表。
- 要在JSP中使用函数，必须使用taglib指令声明一个命名空间。在taglib指令中放一个prefix属性，告诉容器你要调用的函数在哪个TLD里能找到。例如：

```
<%@ taglib prefix="mine"
           uri="/WEB-INF/foo.tld"%>
```





## 可重用的模板部件

你的网站上每个页面都有页眉。它们都一样。另外，每个页面上的页脚也是一样的。如果在Web应用的每个JSP上反复地编写同样的页眉和页脚，这不是很傻吗？

如果你用Java程序员的思路去考虑（你当然会这样，对不对），就会知道，这样做很不符合面向对象的宗旨。单单想一想有那么多重复的代码，这就够让人头疼的了。更别说网站设计人员可能会修改页眉或页脚！哪怕只是一个很小的修改。

你必须把这种修改传播到每一个页面中。

别着急。对于这个问题，JSP中有一个相应的处理机制，这就是包含（include）。你还是照平常的做法编写JSP，不过不要把可重用的部分明确地放在JSP中，而只是告诉容器要把其他文件包含到当前页面上，所包含的内容放在你选择的位置上。形式如下：

```
<html><body>  
<!-- 在这里插入页眉文件 --&gt;<br/>Welcome to our site...  
blah blah blah more stuff here...  
<!-- 在这里插入页脚文件 --&gt;<br/></body></html>
```

在这一节中，我们将看到两种不同的包含机制：`include`指令和`<jsp:include/>`标准动作。

## include指令

include指令告诉容器：复制所包含文件中的所有内容，再把它粘贴到这个文件中，而且就放在这里…

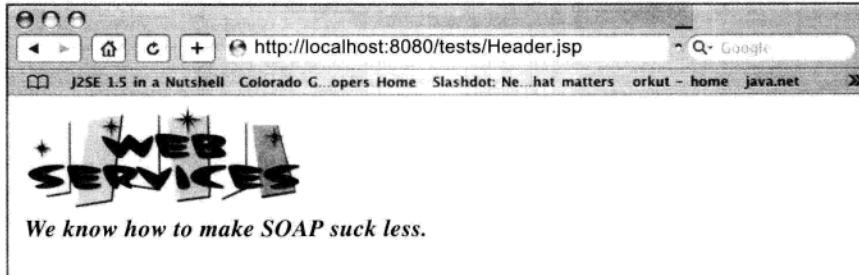
### 标准页眉文件(“Header.jsp”)

我们希望web应用中的每个页面上都显示  
这个HTML内容。

```
<html><body>

 <br>
<em><strong>We know how to make SOAP suck less.</strong></em> <br>

</body></html>
```



### Web应用中的一个JSP(“Contact.jsp”)

```
<html><body>

<%@ include file="Header.jsp"%>

<br>
<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

这是说“把完整的Header.jsp文件插入到当前页面的这个位置，然后是这个JSP的余下部分……”



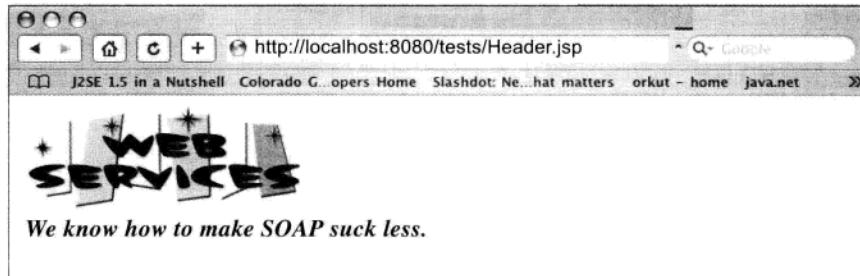
## <jsp:include>标准动作

<jsp:include>标准动作看上去和include指令是一样的。

标准页眉文件(“Header.jsp”)

```
<html><body>  
 <br>  
<em><strong>We know how to make SOAP suck less.</strong></em> <br>  
</body></html>
```

我们希望每一页上都显示这个页面。



Web应用中的一个JSP (“Contact.jsp”)

```
<html><body>  
<jsp:include page="Header.jsp" />  
  
<br>  
<em>We can help.</em> <br><br>  
Contact us at: ${initParam.mainEmail}  
</body></html>
```

这是说“把Header.jsp文件的响应插入到当前页面的这个位置上，然后是这个JSP的余下部分……”



# 内部原理并不同……

<jsp:include>标准动作和include指令看上去一样，而且通常有相同的结果，但是你再看看生成的servlet，就能发现它们并不相同。以下代码直接取自Tomcat生成的servlet代码中的 \_jspService()方法……

为页眉文件生成的servlet代码

```
out.write("\r<html>\r<body>\r<img src=\"images/Web-Services.jpg\" >
<br>\r<em><strong>We know how to make SOAP suck less.</strong></em> <br>\r\r
</body>\r</html>\r");
很简单……只是完成输出。
```

为使用include指令的JSP生成的servlet

```
out.write("<html><body>\r");
粗体显示的部分与为Header.jsp页面生成
的代码完全一样。
out.write("\r<html>\r<body>\r<img src=\"images/Web-Services.jpg\" >
<br>\r<em><strong>We know how to make SOAP suck less.</strong></em> <br>\r\r
</body>\r</html>\r");
out.write("\r<br>\r\r<em>We can help.</em> <br><br>\r\rContact us at: ");
out.write((java.lang.String) org.apache.jasper.runtime.PageContextImpl.
proprietaryEvaluate("${initParam.mainEmail}", java.lang.String.class,
(PageContext) _jspx_page_context, null, false));
out.write("\r\r</body></html>");
include指令只是取得“Header.jsp”文件的内容,
并在转换之前放在“Contact.jsp”页面里!
```

为使用<jsp:include>标准动作的JSP生成的servlet

```
out.write("<html><body>\r");
这是不一样的！原来的Header.jsp 文件并不放在生
成的servlet里！这里只是一个运行时调用……
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response,
"Header.jsp", out, false);

out.write("\r<br>\r\r<em>We can help.</em> <br><br>\r\rContact us at: ");
out.write((java.lang.String) org.apache.jasper.runtime.PageContextImpl.
proprietaryEvaluate("${initParam.mainEmail}", java.lang.String.class,
(PageContext) _jspx_page_context, null, false));

out.write("\r\r</body></html>");
```

## include指令在转换时发生

## <jsp:include>标准动作在运行时发生

如果使用include指令，这与你打开JSP页面，并粘贴上“Header.jsp”的内容没有两样。换句话说，这样就好像你把页眉文件的代码重复放在其他JSP中一样。只不过，容器会在转换时为你完成这个工作，所以你不必亲自到处复制代码。你的任务只是编写页面，其中包括一个include指令，其他的任务则由容器负责，它要在转换之前把页眉代码复制到每个JSP中，再编译为生成的servlet。

<jsp:include>则完全不同。它不是从“Header.jsp”复制源代码，include标准动作会在运行时插入“Header.jsp”的响应。<jsp:include>的关键是，容器要根据页面（page）属性创建一个RequestDispatcher，并应用 include()方法。所分派/包含的JSP针对同样的请求和响应回对象执行，而且在同一个线程中运行。

include指令在转换时插入“Header.jsp”的源代码。  
而<jsp:include>标准动作在运行时插入“Header.jsp”的响应。

**问：**那么，为什么不一直使用<jsp:include>呢？这样的话，就总能保证有最新的内容。

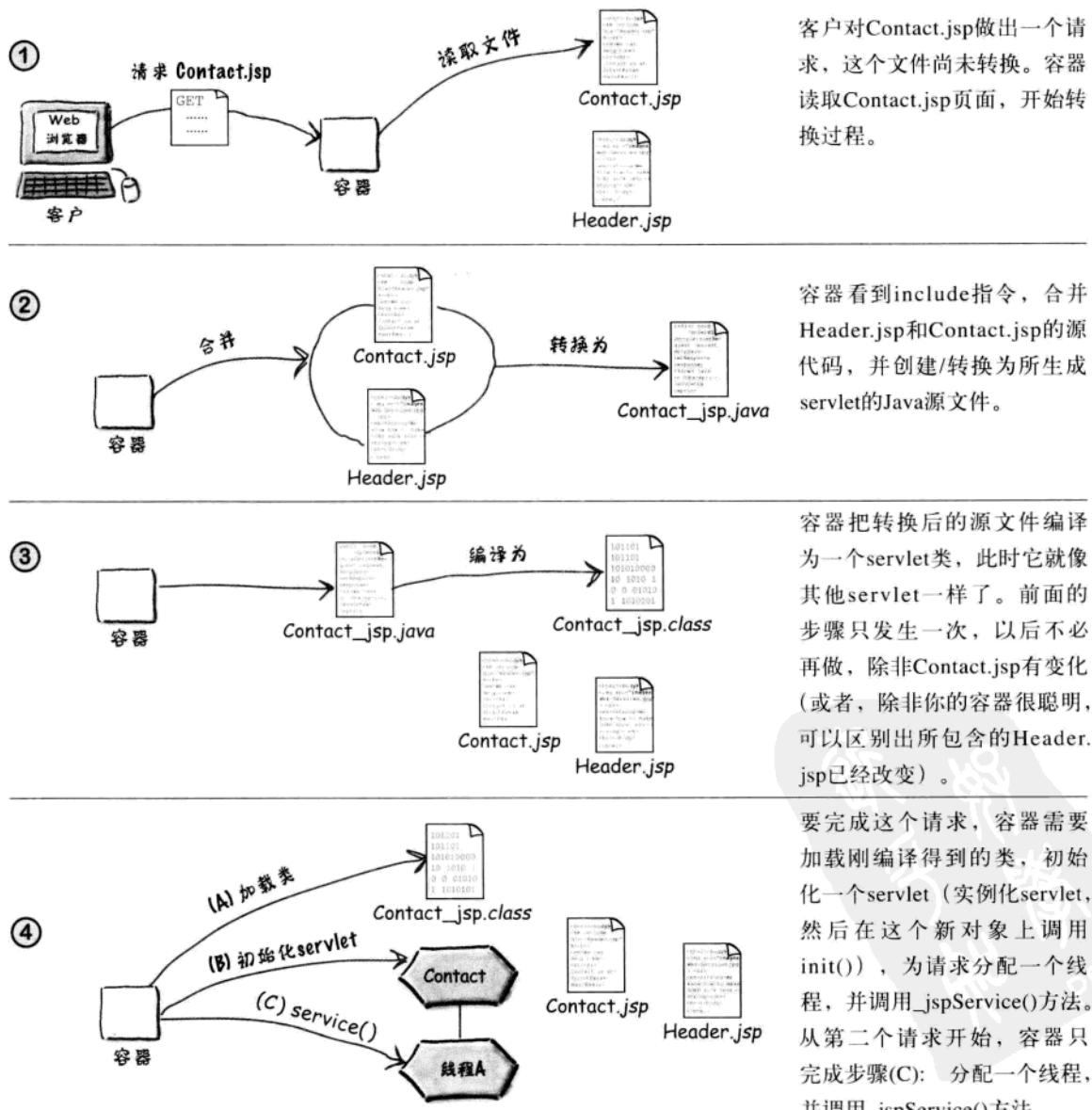
**答：**想想看。每个<jsp:include>都会带来额外的一点性能开销。与此不同，如果使用指令，这种影响只出现一次，也就是转换包含页面（即“外围”页面）的时候才会对性能有影响。所以，如果你能肯定，一旦进入生产阶段所包含的文件就不再改变，那就应该使用指令。当然，使用指令也有一点不太好，生成的Servlet类会大一些。

**问：**我在Tomcat上试过，我建了一个静态的HTML文件，然后用指令包含这个文件，再修改HTML文件、没有重新部署，也没有做其他处理，JSP的输出居然反映出了HTML文件修改前后的差别！如果是这样，为什么还要使用<jsp:include>呢？

**答：**哈……你有一个好容器（比如Tomcat 5）。不错，对于大多数新的容器，所包含的文件发生变化时都有办法检测到，这些容器会重新转换包含文件（“外围”文件），一切都很棒。问题是，规范并不保证这一点！所以，如果你写的代码全依赖于此，你的应用就不一定能移植到其他容器上。

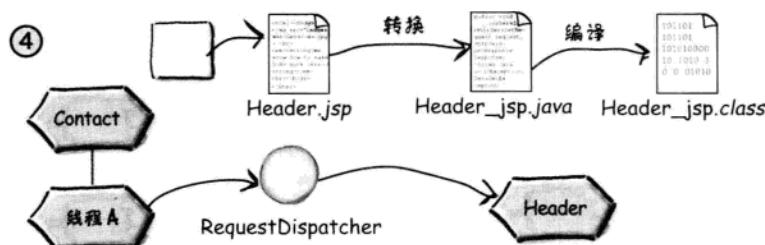
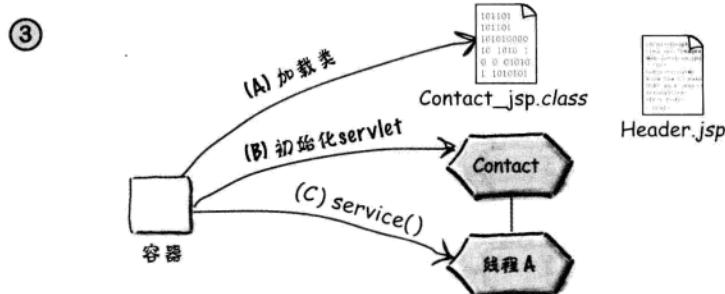
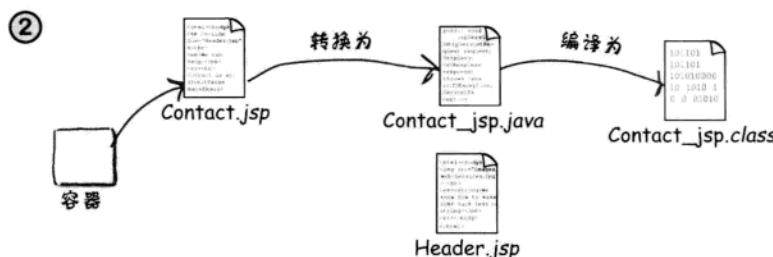
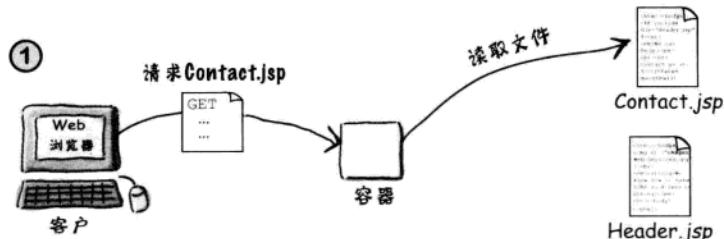
## 对应第一个请求的include指令

利用include指令，容器要做很多工作，不过这些工作只是针对第一个请求才需要做。从第二个请求开始，就再没有额外的运行时开销了。



## 对应第一个请求的<jsp:include>标准动作

如果使用include标准动作，转换时没有多少工作，对应每个请求倒有更多的事情要做，特别当所包含的文件是一个JSP时，每次请求都有一些运行时开销。



Contact servlet调用一个方法来完成动态包含，针对不同的开发商这里可能会有不同的情况！我们只关心一点，由Header servlet生成的响应与Contact servlet生成的响应确实合并（并放在适当的位置上）。（有些步骤在这里没有显示：Header.jsp要在某一点上转换和编译，然后加载和初始化生成的servlet类。）



include指令和<jsp:include/>的属性名是不一样的。

要记住!看看这两种包含机制的属性……能发现有什么不同吗?

```
<%@ include file="Header.jsp"%>
<jsp:include page="Header.jsp" />
```

对。指令的属性是file, 而标准动作的属性是page!为了帮助你记住, 可以这样来想, include指令 <% include file="foo.jsp">只在转换时(像所有指令一样)使用。转换的时候, 容器只关心文件,.jsp到.java, .java到.class。但<jsp:include page="foo.jsp">标准动作不同, 与所有标准动作一样, 它在请求时执行, 此时容器只关心要执行的页面。

**问:** 被包含的JSP可以有自己的动态内容吗? 在你的例子里, Header.jsp也可以是一个静态的Header.html页面。

**答:** 既然是一个JSP, 它当然可以是动态的(不过, 你说得对, 在我们的例子中, 可以把页眉做成一个静态HTML页面, 它与JSP页面的处理完全相同)。但是, 这样会有几个限制: 被包含的页面不能修改响应状态码或设置头部(这说明, 它不能调用addCookies()之类的方法)。如果被包含的JSP尝试去做它不能做的事情, 你不会得到错误, 只是达不到预想的目标。

**问:** 但是如果所包含的东西是动态的, 而且你使用了静态的include指令, 这是不是意味着动态的内容只会计算一次?

**答:** 假设你包含了一个JSP, 其中有一个EL表达式调用了rollIt函数, 这个函数会生成一个随机数。要记住, 使用include指令的话, EL表达式只会简单地复制到包含JSP(外围JSP)。所以每次访问页面时, EL表达式都会运行, 并生成一个新的随机数。一定要牢牢记住: 采用include指令, 被包含页面的源代码将成为有include指令的“外围”页面的一部分。



include指令是位置敏感的!

JSP中只有这个指令的位置会带来区别。例如, 如果使用一个page指令, 可以把它放在页面中的任何位置, 不过, 按惯例大多数人都会把page指令放在最前面。

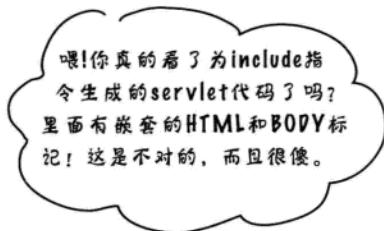
但include指令不同, 它要告诉容器到底把被包含文件的源代码插到哪里!例如, 如果要包含一个页眉和一个页脚, 可能如下所示:

```
<html><body>
<%@ include file="Header.html"%> <br>
<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail} <br>
<%@ include file="Footer.html"%>
</body></html>
```



如果你希望Footer.html的内容出现在JSP的最下面, 这个include指令就必须放在JSP的最后(在结束标记前)。记住, JSP中的所有内容再加上所包含的两个文件会合并成一个大的页面, 它们的顺序很重要!

而且, 当然了, <jsp:include>也是位置敏感的, 不过对于include指令, 这一点更为突出。



## 噢，她说得没错……

想想她说的。我们为页眉建立了一个页面：“Header.jsp”。它本身是一个很正常的JSP，有自己的开始和结束HTML和BODY标记。然后又建立了“Contact.jsp”，这个JSP也有正确的开始和结束标记。我们曾经说过，被包含文件中的所有内容都会粘贴到有include指令的页面中，还记得吧？这意味着全部源代码都会粘贴。

以下代码取自一个生成的servlet，它不一定在所有浏览器上都能工作。不过，我们很幸运，在我们的浏览器上这确实行得通。

```
out.write("<html><body>\r");
out.write("\r<html>\r<body>\r
          ><img src=\"images/Web-Services.jpg\" >
          <br>r<em><strong>We know how to make SOAP
          suck less.</strong></em> <br>\r\r
          ></body>\r</html>\r");
out.write("\r<br>\r\r<em>We can help.</em>
          <br><br>\r\rContact us at: ");
out.write((java.lang.String) org.apache.jasper.runtime.
          PageContextImpl.proprietaryEvaluate("${initParam.
          mainEmail}", java.lang.String.class,
          (PageContext) _jspx_page_context, null, false));
out.write("\r\r</body></html>");
```

**不要把开始和结束HTML和BODY标记放在可重用部件中！**  
设计和编写布局模板部件时（如页眉、导航条等），要假设它们会包含在其他页面中。

别指望我会帮你把冗余的开始和结束标记去掉。



## 正确的做法

把被包含文件中的开始和结束标记去掉。这样一来，被包含文件本身确实不能再生成合法的HTML页面了；它们现在必须包含在另外一个更大的页面中。也就是一个有<html><body>和</body></html>标记的页面。但是，这正是关键，就应当这样设计可重用块，从而能利用较小的部件构成完整的布局，而无需手工地重复编写代码。这些可重用块本身并不是完整的。

### ① Header文件(“Header.jsp”)

```
 <br>
<em><strong>We know how to make SOAP suck less.</strong></em> <br>
```

### ② Contact.jsp

```
<html><body>

<%@ include file="Header.jsp"%> <br>

<em>We can help.</em> <br><br>
Contact us at: ${initParam.mainEmail}

<%@ include file="Footer.html"%>
</body></html>
```

注意，我们从被包含文件中去掉了所有HTML和BODY标记。

### ③ Footer文件(“Footer.html”)

```
<a href="index.html">home page</a>
```

注意：去掉开始和结束标记的做法对两种包含机制（<jsp:include>和include指令）都适用。

① Points to the <img> tag in the Header.jsp code.

② Points to the <em> tag in the Contact.jsp code.

③ Points to the <a href="index.html">home page</a> tag in the Footer.html code.

The browser screenshot shows:

- A logo for "WEB SERVICES".
- The text "We know how to make SOAP suck less."
- The text "We can help".
- An email address: "Contact us at: likewecare@wickedlysmart.com"
- A link labeled "home page".

## 使用<jsp:param>定制包含的内容

OK，这样就有一个页眉，它能以同样的方式显示在每个页面上。但是，如果你想定制页眉的某一部分呢？比如说，你希望页眉上有一个与上下文相关的子标题，它要依页面而定，该怎么做呢？

有两个办法。

**笨办法：**把子标题的信息放在主页面上，作为页眉之后的第一个内容，也就是紧挨着放在页面中包含页眉的include的后面。

**更聪明的办法：**把子标题信息作为新的请求参数传递给所包含的页面！

**为什么这样更酷：**如果子标题信息要作为页眉的一部分，但是这部分可能会变化，你可能还希望模板的页眉能决定子标题在最后页面上如何显示。换句话说，让设计页眉的人来决定如何显示子标题！

完成包含的JSP

```
<html><body>
    <jsp:include page="Header.jsp" >
        <jsp:param name="subTitle" value="We take the sting out of SOAP." />
    </jsp:include>

    <br>
    <em>Web Services Support Group.</em> <br><br>
    Contact us at: ${initParam.mainEmail}
</body></html>
```

看……没有结束反斜线！

<jsp:include>可以有一个体，能增加（或替换）请求参数，供被包含的片段使用。

使用新参数的被包含页眉（“Header.jspf”）

```
 <br>
<em><strong>${param.subTitle}</strong></em> <br>
```

对于被包含的文件来说，用<jsp:param>设置的参数与所有其他请求参数是一样的。  
在此我们使用EL来得到这个参数。

注意：这种使用参数的做法对于include指令没有任何意义（include指令不是动态的），所以只适用于<jsp:include>标准动作。



## <jsp:forward>标准动作

确实可以从一个JSP转发到另一个JSP。或者从一个JSP转发到一个servlet。还可以从一个JSP转发到Web应用中的任何其他资源。

当然，在生产阶段你可能不想这样做，这是因为，如果你在使用MVC，视图就是视图，不要越俎代庖！视图不应该完成控制逻辑。换句话说，查看一个人是否登录，这不是视图的任务，应该由别人（控制器）来做这个决定，然后再决定转发到视图。

不过我们暂且不讲MVC最佳实践，先看看如果确实要从一个JSP页面转发到其他资源，我们能怎样做。

如果你不打算这么做，还有什么必要讲这个问题呢？不过，可能有一天你会遇到某个问题，而且对于那个问题<jsp:forward>可能是一个很有用的解决方案。更重要的是，就像本书（和考试）中的大多数内容一样，<jsp:forward>的使用相当多。在你维护（或重构）的众多JSP中，很可能就潜伏着大量<jsp:forward>。

## 有条件地转发……

假设你就是一个JSP，想由一个请求来调用，请求中包含一个userName参数。因为你要依靠这个参数，所以希望先检查userName参数是否为null。如果不为null，没问题，完成响应就行了。但是，如果userName参数是null，你想停在这里，把整个请求转给别人，比如说另一个询问userName的JSP。

对现在来说，我们知道，这个工作可以用脚本做到：

### 提供条件转发的JSP(Hello.jsp)

```
<html><body>
    Welcome to our page!
    <% if (request.getParameter("userName") == null) { %>
        <jsp:forward page="HandleIt.jsp" />
    <% } %>
    Hello ${param.userName}
</body></html>
```

如果参数为null，则把请求转发到page属性指定的页面（就像使用RequestDispatcher一样）。

如果执行到这里，userName肯定是合法的！如果请求被转发，这个页面中的任何内容都不会出现在响应里。

检查请求参数。

### 请求转发到的目标JSP(HandleIt.jsp)

```
<html><body>
    We're sorry... you need to log in again.
    <form action="Hello.jsp" method="get">
        Name: <input name="userName" type="text">
        <input name="Submit" type="submit">
    </form>
</body></html>
```

这是一个普通的页面，它从用户那里得到请求参数输入，然后请求之前的那个JSP……Hello.jsp。

# 它是怎么运行的……

第一次请求Hello.jsp时，JSP会完成条件测试，发现user-Name没有值，所以转发到HandleIt.jsp。假设用户在name输入域键入一个名字，第二次请求时就不会完成转发，因为user-Name请求参数已经有一个非null的值。

## 对Hello.jsp的第一个请求

We're sorry...you need to log in again.

Name: Johannes

Submit

## 对Hello.jsp的第二个请求

Welcome to our page!

Hello Johannes



为什么第一次请求时没有打印出“Welcome to our page!”文本呢？

等一等……“Welcome to our page!”这几个词会怎么样呢？它们在Hello.jsp中，而且在转发发生之前……为什么第一次请求时不显示出来呢？



## 利用<jsp:forward>, 缓冲区会在转发之前清空

发生转发时，请求转发到的目标资源首先会清空响应缓冲区！

换句话说，转发前写到响应的所有内容都会清掉。

### there are no Dumb Questions

**问：** 如果已经缓存了页面，这是可以的……因为你写的内容会发送到缓冲区，容器只需清空缓冲区。但是，如果你先提交了响应，然后才完成转发呢？比如说，如果你写了点东西，然后在out对象上调用flush()会怎么样呢？

**答：** OK，这么问真是有点过分好奇了，因为这么做显然很傻，而且没有什么意义。你应该很清楚这一点吧。

不过还要知道，考试中没准真的会考这种奇怪的做法，因为你的同事可能很懒，不想认真学，他就很可能在代码里采用这种做法，所以你最好能习惯。

但是，你应该能想到答案是什么了。如果写了下面的代码：

```
<html><body>
```

Welcome to our page!

```
<% out.flush(); %>  
<% if (request.getParameter("userName") == null) {  
%>    <jsp:forward page="HandleIt.jsp" /> <% } %>  
Hello ${param.userName}  
</body></html>
```

容器会很负责地提交（发送）“Welcome to our page!”响应，然后容器看到要转发。唉呀，太晚了，这就会出现一个IllegalStateException异常。

不过没人会看到这个异常！客户只会看到“Welcome to our page!”……别的看不到。转发抛出了一个异常，但是已经太晚了，容器来不及收回响应，所以客户会看到已经输出的东西，仅此而已。转发没有发生，当前页面的余下部分不再处理。这个页面的工作就到此为止。所以千万不要先刷新输出再转发！

如果发生了转发，转发前写的所有内容都不会出现。



## 她还不知道JSTL标记

如果你需要更多的功能，而且只靠标准动作或EL办不到，不用非得指望脚本。下一章中，你会学习如何使用JSP标准标记库1.1（JSTL 1.1）来完成你需要的一切，可以结合使用标记和EL。这里先简单地看一下如何不用脚本就完成条件转发。

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
    Welcome to our page!
    这会替代scriptlet
    测试
    {<c:if test="${empty param.userName}" >
        <jsp:forward page="HandleIt.jsp" />
    </c:if>
    Hello ${param.userName}
</body></html>
```

声明一个taglib指令。  
指定标记所在的库。

顺便说一句……这可能还不能运行，因为你的Web应用中没有JSTL。有关内容将在下一章介绍。

# Bean相关标准动作复习



## 要点

- <jsp:useBean>标准动作会定义一个变量，它可能是一个现有bean属性的引用，如果还不存在这样一个bean，则会创建一个新的bean，这个变量就是新bean的引用。
- <jsp:useBean>必须有一个“id”属性，这个属性声明了JSP中引用bean时所用的变量名。
- 如果<jsp:useBean>中没有“scope”属性，作用域默认为页面（page）作用域。
- “class”属性是可选的，它声明了创建一个新bean时使用的类类型。这个类型必须是公共的、非抽象的，而且有一个公共的无参数构造函数。
- 如果在<jsp:useBean>中放了一个“type”属性，bean必须能强制转换为这种类型。
- 如果有一个“type”属性，但是没有“class”属性，bean必须已经存在，因为你没有指定为这个新bean实例化哪个类类型。
- <jsp:useBean>标记可以有一个体，体中的内容会有条件地运行，只有当创建一个新的bean作为<jsp:useBean>的结果时，才会运行体中的内容 [这说明，指定（或默认）作用域中不存在有该“id”的bean]。
- <jsp:useBean>体的主要作用是使用<jsp:setProperty>设置新bean的性质。
- <jsp:setProperty>必须有一个name属性（它要与<jsp:useBean>的“id”匹配），还要有一个“property”属性。“property”属性必须是一个具体的性质名，或者是通配符“\*”。
- 如果没有包含“value”属性，只有当一个请求参数的名与性质名匹配时，容器才会设置性质值。如果“property”属性使用通配符（\*），容器会为所有与某个请求参数名匹配的性质设置值。（其他性质不受影响）
- 如果请求参数名与性质名不同，但是你想把性质的值设置为请求参数值，可以在<jsp:setProperty>标记中使用“param”属性。
- <jsp:setProperty>动作使用自省将“性质”匹配到一个JavaBean设置方法。如果性质是“\*”，JSP将迭代处理所有请求参数来设置JavaBean性质。
- 性质值可以是String或基本类型，<jsp:setProperty>标准动作会自动完成转换。

# 包含复习

## 要点

- 通过使用两种包含机制之一（`include`指令或`<jsp:include>`标准动作），可以利用可重用的组件建立页面。
- `include`指令在转换时完成包含，只发生一次。所以如果包含的内容一经部署后不太可能改变，使用`include`指令就很合适。
- `include`指令实际上只是复制被包含文件中的所有内容，把它粘贴到有`include`指令的页面中。容器把所有被包含文件的内容合并起来，只编译一个文件来生成servlet。在运行时，有`include`指令的页面将成为一个“大”页面，就像是你自己把所有源代码键入到一个文件中一样。
- `<jsp:include>`标准动作在运行时把被包含页面的响应包含到原页面中。所以如果包含的内容在部署之后可能更新，`include`标准动作就很适用，此时不适用`include`指令。
- 这两种机制都能包含静态HTML页面，也可以包含动态元素（例如，有EL表达式的JSP代码）。
- `include`指令是唯一一个对位置敏感的指令，所包含的内容会插入到页面中`include`指令所在的位置。
- `include`指令和`include`标准动作的属性命名不一致，指令使用“file”属性，而标准动作使用“page”属性。
- 在你的可重用组件中，一定要去掉开始和结束标记。否则，生成的输出会有嵌套的

开始和结束标记，对于这种嵌套的开始和结束标记，并非所有浏览器都能处理。设计和构造可重用部件时，要知道它们会包含/插入到别的页面中。

- 可以在`<jsp:include>`的体中使用`<jsp:param>`标准动作设置（或替换）请求参数，用来定制所包含的文件。
- 尽管在这一章没有介绍，但是要知道，`<jsp:param>`也可以用在`<jsp:forward>`标记的体中。
- `<jsp:param>`只能放在`<jsp:include>` 或`<jsp:forward>`标准动作中。
- 如果`<jsp:param>`中使用的参数名已经有一个值（作为请求参数），新值会覆盖原来的值。否则，就会向请求增加一个新的请求参数。
- 对被包含资源有一些限制：它不能改变响应状态码或设置首部。
- `<jsp:forward>`标准动作可以把请求转发到同一个Web应用中的另一个资源（就像使用`RequestDispatcher`一样）。
- 发生转发时，会首先清空响应缓冲区！请求转发到的目标资源会先清空输出。所以转发前写至响应的所有内容都会清掉。
- 如果在转发之前先提交响应（例如，通过调用`out.flush()`），会把刷新输出的内容发送给客户，但是仅此而已。不会发生转发，原页面的余下部分不会得到处理。



## 作为容器答案

请看这个标准动作:

体不会运行！如果只有  
type而没有class, 在<jsp:  
useBean>标记中放一个体  
是没有意义的！记住，只  
有创建一个新bean时才会有  
执行标记体。但如果标记  
中只声明了一个type (没有  
class)，就根本不可能创  
建bean的新实例。

注意：这里有type,  
但是没有class。

```
<jsp:useBean id="person" type="foo.Employee" scope="request" >
    <jsp:setProperty name="person" property="name" value="Fred" />
</jsp:useBean >
Name is: <jsp:getProperty name="person" property="name" />
```

如果运行到这里，  
会打印“Evan”。

① 如果servlet代码如下，会怎么样？

```
foo.Person p = new foo.Employee();
p.setName("Evan");
request.setAttribute("person", p);
```

请求时会失败：“person”属性存储在请求作用域，所以<jsp:useBean>标记不会工作，因为只指定了一个type。容器知道，如果你只指定了一个type，就必须已经有具有该名和作用域的一个bean属性。

抽象

Person

```
String getName()
void setName(String)
```



Employee

```
int getEmpID()
void setEmpID(int)
```

这两个类都在包  
“foo”中。

② 如果servlet代码如下，会怎么样？

```
foo.Person p = new foo.Person();
p.setName("Evan");
request.setAttribute("person", p);
```

实际上这个servlet无法编译。我们有点误导，因为这个问题实际上不能从“作为容器”来考虑，而更应算是“作为编译器”。  
foo.Person现在是抽象的，所以不能实例化foo.Person。



## 第8章 模拟测验

**1** 给定一个HTML表单，其中使用了复选框，以便用户为一个名为**hobbies**的参数选择多个值。

以下哪个EL表达式能计算得到**hobbies**参数的第一个值？（选出所有正确的答案）

- A. \${param.hobbies}
- B. \${paramValue.hobbies}
- C. \${paramValues.hobbies[0]}
- D. \${paramValues.hobbies[1]}
- E. \${paramValues[hobbies][0]}
- F. \${paramValues[hobbies][1]}

**2** 给定一个Web应用，它把网站管理员的email地址存储在一个名为**master-email**的servlet上下文初始化参数中。

怎样获取这个值？（选出所有正确的答案）

- A. <a href='mailto:\${initParam.master-email}'>  
email me</a>
- B. <a href='mailto:\${contextParam.master-email}'>  
email me</a>
- C. <a href='mailto:\${initParam['master-email']}'>  
email me</a>
- D. <a href='mailto:\${contextParam['master-email']}'>  
email me</a>

**3** 给定以下Java类：

```
1. package com.mycompany;
2. public class MyFunctions {
3.     public static String hello(String name) {
4.         return "Hello "+name;
5.     }
6. }
```

这个类表示了一个函数的处理器，这个函数是标记库的一部分。

```
<%@ taglib uri="http://mycompany.com.tags" prefix="comp" %>
```

以下哪个标记库描述文件项定义了这个定制函数，从而能在EL表达式中使用这个函数？

A. <taglib>

```
...
<tag>
    <name>Hello</name>
    <tag-class>com.mycompany.MyFunctions</tag-class>
    <body-content>JSP</body-content>
</tag>
</taglib>
```

B. <taglib>

```
...
<function>
    <name>Hello</name>
    <function-class>com.mycompany.MyFunctions</function-class>
    <function-signature>java.lang.String hello(java.lang.String)
        </function-signature>
    </function>
</taglib>
```

C. <web-app>

```
...
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>com.mycompany.MyFunctions</servlet-class>
</servlet>
</web-app>
```

D. <taglib>

```
...
<function>
    <name>Hello</name>
    <function-class>com.mycompany.MyFunctions</function-class>
    <function-signature>hello(java.lang.String)</function-signature>
    </function>
</taglib>
```

4

给定：

```

1. package com.example;
2. public class TheBean {
3.     private int value;
4.     public TheBean() { value = 42; }
5.     public int getValue() { return value; }
6.     public void setValue(int v) { value = v; }
7. }
```

假设还没有创建**TheBean**的实例，以下哪些JSP标准动作语句能创建这个bean的一个新实例，并把它存储在请求作用域？（选出所有正确的答案）。

- A. <jsp:useBean name="myBean" type="com.example.TheBean" />
- B. <jsp:makeBean name="myBean" type="com.example.TheBean" />
- C. <jsp:useBean id="myBean" class="com.example.TheBean" scope="request" />
- D. <jsp:makeBean id="myBean" class="com.example.TheBean" scope="request" />

5

给定一个Model 1体系结构，其中由一个JSP页面处理所有控制器函数，这个JSP控制器要把请求分派给另一个JSP页面。

以下哪个标准动作代码能完成这种分派？

- A. <jsp:forward page="view.jsp" />
- B. <jsp:forward file="view.jsp" />
- C. <jsp:dispatch page="view.jsp" />
- D. <jsp:dispatch file="view.jsp" />

6

给定：

```
11. <% java.util.List list = new java.util.ArrayList();  
12.     list.add("a");  
13.     list.add("2");  
14.     list.add("c");  
15.     request.setAttribute("list", list);  
16.     request.setAttribute("listIdx", "1");  
17. %>  
18. <%-- insert code here --%>
```

以下哪些语句插到第18行是合法的，而且计算为c？（选出所有正确的答案）

- A. \${list.2}
- B. \${list[2]}
- C. \${list.listIdx+1}
- D. \${list[listIdx+1]}
- E. \${list['listIdx' + 1]}
- F. \${list[list[listIdx]]}

7

关于. (点号)和[] EL操作符，以下哪些说法是正确的？（选出所有正确的答案）

- A. \${foo.bar} 等价于 \${foo[bar]}
- B. \${foo.bar} 等价于 \${foo["bar"]}
- C. 如果 foo 是一个 Map, \${foo[ "5" ]} 就是正确的语法。
- D. \${header.User-Agent} 等价于  
\${header[User-Agent]}
- E. \${header.User-Agent} 等价于  
\${header["User-Agent"]}
- F. 如果 foo 是一个 List 或数组, \${foo[5]} 就是正确的语法。



**8** 给定一个有以下代码行的JSP页面：

```
 ${101 % 10}
```

会显示什么结果？

- A. 1
- B. 10
- C. 1001
- D. 101 % 10
- E. {101 % 10}

**9** 给定：

```
10. ${param.firstname}
11. ${param.middlename}
12. ${param.lastname}
13. ${paramValues.lastname[0]}
```

如果传递查询串?firstname=John&lastname=Doe，这部分JSP页面的输出是什么？

- A. John Doe
- B. John Doe Doe
- C. John null Doe
- D. John null Doe Doe
- E. 抛出一个空指针异常。

**10** 以下哪些代码正确地使用了EL隐式变量？（选出所有正确的答案）

- A. \${cookies.foo}
- B. \${initParam.foo}
- C. \${pageContext.foo}
- D. \${requestScope.foo}
- E. \${header["User-Agent"]}
- F. \${requestDispatcher.foo}
- G. \${pageContext.request.requestURI}

11 有关<jsp:useBean>标准动作,以下哪些说法是正确的?

(选出所有正确的答案)

- A. **id**属性是可选的。
- B. **scope**属性是必要的。
- C. **scope**属性是可选的,默认为**request**。
- D. **class**或**type**属性都可以指定,但至少指定一个。
- E. 可以同时包括**class**属性和**type**属性,不过它们的值可以不一样。

12 类似于服务器端包含 (server-side include, SSI),如何在JSP中包含动态内容?

(选出所有正确的答案)

- A. <%@ include file="/segments/footer.jspf" %>
- B. <jsp:forward page="/segments/footer.jspf" />
- C. <jsp:include page="/segments/footer.jspf" />
- D. RequestDispatcher dispatcher  
= request.getRequestDispatcher("/segments/footer.jspf");  
dispatcher.include(request, response);

13 如果HTML页面有一个丰富的图形化布局,可以用哪个JSP标准动作把一个图像文件导入到JSP页面中?

- A. <jsp:image page="logo.png" />
- B. <jsp:image file="logo.png" />
- C. <jsp:include page="logo.png" />
- D. <jsp:include file="logo.png" />
- E. 使用JSP标准动作无法做到。

**4** 给定：

```

1. package com.example;
2. public class MyFunctions {
3.     public static String repeat(int x, String str) {
4.         // method body
5.     }
6. }
```

并给定JSP：

```

1. <%@ taglib uri="/WEB-INF/myfuncts" prefix="my" %>
2. <%-- insert code here --%>
```

如果插入到JSP中的第2行，以下哪一个是合法的EL函数调用？

- A. \${repeat(2, "420")}
- B. \${repeat("2", "420")}
- C. \${my:repeat(2, "420")}
- D. \${my:repeat("2", "420")}
- E. 无法确定合法的调用。

**5** 给定：

```

10. public class MyBean {
11.     private java.util.Map params;
12.     private java.util.List objects;
13.     private String name;
14.     public java.util.Map getParams() { return params; }
15.     public String getName() { return name; }
16.     public java.util.List getObjects() { return objects; }
17. }
```

以下哪些代码会导致错误（假设可以找到一个名为mybean而且类型为MyBean的属性）？（选出所有正确的答案）

- A. \${mybean.name}
- B. \${mybean["name"]}
- C. \${mybean.objects.a}
- D. \${mybean["params"].a}
- E. \${mybean.params["a"]}
- F. \${mybean["objects"].a}

---

16 给定一个JSP页面：

1. The user has sufficiently logged in or out:
2. \${param.loggedIn or param.loggedOut}.

如果请求包含查询串“`loggedOut=true`”，这条语句显示的值是什么？

- A. The user has sufficiently logged in or out: `false`.
  - B. The user has sufficiently logged in or out: `true`.
  - C. The user has sufficiently logged in or out: `${param.loggedIn or param.loggedOut}`.
  - D. The user has sufficiently logged in or out: `param.loggedIn or param.loggedOut`.
  - E. The user has sufficiently logged in or out: or `true`.
- 

17 关于EL访问操作符，以下哪些说法是正确的？（选出所有正确的答案）

- A. 能使用.(点号)操作符的地方，都能使用[]。
  - B. 能使用[]操作符的地方，都能使用.(点号)操作符。
  - C. 如果用.(点号)操作符来访问一个bean性质，但是这个性质并不存在，就会抛出一个运行时异常。
  - D. 有些情况下必须使用.(点号)操作符，而另外一些情况下必须使用[]操作符。
- 

18 一个JSP页面中有以下代码片段：

```
<jsp:include page="/jspf/header.html"/>
```

这个JSP页面是一个Web应用的一部分，该应用的上下文根为`myapp`。

给定应用的顶级目录是`myapp`, `header.html`文件的路径是什么？

- A. /`header.html`
- B. /`jspf/header.html`
- C. /`myapp/jspf/header.html`
- D. /`includes/jspf/header.html`

| 9

一个在线珠宝零售商希望为登录用户定制他们的在线商品目录。他们希望对应用客户诞生石相应月份显示一些特价珠宝。公司的特价珠宝存储为一个**Map<String, Special[]>**中，这标识为应用作用域中的**specials**，并会每日更新。

还有一个bean存储为一个会话作用域属性，名为**userInfo**。在这个bean上调用**getBirthdate().getMonth()**会返回用户的诞生石月份。

以下哪个代码片段可以正确地获取适当的特价珠宝？

- A. \${applicationScope[userInfo.birthdate.month.specials]}
- B. \${applicationScope.specials[userInfo.birthdate.month]}
- C. \${applicationScope["specials"].userInfo.birthdate.month}
- D. \${applicationScope["userInfo.birthdate.month"].specials}

20

为一家大型在线影碟租售店建立了一个基于web的应用，其中将一个**List<Movie>**存储为会话属性，这个List中包含用户索要的影片。每次查看用户的主页时，都必须在用户主页上显示从这个列表中随机抽取的一个宣传短片。

管理部门认为，不久的将来在其他显示影片列表的页面上也需要一个类似特性。影片的流动播放使用常规的HTML完成，类似于为页面增加图像，不过使用了更复杂的标记。

开发小组需要一种灵活而且可维护的解决方案。一种可能的解决方法是创建一个EL函数。关于采用EL函数作为这个问题的解决方案，开发小组专门开会进行了讨论，以下就是讨论会上的各种说法。其中哪些说法是正确的？（选出所有正确的答案）

- A. EL函数不能解决这个问题，因为EL函数不能获取会话属性。
- B. 实现EL函数的方法不能声明为静态，以便能够访问会话作用域。
- C. EL函数可以接受**java.util.List**参数，这就允许使用EL将所需影片列表传递给它。
- D. 可能必须使用EL函数在Java代码中间编写HTML标记，这将更难维护。



## 第8章 模拟测验答案

1 给定一个HTML表单，其中使用了复选框，以便用户为一个名为**hobbies**的 (JSP v2.0 2.2.3.)节)参数选择多个值。

以下哪个EL表达式能计算得到**hobbies**参数的第一个值？（选出所有正确的答案）

- A. \${param.hobbies}
- B. \${paramValue.hobbies} B不对，因为没有“paramValue”  
隐式变量。
- C. \${paramValues.hobbies[0]}
- D. \${paramValues.hobbies[1]} D不对，数组的索引从0开始。
- E. \${paramValues[hobbies][0]} E和F的语法不正确。
- F. \${paramValues[hobbies][1]}

2 给定一个web应用，它把网站管理员的email地址存储在一个名为master-email的 (JSP v2.0 2.2.3和  
servlet上下文初始化参数中。 2.3.4.)节)

怎样获取这个值？（选出所有正确的答案）

- A. <a href='mailto:\${initParam.master-email}'> A试图将master藏去  
email me</a> email (做减法)。
- B. <a href='mailto:\${contextParam.master-email}'> B. 没有contextParam隐式变量。  
email me</a>
- C. <a href='mailto:\${initParam['master-email']}'>  
email me</a>
- D. <a href='mailto:\${contextParam['master-email']}'> D. 没有contextParam隐式变量。  
email me</a>

3

给定以下Java类：

(JSP v2.0 2.6.3.1节)

```
1. package com.mycompany;
2. public class MyFunctions {
3.     public static String hello(String name) {
4.         return "Hello "+name;
5.     }
6. }
```

这个类表示了一个函数的处理器，这个函数是标记库的一部分。

```
<%@ taglib uri="http://mycompany.com.tags" prefix="comp" %>
```

以下哪个标记库描述文件项定义了这个定制函数，从而能在EL表达式中使用这个函数？

A. <taglib>

```
...
<tag>
    <name>Hello</name>
    <tag-class>com.mycompany.MyFunctions</tag-class>
    <body-content>JSP</body-content>
</tag>
</taglib>
```

B. <taglib>

B 使用了正确的语法。

```
...
<function>
    <name>Hello</name>
    <function-class>com.mycompany.MyFunctions</function-class>
    <function-signature>java.lang.String hello(java.lang.String)
        </function-signature>
    </function>
</taglib>
```

C. <web-app>

```
...
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>com.mycompany.MyFunctions</servlet-class>
</servlet>
</web-app>
```

D. <taglib>

D 不对，因为函数签名不完整。

```
...
<function>
    <name>Hello</name>
    <function-class>com.mycompany.MyFunctions</function-class>
    <function-signature>hello(java.lang.String)</function-signature>
</function>
</taglib>
```

**4** 给定：

(JSP v2.0 5.1.1节)

```

1. package com.example;
2. public class TheBean {
3.     private int value;
4.     public TheBean() { value = 42; }
5.     public int getValue() { return value; }
6.     public void setValue(int v) { value = v; }
7. }
```

假设还没有创建**TheBean**的实例，以下哪些JSP标准动作语句能创建这个bean的一个新实例，并把它存储在请求作用域？（选出所有正确的答案）

- A. <jsp:useBean name="myBean" type="com.example.TheBean" /> A不对，因为type属性不能用来创建一个新实例，而且必须指定scope属性（否则默认为page）。
- B. <jsp:makeBean name="myBean" type="com.example.TheBean" /> B不对，原因同上，而且jsp:makeBean并不是一个真正的标记。
- C. <jsp:useBean id="myBean" class="com.example.TheBean" scope="request" /> C不对，因为jsp:makeBean不是一个真正的标记。
- D. <jsp:makeBean id="myBean" class="com.example.TheBean" scope="request" />

**5**

给定一个Model 1体系结构，其中由一个JSP页面处理所有控制器函数，这个JSP控制器要把请求分派给另一个JSP页面。

(JSP v2.0 5.5.1节)

以下哪个标准动作代码能完成这种分派？

- A. <jsp:forward page="view.jsp" /> A是对的 (1-10页)。
- B. <jsp:forward file="view.jsp" /> B不对，因为forward动作没有file属性。
- C. <jsp:dispatch page="view.jsp" /> C和D不对，因为没有dispatch动作。
- D. <jsp:dispatch file="view.jsp" />

6

给定：

```

11. <% java.util.List list = new java.util.ArrayList();
12.     list.add("a");
13.     list.add("2");
14.     list.add("c");
15.     request.setAttribute("list", list);
16.     request.setAttribute("listIdx", "1");
17. %>
18. <%-- insert code here --%>
```

以下哪些语句插到第18行是合法的，而且计算为c？（选出所有正确的答案）

- A. \${list.2} A和C不对，因为基本类型不能使用点号操作符。
- B. \${list[2]}
- C. \${list.listIdx+1}
- D. \${list[listIdx+1]} E不对，因为EL试图将‘listIdx’ 强制转换为一个Long，这是不合法的。
- E. \${list['listIdx' + 1]}
- F. \${list[list[listIdx]]}

7

关于. (点号)和[] EL操作符，以下哪些说法是正确的？（选出所有正确的答案）。 (JSP v2.0 (～69页))

- A. \${foo.bar} 等价于 \${foo[bar]} A不对，应当是 foo[“bar”]。

- B. \${foo.bar} 等价于 \${foo[“bar”]}

- C. 如果foo是一个Map, \${foo[“5”]} 就是正确的语法。

- D. \${header.User-Agent} 等价于 \${header[User-Agent]} D和E不对，因为User-Agent中有“-”。  
只有header[“User-Agent”]能正常工作。

- E. \${header.User-Agent} 等价于 \${header[“User-Agent”]}

- F. 如果foo是一个List或数组, \${foo[5]}就是正确的语法。

8

给定一个有以下代码行的JSP页面：

(JSP v2.0 1~71页)

 `${101 % 10}`

会显示什么结果？

- A. 1                    A是对的。取模操作符返回除法操作的余数。
- B. 10
- C. 1001
- D. 101 % 10
- E. {101 % 10}

9

给定：

(JSP v2.0 1~67页和  
79页)

```
10. ${param.firstname}
11. ${param.middlename}
12. ${param.lastname}
13. ${paramValues.lastname[0]}
```

如果传递查询串?firstname=John&amp;lastname=Doe，这部分JSP页面的输出是什么？

- A. John Doe                A不对，因为第13行还会打印用户的姓。
- B. John Doe Doe
- C. John null Doe            C和D不对，因为第11行不会打印“null”，而是什么也不打印。
- D. John null Doe Doe
- E. 抛出一个空指针异常。

10

以下哪些代码正确地使用了EL隐式变量？（选出所有正确的答案）。

(JSP v2.0 1~66页)

- A. \${cookies.foo}            A不对，因为变量是“cookie”。
- B. \${initParam.foo}
- C. \${pageContext.foo}            C不对，因为pageContext不是一个Map，而且没有一个“foo”性质。
- D. \${requestScope.foo}
- E. \${header["User-Agent"]}
- F. \${requestDispatcher.foo}            F不对，因为这不是一个隐式对象。
- G. \${pageContext.request.requestURI}

11

有关`<jsp:useBean>`标准动作,以下哪些说法是正确的?  
(选出所有正确的答案)。

- A. `id`属性是可选的。 A不对,因为`id`是必要的。
- B. `scope`属性是必要的。 B和C不对,因为`scope`是可选的,而且默认为`page`。
- C. `scope`属性是可选的,默认为`request`。
- D. `class`或`type`属性都可以指定,但至少指定一个。
- E. 可以同时包括`class`属性和`type`属性,不过它们的值可以不一样。

(JSP v2.0 1~103页和1~104页)

(JSP v2.0 5.4.1节)

12

类似于服务器端包含(server-side include, SSI),如何在JSP中包含动态内容?  
(选出所有正确的答案)。

- A. `<%@ include file="/segments/footer.jspf" %>`
- B. `<jsp:forward page="/segments/footer.jspf" />`
- C. `<jsp:include page="/segments/footer.jspf" />`
- D. `RequestDispatcher dispatcher = request.getRequestDispatcher("/segments/footer.jspf"); dispatcher.include(request, response);`

A不对,因为它使用了一个`include`指令,这用于静态包含,只在转换时发生。

如果这是一个scriptlet,D就是对的:从功能上看它与C的作用是一样的,但是这种语法只用于servlet。

13

如果HTML页面有一个丰富的图形化布局,可以用哪个JSP标准动作把一个图像文件导入到JSP页面中?

- A. `<jsp:image page="logo.png" />` A和B不对,因为没有`image`标准动作。

(JSP v2.0 5.4.1节)

- B. `<jsp:image file="logo.png" />`

C不对,不是因为`include`动作的语法错,而是因为把图像文件的二进制数据导入到JSP内容中是没有意义的。

- C. `<jsp:include page="logo.png" />`

- D. `<jsp:include file="logo.png" />` D不对,因为`include`动作没有`file`属性。

- E. 使用JSP标准动作无法做到。

这个问题有点难度,因为不可能把一个二进制文件的内容导入到一个JSP页面中,这会生成一个HTML响应。

14

给定:

(JSP v2.0 2.6.) 节)

```

1. package com.example;
2. public class MyFunctions {
3.   public static String repeat(int x, String str) {
4.     // method body
5.   }
6. }
```

并给定JSP:

```

1. <%@ taglib uri="/WEB-INF/myfuncts" prefix="my" %>
2. <%-- insert code here --%>
```

如果插入到JSP中的第2行,以下哪一个是合法的EL函数调用?

- A. \${repeat(2, "420")}
- B. \${repeat("2", "420")}
- C. \${my:repeat(2, "420")}
- D. \${my:repeat("2", "420")}
- E. 无法确定合法的调用。

E是对的。还没有从TLD得到必要的映射信息。

15

给定:

(JSP v2.0 (~68页))

```

10. public class MyBean {
11.   private java.util.Map params;
12.   private java.util.List objects;
13.   private String name;
14.   public java.util.Map getParams() { return params; }
15.   public String getName() { return name; }
16.   public java.util.List getObjects() { return objects; }
17. }
```

以下哪些代码会导致错误(假设可以找到一个名为mybean而且类型为MyBean的属性)?(选出所有正确的答案)

- A. \${mybean.name}
- B. \${mybean["name"]}
- C. \${mybean.objects.a}
- D. \${mybean["params"].a}
- E. \${mybean.params["a"]}
- F. \${mybean["objects"].a}

C和F会导致错误。“a”不是一个List性质,由于“objects”不是一个Map,所以不会完成查找(不同于D和E)。

16 给定一个JSP页面:

(JSP v2.0 1~66页  
(~73页)

1. The user has sufficiently logged in or out:
2. \${param.loggedIn or param.loggedOut}.

如果请求包含查询串“loggedOut=true”，这条语句显示的值是什么？

- A. The user has sufficiently logged in or out: false.
- B. The user has sufficiently logged in or out: true.
- C. The user has sufficiently logged in or out: \${param.loggedIn or param.loggedOut}.
- D. The user has sufficiently logged in or out: param.loggedIn or param.loggedOut.
- E. The user has sufficiently logged in or out: or true.

B是对的，因为如果EL表达式使用“or”时，只要loggedIn或loggedOut任意一个为true，就会返回true。

17 关于EL访问操作符，以下哪些说法是正确的？（选出所有正确的答案）。

(JSP v2.0 1~69页)

- A. 能使用.(点号)操作符的地方，都能使用[]。

- B. 能使用[]操作符的地方，都能使用.(点号)操作符。

- C. 如果用.(点号)操作符来访问一个bean性质，但是这个性质并不存在，就会抛出一个运行时异常。

- D. 有些情况下必须使用.(点号)操作符，而另外一些情况下必须使用[]操作符。D不对，因为点号操作符总能转换为[]操作符。

18 一个JSP页面中有以下代码片段：

(JSP v2.0 5.4.1节)

```
<jsp:include page="/jspf/header.html"/>
```

这个JSP页面是一个Web应用的一部分，该应用的上下文根为myapp。

给定应用的顶级目录是myapp, header.html文件的路径是什么？

- A. /header.html
- B. /jspf/header.html
- C. /myapp/jspf/header.html
- D. /includes/jspf/header.html

路径 /jspf/header.html 用作为 <jsp:include> 动作的 page 属性值时，它要相对于 Web 应用，所以前面的反斜线（“/”）表示“从应用的顶级目录开始”。

19

一个在线珠宝零售商希望为登录用户定制他们的在线商品目录。他们希望对应用用户诞生石相应月份显示一些特价珠宝。公司的特价珠宝存储为一个 `Map<String, Special[]>` 中，这标识为应用作用域中的 `specials`，并会每日更新。

还有一个bean存储为一个会话作用域属性，名为 `userInfo`。在这个bean上调用 `getBirthdate().getMonth()` 会返回用户的诞生石月份。

以下哪个代码片段可以正确地获取适当的特价珠宝？

- A.  `${applicationScope[userInfo.birthdate.month.specials]}`
- B.  `${applicationScope.specials[userInfo.birthdate.month]}`
- C.  `${applicationScope["specials"].userInfo.birthdate.month}`
- D.  `${applicationScope["userInfo.birthdate.month"].specials}`

B能从应用作用域正确地获取 `Map<String, Special[]>`。然后试图从用户的生日得到月份值，并使用这个月份值作为键在 Map 中搜索一个 `Special[]`。假设在 Map 中查找到一个匹配月份，则返回 `Special[]`。这个EL可以用在一个 `c:forEach` 标记中迭代处理返回的特价珠宝。

20

为一家大型在线影碟租售店建立了一个基于 web 的应用，其中将一个 `List<Movie>` 存储为会话属性，这个 List 中包含用户索要的影片。每次查看用户的主页时，都必须在用户主页上显示从这个列表中随机抽取的一个宣传短片。

管理部门认为，不久的将来在其他显示影片列表的页面上也需要一个类似特性。影片的流动播放使用常规的 HTML 完成，类似于为页面增加图像，不过使用了更复杂的标记。

开发小组需要一种灵活而且可维护的解决方案。一种可能的解决方案是创建一个 EL 函数。关于采用 EL 函数作为这个问题的解决方案，开发小组专门开会进行了讨论，以下就是讨论会上的各种说法。其中哪些说法是正确的？（选出所有正确的答案）。

- A. EL 函数不能解决这个问题，因为 EL 函数不能获取会话属性。
- B. 实现 EL 函数的方法不能声明为静态，以便能够访问会话作用域。
- C. EL 函数可以接受 `java.util.List` 参数，这就允许使用 EL 将所需影片列表传递给它。
- D. 可能必须使用 EL 函数在 Java 代码中间编写 HTML 标记，这将更难维护。

A: 影片列表可以作为一个参数传递到函数。

B: 实现 EL 函数的方法必须是公共方法，而且必须是静态。

C: 列表可以传递到函数。

D: 之所以不选择 EL 函数作为总的解决方案，这是最主要的原因。开发小组最终选择使用一个标记文件作为解决方案，不过还创建了一个 EL 函数，它接受一个 Collection，并根据这个集合的大小返回一个随机数。

# 强大的定制标记



你是说，我花了这么  
大功夫写scriptlet是不必要的，  
尽管使用EL和标准动作达不到目  
的，但我完全可以使JSTL，  
是这样吗？

有时只是EL或标准动作还不够。如果你想循环处理一个数组中的数据，并在一个HTML表中每行显示一项，该怎么做？你很清楚，如果在一个scriptlet中使用for循环，只需两秒钟问题就能迎刃而解。不过你想尽量避开脚本。没问题。倘若EL和标准动作力不能及，你还可以使用定制标记。在JSP中，使用定制标记与使用标准动作同样简单。更妙的是，已经有人写了一大堆你很可能需要的定制标记，并把它们打包在JSP标准标记库（JSP Standard Tag Library, JSTL）中。在这一章中，我们将学习如何使用定制标记，下一章再介绍如何创建我们自己的定制标记。

# OBJECTIVES

## 使用标记库建立JSP页面

- 9.1 描述‘taglib’指令的语法和语义：包括标准标记库的相应‘taglib’指令和标记文件库的相应‘taglib’指令。
- 9.2 创建定制标记结构来支持给定的设计目标。
- 9.3 对于以下JSP标准标记库（JSTL v1.1）标记，明确这些标记的语法，并描述标记的动作语义：(a)核心标记：out, set, remove和catch，(b) 条件标记：if, choose, when和otherwise，(c) 循环标记：forEach和(d) 与URL相关的标记：url。

## 内容说明：

这一部分的所有目标在本章都会介绍，不过有些内容在下一章（开发定制标记）还会谈到。

### 安装JSTL 1.1

JSTL 1.1不是JSP 2.0规范的一部分！能访问Servlet和JSP API并不意味着你能访问JSTL。

使用JSTL之前，需要将两个文件（“jstl.jar”和“standard.jar”）放在Web应用的WEB-INF/lib 目录中。这说明每个Web应用都需要一个副本。

在Tomcat 5中，随Tomcat发布的示例应用中已经有这两个文件，所以你只需从其中一个应用的WEB-INF/lib目录将这两个文件复制到你自己的WEB-INF/lib目录下就可以了。

从Tomcat的示例应用复制文件，位置如下：

webapps/jsp-examples/WEB-INF/  
lib/jstl.jar  
webapps/jsp-examples/WEB-INF/  
lib/standard.jar

把它们放在你自己的Web应用的WEB-INF/lib目录中。



## EL和标准动作是有限制的

如果使用EL和标准动作达不到目的，你会怎么做？当然可以用老一套，还是像以前一样写脚本，但你很清楚，这绝不是上策。

开发人员总是希望标准动作越多越好，如果能创建自己的动作就更好了。

这就引入了定制标记。你可能不想用<jsp:setProperty>，而是想要<my:doCustomThing>之类的标记，你的愿望确实能成真。

但是创建标记底层的支持代码并不容易。对于JSP页面创作人员来说，定制标记使用起来比脚本要容易一些。不过，对于Java程序员来说，建立定制标记处理器（JSP使用标记时实际调用的Java代码）反而更困难。

幸运的是，已经有一个标准的定制标记库，这称为JSP标准标记库（JSP Standard Tag Library, JSTL 1.1）。既然JSP不应完成太多的业务逻辑，你会发现，JSTL（并结合EL）一般来说完全够用了。不过，有些时候你可能还需要使用其他定制标记库中的标记，例如一个专门为你的公司开发的定制标记库。

在这一章中，你将了解到如何使用核心JSTL标记，以及如何使用其他库中的定制标记。下一章中，我们再介绍如何具体建立类来处理定制标记的调用，也就是开发你自己的定制标记。

# HTML消失案件(问题重现)

384页上已经看到EL如何将原始内容串直接发送到响应流：

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    ${pageContent.currentTip}
</div>
```

还记得这个吗：<b></b>标记没有作为文本显示，而是呈现为一个加粗的空格。

实际得到的

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/>
    <b></b> tags make things bold!
</div>
```

希望得到的

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    &lt;b&gt;&lt;/b&gt; tags make things bold!
</div>
```

这会成为一个“看不见”的加粗空格。

呈现为

呈现为

&lt; 呈现为 “<” , &gt; 呈现为 “>”。

http://localhost:8080/testJSP1/Test01.do

Tip of the Day:

tags make things bold!

http://localhost:8080/testJSP1/Test01.do

Tip of the Day:

<b></b> tags make things bold!

我们需要一种办法转换这些尖括号，使浏览器能够将转换得到的内容呈现为尖括号，对此有两种方法。这两种方法都使用一个静态Java方法，将HTML特殊字符转换为其相应的实体格式：

使用一个EL函数

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    ${fn:convEntity(pageContent.currentTip)}
</div>
```

使用一个Java辅助方法

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    ${pageContent.convertedCurrentTip}
</div>
```

这就是用来达到这个目的的辅助方法。

```
public String getConvertedCurrentTip() {
    return HTML.convEntity(getCurrentTip());
}
```

# 还有一种更好的办法：使用<c:out>标记

不论使用以上哪一种方法，并不太清楚究竟发生了什么……而且可能必须为你的所有servlet编写这样一个辅助方法。幸运的是，还有一种更好的办法。<c:out>标记就非常适合完成这个任务。下面来看是如何进行转换的：

## 可以显式地声明转换XML实体

如果你知道或认为可能遇到一些需要显示而不只是呈现的XML实体，那么可以使用c:out的escapeXml属性。将这个属性设置为true，这表示所有XML都将转换为web浏览器能呈现的形式，包括尖括号和所有XML内容都将转换：

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    <c:out value='${pageContent.currentTip}' escapeXml='true' />
</div>
```

## 可以显式地声明不转换XML实体

有些情况下，你希望的可能正好相反。也许你在构建一个取内容的页面，而且希望用HTML格式显示这个内容。在这种情况下，可以关闭XML转换：

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    <c:out value='${pageContent.rawHTML}' escapeXml='false' />
</div>
```

## 转换会默认发生

escapeXml属性默认为true，所以如果愿意完全可以不设置这个属性。如果c:out标记没有escapeXML属性，这就等同于一个escapeXML设置为“true”的c:out标记。

```
<div class='tipBox'>
    <b>Tip of the Day:</b> <br/> <br/>
    <c:out value='${pageContent.currentTip}' />
</div>
```

你的HTML会处理为XHTML，并进一步处理为XML……所以这也会影响HTML字符。

这就等价于我们之前的做法……所有HTML标记都会得到计算，而不是作为文本显示。

这二者功能完全相同。

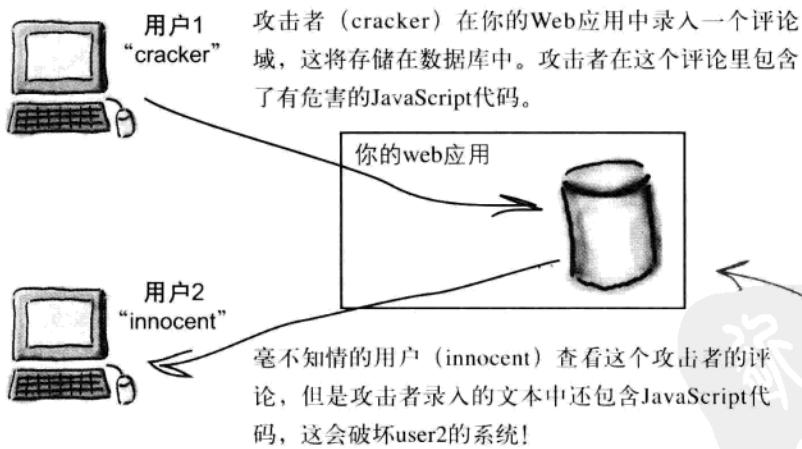
# there are no Dumb Questions

**问：** 哪些HTML特殊字符要转换？

**答：** 其实这种转换相当简单。只有5个字符需要转换：<、>、&以及两个引号，单引号’和双引号”。所有这些都会转换为等价的HTML实体，例如，<会变成&lt;，&变成&amp;，诸如此类。

**问：** 上个月我们公司聘请了一个Web顾问来审计我们的Web应用。她注意到我们到处都在使用EL输出用户录入的串。她说这里存在安全风险，建议我们使用c:out标记输出所有用户串。这是什么意思？

**答：** 你们的顾问说得没错。她所指的安全风险称为跨网站攻击（cross-site hacking）或跨网站脚本攻击（cross-site scripting）。这种攻击是指一个用户使用你的web应用作为传输机制向另一个用户的web浏览器发动攻击。



**问：** 如果EL表达式的值为null会怎么样呢？

**答：** 这个问题问得好。你知道的，EL表达式\${evalsToNull}会在响应输出中生成一个空串，<c:out value="\${evalsToNull}" />也是一样。不过，c:out还不只如此。c:out标记很聪明，它会识别出这个值为null，然后可以完成一个特殊的动作。这个动作会提供一个默认值……

字符	字符实体码
<	&lt;
>	&gt;
&	&amp;
'	&#039;
''	&#034;

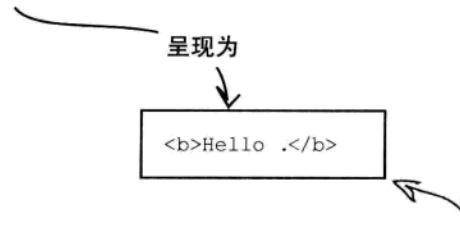
通过在user2的web浏览器中显示<script>标记和JS代码，能够发动跨网站攻击。而使用c:out标记来呈现用户的文本可以避免这种形式的攻击。这样能禁止浏览器解释这些JS代码，防范来自user1的攻击。

# Null值呈现为空文本

假设有一个欢迎页面可以为用户显示“Hello <user>”。但是最近用户没有登录，输出看上去就很奇怪了：

## 如果user为null, EL什么也不打印

```
<b>Hello ${user}.</b>
```



## 如果user为null, JSP表达式标记什么也不打印

```
<b>Hello <%= user %>.</b>
```

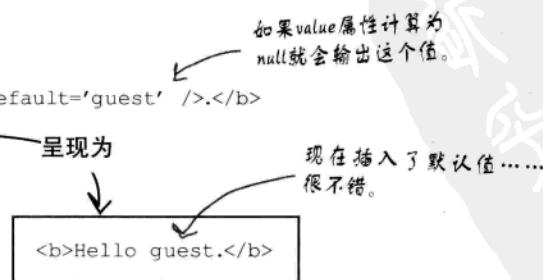


## 用default属性设置一个默认值

假设你希望向这些匿名用户显示这样一个消息：“Hello guest”。这里就非常适合在c:out标记中使用一个默认值。只需增加一个**default**属性，并提供表达式计算为null时你希望打印的值：

### c:out 提供一个default属性

```
<b>Hello <c:out value='${user}' default='guest' />.</b>
```



## 或者可以这样做：

```
<b>Hello <c:out value='${user}'>guest</c:out>.</b>
```

## 不用脚本就能实现循环

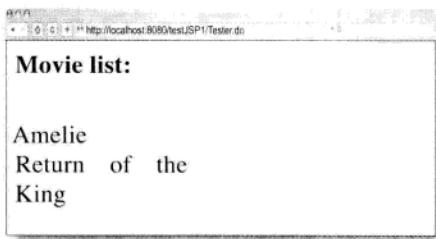
假设你想循环处理一个集合（比如说，一个目录项数组），一次取出一个元素，并在一个动态生成的表行中打印这个元素。硬编码写出整个表是不可能的，因为你不知道运行时会有多少行，而且你当然也不知道集合中有哪些值。解决这个问题的答案就是<c:forEach>标记。你确实需要对HTML表有所了解，不过即便你对这个内容不太熟悉，也不用着急，我们会做一个简单的说明。

顺便说一句，考试时你一定要知道如何结合表使用<c:forEach>。

### Servlet代码

```
...
String[] movieList = {"Amelie", "Return of the King", "Mean Girls"};
request.setAttribute("movieList", movieList);
...
建立电影名的一个String[]数组，并把这个数组设置为一个请求属性。
```

你想得到：



在JSP中，如果使用脚本

```
<table>
<% String[] items = (String[]) request.getAttribute("movieList");
   String var=null;
   for (int i = 0; i < items.length; i++) {
      var = items[i];
   }
%>
<tr><td><%= var %></td></tr>
<% } %>
</table>
```

## <c:forEach>

JSTL的<c:forEach>标记非常适合完成这个任务，它提供了一种简单的方法来迭代处理数组和集合。

(这一章后面还会讨论taglib指令。)

### JSP代码

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<strong> Movie list:</strong>
<br><br>

<table>
  <c:forEach var="movie" items="${movieList}" >
    <tr>
      <td>${movie}</td>
    </tr>
  </c:forEach>
</table>

</body></html>
```

循环处理整个数组（“movieList”属性），并分别在一个新行中打印各个元素（这个表中每行只有一列）。

### HTML表复习

<td>这个单元格的数据</td>	<td>这个单元格的数据</td>	<td>这个单元格的数据</td>
<td>这个单元格的数据</td>	<td>这个单元格的数据</td>	<td>这个单元格的数据</td>
<td>这个单元格的数据</td>	<td>这个单元格的数据</td>	<td>这个单元格的数据</td>

<tr> 代表表行 (Table Row)。  
 <td> 代表表数据 (Table Data)。

</table>

表相当简单。其中包括单元格，单元格组织为行和列，数据放在单元格中。关键是，要告诉表你想要多少行和多少列。

行用<tr>（表行）标记定义，列用<td>（表数据）标记定义。行数由<tr>标记的个数得来，列数由放在<tr></tr>标记中间的<td>标记的个数决定。

要打印/显示的数据只放在<td> </td>标记之间！

## <c:forEach>剖析

<c:forEach>标记完全可以对应于for循环，对于集合（在此如果用到“集合”一词，可以表示数组、Collection、Map或用逗号分隔的String）中的每个元素，都会重复执行一次标记的体。

关键是，这个标记将集合中的每个元素赋给一个变量，也就是你用var属性声明的那个变量。

<c:forEach>标记

这个变量存放集合中的各个元素。  
每次迭代时，这个变量值都会改变。

```
<c:forEach var="movie" items="${movieList}" >
    ${movie}
</c:forEach>
```

具体要循环处理的集合（数组、Collection、Map或者一个用逗号隔的String）。

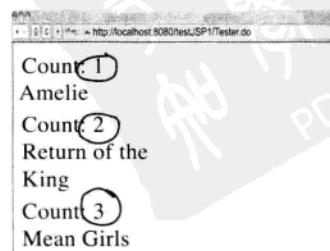
```
String[] items = (String[]) request.getAttribute("movieList");
for (int i = 0; i < items.length; i++) {
    String movie = items[i];
    out.println(movie);
}
```

用可选属性varStatus得到一个循环计数器

varStatus建立一个新的变量，其中保存javax.servlet.jsp.jstl.core. LoopTagStatus的一个实例。

```
<table>
    <c:forEach var="movie" items="${movieList}" varStatus="movieLoopCount" >
        <tr>
            <td>Count: ${movieLoopCount.count}</td>
        </tr>
        <tr>
            <td>${movie} <br><br></td>
        </tr>
    </c:forEach>
</table>
```

很有用，LoopTagStatus类有一个count属性，可以提供迭代计数器的当前值（这与for循环中的“i”很类似）。



# 甚至可以嵌套<c:forEach>标记

如果有一个集合的集合该怎么办？数组的数组呢？对于更复杂的表结构，可以嵌套<c:forEach>标记。在这个例子中，我们把String数组放入一个ArrayList，然后把这个ArrayList设置为一个请求属性。JSP必须循环处理ArrayList来得到每个String数组，然后循环处理各个String数组，打印出数组中的各个具体元素。

## Servlet代码

```
String[] movies1 = {"Matrix Revolutions", "Kill Bill", "Boondock Saints"};
String[] movies2 = {"Amelie", "Return of the King", "Mean Girls"};
java.util.List movieList = new java.util.ArrayList();
movieList.add(movies1);
movieList.add(movies2);
request.setAttribute("movies", movieList);
```

## JSP代码

```
<table>
    

来自第一个 String[] → { Matrix Revolutions  
Kill Bill  
Boondock Saints  
Amelie  
Return of the King  
Mean Girls }



来自第二个 String[] → { }


```

*there are no  
Dumb Questions*

**问：**你怎么知道“varStatus”属性是这样一个类的实例，你是怎么知道它有一个“count”性质的？

**答：**哈哈……我们查的呀。

这些都在JSTL 1.1规范里。如果你还没有这个规范，现在就应该去下载（本书的引子已经告诉你在哪里可以得到考试中涵盖的各个规范）。这是JSTL中所有标记的参考文档，指出了所有可能的属性，这些属性是可选的还是必要的，属性的类型，以及使用标记的其他有关细节。

关于这些标记你要知道的内容都将在本章谈到（对考试来说）。但是有些标记还有更多的选项，这一章没有全部介绍，所以你可能还要去查查规范。

**问：**你知道的不只这些吧……能不能告诉我，有没有办法改变迭代步长？例如，在实际的Java for循环中，我不一定非得写`i++`，而是可以用`i += 3`，这样就能每隔3个元素处理一次，而不是每个元素都要处理……

**答：**没问题。`<c:forEach>`标记有可选的begin和end属性（因为你可能想对集合的一个子集进行迭代处理），另外如果你想跳过某些元素，还提供了可选的step属性。

**问：**`<c:forEach>`中的“c”是一个必要的前缀吗？

**答：**嗯，当然必须有一个前缀；所有标记和EL函数都必须有一个前缀，它会为容器指定该标记或函数名的命名空间。不过，不一定非得使用前缀“c”。这只是一个标准约定，用来表示JSTL中一组所谓的“core”（核心）标记。不过，如果你不想把同事搞糊涂，建议你还是使用“c”前缀。



“var”变量的作用域仅限于标记内部！

没错，标记作用域。不过这不是一个完备的作用域，不能像其他4个作用域那样（页面作用域、请求作用域、会话作用域和应用作用域）绑定属性。标记作用域只表示变量在一个循环内部声明。

你已经知道在Java中这是什么意思。可以看到，对于大多数其他的标记，用“var”属性设置的变量只在你特别指定的作用域中可见（使用可选的“scope”属性来设置），或者，变量就默认认为在页面作用域中。

所以，如果看到以下代码，它想在`<c:forEach>`体标记的下面再使用变量，千万别被骗住了！

```
<c:forEach var="foo" items="${fooList}">
    ${foo} ← OK
</c:forEach>
```

`← 这样不行！！ “foo” 变量  
${foo} ← 已经出了作用域！`

可以把标记作用域想成是普通Java代码中的块作用域，这样可能会有帮助。下面是对你很熟悉的一个for循环例子：

```
for (int i = 0; i < items.length; i++) {
    x + i;
}
doSomething(i); ← 不对！！ “i” 变量已
                    经出了作用域！
```

## 使用<c:if>完成条件包含

假设你有一个页面，用户可以在这里查看其他用户给出的评论。另外，假设只有成员可以提交评论，而非成员的游客不能。所有人都得到同样的页面，但是成员在页面上能“看到”更多的东西。你希望有一个条件性的<jsp:include>，当然，你不想用脚本来做这个工作！

成员能看到的：

Member Comments

This site rocks.  
This site is cool.  
This site is stupid.

Add your comment:

Add Comment

非成员能看到的：

Member Comments

This site rocks.  
This site is cool.  
This site is stupid.

如果客户不是成员，我们不希望他们看到“Add...”部分。

### JSP代码

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<strong>Member Comments</strong> <br>
<hr>${commentList}<hr>
<c:if test="${userType eq 'member' }" >
  <jsp:include page="inputComments.jsp"/>
</c:if>
</body></html>
```

假设有这样一个servlet会根据用户的登录信息设置userType属性。

没错，‘member’两边是单引号。别忘了，标记和EL中单引号和双引号都能用。

### 所包含的页面(“inputComments.jsp”)

```
<form action="commentsProcess.jsp" method="post">
Add your comment: <br>
<textarea name="input" cols="40" rows="10"></textarea> <br>
<input name="commentSubmit" type="button" value="Add Comment">
</form>
```

## 不过，如果需要else怎么办？

如果你希望条件满足时做一件事，而条件不满足时做另一件事，怎么做呢？换句话说，你只想显示其中的一样，但任何人都不可能两样都看到，如何做到？前一页上的<c:if>能很好地工作，这是因为前面的逻辑是：每个人都能看到第一部分，如果测试条件为true，再显示一点额外的内容。

现在假想有这样一种情况：有一个汽车销售网站，你想根据先前在会话中建立的用户属性来定制每个页面上显示的标题新闻。不论用户是谁，页面中大部分内容都是一样的；但是每个用户会看到一个最贴近他个人购买意向的标题新闻（毕竟，我们的目的是让他买汽车，我们想挣钱）。会话刚开始时，显示一个表单，要求用户选择对他来说最重要的东西……

会话刚开始时：

When buying a car, what is most important to you?

Performance  
 Safety  
 Maintenance

Submit

后来会话中的某个时刻：

Now you can stop even if you do drive insanely fast.

**The Brakes**

Our advanced anti-lock brake system (ABS) is engineered to give you the ability to steer even as you're stopping. We have the best speed sensors of any car this size.

假设有家汽车公司的网站。第一个页面询问用户觉得什么最重要。

就像一个好的销售人员一样，展示汽车特性的页面会根据用户的喜好来定制，这样汽车的每个特性就好像是专门针对用户所需求量身订做的样子……

对用户的页面稍加定制，反映他最感兴趣的方面……

# 这种情况下<c:if>标记不适用

如果只使用<c:if>标记，这就没办法做到了，因为它没有一个“else”子句。下面这样好像可以：

使用<c:if>的JSP，但是这样做并不正确……

```
<c:if test="${userPref=='performance'}" >
    Now you can stop even if you <em>do</em> drive insanely fast...
</c:if>
<c:if test="${userPref=='safety'}" >
    Our brakes won't lock up no matter how bad a driver you are.
</c:if>
<c:if test="${userPref=='maintenance'}" >
    Lost your tech job? No problem--you won't have to service these brakes
    for at least three years.
</c:if>
```

但是如果userPref不满足上述所有条件会怎么样呢?  
没有办法指定默认的标题新闻吗?

--> 接下来是所有人都能看到的页面部分 -->

<c:if>是不行的，除非能肯定绝对不需要一个默认值。我们真正需要的是一种  
if/else构造（注1）：

使用脚本的JSP，它能得到我们期望的结果：

```
<html><body><h2>
<% String pref = (String) session.getAttribute("userPref");
if (pref.equals("performance")) {
    out.println("Now you can stop even if you <em>do</em> drive insanely fast.");
} else if (pref.equals("safety")) {
    out.println("Our brakes won't lock up, no matter how bad a driver you are. ");
} else if (pref.equals("maintenance")) {
    out.println(" Lost your tech job? No problem--you won't have to service these
    brakes for at least three years.");
} else {
    // userPref doesn't match those, so print the default headline
    out.println("Our brakes are the best.");
} %>
</h2><strong>The Brakes</strong> <br>
Our advanced anti-lock brake system (ABS) is engineered to give you the ability to steer
even as you're stopping. We have the
best speed sensors of any car this size. <br>
</body></html>
```

假设先前已经在会话中的  
某处设置了“userPref”。

注1：不错，我们同意你的想法，比起一  
连串的if测试，肯定还有更好的办法。  
不过，你先别急，先了解一下  
这是怎么做的……



## <c:choose>标记和它的“同伴” <c:when>和<c:otherwise>

<c:choose>

```
<c:when test="${userPref == 'performance'}">
```

Now you can stop even if you <em>do</em> drive insanely fast.

```
</c:when>
```

```
<c:when test="${userPref == 'safety'}">
```

Our brakes will never lock up, no matter how bad a driver you are.

```
</c:when>
```

```
<c:when test="${userPref == 'maintenance'}">
```

Lost your tech job? No problem--you won't have to service these brakes for at least three years.

```
</c:when>
```

```
<c:otherwise>
```

Our brakes are the best. ←

```
</c:otherwise>
```

```
</c:choose>
```

```
<!-- the rest of the page goes here... -->
```

如果这些<c:when>测试都不为true,  
<c:otherwise>就会作为默认选择运行。

注意：<c:choose>标记不一定有  
<c:otherwise>标记。

# <c:set>标记……比<jsp:setProperty>酷多了

<jsp:setProperty>标记只能做一件事，就是设置bean的性质。

但是，如果你想设置一个Map中的值呢？如果想在Map中创建新的一项呢？或者如果你只想创建一个新的请求作用域属性，该怎么做？

这些事情都可以用<c:set>来做到，但是你必须先学一些简单的规则。有两种不同的设置：var和target。var“版本”用于设置属性变量，target“版本”用于设置bean性质或Map值。这两个版本都有两种形式：有体或没有体。<c:set>体只是放入值的另一种途径。

## 用<c:set>设置属性变量var

### ① 没有体

如果没有一个名为“userLevel”的会话作用域属性，这个标记就会创建这样一个属性（假设value属性不为null）。

```
<c:set var="userLevel" scope="session" value="Cowboy" />
```

scope是可选的；var是必要的。必须指定一个值，不过既可以放入一个value属性，也可以把值放在标记体中（见以下的第2种形式）。

value不必为String……

如果\${person.dog}计算为一个Dog对象，那么“Fido”的类型是Dog。

### ② 有体

记住，标记有体时这里没有斜线。

```
<c:set var="userLevel" scope="session" >
    Sheriff, Bartender, Cowgirl
</c:set>
```

计算体，并用作为变量的值。



如果值计算为null，变量会被删除！真的，会把变量删除！

假设使用了\${person.dog}作为值（标记体中的值或者使用value属性设置的值）。如果\${person.dog}计算为null（这说明没有person，或者person的dog性质为null），那么倘若原来有一个名为“Fido”的变量属性，这个属性就会被删除！（如果你没有指定作用域，就会按顺序在页面作用域、请求作用域……中逐个查找）。即使“Fido”属性原来设置为一个String、Duck或Broccoli，也同样会被删除。

## 对bean和Map使用<c:set>

这一类<c:set>（包括它的两个变种：有体和没有体）只能用来设置两种东西：bean性质和Map值。仅此而已。不能用它向列表或数组中增加元素。用起来很简单，只需提供对象（bean或Map）、性质/键名以及值。

### 用<c:set>设置一个目标性质或值

#### ① 没有体

```
<c:set target="\${PetMap}" property="dogName" value="Clover" />
```

目标不能为null!!

如果目标是一个bean，就设置性质“dogName”的值。

如果目标是一个Map，则设置“dogName”键的相应值。

#### ② 有体

```
<c:set target="\${person}" property="name" value="\${foo.name}" />
```

不要把属性的“id”名放在这里！

没有斜线……考试时一定要当心这一点。

体可以是一个String或表达式。



“target”必须计算为一个对象！不能键入bean或Map属性的String  
“id”名！

这是一个重要的技巧。在<c:set>标记中，标记中的“target”属性看上去有些像<jsp:setProperty>中的“id”。尽管另一个版本的<c:set>中“var”属性取一个String直接量（表示作用域属性的名）。但是……“target”可不是这样！

“target”属性中不能键入表示属性名（属性以这个名字绑定到页面、作用域等）的String直接量，不是这样的，“target”属性需要的值要能解析为真正的对象。这说明，它需要一个EL表达式或一个脚本表达式(<%= %>)，或者是我们还没见过的<jsp:attribute>。

# <c:set>的要点和技巧

没错，<c:set>使用很容易，但是有一些方面需要记住……

- ▶ <c:set>中不能同时有“var”和“target”属性。
- ▶ “scope”是可选的，如果没有使用这个属性，则默认为页面（page）作用域。
- ▶ 如果“value”为null，“var”指定的属性将被删除！
- ▶ 如果“var”指定的属性不存在，则会创建一个属性，但仅当“value”不为null时才会创建新属性。
- ▶ 如果“target”表达式为null，容器会抛出一个异常。
- ▶ “target”中要放入一个能解析为实际对象的表达式。如果放入一个String直接量（表示bean或Map的“id”名），这是不行的。换句话说，“target”并非用来放bean或Map的属性名，而是用于指定具体的属性对象。
- ▶ 如果“target”表达式不是一个Map或bean，容器会抛出一个异常。
- ▶ 如果“target”表达式是一个bean，但是这个bean没有与“property”匹配的性质，容器就会抛出一个异常。记住EL表达式\${bean.notAProperty}也会抛出一个异常。

there are no  
Dumb Questions

**问：**为什么要使用有体的版本，而不是没有体的版本？看上去他们完全一样呀。

**答：**因为……它们做的是同样的事情。有体的版本只是为了提供方便，因为你有时可能希望有更大的地方来放值。例如，值可能是一个很长而且很复杂的表达式，如果把它放在体里，读起来会更容易一些。

**问：**如果没有指定作用域，是不是只会在页面作用域查找属性？还是会从页面作用域开始在所有作用域里做一次搜索？

**答：**如果标记中没有用可选属性“scope”，标记只会在页面作用域查找。很抱歉，你必须知道到底在哪个作用域。

**问：**为什么“属性”这个词有这么多的含义？它表示“放在标记里的内容”，还表示“与4个作用域之一中的对象相绑定的绑定对象”。这么一来，就要这么说了：标记有属性，它的值是页面的一个属性……这不是费解吗？

**答：**我懂你的意思。不过现在就是这么叫的。还是那句话，没有人问过我们。要是我们来取名，会把绑定对象叫做“绑定对象”。



## 可以用<c:remove>

Dick说得没错，使用一个set标记来删除一个变量，感觉是不太对劲（不过，要记住，只有传入一个null值时才会完成删除）。

<c:remove>标记很直观，也很简单：

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

    <c:set var="userStatus" scope="request" value="Brilliant" />
    userStatus: ${userStatus} <br>
    <c:remove var="userStatus" scope="request" />
    userStatus is now: ${userStatus}
</body></html>
```

*scope是可选的，但是如果没有指定作用域，就会从所有作用域删除这个属性。*

```
userStatus: Brilliant
userStatus is now:
```

*userStatus的值被删除，所以，删除之后使用这个EL表达式什么也不会打印出来。*



## 看看你对标记记得怎么样

如果你要参加考试，千万别跳过这个练习。答案在这一章的最后。

- ① 填入可选属性名。

```
<c:forEach var="movie" items="${movieList}"  ="foo" >
    ${movie}
</c:forEach>
```

- ② 填入缺少的属性名。

```
<c:if  ="${userPref=='safety'}" >
    Maybe you should just walk.....
</c:if>
```

- ③ 填入缺少的属性名。

```
<c:set var="userLevel" scope="session"  ="foo" />
```

- ④ 填入缺少的标记名（两种不同类型的标记），以及缺少的属性名。

```
<c:choose>
    <c:   ="${userPref == 'performance'}">
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c:  >

    <c:  >
        Our brakes are the best.
    </c:  >
</c:choose>
```

# 有了<c:import>, 现在有3种包含内容的方法了

到目前为止，我们曾经用过两种不同的方法将一个资源的内容增加到一个JSP中。不过此外还有一种方法，这就是使用JSTL。

## ① include指令

```
<%@ include file="Header.html" %>
```

静态：在转换时将file属性值指定的文件的内容增加到当前页面。

## ② <jsp:include>标准动作

```
<jsp:include page="Header.jsp" />
```

动态：在请求时将page属性值指定的内容增加到当前页面。

## ③ <c:import> JSTL标记

```
<c:import url="http://www.wickedlysmart.com/skyler/horse.html" />
```

动态：在请求时将URL属性值指定的内容增加到当前页面。它与<jsp:include>非常相似，但是更强大，也更灵活。

不同于另外两种包含机制，<c:import>url可以来自Web容器范围之外！

不要把<c:import>与page指令的“import”属性相混淆（前者是一种包含机制，后者将一个Java import语句放在生成的servlet中）。



它们的属性名都不一样！  
(当心“include” vs. “import”)

这三种包含机制都可以把一个资源的内容包含在JSP中，但它们的属性名各不相同。include指令使用file，<jsp:include>使用page，JSTL <c:import>标记使用url。这是有道理的，你可以想想看……不过确实要记得它们都记住。指令原本用于静态的布局模板，如HTML页眉。换句话说，它关心的是“文件”。<jsp:include>更关心来自JSP的动态内容，所以属性取名为“page”来反映这一点。<c:import>的属性正是根据你给它的内容来取名的，即URL！记住，前两种“包含”机制不能超出当前容器范围外，但<c:import>可以。

## <c:import>可以到Web应用之外

使用<jsp:include>或include指令，只能包含当前Web应用的页面。不过，有了<c:import>，还可以把容器之外的内容拿过来。这个简单的例子显示了服务器A上有一个JSP，它导入了服务器B上一个URL的内容。请求时，所导入文件的HTML块会增加到JSP中。所导入的块还使用了一个图像引用，这个图像也在服务器B上。

服务器A，完成导入的JSP

JSP

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

<c:import url="http://www.wickedlysmart.com/skyler/horse.html" />

<br>
This is my horse.

