HZ BOOKS
华章科技

PEARSON

HTML5 Canvas领域的标杆之作，公认的权威经典，Amazon五星级超级畅销书，资深技术专家David Geary最新力作

全面讲解Canvas元素的各种功能特性，以及如何利用Canvas进行图形绘制、动画制作、游戏开发、自定义控件制作和移动应用开发，包含大量实例，可操作性极强

华章程序员书库

Core HTML5 Canvas
Graphics, Animation, and Game Development

HTML5 Canvas核心技术
图形、动画与游戏开发

（美）David Geary 著
爱飞翔 译

机械工业出版社
China Machine Press

Core HTML5 Canvas
Graphics, Animation, and Game Development

Canvas是HTML5技术标准中最令人振奋的功能之一。它提供了一套强大的2D图形API，让开发者能够制作从文字处理软件到电子游戏的各类应用程序。在本书中，畅销书作家David Geary先生以实用的范例程序直接切入这套API，全面讲解其功能，以求让读者实现出内容丰富且界面一致的网络应用程序，并将开发好的程序部署在多种设备及操作系统之上。

利用简洁而又清晰的文笔，本书展示了很多现实工作中的Canvas API用例，诸如互动式地绘制与修改图形，通过存储及恢复绘图表面来绘制临时性的图形与文本，以及实现文本输入控件等。读者将在本书中学到如何利用辅助线程制作出能够及时响应用户输入的图像滤镜程序，如何流畅地播放动画，以及如何利用视差技术画出具有3D效果的分层滚动背景图。此外，本书还详细讲解了制作电子游戏所用的精灵、物理学知识及碰撞检测技术，并且实现了一个游戏引擎及一款精美的弹珠台游戏。本书结尾部分将会告诉大家怎样实现基于Canvas的控件，使之运行在任意HTML5应用程序之中，以及如何在移动设备（包含利用iOS平台的移动设备）上运用Canvas技术。

这本权威的Canvas参考书涵盖了如下内容：

- **canvas元素：**将该元素与其他HTML元素结合使用，处理其中发生的事件，打印canvas，使用离屏canvas。
- **图形：**绘制、拖放、擦除及编辑线段、圆弧、圆形、曲线及多边形。
- **文本：**绘制、定位文本及设置字型属性，构建文本输入控件。
- **图像：**绘制、缩放图像，设置剪辑区域，制作图像滤镜及图像动画。
- **动画：**创建流畅、高效、可移植的动画。
- **精灵：**实现具备绘制器及行为策略的动画精灵对象。
- **物理效果：**针对自由落体、钟摆及抛射体等物理系统建模，实现渐变的非线性运动及动画。
- **碰撞检测：**清晰地讲解检测碰撞所用的各种先进算法。
- **游戏开发：**全面讲解游戏开发所需的知识，例如基于时间的运动与高分榜等，并且实现了一个游戏引擎。
- **自定义控件：**讲述了实现自定义控件所需的基础架构，实现了进度条、滑动条及图像查看器等自定义控件。
- **移动开发：**使Canvas应用程序适应移动设备的屏幕，运用媒体特征查询技术，处理触摸事件，以及掌握iOS5平台的特性（例如应用程序启动图标）。

纵观全书，Geary先生以高质量、可复用的代码向开发者讲解了所要学习的全部技术，没有任何闲言赘语。corehtml5canvas.com网站包含本书的全部代码以及其中某些关键技术的在线演示程序。

PEARSON

PEARSON
www.pearson.com

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

上架指导：计算机/程序设计/HTML5
ISBN 978-7-111-41634-0
9 787111 416340 >
定价：99.00元

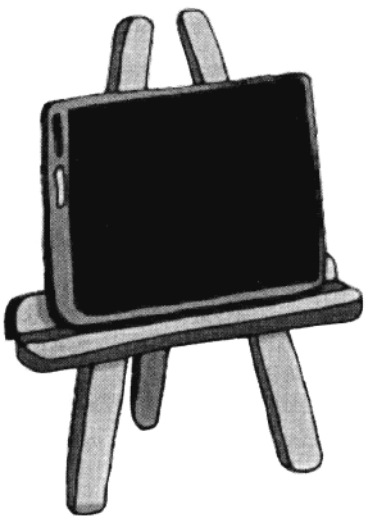
序员书库

Core HTML5 Canvas
Graphics, Animation, and Game Development

HTML5 Canvas核心技术

图形、动画与游戏开发

（美）David Geary 著
爱飞翔 译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

HTML5 Canvas核心技术：图形、动画与游戏开发 /（美）基瑞（Geary, D.）著；爱飞翔译．—北京：机械工业出版社，2013.2
（华章程序员书库）
书名原文：Core HTML5 Canvas: Graphics, Animation, and Game Development

ISBN 978-7-111-41634-0

I. H…　II. ① 基…　② 爱…　III. 超文本标记语言–游戏程序–程序设计　IV. TP312

中国版本图书馆CIP数据核字（2013）第037106号

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2013-0208

本书是 HTML5 Canvas 领域的标杆之作，也是迄今为止该领域内容最为全面和深入的著作之一，是公认的权威经典、Amazon 五星级超级畅销书、资深技术专家 David Geary 最新力作。它不仅全面讲解了 canvas 元素的 API，以及如何利用 Canvas 进行图形绘制、动画制作、物理效果模拟、碰撞检测、游戏开发、移动应用开发，还包含大量实例，可操作性极强。

全书共分 11 章。第 1 章介绍了 canvas 元素及如何在网络应用程序中使用它；第 2 章深入研究了如何使用 Canvas 的 API 进行绘制；第 3 章告诉读者如何绘制并操作 Canvas 中的文本；第 4 章专门讲解图像、图像的操作及视频处理；第 5 章介绍如何实现平滑的动画效果；第 6 章讲解如何用 JavaScript 语言来实现精灵；第 7 章展示了如何在动画中模拟物理效果；第 8 章介绍了进行碰撞检测所用的技术；第 9 章以一个简单但是高效的游戏引擎开始，提供了游戏制作所需的全部支持功能；第 10 章讨论了实现自定义控件的通用方法；第 11 章专门讲述如何实现基于 Canvas 的手机应用程序。

Authorized translation from the English language edition, entitled Core HTML5 Canvas: Graphics, Animation, and Game Development, 9780132761611 by David Geary, published by Pearson Education, Inc., Copyright © 2012 David Geary.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese Simplified language edition published by Pearson Education Asia LTD., and China Machine Press Copyright © 2013.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

机械工业出版社（北京市西城区百万庄大街22号　　邮政编码　100037）
责任编辑：关　敏
藁城市京瑞印刷有限公司印刷
2013年5月第1版第1次印刷
186mm×240mm · 31.75印张（含0.25印张彩插）
标准书号：ISBN 978-7-111-41634-0
定　　价：99.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

译 者 序

HTML5 是一个富有活力的前沿领域，虽说早在 2008 年年初就发布了第 1 份草案，然而该标准的流行则是 2010 年之后的事情了。

HTML5 技术的兴起有多方面的原因。其中比较重要的一点就是，1999 年制定的 HTML4.01 标准在十几年后已经无法满足急速增长的网络开发需求了。与传统的“服务器－客户端”架构相比，越来越多的开发者开始选择以网页的形式来制作应用软件与游戏。这样做能够降低维护成本，将原来更新客户端所花的精力投入到网页程序的完善之上，以便更加及时地满足新出现的客户需求。如此一来，怎样弥补网页程序在图形绘制、设备底层功能调用、文件访问、影音播放等方面的劣势，就成为制定新标准时必须考虑的问题了。HTML5 标准新增的各类 API 能够很好地应对这些状况。

此外，近年来日益兴起的移动开发也引发了人们对 HTML5 技术的关注。在传统的开发方式中，我们必须移植出版本繁多的客户端，以应对那些操作系统、屏幕大小、硬件配置各不相同的手机及平板电脑。如果我们将这种开发流程以 HTML5 应用的形式统一起来，那么就可以省去在各种设备之间进行移植所带来的问题。

不论是在传统的桌面操作系统之上，还是在新兴的移动设备之中，各大浏览器厂商都在努力适应新的 HTML5 标准，力求提供一套功能丰富而且外观统一的 API。尽管 HTML5 标准仍在不停地更新之中，但是其基本开发思路已经受到众多开发者及用户的肯定。所以说，在 HTML5 标准最终定型之前，提前学习新标准，及早推出开发成本适中而且产品内容丰富的各类 HTML5 网络应用软件与游戏，不仅可以提高自身的技术能力，还可以把握软件市场的走向，总体来看，是一项明智的抉择。

网络应用开发是一项外延很广的领域，在企业级开发中，我们要学习各种服务器通信技术、软件架构及开发框架，而在另外一些实用软件及游戏的开发中，则又需要投入大量时间进行网页前端的美化。从开发者学习知识的角度看，如果将 HTML、JavaScript、CSS 等制作网络应用程序所需的技术分开研究，则难以把握它们之间的联系，本书的出现正好解决了这个问题。选择 Canvas 为切入点是恰到好处的，因为在各类网络应用软件与游戏中，都要或多或少地用到与绘制相关的功能。如果能够将网页开发常用的 HTML、JavaScript、CSS 等技术有机地结合起来，那么就可以充分地发挥 Canvas 在绘图方面的优势，制作出效果可与原生应用程序相媲美的产品来。

研究 Canvas 的另一个好处是，我们在学习它的同时，还能掌握与之相关的各项实用技术，诸如图形与曲线的编辑、文本的绘制与输入、图像滤镜，等等。在学习这些技术的过程中，读者将会逐步掌握如何通过 JavaScript 代码来动态地运用 Canvas 所提供的各项功能。本书后半部分集中讲解了 Canvas 的几项重要用途，包括动画与精灵、游戏物理学、碰撞检测、游戏开发，等等。借助一个精美的弹珠台游戏，作者将这些重要技术详细而深入地讲解了一番，学习这部分内容将有助于提高图形开发及游戏制作的水平。

全书最后不仅演示了如何实现圆角矩形、滚动条、滑动条及图像查看器等自定义控件，而且还教给大家实现这些自定义控件所遵循的流程，使我们能够用相似的办法实现一大批符合自己需求的新控件来。此外，作者还详细介绍了如何让开发出来的应用程序能够更好地运行于各类移动设备之上。

本书作者清晰而又透彻的行文风格，不仅把图形绘制、动画制作、游戏开发、自定义控件、移动开发等几项重要知识讲解得十分精彩，而且还启迪了我们的开发思路。读完全书之后，大家更需要思考如何利用 JavaScript 程序，将类似 Canvas 这样的新兴技术与传统的 HTML、CSS 结合起来，开发出更加丰富灵活的网络应用程序来。

在本书的翻译过程中，我得到了机械工业出版社华章公司各位编辑及工作人员的帮助，在此表示由衷的感谢。

本书由爱飞翔翻译，王鹏、舒亚林及张军也参与了部分翻译工作。译者非常愿意与诸位朋友通过微博（weibo.com/eastarlee）或电子邮件（eastarstormlee@gmail.com）探讨各类 HTML5 技术问题。由于水平有限，错误和不当之处在所难免，敬请广大读者批评指正。

前　言

2001 年夏天，笔者阅读了一本有关网络应用程序开发的畅销书。此时，我已经从事了 15 年图形用户界面（Graphics User Interface，GUI）与“图形密集型应用程序”（graphic-intensive application）的开发。当时我并不知道，那本书的作者 Jason Hunter，会在 No Fluff Just Stuff（NFJS）㊀巡回演讲上成为我的好朋友。

看完了 Jason 所著的那本有关 Servlet 的书㊁之后，我将它放在膝上，凝望着窗外。做了这么多年的 Smalltalk、C++ 与 Java 语言的开发工作，并富有激情地写完了 1622 页的《Graphic Java 2: Swing》㊂一书后，我对自己说：是不是真的该像编写 HTML 网页那样，用类似打印语句那样的代码来实现用户界面了？是的，我的确这样做了。

从那时起，我就在自认为是“软件开发的黑暗时代㊃”之中艰难前行。我是 Apache Struts 项目㊄的第二个代码提交者，并且创造了 Struts 模板库（Struts Template Library），该库最终演进成了流行的 Tiles 项目㊅。我在“JavaServer Faces（JSF）㊆专家组（expert group）”中待过 6 年多，还在超过 120 场 NFJS 研讨会及许多其他会议中做过关于服务器端 Java 开发（server-side Java）的演讲，并且与人合著了一本关于 JSF 的书㊇。曾经有一段时间，Google Web Toolkit ㊈与 Ruby on Rails ㊉技术令我感到兴奋，不过到了这段苦日子的最后，整个工作基本都是枯燥的业务逻辑：在客户端向用户展示数据表单，并在服务器端对其进行处理，再也找不回当初做图形开发及图形用户界面时的那种激情了。

㊀ NFJS是一个有关Java与敏捷技术的软件研讨会，其网站是：http://www.nofluffjuststuff.com/。——译者注

㊁ 由Jason Hunter与William Crawford编写的《Java Servlet Programming》，O'Reilly出版社2001年出版。

㊂ 《Graphic Java 2: Mastering the JFC, Volume 2: Swing, 3rd Edition》，David Geary著，由Prentice Hall出版社于1999年出版。（本书中文版为《Java 2图形设计 卷2：SWING》，由机械工业出版社于2000年出版——译者注）

㊃ 欧洲黑暗时代（Dark Ages或Dark Age），指的是约476年至1000年的中世纪前期（Early Middle Ages）。详情参阅：http://zh.wikipedia.org/zh-cn/欧洲黑暗时代。——译者注

㊄ Struts是Apache软件基金会（ASF）赞助的一个开源项目。它通过采用Java Servlet / JSP技术，实现了基于Java EE Web应用的Model-View-Controller（MVC）设计模式的应用框架〔Web Framework〕，是MVC经典设计模式中的一个经典产品。详情参见：http://zh.wikipedia.org/zh-cn/Struts。——译者注

㊅ Apache Tiles，是以Java语言编写的HTML模板框架。其官网是：http://tiles.apache.org/。——译者注

㊆ JavaServer Faces (JSF) 是一种用于构建Java Web 应用程序的标准框架（是Java Community Process 规定的JSR-127标准）。它提供了一种以组件为中心的用户界面（UI）构建方法，从而简化了Java服务器端应用程序的开发。由于由Java Community Process (JCP) 推动，属于Java EE 5中的技术规范，而受到了厂商的广泛支持。详情参阅：http://zh.wikipedia.org/zh-cn/JavaServer_Faces。该项目专家组的网址为：http://www.javaserverfaces.org/specification/expert-group。——译者注

㊇ 《Core JavaServer™ Faces, Third Edition》，由David Geary与Cay Horstmann合著，Prentice Hall出版社于2010年出版。

㊈ Google Web Toolkit（GWT），是一个前端使用JavaScript，后端使用Java的AJAX 框架，它通过编译器将Java程序代码编译成JavaScript，可让开发人员使用Java语言，快速构建与维护复杂但高效能的JavaScript前端应用程序，借此减轻开发人员负担。其项目网址是：https://developers.google.com/web-toolkit。——译者注

㊉ Ruby on Rails，简称RoR或Rails，是使用Ruby语言编写的开源Web应用框架。它严格按照MVC结构开发，可减少应用开发时的编码及配置工作。详情参阅：http://zh.wikipedia.org/zh-cn/Ruby_on_Rails。——译者注

2010 年夏天，在 HTML5 技术以不可阻挡之势流行起来的时候，我偶然看到了一篇讲述 Canvas 元素的文章，那时我就知道，救赎的时刻即将来临。我立刻放下了职场中的一切事务，将所有时间都投入到写作之中，力求写出一本最好的 Canvas 书籍。从那时起，直到 2012 年 3 月本书完成，我完全沉浸在 Canvas 技术和这本书的写作之中。这是迄今为止带给我最多写作乐趣的一本书。

从文字处理到电子游戏，它们所需的全部图形功能，Canvas 元素都会提供给你。尽管它在各个平台中的性能有所差异，不过总的来说，Canvas 的运行速度还是很快的，尤其是在 iOS5 平台上，Mobile Safari 浏览器可以利用硬件加速来支持 Canvas 的渲染。浏览器厂商们在遵循 HTML5 规范方面做得都相当好，所以，编码良好的 Canvas 应用程序无需修改即可在任何兼容 HTML5 的浏览器中运行，偶有轻微的不兼容现象。

HTML5 就像是“软件开发黑暗时代”之后的“文艺复兴”，可以说，Canvas 元素是 HTML5 中最激动人心的技术。本书将深入讲解 Canvas 元素及相关的 HTML5 技术（例如动画时间控制规范），利用这些技术可以开发出能够跨桌面浏览器及移动平台运行的应用程序。

如何阅读本书

本书的写作方式，用禅意一些的话来说，就是可以使你“无需阅读即可领会其内容”（read it without reading）。

在撰写每一章（通常耗时数月）的过程中，我有时一个字都不写，只是一直反复地阅读写作材料。在那段时间中，我致力于准备写作提纲、程序清单、屏幕截图、表格、示意图、条目列表、注解、小窍门及注意事项。这些被我叫做“支架”（scaffolding）[⊖]的东西，才是本书中最为重要的部分。书中的正文，是我在搭好上述支架后才写的，它们是为了给周边的支架材料提供上下文说明，并阐明其中的重要之处。写好后，我又会反复阅读几遍，尽可能地删掉多余的文字。

因为本书专注于提供支架式的学习材料，且惜墨如金，所以这样一本书很容易让读者“无需阅读即可领会其内容”。你可以略读其中的文本，而专注于屏幕截图、程序清单、示意图、表格，以及其他支架材料，无论你想了解什么话题，这么做都能让你学到很多知识。你可以随意将书中的正文视为二等公民，而且，如果乐意的话，等需要查阅正文的时候再来读也不迟。

全书概览

本书由两部分组成。第一部分包括前 4 章，几乎占了本书篇幅的一半。这部分讲解了 Canvas 元素的 API，告诉读者如何在 Canvas 元素上面绘制图形与文本，如何绘制并操作图像。本书剩下的 7 章将告诉读者，如何实现动画效果及动画精灵，如何创建物理模拟效果，如何进行碰撞检测，以及如何开发电子游戏。在本书最后两章之中，有一章会讲如何实现一些自定义控件，例如进度条、滑动条（slider）、图像查看器（image panner），另外一章将会告诉你如何创建基于 Canvas 的手机应用程序。

⊖ 作者在这里使用了“支架式教学”（又名“鹰架理论”，Scaffolding Instruction或Instructional Scaffolding）的一些术语。该教学法在教授学生一项新的概念或技能时，通过提供足够的支援来提升学生的学习能力，当学生具备了自主的学习策略，提升了认知、情感、知识与学习技能时，这些支援会逐渐取走。详情参阅：http://www.hudong.com/wiki/支架式教学。——译者注

第 1 章“基础知识”，介绍了 canvas 元素，并且演示了如何在网络应用程序中使用它。本章有一个小节，介绍了进行 HTML5 开发所需的一般性入门知识，包括浏览器、控制台、调试器、性能分析器、时间轴等。接下来的内容为读者讲解如何实现 canvas 元素的一些基本功能：如何在 canvas 之上绘制、canvas 参数以及绘图表面自身的保存与恢复、打印 canvas，还有一段关于离屏 canvas（offscreen canvas）的简介。本章最后简单地介绍了一些基础的数学知识，包括基本的代数运算、三角函数、向量数学，以及根据计量单位推导等式。

第 2 章“绘制”，这是全书篇幅最长的一章，深入研究了如何使用 Canvas 的 API 进行绘制，告诉读者如何在 canvas 上绘制线条、弧线、曲线、圆、矩形以及任意多边形，如何以纯色、渐变色及图案对其进行填充。本章不仅讲述了绘图机制，还向读者展示了如何使用 Canvas API 来制作一些有实际用途的范例程序，例如：通过绘制临时的辅助矩形来创建橡皮带线条（rubber band）；拖动 canvas 之中的图形；实现一个简单的保留模式图形子系统，用以跟踪 canvas 之中的多边形，以便让用户对其进行编辑；使用剪辑区域，在不影响 Canvas 后方背景的情况下擦除图形。

第 3 章“文本”，告诉读者如何绘制并操作 canvas 之中的文本。你将会学到如何对文本进行描边与填充，如何设置字型属性，以及如何调整文本在 canvas 之中的位置。本章也讲了如何在 Canvas 中实现自制的文本控件。本章最后讲述了如何制作闪烁的文本编辑光标以及可编辑的文本段。

第 4 章“图像与视频”，专门讲解图像、图像的操作以及视频处理。你将会看到如何在 canvas 中绘制与缩放图像，你还将学到如何通过获取每个像素的颜色分量来编辑图像。在这一章中还可以学到更多有关剪辑区域的使用方法，以及如何将图像做成动画的知识。接下来的内容强调了安全与性能的重要性，最后以视频处理来结束本章内容。

第 5 章“动画”，告诉读者如何运用名为 requestAnimationFrame() 的方法来实现平滑的动画效果。该方法定义在 W3C 一个题为“基于脚本动画的定时控制”（Timing control for script-based animations）的规范中。读者还将看到计算动画帧速率的方法，以及如何进行其他活动的排期，例如以另外一种帧速率来刷新动画的用户界面。本章将会告诉读者在播放动画时用来恢复背景图像的三种策略，并且比较了每种方式所存在的性能隐忧。本章还演示了如何实现基于时间的运动效果，动画的背景滚动，使用视差来制作仿 3D 效果，以及在播放动画的过程中检测并响应用户手势。本章收尾部分讲了定时动画，并实现了一个简单的动画定时器，其后讨论了动画制作的最佳实践。

第 6 章“精灵”，向读者讲解了如何以 JavaScript 语言来实现精灵（即带有动画效果的显示对象）。精灵对象具有可视化的表现形式，这通常指的是一幅图像。你可以在 canvas 内移动它们，并且循环地播放某个图像集合中的图片，来以此产生动画效果。精灵是用以制作游戏的基础构建单元。

第 7 章“物理效果”，向读者展示了如何在动画中模拟物理效果，对其进行建模，包括下坠的物体、抛射体的弹道，以及摇晃的钟摆。本章还向读者展示了如何将时间与运动这两个要素封装入动画之中，以模拟真实世界的移动效果，诸如刚起跑的短跑运动员（加速运动）与正在刹闸的汽车（减速运动）。

在许多游戏中，碰撞检测是另一个重要的方面。所以，本书第 8 章“碰撞检测”专门讲解进行精灵之间碰撞检测所用的技术。本章开头讲述了使用外接矩形或外接圆形所进行的简单碰撞检测。这种办法实现起来很简单，然而并不是十分可靠。鉴于在很多情况下，简单的碰撞检测结果都不可靠，所以，本章的大部分内容都用于讲解分割轴定理（Separating Axis Theorem）。该定理

是在任意 2D 或 3D 多边形之间，进行碰撞检测所用的最佳手段之一。然而，它并不是说给那些有数学恐惧症的人（mathematically faint of heart）听的，所以，这一章要花大量的篇幅，并以外行人听得懂的词语来表述这个定理。

第 9 章“游戏开发”，以一个简单但是高效的游戏引擎开始。它提供了游戏制作所需的全部支持功能，包括绘制精灵、维护高分榜、实现基于时间的运动效果，以及播放多声道的音乐。接下来，本章详述了两个游戏。第一个是简单的“Hello World”类型的游戏，它演示了游戏引擎的用法，并且为后续的游戏制作提供了一个便利的出发点。它还向读者演示了如何实现大部分游戏所普遍具有的功能，例如资源管理、平视游戏状态显示[⊖]，以及用于显示高分榜的用户界面。第二个是优秀的弹珠台（pinball）游戏，它利用了本书前面讲到的许多素材，并且演示了如何在实际游戏中进行复杂的碰撞检测。

很多基于 Canvas 的应用程序都需要一些自定义控件，所以，第 10 章“自定义控件”将会教你如何来实现它们。本章讨论了实现自定义控件的通用方法，然后通过 4 个自定义控件来演示这些技术，这 4 个控件是：圆角矩形、进度条、滑动条，以及图像查看器。

本书的最后一章“移动平台开发”，专门讲述如何实现基于 Canvas 的手机应用程序。你将会学到如何控制视窗（viewport）的大小，使得应用程序能够在手机上正常地显示出来。同时还会看到如何利用 CSS3 媒体特征查询（media query）技术来使应用程序适应不同的屏幕大小及显示方向。

你还将了解到如何让基于 Canvas 的应用程序以全屏模式运行，并为其提供桌面图标与启动画面，让它看起来与 iOS5 平台的原生应用程序几乎一模一样。本章的末尾将实现一个虚拟键盘，它使得用户在 iOS5 应用程序中，不需要将输入焦点放在文本域（text field）控件之中即可输入文本。

预备知识

为了有效阅读本书，你必须能够比较熟练地使用 JavaScript、HTML 及 CSS 技术。举例来说，笔者假设你已经知道如何使用 JavaScript 语言的原型继承（prototypal inheritance）来实现一个对象，并且已经熟知一般网络应用程序的开发。

本书也用到了一些你也许很久以前学过，但是早就忘掉的数学知识，诸如基本的代数运算、三角函数、向量数学，以及根据计量单位推导等式。在第 1 章的末尾，你将会看到一个涵盖上述话题的基础知识简介。

本书源代码

本书所有代码的版权均归笔者所有，对这些代码的使用必须遵守随源码发行的许可协议。此许可协议是在 MIT 许可协议的基础上修改而成的，它允许你使用本书代码来做任何事情，包括用其制作可以贩售的软件产品。然而，你不可以使用本书代码来创建教材，包括图书、培训视频、

⊖ Heads-up Display，简称HUD。是目前普遍运用在航空器上的飞行辅助仪器。平视的意思是指飞行员不需要低头就能够看到所需的重要资讯。电子游戏领域借用该词来表示叠加在游戏主画面之上的游戏状态信息，例如主角的生命值、道具、分数、当前关卡等。详情参阅：http://zh.wikipedia.org/zh-cn/平视显示器与http://en.wikipedia.org/wiki/HUD_(video_gaming)。——译者注

简报（presentation），等等。详细情况请参考源码附带的许可协议内容。

在实现这些范例代码时，笔者明智地选择将程序清单中所含的注释量降到最小。为此，笔者尽量将代码本身写得易读一些，每个方法中平均包含大约 5 行代码，非常易于理解。

笔者还严格遵循了 Douglas Crockford 在其经典著作《JavaScript, The Good Parts》[⊖]中所提出的建议。比方说：所有函数作用域内部的变量都声明在函数头部；每个变量的声明都单独占一行代码；总是使用 === 操作符及其同类操作来进行相等性测试（equality testing）。

Canvas 规范与本书将来的发展情况

HTML5 的 API 在持续地进化，每一次进化，大都伴随着新功能的加入。Canvas 标准当然也不例外，实际上，就在这本书将要送去印刷的前几天，WHATWG [⊖]的 Canvas 标准就进行了一次更新。此次更新又向其中加入了如下新功能：

- 用于创建椭圆形路径的 ellipse() 方法。
- 两个名为 getLineDash()、setLineDash() 的新方法，以及用于绘制虚线的 lineDashOffset 属性。
- 进行了功能扩充之后的 TextMetrics 对象，它可以让开发者精确地指定文本的外接矩形框。
- 名为 Path 的路径对象。
- 名为 CanvasDrawingStyles 的绘制风格对象。
- 对点击区域（hit region）的广泛支持。

在那时，尚未有任何浏览器支持这些新加入的功能，所以当时不可能编写代码来测试它们。

在 2012 年 3 月 26 日的这次更新之前，你已经可以使用 Canvas 来绘制弧线及圆形了，但是规范之中尚未有条款规定 Canvas 必须支持椭圆的绘制。这次更新之后，除了可以绘制弧线与圆形之外，你也可以使用新加入的 ellipse() 方法在 2D 环境中的 Canvas 上面来绘制椭圆了。与此类似，该绘制环境现在也明确支持虚线的绘制了。

原来的 TextMetrics 对象只能返回一种度量结果，那就是字符串的宽度。然而，经过了 2012 年 3 月 26 日的规范更新之后，你可以设定 canvas 中某个字符串所占据矩形区域的宽度与高度了。对 TextMetrics 对象的这次功能扩充，使得我们可以更加简单、高效地实现基于 Canvas 的图形控件。

除了椭圆绘制功能的加入与 TextMetrics 对象的改进之外，更新之后的规范还加入了一些归纳在 Path 与 CanvasDrawingStyles 名下的新方法。在规范更新之前，并没有明确规定用于存储路径与绘制风格的机制。现在，不仅有了专门用于表示这些抽象机制的对象，而且很多 2D 环境下的 Canvas 方法也多出来了一个可以接受 Path 对象参数的版本。例如，你可以调用 context.stroke() 方法来在当前绘制环境中进行路径描边，这时所描绘的路径是当前绘制环境中所设定的路径。然而，现在绘制环境对象也可以使用 stroke(Path) 方法来描绘路径了，该方法所描绘的路径是经由参数所传入的那个路径对象，而非当前环境中的路径。当你在 Path 对象上调用 addText() 这样的方法来修改路径时，你可以指定一个 CanvasDrawingStyle 对象，这样的话，路径对象在添加文本时也会使用这个对象中的属性。

⊖ 《JavaScript, The Good Parts》，Douglas Crawford著，O'Reilly出版社2008年出版。

⊖ Web Hypertext Application Technology Working Group，简称WHATWG，是一个以推动网络 HTML 5 标准为目的而成立的组织。在2004年，由Opera、Mozilla基金会和苹果这些浏览器厂商组成。其官方网站是：http://www.whatwg.org/。——译者注

更新之后的规范包含了对点击区域的广泛支持。点击区域是由路径所定义的。你可以在其上关联一个可选的鼠标光标，以及一些无障碍访问（accessibility）参数，诸如一个“无障碍丰富互联网程序”[⊖]，一个搭配有点击区域的标签，等等。一个 canvas 可以有多个点击区域。除了这些好处之外，使用点击区域还可以实现碰撞检测并提升无障碍访问性，使它们变得更为容易且更加高效。

最后，WHATWG 与 W3C[⊜]规范都包含了两个用于无障碍访问的 Canvas 环境方法，这样的话，应用程序就可以在当前路径周围绘制聚焦环（focus ring），以便让用户可以通过键盘在 Canvas 之中进行焦点切换。这项功能并不是 2012 年 3 月 26 日更新规范时添加的，实际上，它被加入规范当中已经有一段时间了。然而，在本书成稿时，尚未有浏览器厂商支持这项功能，所以本书中不会讲到这部分内容。

因为 Canvas 规范在不断地演进，同时浏览器厂商也在不断地实现新的功能，所以本书也会经常地更新。在此期间，可以访问 http://corehtml5canvas.com/ 来查阅这些 Canvas 新功能，并预览本书在下一个版本中对这些新功能的覆盖程度。

配套网站

本书的配套网站是：http://corehtml5canvas.com/。在这里可以下载书中的源代码，运行精选的本书范例程序，并查找其他与 HTML5 及 Canvas 技术有关的资源。

致谢

写书是一项团队运动，我很荣幸能够与诸位优秀的同伴一起完成本书的写作。

首先，我要感谢长期共事的编辑 Greg Doench 先生。他也是我的好朋友。从笔者提出本书写作计划的那一刻起，他就由衷地相信此书一定会是本佳作。同时，他也给了我相当的自由度，让我可以按照自己的想法来写作本书。从概念酝酿到付诸刊印，再到印刷成书，Greg 一直在监督着本书的制作。他做了这么多，我已经很满足啦！

还让笔者感到幸运的是，在制作本书的过程中，Greg 还带来了他自己的优秀团队。Julie Nahil 非常出色地完成了与生产管理有关的工作，她让一切事情都走上了正轨。Alina Kirsanova 将 DocBook 格式的原始 XML 文件转换成读者手中印刷精美的图书。Alina 女士也出色地完成了本书的校对工作，删除了书中的一些小错误及矛盾之处。

笔者又一次高兴地看到 Mary Lou Nohr 女士对本书进行了审稿工作。在 15 年的写书生涯中，Mary Lou 女士一直是审稿编辑，她不仅将每一本书都修改得比我的原文更好，而且还教会了笔者

⊖ Accessible Rich Internet Applications，全名为Web Accessibility Initiative - Accessible Rich Internet Applications，简称WAI-ARIA，是一份由W3C所提出的有关网络可及性的倡议，它针对无障碍网页服务提出了一些规范建议。详情参阅：http://en.wikipedia.org/wiki/WAI-ARIA。该规范的官方网址是：http://www.w3.org/TR/wai-aria/。——译者注

⊜ 万维网联盟（World Wide Web Consortium），又称W3C理事会。1994年10月在麻省理工学院计算机科学实验室成立。建立者是互联网的发明者蒂姆•伯纳斯-李。W3C制定了一系列标准并督促Web应用开发者和内容提供者遵循这些标准。标准内容包括使用语言的规范，开发中使用的导则和解释引擎的行为等。W3C也制定了包括XML和CSS等在内的众多影响深远的标准规范。官方网址是：http://www.w3.org/。——译者注

写作的技巧。

对于任何一本技术类书籍而言，要想获得成功，技术评审都是至关重要的。所以我会主动邀请一些我认为拥有合适的水准、能为本书做出重大贡献的人来当技术评审。我运气真好，找到了一些优秀的技术评审人员，他们帮助我对本书的内容打磨抛光，使之更加精致。首先，要感谢 Philip Taylor 先生，他是我所见过的最为博学且一丝不苟的技术评审之一。Philip 实现了将近 800 个与 Canvas 有关的测试用例，大家可以看看这里：http://philip.html5.org/tests/canvas/suite/tests。在评审本书的每一章时，他都发给我数页极富见解的评论，这些评论只有在对 Canvas 元素了解得细致入微的前提下才能写得出来。他对本书的贡献远远超出了技术评审的范围，仅他一个人的努力，就得以让本书的质量大为提升。

接下来，我要感谢 thirstyhead.com 公司的 Scott Davis 先生，他是位一流的 HTML5 及手机网络应用程序开发专家。Scott 做了很多场有关 HTML5 与移动开发技术的会议演讲，与他人联合创立了 HTML5 Denver User Group，并曾向 Yahoo! 公司的开发者教授过手机开发技术。与 Philip 一样，Scott 对书中许多不同的主题都提出了优秀的建议，这也远远超出了技术评审的要求。由于他严格的评审意见，笔者将本书近四分之一的内容全部改写了一遍，这也让本书的发行时间推迟了整整三个月。重写之后，本书的品质又提升了一个档次，为此，我要由衷地感谢 Scott。

知名游戏 Runfield（http://fhtr.org/runfield/runfield）的作者之一 Ilmari Heikkinen 先生，对本书中与动画、精灵、物理学和检测碰撞有关的章节，提出了一些独到的建议。Ted Neward、Dion Almaer、Ben Galbraith、Pratik Pratel、Doris Chen、Nate Schutta 与 Brian Sam-Bodden 都给出了一些很好的评审意见。

我还要感谢的是 jsperf.com 网站的创立者 Mathias Bynens 先生，他允许笔者使用从那个网站上面截取的图片。

在“物理效果”一章中所用到的那份精灵表（sprite sheet），是由 MJKRZAK 绘制的，笔者感谢这位网友。该精灵表是我从 People's Sprites 网站[⊖]的公共领域区下载的。笔者还要感谢 Ilmari Heikkinen 先生，他允许我在“动画”这一章中使用他所绘制的天空图案作为讲解视差范例时的素材。“精灵”这一章的某些图像，是取材于知名的开源游戏 Replica Island [⊜]。

最后，感谢 Hiroko、Gaspé 与 Tonka 在这段时间中对我的理解，我将这一年半的生活全部都用在了本书的写作之上。

作者简介

David Geary 先生是一位杰出的作家、演讲者与顾问，他从 20 世纪 80 年代就开始用 C 语言及 Smalltalk 语言来实现基于图形的应用程序与用户接口了。David 在波音公司从事了 8 年 C++ 语言与面向对象软件开发的培训工作，在 1994 年至 1997 年间，曾在 Sun Microsystems 公司担任软件工程师。他写了 8 本有关 Java 的书籍，其中有两本讲 Java 组件框架、Swing 与 JavaServer Faces（JSF）技术的书非常畅销。他所著的《Graphic Java 2: Swing》一直都是畅销的 Swing 教程，与 Cay Horstmann 合写的《Core JavaServer ™ Faces》，是 JSF 领域的畅销书。

David 是一位富有激情的演讲者，他在全世界数百场会议中都发表过演讲。他参加了 6 年的

⊖ 该网站网址是：http://panelmonkey.org/。——译者注

⊜ 该游戏的网站是：http://replicaisland.net/。——译者注

No Fluff Just Stuff 巡回演讲，在超过 120 场讨论会中做了发言，并且三度荣获“ JavaOne Rock Star”头衔[⊖]。

2011 年，David 与 Scott Davis 联合创立了 HTML5 Denver Meetup group（网址是 http://www.meetup.com/HTML5-Denver-Users-Group），截至 2012 年本书出版时，该讨论组已经有超过 500 名成员了。

读者可以通过 Twitter（用户名是 davidgeary）及本书的配套网站（网址是 http://corehtml5-canvas.com/）联系到 David。

⊖ JavaOne是一个主要由Java开发人员所参与的周年研讨会。JavaOne Rock Star是向那些发表了精彩演讲并且展示了演说能力的开发者授予的荣誉称号。详情参阅：http://www.oracle.com/javaone/quick-links/rock-star/index.html。——译者注

目　录

译者序

前言

第 1 章　基础知识 …… 1

1.1　canvas 元素 …… 1

1.1.1　canvas 元素的大小与绘图表面的大小 …… 4

1.1.2　canvas 元素的 API …… 5

1.2　Canvas 的绘图环境 …… 6

1.2.1　2d 绘图环境 …… 6

1.2.2　Canvas 状态的保存与恢复 …… 8

1.3　本书程序清单的规范格式 …… 9

1.4　开始学习 HTML5 …… 10

1.4.1　规范 …… 10

1.4.2　浏览器 …… 11

1.4.3　控制台与调试器 …… 11

1.4.4　性能 …… 13

1.5　基本的绘制操作 …… 15

1.6　事件处理 …… 18

1.6.1　鼠标事件 …… 18

1.6.2　键盘事件 …… 22

1.6.3　触摸事件 …… 23

1.7　绘制表面的保存与恢复 …… 23

1.8　在 Canvas 中使用 HTML 元素 …… 25

1.9　打印 Canvas 的内容 …… 32

1.10　离屏 canvas …… 35

1.11　基础数学知识简介 …… 37

1.11.1　求解代数方程 …… 37

1.11.2　三角函数 …… 38

1.11.3　向量运算 …… 39

1.11.4　根据计量单位来推导等式 …… 42

1.12　总结 …… 44

第 2 章　绘制 ……45
2.1　坐标系统 ……46
2.2　Canvas 的绘制模型 ……47
2.3　矩形的绘制 ……48
2.4　颜色与透明度 ……50
2.5　渐变色与图案 ……52
2.5.1　渐变色……52
2.5.2　图案……54
2.6　阴影 ……57
2.7　路径、描边与填充 ……60
2.7.1　路径与子路径……63
2.7.2　剪纸效果……64
2.8　线段 ……69
2.8.1　线段与像素边界……70
2.8.2　网格的绘制……71
2.8.3　坐标轴的绘制……72
2.8.4　橡皮筋式的线条绘制……74
2.8.5　虚线的绘制……79
2.8.6　通过扩展 CanvasRenderingContext2D 来绘制虚线 ……80
2.8.7　线段端点与连接点的绘制……81
2.9　圆弧与圆形的绘制 ……83
2.9.1　arc() 方法的用法 ……83
2.9.2　以橡皮筋式辅助线来协助用户画圆……85
2.9.3　arcTo() 方法的用法 ……86
2.9.4　刻度仪表盘的绘制……88
2.10　贝塞尔曲线……93
2.10.1　二次方贝塞尔曲线 ……93
2.10.2　三次方贝塞尔曲线 ……95
2.11　多边形的绘制……97
2.12　高级路径操作……102
2.12.1　拖动多边形对象 ……102
2.12.2　编辑贝塞尔曲线 ……107
2.12.3　自动滚动网页，使某段路径所对应的元素显示在视窗中 ……115
2.13　坐标变换……116
2.13.1　坐标系的平移、缩放与旋转 ……116
2.13.2　自定义的坐标变换 ……119
2.14　图像合成……123
2.15　剪辑区域……128
2.15.1　通过剪辑区域来擦除图像 ……128

2.15.2 利用剪辑区域来制作伸缩式动画 …… 133
2.16 总结 …… 135

第 3 章 文本 …… 137
3.1 文本的描边与填充 …… 137
3.2 设置字型属性 …… 141
3.3 文本的定位 …… 144
3.3.1 水平与垂直定位 …… 144
3.3.2 将文本居中 …… 146
3.3.3 文本的度量 …… 147
3.3.4 绘制坐标轴旁边的文本标签 …… 148
3.3.5 绘制数值仪表盘周围的文本标签 …… 151
3.3.6 在圆弧周围绘制文本 …… 152
3.4 实现文本编辑控件 …… 154
3.4.1 指示文本输入位置的光标 …… 154
3.4.2 在 Canvas 中编辑文本 …… 159
3.4.3 文本段的编辑 …… 163
3.5 总结 …… 174

第 4 章 图像与视频 …… 175
4.1 图像的绘制 …… 176
4.1.1 在 Canvas 之中绘制图像 …… 176
4.1.2 drawImage() 方法的用法 …… 177
4.2 图像的缩放 …… 179
4.3 将一个 Canvas 绘制到另一个 Canvas 之中 …… 183
4.4 离屏 canvas …… 186
4.5 操作图像的像素 …… 189
4.5.1 获取图像数据 …… 189
4.5.2 修改图像数据 …… 195
4.6 结合剪辑区域来绘制图像 …… 208
4.7 以图像制作动画 …… 211
4.8 图像绘制的安全问题 …… 216
4.9 性能 …… 216
4.9.1 对比 drawImage(HTMLImage)、drawImage(HTMLCanvas) 与 putImageData() 的绘图效率 …… 217
4.9.2 在 Canvas 中绘制另一个 Canvas 与绘制普通图像之间的对比；在绘制时缩放图像与保持原样之间的对比 …… 217
4.9.3 遍历图像数据 …… 218
4.10 放大镜 …… 222

4.10.1 使用离屏 canvas …… 224
4.10.2 接受用户从文件系统中拖放进来的图像 …… 225
4.11 视频处理 …… 227
4.11.1 视频格式 …… 227
4.11.2 在 Canvas 中播放视频 …… 229
4.11.3 视频处理 …… 230
4.12 总结 …… 234

第 5 章 动画 …… 235
5.1 动画循环 …… 235
5.1.1 通过 requestAnimationFrame() 方法让浏览器来自行决定帧速率 …… 237
5.1.2 Internet Explorer 浏览器对 requestAnimationFrame() 功能的实现 …… 241
5.1.3 可移植于各浏览器平台的动画循环逻辑 …… 241
5.2 帧速率的计算 …… 248
5.3 以不同的帧速率来执行各种任务 …… 249
5.4 恢复动画背景 …… 250
5.4.1 利用剪辑区域来处理动画背景 …… 250
5.4.2 利用图块复制技术来处理动画背景 …… 252
5.5 利用双缓冲技术绘制动画 …… 253
5.6 基于时间的运动 …… 254
5.7 背景的滚动 …… 257
5.8 视差动画 …… 261
5.9 用户手势 …… 264
5.10 定时动画 …… 266
5.10.1 秒表 …… 266
5.10.2 动画计时器 …… 269
5.11 动画制作的最佳指导原则 …… 270
5.12 总结 …… 271

第 6 章 精灵 …… 272
6.1 精灵概述 …… 273
6.2 精灵绘制器 …… 275
6.2.1 描边与填充绘制器 …… 275
6.2.2 图像绘制器 …… 279
6.2.3 精灵表绘制器 …… 281
6.3 精灵对象的行为 …… 284
6.3.1 将多个行为组合起来 …… 285
6.3.2 限时触发的行为 …… 287
6.4 精灵动画制作器 …… 289

6.5 基于精灵的动画循环 …… 293
6.6 总结 …… 294

第 7 章 物理效果 …… 295
7.1 重力 …… 295
7.1.1 物体的下落 …… 296
7.1.2 抛射体弹道运动 …… 298
7.1.3 钟摆运动 …… 307
7.2 时间轴扭曲 …… 311
7.3 时间轴扭曲函数 …… 315
7.4 时间轴扭曲运动 …… 317
7.4.1 没有加速度的线性运动 …… 319
7.4.2 逐渐加速的缓入运动 …… 320
7.4.3 逐渐减速的缓出运动 …… 322
7.4.4 缓入缓出运动 …… 323
7.4.5 弹簧运动与弹跳运动 …… 324
7.5 以扭曲后的帧速率播放动画 …… 326
7.6 总结 …… 332

第 8 章 碰撞检测 …… 333
8.1 外接图形判别法 …… 333
8.1.1 外接矩形判别法 …… 333
8.1.2 外接圆判别法 …… 334
8.2 碰到墙壁即被弹回的小球 …… 336
8.3 光线投射法 …… 337
8.4 分离轴定理（SAT）与最小平移向量（MTV） …… 340
8.4.1 使用分割轴定理检测碰撞 …… 340
8.4.2 根据最小平移向量应对碰撞 …… 362
8.5 总结 …… 373

第 9 章 游戏开发 …… 374
9.1 游戏引擎 …… 374
9.1.1 游戏循环 …… 376
9.1.2 加载图像 …… 382
9.1.3 同时播放多个声音 …… 384
9.1.4 键盘事件 …… 385
9.1.5 高分榜 …… 386
9.1.6 游戏引擎源代码 …… 387
9.2 游戏原型 …… 395

