*Dojo: The Definitive Guide*

# Dojo

## 权威指南

*Matthew A. Russell* 著

李松峰 李丽 译

O'REILLY®

机械工业出版社 China Machine Press

HZ BOOKS

Web Development

O'REILLY®

# Dojo权威指南

通过使用Dojo这个工业强度的JavaScript工具箱，我们可以比使用其他任何Ajax框架更高效、更容易地创建JavaScript或Ajax驱动的应用程序和站点。

本书向读者展示了如何充分利用Dojo工具箱中包含的大量实用特性，以前所未有的效率开发出功能丰富、响应敏捷的Web应用程序。读者通过本书能够学习到创建复杂布局和表单控件（常见于高级桌面应用程序）的技巧，掌握精妙的JavaScript独有特性和通信机制。另外，读者还可以：

- 了解适用于Dojo1.x版本的简明介绍。
- 研究大量Dojo应用实例及经过测试的代码。
- 探索Dojo的标准JavaScript库和基础实用程序。
- 学习拖放、后退按钮处理及动画。
- 创建并利用Dijit（Dojo部件）。
- 浏览DojoX子项目、构建工具和Dojo的单元测试框架。

无论读者是使用DHTML构建Web应用程序的自由开发人员，还是大型开发团队中的一员，本书都可以帮你利用已知的设计理念，将自己的构想迅速付诸实践。

**“Matthew不仅以简洁流畅的文笔深入浅出、通俗易懂地讲解了Dojo工具箱，而且他在遇到不正常情况时提出的问题也促进了Dojo工具箱的完善。他的建议推动了Dojo的发展。本书的确是名副其实的Dojo权威指南。”**

——Dylan Schiemann，
SitePen,Inc., CEO，
Dojo工具箱共同创始人

Matthew A. Russell是一位计算机科学家，他目前居住在美国田纳西州的富兰克林市。作为一位博学多才的专家，他也非常喜爱写作和研究前沿技术。

投稿热线：(010) 88379604
购书热线：(010) 68995259，68995264
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

**www.oreilly.com**

O'Reilly Media, Inc.授权机械工业出版社出版



ISBN 978-7-111-26380-7
9 787111 263807
定价：79.00元

# Dojo 权威指南

*Matthew A.Russell* 著

李松峰 李丽 译

O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo*

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

**图书在版编目（CIP）数据**

Dojo权威指南 /（美）拉塞尔（Russell, A.）著；李松峰等译．－北京：机械工业出版社，2009.4

（O'Reilly精品图书系列）

书名原文：Dojo: The Definitive Guide

ISBN 978-7-111-26380-7

I. D… II. ①拉… ②李… III. JAVA语言－程序设计 IV. TP312

中国版本图书馆CIP数据核字（2009）第022328号

北京市版权局著作权合同登记

图字：01-2009-1616号



书　　名/ Dojo权威指南

书　　号/ ISBN 978-7-111-26380-7

责任编辑/ 陈佳媛

封面设计/ Karen Montgomery，张健

出版发行/ 机械工业出版社

地　　址/ 北京市西城区百万庄大街22号（邮编100037）

印　　刷/ 北京京师印务有限公司印刷

开　　本/ 178毫米×233毫米　16开本　29.75印张

版　　次/ 2009年4月第1版　2009年4月第1次印刷

印　　次/ 0001-3000册

定　　价/ 79.00元（册）

**凡购本书，如有倒页、脱页、缺页，由本社发行部调换**

**本社购书热线：(010)68326294**

# O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

# 译者序

平心而论，与现有的JavaScript库或框架相比，Dojo的确很值得O雍eilly为它自己出一本权威指南。

从译者的Web开发经验来看，在使用JavaScript语言创建跨平台、跨浏览器的RWA（Rich Web Applications，富Web应用程序）项目时，选择Dojo作为前端开发框架基本上可以做到别无所求。因为，对于通常的JavaScript库和框架所能解决的问题蜒DOM操作、事件处理、样式修改、外部通信的标准化，Dojo的Base和Core同样也给出了全套解决方案。

当然，就Dojo工具箱的库和框架部分而言，数据抽象和模拟类与继承是另外两个主要的亮点。

除了对开发RIA项目的底层逻辑提供强大支持外，Dijit专门针对设计人员给出大量即装即用的“部件”。部件就是HTML、JavaScript、CSS和其他相关资源（如图片）的集合，也是一个Function对象。基于Dojo部件的用户界面还支持换肤功能。

Util是Dojo独有的，其中包含构建工具、单元测试框架和压缩工具。为优化、测试产品和提高RIA项目的性能提供了有效支持。

在翻译本书的过程中，为确保技术细节的准确，译者参考了《Mastering Dojo》(The Pragmatic Bookshelf)一书。而且，全书的术语基本上做到了前后统一。需要说明的是，书中将Dojo命名空间之下的所有函数（function）统一译为方法。虽然翻译为函数也未尝不可，但译者在此有两点考虑：首先，从面向对象的角度讲，通过对象访问的函数应该叫方法；其次，Dojo跟其他的库和框架一样，都非常强调命名空间，翻译为方法有助于读者进一步明确这一思想。另外，书中在谈及易访问性（Accessibility）时，多次提到了“退化能力”。对此存疑的读者，可以参考译者网站中的相关讨论和介绍。

从翻译一本书的角度讲，序和前言部分一般是最难翻译的。译者在翻译Dojo之父Alex Russell为此书作的序时，得到了李锟、郭晓刚、米全喜、贺师俊的帮助，特别是米全喜抽时间审阅了全篇译文，提出了9处问题。在此一并致谢！

本书由李松峰负责翻译，参加翻译工作的还有李丽、秦绪文、程宝杰、宋连海、付荣艳、左艳坡、刘英、李炜、李雅雯、熊俊芹、封耀杰、贾爱华等。译者的网站：

*http://www.cn-cuckoo.com*；电子邮箱地址：*lsf.email@yahoo.com.cn*。欢迎读者朋友登录译者网站或通过邮件反馈本书翻译中存在的问题，或者提供勘误信息。

另外，在本书中文版交稿时，Dojo1.2 已经发布。译者翻译了“Dojo1.2 发版说明”，有需要的读者可以参考，地址为 *http://www.cn-cuckoo.com/2008/12/18/dojo-1-2-release-notes-265.html*。

译者

2008 年 12 月 18 日于北京

# 作者简介

**Matthew Russell** 是一位具有顽强创业精神的技术专家。目前，他已经完成了40多项技术成果，有的已经发表，有的即将在学术会议及Linux Joumal、Apple Developer Connection、*Make:*等杂志发表。

在United States Air Force Academy（美国空军军官学校）上大学时，Matthew培养起了自己对写作的兴趣。在校期间，他作为主修课程计算机科学的最佳学生，荣获了以Dean W. Gonzalez命名的声望颇高的奖项Dean W. Gonzalez Award。

最近，Matthew组建并领导了一个团队，该团队致力于为智能社区构建按等级加密的端到端的Web应用程序。同时，他还为Defense Intelligence Agency（美国国防情报局）工作，从事于构建政府智能系统的下一代技术的研究和评估。

Matthew当前是Digital Reasoning Systems的一名高级技术主管，他在该公司负责开发基于Web浏览器的用户界面和非结构化文本的高端技术研究。

读者可以订阅Matthew的非正式专栏“Dojo Goodness”，以跟进他撰写的最新Dojo文章。专栏地址为：*http://pipes.yahoo.com/ptwobrussell/dojo_goodness*（译注1）。

# 封面介绍

本书封面上的动物是狮尾猴（英文名Macaca silenus）。这种罕见的猴子主要生活在印度南部的雨林中。它们终日避不见人，食草，喜探寻。

银白色的鬃毛和尾巴上丛生的毛簇是这种猴子（也称为狮尾猿或印度猴）的主要特征。它们按照长幼次序结群而居，每群约10_20只不等，而且雄猴少，雌猴多。雄猴负责保卫和抵御外来侵略，它们会对入侵自己家园的猴子或非本族群的动物发出尖叫声。

---

译注1：在翻译本书时，译者在此链接中找不到相关文章。不过，读者也可以访问 *http://dojotdg.zaffra.com/2008/09/10-helpings-of-dojo-goodness/* 或 *http://tinyurl.com/dojo-tdg-forum* 获取相同的信息。

根据国际自然保护联盟（International Union for Conservation of Nature）公布的资料，狮尾猴是世界上最濒危的灵长类动物，主要原因是人类对它们生活环境的破坏。目前，为了维持这种猴子的生存，许多公园都加入了人工饲养计划。

封面上的图片选自Lydekker's Royal History。封面字体使用的是Adobe ITC Garamond。正文字体是Linotype Birka，标题字体是Adobe Myriad Condensed，代码字体是LucasFont中的TheSans Mono Condensed。

# 目录

# 第二部分 Dijit与Util

# 序

实话实说，我是因为DHTML才中途退学的。

当时，经常是时钟已经指向了凌晨3点，我还在不停地搜索MSDN文档、W3C规范以及数以百计的comp.lang.javascript帖子，然后又将这些来源的信息综合起来，发表几十个类似于“要是……会怎么样”的帖子。就好像是命中注定的一样，我对每个小小的发现都不轻易放过，除非浏览器能够听命于我，否则决不善罢甘休。那时候，一群志同道合的人在一个小社区中做着同样的事情，大家争先恐后地将每个新发现、新技术或一些能够让Netscape正常运行的小技巧在DHTMLCentral论坛中进行共享。与早晨7点有关拉丁文动词变化的讨论和无休止的Java讲义相比，真正吸引我的还是发现闭包的妙处，或者最终完全理解基于原型的继承。就连整个圣诞节假期，我都沉迷于JavaScript的学习和研究当中。我知道女朋友和父母肯定非常担心我，可他们谁也没有让我分心。于是，在付出了中断学业的代价之后，我对开源（*http://opensource.org*）有了深刻的理解，也收获了持久的友谊，最终还创建了Dojo。

时过境迁，DHTML黑客的工作也发生了变化。我们掌握了让浏览器听命于自己的大多数技巧，也知道了浏览器之间在哪些地方有重叠之处。我们已经有能力利用它们了……看看Dijit和DojoX中模块的深度和多样性就能体会到这一点。当前，DHTML/Ajax黑客的工作是如何把有效的技术纳入对用户和开发人员的服务当中，而且采取的方式也要比针对终端用户和准开发人员的方式更好。Dojo的故事其实就是这种技术过渡的故事。一个架构，无论它本身有多好，如果不能有效地将它交付给用户，结果还是失败。同样，外表漂亮但却难以维护的图形及界面也不可能得到开发人员的一致认同。如果设计人员或开发人员因此而难于协作，当然无益于我们愿望的达成。令人欣慰的是，Dojo开发团队正在和Web一起走向成熟，而随着Dojo1.0发布和本书的问世，现在我们终于可以自信地说，Dojo已经完全抵达了成功的彼岸。我们很久以前创立的路线图文档，终于在今

天划上了完美的句号。今天，依赖于Dojo创造的完整用户体验，已经带来了每月数十亿的页面流量。而且，很多大型设计人员和开发人员团队正在以这个工具箱为依托，齐心协力地创造更加美好的体验。

这些成绩的取得绝非一人之功，甚至都不是一个小团队所能企及的。在Dojo发展过程中曾经对它有过贡献（同时对它抱以坚定信心）并致力于共同创造一个更美好的Web的人实在太多了。我们从其他项目中借鉴了我们认为最好的思路和做法，营造了一个高水平的竞技平台，而这个平台的规则无论是对用户、贡献者，还是对发起者都是一视同仁的。Dojo以不争的事实再次向世人证明，开源项目不仅仅是封闭系统可以利用的一个方便的分发模型，其本质在于协作。只要推行用户认同的规则，并且在项目成员之间达成应有的默契，开源项目就能够发展兴盛。在这个工具箱所体现出的各种技术性成果中，我认为最值得自豪的是，我们以真正开放（包括今后可能参与进来的人）、公正的方式实现了它。事实上，这个项目在创建之初，我们就宣布要切实尊重任何形式的贡献，不能仅局限于代码。而且，这个项目也要改变开源软件的发展基调，例如，它应该鼓励面向大学和民间的对话。此外，它应该是一个依赖于社区构建的项目，因而不能把用户看成“他们”——事实上，“他们”即“我们”。鉴于本书明确地传达了这种Dojo工具箱赖以存在的、朴素的开放理念，我们衷心地期待本书读者能够为Dojo将来的发展出谋划策。

在我和Matthew Russell面对面坐在一起的时候，本书差不多快要“出炉”了。开源项目好像就是那么有意思——虽然你和某人共事了好多年，但邮件列表和IRC中的那些只言片语，如果不是因为要谈论本地麦酒（或者，必要时的健力士黑啤）的流行和刺激性口味，恐怕永远都不会发现它们已经落到了实处。直到Matthew和我在旧金山北滩的一家舒适的老式小酒馆中交换意见，我才回到现实：他对技术理解的深度、强烈的求知欲和针对不同读者层次展开讨论的能力，完全够得上一位好老师的标准。因为我是依次审读每一章的草稿，我发现随着审读的进行，自己会根据Matthew逐步展开的阐述陆续删除前面所写的批评意见。Matthew的阐述令Dojo更加平易近人、友好和善，也更加突显了其高效及强大。当读者在阅读本书并与Matthew神交之时，不断迸发的、明亮的思想火花所带来的会心一笑，恐怕就是本书最大的礼物了。

像这样坚持与IRC消息及论坛文章的作者见面差不多已经有4年了。在开源项目中，你所接触的每个人几乎都表现为要解决的技术问题、要修复的bug，或者应该要考虑的什么特性。但是，唯有面对面接触才是了解合作者的最佳方式，而且几乎总能让人兴奋不已。每当此时，那些所谓的仁慈、无私、天才等廉价的口号只能让人感到羞愧难当，特别是在一个人人有份的项目中提到个人牺牲的时候尤其如此。Matthew的这本书让我和我所尊重的优秀团队感到十分荣幸。

我并不想建议读者像我一样为了追求某个理想（而且没人会为此付钱给你）就冒然中断自己的学业。但是，如果有一天你的头脑中真地燃起了星星之火，千万不要忽视它。哪

怕你因此结识只有我所结识的、妙不可言的、现在已成为我朋友的人的一半，那么再多的不眠之夜也是值得的。

Alex Russell
Dojo Tootkit 共同创始人，
Dojo 基金会主席

# 前言

用户要求现在的Web应用程序应该像桌面应用程序一样。家用计算机已经无处不在，而且Web浏览器是一个开放性平台，因此可以说几乎全世界的每个人都可能成为Web应用程序的终端用户。软件开发人员为创建基于浏览器的应用程序，要花上比过去多得多的时间，因为他们必须满足潜在的百万级用户群的需求——其中很多人手里掌握着数十亿美元的广告预算，而另外一些人则会利用应用程序优雅便捷的特性，吸引更多的人前来浏览访问。

当然，就因为Web浏览器是开放性平台，并不意味着它是理想的平台——至少当前还不够理想。任何团体利益都不应该凌驾于对各种Web浏览器技术规范的统一实现之上。但让人难以理解的是，相关协议和标准在经历了近20年的发展后，反而导致在浏览器中部署应用程序变得更加困难。这种局面无论如何是任何人都不可能预知的。

但是，在一个拥有想像力和创造力的世界里，总还是有希望的。

幸运的是，JavaScript提供的丰富而强大的功能使得动态操作、定制和增强网页成为可能。而在此基础之上，为开发人员和Web浏览器的不兼容性之间提供一个隔离层也成为了可能——即使是隔离所有的浏览器。

本书将要介绍的Dojo是一个JavaScript工具箱，它就为开发人员和浏览器的不兼容性之间提供了这样的一个隔离层。为此，Dojo结合了JavaScript和其他有效的Web技术——但是，它并非一个试图重新实现或解决浏览器问题的暂时的表层。无论是那些已经使用了YUI!（注1）的项目，还是因为把负载转移到了客户端而受益的服务器端框架，都能通过Dojo获得更多的特性。

---

注1：*http://developer.yahoo.com/yui/*。

Dojo内置了开发人员由衷期望的标准JavaScript库、大量常见的自定义HTML控件的替代物和CSS布局的高级解决方案，以及方便发布应用程序的构建工具和单元测试框架。Dojo不仅是一个JavaScript工具箱，而且是一个真正的JavaScript工具箱——现在正是学习如何通过它来解放自己并为用户提供各种体验的最佳时机。Dojo正发动Web开发的革命，也必将推动Web开发的迅猛发展。

无论你想实现什么样的Web开发项目，Dojo都能帮助你迅速地把它变为现实，并且保持最低的冗余代码。因此，你得到的将是一个最干净、最容易维护的项目。我最大的希望是读者能够通过本书以最少的时间领会Dojo的本质，并且能够充分利用它完成自己面对的艰巨任务。

## 为什么选Dojo

当然，目前可供选择的JavaScript工具箱有好几个，那么读者恐怕早就想问“为什么Dojo可以让我们别无所求”这个问题了。从理论上说，基于一门纯解释型语言构建的工具箱或库，它所能做到的事情其他工具箱或库也同样能够做到，很难说其他工具箱做不到的而Dojo能做到。但是，Dojo在它的社区支持、指导思想和许可方式等方面确有其独到之处。

也许读者可以这样来想：理论上，只要有一把锤子、一把铲子和足够的钉子就能够建起房子来，但是那要花多大代价呢？显然，在建房的过程中，如果有机械设备和木匠作伴，那么这个过程就完全不一样了。这个比喻对于Dojo同样适用。接下来，我们就尝试从以下几个方面介绍Dojo的闪光点和它的特征（先后顺序不代表重要性）：

*社区支持*

虽然可以把这一条归结为非技术因素，但Dojo的开放性社区的确是它最大的优势。Dojo基金会作为一个非赢利组织，其设立宗旨是作为厂商中立的Dojo知识产权的监护人，是由IBM、AOL、Sun、OpenLaszlo、Nexaweb、SitePen、BEA、Renkoo和世界各地的DHTML黑客发起和支持的。该基金会资助了Dojo工具箱（以及其他值得关注的项目，如Cometd（注2）、DWR（注3）和Open-Record（注4））。如果这些还不能说明Dojo拥有的强大支持，那么还有什么可以说明呢？

Dojo作为一个自由许可的开源项目，加之其极低的加入门槛，任何人都可以在它

注2：参见 *http://www.cometd.com* 或 *http://www.cometdaily.com* 了解对Comet的全面介绍。

注3：参见 *http://getahead.org/dwr* 了解有关Direct Web Remoting的更多信息。

注4：参见 *http://www.openrecord.org* 了解OpenRecord的资料。

的社区中发出自己的声音（如果愿意让别人听到的话）。如果登录IRC并进入*freenode.net*的*#dojo*聊天室说话，要想不让该项目的开发人员或参与者听到也是不太可能的。此外，每周一次的IRC会议目前会于每周三下午3点到6点（太平洋标准时间）在*#dojo-meeting*聊天室举行。这个正式的官方会议讨论的内容既有战略规划，也有战术分析。如果你在开会期间冒然闯进会议室，绝不会遭到拒绝。你可以在里边“偷听”，也可以大胆地主动发言。

能够了解Dojo在战略规划和战术落实方面的信息是非常有吸引力的。在其他JavaScript工具箱和库日益商品化的背景下，Dojo社区越来越显得独树一帜。无论是相关组织，还是开发团队中的一员（更不必说当前正在构建真实站点和应用程序的数千名开发人员），都能通过对Dojo献计献策为它将来的成功夯实基础。

*自由（彻底的）许可*

Dojo作为开源软件，任何人或组织都可以根据BSD（Berkeley Software Distribution，伯克利软件分发）许可修订版或者AFL（Academic Free License，自由研究许可）2.1版获得不受限制的许可。除了某些个别模块中包含的许可文件之外，开发人员可以根据自己的工作选择想要采用的许可方式。所有外部的参与者必须遵守BSD或AFL许可，所有参与者还必须签署CLA（Contributor License Agreement，贡献者许可协议），以确保Dojo基金会对所有衍生产品具有明确的权利，从而保护Dojo工具箱的所有用户免受知识产权纠纷的困扰。这种许可的彻底性与其他流行的JavaScript工具箱相比，具有明显的不同。

*深度和广度*

与某些工具箱专注于特定的问题领域不同，Dojo为基于浏览器的开发提供了端到端的解决方案。从标准的库方法到一应俱全的部件（widget），从构建工具到测试框架，无所不包；因此，可以说有了Dojo在手，基本上别无他求。但是，千万不要以为它覆盖面如此之广，代码肯定会臃肿不堪。因为使用构建工具可以生成它的自定义版本，并能够尽量保持应用程序许可的合理化。

虽然广度的增大一般会影响到深度，但这条经验性规则对Dojo却不适用。即使是在Base（为工具箱其他部分提供基础的微小内核）中，都包含着几乎数不尽的功能，例如，基于CSS3选择符的通用DOM查询机制、Ajax实用方法、对浏览器间事件操作的规范化等。而这些甚至还没有涉及丰富的应用程序、表单库、布局部件或者构建工具。

尽管Dojo的广度和深度带来了不少复杂性，但是它的底层结构仍然由世界上最好的Web黑客在不断地用心改进。无论从代码标准的高质量、统一的命名约定，还是从性能和容易维护方面，都致力于让应用程序开发人员能够更方便地使用它。使用Dojo一定能够创造出引人入胜的用户体验。

*容易移植*

JavaScript语言虽然拥有动态性、效率高和富有表达能力的特点，但在日常开发中仍然不可避免地会出现各种各样的问题。通过反复试验来解决某个核心算法问题或者厘清设计上的含糊性固然有益，但无论如何，你所开发的代码都将成为必须维护、升级、调试和说明的代码。

仅有这些理由似乎还不够。在当前不同浏览器支持的各种特性存在兼容性问题的形势下，需要一个JavaScript标准库则是比较有说服力的动机。当前，尽管开发一种可以在所有现代浏览器中运行的功能并非难于登天，但因此所要承受的痛苦和挫折，恐怕就连经验丰富的专业人士都要望而却步。

毕竟，作为一名应用程序开发人员，你并不会因为能解决所有这些问题而希求任何回报（也许会感觉比较好玩）。相反，你应该义无反顾地选择由整整一个社区的开发人员开发的、经过调试和验证的产品级代码——然后，再考虑对这个社区有所回馈。我们衷心希望，在享受到社区支持的开源软件带来的节省时间和金钱的好处之后，这个“回馈”部分最好是发自内心的。

*实用哲学*

Dojo与JavaScript的结合并不是把它当作一个需要救治的对象，并因此以几乎重新定义它的方式来创建的一个脆弱的人造层。尽管Dojo提供了大量避免开发人员直接与浏览器引擎交互的功能，并且在后台实现了标准化浏览器事件之类的机制，从而使开发人员不必重复工作，但是它却从未试图彻底改造JavaScript。例如，Dojo没有提供删除节点或遍历DOM树之类的方法，因为`childNodes`、`firstChild`、`lastChild`和`removeChild`在所有浏览器中都能正常使用。然而，针对任何已知的不一致性，Dojo都会为开发人员编写可移植的代码提供相应的工具。

在这个问题上，Dojo并没有尝试妨碍或限制开发人员使用其他JavaScript库。实际上，Dojo与其他技术（如DWR或YUI!）协同运行的案例并不鲜见。当然，作为一种客户端技术，Dojo的服务器无关性也可以让开发人员自由选择任何服务器端技术。

通过对流行JavaScript工具箱的全面对比可以发现，正是由于它们流行，所以它们相互之间存在大量的重叠功能。因此，在决定应该选用哪个工具箱或工具箱集合时，有必要把社区支持、透明性、许可方式和指导该项技术的哲学理念等因素考虑在内。也许这个选择不仅仅意味着时间的付出，甚至会意味着金钱的投入。换句话说，你希望在需要时能够得到支持（社区和文档），而不希望投资于一个脆弱得难以维护的项目，或者说一个封闭的项目。同时，你更希望利用既有工具箱提供的现成解决方案来最大限度地节省自己的开发时间。

# 本书内容

本书第一部分基本上是标准库的参考，详细介绍了组成Dojo工具箱中JavaScript标准库的Base和Core的方方面面。Base通过网络传输（注5）时不到30KB，并且从速度、大小和实用性方面都进行了优化。其中封装的功能包括各种Ajax调用、基于CSS选择符语法的DOM查询、标准事件传播机制和函数式编程的实用程序（如map和filter）等。相信读者在接触Base后，很快就会有相见恨晚的感觉。Core中又包含了很多其他特性（如对动画和拖放的支持），这些特性都非常有用，只不过与Base中那些功能的通用性相比要稍差一些。

---

**注意：** 本书将在第11章中介绍Dijit时再详细讨论解析器（parser），原因在于解析器最常见的用途是解析部件。由于解析器在实现拖放功能时也很有用，因此在第7章的附言栏（sidebar）中将会先对它先进行简单介绍。

---

本书第一部分包含以下几章：

*第1章 Dojo工具箱概述*

本章简单介绍Dojo工具箱的基本情况，包括Dojo的架构、如何获取和在页面中使用Dojo，以及展示Dojo实际应用的一些示例。

*第2章 语言及浏览器实用程序*

本章对适用于任何Web应用程序的最为常见和实用的方法进行全面介绍。其中大部分方法主要针对消除浏览器的不兼容性问题而设计，另外一些方法是为JavaScript或DOM操作查漏补缺的，最后一些方法则有助于减少实际开发中的重复编码工作。

*第3章 事件侦听器及发布/预订通信*

本章介绍在页面内部实现通信管理的机制。讨论的两个主要模式包括在DOM中、JavaScript对象上或者基于独立的函数连接发生的事件，和广播主题并允许任意预订程序接收及响应该主题的发布/预订模式。

*第4章 Ajax及服务器通信*

本章围绕AJAX和Dojo工具箱中通过*XMLHttpRequest*对象与服务器通信的机制进行讨论。本章还讨论了Deferred对象，该对象为处理异步事件提供了统一的层；虽

---

注5： 正如本书相关章节将要介绍的，“通过网络传输的大小”指的是压缩后的大小，因为Web服务器通常会以这种方式向客户端传输网页。

然 JavaScript 不支持线程，但在某种程度上却可以把 Deferred 想像成对线程的实现。另外，跨域 JSON、远程过程调用以及 IFRAME 传输也是本章的主题。

*第 5 章 节点操作*

本章主要介绍工具箱中使用CSS选择符语法来查询DOM的机制。而针对查询返回的节点列表，又可以基于事件连缀使用一组方便的内置方法。最后，讨论了一种分离 DOM 节点的行为与在 HTML 标记中定义的具体操作的模式。

*第 6 章 国际化（i18n）*

本章结合示例介绍如何使用工具箱提供的实用程序，实现 Web 应用程序对国际化的支持。同时，也讨论了处理与国际化有自然关联的一些概念（例如，日期、时间、货币及数字的格式化）的方法。

*第 7 章 拖放*

本章提供一个基本独立的教程，展示方便地通过 Dojo 为应用程序添加拖放功能。

*第 8 章 动画和特效*

本章同样是一个基本独立的教程，讲解使用 Dojo 创建基于 CSS 属性动画的内置机制，展示一些典型的动画效果，例如，擦除、滑动和淡入淡出。此外，还介绍与颜色混合等操作有关的实用方法。

*第 9 章 数据抽象*

本章对 Dojo 提供的数据抽象架构进行讨论，该架构在应用程序逻辑与特定的后端数据格式之间提供了一个中间层，而且与数据格式遵循开放标准还是封闭专有无关。

*第 10 章 模拟类和继承*

本章是对第二部分介绍Dijit的一个铺垫，介绍如何使用Dojo模仿面向对象编程中基于类的继承。而模拟类和继承在 Dijit 中得到了广泛应用。

第二部分系统地研究Dojo工具箱的其他组件，包括直接替换那些被编写（及重复编写）了很多次的定制化HTML控件的部件集合 —— Dijit的全面介绍。经过精心设计的Dijit可以由设计人员直接在标记中使用，而且对编程经验的要求极少，甚至为零。有了Dijit之后，只花片刻功夫就构建出令人侧目的页面是完全可能的。毕竟，所有部件的外观和行为都与桌面应用程序中的用户界面控件十分相似。

第二部分最后讨论Util中提供的构建工具和单元测试框架。其中，构建系统包含一个指向 ShinkSafe 的高度可配置的入口和一个利用 Rhino JavaScript 引擎压缩代码的工具 —— 压缩幅度可达 1/3 甚至更多。单元测试框架 DOH 是 Dojo Objective Harness（Dojo 目标套件）的缩写，这个名字的灵感来源于 Homer Simpson 那句家喻户晓的感叹词 “D’oh!”；它是对 JavaScript 代码进行单元测试的一个独立系统。

本书第二部分包含以下几章：

*第 11 章 Dijit 概述*

本章在介绍Dijit的同时，也讨论设计原理、易访问性、解析器（由Core提供的工具，但最常见的用途是解析包含部件的页面）和使用部件的模式。本章最后提供包含所有主要Dijit子项目的简介。

*第 12 章 深入理解 Dijit 及其生命周期*

本章深入探讨Dijit部件在磁盘中的分布模式，同时对其实例化后在内存中的生命周期过程给出全面介绍。本章还围绕生命周期的关键点展示一些简短的示例。理解部件的生命周期是理解后续各章的基础。

*第 13 章 表单部件*

本章首先对常规HTML表单进行简单回顾，然后展开对表单部件的全面介绍。表单部件是目前为止包含的继承关系最为丰富的一组部件。表单部件可以直接取代任何Web设计中常用的表单元素，主要包括按钮、专用的文本框和滑动条等。其他派生的元素还有下拉组合框等已经实现或重复实现了无数次的表单部件。

*第 14 章 布局部件*

本章介绍布局部件，这些部件为那些经常需要各种技巧和大量CSS的复杂布局提供了骨架，包括能根据应用程序状态隐藏/可见或者变换布局模式的选项卡式部件等。

*第 15 章 应用程序部件*

本章介绍工具箱中剩余的部件，这些部件大致能够与常见的应用程序控件对应起来，例如，提示条、模态对话框、菜单、树和富文本编辑器。

*第 16 章 构建工具、测试及程序发布*

本书最后一章讨论与程序发布有关的且经常被人忽视但却非常重要的主题。详细介绍如何使用构建工具轻松地压缩、缩减和合并JavaScript文件，从而保证文件大小和HTTP延时的最小化。另外，介绍Dojo提供的单元测试框架以及优化应用程序性能的相关技巧。

本书还包含两个附录：一个Firebug教程和一个对DojoX的评述。虽然由特殊的实验性扩展组成的DojoX绝对称得上是一个部件与模块的百宝箱，因为其中既包含图表和加密算法，又包含让人赞叹不已且高度灵活的网格部件，但是，与Base、Core和Dijit相比，无论从稳定性还是API一致性方面，DojoX的子项目都要稍差一些；要全面介绍DojoX，恐怕再多出本书现有的篇幅也不够。

另一个附录是一个简明的Firebug教程，该教程可以让读者在短时间内迅速了解Firebug的重要特性，并掌握如何通过其命令行式的界面来调试JavaScript脚本和试验新的思路。如果有读者还没听说过Firebug，那么在这里我可以告诉你，Firebug是Firefox的一个优

秀扩展，可以通过它来彻底解构一个页面的各种组成要素——从查看和操作DOM节点的样式，到监控网络通信状况，乃至使用命令行界面执行JavaScript代码。

## 本书不包括的内容

虽然本书力求为读者提供与Dojo本身的深度和广度相适应的内容，但仍然有一些主题会被排除在外：

*Web开发101*

本书虽然全面深入地介绍Dojo，但并没有为读者提供一个完整的Web开发教程，即读者在本书中看不到对HTML、JavaScript和CSS基础知识的讲解。

*多余的API文档*

本书对Dojo的绝大多数（注6）API给出了明确解释，而且为了便于查看，通常会以表格形式把这些API呈现给读者。由于Dojo的覆盖面的确太广了，因此，如果读者能充分理解这些API，那么一定会对选择使用恰当的特性有所帮助。不过，Dojo本身仍是一个发展中的项目，为保险起见，读者最好能同时参照其在线文档（*http://api.dojotoolkit.org*），以便获得最全面、权威的解释。与编程语言或者趋于刚性的应用框架不同，Dojo是一个由活跃的社区支持的发展中的项目，因此某些API有可能在新发布的版本中得到改进，从而更好地满足开发人员的需要。但是，根据官方的承诺，其1.x版本的API在2.0版之前不会有大的变动。因此，本书介绍的所有API在相当长一段时间内都将是有效的。而且，即使某些API有了变动，其在线文档也会提前给出完备的说明。

*非浏览器宿主环境*

本书也不会详细阐述或针对Dojo在浏览器之外的应用（例如，在Rhino或Adobe AIR中）给出示例。另外，也不会涉及Dojo如何与其他客户端框架（例如，DWR、YUI!或Domino）协同工作的内容。

## 开源软件容易变化

Dojo是一个由活跃的社区支持的发展中的项目，因此，随时可能增加新特性。本书完全针对当前最新的Dojo1.1版本编写。但是，将来的版本可能会增加更多功能。读者应该密切关注Dojo最新版本的发布情况，并确保认真阅读1.1版之后的发布说明。

同样，鉴于Dojo的API在2.0版之前不会有大的变化，因此本书所有示例及相关信息在

注6：　即使保守地估计，本书中包含的Base、Core、Dijit和Util中的API也超过了总数的95%。

过渡期的小版本发布过程中会始终保持正确。而且，非正式的不推荐策略表明，一个主版本中被列为不推荐使用的特性，至少应该保持到下一个主版本发布之后才会被削减。换句话说，1.x版本中的某些特性如果被列为不推荐使用的特性，那么至少到2.0版发布之后（甚至更长时间之后）仍然会保持有效。因此，即使读者在拿到本书时Dojo 2.0已经发布了，但其中的代码示例也仍然应该能够正常运行。

## 本书读者对象

本书面向至少有一定Web开发经验的读者，即读者应该熟悉HTML、JavaScript和CSS等客户端技术。不过，并不要求读者是上述某种技术的专家，而且也无须了解大量Web服务器的知识，因为Dojo是一种客户端技术。只要读者对客户端的技术有一些了解，知道它们的用途和如何使用它们就足够了。

如果你本身就是一名Web开发人员，或者是能够创建简单的网页并运用JavaScript和CSS来增强它的Web开发爱好者，那么绝对应该阅读这本书。假如你从未听说过HTML、JavaScript或者CSS，也从没写过一行代码，那么恐怕在阅读本书之前必须先找一本Web开发入门之类的书好好研读一番。

## 开发工具

提到开发工具，虽然使用任何文本编辑器和浏览器都可以基于Dojo有效地完成开发任务，但本书推荐读者使用Firefox及其优秀的Firebug扩展。因为在Firebug中，可以对网页进行调试和分析，也可以通过它的控制台执行JavaScript代码。如果读者使用其他浏览器（如IE），尽管也可以使用Firebug Lite，但完整版Firebug的功能要比它强大得多，相信它是不会让读者失望的。（一般来说，使用Firefox和Firebug做开发，然后经常性地在IE中测试是个不错的方法。）Firefox的下载地址为*http://getfirefox.com*，Firebug的下载地址为*http://getfirebug.com*。

另外两个基于Firefox的工具也可以考虑，它们分别是Chris Pederick开发的Web Developer Toolbar和Firefox的Clear Cache Button扩展。Web Developer Toolbar提供了Web开发中常用的另外一些功能，下载地址是*http://chrispederick.com/work/web-developer/*；而Clear Cache Button扩展实际上就是一个用于快速清除本地缓存的按钮，安装后可以把它拖动到工具栏中，下载地址是*https://addons.mozilla.org/en-US/firefox/addon/1801*。有时候，如果你的浏览器“不正常”了，显示的都是一些过时的内容，那么清理一下缓存很可能会解决问题。

## 必要的基础知识

闭包、环境和匿名函数是JavaScript中的几个最重要的基础概念。鉴于理解这些概念对掌握Dojo工具箱的重要作用，希望读者能够认真领会本节内容。即使这些概念听起来像是高级内容，但它们对掌握JavaScript和理解Dojo工具箱的底层实现都是最基本的。当然，也可以在用到它们的时候再来试着了解这些概念。可是，如果提前花点时间，那么对于理解相应章节中内容的帮助则会更大。

### 闭包

闭包本质上是数据与包含（或封闭）数据的作用域的结合体。包含一些函数的全局作用域通常比较容易理解，但嵌套的函数则具有一种特殊能力。为展示这种特殊能力，我们看一下例P-1。

例P-1：最小的闭包结构

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  }
}

var x = 5;
var barReference = foo();
barReference(); // 会输出什么呢？ 10还是5
```

如果读者缺乏编程经验，也许上面这段代码会令人感到迷惑。事实上，屏幕上显示的应该是10，因为在JavaScript中对函数求值时，会涉及整个作用域链。对这个具体的示例而言，对foo求值返回的作用域链与bar是相关联的。因此，在对barReference求值时，会导致动态查询x的值，进而追踪到在foo的函数体内声明的变量x。这种情况在很多编程语言中是不存在的，因为它们都是在最直接的作用域中查询变量的值。

---

**注意：** 在JavaScript中，函数是可以作为数据传递、可以赋值给变量的“一类”对象。Dojo的许多设计模式都利用了函数的这个特性。

---

虽然JavaScript闭包确实属于高级主题，但是越早掌握闭包，就能越快地掌握这门语言，从而更好地理解Dojo的很多设计原理。实际上，掌握闭包意味着可以使编程效率更高，可以更快地捕捉到不易觉察的bug，或许还能让你变得更引人注目。（当然，哪怕只达到其中两个目的也不错了。）读者可以参考David Flanagan在他传奇般的《JavaScript:The Definitive Guide》（O'Reilly）一书中对闭包的精彩分析。

## 环境

JavaScript超强的动态能力造就了其巨大的灵活性，而涉及其动态能力的一个最令人着迷、同时也是被理解得最不够的方面就是环境。读者大概知道，在基于浏览器的JavaScript中，默认的this环境是全局window对象。例如，在任何浏览器实现中对下面的语句求值，结果都应该为true：

```
// 文档的默认环境是window对象
console.log(window ==this); //true
//document是对window.document的简写
console.log(document == window.document); //true
```

对于Function对象而言，关键字this则多用于引用其直接环境。例如，读者也许看到过像下面这样使用JavaScript中Function对象的情况：

```
function Dog(sound) {
    this.sound = sound;
}

Dog.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". My name is", name);
}

dog = new Dog("woof");
dog.talk("fido"); //woof, woof. my name is fido
```

如果读者有面向对象编程的经验，应该对这种相对于当前对象向上查询sound值的方式不会陌生。这没有什么好奇怪的。但是，如果把内置的call函数再牵扯进来，问题就复杂了。下面就是一个引入call函数的示例：

```
function Dog(sound) {
    this.sound = sound;
}

Dog.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". my name is", name);
}

dog = new Dog("woof");
dog.talk("fido"); //woof, woof. my name is fido

function Cat(sound) {
    this.sound = sound;
}

Cat.prototype.talk = function(name) {
    console.log(this.sound + "," + this.sound + ". my name is", name);
}
```

```
cat = new Cat("meow");
cat.talk("felix"); //meow, meow. my name is felix
cat.talk.call(dog, "felix") //woof, woof. my name is felix
```

噢！最后一行代码的输出结果真是出人意料。这行代码通过cat对象实例调用了绑定到cat原型的talk方法，然后像以前一样传入了name参数。但是，结果却没有使用绑定到cat的this的sound值，而是使用了绑定到dog的this的sound值，因为dog被换成了执行talk方法的环境。

如果读者感觉这个示例很新鲜，可以花点时间搞清楚call函数在环境转换中所起的作用。在许多非动态的语言中，这种重新定义this的能力几乎被认为是荒唐的。然而，JavaScript引入了这个强有力的语言特性，而像Dojo这样的工具箱也都充分利用了这种固有的动态能力实现了更加令人难以置信的功能。事实上，下一节就提到了一些难以置信的功能。

---

注意：本书的意图当然不是像《JavaScript: The Definitive Guide》这本书那样为读者提供有关JavaScript的详尽资料。但是，仍然有必要在此提到与call函数对应的apply函数。apply函数的原理与call相同，它们唯一的区别就是apply能够接收任意数量的参数并将这些参数传递给目标函数。具体地说，apply接收两个参数，第二个参数是一个数组，其中可以包含任意数量的值，而这些值随后会变成目标函数内置的arguments对象的值。至于使用它们两个中的哪一个更合适，要根据实际需要来决定。

---

## 匿名函数

在JavaScript中，可以像传递其他数据类型一样传递Function对象。虽然在代码中使用匿名函数绝对能减少混乱和提高易维护性，但匿名函数的一个更重要的特性恐怕还是作为保护直接环境的闭包使用。

例如，执行下面的代码块会得到什么结果呢？

```
//变量i未定义

for (var i=0; i < 10; i++) {
    //对i进行操作
}
console.log(i); // ???
```

如果你认为console语句输出的结果是undefined，那就大错特错了。JavaScript有一个不起眼但很重要的特性，即它不支持函数外部的块级作用域。因此，无论是i的值，还是在循环过程或者条件逻辑中定义的任何“临时”变量，都会在其所在的块执行后继续存在。

如果有必要明确地提供一个块级作用域，那么可以把这个块包含在一个Function对象中，并直接在代码中调用该Function对象。例如，下面是经过修改的代码：

```
(function() {
  for (var i=0; i < 10; i++) {
    //对i进行操作
  }
})()
console.log(i); // undefined
```

尽管从语法上看略显笨拙，但这样却可以把可能的冲突因素隔离开来，从而防止讨厌的bug趁人不备，伺机作乱。接下来的各章里介绍的Base中的许多语言函数，都为被执行的代码块提供了闭包（以及语法糖和实用程序）。

# 本书排版约定

本书使用下列排版约定：

纯文本

表示菜单标题、菜单选项、菜单按钮和键盘快捷键（例如，Alt和Ctrl）。

*斜体（Italic）*

表示新术语、URL、电子邮件地址、文件名、文件扩展名、路径名、目录和UNIX实用程序。

等宽字体（Constant Width）

表示代码、命令、选项、开关（switch）、变量、属性、键、函数、类型、类、命名空间、方法、模块、特性、参数、值、对象、事件、事件处理程序、XML标签、HTML标签、宏、文件内容，或命令（代码）的输出结果。

等宽粗体（**Constant Width Bold**）

表示应该由用户直接输入的命令或文本。

等宽斜体（*Constant Width Italic*）

表示应该被用户提供的值代替的文本。

---

**注意：** 此图标表示一个提示、建议或一般备注。

---

**警告：** 此图标表明警告。

---

# 样式约定

还有两个约定有必须提前声明，因为它们涉及有效地传达内容的含义：

*限定性引用*

在上下文比较明确的情况下，一般不会使用完全限定的命名空间。例如，在代码清单中出现过dijit.form.Button部件之后，接下来的讨论可能会简单地使用Button来引用该部件。

某些术语（例如，构造函数）在相同的段落或上下文中可能会以多种方式引用。在这种情况下，具体的差异将以等宽字体来表示。例如，在句子“可以通过调用一个普通的JavaScript构造函数的方式来创建部件，但部件本身也有一个名为constructor的生命周期方法，用于对部件进行初始化”中，就使用了等宽字体来消除方法名（constructor）与术语“构造函数”的冲突。

*API列表*

一般情况下，本书会尽量提供独立的API列表，此时会统一列出标准的构造函数名称和参数类型。例如，对于需要通过代码loadUpArray("foo", 4)调用，并返回["foo", "foo", "foo", "foo"]的方法，相应的API列表如下：

```
loadUpArray(/*String*/value, /*Integer*/length) //返回数组
```

但是，由于JavaScript是一种动态的弱类型语言，因此在有些情况下，函数的参数或返回值可能是任意类型。此时，将统一使用Any来表示参数的类型。如果某个参数是可选的，那么会在其类型后面加一个问号，例如，/*Integer?*/。

如果读者查阅Dojo的源代码，可能会注意到源代码中的某些参数名与本书API中列出的参数名不尽相同。由于JavaScript函数的参数没有明确的名称和位置，而在API列表中给出实际的名称是不现实的；本书利用了JavaScript语言的这一特性，将以最直观的方式来表述API列表。尽管我们会尽最大努力以统一的方式提供API列表，但一些小的偏差恐怕在所难免。

# 使用示例代码

本书的目的就是帮助读者尽快掌握相关领域的知识。通常，读者可以在程序或文档中使用本书中的代码。如果涉及的代码量不是很多，一般不需要得到我们的许可。例如，读者在自己编写的程序中用到了本书的几段代码就不需要得到许可。但是，销售或分发包含O'Reilly图书示例代码的光盘，则必须得到许可。在解答他人的问题时以本书内容为例，或者引用本书的示例代码不需要得到许可。然而，要在产品文档中大量使用本书的示例代码，则必须得到许可。

虽然我们不会要求，但如果读者标出在引用本书代码时注明出处，我们将不胜感激。在注明出处时，通常应该包含书名、作者、出版社和ISBN。例如，“*Dojo: The Definitive Guide, by Matthew A. Russell. Copyright 2008 Matthew A. Russell, 978-0-596-51648-2.*”。

如果读者认为自己对本书代码的使用超出了合理的或上述默认许可的范围，随时可以通过 *permissions@oreilly.com* 与我们联系。

## 注释和问题

我已经测试和验证过本书中所包含的任何信息和程序代码，以保证完全符合我的初衷，但是如果你发现了错误，对本书或者代码有什么建议，麻烦与我联系。用邮政信件，可以通过 O’Reilly 与我联系：

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

你也可以给我发送电子邮件。可以加入邮件列表或者请求一个目录服务，请发送邮件到：

*info@mail.oreilly.com.cn*

如想询问技术问题，或者发表有关本书的评论，请发送邮件到：

*bookquestions@oreilly.com*

如想了解书籍的更多信息，可以到Resource Centers和O'Reilly Network，请从O’Reilly网站进入：

*http://www.oreilly.com*
*http://www.oreilly.com.cn*

正如前面所提到的，本书的网站是：

*http://www.oreilly.com/catalog/9780596516482*（原版书）
*http://www.oreilly.com.cn/book.php?bn=978-7-111-26380-7*

# 致谢

编写这样一本书，至少可以用两个词来形容，那就是挑灯夜战和浴火重生。不过，谈到从什么时候开始准备写这本书，那恐怕就要追溯到我听说Dojo、JavaScript，甚至计算机之前了。本书可以说是我在人生中经历过一系列重要事件后的必然结果——其中涉及许多重要的人物。以下基本按年代顺序给出的简单得不能再简单的人物表，列出了对读者手里拿的这本书的问世起到过这样或那样关键作用的人。

在我的人生旅途中，虽然表面上可以把这本书看成一个完全不同的事件，但实际上，对我前进的每一步产生过重大影响的人都与本书息息相关。因此，我要满怀喜悦地把溢于言表的感谢献给下面的人：

- LORD，感谢他的善良和永远坚定不移的爱！
- Lucille Tabor，她让一个孩子有了妈妈和家。
- Jerry Russell，他为一个可怜的小男孩买了第一台计算机。
- David Kade，他教会了一个有前途的学生如何用另一种语言去思考问题。
- Deborah Pennington，她几乎独立地改变了一个年轻人的命运。
- Kellan Sarles，她教给一个有抱负的作者如何用文字表达思想。
- Gary Lamont，他促使一个机灵的大脑像计算机科学家一样思考。
- Derrick Story，他给了一个想成为作者的人一次机会。
- Abe Music，他第一次把Dojo介绍给我。
- Simon St.Laurent、Tatiana Apandi、Colleen Gorman、Rachel Monaghan和Sumita Mukherji，他们指导我如何编辑手稿，并为我消除书中的问题提供了很多建议。
- 在#dojo中结识的新朋友们，他们慷慨地抽出时间审阅了本书的手稿，并提出了宝贵的反馈意见。他们是（排名不分先后）：Adam Peller、Sam Foster、Karl Tiedt、Bill Keese、Dustin Machi、Pete Higgins、James Burke、Peter Kristoffersson、Alex Russell和Dylan Schiemann。
- Baseeret，她是我一生的挚爱。

Matthew A. Russell

2008年6月

# 第一部分

# Base与Core

本书的这一部分将介绍Base与Core，Dojo工具箱的这两个组成部分包含着强大的JavaScript标准库。Base是工具箱的内核，它囊括的功能数量之大，令人瞠目，而且经过优化后的文件通过网络传输时其大小不超过30KB。Base中包含的每一个特性都以实用性强、执行速度快和代码量少见长。一旦使用Base，你就会发现自己的生活中不能没有它，其实在页面中导入Base很简单：只需写一个SCRIPT标签即可，甚至还可以从AOL的地理边际缓存（edge-cached）服务器中跨域加载这个文件。除了为整个工具箱提供逻辑基础之外，Base中的一切都被包含在dojo基准级别（base-level）的命名空间内，因此访问最常用的方法和数据成员永远只需少量的输入。

Core在Base的基础上补充了很多功能，这些功能我们很快就要介绍到。但是，为了确保Base尽可能简洁，Core被分别封装在了不同的包中，毕竟Core中的特性在开发过程中并非都那么常用。同样，从Core中导入资源也很简单：只需简单地调用`dojo.require`方法，这种方法类似于C语言中的`#include`或者Java中的`import`；导入资源之后，就可以正常地使用它们了。正如第16章中讨论Util时将要介绍的，开发人员实际上可以使用Dojo构建系统来将所需的非Base资源组合到一个脚本中，因而，在生产开发中使用Core与使用Base相比没有更多要求。Core中包含的一些特性涉及动画方法（`dojo.fx`）、拖放工具（`dojo.dnd`）、数据管理层（`dojo.data`）、cookie处理（`dojo.cookie`）等。

要想成为一名高效的Dojo开发人员，熟悉Base和Core提供的各种工具是绝对必需的。无论这些工具因何而存在，也不管读者是否使用过它们，理解这些工具和技术都将是一种有益的提升。在掌握了Base和Core之后，读者不仅能以更少的努力实现那些经常占用开发人员宝贵时间的常见任务，同时还可以把更多的时间花在自己项目中更有价值的地方，而这些地方往往需要更多的新意和不同凡响的思维。

# 第1章

# Dojo工具箱概述

本章将概述Dojo工具箱的架构，讲解如何安装Dojo，并且介绍一些专门的术语。另外，在介绍如何加载引导程序（bootstrapping）的基础上还将展示一些示例代码，以激发读者对后续章节的浓厚兴趣。既然是概述，本章大部分内容都将提纲契领，旨在为本书其余内容奠定基调。通过阅读本章，读者将对Dojo工具箱有全面的了解。

## Dojo的架构

稍后读者就会明白，把Dojo称作工具箱并非心血来潮。Dojo除了提供各式各样的JavaScript标准库方法之外，还包含了功能丰富的一应俱全的部件（使用这些部件几乎不需要编写JavaScript代码）、构建工具、测试框架等。从全局角度来看，Dojo的总体架构大致如图1-1所示。随着学习的深入，读者将会发现本书其余章节的内容基本上都是围绕这个架构展开的。尽管从图中我们看到了DojoX独立于Dijit，但正如自定义部件可以利用Dijit和DojoX一样，DojoX也可能构建于Dijit之上。

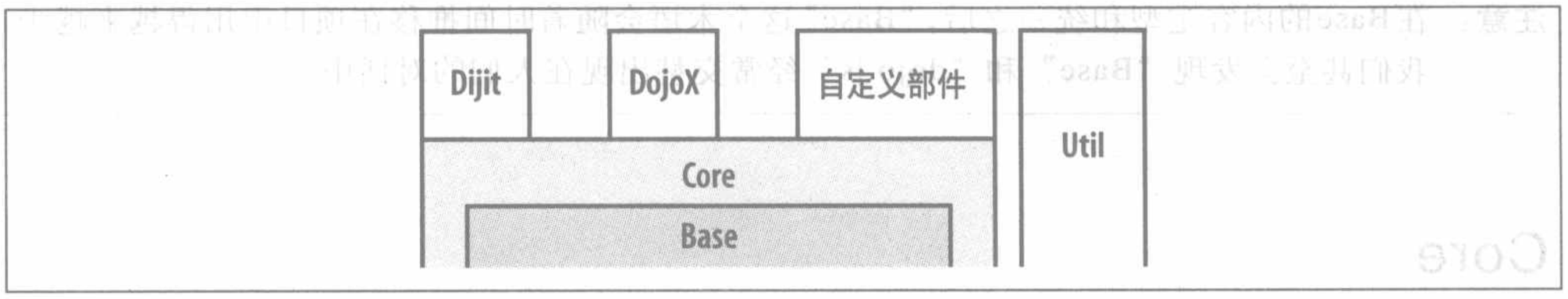



图1-1：Dojo组件之间的关系

### Base

Dojo的内核（Base），是一个极其简洁、高度优化的库，也是工具箱中其他部分的基础。

此外，Base提供了便捷的语言和Ajax实用程序，提供了开发人员可以用来动态加载Dojo资源（而不是在页面加载时一次性导入全部资源）的打包系统。Base还提供了可以用来创建和操作具有继承性层次结构的工具，一种几乎无所不能的使用CSS3选择符来查询DOM的手段和一种跨浏览器标准化DOM事件的构造（fabric）。Base提供的一切都可以在工具箱的顶级命名空间中以`dojo.*`方法或属性的方式访问。打包后的Base保存在一个名为*dojo.js*的文件中，这个文件通过网络传输时只有不到30KB。想像一下，如今Web中泛滥的大多数Flash广告都远远超过了30KB，而Base如此瘦削的身躯真有点令人难以置信。

**注意：** 通过查看磁盘中的*dojo.js*文件，我们会发现它的实际大小约为80KB。不过，由于Web服务器在“跨网络”将文件传输给浏览器时会对文件内容进行压缩，因此压缩后的文件大小决定了下载所需的时间。读者可以亲手试一试对*dojo.js*应用gzip压缩，而压缩后的文件会减少为原来大小的1/3。

Base的另一个非常有趣的方面体现在它被设计为引导程序，即通过将*dojo.js*文件简单地包含进页面内，可以自动导入Dojo的基本组件。大致来说，加载引导程序的过程涉及检测环境、消除浏览器不兼容性和加载`dojo`命名空间。此外，通过指定各种配置选项也可以让Dojo自动解析页面中的部件或者执行其他初始化任务。（所有这些内容都将在后续章节中讨论。）

Base提供了大量实用程序，可以实现通过JavaScript完成的许多标准操作。相信只要你体验到Base所带来的工作效率的巨大提升，那么即便不使用工具箱中的其他特性，它都会成为你不可或缺的宝贵资源。可以说，没有Base就没有Dojo，这个工具箱中的一切都以某种形式依赖于Base，或者基于Base构建。

**注意：** 在Base的内容定型和统一之后，“Base”这个术语会随着时间推移在项目中用得越来越少，我们甚至会发现“Base”和“dojo.js”经常交替出现在人们的对话中。

## Core

Core基于Base构建，提供了解析部件、高级动画效果、拖放工具、国际化（i18n）、后退按钮处理、管理cookie等方面的功能。虽然Core中包含的资源也比较常用，并且也能够对常见操作提供基础性支持，但仍然没有通用到必须包含在Base中的程度。尽管将某些资源包含或不包含在Core中没有绝对的标准，但Dojo的打包系统的确提供了如同C中的`#include`或者Java中的`import`语句一样简单的机制，极大地方便了导入必要模块和资源的操作。

大体而言，要区分Base与Core很容易：任何必须显式导入到页面中的模块或资源，如果它与dojo命名空间有关系，那么它就属于*dojo.js*外部的Core。Core的功能通常不会存在于Base级别的命名空间中，而是位于dojo.fx或dojo.data这样较低级别的命名空间中。

## Dijit

如果只把Dojo看成一个包含各种JavaScript标准库方法的工具箱，那么就说明你对Dojo的理解还非常不全面，因为Dojo还包含一个名为*Dijit*（“Dojo widget”发音的简称）的奇妙而丰富的部件库。Dijit提供了大量即装即用的部件，而且使用这些部件通常不需要编写任何JavaScript代码。Dijit库中的部件遵循普遍认可的易访问性标准（如ARIA（注1））设计，其预配置的国际化选项也能够适用于多数地区。Dijit直接构建于Core（因而强烈依赖于Core的完整性），如果开发人员要使用自己设计的部件，那么就会用到与使用Dijit部件时用到的完全相同的构建块。而且，通过Dojo创建的部件具有极佳的易移植性，可以方便地共享或部署到任何Web服务器中。甚至，在没有Web服务器的情况下通过*file://*protocol也能在本地运行这些部件。

向页面中插入一个dijit部件很简单，简单到只需在普通的HTML标签中指定一个特殊的dojoType属性——布局设计人员和不希望编写过多（甚至不想编写）JavaScript程序的用户终于可以梦想成真了。其实，应用程序开发人员使用Dijit的一个主要好处就是无须关注冗长乏味的实现细节，即可实现极为丰富的功能。对于库开发人员或自定义部件的开发人员，只要遵循Dijit的风格和约定，那么开发出的部件将同样易于移植并且基本符合可重用软件组件的要求。

Dijit中的部件可以粗略地分类为通用部件、布局部件和表单部件。通用部件的例子有进度条和模态对话框，布局部件的例子有选项卡容器和折叠窗格，而表单部件则提供了按钮和各种输入元素等原有表单元素的超级增强版。

## DojoX

DojoX是一组子项目的集合，这些子项目的正式含义是“Dojo扩展”，不过它们也经常被称为“扩展和实验品”（Extensions and Experimental）。DojoX中的“扩展”子项目指的是那些稳定的部件和资源，虽然它们都很有价值，但却不适合归入Core或Dijit中；“实验品”子项目指的则是那些极有可能发生变化的部件，而且更多的是处于构思阶段的部件。

---

注1：正在制定中的Accessible Rich Internet Application标准的地址为：*http://www.w3.org/WAI/intro/aria*。

每个DojoX子项目都必须提供一个*README*文件，用以简要说明该子项目的状态。尽管DojoX子项目会尽力满足易访问性和国际化的要求，以便从一开始就与Dijit保持一致，但这并不表示它们一定会完全符合要求。不过，无论如何DojoX中已经包含了很多解决现实应用问题的重量级程序，既有网格部件，也有针对常用Web服务的数据转换器。DojoX在为创新思维提供沙箱环境和孵化器的同时，又确保了它们不会影响针对Core和Dijit中那些资源的高标准、稳定的API。从某种意义上说，DojoX为所有社区支持的OSS（Open Source Software，开放源代码软件）项目的核心问题找到了一个恰到好处的平衡点。

## Util

Util是一个Dojo实用程序的集合，其中包括一个JavaScript单元测试框架和一些用于创建自定义的Dojo版本(针对产品开发)的构建工具。测试框架DOH(注2)(Dojo Objective Harness，Dojo目标套件）与Dojo没有特殊的耦合关系，它所提供的一组简单的程序可用于自动保证任何JavaScript代码的质量。毕竟，没有人不想为自己的JavaScript代码实现定义明确的、系统级的测试，不是吗？

构建工具的作用是减少代码文件的大小，并将它们整合到一个由多层构成的文件集合中，集合中的每一层将只包含一组JavaScript文件。其中，压缩由ShrinkSafe完成。ShrinkSafe是Mozilla强大的Rhino JavaScript引擎的修补（patch）版，该引擎在压缩JavaScript代码时不会破坏公共API。而整合由一系列自定义的脚本完成，这些脚本也是通过Rhino运行的。Util中其他辅助组件的功能还包括在JavaScript文件中嵌入HTML模板字符串（第11章介绍Dijit时将更详细地讨论相关内容）——这也是减少延时的另一种技巧。

**注意：** 通过阅读本节，虽然读者能够理解构建工具的作用，但对为什么要使用构建工具可能还不是很清楚。简而言之，构建工具通过合并和缩减JavaScript代码能够显著减少HTTP延时，这在产品开发中将是一项重大的性能提升。

和DOH一样，ShrinkSafe也可以脱离Dojo使用。但从产品开发角度考虑，如果能将JavaScript文件的大小普遍减少50%甚至更多，那么几乎没有任何理由不使用它。毕竟，通过一连串同步请求加载多个大型JavaScript文件与只加载一两个压缩后的JavaScript文件的性能是不可同日而语的。

注2： 熟悉动画片The Simpsons的读者可能知道，DOH是Homer Simpson的一句经典感叹词；在测试出错时，测试程序有可能会播放“D'oh!”音效。

# 开发前的准备

要学习如何使用Dojo开发，既不需要奇特的工具，也不需要对Web服务器（如Apache）进行繁琐的配置。事实上，本书中仅有少数示例会用到Web服务器。多数示例所需的资源都可以通过本地机器中的相对路径或者跨域加载获得。因此，在学习本书的过程中，很大程度上只需要你、你喜爱的文本编辑器和浏览器即可。

要作好开发准备，首先必须下载Dojo。下载Dojo的方式主要有3种：一是下载正式版，二是从Subversion中获取包含最新最好特性的版本，三是使用保存在AOL的CDN（Content Delivery Network，内容分发网络）中的跨域（XDomain）构建版。下面我们会逐一介绍这3种方式。尽管下载正式版可能是一种最典型的做法，但其他两种方式也各自有其独特的价值。

## 获取 Dojo

如前所述，使用Dojo主要有3种方式：把正式版下载到本地环境中、从Subversion中下载一份最新副本和使用AOL的CDN中的跨域构建版。本节将详细介绍这3种方式。

### 下载正式版

目前，准备开发的最传统方式就是下载Dojo最新正式版。其实，所谓“正式”版，就是在Subversion版本库（repository）中经过验证并包含帮助信息的快照（snapshot）基础上加了一个标签而已。访问*http://dojotoolkit.org/downloads*可以下载Dojo工具箱的正式版，但要注意的是，下载的正式版中不包含构建工具。如果想获取构建工具，可以使用Subversion，或者访问*http://download.dojotoolkit.org/*并下载一份源文件版。

解压缩下载的文件后，会生成一个命名形式为*dojo-release-x.y.z*的文件夹。其中，“x”、“y”和“z”分别对应某个特定版本的主版本号、小版本号和补丁编号。为了保持设置和URL的简洁，有必要重新命名该文件夹，如将其修改为*js*（即JavaScript的简写）。另外，也可以通过服务器命令把*dojo-release-x.y.z*化名为*js*，或者在Linux和UNIX环境下使用符号链接（symbolic link）来简化对该文件夹的访问。这样，在需要引用Dojo时，就可以使用简单的相对路径*www/js*，而不是*www/dojo-release-x.y.z*了。

---

**注意：** 创建符号链接很简单。在Linux、Mac OS X或UNIX平台中，只需在终端中执行下列形式的命令即可：`ln -s dojo-release-x.y.z js`。通过执行`man ls`命令打开man页面，可以了解更多有关符号链接的信息。

---

下载完Dojo并解压缩之后，读者可能会因为看到其中不止包含一个文件而感到奇怪。但是，只要稍加留意就会发现解压缩后生成的子文件夹恰好与我们前面几节介绍的Base（*dojo/dojo.js*）、Core（*dojo*）、Dijit（*dijit*）、DojoX（*dojox*）和Util（*util*）一一对应。虽然本书将全面、系统地讲解这些组件，但目前来看，为把Base引入页面中所要做的唯一一件事，就是提供一个指向*dojo.js*文件的相对路径（即像引入其他JavaScript文件一样，在页面的SCRIPT标签中通过相对路径*dojo/dojo.js*来引入Base）。应该说，这的确是再简单不过了。

## 从 Subversion 下载

也许读者在开发自己的项目时使用最新的Dojo正式版就可以了。但是，假如你有兴趣维护Subversion版本库的一份副本，从而做到与最前沿的进展保持同步，那么请认真阅读本节内容。本节将介绍通过 Subversion 检验 Dojo 和设置适当开发环境的有关步骤。基于Subversion主干（trunk）开发的好处有很多，例如，可以及早获悉某个bug是否被修复，可以率先体验尚未公之于众的新特性，以及满足那些不掌握第一手内幕消息就寝食难安的超一流程序员的贪心。

---

**注意：** 要了解有关 Subversion 的权威信息，请参考 *http://svnbook.red-bean.com/* 中的 Version Control with Subversion（使用 Subversion 进行版本控制）。

---

Dojo 的 Subversion 版本库位于 *http://svn.dojotoolkit.org/src/*，如果读者只想通过 Web 浏览器查看某些代码，那么这里就是不错的起点。不过，检验代码的更便捷方式则是通过外部视图（external view）一次性下载整个工具箱，外部视图的链接地址为 *http://svn.dojotoolkit.org/src/view/anon/all/trunk*。

---

**注意：** Subversion的svn:externals属性为构建通常需要多个独立环节才能完成的工作副本提供了便利的解决方案。更多相关内容请参考 *http://svnbook.red-bean.com/en/1.0/ch07s03.html*（译注 1）。

---

为了检验代码，可以在终端执行如下命令（本节剩余内容将假设读者已经在一个名为*www*的文件夹中执行了该检验）：

```
svn co http://svn.dojotoolkit.org/src/view/anon/all/trunk ./svn
```

---

译注 1： Subversion1.4 版的链接地址为：*http://svnbook.red-bean.com/en/1.4/svn.advanced.externals.html*，中文版链接地址为：*http://www.subversion.org.cn/svnbook/1.4/svn.advanced.externals.html*。

就如同下载的正式版一样，对 Subversion 检验完成后将生成一个 *svn* 文件夹，其中包含着与工具箱一一对应的子文件夹（*dojo*、*dijit*、*dojox* 和 *util*）。不过，此次 *util* 文件夹中将包含构建脚本（可能还会包含用于支持工具箱的其他辅助性工具）。有关 Subversion 的更详细信息，我们将在后面进行介绍，不过，目前应该知道同时拥有多个 Dojo 版本（如一个最新正式版、一个每夜构建和一个版本库的实际检验）并非难事。而且，可以随时使用服务器命令或其他手段按照需要在这些版本间进行切换。

### AOL 的 CDN

AOL 在其内容分发网络（AOL CDN）中提供了 Dojo 的跨域版。要使用这个 Dojo 的跨域构建，开发人员只需设置几个配置参数，并在页面中包含一个 SCRIPT 标签，让其指向 AOL 的 CDN 服务器上的相应文件即可。由于使用跨域版的确非常简单，本书所有示例都将使用该跨域构建，以便读者在动手试验这些示例时不会感到麻烦。

正如前两节所提到的，要在页面中加载 Dojo 通常需要指向下载到的 *dojo.js* 文件，即像下面这样指定一个相对路径：

```
<script
    type="text/javascript"
    src="www/js/dojo/dojo.js">
</script>
```

使用 AOL 的跨域构建同样也这么简单：修改 src 属性，其他事情交给 Dojo（和 AOL）来完成。下面的 SCRIPT 标签示范了跨域加载 Dojo1.1 的方法：

```
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
</script>
```

注意，上面代码中加粗了 *dojo.xd.js*。这是因为如果在此指定了 *dojo.js*，那么结果很可能是看到一个错误消息，而不是加载到我们想要的 Dojo 文件。另外，要注意的是 src 路径中的 *1.1* 引用的是该版本的最新 bug 版。通过在此指定 1.1.0 或 1.1.1 也可以请求一个具体的 bug 修复版。也许读者有必要把 *http://dev.aol.com/dojo* 加入到自己的书签中，因为这上面包含有在 CDN 中可以访问哪些 Dojo 版本的权威信息。

## 使用 Firebug 调试

即使读者仅完成过少量的 Web 开发，恐怕对调试之难也已经有所体会了，特别是在涉及浏览器间隐秘的差异或浏览器实现与 W3C 标准背离的情况下。而且，当面对一个兼具强大功能与高度复杂性的工具箱时，调试的难度会进一步加大。这种情况在 JavaScript 开发领域中表现得尤其突出，因为 JavaScript 程序的调试不仅牵扯到微妙的闭包和动态

类型，而且通过警告框显示调试信息也极为不便。所幸的是，我们还有*Firebug*，这个令人兴奋的Firefox扩展让调试和Web开发都变得轻松了许多。

根据经验，在Firefox中进行Web开发是最理想的，其主要原因就是Firebug在加快开发进度方面的作用非常之大。在Firebug中，不仅可以实时地查看/操作DOM中的一切（包括样式），还能在其控制台中记录事件，而且通过捕获的具体错误信息往往一眼就能看出真正的问题所在。（有了这些功能作为对比，你还愿意再使用警告框吗？）

当然，为了确保程序能够跨平台运行，还必须在IE和其他浏览器中进行测试。尽管Dojo在解决浏览器不兼容性问题上着实下了一番功夫，但也不能排除意外情况的发生。当然，如果能更快速地发现问题，就更好了。

---

**注意：** 假如读者使用过Firebug，那么就能体会到这个令人兴奋的工具有多么方便。通过在Firefox中开发并使用Firebug调试，肯定能够节省大量宝贵的时间。但是，也不要忘了只有频繁地（如每半小时一次）在IE中测试才能发现那些潜藏在应用程序中不容易发现的问题。举个例子，假如读者在JavaScript关联数组中最后一个键/值对结尾加了一个逗号，Firefox可能会放过你，但IE却不行……尽管通过IE获得的错误消息基本上没什么价值。

到目前为止，Firebug中最常用的功能恐怕就要数`console.log`了，在JavaScript代码中使用该方法可以向Firebug控制台中写入信息。（毕竟，我们都已经厌烦警告框了，不是吗？）

---

由于Dojo致力于紧密整合Firebug，因此它打包提供了Firebug Lite。这就意味着，即使读者必须在其他浏览器中进行开发，也可以使用`console.log`方法。

Firefox和Firebug的下载地址分别为*http://www.getfirefox.com*和*http://www.getfirebug.com*。本书附录A中还提供了一份Firebug入门教程，可能会对读者的开发有所帮助。

## 在本地运行Dojo的浏览器安全设置

本书中的多数示例都会涉及使用本地的*file://*协议来加载数据，而这一般而言都没有什么问题。但是，对于Firefox3用户来说，如果想在本地加载Dojo可能需要对浏览器做如下小小的调整。

1. 在浏览器地址栏中输入`about:config`，然后按回车键。
2. 将`security.fileuri.origin_policy`的值修改为3或更大的数字。

有关此问题的更多信息，请参考*http://kb.mozillazine.org/Security.fileuri.origin_policy*。

## 轻量级服务器响应

本书要介绍的所有内容几乎都不需要通过Web服务器即可说明。但为了透彻说明工具箱中的某些特性，有时也需要服务器提供一些动态内容才行。在有这种需要的情况下，我们会使用以Python编写的一个非常容易使用的Web服务器 —— CherryPy（版本3.1+）。可以访问 *http://cherrypy.org* 下载 CherryPy 并了解相关信息。不过，请读者稍安勿躁，你不会因为要掌握一种新 Web 服务器的使用而影响到对 Dojo 学习的连续性。

安装CherryPy非常简单，简单到只要下载它并按照 *README* 文件中简短的说明去做即可。安装 CherryPy 与安装其他 Python 模块一样，因此并不会存在安装目录。而且，CherryPy 与其他复杂的服务器端技术不同，通过像包含其他 Python 模块一样使用 `import` 语句包含它，就可以马上使用CherryPy。事实上，一个CherryPy应用程序本质上就是一个运行在它自己的多线程Web服务器上的Python应用程序。从这个意义上说，执行“服务器端脚本”也就相当于在终端中运行一条命令。

---

**注意：** 涉及需要提供动态内容或者明确取得服务器响应的所有示例(非常少)，都只不过是在终端执行的一条命令而已，因此读者不必惊慌 —— 这些服务器端示例只能吓住胆小的人！当然，我们举双手赞成你彻底搞明白每个涉及服务器技术的示例；而且，对这些示例我们都会给出详尽的解释。

---

例如，要是我们说请读者执行下面这个保存在名为 *hello.py* 文件中的简单的应用程序，那么你要做的就是在命令行中输入 `python hello.py`。这样，CherryPy 就可以启动并监听来自 8080 端口的请求。在安装了 CherryPy 之后，就可以像例 1-1 中所示的那样通过 `import cherrypy` 语句加载并使用它了。

例 1-1：简单的 CherryPy 应用程序

```
import cherrypy

class Content:

    @cherrypy.expose
    def index(self):
        return "Hello"

    @cherrypy.expose
    def greet(self, name=None):
        return "Hello "+name

cherrypy.quickstart(Content())
```

对于例 1-1 而言，如果读者在浏览器中打开 *http://localhost:8080/*，将会访问 `index` 方

法（第6～7行）并得到字符串“Hello”作为响应。如果打开*http://localhost:8080/greet?name=Dojo*，那么会访问greet方法（第10～11行），该方法会处理name查询字符串参数并给出字符串响应“Hello Dojo”。这就是一般化的使用模式（也体现了Python代码本身的易读性），的确是简单地没法再简单了。

虽然读者不需要编写和理解本书中出现的Python代码，但假如读者某一天需要或想要创建自己的动态内容了，那么前面这段解释就可以说明使用CherryPy和Python有多么方便。如果真有这么一天，读者想要系统地学习Python语言，我们推荐Mark Lutz的《Learning Python》（O'Reilly）一书。另外，Python的官方网站*http://www.python.org*和CherryPy的网站*http://www.cherrypy.org*中也提供了大量的相关文档。

## 重要的术语

下面我们花点时间来明确一下全书讨论中可能会涉及的一些重要术语。如果要对JavaScript的原理给出精确解释，必须用精确的术语来定义专属于这门语言的重要概念。否则，在基于JavaScript构建高效工具箱的过程中，这些术语之间的界限很可能会变得模糊不清，更不必说这个工具箱所要模拟的类在严格的面向对象编程的语境下根本不存在了。

希望读者在阅读并理解了下列术语之后，能在将来我们提及其中某些概念时不致于再迷惑不解。

*工具箱*

工具箱就是一套工具的集合。在计算机编程领域，工具箱是用户界面设计过程中的一个常用术语。从某种意义上说，把Dojo称为工具箱是最合适的。因为它除了在一组相关方法和抽象基础之上提供的支持代码库之外，还提供了部署实用程序、测试工具和打包系统。虽然从本质上来看，把它称为库、框架或工具箱都未尝不可，但毕竟Dojo已经使用了工具箱这个称呼，因此我们还是尊重现实吧。

*模块*

从文件系统的角度上看，Dojo模块无非就是一个JavaScript文件，或者包含一组关系密切的JavaScript文件的目录。因此，顶级目录也就意味着它所包含代码的一个命名空间。从逻辑关系的角度上看，Dojo中的模块与其他编程语言中包的概念类似，即都用来划分相关的软件组件。但是，要格外注意，虽然Dojo的打包系统事实上也在执行确定和获取依赖关系的任务，但Dojo中的模块却不叫做“包”。

*资源*

在必须将Dojo模块分割成多个文件，或者一个模块仅由一个JavaScript文件构成的情况下，一个JavaScript文件就称为一个资源。虽然从理论上讲，资源应该可以

用来表示与模块相关的各种抽象概念，但在Dojo中它也具有最小化一个JavaScript文件大小的含义。通过为资源赋予最小化文件大小的含义，也就意味着用户不会下载用不到的冗余代码，同时也不会下载过多的小文件——所有对文件的请求都是同步请求，因而会占用与Web服务器通信的资源（尽管使用构建工具创建层能够让这种开销微不足道）。

*命名空间*

从文件系统的角度上看，Dojo的命名空间所映射的恰好是指定模块和资源的文件系统层次；从逻辑角度来看，命名空间的概念可以防止相同名称的模块和资源之间发生冲突。注意，虽然命名空间既不是模块，也不是资源，但它所代表的语义思想却可以直接映射到模块和资源。另外，值得一提的是，Dojo会保护页面的全局命名空间，如果实现适当，那么通过Dojo创建的任何模块都不会影响全局命名空间。前面曾经提到过，Base中的一切都位于顶级的dojo命名空间内。

*第一类对象*

在计算机程序设计中，所谓第一类对象（first-class）就是可以不受限制地随意传递，并因此有别于同一门语言中其他实体的事物。例如，在许多编程语言中都不能像传递数值或字符串值等数据类型一样传递函数。那么，在这些编程语言中，函数就不能算作第一类对象。在本书的讨论中，将使用这个术语来强调函数在JavaScript属于第一类对象这个事实。正如我们在本书后面章节中即将介绍的，类似把函数直接赋值给变量和/或把函数置于关联数组中这样的操作是很多Dojo设计模式的基础。

*函数*

函数，是只需定义一次但可以执行多次的代码段。在JavaScript中，函数是和其他任何变量一样可以随意传递的一类对象。其中，构造函数专指通过new运算符使用的函数，结果是创建一个新的Java Script Function对象并对该对象执行初始化。注意，所有JavaScript对象都继承自JavaScript内置的`Object`类型，并且都有一个`prototype`属性。JavaScript正是以这些`prototype`属性为基础实现了基于原型链的强大继承机制。在Dojo的语境下，构造函数这个术语同样也可以指`dojo.declare`的关联数组参数中映射到`constructor`键的匿名函数。这个匿名函数的主要作用就是初始化Dojo类的属性。

*对象*

这个JavaScript中最普通的概念所指的是一种复合数据类型，其中可以包含任意数量的命名属性。例如，这条简单的语句`var o = {}`使用对象直接量语法创建了一个JavaScript对象。由于术语“对象”在本书中使用的频率极高，因此有时候也把`{a : 1,b : 2}`这样的键-值对结构称为“关联数组”，而不称其为对象。从

技术角度上来讲，JavaScript中只有对象没有类，即使Dojo通过dojo.declare方法（一个专为模拟类而设计的一个特殊方法）模拟了一种类表示法。

*属性*

在OOP领域，存储在类中的任何数据片段通常都称为属性。在Dojo的语境下，这个术语可以用来指包含在Function对象中的数据，也可以用来指包含在Dojo类（由dojo.declare定义）中的数据。

*方法*

作为一个类成员的函数，在宽泛的OOP语境下，在JavaScript和Dojo中通常都称为方法。而且，具体到Dojo中，出现在dojo.delcare语句中的匿名函数也称为方法，因为dojo.declare提供了一种基于类的继承机制。一般来说，读者可以将方法想像为在类中定义的因而只能通过对象环境来调用的函数。

*类*

在Dojo中，通过dojo.declare方法（一个专门用来模拟类和继承层次的特殊方法）定义的用于表示一个逻辑实体的声明称为类。同样地，这个术语的使用并不是很严谨，毕竟JavaScript不支持Java和C++等语言中的类。

*部件*

Dojo部件（widget）就是由dojo.declare语句创建的Function对象，这些对象都以dijit._Widget作为祖先（即所有部件的基类）。通常，部件在屏幕上都具有某种可见的外观，而逻辑上则是由HTML、CSS、JavaScript及其他静态资源组合而成的统一实体。因此，对部件可以像对文件一样进行操作、维护和移植。

# 启用Dojo

**注意：** 读者可以在开始时略过本节内容，等到对Dojo有了较为全面的了解之后，再回过头来阅读。

在使用Dojo之前，首先必须在页面中启用它。无论是在本地安装Dojo，还是通过AOL的CDN加载它，我们所要做的就是提供一个SCRIPT标签，令其指向包含某些JavaScript代码的文件，然后浏览器中的魔法精灵就会奇迹般地让一切都“运转起来”，对吗？对，但不够全面。与计算机领域中那些完全自动化的情形类似，Dojo的启用过程也没有什么不同。

**注意：** 术语“启用”（boot strap）的含义是指“通过你自己的启用程序来构建自己的运行环境”。换句话说，就是要在没有外界帮助的前提下设置好自己的运行环境。Dojo中启用程序的概念与我们按下计算机开关“启动”计算机的概念完全相同。

可以肯定地说，例1-2中包含着在HTML页面中跨域加载Dojo功能的绝对最小代码量。有必要指出的是，从CDN中加载Dojo时，通过网络传输的数据量只有不到30KB。读者可以直接使用前面的代码块，也可以在使用之前对它进行一些修改。而把这些代码复制到一个模板中以备复用，则可以节省一些输入的时间。

例1-2：最小化的Dojo应用程序示例

```
<html>
  <head>
    <title>Title Goes Here</title>
    <!-- 用于消除浏览器间视觉差异的一个简单的样式表 -->
   <link rel="stylesheet"  type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />

    <script
      type="text/javascript"
      src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        /* 在必要的情况下，可以使用dojo.require语句以异步方式
       请求并将相应的Dojo模块加载到页面中…… */

        dojo.addOnLoad(function() {

            /* 依赖于dojo.require语句的内容放在这里…… */

        });
    </script>
  </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

**注意：** dojo.addOnLoad接受一个函数作为参数。虽然可以像var init = function() { /*...*/}这样先定义一个函数，然后再将init传递给dojo.addOnLoad，但本书中的示例通常都在此使用匿名函数。匿名函数及其重要性在本书前言部分介绍过。

前面代码示例中包含两个新结构，即dojo.require语句和dojo.addOnLoad块。有关

dojo.require语句的详细讨论将在第2章“通过模块管理源代码”中进行。不过，简单地说，dojo.require的作用就像在Java中使用import或在C编程中使用#include一样，负责把指定的资源加载到页面中以供开发人员使用。另外，dojo.require的一个极其重要的特点就是它对工具箱的本地安装执行同步加载，而在跨域加载工具箱时则执行异步操作。这一差别之所以如此重要，是因为它涉及dojo.addOnLoad。

## dojo.addOnLoad

前面我们提到过，dojo.require语句在跨域的情况下执行异步加载。由于延时和其他因素可能（通常都会）导致网络延迟，因此在页面加载后（注3）不一定能够使用通过dojo.require请求的资源。如果尝试引用通过dojo.require请求但尚未加载的某个模块，就会引发错误，并且很可能会导致整个启用过程意外中断。从技术上讲，这种由于相互竞争的请求之间的时间选择不可预知而导致的结果不明确被称为竞态条件。

基于上述原因，我们的建议是请读者养成使用dojo.addOnLoad的习惯，因为无论是否跨域加载，dojo.addOnLoad都会保证页面尽可能容易移植。

---

**警告：** 对于只有在页面加载完成并且dojo.require语句的请求也返回了结果的情况下才能可靠执行的逻辑，不使用dojo.addOnLoad是一个常见的错误。这种错误经常出现在本地开发之后，又通过SCRIPT标签切换为跨域加载的情况下。

---

在前面使用跨域加载的代码片段中，没有任何地方涉及本地安装的Dojo文件，因此可以说它是在已有页面中启用整个工具箱的真正一步到位的过程。

---

**注意：** 虽然本章的主题不是部件，但毕竟我们谈到了addOnLoad，因此有必要提醒读者注意一下，addOnLoad在页面中的部件没有被解析完成之前是不会被执行的（前提是已经通过配置告诉Dojo在页面加载后解析部件）。

---

虽然从表面上来看，启用Dojo只不过就是“加载一个脚本”的同义语，但在这个加载过程中其实还隐藏着很多表面上看不到的处理。例如，下面就是启用Dojo过程中必须要执行的两项基本操作，尽管实际的顺序不一定如此。

---

注3： 一般来讲，页面加载事件指的可能是Window的onload事件，也可能是Mozilla系列浏览器的DOMContentLoaded事件完成。

*平台配置*

根据开发人员在djConfig中指定的自定义配置选项做出相应处理。djConfig可以是一个关联数组，该数组必须在加载Dojo的SCRIPT标签之前定义；或者，也可以作为加载Dojo的SCRIPT标签的一个属性存在。更多有关djConfig的内容将在本章后面介绍。

确定是跨域加载还是本地加载Dojo。只要Internet连接正常并且在构建时配置了跨域加载程序，跨域加载就能够顺利完成。在默认情况下，针对跨域加载的配置会产生一个*dojo.xd.js*文件（以及其他一系列**.xd.js*文件），该文件用来代替标准的*dojo.js*文件。

根据为特定的Dojo构建指定的环境（通常是浏览器，但也存在其他可能，如Rhino或移动设备等），设置针对该环境的各种特性。在使用针对浏览器的默认Dojo构建时，即使开发人员不需要执行针对浏览器的配置，Base仍然会提供dojo.isIE和dojo.isFF之类的数据成员，以供必要时检测底层浏览器使用。

执行针对浏览器的功能增强操作。例如，在使用Dojo不同的Ajax实用程序执行异步调用时，创建一个XMLHttpRequest（XHR）对象；通过标准化DOM事件、标准化键码映射，以及采取其他减少或防止内存泄漏的手段来修补浏览器的不兼容性行为。

*命名空间的建立和加载*

建立dojo命名空间，以确保工具箱提供的各种实用程序不会与页面中已经存在的标识符发生冲突。

通过组成Base的各种函数和标识符加载dojo命名空间。

---

**注意：** 尽管与dojo.addOnLoad相比，dojo.addOnUnload没有那么常用，但如果某些逻辑需要在页面卸载时执行，那么dojo.addOnUnload仍然是最佳选择。

---

## 配置djConfig

---

**注意：** 当读者编写过一段时间代码之后，一定会发现本节的很多内容其实很有价值。不过，眼下不必在此花费过多的心思，因为这些内容只是作为参考用的。

---

本节介绍djConfig的配置。djConfig是一个配置开关，可以把它作为属性放在启用工具箱的SCRIPT标签中（或在启用Dojo之前定义它），用于定制到哪里去查找资源、是否需要启用调试工具等。

表1-1列出了可以传递给djConfig以配置启用过程的键/值对。(表中有些说明可能会提及尚未介绍的内容。对这些内容，可以先略过不读，等以后需要的时候再回来阅读。)

**警告：** 在加载工具箱的SCRIPT标签之后定义djConfig无效。

表1-1：djConfig配置开关

| 键 | 值类型（默认值） | 说明 |
|---|---|---|
| afterOnLoad | 布尔值（false） | 用于设置在页面加载后将Dojo注入其中。在需要hack某个页面或开发需要延迟加载的（lazy-loaded）部件（如社交网络应用中的部件）时很有用 |
| baseUrl | 字符串（undefined） | 从技术上说，设置这个键可以重新定义本地或跨域加载工具箱时的顶级路径。一般用于解析依赖关系，例如，确定自定义模块的物理位置。但在实践中，设置该键基本上只是为了在跨域启用Dojo时解析本地模块 |
| cacheBust | 字符串 \| 日期（undefined） | 如果为该键设置的是字符串值，那么将该值追加到对模块的请求中，从而重写缓存在本地的上一个版本的页面。通常，这个值可能是由开发人员自己生成的一个随机字符串，也可能是表示所开发应用程序版本的唯一标识符。总之，该值用于防止旧版本模块导致的棘手bug再次出现<br>在开发过程中，可以为这个键设置一个日期值，例如，(new Date()).getTime()。这样，就可以确保每次页面加载时都会产生一个新值，从而避免讨厌的缓存问题 |
| debugAtAllCosts | 布尔值（false） | 设置该键表示要求提供与性能有关的具体调试信息。在知道错误是由*bootstrap.js*、*dojo.js*或*dojo.xd.js*等构建文件引起的情况下，设置这个键有助于追踪到错误所在的行号 |

表1-1：djConfig配置开关（续）

| 键 | 值类型（默认值） | 说明 |
| --- | --- | --- |
| dojoBlankHtmlUrl | 字符串（""） | 用于指定空白HTML文档的地址。在使用dojo.io.iframe.create通过IFRAME传输数据时，必须指定一个空白HTML文档（参见第4章）。默认的文档URL为*dojo/resources/blank.html* |
| dojoIframeHistoryUrl | 字符串（""） | 用于指定一个特殊文件的地址。在处理后退按钮时，需要指定一个与dojo.back组合使用的特殊文件（参见第2章）。默认的文档URL为*dojo/resources/iframe_history.html* |
| enableMozDomContentLoaded | 布尔值（false） | 设置该键可以让基于Gecko的浏览器（如Firefox）以DOMContentLoaded事件作为页面加载完毕的依据。之所以引入DOMContentLoaded事件，是因为当XHR请求涉及的文档大小超过65 536字节时会存在细微的技术差别[注] |
| extraLocale | 字符串或数组（""） | 用于额外指定地区信息，以便Dojo明确处理与提供的本地化模块相关的细节。这个键的值可以是一个字符串值，也可以是一个字符串值的数组 |
| isDebug | 布尔值（false） | 设置该键用于加载Firebug或Firebug Lite以便进行调试。注意，用于调试功能的占位程序（如各种console方法）会默认提供，因此从应用程序中移除诊断信息不会影响代码运行 |
| libraryScriptUri | 字符串（""） | 设置该键可以像在浏览器环境中使用baseUrl一样配置Rhino和SpiderMon-key（均为JavaScript引擎）等非浏览器环境 |
| locale | 字符串（浏览器提供） | 设置该键可以重写浏览器提供给dojo.locale的地区值 |

表 1-1：djConfig 配置开关（续）

| 键 | 值类型（默认值） | 说明 |
| --- | --- | --- |
| modulePaths | 对象（undefined） | 这个键的值是一个键/值对集合，其中每个键/值对分别关联着一个模块及该模块在磁盘中的相对路径。虽然工具箱的根目录下是存放模块的通常位置，但通过这个键可以加载位于其他地方的模块。如果是从 CDN 或其他 XDomain 中加载 Dojo，那么还必须同时指定 baseUrl 参数 |
| parseOnLoad | 布尔值（false） | 用于指定是否在页面加载后自动解析其中的部件（本质是在启用过程中的适当时候执行dojo.parser.parse()函数） |
| require | 数组（[]） | 这个键为在 Base 加载后自动请求模块提供了一种便捷方式。在与 afterOn-Load 共同使用时，该键用于指定当页面加载完毕后向其中注入 Dojo 时应该加载的资源 |
| usePlainJson | 布尔值（true） | 用于指定在没有使用注释式 JSON（comment filtered JSON）时是否通过 console 给出警告。之所以将这个键的默认值设置为 true，是因为一般来讲注释式 JSON 要比非注释式 JSON 更安全 |
| useXDomain | 布尔值（false） | 用于强制执行跨域加载。对于跨域构建会默认采取这种方式 |
| xdWaitSeconds | 数字（15） | 用于指定在跨域加载资源时请求超时的秒数 |

注：参见 *http://trac.dojotoolkit.org/ticket/1704*

**警告：** 虽然 djConfig 中的参数是 modulePaths，但 Base 中用于设置个别模块路径的方法则是 dojo.registerModulePath —— 没有末尾的字母“s”。

在多数情况下，只需在启用工具箱的SCRIPT标签中使用类似Object的语法来定义djConfig。请看下面的示例：

```
<script
    type="text/javascript"
    src=" http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js "
    djConfig="parseOnLoad:true,isDebug:true">
</script>
```

不过，也可以在加载工具箱的SCRIPT标签之前定义djConfig，这种情况通常是因为要设置的配置项很多，或者在某些情形下这样做更方便一些。下面就是以这种方式设置djConfig的示例，它与上一个示例的效果完全相同：

```
<script type="text/javascript">
    djConfig = {
        parseOnLoad : true,
        isDebug : true
    };
</script>

<script
  type="text/javascript"
  src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js ">
</script>
```

# 在 Firebug 中探索 Dojo

虽然本书其余部分将深入介绍整个工具箱，但目前还是有必要先抽点时间通过Firebug控制台来对Dojo进行一番探索。毕竟，在开发程序期间，有时可能需要在隔离的环境下试验一些想法，而Firebug控制台往往就能在这些时候充当一个解释器。

## 注入 Dojo

读者也许想过使用Dojo来修改某个现有站点中的页面，为此，可以借助Firebug或书签的动态脚本插入功能来达到这一目的。Dojo从1.1版开始加入了afterOnLoad和require配置开关，从而确保了在页面加载后能够正常地调用回调序列。

把下面这些代码复制到Firebug控制台中并执行，就可以在现有页面中插入Dojo。唯一的区别就是必须将djConfig定义为一个独立的对象，而不是作为SCRIPT标签的属性。只有这样才能在页面加载完毕后安全地注入Dojo，因为不同的浏览器在处理动态SCRIPT标签时存在一些细微的差别：

—待续—

```
/* 在定义djConfig时将afterOnLoad设置为true。如果需要，也可以添加对更多模
块的请求。在此，我们假设只需要加载dojo.behavior模块。*/
djConfig={afterOnLoad:true,require:['dojo.behavior']}

/* 创建与页面头部中出现的相同的script标签 */
var e = document.createElement("script");
e.type="text/javascript";
e.src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js";

/* 插入该标签。注入Dojo成功 */
document.getElementsByTagName("head")[0].appendChild(e);
```

更简单的方法则是创建一个书签[注]，然后通过按快捷键或单击该书签来加载Dojo。而创建一个书签无非就是将前面的代码块放到一个函数内并调用这个函数。读者可以在自己的浏览器中创建一个名为“Dojo-ify”的书签，并以下面的代码（全部位于一行）作为其地址（译注2）：

```
(function() {djConfig={afterOnLoad:true,require:['dojo.behavior']};
var e = document.createElement("script"); e.type="text/javascript";
e.src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js";
document.getElementsByTagName("head")[0].appendChild(e);})()
```

在执行书签之后，以上脚本会把Dojo加载到页面中，读者就可以调用Dojo的CDN构建版中的方法了。此时可以随心所欲地对页面进行修改。但要注意的是，到Dojo1.1为止，还不存在通知工具箱（以及请求的模块）已经加载完成的addOnLoad对等结构。不过，Dojo1.2中会增加一个addOnLoad对等结构，详情请参阅：*http://www.oreillynet.com/onlamp/blog/2008/05/dojo_goodness_part_7_injecting.html*。

与在页面加载后注入Dojo相关的另一种可能的用途是，在社区网络或公众信息应用程序的概况中延迟加载部件。

注：书签中可以保存JavaScript代码片段，其设计目的是增强页面的功能。

## 探索Base

为了完成探索，读者可以将例1-3中的HTML页面转录到一个本地文件中。其中唯一一处针对Dojo的地方就是执行跨域加载的SCRIPT标签。该标签中包含的djConfig属性是为了在启用Dojo的过程中传递配置信息。在这个例子中，我们明确指定了要使用一个类似Firebug控制台的调试工具。不过，即使读者使用IE浏览器也没关系，因为djConfig="isDebug:true"会保证在IE中加载Firebug Lite。

---

译注2：如果使用书签时浏览器提示URL无效，请读者重新定义书签并在代码前面添加javascript:。

例1-3：用于示范Base中部分特性的简单HTML页面

```
<html>
    <head>
        <title>Fun with Dojo!</title>

          <link rel="stylesheet"  type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css"/>

        <script
             type="text/javascript"
             src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true">
        </script>

      <style type="text/css">
            .blue {color: blue;}
        </style>
</head>
 <body>
    <div id="d1" class="blue">A div with id=d1 and class=blue</div>
    <div id="d2">A div with id=d2</div>
    <div id="d2">Another div with id=d2</div>
    <div id="d4">A div with id=d3.
        <span id="s1">
            This sentence is in a span that's contained in d1.
            The span's id is s1.
        </span>
    </div>
     <form name="foo" action="">
         A form with name="foo"
     </form>
      <div id="foo">
         A div with id=foo
      </div>
    </body>
</html>
```

保存这个文件之后，请在Firefox中打开该页面，然后单击右下角带有小对勾的绿圆圈图标（译注3）以展开Firebug控制台。然后单击紧挨着Firebug搜索框的插入符号（^）图标，从而在新窗口中打开Firebug，结果如图1-2所示。（如果读者还没有看过本书附录A中的Firebug入门教程，现在可能是个不错的时机。）单击“网络”（Net）标签（tab）即可看到Base所在的跨域*dojo.xd.js*文件已经从CDN下载完毕。

现在单击“控制台”（Console）标签再返回上一个界面，在提示符>>>中输入`dojo`并按回车，应该看到Firebug控制台中显示出一条消息：`Object _scopeName=dojo _scopeMap=Object`。这说明确实存在全局JavaScript对象`dojo`。然后输入

译注3：在Firebug1.2中，是彩色的臭虫图标。

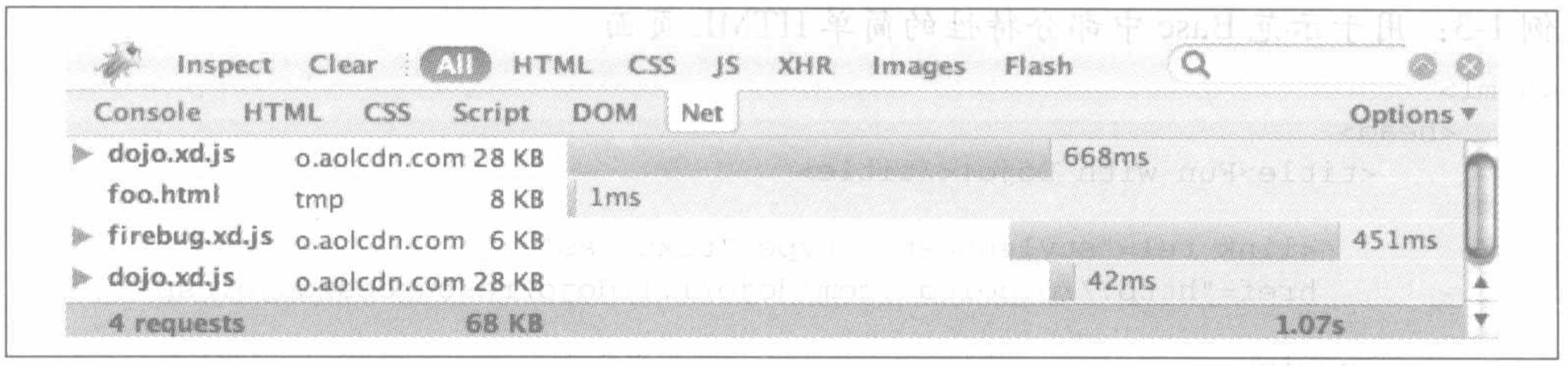



图 1-2：通过 Firebug 可以全面了解网络请求及其他有价值的信息

console.dir(dojo)并按回车，则会看到控制台中以树形结构显示出了Dojo所包含的一切。虽然其中也包含了很多以下划线开头的私有成员，但其他成员也都能够一览无余。大致地看一看输出的内容，就能初步体验到Base中到底都封装了什么。

### dojo.byId

Dojo提供了一个代替document.getElementById的方法dojo.byId。也就是说，如果为该方法传递一个字符串值，如dojo.byId("s1")，那么该方法则会返回一个可以保存在变量中的引用，就和调用document.getElementById一样。不过，除了能够查找id值之外，如果为dojo.byId传递一个DOM节点，该方法也会执行空操作（译注4）。在这个方法内部，会对传入的参数进行检查，而在应用程序层次上，开发人员就不必为此再费心思了。这个方法完整的签名形式如下：

```
dojo.byId(/*String*/ id | /*DomNode*/ node)  // 返回一个DOM节点
```

**注意：** 在本书中，竖线（|）用于在方法签名中存在多个可能项时表示逻辑“或”运算。

读者可能会认为dojo.byId与document.getElementById的用处几乎完全一样，因而会觉得它没有什么价值，事实并非如此！因为dojo.byId还能够帮开发人员消除一些浏览器实现间的微妙差别。例如，IE6和IE7中就存在一个由使用document.getElementById引发的广为人知的bug。为说明这个问题，请读者在示例文档的Firebug Lite控制台中（译注5）输入以下代码，并且见图1-3：

```
document.getElementById("foo") // 结果是否“十分”显而易见呢?!?
```

译注4：即返回传入的DOM节点。

译注5：即在IE浏览器打开的页面中。

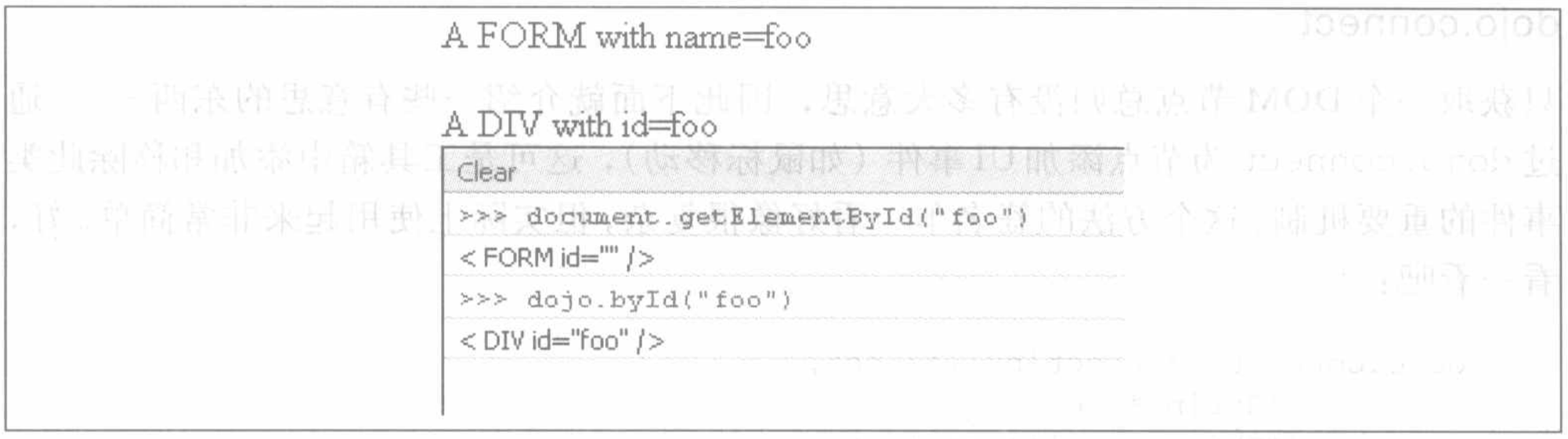



图 1-3：在示例文档中执行 document.getElementById 和 dojo.byId 的结果对比

你大概没有想到它会返回一个 FORM 元素吧，是吗？事实表明，只有表单不先在文档中出现的情况下，我们才能取得想要的 div 元素。导致这个 bug 的原因是 IE 把 name 和 id 属性放在了同一个命名空间中。类似这种明显的跨浏览器兼容性的问题远不止于此啊！图 1-4 展示了 Dojo 在隔离冷酷的浏览器界面时充当的保护者角色，从图中可以看出正是因为 Dojo 帮助我们解决了众多的不一致性问题，才使得应用程序更易于移植。


图 1-4：Dojo 通过在浏览器不一致的行为之上设置隔离层确保代码更易于移植

除了隔离不一致的行为之外，dojo.byId 也会在多个元素具有相同 id 值的情况下只返回第一个元素，因而正常化了两种极端的行为。对于前面的示例文档，要验证 dojo.byId 只返回第一个元素，可以试一试下面这行代码：

```
dojo.byId("d2").innerHTML
```

通过这个简短的小教程，读者最应该领会的一个核心问题是：假如使用一个 JavaScript 工具箱做开发，那么就要利用它的 API 来实现各种任务，而不要使用纯 JavaScript 的方式。有时候，你会觉得某个 API 并不具有额外的利用价值，于是就不由自主地拿自己可以实现同样任务的方法取而代之，千万要抵制住这种诱惑！因为装备精良的 API 绝不会提供没用的功能。

---

**警告：** 当你觉得某个 API “没有价值”时，可能是因为对它的实际价值缺乏了解。每当此时，可以再翻看该 API 的文档，也许会发现自己没有注意到的细节。假如文档也不能让你信服，建议通过邮件列表或 IRC 向别人寻求解释。

---

### dojo.connect

只获取一个DOM节点总归没有多大意思，因此下面就介绍一些有意思的东西——通过dojo.connect为节点添加UI事件（如鼠标移动），这可是工具箱中添加和移除此类事件的重要机制。这个方法的签名乍一看好像很复杂，但实际上使用起来非常简单。好，看一看吧：

```
dojo.connect(/*Object|null*/ obj,
        /*String*/ event,
        /*Object|null*/ context,
        /*String|Function*/ method) //返回一个连接句柄（connection handle）
```

要试用一下dojo.connect，可以在Firebug控制台中执行下面的代码，然后再将鼠标移动到SPAN元素包含的句子内容上方，看一看是否正确设置了mouseover事件。（可能需要单击右下角的插入符号图标将命令行提示符扩展为多行模式。）

```
var handle = dojo.connect(
  dojo.byId("s1"), //环境
  "onmouseover", //事件
  null, //环境
  function(evt) {console.log("mouseover event", evt);} //事件
);
```

读者应该注意到，除了在Firebug控制台能够看到事件发生的确认信息外，还可以看到实际事件的参考信息——通常是与鼠标事件在屏幕上发生的实际位置有关的信息以及其他信息。

结果表明，dojo.connect与dojo.byId相似，能够为我们提供的信息远不止我们开始想像的那么少。实际上，传递给dojo.connect的任意值为null的参数都可以省略不写。因此，前面的方法调用经过精简之后可能会更容易看明白：

```
var handle = dojo.connect(
  dojo.byId("s1"), //环境
  "onmouseover", //事件
  function(evt) {console.log("mouseover event",evt);} //事件
);
```

此外，切断执行的函数与相关DOM事件之间的联系也很容易；而且，在大量使用连接和分离的情况下，切断联系也非常重要。只需基于保存的句柄调用dojo.disconnect，Dojo就会做好剩下的一切：

```
dojo.disconnect(handle);
```

虽然这个例子比较简单，但dojo.connect却为我们揭示了Dojo背后的哲学理念：让A与B之间建立联系变得更加自然、简单。当然，假如读者对JavaScript十分精通，那

么也可以自己来实现建立、维护和切断联系的任务。但是，这样做肯定要付出因打乱设计而产生的成本。而且，不要忘记：你必须维护自己编写的每一行代码。对于胸怀大志的Web开发人员以及像我们一样愿意保持事件简单的人，调用dojo.connect和dojo.disconnect应该是一个不错的选择。

JavaScript尚未做到的事，Dojo同样也做不到，从这一点上来说，其他JavaScript工具箱也都一样。但是，Dojo所能带来的巨大价值主要体现在消除多个浏览器之间的不一致行为，以及确保常见的操作自然而简单。同时，它还能对开发人员编写和维护代码起到保驾护航的作用，以尽可能提高开发效率。

另一个让人爱不释手的美妙特性，是小巧玲珑的dojo.query方法。该方法是Dojo通过CSS3风格的语法来实现快速查询的一种机制。

---

**注意：** 第5章详细介绍了dojo.query并提供了有关CSS3选择符的背景知识。读者可以现在就翻阅一下该章的内容。

---

例如，要查找页面中所有的DIV元素，可以这样做：

```
dojo.query("div") // 查找DOM中所有的div元素
```

如果读者通过Firebug控制台执行这条语句，一定会看到返回的DIV元素列表。而查询页面中是否存在一个特定的DIV元素，同样也像我们想像得那么简单：

```
dojo.query("div#d2") // 检查id="d2"的div元素是否存在
```

下面这行代码是通过类来查询元素：

```
dojo.query(".blue") // 返回带有blue类的元素列表
```

提到类，我们也可以在此加入对元素类型的筛选。但是，由于只有一个DIV元素带有class属性，因此还需要再给另外一个元素添加blue类。不过，在动手编辑页面标记之前，为什么不考虑使用Base中的另一个内置函数dojo.addClass来添加这个类呢：

```
dojo.addClass("s1", "blue");  // 为SPAN元素添加blue类
```

在为s1添加了blue类之后，下面这个dojo.query调用就示范了另一种查询方式：

```
dojo.query("span.blue") // 只返回一个带有blue类的span元素
```

发现规律了吧？当然，对于上述操作我们也可以采取非直接的方式来实现。但是，难道了解一下工具箱的隔离作用让我们免遭折磨，同时还提供了单一易用的函数不是件好事吗？

## 探索 Dijit

虽然可以继续展示Base中包含的易用API，但我们还是把这些内容留给下一章吧。接下来，我们再花点笔墨描述一个简单的示例，通过该示例来体验一下无须编程即可在页面中加入 Dijit 部件有多么容易。

假设有例 1-4 所示的页面。

例 1-4：非常基本的表单示例

```
<html>
    <head>
        <title>Fun with Dijit!</title>
    </head>
        <body>
        Just Use the form below to sign-up for our great offers:<br /><br />
        <form id="registration_form">
            First Name: <input type="text" maxlength=25 name="first"/><br />
            Last Name: <input type="text" maxlength=25 name="last"/><br />
            Your Email: <input type="text" maxlength=25 name="email"/><br /><br />
            <button onclick="alert('Boo!')">Sign Up!</button>
        </form>
    </body>
</html>
```

虽然这个页面的外观不难想像，但通过图 1-5 展示出来会更清楚一些。

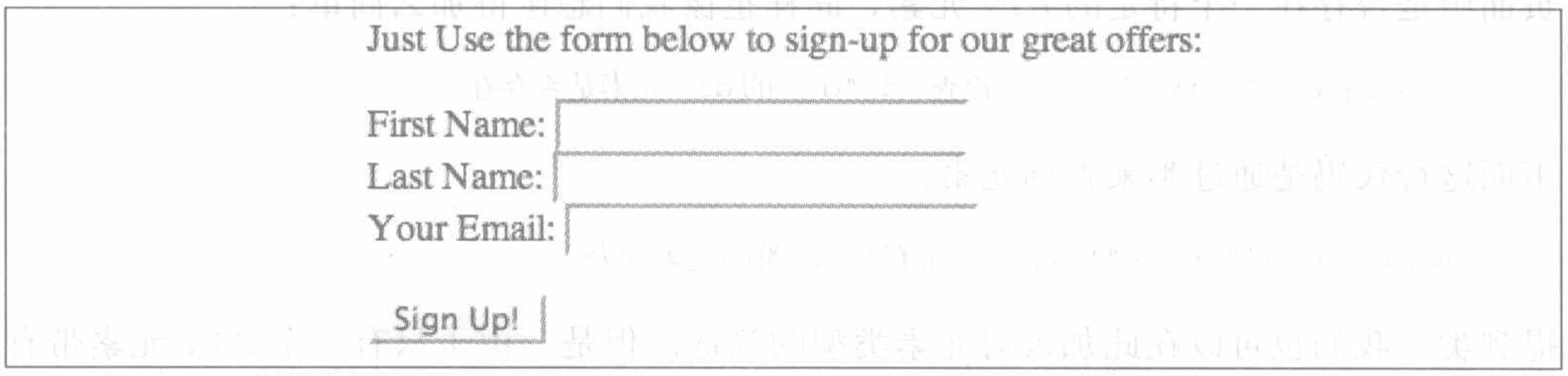



图 1-5：可以使用但却没有美感的表单

端详一下这张表单，时光好像一下子倒退回到了1992年。然而，现在它这个样子我们无论如何是不能接受的。那么，如果此时此刻让你来美化它，你会怎么做：定义几个类，应用这些类，再编写一些实现验证功能的 JavaScript 代码……

为了向读者简单展示一下这个页面在经过Dojo的美化之后会变成什么样子，请看一下例 1-5。不过，眼下还不要过分纠缠于其中的细枝末节；类似的网页见得多了自然就会对各种细节了如指掌了。因此，当前我们的目标就是对在页面中加入Dijit部件后的整体页面结构有一个大致印象。

例1-5：不再那么基本的表单（归功于Dojo的美化作用）

```
<html>
    <head>
        <title>Fun with Dijit!</title>

        <!-- 取得Dojo为美化页面内置提供的tundra 主题的样式表，
        这样，无须更多工作即可获得专业化的样式 -->
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <!-- 添加一些自定义CSS样式，以便更好地对齐表单元素 -->
<style type="text/css">
    h3 {
        margin : 10px;
    }
    label,input {
        display: block;
        float: left;
        margin-bottom: 5px;
     }
     label {
        text-align: right;
        width: 70px;
        padding-right: 20px;
     }
     br {
        clear: left;
     }
     .grouping {
        width:300px;
        border:solid 1px rgb(230,230,230);
        padding:5px;
        margin:10px;
      }
</style>

<!-- 加载Base并指定应该在页面加载后解析部件 -->
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
    djConfig="parseOnLoad: true" >
</script>
<!-- 通过dojo.require以类似C编程中#include或者Java中import
     的方式加载所需的部件 -->
<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.form.TextBox");
    dojo.require("dijit.form.ValidationTextBox");
    dojo.require("dijit.form.Button");
```

```
    </script>
</head>

    <!-- 通过为body标签添加class="tundra"属性，指定为页面中所有元素
      应用默认的tundra主题。(Dijit十分依赖CSS，因此在使用部件时指定主题
      是非常重要的。) -->
    <body class="tundra">

      <h3>Sign-up for our great offers:</h3>

      <form id="registration_form">

        <!-- 通过定义一些标签并分别为它们添加相应的dojoType属性来指定
          所要使用的部件，以便解析器找到它们并将部件插入到页面中 -->
        <div class="grouping">
          <label>First Name:</label>
          <input type="text"
           maxlength=25
           name="first"
           dojoType="dijit.form.TextBox"
           trim="true"
           propercase="true"/><br>

          <label>Last Name:</label>
          <input type="text"
           maxlength=25
           name="last"
           dojoType="dijit.form.TextBox"
           trim="true"
           propercase="true"/><br>

          <label>Your Email:</label>
          <input type="text"
           maxlength=25
           name="email"
           dojoType="dijit.form.ValidationTextBox"
           trim="true"
           lowercase="true"
           regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"
           required="true"
           invalidMessage="Please enter a valid e-mail address"/><br>

          <button dojoType="dijit.form.Button"
           onClick="alert('Boo!')">Sign Up!</button>
        </div>

      </form>
    </body>
</html>
```

嗯，就这样吧，图1-6展示了这个新页面的外观，其中还包含简单的验证功能。


图 1-6：通过即装即用的 Dijit 部件实现的漂亮表单

假如读者通过试验本章的示例产生了进一步学习Dojo的兴趣，那么就请继续往下看吧。本书接下来的章节会系统地介绍这个工具箱的方方面面。不过，在继续之前，还是有必要先简单回忆一下本章讨论过的内容（正如后续每章结尾都要做的一样）。

## 小结

在本章中，我们仅仅接触到了Dojo工具箱的皮毛，不过，也确实学习到了它的很多基础知识。在学习完本章后，读者应该：

- 知道到哪里下载这个工具箱以及如何设置开发环境。
- 理解这个工具箱的架构和每个组件间的关键差别。
- 理解 Dojo 开发中一些常见的特殊用法。
- 认识到在开发过程中使用 Firebug 的好处。
- 理解工具箱启用过程背后的基本思想。
- 对使用 Base 和向页面中添加 Dijit 部件的简单方便有正确的认识。
- 熟悉和适应 Dojo 代码的风格。
- 强烈希望阅读后续章节并愿意学习有关 Dojo 的更多知识。

下一章，我们将讨论语言及浏览器实用程序。

第2章

# 语言及浏览器实用程序

本章，我们正式介绍Base中包含的语言实用程序。这些语言实用程序的目标，就是遵循超级便携和高度优化的设计原则，简化JavaScript编程中最常见的任务处理。本章讨论的内容是使用工具箱中其他组件的必备基础，主要包括操作数组、复制节点、添加和移除类，以及计算DOM节点的外边距和内容盒子等。

## 查找DOM节点

上一章介绍的dojo.byId是Dojo工具箱特有的一种查找DOM节点的机制，该机制相对于document.getElementById而言更方便也更可靠。尽管dojo.byId的使用在Dojo开发中随处可见，但还是有必要在此重申一下第1章讨论过的它的一个重要价值。那就是dojo.byId解决了使用document.getElementById可能存在的问题。下面，我们重温一下dojo.byId的完整API：

```
dojo.byId(/*String*/ id | /*DomNode*/ node, /*DomNode*/doc)   //返回一个DOM节点
```

例2-1列出了其他常见的使用模式。

例2-1：dojo.byId的用法参考

```
var foo = dojo.byId("foo");          //如果存在，返回id="foo"的节点
dojo.byId(foo).innerHTML="bar";      //因为foo是节点，所以查找过程为空操作；
                                     //然后将inneHTML设置为“bar”
var bar = dojo.byId("bar", baz);     //如果baz引用的文档存在，
                                     //那么返回该文档中id="bar"的节点
```

# 类型检查

对于像JavaScript这样的动态类型语言，在操作变量之前先检查变量的类型经常是有必要的（也是为了稳妥起见）。表面看来，检查变量类型不过小事一桩；但在实际编程过程中，一些微妙的差异则经常会成为烦恼和bug的来源。Base为处理这些细微差别提供了一些辅助方法。就和我们到目前为止接触到的问题一样，不同浏览器实现间的微妙差异正是问题的根源所在。下面是Base提供的类型检查方法：