</body></html>
```



(不要忘了：与其他包含机制一样，你导入的应用是一个HTML片段，而不是一个有开始和结束<html><body>标记的完整页面。)

服务器B，所导入的内容

导入的文件

```

```



响应



“horse.html”和“horse.gif”都在服务器B上，而不是JSP所在的服务器。

## 定制包含的内容

在上一章中，应该记得，我们用了一个<jsp:include>来放入布局页眉（一个图片和一些文字），但我们想定制页眉中使用的子标题，我们是怎么做的？我们使用了一个<jsp:param>……

### ① 使用<jsp:include>的JSP

```
<html><body>

<jsp:include page="Header.jsp">

<jsp:param name="subTitle" value="We take the sting out of SOAP." />

</jsp:include>

<br>
<em>Welcome to our Web Services Support' Group.</em> <br><br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

### ② 所包含的文件(“Header.jsp”)

```
 <br>
<em><strong>${param.subTitle}</strong></em>
<br>
```



## 用<c:param>做同样的事情

下面完成前一页上同样的工作，不过这里会结合使用<c:import>和<c:param>。你会看到这个结构与使用标准动作的结构几乎是一样的。

### ① 使用<jsp:import>的JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<c:import url="Header.jsp" > 没有斜线，因为现在
                                     标记有一个体……
    <c:param name="subTitle" value="We take the sting out of SOAP." />
</c:import>

<br>
<em>Welcome to our Web Services Support Group.</em> <br><br>
Contact us at: ${initParam.mainEmail}

</body></html>
```

### ② 所包含的文件(“Header.jsp”)

```
 <br>
<em><strong>${param.subTitle}</strong></em>
<br>
```

这个页面没有任何变化。它并不关心参数是怎么来的，只要有参数就行。

不好意思，打断一下……  
不过我注意到JSP里有一个  
严重的问题！你怎么在不使用  
脚本的前提下确保JSP跟踪会  
话……？



JSP中会话跟踪是自  
动发生的。除非你用一个  
page指令明确地禁用会话跟  
踪，即把session属性指定为  
session=“false”。



他不明白我的意思……我是  
说“确保”。我真正想问的是，如  
果客户不支持cookie，我怎么让URL重  
写发生？我怎么在JSP中得到增加到  
URL的会话ID？



哈哈……显然他不知道  
有<c:url>标记。这个标  
记会自动完成URL重写。



## <c:url>可以满足所有超链接需求

回忆原来写servlet的日子，如果想使用一个会话是怎么做的？首先必须得到会话（可能是已有的会话，也可能是一个新会话）。此时，容器知道它要把发出这个请求的客户与一个特定的会话ID关联起来。容器想使用cookie，希望在响应中包含一个唯一的cookie，以后这个客户每次发出请求时就会把这个cookie再发回给容器。看上去很好，不过有一个问题……客户的浏览器可能会禁用cookie。如果是这样，该怎么办？

如果容器未能从客户得到一个cookie，它会自动地完成URL重写。但是如果使用servlet，你还必须对URL编码。换句话说，必须告诉容器，对于每个重要的URL，“要把这个jsessionid追加到特定URL的最后……”不错，在JSP中也可以做同样的事情，为此要使用<c:url>标记。

### servlet的URL重写

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();

    out.println("<html><body>");
    out.println("<a href=\"" + response.encodeURL("/BeerTest.do") + "\">click</a>");
    out.println("</body></html>");
```

↑                      ↑  
把额外的会话ID信息增加到这个URL。

### JSP的URL重写

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

This is a hyperlink with URL rewriting enabled.

```
<a href="<c:url value='/inputComments.jsp' />">Click here</a>
```

```
</body></html>
```

这会在“value”相对URL的最后增加  
jsessionid（如果禁用了cookie）。

## 如果URL需要编码怎么办？

还记得在HTTP GET请求中，参数会作为一个查询串追加到URL。例如，如果HTML页面上的一个表单有两个文本域（名和姓），请求URL就会把参数名和值追加到请求URL的最后。但是……如果HTTP请求中包含了不安全的字符，这个HTTP请求就无法工作（不过，大多数当前浏览器都在努力改善这一点）。

如果你是一个Web开发人员，这对你来说应该不是什么新闻了，但是如果你刚接触Web开发，就需要知道：URL通常都需要编码。URL编码的意思是，把不安全/保留的字符替换为其他字符，然后再在服务器端完成解码。例如，URL中不允许有空格，但是可以用一个加号“+”来代替空格。问题是，<c:url>不会自动地对URL编码！

### 对查询串使用<c:url>

记住，<c:url>标记所做的只是URL重写，而不是URL编码！

```
<c:set var="last" value="Hidden Cursor" />
<c:set var="first" value="Crouching Pixels"/>

<c:url value="/inputComments.jsp?first=${first}&last=${last}" var="inputURL" />
```

The URL using params is: \${inputURL} <br>

如果你以后想访问这个值，可以使用可选的“var”属性……



### 在<c:url>的体中使用<c:param>

这就能解决问题！现在完成了URL重写和URL编码。

```
<c:url value="/inputComments.jsp" var="inputURL" >
  <c:param name="firstName" value="${first}" />
  <c:param name="lastName" value="${last}" />
</c:url>
```

没有斜线

现在我们安全了，因为<c:param>会负责编码！

现在URL如下所示：

/myApp/inputComments.jsp?firstName=Crouching+Pixels&lastName=Hidden+Cursor

我要打断一会儿你们的  
JSTL谈话，我想说说错误处  
理。我们想做些事情，这些事情可  
能会导致一个异常……



## 你不希望客户看到：

Apache Tomcat/5.0.19 - Error report

http://localhost:8080/kathy/JSP1/ChooseTest.jsp

J2SE 1.5 in a Nutshell Colorado G...opers Home Slashdot: Ne...hat matters orkut - home java.net

**HTTP Status 500 -**

**Type** Exception report

**message**

**description** The server encountered an internal error () that prevented it from fulfilling this request.

**exception**

```
org.apache.jasper.JasperException: / by zero
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:358)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

**root cause**

```
java.lang.ArithmetricException: / by zero
	org.apache.jsp.ChooseTest_jsp._jspService(ChooseTest_jsp.java:62)
	org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:133)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:311)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

**note** The full stack trace of the root cause is available in the Tomcat logs.

Apache Tomcat/5.0.19

# 建立你自己的错误页面

访问你的网站的人并不想看到栈跟踪记录，而且他们也不想得到一个“404 Not Found”标准错误。

当然，错误是在所难免的，你无法保证一个错误也没有，不过，至少可以给用户一种更友好（更有吸引力）的错误响应页面。可以设计一个定制页面来处理错误，然后使用page指令配置错误页面。

## 指定的错误页面(“errorPage.jsp”)

```
<%@ page isErrorPage="true" %>

<html><body>
<strong>Bummer.</strong>

</body></html>
```

向容器做出确认，“不错，这是  
一个正式指定的错误页面。”

## 抛出异常的“坏”页面(“badPage.jsp”)

```
<%@ page errorPage="errorPage.jsp" %>

<html><body>
About to be bad...
<% int x = 10/0; %>
</body></html>
```

告诉容器，“如果出问题，就  
把请求转发到errorPage.jsp。”

请求“badPage.jsp”时会发生什么



请求的是“badPage.jsp”，但是  
该页面抛出一个异常，所以响应  
来自“errorPage.jsp”

我要花那么大功夫在所有JSP中  
放上page指令来指定要使用的错误  
页面。如果我想根据不同的错误使  
用不同的错误页面，该怎么办？如果有  
办法为整个Web应用指定错误页  
面就好了……



### 她不知道<error-page> DD标记。

可以在DD中为整个Web应用声明错误页面，甚至可以为不同的异常类型或HTTP错误码类型（404, 500等）配置不同的错误页面。

容器使用DD中的<error-page>配置作为默认错误页面，但是如果JSP有一个明确的errorPage page指令，容器就会优先使用指令。

# 在DD中配置错误页面

在DD中，可以根据`<exception-type>`或HTTP状态码`<error-code>`声明错误页面。采用这种做法，就可以根据产生错误的问题类型为客户显示不同的错误页面。

## 声明一个“普遍”型错误页面

它应用于Web应用中的所有部分，不只是JSP。可以在单个的JSP中增加一个包括`errorPage`属性的`page`指令覆盖这个设置。

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/errorPage.jsp</location>
</error-page>
```

## 为更明确的异常声明一个错误页面

以下配置了一个错误页面，只有出现`TArithmeticException`异常时才调用这个错误页面。如果既有这个声明，又有以上的“普遍”型声明，非`ArithmeticException`的其他异常都会导致调用“`errorPage.jsp`”。

```
<error-page>
  <exception-type>java.lang.ArithmaticException</exception-type>
  <location>/arithmeticError.jsp</location>
</error-page>
```

## 根据一个HTTP状态码声明错误页面

以下配置了一个错误页面，只有响应的状态码是“404”（文件未找到）时才调用这个错误页面。

```
<error-page>
  <error-code>404</error-code>
  <location>/notFoundError.jsp</location>
</error-page>
```

`<location>`必须相对于web-app根/上下文，这说明它必须以一个斜线开头（不论错误页面根据`<error-code>`声明还是根据`<exception-type>`来设置，都是如此）。

# 错误页面得到一个额外的对象：exception

错误页面实际上就是一个处理异常的JSP，所以容器为这个页面提供了一个额外的exception对象。你可能不想向用户显示异常，但是这个异常确实可以得到。在scriptlet中，可以使用隐式对象exception，在JSP中，则可以使用EL隐式对象\${pageContext.exception}。这个对象的类型是java.lang.Throwable，所以在脚本中可以调用隐式对象exception的方法，利用EL可以访问EL隐式对象exception的stackTrace和message性质。

一个更明确的错误页面(“errorPage.jsp”)

```
<%@ page isErrorPage="true" %>
```

```
<html><body>
<strong>Bummer.</strong><br>
```

```
You caused a ${pageContext.exception} on the server.<br>
```

```

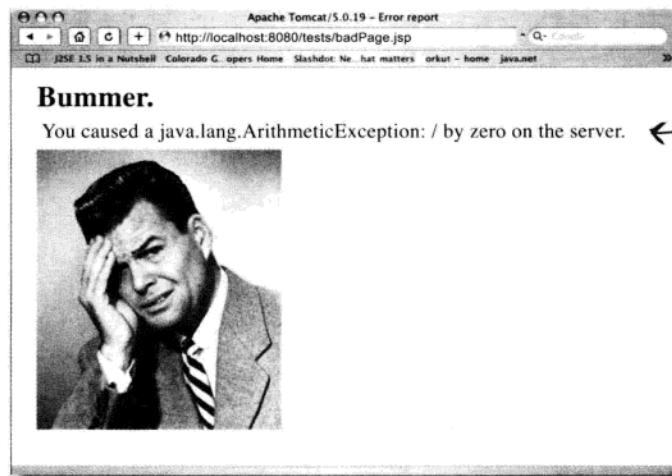
</body></html>
```

注意：exception隐式对象只对错误页面可用  
(有明确定义的page指令)：

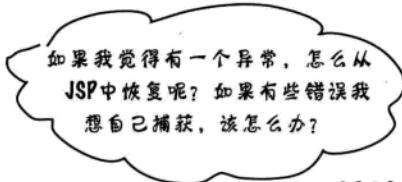
```
<%@ page isErrorPage="true" %>
```

换句话说，在DD中配置错误页面还不够，仅凭这一点容器并不会向页面提供exception隐式对象！

请求“badPage.jsp”时会发生什么



这一次会得到更多详细信息。你可能不想把这些内容显示给用户……我们这样做只是想让你看看。



## <c:catch>标记就像一种try/catch……

如果你的一个页面调用了一个有风险的标记，但是你认为能恢复，这是有办法的。可以使用<c:catch>标记完成一种try/catch，把有风险的标记或表达式包起来。因为倘若不这样，就会抛出异常，默认的错误处理机制就会插手，用户将看到DD中声明的错误页面。一个<c:catch>会同时作为try和catch部分，这听上去有点奇怪，确实没有单独的try标记。你要把有风险的EL或标记调用（或者其他部分）包在<c:catch>的体中，异常就会在这里被捕获。不过，不能把它当成一个catch块，因为一旦异常发生，控制就会跳至<c:catch>标记体的最后（稍后再做更多的介绍）。

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

About to do a risky thing: <br>

<c:catch>

<% int x = 10/0; %>

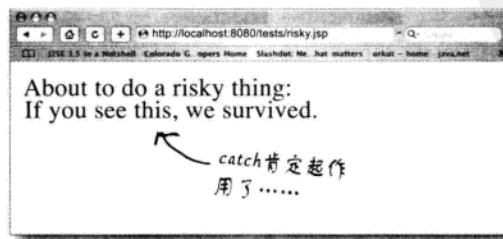
这个scriptlet肯定会导致一个异常……  
不过我们要捕获这个异常，而不是触发错误页面。

</c:catch>

If you see this, we survived.

如果打印出这个内容，就能知道已经通过了异常（在这个例子中，这说明我们成功地捕获了异常）。

</body></html>



但是，我怎么访问Exception对象呢？就是具体抛出的那个异常对象？因为这不是实际的错误页面，隐式对象exception在这里不起作用。



## 可以将异常置为一个属性

在一个实际的Java try/catch中，catch参数就是异常对象。但是对于Web应用错误处理，要记住，只有正式指定的错误页面才能得到异常对象。其他的页面得不到异常。所以下面这样做是不行的：

```
<c:catch>
    Inside the catch...
    <% int x = 10/0; %>
</c:catch>

Exception was: ${pageContext.exception}
```

无法工作，因为这不是一个正式的错误页面，所以它无法得到异常对象。



### 在<c:catch>中使用“var”属性

如果你想在<c:catch>标记结束后访问异常，可以使用可选的var属性。它会把异常对象放在页面作用域，并按你声明的var值来命名。

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

About to do a risky thing: <br>

<c:catch var="myException">

这会创建一个新的页面作用域属性，名为“myException”，并把异常对象赋给这个变量。

Inside the catch...

<% int x = 10/0; %>

</c:catch>

<c:if test="\${myException != null}">

    There was an exception: \${myException.message} <br>

</c:if>

We survived.

</body></html>

现在有一个属性myException，由于它是一个Throwable对象，所以有一个“message”性质（因为Throwable有getMessage()方法）。



<c:catch>中流控制的工作方式与try块是一样的，出现异常后，<c:catch>体中余下的部分不再运行。

在一个常规的Java try/catch中，一旦出现异常，try块中该点以下的代码就不会执行了，控制直接跳至catch块。对于<c:catch>标记，一旦出现异常，会发生两件事：

- 1) 如果使用了可选的“var”属性，会把异常对象赋给这个变量。
- 2) 流控制直接跳到<c:catch>标记体的下面。

```

<c:catch>
    Inside the catch...
    <% int x = 10/0; %>
    After the catch... ← 这些你是看不到的!
</c:catch>
We survived.
  
```

对此要当心。如果想使用“var”异常对象，必须等到到达<c:catch>体的最后。换句话说，在<c:catch>标记体中无法使用有关异常的任何信息。

一般会把<c:catch>标记想成是一个正常Java代码的catch块，但实际上并不是。<c:catch>更像是一个try块，因为你会把有风险的代码放在这里。只不过这是一个不需要（或者没有）catch或finally块的try。是不是搞糊涂了？重点是，要按这个标记本来的作用来学习，不要把它与你原先有关常规try/catch的了解相对照。在考试时，如果看到<c:catch>标记中在抛出异常的位置之后还有代码，别上当。

# 如果需要不在JSTL中的标记呢？

JSTL相当庞大。1.1版有5个库，其中4个是定制标记库，还有一个提供了大量用于串处理的函数。本书中介绍的标记（这也是考试所要求掌握的标记）都是你最需要的一些通用标记，不过，你需要的所有功能几乎都能在这5个库中找到。在下一页上，我们将介绍如果以下标记还不能满足要求该怎么办。

## “核心”库

- 通用
  - <c:out>
  - <c:set>
  - <c:remove>
  - <c:catch>
  - 条件
    - <c:if>
    - <c:choose>
    - <c:when>
    - <c:otherwise>
  - 与URL相关
    - <c:import>
    - <c:url>
    - <c:redirect>
  - <c:param>
  - 循环
    - <c:forEach>
    - <c:forTokens>

这个标记还没有介绍……给定分隔符时，用这个标记可以迭代处理 token。它的工作与 StringTokenizer 很类似。<c:redirect> 和 <c:out> 也没有介绍，不过这正好说明你需要有一个 JSTL 文档。

## “格式化”库

- 国际化
  - <fmt:message>
  - <fmt:setLocale>
  - <fmt:bundle>
  - <fmt:setBundle>
  - <fmt:param>
  - <fmt:requestEncoding>
- 格式化
  - <fmt:timeZone>
  - <fmt:setTimeZone>
  - <fmt:formatNumber>
  - <fmt:parseNumber>
  - <fmt:parseDate>
- “SQL”库
  - 数据库访问
    - <sql:query>
    - <sql:update>
    - <sql:setDataSource>
    - <sql:param>
    - <sql:dateParam>

## “XML”库

- 核心 XML 动作
  - <x:parse>
  - <x:out>
  - <x:set>
- XML 流控制
  - <x:if>
  - <x:choose>
  - <x:when>
  - <x:otherwise>
- 转换动作
  - <x:transform>
  - <x:param>



考试只考“核心”库。

考试中涵盖的 JSTL 库只是“核心”库（按惯例总是有一个前缀“c”）。其他库是专用的，所以我们不会深入介绍。不过，你至少应该知道有这些库可供使用。例如，如果你要处理 RSS 提要、XML 转换标记就有很大帮助。编写你自己的定制标记可能很困难，所以编写自己的标记之前先看一看有没有现成的标记可以直接利用，而不要自己完全重新开始。

## 使用非JSTL的标记库

创建标记的支持代码并不轻松（支持代码就是在JSP中放入标记时所要调用的Java代码）。我们会专门用一章（下一章）来介绍如何开发自己的定制标记处理器。但是这一章最后一部分将介绍如何使用定制标记。例如，如果有人为你的公司或项目创建了一个定制标记库，你怎么知道这些标记是什么，怎么使用这些标记？如果是JSTL，这很容易，JSTL 1.1规范明确规定了各个标记，包括如何使用每一个必要和可选的属性。

并不是所有定制标记都能如此妥善地打包，并有这么完备的文档。即使文档很少，或者根本没有文档，你也必须知道如何了解一个标记，还有一点，你必须知道如何部署一个定制标记库。

你要知道很多事情：

### ① 标记名和语法

显然，标记都有名字。对于标记<c:set>，标记名就是set，前缀是c。可以使用你喜欢的任何前缀，不过这个名字取自TLD。语法包括必要和可选的属性，标记是否有体（如果有体，可以在体中放什么），每个属性的类型，以及属性是否可以是一个表达式（或者只能是一个String直接量）。

### ② 库URI

URI是标记库描述文件（Tag Library Descriptor, TLD）的唯一标识符。换句话说，这是TLD描述的标记库的唯一名。taglib指令中就要放这个URI。它会告诉容器如何标识Web应用中的TLD文件，容器需要这个信息将JSP中使用的标记名映射到实际调用的Java代码。

要使用定制库，  
必须阅读TLD。

你想知道的一切  
都在这里。

# 理解TLD

TLD描述了两个主要内容：定制标记和EL函数。在前一章建立掷骰子函数时我们曾经用过一个TLD，但是那个TLD中只有一个<function>元素。现在我们来看<tag>元素，这要复杂一些。除了前面声明的函数外，以下DD描述了一个标记：advice。

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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-jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.2</tlib-version>           ← 必要（这是说标记是必要的，而不是非得取这个值）。
                                              ← 开发人员放这个标记来声明标记库的版本。

  <short-name>RandomTags</short-name>          ← 必要：主要由工具使用……
  <function>
    <name>rollIt</name>
    <function-class>foo.DiceRoller</function-class>
    <function-signature>int rollDice()</function-signature>      ← 上一章用到的EL函数。
  </function>

  <uri>randomThings</uri>           ← taglib指令中使用的唯一名！
                                              ← 可选，但是有这个标记确实很有好处……

  <tag>
    <description>random advice</description>
    <name>advice</name>           ← 必要！标记中就是要用这个名（例如：
                                  <my:advice>）。
    <tag-class>foo.AdvisorTagHandler</tag-class>      ← 必要！这样容器才知道有人在JSP中
                                              使用这个标记时要调用什么。
    <body-content>empty</body-content>           ← 必要！这说明标记的体中不能有任何内容。
    <attribute>                                ← 如果你的标记有属性，每个标记属性都需要有一个<attribute>元素。
      <name>user</name>           ← 这说明，标记中必须放一个“user”属性。
      <required>true</required>
      <rteprvalue>true</rteprvalue>           ← 这说明“user”属性可以是一个运行时表达式值（也就是说，不必非得是String直
                                              接量）。
    </attribute>

  </tag>
</taglib>

```

# 使用定制“advice”标记

“advice”标记是一个简单的标记，它有一个属性（用户名），并打印一个随机的建议。这非常简单，使用一个普通的EL函数（有一个静态方法 `getAdvice(String name)`）就完全可以做到，但是我们还是把它做成一个简单的标记，来说明它是如何工作的……

## 对应advice的TLD元素

```
<taglib ...>
...
<uri>randomThings</uri>
<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>

    <attribute>
        <name>user</name>
        <required>true</required>
        <rtpvalue>true</rtpvalue>
    </attribute>
</tag>
</taglib ...>
```

这正是你在前一页上看到的标记，不过这里没有加注解。

## 使用这个标记的JSP

```
<html><body>
<%@ taglib prefix="mine" uri="randomThings"%>
Advisor Page<br>
<mine:advice user="\${userName}" />
</body></html>
```

uri与TLD中的<uri>元素  
匹配。

在这里使用EL是可以的，因为TLD中user属性的<rtpvalue>设置为“true”（假设“userName”属性已经存在）。

TLD指出，标记不能有体，所以置为一个空标记  
(这说明，标记以一个斜线结束)。

页面中使用的每个库都需要有自己  
的taglib指令，而且要有唯一的  
前缀。

# 定制标记处理器

这个简单的标记处理器扩展了SimpleTagSupport（这个类下一章再介绍），而且实现了两个关键方法：doTag()和setUser()，doTag()是完成具体工作的方法，setUser()方法接受属性值。

## 完成标记工作的Java类

```

package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;

public class AdvisorTagHandler extends SimpleTagSupport {
    private String user;           // 容器会调用doTag()。
                                    // JSP 使用 TLD 中声明的名字调用标记时。
                                    // 容器调用这个方法将值设置为标记属性的值。它
                                    // 使用 JavaBean 属性命名约定得出应该向 setUser() 方
                                    // 法发送一个 "user" 属性。
                                    // 我们自己的内部方法。
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().write("Hello " + user + "<br>");
        getJspContext().getOut().write("Your advice is: " + getAdvice());
    }

    public void setUser(String user) {
        this.user = user;
    }

    String getAdvice() {
        String[] adviceStrings = {"That color's not working for you.",
            "You should call in sick.", "You might want to rethink that haircut."};
        int random = (int) (Math.random() * adviceStrings.length);
        return adviceStrings[random];
    }
}

```



定制标记处理器不使用定制的方法名！

采用EL函数时，要创建一个有静态方法的Java类，这个方法可以有任意的方法名，然后使用TLD将具体方法<function-signature>映射到函数<name>。不过，如果利用定制标记，方法名必须是doTag()，所以不能为定制标记声明方法名。只有函数使用TLD中的方法签名声明！

SimpleTagSupport 实现了定制标记所要完成的工作。

## 注意<rtexprvalue>

<rtexprvalue>非常重要，因为它会告诉你属性的值是在转换时计算，还是在运行时才计算。如果<rtexprvalue>为false，或者未定义<rtexprvalue>，那么属性值只能是一个String直接量！

如果你看到：

```
<attribute>
    <name>rate</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
</attribute>
```

或：

```
<attribute>
    <name>rate</name>
    <required>true</required> 如果没有<rtexprvalue>,
    <attribute>                                则默认值为false。
    </attribute>
```

就应该知道，这样是不行的！

```
<html><body>
    <%@ taglib prefix="my" uri="myTags"%>
    <my:handleIt rate="${currentRate}" /> 不行！不能是表达式……
</body></html>
```

问：还有一个问题你还没有回答呢，你怎么知道属性的类型呢？

答：我们先来看一个简单的。如果<rtexprvalue>为false（或根本没有<rtexprvalue>），那么属性类型只能是String直接量。但是，如果你使用了一个表达式，就只能看标记描述和属性名，从中得出类型，或者开发人员在<attribute>元素中包括了可选的<type>子元素，从<type>子元素也可以得出类型。<type>取一个完全限定类名作为类型。不论TLD是否声明了类型，容器都希望表达式的类型与标记处理器中该属性设置方法的实参类型匹配。换句话说，如果标记处理器中对于“dog”属性有一个setDog(Dog)方法，那么该属性的表达式值最好计算为一个Dog对象！（或者可以隐式地赋给Dog引用类型。）

# <rtexprvalue>不只是针对EL表达式

属性（或标记体）的值允许有运行时表达式时，可以使用3种表达式。

## ① EL表达式

```
<mine:advice user="${userName}" />
```

## ② 脚本表达式

```
<mine:advice user='<%= request.getAttribute("username") %>' />
```

↑  
必须是一个表达式，而不只是一个scriptlet。所以  
这里有“二”号，而且最后没有分号。

## ③ <jsp:attribute> 标准动作

```
<mine:advice>
  <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```

这是什么？：我以为这个标记没有体呢……



就算是TLD中标记体明确声明为“empty”，仍可以利用<jsp:attribute>在标记的体中放属性!!

<jsp:attribute>只是另一种为标记定义属性的方法。关键是，对于外部标记的每个属性，必须只有一个<jsp:attribute>。所以，如果标记中（而不是体中）有3个属性，在体中就会有3个<jsp:attribute>标记，每个属性都有一个相应的<jsp:attribute>标记。还要注意，<jsp:attribute>自己也有一个属性：name，可以在这里指定你要为外部标记的哪个属性设置值。

下一页还会谈这个问题……

## 标记体里能放什么

只有当标记的<body-content>元素值不是empty时，这个标记才能有体。

<body-content>元素的取值可以是以下3个或4个值之一，这取决于标记的类型。

<body-content>**empty**</body-content> 这个标记不能有体。

<body-content>**scriptless**</body-content> 这个标记不能有脚本元素（scriptlet、脚本表达式和声明），但是可以是模板文本和EL，还可以是定制和标准动作。

<body-content>**tagdependent**</body-content> 标记体要看作是纯文本，所以不会计算EL，也不会触发标记/动作。

<body-content>**JSP**</body-content> 能放在JSP中的东西都能放在这个标记体中。

### 对于没有体的标记，有3种调用方法

如果TLD中标记配置为<body-content>empty</body-content>，

采用以下三种方法来调用这个标记都是可以的。

#### ① 空标记

```
<mine:advice user="${userName}" />
```

在开始标记中放一个斜线，  
就不用使用结束标记了。

#### ② 开始和结束标记之间没有内容的标记

```
<mine:advice user="${userName}"></mine:advice>
```

有开始和结束标记，但二者之间没有任何内容。

#### ③ 在开始和结束标记之间只有<jsp:attribute>标记

```
<mine:advice>
    <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```

如果一个标记的<body-content>设置为empty，那么它的开始和结束标记之间只能放<jsp:attribute>标记！不过这只是一种放属性的方法，<jsp:attribute>标记并不算是“体内容”。

# 标记处理器、TLD和JSP

标记处理器开发人员要创建TLD，告诉容器和JSP开发人员如何使用这个标记。JSP开发人员并不关心TLD中的<tag-class>元素；这是容器要考虑的事情。JSP开发人员主要关心uri、标记名和标记语法。标记能有体吗？属性必须是一个String直接量吗？属性能是一个表达式吗？这个属性是可选的吗？表达式要计算为何种类型？

可以把TLD想成是定制标记的API。你必须知道如何调用定制标记，以及它需要什么参数。

要部署和运行一个使用了标记的Web应用，只需要这3部分——标记处理器类、TLD和JSP。

## 使用标记的JSP

```
<html><body>
<%@ taglib prefix="mine" uri="randomThings"%>
Advisor Page<br>
<mine:advice user="${userName}" />
</body></html>
```

## TLD文件

```
<taglib ...>
...
<uri>randomThings</uri>

<tag>
<description>random advice</description>
<name>advice</name>
<tag-class>foo.AdvisorTagHandler</tag-class>
<body-content>empty</body-content>
<attribute>
    <name>user</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
</attribute>
</tag>
```

## AdvisorTagHandler类

```
void doTag() {
    // tag logic
}

void setUser(String user) {
    this.user=user;
}
```

## taglib <uri>只是一个名，而不是一个位置

TLD中的<uri>元素是标记库的一个唯一名，仅此而已。它不需要表示任何具体的位置（例如，路径或URL）。只要求它是一个名，也就是taglib指令中使用的名字。

“但是”你可能会问，“那JSTL为什么要提供库的完整URL呢？”对应JSTL的taglib指令是：

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

这看上去很像一个Web资源的URL  
但实际上并不是。它只是一个名。  
只不过恰好格式化成URL的形式。

Web容器通常不会从taglib指令指定的uri请求结果。它不需要使用uri作为位置！如果你把它作为一个URL在浏览器中键入，就会重定向到另一个URL，其中包含有关JSTL的信息。这个特定的uri刚好也是一个合法URL（类似“http://……”的东西），但容器并不关心这一点。这只是Sun对uri采用的约定，以确保这是一个唯一名。Sun也可以把JSTL uri 取名为“java\_foo\_tags”，这同样是可以的。重要的是，TLD中的<uri>要与taglib指令中的uri匹配！

不过，作为一个开发人员，确实需要有一种机制能为你的库指定唯一的<uri>值，因为在给定的Web应用中，<uri>名必须是唯一的。例如，同一个Web应用中，不能有两个<uri>相同的TLD文件。所以，采用这种域名约定很不错，不过你的所有内部开发没有必要都采用这种约定。

前面已经说过了，在某种情况下uri也可以用作为一个位置，但是这种做法确实很糟糕。如果TLD中没有指定<uri>，容器会尝试将taglib指令中的uri属性用作为具体TLD的路径。但是，这样就要硬编码写出TLD的位置，这显然不是一个好的想法，所以尽管这样做是可以的，但最好假装你根本不知道这种做法。

容器会查找TLD中<uri>与  
taglib指令中uri值之间的匹配。  
uri不一定是具体标记处理器的  
位置！

# 容器建立一个映射

在JSP 2.0之前，开发人员必须为TLD中的<uri>与TLD文件的具体位置之间指定一个映射。所以，如果JSP页面有如下的一个taglib指令：

```
<%@ taglib prefix="mine" uri="randomThings"%>
```

部署描述文件(web.xml)必须告诉容器到哪里去找有匹配<uri>的TLD文件。为此，要在DD中使用一个<taglib>元素。

## 原先(JSP 2.0之前)将taglib uri映射到TLD文件的方法

```
<web-app>
...
<jsp-config>
  <taglib>
    <taglib-uri>randomThings</taglib-uri>
    <taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
  </taglib>
</jsp-config>
</web-app>
```

在DD中，把TLD中的<uri>映射到TLD文件的具体路径。

## 将taglib uri映射到TLD文件的新(JSP 2.0)方法

DD中没有<taglib>项！

容器会自动建立TLD和<uri>名之间的映射，所以JSP调用一个标记时，容器就能知道在哪里找到描述这个标记的TLD。

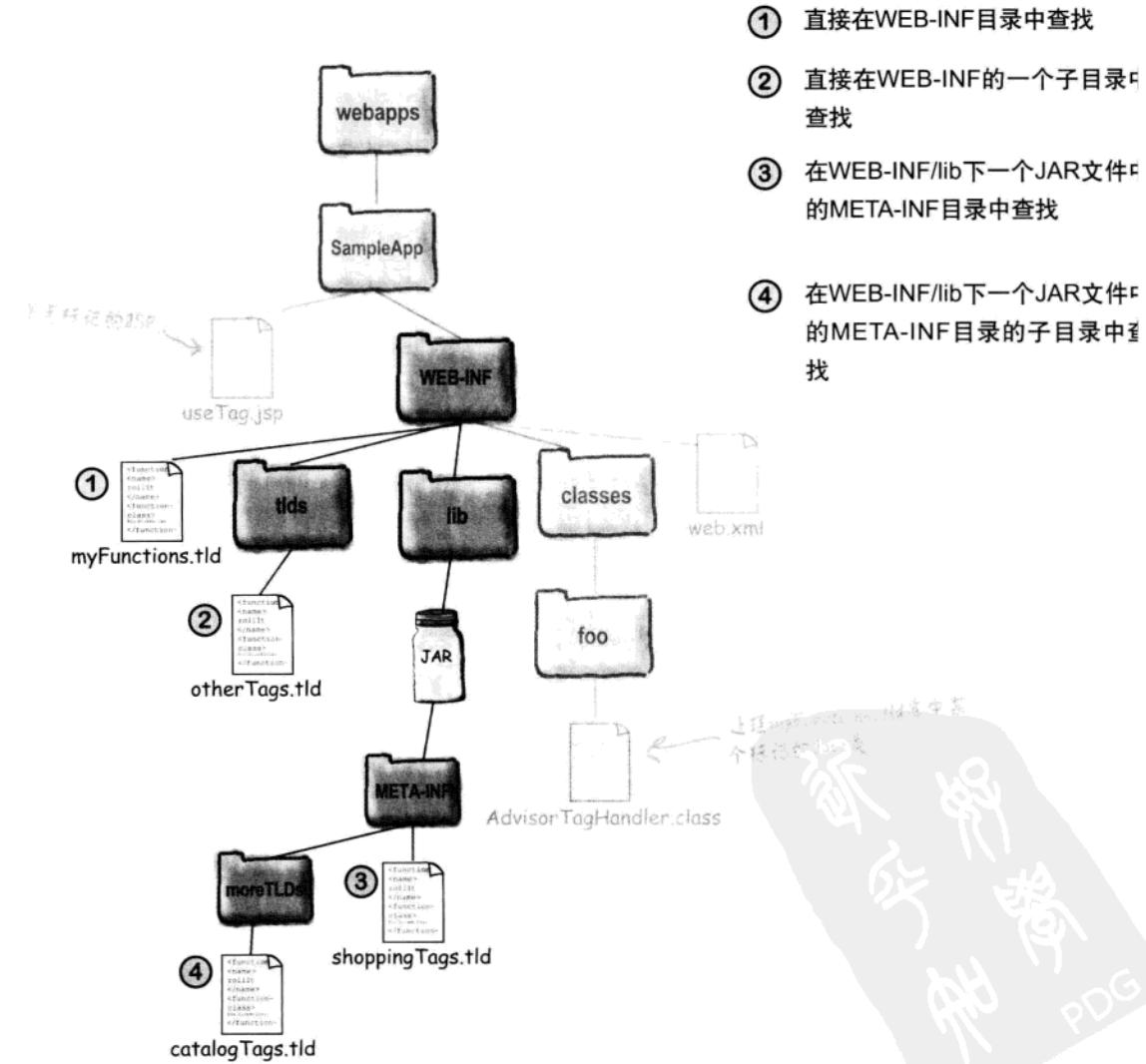
这是怎么做到的？答案是：在能放TLD的所有特定位置上找一遍。部署一个Web应用时，只要把TLD放在容器会搜索的位置上，容器就会发现这个TLD，并为标记库建立一个映射。

如果在DD(web.xml)中明确指定了一个<taglib-location>，JSP 2.0容器就会使用这个位置！实际上，容器开始建立<uri>-TLD映射时，会首先在DD中查找，看是否已经有一些<taglib>项，如果确实有，就会使用这些设置来帮助建立映射。如果你要参加考试，应该了解<taglib-location>，尽管这在JSP 2.0中已经不再必要。

所以我们下一步来看容器会在哪里查找TLD，另外在哪里查找TLD中声明的标记处理器类。

# 容器会在4个位置查找TLD

容器会在很多位置上查找TLD文件，你只需确保TLD放在正确的位  
置上，除此以外，什么也不用做。



# 如果JSP使用了多个标记库

如果你希望在一个JSP中使用多个标记库，每个TLD要有一个单独的taglib指令。有几点要记住……

- ▶ 确保taglib uri名是唯一的。换句话说，多个指令不能有相同的uri值。
- ▶ 不要使用保留的前缀。

保留前缀包括：

jsp:  
jspx:  
java:  
javax:  
servlet:  
sun:  
sunw:



*Sharpen your pencil*

空标记

体必须为空的标记可以有三种调用方法，请分别写出用这三种方法调用标记的例子。

(可以回头看看这一章的内容来检查你的答案。我们可不会告诉你要看哪一页)

① \_\_\_\_\_

② \_\_\_\_\_

③ \_\_\_\_\_



## JSP、TLD和bean属性类的关系

根据你在TLD中看到的信息填空。画出箭头，指出各部分信息如何关联。换句话说，对于每个空，指出填这个空需要哪些信息。

### 使用标记的JSP

```
<html><body>
<%@ taglib prefix="mine" uri="_____"%>
Advisor Page<br>
< _____: _____ = "${foo}" />
</body></html>
```

### AdvisorTagHandler类

```
void doTag() {
    // 标记逻辑
}

void set _____ (String x) {
    // 代码放在这里
}
```

### TLD文件

```
<taglib...>
...
<uri>randomThings</uri>

<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rteprvalue> _____ </rteprvalue>
    </attribute>
</tag>
```



## 看看你对标记记得怎么样 答案

### ① 填入可选属性名。

```
<c:forEach var="movie" items="${movieList}" varStatus = "foo" >
    ${movie}
</c:forEach>
```

根据这个属性对循环计数器变量命名。

### ② 填入缺少的属性名。

```
<c:if test = "${userPref == 'safety'}" >
    Maybe you should just walk...
</c:if>
```

`<c:set>` 标记必须有值，但是可以不作为一个属性，而是把值放在标记体中。

### ③ 填入缺少的属性名。

```
<c:set var="userLevel" scope="session" value = "foo" />
```

### ④ 填入缺少的标记名（两种不同类型的标记），以及缺少的属性名。

```
<c:choose>
    <c: when test = "${userPref == 'performance'}" >
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c: when >

    <c: otherwise >
        Our brakes are the best.
    </c: otherwise >
</c:choose>
```

`<c:otherwise>` 标记是可选的。



## JSP、TLD和bean属性类的关系 答案

### 使用标记的JSP

```
<html><body>
<%@ taglib prefix="mine" uri="randomThings"%>
Advisor Page<br>
<mie:advice user="${foo}" />
</body></html>
```

### AdvisorTagHandler类

```
void doTag() {
    // 标记逻辑
}

void setUser(String user) {
    this.user=user;
}
```

### TLD文件

```
<taglib ...>
...
<uri>randomThings</uri>

<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rteprvalueprvalue

```



## 第9章 模拟测验

1 关于TLD文件，哪个说法是正确的？

- A. TLD文件可以放在**WEB-INF**的任何子目录下。
- B. TLD文件用于配置JSP环境属性，如**scripting-invalid**。
- C. TLD文件可以放在WAR文件的**META-INF**目录中。
- D. TLD文件可以声明简单（Simple）和传统（Classic）标记，但是TLD文件不能用来声明标记文件。

2 假设使用标准JSTL前缀约定，可以用哪些JSTL标记来迭代处理一个对象集合？  
(选出所有正确的答案)

- A. <x:forEach>
- B. <c:iterate>
- C. <c:forEach>
- D. <c:forTokens>
- E. <logic:iterate>
- F. <logic:forEach>

**3** JSP页面包含一个**taglib**指令，其**uri**属性值为**myTags**。哪个部署描述文件元素定义了相关的TLD?

- A. 

```
<taglib>
    <uri>myTags</uri>
    <location>/WEB-INF/myTags.tld</location>
</taglib>
```
- B. 

```
<taglib>
    <uri>myTags</uri>
    <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- C. 

```
<taglib>
    <tld-uri>myTags</tld-uri>
    <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- D. 

```
<taglib>
    <taglib-uri>myTags</taglib-uri>
    <taglib-location>/WEB-INF/myTags.tld</taglib-location>
</taglib>
```

**4** 一个JavaBean **Person**有一个名为**address**的性质。这个性质的值是另一个JavaBean Address，它有以下串性质：**street1**, **street2**, **city**, **stateCode**和**zipCode**。一个控制器servlet创建了一个会话作用域属性，名为**customer**，这是**Person** bean的一个实例。

哪些JSP代码结构会把**customer**属性的**city**性质设置为**city**请求参数？（选出所有正确的答案）

- A. 

```
 ${sessionScope.customer.address.city = param.city}
```
- B. 

```
<c:set target="${sessionScope.customer.address}"
      property="city" value="${param.city}" />
```
- C. 

```
<c:set scope="session" var="${customer.address}"
      property="city" value="${param.city}" />
```
- D. 

```
<c:set target="${sessionScope.customer.address}"
      property="city">
      ${param.city}
</c:set>
```

**5**

对于以下JSP片段，TLD中哪些<body-content>元素组合是合法的？

(选出所有正确的答案)

```
11. <my:tag1>
12.   <my:tag2 a="47" />
13.   <% a = 420; %>
14.   <my:tag3>
15.     value = ${a}
16.   </my:tag3>
17. </my:tag1>
```

- A. tag1 body-content是empty  
tag2 body-content是JSP  
tag3 body-content是scriptless
- B. tag1 body-content是JSP  
tag2 body-content是empty  
tag3 body-content是scriptless
- C. tag1 body-content是JSP  
tag2 body-content是JSP  
tag3 body-content是JSP
- D. tag1 body-content是scriptless  
tag2 body-content是JSP  
tag3 body-content是JSP
- E. tag1 body-content是JSP  
tag2 body-content是scriptless  
tag3 body-content是scriptless

**6**

假设有适当的taglib指令，哪些是正确使用定制标记的例子？

(选出所有正确的答案)

- A. <foo:bar />
- B. <my:tag></my:tag>
- C. <mytag value="x" />
- D. <c:out value="x" />
- E. <jsp:setProperty name="a" property="b" value="c" />

7

给定以下scriptlet代码：

```

11. <select name='styleId'>
12. <% BeerStyle[] styles = beerService.getStyles();%
13.   for ( int i=0; i < styles.length; i++ ) {
14.     BeerStyle style = styles[i]; %>
15.     <option value='<%= style.getObjectID() %>'>
16.       <%= style.getTitle() %>
17.     </option>
18.   <% } %>
19. </select>
```

哪个JSTL代码段会生成同样的结果？

- A. <select name='styleId'>  
    <c:for array='\${beerService.styles}'>  
        <option value='\${item.objectID}'>\${item.title}</option>  
    </c:for>  
  </select>
- B. <select name='styleId'>  
    <c:forEach var='style' items='\${beerService.styles}'>  
        <option value='\${style.objectID}'>\${style.title}</option>  
    </c:forEach>  
  </select>
- C. <select name='styleId'>  
    <c:for var='style' array='\${beerService.styles}'>  
        <option value='\${style.objectID}'>\${style.title}</option>  
    </c:for>  
  </select>
- D. <select name='styleId'>  
    <c:forEach var='style' array='\${beerService.styles}'>  
        <option value='\${style.objectID}'>\${style.title}</option>  
    </c:for>  
  </select>



## 第9章 模拟测验答案

1 关于TLD文件，哪个说法是正确的？

(JSP v2.0  
3~16, 1~160页)

- A. TLD文件可以放在WEB-INF的任何子目录下。 B不对，因为TLD文件配置的是标记处理器，而不是JSP环境。
- B. TLD文件用于配置JSP环境属性，如scripting-invalid。 C不对，因为如果TLD文件放在WAR文件的META-INF中，将无法找到。
- C. TLD文件可以放在WAR文件的META-INF目录中。 D不对，因为在TLD中可以声明标记文件（但这种情况很少）。
- D. TLD文件可以声明简单（Simple）和传统（Classic）标记，但是TLD文件不能用来声明标记文件。

2 假设使用标准JSTL前缀约定，可以用哪些JSTL标记来迭代处理一个对象集  
(JSTL v1.1 42页)

- A. <x:forEach> A不对，因为这是迭代处理XPath表达式所用的标记。
- B. <c:iterate> B不对，因为不存在这样的标记。
- C. <c:forEach>
- D. <c:forTokens> D不对，因为这个标记只用于迭代处理一个串中的tokens。
- E. <logic:iterate> E和F不对，因为前缀‘logic’不是一个标准JSTL前缀（Jakarta Struts包中的标记通常用这个前缀）。
- F. <logic:forEach>

3

JSP页面包含一个taglib指令，其uri属性值为myTags。哪个部署描述文件元素  
定义了相关的TLD?

(JSP v2.0 3~12,13页)

- A. <taglib>  
 <uri>myTags</uri>  
 <location>/WEB-INF/myTags.tld</location>  
 </taglib>
- B. <taglib>  
 <uri>myTags</uri>  
 <tld-location>/WEB-INF/myTags.tld</tld-location>  
 </taglib>
- C. <taglib>  
 <tld-uri>myTags</tld-uri>  
 <tld-location>/WEB-INF/myTags.tld</tld-location>  
 </taglib>
- D. <taglib>  
 <taglib-uri>myTags</taglib-uri>  
 <taglib-location>/WEB-INF/myTags.tld</taglib-location>  
 </taglib>

D指定了合法的  
标记元素。

4

一个JavaBean Person有一个名为address的性质。这个性质的值是另一个  
JavaBean Address, 它有以下串性质: street1, street2, city, state-  
Code和zipCode。一个控制器servlet创建了一个会话作用域属性, 名为cus-  
tomer, 这是Person bean的一个实例。

(JSTL v1.1 4~28页)

哪些JSP代码结构会把customer属性的city性质设置为city请求参数? (选  
出所有正确的答案)

- A. \${sessionScope.customer.address.city = param.city}
- B. <c:set target="\${sessionScope.customer.address}"  
 property="city" value="\${param.city}" />
- C. <c:set scope="session" var="\${customer.address}"  
 property="city" value="\${param.city}" />
- D. <c:set target="\${sessionScope.customer.address}"  
 property="city"  
 \${param.city}  
 </c:set>

A不对, 因为  
EL不允许赋值。C不对, 因为var属性不  
接受运行时值, 而且不  
能同时用property属性。

**5**

对于以下JSP片段，TLD中哪些<body-content>元素组合是合法的?  
(选出所有正确的答案)。

```
11. <my:tag1>
12.   <my:tag2 a="47" />
13.   <% a = 420; %>
14.   <my:tag3>
15.     value = ${a}
16.   </my:tag3>
17. </my:tag1>
```

(JSP v2.0附录JSP.C  
特别是3~21和3~30页)

Tag1包括脚本代码，所以至少有‘JSP’ body-content。Tag2只显示为一个空标记，但是还可以包含‘JSP’或‘scriptless’ body-content。Tag3没有脚本代码，所以可以是‘JSP’或‘scriptless’ body-content。

- A. tag1 body-content是empty  
tag2 body-content是JSP  
tag3 body-content是scriptless
- B. tag1 body-content是JSP  
tag2 body-content是empty  
tag3 body-content是scriptless
- C. tag1 body-content是JSP  
tag2 body-content是JSP  
tag3 body-content是JSP
- D. tag1 body-content是scriptless  
tag2 body-content是JSP  
tag3 body-content是JSP
- E. tag1 body-content是JSP  
tag2 body-content是scriptless  
tag3 body-content是scriptless

A不对，因为tag1不能为‘empty’。

D不对，因为tag1不能为‘scriptless’。

**6**

假设有适当的taglib指令，哪些是正确使用定制标记的例子?  
(选出所有正确的答案)

- A. <foo:bar />
- B. <my:tag></my:tag>
- C. <mytag value="x" /> C不对，因为没有前缀。
- D. <c:out value="x" />
- E. <jsp:setProperty name="a" property="b" value="c" />

(JSP v2.0 第7节)

E不对，因为这是JSP标准动作的一个例子，而不是定制标记的例子。

7

给定以下scriptlet代码:

(JSTL v1.1 6~48页)

```

11. <select name='styleId'>
12. <% BeerStyle[] styles = beerService.getStyles();
13.     for ( int i=0; i < styles.length; i++ ) {
14.         BeerStyle style = styles[i]; %>
15.         <option value='<%= style.getObjectID() %>'>
16.             <%= style.getTitle() %>
17.         </option>
18.     <% } %>
19. </select>
```

哪个JSTL代码段会生成同样的结果?

- A. <select name='styleId'>  
     <c:for array='\${beerService.styles}'>  
         <option value='\${item.objectID}'>\${item.title}</option>  
     </c:for>  
   </select>
- B. <select name='styleId'>  
     <c:forEach var='style' items='\${beerService.styles}'>  
         <option value='\${style.objectID}'>\${style.title}</option>  
     </c:forEach>  
 </select>
- C. <select name='styleId'>  
     <c:for var='style' array='\${beerService.styles}'>  
         <option value='\${style.objectID}'>\${style.title}</option>  
     </c:for>  
 </select>
- D. <select name='styleId'>  
     <c:forEach var='style' array='\${beerService.styles}'>  
         <option value='\${style.objectID}'>\${style.title}</option>  
     </c:forEach>  
 </select>

B是对的，因为它使用了正确的JSTL标记/属性名。

# JSTL也有力不能及的时候……



有时JSTL和标准动作还不够。你需要些定制的东西，但又不想走老路写脚本，那你可以编写自己的标记处理器。这样一来，页面设计人员就能在他们的页面中使用你的标记，所有艰苦的工作都由标记处理器类在后台完成。不过，构建自己的标记处理器有3种不同的方法，所以要学的还很多。在这3种方法中，有两种（简单标记和标记文件）是在JSP 2.0新引入的，它们能让你的日子更好过。不过，在很少见的一些情况下，如果这两种方法还不能满足你的需要，你就得学学传统标记。定制标记开发能赋予你无限的能力，但是你得先学会驾驭它……

# OBJECTIVES

## 建立定制标记库

- 10.1 描述执行各个事件方法（`doStartTag()`、`doAfterBody()`和`doEndTag()`）时“传统”定制标记事件模型的语义；解释各事件方法返回值的含义，并编写标记处理器类。
  
- 10.2 使用`PageContext API`，编写标记处理器来访问JSP隐式变量以及Web应用属性。
  
- 10.3 给定一个场景，编写标记处理器代码来访问父标记和任意的祖先标记。
  
- 10.4 描述执行事件方法（`doTag()`）时“简单”定制标记事件模型的语义；编写一个标记处理器类；并解释标记中对JSP内容的限制。
  
- 10.5 描述标记文件模型的语义；描述标记文件的Web应用结构；编写一个标记文件；解释标记体中对JSP内容的限制。

## 内容说明：

尽管大纲10.1并没有明确提到`BodyTag`相关的生命周期方法（`doInitBody()`和`setBodyContext()`），但考试中可能会考到这些内容！关于传统标记，需要你掌握的所有知识都会在这一章中介绍，其中有些内容在10.1中并没有明确要求。

大纲10.2（`PageContext API`）在这一章中只会简要介绍，因为关于`PageContext API`需要知道的大部分内容在本书前面都已经讨论过。这个要求几乎都有关使用`PageContext`来访问隐式变量和作用域属性，这些内容在“无脚本的JSP”一章已经涵盖，不过这一章会用一页对这个内容再做一个总结。





我很赞赏可重用块的想法，但是<jsp:include>和<c:import>并不完美。要把被包含的文件放在哪个目录中，这一点没有标准可循，JSP读起来很困难，而且你要建立新的请求参数向被包含的文件发送信息，感觉好像不太对劲……

## Include和import可能很成问题

通过使用<jsp:include>或<c:import>，可以动态地向页面增加可重用的内容块，甚至可以设置新的请求参数，提供给被包含文件使用，从而能定制被包含文件的行为。

确实，这样是完全可以的。但是，仅仅为了向被包含文件提供一些定制信息，真的有必要创建新的请求参数吗？

请求参数的本来目的应该是表示表单数据，这些数据要作为请求的一部分从客户发送出来，这样才对吧？当然，在应用中，有时确实很有必要增加或修改请求参数，但是用请求参数向被包含文件发送信息的做法并不是最便利的方法。

在JSP 2.0之前，部署被包含文件没有标准可依，可以把所包含的部分放在Web应用的任何位置。如果JSP中有一大堆<jsp:include>或<c:import>标记，这样的JSP读起来很困难。倘若标记本身就能表明你要包含的是什么，不是更好吗？如果能像下面这样该多好：

<x:logoHeader>或<x:navBar>

这样你就能知道它会放在哪里……

# 标记文件：很像include，但是比include更好

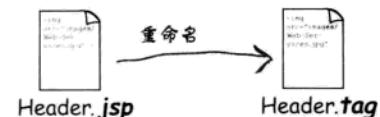
利用标记文件，可以使用一个定制标记调用可重用的内容，而不是使用通用的<jsp:include>或<c:import>。可以把标记文件看作是一种“轻型标记处理器”，这是因为页面开发人员能利用标记文件创建定制标记，而不用编写复杂的Java标记处理器类，但是标记文件只是“美化了”的include。

## 建立和使用标记文件的最简方法

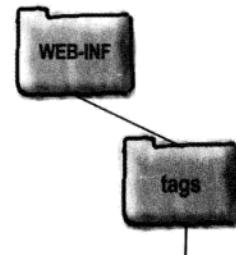
- ① 取一个被包含文件（如“Header.jsp”），把它重命名为有一个.tag扩展名。

```
 <br>
```

↑ 这就是整个文件……记住，我们去掉了开始和结束<html>和<body>标记，以免这些标记在最后的JSP中重复出现。



- ② 把标记文件(“Header.tag”)放在“WEB-INF”目录下一个名为“tags”的目录中。



- ③ 在JSP中放一个taglib指令（有一个tagdir属性），并调用这个标记。

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
```

```
<html><body>
```

```
<myTags:Header/>
```

↑ 标记名就是标记文件名！（但  
是要去掉.tag扩展名）

```
Welcome to our site.
```

```
</body></html>
```

↑ taglib指令中使用“tagdir”属性，而不是指定标记库TLD时所用的“uri”。

所以不是：

```
<jsp:include page="Header.jsp" />
```

而是：

```
<myTags:Header/>
```

# 但是怎么向它发送参数呢？

使用`<jsp:include>`包含一个文件时，可以在`<jsp:include>`中使用`<jsp:param>`标记向被包含文件传递信息。下面复习一下如何使用`<jsp:include>`：

**老办法：被包含文件里使用了param (来自调用JSP中的`<jsp:param>`)**

```
 <br>
<em><strong>${param.subTitle}</strong></em>
```

重申一句，这就是完整的被包含文件，而不是一个片段。

**老办法：JSP中包含`<jsp:include>`和`<jsp:param>`**

```
<html><body>

<jsp:include page="Header.jsp">
  <jsp:param name="subTitle" value="We take the sting out of SOAP." />
</jsp:include>

<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

设置一个新的请求参数，就像其他请求参数一样可由被包含页面使用。

## 结果



# 对于标记文件，发送的不是请求参数，而是标记属性！

你要用一个标记来调用标记文件，而且标记可以有属性。很自然的，标记文件开发人员可能想调用有属性的标记……所以可以把属性发送给标记文件。

## 从JSP调用标记

以前的做法 (使用<jsp:param>设置一个请求参数)

```
<jsp:include page="Header.jsp">
  <jsp:param name="subTitle" value="We take the sting out of SOAP." />
```

现在的做法 (使用有属性的标记)

```
<myTags:Header subTitle="We take the String out of SOAP" />
```

## 在标记文件中使用属性

以前的做法 (使用请求参数值)

```
<em><strong>${param.subTitle}</strong></em>
```

现在的做法 (使用标记文件属性)

```
<em><strong>${subTitle}</strong></em> <br>
```

这些代码放在具体的标记文件中 (也就是被包含的文件)。



所有标记属性都只有标记作用域。没错，只在标记内部有意义。一旦标记结束，标记属性就会出作用域！

你必须搞清楚——<jsp:include>的<jsp:param> 值会作为请求参数。这与请求作用域属性是不同的。记住，对于Web应用来说，<jsp:param> 的名/值对好像是从表单提交得来的一样。我们之所以不喜欢使用<jsp:param>，这也是原因之一，你本来只想把值传给所包含的文件，结果却是：Web应用中参与这个请求的所有组件（如请求转发到的servlet或JSP）都能看到这个值。

但标记文件的标记属性就不存在这个问题，它们只有标记作用域。你一定要了解由此带来的影响。下面这样是不行的：

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
<myTags:Header subTitle="We take the String out of SOAP" />
<br>
${subTitle}
</body></html>
```

这样不行！属性已经出了作用域。

请等等……好像少点东西。  
编写JSP的人怎么知道标记有  
那个属性呢？描述属性类型的  
TLD在哪里？



## 标记属性不在TLD中 声明吗？

如果是定制标记（包括JSTL），标记属性都在TLD中定义，还记得吧？以下是上一章定制<my:advice>标记的TLD：

```

<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>

    <attribute>
        <name>user</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
</tag>
```

所以，如果开发人员要使用标记，他只需要知道这些就够了：属性名是什么，属性是可选的还是必要的？可以是表达式吗？或者必须是一个String直接量吗？

尽管定制标记属性确实要在TLD中指定，但标记文件属性并不在TLD中指定！

这说明我们还要回答一个问题，页面开发人员怎么知道标记接受和/或需要哪些属性？请翻开下一页……

## 标记文件使用attribute指令

还有一个“很炫”的新指令，这个指令只能由标记文件使用。别人都不能用。它与定制标记TLD中<tag>部分的<attribute>子元素有些相似。

标记文件中 (Header.tag)

```
<%@ attribute name="subTitle" required="true" rtexprvalue="true" %>  
 <br>  
<em><strong>${subTitle}</strong></em> <br>
```

这说明属性不是可选的。

可以是一个String直接量，也可以是表达式。

使用标记的JSP中

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>  
<html><body>  
  <myTags:Header subTitle="We take the String out of SOAP" />  
  
  <br>  
  Contact us at: ${initParam.mainEmail}  
</body></html>
```

使用标记时如果没有属性会怎么样

```
<myTags:Header />
```

这可不行……不能少了  
subTitle 属性，因为标  
记文件的attribute指令  
指出required="true"。

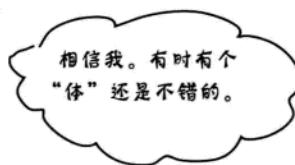


# 如果属性值确实很大

假设有一个标记属性很长，甚至是一段文字。把它们都放在开始标记里就很难看了。所以，可以把这些内容放在标记的体中，然后再作为一个属性使用。

这一次我们把subTitle属性从标记中取出，把它作为<myTags:Header>标记的体。

不再需要attribute指令了！



## 标记文件中(Header.tag)

```
 <br>
<em><strong><jsp:doBody/></strong></em> <br>
```



这是说，“如果一个标记调用这个标记文件，取得该标记体中的内容，并放在这里。”



## 使用此标记的JSP中

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
```

```
<myTags:Header>
```

We take the sting out of SOAP. OK, so it's not Jini,<br>
but we'll help you get through it with the least<br>
frustration and hair loss.

```
</myTags:Header>
```

现在给标记指定一个体，而不是把所有这些都作为开始标记中一个属性的值。

```
<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

不过，现在又回到老问题上了，如果没有TLD，你在哪里声明体内容（body-content）类型呢？



## 声明标记文件的body-content

为标记文件声明body-content类型只有一种方法，这要使用另一个新的标记文件指令：tag指令。标记文件中tag指令就相当于JSP页面中的page指令，它们有很多相同的属性，另外tag指令还有page指令中所没有的一个重要属性：body-content。

对于一个定制标记，TLD中<tag>元素的<body-content>元素是必要的！但是如果默认值（scriptless）可以接受，标记文件就不必声明<body-content>。如果值为scriptless，这说明不能有脚本元素。记住，脚本元素可以是scriptlet (<% ... %>)、脚本表达式(<%= ... %>)和声明(<%! ... %>)。

实际上，标记文件体不允许有脚本，所以对此别无选择。不过，如果希望是另外两个值empty或tagdependent，就可以声明body-content（使用有body-content属性的tag指令）。

有tag指令的标记文件中(Header.tag)

```
<%@ attribute name="fontColor" required="true" %>
<%@ tag body-content="tagdependent" %>
```

↑ 这说明body-content会处理为纯文本，也就是说EL、标记和脚本都  
会计算。另外两个可取值是“empty”或“scriptless”（默认值）。

```
 <br>
<em><strong><font color="${fontColor}"><jsp:doBody/></font></strong></em> <br>
```

使用标记的JSP中

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html>
```

“fontColor”由标记文件中的attribute指令声明。

```
<myTags:Header fontColor="#660099">
```

We take the sting out of SOAP. OK, so it's not Jini,<br>
but we'll help you get through it with the least<br>
frustration and hair loss.

```
</myTags:Header>
```

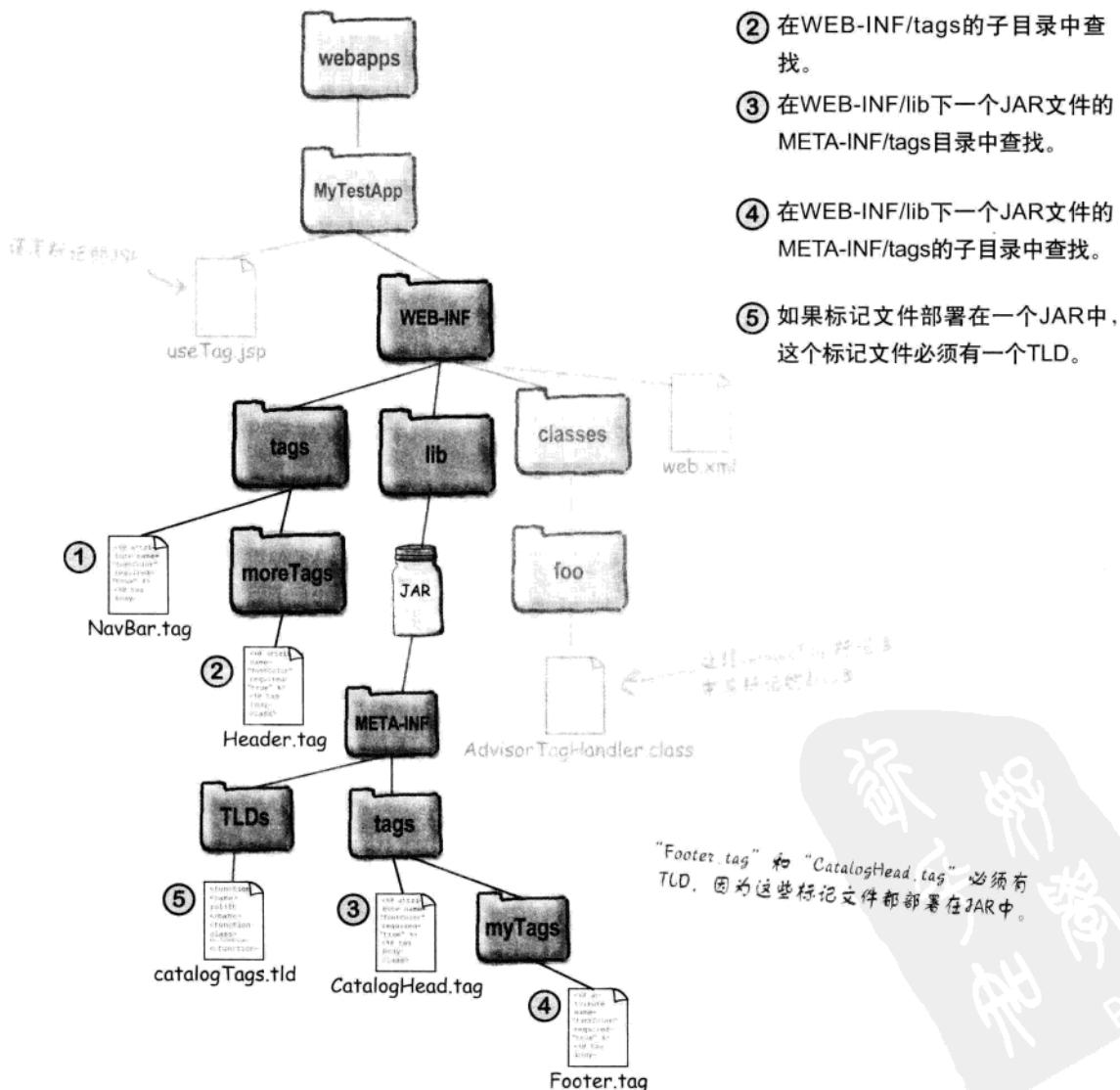
```
<br>
```

Contact us at: \${initParam.mainEmail}<br>
</body></html>

这个body-content的类型在标记文件中使用一个有body-content属性的tag指令声明。

# 容器在哪里查找标记文件

容器会在4个位置上查找标记文件。如果标记文件在一个JAR中部署，就必须有一个TLD，但是，如果直接放在Web应用中（在“WEB-INF/tags”目录或一个子目录中），就不需要TLD。



## there are no Dumb Questions

**问：** 标记文件能访问request和response隐式对象吗？

**答：** 可以！记住，尽管它是一个.tag文件，但最后还是要作为一个JSP的一部分。可以使用隐式对象request和response（如果使用脚本……还可以使用一般的EL隐式对象），而且还能访问JspContext。

但是，不能访问ServletContext，标记文件要使用JspContext而不是ServletContext。

**问：** 我记得上一页你说过，标记文件中不能有脚本！

**答：** 不对，我们不是这样说的。标记文件中可以写脚本，但是如果一个标记要调用标记文件，这个标记的体中是不能写脚本的。

**问：** 能不能将标记文件与同一个目录中定制标记的TLD合并？

**答：** 可以。实际上，如果建立一个引用了标记文件的TLD，容器会认为同一个TLD中提到的标记文件和定制标记都属于同一个库。

**问：** 等等——你说过，标记文件没有TLD，还记得吗？不就是因为这个原因才要使用attribute指令吗？因为你没办法在TLD中声明属性，是这样吗？

**答：** 这个问题有点意思。如果把标记文件部署在

JAR中，就必须有一个TLD来描述它的位置。但是它并不描述属性、body-content等内容。

标记文件的相应TLD项只描述具体标记文件的位置。

一个标记文件的TLD如下所示：

```
<taglib ...>
  <tlib-version>1.0</tlib-version>
  <uri>myTagLibrary</uri>
  <tag-file>
    <name>Header</name>
    <path>/META-INF/tags/Header.tag</path>
  </tag-file>
</taglib>
```

注意，声明一个<tag-file>与声明一个具体的<tag>是完全不同的。

**问：** 为什么要这样做？用同样的办法在TLD中声明定制标记和标记文件不是简单得多吗？这样多不好……还得提供一堆新东西，必须使用新指令来定义属性和body-content。为什么要用不同的方式处理标记和标记文件？

**答：** 如果定制标记和标记文件都以同样的方式在TLD中声明，这当然更简单一些。但问题是，是谁觉得简单？对于定制标记开发人员，当然是会简单一些。但是在规范中加入标记文件时还考虑到了另外一些人，这就是界面设计人员。

利用标记文件，非Java程序员也能建立定制标记，而无需编写Java类来处理标记的功能，也不必为了让标记文件开发人员更轻松而被迫为标记文件建立TLD。（记住，如果标记文件部署在JAR中，确实需要一个TLD，但是非Java程序员不大可能会使用JAR）。

关键是：定制标记必须有TLD，而对于标记文件，可以直接在标记文件中声明属性和body-content，而且只有当标记文件部署在一个JAR中时才需要TLD。



看看你对标记文件记得怎么样

看接下来的新内容之前，先要保证你能自己完成这个练习（答案在这一章的最后）。

- ① 在标记文件中填空，声明该标记有一个必要的属性，名为“title”，它能使用EL表达式作为属性值。

&lt;%@

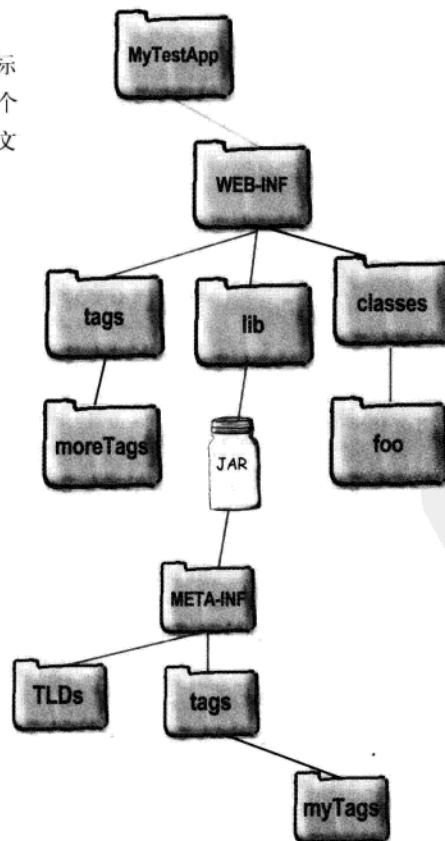
%&gt;

- ② 在标记文件中填空，声明标记不能有体。

&lt;%@

%&gt;

- ③ 容器会在哪些位置查找标记文件，请在相应的各个位置上画一个标记文件文档。



## 如果标记文件还不够…… 有时还需要Java

如果要完成包含，使用标记文件就很合适，处理标记需要做的工作都可以在另一个JSP中完成(这个JSP要重命名为有一个.tag扩展名，并增加适当的指令)。但是，有时这样还不够。有些情况下你还需要原来的Java代码，但又不想用scriptlet，因为使用标记的出发点就是想避免脚本。

需要Java时，你想要的是一个定制标记处理器。标记处理器不同于标记文件，它是一个完成标记工作的Java类。这与EL函数有点相似，但是更强大，也更灵活。EL函数只是一些静态方法，标记处理器类则可以访问标记属性、标记体，甚至还能访问页面上下文，从而得到作用域属性和请求及响应。

定制标记处理器有两种类型：传统和简单。传统标记就是JSP以前版本中的定制标记，不过，在JSP 2.0中又增加了一个更简单的新模型。如果你需要一个定制标记处理器，一般不再需要使用传统模型，因为这个简单模型（特别是结合JSTL和标记文件）几乎就能满足你的所有需要。但是我们还不能完全放弃传统模型，这有两个原因，正是出于这两个原因，传统模型在考试中还可能出现，所以你必须学习这个内容：

- 1) 像脚本一样，传统标记处理器已经广泛使用，尽管你本身可能不会创建传统标记，但很可能需要阅读和维护这样一些代码。
- 2) 在很少见的一些情况下，传统标记处理器是最佳的选择。不过，这一点不太明确。所以第一点是目前学习传统标记最重要的原因。

我们会先从简单标记模型开始热热身。

标记文件利用另一个页面  
(使用JSP) 实现标记功能。

标记处理器利用一个特别的Java类实现标记功能。

有两种类型的标记处理器：简单和传统。

# 建立一个简单标记处理器

对于最简单的简单标记，这个过程……很简单。

## ① 编写一个扩展SimpleTagSupport的类

```
package foo;
import javax.servlet.jsp.tagext.SimpleTagSupport;
// more imports needed

public class SimpleTagTest1 extends SimpleTagSupport {
    // tag handler code here
}
```

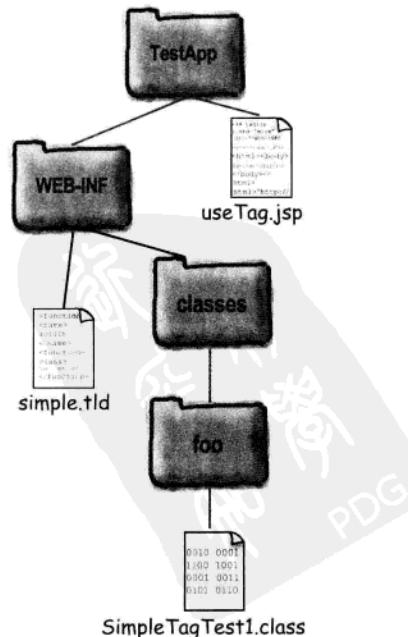
## ② 覆盖doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("This is the lamest use of a custom tag");
}
```

↑  
doTag()方法声明了一个IOException，所以不必把print包装在一个try/catch中。

## ③ 为标记创建一个TLD

```
<taglib ...>
<tlib-version>1.2</tlib-version>
<uri>simpleTags</uri>
<tag>
    <description>worst use of a custom tag</description>
    <name>simple1</name>
    <tag-class>foo.SimpleTagTest1</tag-class>
    <body-content>empty</body-content>
</tag>
</taglib>
```



## ④ 部署标记处理器和TLD

遵循包目录结构，把TLD放在WEB-INF中，并把标记处理器放在WEB-INF/classes下。换句话说，标记处理器类要与所有其他Web应用Java类放在同一个位置上。

## ⑤ 编写一个使用标记的JSP

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
<myTags:simple1/>
</body></html>
```

## 有体的简单标记

如果标记需要体，TLD <body-content>要反映出这一点，而且doTag()方法中需要有一条特殊的语句。

### 使用标记的JSP

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
Simple Tag 2:

<myTags:simple2>
    This is the body <----- 这一次，调用有
    体的标记……
</myTags:simple2>

</body></html>
```

### 标记处理器类

```
package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;

public class SimpleTagTest2 extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        getJspBody().invoke(null);
    }
}
```

这是说，“处理标记的体，并把它打印到响应。”null参数是指输出到响应，而不是输出到作为参数传入的某个书写器（writer）。

### 标记的TLD

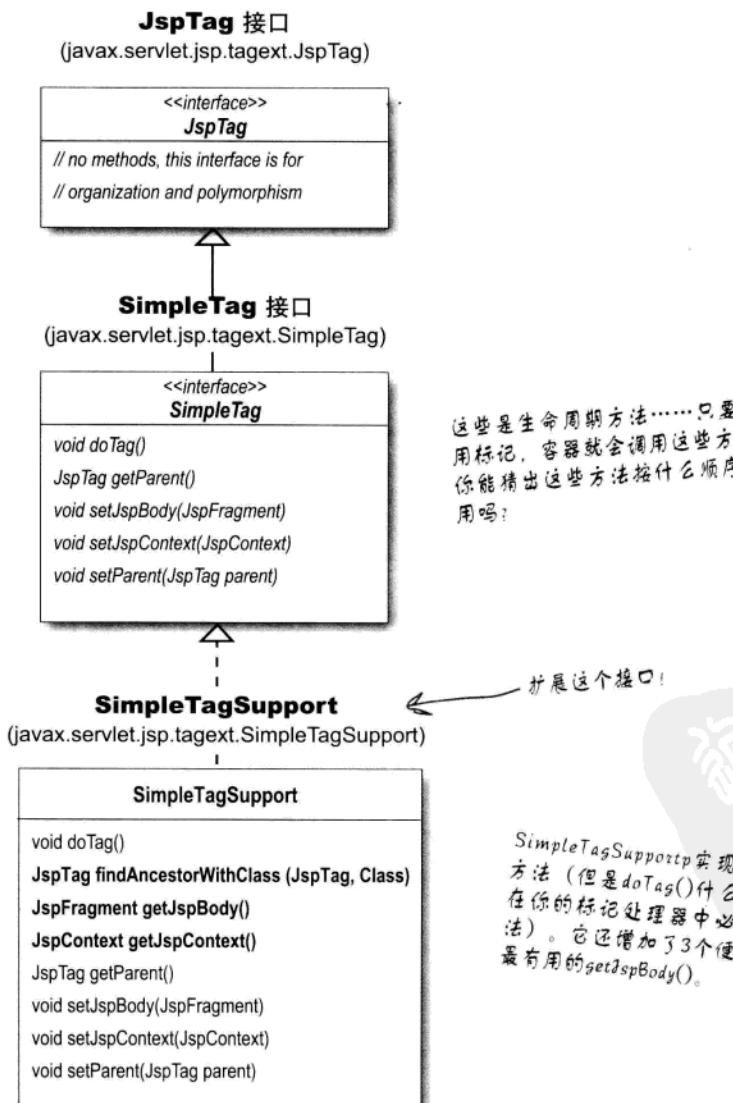
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" ver-
  sion="2.0">

    <tlib-version>1.2</tlib-version>
    <uri>simpleTags</uri>
    <tag>
        <description>marginally better use of a custom tag</description>
        <name>simple2</name>
        <tag-class>foo.SimpleTagTest2</tag-class>
        <body-content>scriptless</body-content>
    </tag>
</taglib>
```

这说明标记可能有体，但是体中不能有脚本（scriptlet、脚本表达式或声明）。

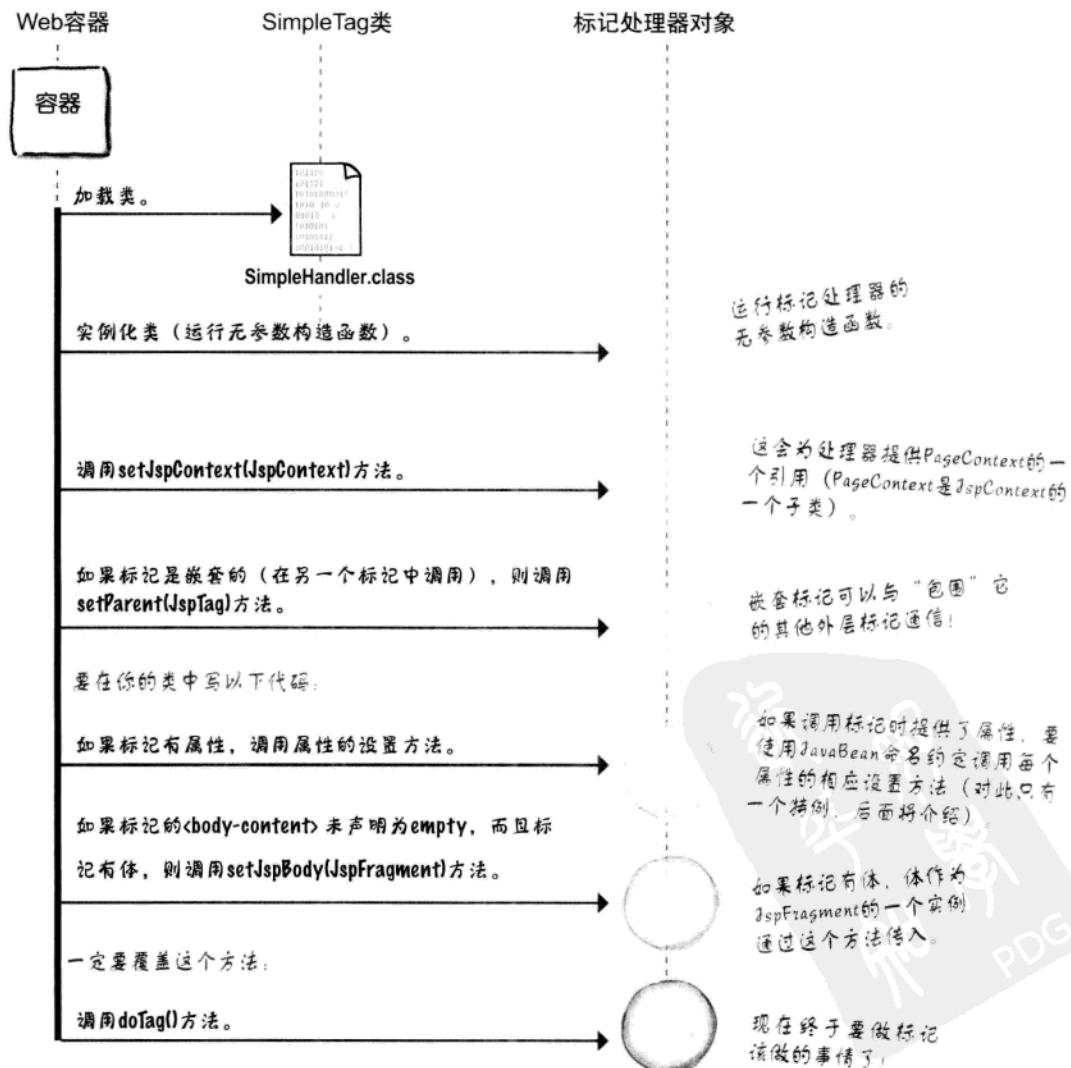
# 简单标记API

简单标记处理器必须实现SimpleTag接口。最容易的方法是扩展SimpleTagSupport，并且只覆盖你需要的方法：doTag()。不一定非得使用SimpleTagSupport，但我们相信99.999999%的简单标记开发人员都会这样做。



## 简单标记处理器的生命周期

JSP调用一个标记时，会实例化该标记处理器类的一个新实例，并调用处理器上的两个或更多方法，doTag()方法完成时，处理器对象撤销（换句说话，这些处理器对象不会重用）。





## 作为容器

查看每组TLD/JSP。假设标记处理器会打印标记的体。对以下每种情况回答下面的问题：结果是什么？如果能工作，会打印出什么？会调用定制标记类的哪些方法？

①

```
<tag>
<description></description>
<name>simple</name>
<tag-class>foo.SimpleTagTest</tag-class>
<body-content>empty</body-content>
</tag>
```

Simple Tag:  
<myTags:simple>  
This is the body of the tag  
</myTags:simple>

在浏览器中会看到什么？

如果能工作，会在处理器中调用哪些SimpleTag生命周期方法？

- void doTag()     JspTag getParent()     void setJspBody()     void setJspContext()     void setParent()

②

```
<tag>
<description></description>
<name>simple</name>
<tag-class>foo.SimpleTagTest</tag-class>
<body-content>scriptless</body-content>
</tag>
```

Simple Tag:  
<myTags:simple>  
\${2\*3}  
</myTags:simple>

在浏览器中会看到什么？

如果能工作，会在处理器中调用哪些SimpleTag生命周期方法？

- void doTag()     JspTag getParent()     void setJspBody()     void setJspContext()     void setParent()

# 作为容器

答案



①  <tag>

```
<description></description>
<name>simple</name>
<tag-class>foo.SimpleTagTest</tag-class>
<body-content>empty</body-content>
</tag>
```

Simple Tag:

<myTags:simple>

This is the body of the tag

</myTags:simple>

在浏览器中会看到什么？

无法工作，因为这个标记的体应该为空。

```
org.apache.jasper.JasperException: /simpleTag1.jsp(1,76)
According to TLD, tag myTags:simple must be empty, but is not
```

如果能工作，会在处理器中调用哪些SimpleTag生命周期方法？

- void doTag()     JspTag getParent()     void setJspBody()     void setJspContext()     void setParent()

②  <tag>

```
<description></description>
<name>simple</name>
<tag-class>foo.SimpleTagTest</tag-class>
<body-content>scriptless</body-content>
</tag>
```

Simple Tag:

<myTags:simple>

$\$\{2*3\}$

</myTags:simple>

在浏览器中会看到什么？

Simple Tag: 6

如果能工作，会在处理器中调用哪些SimpleTag生命周期方法？

- void doTag()     JspTag getParent()     void setJspBody()     void setJspContext()     void setParent()

只有从另一个标记中调用这个标记时，才会调用.setParent()方法。因为这个标记不是嵌套的，所以调用.setParent()。

# 如果标记体使用了表达式呢？

假设有一个有体的标记，而且体中使用了一个属性的EL表达式。现在假设调用这个标记时该属性尚不存在！换句话说，标记体要依赖标记处理器设置属性。这个例子并没有多大的实际意义，这里只是想告诉你它是怎么工作的，以便为后面更大的例子做准备。

## JSP标记调用

```
<myTags:simple3>
    Message is: ${message}
</myTags:simple3>
```

调用标记时，“message”不是  
一个作用域属性！如果从标记取  
这个表达式，会返回null。

## 标记处理器doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().setAttribute("message", "Wear sunscreen.");
    getJspBody().invoke(null);
}
```

标记处理器设置一个  
属性，然后调用体。



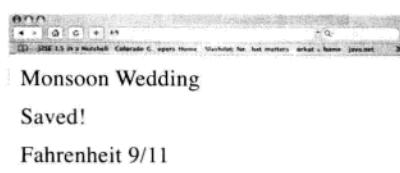
假设有这样一个标记：

```
<table>
<myTags:simple4>
    <tr><td>${movie}</td></tr>
</myTags:simple4>
</table>
```

编写标记处理器doTag()方法来实现  
这个目标。

```
public void doTag() throws JspException,
    IOException {
```

再假设标记处理器可以访问电影名String组成  
的一个数组，而且你希望每行打印数组中的一  
个电影名。在浏览器中将看到：



# 有动态行数据的标记：迭代执行体

在这个例子中，标记体中的EL表达式表示集合中的一个值，我们的目标是让标记为集合中的每个元素生成一行。这很简单，doTag()方法只需在一个循环中完成这个工作，每次迭代时都调用体。

## JSP标记调用

```
<table>
  <myTags:simple4>
    <tr><td>${movie}</td></tr>
  </myTags:simple4>
</table>
```

调用标记时movie属性还不存在。  
标记处理器会设置这个属性，并重  
复调用体。

## 标记处理器doTag()方法

```
String[] movies = {"Monsoon Wedding", "Saved!", "Fahrenheit 9/11"};

public void doTag() throws JspException, IOException {
    for(int i = 0; i < movies.length; i++) {
        getJspContext().setAttribute("movie", movies[i]);
        getJspBody().invoke(null);
    }
}
```

将属性值设置为数  
组中的下一个元素。  
再次调用体。

## JSP

```
<myTags:simple4>
  <tr><td>
    ${movie}
  </td></tr>
</myTags:simple4>
```

## 标记处理器

```
for(int i = 0; i < movies.length; i++) {
    getJspContext().setAttribute("movie", movies[i]);
    getJspBody().invoke(null);
}
```

标记处理器每次循环时都会重新  
设置“movie”属性值，并再次调  
用getJspBody().invoke()。

# 有属性的简单标记

如果标记需要一个属性，就要在TLD中声明，并在标记处理器中为每个属性提供一个bean式的方法。如果调用标记时包括有属性，容器就会为每个属性调用一个设置方法。

## JSP标记调用

```
<table>
<myTags:simple5 movieList="${movieCollection}">
  <tr>
    <td>${movie.name}</td>
    <td>${movie.genre}</td>
  </tr>
</myTags:simple5>
</table>
```

这是一个属性，与其他标记属性没有什么两样。这里负责完成标记工作的是一个简单标记处理器。

## 标记处理器类

```
public class SimpleTagTest5 extends SimpleTagSupport {
    private List movieList; // 声明一个变量来保存属性。
    public void setMovieList(List movieList) { // 为属性编写一个bean式的方法。方法名必须与TLD中的属性名匹配（去掉“set”前缀，而且要把第一个字母改为小写）。
        this.movieList=movieList;
    }
    public void doTag() throws JspException, IOException {
        Iterator i = movieList.iterator();
        while(i.hasNext()) {
            Movie movie = (Movie) i.next();
            getJspContext().setAttribute("movie", movie);
            getJspBody().invoke(null);
        }
    }
}
```

这里没有显示  
import语句……

## 标记的TLD

```
<tag>
  <description>takes an attribute and iterates over body</description>
  <name>simple5</name>
  <tag-class>foo.SimpleTagTest5</tag-class>
  <body-content> scriptless </body-content>
  <attribute>
    <name>movieList</name>
    <required>true</required>
    <rteprvalue>true</rteprvalue>
  </attribute>
</tag>
```

在TLD中使用常规的<tag><attribute>声明，就像其他定制标记一样（但标记文件是例外）。

## 到底什么是JspFragment?

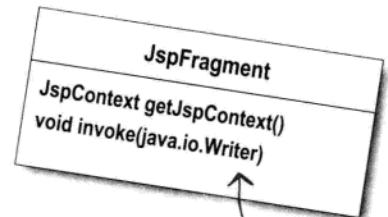
JspFragment是表示JSP代码的一个对象。它存在的意义就是让别人调用。换句话说，它要运行并生成输出。如果标记调用了一个简单标记处理器，这个标记的体就会封装在JspFragment对象中，然后在setJspBody()方法中发送给标记处理器。

关键是必须记住，JspFragment中不能包含任何脚本元素！也就是说，可以包含模板文本、标准和定制动作，以及EL表达式，但是不能出现scriptlet、声明或脚本表达式。

有一点很棒，因为JspFragment是一个对象，所以可以把这个片段传递给其他辅助对象。这些辅助对象再调用JspFragment的另一个方法getJspContext()，从中得到信息。当然，一旦得到上下文，就可以请求属性。所以getJspContext()实际上是标记体向其他对象提供信息的一个途径。

不过，在大多数情况下，使用JspFragment只是为了把标记体输出到响应。但是你可能还想访问体的内容。注意JspFragment没有getContents()或getBody()之类的访问方法。可以把体写到某处，但是不能直接得到体。如果你确实想访问体，可以使用invoke()方法的参数来传入一个java.io.Writer，然后使用该Writer的方法处理标记体的内容。

无论是考试还是实际开发，都不需要了解太多JspFragment的细节，你只要知道这么多就够了，所以这本书不再在这上面浪费笔墨了。



invoke()方法取一个Writer参数。  
如果传入null，会把体发送到响应。  
如果你想直接访问具体的  
输出，如果你想直接访问具体的  
体内容，就要传入一个Writer。

invoke()方法取一个java.io.Writer参数。如果想让体写至  
响应输出，可以向invoke方法传递null参数。

在大多数情况下，都是这么做的。不过，如果你想访问  
具体的体内容，可以传入一个Writer，然后使用这个Writ-  
er以某种方式处理体。

# SkipPageException：停止处理页面……

假设有一个页面调用了标记，而且标记依赖于特定的请求属性（从标记处理器可用的JspContext得到）。

前面假设这个标记找不到它需要的属性，它知道如果自己不成功，页面的余下部分就不能正常工作，你该怎么办呢？可以让标记抛出一个JspException，这就能让页面结束……但是如果只是想让这个页面的余下部分不工作，该怎么办？换句话说，如果你希望页面的前一部分（调用此标记之前已经计算的部分）还会作为响应出现，但响应中不该有“余下部分”（即位于标记抛出异常之后还没有处理的部分），该怎么做？

没问题。就是因为这个目的才有了SkipPageException。

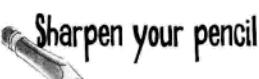
## 标记处理器doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("Message from within doTag().<br>");
    getJspContext().getOut().print("About to throw a SkipPageException");
    if (thingsDontWork) {
        throw new SkipPageException(); ← 在这里，我们决定停止标记的余下部分
    } ← 和页面的余下部分。响应中只会出现前一部分（抛出异常之前的
}
```

部分）。  
部分）。

## 调用标记的JSP

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
About to invoke a tag that throws SkipPageException <br>
<myTags:simple6/> ← 以上doTag()方法（抛出了一个
<br>Back in the page after invoking the tag. ← SkipPageException）中处理的标记。
</body></html>
```



如果thingsDontWork测试为true，会有什么结果？

填入浏览器中显示的结果：



# SkipPageException会显示异常出现之前的所有内容

doTag()方法中SkipPageException之前的所有内容都会在响应中显示出来。但是抛出异常后，标记或页面中余下的部分不再计算。

## JSP中

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
```

```
<html><body>
```

```
About to invoke a tag that throws SkipPageException <br>
<myTags:simple6/>
```

*<br>Back in the page after invoking the tag.*

← 这不会打印出来！

```
</body></html>
```



## 标记处理器中

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("Message from within doTag().<br>");
    getJspContext().getOut().print("About to throw a SkipPageException");
    if (thingsDontWork) {
        throw new SkipPageException();
    }
}
```

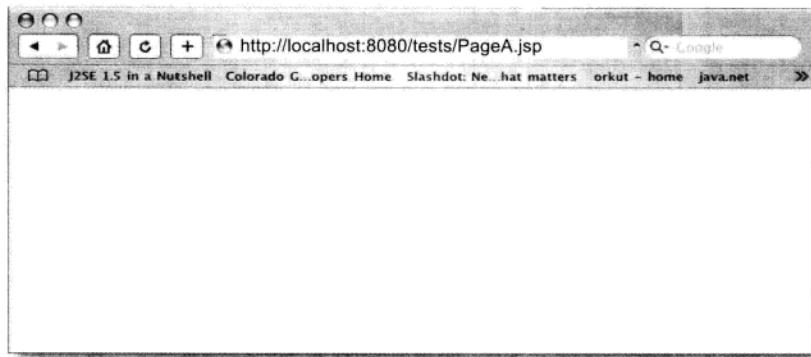
# 如果从一个被包含页面调用标记，会发生什么情况？



请看以下代码，得出浏览PageA时会打印什么。

提示：查看javax.servlet.jsp.SkipPageException的API。

填入浏览器中显示的结果：



## PageA JSP包含PageB

```
<html><body>
This is page (A) that includes another page (B). <br>
Doing the include now:<br>
<jsp:include page="badTagInclude.jsp" />
<br>Back in page A after the include...
</body></html>
```

## PageB（被包含文件）JSP 调用了可能抛出异常的标记

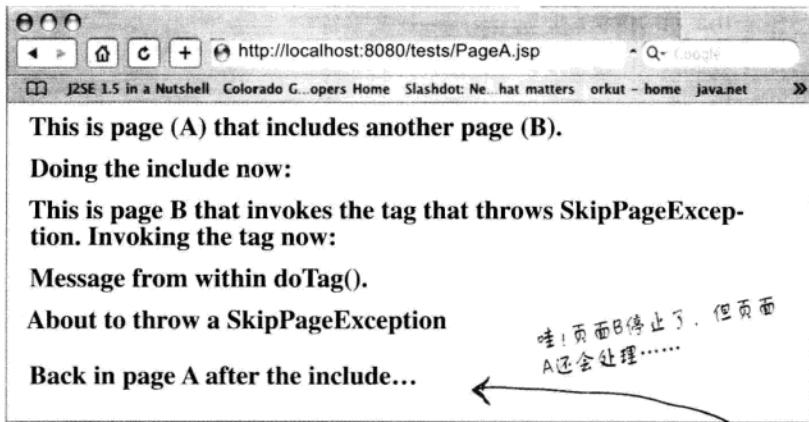
```
<%@ taglib prefix="myTags" uri="simpleTags" %>
This is page B that invokes the tag that throws SkipPageException.
Invoking the tag now:<br>
<myTags:simple6/>
<br>Still in page B after the tag invocation...
```

## 标记处理器doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("Message from within doTag().<br>");
    getJspContext().getOut().print("About to throw a SkipPageException");
    throw new SkipPageException();
}
```

# SkipPageException只停止直接调用标记的页面

如果调用标记的页面包含在另一个页面中，那么只有调用标记的这个页面会停止处理！包含该页面的外层页面在SkipPageException之后仍继续执行。



PageA JSP 包含PageB

```
<html><body>
    This is page (A) that includes another page (B).<br>
    Doing the include now:<br>
    <jsp:include page="badTagInclude.jsp" />
    <br>Back in page A after the include...
</body></html>
```

看到页面A的这一行居然会打印出来，是不是很让人意外？

PageB（被包含文件）JSP 调用了可能抛出异常的标记

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
This is page B that invokes the tag that throws SkipPageException.
Invoking the tag now:<br>
<myTags:simple6/>
```

<br>Still in page B after the tag invocation...

不出所料，这一行没有打印出来。

标记处理器doTag()方法

```
public void doTag() throws JspException, IOException {
    getJspContext().getOut().print("Message from within doTag().<br>");
    getJspContext().getOut().print("About to throw a SkipPageException");
    throw new SkipPageException();
}
```

这会停止页面B，但页面A继续处理。

## there are no Dumb Questions

**问：** 完成`doTag()`之后，SimpleTag处理器会怎么样？容器会保留它并重用吗？

**答：** 不，SimpleTag处理器不会重用！每个标记处理器实例只负责一次调用。所以有些问题你根本不用担心，例如，SimpleTag处理器中的实例变量不可能有不正确的初始值。SimpleTag处理器对象肯定会在调用其方法之前先初始化。

**问：** SimpleTag处理器中属性方法必须是一种能与String来回自动转换的类型吗？换句话说，是不是只能使用基本类型和String值？

**答：** 你没注意前面的几个页面吗？我们发给SimpleTag处理器的属性就是电影组成的一个`ArrayList`。所以，对于你的这个问题，答案是否定的。但是……如果属性（如果把SimpleTag处理器想成是一个bean，也可以把属性认为是这个bean的性质）不是一个String或基本类型，那么TLD中的`<rtexprvalue>`值最好设置

为`true`。标记中无法表示为String的属性值只能通过这种方法来设置。换句话说，如果要求把Dog表示为一个String直接量，不能把Dog发送到标记中。但是如果属性值可以使用表达式，这个表达式就能计算为你需要的任何对象类型，从而与处理器相应设置方法的参数匹配。

**问：** 在一个SimpleTag处理器中，如果标记声明为有体，但是调用的却是一个空标记（因为没有办法指定必须有体），还会调用`setJspBody()`吗？

**答：** 不会！只有满足以下两种情况时才会调用`setJspBody()`：

- 1) TLD中没有将标记声明为体为空。
- 2) 调用标记时有体。

这说明，即使标记声明为体非空，倘若以下面两种方式调用标记，就不会调用`setJspBody()`方法：

`<foo:bar />` (空标记)

`<foo:bar></foo:bar>` (没有体)。



## 要点

- 标记文件使用一个页面来实现标记功能，而标记处理器使用一个Java标记处理器类实现标记功能。
- 标记处理器有两种类型：传统和简单（简单标记和标记文件是JSP 2.0中新增加的）。
- 建立简单标记处理器时，要扩展SimpleTagSupport（这个类实现了SimpleTag接口）。
- 要部署一个标记处理器，必须创建一个TLD，使用<tag>元素来描述标记，JSTL和其他定制标记库也使用这个元素描述标记。
- 如果使用一个有体的简单标记，要保证这个标记的TLD<tag>没有将<body-content>声明为empty。然后调用getJspBody().invoke()来处理体。
- SimpleTagSupport类包括SimpleTag接口中所有方法的实现，另外还提供了3个便利方法，其中包括getJspBody()，可以用这个方法访问标记体的内容。
- 简单标记生命周期：简单标记不会由容器重用，所以每次调用标记时，都会实例化标记处理器，并调用其setJspContext()方法。如果标记本身是从另一个标记中调用的，则会调用setParent()方法。如果调用标记时有属性，对于每个属性会调用一个bean式的设置方法。如果调用标记时有体（假设其TLD没有声明它的体为空），则会调用setJspBody()方法。最后，调用doTag()方法，结束时，撤销标记处理器实例。
- 只有调用标记时确实有体，才会调用setJspBody()方法。如果调用标记时没有体，不论是空标记<my:tag/>，还是开始和结束标
 

记<my:tag></my:tag>之间没有内容，都不会调用setJspBody()方法。记住，如果标记有体，TLD必须反映出这一点，而且<body-content>的值不能是“empty”。
- 简单标记的doTag()方法可以设置标记体使用的一个属性，为此先调用getJspContext().setAttribute()，再调用getJspBody().invoke()。
- doTag()方法声明了一个JspException和一个IOException，所以可以直接写至JspWriter，而无需包装在一个try/catch中。
- 通过在循环中调用体（getJspBody().invoke()），可以迭代处理简单标记的体。
- 如果标记有一个属性，要在TLD中使用<attribute>元素声明这个属性，并在标记处理器类中提供一个bean式的选择方法。调用标记时，会在doTag()之前先调用这个设置方法。
- getJspBody()方法返回一个JspFragment，它有两个方法：invoke(java.io.Writer)和getJspContext()，getJspContext()返回一个JspContext，标记处理器可以用这个JspContext访问PageContext API（访问隐式变量和作用域属性）。
- 如果向invoke()传入null，会把计算的体写至响应输出，不过，如果你想直接访问体内容，可以传入另一个Writer。
- 如果你希望当前页面停止处理，可以抛出一个S。如果调用标记的页面包含在另一个页面中，尽管被包含的页面在抛出异常之后就停止处理，但外层页面仍会继续。

JSP规范的设计者们能提供简单标记和标记文件真是太好了，不过，它们来得太晚了，我的公司已经写了大约一千万个使用传统模型的定制标记……



## 你还要对传统标记处理器有所了解

你可能很幸运，你的公司可能直接采用了JSP 2.0，而且从一开始就使用标记文件和SimpleTag处理器。

这是有可能的。

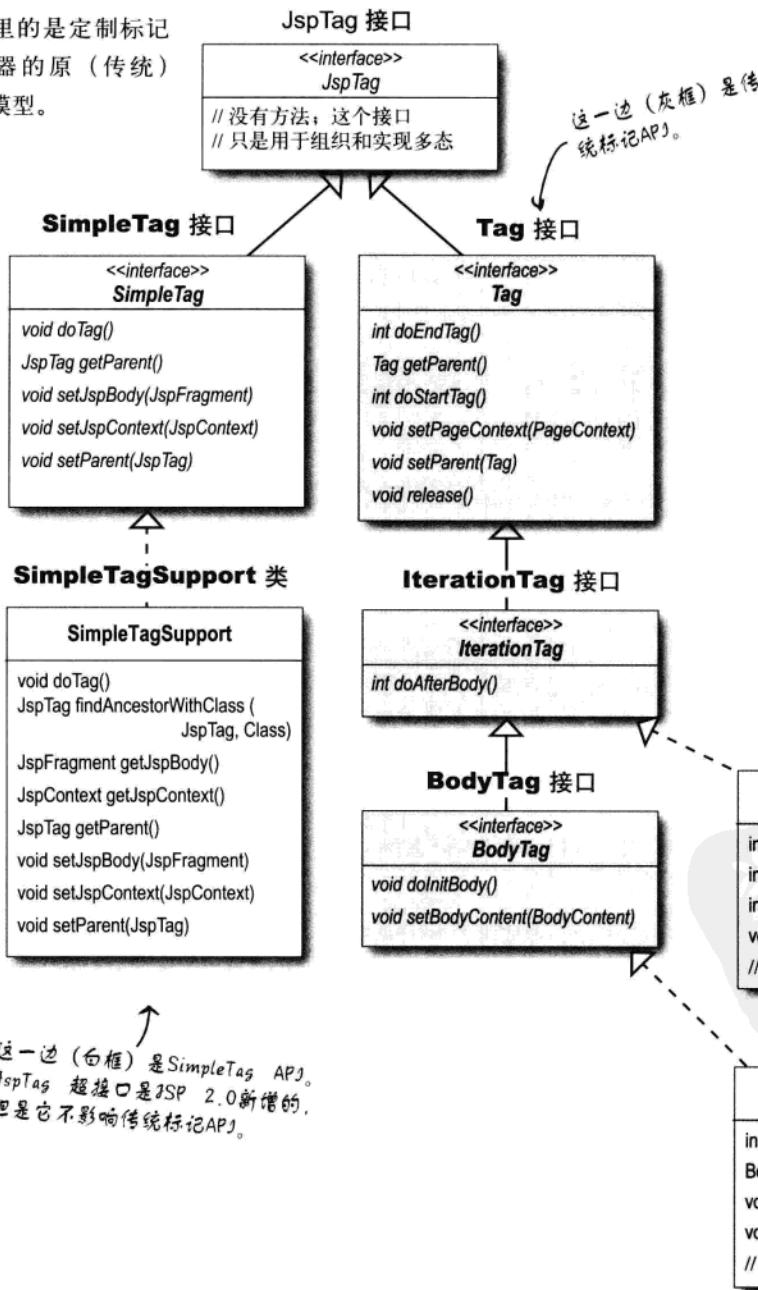
但是也有可能不是这样。没准，你现在的单位（或将来的公司）一直在使用JSP 2.0以前的版本，还在用传统标记模型编写定制标记处理器。

至少，你可能要读一个传统标记处理器的源代码。而且可能让你维护或重构一个传统标记处理器类。

即使你不用读或编写一个传统标记处理器，考试大纲里也有这样一个要求（不过要求不高）。你庆幸吧，以前的考试中可能会遇到7个或8个有关传统标记处理器的问题，现在，这方面的问题可能只有一两个。

# Tag处理器 API

灰框里的是定制标记  
处理器的原（传统）  
标记模型。



标记处理器 API 有 5  
接口和 3 个支持类。

一般都不需要直接  
实现这些接口，所以  
常可能要扩展一个  
支持类。

# 一个非常小的传统标记处理器

这个例子太基础了，它与SimpleTag处理器的doTag()方法几乎没有什么区别。实际上，除非要处理有体的标记，否则确实没有太大的差别（不过，这个内容过一章再讲）。

## 调用传统标记的JSP

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
    Classic Tag One:<br>
    <mine:classicOne />. ← 这个标记使用了一个传统标记处理器。但对于JSP来说，它与其他标记调用看起来是一样的。
</body></html>
```

## 传统标记的TLD <tag>元素

```
<tag>
    <description>ludicrous use of a Classic tag</description>
    <name>classicOne</name>
    <tag-class>foo.Classic1</tag-class> ← 无法知道这个<tag>是否由一个传统标记处理器处理，除非你知道foo.Classic1类实现了Tag接口（而不是SimpleTag）。可以完全替换foo.Classic1代码，让它使用SimpleTag，而TLD可以不做任何改变。
    <body-content>empty</body-content>
</tag>
```

## 传统标记处理器

```
package foo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class Classic1 extends TagSupport {
    public int doStartTag() throws JspException { ← 通过扩展TagSupport，这就实现了Tag和IterationTag。
        JspWriter out = pageContext.getOut(); ← 这个方法声明了JspException，而不是
        try { IOException! (SimpleTag doTag() 声明
            out.println("classic tag output"); ← 变量 (而SimpleTag要使用getJspContext()方法)。
            } catch(IOException ex) { ← 这里必须使用一个try/catch，因为
                throw new JspException("IOException- " + ex.toString()); ← 我们不能声明IOException。
            }
        return SKIP_BODY; ← 必须返回一个int告诉容器接下来做什么。这一点后面还会详细介绍……
    }
}
```

# 有两个方法的传统标记处理器

这个例子覆盖了doStartTag()和doEndTag()方法，不过，完全用doStartTag()也能完成同样的输出。doEndTag()的关键是，它在体计算之后再调用。这里没有给出TLD，因为它与前面的TLD几乎是一样的，只是某些名字有所不同。这个标记声明为没有属性，而且体为空。

## 调用传统标记的JSP

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
    Classic Tag Two:<br>
    <mine:classicTwo />
</body></html>
```

## 传统标记处理器

```
public class Classic2 extends TagSupport {
    JspWriter out;

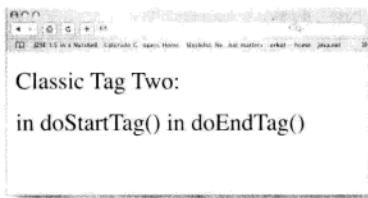
    public int doStartTag() throws JspException {
        out = pageContext.getOut();
        try {
            out.println("in doStartTag()");
        } catch( IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        try {
            out.println("in doEndTag()");
        } catch( IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_PAGE;
    }
}
```

我们将不再显示package或import语句，除非增加了一个新包中的类。

这是说，“如果有体，不要计算体，直接调用doEndTag()方法。”

这是说，“计算页面余下的部分”（不同于SKIP\_PAGE, SkipPageException）。



# 标记有体时：简单标记和传统标记的比较

下面来看与SimpleTag的不同之处。记住，只要你愿意，通过对封装了体的JspFragment调用invoke()，这样就会计算（执行）SimpleTag体。但是在传统标记中，体在doStartTag()和doEndTag()方法之间计算。下面的例子有同样的表现。

## 使用标记的JSP

```
<%@ taglib prefix="myTags" uri="myTags" %>
<html><body>
<myTags:simpleBody>
    This is the body
</myTags:simpleBody>
</body></html>
```

## SimpleTag处理器类

```
// package and imports
public class SimpleTagTest extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().print("Before body.");
        getJspBody().invoke(null); ← 这会导致体计算（执行）。
        getJspContext().getOut().print("After body.");
    }
}
```

如果做同样的事情，传统标记处理器如下：

```
// package and imports
public class ClassicTest extends TagSupport {
    JspWriter out;

    public int doStartTag() throws JspException {
        out = pageContext.getOut();
        try {
            out.println("Before body.");
        } catch(IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_BODY_INCLUDE; ← 在传统标记处理器中这才会导致体计算！
    }

    public int doEndTag() throws JspException {
        try {
            out.println("After body.");
        } catch(IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_PAGE;
    }
}
```

那么你怎么循环处理体呢?看上去`doStartTag()`调用得太平,`doEndTag()`又太晚,我没办法一直反复调用体计算……



### 简单标记

```
// package and imports
public class SimpleTagTest extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        for(int i = 0; i < 3, i++) {
            getJspBody().invoke(null);
        }
    }
}
```

很容易循环处理简单标记的体;只要在`doTag()`中一直调用`invoke()`就可以了。



### 传统标记

```
// package and imports
public class ClassicTest extends TagSupport {

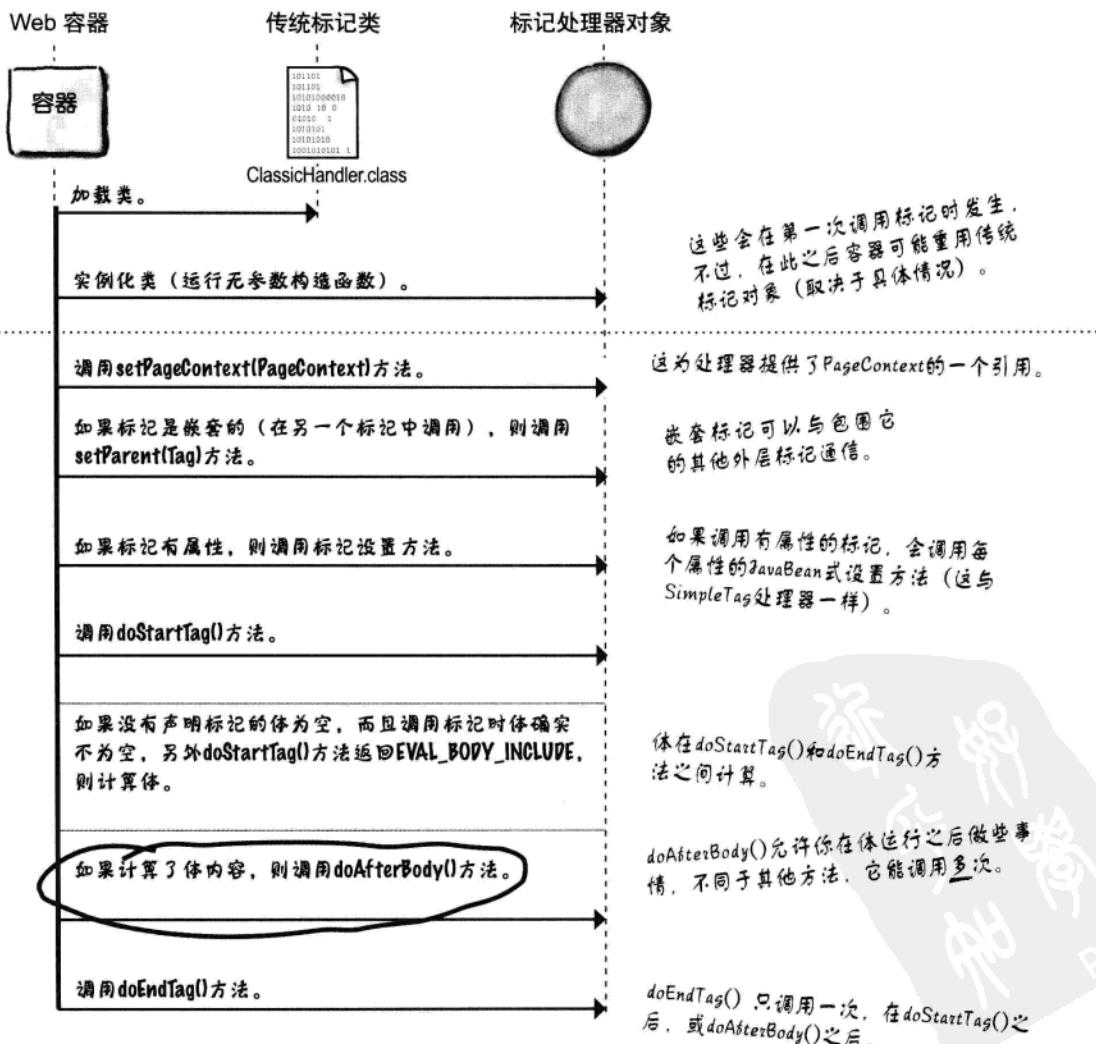
    public int doStartTag() throws JspException {
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }
}
```

但是,如果体在方法之间计算,而不是在`doTag()`之类的某个方法中计算,你在哪里位置上循环处理体呢?