9.2.1 游戏原型程序的 HTML 代码 …… 396
9.2.2 原型程序的游戏循环 …… 399
9.2.3 游戏原型程序的加载画面 …… 400
9.2.4 暂停画面 …… 402
9.2.5 按键监听器 …… 404
9.2.6 游戏结束及高分榜 …… 404
9.3 弹珠台游戏 …… 407
9.3.1 游戏循环弹珠 …… 408
9.3.2 弹珠精灵 …… 410
9.3.3 重力与摩擦力 …… 411
9.3.4 弹板的移动 …… 412
9.3.5 处理键盘事件 …… 413
9.3.6 碰撞检测 …… 416
9.4 总结 …… 425

第 10 章 自定义控件 …… 426
10.1 圆角矩形控件 …… 427
10.2 进度条控件 …… 433
10.3 滑动条控件 …… 437
10.4 图像查看器控件 …… 446
10.5 总结 …… 454

第 11 章 移动平台开发 …… 455
11.1 移动设备的视窗 …… 456
11.2 媒体特征查询技术 …… 461
11.2.1 媒体特征查询与 CSS …… 461
11.2.2 用 JavaScript 程序应对媒体特征的变化 …… 462
11.3 触摸事件 …… 464
11.3.1 TouchEvent 对象 …… 464
11.3.2 TouchList 对象 …… 465
11.3.3 Touch 对象 …… 466
11.3.4 同时支持触摸事件与鼠标事件 …… 466
11.3.5 手指缩放 …… 468
11.4 iOS5 …… 469
11.4.1 应用程序图标及启动画面 …… 469
11.4.2 利用媒体特征查询技术设置 iOS5 系统的应用程序图标及启动画面 …… 470
11.4.3 以不带浏览器饰件的全屏模式运行应用程序 …… 471
11.4.4 应用程序的状态栏 …… 471
11.5 虚拟键盘 …… 472
11.6 总结 …… 485

第1章
基础知识

1939 年，米高梅公司（Metro-Goldwyn-Mayer Studios）曾经发行了一部电影，根据美国国会图书馆（American Library of Congress）的说法，它注定会成为历史上观看人数最多的电影。《绿野仙踪》这部电影讲述了一个名叫 Dorothy 的年轻女孩与宠物狗 Toto 的故事。他们被一阵猛烈的龙卷风从美国中部的堪萨斯州刮到了名为“奥兹王国”（Oz）的一片梦幻国度之中。

这部电影一开始，是乏味与沉闷的黑白场景，等到 Dorothy 与 Toto 到了奥兹王国，电影一下子就切入了色彩鲜明的画面之中，整个冒险也就随之开始了……

在过去二十多年的时间里，软件开发者一直都在编写乏味、沉闷的网络应用程序，这些程序不外乎将一张张老套的表格永无止境般地展示给用户，令人烦躁不堪。终于，我们来到了 HTML5 的国度，它让开发者可以制作出激动人心的应用程序来。这些程序运行在浏览器之中，但是看上去与传统的桌面应用程序很相似。

在这个 HTML5 的奥兹王国中，我们将使用具有魔力的 canvas 元素[⊖]，来在浏览器中做一番奇妙的事情。 我们将会实现一个如图 1-1 所示的图像查看器，一个交互式的放大镜，一个可以在各种得体的浏览器以及 iPad 之中运行的绘图程序。我们还要做很多动画，包含一个优秀的弹珠台游戏在内的许多游戏，一些图像效果滤镜，以及很多其他的网络应用程序。在以前，上述软件完全处于 Flash 开发的领域之中。

咱们开始吧！

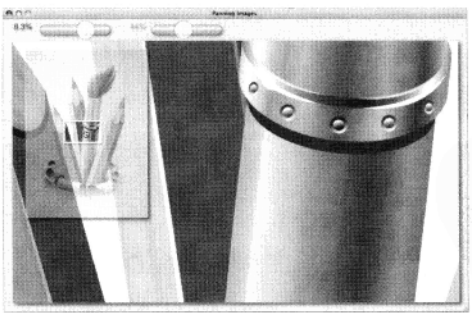


图 1-1　Canvas 提供了功能强大的图形处理 API

1.1　canvas 元素

canvas 元素可以说是 HTML5 元素中功能最强大的一个。然而，你马上就会看到，它真正的

⊖ canvas一词的中文意思是“画布”，在开发中，一般都直呼英文单词，称为“canvas元素”。本书以后均保留Canvas、canvas的英文形式，不做翻译。——译者注

能力是通过 Canvas 的 context 对象[⊖]而表现出来的。该环境变量可以从 canvas 元素身上获取。图 1-2 是一个简单的例子，它使用了 canvas 元素及与之相关的绘图环境。

图 1-2 Hello Canvas 范例程序

图 1-2 中的应用程序非常简单，它将一个字符串显示出来，该字符串大致与 canvas 自身的中心点对齐。该应用程序的 HTML 代码如程序清单 1-1 所示。

程序清单 1-1 中使用了 canvas 元素，为其指定了一个标识符，并设置了该元素的宽度与高度。请注意 canvas 元素内容部分（body）所含的文本，这种文本叫做“后备内容”（fallback content），浏览器仅在不支持 canvas 元素的时候，才会显示该内容。

程序清单 1-1 example.html

```
<!DOCTYPE html>
<html>
   <head>
     <title>A Simple Canvas Example</title>

      <style>
         body {
            background: #dddddd;
         }
         #canvas {
            margin: 10px;
            padding: 10px;
            background: #ffffff;
            border: thin inset #aaaaaa;
         }
      </style>
   </head>

  <body>
    <canvas id='canvas' width='600' height='300'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

除了这个元素之外，程序清单 1-1 中的 HTML 代码，还使用了 CSS 来设置应用程序的背景色以及 canvas 元素自身的某些属性。在默认状况下，canvas 元素的背景色与其父元素的背景色一

⊖ context一词，在HTML5的Canvas开发中，可以被叫做“绘图环境”、“绘图上下文”或“绘图句柄”，以下译文中一律称其为“绘图环境”。——译者注

致。所以，我们用 CSS 来将 canvas 元素的背景色设置成不透明的白色，这样就可以将它同淡灰色背景的应用程序区分开来。

HTML 代码很直白，没有什么有趣的地方。与大多数基于 Canvas 的应用程序一样，其中真正有趣的地方在于 JavaScript 代码。图 1-2 中所示应用程序的 JavaScript 代码，列在了程序清单 1-2 之中。

程序清单 1-2　example.js

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

context.font = '38pt Arial';
context.fillStyle = 'cornflowerblue';
context.strokeStyle = 'blue';

context.fillText('Hello Canvas',canvas.width/2 - 150,
                                canvas.height/2 + 15);

context.strokeText('Hello Canvas',canvas.width/2 - 150,
                                  canvas.height/2 + 15);
```

程序清单 1-2 中的 JavaScript 代码采用了一个诀窍，在开发基于 Canvas 的应用程序时也可以这样做：

（1）使用 document.getElementById() 方法[⊖]来获取指向 canvas 的引用。

（2）在 canvas 对象上调用 getContext('2d') 方法，获取绘图环境变量（注意，“2d”中的“d”必须小写）。

（3）使用绘图环境对象在 canvas 元素上进行绘制。

在获取了 canvas 的绘图环境对象之后，这段 JavaScript 代码设置了环境对象的 font、fillStyle、strokeStyle 等属性，然后对文本进行了填充及描边操作，于是就产生了图 1-2 中所示效果。fillText() 方法使用 fillStyle 属性来填充文本中的字符，strokeText() 方法则使用 strokeStyle 属性来描绘字符的轮廓线。fillStyle 与 strokeStyle 属性可以是 CSS 格式的颜色、渐变色或是图案。在 1.2.1 小节之中，我们会简单地讨论一下这些属性，然后在第 2 章中会深入地讲解以上的属性与方法。

fillText() 与 strokeText() 都需要 3 个参数：要绘制的文本内容，以及在 canvas 中显示文本的横、纵坐标。程序清单 1-2 中的 JavaScript 代码，使用了常量来将文本近似地对齐在 canvas 的中心点。这么做可以在 canvas 中实现文本居中，但却不是一个良好的通用方案。在第 3 章中，我们将会看到有一种更好的办法可以用于居中文本。

警告：在设置 canvas 的宽度与高度时，不能使用 px 后缀

虽说支持 Canvas 的浏览器普遍都允许在设定 canvas 元素的 width 与 height 属性时使用 px 后缀，但是，这么做从技术上来说是不被 Canvas 规范所接受的。根据规范书，这些属性的取值，只能是非负整数。

⊖ 在JavaScript语言中，“方法”与“函数”没有本质区别。具体来说，通常将明确从属于某个对象的代码例程称为“方法”，如上文的document.getElementById()，而将看上去似乎不从属于任何对象的代码例程称为“函数”，如alert()。其实这些“函数”也是默认从属于window对象的“方法”。本书在不致混淆的情况下，一律将“method”对译为“方法”，而将“function”对译为“函数”。更多信息请参考：https://en.wikipedia.org/wiki/JavaScript#Prototype-based与http://javascript.about.com/od/learnmodernjavascript/a/tutorial08.htm。——译者注

提示：默认的 canvas 元素大小是 300×150 个屏幕像素

在默认情况下，浏览器所创建的 canvas 元素是 300 像素宽、150 像素高。可以通过指定 width 与 height 属性值而修改 canvas 元素的大小。

还可以通过 CSS 属性来改变 canvas 元素的大小，不过，在下一小节中将会讲到，通过那种方式来修改 canvas 元素的宽度与高度，可能会产生意外的后果。

1.1.1 canvas 元素的大小与绘图表面的大小

前一小节中的那段应用程序代码是通过设置 canvas 元素的 width 与 height 属性，来改变元素大小的。也可以像程序清单 1-3 那样，通过 CSS 来设置 canvas 元素的大小。不过，使用 CSS 设置 canvas 元素的效果，与通过 width、height 属性值来设定，并不一样。

程序清单 1-3 使用不同的值来设置元素大小与绘图表面大小

```
<!DOCTYPE html>
   <head>
     <title>Canvas element size: 600 x 300,
            Canvas drawing surface size: 300 x 150</title>
      <style>
         body {
            background: #dddddd;
         }
         #canvas {
            margin: 20px;
            padding: 20px;
            background: #ffffff;
            border: thin inset #aaaaaa;
            width: 600px;
            height: 300px;
         }
      </style>
   </head>

  <body>
    <canvas id='canvas'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

使用 CSS 来设置 canvas 元素的大小，与直接设置属性相比，其差别是基于这样一个事实的：canvas 元素实际上有两套尺寸。一个是元素本身的大小，还有一个是元素绘图表面（drawing surface）的大小。

当设置元素的 width 与 height 属性时，实际上是同时修改了该元素本身的大小与元素绘图表面的大小。然而，如果是通过 CSS 来设定 canvas 元素的大小，那么只会改变元素本身的大小，而不会影响到绘图表面。在默认情况下，canvas 元素与其绘图表面，都是 300 像素宽、150 像素高。程序清单 1-3 的代码，使用 CSS 来将 canvas 元素的大小设置成 600 像素宽、300 像素高，然

而，绘图表面的大小依然没有改变，还是默认的 300×150 像素。

这时，有趣的事情就发生了。当 canvas 元素的大小不符合其绘图表面的大小时，浏览器就会对绘图表面进行缩放，使其符合元素的大小。图 1-3 演示了这种效果。

图 1-3　顶部的图中，元素与坐标系统均为 600×300 像素，底部的图中，元素是 600×300 像素，坐标系统是 300×150 像素

图 1-3 顶部所显示的应用程序就是我们在上一节中讨论的那个。它通过修改 width 与 height 属性来设置 canvas 元素的大小，这样的话，元素自身与绘图表面的大小都会被设置成 600×300 像素。

图 1-3 下方的那个应用程序是范例代码 1.3 中 HTML 代码的运行效果。这个程序与上一节中所讲的那个程序很相似，不过它是通过 CSS 来修改 canvas 元素大小的（此外，该程序运行窗口的标题栏也和上一个程序不同）。

由于图 1-3 下方的这个程序是通过 CSS 来修改 canvas 元素的大小，而不是通过 width 与 height 属性，所以，浏览器会将绘图表面从 300×150 像素拉伸到 600×300 像素。

警告：浏览器可能会自动缩放 Canvas

通过 width 与 height 属性而非 CSS 来修改 canvas 元素的大小，这是个好办法。如果使用 CSS 来修改元素的大小，同时又没有指定 canvas 元素的 width 与 height 属性，那么，当元素大小与 canvas 的绘图表面大小不相符时，浏览器会缩放后者，使之符合前者的大小。这样的话，很可能会导致奇怪的、无用的效果。

1.1.2　canvas 元素的 API

canvas 元素并未提供很多 API，实际上，它只提供了两个属性与三个方法，分别概括于表 1-1 与表 1-2 之中。

表 1-1 canvas 元素的属性

属性	描述	类型	取值范围	默认值
width	canvas 元素绘图表面的宽度。在默认状况下，浏览器会将 canvas 元素的大小设定成与绘图表面大小一致。然而，如果在 CSS 中覆写了元素大小，那么浏览器则会将绘图表面进行缩放，使之符合元素尺寸	非负整数	在有效范围内的任意非负整数。数值开头可以添加“+”与空格，但是按照规则，不能给数值加 px 后缀	300
height	canvas 元素绘图表面的高度。浏览器可能会将绘图表面缩放至与元素相同的尺寸。具体请参照对 width 属性的描述	非负整数	在有效范围内的任意非负整数。数值开头可以添加“+”与空格，但是按照规则，不能给数值加 px 后缀	300

表 1-2 canvas 元素的方法

属性	描述
getContext()	返回与该 canvas 元素相关的绘图环境对象。每个 canvas 元素均有一个这样的环境对象，而且每个环境对象均与一个 canvas 元素相关联
toDataURL(type, quality)	返回一个数据地址（data URL），你可以将它设定为 img 元素的 src 属性值。第一个参数指定了图像的类型，例如 image/jpeg 或 image/png，如果不指定第一个参数，则默认使用 image/png。第二个参数必须是 0 ～ 1.0 之间的 double 值，它表示 JPEG 图像的显示质量
toBlob(callback, type, args...)	创建一个用于表示此 canvas 元素图像文件的 Blob。第一个参数是一个回调函数，浏览器会以一个指向 blob 的引用为参数，去调用该回调函数。第二个参数以“image/png”这样的形式来指定图像类型。如果不指定，则默认使用“image/png”。最后一个参数是介于 0.0 ～ 1.0 之间的值，表示 JPEG 图像的质量。将来很可能会加入其他一些用于精确调控图像属性的参数

1.2 Canvas 的绘图环境

canvas 元素仅仅是为了充当绘图环境对象的容器而存在的，该环境对象提供了全部的绘制功能。尽管本书只关注 2d 绘图环境，不过 Canvas 规范也在着手准备支持其他类型的绘图环境。比如说，一个 3d 绘图环境的规范书就正在稳步制订之中。这一节将讲解 2d 绘图环境中的各种属性，同时也略带提一下 3d 绘图环境。

1.2.1 2d 绘图环境

在 JavaScript 代码中，很少会用到 canvas 元素本身，除了像前一节中讲到的那样，偶尔会

通过它来获取 canvas 的宽度、高度或某个数据地址。此外，可以通过 canvas 元素来获取指向 canvas 绘图环境对象的引用，这个绘图环境对象提供功能强大的 API，可以用来绘制图形与文本，显示并修改图像等等。实际上，本书剩余部分基本上专注于对 2d 绘图环境的讲解。

表 1-3 列出了 2d 绘图环境的全部属性。除了指向 canvas 元素自身的 canvas 属性之外，其余的 2d 绘图环境属性都与绘图操作有关。

表 1-3　CanvasRenderingContext2D 对象所含的属性

属性	简介
canvas	指向该绘图环境所属的 canvas 对象。该属性最常见的用途就是通过它来获取 canvas 的宽度与高度，分别调用 context.canvas.width 与 context.canvas.height 即可
fillstyle	指定该绘图环境在后续的图形填充操作中所使用的颜色、渐变色或图案
font	设定在调用绘图环境对象的 fillText() 或 strokeText() 方法时，所使用的字型
globalAlpha	全局透明度设定，它可以取 0（完全透明）～ 1.0（完全不透明）之间的值。浏览器会将每个像素的 alpha 值与该值相乘，在绘制图像时也是如此
globalCompsiteOperation	该值决定了浏览器将某个物体绘制在其他物体之上时，所采用的绘制方式。有效取值请参阅本书 2.14 节
lineCap	该值告诉浏览器如何绘制线段的端点。可以在以下三个值中指定一个：butt、round 及 square。默认值是 butt
lineWidth	该值决定了在 canvas 之中绘制线段的屏幕像素宽度。它必须是个非负、非无穷的 double 值。默认值是 1.0
lineJoin	告诉浏览器在两条线段相交时如何绘制焦点。可取的值是：bevel、round、miter。默认值是 miter
miterLimit	告诉浏览器如何绘制 miter 形式的线段焦点。该属性的详情请参阅本书 2.8.7 小节
shadowBlur	该值决定了浏览器该如何延伸阴影效果。值越高，阴影效果延伸得就越远。该值不是指阴影的像素长度，而是代表高斯模糊方程式中的参数值[⊖]。它必须是一个非负且非无穷量的 double 值，默认值是 0
shadowColor	该值告诉浏览器使用何种颜色来绘制阴影。通常采用半透明色作为该属性的值，以便让后面的背景能显示出来
shadowOffsetX	以像素为单位，指定了阴影效果的水平方向偏移量
shadowOffsetY	以像素为单位，指定了阴影效果的垂直方向偏移量
strokeStyle	指定了对路径进行描边时所用的绘制风格。该值可被设定为某个颜色、渐变色或图案
textAlign	决定了以 fillText() 或 strokeText() 方法进行绘制时，所画文本的水平对齐方式
textBaseline	决定了以 fillText() 或 strokeText() 方法进行绘制时，所画文本的垂直对齐方式

该表展示了 2d 绘图环境对象的所有属性。在第 2 章中，我们将逐个地详述这些属性。

⊖ 高斯模糊（Gaussian blur）是在图像处理软件中广泛使用的处理效果。这种模糊技术生成的图像看起来好像是经过一个半透明的屏幕来观察图像。在二维空间的方程式为：$G(u,v)=\frac{1}{2\pi\sigma^2}e^{-(u^2+v^2)/(2\sigma^2)}$，其中$u$、$v$为距离中心点的横、纵坐标，$r$是模糊半径($r^2=u^2+v^2$)，σ是正态分布的标准偏差。详情参阅：http://zh.wikipedia.org/zh-cn/高斯模糊。根据HTML5规范，该值的1/2将被作为方程式中σ参数的取值。详情参阅：http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#dom-context-2d-shadowblur。——译者注

提示：你可以扩充 2d 绘图环境对象的功能

与每个 canvas 相关联的绘图环境对象都是一个功能强大的图形引擎，它支持很多功能，诸如渐变色、图像合成（image compositing）、动画等等。不过，它也有局限性，比如，绘图环境对象之中就不包含绘制虚线（dashed line）的方法。由于 JavaScript 是一门动态语言，所以，你可以向该绘图环境中加入新的方法，或是对已有方法的功能进行扩充。更多信息，请参阅本书 2.8.6 小节。

3d 绘图环境 WebGL 简介

在 Canvas 中，有一个与 2d 绘图环境对应的 3d 绘图环境，叫做 WebGL，它完全符合 OpenGL ES [⊖] 2.0 的 API。可以访问以下网址，查看这份由 Khronos Group 所维护的 WebGL 标准：http://www.khronos.org/registry/webgl/specs/latest/。

在本书定稿时，浏览器厂商刚开始提供对 WebGL 的支持，仍然有类似 iOS4 及 IE10 这样主要的平台尚未支持 WebGL。然而，Canvas 的 3d 绘图环境是一项振奋人心的开发功能，它为各种各样前沿应用程序的制作提供了极大的便利。

1.2.2 Canvas 状态的保存与恢复

在第 1.2.1 小节之中，我们讲了 Canvas 绘图环境的所有属性。在进行绘图操作时，需要频繁设置这些值。很多时候只是想临时性地改变这些属性，比如说，可能需要在背景中绘制细网格线，然后用粗一些的线条在网格之上进行后续的绘图。在这种情况下，需要于绘制网格线时临时性地修改 lineWidth 属性。

Canvas 的 API 提供了两个名叫 save() 和 restore() 的方法，用于保存及恢复当前 canvas 绘图环境的所有属性。可以像下面讲的这样使用这两个方法：

```
function drawGrid(strokeStyle, fillStyle) {
   controlContext.save(); // Save the context on a stack

   controlContext.fillStyle = fillStyle;
   controlContext.strokeStyle = strokeStyle;

   // Draw the grid...

   controlContext.restore(); // Restore the context from the stack
}
```

save() 与 restore() 方法也许看上去没什么大不了的，不过一旦开始使用 Canvas 进行开发，就会发现它们其实是不可或缺的功能。这两个方法的用法总结在表 1-4 之中。

提示：save() 与 restore() 方法可以嵌套式调用

绘图环境的 save() 方法会将当前的绘图环境压入堆栈顶部。对应的 restore() 方法则会从堆栈顶部弹出一组状态信息，并据此恢复当前绘图环境的各个状态。这意味着可以嵌套式地调用 save()/restore() 方法。

⊖ OpenGL ES (OpenGL for Embedded Systems) 是 OpenGL 三维图形 API 的子集，针对手机、PDA和游戏主机等嵌入式设备而设计。该API由Khronos集团定义推广，Khronos是一个图形软硬件行业协会，主要关注图形和多媒体方面的开放标准。详情参阅：http://zh.wikipedia.org/zh-cn/OpenGL_ES。——译者注

表 1-4 CanvasRenderingContext2D 之中与状态操作有关的方法

方法	描述
save()	将当前 canvas 的状态推送到一个保存 canvas 状态的堆栈顶部。canvas 状态包括了当前的坐标变换（transformation）信息、剪辑区域（clipping region）以及所有 canvas 绘图环境对象的属性，包括 strokeStyle、fillStyle 与 globalCompositeOperation 等 canvas 状态并不包括当前的路径或位图。只能通过调用 beginPath() 来重置路径。至于位图，它是 canvas 本身的一个属性，并不属于绘图环境对象 请注意，尽管位图是 canvas 对象本身的属性，然而也可以通过绘图环境对象来访问它（在环境对象上调用 getImageData() 方法）
restore()	将 canvas 状态堆栈顶部的条目弹出。原来保存于栈顶的那一组状态，在弹出之后，就被设置成 canvas 当前的状态了，规范书规定，浏览器必须要根据该值来设定 canvas 的对应属性。因此，在调用 save() 与 restore() 方法之间，对 canvas 状态所进行的修改，其效果只会持续至 restore() 方法被调用之前

提示：绘图表面的保存与恢复

本小节向读者演示了如何对绘制环境对象的状态进行保存与恢复。与此同时，绘图表面的保存与恢复也是很有用的功能，我们将在 1.7 节之中讲述这个问题。

1.3 本书程序清单的规范格式

本书中的许多程序清单都将使用如下规范格式：

```
<!-- example.html -->

<!DOCTYPE html>
<html>
   <head>
     <title>Canonical Canvas used in this book</title>

      <style>
         ...
         #canvas {
            ...
         }
      </style>
   </head>
      <body>
         <canvas id='canvas' width='600'height='300'>
            Canvas not supported
         </canvas>

         <script src='example.js'></script>
      </body>
   </html>

  // example.js

  var canvas = document.getElementById('canvas'),
      context = canvas.getContext('2d');

  // Use the context...
```

刚才这段范例代码中包含一个 canvas 元素，它的 ID 属性是 canvas，而且这段代码使用了

一个名为 example.js 的 JavaScript 源代码文件。那份文件的代码中有两个变量，一个用于保存 canvas，一个用于保存其绘制环境。前述范例代码先使用 document.getElementById() 方法来获取指向 canvas 对象的引用，然后又获取了一份指向 canvas 绘图环境对象的引用。

本书中的大多数应用程序代码都会遵循上述规范格式，为了行文简洁，以后将会省略 HTML 部分的程序清单。与之类似，对于以行内形式印刷的程序清单，也就是像刚才那段代码一样，没有冠以“程序清单”的代码段，我们将直接使用 canvas 与 context 变量，而不再列出它们的初始化代码。

最后要说的是，为了简洁起见，本书中的程序清单代码并非都是完整的。书中经常会有一些范例，它们是构建在其他范例基础之上的。如果出现这种情况，那么你会看到最终那个范例的完整程序清单，同时也会部分列出其他相关程序清单的代码。

提示：谈一谈 User Agent

在 Canvas 规范书中，将 canvas 元素的实现者称为 User Agent（中文意为“用户代理”，是指代表软件用户发出行为指令的软件程序，业内一般直呼其英文，不做翻译。——译者注），简称 UA。该规范书使用的是这个术语，而不是“浏览器”（browser）一词，因为任何软件都可以实现 canvas 元素的功能，并不只有浏览器才行。

然而，本书还是将 canvas 元素的实现者叫做“浏览器”，因为如果使用“User Agent”这个术语，或是更糟糕地将其简称为 UA 的话，读者看起来会觉得陌生和困惑。

提示：本书中所引用的 URL

在本书中，有时会看到一些作为深入阅读参考资料的 URL 网址。如果它们都是可读的字符，而且不是很长的话，那么将会给出实际的 URL 来。对于那些不便输入的 URL，本书将使用缩略网址（shortened URL），它们虽然很难记忆，但是输入起来却很方便。

1.4 开始学习 HTML5

本节将简述 HTML5 开发环境，包括了运行应用程序所用的浏览器，以及诸如性能分析器（profiler）、时间轴（timeline）等开发过程中经常用到的开发工具。你可以先略读本节，稍后有需要时再来参考它。

1.4.1 规范

以下三个规范与本书所讲内容有关：

- HTML5 Canvas
- 基于脚本的定时控制动画（Timing control for script-based animation）
- HTML5 视频与音频（HTML5 video and audio）

由于历史原因，存在两个几乎完全一样的 Canvas 规范。其中一个是由 W3C 所维护的，它的网址是 http://dev.w3.org/html5/spec；另一个则是由 WHATWG 所维护的，其网址是 http://bit.ly/qXWjOl。而且，虽然在 WHATWG 所维护的规范中已经包含了 Canvas 绘图环境，不过 W3C 也有一份单独的规范，用于描述绘图环境，请参阅：http://dev.w3.org/html5/2dcontext。

长久以来，开发者一直使用 window.setInterval() 或 window.setTimeout() 方法来制作基于网络

的动画。然而，在第 5 章中大家将会看到，这些方法并不适用于对性能要求很高的（performance-critical）动画。应该使用 window.requestAnimationFrame() 方法来取代它们。该方法定义在名为“基于脚本的定时控制动画”规范之中，其网址是：http://www.w3.org/TR/animation-timing。

最后，本书将会向读者演示如何将 HTML5 视频与音频元素集成入基于 Canvas 的应用程序之中。HTML5 视频与音频元素的标准与 HTML5 标准本身都描述在同一份规范中，其网址是：http://www.w3.org/TR/html5/video.html。

1.4.2 浏览器

本书 2012 年年初出版之时，5 大主流浏览器——Chrome、Internet Explorer、Firefox、Opera 及 Safari 都对 HTML5 的 Canvas 元素提供了广泛的支持。有时候会产生些许的不兼容现象，比如本书 2.14 节，就解释了导致合成效果不兼容的原因。虽说如此，浏览器厂商们还是在遵从规范标准与提供高效实现这两件事情上做得很棒。

Chrome、Firefox、Opera 以及 Safari 这 4 个浏览器，对 HTML5 的支持已经有好一阵子了，Microsoft 的 Internet Explorer 稍稍来迟了一步，它从 IE9 版本开始，才提供了对 HTML5 的广泛支持。不过，Microsoft 在 IE9 与 IE10 中对 Canvas 的支持效果很出众，实际上，在本书出版时，这两个版本的浏览器所提供的 Canvas 实现在 5 大浏览器中是最快的。

如果所实现的基于 Canvas 的应用程序必须支持 IE6、IE7 及 IE8 的话，那么有两个选择。其中一个是使用 explorecanvas，它可以在老版本的 Internet Explorer 浏览器中增加对于 Canvas 的支持。另一个则是使用 Google Chrome Frame，它将 IE 引擎替换成 Google Chrome 浏览器的引擎。explorercanvas 与 Google Chrome Frame 都可以在 Google 网站上找到，如图 1-4 所示。

图 1-4 Google 网站中的 explorercanvas 与 Google Chrome Frame 项目，它们都支持 IE61718

1.4.3 控制台与调试器

所有支持 HTML5 的主流浏览器都提供了控制台（console）与调试器（debugger）。实际上，浏览器厂商之间经常互相借鉴设计思想：基于 WebKit [⊖]的 Firefox、Opera 与 IE 浏览器，它们的控制台与调试器都非常相似。

⊖ WebKit是供网页浏览器渲染页面所用的开源排版引擎。详情参考：http://en.wikipedia.org/wiki/WebKit。——译者注

图 1-5 是 Safari 浏览器的控制台与调试器界面。

图 1-5 Safari 浏览器的控制台与调试器

可以调用 console.log() 方法，向控制台写入信息。传递给该方法一个字符串，它就会显示在控制台之中。调试器的界面很标准，可以在此设定断点、监控表达式、查看变量及调用堆栈等等。

要对不同浏览器所提供的开发工具展开论述，则超出了本书的范围。要想深入了解有关 Chrome 浏览器开发工具的更多资讯，请参阅 Chrome Developer Tools 文档，该文档如图 1-6 所示。其余浏览器也都提供了类似的文档。

图 1-6 Chrome Developer Tools 文档

小技巧：以编程的方式来启动与停止性能分析器

正如在图 1-6 中看到的那样，在基于 WebKit 的浏览器中，可以通过点击性能分析窗口下方的实心圆圈按钮来启动性能分析。

然而，以点击按钮的方式来启动性能分析，有时是不够的。举例来说，你也许希望在执行到某几行特定的代码时开始分析，在执行完它们之后就停止性能分析。在基于 WebKit 的浏览器中，调用这两个方法就可以完成该功能：console.profile() 及 console.profileEnd()。可以像这样来使用它们：

```
console.profile('Core HTML5 Animation,
                erasing the background');
//...

console.profileEnd();
```

1.4.4 性能

在大多数情况下，采用 Canvas 来实现的应用程序都会有极好的性能。不过，如果你在制作动画、游戏，或是实现基于 Canvas 的手机应用程序，那么可能需要做一些性能优化。

在本小节之中，我们将简单介绍一些可供开发者使用的性能瓶颈侦测工具。为了演示这些工具的用法，我们会用到如图 1-7 中所示的这个应用程序。本书在第 5 章中将会讲解这个动画的制作，它让三个实心圆形同时在屏幕上运动。

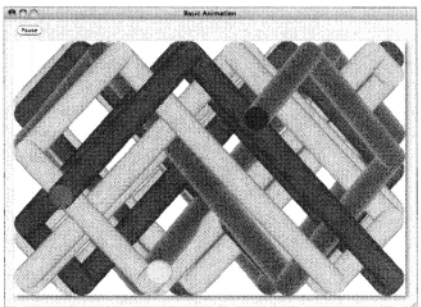


图 1-7　第 5 章中的动画程序

我们要讲解如下三个工具的用法：

- 性能分析器（Profiler）
- 时间轴工具（Timeline）
- jsPerf

上述列表中的前两个工具都直接由浏览器提供，或是可以通过安装“附加元件”（add-on）来取得。然而，jsPerf 则是一个网站，可以在此创建性能测试，并将其发布给大众。在本小节下面几个段落中，我们先来看看 Chrome 与 Safari 浏览器所提供的性能分析器与时间轴工具，然后再了解一下 jsPerf。

1.4.4.1 性能分析器与时间轴工具

要想发现代码中的性能瓶颈，性能分析器与时间轴工具是必不可少的。图 1-8 与图 1-9 分别展示了使用时间轴工具与性能分析器对图 1-7 中的动画程序进行分析时的样子。

时间轴工具可以将应用程序中发生的重要事件记录下来，同时也一并记录了这些事件中的细节，包括事件的持续时间，以及其所影响的窗口区域等等。在基于 WebKit 的 Chrome、Safari 等浏览器之中，可以将鼠标悬停在这些事件之上，这样就可以获取与之相关的细节信息了，如图 1-8 所示。

通过性能分析器，可以更为详尽地观察到程序代码在函数级别的性能表现。正如在图 1-9 中看到的那样，它可以显示出每个函数被调用的次数，以及这些函数调用所花费的时间。你可以看到每个函数的执行时间在总执行时间中所占的百分比，而且还可以精确地看到每个函数被执行一次平均要花费多少毫秒。

图 1-8 时间轴工具

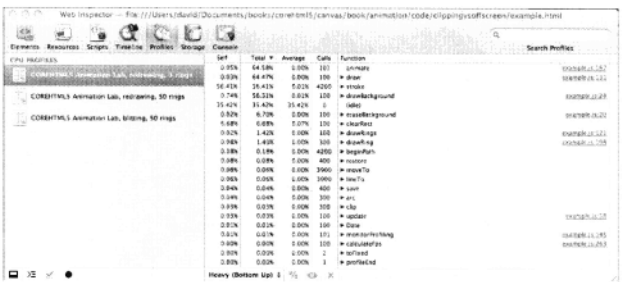


图 1-9 性能分析器

1.4.4.2 jsPerf

图 1-10 中所示的 jsPerf 网站，可以让你创建并分享 JavaScript 性能测试[⊖]。

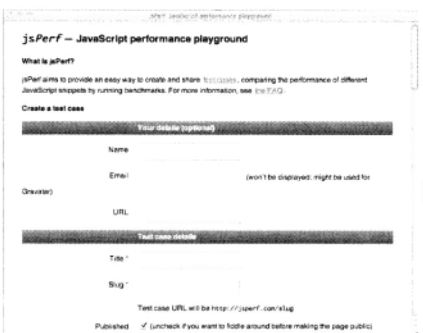


图 1-10 jsperf.com 网站首页

比方说，你可能会好奇，在处理 canvas 中的图像时，究竟何种方式才能够最为高效地将所有像素都遍历一次呢？点击图 1-10 顶部的“test cases”链接，jsPerf 网站就会将所有公开的测试用

⊖ Benchmark，俗称“跑分”。——译者注

例都显示出来，如图 1-11 所示。

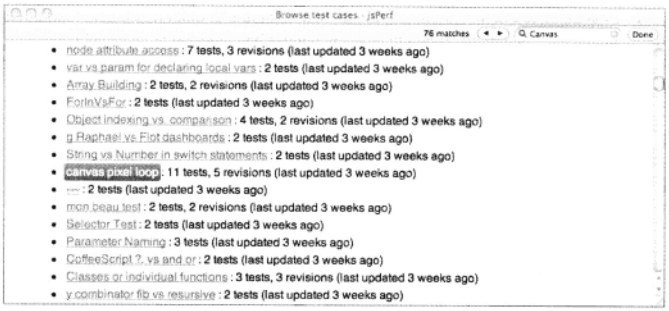


图 1-11　jsperf.com 网站中的 Canvas 测试用例代码

实际上，jsperf.com 网站上不仅有许多与 Canvas 有关的测试用例，而且直接就能找到一个与上段描述相符的测试用例来。该测试用例就是图 1-11 中加亮显示的那一个。如果点击一下那个测试用例的链接，jsPerf 就会将此用例的代码显示出来，如图 1-12 所示。可以自己来运行这个测试用例，运行结果会被加入此测试用例的统计信息之中。此外，还可以看到使用其他浏览器的用户运行此测试用例的结果（这部分内容没有包含在图 1-12 之中）。

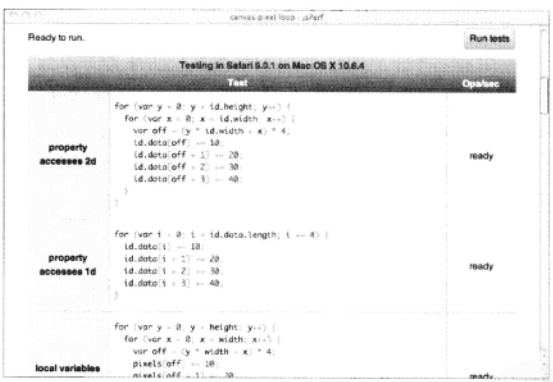


图 1-12　用于遍历图片像素的测试用例

准备知识已经讲完了，现在我们来看看如何在 canvas 之上进行绘制。

1.5　基本的绘制操作

在下一章中，将详细讲述 canvas 的绘制。现在，为了让读者熟悉一下 Canvas 所提供的绘图 API 方法，所以咱们先来看看图 1-13 中的这个程序，它是一个带指针的时钟（analog clock）。

该时钟的代码列在程序清单 1-4 之中，它用到了如下的 Canvas 绘图 API：

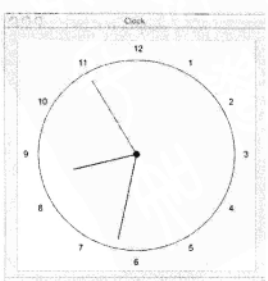


图 1-13　时钟程序

- arc()
- beginPath()
- clearRect()
- fill()
- fillText()
- lineTo()
- moveTo()
- stroke()

像 Adobe Illustrator [⊖]与 Apple 的 Cocoa 框架[⊜]那样，Canvas 也可以让开发者先创建不可见的路径，稍后再调用 stroke() 来描绘路径的边缘，或者调用 fill() 来对路径的内部进行填充，使路径变得可见。可以调用 beginPath() 方法来开始定义某一段路径。

在时钟应用程序的代码中，drawCircle() 方法绘制了一个表示钟面的圆形，该方法先调用 beginPath() 来开始定义路径，然后再调用 arc() 来创建一个圆形的路径。等到应用程序的代码调用了 stroke() 方法之后，刚才定义的路径才会变得可见。与之类似，该程序的 drawCenter() 方法也是通过组合调用 beginPath()、arc() 与 fill() 这几个方法来绘制时钟中心的那个小实心圆的。

该应用程序的 drawNumerals() 方法通过调用 fillText() 来绘制钟面周围的数字，fillText() 这个方法是用来在 canvas 中进行文本填充的。与 arc() 方法不同，fillText() 方法并不会创建路径，而是立即将文本渲染到 canvas 之上。

钟表的指针则是通过应用程序代码中的 drawHand() 方法来绘制的。该方法调用了如下三个用于绘制线段方法将钟表指针绘制出来：moveTo()、lineTo() 与 stroke()。先调用 moveTo() 方法将画笔（graphics pen）移动到 canvas 中的指定地点，然后调用 lineTo() 方法，在该点与另外一个指定的点之间绘制一条不可见的路径，最后使用 stroke() 方法将当前路径变为可见。

应用程序通过调用 setInterval() 方法来制作时钟的动画效果。该方法每秒钟都会调用一次 drawClock() 函数。而后者则使用 clearRect() 方法来擦除 canvas，然后再重绘时钟。

程序清单 1-4　一个基本的时钟程序

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    FONT_HEIGHT = 15,
    MARGIN = 35,
    HAND_TRUNCATION = canvas.width/25,
    HOUR_HAND_TRUNCATION = canvas.width/10,
    NUMERAL_SPACING = 20,
    RADIUS = canvas.width/2 - MARGIN,
    HAND_RADIUS = RADIUS + NUMERAL_SPACING;

//Functions.............................................................
```

⊖ Illustrator是Adobe系统公司推出的基于矢量的图形制作软件。该软件的最大特征在于贝塞尔曲线的使用，使得操作简单功能强大的矢量绘图成为可能。现在它还集成文字处理、上色等功能，不仅在插图制作，在印刷制品（如广告传单、小册子）设计制作方面也广泛使用，事实上，它已经成为桌面出版（DTP）业界的默认标准。该软件的主页是：http://www.adobe.com/products/illustrator/。——译者注

⊜ Cocoa是苹果公司为Mac OS X所创建的原生面向对象的编程环境，对最终用户来说，使用Cocoa编程环境开发的应用程序即为Cocoa 应用。这类应用有独特的外观，因为Cocoa编程环境让程序在多方面自动遵循苹果公司的人机界面守则。详情参见：http://zh.wikipedia.org/zh-cn/Cocoa。——译者注

```
function drawCircle() {
   context.beginPath();
   context.arc(canvas.width/2, canvas.height/2,
               RADIUS, 0, Math.PI*2, true);
   context.stroke();
}

function drawNumerals() {
   var numerals = [ 1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12 ],
       angle = 0,
       numeralWidth = 0;

   numerals.forEach(function(numeral) {
      angle = Math.PI/6 * (numeral-3);
      numeralWidth =context.measureText(numeral).width;
      context.fillText(numeral,
         canvas.width/2 + Math.cos(angle)*(HAND_RADIUS) -
            numeralWidth/2,
         canvas.height/2 + Math.sin(angle)*(HAND_RADIUS) +
            FONT_HEIGHT/3);
  });
}
function drawCenter() {
   context.beginPath();
   context.arc(canvas.width/2, canvas.height/2,5, 0, Math.PI*2, true);
   context.fill();
}

function drawHand(loc, isHour) {
   var angle = (Math.PI*2) * (loc/60) -Math.PI/2,
      handRadius = isHour ? RADIUS -HAND_TRUNCATION-HOUR_HAND_TRUNCATION
                          : RADIUS -HAND_TRUNCATION;

   context.moveTo(canvas.width/2,canvas.height/2);
   context.lineTo(canvas.width/2 +Math.cos(angle)*handRadius,
                  canvas.height/2 +Math.sin(angle)*handRadius);
   context.stroke();
}

function drawHands() {
   var date = new Date,
       hour = date.getHours();

   hour = hour > 12 ? hour - 12 : hour;

   drawHand(hour*5 + (date.getMinutes()/60)*5,true,0.5);
   drawHand(date.getMinutes(), false,0.5);
   drawHand(date.getSeconds(), false,0.2);
}

function drawClock() {
   context.clearRect(0,0,canvas.width,canvas.height);

   drawCircle();
   drawCenter();
   drawHands();
   drawNumerals();
}
//Initialization.............................................

context.font = FONT_HEIGHT + 'px Arial';
loop = setInterval(drawClock, 1000);
```

提示：进一步研究路径、描边与填充

本节的时钟范例程序展示了 canvas 图形绘制的概况，在第 2 章之中，我们将详细地研究如何在 canvas 中进行图形的绘制与处理。

1.6 事件处理

HTML5 应用程序是以事件来驱动的。可以在 HTML 元素上注册事件监听器，并编写用于响应这些事件的实现代码。几乎所有基于 Canvas 的应用程序都会处理鼠标或触摸事件，有些程序则两种事件都会处理，还有许多程序也会处理其他各种类型的事件，比如键盘事件或者拖放事件。

1.6.1 鼠标事件

在 canvas 中检测鼠标事件是非常简单的：可以在 canvas 中增加一个事件监听器，当事件发生时，浏览器就会调用这个监听器了。举例来说，要监听“按下鼠标事件”，你可以这样做：

```
canvas.onmousedown = function (e) {
    // React to the mouse down event
};
```

此外，可以使用更为通用的 addEventListener() 方法来注册监听器：

```
canvas.addEventListener('mousedown', function (e){
    // React to the mouse down event
});
```

除了 onmousedown 之外，也可以使用 onmousemove、onmouseup 与 onmouseout 来注册监听器。

使用 onmousedown、onmousemove 这样的方式来注册监听器，比调用 addEventListener() 要稍微简单些，不过，如果要向某个鼠标事件注册许多个监听器的话，那么就得使用 addEventListener() 方法了。

将鼠标坐标转换为 Canvas 坐标

浏览器通过事件对象传递给监听器的鼠标坐标，是窗口坐标（window coordinate），而不是相对与 canvas 自身的坐标。

大部分情况下，开发者需要知道的是发生鼠标事件的点相对于 canvas 的位置，而不是在整个窗口中的位置，所以必须进行坐标转换。举例来说，在图 1-14 中，canvas 上面显示了一幅精灵表（sprite sheet）的图像。精灵表指的是一张包含许多动画图像的图片。在动画的播放过程中，每次都要在精灵表中选取一张图像显示出来，这意味着你必须知道精灵表中每张图像的精确坐标。

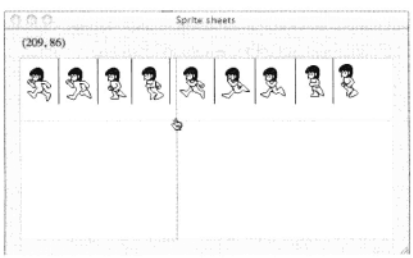


图 1-14 精灵表坐标查看器

借助图 1-14 中所示的应用程序，可以通过移动鼠标来观察屏幕中显示的坐标，以此确定精灵表中每张精灵图像的具体位置坐标。当用户移动鼠标时，应用程序会持续地更新精灵表上方的鼠标坐标，同时还会在屏幕上绘制辅助线。

该应用程序向 canvas 注册了一个 mousemove 事件监听器，然后，等到浏览器回调这个监听器时，应用程序会将相对与窗口的鼠标坐标转换为 canvas 坐标。转换工作是通过类似下面这样的 windowToCanvas() 方法来完成的：

```
function windowToCanvas(canvas, x, y) {
   var bbox = canvas.getBoundingClientRect();

   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top  * (canvas.height /bbox.height)
          };
}

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(canvas, e.clientX,e.clientY);

   drawBackground();
   drawSpritesheet();
   drawGuidelines(loc.x, loc.y);
   updateReadout(loc.x, loc.y);
};
...
```

上述 windowToCanvas() 方法在 canvas 对象上调用 getBoundingClientRect() 方法，来获取 canvas 元素的边界框（bounding box），该边界框的坐标是相对于整个窗口的。然后，windowToCanvas() 方法返回了一个对象，其 x 与 y 属性分别对应于鼠标在 canvas 之中的坐标。

请注意，windowToCanvas() 方法不只是将 canvas 边界框的 x、y 坐标从窗口坐标中减去，而且在 canvas 元素大小与绘图表面大小不相符时，它还对这两个坐标进行了缩放。请参阅本书 1.1.1 小节，该小节解释了 canvas 元素大小与 canvas 绘图表面大小的区别。