`isString(/*Any*/ value)`

如果`value`是`String`类型，返回`true`。

`isArray(/*Any*/ value)`

如果`value`是`Array`类型，返回`true`。

`isFunction(/*Any*/ value)`

如果`value`是`Function`类型，返回`true`。

`isObject(/*Any*/ value)`

如果`value`是`Object`（包括`Array`和`Function`）类型或`null`，返回`true`。

`isArrayLike(/*Any*/ value)`

如果value是`Array`类型，但又允许存在一些例外情况则返回`true`。例如，在Function对象中可以访问的内置`arguments`对象就有些反常，因为它不支持`push`这样的内置方法；不过，`arguments`与数组也很类似，因为它是一个可以通过索引访问的列表。

`isAlien(/*Any*/ value)`

如果value是一个内置函数或ActiveX组件之类的本地函数，但又不能通过像`instanceof Function`这样的常规检查来确认，返回`true`。

# 鸭子类型

鸭子类型是与Python和JavaScript这样的动态编程语言相关的一个概念。鸭子类型为我们刚刚介绍的许多函数提供了线索。有一句古话：“如果它走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子。”鸭子类型也因此而得名。在JavaScript中，这就意味着如果一个数据成员具有作为一种特定数据类型必需的最低限度的属性，那么就完全可以假设该数据成员就是该数据类型。

鸭子类型的一个典型例子，就是内置的`arguments`成员。因为通过`isArrayLike`方法对它进行检查，可以判定它具备成为数组类型的资格。在一门不要求开发人员事先声明

变量并且必须保持该变量为一种特定数据类型（动态绑定）的动态语言中，鸭子类型就成为必要时检查对象类型的重要手段。

例如，对[]这个普通的数组调用typeof运算符会返回object，而Base的isArray方法通过在后台执行某些鸭子类型检查，可以保证在测试[]这个数组时返回true。

**注意：** 鸭子类型在JavaScript和Dojo工具箱中是一个非常基本的编程概念，因此本节的内容虽然不起眼，但相信对我们日常的编程实践会具有很大的指导价值。

总之，Base提供的类型检查方法不仅可以节省开发时间，更有助于我们摆脱浏览器不一致实现的困扰，因此要尽量多用、用好这些方法。

# 字符串工具

提到字符串，相信大多数人都会想到清理字符串空格这项极为常见的任务。不过，下一次再遇到这个问题时就不必再自己编写代码了，可以直接使用Base的trim方法。

**注意：** 即使最不起眼的实用方法也可能导致微小的性能损失，但Dojo作为一个社区集体智慧的结晶，已经对这些问题给予了足够的关注。因此，读者可以尽享这个工具箱的便利了。

下面是使用trim方法的一个示例：

```
var s = "  this is a value with whitespace padding each side    ";
s = dojo.trim(s); //"this is a value with whitespace padding each side"
```

Core的string模块包含了另外一些实用的字符串方法。下面这几个示例假设读者已经通过dojo.require语句加载了dojo.string模块。

dojo.string.pad

插入字符串值并填充指定数量的字符。在默认情况下，每次都在左侧插入字符。提供可选参数会导致从右侧插入字符。

```
dojo.string.pad("", 5); // "00000"
dojo.string.pad("", 5, " "); // "     "
dojo.string.pad("0", 5, "1"); // "11110"
dojo.string.pad("0", 5, "1", true); // "01111"
```

dojo.string.substitute

对字符串进行参数式置换。可选的参数用于指定转换函数和/或另一个对象作为环境。

```
//返回"Jack and Jill went up a hill."
dojo.string.substitute("${0} and ${1} went up a hill.", ["Jack", "Jill"]);
//"Jack and Jill went up a hill."
dojo.string.substitute("${person1} and ${person2} went up a hill.", {person1 :
"Jack", person2: "Jill"});
//"*Jack* and *Jill* went up a hill."
dojo.string.substitute("${0} and ${1} went up a hill.", ["Jack", "Jill"],
function(x) {
  return "*"+x+"*";
});
```

dojo.string.trim

比Base的实现占用的资源更少一些。Core的string模块提供了效率更高的trim方法，可以用在十分关注性能的情况下。

```
dojo.string.trim( /* your string value */);
```

# 数组处理

数组是所有命令式编程语言中最基本的一种数据结构，JavaScript当然也不例外。然而，令人遗憾的是所有浏览器都不支持标准化的数组操作；在这种情况下，就需要一个工具箱来把我们这些“热锅上的蚂蚁”拯救出来。而且，即使主流浏览器的下一个版本都会以完全统一的方式支持数组操作，相信为数众多的使用旧版本浏览器的用户，仍然是必须考虑的受众群体。

---

**注意：** 许多语言工具都会针对性能采取优化措施，而且也会尽可能提供对本地Array实现的包装方法，但也会在一些浏览器（如IE）缺少某些功能时模仿这些功能。

---

值得高兴的是，Dojo始终坚持与Mozilla实现的丰富的Array对象特性（*http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference*）保持同步。因此，可以说只要使用这个工具箱，那么一切尽在掌握之中。要是读者对我们在第1章讨论dojo.byId时的谆谆告诫不以为然，认为那只是理所当然而已，那么下一节的内容没准会激发你的热情，说不定还会令你大吃一惊。

## 查找元素位置

有两个常见的数组操作涉及到查找元素索引，而这同确定元素存在与否事实上是一回事。Base为此提供了两种不言自明的方法，dojo.indexOf和dojo.lastIndexOf。这两个方法在元素存在的情况下都会返回表示该元素索引的整数值，而如果元素不存在则返

回 -1。而且，它们的方法签名也都包含一个可选的参数，用于表示从数组开头或末尾算起的起点位置。下面是这两个方法的签名：

```
dojo.indexOf(/*Array*/ array, /*Any*/ value, /*Integer?*/ fromIndex)
//返回整数值

dojo.lastIndexOf(/*Array*/ array, /*Any*/ value, /*Integer?*/ fromIndex)
//返回整数值
```

**警告：** 如果读者曾面向 Firefox 做过一段开发，恐怕会惊讶于 IE6 和 IE7 本地的 Array 对象居然连 indexOf 方法都不支持。毕竟，越是这样明显而且看似不可发生的语义错误，越有可能导致最难追踪的 bug。

以下是这两个方法的用法示例：

```
var foo = [1,2,3];
var bar = [4,5,6,5,6];
var baz = [1,2,3];

dojo.indexOf([foo, bar], baz); // -1
dojo.indexOf(foo, 3); // 2
dojo.indexOf(bar, 6, 2); // 2
dojo.indexOf(bar, 6, 3); // 4

dojo.lastIndexOf(bar, 6); // 4
```

关于这两个方法，还有必要再说明一点，即它们执行浅比较（shallow comparison）。所谓浅比较，对于像数组这样比较复杂的数据类型而言，意味着它们只比较引用。下面的代码示例可以帮我们理解什么是浅比较：

```
bop = [4,5,6,5,6, foo]; // bop包含一个嵌套的数组
dojo.indexOf(bop, foo); //5，因为bop中包含一个foo（的引用）
dojo.indexOf(bop, [1,2,3]); //-1，因为foo与[1,2,3]不是同一个对象
```

## 根据条件测试元素

在实际开发中，我们经常会遇到想确定是否数组中的每个元素都符合某个特定条件，或者只有其中部分元素符合该条件。为此，Base 提供了 every 和 some 方法。这两个方法的参数包括：一个数组、一个用于测试数组中每个元素的函数（每个元素都要传递给这个函数），以及一个可选的参数，用于提供测试函数的环境（this）：

```
dojo.every([2,4,6], function (x) { return x % 2 == 0 }); //true
dojo.every([2,4,6,7], function (x) { return x % 2 == 0 }); //false

dojo.some([3,5,7], function f(x) { return x % 2 == 0 }); //false
dojo.some([3,5,7,8], function f(x) { return x % 2 == 0 }); //true
```

## 迭代操作元素

forEach方法会将每个数组元素作为参数传递给一个函数，并且不返回任何值。看到这句话的描述，读者可能会想到平常通过for循环来编写迭代数组元素的代码了。forEach方法的签名如下：

```
dojo.forEach(/*Array*/ array, /*Function*/ function) //不返回值
```

最简单的forEach方法的调用代码如下：

```
dojo.forEach([1,2,3], function(x) {
  console.log(x);
});
```

使用forEach方法最明显的好处，就是无须开发人员明确地使用for循环并声明计数器变量，同时还可以将Array变量直接作为参数嵌入到调用中。不过，最关键的恐怕还是它利用作为第二个参数的函数提供的闭包保护了计数器变量的直接执行环境，以及引入循环块中的其他变量的持续性。与其他实用方法类似，forEach还可以接受一个可选的参数，该参数用于为嵌入的函数提供环境。

为了向读者展示forEach怎样帮我们避免意外的结果，请看下面的代码片段：

```
var nodes = getSomeNodes();

for(var x=0; x<nodes.length; x++){
    nodes[x].onclick = function(){
        console.debug("clicked:", x);
    }
}
```

你觉得这里“x”的值应该是多少？由于匿名函数封闭的只是词法变量x，而不是x的值，因此单击每个节点都会得到最后值。对此，forEach通过创建一个新的词法作用域帮我们解决了问题。以下经过修改后的代码片段展示了怎样才能既迭代数组，又能得到期望的值：

```
var nodes = getSomeNodes();
var idx = 0;
dojo.forEach(nodes, function(node, idx){
    node.onclick = function(){
        console.debug("clicked:", idx);
    }
});
```

## 转换元素

虽然map和filter与forEach的方法签名相同，但从对每个数组元素应用自定义逻辑以及返回另外一个数组并且不会修改原始数组来看，它们又与forEach存在明显的差别。

**警告：** 虽然从技术上看，通过自定义map和filter方法可以修改原始数组，但由于一般都认为它们不会有这种“副作用”，因此如果引入了这种“副作用”，必将导致使用时的小心谨慎，同时也必须提供足够多的注释说明。

正如具有函数式语言（或者带有函数式扩展的编程语言，如Ptython）编程背景的开发人员所深知的，map和filter的简洁语法提供了很多功能，相信读者也一定会很快就喜欢上它们。

假如读者是头一次看到map方法，可能会产生一些神秘感；而之所以我们还称它为自描述的方法，是因为它能够通过转换函数基于数组构建一个映射。下面是使用map方法的示例：

```
var z = dojo.map([2,3,4], function(x) {
  return x + 1
}); //返回[3,4,5]
```

可以对比一下，如果在计算z的值时不使用map方法将会怎样：

```
var a = [2,3,4];
var z = [];
for (var i=0; i < a.length; i++) {
  z.push(a[i] +1);
}
```

与forEach方法类似，直接使用map方法的好处之一就是让整个表达式更加清晰，因而代码也更容易维护。此外，通过匿名函数封装的闭包代码块，同样可以确保中间计算过程中引入的变量保持相对独立的执行环境。

从根据函数来筛选数组的角度来说，filter方法同样也具有自描述的特点。请看下面的示例：

```
dojo.filter([2,3,4], function(x) {
  return x % 2 == 0
}); //返回[2,4]
```

虽然实现与此功能等价的代码块也不算困难，但明显需要更多的输入量和更复杂的逻辑，因此，出现拼写错误和bug的可能性也就增加了：

```
var a = [2,3,4];
var z = [];
for (var i=0; i < a.length; i++) {
  if (a[i] % 2 == 0)
    z.push(a[i]);
}
```

与Base提供的其他数组方法一样，也可以为map或filter传递一个代表转换函数执行环境的参数：

```
function someContext() { this.y = 2; }
var context = new someContext;
dojo.filter([2,3,4], function(x) {return x % this.y==0}, context); //返回[2,4]
```

## 字符串式函数参数

Base还为forEach、map、filter、every和some方法提供了支持简化的“字符串式函数（string-as-function）”参数的能力。通常，使用字符串式函数要比编写完整的函数结构更方便，而且特别适合完成简单的任务。本质上说，字符串式函数参数就是将函数体而不是整个函数作为参数。当下列3个特殊的关键字出现在字符串中时，将具有各自特定的含义：

item
: 引用当前正在处理的项

array
: 引用当前正在处理的数组

index
: 引用当前正在处理的项的索引

下面这两个示例展示了实现同一目标的两种等价手段：

```
var a = new Array(1,2,3,...);

// 为实现一个非常简单的任务而输入很多字符
a.forEach(function(x){console.log(x);}); // 手段一

// 更少的输入能够更快地能够实现同样的任务
a.forEach("console.log(item)"); // 手段二
```

**警告：** 虽然使用较短的字符串式函数可以使代码更简洁，但却不利于追踪bug，因此要慎用。例如，对于下面这个代码片段来说：

```
var a = new Array(1,2,3,...);
a.forEach("console.log(items)"); // 哎呀……item多了个“s”
```

由于关键字item多了一个“s”，导致它不能成为一个迭代项，因此最终的forEach方法什么也不会做。除非读者眼力好，一眼就能看出这个多余的字母，否则，在调试时恐怕得花些时间才能发现这个拼写错误。

## 通过模块管理源代码

编写过一些程序的读者应该知道把相关代码片段组织成相关块的思想，这种块可能被称为库、包或模块。这样，当需要它们的时候可以再使用import语句或者#include预处理指令等机制将它们引入到程序代码中。Dojo为实现这一思想提供的手段分别是dojo.provide和dojo.require。

在Dojo术语中，把可重用的代码块称为资源，把由资源组织而成的代码集合称为模块。Base为导入模块和资源提供了两个极为简便的方法：dojo.require和dojo.provide。简单地说，就是当需要把一个文件作为dojo.require语句可以导入的模块或资源时，应该将dojo.provide语句放在该文件中的第一行。不过，dojo.require也不仅仅是一个类似SCRIPT标签的占位符；它还负责将模块映射到磁盘上的特定位置，取得其中的代码并缓存以前请求过的模块和资源。对于每个dojo.require语句而言，如果相应资源尚未加载过，那么至少需要请求一次服务器，而缓存则会在此时起到极为重要的优化作用，即使是把那些暂时只需要一次的资源缓存起来，并保证它们在本地可用也是很大的优化。

## 为什么要使用模块管理源代码

除了那些非常小的项目之外，通过模块管理源代码的好处是不言而喻的。毕竟，容易维护、简化扩展及嵌入代码的操作，有助于迅速而有效地完成任务。事实上，以前述方式导入代码对Web开发人员来说还很陌生，由于采用了不适当的源代码管理方式而导致项目难以维护的事例可以说屡见不鲜。例如，Web开发中的一种典型的工作流程，是引用位于服务器上一个静态目录内的JavaScript文件，并使用SCRIPT标签将该文件插入页面：

```
<script src="/static/someScript.js" type="text/javascript"></script>
```

当然，在只需插入一两个脚本的情况下，这种做法也不会有什么问题。但是，如果有很

多页面都需要该脚本提供的功能，那该怎么办？那么，可能就需要在多个页面中包含同样的SCRIPT标签。随着时间推移，这些脚本会变得越来越分散，而开发人员必须手工跟踪它们的变化，结果往往会导致难以收拾的局面。没错，在过去一个文件中只包含几百行JavaScript代码的情况下，的确不必考虑如何更好地管理源代码；可是现在的Web应用程序中动辄就要包含成千上万行JavaScript代码。如果没有一种可靠的工具，又怎么能做到按需获取和延迟加载（lazy loading）呢？

除了简化配置管理工作之外，dojo.provide和dojo.require还能支持Util中的构建工具完成很多令人惊叹的任务。例如，将多个文件（每个文件都需要一次同步请求）合并为一个文件，通过减少对文件的请求来减少延时。如果没有这种明确定义相关性的抽象，那么构建工具的功能也就成了无源之水。

像dojo.provide和dojo.require这样定义明确的系统的最后一个好处，就是可以通过按命名空间归类的方式来管理相关资源，从而降低命名冲突的风险并保证代码更容易组织和维护。即使dojo命名空间实际上只是一个通过点表示法简化的嵌套对象的层次结构，但对于组织命名空间和达成同样目标而言仍然是一种非常奏效的手段。

事实上，由于按照命名空间组织资源的方式确实很常用，因此Dojo提供了一个名为dojo.setObject的Base方法。这个方法至少接收两个参数。第一个参数是会自动创建的对象层次，第二个参数是一个映射到该层次的值：

```
dojo.setObject(/* String */ object, /* Any */ value, /* Object */ context)
//返回 Any
```

例2-2示范了如何使用该方法。

例2-2：通过dojo.setObject组织命名空间

```
var foo  = {bar : {baz : {qux : 1}}}; //用“笨方法”嵌套对象
console.log(foo.bar.baz.qux); //显示1

//或者抛开花括号而使用一种轻松的语法……
dojo.setObject("foo.bar.baz.qux", 1); //更轻松的语法
console.log(foo.bar.baz.qux); //显示1

//如果提供了可选的环境参数，那么相对于该环境而不是全局环境（dojo.global）来设置对象
var someContext = {};
dojo.setObject("foo.bar.baz.qux",23,someContext);
console.log(someContext.foo.bar.baz.qux); //显示23
```

尽管dojo.setObject方法仍然属于语法糖（syntactic sugar）的范畴，但它确实能够让代码变得更清晰，同时也消除了匹配花括号的烦琐过程。

注意：OpenAjax Alliance（*http://www.openajax.org*）是由很多知名厂商共同成立的一个联盟，旨在推进高级Web技术的开放性和标准化。该联盟强烈建议使用点对象表示法组织命名空间。

## 跨域自定义模块的示例

下面，我们通过一个简短的示例来进一步展示dojo.require和dojo.provide的用法。首先，考虑一个简单的模块，该模块只提供一个简单的功能，比如计算斐波那契数列（译注1）。在例2-3中，资源同样和一个模块关联。虽然把资源组织到模块中不一定是必需的，但这却是一种良好的习惯。纵贯本书，读者会经常看到用dtdg（即本书英文名《Dojo: The Definitive Guide》的单词首字母缩写）来表示模块的命名空间。

### 斐波那契数列

斐波那契数列以13世纪意大利比萨的数学家莱昂纳多·斐波那契命名，揭示了数字的很多奇妙性质。自那时起，斐波那契数列就经常现身于伪随机数生成器、优化技术、音乐、自然研究领域，而且与黄金分割率也有非常紧密的关联。

斐波那契数列的定义如下：

```
fibonacci(0) = 0
fibonacci(x <= 1) = x
fibonacci(x > 1) =fibonacci(x-1) + fibonacci(x-2)
```

有关契波那契的更多信息，请参考*http://en.wikipedia.org/wiki/Fibonacci_nunmber*。

例2-3：一个简单的模块定义（dtdg.foo）

```
/*
    下面的dojo.provide语句指定所在的.js源文件提供了一个dtdg.foo模块。
    从语义上讲，dtdg.foo模块也为包含在磁盘模块中的函数提供了命名空间。即
    这个文件应该命名为foo.js并放在dtdg目录内。
*/
dojo.provide("dtdg.foo");

//注意相对于模块的命名空间定义函数
dtdg.foo.fibonacci = function(x) {
  if (x < 0)
    throw Exception("Illegal argument");
```

译注1：一种整数数列，其中每个数等于前面两数之和。

```
  if (x <= 1)
    return x;

  return dtdg.foo.fibonacci(x-1) + dtdg.foo.fibonacci(x-2);
}
```

**注意：** 应该始终做到把资源组织成逻辑模块并将它们与命名空间关联。这样做一方面是为了保持良好的编程习惯，另一方面也可以防止与全局命名空间中的其他标识符发生冲突，同时也可以避免其他人重复你的工作。毕竟，这正是使用dojo.provide和dojo.require的首要目的之一。

然后，就可以在想要向别人“炫耀”的时候使用dtdg.foo模块了。换句话说，有了这个模块之后，就无须再重新编写计算斐波那契数列的函数，而只需在页面中的某个地方使用dojo.require请求该模块即可，这样也能避免因为出错导致的尴尬。例2-4展示了如何在通过CDN启用工具箱的情况下使用本地模块。这个示例假设下列HTML文件与dtdg目录被保存在了同级目录下，而dtdg目录中则保存着例2-3中所示的模块文件。

例2-4：在跨域启用工具箱的情况下使用本地模块

```
<html>
    <head>
      <title>Fun With Fibonacci!title>

      <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        djConfig="baseUrl:'./'">
      </script>

      <script type="text/javascript">
        dojo.registerModulePath("dtdg","./dtdg");
        dojo.require("dtdg.foo");
        /* 此时，由于使用Dojo的跨域构建，因此dojo.require是通过异步调用执行的。
        应该把对dtdg.foo的所有引用都放在addOnLoad块中 */

        dojo.addOnLoad(function() {
          dojo.body().innerHTML = "guess what? fibonacci(5) = ", dtdg.foo.fibonacci(5);
        });
      </script>

    </head>
    <body>
    </body>
</html>
```

通过前面的代码，读者应该能够体会到使用dojo.require请求资源有多么直观。不过，其中还有几个关键点需要说明一下。

首先，对于本地安装而言，Dojo会基于工具箱的根目录查找模块。但是，在执行跨域加载的情况下，“真正的”工具箱根目录位于AOL的服务器上面。为此，我们为djConfig传递了一个特殊的配置开关——baseUrl，用于指定查找本地模块（dtdg.foo）的起点位置。

**注意：** djConfig只是为工具箱提供配置参数的一个简单手段，本节后面将会介绍到。现在只要知道它的作用即可。

其次，dojo.registerModulePath方法的作用是在顶级命名空间（其第一个参数）与一个相对于baseUrl的物理目录（其第二个参数）之间建立关联。

**警告：** 在通过dojo.registerModulePath配置模块路径时一定要多加注意。如果忘记了相对目录指的是相对于dojo.js文件，而非工具箱的根目录，那么很可能会相差一级目录。另外，在模块路径结尾添加一个斜杠偶尔也会导致问题，因此也要避免这种做法。

最后，自定义模块中的所有代码都可以通过调用dojo.require而生效。也就是说，如果dtdg.foo模块中还包含其他函数或标识符，那它们在dojo.require("dtdg.foo")语句执行之后也会在当前页面中可用。同样，不能在addOnLoad块外部引用dtdg.foo提供的任何功能。

**注意：** 虽然*.js*源文件与其中dojo.provide语句指定的函数不一定一一对应，但这通常也是一种风格约定。不过，也存在例外情况。比如，在将模块看成API的情况下，其中的某些函数由于属于私有性质，因而不应该暴露出来。

读者可能也注意到前面代码清单中调用了dojo.body()。本质上讲，这个调用只是返回当前文档主体的一种快捷方式，它与document.body相对应，只不过后者不够简洁而已。

## 使用本地工具箱时的斐波那契数列示例

为了对比起见，例2-5展示了一个极为相似的示例。不过，这次的示例使用了Dojo的本

地文件，而且dtdg模块也位于工具箱的根目录下，与包含Core的*dojo*目录属于同一级。因此，无须使用`baseUrl`，也不必再调用`registerModulePath`了。之所以会如此方便，是因为Dojo会从dojo目录的同一级目录（从维护的角度来说，该位置既符合逻辑也比较方便）中自动搜索模块。

例2-5：在使用本地工具箱的情况下请求dtdg.foo

```
<html>
    <head>
      <title>Fun With Fibonacci!title>

      <script
        type="text/javascript"
        src="your/relative/path/from/this/page/to/dojo/dojo.js">
      </script>

      <script type="text/javascript">
        dojo.require("dtdg.foo");
        /* 作为一种习惯，即使所有资源都位于本地，我们仍然使用了addOnLoad块 */
        dojo.addOnLoad(function() {
            dojo.body().innerHTML = "guess what? fibonacci(5) = ", dtdg.foo.fibonacci(5);
        });
      </script>

    </head>
    <body>
    </body>
</html>
```

## 构建Genie示例模块

为了帮助读者全面理解本章讨论的相关概念，接下来我们再构建一个模块。由于人活着有时候的确没那么容易，因此创造一个能够随时排忧解难的精灵（Genie）模块可能会带给我们意想不到的惊喜。（Dojo的某些自动化过程有时候的确让人感觉像是在变魔术，但这里所说的精灵展示的是真正的魔术。）

在构建一个模块之前，首先不要忘记为它想好一个命名空间。例2-6仍然沿用了*dtdg*命名空间（该命名空间是本书迄今为止一直在用的），然后又在该命名空间之后关联了一个*Genie*资源。如果读者还没有创建名为*dtdg*的本地目录，请马上创建一个。在该目录中，新建一个文件并命名为*Genie.js*，再把例2-6的代码输入其中。

例2-6：Genie模块的代码

```
// 首先要包含dojo.provide语句
dojo.provide("dtdg.Genie");
```

```
//为精灵模块设置一个命名空间
dtdg.Genie = function(){}

//添加一些预言
dtdg.Genie.prototype._predictions = [
        "As I see it, yes",
        "Ask again later",
        "Better not tell you now",
        "Cannot predict now",
        "Concentrate and ask again",
        "Don't count on it",
        "It is certain",
        "It is decidedly so",
        "Most likely",
        "My reply is no",
        "My sources say no",
        "Outlook good",
        "Outlook not so good",
        "Reply hazy, try again",
        "Signs point to yes",
        "Very doubtful",
        "Without a doubt",
        "Yes",
        "Yes - definitely",
        "You may rely on it"
    ];

//定义一个初始化函数，用于构建界面
dtdg.Genie.prototype.initialize = function(){

    var label = document.createElement("p");
    label.innerHTML = "Ask a question. The genie knows the answer...";

    var question = document.createElement("input");
    question.size = 50;

    var button = document.createElement("button");
    button.innerHTML = "Ask!";
    button.onclick = function() {
        alert(dtdg.Genie.prototype._getPrediction());
        question.value = "";
    }

    var container = document.createElement("div");
    container.appendChild(label);
    container.appendChild(question);
    container.appendChild(button);

    dojo.body().appendChild(container);
}
```

```
// 创建一个负责交互的主函数
dtdg.Genie.prototype._getPrediction = function() {
    // 取得 0～19 之间的一个数并据之取得相应预言
    var idx = Math.round(Math.random()*19)
    return this._predictions[idx];
}
```

从本质上讲，上面的代码提供了一个名为dtdg.Genie的Function对象，该对象提供了一个名为initialize的“公有”方法。

---

**注意：** 在Dojo中，为私有成员添加下划线前缀的做法很常见，本书也遵循此约定。鉴于私有成员非常不稳定，因此真正遵循这一约定非常重要。

---

例2-6的代码中包含很多注释，因此Web开发人员应该很容易理解该代码。（如果情况并非如此，请读者在继续阅读之前先复习一下HTML和JavaScript的基础知识。）

为了使用这个精灵模块，还需要修改一下基本的模板，如例2-7所示。

例2-7：调用精灵模块的网页

```
<html>
    <head>
        <title>Fun With the Genie!</title>

        <script
          type="text/javascript"
          src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
          djConfig="modulePaths:{dtdg:'./dtdg'},baseUrl:'./'">
        </script>

        <script type="text/javascript">
            // 请求精灵模块
            dojo.require("dtdg.Genie");

            // 在 addOnLoad 中可以放心地引用 dtdg.Genie
            dojo.addOnLoad(function() {

                // 创建实例
                var g = new dtdg.Genie;

                // 执行初始化操作
                g.initialize();
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

这个示例体现了dtdg.Genie模块的可重用性和可移植性。我们只需将它引入页面中并进行初始化，精灵就可以随时“回答问题”了。（只要用户不看源代码，就好像它确实有魔力一样）需要说明的一点是在启用工具箱之前通过djConfig对Dojo进行的配置：modulePaths用于限定模块相对于baseUrl的位置，而后者被定义为了页面所在的当前目录。因而，这个示例涉及的物理目录结构应该如图2-1所示。

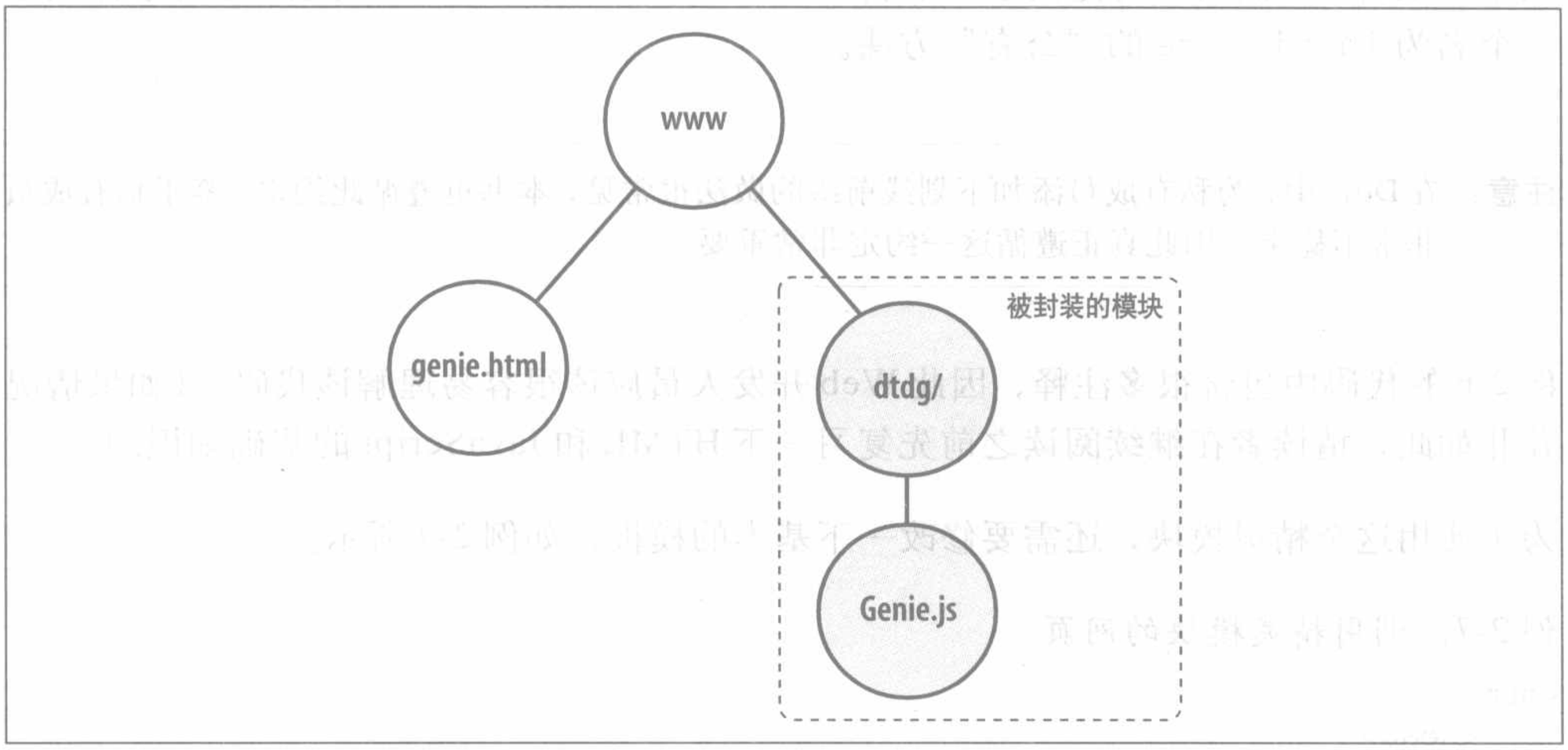



图2-1：模块与页面在磁盘上的相对位置

# JavaScript对象实用程序

Base中提供了3个针对JavaScript对象的实用方法：mixin、extend和clone。

**注意：** Dojo在其dojo.declare实现中使用了mixin和extend方法。dojo.declare是Dojo用于模仿基于类继承的机制，详细介绍将在第10章中进行。

## 对象混入

在JavaScript中，通过以构造函数包装一组属性和方法可以近似地创建轻量级的类。然后，再通过new运算符来调用构造函数就可以创建该类的对象实例。众所周知，有时候为对象实例添加额外属性非常有用，无论是在创建对象实例后动态添加，还是在精心设计类以便重用时提前定义。无论采用哪种方式，mixin方法都为我们处理相关细节提供了一种简捷的途径。

**注意：** 按照JavaScript中面向对象的设计，读者会发现Dojo工具箱中为重用代码块而大量使用了mixin方法。

根据使用mixin方法的API描述，该方法可以接收任意数量的对象作为参数，其中第一个对象将作为混入了其他对象的结果返回：

```
dojo.mixin(/*Object*/ o, /*Object*/ o, ...) //返回对象
```

以下是使用mixin方法的示例：

```
function Man() {
  this.x = 10;
}

function Myth() {
  this.y = 20;
}

function Legend() {
  this.z = 30;
}

var theMan = new Man;
var theMyth = new Myth;
var theLegend = new Legend;

function ManMythLegend() {}
var theManTheMythTheLegend = new ManMythLegend;

//通过混入另外3个对象改变theManTheMythTheLegend
dojo.mixin(theManTheMythTheLegend, theMan, theMyth, theLegend);
```

**警告：** 传递给mixin方法的所有参数都是对象实例，不是函数声明。

## 扩展对象原型

Base的extend方法与mixin类似，它们的区别在于extend是将混入的所有属性和方法都添加到构造函数的原型中。这样，所有基于该构造函数创建的对象实例都将自动包含混入的新属性和新方法：

```
dojo.extend(/*Function*/constructor, /*Object*/props, ... ) //返回Function
```

请注意看下面的示例：

```
function Man() {
  this.x = 10;
}

function Myth() {
  this.y = 20;
}

function Legend() {
  this.z = 30;
}

var theMan = new Man;
var theMyth = new Myth;
var theLegend = new Legend;

function ManMythLegend() {}

var theManTheMythTheLegend = new ManMythLegend;

dojo.extend(ManMythLegend, theMan, theMyth, theLegend);

var theTheManTheMythTheLegend = new ManMythLegend;
```

可见，必须记住的一个主要区别是mixin生成一个对象实例，该实例就是混入了其他对象的结果；而extend实际上修改的是一个函数的原型。

接下来，我们再通过一个更切合实际的示例来加深对extend方法的理解。这个示例展示了使用extend方法以一种更轻量级的方式取代了以前需要修改对象的prototype属性的做法。事实上，使用extend方法不仅是一种编程风格的改变，而且结果也会更简洁。下面的示例使用extend方法重写了例2-6中展示的Genie模块：

```
dojo.provide("dtdg.Genie");

//定义函数对象
dtdg.Genie = function() {}

//扩展函数对象
dojo.extend(dtdg.Genie, {
    _predictions : [
        "As I see it, yet",
        "Ask again later",
        "Better not tell you now",
        "Cannot predict now",
        "Concentrate and ask again",
        "Don't count on it",
        "It is certain",
        "It is decidedly so",
        "Most likely",
```

```
            "My reply is no",
            "My sources say no",
            "Outlook good",
            "Outlook not so good",
            "Reply hazy, try again",
            "Signs point to yes",
            "Very doubtful",
            "Without a doubt",
            "Yes",
            "Yes - definitely",
            "You may rely on it"
        ],

        initialize : function() {
            var label = document.createElement("p");
            label.innerHTML = "Ask a question. The genie knows the answer...";

            var question = document.createElement("input");
            question.size = 50;

            var button = document.createElement("button");
            button.innerHTML = "Ask!";
            button.onclick = function() {
                alert(dtdg.Genie.prototype._getPrediction());
                question.value = "";
            }

            var container = document.createElement("div");
            container.appendChild(label);
            container.appendChild(question);
            container.appendChild(button);

            dojo.body().appendChild(container);
        },

        getPrediction : function() {
            //取得0~19之间的一个数并据之取得相应预言
            var idx = Math.round(Math.random()*19)
            return this._predictions[idx];
        }
    });
```

**警告：** 不要在对象直接量最后一个元素的末尾添加逗号，这种问题在代码重构和大段剪切/粘贴代码时最为常见。虽然Firefox会放过这种做法，但由于IE直接将其视为错误，因此还是记住不要这样做的好。

## 克隆对象

JavaScript在涉及对象和DOM节点的赋值操作时执行的是浅复制，但有时候基于对象层次的深复制，或者说克隆对象才是我们想要的。Base提供的clone方法就是为此目的而设计的。请看下面的示例：

```
function foo() {
    this.bar = "baz";
}

var foo1 = new foo;
var foo2 = foo1; //浅复制

console.log(foo1.bar);
console.log(foo2.bar);

foo1.bar = "qux"; //修改foo1就等于修改了foo2

console.log(foo1.bar);  //qux
console.log(foo2.bar);  //qux

foo3 = new foo
foo4 = dojo.clone(foo3); //深复制

foo3.bar = "qux";

console.log(foo3.bar); //qux
console.log(foo4.bar); //baz
```

## 操作对象环境

在Web应用程序中，尽管全局window对象提供了最外层的环境，但有时候也需要把默认的环境切换为其他对象。例如，需要在用户与应用程序交互期间保持准确的会话状态，或者需要建立一个针对特殊情况的自定义执行环境。每当此时，就可以使用Base提供的window对象实用程序来切换当前环境，不必为配置环境而编码并在各种条件之间进行判断了。

使用下面的方法可以切换到dojo.global和dojo.doc对象。注意，虽然dojo.doc默认只是对window.document的引用，但它却为标识环境的当前文档提供了统一的机制，而且在涉及到管理对象环境的情况下非常有用。另外，dojo.body()是获取文档body节点的一个快捷方式。

---

**警告：** 严格型XHTML文档并没有明确定义body元素，其他一些文档中也没有。

---

至少，应该知道Base为操作环境提供的以下3个方法：

```
dojo.doc //返回Document
dojo.body() //返回DomNode
dojo.setContext(/*Object*/globalObject, /*Document*/globalDocument)
```

最后，出于灵活性的考虑，Base也提供了另外两个方法，以便在不同于当前所在的dojo.global和dojo.doc环境中对函数求值：

```
dojo.withGlobal(/*Object*/globalObject, /*Function*/callback, /*Object*/thisObject,
/*Array*/callbackArgs)
dojo.withDoc(/*Object*/documentObject, /*Function*/callback, /*Object*/thisObject,
/*Array*/callbackArgs)
```

有必要指出的是，使用Dojo方法在另一个document或window对象中进行大量运算并非该工具箱建议的用法。因此，如果读者非要这么做，可能会遇到支持不足的问题。标准的用法通常需要将Dojo加载到每个必要的文档中。不过，对于简单的操作，使用本节讨论的环境方法应该是没有问题的。

## 部分应用参数

Base的partial方法可以让开发人员根据参数的可用情况为函数部分地应用参数，然后在将来某个时刻再最终执行该函数。事实上，也可以一次性地应用所有参数，然后再传递可以执行的函数引用，这样要比同时传递函数及其参数要清晰一些，而且也是Dojo工具箱中常用的一种模式。以下就是partial方法的API：

```
dojo.partial(*/Function|String*/func /*, arg1, ..., argN*/) //返回应用参数后的函数
```

下面这个示例展示了如何使用partial方法为一个求几个数之和的函数分两次应用参数：

```
function addThree(x,y,z){console.log(x+y+z);}

//先应用两个参数
f = dojo.partial(addThree,100,10);

//然后再应用最后一个参数
f = dojo.partial(f,1);

//现在再求值
f(); //111
```

### curry

与partial相关的另一个概念叫做*currying*。curry函数与partial类似，同样支持向返回的函数传递剩余参数；不过，这些函数本身是一等对象，可以逐个向它们传递后续的参数。然后，当全部参数都补齐后，curry函数就会自动执行。这听起来似乎没什么大不了，但只要仔细地想一想就会发现其实它妙不可言。

请注意，partial与curry函数并不相同。虽然可以通过partial为函数部分地应用参数，但当全部参数都应用之后，函数不会自动执行。而且，必须要在应用部分参数及最终执行函数的各个环节明确使用partial，因为它要负责相应的内部处理；也就是说，如果不明确使用partial方法，就不能操作每个处于过渡状态的函数。

假如读者希望更深入地了解curry函数，可以参考dojox.lang.functional.curry模块中的实现。另外，附录B中也简单介绍了DojoX。

## 把对象连接到特定的环境

从把参数部分应用到一个函数的角度上看，Base提供的hitch方法与partial非常相似。不过，hitch方法还可以将函数永久地绑定（或连接）到特定的执行环境中，无论最终的执行环境变成什么。这种方法对于无法预知最终执行环境（即this）的情况下，确保正确执行回调函数特别有用。下面就是它的API：

```
dojo.hitch(/*Object*/scope,/*Function||String*/method /*, arg1, ... , argN*/)
//返回绑定执行环境后的函数
```

以下代码通过重新连接一个对象方法来示范如何使用hitch：

```
var foo = {
    name : "Foo",
    greet : function() {
        console.log("Hi,I'm",this.name);
    }
}

var bar = {
    name : "Bar",
    greet : function() {
        console.log("Hi,I'm",this.name);
    }
}
```

```
foo.greet(); //Hi,I'm Foo
bar.greet(); //Hi,I'm Bar

/* 把bar的greet方法绑定到另一个环境 */
bar.greet = dojo.hitch(foo,"greet");

/ * bar和从前不一样了*/
bar.greet(); // Hi,I'm Foo
```

请注意，由于greet方法明确引用了this这个执行环境，因此以下代码不会导致greet方法的执行环境改变：

```
bar.greet = foo.greet;
bar.greet();
```

**注意：** 如果看到具体的实现，读者可能会感觉更有意思。因为在Base的实现中，hitch方法是partial方法的基础，换句话说，如果在调用hitch方法时将null作为作用域参数，那么结果就和调用partial方法一样。

第4章“将回调绑定到执行环境”中提供了一个示例，其中使用了hitch方法管理异步回调函数中所用数据的环境。该示例展示的是hitch方法的一种典型用途，因为回调函数中的this环境与它的包含对象还是两码事。

## 委托和继承

委托是一种编程模式，即一个对象可以通过其他对象执行某种操作，但不必实现该操作。作为基于原型的语言，JavaScript最为核心的概念就是委托，因为在原型链中解析JavaScript对象的属性遵循的就是委托模式。尽管委托在JavaScript的继承实现（有赖于运行时对原型链的解析）中扮演了至关重要的角色，但在Java和C++等真正基于类的编程语言中，委托作为一种模式仍然和继承有着很大的差别。真正基于类的语言通常（但并不总是）在编译时而非运行时解析类的层次。因此，应该特别注意，作为一种运行时特性，委托的实现依赖于动态绑定的语言特性。

Dojo的delegate方法通过下面的API封装了委托对象函数的细节：

```
dojo.delegate(/*Object*/delegate, properties) //返回对象
```

下面这个示例在前面示例的基础上进一步展开，说明了如何通过委托来分派函数的执行环境：

```
function Foo() {
    this.talk = function() {console.log("Hello, my name is", this.name);}
}
```

```
// 取得一个带有name属性的Function对象
// 但将talk函数的责任分派（或委托）给
// 传入的Foo的实例
var bar = dojo.delegate(new Foo, {name : "Bar"});

// 基于受委托的Foo来解析talk方法
bar.talk();
```

第10章着重介绍由工具箱的dojo.declare方法提供的继承模式，可以使用该方法在JavaScript中模拟基于类的继承；此外，该章还将讨论实现各种继承模式的手段。

# DOM实用程序

前面曾经提到过，Dojo有意不取代JavaScript的核心功能，而是侧重于为该语言增加更有价值的特性，以便开发人员编写的代码更容易移植也更简洁。为此，读者不会看到对常见DOM操作方法，如appendChild、removeChild等的替代方法。但是，Dojo仍然提供了一些有助于简化DOM操作的实用程序。本节就来介绍Base提供的这些DOM实用程序。

## 判定祖先

Base提供了一些能够补充和增强常见DOM操作的方法。第一种方法就是isDescendant，如表2-1所示。该方法接收两个参数（id值或实际的节点对象），第一个参数表示要判定的相关节点，第二个参数是可能的祖先节点。如果要判定的节点是可能的祖先节点的后代，该方法返回true。

表2-1：Base提供的操作和处理DOM的方法

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.isDescendant(/*String \| DomNode*/node, /* String \| DomNode*/potentialAncestor) | 布尔值 | 返回一个布尔值，表示一个节点是否属于某个祖先节点。在嵌套的层次中有效 |

## 设置可选择性

把页面上的文本设置为禁止通过鼠标选中并非少见的需求，而且有时候还能借此增进易用性。虽然在不同的浏览器中实现这一操作要分别采取不同的方式，但我们根本不必担心，因为有Dojo。如果读者在开发中产生了这种需求，只要使用dojo.setSelectable方法即可。该方法的API如下：

```
dojo.setSelectable(/*String | DomNode*/node, /*Boolean*/selectable)
```

**注意：** 众所周知，通过客户端操作来保护敏感内容并不现实。因为，只要是在浏览器中能够看到的内容，都可以通过逆向工程获取。

## 为节点添加样式

Base的dojo.style方法为获取和设置特定节点的个别样式值提供了有效手段。如果为该方法传递的参数是一个节点和一个DOM存取器（accessor）格式的样式名（例如，borderWidth而非border-width），那么就可以取得相应样式的值。如果再为该方法提供第三个DOM存取器格式的样式值参数，那么该方法就会设置相应样式的值。例如，dojo.style（"foo", "height"）会返回id为"foo"的元素的高度，而dojo.style("foo", "height", "100px"）则会将该元素的高度设置为100像素。另外，为该方法传递对象直接量格式的第二个参数可以批量设置元素的样式属性：

```
dojo.style("foo", {
    height : "100px",
    width : "100px",
    border : "1px green"
});
```

虽然多数应用程序都会受益于dojo.style操作具体样式属性的能力，但添加、移除、切换类和检查某个类存在与否的需求同样很常见。Base中用于操作类的方法集合恰好能够满足这些需求，而且这些方法具有相同的方法签名。其中，第一个参数是要操作的DOM节点，第二个参数是一个字符串值，表示要操作的类。例如，为节点添加一个类可以使用dojo.addClass("foo", "someClassName")。注意，其中的类名不包括在样式表中要用的前置句点符号。

表2-2列出了操作节点外观的所有实用方法。

**表2-2：Base提供的样式处理方法**

| 名称 | 说明 |
| --- | --- |
| dojo.style(/*DomNode\|String*/ node, /*String?\|Object?*/style, /*String?*/value) | 用于设置或取得一个节点的样式值 |
| dojo.hasClass(/*DomNode*/node, /*String*/classString) | 如果节点中有特定的类，那么返回true |

表2-2：Base提供的样式处理方法（续）

| 名称 | 说明 |
| --- | --- |
| dojo.addClass(/*DomNode*/node, /*String*/classString) | 为节点添加特定的类 |
| dojo.removeClass(/*DomNode*/node, /*String*/classString) | 从节点中移除特定的类 |
| dojo.toggleClass(/*DomNode*/node, /*String*/classString) | 如果节点中没有特定的类那么添加该类，否则移除该类 |

## 操作属性

在模仿上一节讨论的为节点添加样式的方法基础上，Base也为设置、获取、删除属性，以及检查属性是否存在提供了一系列方法。表2-3列出了这些方法。

表2-3：Base中用于操作节点属性的方法

| 名称 | 说明 |
| --- | --- |
| dojo.attr(/*DOMNode\|String*/node, /*String?\|Object?*/attrs, /*String?*/value) | 用于设置或取得一个节点的属性值 |
| dojo.hasAttr (/*DOMNode\|String*/node, /*String*/name) | 如果节点中有特定的属性，那么返回true |
| dojo.removeAttr (/*DOMNode\|String*/node, /*String*/name) | 从节点中移除一个属性 |

从某种意义上，可以说dojo.attr与dojo.style非常相似，因为如果为它们传递第二和第三个参数来指定属性和值，就可以设置个别属性；如果为它们传递包含属性和值集合的关联数组作为第二个参数，就可以设置多个属性。顾名思义，hasAttr和removeAttr方法的用途也很容易理解。

## 放置节点

虽然通过使用操作DOM内容的appendChild、insertBefore等内置方法也能完成开发任务，但如果有一种放置节点的统一的手段则会更加方便。表2-4中列出的dojo.place方法就提供了这一便利。简单地说，要为该方法传递3个参数：第一个参数表示要放置的节点，第二个参数是参照节点，第三个参数是说明相对位置关系的字符串或数字值。第三个参数的字符串值可以是"before"、"after"、"first"和"last"。其中，

"before"和"after"用于在节点层次中表示横向的前后位置关系，而"first"和"last"则用于在假设参照节点是被放置节点父节点的情况下表示绝对的位置关系。第三个参数也可以用整数值来指定，整数值在这里表示的是被放置的节点在参照节点的子节点中的绝对位置。

表2-4：Base中用于放置节点的方法

| 名称 | 说明 |
| --- | --- |
| dojo.place(/*String\|DomNode*/node, /*String\|DomNode*/refNode, /*String\|Number*/position) | 通过提供相对于参照节点插入节点的统一方法来增强DOM节点的操作能力。返回布尔值 |

## 盒模型

CSS中的盒模型本来是一个很简单的概念，但由于存在太多不一致的实现，因此问题被人为地复杂化了。本节将简单地介绍盒模型的概念，如果想要深入了解各种相关细节，请参考Eric Meyer的权威著作《CSS: The Definitive Guide》（O'Reilly）。

**注意：** 不仅盒模型的实现存在不一致性问题，而且CSS2和CSS3中的盒模型也没有保持一致。CSS2盒模型的相关资料请参考CSS2规范 *http://www.w3.org/TR/REC-CSS2/box.html*，CSS3工作草案的网址为 *http://www.w3.org/TR/css3-box/*。

用最简单的话来讲，所谓的盒模型就是一种具有伸缩性的视觉格式，它通过在页面中排布一系列嵌套的盒子来控制内容的高度和宽度。在解释这句话之前，我们先来看一看图2-2，它传达了盒模型的基本思想。

下面我们通过摘录CSS2.1规范，来回顾一下内容、外边距、内边距和边框盒子的区别。

> 外边距、边框和内边距由左、右、上、下4段组成（例如，在上图中，LM表示左外边距，RP表示右内边距，而TB表示上边框）。图中4个区域（内容、内边距、边框、外边距）的周长称为“边界”，因此每个盒子都有4个边界：
>
> 1－内容边界或内边界
>
> 内容边界围绕的是所呈现的内容；

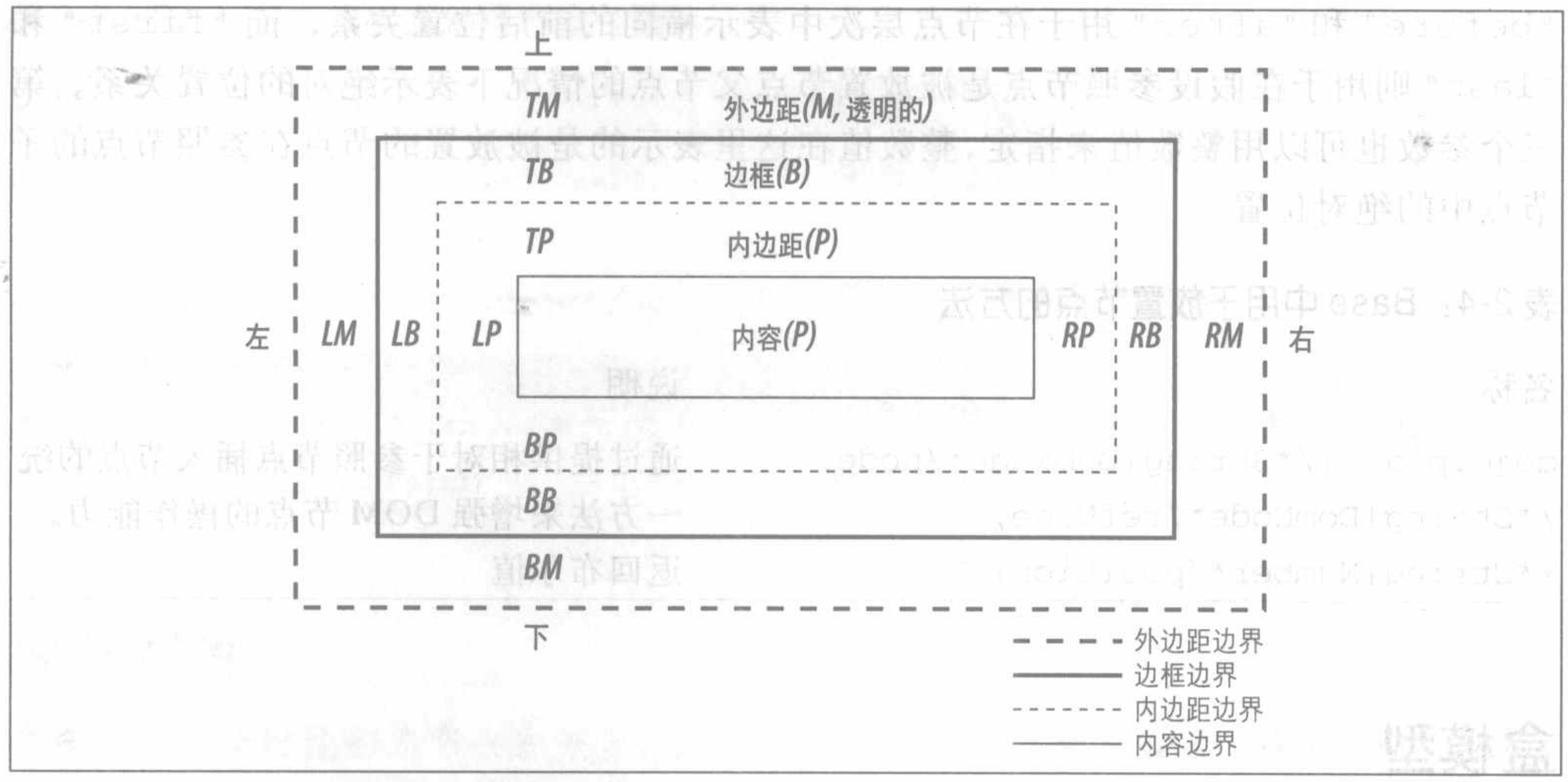



图 2-2：CSS2.1 盒模型定义的宽度和高度

2 －内边距边界

内边距边界围绕着盒子的内边距。如果内边距的宽度为零，那么内边距边界与内容边界重合。盒子的内边距边界定义了由盒子建立的包含块（containing block）的边界；

3 －边框边界

边框边界围绕着盒子的边框。如果边框的宽度为零，那么边框边界与内边距边界重合；

4 －外边距边界或外边界

外边距边界围绕着盒子的外边距。如果外边距的宽度为零，那么外边距边界与边框边界重合。

结果，由于对以上盒模型定义的不同理解，导致了内容盒子与边框盒子的分歧。这两种实现方式之间的根本差别，可以通过回答如下问题得到解释：外边距和边框怎样应用到内容区？基于内容盒子的实现认为，由内边距和边框围绕的任何区域都算作内容区宽度和高度之外的区域。基于边框盒子的实现认为，内边距和边框的区域包含在内容区宽度和高度的范围之内。换句话说，在内容盒子的实现方式下，宽度和高度仅指内容区的宽度和高度；而在边框盒子的实现方式下，宽度和高度指的是包含边框在内的宽度和高度。

---

**注意：** 多数现代的浏览器都支持两种模式：标准模式和quirks模式。标准模式下采用的是内容盒子的实现方式，而quirks模式下采用的是边框盒子的实现方式。

---

假如读者只想为内容周围留出一些间隙，并不想实现非常精确的控制，那么上述差别的影响并不大，因此一般都可以通过几种方式达到这一目的。但是，假如读者想要实现的效果要求非常精确，那么就要看你自己的理解能力了，因为要在多个浏览器间保持完全一致的外观确实很有（或者很没有）意思。

Dojo为消除上述盒模型大小计算方面的差异，提供了dojo.boxModel属性（可能的值为"content-box"或"margin-box"）和dojo.marginBox、dojo.contentBox方法。dojo.boxModel的默认值为"content-box"，而后两个方法可以用来取得盒子的坐标值。无论何时，下面表格中列出的box参数都表示一个对象，其中包含盒子区域的宽度、高度以及盒子区域左上角的坐标值。如果外边距盒子（即调用dojo.marginBox方法）的返回值为{ l: 50, t: 200, w: 300, h: 150 }，那么说明相应节点左侧偏离父元素50px，上方偏离父元素200px，包括外边距在内的宽度和高度分别是300px和150px。

读者可以自己试验一下上述属性和方法。请把下面的代码复制到一个本地文件中，然后在Firefox中打开：

```
<body style="margin:3px">
      <div id="foo" style="width:4px; height:4px; border:solid 1px;"></div>
</body>
```

下列代码中的注释给出了在Firebug中执行它们时显示的结果，而图2-3中则展示了该盒子在浏览器中的位置和大小：

```
console.log("box model", dojo.boxModel); // content-box
console.log("content box", dojo.contentBox("foo")); // l=0 t=0 w=4 h=4
console.log("margin box", dojo.marginBox("foo")); // l=3 t=3 w=6 h=6
```

如同本章介绍的其他方法一样，以一个表示节点的参数调用上述方法返回一个对象值，而以两个参数调用上述方法则会设置相应节点的值。表2-5列出了与盒模型相关的所有方法。

表2-5：与盒模型有关的方法

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.marginBox(<br>/*DomNode\|String*/node,<br>/*Object?*/box) | 对象 | 返回一个对象，包含节点外边距盒子的位置及大小值 |
| dojo.contentBox(<br>/*DomNode\|String*/node,<br>/*Object?*/box) | 对象 | 返回一个对象，包含节点内容盒子的位置及大小值 |

表2-5：与盒模型有关的方法（续）

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.coords(/*HTMLElement*/node, /*Boolean*/includeScroll) | 对象 | 返回一个对象，包含节点的外边距盒子的位置及大小值（绝对定位的位置及大小值包括在内）。除了t、l、w、h值外，x和y值分别表示元素在页面中的绝对坐标；如果将includeScroll设置为true（默认值为false），x和y将包含相对于浏览器视口（viewport）的偏移量。不能使用dojo.coords设置元素位置及大小 |

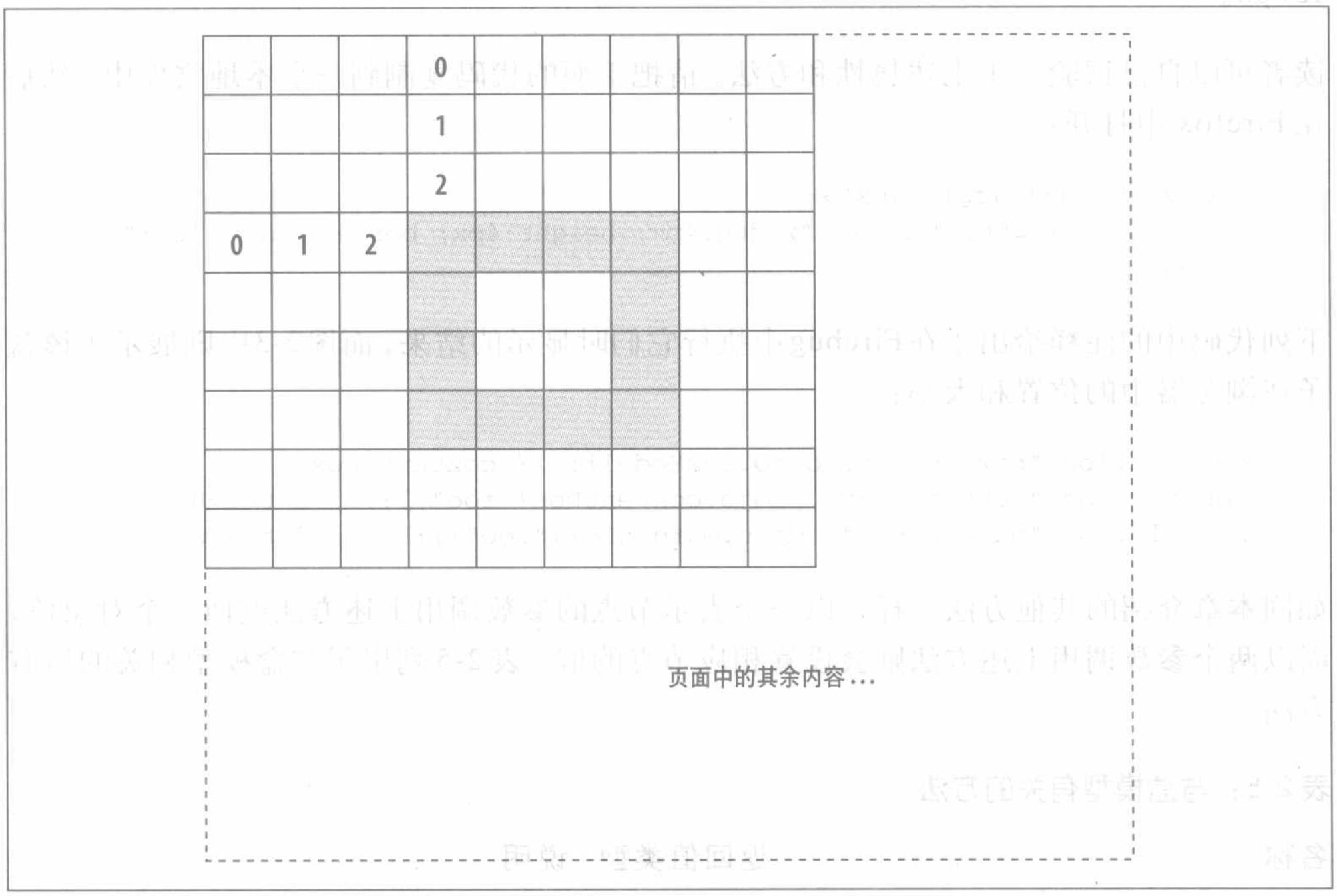



图2-3：浏览器中显示的示例页面

**注意：** Dijit充分利用了盒模型实用程序来创建跨浏览器外观一致的部件。

# 浏览器实用程序

本节介绍Dojo为管理cookie和浏览器的后退按钮（现代Web应用程序开发中两个相当常见的主题）提供的实用程序。由于这两方面的功能是由Core提供的，因此在使用之前不要忘了先通过dojo.require导入它们。

## 浏览器检测

尽管Dojo能够在消除浏览器不一致性方面替我们做很多工作，但有时候读者可能仍然需要检测一下浏览器——无论出于什么原因。每当此时，读者就可以使用下列Base中提供的属性，并将它们当作布尔值来对待（任何大于1的值都相当于true），以省去编写大量繁琐代码的麻烦：

- dojo.isOpera
- dojo.isKhtml
- dojo.isSafari
- dojo.isMozilla
- dojo.isFF
- dojo.isIE
- dojo.isAIR
- dojo.isQuirks

其中，isMozilla对使用Mozilla的Gecko解释引擎的任何实现都返回true，而isFF则仅对在Firefox浏览器中使用的Gecko解释引擎返回true。

如果浏览器当前处于向后兼容（quirks）模式，那么isQuirks属性的值为true。除非页面的第一个元素明确设定了导致其他模式的DTD，否则绝大多数浏览器默认都会运行在quirks模式下。

## cookie

HTTP是一种无状态协议，即当Web服务器发送完一个网页之后，就与客户端浏览器毫无关系了。虽然Web的这一特征在很多方面都表现得不错，但在某些情况下却很不尽人意，例如，在需要根据用户定义的偏好显示个性化页面时。具体来说，如果一个发布天气消息的站点能够记住用户的邮政编码，那么用户就不必每次都重新输入了。

cookie最早是由Netscape为解决上述问题而提出的一种解决方案，即为浏览器赋予一种短期记忆的能力。换句话说，网页设计人员可以通过JavaScript或服务器端脚本创建cookie，在其中以名/值对形式保存用户所访问页面的数据。当用户再次访问同一页面时，脚本可以取得cookie并根据其中的数据动态修改页面的内容和功能。cookie一般都有预先设定的失效日期，而且都与产生该cookie的域紧密关联。

使用纯JavaScript来操作cookie的一个主要问题，就是开发人员必须记住严格的语法，并自己动手构建相应的字符串数据。例如，要针对当前域为foo=bar这个名/值创建一个cookie并加上失效日期，那么相关的代码应该如下所示：

```
document.cookie ='foo=bar; expires=Sun, 15 Jun 2008 12:00:00 UTC; path=/'
```

当然，这还是相对容易的。在要读取这个cookie的值时，还必须自己来解析其中的字符串，而这个字符串中通常都包含很多个名/值对。

Dojo为操作cookie提供了一个基本的包装方法，其使用也很简单。表2-6展示了该方法的基本API。

表2-6：dojo.cookie方法

| 名称 | 说明 |
| --- | --- |
| `dojo.cookie(/*String*/name, /*String*/value, /*Object?*/properties)` | 如果只传递第一个参数，即表示cookie值的名称，那么该方法作为“getter”返回相应cookie的值。如果传递了前两个参数，那么该方法就会作为“setter”设置cookie的值。最后一个参数properties，针对具体的cookie属性可以包含下列键/值对：<br>`expires (Date\|String\|Number)`<br>如果是数字值，表示从今天起cookie失效的天数；如果是日期值，表示cookie失效的日期（如果日期已过，那么cookie会被删除）；如果省略expires或该值为零，那么cookie在浏览器关闭时失效。<br>`path (String)`<br>有权使用cookie的路径。<br>`domain (String)`<br>有权使用cookie的域。<br>`secure (Boolean)`<br>是否只在安全连接的情况下发送cookie |

表 2-6：dojo.cookie 方法（续）

| 名称 | 说明 |
| --- | --- |
| dojo.cookie.isSupported() | 返回布尔值，表示浏览器是否支持cookie |

例如，可以像下面这样设置和取得 cookie 值：

```
dojo.cookie("foo","bar", {expires : 30});
// 设置一个 foo = bar 的键 / 值对，将失效日期设置为从现在起 30 天
dojo.cookie("foo"); // 取得 foo 的值——bar
```

## 后退按钮的处理

对于现代Web应用程序来说，整个应用程序都位于一个页面中，而且从来不需要刷新页面的情况再正常也不过了。但是，由此而必须考虑的一个问题，就是应用程序必须对状态变化作出正确的反应，同时还要支持书签功能。Core 中的 back 模块为此提供了一组简单的实用程序，让开发人员可以根据用户按下后退或前进按钮来明确定义状态并作出反应。表 2-7 列出了相应的 API。

表 2-7：dojo.back 中的方法

| 名称 | 说明 |
| --- | --- |
| init() | 由于 IE 的问题，需要在页面 BODY 内的 SCRIPT 标签中调用。如果读者知道自己开发的应用程序不会运行在 IE 中，也可以不调用这个方法 |
| setInitialState(/*Object*/args) | 用于定义在页面返回到初始状态时应该执行的回调函数。通常，建议在addOnLoad中最先调用这个函数 |
| addToHistory(/*Object*/args) | 用于设置回调状态，参数args中包含在按下后退和前进按钮时应该执行的回调函数，以及是否在状态改变时通过URL给出标记 —— 便于用户添加书签。args 的键 / 值构成如下所示：<br>back (Function)<br>因为按下后退按钮而进入某一状态时应该执行的回调函数。<br>forward (Function)<br>因为按下前进按钮而进入某一状态时应该执行的回调函数。 |

表2-7：dojo.back中的方法（续）

| 名称 | 说明 |
| --- | --- |
| | changeUrl (Boolean\|String)<br>如果值为true，那么为URL末尾添加一个随机的标记，用于程序内部跟踪。如果是字符串值，那么将该字符串添加到URL末尾，既可以用于程序内部跟踪，也便于用户添加书签。不能混合布尔和字符串值，二者只能使用一种 |

**警告：** 请读者务必牢记一点，在传递给addToHistory的args对象参数的changeUrl键中，要么使用布尔值，要么使用字符串值，不能混合使用这两种值。

例2-8简单地展示了back模块的用法，通过提供具有自定义行为的回调函数，基本上可以反映出Dojo对后退按钮的解决之道。注意页面主体中加粗的代码行，那几行代码是确保在IE中正常处理后退按钮的关键。

**注意：** 读者可能会问，为什么非要把SCRIPT标签放到BODY中呢，这样显得多笨啊？这牵扯到在IE中执行document.write的一个特殊问题，因为页面加载后就无法执行该方法了。虽然那几行代码显得有点笨拙，但有了它们就可以保证后退按钮在所有浏览器中都能正常使用。

例2-8：后退按钮处理的示例

```
<html>
    <head>
        <title>Fun with Back!</title>

        <link rel="stylesheet" type="text/css"
         href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
           djConfig="dojoIframeHistoryUrl:'iframe_history.html',isDebug:true"
        ></script>

        <script type="text/javascript">

            dojo.addOnLoad(function() {
               initialState = {
```

```
            back: function() { console.log("Back to initial state"); }
        };
        state1 = {
            back: function() { console.log("Back to state 1"); },
            forward: function() { console.log("Forward to state 1"); },
            changeUrl : true // 也可以使用“state1”这样的字符串标记
        };
        state2 = {
            back: function() { console.log("Back to state 2"); },
            forward: function() { console.log("Forward to state 2"); },
            changeUrl : true // 也可以使用“state2”这样的字符串标记
        };

                    // 设置初始状态，并在历史记录中前移两步
                    dojo.back.setInitialState(initialState);
                    dojo.back.addToHistory(state1);
                    dojo.back.addToHistory(state2);
                });
            </script>
        <head>
            <body>
                <script type="text/javascript"
                    src="http://o.aolcdn.com/dojo/1.1/dojo/back.js"></script>
                <script type="text/javascript">dojo.back.init();</script>
                Press the back button and have a look at the console.
            </body>
</html>
```

# 小结

在学习完本章之后，读者应该：

- 理解Base中包含的各种特性。
- 能够配置djConfig以注册模块路径，并且知道如何在启用工具箱的过程中配置其他选项。
- 理解如何使用dojo.addOnLoad和dojo.addOnUnload方法，并且明白dojo.addOnLoad可以防止代码中出现竞态条件。
- 可以通过dojo.provide和dojo.require构造并为自己的模块创建命名空间。
- 理解如何（以及何时）使用映射（map）、筛选（filter）和forEach方法。
- 知道mixin和extend之间的区别，并且能够恰当地使用它们。
- 能够高效地使用Dojo的hasClass、removeClass、addClass和toggleClass方法操作样式。

- 理解CSS盒模型的基本概念，并且可以使用coords和marginBox等方法修改DOM节点的位置。
- 知道Base的数组处理实用程序。
- 能够在对象和不同的环境之间建立连接。
- 能够处理cookie。
- 能够使用Core提供的实用程序处理单页应用程序中的后退按钮。

下一章，我们将介绍事件侦听器和发布/预订通信。

第3章

# 事件侦听器及发布/预订通信

Base为JavaScript对象、DOM节点以及它们的组合之间通信提供了极其有用和强大的方法。本章将介绍这些方法，并说明每个方法在什么时候使用最恰当。如果想要编写涉及DOM事件的易移植代码，就必须依靠标准化的事件模型。因此，本章还将介绍Dojo如何处理鼠标和键盘事件在不同浏览器间的不一致性问题。本章最后为读者介绍发布/预订通信模型，该模型为构建组件之松散耦合的架构提供了有效手段。

## 事件和键盘标准化

Dojo工具箱中最早有一批代码是专门为消除浏览器间事件模型的不一致而编写的。本节简要介绍使用Dojo开发应用程序时经过标准化的事件，标准化的基础是W3C的事件模型。

### 鼠标和键盘事件的标准化

以下几节讨论的dojo.connect方法经常涉及到在某个DOM节点上发生的鼠标事件。只要使用Dojo开发，那么读者就大可放心地按照W3C标准来运用下列鼠标和键盘事件：

```
onclick
onmousedown
onmouseup
onmouseover
onmouseout
onmousemove
onkeydown
```

```
onkeyup
onkeypress
```

**注意：** 除了支持标准的W3C事件之外，Dojo也支持非标准的onmouseenter和onmouseleave事件。

Dojo不仅让这些事件可以按照标准的方式触发，而且也对传递给事件处理函数的事件对象进行了标准化。事实上，如果读者需要自己来标准化某个事件，可以使用下面的Base方法：

```
dojo.fixEvent(/*DOMEvent*/ evt, /*DOMNode*/ sender) //返回 DOMEvent
```

**注意：** DOMEvent是本书引用DOM事件对象的一种标准约定。

换句话说，只要传入事件和应该被视为当前目标的节点，那么就可以得到一个符合W3C规范的事件对象。表3-1列出了最常用的DOMEvent（注1）属性。

表3-1：常用的DOMEvent属性

| 名称 | 类型 | 说明 |
|---|---|---|
| bubbles | 布尔值 | 表示事件是否能够沿DOM树向上冒泡 |
| cancelable | 布尔值 | 表示事件的默认动作是否可以被取消 |
| currentTarget | DOMNode | 表示正在执行其事件侦听器的当前节点(在事件冒泡时有用) |
| target | DOMNode | 表示最早接收到事件的节点 |
| type | 字符串 | 事件类型，例如，mouseover |
| ctrlKey | 布尔值 | 表示事件发生时用户是否按下了Ctrl键 |
| shiftKey | 布尔值 | 表示事件发生时用户是否按下了Shift键 |
| metaKey | 布尔值 | 表示事件发生时用户是否按下了Meta键（即Apple计算机中的Command键） |
| altKey | 布尔值 | 表示事件发生时用户是否按下了Alt键 |
| screenX | 整数 | 表示事件发生位置在屏幕上的X坐标 |

注1：目前，Dojo是基于DOM2事件规范实施的标准化，DOM2事件规范的网址为*http://www.w3.org/TR/DOM-Level-2-Events/events.html*。DOM3事件规范的网址为*http://www.w3.org/TR/DOM-Level-3-Events/events.html*。

表 3-1：常用的 DOMEvent 属性（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| screenY | 整数 | 表示事件发生位置在屏幕上的 Y 坐标 |
| clientX | 整数 | 表示事件发生位置在浏览器视口中的 X 坐标 |
| clientY | 整数 | 表示事件发生位置在浏览器视口中的 Y 坐标 |

## 键码的标准化

Dojo 工具箱通过 dojo.keys 提供了下表中所列示的命名键码。举例来说，如果要检测用户是否同时按下了 Shift 和 Enter 键，可以使用下面的代码：

```
/* ……省略的代码…… */
  if (evt.keyCode == dojo.keys.ENTER && evt.shiftKey) {
    /* …… */
  }
/* ……省略的代码…… */
```

表 3-2 中列出了访问键盘事件的常量。

表 3-2：通过 dojo.keys 访问键盘事件的常量

| | | |
| --- | --- | --- |
| BACKSPACE | DELETE | NUMPAD_DIVIDE |
| TAB | HELP | F1 |
| CLEAR | LEFT_WINDOW | F2 |
| ENTER | RIGHT_WINDOW | F3 |
| SHIFT | SELECT | F4 |
| CTRL | NUMPAD_0 | F5 |
| ALT | NUMPAD_1 | F6 |
| PAUSE | NUMPAD_2 | F7 |
| CAPS_LOCK | NUMPAD_3 | F8 |
| ESCAPE | NUMPAD_4 | F9 |
| SPACE | NUMPAD_5 | F10 |
| PAGE_UP | NUMPAD_6 | F11 |
| PAGE_DOWN | NUMPAD_7 | F12 |
| END | NUMPAD_8 | F13 |
| HOME | NUMPAD_9 | F14 |
| LEFT_ARROW | NUMPAD_MULTIPLY | F15 |

| | | |
|---|---|---|
| UP_ARROW | NUMPAD_PLUS | NUM_LOCK |
| RIGHT_ARROW | NUMPAD_ENTER | SCROLL_LOCK |
| DOWN_ARROW | NUMPAD_MINUS | |
| INSERT | NUMPAD_PERIOD | |

# 事件侦听器

在Dojo中，通过明确地连接函数和/或DOM事件可以建立一个直接的通信渠道，从而在一个事件发生时自动触发另一个事件。例如，当一个对象被“setter”方法修改时，可以自动触发应用程序界面的变化，或者，每当一个对象被修改时，都可以自动更新另一个对象中的派生属性，等等。总之，可以利用这种能力的情形不胜枚举。

与上述直接通信模式相关的两个主要方法是dojo.connect和dojo.disconnect。简单地说，可以使用dojo.connect连接一系列事件。每次调用dojo.connect都会返回一个句柄(handle)对象，该对象可以在将来需要断开连接时传递给dojo.disconnect。虽然等到页面卸载时所有句柄对象中的连接都能自动断开，但在一个长期运行的应用程序大量使用了临时连接的情况下，手工管理这些句柄会更有利于避免内存泄露（这个问题在IE中尤为突出）。下面就是我们在第1章中提到过的API。

---

**警告：** 记住，不要在页面加载完成之前连接任何事件。在页面加载前使用dojo.connect是一个很常见的错误，它会导致开发人员陷入无休止的调试中，因为在初次面临这种问题时，往往很难找到原因所在。所以，应该始终在位于dojo.addOnLoad中的函数内部建立连接。

---

建立和断开连接非常简单。首先来看一看实现相应操作的API：

```
/* 建立连接 */
dojo.connect(/*Object|null*/ obj,
             /*String*/ event,
             /*Object|null*/ context,
             /*String|Function*/ method) // 返回一个句柄对象

/* 断开连接 */
dojo.disconnect(/*Handle*/handle);
```

---

**注意：** 其实，读者完全可以把调用dojo.connect返回的句柄对象当作一个黑盒子，因为除了将来把它传递给disconnect之外根本用不到它。(如果读者确实想搞清楚这个句柄对象到底是怎么回事，那我可以告诉你它确实没有什么特别的。因为这个对象只是保存着一些与连接有关的信息，以便内部管理连接时使用。)

---

接下来，我们通过一个示例来说明一下dojo.connect的用法：

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm Foo"); }
}

function Bar() {
    this.greet = function() { console.log("Hi, I'm Bar"); }
}

foo = new Foo;
bar  = new Bar;

foo.greet();

//即使foo不知道bar是否存在
//bar也应该也问候一下foo
```

要解决这个难题，其实只要一行代码即可。请修改前面的代码，然后在Firebug中进行测试：

```
function Foo() {
    this.greet = function() { console.log("Hi, I'm foo"); }
}

function Bar() {
    this.greet = function() { console.log("Hi, I'm bar"); }
}

foo = new Foo;
bar  = new Bar;

//只要foo.greet执行，bar.greet也会紧跟着执行……
var handle = dojo.connect(foo, "greet", bar, "greet"); //建立连接
foo.greet(); //现在，bar就会自动地回应foo了!
```

不知读者意识到没有，多写这么一行脚本的代价还是蛮高的。其中，为dojo.connect传递的第二和第四个参数分别是基于各自执行环境的以字符串直接量表示的方法，而返回的handle则可以在将来用于断开连接。一般来说，必须在将来的某个时刻断开以前建立的连接，无论是因为完成既定设计功能的需要，还是为了执行最后的清理工作，例如，销毁对象或页面卸载。断开连接的代码如下所示：

```
var handle = dojo.connect(foo, "greet", bar, "greet");
foo.greet();

dojo.disconnect(handle);

foo.greet(); //这下等待foo的只有沉默了
```

使用dojo.connect为事件建立连接不仅简单，而且代码也很清晰、容易维护。没有冗余操作，没有曲折离奇的代码，无须搭配自己的解决方案，更没有难以维护的恶梦。

根据页面变化触发方法的执行固然很有用，但读者早晚都会遇到需要传递参数的情形。为此，dojo.connect也提供了一个自动传递参数的特性，即可以将参数从第一个环境的函数传递给第二个环境的函数。下面通过示例来说明这一特性：

```
function Foo() {
    this.greet = function(greeting) { console.log("Hi, I'm Foo.",
greeting); };
}
function Bar() {
    this.greet = function(greeting) { console.log("Hi, I'm Bar.",
greeting); };
}

foo = new Foo;
bar = new Bar;

var handle= dojo.connect(foo, "greet", bar, "greet");
foo.greet("Nice to meet you");
```

读者应该能够想像出上面代码的执行结果。事实上，能够自动传递参数的确非常方便，一个典型的例子就是当把一个函数与一个DOM事件（如鼠标单击）连接起来时，通过传递事件对象可以让函数访问到事件目标、鼠标位置等等重要信息。下面这个示例可以说明这一点：

```
//注意，由于此时的处理程序是一个匿名函数，因此第三个参数被省略了
//如果在第三个参数的位置上传递null，结果完全相同

dojo.connect(
dojo.byId("foo"), //某个DOM元素
"onmouseover",
function(evt) {
  console.log(evt);
});
```

如果读者通过一个测试页面建立上面的连接，并观察Firebug控制台中的输出结果，就会看到事件处理函数能够直接访问的事件对象。通过该事件对象，开发人员几乎可以完全掌握所发生事件的全部信息。

“为DOM事件指定处理程序不是很简单吗，为什么还要学习一种奇特的库方法呢？”有的读者可能会提出这个问题。没错，指定几个简单的事件处理程序肯定不用麻烦脑外科医生。但是，如果你面对的是一个复杂的应用程序，其中涉及用户偏好、自定义事件以及很多事件驱动的行为，那么处理起来恐怕就没那么容易了。当然，完全通过手工处理不是不可能，但为什么放着一个现成的，而且是经过了实践检验的统一的接口不用呢？

最后说明一下，虽然前面的示例中只涉及了连接两个事件，但连接并连续执行任意数量的函数、对象方法和DOM事件都是可行的。

## 事件传播

有时候，也需要抑制浏览器对某些DOM事件的内置处理，而不是一味地通过dojo.connect为这些任务提供自定义的处理程序。需要抑制浏览器处理DOM事件的典型情况有两种：一是在用户单击超链接时，不让浏览器自动导航到相应地址；二是当用户按下回车键或单击提交按钮时，避免浏览器自动提交表单。

令人欣慰的是，在你的自定义处理程序执行完成后，要想阻止浏览器处理上述DOM事件，只要调用dojo.stopEvent或者DOMEvent的preventDefault方法，即可防止事件传播到浏览器。其中，stopEvent方法只需要一个DOMEvent对象作为参数：

```
dojo.stopEvent(/*DOMEvent*/evt)
```

**注意：** 虽然可以阻止dojo.connect连接的个别DOM事件，但却无法在函数或JavaScript对象方法的内部阻止由dojo.connect建立的事件链。

以下就是使用stopEvent的一个示例：

```
var foo = dojo.byId("foo"); //某个锚元素

dojo.connect(foo, "onclick", function(evt) {
    console.log("anchor clicked");
    dojo.stopEvent(evt); //抑制浏览器导航并阻止事件冒泡
});
```

同样地，阻止表单的自动提交也很容易，只需更换要连接的执行环境并将匿名函数与submit事件关联即可。不过，这次我们要使用DOMEvent的preventDefault方法来抑制事件，因为应该允许事件继续冒泡：

```
var bar = dojo.byId("bar"); //某个表单元素

dojo.connect(bar, "onsubmit", function(evt) {
    console.log("form submitted");
    evt.preventDefault(); //抑制浏览器导航但允许事件冒泡
});
```

## 通过dojo.connect利用闭包

本节介绍的内容涉及到了中高级主题，如果读者暂时觉得不好理解可以先略过不读。不

过，要相信这些知识迟早会在开发中派上用场，因此将来一定要回过头来把这些内容搞明白。

### 一次性连接

在某些情况下，我们需要建立一个连接，而当该连接涉及的函数被触发后就应该断开该连接。下面这个示例展示了如何以最少的代码实现这一操作：

```
var handle = dojo.connect(
    dojo.byId("foo"),  //某个div元素
    "onmouseover",
    function(evt) {
        //此处省略了处理程序的代码……
        dojo.disconnect(handle);
    }
);
```

如果读者熟悉闭包，那么第一反应可能就是指出前面的示例不可行。毕竟，变量handle是由对dojo.connect的调用返回的，但在作为参数传递给dojo.connect的函数内部却引用了该变量。为了说明这个示例的可行性，我们做了下面的分析。

1. 当dojo.connect执行时，尽管它接收了一个匿名函数作为参数，但是该匿名函数并没有被执行。
2. 匿名函数内部的任何变量（如handle）都将被绑定到其作用域链中，虽然这些变量存在于函数内部，但除非执行该函数，否则它们不会真正被引用。因此，也就不会出现引用错误。
3. 由于dojo.connect在匿名函数执行以前返回handle变量，因此当匿名函数再执行时，该变量实际上已经存在了，因此可以直接传递给dojo.disconnect方法。

### 在循环中建立连接

另一种在开发中常见的情形是在循环体内建立连接。假设一个页面中包含一系列元素：foo0、foo1、foo2、……、foo9，我们想在鼠标移动到这些元素上面时分别记录它们的编号。起初，你可能会编写下列代码，但这些代码并不能达到目的：

```
/* 下列代码无法实现我们设定的目标 */
for (var i=0; i < 10; i++) {
  var foo = dojo.byId("foo"+i);
  var handle = dojo.connect(foo, "onmouseover", function(evt) {
    console.log(i);
    dojo.disconnect(handle);
  });
}
```

如果通过Firebug来基于相应页面运行以上代码，读者将发现它存在一个问题。也就是说，Firebug控制台中始终只会出现编号10，这说明每次执行事件处理程序所引用的都是i的最终值，而且10个处理程序却错误地只断开了最后一个连接。再仔细地想一想，也许你就会突然明白产生这个结果是正常的：由于传递给dojo.connect的匿名函数会创建闭包，而闭包中的变量在执行该函数之前不会解析i的值，因而每次执行处理程序都会得到i的最终值。

下面经过修改后的代码，通过把i的值放入闭包的作用域链中，确保了将来引用该值时解析到的将是执行dojo.connect语句时的值：

```
for (var i=0; i < 10; i++) {
   (function() {
      var foo = dojo.byId("foo"+i);
      var current_i = i; //把i的当前值放入闭包中
      var handle = dojo.connect(foo, "onmouseover",
        function(evt) {
          console.log(current_i);
          dojo.disconnect(handle);
        }
      );
   })(); //立即执行匿名函数
}
```

这段代码乍一看不太好理解，但实际上它非常简单。整个循环体中包含着一个嵌入的匿名函数，该匿名函数为其中的所有变量提供了闭包，而i的值也被“截留”为current_i，以便事件处理程序执行时解析。同样，handle变量的引用也能够得到正确解析，因为该变量也存在于由嵌入的匿名函数创建的闭包当中。

假如读者从未见过如此利用闭包的做法，那么可能就要多花点时间认真研究一下以上代码，并争取把它搞懂搞透。这么说可能会引起个别读者的不快，但在从事JavaScript开发的职业生涯中，如果能透彻地掌握闭包，一定会令你的生活更轻松自在。

## 在标记中建立连接

最后，有必要指出的是，在为Dijit部件建立连接时甚至都不需要编写JavaScript代码，而只要在标记中使用带有专门的dojo/connect属性的SCRIPT标签即可。相关的更多内容将在第11章正式介绍Dijit时再做讨论。

# 发布/预订通信

虽然dojo.connect提供的直接“连接式”通信能够解决不少问题，但在需要多个部件

匿名通信的情况下，则更适合使用间接的“广播式”通信。在这些情况下，可以使用dojo.publish和dojo.subscribe。

一个经典的例子就是按照一对多的关系实现一个JavaScript对象与其他对象之间的通信。此时，与建立并管理多个具有高内聚性的dojo.connect连接相比，更简单的方案应该是让一个部件在事件发生时发布通知（同时也可以传递数据），而让其他部件能够预订该通知并在事件发生时自动执行相应操作。这种方式的妙处在于负责广播的对象不需要知道什么对象预订了通知，甚至无须知道其他对象是否存在。另一个采用这种通信方式的经典例子就是*portlets* —— 类似Dashboard的可插拔界面组件（详见*http://en.wikipedia.org/wiki/Portlet*）。

---

**注意：** 第4章将要介绍到的OpenAjax Hub（*http://www.openajax.org/OpenAjax%20Hub.html*）提倡将发布/预订通信模式作为在同一个页面中使用多个JavaScript库而避免冲突的有效手段。

---

在很多情况下，通过建立连接也可以实现与使用发布/预订通信方式相同的功能，因此是否使用发布/预订通信往往取决于其实用性、方便性，以及要解决的具体问题。

假设现在要决定采用哪种通信方式，那么应该考虑的问题大致如下：

- 是否想（并且能够可靠地）为将要开发的部件提供一个API？如果答案是否定的，那么就应该选择发布/预订通信方式，以便可以了随时放手修改底层设计而不会陷入对应该如何修改API的无休止的争论中。
- 设计中是否包含相同类型的多个部件，而且这些部件是否都要响应同一个事件？如果是，那么就应该选择连接方式，否则就得编写更多的代码来解决哪个部件应该响应哪个通知的问题。
- 设计的部件是否以“has-a（有一个）”的关系包含子部件？如果是，那么应该选择建立和维护连接的方式。
- 设计是否涉及到一对多或多对多的关系？如果是，那么应该选择发布/预订通信才能保持通信负载的最小化。
- 设计是否需要完全匿名并要求尽可能松散耦合？如果是，那么应该使用发布/预订通信方式。

言归正传，下面我们就来看一看支持发布/预订通信的API。注意，调用dojo.subscribe方法时可以省略context参数，该方法在内部能够按照我们的意愿处理参数（与dojo.connect方法一样）：

```
dojo.publish(/*String*/topic, /*Array*/args)
dojo.subscribe(/*String*/topic, *Object|null*/context,
   /*String|Function*/method) //返回一个句柄对象
dojo.unsubscribe(/*Handle*/handle)
```

**注意：** 和dojo.connect返回的句柄对象一样，读者也可以将dojo.subscribe返回的句柄对象看成一个黑盒子。

接下来，我们看一个使用dojo.subscribe和dojo.publish的示例：

```
function Foo(topic) {

  this.topic = topic;

  this.greet = function() {
    console.log("Hi, I'm Foo");

    /* Foo直接发布信息，但不针对某个具体的目标……*/
    dojo.publish(this.topic);
  }
}

function Bar(topic) {

  this.topic = topic;

  this.greet = function() {
    console.log("Hi, I'm Bar");
  }

  / * Bar直接预订信息，但不针对某个具体的来源……*/
  dojo.subscribe(this.topic, this, "greet");
}

var foo = new Foo("/dtdg/salutation");
var bar = new Bar("/dtdg/salutation");

foo.greet(); //Hi, I'm Foo...Hi, I'm Bar
```

**注意：** 虽然没有正式的标准，但Dojo采用了加前缀的约定并使用正斜杠分隔主题（topic）名称。由于在JavaScript代码中正斜杠并不常用，因此可以很容易识别主题名称（假如使用点号分隔主题名称，恐怕要在源代码中找到它们就困难多了）。

我们知道，dojo.connect只能在特定的来源与特定的目标之间建立连接，而发布/预订通信由于采用广播方式，因此对发送通知的来源和接收通知的目标没有任何限制。松

散耦合的架构之所以强大，原因就在于它能够以最低的投入和最简单的设计，使应用程序可以在概念上被理解为一种相关插件的集合。

下面，我们通过对Bar的实现稍加修改来示范如何取消预订。以下代码将导致Bar只对Foo发布的主题做出一次响应：

```
function Bar(topic) {

    this.topic = topic;

    this.greet = function() {
        console.log("Hi,I'm bar");
        dojo.unsubscribe(this.handle);
        //好了好了，不再理会它们了!
    }

    this.handle = dojo.subscribe(this.topic, this, "greet");
}
```

此外，也可以为publish提供第二个参数，该参数必须是一个数组，这个数组最终将被subscribe处理程序以单个命名参数的形式来接收。

---

**警告：** 一个常见的错误，就是忘记传递给dojo.publish数据必须要放入数组中，而dojo.subscribe的处理程序是以单个参数的形式来接收这些参数。

---

最后，我们再考虑一种可能的情形。假设你不确定是否可以修改Foo的greet方法，以便在其中包括对dojo.publish的调用，也许是因为存在禁止这样做的外部限制，或者那些代码一开始就不是你编写的。不过，无论怎样也无须担心，此时可以使用另一个方法dojo.connectPublisher，让它在特定事件发生时负责向我们发布通知：

```
function Foo() {
  this.greet = function() {
    console.log("Hi, I'm foo");
  }
}

function Bar() {
  this.greet = function() {
    console.log("Hi, I'm bar");
  }
}

var foo = new Foo;
var bar = new Bar;
```

```
var topic = "/dtdg/salutation";
dojo.subscribe(topic, bar, "greet");
dojo.connectPublisher(topic, foo, "greet");

foo.greet();
```

**注意：** 事实上，connectPublisher的后台工作原理就是每当特定的函数被调用时，就通过dojo.connect在该函数调用与dojo.publish调用之间建立一个连接。

通过最后这个示例我们得出的结论是，调用dojo.connectPublisher可以实现与在greet方法中调用dojo.publish相同的结果，但却不需要修改相应的源代码。从这个意义上说，foo扮演了通知的间接发送人的角色，它对后续的通信可以一无所知。另一方面，作为通知的预订者，bar则必须明确知道此次通信的过程。从本质上看，上述过程与在典型的dojo.connect调用中，为连接提供环境的对象明确知道为连接提供“目标”的其他对象或函数恰好相反。

# 小结

通过学习本章，读者应该：

- 知道dojo.connect能够标准化传入事件处理程序中的事件对象，因而提供了跨平台的移植能力。
- 理解如何使用dojo.connect连接任意DOM事件、JavaScript对象事件和普通函数，从而创建事件驱动的响应模式。
- 使用发布/预订通信建立连接并在应用程序中实现松散耦合的通信架构。
- 知道在应用程序架构设计中使用dojo.connect与发布/预订通信的利弊得失。

下一章，我们将介绍Ajax及服务器通信。

# 第4章

# Ajax及服务器通信

本章的中心线索是服务器端通信，要介绍的内容涉及执行异步请求、使用IFRAME在后台提交表单、JSON（JavaScript Object Notation，JavaScript对象表示法）的串行化和反串行化以及使用JSONP（JSON with Padding，参数式JSON）。此外，本章还会介绍`Deferred`类，该类为处理异步请求提供了统一的接口，是Dojo工具箱IO子系统的关键所在。

## Ajax简介

Ajax（注1）（Asynchronous JavaScript and XML，异步JavaScript和XML）这种倍受关注的技术为Web开发注入了全新的活力。过去，为了更新页面内容必须向服务器发送同步请求并完全重载页面，而如今，使用JavaScript的`XMLHttpRequest`对象则可以让页面与传统桌面应用程序媲美。XHR是`XMLHttpRequest`对象的简写，一般用来表示该对象支持的任何操作。

如图4-1所示，现在的Web页面可以在后台通过异步请求从服务器取得内容，当内容返回后，会有回调函数对其进行处理。（图4-1基于*http://adaptivepath.com/ideas/essays/archives/000385.php*中的相应插图绘制。）虽然概念很简单，但Ajax的确为用户带来了完全不同的Web体验，也催生了新一代的RIA（Rich Internet Applications，富因特网应用程序）。

---

注1：虽然Ajax中的“X”明确表示XML，但Ajax实际上指的是通过`XMLHttpRequest`对象执行异步请求的任何架构，与服务器返回什么数据类型没有关系。尽管从技术上讲，更精确的术语应该是XHR，但本书将遵循约定俗成的说法，从宽泛的意义上使用Ajax这一术语。

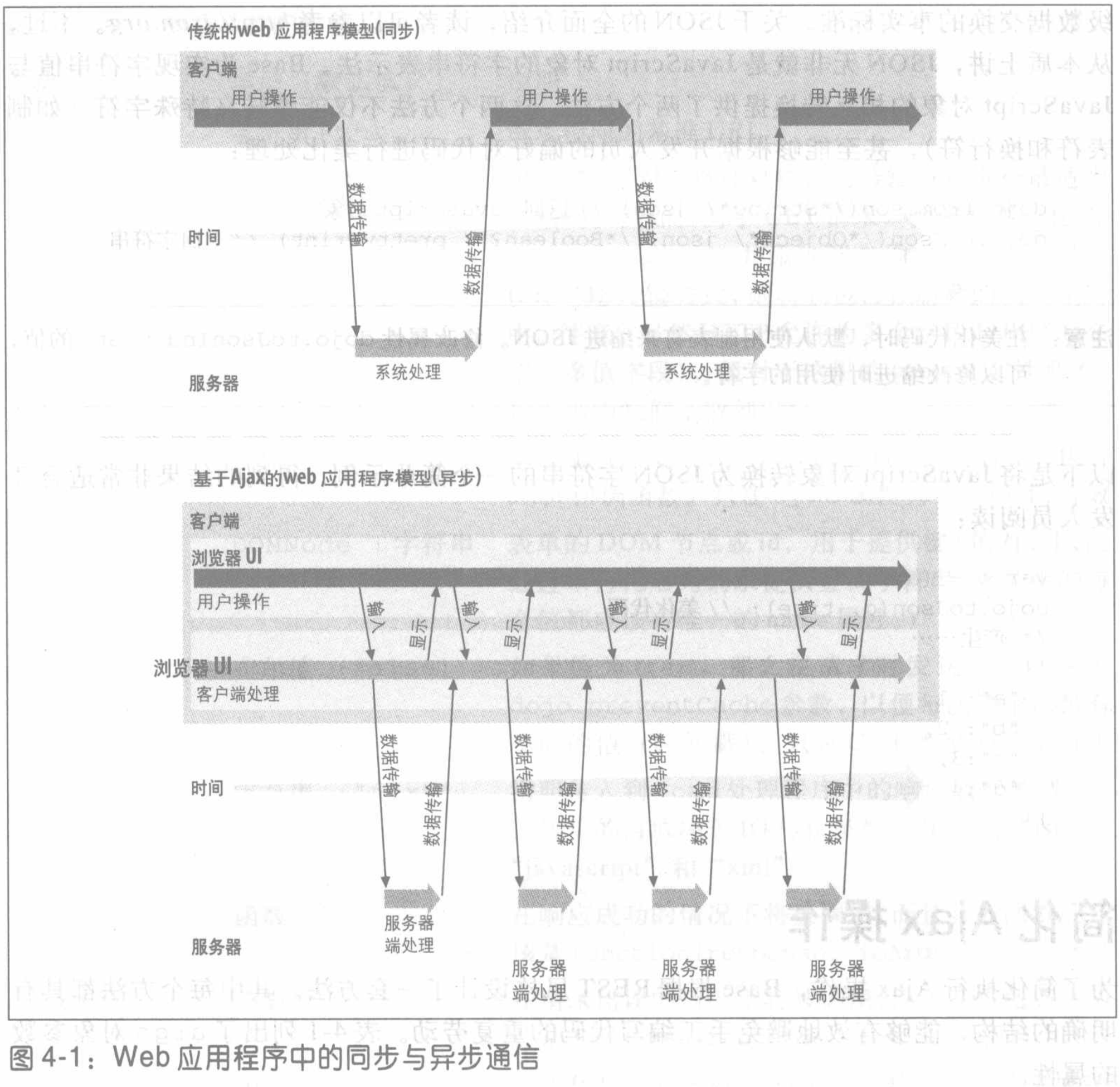



图 4-1：Web 应用程序中的同步与异步通信

使用JavaScript的XMLHttpRequest对象虽然不像火箭研究那样复杂，但与其他技术一样，也会存在一些技巧性的实现细节以及针对常见用途编写的模板代码。例如，异步请求不会绝对保证总能返回值（虽然绝大多数情况下会），因此可能就需要实现一些逻辑，以决定请求经过多长时间算超时及超时后如何处理；可能还需要一些辅助方法自动检查JSON字符串并处理它和JavaScript对象之间的转换；再有，还需要将处理成功请求与错误请求的逻辑分开，等等。

## JSON

在讨论 Ajax 之前，有必要先介绍一下 JSON。JSON 目前已经成为 Ajax 应用程序中轻量

级数据交换的事实标准。关于JSON的全面介绍，读者可以参考*http://json.org*。不过，从本质上讲，JSON无非就是JavaScript对象的字符串表示法。Base为实现字符串值与JavaScript对象的相互转换提供了两个方法。这两个方法不仅能够转义特殊字符（如制表符和换行符），甚至能够根据开发人员的偏好对代码进行美化处理：

```
dojo.fromJson(/*String*/ json) //返回JavaScript对象
dojo.toJson(/*Object*/ json, /*Boolean?*/ prettyPrint) //返回字符串
```

**注意：** 在美化代码时，默认使用制表符来缩进JSON。修改属性`dojo.toJsonIndentStr`的值，可以修改缩进时使用的字符。

以下是将JavaScript对象转换为JSON字符串的一个简单示例，得到的结果非常适合开发人员阅读：

```
var o = {a:1, b:2, c:3, d:4};
dojo.toJson(o, true); //美化代码
/* 产生……
'{
  "a": 1,
  "b": 2,
  "c":3,
  "d":4
}'
```

# 简化Ajax操作

为了简化执行Ajax操作，Base根据REST风格设计了一套方法。其中每个方法都具有明确的结构，能够有效地避免手工编写代码的重复劳动。表4-1列出了`args`对象参数的属性。

## REST

REST是“Representational State Transfer”（表述性状态转移）的简写，是主要与Web相关的一种架构风格。REST风格突出以资源为中心的设计理念，使用URI来定义资源地址。HTTP方法GET、PUT、POST、DELETE分别从语义上对应着对相关资源的操作。例如，GET请求*http://example.com/foo/id/1*，表示要取得`id`为1的*foo*资源，而使用相同URI的DELETE请求则表示要删除该资源。

《RESTful Web Services》是关于REST的权威参考书，作者是Leonard Richardson和Sam Ruby （O'Reilly）。

表4-1：args对象参数的属性

| 名称 | 类型（默认值） | 说明 |
| --- | --- | --- |
| url | 字符串（""） | 请求指向的基准URL |
| content | 对象（{}） | 包含键/值对，将针对特定的传输方式进行最适当的编码。例如，对于GET请求，会将它串行化为name1=value2并添加到查询字符串中；对于IFRAME传输方式，则将它包含在隐藏的表单字段中。注意，虽然HTTP允许为多个字段起相同的名字（多值字段），但对于这里的content属性则不行，因为它们是散列对象 |
| timeout | 整数（Infinity） | 等待响应的毫秒数。如果指定的时间已过，则执行error回调函数。只在sync属性为false时有效 |
| form | DOMNode \| 字符串 | 表单的DOM节点或id，用于提供键/值对，以便经过串行化后为请求提供查询字符串（表单中的每个值都应该有唯一的name属性。） |
| preventCache | 布尔值 （false） | 如果值为true，那么在请求时发送一个特殊的dojo.preventCache参数，以便每次请求都带有不同的值（时间戳）。只对GET类型的请求有用 |
| handleAs | 字符串 （"text"） | 指明传入到load处理程序中的响应数据类型。可以接受的值取决于IO传输类型："text"、"json"、"javascript"和"xml" |
| load | 函数 | 在响应成功的情况下将被调用，而且其方法签名应该是function(response, ioArgs) {/*...*/} |
| error | 函数 | 在请求错误的情况下将被调用，而且其方法签名应该是function(response, ioArgs) {/*...*/} |
| handle | 函数 | 用于代替load和error的函数，无论请求成功与否，都将被调用 |
| sync | 布尔值 （false） | 指明是否执行同步请求 |
| headers | 对象（{}） | 包含在请求中的额外HTTP头部信息 |
| postData | 字符串（""） | 在POST请求体中发送的原始数据。只在使用rawXhrPost时有效 |
| putData | 字符串（""） | 在PUT请求体中发送的原始数据。只在使用rawXhrPut时有效 |

Dojo提供了以下REST风格的XHR方法。到Dojo1.1版为止，这些方法都将自动设置X-Requested-With: XMLHttpRequest头部信息。关于args参数，将在下面讨论。

**注意：** 所有这些XHR方法都返回一个名为Deferred的特殊对象，关于Deferred对象，将在下一节讨论。当前，读者只需理解这些方法即可。

`dojo.xhrGet(/*Object*/args)`

执行XHR GET请求。

`dojo.xhrPost(/*Object*/args)`

执行XHR POST请求。

`dojo.rawXhrPost(/*Object*/args)`

执行XHR POST请求，并允许开发人员提供包含在POST请求主体内的原始数据。

`dojo.xhrPut(/*Object*/args)`

执行XHR PUT请求。

`dojo.rawXhrPut(/*Object*/args)`

执行XHR PUT请求，并允许开发人员提供包含在PUT请求主体内的原始数据。

`dojo.xhrDelete(/*Object*/args)`

执行XHR DELETE请求。

`dojo.xhr(/*String*/ method, /*Object*/ args, /*Boolean?*/ hasBody)`

通用的XHR方法，允许开发人员定义执行异步请求的任何HTTP方法。

表4-1中的各项都比较容易理解，但我们要单独介绍一下传入load和error函数中的参数。第一个参数response，表示服务器返回的响应，至于应该如何解释该响应，则由handleAs指定。handleAs的默认值是"text"，如果将其指定为"json"，那么会导致将响应强制转型为一个JavaScript对象，以便按照JSON格式对其中的数据进行解析。

**注意：** 在load和error 函数中，必须记住返回response值。本章后面将会介绍到，所有XHR实用方法都会返回一个类型为Deferred的对象，只有在load和error函数中返回response值才能保证Deferred中包含的回调函数（callback）和错误处理程序（errback）能够连接在一起。

第二个参数是ioArgs，其中包含发送请求时传递给服务器的最终参数信息。这个ioArgs虽然在开发程序时用处不大，但在调试程序时特别有用。表4-2列出了可以通过ioArgs参数取得的信息。

表4-2：ioArgs参数的属性值

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| args | 对象 | 传递给IO调用的原始参数 |
| xhr | XMLHttpRequest | 在请求中使用的XMLHttpRequest对象 |
| url | 字符串 | 调用中最终使用的URL，由于会添加查询参数，因此一般都和提供的URL不同 |
| query | 字符串 | 只对非GET请求有效，这个值保存着与请求一起传递的查询字符串参数 |
| handleAs | 字符串 | 表示应该如何解释响应 |

## XHR示例

在最低限度下，XHR请求的参数中应该包括表示请求目标的URL以及load函数，但是，一般来说还应该包含一个错误处理函数。因此，如果没有特殊的需求，最好不要省略error函数。请看下面这个最小化的XHR示例：

```
//……省略的代码……
dojo.addOnLoad(function() {
    dojo.xhrGet({

        url : "someText.html",   // 相关的URL

        // 如果请求成功，那么运行这个函数
        load : function(response, ioArgs) {
            console.log("successful xhrGet", response, ioArgs);

            // 设置某个元素的内容……
            dojo.byId("foo").innerHTML= response;

            return response; // 总是返回response
        },

        // 如果请求不成功，那么运行这个函数
        error : function(response, ioArgs) {
            console.log("failed xhrGet", response, ioArgs);

            /* 处理错误的代码……*/

            return response; // 总是返回response
        }
    });
});
//……省略的代码……
```

在实际开发当中，服务器返回的可能不是纯文本，可能需要为请求设置超时时间，也可能要传入额外的查询字符串信息。这些需求在使用XHR时都非常容易满足 —— 只要再增加几个参数即可。例如：

```
dojo.xhrGet({

    url : "someCommentFilteredJSON.html",
    //返回类似/*{'bar':'baz'}*/的响应

    handleAs : "json",
    //清除响应中的注释并将其作为JavaScript对象来求值

    timeout: 5000, //如果在5秒钟后还没有响应就调用error处理函数
    content: {foo:'bar'}, //将foo=bar作为查询字符串添加到URL末尾

    //如果请求成功，则运行这个函数
    load : function(response, ioArgs) {
        console.log("successful xhrGet", request, ioArgs);
        console.log(response);

        //handleAs参数的值告诉Dojo先清除响应中的注释
        //然后再将响应的其他数据转换为JavaScript对象

        dojo.byId("foo").innerHTML= response.bar;
        //这次元素内容将显示baz

        return response; //总是返回response
    },

    //如果请求不成功，则运行这个函数
    error : function(response, ioArgs) {
        console.log("failed xhrGet");
        return response; //总是返回response
    }
});
```

请读者务必注意，如果为handleAs指定的值不适当，则会产生难以追踪的Bug。例如，要是无意中忽略了handleAs参数，但load函数却将响应值作为JavaScript对象来解释，那么就会导致一个令人讨厌的错误 —— 至少在你意识到是因为把文本当成了对象之前，要在其他地方费些时间反复查找错误的原因。毕竟，日志中记录的字符串和对象的值几乎没什么区别。

另外，虽然GET请求是开发中使用最多的请求类型，但在需要PUT、POST、DELETE请求时，可不要忘了更换相应的方法。其他方法的使用与dojo.xhrGet相似，但在使用dojo.rawXhrPut和dojo.rawXhrPost时，还应该分别指定putData或postData参数，用于指定要发送给服务器的数据。以下是dojo.rawXhrPost方法的使用示例：

```
dojo.rawXhrPost({
```

```
    url : "/place/to/post/some/raw/data",
    postData : "{foo : 'bar'}", //要发送的数据是一个JSON直接量
    handleAs : "json",

    load : function(response, ioArgs) {
        /* 请求成功后要执行的某些操作 */
        return response;
    },

    error : function(response, ioArgs) {
        /* 请求失败后要妥善地处理错误 */
        return response;
    }
});
```

## 通用的XMLHttpRequest调用

Dojo1.1版中引入了一个通用的dojo.xhr方法，其签名如下所示：

```
dojo.xhr(/*String*/ method, /*Object*/ args, /*Boolean?*/ hasBody)
```

事实上，本章介绍的所有XHR方法都是在包装这个方法的基础上定义的。例如，dojo.xhrGet实际上就是下面这个包装方法：

```
dojo.xhrGet  = function(args) {
    return dojo.xhr("GET", args); //指定方法名时必须全部大写!
}
```

虽然本节介绍的XHR包装方法能够满足一般需要，但是，当在某些项目开发中需要以编程方式配置XHR请求或者没有现成的包装方法可用时，就需要用dojo.xhr。例如，要执行没有对应包装方法的HEAD请求，可以这样做：

```
dojo.xhr("HEAD", {
    url : "/foo/bar/baz",
    load : function(response, ioArgs) { /*...*/},
    error : function(response, ioArgs) { /*...*/}
});
```

## 将回调绑定到执行环境

我们在第2章中介绍过hitch方法，该方法用于绑定函数的执行环境。由于XHR回调函数的环境与执行该回调函数的代码块的环境不相同（或者说不能混淆），因此使用hitch把XHR回调函数绑定到正确的环境就是一件顺理成章的事了。以下示例首先展示了一种不使用hitch而确保执行环境正确的常见模式，这种模式用到this的别名技术：

```
// 假设有下列 addOnLoad 代码块，而该代码块实际上可以是任何 JavaScript 对象
dojo.addOnLoad(function() {

        //foo 被绑定在了匿名函数的环境中
        this.foo = "bar";

        // 为 this 起个别名，以便在 load 回调函数的内部正确地引用它……
        var self=this;
        dojo.xhrGet({
            url : "./data",
            load : function(response, ioArgs) {
                // 在此，必须使用 this 的别名才能正确地引用 foo……
                console.log(self.foo, response);
            },
            error : function(response, ioArgs) {
                console.log("error", response, ioArgs);
            }
        });

});
```

当然，在这个简单的示例中，不使用hitch似乎也没有什么问题。但是，如果一而再、再而三地使用this的别名，或者再为this起个其他的别名，就很容易把代码搞乱。因此，当你下次再遇到需要为this起别名的情况时，可以试一试下面这个使用hitch的解决方案：

```
dojo.addOnLoad(function() {

        //foo 被绑定在了匿名函数的环境中
        this.foo = "bar";

        // 把一个回调函数绑定到当前环境，以便在其中直接通过 this 引用 foo
        var callback = dojo.hitch(this, function(response, ioArgs) {
            console.log("foo (in context) is", this.foo);
            // 仍然可以继续使用 response 和 ioArgs 参数……
        });

        dojo.xhrGet({
            url : "./data",
            load : callback,
            error : function(response, ioArgs) {
                console.log("error", response, ioArgs);
            }
        });

});
```

此外，由于hitch也可以接收额外的参数，因此就能方便地为回调函数传入必要的参数，例如：

```
dojo.addOnLoad(function() {

        //foo被绑定在了匿名函数的环境中
        this.foo = "bar";

        //把一个回调函数绑定到当前环境，以便在其中直接通过this引用foo
        var callback = dojo.hitch(
            this,
            function(extraParam1, extraParam2, response, ioArgs) {
                console.log("foo (in context) is", this.foo);
                //仍然可以继续使用response和ioArgs参数……
            },
            "extra", "params"
        );

        dojo.xhrGet({
            url : "./data",
            load : callback,
            error : function(response, ioArgs) {
                console.log("error", response, ioArgs);
            }
        });

});
```

如果需要传递的额外参数很多，那么也可以使用arguments，但必须记住response和ioArgs是它的最后两个值。

## Deferred对象

目前，JavaScript不支持线程的概念，但是，它却提供了通过XMLHttpRequest对象执行异步请求以及使用setTimeout函数设置延迟的能力。不过，如果同时生成的异步调用过多，那么问题也会变得很复杂。为此，Base提供了一个Deferred类，用以管理对异步事件的复杂实现。与其他抽象手段类似，Deferred同样在统一的接口下隐藏了各种微妙的逻辑以及/或者重复操作。

如果要用一句话来概括Deferred的特点，恐怕就是它能让开发人员对所有基于网络的I/O操作一视同仁，而不必区分同步还是异步。无论对于执行中的Deferred、失败的Deferred，还是成功的Deferred，连接回调函数（callback）和错误处理函数（errback）的过程都完全相同。可想而知，这必将极大简化当前的编程工作。

**注意：** Dojo中对Deferred的实现是在MochiKit实现基础上稍加改进而成的，后者的实现受到了Twisted相同实现的启发。在*http://www.mochikit.com/doc/html/MochiKit/Async.html#fn-deferred*中可以看到MochiKit实现的背景资料，关于Twisted对Deferred的实现信息，可以参考*http://twistedmatrix.com/projects/core/documentation/howto/defer.html*。

Deferred的主要特性包括连接多个回调函数和错误处理函数，以便按照预订次序执行这些函数；允许开发人员提供一个取消程序，以便明确地中断异步请求。此时此刻，读者也许对上述特性还没有什么认识。事实上，本章前面介绍的所有XHR方法都会返回Deferred对象，但当时还不具备讲解Deferred的条件。不仅如此，Dojo工具箱中所有基于网络的输入/输出机制都使用并返回Deferred，而这完全得益于Deferred在管理基于网络调用的异步操作时所具有的灵活性。

在重温前面的XHR示例之前，我们先来看一个直接使用Deferred对象的例子，该例子为下面进一步讨论相关概念奠定了基础：

```
// 创建Deferred对象
var d = new dojo.Deferred(/* 可选的取消函数 */);

// 添加一个回调函数
d.addCallback(function(response) {
    console.log("The answer is", response);
    return response;
});

// 添加另一个回调函数，在前一个回调函数执行后触发
d.addCallback(function(response) {
    console.log("Yes, indeed. The answer is", response);
    return response;
});

// 添加一个错误处理函数，以防请求过程出错
d.addErrback(function(response) {
    console.log("An error occurred", response);
    return response;
});

// 可以继续添加更多回调函数 / 错误处理函数……

/* 大量的中间代码 */

// 在将来的某个地方，通过下面语句启动回调函数链
d.callback(46);
```

如果在Firebug中试验上面的例子，可以看到以下输出：

```
The answer is 46
Yes, indeed. The answer is 46
```

在继续分析更复杂的示例之前，有必要先看一看表4-3中列出的Deferred提供的API。

表4-3：Deferred对象的方法及属性

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| addCallback(/*Function*/handler) | Deferred | 为成功的回调函数链中添加一个回调函数 |
| addErrback(/*Function*/handler) | Deferred | 为错误的回调函数链中添加一个回调函数 |
| addBoth(/*Function\|Object*/ context, /*String?*/name) | Deferred | 添加一个可以同时处理成功和错误请求的回调函数。用于添加必须保证运行的代码 |
| addCallbacks(/*Function*/callback, /*Function*/errback) | Deferred | 同时添加成功回调函数和错误回调函数 |
| callback(/*Any*/value) | N/A | 执行回调函数链 |
| errback(/*Any*/value) | N/A | 执行错误回调函数链 |
| cancel() | N/A | 取得请求并执行提供给构造函数的取消函数（如果提供了的话） |