# 传统标记的生命周期是不同的

简单标记很简单，只有一个doTag()。但是，传统标记有一个doStartTag()和一个doEndTag()，这就带来一个有意思的问题，什么时候计算体，怎么计算体？这里没有doBody()方法，但有一个doAfterBody()方法，它在计算体之后并在doEndTag()运行前调用。



# 传统标记生命周期取决于返回值

`doStartTag()`和`doEndTag()`方法返回一个int。这个int告诉容器下一步做什么。对于`doStartTag()`，容器的问题是“我该计算体吗？”（假设体，而且假设TLD没有把体声明为空（empty））。

对于`doEndTag()`，容器问的是“我要继续计算调用页面的余下部分吗？”返回值表示为常量，这些常量在Tag和IterationTag接口中声明。

扩展TagSupport时可能的返回值

`doStartTag()`

SKIP\_BODY

EVAL\_BODY\_INCLUDE

`doAfterBody()`

SKIP\_BODY

EVAL\_BODY\_AGAIN

`doEndTag()`

SKIP\_PAGE

EVAL\_PAGE

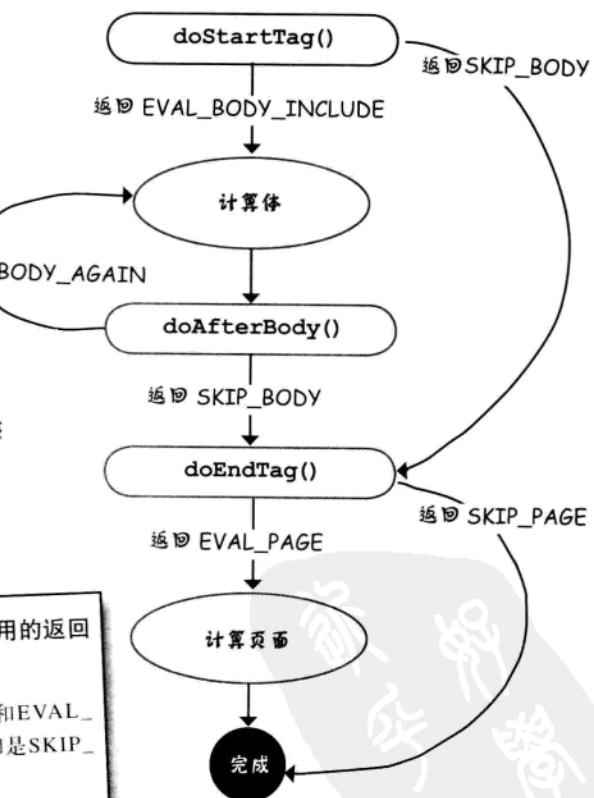
只有这一个返回值常量在  
IterationTag中声明（其他  
常量都在Tag中声明）。



`doStartTag()`和`doEndTag()`所用的返回  
值常量命名不一致！

对于`doStartTag()`，返回值是`SKIP_BODY`和`EVAL_BODY_INCLUDE`。但对于`doEndTag()`，值却是`SKIP_PAGE`和`EVAL_PAGE`。

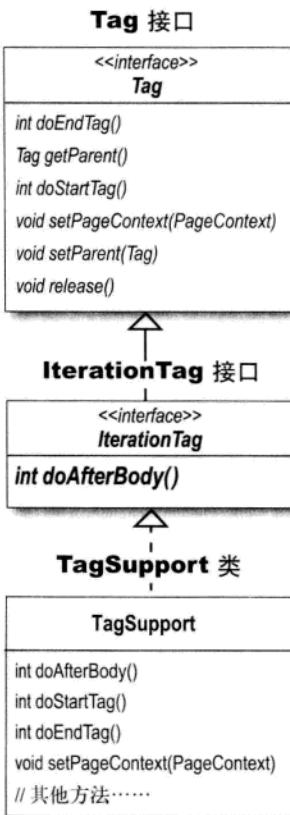
如果取一致的名字，因为`doStartTag()`返回的是`EVAL_BODY_INCLUDE`，所以`doEndTag()`返回`EVAL_PAGE`才对（而不是`EVAL_PAGE`），这样才能一致。但它没有这样命名！所以，考试时有些代码中返回值可  
能看上去是对的（但实际上并非如此），所以别弄错了。



从`doEndTag()`返回`SKIP_PAGE`就像从简单标记抛出`SkipPageException`一样！如果一个页面包含了另一个页面，而后者调用了标记，则当前页面（被包含页面）会停止处理，但包含页面（外层页面）仍继续……

# IterationTag允许体重复执行

编写一个扩展 TagSupport 的标记处理器时，会从 Tag 接口得到所有生命周期方法，另外还能得到 IterationTag 的一个方法——doAfterBody()。如果没有 doAfterBody()，就无法循环处理体，因为 doStartTag() 调用得太早，而 doEndTag() 又太晚。而利用 doAfterBody()，返回值会告诉容器应该再次重复执行体(EVAL\_BODY\_AGAIN)，还是调用 doEndTag() 方法(SKIP\_BODY)。



对应以下 SimpleTag doTag()，尝试在一个传统标记处理器中实现同样的功能。假设 TLD 配置为允许有体内容。

```

public void doTag() throws JspException, IOException {
    String[] movies = {"Spiderman", "Saved!", "Amelie"};
    for(int i = 0; i < movies.length; i++) {
        getJspContext().setAttribute("movie", movies[i]);
        getJspBody().invoke(null);
    }
}
  
```

```

// package 和 import 语句
public class MyIterationTag extends TagSupport {

    public int doStartTag() throws JspException {
        // 处理 start 标记逻辑
    }

    public int doAfterBody() throws JspException {
        // 处理 body 逻辑
    }

    public int doEndTag() throws JspException {
        // 处理 end 标记逻辑
    }
}
  
```

## 作为容器



以下是合法的标记处理器代码，判断它能否给出所示的结果，在此给定以下列出的JSP标记。这也是前一页上ClassicTag处理器生成的结果。不错，我们正是用一个新练习来作为前面练习的答案……

### 标记处理器类

```
// package和import语句
public class MyIteratorTag extends TagSupport {
    String[] movies= new String[] {"Spiderman", "Saved!", "Amelie"};
    int movieCounter;

    public int doStartTag() throws JspException {
        movieCounter=0;

        return EVAL_BODY_INCLUDE;
    }
    public int doAfterBody() throws JspException {

        if (movieCounter < movies.length) {
            pageContext.setAttribute("movie", movies[movieCounter]);
            movieCounter++;
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }
    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }
}
```

### 调用标记的JSP

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
<table border="1">
<mine:iterateMovies>
<tr><td>${movie}</td></tr>
</mine:iterateMovies>
</table>
</body></html>
```

### 所需的结果



# TagSupport的默认返回值

如果没有覆盖返回一个整数的TagSupport生命周期方法，要当心TagSupport方法实现返回的默认值。TagSupport类会认为你的标记没有体（从doStartTag()返回SKIP\_BODY），另外如果确实有要计算的体，它认为你希望只计算一次（从doAfterBody()返回SKIP\_BODY）。它还假设你希望计算页面的余下部分（从doEndtag()返回EVAL\_PAGE）。

如果未覆盖TagSupport的方法实现，TagSupport的默认返回值。

**doStartTag()**

SKIP\_BODY

EVAL\_BODY\_INCLUDE

**doAfterBody()**

SKIP\_BODY

EVAL\_BODY\_AGAIN

**doEndTag()**

SKIP\_PAGE

EVAL\_PAGE

TagSupport类认为你的  
标记没有体，或者如果  
要计算体，这个体也只  
计算一次。

它还认为你总是希望计  
算页面的余下部分。



doStartTag()和doEndTag()  
只运行一次。

要参加考试，你必须很了解这个生命周期。不要忘了无论具体情况是怎样的，总会调用doStartTag()和doEndTag()，而且它们只调用一次。但是doAfterBody()可以运行0到多次，这取决于doStartTag()和上一次doAfterBody()调用的返回值。



如果希望计算标记体，就必须覆盖  
doStartTag()!!

先考虑一下！doStartTag()的默认返回值是SKIP\_BODY，所以如果你希望计算标记体，而且扩展了TagSupport，起码为了返回一个EVAL\_BODY\_INCLUDE也必须覆盖doStartTag()。

显然，如果你想循环运行体，同样必须覆盖doAfterBody()方法，因为它的默认返回值是SKIP\_BODY。

# 作为容器答案



所需的结果



实际结果（除非增加以下突出显示的两行）



## 标记处理器类

```

public class MyIteratorTag extends TagSupport {
    String[] movies= new String[] {"Spiderman", "Saved!", "Amelie"};
    int movieCounter;

    public int doStartTag() throws JspException {
        movieCounter=0;

        pageContext.setAttribute("movie", movies[movieCounter]);
        movieCounter++;

        return EVAL_BODY_INCLUDE;
    }

    public int doAfterBody() throws JspException {
        if (movieCounter < movies.length) {
            pageContext.setAttribute("movie", movies[movieCounter]);
            movieCounter++;
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }
    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }
}

```

必须增加这两行才能生成正确的响应。



这个doAfterBody()方法是正确的，  
是它只在已经处理一次体之后才执行；  
如果不在doStartTag()里加上这两行，  
就会在还没有movie属性的时候  
处理一次体，这样就会得到一个空的  
单元格。

## 调用标记的JSP

```

<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
<table border="1">
<mine:iterateMovies>
    <tr><td>${movie}</td></tr>
</mine:iterateMovies>
</table></body></html>

```

## there are no Dumb Questions

**问：** doStartTag() 和 doAfterBody() 中有重复的代码，这不是很傻吗？

**答：** 是的，是有一些重复的代码。在这个例子中，如果你实现了 TagSupport，而且想设置体能用的值，就必须在doStartTag()中设置这些属性值。不能等到doAfterBody()再设置，因为等到运行doAfterBody()时，体已经处理一次了。

确实，这是有点傻。所以我们说 SimpleTag 要好得多。当然，也可以这样做，写代码时在标记处理器中建立一个私有方法，例如setMovie()，让 doStartTag() 和 doAfterBody() 都调用这个方法。但是这种办法还是不太好。

**问：** 为什么在doStartTag()方法里设置movieCounter实例变量值？为什么不在声明这个变量的时候就初始化呢？

**答：** 呀！这可不像SimpleTag处理器，SimpleTag处理器绝对不会重用，而传统标记处理器可能放在缓冲池里，并由容器重用。这说明，你最好为每个新的标记调用（也就是在doStartTag()中）重置你的实例变量值。否则，这个代码只在第一次能正常工作，以后JSP调用它时，movieCounter变量会是上一次的值，而不是0！



容器可以重用传统标记处理器！

当然，这与SimpleTag处理器截然不同，简单标记处理器绝对不会重用。这说明，你必须特别小心实例变量，要在doStartTag()中重置这些实例变量。Tag接口确实有一个release()方法，但是这个方法只在标记处理器实例要被容器删除时才会调用。所以不要认为可以用release()在标记调用之间重置标记处理器的状态！



## OK, 现在来点真格的……

还记得第3章的啤酒Web应用吗？现在来稍加改进，自动完成HTML表单的一部分：

```
<form method="POST" action="SelectBeer.do">
<p>Select beer characteristics:</p>

Color:
<select name='color' size='1' >
    <option value='light'> light </option>
    <option value='amber'> amber </option>
    <option value='brown'> brown </option>
    <option value='dark'> dark </option>
</select>

<br><br>
<input type="SUBMIT">
</form>
```

我们希望这个<select>标记中选项的设置由应用完成。

如果这些选项作为动态选项，更新和修改将更为容易，而不用调整HTML。实际上，我们希望这些选项由web应用中创建的一个Java **List**生成。为此，以下是我们希望构建的定制标记：

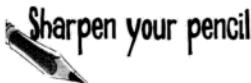
```
<form method="POST" action="SelectBeer.do">
<p>Select beer characteristics:</p>

Color:
<formTags:select name='color' size='1'
    optionsList='${applicationScope.colorList}' />

<br><br>
<input type="SUBMIT">
</form>
```

我们的定制标记将生成选项列表。标记的name和size属性都是直传值。

利用这个标记，应用可以修改选项而无需在HTML表单中硬编码写入业务数据。



你的任务（如果你愿意接受这个任务）是完成这个 select 标记处理器的实现。

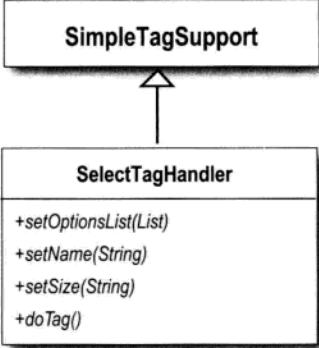
首先，处理器类需要为每个标记属性实现设置方法；以下是一个框架代码，你可以以此作为起点：

```
package com.example.taglib;
// 假设这里是所需的全部import语句
```

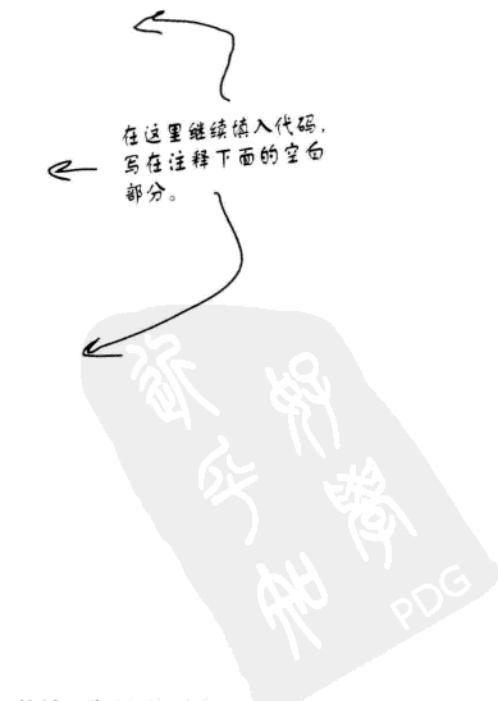
```
public class SelectTagHandler extends SimpleTagSupport {
    // 存储'optionsList'属性
```

```
// 存储'name'属性
```

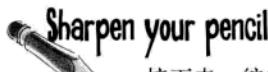
```
// 存储'size'属性
```



在这里继续填入代码。  
写在注释下面的空白部分。



待续，翻开下一页 →



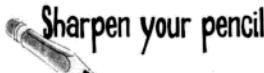
接下来，编写doTag()方法来完成select标记处理器类的实现。

我们已经提供了这个方法的签名，还有一些有用的注释可以提供帮助。不要忘了好好看看542页上需要由这个标记生成的HTML。

```
// 生成<select>和<option>标记
public void doTag() throws JspException, IOException {
    PageContext pageContext = (PageContext) getJspContext();
    JspWriter out = pageContext.getOut();
    // 开始HTML <select>标记, 首先是HTML特定的属性
    // 还要在这里,
    // 这里……以
    // 及这里增加
    // 更多代码。
    // 从optionsList生成<option>标记
    // 结束HTML </select>标记
} // // doTag()方法结束
} // SelectTagHandler结束
```

如果SelectTagHandler中  
需要额外的变量或常量，可  
以增加到这里。

待续，请看下一页 —————



现在需要在TLD文件中配置这个select标记。这里已经提供了TLD的“样板”元素。只需增加适当的元素来声明select标记、其处理器类以及它的所有属性。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.2</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>Forms Taglib</short-name>
    <uri>http://example.com/tags/forms</uri>
    <description>
        An example tag library of replacements for the HTML form tags.
    </description>
    <tag>
        <!-- 增加元素来声明标记名、类和体类型 -->
        <!-- 增加元素来声明optionsList属性 -->
        <!-- 增加元素来声明name属性 -->
        <!-- 增加元素来声明size属性 -->
    </tag>
</taglib>
```





你的任务（如果你愿意接受这个任务）是完成这个select标记处理器的实现。处理器必须实现各个标记属性的设置方法，还必须实现doTag()方法。

```
package com.example.taglib;
// 假设这里是所需的全部import语句
```

```
public class SelectTagHandler extends SimpleTagSupport {
```

```
    private List optionsList;
    // 存储'optionsList'属性
    public void setOptionsList(List value) {
        this.optionsList = value;
    }
```

← optionsList 属性的设置方法和实例变量。

```
    private String name;
    // 存储'name'属性
    public void setName(String value) {
        this.name = value;
    }
```

← name 属性的设置方法和实例变量。

```
    private String size;
    // 存储'size'属性
    public void setSize(String value) {
        this.size = value;
    }
```

← size 属性的设置方法和实例变量。

```
// 其他SelectTagHandler代码
}
```

**SimpleTagSupport**

**SelectTagHandler**

```
+setOptionsList(List)
+setName(String)
+setSize(String)
+doTag()
```



# 解决方案

接下来，编写doTag()方法来完成select标记处理器类的实现。

以下是我们使用的代码：

```
// 生成<select>和<option>标记
public void doTag() throws JspException, IOException {
    PageContext pageContext = (PageContext) getJspContext();
    JspWriter out = pageContext.getOut();

    HTML <select>
    开始标记使用了
    name和size属性。
    { // 开始HTML <select>标记, 首先是HTML特定的属性
        out.print("<select ");
        out.print(String.format(ATTR_TEMPLATE, "name", this.name));
        out.print(String.format(ATTR_TEMPLATE, "size", this.size));
        out.println('>');

        // 从optionsList生成<option>标记
        for ( Object option : this.optionsList ) {
            String optionTag
                = String.format(OPTION_TEMPLATE, option.toString());
            out.println(optionTag);
        }

        // 结束HTML </select>标记
        out.println(" </select>");
    } // // doTag()方法结束

    private static final String ATTR_TEMPLATE = "%s='%s' ";
    private static final String OPTION_TEMPLATE
        = "<option value='%s'> %s </option>";
} // SelectTagHandler结束
```

*optionsList对象用于创建 HTML <option>标记。*

*最后, 标记处理器必须输出结束HTML </select>标记。*

*我们的实现使用了一些 String常量, 使代码更可读。*

# Sharpen your pencil

## 解决方案

接下来，必须在TLD文件中配置select标记。为增加元素来声明select标记，其处理器类和所有属性，我们的做法如下。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>
  <tlib-version>1.2</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Forms Taglib</short-name>
  <uri>http://example.com/tags/forms</uri>
  <description>
    An example tag library of replacements for the HTML form tags.
  </description>

  <tag>
    <name>select</name>          声明标记的名、类
    <tag-class>com.example.taglib.SelectTagHandler</tag-class>← 和体类型。
    <body-content>empty</body-content>

    <attribute>
      <name>optionsList</name>          ← optionsList属性需要指定数
      <type>java.util.List</type>        据类型。这个属性还必须允
      <required>true</required>          许值中出现运行时表达式。
      <rtexprvalue>true</rtexprvalue>
    </attribute>

```



# Sharpen your pencil

## 解决方案

```
<attribute>
  <name>name</name> ←
  <required>true</required>
</attribute>

<attribute> ←
  <name>size</name>
  <required>true</required>
</attribute>

</tag>

</taglib>
```

name和size属性要简单得多，  
因为我们可以接受默认的数据类型(String)。



**BRAIN  
POWER**

你认为name和size属性应当允许运行时值吗？请解释  
为什么允许或为什么不允许？

## 我们的动态`<select>`标记还不完整……



请等等……如果我们的`select`标记想要模拟标记HTML`<select>`标记，那么需要对应`<select>`标记的所有属性在这个`select`标记中包含相应的属性，而不仅仅是`name`和`size`。

HTML`<select>`标记还接受其他更多标记属性，而不只是`name`和`size`：

核心属性：`id`、`class`、`style`和`title`

国际化属性：`lang`和`dir`

事件属性：`onclick`、`ondblclick`、`onmouseup`、`onmousedown`、  
`onmouseover`、`onmousemove`、`onmouseout`、`on-  
keypress`、`onkeyup`和`onkeydown`

表单属性：`name`、`disabled`、`multiple`、`size`、`tabindex`、`on-  
focus`、`onblur`和`onchange`

可以使用这些  
建立列表框和  
列表菜单。



拜托！如果没有事件  
属性，我怎么增加  
那些酷酷的效果？

如果不能应用样式，  
我的水平根本无法发  
挥。

别这么让我束手束  
脚……我需要建立列表框，  
还要建立列表菜单。



# 只需增加更多定制标记属性……



不要着急，这是可以解决的……  
绝对没问题！我只需为处理器类增加更多属性设置方法，并为TLD增加声明。完全不用慌张。

Gary的设计非常简单：需要为所有HTML直传标记属性分别增加一个设置方法。右边给出了这个标记类的UML类图，其中包含需要增加的所有方法。

以下是完成这个工作的代码：

```
public class SelectTagHandler extends SimpleTagSupport {
    // 标记属性(设置方法和实例变量)
    public void setOptionsList(List value) {
        this.optionsList = value;
    }
    private List optionsList = null;           ← 这是我们为select标记
                                              唯一增加的属性。
    public void setId(String id) {
        this.id = id;
    }
    private String id;

    public void setClass(String styleClass) {
        this.styleClass = styleClass;
    }

    private String styleClass;
    // 更多代码见下一页
}
```

## SelectTagHandler

```
+setOptionsList(List)
+setId(String)
+setClass(String)
+setStyle(String)
+setTitle(String)
+setLang(String)
+setDir(String)
+setOnclick(String)
+setOndblclick(String)
+setOnmouseup(String)
+setOnmousedown(String)
+setOnmouseover(String)
+setOnmousemove(String)
+setOnmouseout(String)
+setOnkeypress(String)
+setOnkeydown(String)
+setOnkeyup(String)
+setName(String)
+setSize(String)
+setMultiple(String)
+setDisabled(String)
+setTabindex(String)
+setOnfocus(String)
+setOnblur(String)
+setOnchange(String)
+doTag()
```

其余标记属性都针对web浏览器。这个标记处理器只是把它们直接传递到<select>标记输出。

## 更多标记属性（续）

```

public void setStyle(String style) {
    this.style = style;
}
private String style;

public void setTitle(String title) {
    this.title = title;
}
private String title;

public void setLang(String lang) {
    this.lang = lang;
}
private String lang;

public void setDir(String dir) {
    this.dir = dir;
}
private String dir;

public void setOnclick(String onclick) {
    this.onclick = onclick;
}
private String onclick;

public void setOndblclick(String ondblclick) {
    this.ondblclick = ondblclick;
}
private String ondblclick;

public void setOnmouseup(String onmouseup) {
    this.onmouseup = onmouseup;
}
private String onmouseup;

public void setOnmousedown(String onmousedown) {
    this.onmousedown = onmousedown;
}
private String onmousedown;

public void setOnmouseover(String onmouseover) {
    this.onmouseover = onmouseover;
}
private String onmouseover;

// 更多代码见下一页

```

更多HTML <select>标记属性：其中一些是核心属性，还有一些是事件处理器属性。

不过请等等！除此以外还有一些属性。以上只为24个HTML属性中的11个编写了代码。下一页将列出另外一部分标记属性设置方法。

# 更多标记属性（再续）

没错，就是这样……  
还有更多属性。

```

public void setOnmousemove(String onmousemove) {
    this.onmousemove = onmousemove;
}
private String onmousemove;

public void setOnmouseout(String onmouseout) {
    this.onmouseout = onmouseout;
}
private String onmouseout;

public void setOnkeypress(String onkeypress) {
    this.onkeypress = onkeypress;
}
private String onkeypress;

public void setOnkeydown(String onkeydown) {
    this.onkeydown = onkeydown;
}
private String onkeydown;

public void setOnkeyup(String onkeyup) {
    this.onkeyup = onkeyup;
}
private String onkeyup;

public void setName(String value) {
    this.name = value;
}
private String name;
public void setSize(String value) {
    this.size = value;
}
private String size;

public void setMultiple(String multiple) {
    this.multiple = multiple;
}
private String multiple;

public void setDisabled(String disabled) {
    this.disabled = disabled;
}
private String disabled;

// 下一页还有更多代码

```



## 这些标记属性真让人受够了！

```
public void setTabindex(String tabindex) {  
    this.tabindex = tabindex;  
}  
private String tabindex;  
  
public void setOnfocus(String onfocus) {  
    this.onfocus = onfocus;  
}  
private String onfocus;  
  
public void setOnblur(String onblur) {  
    this.onblur = onblur;  
}  
private String onblur;  
  
public void setOnchange(String onchange) {  
    this.onchange = onchange;  
}  
private String onchange;  
  
// 生成<select>和<option>标记  
public void doTag() throws JspException, IOException {  
    PageContext pageContext = (PageContext) getJspContext();  
    JspWriter out = pageContext.getOut();  
    // 开始HTML <select> 标记, 首先是HTML特定的属性  
    out.print("<select ");  
    // 增加必要属性  
    out.print(String.format(ATTR_TEMPLATE, "name", this.name));  
    // 增加可选属性  
    if (this.id != null)  
        out.print(String.format(ATTR_TEMPLATE, "id", this.id));  
    if (this.styleClass != null)  
        out.print(String.format(ATTR_TEMPLATE, "class", this.styleClass));  
    if (this.style != null)  
        out.print(String.format(ATTR_TEMPLATE, "style", this.style));  
    if (this.title != null)  
        out.print(String.format(ATTR_TEMPLATE, "title", this.title));  
    if (this.lang != null)  
        out.print(String.format(ATTR_TEMPLATE, "lang", this.lang));  
    if (this.dir != null)  
        out.print(String.format(ATTR_TEMPLATE, "dir", this.dir));
```

……谢天谢地，至此标记属性设置方法终于结束了。

不过，不要就此止步。还有更多代码。`doTag()`方法还必须把各个标准HTML  
<select>标记属性写至响应流。除了以上属性，另外还有17个属性，所以还需要  
34行代码，至少还要占一页半。不仅占用篇幅，而且也不大好看……



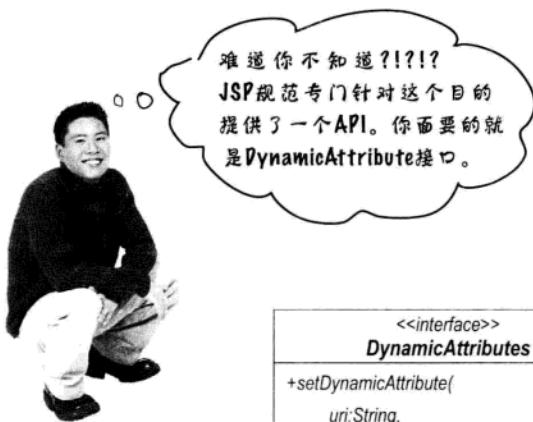
你说的没错。这种解决方案很糟糕。这样做会有太多的代码。更糟的是，如果我们希望创建一组定制标记来扩展其他HTML标记呢？！

标记处理器类必须为TLD中声明的每个标记属性实现一个设置方法。但是这些设置方法所做的可能没有多大意义。这些属性的价值就是要传递到为HTML<select>标记生成的输出中。

对此可以应用一个设计原则：“封装有变化的部分”<sup>\*</sup>。在这里，HTML标记可选属性的设置就是这个标记处理器中变化的部分。一种解决方案是将所有属性放入一个散列表（hashtable）。这样可以泛化标记对象属性的存储，但是所有这些设置方法呢？我们无法把它们去掉，除非有办法告诉JSP引擎使用一种泛型接口设置标记属性。

\* 《Head First Object-Oriented Analysis and Design》一书的250页上有关于这种设计原则的讨论。

当然，我们绝对不会无耻地随便向你推荐其他与此无关的Head First书，对不对？



```
<<interface>>  
DynamicAttributes  
+setDynamicAttribute(  
    uri:String,  
    name:String,  
    value:Object) : void
```

**SimpleTagSupport**

**SelectTagHandler**

```
-optionsList>List  
-tagAttrs:Map<String, Object>  
+setOptionsList(List)  
+setDynamicAttribute(  
    uri:String,  
    name:String,  
    value:Object) : void  
+doTag()
```

最有可能将动态属性  
存储在一个hashmap中。

这个设置方法用于每一个动态属性。`name`参数是  
属性名，`value`参数是属性值。`uri`参数是定义该属  
性的XML命名空间。通常可以忽略这个参数。



**Relax** 考试中不会考这个方法签  
名，而且绝对不会考uri参  
数的作用。

该死，我们自己都不知道这个参数作  
么用。

# 标记处理器代码使用DynamicAttributes接口

下面来分析**DynamicAttributes**如何具体使用。首先，我们的标记处理器类必须实现JSP API的DynamicAttributes接口。这个接口要求必须实现setDynamicAttribute()方法。这个方法需要存储属性名/值对；而存储这个信息最佳的数据结构就是hashmap：

```

package com.example.taglib;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.DynamicAttributes;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 * Version three of the HTML select tag uses the JSP
 * dynamic attributes mechanism to store all of the
 * pass-through HTML attributes in a hashmap.
 */
public class SelectTagHandler
    extends SimpleTagSupport
    implements DynamicAttributes { ← 标记处理器必须实现
        // 存储'optionsList'属性 DynamicAttributes接口。
        public void setOptionsList(List value) {
            this.optionsList = value;
        }
        private List optionsList = null;

        // 存储'name'属性
        public void setName(String value) {
            this.name = value;
        }
        private String name;

        // 存储其他(动态)属性
        public void setDynamicAttribute(String uri, String name, Object value) {
            tagAttrs.put(name, value);
        }
        private Map<String, Object> tagAttrs = new HashMap<String, Object>();
    }
}

```

通常，设置方法只是将各个属性名/值对存储在一个hashmap中。

## 标记处理器其余代码

只剩下 doTag() 方法了。现在唯一的区别是，标准 HTML<select>标记属性的生成现在存储在 hashmap 中。doTag() 方法必须迭代处理这个映射中的每一项，并生成绑定到输出流的 HTML 属性。其他工作都完全相同。

是不是很简单？

```
// 生成<select>和<option>标记
public void doTag() throws JspException, IOException {
    PageContext pageContext = (PageContext) getJspContext();
    JspWriter out = pageContext.getOut();

    // 开始HTML <select>标记
    out.print("<select ");

    // 增加必要属性
    out.print(String.format(ATTR_TEMPLATE, "name", this.name));

    // 增加动态属性
    for ( String attrName : tagAttrs.keySet() ) { ←
        String attrDefinition
            = String.format(ATTR_TEMPLATE,
                attrName, tagAttrs.get(attrName));
        out.print(attrDefinition); ←
    }
    out.println('>');

    // 从optionsList生成<option>标记
    for ( Object option : optionsList ) {
        String optionTag
            = String.format(OPTION_TEMPLATE, option.toString());
        out.println(optionTag);
    }

    // 结束HTML </select>标记
    out.println(" </select>");
} // doTag()方法结束

private static final String ATTR_TEMPLATE = "%s='%" + "s' ";
private static final String OPTION_TEMPLATE
= " <option value='%" + "s'> %" + "s </option>";
} // SelectTagHandler结束
```

从映射键获取属性集。各个  
键就是某个动态属性的名。

属性值存储在映射中。get()方法  
从键（属性名）获取值。

# K, 再在TLD中完成一点配置

合! 你不会认为只是需要编写代码吧? 当然了, 还需要一个配置元素。注意了, 这里所说的是JSP规范。幸运的是变化不大。需要包含的元素名为`<dynamic-attributes>`:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>

    <tlib-version>1.2</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>Forms Taglib</short-name>
    <uri>http://example.com/tags/forms</uri>
    <description>
        An example tag library of replacements for the HTML form tags.
    </description>

    <tag>
        <name>select</name>
        <tag-class>com.example.taglib.SelectTagHandler</tag-class>
        <body-content>empty</body-content>
        <description>
            This tag constructs an HTML form 'select' tag. It also generates
            the 'option' tags based on the set of items in a list passed in
            by the optionsList tag attribute.
        </description>
        <attribute>
            <name>optionsList</name>
            <type>java.util.List</type>
            <required>true</required>
            <rteprvalue>true</rteprvalue>
        </attribute>
        <attribute>
            <name>name</name>
            <required>true</required>
        </attribute>
        <dynamic-attributes>true</dynamic-attributes>
    </tag>
</taglib>

```

还需要声明所有必要属性。  
对应这些属性，在标记处理器中必须有明确的设置方法。

只需这样一个元素来声明这个标记可以接受任意数目的动态属性。

*there are no  
Dumb Questions*

**问：** 你用的是一个简单标记。对传统标记也适用吗？

**答：** 是的，传统标记可以采用与简单标记完全相同的方式实现DynamicAttributes接口。甚至TLD文件中的配置也完全一样。

**问：** 动态属性接受运行时表达式吗，比如说EL或`<%= %>`？

**答：** 当然了。默认地，每个动态属性都可以使用EL或JSP表达式标记来指定属性的值。实际上，你注意到没有？`setDynamicAttribute()`方法的**value**参数的数据类型是**Object**而不是**String**。这说明这个值可以计算为任何Java对象。

**问：** 如果我需要对一个给定动态属性中的数据进行“计算”该怎么做？

**答：** 可以检查**name**参数，决定对该属性的值完成某种计算或转换。不过，如果你需要这种功能，可能要将这个属性置为显式属性，并在该属性的设置方法中完成计算。

**问：** 如果定制标记用户输入一个非法的属性名会怎么样呢？

**答：** 这个问题问得太好了。因为属性名并未在TLD中显式声明，JSP引擎会使用`setDynamicAttribute()`方法将所有其他属性发送给标记处理器。其结果是，JSP开发人员可能把某个标准HTML属性名写错了，而他永远也不知道这一点（至少等到浏览器无法正常调用该属性的行为时才有可能发现）。所以，Gary提出的第一个解决方案（使用带设置方法和TLD声明的显式属性）还是有优点的。你能想出还有哪些原因能够解释Gary的解决方案要胜过Kim的方案？



开动脑筋

Gary的解决方案把所有属性都置为显式属性。Kim的解决方案则把大部分属性置为动态属性。这两种方案各有利弊。还有没有另外的解决方案？

# 标记文件呢？

标记文件也可以包含动态属性。其原理基本上相同，不过对于标记文件，JSP引擎会为你提供Map对象，然后你可以使用forEachJSTL标记检查或迭代处理这个属性/值对的映射。

*dynamic-attributes属性的值是一个页面作用域变量，其中包含一个hashmap。*

```
<%@ tag body-content='empty' dynamic-attributes='tagAttrs' %>
<%@ attribute name='optionsList' type='java.util.List'
   required='true' rtexprvalue='true' %>
<%@ attribute name='name' required='true' %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<select name='${name}'>
  <c:forEach var="attrEntry" items="${tagAttrs}">
    ${attrEntry.key}='${attrEntry.value}'</c:forEach>
  <c:forEach var="option" items="${optionsList}">
    <option value='${option}'> ${option} </option>
  </c:forEach>
</select>
```

*使用JSTL forEach定制标记迭代处理动态属性hashmap中的各项。要记住，各项的键是属性名，项的值就是属性值。*

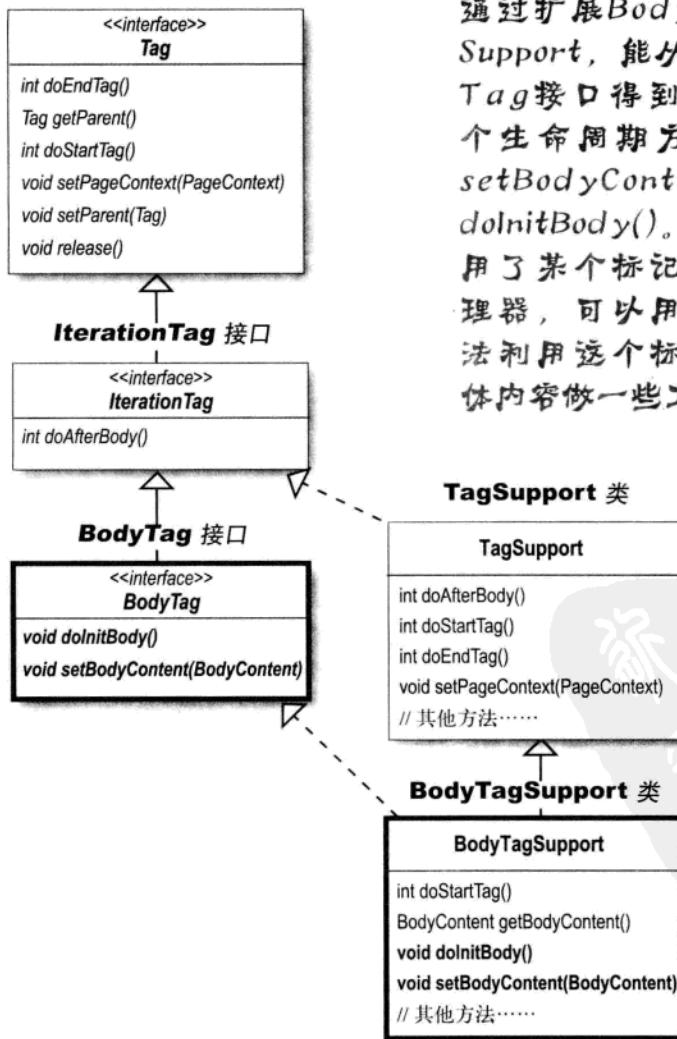
## 要点

- DynamicAttributes接口允许标记处理器类接受任意数目的标记属性。
- TLD中的标记声明必须包含`<dynamic-attributes>`元素。
- 显式标记属性必须有一个设置方法。
- 一般会使用`setDynamicAttribute()`方法利用一个hashmap存储动态属性名/值对。
- 标记文件也可以使用动态属性。
- 标记要使用tag指令的`dynamic-attributes`属性。
- `dynamic-attributes`的值包含动态属性的一个hashmap。
- 一般地，会使用JSTL `forEach`定制动作迭代处理这个映射。

## 如果确实需要访问体内容呢？

你可能会发现，大多数情况下，Tag和IterationTag接口的生命周期方法（通过TagSupport提供）已经足够了。利用3个关键的方法（doStartTag()、doAfterBody()和doEndTag()），几乎可以做任何事情。

但是……你无法直接访问体内容。如果能访问具体的体内容，你就可以做很多事情了，比如在表达式中使用体内容，或者以某种方式过滤或修改。为此可以扩展BodyTagSupport而不是TagSupport，这样你就能访问BodyTag接口方法。

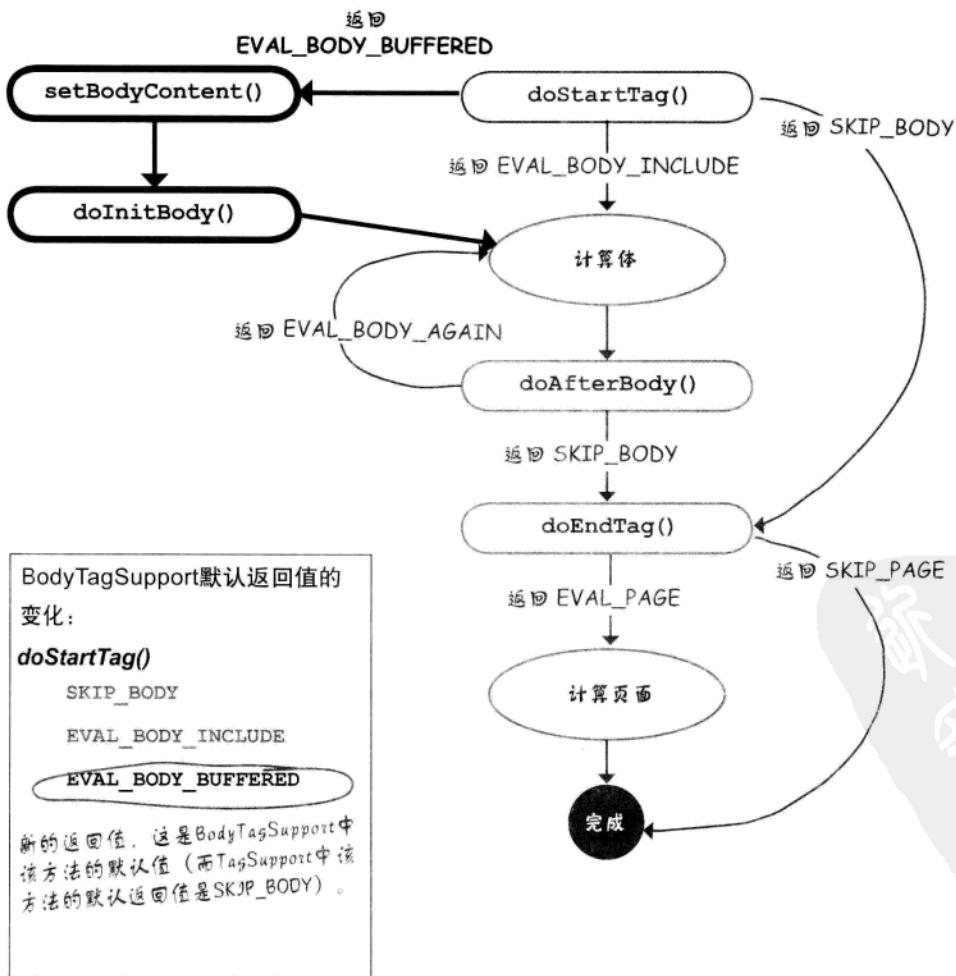


通过扩展BodyTagSupport，能够从BodyTag接口得到另外两个生命周期方法——setBodyContent()和doInitBody()。如果使用了某个标记调用处理器，可以用这些方法利用这个标记的具体内容做一些工作。

# 基于BodyTag，你会得到两个新方法

实现BodyTag时（通过扩展BodyTagSupport），你能得到另外两个生命周期方法——setBodyContent()和doInitBody()。另外，doStartTag()还可以有一个新的返回值：EVAL\_BODY\_BUFFERED。这说明，现在doStartTag()有三个可能的返回值，而扩展TagSupport时只有两个可能的返回值。

实现BodyTag（直接实现或扩展BodyTagSupport）的标记的生命周期。



## 利用BodyTag可以缓存体

setBodyContent()的BodyContent参数实际上是一种java.io.Writer（确实，从面向对象的角度看有点费解）。不过，这说明可以采用某种方式处理体，比如说把体链到另一个IO流或得到原始字节来加以处理。

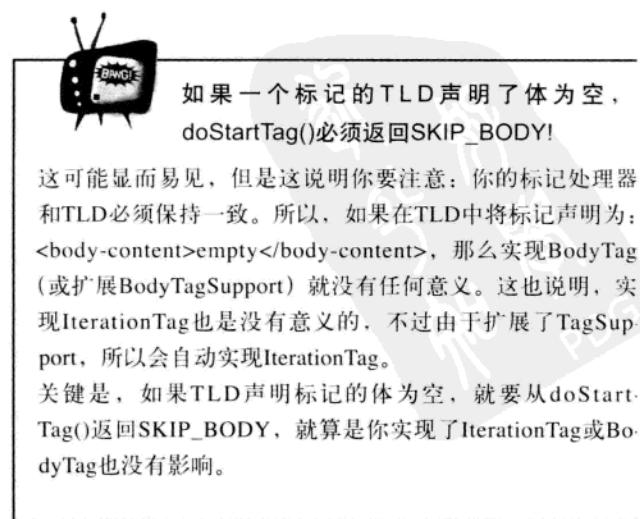
**问：**即使调用标记为空，如果仍返回EVAL\_BODY\_BUFFERED会发生什么？

**答：**如果调用处理器的标记为空，setBodyContent()和doInitBody()方法不会调用！所谓标记为空，是指使用一个空标记<my:tag />来调用，或者开始和结束标记<my:tag></my:tag>之间没有内容。容器知道这一次没内容，它就会跳到doEndTag()方法，所以这通常不是问题。

除非TLD声明标记的体为空！如果TLD声明<body-content>empty</body-content>，那你别无选择，不能从doStartTag()返回EVAL\_BODY\_BUFFERED或EVAL\_BODY\_INCLUDE。

**问：**传统标记中的属性呢？也用简单标记中同样的办法处理吗？

**答：**是的，在简单标记处理器和传统标记处理器的顺序图中，都有一个地方要为每个属性调用一个bean式设置方法。这发生在简单标记的doTag()或传统标记的doStartTag()中。换句话说，在传统标记和简单标记中，标记属性的处理是一样的，而且也以同样的方式在TLD中声明。





### 传统标记处理器的生命周期方法

填写下表。这里已经涵盖了要求你掌握的几乎所有内容，不过，有些地方你可能还得猜猜看（不要翻到下一页）！

	BodyTagSupport	TagSupport
doStartTag() 可能的返回值		
实现类的默认返回值		
可以调用的次数（对于JSP中的 每个标记调用）		
doAfterBody() 可能的返回值		
实现类的默认返回值		
可以调用的次数（对于JSP中的 每个标记调用）		
doEndTag() 可能的返回值		
实现类的默认返回值		
可以调用的次数（对于JSP中的 每个标记调用）		
doInitBody()和 setBodyContent() 在哪些情况下可以调用，每个 标记调用的调用次数		



## 传统标记方法的生命周期返回值

要参加考试，这些都必须掌握！

## 答案

	BodyTagSupport	TagSupport
doStartTag() 可能的返回值	SKIP_BODY EVAL_BODY_INCLUDE EVAL_BODY_BUFFERED	SKIP_BODY EVAL_BODY_INCLUDE
实现类的默认返回值	EVAL_BODY_BUFFERED	SKIP_BODY
可以调用的次数（对于JSP中的每个标记调用）	仅一次	仅一次
doAfterBody() 可能的返回值	SKIP_BODY EVAL_BODY_AGAIN	SKIP_BODY EVAL_BODY_AGAIN
实现类的默认返回值	SKIP_BODY	SKIP_BODY
可以调用的次数（对于JSP中的每个标记调用）	0到多次	0到多次
doEndTag() 可能的返回值	SKIP_PAGE EVAL_PAGE	SKIP_PAGE EVAL_PAGE
实现类的默认返回值	EVAL_PAGE	EVAL_PAGE
可以调用的次数（对于JSP中的每个标记调用）	仅一次	仅一次
doInitBody()和 setBodyContent() 在哪些情况下可以调用，每个 标记调用的调用次数	仅一次，而且仅当doStartTag()返回EVAL_BODY_BUFFERED	从不调用！

# 如果有些标记要一同工作呢？

假设有这样一种情况……你有一个<my:Menu>标记，要建立一个定制导航条，它要有菜单项。所以你使用嵌在<my:Menu>标记中和<my:MenuItem>标记，菜单标记拥有（以某种方式）菜单项，并使用这些菜单项建立导航条。

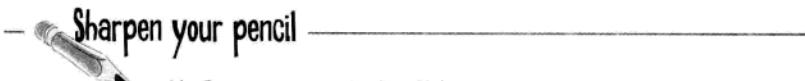
```
<mine:Menu >
  <mine:MenuItem itemValue="Dogs" />
  <mine:MenuItem itemValue="Cats" />
  <mine:MenuItem itemValue="Horses" />
</mine:Menu>
```

← Menu标记需要从嵌入的  
MenuItem标记得到属性值……

问题是，标记怎么相互对话呢？换句话说，Menu标记（外围标记）怎么从MenuItems（内部/嵌套标记）得到属性值？

在JSTL中多处用到了嵌套标记；<c:choose>标记有嵌套的<c:when>和<c:otherwise>标记，这就是一个很好的例子。你可能也需要在你自己的定制开发中使用“合作标记”（规范就称之为“合作标记”）。

幸运的是，有一种在外标记和内标记之间传递信息的机制，而不论嵌套层次有多深。这说明，可以从一个嵌套很深的标记得到信息，这个信息可以提供给嵌套层次结构中的任意一个上层标记，而不只是其直接外层标记。



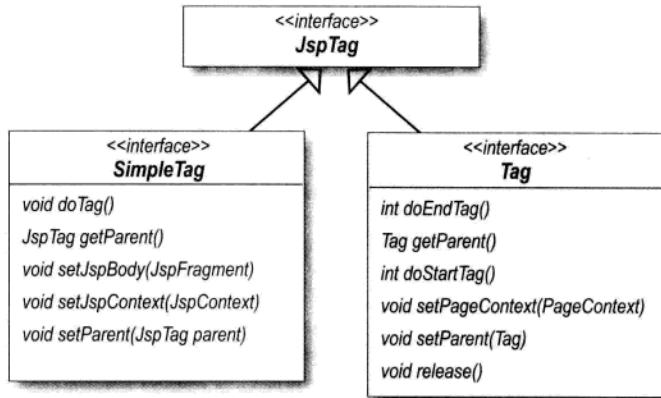
请看Tag API，回顾前面的标记处理器代码，考虑一下合作标记如何相互传递信息。

<b>&lt;&lt;interface&gt;&gt;</b> <b>Tag</b>
<code>int doEndTag()</code> <code>Tag getParent()</code> <code>int doStartTag()</code> <code>void setPageContext(PageContext)</code> <code>void setParent(Tag)</code> <code>void release()</code>



## 标记可以调用其父标记

SimpleTag和Tag都有一个getParent()方法。Tag中的getParent()会返回一个Tag，而SimpleTag的getParent()返回JspTag的一个实例。我们稍后再看这些返回类型的影响。



### 嵌套标记可以访问其父（外围）标记

```

<mine:OuterTag>
  <mine:InnerTag /> ← 在这个关系中，“OuterTag”
  是“InnerTag”的父标记。
</mine:OuterTag>
  
```

### 在传统标记处理器中得到父标记

```

public int doStartTag() throws JspException {
    OuterTag parent = (OuterTag) getParent();
    // 用它做一些处理
    return EVAL_BODY_INCLUDE;
}
  
```

↑ 不要忘记类型  
强制转换！

### 在简单标记处理器中得到父标记

```

public void doTag() throws JspException, IOException {
    OuterTag parent = (OuterTag) getParent();
    // 用它做一些处理
}
  
```

↑ 同样的，不要忘记类型强制转换。  
← 这与传统标记处理器中完全一样。

# 看看嵌套能有多深……

可以沿着祖先标记链一直往上走，只需继续对getParent()返回的标记再调用getParent()，这是因为getParent()可能返回另一个标记（可以对其调用getParent()），或者返回null。

在JSP中

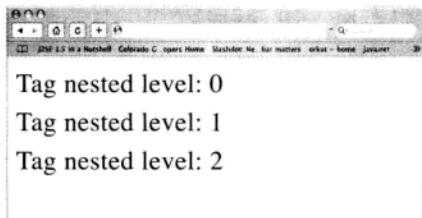
```
<mine:NestedLevel>
  <mine:NestedLevel>
    <mine:NestedLevel/>
  </mine:NestedLevel>
</mine:NestedLevel>
```

在传统标记处理器中

```
package foo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class NestedLevelTag extends TagSupport {
    private int nestLevel = 0;

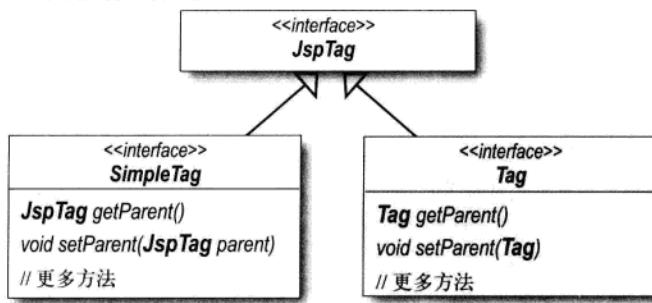
    public int doStartTag() throws JspException {
        nestLevel = 0;
        Tag parent = getParent(); ← 调用继承的getParent()方法。
        while (parent!=null) { ← 如果为null，则已经处在顶层，再没有父标记了。
            parent = parent.getParent();
            nestLevel++;
        }
        try {
            pageContext.getOut().println("<br>Tag nested level: " + nestLevel);
        } catch(IOException ex) {
            throw new JspException("IOException- " + ex.toString());
        }
        return EVAL_BODY_INCLUDE;
    }
}
```

结果



## 简单标记的父标记可以是传统标记

这不成问题，因为SimpleTag的getParent()返回类型是JspTag，而传统标记和简单标记现在都有JspTag超接口。实际上，传统标记可以有简单标记作为父标记，但是需要你稍做些努力才能让它正常工作，因为不能把SimpleTag强制转换为Tag接口getParent()的Tag返回值。我们不会过多地讨论如何从一个传统子标记访问简单父标记（注1），考试（以及实际Web应用开发）只要求你掌握以下内容：通过使用getParent()，传统标记可以访问传统父标记，简单标记既可以访问简单父标记，也可以访问传统父标记。



**使用 getParent() 方法，传统标记可以访问传统父标记，而简单标记既可以访问简单父标记，也可以访问传统父标记。**

### JSP中

```
<mine:ClassicParent name="ClassicParentTag">
  <mine:SimpleInner />
</mine:ClassicParent>
```

如果子标记 (SimpleInner) 想访问父标记的“name”属性呢？

### SimpleInner标记处理器中

```
public void doTag() throws JspException, IOException {
  MyClassicParent parent = (MyClassicParent) getParent();
  getJspContext().getOut().print("Parent attribute is: " + parent.getName());
}
```

SimpleTag访问传统父标记是可以的……

### ClassicParent标记处理器中

```
public class MyClassicParent extends TagSupport {
  private String name;
  public void setName(String name) {
    this.name=name;
  }
  public String getName() { ← 为属性提供一个获取方法，以便子
    return name;          标记得到属性值。
  }
  public int doStartTag() throws JspException {
    return EVAL_BODY_INCLUDE;
  }
}
```

如果返回SKIP\_BODY，就不会处理内标记！

一旦有了父标记，可以像其他Java对象一样对其调用方法，所以可以得到父标记的属性！

注1：如果你真的想知道，请查看J2EE 1.4 API中的TagAdapter类。

# 可以向上走，但是不能向下走……

有一个`getParent()`方法，而没有`getChild()`。但是前面所说的情况是：一个外层 `<my:Menu>` 标记需要访问其中嵌套的`<my:MenuItem>` 标记。我们该怎么做呢？子标记可以得到父标记的一个引用，但是父标记不能得到子标记的引用，那么父标记怎么得到子标记的信息呢？



## — Sharpen your pencil —

父标记怎么从子标记得到属性值？说明如何实现合作  
Menu和MenuItem标记的功能。

## 父标记从子标记获取信息

标记相互合作主要有两种途径：

- 1) 子标记需要从其父标记获得信息（如属性值）。
- 2) 父标记需要各个子标记的信息。

我们已经看到了第一种情况如何实现，子标记使用getParent()得到其父标记的一个引用，然后在父标记上调用获取方法。但是如果父标记想得到子标记的信息该怎么做呢？我们也要做同样的事情。换句话说，如果父标记需要子标记的信息，要由子标记把自己交给父标记！

因为没有自动机制让父标记找到他的子标记，所以只能使用同样的设计方法从子标记得到信息交给父标记，这与子标记从父标记得到信息的做法一样。你要得到父标记的一个引用，然后调用方法。只不过不是获取方法，这一次要调用某种设置或增加方法。

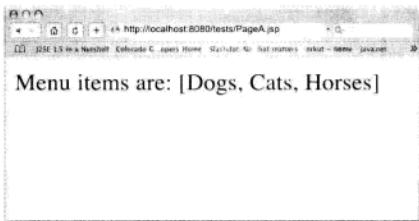
JSP中

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>

<mine:Menu >
    <mine:MenuItem itemValue="Dogs" />
    <mine:MenuItem itemValue="Cats" />
    <mine:MenuItem itemValue="Horses" />
</mine:Menu>

</body></html>
```

结果



在这个例子中，我们实际上没有对菜单项做什么，只是想证明我们确实得到了这些菜单项，但是你可以想想看还能怎么做，比如如何用这些菜单项建立一个导航条……

# Menu和MenuItem标记处理器

子标记MenuItem中：

```
public class MenuItem extends TagSupport {
    private String itemValue;

    public void setItemValue(String value) {
        itemValue = value;
    }

    public int doStartTag() throws JspException {
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException {
        Menu parent = (Menu) getParent();
        parent.addMenuItem(itemValue);
        return EVAL_PAGE;
    }
}
```

MenuItem的TLD中声明了一个属性 itemValue。我们需要把这个值发送给父标记……

← 很简单——得到父标记的引用，并调用它的addMenuItem()方法。

父标记Menu中：

```
public class Menu extends TagSupport {
    private ArrayList items;

    public void addMenuItem(String item) {
        items.add(item);
    }

    public int doStartTag() throws JspException {
        items = new ArrayList();
        ← 不要忘记要在doStartTag()中重置ArrayList。
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException {
        try {
            pageContext.getOut().println("Menu items are: " + items);
        } catch (Exception ex) {
            throw new JspException("Exception: " + ex.toString());
        }
        // 假设这里有一些复杂的构建菜单代码……
        return EVAL_PAGE;
    }
}
```

这不是属性设置方法！这个方法的作用就是让子标记告诉其父标记：这个子标记的属性值是什么（这个方法在doStartTag()和doEndTag()之间调用）。

← 如果没有返回EVAL\_BODY\_INCLUDE，  
因为容器可能会重用标记处理器。

← 如果没有返回EVAL\_BODY\_INCLUDE，  
就不会处理子标记！

## 得到一个任意的祖先

如果你愿意，还可以使用另外一种机制来跳过某些嵌套层次，直接到达某个祖父标记或标记嵌套层次结构中的更高层次。TagSupport和SimpleTagSupport中都有这个方法（不过表现稍有不同），这就是findAncestorWithClass()。

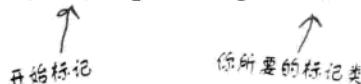
使用getParent()得到直接父标记

```
OuterTag parent = (OuterTag) getParent();
```

使用findAncestorWithClass()得到任意祖先

```
WayOuterTag ancestor = (WayOuterTag) findAncestorWithClass(this, WayOuterTag.class);
```

```
findAncestorWithClass(this, WayOuterTag.class);
```



容器会沿着嵌套层次结构查找，直到找到作为此类实例的一个标记。它会返回找到的第一个标记，所以这样说是不行的：“跳过你找到的第一个作为WayOuterTag.class实例的标记，给我下一个实例……”所以，如果你确实知道你想要该类型祖先标记的第二个实例，就必须得到findAncestorWithClass()的返回值，然后对它调用getParent()或findAncestorWithClass()。

我们不再讨论使用findAncestorWithClass()的细节。考试只要求你知道存在这个方法就行了！



## 简单和传统标记的重要区别

	简单标记	传统标记
Tag接口		
支持实现类		
可能实现的关键生命周期方法		
如何写至响应输出		
如何从一个支持实现访问隐式变量和作用域属性		
如何让体得到处理		
如果让当前页面停止计算		



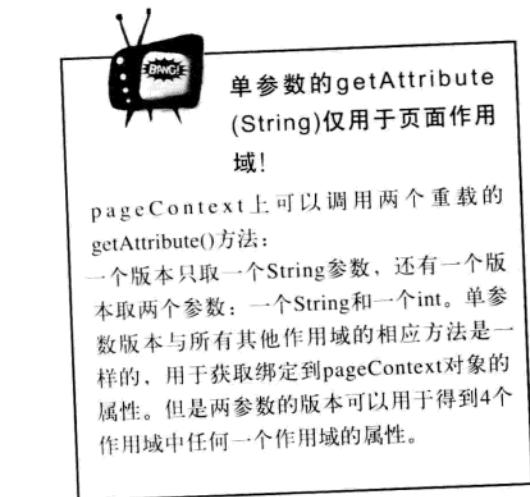
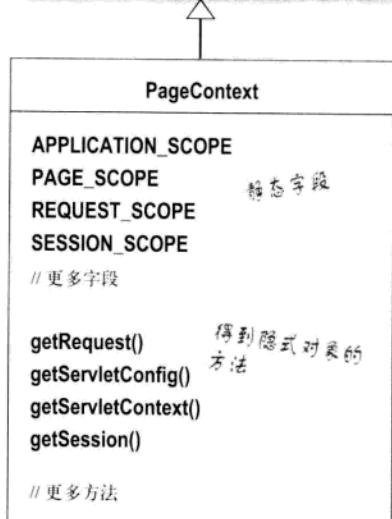
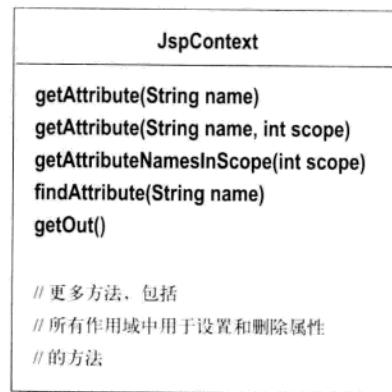
## 简单标记和传统标记的重要区别

	简单标记	传统标记
Tag接口	SimpleTag (扩展 JspTag)	Tag (扩展 JspTag) IterationTag (扩展 Tag) BodyTag (扩展 IterationTag)
支持实现类	SimpleTagSupport (实现 SimpleTag)	TagSupport (实现 IterationTag) BodyTagSupport (扩展 TagSupport, 实现 BodyTag)
可能实现的关键生命周期方法	doTag()	doStartTag() doEndTag() doAfterBody() (对于 BodyTag——还有 doInitBody() 和 setBodyContent())
如何写至响应输出	getJspContext().getOut().println (不需要 try/catch, 因为 SimpleTag 方法声明了 IOException)	pageContext.getOut().println (包装在一个 try/catch 中, 因为传统标记方法没有声明 IOException!)
如何从一个支持实现访问隐式变量和作用域属性	利用 getJspContext() 方法返回一个 JspContext (这通常是一个 PageContext)	利用 pageContext 隐式变量——而不是像 SimpleTag 那样使用一个方法!
如何让体得到处理	getJspBody().invoke(null)	从 doStartTag() 返回 EVAL_BODY_INCLUDE, 或者如果类实现了 BodyTag, 则返回 EVAL_BODY_BUFFERED。
如果让当前页面停止计算	抛出一个 SkipPageException	从 doEndTag() 返回 SKIP_PAGE

# 使用标记处理器的PageContext API

这一页只是无脚本JSP那一章的一个复习，这里之所以会再说一次，原因是它对于标记处理器确实至关重要。记住，标记处理器类不是一个servlet或JSP，所以并不是自动地就能访问一组隐式对象。但是它确实有PageContext的一个引用，利用这个引用，就能得到你需要的所有东西。

记住，尽管简单标记得到的是JspContext的引用，而传统标记得到的是PageContext的引用，但是简单标记的JspContext通常就是一个PageContext实例。所以，如果你的简单标记处理器需要访问PageContext特定的方法或字段，必须把它从JspContext强制转换为PageContext（因为它本来就是PageContext）。





看看你对标记文件记得怎么样  
答案

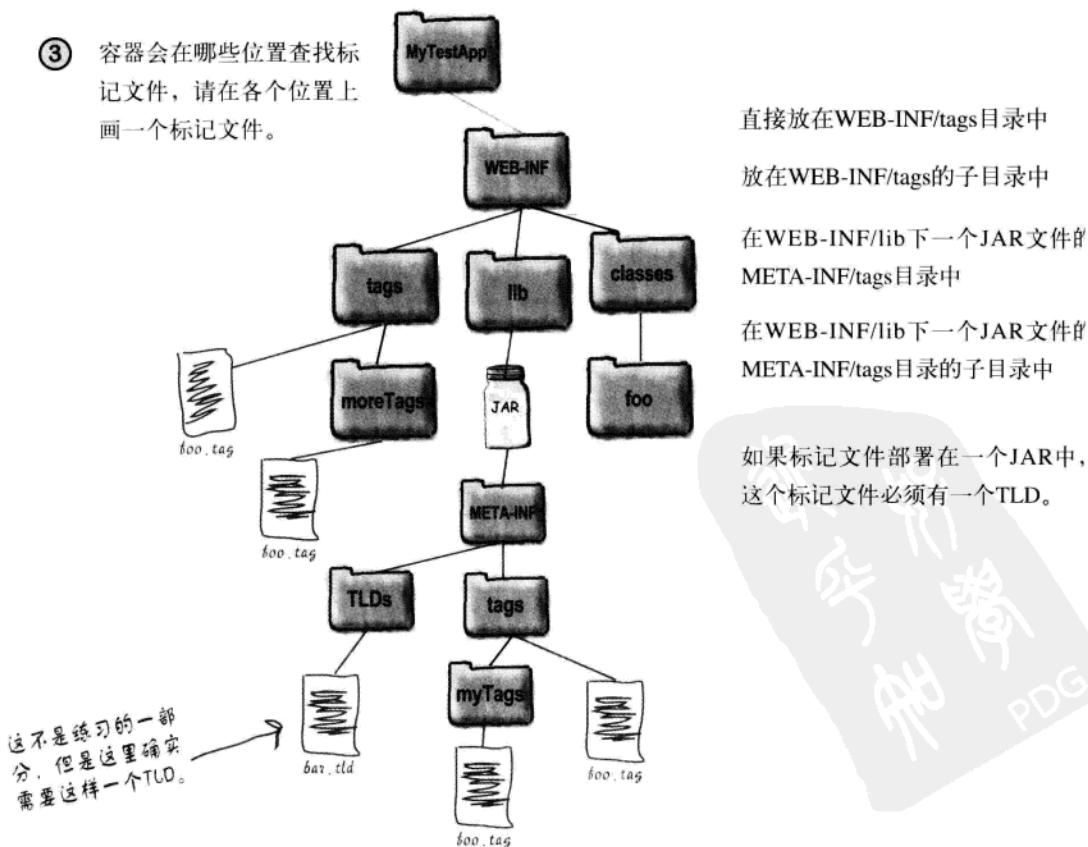
- ① 在标记文件中填空，声明该标记有一个必要的属性，名为“title”，它能使用EL表达式作为属性值。

```
<%@ attribute name="title" required="true" rtextrvalue="true" %>
```

- ② 在标记文件中填空，声明标记不能有体。

```
<%@ tag body-content="empty" %>
```

- ③ 容器会在哪些位置查找标记文件，请在各个位置上画一个标记文件。





## 第10章 模拟测验

1 传统标记处理器如何指示容器忽略调用标记的JSP的余下部分?

(选出所有正确的答案)

- A. `doEndTag()`方法应该返回`Tag.SKIP_BODY`。
- B. `doEndTag()`方法应该返回`Tag.SKIP_PAGE`。
- C. `doStartTag()`方法应该返回`Tag.SKIP_BODY`。
- D. `doStartTag()`方法应该返回`Tag.SKIP_PAGE`。

2 哪些指令和/或标准动作仅用于标记文件? (选出所有正确的答案)

- A. `tag`
- B. `page`
- C. `jsp:body`
- D. `jsp:doBody`
- E. `jsp:invoke`
- F. `taglib`

- 3 一个医疗网站会对未注册的用户有选择地隐藏一些内容。在被隐藏内容所在位置要显示一个消息，建议用户注册。给定以下简单标记处理器代码段：

```
11. public int doTag() throws JspException, IOException {
12.     String level =
13.         (String) getJspContext().findAttribute("accountLevel");
14.     if((level == null || "trial".equals(level))) {
15.         String price = "?"; // TODO get context param
16.         String message = "Content for paying members
17.             only.<br/>" +
18.             "<a href=\"register.jsp\">Sign up now for only
19.                 "+price+"!</a>";
20.         getJspContext().getOut().write(message);
21.     }
22. }
```

在第15行，注册费用应从一个名为registrationFee的上下文参数获取，不过JspContext没有提供获取上下文参数的方法。如何解决这个问题？

- A. 用`pageContext.getServletContext()`  
`.getInitParameter("registrationFee");`获取这个值。
- B. 将`JspContext`强制转换为类型`PageContext`，从而可以使用`PageContext`的方法来获取这个上下文参数。
- C. 用`getJspContext().findAttribute("registrationFee")`;获取这个值。
- D. 抛出一个异常使用户知道未能找到这个价格（注册费用）。
- E. 用简单标记是不可能办到的。必须使用一个传统标记。

哪种简单标记机制会告诉JSP页面停止处理?

- A. 从**doTag**方法返回**SKIP\_PAGE**。
- B. 从**doEndTag**返回**SKIP\_PAGE**。
- C. 从**doTag**抛出一个**SkipPageException**。
- D. 从**doEndTag**抛出一个**SkipPageException**。

5 关于传统标记模型，哪些说法是正确的? (选出所有正确的答案)

- A. **Tag**接口只用于创建空标记。
- B. **SKIP\_PAGE**常量是**doEndTag**方法的一个合法返回值。
- C. **EVAL\_BODY\_BUFFERED**常量是**doAfterBody**方法的一个合法返回值。
- D. **Tag**接口只提供两个值作为**doStartTag**方法的返回值:  
**SKIP\_BODY**和**EVAL\_BODY**。
- E. 有三个标记接口: **Tag**, **IterationTag**和**BodyTag**, 但是只有两个内置的基类: **TagSupport**和**BodyTagSupport**。

6 关于**TagSupport**类中的**findAncestorWithClass**方法，哪些说法是正确的？（选出所有正确的答案）

- A. 它需要一个参数: **Class**。
- B. 这是**TagSupport**类中的一个静态方法。
- C. 这是**TagSupport**类中的一个非静态方法。
- D. 并非由任何JSP标记接口定义。
- E. 它需要两个参数: **Tag**和**Class**。
- F. 它需要一个**String**参数, 它表示要查找的标记的名字。
- G. 它需要两个参数: **Tag**和**String**, 第二个参数表示要查找的标记的名字。

7 如果想使用简单标记处理器的动态属性，必须保证：（选出所有正确的答案）

- A. 简单标记不能声明任何静态标记属性。
- B. 简单标记在TLD中必须使用<**dynamic-attributes**>元素。
- C. 简单标记处理器必须实现**DynamicAttributes**接口。
- D. 简单标记应当扩展**DynamicSimpleTagSupport**类, 它为动态属性提供了默认支持。
- E. 简单标记不能与**jsp:attribute**标准动作一起使用, 因为这个动作仅用于静态属性。

关于标记文件，哪些说法是正确的？（选出所有正确的答案）

- A. 标记文件可以置于**WEB-INF**的任何子目录。
- B. 标记文件的扩展名必须是 **.tag**或**.tagx**。
- C. 必须用TLD文件把符号标记名映射到具体的标记文件。
- D. 标记文件不能放在**WEB-INF/lib**目录的JAR文件中。

给定：

```
10. public class BufTag extends BodyTagSupport {  
11.     public int doStartTag() throws JspException {  
12.         // 在这里插入代码  
13.     }  
14. }
```

假设标记已经适当地配置为允许有体内容。

如果插入到第12行，哪行代码会导致JSP代码

**<mytags:mytag>BodyContent</mytags:mytag>** 输出**BodyContent?**

- A. **return SKIP\_BODY;**
- B. **return EVAL\_BODY\_INCLUDE;**
- C. **return EVAL\_BODY\_BUFFERED;**
- D. **return BODY\_CONTENT;**

---

## 10 关于doAfterBody(), 哪些说法是正确的? (选出所有正确的答案)

- A. **doAfterBody()**只在扩展了**TagSupport**的标记上调用。
- B. **doAfterBody()**只在扩展了**IterationTagSupport**的标记上调用。
- C. 假设没有出现异常, 对于实现了**IterationTag**的任何标记, **doAfterBody()**总会在**doStartTag()**之后调用。
- D. 假设没有出现异常, 对于实现了**IterationTag**而且从**doStartTag()**返回**SKIP\_BODY**的任何标记, 总会在**doStartTag()**之后调用**doAfterBody()**。
- E. 假设没有出现异常, 对于实现了**IterationTag**而且从**doStartTag()**返回**EVAL\_BODY\_INCLUDE**的任何标记, 总会在**doStartTag()**之后调用**doAfterBody()**。

---

## 11 给定一个JSP页面:

```

1. <%@ taglib prefix="my" uri="/WEB-INF/myTags.tld" %>
2. <my:tag1>
3.   <%-- JSP 内容 --%>
4. </my:tag1>
```

my:tag1的标记处理器是**Tag1Handler**, 扩展了**TagSupport**。**Tag1Handler**的实例调用**getParent**方法时会发生什么? (选出所有正确的答案)

- A. 抛出一个 **JspException**。
- B. 返回**null**值。
- C. 抛出一个**NullPointerException**。
- D. 抛出一个**IllegalStateException**。

## 12 关于简单标记的生命周期，哪些说法是正确的？（选出所有正确的答案）

- A. `release`方法在`doTag`方法之后调用。
- B. `setJspBody`方法总在`doTag`方法之前调用。
- C. `setParent`和`setJspContext`方法会在设置标记属性之前调用。
- D. 在标记处理器的`doTag`方法得到调用之前，会由容器调用标记体的`JspFragment`。这个值是一个`BodyContent`对象，并使用`setJspBody`方法传递给标记处理器。

## 13

给定：

```

10. public class ExampleTag extends TagSupport {
11.     private String param;
12.     public void setParam(String p) { param = p; }
13.     public int doStartTag() throws JspException {
14.         // insert code here
15.         // more code here
16.     }
17. }
```

如果插在第14行，哪行代码能保证将请求作用域属性`param`的值赋给局部变量`p`？  
 （选出所有正确的答案）

- A. `String p = findAttribute("param");`
- B. `String p = request.getAttribute("param");`
- C. `String p = pageContext.findAttribute("param");`
- D. `String p = getPageContext().findAttribute("param");`
- E. `String p = (String) pageContext.getRequest().getAttribute("param");`

14

哪些是**PageContext**对象的合法方法调用？（选出所有正确的答案）

- A. `getAttributeNames()`
- B. `getAttribute("key")`
- C. `findAttribute("key")`
- D. `getSessionAttribute()`
- E. `getAttributesScope("key")`
- F. `findAttribute("key", PageContext.SESSION_SCOPE)`
- G. `getAttribute("key", PageContext.SESSION_SCOPE)`

15

要访问肯定在应用作用域中的属性，调用哪一个**JspContext**方法最高效？

- A. `getPageContext()`
- B. `getAttribute(String)`
- C. `findAttribute(String)`
- D. `getAttribute(String, int)`
- E. `getAttributesScope("key")`
- F. `getAttributeNamesInScope(int)`

16

实现一个定制标记时，要查找一个作用域未知的属性的值，最佳策略是什么？

- A. 用一个`pageContext.getAttribute(String)`调用检查所有作用域。
- B. 用一个`pageContext.findAttribute(String)`调用检查所有作用域。
- C. 用多个`pageContext.getAttribute(String, int)`调用检查各个作用域。
- D. 调用`pageContext.getRequest().getAttribute(String)`, 然后调用`pageContext.getSession().getAttribute(String)`, 依此类推。
- E. 以上都不行。

17

给定一个标记`simpleTag`，其处理器使用简单标记模型实现，另外有一个`complexTypeTag`标记，其处理器使用传统标记模型实现。这两个标记在TLD中都声明为非空，而且不依赖于标记。

哪个JSP代码段正确地使用了这些标记？（选出所有正确的答案）

- A. 

```
<my:simpleTag>
    <my:complexType />
</my:simpleTag>
```
- B. 

```
<my:simpleTag>
    <%= displayText %>
</my:simpleTag>
```
- C. 

```
<my:simpleTag>
    <%@ include file="/WEB-INF/web/common/headerMenu.html" %>
</my:simpleTag>
```
- D. 

```
<my:simpleTag>
    <my:complexType>
        <% i++; %>
    </my:complexType>
</my:simpleTag>
```

---

18 关于标记文件模型，哪些说法是正确的？（选出所有正确的答案）

- A. 每个标记文件在TLD文件中必须有相应的项。
- B. JSP页面中能用的所有指令在标记文件中也能用。
- C. 标记文件中能用的所有指令在JSP页面中也能用。
- D. <jsp:doBody>标准动作只能在标记文件中使用。
- E. 标记文件的扩展名只能是.tag 和 tagx。
- F. 对于标记文件中声明和指定的各个属性，容器会创建一个同名的页面作用域属性。

---

19 哪些代码放在标记文件中是合法的？（选出所有正确的答案）

- A. <jsp:doBody />
- B. <jsp:invoke fragment="frag" />
- C. <%@ page import="java.util.Date" %>
- D. <%@ variable name-given="date" variable-class="java.util.Date" %>
- E. <%@ attribute name="name" value="blank" type="java.lang.String" %>

---

20 从一个标记处理器类中调用哪个方法会返回外层标记？（选出所有正确的答案）

- A. getParent()
- B. getAncestor()
- C. findAncestor()
- D. getEnclosingTag()

---

**21**

给定一个Web应用结构：

```
/WEB-INF/tags/mytags/tag1.tag  
/WEB-INF/tags/tag2.tag  
/WEB-INF/tag3.tag  
/tag4.tag
```

哪些标记可以由适当的**taglib**指令使用？（选出所有正确的答案）

- A. **tag1.tag**
- B. **tag2.tag**
- C. **tag3.tag**
- D. **tag4.tag**

---

**22**

一个Web应用中包含很多表单供用户填写和提交。这些页面中没有任何提示指出某个域必须填写。主管认为应当在必填域的文本标签前面放上一个红色星号，但是项目经理提出异议，认为应当把必填域的背景色置为淡蓝色，而另一个部门则要求这个项目的应用要与他们自己的应用一致，即必要域的标签应加粗。

请考虑以上关于如何在页面中标识必填域的不同观点，选择定制标记最可维护的用法。

- A. **<cust:requiredIcon/>First Name: <input type="text" name="firstName"/>**
- B. **<cust:textField label="First Name" required="true"/>**
- C. **<cust:requiredField color="red" symbol="\*" label="First Name"/>**
- D. **<cust:required>  
    First Name: <input type="text" name="firstName"/>  
</cust:required>**



## 第10章 模拟测验答案

**1** 传统标记处理器如何指示容器忽略调用标记的JSP的余下部分?  
(选出所有正确的答案)

(JSP v2.0 pg 2~56)

- A. `doEndTag()`方法应该返回`Tag.SKIP_BODY`。
- B. `doEndTag()`方法应该返回`Tag.SKIP_PAGE`。
- C. `doStartTag()`方法应该返回`Tag.SKIP_BODY`。
- D. `doStartTag()`方法应该返回`Tag.SKIP_PAGE`。

A不对，因为这不是`doEndTag()`的方法返回值。

C不对，因为这只会导致跳过标记体。

D不对，因为这不是`doStartTag()`的方法返回值。

**2** 哪些指令和/或标准动作仅用于标记文件?  
(选出所有正确的答案)

(JSP v2.0 8.5 (1~179页))

JSP v2.0 5.11节

JSP v2.0 5.12节

JSP v2.0 5.13节

- A. `tag`      A是对的(1~179页)。
- B. `page`      B不对，因为标记文件中不允许有`page`指令(1~179页)。
- C. `jsp:body`      C不对，因为`jsp:body`动作既可以在标记文件中，也可以出现在JSP中。
- D. `jsp:doBody`      D是对的 (1~121页)。
- E. `jsp:invoke`      E是对的 (1~119页)。
- F. `taglib`      F不对，因为`taglib`指令既可以在标记文件中，又可以出现在JSP中。

一个医疗网站会对未注册的用户有选择地隐藏一些内容。在被隐藏内容所在位置要显示一个消息，建议用户注册。给定以下简单标记处理器代码段：

```

11. public int doTag() throws JspException, IOException {
12.     String level =
13.         (String) getJspContext().findAttribute("accountLevel");
14.     if((level == null || "trial".equals(level))) {
15.         String price = "?"; // TODO get context param
16.         String message = "Content for paying members
17.             only.<br/>"+
18.             "<a href=\"register.jsp\">Sign up now for only
19.                 "+price+"!</a>";
20.         getJspContext().getOut().write(message);
21.     } else {
22.         getJspBody().invoke(null);
23.     }
24. }
```

在第15行，注册费用应从一个名为registrationFee的上下文参数获取，不过JspContext没有提供获取上下文参数的方法。如何解决这个问题？

A: pageContext变量只对传统标记可用。

- A. 用`pageContext.getServletContext()`  
`.getInitParameter("registrationFee")`; 获取这个值。
- B. 将`JspContext`强制转换为类型`PageContext`，从而可以使用`PageContext`的方法来获取这个上下文参数。  
B正确。我们没有提过这个技巧，参加考试时也不要要求你了解这一点，不过在实际开发中这个技巧可能会很方便。
- C. 用`getJspContext().findAttribute("registrationFee")`; 获取这个值。  
C: 要记住，我们并不是要获取一个属性，这里要得到的是一个上下文参数。
- D. 抛出一个异常使用户知道未能找到这个价格（注册费用）。  
D: 不要这么早就放弃！你要有决心，相信一定能提供一个很好的解决方案！
- E. 用简单标记是不可能办到的。必须使用一个传统标记。  
E: 并不是不可能，只是有些技巧。

4

哪种简单标记机制会告诉JSP页面停止处理?

(JSP v2.0 13.6.1节)

- A. 从doTag方法返回SKIP\_PAGE。

A不对，因为doTag方法不返回值。

- B. 从doEndTag返回SKIP\_PAGE。

B不对，因为简单标记没有doEndTag事件方法。

- C. 从doTag抛出一个SkipPageException。

- D. 从doEndTag抛出一个SkipPageException。

D不对，因为简单标记没有doEndTag事件方法。

5

关于传统标记模型，哪些说法是正确的？（选出所有正确的答案）

(JSP v2.0 13.1和13.2节)

- A. Tag接口只用于创建空标记。

A不对，因为Tag接口可以支持有体的标记，但是不能循环执行体或访问体内容。

- B. SKIP\_PAGE常量是doEndTag方法的一个合法返回值。

- C. EVAL\_BODY\_BUFFERED常量是doAfterBody方法的一个合法返回值。

C不对，因为doAfterBody只能返回SKIP\_BODY或EVAL\_BODY\_AGAIN。

- D. Tag接口只提供两个值作为doStartTag方法的返回值：SKIP\_BODY和EVAL\_BODY。

D不对，因为doStartTag可能返回SKIP\_BODY和EVAL\_BODY\_INCLUDE。

- E. 有三个标记接口：Tag、IterationTag和BodyTag，但是只有两个内置的基类：TagSupport和BodyTagSupport。

6

关于TagSupport类中的findAncestorWithClass方法，哪些说法是正确的？（选出所有正确的答案）。

- A. 它需要一个参数: **Class**。  
C不对，因为这个方法是静态的。
- B. 这是**TagSupport**类中的一个静态方法。  
C和F不对，因为这个方法有两个参数。
- C. 这是**TagSupport**类中的一个非静态方法。
- D. 并非由任何JSP标记接口定义。
- E. 它需要两个参数: **Tag**和**Class**。  
A和F不对，因为第二个参数是一个**Class**。
- F. 它需要一个**String**参数，它表示要查找的标记的名字。
- G. 它需要两个参数: **Tag**和**String**，第二个参数表示要查找的标记的名字。  
G不对，因为第二个参数是一个**Class**。

7

如果想使用简单标记处理器的动态属性，必须保证：（选出所有正确的答案）  
(JSP v2.0 13.3节 2~74, 75页)

- A. 简单标记不能声明任何静态标记属性。  
A不对，因为简单标记中可以有静态属性和动态属性。
- B. 简单标记在TLD中必须使用<**dynamic-attributes**>元素。
- C. 简单标记处理器必须实现**DynamicAttributes**接口。  
D不对，因为内置API中没有这种辅助类。
- D. 简单标记应当扩展**DynamicSimpleTagSupport**类，它为动态属性提供了默认支持。
- E. 简单标记不能与**jsp:attribute**标准动作一起使用，因为这个动作仅用于静态属性。  
E不对，因为动态标记可以使用**jsp:attribute**动作。

8

关于标记文件，哪些说法是正确的？（选出所有正确的答案）

(JSP v2.0 8.4节)

- A. 标记文件可以置于**WEB-INF**的任何子目录。

A不对，因为标记文件必须放在  
**WEB-INF/tags**目录中。

- B. 标记文件的扩展名必须是 **.tag**或**.tagx**。

B是对的。（1~176页，8.4.1节）

- C. 必须用TLD文件把符号标记名映射到具体的标记文件。

C不对，因为容器会在几个已知的位置上  
查找标记文件。容器的这个特性是可选的。

- D. 标记文件不能放在**WEB-INF/lib**目录的JAR文件中。

9

给定：

(JSP v2.0 2~68页)

```
10. public class BufTag extends BodyTagSupport {
11.     public int doStartTag() throws JspException {
12.         // 在这里插入代码
13.     }
14. }
```

假设标记已经适当地配置为允许有体内容。

如果插入到第12行，哪行代码会导致JSP代码

<**mytags:mytag**>BodyContent</**mytags:mytag**> 输出BodyContent?

- A. **return SKIP\_BODY;**

A不对，因为这会导致跳过标记体

- B. **return EVAL\_BODY\_INCLUDE;**

- C. **return EVAL\_BODY\_BUFFERED;** C不对，因为这会导致体置于  
缓冲区，而标记不做处理。

- D. **return BODY\_CONTENT;**

D不对，因为这不是  
一个合法的返回值。

10 关于doAfterBody(), 哪些说法是正确的? (选出所有正确的答案)

(JSP v2.0 (~152页))

- A. doAfterBody()只在扩展了TagSupport的标记上调用。 A不对, 因为doAfterBody()可以在实现了IterationTag接口的任何标记上调用。
- B. doAfterBody()只在扩展了IterationTagSupport的标记上调用。 B不对, 因为没有这样一个类。
- C. 假设没有出现异常, 对于实现了IterationTag的任何标记, doAfterBody()总会在doStartTag()之后调用。 C和D是不对的, 因为只有doStartTag()返回EVAL\_BODY\_INCLUDE时才会调用doAfterBody()。
- D. 假设没有出现异常, 对于实现了IterationTag而且从doStartTag()返回SKIP\_BODY的任何标记, 总会在doStartTag()之后调用doAfterBody()。
- E. 假设没有出现异常, 对于实现了IterationTag而且从doStartTag()返回EVAL\_BODY\_INCLUDE的任何标记, 总会在doStartTag()之后调用doAfterBody()。

11 给定一个JSP页面:

(JSP v2.0 TagSupport API  
2~64页)

```
1. <%@ taglib prefix="my" uri="/WEB-INF/myTags.tld" %>
2. <my:tag1>
3.   <%-- JSP 内容 --%>
4. </my:tag1>
```

my:tag1的标记处理器是Tag1Handler, 扩展了TagSupport。

Tag1Handler的实例调用getParent方法时会发生什么? (选出所有正确的答案)

- A. 抛出一个 JspException.
- B. 返回null值。 B是对的。 getParent方法不抛出任何异常。
- C. 抛出一个NullPointerException.
- D. 抛出一个IllegalStateException.

12

关于简单标记的生命周期，哪些说法是正确的？（选出所有正确的答案）

(JSP v2.0 (3.6节2~80/83页)

- A. **release**方法在**doTag**方法之后调用。

A不对，因为简单标记没有**release**方法。

- B. **setJspBody**方法总在**doTag**方法之前调用。

B不对，如果简单标记是一个空标记，就不会调用**setJspBody**。

- C. **setParent**和**setJspContext**方法会在设置标记属性之前调用。

- D. 在标记处理器的**doTag**方法得到调用之前，会由容器调用标记体的**JspFragment**。这个值是一个**BodyContent**对象，并使用**setJspBody**方法传递给标记处理器。

D不对，因为片段由**doTag**实现调用，而不是在**doTag**之前调用。

13

给定：

(JSP v2.0 2~27页)

```

10. public class ExampleTag extends TagSupport {
11.     private String param;
12.     public void setParam(String p) { param = p; }
13.     public int doStartTag() throws JspException {
14.         // insert code here
15.         // more code here
16.     }
17. }
```

如果插在第14行，哪行代码能保证将请求作用域属性**param**的值赋给局部变量**p**？

(选出所有正确的答案)

- A. **String p = findAttribute("param");**

A不对，因为根本没有这个方法。

- B. **String p = request.getAttribute("param");**

B不对，因为没有**request**实例变量。

- C. **String p = pageContext.findAttribute("param");**

C不对，会在检查请求作用域之前先找到页面作用域中的属性。

- D. **String p = getPageContext().findAttribute("param");**

D不对，因为没有**getPageContext()**方法。

- E. **String p = (String) pageContext.getRequest().getAttribute("param");**

14

哪些是PageContext对象的合法方法调用? (选出所有正确的答案)

(JSP v2.0 2~23页)

- A. `getAttributeNames()`
- B. `getAttribute("key")` A和D不对, 因为没有这些方法。
- C. `findAttribute("key")`
- D. `getSessionAttribute()`
- E. `getAttributesScope("key")`
- F. `findAttribute("key", PageContext.SESSION_SCOPE)` F不对, 因为`findAttribute()`没有`scope`参数。
- G. `getAttribute("key", PageContext.SESSION_SCOPE)`

5

要访问肯定在应用作用域中的属性, 调用哪一个JspContext方法最高效?

(JSP v2.0 2~23页)

- A. `getPageContext()` A不对, 因为根本没有这个方法。
- B. `getAttribute(String)` B不对, 因为这个方法只在页面作用域查找。
- C. `findAttribute(String)` C不对, 因为这个方法没有D效率高, 它会先检查另外3个作用域。
- D. `getAttribute(String, int)`
- E. `getAttributesScope("key")` E不对, 因为根本不存在这个方法。
- F. `getAttributeNamesInScope(int)` F不对, 因为这只是第一步, 即使接下来完成整个过程, 这个做法也比D的效率低。

16

实现一个定制标记时，要查找一个作用域未知的属性的值，最佳策略是什么？

(JSP v2.0 2-23页)

- A. 用一个`pageContext.getAttribute(String)`调用检查所有作用域。 A不对，因为这个方法只检查页面作用域。
- B. 用一个`pageContext.findAttribute(String)`调用检查所有作用域。
- C. 用多个`pageContext.getAttribute(String, int)`调用检查各个作用域。 C和D不对，因为(
- D. 调用`pageContext.getRequest().getAttribute(String)`, 然后调用`pageContext.getSession().getAttribute(String)`, 依此类推。 调用`findAttribute()`说，它们的效率比较低。
- E. 以上都不行。

17

给定一个标记`simpleTag`，其处理器使用简单标记模型实现，另外有一个`complexTag`标记，其处理器使用传统标记模型实现。这两个标记在TLD中都声明为非空，而且不依赖于标记。

(JSP v2.0 7.1.6 (~156页))

哪个JSP代码段正确地使用了这些标记？（选出所有正确的答案）。

- A. `<my:simpleTag>`  
 `<my:complexTag />`  
`</my:simpleTag>`
- B. `<my:simpleTag>`  
 `<%= displayText %>`  
`</my:simpleTag>`
- C. `<my:simpleTag>`  
 `<%@ include file="/WEB-INF/web/common/headerMenu.html" %>`  
`</my:simpleTag>`
- D. `<my:simpleTag>`  
 `<my:complexTag>`  
 `<% i++; %>`  
 `</my:complexTag>`  
`</my:simpleTag>`

A是对的：`simpleTag`的体中可以包含`complexTag`，只要该标记中不包含脚本代码。

B不对，因为`simpleTag`的体中不能包含JSP表达式标记。

C是对的，因为`include`指令要在`simpleTag`的体转换为`JspFragment`之前得到处理；不过，所包含的内容也不能有脚本（所以这个例子包含一个HTML片段）。

D不对，不是因为`complexTag`使用得不正确（不同于A），而是因为`complexTag`体中有脚本。

18

关于标记文件模型，哪些说法是正确的？（选出所有正确的答案）

(JSP v2.0 1~173页)

- A. 每个标记文件在TLD文件中必须有相应的项。
- B. JSP页面中能用的所有指令在标记文件中也能用。
- C. 标记文件中能用的所有指令在JSP页面中也能用。
- D. <jsp:doBody>标准动作只能在标记文件中使用。
- E. 标记文件的扩展名只能是.tag 和 tagx。
- F. 对于标记文件中声明和指定的各个属性，容器会创建一个同名的页面作用域属性。

A不对，因为使用标记文件时，只需要放在适当的位置就行。

B不对，因为标记文件中不能用page指令。

C不对，因为JSP页面中不能用tag、attribute 和 variable 指令。

19

哪些代码放在标记文件中是合法的？（选出所有正确的答案）

(JSP v2.0 1~174页)

- A. <jsp:doBody />
- B. <jsp:invoke fragment="frag" />
- C. <%@ page import="java.util.Date" %> C不对，因为标记文件中不能用page指令。
- D. <%@ variable name-given="date" variable-class="java.util.Date" %>
- E. <%@ attribute name="name" value="blank" type="java.lang.String" %> E不对，因为attribute指令没有value属性。

20

从一个标记处理器类中调用哪个方法会返回外层标记？

(JSP v2.0 2~53页)

（选出所有正确的答案）

- A. getParent() A是对的，这只是可能的方法之一。
- B. getAncestor()
- C. findAncestor()
- D. getEnclosingTag()

21

给定一个Web应用结构:

(JSP v2.0 (~176页))