图 1-14 中所示应用程序的 HTML 代码列在了程序清单 1-5 之中，其 JavaScript 代码列在了程序清单 1-6 之中。

程序清单 1-5　精灵表坐标查看器的 HTML 代码

```
<!DOCTYPE html>
   <head>
     <title>Sprite sheets</title>

     <style>
        body {
           background: #dddddd;
        }

        #canvas {
           position: absolute;
           left: 0px;
           top: 20px;
           margin: 20px;
           background: #ffffff;
           border: thin inset rgba(100,150,230,0.5);
           cursor: pointer;
        }

        #readout {
```

```
            margin-top: 10px;
            margin-left: 15px;
            color: blue;
        }
    </style>
  </head>

  <body>
    <div id='readout'></div>

    <canvas id='canvas' width='500' height='250'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

程序清单 1-6　精灵表坐标查看器的 JavaScript 代码

```
var canvas = document.getElementById('canvas'),
    readout = document.getElementById('readout'),
    context = canvas.getContext('2d'),
    spritesheet = new Image();

//Functions..........................................................

function windowToCanvas(canvas, x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top * (canvas.height /bbox.height)
          };
}

function drawBackground() {
   var VERTICAL_LINE_SPACING = 12,
       i = context.canvas.height;

   context.clearRect(0,0,canvas.width,canvas.height);
   context.strokeStyle = 'lightgray';
   context.lineWidth = 0.5;

   while(i > VERTICAL_LINE_SPACING*4) {
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
      i -= VERTICAL_LINE_SPACING;
   }
}

function drawSpritesheet() {
   context.drawImage(spritesheet, 0, 0);
}

function drawGuidelines(x, y) {
   context.strokeStyle = 'rgba(0,0,230,0.8)';
   context.lineWidth = 0.5;
   drawVerticalLine(x);
   drawHorizontalLine(y);
}

function updateReadout(x, y) {
```

```
    readout.innerText = '(' + x.toFixed(0) + ', '+ y.toFixed(0) + ')';
}

function drawHorizontalLine (y) {
    context.beginPath();
    context.moveTo(0,y + 0.5);
    context.lineTo(context.canvas.width, y + 0.5);
    context.stroke();
}

function drawVerticalLine (x) {
    context.beginPath();
    context.moveTo(x + 0.5, 0);
    context.lineTo(x + 0.5,context.canvas.height);
    context.stroke();
}

// Event handlers.....................................................

canvas.onmousemove = function (e) {
    var loc = windowToCanvas(canvas, e.clientX,e.clientY);

    drawBackground();
    drawSpritesheet();
    drawGuidelines(loc.x, loc.y);
    updateReadout(loc.x, loc.y);
};

//Initialization.....................................................

spritesheet.src = 'running-sprite-sheet.png';
spritesheet.onload = function(e) {
    drawSpritesheet();
};

drawBackground();
```

小技巧：x、y 属性与 clientX、clientY 属性

在 HTML5 规范出现之前，通过浏览器传给事件监听器的事件对象，来获取鼠标事件发生的窗口坐标，其实现方法非常混乱。有些浏览器将坐标存放在 x、y 属性之中，另外一些则将其存放于 clientX、clientY 属性中。所幸，当前支持 HTML5 的浏览器最终达成一致，它们都支持 clientX 与 clientY 属性了。有关这些事件属性的详细信息，请参阅：http://www.quirksmode.org/js/events_mouse.html。

小技巧：让浏览器不再干预事件处理

当你在监听鼠标事件时，一旦相关的事件发生，浏览器就会调用你所注册的监听器。在处理完某个事件后，浏览器可能也会对该事件作出反应。大多数情况下，如果在 canvas 中处理了某个鼠标事件，那么处理完之后，就不会再需要浏览器也去处理它了。因为如果那样的话，可能会导致一些不好的效果，比如浏览器会选中其他 HTML 元素，或者改变光标的位置等等。

幸好事件对象中有一个 preventDefault() 方法，顾名思义，它可以阻止浏览器对该事件作出默认的反应。在你所编写的事件处理器（event handler，也叫事件句柄、事件处理程序，本译文中均使用“事件处理器”这个叫法。——译者注）代码中调用该方法，浏览器就不会再干预该事件的处理了。

提示：Canvas 绘图环境对象的 drawImage() 方法

图 1-14 中那个程序的范例代码，使用了 2d 绘图环境对象的 drawImage() 方法来绘制精灵表。该方法可以将某图像的全部或者一部分从某个地方复制到一个 canvas 之中。如果有需要的话，还可以在复制的过程中对图像进行缩放。

精灵表应用程序的代码，以最为简单的形式调用了该方法：它将存放于 Image 对象中的全部图像内容，未经缩放地绘制到应用程序的 canvas 之中。在本书剩下的章节中，尤其是第 4 章中，将看到更多有关 drawImage() 方法的高级应用。

1.6.2 键盘事件

当在浏览器窗口中按下某个键时，浏览器将会生成键盘事件。这些事件发生在当前拥有焦点的 HTML 元素身上。假如没有元素拥有焦点，那么事件的发生地将会上移至 window 与 document 对象。

canvas 是一个不可获取焦点的元素，所以，根据上一段中所讲的内容，在 canvas 元素上新增键盘事件监听器是徒劳的。如果想要检测键盘事件的话，你应该在 document 或 window 对象上新增键盘事件监听器才对。

一共有三种键盘事件：

- keydown
- keypress
- keyup

keydown 与 keyup 是底层事件，几乎每次按键时，浏览器都会触发这些事件。请注意，有些按键敲击动作，例如某个特定的组合键，会被浏览器或操作系统吞掉，不过，绝大部分按键敲击，包括 Alt、Esc 等按键，还是能够通知给 keydown 与 keyup 事件处理器的。

如果激发 keydown 事件的那个按键会打印出某个字符，那么浏览器将会在触发 keyup 事件之前先产生 keypress 事件。如果在一段时间内持续地按住某个可以打印出字符的键，那么浏览器就会在 keydown 与 keyup 事件之间产生一系列的 keypress 事件。

按键监听器的实现方法同鼠标监听器类似。可以将某个函数赋值给 document 或 window 对象的 onkeydown、onkeyup 或 onkeypress 变量，也可以用 keydown、keyup 或 keyPress 作为第一个参数，用指向某个函数的引用作为第二个参数，来调用 addEventListener() 方法。

想要弄清楚按下的到底是哪个键，这是个非常复杂的事情。原因有两个。第一，全世界各种语言里有海量的字符，你必须将拉丁字母、亚洲语言中的表意字符（Asian ideographic characters），以及印度的许多种语言都考虑在内。这只是举了几个例子而已，要支持所有的语言，将是难以置信的。

第二，虽说浏览器与键盘已经存在很久了，但直到 DOM Level 3 规范[⊖]出现，键值码（key code）才被标准化。 现在很少有浏览器支持这个规范。总而言之，要想弄清楚被按下的到底是哪个键或哪个组合键序列，是一笔糊涂账。

不过，在绝大多数情况下，都可以使用如下两条简单的对策来处理这个问题：

- 处理 keydown 与 keyup 事件时，看看浏览器传给事件监听器的那个事件对象中 keyCode 属性的值。一般来说，如果是可打印的字符，那么属性值就会是其 ASCII 码。请注意刚才那

⊖ 原书作者所说的DOM Level 3规范，具体是指“Document Object Model (DOM) Level 3 Events Specification”，其官方页面是：http://www.w3.org/TR/DOM-Level-3-Events/。——译者注

句话中的“一般”。有一个网站不错，你可以在处理不同浏览器传过来的键值码时参考该页面：http://bit.ly/o3b1L2。键盘事件所产生的事件对象，也包含下述 boolean 属性：

- altKey
- ctrlKey
- metaKey
- shiftKey

- 由于浏览器只会在产生可打印字符时才触发 keypress 事件，所以，在处理该事件时，可以放心地使用如下方式来获取字符：

```
var key = String.fromCharCode(event.which);
```

通常来说，除非要在 canvas 中实现字符控制功能，否则处理鼠标事件的场合要远远多于处理键盘事件。键盘事件的另一个常见用途，就是在游戏之中处理按键动作。那个话题我们将在第 9 章之中讨论。

1.6.3 触摸事件

随着智能手机与平板电脑的出现，HTML 规范也加入了对触摸事件的支持。要了解更多有关触摸事件处理的信息，请参阅本书第 11 章。

1.7 绘制表面的保存与恢复

在本书 1.2.2 小节之中，读者已经看到了如何对绘图环境对象的各个状态进行保存与恢复。绘图环境状态的保存与恢复，可以让开发者很方便地做出临时性的状态改动，这在编码过程中是很常见的。

Canvas 绘制环境对象的另外一个关键功能就是可以对绘图表面自身进行保存与恢复。这种绘制表面的保存与恢复功能，可以让开发者在绘图表面上进行一些临时性的绘制动作，诸如绘制橡皮带线条、辅助线（guidewire）或注解（annotation），这些绘制对于许多工作都是很有用的。例如，图 1-15 中的应用程序可以让用户通过拖动鼠标来交互式地创建多边形，该应用程序会在本书 2.13.1 小节中讨论。

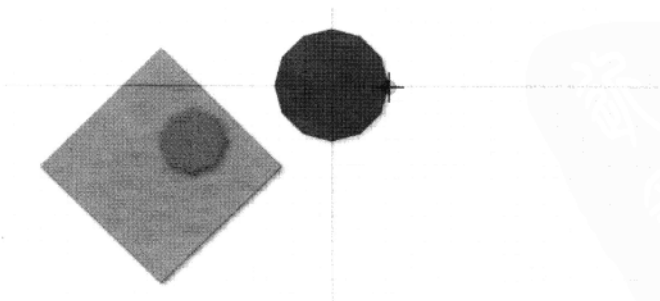


图 1-15 绘制辅助线

检测到鼠标按下的事件之后，应用程序就将绘图表面保存起来。在接下来用户拖动鼠标的过程中，应用程序持续地将按下鼠标那一刻的绘图表面恢复到 canvas 之中，然后再绘制多边形与相

关的辅助线。当用户松开鼠标时，应用程序最后一次将绘图表面恢复到 canvas，再将最终不含辅助线的多边形绘制在其上。

程序清单 1-7 之中，列出了图 1-15 的应用程序中与绘制辅助线有关的那部分代码。此应用程序的完成程序清单，请查看本书 2.11 节。

提示：使用 getImageData() 与 putImageData() 方法来操作图像

图 1.15 中的应用程序，使用 getImageData() 与 putImageData() 方法来保存与恢复绘图环境的绘图表面。与 drawImage() 方法一样，getImageData() 与 putImageData() 方法也有几种不同的用途。一种常见的用法是通过它来实现图像滤镜：先获取图像数据，然后处理，最后将它恢复到 canvas 之上。本书会介绍很多种 getImageData() 与 putImageData() 的用法，其中 4.5.2.3 小节中会讲到如何用这两个方法来实现滤镜效果。

提示：立即模式绘图系统

canvas 元素是采用“立即模式”（immediate-mode）来绘制图形的，这意味着它会立刻将你所指定的内容绘制在 canvas 上。然后，它就会立刻忘记刚才绘制的内容，这表示 canvas 之中不会包含将要绘制的图形对象列表。某些绘图系统，比如 SVG，则会维护一份所绘图形对象的列表。这些绘图系统被叫做“保留模式”（retained-mode ）绘图系统。

由于立即模式并不维护所绘制对象的列表，所以相对于保留模式来说，它是一种更加底层的绘图模式。立即模式也更加灵活，因为它是直接向屏幕上绘制的，而不是像保留模式那样，需要调整由绘图系统传递过来的图形对象。

立即模式的绘制系统更适合制作“绘画应用程序”（paint application），这种程序不需要跟踪记录用户所绘制的东西，而保留模式的绘制系统则更适合制作“画图应用程序”（drawing application），此种应用程序可以让用户操作其所创建的图形对象。

在本书 2.11 节中，你将会看到如何实现一个简单的保留模式绘图系统，该系统将一个画图应用程序所用到的多边形维护在一个数组之中，以便让用户可以通过拖放来改变这些多边形的位置。

程序清单 1-7　通过保存与恢复绘图表面来绘制辅助线

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

// Save and restore drawing surface...................................

function saveDrawingSurface() {
   drawingSurfaceImageData =context.getImageData(0, 0,
                            canvas.width,
                            canvas.height);
}

function restoreDrawingSurface() {
   context.putImageData(drawingSurfaceImageData, 0,0);
}

// Event handlers.....................................................

canvas.onmousedown = function (e) {
   ...
```

```
    saveDrawingSurface();
    ...
};

canvas.onmousemove = function (e) {
    var loc = windowToCanvas(e);

    if (dragging) {
        restoreDrawingSurface();
        ...

        if (guidewires) {
          drawGuidewires(mousedown.x, mousedown.y);
        }
    }
};

canvas.onmouseup = function (e) {
    ...
    restoreDrawingSurface();
};
```

1.8 在 Canvas 中使用 HTML 元素

尽管我们可以说 Canvas 是 HTML5 之中最棒的功能，不过在实现网络应用程序时，很少会单独使用它。在绝大多数情况下，你都会将一个或更多的 canvas 元素与其他 HTML 控件结合起来使用，以便让用户可以通过输入数值或其他方式来控制应用程序。

要将其他 HTML 控件与 canvas 结合起来使用，首先想到的办法可能是将控件嵌入到 canvas 元素之中。不过这么做不行，因为任何放入 canvas 元素主体部分的东西，只有在浏览器不支持 canvas 元素时，才会被显示出来。

浏览器要么显示 canvas 元素，要么显示放在元素之中的 HTML 控件，它不会将两者同时显示出来。所以，必须将控件放在 canvas 元素之外。

为了让 HTML 控件看上去好像是出现在 canvas 范围内，可以使用 CSS 将这些控件放置在 canvas 之上。图 1-16 中的应用程序演示了这个效果。

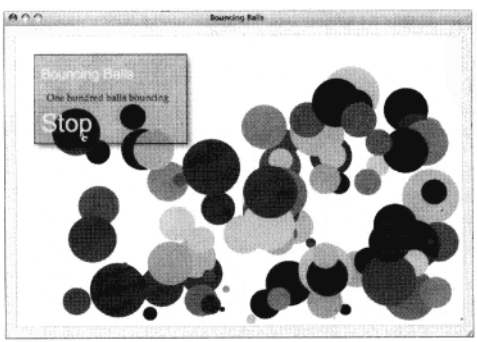


图 1-16 显示在 canvas 之上的 HTML 元素

图 1-16 之中的应用程序显示了 100 个运动的小球，并且提供了一个用于启动或停止动画效果的链接。这个存在于 DIV 元素之中的链接是半透明的，并且浮动在 canvas 之上。我们将这种 DIV 叫做“玻璃窗格”（glass pane），因为它看起来像是一个浮动在 canvas 元素之上的玻璃板。

程序清单 1-8 列出了图 1-16 之中这个应用程序的 HTML 代码。

程序清单 1-8　用于在 canvas 之中显示 HTML 控件的 HTML 代码

```
<!DOCTYPE html>
<html>
   <head>
      <title>Bouncing Balls</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            margin-left: 10px;
            margin-top: 10px;
            background: #ffffff;
            border: thin solid #aaaaaa;
         }

         #glasspane {
            position: absolute;
            left: 50px;
            top: 50px;
            padding: 0px 20px 10px 10px;
            background: rgba(0, 0, 0, 0.3);
            border: thin solid rgba(0, 0, 0, 0.6);
            color: #eeeeee;
            font-family: Droid Sans, Arial,Helvetica, sans-serif;
            font-size: 12px;
            cursor: pointer;
            -webkit-box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
            -moz-box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
            box-shadow: rgba(0,0,0,0.5) 5px 5px 20px;
         }

         #glasspane h2 {
            font-weight: normal;
         }

         #glasspane .title {
            font-size: 2em;
            color: rgba(255, 255, 0, 0.8);
         }

         #glasspane a:hover {
            color: yellow;
         }

         #glasspane a {
            text-decoration: none;
            color: #cccccc;
            font-size: 3.5em;
         }

         #glasspane p {
```

```
            margin: 10px;
            color: rgba(65, 65, 220, 1.0);
            font-size: 12pt;
            font-family: Palatino, Arial,Helvetica, sans-serif;
         }
      </style>
   </head>

   <body>
      <div id='glasspane'>
         <h2 class='title'>Bouncing Balls</h2>

         <p>One hundred balls bouncing</p>

         <a id='startButton'>Start</a>
      </div>

      <canvas id='canvas' width='750' height='500'>
         Canvas not supported
      </canvas>

      <script src='example.js'></script>
   </body>
</html>
```

程序清单 1-8 中的 HTML 代码，通过 CSS 来确定玻璃窗格的绝对位置，让其显示在 canvas 之上，像是这样：

```
#canvas {
   margin-left: 10px;
   margin-top: 10px;
   background: #ffffff;
   border: thin solid #aaaaaa;
}

#glasspane {
   position: absolute;
   left: 50px;
   top: 50px;
   ...
}
```

前面一份 CSS 样式表使用了相对定位方式来指定 canvas 元素的位置，这也是 CSS 之中 position 属性的默认值。然而，后面一份样式表则使用绝对定位方式来指定玻璃窗格的位置。CSS 规范书中规定：采用绝对定位方式的元素将被绘制在采用相对定位方式的元素之上。这也就是图 1-16 中为何玻璃窗格会显示在 canvas 之上的原因。

如果将 canvas 的定位方式也改成绝对定位的话，那么 canvas 将会出现在玻璃窗格上方，这样的话，就看不见玻璃窗格了，因为 canvas 的背景不是透明的。在那种情况下，玻璃窗格位于 canvas 的下方，因为 canvas 元素出现在代表玻璃窗格的 DIV 元素后方。如果改变这两个元素的顺序，那么玻璃窗格又会重新出现在 canvas 之上了。

所以说，将玻璃窗格放置在 canvas 之上，可以采用两种办法：要么对 canvas 使用相对定位，对玻璃窗格使用绝对定位；要么对两个元素都采用相对或绝对定位，并且把代表玻璃窗格的 DIV 元素放置在 canvas 元素之后。

还有第三种办法，那就是对这两个元素都采用相对或绝对定位方式，然后调整其 CSS 之中的 z-index 属性。浏览器会将 z-index 值较大的元素绘制在 z-index 值较小元素的上面。

除了放置好需要显示的 HTML 控件，还需要在 JavaScript 代码中获取指向这些控件的引用，以便访问并修改它们的属性值。

图 1-16 之中的应用程序代码，获取了指向玻璃窗格以及动画控制按钮的引用，并向其增加了事件处理器，像是这样：

```
var context =document.getElementById('canvas').getContext('2d'),
    startButton =document.getElementById('startButton'),
    glasspane =document.getElementById('glasspane'),
    paused = false,
    ...

startButton.onclick = function(e) {
   e.preventDefault();
   paused = ! paused;
   startButton.innerHTML = paused ? 'Start' :'Stop';
};
...

glasspane.onmousedown = function(e) {
   e.preventDefault();
};
```

上述 JavaScript 代码向按钮控件增加了一个 onclick 事件处理器，该监听器根据应用程序当前的状态来启动或暂停动画效果，这段代码同时还向玻璃窗格控件增加了一个 onmousedown 事件处理器，用以阻止浏览器对于鼠标点击的默认反应。onmousedown 事件处理器将会阻止浏览器对该事件做出反应，以避免用户无意间选中了玻璃窗格控件。

提示：实现基于 Canvas 的自定义控件

Canvas 规范中说，应该首先考虑使用内置的 HTML 控件，而非使用 Canvas API 来从头实现控件。这通常是个好的建议。要是想用 Canvas API 来编写全新的控件，一般来说，会涉及大量的工作。在大多数情况下，如果有某种更为简单的方法可用，我们就应该明智一些，不要为了实现它而花费那么多功夫。

然而，在某些情况下，确实有必要去实现基于 Canvas 的控件。在本书第 10 章中，你将看到实现这些控件的动机及其实现方式。

提示：网格的绘制

本节讨论的这个应用程序在跳动的小球背后绘制了网格线，用以强调浮动的 DIV 元素确实是浮在 canvas 之上的。

在本书第 2 章中，我们将讨论绘制网格的方法，不过现在，你可以先放心地往下读，不必担心网格绘制的细节问题。

不可见的 HTML 元素

在前一小节中，读者已经知道了如何将静态的 HTML 控件与 canvas 联合起来使用。本小节我们将研究一种更为高级的 HTML 控件使用方法，也就是在用户拖动鼠标时动态地修改 DIV 元素的大小。

图 1-17 中所示的这个应用程序，采用了一种名为“橡皮筋式”（rubberbanding）选取框的技术来让用户在 canvas 之中选择某个区域。起初，该 canvas 会显示一幅图像，然后当选定图像的

某一部分时，应用程序会将你所选的这部分区域放大。

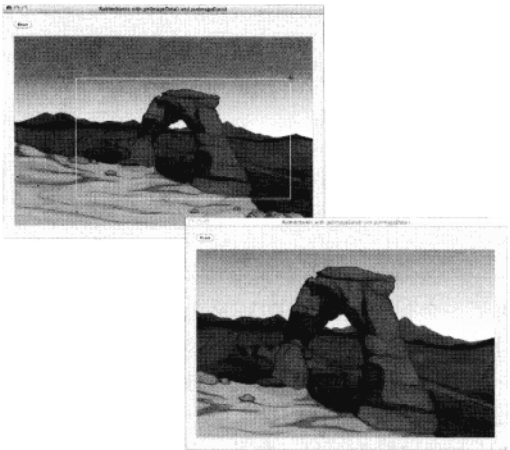


图 1-17　使用 DIV 元素来实现橡皮筋式选取框

首先，我们来看看程序清单 1-9 中列出的应用程序 HTML 代码。

程序清单 1-9　使用浮动的 DIV 元素来实现橡皮筋式选取框

```
<!DOCTYPE html>
<html>
   <head>
     <title>Rubber bands with layered elements</title>

      <style>
         body {
             background: rgba(100, 145, 250, 0.3);
         }

         #canvas {
            margin-left: 20px;
            margin-right: 0;
            margin-bottom: 20px;
            border: thin solid #aaaaaa;
            cursor: crosshair;
            padding: 0;
         }

         #controls {
            margin: 20px 0px 20px 20px;
         }

         #rubberbandDiv {
            position: absolute;
            border: 3px solid blue;
            cursor: crosshair;
            display: none;
         }

      </style>
   </head>
```

```
  <body>
      <div id='controls'>
         <input type='button' id='resetButton'value='Reset'/>
      </div>

      <div id='rubberbandDiv'></div>

      <canvas id='canvas' width='800'height='520'>
         Canvas not supported
      </canvas>

     <script src='example.js'></script>
  </body>
</html>
```

这段 HTML 代码使用了一个包含按钮的 DIV 元素。如果点击了那个按钮，那么应用程序就会像刚开始启动时那样，将整幅图像都绘制出来。

应用程序又定义了一个 DIV 元素，用于实现橡皮筋式选取框。这个 DIV 元素是空的，其 CSS 的 display 属性值被设置为 none，这样的话，它一开始就是不可见的。当用户拖动鼠标时，应用程序会将这第二个 DIV 元素设置为可见，如此一来，它的边框就会被显示出来。当你继续拖动鼠标时，应用程序就会持续地修改该 DIV 的大小，这样就可以制作出图 1-17 中那样的橡皮筋式选取框效果。

图 1-17 中应用程序的 JavaScript 代码列在了程序清单 1-10 当中。

程序清单 1-10　使用 DIV 元素来实现橡皮筋式选取框

```
  var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    rubberbandDiv =document.getElementById('rubberbandDiv'),
    resetButton =document.getElementById('resetButton'),
    image = new Image(),
    mousedown = {},
    rubberbandRectangle = {},
    dragging = false;

// Functions.......................................................

function rubberbandStart(x, y) {
   mousedown.x = x;
   mousedown.y = y;

   rubberbandRectangle.left = mousedown.x;
   rubberbandRectangle.top = mousedown.y;

   moveRubberbandDiv();
   showRubberbandDiv();

   dragging = true;
}
function rubberbandStretch(x, y) {
   rubberbandRectangle.left = x < mousedown.x ? x: mousedown.x;
   rubberbandRectangle.top  = y < mousedown.y ? y: mousedown.y;

   rubberbandRectangle.width  = Math.abs(x -mousedown.x);
   rubberbandRectangle.height = Math.abs(y -mousedown.y);

   moveRubberbandDiv();
```

```
    resizeRubberbandDiv();
}

function rubberbandEnd() {
    var bbox = canvas.getBoundingClientRect();

    try {
      context.drawImage(canvas,
                        rubberbandRectangle.left -bbox.left,
                        rubberbandRectangle.top -bbox.top,
                        rubberbandRectangle.width,
                        rubberbandRectangle.height,
                        0, 0, canvas.width,canvas.height);
    }
    catch (e) {
       // Suppress error message when mouse is released
       // outside the canvas
    }

    resetRubberbandRectangle();

    rubberbandDiv.style.width = 0;
    rubberbandDiv.style.height = 0;

    hideRubberbandDiv();

    dragging = false;
}

function moveRubberbandDiv() {
    rubberbandDiv.style.top  =rubberbandRectangle.top  + 'px';
    rubberbandDiv.style.left =rubberbandRectangle.left + 'px';
}

function resizeRubberbandDiv() {
    rubberbandDiv.style.width  =rubberbandRectangle.width  + 'px';
    rubberbandDiv.style.height =rubberbandRectangle.height + 'px';
}

function showRubberbandDiv() {
    rubberbandDiv.style.display = 'inline';
}

function hideRubberbandDiv() {
    rubberbandDiv.style.display = 'none';
}

function resetRubberbandRectangle() {
    rubberbandRectangle = { top: 0, left: 0, width:0, height: 0 };
}

// Event handlers.................................................

canvas.onmousedown = function (e) {
    var x = e.clientX,
        y = e.clientY;

    e.preventDefault();
    rubberbandStart(x, y);
};

window.onmousemove = function (e) {
```

```
    var x = e.clientX,
        y = e.clientY;

     e.preventDefault();
     if (dragging) {
        rubberbandStretch(x, y);
     }
};

window.onmouseup = function (e) {
   e.preventDefault();
   rubberbandEnd();
};

image.onload = function () {
   context.drawImage(image, 0, 0, canvas.width,canvas.height);
};

resetButton.onclick = function(e) {
   context.clearRect(0, 0, context.canvas.width,
                           context.canvas.height);
   context.drawImage(image, 0, 0, canvas.width,canvas.height);
};

//Initialization.................................................

image.src = 'curved-road.png';
```

在这段代码中，我们又一次超前地使用了 drawImage() 方法同时对图像进行了绘制与缩放。本书 4.1 节将会详细研究这个方法，同时我们还会学习另外一种实现该效果的办法，那就是通过直接修改图像的像素来绘制橡皮筋式选取框。

然而，现在我们要关注的还是用于实现橡皮筋式选取框的 DIV 元素，以及这段代码是如何在用户拖动鼠标时修改该元素大小的。

canvas 的 onmousedown 事件处理器在被触发时会调用 rubberbandStart() 方法，该方法会将 DIV 元素的左上角移动到鼠标按下的地点，并使 DIV 元素可见。由于代表橡皮筋式选取框的这个 DIV 元素，其 CSS 中的 position 值是 absolute，所以在指定其左上角坐标时，必须指定相对于窗口的坐标，而不是相对于 canvas 的坐标。

如果用户拖动鼠标，那么 onmousemove 事件处理器就会调用 rubberbandStretch() 方法，该方法会对代表橡皮筋式选取框的这个 DIV 元素进行移动与缩放操作。

当用户松开鼠标的时候，onmouseup 事件处理器就会调用 rubberbandEnd() 方法，该方法会把选中的那部分图像放大，并绘制出来，同时将表示橡皮筋式选取框的那个 DIV 元素隐藏起来。

最后，请注意，以上三个鼠标事件处理器都会在传入的事件对象上调用 preventDefault() 方法。就像在 1.6.1 小节中讨论的那样，调用该方法可以防止浏览器对鼠标事件做出响应。如果你将 preventDefault() 方法的调用移除，那么浏览器就会试着选中网页上的元素，要是用户在拖动时将鼠标移出了 canvas 的范围，那么就会产生不好的效果。

1.9 打印 Canvas 的内容

如果可以将 canvas 作为图像来访问，那么这对于应用程序的用户来说，通常会是件很方便的事情。比如说，如果你实现了一个像本书第 2 章中所讨论的那种绘图程序，那么用户会要求能够

将他们所绘制的内容打印出来。

在默认情况下，尽管每个 canvas 对象其实都是一幅位图，但是，它并不是 HTML 的 img 元素，所以，用户不能对其进行某些操作。比如，不能在 canvas 上通过右击鼠标然后将其保存到磁盘，也不能将其拖动到桌面上，稍后再打印出来。图 1-18 之中所显示的这个弹出式菜单，就证明了 canvas 并不是图像。

还好，Canvas 的 API 提供了一个叫做 toDataURL( ) 的方法，该方法所返回的引用，指向了某个给定 canvas 元素的数据地址。接下来，你可以将 img 元素的 src 属性值设置为这个数据地址，这样的话，就可以创建一幅表示 canvas 的图像了。

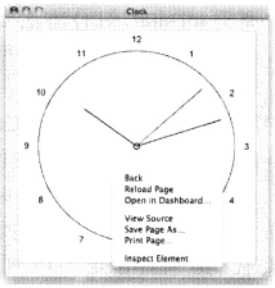


图 1-18　在 canvas 上右击鼠标后所弹出的菜单

在本书 1.5 节中，你已经看到了如何用 Canvas 的 API 来实现一个模拟时钟程序。图 1-19 之中所示的应用程序是经过修改的版本，它可以让用户做一份时钟程序的快照，并按照上面所描述的那种办法将快照以图像的方式显示出来。正如你在图 1-19 中看到的那样，可以在抓取快照之后的图像上右击鼠标，将其保存到磁盘中。因为图片下方所显示的那个时钟图像是一个 img 元素，所以也可以直接将其拖放到操作系统的桌面上。

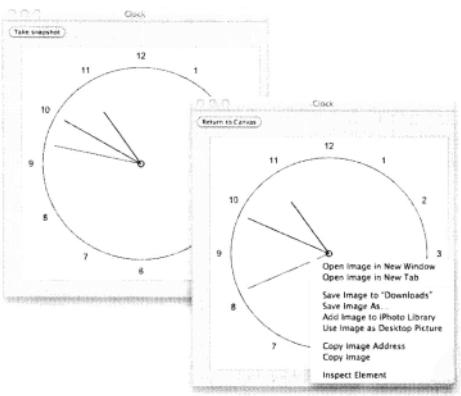


图 1-19　使用 toDataURL() 方法来保存 Canvas 的图像

图 1-19 之中的应用程序所实现的这种打印 canvas 内容的方式，是很常见的：它提供了一个控件，让用户通过该控件来抓取 canvas 的快照。在本应用程序中，这个控件就是那个名为“Take snapshot”的按钮。应用程序将抓取的快照显示为一幅图像，以便用户可以在其上右击鼠标来将此图像保存到磁盘中。接下来，当用户点击“Return to Canvas”按钮时，应用程序就会用原来的 canvas 元素来替换当前窗口中所显示的图像。这种实现方式的步骤如下：

在 HTML 页面的代码中：

- 向网页中加入一个不可见的图像元素，然后给该元素设置好 id 值，但是不要设定其 src 属性。
- 通过 CSS，调整图像的位置与大小，使其刚好覆盖在 canvas 之上。

• 向网页中加入一个用于抓取快照的控件。

在 JavaScript 代码中：

• 获取指向刚才那个不可见图像元素的引用。

• 获取指向快照抓取控件的引用。

• 当用户激活控件以抓取快照时：

（1）调用 toDataURL() 方法来获取数据地址。

（2）将数据地址设定为不可见图像元素的 src 属性值。

（3）将图像元素设置为可见，将 canvas 设置为不可见。

• 当用户激活控件以返回到 Canvas 时：

（1）使 canvas 元素可见，使图像元素不可见。

（2）如有必要，则重绘 canvas。

现在我们来看看如何将上述步骤转化为代码。程序清单 1-11 列出了图 1-19 中那个应用程序的 HTML 代码。程序清单 1-12 列出了该应用程序的 JavaScript 代码。

程序清单 1-11　使用 toDataURL() 方法将 canvas 的内容打印出来（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Clock</title>

     <style>
        body {
           background: #dddddd;
        }

        #canvas {
           position: absolute;
           left: 10px;
           top: 1.5em;
           margin: 20px;
           border: thin solid #aaaaaa;
        }

        #snapshotImageElement {
           position: absolute;
           left: 10px;
           top: 1.5em;
           margin: 20px;
           border: thin solid #aaaaaa;
        }
     </style>
</head>

<body>
    <div id='controls'>
      <input id='snapshotButton' type='button'value='Take snapshot'/>
    </div>

    <img id='snapshotImageElement'/>

    <canvas id='canvas' width='400' height='400'>
      Canvas not supported
    </canvas>

    <script src='example.js'></script>
  </body>
</html>
```

程序清单 1-12　使用 toDataURL() 方法将 canvas 的内容打印出来（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    snapshotButton =document.getElementById('snapshotButton'),
    snapshotImageElement =
       document.getElementById('snapshotImageElement'),
    loop;

// Clock drawing functions are omitted from this listing
// in the interests of brevity. See Example 1.4 ⊖
// for a complete listing of those methods.

// Event handlers.....................................................

snapshotButton.onclick = function (e) {
   var dataUrl;

   if (snapshotButton.value === 'Take snapshot') {
      dataUrl = canvas.toDataURL();
      clearInterval(loop);
      snapshotImageElement.src = dataUrl;
      snapshotImageElement.style.display ='inline';
      canvas.style.display = 'none';
      snapshotButton.value = 'Return to Canvas';
   }
   else {
      canvas.style.display = 'inline';
      snapshotImageElement.style.display = 'none';
      loop = setInterval(drawClock, 1000);
      snapshotButton.value = 'Take snapshot';
   }
};

//Initialization..............................................

context.font = FONT_HEIGHT + 'px Arial';
loop = setInterval(drawClock, 1000);
```

该应用程序的代码访问 canvas 及 img 元素，并且通过 CSS 的绝对定位方式使其互相重叠。当用户点击“Take snapshot”按钮时，应用程序从 canvas 之中获取数据地址，并将其值赋给图像元素的 src 属性。然后，显示图像并隐藏 canvas，同时将按钮文本设置为“Return to Canvas”。

当用户点击“Return to Canvas”按钮时，应用程序隐藏图像并显示 canvas，同时将按钮文本改回原来的“Take snapshot”。

提示：将 Canvas 输出至 blob 文件

在本书写作时，Canvas 规范中新增了一个 toBlob() 方法，所以说，还可以将 canvas 的内容保存至一个 blob 文件[⊜]。直到本书出版时，还未有浏览器支持这个方法。

1.10　离屏 canvas

Canvas 技术的另一项重要功能就是可以创建并操作离屏 canvas[⊜]。举例来说，在大多数情

⊖ 即程序清单1-4，后面的代码注释中也有相似的情况，不再一一注明。——编辑注

⊜ blob是一种用于存放原始二进制数据的文件形式，详情请参阅：http://dev.w3.org/2006/webapi/FileAPI/#blob。——译者注

⊜ offscreen canvas，也叫缓冲canvas、幕后canvas。——译者注

况下，都可以把背景存储在一个或多个离屏 canvas 之中，并将这些离屏 canvas 中的某一部分复制到屏幕上，以此来大幅提高应用程序的性能。

离屏 canvas 的另一个用途就是制作我们在上一节中提到的那个钟表程序。那个应用程序向你展示了一种通用的解决方案，它需要用户通过交互界面将程序的显示方式从 canvas 切换为图像。不过，像时钟这样的应用程序，最好还是能在不需要用户干预的情况下，于幕后完成显示模式的切换。

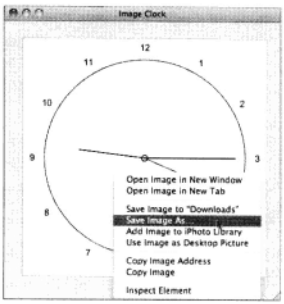


图 1-20　使用离屏 canvas 来绘制时钟图像

图 1-20 之中所显示的是更新之后的时钟程序。应用程序每秒钟都会将时钟绘制到离屏 canvas 之中，然后将该 canvas 的数据地址设置为图形元素的 src 属性值，于是就形成了一幅反映离屏 canvas 内容变化的动画图像。有关 canvas 数据地址的更多信息，请参阅本书 1.9 节。

图 1-20 这个应用程序的 HTML 代码列在了程序清单 1-13 之中。

程序清单 1-13　以图像方式实现的时钟程序（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Image Clock</title>

     <style>
        body {
           background: #dddddd;
        }

        #canvas {
           display: none;
        }

        #snapshotImageElement {
           position: absolute;
           left: 10px;
           margin: 20px;
           border: thin solid #aaaaaa;
        }
     </style>
  </head>

  <body>
     <img id='snapshotImageElement'/>

     <canvas id='canvas' width='400' height='400'>
        Canvas not supported
     </canvas>

     <script src='example.js'></script>
  </body>
</html>
```

请注意 HTML 代码中 canvas 元素的 CSS 设定：canvas 是不可见的，因为其 display 属性被设置为 none 了。这样的话，这个不可见的 canvas 就变成一个离屏 canvas 了。也可以使用编程的方式来创建离屏 canvas，像是这样：

```
var offscreen = document.createElement('canvas');.
```

图 1-20 这个应用程序里面与离屏 canvas 有关的 JavaScript 代码，列在了程序清单 1-14 之中。

程序清单 1-14　以图像方式实现的时钟程序（JavaScript 代码，节选）

```
// Some declarations and functions omitted for brevity.
// See Section 1.9⊖ for a complete listing of
// the clock.

var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

// Functions.........................................................

function updateClockImage() {
   snapshotImageElement.src = canvas.toDataURL();
}

function drawClock() {
   context.clearRect(0, 0, canvas.width,canvas.height);

   context.save();

   context.fillStyle = 'rgba(255,255,255,0.8)';
   context.fillRect(0, 0, canvas.width,canvas.height);

   drawCircle();
   drawCenter();
   drawHands();

   context.restore();

   drawNumerals();

   updateClockImage();
}
...
```

1.11　基础数学知识简介

如果想用 Canvas 实现一些有趣的功能，那么必须得很好地了解一些基本的数学知识，尤其是代数方程、三角函数及向量运算。如果要编写类似电子游戏那样更为复杂的应用程序，那么还需要掌握如何根据给定的计量单位来推导等式。

若是读者已经能够非常流畅地应对基本的代数运算与三角函数，并且能够根据给定的“每秒钟位移多少个像素”以及“每帧持续多少毫秒”来推导出“每帧的位移量”来，那么你略读一下本节就好了。不然的话，还是花些时间来研究一下这节的内容吧，它对于阅读本书的其余部分有很大帮助。

我们先来讲讲代数方程的求解与三角函数，然后再来学习向量运算与如何根据计量单位推导等式。

1.11.1　求解代数方程

对于任意的代数方程，比如 $(10x+5)\times 2=110$，都可以进行如下操作，使等式仍然成立：

⊖ 即1.9节。——编辑注

- 给等式两端同时加上任意一个实数。
- 从等式两端同时减去任意一个实数。
- 给等式两端同时乘以任意一个实数。
- 让等式两端同时除以任意一个实数。
- 给等式的一端或两端同时乘以或除以 1。

举例来说，(10x+5)×2=110 这个方程式，可以这么来求解：先给等式两端同时除以 2，得到 10x+5=55，然后在等式两端同时减去 5，得到 10x=50，最后，让等式两端同时除以 10，于是就求得 x=5。

上述规则中的最后一条看上去很奇怪。为什么要给等式的一端或两端同时乘以或除以 1 呢？在 1.11.4 小节之中，将会学习如何根据计量单位来推导等式，那时我们就会看到如何恰当地运用这条简单的规则。

1.11.2 三角函数

即便是最简单地使用 Canvas，那也需要对三角函数有一个基本的了解才行。举例来说，本书下一章将会讲解如何绘制多边形，而这就需要你对正弦与余弦函数有一定的理解。我们先来简单地讲讲“角”的度量，然后再谈一下直角三角形。

1.11.2.1 角的度量：角度与弧度

Canvas 中所有与角有关的 API，都需要以弧度（radian）的方式来指定该角的值。JavaScript 的 Math.sin()、Math.cos() 与 Math.tan() 函数也都采用弧度值。不过大部分人还是觉得角就是以角度（degree）来度量的，所以你得知道如何将角度转换为弧度。

180 度等于 π 弧度。要想将角度转化为弧度，你可以先根据此关系建立像等式 1.1 这样的代数式：

$$180\text{ 度} = \pi\text{ 弧度}$$

等式 1.1 角度与弧度的关系

根据等式 1.1 可以推导出 1 角度等于多少个弧度，以及 1 弧度等于多少个角度。这两个推导等式分别记为等式 1.2 与等式 1.3。

$$1\text{ 弧度} = (\pi / 180) \times \text{度}$$

等式 1.2 将角度化为弧度所用的等式

$$1\text{ 度} = (180 / \pi) \times \text{弧度}$$

等式 1.3 将弧度化为角度所用的等式

π 约等于 3.14，所以，45 度等于 (3.14/180)×45 弧度，经过运算可以得知，等于 0.7853 弧度。

1.11.2.2 正弦、余弦与正切函数

为了有效地利用 Canvas，你必须对正弦（sine，简称 sin）、余弦（cosine，简称 cos）及正切（tangent，简称 tan）函数有一定的了解，所以说，如果你还不熟悉图 1-21 中所述的内容，那么现在必须把它记住才行。

你也可以像图 1-22 中所画的这样，用圆周上某一点的 X 坐标与 Y 坐标来理解正弦及余弦函数。

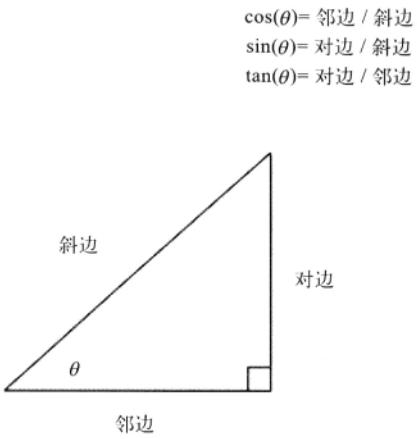


图 1-21　正弦、余弦与正切函数的定义

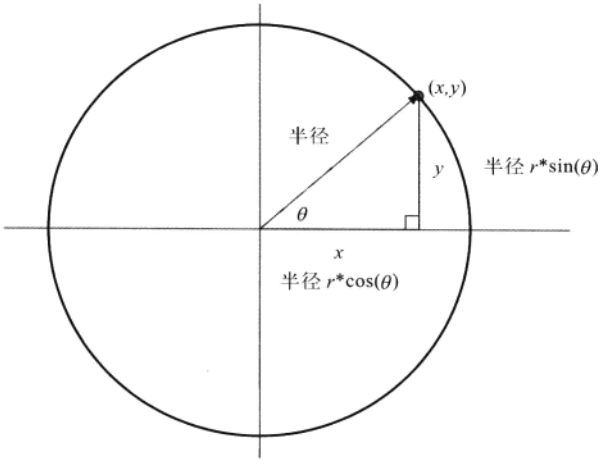


图 1-22　以半径、x 坐标及 y 坐标来理解正弦及余弦函数

给定某个圆的半径，以及一个从 0 度开始的逆时针角，我们就可以分别计算出圆周上某一点的 x 与 y 坐标了。其 x 坐标等于圆半径乘以该角的余弦值，y 坐标等于圆半径乘以该角的正弦值。

提示：用“Soak a toe, ah!”的口诀记住三角函数

有很多种方式可以帮你回忆起如何从直角三角形中推导出正弦、余弦与正切函数的定义来。其中一个办法就是记住“SOHCAHTOA”这个口诀[⊖]。SOH 代表正弦（sine）、对边（opposite）及斜边（hypotenuse）；CAH 表示余弦（cosine）、邻边（adjacent）及斜边（hypotenuse）；TOA 代表正切（tangent）、对边（opposite）及邻边（adjacent）。

1.11.3　向量运算

本书中所用的二维向量（two-dimensional vector）都含有两个值：方向（direction）及大小（magnitude）。这两个值可以表达出各种各样的物理特性来，比如力（force）和运动（motion）。

在本书第 8 章之中，将会频繁地用到向量，所以本小节我们先来讨论一下有关向量数学运算的基础知识。如果你对碰撞检测的实现不感兴趣的话，那么就可以放心地跳过这一小节了。

在第 8 章的末尾，我们将会研究在两个多边形发生碰撞时，如何使其中一个多边形被另一个多边形弹开。该碰撞效果如图 1-23 所示。

在图 1-23 之中，位于顶部的多边形正在向位于底部的多边形前进，这两个多边形即将发生碰撞。顶部那个多边形的碰撞之前的前进速度（incoming velocity）与碰撞之后被弹走的速度（outgoing velocity）都是以向量来建模的。在底部那个多边形中，即将与顶部多边形发生碰撞的那条边，也以向量来建模，我们称之为“边向量”（edge vector）。

如果读者迫不及待地想要知道如何根据给定的前进速度与碰撞边界上的两个点，来计算出碰撞之后被弹走的速度，那么请跳至第 8 章阅读。然而，如果你还不太熟悉基本的向量运算，那么请在翻到第 8 章之前先把本小节的内容读完。

⊖ 这几个字母的发音与“Soak a toe, ah!”类似，后者的意思是“泡泡脚趾吧，（真舒服）啊！”——译者注

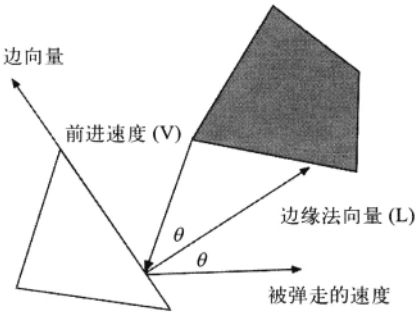


图 1-23 通过向量来计算多边形在发生碰撞时被弹走的速度

1.11.3.1 向量的大小

虽说二维向量是对大小和方向这两个数值进行建模的，不过通常情况下，根据某个给定向量的 x 与 y 值来计算出这两个数值其中的一个，也是很有用的。你可以使用在学校数学课上学过的（或者，从《绿野仙踪》这部电影中看到的）毕达哥拉斯定理（Pythagorean theorem，又称勾股定理）来计算出某个向量的大小，如图 1-24 所示。

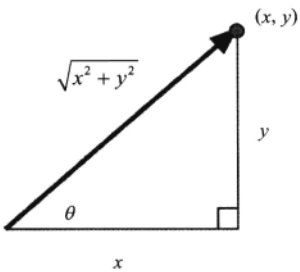


图 1-24 计算向量的大小

毕达哥拉斯定理说：任何直角三角形的斜边，等于另外两边平方和的平方根。结合图 1-24，就能更好地理解该定理了。与该定理相对应的 JavaScript 代码如下：