当下列可能性分别出现或以不同组合形式出现时，Deferred对象将处于错误状态：

- 回调函数或错误回调函数接收到了一个Error对象；
- 回调函数或错误回调函数抛出了异常；
- 回调函数或错误回调函数返回一个Error对象。

注意：一般情况下，不会用到Deferred对象的canceller、silentlyCancelled和fired属性。其中，canceller引用提供给构造函数的取消函数，silentlyCancelled用于确定Deferred是否是在未注册canceller方法的情况下被取消的，而fired则用于确定Deferred的触发状态。fired属性的值包括以下几个。

-1：没有值（初始状态）。

0：成功执行了回调函数链。

1：有错误发生。

## 使用 CherryPy 的 Deferred 示例

接下来，我们编写一个服务器端脚本，让它在短暂的停顿之后提供一些内容。（短暂的停顿是为了模拟异步请求的网络延迟。）

提供上述功能的完整 CherryPy 文件如下所示：

```
import cherrypy
from time import sleep
import os

# foo.html 文件包含执行 XHR 请求的 Dojo 代码
# 而这是通过下面的 config 指令设置的
current_dir = os.getcwd()
config = {'/foo.html' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}

class Content:

    # 这个方法负责提供内容
    @cherrypy.expose
    def index(self):
        sleep(3) # 有意在响应前添加 3 秒钟延迟
        return "Hello"

# 启动 Web 服务器并令其侦听 8080 端口
cherrypy.quickstart(Content(), '/', config=config)
```

### 同源策略

有必要指出的是，这里对 CherryPy 进行了额外的设置，以便它为我们提供一个静态文件，然后我们就可以基于这个静态文件来执行XHR请求。之所以要这样做，是因为出于安全考虑，浏览器不允许JavaScript的`XMLHttpRequest`对象执行跨站点脚本。因此无法在浏览器中打开类似*file:///foo.html*这样的本地文件，并在其中使用`dojo.xhrGet`从*http://127.0.0.1:8080/*请求文件。没错，这两个文件确实都在本地，但是它们却不在同一个域中。下一节将介绍一种名为 JSONP 的技术，可以用该技术来避开同源策略的安全限制并从其他域中加载内容，而且该技术也是创建像 Mashup 这样的应用程序的一种手段。除此之外，其他实现跨域加载内容的方式涉及通过基于 Flash 的插件或 ActiveX 控件打开套接字。但无论如何，当在自己的域中执行来路不明的脚本时，都可能面临安全威胁，因此一定要慎重为之。

假设我们将上面的CherryPy代码保存在一个名为*hello.py*的文件中，那么需要在终端输入python hello.py来启动服务器。然后，如果在浏览器中访问*http://127.0.0.1:8080/*时，经过短暂的停顿显示出了“Hello”，那么说明一切正常。

### 使用 XHR 方法返回的 Deferred 对象

在设置好CherryPy之后，请把下面的代码保存为*foo.html*文件并放在已经运行的*hello.py*旁边。然后，就可以在浏览器中输入 *http://127.0.0.1:8080/foo.html* 并查看结果了：

```
<html>
    <head>
    <title>Fun with Deferreds!</title>
    <script type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        djConfig = "isDebug: true">
    </script>

    <script type="text/javascript">
        dojo.addOnLoad(function() {

            //启动一次异步请求，该请求返回一个Deferred对象
            var d = dojo.xhrGet({
               url: "http://localhost:8080",
               timeout : 5000,
               load : function(response, ioArgs) {
                 console.log("Load response is:", response);
                 console.log("Executing the callback chain now...");
                 return response;
               },
               error : function(response, ioArgs) {
                 console.log("Error!", response);
                 console.log("Executing the errback chain now...");
                 return response;
               }
            });

            console.log("xhrGet fired. Waiting on callbacks or errbacks");

            //添加几个回调函数
            d.addCallback(
              function(result) {
                 console.log("Callback 1 says that the result is ", result);
                 return result;
              }
            );

            d.addCallback(
              function (result) {
                 console.log("Callback 2 says that the result is ", result);
```

```
            return result;
          }
        );

        //添加几个错误处理函数
        d.addErrback(
          function(result) {
            console.log("Errback 1 says that the result is ", result);
            return result;
          }
        );

        d.addErrback(
                function(result) {
                  console.log("Errback 2 says that the result is ", result);
                  return result;
                }
              );
          });
      </script>
    </head>
    <body>
    Check the Firebug console.
    </body>
</html>
```

运行上面这个示例之后，读者应该看到Firebug控制台中显示的下列输出结果：

```
xhrGet fired. Waiting on callbacks or errbacks
Load response is: Hello
Executing the callback chain now...
Callback 1 says that the result is Hello
Callback 2 says that the result is Hello
```

通过理解这个示例，我们最大的收获就是体验到了Deferred为开发人员编写与dojo.xhrGet返回的数据进行交互的代码，提供了清晰、一致的接口，无论是处理成功的响应，还是处理发生的错误。

如果读者想看到触发错误处理函数链的结果，可以将timeout值设置为小于3秒，然后就能看到处理错误后的输出。另外，要是某个回调函数中存在错误，也将导致错误处理函数链被触发。因此，可以在某个回调函数中故意制造一处错误，然后观察一下在触发错误回调函数链之前执行部分回调函数链的结果。

---

**警告：** 千万记住要将传入回调函数和错误处理函数中的值返回，以保证函数链的正常执行。万一因为疏忽没有在某个回调函数中返回该值，就会由于中断回调函数或错误处理函数链的执行而导致怪异行为。现在，读者应该也知道了在XHR方法的load和error处理函数中同样必须返回响应数据的重要性了。

---

图4-2展示了Deferred处理事件的基本流程。其中的关键就是要把Deferred想像成一条事件链。

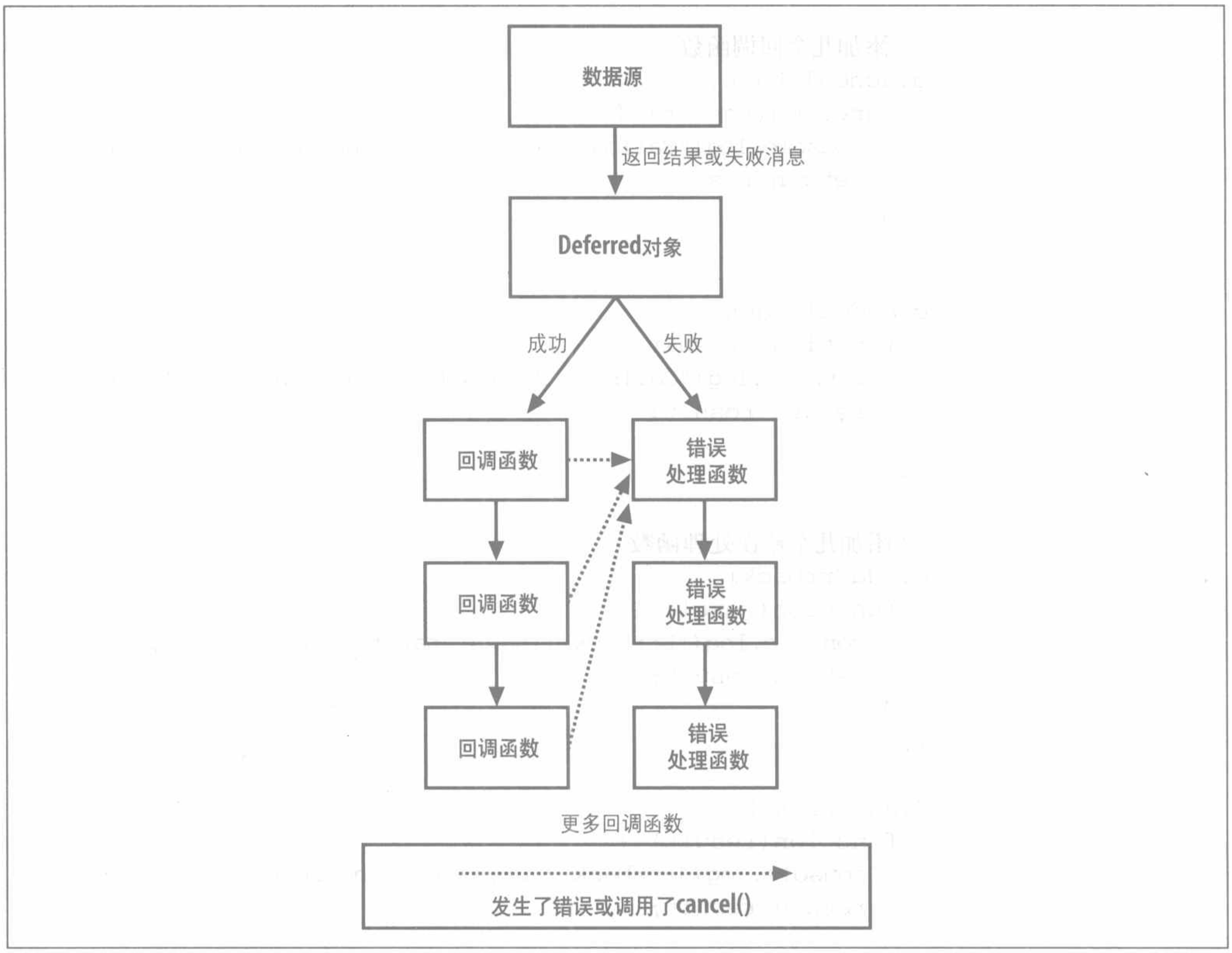



图 4-2：Deferred 处理事件的基本流程

## 向 XHR 方法中注入 Deferred

Deferred提供的另一个有用特性是，它可以让开发人员在异步操作完成之前明确地取消它。下面这个示例通过修改前面的示例代码展示了如何取消正在执行的请求，以及如何将一个 Deferred 对象“注入”到该请求的 load 和 error 处理函数中：

```
<html>
    <head>
        <title>Fun with Deferreds!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig = "isDebug: true">
        </script>
```

```
<script type="text/javascript">
    dojo.addOnLoad(function() {

        var d = new dojo.Deferred;

            //添加几个回调函数
            d.addCallback(
              function(result) {
                console.log("Callback 1 says that the result is ", result);
                return result;
              }
            );

            d.addCallback(
              function (result) {
                console.log("Callback 2 says that the result is ", result);
                return result;
              }
            );

            //添加几个错误处理函数
            d.addErrback(
              function(result) {
                console.log("Errback 1 says that the result is ", result);
                return result;
              }
            );

            d.addErrback(
              function(result) {
                console.log("Errback 2 says that the result is ", result);
                return result;
              }
            );

            //启动一次异步请求，该请求返回一个Deferred对象
            request = dojo.xhrGet({
               url: "http://localhost:8080",
               timeout : 5000,
               load : function(response, ioArgs) {
                  console.log("Load response is:", response);
                  console.log("Executing the callback chain now...");

                  //注入Deferred的回调函数链
                  d.callback(response, ioArgs);

                  //保证dojo.xhrGet的Deferred事件链能够继续执行下去……
                  return response;
               },
               error : function(response, ioArgs) {
                 console.log("Error!", response);
                 console.log("Executing the errback chain now...");
```

```
                    //注入 Deferred 的错误处理函数链
                    d.errback(response, ioArgs);

                    // 保证 dojo.xhrGet 的 Deferred 事件链能够继续执行下去……
                    return response;
                }
            });
        });
    </script>
    </head>
    <body>
        XHR request in progress. You have about 3 seconds to cancel it.
        <button onclick="javascript:request.cancel()">Cancel</button>
    </body>
    </html>
```

运行上面这个示例后，读者应该看到下列输出结果：

```
xhrGet just fired. Waiting on callbacks or errbacks now...
Load response is: Hello
Executing the callback chain now...
Callback 1 says that the result is Hello
Callback 2 says that the result is Hello
```

而如果在 3 秒钟前按下了“Cancel”按钮，则会导致下列结果：

```
xhrGet just fired. Waiting on callbacks or errbacks now...
Press the button to cancel...
Error: xhr cancelled dojoType=cancel message=xhr cancelleddojo.xd.js (line 20)
Error! Error: xhr cancelled dojoType=cancel message=xhr cancelled
Executing the errback chain now...
Errback 1 says that the result is Error: xhr cancelled dojoType=cancel message=xhr
cancelled
Errback 2 says that the result is Error: xhr cancelled dojoType=cancel message=xhr
cancelled
```

### 自定义取消函数

每个 XHR 方法都有一个在调用 cancel()时执行的取消函数。但如果读者希望自定义 Deferred 对象，也可以像下面这样创建符合自己需要的取消函数：

```
var canceller = function() {
    console.log("custom canceller...");
    // 如果不在这里返回一个自定义的 Error 对象，则会返回默认的“Deferred
    //Cancelled”错误
}
var d = new dojo.Deferred(canceller); // 将取消函数传入构造函数中
/* ……执行具体操作的代码…… */
d.cancel(); // 错误处理函数将按照自定义的方式响应“Deferred Cancelled”错误
```

### DeferredList

虽然Deferred是由Base内置的，但Core为了方便开发人员管理多个Deferred对象又提供了DeferredList。使用DeferredList的常见情形包括：

- 当一组Deferred对象中的全部回调函数都被触发时，再触发某个特定的回调函数或回调函数链。
- 当一组Deferred对象中的至少一个回调函数被触发时，再触发某个特定的回调函数或回调函数链。
- 当一组Deferred对象中的至少一个错误处理函数被触发时，再触发某个特定的错误处理函数或错误处理函数链。

DeferredList的API如下所示：

```
dojo.DeferredList(/*Array*/list, /*Boolean?*/fireOnOneCallback, /*Boolean?*/
    fireOnOneErrback,  /*Boolean?*/consumeErrors, /*Function?*/canceller)
```

根据以上签名很容易知道，如果在调用构造函数时只传递一个参数，那么该参数就是一个Deferred对象的数组，而默认行为就是当所有Deferred对象的回调函数链均被触发时，再触发相应的回调函数链；如果传递布尔值参数，那么可以分别控制当Deferred对象中至少一个回调函数或错误处理函数被触发时，是否再触发特定的回调函数或错误处理函数链。

如果将consumeErrros参数设置为true，那么DeferredList就会消除错误，以防数组中的个别Deferred对象直接将错误暴露出来。最后，canceller参数提供了一种传入自定义取消函数的方式，与在普通的Deferred对象中使用取消函数一样。

## 表单和HTTP实用程序

如果设计得当，某些Ajax应用程序的确会令人惊叹。不过，也别忘记那些久经考验的HTML元素，例如表单就远远谈不上过时，而且仍然在许多（有或没有加入Ajax特性的）现代设计中担当着重要角色。Base为转换表单数据提供了3个方法：

```
dojo.formToObject(/*DOMNode||String*/ formNode) //返回对象
dojo.formToQuery(/*DOMNode||String*/ formNode) //返回字符串
dojo.formToJson(/*DOMNode||String*/ formNode) //返回字符串
```

为了示范这几个方法的应用，假设有如下表单：

```
<form id="register">

    <input type="text" name="first" value="Foo">
```

```
        <input type="button" name="middle" value="Baz" disabled>
        <input type="text" name="last" value="Bar">
        <select type="select" multiple name="favorites" size="5">
            <option value="red">red</option>
            <option value="green" selected>green</option>
            <option value="blue" selected>blue</option>
        </select>

    </form>
```

下面就是运行每个方法后的结果。注意，在转换表单的过程中，这些方法跳过了被禁用的元素。

formToObject 产生：

```
{
    first: "Foo",
    last : "Bar",
    favorites: [
        "green",
        "blue"
    ]
};
```

formToQuery 产生：

```
"first=Foo&last=Bar&favorites=green&favorites=blue"
```

formToJson 产生：

```
'{"first": "Foo", "last": "Bar", "favorites": ["green", "blue"]}'
```

此外，Base 还提供了下列辅助方法，以便开发人员在查询字符串与 JavaScript 对象之间进行转换。虽然它们的用途显而易见，但需要提醒读者的是，查询字符串中的值一律会转换为字符串，即使是数字值：

```
dojo.queryToObject(/*String*/ str) //返回对象
dojo.objectToQuery(/*Object*/ map) //返回字符串
```

请参考如下简单示例：

```
//产生{foo : "1", bar : "2", baz : "3"}
var o = dojo.queryToObject("foo=1&bar=2&baz=3");

//再转换回 foo=1&bar=2&baz=3
dojo.objectToQuery(o);
```

# 使用 JSONP 实现跨站点脚本

虽然出于安全考虑，浏览器不允许JavaScript的XMLHttpRequest对象从页面当前所在域的外部加载数据，但SCRIPT标签却不会受到“同源”策略的限制。因此，基于SCRIPT标签的跨域加载数据的非正式标准 —— JSONP 诞生了。使用JSONP技术，开发人员可以从任何服务器获取数据，并将这些数据集成到同一个应用程序中（也称为Mashup应用程序）（注2）。

## JSONP 入门

同其他新技术一样，JSONP也是乍一听起来显得很神秘，而在读者理解它之后，就会发现其实它非常简单。为了理解这个概念，请读者想像一个动态创建的SCRIPT标签，该标签被添加到来源于 *http://oreilly.com* 的一个页面的HEAD标签中。关键的地方在于SCRIPT标签的src属性，因为该属性并不一定要从oreilly.com域中加载数据，而是可以从任何域中加载数据，例如，*http://example.com?id=23*。使用JavaScript实现上述操作的代码很简单：

```
e = document.createElement("SCRIPT");
e.src="http://example.com?id=23";
e.type="text/javascript";
document.getElementsByTagName("HEAD")[0].appendChild(e);
```

虽然SCRIPT标签通常的用途是加载脚本，但实际上可以使用它来加载任何数据，包括JSONP对象。现在的问题是，加载并将JSONP对象添加到页面的HEAD部分并没有什么意义（除了可能会破坏页面的外观之外）。

例如，可能会得到如下代码所示的结果，其中加粗的部分是由前面动态添加到页面HEAD部分的JavasScript脚本产生的输出：

```
<html>
  <head>
    <title>My Page</title>
    <script type="text/javascript" >
      {foo : "bar"}
    </script>
  </head>
  <body>
Some page content.
  </body>
</html>
```

---

注2： 无须借助任何外部插件，JSONP本身完全可以跨域加载数据。不过，像Flash和ActiveX之类的插件也有基于浏览器本身绕过“同源”限制的手段。

当然，显示JavaScript直接量的确没有多大用处，可是，假如能够取得以JSON数据作为参数的函数调用 —— 也就是说，页面中的某个地方已经定义了该函数，又会怎样呢？事实上，最终的结果可以说意义非凡，因为这样一来，就可以通过异步方式随时从外部获取数据，并在数据返回时立即将其传入到处理它的函数中。为此，所有问题都可以归结到一点，即让插入的SCRIPT标签返回以JSON数据作为参数的一个函数调用，如myCallback({foo:"bar"})而不是{foo:"bar"}。如果在SCRIPT标签完成加载之前，页面中已经存在myCallback函数的定义，那么该函数就可以将返回的JSON数据作为参数并立即执行，而该函数就是我们的回调函数。（如果读者对上述解释还不太理解，有必要在此多花点时间把它搞明白。）

不过，还有一个问题没有解决：怎么把JSON对象放到一个能够被触发的回调函数中呢？很简单，*example.com*的开发人员只要为你提供一个额外的查询参数，并允许你通过该参数定义一个要将JSON对象放入其中的函数即可。假设他们要求你通过参数c来传入你的回调函数（即需要一次新的请求将参数c发送到服务器），那么调用*http://example.com?id=23&c=myCallback*就可以返回myCallback({foo:"bar"})了。而这些就是JSONP的全部内涵。

## 核心IO

本节介绍Core提供的dojo.io模块，开发人员使用该模块可以动态注入SCRIPT标签来取得参数式JSON，还可以利用IFRAME实现不同的数据传输任务。

### 在Dojo中使用JSONP

相信读者基于到目前为止对Dojo的了解，一定不会因为它实现的JSONP如此简单而感到惊讶。为完成上一节介绍的功能，可以使用dojo.io.script.get方法，它能够接收的参数与XHR方法相同。但是，handleAs参数不适合JSONP，而应该代之以callbackParamName参数，以便Dojo按照我们的意愿设置和管理回调函数。

请看下面这个示例：

```
//dojo.io.script 不属于 Base，因此必须使用 dojo.require("dojo.io.script");
将其导入页面中
dojo.io.script.get({

callbackParamName : "c", // 由 jsonp 服务指定
url: "http://example.com?id=23",
load : function(response, ioArgs){
  console.log(response);
  return response;
},
```

```
    error : function(response, ioArgs) {
      console.log(response);
      return response;
    }
  });
```

注意，callbackParamName指定的是查询字符串中参数的名称，该名称由*example.com*规定。也就是说，它不是我们作为回调函数定义的函数名。在后台，Dojo会创建一个临时函数并将其与load函数连接，这一点遵循了与其他XHR方法相同的约定，即Dojo会替我们实现数据的转移，而我们只需在load函数中操作返回的数据即可。

---

**警告：** 如果未指定callbackParamName参数，或者指定的参数值不正确，那么就会导致"<some callback function> does not exist"之类的JavaScript错误，原因是动态生成的SCRIPT标签会执行一个并不存在的函数。

---

### 连接到 Flickr 数据源

下面这个示例展示了如何对Flickr数据源生成JSONP调用。读者可以在Firebug中运行该示例并查看结果。建议读者试一试如果不提供callbackParamName参数（或将参数值拼错），会导致什么样的错误：

```
dojo.require("dojo.io.script");
dojo.addOnLoad(function(){
 dojo.io.script.get({
    callbackParamName : "jsoncallback", //由Flickr规定
    url: "http://www.flickr.com/services/feeds/photos_public.gne",
    content : {format : "json"},
    load : function(response, ioArgs) {
        console.log(response);
        return response;
    },
    error : function(response, ioArgs) {
        console.log("error");
        console.log(response);
        return response;
    }
 });
});
```

### 通过 JSONP 调用取得 JavaScript 代码

实际上，使用dojo.io.script.get也可以从服务器上取得纯JavaScript代码。此时，执行请求的方式不变，但应该记住将callbackParamName参数替换成checkString

(检查字符串)值。这里的“检查字符串”是一种机制，它允许 Dojo 检查响应是否已经传输完成。从本质上讲，如果对提供的检查字符串执行 typeof 检查，而结果没有返回 undefined，那么就可以假定 JavaScript 已经加载完成。(换句话说，这是一种 hack。)假设我们有如下简单的 CherryPy 脚本，那么就可以使用一个 checkString 值 o 来表示脚本是否成功加载。因为 o 是我们希望通过 JSONP 调用取得的变量(即如果 typeof(o) != undefined，就可以假设调用完成了)。

首先，看一看提供 JavaScript 的 CherryPy 脚本：

```
import cherrypy
class Content:
    @cherrypy.expose
    def index(self):
        return "var o = {a : 1, b:2}"

cherrypy.quickstart(Content())
```

如果读者已经在 8080 端口启动了 CherryPy，那么可以使用下面的 Dojo 代码取得它提供的 JavaScript：

```
dojo.require("dojo.io.script");
dojo.addOnLoad(function(){
 dojo.io.script.get({
   checkString : "o",
   timeout : 2000,
   url : "http://localhost:8080",
   load : function(response, ioArgs) {
       console.log(o);
       console.log(response)
   },
   error : function(response, ioArgs) {
       console.log("error", response, ioArgs);
       return response;
   }
 });
});
```

---

**注意：** 注意，dojo.io.script.get 会在内部检查 checkString 或 callbackParamName 参数，根据存在哪个参数来确定用户想要加载的是 JavaScript 还是 JSON。

---

## 利用 IFRAME 实现数据传输

Core 提供的 IFRAME 传输机制可以用来方便地实现通常需要页面刷新才能实现的任务。

虽然XHR方法也可以在后台获取数据，但却不支持某些操作。例如，通过表单提交上传文件和下载文件就是IFRAME传输的两个强项。

与其他IO系统遵循的模式相同，使用IFRAME传输也要求传递一个包含关键字的对象参数，而且也会返回Deferred。IFRAME传输允许使用GET或POST这两种HTTP方法，以及一些handleAs参数。事实上，可以根据表4-4的说明提供下列任意参数。

表4-4：IFRAME传输的关键字参数

| 名称 | 类型（默认值） | 说明 |
|---|---|---|
| method | 字符串 （"POST"） | 要使用的HTTP方法。有效值为GET或POST |
| handleAs | 字符串 （"text"） | 指定提供给load或handle回调函数的响应数据格式。有效值包括"text"、"html"、"javascript"和"json"。对于除"html"之外的格式，服务器会响应一个HTML文件，并把该响应包含在一个textarea元素中 |
| content | 对象 | 如果form是另外一个参数，那么content对象就被看成隐藏的表单元素。如果未指定form参数，那么content对象会通过dojo.objectToQuery()被转换成查询字符串 |

**注意：** 从Dojo1.2开始，XML也将通过IFRAME传输方式进行处理。

## 通过IFRAME下载文件

毕竟，通过IFRAME下载文件是最常见的操作，因此我们首先来试一试。以下CherryPy代码会在用户访问*http://localhost:8080/*时提供一个本地文件下载。而为了向服务器发送请求，我们将在dojo.io.iframe.send中使用同样的URL：

```
import cherrypy
from cherrypy.lib.static import serve_file
import os

# 读者需要将下面的路径修改为自己机器中的绝对路径
local_file_path="/tmp/foo.html"

class Content:

    #serve up a file...
    @cherrypy.expose
    def download(self):
```

```
            return serve_file(local_file_path, "application/x-download", "attachment")
    # 启动Web服务器并令其侦听8080端口
    cherrypy.quickstart(Content(), '/')
```

读者需要新建一个文件（foo.html），并将其放在本地硬盘上，同时更新CherryPy脚本中的路径以指向该文件。然后，通过浏览器打开下面利用IFRAME的HTML页面并单击“Download!”按钮，就应该看到下载对话框了（译注1）。

**注意：** 在第一次调用dojo.io.iframe.send时，可能会在瞬间看到IFRAME被创建又随即消失。解决这一问题的常见手段是在页面加载时发送一个空请求，因为此时创建IFRAME不易被觉察。然后，当应用程序发送真正的请求时，就看不到画面闪烁了。

```
<html>
    <head>
        <title>Fun with IFRAME Transports!</title>

        <script type="text/javascript"
          src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
          djConfig="isDebug:true,dojoBlankHtmlUrl:'/path/to/blank.html'">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.io.iframe");

            dojo.addOnLoad(function() {
                download = function() {
                    dojo.io.iframe.send({
                        url : "http://localhost:8080/download/"
                    });
                };
            });
        </script>
    </head>
    <body>
        <button onclick="javascript:download()">Download!</button>
    </body>
</html>
```

译注1：请读者在试验本示例时注意两个问题：一是由于上面CherryPy代码中没有提供本地HTML文件，因此需要把下面的HTML文件保存到*http://localhost*域中再通过浏览器打开，即不能直接打开*http://localhost:8080/foo.html*；二是在使用跨域Dojo构建时，应该将*dojo/resources/blank.html*保存在本地域中，并设置djConfig.dojoBlankHtmlUrl参数指向该*blank.html*文件，也就是要将下面代码中的*/path/to/blank.html*修改为正确的路径。

**警告：** 为了能够多次使用“Download!”按钮，应该为dojo.io.iframe.send提供一个timeout值，以便在超时后停止前一次请求，从而让其他请求得以继续。

## 通过IFRAME提交表单

另一个使用IFRAME的常见情形就是在后台提交表单——也许是一个涉及文件上传的表单，因为正常的表单提交必须刷新页面才行。以下是处理文件上传的CherryPy脚本：

```
import cherrypy

# 读者可以在此设置用于保存上传文件的路径
local_file_path="/tmp/uploaded_file"

class Content:

    #serve up a file...
    @cherrypy.expose
    def upload(self, inbound):
          outfile = open(local_file_path, 'wb')
          inbound.file.seek(0)
          while True:
              data = inbound.file.read(8192)
              if not data:
                  break
              outfile.write(data)
          outfile.close()

          # 返回一个简单的HTML文件作为响应
          return "<html><head></head><body>Thanks!</body></html>"
# 启动Web服务器并令其侦听8080端口
cherrypy.quickstart(Content(), '/')
```

下面是执行文件上传操作的HTML页面。如果读者体验一下这个示例，会发现通过IFRAME上传文件是在后台发生的，无须重载页面；而如果使用表单本身的提交按钮来POST同样的数据，那么会导致页面重载。这里提请读者格外注意的是，我们设置了handleAs参数，要求服务器提供HTML响应。

```
<html>
    <head>
        <title>Fun with IFRAME Transports!</title>

        <script type="text/javascript"
          src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
          djConfig="isDebug:true,dojoBlankHtmlUrl:'/path/to/blank.html'">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.io.iframe");
```

```
dojo.addOnLoad(function() {
    upload = function() {
    dojo.io.iframe.send({
        form : "foo",
        handleAs : "html", //服务器的响应类型
        url : "http://localhost:8080/upload/",
        load : function(response, ioArgs) {
            console.log(response, ioArgs);
            return response;
        },
        error : function(response, ioArgs) {
            console.log("error");
            console.log(response, ioArgs);
            return response;
        }
    });
};
});
</script>
</head>
<body>
<form id="foo" action="http://localhost:8080/upload/" method="post"
enctype="multipart/form-data">
    <label for="file">Filename:</label>
    <input type="file" name="inbound">
    <br />
    <input type="submit" value="Submit Via The Form">
</form>

<button onclick="javascript:upload();">Submit Via the IFRAME Transport
</button>
</body>
</html>
```

下一节，我们讨论如何取得非 HTML 类型的响应。

### 取得非 HTML 类型的响应数据

在前面的示例中，服务器返回的是可以直接操作的HTML文档。但是，对于非HTML数据类型，则必须满足一个特殊的条件，即要把响应的数据包含在一个 `textarea` 标签中。毕竟，在这种传输方式下，HTML 文档是判定响应已经加载完成的唯一可靠的、跨浏览器的方式，而 `textarea` 又是传输基于文本内容的自然选择。当然，Dojo 会在内部将数据内容提取出来并设置为响应。下面这个示例基于前面的示例修改而成，它示范了如何取得纯文本数据，而非 HTML 格式。

**注意：** 注意，虽然前面上传和下载文件的示例不需要CherryPy提供本地HTML文件，但下面这个示例则需要。原因是这里的IFRAME传输涉及到了访问页面的DOM来提取内容，而访问DOM则属于跨站点脚本操作（前面的示例未涉及到任何DOM操作）。

对CherryPy脚本的修改主要有两处：一处是添加配置参数，以便CherryPy提供*foo.html*文件；另一处是将最后的响应包含在一个textareat元素中（译注2）。修改后的结果如下所示：

```
import cherrypy
import os
# foo.html 文件包含执行 XHR 请求的 Dojo 代码
# 而这是通过下面的 config 指令设置的

current_dir = os.getcwd()
config = {'/foo.html' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}

local_file_path="/tmp/uploaded_file"

class Content:

    #serve up a file...
    @cherrypy.expose
    def upload(self, inbound):
          outfile = open(local_file_path, 'wb')
          inbound.file.seek(0)
          while True:
              data = inbound.file.read(8192)
              if not data:
                  break
              outfile.write(data)
          outfile.close()
          return
"<html><head></head><body><textarea>Thanks!</textarea></body></html>"

# 启动 Web 服务器并令其侦听 8080 端口
cherrypy.quickstart(Content(), '/', config=config)
```

对于请求代码的修改，则只涉及handleAs参数：

```
dojo.io.iframe.send({
    form : dojo.byId("foo"),
```

译注2： 最后，在启动服务器时，还必须传递config参数。

```
        handleAs : "text", //服务器的响应类型
        url : "http://localhost:8080/upload/",
        load : function(response, ioArgs) {
            console.log(response, ioArgs); //响应结果为Thanks!
            return response;
        },
        error : function(response, ioArgs) {
            console.log("error");
            console.log(response, ioArgs);
            return response;
        }
    });
```

### 手工创建隐藏的IFRAME

在页面中创建一个隐藏的IFRAME来加载内容，然后在内容加载完成后得到通知也是一种需求。此时，不能再使用发送内容的dojo.io.iframe.send方法了，而是要使用dojo.io.iframe.create方法。为dojo.io.iframe.create方法传递一段JavaScript代码，即可在创建IFRAME的同时执行该代码。dojo.io.iframe.create方法的API如下：

```
dojo.io.iframe.create(/*String*/frameName,/*String*/onLoadString, /*String?*/url)
//返回DOMNode
```

这个方法接收的第一个参数是框架(frame)的名称，第二个参数是一个将作为回调函数被求值的字符串，第三参数是一个可选的URL，用于在框架中加载页面。下面就是一个在隐藏IFRAME中加载URL，并在加载完毕后执行回调函数的示例：

```
<html>
    <head>
    <title>Fun with IFRAME Transports!</title>

    <script type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
         djConfig="isDebug:true,dojoBlankHtmlUrl:'/path/to/blank.html'">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.io.iframe");

        function customCallback() {
            console.log("callback!");

            //可以通过dojo.byId("fooFrame")引用iframe
        }

        create = function() {
            dojo.io.iframe.create("fooFrame", "customCallback()",
              "http://www.exmaple.com");
        }
```

```
        </script>
    </head>
    <body>
        <button onclick="javascript:create();">Create</button>
    </body>
</html>
```

**警告：** 有些网页中包含跳脱框架的JavaScript函数（译注3），加载这种网页会导致前面的示例用法失效。

虽然在IFRAME中直接加载内容的情况比较多，但有时候也可能需要创建空框架。此时，如果使用本地Dojo文件，只要在调用dojo.io.iframe.create时省略第三个参数，就可以创建一个空框架；如果使用跨域Dojo构建，那么需要指定一个为框架提供内容的本地模板文件。在本地Dojo安装目录中有一个模板文件，路径为*dojo/resources/blank.html*，可以将这个文件复制到合适的地方。而且，在创建IFRAME之前，还必须将该模板文件的路径指定给配置参数djConfig.dojoBlankHtmlUrl。

**注意：** 除了Core所提供的IO工具之外，DojoX通过dojox.io模块也提供了IO工具。其中，包括针对XHR分步（multipart）请求的实用程序和针对代理的辅助方法。

## JSON远程过程调用

读者可能会注意到，即使通过Dojo的各种XHR方法（如dojo.xhrGet）来编写代码，由于必须重复为调用提供内容以及编写load回调函数，因此仍然存在导致冗余和错误的可能。所幸的是，我们可以使用Core的dojo.rpc模块提供的RPC（Remote Procedure Call，远程过程调用）机制解决这个问题。要使用该模块的方法，首先需要通过SMD（Simple Method Description，简单方法描述）提供相关的配置信息，然后再使用定义的服务而不是xhrGet等方法。如果应用程序与服务器之间采用了标准的交互方式，而且响应的错误处理方式也十分类似，那么使用dojo.rpc模块就可以获得清晰的设计并把出错的可能性降至最低。

目前，Core提供一个JsonService和一个JsonpService，它们都派生自RpcService基类。

**注意：** dojox.rpc也提供了增强的RPC功能，其中一些功能有望迁移到Core中。

译注3： 这里应该是指包含top.location = self.location代码的函数。

## JSON RPC示例

为了示范RPC机制的用法，下面我们举一个使用JsonService来处理一组数字的例子。这个例子可以计算一组数字的和以及每个数字平方的和。客户端代码中包括提供sum和sumOfSquares两个方法的SMD定义，这两个方法都接收一组数字作为参数：

```
<html>
    <head>
        <title>Fun with JSON RPC!</title>

        <script type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true">
    </script>
    <script type="text/javascript">
        dojo.require("dojo.rpc.JsonService");
        dojo.addOnLoad(function() {

            //将SMD作为一个对象直接量来创建……
            var o = {
                "serviceType": "JSON-RPC",
                "serviceURL": "/",
                "methods":[
                    {
                        "name": "sum",
                        "parameters":[{name : "list"}]
                    },
                    {
                        "name": "sumOfSquares",
                       "parameters":[{name : "list"}]
                    }
                ]
            }

            //实例化服务
            var rpcObject = new dojo.rpc.JsonService(o);

            //调用服务并使用其返回的Deferred对象添加一个回调函数
            var sum = rpcObject.sum([4,8,15,16,23,42]);
            sum.addCallback(function(response) {
                console.log("the answer is ", response);
            });
            //添加更多的回调函数或错误处理函数……

            //以相同方式调用sumOfSquares……
        });
    </script>
    <body>
    </body>
</html>
```

读者应该能够理解，如果有很多方法都以非常标准的方式与服务器通信，那么调用RPC客户端就能极大简化原先混乱的设计。而且，使用dojo.rpc.JsonService返回的Deferred对象，还可以任意添加回调函数和错误处理函数。

这个例子的服务器端代码如下所示。为了简单起见，该脚本中没有引入JSON处理库；不过，在比这个例子更复杂的情况下，当然应该优先使用JSON处理库：

```
import cherrypy
import os
# foo.html 文件包含执行 XHR 请求的 Dojo 代码 # 而这是通过下面的 config 指令设置的

current_dir = os.getcwd()
config = {'/foo.html' :
    {
    'tools.staticfile.on' : True,
    'tools.staticfile.filename' : os.path.join(current_dir, 'foo.html')
    }
}

class Content:

    @cherrypy.expose
    def index(self):
        ##########################################################
        # 为简单起见，本示例没有使用任何 json 库。在开发比这个示例更加
        # 复杂的应用时，最好从 http://json.org 下载一个适用的 json 库
        ##########################################################

        # 读取原始 POST 数据
        rawPost = cherrypy.request.body.read()

        # 转换为对象
        obj = eval(rawPost) # 重大安全漏洞，千万注意……

        # 处理数据
        if obj["method"] == "sum":
            result = sum(obj["params"][0])
        if obj["method"] == "sumOfSquares":
            result = sum([i*i for i in obj["params"][0]])

        # 返回 json 格式的响应
        return str({"result" : result})

# 启动 Web 服务器并令其侦听 8080 端口
cherrypy.quickstart(Content(), '/', config=config)
```

使用JsonpService与使用JsonService非常相似。在你的Dojo本地安装目录中，有一个针对Yahoo!服务的SMD示例，路径为*dojox/rpc/SMDLibrary/yahoo.smd*，读者可以抽时间自己试验一下。

## OpenAjax Hub

OpenAjax Alliance (*http://www.openajax.org/*) 是一个由厂商和相关组织共同发起成立的联盟，致力于开发基于 Ajax 的 Web 互操作技术。当前 Web 开发领域中的一个重要问题，就是如何在一个应用程序中使用多个 JavaScript 库。虽然 Dojo 及其他框架都针对互操作性提出了应用方案（例如，保护全局命名空间），但事实上，同时使用两个库并让它们真正实现互操作，仍然是个不小的挑战。无论是实际地来回传递数据，还是整体编程风格以及学习曲线，都不容乐观。

OpenAjax Alliance 针对库与库之间如何交互提出了 OpenAjax Hub 规范。该规范为实现互操作而提倡的基本技术就是松散耦合的发布/预计模型。最终，Core 在 `OpenAjax` 模块中实现了该规范，并通过全局 `OpenAjax` 对象提供了下列方法：

- `RegisterLibrary`。
- `UnregisterLibrary`。
- `Publish`。
- `Subscribe`。
- `Unsubscribe`。

作为开放标准的支持者，Dojo 将继续关注 OpenAjax Hub 规范的最新发展，该规范的地址为 *http://www.openajax.org/member/wiki/OpenAjax_Hub_Specification*。

## 小结

在学习了本章之后，读者应该能够：

- 通过 Dojo 的 XHR 机制对 Web 服务器执行 REST 风格的操作。
- 理解 `Deferred` 如何模拟线程，即使 JavaScript 不支持线程。
- 知道工具箱的整个 IO 子系统的用途，以及方法调用通常都返回 Deferred。
- 使用 Base 提供的实用程序在表单与对象和 JSON 之间互相转换。
- 能够使用 Core 的 IFRAME 传输机制实现常见操作，例如，上传和下载文件。
- 理解 RPC 机制如何简化应用程序逻辑并产生更容易维护的设计。
- 知道 Core 为实现 OpenAjax Hub 规范提供的方法。

下一章，我们将介绍节点操作。

# 第5章

# 节点操作

本章讨论用于操作DOM节点的简洁而高效的query、behavior和NodeList。其中，查询DOM使用query的选择符语法，从HTML标记中分离事件和操作使用Core的behavior模块，而连缀（chain）一系列操作则要使用NodeList提供的语法糖。

## 莫衷一是的查询

在撰写本章时，发生了很多与查询DOM节点有关的事件。其中比较重要的包括：

- W3C在2007年年底更新了其Selectors API工作草案（*http://www.w3.org/TR/selectors-api/#documentselector*）；
- WebKit宣布已经在本地实现了Selectors API的关键特性querySelector和querySelectorAll（*http://webkit.org/blog/156/queryselector-and-queryselectorall/*）；
- Firefox3在本地实现了Web开发人员众望所归的getElementsByClassName（*http://ejohn.org/blog/getelementsbyclassname-in-firefox-3/*）。

这样说来，本章介绍的实用方法仍然会在未来的几年中，继续在Dojo工具箱及你自己的工具箱中扮演重要角色。即使是对Selectors API的本地支持，也将不可避免地存在不一致性问题，因而需要继续加以解决，况且某些浏览器很可能一开始只会部分地支持该标准。

在所有浏览器都一致地实现Selectors API之前，Dojo都将一如既往地通过本地实现或在缺乏本地实现时模拟该实现，为开发易移植和最优化的代码提供良好的保障。

# query：以不变应万变

在编写JavaScript代码时，我们总会按照某些标准在DOM中查询符合条件的节点。假如只需按照标签名来查询节点，那么只要使用document.getElementsByTagName就行了。但是，如果需要按照类、按照某个特殊属性，或者按照它们的组合条件来查询节点，那么恐怕你就要禁不住发问了：为什么没有现成的getElementsByClass函数呢？显然，所有人都会提出这个问题，而有的人则开发了自己的版本，其中有的版本相对而言更成功一些。

虽然Dojo曾在早期版本中专门实现了getElementsByClass方法，但是现在该工具箱提供了一个统一的方法，供开发人员使用CSS选择符语法来查询DOM节点。为了说明DOM查询的必要性，我们先来看一个实现getElementsByClass函数（极为普遍的需求）的英勇尝试：

```
// 按照类名查询元素，可以选择从指定的父节点开始查找
function getElementsByClassName(/*String*/className, /*DOMNode?*/node) {
  var regex = new RegExp('(^| )' + className + '( |$)');
  var node = node||document.body;
  var elements = node.getElementsByTagName("*");
  var results = [];

  for (var i=0; i < elements.length; i++) {
    if (regex.test(elements[i].className)) {
      results.push(elements[i]);
    }
  }
  return results;
}
```

虽然这个函数只有12行代码，但是仍然也需要一行一行地输入、调试和维护。如果我们又想增加按标签和类查询的功能，那么就必须增加一个参数以表示标签名，并将该参数传入getElementsByTagName函数中。如果还想再增加其他功能，同样还必须修改并维护这些代码。此外，上面这个函数可能还无法保证在所有浏览器中都可以正常使用，况且其中的正则表达式也不是很直观明了。

所幸的是，dojo.query可以让我们把这个自定义查询函数彻底地扔到垃圾筒里面去了。以下就是该方法提供的统一查询API：

```
dojo.query(/*String*/ query, /*String?|DOMNode?*/ root) // 返回NodeList
```

---

**注意：** 尽管我们不会在此过多介绍NodeList，但目前读者真正需要知道的只有NodeList是Array的一个子类，而该类针对节点操作进行了专门的扩展。

---

要通过query实现前面getElementsByClassName函数的功能，只需将一个CSS选择符传入其中即可：

```
dojo.query(".someClassName")
```

查询一个带有某个类名的标签，如DIV，同样也很简单，只要遵照CSS选择符语法更换参数就行了：

```
dojo.query("div.someClass")
```

想继续欣赏使用统一的语法查询DOM的美妙例子吗？当然，看得越多你就会越喜欢。不过，读者最好先浏览一下表5-1，对使用query能够完成的各种操作有一个大致的印象。另外，更详细的相关信息，请参考*http://www.w3.org/TR/css3-selectors/*。

表5-1：常用的CSS选择符语法

| 语法 | 含义 | 示例 |
| --- | --- | --- |
| * | 任何元素 | dojo.query("*") |
| E | 标签为E的元素 | dojo.query("div") |
| .C | 带有类C的元素 | dojo.query(".baz") |
| E.C | 标签为E且带有类C的元素 | dojo.query("div.baz") |
| #ID | ID值为ID的元素 | dojo.query("#quux") |
| E#ID | 标签为E且ID值为ID的元素 | dojo.query("div#quux) |
| [A] | 带有属性A的元素 | dojo.query("[foo]") |
| E[A] | 标签为E且带有属性A的元素 | dojo.query("div[foo]") |
| [A="V"] | 带有A属性且该属性值为V的元素 | dojo.query("[foo='bar']") |
| E[A~='V'] | 标签为E并带有属性A，且该属性值为空格分隔的列表，同时其中一个值恰好等于V的元素 | dojo.query("div[foo~='bar']") |
| E[A^='V'] | 标签为E并带有属性A，且该属性值以V开头的元素 | dojo.query("div[foo^='bar']") |
| E[A$='V'] | 标签为E并带有属性A，且该属性值以V结尾的元素 | dojo.query("div[foo$='bar']") |
| E[A*='V'] | 标签为E并带有属性A，且该属性值中包含子字符串V的元素 | dojo.query("div[foo*='bar']") |
|  | 布尔或 | dojo.query("div,span.baz") |
| E > F | 作为元素E子元素的F元素 | dojo.query("div > span") |
| E F | 作为元素E后代元素的F元素 | dojo.query("E F") |

## 进一步体验

下面，我们就以一个包含故事书片段的简单文档为例，来领略一下dojo.query的魅力。为了节省版面，标记中只包含故事的纲要：

```
<div id="introduction" class="intro">
    <p>
        Once upon a time, long ago...
    </p>
</div>

<div id="scene1" class="scene">...</div>
<div id="scene2" class="scene">
    <p>
        At the table in the <span class="place">kitchen</span>, there were three
bowls of <span class="food">porridge</span>. <span class="person">Goldilocks</span>
was hungry. She tasted the <span class="food">porridge</span> from the first bowl.
    </p>
    <p>
        "This <span class="food">porridge</span> is too hot!" she exclaimed.
    </p>

    <p>
        So, she tasted the <span class="food">porridge</span> from the second bowl.
    </p>

    <p>
        "This <span class="food">porridge</span> is too cold," she said
    </p>
    <p>
        So, she tasted the last bowl of <span class="food">porridge</span>.
    </p>
    <p>
        "Ahhh, this <span class="food">porridge</span> is just right,"
she said happily and she ate it all up.
    </p>
</div>
<div id="scene3" class="scene">...</div>
```

前面曾经提到过，getElementsByTagName返回一个给定类型的DOM节点的数组。与之对等的dojo.query只需接收一个字符串格式的标签名作为参数即可。因此，为了查询页面中的所有div元素，可以使用dojo.query("div")，即：

```
dojo.query("div")
//返回[div#introduction.intro, div#scene1.scene, div#scene2.scene,
//div#scene3.scene]
```

如果不想查询整个文档，而只想查询某个特定节点中的元素，那么还需要指定第二个参

数作为查询的起始节点。例如，只查找scene2中的段落元素，而不是整个文档中的段落元素，需要将scene2节点或仅将该节点的id作为第二个参数，像下面这样：

```
dojo.query("p", "scene2")
//返回[p, p, p, p, p, p]
```

查询文档中带有指定类的元素也很简单，只要使用CSS选择符语法提供要查询的类名即可，根据规范，需要在类名前面加上一个点。例如，要查询带有food类的所有元素，应该使用下面的语句：

```
dojo.query(".food")
//返回[span.food, span.food, span.food, span.food, span.food,
//span.food, span.food]
```

**警告：** Base的addClass和removeClass方法接收的类名参数不带前置的点，如果不小心加上了点，那么会导致错误的结果。刚使用Dojo时间不长的开发人员最容易在这个问题上出错。

同时查询标签和类也很容易，把标签和类写到一起就行。下面这条语句就用来查询带有place类的span元素：

```
dojo.query("span.place")
//返回[span.place]
```

基于一个类选择元素的确方便，但基于多个类选择元素的时候也很多。好在选择多个类同样也只要把这些类组合起来即可。例如，可以像下面这样选择带有food和place类的所有元素：

```
dojo.query(".food,.place")
//返回[span.food, span.food, span.food, span.food, span.food, span.food,
//span.food, span.place]
```

**注意：** 由逗号分隔的CSS表达式组件之间是无关的。这一点与某些数学运算符或语法组件具有左关联的性质不同。

最后，我们示范如何查询一个特定节点的后代元素。对于前面的示例文档而言，如果想找到带有food类且属于scene2后代的所有元素，可以这样做：

```
dojo.query("#scene2 .food")
//返回[span.food, span.food, span.food, span.food, span.food, span.food,
//span.food]
```

注意：在此使用>组合符会导致返回空列表，因为scene2的子元素中没有一个带有food类：

```
dojo.query("#scene2 > .food")
//返回[]
```

**警告**：许多朋友对子组合符（>）和后代组合符（空格）的含义不清楚。其实，它们的区别很明显：子组合符返回直接子节点，而后代组合符则返回DOM结构中的所有后代节点。

虽然这次体验的旅程很短暂，但我们还是要提醒读者注意一点：通过提供最具体的查询条件来尽可能缩小查询范围，能够明显提高查询性能。

## 状态跟踪示例

除了在DOM中查询节点这个明显的用途之外，dojo.query的另一个强大之处体现在它会改变许多常见问题的解决方式，因为它为开发人员提供了创新的可能性。例如，在较为复杂的应用程序设计中，基本上都会涉及到状态跟踪问题。可能是需要确定某段文本是否被选中，或者是判断某个操作是否已经执行过了。虽然可以通过引入变量来实现状态跟踪，但通过CSS类来跟踪状态往往会更优雅。

例如，假设你要开发一个基于Web的高级搜索引擎，该搜索引擎能够突出显示文档中的实体，而且你打算让用户可以在搜索结果中查看人名。那么，假定搜索结果中包含莎士比亚的悲剧《麦克白》（Macbeth）的信息，而你必须把其中的“人”（person）突出显示出来。其中，部分搜索记录如下：

```
.....
<a rel="person">First Witch</a>
When shall we three meet again
In thunder, lightning, or in rain?

<a rel="person">Second Witch</a>
When the hurlyburly's done,
When the battle's lost and won.

<a rel="person">Third Witch</a>
That will be ere the set of sun.

<a rel="person">First Witch</a>
Where the place?

<a rel="person">Second Witch</a>
Upon the heath.

<a rel="person">Third Witch</a>
There to meet with <a rel="person">Macbeth</a>.

.....
```

### 冗长、脆弱的方式

作为一名偏爱易用性的开发人员，你可能想在页面一侧添加一个小控制面板，让用户通过它切换搜索结果中的哪一类实体可以突出显示。如果采用低级JavaScript手段，或者说直接操作DOM的方式，可能会编写下列函数：

```
function addHighlighting(entityType) {
  var nodes  = document.getElementsByTagName("a");
  for (var i=0; i < nodes.length; i++) {
    if (nodes[i].getAttribute('rel')==entityType) {
      nodes[i].className="highlighted";
    }
  }
}

function removeHighlighting(entityType) {
  var nodes = document.getElementByTagName("a");
  for (var i=0; i < nodes.length; i++) {
    if (nodes[i].getAttribute('rel')==entityType) {
      nodes[i].className="";
    }
  }
}
```

也许这两个函数可以解决问题，但是其中隐含的对搜索结果中只会包含highlighted类名的假设总归还是有点不够老到，因为如果搜索结果中包含其他类名，那么这两个函数就得重写。为此，必须重新构思更完善的函数，以便在节点中包含多个类名时适当地添加和移除个别的类。而这就涉及到搜索className包含的字符串值，然后再有选择地添加或移除类名。当然，可以使用第2章介绍的Base提供的addClass和removeClass方法来减少编码量，但仍然不能将代码量减至最少。

### 简短、健壮的方式

下面是使用dojo.query解决该问题的可靠方式，没有任何冗余代码：

```
function addHighlighting(entityType) {
  dojo.query("span[type="+entityType+"]").addClass("highlighted");
}

function removeHighlighting(entityType) {
  dojo.query("span[type="+entityType+"]").removeClass("highlighted");
}
```

对于这个特殊的例子而言，你可以摆脱低级的DOM操作，也不必编写for循环并引入条件逻辑块，只要利用优雅的CSS选择符语法即可，而且，也没有忽略搜索结果文档的实体中可能包含多个类名的可能性。

尽管使用低级方式不能实现的操作使用dojo.query同样也无法实现，但通过前面的示例读者应该能够看到，dojo.query确实为在DOM中查询和操作元素提供了统一的接口。而且，它还在较高的层次上，通过略微增加查询字符串的复杂性取代了冗余的条件逻辑语句。更不必说，即使从表面上看，使用dojo.query也比使用低级的DOM方法显得灵巧多了。

如果读者希望使用query做更多的事，那么就得先了解NodeList提供的灵活性。NodeList是调用query返回的结果类型，也是下一节的主题。

## NodeList

NodeList是专门为高效地操作DOM节点而设计的一个Array的子类。NodeList最吸引人的一个特性，就是它能通过点运算符连缀操作结果。此外，映射、筛选以及查找节点索引也是NodeList的重要特性。

表5-2列出了NodeList提供的方法。这些方法的命名与Base提供的Array方法的命名约定相同。唯一需要注意的是，这些方法返回NodeList而不是Array。

**注意：** 要了解有关Array操作的基本介绍，请参见第2章的“数组处理”。

表5-2：NodeList提供的方法

| 名称 | 说明 |
| --- | --- |
| indexOf(/*DOMNode*/n) | 返回NodeList中某个项第一次出现的位置 |
| lastIndexOf(/*DOMNode*/n) | 返回NodeList中某个项最后一次出现的位置 |
| every(/*Function*/f) | 如果NodeList中每个项传入函数f都返回true，则返回true |
| some(/*Function*/f) | 如果NodeList中至少一个项传入函数f返回true，则返回true |
| forEach(/*Function*/f) | 通过函数f运行每个项，返回原始的NodeList |
| map(/*Function*/f) | 通过函数f运行每个项，返回运行结果组成的NodeList |
| filter(/*Function*/f) | 通过函数f运行每个项，返回符合函数条件的项组成的NodeList，或者对节点列表应用CSS查询筛选 |
| concat(/*Any*/item, ...) | 返回添加新项之后的NodeList，与Array.concat方法类似，只不过它返回NodeList |

表5-2：NodeList提供的方法（续）

| 名称 | 说明 |
| --- | --- |
| splice(/*Integer*/index, /*Integer*/howManyToDelete, /*Any*/item, ...) | 插入、删除或替换NodeList的项，返回被删除项的NodeList，与Array.splice方法类似，只不过它返回NodeList |
| slice(/*Integer*/begin, /*Integer*/end) | 返回切除部分项之后的NodeList，与Array.slice方法类似，只不过它返回NodeList |
| addClass(/*String*/class) | 为NodeList中的每个节点添加类 |
| removeClass(/*String*/class) | 从NodeList中的每个节点删除类 |
| style(/*String\|Object*/style) | 如果style参数是字符串，那么取得或设置特定的样式。如果style参数是对象，那么像dojo.style一样设置多个样式值 |
| addContent(/*String*/ content, /*String?\|Integer?*/ position) | 基于NodeList中每个节点的相对位置插入字符串或节点。有效的position参数值包括first、last、before和after。其中，first和last相对于每个节点的父节点，而 before和after相对于节点本身 |
| place(/*String\|Node*/ queryOrNode, /*String*/ position) | 相对于节点或与查询条件匹配的第一个节点放置NodeList中的每个项。有效的position参数值addContent方法相同(参见上面的addContent方法) |
| coords() | 返回包含NodeList中所有节点的盒子对象的Array——不是NodeList。盒子对象的格式为{ l: 50, t: 200, w: 300: h: 150, x: 100, y: 300 }，其中l表示左距浏览器视口的偏移量，t表示上距浏览器视口的偏移量，w和h分别对应盒子的宽度和高度，而x和y则是坐标的绝对位置 |
| orphan(/*String?*/ filter) | 根据filter参数指定的条件移除DOM节点，并返回由被移除的节点组成的NodeList |
| adopt(/*String\|Array\|DomNode*/ queryOrListOrNode, /*String?*/ position) | 相对于NodeList中的第一个节点插入DOM节点 |
| connect(/*String*/ methodNameOrDomEvent, /*Object*/ context, /*String*/ funcName) | 为NodeList中的每个项添加事件侦听器，这里使用dojo.connect在内部标准化了事件对象。该方法签名与dojo.connect的相似之处表现在，也要为它传递一个方法名或DOM事件，以便与可选的环境(context）及函数名（funcName）建立连接。其中，DOM事件名称应该全部采用小写。在多数情况下，都可以使用本章“DOM事件的快捷方式”中介绍的“快捷方式”来代替这个方法 |

表5-2：NodeList提供的方法（续）

| 名称 | 说明 |
| --- | --- |
| instantiate(/*String\|Object*/ declaredClass, /*Object?*/properties) | 为批量实例化部件提供便利[注]。假设NodeList包含许多来源节点，该方法尝试将它们解析为由declaredClass定义的部件类，同时传入properties参数提供的部件属性 |

注：第11章将详细讨论部件，本章不再举例演示instantiate方法的应用。

## 与数组方法类似的方法

读者或许还记得，Base提供了一些用于操作数组的方法。NodeList也提供了许多相同的方法。其中，indexOf、lastIndexOf、every、some、forEach、map和filter与数组中对应的方法具有类似功能。不过，NodeList的filter方法还会根据传入的参数提供一些特殊功能。（稍后介绍。）

首先，我们需要创建一个NodeList。创建NodeList的语法与创建数组一样，既可以为NodeList提供一些元素，也可以使用NodeList内置的concat方法基于现有的数组创建NodeList。

以下是创建NodeList的几种可能方式：

```
var nl = new dojo.NodeList(); // 创建一个空NodeList

var nl = new dojo.NodeList(foo, bar, baz);
// 创建包含已有节点的NodeList

var a = [foo, bar, baz];
// 假设有一个包含某些节点的数组

a = nl.concat(a); // 将这个数组转换为NodeList
```

---

**警告：** 如果通过下面的方式创建NodeList，结果很可能出乎读者的意料：

```
var nl = new dojo.NodeList([foo, bar, baz]);
```

这行代码返回的是一个包含一个Array对象的NodeList，与new Array([foo,bar,baz])的结果相同。

---

## 连缀NodeList结果

如果不需要把上一次操作的结果传给下一次操作，或者要处理的只有数组，Dojo的数组

方法就够用了。但是，如果你知道了NodeList支持结果连缀，那么恐怕就会发现这么优雅的语法才是你真正想要的。下面这个示例展示了如何实现连缀操作：

```
var nl = new dojo.NodeList(node1,node2,node3,node4,...);

nl.map(
  /* 映射某些元素…… */
  function(x) {
    /* ... */
  }
)
.filter(
  /* 接着从中筛选一些元素…… */
  function f(x) {
    /* ... */
  }
)
.forEach(
  function(x) {
    /* 再对最终结果进行处理…… */
  }
);
```

如果要使用标准的Dojo方法达到同样目的，那么就必须像下面这样引入一些中间的状态变量：

```
var a0 = new Array(node1,node2,node3,node4,...);

/* 映射某些元素…… */
var a1 = dojo.map(a0,
  function(x) {
    /* ... */
  }
);

/* 接着从中筛选一些元素…… */
var a2 = dojo.filter(a1
  function f(x) {
    /* ... */
  }
);

/* 再对最终结果进行处理…… */
dojo.forEach(a2
  function f(x) {
    /* ... */
  }
);
```

## 巧用 NodeList

虽然NodeList这个名字暗示该数据结构是专为操作DOM节点而设计的，但不要忘记它是Array的一个子类，因此可以用它来保存任何类型的数据，不止是节点。例如，要是你确实喜欢使用点运算符连缀操作的语法糖，那么完全可以把NodeList当作Array来使用，因为它也能像下面这样处理数字以及其他原始数据类型：

```
//假设有一个包含数字的NodeList
var nums = new dojo.NodeList(1,2,3,4,5,6,7,8,9,10)

//那么，通过使用连缀语法，就可以省掉反复使用中间状态变量的麻烦

nums
.filter(function(x) {/* ... */})
.map(function(x) { /* ... */})
//继续操作……
//你明白了吗?
;
```

**警告：** 虽然通过点运算符来连缀操作结果可以让代码看上去更美观，但缺少中间变量也会对调试和维护带来一定的负面影响。因此，要斟酌使用。

### 字符串式函数参数

同Base中用于操作数组的方法一样，在使用字符串式函数参数时，也可以使用专门的index、array和item关键字（参见第2章“数组处理”）。请看下面的示例：

```
//假设有一个名为nl的NodeList……

//那么使用item关键字可以不用编写完整的包装函数
nl.forEach("console.log(item)");
```

### 增强筛选功能

NodeList的filter方法除了具有dojo.filter方法处理数组的能力之外，还能够接收表示CSS选择符的字符串参数，并据此进行筛选。例如，前面的示例展示了传递一个函数，并对NodeList中的每个数据段进行操作；下面的示例则使用表示CSS选择符的查询字符串筛选DOM节点：

```
dojo.query("div")
.forEach(
```

```
  /* 输出所有div */
  function f(x) {
    console.log(x);
  })
.filter(".div2") //筛选出具有特殊类的div并再次输出
.forEach(
  /* 现在，只输出所有div.div2 */
  function f(x) {
    console.log(x);
  }
});
```

## 操作样式

在使用CSS选择符取得了一组节点之后，接下来很可能就要对这些节点执行样式操作。果真如此，NodeList也提供了一些辅助方法。其中，style方法特别重要，因为它会根据接收的第二个参数在getter和setter方法间进行转换。当然，这种行为与dojo.style方法很类似。

说到dojo.style方法，我们知道dojo.style(someNode,"margin")会返回一个DOM节点的外边距值，而dojo.style(someNode, "margin", "10px")则会把相应节点的外边距值设置为10像素。

NodeList中DOM节点的样式操作也不例外，只不过无须再传递第一个参数来指定节点了。与NodeList提供的其他处理节点的方法一样，style方法也会为列表中的每个节点应用样式：

```
// dojo.style方式……
var a = [];
/* 向数组中加载一些节点 */

// 遍历节点并应用样式
dojo.forEach(a, function(x) {
  dojo.style(x, "margin", "10px");
});

// NodeList方式……
dojo.query( /* some query */ )
.style("margin", "10px");
```

NodeList也提供了用于添加和删除类的的方法addClass和removeClass，同样，它们的使用方法也与对应的dojo.addClass和dojo.removeClass相似。也就是说，如果我们要操作元素的样式，既可以使用style方法设置其个别样式属性，也可以通过addClass和removeClass来为它们添加和删除类。当然，在没有现成样式类定义的情况下，使用style方法更方便；但如果已经有了预定义的样式类，那么通过addClass

和removeClass执行样式切换更是小事一桩。而且，跟style方法一样，添加和删除类的操作也没有多大变化：

```
dojo.query("span.foo", someDomNode).addClass("foo").removeClass("bar");
dojo.query("#bar").style("color","green");
```

## 放置节点

正如读者所料，NodeList也提供了一些操作节点在文档中位置的方法。首先，我们在前面看到过coords方法，它与对应的dojo.coords方法类似，返回列表中每个节点的坐标对象组成的数组。而NodeList的place方法则可以基于指定的位置，将整个NodeList中的节点按顺序插入DOM树中，这一点跟dojo.place也差不多。

不过，NodeList的addContent方法在工具箱中却找不出第二个；它为我们提供了相对于NodeList中的每个节点插入新节点或文本内容的手段。

下面的示例使用addContent方法在每个页面容器（pageContainer）元素后面插入了一段文本（会被包含在行内span标签中）。这个示例非常适用于页面中包含很多选项卡和堆叠容器的情形：

```
/* 为以pageContainer类标识的每个容器元素添加一条脚注信息 */
var nl = dojo.query("div.pageContainer").addContent("footer goes here!", "after");
```

至于place方法，由于它能够相对于文档中另一个节点插入整个NodeList，因此可以像下面这样将所有节点插入到一个id值为debugPane的容器中：

```
var nl = dojo.query("div.someDebugNodes").place("#debugPane", "last");
```

与对应的dojo.coords相似，但NodeList的coords方法返回包含对象的数组，每个对象以键/值对形式保存着NodeList中每个节点的坐标信息。每个对象中的键分别表示对应节点的上偏移量、左偏移量、宽度和高度，以及绝对的x和y坐标——即相对于浏览器视口的坐标。

---

**警告：** NodeList的coords方法返回数组，而非NodeList。读者可以在Firebug控制台中试验一下下面的代码：

```
dojo.forEach(
  dojo.query("div").coords( ),
  function(x) { console.log(x); }
);
```

---

NodeList提供的一个独特的、在工具箱中没有对应方法的方法是orphan，这个方法能够对每个元素进行简单筛选（基于一个CSS选择符，不能包含逗号），正如orphan的英文含义（“孤儿”）所暗示的，该方法会把符合筛选条件的每个子元素从DOM结构中移除。然后，将这些被移除的（或者说被“遗弃”的）子元素放到NodeList中返回。orphan方法移除DOM节点的方式与使用内置DOM函数的方式（如foo.parentNode.removeChild(foo)）相比，显得更自然一些。

例如，要从DOM中移除作为span元素子元素的所有超链接，并将它们作为一个新NodeList返回，可以像下面这样：

```
var nl = dojo.query("span > a").orphan()
```

**警告：** 必须在>组合符左右两边各添加一个空格。

最后，adopt方法本质上是执行orphan方法的逆操作（adopt的英文含义是“收养”），即可以把元素插回到DOM结构中。这个方法很灵活，它可以接收特定的DOM节点、查询字符串或者一个NodeList作为参数。插入操作是相对于调用adopt方法的NodeList的第一个元素进行的。adopt方法接收的第二个参数用于指定插入节点的相对位置（first、last、after和before）。例如：

```
var n = document.createElement("div");
n.innerHTML="foo";
dojo.query("#bar").adopt(n, "last");
```

## DOM事件的快捷方式

既然可以通过NodeList处理那么多操作，那么读者大概会自然而然地想到也能通过它来批量处理节点与特定DOM事件之间的关系，例如，失去焦点、鼠标移动和按键等等。NodeList提供的另外一些内置方法，确实也简化了根据一个或多个DOM事件触发自定义操作的处理。

NodeList提供了下列DOM事件方法，用于批量处理自定义事件：

- onmouseover
- onmouseenter
- onmousedown
- onmouseup

- onmouseleave
- onmouseout
- onmousemove
- onfocus
- onclick
- onkeydown
- onkeyup
- onkeypress
- onblur

下面，我们看一个捕获在特定元素上发生的鼠标移动事件的例子。这个例子示范了通过使用 onmouseover 方法，可以定义当该事件发生时需要触发的操作：

```
dojo.query("#foobar").onmousemove(
  function(evt) {
    console.log(evt); // 其实，读者可以在此实现更有意义的操作!
  }
);
```

由于内部使用了 dojo.connect 方法，因此 DOM 事件方法传递的事件对象都经过了标准化。而且，dojo.connect 提供的事件模型是遵循 W3C 规范进行标准化的。

目前，对通过 NodeList 的 connect 方法创建的连接，还没有直截了当的方式可以管理或者断开；不过，Dojo 的 1.x 中的某个版本或许会提供这种能力。如果仅靠页面卸载时自动断开连接不能满足需求，而且读者也觉得有必要自己来管理这些连接的话，那么可以使用常规的 dojo.connect 方法代替 NodeList 的 forEach 方法。

例如，对于前面的例子，如果想要手动管理创建的连接，可以像下面这样做：

```
var handles =
  dojo.query("a").map(function(x) {
    return dojo.connect(x, "onclick",
      function(evt) { /* …… */ });
  });

/* 将来的某个时候…… */
dojo.forEach(handles, function(x) {
  dojo.disconnect(x);
});
```

# 动画方法

**注意：** 读者可以先跳过本节内容，等到学习完了第8章再回过头来翻看这一节。第8章全面介绍了与动画有关的主题。

通过DHTML创建动画效果总会让人觉得有些麻烦，而且，事实的确如此。不过，NodeList提供的动画方法可以让我们像使用其他NodeList方法一样轻松地创建动画。从应用程序开发的角度来看，这意味着实现淡入淡出效果不必费吹灰之力，而且通过_Animation.animateProperty方法更可以实现复杂的效果。

**警告：** _Animation的前面有一个下划线。在特定的环境下，这个前置下划线的含义就是相应的API并不是最终版本；对于_Animation而言，读者可以将它看成一个“黑盒子”。虽然本节介绍的内容基于Dojo1.1，而且与_Animation相关的API也已经相当稳定，但也不排除在将来的Dojo版本中会修改其API的可能性。

表5-3列出了当前可用的所有动画方法。不过要使用这些方法，必须事先明确地调用dojo.require("dojo.NodeList-fx")。表中所有方法都接收一个关联数组作为参数，数组中的键/值对用于提供动画的持续时间、起止位置、颜色等信息。

表5-3：NodeList提供的动画方法

| 名称 | 说明 |
| --- | --- |
| fadeIn | 淡入列表中的每个节点 |
| fadeout | 淡出列表中的每个节点 |
| wipeIn | 擦入列表中的每个节点 |
| wipeout | 擦出列表中的每个节点 |
| slideTo | 将列表中的每个元素滑动到指定位置 |
| animateProperties | 将列表中的每个元素的属性变幻到指定的值 |
| anim | 与animateProperties方法类似，但它返回一个已经执行的动画对象。要了解更多信息，请参见dojo.anim |

可想而知，使用这些方法也很简单。跟试验Dojo工具箱中的其他特性一样，读者可以在Firebug控制台中逐个试验这些动画效果。首先，从简单的淡出效果开始：

```
dojo.require("dojo.NodeList-fx");
```

```
//在NodeList-fx加载之后……
dojo.query("p").fadeOut().play()
```

然后，如果想尝试复杂一些的动画设置，可以向关联数组参数中添加一些键/值对，再看看效果如何：

```
dojo.require("dojo.NodeList-fx");

//在NodeList-fx加载之后……
dojo.query("div").animateProperty({
  duration: 5000,
  properties: {
    color: {start: "black", end: "green"},
  }
}).play();
```

注意，各种效果方法都将返回一个_Animation对象，而该对象的play方法是激活动画的标准机制。

## 创建NodeList扩展

虽然NodeList的内置方法都十分有用，但过不了多久，读者可能就会发现手边仍然缺少一些它没有提供的辅助方法。好在，要向NodeList中添加自定义功能其实也很简单。例如，我们看一看下面这个通过query方法返回NodeList中每个元素innerHTML属性值的代码：

```
dojo.query("p").map(function(x) {return x.innerHTML;});
```

事实上，与不使用工具箱相比，这行代码已经可以算是相当简洁了。但是，如果使用前面介绍的字符串式函数参数的话，则还可以将这行代码再次改进如下：

```
dojo.query("p").map("return item.innerHTML;"); //使用特殊的item关键字
```

这当然是一次明显的改进，不过，读者是否觉得还能在此基础上把它变得更易读、更简洁吗？这就涉及到要对NodeList进行扩展了，即可以把映射的相应逻辑放到一个更易读也更优雅的函数调用中，并且为该函数起个直观的名字，让人对使用它完成的操作一目了然：

```
//通过一个函数扩展NodeList的原型
dojo.extend(dojo.NodeList, {
    innerHTML : function() {
        return this.map("return item.innerHTML");
    }
});
```

```
// 调用这个新方法
dojo.query("p").innerHTML();
```

扩展NodeList的可取之处在于，只要事先进行一些细致的规划，就能显著提升设计的清晰程度，同时也能增强程序的易维护性。

另外，在创建包含这种扩展的模块时，建议的操作规程是创建一个名为*ext-dojo*的子模块（译注1），并在其中包含一个名为*NodeList.js*的源文件。这样，最终通过dojo.require语句来加载该模块时，对阅读你代码的任何人而言，都会非常直观明了。因此，扩展NodeList的最终目标就是要能够像下面的代码一样使用该扩展：

```
/* ..... */
dojo.require("dtdg.ext-dojo.NodeList");

/* ..... */

dojo.query("p").innerHTML();
```

当然，如果再较点真儿的话，也可以把源文件命名为*NodeList-innerHTML.js*，这样好像就更清楚了。无论如何，原则只有一个，那就是在确保命名一致的前提下，怎么合适怎么来。

## 分离行为

Core基于query方法提供了一个轻量级的扩展，从而使开发人员能够通过behavior模块将事件和DOM操作从HTML标记中分离出来。对这个概念，读者刚开始可能不会感到太明确，但是，它能够按照与标记无关的方式定义节点的行为，能够为设计增添极大的灵活性。例如，通过它可以在不知道哪里有或者有多少锚元素的情况下，就为所有锚元素指定单击事件处理程序。由于可以使用表5-1中列出的CSS选择符语法来查找为其添加行为的节点，因此能够通过该模块实现的操作几乎是无穷的。

目前，behavior模块提供了两个API。其中，add方法用于按顺序追加行为，而apply方法用于触发行为队列中的行为：

```
dojo.behavior.add(/*Object*/ behaviorObject)
dojo.behavior.apply()
```

也就是说，虽然可以使用add为一组DOM节点添加行为，但在调用apply之前这些行为不会发生。正如第4章我们曾讨论过的，之所以将整个操作分成添加行为和应用行为两个步骤，是因为在最终应用行为之前允许多次添加行为与第4章所讨论的异步通信模式可以大致保持一致。

---

译注1： 即子目录。

**注意：** 第4章介绍了Dojo的IO子系统中的一种主要数据结构——Deferred。Deferred为JavaScript提供了基于线程的特性，因为它本身可以连续地应用多个回调函数和错误处理函数。在理解了Deferred的模式之后，那么独立的add和apply所具有的实用性也就不言而喻了。

add和apply方法接收的对象参数非常灵活，有多种变化。简单地说，这个所谓的行为对象包含的键/值对用于将CSS选择符映射到提供DOM事件处理程序的对象，而后者中的DOM事件处理程序同样以键/值对形式指定。在讲解具体的示例之前，先来看一看表5-4，其中列出了行为对象可能的取值。

表5-4：行为对象的值

| 键 | 值 | 说明 |
| --- | --- | --- |
| Selector (String) | Object | 对象值中包含键/值对，用于将DOM事件名或特殊的found关键字映射到事件处理程序或（发布/预计模式中的）主题名称。例如：<br>`{`<br>`onclick : function(evt) {/*...*/},`<br>`onmouseover : "/dtdg/foo/moveover",`<br>`found : function(node) {/*...*/},`<br>`found : "/dtdg/bar/found"`<br>`}`<br>在有主题被发布的情况下，会将经过标准化的事件对象传递给预订的处理程序<br>对于事件处理程序，会将经过标准化的事件对象传递给相应的函数<br>在指定特殊的found关键字的情况下，会将匹配的节点传给处理程序或者将其与发布的主题一同传递给预订的处理程序 |
| Selector (String) | Function | 当匹配选择符的每个节点执行这个处理程序时，相应的节点作为参数传入处理程序中 |
| Selector (String) | String | 匹配选择符的每个节点都会发布由这个字符串指定的主题名。而且，每个节点都将被传递给预订该主题的处理程序 |

**注意：** 必须按照CSS选择符的语法，以实际的字符串作为行为对象的键。例如，行为对象{div : function(evt) {/*……*/}}是正确的，但{#foo : "/dtdg/foo/topic"}是无效的，因为#foo不是一个有效的标识符。

下面，请读者仔细看一看例5-1，这个例子展示了behavior模块的一些实际应用。

例5-1：dojo.behavior模块的实际应用

```
<html>
   <head>
         <title>Fun with Behavior!</title>

<link rel="stylesheet" type="text/css"
    href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
    djConfig="isDebug:true"
></script>

<script type="text/javascript">
    dojo.require("dojo.behavior");

    dojo.addOnLoad(function() {
            /* 向dojo.behavior中传入行为对象。
               该对象会在页面加载时被自动添加 */
            dojo.behavior.add({

                /* 行为对象的键可以是任何有效的CSS选择符，该选择符
                可以映射到一个行为或一组行为 */
                /* 将一个键映射到一个函数相当于映射到如下对象：
                {found
                 : function(node){ …… }} */
                ".container" : function(node) {
                    // 为匹配的节点应该一些样式

                    dojo.style(node, {
                        border : "solid 1px",
                        background : "#eee"
                    });
                },

                /* 将下面的键映射到一组行为 */
                "#list > li" : {
                    /* 与dojo.connect中的DOM事件一样，可以操作事件对象 */
                    onmouseover : function(evt){dojo.style(evt.target,
                    "background", "yellow");},
                    onmouseout : function(evt){dojo.style(evt.target,
                    "background", "");},
```

```
                /* 字符串值作为主题名发布 */
                onclick : "/dtdg/behavior/example/click",

                /* "found"是一个允许操作节点的通用处理程序 */
                found : function(node) {dojo.style(node,"cursor", "pointer")}
            }
        });

        /* 在代码中的其他地方，设置预订处理程序…… */
        dojo.subscribe("/dtdg/behavior/example/click",function(evt) {
            console.log(evt.target.innerHTML, "was clicked");
        });
    });
    </script>
<head>

    <body>
        <div class="container" style="width:300px">
            Grocery List:
            <ul id="list">
                <li>Bananas</li>
                <li>Milk</li>
                <li>Eggs</li>
                <li>Orange Juice</li>
                <li>Frozen Pizzas</li>
            </ul>
        </div>
    </body>
</html>
```

通过这个示例我们知道，在页面加载之前设置的任何行为都会自动就绪。但是，在页面加载之后，则需要首先通过add添加行为，然后再通过apply应用行为。下面的代码为列表元素添加了另外一个单击处理程序：

```
dojo.behavior.add({
  "#list > li" : {
    onclick : "/dtdg/behavior/example/another/click"
  }

});
dojo.behavior.apply();

dojo.subscribe("/dtdg/behavior/example/another/click", function(evt) {
  console.log("an additional event handler...");
});
```

虽然读者最应该关心的是如何将节点的行为从标记中分离出来，但通过*behavior*模块的apply方法建立连接的好处也是显而易见的：在现有行为基础上添加的行为不会与现有行为冲突。换句话说，新行为不会取代原有行为；即可以把行为添加到不同的层中，而对这些行为的管理则无须我们操心。

# 小结

在学习完本章之后，读者应该：

- 能够使用dojo.query找到页面中的任何节点。
- 基本理解CSS选择符语法。
- 熟悉NodeList并理解工具箱提供的类似Array方法的各种变体方法。
- 能够通过NodeList方法间的连缀清晰快捷地实现对DOM元素的处理。
- 明白通过巧用NodeList可以省去使用工具箱中其他实用方法的麻烦。
- 能够使用NodeList放置DOM节点、处理动画、设置连接及控制样式。
- 理解通过自定义操作扩展NodeList可以做到将处理dojo.query结果的代码最优化。
- 明确从HTML标记中分离DOM事件的好处，以及如何通过behavior模块实现这一点。

下一章，我们将讨论国际化。

第6章

# 国际化（i18n）

本章简要介绍Dojo为模块国际化提供的工具。主要内容涉及基于地区定义语言包和使用Core中的实用方法格式化并解析日期、货币及数字。顺便说一下，*i18n*是国际化的英文拼写internationalization的简写方式，其中i和n是该单词的首尾字母，18表示i和n之间还有另外18个字母。

## 国际化简介

如果你运气不错，开发的应用程序正逐渐得到世界各地网民的喜爱，那么你一定会考虑让自己的应用程序支持多种语言。尽管Dijit（本书第二部分介绍）已经支持了部分地区的国际化，但你自己开发的自定义模块和部件同样可能会有国际化的需求。Dojo为支持国际化提供了高度统一的解决方案，因而你就不必自己开发转换字符的映射系统了。而且，Dojo还会负责管理如何加载，即如何优化加载过程，这个问题你也可以不用考虑了。此外，Dojo还提供了支持常见数字、货币格式化等操作的实用方法。

有必要指出的是，i18n实用程序从技术上属于Core而不属于Base。不过，在跨域构建中，*dojo.xd.js*会以额外2KB的体积增大为代价包含`dojo.i18n`模块，以便解决与i18n语言包有关的加载问题。但无论怎样，我们都应该继续使用`dojo.require("dojo.i18n")`明确加载该模块。

对自定义的模块进行国际化处理很简单，因为语言信息就与JavaScript源文件一起保存在模块目录中的`nls`子目录下——`nls`的含义是*native language support*（本地语言支持）。在`nls`目录中，又以RFC3066标准（Tags for the Identification of Languages，识别语言的标签）规定的地区简写命名各种语言版本的目录（注1）。

注1：　参见*http://www.ietf.org/rfc/rfc3066.txt*。

举例来说，普通英语的简写是*en*，美国英语的简写是*en-us*，普通西班牙语的简写为*es*。在启用工具箱期间，Dojo会通过用户浏览器获取地区信息，然后将该信息内部保存起来。读者可以在Firebug中执行一下`dojo.locale`，看看会返回什么结果。由`dojo.locale`返回的结果，就是Dojo在加载已经国际化的模块时用于确定最合适语言版本的依据。

# 自定义模块的国际化

假设读者对第2章“构建Genie示例模块”中的Genie模块进行了多次扩展，并得到了一个非常灵验的模块（名为`psychic`）。再假设读者的默认语言是英语，而想添加的第一门语言是西班牙语。

## 磁盘目录结构

与其他模块一样，这个灵验的模块就是一个被包含在典型目录结构中的源文件：

```
dtdg/
  psychic/
    Psychic.js /* 这个文件中包含着许多有价值的信息 */
```

当然，基于这个`psychic`模块创建的应用程序将具有预言能力；而模块用户则会在页面中像下面这样加载它：

```
<script type="text/javascript">
    dojo.require("dtdg.psychic");
    dojo.addOnLoad(function() {
      dtdg.psychic.predictFuture();
    });
</script>
```

尽管`predictFuture`方法会执行真正有魔力的操作，但此时此刻我们关心的却是屏幕上显示的字符串值，因为它是国际化成功与否的标志。最终，模块的输出结果将通过下面的逻辑显示在屏幕上：

```
dojo.byId("reading").innerHTML = predictFuture( /* 有魔力的操作 */ );
```

作为国际化的第一步，我们从古英语和古西班牙语开始，不考虑任何方言。在这个前提下，`nls`目录的结构应该如下所示：

```
dtdg/
  psychic/
    Psychic.js
    nls/
```

```
readings.js /* 默认的英语语言包 */
es/
  readings.js /* 西班牙语言包 */
en/
  /* 默认的英语文件夹是空的，因此 Dojo
     会在其上一级目录 nls/ 中查询语言包 */
```

按照约定，包含本地语言翻译信息的*.js*文件被称为语言包。另一个约定是默认语言包必须位于nls的顶级目录下，即不能放在一种特定语言的目录内。这一约定的理论基础是，人们都希望默认的语言包出现在nls目录下，因为这最符合逻辑；而且，在其特定的目录（这里的en）中再包含一个语言包是没有必要的 —— 只会增加维护的麻烦。

## 定义字符串表

下面摘自每个 *readings.js* 文件中的片段，展示了构成最终语言包的部分字符串。

首先，看一下默认的 *readings.js* 文件：

```
{
/* ... */
reading101 : "You're a Libra, aren't ya darling?",
reading102: "Can you please tell me your first name only, and your birthday please?",
reading103: "Yep, that's the Daddy."
/* ... */
}
```

下面是 *es/readings.js* 文件：

```
{
/* ... */
reading101 : "¿Eres un Libra, no, mi corazón?",
reading102: "¿Me puedes dar el nombre y tu cumpleaños por favor?",
reading103: "Sí, el es papá"
/* ... */
}
```

通过 Dojo 实现应用程序本地化的一大特色，就是提供字符串列表的方式非常简单。

## 综合起来

接下来，应该把前面的内容综合起来，展示一下支持多种语言有多容易；不过，我们要先看一看在此过程中涉及的几个相关方法，如表 6-1 所示。

表6-1：本地化方法

| 名称 | 说明 |
| --- | --- |
| dojo.i18n.getLocalization(/*String*/moduleName, /*String*/bundleName, /*String?*/locale) | 返回一个Object，包含给定语言包的本地化信息。在默认情况下，locale参数的值为dojo.locale；指定不同的值可以找到相应的语言包 |
| dojo.i18n.normalizeLocale(/*String?*/locale) | 返回标准格式的地区信息 |
| dojo.requireLocalization(/*String*/moduleName, /*String*/bundleName, /*String?*/locale) | 与dojo.require加载模块一样加载翻译后的资源。注意，该方法属于Base，不属于Core的i18n模块 |

虽然以前要通过psychic.reading102才能从散列值中读取reading102的值，但是现在可以使用工具箱提供的方法。如果现在有了一个针对特定用户地区的语言包，那么一切已经“就绪”了。下面的通用逻辑适合在各种语言包中查找信息：

```
/* 首先加载Dojo的i18n实用程序…… */
dojo.require("dojo.i18n");

/* 然后，加载各种语言包*/
dojo.requireLocalization("psychic", "readings");

function predictFuture() {

  /* 在predictFuture 内部的某一处位置上……*/
  var future= dojo.i18n.getLocalization("psychic", "readings").reading597;
  return future;
}
```

注意，可以通过修改dojo.locale来测试不同的语言包，但最好是在djConfig中修改这个值。以下代码基于Dojo的本地安装文件测试西班牙语言包：

```
<head>
   <script type="text/javascript" src="your/path/to/dojo.js"
    djConfig="dojo.locale:'es'">
    </script>
</head>

<!--
  现在，所有的国际化模块都将使用西班牙语言包
-->
```

**警告：** 与请求其他模块或资源一样，不要将dojo.i18n.getLocalization作为对象的属性定义来调用；而应该在dojo.addOnLoad块中调用它：

```
dojo.addOnLoad(function() {
    //返回一个本地化的对象
    var foo = {bar : dojo.i18n.getLocalization( /* ...*/)}
});
```

这里需要注意的一个细微差别是，如果默认是其他英语地区，而测试的仍然是西班牙语言包，那么将会同时加载 *nls/es/readings.js* 和 *nls/readings.js* 语言包。事实上，包含在 *nls/*目录下的默认语言包总会被加载。读者可以在Firebug的*Net*标签下自己验证这一点。

虽然这个特殊的例子没有涉及任何语言的方言，但在加载本地化语言包时是一定要把方言考虑在内的。例如，在地区为 *en-us* 而且提供了 *en-us* 语言包的情况下，Dojo 会尝试同时加载 *en-us* 和 *en* 语言包，并将它们合并起来以便各种 dojo.i18n.getLocalization调用查询。此时的假设是，当为英语定义针对地区的特殊符号时，人们会尽可能在 *en* 语言包中提供较多的通用信息，从而覆盖或补充其他方言包（如 *en-us*）中的信息。

### 使用构建工具提升性能

在讨论国际化的最后，有必要提醒读者注意的是，Util中提供的Dojo构建工具能够自动处理构建自定义模块过程中涉及到的最小化同步调用及数据冗余的种种细节。也许乍一看这没有什么大不了的，但通过构建工具将众多小型资源文件组合起来，可以消除查找信息过程的时间延迟问题。具体到加快页面加载而言，这一点确实关系重大。有关Util和构建工具的讨论将在第 16 章中进行。

## 日期、数字和货币

Core通过dojo.date、dojo.number和dojo.currency提供了对日期、数字和货币的国际化支持。在本书第二部分中介绍Dijit时，读者将会看到许多常用部件中都用到这些模块。本节旨在向读者展示这些模块的主要特性。

### 日期

表 6-2 列出了 dojo.date 模块的方法。这些方法在内置 Date 对象的基础上，为日常开发提供了重要的补充功能。

表6-2：dojo.date模块的方法

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.date.getDaysInMonth (/*Date*/date) | 整数 | 返回data中月份的天数 |
| dojo.date.isLeapYear (/*Date*/date) | 布尔值 | 如果date是闰年，那么返回true |
| dojo.date.getTimezoneName (/*Date*/date) | 字符串 | 返回由浏览器定义的时区信息。必须传入一个Date对象，因为时区可能因夏令时而有所不同 |
| dojo.date.compare(/*Date*/ date1, /*Date*/ date2, /*String?*/ portion) | 整数 | 如果两个参数相等则返回0；如果data1 > date 2则返回一个正数；如果date 1 < date 2则返回一个负数。在默认情况下，日期和时间都会参与比较。如果将portion参数指定为date或time，则意味着只比较日期或时间 |
| dojo.date.add(/*Date*/date, /*String*/ interval, /*Integer*/ amount) | 日期值 | 用于为Date对象加上一段时间间隔。需要提供数值（amount）和单位类型(interval)，其中单位类型包括"year"、"month"、"day"、"hour"、"minute"、"second"、"millisecond"、"quarter"、"week"或"weekday" |
| dojo.date.difference (/*Date*/date1, /*Date*/ date2, /*String*/ interval) | 整数 | 用于根据指定的单位类型（interval）计算两个日期之间的差值，其中单位类型包括"year"、"month"、"day"、"hour"、"minute"、"second"、"millisecond"、"quarter"、"week"或"weekday" |

**警告：** 到Dojo1.1为止，getTimezonName方法还没有实现本地化。

## 数字

表6-3和表6-4列出了dojo.number模块提供的辅助方法，用于把字符串值解析为数字值、按照特定的模式模板格式化数字以及舍入到特定的小数位。

表6-3：dojo.number模块为表6-4中的dojo.number.format和dojo.number.parse方法提供的格式化选项

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| pattern | 字符串 | 可用于覆盖格式化模式 |
| type | 字符串 | 基于地区的格式类型。有效值包括"decimal"、"scientific"、"percent"、"currency"和"decimal"，其中"decimal"是默认值 |
| places | 数值 | 指定显示的固定位数，该选项覆盖由pattern提供的任何信息 |
| round | 数值 | 基于倍数（multiple）指定舍入属性。例如，5表示舍入为最接近的0.5，而0表示舍入为最接近的整数 |
| currency | 字符串 | 符合ISO4217标准的货币代码。例如，"USD"表示美元 |
| symbol | 字符串 | 本地化的货币符号 |
| locale | 字符串 | 提供特定的地区，用以影响格式化规则 |

表6-4：dojo.number模块的方法

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.number.format(/*Number*/value, /*Object*/options) | 字符串 | 使用基于本地的设置将一个数字值格式化为一个字符串值。其中，options参数的取值范围见表6-3 |
| dojo.number.round(/*Number*/value, /*Number*/places) | 数值 | 将一个数字值的小数部分舍入为指定位数 |
| dojo.number.parse(/*String*/value, /*Object*/options) | 数值 | 使用基于本地的设置将一个字符串值转换为一个数字值。其中，options参数的取值范围是表6-3中的pattern、type、locale、symbol和currency |

## 货币

与dojo.number模块类似，dojo.currency模块同样提供了格式化数值的选项和方法——如表6-5和6-6所示——只不过其货币代码由ISO4217（注2）规定。

注2：参见*http://en.wikipedia.org/wiki/ISO_4217*。

表6-5：dojo.currency模块为表6-6中的dojo.currency.format和dojo.currency.parse方法提供的格式化选项

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| currency | 字符串 | 由ISO4217定义的由3位字母表示的货币代码，如"USD" |
| symbol | 字符串 | 用于覆盖默认货币符号的字符串值 |
| pattern | 字符串 | 用于覆盖默认货币模式 |
| round | 数值 | 用于指定基本的舍入规则：-1表示不舍入，0表示舍入最近的整数，5表示舍入最近的0.5 |
| locale | 字符串 | 覆盖默认的地区，从而影响格式化规则 |
| places | 数值 | 指定能够接受的小数位数（默认由货币规则定义） |

表6-6：dojo.currency模块的方法

| 名称 | 返回值类型 | 说明 |
| --- | --- | --- |
| dojo.currency.format (/*Number*/value, /*Object?*/options) | 字符串 | 使用基于本地的设置将一个数字值格式化为一个字符串值。其中，options参数的取值范围如表6-5所示 |
| dojo.currency.parse (/*String*/ expression, /*Object?*/ options) | 数值 | 将具有适当格式的字符串值转换为数字值。其中，options参数的取值范围如表6-5所示 |

**注意：** Dojo的某些工具可以用来为任何地区和货币提供支持，因为这其中的许多工作量只涉及到构建信息查找表。要了解相关的更多信息，请参考*util/buildscripts/cldr/README*。

# 小结

在学习完本章之后，读者应该能够：

- 对自定义模块进行国际化，使其支持多个地区。
- 知道Core提供了处理货币、数字和日期的实用方法，这些方法在国际化处理中都可能用到。

下一章，我们将讨论拖放。

# 第7章

# 拖放

DnD（Drag-and-Drop，拖放）为Web应用程序增加的类似桌面应用程序般的功能和易用性，使得Web应用程序开始变得与以往不同。本章系统地讨论拖放这个主题，并向读者展示较多的示例及源代码。读者可以在这些示例的基础上进一步扩展，并利用它们来装点自己的应用程序。或者，甚至可以大胆地将拖放的理念与Dojo提供的支持机制整合起来，创建一个供大家在线娱乐的DHTML游戏。无论怎样，相信本章一定会很有意思。

## 拖动

虽然拖放作为桌面应用程序的主要功能已经存在20多年了，但它在Web应用程序中出现还是近几年的事。之所以出现得这么晚，一是因为DOM方法本身比较简单、原始，二是拖放的事件驱动特性决定了构建一个跨平台的统一框架也不是件轻而易举的事。所幸的是，Dojo工具箱在支持拖放方面做得非常出色，它提供的实用方法可以将开发人员从繁琐耗时的手工开发中解脱出来。

## 简单的可移动对象

---

**注意：** 本章假设读者掌握了最低限度的CSS知识。要了解CSS的更多信息，请参考W3C schools提供的CSS教程，链接为*http://www.w3schools.com/css/default.asp*。另外，Eric Meyer的《CSS: The Definitive Guide》（O'Reilly）则是CSS领域的权威著作。

---

## 解析器

解析器是Core中最常用的一种资源。不过，用它用得最多的还是解析页面中的部件。为此，关于解析器的完整讨论将放在第11章正式介绍Dijit时再进行。虽然读者可以现在就翻到第11章阅读相关内容，但这个小附言栏中也为读者提供了此时此刻应该知道的所有信息。

解析器最常见的一种用途就是在页面加载过程中，在addOnLoad触发之前，查找并对页面中的部件进行实例化。解析器查找部件时，本质上是在查找带有特殊的dojoType属性的标签，这个dojoType属性用于指定应该插入页面的相应部件的资源名称。而我们所要做的，就是像加载其他资源一样，通过dojo.require("dojo.parser")请求解析器，在djConfig配置项中添加parseOnLoad:true指令，并在页面中的每个部件占位符（译注1）中指定dojoType属性。（实际上，也可以手工解析部件，不过相关内容我们也放在第11章再进行讨论。）

本章的示例代码中也会以同样方式使用解析器，因为它对于实现拖放功能所起的作用基本相同。换句话说，解析器将在页面中查找通过dojoType属性标识的拖放资源，然后通过后台处理为这些元素添加拖放功能。

首先，我们介绍一个最基本的例子在屏幕上移动一个对象（注1）作为起点：例7-1展示了包含必要标记的基本页面结构。请读者仔细看一遍，特别要注意引入Moveable类的加粗的代码行，然后我们再进行详细地分析。

例7-1：简单的可移动对象

```
<html>
   <head>
      <title>Fun with Moveables!</title>
       <style type="text/css">
             .moveable {
                 background: #FFFFBF;
                   border: 1px solid black;
                   width: 100px;
                   height: 100px;
                   cursor: pointer;
            }
           </style>
```

---

译注1：即XHTML标签。

注1：本章将从一般意义上使用术语对象来特指可移动的DOM节点。此处的对象与面向对象编程中的对象没有任何关系。

```
        <script
          type="text/javascript"
          djConfig="parseOnLoad:true,isDebug:true"
          src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

       <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");
        dojo.require("dojo.parser");
    </script>
</head>
<body>
    <div class="moveable" dojoType="dojo.dnd.Moveable" ></div>
    </body>
</html>
```

相信读者一定感觉到了，创建一个可移动对象非常简单。在把Moveable资源加载到页面中之后，要做的就只剩下以dojoType属性在页面中标记可移动的元素，并通过djConfig选项指定在页面加载时解析该元素了。当然，还有一些让这个可移动对象看起来更美观的CSS样式代码，不过，没有这些样式一切也都会照常的。

一般来说，在页面加载时可以通过解析完成的操作，在页面加载之后也可以通过编程方式来完成。以下就是一个非常类似的示例，该示例以编程方式构建了Moveable：

```
    <!-- ……省略的代码…… -->

    <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");

        dojo.addOnLoad(function() {
            var e = document.createElement("div");
            dojo.addClass(e, "moveable");
            dojo.body().appendChild(e);
            var m = new dojo.dnd.Moveable(e);
        });
    </script>
    </head>
     <body></body>
    </html>
```

表7-1列出了创建及销毁Moveable对象所需的方法。

表7-1：创建及销毁Moveable对象的方法

| 名称 | 说明 |
| --- | --- |
| Moveable(/*DOMNode*/node, /*Object*/params) | 用于标识节点，使其变得可移动的构造函数。其中，params参数可能的值包括：<br>handle (String \| DOMNode)<br>用作鼠标手柄的节点或节点的id。在默认情况下，使用可移动的节点本身。<br>skip (Boolean)<br>基于文本的表单元素是否在mouse down事件发生时忽略拖放操作（默认值为false）。<br>mover (Object)<br>自定义Mover对象的构造函数。<br>delay (Number)<br>延迟移到的像素数（默认值为0） |
| destroy() | 用于解除节点的可移动能力，并删除所有相关引用以便垃圾收集起作用 |

**注意：** Mover是Moveable内部使用的一种更加低级的拖放机制。本章不讨论Mover，在此提到它只为让读者知道。

接下来，我们继续在前面示例的基础上，通过设置skip参数让基于文本的表单元素保持可编辑。这一次，我们要在屏幕上构建一个粘性便条（sticky note），让它既可以移动又可以编辑。例7-2提供了相关的代码。

例7-2：通过Moveable创建一个粘性便条

```
<html>
    <head>
       <title>Even More Fun with Moveables! </title>
        <style type="text/css">
            .note {
                background: #FFFFBF;
                border-bottom: 1px solid black;
                border-left: 1px solid black;
                border-right: 1px solid black;
                width: 302px;
                height: 300px;
                margin : 0px;
                padding : 0px;
            }
```

```
        .noteHandle {
            border-left: 1px solid black;
            border-right: 1px solid black;
            border-top: 1px solid black;
            cursor :pointer;
            background: #FFFF8F;
            width : 300px;
            height: 10px;
            margin : 0px;
            padding : 0px;
        }
    </style>
        <script
            type="text/javascript"
            djConfig="parseOnLoad:true,isDebug:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Moveable");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>
        <div dojoType="dojo.dnd.Moveable" skip=true>
            <div class="noteHandle"></div>
            <textarea class="note">Type some text here</textarea>
         </div>
    </body>
</html>
```

---

**注意：** 如果读者对设置skip参数的效果没有感觉，可以试着将最外层DIV元素中的skip=true删除，看一看如果不忽略该表单元素，是否还能编辑其中的文本。

---

读者会发现无须给这个粘性便条设置拖动手柄，因为最内部的DIV元素是便条中唯一可以拖动的地方。但是，通过设置拖动手柄也可以实现与设置 skip 参数相同的效果 —— 把Moveable对象中能够响应拖动操作的区域限制在特定的部分（即拖动手柄），而让这个特定部分之外的表单元素可以编辑。以下是在上面加粗代码基础上修改而成的：

```
<div id="note" dojoType="dojo.dnd.Moveable" handle='dragHandle'>
    <div id='dragHandle' class="noteHandle"></div>
    <textarea class="note">This form element can't trigger drag action</textarea>
</div>
```

## 拖动事件

有时候，很可能需要检测拖动操作的开始和结束，以便为用户提供该次拖动操作的视觉提示。检测这些事件对于dojo.subscribe和dojo.connect而言只是小事一桩。例7-3展示了另一个连接和预订拖动事件的示例。

例7-3：连接和预订拖动事件

```
<html>
    <head>
        <title>Yet More Fun with Moveable!</title>
         <style type="text/css">
        .note {
            background: #FFFFBF;
            border-bottom: 1px solid black;
            border-left: 1px solid black;
            border-right: 1px solid black;
            width: 302px;
            height: 300px;
            margin : 0px;
            padding : 0px;
        }
        .noteHandle {
            border-left: 1px solid black;
            border-right: 1px solid black;
            border-top: 1px solid black;
            cursor :pointer;
            background: #FFFF8F;
            width : 300px;
            height: 10px;
            margin : 0px;
            padding : 0px;
        }
        .movingNote {
            background : #FFFF3F;
        }
    </style>

    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");

        dojo.addOnLoad(function() {
            // 创建并保存 Moveable 对象的引用，以便后面连接使用
            var m1 = new dojo.dnd.Moveable("note1", {handle : "dragHandle1"});
            var m2 = new dojo.dnd.Moveable("note2", {handle : "dragHandle2"});
```

```
        // 与所有 Moveable 对象相关的系统级的主题
        dojo.subscribe("/dnd/move/start", function(node){
                console.log("Start moving", node);
        });
        dojo.subscribe("/dnd/move/stop", function(node){
                console.log("Stop moving", node);
        });

        // 当便条被移动时，让它突出显示……
        // 连接到 Moveable 对象，而不是原始节点
        dojo.connect(m1, "onMoveStart", function(mover){
                console.log("note1 start moving with mover:", mover);
                dojo.query("#note1 > textarea").addClass("movingNote");
            });
            dojo.connect(m1, "onMoveStop", function(mover){
                    console.log("note1 stop moving with mover:", mover);
                    dojo.query("#note1 > textarea").removeClass("movingNote");
            });
        });

    </script>
</head>
<body>
        <div id="note1" dojoType="dojo.dnd.Moveable">
            <div id='dragHandle1' class="noteHandle"></div>
            <textarea class="note">Note1</textarea>
        </div>
        <div id="note2" dojoType="dojo.dnd.Moveable">
            <div id='dragHandle2' class="noteHandle"></div>
            <textarea class="note">Note2</textarea>
        </div>
</body>
</html>
```

**注意：** 对于dojo.query中的参数"#note1 > textarea"，读者应该还记得它表示id为note1的节点的子元素，即本示例中的textarea节点。要了解dojo.query方法能够接收的CSS选择符语法，请参见表5-1。

在前面的代码中，我们没有连接实际的节点，而是连接的由新的dojo.dnd.Moveable创建的Moveable对象。

正如读者所看到的，既可以利用发布/预订通信模型预订全局拖动事件，也可以通过连接到特定的Moveable节点来响应特定事件。表7-2总结了可以通过dojo.connect连接的事件。

至于发布/预订通信模型，在此则可以使用dojo.subscribe预订"dnd/move/start"和"dnd/move/stop"主题。

表 7-2：Moveable 对象的事件

| 事件 | 说明 |
| --- | --- |
| onMoveStart(/*dojo.dnd.Mover*/mover) | 在每次移动之前调用 |
| onMoveStop(/*dojo.dnd.Mover*/mover) | 在每次移动之后调用 |
| onFirstMove(/*dojo.dnd.Mover*/mover) | 每当第一次移动时调用；便于执行初始化程序 |
| onMove(/*dojo.dnd.Mover*/mover), (/* Object */ leftTop) | 每当发出移动通知时调用；在默认情况下，先调用 onMoving，然后移动 Moveable，再调用 onMoved |
| onMoving(/*dojo.dnd.Mover*/mover), (/*Object*/leftTop) | 在 onMove 之前调用 |
| onMoved(/*dojo.dnd.Mover*/mover), (/*Object */leftTop) | 在 onMove 之后调用 |

## 设置 z-index

前面这个示例可以说越来越完善，但是其中却存在一个潜在的问题，即每个便条元素初始的 z-index 值不会改变：其中一个总是位于上方，而另一个则总是位于下方。如果能让最后选择的便条显示在上方（通过为其设置最高的 z-index）会让人觉得更自然。好在，通过一个函数来调整 z-index 值非常简单，而这个函数则是由于连接到 onMoveStartEvent 事件被触发的。

以下所示的解决方案修改了addOnLoad块中函数的逻辑。由于是通过闭包来捕获状态变量，没有使用模块级变量或全局变量，因而这个方案显得十分优雅：

```
dojo.addOnLoad(function() {
    // 创建并保存到 Moveable 对象的引用，以便后面连接使用
    var m1 = new dojo.dnd.Moveable("note1", {handle : "dragHandle1"});
    var m2 = new dojo.dnd.Moveable("note2", {handle : "dragHandle2"});

    var zIdx = 1; // 利用这个匿名函数的闭包

    dojo.connect(m1, "onMoveStart", function(mover){
        dojo.style(mover.host.node, "zIndex", zIdx++);
    });
    dojo.connect(m2, "onMoveStart", function(mover){
        dojo.style(mover.host.node, "zIndex", zIdx++);
    });
});
```

**警告：** 第2章曾经介绍过，必须为dojo.style传递DOM存取器（accessor）格式的属性，而非样式表格式的属性。因此，在此设置"z-index"样式属性不会起作用。

## 约束Moveable对象

不费吹灰之力就能做到在屏幕上毫无约束地移动对象固然好，但读者很快就会发现有必要编写一些逻辑，以便定义边界、限制重叠或者对Moveable对象实施其他约束。所幸的是，拖放模块为我们编写约束逻辑提供了辅助机制，可以有效减少冗余代码。

拖放模块（dojo.dnd）中提供了3种用于约束可移动对象的机制：一是编写自定义约束函数，以便动态计算边界盒子的范围（即创建constrainedMoveable对象）；二是在创建可移动对象时定义静态的边界盒子（即创建boxConstrainedMoveable对象）；三是在父节点定义的边界内约束可移动对象（即创建parentConstrainedMoveable对象）。以上每种类型的边界盒子都遵循第2章“盒模型”中介绍的约定。

首先，我们从创建constrainedMoveable对象开始。以下是对前面粘性便条示例修改后的代码：

```
<html>
    <head>
       <title>Moving Around</title>
        <style type="text/css">
            .note {
                background: #FFFFBF;
                border-bottom: 1px solid black;
                border-left: 1px solid black;
                border-right: 1px solid black;
                width: 302px;
                height: 300px;
                margin : 0px;
                padding : 0px;
            }
            .noteHandle {
                border-left: 1px solid black;
                border-right: 1px solid black;
                border-top: 1px solid black;
                cursor :pointer;
                background: #FFFF8F;
                width : 300px;
                height: 10px;
                margin : 0px;
                padding : 0px;
            }
            .movingNote {
```

```
            background : #FFFF3F;
        }
        #note1, #note2 {
                width : 302px
        }
    </style>

    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.dnd.Moveable");
        dojo.require("dojo.dnd.move");

        dojo.addOnLoad(function() {
            var f1 = function() {
                //通过巧妙计算来定义边界盒子。
                //将note1的可移动范围限制在note2的右/下方50像素之内
                var mb2 = dojo.marginBox("note2");
                b = {};
                b["t"] = 0;
                b["l"] = 0;
                b["w"] = mb2.l + mb2.w + 50;
                b["h"] = mb2.h + mb2.t + 50;
                return b;
            }

            var m1 = new dojo.dnd.move.constrainedMoveable("note1",
               {handle : "dragHandle1", constraints : f1, within : true});

            var m2 = new dojo.dnd.Moveable("note2", {handle : "dragHandle2"});

            var zIdx = 1;
                dojo.connect(m1, "onMoveStart", function(mover){
                    dojo.style(mover.host.node, "zIndex", zIdx++);
                });
                dojo.connect(m2, "onMoveStart", function(mover){
                    dojo.style(mover.host.node, "zIndex", zIdx++);
                });
            });

        </script>
    </head>
    <body>
            <div id="note1">
                <div id='dragHandle1' class="noteHandle"></div>
                <textarea class="note">Note1</textarea>
            </div>
            <div id="note2">
                <div id='dragHandle2' class="noteHandle"></div>
```

```
                <textarea class="note">Note2</textarea>
            </div>
    </body>
</html>
```

**警告：** 在计算Moveable对象的边界盒子时，必须明确定义在屏幕上移动对象的最外围容器的高度和宽度。例如，不限制粘性便条最外围div元素的宽度会导致不正确的结果，因为可移动的div实际上要比我们在屏幕上看到的黄色盒子宽得多。因而，使用其外边距盒子计算约束条件不会得到想要的结果。

简单地说，首先要明确定义便条对象最外围div元素的边界，以便通过dojo.marginBox精确计算出其外边距盒子的宽度和高度。然后，再自定义一个约束函数，用以将note1的移动范围限制在note2的右侧和下方50像素之内。

**警告：** 如果使用constrainedMoveable却未指定约束函数，那么会导致许多错误。因此，如果读者不想使用约束函数，那么最好直接使用Moveable。

接下来，我们看一看如何使用boxConstrainedMoveable来定义静态边界。定义静态边界不需要传递自定义约束函数，只需传递一个明确的范围。请读者将前面示例中有关note2的代码修改为如下代码，然后在浏览器中查看效果：

```
var m2 = new dojo.dnd.move.boxConstrainedMoveable("note2",
{
    handle : "dragHandle2",
    box : {l : 20, t : 20, w : 500, h : 300}
});
```

结果与前一个示例似乎相差无几，但note2已经不能移动出上面定义的约束盒子了。

最后，创建parentConstrainedMoveable对象也同样简单。只要定义可移动对象，并确保其父节点能够提供足够的空间即可。在标记中添加一个父节点并不影响Dojo代码。下面是前面示例的再次修改版：

```
<!-- ……省略的代码…… -->
.parent {
    background: #BFECFF;
    border: 10px solid lightblue;
    width: 400px;
    height: 700px;
    padding: 10px;
    margin: 10px;
}
```

```
<!-- ……省略的代码…… -->
<script type="text/javascript">
    dojo.require("dojo.dnd.move");

    dojo.addOnLoad(function() {
        new dojo.dnd.move.parentConstrainedMoveable("note1",
            {
                handle : "dragHandle1", area: "margin", within: true
            });
        new dojo.dnd.move.parentConstrainedMoveable("note2",
            {
                handle : "dragHandle2", area: "padding", within: true
            });
        });
    </script>
</head>
<body>
    <div class="parent">
        <div id="note1">
            <div id='dragHandle1' class="noteHandle"></div>
            <textarea class="note">Note1</textarea>
        </div>
        <div id="note2">
            <div id='dragHandle2' class="noteHandle"></div>
            <textarea class="note">Note2</textarea>
        </div>
    </div>
</body>
</html>
```

传递给parentConstrainedMoveable的area参数最值得关注。可以为该参数指定的值包括"margin"、"padding"、"content"和"border"，分别用于将Moveable对象的移动范围限定在父节点的外边距、内边距、内容区和边框区之内。

**注意：** 同使用基本的Moveable创建可移动对象一样，在以上示例中也可以通过发布/预订通信模型来检测全局拖放事件。另外，由于constrainedMoveable和boxConstrainedMoveable都继承自Moveable，因此dojo.connect和dojo.subscribe中所用的事件名称也与表7-2中针对Moveable对象的事件名称相同。

# 放置

到目前为止，本章只介绍了如何实现在屏幕上拖动对象。本节将讨论拖放的另一部分——放置。首先，我们来看一看dojo.dnd.Source这个用于提供拖放源的特殊容器类。Source对象虽然同样可以作为放置的目标，但稍后我们就会看到，通过dojo.dnd.