```
/WEB-INF/tags/mytags/tag1.tag
/WEB-INF/tags/tag2.tag
/WEB-INF/tag3.tag
/tag4.tag
```

哪些标记可以由适当的taglib指令使用? (选出所有正确的答案)

- A. tag1.tag  
 B. tag2.tag  
 C. tag3.tag      C和D不对, 因为标记文件必须放在/WEB-INF/tags目录下, 或者放在/WEB-INF/tags的一个子目录中。  
 D. tag4.tag

22

一个Web应用中包含很多表单供用户填写和提交。这些页面中没有任何提示指出某个域必须填写。主管认为应当在必填域的文本标签前面放上一个红色星号, 但是项目经理提出异议, 认为应当把必填域的背景色设置为淡蓝色, 而另一个部门则要求这个项目的应用要与他们自己的应用一致, 即必要域的标签应加粗。

请考虑以上关于如何在页面中标识必填域的不同观点, 选择定制标记最可维护的用法。

- A. <cust:requiredIcon/>First Name: <input type="text" name="firstName"/>  
 B. <cust:textField label="First Name" required="true"/>  
 C. <cust:requiredField color="red" symbol="\*" label="First Name"/>  
 D. <cust:required> First Name: <input type="text" name="firstName"/> </cust:required>

如果你知道必填域前面有一个符号加以标识, 且可能改变的只是所用的标识符, 那么A也是可以的。不过, 即便如此, C也过于简单, 就好像使用一个img标记放上一个图标目录中的.gif图标一样。

B是最灵活的解决方案。这为你的定制标记提供了充分的灵活性, 可以完全控制标签和文本域的构造, 还可以控制它们的显示。

C: 在标记中指定color和symbol, 这种解决方案不太让人满意, 因为不论修改其中哪一個值, 都要求你更新每一个JSP中所有标记的相应值。

D: 这样做也是可以的, 不过实现标记的类必须解析体, 并进行管理, 这会带来一个维护噩梦。

# 部署Web应用



Web应用终于到了最后的重要时刻。你的页面完美无瑕，代码也已经经过测试和调优，最后期限已经过去了两个星期。但是所有这些东西放到哪里呢？这么多目录，这么多的规则。你怎么对目录命名？客户又怎么认为？客户实际请求的是什么？容器怎么知道到哪里去查找？如果你把整个Web应用都移到另外一个主机上，怎么确保不会不小心漏掉一个目录？如果客户请求的是一个目录而不是特定的文件会怎么样？DD中怎么配置错误页面、欢迎文件和MIME类型？尽管听上去很乱，但实际上没有那么糟……

# OBJECTIVES

## Web应用部署

- 2.1** 构建Web应用的文件和目录结构，其中可能包含(a)静态内容, (b) JSP页面, (c) servlet类, (d) 部署描述文件, (e) 标记库, (f) JAR文件和(g) Java类文件。描述如何保护资源文件避免HTTP访问。
- 2.2** 说明以下部署描述文件元素的作用和语义：error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name和welcome-file。
- 2.3** 为以下各个部署描述文件元素建立正确的结构：error-page, init-param, mime-mapping, servlet, servlet-class, servlet-name和welcome-file。
- 2.4** 解释WAR文件的作用，并说明WAR文件的内容，以及如何构建一个WAR文件。

- 
- 6.3** 编写一个语法正确的JSP文档（采用基于XML的语法）。

### 内容说明：

这个要求在本书其他章中已经介绍过，所以这一章中与这个要求相关的大多数内容要么只是复习，要么会讨论得更详细更深入一些。

要求2.2和2.3主要强调与部署描述文件相关的一些XML标记的细节。虽然这可能是这本书（和考试）中最没意思的部分，但大部分内容都很容易理解，你只要把这些标记记住。

不过有一部分比较难——servlet映射，这也是我们用很多笔墨着重讨论的内容。

之所以把这个要求放在这一章中，有两个原因：1) 这一章大多数内容都与XML有关；2) 我们不想再向JSP那几章里加东西了。在几章里，我们认为你最好把重点更多地放在JSP所有其他部分的语法和行为上，不必同时考虑XML版本的东西（如JSP文档）。不过既然现在你已经是一个专家了……相信你能应付得了。

# 部署的快乐

有意思的内容大多都已经介绍过了，现在再来更详细地讨论部署。

在这一章中，你要考虑3个主要问题：

## ① 你把Web应用中的东西放在哪里？

静态资源放在哪里？JSP页面呢？Servlet类文件呢？JavaBean类文件呢？监听者类文件呢？标记文件呢？标记处理器类呢？TLD呢？JAR文件呢？web.xml DD呢？还有一些东西你不希望容器对外提供，这些东西放在哪里？（换句话说，Web应用的哪些部分要得到保护，不能由客户直接访问？），另外，“欢迎”文件放在哪里？

## ② 容器会在哪里找Web应用中的东西？

客户请求HTML页面时容器会到哪里查找？JSP页面呢？servlet呢？如果请求的并不是一个具体文件（比如，BeerTest.do）会怎么样？容器在哪里查找标记处理器类？容器去哪里找TLD？标记文件？JAR文件？部署描述文件？Servlet依赖的其他类？容器在哪里查找“欢迎”文件？（显然，如果这些问题你都清楚了，那么以上第一条里的问题对你来说就只是小菜一碟了。）

## ③ 客户怎么请求Web应用里的东西呢？

客户要访问HTML页面的话，该在浏览器里键入什么？要访问JSP页面呢？servlet呢？如果请求的东西并不是具体文件会怎么样？哪些地方客户可以直接请求，哪些地方会限制客户直接访问资源？如果客户键入的只是一个目录的路径，而不是特定文件的路径，会怎么样？

## Web应用中的东西要放在哪里

这本书前面有几章介绍过各种文件该放在哪里。例如，在介绍定制标记的那一章中，你已经了解到标记文件必须部署在/WEB-INF/tags或其子目录中，或者部署在/META-INF/libs或其子目录下的一个JAR文件中。如果把标记文件放在别的地方，容器可能会忽略这个标记文件，也可能把它当作可以对外提供的静态内容。

Servlet和JSP规范关于各个东西该放在哪里有许多严格的规则，其中大多数你都必须掌握。因为这些内容都曾以某种方式介绍过，所以这一章的前几页会让你完成一些测试，看看你记住了没有，理解得对不对。不要跳过这些练习！要把后面几页当成是实际的测验题！

### *there are no Dumb Questions*

**问：** 我何必要知道这些东西放在哪里……部署工具不就是做这个用的吗？或者甚至一个ANT构建脚本也能完成部署！？

**答：** 如果你幸运，使用的是一个J2EE部署工具，它会给你提供一系列的向导屏幕让你通过选择和点击完成部署。你的容器使用这些信息来构建XML部署描述文件（web.xml），建立必要的目录结构，并且将你的文件复制到适当的位置。但是就算你很幸运有这样一个工具，难道你不想知道这个工具是怎么做的吗？你可能想了解部署工具到底做了什么：可能想除错；也可能想换个开发商，而那个开发商也许没有提供一个自动化的部署工具。

很多开发人员会使用诸如ANT的构建工具，但是即便如此，还是需要告诉ANT该做什么。

**问：** 但是我只是从网上拿到一个ANT构建脚本，它已经都为我配置好了。

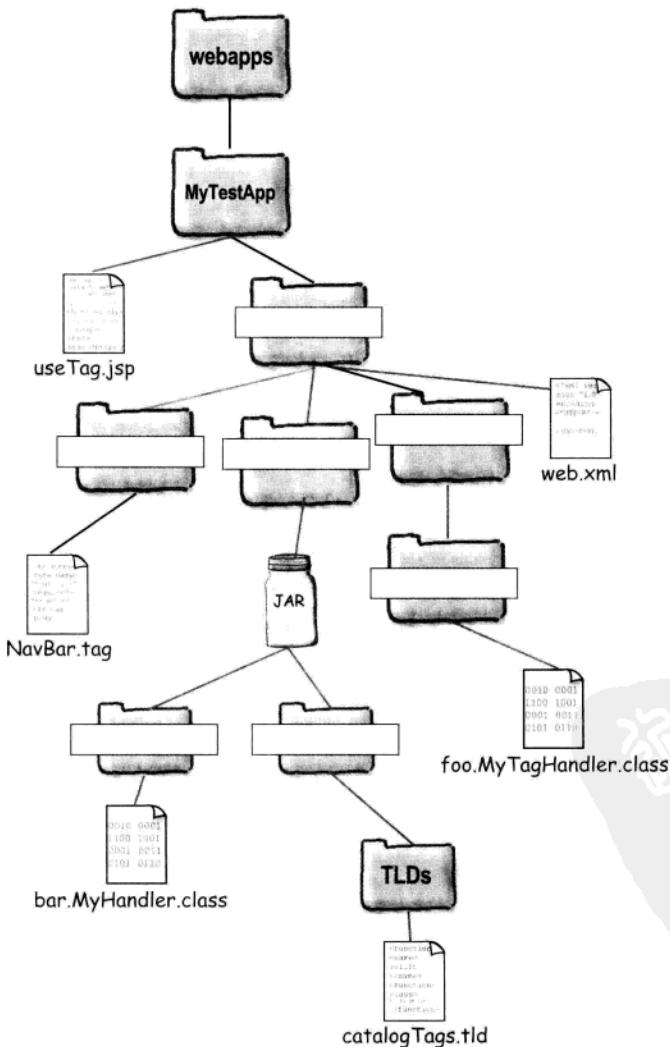
**答：** 再说一遍，这当然很棒，不过还是需要知道到底发生了什么。如果你完全受所用工具的控制，一旦出问题你就该有麻烦了。你要知道如何构建一个Web应用，这就像要知道怎么换轮胎一样，也许你永远也不需要自己换轮胎，但是假设现在是凌晨3点，你走在半路上，如果现在需要换轮胎，而且你知道该怎么做不是很好吗？

另外，如果你要参加考试，那就别无选择。这一章中几乎所有内容都会在考试中考到。



## 填入目录名

给定以下目录中所示的文件，填入正确的目录名。这里的所有的内容都已经在前面的某一章中介绍过，不过就算你没有完全记住，也不用担心。你要在这一章把这些内容牢牢记在脑子里。





## 画出目录和文件结构

请看以下Web应用描述，画出支持这个Web应用的目录结构。

另外，一定要包括文件。可能有不只一种结构；建议你使用最简单的方式组织（也就是说，目录最少）。

应用名: Dating

静态内容和JSP: welcome.html, signup.jsp, search.jsp

Servlet: dating.Enroll.class, dating.Search.class

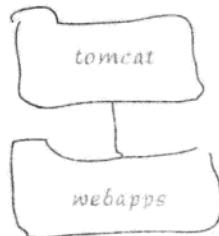
定制标记处理器类: tagClasses.TagOne.class

TLD: DatingTags.tld

JavaBean: dating.Client.class

DD: web.xml

支持JAR文件: DatingJar.jar

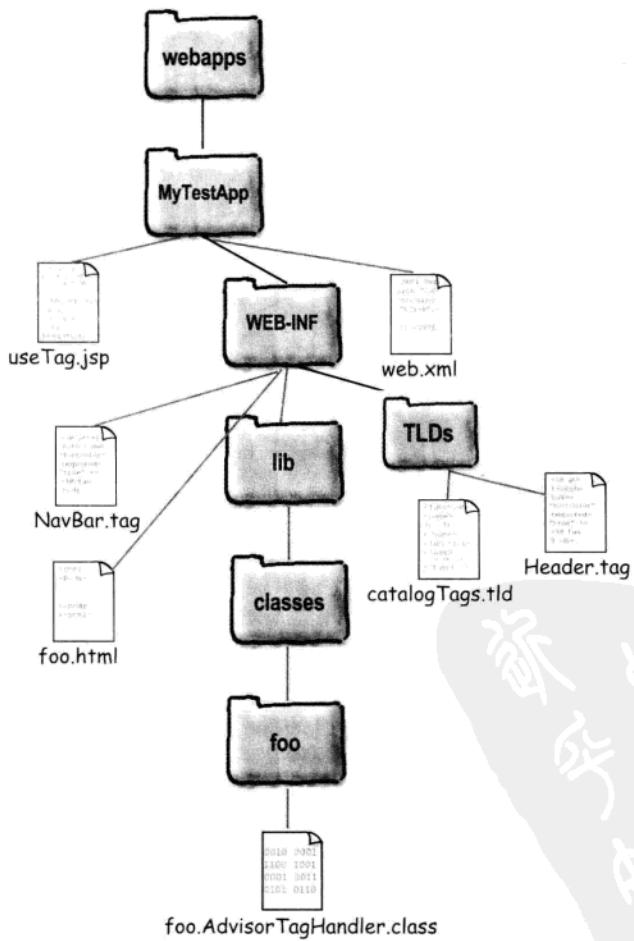


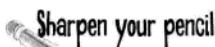
# 作为容器

这个部署有什么问题吗？这里有很多地方没有遵循Servlet或JSP规范，很多东西没有放在该放的位置上。假设所有文件名和扩展名都是正确的。



列出这个图中有错误的地方：

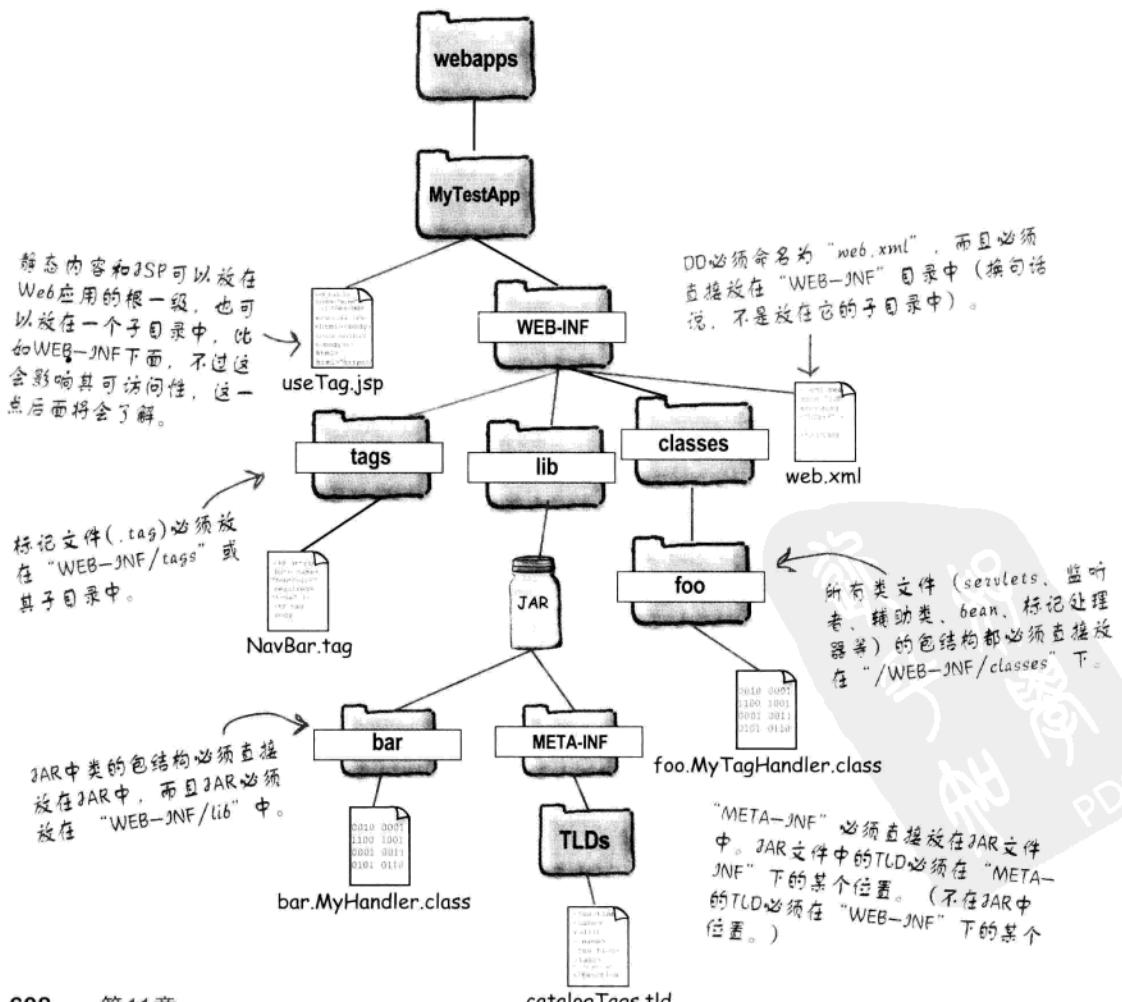




答案

## 填入目录名

要成功地部署一个Web应用，必须遵循以下目录结构。WEB-INF一定要直接放在应用上下文之下（这个例子中，应用上下文就是“MyTestApp”）。 “classes” 目录必须直接放在“WEB-INF”目录中。“classes”目录中必须是类的包结构。“lib”目录要直接置于“WEB-INF”目录之下，JAR文件必须放在“lib”中。“META-INF”目录必须是JAR中的顶级目录，JAR中的TLD文件要放在“META-INF”目录下的某个位置（可以在任何子目录中，目录名不必是“TLDs”）。不在JAR中的TLD必须放在“WEB-INF”下的某个位置。标记文件（扩展名为.tag或.tagx的文件）必须放在“WEB-INF/tags”下的某个地方（除非部署在一个JAR中，如果是这样，这些TLD必须放在“META-INF/tags”下的某个位置）。





答案

## 画出目录和文件结构

这个图中只有两处可能不同：1) 静态内容和JSP可以在“Dating”下的一个子目录中，或者隐藏在“WEB-INF”之下；2) DatingTags.tld可以在WEB-INF的一个子目录中。

应用名： Dating

静态内容和JSP： welcome.html, signup.jsp, search.jsp

Servlet: dating.Enroll class, dating.Search class

定制标记处理器类: tagClasses.TagOne class

TLD: DatingTags.tld

JavaBean类： dating.Client class

DD: web.xml

支持JAR文件： DatingJar.jar

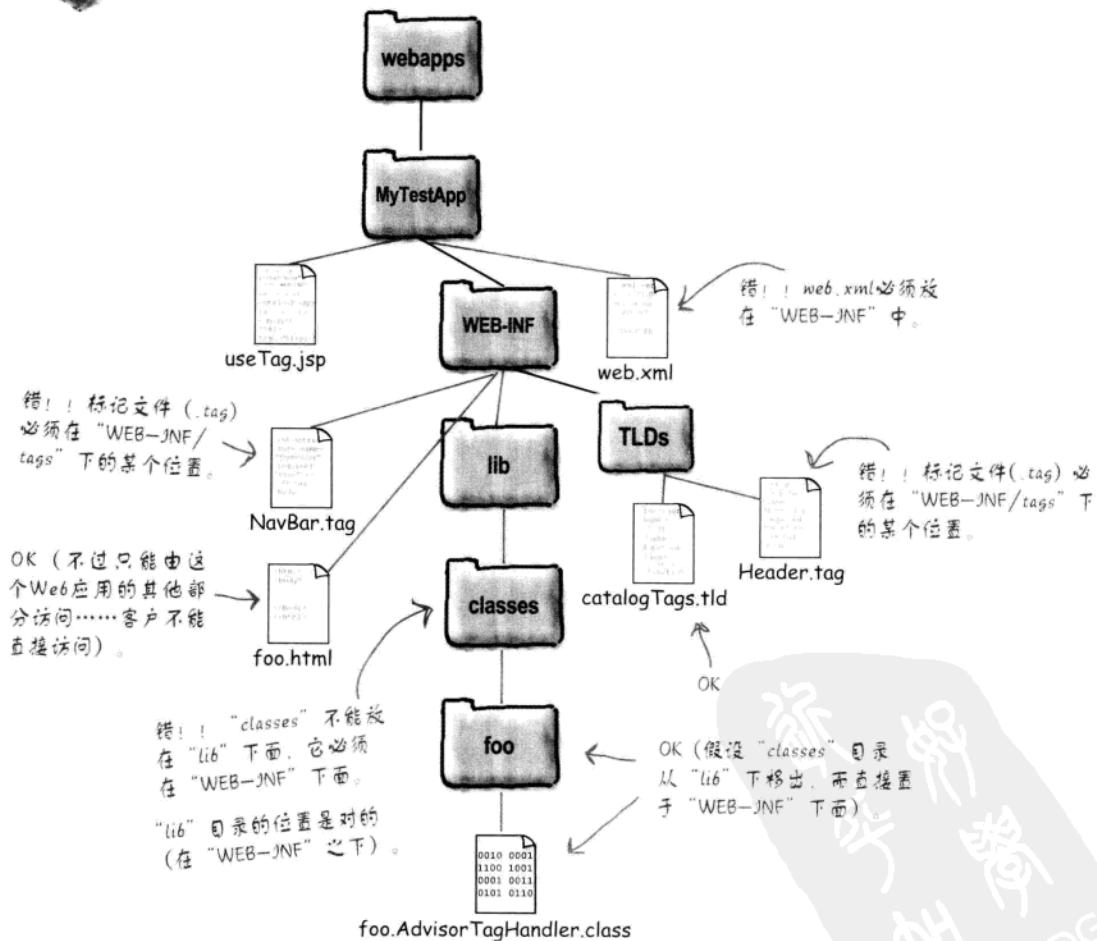


作为容器

答案



这个图中有多少处错误?



噢，如果有办法把整个Web应用部署在一个JAR中就好了，这样我就能把它作为一个文件移动，而不是这么一大堆的文件和目录，而且……



### 她真正想要的是一个WAR文件

Web应用的目录结构要求很严，各个内容只能放在它该放的地方。所以要想移动一个Web应用很让人头疼。

不过还是有办法的，这称为WAR文件，代表Web归档（Web ARchive），你可能认为这有点像JAR文件（Java归档,Java ARchive），确实如此，因为WAR确实就是一个JAR。这个JAR文件的扩展名是.war而不是.jar。

## WAR文件

WAR文件只是Web应用结构的一个快照，采用了一种更可移植的压缩形式（它实际上就是一个JAR文件）。建立WAR文件时，就是把整个Web应用结构（去掉Web应用上下文目录，也就是把WEB-INF之上的一级目录去掉）压缩起来，并指定一个.war扩展名。不过，还有一个问题，如果没有包括特定的Web应用目录（例如，BeerApp），容器怎么知道这个Web应用的名/上下文呢？

这就取决于你的容器了。在Tomcat中，WAR文件的文件名就会成为**Web应用的名字！**假设你把BeerApp部署为tomcat/webapps/BeerApp之下一个正常的目录结构。要把它部署为一个WAR文件，需要压缩BeerApp目录中的所有内容（不过，不包括BeerApp目录本身），然后把得到的JAR文件命名为BeerApp.war。接下来将BeerApp.war文件放在tomcat/webapps目录中。这样就行了。Tomcat会解开WAR，使用WAR文件名创建Web应用上下文目录。不过，再说一次，你的容器可能会以不同的方式处理WAR部署和命名。这里我们只关心规范有什么要求，要知道：不论Web应用是否部署在一个WAR中，几乎没有任何区别！换句话说，还是需要有WEB-INF、web.xml等。前一页的所有要求仍然适用。

**“几乎”一切都适用。**但有一点可能不同，有一件事在使用WAR文件时是可以做的，但没有WAR文件时不能做，**这就是声明库依赖性。**

在一个WAR文件中，可以在META-INF/MANIFEST.MF文件中声明库依赖性，这样在部署时就能检查容器能否找到应用依赖的包和类。这说明，请求到来时，如果容器在其类路径中没有找到所请求资源需要的特定类，就会出问题，而你不必等到真正请求资源时才发现这个问题。



### 不要被WAR文件的问题唬住……规则

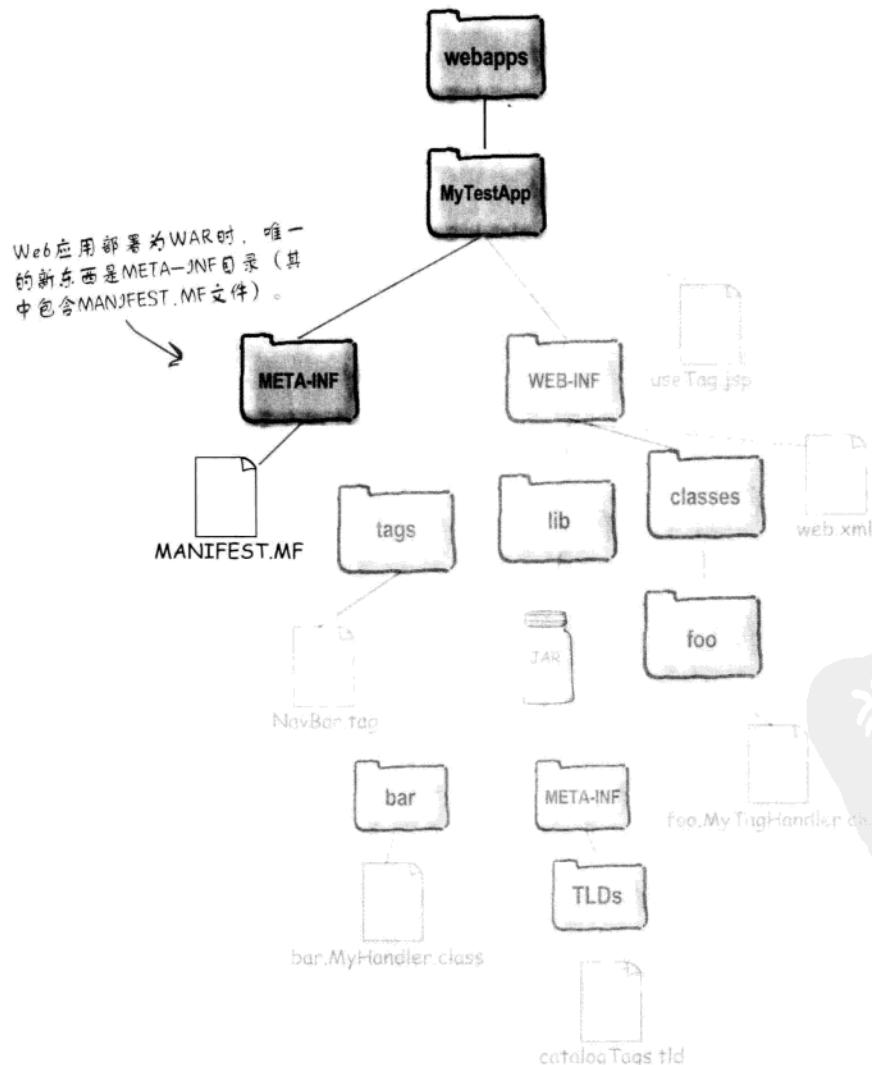
没有变！

快答题：如果应用部署为WAR，还需要一个名为“web.xml”的文件吗？当然。如果部署为WAR，还需要一个“WEB-INF”目录吗？当然。还需要把类放在“WEB-INF”下的“classes”目录吗？当然。这下你知道了吧。规则并不会因为你把应用目录中吗？当然。这下你知道了吧。规则并不会因为你把应用放在一个WAR中就有所改变。唯一一个重要的区别是WAR文件在Web应用上下文下有一个“META-INF”目录（对应于“WEB-INF”目录）。



# 部署后的WAR文件是什么样子

通过把WAR文件放在webapps目录中，在Tomcat中部署Web应用时，Tomcat会解开WAR文件，创建上下文目录（在这个例子中就是MyTestApp），这里只有一个新内容：这就是META-INF目录（其中包含一个MANIFEST.MF文件）。你自己可能不会在META-INF目录中放任何东西，所以不用关心应用是否部署为一个WAR，除非你确实需要在MANIFEST.MF文件中指定库依赖性。



## 使静态内容和JSP可以直接访问

部署静态HTML和JSP时，可以选择是否允许从Web应用外部直接访问。所谓可直接访问是指客户在浏览器中输入资源的路径，服务器就会返回这个资源。但是只要把文件放在WEB-INF下就能避免直接访问，或者如果应用部署为一个WAR文件，可以把不允许直接访问的文件放在META-INF下。

这是Web应用中一个可以  
直接访问的路径。

### 合法请求

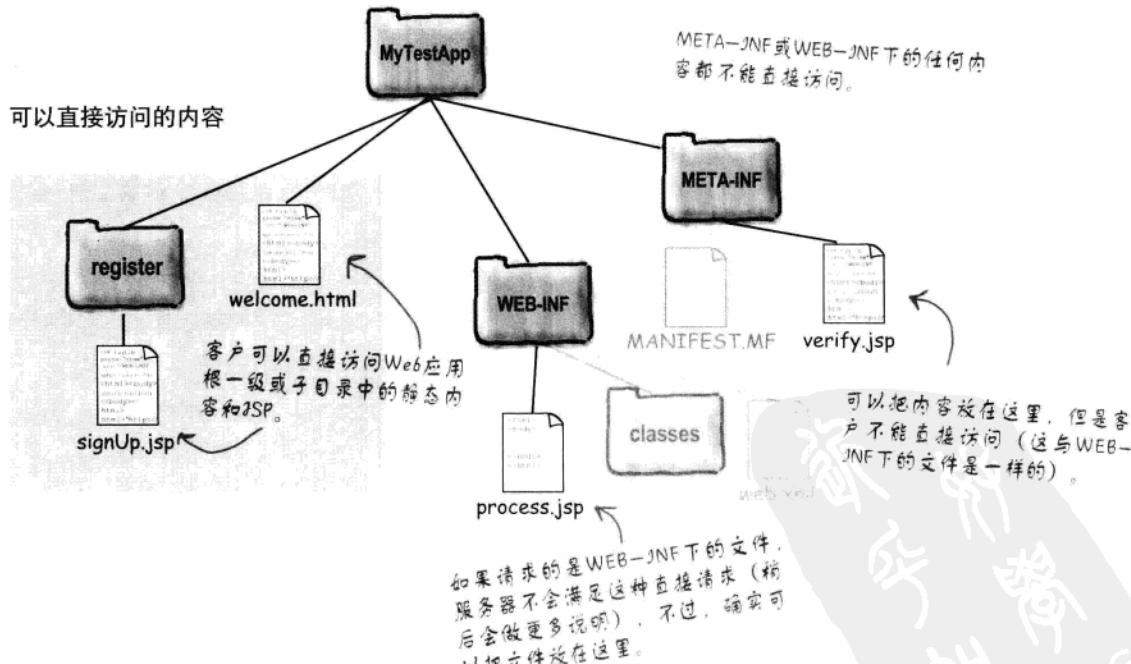
<http://www.wickedlysmart.com/MyTestApp/register/signUp.jsp>

不行！WEB-INF下的任何  
东西都不能直接访问。

### 不合法的请求（产生“404 Not Found”错误）

<http://www.wickedlysmart.com/MyTestApp/WEB-INF/process.jsp>

© 2004 WickedlySmart.com. All rights reserved.



如果服务器得到的客户请求需要WEB-INF或META-INF下的文件，容器肯定会响应一个404 NOT FOUND错误！

# there are no Dumb Questions

**问：** 如果不能对外提供WEB-INF或META-INF中的内容，那么把页面放在这里有什么意义？？！

**答：** 想想看，你可能有一些能在包一级访问（默认访问级别）的Java类和类成员，对不对？这些类和成员不能“公共”访问，只供其他公共类和成员在内部使用。不可直接访问的静态内容和JSP也是一样。把它们放在WEB-INF下（或者如果应用部署为WAR文件，则放在META-INF下），就能保护它们不能由客户直接访问，但这个Web应用的其他部分仍可使用这些静态内容和JSP。

例如，你可能想转发到一个文件或包含一个文件，而且希望确保任何客户都不能直接请求这个文件。如果想保护一个资源，防止直接访问，要使用WEB-INF而不是META-INF，但在考试中，你要知道这个规则对二者都适用。

**问：** 那么WEB-INF/lib下JAR文件中的META-INF目录呢？是不是和WAR文件中的META-INF一样能提供保护呢？

**答：** 嗯……对。但事实是，内容是否放在META-INF中并不重要。在这种情况下，你说的是一个位于WEB-INF下lib目录中的JAR文件，只要是在WEB-INF下，就会得到保护，不允许直接访问！所以，内容究竟在WEB-INF下的哪个具体位置并不重要，不论放在哪里都会得到保护。我们说META-INF是受保护的，实际上是指WAR文件中的META-INF，而WEB-INF/lib下JAR文件中的META-INF总会得到保护，原因是它们位于WEB-INF目录下。

**问：** 前一页上你提到了要在META-INF/MANIFEST.MF文件中指定库依赖性。必须这样做吗？不是说WEB-INF/lib jar文件和WEB-INF/classes目录中的所有类都会自动放在这个应用的类路径上吗？

**答：** 不错，如果使用WEB-INF/classes目录或WEB-INF/lib中的一个JAR文件在Web应用中部署类，那么这些

类就能自动得到，无需你额外说什么或做什么，它们自然会工作的。但是……你的容器的类路径上可能有一些可选的包，你也许会依赖其中某些包，或者可能会依赖某个特定版本的库！有了MANIFEST.MF文件，就能告诉容器你必须访问哪些可选的库。如果容器无法提供这些库，就不能成功地部署应用。否则，如果你部署了应用，然后到请求时才发现遭遇可怕的运行时错误（或者更糟糕，运行时错误可能很微妙），相比之下，无法部署反而更好一些。

**问：** 容器怎样访问WEB-INF/lib中JAR文件里的内容？

**答：** 容器自动地将JAR文件放在其类路径中，所以servlet、监听者、bean等类都可用，就好像你把这些类放在WEB-INF/classes目录中一样（当然，要按正确的包目录结构）。换句话说，类在不在JAR中并不重要，只要它们们在正确的位置上就行。

不过，要记住，容器查看WEB-INF/lib中的JAR文件之前，会先查找WEB-INF/classes目录中的类。

**问：** OK，类文件我搞清楚了，那其他类型的文件呢？如果我需要访问一个文本文件，它部署在WEB-INF/lib中的一个JAR里，该怎么做？

**答：** 这就不同了。如果你的Web应用代码需要直接访问一个资源（文本文件、JPEG等），而这个资源在一个JAR中，就要使用类加载器（classloader）的getResource()或getResourceAsStream()方法，这只是普通的J2SE机制，并非servlet所特有。

你可能已经知道这两个方法（getResource()和getResourceAsStream()），因为ServletContext API中也有这两个方法。区别在于，ServletContext中的方法只用于Web应用中未部署在JAR文件中的资源（考试要求你知道：可以使用标准J2SE机制从JAR文件得到资源，但是无需了解任何细节）。

# servlet映射到底是怎么回事

从最早的教程开始，在前面章节使用的部署描述文件中，你已经见过一些servlet映射的例子了。

每个servlet映射都有两部分——`<servlet>`元素和`<servlet-mapping>`元素。

`<servlet>`定义一个servlet名和类，`<servlet-mapping>`定义了映射至一个servlet名（在DD中的另外某个位置上定义）的URL模式。

```
<web-app ...>
  <servlet>
    <servlet-name>Beer</servlet-name>
    <servlet-class>com.example.BeerSelect</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Beer</servlet-name>
    <url-pattern>/Beer/SelectBeer.do</url-pattern>
  </servlet-mapping>
</web-app>
```

这个名主要用于DD的其他部分。  
这不是客户知道的名字。

```
<servlet>
  <servlet-name>Beer</servlet-name>
  <servlet-class>com.example.BeerSelect</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>Beer</servlet-name>
  <url-pattern>/Beer/SelectBeer.do</url-pattern>
</servlet-mapping>
```

```
</web-app>
```

如果客户请求要的是“/Beer/SelectBeer.do”，这是指名为“Beer”的servlet。



如果有这样的请求到来，容器会在`<servlet>`元素中找到匹配的`<servlet-name>`，得出哪个类负责处理这个请求。

我看到这有一个`<servlet>`的`<servlet-name>`是“Beer”，它告诉我哪个servlet类处理这个请求。



但是我  
没看到一个名为“Beer”  
的目录呀？而且也没有  
一个名为“SelectBeer.do”  
的文件。

这是这个特定Web应  
用的上下文/根。

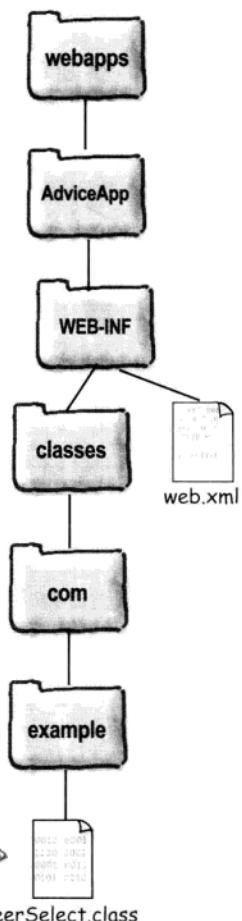
这只在DD中有意义！

<http://www.wickedlysmart.com/AdviceApp/Beer>SelectBeer.do>

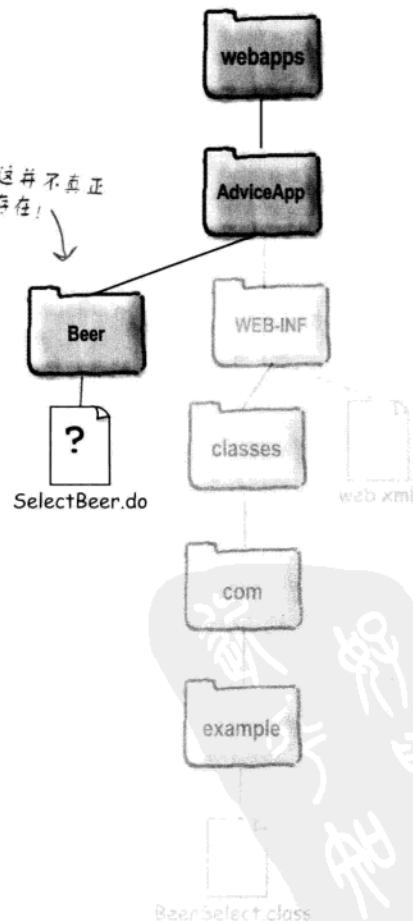
具体的（物理）  
目录结构



这是处理/Beer/  
BeerSelect.do请求  
的Servlet。



虚拟的（逻辑）  
目录结构



# Servlet映射可能是“假的”

servlet映射中的URL模式可能完全是假的。这是虚拟的、假想的。只是提供给客户的一个逻辑名。这些客户不需要了解Web应用的实际物理结构。

利用servlet映射，要组织建立两个结构：实际的物理目录和文件结构，这也是Web应用资源所在的具体结构，还有一个是虚拟/逻辑结构。

三种<url-pattern>元素

## ① 完全匹配

```
<url-pattern>/Beer/SelectBeer.do</url-pattern>
```

必须以一个斜线 (/) 开头。 可以有扩展名，但不必要。

## ② 目录匹配

```
<url-pattern>/Beer/*</url-pattern>
```

必须以一个斜线  
开头 (/)。 总是以一个斜线加星  
号 (\*) 结束。  
可以是一个虚拟目录  
或实际目录。

## ③ 扩展名匹配

```
<url-pattern>*.do</url-pattern>
```

必须以一个星号(\*)开  
头（不能以斜线开  
头）。 星号的后面必须有一个点加  
扩展名（.do, .jsp等）。

称存在虚拟/逻辑结构，  
只是因为你说它存在（而  
并非实际存在）！

DD中的URL模式不会  
映射到DD中<servlet-  
name>元素以外的其他元  
素。

<servlet-name>元素是  
servlet映射的键，它们把  
请求<url-pattern>映射  
到具体的servlet类。

要点：客户按<url-  
pattern>请求servlet，  
而不是按<servlet-  
name>或<servlet-  
class>！

# 作为容器

给定所示的DD  
Servlet映射和客户请求，容器会选择哪个  
servlet? 实际考试中也会有  
这种题目。

映射:

```
<servlet>
  <servlet-name>One</servlet-name>
  <servlet-class>foo.DeployTestOne</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>One</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Two</servlet-name>
  <servlet-class>foo.DeployTestTwo</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Two</servlet-name>
  <url-pattern>/fooStuff/bar</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Three</servlet-name>
  <servlet-class>foo.DeployTestThree</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Three</servlet-name>
  <url-pattern>/fooStuff/*</url-pattern>
</servlet-mapping>
```

## 有关servlet映射的重要规则:

- 1) 容器会按上一页显示的顺序查找匹配。换句话说，首先查找完全匹配。如果找不到完全匹配，再查找目录匹配，如果目录匹配也找不到，就查找扩展名匹配。
- 2) 如果一个请求与多个目录<url-pattern>匹配，容器会选择最长的匹配。换句话说，如果请求/foo/bar/myStuff.do，它就会映射到<url-pattern> /foo/bar/\*，尽管这个请求与<url-pattern> /foo/\*也匹配，但是前者更长一些。总是取最特定的匹配。

请求:

http://localhost:8080/MapTest/blue.do

容器选择:

http://localhost:8080/MapTest/fooStuff/bar

容器选择:

http://localhost:8080/MapTest/fooStuff/bar/blue.do

容器选择:

http://localhost:8080/MapTest/fooStuff/blue.do

容器选择:

http://localhost:8080/MapTest/fred/blue.do

容器选择:

http://localhost:8080/MapTest/fooStuff

容器选择:

http://localhost:8080/MapTest/fooStuff/bar/foo.fo

容器选择:

http://localhost:8080/MapTest/fred/blue.fo

容器选择:

# 作为容器

答案



映射：

```
<servlet>
  <servlet-name>One</servlet-name>
  <servlet-class>foo.DeployTestOne</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>One</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Two</servlet-name>
  <servlet-class>foo.DeployTestTwo</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Two</servlet-name>
  <url-pattern>/fooStuff/bar</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Three</servlet-name>
  <servlet-class>foo.DeployTestThree</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Three</servlet-name>
  <url-pattern>/fooStuff/*</url-pattern>
</servlet-mapping>
```

请求：

http://localhost:8080/MapTest/blue.do  
容器选择：DeployTestOne  
(与\*.do 扩展名模式匹配)

http://localhost:8080/MapTest/fooStuff/bar  
容器选择：DeployTestTwo  
(与/fooStuff/bar模式完全匹配)

http://localhost:8080/MapTest/fooStuff/bar/blue.do  
容器选择：DeployTestThree  
(与/fooStuff/\* 目录模式匹配)

http://localhost:8080/MapTest/fooStuff/blue.do  
容器选择：DeployTestThree  
(与/fooStuff/\*目录模式匹配)

http://localhost:8080/MapTest/fred/blue.do  
容器选择：DeployTestOne  
(与\*.do扩展名模式匹配)

http://localhost:8080/MapTest/fooStuff  
容器选择：DeployTestThree  
(与/fooStuff/\*目录模式匹配)

http://localhost:8080/MapTest/fooStuff/bar/foo  
容器选择：DeployTestThree  
(与/fooStuff/\*目录模式匹配)

http://localhost:8080/MapTest/fred/blue.foo  
容器选择：404 NOT FOUND  
(均不匹配)

下一页练习的答案：

: 1) DeployTestFour 2) DeployTestTwo

# 小问题……

为了保证你确实理解了servlet映射，下面再举一个例子。不要跳过这个例子，请仔细看映射和请求（这个“作为容器”小练习的答案在上一页的最下面，不要偷看）。

## 作为容器

容器选择哪个  
servlet?



### DD中的映射

```
<servlet>
    <servlet-name>Two</servlet-name>
    <servlet-class>foo.DeployTestTwo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Two</servlet-name>
    <url-pattern>/fooStuff/bar</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>Four</servlet-name>
    <servlet-class>foo.DeployTestFour</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Four</servlet-name>
    <url-pattern>/fooStuff/bar/*</url-pattern>
</servlet-mapping>
```

请求：

- ① <http://localhost:8080/test/fooStuff/bar/>

容器选择：

- ② <http://localhost:8080/test/fooStuff/bar>

容器选择：

## 在DD中配置欢迎文件

你已经知道，如果键入Web网站名，而没有指定特定的文件，通常还是能看到一个页面。在你的浏览器中输入`http://www.oreilly.com`，就会带你来到O'Reilly网站，尽管你并没有指定一个特定的资源（如“home.html”），但仍能得到一个默认页面。

可以配置服务器为整个网站定义一个默认页面，但是这里关心的是如何为各个Web应用配置默认页面（也称为“欢迎”页面）。你在DD中配置欢迎页面，DD再确定当客户输入一个部分URL时容器要选择哪个页面，所谓部分URL是指这个URL只有一部分（例如，一个目录），而不包括目录中的特定资源。

换句话说，如果客户有以下请求，会怎么样呢？

`http://www.wickedlysmart.com/foo/bar` ← “bar” 只是一个目录

这里“bar”只是一个目录，没有特定的servlet与这个URL模式匹配。客户会看到什么？

DD中：

```
<web-app ...>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

← 不能以斜线开头  
或结束！

假设有一个Web应用，其中多个不同的目录有自己的默认HTML页面，名为“index.html”。但是还有些目录使用的是“default.jsp”。如果要为需要默认页面或JSP的各个目录分别指定，这就太麻烦了。可以不这样，而是按顺序指定一个页面列表，对于部分请求指定的目录，你希望容器能按你指定的页面顺序在这个目录中查找。换句话说，不论请求哪个目录，容器总是按同一个列表查找，也就是这个唯一的`<welcome-file-list>`。

容器会从`<welcome-file-list>`所列的第一个欢迎文件开始，选择找到的第一个匹配。



多个欢迎文件放在一个DD元素中。

不论可能列出多少个欢迎文件，它们都要放在DD中的一个元素中：`<welcome-file-list>`。你可能会认为每个文件应该放在一个单独的`<welcome-file>`元素里，但事实上并非如此！每个文件有其自己的`<welcome-file>`元素，但是所有这些元素都放在一个`<welcome-file-list>`中。



`<welcome-file>`元素中的文件不以斜线开头！

别弄错了。容器匹配和选择欢迎文件的方法与匹配URL模式的方法不同。如果在文件名前面放上斜线，就会违反规范，接下来会有糟糕的事情等着你。

# 作为容器

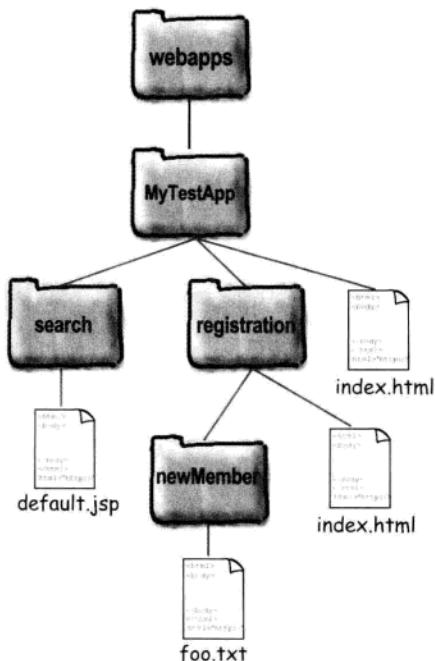
给定所示的DD和客户请求，容器会选择哪些欢迎文件？考试中也会有这种题目。



## DD:

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

## 目录结构:



### 请求:

<http://localhost:8080/MyTestApp/>

### 容器选择:

<http://localhost:8080/MyTestApp/registration>

### 容器选择:

<http://localhost:8080/MyTestApp/search>

### 容器选择:

<http://localhost:8080/MyTestApp/registration/newMember>

### 容器选择:

# 作为容器

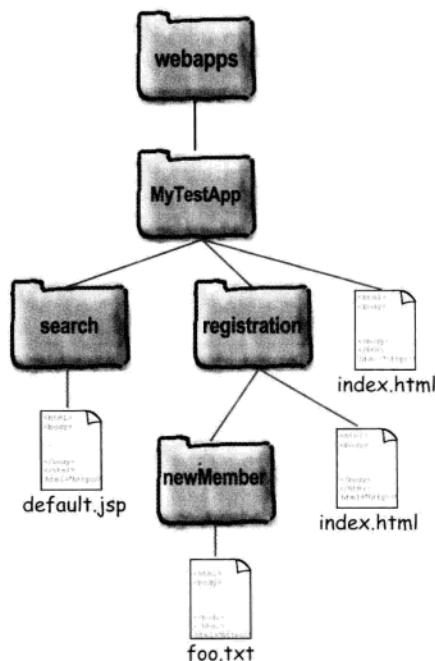
答案



DD:

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

目录结构:



请求:

<http://localhost:8080/MyTestApp/>

容器选择:

MyTestApp/index.html

<http://localhost:8080/MyTestApp/registration/>

容器选择:

MyTestApp/registration/index.html

<http://localhost:8080/MyTestApp/search>

容器选择:

MyTestApp/search/default.jsp

(如果“search”目录中同时有一个default.jsp和一个index.html，容器会选择“index.html”文件，因为这是DD中所列的第一个文件。)

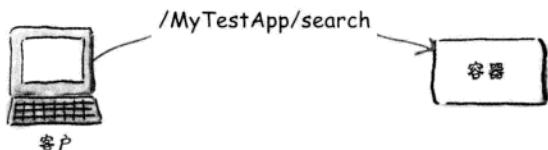
<http://localhost:8080/MyTestApp/registration/newMember>

容器选择:

如果找不到<welcome-file-list>中的文件，具体行为根据开发商不同而有所不同。Tomcat会显示newMember的一个目录列表（将显示“foo.txt”），其他容器则可能显示一个“404 Not Found”错误。

# 容器如何选择欢迎文件

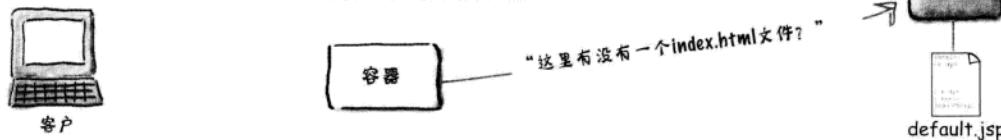
- ① 客户请求: `http://www.wickedlysmart.com/MyTestApp/search`



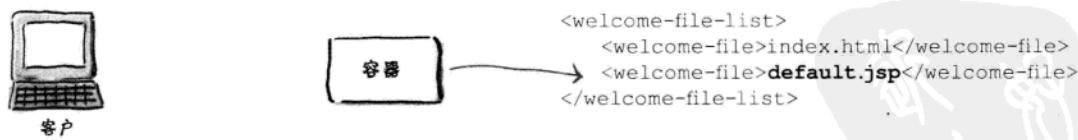
- ② 容器在DD中寻找servlet映射，没有找到匹配。接下来，容器在`<welcome-file-list>`中查找，在最前面发现“index.html”。



- ③ 容器在`/MyTestApp/search`目录查找一个“index.html”文件，但是没有找到。



- ④ 容器在DD的`<welcome-file-list>`中查找下一个`<welcome-file>`，看到“default.jsp”。



- ⑤ 容器在`/MyTestApp/search`目录中查找一个“default.jsp”文件，而且找到了，向客户提供其响应。



# 在DD中配置错误页面

当然，如果用户不知道访问你的网站或Web应用时应该具体请求哪一个资源，你想更友好一些，所以你指定了默认/欢迎文件。但是，如果出了问题你是不是也希望友好一些？这一点在“使用定制标记”一章中我们已经有所了解，所以这里只是一个复习。

## 声明一个“普遍”型错误页面

这应用于Web应用中的所有资源，而不只是JSP。

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/errorPage.jsp</location>
</error-page>
```

(另外：可以增加一个有errorCode属性的page指令，覆盖单个JSP的错误页面设置。)

## 为更明确的异常声明一个错误页面

以下配置了一个错误页面，只有存在ArithmaticException异常时才调用这个错误页面。如果既有这个声明，又有以上的“普遍”型声明，那么只要不是ArithmaticException，其他异常都会导致调用“errorPage.jsp”。

```
<error-page>
    <exception-type>java.lang.ArithmaticException</exception-type>
    <location>/arithmeticError.jsp</location>
</error-page>
```

## 根据一个HTTP状态码声明错误页面

以下配置了一个错误页面，只有响应的状态码是“404”（文件未找到）时才调用这个错误页面。

```
<error-page>
    <error-code>404</error-code>
    <location>/notFoundError.jsp</location>
</error-page>
```



# there are no Dumb Questions

**问：** <exception-type>中可以声明哪些异常类型?

**答：** 只要是Throwable就行，所以这包括java.lang.Error、运行时异常和所有受查异常（当然，受查异常类要在容器的类路径上）

**问：** 谈到错误处理，能自己通过程序生成错误码吗？

**答：** 能。可以调用HttpServletResponse的sendError()方法，它会告诉容器生成错误，就好像是容器自己生成的一样。如果已经根据这个错误码配置了一个要发送给客户的错误页面，客户就会得到这个错误页面。顺便说一句，“错误”码也称为“状态”码，所以如果看到这两种说法，要知道它们指的都是同一个东西，都是错误的HTTP码。



**问：** 举一个自己生成错误码的例子好吗？

**答：** 好的，下面就是一个例子：

```
response.sendError(HttpServletResponse.SC_FORBIDDEN);
```

它与下面这行代码是一样的：

```
response.sendError(403);
```

如果查找HttpServletResponse接口，你会发现为常用的HTTP错误/状态码定义了许多常量。要记住，在考试中，不需要记住这些状态码！你要知道你能生成错误码，而且生成错误码的方法是response.sendError()。另外从DD中定义的错误页面来看，或者从JSP中的任何其他错误处理来看，HTTP错误由容器生成还是由程序员生成并没有区别。你知道这么多就足够了。403就是403，不论它是谁发出的错误。哦，对了，还有一个重载的两参数sendError()版本，它要取一个int和一个String消息作为参数。

## 在DD中配置servlet初始化

你已经知道，servlet默认地会在每一个请求到来时初始化。这说明，第一个客户要承受类加载、实例化和初始化（建立一个ServletContext、调用监听者等）等一系列开销，然后容器才能正常地工作：分配一个线程，并调用servlet的service()方法。

如果你希望在部署时（或在服务器重启时）加载servlet，而不是等到第一个请求到来时才加载，可以在DD中使用<load-on-startup>元素。如果<load-on-startup>的值非负，就是在告诉容器要在应用部署时（或服务器重启时）初始化servlet。

如果你想预加载多个servlet，而且想控制它们初始化的顺序，完全可以靠<load-on-startup>的值来决定顺序！换句话说，非负值就意味着要早加载，不过servlet加载的具体顺序则取决于不同<load-on-startup>元素的值。

### DD中

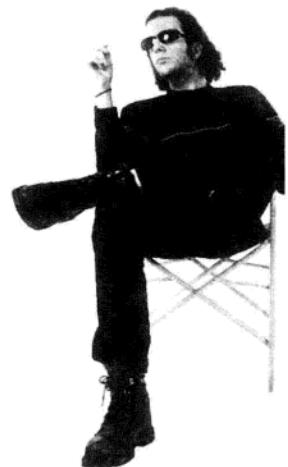
```
<servlet>
    <servlet-name>KathyOne</servlet-name>
    <servlet-class>foo.DeployTestOne</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

只要是大于0，就表示“在部署时或服务器启动时初始化这个servlet，而不要等到第一个请求到来才初始化。”

**问：**难道你不这样用吗？每个人都应该默认地使用<load-on-startup>1</load-on-startup>，难道不是吗？

**答：**要回答这个问题，你先问问自己，“我的应用里有多少个servlet，它们都会怎么用？”另外你还要问一问“每个servlet加载要花多长时间？”有些servlet很少使用，所以你可能想节省资源，不提前加载很少使用的servlet。但是有些servlet初始化的时间太长（如Struts ActionServlet），就算只有一个客户要经受这种延迟你也于心不忍。所以，只有你自己能决定，可能要根据每个servlet加载的时间和使用的频度对各个servlet分别确定。

作为第一个请求servlet的客户真是倒霉，除非开发人员使用了<load-on-startup>。



**值大于0并不影响servlet实例的个数！**

如果你使用的值是：  
<load-on-startup>4</load-on-startup>，  
这并不表示“加载4个servlet实例”。  
它只是说这个servlet应当在<load-on-startup>值小于4的其他servlet加载后  
再加载。  
如果有多个servlet的<load-on-startup>都是4怎么办？容器加载有  
相同<load-on-startup>值的servlet时，  
会按DD中声明这些servlet的顺序来  
加载。

# 建立一个XML兼容的JSP：JSP文档

这个内容放在别的任何地方都不合适，所以我们决定把它作为这一章的内容，这是因为我们一直在大谈XML。考试不要求你是一个XML专家，但是有两点你必须知道：关键DD元素的语法，以及怎么才算一个JSP文档（“要不是什么呢？如果一个正常的JSP不是JSP文档，那它是什么？”你是不是想问这个问题？可以这样来考虑，正常的JSP是一个页面，除非采用了与正常JSP语法对应的XML语法来编写，这样来它就成为一个JSP文档）。

这说明，实际上可以用两种语法来建立一个JSP。不论是哪种语法，灰色的文本都是一样的。

## 正常的JSP页面语法

## JSP文档语法

### 指令

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

```
<jsp:directive.page import="java.util.*"/>
```

(taglib除外)

### 声明

```
<%! int y = 3; %>
```

```
<jsp:declaration>
    int y = 3;
</jsp:declaration>
```

### Scriptlet

```
<% list.add("Fred"); %>
```

```
<jsp:scriptlet>
    list.add("Fred");
</jsp:scriptlet>
```

### 文本

```
There is no spoon.
```

```
<jsp:text>
    There is no spoon.
</jsp:text>
```

### 脚本表达式

```
<%= it.next() %>
```

```
<jsp:expression>
    it.next()
</jsp:expression>
```



Relax 考试中有关JSP文档只会考这么多。

我们不想对此说太多，因为编写XML兼容的JSP文档可能不是你要做的事情。XML语法主要由工具使用，上表只是显示了工具如何将正常的JSP语法转换为一个XML文档。如果你想自己手工编写，还有很多内容需要了解，例如，整个文档必须包围在一个`<jsp:root>`标记中(它还包括一些其他的元素)，taglib指令要放在`<jsp:root>`开始标记里，而不是作为一个`<jsp:directive>`。不过，考试只会考上表中所列的内容，所以不用担心。

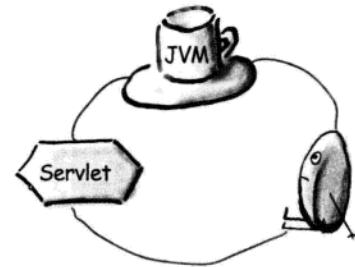
## 记住与EJB有关的DD标记

考试要考的是Web组件，而不是业务组件（不过，在关于“模式”的一章中会对业务组件稍做介绍）。但是，如果你在部署一个J2EE应用，业务层使用了Enterprise JavaBeans (EJB)，你的一些Web组件可能就需要查找和访问企业bean。如果在一个完备的J2EE兼容容器（其中还包括一个EJB容器）中部署应用，可以在DD中定义EJB的引用。除了在DD中声明的EJB引用外，考试中不要求你对EJB知道多少，所以这里不再赘述（注1）。

### 本地bean的引用

```
<ejb-local-ref>
  <ejb-ref-name>ejb/Customer</ejb-ref-name>
  代码中要用的JNDI查找名
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>com.wickedlysmart.CustomerHome</local-home>
  <local>com.wickedlysmart.Customer</local>
</ejb-local-ref>
```

这些必须是bean的公开接口  
的完全限定名。



本地bean是指：客户（在这里就是一个servlet）和bean必须在同一JVM中运行

### 远程bean的引用

```
<ejb-ref>
  <ejb-ref-name>ejb/LocalCustomer</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.wickedlysmart.CustomerHome</home>
  <remote>com.wickedlysmart.Customer</remote>
</ejb-ref>
```

（这些标记的可选子元素包括：`<description>`和  
`<ejb-link>`，但考试对这个内容不做要求）。



远程bean是指：客户（在这里是一个servlet）和bean可能在不同的JVM中运行（可能还在不同的物理主机上）。

注1：不过，如果你确实对EJB感兴趣，我们推荐一本非常棒的书……

（《Head First EJB》）



本地和远程标记不一致!

本地和远程bean DD标记有两个相同的元素：  
`<ejb-ref-name>`列出代码中在企业bean home接口上完成JNDI查找时所用的逻辑查找名。(如果你以前没用过EJB，甚至不明白这句话是什么意思，也不用担心，考试不要求你掌握EJB的知识)。`<ejb-ref-type>`描述了这是一个实体bean还是会话bean。这两个元素(查找名和bean类型)与bean是否是本地bean没有关系，也就是说bean是本地的(在Web组件所在JVM中运行)还是远程的(可能在另外一个JVM中运行)对这两个元素没有影响。

不过……再看看其他元素，从外层标记开始：`<ejb-local-ref>`和`<ejb-ref>`。你可能以为它应该是：

`<ejb-local-ref>` ← 对

~~`<ejb-remote-ref>`~~ ← 错！！

但是，不是这样的！对于远程bean，只应该是：

`<ejb-ref>` ← 对！标记中没有“remote”这个词。

换句话说，本地引用指出它是本地的，但是远程引用在标记元素名中没有“remote”这个词。为什么？因为最初定义`<ejb-ref>`时，根本没有“本地”EJB这样的东西。因为那时所有企业bean都是“远程的”；没有必要区别本地和远程，所以没有必要在标记名中加上“remote”一词。

这也解释了另一个标记命名的不一致，对应Bean home接口的标记名也不一致。本地bean使用以下标记：

`<local-home>` ← 对的

但远程bean使用的不是：

~~`<remote-home>`~~ ← 错！！

对于远程bean只应该是：

`<home>`



## 记住JNDI <env-entry> DD 标记

如果你熟悉EJB和/或JNDI，这就很有意义。如果你不熟悉，其实对考试也没有影响，因为考试只要求你记住这个标记就行了。（有关JNDI环境项的详细内容在EJB/J2EE书里都有介绍，比如说那本很棒的《Head First EJB》。）

可以把环境项认为是你的应用可以使用的某种部署时常量，就像servlet和上下文初始化参数一样。换句话说，部署人员可以通过环境项把值传入servlet（或者如果EJB作为企业应用的一部分部署在一个完全J2EE兼容的服务器中，还可以通过环境项将值传入EJB）。

在部署时，容器读取DD，使用你在DD标记中提供的名和值，建立一个JNDI项（再次指出，假设这是一个完全J2EE兼容的应用，而不只是一个只有Web容器的服务器）。在运行时，应用中的组件可以使用DD中所列的名字在JNDI中查找值。你可能不关心<env-entry>，除非你正在使用EJB进行开发。之所以要记住这个标记，只是因为考试中可能会考。

### 声明应用的JNDI环境项

```
<env-entry>
  <env-entry-name>rates/discountRate</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>10</env-entry-value>
</env-entry>
```

这会作为一个String传入（或者，如果<env-entry-type>是java.lang.Character，则作为单个Character传入）。

代码中使用的查找名。

这可以是任何类型，只要这种类型取一个String作为构造函数参数就行（或者，如果类型是java.lang.Character，构造函数则只取一个Character参数）。



<env-entry-type>不能是基本类型！

看到一个有整数值的<env-entry-value>时（如上例），你可能认为<env-entry-type>可以是一个基本类型。但是……这样想是错的。

你还可能认为只能是String和包装器类型，不过这种想法也是错的，只要是构造函数取一个String参数的任何类型都可以（或者对于Character类型构造函数要取一个Character参数）。

注意：还可以包括一个可选的<description>，加上这个描述很有好处。

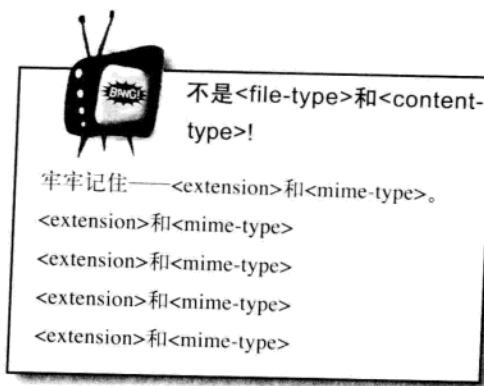
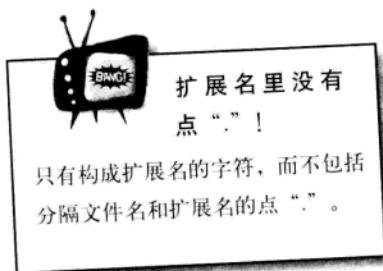
## 记住<mime-mapping> DD 标记

在DD中可以配置扩展名和MIME类型之间的映射。这可能是最容易记的标记了，因为它很好理解，你要建立扩展名（extension）和一个MIME类型（mime-type）之间的映射，猜猜看怎么来建立这个映射？非常简单，非常清晰，子元素名就是“extension”和“mime-type”。这说明，你只需记住一点：这些标记元素是什么就用什么来命名！

除非你刚开始把它想成是“file-type”（文件类型）和“content-type”（内容类型）。不要这样想。应该这样来记。

声明<mime-mapping>

```
<mime-mapping>
    <extension>mpg</extension>
    <mime-type>video/mpeg</mime-type>
</mime-mapping>
```





## 都放在哪里？

填写下表，明确地说明给定资源必须放在Web应用中的哪个位置。我们已经填了第一个空。答案在636页上。

资源类型	部署位置
部署描述文件(web.xml)	直接放在WEB-INF中（这个目录就在Web应用的根目录下）。
标记文件 (.tag或.tagx)	
HTML和JSP (可以直接访问的 HTML和JSP)	
HTML和JSP(你想“隐藏”的HTML和JSP，不 允许客户直接访问)	
TLD (.tld)	
Servlet类	
标记处理器类	
JAR文件	



## 记住DD标记

如果你不打算参加考试，那这些空填错了也没关系（不过，最后两个元素对每个人都很重要）。

如果你想参加考试，一定要花些时间把这些都记住。

```

<_____>
<_____ejb/Customer<_____>
<ejb-ref-type>Entity</ejb-ref-type>
<____>com.wickedlysmart.CustomerHome<____>
<local>com.wickedlysmart.Customer</local>
<_____>


---


<ejb-ref>
  <_____ejb/LocalCustomer<_____>
  <ejb-ref-type>Entity</ejb-ref-type>
  <____>com.wickedlysmart.CustomerHome<____>
  <____>com.wickedlysmart.Customer<____>
</ejb-ref>


---


<env-entry>
  <_____>rates/discountRate<_____>
  <_____>java.lang.Integer<_____>
  <env-entry-value>10</env-entry-value>
</env-entry>


---


<error-page>
  <_____>java.io.IOException<_____>
  <____>/myerror.jsp<____>
</error-page>


---


<_____>
  <welcome-file>index.html</welcome-file>
  <_____>

```



## 都放在哪里？

填写下表，明确地说明给定资源必须放在Web应用中的哪个位置。我们已经填了第一个空。

资源类型	部署位置
部署描述文件(web.xml)	直接放在WEB-INF中（这个目录就在Web应用的根目录下）。
标记文件 .tag或.tagx)	如果未部署在JAR中，标记文件必须放在WEB-INF/tags中，或WEB-INF/tags的一个子目录中。如果部署在一个JAR文件中，标记文件则必须放在META-INF/tags或META-INF/tags的一个子目录中。注意：如果标记文件部署在JAR中，那么在JAR中还必须有一个TLD。
HTML和JSP (可以直接访问的 HTML和JSP)	客户能访问的HTML和JSP可以放在Web的根目录下，或者它的任何子下，但是不能放在WEB-INF下（包括其子目录）。如果在WAR文件中，些页面不能放在META-INF下（包括其子目录）。
HTML和JSP(你想“隐藏”的HTML和JSP， 不允许客户直接访问)	客户不能直接访问WEB-INF（以及WAR文件中的META-INF）下的页面。
TLD .tld)	如果不不在JAR中，TLD文件必须放在WEB-INF下，或者放在WEB-INF的一个子目录下。如果部署在一个JAR中，TLD文件必须放在META-INF下，或者META-INF的一个子目录下。
Servlet类	Servlet类必须放在与包结构匹配的一个目录结构里，置于WEB-INF/classes下的一个目录中（例如，类com.example.Ring要放在WEB-INF/classes/com/example中），或者放在WEB-INF/lib下一个JAR文件里的适当包结构中。
标记处理器类	实际上，Web应用所用的所有类（除非是类路径上类库的一部分）都必须像servlet类一样遵循同样的规则，要放在WEB-INF/classes下，而且要有与包结构匹配的目录结构（或者放在WEB-INF/lib下一个JAR文件里的适当包结构中）。
JAR文件	JAR文件必须放在WEB-INF/lib目录中。



## 记住DD标记

答案

如果你要参加考试，一定要花些时间把这些都记住（还有这一章的其他标记，以及下一章介绍的一些与安全有关的标记）。

有“远程”接口的bean的引用。

利用环境项，可以将部署时常量传递到J2EE应用中。

告诉容器出现指定的`<exception-type>`时显示哪个页面。

告诉容器如果到来的请求未与指定的资源匹配，要查找哪个页面：`<welcome-file-list>`中可能指定了不止一个`<welcome-file>`。

有一个“本地”接口的bean的引用。

```
<ejb-local-ref>
<ejb-ref-name>ejb/Customer</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<local-home>com.wickedlysmart.CustomerHome</local-home>
<local>com.wickedlysmart.Customer</local>
</ejb-local-ref>
```

```
<ejb-ref>
<ejb-ref-name>ejb/LocalCustomer</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.wickedlysmart.CustomerHome</home>
<remote>com.wickedlysmart.Customer</remote>
</ejb-ref>
```

```
<env-entry>
<env-entry-name>rates/discountRate</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>10</env-entry-value>
</env-entry>
```

```
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/myerror.jsp</location>
</error-page>
```

```
<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>
```



## 第11章 模拟测验

1 DD中<init-param>元素可以出现在哪里？

(选出所有正确的答案)

- A. 作为<servlet>的子元素。
- B. 作为<web-application>元素的直接子元素。
- C. 放在文档类型声明后面。
- D. 希望声明一个上下文初始化参数时放在<context-param>元素中。

2 Web应用中标记库描述文件（TLD）存放在哪里？（选出所有正确的答案）

- A. 只能放在/WEB-INF/lib中。
- B. 只能放在/WEB-INF/classes中。
- C. 放在/WEB-INF/lib中一个JAR文件的/META-INF目录下。
- D. 放在应用的顶级目录。
- E. 在/WEB-INF或它的一个子目录中。

3 关于WAR文件哪些说法是正确的？（选出所有正确的答案）

- A. WAR代表Web应用资源文件（Web Application Resources file）。
- B. 合法的WAR文件必须包含一个部署描述文件。
- C. 多个WAR文件可以组成一个Web应用。
- D. WAR文件不能包含内嵌的JAR文件。

4 DD中声明了以下servlet:

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>com.myorg.ServletClass</servlet-class>
</servlet>
```

这个servlet类可以存放在Web应用的哪个位置? (选出所有正确的答案)

- A. 在JAR文件的/META-INF中。
- B. 在与包相关的目录树中 (这个目录树从应用目录的顶级目录开始)。
- C. 在/WEB-INF/classes中, 或者在/WEB-INF/lib下的一个JAR文件中。
- D. 在/WEB-INF/lib下且在JAR文件之外。

5 部署描述文件(DD)的作用是什么? (选出所有正确的答案)

- A. 允许代码生成工具根据一个XML文件动态地创建servlet。
- B. 从开发人员把Web应用配置信息传递给应用组装人员和部署人员。
- C. 配置应用中特定于开发商的方面。
- D. 只配置Web应用的数据库和EJB访问。

6 web.xml存放在WAR文件的哪个位置? (选出所有正确的答案)

- A. 在/WEB-INF/classes中。
- B. 在/WEB-INF/lib中。
- C. 在/WEB-INF中。
- D. 在/META-INF中。

7 给定：

```
10. <%@ page import="java.util.*" %>
11. <jsp:import import="java.util.*" />
12. <jsp:directive.page import="java.util.*" />
13. <jsp:page import="java.util.*" />
```

假设前缀“jsp”映射到命名空间

<http://java.sun.com/JSP/Page>.

以下哪些说法是正确的？（选出所有正确的答案）

- A. 第10和12行在任何类型的JSP页面中都是等价的。
- B. 第10行在JSP文档（基于XML的文档）中是不合法的。
- C. 第11行会正确地导入java.util包。
- D. 第12行会正确地导入java.util包。
- E. 第13行会正确地导入java.util包。

8

关于<init-param> DD元素，哪些说法是正确的？（选出所有正确的答案）

- A. 它们用于为特定的servlet声明初始化参数。
- B. 它们用于为整个Web应用声明初始化参数。
- C. 获取这些参数的方法的签名返回一个Object。
- D. 获取这些参数的方法取一个String。

9

哪些元素提供了对J2EE组件的JNDI访问？（选出所有正确的答案）

- A. <ejb-ref>
- B. <entity-ref>
- C. <ejb-local-ref>
- D. <session-ref>
- E. <ejb-remote-ref>

10

DD中注册了以下servlet:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>com.myorg.ActionClass</servlet-class>
</servlet>
```

选择这个servlet的正确映射。（选出所有正确的答案）

- A. 

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- B. 

```
<servlet-mapping>
    <servlet-name>com.myorg.ActionClass</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- C. 

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/controller</url-pattern>
</servlet-mapping>
```
- D. 

```
<servlet-mapping>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```
- E. 

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
</servlet-mapping>
```

11

可以为哪些类型的Web应用组件定义依赖性？（选出所有正确的答案）

- A. JSP文件
- B. WAR文件
- C. 类
- D. 库
- E. 清单文件

---

**12** 以下哪些是JSP文档（基于XML的文档）中的合法声明？  
(选出所有正确的答案)

- A. 

```
<jsp:declaration
    xmlns:jsp="http://java.sun.com/JSP/Page">
    int x = 0;
</jsp:declaration>
```
- B. 

```
<jsp:declaration
    xmlns:jsp="http://java.sun.com/JSP/Page">
    int x;
</jsp:declaration>
```
- C. 

```
<%! int x = 0; %>
```
- D. 

```
<%! int x; %>
```

---

**13** 哪些部署描述文件元素可以出现在<web-app>元素之前？(选出所有正确的答案)

- A. `<listener>`
- B. `<context-param>`
- C. `<servlet>`
- D. <web-app>元素的前面不能出现任何XML元素。

---

**14** 关于容器类加载器，哪些说法是正确的？(选出所有正确的答案)

- A. Web应用不能覆盖容器实现类。
- B. Web应用不能使用J2SE的getResource方法加载WAR文件中的资源。
- C. Web应用可以覆盖 javax.\* 命名空间中的任何J2EE类。
- D. Web开发人员可以覆盖J2EE平台类，条件是它们包含在WAR中的一个库JAR中。



## 第11章 模拟测验答案

1 DD中<init-param>元素可以出现在哪里?

(Servlet规范107页)

(选出所有正确的答案)

- A. 作为<servlet>的子元素。 B不对，因为web.xml不包含一个名为<web-application>的元素。
- B. 作为<web-application>元素的直接子元素。
- C. 放在文档类型声明后面。
- D. 希望声明一个上下文初始化参数时放在<context-param>元素中。 D不对，因为<context-param>元素不包含<init-param>。

2 Web应用中标记库描述文件(TLD)存放在哪里? (选出所有正确的答案)

(JSP规范196页)

- A. 只能放在/WEB-INF/lib中。
- B. 只能放在/WEB-INF/classes中。
- C. 放在/WEB-INF/lib中一个JAR文件的/META-INF目录下。
- D. 放在应用的顶级目录。
- E. 在/WEB-INF或它的一个子目录中。

如果TLD放在/WEB-INF/classes或/WEB-INF/lib中，容器无法自动发现这些TLD。

3 关于WAR文件哪些说法是正确的? (选出所有正确的答案)

(Servlet规范9.5 & 9.6)

- A. WAR代表Web应用资源文件(Web Application Resources file)。
- B. 合法的WAR文件必须包含一个部署描述文件。
- C. 多个WAR文件可以组成一个Web应用。
- D. WAR文件不能包含内嵌的JAR文件。

WAR代表Web归档(Web ARchive)，WAR文件中不能只包含Web应用中的一部分，WAR文件中只能是完整的应用。

4 DD中声明了以下servlet:

(Servlet 规范70页)

```
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.myorg.ServletClass</servlet-class>
</servlet>
```

这个servlet类可以存放在Web应用的哪个位置? (选出所有正确的答案)

- A. 在JAR文件的/META-INF中。
- B. 在与包相关的目录树中 (这个目录树从应用目录的顶级目录开始)。
- C. 在/WEB-INF/classes中, 或者在/WEB-INF/lib下的一个JAR文件中。
- D. 在/WEB-INF/lib下且在JAR文件之外。

D不对, 因为/WEB-INF/lib就设计为存放  
JAR文件的容器 (即用于存放JAR文件)。

5 部署描述文件(DD)的作用是什么? (选出所有正确的答案)

(Servlet 规范103页)

- A. 允许代码生成工具根据一个XML文件动态地创建servlet。
- B. 从开发人员把Web应用配置信息传递给应用组装人员和部署人员。
- C. 配置应用中特定于开发商的方面。
- D. 只配置Web应用的数据库和EJB访问。

D不对, 因为这只是  
DD作用的一部分。

6 web.xml存放在WAR文件的哪个位置? (选出所有正确的答案)

(Servlet 规范70页)

- A. 在/WEB-INF/classes中。
- B. 在/WEB-INF/lib中。
- C. 在/WEB-INF中。
- D. 在/META-INF中。

不论部署为一个WAR还是一个展开的目录结构,  
web.xml都应该存放在/WEB-INF中。

7 给定：

```

10. <%@ page import="java.util.*" %>
11. <jsp:import import="java.util.*" />
12. <jsp:directive.page import="java.util.*" />
13. <jsp:page import="java.util.*" />

```

(JSP v2.0 1~139页)

假设前缀“jsp”映射到命名空间

<http://java.sun.com/JSP/Page>.

以下哪些说法是正确的？（选出所有正确的答案）

A 不对，因为第10行在JSP文档（基于XML的文档）中是不合法的。

- A. 第10和12行在任何类型的JSP页面中都是等价的。
- B. 第10行在JSP文档（基于XML的文档）中是不合法的。
- C. 第11行会正确地导入 `java.util` 包。
- D. 第12行会正确地导入 `java.util` 包。
- E. 第13行会正确地导入 `java.util` 包。

C和E不对，因为它们不是 `http://java.sun.com/JSP/Page` 命名空间中的合法元素。

8

关于 `<init-param>` DD元素，哪些说法是正确的？（选出所有正确的答案）

(servlet 规范 SRV.B &amp; APJ)

- A. 它们用于为特定的servlet声明初始化参数。
- B. 它们用于为整个Web应用声明初始化参数。
- C. 获取这些参数的方法的签名返回一个 `Object`。
- D. 获取这些参数的方法取一个 `String`。

初始化参数可以有 Web应用作用域或servlet作用域。有servlet作用域的初始化参数在DD中指定为 `<init-param>`，取一个 `String`，并返回一个 `String`。有Web应用作用域的初始化参数在DD中指定为 `<context-param>`，也取一个 `String`，并返回一个 `String`。

9

哪些元素提供了对J2EE组件的JNDI访问？（选出所有正确的答案）

(servlet 规范 9.11)

- A. `<ejb-ref>`
- B. `<entity-ref>`
- C. `<ejb-local-ref>`
- D. `<session-ref>`
- E. `<ejb-remote-ref>`

另外，`<ejb-local-ref>` 也为Web应用创建者提供了J2EE组件的一个JNDI引用。

10

DD中注册了以下servlet:

(servlet 规范86页)

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>com.myorg.ActionClass</servlet-class>
</servlet>
```

选择这个servlet的正确映射。（选出所有正确的答案）

 A. <servlet-mapping>

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

 B. <servlet-mapping>

```
<servlet-name>com.myorg.ActionClass</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

B不对，因为它把servlet名与servlet类搞混了。

 C. <servlet-mapping>

```
<servlet-name>action</servlet-name>
<url-pattern>/controller</url-pattern>
</servlet-mapping>
```

 D. <servlet-mapping>

```
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

D不对，因为&lt;servlet-mapping&gt;里少了&lt;servlet-name&gt;子元素。

 E. <servlet-mapping>

```
<servlet-name>action</servlet-name>
</servlet-mapping>
```

11

可以为哪些类型的Web应用组件定义依赖性？（选出所有正确的答案）

(servlet 规范9.7.1)

 A. JSP文件 B. WAR文件 C. 类

库依赖性可以在/META-INF/MANIFEST.MF文件中定义。

 D. 库 E. 清单文件

2

以下哪些是JSP文档（基于XML的文档）中的合法声明？

(JSP v2.0 1~139页)

(选出所有正确的答案)

A. <jsp:declaration>

```
    xmlns:jsp="http://java.sun.com/JSP/Page">
    int x = 0;
    </jsp:declaration>
```

B. <jsp:declaration>

```
    xmlns:jsp="http://java.sun.com/JSP/Page">
    int x;
    </jsp:declaration>
```

C. <%! int x = 0; %>

C和D不对，因为JSP文档中只有

D. <%! int x; %>

<jsp:declaration>语法才正确。

3

哪些部署描述文件元素可以出现在<web-app>元素之前？

(Servlet 规范 107页)

(选出所有正确的答案)

A. <listener>

<web-app>元素是Web应用部署描述文件的根元素。

B. <context-param>

C. <servlet>

D. <web-app>元素的前面不能出现任何XML元素。

14

关于容器类加载器，哪些说法是正确的？

(Servlet 规范 9.7.2)

(选出所有正确的答案)

A. Web应用不能覆盖容器实现类。

B不对，因为web应用可以使用Web应用类加载器的getResource方法访问任何WAR文件。

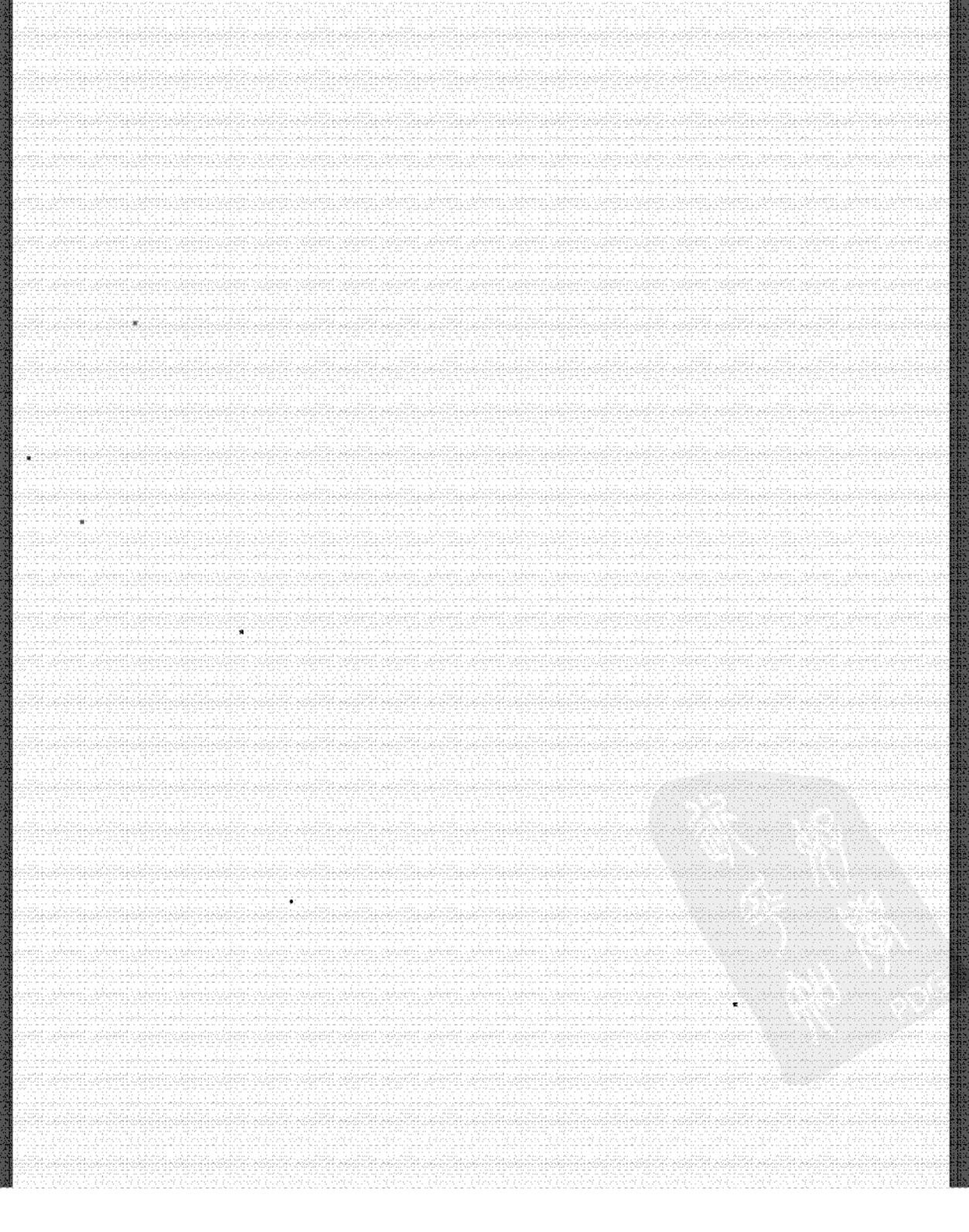
B. Web应用不能使用J2SE的getResource方法加载WAR文件中的资源。

C. Web应用可以覆盖 javax.\* 命名空间中的任何J2EE类。

D. Web开发人员可以覆盖J2EE平台类，条件是它们包含在WAR中的一个库

JAR中。

C和D不对，因为web应用不应覆盖java.\*或javax.\*命名空间中的任何类。



# 要保密，要安全

他们都在哪儿……那帮坏蛋……到处都是他们的人！我得学习认证和授权……我还得学习怎么安全地传输数据……喂……喂……听得见吗？



你的Web应用危险重重。网络的每个角落都潜伏着危险，黑客、捣乱的家伙，甚至犯罪分子会竭尽全力侵入你的系统，窃取你的秘密、利用你的信息，或者只是和你的网站开个玩笑。你不希望这些坏家伙监听网上商店的交易，不希望他们窃取信用卡号。你不希望这些坏蛋骗你的服务器，说他们就是那些拿大折扣的大客户。另外你也不希望有人（不管是好人还是坏人）偷看机密的员工数据。市场部的Jim有必要知道工程部的Lisa拿的薪水是他的3倍吗？你真的愿意Jim自己动手，非法地登录UpdatePayroll servlet吗？

# OBJECTIVES

## Web应用安全

- 5.1 根据servlet规范，对照比较以下安全问题：  
(a) 认证, (b) 授权, (c) 数据完整性和(d) 机密性。
- 5.2 在部署描述文件中声明以下内容：安全约束、Web资源、传输保证、登录配置和安全角色。
- 5.3 给定认证类型（BASIC, DIGEST, FORM和CLIENT-CERT），描述各种认证的机制。

### 内容说明：

这一部分的所有要求都将在这一章全面介绍，其中包括关于“部署”的一章未谈到的与安全相关的DD元素。

看完这一章并不会使你成为全面的安全专家，但是这一章的内容可以作为起点，而且考试只要求掌握这么多。

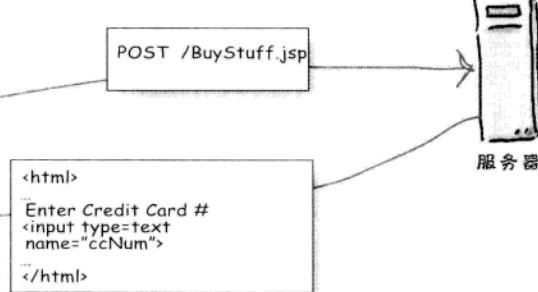
# 坏人无处不在

作为一个Web应用开发人员，你要保护好你的网站。得当心3种坏人：

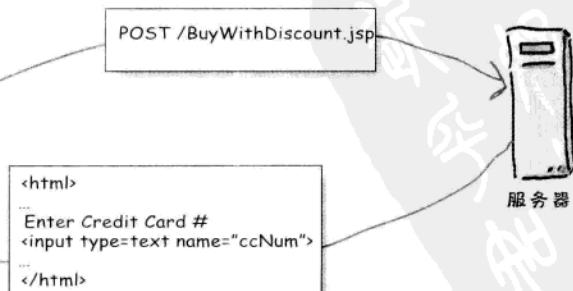
假冒者（Impersonator）、非法升级者（Upgrader）和窃听者（Eavesdropper）。



居心不良的假冒者

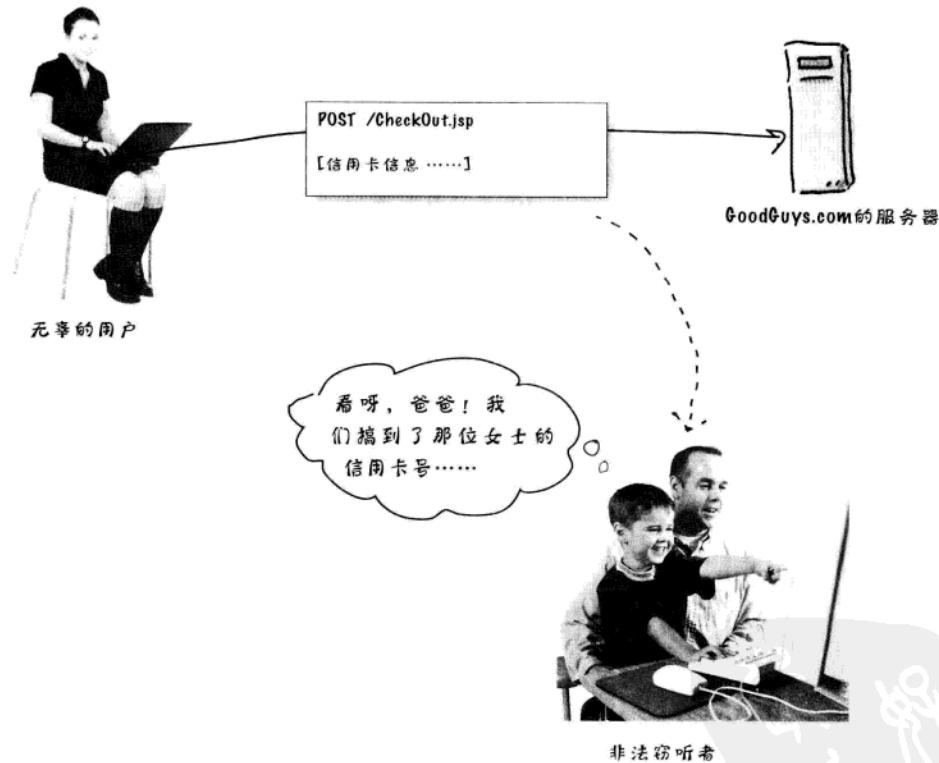


调皮捣蛋的升级者



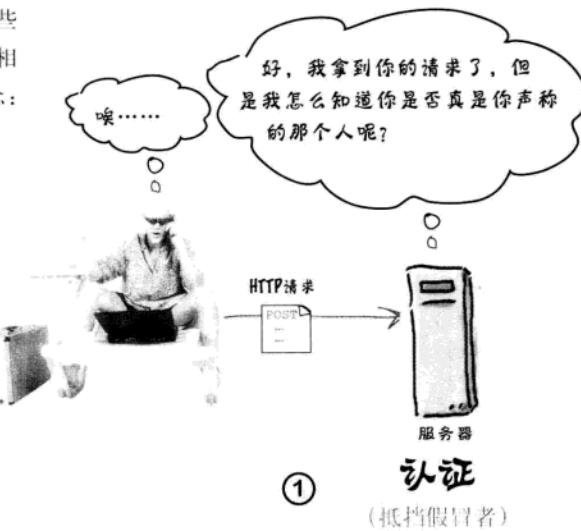
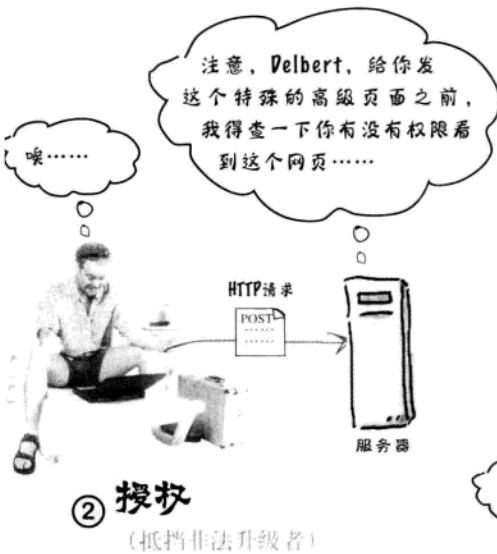
## 受害的不只是服务器……

窃听者可能最坏。他们不光想刺探你的Web应用，还可能会伤害你的一些优秀客户。这可是个大麻烦。如果窃听者成功地实施了攻击，他可能会偷走客户的信用卡号，然后疯狂消费。



# servlet安全的4大要素

servlet安全能帮助你（Web应用开发人员）抵挡这些假冒者、非法升级者和窃听者。就servlet规范来说（相应地，对考试来说），servlet安全可以划分为4大概念：**认证、授权、机密性和数据完整性**。



## 一个安全小故事

有一天，Bob的老板把Bob叫到他的办公室。“我要交给你一个有意思的新项目！”老板说。Bob只是轻轻地嗯了一声，没有太积极的响应。“我知道，以前交给你的几个项目不太好，不过这一次不同了，确实很有趣……我希望你为我们公司的新电子商务网站做安全设计。”“安全？”Bob回答道，“安全很难做，而且很麻烦。”“不，不，这你可说错了……”老板说。“在J2EE 1.4中，servlet安全相当酷。”

老板继续说，“我来给你一点提示，等你想清楚了，我们再来谈具体的。”“那好吧，”Bob只好点头。“我来试试。”

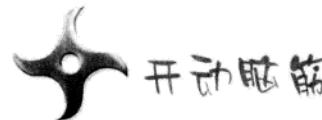
“你知道的，这个啤酒网站现在相当热门。我们增加了很多新的特性，而且反映不错。有些用户很喜欢我们提供的免费酒水，但更多的人（出乎我们的意料）想购买稀有啤酒和其他高级配料。哦，我们的Frequent Brewer程序要有一个特色。如果一个用户认为他会再次购买某种配料，可以先付一次钱，升级到Brew Master状态，Brew Master可以得到特殊折扣，而且可以挣得Frequent Brewer点数，凭点数还能换取啤酒奖励。”

Bob继续听着，心里盘算着实现这些要写多少代码，郁闷地想原来计划的热带之旅就要泡汤了。老板还在说……

“不过，现在我们必须保证，一个用户购买时，其他人不能偷看到他的信用卡信息。哦，还有一件事，最好还要保证一点，一个成员登录的时候，必须确确实实是他自己，而不是他的哪个朋友冒名进来。我想，从现在开始，成员应该有口令才行。”

“这些都很好。”Bob说。“不过，用户下了一个单时，我们要给他某种确认码吗？”“好主意”老板说，“对了，还有一件事我差点忘了，你最保证只有Frequent Brewer才能得到特殊折扣。”

“我想这应该就够了”，老板说。“不过你知道…这样运作下去，不用多久，肯定就要有白金会了……”



这个故事里提到了哪些安全概念？

再看一遍这个故事，把老板的以下安全需求标出来：

- 认证
- 授权
- 机密性
- 数据完整性

（没错，我们知道这些很明显，但是在真正深入之前，还是先来热热身。）

## 一个安全小故事

有一天，Bob的老板把Bob叫到他的办公室。“我要交给你一个有意思的新项目！”，老板说。Bob只是轻轻地嗯了一声，没有太积极的响应。“我知道，以前交给你的几个项目不太好，不过这一次不同了，确实很有趣……我希望你为我们公司的新电子商务网站做安全设计。”“安全？”Bob回答道，“安全很难做，而且很麻烦。”“不，不，这你可说错了……”老板说。“在J2EE 1.4中，servlet安全相当酷。”

老板继续说，“我来给你一点提示，等你想清楚了，我们再来谈具体的。”“那好吧，”Bob只好点头。“我来试试。”

“你知道的，这个啤酒网站现在相当热门。我们增加了很多新的特性，而且反映不错。有些用户很喜欢我们提供的免费酒水，但更多的人（出乎我们的意料）想购买稀有啤酒和其他高级配料。哦，我们的Frequent Brewer程序要有一个特色。如果一个用户认为他会再次购买某种配料，可以先付一次钱，升级到Brew Master状态，Brew Master可以得到特殊折扣，而且可以挣得Frequent Brewer点数，凭点数还能换取啤酒奖励。”

Bob继续听着，心里盘算着实现这些要写多少代码，郁闷地想原来计划的热带之旅就要泡汤了。老板还在说……

“不过，现在我们必须保证，一个用户购买时，其他人不能偷看到他的信用卡信息。哦，还有一件事，最好还要保证一点，一个成员登录的时候，必须确确实实是他自己，而不是他的哪个朋友冒名进来。我想，从现在开始，成员应该有口令才行。”

“这些都很好。”Bob说。“不过，用户下了一个订单时，我们要给他某种确认码吗？”“好主意”老板说，“对了，还有一件事我差点忘了，你最好保证只有Frequent Brewer才能得到特殊折扣。”

“我想这应该就够了”，老板说。“不过你知道……这样运作下去，不用多久，肯定就要有白金会员了……”

机密性和数据完整性——此时服务器正在返回重要的私密信息。如果这个信息被窃听者看到或篡改，那就太糟糕了。

授权——一旦确定对话人的身份，我们想确保他们有权做想做的事情。

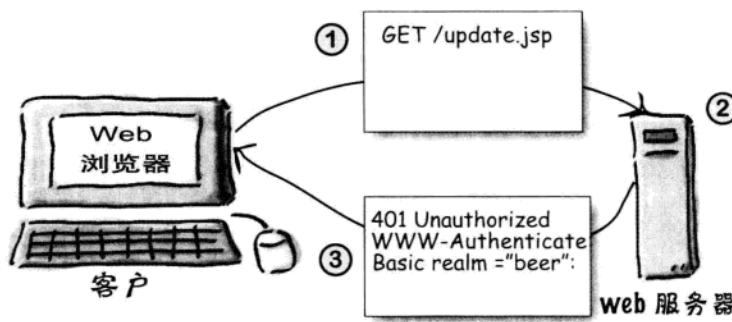
机密性——如果用户的信用卡号落到别人手里，这可是一个严重的安全问题！

认证——只要有人提到口令，他说的可能就是认证……用户真的是他声称的那个人吗？如果是，他应该知道口令才对！

# HTTP世界中如何认证： 安全交易的开始

先来看一下，客户请求网站上的一个安全资源时，浏览器和Web容器之间怎样通信。这确实很基本（BASIC）。

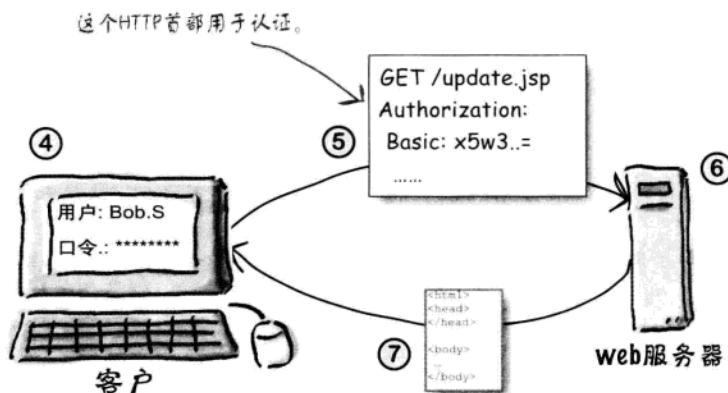
## HTTP观点……



**1** 浏览器向一个Web资源（“update.jsp”）发出请求。

**2** 服务器确定这个“update.jsp”是一个受限资源。

**3** 容器发回一个HTTP 401（“Unauthorized”，未授权），包含一个www-authenticate首部和realm（领域）信息。



**4** 浏览器得到401，根据Realm信息要求用户提供用户名和口令。

**5** 浏览器再次请求“update.jsp”（要记住，通信是无状态的），但是这一次，请求还包括一个安全HTTP首部以及用户名和口令。

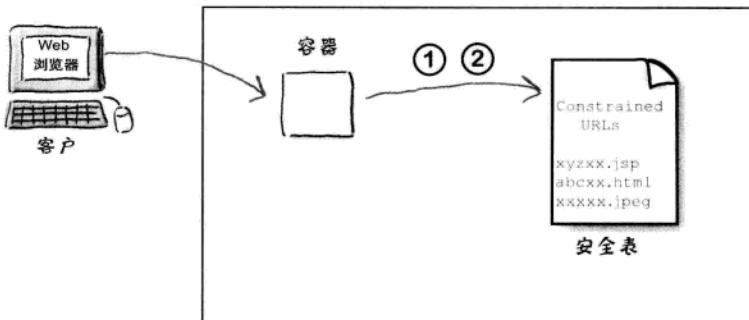
**6** 容器验证用户名和口令是否匹配。如果匹配，则完成授权。

**7** 如果所有安全信息都是对的，容器会返回HTML，否则，再返回一个HTTP 401……

# 再来仔细看看容器如何完成认证和授权

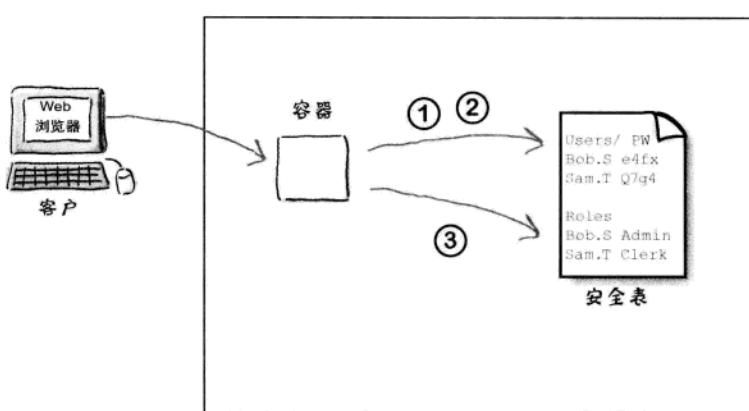
上一页没有具体说明容器做了什么。在这一章中，我们会从不同层次、不同的详细程度来介绍，现在就要开始有点细化了……

## 容器观点……



**1** 接收到请求时，容器在“安全表”中查找URL（安全表存储在容器中，用于保存安全信息）。

**2** 如果容器在安全表中找到URL，会查看所请求的资源是否是受限的。如果是，则返回401……



**1** 容器接收到有用户名和口令的请求时，在安全表中查找URL。

**2** 如果在安全表中找到URL（而且发现这是一个受限资源），则检查用户名和口令，确保它们匹配。

**3** 如果用户名和口令确实匹配，容器会查看为这个用户指派的“角色”是否允许访问这个资源（即授权）。如果可以，则把资源返回给客户。

# 容器是怎么做的？

前面对容器如何处理认证和授权有了一个大致的认识。那么容器为此到底做了些什么？下面来推测一下，在后台，在容器内部发生了什么……

容器做的事情：

## ① 查找所请求的资源

我们已经知道，容器非常擅长查找资源。不过，一旦它找到资源，还必须确定这是一个任何人都能看的资源还是一个有安全约束（限制）的资源。servlet本身有某种安全标志吗？有没有一个相应的安全表？

## ② 完成某种认证

一旦容器确定处理的是一个受保护的资源，就必须对客户进行认证。换句话说，要明确“Bob”是否真的是Bob。（最常用的方法是看Bob知不知道他自己的口令。）

## ③ 完成某种授权

一旦容器确定请求资源的人确实是真的Bob，容器还必须查看Bob是否有权访问这个资源。来看看，如果我们有2,000,000个用户，而Web中有100个servlet，放在一起这个表就要有200,000,000个单元格……

哇！稍不留心就会失控。

安全很费功夫，为此要花费很多时间！如果  
你能更高效地实现安全，肯定对性能大有帮助。



服务器



哪些安全逻辑和信息应该硬编码写在servlet里？

用户名和口令？

用户角色？

每个servlet的访问规则？

# 码里不要有安全信息!

大多数Web应用，大多数情况下，Web应用的安全约束都应声明方式处理，也就是要在部署描述文件中指定，为什么

安全约束就是限制的意思

## 以声明方式处理安全的十大原因

- ⑩ 谁不需要更多的XML练习呢？
- ⑨ 通常能自然地映射到公司IT部门现有的任务角色。
- ⑧ 让你的简历更出色。
- ⑦ 你可以用更灵活的方式使用以前写的servlet。
- ⑥ 考试中会考到。
- ⑤ 允许应用开发人员重用servlet，而不用去纠缠源代码。
- ④ 这样真的很酷。
- ③ 随着应用的扩展，可以减少可能的维护。
- ② 还有一点，这正好能体现容器的价值……
- ① 支持基于组件的开发思想。

## 谁来实现Web应用中的安全？

我的任务很容易。大多数情况下，我编写servlet时甚至不用考虑安全。这很好，因为我的哲学就是“安全很难……不要实现安全。”



Kim

servlet 提供者

我的任务稍微复杂一些。我要确定应用中有哪些角色。对于Kim啤酒应用，重要角色是Guest、Member和Admin。我再为容器用户文件中的用户增加这些角色。因为我们使用的是tomcat，所以这个用户文件叫做tomcat-users.xml。



Annie

应用管理员

我的任务最艰巨！我拿到Annie的角色表，得到Kim的servlet的有关描述，确定哪些角色可以访问哪些servlet。通过部署描述文件，我能很容易地告诉容器谁能访问哪些servlet（只是稍有些罗嗦），另外，还得告诉你他们给我的薪水太低了……



Dick

部署人员

# there are no Dumb Questions

**问：** 我被搞糊涂了——既然是我在创建servlet，为什么不该由我来考虑安全问题呢？

**答：** 不错，你是该考虑安全问题：这个servlet提供者（Kim）有点“讽刺”的意思。设计servlet时的关键是servlet的模块性。例如，把浏览功能与更新功能分开就很有意义。如果这两个用例在不同的servlet中实现，部署人员就能很容易地为这两个servlet分别指定不同的安全约束。

**问：** 我不知道你在哪里高就，但是对我来说，我要身兼数职：开发人员、管理员和部署人员。

**答：** 这是一个非常常见的情况。不过，我们还是建议你分阶段实现安全，每次“假想”自己只是一种身份。

**问：** 怎么通过程序实现安全呢？

**答：** 这一章后面会谈到通过程序实现安全（程序式安全）。对现在来说，重要的是要知道servlet中所做的95%的安全工作都是通过声明方式实现的。程序式安全用得很少（见“十大原因”……）。

**问：** 到目前为止，你说的所有内容都与认证和授权有关，那么“4大要素”中的另外两项呢？

**答：** 这一章后面会谈到机密性和数据完整性。根据servlet规范，实现这两个概念相当容易，所以我们把重点放在认证和授权上，因为这两个概念理解和实现起来最复杂，而且还有一个重要提示：认证和授权在考试时很可能考。

**问：** 谈到servlet安全时，好像“角色”这个词有多重含义……

**答：** 这个问题问得好！Sun设计J2EE规范（EJB、servlet、JSP）时，是从可能创建和管理这些组件的人的角度来考虑的。换句话说，就是与IT相关的任务角色。开发人员处理Web应用的安全问题时，他们考虑的是可能存在哪些类型的用户。例如，“guest”（游客）在Web应用中权限最少，“member”（会员）权限可能多一些。这些“用户角色”要在部署描述文件中定义、映射和建立。

**问：** 我听说有一种“跨网站”攻击。那是怎么回事？

**答：** 如果一个网站会显示其他用户输入的任意表单文本（例如，用户图书评论），就可能发生“跨网站”（cross-site）攻击。如果有恶意用户在一个文本域中键入一些HTML（如Javascript），服务器未能发现，毫不设防的浏览器在提供页面时不仅会显示合法的HTML，还会执行可能有危险的隐藏代码。换句话说，服务器向用户发送了另一个用户键入的一些东西，而未对它做恶意脚本代码检查或处理。

**问：** 我们已经了解了“4大要素”。这些安全方面设置和维护的难度有多大，我是说，会不会很麻烦？

**答：** 是的，恐怕是有点麻烦。确实，安全的某些方面开销很小，但另外一些方面确实要做很多工作。不过这些都不算复杂，只是有点烦。

## servlet安全中的重要任务

下表会让你对servlet安全中的重要方面有所认识。授权的实现最耗费时间，认证次之。从servlet观点看，机密性和数据完整性很容易建立（注1）。

安全概念	谁负责？	复杂程度	耗时程度	对考试的重要程度
认证	管理员	中	高	中
授权	部署人员 (大多数情况下)	高	高	高
机密性	部署人员	低	低	低
数据完整性	部署人员	低	低	低

这一章我们会强调授权，因为在与具体开发商无关的安全概念中，这是最重要也最复杂的一个概念。

注1：实际上，得到SSL证书并不简单，所以所谓的“容易”是指“你不用在你的servlet代码中做任何工作。”

# 认证就谈到这里，下面讨论授权

本章后面还会更深入地讨论认证，不过，对现在来说，我们认为系统中已经有了足够的认证数据，可以重点讨论授权了。如果用户没有经过认证，就无法得到授权。

servlet规范没有指出容器应该如何实现对认证数据（包括用户名和口令）的支持。但是一般做法是，容器会提供开发商特定的一个表，其中包含用户名和相关的口令和角色。但是，几乎所有开发商都并没有就此止步，还会提供某种方法来维护你的公司特定的认证数据，通常存储在一个关系数据库或LDAP系统中（这超出了本书的范围）。一般地，这些数据由管理员来维护。

## 安全“领域”

遗憾的是，安全世界中“领域”（realm）一词也有多重含义。就servlet规范来说，领域就是存储认证信息的地方。在Tomcat中测试你的应用时，可以使用一个名为“tomcat-users.xml”的文件（位于Tomcat的conf目录下，而不在Web应用中）。这个“tomcat-users.xml”文件应用于web-apps下部署的所有应用。通常称之为内存领域（memory realm），因为Tomcat会在启动时将这个文件读入内存。尽管这对于测试来说很棒，但是生产阶段不建议这样做。其中一个原因是，如果不重启Tomcat就无法修改领域的内容。

### tomcat-users.xml文件