```
var vectorMagnitude =Math.sqrt(Math.pow(vector.x, 2) +
                               Math.pow(vector.y,2));
```

上述 JavaScript 代码片段可以从一个名为 vector 的向量引用中计算出该向量的大小。

讲完了如何计算某个向量的大小之后，我们来看看如何计算向量的另一个值：方向。

1.11.3.2 单位向量

向量运算经常会用到一个名叫“单位向量”（unit vector）的东西。单位向量是像图 1-25 所示的只用来指示方向的向量。

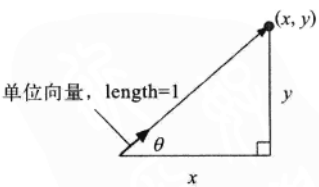


图 1-25 单位向量

之所以叫它单位向量，是因为其长度永远是“单位 1”（1 unit）。如果要根据给定的向量来计算其单位向量，需要先把原来向量的大小去掉，只留下方向即可。在 JavaScript 代码中，可以这么来做：

```
var vectorMagnitude =Math.sqrt(Math.pow(vector.x, 2) +
                               Math.pow(vector.y,2)),
    unitVector = new Vector();

    unitVector.x = vector.x / vectorMagnitude;
    unitVector.y = vector.y / vectorMagnitude;
```

上述程序首先根据一个叫做 vector 的向量，沿用本小节前面讲过的办法，来计算出它的大小。然后，创建了一个新的 vector 对象，并将原有向量的 x 与 y 值分别除以原向量的大小，作为

这个新单位向量的 x 与 y 值。

读者已经学习了如何计算二维向量的两个分量，现在来看看如何对两个向量进行联合运算。

1.11.3.3 向量的加法与减法

向量的加减法也是很有用的运算。举例来说，如果有两个力同时作用于某个物体，你可以把代表这两个力的向量相加，计算出这两个力的合力。同样，也可以从一个方位向量[⊖]之中减去另一个，以求得两点之间的边界。

图 1-26 演示了如何根据两个给定的向量 A、B，来进行向量加法运算。

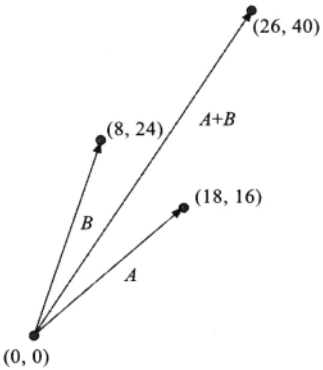


图 1-26 两个向量的加法运算

向量的加法运算很简单，像下面列出的程序清单这样，将向量的两个分量分别相加即可：

```
var vectorSum = new Vector();

vectorSum.x = vectorOne.x + vectorTwo.x;
vectorSum.y = vectorOne.y + vectorTwo.y;
```

向量的减法同样也很简单，像下面所列程序清单一样，将向量的两个分量分别相减即可：

```
var vectorSubtraction = new Vector();

vectorSubtraction.x = vectorOne.x -vectorTwo.x;
vectorSubtraction.y = vectorOne.y -vectorTwo.y;
```

图 1-27 演示了将一个向量从另一个之中减去所得到的第三个向量，其方向恰好与两向量终点之间的边界方向是一致的。在图 1-27 之中，向量 A-B 与 B-A 互相平行，且两者都平行于向量 A、B 终点所构成的“边向量”。

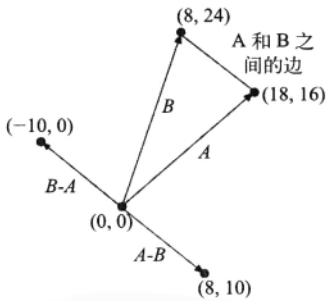


图 1-27 两个向量的减法运算

读者已经学会了如何进行向量的加减法，更为重要的是，了解到加减法所表示的几何意义，既然如此，我们再来看看向量的另一个数值：点积[⊜]。

1.11.3.4 两个向量的点积

要计算两个向量的点积，需要将两个向量的对应分量相乘，然后再将乘积相加。下面这段代码可以计算出两个二维向量的点积：

```
var dotProduct = vectorOne.x * vectorTwo.x +vectorOne.
y * vectorTwo.y;
```

计算两个向量之间的点积是很简单的，不过，这个点积的意义理解起来可就有些不太直观了。首先请注意，与两个向量的加减法运算结果不同，点积不是向量。专业人员把这种值叫做“标量”（scalar，也叫“纯量”），就是说，它仅仅是个数字而已。为了理解这个数字的意义，咱们研究一下图 1-28。

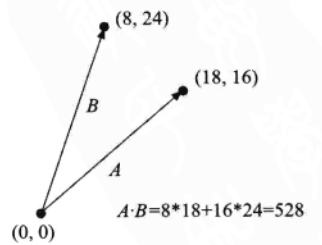


图 1-28 两向量点积为正值的情况

图 1-28 中所示的两个向量，其点积为 528。该数值的重要

⊖ positional vector，又叫position vector，详情参阅：http://en.wikipedia.org/wiki/Position_vector。——译者注

⊜ dot product，又叫数量积、纯量积、点乘。详情参阅：http://zh.wikipedia.org/wiki/数量积。——译者注

性并不在于其大小，而在与其“大于 0”这个事实。这意味着两个向量大概处在同一方向上。

再来看看图 1-29，该图中的两个向量，其点积为 −528。因为该值小于 0，所以我们可以推测出，这两个向量所指的方向大概不太一样[⊖]。

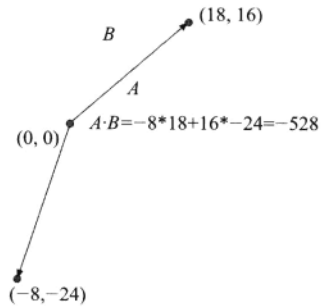


图 1-29 两向量点积为负值的情况

判断两个矢量的终点是不是大致指向同一个方向，对于响应物体之间的碰撞来说，是一项很关键的技术。当某个运动的物体同某个静止的物体发生碰撞时，如果我们需要让运动的物体被静止物体弹开，那么就必须确保这个运动的物体在碰撞之后朝着远离静止物体的方向运动，而不是朝着静止物体的中心运动。通过计算两个向量的点积，我们可以精确地做到这一点，本书第 8 章将会讲解该问题。

关于检测碰撞所用的向量知识，咱们就讲这么多，接下来，我们看看基础数学知识的最后一个小节：根据计量单位来推导等式。

1.11.4 根据计量单位来推导等式

读者在第 5 章中将会看到，动画的移动应该是以时间为基准的。因为一个物体的移动频率不应该随着动画的帧速率而改变。基于时间的运动（time-based motion）对于多人游戏来说尤为重要：你肯定不希望使用高配置电脑的玩家比别人移动地更快吧。

为了实现基于时间的运动效果，本书中我们采用“每秒移动的像素数”（pixels per second）作为计量移动速度的单位。因此，为了计算动画当前帧所移动的像素数，我们需要知道两个信息：物体的移动速度是每秒多少个像素，以及当前动画的帧速率（frame rate）是每帧持续多少毫秒（milliseconds per frame）。我们所要计算的是：给定一个物体，求出它每帧所要移动的像素数（pixels per frame）。为了求得该值，我们得推导出一个等式来。等式的左端是每帧移动的像素数，右端则要包含每秒钟移动的像素数（也就是物体的速度）与每帧持续的毫秒数（也就是当前的帧速率），如等式 1.4 所示。

$$\frac{\text{像素}}{\text{帧}} \neq \frac{\text{X 毫秒}}{\text{毫秒}} \times \frac{\text{Y 毫秒}}{\text{秒}}$$

等式 1.4 推导基于时间的运动所用的计量单位，第 1 部分

在上述“不等式（inequality）”之中[⊜]，X 表示以“每帧持续毫秒数”为计量单位的动画帧速率，而 Y 代表以“每秒移动的像素数”为计量单位的物体运动速度。 正如该不等式所述，你不能简单地将“每帧持续毫秒数”与“每秒移动的像素数”相乘，因为如果那样做的话，将会得到“（毫秒 × 像素）/（帧数 × 秒）”这样没有意义的运算结果。那么，应该怎么算呢？

回想一下 1.11.1 小节中讲述求解代数方程时所说的最后一条规则：你可以在等式的一端或两

⊖ 如果两个向量的点积是0，则其必定互相垂直。——译者注

⊜ 其实等式1.4也是成立的，只是计量单位没有统一，作者为了强调这一点，告诉我们不能直接将数字相乘，所以称其为“不等式”，特此说明。——译者注

端同时乘以或除以 1。已知 1 秒等于 1000 毫秒，所以“1 秒 /1000 毫秒”等于 1，根据该规则，我们可以给等式的右端乘以这个分数，于是得到了等式 1.5。

$$\frac{像素}{帧} = \frac{X\ 毫秒}{帧} \times \frac{1\ 秒}{1000\ 毫秒} \times \frac{Y\ 像素}{秒}$$

等式 1.5　推导基于时间的运动所用的计量单位，第 2 部分

现在，我们可以来化简这个等式了。因为如果将两个分数相乘的话，那么其中一个分数分子中所含的计量单位就可以与另外一个分数分母中所含的相同单位互相消去。在本例中，我们按照等式 1.6 所述的方式进行化简。

$$\frac{像素}{帧} = \frac{X\ 毫秒}{帧} \times \frac{1\ 秒}{1000\ 毫秒} \times \frac{Y\ 像素}{秒}$$

等式 1.6　推导基于时间的运动所用的计量单位，第 3 部分

消去这些计量单位之后，我们得到了等式 1.7。

$$\frac{像素}{帧} = \frac{X}{帧} \times \frac{Y\ 像素}{1000}$$

等式 1.7　推导基于时间的运动所用的计量单位，第 4 部分

将等式右端两个相乘的分数以简化形式写出来，就可以得到等式 1.8。

$$\frac{像素}{帧} = \frac{X \times Y}{1000}$$

X= 用毫秒 / 帧表示的帧速率

Y= 用像素 / 秒表示的速度

等式 1.8　推导基于时间的运动所用的计量单位，第 5 部分

当推导出一个等式时，应该向等式中带入一些简单的数值来检验一下，看它是否成立。在本例中，如果某个对象的移动速度是每秒 100 像素，且其帧速率是每 500 毫秒变换 1 帧，那么，不需要借助等式，即可看出，该物体在 1/2 秒内的移动距离是 50 个像素。

将以上数字带入到等式 1.8 之中，就是 500×100/1000，等于 50，所以看起来我们所推导的这个等式的确可以根据移动速度与帧速率正确地计算出每帧移动的像素数。

通常来说，如果要根据某些已知其计量单位的变量来推导换算等式的话，应该遵循如下步骤：

（1）以一个不等式开始，将要推的结果放在左端，将其余变量放在右端。

（2）根据等式两端的计量单位，给等式的右端乘以一个或多个分数。这些分数的值都要是 1，它们所采取的计量单位要能够消去原来等式右端的计量单位，使其符合等式左端的单位。

（3）将等式右端相同的计量单位消去。

（4）将等式右端的各个分数相乘，合并为一个分数。

（5）带入一些很容易被验证的简单数值进去，以确保推导好的等式能够算出期望的结果。

1.12 总结

本章向读者介绍了 canvas 元素及其相关的 2d 绘图环境，并且演示了该绘图环境的一些重要特性，例如 canvas 元素大小与 canvas 绘图表面大小的区别。

讲完了上述内容之后，我们快速地概述了包括浏览器、控制台、调试器及性能测试工具在内的开发环境。

然后，我们讲述了 canvas 的一些重要用法，包括基本的绘图操作、事件处理、绘图表面的保存与恢复、canvas 与 HTML 元素的结合使用、canvas 内容的打印以及离屏 canvas 等。读者将在本书中多次看到对这些关键功能的运用，而且在自己编写基于 Canvas 的应用程序时，也会用到它们。

最后，我们在本章末尾讲述了一些基础的数学知识，可以在阅读本书其余内容时参考它们。

在下一章中，我们将深入地讲解 canvas 的绘制。在那一章中，读者将学到 Canvas 的绘制 API，并且会看到如何合理地运用这些 API，来将一个强大画图程序的绝大部分功能都实现出来。

第2章 绘　　制

在 HTML5 技术中，Canvas 的 2d 绘图环境提供了一套功能强大的绘图 API，可以用它来实现一些能够在浏览器中运行的、精致而且引人注目的图形应用程序。

图 2-1 中所示的这个画图程序，可以绘制文本、线条、矩形、圆形及贝塞尔曲线（bézier curves），也可以根据鼠标的移动来绘制任意的路径。要擦除内容，可以先选择工具栏最下方的那个图标，然后在绘制区域拖动鼠标就可以了（第 2.15.1 小节会讲到）。通过页面顶部的 HTML 控件可以修改绘制属性，也可以抓取应用程序的快照，以便让用户将所画的内容保存成图片。

这个画图应用程序采用橡皮筋技术来完成交互式绘制：当用户通过拖动鼠标来创建新的图形，比如圆形或矩形时，应用程序会持续地在即将被创建的图形外围绘制轮廓线。当用户松开鼠标后，应用程序就将当前的形状确定并绘制出来，同时有可能还会对其进行填充。如果用户在选择表示某种图形的按钮时，点击的是该按钮的右下部分，那么应用程序则会对创建出来的图形进行填充。[⊖]

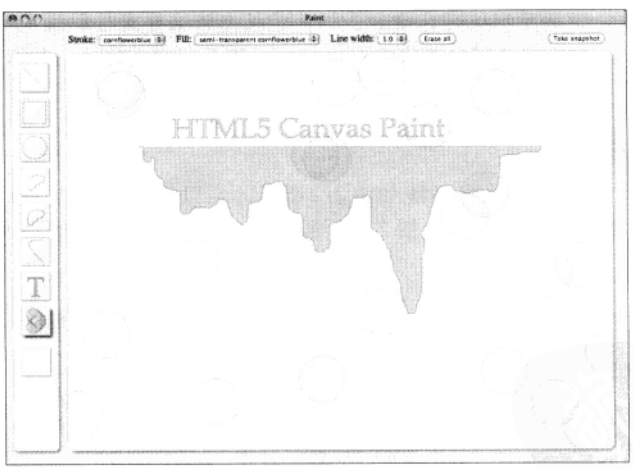


图 2-1　画图应用程序

本章会以这个绘图程序以及其他一些范例来向读者讲述 Canvas API 所具备的一切功能。将会学到的技能包括：

- 对线条、弧形、圆、曲线及多边形进行描边与填充。
- 通过设置绘图环境的属性来改变所绘图形的外观。
- 绘制圆角矩形。

⊖ 按钮之中有一条对角线，将其分成了两个部分，左上部分具有浅色背景，右下部分则具有深色背景。——译者注

- 绘制并编辑贝塞尔曲线。
- 对 2d 绘制环境进行扩展，使之可以绘制虚线。
- 使用纯色、渐变色及图案来对图形进行描边及填充。
- 用阴影效果来模拟具有深度的立体图形效果。
- 在不影响背景的情况下，使用“剪辑区域”技术来擦除图形与文本。
- 实现橡皮筋式辅助线技术，以便让用户可以交互式地绘制图形。
- 在 canvas 中拖动图形对象。
- 坐标系统的变换。

提示：画图应用程序

图 2-1 中所示的画图应用程序，是用了大约 1100 行 JavaScript 代码来实现的。本书不可能列出如此长的程序清单来，不过，你可以在 corehtml5canvas.com 网站下载本应用程序的代码。

尽管本书并没有完整地列出这个画图应用程序的代码，不过它的大部分功能在本章中都会以一些稍小的应用程序来演示。这些小程序，以 2.8.4 小节讲述的内容开始，到 2.15.1 小节讲述的内容结束。

提示：为移动设备开发画图应用程序

本章讲述的是如何在带有鼠标的设备上实现画图应用程序。在第 11 章中，你将会看到如何对本应用程序进行修改，使其可以运行在诸如手机及平板电脑这样支持触摸事件而非鼠标事件的设备之上。在那一章中，你还将学到如何修改应用程序，使其可以更好地运行在移动设备之上。

2.1 坐标系统

在默认情况下，Canvas 的坐标系统如图 2-2 所示，它以 canvas 的左上角为原点，X 坐标向右方增长，而 Y 坐标则向下方延伸。图 2-2 中所画的，是一个具有默认 300×150 像素大小的 canvas 坐标系。

图 2-2 Canvas 的坐标系统（默认大小）

然而，Canvas 的坐标系并不是固定的。正如图 2-3 中所示，可以对坐标系统进行平移及旋转。事实上，可以采用如下方式来变换坐标系统：

- 平移（translate）

- 旋转（rotate）
- 缩放（scale）
- 创建自定义的变换方式，例如切变[⊖]

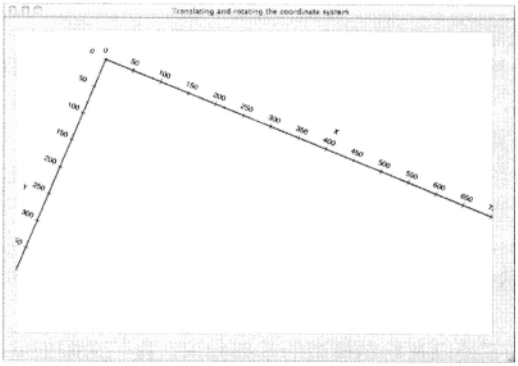


图 2-3　对坐标系统进行平移及旋转操作

在本章及本书的其余部分，读者将会看到：坐标系统的变换是 Canvas 之中一项非常基本的功能，它在很多不同的场合之中都很有用。举例来说，对坐标系统进行变换——也就是移动其原点——可以极大地简化在对图形及文本进行绘制与填充操作时所需的例行数值计算。

本书的 2.13 小节将会详细地讲解坐标系统的变换。现在，我们先来看一下 Canvas 的绘制模型，同时也看看如何利用该模型来绘制一些简单的图形与文本。

2.2　Canvas 的绘制模型

有时候要使用 Canvas，你得对它有一个很好的理解才行，这包括了 Canvas 究竟是如何绘制图形、图像与文本的。而想要了解这部分内容，则需要理解阴影、alpha 通道、剪辑区域及图像合成等内容。实际上，学完本章的内容以后，就可以很好地掌握这些知识了。现在，并不需要理解它们，所以在第一次阅读本书时可以跳过这一节，稍后再把本节内容作为参考资料来看。

在向 canvas 之上绘制图形或图像时，浏览器要按照如下步骤来操作：

（1）将图形或图像绘制到一个无限大的透明位图中，在绘制时遵从当前的填充模式、描边模式以及线条样式。

（2）将图形或图像的阴影绘制到另外一幅位图中，在绘制时使用当前绘图环境的阴影设定。

（3）将阴影中每一个像素的 alpha 分量乘以绘图环境对象的 globalAlpha 属性值。

（4）将绘有阴影的位图与经过剪辑区域剪切过的 canvas 进行图像合成。在操作时使用当前的合成模式参数。

（5）将图形或图像的每一个像素颜色分量，乘以绘图环境对象的 globalAlpha 属性值。

（6）将绘有图形或图像的位图，合成到当前经过剪辑区域剪切过的 canvas 位图之上，在操作

⊖ shear，也叫“错切”，在数学中，它是一种特殊类型的线性变换。详情请参阅：http://zh.wikipedia.org/zh-cn/错切。——译者注

时使用当前的合成操作符（composition operator）。

只有在启用阴影效果时才会执行第 2 ～ 4 步。

浏览器起初会将图形或图像绘制到一张无限大的位图上，在绘制时会使用 Canvas 绘图环境对象中与图形的填充及描边有关的那些属性。当然，实际上并没有这种所谓“无限大的位图”，不过浏览器在进行操作时是假设有它存在的。

接下来，浏览器将会按照上述的步骤 2 至步骤 4 来处理阴影。如果你像本书 2.6 节中讨论的那样，启用了阴影效果的话，那么浏览器将会把阴影渲染到另外一张位图上。并将阴影中每个像素的 alpha 值乘以 globalAlpha 属性，把运算结果设置为该阴影像素的透明度，并将阴影与 canvas 元素进行图像合成。操作时采用当前的合成设定，并按照当前的剪辑区域对合成之后的位图进行剪切。

最后，浏览器会根据当前的合成设定与剪辑区域，将图形或位图与 canvas 元素进行图像合成。

如果你刚接触 Canvas，在读完刚才这几段之后不是太理解，也别灰心。等到将来你很好地理解了阴影、alpha 通道、剪辑区域以及图像合成之后，再回过头来读本节，你就会理解刚说的那些内容了。

既然已经了解了坐标系统以及绘制模型的基础知识，接下来咱们就言归正传，开始学习如何使用 Canvas 来绘制简单的图形与文本。

2.3 矩形的绘制

Canvas 的 API 提供了如下三个方法，分别用于矩形的清除、描边及填充：

- clearRect(double x, double y, double w, double h)
- strokeRect(double x, double y, double w, double h)
- fillRect(double x, double y, double w, double h)

图 2-4 中所示的这个简单的应用程序，用到了上面讲的那三个方法。

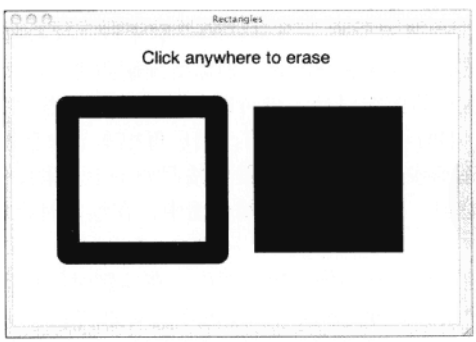


图 2-4 绘制简单的矩形

该应用程序使用 strokeRect() 方法来绘制左边的矩形，使用 fillRect() 方法来绘制右边的矩形。当用户在 canvas 内任意处点击鼠标时，程序调用 clearRect() 方法将整个 canvas 的内容清除。

图 2-4 所示应用程序的代码列在了程序清单 2-1 之中。

通常情况下，正如你在本书中经常会遇到的那样，strokeRect() 方法所绘制的是方角（square

corner）矩形。不过，本应用程序将绘图环境的 lineJoin 属性设置成了 round，因此，你看到的效果是图 2-4 中那样：左边的矩形是圆角矩形。在本书的 2.8.7 小节当中，将会详细地讲解如何通过 lineJoin 属性来控制两个线段焦点处的显示效果。

程序清单 2-1　绘制简单的矩形

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

context.lineJoin = 'round';
context.lineWidth = 30;

context.font = '24px Helvetica';
context.fillText('Click anywhere to erase', 175,40);

context.strokeRect(75, 100, 200, 200);
context.fillRect(325, 100, 200, 200);

context.canvas.onmousedown = function (e) {
   context.clearRect(0, 0, canvas.width,canvas.height);
};
```

除了要考虑 lineJoin 属性之外，strokeRect() 方法还需要考虑 lineWidth 属性。该属性以像素为单位，指定所绘线段的宽度。表 2-1 总结了 clearRect()、strokeRect() 及 fillRect() 方法。

表 2-1　矩形的清除、描边与填充

方　法	描　述
clearRect(double x, double y, double w, double h)	将指定矩形与当前剪辑区域相交范围内的所有像素清除。 在默认情况下，剪辑区域的大小就是整个 canvas。所以，如果你没有改动剪辑区域的话，那么在参数所指范围内的所有像素都会被清除。 所谓“清除像素”，指的是将其颜色设置为全透明的黑色。这在实际效果上就等同于“擦除”（erase）或者“清除”（clear）了某个像素，从而使得 canvas 的背景可以透过该像素显示出来
strokeRect(double x, double y, double w, double h)	使用如下属性，为指定的矩形描边： • strokeStyle • lineWidth • lineJoin • miterLimit 如果宽度（w 参数）或高度（h 参数）有一个为 0 的话，那么该方法将会分别绘制一条竖线或横线。如果两者都是 0，则不会绘制任何东西
fillRect(double x, double y, double w, double h)	使用 fillStyle 属性填充指定的矩形。如果宽度或高度是 0 的话，那么该方法会以为调用者办了一件傻事，所以它不会进行任何绘制

小技巧：圆角矩形的绘制

本节中的范例程序是通过 lineJoin 属性来绘制圆角矩形的。Canvas 规范描述了绘制这些圆角的详细流程，没有留下什么自由发挥（improvisation）的余地。如果想要控制诸如圆角半径之类的一些属性，那么你必须自己来绘制这些个圆角才行。在本书 2.9.3 小节中将会介绍具体做法。

2.4 颜色与透明度

上一节的应用程序（参见图 2-4）使用了默认颜色，也就是不透明的（opaque）黑色，来对矩形进行描边与填充。在实际应用中，肯定会用到其他的颜色，这可以通过绘图环境的 strokeStyle 与 fillStyle 属性来设置。图 2-5 中所示的应用程序与图 2-4 中的类似，不过，它在绘制这两个矩形时，所使用的颜色并非不透明的黑色。[⊖]

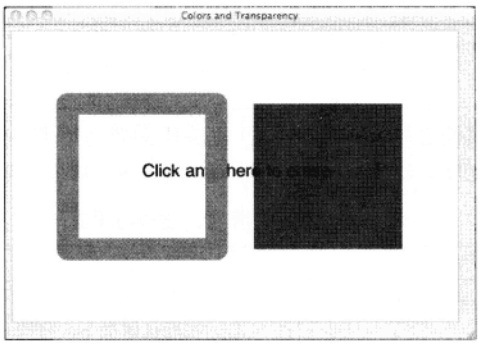


图 2-5 颜色与透明度

图 2-5 中所示应用程序的代码参见程序清单 2-2。

程序清单 2-2 颜色与透明度

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

context.lineJoin = 'round';
context.lineWidth = 30;

context.font = '24px Helvetica';
context.fillText('Click anywhere to erase', 175,200);

context.strokeStyle = 'goldenrod';
context.fillStyle = 'rgba(0,0,255,0.5)';

context.strokeRect(75, 100, 200, 200);
context.fillRect(325, 100, 200, 200);

context.canvas.onmousedown = function (e) {
   context.clearRect(0, 0, canvas.width, canvas.height);
};
```

⊖ 透明度、渐变及其他一些效果可参见彩插。——编辑注

该应用程序使用了两种颜色：用菊花黄[㊀]来描边，用半透明的蓝色来填充。在图 2-5 中可以看到这种透明效果，我们能通过半透明的蓝色看到 canvas 背景上面的文本。请注意，文本并没有透过左边矩形的边框显示出来，因为该边框用的是不透明色。

strokeStyle 与 fillStyle 的属性值可以是任意有效的 CSS 颜色字串。在 http://dev.w3.org/csswg/css3-color 可以找到一份完整的规范书，其中列举了所有可以用来指定 CSS 颜色字串的方式。可以用 RGB（red/green/blue，红 / 绿 / 蓝）、RGBA（red/green/blue/alpha，红 / 绿 / 蓝 / 透明度）、HSL（hue/saturation/lightness，色相 / 饱和度 / 亮度）、HSLA（hue/saturation/lightness/alpha，色相 / 饱和度 / 亮度 / 透明度）以及十六进制 RGB 标注法来指定颜色，也可以通过“yellow”（黄色）、“silver”（银色）或“teal”（青色）这样的颜色名称来指定。除此之外，还可以使用 SVG 1.0 规范中的颜色名称[㊁]，像是“goldenrod”（菊花黄）、“darksalmon”（深澄红色）或“chocolate”（巧克力色）。 下面还列举了一些颜色字串的例子：

- #ffffff
- #642
- rgba(100,100,100,0.8)
- rgb(255,255,0)
- hsl(20,62%,28%)
- hsla(40,82%,33%,0.6)
- antiquewhite（古董白）
- burlywood（硬木色或实木色）
- cadetblue（军服蓝）

小技巧：你的浏览器可能并不支持全部 SVG 1.0 标准的颜色名称

CSS3 颜色规范中这样说：

工作组并不要求所有 CSS3 的实现程序都支持所有的属性或数值。

读过这段声明之后，你也就不会惊讶于为何某些浏览器并不支持 CSS3 颜色规范中的全部颜色名称了。

提示：HSL 格式的颜色值

CSS3 颜色规范中说道，之所以要增加对 HSL 格式的支持，是因为以 RGB 方式来指定颜色，主要有两个缺陷：第一，它是以硬件为导向的。这种表述颜色的形式，是基于“阴极射线管”[㊂]的；第二，它不直观。

与 RGB 格式一样，HSL 格式也有三个分量，不过 RGB 的三个分量是红、绿、蓝，而 HSL 的三个分量则是色相、饱和度与亮度。HSL 颜色可以通过一个颜色轮盘（color wheel）来选取。在轮盘中，红色位于 0 度角（同时也位于 360 度角），绿色位于 120 度角，而蓝色则位于 240 度角。

首先在颜色轮盘中要通过角度来指定 HSL 颜色的第一个分量值。第二个与第三个分量值表示

㊀ goldenrod，又叫金麒麟色、浓黄色。详情参见：http://en.wikipedia.org/wiki/Goldenrod_(color)。——译者注

㊁ SVG规范中的颜色部分，请参阅：http://www.w3.org/TR/SVG11/color.html。其中支持的颜色名称，请参阅：http://www. w3.org/TR/SVG11/types.html#ColorKeywords。——译者注

㊂ cathode ray tube，简称CRT，中文又叫“显像管”、“布劳恩管”。——译者注

饱和度与亮度。对于饱和度来说，100%表示完全饱和，0%则表示灰色。对于亮度来说，100%表示白色，而50%表示正常颜色。（请注意：CSS3颜色规范中将“正常”这个词两边打上了引号，然而并未详述究竟如何才算是“正常”。）

可以很方便地将HSL格式的颜色值转换为RGB格式，反过来也是。你可以自行判断哪种表述形式更为直观，并选定自己所使用的格式。

提示：globalAlpha属性

除了可以通过rgba()或hsla()方法来指定半透明色的alpha分量外，还可以通过globalAlpha属性来改变透明度。浏览器会将该属性指定的透明度应用于所有绘制的图形与图像上面。该属性的值必须介于0.0（完全透明）与1.0（完全不透明）之间。默认的globalAlpha属性值是1.0。

提示：为什么strokeStyle和fillStyle属性不叫做strokeColor与fillColor呢

读者可能会问，为什么strokeStyle与fillStyle属性不被命名为strokeColor与fillColor呢？原因是，尽管可以将CSS3颜色字串用于strokeStyle与fillStyle属性，然而它们也可以被设置为渐变色或者图案。在下一节中，我们就会讲到渐变色和图案了。

2.5 渐变色与图案

除了颜色值之外，也可以为strokeStyle与fillStyle属性指定渐变色与图案。那么我们就来看看应该如何指定吧。

2.5.1 渐变色

Canvas元素支持线性（linear）渐变与放射（radial）渐变。我们先来看看前者。

2.5.1.1 线性渐变

程序清单2-3中展示了几种创建线性渐变的方式。

图2-6中所示应用程序的代码，列在了程序清单2-3之中。

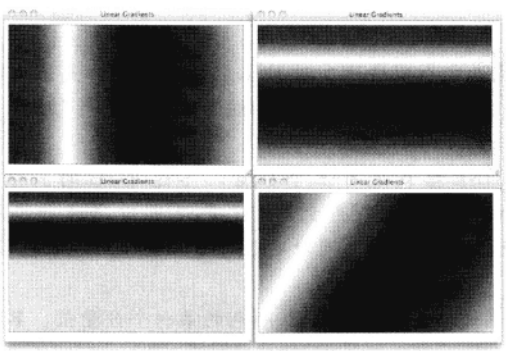


图2-6 线性渐变

该应用程序的代码通过调用createLinearGradient()方法来创建线性渐变。需要向该方

法传入两个点的 x、y 坐标，两点之间的连线就是 canvas 建立颜色渐变效果的依据。调用 createLinearGradient() 方法之后，该方法会返回一个 CanvasGradient 实例。最后，应用程序将该渐变色设置为 fillStyle 属性的值，这样的话，接下来调用 fill() 方法时，都会使用此渐变色进行填充，直到将填充属性设置成其他值为止。

在创建好渐变色之后，该段代码通过调用 CanvasGradient 之中唯一的方法 addColorStop() 来向渐变色中增加 5 个“颜色停止点”（color stop）。该方法接受两个参数：一个是位于 0 ～ 1.0 之间的 double 值，代表颜色停止点在渐变线上的位置；另一个是 DOMString 类型的 CSS3 颜色字串值。

程序清单 2-3　线性渐变

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    gradient = context.createLinearGradient(0, 0,canvas.width, 0);

gradient.addColorStop(0,    'blue');
gradient.addColorStop(0.25, 'white');
gradient.addColorStop(0.5,  'purple');
gradient.addColorStop(0.75, 'red');
gradient.addColorStop(1,    'yellow');

context.fillStyle = gradient;
context.rect(0, 0, canvas.width, canvas.height);
```

以上这段代码所创建的渐变色，如图 2-6 左上角所示。图 2-6 中的所有截屏效果，都是由该应用程序所创建的，它们之间唯一的差别就在于应用程序是如何创建那个渐变色的。从右上角开始，按顺时针方向，其余三幅截屏之中的渐变效果分别是用如下渐变色来绘制的：

```
gradient = context.createLinearGradient(0, 0, 0,canvas.height);
```

上述渐变色是根据一条垂直的线段来创建的，其效果如图 2-6 右上角所示。

```
gradient =
   context.createLinearGradient(0, 0,canvas.width, canvas.height);
```

图 2-6 右下角的渐变效果是由上述代码创建的，该渐变色所依据的线段是带有一定角度的斜线。

```
gradient = context.createLinearGradient(0, 0, 0,canvas.height/2);
```

最后，上述代码创建了图中左下角的渐变效果。请注意，上述渐变色所依据的线段是从 canvas 的顶部延伸到中部，而应用程序则要用该渐变色来填充整个 canvas。此时，Canvas 元素将会使用渐变色中的最后一个颜色来填充 canvas 的下半部分。

2.5.1.2　放射渐变

正如在上一小节中看到的那样，创建线性渐变时需要指定一条渐变线。要创建放射渐变，必须指定两个圆形，它们表示某个圆锥的起止部位。放射渐变的效果如图 2-7 所示。

图 2-7 所示应用程序的代码列在了程序清单 2-4 之中。

该段代码在 canvas 底部指定了一个半径为 10 像素的小圆，又在顶部指定了一个半径为 100 像素的大圆，然后根据这两个圆来创建放射渐变效果。这两个圆在水平方向上都与 canvas 呈居中对齐的关系。

与 createLinearGradient() 方法一样，createRadialGradient() 方法也会返回一个 CanvasGradient 的实例。应用程序的代码在渐变色中加入了 4 个颜色停止点，然后将该渐变色设置为 fillStyle 属性的值。

请注意，该段代码是要将整个 canvas 都以该渐变色来填充。然而，与线性渐变不同，放射渐

变的填充范围仅局限于由传递给 createRadialGradient() 方法的那两个圆形所定义的圆锥区域内，而不是像线性渐变那样，使用最后一个渐变色来填充渐变线以外的区域。

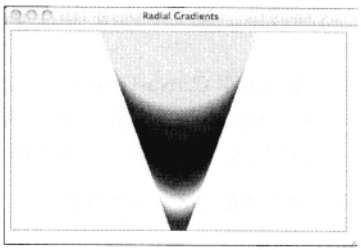


图 2-7 放射渐变

程序清单 2-4 放射渐变

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    gradient = context.createRadialGradient(
                  canvas.width/2, canvas.height,10,
                  canvas.width/2, 0, 100);

gradient.addColorStop(0, 'blue');
gradient.addColorStop(0.25, 'white');
gradient.addColorStop(0.5, 'purple');
gradient.addColorStop(0.75, 'red');
gradient.addColorStop(1, 'yellow');

context.fillStyle = gradient;
context.rect(0, 0, canvas.width, canvas.height);
context.fill();
```

表 2-2 总结了 createLinearGradient() 与 createRadialGradient() 方法的用法。

表 2-2 渐变色

方法	描述
CanvasGradient createLinearGradient(double x0, double y0, double x1, double y1)	创建线性渐变。传入该方法的参数表示渐变线的两个端点。该方法返回一个 CanvasGradient 实例，可以通过 CanvasGradient.addColorStop() 方法来向该渐变色增加颜色停止点
CanvasGradient createRadialGradient(double x0, double y0, double r0, double x1, double y1, double r1)	创建放射渐变。该方法的参数代表位于圆锥形渐变区域两端的圆形。与 createLinearGradient() 方法一样，该方法也返回一个 CanvasGradient 实例

2.5.2 图案

除了颜色与渐变色，Canvas 元素也允许使用图案来对图形和文本进行描边与填充。这里的图案可以是以下三种之一：image 元素、canvas 元素或 video 元素。

可以用 createPattern() 方法来创建图案，该方法接收两个参数：图案本身，以及一个用来告知浏览器应该如何重复图案的字符串。第二个参数可以取如下的值：repeat、repeat-x、repeat-y 或

no-repeat。这些值所对应的效果如图 2-8 所示，图中的这个应用程序采用一幅图像来创建图案对象，并且将 fillStyle 属性设置为该图案，然后用此图案来填充整个 canvas。

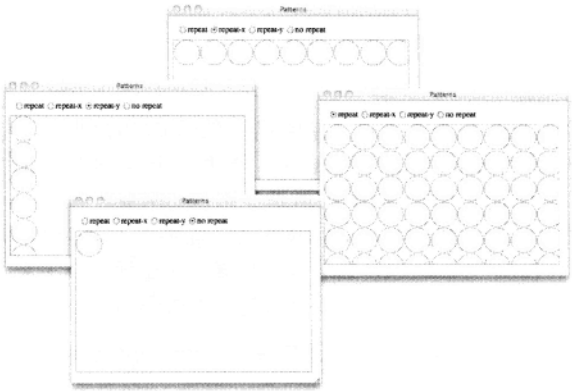


图 2-8　通过参数来控制图案的重复方式

图 2-8 中应用程序的代码，参见程序清单 2-5。

该程序的 HTML 代码建立了单选按钮（radio button）、canvas，并引入了范例程序所用到的 JavaScript 代码。这段 JavaScript 代码列在程序清单 2-6 之中。

此段 JavaScript 代码先是创建了一幅图像，然后以该图像创建了一个图案对象，并根据用户所选的单选按钮来设置图案的重复模式参数。接下来，应用程序使用这个图案来填充整个 canvas。

程序清单 2-5　图案的用法（HTML 代码）

```
<!DOCTYPE html>
   <head>
      <title>Patterns</title>

      <style>
         #canvas {
            background: #eeeeee;
            border: thin solid cornflowerblue;
         }

         #radios {
            padding: 10px;
         }
      </style>
   </head>

   <body>
      <div id='radios'>
         <input type='radio'
                id='repeatRadio'name='patternRadio' checked/>repeat
         <input type='radio'
                id='repeatXRadio'name='patternRadio'/>repeat-x
         <input type='radio'
                id='repeatYRadio'name='patternRadio'/>repeat-y
         <input type='radio'
                id='noRepeatRadio'name='patternRadio'/>no repeat
```

```
    </div>
    <canvas id='canvas' width='450'height='275'>
     Canvas not supported
    </canvas>
    <script src='example.js'></script>
  </body>
</html>
```

请注意，每次用户点击单选按钮时，应用程序的代码都会调用 createPattern() 方法来创建一个新的 CanvasPattern 对象。在这种情况下，创建一个新的图案对象是有必要的。因为 CanvasPattern 对象是 JavaScript 语言中的那种所谓“不透明对象”（opaque object），它没有提供用于操作其内容的属性及方法。假若 CanvasPattern 对象提供了一个 setPattern() 方法的话，那么，你就可以只建立一个 CanvasPattern 对象，然后很容易就能通过那个方法来修改其中的图案了。然而，实际上并不能这么做，因为 CanvasPattern 对象是“不透明的”。

程序清单 2-6　图案的用法（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    repeatRadio   =document.getElementById('repeatRadio'),
    noRepeatRadio =document.getElementById('noRepeatRadio'),
    repeatXRadio  =document.getElementById('repeatXRadio'),
    repeatYRadio  =document.getElementById('repeatYRadio'),
    image = new Image();

// Functions.......................................................

function fillCanvasWithPattern(repeatString) {
   var pattern = context.createPattern(image,repeatString);
   context.clearRect(0, 0, canvas.width,canvas.height);
   context.fillStyle = pattern;
   context.fillRect(0, 0, canvas.width,canvas.height);
   context.fill();
}

// Event handlers..................................................

repeatRadio.onclick = function (e) {
  fillCanvasWithPattern('repeat');
};

repeatXRadio.onclick = function (e) {
  fillCanvasWithPattern('repeat-x');
};

repeatYRadio.onclick = function (e) {
  fillCanvasWithPattern('repeat-y');
};

noRepeatRadio.onclick = function (e) {
  fillCanvasWithPattern('no-repeat');
};

// Initialization.................................................

image.src = 'redball.png';
image.onload = function (e) {
   fillCanvasWithPattern('repeat');
};
```

表 2-3 描述了 createPattern() 方法的用法。

表 2-3　createPattern() 方法的用法

方　法	描　述
CanvasPattern createPattern(HTMLImageElement | HTMLCanvasElement | HTMLVideoElement image, DOMString repetition)	创建一个可以用来在 canvas 之中对图形和文本进行描边与填充的图案。该方法的第 1 个参数指定了图案所用的图像，它可以是 image 元素、canvas 元素或者 video 元素。第 2 个参数告诉浏览器，在对图形进行描边或填充时，应该如何重复该图案。这个参数的有效取值是 repeat、repeat-x、repeat-y 及 no-repeat

2.6　阴影

在 canvas 之中进行绘制时，不管要画的是图形、文本，还是图像，都可以通过修改绘图环境中的如下 4 个属性值来指定阴影效果：

- shadowColor：CSS3 格式的颜色。
- shadowOffsetX：从图形或文本到阴影的水平像素偏移。
- shadowOffsetY：从图形或文本到阴影的垂直像素偏移。
- shadowBlur：一个与像素无关的值。该值被用于高斯模糊方程之中，以便对阴影进行模糊化处理。

如果满足以下条件，那么使用 Canvas 的绘图环境对象就可以绘制出阴影效果了：

1. 指定的 shadowColor 值不是全透明的。
2. 在其余的阴影属性之中，存在一个非 0 的值。

图 2-9 展示了图 2-1 中那个画图应用程序上面的按钮图标。

这个画图应用程序的所有按钮图标都运用了阴影效果，这使得它们看上去好像是浮动在页面上一样。然而，该应用程序对被选中的按钮图标使用了与其余图标不同的阴影属性。该引用属性的偏移量与模糊值比其他图标要大，这样的话，它看起来好像要比其他按钮浮动得更高一些。通过图 2-9 中的 Text 按钮（上面写有“T”字母的按钮）图标可以看出来这一点。

图 2-9　使用阴影效果制作具有深度感的按钮

这个画图应用程序用于设置图标阴影效果的方法，列在了程序清单 2-7 之中。

程序清单 2-7　使用阴影效果来制作具有深度感的按钮

```
var SHADOW_COLOR = 'rgba(0,0,0,0.7)';
...

function setIconShadow() {
   iconContext.shadowColor = SHADOW_COLOR;
   iconContext.shadowOffsetX = 1;
   iconContext.shadowOffsetY = 1;
   iconContext.shadowBlur = 2;
}
function setSelectedIconShadow() {
```

```
    iconContext.shadowColor = SHADOW_COLOR;
    iconContext.shadowOffsetX = 4;
    iconContext.shadowOffsetY = 4;
    iconContext.shadowBlur = 5;
}
```

图 2-9 之中的图标都是带有阴影效果的填充矩形，不过，Canvas 绘图环境对象也可以在对文本或路径进行描边时绘制阴影效果。图 2-10 演示了进行描边与填充时，所绘阴影效果的区别。

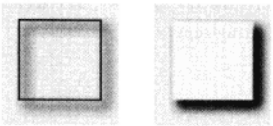


图 2-10 描边时的阴影效果（左）与填充时的阴影效果（右）

小技巧：使用半透明色来绘制阴影

通常来说，使用半透明色来绘制阴影是个不错的选择，因为这样一来，背景就可以透过阴影显示出来了。

提示：开启与禁用阴影效果

根据 Canvas 规范，浏览器只应在符合如下情形时，才去绘制阴影：a) 指定了一个非全透明的 shadowColor 属性值；b) 在其余的三个属性 shadowBlur、shadowOffsetX 与 shadowOffsetY 之中，至少有一个属性不是 0。因此，有一种简单的办法可以用来禁用阴影效果，那就是将 shadowColor 属性设置为 undefined。然而，在编写本书时，只有在 WebKit 内核的浏览器上，才可以通过将 shadowColor 属性设置为 undefined 来禁用阴影效果，这种方法并不适用于 Firefox 或 Opera 浏览器。为了确保能够在所有的浏览器上都开启或禁用阴影效果，你应该将所有与阴影相关的属性都设置到位。这一点可以通过手动编写代码或是使用绘图环境对象的 save() 与 restore() 方法来做到。