Target则能够指定一个“纯粹的”目标。Source对象既可以作为来源也可以作为目标，但 Target 对象则只能作为目标。

创建Source对象与创建Moveable对象类似，即在调用Source构造函数时也要将一个节点作为第一个参数，将一个对象直接量作为第二个参数。表7-3中列出了相关的方法。

**表 7-3：创建和销毁 Source 对象的方法**

| 名称 | 说明 |
| --- | --- |
| dojo.dnd.Source(/*DOMNode*/node, /*Object*/params) | 用于创建 Source 对象的构造函数。其中，params 参数可取的值参见表 7-1 |
| destroy() | 解除对象的引用，以便垃圾回收机制将其销毁 |

表 7-4 总结了创建 Source 对象时可能用到的关键字参数。

**表 7-4：表 7-3 中 params 参数可用的配置参数**

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| isSource | 布尔值 | 默认值为 true；如果指定为 false，则忽略拖动操作 |
| horizontal | 布尔值 | 默认值为 false；如果指定为 true，则构建水平布局（要求是行内 HTML 元素） |
| copyOnly | 布尔值 | 默认值为 false；如果指定为 true，则复制而不是移动对象（无须按住 Ctrl 键） |
| skipform | 布尔值 | 默认值为 false；与 Moveable 类似，用于控制基于文本的表单元素是否可编辑 |
| withHandles | 布尔值 | 默认值为 false；如果指定为 true，则只能通过手柄拖动对象 |
| accept | 数组 | 默认值为["text"]。用于指定可以放置的对象类型 |

Source对象的一个常见的用途，就是在实现通过拖放重排列表项顺序时，消除繁琐的编码工作量。请看下面的代码示例：

```
<html>
    <head>
        <title>Fun with Source!</title>
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dnd.css" />
```

```
        <script
            type="text/javascript"
            djConfig="parseOnLoad:true"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.dnd.Source");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>
        <div dojoType="dojo.dnd.Source" class="container">
            <div class="dojoDndItem">foo</div>
            <div class="dojoDndItem">bar</div>
            <div class="dojoDndItem">baz</div>
            <div class="dojoDndItem">quux</div>
        </div>
    </body>
</html>
```

乍一看，似乎这个简单的示例并不起眼，但其中却浓缩了需要大量编码才能实现的功能。首先，我们注意到其中唯一与Dojo有关的类是Source。要创建一个包含多个项的容器，并且让这些项在其中可以拖放，只需为相应的容器元素指定dojoType属性，并确保该元素能够被解析即可。同其他示例类似，解析元素由djConfig配置项中的parseOnLoad参数负责。

接下来，我们再全面了解一下这个示例。因为能够拖放单个列表项是显而易见的，但恐怕你还没有注意到Source对象提供的以下更多功能：

- 单击：选中一个元素并取消对其他元素的选中。
- Ctrl-单击：切换元素的选中状态并允许连续选中多个元素；也可以在选中多个元素的情况下，取消对个别元素的选中。
- Shift-单击：选中最近一次选中的元素和当前被单击元素之间的元素。取消最近一次选中元素之前所选元素的选中状态。
- Ctrl-Shift-单击：选中最近一次选中的元素和当前被单击元素之间的元素，但保留最近一次选中元素之前所选元素的选中状态。
- 在放置时按住Ctrl键会复制选项。图7-1展示了其中一些操作。

## 纯粹的目标

本章前面曾经提到过，适合使用Target作为放置目标的情况也很多，即一旦将元素放

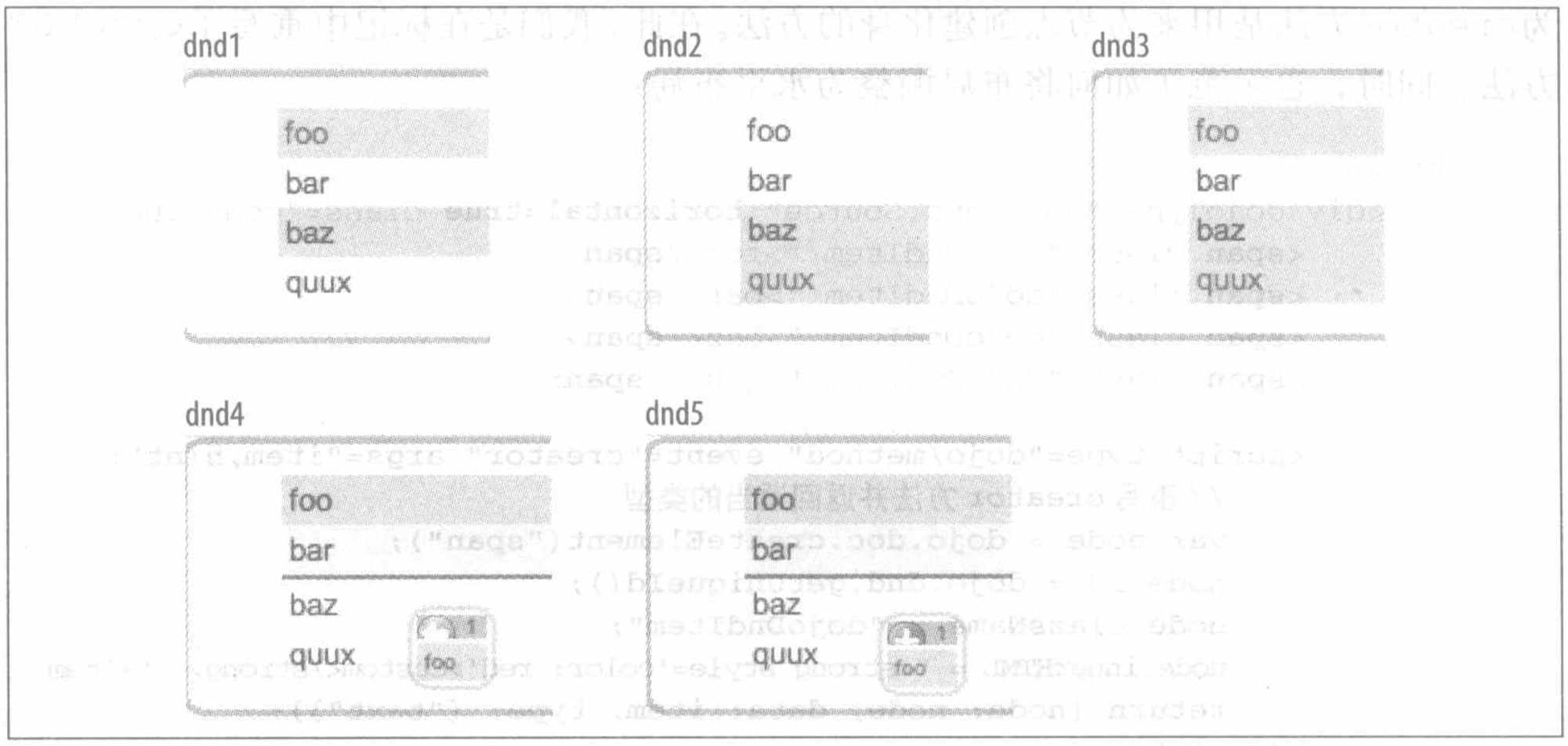



图 7-1：dnd1 所示为使用 Ctrl- 单击最初选中的元素；dnd2 为对 quux 执行 Shift- 单击之后的结果；dnd3 为对 quux 执行 Ctrl-Shift- 单击之后的结果；dnd4 是在未按 Ctrl 键的情况下拖动元素的情景；dnd5 是在按 Ctrl 键时拖动导致元素被复制的情景

入其中，就不能再被移动或重新排序了。以下是对前面示例代码稍加修改后的结果，其中使用了 Target 对象。

```
<body>
    <div dojoType="dojo.dnd.Source" class="container">
        <div class="dojoDndItem">foo</div>
        <div class="dojoDndItem">bar</div>
        <div class="dojoDndItem">baz</div>
        <div class="dojoDndItem">quux</div>
    </div>
    <!-- 添加到 dojo.dnd.Target 中的项不能再被移动或重新排序 -->
    <div dojoType="dojo.dnd.Target" class="container"></div>
</body>
```

相信读者已经了解到，这寥寥几行代码其实封装了大量的功能。另外，虽然这里使用div元素作为示例，但使用其他HTML元素也能得到相同的结果。在实践中，使用最多的还是通过 ul 和 li 元素定义的无序列表。

## 自定义化身

当拖动 Source 中的项时，屏幕上临时出现的小图标就是化身（avatar）。虽然 Dojo 工具箱中提供的标准化身也不错，但自定义化身的需求仍然很常见。以下代码对前面的示例进行了一番调整，并通过重写creator方法实现了对化身文本和样式的自定义，；因

为creator方法是用来为节点创建化身的方法。在此，我们是在标记中重写了creator方法。同时，也示范了如何将布局调整为水平布局：

```
<body>
    <div dojoType="dojo.dnd.Source" horizontal=true class="container">
        <span class="dojoDndItem ">foo</span>
        <span class="dojoDndItem ">bar</span>
        <span class="dojoDndItem ">baz</span>
        <span class="dojoDndItem ">quux</span>

        <script type="dojo/method" event="creator" args="item,hint">
            //重写creator方法并返回适当的类型
            var node = dojo.doc.createElement("span");
            node.id = dojo.dnd.getUniqueId();
            node.className = "dojoDndItem";
            node.innerHTML = "<strong style='color: red'>Custom</strong>  "+item;
            return {node: node, data: item, type: ["text"]};
        </script>
    </div>
    <div dojoType="dojo.dnd.Target" horizontal=true class="container"></div>
</body>
```

在重写creator方法时，我们为它传递了两个参数：item和hint，分别表示实际被移动的项和说明要创建的外观的“提示”值。除非读者自己实现底层机制，否则hint的值将始终是"avatar"。最后，creator方法必须返回一个表示item的对象，该对象中包含实际的DOM节点、数据表示（representation）以及表示的类型。前面提到过，Source对象接受的默认表示类型为"text"。

## 放置事件

通过dojo.subscribe和dojo.connect预订及连接事件也像用Moveable对象一样容易。表7-5总结了发布/预订以及连接通信模型涉及的公共事件，相应的代码示例随后。

表7-5：放置事件

| 类型 | 事件 | 参数 | 说明 |
| --- | --- | --- | --- |
| subscribe | "/dnd/source/over" | /* Node */ source | 当鼠标移动到一个Source容器上时发布；参数source指的就是该容器。当鼠标离开该Source容器时发布另一个/dnd/source/over主题，但source参数的值为null |

表7-5：放置事件（续）

| 类型 | 事件 | 参数 | 说明 |
| --- | --- | --- | --- |
| subscribe | "/dnd/start" | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | 当拖动开始时发布。参数source指的就是作为放置操作来源的Source容器。参数copy为true表示复制操作，为false表示移动操作。参数nodes是放置操作涉及的元素数组 |
| subscribe | "/dnd/drop" | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | 当放置发生（拖动正式结束）时发布。参数source指的是作为放置操作来源和目标的Source容器。参数copy为true表示复制操作，为false表示移动操作。参数nodes是放置操作涉及的元素数组 |
| subscribe | "/dnd/cancel" | N/A | 当放置操作被取消时发布（例如，按下Esc键） |
| connect | onDndSourceOver | /* Node */ source | 当鼠标移动到Source容器上时调用；参数source指的就是该容器。当鼠标离开该Source容器时调用另一个onDndSourceOver事件，但参数source的值为null |
| connect | onDndStart | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | 当拖动开始时调用。参数source指的就是作为放置操作来源的Source容器。参数copy为true表示复制操作，为false表示移动操作。参数nodes是放置操作涉及的元素数组 |

表7-5：放置事件（续）

| 类型 | 事件 | 参数 | 说明 |
| --- | --- | --- | --- |
| connect | onDndDrop | /* Node */ source<br>/* Array */ nodes<br>/* Boolean */ copy | 当放置发生（拖动正式结束）时调用。参数source指的是作为放置操作来源和目标的Source容器。参数copy为true表示复制操作，为false表示移动操作。参数nodes是放置操作涉及的元素数组 |
| connect | onDndCancel | N/A | 当放置操作被取消时调用（例如，按下Esc键） |

接下来，请读者把下面完整的示例加载到浏览器中，并通过Firebug检查预订的多个主题在控制台中的输出结果。注意，可以在不同的Source容器间拖放元素。图7-2展示了结果。真的很绝啊！

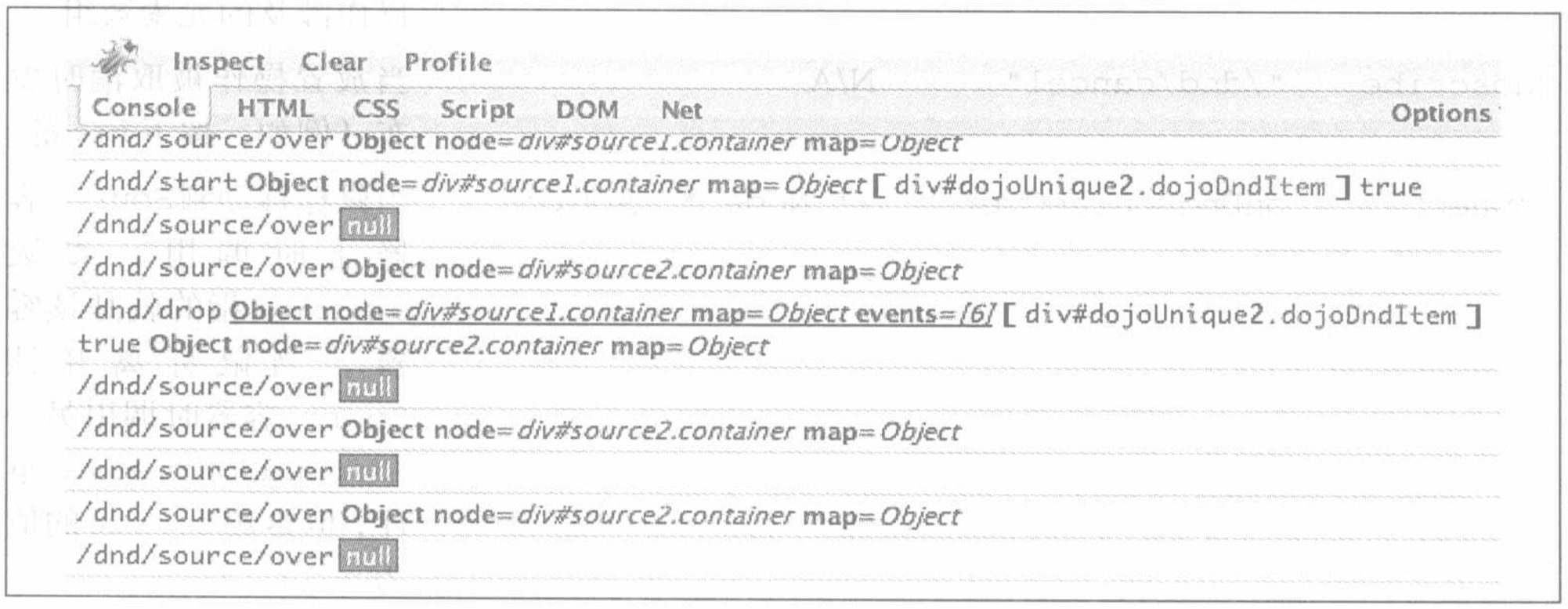



图7-2：Firebug非常适合摸清拖放操作的内部线索

```
<html>
    <head>
        <title>More Fun with Drop!</title>
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
```

```
        href="http://o.aolcdn.com/dojo/1.1/dojo/tests/dnd/dndDefault.css" />
    <script
        type="text/javascript"
        djConfig="parseOnLoad:true"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
        dojo.require("dojo.dnd.Source");
        dojo.require("dojo.parser");

        dojo.addOnLoad(function() {
            dojo.subscribe("/dnd/source/over", function(source) {
                console.log("/dnd/source/over", source);
            });
            dojo.subscribe("/dnd/start", function(source, nodes, copy) {
                console.log("/dnd/start", source, nodes, copy);
            });
            dojo.subscribe("/dnd/drop", function(source, nodes, copy) {
                console.log("/dnd/drop", source, nodes, copy);
            });
            dojo.subscribe("/dnd/cancel", function() {
                console.log("/dnd/cancel");
            });
        });
    </script>
</head>
<body>
    <div id="source1" dojoType="dojo.dnd.Source" class="container">
        <div class="dojoDndItem">foo</div>
        <div class="dojoDndItem">bar</div>
        <div class="dojoDndItem">baz</div>
        <div class="dojoDndItem">quux</div>
    </div>
    <div id="source2" dojoType="dojo.dnd.Source" class="container">
        <div class="dojoDndItem">FOO</div>
        <div class="dojoDndItem">BAR</div>
        <div class="dojoDndItem">BAZ</div>
        <div class="dojoDndItem">QUUX</div>
    </div>
</body>
</html>
```

下面，我们要在另一个不同的addOnLoad块中示范连接。由于需要对Source对象（而非DOM节点）的引用，因此必须通过编程方式创建Source，而不能依赖解析器对在标记中定义的部件进行实例化。请读者将前一示例中的addOnLoad代码块替换成如下代码，并去掉djConfig的parseOnLoad参数，然后再次检查Firebug控制台的输出：

```
dojo.addOnLoad(function() {
        // 保存对 Source 对象的引用，以便后面连接时使用
```

```
var s1 = new dojo.dnd.Source("source1");

dojo.connect(s1, "onDndSourceOver", function(source) {
    console.log("onDndSourceOver for", s1, source);
});
dojo.connect(s1, "onDndStart", function(source, nodes, copy) {
    console.log("onDndStart for ", s1, source, nodes, copy);
});
dojo.connect(s1, "onDndDrop", function(source, nodes, copy, target) {
    console.log("onDndDrop for",  s1, source, nodes, copy, target);
});
dojo.connect(s1, "onDndCancel", function() {
    console.log("onDndCancel for ", s1);
});
```

## 以编程方式操作可放置对象

通过前面的示例我们知道，可以使用Source构造函数将节点转换为可放置。但除此之外，以编程方式还能实现更多功能。表7-6总结了dojo.dnd中一个较底层的类Selector所提供的功能。由于Source继承自Selector，因此表中的方法都可以通过Source对象调用，当然，通过Selector本身调用这些方法同样有它们的用途。

表7-6：Selector API

| 方法 | 说明 |
| --- | --- |
| getSelectedNodes() | 返回被选中的节点数组 |
| selectNone() | 取消对所有节点的选择 |
| selectAll() | 选择所有节点 |
| deleteSelectedNodes() | 删除所有选中的节点 |
| insertNodes(/* Boolean */ addSelected, /* Array */ data, /* Boolean */ before, /* Node */ anchor) | 插入节点数组，通过addSelected可以指定是否选中节点。如果未提供anchor参数，那么节点被插入到Selector的第一个子元素之前。否则，根据before的值决定将它们插入到anchor节点之前或者之后 |
| destroy() | 将对象准备好以供垃圾收集机制回收 |
| onMouseDown(/* Object */ event) | 可以通过dojo.connect连接检测onmousedown事件，但应该优先考虑更高级的onDnd方法。参数event提供标准事件信息 |

表 7-6：Selector API（续）

| 方法 | 说明 |
| --- | --- |
| onMouseUp(/* Object */ event) | 可以通过 dojo.connect 连接检测 onmouseup 事件，但应该优先考虑更高级的 onDnd 方法。参数 event 提供标准事件信息 |
| onMouseMove(/* Object */ event) | 可以通过 dojo.connect 连接检测鼠标移动，但应该优先考虑更高级的 onDnd 方法。参数 event 提供标准事件信息 |
| onOverEvent(/* Object */ event) | 可以通过 dojo.connect 连接检测到鼠标是否进入了相应区域，但应该优先考虑更高级的 onDnd 方法。参数 event 提供标准事件信息 |
| onOutEvent(/* Object */ event) | 可以通过 dojo.connect 连接检测到鼠标是否离开了相应区域，但应该优先考虑更高级的 onDnd 方法。参数 event 提供标准事件信息 |

**注意：** 要想了解真正具有实用价值的拖放示例，请参考第 15 章 “在 Tree 中使用拖放”，该示例展示了在操作 Tree 部件时如何利用拖放模块解决实际问题。对于 Tree 这种杰出的技术而言，dojo.dnd 模块只能使其变得更强大。

## 小结

在学习完本章后，读者应该能够：

- 构建不受约束的 Moveable 对象，并在屏幕上拖动它们。
- 为 Moveable 对象定义约束以控制它们的行为。
- 实现 Source 和 Target 容器并创建可拖放的选项，以便在容器内部、容器之间拖放这些选项。
- 创建自定义化身，以便为用户给出放置操作的提示。
- 使用 dojo.connect 和 dojo.subscribe 接收拖放对象的事件通知。

下一章，我们将讨论动画和特效。

# 第8章

# 动画和特效

平淡无奇的应用程序会因为有了动画而变得生动活泼。本章将系统介绍Base中以及Core中的dojo.fx（其中fx的发音为“effects”）模块提供的与动画相关的实用方法。本章的内容和源代码与前几章关联不多，因此基本上可以把本章当作一个独立的参考。

## Base中的动画方法

Dojo工具箱在Base中提供了基本的动画方法，在dojo.fx模块中提供了增补的功能。Base中的动画功能主要来自_Animation类，这个具有委托能力的类会根据相应配置触发回调函数，而被触发的回调函数则通过操作节点的属性来生成动画。在创建_Animation类的实例之后，接着调用该实例的play方法就可以产生动画效果了。

---

**警告：** _Animation类前面的下划线至少表明了以下两点：

- 这个API还没有最终停止更新，虽然它已经非常稳定并且从1.1版到最终版本也不会再有太大变化（如果有变化的话）；
- 一般来说，不能直接对_Animation进行实例化，而要通过Base和dojo.fx提供的辅助方法来创建、包装和操作它的对象。不过，通常都需要运行它的play方法才能启动动画。

---

## 简单的淡入淡出

在展示高级动画功能之前，我们先看一个大概是最简单的示例：屏幕上有一个方块，当用户单击它时就会淡出屏幕。这个示例使用了Base中的淡出方法fadeOut。fadeOut

及其对应方法fadeIn都接收3个关键字参数，如表8-1所示。另外，图8-1展示了fadeOut使用的默认缓动（easing）函数。

表8-1：Base中淡入淡出方法的参数

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| node | DOM Node | 产生淡入淡出动画的节点 |
| duration | 整数 | 淡入淡出持续的毫秒数。默认值为350 |
| easing | 函数 | 用于调整动画执行过程中加速及减速的函数。默认值为(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)。注意，对于fadeIn和fadeOut方法而言，这个缓动函数定义的范围仅介于0～1 |

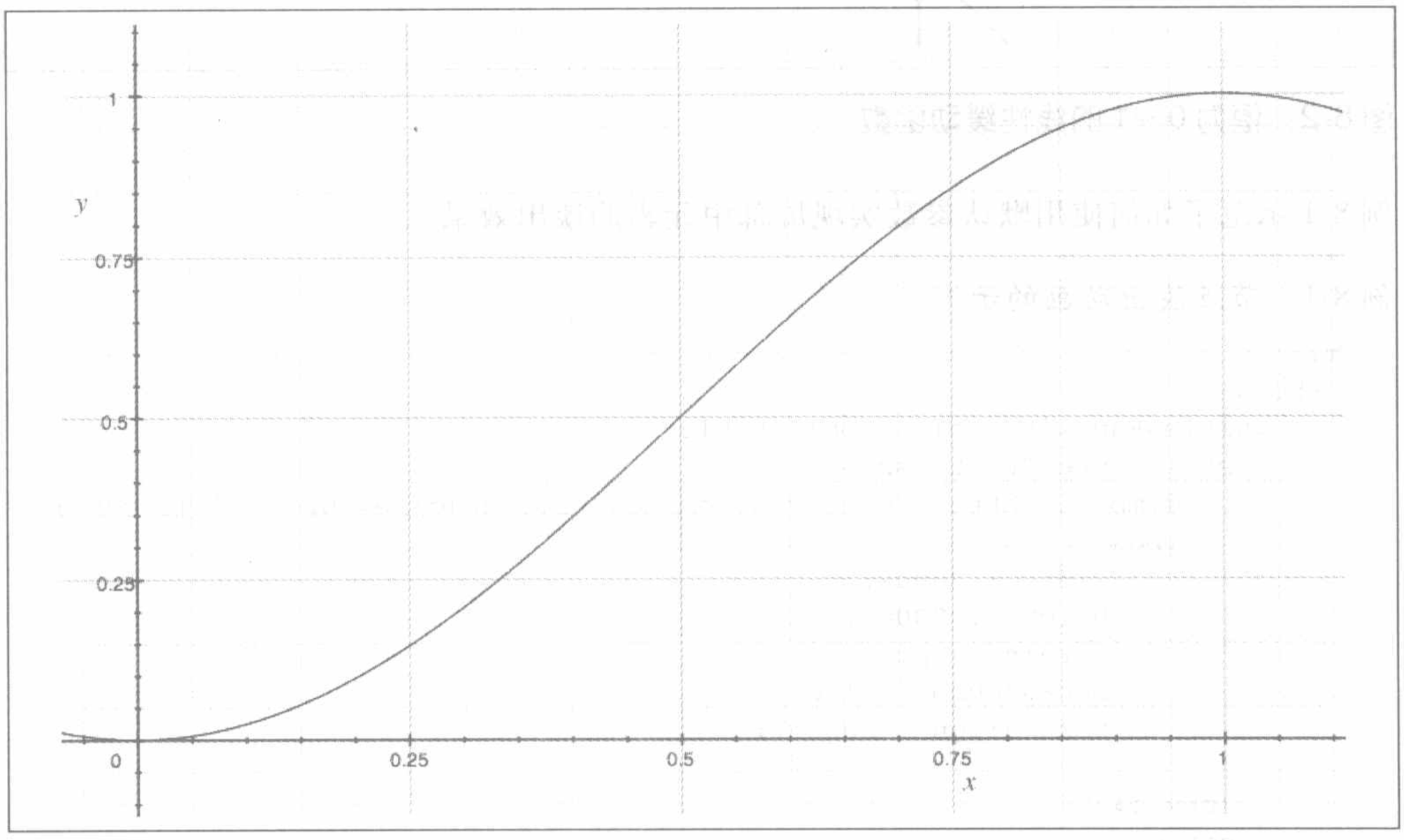



图8-1：默认缓动函数的曲线图；fadeIn和fadeOut方法默认的缓动函数只定义了从0～1的范围

在这3个参数中，node和duration没什么难理解的，倒是easing参数表示的缓动函数让人有点不好琢磨。简单地说，所谓缓动函数就是一个用于控制动画效果（即这里的_Animation对象）速度变化的函数。如缓动函数function(x) { return x; }是线性的：对于每个输入的值，都会返回相同的值。因而，对于0～1之间的小数而言，该

函数总会返回相同的值。如果把这个函数绘制成图形，那么就是一条具有固定斜率的直线，如图 8-2 所示。这个固定的斜率会导致平滑和匀速的动画效果。

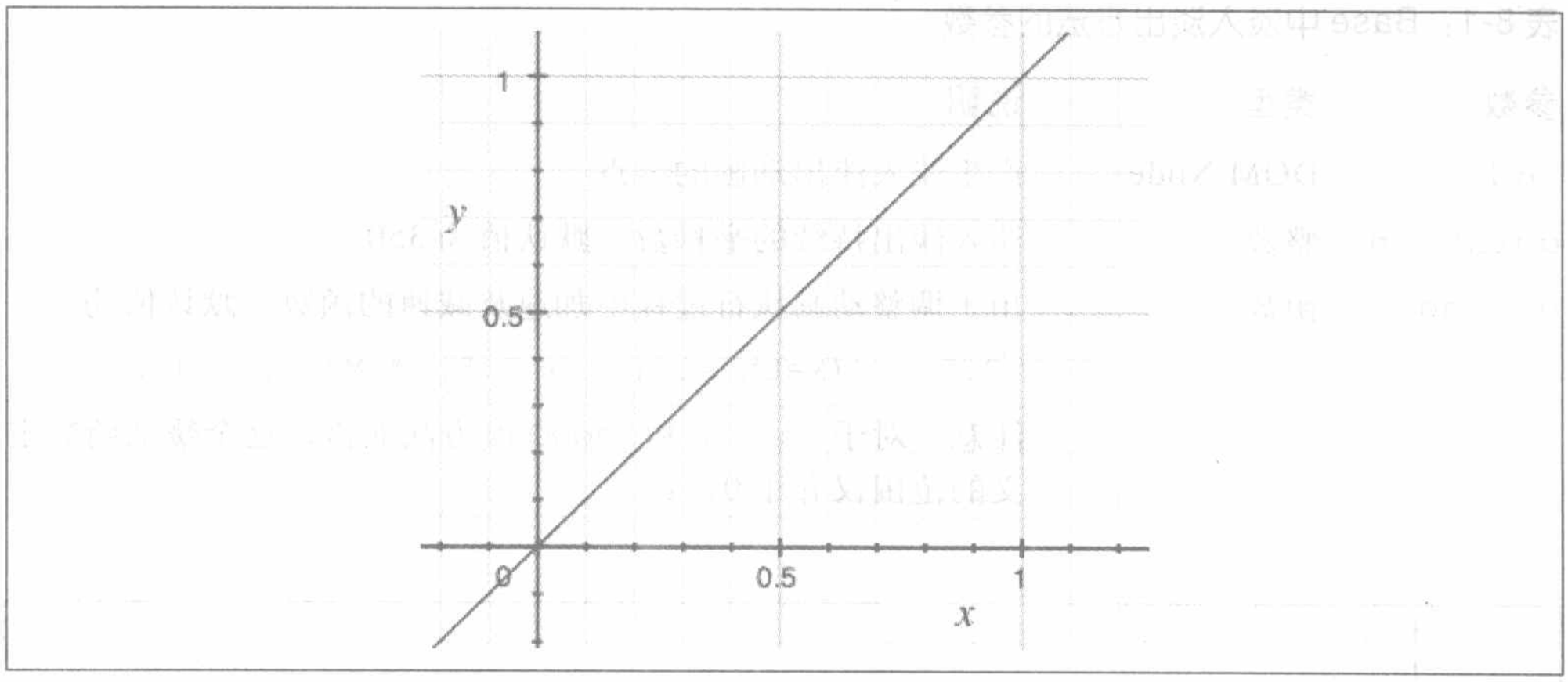



图 8-2：值为 0～1 的线性缓动函数

例 8-1 示范了如何使用默认参数实现屏幕中元素的淡出效果。

例 8-1：节点淡出动画的示例

```
<html>
    <head>
       <title>Fun with Animation!</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                margin : 5px;
                background : blue;
                text-align : center;
            }
    </style>
    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>
    <script type="text/javascript">
        dojo.addOnLoad(function() {
            var box = dojo.byId("box");
            dojo.connect(box, "onclick", function(evt) {
                var anim = dojo.fadeOut({node:box});
                anim.play();
            });
```

```
            });
        </script>
    </head>
    <body>
        <div id="box" class="box">Fade Me Out</div>
    </body>
</html>
```

下面，我们使用另一个缓动函数来改变fadeOut方法的默认行为，该缓动函数的曲线图如图8-3所示，相应的addOnLoad代码块如下所示。注意，默认的缓动函数在从0～1的变化过程中相对平滑，而自定义的缓动函数则把所有动画效果都延迟到了最后。此外，这个示例还使用点运算符基于_Animation对象而不是它的引用直接执行了play方法，这样显得既清晰又自然。

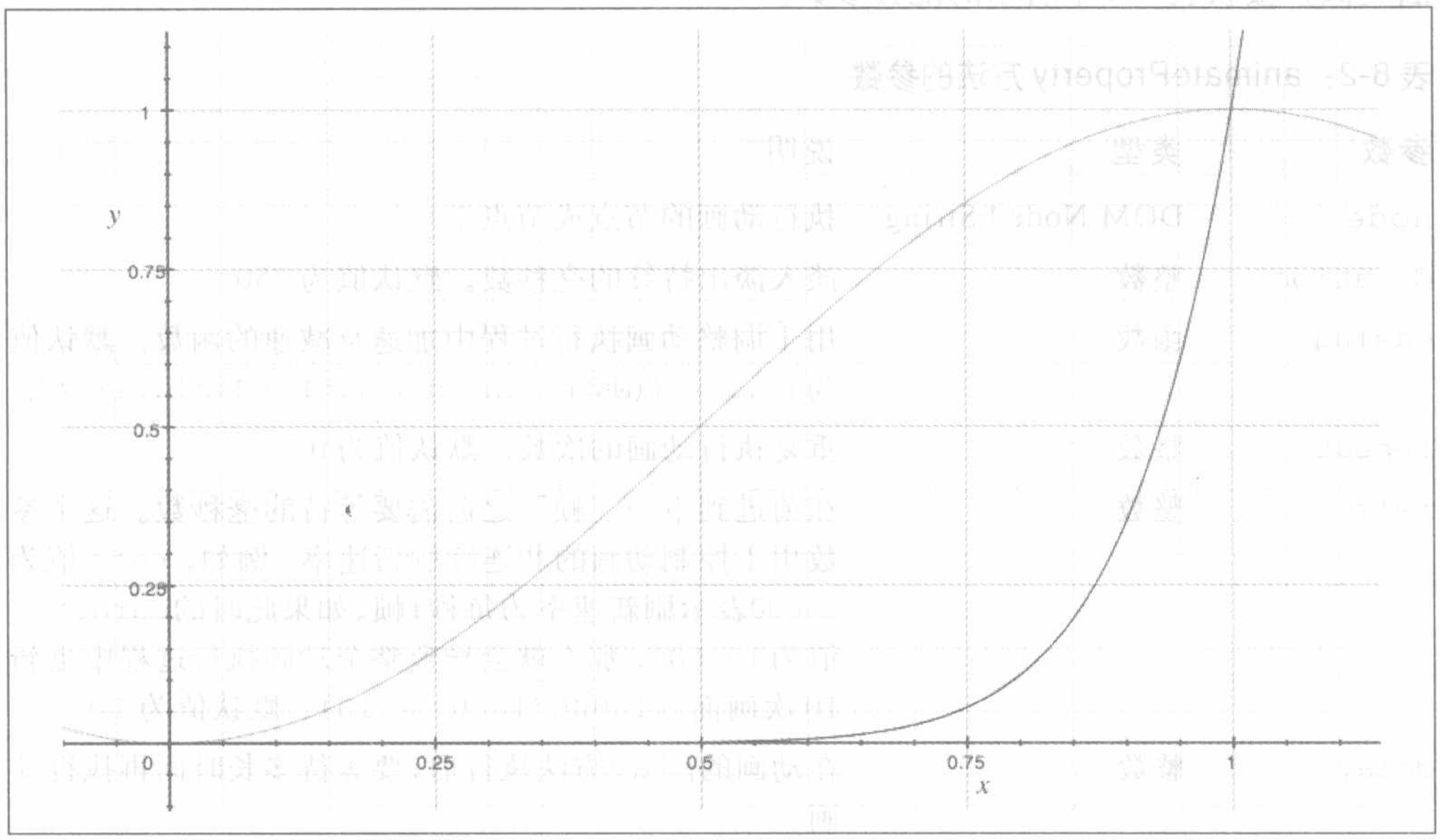



图8-3：自定义缓动函数与默认缓动函数的比较

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
   dojo.connect(box, "onclick", function(evt) {
        var easingFunc = function(x) {
            return Math.pow(x,10);
        }
        dojo.fadeOut({
            node:box,
            easing : easingFunc,
            duration : 3000
        }).play();
}); });
```

**注意：** dojo.fx.easing模块中包含许多极好的缓动函数。读者如果想通过它们来激发自己的创造力，可以抽时间尝试一下。

这个简单的淡出动画固然不错，而且调用Base中的一个方法就能实现该效果也确实令人惊叹。可是，相信读者不会就此满足，一定是想知道_Animation还能提供哪些绝妙的动画效果。

## CSS属性动画

接下来，我们就介绍另一个方法animateProperty，该方法与fadeIn和fadeOut方法类似，接收表8-2中所示的配置参数。

表8-2：animateProperty方法的参数

| 参数 | 类型 | 说明 |
|---|---|---|
| node | DOM Node \| String | 执行动画的节点或节点id |
| duration | 整数 | 淡入淡出持续的毫秒数。默认值为350 |
| easing | 函数 | 用于调整动画执行过程中加速及减速的函数。默认值为(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2) |
| repeat | 整数 | 重复执行动画的次数。默认值为0 |
| rate | 整数 | 在前进到下一“帧”之前需要等待的毫秒数。这个参数用于控制动画的非连续刷新速率。例如，rate值为1 000表示刷新速率为每秒1帧。如果此时的duration值为10 000，那么就会导致整个动画执行过程中更新10次画面（10000/1000 = 10）。默认值为10 |
| delay | 整数 | 在动画的play方法执行后，要等待多长时间再执行动画 |
| properties | 对象 | 指定发生动画的CSS属性，需要提供起始值（start）、终止值（end）和单位值（unit）。其中，start和end可以是CSS属性的直接值，也可以是用于推导出计算值的函数：<br>start (String)<br>属性的起始值<br>end (String)<br>属性的终止值<br>unit (String)<br>属性的单位类型px（默认值）、em等 |

读者可以用以下修改后的代码代替原来的addOnLoad函数，并在浏览器中测试animateProperty方法的效果。这个示例展示了以动画方式将节点宽度由200像素变化到400像素：

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
            dojo.animateProperty({
              node : box,
              duration : 3000,
              properties : {
                  width : {start : '200', end : '400'}
              }
            }).play();
        });
});
```

由于animateProperty是实现大多数dojo.fx动画的基础，而且它又是我们经常使用的方法，因此读者有必要在此花点时间多试验几次这个方法，并熟练掌握通过它来实现各种创意的方式。事实上，该方法能够以统一的接口接收任何CSS属性。例8-2展示了动画效果会导致页面中的其他内容做同步调整的示例。单击页面中的蓝色盒子会导致它在$x$和$y$双方向上扩展，与此同时，红色盒子和绿色盒子也会相应调整各自的位置。

例8-2：扩展节点大小的示例

```
<html>
    <head>
       <title>More Fun With Animation!</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                margin : 5px;
                text-align : center;
            }
            .blueBox {
                background : blue;
                float : left;
            }
            .redBox {
                background : red;
                float : left;
            }
            .greenBox {
                background : green;
                clear : left;
            }
    </style>
```

```
<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
</script>
<script type="text/javascript">

    dojo.addOnLoad(function() {
        var box = dojo.byId("box1");
        dojo.connect(box, "onclick", function(evt) {
            dojo.animateProperty({
                node : box,
                duration : 3000,
                properties : {
                    height : {start : '200', end : '400'},
                    width : {start : '200', end : '400'}
                }
            }).play();
        });
    });

    </script>
</head>
<body>
    <div id="box1" class="box blueBox">Click Here</div>
    <div id="box2" class="box redBox"></div>
    <div id="box2" class="box greenBox"></div>
</body>
</html>
```

如果读者现在对animateProperty方法的参数还不完全理解，那正好利用前面的示例代码来熟悉各个参数的作用。例如，像下面这样修改animateProperty方法，会导致蓝盒子分10个画面不连续地扩张，而不是前面看到的平滑渐变效果（请记住，duration除以rate得到帧数）。

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            rate : 1000,
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

另外，在使用默认缓动函数的情况下，动画效果非常平滑。不过，读者可以试着自定义一个曲线形状陡峭一些的函数，看看它对动画效果会产生什么影响。例如，下面的代码使用了二次曲线作为缓动函数（如图8-4所示），因此随着范围值的增加，返回值会越来越大，而实际的效果通过10个不连续的画面很容易看清楚：

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            rate : 1000,
            easing : function(x) { return x*x; },
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

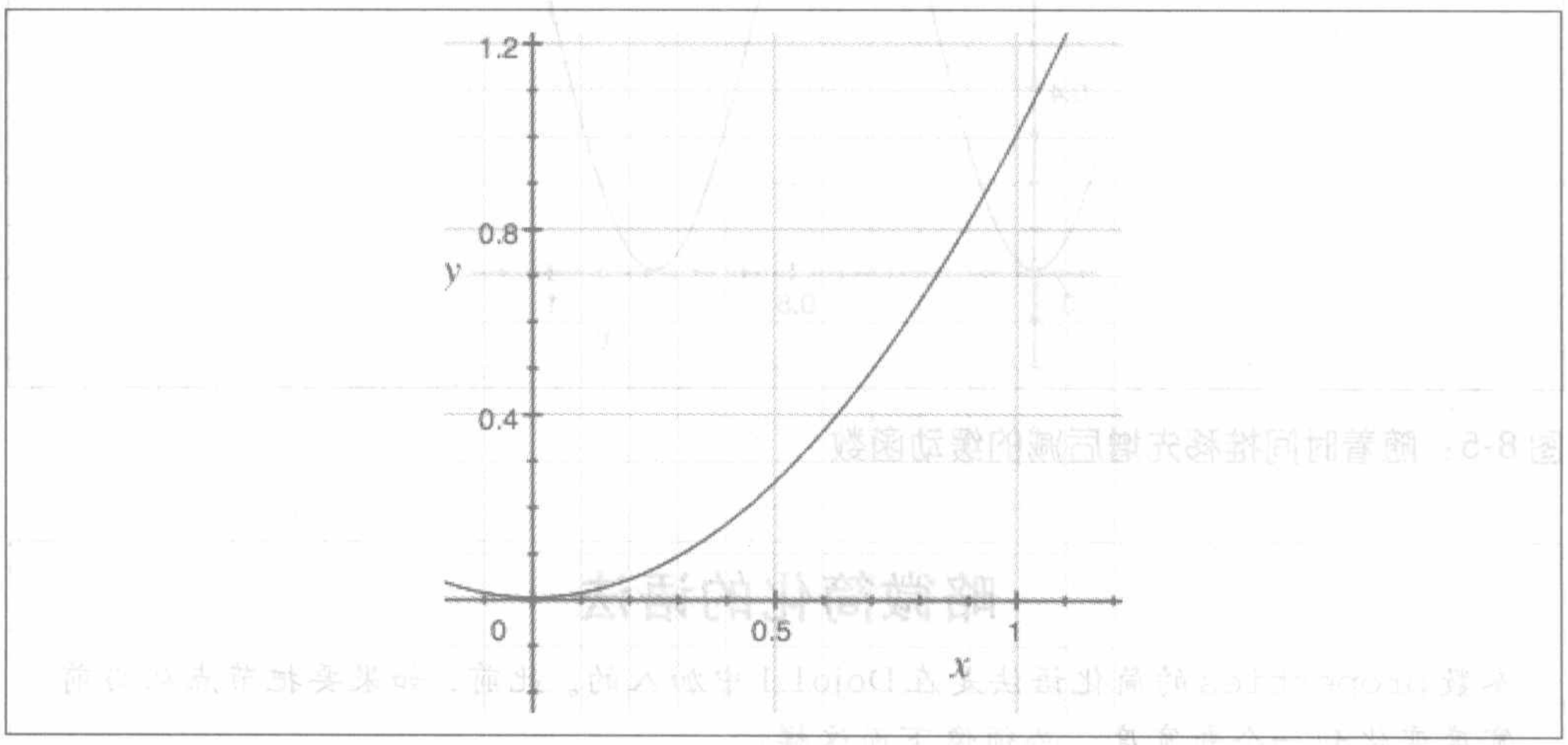



图8-4：以二次曲线作为缓动函数的示例

到目前为止，虽然我们所举例子中的缓动函数都呈单调变化（注1），但事实上也不一定。例如，可以像下面这样使用一个非单调变化的缓动函数（如图8-5所示），再看一看它的效果：

---

注1：　如果函数在任何一个方向上保持相同的变化趋势（如不断增长或不断减少），则该函数就是单调函数。

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box1");
    dojo.connect(box, "onclick", function(evt) {
        dojo.animateProperty({
            node : box,
            duration : 10000,
            easing : function(x) {return Math.pow(Math.sin(4*x),2);},
            properties : {
                height : {start : '200', end : '400'},
                width : {start : '200', end : '400'}
            }
        }).play();
    });
});
```

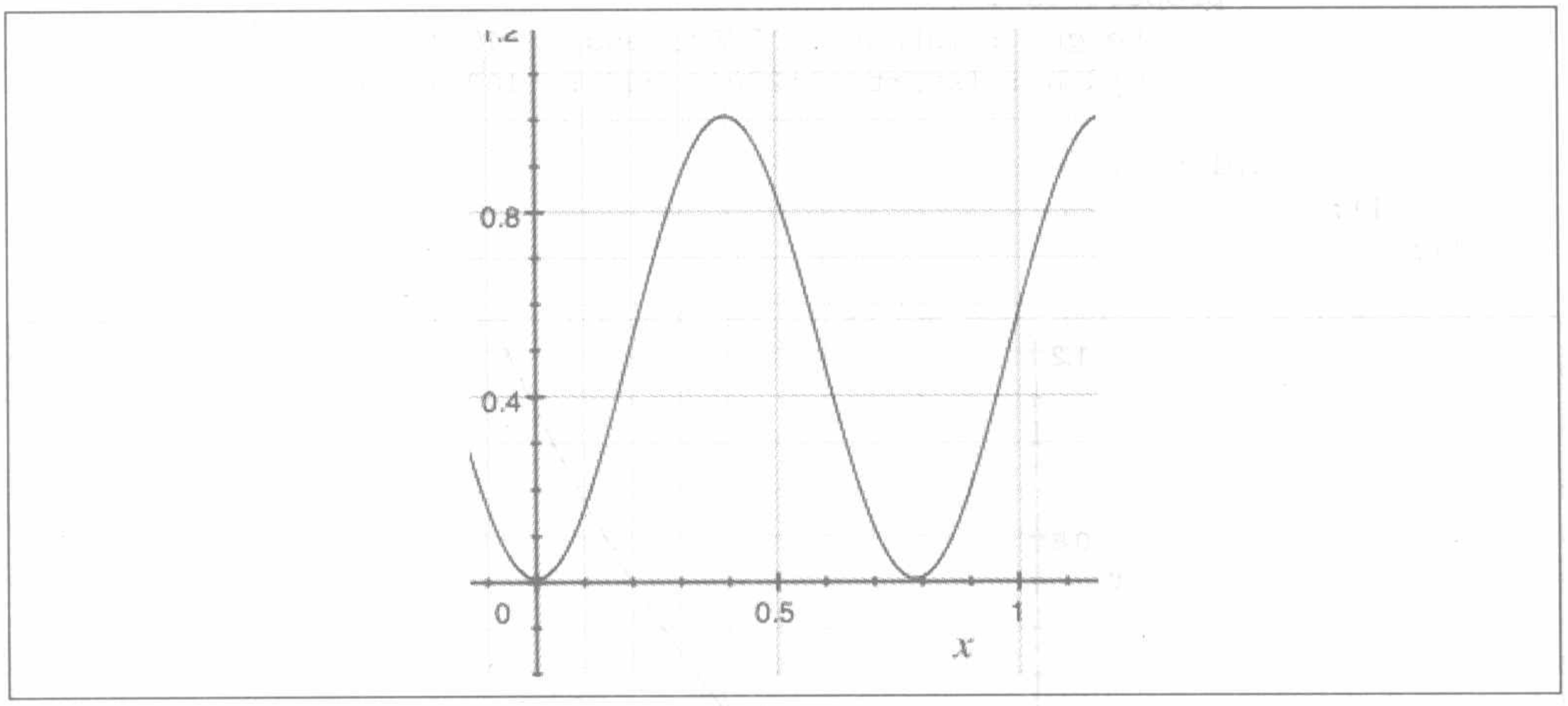



图 8-5：随着时间推移先增后减的缓动函数

## 略微简化的语法

参数properties的简化语法是在Dojo1.1中加入的。此前，如果要把节点从当前宽度变化到一个新宽度，必须像下面这样：

```
dojo.animateProperty({
    node: "foo",
    properties: { width: { end: 500 } } // 多一对花括号
}).play();
```

现在，如果只为properties提供了一个整数值，那么就相当于指定了end。也就是说，前面的代码也可以写成这样：

—待续—

```
    dojo.animateProperty({
        node: "foo", properties: { width: 500 } //少一对花括号
    }).play();
```

此外，Dojo1.1中还新增了另一个方法dojo.anim，该方法提供了两点增强：首先，它的工作原理类似于animateProperty，但是却能够自动播放动画，即无须明确调用play函数；其次，当该方法返回_Animation对象时，动画其实已经就开始了。（再调用play()只会执行空操作。）另外，还有两个常用的属性被从properties对象参数中拿到外面作为位置参数。

结果，这个方法的完整签名如下所示：

```
dojo.anim(/*DOMNode|String*/node, /*Object*/props, /*Integer?*/duration,
/*Function?*/easing, /*Function?*/onEnd, /*Integer?*/delay)
//返回播放中的_Animation
```

假如这些新添加的增强令你感到有些无所适从，那么大可不必马上使用它们；毕竟，这些增强只是为了让人感觉更方便，而不是更迷惑。

## 以编程方式控制动画

虽然在开发中一般不会创建原始的_Animatioin对象，但对它们的控制手段仍然很完备。例如，对于正在播放的动画，我们可以将它暂停、重启、提前终止、获取其状态，或者让它在特定点上给出提示。表8-3中列出了_Animation为实现上述常见任务提供的方法。

表8-3：_Animation提供的控制方法

| 方法 | 参数 | 说明 |
| --- | --- | --- |
| stop | /* Boolean */ goToEnd | 终止动画。如果goToEnd值为true，那么_Animation会前进到终点，因而再次播放动画时将从起点开始。goToEnd的默认值为false |
| pause | N/A | 暂停动画 |
| play | /* Integer */ delay<br>/* Boolean */ goToStart | 播放动画，允许传入可选参数delay（毫秒数）以指定在播放前等待的时间。如果播放的是先前被暂停的动画，那么指定goToStart为true会导致重启动画，而不是继续从暂停位置开始播放 |

表8-3：_Animation提供的控制方法（续）

| 方法 | 参数 | 说明 |
| --- | --- | --- |
| status | N/A | 返回动画的状态。可能的状态值包括"paused"、"playing"、"stopped" |
| gotoPercent | /* Decimal */ percent<br>/* Boolean */ andPlay | 终止动画并按百分比前进到0.0～1.0间的某一点。将andPlay设置为true（默认值为false）重启动画 |

警告：*gotoPercent*不是混合大小写的（mixedCase），如*goToPercent*。这个方法是Dojo工具箱中少数几个没有采取混合大小写格式的方法之一，因而也非常容易拼错。

此外，表8-4中列出的方法也都可以作为animateProperty的参数。表中总结了这些方法的作用，而随后的代码则展示了如何调用这些方法。

表8-4：用作animateProperty参数的方法

| 方法 | 参数 | 说明 |
| --- | --- | --- |
| beforeBegin | N/A | 在动画开始前触发，以便在任何操作发生之前提供对_Animation的访问和对相关节点进行修改 |
| onBegin | /* Object */ value | 在动画开始循环时触发，因此这个方法实际上是异步的。参数value是一个包含CSS样式属性当前值的对象 |
| onAnimate | /* Object */ value | 在动画每播放单独一帧时调用。参数value是一个包含CSS样式属性当前值的对象 |
| onEnd | N/A | 当动画结束时自动调用 |
| onPlay | /* Object */ value | 每次调用play时调用（包括第一次调用play时）。参数value是一个包含CSS样式属性当前值的对象 |
| onPause | /* Object */ value | 当动画暂停时调用。参数value是一个包含CSS样式属性当前值的对象 |
| onStop | /* Object */ value | 每次调用stop时调用。参数value是一个包含CSS样式属性当前值的对象 |

下面就是表8-4中所列方法的实际应用：

```
dojo.animateProperty({
    node : "box1",
    duration:10000,
    rate : 1000,
    beforeBegin:function(){ console.log("beforeBegin: ", arguments); },
    onBegin:function(){ console.log("onBegin: ", arguments); },
    onAnimate:function(){ console.log("onAnimate: ", arguments); },
    onEnd:function(){ console.log("onEnd: ", arguments); },
    onPlay:function(){ console.log("onPlay: ", arguments); },
    properties : {height : {start : "200", end : "400"} }
}).play();
```

下面的代码展示了控制 _Animation 的几个方法的基本应用：

```
<!-- snip -->
<script type="text/javascript">
        dojo.addOnLoad(function() {
            var box = dojo.byId("box1");
            var anim;
            dojo.connect(box, "onclick", function(evt) {
                    anim = dojo.animateProperty({
                    node : box,
                    duration : 10000,
                    rate : 1000,
                    easing : function(x) { console.log(x); return x*x; },
                    properties : {
                        height : {start : '200', end : '400'},
                        width : {start : '200', end : '400'}
                     }
                });
                anim.play();
                dojo.connect(dojo.byId("stop"), "onclick", function(evt) {
                    anim.stop(true);
                    console.log("status is ", anim.status());
                });
                dojo.connect(dojo.byId("pause"), "onclick", function(evt) {
                    anim.pause();
                    console.log("status is ", anim.status());
                });
                dojo.connect(dojo.byId("play"), "onclick", function(evt) {
                    anim.play();
                    console.log("status is ", anim.status());
                });
                dojo.connect(dojo.byId("goTo50"), "onclick", function(evt) {
                    anim.gotoPercent(0.5, true);
                    console.log("advanced to 50%");
                });
            });
        });

    </script>
</head>
```

```
    <body>
        <div>
            <button id="stop"  style="margin : 5px">stop</button>
            <button id="pause" style="margin : 5px">pause</button>
            <button id="play"  style="margin : 5px">play</button>
            <button id="goTo50"  style="margin : 5px">50 percent</button>
        </div>
        <div id="box1" class="box blueBox">Click Here</div>
        <div id="box2" class="box redBox"></div>
        <div id="box2" class="box greenBox"></div>
    </body>
</html>
```

## 深入了解 _Animation

如前所述，动画效果一般都可以通过辅助方法（特别是animateProperty）来实现。而像animateProperty这样的方法实际上就像是一个配置_Animation对象的包装函数，但它们仍然返回_Animation对象以便播放、暂停动画时使用。

谈到_Animation，读者也许感觉它有些神秘。实际上，_Animation非常简单。除了node和properties参数之外，_Animation也和animateProperty方法一样接收表8-2中所列的参数（因为animateProperty方法本来就是一个实现动画创建和操作的包装函数）。此外，还接收一个curve（中文即“曲线”。—— 译者注）参数，这个参数定义的是缓动函数的范围。像animateProperty、fadeIn这样内置的动画方法，它们默认的范围是0～1。但是，curve参数能够接收的范围却没有任何限制。（如果读者觉得*curve*有点措词不当，那一点也不意外，因为你并不是第一个有这种想法的人。毕竟，curve只是个一维概念，而curves才是我们常见的二维概念。）

另外，有关的各个参数，像curve、duration和rate等，都有必要在此重新归纳一下它们的作用：

duration

控制动画播放多长时间。

rate

多长时间刷新一次动画；用rate除duration就可以得到动画中全部独立的帧数。rate所代表的帧速率（frame rate）的概念之所以重要，是因为动画播放的过程相当于从当前帧数执行到总帧数。

— 待续 —

curve

传入到onAnimate方法中的可能值的范围。对于传入到onAnimate中的每个值，都会通过将缓动函数的结果投射到该范围进行计算。每个curve值以一个数组表示，该数组中包含着两个数字值，一个表示范围的起点，另一个表示范围的终点。

easing

一个接收0～1中任意数值的函数，它接收的数值对应着动画的进程。缓动函数被投射到“curve”上的结果就是要传入onAnimate中的值。如果缓动函数返回的值大于1.0，那么传入onAnimate中的值将扩展由curve定义的终点，这很正常。

为了将_Animation的各个参数包装起来，我们假设有一个动画，它的持续时间（duration）为10秒，帧速率（rate）为1秒，由curve定义的输入范围被设置为50～100，而缓动函数就是简单的function(x){ return 2*x; }。

根据以上配置，缓动函数将会接收到10个不同的值0.1、0.2、0.3、……、1.0，然后乘上系数2，再返回0.2、0.4、0.6、……、2.0。接着，缓动函数的输出被投射到由curve定义的范围上，并将计算结果传入onAnimate方法中；因此，onAnimate方法依次接收到的值就是50、60、70、80、90、……、150。

以下就是验证上述示例代码：

```
new dojo._Animation({
  duration:10000,
  rate : 1000,
  curve: [50,100],
  easing : function(x) {
     console.log("easing: ", 2*x);
     return 2*x;
  },
  onAnimate:function(x){
    console.log("onAnimate: ", x);
  },
  onEnd:function(){
    console.log('all done.');
  }
}).play();
```

# Core的fx模块

到目前为止，本章的内容完全集中在介绍Base提供的动画方法上。Base中包含的fadeIn、fadeOut和animateProperty方法虽然能够满足多数情况下的需要；然而，

Core的fx模块中也提供了一些补充的动画功能，而要想使用这些功能，需要多写一条dojo.require语句。这些补充的动画方法包括滑动节点、擦入擦出节点，以及连缀、组合及切换动画等。

## 滑动

滑动节点与淡入淡出节点一样简单，只需向dojo.fx.slideTo方法传入一个包含配置参数的对象即可，就和使用animateProperty方法一样。表8-5总结了相关的参数。

表8-5：Core提供的滑动方法的参数

| 参数 | 类型 | 说明 |
| --- | --- | --- |
| node | DOM Node | 产生滑动动画的节点 |
| duration | 整数 | 滑动动画持续的毫秒数。默认值为350 |
| easing | 函数 | 用于调整动画执行过程中加速及减速的函数。默认值为：(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)。注意，对于slideTo方法而言，这个缓动函数定义的范围仅介于0～1 |
| left | 整数 | 滑动动画结束时节点左边的位置 |
| top | 整数 | 滑动动画结束时节点上边的位置 |

例8-3展示了滑动方法的使用，其中唯一有别于前面淡入淡出方法示例的地方，就是加粗的代码。

例8-3：滑动节点的示例

```
<html>
    <head>
       <title>Animation Station</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                margin : 5px;
                background : blue;
                text-align : center;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
```

```
<script type="text/javascript">
    dojo.require("dojo.fx");

    dojo.addOnLoad(function() {
        var box = dojo.byId("box");
        dojo.connect(box, "onclick", function(evt) {
            dojo.fx.slideTo({
                node:box,
                top : "200",
                left : "200"
            }).play();
        });
    });
    </script>
  </head>
  <body>
      <div id="box" class="box">Slide Me</div>
  </body>
</html>
```

## 擦除

滑动和淡入淡出虽然比较有趣，但擦除的使用也不少见，因此fx中同样提供了相应方法。应该说，如何使用擦除方法已经不是问题了，关键在于如何应用那些大致相似的参数。表 8-6 列出了擦除方法的参数。

表 8-6 ：Core 提供的擦除方法的参数

| 参数 | 类型 | 说明 |
|---|---|---|
| node | DOM Node | 产生擦除动画的节点 |
| duration | 整数 | 擦除动画持续的毫秒数。默认值为 350 |
| easing | 函数 | 用于调整动画执行过程中加速及减速的函数。默认值为：(0.5 + ((Math.sin((n + 1.5) * Math.PI))/2)。注意，对于wipeIn和wipeOut方法而言，这个缓动函数定义的范围仅介于0~1 |

**警告：** 在某些布局中，当擦除动画完成后，节点的边框、外边距和内边距值有可能影响原有布局。

下面继续看一个示例，从例 8-4 开始。

例8-4：擦除节点的示例

```
<html>
    <head>
       <title>Animation Station</title>
        <style type="text/css">
            @import "http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css";
            .box {
                width : 200px;
                height : 200px;
                text-align : center;
                float : left;
                position : absolute;
                margin : 5px;
            }
        </style>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.fx");
            dojo.addOnLoad(function() {
                var box = dojo.byId("box");
                dojo.connect(box, "onclick", function(evt) {
                    dojo.fx.wipeOut({
                        node:box
                    }).play();
                });
            });
        </script>
    </head>
    <body>
        <div class="box">Now you don't</div>
        <div id="box" style="background : blue" class="box">Now you see me...</div>
    </body>
</html>
```

此外，在擦除方法中使用自定义的缓动函数也非常值得尝试。在下面的addOnLoad代码块中，我们使用了前面自定义的、非单调的缓动函数，它会导致动画产生弹跳的效果：

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        dojo.fx.wipeOut({
            node:box,
            easing : function(x) { return Math.pow(Math.sin(4*x),2);},
            duration : 5000
        }).play();
    });
});
```

因为缓动函数返回的值先增加、再减少，然后又增加……因此，wipeOut 内部使用的 _Animation 会相应地变化节点的高度。

## 连缀和组合

虽然淡入淡出、滑动擦除动画本身都有可圈可点之处，但我们能够做的还不止如此：使用 fx 模块提供的 dojo.fx.chain 方法还可以将动画连缀起来。说起 dojo.fx.chain 方法，其实很简单，因为它只接收一个包含 _Animation 对象的数组作为参数，并返回另一个 _Animation 对象以便调用 play 方法播放连缀后的动画。下面，我们就通过这个方法让页面中盒子的动画更加活泼一些。表 8-7 列出了用于连缀和组合动画的方法。

**警告：** 到 Dojo1.1 为止，在组合多个动画效果的情况下，使用本节介绍的 chain 和 combine 方法在处理 beforeBegin 和 onEnd 事件时存在一些问题。避免这些问题的解决方案是在需要这些事件支撑应用程序逻辑的时候，使用 dojo.connect 和 dojo.subscribe 来自己定义连缀和组合动画。当然，对于不那么复杂的任务，chain 和 combine 方法仍然很合适。

表 8-7：连缀和组合动画的方法

| 方法 | 说明 |
| --- | --- |
| dojo.fx.chain(/* Array */ animations) | 将作为参数传入的数组中的动画对象连缀在一起，并返回一个统一的、可以像平常一样播放的动画对象。返回的动画对象会连续地播放连缀起来的每个动画 |
| dojo.fx.combine(/* Array */ animations) | 将作为参数传入的数组中的动画对象组合在一起，并返回一个统一的、可以像平常一样播放的动画对象。返回的动画对象会并行地播放组合起来的每个动画 |

例 8-5 展示了一个在屏幕上斗折蛇行的盒子。注意，可以跟往常一样使用自定义的缓动函数和其他参数。

例 8-5：连缀动画的示例

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        var easing = function(x) { return x; };
        var a1 = dojo.fx.slideTo({
            node:box,
            easing : easing,
```

```
        duration : 1000,
        top : "150",
        left : "300"
    });
    var a2 = dojo.fx.slideTo({
        node:box,
        easing : easing,
        duration : 400,
        top : "20",
        left : "350"
    });
    var a3 = dojo.fx.slideTo({
        node:box,
        easing : easing,
        duration : 800,
        top : "350",
        left : "400"
    });
    dojo.fx.chain([a1,a2,a3]).play();
});
});
```

想在上面动画播放的同时再加入淡入淡出和滑动？没问题。只要像使用dojo.fx.chain方法一样使用dojo.fx.combine方法就可以了。任何在数组中传入dojo.fx.combine方法的动画对象都将同时播放。首先，我们只组合滑动和淡出动画。例8-6是修改后的addOnLoad代码块。

例8-6：简单的组合动画示例

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {
        var a1 = dojo.fx.slideTo({
            node:box,
            top : "150",
            left : "300"
        });
        var a2 = dojo.fadeOut({
            node:box
        });
        dojo.fx.combine([a1,a2]).play();
    });
});
```

**警告：** 由于连缀和组合动画很方便，因此我们很容易忘记slideTo是dojo.fx模块的方法，而fadeIn和fadeOut是Base提供的方法。结果，很可能会错误地调用dojo.fx.fadeIn。而且，如果在没有使用dojo.require("dojo.fx")语句的情况下就调用dojo.fx中的方法，同样会导致错误。

既然chain返回一个独立的_Animation对象，那么我们可以试一试更高级的操作（不过，也十分简单），毕竟连缀和组合的原理都是一样的。在例8-7中，我们先连缀了几个滑动动画和淡入淡出动画，然后又将得到的连缀动画组合到了一起。

例8-7：连缀并组合动画的示例

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    dojo.connect(box, "onclick", function(evt) {

        //连缀几个滑动动画
        var a1 = dojo.fx.slideTo({
            node:box,
            top : "150",
            left : "300"
        });
        var a2 = dojo.fx.slideTo({
            node:box,
            top : "20",
            left : "350"
        });
        var a3 = dojo.fx.slideTo({
            node:box,
            top : "350",
            left : "400"
        });
        var slides = dojo.fx.chain([a1,a2,a3]);

        //连缀几个淡入淡出动画
        var a1 = dojo.fadeIn({
            node:box
        });
        var a2 = dojo.fadeOut({
            node:box
        });
        var a3 = dojo.fadeIn({
            node:box
        });
        var fades = dojo.fx.chain([a1,a2, a3]);

        //再将两个连缀的动画组合起来
        dojo.fx.combine([slides, fades]).play();
    });
});
```

## 切换

从本质上讲，dojo.fx.Toggler类封装了对切换（显示和隐藏）一个节点的动画配置。这个类的构造函数接收一个关联数组参数，开发人员要在这个参数中指定用于显示和隐

藏的方法，以及显示和隐藏方法持续的时间。Toggler类的使用也非常简单，只要告诉它使用哪个方法显示，哪个方法隐藏，并指定持续的时间，就可以手工调用show和hide方法执行动画了。另外，show和hide都可以接收一个可选的延迟参数，用于指定延迟多长时间再播放动画。表8-8展示了Toggler类接收的参数。

表8-8：Core的Toggler类接收的参数

| 参数 | 类型 | 说明 |
|---|---|---|
| node | DOM Node | 要切换的节点 |
| showFunc | 函数 | 返回用于显示节点的_Animation对象的函数。默认值为dojo.fadeIn |
| hideFunc | 函数 | 返回用于隐藏节点的_Animation对象的函数。默认值为dojo.fadeout |
| showDuration | 整数 | showFunc运行的毫秒数。默认值为200（毫秒） |
| hideDuration | 整数 | hideFunc运行的毫秒数。默认值为200（毫秒） |

表8-9列出了Toggler类的方法。

表8-9：Toggler类的方法

| 方法 | 说明 |
|---|---|
| show(/*Integer*/delay) | 调用showFunc在showDuration指定的时间内显示节点。可选的delay参数指定在执行动画之前等待多长时间 |
| hide(/*Integer*/delay) | 调用hideFunc在hideDuration指定的时间内隐藏节点。可选的delay参数指定在执行动画之前等待多长时间 |

例8-8中的代码在例8-4的基础上进行了必要的修改，这个addOnLoad代码块展示了如何使用Toggler实现动画切换。

例8-8：切换节点动画的示例

```
dojo.addOnLoad(function() {
    var box = dojo.byId("box");
    var t = new dojo.fx.Toggler({
        node : box,
        showDuration : 1000,
        hideDuration : 1000
    });
    var visible = true;
    dojo.connect(box, "onclick", function(evt) {
        if (visible)
            t.hide();
```

```
        else
            t.show();

        visible = !visible;
    });
});
```

在试验这个示例时，单击“Now you see me...”盒子会导致它淡出屏幕，而单击随之出现的“Now you don't”盒子，则会导致第一个盒子淡入到屏幕中。

## 动画 + 拖放 = 酷

把拖放和动画放到一起应该是一个非常值得期待的组合。首先，请读者先体验下面的代码示例，该示例组合了上一章介绍的拖放和本章前面介绍的动画效果；其中动画的曲线如图 8-6 所示。

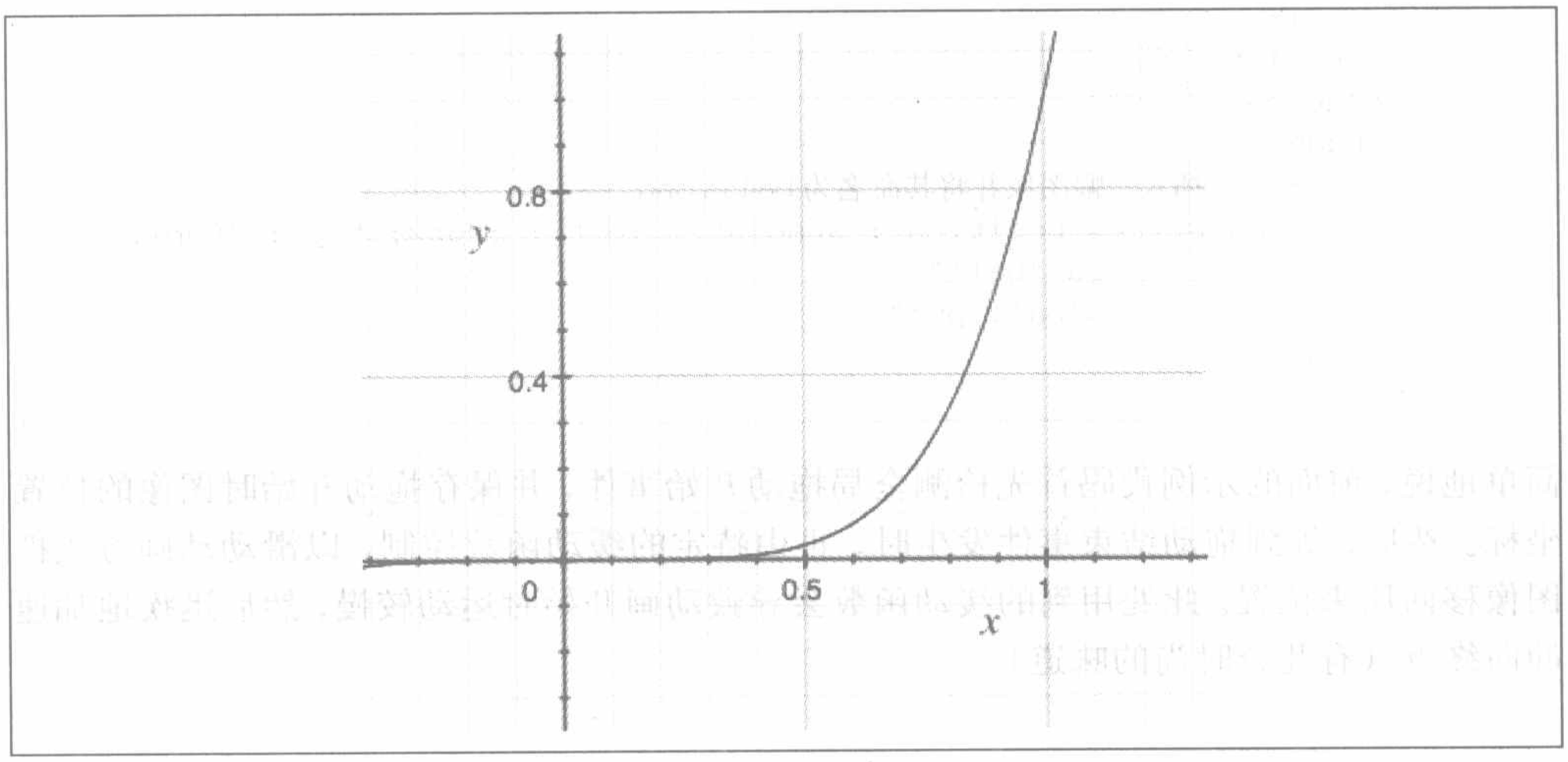



图 8-6：x^5 的缓动函数曲线

```
<html>
    <head>
        <title>Animation + Drag and Drop = Fun!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
    <script type="text/javascript">
        dojo.require("dojo.fx");
```

```
dojo.require("dojo.dnd.move");
dojo.addOnLoad(function(){
  var move = new dojo.dnd.Moveable(dojo.byId("ball"));
  var coords;
  dojo.subscribe("/dnd/move/start",function(e){
    //在拖动开始时，保存当前坐标
    coords = dojo.coords(e.node);
  });

  //以前面保存的坐标为依据，控制图像滑动的终点
  dojo.subscribe("/dnd/move/stop",function(e){
    dojo.fx.slideTo({
      node: e.node,
      top: coords.t,
      left: coords.l,
      duration:1200,
      easing : function(x) { return Math.pow(x,5);}
    }).play();
  });
});
</script>
</head>
<body>
    <!-- 插入一幅图像并将其命名为ball.png -->
    <img style="position : absolute; left : 300px; top : 300px;"
         id="ball"
       src="ball.png"/>
</body>
</html>
```

简单地说，前面的示例代码首先检测全局拖动开始事件，并保存拖动开始时图像的位置坐标。然后，等到拖动结束事件发生时，再由特定的缓动函数控制，以滑动动画方式把图像移回原来位置。此处用到的缓动函数会导致动画开始时运动较慢，然后迅疾地加速冲向终点（有几分时尚的味道）。

# 颜色

网页中的动画和特效经常依赖于对特定颜色值的计算。Base为此提供了一个基本的Color类，封装了大量颜色计算逻辑，以方便各种颜色表示法之间的相互转换。此外，Color类也提供了操作颜色的常用辅助方法。

## 创建和混合颜色

Color类拥有一个适应能力很强的构造函数，可以接收颜色的命名字符串值、十六进制

字符串值或者RGB（注2）值数组。例8-9展示了创建两个Color对象和使用Base提供的blendColors方法混合颜色的过程。

例8-9：混合Color对象

```
<html>
    <head>
        <title></title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.addOnLoad(function() {
                var blue = new dojo.Color("#0000ff"); //也可以使用"blue"
                var red = new dojo.Color([255, 0, 0]);
                var purple = dojo.blendColors(blue, red, 0.5);
                dojo.style("foo", "background", purple.toCss());
            });
        </script>
    </head>
    <body>
        <div id="foo" style="width:200px; height:200px; padding:5px;"></div>
    </body>
</html>
```

其中的blendColors方法接收了red和blue两个Color对象，并将它们按照各取50%的原则混合产生了RGB值（128,0,128），即紫色的中性灰。另外一种混合颜色的做法就是直接输入数字——虽然不是什么高科技，但也没有想像得那么好玩！

表8-10总结了Base提供的Color类的功能。

表8-10：Base提供的Color类的方法

| 方法 | 说明 |
| --- | --- |
| Color(/* Array \| String */color) | 构造函数，接收一个RGB或RGBA值的数组，或者一个字符串值（如“blue”）或一个十六进制字符串（如“#000ff”）。如果未传递参数，那么以RGBA元组（255, 255, 255, 1）创建Color对象 |

注2：RGB是“red green blue”的简写，是CSS中表示颜色的标准方法。RGBA则是“red green blue alpha”的简写，其中的alpha表示颜色的不透明度。

表8-10：Base提供的Color类的方法（续）

| 方法 | 说明 |
| --- | --- |
| setColor(/* Array \| String \| Object */ color) | 操作已有的Color对象，采用与其构造函数类似的方式配置颜色值；是重用已有 Color 对象的首选方式 |
| toRgb() | 返回一个字符串值，表示RGB颜色值，例如，(128, 0, 128) |
| toRgba() | 返回一个字符串值，表示RGBA颜色值，例如（128, 0, 128, 0.5） |
| toHex() | 返回一个字符串值，表示十六进制颜色值，例如，“#80080” |
| toCss(/* Boolean */ includeAlpha) | 返回一个符合CSS标准的字符串值，表示RGB颜色值，如（128, 0, 128）。在默认情况下，includeAlpha的值为false。此方法是将Color对象转换为CSS样式中所用颜色值的首选方法 |
| toString | 返回一个表示RGBA颜色值的标准字符串 |

**警告：** 当前，多数浏览器的实现都与CSS2规范存在偏差，它们不支持RGBA元组表示的颜色值，因此Color对象的toCss()方法（不传入参数）恐怕是我们产生dojo.style之类的方法所能接受的颜色值的重要手段。如果读者需要设置节点的透明度，可以使用opacity样式。

## CSS3中对RGBA的支持

CSS3规范中包含的用于表示颜色值的RGBA语法简洁明了。下面的代码片段定义了两个相互重叠的蓝色和红色方块，图8-7展示了在这两个方块相交的区域，由于重叠混合显示出的紫色方块：

```
<div style="width:200px; height:200px; padding:5px; background:rgba(0,0,255,
0.5)"></div>

<div style="position:absolute; left:100px; top:100px; width:200px; height:
200px; padding:5px; background:rgba(255,0,0,0.5)"></div>
```

Firefox 3本地支持RGBA样式值。要了解Firefox 3支持的更多CSS3特性，请参见*http://developer.mozilla.org/en/docs/CSS_improvements_in_Firefox_3*。

**注意：** 透明度（transparency）和不透明度（opacity）是一对相反的概念。如果不透明度值为1.0，即表示完全不透明，那么其透明度就是0.0。如果不透明度是0.1，也就是说90%透明，那么人眼会很难看到相应的节点。

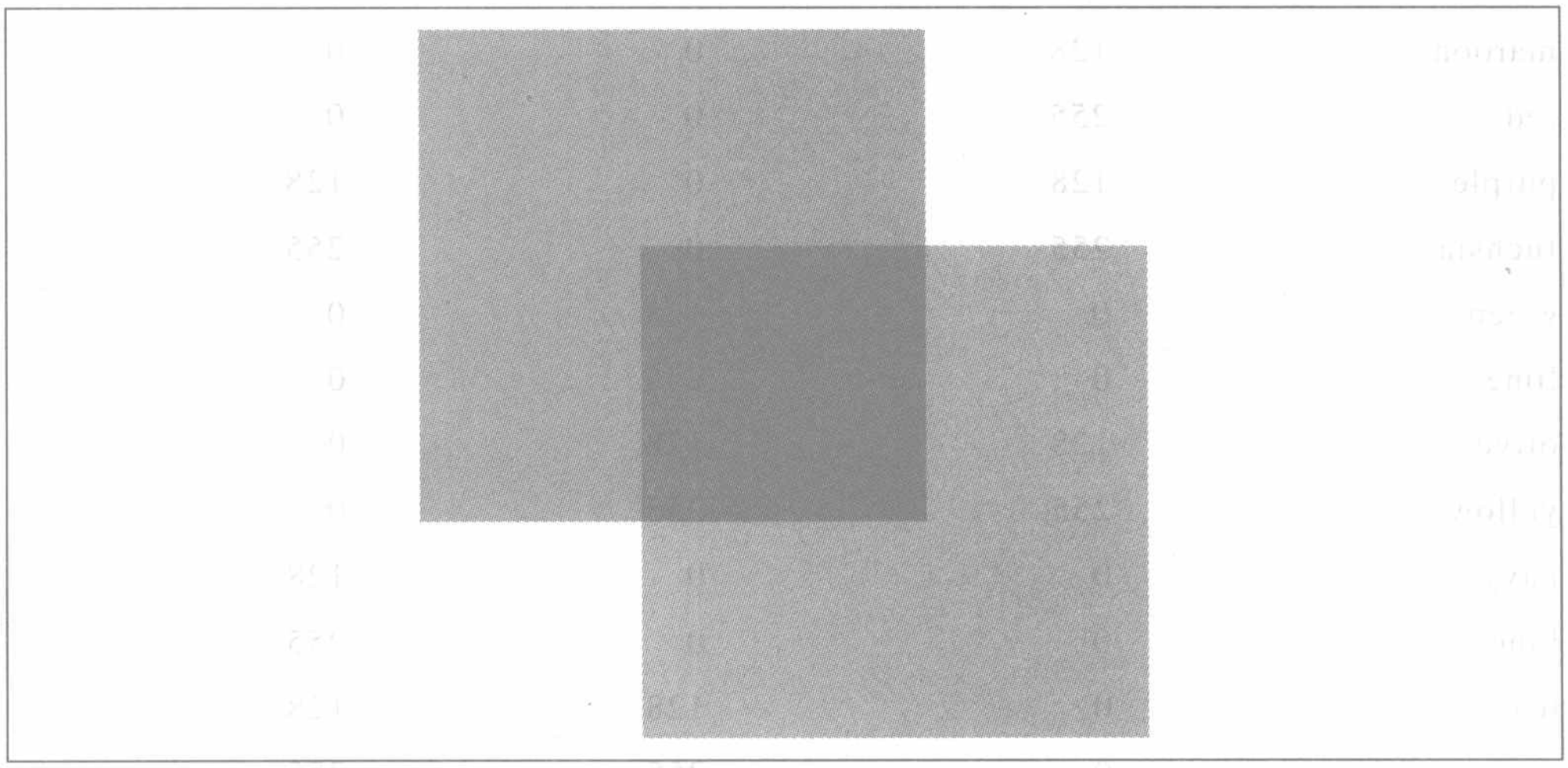


图8-7：RGBA语法定义的透明度在Firefox 3中的示例

## Base中可用的命名颜色值

Base还提供了dojo.Color.named对象，该对象是颜色名与对应RGB值的映射。例如，要使用暗红色（maroon）的RGB值，可以引用dojo.Color.named.maroon来取得RGB数组[128,0,0]。表8-11总结了Base中内置的颜色名。虽然直接使用和操作Color对象很方便，但dojo.Color.named在开发中同样也很有用。

**警告：** 不能通过Color对象来访问dojo.Color.named。实际上，dojo.Color.named是一个颜色值的静态集合，使用它不必创建Color对象。如果要访问Color对象的.named属性，会导致一个错误。

表8-11：Base中可用的命名颜色值

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| black | 0 | 0 | 0 |
| silver | 192 | 192 | 192 |

表8-11：Base中可用的命名颜色值（续）

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| gray | 128 | 128 | 128 |
| white | 255 | 255 | 255 |
| maroon | 128 | 0 | 0 |
| red | 255 | 0 | 0 |
| purple | 128 | 0 | 128 |
| fuchsia | 255 | 0 | 255 |
| green | 0 | 128 | 0 |
| lime | 0 | 255 | 0 |
| olive | 128 | 128 | 0 |
| yellow | 255 | 255 | 0 |
| navy | 0 | 0 | 128 |
| blue | 0 | 0 | 255 |
| teal | 0 | 128 | 128 |
| aqua | 0 | 255 | 255 |

## Core中可用的更多颜色值

虽然没有包含在Base中，但执行`dojo.require("dojo.colors")`语句即可扩展`dojo.Color.named`，让它再多包含100多个颜色值。这些颜色值涵盖了CSS3中全部的命名颜色和SVG1.0中颜色名的变体拼写（参见表8-12）。别忘了，前面我们介绍的`animateProperty`方法也可以生成颜色动画。例如，可以在`start`和`end`值中指定“black”和“white”、“white”和“#43fab4”等等。

---

**注意：** 除了扩展`dojo.Color.named`之外，`dojo.colors`还增强了`Color`构造函数，使它能够接收HSL和HSLA颜色模型的格式。HSL颜色空间旨在通过色相（Hue）、饱和度（Saturation）和亮度（Lightness）来更精确地表现人眼能够感知的色彩。要了解CSS颜色模型的相关信息，读者可以参见*http://www.w3.org/TR/css3-iccprof*。

---

表 8-12：Core 中可用的更多颜色值

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| aliceblue | 240 | 248 | 255 |
| antiquewhite | 250 | 235 | 215 |
| aquamarine | 127 | 255 | 212 |
| azure | 240 | 255 | 255 |
| beige | 245 | 245 | 220 |
| bisque | 255 | 228 | 196 |
| blanchedalmond | 255 | 235 | 205 |
| blueviolet | 138 | 43 | 226 |
| brown | 165 | 42 | 42 |
| burlywood | 222 | 184 | 135 |
| cadetblue | 95 | 158 | 160 |
| chartreuse | 127 | 255 | 0 |
| chocolate | 210 | 105 | 30 |
| coral | 255 | 127 | 80 |
| cornflowerblue | 100 | 149 | 237 |
| cornsilk | 255 | 248 | 220 |
| crimson | 220 | 20 | 60 |
| cyan | 0 | 255 | 255 |
| darkblue | 0 | 0 | 139 |
| darkcyan | 0 | 139 | 139 |
| darkgoldenrod | 184 | 134 | 11 |
| darkgray | 169 | 169 | 169 |
| darkgreen | 0 | 100 | 0 |
| darkgrey | 169 | 169 | 169 |
| darkkhaki | 189 | 183 | 107 |
| darkmagenta | 139 | 0 | 139 |
| darkolivegreen | 85 | 107 | 47 |
| darkorange | 255 | 140 | 0 |
| darkorchid | 153 | 50 | 204 |
| darkred | 139 | 0 | 0 |
| darksalmon | 233 | 150 | 122 |

表8-12：Core中可用的更多颜色值（续）

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| darkseagreen | 143 | 188 | 143 |
| darkslateblue | 72 | 61 | 139 |
| darkslategray | 47 | 79 | 79 |
| darkslategrey | 47 | 79 | 79 |
| darkturquoise | 0 | 206 | 209 |
| darkviolet | 148 | 0 | 211 |
| deeppink | 255 | 20 | 147 |
| deepskyblue | 0 | 191 | 255 |
| dimgray | 105 | 105 | 105 |
| dimgrey | 105 | 105 | 105 |
| dodgerblue | 30 | 144 | 255 |
| firebrick | 178 | 34 | 34 |
| floralwhite | 255 | 250 | 240 |
| forestgreen | 34 | 139 | 34 |
| gainsboro | 220 | 220 | 220 |
| ghostwhite | 248 | 248 | 255 |
| gold | 255 | 215 | 0 |
| goldenrod | 218 | 165 | 32 |
| greenyellow | 173 | 255 | 47 |
| grey | 128 | 128 | 128 |
| honeydew | 240 | 255 | 240 |
| hotpink | 255 | 105 | 180 |
| indianred | 205 | 92 | 92 |
| indigo | 75 | 0 | 130 |
| ivory | 255 | 255 | 240 |
| khaki | 240 | 230 | 140 |
| lavender | 230 | 230 | 250 |
| lavenderblush | 255 | 240 | 245 |
| lawngreen | 124 | 252 | 0 |
| lemonchiffon | 255 | 250 | 205 |
| lightblue | 173 | 216 | 230 |

表 8-12：Core 中可用的更多颜色值（续）

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| lightcoral | 240 | 128 | 128 |
| lightcyan | 224 | 255 | 255 |
| lightgoldenrodyellow | 250 | 250 | 210 |
| lightgray | 211 | 211 | 211 |
| lightgreen | 144 | 238 | 144 |
| lightgrey | 211 | 211 | 211 |
| lightpink | 255 | 182 | 193 |
| lightsalmon | 255 | 160 | 122 |
| lightseagreen | 32 | 178 | 170 |
| lightskyblue | 135 | 206 | 250 |
| lightslategray | 119 | 136 | 153 |
| lightslategrey | 119 | 136 | 153 |
| lightsteelblue | 176 | 196 | 222 |
| lightyellow | 255 | 255 | 224 |
| limegreen | 50 | 205 | 50 |
| linen | 250 | 240 | 230 |
| magenta | 255 | 0 | 255 |
| mediumaquamarine | 102 | 205 | 170 |
| mediumblue | 0 | 0 | 205 |
| mediumorchid | 186 | 85 | 211 |
| mediumpurple | 147 | 112 | 219 |
| mediumseagreen | 60 | 179 | 113 |
| mediumslateblue | 123 | 104 | 238 |
| mediumspringgreen | 0 | 250 | 154 |
| mediumturquoise | 72 | 209 | 204 |
| mediumvioletred | 199 | 21 | 133 |
| midnightblue | 25 | 25 | 112 |
| mintcream | 245 | 255 | 250 |
| mistyrose | 255 | 228 | 225 |
| moccasin | 255 | 228 | 181 |
| navajowhite | 255 | 222 | 173 |

表8-12：Core中可用的更多颜色值（续）

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| oldlace | 253 | 245 | 230 |
| olivedrab | 107 | 142 | 35 |
| orange | 255 | 165 | 0 |
| orangered | 255 | 69 | 0 |
| orchid | 218 | 112 | 214 |
| palegoldenrod | 238 | 232 | 170 |
| palegreen | 152 | 251 | 152 |
| paleturquoise | 175 | 238 | 238 |
| palevioletred | 219 | 112 | 147 |
| papayawhip | 255 | 239 | 213 |
| peachpuff | 255 | 218 | 185 |
| peru | 205 | 133 | 63 |
| pink | 255 | 192 | 203 |
| plum | 221 | 160 | 221 |
| powderblue | 176 | 224 | 230 |
| rosybrown | 188 | 143 | 143 |
| royalblue | 65 | 105 | 225 |
| saddlebrown | 139 | 69 | 19 |
| salmon | 250 | 128 | 114 |
| sandybrown | 244 | 164 | 96 |
| seagreen | 46 | 139 | 87 |
| seashell | 255 | 245 | 238 |
| sienna | 160 | 82 | 45 |
| skyblue | 135 | 206 | 235 |
| slateblue | 106 | 90 | 205 |
| slategray | 112 | 128 | 144 |
| slategrey | 112 | 128 | 144 |
| snow | 255 | 250 | 250 |
| springgreen | 0 | 255 | 127 |
| steelblue | 70 | 130 | 180 |
| tan | 210 | 180 | 140 |

表 8-12：Core 中可用的更多颜色值（续）

| 颜色名 | 红 | 绿 | 蓝 |
|---|---|---|---|
| thistle | 216 | 191 | 216 |
| tomato | 255 | 99 | 71 |
| transparent | 0 | 0 | 0 |
| turquoise | 64 | 224 | 208 |
| violet | 238 | 130 | 238 |
| wheat | 245 | 222 | 179 |
| whitesmoke | 245 | 245 | 245 |
| yellowgreen | 154 | 205 | 50 |

# 小结

本章系统地展示了 Base 和 Core 中包含的动画方法。网页中的动画，在使用适当的情况下，能够为应用程序增加几分魅力，并使应用程序显得与众不同。在学习完本章之后，读者应该：

- 能够使用 Base 的动画方法淡入淡出节点。
- 能够使用 Base 的 `animateProperty` 方法生成任意 CSS 属性的动画。
- 理解 `_Animation` 对象的缓动函数、持续时间和帧速率的作用。
- 知道 Core 对 Base 中动画方法提供补充支持。
- 能够使用 Core 中提供的动画方法实现滑动和擦除动画。
- 能够使用`dojo.fx.chain`方法连缀顺序执行的动画，也能够使用dojo.fx.combine方法组合并同时播放多种动画效果。
- 能够使用`dojo.fx.Toggler`提供的简单的、统一的接口实现对节点的显示和隐藏。
- 理解如何组合动画和拖放，以创建具有高度交互性的页面元素。
- 能够创建并有效地使用 Color 对象以消除在代码中手工计算颜色值的麻烦。

---

**注意：** 在`dojo.gfx`模块中，还提供了基于SVG、VML和Silverlight的令人称奇的图形和动画工具。

---

下一章，我们将介绍数据抽象。

# 第9章

# 数据抽象

Web开发中的一个主要障碍，就是编写代码解析那些从服务器返回的数据，以便将这些数据转换为核心逻辑易于操作的格式。虽然对于像CSV（Comma-Separated Values，逗号分隔的值）及JSON等常见的响应类型，已经有了许多不错的解析程序，但从整体上来说，无论是向服务器传送更新后的数据，还是在保持本地数据与服务器同步方面，仍然会涉及许多繁琐的工作量。本章介绍Dojo工具箱中的数据API，这些API为处理不同的数据源提供了统一的接口，从而使开发人员不必再因为数据保存在不同位置、访问不同传输层次中的数据，以及区分不同的数据格式而烦恼。

## 重建数据访问模式

Dojo工具箱中用于操作数据源的机制也不是什么航天科技，不过确实要求开发人员从以往数据访问的思维定式中摆脱出来，因为这种机制要求我们将数据看成是能够通过统一的API访问的本地资源。传统的数据访问模式通常是把数据看成远程资源，因而需要通过既定的方式取得这些资源，然后再把更新后的数据写回到服务器，并且还要维护本地数据与服务器数据间的同步，最后还必须处理好各种各样的数据格式。回顾历史可以发现，存在于数据访问方面的一个核心问题就是不计其数的人都在搞重复建设，几乎每个应用程序都在使用自有的脆弱手段来应对数据处理的重任。

如图9-1所示，Dojo通过`dojo.data`模块提供了一组API，这组API为与任意数据源进行交互提供了一种标准手段。有了这些API，相关的程序开发人员就能从获取、解析和管理数据的纷繁任务中解脱出来。无论本地数据还是远程数据，都能通过`dojo.data`模块提供的API以标准方式与之交互——这在应用程序规模不断增大，需要处理的数据量也越来越惊人的背景下尤为难能可贵。最值得一提的是，当开发人员为某种数据格式

实现了一个接口后，该接口就可以作为一种通用资源加以重用或者随意分发。一般而言，开发人员利用这种通用资源不仅能够更有效地处理数据，关键还能够将更多的精力转移到最重要的任务上。

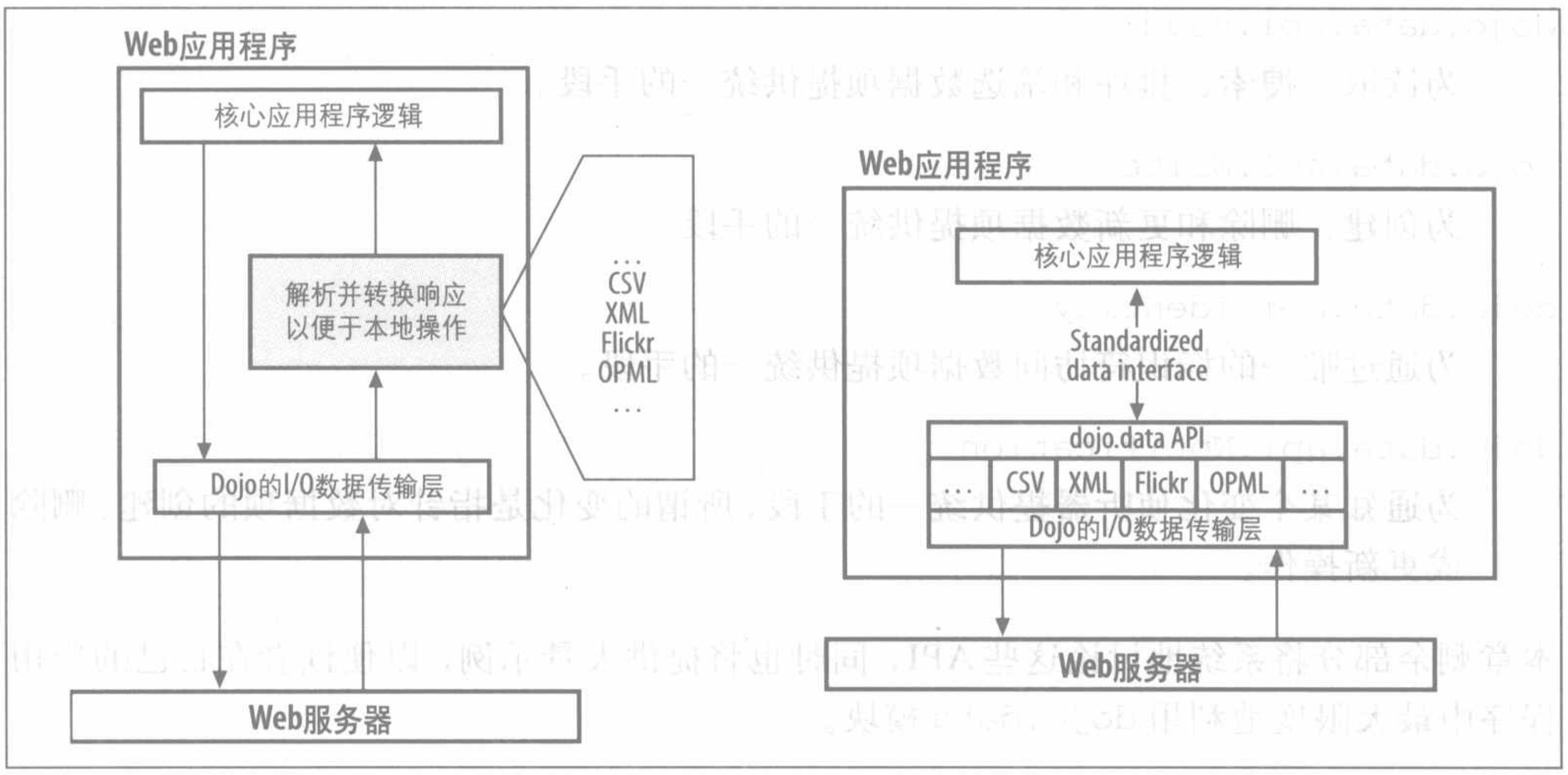



图 9-1：左图为应用程序中访问任意数据源的传统模式；右图为 Dojo 工具箱中的 dojo.data 模块提供的对任意数据源的访问模式

# 数据 API 概览

在 dojo.data 提供的 API 中，最基本的数据单位叫做数据项（item），数据项由键 / 值对组成，而键和值在 dojo.data 的语境下分别叫做属性和属性值。从概念上来说，可以把一个数据项想像成为 JavaScript 中的一个对象。但是，即使数据项的底层实现就是一个 JavaScript 对象，也必须使用模块提供的 API 来访问它，毕竟数据项的内部表示方法与 JavaScript 对象有可能完全不同。例如，在某些数据抽象的条件下，可能会出于效率原因使用 DOM 模型来存储某种类型的数据，也可能会动态地延迟加载数据，尽管看起来数据似乎就在本地。如果在这些情况下，像访问 JavaScript 对象一样访问数据项，就会导致意想不到的错误。下一节将介绍访问数据项的具体 API。

---

**注意：** 说某个数据项有一个属性，但该属性没有值，就相当于说这个数据项没有该属性。换句话说，有属性却没有值是毫无意义的，毕竟每个属性都应该具有特定的状态。

---

在具体介绍API之前，有必要先了解一些Dojo提供的数据API的构成。以下列出了dojo.data中包含的API及简介。注意，这些API都是接口，而非实现；任何具体的dojo.data数据存储模式（data store）都必须实现下列一个或多个API。

dojo.data.api.Read

为读取、搜索、排序和筛选数据项提供统一的手段。

dojo.data.api.Write

为创建、删除和更新数据项提供统一的手段。

dojo.data.api.Identity

为通过唯一的标识符访问数据项提供统一的手段。

dojo.data.api.Notification

为通知某个变化侦听器提供统一的手段，所谓的变化是指针对数据项的创建、删除或更新操作。

本章剩余部分将系统地讨论这些API，同时也将提供大量示例，以便读者在自己的应用程序中最大限度地利用dojo.data模块。

# 深入理解API

本节将全面介绍数据API。如果读者以前从未接触到这些API的应用，可以先跳过本节。下一节“Core对数据API的实现”中包含大量具体的示例，读者在通过示例体验到这些API的应用之后，可以再回到这里来深入研究这些API。

## Read API

所有数据存储模式都必须实现dojo.data.api.Read API，因为该API定义了获取、处理和访问数据的方法，而这些方法无疑是执行其他操作的先决条件。完整的API规范如表9-1所示，下一节将讨论Dojo工具箱对这个API的实现——ItemFileReadStore。

---

**注意：** 即使数据项的概念有些抽象，但本章仍会使用dojo.data.api.Item描述符（descriptor）来传达dojo.data数据项的概念。

---

表 9-1：dojo.data.api.Read API

| 名称 | 说明 |
| --- | --- |
| getValue(/*dojo.data.api.Item*/item, /*String*/attribute, /*Any?*/default) | 根据给出的数据项和属性名返回属性值。如果给出的属性不存在，那么返回 undefined（只有在将 null 明确设置为属性值的情况下，才会返回 null）。可选的参数 default 用于在给出的属性不存在时作为默认值返回 |
| getValues(/*dojo.data.api.Item*/item, /*String*/attribute) | 与 getValue 的工作原理相似，但它允许取得多值属性的值。无论属性有几个值，都会返回一个数组。在读取多值属性时，必须使用 getValues |
| getAttributes(/*dojo.data.api.Item*/item) | 迭代数据项并返回该数据项包含的所有属性的字符串值的数组。如果数据项中没有属性，那么返回空数组 |
| hasAttribute(/*dojo.data.api.Item*/item, /*String*/attribute) | 如果数据项中包含指定的属性则返回 true |
| containsValue(/*dojo.data.api.Item*/item, /*String*/attribute, /*Any*/value) | 如果数据项中包含指定的属性，且该属性的值与指定值相符，那么返回 true |
| isItem(/*Any*/item) | 如果参数是一个数据项且来自特定的数据存储模式，那么返回 true。如果数据项是一个字面值或者该数据项来自其他数据存储模式，那么返回 false（对于本地变量引用不变数据项的情况特别有用，而这在实现 Write API 的数据存储模式中很常见。） |
| isItemLoaded(/*Any*/item) | 如果指定的数据项已经加载完毕，可以在本地访问了，那么返回 true |
| loadItem(/*Object*/args) | 加载指定的数据项以便后面对 isItemLoaded 的调用会返回 true。其中，args 对象包含下列键：<br>item<br>在加载数据项时作为匹配条件的对象（可能是一个标识符或识别数据的子集） |

表9-1：dojo.data.api.Read API（续）

| 名称 | 说明 |
| --- | --- |
| loadItem(/*Object*/args)(续) | onItem(/*dojo.data.api.Item*/item)<br>在加载数据项完成后调用的回调函数；被加载的数据项作为该函数的参数<br>onError(/*Object*/error)<br>在加载数据项发生错误后调用的回调函数；错误对象作为该函数的参数<br>scope<br>为回调函数提供执行环境的对象 |
| fetch(/*Object*/args) | 执行指定的查询，并针对与查询执行有关的事件提供各种异步回调函数。返回一个dojo.data.api.Request对象，主要目的是提供一个abort()方法，以便中断查询过程。其中，args对象包含下列键。<br>query<br>在查询时作为匹配条件的字符串或对象（类似SQL中的SELECT语句）。不过，具体的查询语句可能因实现而异<br>queryOptions<br>为查询提供额外选项的对象。所有数据存储模式都应该尽力实现ignoreCase参数（布尔值，默认为false）——用于执行不区分大小写的搜索，和deep参数（布尔值，默认为false）——用于触发对所有项及其子项的查询，而不仅仅查询数据存储模式中的顶级项 |

表9-1：dojo.data.api.Read API（续）

| 名称 | 说明 |
| --- | --- |
| fetch(/*Object*/args)(续) | onBegin (/*Integer*/size, /*dojo.data.api.Request*/request)<br>在第一个onItem回调函数执行前被调用。其中，参数size表示查询到的结果总数，而参数request表示查询的原始请求对象。如果参数size未知，那么视为-1。size也可能不等于返回的结果总数，因为返回的结果总数可能会通过start和count被减少 |
| | onComplete (/*Array*/items, /*dojo.data.api.Request*/request)<br>在最后一次onItem回调函数执行后调用。如果没有onItem回调函数，那么参数items就是一个查询中匹配的所有结果的数组；否则，就是null。参数request是原始的请求对象 |
| | onError(/*Object*/error, /*dojo.data.api.Request*/request)<br>当执行查询过程中发生错误时调用。参数error中包含错误信息，参数request是原始的请求对象 |
| | onItem(/*dojo.data.api.Item*/item, /*dojo.data.api.Request*/request)<br>基于每个返回的查询结果调用，并将每个结果作为item参数；参数request是原始的请求对象 |
| | scope (Object)<br>为每个回调函数提供执行环境；如果未提供这个对象，那么在全局环境下执行回调函数 |
| | start (Integer)<br>指定对返回结果的起始偏移量（类似SQL查询中的OFFSET） |

表9-1：dojo.data.api.Read API（续）

| 名称 | 说明 |
| --- | --- |
| fetch(/*Object*/args)(续) | count (Integer)<br>指定对返回结果数量的限制（类似SQL查询中的LIMIT） |
| | sort(Array)<br>为每个属性提供排序标准的JavaScript对象的数组。每个对象将被按顺序应用，而且必须有一个attribute键用于标识属性名和一个descending键用于标识排序方式 |
| getFeatures() | 返回一个包含键/值对的对象，说明实现的是哪个dojo.data API。例如，实现dojo.data.api.Read的数据存储模式都必须返回只读的{'dojo.data.api.Read' : true} |
| close(/*dojo.data.api.Request*/request) | 用于停止与特定请求关联的所有操作，涉及清理缓存、关闭连接等等。参数request是由fetch返回的请求对象。对于某些数据存储模式而言，只执行空操作 |
| getLabel(/*dojo.data.api.Item*/item) | 用于返回人类容易看懂的标签，通常是数据项的某方面描述信息。这里的标签可以是几个属性的集合 |
| getLabelAttributes (/*dojo.data.api.Item*/item) | 用于提供生成数据项的标签的属性数组。对于UI开发人员有用，可以通过它获悉哪个属性适合显示，因而可以在显示数据项的标签时隐藏冗余信息 |

## Identity API

表9-2所示的Identity API是基于Read API构建，提供了另外一些基于唯一标识符（Identity）取得数据项的方法。我们注意到，Read API中并没有限制数据项必须唯一，而且在某些情况下唯一标识符也是无关紧要的，因此有了专门的Identity API。例如，可以把数据库大致想像成Identity API的实现，因为它为数据表中的每条记录都提供了用于标识该记录的主键。此外，在基于数据的Dijit部件中也要用到Identity API，特别是在提供写入（Write）功能时。（Write API稍后介绍。）

表 9-2：dojo.data.api.Identity API

| 名称 | 说明 |
| --- | --- |
| getFeatures() | 参见 dojo.data.api.Read。返回：<br>{<br>'dojo.data.api.Read' : true,<br>'dojo.data.api.Identity : true<br>} |
| getIdentity(/*dojo.data.api.Item*/item) | 返回数据项的唯一标识符，可能是一个字符串或者一个包含toString方法的对象 |
| getIdentityAttributes (/*dojo.data.api.Item*/item) | 返回用于生成唯一标识符的属性名的数组。在多数情况下，只有一个属性用来提供唯一的标识符，但根据数据源的情况不同，也可能有多个属性。经常用于在显示时可选地隐藏构成唯一标识符的属性 |
| fetchItemByIdentity(/*Object*/args) | 通过唯一标识符取得数据项；如果没有找到与标识符对应的数据项，返回null。其中，args对象包含下列键。<br>identity<br>一个字符串或包含toString方法的对象，其中toString方法用于说明相应的数据项<br>onError(/*Object*/error)<br>在执行查询过程中发生错误时调用。参数 error 中包含错误信息<br>onItem(/*dojo.data.api.Item*/item)<br>基于每个返回的查询结果调用，并将每个结果作为 item 参数<br>scope (Object)<br>如果提供，那么在这个对象的环境下执行所有回调函数；否则，在全局环境下执行回调函数 |

## Write API

Write API（如表 9-3 所示）扩展了 Read API，并提供了用于创建、更新和删除数据

项的方法，这些方法可以解决写入数据的相关问题（例如，数据项中是否保存脏数据（dirty）——即内存中的数据与服务器上的数据不一致）以及保存等I/O操作。

表9-3：dojo.data.api.Write API

| 名称 | 说明 |
| --- | --- |
| getFeatures | 参见dojo.data.api.Read。返回：<br>{<br>'dojo.data.api.Read' : true,<br>'dojo.data.api.Write : true<br>} |
| newItem(/*Object?*/args,<br>/*Object?*/parentItem) | 返回新创建的数据项，基于指定的args对象设置属性，通常是将args中的键/值对映射为属性和属性值。对于支持创建分层数据项的数据存储模式而言，参数parentItem用于指定新数据项的父项唯一标识符以及新数据项必须指定的父项属性（此时，通常意味着属性是多值的，而新数据项会被追加到该属性值中） |
| deleteItem(/*dojo.data.api.Item*/item) | 从存储模式中删除数据项。返回一个表示删除操作成功与否的布尔值 |
| setValue(/*dojo.data.api.Item*/item,<br>/*String*/attribute, /*Any*/value) | 设置数据项中的一个属性，替换原有的属性值。返回一个表示设置操作成功与否的布尔值 |
| setValues(/*dojo.data.api.Item*/item,<br>/*String*/attribute, /*Array*/values) | 设置属性的值，替换原有的值。返回一个表示设置操作成功与否的布尔值 |
| unsetAttribute(/*dojo.data.api.Item*/item,<br>/*String*/attribute) | 通过删除属性值来删除属性。返回一个表示删除操作成功与否的布尔值 |
| save(/*Object*/args) | 在内存中保存所有本地修改，并将输出传入args参数包含的一个回调函数中，args对象包含下列键。<br>onError(/*Object*/error)<br>在保存出错时调用；参数error中包含错误信息<br>onComplete()<br>在保存成功时调用，通常没有参数 |

表 9-3：dojo.data.api.Write API（续）

| 名称 | 说明 |
| --- | --- |
| save(/*Object*/args)(续) | scope (Object)<br>如果指定，那么所有回调函数都在这个对象的环境下执行；否则，在全局环境下执行回调函数。<br>还提供了一个 _saveCustom扩展点，通过重写该扩展点可以将数据回传到服务器 |
| revert() | 放弃任何本地修改。返回一个表示恢复操作成功与否的布尔值 |
| isDirty(/*dojo.data.api.Item?*/) | 返回一个布尔值，表示指定的数据项在最后一次 save（保存）操作后是否被修改过。如果不提供参数，那么返回的布尔值表示整个数据存储模式中是否存在被修改的项 |

## Notification API

Notification API（如表 9-4 所示）基于 Read API 构建并对 Write API 给予了补充，它为响应典型的create、update和delete事件提供了统一的接口。利用Notification API可以从视觉上反映存储状态的变化，存储状态一般在读取和写入操作发生时会发生变化或更新。(dijit.Tree和dojox.grid.Grid部件是利用Notification API的典型实例。)

表 9-4：dojo.data.api.Notification API

| 名称 | 说明 |
| --- | --- |
| getFeatures | 参见 dojo.data.api.Read。返回<br>{<br>'dojo.data.api.Read' : true,<br>'dojo.data.api.Notification: true<br>} |

表9-4：dojo.data.api.Notification API（续）

| 名称 | 说明 |
| --- | --- |
| onSet(/*dojo.data.api.Item*/item, /*String*/attribute, /*Object\|Array*/old, /*Object\|Array*/new) | 每当setValue、setValues或unsetAttribute修改一个数据项时调用，提供了监控数据存储模式中数据项是否被修改的功能。它的参数基本上都可以顾名思义，即分别表示被修改的数据项、被修改的属性、原来的属性值以及新属性值 |
| onNew(/*dojo.data.api.Item*/item, /*Object?*/parentItem) | 当创建一个新数据项时调用，其中参数item是新创建的数据项；如果新创建的数据项是顶级项，那么不会传入parentItem参数，否则会传入parentItem参数（注意，当parentItem的属性被修改时不会生成onSet通知，因为parentItem是隐含的，而且通过parentItem参数可以访问父项的信息） |
| onDelete(/*dojo.data.api.Item*/item) | 当删除一个数据项时调用，并将被删除的数据项作为item参数 |

# Core对数据API的实现

上一节全面介绍了4个当前的主要数据API。本节讨论Core提供的两个实现：ItemFileReadStore和ItemFileWriteStore。稍后我们将介绍到，ItemFileReadStore实现了Write API和Identity API，而ItemFileWriteStore则实现所有4个API。理解了这两个数据存储模式，有助于对它们进行扩展以适应项目开发的需要，甚至实现自己的数据存储模式。

**注意：** 虽然本书没有明确讨论到，但dojox.data子项目中包含着对dojo.data中常见任务的强大补充，例如，与CSV、Flickr、XML、OPML、Picasa及其他存储模式进行交互的能力。由于这些补充的实现同样遵循了上一节介绍的API，因此熟悉它们也并非难事。

## ItemFileReadStore

虽然在特定的情形下，只有使用自定义的dojo.data.api.Read实现才会最有效率，也

最方便，但Dojo工具箱也提供了实现Read和Identity接口的ItemFileReadStore，它能够处理灵活的JSON数据。对于一些来不及开发自定义实现的场合，可以直接使用ItemFileReadStore，而你所要做的就是让应用程序输出它能够处理的数据格式，然后，就可以使用这个数据存储模式了。

---

**注意：** 有关ItemFileReadStore需要提前知道的一点是，由于它要处理的全部数据集都会在第一次请求时加载到内存中，因此，isItemLoaded和loadItem之类的操作基本上没有什么用。

---

### 分层的JSON和带引用的JSON

尽管ItemFileReadStore实现的是Read API，但其中也内置了不少专有的特性，例如，特殊的数据格式、查询语法、反串行化特殊属性值的手段、匹配数据项的特殊标识符，等等。在解释这些特殊的地方之前，有必要先看一看ItemFileReadStore能够处理的数据示例。ItemFileReadStore能够处理的数据根据如何表示嵌套关系可以分为两种基本格式：分层的JSON（hierarchical JSON）和带引用的JSON（JSON with references），分层的JSON中直接嵌套具体的数据项的实例，而带引用的JSON中嵌套的则是指向实际数据项的引用。

为了说明这两种格式的不同，下面来看一看用这两种格式表示的一个简单的数据集示例。首先，是分层的JSON格式：

```
{
    identifier : id,
    items : [
        {
            id : 1, name : "foo", children : [
                {id : 2, name : "bar"},
                {id : 3, name : "baz"}
            ]
        }
        /* more items... */
    ]
}
```

接下来，是带引用的JSON格式：

```
{
    identifier : id,
    items : [
        {
            id : 1, name : "foo", children : [
              {_reference: 2},
```

```
                }{_reference: 3}
            ]
        },
        {id : 2, name : "bar"},
        {id : 3, name : "baz"}
        /* more items... */
    ]
}
```

在这个简单的数据集中，foo项有两个子数据项。在分层的JSON格式中，这两个子数据项被直接嵌套在foo项之下，而在带引用的JSON格式中，则只包含了指向每个数据项标识符的引用。带引用的JSON格式以其灵活性见长，因为这种格式允许某数据项拥有多个父项，而且所有数据项都可以作为顶级项出现。这两个优势恰恰是许多实际应用中所需要的。

---

**注意：** 第15章介绍的Tree Dijit部件充分利用了带引用的JSON格式，是体现该格式灵活性（及某些不足）的典型范例。

---

## 认识ItemFileReadStore

为了理解和熟悉ItemFileReadStore，下面我们看一看例9-1展示的以分层的JSON格式表示的一个数据集。在该数据集中，每个数据项的标识符为name，而且identifier、label和items是最外层唯一的几个键。

例9-1：有关咖啡的数据集示例

```
{
    identifier : "name",
    label : "name",
    items : [
        {name : "Light Cinnamon", description : "Very light brown, dry , tastes like
toasted grain with distinct sour tones, baked, bready"},
        {name : "Cinnamon", description : "Light brown and dry, still toasted grain with
distinct sour acidy tones"},
        {name : "New England", description : "Moderate light brown , still sour but not
bready, the norm for cheap Eastern U.S. coffee"},
        {name : "American or Light", description : "Medium light brown, the traditional
norm for the Eastern U.S .."},
        {name : "City, or Medium", description : "Medium brown, the norm for most of the
Western US, good to taste varietal character of a bean."},
        {name : "Full City", description : "Medium dark brown may have some slight oily
drops, good for varietal character with a little bittersweet."},
        {name : "Light French", description : "Moderate dark brown with oily drops, light
surface oil, more bittersweet, caramelly flavor, acidity muted."},
        {name : "French", description : "Dark brown oily, shiny with oil, also popular for
```

```
espresso; burned undertones, acidity diminished"},
        {name : "Dark French", description : "Very dark brown very shiny, burned tones
become more distinct, acidity almost gone."},
        {name : "Spanish", description : "Very dark brown, nearly black and very shiny,
charcoal tones dominate, flat."}
    ]
}
```

假设这些数据保存在磁盘上一个名叫coffee.json的文件中，例9-2所示的页面将加载这个文件并将其中的数据赋予JavaScript全局变量coffeeStore。

例9-2：*以编程方式加载ItemFileReadStore*

```
<html>
    <head>
        <title>Fun with ItemFileReadStore!</title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.data.ItemFileReadStore");

            dojo.addOnLoad(function() {
                coffeeStore = new dojo.data.ItemFileReadStore({url:"coffee.json"});
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

虽然第11章才会正式介绍parser（解析器），但parser的确太常用了，因此例9-3展示了使用parser解析标记的页面，该页面也能够实现与例9-2相同的效果。

例9-3：*在标记中加载ItemFileReadStore*

```
<html>
    <head>
        <title>Fun with ItemFileReadStore!</title>
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dojo.data.ItemFileReadStore");
        </script>
    </head>
    <body>
```

```
        <div dojoType="dojo.data.ItemFileReadStore" url="./coffee.json"
          jsId="coffeeStore"></div>
    </body>
</html>
```

不论如何声明数据存储对象，它们API的工作原理都是一样的。因此，读者应该多花几分钟时间在Firebug控制台中以现有的数据集为样本练习一下相关API的用法。本节剩余部分将展示很多命令和代码片段，供读者通过学习ItemFileReadStore来熟悉Read和Identity API。

---

**注意**：在创建数据存储对象时，除了通过url参数为ItemFileReadStore传递一个指向数据集所在文件的路径，也可以通过data参数为它传递一个引用JavaScript对象的变量。

---

根据唯一标识符取得数据项。从ItemFileReadStore中取得数据项的方式有两种，但这两种方式非常类似。要通过标识符取得数据项，应该使用Identity API的fetchItemByIdentity方法，该方法接收一组命名的参数，包括标识符、当取得数据项时执行什么操作以及发生错误时执行什么操作的回调函数。例如，要在coffeeStore中查询Spanish咖啡，可以像例9-4中所示的那样做。

例9-4：根据唯一标识符取得数据项并检查该项

```
var spanishCoffeeItem;
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        //捕获item……或者对它执行某些操作
        spanishCoffeeItem = item;
    },
    onError : function(item, request) {
        /* 处理错误…… */
    }
});

// 现在可以对spanishCoffeeItem 执行某些操作了……
//比如取得它的描述（description）……
coffeeStore.getValue(spanishCoffeeItem, "description"); //Very dark brown...

//或取得它的名称（name）……
coffeeStore.getValue(spanishCoffeeItem, "name"); // Spanish

//对于这个数据集而言，名称（name）和标签（label）都一样……
coffeeStore.getLabel(spanishCoffeeItem); // Spanish

//如果不知道某个数据项的标识符……
coffeeStore.getIdentity(spanishCoffeeItem); //Spanish
```

警告：初次接触数据API的读者很容易混淆数据项的唯一标识符与数据项本身，这种概念的模糊很可能导致不易察觉的语义bug，因为代码看起来似乎始终“很正确”。例如，通过 `item = coffeeStore.fetchItemByIdentity("Spanish")` 来取得Spanish咖啡的数据项似乎很有道理。但是，仔细看一看相应的API就会发现，这行代码中至少存在两处问题：一是该调用不会返回数据项；二是应该传递一组命名参数，而不仅是一个唯一标识符的值。

**根据任意条件取得数据项。**如果想根据除唯一标识符之外的其他条件取得数据项，那么可以使用更通用的fetch方法，而不是 `fetchItemByIdentity`。例如：

```
coffeeStore.fetch({
    query: {name : "Spanish"},
    onItem : function(item, request){console.log(item);}
});
```

除了接收完全限定的属性值之外，`fetch`方法也可以处理少量但非常实用的筛选条件，从而实现基本的regex（正则表达式）风格的匹配。例如，要查找咖啡的描述（description）中包含单词“dark”的数据项，且不区分大小写，就可以通过例9-5所示代码来达到目的。

例9-5：根据任意条件取得数据

```
coffeeStore.fetch({
    query: {description : "*dark*"},
    queryOptions:{ignoreCase : true},
    onItem : function(item, request) {
        console.log(coffeeStore.getValue(item, "name"));
    }
    /* 这里可以包含其他回调函数……*/
});
```

警告：应该通过数据存储对象调用`getValue`方法来取得数据项的属性值，而不要直接通过数据项来访问属性，因为存储对象的底层实现很可能不支持直接访问。例如，在`onItem`回调函数中像 `onItem: function(item, request) { console.log(item.name); }` 这样访问一个数据项的属性是不允许的。这种抽象不仅为数据存储模式提供了底层的缓冲机制，而且也为优化存储模式的性能准备了条件。

如果读者要实现自己的数据存储模式，应该了解一下`dojo.data.util.filter`，因为它也可以提供regex风格的匹配，就和`ItemFileReadStore`在`fetch`方法中实现的一样。此外，`dojo.data.util.simpleFetch`也为其8个参数（`onBegin`、`onItem`、`onComplete`、`onError`、`start`、`count`、`sort`和`scope`）提供了相关的逻辑。

### 查询子数据项

目前我们使用的有关咖啡的数据集还比较简单，因为其所有数据项都分布在一维的列表中。例9-6在该数据集基础上为有的数据项添加了一些子项，得到了带有嵌套数据项的结构。由于ItemFileReadStore明确要求使用children属性来维护一组子数据项，因此我们通过带引用的JSON格式创建了这个按照不同的烘焙方式（roast）进行分类的咖啡数据集。注意，我们有意将Light French同时放在Medium Roasts和Dark Roasts中，以展示引用的灵活性。鉴于每个数据项都必须拥有唯一标识符，因此使用其他格式表示一个子数据项同属于多个父项是不可能的。