```
<tomcat-users>
    <role rolename="Guest"/>
    <role rolename="Member"/>
    <user username="Bill" password="coder" roles="Member, Guest" />
    ...
</tomcat-users>
```

你的应用服务器的具体做法可能和这里不一样……不过总会以某种方式把用户映射到口令和角色。



对认证的控制就放在这样一种数据结构中。在Tomcat中，可以使用一个名为“tomcat-users.xml”的XML文件，其中包括一些用户名-口令-角色设置，容器在认证时会使用这些设置。

记住！这不是DD的一部分：  
这特定于具体的开发商。

## 启用认证

如果要进行认证（换句话说，要让容器询问用户名和口令），你需要在DD中放点什么。现在先不用担心这是什么意思，不过如果你想尝试一下认证，可以先用下面的配置：

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

这个内容本章后面还会谈到，不过需要知道，你的DD中要有这个配置才能进行认证。

## 授权第1步：定义角色

servlet中授权最常用的形式是：由容器确定一个特定的servlet（和HTTP请求方法）能否由已经指派了某个安全“角色”的某个用户调用。所以，第一步就是把开发商特定“用户”文件中的角色映射到部署描述文件中建立的角色。

Annie是一个“Admin”、“Member”和“Guest”。



开发商特定：

tomcat-users.xml中的<role>元素

```
<tomcat-users>
    <role rolename="Admin"/>
    <role rolename="Member"/>
    <role rolename="Guest"/>
    <user username="Annie" password="admin" roles="Admin, Member, Guest" />
    <user username="Diane" password="coder" roles="Member, Guest" />
    <user username="Ted" password="newbie" roles="Guest" />
</tomcat-users>
```

开发商特定的用户  
和角色数据结构。

Diane是一个“Member”和“Guest”  
Ted是一个“Guest”

在Tomcat中，tomcat-users.xml大致如此。  
注意一个用户可能有多个角色。

SERVLET规范：

web.xml中的DD <security-role>元素

```
<security-role><role-name>Admin</role-name></security-role>
<security-role><role-name>Member</role-name></security-role>
<security-role><role-name>Guest</role-name></security-role>
```

授权时，容器会把开发商特定的“角色”信息映射到在DD <security-role>元素中找到的<role-name>。

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

别忘了，如果你想启用认证，必须有<login-config>元素。

部署人员在DD中创建<role-name>元素，以便容器将角色映射到用户。

## 授权第2步：定义资源/方法约束

最后一步，也是最酷的一步。在这一步中，我们要以声明方式指定一个给定的资源/方法组合，只能由特定角色的用户访问。你要做的大多数安全工作都可能与DD中的`<security-constraint>`元素有关（后面还会介绍许多严格的规则）。

### DD中的`<security-constraint>`元素：

```

<web-app...>
...
<security-constraint>          这个名是必要的，由工具使用。
<web-resource-collection>       在别的地方不会看到这个名.....      ↴
    <web-resource-name>UpdateRecipes</web-resource-name>
    <url-pattern>/Beer/AddRecipe/*</url-pattern>      ← <url-pattern>元素定义要约束的资源。
    <url-pattern>/Beer/ReviewRecipe/*</url-pattern>

    <http-method>GET</http-method>           ↴ <http-method>元素描述了对于URL模
    <http-method>POST</http-method>           式指定的资源哪些HTTP方法是受约束的
    </web-resource-collection>             (受限的)。

    <auth-constraint>                     ← 可选的<auth-constraint>元素列出了哪些角
        <role-name>Admin</role-name>          色可以调用受约束的HTTP方法。换句话说，
        <role-name>Member</role-name>         它指出了谁能在特定URL模式所指定资源上
    </auth-constraint>                   调用GET和POST方法。
    </security-constraint>

</web-app>

```



因为 Diane 和 Annie 都有“Member”角色，她们可以在与`<url-pattern>`元素匹配的资源上调用 GET 和 POST。Ted 只是一个“Guest”，所以他不能完成 GET 或 POST。

# <web-resource-collection>元素的<security-constraint>规则

记住：<web-resource-collection>子元素的作用是告诉容器哪些资源和HTTP方法组合要以某种方式受约束，即只能由相应<auth-constraint>标记中的角色访问。希望我们没有让你紧张，不过你确实要知道这些元素的细节。就算只是在DD的安全部分犯一个小小的错误，也可能让应用中最机密的部分暴露给……每一个人。

## <security-constraint>的<web-resource-collection>子元素

```

<web-app...>
...
<security-constraint>

    <web-resource-collection>
        <web-resource-name>
            UpdateRecipes
        </web-resource-name>
        这些是有约束的目录。
        ↓
        <url-pattern>/Beer/AddRecipe/*</url-pattern>
        <url-pattern>/Beer/ReviewRecipe/*</url-pattern>

        <http-method>GET</http-method>
        ←
    </web-resource-collection>

    <auth-constraint>
        ...
    </auth-constraint>

</security-constraint>
</web-app>

```

这里指出只有<auth-constraint>中定义的角色可以访问GET方法。

但是其他方法是没有约束的，所以任何角色都可以访问这些方法。

### 关于

- ▶ <web-resource-collection>的要点 —
- ▶ <web-resource-collection>元素有两个主要的子元素：<url-pattern>（一个或多个）和<http-method>（可选，0或多个）。
- ▶ URL模式和HTTP方法共同定义受的资源请求，这些资源只能由<auth-constraint>中定义的角色访问。
- ▶ <web-resource-name>元素是必要的（尽管你自己可能不会用到它）。（可以认为它要由IDE使用，或将来可使用）。
- ▶ <description>元素是可选的。
- ▶ <url-pattern>元素使用servlet标准命名映射规则（有关URL模式的详细内容以返回去看关于“部署”那一章）。
- ▶ 必须至少指定一个<url-pattern>，不也可以有多个<url-pattern>。
- ▶ <http-method>元素的合法方法包括GET, POST, PUT, TRACE, DELETE和OPTIONS。
- ▶ 如果没有指定任何HTTP方法，那么所有方法都是受约束的（这表示，它只能由<auth-constraint>中定义的角色访问）！！
- ▶ 如果确实指定了<http-method>，么只有所指定的方法是受约束的。换句话说，一旦指定了一个<http-method>，就会使未指定的HTTP方法自动启用（即不受约束）。
- ▶ 一个<security-constraint>中可以有个<web-resource-collection>元素。
- ▶ <auth-constraint>元素应用于<security-constraint>中的所有<web-resource-collection>元素。



不是在资源层次上建立约束，约束建立在HTTP请求层次上。

很容易认为是资源本身受到约束，但实际上应该是资源+HTTP方法组合受到约束。如果说“这是一个受限资源”，实际上应该说是“对于HTTP GET来说，这是一个受限资源”。资源总是会针对某个HTTP方法受约束（按HTTP方法来指定），不过可以通过配置`<web-resource-collection>`使得所有方法都受约束，为此，在`<web-resource-collection>`中不要放置任何`<http-method>`元素。

`<auth-constraint>`元素并不是定义哪些角色可以访问`<web-resource-collection>`中的资源。相反，它只是定义了哪些角色可以做出受约束的请求。不要这样想：“Bob是一个Member，所以Bob可以访问AddRecipe servlet”。而应该是“Bob是一个Member，所以Bob可以对AddRecipe servlet做一个GET或POST请求。”



如果指定一个`<http-method>`元素，所有未指定的HTTP方法都不受约束！

Web服务器的任务是提供服务，所以默认的假设是希望HTTP方法不受约束，除非已经明确指出（使用`<http-method>`）你希望某个方法受约束（针对与`<url-pattern>`匹配的资源）。如果在安全约束中只放了一个`<http-method>GET</http-method>`，那么 POST, TRACE, PUT等等都不受约束！这说明，任何人都能调用这些HTTP方法，而不论安全角色是什么，也不论客户是否经过认证。

不过……这里有一个前提，只有至少指定了一个`<http-method>`元素时才是如此。如果没有指定任何`<http-element>`，就会对所有HTTP方法都施加约束（你可能永远也不会这样做，因为安全约束的意义就是针对一组特定的资源限制某些特定的HTTP请求）。

当然，除非你覆盖了相应的doXXX()方法，否则在servlet中这些HTTP方法将无法工作，所以，如果servlet中只有一个doGet()，而你只为GET指定了一个`<http-method>`元素，那么谁都无法调用POST方法，因为服务器知道你并不支持POST方法。

所以，这个规则要稍做修改：除非指定了以下两点之一，否则servlet支持的所有HTTP方法都不受约束（你已经覆盖了相应的服务方法）：

- 1) `<security-constraint>`中未指定任何`<http-method>`元素，这说明所有方法对于`<auth-constraint>`中的角色都是受约束的。
- 2) 使用`<http-method>`元素明确地列出了方法。

要记住，一旦安全约束中有哪怕一个`<http-method>`，所支持的所有其他HTTP方法都将不受约束。

## <security-constraint>中 <auth-constraint>子元素的规则

虽然名字里有“约束”（constraint）这个词，但实际上这个子元素指定的是哪些角色允许访问<web-resource-collection>子元素指定的Web资源。

### <security-constraint>的<auth-constraint>子元素

```
<web-app.....>
...
<security-constraint>
    <web-resource-collection>
        ...
    </web-resource-collection>
    <auth-constraint>
        <role-name>Admin</role-name>
        <role-name>Member</role-name>
    </auth-constraint>
</security-constraint>
</web-app>
```



这说明Admin和Member都能访问<web-resource-collection>中定义的资源 / HTTP 方法组合。它没有提到“Guest”，所以“Guest”不能完成受约束的请求。

#### <role-name> 规则

- ▶ 在<auth-constraint>元素中，<role-name>元素是可选的。
- ▶ 如果存在<role-name>元素，它们会告诉容器哪些角色得到许可。
- ▶ 如果存在一个<auth-constraint>元素，但是没有任何<role-name>元素，那么所有用户都遭拒绝。
- ▶ 如果有<role-name>\*</role-name>，那么所有用户都是允许的。
- ▶ 角色名区分大小写。

#### <auth-constraint>规则

- ▶ 在<security-constraint>元素中，<auth-constraint>元素是可选的。
- ▶ 如果存在一个<auth-constraint>，容器必须对相关的URL完成认证。
- ▶ 如果不存在<auth-constraint>，容器允许不经认证就能访问这些URL。
- ▶ 为了提高可读性，可以在<auth-constraint>中增加一个<description>。

# <auth-constraint>怎样做

<auth-constraint>的内容

哪些角色能访问



```
<security-constraint>
```

```
  <auth-constraint>
    <role-name>Admin</role-name>
    <role-name>Member</role-name>
  </auth-constraint>
```

```
</security-constraint>
```

```
<security-constraint>
```

```
  <auth-constraint>
    <role-name>Guest</role-name>
  </auth-constraint>
```

```
</security-constraint>
```

```
<security-constraint>
```

```
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
```

```
</security-constraint>
```

**Admin**  
**Member**



**Guest**



**所有人**



如果没有  
<auth-constraint>

这两个的效果一样。

**所有人**



```
<security-constraint>
```

```
  <auth-constraint/>
```

```
</security-constraint>
```

唉呀！如果放入一个空标记，  
那么任何角色都不能访问。

**没有人**



没有<auth-constraint>与空<auth-constraint/>的作用刚好相反！

要记住：如果没有说哪些角色受约束，则任何角色都不受约束。但是，一旦放入一个<auth-constraint>，那么只有明确指定的角色允许访问（除非<role-name>使用了通配符“\*”）。如果希望任何角色都不能访问，就必须放入<auth-constraint/>，但是它必须为空。这就会告诉容器，“我明确地指出了哪些角色是允许的，而且实际上，这样的角色一个也没有！”



## 多个<security-constraint>元素如何交互

你可能觉得已经把<security-constraint>搞清楚了，此时你会认识到：多个<security-constraint>元素可能会冲突。请看下面的DD片段，想象一下可能会使用哪些<auth-constraint>配置组合。例如，对于同一个受限资源和相同的角色，如果一个<security-constraint>不允许访问，而另一个<security-constraint>明确地授予访问权……会怎么样呢？哪个<security-constraint>会赢？所有答案都在下一页的表中。

URL模式和<http-method>元素相同（或部分匹配）的<security-constraint>元素：

```
<web-app...>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Recipes
      </web-resource-name>
      <url-pattern>/Beer/DisplayRecipes/*
      </url-pattern>
      <url-pattern>/Beer/UpdateRecipes/*
      </url-pattern>
      <http-method>POST</http-method>
    </web-resource-collection>
```

(A)

```
</security-constraint>
```

这两个<security-constraint>元素都指定了定义在“/Beer/UpdateRecipes/\*”中的资源。

有不同角色名的<auth-constraint>元素

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Update
    </web-resource-name>
    <url-pattern>/Beer/UpdateRecipes/*
    </url-pattern>
    <url-pattern>/Beer/UpdateUsers/*
    </url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
```

(B)

```
</security-constraint>
```

```
</web-app>
```

如果一个资源在多个<security-constraint>中用到，容器怎样处理授权？

# <auth-constraint>元素决战

如果两个或多个<security-constraint>元素有部分或完全重叠的<web-resource-collection>元素，容器会如下处理对重叠资源的访问。A和B是指前一页上的DD片段。

(A) 的内容	(B) 的内容	谁能访问 ‘UpdateRecipes’
1 <auth-constraint> <role-name>Guest</role-name> </auth-constraint>	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	Guest和Admin   
2 <auth-constraint> <role-name>Guest</role-name> </auth-constraint>	<auth-constraint> <role-name>*</role-name> </auth-constraint>	所有人   
3 <auth-constraint/> 空标记	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	没有人   
4 没有<auth-constraint>元素	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	所有人   

这个表的解释规则：

1 合并单个的角色名时，所列的所有角色名都允许访问。

2 角色名“\*”与其他设置合并时，所有人都允许访问。

3 空的<auth-constraint>标记与其他设置合并时，所有人都不允许访问！换句话说，空的<auth-constraint>就是最后“宣判”！

4 如果某个<security-constraint>元素没有<auth-constraint>元素，它与其他设置合并时，所有人都允许访问。

如果两个不同的非空  
<auth-constraint>元  
素应用于同一个受限资  
源，那么两个<auth-  
constraint>元素中所有  
角色的并集都允许访问。

*there are no  
Dumb Questions*

**问：** 放一个空的`<auth-constraint/>`元素会告诉容器：任何角色的任何人都不能访问受限资源，这我能理解。但是，我不能理解你为什么要这样做。如果一个资源谁也不能访问，那它还有什么意义？

**答：** 我们说“没有人能访问”是指“Web应用之外的任何人都不允许访问”。换句话说，客户不能访问受限资源，但是Web应用的其他部分是可以的。你可能想使用一个请求分派器转发到Web应用的另一部分，但是你不希望客户直接请求这个资源。可以把受限资源想成是Java类中的某种私有方法，仅供内部使用。

**问：** 为什么`<auth-constraint>`元素放在`<security-constraint>`里，而不是`<web-resource-collection>`元素里？

**答：** 这样一来就可以指定一个`<auth-constraint>`元素（它能包括多个角色），然后指定该`<auth-constraint>`角色列表适用的多个资源集合。例如，可以为一个Frequent Buyer角色定义一个`<auth-constraint>`，然后为Web应用中Frequent Buyer能访问的所有不同部分建立多个`<web-resource-collection>`元素。

**问：** 我非得老老实实坐在这里敲入每个用户的口令和角色吗？

**答：** 如果你使用Tomcat的测试内存领域，确实别无选择。不过，在实际中，你很可能使用了一个生产服务器，也许允许你连接存放实际用户安全信息的LDAP或数据库。

内存领域指容器会读取`tomcat-users.xml`加载到内存

# Alice的recipe servlet，关于程序式安全……

Alice知道，大多数情况下都应该使用声明式安全。这种方法很灵活、很强大、可移植，而且很健壮。随着Web应用结构的发展，单个servlet变得越来越特定。原先可能用一个servlet提供业务逻辑来同时支持员工和经理。如今，这些功能可能至少分到两个不同的servlet中。

不过，Alice很幸运，她得到了别人写的一个“RecipeServlet”。Alice听说RecipeServlet使用的是程序式安全，所以她开始查看源代码，并且发现下面的代码段……

```
if( request.isUserInRole("Manager") ) {
    // 处理UpdateRecipe页面
    ...
} else {
    // 处理ViewRecipe页面
    ...
}
```

谁提出“Manager”作为一个角色名？如果写这个servlet的人不了解你公司的角色安排呢？



## 有什么含义？

考虑一下你在这一章学到的知识，看看上面这个很短的代码段，回答以下问题。

这个代码段运行前必须完成哪  
个安全步骤？

这个代码段指示哪一个安全  
步骤？？

在这个代码段中DD起什么作  
用（如果有的话）？

你认为这个代码段如何工作？

如果你的容器中不存在角色  
“Manager”会怎么样？



## 定制方法：`isUserInRole()`

在HttpServletRequest中，有3个方法与程序式安全有关：

getUserPrincipal()，这个方法主要用于EJB。这本书不做介绍  
(注2)。

getRemoteUser()，可以用于检查认证状态。这个方法很少使用，所以这本书不做介绍（考试对这个方法也没有其他要求）。

isUserInRole()，我们现在就来看这个方法。可以在HTTP方法层次（GET、POST等）上完成授权，而是对方法中的某些部分建立访问授权（是否允许访问方法中的某一部分）。这样一来，你就能根据用户的角色定制服务方法的行为。如果正在处理服务方法（doGet()、doPost()等），而且用户已经完成声明式授权。但是现在你想在方法中根据条件做些事情，也就是根据用户的特定角色做不同的安排。该怎么做？

如何工作：

- ① 调用isUserInRole()前，用户要得到认证。如果对一个未经认证的用户调用这个方法，容器总会返回false。
- ② 容器得到isUserInRole()的参数，在这个例子中参数就是“Manager”，把它与请求中为此用户定义的角色进行比较。
- ③ 如果用户可以映射到这个角色，容器会返回true。



DD中的角色如何与servlet中的角色匹配？

注2：不过，确实有一本极好的  
EJB书值得推荐……

# 程序式安全也有声明的一面

程序员把安全角色名硬编码写在servlet中时（用作为isUserInRole()的参数），程序员很可能只是建立了一个“人造名”。他也许不知道实际的角色名，也可能在写一个可重用的组件，这个组件要由多个公司所用，而且这些公司不太可能刚好有这个程序员所用的角色名（当然，如果程序员确实想构建可重用的组件，那么硬编码写入角色名的做法实在很糟糕，不过现在我们先不讨论这个问题）。

可以看到，部署描述文件有一种机制可以把servlet中硬编码的（人造的）角色名映射到容器中“正式

servlet中

```
if( request.isUserInRole("Manager") ) {
    // 处理UpdateRecipe页面
    ...
}

} else {
    // 处理ViewRecipe页面
    ...
}
```

在这种情况下，如果`<security-role-ref>`不存在，就会失败，因为没有名为“Manager”的`<security-role>`。



就算是程序中的角色名确实与一个“实际”`<security-role>`名匹配，容器也会使用`<security-role-ref>`映射。

容器看到“`isUserInRole()`”的参数时，它会首先查找一个匹配的`<security-role-ref>`。如果找到，就会使用这个映射，而不管硬编码的角色名是不是正好与一个`<security-role>`名匹配。请考虑一下，你的公司也许确实有一个“Manager”安全角色，但是这个角色可能与程序员的本意相去甚远。所以，举例来说，你可以把硬编码的“Manager”映射到“Admin”，然后把一个硬编码的“Director”映射到“Manager”。所以，如果有相同的`<role-name>`，总是`<security-role-ref>`占上风。

的”`<security-role>`声明。例如，假设程序员使用“Manager”作为`isUserInRole()`的参数，但是你的公司使用了“Admin”作为`<security-role>`，而且你根本没有一个“Manager”安全角色。所以，就算你无法制止程序员硬编码写入角色名，起码应该知道，当硬编码的角色与你的实际角色名不匹配时用什么办法解决。因为，即使你确实拿到了servlet源代码，我只是想把所有“Manager”换成“Admin”，你真的想要为此修改、重编译，而且重新测试这个代码吗？

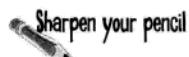
DD中

```
<web-app...>
    <servlet>
        <security-role-ref>
            <role-name>Manager</role-name>
            <role-link>Admin</role-link>
        </security-role-ref>
        ...
    </servlet>
    ...
    <security-role>
        <role-name>Admin</role-name>
    </security-role>
    ...
</web-app>
```

`<security-role-ref>`元素把程序中的（硬编码）角色名映射到声明的`<security-role>`元素。



## 安全练习



假设以下所有安全约束都有相同的<url-pattern>和<http-method>元素。根据所示的组合，确定谁能直接访问受限资源。

	没有人	Guest	Member	Admin	所有人
①					
②					
③					
④					
⑤					
⑥					

OK，我已经了解授权了，但是我还不清楚认证是怎么回事，我该怎么做才能让容器询问用户名和口令……



## 再来看认证

对于一个J2EE容器，认证可以总结为：询问一个用户名和口令，再验证它们是否匹配。

未经认证的用户第一次请求一个受限资源时，容器会自动开始认证过程。容器可以提供4种类型的认证，它们的主要区别是“所传输的用户名和口令信息有多安全？”

### 4种类型的认证

基本（BASIC）认证以一种编码形式（未加密）传输登录信息。听上去可能很安全，但是你可能已经知道了，由于编码机制（base64）已经广为人知，所以基本认证的安全性很弱。

摘要（DIGEST）认证以一种更安全的方式传输登录信息，但是由于加密机制没有得到广泛使用，并不要求J2EE容器一定要支持摘要认证。有关摘要认证的更多信息请查看IETF RFC 2617 ([www.ietf.org/rfc/rfc2617.txt](http://www.ietf.org/rfc/rfc2617.txt))。

客户证书（CLIENT-CERT）认证以一种非常安全的形式传输登录信息，它使用了公共密钥证书（Public Key Certificates, PKC）。这种机制的缺点是，你的客户必须先有一个证书才能登录你的系统。客户很少有证书，所以客户证书认证主要用于B2B应用。

前面的3种认证（基本认证、摘要认证和客户证书认证）都使用了浏览器的标准弹出表单来输入用户名和口令。但是第4种认证（即表单认证）与之不同。

表单（FORM）认证允许你利用合法的HTML建立自己的定制登录表单。但是……在这4种认证中，基于表单的信息会以最不安全的方式传输。用户名和口令都在HTTP请求中发回，而且未经加密。

## 实现认证

这一部分很简单——只是在DD中声明认证机制。关于认证，主要DD元素是`<login-config>`。

4个`<login-config>`例子：

```
<web-app...>
  ...
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
</web-app>
```

BASIC认证确实很基本，一旦在DD中声明了这个元素，余下的一切都将由容器负责，请求一个受限资源时，容器会自动地询问用户名和口令。

— 或 —

```
<web-app...>
  ...
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
</web-app>
```

如果你的容器支持摘要认证(DIGEST)，它会处理所有细节。

— 或 —

```
<web-app...>
  ...
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
</web-app>
```

客户端证书认证(CLIENT-CERT)很容易配置，但是你的客户必须要有证书。不过这种认证确实能提供最大强度的保护！

— 或 —

```
<web-app...>
  ...
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/loginPage.html</form-login-page>
    <form-error-page>/loginError.html</form-error-page>
  </form-login-config>
</login-config>
</web-app>
```

表单认证实现起来最复杂，将在  
下一页详细介绍

除了表单认证，一旦在DD中声明了`<login-config>`元素，实现认证的工作就完成了！

(假设已经在服务器中配置了用户名/口令/角色信息。)

# 基于表单的认证

与其他形式的认证比起来，尽管基于表单的认证实现起来要多做一些工作，但这并不表示这种认证最不好。首先，要为用户的登录创建你自己的定制HTML表单（不过，这当然也能由一个JSP生成），然后创建一个定制的HTML错误页面，当用户产生一个登录错误时容器就可以使用这个错误页面。最后，在DD中使用`<login-config>`元素把两个表单关联起来。注意：如果你在使用基于表单的认证，一定要启用SSL或会话跟踪，否则返回登录表单时容器可能不认识！

## 你要做的事情：

- ① 在DD中声明 `<login-config>`
- ② 创建一个HTML登录表单
- ③ 创建一个HTML错误表单

HTML登录表单中有3项是与容器通信的关键：

- `j_security_check`
- `j_username`
- `j_password`

### ① DD中……

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/loginPage.html</form-login-page>
    <form-error-page>/loginError.html</form-error-page>
  </form-login-config>
</login-config>
```

### ② loginPage.html中……

```
Please login daddy-o
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
  <input type="submit" value="Enter">
</form>
```

要让容器工作，HTML登录表单的  
action必须是：`j_security_check`

容器要求HTTP请求把用  
户名存储在`j_username`中

容器要求HTTP请求把口令  
存储在`j_password`中

### ③ loginError.html中……

```
<html><body>
  Sorry dude, wrong password
</body></html>
```

**千万  
注意！**

这一页的所有内容都是考试所要  
求的！

## 认证类型小结

下表总结了4种认证的主要性质。“规范”是指这种认证机制在HTTP规范还是J2EE规范中定义（提示：参加考试时你必须记住这个表）。

类型	规范	数据完整性	注释
BASIC	HTTP	Base64 - 弱	HTTP标准，所有浏览器都支持
DIGEST	HTTP	强一些 - 但不是SSL	对于HTTP和J2EE容器是可选的
FORM	J2EE	非常弱，没有加密	允许有定制的登录屏幕
CLIENT-CERT	J2EE	强 - 公共密钥(PKC)	很强，但是用户必须有证书

*there are no  
Dumb Questions*

**问：** 数据完整性与认证有什么关系？

**答：** 认证用户时，他会给你发送用户名和口令。数据完整性和机密性是指窃听者窃取或篡改此信息的程度。稍后将讨论如何在登录期间实现数据完整性和机密性。

数据完整性是指到达的数据与发出的数据是一样的。换句话说，在发送过程中没有人篡改数据。数据机密性是指，传送过程中别人都无法看到数据。不过，大多数情况下，我们认为数据完整性和机密性目标是一样的，都是在传输过程中保护数据。

*Sharpen your pencil*

为这个基于表单的认证应用填空。这只是要帮助你记住DD和HTML表单中与认证有关的部分（答案在前一页上）。

### DD

```
<login-config>
    <auth-method>[ ]</auth-method>
    <form-login-config>
        <[ ]>/loginPage.html</ [ ]>
        <form-error-page>/loginError.html</form-error-page>
    </form-login-config>
</login-config>
```

### HTML

```
Please login daddy-o
<form method="POST" action=[ ]>
    <input type="text" name=[ ]>
    <input type="password" name="j_password">
    <input type="submit" value="Enter">
</form>
```



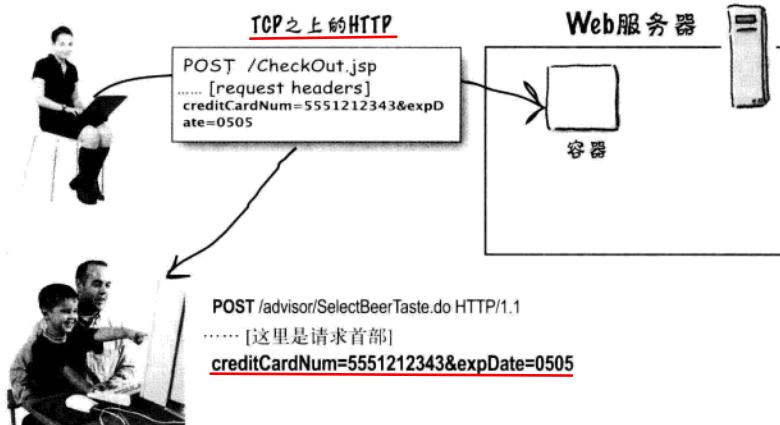
## 她不知道J2EE的 “有保护的传输层连接”

别着急。定制登录和安全可以兼得。登录数据也是数据，所以可以用保护网上购物者信用卡号的方式来保证安全，为此可以使用J2EE兼容容器的数据完整性和机密性特性。

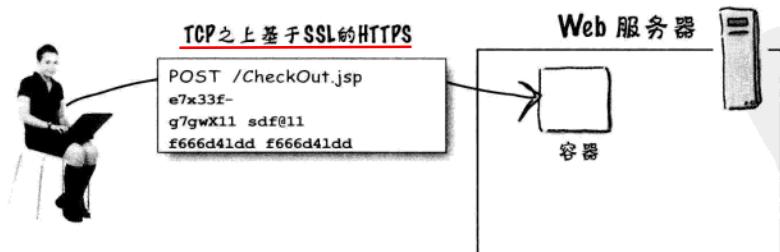
## 保护正在传输的数据的解决之道：HTTPS

你告诉一个J2EE容器你想实现数据机密性和/或完整性，J2EE规范可以保证所传输的数据会经过一个“有保护的传输层连接”传送。换句话说，容器不必使用任何特定的协议来处理安全传输，但在实际中几乎都会使用SSL之上的HTTPS。

### HTTP请求——不安全



### SSL请求之上的安全HTTPS





## Sharpen your pencil

每个请求和响应都必须安全吗？

如果不是，应用中哪一部分需要有保护的传输？

你觉得数据机密性是什么意思？

考虑一下这一章前面介绍的内容。如果你的Web应用要做到快速、高效和安全，请回答一些问题……（这道题没有答案，你要自己得出答案）。

你觉得数据完整性是什么意思？

如果能只对某些请求和响应应用传输安全策略，你怎么告诉容器究竟是哪些请求和响应？

有没有其他DD元素能以同样的粒度来声明有保护的传输？

# 如何以声明方式保守地实现数据机密性和完整性

再来看DD。实际上，我们还会用“老朋友”<security-constraint>来实现机密性和数据完整性，为此要增加一个<user-data-constraint>元素。你可以想想看，这是有道理的。如果你要为一个资源授权，很可能要考虑是否希望安全地传输数据。

```

<web-app>
    ...
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Recipes</web-resource-name>
            <url-pattern>/Beer/UpdateRecipes/*</url-pattern>
            <http-method>POST</http-method>
        </web-resource-collection>

        <auth-constraint>
            <role-name>Member</role-name>
        </auth-constraint>

        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

```

就是这里！所有数据完整性和机密性都在<user-data-constraint>元素中处理。

<transport-guarantee>的合法值

把这3个子元素放在一起来看，可以读作：

只有Member可以对UpdateRecipes目录中找到的资源完成POST请求，而且确保传输是安全的。

你可能不会指定NONE，因为如果你不打算保护数据，就没有必要使用<user-data-constraint>！

NONE

这是默认值，意味着没有数据保护。

INTEGRAL

数据在传输过程中不能更改。

CONFIDENTIAL

数据在传输过程中不能被别人看到。

注意：尽管规范里没有要求，但实际中几乎所有容器都使用了SSL来实现可靠传输，这说明INTEGRAL和CONFIDENTIAL的效果是一样的，任意一个都能同时提供机密性和数据完整性。因为每个<security-constraint>只有一个<user-data-constraint>，所以有人会建议使用CONFIDENTIAL，不过，还是那句话，在实际中这没有任何影响，除非你转而采用一个不使用SSL的新容器（这很少见）。



## 保护请求数据

记住，在DD中，`<security-constraint>`针对的是请求之后发生的事情。换句话说，如果容器开始查看`<security-constraint>`元素来决定如何响应，此时客户已经做出了请求。请求数据已经在传输了。你怎么可能这样提醒浏览器，“哦，顺便说一句……如果用户要请求这个资源，发送请求前请先换用安全套接字（secure sockets, SSL）。”这是不可能的。

那该怎么做？

你已经知道怎么为客户提供一个登录屏幕，为此要在DD中定义一个受限资源，未经认证的客户做出请求时，容器就会自动地触发认证过程。

现在我们要明确怎么保护来自请求的数据……即使（有时还特别强调这一点）此时客户还没有登录。

我们可能想保护他们的登录数据！

翻到下一页，看看该怎么做……

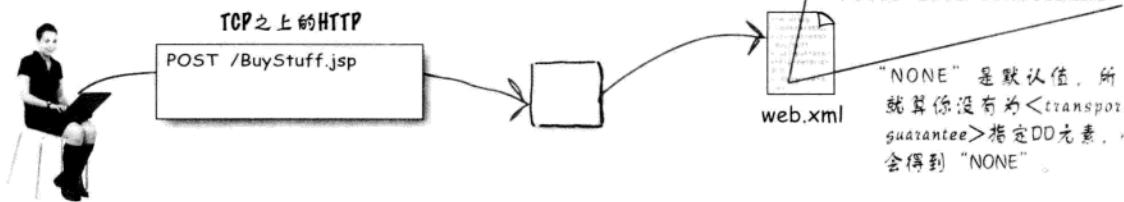
未经认证的客户请求一个没有传输保证的受限资源。

- ① 客户请求/BuyStuff.jsp，DD中已经为这个页面配置了<security-constraint>。

容器检查<security-constraint>，并发现/BuyStuff是一个受限资源……这说明用户必须经过认证。容器发现这个请求没有传输保证。

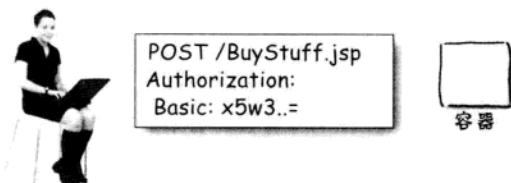
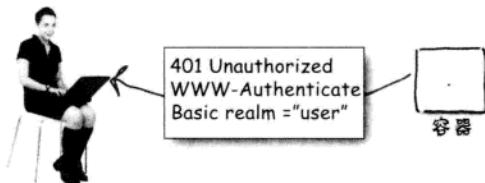
```
<user-data-constraint>
  <transport-guarantee>
    NONE
  </transport-guarantee>
</user-data-constraint>
```

“NONE”是默认值，所就算你没有为<transport-guarantee>指定DD元素，会得到“NONE”。



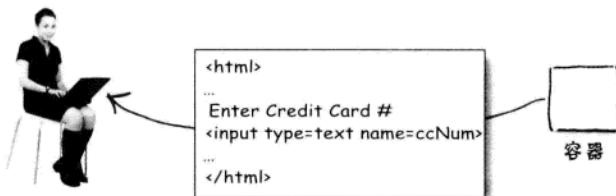
- ② 容器向客户发送一个401响应，告诉浏览器要从用户获得登录信息。

- ③ 浏览器再做这个请求，但是这一次还会在首部中提供用户的登录信息。



呀！客户的登录信息没有安全发送。用户名和口令没有得到保护！

- ④ 容器对用户进行认证（检查用户名和口令与服务器中配置的用户数据是否匹配）。然后容器对请求授权，确保这个用户的角色可以得到受限资源。一切都检查得当，然后发送响应。

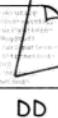
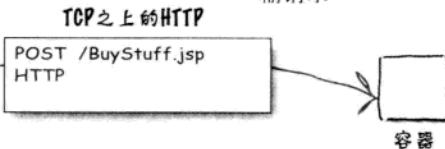


## 未经认证的客户请求一个受限资源，这个资源有机密性传输保证。

- ① 客户请求/BuyStuff.jsp，这个受限资源有传输保证。

容器看到这个受限资源有一个传输保证，而且没有安全地传输请求……

```
<user-data-constraint>
<transport-guarantee>
  CONFIDENTIAL
</transport-guarantee>
</user-data-constraint>
```

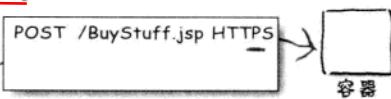


- ② 容器向客户发送一个301响应，这就告诉浏览器使用一个安全传输来完成请求的重定向。

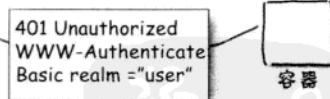


不错，“301”确实用于正常的重定向。不过，在这里它表明容器在这样告诉浏览器：“嘿，下一次你得通过一个安全连接过来，然后我们再看我们能不能对话……”

- ③ 浏览器再做一次资源请求，不过这一次会通过一个安全连接。换句话说，资源还是一样的，但是现在用的协议是HTTPS。

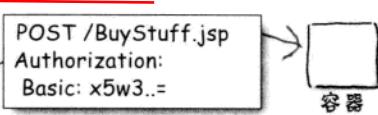


- ④ 现在容器看到资源是受限的，而且这个用户未经认证，所以容器开始认证过程，向浏览器发送一个“401”……



关键：请求到来时，容器首先查看<transport-guarantee>，如果有这样一个元素，容器处理时会先问一个问题：“这个请求来自一个安全连接吗？”如果不是，容器根本不会去查看认证/授权信息，而是告诉客户“等你安全时再回来，然后我们再说……”

- ⑤ 浏览器再做这个请求（不错，第三次了），但是这一次请求的首部中提供了用户的登录数据，而且请求使用一个安全连接传送。所以，这一次客户的登录数据会安全地传输！





为了确保用户的登录数据能安全地传输到服务器，要对每个可能触发登录过程的受限资源设置一个传输保证！

记住，使用声明式认证时，客户不会直接请求登录。实际上，客户请求受限资源时就会触发登录/认证过程。所以，如果你希望确保客户的登录数据能通过一个安全连接到达服务器，就要对每个可能为客户触发登录表单的受限资源设置一个`<transport-guarantee>`！

这样一来，容器得到对受限资源的请求时，在要求浏览器获得客户的登录数据之前，容器会告诉浏览器，“如果你还没有使用安全连接，那你别打算做这个请求”。等客户第二次回来时，容器则会说，“哦，我看到你使用了一个安全连接，但是我还需要用户的认证数据”。浏览器为用户提供登录表单，得到用户的信息，再通过一个安全连接返回（这已经是第三次了）。

### *there are no Dumb Questions*

**问：**我不明白，当请求并非通过一个安全连接到来时，为什么容器要向客户发回一个REDIRECT (301)。这不就是重定向到原来的请求吗？

**答：**通常你认为重定向就是“嘿，浏览器，再到另一个URL去”。记住，重定向对客户是不可见的：客户的浏览器会自动地做出一个新请求，目标是服务器返回的重定向(301)头部中指定的URL。

但是对于传输安全则稍有不同。此时容器不是这样告诉客户浏览器：“重定向到一个不同的资源”，而是：“重定向到同一个资源，但是要使用另一个协议，要使用HTTPS而不是HTTP”。

**问：**那么，SSL之上的HTTPS是不是以某种方式内置在容器中？

**答：**规范中没有要求，但是你的容器极有可能使用SSL（安全套接字）之上的HTTPS。不过，这不一定是自动的！你可能必须在你的容器中配置SSL。而且更重要地，你需要一个证书！

请查看容器的文档，你的容器很有可能会生成一个证书供你测试使用，但是在生产阶段，还是要从一个“官方”机构（如VeriSign）得到一个公共密钥证书。

（顺便说一句，证书以及HTTPS和SSL等安全协议在考试中不会考。你只需知道必须在DD中做什么以及为什么这么做就行了。没有必要成为系统管理员和网络安全大师。）



配置一个Web应用的安全有关方面，填写DD中的以下3部分。这个Web应用必须有以下行为：

你希望所有人都能对Beer/UpdateRecipes目录（包括其所有子目录）中的资源完成GET调用，但是只有安全角色为“Admin”的人才能对该目录中的资源做POST调用。另外，你希望数据得到保护，以防被人窃听。

<web-app...>

<security-constraint>

</security-constraint>

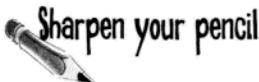
...

</web-app>



填下表，写出相关的DD元素。翻到下一页就会看到答案（但不要偷看！）

安全目标	DD中该放什么
你希望容器自动完成基本认证。	
你希望使用自己的定制表单页面，名为“loginPage.html”（就部署在Web应用的根目录下），而且如果客户无法通过认证，你希望显示“loginError.html”。	
(不必包括配置登录信息所需的DD元素。)	
(不必包括配置登录信息所需的DD元素。)	



## 答案

你希望所有人都能对Beer/UpdateRecipes目录（包括其所有子目录）中的资源完成GET调用，但是只有安全角色为“Admin”的人才能对该目录中的资源做POST调用。另外，你希望数据得到保护，以防被人窃听。

```
<web-app...>
```

```
  <security-constraint>
```

```
    <web-resource-collection>
```

```
      <web-resource-name>Recipes</web-resource-name>
```

```
      <url-pattern>/Beer/UpdateRecipes/*</url-pattern>
```

```
      <http-method>POST</http-method>
```

```
    </web-resource-collection>
```

记住，受保护目录的URL模式要以“/\*”结尾。

```
  <auth-constraint>
```

```
    <role-name>Admin</role-name>
```

```
  </auth-constraint>
```

如果没有指定任何<auth-constraint>，每个人都能完成POST调用。放入Admin是指，只有Admin才能访问这个URL模式/HTTP方法组合。

```
  <user-data-constraint>
```

```
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
```

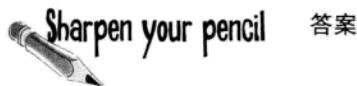
```
  </user-data-constraint>
```

这里也可以是INTEGRAL，几乎在所有容器上配置INTEGRAL也能得到机密性，因为容器会使用SSL来提供传输保证（不过规范中对这一点没有明确要求）。

```
</security-constraint>
```

```
...
```

```
</web-app>
```



## 安全目标

## DD中该放什么

你希望容器自动完成基本认证。

```
<web-app...>
...
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>

</web-app>
```

你希望使用自己的定制表单页面，名为“`loginPage.html`”（就部署在Web应用的根目录下），而且如果客户无法通过认证，你希望显示“`loginError.html`”。

```
<web-app...>
...
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/loginPage.html</form-login-page>
        <form-error-page>/loginError.html</form-error-page>
    </form-login-config>
</login-config>

</web-app>
```

你想限制所有扩展名为“.do”的资源，使得所有客户都能对其调用GET，但是只有Member才能完成POST调用。

要配置两点：受限资源（也就是URL模式和HTTP方法），以及`<auth-constraint>`（定义可以在指定`<url-pattern>`上访问指定`<http-method>`的安全角色）。

```
<web-app...>
...
<security-constraint>
    <web-resource-collection>
        <web-resource-name>CoolThings</web-resource-name>
        <url-pattern>*.do</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>Member</role-name>
    </auth-constraint>

</security-constraint>
</web-app>
```

使用扩展名URL模式，总是以星号(\*)开头。

你想限制`foo/bar`目录中的所有资源，使得只有安全角色为Admin的人才能对这些资源调用任何HTTP方法。

```
<web-app...>
...
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Stuff</web-resource-name>
        <url-pattern>/foo/bar/*</url-pattern>
    </web-resource-collection> ↗ 没有<http-method>，所以除了Admin角色的人之外，其他人不得访问任何HTTP方法。

    <auth-constraint>
        <role-name>Admin</role-name>
    </auth-constraint>

</security-constraint>
</web-app>
```


**Sharpen your pencil** 答案

	没有人	Guest	Member	Admin	所有人
①		X			
②	X				
③		X		X	
④					X
⑤					X
⑥	X				

假设没有定义  
    <auth-constraint>



## 第12章 模拟测验

1 哪些安全机制总是独立于传输层完成？（选出所有正确的答案）

- A. 授权
- B. 数据完整性
- C. 认证
- D. 机密性

2 给定一个部署描述文件，其中有3个合法的<security-constraint>元素，这3个元素都指定了对Web资源A的约束，3个<security-constraint>元素各自的<auth-constraint>子元素如下：

```
<auth-constraint>
    <role-name>Bob</role-name>
</auth-constraint>
<auth-constraint/>
<auth-constraint>
    <role-name>Alice</role-name>
</auth-constraint>
```

谁能访问资源A？

- A. 没有人
- B. 所有人
- C. 仅Bob
- D. 仅Alice
- E. 仅Bob和Alice
- F. 除了Bob或Alice

3 以下哪些行为可以通过J2EE 1.4容器的数据完整性机制来处理？（选出所有正确的答案）

- A. 验证一个特定用户是否可以访问一个特定的HTML页面。
- B. 确保窃听者不能读到从客户发送到容器的HTTP消息。
- C. 验证对一个受限JSP做出请求的客户有适当的角色凭证来访问这个JSP。
- D. 确保从容器向客户传送一个HTTP消息时，黑客不能篡改这个HTTP消息的内容。

4 使用基于表单的认证时，登录表单中哪些域是必要的？（选出所有正确的答案）

- A. **pw**
- B. **id**
- C. **j\_pw**
- D. **j\_id**
- E. **password**
- F. **j\_password**

5 哪些类型的认证需要一种特定的HTML action？（选出所有正确的答案）

- A. HTTP 基本认证
- B. 基于表单的认证
- C. HTTP摘要认证
- D. HTTPS客户认证

6 哪些安全机制可以使用HttpServletRequest接口中的一个方法来实现？（选出所有正确的答案）

- A. 授权
- B. 数据完整性
- C. 认证
- D. 机密性

7 哪个HttpServletRequest方法与<security-role-ref>元素的使用最为相关？

- A. `getHeader`
- B. `getCookies`
- C. `isUserInRole`
- D. `getUserPrincipal`
- E. `isRequestedSessionIDValid`

8 哪些部署描述文件元素可以包含一个<transport-guarantee>子元素？  
(选出所有正确的答案)

- A. <auth-constraint>
- B. <security-role-ref>
- C. <form-login-config>
- D. <user-data-constraint>

9 哪种认证机制建议只在启用了cookie或SSL会话跟踪时才使用？

- A. HTTP基本认证
- B. 基于表单的认证
- C. HTTP摘要认证
- D. HTTPS客户认证



## 第12章 模拟测验答案

( servlet规范：第12章 )

1 哪些安全机制总是独立于传输层完成？（选出所有正确的答案）

- A. 授权
- B. 数据完整性
- C. 认证
- D. 机密性

A是对的，一旦发生认证，授权就完全在容器中进行。  
认证可以根据<auth-method>元素的设置影响传输层。

2 给定一个部署描述文件，其中有3个合法的<security-constraint>元素，( servlet规范：12.8.1 )

这3个元素都指定了对Web资源A的约束，3个<security-constraint>元素各自的<auth-constraint>子元素如下：

```
<auth-constraint>
    <role-name>Bob</role-name>
</auth-constraint>
<auth-constraint/>
<auth-constraint>
    <role-name>Alice</role-name>
</auth-constraint>
```

谁能访问资源A？

- A. 没有人
- B. 所有人
- C. 仅Bob
- D. 仅Alice
- E. 仅Bob和Alice
- F. 除了Bob或Alice

A是对的，如果存在空的<auth-constraint>元素，会覆盖有关该资源的所有其他<auth-constraint>元素，所以任何人都不允许访问。

3

以下哪些行为可以通过J2EE 1.4容器的数据完整性机制来处理？（选出所有正确的答案）  
(Servlet规范：12.1)

- A. 验证一个特定用户是否可以访问一个特定的HTML页面。
- B. 确保窃听者不能读到从客户发送到容器的HTTP消息。  
B描述的是机密性。
- C. 验证对一个受限JSP做出请求的客户有适当的角色凭证来访问这个JSP。
- D. 确保从容器向客户传送一个HTTP消息时，黑客不能篡改这个HTTP消息的内容。  
D是对的，这通常使用HTTPS来完成。

4

使用基于表单的认证时，登录表单中哪些域是必要的？（选出所有正确的答案）

- A. pw
- B. id
- C. j\_pw
- D. j\_id
- E. password
- F. j\_password  
F是对的，用户的口令必须存放在名为j\_password的域中。  
另外，用户名必须存放在j\_username中。

(Servlet规范：12.5.3)

5

哪些类型的认证需要一种特定的HTML action？（选出所有正确的答案）

- A. HTTP 基本认证
- B. 基于表单的认证  
B是对的，要进行基于表单的认证，登录表单的action必须是j\_security\_check。
- C. HTTP摘要认证
- D. HTTPS客户认证

(Servlet规范：12.5.3.1)

哪些安全机制可以使用**HttpServletRequest**接口中的一个方法来实现? (Servlet 规范: 12.3)

(选出所有正确的答案)

A. 授权

A是对的, 可以在程序中使用`isUserInRole`方法, 帮助确定客户的角色是否允许访问一个给定资源。

B. 数据完整性

C. 认证

C是对的, 可以通过程序使用`getRemoteUser`方法, 帮助确定客户是否经过认证。

D. 机密性

哪个**HttpServletRequest**方法与<**security-role-ref**>元素的使用最为相关? (Servlet 规范: 12.3)

1

A. `getHeader`

B. `getCookies`

C. `isUserInRole`

C是对的, <`security-role-ref`>元素用于将servlet中硬编码的角色映射到部署描述文件中声明的角色。可以在servlet中使用`isUserInRole`方法来测试<`security-role-ref`>元素的内容。

D. `getUserPrincipal`

E. `isRequestedSessionIDValid`

哪些部署描述文件元素可以包含一个<**transport-guarantee**>子元素? (Servlet 规范: 13.4)

(选出所有正确的答案)

A. <`auth-constraint`>

B. <`security-role-ref`>

C. <`form-login-config`>

D. <`user-data-constraint`>

D是对的, <`transport-guarantee`>元素在<`user-data-constraint`>元素中用于指定一个Web资源集是否应当使用诸如SSL之类的机制来传输。

9 哪种认证机制建议只在启用了cookie或SSL会话跟踪时才使用? (Servlet 规范: 12.5.3.1)

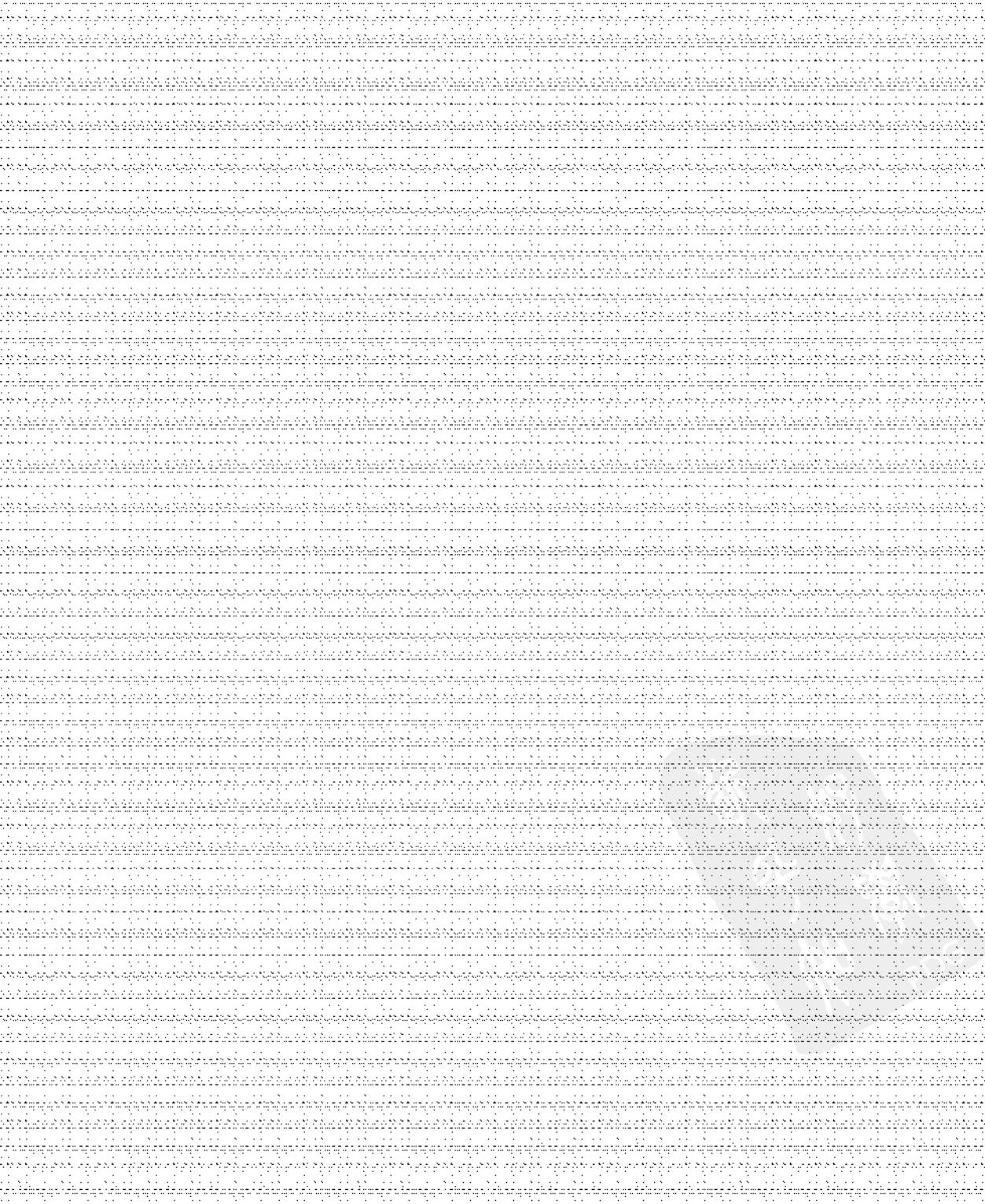
A. HTTP基本认证

B. 基于表单的认证

B是对的, 基于表单的登录会话跟踪很难实现, 因此建议采用一种单独的会话跟踪机制。

C. HTTP摘要认证

D. HTTPS客户认证



# 过滤器的威力



过滤器允许你拦截请求。如果能拦截请求，那你还控制响应。  
最棒的是，servlet对此一无所知。它不知道在客户做出请求和容器调用  
servlet的service()方法之间已经有人介入。对你来说，这意味着什么？这  
说明你会有更充裕的假期！因为照往常你可能要重写某个servlet才行，  
但现在则大可不必，只需编写和配置一个过滤器，它就能影响所有的  
servlet。想为应用中的每个servlet都增加用户请求跟踪吗？没问题。想要  
管理应用中每个servlet的输出吗？也没问题。而且，你甚至不用“接  
触”servlet代码。过滤器可能是最强大的Web应用开发工具了。

# OBJECTIVES

## 过滤器

### 内容说明：

- 3.3 描述Web容器请求处理模型；编写和配置过滤器；创建请求或响应包装器；给定一个设计问题，描述如何应用过滤器或包装器。

- 11.1 给定一个场景描述，列出了一系列问题，选择能够解决这些问题的模式。你必须了解的模式包括：拦截过滤器（Intercepting Filter）、模型-视图-控制器（Model-View-Controller）、前端控制器（Front Controller）、服务定位器（Service Locator）、业务委托（Business Delegate）和传输对象（Transfer Object）。
- 11.1 对于以下设计模式，将各模式与使用该模式可能带来的好处相匹配：拦截过滤器、模型-视图-控制器、服务定位器、业务委托和传输对象。

这个要求将在这一章中全面介绍。

本章介绍的过滤器是“拦截过滤器”模式一个例子（完全可以想见）。下一章将介绍模式，在此之前，我们并不讨论与模式有关的信息，不过，这一章你才会真正看到一个展示“拦截过滤器”模式的设计。

## 提升整个Web应用

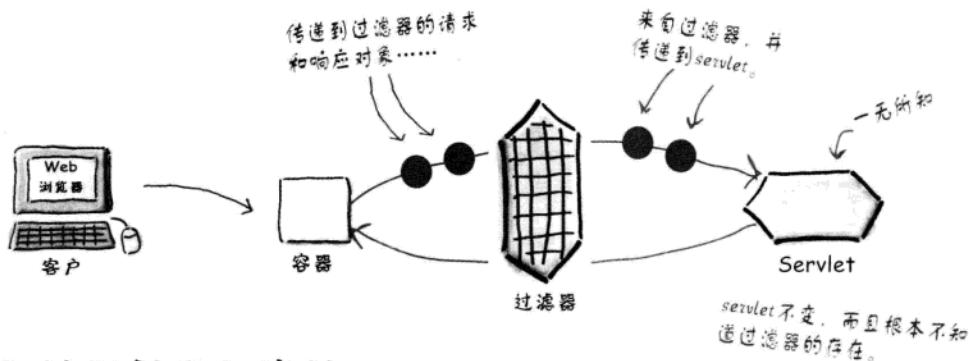
有时你要以某些方式来提升你的系统，让它能覆盖不同的用例或请求。例如，你可能想针对所有不同的用户交互跟踪系统的响应时间。



## 来个“过滤器”怎么样？

与servlet非常类似，过滤器就是Java组件，请求发送到servlet之前，可以用过滤器截获和处理请求，另外servlet结束工作之后，但在响应发回给客户之前，可以用过滤器处理响应。

容器根据DD中的声明来确定何时调用过滤器。在DD中，部署人员要建立映射，明确对于哪个请求URL模式要调用哪些过滤器。所以，要由部署人员（而不是程序员）来确定哪些请求或响应应当由哪些过滤器处理。



## 过滤器要做的事情

请求过滤器可以：

- ▶ 完成安全检查
- ▶ 重新格式化请求首部或体
- ▶ 建立请求审计或日志

响应过滤器可以：

- ▶ 压缩响应流
- ▶ 追加或修改响应流
- ▶ 创建一个完全不同的响应



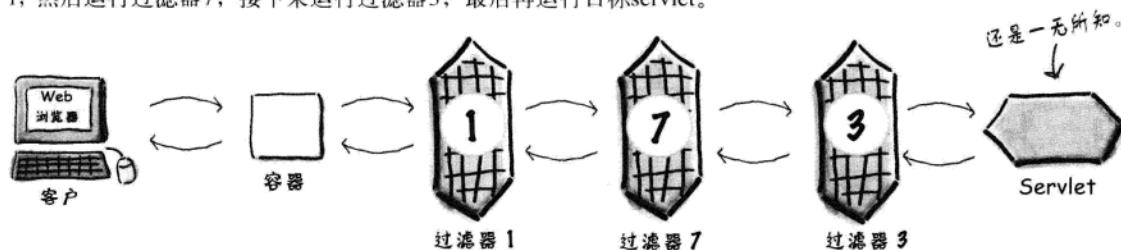
# 过滤器是模块化的，而且可以在DD中配置

过滤器可以链到一起，一个接一个地运行。过滤器设计为完全自包含的。过滤器并不关心在它前面运行了哪些过滤器（如果有的话），也不关心后面还会运行哪个过滤器。<sup>\*</sup>

过滤器运行的顺序由DD控制，本章稍后将讨论DD中过滤器的配置。

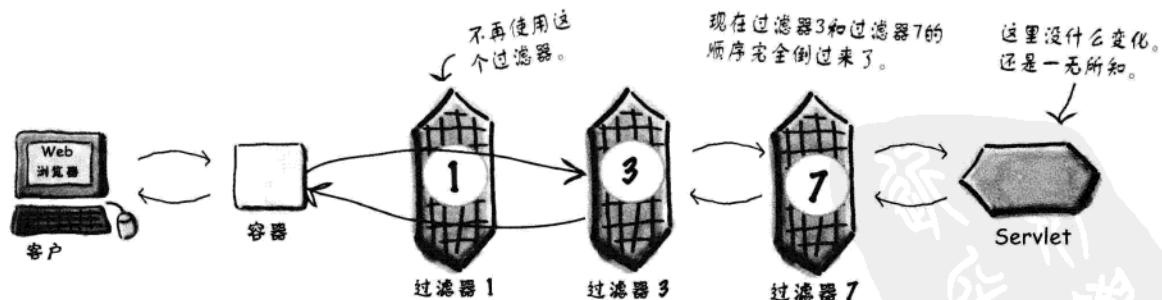
## DD配置1：

使用DD，可以这样告诉容器把过滤器链在一起：“对于这些URL，运行过滤器1，然后运行过滤器7，接下来运行过滤器3，最后再运行目标servlet。”



## DD配置2：

然后对DD稍做修改，可以删除或交换这些过滤器：“对于这些URL，运行过滤器3，然后运行过滤器7，再运行目标servlet。”



<sup>\*</sup> 我们说的也不完全对。部署人员通常需要根据过滤器所完成转换的结果来配置过滤器的顺序。例如，如果已经对图像应用了一个压缩过滤器，就不能再为它增加水印。在这种情况下，水印过滤器必须在数据到达压缩过滤器之前完成工作。不过关键是：作为程序员，你不能把这种依赖性写到代码里。



如果过滤器就像是servlet，我想它们必须由容器调用。而且，与servlet类似，他们可能有自己的生命周期……

## 过滤器在3个方面很像servlet

Kim说得没错，过滤器也存在于容器中。在很多方面，过滤器都与同在容器中的Servlet很类似。下面是过滤器和servlet的相似之处：

### 容器知道过滤器API

过滤器有自己的API。如果一个Java类实现了Filter接口，对容器来说就有很大不同，这个类会从原先一个普通的Java类摇身一变而成为一个正式的J2EE过滤器。过滤器API的其他成员允许过滤器访问ServletContext，而且可以与其他过滤器链接。

### 容器管理过滤器的生命周期

就像servlet一样，过滤器也有一个生命周期。类似于servlet，过滤器有init()和destroy()方法。对应于servlet的doGet()/doPost()方法，过滤器则有一个doFilter()方法。

### 都在DD中声明

Web应用可以有很多的过滤器，一个给定请求可能导致执行多个过滤器。针对请求要运行哪些过滤器，以及运行的顺序如何，这些都要在DD中声明。

# 建立请求跟踪过滤器

我们的任务是提升啤酒应用，只要有人请求与更新调酒配方有关的任何资源，就要跟踪到是谁做出的请求。以下就是这样一个过滤器。

```

package com.example.web;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;

public class BeerRequestFilter implements Filter {
    private FilterConfig fc; // Filter 和 FilterChain 都在 javax.servlet 中
    public void init(FilterConfig config) throws ServletException {
        this.fc = config; // 每个过滤器都必须实现 Filter 接口。
    }

    public void doFilter(ServletRequest req, // 必须实现 init()，通常只需在其
                        ServletResponse resp, // 中保存配置 (config) 对象。
                        FilterChain chain) // doFilter() 中才做具体的工作……注意这个方法并不取 HTTP 请求和响应对象做参数……而只是常规的
    throws ServletException, IOException { // ServletRequest 和 ServletResponse 对象。
        HttpServletRequest httpReq = (HttpServletRequest) req; // 但是我们能确定请求和响应对象肯定可以强制转换为相应的 HTTP 子类型。
        String name = httpReq.getRemoteUser();

        if (name != null) {
            fc.getServletContext().log("User " + name + " is updating");
        }
        chain.doFilter(req, resp); // 这是接下来要调用的过滤器或 servlet，后面还会更多地介绍这个内容。
    }

    public void destroy() {
        // 完成清理工作
    }
}

必实现 destroy()，但通常这个方法都为空。

```

过滤器不知道谁来调用它们，也不知道过滤器链中下一个是谁！

# 过滤器的生命周期

每个过滤器都必须实现Filter接口中的三个方法：  
init()、doFilter()和destroy()。

## 首先要有一个init()

容器决定实例化一个过滤器时，就要把握住机会，在init()方法中完成调用过滤器之前的所有初始化任务。前一页显示了最常见的实现：也就是保存FilterConfig对象的一个引用，以备过滤器以后使用。

## 真正的工作在doFilter()中完成

每次容器认为应该对当前请求应用过滤器时，就会调用doFilter()方法。doFilter()方法有3个参数：

- ▶ 一个ServletRequest  
(而不是HttpServletRequest)!
- ▶ 一个ServletResponse  
(而不是HttpServletResponse)!
- ▶ 一个FilterChain

过滤器的功能要在doFilter()方法中实现。如果过滤器要把用户名记录到一个文件中，就要在doFilter()中完成。你想压缩响应输出吗？也要在doFilter()中实现。

## 最后是destroy()

容器决定删除一个过滤器实例时，会调用destroy()方法，这样你就有机会在真正撤销实例之前完成所需的所有清理工作。

there are no  
Dumb Questions

**问：** FilterChain是什么？

**答：** 过滤器世界里，最酷的就要算FilterChain了。过滤器设计为一些模块化的

“积木”，可以采用多种不同方式把这些过滤器结合起来，共同完成一些事情，要让这一切成为可能，FilterChain功不可没。它知道接下来要发生什么。我们已经提到过，过滤器（更不用说servlet）并不知道请求所涉及的其他过滤器……但是必须有人知道过滤器执行的顺序才行，这个人就是FilterChain，它由DD中指定的filter元素驱动。

还要说一句，FilterChain与Filter都在同一个包里，即javax.servlet。

**问：** 我注意到，在你的doFilter()方法中做了这样一个调用：chain.doFilter()……在doFilter()调用一个doFilter()做什么？会不会造成无限递归呢？

**答：** FilterChain接口的doFilter()与Filter接口的doFilter()稍有不同。下面来告诉你它们的主要区别：

FilterChain的doFilter()方法要负责明确接下来调用谁的doFilter()方法（如果已经到了链尾，则是确定调用哪个servlet的service()方法）。但是Filter接口的doFilter()方法通常只完成过滤，这正是创建过滤器的初衷。

这说明，FilterChain可以调用一个过滤器或一个servlet（取决于是否到达链尾）。假设容器能把请求URL映射到一个servlet或JSP，那么链尾总是一个servlet或JSP（当然，这是指JSP生成的servlet）。

（如果容器无法找到所请求的正确资源，就不会调用过滤器。）

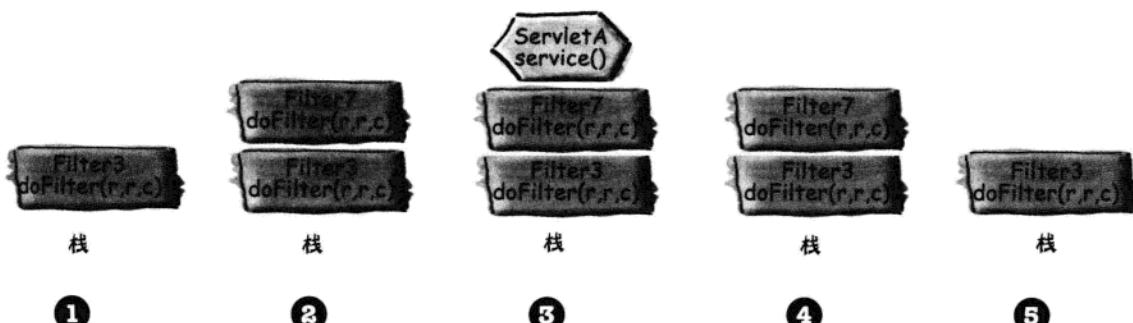
# 可以认为过滤器“可入栈”

Servlet规范并没有明确指出容器中如何处理chain.doFilter(req, resp)方法。不过，实际中考虑过滤器相互链接的过程时，可以把它们想像成栈上的方法调用。我们知道，容器内部所做的不止这些，但具体如何我们并不关心，要了解过滤器如何运行，只要一个概念性的栈就足够了（可能并不存在这样一个实际的物理栈）。

## 一个概念调用栈例子

在这个例子中，对ServletA的请求经过两个过滤器过滤，先是Filter3，然后是Filter7。

这个“概念栈”只是考虑过滤器链调用的一种思路。我们并不知道（也不关心）容器具体如何实现，不过这样考虑就完全能想像出过滤器链是怎样运行的。



等到请求时，容器会调用Filter3的doFilter()方法，它会在此运行，直到遇到其中的chain.doFilter()调用。

容器把Filter7的doFilter()方法压入栈顶，并执行这个方法，直到遇到其中的chain.doFilter()调用。

容器把ServletA的service()方法压入栈顶，执行这个方法，直到结束，然后从栈中弹出。

容器把控制交给Filter7，其doFilter()方法完成，然后从栈中弹出。

容器把控制交给Filter3，其doFilter()方法完成，从栈中弹出。然后容器完成响应。

# 声明和确定过滤器顺序

在DD中配置过滤器时，通常会做3件事：

- ▶ 声明过滤器。
- ▶ 将过滤器映射到你想过滤的Web资源。
- ▶ 组织这些映射，创建过滤器调用序列。

## 声明过滤器

```
<filter>
  <filter-name>BeerRequest</filter-name>
  <filter-class>com.example.web.BeerRequestFilter
    </filter-class>
  <init-param>
    <param-name>LogFileName</param-name>
    <param-value>UserLog.txt</param-value>
  </init-param>
</filter>
```

## 声明对应URL模式的过滤器映射

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```

## 声明对应Servlet名的过滤器映射

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <servlet-name>AdviceServlet</servlet-name>
</filter-mapping>
```

### <filter>的规则

- ▶ 必须有<filter-name>。
- ▶ 必须有<filter-class>。
- ▶ <init-param>是可选的，可以有多个<init-param>。

### <filter-mapping>的规则

- ▶ 必须有<filter-name>，用于链接到适当的<filter>元素。
- ▶ <url-pattern>或<servlet-name>元素二者中必须有一个。
- ▶ <url-pattern>元素定义了哪些Web应用资源要使用这个过滤器。
- ▶ <servlet-name>元素定义了哪个Web应用资源要使用这个过滤器。

### 重要提示：关于确定过滤器顺序的容器规则：

当多个过滤器映射到一个给定资源时，容器会使用以下规则：

1)先找到与URL模式匹配的所有过滤器。客户做出一个资源请求时，容器会采用URL映射规则来选择“适当的资源”，但这里不同，因为所有匹配的过滤器都会放在链中！！与URL模式匹配的过滤器会按DD中声明的顺序组成一个链。

2)一旦将与URL匹配的所有过滤器都放在链中，容器会用同样的办法确定与DD中<servlet-name>匹配的过滤器。

这不是太典型了吗……这样当然可以过滤来自客户的请求，但是他们忘了还有一些请求是通过转发和请求分派产生的。难道说……他们把请求分派看成是一种二等调用技术吗？！



## 特大新闻：2.4版本中，过滤器可以应用于请求分派器

想想看。过滤器可以应用于直接来自客户的请求，这当然很棒。但是如果请求来自转发或包含、请求分派和/或错误处理器呢？Servlet规范2.4提供了解决办法。

### 为通过请求分派请求的Web资源声明一个过滤器映射

```
<filter-mapping>
<filter-name>MonitorFilter</filter-name>
<url-pattern>*.do</url-pattern>
<dispatcher>REQUEST</dispatcher>
```

-和/或-

```
<dispatcher>INCLUDE</dispatcher>
```

-和/或-

```
<dispatcher>FORWARD</dispatcher>
```

-和/或-

```
<dispatcher>ERROR</dispatcher>
</filter-mapping>
```

### 声明规则

- ▶ 必须要有<filter-name>。
- ▶ 必须要有<url-pattern>或<servlet-name>元素其中之一。
- ▶ 可以有0~4个<dispatcher>元素。
- ▶ REQUEST值表示对客户请求启用过滤器。如果没有指定<dispatcher>元素，则默认为REQUEST。
- ▶ INCLUDE值表示对由一个include()调用分派来的请求启用过滤器。
- ▶ FORWARD值表示对由一个forward()调用分派来的请求启用过滤器。
- ▶ ERROR值表示对错误处理器调用的资源启用过滤器。



根据以下DD片段，写出对于每个请求路径，过滤器将以何种顺序执行。假设Filter1到Filter5已经得到适当的声明，而且servlet与其映射同名。（答案在这一章最后。）

```
<filter-mapping>
    <filter-name>Filter1</filter-name>
    <url-pattern>/Recipes/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Filter2</filter-name>
    <servlet-name>/Recipes/HopsList.do</servlet-name>
</filter-mapping>

<filter-mapping>
    <filter-name>Filter3</filter-name>
    <url-pattern>/Recipes/Add/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Filter4</filter-name>
    <servlet-name>/Recipes/Modify/ModRecipes.do</servlet-name>
</filter-mapping>

<filter-mapping>
    <filter-name>Filter5</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

### 请求路径

### 过滤器执行序列

**/Recipes/HopsReport.do**

过滤器：

**/Recipes/HopsList.do**

过滤器：

**/Recipes/Modify/ModRecipes.do**

过滤器：

**/HopsList.do**

过滤器：

**/Recipes/Add/AddRecipes.do**

过滤器：

## 用一个响应端过滤器压缩输出

前面我们展示了一个非常简单的请求过滤器。现在来看一个响应过滤器。响应过滤器稍微难一些，但是它们可能非常有用。利用响应过滤器，可以在servlet完成工作之后，而且是响应发送到客户之前，对响应输出做些处理。所以，不必一开始就介入，不用在servlet拿到请求之前就插手，完全可以在最后再接手，也就是servlet得到请求并生成了响应之后。

恩，我们得想想了……过滤器在链中总是在servlet之前调用，servlet肯定在链尾。  
不存在servlet之后才调用的过滤器。但是……还记得前面的栈图吧。在servlet完成其工作并从（虚拟）栈中弹出后，过滤器还会得到机会执行！



# 响应过滤器体系结构

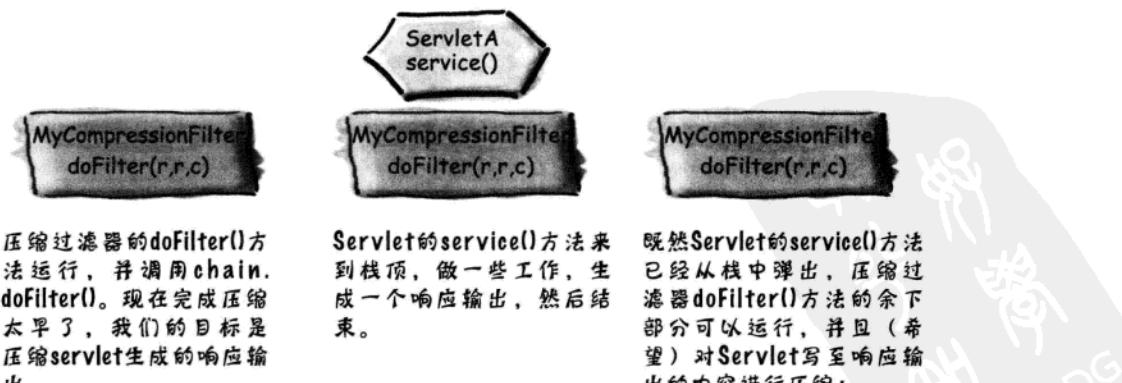
Rachel这样介绍doFilter()方法中代码的基本结构——首先，完成与请求相关的工作，然后调用chain.doFilter()，最后，当servlet（以及链中当前过滤器之后的所有其他过滤器）工作结束，而且控制返回到原先的doFilter()方法时，可以对响应再做点工作。

## Rachel为压缩过滤器设计的伪代码

```
class MyCompressionFilter implements Filter {
    init();
    public void doFilter(request, response, chain) {
        // 请求处理放在这里
        chain.doFilter(request, response); ←
        // 这里完成压缩逻辑 ←
        }                               既然servlet的工作已经结束，可以对
        destroy();                         servlet生成的响应进行压缩了.....
    }
}
```

*servlet在这里完成它的工  
作。*

## 概念调用栈

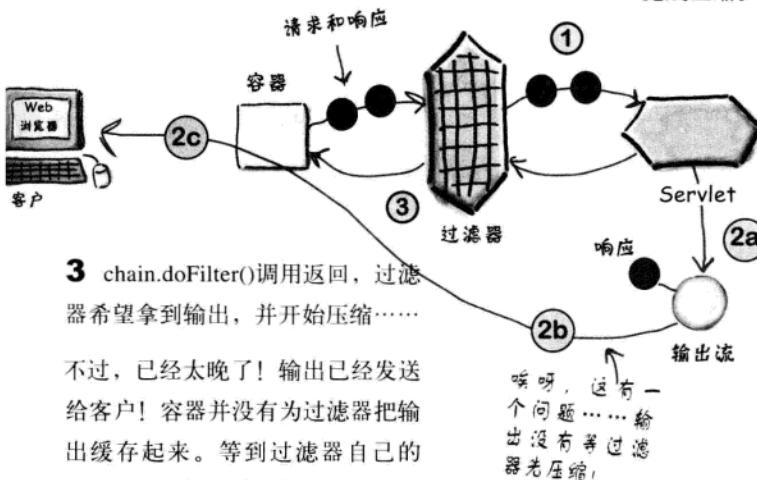


# 不过，真的这么简单吗？

压缩响应真的这么简单吗？只需等待servlet结束，然后压缩servlet的响应输出，是这样吗？毕竟，过滤器的doFilter()方法有响应对象的一个引用，这也是交给servlet的同一个响应对象，所以，从理论上讲，过滤器应该能访问响应输出……

```
public void doFilter(request, response, chain) {
    // 请求处理放在这里
    chain.doFilter(request, response); ① ②
    // 这里完成压缩逻辑 ③
}
```

**1** 过滤器把请求和响应传递给servlet，并且耐心地等待机会完成压缩。



**3** chain.doFilter()调用返回，过滤器希望拿到输出，并开始压缩……

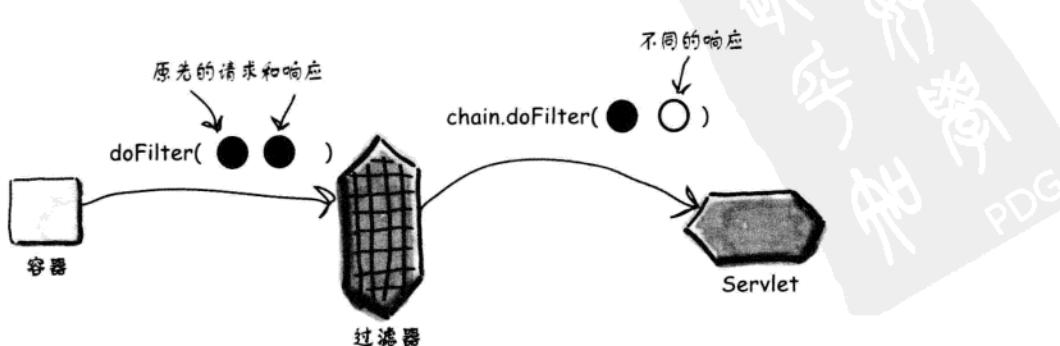
不过，已经太晚了！输出已经发送给客户！容器并没有为过滤器把输出缓存起来。等到过滤器自己的doFilter()方法置于（概念）栈的栈顶时，过滤器再想影响（处理）输出为时已晚。

**2a** servlet完成它的⼯作，创建输出，完全不知道这个输出要被压缩。

**2b** 输出通过容器返回……

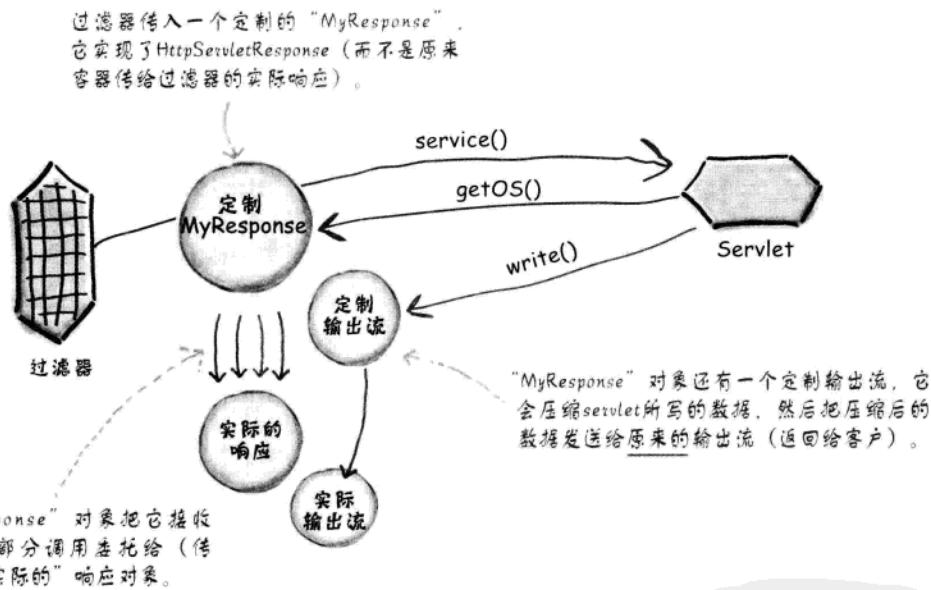
**2c** 发回给客户！唔，这就有问题了。过滤器本来希望能在输出交给客户之前先对输出做些处理的（压缩输出）。

## 输出改头换面



# 可以实现自己的响应

容器已经实现了HttpServletResponse接口；doFilter()和service()方法就是以这样一个响应作为参数。但是要让这个压缩过滤器正常工作，必须建立我们自己的HttpServletResponse接口定制实现，并把它通过chain.doFilter()调用传递到servlet。而且这个定制实现还必须包含一个定制输出流，因为这正是我们的目标，在servlet写输出之后并且在输出返回给客户之前，过滤器就能截获这个输出。



**问：** 过滤器把ServletRequest和ServletResponse对象传给链中的下一个过滤器（或servlet），而不是传递HttpServletResponse响应！那你为什么要实现HttpServletResponse呢？

**答：** 过滤器设计得要通用，所以理论上讲，你说的是对的。如果考虑到某个过滤器必须用于一个非Web应用，确实要实现非HTTP接口（ServletResponse），但是如今，开发非HTTP servlet的可能性几乎为0，所以不用担心。而且由于ServletResponse是HttpServletResponse的超类，所以如果一个地方本来需要传入ServletResponse，但实际传入了一个HttpServletResponse，这完全没问题。



### HttpServletResponse

接口太复杂了……必须实现所有这些方法，还要把调用委托给实际响应，要是有别的办法就好了……

### 她不知道servlet包装器类

创建你自己的定制HttpServletResponse实现可能很费劲。特别是当你只是想实现很少的几个方法时更是麻烦。由于HttpServletResponse接口扩展了另一个接口，所以要实现你自己的定制响应，就必须实现HttpServletResponse及其超接口ServletResponse中的所有方法。

不过，好在Sun公司已经有人为你做了这个工作，他们创建了一个便利的支持类，这个类实现了HttpServletResponse接口。这个类中的所有方法将调用分别委托给由容器创建的底层实际响应。

**ServletResponse** 接口  
(javax.servlet.ServletResponse)

<<interface>>  
**ServletResponse**

getBufferSize()  
setContent-Type()  
getOutputStream()  
getWriter()  
// 更多方法……

**HttpServletResponse** 接口  
(javax.servlet.http.HttpServletResponse)

<<interface>>  
**HttpServletResponse**

addCookie()  
addDateHeader()  
addHeader()  
encodeRedirectURL()  
encodeURL()  
sendError()  
sendRedirect()  
 setDateHeader()  
setHeader()  
setStatus()  
// 更多方法

↑  
记住，要实现HttpServletResponse，必须实现这个接口及其超接口ServletResponse中的所有方法。

# 包装器

servlet API中的包装器类功能极其强大，它们为你要包装的东西实现了所需的所有方法，并将所有调用委托给底层的请求或响应对象。你要做的只是扩展某个包装器，如果要在哪些方法中做些特殊的定制工作，只覆盖这些方法就行了。

当然，你在J2SE API中已经见过支持类，比如GUI的监听者适配器类。JSP API中也有，如定制标记支持类。尽管这些支持类及请求和响应包装器都是便利类，但包装器稍有不同，因为它们包装的对象正是他们实现的类型。换句话说，它们不只是提供了一个接口实现，还确实包含有相同接口类型的对象的一个引用，可以把方法调用委托给这个对象。顺便说一句，这与J2SE 中的“基本类型包装器”类（如Integer、Boolean、Double等）没有任何关系。

创建一个特定版本的请求或响应，这在创建过滤器时实在太常用了，所以Sun创建了4个“便利”类，以便更容易地完成这个任务：

- ▶ ServletRequestWrapper
- ▶ HttpServletRequestWrapper
- ▶ ServletResponseWrapper
- ▶ HttpServletResponseWrapper

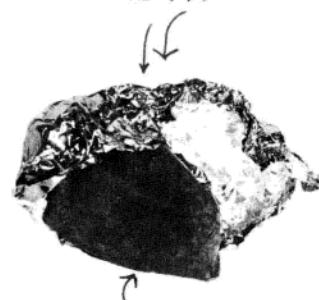


尽管在考试大纲里没有明确列出，但考试中可能会出现“装饰器”模式。

如果你熟悉普通的（非J2EE）设计模式，可能会认出，这种包装器类就是使用装饰器模式的一个例子（不过有时也称为包装器模式）。装饰器/包装器是指，装饰/包装某种有“提升”实现的对象，所谓“提升”，是指“增加新的功能”，而且仍然能做原来被包装对象所做的所有工作。

这就像是在说，“与我包装的东西相比，我是一个更好的版本，它做的我都能做，而且我还能做得更多。”装饰器/包装器模式的一个特点是，它把方法调用委托给被包装的对象，而不是完全替换。

包装器（定制的  
响应对象）



被包装对象（容器原来  
创建的响应对象）

**如果你想创建定制请求或响应对象，只需要派生某个便利请求或响应“包装器”类。**

**包装器包装了实际的请求或响应对象，而且把调用委托给（传给）实际的对象，还允许你对定制请求或响应做所需的额外处理。**

# 为设计增加一个简单的包装器

下面来提升Rachel的第一个伪代码设计，我们要增加一个包装器。

## 压缩过滤器设计（第2版，伪代码）

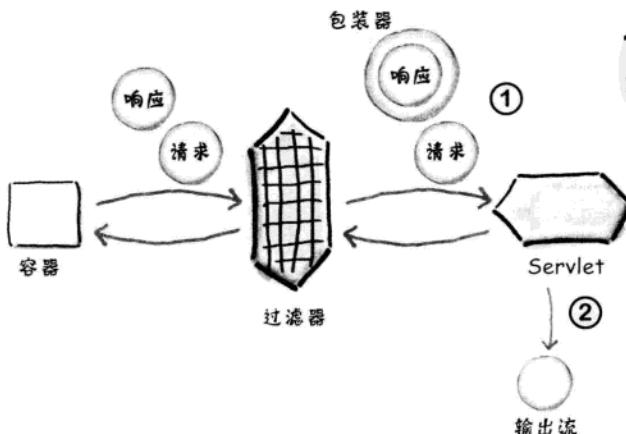
```
class CompressionResponseWrapper extends HttpServletResponseWrapper {
    // 覆盖你想定制的方法
}

class MyCompressionFilter implements Filter {
    public void init(FilterConfig cfg) { }

    public void doFilter( request, response, chain) {
        CompressionResponseWrapper wrappedResp
            = new CompressionResponseWrapper(response);
        chain.doFilter(request, wrappedResp); ← 现在沿着过滤器链发送这个对象。
        // 这里完成压缩逻辑
    }
    public void destroy() { }
}
}
```

出于我们自己的目的，派生这个包装器类……  
后面几页会做一些真正的覆盖！

就是这里用定制包装器类“包装”响应。



**1** 过滤器向servlet传递请求对象和一个定制响应回对象。

**2** 不过，因为我们没有覆盖包装器中的任何方法，所以输出流没有任何变化。

# 增加一个输出流包装器

上面增加第2个包装器……

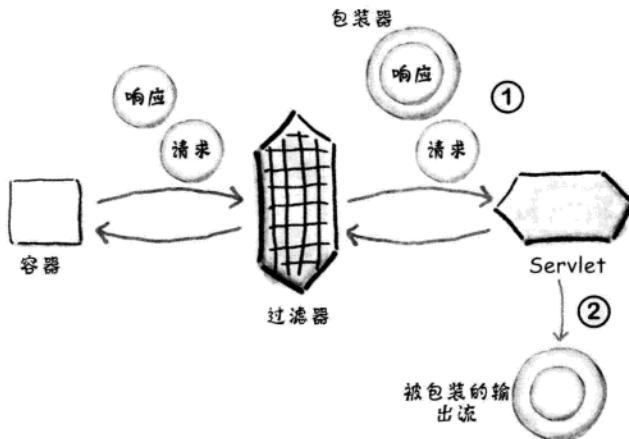
## 压缩过滤器设计（第3版，伪代码）