在本书写作之时，浏览器对阴影效果的绘制并不考虑当前的图像合成设定。这项合成设定是用来决定浏览器应该如何将某个物体绘制于另一个物体之上的。然而，有些浏览器厂商想要将规则改为：只有当图形合成模式被设置为 source-over 时，浏览器才绘制阴影效果。关于图形合成模式的设置，本书 2.14 节将会提供更多信息。

内嵌阴影

如果你为 shadowOffsetX 与 shadowOffsetY 属性设置了非 0 的正数值，那么不论你绘制什么内容，它看起来都像是浮在了 canvas 之上。这些属性的值越大，我们就会觉得它在 canvas 上面浮动得越高。

你也可以为这些属性指定负值，其效果如图 2-11 所示。

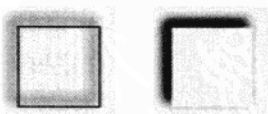


图 2-11 负偏移量的阴影效果：左图为描边，右图为填充

负偏移量可以用来制作如图 2-12 那样的内嵌阴影（inset shadow）效果。图中显示的是画图程序里面的橡皮擦工具，它有一个淡淡的内嵌阴影，使得橡皮擦的表面看上去有种凹陷的效果。

程序清单 2-8 列出了画图应用程序中与橡皮擦及其内嵌阴影的绘制有关的代码。该应用程序通过将阴影的 X、Y 偏移量都指定为 −5 像素来制作内嵌阴影，这样做出来的效果与图 2-11 中所展示的效果类似。

请注意，程序在绘图环境对象上调用了 clip() 方法，该调用使得后续被调用的 stroke() 方法以及此方法所生成的阴影，都被局限在这个圆形的范围之内。这意味着，与图 2-11 中所示的描边矩

形效果不同，浏览器不会在圆形范围之外进行绘制。在本书的 2.15 节中将会更加详细地讨论剪辑区域与 clip() 方法。

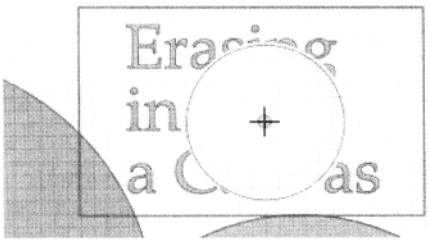


图 2-12　画图程序中所用的内嵌阴影效果

程序清单 2-8　绘制内嵌阴影

```
var drawingContext =
      document.getElementById('drawingCanvas').getContext('2d'),

    ERASER_LINE_WIDTH     =  1,
    ERASER_SHADOW_STYLE   =  'blue',
    ERASER_STROKE_STYLE   =  'rgba(0,0,255,0.6)',
    ERASER_SHADOW_OFFSET  =  -5,
    ERASER_SHADOW_BLUR    =  20,
    ERASER_RADIUS         =  60;

// Eraser.....................................................

function setEraserAttributes() {
  drawingContext.lineWidth      =ERASER_LINE_WIDTH;
  drawingContext.shadowColor    =ERASER_SHADOW_STYLE;
  drawingContext.shadowOffsetX =ERASER_SHADOW_OFFSET;
  drawingContext.shadowOffsetY =ERASER_SHADOW_OFFSET;
  drawingContext.shadowBlur     =ERASER_SHADOW_BLUR;
  drawingContext.strokeStyle    =ERASER_STROKE_STYLE;
}

function drawEraser(loc) {
   drawingContext.save();
   setEraserAttributes();

   drawingContext.beginPath();
   drawingContext.arc(loc.x, loc.y,ERASER_RADIUS,
                      0, Math.PI*2, false);
   drawingContext.clip();
   drawingContext.stroke();

   drawingContext.restore();
}
```

表 2-4 总结了控制阴影绘制方式的 4 个属性。

小技巧：阴影效果的绘制可能很耗时

正如 2.2 节中说的那样，为了绘制阴影，浏览器需要将阴影先渲染到一个辅助的位图之中，最后这个辅助位图中的内容会与屏幕上 canvas 之中的内容进行图像合成。由于使用了这种辅助位

图，所以绘制阴影可能是一项很耗时的操作。

如果你要绘制的是简单的图形、文本或图像，那么其阴影的绘制可能并不会带来性能问题。然而，如果你对 canvas 之中的动画对象运用阴影效果的话，那么其性能肯定比不用阴影效果时要差一些。本书第 5.11 节将会详述动画与阴影效果。

表 2-4 CanvasRenderingContext2D 之中与阴影效果有关的属性

属　性	描　述
shadowBlur	表示阴影效果如何延伸的 double 值。浏览器在阴影之上运用高斯模糊时，将会用到该值。它与像素无关，只会被用在高斯模糊方程之中。其默认值为 0
shadowColor	CSS 格式的颜色字串。默认值是 rgba(0,0,0,0)，意思就是完全透明的黑色
shadowOffsetX	阴影在 X 轴方向的偏移量，以像素为单位。默认值是 0
shadowOffsetY	阴影在 Y 轴方向的偏移量，以像素为单位。默认值是 0

2.7 路径、描边与填充

迄今为止，在本章之中我们所绘制的唯一图形，就是通过在 Canvas 的绘图环境对象上调用 strokeRect() 方法所画的矩形。我们也通过调用 fillRect() 方法对其进行了填充。这两个方法都是立即生效的。实际上，它们是 Canvas 绘图环境中仅有的两个可以用来立即绘制图形的方法（strokeText() 与 fillText() 方法也是进行立即绘制的，但文本不算是图形）。绘图环境对象中还有一些方法，用于绘制诸如贝塞尔曲线（bézier curve）这样更为复杂的图形，这些方法都是基于路径（path）的。

大多数绘制系统，例如 Scalable Vector Graphics（可缩放向量图形，简称 SVG）、Apple 的 Cocoa 框架，以及 Adobe Illustrator 等，都是基于路径的。使用这些绘制系统时，你需要先定义一个路径，然后再对其进行描边（也就是绘制路径的轮廓线）或填充，也可以在描边的同时进行填充。图 2-13 演示了这三种绘制方式。

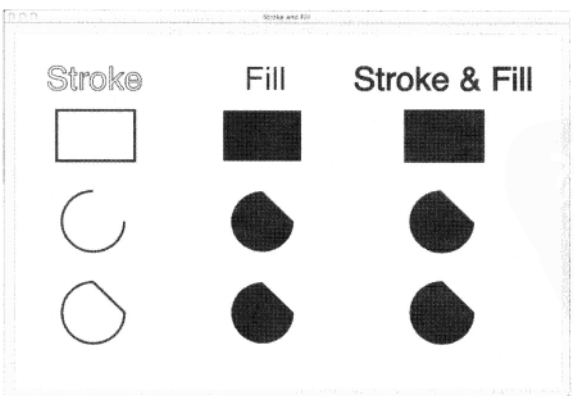


图 2-13 图形的描边与填充效果

该应用程序创建了 9 个不同的路径，对左边一列的路径进行了描边操作，对中间一列的路径进行了填充，并对右边一列的路径同时进行描边与填充。

第一行的矩形路径与最后一行的圆弧路径都是封闭路径（closed path），而中间一行的弧形路径则是开放路径（open path）。请注意，不论一个路径是开放或是封闭，你都可以对其进行填充。当填充某个开放路径时，浏览器会把它当成封闭路径来填充。图中右边一列的中间那个图形，就是这种效果。

程序清单 2-9 列出了图 2-13 中那个应用程序的代码。

程序清单 2-9　文本、矩形与圆弧的描边及填充

```
var context =document.getElementById('drawingCanvas').getContext('2d');

// Functions.......................................................

function drawGrid(context, color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

// Initialization..................................................

drawGrid(context, 'lightgray', 10, 10);

// Drawing attributes..............................................

context.font = '48pt Helvetica';
context.strokeStyle = 'blue';
context.fillStyle = 'red';
context.lineWidth = '2';        // Line width set to 2 for text

// Text............................................................

context.strokeText('Stroke', 60, 110);
context.fillText('Fill', 440, 110);

context.strokeText('Stroke & Fill', 650, 110);
context.fillText('Stroke & Fill', 650, 110);

// Rectangles......................................................

context.lineWidth = '5';        // Line width set to 5 for shapes
context.beginPath();
context.rect(80, 150, 150, 100);
context.stroke();

context.beginPath();
context.rect(400, 150, 150, 100);
context.fill();

context.beginPath();
context.rect(750, 150, 150, 100);
context.stroke();
context.fill();


// Open arcs.......................................................

context.beginPath();
context.arc(150, 370, 60, 0, Math.PI*3/2);
context.stroke();

context.beginPath();
```

```
context.arc(475, 370, 60, 0, Math.PI*3/2);
context.fill();

context.beginPath();
context.arc(820, 370, 60, 0, Math.PI*3/2);
context.stroke();
context.fill();

// Closed arcs.....................................................

context.beginPath();
context.arc(150, 550, 60, 0, Math.PI*3/2);
context.closePath();
context.stroke();

context.beginPath();
context.arc(475, 550, 60, 0, Math.PI*3/2);
context.closePath();
context.fill();

context.beginPath();
context.arc(820, 550, 60, 0, Math.PI*3/2);
context.closePath();
context.stroke();
context.fill();
```

首先调用 beginPath() 方法来开始一段新的路径，rect() 与 arc() 方法分别用于创建矩形及弧形路径。然后，应用程序在绘图环境对象上调用 stroke() 与 fill() 方法，对刚才那些路径进行描边或填充。

描边与填充操作的效果取决于当前的绘图属性，这些属性包括了 lineWidth、strokeStyle、fillStyle 以及阴影属性等。比如，程序清单 2-9 中的这个应用程序，将 lineWidth 属性值设置为 2，然后对文本进行描边，其后又将其重置为 5，再对路径进行描边。

由 rect() 方法所创建的路径是封闭的，然而，arc() 方法创建的圆弧路径则不封闭，除非你用它创建的是个圆形路径。要封闭某段路径，必须像程序清单 2-9 中那样，调用 closePath() 方法才行。

表 2-5 总结了本应用程序中与路径相关的方法。

表 2-5 CanvasRenderingContext2D 之中与路径有关的方法

方　法	描　述
arc()	在当前路径中增加一段表示圆弧或圆形的子路径。与 rect() 方法不同，可以通过一个 boolean 参数来控制该段子路径的方向。如果此参数是 true，那么 arc() 所创建的子路径就是顺时针的，否则就是逆时针的。如果在调用该方法时已经有其他的子路径存在，那么 arc() 方法则会用一条线段把已有路径的终点与这段圆弧路径的起点连接起来
beginPath()	将当前路径之中的所有子路径都清除掉，以此来重置当前路径。有关子路径的更多内容，请参阅 2.7.1 小节。在需要开始一段新的路径时，应该调用此方法
closePath()	显式地封闭某段开放路径。该方法用于封闭圆弧路径以及由曲线或线段所创建的开放路径
fill()	使用 fillStyle 对当前路径的内部进行填充
rect(double x, double y, double width, double height)	在坐标 (x,y) 处建立一个宽度为 width、高度为 height 的矩形子路径。该子路径一定是封闭的，而且总是按逆时针方向来创建的
stroke()	使用 strokeStyle 来描绘当前路径的轮廓线

提示：路径与隐形墨水

有一个很恰当的比喻，可以用来说明“创建路径随后对其进行描边或填充”这个操作。我们可以将该操作比作“使用隐形墨水来绘图”。

你用隐形墨水所绘制的内容并不会立刻显示出来，必须进行一些后续操作，像是加热、涂抹化学药品、照射红外线等，才可以将你所画的内容显示出来。如果读者关注这个话题，可以在 http://en.wikipedia.org/wiki/Invisible_ink 读到所有关于隐形墨水的知识。

使用 rect() 与 arc() 这样的方法来创建路径，就好比使用隐形墨水来进行绘制一样。这些方法会创建一条不可见的路径，稍后可以调用 stroke() 或 fill() 令其可见。

2.7.1 路径与子路径

在某一时刻，canvas 之中只能有一条路径存在，Canvas 规范将其称为“当前路径”（current path）。然而，这条路径却可以包含许多子路径（subpath）。而子路径，又是由两个或更多的点组成的。比方说，可以像这样绘制出两个矩形来：

```
context.beginPath();                // Clear all subpaths from
                                    // the current path
context.rect(10, 10, 100, 100);     // Add a subpath with four points
context.stroke();                   // Stroke the subpath containing
                                    // four points

context.beginPath();                // Clear all subpaths from the
                                    // current path
context.rect(50, 50, 100, 100);     // Add a subpath with four points
context.stroke();                   // Stroke the subpath containing
                                    // four points
```

以上这段代码通过调用 beginPath() 来开始一段新的路径，该方法会将当前路径中的所有子路径都清除掉。然后，这段代码调用了 rect() 方法，此方法向当前路径中增加了一个含有 4 个点的子路径。最后，调用 stroke() 方法，将当前路径的轮廓线描绘出来，使得这个矩形出现在 canvas 之中。

接下来，这段代码又一次调用了 beginPath() 方法，该方法清除了上一次调用 rect() 方法时所创建的子路径。然后，再一次调用 rect() 方法，这次还是会向当前路径中增加一段含有 4 个点的子路径。最后，对该路径进行描边，使得第二个矩形也出现在了 canvas 之中。

现在考虑一下，如果将第二个 beginPath() 调用去掉，会怎么样呢？像是这样：

```
context.beginPath();                // Clear all subpaths from the
                                    // current path
context.rect(10, 10, 100, 100);     // Add a subpath with four points
context.stroke();                   // Stroke the subpath containing
                                    // four points

context.rect(50, 50, 100, 100);     // Add a second subpath with
                                    // four points
context.stroke();                   // Stroke both subpaths
```

上面这段代码在一开始与刚才那段是一样的：先调用 beginPath() 来清除当前路径中的所有子路径，然后调用 rect() 来创建一条包含矩形 4 个点的子路径，再调用 stroke() 方法使得这个矩形出现在 canvas 之上。

接下来，这段代码再次调用了 rect() 方法，不过这一次，由于没有调用 beginPath() 方法来清

除原有的子路径，所以第二次对 rect() 方法的调用，会向当前路径中增加一条子路径。最后，该段代码再一次调用 stroke() 方法，这次对 stroke() 方法的调用，将会使得当前路径中的两条子路径都被描边，这意味着它会重绘第一个矩形。

填充路径时所使用的“非零环绕规则”

如果当前路径是循环的，或是包含多个相交的子路径，那么 Canvas 的绘图环境变量就必须要判断，当 fill() 方法被调用时，应该如何对当前路径进行填充。Canvas 在填充那种互相有交叉的路径时，使用“非零环绕规则”（nonzero winding rule）来进行判断。图 2-14 演示了该规则的运用。

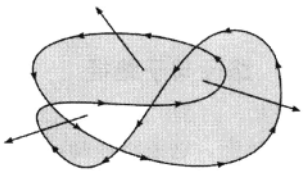


图 2-14　在填充路径时运用“非零环绕规则”来进行判断

“非零环绕规则”是这么来判断有自我交叉情况的路径的：对于路径中的任意给定区域，从该区域内部画一条足够长的线段，使此线段的终点完全落在路径范围之外。图 2-14 中的那三个箭头所描述的就是上面这个步骤。

接下来，将计数器初始化为 0，然后，每当这条线段与路径上的直线或曲线相交时，就改变计数器的值。如果是与路径的顺时针部分相交，则加 1，如果是与路径的逆时针部分相交，则减 1。若计数器的最终值不是 0，那么此区域就在路径里面，在调用 fill() 方法时，浏览器就会对其进行填充。如果最终值是 0，那么此区域就不在路径内部，浏览器也就不会对其进行填充了。

可以从图 2-14 中看出“非零环绕规则”是如何运用的。左边的那个带箭头的线段，先穿过了路径的逆时针部分，然后又穿过了路径的顺时针部分。这意味着其计数值是 0，所以该线段起点所在的那个区域就不在范围内，在调用 fill() 方法时，浏览器也就不会对其进行填充。然而，其余两条带箭头的线段，其计数值都不是 0，所以它们的起点所在的区域就会被浏览器填充。

2.7.2　剪纸效果

我们来运用一下所学到的路径、阴影以及非零环绕原则等知识，实现如图 2-15 所示的剪纸（cutout）效果。

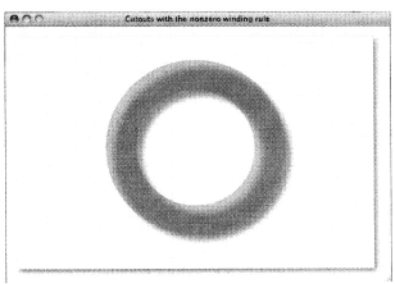


图 2-15　用两个圆形做出的剪纸效果

图 2-15 所示的应用程序，其 JavaScript 代码列在了程序清单 2-10 之中。

这段 JavaScript 代码创建了一条路径，它由两个圆形所组成，其中一个圆形在另一个的内部。通过设定 arc() 方法的最后一个参数值，该应用程序以顺时针方向绘制了内部的圆形，并且以逆时

针方向绘制了外围的圆形。绘制效果如图 2-16 上方所示。

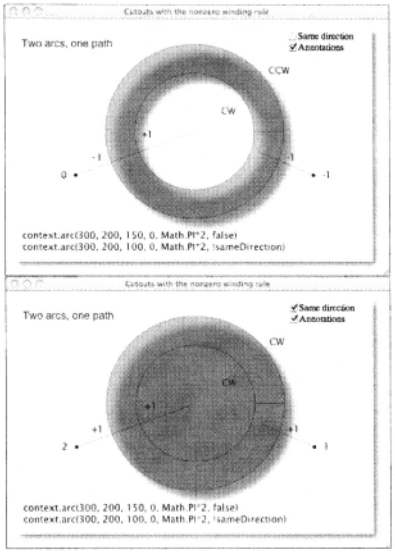


图 2-16　运用“非零环绕规则”来实现剪纸效果

在创建好路径之后，图 2-15 中的那个应用程序对该路径进行了填充。浏览器运用“非零环绕规则”，对外围圆形的内部进行了填充，不过填充的范围并不包括里面的圆，这就产生了一种剪纸图案的效果。你也可以利用此技术来剪出任意想要的形状来。

程序清单 2-10　图 2-15 所示应用程序的 JavaScript 代码

```
var context =document.getElementById('canvas').getContext('2d');

// Functions.................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function drawTwoArcs() {
   context.beginPath();
   context.arc(300, 190, 150, 0, Math.PI*2,false); // Outer: CCW
   context.arc(300, 190, 100, 0, Math.PI*2,true);  // Inner: CW

   context.fill();
   context.shadowColor = undefined;
   context.shadowOffsetX = 0;
   context.shadowOffsetY = 0;
   context.stroke();
}

function draw() {
   context.clearRect(0, 0, context.canvas.width,
                           context.canvas.height);
```

```
    drawGrid('lightgray', 10, 10);

    context.save();

    context.shadowColor = 'rgba(0,0,0,0.8)';
    context.shadowOffsetX = 12;
    context.shadowOffsetY = 12;
    context.shadowBlur = 15;

    drawTwoArcs();

    context.restore();
}

// Initialization.............................................

context.fillStyle = 'rgba(100,140,230,0.5)';
context.strokeStyle = context.fillStyle;
draw();
```

图 2-16 之中的例子是对图 2-15 所示应用程序的一种扩展，它会告诉你如果两个圆形子路径都在同一个方向上，绘制效果会如何，同时它也增加了一些注释信息，用以显示圆形子路径的绘制方向以及“非零环绕规则”的计算过程。而且，这个程序还显示了创建圆形子路径所调用的 arc() 方法。

提示：图 2-16 之中的那条横线是怎么回事

请注意图 2-16 中两个圆之间的那条横线。在图 2-15 之中也有这样一条线，不过图 2-16 使用了更深一些的描边颜色，把它画得更加明显了。

根据 Canvas 规范，当使用 arc() 方法向当前路径中增加子路径时，该方法必须将上一条子路径的终点与所画圆弧的起点相连。

制作剪纸图形

图 2-17 之中的应用程序在矩形内剪出了三个图形。与上一小节中所讨论的那个程序不同，图 2-17 所示的应用程序采用了完全不透明的颜色来填充这个包含剪纸图形的矩形。

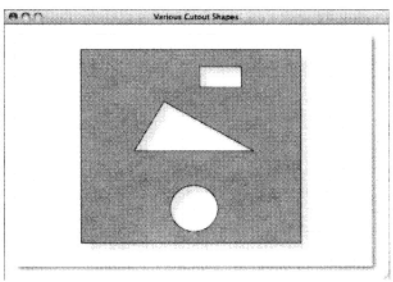


图 2-17　各种剪纸图形

该应用程序有两个值得注意的地方。首先，包围剪纸图形的是一个矩形而不是圆形。这个矩形的使用向你表明，可以用任意形状的路径来包围剪纸图形，并不一定非要用圆形。该程序建立剪纸图形所用的代码如下：

```
function drawCutouts() {
   context.beginPath();

   addOuterRectanglePath(); // Clockwise (CW)

   addCirclePath();          // Counter-clockwise (CCW)
   addRectanglePath();       // CCW
   addTrianglePath();        // CCW

   context.fill(); // Cut out shapes
}
```

addOuterRectanglePath()、addCirclePath()、addRectanglePath() 及 addTrianglePath() 方法分别向当前路径中添加了表示剪纸图形的子路径。

图 2-17 中的应用程序还有一个有意思的地方，就是其中的矩形剪纸图案。arc() 方法可以让调用者控制圆弧的绘制方向，然而 rect() 方法则没有那么方便，它总是按照顺时针方向来创建路径。可是，在本例这种情况下，需要的是一条逆时针的矩形路径，所以我们自己创建了一个 rect() 方法，此方法像 arc() 一样，可以让调用者控制矩形路径的方向：

```
function rect(x, y, w, h, direction) {
  if (direction) { // CCW
      context.moveTo(x, y);
      context.lineTo(x, y + h);
      context.lineTo(x + w, y + h);
      context.lineTo(x + w, y);
  }
  else {
      context.moveTo(x, y);
      context.lineTo(x + w, y);
      context.lineTo(x + w, y + h);
      context.lineTo(x, y + h);
  }
  context.closePath();
}
```

上述代码使用 moveTo() 与 lineTo() 方法来创建顺时针或者逆时针的矩形路径。在 2.8 节中我们将详细讲述这些方法。

该应用程序在建立路径时，分别使用了两种不同的方式来创建外围矩形及内部的矩形剪纸图形：

```
function addOuterRectanglePath() {
   context.rect(110, 25, 370, 335);
}

function addRectanglePath() {
   rect(310, 55, 70, 35, true);
}
```

addOuterRectanglePath() 方法使用了绘图环境对象的 rect() 方法，此方法总是按照顺时针方向来绘制矩形的，并没有提供逆时针绘制的选项。addRectanglePath() 方法创建了矩形剪纸图形的路径，它使用上面列出的那个 rect() 方法来绘制逆时针的矩形路径。

图 2-17 所示应用程序的 JavaScript 代码列在了程序清单 2-11 之中。

小技巧：路径方向真的很重要

arc() 方法的最后一个 boolean 参数用于控制所绘圆弧路径的方向。如果该参数是默认值 true，那么浏览器就会以顺时针方向来绘制路径，否则，浏览器就按照逆时针（counterclockwise）方向来绘制（或者按照 Canvas 规范中的说法，“反时针方向”（anti-clockwise））。

提示：arc() 方法可以控制路径方向，而 rect() 方法则不行

arc() 方法与 rect() 方法都可以向当前路径中添加子路径，然而 arc() 方法可以让调用者来控制路径的绘制方向。幸好可以非常容易地实现一个函数，用它来创建具有特定方向的矩形路径。程序清单 2-11 中的 rect() 方法就演示了这种做法。

小技巧：去掉由 arc() 方法所产生的那条不太美观的连接线

如果在当前路径中存在子路径的情况下调用 arc() 方法，那么此方法就会从子路径的终点向圆弧的起点画一条线。通常情况下，你并不想看到这条线段。

如果不想让这条连线出现，可以在调用 arc() 方法来绘制圆弧之前，先调用 beginPath() 方法。调用此方法会将当前路径下的所有子路径都清除掉，这样一来，arc() 方法也就不会再绘制那条连接线了。

程序清单 2-11　绘制剪纸图形的代码

```
var context =document.getElementById('canvas').getContext('2d');

// Functions.........................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function draw() {
   context.clearRect(0, 0, context.canvas.width,
                           context.canvas.height);
   drawGrid('lightgray', 10, 10);

   context.save();

   context.shadowColor = 'rgba(200,200,0,0.5)';
   context.shadowOffsetX = 12;
   context.shadowOffsetY = 12;
   context.shadowBlur = 15;

   drawCutouts();
   strokeCutoutShapes();
   context.restore();
}

function drawCutouts()  {
   context.beginPath();
   addOuterRectanglePath(); // CW


   addCirclePath();     // CCW
   addRectanglePath(); // CCW
   addTrianglePath();  // CCW

   context.fill(); // Cut out shapes
}

function strokeCutoutShapes() {
   context.save();

   context.strokeStyle = 'rgba(0,0,0,0.7)';

   context.beginPath();
```

```
    addOuterRectanglePath(); // CW
    context.stroke();

    context.beginPath();

    addCirclePath();
    addRectanglePath();
    addTrianglePath();
    context.stroke();

    context.restore();
}

function rect(x, y, w, h, direction) {
  if (direction) { // CCW
      context.moveTo(x, y);
      context.lineTo(x, y + h);
      context.lineTo(x + w, y + h);
      context.lineTo(x + w, y);
      context.closePath();
  }
  else {
      context.moveTo(x, y);
      context.lineTo(x + w, y);
      context.lineTo(x + w, y + h);
      context.lineTo(x, y + h);
      context.closePath();
  }
}

function addOuterRectanglePath() {
   context.rect(110, 25, 370, 335);
}

function addCirclePath() {
   context.arc(300, 300, 40, 0, Math.PI*2, true);
}

function addRectanglePath() {
   rect(310, 55, 70, 35, true);
}

function addTrianglePath() {
   context.moveTo(400, 200);
   context.lineTo(250, 115);
   context.lineTo(200, 200);
   context.closePath();
}

// Initialization.....................................................

context.fillStyle = 'goldenrod';
draw();
```

2.8　线段

Canvas 绘图环境提供了两个可以用来创建线性路径的方法：moveTo() 与 lineTo()。要使线性路径（或者俗称“线段”）出现在 canvas 之中，您必须在创建路径之后调用 stroke() 方法，这样才可以像图 2-18 之中的应用程序那样，在 canvas 之中画出两条线段来。

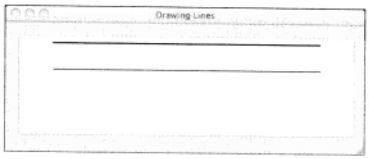


图 2-18 线段的绘制

图 2-18 所示应用程序的代码，参见程序清单 2-12。

程序清单 2-12 在 canvas 中绘制两条线段

```
var context =document.getElementById('canvas').getContext('2d');

context.lineWidth = 1;
context.beginPath();
context.moveTo(50, 10);
context.lineTo(450, 10);
context.stroke();
context.beginPath();
context.moveTo(50.5,  50.5);
context.lineTo(450.5, 50.5);
context.stroke();
```

在将 lineWidth 属性设置为 1 像素之后，该段代码将路径的起点移动到（50, 10）坐标处，并绘制了一条终点为（450, 10）的线段。组合运用 moveTo()/lineTo() 方法，可以创建线性路径，创建好路径之后，代码对其进行描边操作，于是，这条水平线段就能显示在 canvas 之中了。

接下来，应用程序调用 beginPath() 方法，该方法将当前路径中的线性子路径移除。然后这段代码在上一条线段的下方又绘制了一条水平线段。

程序清单 2-12 之中的代码很简单，不过，如果你仔细看看图 2-18，你就会发现有个地方很奇怪。尽管代码在绘制之前已经先将 lineWidth 属性设置成 1 像素了，但是，上边那条线段画出来却是 2 个像素宽。下一小节将解释出现这种现象的原因。

表 2-6 总结了 moveTo() 与 lineTo() 方法的用法。

表 2-6 moveTo() 与 iineTo() 方法的用法

方　法	描　述
moveTo(x, y)	向当前路径中增加一条子路径，该子路径只包含一个点，就是由参数传入的那个点。该方法并不会从当前路径中清除任何子路径
lineTo(x, y)	如果当前路径中没有子路径，那么这个方法的行为与 moveTo() 方法一样：它会创建一条新的子路径，其中包含了经由参数所传入的那个点。如果当前路径中存在子路径，那么该方法会将你所指定的那个点加入子路径中

2.8.1 线段与像素边界

如果你在某 2 个像素的边界处绘制一条 1 像素宽的线段，那么该线段实际上会占据 2 个像素的宽度，如图 2-19 所示。

如果在像素边界处绘制一条 1 像素宽的垂直线段，那么 canvas 的绘图环境对象会试着将半个像素画在边界中线的右边，将另外半个像素画在边界中线的左边。

然而，在一个整像素的范围内绘制半个像素宽的线段是不可能的，所以左右两个方向上的半像素都被扩展为 1 个像素。在图 2-19 之中，本来我们想要将线段绘制在深灰色的区域内，但实际上浏览器却将其延伸绘制到整个浅灰色的范围内。

另一方面，我们来看看如果将线段绘制在某 2 个像素之间的那个像素中，效果会怎么样。这时的效果如图 2-20 所示。

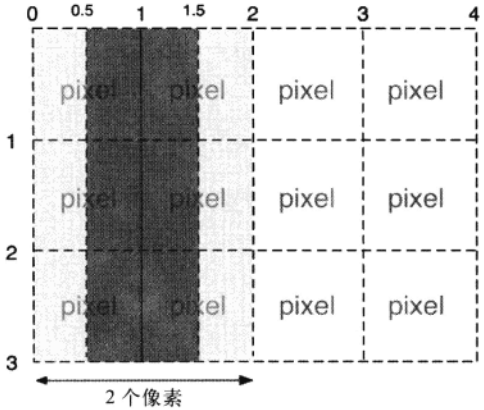


图 2-19　在像素边界处绘制线段

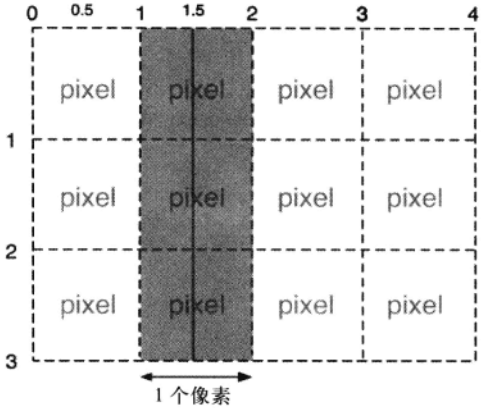


图 2-20　在某个像素范围内绘制线段

在图 2-20 之中，这条垂直线段绘制在两个像素之间，这样的话，中线左右两端的那半个像素就不会再延伸了，它们合起来恰好占据 1 个像素的宽度。所以说，如果要绘制一条真正 1 像素宽的线段，你必须将该线段绘制在某两个像素之间的那个像素中，而不能将它绘制在两个像素的交界处。在图 2-18 之中，我们可以注意到，两像素宽的那条线段，是绘制在像素交界处的，而 1 个像素宽的那条线段，则是绘制在两个像素之间的那个像素位置上的。

既然已经明白了如何绘制真正占据 1 像素的线段了，那么我们就来把这项知识运用到网格的绘制中去。

2.8.2　网格的绘制

图 2-21 展示了由应用程序所绘制的网格。

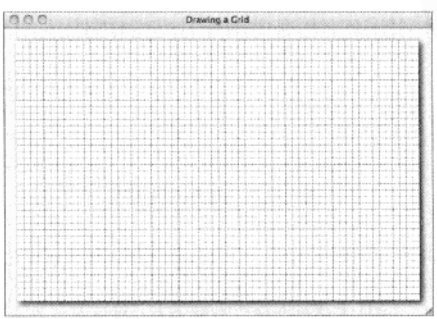


图 2-21　绘制网格

该图所示应用程序的代码列在了程序清单 2-13 之中。

程序清单 2-13　绘制网格

```
var context =document.getElementById('canvas').getContext('2d');

// Functions.....................................................

function drawGrid(context, color, stepx, stepy) {
   context.strokeStyle = color;
   context.lineWidth = 0.5;

   for (var i = stepx +  0.5; i <context.canvas.width; i += stepx) {
      context.beginPath();
      context.moveTo(i, 0);
      context.lineTo(i, context.canvas.height);
      context.stroke();
   }

   for (var i = stepy + 0.5; i <context.canvas.height; i += stepy) {
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
  }
}

// Initialization...............................................

drawGrid(context, 'lightgray', 10, 10);
```

这段 JavaScript 代码不仅像上一小节中讨论的那样，将线段绘制在了两个像素之间的像素上，而且绘制出来的线段仅有 0.5 像素宽。虽说 Canvas 规范没有明文规定，不过所有浏览器的 Canvas 实现都使用了“抗锯齿”[⊖]技术，以便创建出“亚像素”[⊜]线段的绘制效果来。

2.8.3　坐标轴的绘制

图 2-22 中所示的应用程序可以在屏幕上绘制坐标轴。该应用程序的 JavaScript 代码列在了程序清单 2-14 之中。

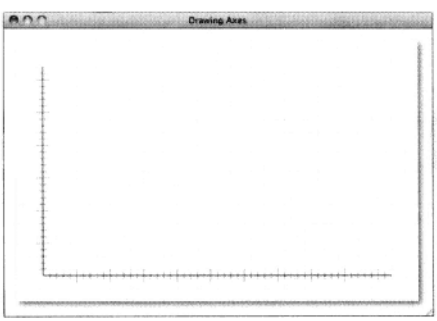


图 2-22　绘制坐标轴

⊖ anti-aliasing，又叫“反锯齿”、“反走样”，该技术对图像边缘进行柔化，使之看起来更平滑，更接近真实效果。详情参见：http://en.wikipedia.org/wiki/Supersampling。——译者注

⊜ subpixel，又叫“亚像元”，该技术将单个像素分解为更小的区域，在这些区域内分别进行渲染，以提高在LCD及OLED显示屏上的视觉分辨率。详情参见：http://en.wikipedia.org/wiki/Subpixel_rendering。——译者注

程序清单 2-14　绘制坐标轴

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    AXIS_MARGIN = 40,
    AXIS_ORIGIN = { x: AXIS_MARGIN, y:canvas.height-AXIS_MARGIN },

    AXIS_TOP   = AXIS_MARGIN,
    AXIS_RIGHT = canvas.width-AXIS_MARGIN,

    HORIZONTAL_TICK_SPACING = 10,
    VERTICAL_TICK_SPACING = 10,

    AXIS_WIDTH  = AXIS_RIGHT - AXIS_ORIGIN.x,
    AXIS_HEIGHT = AXIS_ORIGIN.y - AXIS_TOP,

    NUM_VERTICAL_TICKS   = AXIS_HEIGHT /VERTICAL_TICK_SPACING,
    NUM_HORIZONTAL_TICKS = AXIS_WIDTH  /HORIZONTAL_TICK_SPACING,

    TICK_WIDTH = 10,
    TICKS_LINEWIDTH = 0.5,
    TICKS_COLOR = 'navy',

    AXIS_LINEWIDTH = 1.0,
    AXIS_COLOR = 'blue';

// Functions.......................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function drawAxes() {
   context.save();
   context.strokeStyle = AXIS_COLOR;
   context.lineWidth = AXIS_LINEWIDTH;

   drawHorizontalAxis();
   drawVerticalAxis();

   context.lineWidth =  0.5;
   context.lineWidth = TICKS_LINEWIDTH;
   context.strokeStyle = TICKS_COLOR;

   drawVerticalAxisTicks();
   drawHorizontalAxisTicks();

   context.restore();

}

function drawHorizontalAxis() {
   context.beginPath();
   context.moveTo(AXIS_ORIGIN.x, AXIS_ORIGIN.y);
   context.lineTo(AXIS_RIGHT,    AXIS_ORIGIN.y);
   context.stroke();
}

function drawVerticalAxis() {
   context.beginPath();
   context.moveTo(AXIS_ORIGIN.x, AXIS_ORIGIN.y);
```

```
    context.lineTo(AXIS_ORIGIN.x, AXIS_TOP);
    context.stroke();
}
function drawVerticalAxisTicks() {
    var deltaY;

    for (var i=1; i < NUM_VERTICAL_TICKS; ++i) {
        context.beginPath();
        if (i % 5 === 0) deltaX = TICK_WIDTH;
        else             deltaX = TICK_WIDTH/2;

        context.moveTo(AXIS_ORIGIN.x - deltaX,
                       AXIS_ORIGIN.y - i *VERTICAL_TICK_SPACING);
        context.lineTo(AXIS_ORIGIN.x + deltaX,
                       AXIS_ORIGIN.y - i *VERTICAL_TICK_SPACING);
        context.stroke();
    }
}

function drawHorizontalAxisTicks() {
    var deltaY;

    for (var i=1; i < NUM_HORIZONTAL_TICKS; ++i) {
        context.beginPath();
        if (i % 5 === 0) deltaY = TICK_WIDTH;
        else             deltaY = TICK_WIDTH/2;

        context.moveTo(AXIS_ORIGIN.x + i *HORIZONTAL_TICK_SPACING,
                       AXIS_ORIGIN.y - deltaY);
        context.lineTo(AXIS_ORIGIN.x + i *HORIZONTAL_TICK_SPACING,
                       AXIS_ORIGIN.y + deltaY);
        context.stroke();
    }
}
// Initialization.............................................

drawGrid('lightgray', 10, 10);
drawAxes();
```

以上这段 JavaScript 代码使用常量来计算坐标轴的属性，例如坐标轴的宽度与高度，刻度线之间的距离，等等。在计算好这些属性之后，剩余部分的代码基本上都在按照 beginPath()、moveTo()、lineTo() 与 stroke() 的顺序来调用绘图环境的这些方法，以绘制出坐标轴与其上的刻度线。

既然大家已经学会了线段的绘制，那么接下来咱们看看如何让用户以互动式的方法来画线。

2.8.4 橡皮筋式的线条绘制

在本章开头讨论的那个绘画应用程序中，用户可以通过拖拽鼠标的方式在 canvas 的背景上互动式地画线。图 2-23 所示的应用程序也具备这样的功能。

图 2-23 所示应用程序的 HTML 与 JavaScript 代码分别列在了程序清单 2-15 与程序清单 2-16 之中。请注意其中以“Rubber bands ”注释所标注的那一段代码，以及鼠标事件处理器的代码。

在 onmousedown 事件的处理器中，应用程序将窗口坐标转换为 canvas 坐标，并且调用了事件对象的 preventDefault() 方法，以禁止浏览器对该事件做出默认的反应。

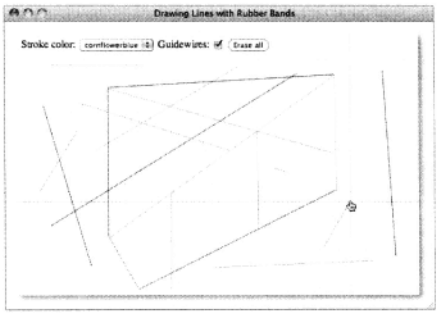


图 2-23　橡皮筋式的线条绘制

接下来，onmousedown 事件处理器会将绘图表面保存起来，记录鼠标按下事件的发生位置，并将名为 dragging 的 boolean 标志设置为 true。

程序清单 2-15　橡皮筋式的线条绘制（HTML 代码）

```
<!DOCTYPE html>
<html>
   <head>
      <title>Drawing Lines with Rubber Bands</title>

      <style>
         body {
            background: #eeeeee;
         }

         #controls {
            position: absolute;
            left: 25px;
            top: 25px;
         }

         #canvas {
            background: #ffffff;
            cursor: pointer;
            margin-left: 10px;
            margin-top: 10px;
            -webkit-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
            -moz-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
            -box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
         }
      </style>
   </head>

   <body>
      <canvas id='canvas' width='600'height='400'>
         Canvas not supported
      </canvas>

      <div id='controls'>
         Stroke color: <selectid='strokeStyleSelect'>
         <option value='red'>red</option>
         <option value='green'>green</option>
         <option value='blue'>blue</option>
         <option value='orange'>orange</option>
```

```
        <option value='cornflowerblue'selected>cornflowerblue</option>
        <option value='goldenrod'>goldenrod</option>
        <option value='navy'>navy</option>
        <option value='purple'>purple</option>
        </select>

        Guidewires:
        <input id='guidewireCheckbox'type='checkbox' checked/>
        <input id='eraseAllButton'type='button' value='Erase all'/>
      </div>

      <script src = 'example.js'></script>
   </body>
</html>
```

程序清单 2-16　橡皮筋式的线条绘制（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    eraseAllButton =document.getElementById('eraseAllButton'),
    strokeStyleSelect =document.getElementById('strokeStyleSelect'),
    guidewireCheckbox =document.getElementById('guidewireCheckbox'),
    drawingSurfaceImageData,
    mousedown = {},
    rubberbandRect = {},
    dragging = false,
    guidewires = guidewireCheckbox.checked;

// Functions.......................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top  * (canvas.height /bbox.height) };
}

// Save and restore drawing surface................................

function saveDrawingSurface() {
   drawingSurfaceImageData =context.getImageData(0, 0,
                                                 canvas.width,
                                                 canvas.height);
}

function restoreDrawingSurface() {
   context.putImageData(drawingSurfaceImageData,0, 0);
}

// Rubber bands....................................................

function updateRubberbandRectangle(loc) {
   rubberbandRect.width  = Math.abs(loc.x -mousedown.x);
   rubberbandRect.height = Math.abs(loc.y -mousedown.y);

   if (loc.x > mousedown.x) rubberbandRect.left= mousedown.x;
   else                     rubberbandRect.left= loc.x;
```

```
    if (loc.y > mousedown.y) rubberbandRect.top= mousedown.y;
    else                     rubberbandRect.top= loc.y;
}

function drawRubberbandShape(loc) {
    context.beginPath();
    context.moveTo(mousedown.x, mousedown.y);
    context.lineTo(loc.x, loc.y);
    context.stroke();
}

function updateRubberband(loc) {
    updateRubberbandRectangle(loc);
    drawRubberbandShape(loc);
}

// Guidewires.......................................................

function drawHorizontalLine (y) {
    context.beginPath();
    context.moveTo(0,y+0.5);
    context.lineTo(context.canvas.width, y+0.5);
    context.stroke();
}

function drawVerticalLine (x) {
    context.beginPath();
    context.moveTo(x+0.5,0);
    context.lineTo(x+0.5,context.canvas.height);
    context.stroke();
}

function drawGuidewires(x, y) {
    context.save();
    context.strokeStyle = 'rgba(0,0,230,0.4)';
    context.lineWidth = 0.5;
    drawVerticalLine(x);
    drawHorizontalLine(y);
    context.restore();
}

// Canvas event handlers............................................

canvas.onmousedown = function (e) {
    var loc = windowToCanvas(e.clientX,e.clientY);

    e.preventDefault(); // Prevent cursor change
    saveDrawingSurface();
    mousedown.x = loc.x;
    mousedown.y = loc.y;
    dragging = true;
};

canvas.onmousemove = function (e) {
    var loc;

    if (dragging) {
        e.preventDefault(); // Prevent selections

        loc = windowToCanvas(e.clientX,e.clientY);
        restoreDrawingSurface();
```

```
        updateRubberband(loc);

        if(guidewires) {
           drawGuidewires(loc.x, loc.y);
        }
     }
};

canvas.onmouseup = function (e) {
   loc = windowToCanvas(e.clientX, e.clientY);
   restoreDrawingSurface();
   updateRubberband(loc);
   dragging = false;
};

// Controls event handlers.....................................

eraseAllButton.onclick = function (e) {
   context.clearRect(0, 0, canvas.width,canvas.height);
   drawGrid('lightgray', 10, 10);
   saveDrawingSurface();
};

strokeStyleSelect.onchange = function (e) {
   context.strokeStyle =strokeStyleSelect.value;
};

guidewireCheckbox.onchange = function (e) {
   guidewires = guidewireCheckbox.checked;
};

// Initialization.............................................

context.strokeStyle = strokeStyleSelect.value;
drawGrid('lightgray', 10, 10);
```

然后，当用户拖动鼠标时，应用程序会维护一个名为 rubberbandRect 的矩形对象之中的属性。它用来表示橡皮筋式线条的矩形外框线。如图 2-24 所示，该矩形是由两个对角来定义的：一个是发生鼠标按下事件的位置，另一个是鼠标的当前位置。

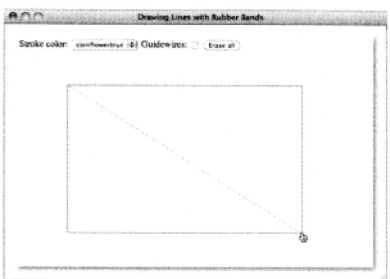


图 2-24　橡皮筋式线段的矩形外框

针对用户在拖拽鼠标过程中所发生的每一个鼠标事件，应用程序都要做如下三件事：

（1）恢复绘制表面。

（2）更新 rubberbandRect。

（3）从按下鼠标的位置向鼠标的当前位置画一条线。

应用程序的 onmousedown 事件处理器会将绘制表面保存起来，这样的话，在 onmousemove 事件处理器中进行绘制表面的恢复操作，就等于是将橡皮筋式的线段擦除了。

提示：橡皮筋式选取框在将来的用途

程序清单 2-16 之中的应用程序，在用户拖拽鼠标时维护了一个代表橡皮筋式矩形选取框的变量之中的属性。请注意，绘制橡皮筋式线条的那个函数，叫做 drawRubberbandShape()。由于应用程序维护了一个橡皮筋式的矩形选取框，所以我们可以修改 drawRubberbandShape() 方法，让它支持所有能够容纳在矩形范围内的形状，例如圆形或任意多边形。

实际上，我们在后面的几页书中就要开始重新实现 drawRubber-bandShape() 方法了。

2.8.5　虚线的绘制

在编写本书时，Canvas 的绘图环境对象尚未提供用于绘制虚线（dashed line）或者点划线（dotted line）的方法。不过，自己来实现这个功能也很简单。图 2-25 展示了一个可以绘制虚线的应用程序。