**注意：** 虽然本章使用的数据存储模式最多只嵌套了两层，但我们示范的API却适用于嵌套了任意层的数据集。

**例9-6：具有嵌套结构的咖啡数据集**

```
{
    identifier : "name",
    items : [
        {
            name : "Light Roasts",
            description : "A number of delicious light roasts",
            children : [
                {_reference: "Light Cinnamon"},
                {_reference: "Cinnamon"},
                {_reference: "New England"}
            ]
        },

        {
            name : "Medium Roasts",
            description : "A number of delicious medium roasts",
            children : [
                {_reference: "American or Light"},
                {_reference: "City, or Medium"},
                {_reference: "Full City"},
                {_reference: "Light French"}
            ]
        },

        {
            name : "Dark Roasts",
            description : "A number of delicious dark roasts",
            children : [
                {_reference: "Light French"},
                {_reference: "French"},
                {_reference: "Dark French"},
```

```
            {_reference: "Spanish"}
        ]
    },
    {name : "Light Cinnamon", description : "Very light brown, dry , tastes like
toasted grain with distinct sour tones, baked, bready"},
    ...
]
}
```

查询数据项的子项是很常见的任务。在此，我们要做的是查询指定烘焙方式下每种咖啡的名称。例 9-7 展示了查询 Dark Roasts 数据项的代码。

例 9-7：取得一个数据项并迭代其子项

```
coffeeStore.fetch({
    query: {name : "Dark Roasts"},
    onItem : function(item, request) {
        dojo.forEach(coffeeStore.getValues(item, "children"), function(childItem) {
            console.log(coffeeStore.getValue(childItem, "name"));
        });
    }
});
```

下面稍加解释一下。首先，我们通过 fetch 方法查询父数据项 Dark Roasts，待查询返回后再使用 getValues 方法取得多值的 children 属性。然后，通过 dojo.forEach 方法迭代每个子数据项，而在最终访问子数据项的 name 属性时，必须记住要使用 getValue 方法。

注意，数据集中表示子数据项的{_reference: someIdentifier}涉及实现细节。由于 _reference 并非一个可以查询的 _reference 属性，而只是内部数据引用的一种标准方式，因此不能基于 _reference 属性进行查询。只要不是从 dojo.data 开发人员的角度考虑问题，那么就应该把 dojo.data 模式中的一切都当作纯粹的数据项。

现在，读者可能已经体会到了，ItemFileReadStore 确实非常灵活，而且也很强大，它作为一种数据存储模式可以满足很多实际的需求 —— 特别是在要建立应用程序原型，但又来不及开发自定义实现的情况下。在一切从简的条件下，通过服务器端程序生成适当的数据格式，而在客户端使用 dojo.data 来处理这些数据并非难事。不过，别忘了你随时都可以根据需要继承或扩展 ItemFileReadStore，甚至，也实现自己的数据存储模式。

### ItemFileWriteStore

勿庸置疑，好的数据抽象能够显著减少服务器提供的数据，以及减小显示这些数据的工作量；但是，当这些数据在客户端被修改后，也很少有不需要将它们写回服务器的时候，

而这正是ItemFileWriteStore的用武之地。与ItemFileReadStore为读取数据集提供的良好抽象一样，ItemFileWriteStore则为创建新数据项、删除数据项以及修改数据项提供了类似的抽象。在遵从dojo.data API方面，ItemFileWriteStore实现了所有Read、Identity、Write和Notification接口。

为了帮助读者熟悉ItemFileWriteStore，我们仍将以coffee.json文件中保存的JSON数据为样本，像解释ItemFileReadStore那样来解释ItemFileWriteStore的各方面细节。相信读者到最后会发现，ItemFileWriteStore也没有什么难懂的地方，它不过是忠实地实现了本章前面介绍的API而已。

### 修改已有的数据项

如例9-8所示，使用setValue方法修改数据项的属性是一项常规性的操作。为此，必须为setValue方法传递相应的数据项、要修改的属性以及该属性的新值。如果相应的数据项没有指定的属性，那么会自动添加该属性。

例9-8：设置数据项的属性

```
//跟以前一样取得一个数据项……
var spanishCoffeeItem;
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});

//然后，添加一个新属性foo=bar
coffeeStore.setValue(spanishCoffeeItem, "foo", "bar");
coffeeStore.getValue(spanishCoffeeItem, "foo"); //bar

//同样，除了不可以修改唯一标识符外，可以修改其他任何属性的值
coffeeStore.setValue(spanishCoffeeItem, "description", "El Matador...?!?");
```

---

**警告：** 跟多数其他数据模式一样，修改数据项的唯一标识符通常没有什么意义，毕竟所谓唯一标识符是一种不变的特征。为了遵循这一成例，ItemFileWriteStore没有支持这种操作；并且，我们建议读者在自己的自定义实现中也要考虑到这一点。

---

这里有一点需要特别注意，即把一个属性设置为空字符串跟删除该属性不是一回事。而通过调用Write API的hasAttribute方法检查相应的属性是否存在就可以看出这两种操作的区别，请读者看一看例9-9。

例 9-9：设置和删除数据项的属性

```
coffeeStore.hasAttribute(spanishCoffeeItem, "foo"); //true
coffeeStore.setValue(spanishCoffeeItem, "foo", ""); //foo=""
coffeeStore.hasAttribute(spanishCoffeeItem, "foo"); //true
coffeeStore.unsetAttribute(spanishCoffeeItem, "foo"); //删除该属性
coffeeStore.hasAttribute(spanishCoffeeItem, "foo"); //false
```

本节前面的示例都与修改已有的数据项有关，但我们知道仅仅修改而不保存，还不是完整的操作。为此，ItemFileReadStore在内部跟踪并维护着一个脏数据项（即已经被修改，但还没有保存的数据项）的集合。如例9-10所示，在修改了spanishCoffeeItem之后，可以调用isDirty方法来检查该数据项是否在修改后尚未保存。但是，当保存了被修改的数据项之后，它就不再是脏数据项了。在例9-10中，保存操作实际上更新的只是内存中的副本，稍后我们就会讨论如何把数据保存回服务器。

例 9-10：检查数据项是否为脏数据项

```
/* 首先修改spanishCoffeeItem…… */
coffeeStore.isDirty(spanishCoffeeItem); //true
coffeeStore.save(); //更新数据集在内存中的副本
coffeeStore.isDirty(spanishCoffeeItem); //false
```

在此，读者也许看不出通过明确的save（保存）操作来提交修改的价值所在。事实上，在执行保存操作之前，如果宏观事务中的某一操作产生了错误，或者发生了其他意外的中断，那么是可以恢复此前所作的修改。在关系型数据库中，这种恢复操作通常称为回滚。例9-11展示了恢复dojo.data数据集的操作，并且将其中细微但却非常重要的代码加粗显示，该行代码涉及包含数据项引用的本地变量。

例 9-11：ItemFileWriteStore 数据集中的恢复操作

```
var spanishCoffeeItem;
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});

coffeeStore.getValue(spanishCoffeeItem, "description"); //Very dark...

coffeeStore.setValue(spanishCoffeeItem, "description", "El Matador...?!?");

/* 现在，spanishCoffeeItem变量和数据集中都有修改后的描述（description属性）*/

coffeeStore.fetchItemByIdentity({
```

```
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});

coffeeStore.getValue(spanishCoffeeItem, "description"); //El Matador...?!?

coffeeStore.isDirty(spanishCoffeeItem); //true

coffeeStore.revert(); //revert the store.

// 在执行 revert()后，本地变量spanishCoffeeItem
// 就不再表示数据集中的一个数据项了
coffeeStore.isItem(spanishCoffeeItem); //false

//再次取得同一个数据项……
coffeeStore.fetchItemByIdentity({
    identity: "Spanish",
    onItem : function(item, request) {
        spanishCoffeeItem = item;
    }
});

coffeeStore.isDirty(spanishCoffeeItem); //false

coffeeStore.getValue(spanishCoffeeItem, "description"); //Very dark...
```

**警告：** 虽然从理论上讲，实现一个自定义的存储模式并通过dojo.connect或发布/预订通信在后台确保本地数据项不失效是有可能的，但是ItemFileWriteStore却没有做到这一点。如果读者认为有必要检查某个数据项引用是否有效，那么大可以使用isItem方法来得到可靠的结果。

### 创建和删除数据项

在掌握了上一节讨论的修改已有数据项的各种方法之后，接下来我们介绍如何添加和删除数据项。事实上，添加和删除数据项的原理与保存和恢复数据项的原理相同，确实没有太大的区别。如例9-12所示，我们首先向数据集中添加一个顶级数据项。添加数据项需要提供一个JSON对象，这个JSON对象应该与服务器提供的原始数据集具有相同的格式。

例9-12：从ItemFileWriteStore中添加和删除数据项

```
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane to get you through till
5 o'clock."
```

```
});

coffeeStore.isItem(newItem); //true
coffeeStore.isDirty(newItem); //true

/* 查询这个数据项、保存数据集、恢复数据集等等 */

// 或者删除这个数据项……
coffeeStore.deleteItem(newItem);
coffeeStore.isItem(newItem); //false
```

虽然添加和删除顶级数据项很简单，但如果要求这个顶级数据项能够以引用方式被添加为其他数据项的子项，就要复杂一些了。例9-13展示了这个过程。其基本思路是先创建顶级数据项，然后取得要将这个数据项加入其中的子数据项，最后再把它添加到取得的子数据项的集合中。

例 9-13：在带引用的 JSON 数据集中添加子数据项

```
// 创建新数据项
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane to get you thorugh till
5 o'clock."
});

// 取得带有子数据项的父项的引用
var darkRoasts;
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});

// 使用 getValues 取得子数据项
var darkRoastChildren = coffeeStore.getValues(darkRoasts, "children");

// 然后使用 setValues 将新数据项添加到子数据项的集合中
coffeeStore.setValues(darkRoasts, "children", darkRoastChildren.concat(newItem);

// 通过迭代子数据项可以看到新添加的子数据项……
dojo.forEach(darkRoastChildren, function(x) {
    console.log(coffeeStore.getValue(x, "name"));
});
```

---

**警告**：记住，在取得多值属性时要使用 getValues，而不是 getValue。

---

删除数据项也没有什么特别之处。不过，如果删除的顶级数据项包含子项，那么将只删除该顶级数据项，而保留子数据项，如例 9-14 所示。

例 9-14：从 ItemFileWriteStore 中删除顶级数据项

```
var darkRoasts;
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});

coffeeStore.deleteItem(darkRoasts);

coffeeStore.fetch({
    query : {name : "*"},
    onItem : function(item, request) {
        console.log(coffeeStore.getValue(item, "name"));
    }
});

/* 保存数据集，或者恢复数据集，或者…… */
```

显然，如果也要删除子数据项，那么可以先查询并删除其子数据项，然后再删除顶级数据项。

### 自定义保存

读者可能会想，在内存中保存数据集固然好，但要想把数据传送回服务器又该怎么办呢？事实上，ItemFileWrite Store 为此提供了一个 _saveCustom 扩展点（extension point），可以通过实现该扩展点自定义一个例程，以便在调用 save 方法时触发该例程并将数据传送回服务器；这样，除了更新内存中的本地副本并清除脏数据标志之外，也可以将数据同步上传到服务器或者执行其他有用的操作。实现这个扩展点会涉及到使用前面介绍的 API。不过，一般来说，“完整的保存操作”大概要涉及到迭代整个数据集、将数据项串行化为一种定制的格式（酷似执行 dojo.toJson 操作），然后，再将串行化之后的数据传送回服务器。正如 Write API 所规定的，在保存数据时可以提供一个对象参数，该参数包含可选的回调函数 onComplete 和 onError，分别会在保存成功和发生错误时被调用；还包含一个可选的 scope 参数，以便为上述两个回调函数提供执行环境。不过，这些关键字参数是传递给 save 方法的，不能传递给 _saveCustom 扩展点。

例 9-15 展示了如何实现 _saveCustom 处理程序，从而在 save() 被调用时将数据传送回服务器。正如读者所看到的，_saveCustom 处理程序所执行的不过是一种例行任务而已。

例 9-15：在 ItemFileWriteStore 中触发自定义的保存处理程序

```
<html>
    <head>
```

```
    <title>Fun with ItemFileWriteStore!</title>
    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>
    <script type="text/javascript">
        dojo.require("dojo.data.ItemFileReadStore");
        dojo.addOnLoad(function() {
            coffeeStore = new dojo.data.ItemFileReadStore({url:"coffee.json"} );
            coffeeStore._saveCustom = function() {
                /* 此处是将数据传送回服务器的代码。每当调用 save()时就会执行这些代码 */
            }
        });
    </script>
</head>
<body>
</body>
</html>
```

事实上，_saveCustom的使用并没有我们想像得那么频繁，因为调用它通常意味着要把所有数据传送回服务器，而这种需求除非是在批量初始化空数据库时才会产生。多数情况（特别是要操作非常庞大的数据集时），下还是使用我们下一节介绍的Notification API来处理少量的数据变化更为合适。

### 对通知作出响应

在本节、也是本章的最后，我们简单地介绍一下ItemFileWriteStore实现的Notification API。当需要响应创建、删除和修改数据项的相关通知（notification）时，可以分别调用onNew、onDelete和onSet回调函数。

鉴于读者对于前面介绍的API已经非常熟悉，也许都可以按照相应API实现自己的数据存储模式了，因此对于下面示例中添加、删除和修改数据项的代码，我们就不再多作解释。不过，还是要提醒大家一句，例9-16是在例9-13的基础上经过修改而成的。

例9-16：使用ItemFileWriteStore实现的Notification API

```
/* 通知处理程序开始 */
coffeeStore.onNew = function(item, parentItem) {
    var itemName = coffeeStore.getValue(item, "name");
    console.log("Just added", itemName, "which had parent", parentItem);
}

coffeeStore.onSet = function(item, attr, oldValue, newValue) {
    var itemName = coffeeStore.getValue(item, "name");
    console.log("Just modified the ", attr, "attribute for", itemName);
```

```
/* 由于children是多值属性，因此oldValue和newValue是需要频繁迭代和检查的数组，
可以只将newValue传送回服务器，以记录更新 */
}

coffeeStore.onDelete = function(item) {
    // coffeeStore.isItem(item)返回false，因此不能访问这个数据项
    console.log("Just deleted", item);
}
/* 通知处理程序结束 */

/* 使用上面处理程序的代码 */

// 添加一个顶级数据项——触发一个通知
var newItem = coffeeStore.newItem({
    name : "Really Dark",
    description : "Left brewing in the pot all day...extra octane to get you thorugh till
5 o'clock."
});

var darkRoasts;
coffeeStore.fetchItemByIdentity({
    identity : "Dark Roasts",
    onItem : function(item, request) {
        darkRoasts = item;
    }
});

var darkRoastChildren = coffeeStore.getValues(darkRoasts, "children");

// 修改这个数据项——触发一个通知
coffeeStore.setValues(darkRoasts, "children",darkRoastChildren.concat(newItem));

// 删除这个数据项——触发一个通知
coffeeStore.deleteItem(newItem)
```

运行这个示例应该看到如下输出结果：

```
Just added Really Dark, which had parent null
Just modified the children attribute for Dark Roasts
Just modified the children attribute for Dark Roasts
Just deleted Object _0=13 name=[1] _RI=true description=[1]
```

也就是说，在创建顶级数据项时我们会收到相应的通知，在把新数据项作为子项添加到另一个数据项的children属性中时会收到一个通知，而在移除该子数据项时会收到另一个通知，最后，在删除这个数据项时也会收到一个通知。

---

注意：关于例9-16有一个问题需要读者注意：由于在onDelete通知中取得的数据项的引用已经从数据集中被删除了，即不能再合法地调用其常规的方法，因而对它的操作会受到限制。

---

## 串行化和反串行化自定义数据类型

尽管一直没有提到，但想必读者也一定猜得到ItemFileReadStore和ItemFileWrite-Store提供的另一个特性：串行化和反串行化自定义数据类型。由于经常要处理不是原始数据类型、对象直接量和数组的属性值，因此就要用到类型映射（type map）。在这些情况下，要么手工来构建各种属性（同时也在核心逻辑中埋下问题的隐患），要么利用串行化逻辑自动解决问题。

### 隐式类型映射

当数据中出现_type和_value这两个特殊属性时，就会发生隐式类型映射。其中，_type指定被调用的构造函数，而在调用该构造函数时以_value的值作为参数。JavaScript内置的Date对象是得益于类型映射的最常见的数据类型。例9-17展示了在前面数据集基础上经过修改得到的一个使用日期值的数据项的示例。

例9-17：使用自定义类型映射反串行化属性值

```
...
{
    name : "Light Cinnamon",
    description : "Very light brown, dry , tastes like toasted grain with distinct sour
tones, baked, bready"
    lastBrewed : {
        '_type' : "Date",
        '_value':"2008-06-15T00:00:00Z"}
    }
}
...
```

这看起来似乎太简单了，不过在Date构造函数已有定义的情况下，就这么简单！当数据被反串行化之后，lastBrewed属性的所有值都将忠实地成为一个Date对象——而不再是字符串：

```
var coffeeItem;
coffeeStore.fetchItemByIdentity({
    identity : "Light Cinnamon",
    onItem : function(item, request) {
        coffeeItem = item;
    }
});
coffeeStore.getValue(coffeeItem, "lastBrewed"); //一个真正的Date对象
```

### 自定义类型映射

除了使用内置对象之外，也可以自定义JavaScript对象并提供一个命名的deserialize

（反串行化）函数和一个用于构建该值的type参数。对于ItemFileWriteStore而言，也可以指定serialize（串行化）函数。下面继续以操作前面的Date对象为例，基于ItemFileWriteStore自定义的可以作为类型映射传入的JavaScript对象如例9-18所示。

例9-18：向ItemFileReadStore中传入一个自定义的类型映射

```
dojo.require('dojo.date');
dojo.addOnLoad(function() {
    var map = {
        "Date": {
            type: Date,
            deserialize: function(value){
                return dojo.date.stamp.fromISOString(value);
            },
            serialize: function(object){
                return dojo.date.stamp.toISOString(object);
            }
        }
    };

    coffeeStore = new dojo.data.ItemFileReadStore({
        url:"coffee.json",
        typeMap : map
    });
});
```

---

注意：虽然本章未深入介绍dojox.data，但如果不提供一些dojox.data.QueryRead-Store的参考资料恐怕有失周全。读者可以参考*http://www.oreillynet.com/onlamp/blog/2008/04/dojo_goodness_part_6_a_million.html*中有关这个存储模式的使用示例，其中还涉及到了自定义的服务器端程序。这个独特的示例展示了如何通过赫赫有名的DojoX Grid部件显示上百万条数据记录。

---

# 小结

在学习了本章之后，读者应该：

- 熟悉dojo.data API并理解每个API的基本价值。
- 理解Read、Identity、Write和Notification API是抽象的，任何实现都有可能。
- 知道dojox.data子项目提供了一些真正有用的自定义数据存储模式，可以用来操作CSV、Flickr等常见的数据。

- 知道 Dojo 工具箱提供了通用但十分强大的 `ItemFileReadStore` 和 `ItemFileWriteStore`作为`dojo.data`的实现，既可以对这两个实现进行自定义，也可以在它们的基础上开发自己的实现。
- 理解使用自定义类型映射节省手工串行化和反串行化数据类型的价值。

下一章，我们将介绍模拟类和继承。

# 第10章

# 模拟类和继承

尽管JavaScript中的Function对象与类近似，而且JavaScript也提供了基于原型的继承机制，但这与Java或C++中的类和继承仍然存在差异。Dojo基于本地JavaScript实现提供了模拟类和基于类的继承机制。本章将深入讨论Dojo工具箱中用于模拟类和继承的重要方法`dojo.declare`，并借此为理解第二部分要介绍的Dijit实现原理打下良好的基础。

## JavaScript不是Java

在深入讨论通过Dojo模拟继承结构和类之前，必须首先认识到JavaScript不是Java，而Dojo也没有试图对抗JavaScript的风格并越俎代疱地重写底层的某些实现，一切都顺其自然！JavaScript依然是动态的、弱类型语言，而Java则是强类型语言，拥有真正的类和基于类的继承结构，且类的继承关系在编译时确定。但是，JavaScript拥有的原型式继承可以用来模拟类，而且完全是解释执行的。

Dojo在保持JavaScript本来面目的基础上，从最具互补性的角度利用了相关特性，做到了与该语言的发展相辅相成，消除了因存在大量冗余代码而带来的维护工作量。最终，它为开发人员提供了高效敏捷的实现，为适应当代技术领域“早发布、勤更新”的开发理念打下了基础。

除了在第13章介绍的表单部件之外，Dojo中没有多少面向对象的设计，因为这不是Dojo开发的指导思想。Dojo开发的指导思想是立足原型式继承，通过Base提供的`mixin`和`extend`方法来发挥JavaScript的长处。同时，Dojo也从实事求是的角度出发，在某些领域中引入了面向对象的设计，否则这些领域的目标将难于实现。而之所以将本章作为第一部分的结尾，真正的原因是第二部分将要介绍Dijit，而Dijit则非常适合基于类进行设计。

在阅读本章时，请读者务必牢记我们这里说过的话，因为如果你长期习惯于运用Java或C++等面向对象语言中的类，那么很可能会对Dojo寄予不应有的期望，并试图把任何东西都转换为继承关系，而这无论从风格还是从性能上来说，都不是一件自然而然的事。

### 要顺其自然

JavaScript作为一门动态类型的解释性语言，其中的很多特性并没有想像得那么容易理解。特别是对于具有C++或Java等静态类型的编译性语言背景的人来说，要适应它的风格必须付出一番努力才行。如果读者在实际开发中发现自己对某个问题或bug的起因百思不解，那么最好重温一下相关的基本特性，确保理解了其中的微妙之处，并尽可能地在开发中利用这门语言的内在特性。

虽然把某个算法的代码从Java等面向对象语言逐行地转换成JavaScript不是不可能，但编程风格的大变却会导致时间成本、代码易维护性、运行效率以及可靠性等诸多问题。因此，要顺其自然；首先必须理解JavaScript，然后再考虑如何有效地使用它。

## 一题多解

如果读者恰好对在JavaScript中如何实现继承存有疑问，那本节就来展示解决一个问题的多种途径，以多种方式为Circle（圆）对象建模。本节的示例除了展示非常规的思维之外，还运用了在第2章中介绍的一些语言工具。

### 典型的JavaScript继承

相当长一段时间以来，JavaScript开发人员都在使用Function对象来模拟类，有时候对发挥这门语言的优势并不利，但有时候则能有效地解决某个特定的问题。JavaScript中Function对象的存在目的就是为模拟类提供一个立足点。即，它的实例可以作为构造函数与new运算符连用以创建对象实例，同时也为创建的对象实例提供一个模板。

为了说明这一点，例10-1展示了在JavaScript中模拟简单的Shape类的代码片段。注意，类按照约定一般都以大写字母开头。

---

**注意：** 为了简单起见，本章的示例没有使用命名空间来限定对象。但是，使用命名空间在实际开发中已经成为一种通用的做法，第12章我们就重新开始使用命名空间。

---

例 10-1：典型的 JavaScript 类

```
//定义类
function Shape(centerX, centerY, color)
{
  this.centerX = centerX;
  this.centerY = centerY;
  this.color = color;
};

// 创建实例
s = new Shape(10, 20, "blue");
```

这样，当 JavaScript 解释器执行到这个函数定义之后，内存中就会生成一个作为原型的 Shape 对象；而当使用 new 运算符调用这个构造函数时，就能基于它来创建实例。

在 Dojo 工具箱中，也可以使用 Base 的 extend 方法以更简洁的方式来定义 Shape 对象：

```
// 创建 Function 对象
function Shape() {}

// 以一些合理的默认值扩展其原型
dojo.extend(Shape, {
    centerX : 0,
    centerY : 0,
    color : ""
});
```

但是，只创建 Shape 对象并不是我们的最终目的，我们的最终目的是创建一种具体的形状（shape）—— 例如，圆形（circle）。虽然也可以重新创建一个新的圆形类，但更容易维护的做法则是让圆形类继承前面已经定义的形状类，毕竟圆形是一种形状。并且，既然我们已经定义了一个 Shape 类，那为什么不把它派上用场呢？

例 10-2 展示了在 JavaScript 中实现上述继承关系的一种方式。

例 10-2：典型的 JavaScript 继承

```
// 定义子类
function Circle(centerX, centerY, color, radius)
{
  // 为了确保获得超类的属性，首先要把超类的构造函数
  // 赋予子类的一个属性，再在子类中调用这个构造函数
  this.base = Shape;
  this.base(centerX, centerY, color);

  // 为子类定义其他的属性
  this.radius = radius;
};
```

```
// 通过将子类的原型对象重写为超类的实例，可以确保
// 动态添加到超类中的属性能够在子类中得到反映
Circle.prototype = new Shape;

// 创建实例
c = new Circle(10, 20, "blue", 2);

// 这样，Circle作为一种具体的形状，就继承了Shape
```

虽然这个示例可以作为一个有意思的练习，但却不如我们想像得那么简单明了。而且，这种继承方式在真正高级的Web应用程序中恐怕也难以成为主流方式。

## 混入模式

为了展示对典型继承方式的改进，例10-3以一种不同的方式构建了Shape和Circle类。其中，特别值得注意的是混入过程对鸭子类型及“有一个”关系的利用。如前所述，所谓鸭子类型就是指某物叫起来像鸭子，走起路来也像鸭子，那么就可以把该物当作鸭子。在目前的情况下，鸭子类型的比喻可以具体化为：如果某个对象中存在我们想要的Shape或Circle的属性，那么把这个对象看成是Shape或Circle也不为过。换句话说，无论这个对象到底是什么，只要它具有合适的属性就足够了。

例10-3：通过混入实现继承

```
//创建一个简单的对象，作为混入的目标
var shape = {}

//混入需要让这个对象“看起来像形状对象（鸭子）”的属性
dojo.mixin(shape, {
    centerX : 10,
    centerY : 20,
    color : "blue"
});

//如果以后还需要什么属性，没问题，继承混入即可
dojo.mixin(shape, {
    radius : 2
});

//这样，shape对象又有了radius属性
```

实事求是地讲，这个混入的示例并不是对前面原型继承方式的严格替代方案；相反，我们举出这个示例的意图，只是为了说明解决同一个问题的方法并不是唯一的。

## 委托模式

下面，我们再通过例10-4展示构建Shape与Circle之间关系的另一种手段。与混入模式

向一个对象实例中复制属性不同，委托（delegation）模式的原理则是向拥有相应属性的对象转嫁责任。

例 10-4：通过委托实现继承

```
// 创建一个简单的对象
var shape = {}

// 向这个实例中混入所需的属性
dojo.mixin(shape, {
    centerX : 10,
    centerY : 20,
    color : "blue"
});

// 将为circle提供centerX、centerY和color属性的责任委托给shape
// 与此同时，直接为circle混入radius属性
circle = dojo.delegate(shape, {
    radius : 2
});
```

这种实现方式的核心在于，对象直接量中的radius属性被直接混入了circle中，但其他形状（shape）的公有属性则没有混入circle，而是委托给了shape。也就是说，每当读取circle中没有定义的属性时，最终都会由shape提供该属性的值。总结起来，就是以下几项：

- 读取radius属性时，由circle直接提供，因为该属性已经被混入其中。
- 读取centerX、centerY和color属性时，则由被委托的shape提供，因为它们在circle中不存在（粗略地说）。
- 读取其他属性则返回undefined，因为根据定义circle和shape中不存在其他属性。

尽管以上示例都比较简单，但不难看出，混入模式具有广泛的应用前景，而委托模式当然也有自己的适用场合，特别是在大量对象都要共享同一组属性的情况下。

## 使用 Dojo 来模拟类

前面，我们探讨了实现对象继承的一些可能的途径。接下来，我们介绍Dojo工具箱中为声明（declare）类和模拟各种继承关系提供的实用方法。Dojo把与实现类的声明和继承有关的大量细节都封装到了一个优雅的小方法中，这个方法就是Base提供的dojo.declare。这个方法很容易记住，只要把创建类想像成要声明它即可。表10-1展示了dojo.declare方法的API。

表 10-1：dojo.declare API

| 名称 | 说明 |
| --- | --- |
| `dojo.declare (/*String*/ className, /*Function\|Function[]*/ superclass, /*Object*/ props)` | 提供了声明构造函数的一种简洁方式。其中，参数 `className` 是要创建的构造函数的名称，而参数 `superClass` 既可以是一个 Function 对象，也可以是一个 Function 对象的数组，用于表示当前类所要继承的超类（列表）。另外，参数 `props` 是一个对象，其中的属性将被复制到构造函数的 prototype 属性中 |

**注意：** 实际上，`declare` 是在 `extend`、`mixin` 和 `delegate` 等方法的基础上提供了一种更复杂的抽象机制，从而能够实现单独使用这几种模式无法实现的效果。

例 10-5 展示了如何使用 `dojo.declare` 构建 Shape 和 Circle 之间的继承关系。目前，读者只需孤立地看待这个示例，稍后我们会介绍 `dojo.declare` 的更高级应用。

例 10-5：使用 dojo.declare 模拟基于类的继承

```
// “声明”Shape 构造函数
dojo.declare(
  "Shape", // 类名
  null, // 没有超类，因此以 null 作为占位符
  {
    centerX : 0, // 属性
    centerY : 0,
    color : "",

    // 执行 "new Shape" 时要调用的构造函数
    constructor : function(centerX, centerY, color)
    {
      this.centerX = centerX;
      this.centerY = centerY;
      this.color = color;
    }
  }
);

// 此时，通过 var s = new Shape(10, 20, "blue");
// 可以创建一个 Shape 的实例

// “声明”Circle 构造函数
dojo.declare(
```

```
"Circle", //类名
Shape, //超类
{
  radius : 0,

  //执行"new Circle"时要调用的构造函数
  constructor : function(centerX, centerY, color, radius)
  {
    // Shape的构造函数会以这些参数被自动调用。
    // 但届时，将使用前3个参数，而忽略radius
    this.radius = radius; //为Circle的属性赋值
  }
}
);

// JavaScript构造函数中的参数将被传递给
// dojo.declare中的constructor
c = new Circle(10,20,"blue",2);
```

通过这个示例，读者会发现使用dojo.declare模拟类更加清晰、易维护，而且容易理解。如果能够在代码中恰当地运用空白和换行符，那么这些代码甚至会像是使用基于类的编程语言写出来的一样。其中唯一一处不为人注意的地方，就是传递给Circle的constructor的参数同样也会传递给Shape的constructor。不过，这不会造成任何问题，因为Shape的constructor只接收前3个参数，而忽略其他参数。(有关这一点稍后还会讨论到。)

## JavaScript函数的参数

在许多编程语言中，如果没有事先声明，都不能传递任意数量的参数。例如，在C++中，需要使用...符号来表示参数的个数不定。

但是，JavaScript的函数则可以接收任意个参数。如果接收到的参数比声明的多，就会忽略多出来的参数；如果接收的参数比声明的少，那么缺少参数的值就是JavaScript的一种类型：undefined。

当然，要想访问传入的所有参数，可以使用特殊的arguments变量。但是我们想提醒读者，虽然可以像访问数组一样从arguments变量中读取参数，但arguments并非真正的数组。例如，arguments没有push或pop方法。

不过，在与继承有关的多数情况下，由于我们要用的都是Dojo类的constructor函数中定义的部分参数(如前面的例10-5所示)，因此基本不会用到arguments变量。

**注意：** 这里，出现在dojo.delare方法第三个参数中的constructor，比较容易跟创建JavaScript对象时通过new调用的构造函数混淆。其实，要区分这两个概念也很简单，在本书中，JavaScript的构造函数就是构造函数，而dojo.delare的参数则是constructor。

## 基本的类创建模式

dojo.declare方法提供了一种处理类的基本能力，理解这种能力非常重要，因为Dijit在这种能力基础上衍生出了一种创建部件的灵活模式，该模式能够自动完成部件创建过程中的所有细节。第12章将继续深入探讨这个主题。

虽然本章主要介绍最常用的constructor函数，但下面的模式将展示dojo.declare提供的另外两个函数：preamble和postscript。其中，preamble在constructor之前执行，而postscript则在constructor之后执行。

```
preamble(/*Object*/ params, /*DOMNode*/node)
    // 先于constructor执行

constructor(/*Object*/ params, /*DOMNode*/node)
    // 触发superclass构造函数
    // 触发混入的构造函数
    // 触发当前类的构造函数——如果有

postscript(/*Object*/ params, /*DOMNode*/node)
    // 主要用于触发部件的创建
```

为了验证这几个方法执行的先后次序，读者可以试一试例10-6。

例10-6：Base中dojo.declare的创建模式

```
dojo.addOnLoad(function() {
    dojo.declare("Foo", null, {

        preamble: function() {
            console.log("preamble", arguments);
        },

        constructor : function() {
            console.log("constructor", arguments);
        },
        postscript : function() {
            console.log("postscript", arguments);
        }
});

    var foo = new Foo(100); //calls through to preamble, constructor, and postscript
});
```

创建类时的大多数操作通常都发生在constructor中，而preamble和postscript函数也有各自的用途。preamble主要用于处理超类的参数。虽然在这个示例中，我们通过JavaScript构造函数（即new Foo(100)）传入的参数依次被传给了preamble、constructor和postscript，但在实际构建继承关系时也不一定总是这种情况。在下一节正式介绍完继承之后，我们将在本章的附言栏“高级参数处理”中再次讨论这个主题。postscript主要用于触发部件的创建过程。第12章差不多完全围绕着部件的生命周期展开介绍。

## 单继承示例

下面，我们通过深入一些的示例来进一步展示dojo.declare的能力。第一个示例主要展示了使用dojo.delare内部的constructor函数的重要差别，请读者注意看代码中的注释：

```
<html>
   <head>
        <title>Fun with Inheritance!</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
          dojo.addOnLoad(function(){
              // JavaScript中简单的Function对象的定义
              function Point(x,y)  {}
              dojo.extend(Point, {
                  x : 0,
                  y : 0,
                  toString : function() {return "x=",this.x," y=",this.y;}
});

dojo.declare(
    "Shape",
    null,
  {
      //首先明确地定义成员，但要在Dojo的constructor中初始化它们。
      //记住，不能在这个关联数组中初始化Function对象，除非想让这些
      //成员的值被当前类的所有实例共享——但事实上，这种情况并不多见

      //变量名前加下划线是表示该变量为私有成员，这是一个常见约定

      _color: "",
      _owners: null,

      //Dojo为当前类提供了一个特殊的构造函数，即下面的constructor函数。
```

```
        //注意，这个构造函数在被调用时，也会接收到调用Circle构造函数时
        //传入的相同参数——尽管届时不会直接调用这个超类构造函数

        constructor: function(color)
        {
            this._color = color;
            this._owners = [0]; //参见下面的注释了解与初始化对象有关的说明
            //objects
            console.log("Created a shape with color",
              this._color, "owned by", this._owners);

        },

        getColor : function() {return this._color;},
            addOwner : function(oid) {this._owners.push(oid);},
            getOwners : function() {return this._owners;}

            //在最后一个元素后面不要加逗号。并非所有浏览器都对此处的逗号视而不见
            //——否则，很可能引发一条错误消息。
            //建议读者把这条注释铭记在心!
        }

);

//重要约定：
//对于单继承链而言，必须在子类的constructor函数中首先列出
//超类的参数，然后再依次列出子类（当前类）的参数。
//这样，当以全部参数调用子类的constructor函数时，由于参数
//的先后顺序没有问题，并且假设我们不会在子类的construcotr
//函数中操作超类的参数，那么一切都会保持正常

//记住dojo.declare的第一个参数是一个字符串，而第二个参数则
//是一个Function对象
dojo.declare(
"Circle",
Shape,
{
    _radius: 0,
    _area: 0,
    _point: null,

    constructor : function(color,x,y,radius)
    {
        this._radius = radius;
        this._point = new Point(x,y);
        this._area = Math.PI*radius*radius;

        //注意，此时已经可以使用继承的成员_color了!
        console.log("Circle's inherited color is " + this._color);
    },

    getArea: function() {return this._area;},
```

```
            getCenter : function() {return this._point;}
        }
    );

    console.log(Circle.prototype);

    console.log("Circle 1, coming up...");
    c1 = new Circle("red", 1,1,100);
    console.log(c1.getCenter());
    console.log(c1.getArea());
    console.log(c1.getOwners());
    c1.addOwner(23);
    console.log(c1.getOwners());

        console.log("Circle 2, coming up...");
        c2 = new Circle("yellow", 10,10,20);
        console.log(c2.getCenter());
        console.log(c2.getArea());
        console.log(c2.getOwners());
    });
    </script>
</head>
<body>
</body>
</html>
```

**警告：** 在此，我们重申一下注释中的提醒：对象直接量中最后一个元素结尾的逗号，很可能在除Firefox之外的浏览器中制造麻烦。有些语言允许这种结尾的逗号，例如，Python；因此，如果读者经常使用类似的语言，那么一定要保持警惕。

图10-1展示了运行这个示例之后，Firebug控制台中显示的输出结果。

这里需要读者格外注意的一点是，当dojo.declare语句执行完毕后，内存中就会生成一个Function对象（一个实实在在的后台Function对象），该对象的原型中保存着dojo.declare方法第三个参数中的所有变量。结果，这个对象就成为将来创建所有对象的原型对象。如果读者对基于原型的继承没有完全搞清楚，那么很可能对此心存疑惑。不过，这正是我们下一节所要介绍的主题。

### 基于原型继承的一个常见陷阱

读者在前面的示例中已经看到了，Point的定义与Dojo没有任何关系，它只是一个简单的Function对象。正因为如此，我们不应该在Shape的关联数组参数中与其他属性一道来初始化Point。如果这样做了，那么它就会像一个静态成员一样存在于将来创建的所有Shape对象实例中，假如我们再不留意，则很可能因此导致意料之外的行为。

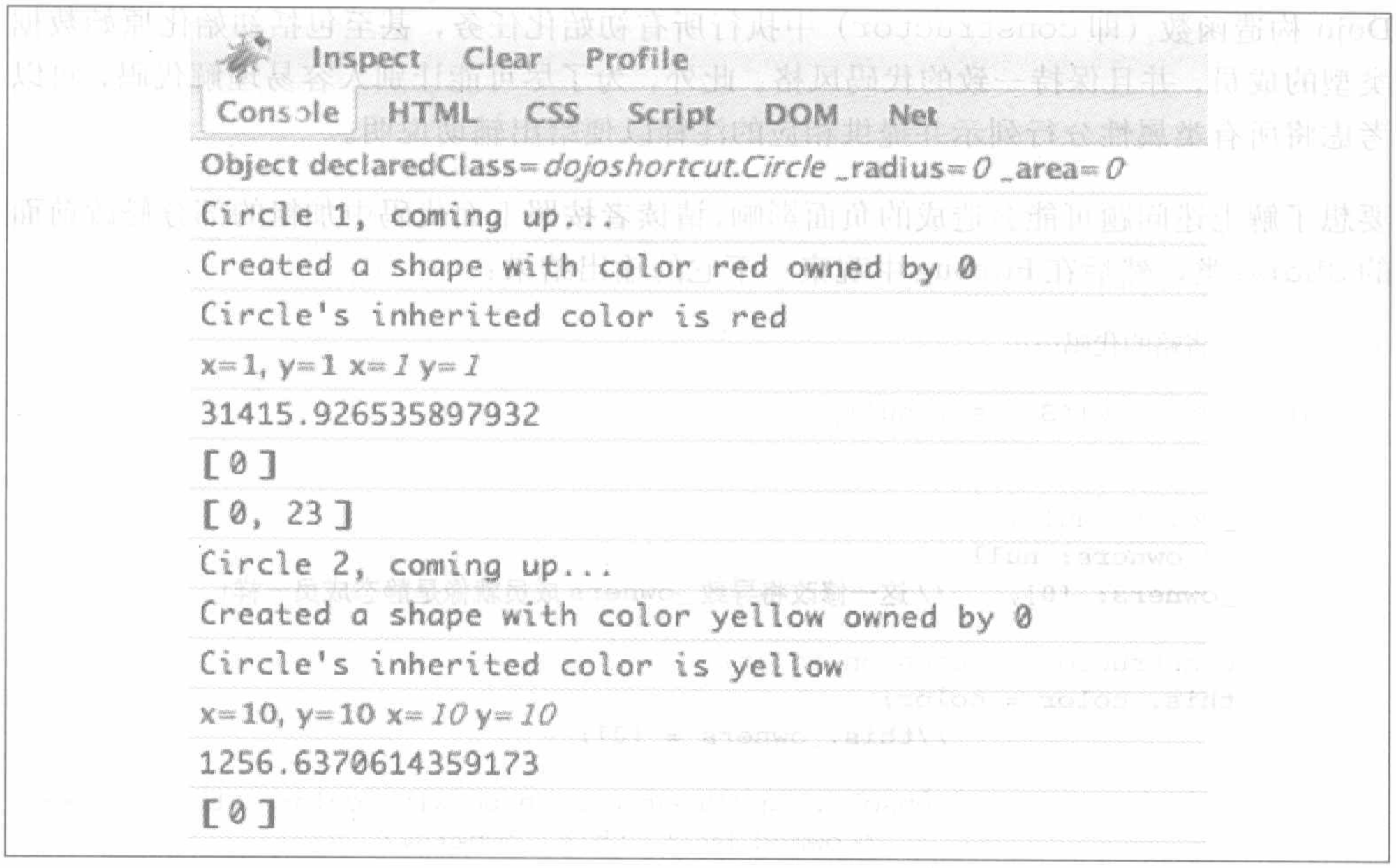



图 10-1：例 10-6 在 Firebug 中的输出结果

问题在于，`declare` 方法会在后台将所有属性都保存到 `Object` 的 `prototype` 属性中，而 `prototype` 是由所有实例共享的属性。对于数值或字符串等不变类型，修改这个属性只会导致局部变化。但是对于对象和数组等可变类型，修改这个属性则会导致变化的扩散。如例 10-7 中的代码片段所示，这个问题也可以得到适当解决。

例 10-7：所有实例都会共享 prototype 属性

```
function Foo() {}
Foo.prototype.bar = [100];

// 创建两个 Foo 的实例
foo1 = new Foo;
foo2 = new Foo;

console.log(foo1.bar); // [100]
console.log(foo2.bar); // [100]

// 这条语句修改 prototype 属性，而该属性由所有对象实例共享……
foo1.bar.push(200);

//……因此，当前的两个实例都会体现这一变化
console.log(foo1.bar); // [100,200]
console.log(foo2.bar); // [100,200]
```

在声明构造函数时，为了确保不会意外地初始化非原始数据类型成员，应该在标准的

Dojo 构造函数（即 constructor）中执行所有初始化任务，甚至包括初始化原始数据类型的成员，并且保持一致的代码风格。此外，为了尽可能让别人容易理解代码，可以考虑将所有类属性分行列示并提供相应的注释以便给出辅助说明。

要想了解上述问题可能会造成的负面影响，请读者按照下面代码中加粗的部分修改前面的 Shape 类，然后在 Firebug 中观察一下它的输出结果：

```
//……省略的代码……
dojo.declare("Shape", null,
 {
 _color: null,
 //_owners: null,
 _owners: [0],    //这一修改将导致 _owners 成员就像是静态成员一样!

 constructor : function(color) {
 this._color = color;
                 //this._owners = [0];

                 console.log("Created a shape with color ",this._colora
                     " owned by ", this._owners);
             },

             getColor : function() {return this._color;},
             addOwner : function(oid) {this._owners.push(oid);},
             getOwners : function() {return this._owners;}
         }

     );
//……省略的代码……
```

做完以上修改并在 Firefox 中刷新页面之后，读者应该看到图 10-2 中所示的 Firebug 控制台的输出结果。

### 调用继承的方法

在基于类的面向对象编程中，一种常见的模式就是在子类中重写超类的方法，然后在子类执行自定义的实现之前先调用继承自超类的这个方法。虽然不一定是这种情形，但先运行超类的基准实现，然后子类在该基准实现的基础上提供现有实现则是很正常的。通过dojo.declare创建的任何类都可以访问一个特殊的inherited方法，当调用该方法时将执行超类中对应的被重写的方法。（注意，constructor链是自动被调用的，无须使用 inherited。）

例 10-8 展示了如何调用继承的方法。

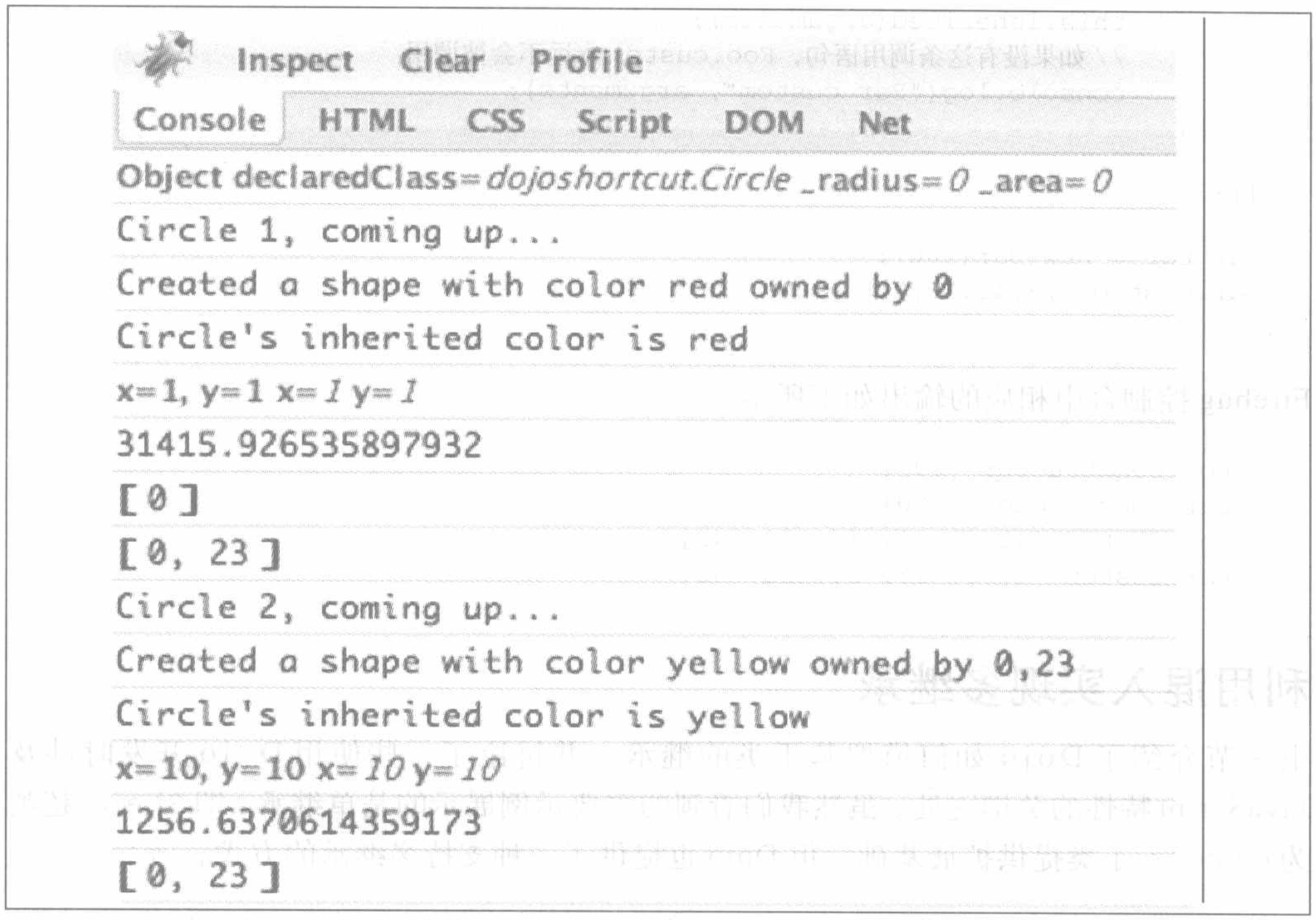



图 10-2：Firebug 控制台中的输出结果

例 10-8：在子类中调用重写的超类方法

```
dojo.addOnLoad(function() {
   dojo.declare("Foo", null, {

       constructor : function() {
           console.log("Foo constructor", arguments);
       },

       custom : function() {
           console.log("Foo custom", arguments);
       }
});

dojo.declare("Bar", Foo, {

       constructor : function() {
           console.log("Bar constructor", arguments);
       },

       custom : function() {
           // 自动调用 Foo 的 custom 方法并传入相同的参数
           // 如果有必要的话，当然也可以修改参数
```

```
            this.inherited(arguments);
            //如果没有这条调用语句，Foo.custom永远不会被调用
            console.log("Bar custom", arguments);
        }

    });

    var bar = new Bar(100);
    bar.custom(4,8,15,16,23,42);
});
```

Firebug控制台中相应的输出如下所示：

```
Foo constructor [100]
Bar constructor [100]
Foo custom [4, 8, 15, 16, 23, 42]
Bar custom [4, 8, 15, 16, 23, 42]
```

## 利用混入实现多继承

上一节介绍了Dojo如何模拟基于类的继承，并讨论了一些使用Dojo开发时涉及JavaScript特性的关键之处。虽然我们看到的主要示例展示的是单继承，即Shape超类为Circle子类提供扩展基础，但Dojo也提供了一种支持多继承的方式。

通过将Function对象联结起来而定义继承关系的过程称为原型连缀（prototype chaining），因为继承关系中的所有Function对象都是以JavaScript的Object.prototype属性为连接点而连缀起来的；在这个过程中，Object.prototype属性确定了Function对象之间的继承关系。（例10-2中通过手工编码方式定义Shape和Circle之间继承关系的过程就体现了这种思想。）

Dojo通过在原型连缀的思想之上模拟基于类的继承实现了对单继承关系的支持。但是，由于JavaScript限制每个Function对象只能有一个内置的prototype属性，因此实现多继承关系则稍微有点不同。

正如读者所知，要解决这个问题有许多方法。Dojo的方法是利用原型连缀在prototype中定义一个原型超类（prototypical ancestor），并以该超类作为原型连缀的基础，与此同时，允许向这个原型超类中注入其他的混入类（mixin）。换句话说，一个类只能有一个原型，但这个类所创建的Function对象则可以容纳任意多个构造函数的执行结果。当然，这些构造函数的原型在创建对象之后不会被记录在案，但我们却可以利用它们实现强大的功能。读者可以将这些混入类想像成“实际上执行某些操作的接口”或者“接口＋实现”。

**注意：** 在多继承关系当中，超类是被放在列表中提供给dojo.declare方法的。列表中第一个元素被称为原型超类，而随后的元素则通常被称为混入超类，或者更简洁的被叫做“混入类”。

以下是通过Dojo实现多继承关系的示例。其中与单继承唯一的不同之处，就是dojo.declare的第三个参数是一个Function对象列表，而不是一个Function对象。列表中的第一个元素是原型超类，其他元素是混入类：

```
<html>
    <head>
        <title>Fun with Multiple Inheritance!</title>

<script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
</script>

<script type="text/javascript">
    dojo.addOnLoad(function() {
        // 一个纯粹的Dojo类，带有适当的构造函数，而且没有直接超类
        dojo.declare("Tiger", null, {
            _name: null,
            _species: null,

            constructor : function(name)
            {
                this._name = name;
                this._species = "tiger";
                console.log("Created ",this._name,"the ",this._species);
            }
        });

// 另一个纯粹的Dojo类，带有适当的构造函数，而且没有直接超类
dojo.declare("Lion", null, {
    _name: null,
    _species: null,

    constructor: function(name)  {
        this._name = name;
        this._species = "lion";
        console.log("Created ",this._name," the ",this._species);
    }
});

// 一个有多个超类的Dojo类。其中，第一个超类是原型超类，而第二个
// （及其后续的Function对象）都是混入类。
// 特别要注意的是每个超类的构造函数会在这个子类的构造函数执行之前
// 先执行——没有任何办法改变它们的执行顺序。
dojo.declare("Liger", [Tiger, Lion], {
```

```
                _name: null,
                _species: null,

                constructor : function(name) {
                    this._name = name;
                    this._species = "liger";
                    console.log("Created ",this._name , " the ", this._species);
                }
            });

            lucy = new Liger("Lucy");
            console.log(lucy);
        });
        </script>
    </head>
    <body>
    </body>
</html>
```

在Firefox中运行上面示例后，读者会在Firebug控制台中看到如图10-3所示的输出结果，Tiger和Lion都是在Liger之前创建的。正如以前展示的Shape类的示例一样，在构建多继承关系时最终也会得到相应的子类，但必须首先创建必要的超类并执行超类的构造函数。

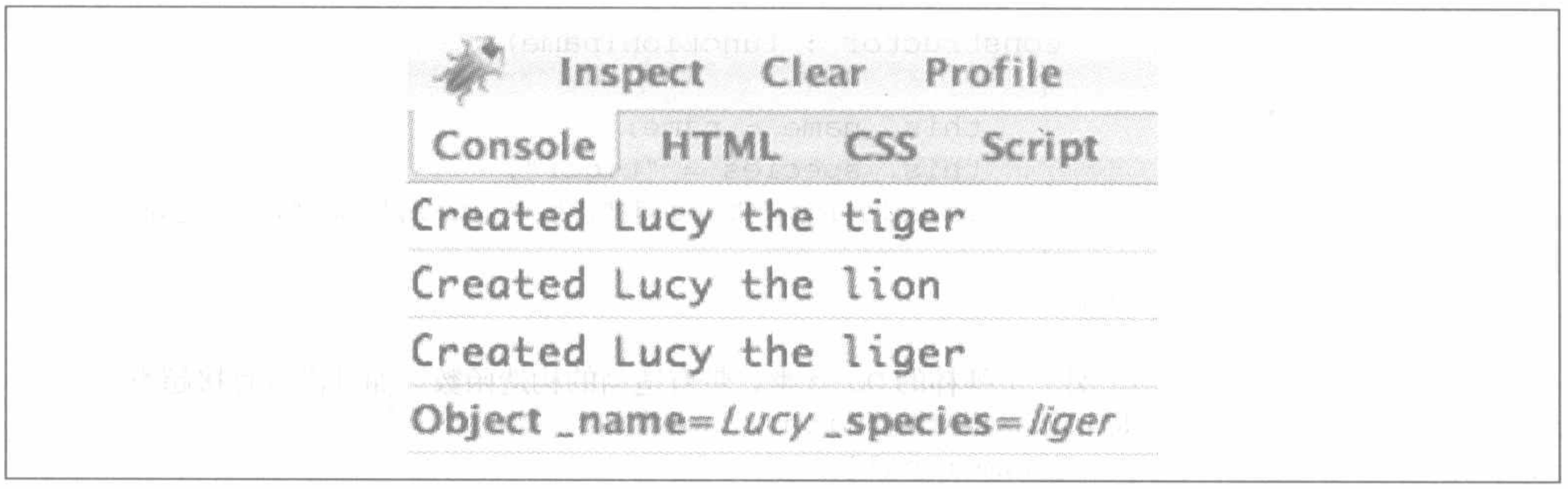



图10-3：虽然最后会得到Liger，但必须首先创建必要的超类并完成相应的初始化

## 多继承的问题

在以前展示的Shape类的示例中，我们无须关心由Circle传递给Shape的参数列表，因为Circle直接扩展了Shape。而且，将Shape的constructor的参数作为Circle的constructor的前位参数也很正常，甚至会很方便。在刚才看到的狮子（Lion）、老虎（Tiger）和狮虎（Liger）的示例中，所有构造函数都只有一个用于初始化动物名字的参数，因此都不存在真正的参数传递问题。

但是，假如Tiger和Lion分别有自己不同的constructor又会怎样呢？例如，Tiger的constructor可能会同时初始化老虎的名字和它的斑纹数目，而Lion的constructor则可能会同时初始化狮子的名字及其鬃毛长度。此时，应该怎样定义Liger的constructor呢？毕竟，传入Liger的constructor中的参数也会原样不动地传入Tiger和Lion的constructor中，而这根本没有任何意义。

在这种特殊的情况（即两个或多个超类分别需要自己特殊的参数）下，最好的办法就是将命名参数放到一个关联数组内，然后将这个关联数组传入子类的constructor中，即不再直接依赖arguments列表。Dojo1.1对在多继承关系中向超类传入自定义参数的支持并不好，但将来的版本有可能将我们上面讨论的解决方案纳入其中。

## 高级参数处理

Dojo1.0引入了一个新特性（源自dojo.delare），这个新特性是用来处理传入超类的constructor中的参数的一种高级手段。

简单地说，preamble会在constructor被调用之前运行，因此可以使用它来修改传入超类constructor中的参数。到Dojo1.1为止，preamble返回的任何参数都将传入所有超类的constructor函数中。尽管preamble也不能解决我们前面提到的Liger示例中的问题，但它在另外一种情况下却相当有用。

以下代码片段展示了preamble的工作原理：

```
dojo.declare("Foo", null, {
  preamble: function(){
    console.log("Foo preamble: ", arguments);
  },

  constructor: function(){
    console.log("Foo constructor: ", arguments);
  }
});

dojo.declare("Bar", null, {
  preamble: function(){
    console.log("Bar preamble: ", arguments);
  },
  constructor: function(){
    console.log("Bar constructor: ", arguments);
  }
});
```

— 待续 —

```
dojo.declare("Baz", [Foo, Bar], {
  preamble: function(){
    console.log("Baz preamble: ", arguments);
    return ["overridden", "baz", "arguments"];
  },

  constructor: function(){
    console.log("Baz constructor: ", arguments);
  }
});

var obj = new Baz("baz", "arguments", "go", "here");
```

以下是在初始化Baz时，Firebug控制台中显示的按时间先后顺序调用preamble和constructor的输出结果。注意，传入超类Foo和Bar的constructor中的参数都是由相应的preamble返回的，而Baz的constructor接收到的仍然是传入它自己的preamble中的参数：

```
Baz preamble:  ["baz", "arguments", "go", "here"]
Foo preamble: ["overridden", "baz", "arguments"]
Foo constructor: ["overridden", "baz", "arguments"]
Bar preamble: ["overridden", "baz", "arguments"]
Bar constructor: ["overridden", "baz", "arguments"]
Baz constructor: ["baz", "arguments", "go", "here"]
```

一般而言，设计多继承关系时的一条捷径就是让超类的constructor不接收任何参数。这种做法的好处在于，通过有目的地定义不带参数的超类，可以让子类自由接收和处理任意自定义参数，同时还能确保位于继承链上端的超类不会受这些参数的影响。因为超类不接收参数，所以也就不会受影响。

# 小结

在学习完本章之后，读者应该：

- 理解如何在Dojo中使用dojo.declare来模拟类。
- 能够在Dojo中实现单继承和多继承关系。
- 知道在Dojo类的constructor外部初始化JavaScript对象的相关风险。
- 了解Function对象是用来模拟JavaScript类的后台机制；记住，JavaScript中不存在“真正的”类。
- 理解基于原型继承和基于类继承的一些差异。

- 对 Dojo 如何利用 JavaScript 基于原型的继承在后台模拟基于类的继承有大致的了解。

在接下来的第二部分中，我们将介绍 Dijit 和 Util。

# 第二部分

# Dijit与Util

本书第一部分介绍了Base和Core，作为一个JavaScript库来说，它们能够简化任何Web开发任务。在第二部分中，我们将分别介绍Dojo工具箱中的可视化元素*Dijit*（各种令人赞叹的部件（widget））和*Util*（包含构建工具及独立的单元测试框架）。

Dijit由各种即装即用的部件组成，为了在Web上实现丰富的用户体验提供了从表单元素到布局容器，乃至其他常用控件的完备集合。Dijit直接构建于Base和Core的基础之上，因而其本身就是通过强大的标准库来隔离浏览器不一致性，并且减少编程、调试、测试以及维护过程中产生冗余代码的一个极好范例。从这个意义上来讲，Dijit应该算是Base和Core非常自然而然的应用。

本书第一部分向读者展示了Dojo工具箱基本的构建块（building block），其中涉及的内容关系到读者能否成为一名更好的JavaScript开发人员。本书的这一部分则把重点放在介绍各种Dojo部件上面，只要把这些现成的部件放到页面中，它们就可以“立即运行”，而整个过程仅需少量编码，甚至无须编码。鉴于本部分全面翔实地介绍了所有Dojo部件的用途，以及在HTML标记中创建部件的过程，因此Web设计人员可能会非常关注这一部分。

当然，编写自定义的Dojo部件也是一个极为重要的主题，无论是零起点，还是在现有部件基础之上创建。因此，本部分也将深入介绍如何自定义Dojo部件。另外，对部件剖析与生命周期的探讨、对在标记中声明部件和通过编程创建部件这两种方式的解释、对易访问性（a11y是对易访问性的英文accessibility的简写，因为这个单词以字母a、y开头和结尾，而这两个字母之间又包含11个字母）的阐述也是本部分的主要内容。

第二部分在全面介绍Dijit之后，将以对Util的讨论结束。Util中提供了很多实用的构建工具，比如ShrinkSafe，它是基于久经考验的Rhino JavaScript引擎的JavaScript代码压缩系统；还有DOH（Dojo Objective Harness，Dojo目标套件），它是一款独立的单元测试框架，用于辅助应用程序测试和质量保证。

# 第11章

# Dijit概述

Dijit是Dojo工具箱为替换标准HTML Web控件提供的一组令人赞叹的部件。本章作为第二部分的开篇，主要从非技术角度来揭示Dijit作为一套现成资源产生的内在动机、基本原理以及主要目标。这几方面的内容又恰好与设计人员及原始页面作者不用编程即可在HTML标记中使用Dojo部件的讨论相辅相成。此外，本章还对Dijit中包含的全部内容进行了一番提纲契领的展示。

## Dijit产生的动机

Web开发在很大程度上属于工程技术的范畴，在它引人瞩目的发展历程中，各种聪明的技巧和美妙的创意层出不穷。虽然从概念上来讲，浏览器是Web应用程序最终的运行平台，但从一名工程师的角度来看，无论何时要创建丰富的用户体验，都不得不针对浏览器进行个别设计或对它们进行某种形式的增强。目前，由于产业主导者对顺应标准的漠视，保守地说也至少形成了浏览器“五足鼎立”的格局。而且，随着支持Web浏览的移动设备相继问世，这个数字还会继续增加。

因此，开发易于维护的Web应用程序只能比以往变得更加困难；但是，如果不考虑支持尽可能多的Web运行平台，那么就意味着丢掉市场份额、失去知名度和减少盈利。而通过以特定方式混合HTML、CSS及JavaScript来支持多平台的做法虽然常见，但效率却极为低下，最终导致跨浏览器兼容似乎成了可望而不可及的理想。

我们知道，Base和Core可以把浏览器的不一致性隔离开来，并且能把编写专有代码的工作量降至最低；Dijit充分利用了Base和Core提供的这些优秀特性，为构建模块化、易重用的用户界面部件搭建起来易扩展的框架。

尽管从技术上看不正确，但许多Dojo用户都把“Dijit”看成“Dojo”的代名词，因为

其中的部件非常受欢迎。不过，我们在此需要重申，Dijit 本身只是 Dojo 工具箱的一个子项目，由于它在逻辑上与工具箱的其他部分相对独立，因此无论管理还是将来改进都非常方便。除了为开发人员提供现成的部件之外，Dijit也为他们构建自定义的部件提供了基础架构，而且是与内置部件相同的基础架构。

Dijit 的主要目标如下：

- 像 Swing 为 Java 应用程序提供接口，或者像 Cocoa 为 OS X 应用程序提供界面控件一样，为 Web 开发提供标准的常用部件。
- 利用 Core 和 Base 中已有的机制，保持部件的实现尽可能简单且容易移植。
- 符合易访问性（a11y）标准，按照 ARIA（Accessibility for Rich Internet Applications，富因特网应用程序的易访问性）规范要求支持视力残障或行动不便的用户。
- 要求所有部件全球化，通过确保部件本地化及支持文化格式（cultural format）和双向（bidirectional，bidi）内容来简化国际化倡议。
- 维护具有相关性的 API，以便开发人员使用不同的部件并共享解决问题的模式。
- 通过样式表实现一致的部件外观，也便于自定义部件。
- 确保在标记中创建部件与通过 JavaScript 创建部件一样简单（甚至更容易）。
- 无论是使用单个部件增强现有页面，还是将多个部件嵌入成熟的应用程序，都应该简单易行。
- 对双向文本提供完整支持（自 Dojo1.1 开始）。
- 跨平台支持最常用的浏览器，包括 Internet Explorer 6+、Firefox 2+ 和 Safari 3+（注 1）。

## 低耦合，高内聚

或许，Dijit带给Web开发人员最重要的好处，就是能把用户界面组件封装到一个单独的部件中。假如读者曾经从事过某些Web开发工作，那很可能产生过这种需求：即想要把

---

注 1：Dijit并没有正式支持与Base和Core支持得一样多的浏览器。这一决定背后的现实考虑在于，对像Opera或Konqueror这样的浏览器，还没有足够的用户群值得额外的编码、测试和维护。但是，虽然Dijit没有“正式”支持这些浏览器，并不意味着在这些平台中使用 Dijit 一定不可行，特别是对 Konqueror、Firefox 和 WebKit（Safari 的核心）这些开源项目更是如此。

用户界面中的HTML、CSS和JavaScript源文件封装到一个易于移植的包中，这个包本身要具有实例化的能力，并且能在恰当的时候以最少的干预提供恰当的功能。

**注意：** 在编程领域中，开发模块化的用户界面组件的问题可以归结为两个目标：内聚性和耦合性。其中，内聚性用来衡量源代码及相关资源共同提供某种功能时的独立程度，而耦合性则用来表示一个模块对另一个模块的依赖程度。在设计部件时，我们的目标应该是高内聚性和低耦合性。

Dijit对这种需求给出了极好的回应，而且更值得称道的是，它本身作为一个基础架构层，无须开发人员再行编码、调试和维护。如图11-1所示，以Base的`dojo.declare`方法为核心来模拟类的Dijit，不仅提供了用于创建和销毁部件的标准生命周期方法，提供了响应键盘和鼠标事件的标准方式，而且还具备封装部件视觉表现的能力。此外，由于部件可以通过HTML标记和JavaScript来组织管理，因而就为设计和开发人员分别提供了效果相同的控制手段。

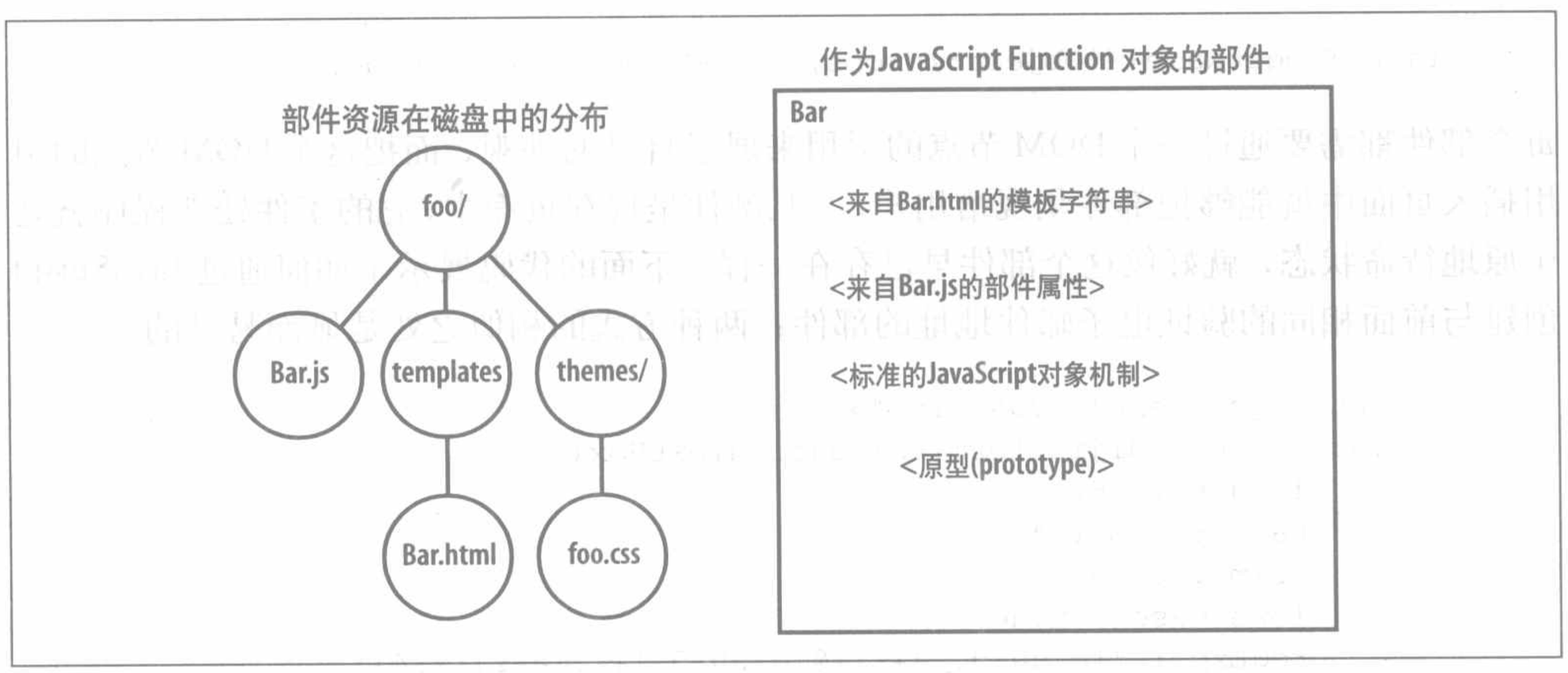



图11-1：一个Dojo部件的资源在磁盘中的物理分布及该部件作为JavaScript Funtion对象的示意图

作为一名设计人员，要在HTML标记中插入一个部件，只需在相应的标记中添加一个解析器能够识别的`dojoType`属性，即可将该标记实例化为一个事件驱动的DHTML控件。例如，下面这个取自第1章示例的代码片段，展示了如何在表单中插入一个自定义的文本框部件，并让该部件能够近似地验证在其中输入的电子邮件地址，完全在标记中实现：

```
<input type="text"
    length=25
    name="email"
```

```
    dojoType="dijit.form.ValidationTextBox"
    trim="true"
    lowercase="true"
    regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"
    required="true"
    invalidMessage="Please enter a valid e-mail address"/>
```

仅此而已，使用这个部件不必编写一行JavaScript代码。没错，开发人员在开发或者扩展部件时需要编写JavaScript代码，但问题的关键在于，一旦编码完成，新部件也会成为一种通用的部件，可以随时取用。当页面加载后，解析器会自动查找标记中的dojoType属性，并在必要时在后台向服务器请求所需的资源，最终将一个DHTML部件展现在页面中。对于设计人员来说，布局用户界面组件就应该如此简单！

当然，通过标记插入的部件也能够通过JavaScript插入。开发人员可以像创建任何JavaScript对象一样创建一个完全相同的部件对象，然后再轻而易举地把它插入页面中。

作为一种通用的模式，Dojo部件的构造函数都具有下列签名形式，并且接收一个配置属性的集合和一个节点引用作为参数：

```
dijit.WidgetName(/*Object*/props, /*DOMNode|String*/node)
```

每个部件都需要通过一个DOM节点的引用来展示自己的外观，而把这个DOM节点的引用插入页面中就能够把部件呈现给用户。一旦部件呈现在页面中，它的事件处理程序就处于原地待命状态，就好像这个部件早已存在一样。下面的代码展示了如何通过JavaScript创建与前面相同的验证电子邮件地址的部件；两种方式的相似之处是显而易见的：

```
<script type="text/javascript">
    var w = new dijit.form.ValidationTextBox({
        length : 25,
        name : "email",
        trim : true,
        lowercase : true,
        regExp : "[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}",
        required : true,
        invalidMessage : "Please enter a valid e-mail address"
    }, n); // n是对页面中某个节点的引用
</script>
```

# 易访问性（a11y）

信息时代，易访问性是越来越受到人们关注的一个重要话题。除了尽可能向最广泛的受众（包括残疾人）交付内容这个一般性的目标之外，508条款（注2）及其他法律条文都

注2：508条款是指1973年制定的美国康复法案中的相关条款，该条款要求联邦机构对美国残疾人提供合理的便利条件。

以公权形式对技术上应该如何确保残疾人无障碍地使用信息制定了最低的标准，而且这里面还有经济上的动机：据美国劳工部估算，残疾人士能够自主决定的开销每年可达1 750亿美元（参见*http://www.usdoj.gov/crt/ada/busstat.htm*）。无论读者对这个问题怎么看，也不管动机何在，a11y 都已经成为一个不可忽视的问题。

## 常见的 a11y 问题

虽然短短一节内容不可能讨论成功实现Web应用程序的方方面面，但我们希望本节能够引起读者对相关问题的重视，特别是Dijit在解决这些问题时的处理方式。实现易访问性的两个主要目标是：支持需要使用屏幕阅读器的视障用户和支持只能通过键盘来导航应用程序的用户。

在默认情况下，Dijit能够同时支持上述两类用户。其中，对于视障用户，为了保证他们无障碍地使用应用程序，首先会检查用户的浏览器是否运行于高对比度模式，其次针对Internet Explorer和Firefox（注3），还要检查是否屏蔽了图像显示功能。无论哪项需要易访问性支持（accessibility-enabling）的检查得到确认，都将视情形基于增强后的样式、图像或模板来呈现部件。

例如，图11-2展示了`dijit.ProgressBar`部件在高对比度及标准模式下的呈现效果。

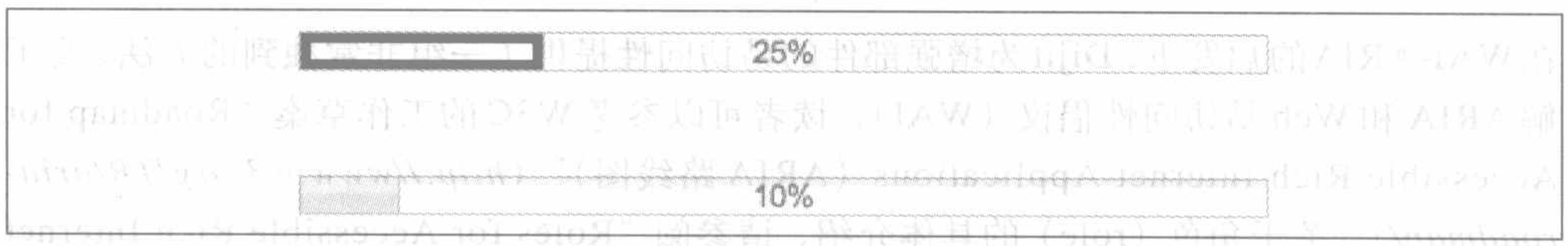



图11-2：上：在满足易访问性条件时，自动呈现的高对比度的dijit.ProgressBar；下：标准的dijit.ProgressBar呈现效果

虽然实现起来可能较为麻烦，但要为盲人用户提供易访问的部件，则最好遵循下面这条实践中得来的经验法则：页面的功能不能依赖于图像（包括CSS背景图像和标准的`IMG`标签图像），即页面中的图像即使因故无法显示，也不会影响页面功能。进而，必然的连带条件就是要保证提供`alt`属性；虽然添加`alt`属性绝对简单，而且通常也不需要太多，但做到这一点往往正是达成易访问性的关键。

Dijit内置的部件通过使用标准的`tabIndex`属性来控制应用程序中的焦点移动，也提供

注3：虽然在Internet Explorer的Windows版本中检查高对比度模式很顺利，但在其Mac版本及其他浏览器中则存在一些问题。因为并非所有平台或浏览器都对a11y提供了口径相同的支持，所以这个问题要视浏览器而定。

了对全键盘操作的支持。另外，还提供了明确管理复杂控件中焦点移动的机制，以便在必要时显示提示条，而这对于使用屏幕阅读器的用户而言无疑是最重要的。

## WAI-ARIA

虽然保证视障和肢残用户无障碍获取网上信息的倡议正得到越来越多的人的响应，但对于现代的RIA（Rich Internet Application，富因特网应用程序）而言，还需要更多的支持才行。所谓更多的支持，包括不会明显引起页面重载的XHR调用状态发生变化时必须让用户知道，以及恰当地处理好与选择操作对应的后退按钮功能。

WAI-ARIA（W3C Web Accessibility Initiative for Accessible Rich Internet Applications，W3C对RIA易访问性的倡议）就是针对模仿桌面功能的Ajax应用程序提出来的，其中包括向残疾用户交付内容和功能的一组指导方针。早在20世纪90年代，屏幕阅读器能够顺利地读出格式良好的HTML文档。时至今日，Web应用程序中的各种部件大都由层层嵌套的DIV元素组成，并且对它们的操作往往也要通过Ajax技术来实现，而这些新事物对屏幕阅读器来说却毫无意义。在这种背景下，WAI-ARIA为向盲人用户有效传达信息提供了必要的语义。例如，根据这些语义，屏幕阅读器会知道某个嵌套的DIV集合是树形部件，而部件中的某个节点当前正拥有焦点，因此按下“Tab”键就能把焦点切换到“下一个”元素上。

在WAI-ARIA的启发下，Dijit为增强部件的易访问性提供了一组非常独到的方法。要了解ARIA和Web易访问性倡议（WAI），读者可以参考W3C的工作草案“Roadmap for Accessible Rich Internet Applications（ARIA路线图）”（*http://www.w3.org/TR/aria-roadmap/*）。关于角色（role）的具体介绍，请参阅“Roles for Accessible Rich Internet Applications(ARIA中的角色)”（*http://www.w3.org/TR/aria-role/*），而关于状态的内容则可以参阅“States and Properties Module for Accessible Rich Internet Applications（ARIA的状态和属性模块）”（*http://www.w3.org/TR/aria-state/*）。

表11-1列出了与WAI相关的方法。

表11-1：与WAI相关的方法

| 方法 | 说明 |
| --- | --- |
| `onload()` | 自动调用以检查页面是否处于高对比度模式或者屏蔽了图像。通常不必直接调用这个方法，因为页面在加载时会自动调用它 |
| `hasWaiRole(/* DOMNode */ node)` | 如果node节点中包含role属性那么返回true |
| `getWaiRole(/* DOMNode */ node)` | 返回node节点中的role属性 |

表 11-1：与 WAI 相关的方法（续）

| 方法 | 说明 |
| --- | --- |
| setWaiRole(/* DOMNode */ node, /* String */ role) | 设置 node 节点中的 role 属性 |
| removeWaiRole(/* DOMNode */ node) | 删除 node 节点中的 role 属性 |
| hasWaiState(/* DOMNode */ node, /* String */ state) | 如果 node 节点中包含一个特定 state 属性那么返回 true |
| getWaiState(/* DOMNode */ node, /* String */ state) | 返回 node 节点的 state 属性 |
| setWaiState(/* DOMNode */ node, /* String */ state, * String */ value) | 设置 node 节点的 state 属性 |
| removeWaiState(/* DOMNode */ node, /* String */ state) | 删除 node 节点的 state 属性 |

根据 WAI-ARIA 的规定，role 描述的是一个控件的角色，它的可能取值包括 link、checkbox、toolbar 或者 slider。state 描述的是一个控件的状态，而且状态值不一定是非此即彼。例如，role 值为 checkbox 的控件有一个“checked”状态，而当该控件只是部分被选中时其值为 mixed。此外，checked 和 disabled（都是二元值，即非 true 即 false）也是可能的 state 值。

# 设计人员需要了解的 Dijit

要在网页标记中使用已有的 Dojo 部件非常简单：首先，通过标记的 dojoType 属性指定要在页面中插入的部件类型；然后通过标记的属性传递用于创建部件的数据。最后，通过扩展点（extension point）来重写已有的部件行为。其中，dojoType 属性是必需的，其他传递数据的属性通常只需设置为合理的默认值即可，而扩展点则是为更好地满足必要的合理实现准备的。

---

**注意：**“方法”和“扩展点”之间的不同主要体现在语义上面：方法是开发人员为扩展部件而直接调用的操作；扩展点同样也是方法，但不是由开发人员直接调用，而是在适当的条件具备时由部件自动调用。例如，某个部件可能拥有一个名为 setValue 的方法，开发人员可以调用这个方法来设置部件的值。而这个部件的 onKeyUp 方法，由于用户每次按键时都会自动被调用，因此就称其为扩展点。

---

在表11-2中列出了一些在创建部件需要设置的属性，这些属性将被直接应用到部件的DOM节点上，熟悉这些属性对于有效地使用内置部件格外重要。通过合理地设置这些属性，可以确保Dojo部件底层的HTML标记达到可定制同时又“恰如其分”的状态。

## 关于DOCTYPE验证

从技术角度讲，能够做到在页面中包含Dojo部件并使该页面满足HTML 4.01严格型DOCTYPE验证的条件。但前提是必须以编程方式创建Dojo部件，而且部件的模板必须符合规范。如果是在标记中创建部件，那么dojoType及其他非标准属性都会违反验证程序中的有效性规则。而且，即使某种程度上可以编写自定义的DTD（Document Type Definition，文档类型定义），并在其中定义非标准的dojoType等属性，有些验证程序仍然会报错。为此，本书示例的HTML页面中都没有包含首行的DOCTYPE标签。

标准虽然很重要，但是，估算出在某些情况下不遵循标准的代价也同等重要。例如，在Dojo这个健壮、开源、社区支持的，且在透明性方面丝毫不亚于其他项目的研究成果中使用非标准的属性，绝对不能和粗心大意地在网页中使用非标准的属性混为一谈。Dojo代表着最前沿的思想，在经过世界上最伟大的一些DHTML黑客深思熟虑之后才做出的这个决定，显然会在表面的非标准之下蕴藏着更为重要的收益。

也正因为如此，Dojo从未宣称自己是一个完全符合标准的项目，或者始终会遵循任何规范的约束（HTML 4.01严格型DOCTYPE就是一个规范）。然而，Dojo却能保证在某些浏览器范围内实现特定的功能，这又是另一种约束（当然，可想而知）。

表11-2：Dijit部件的常用属性

| 属性 | 类型 | 说明 |
| --- | --- | --- |
| id | 字符串 | 部件的唯一标识符。在默认情况下，这个值会自动生成且能够确保唯一性。如果用户明确地提供了一个不具有唯一性的值，那么该值将被忽略，并自动生成一个唯一值 |
| lang | 字符串 | 用于显示部件的语言。默认使用浏览器的地区设置。如果要设置另外的地区，那么需通过djConfig.extraLocale指定相应的语言包。（一般来讲，除非必须在一个页面中显示多种语言，否则不会用到这个属性。） |
| dir | 字符串 | 按照HTML的DIR属性为页面提供双向显示支持。在默认情况下，这个属性设置为ltr（从左到右），在少数情况下需要设置为rtl（从右到左）。除了这两个值之外，这个属性没有其他有效值 |

表 11-2：Dijit 部件的常用属性（续）

| 属性 | 类型 | 说明 |
| --- | --- | --- |
| `style` | 字符串 | 指定传递给部件最外部 DOM 节点的 HTML 样式属性。在默认情况下，不传递额外的样式属性 |
| `title` | 字符串 | 标准的 HTML `title` 属性，用于在鼠标悬停在 Diji 部件的 DOM 节点上时向用户显示提示条 |
| `class` | 字符串 | 应用到最外部 DOM 节点的 CSS 类信息。这个属性在全部或部分覆盖默认主题时特别有用 |

## 主题

Dijit 主题就是针对一整套部件设计的一组完整的 CSS 规则。换句话说，可以把 Dojo 部件想像成具有换肤能力，而主题就是可以更换的一种皮肤。如果读者只想利用并基于现有部件开发，那么主题就显得格外重要，因为就无须再考虑如何编写 CSS 规则了。到 Dojo1.1 为止，工具箱中内置了 3 款非常漂亮的主题：

*Tundra（极地冻原）*

受北极地貌启发而命名的以亮灰色为主色调并搭配亮蓝色的主题。

*Soria（索里亚）*

在浅蓝色调上突出亮蓝色，总之是明亮的蓝色调组合起来的主题。这款主题中的控件具有明亮的“Web 2.0 辉光”。它的灵感来自于在西班牙旅游城市索里亚拍摄的一组风景片中美丽的蓝色天空。

*Nihilo（有无相生）*

以白色为主色调，搭配柔软的灰色轮廓线和暗蓝色文本的主题。这款主题中的某些控件使用了暗黄色调。据说，这款主题的灵感来自于“有生于无”（ex nihilo）的哲学观（即以无形造有形），由于它高度简洁，读者如果不留心也许根本注意不到它。

在 Dojo 工具箱的本地目录中，可以找到一个主题测试程序，路径为 *dijit/themes/themeTester.html*。在浏览器中打开这个页面，选择不同的主题即可看到应用该主题后各种部件的实际外观。俗话说百闻不如一见，计算机屏幕上显示的这些主题的实际效果通过黑白印刷的纸面无论如何也是无法表现的。

主题目录具有以下结构，其中每个主 CSS 文件基本上都使用 `@import` 语句导入了其他 CSS 文件，以便保证设计方案的容易维护（由于构建工具可以合并 CSS 文件，因此正式的构建中只包含着合并后的最终文件，旨在减少取得资源过程中的 HTTP 等待时间）：

```
themes/
    tundra/
        tundra.css
        images/
            大量静态的图像文件
    soria/
        soria.css
        images/
            大量静态的图像文件
    nihilo/
        nihilo.css
            images/
                大量静态的图像文件
    <按照上面的模式，可以在此继承创建自定义的主题……>
```

例11-1着重强调了页面中与主题相关的代码。

例11-1：使用主题

```
<html>
    <head>
        <title>Fun With the Themes!</title>

        <!-- 加载tundra主题 -->
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        ></script>

        <script type="text/javascript">
            //require your dijits here
        </script>
    <head>

    <body class="tundra">
        <!-- 在此插入Dojo部件 -->
    </body>
</html>
```

我们注意到，使用某个主题只需加载相应的样式表并为BODY标签添加适当的类即可。而且，如果不想将一个主题应用到整个页面的话，也可以把类名添加给页面中的其他标签。主题可以应用给页面中的任意元素，不一定要应用给整个页面，例如，应用给某个DIV元素或者某个部件等。另外，上面的代码也同时加载了*dojo.css*文件，这是因为该文件中包含着一些基准样式。

把当前主题更换成其他主题也非常简单，只要把所有的 *tundra* 替换成 *soria* 或 *nihilo* 就可以了。就这么简单。而且，这和我们通常想像的为页面换肤一样方便。

对于主题，我们就不再深入介绍了。毕竟，一个主题无非就是一套设计好的CSS规则而已，因此只要我们知道用户对Dojo部件的外观总会有不同的要求就可以了。不过，假如读者确实对主题很感兴趣，那么一定要先找一本不错的CSS参考手册，然后再开始通读主题中的所有 CSS 文件。在通读 CSS 样式规则时，读者可能会看到类似 `.tundra .dojoButton { /* 样式声明 */ }`的规则，通过 Firebug 应该很容易在 Dijit 模板文件或页面中找到与规则中的选择符对应的元素。

## 节点与 Dojo 部件，DOM 事件与 Dojo 部件方法

Dojo 部件与 DOM 节点之间存在重要差别：Dojo 部件是一个 JavaScript 的 Function 对象，该对象基于一组资源实例化而成，这些资源包括 HTML 标记、CSS、JavaScript 和其他静态资源（如图像）。通过指定 Dojo 部件的 domNode 属性（其模板最外部的节点）可以将其视觉表现插入到页面中。

另外，通过比较 `dojo.byId` 和 `dijit.byId` 方法可以进一步理解 DOM 节点与 Dojo 部件的区别。首先，`dojo.byId`根据一个字符串值返回相应的DOM节点，而Dijit专有的 `dijit.byId`方法则返回一个与特定DOM节点关联的部件。这两个方法的具体区别如表 11-3 所示。基于下面的 Button 部件在 Firebug 中执行这两个方法可以看出它们的区别：

```
<button id="foo" dojoType="dijit.form.Button">click me</button>
```

表 11-3：dojo.byId 和 dijit.byId 之间的区别

| 方法 | Firebug 控制台显示的结果 |
| --- | --- |
| dojo.byId("foo") | <button<br>id="foo"<br>class="dijitStretch<br>dijitButtonNode<br>dijitButtonContents"<br>waistate="labelledby-foo_label"<br>wairole="button"<br>type="button"<br>dojoattachpoint="focusNode,titleNode"<br>role="wairole:button"<br>labelledby="foo_label" |

表 11-3：dojo.byId 和 dijit.byId 之间的区别（续）

| 方法 | Firebug 控制台显示的结果 |
| --- | --- |
| | tabindex="0" |
| | valuenow="" |
| | disabled="false"> |
| dijit.byId("foo") | [Widget dijit.form.Button, foo] _connects=[4] _attaches=[0] |
| | id=foo |

可见，dojo.byId返回的是为dijit.form.Button实例提供视觉表现的DOM节点，而dijit.byId返回的则是一个JavaScript Function对象，它反映的是标准Dojo部件的工作机制。

**警告：** 一个极为常见的错误是基于 dojo.byId 返回的结果调用部件的方法。别忘了 dojo.byId 返回的 DOM 节点中不包含任何与 Dojo 部件相关的方法。

Dojo部件与DOM节点之间存在的这一区别，必然导致了Dijit事件与DOM事件的不同。例如，虽然 Dojo 部件的 onClick 事件与 DOM 节点的 onclick 事件的命名方式极其相似，但这两个事件却完全不一样。通过加载并运行下面的页面示例，并单击页面中的按钮部件，读者可以在 Firebug 控制台的输出结果中看出差别所在：

```
<html>
   <head>
        <title>Fun with Button Clicking!</title>

        <link rel="stylesheet" type="text/css"
  href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
<link rel="stylesheet" type="text/css"
  href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

<script
    djConfig="parseOnLoad:true"
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
></script>

<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.form.Button");
    dojo.addOnLoad(function() {
        dojo.connect(dojo.byId("foo"), "onclick", function(evt) {
            console.log("connect fired for DOM Node onclick");
        });
```

```
                dojo.connect(dijit.byId("foo"), "onclick", function(evt) {
                    console.log("connect fired for dijit onclick");    //绝不要这样做!
                });

                dojo.connect(dijit.byId("foo"), "onClick", function(evt) {
                    console.log("connect fired for dijit onClick");
                });
            });
        </script>
    <head>
    <body class="tundra">
        <button id="foo" dojoType="dijit.form.Button" onclick="foo">click me
            <script type="dojo/method" event="onClick" args="evt">
                console.log("Button fired onClick");
            </script>
        </button>
    </body>
</html>
```

这个示例在页面的标记中为一个简单的Button部件定义了一个简单的方法，即为部件的onClick方法提供了简单的实现，同时也定义了3个连接（connection）：与DOM节点的onclick事件的连接、与Dojo部件的onclick和onClick事件的连接。但是，Dojo部件根本就没有onclick事件。因此，这个示例也提醒我们，如果连接部件的单击事件时使用了onclick，那么它很可能会成为一个难以查找和修复的bug。

# 解析器

作为Core所提供的一种资源，Dojo解析器是初始化在标记中定义的部件，并确保与domNode关联的部件的视觉表现被插入页面中的标准手段。实际上，在把domNode插入到页面中之后，浏览器会负责呈现页面中的部件。因此，虽然部件的DOM节点对于Dijit的视觉表现而言至关重要，但Dojo部件的整体运行仍然需要多方面支持。本节主要介绍解析器，同时详细介绍解析器的工作原理。

## 页面加载时解析部件

到目前为止，除了在第1章介绍过一些基本内容，在第7章的拖放示例中进行了某些探讨之外，应该说还没有正式地介绍解析器。事实上，解析器最常见的用途就是在页面中实例化部件。闲话少说，下面我们就来看一个解析器实例化标记中部件的示例。请读者注意例11-2中的粗体代码，那些代码是解析器正常工作所必需的。

例11-2：自动解析部件

```
<html>
    <head>
        <title>Fun With the Parser!</title>
```

```
    <!-- 导入为内置部件应用样式的标准CSS文件 -->
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        djConfig="parseOnLoad:true"
    ></script>

    <script type="text/javascript">
        dojo.require("dojo.parser");
        dojo.require("dijit.form.Button");
    </script>
  <head>

  <body class="tundra">
    <button dojoType="dijit.form.Button" >Sign Up!</button>
  </body>
</html>
```

这个示例展示了一个简单的页面中包含着一个现成的Dojo部件，该部件除了漂亮的外观之外不具备任何功能。不过，对于说明解析器的工作原理而不深入掌握Dojo部件的细节而言，这样也足够了。此时此刻，读者所要知道的就是必须通过dojo.require语句取得Button部件，然后再通过dojoType属性将其插入页面中。

---

**注意：** 在addOnLoad块中添加的任何逻辑，都会在部件被解析之后执行，从而确保该块中的代码可以引用这些部件。

---

在例11-2中，我们看不到直接调用解析器的任何语句；而这正是刻意设计的。在大多数情况下，都只需使用dojo.require将部件请求到页面中，再设置djConfig中的parseOnLoad标志，然后其他操作就在后台自动进行了。应该说，以上就是运行这个示例的整个过程。值得回味的是，从获取一个现成的部件到把这个部件插入到页面中，不过是敲击几下键盘而已。不用考虑多余的问题，也没有反复的权衡比较，更没有丝毫的踌躇和迷惑，这一切绝对堪称优雅至极。

## 手动解析部件

当然，手动解析页面或DOM中某些节点的情形也很常见。好在，手动解析也同样简单到只需调用一个方法即可。例11-3展示了如何手动解析页面中的Dojo部件。

例 11-3：手动解析页面

```
<html>
    <head>
        <title>Hello Parser</title>

         <!-- 导入为内置部件应用样式的标准 CSS 文件 -->

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:false"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.Button");
            dojo.addOnLoad(function() {
              dojo.parser.parse(); // 在页面加载后手动解析部件
            });
        </script>
    <head/>
    <body class="tundra" >
      <button dojoType="dijit.form.Button" >Sign Up!</button>
    </body>
</html>
```

除了手动解析整个页面之外，在更多情况下还需要手动解析某个 DOM 节点。此时，解析器需要接收一个DOM树中根节点的引用作为参数，然后扫描其中的dojoType属性并完成实例化。也就是说，必须为 parse 方法提供实际要解析节点的父节点。以下是基于前面的示例，手动解析 DOM 节点的代码：

```
<script type="text/javascript">
  dojo.require("dojo.parser");
  dojo.require("dijit.form.Button");
  dojo.addOnLoad(function() {
    // 解析器会遍历传入的 DOM 树并完成部件的实例化。
    // 此例中，按钮部件只是 DOM 树的一个分支节点，因此会被解析
   dojo.parser.parse(document.getElementsByTagName("button")[0].parentNode);
  });
</script>
```

---

**警告：** 在手动解析页面中的部件时，如果传递给parse方法的是部件的DOM节点，会导致解析失败并接收到解析失败的视觉提示。好在，找到一个节点的引用之后，再引用其父节点，也只不过多敲入 parentNode 这几个字母而已。

---

## 解析器揭秘

虽然解析器在后台完成的工作似乎很神秘，但归根结底仍然是一种设计精准的自动化处理过程。如前所述，解析器的用法主要有两种：一是通过djConfig="parseOnLoad:true"设置加载后解析页面，二是手动解析某个部件。本节就来详细介绍这两种解析器工作方式的细节。

在页面加载后解析部件需要具备下面3个基本条件：

- 以键/值对形式在djConfig配置项中添加parseOnLoad:true标志；解析器在加载后会检测该标志并据以触发自动解析过程。
- 通过dojo.require("dojo.parser")请求解析器，以便加载解析器并注册在页面加载后自动调用dojo.parser.parse()。由于调用parse方法时未传递任何参数，因此以整个页面主体作为解析目标。
- 在标记中为需要解析的部件添加dojoType属性。

在页面加载后手动解析在标记中定义的部件也很类似：

- 通过dojo.require("dojo.parser")请求解析器。由于parseOnLoad的值不为true，因此不会自动调用dojo.parser.parse()。
- 为部件所在的标记指定相应的dojoType属性，甚至可以在页面加载完成后再动态指定。
- 手动调用dojo.parser.parse()；也可以传递一个特定的DOM节点，作为解析操作的起点。

但是，实际的解析过程又是怎样的呢？就是找到所有dojoType属性并把相应的标记实例化为部件吗？当然，在彻底明白之后，读者会认识到它只不过是一个简单的自动化的过程。以下就是实际的解析过程：

- 调用dojo.query("[dojoType]")，以便确定页面中需要解析的节点。
- 从每个节点中提取出类（正如dojo.declare所声明的）信息，迭代遍历所有属性并执行简单的类型转换。转换后的属性将作为类的constructor函数的参数。
- 检测节点内部带有值为dojo/method或dojo/connect的type属性的SCRIPT标签，并做好处理准备。（更多内容请参考本章后面的“在标记中定义方法”。）
- 在未定义markupFactory方法的情况下，通过constructor函数创建类的实例；否则，使用markupFactory创建类的实例。可以通过markupFactory这个特殊

的方法来为部件定义一个自定义的构造函数，以便部件具有不同于以编程方式创建的部件的初始状态。所有Dojo部件的基类是_Widget，该类提供了一组标准的生命周期方法。其中一个生命周期方法用于把部件的domNode插入页面，以便部件在页面中显示出来。下一章将详细介绍部件的生命周期方法。

- 如果存在jsId属性，那么把类的实例映射到全局JavaScript命名空间当中。（常用于数据存储和需要在全局作用域中访问的部件。）
- 处理type属性为dojo/method或dojo/connect的SCRIPT标签定义的连接（本章稍后介绍相关内容），并调用每个部件的startup生命周期方法。startup也是继承自_Widget的一个标准生命周期方法（下一章将会讨论到），可以通过该方法来操作包含在刚刚实例化之后的部件中的部件。

但愿读者在知道了解析器的工作原理之后，不会因为我们揭去了它的神秘面纱而感到失望，毕竟这些原理迟早都是要知道的。在下一章专门讨论Dojo部件的生命周期方法时，我们将对上面提及的概念再进行深入讨论。

# 动手构建 NumberSpinner 部件

为了给读者对后面几章的学习打下基础，本节将展示一个非常直观的 Dojo 部件 —— dijit.form.NumberSpinner的创建过程。首先，看一看如何在标记中创建这个部件，然后，我们再以编程方式来创建同样的部件。

## 在标记中创建部件

通过前面对解析器的介绍我们知道，向页面中插入Dojo部件非常简单。首先要加载相应的资源，再在标记中指定dojoType属性，然后解析器就可以在后台自动完成对部件的初始化了。例 11-4 展示了一个遵循这种模式实例化 NumberSpinner 部件的完整示例。

例 11-4：在标记中创建 NumberSpinner 部件

```
<html>
    <head>
        <title>Number Spinner Fun!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
```

```
            djConfig="parseOnLoad: true"
        ></script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.NumberSpinner");
        </script>
    <head>
    <body class="tundra">
        <form> <!-- some really awesome form -->
            <input dojoType="dijit.form.NumberSpinner"
                constraints="{min:0,max:10000}" value=1000>
            </input>
            </form>
        </body>
    </html>
```

## 以编程方式创建部件

在标记中创建部件是一种常用方式，而以编程方式创建部件则是另一种常用方式。以编程方式创建部件的过程与创建其他Function对象的过程非常相似，毕竟，一个部件本身就是一个Function对象。通常，部件对象的constructor接收两个参数：第一个参数是一个对象，包含着需要传入的属性，这些属性与在标记中创建部件时为标签指定的属性相同；第二个参数是一个源节点或源节点的ID，表示部件应该替换的DOM元素：

```
var d = new module.DijitName(/*Object*/props, /*DOMNode|String*/node)
```

例11-5展示了以编程方式创建NumberSpinner部件的完整代码，结果与例11-4完全相同。

例11-5：以编程方式创建NumberSpinner部件

```
<html>
    <head>
        <title>Number Spinner Fun!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />
        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        ></script>

        <script type="text/javascript">
            dojo.require("dijit.form.NumberSpinner");
            dojo.addOnLoad(function() {
```

```
                var ns = new dijit.form.NumberSpinner(
                    //属性
                    {
                        constraints : {min:0,max:10000},
                        value : 1000
                    },
                    "foo" //节点 id
                );

                // 对 ns 执行其他操作的代码……
            });
        </script>
    <head>
    <body class="tundra">
        <form>
            <input id="foo"></input>
        </form>
    </body>
</html>
```

## 优雅的特性

前面示例中创建的NumberSpinner部件是一个非常精美的、带有数值调整控件（上下箭头）的小文本框，这种部件通常称为数值微调器（NumberSpinner）。用户可以通过键盘中的方向键或者单击部件上的调整控件，或者通过键盘输入来调整文本框中的数值。微调器可以调整的最小和最大数值由 constraints 属性的 min 和 max 键/值对设定。无论是使用方向键还是单击调整控件，文本框中的值都不会超出设定的范围。而value属性则用来指定文本框的默认值，就像是在HTML中一样。当然，通过键盘输入也可以直接修改文本框的值；不过，如果输入的值超出了范围或不是数值，那么会触发一个提示条并在其中显示错误消息。图 11-3 展示了这个微调器部件及错误提示条。

**注意：** 对Web应用程序开发中的现有构造而言，Dojo所提供的功能只是增强而不是重写它们。因此，表单元素中的常用属性，如前面代码中的 input 元素的 value 属性，照样有效。

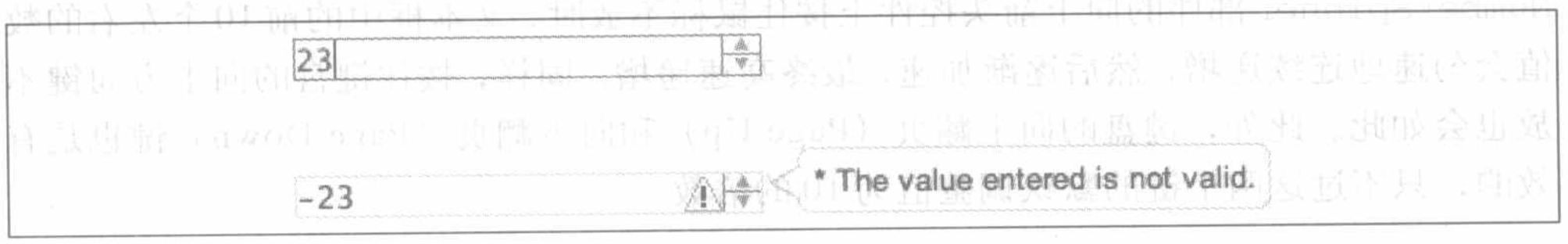



图 11-3：上：可以通过键盘方向键或调整控件调整数值的NumberSpinner部件；下：当通过键盘输入的数值越界时显示的错误提示消息

警告：尽管SCRIPT标签的djConfig配置项中构成对象的键/值对没有最外部的花括号，如：djConfig="parseOnLoad:true,isDebug:true"，但这只是一个例外。在创建Dojo部件时，位于标记中的对象属性值必须用花括号括起来，如：constraints="{min:0,max:100}"。

从NumberSpinner部件的键盘输入操作，读者很可能会联想到a11y。事实上，这个部件还包括其他一些有价值的优雅特性。首先，这个部件能够自动为超过999的数值添加千分位分隔符。而且，如果是其他受支持地区的用户在查看这个页面，那么这个部件会自动应用该地区的数位分隔符：如果地区是*en-us*（美国），那么会使用逗号作为分隔符，如1,000；如果地区是*es-es*（西班牙），就会使用句点作为分隔符，如1.000。图11-4展示了这两种分隔符。为了验证，读者可以在djConfig中修改默认的地区项。例如，要把默认地区设置为西班牙，应该添加下面这个项：

```
djConfig="locale:'es-es'"
```

警告：记住，djConfig中的任何字符串值都需要加上引号。但是，在行内声明关联数组的语法很容易让人忘记这一点。而且，由此导致的错误消息往往也不能明确指出真正的问题所在。另外，djConfig中的任意配置项都必须在Base启用之前加载。

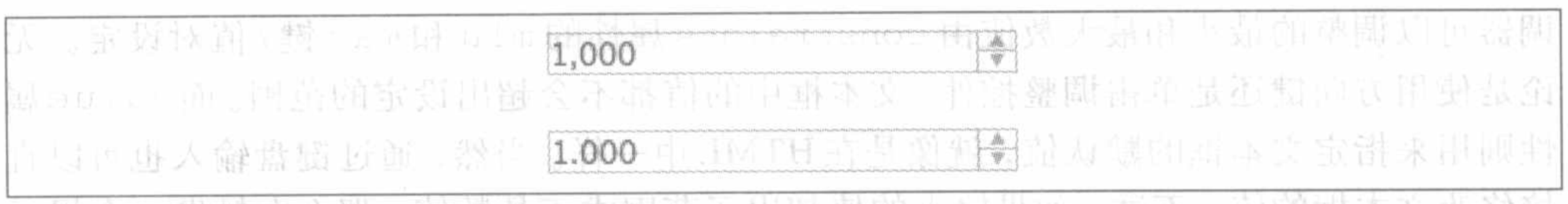



图11-4：无须任何额外的配置，Dijit就能够自动处理受支持地区的特殊格式；图中上方的NumberSpinner部件是将地区设置为en-us时的显示效果，下方的NumberSpinner部件是将地区设置为es-es时的显示效果，所有细节的处理都由Dijit在内部完成

Dojo部件默认支持的另一个优雅特性是控件能够自动击键（typematic）——即在按住鼠标或方向键不放的情况下，控件能够连续响应单击或按键事件。具体来说，当在NumberSpinner部件的向上箭头控件上按住鼠标不放时，文本框中的前10个左右的数值会匀速地连续递增，然后逐渐加速，最终高速递增。同样，按住键盘的向上方向键不放也会如此。此外，键盘的向上翻页（Page Up）和向下翻页（Page Down）键也是有效的，只不过这两个键的默认调整值为10的倍数。

## 在标记中定义方法

除了支持以编程方式编写JavaScript来控制和扩展部件，Dojo也支持直接在标记中定义

JavaScript逻辑，需要在SCRIPT标签中添加一个特殊的type="dojo/method"属性。这种能力无论是对设计者而言还是对其他人而言，都能很方便地帮助他们快速验证一种新想法。在标记中定义方法时，特别要注意关键字this引用的是包含部件，而这又为访问当前的执行环境提供了便利。

下面我们看一看基于前面示例，在标记中定义方法的代码片段：

```
<!-- 省的代码略 -->

        <script type="text/javascript">
                dojo.require("dojo.parser");
                dojo.require("dijit.form.NumberSpinner");
                dojo.require("dijit.form.Button");
         </script>
        </head>
        <body class="tundra">
            <form>
                <div dojoType="dijit.form.NumberSpinner" jsId="mySpinner"
                    constraints="{min:0,max:10000}" value=1000>
                    <script type="dojo/method">
                        dojo.mixin(this, {
                            reset : function() { this.setValue(1000); }
                        });
                    </script>
                </div>
            </form>
            <button dojoType="dijit.form.Button" onClick="mySpinner.reset()">reset</
button>
        </body>
</html>
```

为了使用定义的方法，我们通过jsId属性为NumberSpinner部件声明了一个全局变量名mySpinner，然后在Button的onClick方法中就引用了这个变量名。这里reset方法的方法体是在位于Dojo部件内部的一个特殊的SCRIPT标签中定义的。这个提供匿名dojo/method的SCRIPT标签会在部件的constructor函数运行之后执行，因此通过标记属性传入部件的值对于这个方法而言都是有效的。

此外，要注意前面的示例在创建微调器时使用的是input元素，而这个示例使用的则是div标签。之所以这个示例中没有使用input标签，是因为input不支持innerHTML属性；只有转换后的元素支持innterHTML，才能确保方法有效。至于为什么没有全都采用div标签，一个根本原因就是：在页面中使用语义正确的标签，能够保证用户在没有JavaScript的情况下仍然可以使用该页面。换句话说，前面示例的表单中使用了input这个语义正确的标签，可以保证用户在JavaScript无效（无论什么原因）的时候仍然可以使用输入控件。当然，多数情况下这都不是个问题。但是，在设计可退化的（degradable）页面时，掌握基本的HTML知识并考虑到类似这样的问题就显得十分重要了。

**警告：** 在不支持innerHTML属性的元素中，通过dojo/method和dojo/connect来定义方法的SCRIPT标签无效。问题并不在于Dojo，而是遵循HTML规范的结果。另外，虽然我们没有在此给出示例，但使用包含type="dojo/connect"的SCRIPT标签可以在标记中以相同模式来建立连接。

事实上，添加重置按钮只是为了引入基于鼠标的控件；如果没有这个控件，通过键盘的Esc键同样能把微调器的值重设为原始值。

而且，要达到相同的目的，还有另一种代码量更少的实现方案。读者可以按照下面的代码修改dojo/method SCRIPT标签：

```
<script type="dojo/method" event="reset">
    this.setValue(1000);
</script>
```

与在constructor函数之后自动执行一次提供匿名dojo/method的SCRIPT标签不同，这种方案实际上是在创建了reset方法之后又把它添加给了部件。如果在此还需要为方法传递参数的话，那么就要再指定一个args属性。例如，通过args="foo,bar,baz"可以向在标记中定义的方法传递3个命名参数。

# 内置部件一览

鉴于Dojo内置的部件实在太多，不容易记清楚，本节提供了一个简明的Dojo部件清单，以便读者随时参考。

## 表单部件

“表单部件”这个名称暗示了该类部件的主要设计目标就是在表单中使用。尽管这个结论没错，但把这类部件用在表单外部，或者用在一个特殊的、提供额外方法和扩展点的dijit.form.Form部件中也是可以的。以下我们就简单地介绍一下本书随后几章将要详细讨论的部件概况。请读者注意，所有Dojo部件都符合a11y标准，而且在需要时都可以方便地进行国际化处理。

**注意：** 访问*http://archive.dojotoolkit.org/nightly/*可以查看包含所有部件的Dijit测试套件。这是一种体验Dojo部件覆盖面之广之深的极好方式。

Form

一个用于放置表单部件的特殊容器，提供了便捷的方法和扩展点，支持串行化JSON、验证表单内容、一次性设置表单的值以及提交表单时的事件处理。

Button 及其变体

替代基于 BUTTON 元素的普通按钮，以及其他形式的按钮 —— 基于 INPUT 元素的控件，如复选框和单选按钮。Button部件的变体包括常见于工具条中具有下拉项的菜单式按钮（类似组合框）和切换按钮（如“加粗”和“斜体”按钮）。

ComboBox

兼具普通的 SELECT 组合框和由 INPUT 元素定义的文本字段的功能，用户可以从中选择预填充的选项或者输入自己认为合适的值。

FilteringSelect

替代普通的SELECT元素。由于可以动态填充选项，因此特别适合可能的选项非常多的场合。

NumberSpinner

类似基于INPUT元素的文本框，但可以通过调整控件渐近地调整文本框中的数值。

Slider

在垂直或水平的刻度尺上附着了一个可以拖动的手柄。这个部件为在二维空间内实时调整对象的数值提供了一种更具交互性的方式，通常还会带一个显示数值的字段。

Textarea

替代普通的 TEXTAREA 元素，但该部件能够根据其中的内容自动调整大小，从而确保在内容长度无法预知或长短不定的情况下节省宝贵的屏幕空间。

SimpleTextarea

替代普通的TEXTAREA元素，同时也提供了与容器部件Form和布局部件进行交互的机制。

MultiSelect

替代普通的SELECT元素，但multiple属性被设置为true。与SimpleTextarea部件类似，MultiSelect 部件也提供了与 Form 部件进行交互的机制。

TextBox 及其变体

基于 INPUT 元素构建的一组功能丰富的部件集合，支持对用户输入值的自定义验证和常见字段格式的验证，例如，日期、时间、货币、数字等。这组部件中包含的实用功能极多。

## 布局部件

处理复杂布局的传统技术往往要涉及庞杂的CSS编码工作。虽然CSS并不是什么尖端科技，但要基于多浏览器编写、测试和维护CSS也不是件轻而易举的事，除非你是真正的CSS专家。布局部件的基础是标记，不是嵌套的表格，因此用它们来实现布局设计相对更容易。一般来说，布局部件之间可以随意嵌套，从而能够完成极其复杂精密的设计，而且所花时间也不过是使用传统CSS技术所花时间的几分之一。以下是对各种布局部件的简单介绍。

ContentPane

它是进行布局的最基本构建块，为安排布置子部件提供了事实上的容器。虽然这种部件可以单独使用，但在更多情况下则是将一个或多个ContentPane作为容器部件的组成部分。

TabContainer

用于构建人们熟悉的、带标签页的可切换式布局。其中标签页可以横向，也可以纵向排列。最简单的TabContainer布局一般也要涉及到TabContainer和ContentPane的组合；不过，这两种部件之间的任意嵌套也是允许的。注意，初始状态并不显示的标签页内容可以延迟加载。

StackContainer

能够同时显示多个信息容器，但每次只有其中一个容器可见。例如，多个ContentPane部件中分别包含着用于展示的幻灯片。对于既要显示多个“屏幕”，又不想页面重载的情形，使用StackContainer部件会非常方便。

AccordionContainer

每次只显示一个窗口，当选中另一个窗口的标题时，以前显示的窗口会通过平滑的动画折叠起来。初始状态下不显示的窗口内容可以延迟加载。

BorderContainer

便于创建典型的“大标题”风格或“侧边栏”风格的布局，这种布局通常由多个区块构成，其中的某个区块可能会与整个布局同宽或同高，而其他区块则长短不一。在屏幕上构建由5个区块组成的复杂的“边框式”布局（其中4个区块各占一边，而中间区块则占据剩余空间）不过举手之劳。

## 应用程序部件

应用程序部件属于“其他”类别；这些部件对于要构建的任何RIA功能而言，都是不可或缺的常用元素。例如，菜单、工具条、对话框（覆盖下方内容）、富文本编辑器都是常见的应用程序元素。而用来构建这些元素的部件，简直具有让人难以抗拒的易用性。

Menu

用于创建上下文弹出菜单，与在桌面应用程序上单击鼠标右键时调出的菜单类似。Menu 也可以用来构建像 ComboButton 和 DropDownButton 一样的复杂按钮，以实现高级功能。

Toolbar

为 ToggleButton（用于向 Toolbar 中提供控件）等复杂按钮提供容器。当然，任何按钮部件都可以包含在 Toolbar 部件中。

Dialog

模拟普通的桌面对话框，能够以半透明效果覆盖于其他部件之上，从而阻止用户与对话框“下方”的内容交互。在多数情况下，都可以用外观漂亮、维护方便的Dialog部件替代弹出窗口，尤其是需要在多窗口间实现某种通信或 DOM 操作的情况下（注 4）。

TooltipDialog

是 Tooltip 和 Dialog 的组合，用于在工具条中创建对话框风格的输入字段。Dialog与TooltipDialog之间的重要区别在于，单击TooltipDialog之外的任何地方都可以使其消失；而Dialog则提供了半透明的中间层，可以在Dialog被关闭之前阻止用户与其他部件交互。

ProgressBar

模拟在桌面应用程序中常见的普通进度条。ProgressBar部件是为费时操作（例如，对服务器的异步请求需要几秒钟才能返回）向用户给出反馈的标准方式。ProgressBar部件分为确定性进度条和不确定性进度条两种。确定性进度条显示完成的百分比作为提示，不确定性进度条则显示任意动画效果以示操作正在进行。

TitlePane

用于显示一个上方带有标题栏的信息窗格。窗格区域可以通过单击标题栏中的图标关闭和展开，但标题栏始终可见。

Tooltip

替代普通 HTML 提示控件的一种更灵活方式。提示文本中可以包含任意 HTML 代码，而且还具有定时隐藏功能。

InlineEditBox

像标签一样显示部件值的一种容器。但是，在单击文本后该部件会转换成可编辑状态。（包含很多功能。）

---

注 4：　在某些浏览器中，由于存在安全限制，不能操作另一个窗口中的 DOM 节点。

ColorPalette

在默认情况下，以 3 × 4 或 7 × 10 的表格显示常用颜色，便于用户选择。对 ColorPalette 进行配置后，可以通过它来显示任意颜色。

Editor

具有最基本的富文本编辑器功能，工具条中包含预配置的剪切/复制/粘贴、撤销/恢复、文本对齐、加粗/斜体/删除线以及项目列表按钮。工具条可以自定义。Editor 部件中内置了大量功能，但却比通常的编辑器更容易使用，因为它构建于特殊的本地控件（例如，Firefox 的 Midas 富文本编辑器）基础之上。