```
class CompressionResponseWrapper extends HttpServletResponseWrapper {
    public ServletOutputStream getOutputStream() throws... { // 覆盖这个方法返回一个定制输出流。
        ...
        servletGzipOS = new GzipSOS(resp.getOutputStream()); // 用我们定制的ServletOutputStream包装器类“包装”ServletOutputStream。这里假设GzipServletOutputStream（即代码中的GzipSOS）扩展了ServletOutputStream。
        return servletGzipOS;
    }
    // 可能覆盖其他方法。
}
```

向请求者返回一个“特殊的”  
ServletOutputStream。

```
class MyCompressionFilter implements Filter {
    public void init(FilterConfig cfg) { }

    public void doFilter( request, response, chain) {
        CompressionResponseWrapper wrappedResp
            = new CompressionResponseWrapper(response);
        chain.doFilter(request, wrappedResp);
        // 这里完成压缩逻辑
    }
    public void destroy() { }
}
```



1 过滤器向servlet传递请求对象和一个定制响应对象。  
定制响应有一个特殊的getOutputStream方法。

2 servlet请求一个输出流，但它不知道会得到一个“特殊”的输出流。

## 具体的压缩过滤器代码

该写代码了。在这一章的最后，我们来看一看压缩过滤器和它所用的包装器的代码。我们会对先前的讨论有所扩展，尽管这里有一些新的内容，但这些基本上都是普通的Java代码。

这个过滤器提供了一种压缩响应体内容的机制。这种过滤器通常应用于文本内容，如HTML，但是对PNG或MPEG等大多数媒体格式并不适用，因为这些媒体格式已经是压缩的。

```
package com.example.web;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.zip.GZIPOutputStream;

public class CompressionFilter implements Filter {

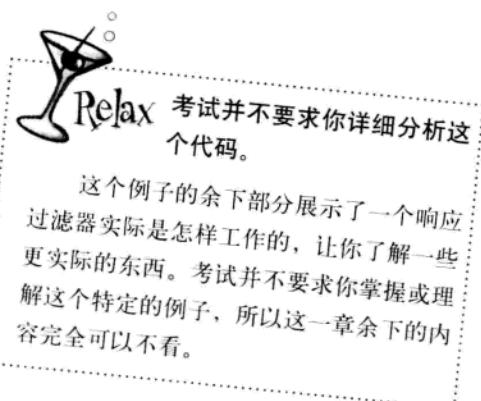
    private ServletContext ctx;
    private FilterConfig cfg;

    public void init(FilterConfig cfg) throws ServletException {
        this.cfg = cfg;
        ctx = cfg.getServletContext();
        ctx.log(cfg.getFilterName() + " initialized.");
    }

    public void doFilter(ServletRequest req,
                        ServletResponse resp,
                        FilterChain fc)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;

        String valid_encodings = request.getHeader("Accept-Encoding"); ← 客户接受GZIP压缩吗?
        if (valid_encodings.indexOf("gzip") > -1) {

            CompressionResponseWrapper wrappedResp ← 如果是，则用一个压缩
            = new CompressionResponseWrapper(response);
        }
    }
}
```



这个例子的余下部分展示了一个响应过滤器实际是怎样工作的，让你了解一些更实际的东西。考试并不要求你掌握或理解这个特定的例子，所以这一章余下的内容完全可以不看。

**init**方法保存配置对象，并保存servlet上下文对象的一个直接引用（以便完成日志记录）。

这个过滤器的核心是用装饰器包装响应对象，它用一个压缩I/O流包装输出流。当且仅当客户包含一个Accept-Encoding头部（具体为gzip）时，才会完成输出流的压缩。

## 压缩过滤器代码（续）

### 调试提示！

测试这个过滤器时，将这行代码注释掉。  
你会在浏览器中看到无法读懂的压缩数据。

```
wrappedResp.setHeader("Content-Encoding", "gzip");
```

声明响应内容用  
GZIP格式编码。

```
fc.doFilter(request, wrappedResp);
```

链到下一个组件。

```
GZIPOutputStream gzos = wrappedResp.getGZIPOutputStream();
gzos.finish();
```

GZIP压缩流必须“结束”，这  
也会刷新输出GZIP流缓冲区，  
将所有数据发送到原来的响应  
流。

容器处理余下的工作。

```
ctx.log(cfg.getFilterName() + ": finished the request.");
} else {
    ctx.log(cfg.getFilterName() + ": no encoding performed.");
    fc.doFilter(request, response);
}
```

```
public void destroy() {
    // 把实例变量置为null
    cfg = null;
    ctx = null;
}
```

### “题外话”

#### 压缩与HTTP

服务器怎么知道它能发送压缩的数据呢？浏览器怎么知道得到了压缩数据？看来，HTTP“懂压缩”；它的工作原理如下：

- ▶ 浏览器发送的某个首部（“Accept-Encoding: gzip”）告诉服务器：浏览器能够处理不同类型的内容。
- ▶ 如果服务器看到浏览器可以处理压缩数据，就会完成压缩，并向响应增加一个首部（“Content-Encoding: gzip”）。
- ▶ 浏览器接收到响应时，“Content-Encoding: gzip”首部告诉浏览器在显示数据之前先对其进行解压缩。

到现在为止还不错。区区包  
装器，能有多难？（这句话最  
近很有名……）



## 压缩包装器代码

前面已经了解了压缩过滤器；下面来看它使用的包装器。这是Servlet世界中最复杂的问题之一，如果你一下子没搞懂也不用担心。

这个响应包装器装饰了原来的响应对象，在原servlet输出流上增加了一个压缩装饰器。

```
package com.example.web;

// Servlet imports
import javax.servlet.http.*;
import javax.servlet.*;
// I/O imports
import java.io.*;
import java.util.zip.GZIPOutputStream;

class CompressionResponseWrapper extends HttpServletResponseWrapper {
```

```
    private GZIPOutputStream servletGzipOS = null;
```

```
    private PrintWriter pw = null; ← 压缩输出流的PrintWriter对象。
```

```
    CompressionResponseWrapper(HttpServletRequest resp) {
        super(resp);
    }
```

*super构造函数完成装饰器的职责：  
保存所装饰对象的一个引用，在这里被装饰的对象就是HTTP响应对象。*

```
    public void setContentLength(int len) { } ← 忽略这个方法——输出会得到压缩。
```

```
    public GZIPOutputStream getGZIPOutputStream() {
        return this.servletGzipOS.internalGzipOS;
    }
```

*过滤器使用的这个装饰器方法为压缩过滤器提供一个GZIP输出流的句柄，以便过滤器“完成”和刷新输出GZIP流。*

## 压缩包装器代码（续）

```

private Object streamUsed = null;

public ServletOutputStream getOutputStream() throws IOException {
    if ((streamUsed != null) && (streamUsed != pw)) {
        throw new IllegalStateException();
    }
    if ( servletGzipOS == null ) {
        servletGzipOS
            = new GZIPOutputStream(getResponse()
                .getOutputStream());
        streamUsed = servletGzipOS;
    }
    return servletGzipOS;
}

public PrintWriter getWriter() throws IOException {
    if ( (streamUsed != null) && (streamUsed != servletGzipOS)) {
        throw new IllegalStateException();
    }
    if ( pw == null ) {
        servletGzipOS
            = new GZIPOutputStream(getResponse()
                .getOutputStream());
        OutputStreamWriter osw
            = new OutputStreamWriter(servletGzipOS,
                getResponse().getCharacterEncoding());
        pw = new PrintWriter(osw);
        streamUsed = pw;
    }
    return pw;
}

```

过滤器和包装器

允许访问所装饰的  
servlet输出流。

仅当servlet还没有访问打印书写器  
时，允许servlet访问servlet输出流。

用我们的压缩输出流包装  
原来的servlet输出流。

允许访问所装饰的打印书写器。

当且仅当servlet还没有访问servlet输出  
流时，允许servlet访问打印书写器。

要建立一个打印书写器，必须  
首先包装servlet输出流，然后  
把压缩servlet输出流包装在另  
外两个输出流装饰器中：首先  
OutputStreamWriter把字符转换  
为字节，再用PrintWriter包装  
OutputStreamWriter对象。

## 压缩包装器，辅助类代码

辅助类是扩展了ServletOutputStream抽象类的一个装饰器，它使用了一个标准GZIP输出流，压缩所生成内容的具体工作就委托给这个GZIP输出流。

这个装饰器只需实现ServletOutputStream中的一个抽象方法：write(int)。神奇的委托就在这里发生！

```
class GZIPOutputStream extends ServletOutputStream {
    GZIPOutputStream internalGzipOS; ← 保存原始GZIP流的一个引用。这个实例变量在包范围内私有，所以响应包装器可以访问这个变量。
    /**
     * 装饰器构造函数 */
    GZIPOutputStream(ServletOutputStream sos) throws IOException {
        this.internalGzipOS = new GZIPOutputStream(sos);
    }

    public void write(int param) throws java.io.IOException {
        internalGzipOS.write(param); ← 这个方法把write()调用委托给GZIP压缩流，从而实现压缩装饰。GZIP压缩流包装了原来的ServletOutputStream（ServletOutputStream则包装了返回客户的TCP网络输出流）。
    }
}
```



## 答案

写出对于每个请求路径，过滤器以何种顺序执行。  
假设Filter1~Filter5已经得到适当的声明。

```
<filter-mapping>
  <filter-name>Filter1</filter-name>
  <url-pattern>/Recipes/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter2</filter-name>
  <servlet-name>/Recipes/HopsList.do</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter3</filter-name>
  <url-pattern>/Recipes/Add/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter4</filter-name>
  <servlet-name>/Recipes/Modify/ModRecipes.do</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter5</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

## 请求路径

## 过滤器序列

/Recipes/HopsReport.do

过滤器: **1, 5**

/Recipes/HopsList.do

过滤器: **1, 5, 2**

/Recipes/Modify/ModRecipes.do

过滤器: **1, 5, 4**

/HopsList.do

过滤器: **5**

/Recipes/Add/AddRecipes.do

过滤器: **1, 3, 5**



## 第13章 模拟测验

1 关于过滤器，哪些说法是正确的？（选出所有正确的答案）

- A. 过滤器只能过滤请求或响应回对象，但不能二者同时应用。
- B. **destroy**方法一定是容器回调方法。
- C. **doFilter**方法一定是容器回调方法。
- D. 过滤器只能通过一种途径调用，就是通过DD中的声明。
- E. 过滤器链中的下一个过滤器可以由前一个过滤器指定，或者在DD中指定。

2 关于在DD中声明过滤器，哪些说法是正确的？

（选出所有正确的答案）

- A. 不同于servlet，过滤器不能声明初始化参数。
- B. 过滤器链顺序总是由DD中元素出现的顺序决定。
- C. 扩展API请求或响应包装器类的类必须在DD中声明。
- D. 扩展API请求或响应包装器类的类使用了拦截过滤器模式。
- E. 过滤器映射通过<url-pattern>声明还是通过<servlet-name>声明，这会影响过滤器链的顺序。

3

给定类UserRequest是HttpServletRequest的一个实现，另外给定一个原本适当定义的**Filter**中有以下方法：

```
20. public void doFilter(ServletRequest req,
21.                         ServletResponse response,
22.                         FilterChain chain)
23.         throws IOException, ServletException {
24.     HttpServletRequest request = (HttpServletRequest) req;
25.     HttpSession session = request.getSession();
26.     Object user = session.getAttribute("user");
27.     if (user != null) {
28.         UserRequest ureq = new UserRequest(request, user);
29.         chain.doFilter(ureq, response);
30.     } else {
31.         RequestDispatcher rd = request.getRequestDispatcher("/login.jsp");
32.         rd.forward(request, response);
33.     }
34. }
```

哪种说法是正确的？

- A. 如果执行第31行，总会抛出一个异常。
- B. 第28行不对，因为必须把**request**作为第一个参数。
- C. 必须把**chain.doFilter(request, response)**插入到**else**块中的某个位置。
- D. 这个方法没有正确地实现**Filter.doFilter()**，因为方法签名是错误的。
- E. 以上都不对。

---

**4** 给定部署描述文件中的一部分：

```
11. <filter>
12.   <filter-name>My Filter</filter-name>
13.   <filter-class>com.example.MyFilter</filter-class>
14. </filter>
15. <filter-mapping>
16.   <filter-name>My Filter</filter-name>
17.   <url-pattern>/my</url-pattern>
18. </filter-mapping>
19. <servlet>
20.   <servlet-name>My Servlet</servlet-name>
21.   <servlet-class>com.example.MyServlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24.   <servlet-name>My Servlet</servlet-name>
25.   <url-pattern>/my</url-pattern>
26. </servlet-mapping>
```

哪些说法是正确的？（选出所有正确的答案）

- A. 这个文件不对，因为URL模式/**my** 同时映射到一个servlet和一个过滤器。
- B. 这个文件不对，因为servlet名或过滤器名都不允许包含空格。
- C. 对于与模式/**my**匹配的各个请求，过滤器**MyFilter**会在**MyServlet** servlet执行后调用。
- D. 对于与模式/**my**匹配的各个请求，过滤器**MyFilter**会在**MyServlet** servlet之前调用。
- E. 这个文件不对，因为<b><filter></filter></b>元素必须包含一个<b><servlet-name></servlet-name></b>元素，其中定义这个过滤器应该应用到哪个servlet。

---

5 关于过滤器，哪些说法是正确的？（选出所有正确的答案）

- A. 过滤器可以用于创建请求或响应包装器。
- B. 包装器可以用于创建请求或响应过滤器。
- C. 与servlet不同，所有过滤器初始化代码都应当放在构造函数中，因为没有**init()**方法。
- D. 过滤器支持一种初始化机制，它包含一个**init()**方法，使用过滤器处理请求之前肯定会调用这个方法。
- E. 过滤器的**doFilter()**方法必须按顺序在输入**FilterChain**对象上调用**doFilter()**，以确保所有过滤器都有机会执行。
- F. 在输入**FilterChain**上调用**doFilter()**时，必须将传入该方法的**ServletRequest**和**ServletResponse**对象传入过滤器的**doFilter()**方法。
- G. 过滤器的**doFilter()**可以阻塞进一步的请求处理。

---

6 关于servlet包装器类，哪些说法是正确的？（选出所有正确的答案）

- A. 唯一提供了包装**ServletResponse**对象的机制。
- B. 可以用于装饰实现了**Filter**的类。
- C. 应用不支持HTTP时也可以使用。
- D. 这个API为**ServletRequest**、**ServletResponse**和**FilterChain**对象提供了包装器。
- E. 实现了拦截过滤器模式。
- F. 派生一个包装器类时，必须至少覆盖包装器类的某个方法。



## 第13章 模拟测验答案

1

关于过滤器，哪些说法是正确的？选出所有正确的答案）

(Servlet v2.4 第6节)

- A. 过滤器只能过滤请求或响应回对象，但不能二者同时应用。
- B. `destroy`方法一定是容器回调方法。
- C. `doFilter`方法一定是容器回调方法。
- D. 过滤器只能通过一种途径调用，就是通过DD中的声明。
- E. 过滤器链中的下一个过滤器可以由前一个过滤器指定，或者在DD中指定。

C不对，`doFilter`既是一个回调方法，也是一个内联方法。

E不对，过滤器执行的顺序总是在DD中确定。

2

关于在DD中声明过滤器，哪些说法是正确的？

(Servlet v2.4 第6节)

- (选出所有正确的答案)
- A. 不同于servlet，过滤器不能声明初始化参数。
  - B. 过滤器链顺序总是由DD中元素出现的顺序决定。
  - C. 扩展API请求或响应包装器类的类必须在DD中声明。
  - D. 扩展API请求或响应包装器类的类使用了拦截过滤器模式。
  - E. 过滤器映射通过`<url-pattern>`声明还是通过`<servlet-name>`声明，这会影响过滤器链的顺序。

B不对，因为`<url-pattern>`映射总是在`<servlet-name>`映射前面。

D不对，因为包装器是装饰器模式的例子。

3

给定类UserRequest是HttpServletRequest的一个实现，另外给定一个原本适当定义的Filter中有以下方法：

(Servlet v2.4 49页)

```

20. public void doFilter(ServletRequest req,
21.                     ServletResponse response,
22.                     FilterChain chain)
22.     throws IOException, ServletException {
23.     HttpServletRequest request = (HttpServletRequest) req;
23.     HttpSession session = request.getSession();
25.     Object user = session.getAttribute("user");
26.     if (user != null) {
27.         UserRequest ureq = new UserRequest(request, user);
28.         chain.doFilter(ureq, response);
29.     } else {
30.         RequestDispatcher rd = request.getRequestDispatcher("/login.jsp");
31.         rd.forward(request, response);
32.     }
33. }
```

哪种说法是正确的？

- A. 如果执行第31行，总会抛出一个异常。      A不对，因为过滤器完全可以转发请求。
- B. 第28行不对，因为必须把request作为第一个参数。      B不对，因为过滤器包装请求是合法的（注意，UserRequest必须实现ServletRequest）。

- C. 必须把chain.doFilter(request, response)插入到else块中的某个位置。      C不对，因为doFilter方法不一定非得调用chain.doFilter()。
- D. 这个方法没有正确地实现Filter.doFilter(), 因为方法签名是错误的。      D不对，因为这个方法签名是正确的。
- E. 以上都不对。

4

给定部署描述文件中的一部分：

```

11. <filter>
12.   <filter-name>My Filter</filter-name>
13.   <filter-class>com.example.MyFilter</filter-class>
14. </filter>
15. <filter-mapping>
16.   <filter-name>My Filter</filter-name>
17.   <url-pattern>/my</url-pattern>
18. </filter-mapping>
19. <servlet>
20.   <servlet-name>My Servlet</servlet-name>
21.   <servlet-class>com.example.MyServlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24.   <servlet-name>My Servlet</servlet-name>
25.   <url-pattern>/my</url-pattern>
26. </servlet-mapping>

```

哪些说法是正确的？（选出所有正确的答案）

- A. 这个文件不对，因为URL模式/**my**同时映射到一个servlet和一个过滤器。 A不对，映射到过滤器和映射到servlet的URL模式如果相同，这种语法本身并无不妥，这是正确的。
- B. 这个文件不对，因为servlet名或过滤器名都不允许包含空格。 B不对，因为没有这个限制。
- C. 对于与模式/**my**匹配的各个请求，过滤器**MyFilter**会在**MyServlet** servlet执行后调用。 C不对，因为过滤器总是在servlet之前执行，而不是之后。
- D. 对于与模式/**my**匹配的各个请求，过滤器**MyFilter**会在**MyServlet** servlet之前调用。
- E. 这个文件不对，因为<**filter**>元素必须包含一个<**servlet-name**>元素，其中定义这个过滤器应该应用到哪个servlet。 E不对，因为<**servlet-name**>元素或<**url-pattern**>可以放在<**filter-mapping**>元素中。

(Servlet v2.4 51页)

5

关于过滤器，哪些说法是正确的？选出所有正确的答案)

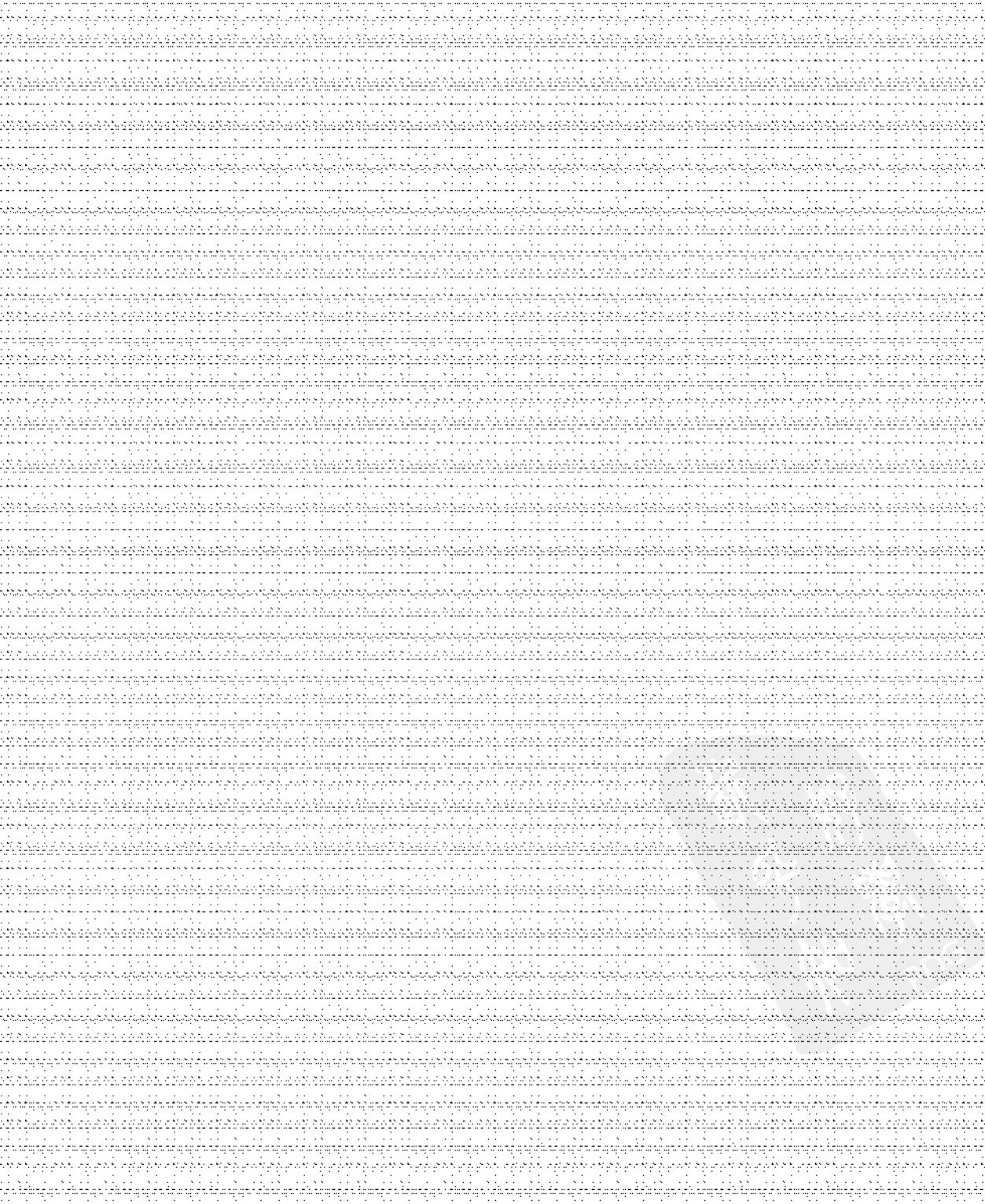
- A. 过滤器可以用于创建请求或响应包装器。      B不对，因为完全说反了。
- B. 包装器可以用于创建请求或响应过滤器。
- C. 与servlet不同，所有过滤器初始化代码都应当放在构造函数中，  
因为没有**init()**方法。      C不对，因为确实有一个用于过  
滤器初始化的**init()**方法。
- D. 过滤器支持一种初始化机制，它包含一个**init()**方法，使用过  
滤器处理请求之前肯定会调用这个方法。
- E. 过滤器的**doFilter()**方法必须按顺序在输入**FilterChain**对象  
上调用**doFilter()**，以确保所有过滤器都有机会执行。      E不对，因为如果一个过滤  
器想阻塞进一步的请求处理，  
不一定要调用**doFilter()**。
- F. 在输入**FilterChain**上调用**doFilter()**时，必须将传入该方法  
的**ServletRequest**和**ServletResponse**对象传入过滤器的  
**doFilter()**方法。      F不对，因为过滤器可能会“包装”请  
求或响应对象，再传递包装后的对象。
- G. 过滤器的**doFilter()**可以阻塞进一步的请求处理。

6

关于servlet包装器类，哪些说法是正确的？（选出所有正确的答案）

(API)

- A. 唯一提供了包装**ServletResponse**对象的机制。      A不对，因为可以创建你自  
己的包装器类。
- B. 可以用于装饰实现了**Filter**的类。      B不对，因为这些类用于包装请求和响应。
- C. 应用不支持HTTP时也可以使用。
- D. 这个API为**ServletRequest**、  
**ServletResponse**和**FilterChain**对象提供了包装器。      D不对，因为API没有提供  
**FilterChain**包装器。
- E. 实现了拦截过滤器模式。      E不对，因为这些包装器实现的是装饰器模式。
- F. 派生一个包装器类时，至少要覆盖包装器类的某个方法。



# 企业设计模式



已经有人做过了。如果你刚开始用Java开发Web应用，那你可真是幸运。已经有数百万的开发人员在这条路上摸索了很久，你能轻松地享用到他们智慧和经验的结晶。通过使用J2EE特有的设计模式以及其他设计模式，可以大大简化你的代码，你也能更轻松。Web应用最重要的设计模式是MVC，对此甚至还有一个相当流行的框架Struts。利用这个框架，你能构建一个灵活而且可维护的servlet前端控制器。一定要充分利用别人的成果，这样你才能把宝贵的时间用在更重要的事情上（比如滑雪、打高尔夫、跳舞、足球、打扑克、弹琴……）。

# OBJECTIVES

## J2EE模式

- 11.1 给定一个场景描述，列出一系列问题，选择以下哪个模式可以解决这些问题：拦截过滤器（Intercepting Filter）、模型－视图－控制器（Model-View-Controller）、前端控制器（Front Controller）、服务定位器（Service Locator）、业务委托（Business Delegate）和传输对象（Transfer Object）。
- 11.2 对于以下设计模式，将各模式与使用该模式可能带来的好处相匹配：拦截过滤器（Intercepting Filter）、模型－视图－控制器（Model-View-Controller）、前端控制器（Front Controller）、服务定位器（Service Locator）、业务委托（Business Delegate）和传输对象（Transfer Object）。

## 内容说明：

这一部分的要求会在本章中全面介绍。不仅仅是全面介绍，应该说是更深入地介绍。考试中，有关模式的问题是最容易的，所以对这一部分大可放心。

如果你对基本的企业设计模式已经很熟悉了，没准能直接回答出考试中有关模式的题目。

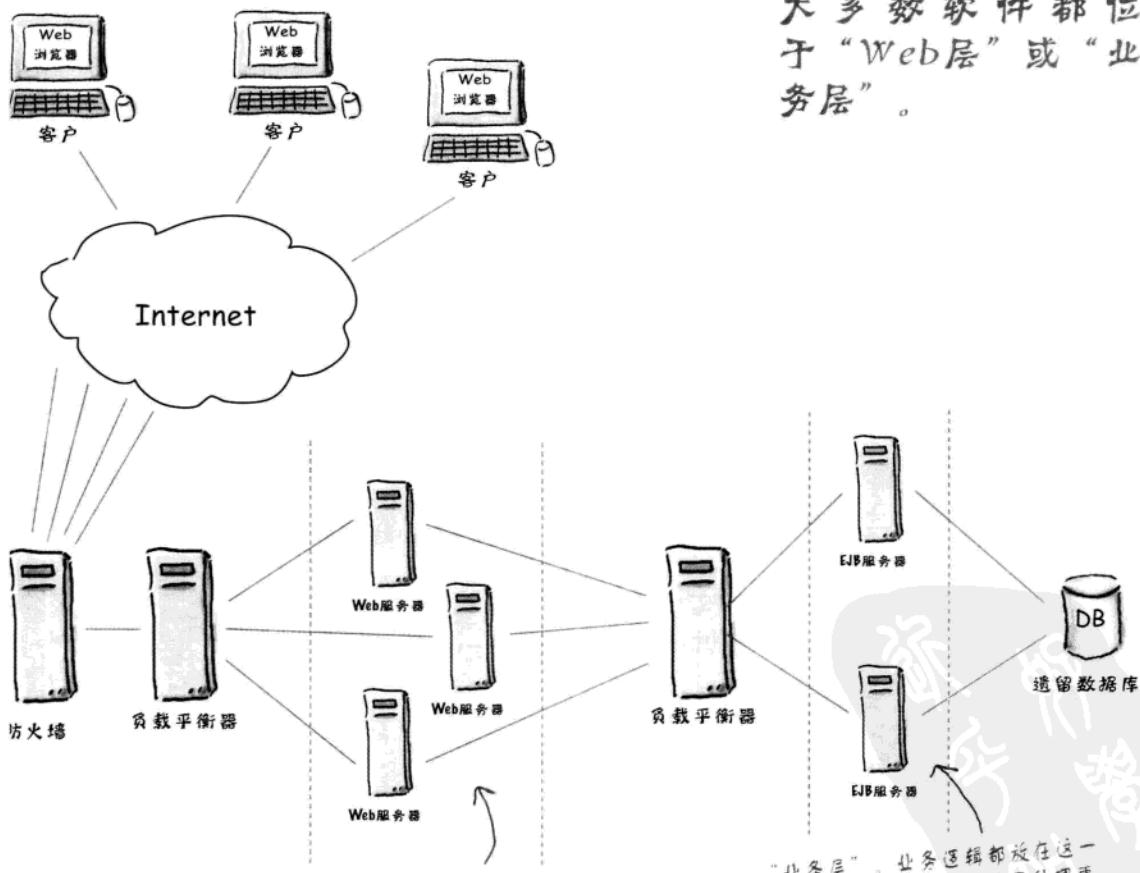
尽管考试中不要求掌握Struts，但这一章还是提供了Struts的一个介绍，这是目前MVC Web应用最常用的框架。

# 网站硬件可能很复杂

实际应用中，Web应用可能很复杂。一个热门网站每天的点击率可能会高达数百万次。要处理这么大的流量，大多数大型网站都建立了复杂的硬件体系结构，并且软件和数据分布在多个机器上。

你可能很熟悉的某个常用体系结构就是按功能“层”或“层次”来配置硬件。在一层上增加更多的计算机称为水平扩展（horizontal scaling），这也是提高吞吐量的最佳方法之一。

大型Web应用中，  
大多数软件都位于  
“Web层”或“业务层”。



“业务层”。业务逻辑都放在这一层。如果一个Web网站需要处理更大的流量，可以增加更多服务器。

# Web应用软件可能很复杂

我们已经看到了，一个Web应用可能由多种不同的软件组件所组成，这是相当常见的。Web层通常包括HTML页面、JSP、servlet、控制器、模型组件、图像等。业务层可能包括EJB、遗留的应用和查找注册库，大多数情况下还包括数据库驱动程序和数据库。



## 好在我们有J2EE模式

幸好，很多人一直在使用J2EE模式，利用这些模式往往能解决你遇到的问题。他们发现，要处理的问题本质上存在着一些重复性，针对这些问题，他们提出了可重用的解决方案。这些设计模式已经得到了其他开发人员的使用，经过了他们的测试和改进，所以你无需完全从头开始，亲力而为。

软件设计模式是“对常见软件问题的一种可重复的解决方案”。

### 常见的任务

Web应用最重要的任务是为最终用户提供一种可靠、有用而且正确的体验。换句话说，程序必须满足一些功能需求，如“选择啤酒种类”或者“在我的购物车里增加麦芽啤酒”。你要确保系统支持这些用例，除此之外，很可能还会面对另外一些后台需求，也就是非功能性需求。



#### 什么是“……性”？

你完成的（或可能要完成的）系统中，有哪些重要的非功能性需求？可以给你一个提示，大多数这种需求最后有以“……性”（ility）结尾，例如，可维护性（maintainability）。

## 性能（和“……性”）

下面是你很可能遇到的3个最主要的非功能性需求。

### ① 性能

如果你的网站太慢，用户就会流失（这是显而易见的）。在这一章中，你会了解到如何利用模式使用户得到更快的响应时间，以及如何利用模式帮助你的系统支持更多的并发用户（吞吐量）。（讨论传输对象时还会更多地介绍这个内容。）

### ② 模块性

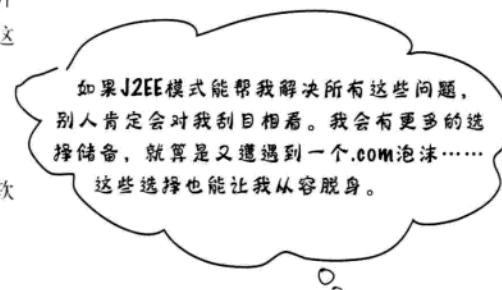
为了让应用的不同部分能同时在不同主机上运行，你的软件必须是模块化的……而且要以适当的方式模块化。

### ③ 灵活性、可维护性和可扩展性

**灵活性：**需要能够修改系统，而且无需经过太长的开发周期。可能有些组件“大减价”，但具有有效期限，你希望在系统中临时换用这些组件。在使用过程中，可能发现新组件中存在一个bug，又想换回原来的组件。这就要求系统必须灵活。

**可维护性：**也许你想更换一个数据库开发商，并以最快速度完成系统的更新；也可能遇到一些不太明确的bug，想尽快跟踪找出来；或者，管理员可能决定对公司的命名服务进行重构，而你必须立即相应调整！这就要求你的系统是可维护的。

**可扩展性：**市场部的人可能需要为大客户提供一个新的功能。或者用户的浏览器支持一个全新的特性，这些用户也要求你支持这个特性。所以你的系统最好是可扩展的！



## 明确我们的术语……

所有J2EE模式都相当依赖于你可能已经非常熟悉的一些常用软件设计原则。在后面几页，我们将介绍这些设计原则的一些有关术语。对于这些术语，不同的人和不同的书可能会有不同的看法，所以下面先明确我们的定义，以便你了解我们指的是什么。

## 遵循接口编写代码

你应该记得，接口就是两个对象之间的某种契约。一个类实现某个接口时，相当于表示：“我的对象会讲你的语言。”接口还有一个重要作用：**多态**。多个类可以实现同一个接口。作为调用者的对象并不关心它在与谁交谈，只要遵循这个契约就行。例如，Web容器可以使用实现了Servlet接口的任何组件。

## 关注点分离和内聚

我们很清楚，如果软件组件功能专一，组件将更容易创建、维护和重用。分离关注点有一个很自然的附加结果，这就是内聚度会随之增加。所谓内聚，就是设计类来完成某个任务或用途时“凝聚”程度如何。

## 隐藏复杂性

隐藏复杂性通常与关注点分离密不可分。例如，如果你的系统需要与一个查找服务通信，最好将这个操作的复杂性隐藏在某个组件中，并让所有其他需要访问查找服务的组件都能使用这个专用组件。通过这种方法，就能使所涉及的所有系统组件都得到简化。

## 更多设计原则……

### 松耦合

究其本质，面向对象系统就是由相互通信的对象组成。如果遵循接口来编写代码，一个类与另一个类通信时，就不用对另一个类了解太多。两个类相互了解得越少，它们相互之间的耦合就越松。类A想使用类B中的方法时，一种常见的做法是在二者之间创建一个接口。一旦类B实现这个接口。类A就可以通过此接口使用类B。这很有用，因为以后你还可以使用一个更新的类B，甚至一个完全不同的类，只要它遵循接口的契约就可以。

### 远程代理

如今，如果一个Web网站要扩展，采用的手段往往是集成更多的服务器，而不是升级一个大型服务器。这就带来一个结果，Java对象会位于不同的主机和各自不同的堆中，而且必须相互通信。

基于接口的强大能力，对“客户”对象（可能是一个远程对象）来说，远程代理对象在本地（之所以说它是远程代理，这是因为相对于所代理的“服务”对象它是远程的）。客户对象与这个代理通信，代理则处理与具体“服务”对象通信的所有网络复杂性。对客户对象来说，它只是在与一个本地对象通信。

### 增强声明性控制

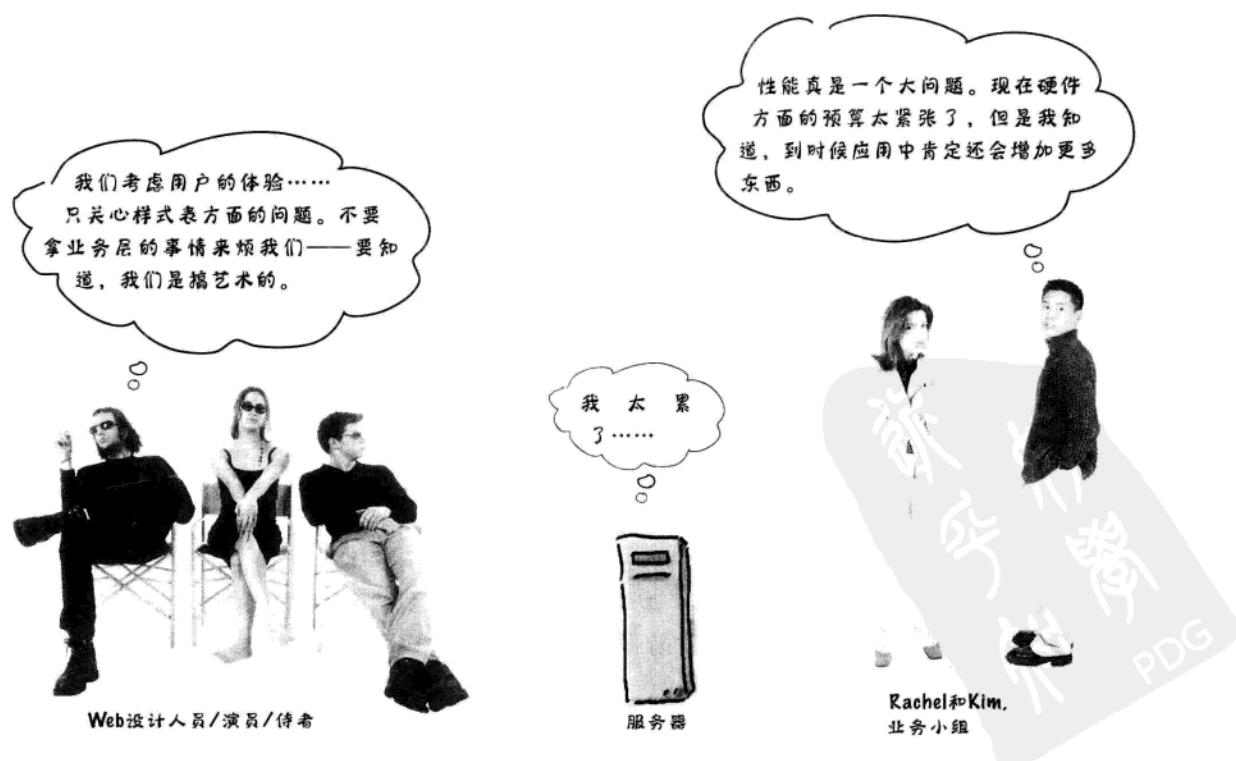
以声明方式控制应用，这是J2EE容器的一个强大特性。通常，这种声明式控制使用应用的部署描述文件(或DD)来实现。通过修改DD，我们就能改变系统行为而无需修改代码。DD是一个XML文件，可以由非程序员维护和更新。编写Web应用时如果能更充分地利用DD的作用，代码就能更抽象，更通用。

## 支持远程模型组件的模式

我们已经从理论上讨论了J2EE模式对于简化复杂的Web应用有何帮助，还介绍了J2EE模式底层的软件设计原则。在此基础上，下面更具体一些，我们来看一些比较简单的J2EE模式。以下讨论的这3种模式有一个共同的目标，都是让远程模型组件更可管理。

### 并于扩展啤酒应用的故事

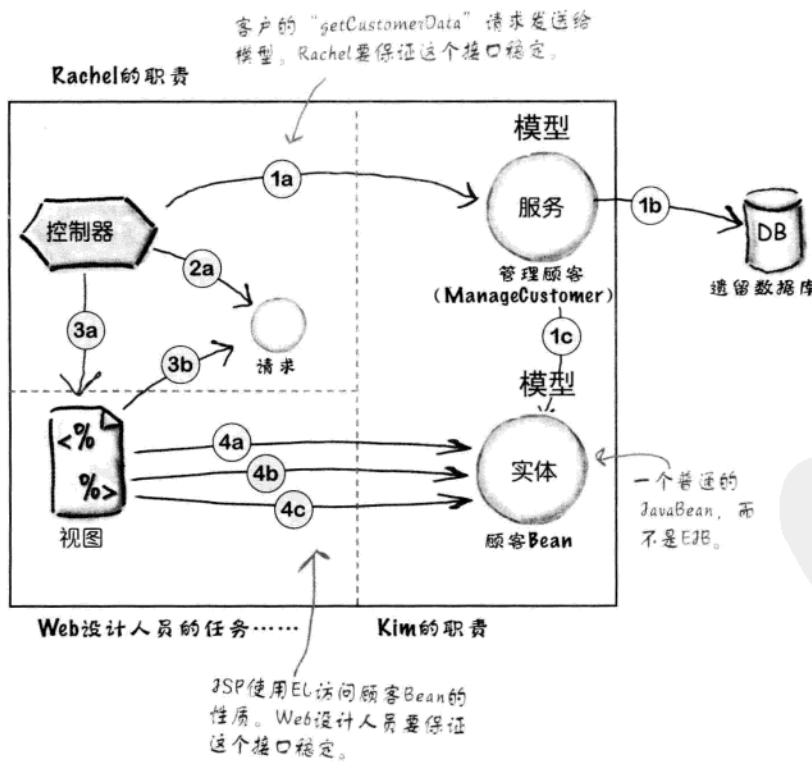
从前，有一个很小的.com公司，这个公司有一个网站，能为喜欢啤酒的人提供家庭调酒配方、建议、配料和有关服务。作为一个小公司（但野心勃勃），只有一个生产服务器支持网站，不过他们建立了两个独立的软件开发小组来扩展应用。第一个小组称作“Web设计人员”，他们强调的是系统的视图组件。第二个小组称为“业务小组”，关注控制器组件（这是Rachel的任务）和模型组件（这是Kim关心的领域）。



# MVC组件都在同一个JVM中运行时，业务小组如何支持Web设计人员

只要业务小组的人保证模型组件的接口是一致的，那就皆大欢喜了。在他们的设计中，有两个关键的接口点，首先是控制器第一次与模型组件交互时（下面的第1步和第2步），然后是JSP视图与所需的bean交互时（下面的第3步和第4步）。

为客户获得顾客数据……



**1** 接收到有关顾客信息的请求时，控制器调用ManageCustomer服务组件（模型）。这个服务组件对遗留数据库做一个JDBC调用，然后创建一个顾客bean（这不是EJB，而只是一个普通的JavaBean），其中填入了数据库的顾客数据。

**2** 控制器将顾客bean的引用作为一个属性增加到请求对象。

**3** 控制器转发到视图JSP。这个JSP从请求对象得到顾客bean的引用。

**4** 视图JSP使用EL得到为满足最初请求所需要的顾客Bean的性质。

# 如何处理远程对象？

所有Web应用组件（模型、视图、控制器）都在同一个服务器上，而且都在同一个JVM中运行，这种情况下问题相当简单。这只是普通的Java开发，先得到一个引用，然后调用一个方法而已。但是Kim和Rachel现在必须搞清楚，如果他们的模型组件对于Web应用来说是远程组件该怎么办。

## JNDI和RMI速览

对象跨网络通信时存在一些问题，Java和J2EE提供了一些机制，可以解决其中最常见的两个难题：查找远程对象，以及处理本地和远程对象之间的底层网络I/O通信。换句话说，如何找到远程对象，以及如何调用其方法。

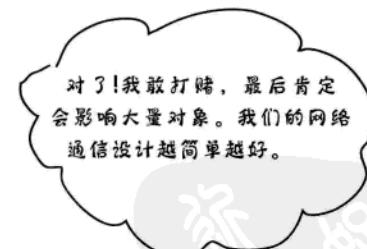
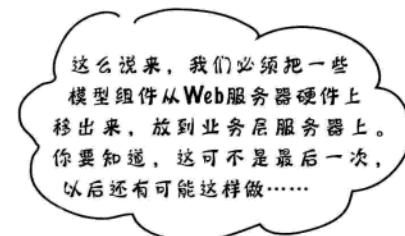
### JNDI核心手册

JNDI代表Java命名和目录接口（Java Naming and Directory Interface），这是一个访问命名和目录服务的API。基于JNDI，可以在网络中的一个集中位置上完成查找。如果你有一些对象，而且希望网络上的其他程序找到并访问这些对象，就要向JNDI注册这些对象。其他程序想使用你的对象时，则可以使用JNDI来查找。

有了JNDI，就能更容易地在网络上重新放置组件。一旦重新放置了一个组件，你要做的只是把这个新位置告诉JNDI。这样一来，其他客户组件只需知道如何找到JNDI，而无需知道向JNDI注册的对象具体在什么位置。

### RMI核心手册

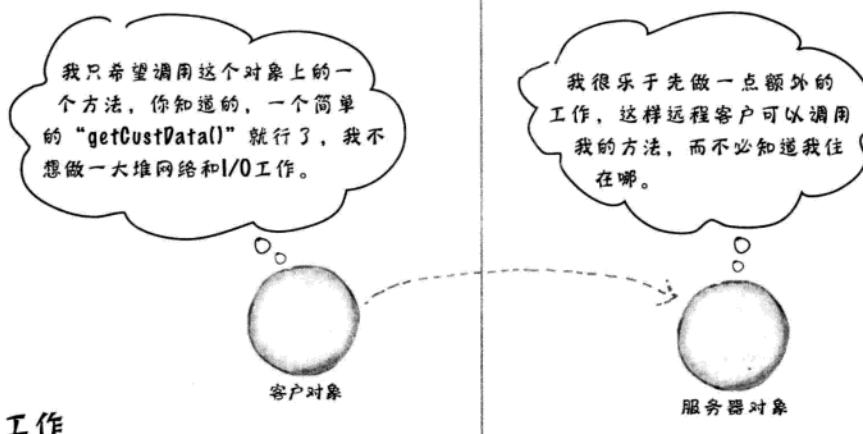
RMI代表远程方法调用（Remote Method Invocation），利用这种机制，获得对象并跨网络通信的过程就能大大简化。如果你有点忘了，可以翻到下一页，我们会做一个简单的回顾。为什么这里要考虑RMI？因为它有助于我们理解和掌握后面要讲到的两个J2EE设计模式。



# RMI让一切更简单

你希望对象能跨网络通信。换句话说，希望一个JVM中的对象调用一个远程对象上的方法（也就是说，这个远程对象在另一个JVM中），但是，你想假装成好像在调用本地对象的方法一样。这正是RMI的作用，利用RMI，你（几乎）就能假装成正在完成一个常规的普通本地方法调用。

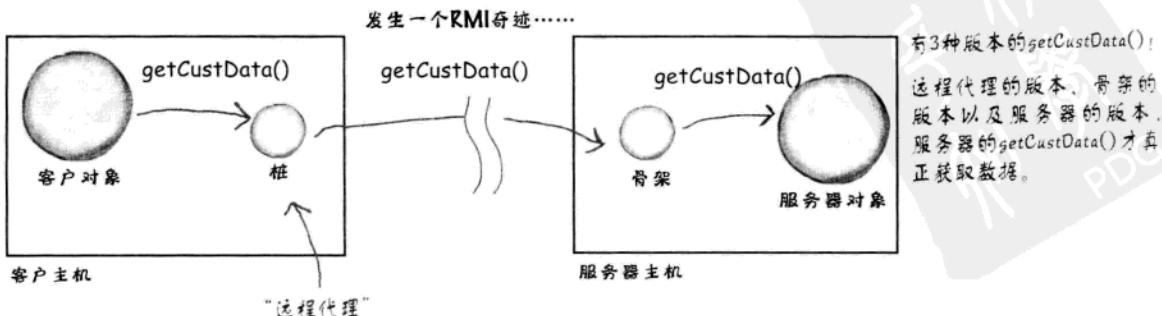
我们希望……



## RMI如何工作

假设你是“业务小组”中的一员，你希望远程客户能够使用某个对象。通过RMI，你创建了一个代理，并把这个对象注册到某种注册表。如果客户想调用你的方法，都能在这个注册表上进行查找，并得到远程代理的一个副本。然后，客户对这个远程代理做调用，**假装它就是实际的服务器对象**。远程代理（称为桩）处理所有通信细节，如套接字、I/O流、TCP/IP、对方法参数和返回值的串行化和逆串行化、异常处理等。

哦，顺便说一句，服务器端通常也有一个代理（一般称为“骨架”），它会在远程（服务）对象所在的服务器端完成类似的通信事务。



# RMI的更多内容

虽然我们不提供一个完整的RMI教程（注1），不过下面还是再来看一些高层的RMI主题，以免我们的看法出现分歧。具体地，我们将分别分析使用RMI的服务器端和客户端。

## 服务器端RMI的4大步骤：

（如何启用在服务器上运行的一个远程模型服务）

- ① 创建一个远程接口。`getCustData()`等方法的签名就放在这里。桩（代理）和具体的模型服务（远程对象）要实现这个接口。
- ② 创建远程实现，换句话说，就是创建具体的模型对象（位于模型服务器）。这一步还包括向一个已知注册服务（如JNDI或RMI注册表）注册模型。
- ③ 生成桩，还可能要生成骨架。RMI提供了一个名为rmic的编译器，会为你创建代理。
- ④ 启动/运行模型服务（它会自行向注册表注册，并等待远程客户的调用）。

## 使用和不使用RMI的客户端

对使用RMI的客户和未使用RMI的客户做一个比较，下面列出了它们的伪代码。

### 不用RMI的客户

```
public void goClient() {
    try {
        // 得到一个新的Socket
        // 得到一个OutputStream
        // 把它链到一个ObjectOutputStream
        // 发送opcode & op参数
        // 刷新输出OS(输出流)
        // 得到InputStream
        // 把它链到一个ObjectInputStream
        // 读取返回值和/或
        // 处理异常
        // 完成关闭工作
    } // 捕获和处理远程异常
}
```

### 使用RMI的客户

```
public void goClient() {
    try {
        // 查找远程对象(桩)
        // 调用远程对象的方法
    } // 捕获和处理远程异常
}
```

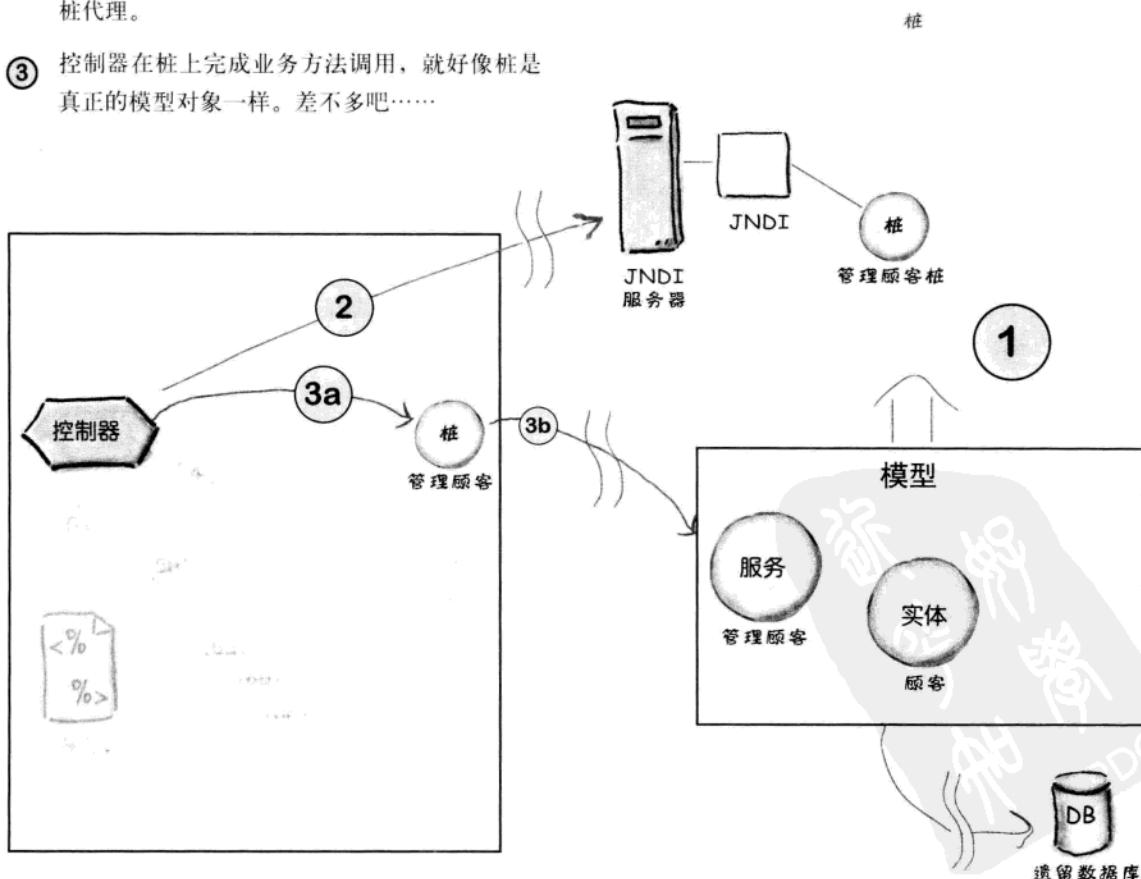
**注1：**如果你确实不熟悉RMI，赶快到附近的书店去挑一本（但不要买）《Head First Java》，读读有关RMI的部分。然后把书放回书架，不过要正面朝前，放在其他同类书前面。一定要把封面上的灰尘掸干净，另外别不小心洒上咖啡了。

# 为控制器增加RMI和JNDI

下面我们主要来看怎么让Rachel的工作尽可能的简单。换句话说，增加JNDI和RMI对控制器有什么影响？

## 使用远程对象的3个步骤：

- ① Kim（也就是负责模型的人）向JNDI服务注册了他的模型组件。
- ② Rachel的控制器得到一个请求，控制器代码完成一个JNDI查找，得到Kim的远程模型服务的桩代理。
- ③ 控制器在桩上完成业务方法调用，就好像桩是真正的模型对象一样。差不多吧……



当然，这些方法调用与使用本地模型时所用的方法调用确实很像，但是，我还是得修改控制器代码，要把整个JNDI查找都放在控制器代码里。我希望不论模型是本地的还是远程的，我都能使用同一个控制器。



## 怎样改进这个设计？

1. 这个设计有什么问题（至少列出两点）？
2. 如何修改这个设计来处理这些问题？

问题：

解决方案：

# 来一个“中间”对象如何？

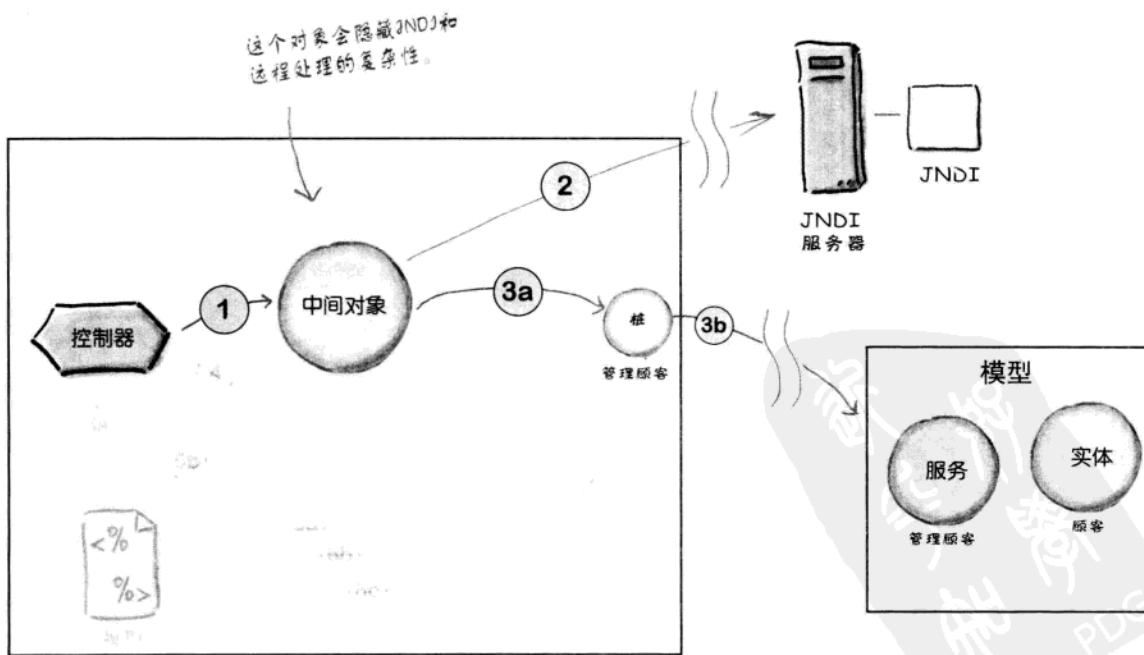
针对上一个练习留出的设计问题，常用的解决方案是创建一个新对象：一个简单的“中间”对象，控制器与这个中间对象对话，而不是直接与模型对象通信，这样控制器就无需考虑模型对象是否为远程。

## 问题1：隐藏复杂的JNDI查找。

如果Rachel的控制器让一个“中间”对象处理JNDI查找，控制器代码就能更简单，而且不必了解在哪里（以及如何）查找模型。

## 问题2：隐藏“与远程有关的复杂性”。

如果“中间”对象可以处理与桩的通信，Rachel的控制器中就不用考虑远程问题了，这也包括远程异常。



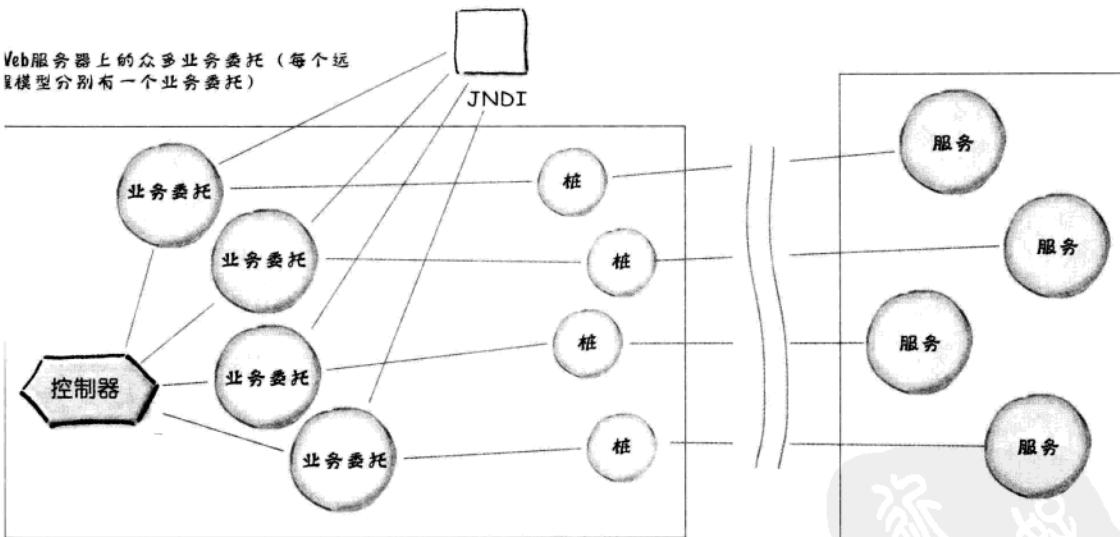
# “中间”对象是一个业务委托

面来看一个典型业务委托的伪代码，并分析如何在Web容器中部署业务委托。

注意Web层可能有很多业务委托。

## 业务委托的伪代码

```
// 得到请求，并完成一个JNDI查找  
// 得到一个桩  
  
// 调用业务方法  
// 处理并抽出所有远程异常  
// 把返回值发送给控制器
```



## Sharpen your pencil

唉呀，警告：有重复代码。

(说明哪些地方存在重复代码，  
怎样解决这个问题。)

## 用服务定位器简化业务委托

除非你的业务委托使用了一个服务定位器，否则处理查找服务时就会存在重复代码。

要实现一个服务定位器，应当取出完成JNDI查找的所有逻辑，把这些逻辑从多个业务委托中移出，放在一个服务定位器中。

在J2EE应用中，通常可能有多个组件使用相同的JNDI服务。有些复杂的应用可能会为Web服务端点使用多个不同的注册表（如JNDI和UDDI），但是单个组件一般只需要访问一个注册表。一般地，一个服务定位器就支持一个特定的注册表。

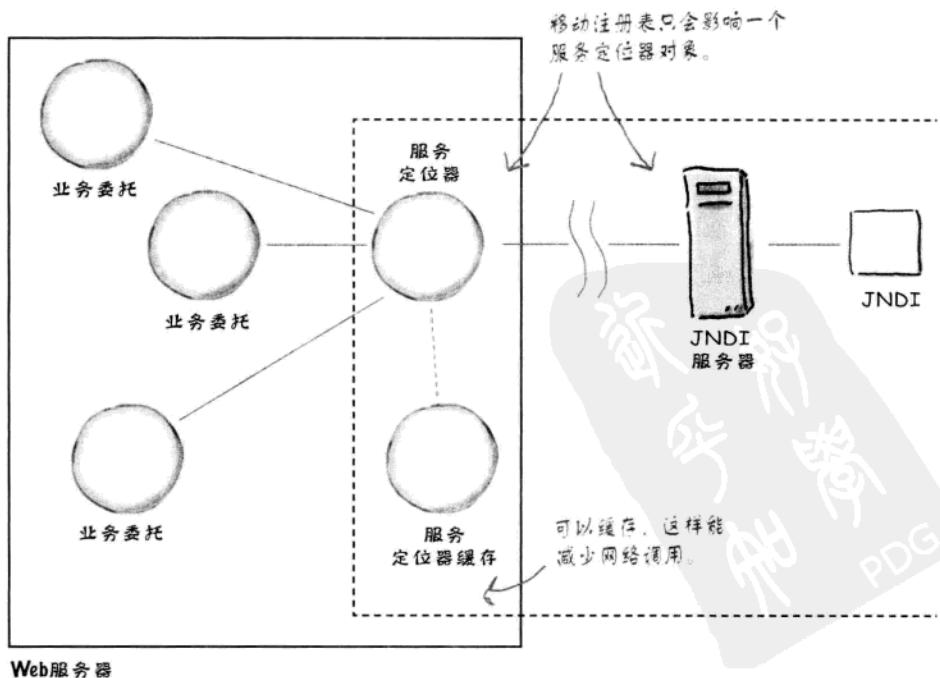
让业务委托对象只处理业务方法，而不要同时处理注册查找，这样就能提高业务委托的内聚度。

### 服务定位器的伪代码

```
// 得到一个InitialContext对象  
// 完成远程查找  
// 处理远程问题  
// 还可以缓存引用
```

可以提高所有这些业务委托的内聚度。

现在得到桩的工作是由服务定位器来处理的。所有业务委托要做的只是处理桩上的业务方法。



# there are no Dumb Questions

**问：**前面的讨论都假设使用了RMI，如果我们公司用的是CORBA呢？

**答：**我们讨论的所有模式基本上都不依赖于J2EE技术。不可否认，在J2EE中实现是最容易的，但是在其他情况下这些模式也同样适用。

**问：**JNDI也是这样吗？

**答：**嗯，除了JNDI——RMI和Jini外，还有另外一些与Java相关的注册表。在这三者当中，JNDI可能是大多数Web应用的最佳选择，它不仅简单，而且功能很强大（不过，我们个人更偏爱Jini，希望它在分布式领域中能有一席之地）。你可能还会使用非Java注册表，如UDDI。当然不管怎样，这些模式都适用，尽管代码可能有变化。

**问：**看上去这些模式都是在体系结构中增加一个新的对象层。为什么这种方法这么常用？

**答：**你说得没错，这确实是许多模式的一个共同点。假设你有一个很好的设计，想想看这样做对软件设计会有什么好处……

**问：**嗯，是不是内聚……

**答：**对！业务委托和服务定位器都能增强所支持对象的内聚度。还有一个重要原因，这样还有助于增加网络透明性。通过增加一层，通常能使现有对象不必考虑网络。当然，与内聚紧密相关的还有关注点分离。

**问：**关注点分离能给我带来什么好处……

**答：**就以服务定位器为例子来说吧。如果你的注册表得到一个新的网络地址和/或注册接口，相对于修改所有业务委托来说，只修改一个服务定位器显然容易得多。一般地，关注点分离能提供很大的灵活性和可维护性。

**问：**在你先前的例子中，先是建立本地POJO，再把这些对象设计为远程对象。但往往要把现有的EJB集成到Web应用中，这种情况更常见吧？

**答：**所谓的POJO，当然你认为这就是“普通的Java对象”（“Plain Old Java Objects”）。没错，你可能要把EJB集成到应用中，实际上，使用这两种模式还有一个原因……你的控制器（视图）不用关心模型是一个本地JavaBean、一个远程POJO，还是一个企业JavaBean（EJB）。如果不使用服务定位器或业务委托，不同的模型就会带来很大的差别，企业bean和普通的远程对象所用的查找代码根本不一样！

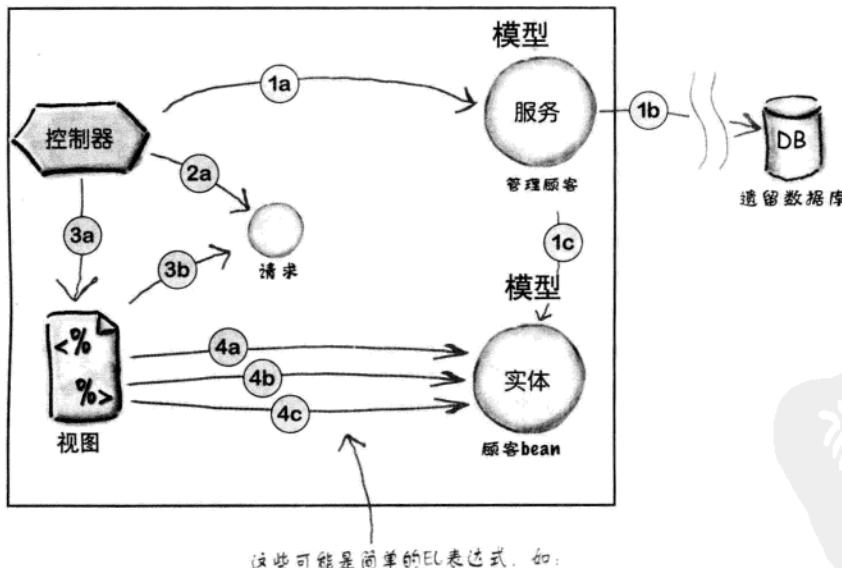
但是，倘若使用了这些模式，就能对如何（以及在哪里）发现和使用模型完成封装，这样控制器就无需知道这些细节。所以，业务小组的人在业务层上做了修改或移动时，你根本不用修改控制器代码。只需更新服务定位器就够了（也可能要修改业务委托）。

# 简化Web设计人员的JSP， 避开远程模型复杂性

通过使用业务委托和服务定位器模式，Rachel的控制器可以避开远程模型组件的复杂性。下面来看看对于Web设计人员的JSP如何做到这一点。

先来复习一下原来非远程的方式，也就是JSP使用EL从本地模型得到信息。

这个图与这一章前面的图很相似。JSP从请求对象得到bean引用（第3步），然后在bean上调用获取方法（第4步）。



**1** 接收到有关顾客信息的请求时，控制器调用Manage Customer模型组件。这个模型组件对遗留数据库做一个远程调用，然后创建一个顾客bean，其中填入了数据库中的顾客数据。

**2** 控制器将顾客bean的引用作为一个属性增加到请求。

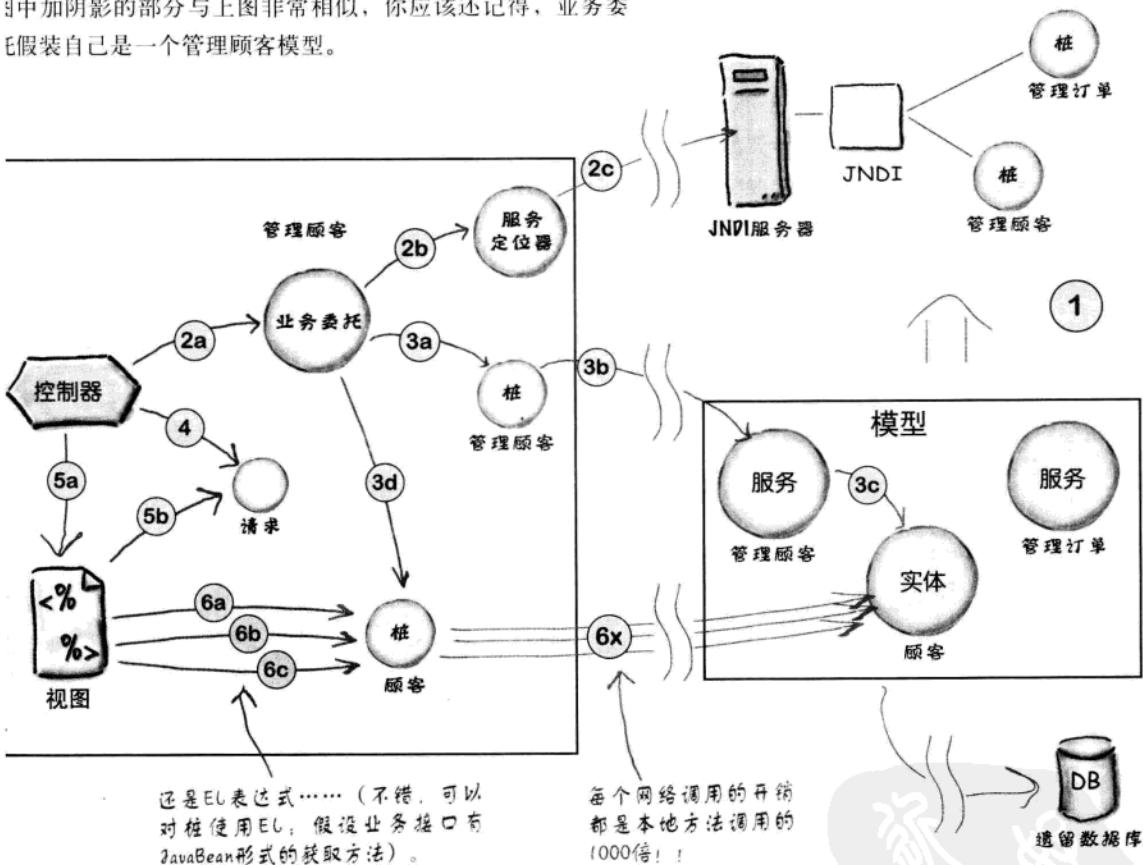
**3** 控制器转发到视图JSP。这个JSP从请求对象得到顾客bean的一个引用。

**4** 视图JSP使用EL得到为满足最初请求所需的顾客bean性质。

# 本地模型图与远程模型图的比较

别害怕！

图中加阴影的部分与上图非常相似，你应该还记得，业务委托假装自己是一个管理顾客模型。



复习这6个步骤：

- 1 向JNDI注册服务。
- 2 使用业务委托和服务定位器从JNDI得到管理顾客桩。
- 3 使用业务委托和桩来得到“顾客Bean”，在这里，“顾客”也是一个桩。把这个桩的引用返回给控制器。

4 将顾客桩引用增加到请求。

5 控制器转发到视图JSP。这个JSP从请求对象得到顾客bean（桩）的一个引用。

6 视图JSP使用EL得到为满足最初请求所需的顾客bean性质。

**重要提示：**JSP每次调用一个获取方法时，顾客桩都会完成一个网络调用。

## 有好消息，也有坏消息……

前面的体系结构成功地对控制器和JSP隐藏了复杂性，而且很好地利用了业务委托和服务定位器模式。

### 不好的是：

JSP获取数据时，存在两个问题，这两个问题都与一个事实有关，即JSP处理的bean实际上是一个远程对象的桩。

1. 所有这些细粒度的网络调用都会大大影响性能。可以想想看。每个EL表达式都会触发一个远程方法调用。这不仅是一个带宽/延迟问题，所有这些调用还可能导致服务器出问题。每个调用可能会在服务器上导致一个单独的事务和数据库加载（甚至还会导致数据库存储！）。

2. 如果远程服务器崩溃，JSP不适合处理可能由此产生的异常。

与其让JSP与一个桩通信，为什么不让JSP与一个普通的bean直接对话呢？

**问：** 如果你希望JSP与一个JavaBean直接对话，这个bean从哪里来呢？

**答：** 嗯，原先都来自本地模型/服务对象，所以，为什么不能来自远程模型/服务对象呢？

**问：** 那你怎么跨网络得到一个bean？

**答：** 嘿，只要它是可串行化的，利用RMI就完全可以跨网络发送对象。

**问：** 这又有什么好处？

**答：** 首先，我们可以完成一个大规模的网络调用，而不是许多小调用。其次，由于JSP与一个本地对象通信，所以不用担心远程异常！

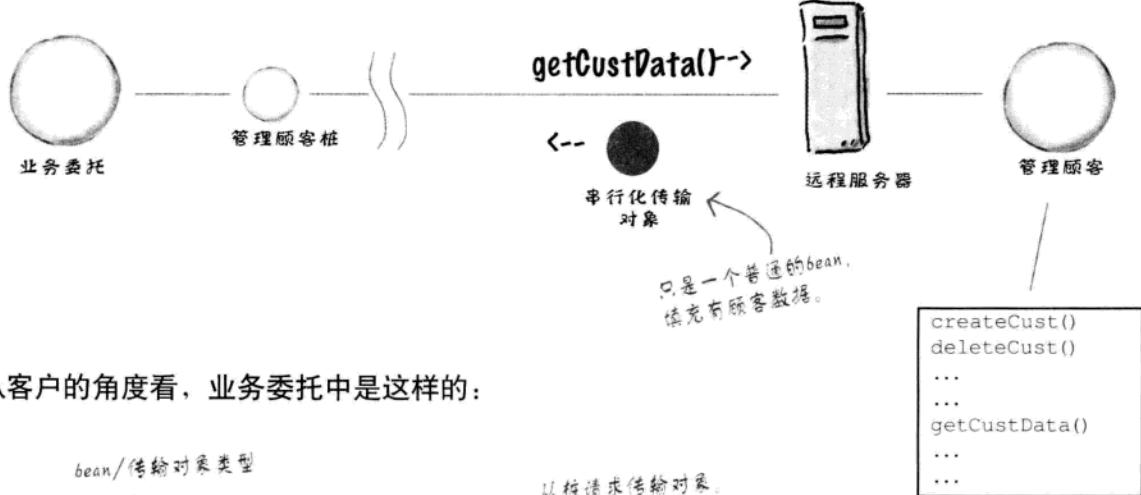
**问：** 先等等……我发现这里有一个小问题，也可能是一个严重的问题，如果你在客户端使用一个bean，发送这个bean时它的数据难道不会过时吗？

**答：** 是的，你说得对，这里需要权衡：到底是性能重要，还是更强调数据必须最新。你要根据自己的需求来做出决定。如果视图组件使用的数据必须完全反映数据库任何时刻的当前状态，就需要一个远程引用。例如，如果你对顾客做了3个调用：getName()、getAddress()和getPhone()，在调用getName()和getAddress()之间顾客的电话号码可能会改变，为此需要（通过远程对象）回过头去访问数据库，但是这个信息可能不会太快改变，如果只是考虑到存在这种情况就总是去访问数据库，那就不太值了。

另一方面，在一个高度动态的环境中，假设一个顾客全天候（24/7）都在交易，就必须显示最新的信息。倘若只是为客户发回一个JavaBean，这意味着视图会得到填bean时数据库的一个快照，但是由于bean并没有与数据库连接，所以数据会立即过时。

## 该介绍传输对象了吧？

可能会要求一个业务服务通过一个很大的粗粒度消息来发送或接收全部或大部分数据，为此，这个服务一般会在API中提供这个特性。通常，业务服务会创建一个可串行化的Java对象，其中包含大量实例变量。Sun把这个对象称为传输对象。除此以外，还有一个称为数据传输对象的模式。你应该能想到，“传输对象”和“数据传输对象”都是一样的（我们是这么认为的）。



从客户的角度看，业务委托中是这样的：

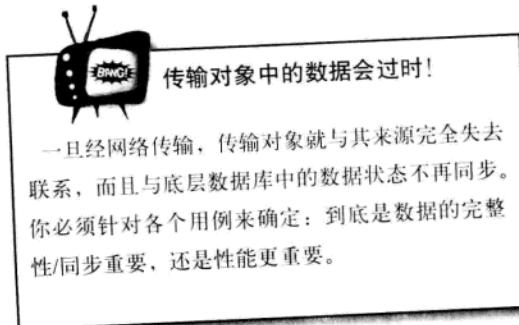
```

try {
    Customer c = custStub.getCustomerData(custID);
} catch (RemoteException re) {
    throw new CustomerException();
}
  
```

注释：

- 从客户角度看：从桩请求传输对象。
- 从远程服务器角度看：捕获远程异常，并包装在一个更高层异常中。

仅此而已。在底层，传输对象会得到串行化、传输，然后逆串行化到客户的本地JVM堆上。此时，它与其他本地bean没有什么不同。



## 服务定位器和业务委托都能简化模型组件

来听听两个高手的争论，服务定位器和业务委托到底哪一个模式更好。

服务  
定位器



业务委托



服务定位器模式更棒。首先，与业务委托不同，一个服务定位器实例可以支持整个应用层。

这当然不假，但是服务定位器只需要与一个远程实体通信，而业务委托必须处理多个实体对象。

服务定位器能提高网络调用的效率。一旦找到桩或服务桩，它能缓存这些桩的引用，这就能减少后续调用的网络开销。

有一点你忘了吧，服务定位器的任务要容易多了。业务委托的任务才重大呢！它必须与一个动态对象通信，这个对象的数据可能随时会改变。

任务重大？小小业务数据算得了什么。

与服务定位器相比，业务委托带给Web应用程序员的好处更多。

哈，也许程序员会满意，但是看起来这种简单的模式忘了一个问题，要知道，它往往在一个网络环境中运作。但它完全不考虑远程调用的开销，可能会全无约束地多次调用业务服务。

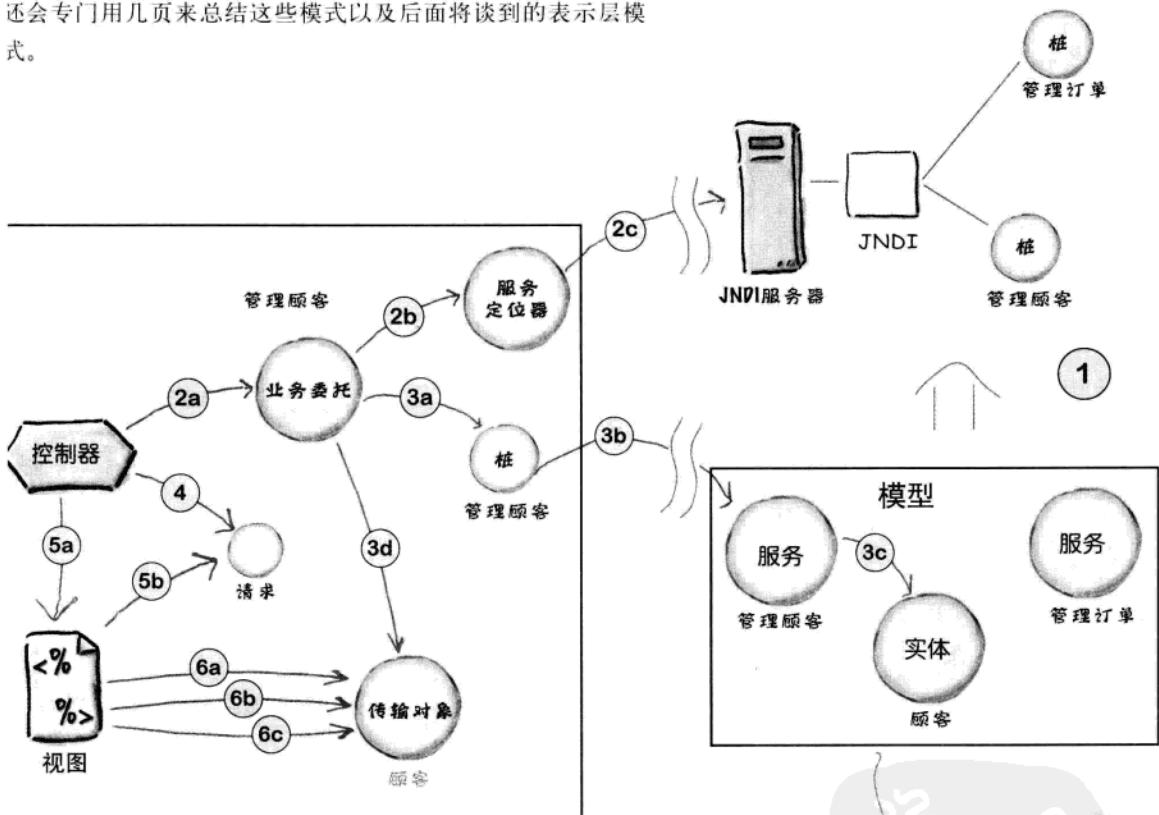
哈哈！业务委托可以与传输对象联手！通过协作，就能帮助程序员尽可能减少远程调用。

没错，因为你的模式能力太弱，所以需要帮助，这一点我们都明白。但是你和传输对象在一起又怎么样呢？还有新的问题在等着你……数据可能会过时，还有并发问题，这些你没忘吧？

不，我当然没忘记。但是即使出现这些问题，也能很好地得到解决。谁能随随便便就成功……J2EE世界里，没有彻头彻尾的好与差。

# 业务层模式：快速复习

我们对业务层模式的讨论做一个总结，下面的图显示了实际运用的业务委托、服务定位器和传输对象模式。在这一章的最后还会专门用几页来总结这些模式以及后面将谈到的表示层模式。



## 复习6大步骤：

- 1 向JNDI注册你的服务。
- 2 使用业务委托和服务定位器从JNDI得到管理顾客桩。
- 3 使用业务委托和桩得到“顾客Bean”，在这里会得到一个传输对象。把这个传输对象的引用返回给控制器。
- 4 把bean的引用增加到请求。
- 5 控制器转发到视图JSP。这个JSP从请求对象得到顾客传输对象bean的引用。
- 6 视图JSP使用EL得到为满足最初请求所需的顾客传输对象bean的性质。

# 再来看介绍过的第一种模式……MVC

碰巧的是，这本书里一直使用的MVC模式也会在考试中出现。像拦截过滤器模式一样，我们要介绍的最后两个模式也是表示层模式。首先，我们先来看讨论MVC时哪里还没有讲到。我们将由此引入Struts以及最后的前端控制器。

## 哪里还没有讲到……

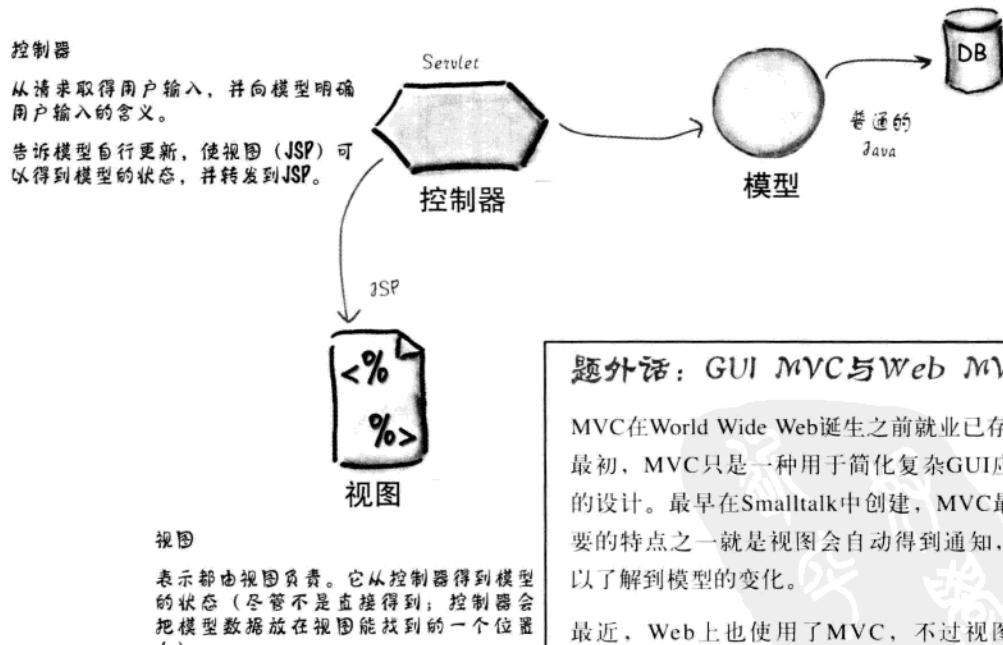
下面来简单复习一下第2章还有哪些内容没有讲到。

### 模型

保存真正的业务逻辑和状态。换句话说，它了解获取和更新状态的规则。

购物车的内容（和处理购物车的有关规则）都是MVC中模型的一部分。

应用中只有这部分会与数据库通信。



# 实际Web应用中的MVC

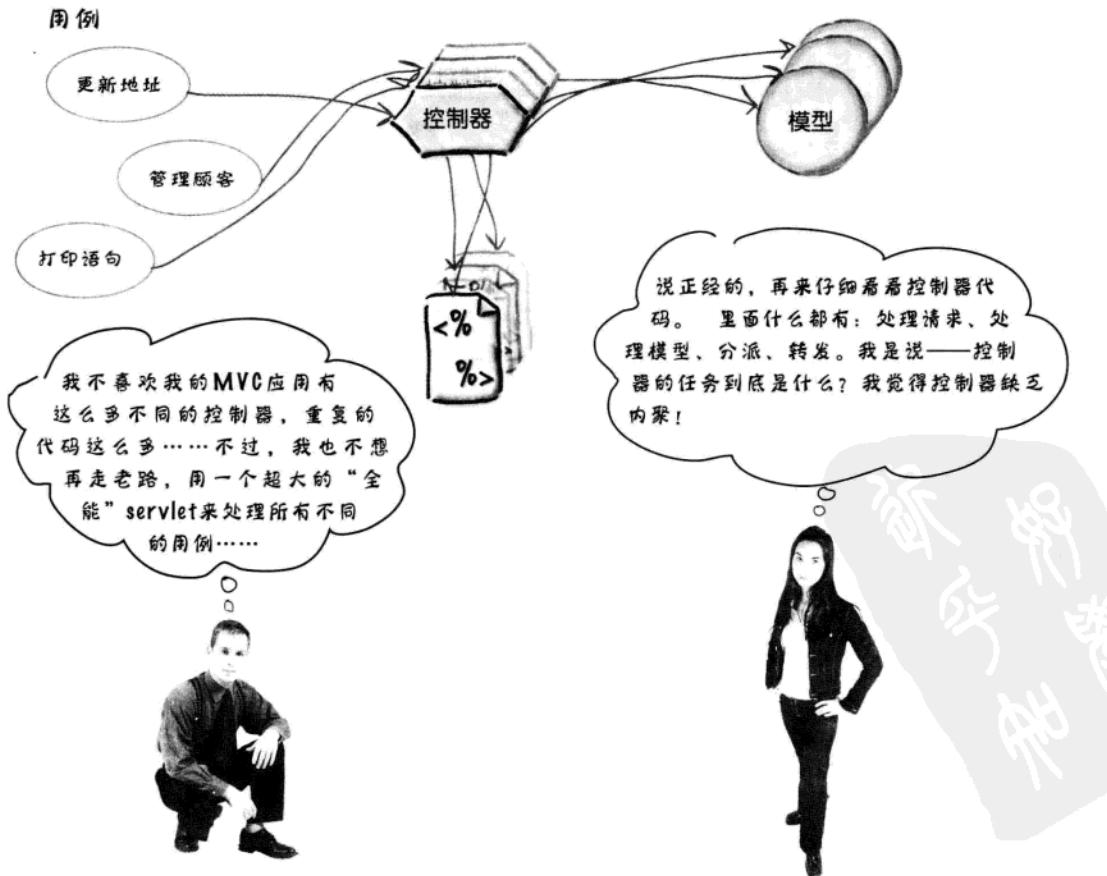
再回到第2章，我们留了一个“开动脑筋”练习，让你自己想想速配应用MVC体系结构还存在什么问题。下面来回顾一下哪些还没有讲到，并回答这么久以来一直让你耿耿于怀的问题：还有什么能比MVC更好？

每个浏览器用例都有一组相应的模型、视图和控制器组件。根据不同用例，这些组件可以采用多种不同的方式混合、匹配和重新组合。

在速配应用中，我们要处理的问题是：有许多专用的控制器，从面向对象角度看这好像很好，但是这样一来，应用的所有控制器中都会有重复的代码，而且可维护性和灵活性都不令人满意。

一个MVC应用可能有多个模型、视图和控制器。

用例



# 分析MVC控制器

下面来看控制器是不是确实存在这些问题。首先，先来回忆一下控制器servlet的任务：

## 通用MVC控制器的伪代码

```
public class ControllerServlet extends HttpServlet {
```

```
    public void doPost(HttpServletRequest request, HttpServletResponse response) {
```

**①** String c = req.getParameter("startDate");

// 对日期参数完成数据转换

// 验证日期在合法范围内

// 如果验证中发生错误，  
// 转发到硬编码的一个“重试” JSP

处理  
请求参数

**②** // 调用硬编码的模型组件

// 把模型结果增加到请求对象。  
// (可能是一个bean引用)

处理模型

**③** // 分派到视图JSP  
// (当然它也是硬编码的)

处理视图

```
}
```

```
}
```



Sharpen your pencil

这个组件违反了哪些原则？

列出这个伪代码违反的3个或更多软件设计原则。

# 改进MVC控制器

除了缺乏内聚，这个控制器与模型和视图组件紧密耦合。这里还存在重复代码问题。怎么解决呢？

## 控制器的3大任务

## 有更好的解决办法吗？

① 获得和处理请求参数	把这个任务交给另外一个单独的表单验证组件，由它获得表单参数、完成转换、进行验证、处理验证错误，并创建一个对象保存参数值
② 调用模型	嗯……我们不想把模型硬编码写到控制器里，所以可以用声明的方式来完成，在我们自己的定制部署描述文件中列出一些模型，控制器可以读取这些模型，并且根据请求来决定要使用哪些模型
③ 分派到视图	这个工作也采用声明方式来实现不是很好吗？这样一来，根据请求URL，控制器就能（从定制的部署描述文件）区分出要分派到哪个视图

## 新的控制器伪代码（而且更简短）

```
public class ControllerServlet extends HttpServlet {
    public void doPost(request, response) {
        // 以声明方式调用一个验证组件
        // (验证错误也由它处理！)

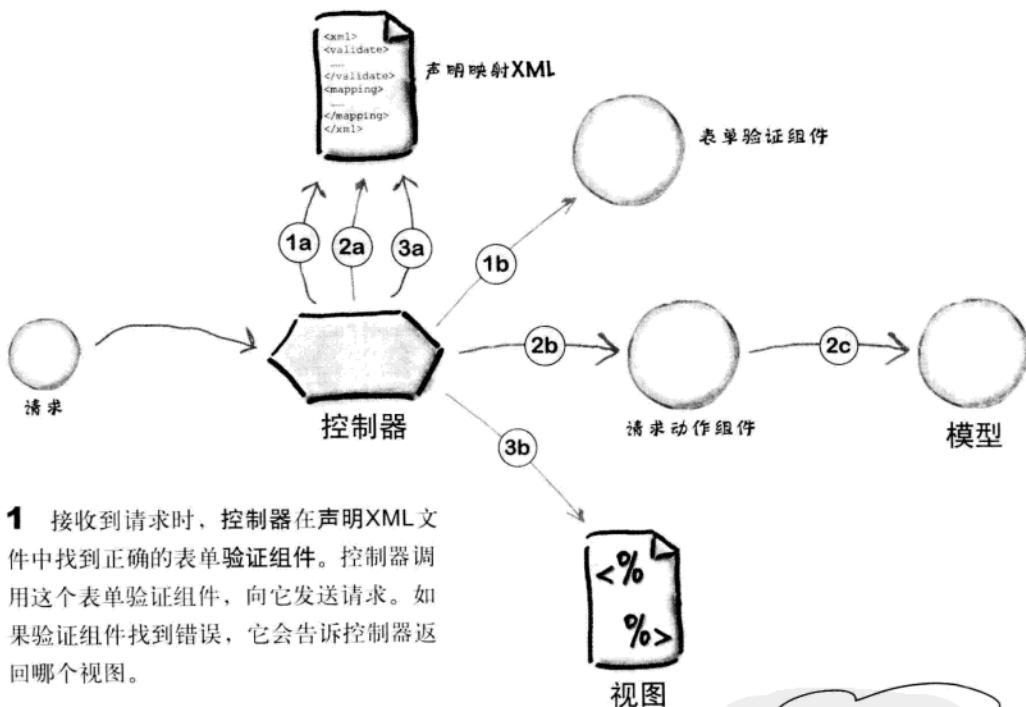
        // 以声明方式调用一个请求处理组件,
        // 来调用一个模型组件

        // 以声明方式分派到视图JSP
    }
}
```



# 设计我们的“超级”控制器

下面再画一个体系结构图，看一看这个控制器和相应的支持组件是什么样子。



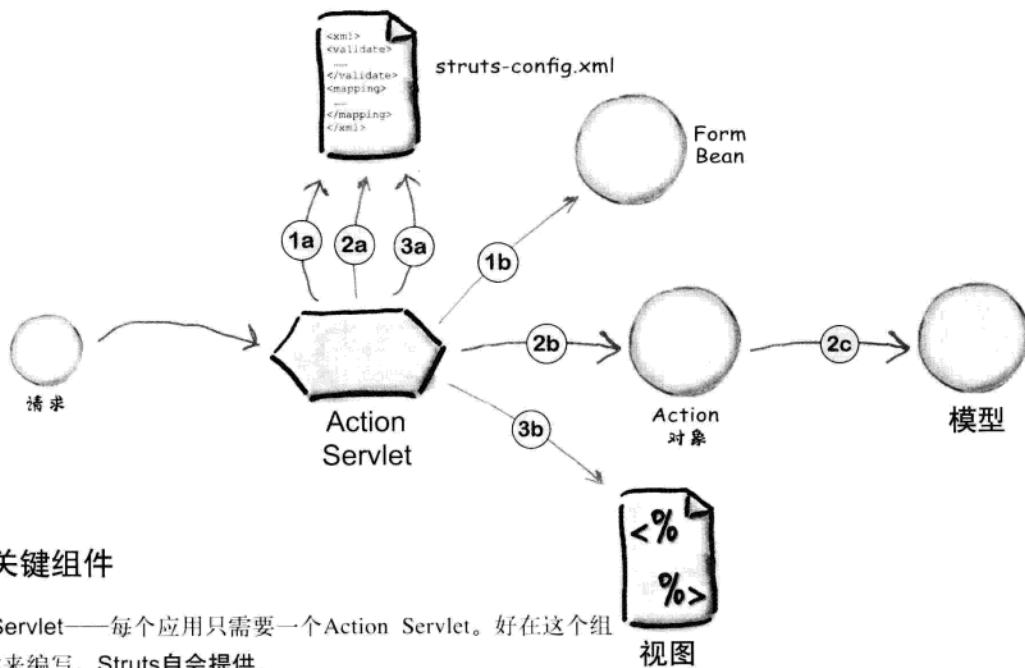
等一等……这个图  
好像似曾相识。你想伪装成  
**STRUTS!**



控制器

# 没错！实际上这正是Struts

虽然这只是一个概要介绍，我们省去了很多细节，不过这正是Struts框架的基本思想。下面来进一步看有关的细节，首先对组件统统改名……



## Struts关键组件

**Action Servlet**——每个应用只需要一个Action Servlet。好在这个组件无需你来编写，Struts自会提供。

**Form Bean**——对于应用需要处理的各个HTML表单，都要为它写这样一个Form Bean。它们都是Java bean，而且一旦Struts Action Servlet调用了表单bean上的设置方法（在bean中填入表单参数），它就会调用bean的validate()方法，所以完全可以把数据转换和错误处理逻辑放在这里。

**Action对象**——一般地，动作（action）映射到用例中的一个活动。它有一个类似回调的方法，名为execute()，可以在这个方法中获得验证表单参数，并调用模型组件。可以把Action对象认为是一种“轻量级的servlet”。

**struts-config.xml**——这是Struts特定的部署描述文件。你要在这个文件中定义以下映射：请求URL到Action、Action到Form bean，以及Action到视图。



# Struts是一个容器吗？

正式地讲，一般认为Struts是一个框架。

框架是一些接口和类的集合，这些接口和类设计为共同处理某种特定类型的问题。对于Struts，问题空间就是Web应用。框架的目标是“帮助程序员开发和维护复杂的应用”。

所以，Struts不是一个容器，但是在某些方面，它确实很像一个容器。

## Struts与servlet容器相似的5个方面

**1 声明方式：**都使用一个XML文件以声明方式配置应用。

**2 生命周期：**都为预定类型的对象提供了生命周期。

**3 回调：**都会完成主要生命周期方法的自动回调。

**4 API：**都为所支持的主要对象提供了API。

**5 应用控制：**都提供了一个可供应用运行的受控环境。它们是应用对外界的窗口。



Relax 考试不会考Struts的内容！

你确实要知道前端控制器的作用和功能（而且Struts实际上就是一个乔装改扮的前端控制器），但是你不会看到有关Struts框架的任何问题。所以，完全可以放心，不要求记住这里的每一个细节。

我觉得有些地方似曾相识……你说过Struts有“回调”方法，还有一个部署描述文件。那么，Struts是不是像一个“小容器”呢？



在Struts中，我被提升为“Action Servlet”。有时也有人把我叫做“前端控制器”。（顺便说一句，考试时会出现前端控制器。）



Action  
Servlet

# 前端控制器是什么？

哦，是这样。前端控制器也是一种J2EE模式，而且考试时会考。实际上，Struts只是使用前端控制器模式的一个有意思的例子。前端控制器模式的基本思想是：一个组件（通常是一个servlet，但也可能是JSP）作为Web应用表示层的一个控制点。采用前端控制器模式的话，应用的所有请求都会通过一个控制器，由它处理，并将请求分派到适当的地方。

在实际中，很少需要自行实现一个前端控制器。即使一个应用实现确实很简单，通常也还会包括另外一个J2EE模式，称为应用控制器。Struts包括一个类，名叫RequestProcessor，它会最终负责HTTP请求的处理。