图 2-25　虚线的绘制

图 2-25 所示应用程序的代码，列在了程序清单 2-17 之中。

这段代码计算虚线的总长度，然后根据其中每条短划线（dash）的长度，算出整个虚线中应该含有多少这样的短划线。代码根据计算出的短划线数量，通过反复绘制多条很短的线段来画出整个虚线。

程序清单 2-17　虚线的绘制

```
var context =document.getElementById('canvas').getContext('2d');

function drawDashedLine(context, x1, y1, x2,y2, dashLength) {
   dashLength = dashLength === undefined ? 5 :dashLength;

   var deltaX = x2 - x1;
   var deltaY = y2 - y1;
   var numDashes = Math.floor(
       Math.sqrt(deltaX * deltaX + deltaY *deltaY) / dashLength);

   for (var i=0; i < numDashes; ++i) {
      context[ i % 2 === 0 ? 'moveTo' :'lineTo' ]
         (x1 + (deltaX / numDashes) * i, y1 +(deltaY / numDashes) * i);
   }

   context.stroke();
};

context.lineWidth = 3;
context.strokeStyle = 'blue';
drawDashedLine(context, 20, 20,context.canvas.width-20, 20);
drawDashedLine(context,context.canvas.width-20, 20,
   context.canvas.width-20,context.canvas.height-20, 10);
```

```
drawDashedLine(context,context.canvas.width-20,
   context.canvas.height-20, 20,context.canvas.height-20, 15);
drawDashedLine(context, 20,context.canvas.height-20, 20, 20, 2);
```

2.8.6 通过扩展 CanvasRenderingContext2D 来绘制虚线

上一小节所介绍的 drawDashedLine() 函数可以在某个特定的绘图环境对象之中画虚线。不过，如果你想在 Canvas 的绘图环境对象中增加一个类似 lineTo() 那样的 dashedLineTo() 方法，那么该怎么办呢？

向 Canvas 绘图环境中增加 dashedLineTo() 方法的主要障碍在于，我们没有办法获取到上一次调用 moveTo() 时所传入的位置参数。由于该参数是线段的起点，所以说，CanvasRenderingContext2D. dashedLineTo() 必须要知道这个位置才行。

尽管 Canvas 的绘图环境对象并没有提供直接的办法用以获取上次调用 moveTo() 时所用的参数，不过，你可以按照如下步骤将获得该参数值的功能加入到绘图环境对象中去：

（1）获取指向绘图环境对象中 moveTo() 方法的引用。

（2）向 Canvas 绘图环境对象中新增一个名为 lastMoveToLocation 的属性。

（3）重新定义绘图环境对象的 moveTo() 方法，将传给该方法的点保存到 lastMoveToLocation 的属性之中。

一旦取得了上次传入 moveTo() 方法中的位置参数，那么要实现新加入 CanvasRenderingContext2D 原型对象（prototype object）中的 dashedLineTo() 方法，就会变得很容易了。程序清单 2-18 列出了完成上述步骤所用的代码。

程序清单 2-18　扩展 CanvasRenderingContext2D 对象

```
var context =document.getElementById('canvas').getContext('2d'),
    moveToFunction =CanvasRenderingContext2D.prototype.moveTo;
CanvasRenderingContext2D.prototype.lastMoveToLocation= {};


CanvasRenderingContext2D.prototype.moveTo =function (x, y) {
   moveToFunction.apply(context, [x,y]);
   this.lastMoveToLocation.x = x;
   this.lastMoveToLocation.y = y;
};

CanvasRenderingContext2D.prototype.dashedLineTo =
      function (x, y, dashLength) {
   dashLength = dashLength === undefined ? 5 :dashLength;

   var startX = this.lastMoveToLocation.x;
   var startY = this.lastMoveToLocation.y;

   var deltaX = x - startX;
   var deltaY = y - startY;
   var numDashes = Math.floor(Math.sqrt(deltaX *deltaX
                              + deltaY * deltaY) /dashLength);

   for (var i=0; i < numDashes; ++i) {
      this[ i % 2 === 0 ? 'moveTo' : 'lineTo' ]
         (startX + (deltaX / numDashes) * i,
            startY + (deltaY / numDashes) * i);
   }

   this.moveTo(x, y);
};
```

按照上述方法修改 CanvasRenderingContext2D 对象之后，你就可以像这样来画虚线了：

```
context.lineWidth = 3;
context.strokeStyle = 'blue';

context.moveTo(20, 20);
context.dashedLineTo(context.canvas.width-20,20);
context.dashedLineTo(context.canvas.width-20,
                     context.canvas.height-20);
context.dashedLineTo(20,context.canvas.height-20);
context.dashedLineTo(20, 20);
context.dashedLineTo(context.canvas.width-20,
                     context.canvas.height-20);
context.stroke();
```

图 2-26 展示了上段代码的绘制效果。

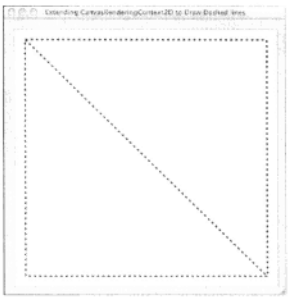


图 2-26　通过扩展 2d 绘图环境对象以实现虚线的绘制

警告：在扩展 CanvasRenderingContext2D 对象的时候要小心

尽管某些开发者认为 JavaScript 是一门玩具语言（toy language），然而事实上，正如本小节中的代码所展示的那样，它的功能还是非常强大的。本小节的代码所用到的这种技术有很多种叫法，例如“元编程”（metaprogramming）、“猴子补丁”[⊖]及“方法覆盖”[⊜]等，意思是先获取一个指向某对象方法的引用，然后重新定义那个方法，在定义新方法的时候有选择地调用原有方法。

不过，在扩展绘图环境对象的功能时，最好还是小心一些为妙。如果你按照本小节中所述的办法，在绘图环境对象中新加入了一个 drawDashedLineTo() 方法对其进行扩展，而不巧的是，将来的 CanvasRenderingContext2D 规范恰好也增加了一个 drawDashedLineTo() 方法或 lastMoveToLocation 属性，这样一来，你向绘图环境中加入的那些功能，就有可能对官方的 CanvasRenderingContext2D 规范所引入的新功能造成干扰。

提示：HTML5 的 Canvas 规范正在持续地演进

在本书出版时，对虚线的支持已经被加入到 Canvas 规范之中了。一定要牢记：HTML5 规范是在持续地演进。有鉴于此，我们最好是偶尔去看看新版本的规范都带来了哪些改动。

2.8.7　线段端点与连接点的绘制

在绘制线段时，你可以控制线段端点，也就是“线帽”（line cap）的样子，如图 2-27 所示。在 Canvas 的绘图环境对象中，控制线段端点绘制的属性正好也叫做 lineCap。

线段端点样式的默认值是 butt，也就是将端点原模原样地绘制出来。round 和 square 样式都会给线段的端点画上一顶“帽子”。round 是给端点处多画一个半圆，其半径等于线段宽度的一半，而 square 则是向端点处多画一个矩形，它的长度与线宽一致，宽度等于线宽的一半。

⊖ monkey patching，本来写作guerrilla patching，意思是“游击式修补”，而guerrilla与gorilla一词谐音，后者意为“大猩猩”，所以也被戏称作“猴子补丁”。详情参见：http://en.wikipedia.org/wiki/Monkey_patch。——译者注

⊜ clobbering methods，clobber一词在编程领域的用例请参见：http://en.wiktionary.org/wiki/clobber#Verb。——译者注

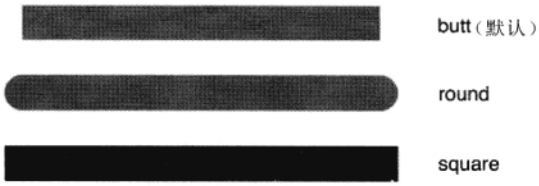


图 2-27　线段端点的绘制

如图 2-28 所示，在绘制线段或矩形时，你可以控制两条线段连接处的那个拐弯，也就是“线段连接点”（line join）应该怎么画。线段连接点的绘制是由 lineJoin 属性控制的。

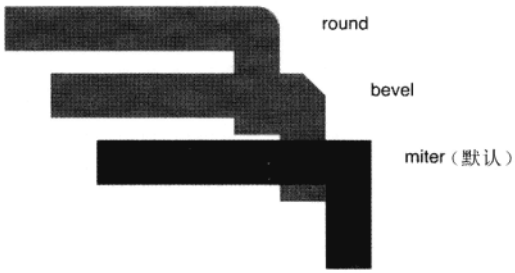


图 2-28　线段连接点的绘制

如果 lineJoin 属性设置为 bevel，那么在两个线段相交的时候，将会用一条直线来连接两个拐角外部的点，使之构成一个三角形。而如果设置为 miter，也就是 lineJoin 属性默认值的话，那么效果将与 bevel 相同，只是它还会再画一个三角形，使两个线段的接合处变为一个矩形。最后如果将 lineJoin 属性设置为 round，那么两个线段的拐角处就会画上一段填充好的圆弧。图 2-29 进一步说明了线段连接点的绘制。

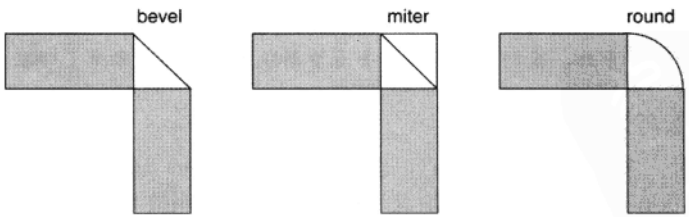


图 2-29　线段的连接点具体构建方式

当你使用 miter 样式来绘制线段连接点时，还可以指定一个 miterLimit 属性，它表示斜接线（miter）的长度与二分之一线宽的比值。斜接线长度的计量方式如图 2-30 所示。

通过图 2-30 可以看出，如果两个线段的夹角很小的话，那么斜接线的长度有可能会变得非常长。如果斜接线的长度太长，其比值（也就是斜接线长度除以二分之一的线宽）已经超过了你所指定的 miterLimit 属性的话，那么浏览器将会以 bevel 样式来处理两个线段的接合处，如图 2-31 所示。

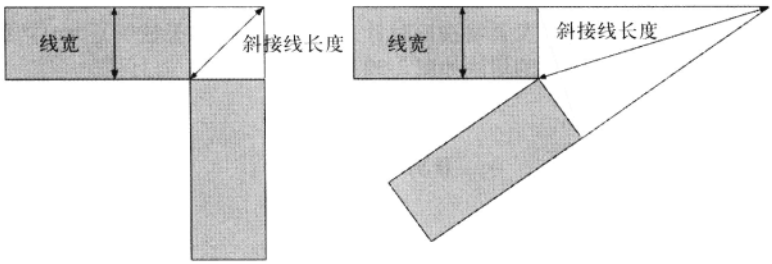


图 2-30　斜接线长度的计量方式

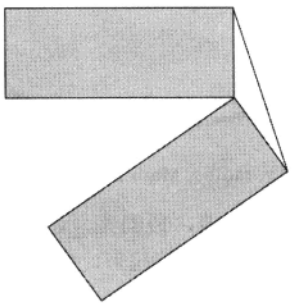


图 2-31　如果斜接线太长，浏览器就会以 bevel 样式来处理线段连接点

表 2-7 总结了 Canvas 绘图环境中与线段有关的属性。

表 2-7　CanvasRenderingContext2D 对象中与线段绘制有关的属性

属性	描述	类型	取值范围	默认值
lineWidth	以像素为单位的线段宽度	double	非零的正数	1.0
lineCap	该值决定浏览器如何绘制线段的端点	DOMString	butt、round、square	butt
lineJoin	该值决定浏览器如何绘制线段的连接点	DOMString	round、bevel、miter	bevel
miterLimit	斜接线长度与二分之一线宽的比值。如果斜接线的长度超过了该值，浏览器就会以 bevel 方式来绘制线段的连接点	double	非零的正数	10.0

2.9　圆弧与圆形的绘制

Canvas 绘图环境提供了两个用于绘制圆弧与圆形的方法：arc() 与 arcTo()。本节我们就来讲讲这两个方法。

2.9.1　arc() 方法的用法

arc() 方法有 6 个参数：arc(x, y, radius, startAngle, endAngle, counterClockwise)。前 2 个参数表示圆心坐标，第 3 个参数表示圆的半径，第 4 个和第 5 个参数分别表示浏览器在圆周上绘制圆

弧的起始角度和终结角度，最后一个参数是可选的，它代表浏览器绘制圆弧的方向。如果该值是false，也就是默认值的话，那么浏览器将按顺时针来绘制圆弧，如果该值是true，那么浏览器则会按逆时针来绘制圆弧。arc() 方法的用法如图 2-32 所示。

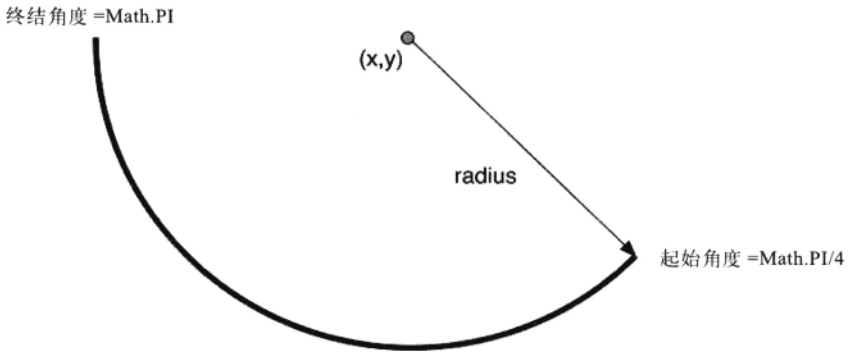


图 2-32　使用 arc(x, y, radius, Math.PI/4, Math.PI, false) 来绘制圆弧

然而，arc() 方法所绘制的可能不仅仅是一段圆弧，如果在当前路径中有子路径的话，那么浏览器会将子路径的终点与圆弧的起点用线段连接起来，如图 2-33 所示。

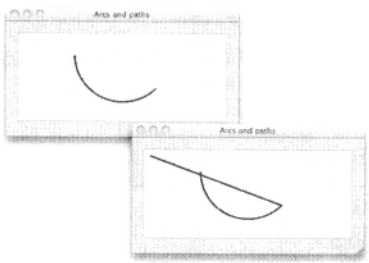


图 2-33　在清除已有子路径后绘制圆弧（上图）与不清除子路径即绘制圆弧（下图）的效果对比

该图中位于顶部的那个圆弧是用如下代码创建的：

```
context.beginPath();
context.arc(canvas.width/2, canvas.height/4,80, Math.PI/4,
            Math.PI, false);
```

在调用 arc() 之前，上述代码先调用了 beginPath() 方法。在 2.8 节中我们讲过，这个方法会清除掉当前路径中的所有子路径。

图 2-33 底部的圆弧，是通过如下代码来创建的：

```
context.beginPath();
context.moveTo(10, 10);
context.arc(canvas.width/2, canvas.height/4,80, Math.PI/4,
            Math.PI, false);
```

上述代码在调用 arc() 方法之前先调用了 moveTo() 方法。本书 2.8 节讲过，moveTo() 方法会向当前路径中加入一条仅包含一个点的子路径。在本例这种情况下，这个点就是 (10, 10)。在绘制圆弧之前，浏览器会先从此点向圆弧的起点处绘制一条线段。

2.9.2　以橡皮筋式辅助线来协助用户画圆

图 2-34 中所示的应用程序，可以让用户以拖动鼠标的方式画圆。当拖动鼠标时，该应用程序会持续地绘制圆形。

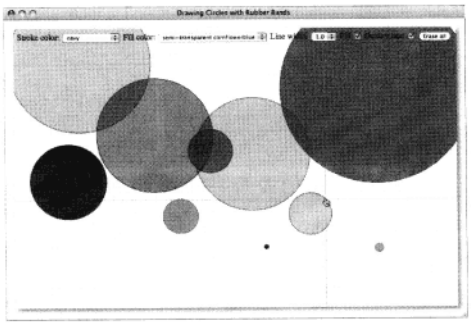


图 2-34　以橡皮筋式辅助线来协助用户画圆

在程序清单 2-16 之中，读者已经看到了绘制橡皮筋式辅助线所用的代码。回忆一下那一节之中的应用程序，当时我们说过，除了可以绘制线段之外，还可以通过重新实现 drawRubberbandShape() 函数来支持其他图形的绘制。这正是图 2-34 之中这个应用程序的实现方式。该程序的 drawRubberbandShape() 方法列在了程序清单 2-19 之中。

传递给 drawRubberbandShape() 方法的 loc 对象包含鼠标当前的 X、Y 坐标。鼠标按下事件所发生的地点记录在名为 mousedown 的变量之中，它与 loc 对象一样，也包含一对 X、Y 坐标。

应用程序计算按下鼠标的地点与当前鼠标位置之间的距离，它首先判断两者是不是在同一条水平线上[⊖]。然后应用程序的代码会根据这个距离来推算出所画圆形的半径。

程序清单 2-19　以橡皮筋式辅助线来协助用户画圆

```
function drawRubberbandShape(loc) {
   var angle,
       radius;

   if (mousedown.y === loc.y) { // Horizontal line
      // Horizontal lines are a special case. See the else
      // block for an explanation

      radius = Math.abs(loc.x - mousedown.x);
   }
   else {
      // For horizontal lines, the angle is 0, and Math.sin(0)
      // is 0, which means we would be dividing by 0 here to get NaN
      // for radius. The if block above catches horizontal lines.

      angle =Math.atan(rubberbandRect.height/rubberbandRect.width),
```

⊖ 这段代码在计算半径时所采用的算法是，先根据橡皮筋式辅助矩形的高度与宽度比值，利用反三角函数推出圆心（即起初按下鼠标的地点）和当前鼠标位置之间的连线，也就是当前这个圆的半径，与x轴的夹角。然后用橡皮筋式辅助矩形的高度除以这个夹角的正弦值，来得出最终所画圆形的半径。根据该算法，如果鼠标按下的点与当前鼠标位置在同一水平线上，则夹角为0，其正弦值也为0。这样的话算法第二步在计算圆半径的时候就会遇到用0当除数的情况，所以先要对特殊状况进行单独处理。——译者注

```
        radius = rubberbandRect.height /Math.sin(angle);
    }

    context.beginPath();
    context.arc(mousedown.x, mousedown.y, radius, 0,Math.PI*2, false);
    context.stroke();

    if (fillCheckbox.checked)
        context.fill();
}
```

警告：某些情况下不能省略可选参数

Canvas 规范说得非常清楚：arc() 方法的最后一个参数是用来决定绘制方向是顺时针还是逆时针的，它是个可选参数。这意味着针对除 Opera 浏览器之外的其他浏览器进行编程时，你不需要指定该参数的值。在写作本书时，Opera 浏览器仍然需要调用者来指定最后的那个可选参数，而且更为糟糕的是，如果你没有指定那个可选参数，那么程序的运行则会悄无声息地失败。

2.9.3 arcTo() 方法的用法

除了 arc() 之外，Canvas 的绘图环境对象还提供了另一个用于创建圆弧路径的方法，那就是 arcTo()。该方法接受 5 个参数：arcTo(x1, y1, x2, y2, radius)。

arcTo() 方法的参数分别代表两个点以及圆形的半径。该方法以指定的半径来绘制一条圆弧，此圆弧与当前点到第一个点 (x1, y1) 的连线相切，而且与第一个点到第二个点 (x2, y2) 的连线也相切[⊖]。 该方法的这些特性，使得它非常适合用来绘制矩形的圆角，如图 2-35 所示。

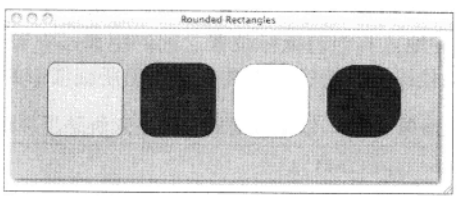


图 2-35 圆角矩形的绘制：从左至右 4 个矩形圆角半径分别是 10、20、30 与 40 像素

程序清单 2-20 列出了图 2-35 所示应用程序的代码。

程序清单 2-20 使用 arcTo() 方法

```
var context =document.getElementById('canvas').getContext('2d');

// Functions.........................................................

function roundedRect(cornerX, cornerY,
                     width, height, cornerRadius) {
   if (width > 0) context.moveTo(cornerX +cornerRadius, cornerY);
   else           context.moveTo(cornerX -cornerRadius, cornerY);

   context.arcTo(cornerX + width, cornerY,
```

⊖ 此处原书表述有误，翻译时根据Canvas规范进行了修正。详情参见：http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#dom-context-2d-arcto及http://www.w3school.com.cn/htmldom/met_canvasrenderingcontext2d_arcto.asp。后文还有几处也据此进行了修改，特此说明。——译者注

```
                cornerX + width, cornerY + height,
                cornerRadius);

   context.arcTo(cornerX + width, cornerY + height,
                cornerX, cornerY + height,
                cornerRadius);

   context.arcTo(cornerX, cornerY + height,
                cornerX, cornerY,
                cornerRadius);


   if (width > 0) {
      context.arcTo(cornerX, cornerY,
                    cornerX + cornerRadius, cornerY,
                    cornerRadius);
   }
   else {
      context.arcTo(cornerX, cornerY,
                    cornerX - cornerRadius, cornerY,
                    cornerRadius);
   }
}

function drawRoundedRect(strokeStyle, fillStyle,cornerX, cornerY,
                         width, height,cornerRadius) {
   context.beginPath();

   roundedRect(cornerX, cornerY, width, height,cornerRadius);

   context.strokeStyle = strokeStyle;
   context.fillStyle = fillStyle;

   context.stroke();
   context.fill();
}

// Initialization.............................................

drawRoundedRect('blue',   'yellow',  50,  40,  100,  100,10);
drawRoundedRect('purple', 'green',  275,  40,-100,  100, 20);
drawRoundedRect('red',    'white',  300, 140,  100,-100, 30);
drawRoundedRect('white',  'blue',   525, 140, -100,-100, 40);
```

与 arc() 方法一样，arcTo() 方法也会将当前路径中最新子路径的最后一个点与 arcTo() 方法所画圆弧的起点用线段相连。这就是为何程序清单 2-20 之中的 roundedRect() 方法并未直接绘制线段的原因。

表 2-8 总结了 arc() 与 arcTo() 方法的用法。

提示：将 roundedRect() 方法加入 CanvasRenderingContext2D 对象

你可以轻而易举地将 roundedRect() 方法加入到 Canvas 的绘图环境对象中去。不过，如果真的决定要加入的话，你应该意识到这么做是有风险的。更多与向 Canvas 绘图环境对象中新增方法及其带来的风险有关的信息，请参考本书 2.8.6 小节。

表 2-8 CanvasRenderingContext2D 对象中用于绘制圆弧及圆形的方法

方法	描述
arc(double x, double y, double radius, double startAngle, double endAngle, boolean counter-clockwise)	创建一条以 (x, y) 为圆心，以 radius 为半径，以 startAngle 与 endAngle 为起止角的圆弧路径。角的单位是弧度，不是角度（180 角度 = π 弧度）。最后一个参数是可选的。如果为 true，则按逆时针画弧，如果是 false（也就是默认值），则按顺时针画弧。 如果在调用该方法时，当前路径中有子路径存在，那么浏览器就会将子路径的终点与所画圆弧的起点以线段相连
arcTo(double x1, double y1, double x2, double y2, double radius)	参考 (x1, y1) 与 (x2, y2) 两个点，创建一条以 radius 为半径的圆弧路径。该圆弧与当前点到 (x1, y1) 点的连线相切，同时也与 (x1, y1) 到 (x2, y2) 的连线相切。 与 arc() 方法一样，如果在调用该方法时，当前路径中有子路径存在，那么浏览器将会从子路径的终点向圆弧路径的起点处画一条线段

2.9.4 刻度仪表盘的绘制

圆弧，尤其是圆，通常被用做描绘一些实物，比如在本书 1.5 节之中，我们讲了如何来实现一个圆形的钟表盘。图 2-36 所示的应用程序用 5 个圆形实现了一个仪表盘。仪表盘上的刻度代表了圆周上的角度值，在本书 2.13.1 小节中，将会用到这个仪表盘。用户可以通过它来交互式地旋转多边形物体。

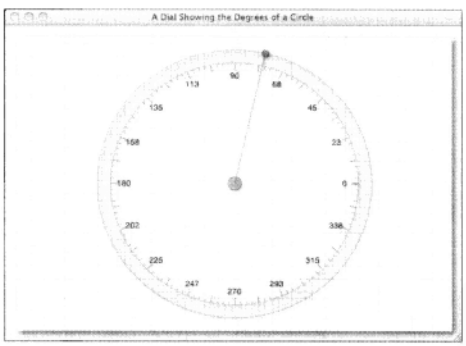


图 2-36 仪表盘的绘制

图 2-36 中的应用程序使用了本章到目前为止所讲的很多技术。为了绘制这个仪表盘，该应用程序画了许多圆形与线段，使用了各种颜色及透明度，对圆形路径进行了描边与填充，同时为了使盘面上的刻度看起来有深度感，它还运用了阴影效果。该程序还运用了 2.7.2 小节中所讲的剪纸效果，使得仪表盘外围的那一圈儿看起来有半透明的效果。

程序清单 2-21 列出了图 2-36 所示应用程序的一部分 JavaScript 代码。该程序的 drawDial() 函数通过调用其他函数来绘制仪表盘的各个部件：

```
function drawDial() {
   var loc = {x: circle.x, y: circle.y};
```

```
    drawCentroid();
    drawCentroidGuidewire(loc);

    drawRing();
    drawTickInnerCircle();
    drawTicks();
    drawAnnotations();
}
```

在浏览程序清单 2-21 时，请注意被 drawDial() 所调用的那些函数。

程序清单 2-21　仪表盘的绘制

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    CENTROID_RADIUS = 10,
    CENTROID_STROKE_STYLE = 'rgba(0,0,0,0.5)',
    CENTROID_FILL_STYLE   = 'rgba(80,190,240,0.6)',

    RING_INNER_RADIUS = 35,
    RING_OUTER_RADIUS = 55,

    ANNOTATIONS_FILL_STYLE = 'rgba(0,0,230,0.9)',
    ANNOTATIONS_TEXT_SIZE = 12,

    TICK_WIDTH = 10,
    TICK_LONG_STROKE_STYLE ='rgba(100,140,230,0.9)',
    TICK_SHORT_STROKE_STYLE ='rgba(100,140,230,0.7)',

    TRACKING_DIAL_STROKING_STYLE ='rgba(100,140,230,0.5)',

    GUIDEWIRE_STROKE_STYLE = 'goldenrod',
    GUIDEWIRE_FILL_STYLE = 'rgba(250,250,0,0.6)',

    circle = { x: canvas.width/2,
               y: canvas.height/2,
               radius: 150
             };

// Functions.......................................................

function drawGrid(color, stepx, stepy) {
   context.save()
   context.shadowColor = undefined;
   context.shadowOffsetX = 0;
   context.shadowOffsetY = 0;
   context.strokeStyle = color;
   context.fillStyle = '#ffffff';
   context.lineWidth = 0.5;
   context.fillRect(0, 0, context.canvas.width,
                          context.canvas.height);

   for (var i = stepx + 0.5;
            i < context.canvas.width; i += stepx) {
     context.beginPath();
     context.moveTo(i, 0);
     context.lineTo(i, context.canvas.height);
     context.stroke();
   }

   for (var i = stepy + 0.5;
```

```
            i < context.canvas.height; i += stepy){
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
   }
   context.restore();
}

function drawDial() {
   var loc = {x: circle.x, y: circle.y};

   drawCentroid();
   drawCentroidGuidewire(loc);
   drawRing();
   drawTickInnerCircle();
   drawTicks();
   drawAnnotations();
}

function drawCentroid() {
   context.beginPath();
   context.save();
   context.strokeStyle = CENTROID_STROKE_STYLE;
   context.fillStyle = CENTROID_FILL_STYLE;
   context.arc(circle.x, circle.y,
               CENTROID_RADIUS, 0, Math.PI*2,false);
   context.stroke();
   context.fill();
   context.restore();
}

function drawCentroidGuidewire(loc) {
   var angle = -Math.PI/4,
       radius, endpt;

  radius = circle.radius + RING_OUTER_RADIUS;

  if (loc.x >= circle.x) {
      endpt = { x: circle.x + radius *Math.cos(angle),
                y: circle.y + radius *Math.sin(angle)
      };
   }
   else {
      endpt = { x: circle.x - radius * Math.cos(angle),
                y: circle.y - radius * Math.sin(angle)
      };
   }


   context.save();

   context.strokeStyle = GUIDEWIRE_STROKE_STYLE;
   context.fillStyle = GUIDEWIRE_FILL_STYLE;

   context.beginPath();
   context.moveTo(circle.x, circle.y);
   context.lineTo(endpt.x, endpt.y);
   context.stroke();

   context.beginPath();
```

```
    context.strokeStyle = TICK_LONG_STROKE_STYLE;
    context.arc(endpt.x, endpt.y, 5, 0, Math.PI*2,false);
    context.fill();
    context.stroke();

    context.restore();
}

function drawRing() {
    drawRingOuterCircle();

    context.strokeStyle = 'rgba(0,0,0,0.1)';
    context.arc(circle.x, circle.y,
                circle.radius + RING_INNER_RADIUS,
                0, Math.PI*2, false);

    context.fillStyle = 'rgba(100,140,230,0.1)';
    context.fill();
    context.stroke();
}

function drawRingOuterCircle() {
    context.shadowColor = 'rgba(0,0,0,0.7)';
    context.shadowOffsetX = 3,
    context.shadowOffsetY = 3,
    context.shadowBlur = 6,
    context.strokeStyle =TRACKING_DIAL_STROKING_STYLE;
    context.beginPath();
    context.arc(circle.x, circle.y, circle.radius +
                RING_OUTER_RADIUS, 0, Math.PI*2,true);
    context.stroke();
}


function drawTickInnerCircle() {
    context.save();
    context.beginPath();
    context.strokeStyle = 'rgba(0,0,0,0.1)';
    context.arc(circle.x, circle.y,
                circle.radius + RING_INNER_RADIUS -TICK_WIDTH,
                0, Math.PI*2, false);
    context.stroke();
    context.restore();
}

function drawTick(angle, radius, cnt) {
    var tickWidth = cnt % 4 === 0 ? TICK_WIDTH :TICK_WIDTH/2;

    context.beginPath();
    context.moveTo(circle.x + Math.cos(angle) * (radius - tickWidth),
                   circle.y + Math.sin(angle) * (radius - tickWidth));

    context.lineTo(circle.x + Math.cos(angle) * (radius),
                   circle.y + Math.sin(angle) * (radius));
    context.strokeStyle = TICK_SHORT_STROKE_STYLE;
    context.stroke();
}

function drawTicks() {
    var radius = circle.radius + RING_INNER_RADIUS,
        ANGLE_MAX = 2*Math.PI,
```

```
        ANGLE_DELTA = Math.PI/64,
        tickWidth;

    context.save();


    for (var angle = 0, cnt = 0; angle < ANGLE_MAX;
                                 angle +=ANGLE_DELTA, cnt++) {
       drawTick(angle, radius, cnt++);
    }

    context.restore();
}

function drawAnnotations() {
    var radius = circle.radius + RING_INNER_RADIUS;

    context.save();
    context.fillStyle = ANNOTATIONS_FILL_STYLE;
    context.font = ANNOTATIONS_TEXT_SIZE + 'px Helvetica';

    for (var angle=0; angle < 2*Math.PI; angle +=Math.PI/8) {
       context.beginPath();
       context.fillText((angle * 180 /Math.PI).toFixed(0),
          circle.x + Math.cos(angle) * (radius -TICK_WIDTH*2),
          circle.y - Math.sin(angle) * (radius -TICK_WIDTH*2));
    }
    context.restore();
}

// Initialization.................................................

context.shadowColor = 'rgba(0,0,0,0.4)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur = 4;

context.textAlign = 'center';
context.textBaseline = 'middle';

drawGrid('lightgray', 10, 10);
drawDial();
```

在程序清单 2-21 的 JavaScript 代码中，有一些问题值得注意。首先，像往常一样，该应用程序在每次调用 arc() 方法之前，几乎都会调用 beginPath() 方法，以便在创建弧形路径之前先开始一段新的路径。原来说过，arc() 方法会将上一条子路径的终点与圆弧路径的起点相连，所以我们调用 beginPath()，将当前路径的所有子路径都清除，这样的话，arc() 方法就不会画出那些不美观的线段了。

该应用程序使用了绘制剪纸效果的技巧，来让表盘背景看起来有些半透明。代码调用 arc() 方法，按照顺时针方向来绘制外围的圆形，且按照逆时针方向来绘制里面的圆形。在这种情况下，为了做出剪纸效果，应用程序并没有在第二次调用 arc() 方法之前先调用 beginPath() 方法。

第二个要注意的地方是，save() 方法与 restore() 方法之间的那段代码，对绘图环境对象的某些属性做了临时性的修改，例如 strokeStyle 与 fillStyle 等。通过 Canvas 绘图环境的 save() 与 restore() 方法，你可以实现各自独立且互不干扰的绘图函数来。

最后，请注意该应用程序是如何绘制仪表盘周围文字的。先把绘图环境对象的 textAlign 与

textBaseline 属性分别设置为 center 与 middle，这样的话，应用程序就可以很容易地计算出绘制文本的位置了。我们将在 3.3.5 小节中讨论这项技术。

2.10　贝塞尔曲线

贝塞尔曲线（bézier curve）最初是由法国物理学家与数学家 Paul de Casteljau 发明的，它的广泛运用则要归功于法国工程师皮埃尔·贝塞尔（Pierre Bézier）。

贝塞尔曲线起初被用在汽车车身的设计上，现在则多用于计算机图形系统之中，例如 Adobe Illustrator、Apple 的 Cocoa 框架以及 HTML5 的 Canvas。

贝塞尔曲线分为两种：平方（quadratic）贝塞尔曲线及立方（cubic）贝塞尔曲线。平方贝塞尔曲线是一种二次曲线（second degree curve），意思就是说，它们是由三个点来定义的：两个锚点（anchor point）及一个控制点（control point）。而立方贝塞尔曲线则是一种三次曲线（third-degree curve），是由四个点来控制的：两个锚点及两个控制点。

Canvas 支持平方及立方贝塞尔曲线。在接下来的数个小节中，我们将深入讲解如何使用 Canvas 来生成这些曲线。

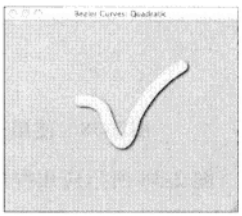


图 2-37　使用二次方贝塞尔曲线来绘制复选框标记

2.10.1　二次方贝塞尔曲线

二次方贝塞尔曲线是那种只向一个方向弯曲的简单曲线。图 2-37 所展示的是用三条二次方贝塞尔曲线所拼合而成的一个复选框（checkbox）标记。

图 2-37 所示应用程序的 JavaScript 代码，列在了程序清单 2-22 之中。

程序清单 2-22　绘制二次方贝塞尔曲线

```
var context =document.getElementById('canvas').getContext('2d');

context.fillStyle     = 'cornflowerblue';
context.strokeStyle   = 'yellow';

context.shadowColor   = 'rgba(50,50,50,1.0)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur    = 4;

context.lineWidth = 20;
context.lineCap = 'round';

context.beginPath();
context.moveTo(120.5, 130);
context.quadraticCurveTo(150.8, 130, 160.6,150.5);
context.quadraticCurveTo(190, 250.0, 210.5,160.5);
context.quadraticCurveTo(240, 100.5, 290, 70.5);

context.stroke();
```

你可以通过 quadraticCurveTo() 方法来绘制二次方贝塞尔曲线，该函数接受四个参数，分别表示两个点的 X 与 Y 坐标。第一个点是曲线的控制点，用于决定该曲线的形状，第二个点是锚点。quadraticCurveTo() 方法所绘制的贝塞尔曲线，会将锚点与当前路径中的最后一个点连接起来。

二次方贝塞尔曲线的用途很多，举例来说，图 2-38 之中的应用程序使用二次方贝塞尔曲线来绘制箭头形状的三个尖端。该应用程序还将每条曲线的控制点与锚点也标注了出来。

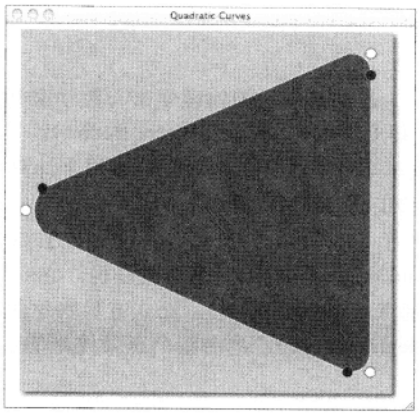


图 2-38　使用贝塞尔曲线来绘制圆角：白色的点表示控制点，深色的点表示锚点

图 2-38 所示应用程序的代码列在了程序清单 2-23 之中。

程序清单 2-23　带有圆角的箭头图案

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ARROW_MARGIN = 30,
    POINT_RADIUS = 7,
    points = [
       { x: canvas.width - ARROW_MARGIN,
         y: canvas.height - ARROW_MARGIN },

       { x: canvas.width - ARROW_MARGIN*2,
         y: canvas.height - ARROW_MARGIN },

       { x: POINT_RADIUS,
         y: canvas.height/2 },

       { x: ARROW_MARGIN,
         y: canvas.height/2 - ARROW_MARGIN },

       { x: canvas.width - ARROW_MARGIN,
         y: ARROW_MARGIN },

       { x: canvas.width - ARROW_MARGIN,
         y: ARROW_MARGIN*2 },
    ];

//Functions.............................................................

function drawPoint(x, y, strokeStyle, fillStyle) {
    context.beginPath();
    context.fillStyle = fillStyle;
    context.strokeStyle = strokeStyle;
    context.lineWidth = 0.5;
    context.arc(x, y, POINT_RADIUS, 0, Math.PI*2,false);
```

```
    context.fill();
    context.stroke();
}

function drawBezierPoints() {
    var i,
        strokeStyle,
        fillStyle;
    for (i=0; i < points.length; ++i) {
       fillStyle   = i % 2 === 0 ? 'white' :'blue',
       strokeStyle = i % 2 === 0 ? 'blue' :'white';
       drawPoint(points[i].x, points[i].y,
                 strokeStyle, fillStyle);
    }
}

function drawArrow() {
    context.strokeStyle = 'white';
    context.fillStyle = 'cornflowerblue';

    context.moveTo(canvas.width - ARROW_MARGIN,ARROW_MARGIN*2);

    context.lineTo(canvas.width - ARROW_MARGIN,
                   canvas.height -ARROW_MARGIN*2);

    context.quadraticCurveTo(points[0].x,points[0].y,
                             points[1].x,points[1].y);

    context.lineTo(ARROW_MARGIN, canvas.height/2 +ARROW_MARGIN);

    context.quadraticCurveTo(points[2].x,points[2].y,
                             points[3].x,points[3].y);

    context.lineTo(canvas.width - ARROW_MARGIN*2,ARROW_MARGIN);

    context.quadraticCurveTo(points[4].x,points[4].y,
                             points[5].x,points[5].y);
    context.fill();
    context.stroke();
}

//Initialization.....................................................

context.clearRect(0, 0, canvas.width,canvas.height);
drawArrow();
drawBezierPoints();
```

表 2-9 总结了 quadraticCurveTo() 方法的用法。

表 2-9　quadraticCurveTo() 方法的用法

方法	描述
quadraticCurveTo(double cpx, double cpy, double x, double y)	创建一条表示二次方贝塞尔曲线的路径。该方法需要传入两个点，第一个是曲线的控制点，第二个是锚点

2.10.2　三次方贝塞尔曲线

上一小节我们讲了如何创建二次方贝塞尔曲线，那些曲线都是二维的，意思是说，它们都只能向一个方向弯曲。如果需要像图 2-39 这样，能够向两个方向弯曲的曲线，那么你需要的就是三

次曲线（third-order curve），即三次方贝塞尔曲线。

图 2-39 所示应用程序使用 bezierCurveTo() 方法创建了一条代表三次方贝塞尔曲线的路径。该应用程序的代码列在了程序清单 2-24 之中。

这段代码除了绘制曲线本身，还填充了表示曲线控制点与锚点的小圆圈。

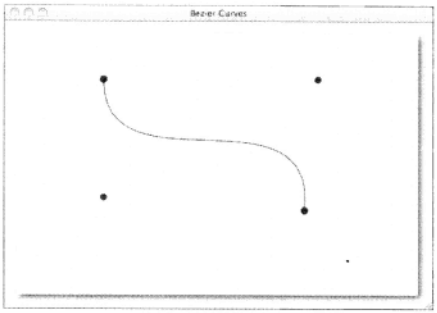


图 2-39　三次方贝塞尔曲线

程序清单 2-24　绘制三次方贝塞尔曲线

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    endPoints      = [ { x: 130, y: 70 },  { x:430, y: 270 }, ],
    controlPoints = [ { x: 130, y: 250 }, { x:450, y: 70 }, ];

//Functions.............................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function drawBezierCurve() {
   context.strokeStyle = 'blue';

   context.beginPath();
   context.moveTo(endPoints[0].x,endPoints[0].y);
   context.bezierCurveTo(controlPoints[0].x,controlPoints[0].y,
                         controlPoints[1].x,controlPoints[1].y,
                         endPoints[1].x,endPoints[1].y);
   context.stroke();
}

function drawEndPoints() {
   context.strokeStyle = 'blue';
   context.fillStyle = 'red';

   endPoints.forEach( function (point) {
      context.beginPath();
      context.arc(point.x, point.y, 5, 0,Math.PI*2, false);
      context.stroke();
      context.fill();
   });
}
```

```
function drawControlPoints() {
   context.strokeStyle = 'yellow';
   context.fillStyle = 'blue';

   controlPoints.forEach( function (point) {
      context.beginPath();
      context.arc(point.x, point.y, 5, 0,Math.PI*2, false);
      context.stroke();
      context.fill();
   });
}

//Initialization.....................................................

drawGrid('lightgray', 10, 10);

drawControlPoints();
drawEndPoints();
drawBezierCurve();
```

表 2-10 总结了 bezierCurveTo() 方法的用法。

表 2-10　bezierCurveTo() 方法的用法

方　法	描　述
bezierCurveTo(double cpx, double cpy, double cp2x,double cp2y, double x, double y)	创建一条代表三次方贝塞尔曲线的路径。你需要向该方法传入三个点的坐标，前两点是该曲线的控制点，最后一个点是锚点

2.11　多边形的绘制

现在，我们已经将 Canvas 绘图环境对象所支持的全部基本图形都讲完了，它们包括：线段、矩形、圆弧、圆形以及贝塞尔曲线。但是，我们肯定需要在 canvas 之中绘制除此以外的其他图形，比方说，三角形、六边形和八边形。在本节中，你将会学到如何通过如图 2-40 所示应用程序，对任意多边形进行描边与填充。

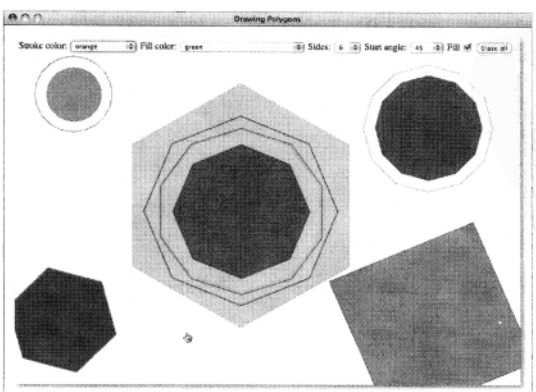


图 2-40　多边形的绘制

使用 moveTo() 与 lineTo() 方法，再结合一些简单的三角函数，就可以绘制出任意边数的多边形。图 2-41 演示了如何利用简单的三角函数绘制出多边形的某个部分。

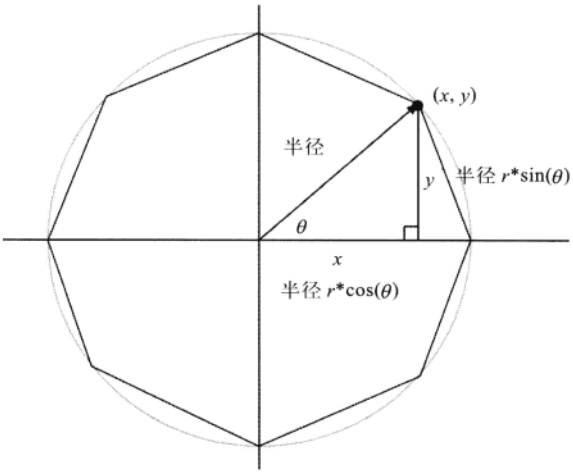


图 2-41　计算多边形的顶点坐标

图 2-41 展示了如何根据多边形外接圆的圆心及半径，来计算某个多边形的顶点。程序清单 2-25 列出了图 2-40 所示应用程序的 JavaScript 代码，这段代码演示了如何使用计算出来的顶点坐标来绘制任意的多边形。

程序清单 2-25　多边形的绘制代码（节选）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    sidesSelect =document.getElementById('sidesSelect'),
    startAngleSelect =document.getElementById('startAngleSelect'),

    fillCheckbox =document.getElementById('fillCheckbox'),

    mousedown = {},
    rubberbandRect = {},

    Point = function (x, y) {
       this.x = x;
       this.y = y;
    },

//Functions.............................................................

function getPolygonPoints(centerX, centerY, radius,sides, startAngle) {
   var points = [],
       angle = startAngle || 0;

   for (var i=0; i < sides; ++i) {
      points.push(new Point(centerX + radius *Math.sin(angle),
                            centerY - radius *Math.cos(angle)));
      angle += 2*Math.PI/sides;
```

```
    }

    return points;
}

function createPolygonPath(centerX, centerY, radius,sides, startAngle) {
    var points = getPolygonPoints(centerX, centerY,radius, sides,
                                  startAngle);
    context.beginPath();
    context.moveTo(points[0].x, points[0].y);

    for (var i=1; i < sides; ++i) {
        context.lineTo(points[i].x, points[i].y);
    }
    context.closePath();
}

function drawRubberbandShape(loc, sides, startAngle){
    createPolygonPath(mousedown.x, mousedown.y,
                      rubberbandRect.width,
                      parseInt(sidesSelect.value),
                      (Math.PI / 180) *parseInt(startAngleSelect.value));
    context.stroke();

    if (fillCheckbox.checked) {
        context.fill();
    }
}
```

程序清单 2-25 之中的代码首先获取了指向 Canvas 绘图环境对象的引用，并且定义了一个名为 Point 的对象。

getPolygonPoints() 函数创建并返回了一个含有多边形顶点的数组，这个多边形是由该函数的 5 个参数所确定的。该函数运用图 2-41 中所写的算式来计算各个顶点坐标，并创建了一个包含这些顶点的数组。

createPolygonPath() 函数调用 getPolygonPoints() 函数，以获取包含指定多边形各个顶点的数组。该函数先移动到第一个顶点，然后创建一条包含此多边形所有顶点的路径。

最后，应用程序调用 drawRubberbandShape() 函数来完成多边形的绘制，这个函数是由 2.8.4 小节中的那个 drawRubberbandShape() 函数改编而来的。图 2-40 中的应用程序使用该函数让用户可以通过拖动鼠标来交互式地绘制多边形。有关橡皮筋式图形绘制的更多信息，请参考本书 2.8.4 小节。

多边形对象

回想一下，我们原来说过，HTML5 的 Canvas 是一种立即模式绘图系统。当向 canvas 之中绘制内容时，浏览器立刻就会将这些内容画到 canvas 之中，然后立刻忘掉刚才所画的东西。如果要实现的是一个绘画应用程序的话，那么立即模式绘图系统就很合适，不过，如果要实现的是那种可以让用户创建并操作绘图对象的画图应用程序，那么最好还是将可以编辑并绘制的那些图形对象保存到一份列表之中。

在本小节中，我们将会修改上一小节讲到的那个应用程序，让其维护一份多边形对象的列表。程序清单 2-26 列出了一部分修改之后的应用程序代码。

程序清单 2-26　使用多边形对象来实现画图应用程序

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    startAngleSelect =document.getElementById('startAngleSelect'),
    sidesSelect =document.getElementById('sidesSelect'),
    ...

    mousedown = {},
    rubberbandRect = {};

function drawRubberbandShape(loc, sides, startAngle){
   var polygon = new Polygon(mousedown.x,mousedown.y,
                     rubberbandRect.width,
                     parseInt(sidesSelect.value),
                     (Math.PI / 180) *parseInt(startAngleSelect.value),
                     context.strokeStyle,
                     context.fillStyle,
                     fillCheckbox.checked);

   context.beginPath();
   polygon.createPath(context);
   polygon.stroke(context);

   if (fillCheckbox.checked) {
      polygon.fill(context);
   }
   else {
      polygons.push(polygon);
   }
}
```

应用程序在用户按下并拖动鼠标时，会调用 drawRubberbandShape() 函数，以创建多边形。此函数先创建一个表示多边形的对象，然后调用该对象的 createPath() 方法，最后对创建出的路径进行描边，必要时还会填充它。

如果用户在拖动之后将鼠标松开，那么 drawRubberbandShape() 函数就会将当前这个多边形加入到应用程序所维护的那份多边形对象列表之中。

本小节中所实现的多边形对象包含下列方法：

- points[] getPoints()
- void createPath(context)
- void stroke(context)
- void fill(context)
- void move(x, y)

程序清单 2-27 列出了 Polygon 对象的实现代码。

程序清单 2-27　多边形对象的实现代码

```
// Point constructor.....................................................

var Point = function (x, y) {
   this.x = x;
   this.y = y;
};

// Polygon constructor...................................................
```

```
var Polygon = function (centerX, centerY, radius,
              sides, startAngle, strokeStyle,fillStyle, filled) {
   this.x = centerX;
   this.y = centerY;
   this.radius = radius;
   this.sides = sides;
   this.startAngle = startAngle;
   this.strokeStyle = strokeStyle;
   this.fillStyle = fillStyle;
   this.filled = filled;
};

// Polygon prototype.................................................