Tree

用于构建可以任意嵌套、随便开合的节点树。这种界面控件适合展示内容较多，而且层次分明的信息项。默认不处于展开状态的节点内容可以延迟加载；该部件使用功能强大的 dojo.data API 传送内容。

# Dijit API 简介

表 11-4 列出了最常用的部件方法。这些方法由 Dijit Base 提供，并在向页面中请求 Dijit 资源时生效。也可以通过 dojo.require("dijit.dijit") 语句让这些方法生效，因为它们都包含在标准的构建文件中，具体内容请参见第 16 章。

**注意：** 要全面了解 Dijit API 文档，请参考 Dojo 的在线文档，地址为 *http://api.dojotoolkit.org*。

**表 11-4：常用的 Dijit 方法**

| 方法/成员 | 说明 |
| --- | --- |
| dijit.registry() | 注册表中包含页面中所有部件的记录，可以基于注册表对所有部件进行迭代遍历、检查某个部件是否存在等。例如，要对页面中的所有部件执行统一操作，可以使用 dijit.registry.forEach 方法；要查询页面中特定类型的部件，可以使用 dijit.registry.byClass（这里的 Class 是 OOP 意义上的“类”）方法 |
| dijit.byNode(/* DOM Node */ node) | 基于给定节点返回表示该节点的部件 |

表 11-4：常用的 Dijit 方法（续）

| 方法 / 成员 | 说明 |
| --- | --- |
| `dijit.getEnclosingWidget (/* DOM Node */ node)` | 基于给定节点返回 DOM 树中包含该节点的部件。这个方法特别适合通过 DOM 事件引用部件的情形。例如，当用户单击部件的某一部分时，将事件对象的 `target` 属性传递给这个方法，就可以取得相应的部件 |
| `dijit.getViewport()` | 返回浏览器窗口中可浏览区域的大小及滚动位置，在实际的屏幕分辨率和窗口大小无法预知的情况下，可以利用这个方法在屏幕上以编程方式设置对象的位置。常用于创建动画效果 |
| `dijit.byId(/* String */ id)` | 基于 `dojoType` 属性所在标签的 `id` 值或者以编程方式创建部件时传入的 `id` 值查询页面中的部件。这个方法与 `dojo.byId` 的区别在于，`dojo.byId` 返回一个 DOM 节点，而它返回实际的部件（即 Function 对象） |

虽然读者有必要了解的 API 方法不止这些，但这些却是其中最常用的，牢记这些方法一定会受益匪浅。

# 小结

在学习了本章之后，读者应该理解：

- Dijit 设计目标背后的基本思想。
- 低耦合和高内聚在开发复杂应用程序时的重要性，以及 Dijit 如何利用这些思想在部件中实施封装功能。
- 应用程序易访问性（a11y）和 W3C WAI-ARIA 的重要性，以及 Dijit 实现 a11y 的基本手段。
- 在标记中创建的部件与以编程方式创建的部件具有完全相同的功能，以及如何为标记中的部件应用 `dojo/method` 和 `dojo/connect` SCRIPT 标签。
- DOM 节点与 Dojo 部件的区别；`dojo.byId` 与 `dijit.byId` 的区别；DOM 事件与 Dojo 部件事件的区别。

- 解析器实例化在标记中定义的部件时所要经历的基本步骤。
- 由表单部件、布局部件及通用的应用程序部件构成的Dijit的基本架构，以及到哪里查找并体验特定部件的功能。

下一章，我们将剖析Dojo部件及其生命周期。

# 第12章

# 深入理解Dijit及其生命周期

与其他编程语言中面向对象的概念类似，Dojo部件（Dijit）在处理其生命周期事件（如创建和销毁部件）时同样遵循了特定的模式，也建立在特定的组织结构之上，并且需要通过专门的词汇表来描述。本章将对这些内容给出简明的介绍，并进一步讨论自定义部件设计的基本原理。

## 理解Dijit

虽然读者已经知道Dijit表示的是“Dojo部件”了，但在深入分析Dijit之前，再对它多了解一些还是很有必要的。确切地讲，一个部件就是一个Dojo类，它继承了名为_Widget的Dijit基类。这个基类是工具箱中基础Dijit模块的一部分，因此其完全限定名是dijit._Widget。Dijit也提供了其他一些基类，这些基类以后会介绍到，但只有_Widget才是所有部件的一个最主要的超类（ancestor）。

正如我们在第10章介绍过的，dojo.declare为开发人员消除了重复编写冗余代码的麻烦；Dijit则照例将大量的复杂性一股脑地封装到了像_Widget这样的类中。稍后我们会讨论到，这些基类中提供了很多可以重写的方法，让开发人员能够方便地实现自定义行为，从而降低了工作量。

---

**注意：** 读者可能对_Widget类名前的下划线感到奇怪。事实上，当使用Dojo部件时，类名前面的下划线通常用于告诉开发人员不应该对该类直接进行实例化，而应该把这些类作为继承关系中的超类。

---

接下来，对Dijit的讨论首先从我们最熟悉的部件的物理层开始，包括HTML、CSS、

JavaScript以及始终都要使用的其他静态资源。在了解这些基本内容之后，再介绍部件的生命周期模型，而理解生命周期模型则是以对dojo.declare的理解为基础的。最后，我们通过一个比较复杂的代码示例来说明设计自定义部件的各方面细节。

## 回顾 Web 开发的历史

稍有计算机知识的人都知道，HTML 是在 Web 浏览器中显示信息的事实标准。对于标题、段落、表单字段等标准化的内容，通常只需使用相应的标记就可以显示文本或者图像信息。不过，HTML 本身的外观表现却很难尽如人意：它不能对文本进行适当的格式化，而且内容完全静止。页面的整体结构非常简单，不能对用户的操作给出响应。随着时间推移，人们对仅仅由 HTML 构成的 Web 越来越感到难以忍受。

然而，只要加入少量CSS规则，页面马上就能变得大不相同。突然之间，页面的艺术性得到了极大的增强。HTML本身提供的视觉吸引力固然很少，但以它为基础的CSS则为页面注入了布局和排版的活力。可是，添加样式之后的页面并没有“活”起来，因此并没有满足人类本能渴望的互动需求。换句话说，虽然可以使用HTML和CSS创建美观的页面，但仍然缺少生机。

JavaScript 为应用样式的 HTML 补充了极度缺乏的动态性，并且催生了 DHTML。DHTML带动了交互性在线体验的发展和繁荣，最终进一步导致了现代的富因特网应用程序（RIA）的诞生。JavaScript 为网页注入了生机 —— 当我们轻点鼠标、做出选择，或者随意按下键盘，网页随之应声而动时，我们会因为计算机能够理解自己的想法而获得极大的满足。

虽然大家希望每个网页都经过了精心设计，而且具有交互能力，但要真正做到这一点却没有那么容易。用来控制HTML和CSS的JavaScript日益复杂，即使最高明的实现都可能在易维护性上存在缺陷，或者难有可圈可点之处。其中最核心的问题在于很难把 HTML、CSS 和 JavaScript 整合到一个单独的、统一的框架之内。如果在设计中增加了内聚性，协同的问题就会浮出水面；如果为协同而选择了折衷，那么呆板的实现就会导致活力荡然无存。雪上加霜的是，编写 JavaScript 代码的艰苦历程会很快消磨光人们对构建应用程序的热情和创意。

## Dijit 来拯救

所幸的是，Dijit为开发人员提供了构建复杂Web设计的基础。所有部件都是将HTML、CSS 和 JavaScript 整合到一起的产物，尽管还不算完美，但通过它们却能够以面向对象的方式更有效地思考问题，因而最终可以在精力和创意干涸之前，迅速构建起应用程序的核心部分。也许以前你必须使用自己的方式将 HTML、CSS 和 JavaScript 整合起来，

但现在 Dojo 已经替你把这些工作都做完了，而你要做的就是更有成效地挥洒自己的创意。

就像独立的类一样，Dojo部件也以自包含的形式存在于与一个命名空间对应的目录下。不过，除了一个 JavaScript 文件外，目录中还会包含图像、样式表等相关资源。清晰明了的目录结构天然地具有易移植性，从而也让共享、部署和升级部件变得异常简单。此外，由于不存在需要解析的二进制数据，因此部件的所有组成部分都能够以一个独立文件的形式，纳入到像 Subversion 这样的版本控制系统当中。

虽然把构成一个部件的所有资源一股脑地放入一个目录中也是一种方法，但Dojo却采用了另一种结构在磁盘上保存部件，如图 12-1 所示。从本质上讲，这种结构只不过是将部件划分为不同的子目录，但这种划分却能够保证部件容易管理。

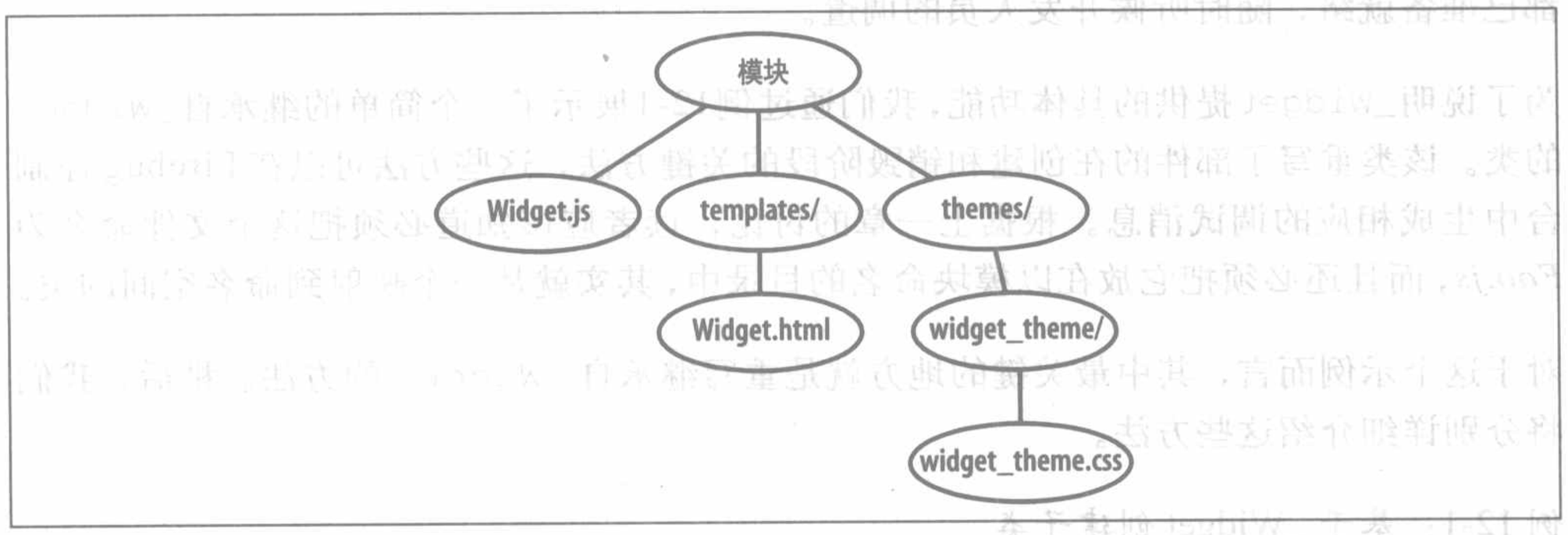



图 12-1：部件在磁盘上的目录结构

Dijit整合HTML、CSS和JavaScript形成的部件都是Web开发中最常用的界面控件，为开发人员提供了统一的、标准的应用程序构造和创意素材。最终，将为开发人员节省大量时间和精力，而且还有助于得到更有效的设计结果。注意，对于不包含模板及CSS的最小化的 Dojo 部件而言，它的目录结构中只包含一个 JavaScript 文件目录。

图 12-1 中的目录结构显示，模板包含在一个单独的 HTML 文件中。这是一种典型的面向开发的设计方案，因为它允许开发团队的成员分别编写模板、CSS和JavaScript文件。

另外，取得模板还需要JavaScript引擎向服务器发送相应的同步调用；不过，Dojo以一种非常高效的方式优化了这些同步调用，因而开发人员在JavaScript文件中使用一个行内字符串就可以加载模板文件。稍后我们会通过大量的示例来说明这个过程有多么简单。

# Dijit的生命周期方法

接下来，我们就把注意力放在_Widget提供的关键的部件生命周期方法上。应该说，_Widget封装了很多强大的功能，而开发人员要使用这些功能则只需付出极少的努力。通过将_Widget作为继承关系中的一个主要超类，那么相应的子类就会马上拥有它所提供的标准生命周期方法。当然，在创建和销毁部件期间，开发人员为了自定义部件的行为可以随时重写这些方法。

例如，_Widget在一个部件出现在屏幕上之前、刚刚出现之后以及即将被销毁时，都提供了可以重写的方法。而这几个重要的瞬间，往往正是与服务器端模型同步、明确地销毁对象（以避免众所周知的内存泄漏问题），或者执行某些策略性DOM操作的最佳时机。但是，无论具体是哪种情况，都无须开发人员自己动手实现其中的细节；因为一切都已准备就绪，随时听候开发人员的调遣。

为了说明_Widget提供的具体功能，我们通过例12-1展示了一个简单的继承自_Widget的类。该类重写了部件的在创建和销毁阶段的关键方法，这些方法可以在Firebug控制台中生成相应的调试消息。根据上一章的讨论，读者应该知道必须把这个文件命名为*Foo.js*，而且还必须把它放在以模块命名的目录中，其实就是一个映射到命名空间的类。

对于这个示例而言，其中最关键的地方就是重写继承自_Widget的方法。稍后，我们将分别详细介绍这些方法。

例12-1：基于_Widget创建子类

```
dojo.require("dijit._Widget");
dojo.addOnLoad(function() {
    dojo.declare(
      "dtdg.Foo", // 子类
      dijit._Widget, // 超类
      {
        /* 按照调用时间的先后顺序排列的常用构建方法 */
        constructor : function() {console.log("constructor");},
        postMixInProperties : function() {console.log("postMixInProperties") ;},
        postCreate : function() {console.log("postCreate");},

        /* 以下是自定义的逻辑 */
        talk : function() {console.log("I'm alive!");},

        /* 正式的析构函数，通过destoryRecursive()隐式调用 */
        uninitialize : function() {console.log("uninitialize");}
      }
    );
});
foo = new dtdg.Foo();
```

```
foo.talk();
foo.destroyRecursive(); /* 调用uninitialize */
```

在运行这个示例后，读者应该在 Firebug 控制台中看到下列输出结果：

```
constructor
postMixInProperties
postCreate
I'm alive!
uninitialized
```

## _Widget 的生命周期

以下是我们曾在第10章讨论过的通过dojo.declare创建类的基本模式，从中可以看到插入的 _Widget 的生命周期方法：

```
preamble(/*Object*/params, /*DOMNode*/node)
    //先于constructor函数执行；可以操作超类构造函数的参数

constructor(/*Object*/ params, /*DOMNode*/node)
    // 触发所有超类构造函数
    // 触发所有混入的构造函数
    // 触发本地类的构造函数（如果有）

postscript(/*Object*/ params, /*DOMNode*/node)
    //_Widget实现了postscript，以启动create方法……
    _Widget.create(/*Object*/params, /*DOMNode*/node)
    _Widget.postMixInProperties()
    _Widget.buildRendering()
    _Widget.postCreate()
```

由此可以得出两个结论：

- _Widget完全基于dojo.declare构建，并且实现了postscript方法，以便触发系统地调用 _Widget 生命周期方法的 create 方法；
- 作为超类的 _Widget 部件，就是一个地道的 JavaScript Function 对象。当然，这个对象封装了大量实用的功能，不过它最终还是作为基类存在。

### 生命周期方法

以下是对生命周期及其提供的每个生命周期方法的介绍。从发生时间的先后顺序上来看，部件的生命周期开始于 preamble 方法，因为重写 postscript 或 create 方法的情形并不多见（假如读者想设计自己的部件生命周期方法而不使用标准方法，当然也没有问题）。本章稍后将给出全面展示每个方法用途的相关示例。

*preamble（源自* dojo.declare*）*

preamble为在constructor接收到参数之前处理参数提供了机会。如果读者要重写preamble方法，那么一定要记住正常情况下传入constructor中的参数都会传入preamble中。而且，无论preamble返回什么，其返回结果都会传递给constructor。与其他生命周期方法相比，这个方法拥有一些不太常用的高级特性，例如，postCreate。

*constructor（源自* dojo.declare*）*

这是在部件创建期间为实现自定义行为可以重写的第一个方法。constructor中会执行两项特别基本的操作。一项是对非原始类型的部件属性进行初始化。（第10章曾经介绍过，如果将对象或列表等复杂类型作为对象参数的属性来声明，那么这个对象或列表将被所有对象实例共享）另一项基本操作是添加其他下游生命周期方法必需的属性。

*postMixInProperties（源自* dijit._Widget*）*

这个方法会在Dojo处理继承关系并向子类中混入所有基类属性时被调用。从字面上理解*postMixInProperties*这个名称，指的就是所有部件的属性被混入到特定对象实例之后的时刻。因此，当这个方法执行时，子类已经可以访问所有继承的属性，并在部件出现在屏幕上之前操作这些属性了。正如稍后的一个根据模板派生部件的示例所展示的，这个方法是用来修改或派生模板标记中占位符（格式为${someWidgetProperty}）的恰当位置。

*buildRendering （源自* dijit._Widget*）*

在_Widget的实现中，这个方法只用来将内部的_Widget.domNode属性设置为一个实际的DOM节点，以便部件真正成为页面的一部分。正因为这个方法在postMixInProperties方法之后被调用，所以才说明postMixInProperties是修改部件模板的恰当位置。

稍后我们将会介绍，另一个基础的部件类_Templated会重写这个方法，以便执行与取得和实例化部件模板相关的各种操作。此外，需要注意的是，在buildRendering被调用之后，部件本身会立即被添加到Dojo的部件管理对象，从而保证部件能在明确调用析构方法和/或页面卸载时被适当销毁。鉴于某些浏览器存在人所共知的内存泄漏问题，而这个问题对于长周期运行的应用程序尤为紧要，因此Dojo的部件管理对象作为部件注册跟踪机制的核心，是Dojo解决内存泄漏问题的关键所在。重写这个方法的必要性并不大；通常只要使用_Widget或_Templated提供的默认实现就可以了。

*postCreate （源自* dijit._Widget*）*

这个方法会在部件已经创建完成并且出现在页面上时执行，因此可以通过它来执行此前不可能执行或者需要谨慎执行的任何操作。换句话说，通过postMixInProperties执行那些影响部件样式或在屏幕上位置的操作时务必多加小心，因为这些操作是在部件可见之前发生的。但是，在postCreate中执行类似的操作有时又会导致页面短暂地“抽搐（jerks）”，因为这个方法操作是已经可见的部件；而且，如果对postMixInProperties 和postCreate方法的区别又没有搞清楚，那么查找原因和解决问题将会变得很困难。此外，如果一个部件中包含子部件，那么在这个方法中访问子部件也不安全。要想安全地访问子部件，应该使用另一个生命周期方法 startup。如果要安全地访问非子部件，那么必须等到页面加载之后使用dojo.addOnLoad。

*startup（源自* dijit._Widget*）*

对于在标记中声明的子部件，这个方法会在当前部件及其所有子部件都创建完成后被自动调用。因此，该方法是子部件可以安全地引用其他子部件的可靠位置。正因为听起来很简单，所以经常有人在postCreate方法中尝试引用子部件，但那样做会导致不一致的行为，而且要查找原因和解决问题也很困难。对于通过编程方式创建的部件，并且该部件包含符合“有一个”关系的其他子部件，那么就需要在所有子部件都创建完毕后手动调用startup方法。之所以需要在以编程方式创建包含子部件的部件时手动调用startup方法，原因就是在所有子部件都各就各位之前调整大小和呈现部件没有意义。（否则，很可能导致许多不成功的启动。）这个方法是在部件创建期间用于加入自定义行为的最后一个可以重写的方法。

*destroyRecursive（源自* dijit._Widget*）*

这是一个为清理部件及其所有子部件而调用的通用的析构函数。在销毁部件的过程中，这个方法会调用uninitialize方法，而uninitialize是可以重写、以便加入自定义清理操作的基本方法。但是，千万不要重写destroyRecursive。如有必要，可以在uninitialize中加入自定义的清理操作，然后再调用destroyRecursive方法（它不会被自动调用），然后它就会替我们完成相应的处理。

*uninitialize （源自* dijit._Widget*）*

重写这个方法可以在销毁部件时加入自定义的清理操作。例如，可以向服务器发送请求以保存会话状态，或者显式地清除DOM引用。这个方法是所有部件在执行销毁操作时都应该使用的恰当位置。

警告：搞清楚各种生命周期方法之间错综复杂的关系是绝对必需的。尤其要注意什么类型的行为应该发生在哪个方法中。

在使用生命周期方法时，最常见的问题包括：

- 在postMixInProperties方法被调用之后操作模板。
- 在postMixInProperties方法被调用之后修改部件的初始外观。
- 在postMixInProperties方法而不是startup方法中访问子部件。
- 忘记在uninitialize中执行必要的销毁操作。
- 调用uninitialize而不是调用destroyRecursive。

### 基本的属性

除了前面刚刚介绍的_Widget方法之外，还有一些非常重要的属性。与部件方法类似，可以通过点运算符来引用这些属性。而且，通常都应该把下面这些属性看成是只读的：

id

这个值中保存着指定给部件的唯一标识符。如果没有为部件指定唯一标识符，Dojo会自动指定一个。绝对不能修改这个值；而且在大多数情况下，也用不到这个值。

lang

Dojo支持的一种国际化特性，这个值可以用来自定义显示部件的语言。在默认情况下，这个值根据浏览器的设置定义，而该值通常与操作系统的语言一致。

domNode

这个属性中保存着部件最外部节点的引用。这个节点是在屏幕上呈现部件视觉外观的正式节点，除非是在调试的时候，一般不需要直接操作这个属性。如前所述，_Widget默认的buildRendering实现会设置这个属性，因此，在重写buildRendering方法时，不要忘记设置该属性；否则，就可能导致意想不到的结果。在部件尚未出现在屏幕上面时，这个属性中保存的值是undefined。

为了印证上述内容，下面这个示例会在Firebug控制台中输出部件的属性。同样，所有这些属性都继承自_Widget，并且可以通过this访问，此时的this引用的是作为dojo.declare第三个参数的关联数组：

```
dojo.require("dijit._Widget");
dojo.addOnLoad(function() {
    dojo.declare(
      "dtdg.Foo",
      dijit._Widget,
```

```
        {
          talk() : function() {
            console.log("id:", this.id);
            console.log("lang:", this.lang);
            console.log("dir:", this.dir);
            console.log("domNode:", this.domNode);
          }
        }
    );
});
foo = new dtdg.Foo();
foo.talk();
```

## 混入 _Templated

_Widget 提供了可以重写的基本方法，用于在部件生命周期的创建和销毁阶段加入自定义操作。_Templated则是前面曾提到过的另一个基类，这个基类主要用于处理在标记中定义的部件模板，而使用替换（substitution）和附着点（attach point）可以为这个基类添加功能。总而言之，有了这个基类，设计人员的任务才能从编写代码中独立出来。

_Templated 的核心任务在于解析和替换模板文件。完成这个任务的关键是重写_Widget的buildRendering方法，该方法用于执行与模板相关的操作。与模板相关的概念主要有以下 3 个：

*替换*

Dijit使用dojo.string模块来替换模板中以${xxx}风格的dojo.string语法表示的字符串。这种替换支持以创建部件时传入的部件属性来自定义模板。

*附着点*

当在模板节点中使用特殊的dojoAttachPoint属性时，就可以通过该属性的值来直接引用模板节点。例如，如果模板中有节点...，那么就可以（在postCreate或之后的方法中）通过this.foo来直接引用该节点。

*事件点*

与附着点相似，可以使用特定的dojoAttachEvent属性在模板节点的DOM事件与部件方法之间建立联系，以便在DOM事件发生时调用相应的部件方法。例如，如果有一个节点...，那么每当该节点上发生单击事件时，都会调用这个部件的foo方法。使用逗号作为分隔符可以定义多个事件。

与介绍_Widget时一样，稍后我们也要展示使用_Templated的示例。而现在先了解一下 _Templated 的相关概念有助于读者从整体上把握其用途。

### 生命周期方法

混入_Templated后最值得注意的效果，就是它会重写_Widget的buildRendering方法。以下是重写buildRendering方法的简要说明。

*buildRendering*

这个方法虽然由_Widget提供，但为处理与取得和实例化部件的模板文件相关的种种细节，以便在屏幕上显示部件，_Templated重写了它。一般来说，开发人员不需要实现自己的buildRendering方法。但是，如果确实要重写该方法，那么最好首先理解_Templated对它的实现。

### 基本的属性

以下是_Templated的基本属性：

templatePath

保存部件模板文件的相对路径，模板文件就是HTML文件。注意，取得部件的模板文件需要使用一次同步网络调用，然后Dojo会将取得的模板字符串缓存起来。第16章将讨论如何使用Util提供的工具为部件生成自定义构建，以便将所有模板字符串内置化（interned）。

templateString

对于已经设计或构建完成，并且相应的模板字符串已经内置于JavaScript文件中的部件而言，这个属性的值表示模板本身。如果同时定义了templatePath和templateString，优先使用templateString。

widgetsInTemplate

如果部件是在模板中被定义的（通过路径或字符串），那么应该将这个属性的值明确设置为true，以便Dojo解析器能够找到并实例化该部件。在默认情况下，这个属性的值为false。将部件包含在其他部件的模板中有时也很有用。向出现在父部件模板中的子部件传递值的一个常用机制，就是通过用于替换的${someWidgetProperty}表示法来实现的。

containerNode

这个值中保存着一个DOM元素的引用，该DOM元素是包含自定义部件的页面中的dojoAttachPoint属性的映射。如果自定义部件作为一个容器包含一组子部件，那么它保存的就是为部件添加新子节点的元素。（作为容器的部件需要额外继承Dijit的_Container类，而被包含的子部件则继承自Dijit的_Contained类。）

# 自定义部件示例：HelloWorld

在前面介绍了那么多理论性的内容之后，想必读者早就希望看到一些实际的代码了。本节中，我们通过一系列相对复杂的“HelloWorld”示例来展示如何设计自定义部件。

我们即将构建的部件名叫HelloWorld，在设计这个自定义部件的过程中，读者将更实际地理解前面讨论的一些问题。虽然本节的示例都围绕构建同一个部件展开，但其中涉及的种种问题却是开发任何部件都必须考虑的。

图12-2展示了HelloWorld部件在磁盘上的目录结构。创建目录没有什么技术性可言；而且这幅图与前面介绍的基本目录结构完全一致。

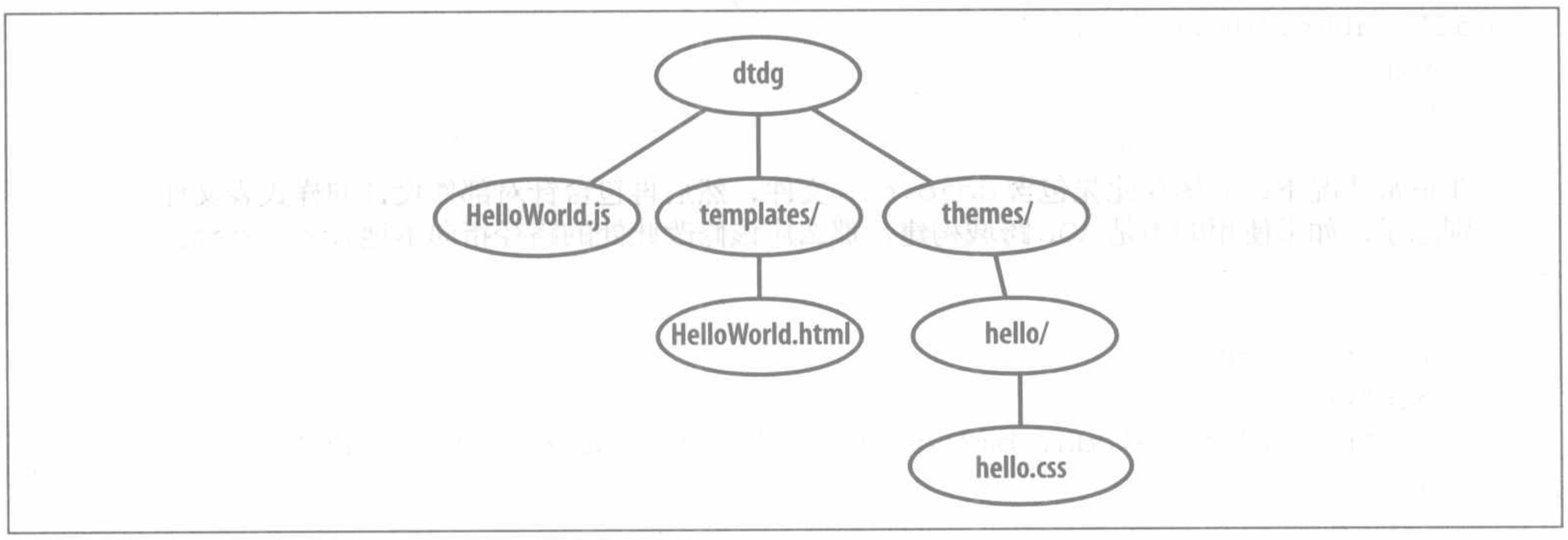



图 12-2：简单的 HelloWorld 部件的目录结构

## HelloWorld 部件（第一步：基本框架）

创建HelloWorld部件的第一步给出了自定义部件涉及的所有组件的完整代码。为了简明起见，后面的各个步骤将只展示被修改组件的相关部分。相对于部件在磁盘上的目录结构而言，我们假设包含这个部件的主 HTML 文件与包含部件代码的 *dtdg* 模块目录是并列的。

### HTML 页面

首先，我们看一看包含这个自定义部件的 HTML 页面，如例 12-2 所示。其中的注释详细说明了页面的各个部分。

例 12-2：HelloWorld 部件（第一步）

```
<html>
 <head>
    <title>Hello World, Take 1</title>
```

```
<!--
    由于Dojo是从AOL服务器上加载的，因此必须在djConfig中指定相应的配置项，
    以便跨域构建（dojo.xd.js）能够起到作用。

    这里，我们通过指定baseUrl和命名空间映射，将“dtdg”命名空间与磁盘上的一个
    特定目录建立关联。如果我们使用的是本地Dojo文件，只要把dtdg目录放在dojo
    目录旁边，Dojo就能自动找到相应部件。

    指定在页面加载后解析页面中的部件，是当页面中包含带有dojoType属性的标签时的
    一种标准做法。

-->
  <script
    type="text/javascript"
    src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
    djConfig=isDebug:true,parseOnLoad:true,baseUrl:'./
',modulePaths:{dtdg:'dtdg'}">
  </script>

<!--
    在正常情况下，应该在此先包含dojo.css文件，然后再包含针对部件设计的样式表文件。
    别忘了，如果使用的不是AOL跨域构建，那么应该修改此处的路径指向本地dojo.css。
-->
<link
    rel="stylesheet"
    type="text/css"
    href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css">
</link>

<link
    rel="stylesheet"
    type="text/css"
    href="dtdg/themes/hello/hello.css">
</link>

<script type="text/javascript">

    dojo.require("dojo.parser");

    // 告诉Dojo取得与dtdg命名空间关联的名为HelloWorld的部件，以便在页面主体
    // 中使用该部件。
    //Dojo会使用djConfig.modulePaths的值查找部件。
    dojo.require("dtdg.HelloWorld");
</script>
</head>
<body>
    <!--
    Dojo解析器基于parseOnLoad:true选项，在此插入dojo.require语句加载的部件。
    应用给部件的所有样式都由前面包含的样式表提供。
    -->
    <div dojoType="dtdg.HelloWorld"></div>
</body>
</html>
```

应该说，这个包含部件的HTML页面绝对是简单得不能再简单了。它首先引入了美化部件的相关样式表，并按惯例加载 Base 以启用 Dojo 工具箱，然后通过 dojo.require 明确请求了解析器和将在页面主体中用到的HelloWorld部件。其中唯一比较陌生的地方就是通过 djConfig.modulePaths 选项将 dtdg 模块映射到其本地磁盘路径。

## CSS

部件的样式由普通的CSS和必需的静态资源（如图像）提供。然而，巧妙的是，部件的样式实际上会映射到部件的模板，而非指定 dojoType 属性的DOM元素。这种设计从根本上保证了部件的换肤能力，或者用Dojo的表达方式，就是可以为部件定义主题并通过切换样式表来更换主题。

在这个示例部件中，虽然为单个DIV元素定义的样式非常简单，但也足以说明如何为自定义部件应用样式了。在 HelloWorld 主题中，只包含一个 CSS 文件，其中也只有一条 CSS 规则：

```
div.hello_class {
    color: #009900;
}
```

## 模板

与样式类似，HelloWorld 部件的 HTML 模板同样也是最小化的。我们只是告诉 Dojo 在取得HTML页面中指定的DIV标签后，要将其替换成模板中定义的元素，在此，模板中包含的只是带有 class 属性和内部文本“Hello World”的另一个 DIV 元素。

实际的模板文件只包含下面的 HTML 标记：

```
<div class="hello_class">Hello World</div>
```

## JavaScript

虽然自定义部件的JavaScript代码看起来有点可怕，但其中大部分内容都有详尽的注释。实际上，这里展示的 JavaScript 只不过是我们曾经介绍过的基本构建而已，而且确实非常简单。下面，我们先从头到尾看一遍这些代码，然后再稍加解释。没错，这个JavaScript文件就是一个标准的 Dojo 模块：

```
// 摘要：用以展示基本 Dojo 部件设计模式的 HelloWorld 部件

// 模块文件的第一行必须是一条 dojo.provide 语句，用以指定资源
// 和与父模块的关系。资源名称必须与 .js 文件名称相同。
dojo.provide("dtdg.HelloWorld");
```

```
// 在使用资源之前必须先请求资源。这里我们请求了两个资源，是
// 因为在部件的继承关系中需要用到它们。
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

// 可以用来声明Dojo“类”的功能强大的构造器。
dojo.declare(
    "dtdg.HelloWorld",

    //dijit._Widget是原型超类，它提供重要的可重写方法。
    //dijit._Templated是混入类，它重写了dijit._Widget的
    //buildRendering方法，该方法基于模板构建部件的UI
    [dijit._Widget, dijit._Templated],
    {
        // 指定部件模板所在路径。dijit._Templated需要使用这个路径
        // 通过同步调用从相应文件中获取模板。
        templatePath: dojo.moduleUrl("dtdg", "templates/HelloWorld.html")
    }
);
```

在继承链当中，_Widget是自定义部件必须继承的原型超类。因为示例的第一步很简单，所以不需要在此重写任何_Widget的生命周期方法；不过，稍后的步骤中将会重写相关方法。混入超类_Templated通过重写_Widget.buildRendering方法为模板加入了新的功能。实际的模板路径保存在自定义部件的templatePath属性中。虽然使用templatePath而不是templateString会导致一次对服务器的同步调用，但在取得模板后，Dojo就会将模板缓存起来。因此，如果页面中需要再次使用HelloWorld部件，那么就不必再执行另一次同步调用了。

---

**注意：** Dojo在第一次获取部件的模板文件时，会执行一次对服务器的同步调用。然后，模板就被缓存起来了。

---

尽管我们这个示例只会在屏幕上显示一条简单的消息，但后台为显示这条消息而执行的操作远不止一条print语句那么简单。而且，构建HelloWorld部件所涉及的各种考虑和技巧，同时也是构建其他部件的最低要求。

为了加深读者对自定义部件的理解，下面我们就来通过重写生命周期方法来增强HelloWorld部件的功能。不过，在此我们并没有直截了当，而是采用了迂回策略。毕竟，这样才是更好的学习方式。

## HelloWorld部件（第二步：修改模板）

假设我们希望自定义部件能够更通用一些。那么，与其在每次页面加载时都显示相同的静态消息，不如对其加以改进让它能够动态显示自定义的消息。为此，就要用到Dojo为

保证 Diji 部件的逻辑完整性而提供的一种机制，这种机制允许我们在模板中引用在 JavaScript 源文件中定义的部件属性。虽然在模板中引用部件属性只在 _Templated 的 buildRendering方法执行以前才有效，但对于在部件显示到屏幕之前初始化其某些部分而言，这种做法仍然是很常用的。

在模板文件中引用部件的属性很简单。下面是修改后的 HelloWorld 部件的模板文件：

```
<div class="hello_class">${greeting}</div>
```

简而言之，就是可以在模板文件的内部引用部件的任何属性，并通过这些属性来控制部件的初始显示、样式等等。但是，这里还有一个不起眼但极为重要的条件：必须在正确的时候来做这件事。通常，在postMixInProperties方法中操作被模板引用的部件属性是最合适的。因为postMixInProperties方法会在buildRendering方法之前被调用，而后者则是部件被插入 DOM 并可见的转折点。

---

**警告：** 操作模板字符串的正确位置在部件的生命周期方法postMixInProperties中，该方法继承自_Widget。如果在这个方法之后再操作模板字符串，那么可能导致令人讨厌的屏幕抖动问题。

---

言归正传，例12-3展示了如果想在模板中引用部件的属性以显示自定义的问候消息，应该如何修改部件的 JavaScript 文件。

例 12-3：HelloWorld 部件（第二步：postMixInProperties）

```
//通过postMixInProperties 方法恰当操作模板字符串中
//引用的部件属性
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
        "dtdg.HelloWorld",
        [dijit._Widget, dijit._Templated],
    {
        greeting : "",

        templatePath: dojo.moduleUrl(
            "dtdg",
            "templates/HelloWorld.html"
        ),

        postMixInProperties: function() {
            //恰当地操作在模板中引用的属性
            this.greeting = "Hello World"; //提供一条静态的问候消息
```

```
    }
  }
);
```

## HelloWorld 部件（第三步：内置模板）

正如前面所提到的，如果在JavaScript文件中直接指定模板字符串，那么可以节省一次到服务器的同步调用。例12-4中修改后HelloWorld部件的JavaScript文件展示了手动指定模板字符串有多么简单。不过，请读者注意，Util中提供的Dojo构建脚本会在部署例程中自动处理所有自定义部件模板的内置操作。

例12-4：HelloWorld（第三步：templateString）

```
dojo.provide("dtdg.HelloWorld");
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "dtdg.HelloWorld",
    [dijit._Widget, dijit._Templated],
    {
        greeting : "",

        //像下面这样在JavaScript文件内部指定模板字符串……
        templateString : "<div class='hello_class'>${greeting}</div>",

        postMixInProperties: function() {
            console.log ("postMixInProperties");

            //仍然可以像往常一样操作模板字符串
            this.greeting = "Hello World";
        }
    }
);
```

在这个示例中，我们通过templateString在JavaScript文件内部定义模板，因此就不需要单独的模板文件了。而且，这样也意味着节省了一次对服务器的同步调用。想像一下，大量的部件需要大量的模板字符串，如果把这些模板字符串都内置于部件的JavaScript文件中，那么一定会明显减少加载页面的时间。在现实的项目开发中，可以通过Util中的构建系统（参见第16章）来自动完成这种性能优化任务。

## HelloWorld 部件（第四步：传入参数）

接下来，我们再对HelloWorld部件做一次改进，同时介绍如何通过模板向部件中传递自定义参数。在前面示例的基础上，假设我们想要在包含dojoType属性的标记中提供一条部件显示的自定义问候消息。不难，只要像下面这样传入问候消息即可：

```
<div dojoType="dtdg.HelloWorld" greeting="Hello World"></div>
```

同样，在以编程方式创建部件时传入参数也很简单：

```
var hw = new dtdg.HelloWorld({greeting :  "Hello World"}, theWidgetsDomNode);
```

当然，可以传入的参数值并不局限于模板中引用的部件属性。传入的参数也可以用于其他目的。例如，下面的DIV元素中引用了HelloWorld部件，同时还指定了两个键/值对参数：

```
<div foo="bar" baz="quux" dojoType="dtdg.HelloWorld"></div>
```

能够像这样为部件传入自定义数据以初始化部件，对于应用级的开发人员而言，是不是就不必查看源代码，而只要设计好模板就可以呢？好的，女士们、先生们，完全没有问题，例 12-5 中的 JavaScript 文件就为我们展示了如何实现这一点。

例 12-5：HelloWorld 部件（第四步：自定义参数）

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
        "dtdg.HelloWorld",
        [dijit._Widget, dijit._Templated],
    {
        templateString : "<div class='hello_class'>Hello World</div>",

        foo : "",

        //不能设置不存在的部件属性
        //baz : "",
        //只有部件中定义了相应的属性，在包含 dojoType 属性的标签中指定的属性
        //才能作为参数被传入构造函数中。换句话说，由于这个部件没有定义名为 baz
        //的属性，因此 baz="quux" 不会产生任何影响
        constructor: function() {

            console.log("constructor: foo=" , this.foo);
            console.log("constructor: baz=" , this.baz);
        }
    }
);
```

读者大概已经注意到了，代码的注释中强调只能为存在的部件属性传入值；在包含 dojoType 属性的标签中随意添加的属性，并不会导致创建新的部件属性。如果读者运行前面的代码示例，就会在 Firebug 控制台中看到下面的输出结果：

```
constructor: foo=bar
constructor: baz=undefined
```

虽然为部件传入字符串值很方便，但只传入字符串值却不能满足全部需求。不过，读者不必担心：Dojo也允许我们为部件传入列表和关联数组。为此，我们要做的就是在JavaScript文件中定义适当类型的部件属性，其他事情Dojo就可以帮我们处理了。

下面的示例展示了如何通过模板为部件传入列表和关联数组参数。

首先，在包含dojoType属性的元素中加入相应的参数：

```
<div
    foo="[0,20,40]"
    bar="[60,80,100]"
    baz="{'a':'b', 'c':'d'}"
    dojoType="dtdg.HelloWorld"
></div>
```

然后，在JavaScript文件中定义适当类型的属性（也很直观）：

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
        "dtdg.HelloWorld",

        [dijit._Widget, dijit._Templated],

    {
        templateString : "<div class='hello_class'>Hello World</div>",

        foo : [], //定义数组属性
        bar : "", //定义字符串属性
        baz : {}, //定义对象属性
         postMixInProperties: function() {
            console.log("postMixInProperties: foo[1]=" , this.foo[1]);
            console.log("postMixInProperties: bar[1]=" , this.bar[1]);
            console.log("postMixInProperties: baz['a']=",  this.baz['a']);
         }
    }
);
```

以下是Firebug控制台中显示的输出结果：

```
postMixInProperties: foo[1]=20
postMixInProperties: bar[1]=6
postMixInProperties: baz['a']=b
```

我们注意到，虽然在包含模板的页面中为部件的bar属性传递的是一个列表值，但在JavaScript文件中，这个属性则被声明为一个字符串值。因此，Dojo会把传入的参数当

作一个字符串来对待，最终的结果是返回被剪切后的字符串。一般而言，解析器会将传入的参数值按照鸭子类型检测的标准解析为相应的类型。

**警告：** 不能在 JavaScript 文件中错误地定义属性类型。否则，将会浪费宝贵的调试时间！

## HelloWorld 部件（第五步：关联事件）

在前面创建的 HelloWorld 部件的基础上，可以进一步考虑将鼠标单击或鼠标悬停等 DOM 事件与部件建立关联。Dojo 可以方便地在部件与事件之间建立关联，方法就是在模板的 dojoAttachEvent 属性中指定 DOMEvent：dijitMethod（DOM 事件:部件方法）形式的键/值对。要指定多个键值对或者说要关联多种本地 DOM 事件，可以使用逗号作为它们的分隔符。

为了示范 dojoAttachEvent 的用法，我们可以先在样式表中定义一个样式类，然后在 mouseover 事件发生时为部件添加这个类，而在 mouseout 事件发生时移除这个类。由于 DIV 元素默认与浏览器窗口同宽，因此最好将它改为行内的 SPAN 元素，这样就能保证只有鼠标直接位于文本之上时才会触发鼠标事件。在此，我们为 SPAN 元素应用指针（cursor）样式。

修改样式表中的规则很简单，先把 DIV 元素替换成 SPAN 元素，再把鼠标指针样式修改为 pointer：

```
span.hello_class {
    cursor: pointer;
    color: #009900;
}
```

例 12-6 中的 JavaScript 文件中包含了修改后的模板字符串，其中添加的 dojoAttachEvent 属性同样也很直观。

例 12-6：HelloWorld（第五步：dojoAttachEvent）

```
dojo.provide("dtdg.HelloWorld");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
        "dtdg.HelloWorld",
        [dijit._Widget, dijit._Templated],
    {
        templateString :
```

```
    "<span class='hello_class' dojoAttachEvent='onmouseover:onMouseOver, onmouseout:
    onMouseOut'>Hello World</span>",

    onMouseOver : function(evt) {
        dojo.addClass(this.domNode, 'hello_class');
        console.log("applied hello_class...");
        console.log(evt);
    },

    onMouseOut : function(evt) {
        dojo.removeClass(this.domNode, 'hello_class');
        console.log("removed hello_class...");
        console.log(evt);
    }
);
```

还不够简单吗？这样，当在SPAN元素的文本上触发onmouseover事件时，Base提供的dojo.addClass方法就会为部件应用样式。然后，当再次触发onmouseout事件时，样式又会被移除。真棒！

另外，读者是否注意到部件的事件处理方法中还接收了一个evt参数？这个参数中包含着与事件相关的详细信息。不难猜到的是，Dojo在内部使用了dojo.connect方法在此为我们标准化了事件对象。以下是运行这个示例后在Firebug控制台中显示的输出结果，从中也可以看到传入部件事件处理程序中的事件信息：

```
applied hello_class...
mouseover  clientX=64, clientY=11
removed hello_class clientX=65, clientY=16
mouseover  clientX=65, clientY=16
```

---

**警告：** 不要把本地DOM事件的名称拼错，同时还要保证其名称中的所有字母都采用小写形式。例如，ONMOUSEOVER或onMouseOver都不能表示onmouseover事件。而且，Firebug也不会对这种错误给出任何提示。由于可以随意命名部件的事件处理方法（可以采用任何大小写形式），因此在拼写DOM事件时很可能会受其影响而忘记这一要求。

---

显而易见的是，前面示例中的onmouseover映射到了onMouseOver，而onmouseout映射到了onMouseOut。这种对应关系的约定极其简单，而得到的代码也更容易一目了然。同样，应该注意的是，onmouseover和onmouseout是DOM事件，而onMouseOver和onMouseOut则是与特定部件关联的方法。由于它们名称的英文发音相同，因此不太容易一下子就搞清楚它们的区别。但是，DOM事件与部件方法之间的区别是学习使用Dijit过程中必须掌握的。虽然它们的语义类似，但在很多方面仍然存在不同之处。

## _Container 和 _Contained 与父子关系

在使用_Widget和_Templated一段时间之后，读者很快就会发现在部件中包含子部件是一种方便的做法。编程中极为常见的“有一个”的关系模式，在Dojo中也没有什么不同。为此，Dojo提供了_Container和_Contained这两个混入类，以便于父部件与子部件之间的相互引用。表12-1总结了相关的API。

表12-1：_Container和_Contained混入类

| 名称 | 说明 |
| --- | --- |
| removeChild(/*Object*/dijit) | 从父部件中移除子部件（如果传入的部件不是子部件或者容器中根本就不包含子部件，这个方法会静默地失败。） |
| addChild(/*Object*/ dijit, /*Integer?*/ insertIndex) | 将子部件添加到父部件中，可以使用insertIndex指定子部件的位置 |
| getParent() | 子部件使用这个方法可以引用其父部件。返回一个部件的实例 |
| getChildren() | 父部件使用这个方法可以方便地枚举其子部件。返回一个部件实例的数组 |
| getPreviousSibling() | 子部件使用这个方法可以引用其前面的同辈部件，也就是其“左边的”那个部件。返回一个部件的实例 |
| getNextSibling() | 子部件使用这个方法可以引用其后面的同辈部件，也就是其“右边的”那个部件。返回一个部件的实例 |

在本书后面讨论布局部件时，读者将会看到对这两个混入类的广泛应用。接下来，我们再看一个示例。

## 在标记中快速构建部件

现在，读者对部件的生命周期已经有了足够的了解，也研究了不少示例。下面，我们再介绍一种能够快捷、方便地创建部件的方法。这种方法就是使用Declaration这个Dijit资源，不用单独定义JavaScript文件就在标记中声明部件。掌握了这种方法，对于在实际开发中迅速捕获和验证一种思路是极为有用的。

例12-7展示了如何使用Declaration在一个完全自包含的页面中创建最初的HelloWorld的部件。

例 12-7：HelloWorld 部件（第六步：Declaration）

```
<html>
    <head>
        <title>Hello World, Take 6</title>

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="isDebug:true,parseOnLoad:true">
        </script>

        <link
            rel="stylesheet"
            type="text/css"
            href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css">
        </link>

        <!-- 定义嵌入的CSS规则 -->
        <style type="text/css">
            span.hello_class {
                color: #009900;
                cursor: pointer;
            }
        </style>

        <script type="text/javascript">
            dojo.require("dijit.Declaration");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body>

        <!-- 完全在标记中声明自定义部件 -->
        <div
        dojoType="dijit.Declaration"
        widgetClass="dtdg.HelloWorld"
        defaults="{greeting:'Hello World'}">
            <span class="hello_class"
            dojoAttachEvent='onmouseover:onMouseOver, onmouseout:onMouseOut'>
                ${greeting}
            </span>

            <script type="dojo/method" event="onMouseOver" args="evt">
                dojo.addClass(this.domNode, 'hello_class');
                console.log("applied hello_class...");
                console.log(evt);
            </script>

            <script type="dojo/method" event="onMouseOut" args="evt">
                dojo.removeClass(this.domNode, 'hello_class');
                console.log("removed hello_class...");
```

```
                console.log(evt);
            </script>
        </div>

        <!-- 接下来，像往常一样在页面中包含这个部件 -->
        <div dojoType="dtdg.HelloWorld"></div>
    </body>
</html>
```

通过这个示例，相信读者一定会觉得Declaration在不依赖任何资源的条件下迅速创建部件的能力的确非同凡响。既不必切换和维护多个文件，也无须声明模块路径，总之不用处理与核心任务无关的事情。因此，才能把精力集中于手头的任务上，并尽可能高效地解决问题。表 12-2 列出了 Declaration 的 API。

表 12-2：Declaration 的属性

| 属性 | 说明 |
| --- | --- |
| widgetClass | 部件的类 |
| defaults | 在创建部件时以参数形式传入的属性值 |
| mixins | 定义混入超类的数组 |

**警告：** 在标记中声明的Declaration的mixins属性必须是一个数组。这一点与dojo.declare方法不同，dojo.declare既可以接收对象超类，也可以接收对象超类的数组。

通常，当某个创建部件的思路定型之后，都需要对以上临时性的工作进行重构。但是，要想在短时间内就迅速构建一个部件的原型，也确实没有比使用Declaration更快的方式了。

# 小结

在学习了本章之后，读者应该：

- 能够讲清楚 Dojo 部件如何把 HTML、CSS 和 JavaScript 封装到一个独立、可移植的代码单元中。
- 理解 _Widget 以可重写方法形式提供的关键生命周期事件，以及这些可重写方法的执行顺序。
- 理解_Templated怎样充当_Widget的混入超类，并通过为部件添加模板支持而增强 _Widget 的功能。

- 理解在创建部件模板时，使用templatePath和templateString的差别和优缺点。
- 能够在部件显示在屏幕上之前，恰当地操作部件的模板。
- 能够通过模板向部件中传入参数。
- 能够以编程方式创建并在页面中插入部件。
- 知道如何在部件中加入对onmouseover等DOM事件的支持。
- 能够使用Declaration迅速地在标记中创建部件原型。

下一章，我们将讨论表单部件。

# 第13章

# 表单部件

本章将系统地介绍表单部件，使用表单部件只需少量编码，就能创建出美观新颖的表单。与Dijit包含的其他部件一样，表单部件也完全可以在标记中定义、只需编写很少的JavaScript代码，并且设计时就融合了易访问性的理念 。应该说，由于dijit.form的功能极为强大，涉及表单的方方面面，因此本章的内容相对比较多一些。而且，表单部件也是工具箱中迄今为止最能体现面向对象特性的部件。通过对本章的学习，读者会发现在本书其他章节中看不到的基于dojo.declare实现的深层继承关系。

## 表单的简单回顾

尽管HTML4.01规范（*http://www.w3.org/TR/html401/*）对表单做出了权威的描述，而且很值得去研究，但是为了让读者能够更好地理解本章的内容，本节仍然会重申一些有关表单的有用信息。而且，我们首先会假设读者在阅读本章之前至少曾设计过一两个表单，因此就不必赘述表单是一组用于收集用户信息的控件集合，并负责把收集到的信息传送回服务器以供处理这一事实了。

不过，值得一提的是，Ajax确实导致了向服务器传送待处理数据的模式发生了变革。在Ajax出现之前，表单提交给服务器处理程序的数据会被组织成简单的键/值对形式，当服务器处理完这种散列形式的数据之后，会返回给浏览器一个新页面，该页面通常会反映出用户在表单中所做的选择。为了让整个过程更优雅一些，人们甚至把表单包含在一个iframe中，从而将页面重载的范围降至最低限度。现在，有了XMLHttpRequest（XHR）对象，使用这个对象能够不通过表单提交或者无须任何页面重载，就可以向服务器异步发送小数据块。

当然，XHR 对象，或者说 Ajax，以及由此衍生的与用户交互的流畅方式，并不会导致表单被废弃不用。表单依旧是一种久经考验的标准数据传输手段；它能够在 JavaScript 无效时使用，因而对易访问性的实现至关重要。一般来说，为任何向服务器传输数据的新潮方式都准备一套可退化（degradable）和易访问的备用方案始终都是有必要的。换句话说，“表单或者 Ajax”不是最佳方案，最佳方案应该是“表单和 Ajax”。

以例 13-1 为例，其中展示了对第 1 章中那个简单表单的增强版。

例 13-1：简单的表单

```
<html>
    <head>
        <title>Register for Spam</title>
        <script type="text/javascript">
            function help() {
                var msg="Basically, we want to sell your info to a 3rd party.";
                alert(msg);
                return false;
            }

            //简单的验证函数
            function validate() {
                var f = document.getElementById("registration_form");
                if (f.first.value == "" || f.last.value == "" || f.email.value == "") {
                        alert("All fields are required.");
                        return false;
                 }

                return true;
            }
        </script>
    <head>
    <body>
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form id="registration_form"
            method="POST"
            onsubmit="javascript:return validate()"
            action="http://localhost:8080/register/">

            First Name: <input type="text"  name="first"/><br>
            Last Name: <input type="text" name="last"/><br>
            Your Email: <input type="text" name="email"/><br>
            <button type="submit">Sign Up!</button>
            <button type="reset">Reset</button>
            <button type="button" onclick="javascript:help()">Help</button>

        </form>
    </body>
</html>
```

这个表单虽然没有什么令人惊叹之处，但它确实具备一定的功能，而且可以在任何浏览器中正常使用。如果再辅之以一份精致的CSS样式表，那么它的外观也一定不落俗套。此外，页面中甚至还提供了一个Help按钮，用以提示用户表单的用途。在服务器端，应该有一个简单的脚本负责处理这个表单，基本上是在Web服务器从表单提交的原始数据中提取出命名字段之后。例13-2就展示了一个能够处理这个表单的CherryPy脚本。

例13-2：用于处理表单的CherryPy脚本

```
import cherrypy
class Content:
    """
    用于处理表单提交的例程。    命名的表单字段容易访问。    """
    @cherrypy.expose
    def register(self, first=None, last=None, email=None):

        # 这里是将用户信息添加到数据库中的代码……

        # 返回下面这个定制的HTML页面
        return """
        <html>
            <head><title>You're now on our spam list!</title></head>
            <body>
                <p>Congratulations %s %s, you're gonna get spammed!</p>
            </body>
        </html>
        """ % (first, last) #substitute in variables

cherrypy.quickstart(Content())
```

虽然非常简单，但以上示例也涉及了与表单有关的几个关键点：

- 表单控制应该被包含在一个FORM标签内。
- FORM标签几乎总要包含name、method、onsubmit、enctype、action等属性，这些属性指定了应该如何处理表单的信息。
- onsubmit属性是执行客户端验证的标准方式。在验证程序中返回false将会阻止表单把数据提交到服务器。
- action属性指定的是提交表单的URL。
- 表单中有意义的字段应该包含name属性，多数服务器端框架都根据这个属性来组织键/值对数据，然后再把数据传递给处理表单提交的程序。
- 表单天生可以通过键盘访问；按Tab键可以切换表单字段，而按回车键则可以提交表单。尽管没有在这个示例中出现，但tabindex属性可以改变默认的切换顺序。

- 通常，表单会包含多种控件，控件的类型由其type属性指定。这个示例展示了3种不同的按钮，分别用于触发onsubmit事件、重置表单和处理自定义操作。
- 提交表单必须重载页面，在指定action属性的前提下，重载的页面由服务器返回。如果没有指定action属性，那么通过将脚本指定给onclick等DOM事件可以执行自定义JavaScript或DHTML操作。

---

**注意：** 在本章中，术语“属性”既用于描述表单属性，也用于描述对象属性。当它们在各自相关的语境中出现时，意思都是很明确的，相信读者不会因此而混淆。

---

虽然远远不够全面，但我们希望本节这个简短的回顾能为后面讨论各种表单部件打下良好的基础。如果读者想找一本详细介绍HTML表单的参考书，可以考虑由Chuck Musciano和Bill Kennedy合著的《HTML & XHTML: The Definitive Guide》(O'Reilly)。

## 表单部件

适合在真实的HTML表单中使用的表单部件（就像使用FORM标签定义的一样），包含在dijit.form命名空间内。本节将介绍这个命名空间包含的所有部件，并提供相应的示例代码以及应用Dijit内置的*tundra*主题的屏幕截图。不过，有必要首先重申的一点是，所有表单部件都是按照可退化及易访问的标准进行设计的；换句话说，即使JavaScript、CSS和图像都无法使用，这些表单控件仍然具备完整的功能，而且，还会支持仅通过键盘对表单的操作。另外，在Windows环境下，易访问性的功能还包括支持高对比度模式及屏幕阅读器。

图13-1展示了dijit.form模块的总体继承结构。这幅示意图中没有包含所有独立的混入类，但明确传达了部件之间的相互关系。相信读者在熟悉了表单部件之后，通过观察这幅图就能够想像出每个部件源代码的布局。

除了继承自_Widget的标准部件属性（如domNode等）和HTML4.01规范中规定的普通HTML属性（例如，disabled和tabIndex）之外，表单部件都继承了一个基类——_FormWidget，这个基类支持表13-1中列出的属性、方法和扩展点。

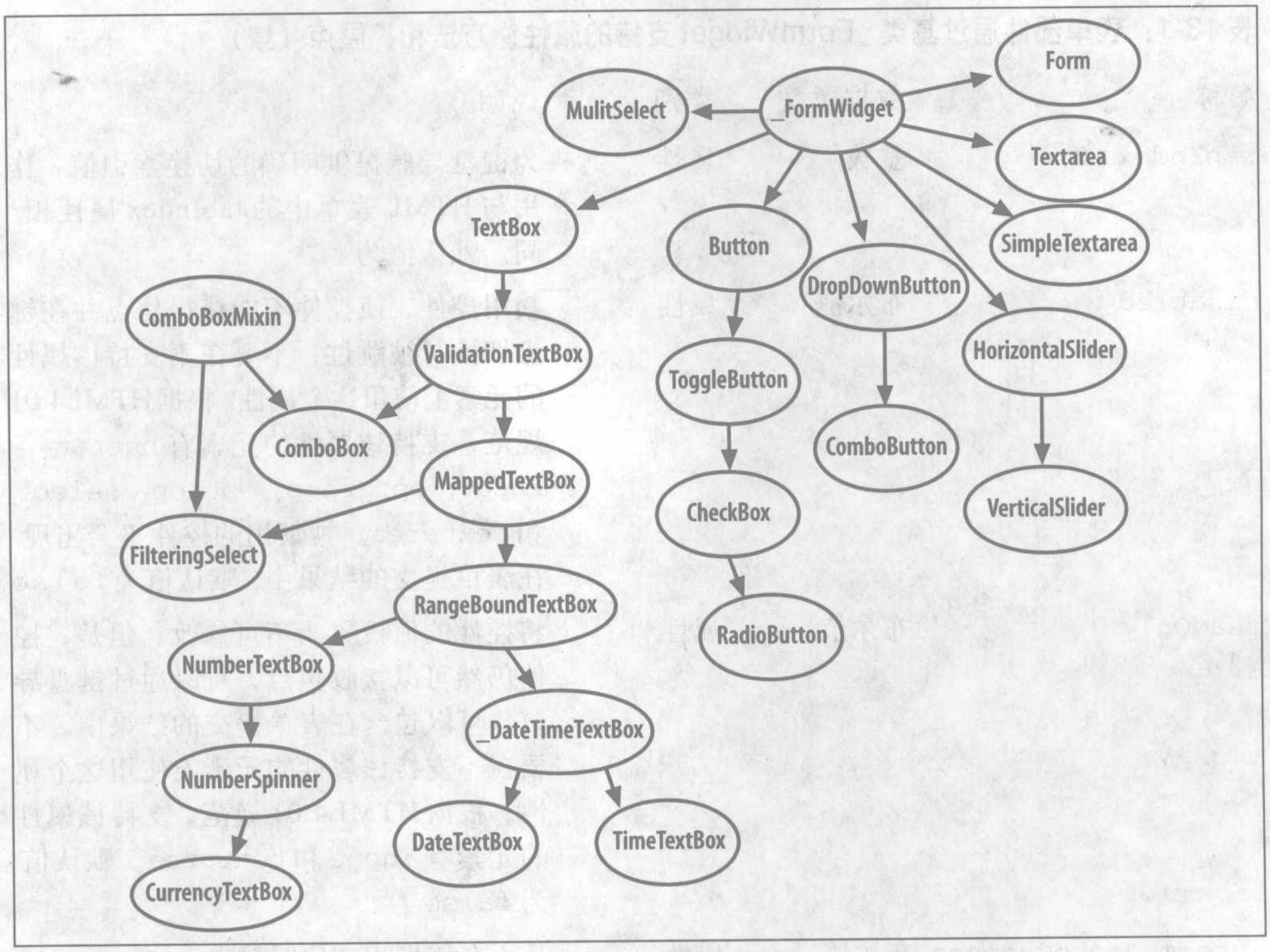



图 13-1：包含丰富继承关系的 dijit.form 模块

表 13-1：表单部件通过基类 _FormWidget 支持的属性、方法和扩展点

| 名称 | 数据类型 | 类别 | 说明 |
|---|---|---|---|
| value | 字符串 | 属性 | 部件的当前值；作用与HTML表单中的value属性相同 |
| name | 字符串 | 属性 | 部件值的名称；作用与HTML表单中的name属性相同，供服务器端程序处理表单提交使用 |
| alt | 字符串 | 属性 | 当浏览器无法显示该部件时显示的替代文本（虽然对表单而言这种情况很少发生，但对图像而言并不罕见）；作用与HTML表单中的alt属性相同 |
| type | 字符串 | 属性 | 在有多种可能的类型时指定元素类型。例如，type="submit"的按钮用于触发表单的onsubmit操作；作用与HTML表单中的type属性相同。默认值为"text" |

表13-1：表单部件通过基类_FormWidget支持的属性、方法和扩展点（续）

| 名称 | 数据类型 | 类别 | 说明 |
| --- | --- | --- | --- |
| tabIndex | 整数 | 属性 | 为键盘导航提供明确的切换索引值；作用与HTML表单中的tabIndex属性相同。默认值为"0" |
| disabled | 布尔值 | 属性 | 禁用控件，使控件不能获得焦点并在键盘切换时被跳过；不要在不支持该属性的元素上使用这个属性。根据HTML4.01规范，支持该属性的元素有button、input、optgroup、option、select和textarea。被禁用的控件不会出现在表单提交的结果中。默认值为false |
| readOnly | 布尔值 | 属性 | 将控件的值设置为不可修改；但是，控件仍然可以接收焦点，可以通过键盘导航，可以包含在表单提交的结果中。不要在不支持该属性的元素上使用这个属性。根据HTML4.01规范，支持该属性的元素有input和textarea。默认值为false |
| intermediateChanges | 布尔值 | 属性 | 是否在值发生变化时触发onChange扩展点。默认值为false |
| setAttribute (/* String */ attr, /* Any */ value) | 函数 | 方法 | 设置部件属性值的正确方式。例如，要把一个部件的value属性设置为"foo"，要通过部件调用这个方法：<部件名称>.setAttribute("value", "foo") |
| focus() | 函数 | 方法 | 设置控件的焦点 |
| isFocusable() | 函数 | 方法 | 返回控件是否可以接收焦点的信息 |
| forWaiValuenow() | 函数 | 扩展点 | 默认返回用作WAI-ARIA的valuenow状态的当前部件状态，该状态通过dijit.removeState和dijit.setWaiState设置 |
| onChange(/* Any */ val) | 函数 | 扩展点 | 重写这个扩展点以定义值每次被修改时调用的回调函数 |

**注意**：要全面了解HTML4.01规范，请参考*http://www.w3.org/TR/html401/interact/forms.html*。

# TextBox 及其变体

HTML 的 input 元素是用于文本输入的最常用的表单字段。正是这些字段中并不太长的文本内容，往往会让人花上几个小时的时间去修正其格式，验证其有效性；而且，用于支持输入框的辅助性脚本数量经常会在支持网页的脚本集合中独占鳌头。如果有读者对这番评论深有感触的话，那么 Dijit 的 TextBox 系列部件肯定会带给你意外的惊喜。

接下来，我们将分别介绍 TextBox 家族中的每一位成员，并改进本章前面展示的示例表单。TextBox 家族中最基本的成员就是 TextBox 本身，它封装了一些自定义的格式化操作，同时也通过 format 和 parse 扩展点为开发人员提供了创建自定义行为的能力。TextBox 提供的属性和扩展点如表13-2所示。从技术上讲，TextBox 是一种 input 元素，因此虽然表 13-2 中没有列出 input 元素的标准 HTML 属性，但那些属性对 TextBox 仍然适用。

---

**注意：** TextBox 的所有变体（家族成员）都继承了它的属性和扩展点；由于这些属性和扩展点应用十分广泛，因此明确这一点非常重要。

---

## TextBox

表 13-2 列出了最基本的 TextBox 部件的相关特性。

**表 13-2：TextBox 的属性和扩展点**

| 名称 | 类别 | 说明 |
|---|---|---|
| trim | 属性 | 删除文本前后的空格。默认值为 false |
| uppercase | 属性 | 把所有字母转换为大写形式。默认值为 false |
| lowercase | 属性 | 把所有字母转换为小写形式。默认值为 false |
| propercase | 属性 | 把每个词的第一个字母转换为大写形式。默认值为 false |
| maxLength | 属性 | 用于传入标准 HTML input 标签的 maxlength 属性。默认值为 "" |
| format(/* String */ value, /*Object*/constraints) | 扩展点 | 一个可重写的方法，用于将输入的值转换为具有适当格式的字符串值。默认实现是返回调用输入值的 toString 方法的结果；如果输入值没有 toString 方法，那么返回输入的原始值。对于 null 或 undefined 值返回空字符串 |

表13-2：TextBox的属性和扩展点（续）

| 名称 | 类别 | 说明 |
| --- | --- | --- |
| parse(/* String */ value) | 扩展点 | 可以用来定义一个解析函数，以便将格式化的字符串转换为一个值，然后再返回该值。这个函数对所有表单部件都很常用。默认实现是返回原始的字符串值 |
| setValue(/*String*/value) | 方法 | 用于设置TextBox及其子类的字符串值。不能在此处的子类层次中使用_FormWidget的setAttribute('value', /*-*/)方法 |
| getValue() | 方法 | 用于取得TextBox及其子类的字符串值。不能直接访问部件的value属性，应该使用这个方法间接取得 |

**警告：** 从Dojo1.1开始，不再推荐使用_FormWidget的setValue和getValue方法，而是建议在适合通过.value设置和取得属性的情况下使用setAttribute('value', /*...*/)方法。但是，TextBox及其子类，还有其他一些部件重写了setValue和getValue方法，并可以正常使用。根据经验，一般只针对部件值使用setValue和getValue方法。例如，TextBox明显有一个值（因此，可以使用setValue和getValue方法）。但是，对Button这样的部件而言，尽管它也有一个随表单提交的值，但它却没有部件值，因此应该对它使用setAttribute方法。

为了示范这些属性和方法的基本应用，例13-3在前面表单示例的基础上插入了几个文本框，并为表单中的first和last字段开启了propercase和trim属性

例13-3：通过TextBox和主题重建的表单

```
<html>
    <head>
        <title>Register for Spam</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
```

```
        dojo.require("dojo.parser");
        dojo.require("dijit.form.TextBox");

        function help() {
            var msg="Basically, we want to sell your info to a 3rd party.";
            alert(msg);
            return false;
        }

        //简单的验证函数
        function validate() {
            var f = document.getElementById("registration_form");

            if (f.first.value == "" ||
                f.last.value == "" ||
                f.email.value == "") {
                    alert("All fields are required.");
                    return false;
                }

                return true;
            }
        </script>
    <head>
    <body class="tundra">
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form id="registration_form"
            method="POST"
            onsubmit="javascript:return validate()"
            action="http://localhost:8080/register/">

            First Name:
              <input dojoType="dijit.form.TextBox" propercase=true
               trim=true name="first"><br>
            Last Name:
              <input dojoType="dijit.form.TextBox" propercase=true
               trim=true name="last"><br>
              Your Email:
            <input dojoType="dijit.form.TextBox" length=25 name="email"><br>
              <button type="submit">Sign Up!</button>
              <button type="reset">Reset</button>
              <button type="button" onclick="javascript:help(  )">Help</button>
        </form>
    </body>
</html>
```

---

**警告：** 在使用部件时，如果没有正确地包含*dojo.css*文件及相关主题资源，部件仍然是可以使用的——不过，看起来恐怕不会太美观。刚开始使用Dijit的开发人员经常会忘记加载CSS文件，或者忘记为BODY标记设置适当的class属性。

---

使用TextBox部件除了能够改善控件的外观，还能节省开发人员编写脚本的工作量。例如，通过重写format扩展点，开发人员可以根据需要实现自定义的格式化逻辑。方法就是定义一个JavaScript函数，然后把该函数指定给format扩展点。下面这个格式化函数接收一个字符串，并将这个字符串转换成*MiXeD CaPiTaLiZaTiOn*格式：

```
function mixedCapitalization(value) {
    var newValue = "";
    var upper = true;

    dojo.forEach(value.toLowerCase(), function(x) {
        if (upper)
            newValue += x.toUpperCase();
        else
            newValue += x;

        upper = !upper;
    });

    return newValue;
}
```

在TextBox部件中使用这个函数当然非常简单：

```
<input dojoType="dijit.form.TextBox" format="mixedCapitalization"
    trim=true name="first">
```

这样，当焦点从上面这个文本框中转移时，文本框的值就会被这个函数所转换。同重写format一样，也可以以相同方式重写parse函数，以便对返回的值进行标准化处理。常见的操作包括将数字类型的值转换成Number值，或者标准的string值。

---

**注意：** 自定义的format和parse扩展点在每次调用setValue或getValue时也会执行，并不仅仅在响应用户与表单的交互时执行。

---

## ValidationTextBox

现在，读者脑海中可能会浮现出编写讨厌的验证函数时的情景。验证函数不仅要验证字段非空，还要正确地验证电子邮件地址。但是，这样的函数很难一下子就很完善。好在，我们可以使用ValidationTextBox了！

表13-3完整地列出了ValidationTextBox提供的所有功能。

表 13-3：ValidationTextBox 的属性

| 名称 | 类别 | 说明 |
| --- | --- | --- |
| required | 布尔值 | 属性，用于确定当前字段是否为必填字段。如果将这个属性设置为 true，那么留空当前字段就无法通过验证。默认值为 false |
| promptMessage | 字符串 | 属性，用于定义在当前字段获得焦点时显示的提示消息 |
| invalidMessage | 字符串 | 属性，用于定义在当前字段未通过验证时显示的提示消息 |
| constraints | 对象 | 属性，用于定义一个包含约束条件的对象，该对象可以（如有必要动态地）为 regExpGen 属性提供约束条件。这个属性在 DateTextBox 等部件中被广泛应用于为显示需要提供自定义格式 |
| regExp | 字符串 | 属性，提供用于验证的正则表达式。如果指定了 regExpGen 属性，就不要再指定这个属性。默认值为 ".*"（匹配任何值的正则表达式） |
| regExpGen | 函数 | 属性，表示一个用户可重写的函数，该函数用于根据constraints属性提供的键/值对生成自定义的正则表达式；用于动态定义验证条件。如果指定了 regExp 属性，就不要再指定这个属性。默认情况下，这个函数返回 ".*"（匹配任何值的正则表达式） |
| tooltipPosition | 数组 | 属性，用于定义提示条出现在相对于控件的什么方位，可用作数组元素的值包括：above（上）、below（下）、before（前）、after（后）。默认返回dijit.Tooltip.defaultPosition的值，该值在 dijit.Tooltip 部件内部定义 |
| isValid() | 函数 | 方法，用于调用validator扩展点以执行验证，返回布尔值 |
| validator(/* String */ value,/* Object */ constraints) | 函数 | 扩展点，当onblur、oninit和onkeypress等DOM事件发生时被调用 |
| displayMessage (/* String */ message) | 函数 | 扩展点，对它进行扩展可以自定义验证错误或提示。默认使用 dijit.Tooltip |

**注意：** dijit.Tooltip部件将在第15章介绍。

将前面示例中的TextBox都替换成ValiationTextBox，以及为它们指定required属性并添加用于验证电子邮件地址的正则表达式非常直观明了。例13-4采用了ValiationTextBox部件，并去掉了前面编写的JavaScript代码；现在，Help按钮被替换成了能达到同样目的但更优雅的提示条。

例13-4：使用ValiationTextBox重建表单

```
<html>
    <head>
        <title>Register for Spam</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true"
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.ValidationTextBox");
        </script>

        <!-- 原先嵌入在此处的JavaScript代码被删除了 -->

    <head>
    <body class="tundra">
        <p>Just Use the form below to sign-up for our great offers:</p>
        <form  id="registration_form"
            method="POST"
            action="http://localhost:8080/register/">

            First Name:
            <input dojoType="dijit.form.ValidationTextBox"
              properCase="true" trim=true required="true"
              invalidMessage="Required." name="first"><br>

            Last Name:
            <input dojoType="dijit.form.ValidationTextBox"
              properCase="true" trim=true required="true"
              invalidMessage="Required." name="last"><br>

            Your Email:
            <input dojoType="dijit.form.ValidationTextBox"
```

```
                promptMessage="Basically, we want to sell your info to a 3rd party."
                regExp="[a-z0-9._%+-]+@[a-z0-9-]+\.[a-z]{2,4}"  required
                name="email"><br>
            <button type="submit">Sign Up!</button>
            <button type="reset">Reset</button>
            <!-- 通过提示条替代了帮助按钮 -->

        </form>
    </body>
</html>
```

少量的投入换来的却是功能丰富、外观精美的表单，同时需要维护的代码量也大为减少。

下一节，我们还需要对表单中的按钮再做些调整；不过，接下来还是继续把TextBox家族的成员先介绍完吧。

## MappedTextBox 和 RangeBoundTextBox

本章尚未提及的表单部件还有定义明确的MappedTextBox和RangeBoundTextBox。基本上，MappedTextBox提供了一些方法，用于通过自定义的toString方法将其数据串行化为一个字符串值。而RangeBoundTextBox可以保证接收的值位于传递给constraints对象的max和min值之间。虽然从表面上看，限定取值范围似乎应该是ValidationTextBox进行范围检查的一项“验证”任务，但实际上，ValidationTextBox适用于通过正则表达式验证字符串值，而RangeBoundTextBox则明确用于处理数字类型的值。

简言之，这两个类主要作为本章尚未介绍部件的中间部件存在，从而较大程度地简化其他部件的内部设计。虽然读者在创建高度自定义的表单部件时可能会想到这两个类，但它们确实不是为通用目的而设计的。

## TimeTextBox 和 DateTextBox

为了验证日期和时间而编写验证程序是另一个几乎所有Web开发人员都必须面对的问题。虽然日期和时间的格式比较固定，应用也相当普遍，但过分通用的HTML INPUT元素却没有对此提供支持。因此，验证和自定义格式的任务就被交给了JavaScript。幸运的是，Dijit为日期和时间的选取提供了应有的便利。TimeTextBox和DateTextBox都经过了预先配置，能够支持大多数常见的地区。而扩展这两个部件的内置地区也很简单。

---

**注意：** DateTextBox和TimeTextBox部件使用的是格里历，而这也是dojo.date的默认设置。

---

假设你不愿意被人意外地打扰，而公司也不希望在你辛苦地工作了一天回到家后，再贸然给你打电话。自然，公司应该考虑提前收集你什么时候方便接听电话的信息，以免双方都不愉快。假设相关负责人非常明智，选择了Dojo来降低开发这套信息系统的预算，那么就可能存在下面的表单字段：

```
<!-- 不要忘记在使用这些部件之前，先通过dojo.require加载它们！ -->

Best Day to call:
<input dojoType="dijit.form.DateTextBox"><br>

Best Time to call:
<input dojoType="dijit.form.TimeTextBox"><br>
```

就这么简单！无须进行额外处理。当鼠标单击第一个INPUT元素时，如图13-2所示的一个漂亮的小日历（DateTextBox）就会自动弹出，而当鼠标单击第二个INPUT元素时，则会出现如图13-3所示的包含时间的滚动列表（TimeTextBox），其中的时间以15分钟为单位递增。

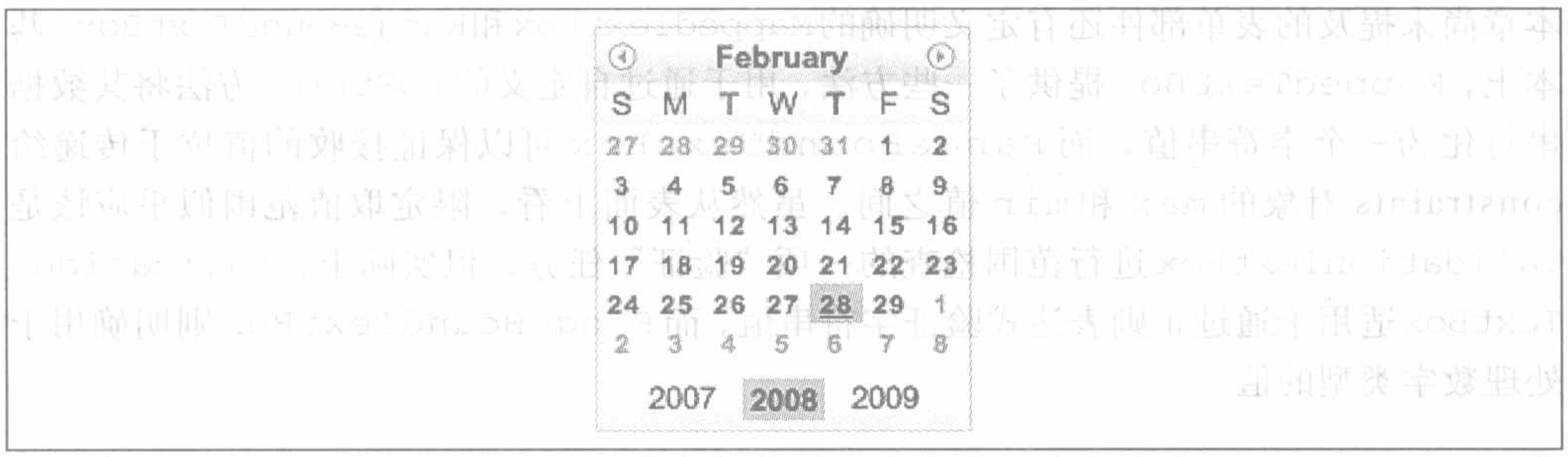



图13-2：弹出的DateTextBox部件的外观

事实上，以编程方式创建这两个部件同样简单：

```
var t = new dijit.form.TimeTextBox();
var d = new dijit.form.DateTextBox();

/* 接下来，就可以通过它们的domNode属性将它们放到页面中了 */
```

这两个部件除了使用简单之外，还允许开发人员定制显示值的格式，凡是通过dojo.date能够做到的，通过这两个部件照样能够做到（它们内部使用了dojo.date）。特别是可以在constraints对象中指定formatLength、timePattern和datePattern属性，以便得到相应的结果。

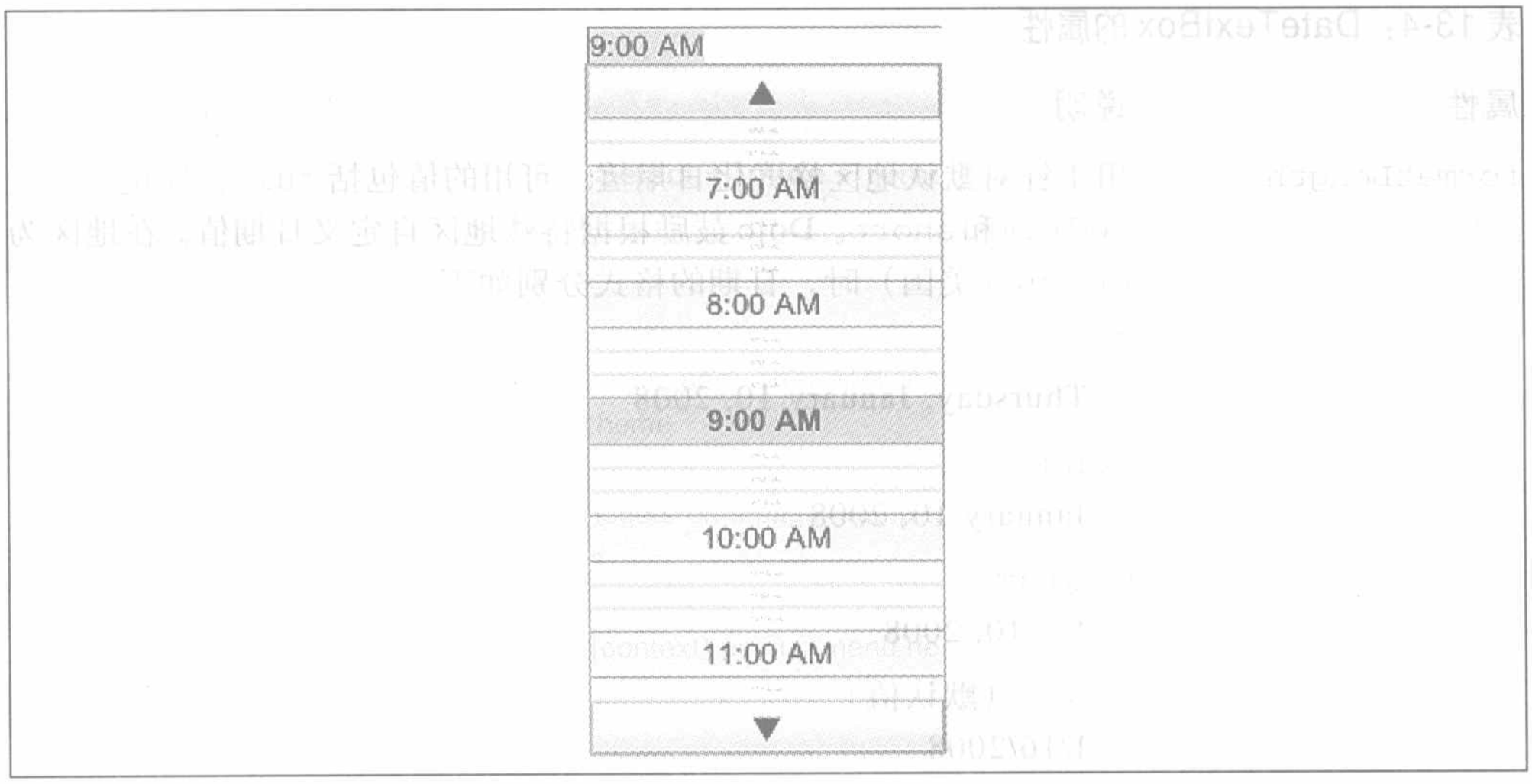



图 13-3：弹出的 TimeTextBox 部件的外观

## 祖鲁（Zulu）、格林威治（Greenwich）、格里历（Gregorian）……都是什么

读者对本章提及的一些术语可能只听说过但并没有研究过。以下就简单地介绍一下这几个术语的含义：

- 祖鲁时间指的是UTC(Coordinated Universal Time，世界时间)时区时间，UTC时区位于西欧，参考的是英国格林威治天文台的时间。这个时区在历史上使用字母Z来表示。在军队及其他组织使用的地图上，通常以单词Zulu取代字母Z(以防与发音相同的字母混淆)。
- 就时区而言，格林威治标准时间与UTC是相同的。
- 格里历是以罗马教皇格里高利十三世命名的。1582年，格里高利十三世修订罗马儒略历（由凯撒大帝在公元前一世纪作为对罗马历的修订颁布，该历法存在春分飘离的现象，历年稍长），颁布了格里历。

表 13-4 和表 13-5 分别列出了 DateTextBox 和 TimeTextBox 部件的属性。通常，对这两个部件要么是指定格式长度（formatLength）属性，要么是根据期望的间隔时间指定时间（timePattern）或日期模式（datePattern）属性。

表13-4：DateTextBox的属性

| 属性 | 说明 |
| --- | --- |
| formatLength | 用于针对默认地区格式化日期值。可用的值包括full、long、medium和short。Dojo鼓励根据特殊地区自定义日期值。在地区为en-us（美国）时，日期的格式分别如下：<br>full<br>Thursday, January 10, 2008<br>long<br>January 10, 2008<br>medium<br>Jan 10, 2008<br>short（默认值）<br>1/16/2008 |
| datePattern | 用于针对所有地区自定义日期格式。根据类似Java中的约定接收一个格式化字符串。参见*http://www.w3.org/TR/NOTE-datetime*。常见日期格式的示例如下：<br>yyyy<br>2008<br>yyyy-MM<br>2008-01<br>MMM dd, yyyy<br>Jan 08, 2008 |
| strict | 如果值为true，那么允许放宽某些缩写和空格。默认值为false |
| locale | 用于针对当前部件覆盖默认地区。使用这个属性要确保通过djConfig.extraLocale配置额外的地区，否则将产生错误或导致意外结果 |
| selector | 当提交表单时，这个属性的值用于决定日期、时间或者两者是否被一起提交，尽管在显示的值中只有一个日期或时间可见。默认同时传递日期和时间，相应地指定日期或时间 |

表13-5：TimeTextBox的属性

| 属性 | 说明 |
| --- | --- |
| clickableIncrement | 用于表示时间拾取器（picker）中每个可单击元素时间增量的字符串值。应该用不带时区的非祖鲁时间设置这个属性，并且能够被visibleIncrement整除。默认值"T00:15:00"表示时间增量为15分钟 |

表 13-5：TimeTextBox 的属性（续）

| 属性 | 说明 |
| --- | --- |
| visibleIncrement | 用于表示可见时间增量的字符串值。默认值 "T01:00:00" 表示每小时创建一条表示时间增量的文本。应该用不带时区的非祖鲁时间设置这个属性 |
| visibleRange | 用于表示可见时间范围的字符串值。默认值为 "T05:00:00"，即5个小时。应该用不带时区的非祖鲁时间设置这个属性 |
| formatLength | 用于针对默认地区格式化时间值。可用的值包括 long 和 short。Dojo 鼓励根据特殊地区自定义时间值。在地区为 en-us（美国）时，时间的格式分别如下：<br>long<br>10:00:00PM CST<br>short<br>10:00 PM |
| timePattern | 用于针对所有地区自定义时间格式。根据类似 Java 中的约定接收一个格式化字符串。参见 *http://www.w3.org/TR/NOTE-datetime*。常见时间格式的示例如下：<br>hh:mm<br>08:00<br>h:mm<br>8:00<br>h:mm a<br>8:00 PM<br>HH:mm<br>22:00<br>hh:mm:ss<br>08:00:00<br>hh:mm:ss.SSS<br>08:00:00.000 |
| strict | 如果值为 true，那么允许放宽某些缩写（例如，am 变成 a.m. 等）和空格。默认值为 false |
| locale | 用于针对当前部件覆盖默认地区。使用这个属性要确保通过djConfig.extraLocale配置额外的地区，否则将产生错误或导致意外结果 |
| selector | 当提交表单时，这个属性的值用于决定日期、时间或者两者是否被一起提交，尽管在显示的值中只有一个日期或时间可见。默认同时传递日期和时间，相应地指定日期或时间 |

在标记中，可以像指定其他属性一样指定constraints对象：

```
<input constraints="{datePattern:'MMM dd, yyyy'}" dojoType="dijit.form.DateTextBox">
```

当然，以编程方式创建部件时仍然只需直接转换即可：

```
var d = new dijit.form.DateTextBox({datePattern:'MMM dd, yyyy'});
```

### DateTextBox与TimeTextBox的共有特性

TimeTextBox和DateTextBox都具有另外两个方法：getDisplayedValue和setDisplayedValue。这两个方法与原始的getValue及setValue之间的区别在于，前两个方法操作的是显示在部件中的值，而后两个方法操作的是部件内部使用的数据类型。TimeTextBox和DateTextBox在内部都使用JavaScript的Date对象，而取得Date对象只不过是一次方法调用的事。

由于它们都继承了RangeBoundTextBox，因此也可以在constraints对象中使用min和max值，这对于防止用户从弹出的拾取器中选择无效值是非常有用的。例如，如果想把用户可选的有效日期范围限定在从2007年12月1日到2008年6月30日，那么就可以这样：

```
<input constraints="{min:'2007-12', max:'2008-06', datePattern:'MMM dd, yyyy'}"
 dojoType="dijit.form.DateTextBox">
```

此外，MappedTextBox还为这两个部件赋予了通过toString方法串行化值的能力；因此，在必要时可以取得一个符合ISO-8601规范的字符串，以便向服务器发送数据。

---

**注意：** 理解datePattern、timePattern与ISO-8601规范的对偶性（duality）也很重要：从根本上讲，它们之间没有联系。datePattern和timePattern用于操作用户看得见的部件值的格式，而ISO-8601字符串则是解析器接收并发送给服务器处理的格式。

---

在TimeTextBox和DateTextBox部件共有的getDisplayedValue和setDisplayedValue方法中，后者会产生与执行setAttribute('value', /*...*/)相同的结果，前者用于返回部件中显示的值，而解析部件的.value属性返回的则是一个JavaScript的Date对象。

表13-6列出了TimeTextBox和DateTextBox部件的共有特性。

表 13-6：TimeTextBox 和 DateTextBox 部件的共有特性

| 名称 | 说明 |
| --- | --- |
| `getDisplayedValue()` | 取得表单元素中实际显示的经过格式化的值，但 `getValue` 取得的则是一个 `Date` 对象 |
| `setDisplayedValue(/*Date*/ date)` | 同时设置部件显示的值和内部使用的值（调用 `setValue` 也能得到同样的结果） |
| `toString()` | 返回一个符合 ISO-8601 规范的日期或时间值 |
| `min` 和 `max` 值约束 | 为弹出的拾取器中可选的值设置约束条件 |
| `serialize()` | 扩展点，可以用来为 `toString` 方法指定一个自定义的实现。这个扩展点操作的是表单提交时提供给服务器的值 |

### 串行化发送给服务器的数据

由于这两个部件都提供了`serialize`扩展点，因此在与服务器端组件之间来回传递特殊格式的数据时会特别方便。例如，可以像例13-5中所示的那样，通过扩展`DateTextBox`实现调用 `toString` 方法时产生自定义格式的数据。例 13-5 还展示了如何自定义 `DateTextBox`，以便提交与显示格式不同的数据值。

例 13-5：通过 DateTextBox 自定义发送给服务器的串行化数据

```
<html>
    <head>
        <title>Custom DateTextBox</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:false"
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.form.DateTextBox");

            dojo.addOnLoad(function() {
                dojo.declare("dtdg.CustomDateTextBox",[dijit.form.DateTextBox], {
                    serialize: function(d, options) {
                        return dojo.date.locale.format(d,
                        {
```

```
                        selector:'date',
                        datePattern:'dd-MMM-yyyy'}).toUpperCase();
                }
            });
            dojo.parser.parse(dojo.body());
        });
    </script>
</head>
<body class="tundra">
    <form action="http://localhost:8080" type="POST">
        <input dojoType="dtdg.CustomDateTextBox" name="customDate"/>
        <input type="submit" value="Submit"/>
    </form>
</body>
</html>
```

下面是接收并显示表单提交数据的最简单的CherryPy类：

```
import cherrypy

class Content:
    @cherrypy.expose
    def index(self, **kwargs):
        return str(kwargs)

cherrypy.quickstart(Content())
```

### 不要忘记继承的属性

在提到从TextBox和ValidationTextBox继承的属性时，读者可能会顾虑到继承层次太深的问题。实际上，无论继承层次有多深，从这两个部件继承来的属性不仅可以正常使用，而且在很多情况下还是必要的。另外，回顾一下第6章介绍过的dojo.date对于理解TimeTextBox和DateTextBox部件同样大有裨益。

## NumberTextBox

NumberTextBox继承了RangeBoundTextBox及其超类的所有特性，并且又通过dojo.number扩展了它们，实现了自有的数值类型特性。简单地说，数值的格式默认为当前地区使用的格式，并且可以通过表13-7所列的项以constraints对象形式提供约束条件。

表13-7：NumberTextBox的constraints项

| 名称 | 说明 |
| --- | --- |
| min和max值约束 | 作为检查输入范围的依据，与RangeBoundTextBox的其他子类相同 |
| pattern | 用于指定小数点后面的数字个数及其他格式，如后跟的百分号 |

表 13-7：NumberTextBox 的 constraints 项（续）

| 名称 | 说明 |
| --- | --- |
| type | 用于指定值是小数还是百分数 |
| places | 用于指定小数点后面的位数（同时指定此项与 pattern 项，会导致覆盖 pattern 项） |

例如，要让小数点后面保留两位数字并且带一个百分号，可以这样做：

```
<input constraints="{pattern: '#.##%'}" dojoType="dijit.form.NumberTextBox">
```

虽然这个例子中的小数位之前只有一个#符号，但实际上允许有多个数字。假如想让小数位之前没有数字，可以使用不带前面那个#符号的模式，如{pattern:'.##%'}。需要注意的是，在输入开始时，显示的值会自动转换为纯数字值；而当输入结束后，该值会转换回格式化的数值。

理解第6章介绍的dojo.number的各种特性，就能一劳永逸地理解数字格式化及相关操作。毕竟，NumberTextBox直接构建于这些特性之上。

## NumberSpinner

我们曾在第 11 章中介绍过 NumberSpinner 部件。可以把 NumberSpinner 想像成奇特的NumberTextBox，在它的文本框右侧还带有两个可以用来控制值递增/递减的小按钮（称为数值微调器）。这两个按钮具有自动击键（typematic）能力，即按住它们不放可以持续地改变值。另外，在为 NumberSpinner 指定 min 和 max 约束条件的情况下，其约束也会通过两个小按钮表现出来，即不能通过按钮生成超出范围的值。

NumberSpinner 提供的属性如表 13-8 所示。

表 13-8：NumberSpinner 的属性

| 名称 | 说明 |
| --- | --- |
| defaultTimeout | 在启用自动击键功能之前，持续按住键或按钮的毫秒数。默认值为 500 |
| timeoutChangeRate | 用于改变自动击键事件发生频率的时间因子。值为1.0表示每次自动击键都间隔defaultTimeout指定的时长。值小于1.0表示自动击键的时间间隔按照这个值指定的比例逐渐缩短，因此击键速度越来越快。默认值为 0.90 |
| smallDelta | 用于调整通过按箭头键或按钮每次能够改变的值。默认值为 1 |
| largeDelta | 用于调整通过按 Page Up 或 Page Down 键每次能够改变的值。默认值为 10 |

创建NumberSpinner与创建其他部件完全一样：

```
<input dojoType="dijit.form.NumberSpinner" smallDelta="2" largeDelta="4"
constraints="{min:100,max:120}" value="100">
```

## CurrencyTextBox

CurrencyTextBox是距离公有超类最“远”的一个部件，它也继承了NumberTextBox，同时又利用dojo.currency的大量特性来处理其格式化操作。

不过，这个部件只额外地提供了一个属性currency，用于指定针对哪个地区进行格式化。currency属性的值必须采用ISO4217货币代码标准（*http://en.wikipedia.org/wiki/ISO_4217*）中规定的3个字母的序列。

---

**警告：** 无论什么时候，只要用到像货币符号之类的国际化字符，就应该特别留意浏览器使用的字符编码，以保证所有符号都能正确地显示。依靠Web服务器在响应头部提供编码信息并不是可靠的办法。

在HTML页面中，指定字符编码的标准方式是在页面头部添加一个特殊的META标签，而Dijit也将这种方式作为一种最佳实践。下面的示例使用META标签为页面设置了UTF-8字符集，这样做基本上就可以确保万无一失了：

```
<META http-equiv="Content-Type"
content="text/html; charset=UTF-8"/>
```

到Dojo1.1为止，如果要通过AOL的CmDN加载Dojo，就需要像上面示例一样指定字符编码，因为服务器没有在响应头部中包含编码信息 —— 当然，这是指定编码的另一种方式。否则，货币及某些Unicode符号可能无法正确显示。

---

下面的代码片段示范了一个货币文本框部件，该部件通过fractional约束项要求输入一个小数点后带分值的美元数值。除继承的约束项之外，这是该部件提供的唯一一个额外的约束项：

```
<input dojoType="dijit.form.CurrencyTextBox"
constraints="{min:1,max:100,fractional:true}" currency="USD"/>
```

与NumberTextBox部件一样，在输入开始时，CurrencyTextBox部件中显示的值也会自动转换为纯数字值；而当blur事件发生时，该值会转换回货币值的格式。

## ComboBox

ComboBox能够像HTML中的SELECT元素一样提供一个包含可选值的下拉列表；但是，

## 关于字符编码

UCS（Universal Character Set，通用字符集）是由ISO（International Organization for Standardization，国际标准化组织）和IEC（International Electrotechnical Commission，国际电工委员会）共同制定的一项国际标准。UCS规定了大约10万个字符并赋予每个字符一个唯一的数字，称为“编码点”。

Unicode是作为一项与UCS并列的产业标准而开发出来的。Unicode常常被认为是UCS的一个实现，或者是UCS背后的一种思想。由于它们之间的关系模糊，因此经常会成为一些学术争论的话题。不过，问题的关键在于这两项标准相互之间是同步的。

随着信息系统处理语言（甚至今天人们已经很少使用的古代语言）中任意字符的需求日益增长，字符编码也变得越来越重要。Unicode为产业的重要参与者提供了一种标准化常用系统的手段，从而确保了最大限度的兼容性。勿庸置疑的是，任何Web服务器与浏览器、邮件服务器与邮件客户端等系统之间能够随意通信的价值无法估量。

虽然7位的ASCII（American Standard Code for Information Interchange，美国标准信息交换编码）能够很好地满足处理英语字母的需要，但作为一种通用标准是远远不够的。UTF-8（Unicode Transformation Format，Unicode转换格式）编码能够表示Unicode标准规定的任何字符，并能够完全向后兼容ASCII。因此，UTF-8变得日益流行，并在网页、电子邮件等应用领域被广泛采用。

为了维护与ASCII的向后兼容，UFT-8在实现上使用了可变长度的编码，即任何字符都可能使用1个或4个8位字节来表示。为此，Web服务器通过响应头部标识特殊的编码，或者网页本身使用META标签来指定特殊的编码，对于正确地显示字符就显得尤为重要。在不知道编码的情况下，浏览器就无法将它接收到的字节流正确转换成离散的字符。

要了解有关Unicode的更全面的资料，可以参考 *http://www.unicode.org*。

ComboBox是在普通的INPUT元素基础上构建的。因此，如果列表中的值不合适，用户还可以手工输入。ComboBox继承自ValidationTextBox，所以任何验证特性对它都适用。除了继承的验证特性之外，它还支持根据输入的开头字符自动筛选列表中的可能值。可选值列表既可以是根据预定义值生成的静态列表，也可以是基于从服务器获取数据的dojo.data存储模式生成的动态列表。

在最简单的情况下，可以使用ComboBox提供常用选项的列表，同时允许用户输入自己的选项。下面的代码示范了如何使用静态数据，同时启用自动完成功能。

```
<select name="coffee" dojoType="dijit.form.ComboBox" autoComplete="true">
  <option>Verona</option>
  <option>French Roast</option>
  <option>Breakfast Blend</option>
  <option selected>Sumatra</option>

  <script type="dojo/method" event="onChange" args="newValue">
    console.log("value changed to ", newValue);
  </script>
</select>
```

把ComboBox与ItemFileReadStore数据源连接起来也非常简单，只要为ComboBox指明该数据源即可。例如，如果有一个包含咖啡及其描述信息的数据源，格式如下：

```
{identifier : "name",
    items : [
        {name : "Light Cinnamon", description : "Very light brown, dry , tastes like
toasted grain with distinct sour tones, baked, bready"},
        {name : "Cinnamon", description : "Light brown and dry, still toasted grain
with distinct sour acidy tones"},

        ……更多咖啡……
    ]
}
```

而我们要通过其中的name字段来填充ComboBox；当用户改变选项时，再以某种有意义的方式来利用该数据源中的描述信息。那么，例13-6就是实现这个构想的一种方案。

例13-6：ComboBox的应用

```
<html>
  <head>
    <title>Pick a coffee roast, any coffee roast</title>

    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

    <script
      djConfig="parseOnLoad:true"
      type="text/javascript"
      src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
    </script>

    <script type="text/javascript">
      dojo.require("dojo.parser");
      dojo.require("dojo.data.ItemFileReadStore");
      dojo.require("dijit.form.ComboBox");
```

```
            dojo.require("dijit.form.Button");
            dojo.require("dijit.form.Form");
        </script>
    <head>
    <body class="tundra">

        <div dojoType="dojo.data.ItemFileReadStore"
            jsId="coffeeStore" url="./coffee.json"></div>

        <form action="localhost" dojoType="dijit.form.Form">
            <select name="coffee" dojoType="dijit.form.ComboBox"
                store="coffeeStore" searchAttr="name">

                <script type="dojo/method" event="onChange" args="newValue">
                    console.log("value changed to ", newValue);
                    var f = function(item) {
                        console.log("new description is ",
                          coffeeStore.getValue(item, "description")
                        );
                    };
                    coffeeStore.fetchItemByIdentity(
                        {identity : newValue, onItem : f}
                    );
                </script>
            </select>
          <button dojoTyype="dijit.form.Button">Submit</button>
        </form>
        </body>
    </html>
```

在这个示例中，我们通过store属性将ComboBox与ItemFileReadStore联系在了一起，并通过 searchAttr 属性告诉 ComboBox 显示哪个字段。然后，当用户改变选项时，ComboBox的onChange方法会做出反应，并使用新值从数据源中查找描述信息。

---

**注意：** ComboBox只在内部实现了dojo.data.Read/Notification API的一个子集，这个子集对于它利用 dojo.data 存储模式是必需的。ComboBox 实现的相关方法包括：

- getValue
- isItemLoaded
- fetch
- close
- getLabel
- getIdentity
- fetchItemByIdentity
- fetchSelectedItem

---

为了完整起见，表13-9列出了ComboBox中可用的属性。

表13-9：ComboBox的属性

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| item | 对象 | 当前选中的项。默认值为null |
| pageSize | 整数 | 指定每页的结果数量(通过ItemFileReadStore的fetch方法的count键)。在查询大数据集时有用。默认值为Infinity |
| store | 对象 | 引用提供数据的对象，如ItemFileReadStore的实例。默认值为null |
| query | 对象 | 可以被传入存储模式中的查询对象，用于在基于searchAttr和当前输入的关键字对数据正式筛选之前的初步筛选。默认值为{} |
| autoComplete | 布尔值 | 是否为当前输入的关键字显示一个选项列表（以queryExpr作为搜索依据)。默认值为true |
| searchDelay | 整数 | 指定在按键后与开始搜索输入的值之间等待毫秒数。默认值为100 |
| searchAttr | 字符串 | 指定搜索要显示的值时使用的匹配模式。默认值为name |
| queryExpr | 字符串 | 由dojo.data的查询表达式使用。(默认的表达式搜索以当前输入值开头的任何值。）默认值为"${0}*" |
| ignoreCase | 布尔值 | 指定查询是否区分大小写。默认值为true |
| hasDownArrow | 布尔值 | 指定是否显示向下箭头作为可以打开下拉列表的指示器。默认值为true |

# FilteringSelect

FilteringSelect部件增强了普通的HTML SELECT元素，它提供了一个包含固定值的下拉列表，并且能够提交隐藏及可见的值。虽然FilteringSelect与ComboBox看起来相似，而且也具有一些相同的特性 —— 包括能够随着文本的输入而筛选下拉列表项和通过存储模式从服务取得数据，但它是基于HTML SELECT元素构建的。

有必要指出的是，FilteringSelect与ComboBox之间存在以下重要区别：

- FilteringSelect基于普通的SELECT元素构建，而通过submit事件提交给服务器的值是Select控件隐藏的值，而非控件中可见的值。这个区别是一个很重要的特性，因为FilteringSelect在退化后非常类似一个普通的SELECT元素。

- FilteringSelect继承自MappedTextBox（一个可串行化的TextBox），而不是ValidationTextBox。由于用户不能在这个控件中随意输入文本，因此对它不需要验证。
- FilteringSelect的标签（label）中不仅可以显示文本，也可以显示HTML。也就是说，可以为选项列表中加入自定义的标记，例如，图像。

除了getValue、setValue、getDisplayedValue、setDisplayedValue等常见的dijit.form操作，以及各种ComboBox选项之外，FilteringSelect还另外提供了两个属性和一个函数，如表13-10所示。

表13-10：FilteringSelect新增的特性

| 名称 | 说明 |
| --- | --- |
| labelAttr | 用于指定显示在控件中的文本。如果未指定这个属性，那么使用searchAttr |
| labelType | 用于指定将文本标签当作标记还是纯文本。可用的值包括'text'和'html' |
| labelFunc (/*Object*/ item, /*dojo.data.store*/ store) | 当标签改变时被调用的事件处理函数；返回应该被显示标签 |

# MultiSelect

MultiSelect是一个简单的包装部件（表13-11列出了它的属性），它包装的是multiple="true"的SELECT元素。MultiSelect直接继承自_FormWidget。在Dijit中加入这个部件的原因，主要是它能够方便与dijit.Form包装部件（本章后面将介绍到）的交互，并消除自己设计SELECT元素的麻烦。

表13-11：MultiSelect的属性

| 名称 | 说明 |
| --- | --- |
| size | 指定一页中显示的元素数量。默认值为7 |
| addSelected(/*dijit.form.MultiSelect*/ select) | 用于从另一个MultiSelect中将选中的节点移动到当前MultiSelect中 |
| getSelected() | 返回选中的节点 |
| setValue(/*Array*/values) | 根据values数组中提供的有序的值设置部件中的每个节点 |
| invertSelection(/*Boolean*/ fireOnChange) | 反转当前选中的项。如果fireOnChange值为true，那么会触发一次onChange事件 |

鉴于MultiSelect只是简单地封装了对等的HTML控件，因此从Dojo的角度来说，没有太多需要介绍的。例13-7展示了如何在标记中定义MultiSelect部件。

例13-7：在标记中创建MultiSelect部件

```
<select multiple="true" name="foo" dojoType="dijit.form.MultiSelect"
 style="height:100px; width:100px; border:3px solid black;">

   <option value="TN" selected="true">Tennessee</option>
   <option value="VA">Virginia</option>
   <option value="WV">West Virginia</option>
   <option value="OH">Ohio</option>

</select>
```

# Textarea及其变体

在传统的Web开发中，占据屏幕上固定空间的TEXTAREA元素经常会成为棘手的问题。一方面，要给它分配足够的空间以确保其可见区域最大化；另一方面，还要保证它不会过分占用宝贵的屏幕资源。

## Textarea

直接继承自_FormWidget的Textarea部件对上述难题给出了两全其美的解决方案：它既支持普通TEXTAREA元素支持的标准HTML属性，又能在宽度固定的前提下实现随内容增多而垂直扩展。Textarea部件的API简单到在正常情况下只需使用setValue和getValue方法就足够了。另外，如果想在内容发生变化时定义一个回调函数，那么可以使用onChange扩展点：

```
<textarea dojoType="dijit.form.Textarea" style="width:300px">
    One fish, two fish...
</textarea>
```

## SimpleTextarea

虽然可垂直扩展的Textarea部件解决了普通TEXTAREA元素存在的主要问题，但是，在需要填满其包装容器（例如，将在第14章介绍的布局部件）的情况下却不能胜任。为此，Dijit中引入了SimpleTextarea部件。实际上，SimpleTextarea的行为与普通的TEXTAREA元素非常相似，只不过它的大小能够扩展和收缩。可以像使用普通的TEXTAREA元素一样，通过rows和cols属性来定义它，也可以像使用Textarea一样，通过setValue和getValue来操作其中的文本。

# Button及其变体

Dijit为标准的按钮和复选框给出了简便、可退化的替代部件，当然也为这些部件提供了许多通常在工具栏中才能见到的高级选项。下面，我们就从基本的Button部件开始，循序渐进地展示更高级的选项。

## Button

图13-4展示了一个按钮，而表13-12则列出了这个直接继承自_FormWidget的Button部件的最主要特性。

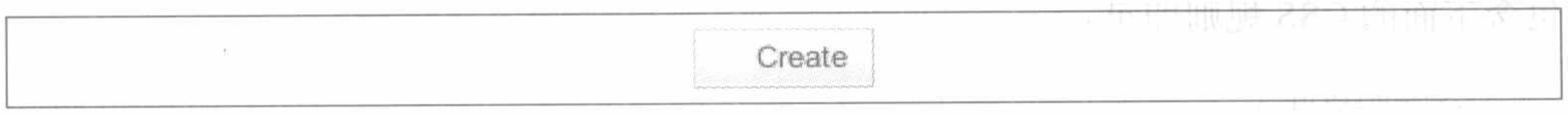



图13-4：典型的按钮

表13-12：Button部件的特性

| 名称 | 注意 |
| --- | --- |
| label | 用于在标记中或者通过编程方式为按钮指定标签(label) |
| showLabel | 表示是否在Button部件中显示文本标签的布尔值。默认值为true |
| iconClass | 用于为Button 部件指定一个关联图像的CSS类，可以让按钮变得像图标一样 |
| onClick(/* DOM Event*/ evt) | 扩展点，在用户单击按钮时调用。是一个经常需要重写的方法 |
| setLabel(/* String */ label) | 接收HTML字符串参数，可以改变Button 部件的标签 |

**注意：** 与TextBox及其变体（子类部件）不同，Button部件的值需要使用继承自_FormWidget的setAttribute('value', /*...*/)方法来设置。因为Button不像其他表单部件那样拥有最终会传送给服务器的部件值。

例13-8在例13-4的基础上，替换了其中难看的按钮元素，让表单焕发了新的光彩。除了不要忘记在页面的头部添加必须的dojo.require("dojo.form.Button")语句，其他的代码都很直观。注意，在这种情况下通过标记来定义onClick处理程序也是很方便的。

例13-8：典型的Button应用

```
<button dojoType="dijit.form.Button" type="submit">Sign Up!
    <script type="dojo/method" event="onClick" args="evt">
        alert("You just messed up...but it's too late now! Mwahahaha");
    </script>
</button>
<button dojoTye="dijit.form.Button" type="reset">Reset</button>
```

Button的iconClass属性非常值得一提，因为它并不是使用一个图标来替换整个按钮，而是将图标嵌入到按钮中；如果同时为按钮指定了label属性且showLabel为true的话，图标和标签会同时显示。如果想在前面那个“Sign Up!”按钮中嵌入一个20 × 20px的图标，首先要在button标签中加入iconClass="spamIcon"属性，然后再在页面中包含下面的CSS规则即可：

```
.spamIcon {
    background-image:url('spam.gif');
    background-repeat:no-repeat;
    height:20px;
    width:20px;
}
```

当然，无论使用行内样式还是通过自定义的类，都能再为按钮应用更符合整体风格的样式。

## ToggleButton

我们知道，表单部件最大限度地利用了继承关系，因此许多部件都继承了为它们提供公共特性的超类。ToggleButton就是这样一个部件；它继承自Button，并为按钮添加了开/关功能，就像RadioButton和CheckBox一样。ToggleButton特有的唯一一个重要属性是checked，可以通过setAttribute方法来切换这个属性。

虽然有时候使用CheckBox来指明开关状态更方便，但直接使用ToggleButton，或者对它进行扩展以实现一个自定义的ToggleButton也是没有问题的。对于ToggleButton而言，onChange扩展点（很多表单部件都有）是一个特别有用的特性：

```
<button dojoType="dijit.form.ToggleButton">
  <script type="dojo/method" event="onChange" args="newValue">
     console.log(newValue);
  </script>
</button>
```

事实上，工具栏上用于格式化文本的按钮，如斜体、粗体、下划线等等，都使用ToggleButton。相关的Menu和MenuItem部件将在第15章介绍。

**警告：** 有些按钮部件并没有包含在自己专门的资源文件中。例如，dojo.require("dijit.form.Button")语句可以加载Button、ToggleButton、DropDownButton和ComboButton。虽然这看起来有点风马牛不相及，但实际上由于这几种按钮（继承关系）的实现非常相似，所以就把它们放到了一个文件中。而且，这样做还能降低向服务器请求资源时的系统开销。也就是说，通过dojo.declare模拟的类与资源文件之间也不一定是一对一的关系（尽管从传统面向对象编程的角度上看通常如此）。

不过，这种做法可能会在人们使用Dojo的过程造成一些误会，而且如果在项目开发中使用Util提供的工具将应用程序的每个页面都作为一层来优化，那么对服务器发送同步请求的开销是否可以接受也存在争议。

类似这种实现上的微小差别经常会在Dojo社区中引起人们激烈的（也是善意的）辩论。

## CheckBox

CheckBox直接扩展了ToggleButton，是普通<input type="checkbox">元素的替代部件。要使用这个部件，同样只要将它请求到页面中，然后再为一个标签指定dojoType属性即可。我们可以在例13-4的页面中使用CheckBox并禁用“Sign Up!”按钮，直到用户通过单击CheckBox确认他们明白我们的意图再启用该按钮：

```
<div name="confirmation" dojoType="dijit.form.CheckBox">
  <script type="dojo/method" event="onClick" args="evt">
    if (this.checked)
      dijit.byId("signup").setAttribute('disabled', false);
    else
      dijit.byId("signup").setAttribute('disabled', true);
  </script>
</div> I understand that you intend to spam me.<br>
<button id="signup" disabled dojoType="dijit.form.Button" type="submit">
Sign Up!
</button>
```

图13-5展示了CheckBox的几种状态。


图13-5：几种CheckBox部件

**警告：** 前面的示例之所以没有使用INPUT标签而使用DIV标签，是因为在INPUT标签中不能嵌入SCRIPT标签，否则浏览器几乎肯定会忽略它们。因此，如果要在部件内部使用SCRIPT标签的话，就必须牢记不能使用INPUT标签。假如要求应用程序必须可退化，那么就不要在标记中定义方法，仍然以纯JavaScript方式写个方法即可。

如果以编程方式创建CheckBox，就要用到setValue(true)方法。这个方法会为复选框打上对勾（即将CheckBox的checked属性设置为true）并将其value属性设置为true。

**注意：** 如果要尽最大努力确保每个页面都能合理地退化，还可以再多做一步，即在标签中明确包含原始的HTML属性。例如，<input dojoType="dijit.form.CheckBox"/>应该写成<input dojoType="dijit.form.CheckBox type="checkbox"/>。