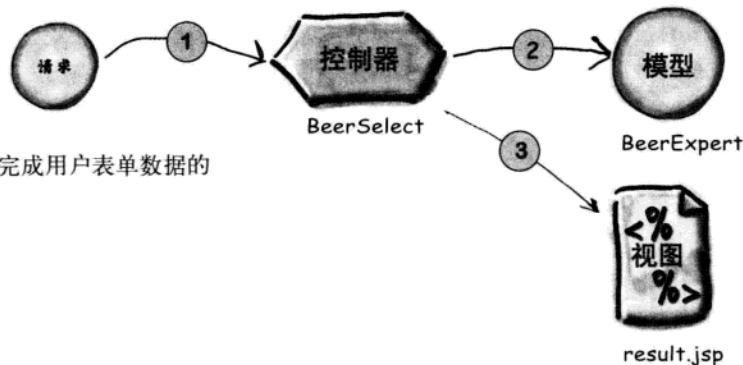
尽管考试中可能会有关于前端控制器模式的问题，但是不会考Struts，你只要记住Struts的好处，还有Struts只是一个前端控制器（具备其所有特点），这就足够了。

## Struts为前端控制器增加的8个特性

- 1 **声明式控制：**Struts允许在请求URL、验证对象、调用模型的对象以及视图之间建立声明方式的映射。
- 2 **自动请求分派：**Action.execute()方法返回一个指示性的ActionForward，它告诉Action-Servlet要分派到哪个视图。这样就在控制器和视图组件之间提供了另外一个抽象层（并使二者实现松耦合）。
- 3 **数据源：**Struts可以提供数据源管理。
- 4 **定制标记：**Struts提供了数十个定制标记。
- 5 **国际化支持：**错误类和定制标记都有国际化支持。
- 6 **声明式验证：**Struts提供了一个验证框架，这样就无需在表单bean中再编写验证方法。验证表单的规则可以在XML文件中配置，而且不必影响表单bean代码就能修改验证表单的规则。
- 7 **全局异常处理：**Struts提供了一种声明方式的错误处理机制，与DD中的<error-page>很类似。不过，对于Struts，异常可能是Action对象中应用代码特定的异常。
- 8 **插件：**Struts提供了一个PlugIn接口，它有两个方法：init()和destroy()。可以创建自己的插件来改进你的Struts应用，它们会为你管理这些插件。例如，验证器框架就使用插件初始化。

# 针对Struts重构啤酒应用

道理已经讲得够多了，下面来写一个Struts应用。首先，先来复习第3章的MVC啤酒应用。重构到Struts时，只有与MVC 控制器有关的代码需要修改（模型和视图丝毫不受影响）。



- 1 接收到请求时，控制器完成用户表单数据的验证。
- 2 控制器调用模型组件。
- 3 控制器转发到视图。

## MVC 控制器代码（取自第3章）

```
package com.example.web;
import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class BeerSelect extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {
        String c = request.getParameter("color"); ← 这里没有大量表单验证。
        BeerExpert be = new BeerExpert();
        List result = be.getBrands(c); ← 调用模型。
        request.setAttribute("styles", result);
        RequestDispatcher disp =
            request.getRequestDispatcher("result.jsp");
        disp.forward(request, response);
    }
}
```

# Struts啤酒应用体系结构

下面是啤酒应用体系结构，都在Struts中实现……



- 1** 接收到一个请求时，ActionServlet使用struts-config.xml文件找到正确的表单bean。ActionServlet调用表单bean的验证逻辑。如果表单bean找到错误，则填写一个ActionErrors对象。
- 2** 通过使用struts-config.xml文件，ActionServlet找到并调用Action对象，由这个对象调用模型，并向ActionServlet返回一个ActionForward对象。
- 3** 从struts-config.xml中抽出必要的映射，ActionServlet使用ActionForward对象分派至正确的视图组件。

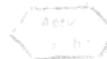
OK，视图在Struts Web应用中还是要修改的。一方面，Struts提供了一个标记库，其中有一个标记<html:errors/>，这个标记会显示表单bean验证错误。另外HTML标记库还提供了一些标记，它们会根据错误重新填写表单。

# 表单bean一览

记住，表单bean的任务是验证用户的表单参数。Struts的一大优点就是验证步骤直接放在了体系结构中。



BeerSelectForm



```
package com.example.web;

// Struts imports
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionErrors;

import javax.servlet.http.HttpServletRequest;

public class BeerSelectForm extends ActionForm {

    private String color;
    public void setColor(String color) {
        this.color = color;
    }
    public String getColor() {
        return color;
    }
    private static final String VALID_COLORS = "amber,dark,light,brown";
    public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if ( VALID_COLORS.indexOf(color) == -1 ) {
            errors.add("color", new ActionMessage("error.colorField.notValid"));
        }
        return errors;
    }
}
```

Form bean必须扩展  
ActionForm。

通常，Form bean要有所有表单参数的获取方法和  
设置方法。

ActionServlet调用  
validate()  
Struts提供了Action-  
Errors来管理验证错误。

ActionError构造函数取一个String参数，  
这是一个资源包的符号键，使用资源包  
有利于支持国际化。

# Action对象怎么做

Action对象实际上就是分派器。它由ActionServlet调用，ActionServlet会调用Action对象的execute()方法。



```

package com.example.web;

// Model imports
import com.example.model.*;
import java.util.*;

// Struts imports
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;

// Servlet imports
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class BeerSelectAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) {
        // 把表单转换为应用特定的表单
        BeerSelectForm myForm = (BeerSelectForm) form;

        // 处理业务逻辑
        BeerExpert be = new BeerExpert();
        List result = be.getBrands(myForm.getColor());

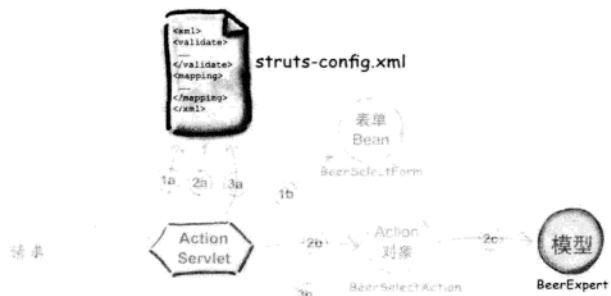
        // 转发到结果视图
        // (并把数据存储在请求作用域)
        request.setAttribute("styles", result);
        return mapping.findForward("show_results");
    }
}

```

控制器必须扩展Action类。  
从ActionServlet发出，所以可以返回正确的视图。  
允许访问经过验证的用户参数。  
将用户表单参数发送到模型组件。  
execute方法向ActionServlet返回一个ActionForward，它指示Struts分派到下一个正确的视图。这些将通过“转发”在struts-config.xml文件中声明。

# struts-config.xml: 集成在一起

struts-config.xml 文件类似于DD。实际上你也可以把它叫做其他名字，不过一般都会用 struts-config.xml 作为它的文件名。类似于部署描述文件，要在这个文件中声明和映射 Web 应用中的 Struts 组件。这种机制可以使你的应用做到更松的耦合。



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
    "http://struts.apache.org/dtds/struts-config_1_3.dtd">
```

```
<struts-config>
    <form-beans>
        <form-bean name="selectBeerForm"
            type="com.example.web.BeerSelectForm" />
    </form-beans>
```

← <form-bean> 元素声明表单 bean 对象的符号名和类。

```
    <action-mappings>
        <action path="/SelectBeer"
            type="com.example.web.BeerSelectAction"
            name="selectBeerForm" scope="request"
            validate="true" input="/form.jsp">
```

← <action> 元素将 URL 路径映射到 控制器类；注意，路径的 .do 扩展名不包括在 Struts 配置中。

```
            <forward name="show_results"
                path="/result.jsp" />
        </action>
    </action-mappings>
```

← <action> 还用于将一个表单 bean 与动作 (action) 相关联。这由表单 bean 符号名指定。Struts 会创建这个 bean，并保存在特定的作用域。如果进行验证，而且从验证方法返回了错误，input 属性会声明负责显示错误消息的视图；这通常就是提交这个动作 (action) 的表单。

← <forward> 元素在视图符号名（由 Action 对象使用）和视图组件物理路径之间创建一个映射。

```
<message-resources parameter="ApplicationResources" />
</struts-config>
```

# 在web.xml DD中指定Struts

对容器来说，ActionServlet只是一个servlet。所以必须声明ActionServlet，并确保将Web应用的所有请求都映射到这个servlet。

```

<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">

  <!-- Define the controller servlet -->
  <servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

    <!-- Name the struts configuration file -->      "config" 初始化参数告诉ActionServlet在
    <init-param>                                     哪里查找Struts配置文件。
      <param-name>config</param-name>                  ←
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>

    <!-- Guarantee that this servlet is loaded on startup. -->
    <load-on-startup>1</load-on-startup>             ActionServlet有一个复杂的init方法。
  </servlet>                                         最好启动时就加载这个servlet。

  <!-- The Struts controller mapping -->
  <servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!-- END: The Struts controller mapping -->

</web-app>

```

 应当将Struts DD命名为“struts-config.xml”

否则，在你的web.xml DD中就必须声明一个初始化参数“config”来定义Struts DD的名字。如果使用“struts-config.xml”作为DD名，Struts就会自动找到这个DD，而不需要初始化参数，不过在DD中声明初始化参数仍被认为是一种“好的做法”。

# 安装Struts，然后运行！

安装Struts很简单。

这一页上提到的链接和版本在写这本书的时候都是有效的。这对你来说可能没有什么帮助，在此只是想告诉你：我们不知道你看到这本书时的实际情况是什么，但是我们确实已经尽力了。

安装Struts只需7个简单的步骤：

- ① 打开浏览器，导航到：

<http://struts.apache.org/downloads.html>

- ② 在一般可用性（General Availability）列表中点击最新的Struts v1.3.\*链接。

- ③ 选择所需的JAR文件。最小的JAR是只包含库的Struts版本。

`struts-1.3.8-lib.zip`

- ④ 把zip文件下载到一个临时目录。

- ⑤ 将文件解压缩，它会解开为：

```
struts-1.3.8/
NOTICE.txt
lib/
    struts-core-1.3.8.jar
    struts-taglib-1.3.8.jar
    commons-beanutils-1.7.0.jar
    commons-digester.jar
    commons-chain-1.1.jar
```

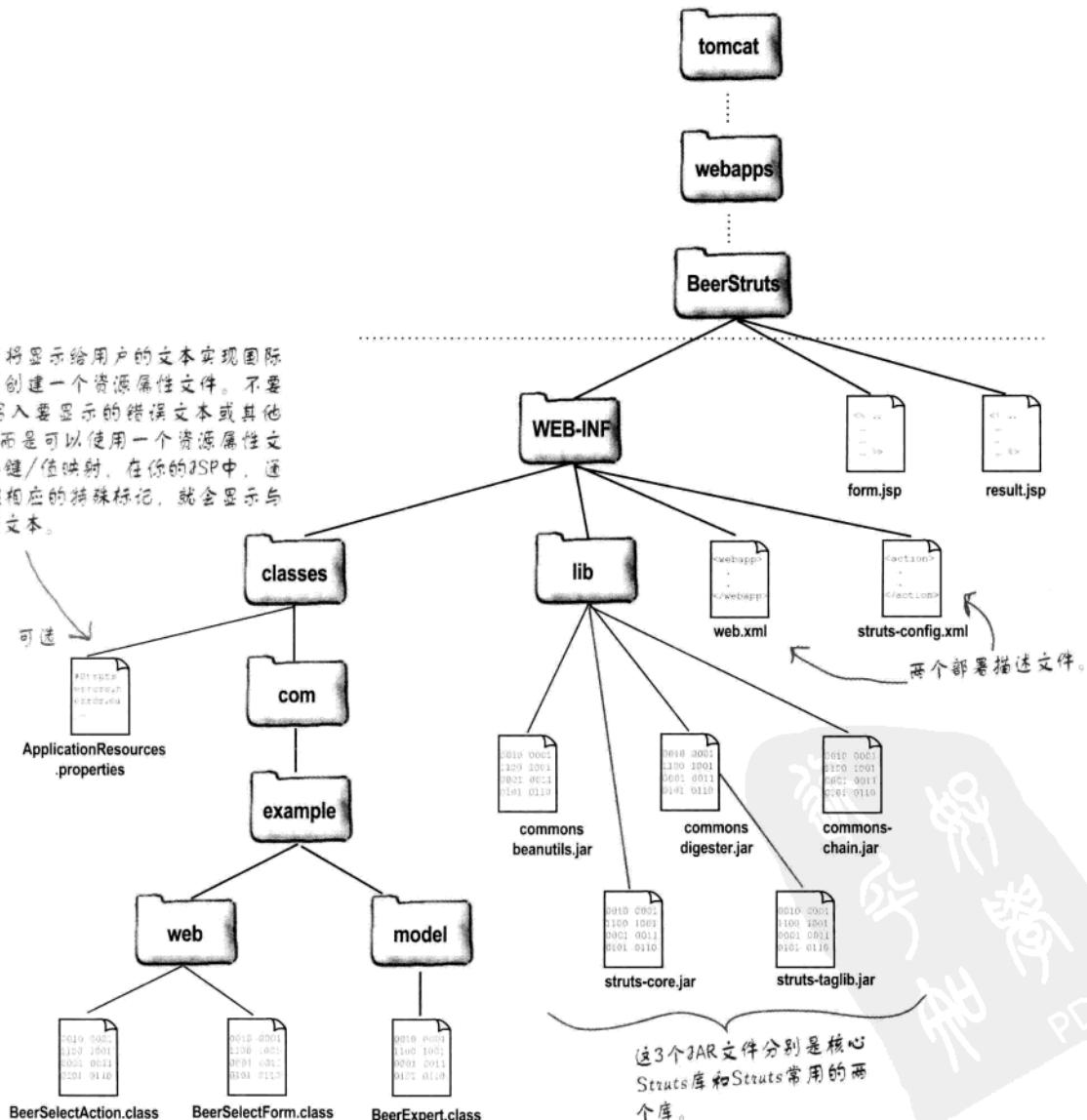
- ⑥ 把第5步中所列的文件复制到Web应用的WEB-INF/lib/目录。

- ⑦ 供参考：编译表单bean和动作对象时，确保你的类路径上有Struts核心JAR文件的一个副本。（记住，Action-Servlet前端控制器会自动创建）。

# 创建部署环境

创建这个目录结构来运行Struts版的啤酒应用。

如果希望将显示给用户的文本实现国际化，就要创建一个资源属性文件。不要直接编写入要显示的错误文本或其他string，而是可以使用一个资源属性文件来提供键/值映射，在你的JSP中，通过调用键相应的特殊标记，就会显示与映射的文本。

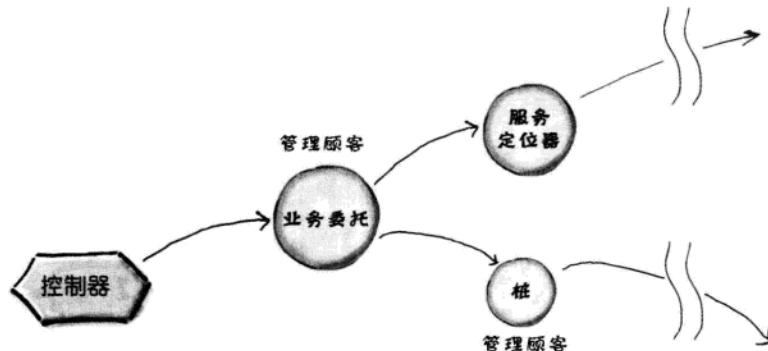


# SCWCD的模式复习

在最后两章我们介绍了许多模式。后面几页将对SC-WCD考试要考的有关模式细节做一个总结。

## 业务委托

使用业务委托模式，Web层控制器就不用考虑应用的模型组件是否是远程的。



### 业务委托特性

- 相当于一个代理，实现了远程服务的接口。
- 初始化与远程服务的通信。
- 处理通信细节和异常。
- 从控制器组件接收请求。
- 转换请求，并转发到业务服务（通过桩）。
- 转换响应，并返回给控制器组件。
- 处理远程组件查找和通信的有关细节，使控制器更为内聚。

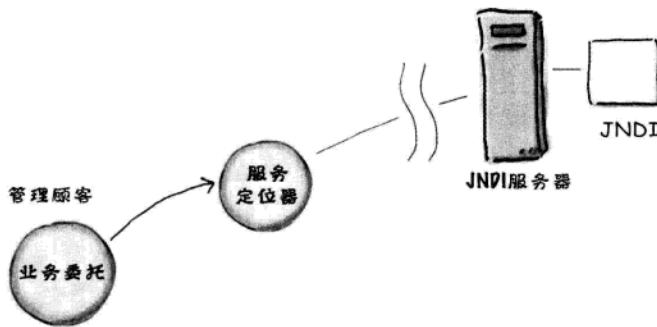


### 业务委托原则

- 业务委托建立在以下原则基础上：
  - 隐藏复杂性。
  - 根据接口编写代码。
  - 松耦合。
  - 关注点分离。
- 尽可能减少业务层改变对Web层的影响。
- 减少层间的耦合。
- 向应用增加一层，这会增加复杂性。
- 对业务委托的方法调用应当是粗粒度的，以减少网络流量。

# 服务定位器

使用服务定位器模式来完成注册表查找，使其他完成JNDI查找（或其他类型的注册表查找）的组件（如业务委托）得到简化。



## — 服务定位器特性

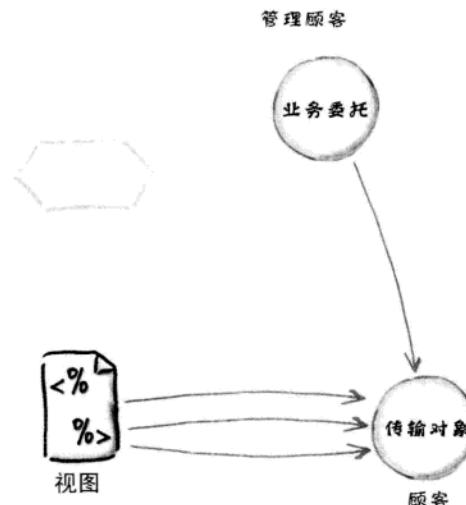
- 得到InitialContext对象。
- 完成注册表查找。
- 处理通信细节和异常。
- 通过缓存先前得到的引用，可以提高性能。
- 可以用于多种类型的注册表，如JNDI、RMI、UDDI和COS命名。

## 服务定位器原则

- 服务定位器建立在以下原则基础上：
  - 隐藏复杂性。
  - 关注点分离。
- 当远程组件改变位置或容器时，尽可能减少对Web层的影响。
- 减少层间的耦合。

# 传输对象

使用传输对象模式，通过提供粗粒度远程组件（通常是一个实体）的本地表示，尽量减少网络流量。



## 传输对象功能

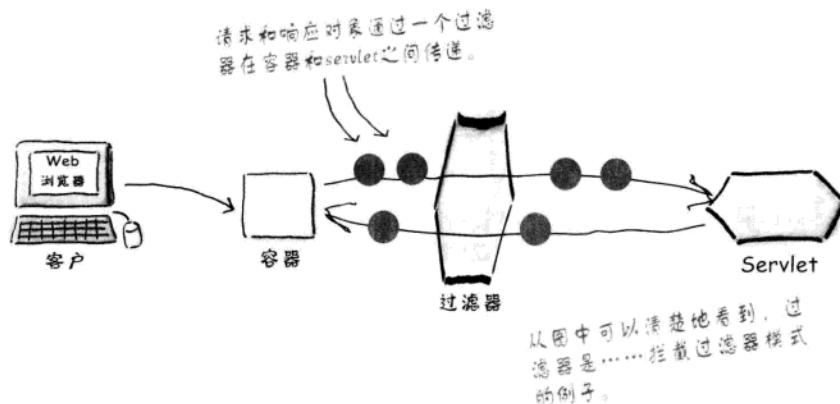
- 提供远程实体（也就是说，维护一些数据状态的对象）的一个本地表示。
- 尽量减少网络流量。
- 可以遵循Java bean约定，以便其他对象轻松地访问。
- 实现为一个可串行化的对象，从而能跨网络移动。
- 通常视图组件能很容易地访问。

## 传输对象原则

- 传输对象建立在以下原则基础上：
- 减少网络流量。
- 通过细粒度调用访问远程组件的数据时，利用传输对象能尽量减少对Web层性能的影响。
- 减少层间的耦合。
- 缺点是，访问传输对象的组件可能接收到过时的数据，因为传输对象的数据实际上表示在别处存储的状态。
- 要让可更新的传输对象安全地实现并发，这通常很复杂。

# 拦截过滤器

使用拦截过滤器模式来修改发送至servlet的请求，或者修改发送给用户的响应。



## 拦截过滤器功能

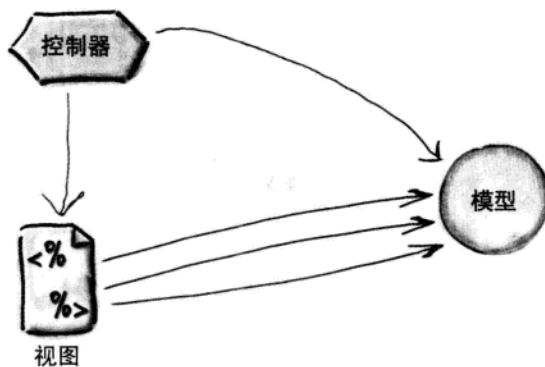
- 可以在请求到达servlet之前拦截/修改请求。
- 在响应返回给客户之前可以拦截/修改响应。
- 过滤器要使用DD以声明方式部署。
- 过滤器是模块化的，因此可以串成链执行。
- 过滤器的生命周期由容器管理。
- 过滤器必须实现容器回调方法。

## 拦截过滤器原则

- 拦截过滤器建立在以下原则基础上：
  - 内聚。
  - 松耦合。
  - 增强声明式控制。
- 通过声明式控制，可以很容易地暂时或永久地实现过滤器。
- 利用声明式控制，可以很容易地更新调用顺序。

# 模型-视图-控制器(MVC)

使用MVC模式来创建一个逻辑结构，将代码分离到应用的3种基本组件（模型、视图和控制器）。这会增加各组件的内聚度，并且能得到更大的可重用性，特别是模型组件。



## 模型-视图-控制器特性

- 可以独立地修改视图，而不影响控制器和模型。
- 模型组件对视图和控制器组件隐藏了内部细节（数据结构）。
- 如果模型遵循一个严格的契约（接口），那么这些组件可以在其他应用领域重用，如GUI或J2ME。
- 模型代码与控制器代码分离，这样可以更容易地移植为使用远程业务组件。

## 模型-视图-控制器原则

- 模型-视图-控制器建立在以下原则基础上：
  - 关注点分离。
  - 松耦合。
- 提高各组件的内聚度。
- 增加应用的整体复杂性（确实如此，因为尽管各个组件变得更加内聚，但是MVC会向应用增加许多新组件）。
- 尽量减少应用中其他层改动所带来的影响。

# 前端控制器

采用前端控制器模式，将常用的（而且经常是冗余的）请求处理代码收集到一个组件中。这能使控制器更加内聚，并且降低复杂性。



## — 前端控制器特性 —

- 把Web应用的初始请求处理任务都集中在一个组件中。
- 结合其他模式使用前端控制器时，可以采用声明方式建立表示层分派，从而提供松耦合。
- 前端控制器（这是指前端控制器本身，而不是Struts）的一个缺点是，与Struts相比，它相当简单。要使用前端控制器模式从头创建一个合理地应用，可能要重写Struts中已经有的许多特性。

## — 前端控制器原则 —

- 前端控制器建立在以下原则基础上：
  - 隐藏复杂性。
  - 关注点分离。
  - 松耦合。
- 提高应用控制器组件的内聚度。
- 减少应用的整体复杂性。
- 改善基础代码的可维护性。



## 第14章 模拟测验

1 给定以下性质：

- 与拦截过滤器相关。
- 支持开发人员之间的角色分离。
- 增加可重用性。

所描述的是哪个设计模式？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 业务委托

2 设计Web应用时，要求对每个接收到的请求采取某种安全措施。不论请求的类型如何，都必须应用某些安全检查。

可以使用哪个设计模式来实现这个设计需求？

- A. 传输对象
- B. 服务定位器
- C. 组合实体
- D. 业务委托
- E. 拦截过滤器

3 你的公司想充分利用其分布式仓库。你的任务是将应用的Web服务端点与其DAO无缝地集成。另外，必须改进粗粒度的控制器定位器，使之支持J2ME和UDDI注册表。

可以使用哪个设计模式来实现这些设计需求？

- A. 域激活器
- B. 拦截观察者
- C. 组合委托
- D. 传输外观

**4**

下面这句话描述了一个设计模式可能带来的好处：

这种模式减少了客户和企业bean之间的网络往返通信，在一个方法调用后，会为客户提供数据的一个本地副本，数据由一个企业bean封装，而不需要多个方法调用。这里描述的是哪一个设计模式？

- A. 传输对象
- B. 拦截过滤器
- C. 模型-视图-控制器
- D. 业务委托

**5**

你的公司（Models ‘R’ Us）正在创建一个高级的目录最大化组件，可供所有主要的J2EE容器开发商使用。你的任务是设计这个组件的一部分，完成JNDI查找，而不论客户使用哪个开发商的注册库。

哪个设计模式有助于你完成这个任务？

- A. 传输对象
- B. 拦截过滤器
- C. 模型-视图-控制器
- D. 业务委托
- E. 服务定位器

**6**

在对多层J2EE业务应用调优时，你会发现，如果减少应用所做远程请求的个数，并增加为每个请求收集的数据量，就能得到更好的性能。

你认为应当用哪个设计模式在应用中实现这个改变？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 拦截过滤器
- E. 模型-视图-控制器

7 给定以下性质：

- 与服务定位器相关。
- 减少耦合。
- 可能增加一层，并增加一定的复杂性。

所描述的是哪一个设计模式？

- A. 传输对象
- B. 前端控制器
- C. 业务委托
- D. 拦截过滤器
- E. 模型-视图-控制器

8 你的Web应用使用一个分布式应用中的SessionBean组件来完成专门的计算，如验证信用卡号。不过，你希望你的Web组件中没有查找SessionBean组件以及使用其接口的相关代码。你想将本地应用类与分布式组件的查找和使用解耦合，因为分布式组件的接口可能会改变。在这种情况下可以使用哪个J2EE设计模式？

- A. 传输对象
- B. 服务定位器
- C. 模型-视图-控制器
- D. 业务委托

9 给定以下性质：

- 与业务委托相关。
- 改善网络性能。
- 可以通过缓存提高客户性能。

所描述的是哪一个设计模式？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 拦截过滤器
- E. 模型-视图-控制器



## 第14章 模拟测验答案

1 给定以下性质：

- 与拦截过滤器相关。
- 支持开发人员之间的角色分离。
- 增加可重用性。

(核心J2EE模式, 180页)

所描述的是哪个设计模式？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 业务委托

这种模式（不只这一种）有助于将应用开发人员完成的任务与Web设计人员完成的任务分离。

2 设计Web应用时，要求对每个接收到的请求采取某种安全措施。不论请求的类型如何，都必须应用某些安全检查。

(核心J2EE模式, 144页)

可以使用哪个设计模式来实现这个设计需求？

- A. 传输对象
- B. 服务定位器
- C. 组合实体
- D. 业务委托
- E. 拦截过滤器

如果你想在正常的请求处理发生之前拦截和  
处理请求，拦截过滤器是一个很好的选择。

3 你的公司想充分利用其分布式仓库。你的任务是将应用的Web服务端点与其DAO无缝地集成。另外，必须改进粗粒度的控制器定位器，使之支持J2ME和UDDI注册表。

(设计模式, 第7章)

可以使用哪个设计模式来实现这些设计需求？

- A. 域激活器
- B. 拦截观察者
- C. 组合委托
- D. 传输外观

由于需求很容易改变，组合委托模  
式能提供最大程度的重构灵活性。

**4**

下面这句话描述了一个设计模式可能带来的好处：

(核心J2EE模式, 424页)

这种模式减少了客户和企业bean之间的网络往返通信，在一个方法调用后，会为客户提供数据的一个本地副本，数据由一个企业bean封装，而不需要多个方法调用。这里描述的是哪一个设计模式？

- A. 传输对象
- B. 拦截过滤器
- C. 模型-视图-控制器
- D. 业务委托

传输对象的一个主要优点就是减少网络流量。

**5**

你的公司 (Models 'R Us) 正在创建一个高级的目录最大化组件，可供所有主要的J2EE容器开发商使用。你的任务是设计这个组件的一部分，完成JNDI查找，而不论客户使用哪个开发商的注册库。

(核心J2EE模式, 316页)

哪个设计模式有助于你完成这个任务？

- A. 传输对象
- B. 拦截过滤器
- C. 模型-视图-控制器
- D. 业务委托
- E. 服务定位器

如果你想封装与特定开发商相关的服务查找，可以使用服务定位器模式。使用这种模式有助于隔离不同开发商特有的代码。

**6**

在对多层次J2EE业务应用调优时，你发现，如果减少应用所做远程请求的个数，并增加为每个请求收集的数据量，就能得到更好的性能。

(核心J2EE模式, 415~416页)

你认为应当用哪个设计模式在应用中实现这个改变？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 拦截过滤器
- E. 模型-视图-控制器

传输对象可以用于把多个细粒度的远程调用聚集为一个调用。通常，与编写一个更大对象所带来的开销相比，网络流量的减少是很值得的，这样能得到性能的提升。

7 给定以下性质：

- 与服务定位器相关。
- 减少耦合。
- 可能增加一层，并增加一定的复杂性。

所描述的是哪一个设计模式？

- A. 传输对象
- B. 前端控制器
- C. 业务委托
- D. 拦截过滤器
- E. 模型-视图-控制器

尽管增加了一层，但这种模式有很多好处（如减少耦合，而且提供了一个更简单的业务层接口），所以很值得。

8

你的Web应用使用一个分布式应用中的SessionBean组件来完成专门的计算，如验证信用卡号。不过，你希望你的Web组件中没有查找SessionBean组件以及使用其接口的相关代码。你想将本地应用类与分布式组件的查找和使用解耦合，因为分布式组件的接口可能会改变。在这种情况下可以使用哪个J2EE设计模式？

- A. 传输对象
- B. 服务定位器
- C. 模型-视图-控制器
- D. 业务委托

业务委托的一个主要优点是减少表示层和业务层之间的耦合。

(核心J2EE模式，  
308页)

9

给定以下性质：

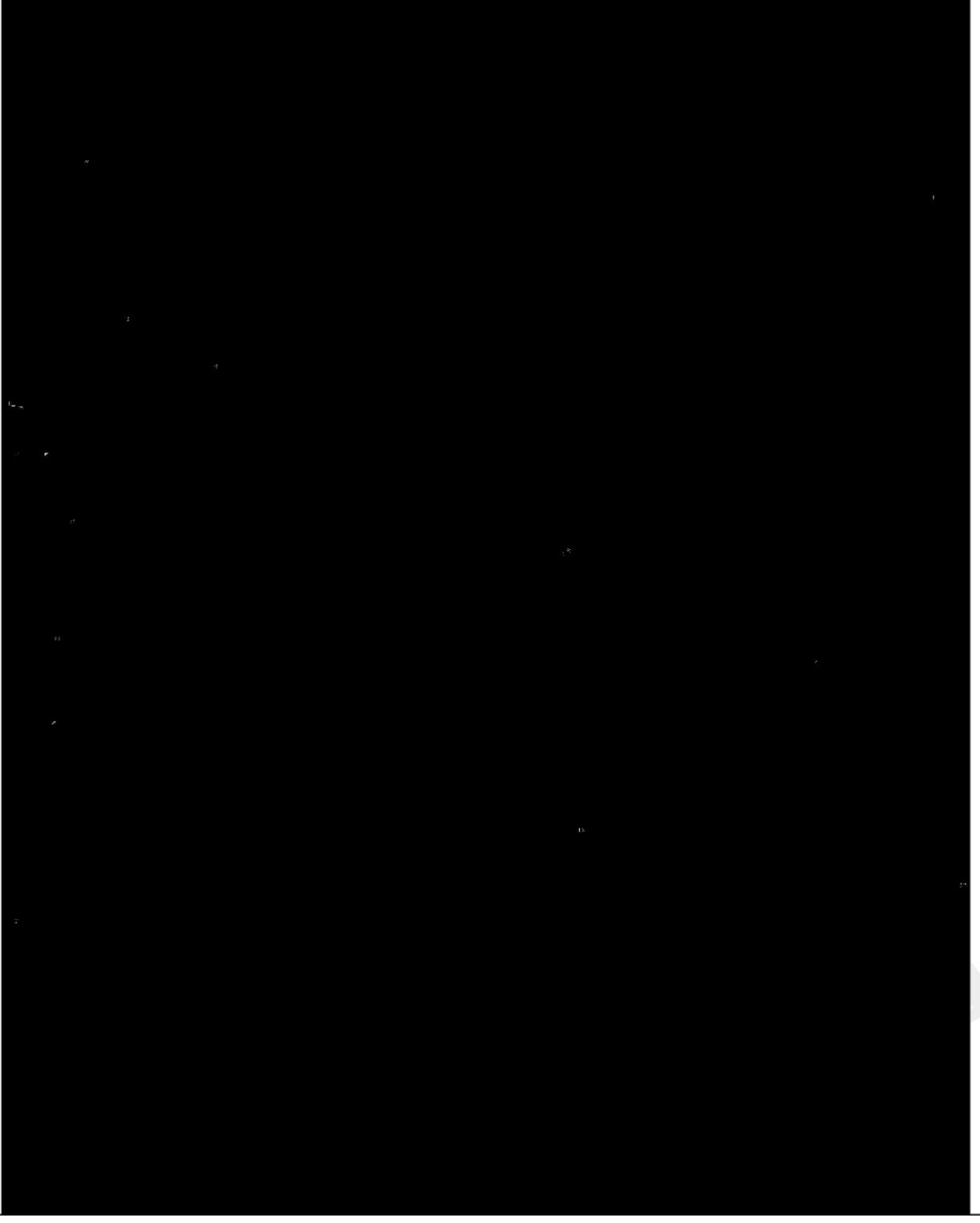
(核心J2EE模式，329页)

- 与业务委托相关。
- 改善网络性能。
- 可以通过缓存提高客户性能。

所描述的是哪一个设计模式？

- A. 传输对象
- B. 服务定位器
- C. 前端控制器
- D. 拦截过滤器
- E. 模型-视图-控制器

通过使用这个模式，可以合并查找所涉及的网络调用，并创建业务对象。



# 附录： 最终模拟测验



如果你还没有把握是不是真的准备好了，那么先不要做这个模拟题。如果做得太早，等你下次再做的时候，这就不是一份新考卷了，你会对其中的一些题目留有印象，尽管可能会得到高分，但很可能与实际不符。我们真的很希望你能一次就通过考试。（不过，要是你没能通过考试，每次重考时都再买一次这本书，那我们倒是很欢迎……）

为了避免“我记得见过这个题目”情况的发生，这个模拟测验要比实际考试稍难一点，这里没有告诉你每道题到底有多少个正确答案。我们的问题和答案从语调、风格、难度到内容都与实际考试几乎一样，但是由于没有告诉你该选择多少个答案，所以你无法根据这个提示排除答案。我们确实很严格，不过希望你知道，这种题目让我们自己也觉得很棘手，甚至比你更头疼（不过你该庆幸，就在几年前，Sun实际的Java考试采用的就是这种题型，大多数问题的后面都要求“选出所有正确的答案”）。

参加考试的大多数人都说我们的模拟测验比实际的SCWCD要难一些，但是他们的测验成绩与实际考试的得分却很接近。这个模拟测验可以很好地评测出你准备得怎么样了，但是要求你做到以下几点：

- (1) 必须在2小时15分钟内完成，就像参加真正的考试一样。
- (2) 做模拟测验的时候不能看书！
- (3) 不要一遍一遍地反复做。等做到第4次的时候，你可能98%都能做对，但仍然通不过实际的考试，这只是因为你把这个模拟测验的题目和答案记住了。
- (4) 在做模拟测验之前不要喝太多的酒或者其他提神的东西……



## 最终模拟测验

1 一个程序员为他的Java EE web应用（名为MyWebApp）建立了一个配置正确的目录结构。可以将一个名为myTag.tag的文件放在其中哪两个目录中从而能够由容器正确地访问？（有两个选择）

- A. MyWebApp/WEB-INF
- B. MyWebApp/META-INF
- C. MyWebApp/WEB-INF/lib
- D. MyWebApp/WEB-INF/tags
- E. MyWebApp/WEB-INF/TLDs
- F. MyWebApp/WEB-INF/tags/myTags

2 以下哪些是合法的EL？（选出所有正确的答案）

- A. \${"1" + "2"}
- B. \${1 plus 2}
- C. \${1 eq 2}
- D. \${2 div 1}
- E. \${2 & 1}
- F. \${"head"+"first"}

3

一个Java论坛网站的TLD中包含以下标记定义：

```
<tag>
    <name>avatar</name>
    <tag-class>hf.AvatarTagHandler</tag-class>
    <body-content>empty</body-content>

    <attribute>
        <name>userId</name>
        <required>true</required>
        <rteprvalue>true</rteprvalue>
    </attribute>
    <attribute>
        <name>size</name>
        <required>false</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
</tag>
```

关于AvatarTagHandler以下哪些说法是正确的？这里假设它扩展了SimpleTagSupport，并输出将显示用户avatar图像的HTML（选出所有正确的答案）。

- A. 这个类应当有一个size成员，它至少有一个设置方法。
- B. 代码中不需要size变量，因为TLD指出这个变量是不必要的。
- C. 需要一个覆盖的**doTag** 生命周期方法。
- D. 需要一个覆盖的**doStartTag** 生命周期方法。
- E. 这个类必须重载已实现的所有生命周期方法，这些重载的版本中，对应TLD中定义的每一个属性都要包括一个额外的参数。在这里只有一个需要重载的生命周期方法。

**4** 一个Servlet在转发到JSP之前建立一个bean。

给定：

```

20. foo.User user = new foo.User();
21. user.setFirst(request.getParameter("firstName"));
22. user.setLast(request.getParameter("lastName"));
23. user.setStreet(request.getParameter("streetAddress"));
24. user.setCity(request.getParameter("city"));
25. user.setState(request.getParameter("state"));
26. user.setZipCode(request.getParameter("zipCode"));
27. request.setAttribute("user", user);

```

如果在一个JSP中放入以下代码段，其中哪些代码段能够替换以上Servlet代码？（选出所有正确的答案）

- A. <jsp:useBean id="user" type="foo.User"/>
- B. <jsp:useBean id="user" type="foo.User">
 <jsp:setProperty name="user" property="\*"/>
</jsp:useBean>
- C. <jsp:useBean id="user" class="foo.User">
 <jsp:setProperty name="user" property="first" param="firstName"/>
 <jsp:setProperty name="user" property="last" param="lastName"/>
 <jsp:setProperty name="user" property="street" param="streetAddress"/>
 <jsp:setProperty name="user" property="city"/>
 <jsp:setProperty name="user" property="state"/>
 <jsp:setProperty name="user" property="zipCode"/>
</jsp:useBean>
- D. <jsp:useBean id="user" class="foo.User">
 <jsp:setProperty name="user" property="\*"/>
 <jsp:setProperty name="user" property="first" param="firstName"/>
 <jsp:setProperty name="user" property="last" param="lastName"/>
 <jsp:setProperty name="user" property="street" param="streetAddress"/>
</jsp:useBean>

5 比较业务委托对象和服务定位器对象的优点、局限性和用法时，以下哪些说法是正确的？（选出所有正确的答案）

- A. 它们都有可能建立网络调用。
- B. 它们都有可能调用传输对象中的方法。
- C. 它们都有可能由一个控制器对象直接调用。
- D. 服务定位器通常可以认为是业务委托的一个服务器。
- E. 如果实现时都建立了缓存，那么数据过时问题对于业务委托来说更为严重。

6 关于创建会话监听者，以下哪些说法是正确的？（选出所有正确的答案）

- A. 它们都在DD中声明。
- B. 并不是所有会话监听者都必须在DD中声明。
- C. 用来声明会话监听者的DD标记是`<listener>`。
- D. 用来声明会话监听者的DD标记是`<session-listener>`。
- E. 用来声明会话监听者的DD标记放在`<web-app>`标记中。
- F. 用来声明会话监听者的DD标记放在`<servlet>`。

7 有些用户抱怨说，在一台机器上打开两个浏览器窗口时，如果这两个窗口同时访问应用会发生一些奇怪的事情。你想测试一些浏览器，查看是否会在多个窗口间共享一个会话。为此，你决定在一个JSP中输出`JSESSIONID`。假设你的测试浏览器上启用了cookie，如何实现这个测试？（选出所有正确的答案）

- A.  `${cookie.JSESSIONID}`
- B.  `${cookie.JSESSIONID.value}`
- C.  `${cookie["JSESSIONID"]["value"]}`
- D.  `${cookie.JSESSIONID["value"]}`
- E.  `${cookie["JSESSIONID"].value}`
- F.  `${cookieValues[0].value}`

---

8 哪个隐式对象可以访问**ServletContext**的属性?

- A. **server**
- B. **context**
- C. **request**
- D. **application**
- E. **servletContext**

---

9 哪些方法在**HttpServlet**中? (选出所有正确的答案)

- A. **doGet**
- B. **doTrace**
- C. **doError**
- D. **doConnect**
- E. **doOptions**

---

10 你决定你的Web应用中的某些功能将要求用户是注册会员。另外，你的Web应用有时会处理用户希望你保密的一些用户数据。

以下哪些说法是正确的? (选出所有正确的答案)

- A. 只有在应用验证了用户的口令之后才能保证所传输数据的机密性。
- B. 在Java EE容器保证的各种认证中，只有基本认证（BASIC）、摘要认证（Digest）和基于表单的认证（Form Based）要求用户名与口令匹配。
- C. 不论你使用何种类型的Java EE认证机制，只有在请求受限资源时才会被激活。
- D. Java EE保证的各种类型的认证都能提供高强度的数据安全性，而不需要实现支持安全特性。

11

以下是一个Java EE DD中的某标记中取出的代码段：

```
343.    <web-resource-collection>
344.        <web-resource-name>Recipes</web-resource-name>
345.        <url-pattern>/Beer/Update/*</url-pattern>
346.        <http-method>POST</http-method>
347.    </web-resource-collection>
...
367.    <auth-constraint>
368.        <role-name>Member</role-name>
369.    </auth-constraint>
...
385.    <user-data-constraint>
386.        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
387.    </user-data-constraint>
```

以下哪些说法是正确的？（选出所有正确的答案）

- A. Java EE DD可以包含某一个标记，在其中以上所有这些标记都能合法地共存。
- B. 在选项A中所述的单个标记中还可以存在更多`<auth-constraint>`实例。
- C. 在选项A中所述的单个标记中还可以存在更多`<user-data-constraint>`实例。
- D. 在以上所述的`<web-resource-collection>`标记中还可以存在更多`<url-pattern>`实例。
- E. 与选项A中所述单个外围标记同类型的其他标记可以有与以上标记相同的`<url-pattern>`。
- F. 这个标记表明，Web应用的授权、认证和数据完整性等安全特性都已经得到声明。

12

你声明了一个JSP文档，它生成一个动态SVG图像（由一个XML文档结构表示）。这个JSP必须将HTTP响应首部'Content-Type'声明为'image/svg+xml'，从而使web浏览器将响应显示为一个SVG图像。

以下哪个JSP代码段声明这个JSP文档是一个SVG响应？

- A. <%@ page contentType='image/svg+xml' %>
- B. <jsp:page contentType='image/svg+xml' />
- C. <jsp:directive.page contentType='image/svg+xml' />
- D. <jsp:page.contentType>image/svg+xml</jsp:page.contentType>

13

给定一个JSP页面中有以下代码：

```
<%-- out.print("Hello World"); --%>
```

它的HTML输出是什么？

- A. Hello World
- B. out.print("Hello World");
- C. <!-- Hello World -->
- D. 这行代码不会生成任何输出。

14

关于HTTP会话支持，以下哪些说法是正确的？（选出所有正确的答案）

- A. Java EE 容器必须支持HTTP cookie。
- B. Java EE 容器必须支持URL重写。
- C. Java EE 容器必须支持安全套接字层。
- D. Java EE 容器必须支持HTTP会话，甚至包括不支持cookie的客户。
- E. Java EE 容器必须能够识别HTTP终止信号，发出HTTP终止信号则指示一个客户会话不再活动。

15

你的公司购买了一个第三方JavaScript库（用于构建菜单）的许可。你的开发小组错误地滥用了这个库，以至于遭遇无数错误，另外用户坚持认为某些菜单项只有得到授权的用户才能看到。一个使用简单标记处理器的定制标记库可以避免开发人员产生JavaScript语法错误，并且能够提供用户所需的安全特性。

一次设计会议之后，小组的领导人声明他希望菜单如下所示：

```
<menu:main>
  <menu:headItem text="My Account" url="/myAccount.do"/>
  <menu:headItem text="Transactions">
    <menu:subItem text="Incoming" url="/incomingTx.do"/>
    <menu:subItem text="Outgoing" url="/outgoingTx.do"/>
    <menu:subItem text="Pending" url="/pendingTx.do"
      requireRole="accountant"/>
  </menu:headItem>
  <menu:headItem text="Admin" url="/admin.do"
    requireRole="admin"/>
</menu:main>
```

你希望完全由外部标记处理器负责生成输出，这里假设很容易将显示逻辑集中在一起。外部`<menu:main>`标记处理器需要访问其子孙标记来完成这个任务。以下哪个选项可以提供最佳方法？

- A. 每个内部标记应当直接向其父标记注册。  
直接父标记可以将其子标记存储在一个有序集合中。
- B. 每个内部标记应当直接向外部标记处理器注册，  
外部标记处理器可以将它们都存储在一个`HashSet`中。
- C. 不同于传统标记，`SimpleTagSupport`提供了方法  
`findDescendentWithClass()`和`getChildren()`。  
这样外部标记完全可以访问其子标记，而无需另外编写代码。
- D. 让各个内部标记将其自己保存为一个页面作用域属性，  
其`text`值作为属性的键。

16 请求一个JSP页面时, JSP生命周期中的哪个阶段可能导致返回一个HTTP 500状态码? (选出所有正确的答案)

- A. JSP 页面编译阶段
- B. 执行服务方法时
- C. 执行撤销方法时
- D. 执行初始化方法时

17 给定`session`是一个合法`HttpSession`的引用, “`myAttr`”是绑定到`session`的对象的名, 可以用以下哪个方法解除对象与会话的绑定? (选出所有正确的答案)

- A. `session.unbind();`
- B. `session.invalidate();`
- C. `session.unbind("myAttr");`
- D. `session.remove("myAttr");`
- E. `session.invalidate("myAttr");`
- F. `session.removeAttribute("myAttr");`
- G. `session.unbindAttribute("myAttr");`

18 如果`req`是一个`HttpServletRequest`的引用, 当前没有会话, 关于`req.getSession()`以下哪种说法是正确的? (选出所有正确的答案)

- A. 调用`req.getSession()`将返回null。
- B. 调用 `req.getSession(true)`将返回null。
- C. 调用 `req.getSession(false)`将返回null。
- D. 调用 `req.getSession()`将返回一个新会话。
- E. 调用 `req.getSession(true)`将返回一个新会话。
- F. 调用 `req.getSession(false)`将返回一个新会话。

19

遗留代码中有一个传统标记处理器。代码的作者写了一个处理器，将其标记体计算100次，用来测试其他生成随机内容的标记。

给定以下代码：

```

06. public class HundredTimesTag extends TagSupport {
07.     private int iterationCount;
08.     public int doTag() throws JspException {
09.         iterationCount = 0;
10.         return EVAL_BODY_INCLUDE;
11.     }
12.
13.     public int doAfterBody() throws JspException {
14.         if(iterationCount < 100){
15.             iterationCount++;
16.             return EVAL_BODY_AGAIN;
17.         }else{
18.             return SKIP_BODY;
19.         }
20.     }
21. }
```

关于以上代码哪个说法是正确的？

- A. 标记处理器不是线程安全的，所以如果多个用户同时访问这个页面`iterationCount`可能不同步。
- B. `doAfterBody`方法永远不会得到调用，因为它不是标记处理器生命周期的一部分。开发者应当扩展`IterationTagSupport`类，将这个方法包括在生命周期中。
- C. `doTag`方法应当是`doStartTag`。按照以上代码，会调用`TagSupport`的默认`doStartTag`，它只是返回`SKIP_BODY`，这会导致`doAfterBody`永远也不会得到调用。
- D. `doAfterBody`返回`EVAL_BODY_AGAIN`时，会再次调用`doTag`方法。`doTag`方法将`iterationCount`重置为0，导致一个无限循环，并抛出一个`java.lang.OutOfMemoryError`。

20

给定一个Web应用DD中的以下代码片段：

```
72. <session-config>
73.   <session-timeout>10</session-timeout>
74. </session-config>
```

另外，给定**session**是一个合法**HttpSession**的引用，其相应的servlet代码如下：

```
30. session.setMaxInactiveInterval(120);
```

执行第30行后，以下哪个说法是正确的？（选出所有正确的答案）

- A. DD代码片段不合法。
- B. 调用**setMaxInactiveInterval**将修改**<session-timeout>**标记中的值。
- C. 根据以上条件不可能确定会话的超时时限。
- D. 如果容器在2个小时内没有接收到对这个会话的客户请求，容器将置会话无效。
- E. 如果容器在2分钟内没有接收到对这个会话的客户请求，容器将置会话无效。
- F. 如果容器在10秒内没有接收到对这个会话的客户请求，容器将置会话无效。
- G. 如果容器在10分钟内没有接收到对这个会话的客户请求，容器将置会话无效。

21

你已经创建了一个合法的目录结构，并为你的Java EE Web应用创建了一个合法的WAR文件。给定以下条件：

- **ValidApp.war**是WAR文件名。
- **WARdir**表示每个WAR文件中必须有的目录。
- **APPdir**表示每个Web应用中必须有的目录。

以下哪个说法是正确的？

- A. **WARdir**的具体名是无法预知的。
- B. 应用名是无法预知的。
- C. 在这个目录结构中，**APPdir**将在**WARdir**中。
- D. 在这个目录结构中，应用的部署描述文件将在**WARdir**所在的目录中。
- E. 将应用置于一个**WAR**文件时，这样可以为容器提供一个选择，能够完成额外的运行时检查，否则这一点是无法保证的。

**22**

比较HTTP **GET**和HTTP **POST**时，以下哪些说法是正确的？（选出所有正确的答案）

- A. 只有HTTP **GET**是幂等的。
- B. 这两个方法都要求在HTML表单标记中有一个显式声明。
- C. 只有HTTP **POST**可以在一个请求中支持多个参数。
- D. 二者都支持发送多个值的单参数请求。
- E. 只有HTTP **POST**请求应当通过覆盖servlet的**service()**方法来处理。

**23**

给定一个servlet中的以下代码。

```
82. String s = getServletConfig().getInitParameter(myThing);
```

哪个DD片段将把s 赋值为myStuff?

- A. <init-param>  
    <param>myThing</param>  
    <value>myStuff</value>  
  </init-param>
- B. <init-param>  
    <name>myThing</name>  
    <value>myStuff</value>  
  </init-param>
- C. <init-param>  
    <param-name>myThing</param-name>  
    <param-value>myStuff</param-value>  
  </init-param>
- D. <servlet-param>  
    <name>myThing</name>  
    <value>myStuff</value>  
  </servlet-param>
- E. <servlet-param>  
    <param-name>myThing</param-name>  
    <param-value>myStuff</param-value>  
  </servlet-param>

**24**

给定一个String存储为某个作用域中一个名为**accountNumber**的属性，以下哪个（哪些）scriptlet将输出这个属性？

- A. `<%= pageContext.findAttribute("accountNumber") %>`
- B. `<%= out.print("${accountNumber}") %>`
- C. `<% Object accNum = pageContext.getAttribute("accountNumber");  
if(accNum == null){  
 accNum = request.getAttribute("accountNumber");  
}  
if(accNum == null){  
 accNum = session.getAttribute("accountNumber");  
}  
if(accNum == null){  
 accNum = servletContext.getAttribute("accountNumber");  
}  
out.print(accNum);  
%>`
- D. `<% requestDispatcher.include("accountNumber"); %>`

**25**

你得到了一个遗留的JSP Web应用，其中有大量脚本代码。你的经理要求必须对每一个JSP重构，删除其中的脚本代码。他希望你能保证JSP代码基中不再出现scriptlet代码，并让Web容器强制一种“无脚本”策略。

哪个 `web.xml` 配置元素可以完成这个目标？

- A. `<jsp-property-group>  
 <url-pattern> *.jsp </url-pattern>  
 <permit-scripting> false </permit-scripting>  
</jsp-property-group>`
- B. `<jsp-config>  
 <url-pattern> *.jsp </url-pattern>  
 <permit-scripting> false </permit-scripting>  
</jsp-config>`
- C. `<jsp-property-group>  
 <url-pattern> *.jsp </url-pattern>  
 <scripting-invalid> true </scripting-invalid>  
</jsp-property-group>`
- D. `<jsp-config>  
 <url-pattern> *.jsp </url-pattern>  
 <scripting-invalid> true </scripting-invalid>  
</jsp-config>`

26

给定以下代码：

```

01. <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
02.
03. <%
04. java.util.List books = new java.util.ArrayList();
05. // add line here
06. request.setAttribute("myFavoriteBooks", books);
07. %>
08.
09. <c:choose>
10. <c:when test="${not empty myFavoriteBooks}">
11.     My favorite books are:
12.     <c:forEach var="book" items="${myFavoriteBooks}">
13.         <br/> * ${book}
14.     </c:forEach>
15. </c:when>
16. <c:otherwise>
17.     I have not selected any favorite books.
18. </c:otherwise>
19. </c:choose>
```

如果独立地插入到第5行，以下哪些代码行会导致显示出c:otherwise标记中的文本？（选出所有正确的答案）

- A. books.add("");
- B. books.add(null);
- C. books.clear();
- D. books.add("Head First");
- E. books = null;

27

你在开发一个管理企业名单目录的应用。

给定以下代码：

```
29. <c:forEach var="phoneNumber" items='${company.
   contactInfo.phoneNumbers}'>
30.   <c:if test='${verify:isTollFree(phoneNumber)}'>
31.     
32.   </c:if>
33.   ${phoneNumber}<br/>
34. </c:forEach>
```

以上代码段在免计费电话号码前增加一个特殊的图标。关于这个代码段中的EL函数，以下哪个说法肯定是正确的？

- A. EL函数必须声明为公共（public）和静态（static）
- B. EL函数不能返回任何值，而且必须声明为void
- C. EL函数TLD中的<uri>值必须是 Verify
- D. 实现EL函数的类必须命名为Verify
- E. 如果 phoneNumber 是一个String，TLD中的<function-signature>值应当是isTollFree(String)

28

`HttpServletRequest` 的哪些方法可以获取请求体？（选出所有正确的答案）

- A. `getReader()`
- B. `getStream()`
- C. `getInputReader()`
- D. `getInputStream()`
- E. `getServletReader()`
- F. `getServletStream()`

29

给定一个 Java EE Web 应用，其中，对于以下浏览器请求：

<http://www.wickedlysmart.com/MyApp/myDir/DoSomething>

— 将由应用中的一个 servlet 处理，以下哪 3 种说法是正确的？（选出 3 个正确的选项）

- A. 部署描述文件必须包含按指定要求处理这个请求的指令。
- B. 可以按指定要求处理这个请求，而部署描述文件中无需相关的指令。
- C. 处理这个请求的 servlet 必须命名为 **DoSomething.class**
- D. 根据所提供的信息无法预知 servlet 名。
- E. 这个应用必须包含一个名为 **myDir** 的目录。
- F. 根据所提供的信息无法预知 servlet 所在目录的目录名。

30

你的 Web 应用有一个合法的部署描述文件，其中只定义了 **student** 和 **sensei** 安全角色。部署描述文件包含两个安全约束，声明了对同一个资源的安全约束。第一个安全约束包含：

```
234. <auth-constraint>
235.   <role-name>student</role-name>
236. </auth-constraint>
```

第二个安全约束包含：

```
251. <auth-constraint/>
```

以下哪个说法是正确的？（选出所有正确的答案）

- A. 根据现在的部署描述文件，这两个角色都可以访问这个受限资源。
- B. 根据现在的部署描述文件，只有 **sensei** 用户能访问这个受限资源。
- C. 根据现在的部署描述文件，只有 **student** 用户能访问这个受限资源。
- D. 如果去掉第二个 **<auth-constraint>** 标记，这两个角色都可以访问这个受限资源。
- E. 如果去掉第二个 **<auth-constraint>** 标记，只有 **sensei** 用户能访问这个受限资源。
- F. 如果去掉第二个 **<auth-constraint>** 标记，只有 **student** 用户能访问这个受限资源。

31

以下哪个定制标记肯定会失败？（选出所有正确的答案）

- A. <mine:border>  
<mine:photos album="\${albumSelected}">  
</mine:border>  
</mine:photos>
- B. <mine:border>  
    <mine:photos album="\${albumSelected}"/>  
</mine:border>
- C. <mine:border>  
    \${albumSelected.title}  
    <mine:photos>\${albumSelected}</mine:photos>  
</mine:border>
- D. <mine:photos includeBorder="\${userPreference.border}"  
    album="\${albumSelected}" />

32

你的n层Web应用使用了一些Java EE模式（它们是应用希望访问远程注册表时最常使用的模式）。这些模式有哪些好处？（选出所有正确的答案）

- A. 增加内聚度。
- B. 性能更优。
- C. 可维护性更好。
- D. 减少网络流量。
- E. 浏览器功能交互性更强。

33

关于servlet的生命周期，以些哪些说法总是正确的？（选出所有正确的答案）

- A. 不能为servlet编写构造函数。
- B. 不能覆盖servlet的**init()**方法。
- C. 不能覆盖servlet的**doGet()**方法。
- D. 不能覆盖servlet的 **doPost()**方法。
- E. 不能覆盖servlet的**service()**方法。
- F. 不能覆盖servlet的**destroy()**方法。

34

给定一个Java EE .war 文件目录结构的一部分：

```

MyApp
| -- META-INF
|   | -- MANIFEST.MF
|   | -- web.xml
|
| -- WEB-INF
|   | -- index.html
|   | -- TLDs
|
|   | -- Header.tag

```

要使这个结构合法，而且资源可访问，需要做哪些修改？（选出所有正确的答案）

- A. 不需做任何修改。
- B. `web.xml` 文件必须移动。
- C. `index.html` 文件必须移动。
- D. `Header.tag` 文件必须移动。
- E. `MANIFEST.MF` 文件必须移动。
- F. `WEB-INF` 目录必须移动。
- G. `META-INF` 目录必须移动。

35

你考虑在你的Java EE n层应用中实现某种MVC。以下哪些说法是正确的？（选出所有正确的答案）

- A. 这个设计通常为业务委托对象服务。
- B. 通常通过缓存位于远程的数据来减少网络流量。
- C. 这个设计目标可以简化与异构资源注册表的通信。
- D. 尽管MVC解决方案有很多优点，但通常会增加设计复杂性。
- E. 前端控制器模式和Struts都可以认为是这种设计目标的解决方案。
- F. 这个设计将使你能够轻松地重新结合请求和响应处理器。

36

给定一个JSP页面中有以下代码行：

```
<% List myList = new ArrayList(); %>
```

能用以下哪个JSP代码段导入这些数据类型？（选出两个正确答案）

- A. <%! import java.util.\*; %>
- B. <%@ import java.util.List java.util.ArrayList %>
- C. <%@ page import='java.util.List,java.util.ArrayList' %>
- D. <%! import java.util.List; import java.util.ArrayList; %>
- E. <%@ page import='java.util.List' %> <%@ page import='java.util.ArrayList' %>

37

你要完成一个任务，向公司的Java EE web应用增加一些安全特性。具体地，需要创建多种不同类别的用户，另外根据用户的类别，需要对他们做出限制，从而只能使用应用中的某些页面。为了限制用户的访问，必须确定用户确实是他们所声称的那个人。

以下哪些说法是正确的？（选出所有正确的答案）

- A. 如果需要验证用户确实是他们所声称的那些人，就必须使用应用的部署描述文件来实现这个需求。
- B. 必须使用Java EE的授权功能来确定用户是他们所声称的那些人。
- C. 为了帮助你确定用户确实是他们所声称的那些人，可以使用部署描述文件的<login-config> 标记。
- D. 为了帮助你确定用户确实是他们所声称的那些人，可以使用部署描述文件的<user-data-constraint> 标记。
- E. 根据你使用的方法，要确定用户是他们所声称的那些人，可能需要包含一个“realm”。

38

ValidApp是一个Java EE应用，有一个合法的目录结构。ValidApp将.gif图像文件包含在目录结构的三个位置中。

- ValidApp/imageDir/
- ValidApp/META-INF/
- ValidApp/WEB-INF/

在以下哪些位置上用户可以直接访问这些.gif文件？

- A. 只有ValidApp/META-INF/
- B. 只有ValidApp/imageDir/
- C. 以上所有位置
- D. 只有ValidApp/imageDir/ 和 ValidApp/WEB-INF/
- E. 只有 ValidApp/imageDir/ 和 ValidApp/META-INF/

39

给定req是一个合法HttpServletRequest的引用，有以下代码：

```

13. String[] s = req.getCookies();
14. Cookie[] c = req.getCookies();
15. req.setAttribute("myAttr1", "42");
16. req.setAttribute("myAttr2", 42);
17. String[] s2 = req.getAttributeNames();
18. String[] s3 = req.getParameterValues("attr");

```

哪些代码行无法编译？（选出所有正确的答案）

- A. 第13行
- B. 第14行
- C. 第15行
- D. 第16行
- E. 第17行
- F. 第18行

**40**

名为 **Products.tag** 的标记文件显示了一个产品列表。

给定标记文件中的以下代码段：

```
1. <%@ attribute name="header" required="false" rtxexprvalue="false" %>
2. <%@ attribute name="products" required="true" rtxexprvalue="true" %>
3. <%@ tag body-content="tagdependent" %>
```

以下哪些是这个标记文件的合法使用？（选出所有正确的答案）

- A. <display:Products header="Shopping Cart" products="\${shoppingCart}" />
- B. <display:Products header="Wish List" products="\${wishList}" body-content="\${body}" />
- C. <display:Products header="Similar Products" products="\${similarProductCustomers who bought this item also bought:}</display:Products>
- D. <display:Products header='<%= request.getParameter("listType") %>' />

**41**

你在参与一个项目，要从一个大型银行遗留Web应用的JSP中去除scriptlet页。你遇到以下代码：

```
<% if((com.yourcompany.Account)request.
    getAttribute("account")).  

    isPersonalChecking()){ %>  

    Checking that fits your lifestyle.  

<% } %>
```

如何使用JSTL替换以上代码？（选出所有正确的答案）

- A. <c:if test='\${account.personalChecking}'>Checking  
that fits your lifestyle.</c:if>
- B. <c:if test='\${account["personalChecking"]}'>Checking  
that fits your lifestyle.</c:if>
- C. <c:if test="\${account['personalChecking']}'>Checking  
that fits your lifestyle.</c:if>
- D. <c:if test='\${account.isPersonalChecking}'>Checking  
that fits your lifestyle.</c:if>

**42**

给定以下事件类型：

- **HttpSessionEvent**
- **HttpSessionBindingEvent**
- **HttpSessionAttributeEvent**

将上面的事件类型与其各自的监听者接口匹配。(注意，一个事件类型可以与多个Listener匹配)

<b>HttpSessionAttributeListener</b>	.....
<b>HttpSessionListener</b>	.....
<b>HttpSessionActivationListener</b>	.....
<b>HttpSessionBindingListener</b>	.....

**43**

关于servlet的生命周期以下哪些说法是正确的？(选出所有正确的答案)

- A. 接收一个新请求时，**service()** 方法是容器最先调用的方法。
- B. **doPost()**或**doGet()**完成一个请求之后会调用**service()**方法。
- C. 每次调用**doPost()**时，会在其自己的线程中运行。
- D. 每次**doGet()**调用完成后会调用**destroy()**方法。
- E. 容器为每个客户请求建立一个单独的线程。

**44**

什么情况下JSP会得到转换？(选出所有正确的答案)

- A. 开发人员编译src文件夹中的代码时。
- B. 启动应用时。
- C. 用户第一次请求JSP时。
- D. 调用**jspDestroy()**之后JSP会得到转换。

45

以下是取自一个合法doGet()方法的代码段：

```
12.     OutputStream os = response.getOutputStream();
13.     byte[] ba = {1,2,3};
14.     os.write(ba);
15.     RequestDispatcher rd = request.getRequestDispatcher("my.jsp");
16.     rd.forward(request, response);
```

假设“my.jsp”向响应增加了字节4, 5和6, 结果是什么?

- A. 123。
- B. 456。
- C. 123456。
- D. 456123。
- E. 会抛出一个异常。

46

一个程序员需要更新一个正在运行的servlet的初始化参数，使Web应用立即开始使用这些新参数。

为了完成这个任务，必须完成以下哪些步骤（不过可能并不充分）？（选出所有正确的答案）

- A. 对于每个参数，必须修改一个指定servlet名、参数名以及新参数值的DD标记。
- B. servlet的构造函数必须从servlet的**ServletConfig**对象获取更新后的DD参数。
- C. 容器必须撤销然后重新初始化这个servlet。
- D. 对于每个参数，DD必须有一个单独的<init-param> 标记。

47

哪些类型可以与**HttpServletResponse** 方法结合使用来输出数据？（选出所有正确的答案）

- A. `java.io.PrintStream`
- B. `java.io.PrintWriter`
- C. `java.io.OutputStream`
- D. `java.io.FileOutputStream`
- E. `java.io.ServletOutputStream`
- F. `java.io.ByteArrayOutputStream`

**48** 你的Web应用有一个合法的DD, 其中只有一个<**security-constraint**> 标记。在这个标记中:

- 有一个声明为**directory1**的uri模式
- 有一个声明为**POST**的HTTP方法
- 有一个声明为**GUEST**的角色名

如果应用的所有资源都在**directory1** 和**directory2**中, 而且**MEMBER** 也是一个合法的角色, 以下哪种说法是正确的? (选出所有正确的答案)

- A. **GUEST**不能在**directory1**中完成**GET**请求。
- B. **GUEST**在两个目录中都能完成**GET** 请求。
- C. **GUEST**只能在**directory2**中完成**POST**请求。
- D. **MEMBER**在两个目录中都能完成**GET**请求。
- E. **GUEST**在两个目录中都能完成**POST**请求。
- F. **MEMBER**在**directory1**中只能完成**POST** 请求。

**49** 给定以下代码:

```
1. <%@ taglib prefix="c" uri="http://java.sun.com/jsp/
   jstl/core" %>
2. <%@ taglib prefix="tables" uri="http://www.javaranch.
   com/tables" %>
3. <%@ taglib prefix="jsp" tagdir="/WEB-INF/tags" %>
4. <%@ taglib uri="UtilityFunctions" prefix="util" %>
```

以上taglib指令会导致JSP无法工作, 对此下面的哪个说法是正确的?

- A. 第4行不对, 因为prefix必须在uri属性前面。
- B. 第3行不对, 因为不存在uri属性。
- C. 第4行不对, 因为uri值必须以**http://**开头。
- D. 第3行不对, 因为前缀**jsp** 是为标准动作保留的。

50

给定 `resp` 是一个合法 `HttpServletResponse` 对象的引用, 这个响应中包含以下首部 (还可能包含有其他首部) :

`Content-Type: text/html`

`MyHeader: mydata`

并有以下调用:

```
25. resp.addHeader(MyHeader, mydata2);  
26. resp.setHeader(MyHeader, mydata3);  
27. resp.addHeader(MyHeader, mydata);
```

`MyHeader` 首部中存在哪些数据?

- A. `mydata`
- B. `mydata3`
- C. `mydata3,mydata`
- D. `mydata3,mydata2`
- E. `mydata,mydata2,mydata3`
- F. `mydata,mydata2,mydata3,mydata`

51

以下是一个遗留应用的 web.xml 中的一部分:

```
<jsp-config>  
  <taglib>  
    <taglib-uri>prettyTables</taglib-uri>  
    <taglib-location>/WEB-INF/tlds/prettyTables.tld</taglib-location>  
  </taglib>  
</jsp-config>
```

假设运行代码的服务器现在支持 Java 1.4 EE 或更高版本, 怎样才能去除以上 `<jsp-config>` 标记而使代码仍能正常工作?

- A. 将 JSP 中 `taglib` 指令的 `uri` 属性改为 "`*`", 容器就会自动映射到它。
- B. 在 TLD 文件中放入 `<uri>prettyTables</uri>`。
- C. 删除 JSP 中使用这个映射的 `taglib` 指令。容器会自动进行处理。
- D. 这是不可能的。要让容器将 TLD 映射到 JSP 中引用的 `uri`, 就必须有这里的 `<jsp-config>`。

52

对于一个列出购物车商品的页面，当购物车为空时必须显示消息“Your shopping cart is empty.”（你的购物车为空）。以下哪个代码段可以实现这个功能，在此假设作用域属性cart是一个商品List? (选出所有正确的答案)

- A. 

```
<c:if test='${empty cart}'>
    Your shopping cart is empty.
</c:if>
<c:forEach var="itemInCart" items="${cart}">
    <shop:displayItem item="${itemInCart}" />
</c:forEach>
```
- B. 

```
<c:forEach var="itemInCart" items="${cart}">
    <c:choose>
        <c:when test='${empty itemInCart}'>
            Your shopping cart is empty.
        </c:when>
        <c:otherwise>
            <shop:displayItem item="${itemInCart}" />
        </c:otherwise>
    </c:choose>
</c:forEach>
```
- C. 

```
<c:choose>
    <c:when test='${empty cart}'>
        Your shopping cart is empty.
    </c:when>
    <c:when test='${not empty cart}'>
        <c:forEach var="itemInCart" items="${cart}">
            <shop:displayItem item="${itemInCart}" />
        </c:forEach>
    </c:when>
</c:choose>
```
- D. 

```
<c:choose>
    <c:when test='${empty cart}'>
        Your shopping cart is empty.
    </c:when>
    <c:otherwise>
        <c:forEach var="itemInCart" items="${cart}">
            <shop:displayItem item="${itemInCart}" />
        </c:forEach>
    </c:otherwise>
</c:choose>
```

53

给定一个servlet中的以下代码,而且myVar 是HttpSession或ServletContext的一个引用:

```
15. myVar.setAttribute(myName, myVal);  
16. String s = (String) myVar.getAttribute(myName);  
17. // more code
```

执行第16行之后,以下哪种说法是正确的? (选出所有正确的答案)

- A. s的值不能保证。
- B. 如果 myVar是一个HttpSession, 编译会失败。
- C. 如果 myVar是一个ServletContext, 编译会失败。
- D. 如果 myVar是一个HttpSession, s 的值肯定是“myVal”。
- E. 如果 myVar是一个ServletContext, s 的值肯定是“myVal”。

54

以下是Java EE web应用部署描述文件中的一部分:

```
62. <error-page>  
63.   <exception-type>IOException</exception-type>  
64.   <location>/mainError.jsp</location>  
65. </error-page>  
66. <error-page>  
67.   <error-code>404</error-code>  
68.   <location>/notFound.jsp</location>  
69. </error-page>
```

以下哪些说法是正确的?

- A. 这个部署描述文件不合法。
- B. 如果应用抛出一个IOException异常, 不会提供任何页面。
- C. 如果应用抛出一个IOException异常, 会提供notFound.jsp。
- D. 如果应用抛出一个IOException异常, 会提供mainError.jsp。

**55**

给定以下JSP：

1. <%! String GREETING = "Welcome to my page"; %>
2. <% request.setAttribute("greeting", GREETING); %>
3. Greeting: \${greeting}
4. Again: <%= request.getAttribute("greeting") %>

如果试图将以上JSP转换为一个JSP文档：

01. <jsp:declaration>
02. String TITLE = "Welcome to my page";
03. </jsp:declaration>
04. <jsp:scriptlet>
05. request.setAttribute("greeting", GREETING);
06. </jsp:scriptlet>
07. Greeting: \${greeting}
08. Again: <jsp:expression>
09. request.getAttribute("greeting");
- 10.</jsp:expression>

这个新JSP文档哪里有问题？（选出所有正确的答案）

- A. 没有声明<jsp:root>。
- B. 模板文本应当用 <jsp:text> 标记包围起来。
- C. EL表达式在JSP文档中是不允许的。
- D. <jsp:expression> 的内容不能有分号。

**56**

以下哪一个组件最不可能建立或接收网络调用？

- A. JNDI 服务器
- B. 传输对象
- C. 服务定位器
- D. 前端控制器
- E. 拦截过滤器

**57**

给定以下代码：

```

10. ${questionNumber}: ${question}
11. <c:forEach var="answer" items="${answers}">
...
16. </c:forEach>
```

question属性是一个String，其中可能包含必须在浏览器中显示为常规文本的XML标记。基于以上代码段，浏览器不会显示XML标记。应当如何修改来修正这个问题？（选出所有正确的答案）

- A. \${question} 替换为 <c:out value="\${question}"/>
- B. \${question} 替换为 <c:out>\${question}</c:out>
- C. \${question} 替换为 <c:out escapeXml="true" value="\${question}"/>
- D. \${question} 替换为 <%= \${question} %>

**58**

你的Java EE Web应用用户越来越多，你决定再增加一个服务器来支持更多的客户请求。对于会话从一个服务器迁移到另一个服务器，以下哪些说法是正确的？（选出所有正确的答案）

- A. 在会话中这些迁移是不可能的。
- B. 迁移会话时，其**HttpSession** 会随之迁移。
- C. 迁移会话时，其**ServletContext**会随之迁移。
- D. 迁移会话时，其**HttpServletRequest**会随之迁移。
- E. 如果使用**HttpSession.setAttribute**增加对象，这个对象必须是**Serializable**才能从一个服务器迁移到另一个服务器。
- F. 如果使用**HttpSession.setAttribute**增加对象，而且这个对象的类实现了**Serializable.readObject**和**Serializable.writeObject**，会话迁移时，容器会调用这些**readObject**和**writeObject**方法。
- G. 如果会话属性实现了**HttpSessionActivationListener**，容器唯一的需求就是一旦在新服务器上激活会话就要通知监听者。

59

一个Java EE部署描述文件声明了多个过滤器，它们的URL都与一个给定请求匹配，另外部署描述文件中还声明了多个过滤器的<**servlet-name**>标记与同一个请求匹配。

针对这个请求，容器将使用哪些规则调用过滤器，对此以下哪些说法是正确的？（选出所有正确的答案）

- A. 只有<**servlet-name**>匹配的过滤器会得到调用。
- B. 在URL匹配的过滤器中，只会调用第一个。
- C. 在<**servlet-name**>匹配的过滤器中，只会调用第一个。
- D. <**servlet-name**>匹配的过滤器会在URL匹配的过滤器之前调用。
- E. 所有URL匹配的过滤器都会得到调用，不过调用顺序不确定。
- F. 所有URL匹配的过滤器都会得到调用，并按照其出现在DD中的顺序调用。

60

比较servlet初始化参数和上下文初始化参数时，关于二者的共同点以下哪些说法是正确的？（选出所有正确的答案）

- A. 在其各自的DD标记中，都有一个<**param-name**>和一个<**param-value**>标记。
- B. 它们各自的DD标记都直接放在<**web-app**> 标记之下。
- C. 它们用于获取初始化参数值的方法都是**getInitParameter**。
- D. 它们都可以从JSP直接访问。
- E. 只有DD中上下文初始化参数的改变能够访问而无需重新部署Web应用。

61

一个JSP开发人员希望将文件**copyright.jsp**的内容包含到所有主JSP页面中。

以下哪种机制可以做到这一点？（选出所有正确的答案）

- A. <jsp:directive.include file="copyright.jsp" />
- B. <%@ include file="copyright.jsp" %>
- C. <%@ page include="copyright.jsp" %>
- D. <jsp:include page="copyright.jsp" />
- E. <jsp:insert file="copyright.jsp" />

**62**

你在为一个提供电话、电缆和Internet服务的公司开发应用，用来管理客账户。很多页面都包含一个搜索功能。每个页面上的搜索框看上去要一样，不过有些页面限制为只能搜索电话、电缆或Internet账户。

另外有一个名为Search.jsp的JSP：

```

1. <form action="/search.go">
2.   Find ${param.accountType} Account:
3.   <input type="text" name="searchText"/>
4.   <input type="hidden" name="accountType" value="${param.accountType}" />
5.   <input type="submit" value="Search "
6. </form>
```

在需要搜索电缆账户的JSP中应当使用什么标记？

- A. <jsp:include page="Search.jsp" accountType="Cable"/>
- B. <jsp:include page="Search.jsp">
 <jsp:param name="accountType" value="Cable"/>
 </jsp:include>
- C. <jsp:include file="Search.jsp" accountType="Cable"/>
- D. <jsp:include file="Search.jsp">
 <jsp:attribute name="accountType" value="Cable"/>
 </jsp:include>

**63**

测试不同标记和scriptlet如何工作时，一个开发人员创建了以下JSP：

```

1. <% request.setAttribute("name", "World"); %>
2. <!-- Test -->
3. <c:out value='Hello, ${name}' />
```

让这个开发人员惊讶的是，获取他的JSP时，浏览器什么也没有显示。如果开发人员查看页面的HTML源代码，会在输出中发现什么？

- A. <!-- Test -->
- B. <!-- Test -->
 <c:out value='Hello, \${name}' />
- C. <!-- Test -->
 <c:out value='Hello, World' />
- D. 没有任何输出

14

有一个速配服务应用，会询问其用户一系列问题。已经存在一个会话作用域属性，名为**compatibilityProfile**，类型为**HashMap**，所提交的各组问题ID和相应的答案就存储在这个属性中。

给定以下代码：

```
22. <% ((java.util.HashMap)request.getSession().getAttribute("compatibilityProfile")).put(
23.         request.getParameter("questionIdSubmitted"),
24.         request.getParameter("answerSubmitted"));
25. %>
```

如何替换这些代码而不使用scriptlet？（选出所有正确的答案）

- A. <c:map target="\${compatibilityProfile}" key ="\${param.questionIdSubmitted}" value ="\${param.answerSubmitted}"/>
- B. <jsp:useBean id="compatibilityProfile" class="java.util.HashMap" scope="session">
 <jsp:setProperty name="compatibilityProfile" property ="\${param.questionIdSubmitted}" value ="\${param.answerSubmitted}"/>
 </jsp:useBean>
- C. \${compatibilityProfile[param.questionIdSubmitted]} = param.answerSubmitted
- D. <c:set target="\${compatibilityProfile}" property ="\${param.questionIdSubmitted}" value ="\${param.answerSubmitted}"/>

**65**

一个程序员正在为一个Java EE web应用创建过滤器。给定以下代码：

```

7. public class MyFilter implements Filter {
8.     public void init(FilterConfig config) throws FilterException { }
9.
10.    public void doFilter(HttpServletRequest request,
11.                          HttpServletResponse response,
12.                          FilterChain chain)
13.        throws IOException, ServletException { }
14.
15. }

```

必须做哪些修改才能创建一个合法的过滤器？（选出所有正确的答案）

- A. 不需要做任何修改。
- B. 必须增加一个 `destroy()` 方法。
- C. 必须修改 `doFilter()` 方法的体。
- D. 必须修改 `init()` 方法的签名。
- E. 必须修改 `doFilter()` 方法的参数。
- F. 必须修改 `doFilter()` 方法的异常。

**66**

你的公司希望增加一个醒目页面 `SplashAd.jsp`，使用户第一次进入网站时能够看到公司其他产品的广告。在这个新页面上，将为用户提供一个选项，允许用户选中广告页面上的一个显示“Do not show me this offer again”的复选框，并点击一个显示“Continue to My Account”的提交按钮。如果用户提交表单时这个复选框选中，接收Servlet会为用户的浏览器设置一个Cookie，其名为“skipSplashAd”，然后将控制传回主JSP。

主JSP负责将请求转发到这个醒目页面。可以在主页面最前面增加以下哪个代码段，从而当用户未选择这个复选框来避免显示广告时能够为用户显示这个醒目页面？

- A. `<c:if test="${empty cookie.skipSplashAd and pageContext.session.new}">`  
 `<jsp:forward page="SplashAd.jsp"/>`  
`</c:if>`
- B. `<jsp:forward page="SplashAd.jsp" flush="${empty cookie.skipSplashAd}"/>`
- C. `<jsp:redirect page="SplashAd.jsp"/>`
- D. `<jsp:redirect file="SplashAd.jsp"/>`
- E. `<% if(cookie.get("skipSplashAd") == null & session.isNew()) { %>`  
 `<jsp:forward page="SplashAd.jsp"/>`  
`<% } %>`

67

一个程序员想要实现一个**ServletContextListener**。给定以下DD代码片段：

```

101.    <!-- insert tag1 here -->
102.    <param-name>myParam</param-name>
103.    <param-value>myValue</param-value>
104.    <!-- close tag1 here -->
105.    <listener>
106.        <!-- insert tag2 here -->
107.        com.wickedlysmart.MySCListener
108.        <!-- close tag2 here -->
109.    </listener>
```

并给定以下监听者类伪代码：

```

5. // packages and imports here
6. public class MySCListener implements ServletContextListener {
7.     // method 1 here
8.     // shutdown related method here
9. }
```

以下哪些说法是正确的？（选出所有正确的答案）

- A. 这个DD 代码段不可能合法。
- B. tag1应当是<context-param>
- C. tag1应当是<servlet-param>
- D. tag2应当是<listener-class>
- E. tag2应当是<servlet-context-class>
- F. method1应当是initializeListener
- G. method1应当是contextInitialized

68

wickedlysmart 网站有一个正确部署的Java EE web应用和一个部署描述文件，其中包含以下代码：

```
<welcome-file-list>
    <welcome-file>welcome.html</welcome-file>
    <welcome-file>howdy.html</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

以下是这个Web应用目录结构的一部分：

```
MyWebApp
|
|-- index.html
|
|-- welcome
|   |-- welcome.html
|
|-- foobar
    |-- howdy.html
```

如果应用接收到以下两个请求：

<http://www.wickedlysmart.com/MyWebApp/foobar>

<http://www.wickedlysmart.com/MyWebApp>

会提供哪一组响应？

- A. `howdy.html`然后是404。
- B. `index.html`然后是404。
- C. `welcome.html`然后是404。
- D. `howdy.html`然后是`index.html`。
- E. `index.html`然后是`index.html`。
- F. `howdy.html`然后是`welcome.html`。
- G. `welcome.html`然后是`index.html`。

69

你的Web应用有一个合法的DD，其中只包含一个<**security-constraint**>标记。这个标记中有：

- 一个声明为**GET**的HTTP方法

应用中的所有资源都在**directory1**和**directory2**中，另外只定义了**BEGINNER**和**EXPERT**角色。

如果想要限制**BEGINNER**不能使用**directory2**中的资源，应当如何声明url和role标记，对此以下哪些说法是正确的？（选出所有正确的答案）

- A. 应当有一个url标记声明**directory1**，而且有一个role标记声明**EXPERT**。
- B. 应当有一个url标记声明**directory2**，而且有一个role标记声明**EXPERT**。
- C. 应当有一个url标记声明**directory1**，而且有一个role标记声明**BEGINNER**。
- D. 应当有一个url标记声明**directory2**，而且有一个role标记声明**BEGINNER**。
- E. 应当有一个url标记声明**ANY**，而且其role标记声明**EXPERT**，另一个url标记声明**directory2**，其role标记声明**BEGINNER**。
- F. 应当有一个url标记声明这两个目录，其role标记声明**EXPERT**，另一个url标记声明**directory1**，其role标记声明**BEGINNER**。





## 最终模拟测验答案

1 一个程序员为他的Java EE web应用（名为MyWebApp）建立了一个配置正确的目录结构。可以将一个名为myTag.tag的文件放在其中哪两个目录中从而能够由容器正确地访问？（有两个选择）

jsp 8. 本书  
608页

- A. MyWebApp/WEB-INF
- B. MyWebApp/META-INF
- C. MyWebApp/WEB-INF/lib
- D. MyWebApp/WEB-INF/tags
- E. MyWebApp/WEB-INF/TLDs
- F. MyWebApp/WEB-INF/tags/myTags

D和F：标记文件必须放在tags目录或tags的一个子目录中。

2 以下哪些是合法的EL？（选出所有正确的答案）

JSP v2.0 2.3.5节。  
本书396页

- A. \${"1" + "2"} A: "1"和"2"可以转换为long类型，这会输出3。
- B. \${1 plus 2} B: plus不是一个EL操作符。
- C. \${1 eq 2} C是合法的EL，这会输出false。
- D. \${2 div 1} D是合法的，这会输出2.0。
- E. \${2 & 1} E: 与&&或and不同，&不是一个合法的EL操作符。
- F. \${"head"+ "first"} F: 不能用+操作符连接Strings。EL无法将String值强制转换为Double类型。

3 一个Java论坛网站的TLD中包含以下标记定义：

JSP v2.0 7.4.1.1节  
本书476-480页

```
<tag>
  <name>avatar</name>
  <tag-class>hf.AvatarTagHandler</tag-class>
  <body-content>empty</body-content>

  <attribute>
    <name>userId</name>
    <required>true</required>
    <rtpvalue>true</rtpvalue>
  </attribute>
  <attribute>
    <name>size</name>
    <required>false</required>
    <rtpvalue>false</rtpvalue>
  </attribute>
</tag>
```

关于AvatarTagHandler以下哪些说法是正确的？这里假设它扩展了SimpleTagSupport，并输出将显示用户avatar图像的HTML（选出所有正确答案）

- A. 这个类应当有一个size成员，它至少有一个设置方法。
- B. 代码中不需要size变量，因为TLD指出这个变量是不必要的。
- C. 需要一个覆盖的**doTag** 生命周期方法。
- D. 需要一个覆盖的**doStartTag** 生命周期方法。
- E. 这个类必须重载已实现的所有生命周期方法，这些重载的版本中，对应TLD中定义的每一个属性都要包括一个额外的参数。在这里只有一个需要重载的生命周期方法。

A: 标记处理器应当保存size，尽管使用这个标记时size并不必要。

C: 除非覆盖了这个方法并提供了所需的行为，否则什么也做不了。SimpleTagSupport中有一个默认实现，但是它什么也不做。

D: doStartTag 对应于传统标记处理器。

E: 简单标记处理器只有一个生命周期方法，容器无法识别它的任何重载版本。

给定：

```

20. foo.User user = new foo.User();
21. user.setFirst(request.getParameter("firstName"));
22. user.setLast(request.getParameter("lastName"));
23. user.setStreet(request.getParameter("streetAddress"));
24. user.setCity(request.getParameter("city"));
25. user.setState(request.getParameter("state"));
26. user.setZipCode(request.getParameter("zipCode"));
27. request.setAttribute("user", user);

```

如果在一个JSP中放入以下代码段，其中哪些代码段能够替换以上Servlet代码？（选出所有正确的答案）

- A. <jsp:useBean id="user" type="foo.User"/> A和B都使用了type属性，要求这个bean保存到某个作用域。尽管它们使用了class属性，但这不足以填写bean的所有性质。
- B. <jsp:useBean id="user" type="foo.User">
 <jsp:setProperty name="user" property="\*"/>
 </jsp:useBean>
- C. <jsp:useBean id="user" class="foo.User">
 <jsp:setProperty name="user" property="first" param="firstName"/>
 <jsp:setProperty name="user" property="last" param="lastName"/>
 <jsp:setProperty name="user" property="street" param="streetAddress"/>
 <jsp:setProperty name="user" property="city"/>
 <jsp:setProperty name="user" property="state"/>
 <jsp:setProperty name="user" property="zipCode"/>
 </jsp:useBean>
- D. <jsp:useBean id="user" class="foo.User">
 <jsp:setProperty name="user" property="\*"/>
 <jsp:setProperty name="user" property="first" param="firstName"/>
 <jsp:setProperty name="user" property="last" param="lastName"/>
 <jsp:setProperty name="user" property="street" param="streetAddress"/>
 </jsp:useBean>

C和D：当名不匹配时，必须分别用各个<jsp:setProperty>标记将参数映射到bean性质。对于匹配的参数名，可以用property="\*"自动将其统统传入bean。

5

比较业务委托对象和服务定位器对象的优点、局限性和用法时，以下哪些说法是正确的？（选出所有正确的答案）

- A. 它们都有可能建立网络调用。
- B. 它们都有可能调用传输对象中的方法。
- C. 它们都有可能由一个控制器对象直接调用。
- D. 服务定位器通常可以认为是业务委托的一个服务器。
- E. 如果实现时都建立了缓存，那么数据过时问题对于业务委托来说更为严重。

A: 通常业务委托会让另一个对象建立网络调用。

B: 通常服务定位器不会使用传输对象。

C: 通常控制器会建立业务委托请求，而业务委托会在必要时建立服务定位器请求。

6

关于创建会话监听者，以下哪些说法是正确的？（选出所有正确的答案）

- A. 它们都在DD中声明。
- B. 并不是所有会话监听者都必须在DD中声明。
- C. 用来声明会话监听者的DD标记是<listener>。
- D. 用来声明会话监听者的DD标记是<session-listener>。
- E. 用来声明会话监听者的DD标记放在<web-app>标记中。
- F. 用来声明会话监听者的DD标记放在 <servlet>。

A: HttpSessionBindingListener  
不在DD中声明。Servlet附录B,  
本书256~263页

C: 希望你不用专门去记就能知道这一点。

F: 记住会话可能跨多个  
servlet。

7

有些用户抱怨说，在一台机器上打开两个浏览器窗口时，如果这两个窗口同时访问应用会发生一些奇怪的事情。你想测试一些浏览器，查看是否会在多个窗口间共享一个会话。为此，你决定在一个JSP中输出JSESSIONID。假设你的测试浏览器上启用了cookie，如何实现这个测试？（选出所有正确的答案）

- A. \${cookie.JSESSIONID}
- B. \${cookie.JSESSIONID.value}
- C. \${cookie["JSESSIONID"]["value"]}
- D. \${cookie.JSESSIONID["value"]}
- E. \${cookie["JSESSIONID"].value}
- F. \${cookieValues[0].value}

A: 计算为一个Cookie对象，它会输出Cookie对象的引用，而不是其内部的值。

B, C, D, E: cookie EL隐式对象是Cookie对象的一个映射。这些选项都可以得到JSESSIONID Cookie，并调用其getValue()方法。

F: cookieValues不是一个EL隐式对象。

JSP v2.0 2.2.3节  
Servlet v2.4的7.1.1节  
本书232和390页

8

哪个隐式对象可以访问**ServletContext**的属性?