Polygon.prototype = {
   getPoints: function () {
      var points = [],
          angle = this.startAngle || 0;

      for (var i=0; i < this.sides; ++i) {

         points.push(new Point(this.x + this.radius *Math.sin(angle),
                               this.y - this.radius *Math.cos(angle)));
         angle += 2*Math.PI/this.sides;
      }
      return points;
    },

    createPath: function (context) {
       var points = this.getPoints();

       context.beginPath();
       context.moveTo(points[0].x, points[0].y);

       for (var i=1; i < this.sides; ++i) {
           context.lineTo(points[i].x, points[i].y);
       }
       context.closePath();
    },

   stroke: function (context) {
      context.save();
      this.createPath(context);
      context.strokeStyle = this.strokeStyle;
      context.stroke();
      context.restore();
   },

   fill: function (context) {
      context.save();
      this.createPath(context);
      context.fillStyle = this.fillStyle;
      context.fill();
      context.restore();
   },

   move: function (x, y) {
      this.x = x;
      this.y = y;
   }
};
```

在创建多边形时，需要指定其位置。该位置指的是多边形外接圆的圆心。同时需要指定的还有：外接圆的半径、多边形的边数、多边形第一个顶点的起始角度、多边形的描边与填充风格，以及该多边形是否需要被填充。

Polygon 对象可以生成一个用以表示其顶点的数组，它可以根据这些点来创建代表此多边形的路径，也可以对该路径进行描边或填充操作。开发者可以调用其 move() 方法来移动它的位置。

2.12 高级路径操作

为了追踪所画的内容，诸如画图应用程序、计算机辅助设计系统（computer-aided design system，简称 CAD 系统）以及游戏等许多应用程序，都会维护一份包含当前显示对象的列表。通常来说，这些应用程序都允许用户对当前显示在屏幕上的物体进行操作。比方说，在 CAD 应用程序中，我们可以对设计中的元素进行选择、移动、缩放等操作。

用户一般会通过点击鼠标或触碰屏幕来选择当前显示的物体。为了方便选择功能的开发，Canvas 的 API 提供了一个名为 pointInPath() 的方法，如果某个点在当前路径中，那么该方法就返回 true。在本节中，我们将利用该方法来对 2.11.1 小节中所讨论的那个可以让用户通过拖动鼠标来绘制多边形的应用程序进行功能扩展。

在 2.12.2 小节之中，我们还将研发一款应用程序，让用户可以创建并编辑贝塞尔曲线。

2.12.1 拖动多边形对象

在本小节中，你将会学到如何将用户创建的多边形对象维护在一份多边形列表之中。这个多边形列表可以用来实现一些很有趣的功能，比如说，用户可以像图 2-42 所示那样，来拖动多边形，也可以像 2.13 节将要讲到的那样，对多边形进行旋转操作。

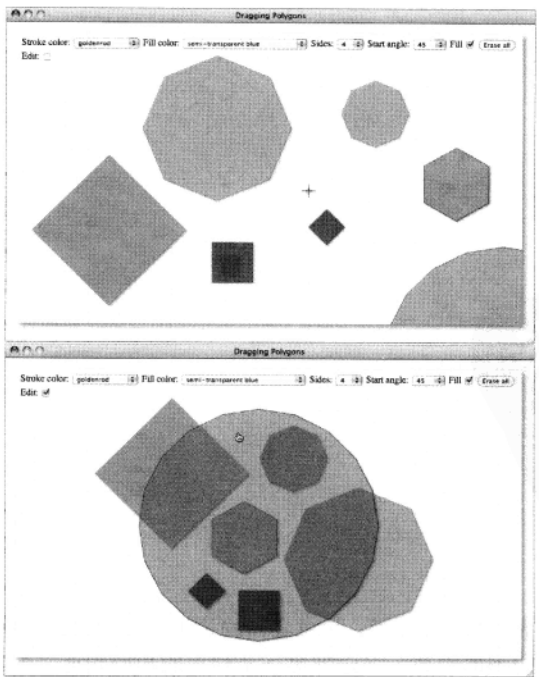


图 2-42 拖动多边形

图 2-42 中所示的应用程序有两种模式：绘制（draw）和编辑（edit）。一开始，应用程序是处于绘制模式下的，这时用户可以通过拖动鼠标来创建多边形。接下来，如果你点击“Edit”复选框，那么应用程序就会切换到编辑模式，此时用户可以拖动被创建出来的那些多边形。

该应用程序维护一份含有 Polygon 对象的数组。当在编辑模式下检测到鼠标按下事件时，应用程序会遍历这个数组，为每个多边形都创建一条路径，然后检测鼠标按下的位置是否在路径内。如果是的话，那么应用程序就会将指向该多边形的引用保存起来，同时还会保存多边形左上角与鼠标按下位置之间的 X、Y 坐标偏移量。

从这时起，应用程序中的鼠标移动事件处理器就会根据鼠标的移动来同时移动被选中的那个多边形。当用户取消对“Edit”复选框的选定后，应用程序就会返回绘制模式了。

图 2-42 之中，与拖动多边形有关的 JavaScript 代码列在了程序清单 2-28 之中。为了简洁起见，该应用程序的 HTML 代码就不再列出了。

程序清单 2-28　用于处理拖动多边形操作的 JavaScript 代码

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    eraseAllButton =document.getElementById('eraseAllButton'),
    strokeStyleSelect =document.getElementById('strokeStyleSelect'),
    fillStyleSelect =document.getElementById('fillStyleSelect'),
    fillCheckbox =document.getElementById('fillCheckbox'),
    editCheckbox =document.getElementById('editCheckbox'),
    sidesSelect =document.getElementById('sidesSelect'),

    drawingSurfaceImageData,

    mousedown = {},
    rubberbandRect = {},

    dragging = false,
    draggingOffsetX,
    draggingOffsetY,

    sides = 8,
    startAngle = 0,

    guidewires = true,
    editing = false,
    polygons = [];

//Functions.......................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top  * (canvas.height /bbox.height)
          };
}

// Save and restore drawing surface................................

function saveDrawingSurface() {
```

```
    drawingSurfaceImageData = context.getImageData(0,0,
                              canvas.width,
                              canvas.height);
}

function restoreDrawingSurface() {
    context.putImageData(drawingSurfaceImageData, 0,0);
}

// Draw a polygon.....................................................

function drawPolygon(polygon) {
    context.beginPath();
    polygon.createPath(context);
    polygon.stroke(context);

    if (fillCheckbox.checked) {
        polygon.fill(context);
    }
}

// Rubber bands.......................................................

function updateRubberbandRectangle(loc) {
    rubberbandRect.width = Math.abs(loc.x -mousedown.x);
    rubberbandRect.height = Math.abs(loc.y -mousedown.y);

    if (loc.x > mousedown.x) rubberbandRect.left =mousedown.x;
    else                     rubberbandRect.left =loc.x;

    if (loc.y > mousedown.y) rubberbandRect.top =mousedown.y;
    else                     rubberbandRect.top =loc.y;
}

function drawRubberbandShape(loc, sides, startAngle){
    var polygon = new Polygon(mousedown.x,mousedown.y,
                      rubberbandRect.width,
                      parseInt(sidesSelect.value),
                      (Math.PI / 180) *parseInt(startAngleSelect.value),
                      context.strokeStyle,
                      context.fillStyle,
                      fillCheckbox.checked);
    drawPolygon(polygon);

    if (!dragging) {
        polygons.push(polygon);
    }
}

function updateRubberband(loc, sides, startAngle) {
    updateRubberbandRectangle(loc);
    drawRubberbandShape(loc, sides, startAngle);
}

//Guidewires.........................................................

function drawHorizontalLine (y) {
    context.beginPath();
    context.moveTo(0,y+0.5);
    context.lineTo(context.canvas.width,y+0.5);
    context.stroke();
```

⋮

```
}

function drawVerticalLine (x) {
   context.beginPath();
   context.moveTo(x+0.5,0);
   context.lineTo(x+0.5,context.canvas.height);
   context.stroke();
}

function drawGuidewires(x, y) {
   context.save();
   context.strokeStyle = 'rgba(0,0,230,0.4)';
   context.lineWidth = 0.5;
   drawVerticalLine(x);
   drawHorizontalLine(y);
   context.restore();
}

function drawPolygons() {
   polygons.forEach( function (polygon) {
      drawPolygon(polygon);
   });
}

//Dragging.........................................................

function startDragging(loc) {
   saveDrawingSurface();
   mousedown.x = loc.x;
   mousedown.y = loc.y;
}

function startEditing() {
   canvas.style.cursor = 'pointer';
   editing = true;
}

function stopEditing() {
   canvas.style.cursor = 'crosshair';
   editing = false;
}

// Event handlers...................................................

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(e.clientX, e.clientY);

   e.preventDefault(); // Prevent cursor change

   if (editing) {
     polygons.forEach( function (polygon) {
        polygon.createPath(context);
        if (context.isPointInPath(loc.x, loc.y)) {
           startDragging(loc);
           dragging = polygon;
           draggingOffsetX = loc.x - polygon.x;
           draggingOffsetY = loc.y - polygon.y;
           return;
        }
     });
   }
```

```
    else {
      startDragging(loc);
      dragging = true;
    }
};

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(e.clientX, e.clientY);

   e.preventDefault(); // Prevent selections

   if (editing && dragging) {
      dragging.x = loc.x - draggingOffsetX;
      dragging.y = loc.y - draggingOffsetY;
      context.clearRect(0, 0, canvas.width,canvas.height);
      drawGrid('lightgray', 10, 10);
      drawPolygons();
   }
   else {
     if (dragging) {
        restoreDrawingSurface();
        updateRubberband(loc, sides, startAngle);

        if (guidewires) {
           drawGuidewires(mousedown.x, mousedown.y);
        }
     }
   }
};

canvas.onmouseup = function (e) {
   var loc = windowToCanvas(e.clientX, e.clientY);

   dragging = false;

   if (editing) {
   }
   else {
      restoreDrawingSurface();
      updateRubberband(loc);
   }
};

eraseAllButton.onclick = function (e) {
   context.clearRect(0, 0, canvas.width,canvas.height);
   drawGrid('lightgray', 10, 10);
   saveDrawingSurface();
};

strokeStyleSelect.onchange = function (e) {
   context.strokeStyle = strokeStyleSelect.value;
};

fillStyleSelect.onchange = function (e) {
   context.fillStyle = fillStyleSelect.value;
};

editCheckbox.onchange = function (e) {
   if (editCheckbox.checked) {
      startEditing();
   }
```

```
    else {
        stopEditing();
    }
};

//Initialization.....................................................

context.strokeStyle = strokeStyleSelect.value;
context.fillStyle = fillStyleSelect.value;

context.shadowColor = 'rgba(0,0,0,0.4)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur = 4;

drawGrid('lightgray', 10, 10);
```

2.12.2　编辑贝塞尔曲线

我们在 2.12.1 小节中学习了如何在 canvas 之中拖动图形，此项功能为许多应用程序的制作提供了可能性。例如，图 2-43 所示的应用程序可以让用户绘制贝塞尔曲线，并且在稍后通过拖动端点与控制点来编辑它们。

在图 2-43 中，上方的截图演示了如何通过拖动鼠标来绘制曲线。下方截图所演示的场景是：当用户不再拖动鼠标后，应用程序会显示如何编辑画好的曲线。在关闭了这个操作提示对话框之后，用户可以通过拖动端点或控制点来调整曲线形状，如图 2-44 中顶部的截图所示。

调整好曲线之后，在端点或控制点之外的地方点击一下，应用程序就会结束对该曲线的编辑，如图 2-44 下方的截图所示。

图 2-43 中所示应用程序的 HTML 代码列在了程序清单 2-29 之中。

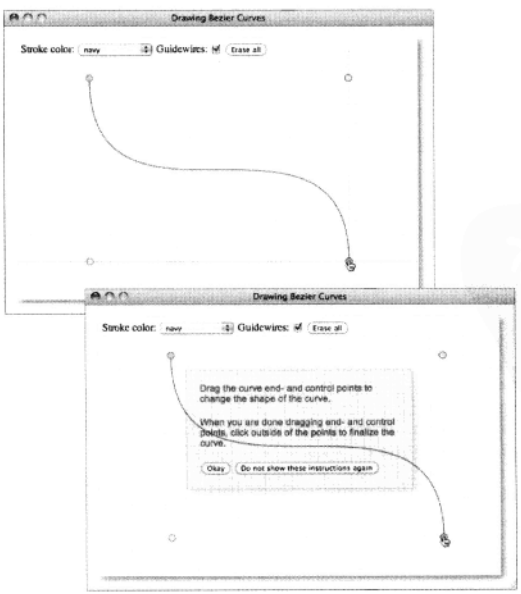


图 2-43　编辑贝塞尔曲线

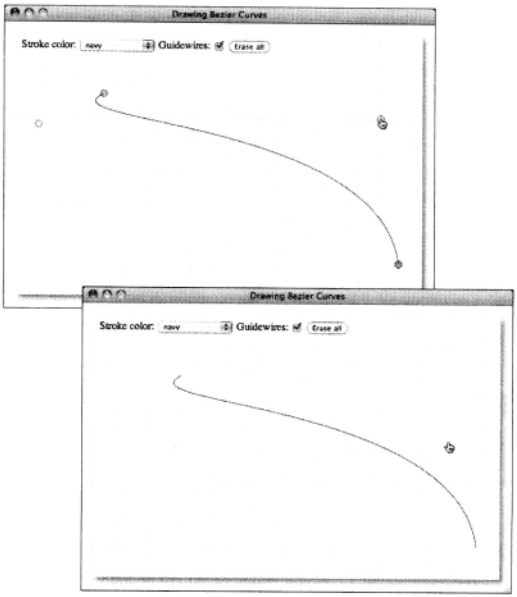


图 2-44　拖动贝塞尔曲线的端点与控制点

程序清单 2-29　通过拖动端点与控制点来编辑贝塞尔曲线（HTML 代码）

```
<!DOCTYPE html>
<html>
  <head>
    <title>Drawing Bezier Curves</title>

    <style>
      body {
         background: #eeeeee;
      }

      .floatingControls {
         position: absolute;
         left: 150px;
         top: 100px;
         width: 300px;
         padding: 20px;
         border: thin solid rgba(0,0,0,0.3);
         background: rgba(0,0,200,0.1);
         color: blue;
         font: 14px Arial;
         -webkit-box-shadow: rgba(0,0,0,0.2) 6px6px 8px;
         -moz-box-shadow: rgba(0,0,0,0.2) 6px6px 8px;
         box-shadow: rgba(0,0,0,0.2) 6px 6px 8px;
         display: none;
      }

      .floatingControls p {
         margin-top: 0px;
         margin-bottom: 20px;
```

```
      }

      #controls {
         position: absolute;
         left: 25px;
         top: 25px;
      }

       #canvas {
          background: #ffffff;
          cursor: pointer;
          margin-left: 10px;
          margin-top: 10px;
          -webkit-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
          -moz-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
          -box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
       }
    </style>
  </head>

   <body>
      <canvas id='canvas' width='605'height='400'>
         Canvas not supported
      </canvas>

   <div id='controls'>
      Stroke color:<select  id='strokeStyleSelect'>
         <option value='red'>red</option>
         <option value='green'>green</option>
         <option value='blue'>blue</option>
         <option value='orange'>orange</option>
         <option value='cornflowerblue'>cornflowerblue</option>
         <option value='goldenrod'>goldenrod</option>
         <option value='navy'selected>navy</option>
         <option value='purple'>purple</option>
         </select>
      Guidewires:
      <input id='guidewireCheckbox'type='checkbox' checked/>
      <input id='eraseAllButton' type='button'value='Erase all'/>
    </div>

    <div id='instructions'class='floatingControls'>
      <p>Drag the curve end- and control points to
         change the shape of the curve.</p>

      <p>When you are done dragging end- and control points,
         click outside of the points to finalize the curve.</p>

      <input id='instructionsOkayButton'type='button'
             value='Okay' autofocus/>
      <input id='instructionsNoMoreButton'type='button'
             value='Do not show these instructions again'/>
    </div>

    <script src = 'example.js'></script>
  </body>
</html>
```

应用程序代码使用了本书 1.8 节中所讨论的方法，将操作说明放在浮动于 canvas 之上的玻璃窗格里。包含操作说明的那个 DIV 元素，其 class 属性为 floatingControls，CSS 代码设置了该类

型元素的背景色及位置，使 DIV 元素能够浮动在 canvas 之上。

这段 HTML 也创建了用于选择描边颜色、切换辅助线显示以及擦除所有曲线所用的控件。

程序清单 2-30 列出了图 2-43 所示应用程序的 JavaScript 代码。

在浏览这部分代码时，请注意 cursorInEndPoint() 与 cursorInControlPoint() 函数。这两个函数的代码占据了整个程序清单三分之二的篇幅，它们分别用于判断用户点击鼠标的位置是否在端点或控制点之内。

同样也请注意鼠标移动事件处理器的代码。不论用户是拖动鼠标来绘制曲线，还是拖动已画好曲线的端点或控制点，只要有这种情况发生，事件处理器就会将绘图表面的内容恢复到屏幕上，如果显示辅助线功能被打开了，那么它还会向 canvas 之中绘制临时的辅助线。

如果用户正在绘制曲线，那么接下来，鼠标移动事件处理器就会重新绘制曲线本身及其端点与控制点。如果用户正在拖动端点或者控制点，那么应用程序就会更新该点的位置，并且会将曲线本身及其端点与控制点重新绘制。

程序清单 2-30　通过拖动端点与控制点来编辑贝塞尔曲线（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    eraseAllButton =document.getElementById('eraseAllButton'),
    strokeStyleSelect =document.getElementById('strokeStyleSelect'),
    guidewireCheckbox =document.getElementById('guidewireCheckbox'),
    instructions =document.getElementById('instructions'),
    instructionsOkayButton =
       document.getElementById('instructionsOkayButton'),
    instructionsNoMoreButton =
        document.getElementById('instructionsNoMoreButton'),

    showInstructions = true,

    AXIS_MARGIN = 40,
    HORIZONTAL_TICK_SPACING = 10,
    VERTICAL_TICK_SPACING = 10,
    TICK_SIZE = 10,

    AXIS_ORIGIN = { x: AXIS_MARGIN, y:canvas.height-AXIS_MARGIN },
    AXIS_TOP   = AXIS_MARGIN,

    AXIS_RIGHT = canvas.width - AXIS_MARGIN,
    AXIS_WIDTH  = AXIS_RIGHT    - AXIS_ORIGIN.x,
    AXIS_HEIGHT = AXIS_ORIGIN.y - AXIS_TOP,

    NUM_VERTICAL_TICKS   = AXIS_HEIGHT /VERTICAL_TICK_SPACING,
    NUM_HORIZONTAL_TICKS = AXIS_WIDTH  /HORIZONTAL_TICK_SPACING,

    GRID_STROKE_STYLE = 'lightblue',
    GRID_SPACING = 10,

    CONTROL_POINT_RADIUS = 5,
    CONTROL_POINT_STROKE_STYLE = 'blue',
    CONTROL_POINT_FILL_STYLE ='rgba(255,255,0,0.5)',

    END_POINT_STROKE_STYLE = 'navy',
    END_POINT_FILL_STYLE   ='rgba(0,255,0,0.5)',

    GUIDEWIRE_STROKE_STYLE ='rgba(0,0,230,0.4)',

    drawingImageData,      // Image data stored on mouse down events
```

```
    mousedown = {},          // Cursor location for last mouse down event
    rubberbandRect = {},     // Constantly updated for mouse move events

    dragging = false,        // If true, user is dragging the cursor
    draggingPoint = false,   // End- or control point user is dragging

    endPoints     = [ {}, {} ],  // Endpoint locations (x, y)
    controlPoints = [ {}, {} ],  // Control point locations (x, y)
    editing = false,             // If true, user is editing the curve

    guidewires = guidewireCheckbox.checked;

//Functions.........................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();

   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top  * (canvas.height /bbox.height)
          };
}

// Save and restore drawing surface..................................

function saveDrawingSurface() {
   drawingImageData = context.getImageData(0, 0,
                          canvas.width,canvas.height);
}

function restoreDrawingSurface() {
   context.putImageData(drawingImageData, 0, 0);
}

// Rubber bands......................................................

function updateRubberbandRectangle(loc) {
   rubberbandRect.width  = Math.abs(loc.x -mousedown.x);
   rubberbandRect.height = Math.abs(loc.y -mousedown.y);

   if (loc.x > mousedown.x) rubberbandRect.left= mousedown.x;
   else                     rubberbandRect.left= loc.x;

   if (loc.y > mousedown.y) rubberbandRect.top =mousedown.y;
   else                     rubberbandRect.top =loc.y;
}

function drawBezierCurve() {
   context.beginPath();
   context.moveTo(endPoints[0].x,endPoints[0].y);
   context.bezierCurveTo(controlPoints[0].x,controlPoints[0].y,
                         controlPoints[1].x,controlPoints[1].y,
                         endPoints[1].x,endPoints[1].y);
   context.stroke();
}

function updateEndAndControlPoints() {
   endPoints[0].x = rubberbandRect.left;
```

```
    endPoints[0].y = rubberbandRect.top;

    endPoints[1].x = rubberbandRect.left +rubberbandRect.width;
    endPoints[1].y = rubberbandRect.top  +rubberbandRect.height;

    controlPoints[0].x = rubberbandRect.left;
    controlPoints[0].y = rubberbandRect.top +rubberbandRect.height;

    controlPoints[1].x = rubberbandRect.left +rubberbandRect.width;
    controlPoints[1].y = rubberbandRect.top;
}

function drawRubberbandShape(loc) {
    updateEndAndControlPoints();
    drawBezierCurve();
}

function updateRubberband(loc) {
    updateRubberbandRectangle(loc);
    drawRubberbandShape(loc);
}

//Guidewires.......................................................

function drawHorizontalGuidewire (y) {
    context.beginPath();
    context.moveTo(0, y + 0.5);
    context.lineTo(context.canvas.width, y +0.5);
    context.stroke();
}

function drawVerticalGuidewire (x) {
    context.beginPath();
    context.moveTo(x + 0.5, 0);
    context.lineTo(x + 0.5,context.canvas.height);
    context.stroke();
}

function drawGuidewires(x, y) {
    context.save();
    context.strokeStyle = GUIDEWIRE_STROKE_STYLE;
    context.lineWidth = 0.5;
    drawVerticalGuidewire(x);
    drawHorizontalGuidewire(y);
    context.restore();
}

// Endpoints and control points....................................

function drawControlPoint(index) {
    context.beginPath();
    context.arc(controlPoints[index].x,controlPoints[index].y,
                CONTROL_POINT_RADIUS, 0,Math.PI*2, false);
    context.stroke();
    context.fill();
}
function drawControlPoints() {
    context.save();
    context.strokeStyle =CONTROL_POINT_STROKE_STYLE;
    context.fillStyle   =CONTROL_POINT_FILL_STYLE;
    drawControlPoint(0);
```

```
    drawControlPoint(1);
    context.stroke();
    context.fill();
    context.restore();
}

function  drawEndPoint(index)  {
    context.beginPath();
    context.arc(endPoints[index].x,endPoints[index].y,
                CONTROL_POINT_RADIUS,  0,Math.PI*2,  false);
    context.stroke();
    context.fill();
}

function  drawEndPoints() {
     context.save();
     context.strokeStyle =END_POINT_STROKE_STYLE;
     context.fillStyle   = END_POINT_FILL_STYLE;

     drawEndPoint(0);
     drawEndPoint(1);

     context.stroke();
     context.fill();
     context.restore();
}

function  drawControlAndEndPoints()  {
    drawControlPoints();
    drawEndPoints();
}

function  cursorInEndPoint(loc)  {
    var pt;

    endPoints.forEach(function(point)  {
       context.beginPath();
       context.arc(point.x, point.y,
                   CONTROL_POINT_RADIUS, 0,Math.PI*2,  false);

        if  (context.isPointInPath(loc.x, loc.y)){
            pt = point;
        }
     });

     return pt;
}

function cursorInControlPoint(loc)  {
    var pt;

     controlPoints.forEach( function(point) {
        context.beginPath();
        context.arc(point.x, point.y,
                    CONTROL_POINT_RADIUS, 0,Math.PI*2, false);

        if (context.isPointInPath(loc.x, loc.y)){
           pt = point;
        }
     });
     return pt;
```

```
}

function updateDraggingPoint(loc) {
   draggingPoint.x = loc.x;
   draggingPoint.y = loc.y;
}

// Canvas event handlers.............................................

canvas.onmousedown =  function (e) {
   var loc = windowToCanvas(e.clientX,e.clientY);

    e.preventDefault(); // Prevent cursor change

    if (!editing) {
       saveDrawingSurface();
       mousedown.x = loc.x;
       mousedown.y = loc.y;
       updateRubberbandRectangle(loc);
       dragging = true;
    }
    else  {
       draggingPoint =cursorInControlPoint(loc);

       if (!draggingPoint)  {
          draggingPoint = cursorInEndPoint(loc);
       }
    }
};

canvas.onmousemove = function (e) {
   var loc = windowToCanvas(e.clientX,e.clientY);

   if (dragging || draggingPoint) {
      e.preventDefault(); // Prevent selections
      restoreDrawingSurface();

      if(guidewires) {
         drawGuidewires(loc.x, loc.y);
       }
    }

    if (dragging) {
       updateRubberband(loc);
       drawControlAndEndPoints();
    }
    else if (draggingPoint) {
       updateDraggingPoint(loc);
       drawControlAndEndPoints();
       drawBezierCurve();
    }
};

canvas.onmouseup =  function  (e)  {
   loc =  windowToCanvas(e.clientX, e.clientY);

   restoreDrawingSurface();

   if (!editing) {
      updateRubberband(loc);
      drawControlAndEndPoints();
```

```
        dragging =  false;
        editing =  true;
        if  (showInstructions)  {
             instructions.style.display ='inline';
        }
    }
    else {
        if (draggingPoint)drawControlAndEndPoints();
        else                editing =  false;

        drawBezierCurve();
        draggingPoint = undefined;
    }
};

// Control event handlers.........................................

eraseAllButton.onclick =  function (e) {
    context.clearRect(0, 0, canvas.width,canvas.height);
    drawGrid(GRID_STROKE_STYLE, GRID_SPACING,GRID_SPACING);

    saveDrawingSurface();

    editing = false;
    dragging = false;
    draggingPoint = undefined;
};

strokeStyleSelect.onchange = function (e) {
    context.strokeStyle =strokeStyleSelect.value;
};

guidewireCheckbox.onchange = function  (e) {
    guidewires = guidewireCheckbox.checked;
};

// Instructions event handlers....................................

instructionsOkayButton.onclick = function (e) {
    instructions.style.display = 'none';
};

instructionsNoMoreButton.onclick = function (e){
    instructions.style.display = 'none';
    showInstructions = false;
};

//Initialization...............................................

context.strokeStyle = strokeStyleSelect.value;
drawGrid(GRID_STROKE_STYLE, GRID_SPACING,GRID_SPACING);
```

2.12.3　自动滚动网页，使某段路径所对应的元素显示在视窗中

在编写本书时，Canvas 规范在绘图环境对象中新增了一个名为 scrollPathIntoView() 的方法。该方法会让网页自行滚动，使当前路径所对应的元素显示在视窗中，不过，直到本书出版时，还未有浏览器实现它，所以书里并没有包含使用此方法的范例程序。如果将来很多浏览器都实现了这项功能，那么你就可以在本书的配套网站 http://corehtml5canvas.com 之中找到一个能够运行的范例程序了。

提示：scrollPathIntoView() 方法主要用于开发移动应用程序

Canvas 规范之中新增的 scrollPathIntoView() 方法主要用于在小屏幕的手机上进行应用程序开发。开发者可以使用这个方法让网页自行滚动，从而将屏幕外的某部分 canvas 内容显示到视窗之内。

2.13 坐标变换

正如本书 2.1 节中提到的那样，你可以对 Canvas 坐标系统进行移动、旋转、缩放等操作。在很多场合之中，我们都有足够的理由进行这些操作之中的某一个，或是同时执行它们。

将坐标原点从其默认位置屏幕左上角，移动到其他地方，通常是非常有用的。最重要的是，在计算 canvas 之中的图形与文本位置时，通过移动坐标原点，可以简化计算过程。比如，可以通过如下代码在 canvas 的中心画一个矩形：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    RECTANGLE_WIDTH = 100,
    RECTANGLE_HEIGHT = 100;

context.strokeRect(canvas.width/2 -RECTANGLE_WIDTH/2,
                   canvas.height/2  -RECTANGLE_HEIGHT/2,
                   RECTANGLE_WIDTH,RECTANGLE_HEIGHT);
```

上述代码在计算矩形左上角的 X、Y 坐标时，分别从 canvas 的中心点坐标之中减去该矩形的一半宽度及一半高度。

如果我们将坐标原点移动到刚才算出来的那个地方，那么就可以简化对 strokeRect() 方法的调用了。

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    RECTANGLE_WIDTH = 100,
    RECTANGLE_HEIGHT = 100;

context.translate(canvas.width/2 -RECTANGLE_WIDTH/2,
                  canvas.height/2 -RECTANGLE_HEIGHT/2);

context.strokeRect(0, 0, RECTANGLE_WIDTH,RECTANGLE_HEIGHT);
```

你可能会说，在这种情况下，移动坐标原点并不会简化代码，因为还是需要计算新原点的坐标，两者的唯一差别就在于刚才这段代码是将那个点的坐标传给了 translate() 方法，而不是 strokeRect() 方法。这么说也对，不过，如果你要在 canvas 的中心绘制很多图形的话，那么移动原点坐标就可以极大地简化接下来在绘制其他图形时所需的计算了。

2.13.1 坐标系的平移、缩放与旋转

图 2-45 所示的应用程序可以让用户交互式地旋转多边形，以此来演示坐标系的平移与旋转。

如果你选中“Edit”复选框并点击某个多边形的话，那么图 2-45 中的应用程序就会在该多边形周围画出一个用于表示旋转角度的仪表盘，并增加一条用于指示当前旋转角度的辅助线。有关如何实现仪表盘与辅助线的更多信息，请参阅本书 2.9.4 小节。

在选定了将要旋转的多边形之后，可以通过移动鼠标来改变旋转角度。辅助线会随着鼠标而移动，同时，应用程序还将绘制一个表示当前旋转角度的多边形来。此时，如果用户再次点击鼠标，那么应用程序就会移除多边形周围的仪表盘，并将被选中的多边形旋转到当前所选定的角度上。

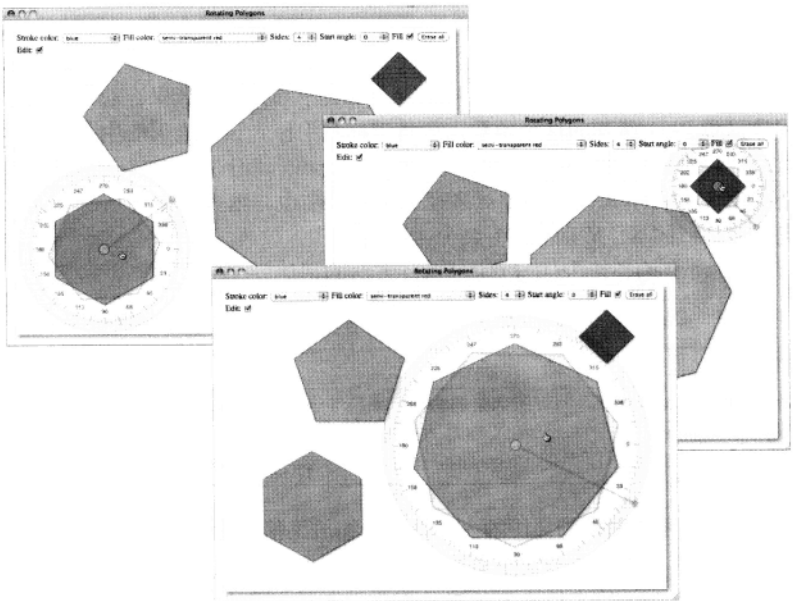


图 2-45　坐标系的平移与旋转

为了简洁起见，本书不会将图 2-45 所示应用程序的全部代码都列出来。读者可以在 http://corehtml5canvas.com 网站上运行这个范例程序并下载其源代码。程序清单 2-31 所列出的这个函数，用于在应用程序之中绘制具有某一给定旋转角度的多边形。

程序清单 2-31　坐标系的平移与旋转

```
function drawPolygon(polygon, angle) {
   var tx = polygon.x,
       ty = polygon.y;

   context.save();

   context.translate(tx, ty);

   if (angle) {
     context.rotate(angle);
   }

   polygon.x = 0;
   polygon.y = 0;

   polygon.createPath(context);
   context.stroke();

   if (fillCheckbox.checked) {
      context.fill();
   }

   context.restore();
```

```
    polygon.x = tx;
    polygon.y = ty;
}
```

图 2-45 中的应用程序所实现的多边形对象，会将其中心点的位置保存起来。为了绘制旋转后的多边形，程序清单 2-31 所列的这个函数，先将坐标系平移到该多边形的中心点，然后再旋转一定的角度。然后，应用程序的代码调用多边形对象的 createPath() 方法对多边形进行描边，如果有必要的话，还会对其进行填充。当函数对多边形完成了描边与填充操作之后，它就会将绘图环境对象的状态以及多边形的 X、Y 坐标恢复到原来的值。

表 2-11 总结了 rotate()、scale() 与 translate() 方法的用法。

表 2-11　CanvasRenderingContext2D 对象中用于平移、旋转坐标系的方法

方　法	描　述
rotate(double angleInRadians)	按照给定的角度来旋转坐标系。（注意：π 弧度等于 180 角度）
scale(double x, double y)	在 X 与 Y 方向上分别按照给定的数值来缩放坐标系
translate(double x, double y)	将坐标系平移到给定的 X、Y 坐标处

小技巧：有时在编码过程中对绘图环境的坐标系进行缩放是很有用的

如果需要在 canvas 中进行一些复杂的绘制，比如要实现像第 10 章所讲的滑动条那样的定制控件，那么最好是对 canvas 的坐标系进行缩放操作，这样才能更为清楚地看到绘制出来的内容。举例来说，可以在程序中调用 context.scale(2.5, 2.5) 语句来将当前绘制的东西放大，在结束编码之后，我们再将这行语句从代码中删掉就可以了。

如果在编码过程中对绘图环境的坐标系进行了放大，那么可能有些东西会跑到当前视窗的外面去，因为整个绘图环境是作为一个整体而被放大的。要是出现了这种情况，那么可以调用 context.translate() 方法，对绘图环境的坐标系进行临时性的平移操作，以便将你想要放大的那部分内容显示在当前视窗之中。

镜像

坐标系的变换可以用于实现很多不同的效果。比如说，在绘制了某个图形后，可以调用 scale(−1, 1) 来绘制其水平镜像（或者调用 scale(1, −1) 来绘制垂直镜像），如图 2-46 所示的应用程序那样。

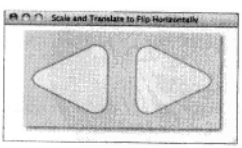


图 2-46　利用 scale() 方法来绘制水平镜像

该应用程序通过如下代码来绘制箭头形状及其水平镜像：

```
drawArrow(context);

context.translate(canvas.width, 0);
context.scale(-1, 1);

drawArrow(context);
```

上面这段代码首先调用了一个名为 drawArrow() 的方法来绘制图 2-46 左方的箭头图形。为了绘制出该图形相对于 canvas 垂直中心线的镜像，应用程序将坐标原点平移到 canvas 的右边界，然后调用 scale(−1, 1)，再按照原来的方式重新绘制箭头图形。drawArrow() 方法的实现细节请参阅本书 2.10.1 小节。

2.13.2 自定义的坐标变换

在前一小节中，读者已经看到了如何使用 scale()、rotate() 及 translate() 方法来变换坐标系，那三个方法提供了一种简便的手段，用于操作绘图环境对象的变换矩阵（transformation matrix）。不论你向 canvas 之中绘制的是图形、文本，还是图像，浏览器都会在所要绘制的物体之上运用变换矩阵。在默认情况下，这个变换矩阵就是“单位矩阵”（identity matrix），它并不会影响所要绘制的物体。当调用了 scale()、rotate() 或 translate() 方法之后，变换矩阵就会被修改，从而也会影响到所有后续的绘图操作。

在大多数情况下，这三个方法就足够用了，不过，有些时候可能需要自己直接来操作变换矩阵。比方说，如果要对所绘对象进行“错切”（shear），那么就没有办法通过组合运用这三个方法来达成此效果。在这种情况下，就必须直接操作变换矩阵了。

Canvas 的绘图环境对象提供了两个可以直接操作变换矩阵的方法：一个是 transform()，该方法可以在当前的变换矩阵之上叠加运用另外的变换效果；另一个则是 setTransform()，它会将当前的变换矩阵设置为默认的单位矩阵，然后在单位矩阵之上运用用户指定的变换效果。其要点是：多次调用 transform() 方法所造成的变换效果是累积的，而每次只要调用 setTransform() 方法，它就会将上一次的变换矩阵彻底清除。

由于 translate()、rotate() 及 scale() 这三个方法都是通过操作变换矩阵来实现其功能的，所以也可以直接用 transform() 及 setTransform() 方法来操作变换矩阵，以做出平移、旋转及缩放等效果来。直接调用 transform() 及 setTransform() 方法操作变换矩阵，有两个好处：

（1）可以做出一些诸如“错切”这样无法通过 scale()、rotate() 及 translate() 方法所达成的效果。

（2）只需调用一次 transform() 或 setTransform() 方法，就可以做出结合了缩放、旋转、平移及错切等诸多操作的效果来。

使用 transform() 及 setTransform() 方法的主要缺点则是，这两个方法不像 scale()、rotate() 及 translate() 方法那样直观。

transform() 与 setTransform() 方法均接受 6 个参数。在本小节中，你将会学到这些参数的意义，还将了解到如何通过指定它们的值来做出包含平移、缩放、旋转与错切在内的任意类型的变换操作，以改变 canvas 中所绘图形的显示效果。

2.13.2.1 坐标变换所用的代数方程

我们先来看一些用于执行平移、缩放及旋转操作所用的简单代数方程。首先，等式 2.1 列出了将平移前的旧坐标 (x, y) 换算到平移后的新坐标 (x', y') 所用的等式。

$$x' = x + dx$$

$$y' = y + dy$$

等式 2.1　计算平移后的新坐标所用的等式

在这一组等式中，新旧坐标系的横向距离差记为 dx，将其加到旧的 x 坐标值之中，就可以得出新的横坐标了；同时，新旧坐标系的纵向距离差记为 dy，将其加到旧的 y 坐标之中，就可以的出新的纵坐标了。比如，如果将原来坐标系的 (5, 10) 这个点移动到 (10, 20) 这个位置，那么坐标系的横向移动距离就是 5，而纵向移动距离则是 10，于是，我们就可以得到等式 2.2。

$$x' = 5+5$$

$$y' = 10+10$$

等式 2.2　将原有坐标系的点 (5,10) 平移到 (10, 20)

等式 2.3 可以用于换算缩放坐标系之后的新坐标。

$$x' = x \times sx$$
$$y' = y \times sy$$

等式 2.3　计算缩放之后的新坐标所用的等式

在这组等式之中，坐标轴的横向缩放倍数记为 sx，将原有 x 坐标乘以它，即可得出新的横坐标；同时，坐标轴的纵向缩放倍数记为 sy，将原有的 y 坐标乘以它，即可得出新的纵坐标。比如，要将原来坐标系中的 (5, 10) 这个点放大至 (40, 60)，那么横向放大倍数就是 8，而纵向放大倍数则是 6，于是我们就得到了等式 2.4。

$$x' = 5 \times 8$$
$$y' = 10 \times 6$$

等式 2.4　将原有坐标系的点 (5, 5) 放大至 (10, 20)

进行坐标系旋转操作的等式需要用到一些三角函数，如等式 2.5 所示。

$$x' = x \times \cos(\text{angle}) - (y \times \sin(\text{angle}))$$
$$y' = y \times \cos(\text{angle}) + (x \times \sin(\text{angle}))$$

等式 2.5　计算旋转之后的新坐标所用的等式

如果将原有坐标系中的 (5, 10) 这个点以 (0, 0) 为中心，旋转 45 度，那么将会使其落在 (3.5, 10.6) 这个位置上，如等式 2.6 所示。

$$x' = 5 \times \cos(\pi/4) - (10 \times \sin(\pi/4))$$
$$y' = 10 \times \cos(\pi/4) + (5 \times \sin(\pi/4))$$

等式 2.6　将原有坐标系的 (5, 10) 点围绕 (0, 0) 旋转 45 度

2.13.2.2　transform() 与 setTransform() 方法的用法

在掌握了用于旋转、缩放及平移坐标系所用的一些基本等式之后，咱们回头来讲讲 tranform() 与 setTransform() 方法所用的 6 个参数。这两个方法都具有如下形式的参数：