然而，与普通的HTML复选框元素类似，CheckBox部件的状态和它的值并不是一回事。复选框的状态指复选框是否被选中，并且通过标准的checked属性可以检测这个状态。而value属性则可以接收一个非布尔值并将该值通过表单提交到服务器。例如，标签<input name="pleaseSpamMe" value="yes"/>会在以GET方式提供表单时，将pleaseSpamMe=yes追加到查询字符串中。（value属性的默认值是 "on"。）

于是，问题就来了。有时我们会发现getValue方法返回的值与value属性的值可能会不一致。原因在于，getValue方法只返回复选框是否被选中，而不管value属性的值被设置成什么。毕竟，getValue方法最常见的用途是确定部件在视觉上的开/关状态，并非取得该部件的实际值，而且不能用这个值来确定开/关状态。

为了展示各种可能性，我们以下面常见的CheckBox部件为例：

```
<input id="foo" dojoType="dijit.form.CheckBox"></input>
```

例13-9中给出了一系列的调用，并穿插了一些注释。

例13-9：典型的CheckBox应用

```
/* 检查初始状态 */
dijit.byId("foo").checked // false
dijit.byId("foo").getValue() // false

/* 以true调用setValue */
dijit.byId("foo").setValue(true) // 选中复选框并将value设置为true
dijit.byId("foo").checked // true
dijit.byId("foo").getValue() // "on"
```

```
/* 以false调用setValue */
dijit.byId("foo").setValue(false) //取消选中并将value设置为false
dijit.byId("foo").checked // false
dijit.byId("foo").getValue() // false

/* 以字符串调用setValue */
dijit.byId("foo").setValue("bar") //选中复选框并将value设置为"bar"
dijit.byId("foo").checked //true
dijit.byId("foo").getValue() // "bar"
```

在使用CheckBox时，最常见的做法是使用getValue和以布尔值作为参数调用setValue，因此即使偶尔混淆了状态与值的区别也不会导致什么问题。

---

**警告：** 以下是在混淆了状态和值的情况可能写出的代码，对这种情况应该特别留意：

```
dijit.byId("foo").setAttribute("value", "foo")
// 修改value属性不会选中复选框
dijit.byId("foo").value // "foo"
dijit.byId("foo").getValue()
//false，因为复选框未被选中
```

由于在JavaScript编程中，对一个不是""、null以及undefined的字符串值进行测试，结果会返回true，因此人们会凭直觉认为，设置了字符串值之后getValue()不应该返回false。但是，请读者务必记住，getValue()返回的始终都是复选框的选中状态，而不是value的实际属性值。在这个示例中，复选框并没有被选中，于是getValue()返回false。

同样，这个部件的onChange事件也不会由于dijit.byId("foo").setAttribute("value", "foo")方法的执行被触发，因为复选框的状态在视觉上并没有发生变化。

---

## RadioButton

RadioButton继承了CheckBox，是对其HTML中对应元素的替代部件；与其HTML对应元素一样，RadioButton仍然需要组织成一组选择按钮，而且每次只能选择其中一个。我们知道，单选按钮组中的每个按钮都具有相同的name值和不同的value值。图13-6展示了一个RadioButton组。


图13-6：RadioButton组

为了示范RadioButton的用法，恐怕接下来还得继续修改前面的表单（例13-4），要求用户选择一天大概可以找他们几次。例13-10中的单选按钮可以帮我们达到这个目的，不过，必须首先在页面中请求dojo.dijit.CheckBox。

警告：或许有读者认为上一段的最后一句话中有排版错误，其实没错。前面曾经提到过，dojo.require语句请求的是资源，并非个别的部件。换句话说，dijit.form.CheckBox资源中包括dijit.form.CheckBox和dijit.form.RadioButton。

当然，前面关于dijit.form.Button资源提供多个部件实现的说明，同样也适用于dijit.form.CheckBox。

例13-10：典型的RadioButton应用

```
<input name="spamFrequency" value="1 per day" dojoType="dijit.form.RadioButton">
  1 per day<br>
<input name="spamFrequency" value="2 per day" dojoType="dijit.form.RadioButton">
  2 per day<br>
<input name="spamFrequency" value="3+ per day" dojoType="dijit.form.RadioButton">
  3+ per day<br>
```

## DropDownButton

DropDownButton是Button的直接子类，这个部件在被单击时会生成一个包含选项的下拉菜单，就跟我们在工具栏中看到的一样。DropDownButton与dijit.Menu的关系比较紧密，因为Menu经常作为下拉菜单的载体；TooltipDialog则是另一个备选部件。图13-7展示了DropDownButton部件。

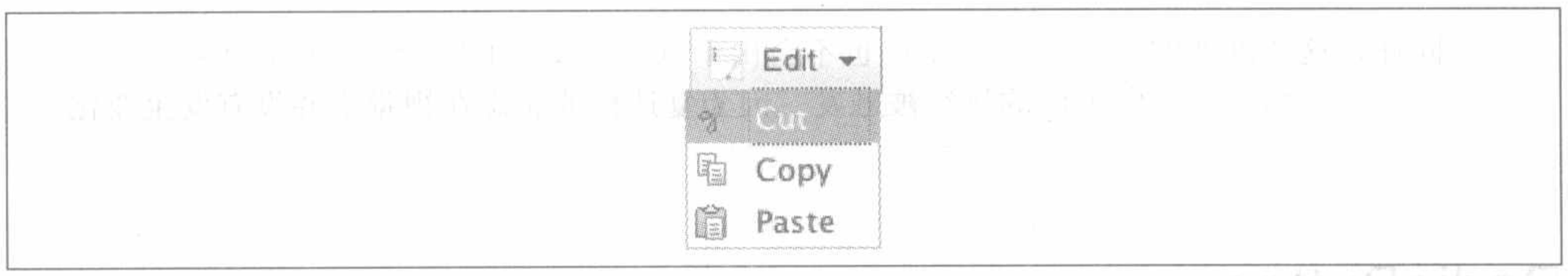



图13-7：DropDownButton部件

有关Menu（以及它所包含的MenuItem）的全面介绍将在第15章中进行。但是，例13-11为了示范DropDownButton的用法需要提前用到它们。注意，作为父节点的DropDownButton的第一个子节点是一个标签，该标签将出现在按钮上。

例13-11：典型的DropDownButton应用

```
<button dojoType="dijit.form.DropDownButton">
    <span>Save...</span>
    <div dojoType="dijit.Menu">
        <div dojoType="dijit.MenuItem" label="Save">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
```

```
        </div>
        <div dojoType="dijit.MenuItem" label="Save as...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save to FTP...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
    </div>
</button>
```

如果想让DropDownButton部件的值能够随表单一起提交，需要创建一个隐藏的INPUT元素，然后在构成选项的MenuItem的onClick方法中以编程方式设置该元素的值。DropDownButton最大的用途是实现应用程序级的行为，例如，保存文档。

---

**注意：** 一般来说，能够通过表单将其数据提交给服务器的表单字段在提交时对用户都是可见的。从这个意义上说，可能有人会认为把DropDownButton放在dijit.form中不太恰当，因为它不属于前面所说的那种表单部件。但是，之所以要在本节介绍它，原因在于它是Button的子类，而让Button的子类寄居在其他命名空间中显然不合理。

---

## ComboButton

ComboButton扩展了DropDownButton，但又加入了新的特性：它在按钮上提供了专门的区域，单击该区域才能出现下拉菜单；如果单击的是按钮上的“其他”区域，那么会执行默认操作。例如，单击“Save”按钮会触发普通的保存操作，而单击按钮的扩展区则会打开一个包含“Save”、“Save as . . . ”、“Save to FTP site”等选项的下拉菜单。图13-8展示了单击扩展区前后的ComboButton。

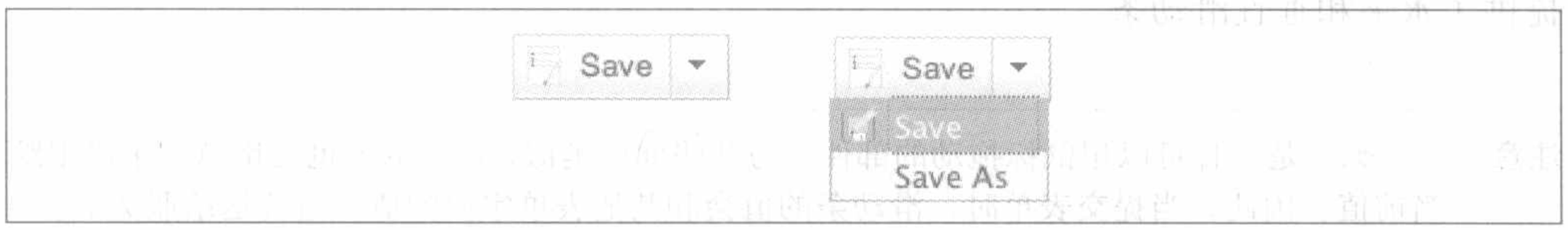



图13-8：左：单击扩展区前的ComboButton；右：单击扩展区后的ComboButton

例13-12示范了如何使用ComboButton。

例 13-12：典型的 ComboButton 应用

```
<button dojoType="dijit.form.ComboButton">
    <span>Save</span>

    <script type="dojo/method" event="onClick" args="evt">
        console.log("you clicked the button itself");
    </script>
    <div name="foo" dojoType="dijit.Menu">
        <div dojoType="dijit.MenuItem" label="Save">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Save As...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
        <div dojoType="dijit.MenuItem" label="Savè to FTP...">
            <script type="dojo/method" event="onClick" args="evt">
                console.log("you clicked", this.label);
            </script>
        </div>
    </div>
</button>
```

在这个示例中，我们注意到ComboButton的标签仍然由第一个子元素（即Save）提供，而包含选项的下拉菜单与 DropDownButton 中的相同。

# Slider

尽管滑动条（slider）并非原生的HTML表单控件，但对于高度形象化的用户界面而言，滑动条似乎又是不可或缺的。无论是调整图像的透明度，还是调整某种颜色在混合色中的比例，亦或调整屏幕上其他控件的大小，滑动条无疑都是非常直观简便的方式。Dijit提供了水平和垂直滑动条。

---

**注意：** Slider是一种可以用鼠标拖动的部件。与某些部件类似，Slider也使用隐藏字段跟踪当前值。因此，当提交表单时，滑动条的值会和其他表单字段的值一道传送给服务器。

---

要将所有 Slider 部件都加载到页面中，只需一条 dojo.require("dijit.form.Slider")语句。除了 VerticalSlider 和 HorizontalSlider 之外，一同加载到页面中的还包括支持刻度尺及标签的辅助类。下面，我们就从简单的应用着手，逐步增加示例的复杂性，让读者体会一下这个奇特部件的自定义空间到底有多大。

## HorizontalSlider

假如你是一名咖啡因爱好者，那么可能会想到创建一个水平滑动条来指示不同饮料中的咖啡因含量。为此，应该先试着在页面中创建一个最简单的滑动条，如例13-13所示，不过别忘了先把dijit.form.Slider加载到页面中。

例13-13：HorizontalSlider（第一步）

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
    value="100"
    maximum="175"
    minimum="2"
    style="margin:5px;width:300px;height:20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
</div>
```

以上代码创建了一个不带任何标签的滑动条部件；滑动条显示的值的范围是2～175，行内样式定义了滑动条占用的空间大小。滑动条的默认值为100，而当滑动条移动时，onChange方法会取得最新的值并将该值显示到Firebug控制台中。另外，单击滑动条上的某个点，滑块也会自动移至该点，并获取新值。目前来看，一切顺利。

接下来继续增强滑动条的功能。首先，通过添加showButtons="false"属性将部件两端的按钮隐藏起来。然后，再在滑动条上方添加一个HorizontalRule和几个HorizontalRuleLabels。由于所需资源已经加载到了页面中，因此直接编写添加刻度尺和标签的代码就行了。不过，为了便于格式化输出结果，还有必要加载dojo.number模块。

如例13-14所示，可以在前面创建的滑动条标记中直接添加代码。

例13-14：HorizontalSlider（第二步）

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
    value="100"
    maximum="175"
    minimum="2"
    showButtons="false"
    style="margin: 5px;width:300px; height: 20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(dojo.number.format(newValue,{places:1,pattern:'#mg'}));
    </script>

    <ol dojoType="dijit.form.HorizontalRuleLabels" container="topDecoration"
        style="height:10px;font-size:75%;color:gray;" count="6">
    </ol>
```

```
    <div dojoType="dijit.form.HorizontalRule" container="topDecoration"
        count=6 style="height:5px;">
    </div>
</div>
```

真是立竿见影啊！现在的滑动条看起来更有形了，不仅出现了刻度，而且还有百分比表示的刻度值。注意，虽然刻度尺不一定要跟刻度值一一对应，但这个示例的效果还是不错的。另外，container属性使用一个由滑动条定义的枚举值"topDecoration"，指定了刻度尺和标签的位置。

虽然滑动条包含了百分率，但如果能在它的下方添加一些与范围对应的饮料名称就更好了。基本的模式与前面一样，只不过这次要占用滑动条的下方容器（container="bottomDecoration"），而非上方容器（container="topDecoration"）。此外，与其接受这个部件默认的某些乏味的数字值，不如像例13-15所示的那样提供一个内容列表，其中还为字数较多的饮料名称添加了<br>标签，以便显示结果清晰明了。最终效果如图13-9所示。

例13-15：HorizontalSlider（第三步）

```
<div dojoType="dijit.form.HorizontalSlider" name="caffeine"
    value="100"
    maximum="175"
    minimum="2"
    showButtons="false"
    style="margin: 5px;width:300px; height: 20px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
    <ol dojoType="dijit.form.HorizontalRuleLabels" container="topDecoration"
        style="height:10px;font-size:75%;color:gray;" count="6">
    </ol>

    <div dojoType="dijit.form.HorizontalRule" container="topDecoration"
        count=6 style="height:5px;">
    </div>

    <div dojoType="dijit.form.HorizontalRule" container="bottomDecoration"
        count=5 style="height:5px;">
    </div>

    <ol dojoType="dijit.form.HorizontalRuleLabels" container="bottomDecoration"
        style="height:10px;font-size:75%;color:gray;">
        <li>green<br>tea</li>
        <li>coffee</li>
        <li>red<br>bull</li>
    </ol>
</div>
```

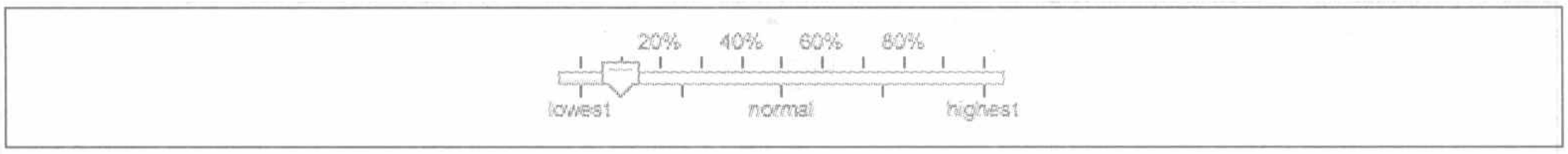



图 13-9：HorizontalSlider

## VerticalSlider

除了是沿着Y轴显示、要使用leftDecoration和rightDecoration代替topDecoration和bottomDecoration指定刻度尺和标签的container属性值，以及要按照垂直而非水平布局来应用样式之外，VerticalSlider的其他方面与HorizontalSlider都一样。例 13-16 展示了与例 13-15 相同的滑动条，只不过方向是垂直的。图 13-10 是最终结果。

例 13-16：VerticalSlider

```
<div dojoType="dijit.form.VerticalSlider" name="caffeine"
    value="100"
    maximum="175"
    minimum="2"
    showButtons="false"
    style="margin: 5px;width:75px; height: 300px;">

    <script type="dojo/method" event="onChange" args="newValue">
        console.log(newValue);
    </script>
    <ol dojoType="dijit.form.VerticalRuleLabels" container="leftDecoration"
        style="height:300px;width:25px;font-size:75%;color:gray;" count="6">
    </ol>

    <div dojoType="dijit.form.VerticalRule" container="leftDecoration"
        count=6 style="height:300px;width:5px;">
    </div>

    <div dojoType="dijit.form.VerticalRule" container="rightDecoration"
        count=5 style="height:300px;width:5px;">
    </div>

    <ol dojoType="dijit.form.VerticalRuleLabels" container="rightDecoration"
        style="height:300px;width:25px;font-size:75%;color:gray;">
        <li>green tea</li>
        <li>coffee</li>
        <li>red bull</li>
    </ol>
</div>
```

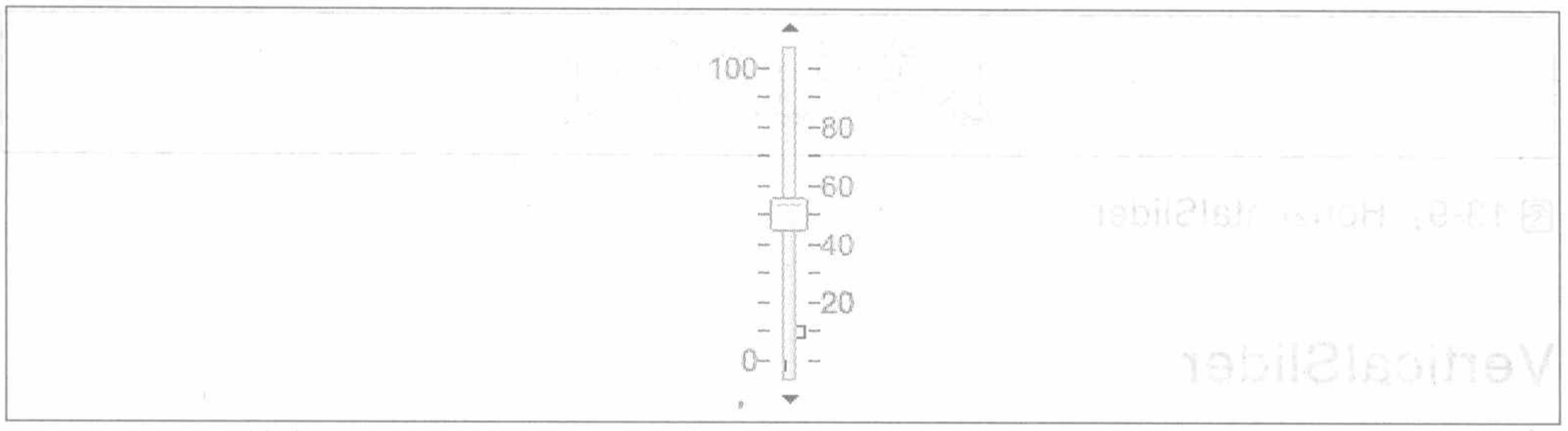



图 13-10：VerticalSlider

表 13-13、13-14 和 13-15 分别列出了 dijit.form.Slider（即两种滑动条、刻度尺和标签）的重要特性。记住，所有继承自表单的特性，如 setValue 等，都能照常使用。

表 13-13：HorizontalSlider 和 VerticalSlider 的特性

| 名称 | 类型 | 说明 |
|---|---|---|
| showButtons | 布尔值 | 用于指定是否显示滑动条部件两端的递增及递减按钮。默认值为 true |
| minimum | 整数 | 用于指定允许的最小值。默认值为 0 |
| maximum | 整数 | 用于指定允许的最大值。默认值为 100 |
| discreteValues | 整数 | 用于指定最小和最大值（不含）之间的非连续值的个数。Infinity 表示连续增减。大于 1 的值产生不连续的效果。默认值为 Infinity |
| pageIncrement | 整数 | 通过 Page Up 和 Page Down 键调整滑块的增量值。默认值为 2 |
| clickSelect | 布尔值 | 用于指定单击进度条是否将滑块移至相应位置，即导致部件值改变。默认值为 true |
| slideDuration | 数值 | 用于指定滑块从 0% 滑动到 100% 的毫秒数。用于以编程方式改变滑动条的值。默认值为 1000 |
| increment() | 函数 | 以 1 单位为步长增加滑动条的值 |
| decrement() | 函数 | 以 1 单位为步长减少滑动条的值 |

表 13-14：HorizontalRule 和 VerticalRule 的特性

| 名称 | 类型 | 说明 |
|---|---|---|
| ruleStyle | 字符串 | 用于指定为刻度线应用样式的 CSS 类 |
| count | 整数 | 用于指定生成的刻度线的数量。默认值为 3 |

表 13-14：HorizontalRule 和 VerticalRule 的特性（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| container | DOM Node | 用于指定相对于滑动条应用标签的位置：HorizontalSlider 使用 topDecoration 或 bottomDecoration，VerticalSlider 使用 leftDecoration 或 rightDecoration |

**注意：** HorizontalRuleLabel 和 VerticalRuleLabel 分别继承自 HorizontalRule 和 VerticalRule。

表 13-15：HorizontalRuleLabel 和 VerticalRuleLabel 的特性

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| labelStyle | 字符串 | 用于指定为文本标签应用样式的 CSS 类 |
| labels | 数组 | 要显示的文本标签数组，按从左到右，或从上到下的次序平均显示。默认值为[] |
| numericMargin | 整数 | 用于指定在滑动条两端应该省略的数字标签的个数（对于省略显而易见的起始或结尾值很有用处，例如，省略默认的 0 和 100。） |
| minimum | 整数 | 在没有指定标签数组的情况下，以这个值作为最左（下）端标签。默认值为 0 |
| maximum | 整数 | 在没有指定标签数组的情况下，以这个值作为最右（上）端标签。默认值为 1 |
| constraints | 对象 | 在没有指定标签数组的情况下，使用这个对象中包含的模式（来自 dojo.number）生成数字标签。默认值为 {pattern:"#%"} |
| getLabels | 函数 | 返回 labels 数组 |

# Form

虽然可以把表单部件包装在 HTML 的 form 标签内，但 dijit.form.Form 部件则提供了更加方便的功能。在本章最后一节中，我们先回顾一下普通的 HTML 表单，然后再介绍 dijit.form.Form 提供的新特性。许多刚刚接触 Dijit 的开发人员经常依照使用普通 HTML 的经验来使用 Dijit，而这恰恰是产生各种迷惑的根源。前面曾提到过，Dojo 背

后的一个设计思想就是不重复提供已有的Web特性；相反，Dojo致力于根据需求来补充和增强现有Web技术的不足或者不标准之处。

## HTML Form标签梗概

dijit.form.Form支持HTML4.01规范定义的标准表单属性。它假设所有属性值都以字符串形式被包含在引号中，即使对需要明确表示脚本操作的DOM事件（如onclick）也不例外，例如，onclick="javascript:someScriptAction()"或onclick="javascript:return someValidationAction()"。对于鼠标事件，假设“左键单击”。

## Form

Form部件通过提供方法和扩展点来补充了标准的HTML表单属性。调用Form部件的方法可以直接对部件执行操作，而重写它的扩展点则可以自定义如何在内部响应用户的操作。表13-16列出了Form的重要特性。

表13-16：Form的方法和扩展点

| 名称 | 类别 | 说明 |
|---|---|---|
| getValues() | 方法 | 返回JSON，包含键/值对形式的表单数据 |
| isValid() | 方法 | 如果表单中每个可用值的isValid方法返回true，那么此方法返回true |
| setValues(/*Object*/values) | 方法 | 通过每个键与表单中的每个字段一一对应的JOSN对象，一次性设置表单所有字段的值 |
| submit() | 方法 | 用于以编程方式提交表单 |
| reset() | 方法 | 系统地调用表单包含的每个部件的reset()方法，以重置它们的值 |
| onSubmit() | 扩展点 | 当submit()方法执行时在内部调用。这个扩展点的目的是用来以返回false的方式取消表单提交。在默认情况下，它返回isValid()返回的值 |
| validate() | 方法 | 如果表单有效则返回true，从某种意义上说与isValid一样。但是，它还能突出显示有效的表单部件，并在表单中包含的第一个无效部件上调用focus() |

把整个表单都封装到dijit.form.Form部件中，与使用其他表单部件代替对应元素一样，如图13-17所示。

例 13-17：典型的 Form 应用

```
<form id="registration_form" dojoType="dijit.form.Form">

  <!-- form elements go here -->

  <!-- override extension points as usual...-->

  <script type="dojo/method" event="onSubmit" args="evt">
    // 如果表单不应该被提交则返回 false。
    // 默认返回 dijit.form.Form 的 isValid() 方法返回的值
  </script>
</form>
```

# 小结

本章讨论了一些重要的基础知识。在学习完本章后，读者应该：

- 理解普通表单的工作原理。
- 理解如何使用表单部件代替标准的表单元素。
- 熟悉表单部件的总体分类方法，理解表单部件之间的继承关系。
- 能够在标记中或者以编程方式创建各种表单部件。
- 理解方法、属性及扩展点的区别。
- 理解可退化表单的含义，并且能够根据情况权衡如何实现可退化的设计。

下一章，我们将介绍布局部件。

# 第 14 章

# 布局部件

当前，为数众多的Web应用程序仍然要在实现和重构CSS布局上投入大量时间，而这些布局实际上早就屡见不鲜了，因此提高创建这些布局的效率势在必行。本章介绍Dijit中提供的、用于在标记中创建常见布局的实用容器 —— 布局部件。通过使用布局容器，设计人员无须考虑如何自定义CSS样式，也不必理会怎样计算相对位移，常见的标签页布局和任意组合的平面布局都可以一一信手拈来。与上一章介绍表单部件时不同，本章篇幅较短，内容相对简单，而且部件间的继承关系也更加分明。由于布局部件并不多，加之它们的配置项也比较少，因此掌握起来应该会很轻松。

## 布局部件的共同特性

所有布局部件都位于`dijit.layout`命名空间之下，且共享着屈指可数（因而也容易记住）的几个基本特性。除了都继承自`_Widget`、`_Conainer`和`_Contained`外，布局部件还有其他一些共同的特性。表14-1列出了这些共同的特性，这些特性都非常容易理解。

表 14-1：布局部件的公有方法

| 名称 | 说明 |
| --- | --- |
| `isLayoutContainer` | 返回部件是否为一个布局容器 |
| `layout()` | 由部件重写，以便对包含的内容（即子部件）进行缩放和定位。这个方法于`startup`执行后被调用，此时部件的内容盒子已经设置，而且部件的大小已经通过`resize`方法调整完毕 |

表 14-1：布局部件的公有方法（续）

| 名称 | 说明 |
| --- | --- |
| resize(/*Object*/ size) | 用于明确设置布局部件的大小；接收一个{w ： 整数，h: 整数，l ： 整数，t ： 整数}形式的对象参数，指定部件的左上角位置及宽度和高度(无论什么时候重写resize方法，都应该在重写的方法中调用layout方法，因为layout是处理被包含的子部件大小和位置的恰当场所) |

通过表14-1，最重要的是理解layout和resize之间的关系。应该说，resize用于改变部件大小,而为了对部件大小的变化做出反应,几乎总要在resize中调用layout以便调整其子部件的大小。一般而言，子节点不能设置自己的布局，它们的布局必须由父节点在layout中进行设置。因此，通常的模式就是startup生命周期方法首先调用resize，然后resize再调用layout。

布局部件充分利用了_Container和Contained提供的特性，因此表14-2也列出了这些特性。

表 14-2：布局部件的容器操作机制

| 名称 | 说明 |
| --- | --- |
| removeChild(/*Object*/ dijit) | 从容器中删除子部件(如果指定部件不是容器的子部件，或者容器不包含任何子部件，该方法静默地失败) |
| addChild(/*Object*/ dijit, /*Integer?*/ insertIndex) | 向容器中添加一个子部件，可以使用 insert-Index 指定子部件的位置 |
| getParent() | 布局容器中的子部件用它来取得父部件。返回一个部件实例 |
| getChildren() | 布局容器用它来枚举自己的子部件。返回一个部件实例的数组 |
| getPreviousSibling() | StackContainer的子类通过这个方法来引用前面的(即“左边的”)同辈部件。返回一个部件实例 |
| getNextSibling() | StackContainer的子类通过这个方法来引用后面的(即“右边的”)同辈部件。返回一个部件实例 |

## 以编程方式创建布局部件

正如在接下来的示例中所要看到的,以编程方式创建布局部件同样遵循创建部件的基本模式，即：第一个参数是用于初始化部件的属性，而第二个参数则是对布局部件所在的

源节点的引用。在创建布局部件之后，源节点的引用就变成了部件的domNode。这一过程全都发生在_Widget的create方法中，第12章在讨论部件的生命周期方法时曾介绍过create。但是，与以前我们介绍的其他部件不同，以编程方式创建布局部件时必须明确调用该部件的startup方法。因为布局部件经常会包含子部件，而通过startup方法可以针对该部件的子部件已经添加完成发出通知，以便布局可以继续进行。毕竟，让一个部件设置自己的布局，进而导致其他同辈部件反复重启布局过程是不明智的。为此，父部件的startup方法也会在每个子部件上调用startup方法，而这正是开始呈现部件的信号。

**注意：** 在实现父容器时，startup是子部件显示在屏幕上之前操作它们的最后机会。

## 键盘支持

与其他部件一样，布局部件也对键盘提供了强大的支持。因此，几乎在任何情况下都能使用“理所当然”的键。例如，在导航AccordionPane时，可以使用上、下箭头键，也可以使用Page Up和Page Down键。除了为实现部件的a11y目标而增添易访问性之外，对键盘的全面支持同样也能为用户带来美好的体验。

# ContentPane

ContentPane是最基本的布局部件，它直接继承了_Widget；从概念上讲，ContentPane就像iframe的一个超级变体，但它除了具备在页面中呈现任何HTML片段、通过XHR按需重新加载内容、呈现部件以及融入页面主题的能力之外，还带有各种特殊的功能（不仅仅是一个完整的文档）。尽管ContentPane本身也适用于不少场合，但它也经常被包含在其他部件（如TabContainer）中。

如例14-1所示，布局窗格（pane）最基本的用法并没有什么特别之处。

例14-1：在标记中创建ContentPane

```
<html>
    <head><title>Fun with ContentPane!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
```

```
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.layout.ContentPane");
        </script>

    </head>
    <body class="tundra">
        <div dojoType="dijit.layout.ContentPane">
            Nothing special going on here.
        </div>
    </body>
    </html>
```

**注意：** 在查看Dojo的源代码或者阅读其在线文档时，读者会发现还有一个LinkPane部件。随着时间推移，ContentPane在不断吸收LinkPane的功能。由于LinkPane提供的内置功能与ContentPane相比差别已经越来越小，因此很可能会被列入不推荐使用的部件名单中。

然而，只要为ContentPane提供一个指向服务器端URL的引用，就可以轻而易举地取得并在页面上呈现来自服务器的内容。假设有一个名为*foo*的服务器端URL，该URL返回一段文字。那么就可以像下面这样通过ContentPane来显示这段文字：

```
<div id="foo" preload="false"  dojoType="dijit.layout.ContentPane" href="foo">
```

在此，*foo*返回的可能只是一个简单字符串值，但该URL也可以返回一个部件，而返回的部件也能够在页面中被自动解析和呈现。只不过这个部件必须是已经通过dojo.require加载到页面中的。例如，假设该URL返回了部件 ，那么这个部件默认也会被插入到页面中。

表14-3总结了ContentPane支持的所有特性。

表14-3：ContentPane API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| href | 字符串 | 用于指定窗格应该加载的外部数据 |
| extractContent | 布尔值 | 如果为true，表示要将ContentPane取得的文档的BODY标签之间的内容提取出来放到窗格中。默认值为falses |
| parseOnLoad | 布尔值 | 如果为true，数据中返回的任何部件都将被自动解析和呈现。默认值为true |

表14-3：ContentPane API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| preventCache | 布尔值 | 与dojo.xhrGet的preventCache参数类似。如果为true，那么会在每次请求时传递一个额外的参数，以避免缓存。默认值为false |
| preload | 布尔值 | 用于强制窗格加载内容，即使在初始不可见的情况下。（如果节点应用了display:none样式，那么除非preload为true，否则不会加载内容。）默认值为false |
| refereshOnShow | 布尔值 | 用于表示每当窗格从隐藏状态转换为可见状态时，是否重新加载内容。默认值为false |
| loadingMessage | 字符串 | 在dijit.nls.loading中定义。用于在加载内容的过程中为用户提供一条提示消息。默认值为"Loading..." |
| errorMessage | 字符串 | 在dijit.nsl.loading中定义。用于在加载内容失败时为用户提供一个错误消息。默认值为"Sorry, an error occurred" |
| isLoaded | 布尔值 | 用于确定内容是否加载完成。对于经常更新的内容比较有用 |
| refresh() | 函数 | 用于强制窗格通过刷新重新加载内容 |
| setHref(/*String*/ href) | 函数 | 用于修改部件所要加载的外部内容的位置。如果preload为false，那么在部件再次可见之前不会开始下载内容 |
| setContent(/*String \| DOMNode \| NodeList */data) | 函数 | 用于明确设置窗格的本地内容 |
| cancel() | 函数 | 用于取消对内容的下载操作 |
| onLoad(/*Event*/evt) | 函数 | 扩展点，在加载事件（以及可选地解析部件）后调用 |
| onUnload(/*Event*/evt) | 函数 | 扩展点，在通过刷新（如setHref或setContent）清除现有内容之前调用 |
| onDownloadStart | 函数 | 扩展点，在下载开始之前调用。在默认情况下，返回一个可用于显示的表示正在下载的字符串 |

表 14-3：ContentPane API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| onContentError(/*Error*/ e) | 函数 | 扩展点，在发生 DOM 错误时调用。返回显示给用户的字符串 |
| onDownloadError(/*Error*/ e) | 函数 | 扩展点，在下载期间发生错误时调用。返回显示给用户的字符串 |
| onDownloadEnd() | 函数 | 扩展点，在下载完成时调用 |

在给出名为 foo 的节点的情况下，可以像下面这样以编程方式创建 ContentPane：

```
var contentPane = new dijit.layout.ContentPane({/* properties*/, "foo");
contentPane.startup(); //要养成总是调用 startup 方法的好习惯
```

由于 ContentPane 并非 _Container 的子类，因此 ContentPane 没有用来添加子部件的内置方法。不过，我们可以通过 ContentPane 的 domNode 引用，使用纯 JavaScript 的 API 来为它添加子节点。以前面示例中的内容窗格为例：

```
contentPane.domNode.appendChild(someOtherDijit.domNode);
```

## 更好地利用 ContentPane

像 JavaScript 这样纯解释、高动态的语言有一种非常吸引人的特性，即能够在需要时动态地扩展功能。前面曾提到过，如果要通过 ContentPane 取得部件，例如，Textarea，那么就必须首先将 Textarea 加载到页面中。为此，就要在页面头部区域加上一条 dojo.require("dijit.form.Textarea") 语句，而且这也适用于大多数情形。可是，如果我们想让 ContentPane 能够呈现服务器返回的任意部件该怎么办呢？

没问题 —— 如果客户端请求一个特定的部件（也许是通过 URL 的查询字符串），那么可以在响应更新 ContentPane 之前，动态地使用 dojo.require 加载该部件。如果无法预知服务器会返回什么部件，那问题就变得复杂了 —— 不过，也不是没有办法解决。可以先将 ContentPane 的 parseOnLoad 设置为 false，即不自动解析部件（由于相应的 dojo.require 语句尚未执行，因此会导致一个错误），然后通过 onLoad 扩展点找到部件节点，将它们加载到页面中，最后再手动来解析 ContentPane。

以下是一个标记示例：

— 待续 —

```
<div dojoType="dijit.layout.ContentPane"
     href="bar" parseOnLoad="false">
     <script type="dojo/method" event="onLoad">
        dojo.query("[dojoType]", this.domNode)
        .forEach(function(x) {
          var _resource = dojo.attr(x, "dojoType");
          dojo.require(_resource);
            //不要在加载模块之前解析
            var _interval = setInterval(function() {
                 if (eval(_resource)) { //这个对象存在吗?
                    clearInterval(_interval);
                    dojo.parser.parse(x.parentNode);
                 }
            }, 100);
        });
     </script>
</div>
```

注意：读者也许想知道为什么ContentPane不直接支持_Container提供的API？对于这个问题，非官方的答案是：当ContentPane由于某种原因包含子部件时，一般不需要它执行特殊操作。为ContentPane添加子部件的原因多种多样。不过，如果读者确实想让ContentPane支持 _Container 的 API，可以通过 dojo.mixin 或 dojo.extend 来实现。

# BorderContainer

注意：BorderContainer是Dojo1.1中引入的一个新布局部件。BorderContainer的引入直接导致了LayoutContainer和SplitContainer成为不推荐使用的部件，原因是BorderContainer本质上包含了这两个部件的功能。虽然在网上还能看到LayoutContainer和SplitContainer的示例，但最好不要在应用程序中再使用这两个不推荐的特性。正因为如此，本书也不介绍这两个部件。

BorderContainer为创建通常需要几个布局组件（tile）才能实现的布局提供了便捷方式，例如：“上/下/左/右/中”布局、“上/下/中”布局，或者“左/右/中”布局等。而且，这些布局中的组件还可以带有调整大小的手柄。从把繁琐的任务转化为极其简单的部件应用的角度来讲，可以说BorderContainer是一种具有很大附加价值的部件。也许有读者已经猜到了，之所以把这个部件称为“边框”容器（border container），就是因为可以有多达4个组件围绕在它的边框四周，而中间还可以填充其他内容。

表14-4列出了BorderContainer的API。

表 14-4：BorderContainer 的 API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| design | 字符串 | 可用的值包括 "headline"（默认）和 "sidebar"，用以决定是由上和下组件来扩展容器的高度，还是由左和右组件扩展容器的宽度 |
| liveSplitters | 布尔值 | 用以决定是在发生 onmouseup 事件时调整大小，还是在鼠标拖动时连续地调整大小 |
| persist | 布尔值 | 用以表示是否将分割后的位置信息保存在 cookie 中 |

在使用 BorderContainer 时，表 14-5 中列出的属性也是它所需要的，这些属性都通过 ContentPane 起作用。

**注意：** 所谓的这些属性通过 ContentPane 起作用，其实就是 BorderContainer 的资源文件在后台扩展了 _Widget 的原型，从而达到了包含这些值的目的。这个解决问题的方案很聪明，因为它利用JavaScript的动态能力在必要时提供额外的特性，从而不需要预备好的方案；否则，就会为 ContentPane 的实现增加不必要的耦合性。

表 14-5：BorderContainer 的子部件的可用属性

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| minSize | 整数 | 如果指定这个值，那么 ContentPane 的最小像素尺寸限制为该值。默认值为 0 |
| maxSize | 整数 | 如果指定这个值，那么 ContentPane 的最大像素尺寸限制为该值。默认值为 Infinity |
| splitter | 布尔值 | 如果指定这个值，ContentPane 的边缘会出现一个分割手柄，可以用它来调整内容区大小。默认值为 false，即不能调整内容区大小 |
| region | 字符串 | BorderContainer 的布局组件是 ContentPane 部件，因此每个 ContentPane 都应该有一个 region 属性来说明它在布局中的位置。可用的值包括 "top"、"bottom"、"left"、"right" 和 "center"。这个属性的默认值为一个空字符串。此外，在双向布局的情况下，"leading" 和 "trailing" 用以取代 "left" 和 "right" |

包含 top/bottom 组件、扩展容器高度的布局称为标题（headline）布局，而包含 left/right 组件、扩展容器宽度的布局则称为侧边栏（sidebar）布局。这两种布局都允许包含额外

的布局组件，以便将整个布局扩展至3～5个布局区域。在任何情况下，剩余的空间就是中央区域，由 center 组件填充。

例14-2展示了一个在标记中定义的简单的标题布局。布局上方是一个蓝色窗格，下方是一个红色窗格，而中部则保持白色背景。其中，上方窗格的最小高度是10像素，最大高度为100像素（即其默认高度）。

例 14-2：在标记中创建 BorderContainer

```
<html>
    <head><title>Fun with BorderContainer!</title>
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.layout.ContentPane");
            dojo.require("dijit.layout.BorderContainer");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body class="tundra">

        <div dojoType="dijit.layout.BorderContainer" design="headline"
            style="height:500px;width:500px;border:solid 3px;">
            <div dojoType="dijit.layout.ContentPane" region="top"
              style="background-color:blue;height:100px;" splitter="true"
                 minSize=10 maxSize=100>top</div>

            <div dojoType="dijit.layout.ContentPane" region="center">center</div>

            <div dojoType="dijit.layout.ContentPane" region="bottom"
              style="background-color:red;height:100px;" splitter="true">bottom</div>

        </div>
    </body>
</html>
```

再加入两个ContentPane部件，就可以为布局添加左、右两个侧边栏，如图14-1所示。修改后的 BODY 标签如下所示：

```
<body class="tundra">

        <div dojoType="dijit.layout.BorderContainer"
          design="headline" style="height:500px;width:500px;border:solid 3px;">

            <div dojoType="dijit.layout.ContentPane" region="top"
              style="background-color:blue;height:100px;" splitter="true"
                minSize=10 maxSize=100>top</div>

            <div dojoType="dijit.layout.ContentPane"
region="center">center</div>

            <div dojoType="dijit.layout.ContentPane" region="bottom"
              style="background-color:red;height:100px;" splitter="true">bottom</div>

                <div dojoType="dijit.layout.ContentPane" region="left"
                  style="background-color:yellow;width:100px;" splitter="true">left</div>

                <div dojoType="dijit.layout.ContentPane" region="right"
                  style="background-color:green;width:100px;" splitter="true">right</div>

        </div>
</body>
```

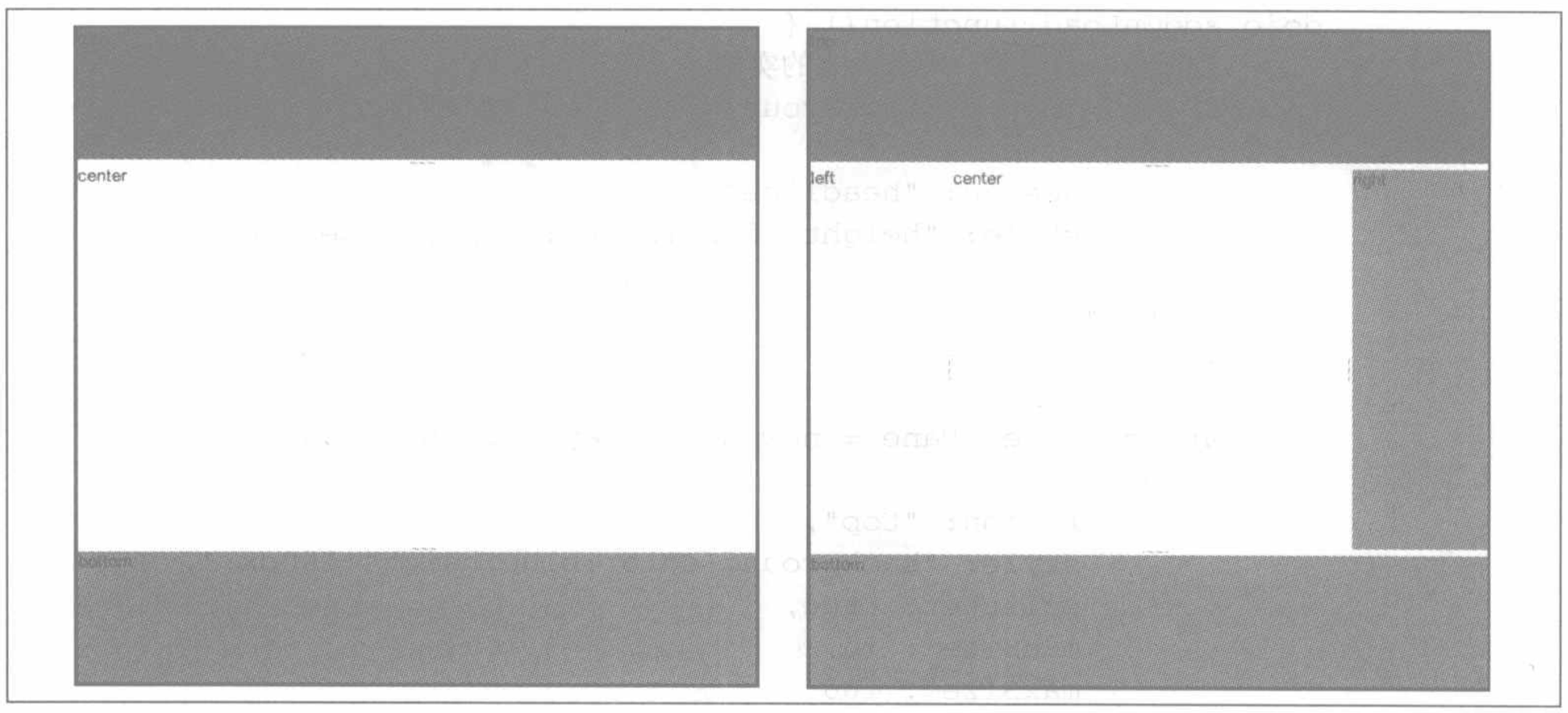



图 14-1：左：为 BorderContainer 添加左、右窗格之前；右：为 BorderContainer 添加左、右窗格之后

与其他所有部件一样，以编程方式创建 BorderContainer 同样也需要使用同名的构造函数，并为该构造函数传入一个属性集合和一个源节点。为 ContentPane 添加子部件则

需要依次创建和添加这些部件。虽然比在标记中创建BorderContainer更繁琐一些，但模式并未改变。例14-3展示了如何以编程方式创建例14-2所示的布局。

例14-3：以编程方式创建BorderContainer

```
<html>
    <head><title>Fun with BorderContainer!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            djConfig="parseOnLoad:true",
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js">
        </script>
        <script type="text/javascript">
            dojo.require("dijit.layout.BorderContainer");
            dojo.require("dijit.layout.ContentPane");
            dojo.require("dojo.parser");
            dojo.addOnLoad(function() {
                // 创建BorderContainer的实例
                var bc = new dijit.layout.BorderContainer(
                    {
                        design: "headline",
                        style: "height:500px;width:500px;border:solid 3px"
                    },
                    "bc"
                );

                var topContentPane = new dijit.layout.ContentPane(
                    {
                        region: "top",
                        style: "background-color:blue;height:100px;",
                        splitter: true,
                        minSize : 10,
                        maxSize : 100
                    },
                    document.createElement("div")
                );

                var centerContentPane = new dijit.layout.ContentPane(
                    {
                        region: "center"
                    },
```

```
        document.createElement("div")
);
var bottomContentPane = new dijit.layout.ContentPane(
    {
        region: "bottom",
        style: "background-color:red;height:100px;",
        splitter: true
    },
    document.createElement("div")
);

bc.startup(); // 一定要初始化布局（即使还不包含子部件）

// 下面添加子部件
        bc.addChild(topContentPane);
        bc.addChild(centerContentPane);
        bc.addChild(bottomContentPane);

    });
  </script>
 <head>
 <body class="tundra">
    <div id="bc"></div>
 </body>
</html>
```

---

**注意：** 前面这个示例中调用了startup()来初始化布局，然后又使用addChild方法来添加子部件。下面这些代码也可以达到同样目的：

```
bc.domNode.appendChild(topContentPane.domNode);
bc.domNode.appendChild(centerContentPane.domNode);
bc.domNode.appendChild(bottomContentPane.domNode);
bc.startup();
```

---

BorderContainer部件非常灵活，如有必要，还可以对它们随意嵌套。虽然在项目开发中通常应该优先考虑以CSS来实现布局，但使用BorderContainer来建立标题或侧边栏风格的布局，确实不费吹灰之力。

## StackContainer

StackContainer作为一个布局部件，每次只显示一系列布局组件中的一个。从概念上讲，StackContainer就像一套幻灯片，可以在一“套”部件组件中向前或向后翻页。与BorderContainer类似，可以为StackContainer添加任意数量的子部件，而

StackContainer会负责显示其中的子部件。StackContainer最基本的应用，就是像例14-4一样，生成一套可以翻页的布局组件。

例14-4：在标记中创建StackContainer

```
<div id="stack" dojoType="dijit.layout.StackContainer"
  style="width:100px; height:100px; margin:5px; border:solid 1px;">

    <div dojoType="dijit.layout.ContentPane">
        One fish...
    </div>
    <div dojoType="dijit.layout.ContentPane">
        Two fish...
    </div>
    <div dojoType="dijit.layout.ContentPane">
        Red fish...
    </div>
    <div dojoType="dijit.layout.ContentPane">
        Blue fish...
    </div>

</div>

<button dojoType="dijit.form.Button">&lt;
    <script type="dojo/method" event="onClick" args="evt">
dijit.byId("stack").back();
    </script>
</button>

<button dojoType="dijit.form.Button">&gt;
    <script type="dojo/method" event="onClick" args="evt">
dijit.byId("stack").forward();
    </script>
</button>
```

通常的容器及布局方法对StackContainer同样适用。此外，表14-6还列出了StackContainer支持的一些有用特性。

表14-6：StackContainer的API

| 名称 | 类型 | 说明 |
|---|---|---|
| doLayout | 布尔值 | 表示是否修改子部件当前显示的大小，从而与容器大小匹配。默认值为true |
| selectedChildWidget | 对象 | 引用当前选中的子部件。默认值为null |
| selectChild(/*Object*/page) | 函数 | 用于选择特定的子部件 |
| forward() | 函数 | 用于向前翻页到下一个子部件 |
| back() | 函数 | 用于向后翻页到上一个子部件 |

注意：StackContainer还支持几个额外的特性：

- closeChild(/*Object*/ child)方法
- onClose()扩展点
- 可以为其子部件指定closeable、title和selected属性
- 在添加、删除和选中子部件时，分别发布<id>-addChild、<id>-removeChild和<id>-selectChild主题

由于这些特性对于TabContainer（它继承自StackContainer）才是最常用的，因此下一节我们再详细介绍它们。

说到要以编程方式来创建StackContainer，相信读者一定十分熟悉例14-5中展示的代码。

例14-5：以编程方式创建TabContainer

```
var container = new dijit.layout.StackContainer({}, "foo");

var leftChild = new dijit.layout.ContentPane({});
leftChild.domNode.innerHTML="page 1";

var rightChild = new dijit.layout.ContentPane({});
rightChild.domNode.innerHTML="page 2";

container.addChild(leftChild);
container.addChild(rightChild);

container.startup();

/* 从第一页跳转到第二页…… */
dijit.byId("foo").forward();
```

## 拖延（亦称延迟加载）可能得到更好的性能

在前面的示例中，我们使用两个按钮来明确地实现了分页导航。事实上，对于由多页构成的应用程序，以StackContainer作为该程序的容器也不失为一个解决方案。例如，搜索应用程序在初始时可以只显示一个搜索框，当用户按下按钮触发搜索操作后，可以向前翻页并在下一页显示搜索结果。如果读者把应用程序的每个“页面”都作为StackContainer的一个子部件来定义，那么应用程序就具备了无须重新加载页面的优点——至少可以算作一种Web 2.0式的界面。

尽管在有些情况下，一次性加载应用程序的所有内容可能是有必要的，但通过指定ContentPane部件的href属性，则可以实现延迟加载内容。前面介绍过，延迟加载行

为由ContentPane的preload属性控制。如果preload属性的值为false（默认值），那么在ContentPane部件可见之前不会加载外部数据。例如，设置下面值为*blueFish*的URL（指向例14-4中的文本“Blue fish...”），会导致StackContainer的第四个页面延迟加载：

```
<div dojoType="dijit.layout.ContentPane" href="blueFish"></div>
```

延迟加载非常适合应用程序中那些不常用的基本功能。例如，参数窗格一般不会出现在应用程序的普通页面上，其中包含的大量选项都可以考虑延迟加载。

## TabContainer

应该说，TabContainer实际上只是StackContainer的一个特别版，二者之间的主要区别就是TabContainer本身就带有一组时尚的标签页（tab），可以通过它们直接控制显示哪个页面。因此，继承自StackContainer的TabContainer只提供了一些针对标签页列表的额外特性。例14-6展示了TabContainer的基本应用。

例14-6：在标记中创建TabContainer

```
<div dojoType="dijit.layout.TabContainer"
  style="width:225px; height:100px; margin:5px; border:solid 1px;">

    <div dojoType="dijit.layout.ContentPane" title="one">
        One fish...
    </div>

    <div dojoType="dijit.layout.ContentPane" title="two">
        Two fish...
    </div>

    <div dojoType="dijit.layout.ContentPane" title="red"
      closable=
    "true">Red fish...
        <script type="dojo/method" event="onClose" args="evt">
            console.log("Closing", this.title);
            return true; // 为确保关闭标签页，必须返回true
        </script>
    </div>

    <div dojoType="dijit.layout.ContentPane" title="blue">
        Blue fish...
    </div>
</div>
```

值得一提的是，每个标签页都是自动创建的；设计人员只要为TabContainer的每个子部件指定相应的title属性，一切就会在后台自动完成（大概是最好的方式）。另外，将

closable属性指定为true会为标签页添加一个关闭按钮（图标），而通过onClose扩展点则可以定义在关闭标签页时要执行的操作。注意，如果onClose不返回true，标签页不会被关闭。

表14-7列出了与TabContainer相关的特性。

表14-7：TabContainer的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| title | 字符串 | 由StackContainer混入_Widget中的属性。用于为标签页指定标题 |
| closeable | 布尔值 | 由StackContainer混入_Widget中的属性。用于指定标签页是否可以被关闭。当值为true时，标签页上会出现一个可以单击并关闭标签页的小图标。默认值为false |
| onClose() | 函数 | 扩展点，由StackContainer混入_Widget中，用于增强子部件在被关闭时的行为。默认的返回值为true |
| tabPosition | 字符串 | 用于指定标签页出现的位置。可用的值包括"top"（默认值）、"button"、"left-h"和"right-h" |
| <id>-addChild<br><id>-removeChild<br><id>-selectChild | dojo.publish主题 | 这个功能继承自StackContainer。当添加、删除或选中子部件时，会分别发布相应的主题。<id>表示TabContainer的id值 |

**注意：** TabContainer中的标签页都是dijit.form.Button的实例；因此，可以按照需要对这些标签页进行定制。

同样，只要将ContentPane的prload属性设置为false，就可以像在StackContainer中一样，在TabContainer中延迟加载内容。

不过，下面我们要展示的还是如何以编程方式创建TabContainer（见例14-7）

例14-7：*以编程方式创建*TabContainer

```
var container = new dijit.layout.TabContainer({
    tabPosition: "left-h",
    style : "width:200px;height:200px;"
}, "foo");
```

```
var leftChild = new dijit.layout.ContentPane({title : "tab1"});
leftChild.domNode.innerHTML="tab 1";

var rightChild = new dijit.layout.ContentPane({title : "tab2", closable:
true});
rightChild.domNode.innerHTML="tab 2";

container.addChild(leftChild);
container.addChild(rightChild);

container.startup();
```

## AccordionContainer

与TabContainer相似，AccordionContainer也继承自StackContainer；而且，也是一种每次只显示一个子部件的容器。只不过从视觉上来说，AccordionContainer在垂直方向上是可折叠的，并且在选中一个子部件时，其他子部件都会以动画形式折叠起来。

另外，在使用AccordionContainer方面也有一个很重要的区别，即必须通过名为AccordionPane的子容器来包含其子部件。至于为什么要这样做，主要还是与底层的实现有关，探究起来也没有太大意义。如果读者愿意的话，大可将AccordionPane当成ContentPane来使用。

---

**警告：** 到Dojo1.1为止，还不能在AccordionPane中嵌套布局部件，例如，SplitContainer；但是，对于其他类型的内容，则没有问题。

---

例14.8展示了如何在标记中创建AccordionPane。

例14-8：在标记中创建AccordionPane

```
<div id="foo" dojoType="dijit.layout.AccordionContainer"
  style="width:150px; height:150px; margin:5px">
    <div dojoType="dijit.layout.AccordionPane" title="one">
            <p>One fish...</p>
    </div>
    <div dojoType="dijit.layout.AccordionPane" title="two">
            <p>Two fish...</p>
    </div>
    <div dojoType="dijit.layout.AccordionPane" title="red">
            <p>Red fish...</p>
    </div>
    <div id="blue" dojoType="dijit.layout.AccordionPane" title="blue">
        <div dojoType="dijit.layout.ContentPane" href="blueFish"></div>
```

```
    </div>
</div>
```

说到API，AccordionContainer只比StackContainer多提供了一个额外属性，如表14-8所示。

表14-8：AccordionContainer的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| duration | 整数 | 用于指定通过滑动动画显示被选窗格的持续时间，单位为毫秒。默认值为250 |

尽管可以让感兴趣的读者自己练习以编程方式创建AccordionContainer，但此时的创建模式却稍有不同。如例14-9所示，原因在于AccordionPane本身也是一个部件。

例14-9：以编程方式创建AccordionPane

```
var container = new dijit.layout.AccordionContainer({}, "foo");

var child1 = dojo.doc.createElement("div");
child1.innerHTML="pane 1";

var content1 = dojo.doc.createElement("p");
content1.innerHTML = "content 1";

var ap1 = new dijit.layout.AccordionPane({title: "pane1", selected : true}, content1);
container.addChild(ap1);

var child2 = dojo.doc.createElement("div");
child2.innerHTML="pane 2";

var content2 = dojo.doc.createElement("p");
content2.innerHTML = "content 2";

var ap2 = new dijit.layout.AccordionPane({title: "pane2"}, content2);
container.addChild(ap2);

container.startup();
```

## 呈现与可见

在试验本章示例的过程中，读者大概已经注意到了布局是随页面加载而发生的。换句话说，读者刚一开始看到的是一些表现布局内容的文本，随后这些文本便突然魔术般地转换成了漂亮的布局。尽管也不是完全不能接受，但在某些情况下让用户看到呈现过程恐怕还是不太合适的。

对此，一种常见的解决方案是在开始时将页面主体设置为隐藏；然后，当页面加载完成时，再让全部内容一起可见。要实施这个方案也很简单，首先必须为页面的主体定义一个表示隐藏的样式（或类），例如，<body style="visibility:hidden;">就可以。然后，再在适当的时候通过脚本把页面主体设置为可见就行了。在初始隐藏整个页面主体的情况下，将dojo.style(dojo.body(), "visibility", "visible")添加到dojo.addOnLoad中就可以达到显示页面内容的目的。如果出于某些原因，需要在特定事件发生（例如，为某个自定义部件提供数据的异步调用事件）时再显示页面，那么可以将上述语句添加到相应的回调函数中。

**注意：** 在CSS中，visibility和display属性的差别在于是否占用屏幕空间。通常，应用visibility:hidden样式的节点在隐藏的同时仍然会占用空间；而应用display:none的节点则既不可见，也不占用空间——正因为如此，当通过display:block将内容切换为可见时，其下方的内容会发生明显的偏移。

另外一个值得注意的问题是，当在初始隐藏的元素中创建布局容器时，这些布局容器可能会无法显示。如果读者发现应该显示的布局容器没有显示，可以手动调用这些容器的resize()方法，以强制它们正确地呈现。根据以往的经验，当在dijit.Dialog中显示布局容器时，经常会出现这个问题。

**警告：** 在不是立即可见的环境下创建的布局部件有可能不会正确地被呈现，但此时只要调用布局容器的resize()方法，基本上就可以使它们得到正确的呈现。

## 小结

在学习完本章之后，读者应该能够：

- 认识到布局部件能够解决的常见布局问题（例如，标签页式布局）。
- 理解各种布局部件提供的基本特性。
- 在标记中或者以编程方式通过布局部件实现各种布局。
- 使用BorderContainer创建包含多个区域的灵活的可缩放布局。
- 使用ContentPane延迟加载内容，无论是单独使用，还是将其作为其他部件的子部件使用。

- 使用 StackContainer 和 TabContainer 在应用程序中实现分页显示数据。
- 理解为什么要将部件设置为初始隐藏，这样做的问题及解决方案。
- 理解 AccordionPane 在嵌入布局部件时存在的限制。
- 理解 _Container 和 _Contained 这两个基类在布局部件中扮演的角色。

下一章，我们将讨论应用程序部件。

# 第15章

# 应用程序部件

本章系统地介绍Dijit中所有通用的应用程序部件。从很多方面来看，应用程序部件都是工具箱中最令人兴奋的部件。它们既不像表单部件那样为人熟知，也不同于各种布局部件，它们的特点是提供了数量极大的交互性功能。ProgressBar、Toolbar、Editor和Tree只是其中几个激动人心的部件而已。相信读者在本章中一定会发现不少曾为之倾心过的高水平DHTML杰作——特别是在本章最后部分。

**注意：** 尽管可能没有完全体现出来，但本章的部件与其他所有Dijit部件一样，同样都具有充分的易访问性。

## Tooltip

提示条（tooltip）是为页面上的控件提供辅助信息的常用手段。虽然从1990年左右开始，Web应用程序都依赖于HTML中的title属性，但当前的Web开发领域则呼唤功能更丰富的提示条。Tooltip部件本质上就是将提示条只能显示纯文本的能力扩展为能够显示任意HTML标记。在前面章节学习ValidationTextBox及其子部件的时候，读者大概已经提前目睹了Tooltip的风采。不过，也许读者还不知道，Tooltip也可以单独使用。

例15-1展示了Tooltip的某些重要特性，生成的结果如图15-1所示。

例15-1：典型的Tooltip应用

```
One <span id="one">fish</span>, two <span id="two">fish</span>.

<div dojoType="dijit.Tooltip" connectId="one,two">
```

```
    A limbless cold-blooded vertebrate...<img src='./fish.jpeg'/>
</div>
```

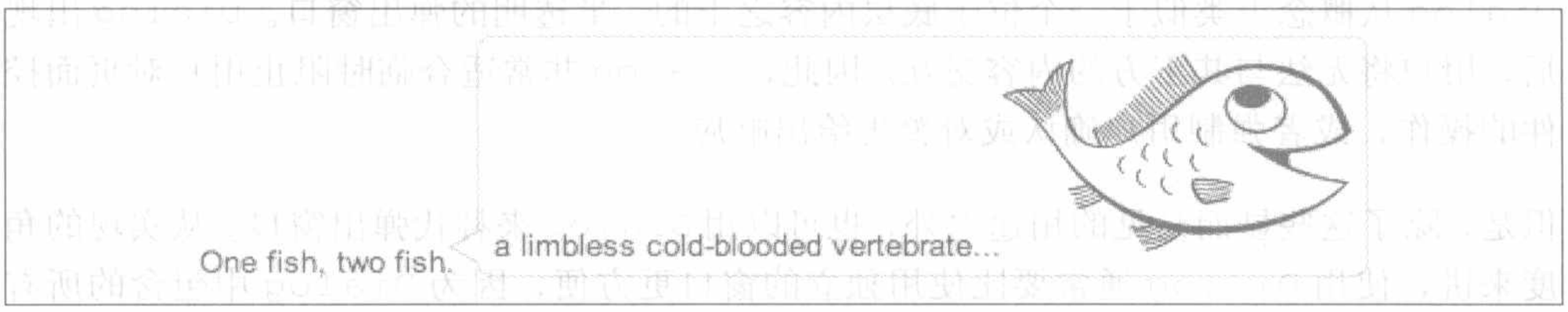



图 15-1：当鼠标放在任何一个包含“fish”的标签上时都会出现这个提示条

> **警告：** 为connectId传入多个值的语法与常规的JavaScript数组语法不同。不需要为传入的多个值加方括号，也不需要为单个字符串值加引号，即：connectId="id1,id2"。这种语法有望在将来得到规范化，以便消除这种例外。

在例 15-1 中，我们注意到 Tooltip 可以呈现任何 HTML 标记。Tooltip 应该用于显示只读内容；下一节将要介绍的TooltipDialog则非常适合显示输入字段或交互按钮。表 15-1 列出了 Tooltip 的全部特性。

表 15-1：Tooltip 的 API

| 名称 | 类型 | 说明 |
|---|---|---|
| connectId ("") | 字符串 | 以逗号分隔的值的列表，用于指定应该显示Tooltip的节点的 id 值 |
| label ("") | 字符串 | 用于指定在Tooltip 中显示的文本。尽管这个标签（label）可以包含任意 HTML 标记，但最好还是将 HTML 标记包含在封装的标记中 |
| showDelay (400) | 整数 | 用于指定将Tooltip 显示给用户之前需要等待的毫秒数 |

# Dialog 部件

Dijit提供了两个具有对话框（dialog）功能的相关部件：Dialog和TooltipDialog。其中，Dialog与我们常见的网页对话框（alert box）类似（只不过看起来更美观也更灵活）；TooltipDialog则更像是提示条，但它能够呈现其他部件并提供比提示条更多的交互功能。

## Dialog

Dialog从概念上类似于一个位于底层内容之上的、半透明的弹出窗口。Dialog出现后，用户将无法与其下方的内容交互。因此，Dialog非常适合临时阻止用户对页面控件的操作，或者强制用户确认或对警告给出响应。

但是，除了这些显而易见的用途之外，也可以用Dialog来替代弹出窗口。从实现的角度来讲，使用Dialog通常要比使用独立的窗口更方便，因为Dialog中包含的所有内容都能够通过当前页面的DOM访问（注1）。可以像对页面中的其他内容一样，对Dialog中的内容执行查询或其他操作，即使Dialog处于不可见的状态。

Dialog中可以包含任何DOM内容，无论是简单的HTML片段，还是复杂的布局部件，甚至自定义的部件。例15-2展示了Dialog的最基本用法；在这个示例中，Dialog会随页面加载自动显示。

---

**警告：** 上一章曾经提到过，对于在隐藏的容器中创建的布局部件，一般都需要手动调用该布局部件的resize方法强制它重新绘制自己，才能使它正确地显示出来。同样，如果创建并将布局部件嵌入到一个Dialog中，也需如此。

---

例15-2：典型的Dialog用法

```
<html>
    <head>
        <title>Fun With Dialog!</title>
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true">
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.Dialog");
            dojo.addOnLoad(function() {
```

---

注1： 事实上，某些浏览器根本不允许在一个窗口中操作另一个窗口中的DOM——即使这两个窗口同源也不行。

```
            dijit.byId("dialog").show();
        });
    </script>
</head>
<body class="tundra">
    <div id="dialog" dojoType="dijit.Dialog">
        So foul and fair a day I have not seen...
    </div>
</body>
</html>
```

使用Dialog的setContent方法（以一个DOM节点为参数），也可以以编程方式方便地创建Dialog部件。下面这个示例将会强迫用户单击位于Dialog中的Button，即使已经明确告诉用户不要那么做（译注1）：

```
dojo.addOnLoad(function() {
    var d = new dijit.Dialog();

    //向用户隐藏原有的关闭按钮……
    dojo.style(d.closeButtonNode, "visibility", "hidden");

    var b = new dijit.form.Button({label: "Do not press this button"});
    var handle = dojo.connect(b, "onClick", function() {
        d.hide();
        dojo.disconnect(handle);
    });
    d.setContent(b.domNode);
    d.show();
});
```

---

注意：Dialog的模板中包含许多有用的附着点（attach point），包括前面示例中使用的closeButtonNode。closeButtonNode附着点在这里被用来隐藏正常情况下可见的、关闭Dialog的图标。

---

与其他部件相似，Dialog也有很多共有的方法和属性；不过，表15-2则列出了Dialog的其他特性。

---

注意：Dialog继承自ContentPane，因此ContentPane的所有属性、方法和扩展点对于Dialog都是可用的。具体API请参见表14-3。

---

译注1：“Do not press this button”。

表15-2：在ContentPane的API基础上实现的Dialog API

| 名称 | 类型 | 说明 |
|---|---|---|
| open | 布尔值 | 返回对话框的状态。对话框打开时值为true，否则，值为false（默认值） |
| duration | 整数 | 用于指定对话框淡入/淡出的持续时间，以毫秒为单位。默认值为400 |
| duration() | 函数 | 用于隐藏对话框 |
| layout() | 函数 | 用于定位对话框及其下方内容 |
| show() | 函数 | 用于显示对话框 |

## TooltipDialog

TooltipDialog继承自Dialog，但它提供的功能多少有些类似于DropDownButton外部的菜单，除了能够在其中进行编辑操作。事实上，当前的TooltipDialog必须被包含在DropDownButton或ComboButton内才能体现出其作用，尽管从理论上讲，通过调整样式可以让按钮看起来不像按钮。说到这，读者可能会想起电子表格软件中常见的TooltipDialog的概念（译注2）。

**警告：** 要加载TooltipDialog，必须使用dojo.require("dijit.Dialog")，因为TooltipDialog包含在Dialog的资源文件中。

除了不能以编程方式单独创建并显示TooltipDialog之外，TooltipDialog其他的特性都与Dialog非常相似。但是，TooltipDialog不支持show()方法。此外，TooltipDialog还提供了一个标准的title属性，以便设计人员通过它来实现易访问性。

TooltipDialog的典型应用莫过于为图像就地加标签（tagging）了。为此，可以通过DropDownButton的iconClass属性提供一幅图像，然后当用户单击该图像时以交互方式显示TooltipDialog。下面的代码片段示范了为图像提供就地加标签功能的基本过程，该段代码产生的结果如图15-2所示。

```
<!-- somewhere out there...
<style type="text/css"> .customImage {
                background-image : url('/static/path/to/apple.jpeg');
                backgrond-repeat : no-repeat;
```

译注2：估计类似于为单元格添加的批注，因为可以直接在批注框中编辑内容。

```
                width : 120px;
                height : 120px;
            }
</style>
-->

<button dojoType="dijit.form.DropDownButton" iconClass="customImage"
  showLabel="false">
    <span>This label is hidden...</span>

    <div dojoType="dijit.TooltipDialog">
        <span>Tag this image...<span>
        <div dojoType="dijit.form.TextBox"></div>
    </div>
</button>
```

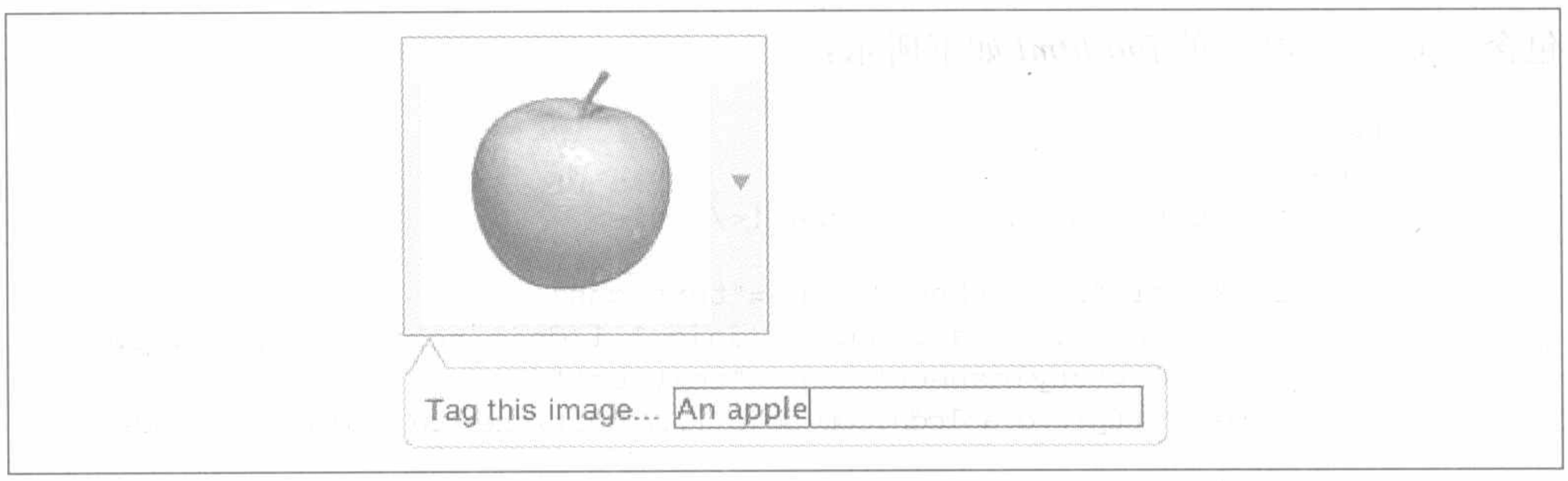



图 15-2：通过 DropDownButton 和 TooltipDialog 创建的为图像就地加标签的功能

# ProgressBar

ProgressBar部件的行为与应用程序中常见的进度条（progress bar）没有什么不同，而且还提供了确定型和非确定型两个变体。有关 ProgressBar，要介绍的内容实在不多。实际上，例 15-3 完全可以说明 ProgressBar 的应用。

例 15-3：典型的非确定型 ProgressBar 的应用

```
<div dojoType="dijit.ProgressBar" indeterminate="true" style="width:300px"></div>
```

当然，如果能够从服务器上获得实时的更新数据，也可以通过ProgressBar来显示实际的进度。假设有一个服务器端例程，该例程能够以某种形式返回进度信息。那么，可以通过下面的代码来模拟这个例程：

```
import cherrypy

config = {
    #serve up this static file...
    '/foo.html' :
```

```
        {
            'tools.staticfile.on' : True,
            'tools.staticfile.filename' : '/absolute/path/to/foo.html'
        }
    }

    class Content:
        def _  _init_  _(self):
            self.progress = 0

        @cherrypy.expose
        def getProgress(self):
            self.progress += 10
            return str(self.progress)

    cherrypy.quickstart(Content(), '/', config=config)
```

包含 ProgressBar 的 *foo.html* 如下所示：

```
    <html>
        <head>
            <title>Fun with ProgressBar!</title>

            <link rel="stylesheet" type="text/css"
              href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
            <link rel="stylesheet" type="text/css"
              href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

            <script
                type="text/javascript"
                src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
                djConfig="parseOnLoad:true">
            </script>

            <script type="text/javascript">
                dojo.require("dojo.parser");
                dojo.require("dijit.ProgressBar");

                dojo.addOnLoad(function() {
                    var progressInterval = setInterval(function() {
                        dojo.xhrGet({
                            url : "http://localhost:8080/getProgress",
                            load : function(response, ioArgs) {
                                console.log("load", response);
                                if (response <= 100) {
                                    dijit.byId("pb").update({progress : response});
                                }
                                else  {
                                    clearInterval(progressInterval);
                                }
                            }
                        });
```

```
                }, 1000);
            });
        </script>
    </head>
    <body style="padding:100px" class="tundra">
        <div>Loading...</div>
        <div id="pb" dojoType="dijit.ProgressBar" style="width:300px"></div>
        </div>
    </body>
</html>
```

简言之，addOnLoad每过一秒钟就会检查一次/getProgress URL，以便获得更新信息并将该信息通过update函数传给ProgressBar。JavaScript函数setInterval与ProgressBar是一种典型的搭配用法。

---

**警告：** 不要混淆setInterval和setTimeout。前者根据设定的时间间隔反复执行函数，后者只在经过指定的时间后执行一次函数。

---

表15-3展示了ProgressBar的全部特性。

**表15-3：ProgressBar的API**

| 名称 | 类型 | 说明 |
|---|---|---|
| indeterminate | 布尔值 | 是显示非确定型的进度条（即一幅动画图像），还是通过update函数呈现实际的进度 |
| maximum | 浮点数 | 最大值。虽然通常是0～100（默认范围值），但可以使用任何范围值 |
| places | 数值 | 对于确定型进度条，用于指定显示的小数位数。默认值为0 |
| progress | 字符串 | 进度条的初始值。也可以提供一个百分比值，如“50%”，来表示相对数量 |
| update(/*Object*/progress) | 函数 | 用于更新进度信息。在更新期间，可以传入progress、maximum和determinate来配置进度条 |
| onChange() | 函数 | 扩展点，在每次更新进度之后被调用 |

最后，如果需要在阻塞状态下显示ProgressBar，那么可以将ProgressBar放到一个Dialog中，以便用户等待后台操作完成。相关的示例代码如下：

```
var pb = new dijit.ProgressBar;
var d = new dijit.Dialog;
```

```
d.setContent(pb.domNode);
d.show();
```

# ColorPalette

ColorPalette是另外一种可以独立使用的部件，它能够以更形象和更具有交互性的方式为用户选择颜色提供便利，特别是在允许用户自定义应用程序主题的情况下。调色板（color palette）默认有两种既定的尺寸：3 × 4和7 × 10，同时预先选中了流行的Web颜色。

注意：有读者可能会奇怪，为什么不让用户自己来配置调色板中的颜色呢？其实，我们看到的调色板是图像，而非基于HTML标记生成的小方格。之所以这样来设计，是出于满足a11y的考虑，尽管还不够理想。因此，如果读者想扩展ColorPalette以显示定制的选项，（当然没问题），那么，恐怕就要阅读源代码并动手修改某些私有属性了。

在标记中使用ColorPalette非常简单；请看下面示例：

```
<div dojoType="dijit.ColorPalette">
    <script type="dojo/method" event="onChange" args="selectedColor">
        /* 或者，隐藏调色板？ */
        console.log(selectedColor);
    </script>
</div>
```

与ProgressBar相似，ColorPalette也是一个简单的独立部件。表15-4展示了ColorPalette支持的全部特性。

表15-4：ColorPalette的API

| 名称 | 类型 | 说明 |
|---|---|---|
| defaultTimeout | 整数 | 在自动击键（typematic）之前需要按多长时间的键，以毫秒为单位。默认值为500 |
| timeoutChangeRate | 数值 | 改变自动击键速度的时间。值为1.0表示按常规时间间隔自动击键，小于1.0的值表示以加速度自动击键。默认值为0.9 |
| palette | | 字符串 调色板的尺寸，必须是7x10（默认值）或3x4 |
| onChange(/*String*/ hexColor) | 函数 | 扩展点，在选中颜色时被调用 |

以编程方式创建ColorPalette也很简单：

```
var cp = new dijit.ColorPalette({/*属性*/});
/* 然后，将调色板放到页面中……*/
dojo.body().appendChild(cp.domNode);
```

# Toolbar

想必读者对工具条（Toolbar）都非常熟悉，它通过将常用命令集中到一起为用户提供便利。简单地说，Toolbar只是为一组Button部件提供容器而已。通过为Toolbar中的每个Button应用样式，可以让它们个个看起来都很漂亮。Dijit内置的多种主题都为常见的操作（如剪切/粘贴、粗体/斜体等等）提供了相应的类，可以将这些类直接应用给Button的iconClass属性。

下面的示例代码展示了如何在页面中创建Toolbar，并将每个按钮都与一个自定义的事件处理程序建立关联。

---

**注意：** 这个特殊的示例尝试自动为按钮添加处理程序。注意，在forEach块中连接到x.parentNode，而不是连接到x，原因在于Button部件的实现。换句话说，单击图标后接收到单击事件，实际上是包含该图标的节点；在Firebug控制台中可以检测到这一点。

---

```
<html>
    <head>
        <title>Fun with Toolbar!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true,isDebug:true"
        </script>

        <script type="text/javascript">
            dojo.require("dojo.parser");
            dojo.require("dijit.Toolbar");
            dojo.require("dijit.form.Button");

            dojo.addOnLoad(function() {
                var bold = function() {console.log("bold");}
                var italic= function() {console.log("italic");}
                var underline = function() {console.log("underline");}
                var superscript = function() {console.log("superscript");}
                var subscript = function() {console.log("subscript");}
```

```
            dojo.query(".dijitEditorIcon").forEach(function(x) {
                    if (dojo.hasClass(x, "dijitEditorIconBold"))
                        dojo.connect(x.parentNode, "onclick", bold);
                    else if (dojo.hasClass(x, "dijitEditorIconItalic"))
                        dojo.connect(x.parentNode, "onclick", italic);
                    else if (dojo.hasClass(x, "dijitEditorIconUnderline"))
                        dojo.connect(x.parentNode, "onclick", underline);
                    else if (dojo.hasClass(x, "dijitEditorIconSubscript"))
                        dojo.connect(x.parentNode, "onclick", superscript);
                    else if (dojo.hasClass(x, "dijitEditorIconSuperscript"))
                        dojo.connect(x.parentNode, "onclick", subscript);
                });
        });
    </script>
</head>
<body style="padding:100px" class="tundra">
    <div dojoType="dijit.Toolbar" style="width:175px">
        <button dojoType="dijit.form.Button"
            iconClass="dijitEditorIcon dijitEditorIconBold" ></button>
        <button dojoType="dijit.form.Button"
            iconClass="dijitEditorIcon dijitEditorIconItalic" ></button>
        <button dojoType="dijit.form.Button"
            iconClass="dijitEditorIcon dijitEditorIconUnderline" ></button>

        <span dojoType="dijit.ToolbarSeparator"></span>
        <button dojoType="dijit.form.Button"
            iconClass="dijitEditorIcon dijitEditorIconSubscript"></button>
        <button dojoType="dijit.form.Button"
            iconClass="dijitEditorIcon dijitEditorIconSuperscript"></button>
    </div>
</body>
</html>
```

有意思的是，Dijit主题中定义了大量与Editor相关的图标类（在主题的样式表中定义）。这些图标类可以直接在Toolbar中使用。（Editor部件将在后面详细讨论。）

Toolbar有一套简单的API，如图15-5所示，这表明了它是_Container的子类。

表15-5：Toolbar的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| addChild(/*Object*/ child, /*Integer?*/ insertIndex) | 函数 | 用于向工具条中插入部件 |
| getChildren() | 函数 | 返回工具条中包含的部件的数组 |
| removeChild(/*Object*/ child) | 函数 | 用于从工具条中删除部件（只删除其domNode，并非销毁该部件；如果要销毁该部件，必须手动调用destroyRecursive()） |

# Menu

Menu用于模拟右击桌面上的图标时出现的上下文菜单。Menu中可以包含MenuItem或PopupMenuItem部件。MenuItem是可以选择的真实菜单项；而PopupMenuItem则用于提供另一种层次的菜单项（类似Windows的“开始”菜单）。PopupMenuItem的子部件又是Menu。虽然从易用性的角度上看也许不够明智，但理论的确可以嵌套任意数量的PopupMenuItem和MenuItem部件。

首先，我们来看一个只包含MenuItem子部件的简单Menu，如例15-4所示。

**注意：** 通过执行dojo.require("dijit.Menu")可以同时加载MenuItem和PopupMenuItem。

## 通过图像切割实现延迟时间最小化

为了避免客户端与服务器之间过多的同步请求占用HTTP资源，Dijit主题将大量小图像并列组合为一幅图像。然后，再像下面这样通过CSS样式来切割出需要显示的小图像：

```
.tundra .dijitEditorIcon
/* 所有内置的Editor图标都包含这个类 */
{
        background-image: url('images/editor.gif');
        background-repeat: no-repeat;
        width: 18px;
        height: 18px;
        text-align: center;
}
/* 像下面这样切割出个别的图标…… */
.tundra .dijitEditorIconUnderline { background-position: -648px; }
```

下面的列表给出了其他可用的Editor图标类：

- dijitEditorIconSep
- dijitEditorIconBackColor
- dijitEditorIconBold
- dijitEditorIconCancel
- dijitEditorIconCopy

— 待续 —

- dijitEditorIconCreateLink
- dijitEditorIconCut
- dijitEditorIconDelete
- dijitEditorIconForeColor
- dijitEditorIconHiliteColor
- dijitEditorIconIndent
- dijitEditorIconInsertHorizontalRule
- dijitEditorIconInsertImage
- dijitEditorIconInsertOrderedList
- dijitEditorIconInsertTable
- dijitEditorIconInsertUnorderedList
- dijitEditorIconItalic
- dijitEditorIconJustifyCenter
- dijitEditorIconJustifyFull
- dijitEditorIconJustifyLeft
- dijitEditorIconJustifyRight
- dijitEditorIconLeftToRight
- dijitEditorIconListBulletIndent
- dijitEditorIconListBulletOutdent
- dijitEditorIconListNumIndent
- dijitEditorIconListNumOutdent
- dijitEditorIconOutdent
- dijitEditorIconPaste
- dijitEditorIconRedo
- dijitEditorIconRemoveFormat

—待续—

- dijitEditorIconRightToLeft
- dijitEditorIconSave
- dijitEditorIconSpace
- dijitEditorIconStrikethrough
- dijitEditorIconSubscript
- dijitEditorIconSuperscript
- dijitEditorIconUnderline
- dijitEditorIconUndo
- dijitEditorIconWikiword
- dijitEditorIconToggleDir

例 15-4：典型的 Menu 应用

```
<body class="tundra">
    <!-- 右击此处将出现上下文菜单 -->
    <div id="context" style="background:#eee; height:300px; width:300px;"></div>

    <div dojoType="dijit.Menu" targetNodeIds="context" style="display:none">
        <div dojoType="dijit.MenuItem">foo
            <script type="dojo/method" event="onClick" args="evt">
                console.log("foo");
            </script>
        </div>
        <div dojoType="dijit.MenuItem">bar
            <script type="dojo/method" event="onClick" args="evt">
                console.log("bar");
            </script>
        </div>
        <div dojoType="dijit.MenuItem">baz
            <script type="dojo/method" event="onClick" args="evt">
                console.log("baz");
            </script>
        </div>
    </div>
</body>
```

**警告：** 与Tooltip相似，为Menu传入的值是一个逗号分隔的字符串，不包含JavaScript数组语法中的一对方括号。以后的 Dojo 版本可能会将这种不规则的语法规范化。

通过例15-4可以看出，在标记中创建Menu很简单。代码中加粗的部分将display属性设置为none很关键，否则Menu一开始就将是可见的。

下面，假设我们想让baz成为一个PopupMenuItem，同时还希望Menu成为整个窗口的上下文菜单。那么，可以做如下修改：

```
<div dojoType="dijit.Menu" style="display:none" contextualMenuForWindow="true">
    <div dojoType="dijit.MenuItem">foo
        <script type="dojo/method" event="onClick" args="evt">
            console.log("foo");
        </script>
    </div>
    <div dojoType="dijit.MenuItem">bar
        <script type="dojo/method" event="onClick" args="evt">
            console.log("bar");
        </script>
    </div>
    <div dojoType="dijit.PopupMenuItem">
        <span>baz</span>
        <div dojoType="dijit.Menu">
          <!-- 按需要为每一项定义onClick处理程序 -->
            <div dojoType="dijit.MenuItem">yabba</div>
            <div dojoType="dijit.MenuItem">dabba</div>
            <div dojoType="dijit.MenuItem">doo</div>
        </div>
    </div>
</div>
```

由此可见，创建PopupMenuItem过程中唯一不同的地方，就是需要明确地定义一组节点。而且，其中的第一个节点很特殊，因为它用于定义标题。

本节最后，我们通过表15-6给出Menu其余的API。注意，作为_Container的子类，Menu与Tooltip一样，都有标志性的添加、删除和获取子部件的方法。MenuItem和PopupMenuItem的API如表15-7所示。

表15-6：Menu的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| contextMenuForWindow | 布尔值 | 如果值为true，那么右击窗口的任何地方都可以打开菜单。如果值为false，那么应该通过targetNodeIds属性指定一个或多个用来触发菜单的节点。默认值为false |

表 15-6：Menu 的 API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| popupDelay | 整数 | 当右击事件发生时，需要等多少毫秒才能显示上下文菜单。（在这段时间结束前，如果发生另一次右击事件，那么时间周期重置并重新开始计时。）默认值为500 |
| targetNodeIds | 数组 | 用于指定能够触发菜单的一组节点的 id 值。默认值为[] |
| parentMenu | 对象 | 如果有的话，是一个指向父菜单的引用。默认值为 null |
| addChild(/*Object*/ child, /*Integer?*/ insertIndex) | 函数 | 用于向菜单中插入一个部件 |
| getChildren() | 函数 | 返回菜单中包含的部件的数组 |
| removeChild(/*Object*/ child) | 函数 | 用于从菜单中删除部件（只删除其 domNode，但不销毁部件；要销毁部件，必须手动调用 destroyRecursive()。） |
| bindDomNode(/*String\|DOMNode*/node) | 函数 | 用于将菜单添加到特定的节点（对于上下文菜单比较有用） |
| unBindDomNode(/*String\|DOMNode*/node) | 函数 | 解除菜单与特定节点之间的绑定 |
| onClick(/*Object*/item, /*Event*/evt) | 函数 | 扩展点，用于处理单击事件 |
| onItemHover(/*MenuItem*/item) | 函数 | 当鼠标位于 MenuItem 上时调用 |
| onItemUnhover(/*MenuItem*/item) | 函数 | 当鼠标离开 MenuItem 时调用 |
| onCancel() | 函数 | 扩展点，用于处理用户取消当前菜单的操作 |
| onExecute() | 函数 | 扩展点，用于处理用户执行当前菜单的操作 |

表 15-7：MenuItem 和 PopupMenuItem 的 API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| label | 字符串 | 用于指定出现在MenuItem 中的文本 |
| iconClass | 字符串 | 用于让MenuItem 显示为图标的CSS 类（使用 CSS 定义的背景图像） |

表 15-7：MenuItem 和 PopupMenuItem 的 API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| disabled | 布尔值 | 用于指定MenuItem 是否无效。默认值为 false |
| setDisabled(/*Boolean*/ value) | 函数 | 用于以编程方式控制MenuItem 是否无效 |
| onClick(/*DomEvent*/ evt) | 函数 | 用于为MenuItem添加单击事件处理程序 |

# TitlePane

TitlePane 部件始终只显示标题，但是该部件的主体部分可以按照需要扩展或折叠；部件大小的调整是通过擦入 / 擦出（wipe-in/ wipe-out）动画实现的。虽然本节没有明确提及，但 TitlePane作为 ContentPane的子部件，一样也能够访问相应的方法并加载远程内容。（相关内容请读者参考前一章中对 ContentPane 的介绍。）例 15-5 展示 TitlePane 的基本应用。

例 15-5：典型的 TitlePane 的应用

```
<div dojoType="dijit.TitlePane" title="Grocery list:" style="width:300px">
    <ul>
        <li>Eggs</li>
        <li>Milk</li>
        <li>Bananas</li>
        <li>Coffee</li>
    </ul>
</div>
```

表 15-8 列出 TitlePane 支持的特性。

表 15-8：TitlePane 的 API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| title | 字符串 | 窗格的标题 |
| open | 布尔值 | 用于指定窗格打开还是关闭。默认值为true |
| duration | 整数 | 擦除动画经历的毫秒数。默认值为 250 |
| setContent(/*DomNode\|String*/) | 函数 | 用于以编程方式设置窗格的内容 |
| setTitle(/* String */ title) | 函数 | 设置窗格的标题 |
| toggle() | 函数 | 如果窗格处理打开状态，调用它可以关闭窗格；否则，打开窗格 |

虽然可以使用TitlePane来创建静态的界面组件，但用它来创建更具有交互性的控件同样很合适。例如，许多应用程序中都会有的粘性注释（sticky note）之类的控件，就可以通过TitlePane来模仿。例15-6展示在TitlePane中插入Textarea，并在窗格关闭时重设标题的过程。

例15-6：通过TitlePane模仿粘性注释

```
dojo.addOnLoad(function() {
    var ed = new dijit.form.Textarea({id : "titlePaneContent"});
    dijit.byId("tp").setContent(ed.domNode);
});

// 接下来的ContentPane可以在标记中创建：
<div id="tp" dojoType="dijit.TitlePane" style="width:300px">
   <script type="dojo/connect" event="toggle">
        if (!this.open) {
            var t = dijit.byId("titlePaneContent").getValue();
            if (t.length > 15)
                t = t.slice(0,12)+"...";
            this.setTitle(t);
        }
    </script>
</div>
```

然后，在此基础上再添加一些样式以及拖放功能，一个精巧的粘性注释应用程序就诞生了。

## InlineEditBox

InlineEditBox经常以一个包装部件的形式出现，因为它为真正可以编辑的控件提供了静态显示的容器，然后，当需要编辑部件内容时，只要单击它就可以立即就地编辑。例如，可以把那些具有固定尺寸且始终显示在屏幕中的可编辑TextBox包装到一个InlineEditBox中。此时，InlineEditBox就像一个普通的标签（label）一样。但是，只要单击一下InlineEditBox，它马上就会变成一个待编辑的TextBox。当表示编辑完成的某个事件发生后（例如，按下回车键），TextBox会立即切换回普通的文本形式。

在最简单的情况下，可以在InlineEditBox中包装一个TextBox，从而构建一个表单书信应用程序，如下面的代码片段所示。在这个示例中，原先可能会显示出来并搞乱布局的TextBox只作为纯文本存在，而当单击InlineEditBox时，它就变成了可编辑的控件：

```
Dear <span dojoType="dijit.InlineEditBox" autoSave="false"
  editor="dijit.form.TextBox">Valued Customer</span>:
```

```
<div>We have received your request to be removed from our spam list. Not to worry,
we'll remove you when we're good and ready. In the meanwhile, please do not hesitate
to contact us with further complaints.</div>

<div>Sincerely,</div>
<span dojoType="dijit.InlineEditBox" autosave="false"
  editor="dijit.form.TextBox">Customer Service</span>
```

在此，将autosave属性设置为false会导致控件中显示“保存”和“取消”按钮（在正常情况下，输入的文本会被保存，并不显示按钮）。这就是InlineEditBox的基本应用。接下来，我们在这个示例的基础上扩展一下，再创建另外一种编辑器。

下面简单的示例展示了如何在InlineEditBox中包装Textarea。其中，将renderAsHtml属性设置为true表明可以在文本区输入标记，而且输入的标记可以当场自动呈现：

```
Dear <span dojoType="dijit.InlineEditBox" autoSave="false"
  editor="dijit.form.TextBox">Valued Customer</span>:

<div dojoType="dijit.InlineEditBox" autoSave="false" editor="dijit.form.Textarea"
  renderAsHtml="true"> Insert<br>
    Form<br>
    Letter<br>
    Here<br>
</div>

<div>Sincerely,</div>

<span dojoType="dijit.InlineEditBox"
  autoSave="false" editor="dijit.form.TextBox">Customer Service</span>
```

与本章前面展示的部件相似，它们的基本应用都很简单。不过，表15-9也列出了需要读者了解并学会使用的其他配置项。

表15-9：InlineEditBox的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| editing | 布尔值 | 返回InlineEditBox 的编辑状态。当处于编辑模式时，值为true |
| autoSave | 布尔值 | 是否无须任何提示而自动保存修改后的值。默认值为true |
| buttonSave | 字符串 | 在“保存”按钮中显示的文本字符串。默认值为空字符串 |
| buttonCancel | 字符串 | 在“取消”按钮中显示的文本字符串。默认值为空字符串 |
| renderAsHtml | 布尔值 | 如果值为true，那么将InlineEditBox的内容按照HTML来呈现。默认值为false |

表 15-9：InlineEditBox 的 API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| editor | 字符串 | 用于指定作为编辑器的部件类的名称。默认值为 dijit.form.TextBox |
| editorParams | 对象 | 当在 InlineEditBox 中构建编辑器时，应该传入的参数 |
| width | 字符串 | 设置编辑器的宽度。默认值为 100% |
| value | 字符串 | 设置部件处于只读模式时显示的内容 |
| noValueIndicator | 字符串 | 占位字符串，在没有文本值的情况下显示（以便让用户能够通过单击该字符串触发编辑模式）。默认值为一个 wingding 字符 |
| setDisabled(/*Boolean*/disabled) | 函数 | 用于禁用或启用部件 |
| setValue(/*String*/val) | 函数 | 用于设置部件的值 |
| save | 函数 | 保存编辑器的内容并恢复到显示模式 |
| cancel | 函数 | 放弃在编辑器中所做的任何更改并恢复到显示模式 |
| onChange | 函数 | 扩展点，用于通知编辑器的值已经改变 |
| enableSave | 函数 | 用户可以重写的函数，用于启用或禁用“保存”按钮（例如，可以在内容未被更改时禁用该按钮） |

# Tree

Tree 是一种设计极为精巧的部件。在完全使用本地 DHTML 的基础上，无论从外观还是从行为上看，Tree 都如我们所期望的那样，就是一个层次分明的树状结构。Tree 不仅支持拖放操作，而且还能够绑定任何数据源。就和研究任何复杂的机械部件一样，在开始体验 Tree 之前，也必须了解一些基础知识。不过，这些基础知识也都非常容易理解。本节的篇幅之所以比较长，主要也是因为要介绍 Tree 提供的强大功能和大量特性。另外，虽然我们不会详细探讨 a11y，但读者应该知道 Tree 的确是非常易访问的，例如，通过键盘上的箭头键或回车键同样可以操作它。

**注意：** 深入理解 dojo.data API 对使用 Tree 部件非常有好处。有关 dojo.data API 的详细内容，请参考第 9 章。

在研究代码示例之前，首先了解下面的基础知识是非常有必要的。

*树和森林*

所谓树，就是一种只包含一个根节点的层次化的数据结构。森林，也是一种与树类似的层次结构，但它不只包含一个根节点，而是包含多个根节点。稍后我们就会看到，对树和森林的区分是一个常见的问题，因为虽然许多数据视图通过只有一个根节点的树来表现很方便，但支持该视图的后台数据却是一个带有隐含根节点的森林。

*节点*

树是由相互连接的节点按照层次关系组织起来的结构。被dijit.Tree使用的这种特殊节点类型是dijit._TreeNode —— 部件名称前面的下划线意味着不应该在Tree部件之外使用_TreeNode。不过，正如在稍后的示例中可以看到的，直接操作_TreeNode的某些属性倒是很有用。

*数据无关*

Tree部件与支持它的数据源无关。在Dojo1.1之前，Tree部件需要直接从dojo.data API的实现中读取数据，这种方式不仅灵活，而且也提供了统一的数据访问层。但是，从Dojo1.1开始，dojo.data模型与Tree之间又添加了增强的中间层。这些中间层包括dijit.tree.TreeStoreModel和dijit.tree.ForestStoreModel。做出这种修改的动机，主要是为了保证Tree能够更健壮以及更好地支持拖放操作。

---

**注意：** 在执行dojo.require("dijit.Tree")时，ForestStoreModel和TreeStoreModel会随同Tree部件一起被加载。

---

# 简单的树

为了循序渐进地介绍Tree的应用，我们先从最简单的数据源开始。假设有以下适合dojo.data.ItemFileReadStore读取的JSON数据文件：

```
{
    identifier : 'name',

    label : 'name',

    items : [
        {
            name : 'Programming Languages',
            children: [
```

```
                {name : 'JavaScript'},
                {name : 'Python'},
                {name : 'C++'},
                {name : 'Erlang'},
                {name : 'Prolog'}
            ]
        }
    ]
}
```

到目前为止，一切都很顺利。不过，我们不必手工在客户端解析数据，而是可以通过dojo.data来实现数据抽象。要在这个数据文件与ItemFileReadStore之间建立联系，只需将其URL指向相应数据源即可。如果以上数据以*programmingLanguages.json*文件的形式与页面保存在相同的目录中，那么当解析器实例化下面的标签后，就可以查询其中的数据了。而且，可以通过全局的dataStore标识符来访问这个数据源：

```
<div dojoType="dojo.data.ItemFileReadStore"
  jsId="dataStore" url="./programmingLanguages.json"></div>
```

但是，数据在被反馈给Tree之前，首先会被转交给TreeStoreModel。（稍后将展示如何使用ForestStoreModel。）有关中间层TreeStoreModel的完整API后面再具体介绍，但我们目前所关心的，只是必须在TreeStoreModel中指明ItemFileReadStore并提供一个查询。下面的TreeStoreModel将通过全局标识符dataStore查询dojo.data数据源中的所有name值：

```
<div dojoType="dijit.tree.TreeStoreModel" jsId="model" store="dataStore"
  query="{name:'*'}"></div>
```

最后，我们要做的就是像下面这样在Tree部件中指明TreeStoreModel：

```
<div dojoType="dijit.Tree" model="model"></div>
```

这样就可以了。例15-7将上述代码放到一起构成了一个完整的示例，结果如图15-3所示。

例15-7：带有一个根节点的Tree

```
<html>
    <head>
        <title>Tree Fun!</title>

        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
        <link rel="stylesheet" type="text/css"
          href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

        <script
            type="text/javascript"
```

```
            src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
            djConfig="parseOnLoad:true,isDebug:true">
        </script>

        <script type="text/javascript">
            dojo.require("dijit.Tree");
            dojo.require("dojo.data.ItemFileReadStore");
            dojo.require("dojo.parser");
        </script>
    </head>
    <body class="tundra">
        <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
          url="./programmingLanguages.json"></div>
        <div dojoType="dijit.tree.TreeStoreModel" jsId="model" store="dataStore"
            query="{name:'*'}"></div>
        <div dojoType="dijit.Tree" model="model"></div>
    </body>
</html>
```

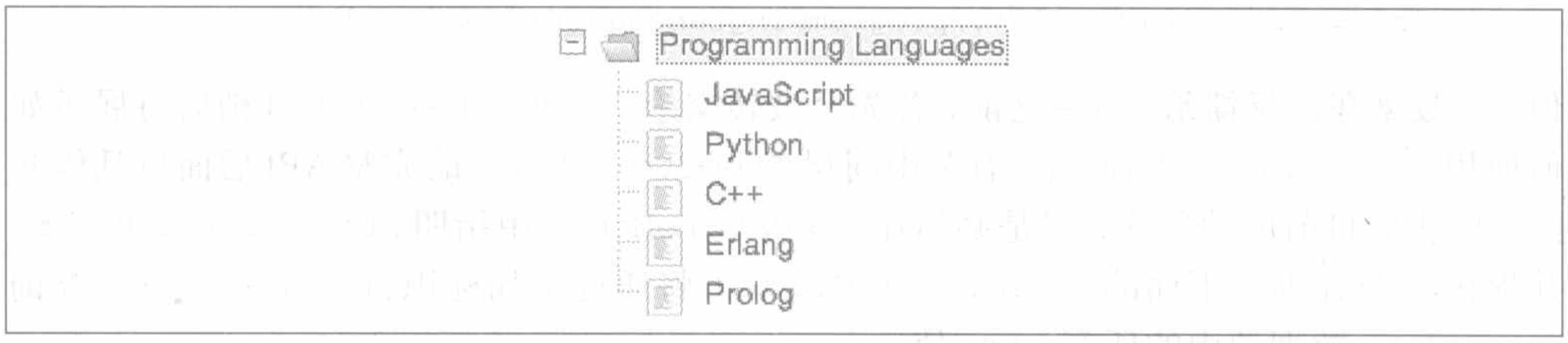



图 15-3：根据 dojo.data 数据源生成的 Tree；单击可扩展的节点能够关闭树

## 简单的森林

许多应用程序中的数据都不会简单地表现为只包含一个根节点。我们可以在前面示例的基础上稍加改造，把树变成森林，以便了解它们的区别。首先，森林必须对应着不止包含一个根节点的数据源。下面的数据源示例没有包含“Programming Languages”节点，而是按照类别对编程语言进行了划分，因此适合作为森林的数据源：

```
{
    identifier : 'name',

    label : 'name',

    items : [
        {
            name : 'Object-Oriented',
            type : 'category',
            children: [
                {name : 'JavaScript', type : 'language'},
                {name : 'Java', type : 'language'},
```

```
                {name : 'Ruby', type : 'language'}
            ]
        },
        {
            name : 'Imperative',
            type : 'category',
            children: [
                {name : 'C', type : 'language'},
                {name : 'FORTRAN', type : 'language'},
                {name : 'BASIC', type : 'language'}
            ]
        },
        {
            name : 'Functional',
            type : 'category',
            children: [
                {name : 'Lisp', type : 'language'},
                {name : 'Erlang', type : 'language'},
                {name : 'Scheme', type : 'language'}
            ]
        }
    ]
}
```

在修改后的JSON数据中，找不到唯一的根节点，因此这种数据只能通过森林视图来表现。与例15-7相比，代码中关键的修改包括：为Tree部件添加showRoot参数，以便隐藏它的根节点；修改query中的查询条件，以便正确地识别树的顶级节点；再有，就是把TreeStoreModel修改为ForestStoreModel。例15-8展示了更新后的代码，并加粗了修改的属性。

例15-8：修改后展示森林而不是树的代码

```
<body class="tundra">
    <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
        url="./programmingLanguages.json"></div>
    <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
        query="{type:'category'}"></div>
     <div dojoType="dijit.Tree" model="model" showRoot=false></div>
</body>
```

仅仅因为数据源适合以森林视图显示，并不意味着不能通过树来显示该数据源。如例15-9所示，通过为ForestStoreModel指定rootId和rootLabel属性凭空添加一个根级dojo.data项，可以虚构出树视图。

例15-9：通过虚构根节点使森林显示为树

```
<body class="tundra">
    <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
        url="./programmingLanguages.json"></div>
```

```
        <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
          query="{type:'category'}" rootId="root" rootLabel="Programming Languages"></div>
         <div dojoType="dijit.Tree" model="model" ></div>
</body>
```

实际上，现在就可以通过dojo.data API（例如，getLabel或getValue）对这个虚构的根节点进行统一地处理了。虽然看起来没有什么，但能够统一处理虚构的节点的确非常方便，因为这样就不必把它们当作特殊情况来对待了。图15-4展示了简单的森林。

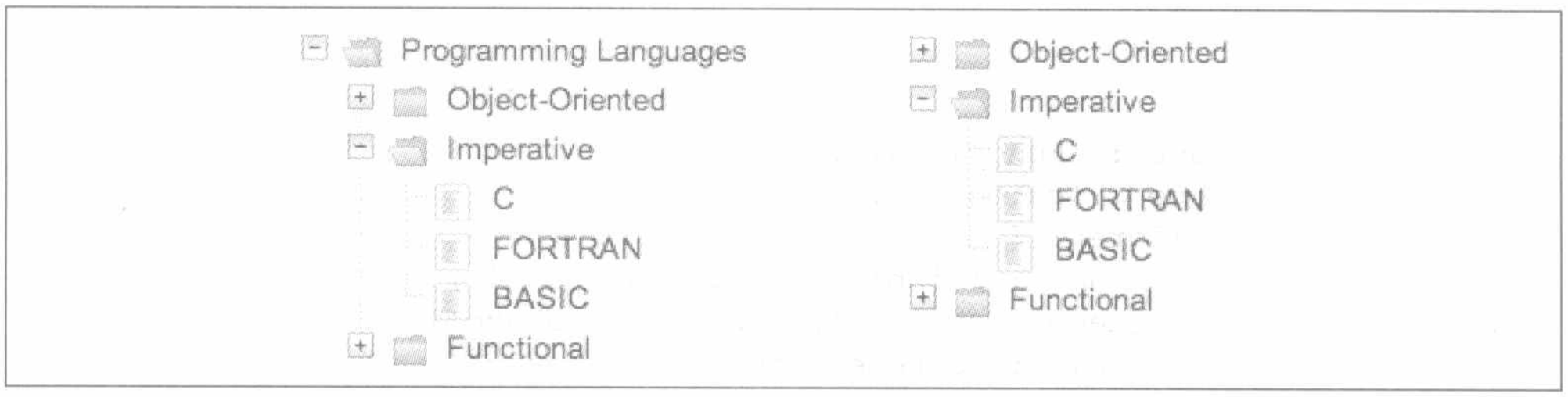



图15-4左：基于相同数据源生成的Tree（包含一个虚构的根节点）；右：显示为森林的Tree（没有根节点）

## 响应单击事件

通过树来显示信息固然好，如果能再让它响应事件（如鼠标单击）难道不是更好吗？下面，我们就通过实现onClick扩展点来展示让不同的项响应单击事件的可能性。当单击事件发生时，被单击的_TreeNode和dojo.data项会一起被传入onClick处理程序中以供处理。要处理单击事件，可以如例15-10所示添加相应的脚本。

例15-10：让Tree响应单击事件

```
<body class="tundra">
        <div dojoType="dojo.data.ItemFileReadStore" jsId="dataStore"
          url="./programmingLanguages.json"></div>
        <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
          query="{type:'category'}" rootId="root" rootLabel="Programming Languages"></div>
         <div dojoType="dijit.Tree" model="model" >
            <script type="dojo/method" event="onClick" args="item,treeNode">
              // 可以使用 item，也可以使用 treeNode……
              console.log("onClick:",dataStore.getLabel(item); // 显示标签（label）
            </script>
         </div>
</body>
```

虽然ForestStoreModel在Tree和dojo.data数据源之间提供了抽象的中间层，但

这里仍然通过数据源直接访问数据项；没有必要因为引入了方便数据显示的中间层，而在dojo.data数据项与访问它的常规手段之间增加多余的操作。

## 与Tree相关的API

如果读者逐一试验了前面的示例，并且对dojo.data API理解得比较透彻，那么对Tree的认识一定会更深一层。同样，表15-10中列出的API不仅可以作为参考，而且对于下一节介绍在Tree中实现拖放功能也很有帮助。显而易见的是，Tree本身只包含几个简单的属性，大多数复杂的处理过程都被转移到了dijit.tree.model API中或者在后台完成了。

---

注意：在Dojo1.1中，从技术上仍然可以把Tree直接与dojo.data数据源连接起来；但是，由于将来的2.0版本很可能会剥离这种模式，并且为用好Tree而进一步增强现有模式，因此本章不会介绍这一主题，下面的API列表中也不会包含相关特性。

---

表15-10：Tree的API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| model | dijit.tree.model | 统一访问数据的接口 |
| query | 对象 | 为树返回顶级项的数据源查询。如果查询只返回一项，那么应使用TreeStoreModel作为中间层；否则，应使用ForestStoreModel |
| showRoot | 布尔值 | 是否显示树的根节点；通常用于针对ForestStoreModel隐藏根节点 |
| childrenAttr | 数组 | 一个字符串数组，可以用来枚举包含树的子项的属性。默认值为["children"] |
| openOnClick | 布尔值 | 如果设置为true，单击节点的标签可以打开节点（与调用onClick相对，但onClick除了处理打开节点，还可用于处理其他操作）。默认值为false |
| persist | 布尔值 | 是否通过cookie保存节点的扩展和折叠状态。默认值为true |

表15-10：Tree的API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| onClick(/*dojo.data.Item*/ item,/*TreeNode*/node) | 函数 | 扩展点，用于处理对项的单击（及在该项上按下回车键）。可以处理同时传入的item和node参数 |

表15-11列出了dijit.Tree.model的API。这个接口中存在的任何特性对于前面示例中使用的TreeStoreModel都是有效的。与其他API的情况类似，只要遵循这个规范，就可以为生成Tree而创建任何数据抽象 —— 无论底层是什么数据源 —— 也许是dojo.data API、其他开放API，或者完全专有的API。

表15-11：dijit.Tree.TreeStoreModel的API

| 名称 | 说明 |
| --- | --- |
| getRoot(/*Function*/onItem, /*Function*/onError) | 用于遍历树。基于树的根项调用onItem函数，这里的根项可能是也可能不是虚构的。如果发生错误则运行onError函数 |
| mayHaveChildren(/*dojo.data.Item*/item) | 用于遍历树。返回某个项是否可能包含子项的信息，其用处在于当可扩展的项被单击之前，它不会检查该元素是否包含子项 |
| getChildren(/*dojo.data.Item*/parentItem, /*Function*/onComplete) | 用于遍历树。以parentItem的所有子项调用onComplete函数 |
| getIdentity(/*dojo.data.Item*/item) | 用于检查项。返回指定项的标识符 |
| getLabel(/*dojo.data.Item*/item) | 用于检查项。返回指定项的标签 |
| newItem(/*Object?*/args, /*dojo.data.Item?*/parent) | 属于Write接口。按照dojo.data.api.Write的方式创建一个新的dojo.data数据项 |
| pasteItem(/*dojo.data.Item*/childItem, /*dojo.data.Item*/oldParentItem, /*dojo.data.Item*/newParentItem, /*Boolean*/copy) | 属于Write接口。将某个项从其父项移动或复制到其他父项中，用于实现拖放操作。如果提供了oldParentItem并且copy值为false，则从oldParentItem中删除子项；如果提供了newParentItem，那么将childItem添加到其中 |

表 15-11：dijit.Tree.TreeStoreModel 的 API（续）

| 名称 | 说明 |
| --- | --- |
| onChange(/*dojo.data.Item*/item) | 用于更新标签或图标的回调函数。变成某个项的子项或父项触发 onChildrenChange，因此在 onChange 中应该忽略那些改变 |
| onChildrenChange(/*dojo.data.Item*/ parent,/*Array*/ newChildren) | 用于响应添加、修改或删除项的回调函数 |
| destroyRecursive() | 销毁对象并释放对数据源的连接，以便垃圾回收机制起作用 |

在 TreeStoreModel 的基础上，ForestStoreModel 又提供了两个函数（如表 15-12 所示），用于响应与虚构根节点（即添加或删除顶级项）有关的事件。当改变发生时，这两个函数对于调整查询条件以便保持树的顶级项有效是必需的。作为与数据无关的视图，Tree 本身不负责对其中项的更新及操作；确保查询条件能够得到满足的责任应该由开发人员来承担。而这两个函数存在的目的就是对此提供支持。

对于修改例 15-9 而言，调整某一项以符合顶级查询条件，只要将该项的 type 修改为“category”而不是“language”即可。例如，可以将“Java”移动到顶级，将其 type 修改为“category”，然后再通过操作将具体的 Java 实现（type 为“language”）添加为其子项。下一节我们将会看到，大多数需要遵循这些约定的情形基本上都与拖放有关。

表 15-12：dijit.tree.ForestStoreModel 增加的 API

| 名称 | 说明 |
| --- | --- |
| onAddToRoot(/*dojo.data.Item*/item) | 当一项被添加到树的顶级时调用；通过重写这个函数可以修改该项，以便它与查询顶级项的条件匹配 |
| onLeaveRoot(/*dojo.data.Item*/item) | 当一项被从树的顶级删除时调用。通过重写这个函数可以修改该项，以便它不再与查询顶级项的条件匹配 |

## 在 Tree 中实现拖放

实现上一节中详细介绍的 dijit.tree.model API，很大程度上是为了简化和协调在 Tree 中实现拖放操作。不过，拖放操作也并非是适合所有情况的万能方案。因此，如果读者想创建能够响应拖放事件的高级自定义部件，一定要做好迎接挑战的心理准备。当然，充分思考并回答下列常见问题是至关重要的。

- 拖动开始时会发生什么？
- 尝试放置时会发生什么？
- 取消放置时会发生什么？

当前，在树中实现拖放的架构要求按照dojo.dnd模块（参见第7章）的定义实现很多API，并通过Tree的dndController属性传入该实现。由于从头开始完成这一任务并非易事，因此Dojo1.1中提供了实现上述API的dijit._tree模块，以便开发人员在需要时使用：可以使用它的子类，或者有选择地重写它的方法，也可以将其他操作混入其中，甚至仅仅将它作为参考，为自己从头开始实现提供灵感。无论怎样，只要最终得到实现dojo.dnd.Source的类，并且该类能够与支持Tree的dijit.tree.model实现适当交互，就是成功的。特别是，你对dojo.dnd.Source的实现应该着重考虑并至少包含Tree的dndController必需的下列的关键方法，如表15-13所示。

表15-13：Tree的dndController接口

| 名称 | 说明 |
| --- | --- |
| onDndDrop(/*Object*/source, /*Array*/nodes, /*Boolean*/copy) | 针对/dnd/drop主题事件的处理程序（当放置操作完成时调用），用于根据操作的源和目标来更新数据源的项，以便这三者都能得到适合的更新 |
| onDndCancel() | 针对/dnd/cancel主题事件的处理程序，用于处理取消放置的操作 |
| checkAcceptance(/*Object*/source,/*Array*/nodes) | 用于检查目标是否能够接受来自源（source）的节点。经常用于根据节点的某些属性拒绝放置操作 |
| checkItemAcceptance(/*DOMNode*/target,/*Object*/source) | 用于检查目标是否能够接受来自源（source）的节点。经常用于根据目标的某些属性拒绝放置操作 |
| itemCreator(/*Array*/nodes) | 当把某个项从源拖放到由不同数据源支持的目标中时，为了让该数据源接受放置操作，必须针对节点中的每个元素创建一个新项。这个方法为源和目标由不同数据源支持的情况提供了创建节点的手段 |

**警告：** 在使用dndController方法时，需要注意一个细节：如果想在标记中引用它们，必须在解析器解析Tree所在页面时将它们定义为全局变量；这样，就不能在dojo.addOnLoad块中声明这些方法了，因为dojo.addOnLoad块中的代码要等到解析器执行完毕后才会运行。然而，我们完全可以不在标记中引用这些方法，而是等到dojo.addOnLoad块执行时再使用它们——这正是后面示例中所采用的方式。

这里有一个极为重要的现实问题，即拖放涉及的是DOM节点，而非_TreeNode；但是，我们通常在Tree中使用的都是_TreeNode，因为它作为底层数据来源为我们提供了所需的信息，而DOM节点则不能提供那些信息。如果需要使用_TreeNode（表15-13中的方法就需要），可以使用dijit.getEnclosingWidget方法，该方法可以将DOM节点转换为_TreeNode。