- A. **server** A, B和E都不对，因为这些都不是合法的JSP隐式对象名。
- B. **context**
- C. **request** C不对，因为'request'隐式对象只能访问请求作用域属性。
- D. **application** D是对的。'application'隐式对象等价于**ServletContext**。
- E. **servletContext**

9

哪些方法在**HttpServlet**中? (选出所有正确的答案)HTTP 1.1  
本书第4章

- A. **doGet**
- B. **doTrace**
- C. **doError** C: 也不存在HTTP ERROR方法。
- D. **doConnect** D: HTTP有一个CONNECT方法，但是与其他HTTP方法不同，这里有一个例外，这是唯一一个没有对应到HttpServlet的HTTP方法。
- E. **doOptions**

10

你决定你的Web应用中的某些功能将要求用户是注册会员。另外，你的Web应用有时会处理用户希望你保密的一些用户数据。

本书677~684页

以下哪些说法是正确的? (选出所有正确的答案)

- A. 只有在应用验证了用户的口令之后才能保证所传输数据的机密性。
- B. 在Java EE容器保证的各种认证中，只有基本认证(BASIC)、摘要认证(Digest)和基于表单的认证( Form Based) 要求用户名与口令匹配。
- C. 不论你使用何种类型的Java EE认证机制，只有在请求受限资源时才会被激活。
- D. Java EE保证的各种类型的认证都能提供高强度的数据安全性，而不需要实现支持安全特性。

以下是一个Java EE DD中的某标记中取出的代码段：

Servlet 12.  
本书684

```

343.   <web-resource-collection>
344.     <web-resource-name>Recipes</web-resource-name>
345.     <url-pattern>/Beer/Update/*</url-pattern>
346.     <http-method>POST</http-method>
347.   </web-resource-collection>

...
367.   <auth-constraint>
368.     <role-name>Member</role-name>
369.   </auth-constraint>

...
385.   <user-data-constraint>
386.     <transport-guarantee>CONFIDENTIAL</transport-guarantee>
387.   </user-data-constraint>
```

以下哪些说法是正确的？（选出所有正确的答案）

- A. Java EE DD可以包含某一个标记，在其中以上所有这些标记都能合法地共存。
- B. 在选项A中所述的单个标记中还可以存在更多`<auth-constraint>`实例。
- C. 在选项A中所述的单个标记中还可以存在更多`<user-data-constraint>`实例。
- D. 在以上所述的`<web-resource-collection>`标记中还可以存在更多`<url-pattern>`实例。
- E. 与选项A中所述单个外围标记同类型的其他标记可以有与以上标记相同的`<url-pattern>`。
- F. 这个标记表明，Web应用的授权、认证和数据完整性等安全特性都已经得到声明。

C. 合法的  
`<security-constraint>`标记就可以作为此类外围标记，但它只能声明一种数据完整性。

**12**

你声明了一个JSP文档，它生成一个动态SVG图像（由一个XML文档结构表示）。这个JSP必须将HTTP响应首部'Content-Type'声明为'image/svg+xml'，从而使web浏览器将响应显示为一个SVG图像。

以下哪个JSP代码段声明这个JSP文档是一个SVG响应？

- A. <%@ page contentType='image/svg+xml' %>
- B. <jsp:page contentType='image/svg+xml' />
- C. <jsp:directive.page contentType='image/svg+xml' />
- D. <jsp:page.contentType>image/svg+xml</jsp:page.contentType>

C是对的，因为'jsp:directive.page'是适当的标准JSP文件。

A不对，因为标准JSP指令语法'<%@ ... %>'在JSP文档格式中不合法。

B不对，因为JSP文档中不存在'jsp:page'标准标记。

D不对，因为JSP文档中不存在'jsp:page.contentType'标准标记。

**13**

给定一个JSP页面中有以下代码：

JSP v 2.0 1.5.2页。  
本书304页

```
<%-- out.print("Hello World"); --%>
```

它的HTML输出是什么？

- A. Hello World
- B. out.print("Hello World");
- C. <!-- Hello World -->
- D. 这行代码不会生成任何输出。

**14**

关于HTTP会话支持，以下哪些说法是正确的？（选出所有正确的答案）

Servlet 7.  
本书231-240页

- A. Java EE 容器必须支持HTTP cookie。
- B. Java EE 容器必须支持URL重写。
- C. Java EE 容器必须支持安全套接字层。
- D. Java EE 容器必须支持HTTP会话，甚至包括不支持cookie的客户。
- E. Java EE 容器必须能够识别HTTP终止信号，发出HTTP终止信号则指示一个客户会话不再活动。

B: 如果cookie不可用，总会将URL重写作为退路。不过这并不是对容器的要求。

E: HTTP没有会话终止信号。

## 15

你的公司购买了一个第三方JavaScript库（用于构建菜单）的许可。你的开发小组错误地滥用了这个库，以至于遭遇无数错误，另外用户坚持认为某些菜单项只有得到授权的用户才能看到。一个使用简单标记处理器的定制标记库可以避免开发人员产生JavaScript语法错误，并且能够提供用户所需的安全特性。

一次设计会议之后，小组的领导人声明他希望菜单如下所示：

```
<menu:main>
    <menu:headItem text="My Account" url="/myAccount.do"/>
    <menu:headItem text="Transactions">
        <menu:subItem text="Incoming" url="/incomingTx.do"/>
        <menu:subItem text="Outgoing" url="/outgoingTx.do"/>
        <menu:subItem text="Pending" url="/pendingTx.do"
            requireRole="accountant"/>
    </menu:headItem>
    <menu:headItem text="Admin" url="/admin.do"
        requireRole="admin"/>
</menu:main>
```

你希望完全由外部标记处理器负责生成输出，这里假设很容易将显示逻辑集中在一起。外部`<menu:main>`标记处理器需要访问其子孙标记来完成这个任务。以下哪个选项可以提供最佳方法？

- A. 每个内部标记应当直接向其父标记注册。  
直接父标记可以将其子标记存储在一个有序集合中。
- B. 每个内部标记应当直接向外部标记处理器注册，  
外部标记处理器可以将它们都存储在一个`HashSet`中。
- C. 不同于传统标记，`SimpleTagSupport`提供了方法  
`findDescendentWithClass()`和`getChildren()`，  
这样外部标记完全可以访问其子标记，而无需另外编写代码。
- D. 让各个内部标记将自己保存为一个页面作用域属性，  
其`text`值作为属性的键。

A是最简单的解决方案，因为这将为标记创建一个简单的树结构，使`<menu:main>`能够访问其所有子孙标记。

B和D并没有为外部标记提供内部标记如何建构的任何线索。

C这些方法根本不存在。只能使用API中的`findAncestorWithClass()`和`getParent()`方法。

16

请求一个JSP页面时, JSP生命周期中的哪个阶段可能导致返回一个HTTP 500状态码? (选出所有正确的答案)

A. JSP 页面编译阶段

A是对的, 这是因为, 如果JSP servlet代码无法编译, 容器就必须生成一个服务器端错误。

B. 执行服务方法时

B是对的, 因为JSP中抛出的任何运行时异常都必须由容器处理, 它必须生成一个服务器端错误。

C. 执行撤销方法时

D. 执行初始化方法时

D是对的, 因为如果初始化方法抛出一个异常, 容器将无法向JSP发出请求, 而必须发送一个服务器端错误。

C不对; 撤销方法不会导致一个500错误。

17

给定`session`是一个合法`HttpSession`的引用, `myAttr`是绑定到`session`的对象的名, 可以用以下哪个方法解除对象与会话的绑定? (选出所有正确的答案)

API, 本书第6章

A. `session.unbind();`

B. `session.invalidate();`

C. `session.unbind(myAttr);`

D. `session.remove(myAttr);`

E. `invalidate()`用于将绑定到会话的所有对象解除绑定。

E. `session.invalidate(myAttr);`

F. `session.removeAttribute(myAttr);` F. `removeAttribute()`用于解除一个对象的绑定。

G. `session.unbindAttribute(myAttr);`

18

如果`req`是一个`HttpServletRequest`的引用, 当前没有会话, 关于`req.getSession()`以下哪种说法是正确的? (选出所有正确的答案)

API, 本书232~233页

A. 调用`req.getSession()`将返回null。

A和B: 这些情况下会创建一个新的会话。

B. 调用`req.getSession(true)`将返回null。

C. 调用`req.getSession(false)`将返回null。

D. 调用`req.getSession()`将返回一个新会话。

E. 调用`req.getSession(true)`将返回一个新会话。

F. 调用`req.getSession(false)`将返回一个新会话。

19

遗留代码中有一个传统标记处理器。代码的作者写了一个处理器，将其标记体计算100次，用来测试其他生成随机内容的标记。

TagSupport API  
JSP v2.0 13.1节  
本书536~537页

给定以下代码：

```

06. public class HundredTimesTag extends TagSupport {
07.     private int iterationCount;
08.     public int doTag() throws JspException {
09.         iterationCount = 0;
10.         return EVAL_BODY_INCLUDE;
11.     }
12.
13.     public int doAfterBody() throws JspException {
14.         if(iterationCount < 100){
15.             iterationCount++;
16.             return EVAL_BODY_AGAIN;
17.         }else{
18.             return SKIP_BODY;
19.         }
20.     }
21. }
```

A: 标记处理器是线程安全的，所以完全可以在其中存储状态。

关于以上代码哪个说法是正确的？

- A. 标记处理器不是线程安全的，所以如果多个用户同时访问这个页面 `iterationCount` 可能不同步。
- B. `doAfterBody` 方法永远不会得到调用，因为它不是标记处理器生命周期的一部分。开发者应当扩展 `IterationTagSupport` 类，将这个方法括在生命周期中。
- C. `doTag` 方法应当是 `doStartTag`。按照以上代码，会调用 `TagSupport` 的默认 `doStartTag`，它只是返回 `SKIP_BODY`，这会导致 `doAfterBody` 永远也不会得到调用。
- D. `doAfterBody` 返回 `EVAL_BODY_AGAIN` 时，会再次调用 `doTag` 方法。`doTag` 方法将 `iterationCount` 重置为 0，导致一个无限循环，并抛出一个 `java.lang.OutOfMemoryError`。

B: `IterationTagSupport` 不是一个实际的类。`doAfterBody` 方法是 `IterationTag` 接口的一部分，`TagSupport` 实现了这个接口。

C: 改变这个方法名就能修正问题。如果刚好使用 Java 5 SE，可以在这些生命周期方法上使用 `@Override` 注释来确保诸如此类的错误不发生。这是一个很好的想法。

D: 即使如选项C改变了方法名也不会出现无限循环，因为传统标记生命周期不会多次调用 `doStartTag`。

20

给定一个Web应用DD中的以下代码片段：

APJ, 本书244~245页

```

72.  <session-config>
73.    <session-timeout>10</session-timeout>
74.  </session-config>

```

另外，给定`session`是一个合法`HttpSession`的引用，其相应的servlet代码如下：

```
30.    session.setMaxInactiveInterval(120);
```

执行第30行后，以下哪个说法是正确的？（选出所有正确的答案）

- A. DD代码片段不合法。
- B. 调用`setMaxInactiveInterval`将修改`<session-timeout>`标记中  
的值。  
B: 这个方法只覆盖  
这个会话的超时值。
- C. 根据以上条件不可能确定会话的超时时限。
- D. 如果容器在2个小时内没有接收到对这个会话的客户请求，容器将置会话  
无效。
- E. 如果容器在2分钟内没有接收到对这个会话的客户请求，容器将置会话无效。  
E: 这个方法的参  
数表示秒数，不过  
标记中的值表示分  
钟数。
- F. 如果容器在10秒内没有接收到对这个会话的客户请求，容器将置会话无效。
- G. 如果容器在10分钟内没有接收到对这个会话的客户请求，容器将置会话  
无效。

21

你已经创建了一个合法的目录结构，并为你的Java EE Web应用创建了一个合法的  
**WAR**文件。给定以下条件：Servlet 9,  
本书612页

- **ValidApp.war**是**WAR**文件名。
- **WARdir**表示每个**WAR**文件中必须有的目录。
- **APPdir**表示每个web应用中必须有的目录。

以下哪个说法是正确的？

- A. **WARdir**的具体名是无法预知的。
- B. 应用名是无法预知的。  
A: 这个目录名为META-INF。
- C. 在这个目录结构中，**APPdir**将在**WARdir**中。  
B: 通常容器对应用命名时会使用  
WAR文件名，不过并不一定如此。
- D. 在这个目录结构中，应用的部署描述文件将在**WARdir**所在的目录中。
- E. 将应用置于一个**WAR**文件时，这样可以为容器提供一个选择，能够完成额外的  
运行时检查，否则这一点是无法保证的。  
E: WAR文件只是为完成  
额外的部署时检查提供  
了一个选择。

22

比较HTTP GET和HTTP POST时，以下哪些说法是正确的？（选出所有正确的答案）

- A. 只有HTTP GET 是幂等的。
- B. 这两个方法都要求在HTML表单标记中有一个显式声明。
- C. 只有HTTP POST 可以在一个请求中支持多个参数。
- D. 二者都支持发送多个值的单参数请求。
- E. 只有HTTP POST 请求应当通过覆盖servlet的service()方法来处理。

A: 如果一个表单没有显式声明一个方法，就会认为是GET方法。

D: 这两个方法都能处理这种情况。

E: 要参加考试，你要记住绝对不能覆盖service()方法。

23

给定一个servlet中的以下代码。

Serv: 附录B.  
本书150页

```
82. String s = getServletConfig().getInitParameter("myThing");
```

哪个DD片段将把s赋值为“myStuff”？

- A. <init-param>
 

```
<param>myThing</param>
      <value>myStuff</value>
    </init-param>
```
- B. <init-param>
 

```
<name>myThing</name>
      <value>myStuff</value>
    </init-param>
```
- C. <init-param>
 

```
<param-name>myThing</param-name>
      <param-value>myStuff</param-value>
    </init-param>
```

C是<init-param>标记的正确语法。
- D. <servlet-param>
 

```
<name>myThing</name>
      <value>myStuff</value>
    </servlet-param>
```
- E. <servlet-param>
 

```
<param-name>myThing</param-name>
      <param-value>myStuff</param-value>
    </servlet-param>
```

24

给定一个String存储为某个作用域中一个名为**accountNumber**的属性，以下哪个（哪些）scriptlet将输出这个属性？

- A. `<%= pageContext.findAttribute("accountNumber") %>`
- B. `<%= out.print("${accountNumber}") %>`
- C. `<% Object accNum = pageContext.getAttribute("accountNumber"); if(accNum == null){ accNum = request.getAttribute("accountNumber"); } if(accNum == null){ accNum = session.getAttribute("accountNumber"); } if(accNum == null){ accNum = servletContext.getAttribute("accountNumber"); } out.print(accNum); %>`
- D. `<% requestDispatcher.include("accountNumber"); %>`

A. 如果必须使用scriptlet，这是最容易的办法。

B. scriptlet中的EL不会计算。这完全是scriptlet的一部分，非法使用，所以不要认为这是一个技巧！

C. 非常接近了。servletContext不是一个合法的EL表达式对象。应该使用application。

D. requestDispatcher不是一个隐式对象。即使有这样一个隐式对象，这也是错的。

25

你得到了一个遗留的JSP Web应用，其中有大量脚本代码。你的经理要求必须对每一个JSP重构，删除其中的脚本代码。他希望你能保证JSP代码基中不再出现scriptlet代码，并让Web容器强制一种“无脚本”策略。

JSP Version 2.0  
3.3.3节

哪个**web.xml**配置元素可以完成这个目标？

- A. `<jsp-property-group>
 <url-pattern> *.jsp </url-pattern>
 <permit-scripting> false </permit-scripting>
</jsp-property-group>`
- B. `<jsp-config>
 <url-pattern> *.jsp </url-pattern>
 <permit-scripting> false </permit-scripting>
</jsp-config>`
- C. `<jsp-property-group>
 <url-pattern> *.jsp </url-pattern>
 <scripting-invalid> true </scripting-invalid>
</jsp-property-group>`
- D. `<jsp-config>
 <url-pattern> *.jsp </url-pattern>
 <scripting-invalid> true </scripting-invalid>
</jsp-config>`

A不对，因为`<permit-scripting>`不是一个合法的配置元素。

B不对，因为`<jsp-config>`和`<permit-scripting>`都不是合法的配置元素。

D不对，因为`<jsp-config>`不是一个合法的配置元素。

26

给定以下代码：

```

01. <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
02.
03. <%
04. java.util.List books = new java.util.ArrayList();
05. // add line here
06. request.setAttribute("myFavoriteBooks", books);
07. %>
08.
09. <c:choose>
10. <c:when test="${not empty myFavoriteBooks}">
11.     My favorite books are:
12.     <c:forEach var="book" items="${myFavoriteBooks}">
13.         <br/> * ${book}
14.     </c:forEach>
15. </c:when>
16. <c:otherwise>
17.     I have not selected any favorite books.
18. </c:otherwise>
19. </c:choose>
```

如果独立地插入到第5行，以下哪些代码行会导致显示出c:otherwise标记中的文本？  
 (选出所有正确的答案)

- A. books.add("");      A. B和D都向books List增加某些内容，使之非空。
- B. books.add(null);      C. 清空已经为空的list。
- C. books.clear();      D. books.add("Head First");

E: 置List引用为null值。这将使empty操作得  
结果为true。

27

你在开发一个管理企业名单目录的应用。

JSP v 2.0 2.6节  
本章388~391页

给定以下代码：

```

29. <c:forEach var="phoneNumber" items='${company.
   contactInfo.phoneNumbers}'>
30.   <c:if test='${verify:isTollFree(phoneNumber)}'>
31.     
32.   </c:if>
33.   ${phoneNumber}<br/>
34. </c:forEach>

```

以上代码段在免计费电话号码前增加一个特殊的图标。关于这个代码段中的EL函数，以下哪个说法肯定是正确的？

- A. EL函数必须声明为公共（public）和静态（static）
- B. EL函数不能返回任何值，而且必须声明为void
- C. EL函数TLD中的<uri>值必须是 Verify
- D. 实现EL函数的类必须命名为Verify
- E. 如果 phoneNumber 是一个String，TLD中的<function-signature> 值应当是 isTollFree(String)

A: 所有EL函数都必须声明为 public 和 static。

B: EL函数应当返回一个布尔值，从而能够用于&lt;c:if&gt;标记。

C: &lt;uri&gt;值应当与 JSP taglib 指令中声明的值匹配，在这里没有给出。

D: 在 TLD 中使用&lt;function-class&gt; 匹配完全限定类名，不必与 EL 函数使用的特定命名约定匹配。

E: <function-signature> 要求声明一个返回类型。它还要求所有类类型都必须是完全限定类名，所以 String 应当是 java.lang.String。

28

HttpServletRequest 的哪些方法可以获取请求体？（选出所有正确的答案）

- A. getReader() A: getReader() 将体读取为字符数据。
- B. getStream()
- C. getInputReader()
- D. getInputStream() D: 这个方法将体读取为二进制数据。
- E. getServletReader()
- F. getServletStream()

29

给定一个 Java EE web应用，其中，对于以下浏览器请求：

Serv 11. 本书  
616页

<http://www.wickedlysmart.com/MyApp/myDir/DoSomething>

一将由应用中的一个servlet处理，以下哪3种说法是正确的？

(选出3个正确的选项)

- A. 部署描述文件必须包含按指定要求处理这个请求的指令。
- B. 可以按指定要求处理这个请求，而部署描述文件中无需相关的指令。
- C. 处理这个请求的servlet必须命名为**DoSomething.class**。
- D. 根据所提供的信息无法预知servlet名。
- E. 这个应用必须包含一个名为**myDir**的目录。
- F. 根据所提供的信息无法预知servlet所在目录的目录名。

A: DD中必须指定一个  
<servlet-mapping>标记。

C和E: myDir和  
DoSomething都  
是只有DD知道  
的虚拟名。

30

你的Web应用有一个合法的部署描述文件，其中只定义了**student**和**sensei**安全角色。部署描述文件包含两个安全约束，声明了对同一个资源的安全约束。第一个安全约束包含：

```
234.    <auth-constraint>
235.        <role-name>student</role-name>
236.    </auth-constraint>
```

Servlet 12.8.  
本书668~669页

第二个安全约束包含：

```
251.    <auth-constraint/>
```

以下哪个说法是正确的？(选出所有正确的答案)

A,B和C: 第二个标记  
为“空”，这说明任  
何角色都不能使用这  
个资源。

- A. 根据现在的部署描述文件，这两个角色都可以访问这个受限资源。
- B. 根据现在的部署描述文件，只有**sensei**用户能访问这个受限资源。
- C. 根据现在的部署描述文件，只有**student**用户能访问这个受限资源。
- D. 如果去掉第二个<auth-constraint>标记，这两个角色都可以访问这个受限资源。
- E. 如果去掉第二个<auth-constraint>标记，只有**sensei**用户能访问这个受限资源。
- F. 如果去掉第二个<auth-constraint>标记，只有**student**用户能访问这个受限资源。

31

以下哪个定制标记肯定会失败？（选出所有正确的答案）

JSP v2.0 (~31)  
本书第10章

- A. <mine:border>  
 <mine:photos album="\${albumSelected}">  
 </mine:border>  
 </mine:photos>
- B. <mine:border>  
 <mine:photos album="\${albumSelected}" />  
 </mine:border>
- C. <mine:border>  
 \${albumSelected.title}  
 <mine:photos>\${albumSelected}</mine:photos>  
 </mine:border>
- D. <mine:photos includeBorder="\${userPreference.border}"  
 album="\${albumSelected}" />

A: 标记&lt;mine:photos&gt;没有正确地嵌套。

B, C和D都可能是定制标记的合法用法。

32

你的n层Web应用使用了一些Java EE模式（它们是应用希望访问远程注册表时最常使用的模式）。这些模式有哪些好处？（选出所有正确的答案）

核心jee 315~318页  
本书754页

- A. 增加内聚度
- B. 性能更优
- C. 可维护性更好
- D. 减少网络流量

这里使用的模式是业务委托和服务定位器。通过结合使用这两种模式，各个组件的职责更专一，体系结构发生变化时，维护也将更容易。

浏览器功能交互性更强

D: 如果选了选项D也是可以的—服务定位器实现有缓存时，确实可以减少网络流量。不过，缓存总会带来自身的一些缺点，所以这不是最标准的解决方案。

1

33

的生命周期，以些哪些说法总是正确的？（选出所有正确的答案）

API, Servlet,  
本书97~99页

- 为servlet编写构造函数。
- 覆盖servlet的init()方法。
- 覆盖servlet的doGet()方法。
- E. 不重写servlet的doPost()方法。
- 不能覆盖servlet的service()方法。

B和F中的方法通常在servlet需要创建和撤销servlet使用的资源（如数据库连接）时覆盖。

附录 的destroy()方法。

Serv 9, 本书  
612~613页

34 给定一个Java EE .war 文件目录结构的一部分：

```

MyApp
| -- META-INF
|   | -- MANIFEST.MF
|   | -- web.xml
|
| -- WEB-INF
|   | -- index.html
|   | -- TLDs
|   | -- Header.tag

```

要使这个结构合法，而且资源可访问，需要做哪些修改？（选出所有正确的答案）

- A. 不需做任何修改。
- B. **web.xml**文件必须移动。 B: **web.xml**必须放在WEB-INF目录中。
- C. **index.html**文件必须移动。 C: 没错，不过客户不能直接访问。
- D. **Header.tag**文件必须移动。 D: **.tag**文件必须放在结构树中的WEB-INF/tags/部分。
- E. **MANIFEST.MF**文件必须移动。
- F. **WEB-INF**目录必须移动。
- G. **META-INF**目录必须移动。

核心jee 166页。  
本书第14章

35

你考虑在你的Java EE n层应用中实现某种MVC。以下哪些说法是正确的？

(选出所有正确的答案)

- A. 这个设计通常为业务委托对象服务。 A: 业务委托为控制器服务。
- B. 通常通过缓存位于远程的数据来减少网络流量。 B: 支持MVC的对象可以缓存，不过MVC本身通常不会。
- C. 这个设计目标可以简化与异构资源注册表的通信。 C: 这是服务定位器的任务。
- D. 尽管MVC解决方案有很多优点，但通常会增加设计复杂性。
- E. 前端控制器模式和Struts都可以认为是这种设计目标的解决方案。
- F. 这个设计将使你能够轻松地重新结合请求和响应处理器。 F: 这是拦截过滤器的任务，这种模式可以与MVC合作，但二者是不同的模式。

**36**

给定一个JSP页面中有以下代码行：

&lt;% List myList = new ArrayList(); %&gt;

能用以下哪个JSP代码段导入这些数据类型？（选出两个正确答案）

 A. <%! import java.util.\*; %>A不对，因为JSP声明标记不能  
用来将import语句插入到转换  
后的servlet代码中。 B. <%@ import java.util.List java.util.ArrayList %>B不对，因为没有  
import JSP指令。 C. <%@ page import='java.util.List,java.util.ArrayList' %> D. <%! import java.util.List; import java.util.ArrayList; %>D不对，因为JSP声明标记  
不能用来将import语句插入  
到转换后的servlet代码中。 E. <%@ page import='java.util.List' %> <%@ page  
import='java.util.ArrayList' %>

E是对的，因为page指令的import属性可以多次指定。

**37**

你要完成一个任务，向公司的Java EE Web应用增加一些安全特性。具体地，需要创建多种不同类别的用户，另外根据用户的类别，需要对他们做出限制，从而只能使用应用中的某些页面。为了限制用户的访问，必须确定用户确实是他们所声称的那个人。

Servlet 12.  
本书第12章

以下哪些说法是正确的？（选出所有正确的答案）

 A. 如果需要验证用户确实是他们所声称的那些人，就必须使用应用的部署描  
述文件来实现这个需求。A: 还可以通过程序  
方式完成认证。 B. 必须使用Java EE的授权功能来确定用户是他们所声称的那些人。B: 这是关于认证的问题  
(而不是授权)。 C. 为了帮助你确定用户确实是他们所声称的那些人，可以使用部署描述文件的  
<login-config> 标记。 D. 为了帮助你确定用户确实是他们所声称的那些人，可以使用部署描述文件的  
<user-data-constraint> 标记。D: 这个标记用于实  
现数据完整性。 E. 根据你使用的方法，要确定用户是他们所声称的那些人，可能需要包含一个  
“realm”。

38

ValidApp是一个Java EE应用，有一个合法的目录结构。ValidApp将.gif图像文件包含在目录结构的三个位置中。

Servlet 9.  
本书614页

- ValidApp/imageDir/
- ValidApp/META-INF/
- ValidApp/WEB-INF/

在以下哪些位置上用户可以直接访问这些.gif文件？

- A. 只有ValidApp/META-INF/
- B. 只有ValidApp/imageDir/ B:如果一个客户试图访问WEB-INF或META-INF中的文件，容器会返回一个404。
- C. 以上所有位置
- D. 只有ValidApp/imageDir/ 和 ValidApp/WEB-INF/
- E. 只有ValidApp/imageDir/ 和 ValidApp/META-INF/

39

给定req是一个合法HttpServletRequest的引用，有以下代码：

API

```

13. String[] s = req.getCookies();
14. Cookie[] c = req.getCookies();
15. req.setAttribute("myAttr1", "42");
16. req.setAttribute("myAttr2", 42);
17. String[] s2 = req.getAttributeNames();
18. String[] s3 = req.getParameterValues("attr");

```

哪些代码行无法编译？（选出所有正确的答案）

- A. 第13行 A:getCookies()返回一个Cookie数组。
- B. 第14行
- C. 第15行 D:setAttribute()取一个String和一个Object参数。
- D. 第16行 对于Java 5, 42可以装箱为一个Object。
- E. 第17行 E:getAttributeNames()返回一个Enumeration。
- F. 第18行

我们知道，这种问题确实需要“死记”，不过遗憾的是，考试中很可能遇到这种问题。



40

名为 **Products.tag** 的标记文件显示了一个产品列表。

给定标记文件中的以下代码段：

```
1. <%@ attribute name="header" required="false" rtexprvalue="false" %>
2. <%@ attribute name="products" required="true" rtexprvalue="true" %>
3. <%@ tag body-content="tagdependent" %>
```

以下哪些是这个标记文件的合法使用？（选出所有正确的答案）

- A. <display:Products header="Shopping Cart" products="\${shoppingCart}" />
- B. <display:Products header="Wish List" products="\${wishList}" body-content="\${body}" /> B: body-content不是一个合法的属性
- C. <display:Products header="Similar Products" products="\${similarProductCustomers who bought this item also bought: \${tagdependent body-content}}> C: 允许有体，因为tag标记中有 tagdependent body-content值。
- D. <display:Products header='<%= request.getParameter("listType") %>' /> D: products是一个必要属性。另外，header不能包含scriptlet，因为已经定义header的 rtexprvalue设置为false。

41

你在参与一个项目，要从一个大型银行遗留Web应用的JSP中去除scriptlet页。你遇到以 JSP v2.0, 3.4节，  
下代码： 本书370~378页

```
<% if((com.yourcompany.Account)request.  
       getAttribute("account")).isPersonalChecking()) {  
%>  
    Checking that fits your lifestyle.  
<% } %>
```

如何使用JSTL替换以上代码？（选出所有正确的答案）

- A. <c:if test='\${account.personalChecking}'>Checking  
that fits your lifestyle.</c:if>
- B. <c:if test='\${account["personalChecking"]}'>Checking  
that fits your lifestyle.</c:if>
- C. <c:if test="\${account['personalChecking']}>Checking  
that fits your lifestyle.</c:if>
- D. <c:if test='\${account.isPersonalChecking}'>Checking  
that fits your lifestyle.</c:if>

A: 找到名为account的属性，并在Account对象上调用isPersonalChecking()。

B 和 C: 注意，单引号和引号都是可以的，不过如果EL位于一个要计算的表达式中，EL中的引号必须与包围这个EL所用的引号不同。这条规则对模板文本标记不适用，它们不会计算，如<a href="#">`$initParam["coi-email"]`

D: 将在Account上查找getIsPersonalChecking方法，未找到时会抛出一个异常。

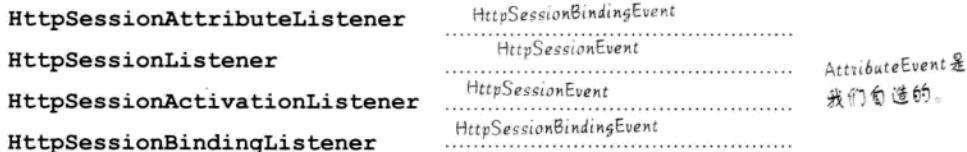
42

给定以下事件类型：

- HttpSessionEvent
- HttpSessionBindingEvent
- HttpSessionAttributeEvent

APJ.  
本书264页

将上面的事件类型与其各自的监听者接口匹配。(注意,一个事件类型可以与多个Listener匹配)



43

关于servlet的生命周期以下哪些说法是正确的? (选出所有正确的答案)

serv 2. 本书  
97~101页

- A. 接收一个新请求时, **service()** 方法是容器最先调用的方法。
- B. **doPost()**或**doGet()**完成一个请求之后会调用**service()**方法。
- C. 每次调用**doPost()**时, 会在其自己的线程中运行。
- D. 每次**doGet()**调用完成后会调用**destroy()**方法。
- E. 容器为每个客户请求建立一个单独的线程。

A: init()方法会首先调用。  
B: service()方法会调用  
doGet()或doPost()

D: 容器决定删除一个  
servlet时才会调用destroy()。

44

什么情况下JSP会得到转换? (选出所有正确的答案)

JSP v2.0 1.1.4节  
本书 308页

- A. 开发人员编译src文件夹中的代码时。
- B. 启动应用时。
- C. 用户第一次请求JSP时。
- D. 调用**jspDestroy()**之后JSP会得到转换。

A: JSP不在src文件夹中, 开发人员  
不会像编译代码那样编译JSP。

B和C: 第一次部署到JSP容器之后,  
而且在处理客户的页面请求之前,  
转换可能发生在此期间的任意时刻。

D: 不会导致对同一页面再完成一次转换。

**45**

以下是取自一个合法doGet()方法的代码段:

```

12.     OutputStream os = response.getOutputStream();
13.     byte[] ba = {1,2,3};
14.     os.write(ba);
15.     RequestDispatcher rd = request.getRequestDispatcher("my.jsp");
16.     rd.forward(request, response);

```

假设“my.jsp”向响应增加了字节4, 5和 6 , 结果是什么?

- A. 123      B. 因为没有调用`os.flush()`, 所以会清除未提交的输出(123), 并调用`forward`而无异常。如果`os.flush()`在`forward`之前调用, 则会抛出一个`IllegalStateException`异常。
- C. 123456
- D. 456123
- E. 会抛出一个异常

**46**

一个程序员需要更新一个正在运行的servlet的初始化参数, 使web应用立即开始使用这些新参数。

为了完成这个任务, 必须完成以下哪些步骤 (不过可能并不充分) ?  
(选出所有正确的答案)

- A. 对于每个参数, 必须修改一个指定servlet名、参数名以及新参数值的DD标记。
- B. servlet的构造函数必须从servlet的**ServletConfig**对象获取更新后的DD参数。
- C. 容器必须撤销然后重新初始化这个servlet。
- D. 对于每个参数, DD必须有一个单独的**<init-param>**标记。

A: `<init-param>`标记必须放在`<servlet>`标记中, 因此`<init-param>`标记没有servlet的名。B: 在构造函数运行之前; 能获取**ServletConfig**对象。C: 必须初始化一个新的Servlet来保存新的**ServletConfig**。**47**哪些类型可以与**HttpServletResponse**方法结合使用来输出数据?

(选出所有正确的答案)

- A. `java.io.PrintStream`      A: `getWriter()`方法返回一个`PrintWriter`
- B. `java.io.PrintWriter`
- C. `java.io.OutputStream`
- D. `java.io.FileOutputStream`
- E. `java.io.ServletOutputStream`      E: `getOutputStream()`方法返回一个`ServletOutputStream`
- F. `java.io.ByteArrayOutputStream`

48

你的Web应用有一个合法的DD，其中只有一个<**security-constraint**> 标记。在这个标记中：

Servlet 12.8,  
本书666页

- 有一个声明为**directory1**的uri模式
- 有一个声明为**POST**的HTTP方法
- 有一个声明为**GUEST**的角色名

如果应用的所有资源都在**directory1** 和 **directory2**中, 而且**MEMBER** 也是一个合法的角色, 以下哪种说法是正确的? (选出所有正确的答案)

- A. **GUEST**不能在**directory1**中完成**GET**请求。
- B. **GUEST**在两个目录中都能完成**GET** 请求。
- C. **GUEST**只能在**directory2**中完成**POST**请求。
- D. **MEMBER**在两个目录中都能完成**GET**请求。
- E. **GUEST**在两个目录中都能完成**POST**请求。
- F. **MEMBER**在**directory1**中只能完成**POST**请求。

这种情况下~~的限制是~~: 只有  
**GUEST**能够在**directory1**中完成  
**POST**请求。

49

给定以下代码:

JSP v2.0 1.10.2节:  
本书314, 502页

```
1. <%@ taglib prefix="c" uri="http://java.sun.com/jsp/
   jstl/core" %>
2. <%@ taglib prefix="tables" uri="http://www.javaranch.
   com/tables" %>
3. <%@ taglib prefix="jsp" tagdir="/WEB-INF/tags" %>
4. <%@ taglib uri="UtilityFunctions" prefix="util" %>
```

以上taglib指令会导致JSP无法工作, 对此下面的哪个说法是正确的?

- A. 第4行不对, 因为prefix必须在uri属性前面。      A: 属性可以是任何顺序。
- B. 第3行不对, 因为不存在uri属性。      B: 使用标记文件时, 会使用  
*tagdir*而不是uri。
- C. 第4行不对, 因为uri值必须以**http://**开头。      C: URL必须与容器标识TLD的方式匹配。
- D. 第3行不对, 因为前缀**jsp** 是为标准动作保留的。      D: *jsp*前缀是为标准动作保留的。

**50**

给定**resp**是一个合法**HttpServletResponse**对象的引用,这个响应中包含以下首部(还可能包含有其他首部):

```
Content-Type: text/html
MyHeader: mydata
```

并有以下调用:

```
25. resp.addHeader("MyHeader", "mydata2");
26. resp.setHeader("MyHeader", "mydata3");
27. resp.addHeader("MyHeader", "mydata");
```

**MyHeader**首部中存在哪些数据?

- A. mydata
- B. mydata3
- C. mydata3,mydata C: setHeader()替换首部中现有的数据;  
addHeader()为现有数据再增加新的数据。
- D. mydata3,mydata2
- E. mydata,mydata2,mydata3
- F. mydata,mydata2,mydata3,mydata

**51**

以下是一个遗留应用的web.xml中的一部分:

```
<jsp-config>
  <taglib>
    <taglib-uri>prettyTables</taglib-uri>
    <taglib-location>/WEB-INF/tlds/prettyTables.tld</taglib-location>
  </taglib>
</jsp-config>
```

假设运行代码的服务器现在支持Java 1.4 EE 或更高版本,怎样才能去除以上**<jsp-config>**标记而使代码仍能正常工作?

- A. 将JSP中taglib指令的uri属性改为"\*",容器就会自动映射到它。
- B. 在TLD文件中放入<uri>prettyTables</uri>。
- C. 删除JSP中使用这个映射的**taglib** 指令。容器会自动进行处理。
- D. 这是不可能的。要让容器将TLD映射到JSP中引用的uri,就必须有这里的**<jsp-config>**。

D: 这并非不可能,见选项B!

B: 正确。可以看到,TLD放在WEB-INF下,所以容器会找到它。如果TLD包含一个<uri>,容器就会隐式地将这个值映射到适当的TLD位置。

C: 如果从JSP删除taglib指令,对应prettyTables的标记就会作为模板文本。

52

对于一个列出购物车商品的页面，当购物车为空时必须显示消息“Your shopping cart is empty.”（你的购物车为空）。以下哪个代码段可以实现这个功能，在此假设作用域属性cart是一个商品List? (选出所有正确的答案)

JSTL v1.1  
5.3-5.6和6.2节  
本书447 454页

- A. <c:if test='\${empty cart}'>  
    Your shopping cart is empty.  
</c:if>  
<c:forEach var="itemInCart" items="\${cart}">  
    <shop:displayItem item ="\${itemInCart}" />  
</c:forEach>
- A. C和D都是合法的。A是最简单的首选方案。
- B. <c:forEach var="itemInCart" items="\${cart}">  
    <c:choose>  
        <c:when test='\${empty itemInCart}'>  
            Your shopping cart is empty.  
        </c:when>  
        <c:otherwise>  
            <shop:displayItem item ="\${itemInCart}" />  
        </c:otherwise>  
    </c:choose>  
</c:forEach>
- B. 如果cart为null或null,  
c:forEach不会执行其体。  
购物车为空时不会看到这个消息。
- C. <c:choose>  
    <c:when test='\${empty cart}'>  
        Your shopping cart is empty.  
    </c:when>  
    <c:when test='\${not empty cart}'>  
        <c:forEach var="itemInCart" items="\${cart}">  
            <shop:displayItem item ="\${itemInCart}" />  
        </c:forEach>  
    </c:when>  
</c:choose>
- D. <c:choose>  
    <c:when test='\${empty cart}'>  
        Your shopping cart is empty.  
    </c:when>  
    <c:otherwise>  
        <c:forEach var="itemInCart" items="\${cart}">  
            <shop:displayItem item ="\${itemInCart}" />  
        </c:forEach>  
    </c:otherwise>  
</c:choose>

**53**

给定一个servlet中的以下代码,而且myVar 是 HttpSession或ServletContext 的一个引用:

Servlet 2, 本书  
190~199页

```
15. myVar.setAttribute("myName", "myVal");
16. String s = (String) myVar.getAttribute("myName");
17. // more code
```

执行第16行之后,以下哪种说法是正确的? (选出所有正确的答案)

- A. s的值不能保证。
- B. 如果 myVar是一个HttpSession, 编译会失败。
- C. 如果 myVar是一个ServletContext, 编译会失败。
- D. 如果 myVar是一个HttpSession, s 的值肯定是"myVal"。
- E. 如果 myVar是一个ServletContext, s 的值肯定是"myVal"。

**54**

以下是Java EE web应用部署描述文件中的一部分:

Sev: 附录B,  
本书627页

```
62. <error-page>
63.   <exception-type>IOException</exception-type>
64.   <location>/mainError.jsp</location>
65. </error-page>
66. <error-page>
67.   <error-code>404</error-code>
68.   <location>/notFound.jsp</location>
69. </error-page>
```

以下哪些说法是正确的?

A: 在DD中指定一个异常类型时,必须使用  
完全限定类名 (如java.io.IOException)。

- A. 这个部署描述文件不合法。
- B. 如果应用抛出一个IOException异常, 不会提供任何页面。
- C. 如果应用抛出一个IOException异常, 会提供notFound.jsp。
- D. 如果应用抛出一个IOException异常, 会提供mainError.jsp。

55

给定以下JSP:

```

1. <%! String GREETING = "Welcome to my page"; %>
2. <% request.setAttribute("greeting", GREETING); %>
3. Greeting: ${greeting}
4. Again: <%= request.getAttribute("greeting") %>

```

JSP v2.0 6.2.2 和  
6.3.2 节  
本书629页

如果试图将以上JSP转换为一个JSP文档:

```

01. <jsp:declaration>
02.   String TITLE = "Welcome to my page";
03. </jsp:declaration>
04. <jsp:scriptlet>
05.   request.setAttribute("greeting", GREETING);
06. </jsp:scriptlet>
07. Greeting: ${greeting}
08. Again: <jsp:expression>
09.   request.getAttribute("greeting");
10.</jsp:expression>

```

这个新JSP文档哪里有问题? (选出所有正确的答案)

- A. 没有声明<jsp:root>。 A: <jsp:root>不是必要的标记。
- B. 模板文本应当用 <jsp:text> 标记包围起来。 B: 否则, 这就不是合法的XML!
- C. EL表达式在JSP文档中是不允许的。
- D. <jsp:expression> 的内容不能有分号。 D: 哇呀! 裁错了!

56

以下哪一个组件最不可能建立或接收网络调用?

核心j2ee 302页。  
本书761页

- A. JNDI 服务器 A: 如果看到一个模式或组件不是大纲所要求的, 就可以把它排除, 它绝对不会是正确答案!
- B. 传输对象 B: 传输对象通常在网络调用中发送, 但是它们很少发起网络调用或者对网络调用做出响应。
- C. 服务定位器
- D. 前端控制器
- E. 拦截过滤器

57

给定以下代码：

JSTL v1.1 4.2节

```

10. ${questionNumber}: ${question}
11. <c:forEach var="answer" items="${answers}">
...
16. </c:forEach>

```

question属性是一个String，其中可能包含必须在浏览器中显示为常规文本的XML标记。基于以上代码段，浏览器不会显示XML标记。应当如何修改来修正这个问题？（选出所有正确的答案）

- A. \${question} 替换为 <c:out value="\${question}"/>
  - B. \${question} 替换为 <c:out>\${question}</c:out>
  - C. \${question} 替换为 <c:out escapeXml="true" value="\${question}"/>
  - D. \${question} 替换为 <%= \${question} %>
- D: 抱歉，不过这个选项差得太远了。  
不能把EL放在scriptlet中。

A和C: escapeXml默认为true，所以A和C都是正确的。<c:out>的escapeXml可以将XML字符(<, >, &, ', ")转换为特殊代码，从而使浏览器能够适当地显示而不会错误地作为html。

B: value属性对于<c:out>是必要的。即使<c:out>可以有一个值，但体会替换default属性而不是value属性。

58

你的Java EE web应用用户越来越多，你决定再增加一个服务器来支持更多的客户请求。对于会话从一个服务器迁移到另一个服务器，以下哪些说法是正确的？（选出所有正确的答案）

Servlet 7.  
本书257~264页

- A. 在会话中这些迁移是不可能的。
- B. 迁移会话时，其**HttpSession**会随之迁移。
- C. 迁移会话时，其**ServletContext**会随之迁移。
- D. 迁移会话时，其**HttpServletRequest**会随之迁移。
- E. 如果使用**HttpSession.setAttribute**增加对象，这个对象必须是**Serializable**才能从一个服务器迁移到另一个服务器。
- F. 如果使用**HttpSession.setAttribute**增加对象，而且这个对象的类实现了**Serializable.readObject**和**Serializable.writeObject**，会话迁移时，容器会调用这些**readObject**和**writeObject**方法。
- G. 如果会话属性实现了**HttpSessionActivationListener**，容器唯一的需求就是一旦在新服务器上激活会话就要通知监听者。

E: 除非对象是**Serializable**，否则没有办法迁移对象。

F: 不能保证一定有这些调用！

G: 容器还必须发送一个纯化通知。

59

一个Java EE部署描述文件声明了多个过滤器，它们的URL都与一个给定请求匹配，另外外部部署描述文件中还声明了多个过滤器的 <filter-name> 标记与同一个请求匹配。

Servlet 6,  
本书710页

针对这个请求，容器将使用哪些规则调用过滤器，对此以下哪些说法是正确的？

(选出所有正确的答案)

- A. 只有<filter-name> 匹配的过滤器会得到调用。
- B. 在URL匹配的过滤器中，只会调用第一个。
- C. 在<filter-name> 匹配的过滤器中，只会调用第一个。
- D. <filter-name> 匹配的过滤器会在URL匹配的过滤器之前调用。
- E. 所有URL匹配的过滤器都会得到调用，不过调用顺序不确定。
- F. 所有URL匹配的过滤器都会得到调用，并按照其出现在DD中的顺序调用。

首先，容器会按照DD声明顺序调用所有URL匹配的过滤器，然后会调用<filter-name>匹配的过滤器，仍以DD声明顺序调用。

60

比较servlet初始化参数和上下文初始化参数时，关于二者的共同点以下哪些说法是正确的？ (选出所有正确的答案) serv 9, 13本书  
157~160页

- A. 在其各自的DD标记中，都有一个<param-name>和一个<param-value>标记。
- B. 它们各自的DD标记都直接放在<web-app> 标记之下。 <web-app>标记下。
- C. 它们用于获取初始化参数值的方法都是getInitParameter。
- D. 它们都可以从JSP直接访问。 & 只有上下文参数可以从JSP直接访问。
- E. 只有DD中上下文初始化参数的改变能够访问而无需重新部署web应用。 E: DD中对二者的改变都不能动态访问。

61

一个JSP开发人员希望将文件copyright.jsp的内容包含到所有主JSP页面中。

以下哪种机制可以做到这一点？(选出所有正确的答案)

- A. <jsp:directive.include file="copyright.jsp" /> A: 是对的，因为这种语法对于JSP文档是正确的。
- B. <%@ include file="copyright.jsp" %> B: 正确，因为这种语法对于JSP页面是正确的。
- C. <%@ page include="copyright.jsp" %> C: 不对，因为不能使用page指令来导入内容。
- D. <jsp:include page="copyright.jsp" /> D: 是对的，因为这个标准动作会在运行时完成内容包含。
- E. <jsp:insert file="copyright.jsp" /> E: 不对，因为根本不存在这个标准动作。

JSP Version 2.0  
section 1.10.5

62

你在为一个提供电话、电缆和Internet服务的公司开发应用，用来管理客户账户。很多页面都包含一个搜索能。每个页面上的搜索框看上去要一样，不过有些页面限制为只能搜索电话、电缆或Internet账户。

另外有一个名为Search.jsp的JSP：

```
1. <form action="/search.go">
2.   Find ${param.accountType} Account:
3.   <input type="text" name="searchText"/>
4.   <input type="hidden" name="accountType" value="${param.accountType}" />
5.   <input type="submit" value="Search"
6. </form>
```

JSP v2.0 5.4, 5.6节  
本书400~408页

在需要搜索电缆账户的JSP中应当使用什么标记？

- A. <jsp:include page="Search.jsp" accountType="Cable"/> A: <jsp:include>不能有一个名为accountType的属性。
- B. <jsp:include page="Search.jsp">  
 <jsp:param name="accountType" value="Cable"/> B: \${param.accountType}会找到用<jsp:param>传递的Cable参数。
- C. <jsp:include file="Search.jsp" accountType="Cable"/> C和D: <jsp:include> 使用page属性。file属性用在include指令中。
- D. <jsp:include file="Search.jsp">  
 <jsp:attribute name="accountType" value="Cable"/>  
</jsp:include>

63

测试不同标记和scriptlet如何工作时，一个开发人员创建了以下JSP：

JSP v2.0 1.3.1(和1.5节)  
本书304, 483页

```
1. <% request.setAttribute("name", "World"); %>
2. <!-- Test -->
3. <c:out value='Hello, ${name}' />
```

让这个开发人员惊讶的是，获取他的JSP时，浏览器什么也没有显示。如果开发人员查看页面的HTML源代码，会在输出中发现什么？

- A. <!-- Test -->
- B. <!-- Test -->  
<c:out value='Hello, \${name}' />
- C. <!-- Test -->  
<c:out value='Hello, World' /> C: \${name} EL确实会计算，但是JSP无法识别<c:out>标记，只是把它作为模板文本，因为JSP中没有声明taglib。
- D. 没有任何输出

64

有一个速配服务应用，会询问其用户一系列问题。已经存在一个会话作用域属性，名为 **compatibilityProfile**，类型为 **HashMap**，所提交的各组问题ID和相应的答案就存储在这个属性中。

JSTL v1.1 4.3节  
本书455~457页

给定以下代码：

```
22. <% ((java.util.HashMap)request.getSession().getAttribute("compatibilityProfile")).put(
23.         request.getParameter("questionIdSubmitted"),
24.         request.getParameter("answerSubmitted"));
25. %>
```

如何替换这些代码而不使用scriptlet？（选出所有正确的答案）

- A. `<c:map target="${compatibilityProfile}" key ="${param.questionIdSubmitted}" value ="${param.answerSubmitted}" />`

A: `<c:map>`不是真正的标记。
- B. `<jsp:useBean id="compatibilityProfile" class="java.util.HashMap" scope="session">
<jsp:setProperty name="compatibilityProfile" property ="${param.questionIdSubmitted}" value ="${param.answerSubmitted}" />
</jsp:useBean>`

B: `<jsp:useBean>`只用于bean，而不能应用于map！
- C.  `${compatibilityProfile[param.questionIdSubmitted]} = ${param.answerSubmitted}`

C: EL本身无法为一个对象设置值。
- D. `<c:set target="${compatibilityProfile}" property ="${param.questionIdSubmitted}" value ="${param.answerSubmitted}" />`

D: `<c:set>`可以用于将值置于一个map。

65

一个程序员正在为一个Java EE web应用创建过滤器。给定以下代码：

API 本节  
707页

```

7. public class MyFilter implements Filter {
8.     public void init(FilterConfig config) throws FilterException { }
9.
10.    public void doFilter(HttpServletRequest request,
11.                          HttpServletResponse response,
12.                          FilterChain chain)
13.        throws IOException, ServletException { }
14.
15. }

```

必须做哪些修改才能创建一个合法的过滤器？（选出所有正确的答案）

A. 不需要做任何修改。

B. 必须增加一个 `destroy()` 方法。

C: 除了其他修改，`doFilter()` 必须调用 `chain.doFilter()`。

C. 必须修改 `doFilter()` 方法的体。

D. 必须修改 `init()` 方法的签名。

D: `init()` 抛出一个 `ServletException`。

E. 必须修改 `doFilter()` 方法的参数。

E: `doFilter()` 改 `ServletRequest` 和 `ServletResponse` 参数。

F. 必须修改 `doFilter()` 方法的异常。

66

你的公司希望增加一个醒目页面 `SplashAd.jsp`，使用户第一次进入网站时能够看到公司其他产品的广告。在这个新页面上，将为用户提供一个选项，允许用户选中广告页面上的一个显示“Do not show me this offer again”的复选框，并点击一个显示“Continue to My Account”的提交按钮。如果用户提交表单时这个复选框选中，接收Servlet会为用户的浏览器设置一个Cookie，其名为“skipSplashAd”，然后将控制传回主JSP。

JSP v2.0 5.1  
本书 409~411

主JSP负责将请求转发到这个醒目页面。可以在主页面最前面增加以下哪个代码段，从而当用户未选择这个复选框来避免显示广告时能够为用户显示这个醒目页面？

A. <c:if test="\${empty cookie.skipSplashAd and pageContext.session.new}">

<jsp:forward page="SplashAd.jsp"/>  
</c:if>

A: 正确。仅当未设置这个Cookie时才会发生转发。要注意，如果用户禁用了cookie，采用这个解决方案时这些用户将无法跳过广告。

B. <jsp:forward page="SplashAd.jsp" flush="\${empty cookie.skipSplashAd}" />

B: flush 属性在这里没有任何帮助。

C. <jsp:redirect page="SplashAd.jsp"/>

C 和 D: 不存在 <jsp:redirect> 标记。

D. <jsp:redirect file="SplashAd.jsp"/>

E: 这里的 scriptlet 是不合法的。cookie 是 EL 中的一个隐式对象，而不是 scriptlet 中的隐式对象。

E. <% if(cookie.get("skipSplashAd") == null && session.isNew()) { %>

<jsp:forward page="SplashAd.jsp"/>  
<% } %>

67

一个程序员想要实现一个**ServletContextListener**。给定以下DD代码片段：

```

101.    <!-- insert tag1 here -->
102.    <param-name>myParam</param-name>
103.    <param-value>myValue</param-value>
104.    <!-- close tag1 here -->
105.    <listener>
106.        <!-- insert tag2 here -->
107.        com.wickedlysmart.MySCListener
108.        <!-- close tag2 here -->
109.    </listener>
```

API, Servlet  
本书171~174页

并给定以下监听者类伪代码：

```

5. // packages and imports here
6. public class MySCListener implements ServletContextListener {
7.     // method 1 here
8.     // shutdown related method here
9. }
```

以下哪些说法是正确的？（选出所有正确的答案）

- A. 这个DD 代码段不可能合法。
- B. tag1应当是<context-param>
- C. tag1应当是<servlet-param>
- D. tag2应当是<listener-class>
- E. tag2应当是<servlet-context-class>
- F. method1应当是initializeListener
- G. method1应当是contextInitialized

有些东西必须死记。

68

wickedlysmart 网站有一个正确部署的Java EE web应用和一个部署描述文件，  
其中包含以下代码：

第9章  
625页

```
<welcome-file-list>
    <welcome-file>welcome.html</welcome-file>
    <welcome-file>howdy.html</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

以下是这个Web应用目录结构的一部分：

```
MyWebApp
|
|-- index.html
|
|-- welcome
|   |-- welcome.html
|
|-- foobar
|   |-- howdy.html
```

如果应用接收到以下两个请求：

```
http://www.wickedlysmart.com/MyWebApp/foobar
http://www.wickedlysmart.com/MyWebApp
```

会提供哪一组响应？

- A. `howdy.html`然后是404。
- B. `index.html`然后是404。
- C. `welcome.html`然后是404。
- D. `howdy.html`然后是`index.html`。
- E. `index.html`然后是`index.html`。
- F. `howdy.html`然后是`welcome.html`。
- G. `welcome.html`然后是`index.html`。

D: 如果DD未包含一个servlet映射，它会搜索请求中指定的目录，并提供*welcome*列表中找到的与所请求目录中某个文件匹配的第一个文件。

69

你的Web应用有一个合法的DD，其中只包含一个<**security-constraint**>标记。这个标记中有：

Servlet 12.8.  
本书664~665页

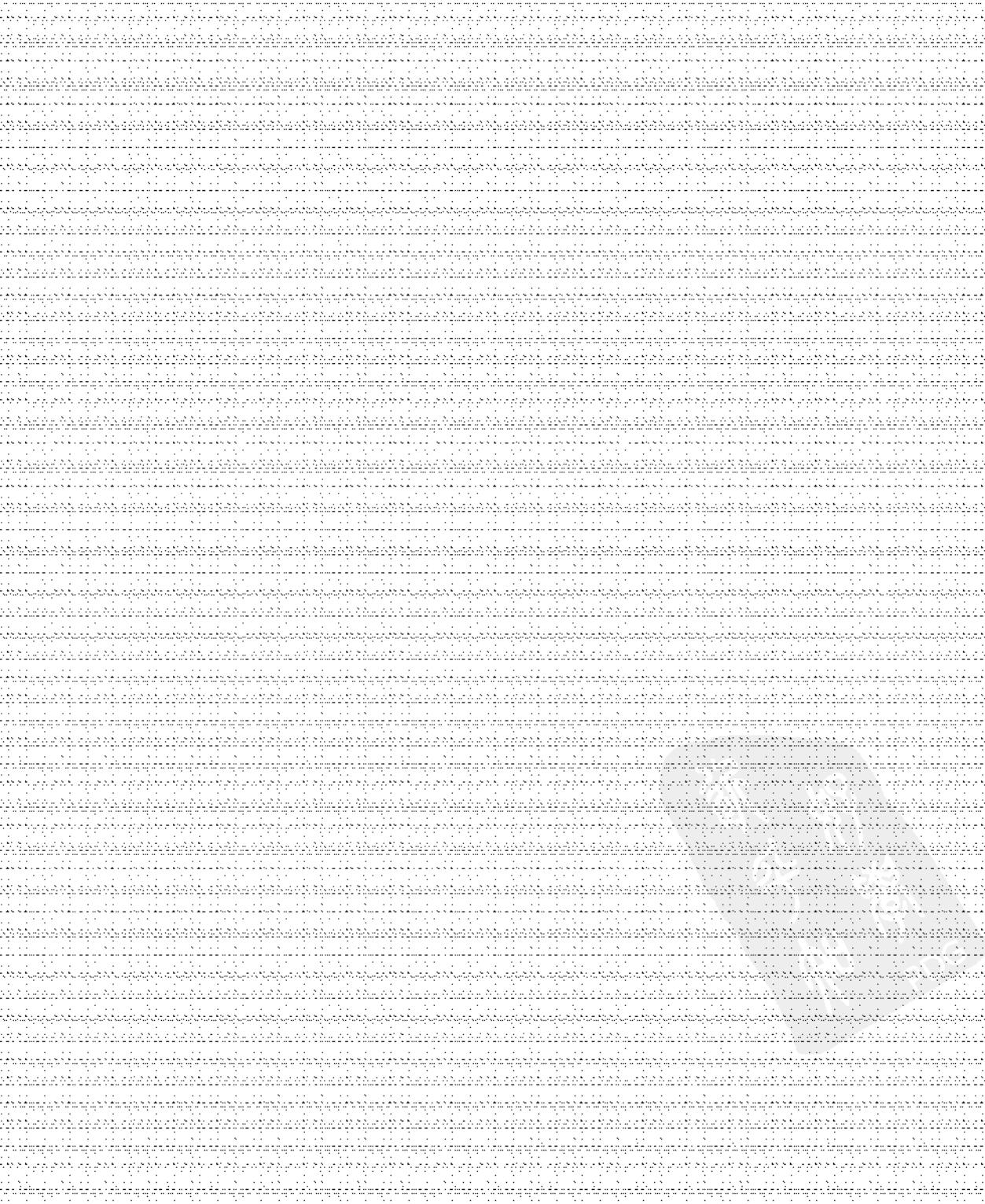
- 一个声明为**GET**的HTTP方法

应用中的所有资源都在**directory1**和**directory2**中，另外只定义了**BEGINNER**和**EXPERT**角色。

如果想要限制**BEGINNER**不能使用**directory2**中的资源，应当如何声明url和role标记，对此以下哪些说法是正确的？（选出所有正确的答案）

- A. 应当有一个url标记声明**directory1**，而且有一个role标记声明**EXPERT**。
- B. 应当有一个url标记声明**directory2**，而且有一个role标记声明**EXPERT**。
- C. 应当有一个url标记声明**directory1**，而且有一个role标记声明**BEGINNER**。
- D. 应当有一个url标记声明**directory2**，而且有一个role标记声明**BEGINNER**。
- E. 应当有一个url标记声明**ANY**，而且其role标记声明**EXPERT**，另一个url标记声明**directory2**，其role标记声明**BEGINNER**。
- F. 应当有一个url标记声明这两个目录，其role标记声明**EXPERT**，另一个url标记声明**directory1**，其role标记声明**BEGINNER**。

记住DD中总是  
声明约束。



# 索引

## 符号与标记

!(EL operator) (! (EL 操作符)) 396  
 "character (HTML) (" 字符(HTML)) 444  
 && 396  
 & character (HTML) (& 字符 (HTML)) 444  
 'character (HTML) (' 字符 (HTML)) 444  
 <%! %> 294  
 <%= %> 291  
 <%@ page import %> 287  
 <attribute> 477  
 <auth-constraint> 665, 668  
 <auth-method> 663, 678  
 <body-content> 477, 482. *See* body-content  
 <c:catch> 472  
 <c:choose> 454  
 <c:forEach> 447  
 <c:if> 451  
 <c:import> 460  
 <c:otherwise> 454  
 <c:out>  
     default attribute (default属性) 445  
     escapeXml attribute (escapeXml属性) 443  
 <c:param> 463  
 <c:remove> 458  
 <c:set> 455  
     gotchas (技巧) 457  
 <c:url> 465  
 <c:when> 454  
 <context-param> 157  
 < character (HTML) 444  
 <dispatcher> 711  
 <dynamic-attributes> element (<dynamic-attributes> 元素)  
     559, 561  
 <ejb-local-ref> 630

<ejb-ref-name> 630  
 <ejb-ref-type> 630  
 <ejb-ref> 630  
 <el-ignored> 322  
 <env-entry-name> 632  
 <env-entry-type> 632  
 <env-entry-value> 632  
 <env-entry> 632  
 <error-code> 470  
 <error-page> 470  
 <exception-type> 470  
 <extension> 633  
 <filter-class> 710  
 <filter-mapping> 710  
 <filter-name> 710  
 <filter> 710  
 <form-error-page> 679  
 <form-login-config> 679  
 <form-login-page> 679  
 <function> 393  
 <home> 630  
 <http-method> 665  
 <init-param> 150  
 <jsp-config> 321  
 <jsp-file> 310  
 <jsp-property-group> 321  
 <jsp:attribute> 481  
 <jsp:forward> 413  
 <jsp:getProperty> 349  
 <jsp:include> 404  
 <jsp:param> 412  
 <jsp:setProperty> 351  
 <jsp:useBean> 349, 354  
 <listener-class> 169, 261

<listener> 169, 261  
 <load-on-startup> 628  
 <local-home> 630  
 <local> 630  
 <location> 470  
 <login-config> 663, 678, 679  
 <max-inactive-interval> 248  
 <mime-mapping> 633  
 <mime-type> 633  
 <param-name>  
     for context init parameters (上下文初始化参数) 157  
     for servlet init parameters (servlet初始化参数) 150  
 <param-value>  
     for context init parameters (上下文初始化参数) 157  
     for servlet init parameters (servlet初始化参数) 150  
 <remote> 630  
 <required> 477  
 <role-name> 664, 668  
 <rtpexprvalue> 477, 480  
 <scripting-invalid> 321  
 <security-constraint> 665, 670  
 <security-role> 664  
 <select>  
     automating (自动化) 542–549  
     core attributes (核心属性) 550  
     event attributes (事件属性) 550  
     form attributes (表单属性) 550  
     internationalization attributes (国际化属性) 550  
     tag attribute setters (标记属性设置方法) 552–554  
 <servlet-class> 76  
 <servlet-mapping> 619  
 <servlet-name> 76  
 <servlet> 76  
 <session-config> 248  
 <session-timeout> 245  
 <tag-class> 477  
 <tag> 477  
 <timeout> 248  
 <tomcat-users> 663  
 <transport-guarantee> 684  
 <uri> 393, 477, 484  
 <url-pattern> 76  
     security constraints (安全约束) 665  
     the real details (具体细节) 618  
 <user-data-constraint> 684  
 <web-resource-collection> 665

<web-resource-name> 665  
 <welcome-file> 622  
 <welcome-file-list> 622  
 > character (HTML) (> 字符 (HTML)) 444  
 [] operator ([] 操作符) 371  
 || (EL operator) (|| (EL 操作符)) 396

## A

Action object (Action 对象) 773  
 ActionServlet 773  
 addCookie() 251  
 addHeader() 133  
 ancestor (classic tags) (祖先 (传统标记)) 574  
 Apache, directory structure (Apache, 目录结构) 22  
 API, servlet 98  
 application/context scope (应用/上下文作用域) 186  
 APPLICATION\_SCOPE 312  
 attribute, in a simple tag handler (属性, 简单标记处理器中) 521  
 attributeAdded 182  
 attribute directive (attribute指令) 506  
 attributeRemoved 182  
 attributeReplaced 182  
 attributes (属性)  
     listeners 参见 listeners  
     scope (作用域) 187  
         in a JSP (JSP中) 311  
     thread safety issues (线程安全问题) 192  
     what are they? (它们是什么?) 185  
 auth-constraint 665  
 authentication (认证) 653, 657, 677  
     BASIC (基本) 677  
     CLIENT-CERT (客户证书) 677  
     DIGEST (摘要) 677  
     FORM (表单) 677  
 authorization (授权) 653, 657

## B

BASIC (基本) 677  
 bean-related standard actions (与bean相关的标准动作) 348  
     bullet points (要点) 418  
 BE the Container (作为容器)  
     deployment (部署) 607  
     sessions (会话) 247  
     standard actions (标准动作) 358

- tag handlers (标记处理器) 537  
TLD/JSP 517
- body-content  
empty (空) 482  
in Tag Files (标记文件中) 508  
JSP 482  
scriptless 482  
tagdependent 482  
tag element (标记元素) 482
- BodyTag 530, 562  
BodyTagSupport 530
- Bullet Points (要点)  
<dynamic-attributes> element (<dynamic-attributes>元素) 561  
<web-resource-collection> 666  
bean-related standard actions (与bean相关的标准动作) 418  
chapter 1-intro (第1章) 35  
chapter 4-servlet lifecycle and API (第4章servlet生命周期和API) 124  
DynamicAttributes interface (DynamicAttributes接口) 561  
EL review (EL复习) 400  
HTTP and HttpServletRequest (HTTP 和 HttpServletRequest) 125  
HttpServletResponse 140  
review of include (包含复习) 418  
setDynamicAttribute() method (setDynamicAttribute()方法) 561  
simple tags (简单标记) 523
- Business Delegate (业务委托) 753  
Business tier patterns (业务层模式) 761
- C**
- CGI 27  
chain (filter) (链 (过滤器)) 714  
chain.doFilter() 714  
class attribute (<select>) (class属性 (<select>)) 550  
Classic tag handlers (传统标记处理器)  
ancestors (祖先) 574  
API 530  
BodyTag 562  
default return values (默认返回值) 537  
IterationTag 537  
lifecycle (生命周期) 533  
lifecycle return values table (生命周期返回值表) 563  
Parent/getParent() 568
- Classic tags, DynamicAttributes interface (传统标记, DynamicAttributes接口) 560
- class variables, thread-safe (类变量, 线程安全) 203  
CLIENT-CERT (客户证书) 677  
Code Magnets (代码贴)  
configuring DD init params (配置DD初始化参数) 161  
EL 380  
JSP elements (JSP元素) 325  
making a JSP (建立JSP) 300  
servlet/DD intro (servlet/DD简介) 60  
session/timeout (会话/超时) 246
- Coffee Cram. 参见 Mock Exam Questions  
comments (in a JSP) (注释 (JSP中)) 304  
compiling, example (编译, 示例) 81  
CompressionFilter 722  
compression filter (压缩过滤器) 711  
CompressionResponseWrapper 724  
CONFIDENTIAL 684  
confidentiality (机密性) 653, 684  
constraints (security) (约束 (安全)) 665  
Container, what it does (intro) (容器, 做什么 (引子)) 39-41  
Container-generated servlet (容器生成的servlet) 293  
contentType 315  
content type (内容类型) 130  
Context (上下文)  
attributes (属性) 187  
not thread-safe (非线程安全) 192  
listener 166. 参见 ServletContextListener  
scope (作用域) 187  
not thread-safe (非线程安全) 192
- context (init) parameters (上下文 (初始化) 参数) 157  
in a JSP (JSP中) 390  
vs. servlet init parameters (与servlet初始化参数) 158  
contextDestroyed 182  
contextDestroyed(ServletContextEvent) 166  
contextInitialized 182  
contextInitialized(ServletContextEvent) 166  
Controller (控制器)  
a first look (初览) 54  
first code example (第一个代码示例) 80  
Cookie (class API) 251  
cookie (EL implicit object) (cookie (EL隐式对象)) 390  
cookies 232, 250  
custom cookie example (定制cookie示例) 252-253  
vs. headers (与首部) 253  
cross-site hacking (跨网站攻击) 444  
custom tags (定制标记)  
attributes (属性) 551

development (开发) 543–549  
 invalid attribute name (不合法的属性名) 560

**D**

Dating Query Language (Dating查询语言) 50  
 DD (部署描述文件) 150  
   a first look (初览) 48  
   <env-entry> 632  
   <mime-mapping> configuration (<mime-mapping>配置) 633  
   <scripting-invalid> 321  
   authentication configuration (认证配置) 678  
   context init parameters (上下文初始化参数) 157  
   disabling scripting (禁用脚本) 321  
   EJB-related tags (与ELB相关的标记) 630  
   error page configuration (错误页面配置) 470, 626  
   filter configuration (过滤器配置) 710  
   ignoring EL (忽略EL) 321  
   listener configuration (监听者配置) 174  
   security configuration (安全配置) 670–674  
   servlet init parameters (servlet初始化参数) 150  
   servlet mapping (servlet映射) 616–619  
   session-timeout configuration (session-timeout配置) 245  
   welcome files (欢迎文件) 622  
 declarations, JSP (声明, JSP) 295  
 declarative security (声明式安全) 657  
 Decorator pattern (装饰器模式) 719  
 Deployment Descriptor. 参见 DD  
 deployment environment (部署环境) 73, 613  
   directory structures (目录结构) 607  
   META-INF 613  
   WAR files (WAR文件) 612  
 design patterns (设计模式)  
   Business Delegate (业务委托) 778  
   Front Controller (前端控制器) 783  
   Intercepting Filter (拦截过滤器) 781  
   MVC (a first look) (MVC (初览)) 53  
   MVC (more serious) (MVC (更正式)) 782  
   review (复习) 778  
   Service Locator (服务定位器) 779  
   Transfer Object (传输对象) 780  
 destroy() (Filter interface) (destroy() (Filter接口)) 708  
 development environment (开发环境) 72  
 DIGEST (摘要) 677  
 dir attribute (<select>) (dir 属性 (<select>)) 550  
 directive (指令) 287  
   include 314, 403  
   tag directive for Tag Files (用于标记文件的tag指令) 508  
   Tag File attribute directive (标记文件attribute指令) 506

taglib 314  
   prefix attribute (prefix属性) 393  
 disabled attribute (<select>) (disabled 属性 (<select>)) 550  
 disabling scripting (禁用脚本) 321  
 dispatch 134, 138, 206. 参见 RequestDispatcher  
 div (EL operator) (div(EL操作符)) 396  
 doEndTag() 532  
 doFilter() 708  
 doGet(), servlet method (first look) (doGet(), servlet方法 (初探)) 99  
 doPost(), servlet method (first look) (doPost(), servlet方法 (初探)) 99  
 doStartTag() 531  
 doTag() 513, 558  
 dot operator in EL (EL中的点操作符) 370  
 dynamic (动态) 24  
 dynamic attributes (动态属性)  
   runtime expressions (运行时表达式) 560  
   Tag Files (标记文件) 561  
 DynamicAttributes interface (DynamicAttributes接口) 556–561  
   Classic tags (传统标记) 560  
   doTag() method (doTag()方法) 558  
   setDynamicAttribute() method (setDynamicAttribute()方法) 557, 560, 561

**E**

EJB, related DD tags (EJB, 相关的DD标记) 630  
 EL 369–374  
   bullet points (要点) 400  
   functions (函数) 392–394  
   HTML 384, 442–445  
   implicit objects (隐式对象) 369, 385  
     cookie 390  
     initParam 390  
     param and paramValues (param和paramValues) 386  
     scope (作用域) 389  
   naming rules (命名规则) 370  
   null expression (null表达式) 444–445  
   null values (null值) 399  
   operators (操作符) 396  
   raw text rendering (原始文本显示) 384  
   security risks (安全风险) 444  
   the [] operator ([]操作符) 371  
 empty (空)  
   <body-content> 482  
   tag (标记) 482  
 encodeRedirectURL() 239

encodeURL() 238  
 eq (EL operator) (eq(EL操作符)) 396

error pages (错误页面) 468, 626

errorPage 315, 468

EVAL\_BODY AGAIN 539

EVAL\_BODY\_BUFFERED 563

EVAL\_BODY\_INCLUDE 533, 539

EVAL\_PAGE 532

exam. 参见 Objectives (official exam)

exception, implicit object (异常, 隐式对象) 471

Exercises (练习)

BE the Container (作为容器)

deployment (部署) 607

sessions (会话) 247

standard actions (标准动作) 358

tag handlers (标记处理器) 537

TLD/JSP 517

Code Magnets (代码贴)

configuring DD init params (配置DD初始化参数)  
161

EL 380

JSP elements (JSP元素) 325

Making a JSP (建立JSP) 300

servlet/DD intro (servlet/DD简介) 60

sessions (会话) 246

Deployment exercise (部署练习) 634

EL and scripting evaluation (EL和脚本计算) 324

Request/Response intro (请求/响应简介) 29

Who's responsible? (谁负责?) 59

expression (表达式)

JSP 288–289

## F

Filter (过滤器)

interface methods (接口方法) 708

lifecycle (生命周期) 708

filter (过滤器)

BeerRequestFilter example (BeerRequestFilter示例) 707

mapping (映射) 710

using with dispatcher (用于分派器) 711

FilterChain 708

findAncestorWithClass() 574

findAttribute() (pageContext) 313

Five Minute Mystery, Case of the Missing Content (5分钟揭秘, 内容丢失案件) 383

form, parameters 120–121. 参见 parameters

FORM-based security (基于表单的安全) 677, 679

j\_password 679

j\_security\_check 679

j\_username 679

form bean (表单bean) 772

forward() 206

forward (standard action) (forward (标准动作)) 414

Front Controller (前端控制器) 769

functions (in EL) (函数 (EL中)) 392–394

## G

ge (EL operator) (ge(EL操作符)) 396

GeekDates 50

GenericServlet, API 98

GET 12–15

and idempotency (幂等性) 116

vs. POST (differences) (与POST (差别)) 110

getAttribute() 186

using pageContext (使用pageContext) 313

getCookies() 251

getCreationTime() 243

getHeader() 123

getInitParameter() 150, 163

getIntHeader() 123

getJspBody() 514

getLastAccessedTime() 243

getLocalPort() 123

getMaxInactiveInterval() 243

getOutputStream() 132

getParameter() 121

getParameterValues() 121

getParent() 568

getRemotePort() 123

getRemoteUser() 674

getRequestDispatcher() 206

getServerPort() 123

getServletConfig() 150, 163

getServletContext() 157, 163

getSession() 233

getWriter() 132

gt (EL operator) (gt(EL操作符)) 396

## H

header (EL implicit object) (首部 (EL隐式对象)) 387

headers (首部) 123

adding/setting (增加/设置) 133

vs. cookies 253  
**HTML 6–8**  
 formatting (格式化) 442–445  
 Java helper method (Java辅助方法) 442  
 rendering (显示) 384  
 special characters (特殊字符) 384, 442–444  
**HTTP 6, 10**  
 GET 12–15. 参见 GET  
 Methods (方法) 108–110  
 difference between GET and POST (GET和POST之间的差别) 110  
**POST 16. 参见 POST**  
 request (introduction) (请求 (介绍)) 12–13  
 response (introduction) (响应 (介绍)) 10–11  
**http-method 665**  
**HttpServletRequest, API 98**  
**HttpServletRequest 106, 122, 189**  
 bullet points (要点) 125  
**HttpServletResponse 106, 126**  
 bullet points (要点) 140  
**HttpServletResponseWrapper 720**  
**HttpSession 227**  
 API 243  
**HttpSessionActivationListener 182, 260, 263**  
**HttpSessionAttributeListener 182, 262**  
**HttpSessionBindingEvent 182**  
**HttpSessionBindingListener 182, 256, 263**  
**HttpSessionEvent 182**  
**HttpSessionListener 182, 261**

'<select> (id属性 (<select>)) 550  
 (幂等的) 112–114  
 · (隐式对象)  
 · 参见 EL, implicit objects (隐式对象)

**M**  
**rev**  
**Serv**  
**Trans**  
**destroy()**  
**developm**  
**DIGEST**  
**dir attribu**  
**directive**  
 inclu  
 tag d  
 Tag I ...

'<c:import>标记) 460  
 'bute (page指令属性) 287  
 指令) 314, 403  
 Fi'<import>标记) 460  
 serv  
**870 索**  
 708  
 vlet方法(初探)) 99

init parameters (初始化参数) 150–151, 158  
**Initialization (初始化)**  
 context init parameters (上下文初始化参数) 157  
 using EL (使用EL) 390  
**JSP 310**  
 context init parameters (using EL) (上下文初始化参数 (使用EL)) 390  
 jspInit() 310  
 servlet init params (servlet初始化参数) 310  
 servlet 103  
 servlet init parameters (servlet初始化参数) 150–151  
 web app/servlet context (web应用/servlet上下文) 159  
**instance variables (实例变量)**  
 SingleThreadModel 201  
**INTEGRAL 684**  
**integrity (完整性) 653, 684**  
**invalidate() 243, 245**  
**invoke() (JSP body) 514**  
**isELIgnored 315**  
**isErrorPage 315, 468**  
**isNew() 234**  
**isThreadSafe 315**  
**isUserInRole() 674**  
**items (<c:forEach> attribute) (items (<c:forEach> 属性)) 449**  
**iterating, a simple tag body (迭代, 一个简单标记体) 520**  
**IterationTag 530, 537**

**J**

j\_password 679  
 j\_security\_check 679  
 j\_username 679  
**J2EE 65**  
**J2SE 1.4 xxvi**  
**JavaBean, standard actions (JavaBean, 标准动作) 参见 standard actions (标准动作)**  
**Java EE 1.5 exam (Java EE 1.5 考试) xxviii–xxix**  
**JNDI 747**  
**JSESSIONID 232**  
 URL rewriting (URL重写) 237–239  
**JSP**  
 a first look (初览) 87  
 becomes a servlet... (成为servlet...) 283  
 comments (注释) 304  
 Container-generated servlet (容器生成的servlet) 293, 297  
 declarations (声明) 295

- directive (指令) 287  
 error handling (错误处理) 468-471  
   <c:catch> 472-474  
 expressions (表达式) 288-289  
 initialization (初始化) 310  
   jspInit() 310  
     servlet init params (servlet初始化参数) 310  
 jspInit() 310  
 lifecycle (生命周期) 306  
 page directive (page指令) 287  
 scriplet 288  
 translation and compilation (转换和编译) 308
- JSP 2.0 xxvi  
 JspContext 312. 参见 PageContext  
 JSP Document (JSP文档) 629  
 JSP Expression Language (JSP表达式语言) 369, 384. 参见 EL  
 JSP expression tag, null user (JSP表达式标记, null用户) 445  
 JspFragment 522  
 jspInit() 310  
 \_jspService 297  
 JspTag 530  
 JSTL 475  
 JSTL 1.1 xxvi  
 JSTL tags (JSTL标记)  
   <c:catch> 472  
   <c:choose> 454  
   <c:forEach> 447  
     items attribute (items属性) 449  
     var attribute (var属性) 449  
   <c:if> 451  
     test attribute (test属性) 451  
   <c:import> 460  
   <c:otherwise> 454  
   <c:param> 463  
   <c:remove> 458  
   <c:set> 455  
     gotchas 457  
   <c:url> 465  
   <c:when> 454  
     test attribute (test属性) 454
- L**
- lang attribute (<select>) (lang属性(<select>)) 550  
 le (EL operator) (le(EL操作符)) 396  
 lifecycle (生命周期)  
   Classic tag handlers (传统标记处理器) 533  
   JSP 306
- methods (servlet) (方法 (servlet)) 98  
   doGet() 99  
   doPost() 99  
   init() 99  
   service() 99  
 session (会话) 255  
 listeners (监听者)  
   callback methods. 参见 methods, listener callbacks  
   examples (示例) 261  
   HttpSessionActivationListener 182, 260, 263  
   HttpSessionAttribute Listener 182, 262  
   HttpSessionBindingListener 182, 256, 263  
   HttpSessionListener 182, 261  
   listener events (监听者事件)  
     HttpSessionBindingEvent 182  
     HttpSessionEvent 182  
     ServletContextAttributeEvent 182  
     ServletContextEvent 182  
     ServletRequestAttributeEvent 182  
     ServletRequestEvent 182  
   ServletContextAttributeListener 182  
   ServletContextListener 166, 182  
   ServletRequestAttributeListener 182  
   ServletRequestListener 182  
   session (会话) 255  
   table of session-related listeners (会话相关监听者表) 264  
   The eight listeners (8个监听者) 182  
 lt (EL operator) (lt(EL操作符)) 396

## M

- mapping (映射)  
   filters in the DD (DD中的过滤器) 710  
   servlets (a first look) (servlet (初览)) 46-47  
   servlets in the DD (DD中的servlet) 616  
 Matchmaking Site (速配网站) 50  
 META-INF 613  
 methods (方法)  
   listener callbacks (监听者回调)  
     attributeAdded 182  
     attributeRemoved 182  
     attributeReplaced 182  
     contextDestroyed 182  
     contextInitialized 182  
     requestDestroyed 182  
     requestInitialized 182  
     sessionCreated 182  
     sessionDestroyed 182  
     sessionDidActivate 182  
     SessionWillPassivate 182

- valueBound 182
- valueUnbound 182
- servlet lifecycle API (servlet生命周期API) 98
- migration (session) (迁移 (会话)) 257
- MIME 17
  - <mime-mapping> in the DD (DD中<mime-mapping>) 633
  - content type (内容类型) 130
- Mock Exam, request and response (模拟测验, 请求和响应) 141
- Mock Exam Questions (模拟测验问题)
  - Q10 confidential data (Q10机密数据) 796, 832
  - Q15 outer tag handler (Q15外部标记处理器) 799, 835
  - API
    - Q17 session reference (Q17会话引用) 800, 836
    - Q18 req reference to HttpServletRequest (Q18HttpServletRequest引用) 800, 836
    - Q20 session reference (Q20会话引用) 801, 838
    - Q28 HttpServletRequest 806, 842
    - Q33 servlet lifecycle (Q33servlet生命周期) 808, 844
    - Q39 HttpServletRequest 811, 847
    - Q42 event types (Q42事件类型) 813, 849
    - Q45 doGet() method (Q45 doGet()方法) 814, 850
    - Q47 HttpServletResponse, streaming output data (Q47 HttpServletResponse,流式输出数据) 814, 850
    - Q65 creating filter for Java EE web application (Q65为Java EE web应用创建过滤器) 824, 860
    - Q67 ServletContextListener 825, 861
  - Core J2EE (核心J2EE)
    - Q05 Business Delegate object and Service Locator object (Q05业务委托对象和服务定位器对象) 795, 831
    - Q32 Java EE patterns (Q32 Java EE模式) 808, 844
    - Q35 MVC in Java EE n-tier application (Q35 Java EE n层应用中的MVC) 809, 845
    - Q56 network calls (Q56网络调用) 819, 855
- HTTP 1.1
  - Q09 HttpServlet 796, 832
  - Q22 HTTP GET versus HTTP POST (Q22 HTTP GET与HTTP POST) 803, 839
- JSP 8, Q01 file location (JSP 8, Q01文件位置) 792, 828
- JSP v2.0
  - Q02 EL 792, 828
  - Q03 tag definitions (Q03标记定义) 793, 829
  - Q04 replacing Servlet code (Q04替换Servlet代码) 794, 830
  - Q07 testing browser windows (Q07测试浏览器窗口) 795, 831
  - Q08 ServletContext 796, 832
  - Q12 JSP Document (Q12 JSP文档) 798, 834
  - Q13 HTML output of JSP page (Q13 JSP页面的HTML输出) 798, 834
- Q16 HTTP 500 status code (Q16 HTTP 500状态码) 800, 836
- Q19 Classic tag handler (Q19传统标记处理器) 837
- Q24 scriptlets and strings (Q24 scriptlet和串) 804, 840
- Q25 removing scriptlet code (Q25去除scriptlet代码) 804, 840
- Q26 <c:otherwise> 805, 841
- Q27 business listing directory (Q27企业名单目录) 806, 842
- Q31 custom tags (Q31定制标记) 808, 844
- Q36 JSP page, importing data types (Q36 JSP页面, -人数据类型) 810, 846
- Q41 removing JSP scriptlets (Q41去除JSP scriptlet) 812, 848
- Q44 translating JSP (Q44转换JSP) 813, 849
- Q49 taglib directives (Q49 taglib指令) 815, 851
- Q51 <jsp-config> 816, 852
- Q55 converting JSP to JSP Document (Q55 JSP转换JSP文档) 819, 855
- Q61 including file contents (Q61包含文件内容) 825
- Q62 search functionality (Q62搜索功能) 822, 858
- Q63 testing tags and scriptlets (Q63测试标记和scriptlet) 822, 858
- Q66 splash page (Q66醒目页面) 824, 860
- JSTL v1.1
  - Q52 shopping cart items (Q52购物车商品) 817, 851
  - Q57 XML tags, displaying (Q57 XML标记, 显示) 820, 856
  - Q64 session scoped attribute (Q64会话作用域属性) 823, 859
- Servlet
  - Q06 session listeners (Q06会话监听者) 795, 831
  - Q07 testing browser windows (Q07测试浏览器窗口) 795, 831
  - Q11 Java EE DD 797, 833
  - Q14 HTTP session support (Q14 HTTP会话支持) 798, 834
  - Q21 WAR file (Q21 WAR文件) 802, 838
  - Q23 DD fragment (Q23 DD片段) 803, 839
  - Q29 Java EE web application browser request (Q29 Java EE web应用浏览器请求) 807, 843
  - Q30 deployment descriptor (Q30部署描述文件) 843
  - Q33 servlet lifecycle (Q33 servlet生命周期) 808, 842
  - Q34 Java EE .war file's directory structure (Q34 Java E .war文件的目录结构) 809, 845
  - Q37 Java EE web application security features (Q37 Java EE web应用安全特性) 810, 846
  - Q38 Java EE application directory structure (Q38 Java E 应用目录结构) 811, 847
  - Q40 Tag File (Q40标记文件) 812, 848

- Q43 servlet lifecycle (Q43 servlet生命周期) 813, 849  
 Q46 updating live, running servlet's initialization parameters (Q46 实时更新, 运行servlet的初始化参数) 814, 850  
 Q48 <security-constraint> (Q48 <security-constraint>) 815, 851  
 Q50 resp reference to HttpServletResponse (Q50 HttpServletResponse的响应引用) 816, 852  
 Q53 myVar reference (Q53 myVar引用) 818, 854  
 Q54 deployment descriptor (Q54 部署描述文件) 818, 854  
 Q58 session migration (Q58 会话迁移) 820, 856  
 Q59 invoking filters for request (Q59 为请求调用过滤器) 821, 857  
 Q60 comparing servlet initialization parameters to context initialization parameters (Q60 比较servlet初始化参数和上下文初始化参数) 821, 857  
 Q67 ServletContextListener 825, 861  
 Q68 responses to requests (Q68 响应与请求) 826, 862  
 Q69 <security-constraint> 827, 863  
 TagSupportAPI, Q19 Classic tag handler (TagSupportAPI, Q19 传统标记处理器) 801, 837  
 mod 396  
 Model, a first look (模型 (初览)) 54  
 multiple attribute (<select>) (multiple属性(<select>)) 550  
 MVC  
     a first look (初览) 53  
     more serious look (更正式的介绍) 763

## N

- name attribute (<select>) (name属性(<select>)) 550  
 ne (EL operator) (ne(EL操作符)) 396  
 NONE 684  
 null (in EL) 399  
 null values (null值) 445

## O

- Objectives (official exam) (大纲 (正式考试))  
     Building a Custom Tag Library (构建定制标记库) 500  
     Building JSP pages using tag libraries (使用标记库构建JSP页面) 440  
     Building JSP pages using the Expression Language (EL) and Standard actions (使用表达式语言 (EL) 和标准动作构建JSP页面) 344  
 Filters (过滤器) 702  
 High-level Web App Architecture (高层web应用体系结构) 38

- J2EE Patterns (J2EE模式) 738  
 JSP Technology Model (JSP技术模型) 282  
 Servlets & JSP overview (Servlets & JSP概述) 2  
 Session Management (会话管理) 224  
 The Servlet Technology Model (Servlet技术模型) 94  
 The Web Container Model (Web容器模型) 148  
 Web Application Deployment (Web应用部署) 68, 602  
 Web Application Security (Web应用安全性) 650  
 onblur attribute (<select>) (onblur属性(<select>)) 550  
 onchange attribute (<select>) (onchange属性(<select>)) 550  
 onclick attribute (<select>) (onclick属性(<select>)) 550  
 ondblclick attribute (<select>) (ondblclick属性(<select>)) 550  
 onfocus attribute (<select>) (onfocus属性(<select>)) 550  
 onkeydown attribute (<select>) (onkeydown属性(<select>)) 550  
 onkeypress attribute (<select>) (onkeypress属性(<select>)) 550  
 onkeyup attribute (<select>) (onkeyup属性(<select>)) 550  
 onmousemove attribute (<select>) (onmousemove属性(<select>)) 550  
 onmouseout attribute (<select>) (onmouseout属性(<select>)) 550  
 onmouseover attribute (<select>) (onmouseover属性(<select>)) 550  
 onmouseup attribute (<select>) (onmouseup属性(<select>)) 550  
 operators (EL) (操作符 (EL)) 396  
 out implicit object (out隐式对象) 298

## P

- PAGE\_Scope 312  
 PageContext, API 312  
 pageContext 311  
     get/set attributes (get/set属性) 313  
 page directive (page指令) 287  
     attributes (属性) 315  
         contentType 315  
         errorPage 315  
         import 315  
         isErrorPage 315  
         isThreadSafe 315  
 page scope (页面作用域) 311  
 param, EL implicit object (param, EL隐式对象) 386  
 param attribute (param属性) 360  
 parameters (参数)  
     context parameters (上下文参数) 157

form parameters (a first look) (表单 (初览)) 119–120  
 init parameters (初始化参数) 150–151  
     in a JSP (JSP中) 310  
 paramValues, EL implicit object (paramValues, EL隐式对象) 386  
 Parent 568  
 POST 16  
     and forms (表单) 117  
     not idempotent (非幂等) 116  
     parameters (参数) 119  
     vs. GET (differences) (与GET (差别)) 110  
 prefix (taglib directive attribute) (prefix (taglib指令属性)) 393  
 PrintWriter 132

## R

redirect (重定向) 134–136  
 relative URL (相对URL) 136  
 Request. 参见 HttpServletRequest, ServletRequest  
     and threads (线程) 101  
     API 106, 122  
     attributes (属性) 187  
         thread-safety (线程安全) 204  
     getSession() 233  
     introduction (介绍) 12–13  
     RequestDispatcher 206–207  
     scope (作用域) 186–187  
     wrappers (包装器) 719  
 REQUEST\_Scope 312  
 requestDestroyed 182  
 request dispatch 138  
 RequestDispatcher 206–207  
     forward() 206  
     include() 206  
 requestInitialized 182  
 requests (请求)  
     queuing (排队) 201–202  
     sending through pool (通过池发送) 201–202  
 requestScope, EL implicit object (requestScope, EL隐式对象) 388  
 Response 126. 参见 HttpServletResponse; ServletResponse  
     API 106  
     introduction (介绍) 10–11  
     wrappers (包装器) 719  
 response filter (响应过滤器) 711  
 rewriting (URL) (重写 (URL)) 237–239  
     encodeURL() 238  
 RMI 748

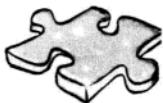
role-name 664  
 rtxprvalue 477, 480

## S

scope (作用域)  
     application/context (应用/上下文) 186  
     EL implicit objects (EL隐式对象) 389  
     page (页面) 311  
     request (请求) 186  
     session (会话) 186  
 scriptless <body-content> (无脚本的<body-content>) 482  
 scriptlet 288  
 SCWCD exam (SCWCD考试) xxviii–xxix  
 security (安全性) 653  
     <security-constraint> 670  
     <tomcat-users> file (<tomcat-users>文件) 663  
     <web-resource-collection> 666  
     constraints (约束) 665  
     data confidentiality (数据机密性) 684  
     data integrity (数据完整性) 684  
     how auth-constraint works (auth-constraint如何工作) 669  
     the Big 4 (4大要素) 653  
 security-constraint 670  
 security-role 664  
 security risks and EL (安全风险和EL) 444  
 security roles (安全角色) 664  
 SelectTagHandler 551  
 sendRedirect() 136. 参见 redirect  
 Serialization, in session migration (串行化, 会话迁移中) 260  
 service()  
     servlet method (first look) (servlet方法(初探)) 99  
     synchronizing (bad idea) (同步 (不好的想法)) 195  
 Service Locator (服务定位器) 754  
 Servlet  
     class (API) 98  
     servlet 97  
         a first look at code (代码初览) 30  
         initialization (初始化) 103  
             init parameters 150–151, 158. 参见 parameters  
             lifecycle (生命周期) 97  
             mapping (first look) (映射 (初览)) 46–47  
             mapping (the details) (映射 (详细内容)) 616  
             redirect. (重定向) 134–136  
             tutorial: simple beer Controller (教程: 简单的beer控制器) 80  
 Servlet 2.4 xxvi

ServletConfig 104, 151, 159  
 ServletContext 104, 159, 162–163, 189  
 ServletContextAttributeEvent 182  
 ServletContextAttributeListener 182  
 ServletContextEvent 168, 182  
 ServletContextListener 166, 182  
 ServletOutputStream 132  
 ServletRequest 106, 122, 189  
 ServletRequestAttributeEvent 182  
 ServletRequestAttributeListener 182  
 ServletRequestEvent 182  
 ServletRequestListener 182  
 ServletResponse 106, 126  
 servlets  
     one request at a time (一次一个请求) 201  
     specification (规范) 203  
 session (会话)  
     attributes (属性) 187  
         thread safety issues (线程安全问题) 197  
     cookies 232. 参见 cookies  
     creating/getting (创建/获取) 233–235  
     getSession() 233–234  
     intro to sessions (会话介绍) 227–229  
     invalidation (验证) 245  
     isNew() 234  
     lifecycle (生命周期) 255  
     listeners (监听者) 255  
         API 264  
     migration (迁移) 257–259  
     scope (作用域) 186  
     session ID (会话ID) 232  
     timeout (超时) 245  
 SESSION\_SCOPE (作用域) 312  
 sessionCreated 182  
 sessionDestroyed 182  
 sessionDidActivate 182  
 sessionWillPassivate 182  
 setAttribute() 186  
 setContentType() 130  
 setDir 552  
 setDisabled 553  
 setDynamicAttribute() method (setDynamicAttribute()方法) 557, 560, 561  
 setHeader() 133  
 setLang 552  
 setMaxAge(int) 251  
 setMaxInactiveInterval() 243  
 setMultiple 553  
 setName 553  
 setOnblur 554  
 setOnchange 554  
 setOnclick 552  
 setOndblclick 552  
 setOnfocus 554  
 setOnkeydown 553  
 setOnkeypress 553  
 setOnkeyup 553  
 setOnmousedown 552  
 setOnmousemove 553  
 setOnmouseout 553  
 setOnmouseover 552  
 setOnmouseup 552  
 setSize 553  
 setStyle 552  
 setTabindex 554  
 setTitle 552  
 SimpleTag 530  
 simple tags (简单标记) 513. 参见 SimpleTagSupport API 515  
     attribute (属性) 521  
     bullet points (要点) 523  
     lifecycle (生命周期) 515  
 SimpleTagSupport 513, 530  
     API 515  
 SingleThreadModel 201–203  
 size attribute (<select>) (size属性(<select>)) 550  
 SKIP\_BODY 531, 532  
 SkipPageException 523  
 specifications (规范) xxvi  
     servlets 203  
 standard actions (标准动作) 323  
     <jsp:forward> 413  
     <jsp:getProperty> 349  
     <jsp:include> 404  
         <jsp:param> 412  
     <jsp:setProperty> 351  
         param attribute (param属性) 360–361  
         property attribute (property属性) 362  
     <jsp:useBean> 349, 354  
         type and class (类型和类) 356  
     bean-related (bean相关) 348  
         bullet points (要点) 418  
 STM strategies (STM策略) 201–203  
     queuing requests versus sending through pool (请求排队与

- 通过池发送) 202
- Struts 767  
    installing (安装) 776
- struts-config.xml 774
- style attribute (<select>) (style属性(<select>)) 550
- synchronizing (同步)  
    on the context (上下文中) 197  
    service() 195
- system requirements for this book (本书系统需求) xxvi
- T**
- tabindex attribute (<select>) (tabindex属性(<select>)) 550
- tag. 参见 Classic tag handlers (标记处理器)  
    <body-content>. 参见 body-content  
    attributes (属性) 504  
    custom tag handler (定制标记处理器) 477  
        Classic tag handler API (传统标记处理器API) 530  
        simple vs. classic (简单与传统) 574  
    empty (空) 482  
    TLD element (TLD元素) 477
- tag attribute setters (tag属性设置方法) 552-554
- Tag Files (标记文件) 502  
    body-content 508  
    dynamic attributes (动态属性) 561  
    locations (位置) 509  
    tag directive (tag指令) 508
- Tag interface (Tag接口) 530
- taglib 393
- taglib directive (taglib指令) 314
- tag library (标记库) 476  
    custom. 参见 JSTL; JSTL tags  
    JSTL. 参见 JSTL; JSTL tags
- Tag Library Descriptor 392. 参见 TLD
- TagSupport 530
- TCP port 21
- templates (reusable JSP chunks) (模板 (可重用JSP块)) 402
- test <c:if> attribute (测试<c:if>属性) 451
- threads (线程)  
    and scope (作用域) 192-196  
    for requests (a first look) (请求 (初览)) 101
- timeout (session) (超时 (会话)) 245
- title attribute (<select>) (title属性(<select>)) 550
- TLD 392, 477, 483  
    for simple tags (简单标记) 513  
    locations in web app (web应用中的位置) 486
- Tomcat
- deploy/hot redeploy (部署/热重部署) 153
- deployment environment (部署环境) 73
- generated servlet (生成的servlet) 297
- starting Tomcat (启用Tomcat) 77
- Tomcat, getting and installing (Tomcat, 获得和安装) xxvi
- Transfer Object (传输对象) 759
- translation and compilation (JSP) (转换和编译 (JSP)) 308
- U**
- uri (taglib element) (uri (taglib元素)) 484
- URL  
    introduction (介绍) 20  
    relative (相对) 136  
    rewriting (重写) 237-239  
        encodeURL() 238
- url-pattern 618
- useBean 349
- V**
- valueBound 182
- valueUnbound 182
- var <c:forEach> attribute (var <c:forEach>属性) 449
- View, a first look (视图, 初览) 54
- W**
- WAR files (WAR文件) 612  
    META-INF 613
- WEB-INF 613-614
- web.xml 154. 参见 DD
- web app (web应用)  
    initialization 159. 参见 Initialization; ServletContext  
    web containers, one request at a time (web容器, 一次一个请求) 201
- welcome files (欢迎文件) 622  
    Container choosing (容器选择) 625
- Wrappers (request and response) (包装器 (请求和响应)) 719
- X**
- XML-compliant JSP (JSP document) (XML兼容的JSP (文档)) 629
- XML entities (XML实体) 443

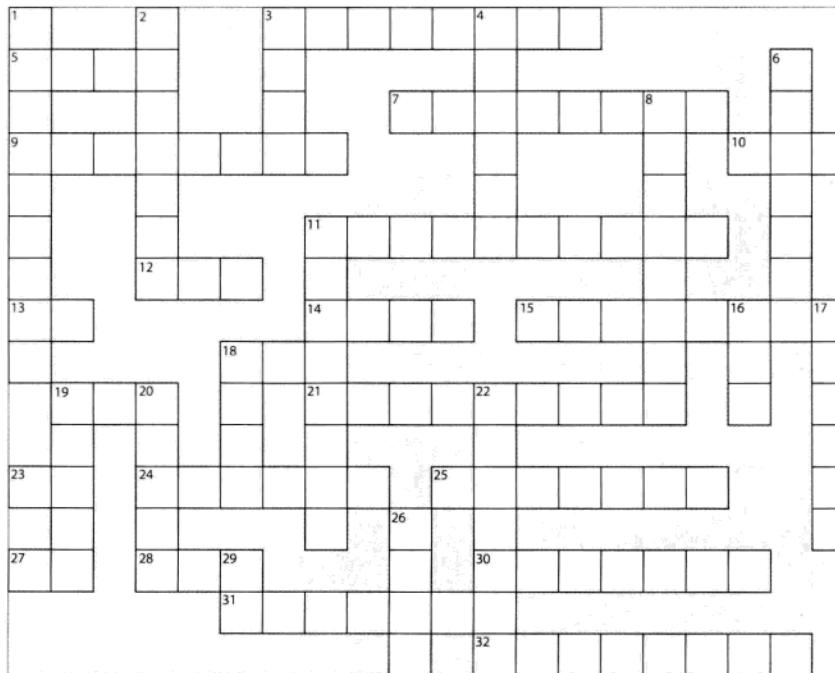


## JavaCross 2.4

让你的右脑活动一下。

这是一个标准的猜字游戏，  
谜底中的单词几乎全部来自  
于第13章和第14章。

如果你不是一个字谜高手，  
可以看一下这一页最后的  
提示。



### 横向

- 3. Performance pattern
- 5. Architecture layer
- 7. Don't invoke me
- 9. Trickier filters
- 10. HTTP's stomping ground
- 11. One per Struts app.
- 12. Socket shielder
- 13. Could be moldy (abbr.)
- 14. Browser jockey
- 15. A Struts callback
- 18. Web developer's tool

- 19. Smalltalk pattern
- 21. Some kind of event
- 23. XML hotbed
- 24. Could be a browser
- 25. Crupi's bailiwick
- 27. JavaScript inspired
- 28. Fancy models
- 30. Inter-server ether
- 31. 1st step towards looseness
- 32. Hand off

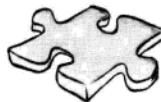
### 特别提示：

- Container invokes me
- 25. MVC is one
- 28. SCBCD stars
- 2. Local cell, note
- 3. Could be stable
- 1. All scopes have 'em
- 11. Less hard-coding
- 26. Getters and setters
- 6. XML chunk
- 31. Announce
- 3. The big mapper

### 纵向

- 1. Data communicator
- 2. Convenient class
- 3. JSP super-chargers
- 4. Stackable component
- 6. DD brick
- 8. Single minded
- 11. Declarative reducer
- 16. The blueprints
- 17. State component
- 18. The yellow pages
- 19. Blind to the GUI
- 20. SL's shortcut
- 22. Filter's fate
- 26. Validate's home
- 29. Controller protect<sup>s</sup> (abbr.)

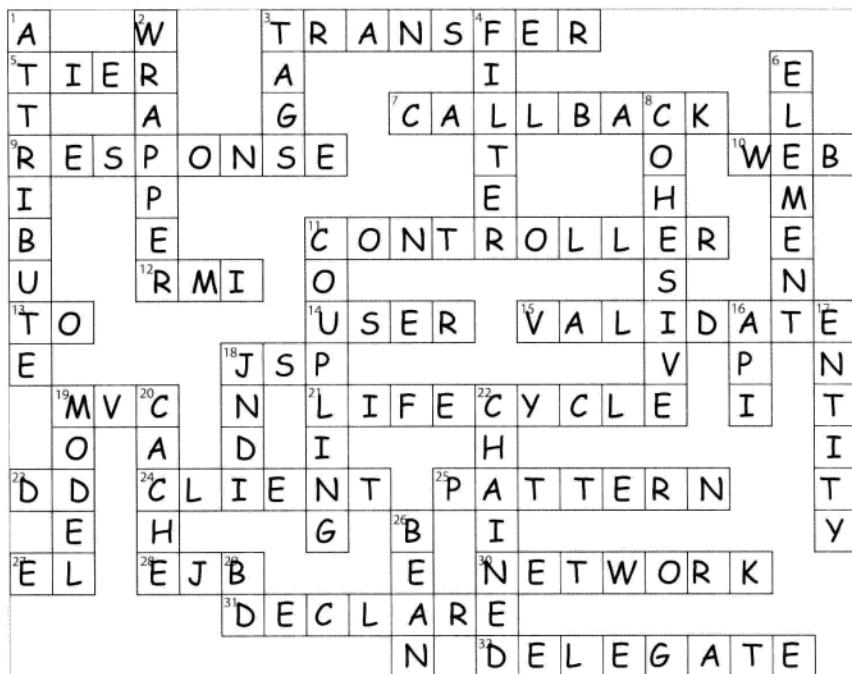
# 答案！



## JavaCross 2.4

让你的右脑活动一下。

这是一个标准的猜字游戏，谜底中的单词几乎全部来自于第13章和第14章。



### 横向

- 3. Performance pattern
- 5. Architecture layer
- 7. Don't invoke me
- 9. Trickier filters
- 10. HTTP's stomping ground
- 11. One per Struts app.
- 12. Socket shielder
- 13. Could be moldy (abbr.)
- 14. Browser jockey
- 15. A Struts callback
- 18. Web designer's tool
- 19. Smalltalk pattern
- 21. Some kind of event
- 23. XML hotbed
- 24. Could be a browser
- 25. Crupi's bailiwick
- 27. JavaScript inspired
- 28. Fancy models
- 30. Inter-server ether
- 31. 1st step towards looseness
- 32. Really fancy proxy

### 纵向

- 1. Communicator
- 2. Convenient class
- 3. JSP super-chargers
- 4. Stackable component
- 6. DD brick
- 8. Single minded
- 11. Declarative reducer
- 16. The blueprints
- 17. State component
- 18. The yellow pages
- 19. Blind to the GUI
- 20. SL's shortcut
- 22. Filter's fate
- 26. Validate's home

# 不是说再见

要用心地看看wickedlysmart.com。

你不知道wickedlysmart.com网站上还有很多好东西吧？如果你想顺利通过考试，一定要先去javaranch.com看看，在SCWCD学习论坛里花点时间。那里的人真的非常非常友好，你肯定会喜欢的。

通过考试后别忘了写信告诉我们！

Ikickedbutt@wickedlysmart.com  
我们会为你喝上一杯。



老魏  
PDG

[ General Information ]

书名=Head First Servlets and JSP 中文版 第2版

作者=(美)巴萨姆(Basham, B)等著;荆涛等译

页数=879

出版社=北京市:中国电力出版社

出版日期=2010.08

S S号=12738976

D X号=000007646994

URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=0000

07646994&d=26D6B6D9AB5E530A0C27020A47A5CAB9

- 1 为什么使用 S e r v l e t s & J S P : 前言与概述
- 2 W e b 应用体系结构 : 高层概述
- 3 M V C 迷你教程 : M V C 实战
- 4 作为 S e r v l e t : 请求和响应
- 5 作为 W e b 应用 : 属性和监听者
- 6 会话状态 : 会话管理
- 7 作为 J S P : 使用 J S P
- 8 没有脚本的页面 : 无脚本的 J S P
- 9 强大的定制标记 : 使用 J S T L
- 1 0 J S T L 也有力不能及的时候 : 定制标记开发
- 1 1 部署 W e b 应用 : W e b 应用部署
- 1 2 要保密 , 要安全 : W e b 应用安全
- 1 3 过滤器的威力 : 过滤器和包装器
- 1 4 企业设计模式 : 模式和 s t r u c t s
- A 附录 A : 最终模拟测验
- i 索引