### 可拖放的Tree的示例

由于这些方法的确极为常用，因此可以在构建Tree时传入它们。而且，这样做对于最大限度地利用dijit._tree提供的功能很关键。说到这里，下面就应该看一个示例了。我们可以将例15-9改造成一个可拖放的树。为了把工作量减至最低，需要利用dijit._tree的功能。此外，鉴于拖放不是只读操作，因此还必须将数据源由ItemFileReadStore切换成ItemFileWriteStore。

---

**注意：** 尽管在拖放树中的项时，树会根据操作更新显示以反应该操作，而这似乎是Tree在更新自己，但是不要忘记——**Tree**只是一个视图而已。实际上，我们看到任何变化都是由更新数据源所导致的，即是数据源触发了视图的更新。

---

为了保证示例的完整性，需要防止用户在其他项上面放置项，因为项与dojo.data数据源中项的category（类别）有着本质的不同。例15-11展示代码，图15-5展示可以拖放的树。

例15-11：简单的可拖放的Tree

```
<html>
  <head>
    <title>Drag and Droppable Tree Fun!</title>

    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css" />
    <link rel="stylesheet" type="text/css"
      href="http://o.aolcdn.com/dojo/1.1/dijit/themes/tundra/tundra.css" />

    <script
        type="text/javascript"
        src="http://o.aolcdn.com/dojo/1.1/dojo/dojo.xd.js"
        djConfig="parseOnLoad:true,isDebug:true">
    </script>

    <script type="text/javascript">
        dojo.require("dijit.Tree");
        dojo.require("dojo.data.ItemFileWriteStore");
```

```
        dojo.require("dijit._tree.dndSource");
        dojo.require("dojo.parser");

        dojo.addOnLoad(function() {
            //添加checkItemAcceptance处理程序……
            dijit.byId("tree").checkItemAcceptance = function(target, source) {
                //将目标（DOM节点）转换为树节点
                //然后，通过树节点取得项
                var item = dijit.getEnclosingWidget(target).item;

                //不允许在（虚构的）顶级放置项
                //也不允许在非类别项上放置项
                return (item.id != "root" && item.type == "category");
            }
        });
    </script>
</head>
<body class="tundra">
    <div dojoType="dojo.data.ItemFileWriteStore" jsId="dataStore"
        url="./programmingLanguages.json"></div>
    <div dojoType="dijit.tree.ForestStoreModel" jsId="model" store="dataStore"
        query="{type:'category'}" rootId="root" rootLabel="Programming Languages"></div>
     <div id="tree" dojoType="dijit.Tree" model="model"
        dndController="dijit._tree.dndSource"></div>
</body>
</html>
```

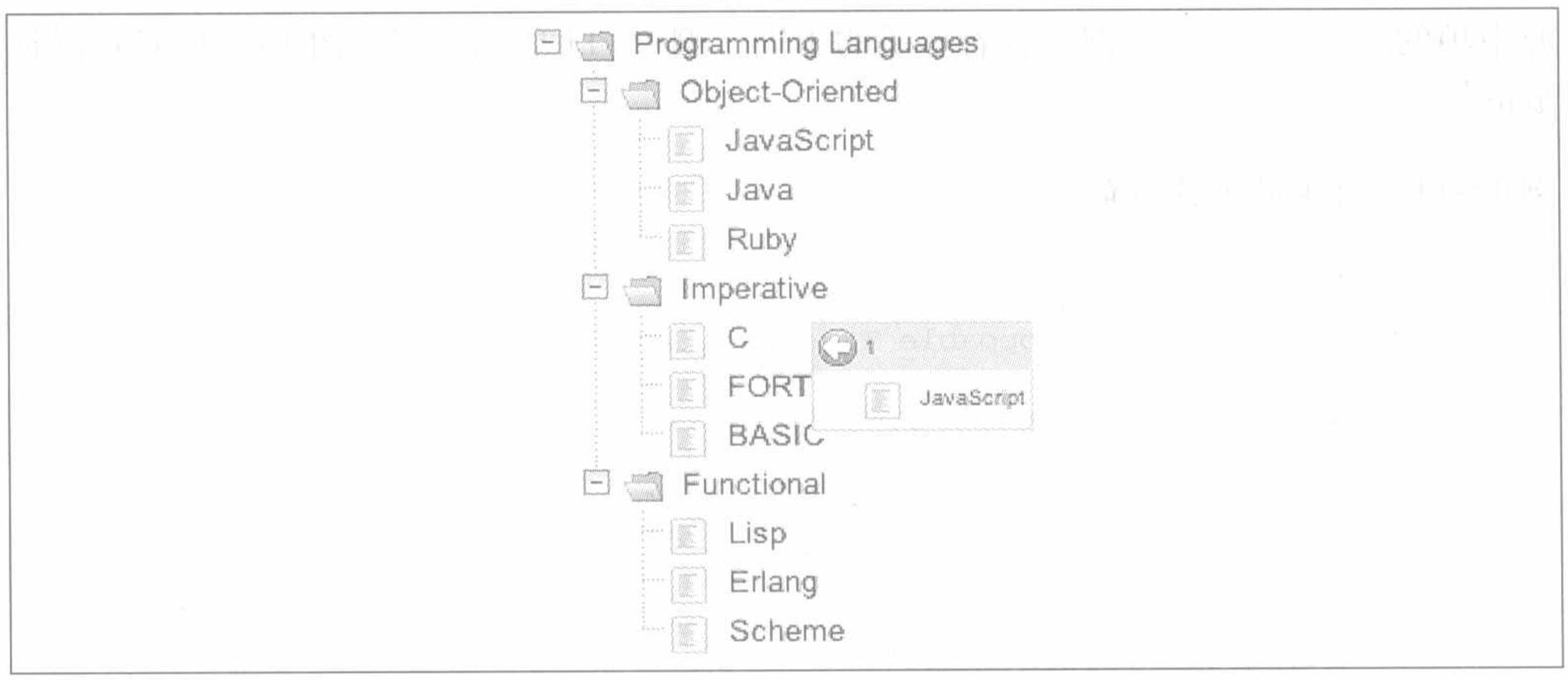



图15-5：将一个编程语言项移至不同的类别中

如果读者需要采用可拖放Tree的实现，建议首先认真研究一下dijit._tree中的代码。任何一种需要拖放功能的情况都有其特殊性，而企图找到无须自定义实现的现成方案事实上并不可行。

# Editor

**警告：** 在本书编写时，Editor及其插件架构正在进行较大幅度地改进。因此，读者可能会发现本节的内容与本书其他章节相比不太侧重于技术细节。

越来越多的应用程序开始利用富文本编辑功能；实际上，如果在华丽的RIA界面中放入一个普通的textarea元素（哪怕是一个Textarea部件），那么用户肯定会觉得它十分不协调。好在，Editor部件几乎提供了所有富文本编辑功能，可以让我们以最小的努力解决界面不协调的问题。

**注意：** 读者会发现以下链接对理解本节后面的内容非常有帮助：*http://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla*。

Dojo是在浏览器提供的其内容可以编辑的控件基础上构建富文本编辑功能的。如果谈到富文本编辑的历史，可以追溯到IE4.0引入的设计模式（design mode）概念，而这个概念使得在浏览器中使用与富文本编辑器类似的特性成为可能。Mozilla1.3紧随其后，实现了与IE4.0本质上相同的API，并最终推出了Midas Specification（*http://www.mozilla.org/editor/midas-spec.html*）。其他浏览器基本上也遵循了相同的思路，但存在细微的差别。无论如何，只要先明确将文档设置为可编辑，然后就可以使用JavaScript的execCommand函数产生实际的标记了。根据Midas Specification，下面几行代码就可以实现富文本编辑功能：

```
// 将节点设置为可编辑……也许该节点是一个有明确宽度和高度的DIV元素
document.getElementById("foo").contentDocument.designMode="on";

/* 选择其中的一些文本…… */

// 将选中的文本设置为斜体。不需要额外的参数。
editableDocument.execCommand("Italic", false, null);
```

读者大概已经猜到了，通过使用execCommand函数、规范化浏览器间的差异、构建一个便捷的工具条、应用一些标准的样式，然后再将这些封装到一个易移植的部件中，就能够执行很多操作内容的命令。实际上，这正是Dijit的Editor所实现的。虽然Editor提供的大量特性一开始会令人眼花缭乱，但它们的基本用法都非常简单。例15-12展示了一个现成的Editor的示例，其中包括标记、样式及与编辑器交互的按钮。

**注意：** 在不添加样式的情况下，Editor没有边框，其宽度将与容器相同，默认高度为300像素。示例中的样式只是设置了背景颜色，把Editor的高度调整为略小于其容器，以便内容不会跑到可见的背景之外并进入按钮区。

例15-12：典型的Editor应用

```
<div style="margin:5px;background:#eee; height: 400px; width:525px">
    <div id="editor" height="375px" dojoType="dijit.Editor">
        When shall we three meet again?<br>
        In thunder, lightning, or in rain?
    </div>
</div>
<button dojoType="dijit.form.Button">Save
    <script type="dojo/method" event="onClick" args="evt">
        /* 随便以什么方式来保存编辑器的值 */
        console.log(dijit.byId("editor").getValue());
    </script>
</button>
<button dojoType="dijit.form.Button">Clear
    <script type="dojo/method" event="onClick" args="evt">
        dijit.byId("editor").replaceValue("");
    </script>
</button>
```

我们建议读者抽时间亲自体验一下Editor提供的各种特性，从而实现富文本编辑功能并非痴人说梦。注意，Editor呈现的是纯HTML标记，因此在保存和读取内容时不应该进行任何不必要的转换。最后，如果想了解Editor提供了哪些典型的API，能够通过这些API实现什么功能，请参考表15-14。

**注意：** 到目前为止，Editor API是Dijit中最复杂的部分；而且，在本书编写时，致力于消除这些复杂性的建议已经得到正式采纳。因此，下表中只包含了Editor API中最有用的一小部分。如果确实想探知所有Editor API，可以查看其源文件中的文档。

表15-14：Editor最常用的一小部分API

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| focusOnLoad | 布尔值 | 是否在页面加载后把焦点转移到编辑器中 |
| height | 字符串 | 编辑器的初始高度。默认值为300像素 |
| inheritWidth | 布尔值 | 如果值为true，那么继承父节点的宽度；否则，横跨全部宽度。默认值为false |
| minHeight | 字符串 | 编辑器的最小高度。默认值为1em |

表 15-14：Editor 最常用的一小部分 API（续）

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| name | 字符串 | 如果提供这个属性，那么当用户离开页面并返回时，保存并恢复编辑器的内容 |
| plugins | 数组 | 用于为编辑器指定应该加载的基准插件。在默认情况下，常见的值包括 bold、italic、underline 等 |
| extraPlugins | 数组 | 用于为编辑器指定在 plugins 定义的基准之上应该额外加载的插件 |
| getValue () | 函数 | 返回编辑器的值 |
| setValue(/*String*/val) | 函数 | 设置编辑器的值 |
| undo() | | 函数撤销上一次操作 |
| onDisplayChanged(/*Event*/evt) | 函数 | 连接到这个事件可以在显示的内容改变时执行自定义操作 |
| close() | 函数 | 关闭编辑器并将内容串行化后写回到来源节点 |
| contentPreFilters | 数组 | 在将文本内容反串行化之后、转换为DOM树之前，对文本内容应用的过滤函数 |
| contentDomPreFilters | 数组 | 在将文本内容反串行化之后、加载到编辑器之前，对 DOM 树应用的过滤函数 |
| contentDomPostFilters | 数组 | 在将编辑器内容串行化之前，对 DOM 树应用的过滤函数 |
| contentDomFilters | 数组 | 在将编辑器内容串行化之后，对文本内容应用的过滤函数 |
| execCommand | 函数 | 对富文本区执行命令。其行为与 JavaScript标准的 exeCommand类似，但在不同浏览器实现中存在差别 |

## Editor 的架构

Editor的生命周期支持3个基本阶段，如图15-6所示。以下总结了这几个阶段的工作原理：

*反串行化内容*

加载阶段需要加载 DOM 节点提供的文本流，将该文本流转换为 DOM 树并显示出来，以便用户操作。为了在此阶段过滤或转换内容，可以按照先后顺序为文本流和

DOM树应用过滤函数。过滤内容的常见例子是把纯文本文档中的换行符转换为<br>标签，以便在编辑器中正确地显示内容。

*与内容交互*

交互阶段与在其他富文本编辑器中操作内容的过程相似。常见的操作包括为文本添加标记和基于可撤销的操作栈重复或撤销操作，操作栈可能是根据时间间隔，也可能是根据每次显示内容的变化建立的。

*串行化内容*

当调用Editor的close方法结束编辑时，将内容由DOM树串行化为文本流，然后再将文本流写回到来源节点中。此时，可以通过事件处理程序将文本流发送到服务器保存起来。与反串行化内容阶段类似，也可以为串行化前后的内容应用JavaScript函数。

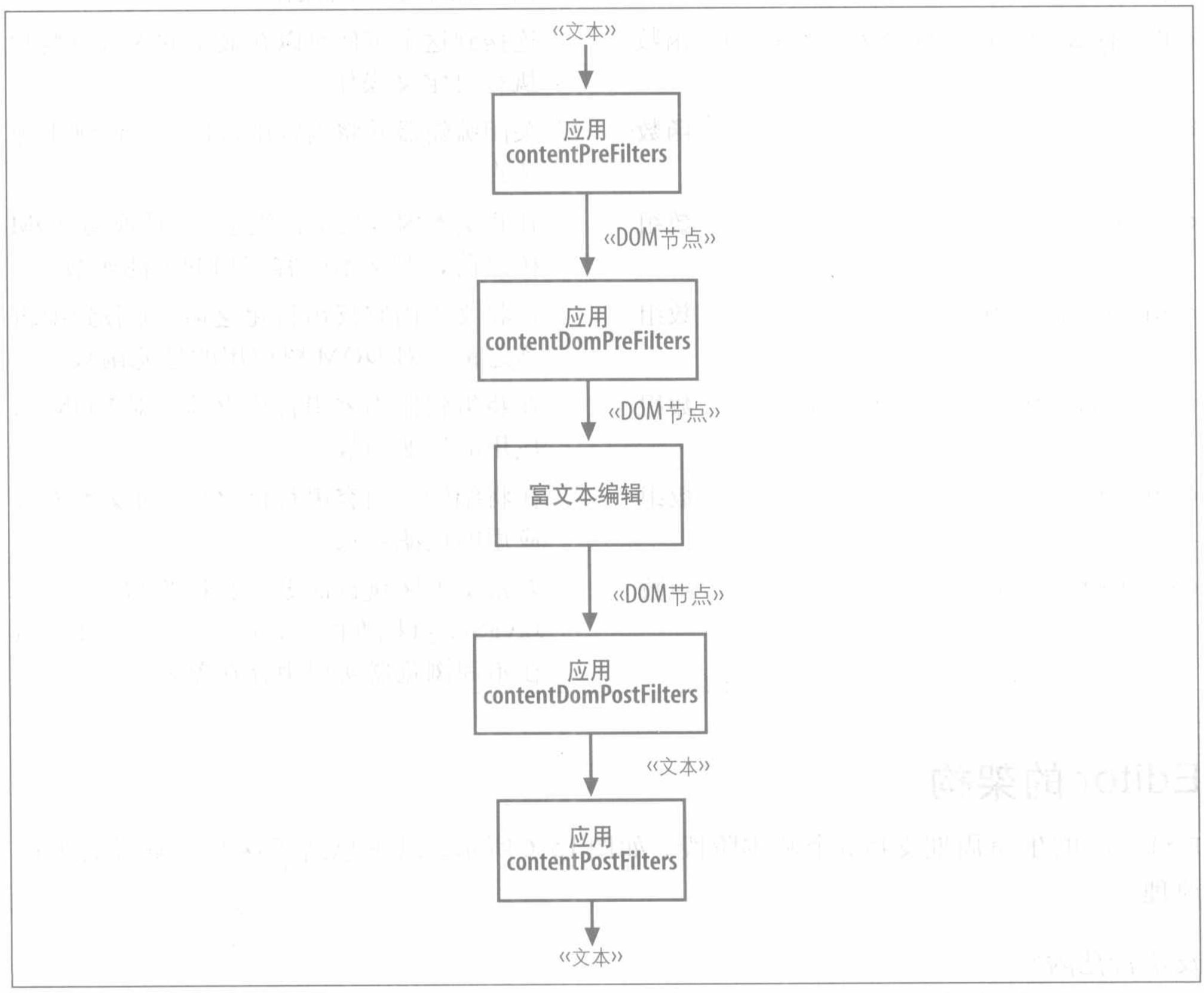



图15-6：Editor架构支持的基本阶段

## Editor的插件

尽管Editor本身提供了一大批非常实用的特性，但整合自定义功能的需求迟早还是会产生。Editor的插件架构就是使这一切成为可能的基础。所谓插件，就是指封装了某些有用功能、但又不是内置组件的程序；插件既可以是某些特别功能的组合，也可以在某些完成自动化任务的固有命令基础上提供自定义菜单项。

向Editor中添加插件非常简单。事实上，读者或许还没有发现，Editor工具条上那些内置的组件，从技术讲都是包含下列自描述（self-descriptive）值之一的插件。

```
undo                        justifyLeft
redo                        justifyRight
cut                         delete
copy                        selectAll
paste                       removeFormat
insertOrderedList           bold
insertUnorderedList         italic
indent                      underline
outdent                     strikethrough
justifyCenter               subscript
justifyFull                 superscript
```

在配置插件时，需要指定plugins或extraPlugins属性，并提供已经通过dojo.require语句加载到页面中的可用插件的列表。在默认情况下，plugins中包含着编辑器工具条中所有的默认选项。如果把plugins重写为plugins="['bold','italic']"，那么工具条中将只显示列表中指定的插件。不过，通过extraPlugins属性可以在plugins基础上再指定一些额外的插件。

Dojo工具箱提供了几个预制的插件包，这些插件包经常作为extraPlugins的值使用。这几个插件包位于*dijit/_editor/plugins*目录中，它们分别是：

AlwaysShowToolbar

在必要的时候移动工具条的内容，以便以多行显示控件，从而确保工具条始终可见。（如果把窗口宽度调整到小于工具条的宽度，那么默认操作是在窗口中显示水平滚动条，只显示在正常情况下可见的部分工具条控件。）要启用这个插件，必须为plugins或extraPlugins传入dijit._editor.plugins.AlwaysShowToolbar。

EnterKeyHandling

为按下回车键时所要执行的操作提供了一种跨浏览器的处理方式。例如，可以指定是否插入一系列段落标签来包含新输入的文本，还是插入换行标签、DIV标签，甚至完全停止对按下回车键的响应。要启用这个插件，必须为plugins或extraPlugins传入dijit._editor.plugins.EnterKeyHandling。

**警告：** Editor的插件架构还有待改进，与此有关的讨论仍在进行中。而且，这项工作已经有了新的进展，读者可以通过*http://trac.dojotoolkit.org/ticket/5707*来及时了解有关信息。换句话说，如果想创建自定义插件，那么在插件架构变得更完善之前，很可能必须手工修改*Editor.js*中的源代码。

同样，也不能忘记先通过dojo.require语句把将要使用的插件加载到页面中。插件架构至今还不会对此执行任何自动检测。

当前，处理回车键的默认方式由EnterKeyHandling的属性blockNodeForEnter决定，该属性的默认值是'P'。而且，目前还没有比扩展这个插件的原型并重写该属性更好的自定义方式：

```
dojo.addOnLoad(function() {
    dojo.extend(dijit._editor.plugins.EnterKeyHandling, {
        blockNodeForEnter : "div" // 或者"br"，或者"empty"
    });
});
```

FontChoice

提供一个带下拉菜单的按钮，以便选择字体名称、字体大小和格式块。plugins或extraPlugins的参数是fontName、fontSize或formatBlock。

LinkDialog

提供一个带按钮的对话框，用于输入超链接的源和显示的值。plugins或extraPlugins的参数是createLink。

TextColor

为指定一段选中文本的前景色或背景色提供选项。plugins或extraPlugins的参数是foreColor或hiliteColor。

ToggleDir

为编辑器提供一个与HTML的dir属性类似的功能（无论页面其余部分如何布局）。因此，编辑中的内容可能是从左到右或者从右到左。plugins或extraPlugins的参数是toggleDir。

为了简化问题，读者可在创建编辑器时参考表15-15中列出的这几段标记。

表 15-15：创建编辑器的不同方式

| 代码 | 效果 |
| --- | --- |
| `<div dojoType="dijit.Editor">` | 创建带默认工具条的编辑器 |
| `<div dojoType="dijit.Editor" plugins="['bold', 'italic']">` | 创建工具条中只包含加粗和斜体按钮的编辑器 |
| `<div dojoType="dijit.Editor" extraPlugins="['hiliteColor']">` | 创建的编辑器带有默认工具条，外加一个用于突出显示文本的按钮——假设已经执行了`dojo.require("dijit._editor.plugins.TextColor")`语句 |
| `<div dojoType="dijit.Editor" plugins="['bold', 'italic']" extraPlugins="['fontName']">` | 创建的编辑器带有一个只包含加粗和斜体按钮的工具条，外加一个可以选择字体的控件（假设已经执行了`dojo.require("dijit._editor.plugins.FontChoice")`语句）。注意，这和在`plugins`属性中指定这 3 个插件的效果相同 |

# 小结

在学习完本章之后，读者应该能够：

- 理解应用程序部件在整个 Dijit 架构中所处的地位，并明确它们在设计富用户体验时扮演的特殊角色。
- 在标记中和以编程方式创建应用程序部件。
- 理解何时应该使用 `Tooltip`，何时应该使用 `TooltipDialog`。
- 使用 `Editor` 为输入和编辑富文本提供控件。
- 使用 `Toolbar` 和 `Menu` 在应用程序中为用户提供一组命令和控件。
- 在 `DropDownButton` 中嵌入 `TooltipDialog`。
- 使用 `ProgressBar` 向用户显示确定型和非确定型的进度信息。
- 使用`Dialog`向用户提供一个模态对话框，或者在`Dialog`中嵌入供用户操作的内容。
- 使用 `InlineEditBox` 让用户能够动态编辑以纯标记显示的内容。
- 使用 `Tree` 部件以交互方式显示层次化的数据。

下一章，我们将介绍构建工具、测试和程序发布。

第16章

# 构建工具、测试及程序发布

在使用Dojo完成了艰苦的开发工作之后，下一步就该准备发布程序了。Util提供了非常好的构建工具和测试框架，可以自动完成程序发布的各项工作。事实上，Dojo正式发布的每个版本都是通过Util提供的构建工具生成的。而DOH（Dojo Objective Harness，Dojo目标套件）作为一个单元测试框架，则能够帮我们在程序发布之前把好最后一道质量关。

## 构建工具

对任何面临发布的程序而言，尽可能减少JavaScript文件的体积和需要向服务器发送同步请求的数量都是绝对必要的。如果想让用户在页面加载时体验到爽快的感觉，那么通过`dojo.require`语句以多个同步请求方式分别加载个别资源，与只向服务器发送一两次回调相比，结果可是大不一样的。

Dojo提供的构建工具能够难以置信地简化人工处理起来极为麻烦的任务。简单地说，这些构建工具能够自动完成如下所示任务。

- 把多个模块合并为被称为一层（layer）的一个JavaScript文件中。
- 把模板字符串内置于JavaScript文件（包括层）中，从而抛开单独的模板。
- 应用ShrinkSafe（基于Rhino的一个JavaScript代码压缩程序），通过删除空格、换行符、注释以及缩简变量名来把所有层的体积减到最小。
- 把经过“构建”的所有文件复制到一个单独的目录中，以便通过复制部署到Web服务器。

如果读者至今还没有发现我们所说的构建工具在哪里，原因可能是它们并不包含在正式

版的*util*目录内。要获得构建工具，必须下载Dojo的源代码版（源代码版的文件名中应该包含*-src*后缀），或者从Subversion主干（trunk）中直接取得源代码。第1章曾介绍过如何从Subversion下载Dojo工具箱，整个过程无非就是在终端中输入Dojo版本库（repository）的地址——无论是在trunk还是在其他标签下，然后耐心等到所有文件都下载完成。

不管通过什么途径获得Dojo的源代码版，读者同样都会发现*util*目录中又多出了几个目录；其中一个目录就是*buildscripts*，这个目录就包含着我们要找的构建工具。

**注意：** *http://svnbook.red-bean.com/*中包含非官方的Subversion图书，有多种格式和语言版本。现在，把这个有用的链接加为书签收藏起来，希望它能在将来节省你的时间。

要运行构建工具，必须安装Java 1.4.2或更高版本（因为ShrinkSafe基于Rhino开发，而Rhino则是使用Java编写的），下载地址为*http://java.sun.com*。不过，并不是只有Java程序员才能使用ShrinkSafe；ShrinkSafe以一个*jar*文件（或可执行的Java文档）的形式存在，因此可以把它看成一个可执行文件。

### 关于Rhino

Rhino是一个用Java编写的JavaScript引擎，并以David Flanagan那本众所周知的《JavaScript: The Definitive Guide》（O'Reilly）封面上的犀牛（rhinoceros）命名。Rhino最初是由Netscape于20世纪90年代末发起的一个闭源项目，但该项目被转交给Mozilla基金会之后就变成开源项目。Rhino的用途是将JavaScript脚本转换为Java类，而开发Rihno的目的是可以将它嵌入到应用程序中。

SpiderMonkey则是另外一个JavaScript引擎，用C编写而成，同样是由Netscape开发而后来移交给了Mozilla维护。与Rihno类似，SpiderMonkey也是一种可嵌入的技术。SpiderMonkey在很多应用程序中得到了广泛应用，包括Firefox和Yahoo! Widgets（即以前的Konfabulator）。

## 执行构建

启动构建过程的主要入口点是执行*buildscripts/build.sh*（或者在Windows中是*build.bat*），而实际过程就是调用定制的Rhino *jar*文件，而该文件则基于预定义的配置文件（profile）完成所有工作（稍后马上介绍配置文件）。不过，作为普通的可执行文件，也可以通过*Make*和*ant*等构建工具在构建过程中简单地包含这个*jar*文件。这对于构建基于编译语言的服务器端组件来说尤为方便。

在执行相应的构建脚本或者说执行*jar*文件时，如果不提供命令行选项，那么会生成一个看起来有点复杂的选项列表。表16-1中的内容直接取自生成的这个选项列表。

表16-1：构建脚本的参数

| 选项 | 说明 |
| --- | --- |
| xdScopeArgs | 如果指定loader=xdomain构建选项，那么在*.xd.js*文件中定义模块的函数将使用这个选项的值作为参数。这样就可以实现在一个页面中包含同一模块的多个版本。参考有关djConfig.scopeMap的文档可以了解更多内容 |
| cssOptimize | 指定如何优化CSS文件。如果值为comments，那么会将代码中的注释和换行符删除。如果值为comments.keepLines，那么只删除代码中的注释，但保留换行符。但在任何情况下，@import语句都不会独占一行 |
| releaseName | 指定版本的名称。将以这个名称在*releaseDir*中创建一个新目录。默认值为*dojo* |
| localeList | 指定在压缩与i18n有关的包时要使用的地区列表。默认值为cs,de-de,en-gb,en-us,es-es,fr-fr,hu,it-it,ja-jp,ko-kr,pl,pt-br,ru,zh-tw,zh-cn |
| releaseDir | 指定构建操作最终建立的顶级版本目录。*releaseName*指定的目录将在此目录内创建。默认值为*../../release/* |
| copyTests | 指定是否复制测试文件。默认值为true |
| symbol | 将函数符号作为全局引用插入，以便在各种调试程序中调试匿名函数（特别是在IE中，IE不会根据定义函数的环境推断函数名称）。可用的值为long和short。如果值为short，那么会在每个以模块名为前缀的版本目录中生成一个*symboltables.txt*文件，该文件中保存着与短符号名对应的描述性的长符号名 |
| action | 指定要执行的构建操作。要指定多项操作，应该使用逗号分隔的列表，如action=clean,release。可能的构建操作有clean和release。默认值为help |
| internStrings | 指定是否内置部件的模板文件。默认值为true |
| scopeMap | 用于将默认的*dojo*、*dijit*和*dojox*这几个作用域名称修改为其他名称。在把Dojo整合到另一个JavaScript库，同时保持该库独立，即不引用外部*dojo/dijit/dojox*的情况下使用。值为一个不包含空格的字符串，与djConfig.scopeMap的值类似（注意下面字符串中的反斜杠是必需的，用于防止命令解释程序转义双引号）：<br>`scopeMap: [[\"dojo\",\"mydojo\"],[\"dijit\",\"mydijit\"],`<br>`[\"dojox\",\"mydojox\"]]` |

表 16-1：构建脚本的参数（续）

| 选项 | 说明 |
| --- | --- |
| optimize | 指定如何优化模块文件。如果值为 comments，那么表示删除代码中的注释。如果值为 shrinksafe，那么以 Dojo 压缩程序来压缩这些文件，删除换行符。如果值为 shrinksafe.keepLines，那么使用 Dojo 压缩程序来压缩这些文件，但保留换行符。如果值为 packer，那么使用 Dean Edwards 的 Packer 压缩程序（参见 *http://dean.edwards.name/packer/*） |
| loader | 指定要使用的 dojo 加载程序的类型。可用的值为 default（默认值）和 xdomain |
| log | 指定日志记录的级别（logging verbosity）。要了解可用的整数值，请参考 *util/buildscripts /jslib/logger.js*。默认值为 0 |
| profileFile | 指定配置文件的路径。如果配置文件在 profiles 目录外部，那么需要指定这个选项。但是，如果指定了 profileFile，就不能再指定 profile |
| xdDojoPath | 如果指定了 loader=xdomain 选项，那么在为 *dojo*、*dijit* 和 *dojox* 调用 dojo.registerModulePath()时，将使用 *xdDojoPath* 的值。*xdDojoPath* 的值应该是包含 *dojo*、*dijit* 和 *dojox* 的目录，而且该目录不能以斜杠结尾，例如，*http://www.example.com/path/to/dojo* |
| version | 指定构建所要标记的版本字符串。默认值为 0.0.0.dev |
| profile | 指定构建时使用的配置文件。这个选项的值必须是 *profiles/* 目录中配置文件名的第一部分。例如，要使用 *base.profile.js*，应该指定 profile=base（这也是默认值） |
| layerOptimize | 指定如何优化层文件。如果值为 comments，那么删除代码中的注释。如果值为 shrinksafe，那么使用 Dojo 压缩程序压缩这些文件，删除换行符。如果值为 shrinksafe.keepLines，那么使用 Dojo 压缩程序压缩这些文件，但保留换行符。如果值为 packer，则使用 Dean Edward 的 Packer 压缩程序。默认值为 shrinksafe |
| xdDojoScopeName | 如果指定了 loader=xdomain 选项，那么在 *.xd.js* 文件中调用 dojo._xdResourceLoaded()时，将使用 *xdDojoScopeName* 的值取代 dojo（默认值）。指定这个选项不仅可以在不同的作用域名称之下使用 dojo，而且还能使用这个作用域名称实现跨域加载 |
| cssImportIgnore | 可以使用 cssOptimize=comments 强制 @import 内嵌处理忽略一组文件。这个选项的值应该是以逗号分隔的要忽略的 CSS 文件名列表。其中，文件名应该与 @import 调用中使用的字符串值相符 |

表16-1：构建脚本的参数（续）

| 选项 | 说明 |
| --- | --- |
| buildLayers | 一个以逗号分隔的要构建的层名称列表。如果指定这个选项，那么只有列表中包括的层会被构建。对于层的快速开发及测试有帮助。如果指定这个选项不能满足需要，可以不指定它，并使用action=clean,release执行一次完整的构建。使用这个选项前，至少应该执行过一次完整的构建 |
| symbol | 将函数符号作为全局引用插入，以便在各种调试程序中调试匿名函数（特别是在IE中，IE不会根据定义函数的环境推断函数名称）。可用的值为long和short。如果值为short，那么会在每个以模块名为前缀的版本目录中生成一个*symboltables.txt*文件，该文件中保存着与短符号名对应的描述性的长符号名 |
| scopeDjConfig | 在建构的*dojo.js*文件中加入一个djConfig对象。如果要基于自己的作用域执行构建，并且想在构建的版本中使用本地djConfig对象，那么指定这个选项可以避免本地djConfig对象与页面中以全局对象形式存在的djConfig发生冲突。在构建生成的源代码中，这个选项的值必须是一个等价于JavaScript对象直接量的字符串。此外，如果是在其他JavaScript库中使用Dojo，并且该库是不依赖外部*dojo*、*dijit*及*dojox*的独立的库，那么也可以指定这个选项。例如，<br>scopeDjConfig={isDebug:true,scopeMap:[[\"dojo\",\"mydojo\"],<br>[\"dijit\",\"mydijit\"], [\"dojox\",\"mydojox\"]]}<br>注意，如果在命令行中输入这个选项，那么不要忘记其中的反斜杠；反斜杠用于防止命令解释程序转义双引号 |

虽然这些选项看起来比较多，但例行的构建过程其实很简单，往往只需要其中几个选项。不过，在执行构建之前，必须先准备好配置文件。

## 准备配置文件

所谓配置文件（profile），就是通过profile或profileFile选项为构建脚本提供配置信息的文件。配置文件最基本的功能，就是指定要将哪些Dojo资源合并到一个单独的JavaScript文件中，这个单独的JavaScript文件又叫做层（layer）。一般来讲，应用程序中的每个页面都应该有自己独立的层。层的优点在于，它本身就是一个JavaScript文件，能够直接被包含在页面的头部，因此只要向服务器发送一次同步请求，就可以把合并到这个文件中的资源一股脑地加载到页面中——嗯，大致如此吧。根据惯例，由于Base极为常用，最好要把它包含在单独的*dojo.js*文件中。换句话说，在正常情况下必须要发送两次同步调用，一次加载Base，另一次加载你自己的层。

### 建立配置文件

假设有一个包含3个页面的应用程序，那么构建后应该生成3个层文件和1个Base的副本。

---

**注意：** 如果想把自定义的模块都打包到通常只包含Base的*dojo.js*文件中，可以把层命名为*dojo.js*。然而，我们认为还是保持Base的独立更好一些，因为应用程序中的每个页面都要用到Base，而且它能够被Web浏览器缓存起来。

---

实际上，配置文件就是一个包含JSON对象的文件。例16-1展示了一个配置文件，该文件指明了要合并几个需要使用dojo.require语句加载的表单部件。配置文件中指定资源的依赖关系全部被自动跟踪。就和使用dojo.require一样，只要在配置文件中直接声明想使用什么即可，构建程序会在后台自动实现对依赖关系的跟踪。

例16-1：简单的构建配置文件

```
dependencies ={
    layers:  [
        {
            name: "form.js",
            dependencies: [
                "dijit.form.Button",
                "dijit.form.Form",
                "dijit.form.ValidationTextBox"
            ]
        }
    ],
    prefixes: [
        [ "dijit", "../dijit" ]
    ]
};
```

假设前面配置文件的路径是*util/buildscripts/profiles/form.profile.js*，而构建时使用的命令解释程序为Bash，那么在*util/buildscripts*目录下执行下面的命令将会启动一次构建过程。注意，profile选项要求以<*配置文件名*>*.profile.js*的形式保存配置文件，但只以<*配置文件名*>作为选项的值：

```
bash build.sh profile=form action=release
```

---

**注意：** 如果不想把配置文件保存为*util/buildscripts/profiles/form.profile.js*，而保存在其他目录中，可以使用profileFile选项代替profile选项。

---

在执行命令后，通过观察生成的输出文件可以判断构建已经发生，而且模板字符串已经取代模板文件被置入了JavaScript文件中。构建生成的输出是一个版本（release）目录，其中包含*dojo*、*dijit*和*util*。而在*dojo*目录中，除了常规的文件之外，有4个特别重要的文件是通过构建生成的，以下就是这4个文件：

- 经过压缩和未经压缩的Base，即*dojo.js*和*dojo.js.uncompressed.js*。
- 经过压缩和未经压缩的表单层，即*form.js*和*form.js.uncompressed.js*（打开这个文件可以看到未经压缩的源代码）。

但是，如果需要自定义层文件中没有包含的资源该怎么办呢？没问题！如果配置文件中没有包含相应资源，那么照旧可以在页面中通过`dojo.require`语句来加载它们。也就是说，当你把完整的版本目录放到服务器上之后，请求未包含在层中资源的`dojo.require`语句还将跟往常一样有效，只不过会因此导致浏览器再向服务器发送一次小小的请求罢了。

通过`dojo.require`请求Base中的特性和已经包含在层中的资源，不会导致再向服务器发送请求，因为它们已经被加载到本地了。然而，对于未包含在层中的资源，则会导致例行的同步HTTP请求（见图16-1）。

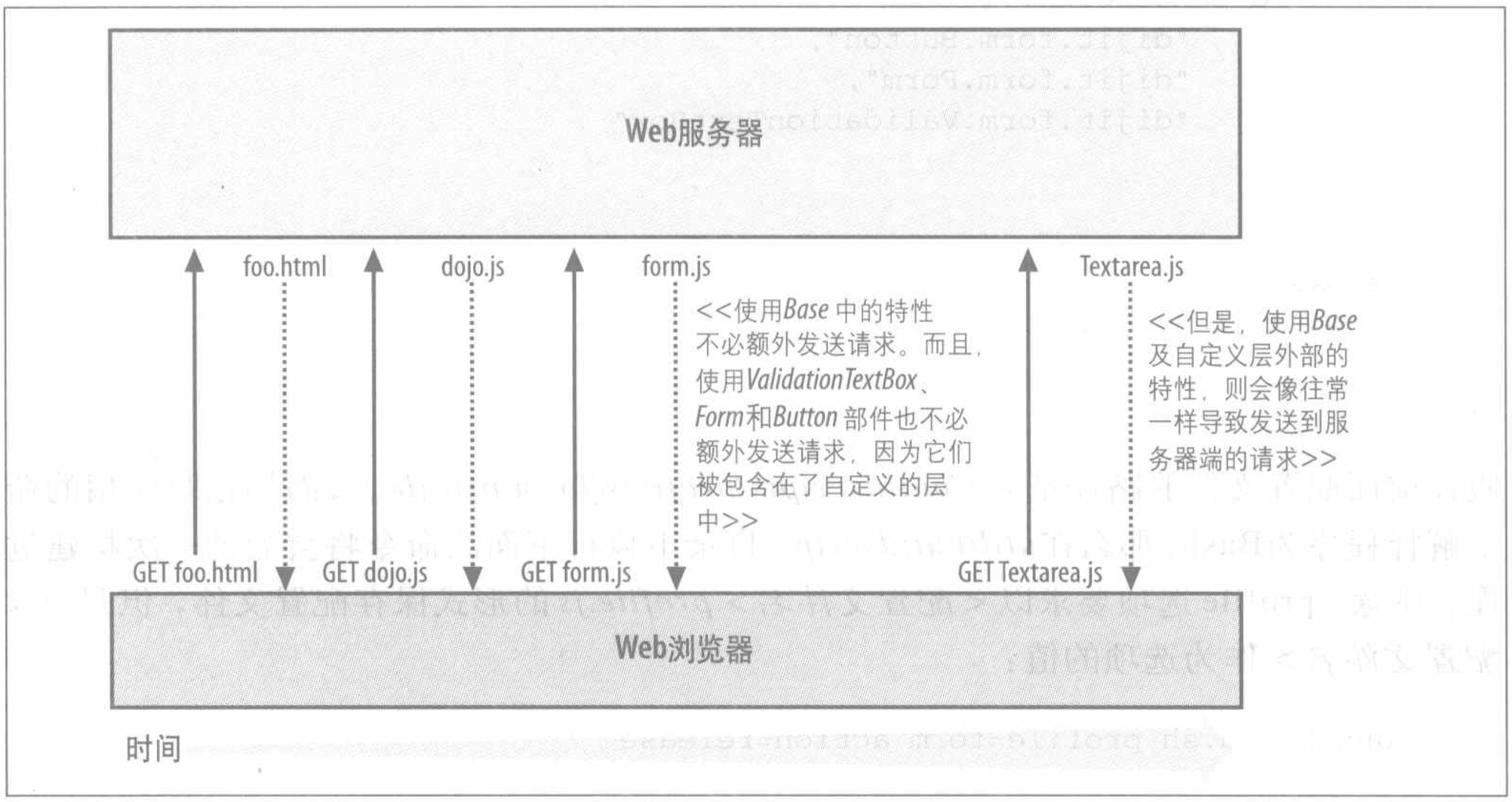



图16-1：请求多个JavaScript文件时要向服务器发送请求的示意图

虽然在执行构建时，我们通常都希望把要用到的资源尽可能包含进来，但在某些情况下延迟加载资源则是更好的选择。当然，这涉及一个永恒的话题，即要反复权衡初始化时

通过网络发送的资源“足够小”与将来由dojo.require导致的同步加载次数“尽量少”之间的关系。

---

**警告：** 如果在建立配置文件时意外地拼错了字母，或者给出了不存在的依赖关系，那么即使ShrinkSafe无法确认全部依赖关系，它也仍然会完成构建过程。例如，假设把dijit.form.Button意外地拼成了dijit.Button，那么构建仍然会成功。但是，由于通过dojo.require("dijit.form.Button")能够从服务器取得该按钮部件，而且应用程序依旧运行正常，因此你可能永远不会发现dijit.form.Button并没有被打包到层文件中。

为此，我们建议读者在执行完构建之后，最好通过Firebug的Net标签对构建文件进行复查，以确保应该打包的资源确已被打包。

---

### 建立配置文件的另一种（更聪明的）方式

比刚刚讨论的建立配置文件的方式更聪明一些的做法，是创建一个自定义模块，在这个模块中请求那些以前通过配置文件指定要放到层中的资源。然后，再在配置文件中包含这个自定义模块，把这个模块作为层的唯一依赖关系。

首先，例16-2展示了如何创建上述自定义模块。我们在此假设模块名为*dtdg.page1*，并且保存路径为*dtdg/page1.js*。

例16-2：用于建立配置文件的自定义模块

```
dojo.provide("dtdg.page1");

dojo.require("dijit.form.Form");

dojo.require("dijit.form.Button");
dojo.require("dijit.form.ValidationTextBox");
```

接着，只要在配置文件中指定这个自定义模块就可以了，因为其他依赖关系都已经通过这个模块确定，并且都能够被自动跟踪。例16-3展示了在此基础上修改之后的配置文件，在此，假设自定义模块的目录与*util*目录相邻。

例16-3：修改后的配置文件

```
dependencies ={
    layers:  [
        {
            name: "form.js",
            dependencies: [
                "custom.page1"
            ]
        }
    ],
```

```
    prefixes: [
        [ "custom", "../custom" ]
    ]
};
```

最后，要在页面中加载这个模块和Base，可以使用下面的SCRIPT标签：

```
<script type="text/javascript"
djConfig="baseUrl: './',modulePaths: {custom:'path/to/custom/page1.js'},
    require: ['custom.page1']"
src="scripts/dojo.js"></script>
```

### 标准的配置文件

在*util/buildscripts/profiles*目录中，包含着一些示例配置文件。其中，有一个*standard.profile.js*文件，它是为构建标准Dojo版本指定层的标准配置文件。标准配置文件用于构建Base、基准Dijit及其他有用的层。基准Dijit层中包含着适用于任何与部件有关情形的特性。需要注意的是，*standard.profile.js*文件中指明的任何层都可以通过AOL的CDN加载。例如，要取得基准Dijit层，可以执行下面的语句：

```
dojo.require("dijit.dijit");
```

不过，始终要记住的是，页面中的第一个SCRIPT标签必须用于加载Base（*dojo.xd.js*）；即，要在加载Base后的SCRIPT标签中加载其他层。

### ShrinkSafe优化及其他常用选项

在任何程序发布之前，还应该考虑通过ShrinkSafe来缩减文件大小。虽然在前面的示例中，构建过程也缩减了*dojo.js*、*form.js*，并且内置了模板字符串，但ShrinkSafe则能够缩减新版本中的每个文件。

从有效负载的角度来讲，只有“通过网络传输”的文件大小才是影响性能的关键所在。虽然文件在服务器上的大小是一定的，但如果Web浏览器能够处理经过*gzip*压缩的文件，那么大多数服务器都可以对文件进行*gzip*压缩。ShrinkSafe在缩减JavaScript文件时，采取的是删除空格、注释等非代码字符的方式，但进一步压缩文件仍然是有可能的。代码中总会存在一些重复出现的公共符号，如dojo、dijit，以及开发人员自定义的标识符，这些重复的符号就是压缩的对象。

---

**注意：** 缩减是通过删除逗号、空格、换行符等非代码字符来达到减小文件大小的目的。而压缩则是一种算法处理，它会查找代码中同一符号的多个实例，然后使用较短的占位符来代替这些重复的符号，最终达到减小文件大小的目的。要了解有关*gzip*压缩的资料，请参考*http://en.wikipedia.org/wiki/Gzip*。

---

ShrinkSafe的一个重要特性是它从不破坏公共的API；这一点与某些JavaScript工具试图以应用正则表达式或加入其他费解逻辑的方式加密JavaScript，从而“保护”脚本的做法明显不同。实际上，试图保护JavaScript代码的想法在多数情况下没有太大意义。作为运行于浏览器中的解释型语言，JavaScript应用程序的用户肯定能够看到它的源代码。而通过解密程序将受保护的脚本处理成能够理解的代码，也不是件十分困难的事情。

**注意：** ShrinkSafe不是只对Dojo代码才有效的工具；它的在线演示程序（*http://shrinksafe.dojotoolkit.org/*）可以让任何JavaScript文件享受到“免费减肥”的好处。OS X用户可以到*http://dojotoolkit.org/downloads*中下载相应的版本，其他平台的用户则可以从下面的链接获取一个独立的自定义Rhino jar文件：*http://svn.dojotoolkit.org/dojo/trunk/buildscripts/lib/custom_rhino.jar*。

换句话说，ShrinkSafe在缩小JavaScript文件时不会修改公共的变量名。事实上，通过查看前面构建示例中生成的*form.js*，读者就会看到ShrinkSafe只删除了注释、合并及/或去掉了没有意义的空白符（包括换行符），并以较短的标识符替换了非公共的符号。相反，以较短的、含义模糊的名称替换所有符号，属于试图加密代码的行为，对于解密程序而言往往不会奏效。

下面，我们就来看一看应该如何修改前面的配置文件。

- 指定`optimize="shrinksafe"`选项，以缩减所有文件的大小。
- 指定应该出现在每个缩减后的JavaScript文件顶部的自定义注意事项，由*CUSTOM_FILE_NOTICE.txt*提供的另一个（虚构的）`foo`模块提供。
- 指定应该出现在最终的*form.js*文件顶部的自定义注意事项，同样由*CUSTOM_LAYER_NOTICE.txt*提供。
- 指定`releaseName="form"`选项，为版本目录提供自定义的名称。
- 指定`version="0.1.0."`选项，为构建的程序提供自定义的版本号。

以下就是在例16-1的基础上修改得到的*form.profile.js*文件。值得注意的是，自定义注意事项的内容必须以JavaScript注释的形式定义；而自定义注释的路径应该相对于*util/buildscripts*，否则应该使用绝对路径：

```
dependencies ={
    layers:  [
        {
            copyrightFile : "CUSTOM_LAYER_NOTICE.txt",
            name: "form.js",
```

```
                dependencies: [
                    "dijit.form.Button",
                    "dijit.form.Form",
                    "dijit.form.ValidationTextBox"
                ]
            }
        ],
        prefixes: [
            [ "dijit", "../dijit" ],
            [ "foo", "../foo", "CUSTOM_FILE_NOTICE.txt" ]
        ]
};
```

重新启动构建的新命令也很容易看懂；它会在与*dojo*源目录旁边的*release/form*目录中创建新文件：

```
bash build.sh profile=form action=release optimize=shrinksafe releaseName=form
version=0.1.0
```

如果想使用这个自定义的构建版本，只要像下面这样在页面头部通过SCRIPT标签包含压缩后的*dojo.js*和*form.js*文件即可。必须先包含*dojo.js*层，因为*form.js*依赖于它：

```
<html>
  <head><title>Fun With Forms!</title>
    <!--  包含样式表，等等 -->
    <script type="text/javascript" path="relative/path/to/form/dojo.js"></script>
    <script type="text/javascript" path="relative/path/to/form/form.js"></script>
  </head>
  <!--  页面的其余内容 -->
```

如此而已。只要发送两个同步请求就可以把所有JavaScript代码（其中还包含内置的模板）加载到页面中；由prefixes列表指定包含在构建文件中的其他资源则可以通过dojo.require取得。

如果除了*dojo.js*及构建生成的层文件之外，你完全能够保证不需要任何其他JavaScript资源，那么也可以把这些个别的有用资源从版本目录中单独提取出来。但在这样做之前，必须考虑到这些资源与内置CSS主题（如*tundra*）的依赖关系，因为有些样式表的@import语句中可能会使用相对路径和相对URL。

---

**警告：** 在从版本目录中提取有用资源时，Firebug的Net标签对跟踪依赖关系很有帮助。但要注意的是，Firebug不会因样式表中使用的@import语句找不到相应资源而显示404（Not Found）错误。

---

### 创建基于 Rhino 的构建

虽然Dojo构建的正式平台是浏览器，但通过构建系统也可以生成能够在Rhino内部使用的Dojo构建。如果你想在JavaScript中调用Java类，或者使用Helma（*http://dev.helma.org*）之类的工具以JavaScript作为服务器端的脚本编程语言，那么基于Rhino的构建就可能派上用场。

为此，配置文件中只需包含`hostenvType= "rhino"`即可；就这么简单。不过，如果你还想在Rhino构建的版本目录中运行DOH单元测试，那么就必须再包含一个shrinksafe前缀（prefix）。以下是生成自定义Rhino构建的示例配置文件：

```
hostenvType = "rhino";

dependencies = {
  layers : [];
  prefixes : [
    ["dojox", "../dojox"],
    ["shrinksafe", "../util/shrinksafe"]
  ]
};
```

## Dojo 目标套件（DOH）

对Web应用程序进行自动测试的做法越来越普及，究其原因，一是绝对代码量的激增，二是今天的RIA应用程序也日益复杂。DOH在内部使用Dojo，但不是只对Dojo程序才有效；与ShrinkSafe类似，也可以通过DOH针对任何JavaScript脚本创建单元测试，尽管不能使用操作DOM和针对浏览器的功能。

DOH为支持自动测试提供了3个简单的断言结构。每个断言都通过全局对象`doh`提供，相应的结构如下：

- `doh.assertEqual(expected, actual)`
- `doh.assertTrue(condition)`
- `doh.assertFalse(condition)`

在展示复杂的DOH应用示例之前，我们首先来看一个能够在命令行中通过Rhino运行的简单测试套件，以便大致了解一下DOH的功能。下面的这个测试套件展示了DOH可以通过常规函数对象或者测试夹具（text fixture）运行独立的测试。所谓测试夹具，就是一种具有初始化和清理功能的测试方式。

## 不使用 Dojo 的 Rhino 测试套件

闲话少叙，下面我们就看一看这个测试套件。注意，这个套件没有涉及任何Dojo特性；它仅仅使用了doh对象。特别是，在这里使用的是doh.register函数，该函数的第一个参数是模块名称（即与*util*目录相邻的一个JavaScript文件），第二个参数是测试函数或夹具的列表：

```
doh.register("testMe", [

    // 能通过测试的测试夹具
    {
        name : "fooTest",
        setUp : function() {},
        runTest : function(t) { t.assertTrue(1); },
        tearDown : function() {}
    },

    // 不能通过测试的测试夹具
    {
        name : "barTest",
        setUp : function() { this.bar="bar"},
        runTest : function(t) { t.assertEqual(this.bar, "b"+"a"+"rr"); },
        tearDown : function() {delete this.bar;}
    },

    // 能通过测试的函数
    function baz() {doh.assertFalse(0)}
]);
```

假设这个测试套件保存在*testMe.js*文件中，并且位于*util*目录所在的目录下，那么就可在*util/doh*目录中执行下面的命令来运行该套件（虽然这里使用的是构建工具中包含的Rhino *jar*文件，但实际上任何最近的Rhino *jar*文件都可以正常使用）：

```
java -jar ../shrinksafe/custom_rhino.jar runner.js dojoUrl="../../dojo/dojo.js"
testModule=testMe
```

这个命令就是告诉Rhino *jar*通过*runner.js*这个JavaScript文件（DOH的核心），使用指定的Base副本来运行testMe模块。虽然测试套件本身没有涉及任何Dojo特性，但DOH则需要在内部使用Base，因此必须为测试命令提供一个路径。

在试验完了这个DOH示例之后，接下来就可以通过表16-2来了解doh对象提供的更多特性了。

表 16-2：doh 模块的特性

| 特性 | 说明 |
| --- | --- |
| `registerTest(/*String*/group, /* Function \|\| Object */ test)` | 向指定的测试组（group）添加测试或夹具对象 |
| `registerTests(/*String*/group, /*Array*/ tests)` | 自动注册 tests 数组中包含的一组测试 |
| `registerTestNs(/*String*/group, /*Object*/ns)` | 将 ns 对象包含的函数添加到测试组中，但不包含以下划线开头的函数，因为下划线通常表示函数是私有的 |
| `register(/* ...*/)` | 可以根据参数决定并应用适当的注册函数 |
| `assertEqual(expected, actual)` | 用于断言两个应该相等的值 |
| `assertTrue(/*Boolean*/condition)` | 用于断言应该求值为 `true` 的值 |
| `assertFalse(/*Boolean*/ condition)` | 用于断言应该求值为 `false` 的值 |
| `is(expected, actual)` | `assertEqual` 的简写 |
| `t(/*Boolean*/condition)` | `assertTrue` 的简写 |
| `f(/*Boolean*/condition)` | `assertFalse` 的简写 |
| `registerGroup(/*String*/ group, /*Array\|\|Function\|\|Object*/tests, /*Function*/ setUp, /*Function*/tearDown)` | 将 tests 中包含的整个测试组一次性添加到 group 中。如果提供的话，那么使用自定义的 `setUp` 和 `tearDown` 函数 |
| `run()` | 用于以编程方式运行测试 |
| `runGroup(/*String*/groupName)` | 用于以编程方式运行一组测试 |
| `pause` | 用于以编程方式暂停正在运行的测试；随后可以通过 `run()` 恢复测试 |
| `togglePaused` | 用于按顺序暂停和运行测试 |

另外，注意 *runner.js* 文件可以接收表 16-3 中列出的任何选项。

表 16-3：针对 runner.js 文件的选项

| 特性 | 说明 |
| --- | --- |
| `dojoUrl` | 指定 *dojo.js* 文件的路径 |
| `testUrl` | 指定测试文件的路径 |
| `testModule` | 指定应该执行的测试模块列表，以逗号分隔模块名称，例如，`foo.bar, foo.baz` |

## 使用 Dojo 的 Rhino 测试套件

虽然使用DOH时可以不涉及Dojo特性，但通过Rhino来测试Dojo程序也很常见。Core中包含了一些典型的示例，而通过执行*runner.js*运行这些示例无须添加任何参数。默认值会指向 *dojo/tests* 中的测试，并使用 *dojo/dojo.js* 中的 Base 版本。

通过查看Core中的测试文件，读者会发现其用法非常简单。每个文件都以一条指定测试模块名称的`dojo.provide`语句开头，并以`dojo.require`请求被测试的资源，然后使用一系列 register 函数创建用于测试的夹具。

假设有一个自定义的`foo.bar`模块（*/tmp/foo/bar.js*）和一个*testBar.js*测试套件（*/tmp/testBar.js*）。这两个 JavaScript 文件的内容如下。

首先是*testBar.js*：

```
/* 像任何模块一样通过dojo.provide定义要测试的模块名称 */
dojo.provide("testBar");

/* 如果是使用dojo根目录之外的自定义模块，
   那么需要注册模块路径*/
dojo.registerModulePath("foo.bar", "/tmp/foo/bar");

/* 通过dojo.require请求所需的资源 */
dojo.require("foo.bar");

/* 通过register函数注册模块 */
doh.register("testBar", [

    function() { doh.t(alwaysReturnsTrue()); },
    function() { doh.f(alwaysReturnsFalse()); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    function() { doh.is(alwaysReturnsOdd()%2, 1); },
    {
        name : "BazFixture",
        setUp : function() {this.baz = new Baz;},
        runTest : function() {doh.is(this.baz.talk(), "hello");},
        tearDown : function() {delete this.baz;}
    }
]);
```

下面，是*foo/bar.js*中包含的`foo.bar`模块：

```
/* 一组并非很有用的函数 */
dojo.provide("foo.bar");

function alwaysReturnsTrue() {
    return true;
}
```

```
function alwaysReturnsFalse() {
    return false;
}

function alwaysReturnsOdd() {
    return Math.floor(Math.random()*10)*2-1;
}

// 哈哈，居然还有一个“类”
dojo.declare("Baz", null, {
    talk : function() {
        return "hello";
    }
});
```

那么，在 *util/buildscripts* 中运行下面的命令就可以启动这次测试：

```
java -jar ../shrinksafe/custom_rhino.jar runner.js dojoUrl=../../dojo/dojo.js
testUrl=/tmp/testBar.js
```

---

**警告：** 测试套件在请求foo.bar模块之前，首先明确地注册了它的模块路径。对于保存在dojo根目录外部的资源，这个额外的步骤对于查找自定义模块是必需的。

---

如果一切都按计划进行，那么读者应该看到包含所有测试通过或失败信息的测试报告。可以通过相同的方式注册一组共享某些常见设置和清理操作的测试 —— 除了要使用doh.registerGroup函数代替doh.register函数(或者其中一个更具体的函数变体)。

如果想要更好地控制测试的执行过程，可以通过编程方式来暂停和重启测试。为此，要对 *testBar.js* 进行如下修改：

```
/* 加载dojo.js和runner.js */
load("/usr/local/dojo/dojo.js");
load("/usr/local/dojo/util/doh/runner.js");

/* 像任何模块一样通过dojo.provide定义要测试的模块名称 */
dojo.provide("testBar");

/* 如果是使用dojo根目录之外的自定义模块，
   那么需要注册模块路径 */
dojo.registerModulePath("foo.bar", "/tmp/foo/bar");

/* 通过dojo.require请求所需的资源 */
dojo.require("foo.bar");

/* 通过register函数注册模块 */
doh.register("testBar", [
    function() { doh.t(alwaysReturnsTrue()); },
    function() { doh.f(alwaysReturnsFalse()); },
```

```
        function() { doh.is(alwaysReturnsOdd()%2, 1); },
        function() { doh.is(alwaysReturnsOdd()%2, 1); },
        function() { doh.is(alwaysReturnsOdd()%2, 1); },
        {
            name : "BazFixture",
            setUp : function() {this.baz = new Baz;},
            runTest : function() {doh.is(this.baz.talk(), "hello");},
            tearDown : function() {delete this.baz;}
        }
]);

doh.run();

    /* 按照需要暂停或重启测试…… */
```

尽管这个示例并没有利用testBar本身就是通过dojo.provide定义的事实，但就和其他通过dojo.provide定义的模块一样，我们都可以非常容易地使用dojo.require将多个测试整合到一起。

虽然使用Rhino也可以运行异步测试，但我们要把异步测试的示例放到下一节介绍。毕竟，异步测试对于涉及网络输入/输出和事件（如动画）的基于浏览器的测试才是最有用的。

## 基于浏览器的测试套件

DOH不仅支持基于Rhino运行测试，而且也提供了一个套件来支持在浏览器窗口中自动运行测试。为此，测试人员只要以普通HTML页面的形式定义一个测试，然后在runner的URL中通过查询字符串参数把这个测试页面加载到DOH测试运行程序中即可。在内部，测试运行程序中的JavaScript代码会分析查询字符串，提取出其中的配置值（例如，testUrl），并利用它们把测试页面注入到一个框架中。

当然，不通过DOH测试运行程序也可以运行基于浏览器的测试。不过，要是你愿意通过控制台输出了解测试结果，那么恐怕就得不到美观的视觉显示，也听不到可选的Homer Simpson提示音效果了（译注1）。

### 浏览器测试示例

下面就是一个以普通HTML页面形式定义的测试示例。注意，这个示例中使用了Dojo的本地版本，因为到Dojo1.1为止，DOH还没有通过AOL的CDN分发：

---

译注1： Homer Simpson是经典动画剧集The Simpsons中的主人公，配音者为Dan Castellaneta。

```
<html>
    <head><title>Fun with DOH!</title>

        <script
        type="text/javascript"
        src="local/path/to/dojo/dojo.js">
        </script>

        <script type="text/javascript">
            dojo.require("doh.runner");

            dojo.addOnLoad(function() {
                doh.register("fooTest", [
                    function foo() {
                        var bar = [];
                        bar.push(1);
                        bar.push(2);
                        bar.push(3);

                        doh.is(bar.indexOf(1), 0); //not portable!
                    }
                ]);

                doh.run();
            });
        </script>

    </head>
    <body></body>
</html>
```

## 异步浏览器测试示例

几乎任何有价值的Web应用程序测试套件都要涉及一些依赖异步条件的测试，例如，等待动画发生、服务器端回调函数启动等。例16-4展示了使用DOH创建异步测试的过程。其中的关键概念在于，doh.Deferred（基本上就是稍加改造的普通dojo.Deferred）是DOH内部的，因此没有外部依赖关系。有关Deferred的详细讨论，请参考第4章。

在展示相关的示例代码之前，我们首先介绍一下通过DOH实现异步测试的基本模式。

- 创建用于验证异步函数（返回dojo.Deferred）执行结果的doh.Deferred。
- 调用异步函数返回的dojo.Deferred并保存一个对它的引用。
- 为dojo.Deferred添加callback和errback，以便将异步函数的结果传递给doh.Deferred的callback和errback。

例16-4：异步测试的大致框架

```
doh.register("foo", [

    function() {
        var dohDfd = new doh.Deferred();
        var expectedResult = "baz";

        var  dojoDfd = asynchronousBarFunction();
        dojoDfd.addBoth(function(response, io) {

            //必要时引用dohDfd……
            if (response == expectedResult) {
                dohDfd.callback(true);
            }
            else {
                dohDfd.errback(new Error( /* …… */));
            }
        });

        //……返回dohDfd
        return dohDfd;
    }
]);
```

根据具体的测试约束条件，有可能需要明确提供一个timeout值，以确保涉及超时控制的异步操作符合测试标准。无论如何，问题的关键在于实现异步测试并没有想像得那么复杂；由于Deferred简化了大部分复杂性，因此测试人员只要关注手边的任务就行了。

# 性能问题

**注意：** 本节介绍一些与提升前台性能有关的简便易行的技巧。如果读者想全面了解与改进性能相关的主题，我们推荐Steve Souders编著的《High Performance Web Sites: Essential Knowledge for Front-End Engineers》（O'Reilly）一书。该书是一本适合快速阅读并能够真正解决问题的好书。另外，该书中的很多内容在*http://developer.yahoo.com/performance/rules.html*中也有介绍。

尽管规范的JavaScript代码对于保证Web应用程序的响应速度绝对有必要，但从产品的角度来说，还有其他一些特别值得重视的问题。其中，仅优化Web应用程序的性能这个主题，恐怕就得需要一本书的篇幅来阐述。在此，我们有选择性地介绍一些提升Web应用程序性能的简便易行的手段：

*Dojo构建工具*

构建工具可以自动完成许多开发人员费力却做不好的基本任务。构建过程能够缩减

源代码文件的大小，降低有效负载总量，最终通过将多个JavaScript文件合并为层及内置模板字符串达到显著减少HTTP延时的目的。

*延迟加载*

虽然本章已经就尽可能减少应用程序层文件的重要意义谈论了很多，但延迟加载有时候仍然是最合适的选择。例如，对于用户很少使用而且代码量又不多的某些特性，完全可以不把它们打包到层中，而是在需要时再通过dojo.require动态加载它们。

另一种与延迟加载有关的情形是合理使用布局部件动态加载内容。例如，可以在初始状态下只加载一个TabContainer的可见标签页，然后要么根据请求动态加载其他内容，要么等到页面其他部分都加载完成后再获取其他标签页。ContentPane部件是实现延迟加载的常用手段。

*Web服务器配置*

配置Web服务器，通过响应的Expires头部设置一个较长的过期时间，可以让Web浏览器主动缓存JavaScript文件和其他静态内容。另外，对于*gzip*压缩这样的常用配置选项，最好也能够善加利用。

*最大化静态内容*

由于服务器提供静态内容的速度最快，因此应用程序中的静态内容越多，Web服务器响应每个请求的时间就越短。对于能够通过cookie或XHR请求生成用户专有内容的页面，应该尽可能使用静态HTML文件。例如，如果登录页面的唯一差异之处就是包含不同用户信息的几百字节文本，那么就大可在此使用静态页面，然后再通过脚本异步获取针对用户的少量数据；最好不要动态生成整个页面。

*性能分析*

如果页面看起来特别慢，或者页面加载后性能总是波动，可以使用Firebug内置的性能分析程序找出耗时最长的JavaScript逻辑，然后再考虑如何对相应的代码进行优化。

### 跨域（XDomain）构建的优点

如果你选择使用跨域构建来创建应用程序，那么尽管一开始没有那么明显，但实际上会获得很多潜在的好处：

- 可以在专用主机上托管Dojo，并在多个应用程序中共享该构建，无论这些应用程序是否在相同的域中。
- 页面加载时碰到的`dojo.require`语句满足异步而非同步（使用默认构建时是同步）的条件，由于异步请求不会阻塞页面，因此能够提高页面加载速度。

- 某些浏览器（如IE）在默认情况下会限制每个子域只能同时打开两个连接，而使用跨域构建本质上能够支持应用程序同时打开双倍的连接，两个用于Dojo，两个用于本地域的资源。
- 如果多个应用程序都使用跨域构建，那么客户端需要等待的HTTP潜伏总时间很可能减少，因为这些客户端浏览器能够在本地缓存的内容总量会增加。

### 不要过早优化

最后，我们要对读者给出的忠告是：不要过早地优化应用程序；或者说，永远不要毫无来由地、盲目地优化应用程序。在优化之前，最好能够拿到作为依据的性能分析报告或者服务器日志。单就性能优化来说，人类的本能往往不会起什么好作用。同时，不要忘记：Firebug是个值得依赖的好伙伴。

## 小结

在学习完本章之后，读者应该：

- 能够使用Dojo的构建工具为Web应用程序创建统一的、压缩过的层。
- 熟悉创建自定义构建时最常用的配置选项。
- 知道*dojo.js*通常保持为一个单独的JavaScript文件；它不应该出现在自定义的层中。
- 能够使用DOH为JavaScript函数编写单元测试。
- 更加熟悉Rhino并理解它在构建工具及DOH中扮演的角色。
- 了解虽然ShrinkSafe和DOH是Dojo工具箱的重要组成部分，但它们的适用对象并不局限于Dojo，也可以在其他地方使用它们。
- 掌握提升Web应用程序性能的一些简便易行的技巧。

附录 A

# Firebug 入门教程

作为一名Web开发人员，Mozilla Firefox浏览器（*http://www.getfirefox.com*）的Firebug扩展（*http://www.getfirebug.com*）是一个不容错过的好工具，尤其是你在使用高效JavaScript工具箱，通过Firebug提高自己的调试效率时，绝对是明智的选择。本附录将系统地介绍Firebug的一些关键特性，希望对读者掌握调试Web应用程序（或者因为好玩而解构某个页面）的最佳方式有所帮助。

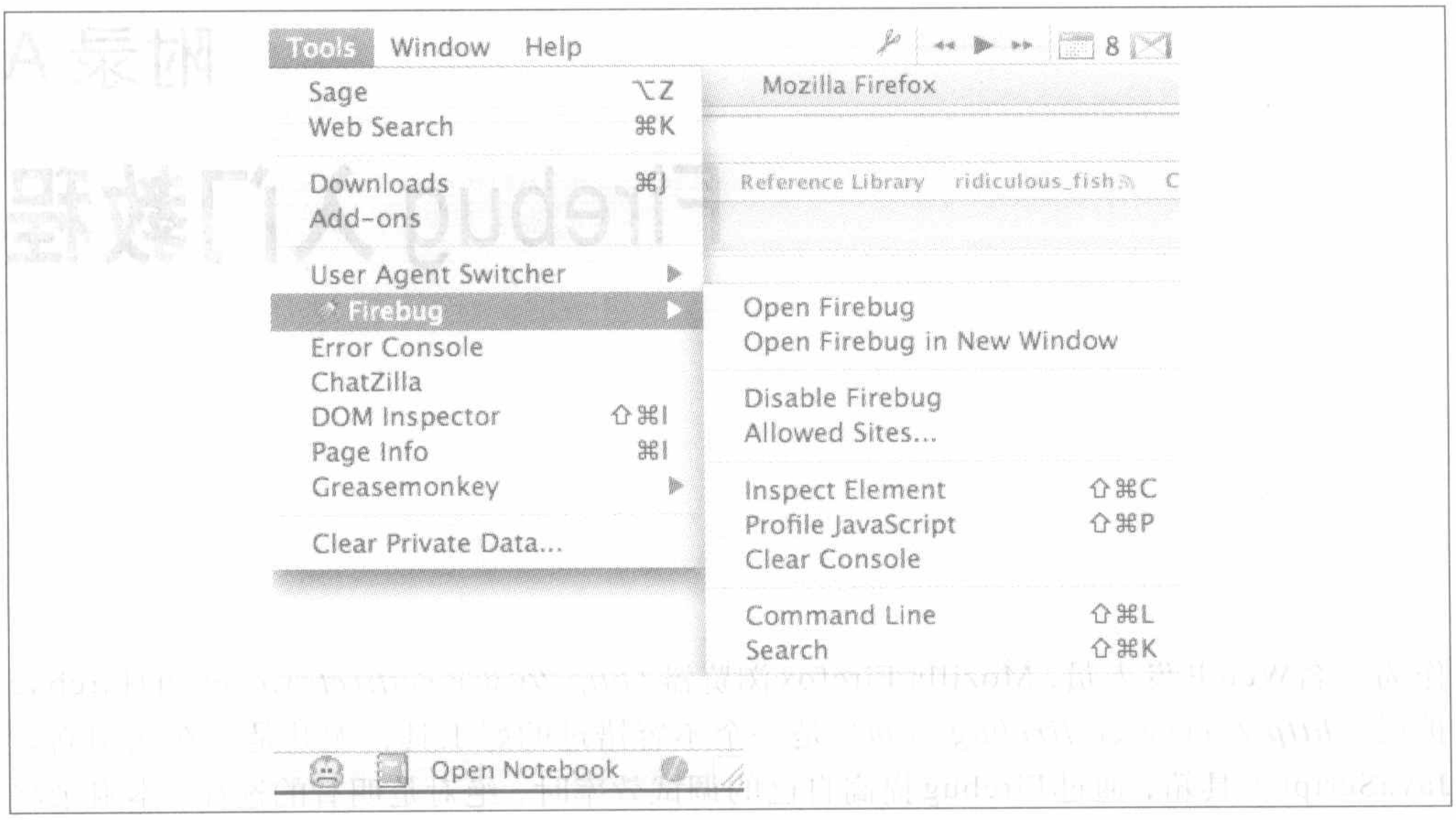
**注意：** 本附录只是一篇针对Firebug新手的入门教程，并非是面面俱到的使用指南。

## 安装

与安装其他Firefox扩展一样，安装Firebug也很简单。在Firefox中打开*http://www.getfirebug.com*，然后单击“INSTALL FIREBUG”按钮。此时，屏幕顶部可能会出现一个阻止安装的黄色警告条，单击“Edit Options...”按钮并同意安装。重启Firefox之后，“工具”菜单中应该有一个Firebug项，而且浏览器右下角也应该有一个Firebug图标，如图A-1所示。

## 允许还是不允许

Firebug包含许多令人叫绝的特性，例如，与DOM操作和JavaScript剖析有关的特性。不过，在介绍这些特性之前，我们还是先来看一看应该怎样设置，才能在特定的站点中自动启用或禁用Firebug。虽然这个特性常常被人忽视，但它却非常方便，因为它允许我们根据开发的需要来自定义Firebug。例如，可以将Firebug设置为默认禁用，因为它会




图A-1：左：安装Firebug之后，“工具”菜单中会出现相应的项，该项包含一些标准的选项；右：除了“工具”菜单项之外，浏览器窗口右下角还会出现一个小图标

明显拖慢那些大量使用JavaScript的Web应用程序（例如，Gmail）。但是，我们可以为Firebug设置一组指向当前正在开发的应用程序的URL列表，让它能够针对这些URL自动启用，以免自己在不同的浏览器标签中手动切换它的启用/禁用状态。当然，如果你觉得在另外一些站点中也需要使用Firebug，只要把这些站点的URL添加到被允许的站点列表中即可。

右击浏览器窗口右下角的Firebug图标，打开关联菜单。菜单中会包含配置Firebug默认启用站点的选项。具体一点说，菜单中包含如图A-2所示的选项（译注1）：

*禁用（Disable Firebug）*

禁用Firebug，除非取消对该项的选定，或者导航到了一个在Firebug的“站点(Allowed Sites . . .)”菜单中指定的站点。

*为……禁用控制台（Disable Firebug for . . .）*

阻止在当前站点中启用Firebug，并将该站点添加到禁用站点列表——在默认启用Firebug而只想针对自定义站点列表禁用它时有用。

---

译注1：在Firebug1.2中，下面列举的这些选项不会出现在上述右键菜单中。要找到这些选项，先单击浏览器窗口右下角的Firebug图标打开Firebug窗口，然后单击左上角“控制台(Console)”标签右侧的小三角形图标。

- *站点（Allowed Sites . . . ）*

  打开“启用或禁用控制台（Firebug Allowed Sites）”窗口，在其中可以指定默认启用或禁用的站点列表。

“在新窗口中打开 Firebug（Open Firebug in New Window）”会在一个独立的窗口中打开Firebug —— 如果你觉得出现在浏览器窗口底部的默认面板看着不舒服，可以选择此项。

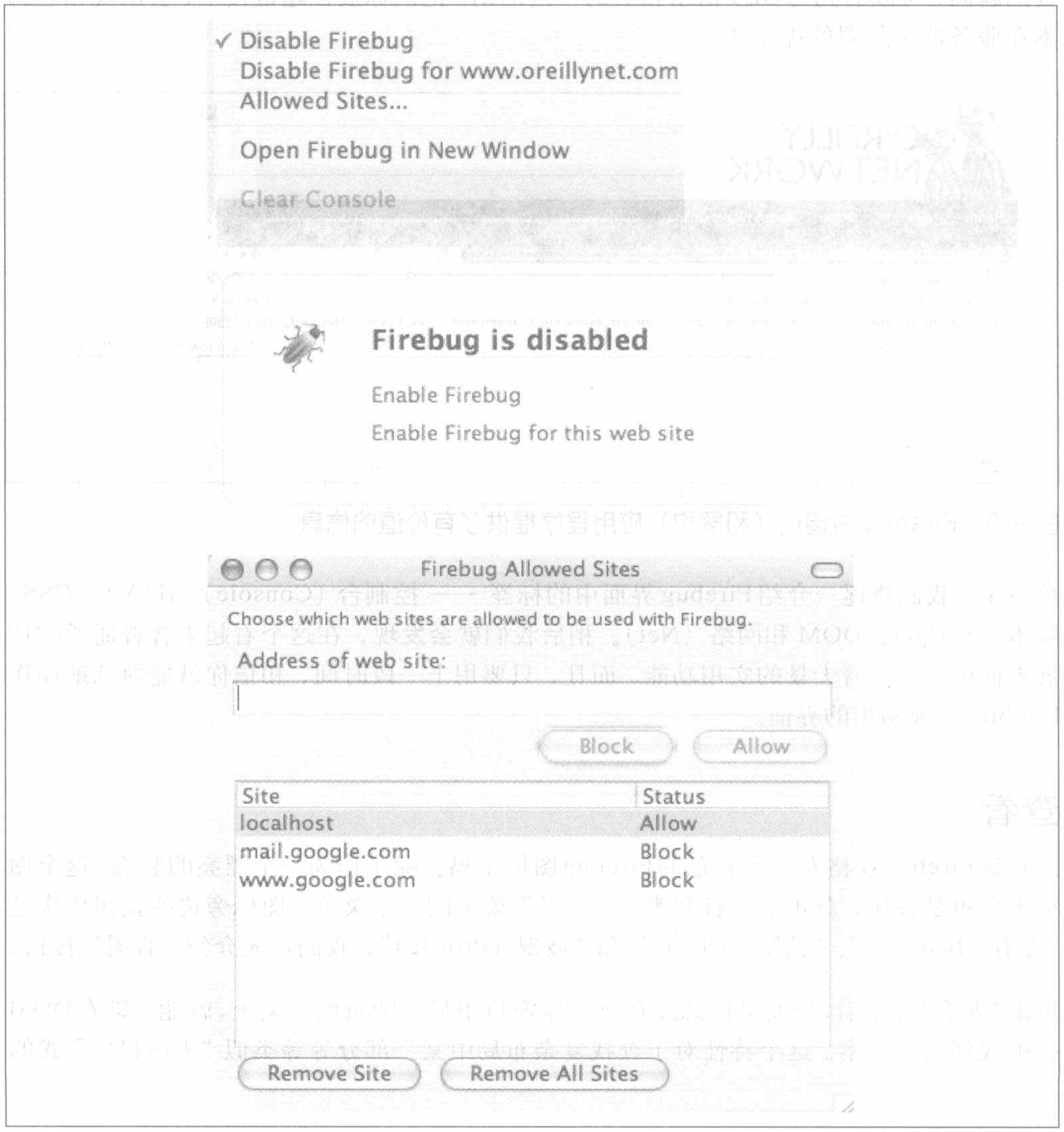



图 A-2：配置 Firebug 的各种选项

# 好戏上演

既然已经为特定站点设置好了Firebug，接下来我们就体验一些Firebug中的独到特性。要选择一个起点，位于 *http://www.oreillynet.com* 的O'Reilly Network主页就不错。

在打开*http://www.oreillynet.com*并启用Firebug之后，浏览器窗口右下角的灰色图标变成了带对勾的绿色圆形图标，如图A-3所示（译注2）。在选择“控制台（Console）”标签的瞬间，可以看到与GET请求相关的一些活动。把鼠标放在超链接上，会出现相应脚本在服务器上位置的提示条。

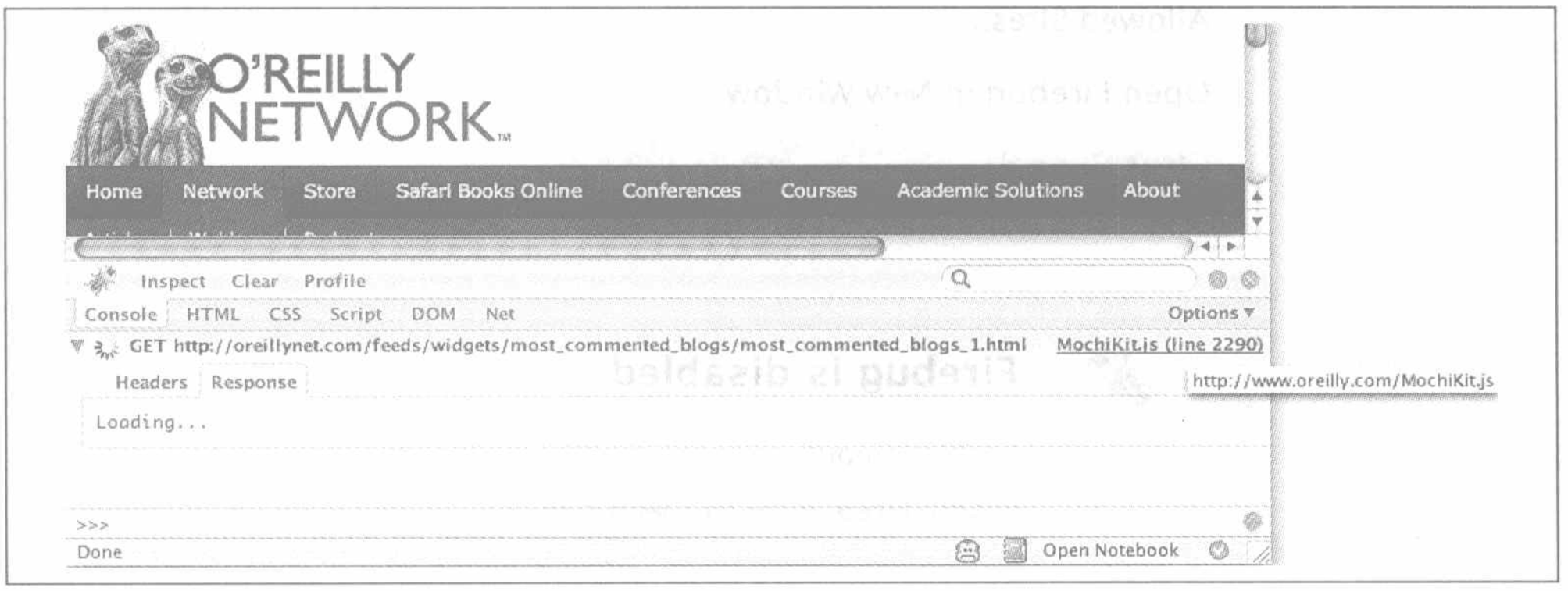



图 A-3：Firebug为调试（和解构）应用程序提供了有价值的信息

接下来，我们将逐一介绍Firebug界面中的标签 —— 控制台（Console）、HTML、CSS、脚本（Script）、DOM和网络（Net）。稍后我们就会发现，在这个看起来普普通通的扩展界面中，包含着大量的实用功能。而且，只要用上一段时间，相信你就能熟练地操作Firebug直观易用的界面。

# 查看

注意到Firebug窗格左上角彩色的Firebug图标了吗？除了作为一个漂亮的装饰，这个图标本身也是按钮，单击它会打开类似“工具”菜单的一个菜单。图标旁边的按钮依次是“查看（Inspect）”、“清除（Clear）”和“概况（Profile）”。我们首先介绍“查看”按钮。

单击“查看”按钮后，只要鼠标悬停在浏览器窗口中显示的页面元素上，就能立即在DOM树中找到这个元素。这个特性对于查找复杂布局中某一部分等等类似“大海捞针”式的

---

译注2：在Firebug1.2中，图标为彩色的臭虫。

搜索非常有用。单击“查看”按钮后的效果如图 A-4 所示，此时按钮处理凹陷状态。主菜单中始终包含所有典型的主菜单项，并且总是位于左上角。而无论选择哪个标签始终位于主菜单旁边的“查看”按钮，则是一个非常方便的常用特性。当鼠标悬停在页面中的不同元素上面时，注意观察 Firebug 切换到“HTML”标签（tab）并显示的 HTML 和 CSS。与当前悬停项相关的 HTML 代码片段在“HTML”标签中会突出显示出来，从而方便用户查看整个元素及其环境。

单击页面中悬停的元素会导致“HTML”标签中的相关内容突出显示，Firebug 同时停止对悬停事件的响应，以便用户能够重新控制鼠标并且不会丢掉刚刚选择的元素。在找到特定的 DOM 元素后，通过“HTML”标签几乎可以对该元素执行任何操作，如动态编辑节点的内容、添加属性、删除属性、查看特定节点的 CSS 样式等。

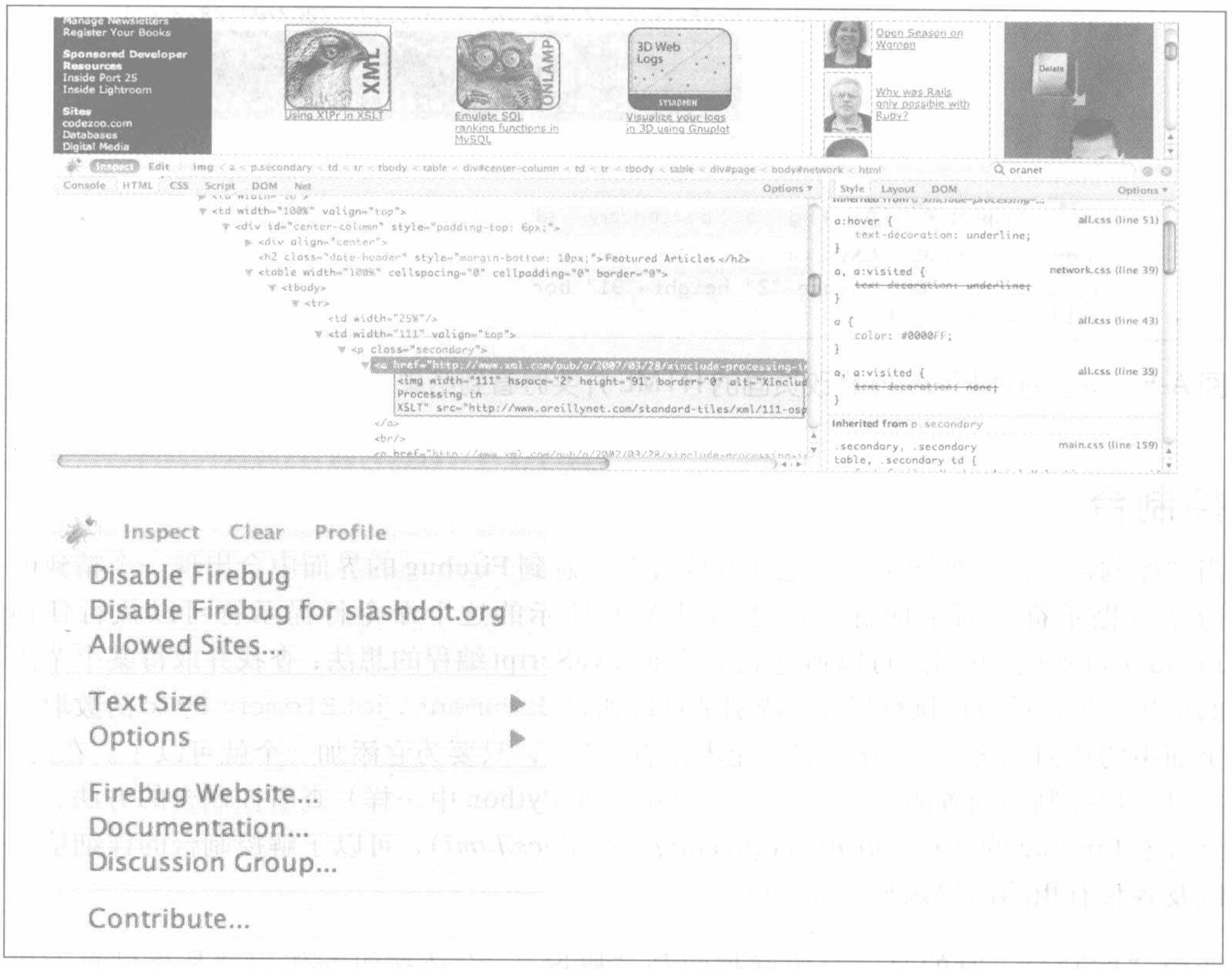



图 A-4 ： 单击“查看”按钮之后的效果及主菜单

读者可以通过探索 O’Reilly Network 主页的各个部分加深对“查看”特性的理解。在这个过程中，特别值得尝试的是修改 DOM 树中的元素，因为所作的修改会在浏览器中立

即反映出来。例如，图 A-5 展示了把主页中一幅图像调整到很宽的效果。而通过单击 HTML 代码直接修改标签的任何属性，也都可以立即看到效果。如果想为DOM 元素添加一个有效的属性，可以先单击该元素，然后单击“编辑”按钮。

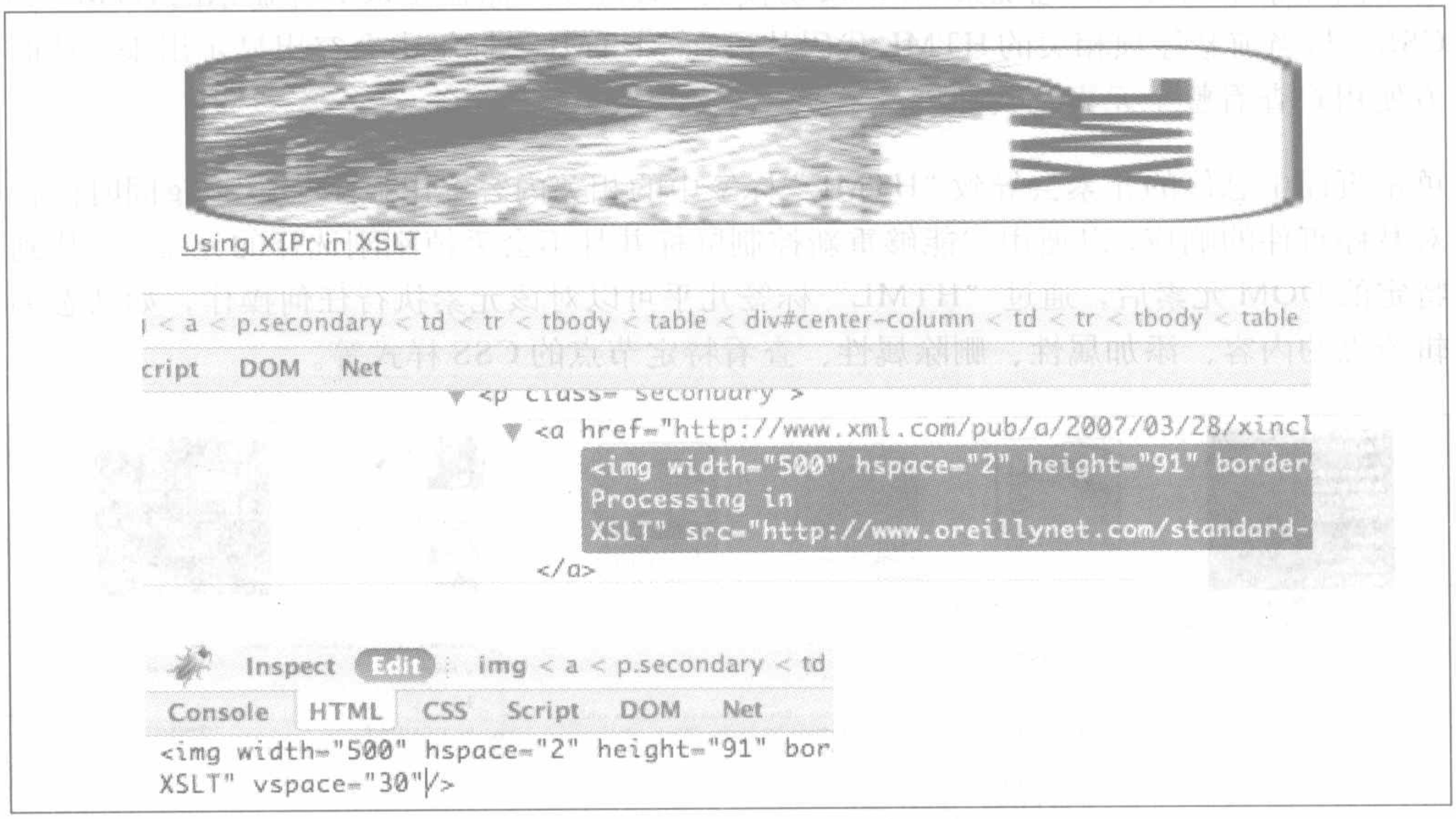



图 A-5：可以通过 Firebug 修改页面的 HTML 并实时看到结果

## 控制台

当“控制台”标签被选中时，也许有读者会注意到 Firebug 的界面中会出现一个精致的命令行提示符。简单地讲，通过如图 A-6 所示的这个命令行提示符可以执行任何 JavaScript 代码。因此，可以通过它来验证JavaScript 编程的想法；查找并取得某个节点的引用，然后对节点执行操作。特别是可以通过 `document.getElementById` 函数取得页面中的任何内容 —— 万一哪个元素没有 `id` 值，只要为它添加一个就可以了。在此，可以使用控制台内置的 `console.dir`（就像在 Python 中一样）查看控制台的方法。通过查看 Firebug 的文档（*http://getfirebug.com/docs.html*），可以了解控制台的详细信息以及各种有用的内置函数，如 `dir`，等等。

选中“控制台”时的另一个关联按钮是“概况”，而该按钮的作用就是提供页面中 JavaScript 执行的相关信息，如图 A-7 所示。单击这个按钮可以启动概况分析，而再次单击它会停止分析并显示搜集到的统计信息。还能有比这更简单的吗？下面这个访问 *http://jobs.oreilly.com* 时截取的快照，显示了在单击该页面左侧导航条时执行的相应 JavaScript 脚本的状态信息。

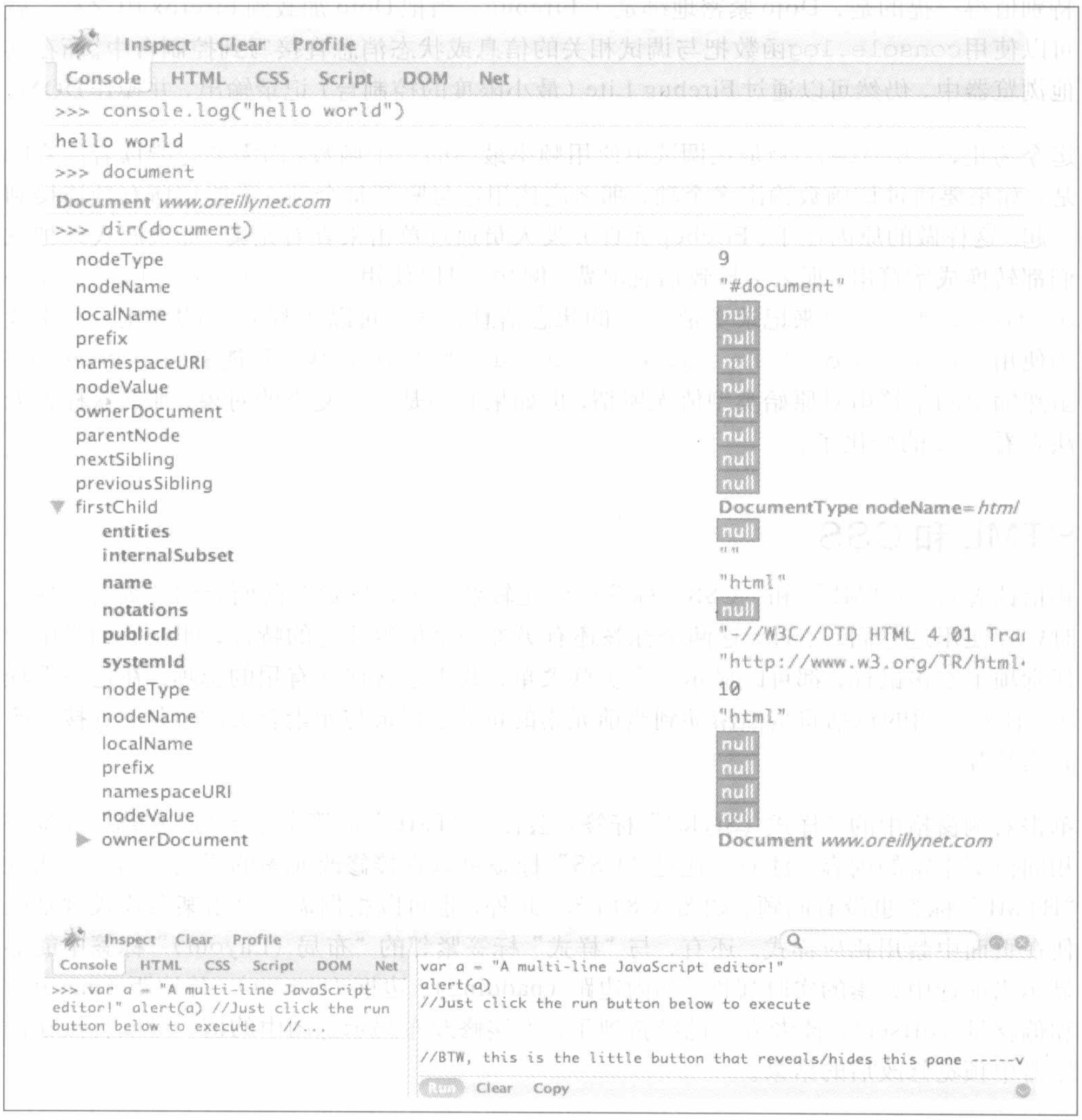



图 A-6 左：控制台是对任何页面进行脚本编程的界面；右：单击命令行提示符最右侧类似^的小按键，可以显示多行 JavaScript 编辑器

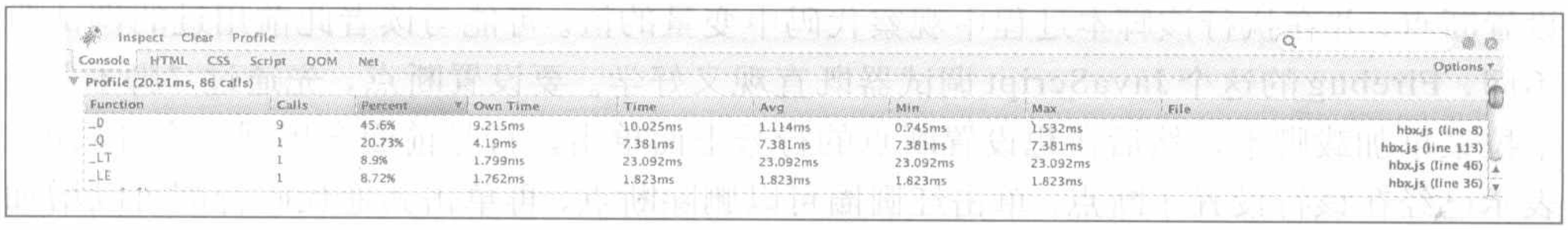



图 A-7：Firebug 中带有非常方便的分析程序，能够辅助开发人员分析页面中 JavaScript 的执行效率

特别值得一提的是，Dojo紧密地绑定了Firebug。当把Dojo加载到Firefox中之后，就可以使用console.log函数把与调试相关的信息或状态消息直接写到控制台中。而在其他浏览器中，仍然可以通过Firebug Lite（最小限度的控制台）记录输出，并操作DOM。

迄今为止，console.log是在调试中使用频率最高的一个函数；而需要提醒读者注意的是，如果要通过该函数输出多个项，那么应该用逗号隔开每个项，不能把所有项连接到一起。这样做的原因在于，Firebug允许开发人员通过单击来查看元素，如果隐式地把它们都转换成字符串，那么会导致信息浪费。例如，可以使用console.log("the value of foo is", foo)来记录变量foo的状态信息，从而可以了解foo的变化。如果改为使用console.log("the value of foo is "+foo)，结果只能看到一个字符串。虽然输出的字符串对原始类型值无所谓，但如果foo是一个复杂的对象，那么这样就无法查看foo的变化了。

## HTML和CSS

相信读者对“HTML”和“CSS”标签已经比较熟悉了，毕竟在前面介绍“查看”按钮时曾经使用过它们。不过，这两个标签还有另外一个值得注意的特性，即在它们当中的任意项上右击鼠标，都可以显示一个关联菜单，其中包含许多有用的选项。在这些选项中，比较有用的包括将页面滚动到当前元素的位置、记录与元素有关的事件、直接修改元素等等。

单击右侧窗格中的“样式（Style）”标签，会在“HTML”标签中显示与“CSS”标签中相同但更丰富的内容。注意，通过“CSS”标签可以直接修改元素的样式；当然，通过“HTML”标签也没有问题，如图A-8所示。此外，也可以根据需要单击某条样式规则以便在页面中禁用相应样式。还有，与“样式”标签紧邻的“布局（Layout）”标签中能够显示当前选中元素的实时属性，如内边距（padding）、边框（border）、外边距（margin）和偏移量（offset）。读者或许已经猜到了，直接修改布局示意图中的值，可以直接在浏览器中预览修改后的结果。

## 脚本和DOM

Firebug的“脚本”标签本身是一个强大的JavaScript调试器，可以在其中为特定的脚本设置断点，并在执行该脚本过程中观察代码中变量的值。可能与读者此前用过的调试器不同，Firebug的这个JavaScript调试器既直观又好学。要设置断点，先通过“脚本”的下拉菜单加载脚本，然后在想设置断点的行号上面单击。行号前面会出现一个红圆圈，表示已经在该行设置了断点。单击红圆圈可以删除断点，再单击其他代码行则可以增加断点。而且，通过旁边配套的“监控（Watch）”标签可以输入变量名或监控表达式。然

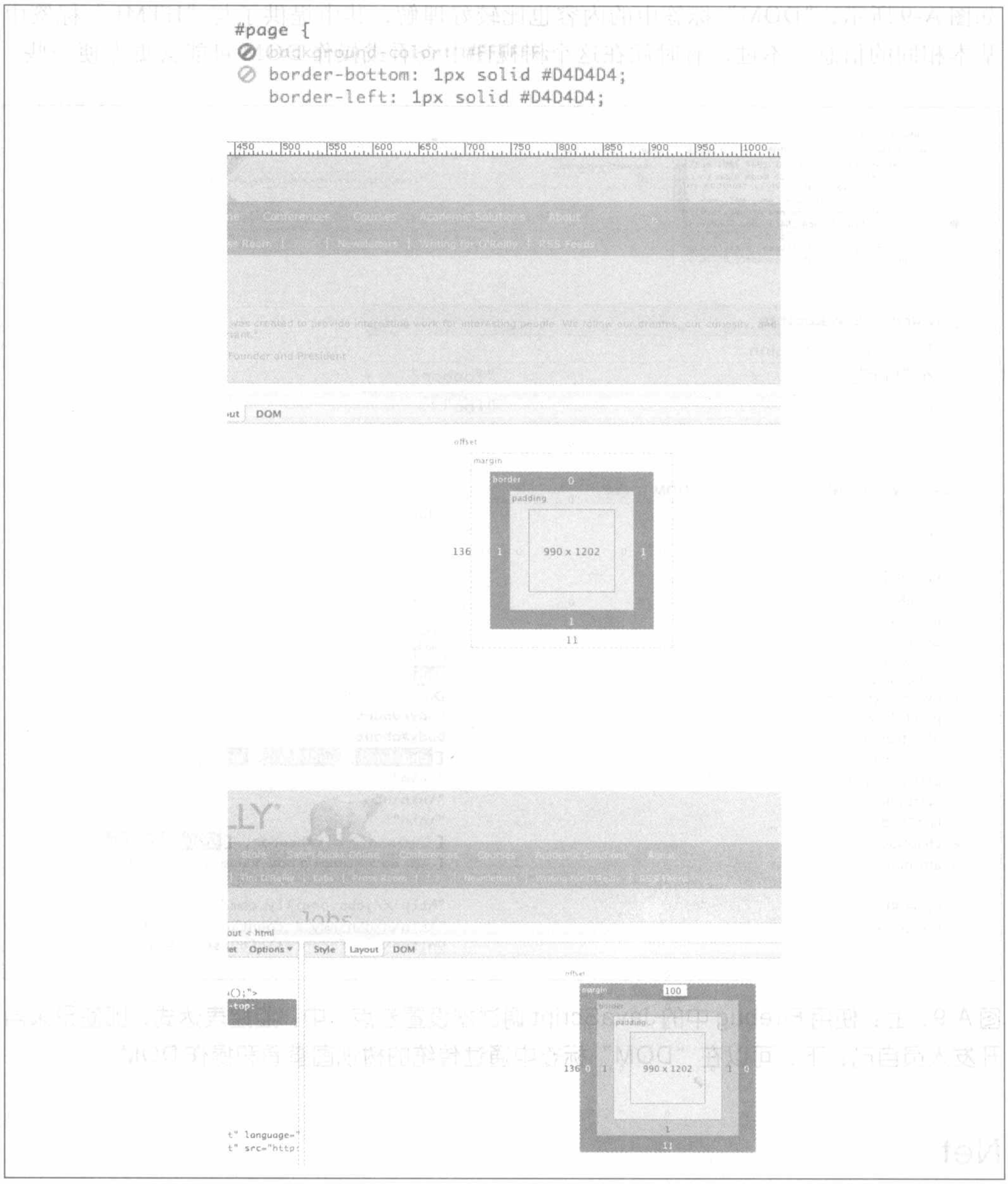



图 A-8 上：禁用元素的样式；中：Firebug 的“布局”标签；下：修改 DOM 元素的外边距或其他相关的布局属性（这里展示的是将上外边距修改为 100px）

后，刷新页面执行脚本，就可以看到变量值在经过不同断点时的变化情况了。有了这个工具，调试 JavaScript 是不是更简单了？

如图A-9所示，“DOM”标签中的内容也比较好理解，其中提供了与“HTML”标签中基本相同的信息。不过，有时候在这个树视图中查看或操作DOM可能会更方便一些。


图A-9：上：使用Firebug中的JavaScript调试器设置断点；中：监控表达式，即使是来自开发人员自己；下：可以在“DOM”标签中通过传统的树视图查看和操作DOM

## Net

到现在为止，我们已经介绍了Firebug中的大多数既好学又实用的特性；但是，图A-10中所示的“Net”标签才是“老鼠拉木锨”中所说的“木锨”。基本上说，“Net”标签可以为我们分析Web应用程序的性能提供所有信息，因为这些信息都与页面加载有关；而且，各种媒体类型都按照逻辑分类为组，例如，CSS、JavaScript、图片等等。作为一种特殊情况，甚至还为查看与Ajax请求相关的网络信息提供了一个XHR组。总而言之，

“Net”标签为发现潜在的性能瓶颈和分析与内容有关的加载时间提供了极有价值的信息。如果读者开发的项目包含大量动态元素，那么手边有这样天然的网络分析器应该会非常方便。

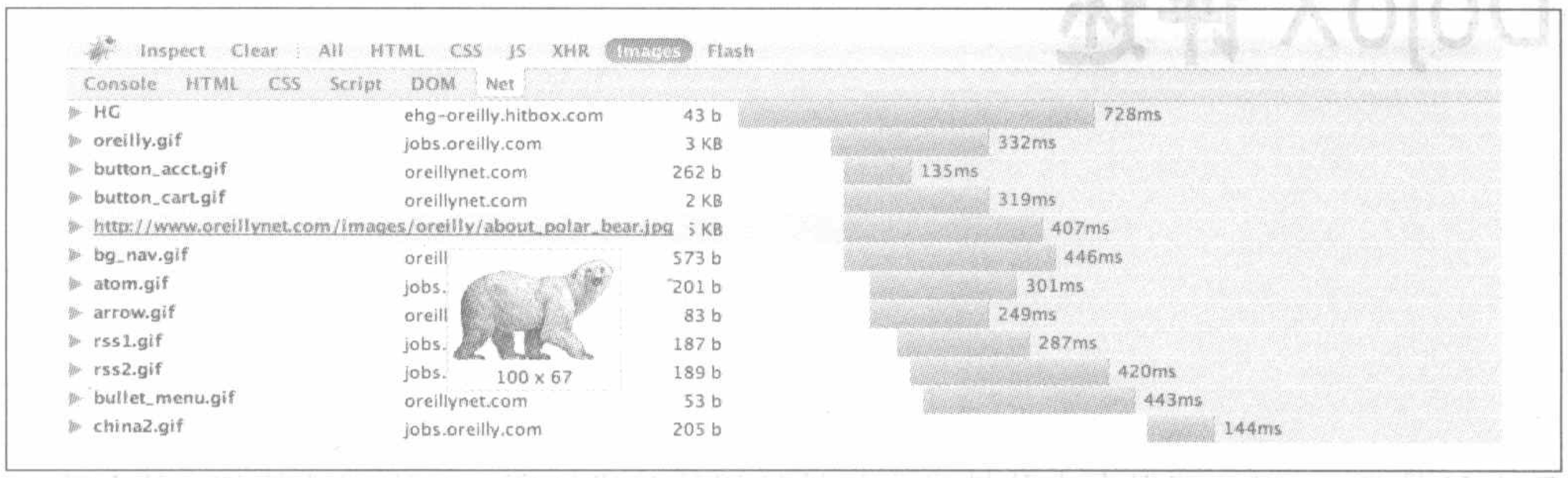



图 A-10：Firebug 的“Net”标签中按类别列出了页面中加载的媒体并提供了加载时间信息

## 继续，动手分析

前面已经介绍了 Firebug 的各种特性，但要用好这些特性，最好的方式就是实际动手分析一些页面的设计。比如，可以选择一个布局稍微复杂些的站点，投入一些时间把设计人员的思路搞清楚。除了能够加深对客户端与服务器通信的理解之外，相信在其他方面你也会受益匪浅。然后，可以试着简单地调试一些自己开发的项目 —— 凭你目前掌握的这些 Firebug 技能，应该不费吹灰之力就可以解决可能存在的问题。（不过，需要重申的是，本附录不是 Firebug 的全面使用指南，它只能作为读者使用 Firebug 的起点。最后，也希望它能对读者认识到 Firebug 在节省开发时间方面的重要性有所帮助。）

# 附录B

# DojoX评述

DojoX是Dojo工具箱保存实验性或专门性扩展的正式场所。与本书其他部分详细介绍Base、Core、Dijit和Util不同，本附录只是以评述的方式介绍DojoX。事实上，要全面深入地介绍DojoX，没有一本书（一本厚度两倍于本书的书）的篇幅是做不到的。不过，在透彻理解了工具箱的其他组成部分之后，从DojoX中找出并运行一些有用的特性应该就非常简单了。

---

**注意：** 本书作者在*http://pipes.yahoo.com/ptwobrussell/dojo_goodness*（译注1）中开辟了一个专栏，名为“Dojo Goodness”，其中讨论了DojoX的子项目。因此，读者可以通过该文章了解有关DojoX的更多信息。

---

DojoX以子项目为单位进行管理，而且这些项目所处的状态也有很大区别。虽然某些子项目（如*cometd*和*charting*）非常稳定，但其他项目很大程度上还处于初创阶段，因此DojoX就像是这些项目的试验场。不过，DojoX中的所有子项目也有一些共性，即它们都应该包含一个*README*文件，用来说明项目的基本情况、版本信息，以及作者的联系信息。DojoX子项目有可能会依赖Base、Core或Dijit；然而，有一些也可能是完全独立的项目。与Dijit不同的是，DojoX不保证其子项目的易访问性和国际化能力。而且，与Dijit比较一致的实现方式相比，DojoX的整体实现风格也存在较大的差异。

还有一个必须注意的地方，即Dojo工具箱的其他部分目前都有严格的广度和深度覆盖面；但是，DojoX不同。表B-1给出了Base、Core、Dijit和DojoX中包含的函数（包含行内声明的匿名函数）及语句的粗略统计数据，统计方法是：

---

译注1：在翻译本书时，译者在此链接中找不到相关文章。不过，读者也可以访问*http://dojotdg.zaffra.com/2008/09/10-helpings-of-dojo-goodness/*或*http://tinyurl.com/dojo-tdg-forum*获取相同的信息。

```
grep -rc 'function' * | grep -v \.svn | cut -d : -f 2 |  awk '{for (i=1; i<=NF; i++)
s=s+$i}; END{print s}'
```

和

```
grep -rc '\;' * | grep -v \.svn | cut -d : -f 2 |  awk '{for (i=1; i<=NF; i++)
s=s+$i}; END{print s}'
```

表 B-1：对 Dojo1.1 中包含的函数和语句的粗略估算（单位：千个）

| Base | | Core | | Dijit | | DojoX | |
|---|---|---|---|---|---|---|---|
| 函数 | 语句 | 函数 | 语句 | 函数 | 语句 | 函数 | 语句 |
| 0.7 | 2.2 | 1.9 | 9.5 | 1.6 | 15.1 | 7.1 | 54 |

通过这些粗略的统计数据，可以看出DojoX中包含着数量惊人的源代码。无论从广度上还是从深度上看，DojoX提供的覆盖面都是令人难以置信的。而且，由于DojoX中包含的子项目有着鲜明的前沿性和专用性，因此 DojoX 可能要比 Dijit 更令人神往。本书虽然没有详细讨论DojoX，但这并不说明DojoX中不包含那些符合你需要、能够为你节省大量时间的优秀特性。

表B-2中列出了到Dojo1.1为止DojoX中包含的子项目。不过，鉴于DojoX的高度易变性，了解 DojoX 最新动态的最好方式，莫过于从 *http://archive.dojotoolkit.org/nightly/* 中下载每夜构建，并直接查看相应的 *README* 文件。

表 B-2：DojoX 包含的子项目

| 子项目 | 简介 |
|---|---|
| *analytics* | 分析和客户端监控系统，可用于将不同的事件（例如，鼠标单击、停顿行为、`console.*` 消息）发送到服务器，以便记录 |
| *av* | 支持 Flash 和 QuickTime 影片的音频 / 视频项目 |
| *charting* | 基于 *dojox.gfx* 和 *dojox.gfx3d* 的高级图表引擎 |
| *collections* | 包含一组用于增强数据结构（例如，栈、集合、队列）的函数，以及为散列、数组等提供的增强功能 |
| *color* | 对 CMYK、HSL 和 HSV 等颜色空间提供支持 |
| *cometd* | 对 Bayeaux 协议的实现。Bayeaux 协议是一个从服务器到客户端的低延时的数据传输技术 |
| *data* | 支持实现 `dojo.data` API 的自定义数据源，如 FlickrRestStore、XmlStore、CsvStore 等，同时也为 `dojo.data` 提供了额外的实用功能 |
| *date* | 一种占位符，常见于其他编程语言中的日期操作（例如格式化程序 —— formatter）或者 PHP 等服务器端技术 |

表B-2：DojoX包含的子项目（续）

| 子项目 | 简介 |
| --- | --- |
| *embed* | 方便嵌入那些通常需要使用OBJECT或EMBED标签嵌入的外部对象 |
| *dtl* | 致力于完全实现Django Template Language（Django模板语言）的项目 |
| *encoding* | 一组支持常用编码算法（如加密、摘要和压缩）的例程 |
| *flash* | 致力于方便地把Flash功能扩展到DHTML环境下的项目 |
| *form* | 一组有用的表单部件，包括密码验证器、使用复选框而非Ctrl+单击的多选下拉列表等 |
| *fx* | 一组动画效果，扩展并增强了Base和Core的功能 |
| *gfx* | 一个易移植的2D图形库，利用了VML、SVG等技术，提供生成静态或动态高级图形的支持 |
| *gfx3d* | 一个易移植的3D图形库。基于*gfx*提供的特性构建 |
| *grid* | 能够通过数据网格呈现数据源中任意数量的数据 |
| *highlight* | 一种语法高亮引擎，为各种编程语言的<CODE>块提供客户端语法高亮支持 |
| *image* | 对常见的图像操作提供支持，例如，拖放幻灯片、缩放、提取缩略图、亮盒显示（light box）等 |
| *io* | 支持XHR的多部分（multipart）功能和一个用于实现跨域XmlHttpRequests的XHR IFRAME代理 |
| *jsonPath* | 与Xpath类似，但用于查询JavaScript对象；可以用来非常方便地查询JSON数据结构 |
| *lang* | 为操作数据、散列提供更多支持，基于函数型编程（lambda）扩展而来 |
| *layout* | 更多的布局部件 |
| *math* | 一组高级数据函数，用于抽象曲线定义、点计算等 |
| *off* | 封装了Google Gears，为Web应用程序提供离线操作功能 |
| *presentation* | 为各种基于显示的任务（例如，幻灯片展示）提供支持 |
| *rpc* | 增强了dojo.rpc，用于执行远程过程调用 |
| *sketch* | 基于dojox.gfx模块的一个跨浏览器的绘图编辑器 |
| *storage* | 为数据持久化存储提供有限支持的一个JavaScript抽象，通过Flash或Google Gears等本地浏览器扩展实现 |
| *string* | 提供了各式各样的字符串实用函数 |
| *timing* | 支持高级时间控制技术 |
| *uuid* | 按照RFC4122的规定，对Universally Unique Identifier（通用唯一标识符）的实现 |

表 B-2：DojoX 包含的子项目（续）

| 子项目 | 简介 |
| --- | --- |
| *validate* | 支持常见的验证功能，例如，电子邮件地址、社会保障号码等等 |
| *widget* | 一组部件，与 Dijit 中的部件类似，包括高级颜色拾取器、鱼眼式列表、吐司机（toaster）、向导、放大镜、高级卷动面板等等 |
| *wire* | 通过提供通用数据绑定和服务调用库，在客户端中支持简化MVC模式的API |
| *xml* | 支持 XML 处理的实用程序 |