```
transform(a, b, c, d, e, f)
setTransform(a, b, c, d, e, f)
```

这 6 个参数将会在一组等式中用到，该等式可以涵盖我们已经讲过的所有用于平移、缩放及旋转的那些方程。这一组方程如等式 2.7 所示。

$$x' = ax + cy + e$$
$$y' = bx + dy + f$$

等式 2.7　用于坐标变换的通用等式

在等式 2.7 中，字母 a ～ f 分别表示 transform() 及 setTransform() 方法中所用的 6 个参数。如果 a=1, b=0, c=0 且 d=1，那么通过参数 e 和 f 就可以进行单纯的坐标系平移操作。在这种情况下，计算 x' 的等式变成了 $x' = 1 \times x + 0x + e$，而计算 y' 的等式则变成了 $y' = 0 \times x + 1y + f$，化简之后可得到等式 2.8。

$$x' = x + e$$
$$y' = y + f$$

等式 2.8　通过 transform() 与 setTransform() 方法进行单纯的坐标系平移操作所用的参数

所以说，如果要用 transform() 或 setTransform() 方法来平移坐标系的话，那么就要将第 5 个参数 e 的值设定为坐标系在 x 轴方向的偏移量，同时将第 6 个参数 f 设置为坐标系在 y 轴方向上的偏移量，并且将 a 与 d 的值设为 1，b 与 c 的值设为 0。

如果要通过 transform() 或 setTransform() 方法来缩放坐标系，那么可以将参数 a 与 d 分别设置成坐标系在 X 与 Y 轴方向上的缩放倍数，并将其他参数都设置为 0。在这种情况下，刚才讲的那组通用等式就变成了 $x' = a \times x + 0x + 0y + 0$ 与 $y' = 0 \times x + dy + 0$，化简之后可得等式 2.9。

$$x' = ax$$
$$y' = dy$$

等式 2.9　通过 transform() 与 setTransform() 方法进行坐标系缩放所用的参数

如果要将坐标系围绕原点旋转一定的弧度，则使用如下参数来调用 transform() 与 setTranform() 方法：a=cos(angle)，b=sin(angle)，c=-sin(angle)，d=cos(angle)，e=0，f=0，如等式 2.10 所示。

$$x' = \cos(angle) \times x - \sin(angle) \times y + 0$$
$$y' = \sin(angle) \times x + \cos(angle) \times y + 0$$

等式 2.10　通过 transform() 与 setTransform() 方法进行坐标系旋转所用的参数

2.13.2.3　通过 transfrom() 与 setTranform() 方法进行坐标系的平移、旋转与缩放

图 2-47 所示的应用程序会使用如下代码，对文本进行旋转与缩放：

```
context.clearRect(-origin.x, -origin.y,canvas.width, canvas.height);
context.rotate(clockwise ? angle :  -angle);
context.scale(scale, scale);
drawText();
```

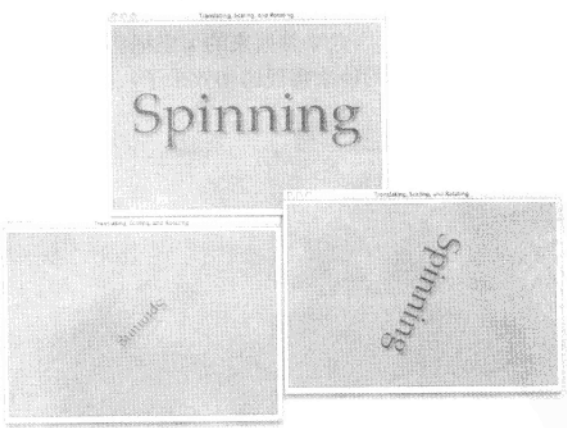


图 2-47　旋转的文本

也可以用 transform() 方法来实现同样的效果，代码如下：

```
var sin = clockwise ? Math.sin(angle) :Math.sin(-angle),
    cos = clockwise ? Math.cos(angle) :Math.cos(-angle);

if (!paused) {
```

```
    context.clearRect(-origin.x,  -origin.y,
                      canvas.width, canvas.height);
    context.transform(cos, sin, -sin, cos, 0, 0);
    context.transform(scale, 0, 0, scale, 0, 0);
    drawText();
}
```

在上面这段代码中，第一次调用 transform() 方法是为了旋转坐标系，而第二次调用它则是为了对坐标系进行缩放。

也可以将两个 transform() 方法调用合并起来，如下所示：

```
context.transform(scale*cos, sin, -sin, scale*cos,0, 0);
```

本小节讲述了如何使用 transform() 与 setTransform() 方法对坐标系进行平移、缩放及旋转，这些操作也可以分别通过 translate()、scale() 及 rotate() 方法来完成。现在我们再来用 transform() 及 setTransform() 方法做一些无法通过 translate()、scale() 及 rotate() 方法所完成的变换效果。

2.13.2.4 错切

等式 2.11 再次列出了刚才在 2.13.2.2 小节中讲到的那个坐标变换通用等式。

$$x' = ax+cy+e$$
$$y' = bx+dy+f$$

等式 2.11　再次列出用于坐标变换的通用等式

再回想一下，等式 2.11 之中，字母 a ～ f 分别对应于 transform() 与 setTransform() 方法的 6 个参数：

```
transform(a, b, c, d, e, f)
setTransform(a, b, c, d, e, f)
```

现在请注意等式 2.11 之中的 c 与 b。在计算 x' 的值时，需要使用 c 与 y 的乘积，同时，在计算 y' 的值时，也要用到 b 与 x 的乘积。这意味着原来的 x 坐标将会影响到变换之后的 y 坐标，反之亦然。于是，我们就可以通过 c 与 b 的值来实现错切效果了，如图 2-48 所示。

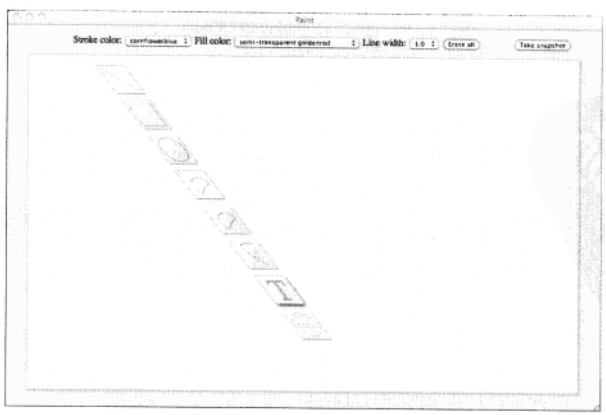


图 2-48　使用错切操作来实现具有 3D 效果的浮动图标

图 2-48 所展示的是一个简单的绘画应用程序，它单独使用一个 canvas 元素来绘制程序所用

到的图标。在绘制这些图标之前，应用程序首先使用如下代码来对 canvas 的绘图环境对象进行坐标变换：

```
controlsContext.transform(1, 0, 0.75, 1, 0, 0);
```

如果把上述 6 个参数代入等式 2.11，那么将会得到两个式子：$x' = 1 \times x + 0.75 \times y + 0$ 与 $y' = 0 \times x + 1 \times y + 0$，化简之后如等式 2.12 所示：

$$x' = x + 0.75y$$
$$y' = y$$

等式 2.12　制作水平方向错切效果所用的等式

上述等式对所绘图标中每个点的 x 坐标进行了错切操作，同时保持 y 坐标不变，于是就产生了图 2-48 所示的效果。表 2-12 总结了 transform() 与 setTransform() 方法的用法。

提示：浮动图标的绘制

图 2-48 之中的图标看起来好像是浮在绘制表面的上方。本书 1.8 节讨论了如何创建具有浮动效果的控件。

表 2-12　CanvasRenderingContext2D 对象之中用于坐标变换的方法

方　法	描　述
transform(double a, double b, double c, double d, double e, double f)	按照这 6 个参数所代表的矩阵对当前坐标系进行变换
setTransform(double a, double b, double c, double d, double e, double f)	先将当前的变换矩阵重置为单位矩阵，然后再按照这 6 个参数所代表的矩阵对坐标系进行变换

2.14　图像合成

在默认情况下，如果在 canvas 之中将某个物体（源）绘制在另一个物体（目标）之上，那么浏览器就会简单地把源物体的图像叠放在目标物体的图像上面。这种图像合成（compositing）行为一点也不奇怪，你要知道，如果在一张纸上将某个东西画在另一个之上，那么也会产生同样的效果。

然而，可以通过设置 Canvas 绘图对象的 globalCompositeOperation 属性来改变默认的图像合成行为，该属性可以取表 2-13 中所列出的任意一个值。这些值叫做 Porter-Duff 操作符，它们描述在一篇由 LucasFilm Ltd.[⊖]的 Thomas Porter 与 Tom Duff 所写的文章中，该文发表在 1984 年 7 月的《Computer Graphics》杂志上。你可以在 http://keithp.com/~keithp/porterduff/p253-porter.pdf 读到这篇文章。

表 2-13 除了列出所有 globalCompositeOperation 属性可以取的值，还演示了在每种合成模式下，源图像是如何与目标图像进行合成的。在演示图样中，圆形代表源物体，正方形代表目标物体。该表着重强调了默认的图像合成模式 source-over。

为了演示 globalCompositeOperation 属性的用法，图 2-49 之中的应用程序绘制了一个跟随鼠标移动的橙色圆形。

⊖ Lucasfilm Ltd.，中文叫做“卢卡斯影业有限公司”，是乔治•卢卡斯于1971年建立的美国电影公司，最著名的作品是星球大战系列影片。该公司是电影行业视觉特效、声音特效和计算机动画的业界领袖。想了解更多详情，请参阅：http://zh.wikipedia.org/zh-cn/卢卡斯影业。——译者注

表 2-13 CanvasRenderingContext2D 对象所支持的图像合成操作

合成模式	样例	合成模式	样例
source-atop		source-in	
source-out		source-over	
destination-atop		destination-in	
destination-out		destination-over	
lighter		copy	
xor			

正如图 2-49 所演示的那样，我们可以通过 globalCompositeOperation 属性来实现各种各样的特效。最右方的截图展示了使用 lighter 合成操作的效果，该合成操作使得橙色的圆形在文本上移动的时候，看起来像是一盏聚光灯。

程序清单 2-32 列出了图 2-49 所示的应用程序的 HTML 代码。

这段 HTML 代码创建了一个用于选择 globalCompositeOperation 属性值的 select 元素，同时还创建了一个 canvas 元素。在 HTML 代码之中，我们通过 CSS 将 canvas 元素放在 select 元素的右方。

程序清单 2-32 图像合成模式演示程序（HTML 代码）

```
<!DOCTYPE html>
<html>
   <head>
      <title>Canvas Composite Operations</title>

      <style>
         #canvas {
            border: 1px solid cornflowerblue;
            position: absolute;
            left: 150px;
```

```
            top: 10px;
            background: #eeeeee;
            border: thin solid #aaaaaa;
            cursor: pointer;
            -webkit-box-shadow:rgba(200,200,255,0.9) 5px 5px 10px;
            -moz-box-shadow:rgba(200,200,255,0.9) 5px 5px 10px;
            box-shadow: rgba(200,200,255,0.9) 5px 5px 10px;
         }
      </style>
   </head>
   <body>
      <select id='compositingSelect' size='11'>
         <option value='source-atop'>source-atop</option>
         <option value='source-in'>source-in</option>
         <option value='source-out'>source-out</option>
         <option value='source-over'>source-over (default)</option>
         <option value='destination-atop'>destination-atop</option>
         <option value='destination-in'>destination-in</option>
         <option value='destination-out'>destination-out</option>
         <option value='destination-over'>destination-over</option>
         <option value='lighter'>lighter</option>
         <option value='copy'>copy</option>
         <option value='xor'>xor</option>
      </select>
      <canvas id='canvas' width='600'height='420'>
         Canvas not supported
      </canvas>
      <script src='example.js'></script>
   </body>
</html>
```

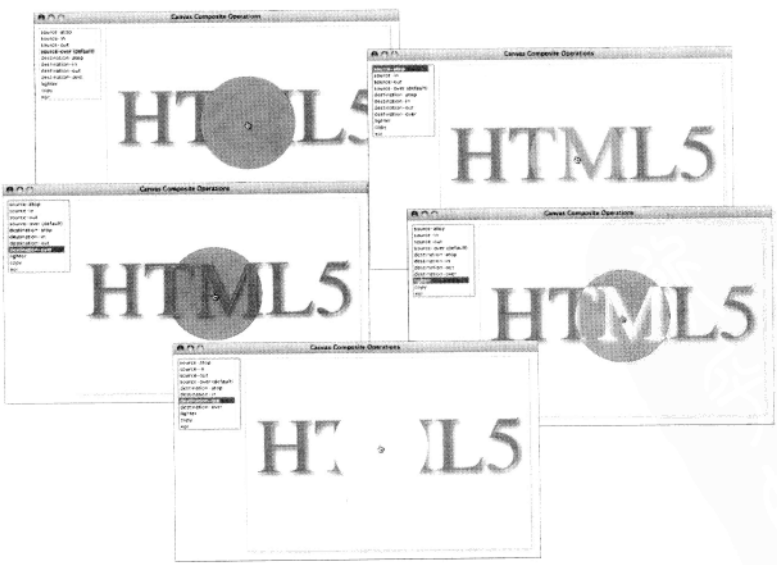


图 2-49　图像合成模式。从上方按顺时针顺序依次为：source-over、source-top、lighter、destination-out 和 destination-over

图 2-49 所示应用程序的 JavaScript 代码列在了程序清单 2-33 之中。

程序清单 2-33　图像合成模式演示程序（JavaScript 代码）

```
var context =document.getElementById('canvas').getContext('2d'),
    selectElement =document.getElementById('compositingSelect');

// Functions.....................................................

function drawText() {
   context.save();

   context.shadowColor   = 'rgba(100, 100, 150, 0.8)';
   context.shadowOffsetX = 5;
   context.shadowOffsetY = 5;
   context.shadowBlur    = 10;
   context.fillStyle = 'cornflowerblue';

   context.fillText('HTML5', 20, 250);

   context.strokeStyle = 'yellow';
   context.strokeText('HTML5', 20, 250);

   context.restore();
}

// Event handlers...............................................

function windowToCanvas(canvas, x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top  * (canvas.height /bbox.height)
          };
}

context.canvas.onmousemove = function(e) {
   var loc = windowToCanvas(context.canvas,e.clientX, e.clientY);
   context.clearRect(0, 0, context.canvas.width,
                     context.canvas.height);
   drawText();

   context.save();
   context.globalCompositeOperation =selectElement.value;
   context.beginPath();
   context.arc(loc.x, loc.y, 100, 0, Math.PI*2,false);
   context.fillStyle = 'orange';
   context.stroke();
   context.fill();

   context.restore();
}

// Initialization...............................................

selectElement.selectedIndex = 3;
context.lineWidth = 0.5;
context.font = '128pt Comic-sans';
drawText();
```

上面这段 JavaScript 代码实现了一个鼠标移动事件处理器，它会持续地绘制一个随着用户鼠标而移动的橙色圆形。该处理器的代码根据 compositingSelect 元素的值来设定 Canvas 绘图环境对象的 globalCompositeOperation 属性。

顺便说一句，drawText() 方法会临时性地开启阴影效果。在绘制文本时，它会将绘制代码夹在

save() 与 restore() 方法之间。这样的话，绘制出来的文本就会带有阴影了，而橙色的圆形则不会有。

关于图像合成效果的争论

在编写本书时，浏览器厂商尚未对如何实现 5 种 globalCompositeOperation 合成操作的效果达成一致。表 2-14 列出了这 5 个值，还分别列出了 Safari、Chrome 浏览器与 Firefox、Opera 浏览器的实现效果。

表 2-14　不可移植的图像合成操作模式

合成模式	Chrome与Safari的绘制效果	FireFox与Opera的绘制效果
source-in		
source-out		
destination-in		
destination-atop		
copy		

之所以要讲解各种浏览器对于这些合成效果的不同实现方式，意思是想要让你明白，表 2-14 中所列的这些合成模式是无法移植的。如果你对这两种不同的实现方式之中的技术细节不感兴趣，那么请直接跳到下一节。

Chrome 与 Safari 浏览器所使用的实现方式叫做“局部合成”（local compositing），意思是在执行合成操作时，只考虑构成源图像的那些像素。而另一方面，Firefox 与 Opera 浏览器则使用“全局合成”（global compositing）方式，它们在执行合成操作时，要考虑到 canvas 剪辑区域内的所有像素。

从表 2-14 之中，可以明显地看到 Chrome 与 Safari 浏览器所使用的局部合成方式与 Firefox 及 Opera 浏览器所使用的全局合成方式之间的差别。局部合成方式并不影响目标图像，然而，全局合成方式则要将目标图像之中位于源图像范围以外的那部分擦除。

在本书编写之时，Canvas 规范所规定的合成方式是全局合成，也就是 Firefox 和 Opera 浏览器所用的那种方式。不过，该规范在将来很有可能将其修改为 Chrome 与 Safari 浏览器所使用的局部合成方式。

2.15 剪辑区域

本节要讨论的内容可以说是 Canvas 之中最为有用的一个功能：剪辑区域（clipping region）。它是在 canvas 之中由路径所定义的一块区域，浏览器会将所有的绘图操作都限制在本区域内执行。在默认情况下，剪辑区域的大小与 canvas 一致。除非你通过创建路径并调用 Canvas 绘图环境对象的 clip() 方法来显式地设定剪辑区域，否则默认的剪辑区域不会影响 canvas 之中所绘制的内容。然而，一旦设置好剪辑区域，那么你在 canvas 之中绘制的所有内容都将局限在该区域内，这意味着在剪辑区域以外进行绘制是没有任何效果的。

在本节中，我们将通过两个例子来演示剪辑区域的使用方法。第一个例子实现了一个橡皮擦，而第二个例子则实现了“伸缩式动画”（telescoping animation）效果。

提示：Canvas 元素中的“瑞士军刀”

剪辑区域可以说是 Canvas 元素中的瑞士军刀[⊖]，因为你可以用它做出各种各样的效果来。在本书接下来的这部分内容里面，你将多次看到剪辑区域的运用，这其中也包括 4.10 节之中所实现的放大镜程序。 在第 10 章之中，你还会看到如何利用剪辑区域技术来实现一个可以在 canvas 中查看大型图片的查看器。

2.15.1 通过剪辑区域来擦除图像

图 2-50 所示的应用程序利用剪辑区域技术实现了一个橡皮擦。

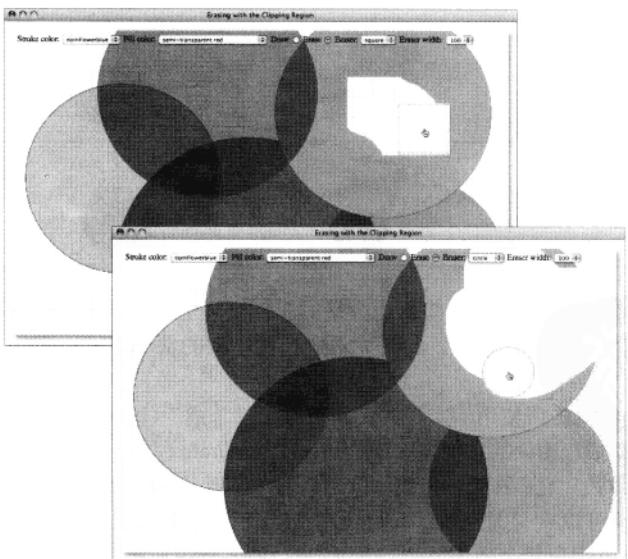


图 2-50 利用剪辑区域技术来擦除图像

⊖ Swiss Army knife，常称为万用刀，是含有许多工具在一个刀身上的折叠小刀，由于瑞士军方为士兵配备这类工具刀而得名。详情参见http://zh.wikipedia.org/zh-cn/瑞士军刀。——译者注

橡皮擦的使用方式很简单：在 canvas 之中拖动鼠标时，应用程序就会擦除鼠标所在位置周围的圆形或矩形区域之中的内容。橡皮擦的形状取决于用户在“Eraser”下拉列表框中所选取的值。

橡皮擦的实现也很简单：如果它是矩形的，那么在用户拖动鼠标时，应用程序就会将剪辑区域设置为鼠标周围的那个矩形区域，然后调用 clearRect(0, 0, canvas.width, canvas.height) 方法。在这一章以及整本书之中，读者将会多次看到：调用之前必须设置剪辑区域，否则 clearRect() 方法就会将整个 canvas 的内容都清除掉。如果先设置好了剪辑区域，然后再调用 clearRect() 方法，那么该方法所擦除的区域就会仅仅局限在剪辑区域之内，这样的话，我们就实现了一个橡皮擦。

图 2-50 所示应用程序的全部 JavaScript 代码都列在了程序清单 2-34 之中。

程序清单 2-34　使用剪辑区域来实现橡皮擦

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    strokeStyleSelect =document.getElementById('strokeStyleSelect'),
    fillStyleSelect =document.getElementById('fillStyleSelect'),
    drawRadio =document.getElementById('drawRadio'),
    eraserRadio =document.getElementById('eraserRadio'),
    eraserShapeSelect =document.getElementById('eraserShapeSelect'),
    eraserWidthSelect =document.getElementById('eraserWidthSelect'),

    ERASER_LINE_WIDTH = 1,
    ERASER_SHADOW_COLOR = 'rgb(0,0,0)',


    ERASER_SHADOW_STYLE  = 'blue',
    ERASER_STROKE_STYLE  = 'rgb(0,0,255)',
    ERASER_SHADOW_OFFSET = -5,
    ERASER_SHADOW_BLUR   = 20,

    GRID_HORIZONTAL_SPACING = 10,
    GRID_VERTICAL_SPACING = 10,
    GRID_LINE_COLOR = 'lightblue',
    drawingSurfaceImageData,

    lastX,
    lastY,
    mousedown = {},
    rubberbandRect = {},
    dragging = false,
    guidewires = true;

// Functions.......................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for a complete listing.
}

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  /bbox.width),
            y: y - bbox.top * (canvas.height /bbox.height)
}

// Save and restore drawing surface................................

function saveDrawingSurface() {
```

```
    drawingSurfaceImageData =context.getImageData(0, 0,
                              canvas.width,
                              canvas.height);
}

function restoreDrawingSurface() {
   context.putImageData(drawingSurfaceImageData, 0,0);
}

// Rubber bands.......................................................

function updateRubberbandRectangle(loc) {
   // Listing omitted for brevity. See Example 2.16
   // for a complete listing.
}

function drawRubberbandShape(loc) {
   var angle =Math.atan(rubberbandRect.height/rubberbandRect.width),
       radius = rubberbandRect.height /Math.sin(angle);

   if (mousedown.y === loc.y) {
      radius = Math.abs(loc.x - mousedown.x);
   }

   context.beginPath();
   context.arc(mousedown.x, mousedown.y, radius, 0,Math.PI*2, false);
   context.stroke();
   context.fill();
}

function updateRubberband(loc) {
   updateRubberbandRectangle(loc);
   drawRubberbandShape(loc);
}

// Guidewires.........................................................

function drawGuidewires(x, y) {
   // Listing omitted for brevity. See Example 2.16
   // for a complete listing.
}

// Eraser.............................................................

function setDrawPathForEraser(loc) {
   var eraserWidth =parseFloat(eraserWidthSelect.value);

   context.beginPath();

   if (eraserShapeSelect.value === 'circle') {
      context.arc(loc.x, loc.y,
                  eraserWidth/2,
                  0, Math.PI*2, false);
   }
   else {
      context.rect(loc.x - eraserWidth/2,
                   loc.y - eraserWidth/2,
                   eraserWidth, eraserWidth);
   }
   context.clip();
}
```

```
function setErasePathForEraser() {
   var eraserWidth =parseFloat(eraserWidthSelect.value);

   context.beginPath();

   if (eraserShapeSelect.value === 'circle') {
      context.arc(lastX, lastY,
                  eraserWidth/2 +ERASER_LINE_WIDTH,
                  0, Math.PI*2, false);
   }
   else {
   context.rect(lastX - eraserWidth/2 -ERASER_LINE_WIDTH,
                lastY - eraserWidth/2 -ERASER_LINE_WIDTH,
                eraserWidth + ERASER_LINE_WIDTH*2,
                eraserWidth + ERASER_LINE_WIDTH*2);
   }
   context.clip();
}

function setEraserAttributes() {
  context.lineWidth     = ERASER_LINE_WIDTH;
  context.shadowColor   = ERASER_SHADOW_STYLE;
  context.shadowOffsetX = ERASER_SHADOW_OFFSET;
  context.shadowOffsetY = ERASER_SHADOW_OFFSET;
  context.shadowBlur    = ERASER_SHADOW_BLUR;
  context.strokeStyle   = ERASER_STROKE_STYLE;
}

function eraseLast() {
   context.save();

   setErasePathForEraser();
   drawGrid(GRID_LINE_COLOR,
            GRID_HORIZONTAL_SPACING,
            GRID_VERTICAL_SPACING);

   context.restore();
}

function drawEraser(loc) {
   context.save();

   setEraserAttributes();
   setDrawPathForEraser(loc);
   context.stroke();

   context.restore();
}

// Canvas event handlers.......................................

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(e.clientX, e.clientY);

   e.preventDefault(); // Prevent cursor change

   if (drawRadio.checked) {
      saveDrawingSurface();
   }

   mousedown.x = loc.x;
```

```
    mousedown.y = loc.y;

    lastX = loc.x;
    lastY = loc.y;

    dragging = true;
};

canvas.onmousemove = function (e) {
    var loc;

    if (dragging) {
        e.preventDefault(); // Prevent selections

        loc = windowToCanvas(e.clientX, e.clientY);

        if (drawRadio.checked) {
            restoreDrawingSurface();
            updateRubberband(loc);

            if(guidewires) {
                drawGuidewires(loc.x, loc.y);
            }
        }
        else {
            eraseLast();
            drawEraser(loc);
        }
        lastX = loc.x;
        lastY = loc.y;
    }
};

canvas.onmouseup = function (e) {
    loc = windowToCanvas(e.clientX, e.clientY);

    if (drawRadio.checked) {
        restoreDrawingSurface();
        updateRubberband(loc);
    }

    if (eraserRadio.checked) {
        eraseLast();
    }

    dragging = false;
};

// Controls event handlers.....................................

strokeStyleSelect.onchange = function (e) {
    context.strokeStyle = strokeStyleSelect.value;
};

fillStyleSelect.onchange = function (e) {
    context.fillStyle = fillStyleSelect.value;
};

//Initialization.............................................

context.strokeStyle = strokeStyleSelect.value;
```

```
context.fillStyle = fillStyleSelect.value;
drawGrid(GRID_LINE_COLOR,
         GRID_HORIZONTAL_SPACING,
         GRID_VERTICAL_SPACING);
```

咱们看看上述程序清单之中鼠标移动事件处理器的代码。当用户拖动鼠标时，该事件处理器会将上次橡皮擦所在区域的内容擦除，同时在当前位置绘制橡皮擦的图样。该应用程序先将剪辑区域设置为橡皮擦的路径，然后重绘整个背景，以此来完成图像的擦除。

调用 clip() 方法，将会把剪辑区域设置为当前剪辑区域与当前路径的交集，了解这一点非常重要。如果橡皮擦是矩形的，那么在注释掉程序清单 2-34 之中对 save() 与 restore() 方法的调用之后，就会发现，不管用户在 canvas 里面怎么用力地拖动鼠标，也不管拖动了多长时间，所擦除的区域总是一开始那块 60 像素宽 40 像素高的矩形。之所以会这样，是因为第一次调用 clip() 方法的时候会将剪辑区域设定成起初那个矩形，而以后调用 clip() 方法的时候，所定义的剪辑区域总是局限于起初的那个矩形范围之内。

从上一段的讨论中我们看到，clip() 方法总是在上一次的剪辑区域基础上进行操作的，所以说，我们很少看到哪一次对 clip() 方法的调用没有被嵌入到 save() 与 restore() 方法之中。

既然读者已经对剪辑区域有了充分的理解，那么咱们再来看一个利用剪辑区域来实现动画效果的范例程序。

2.15.2　利用剪辑区域来制作伸缩式动画

图 2-51 所示的应用程序实现了伸缩式动画效果。顶部的那张截图是应用程序刚启动时的样子。从顶端开始，按顺时针顺序，其他截图依次展示了应用程序通过操作剪辑区域来逐渐“吞食”文本的过程。

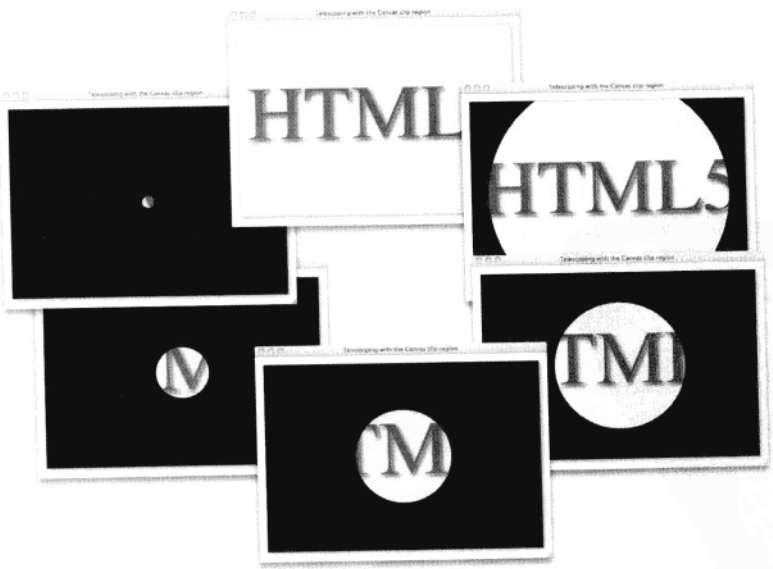


图 2-51　利用剪辑区域来实现伸缩式动画

等到 canvas 全部变黑之后，应用程序又会将其回复到初始状态，于是又回到顶部那张截图的样子。

图 2-51 中所示应用程序的 JavaScript 代码列在了程序清单 2-35 之中。所有的操作都在 animate() 函数之内完成。

onmousedown 事件处理器的代码会调用 animate() 函数，该函数以每秒刷新 60 次的帧速率执行 50 次[⊖]。 每次执行循环时，animate() 函数都会用深褐色（charcoal）来填充整个 canvas，并绘制动画当前帧。在绘制每一帧动画时，都使用淡灰色来填充 canvas，并绘制含有“HTML5”字样的文本，这些绘制操作都局限在伸缩式镜头（telescope）的范围内。

程序清单 2-35　利用剪辑区域来实现伸缩式动画

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

// Functions.......................................................

function drawText() {
   context.save();
   context.shadowColor = 'rgba(100,100,150,0.8)';
   context.shadowOffsetX = 5;
   context.shadowOffsetY = 5;
   context.shadowBlur = 10;

   context.fillStyle = 'cornflowerblue';
   context.fillText('HTML5', 20, 250);
   context.strokeStyle = 'yellow';
   context.strokeText('HTML5', 20, 250);
   context.restore();
}

function setClippingRegion(radius) {
   context.beginPath();
   context.arc(canvas.width/2, canvas.height/2,
               radius, 0, Math.PI*2, false);
   context.clip();
}

function fillCanvas(color) {
   context.fillStyle = color;
   context.fillRect(0, 0, canvas.width,canvas.height);
}

function endAnimation(loop) {
   clearInterval(loop);

   setTimeout( function (e) {
      context.clearRect(0, 0, canvas.width,canvas.height);
      drawText();
   }, 1000);
}

function drawAnimationFrame(radius) {
   setClippingRegion(radius);
   fillCanvas('lightgray');
   drawText();
```

⊖ 原书此处为100，但从源代码来看，应是50，故这里改为50。——译者注

```
}

function animate() {
   var radius = canvas.width/2,
       loop;

   loop = window.setInterval(function() {
        radius -= canvas.width/100;

        fillCanvas('charcoal');

        if (radius > 0)  {
           context.save();
           drawAnimationFrame(radius);
           context.restore();
        }
        else {
           endAnimation(loop);
        }
     }, 16);
  };

// Event handlers.....................................................

canvas.onmousedown = function (e) {
   animate();
};

// Initialization....................................................

context.lineWidth = 0.5;
context.font = '128pt Comic-sans';
drawText();
```

表 2-15 总结了 clip() 方法的用法。

表 2-15　clip() 方法的用法

方　法	描　述
clip()	将剪辑区域设置为当前剪辑区域与当前路径的交集。在第一次调用 clip() 方法之前，剪辑区域的大小与整个 canvas 一致。 因为 clip() 方法会将剪辑区域设置为当前剪辑区域与当前路径的交集，所以对该方法的调用一般都是嵌入 save() 与 restore() 方法之间的。否则，剪辑区域将会越变越小，这通常不是我们想要的效果

2.16　总结

本章深入讲解了如何在 canvas 之中进行绘制。首先，讨论了坐标系统以及 Canvas 的绘制模型，然后，我们了解到如何绘制简单的矩形，如何指定纯色和透明色，如何使用渐变色与填充图案，以及如何运用阴影效果等。

其后，我们又讲了路径和子路径，还有描边、填充等操作。此外，我们也学习了 Canvas 在填充有交叉的路径时所依据的非零环绕规则，同时还学会了如何运用这项知识来实现剪纸效果。

在讲完上述内容之后，接下来我们关注了线段的绘制问题，学到了如何绘制宽度刚好为 1 像

素的线段，以及如何绘制看起来比 1 像素还要细的线段等技术。读者学到了如何使用线段来绘制网格与坐标轴，还学会了如何让用户得以交互式地绘制橡皮筋式线段。在这之后，读者又学到了如何绘制 Canvas 绘图环境对象尚未明确支持的虚线，并且学会了如何对 Canvas 的绘图环境对象进行扩展，为其添加显式的虚线支持。在本节最后，我们又研究了线帽与线段连接点等属性，它们分别决定了 Canvas 绘图环境对象绘制线段的两个端点及交点方式。

学习完线段的绘制之后，我们开始讲解圆弧与圆形的绘制。读者看到了如何让用户以拖动鼠标的方式来互动式地创建圆形，此外还学到了如何使用 arcTo() 方法来绘制圆角矩形，以及如何实现仪表盘与其上的数值刻度。

讲完了圆弧与圆形的绘制以后，我们接下来又讲了平方贝塞尔曲线及立方贝塞尔曲线的绘制，并且教会大家如何使用这些类型的曲线来实现复选框标记与箭头图案。然后，我们讲了多边形的绘制，编写了一个 Polygon 对象，并且使用 Canvas 绘图环境对象的 isPointInPath() 方法来实现了多边形对象的拖放操作。而且，读者还学到了如何使用 isPointInPath() 方法来实现一个可以创建贝塞尔曲线的互动式编辑器。

学完了上述内容后，我们学习了坐标变换技术，这其中包括如何对 Canvas 的坐标系进行平移、旋转及缩放。读者也看到了如何创建诸如错切这样的自定义变换效果。

在本章最后，我们学习了图像合成技术，该技术决定了 Canvas 如何将某个图形绘制在另一个图形之上。我们以对剪辑区域的讲解来结束这一章的内容，该技术可谓 Canvas 的“瑞士军刀”，读者学会了如何使用剪辑区域来实现橡皮擦及伸缩式动画效果。

此时，你已经学会了如何将想要绘制的东西画在 canvas 之中。在接下来的数章中，我们将会讲解如何把本章所学的知识运用到图像、动画、精灵、物理学、碰撞检测、游戏开发，以及自定义控件的实现之中，我们还会用本章学到的知识操作正在 canvas 中播放的视频文件之中的帧。此外，我们也会研究如何用 Canvas 来制作可以运行在智能手机或平板电脑之上的移动应用程序。

Canvas 所提供的这套功能强大的绘制 API，是基于一些诸如 Adobe Illustrator 及 Apple 的 Cocoa 等成熟的图形系统之上的。在本书下面的内容中，我们将继续研究这套 API。

第3章
文　本

几乎每个基于 Canvas 的应用程序都要同文本打交道。某些应用程序仅仅是配置并显示文本而已，然而另一些应用程序则会支持复杂的文本编辑。

canvas 元素只支持极少数的文本操作功能，在写作本书时，它尚未提供那些在很多基础的文本编辑器中都能找到的功能，例如文本选择、复制与粘贴，以及文本滚动等。不过，它也提供了一些必备的基本功能，例如文本的描边与填充，向 canvas 之中放置文本，以及用像素为单位来计算任意字符串的宽度等。Canvas 的绘图环境对象提供了如下 3 个与文本有关的方法：

- strokeText(text, x, y)
- fillText(text, x, y)
- measureText(text)

measureText() 方法所返回的对象中，包含一个名为 width 的属性，它的值代表你传递给该方法的文本所占据的宽度。Canvas 的绘图环境对象中有三个与文本有关的属性：

- font
- textAlign
- textBaseline

用户可以通过 font 属性来设置稍后绘制时所使用的字型，而 textAlign 与 textBaseline 属性则可以让用户设置文本在 canvas 之内的定位方式。现在我们就来详细地讲讲这些方法和属性。

3.1　文本的描边与填充

图 3-1 中所示的应用程序演示了文本的描边与填充效果。

图 3-1　文本的描边与填充

该应用程序提供了一些复选框，让用户可以通过它们来控制是否要对所绘文本进行描边、填充及运用阴影效果。

为了简洁起见，本书省略了图 3-1 所示应用程序的 HTML 代码。那段代码创建了复选框控件以及 canvas 元素，并将应用程序的 JavaScript 代码含入其中。该段 JavaScript 代码列在了程序清单 3-1 之中。

该段 JavaScript 代码获取了指向那三个复选框对象的引用，并分别向每个控件中增加了一个用于绘制背景及文本的 onchange 事件处理器。

此范例应用程序分别使用 fillText() 与 strokeText() 方法来对文本进行填充与描边操作。这两个方法都接受三个参数，第一个参数是所要绘制的文本，剩下两个参数用来指定文本的绘制位置。所画文本的精确位置，要取决于 textAlign 与 textBaseline 属性。在 3.3 节之中，将会讨论这两个属性。

程序清单 3-1　文本的描边与填充

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    fillCheckbox   = document.getElementById('fillCheckbox'),
    strokeCheckbox = document.getElementById('strokeCheckbox'),
    shadowCheckbox = document.getElementById('shadowCheckbox'),
    text='HTML5';

// Functions.........................................................

function draw() {
   context.clearRect(0, 0, canvas.width, canvas.height);
   drawBackground();

   if (shadowCheckbox.checked)  turnShadowsOn();
   else                         turnShadowsOff();

   drawText();
}

function drawBackground() {  // Ruled paper
   var STEP_Y =  12,
       TOP_MARGIN = STEP_Y * 4,
       LEFT_MARGIN = STEP_Y * 3,
       i = context.canvas.height;

   // Horizontal lines

   context.strokeStyle = 'lightgray';
   context.lineWidth = 0.5;

   while(i > TOP_MARGIN) {
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
      i -= STEP_Y;
  }

  // Vertical line

  context.strokeStyle = 'rgba(100,0,0,0.3)';
  context.lineWidth = 1;
  context.beginPath();
```

```
  context.moveTo(LEFT_MARGIN,0);
  context.lineTo(LEFT_MARGIN,context.canvas.height);
  context.stroke();
}

function turnShadowsOn() {
   context.shadowColor = 'rgba(0,0,0,0.8)';
   context.shadowOffsetX = 5;
   context.shadowOffsetY = 5;
   context.shadowBlur    = 10;
}

function turnShadowsOff() {
   context.shadowColor = undefined;
   context.shadowOffsetX = 0;
   context.shadowOffsetY = 0;
   context.shadowBlur    = 0;
}

function drawText() {
   var TEXT_X = 65,
       TEXT_Y = canvas.height/2 + 35;

   context.strokeStyle = 'blue';

   if (fillCheckbox.checked)   context.fillText  (text, TEXT_X, TEXT_Y);
   if (strokeCheckbox.checked) context.strokeText(text, TEXT_X, TEXT_Y);
}

// Event handlers.....................................................

fillCheckbox.onchange   = draw;
strokeCheckbox.onchange = draw;
shadowCheckbox.onchange = draw;

// Initialization....................................................

context.font = '128px Palatino';
context.lineWidth = 1.0;
context.fillStyle = 'cornflowerblue';

turnShadowsOn();
draw();
```

fillText() 与 strokeText() 方法还会接受一个可选的第 4 参数，该参数以像素为单位指定了所绘文本的最大宽度。图 3-2 顶部那张截图所演示的情况，是图 3-1 所示的应用程序在正常地绘制文本，而下方的截图所演示的情况，则是应用程序在调用 strokeText() 与 fillText() 方法时，通过可选的第 4 参数来限制所绘文本的最大宽度。

如果在调用 strokeText() 或 fillText() 方法时通过可选的第 4 参数来指定所绘文本的最大宽度，而绘制的文本又超过了此宽度，那么，Canvas 规范则要求浏览器缩小文本的尺寸，使之符合调用方法时所指定的最大宽度。浏览器可以变更字型的大小或横向缩小文本，但是不管采用哪种方式，必须要让缩小后的文本仍然可读才行。

在本书成稿时，各浏览器对 maxWidth 参数的支持情况不大一致。Safari 和 Chrome 浏览器都还没有支持这个参数，而 Firefox 浏览器自从 5.0 版本起就开始支持它了，正如图 3-2 所证实的那样，Internet Explorer 浏览器从 IE9 版本起，也支持此参数了。

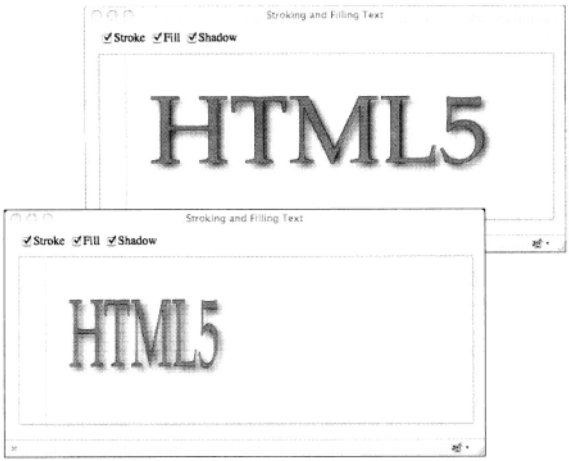


图 3-2　在绘制文本时限定其最大宽度

像绘制图形时一样，在对文本进行填充及描边操作时，除了使用纯色，还可以使用图案及渐变色，如图 3-3 所示。

图 3-3 所示应用程序的 JavaScript 代码列在了程序清单 3-2 之中。

正像本书 2.5.1.1 小节中所做的那样，程序清单 3-2 的代码创建了线性渐变色对象及填充图案对象。在绘制图 3-3 顶部的那行文本之前，应用程序先将填充样式属性设置为刚创建好的那个渐变色对象，同理，在绘制底部的文本前，应用程序也会先将填充样式属性设置为刚创建好的填充图案对象。

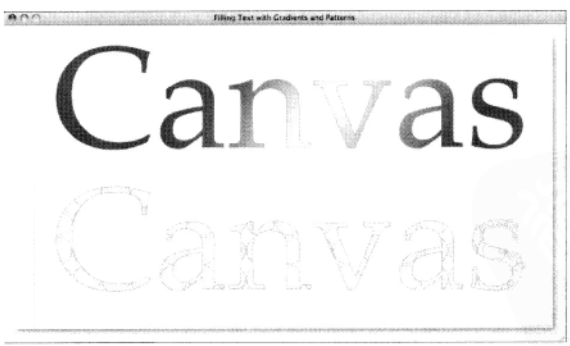


图 3-3　使用图案及渐变色来填充文本

程序清单 3-2　使用渐变色及图案来填充文本

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    image = new Image(),
    gradient = context.createLinearGradient(0, 0,
                canvas.width, canvas.height),
    text = 'Canvas',
```

```
    pattern; // Create pattern after image loads

// Functions.........................................................

function drawBackground() {
   // Listing omitted for brevity. See Example 3.1
   // for a complete listing.
}

function drawGradientText() {
   context.fillStyle = gradient;
   context.fillText(text, 65, 200);
   context.strokeText(text, 65, 200);
}

function drawPatternText() {
   context.fillStyle = pattern;
   context.fillText(text, 65, 450);
   context.strokeText(text, 65, 450);
}

// Event handlers....................................................

image.onload = function (e) {
   pattern = context.createPattern(image, 'repeat');
   drawPatternText();
};

// Initialization...................................................

image.src = 'redball.png';

context.font = '256px Palatino';
context.strokeStyle = 'cornflowerblue';

context.shadowColor   = 'rgba(100,100,150,0.8)';
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur    = 10;

gradient.addColorStop(0,    'blue');
gradient.addColorStop(0.25,'blue');
gradient.addColorStop(0.5,  'white');
gradient.addColorStop(0.75, 'red');
gradient.addColorStop(1.0,  'yellow');

drawBackground();
drawGradientText();
```

读者已经学会了如何对文本进行描边与填充，现在咱们看看怎样来设置字型属性。

3.2　设置字型属性

可以通过绘图环境对象的 font 属性，来设置绘制在 canvas 之中的文本所采用的字型。该属性是一个 CSS3 格式的字型字符串，它的各个分量如表 3-1 所示。开发者在设置绘图环境的 font 属性时，需要按照该表从上至下所列的顺序来依次指定这些分量的值。

Canvas 默认的字型是 10px sans-serif。font-style、font-variant 与 font-weight 的默认值均为 normal。

图 3-4 中所示的应用程序采用不同的字型来填充文本。

该应用程序首先把每个要绘制的字符串设置为绘图环境对象的 font 值，然后再调用绘图环境对象的 fillText() 方法将其打印到屏幕上，于是就生成了图 3-4 中所看到的这些字符串了。

表 3-1 font 属性的各个分量

字型属性分量	有效取值
font-style	可以取如下三个值：normal、italic、oblique
font-variant	可以取这两个值：normal、small-caps
font-weight	决定该字型的字符笔画粗细，可取如下值：normal、bold、bolder（比基准字型的笔画略粗一级）、lighter（比基准字型的笔画略细一级）、100、200、300……900。其中 normal 相当于数值 400，而 bold 则相当于数值 700
font-size	字型的大小，可取如下值：xx-small、x-small、medium、large、x-large、xx-large、smaller、larger、length[⊖] 与 %[⊜]
line-height	浏览器会将该属性强制设定为其默认值 normal，如果你设置了该值，浏览器会忽略你所设定的值
font-family	可以用两种方式来设置字体集（font family），一种方式是以“family-name”格式来指定此属性，这时可取的值为 helvetica、verdana、palatino 等等；另一种方式是以“generic-family”格式来指定此属性，其值可取 serif、sans-serif、monospace、cursive 及 fantasy 等等。在设置 font-family 分量的时候，可以仅以 family-name 或 generic-family 格式来指定其值，也可以同时使用这两种格式

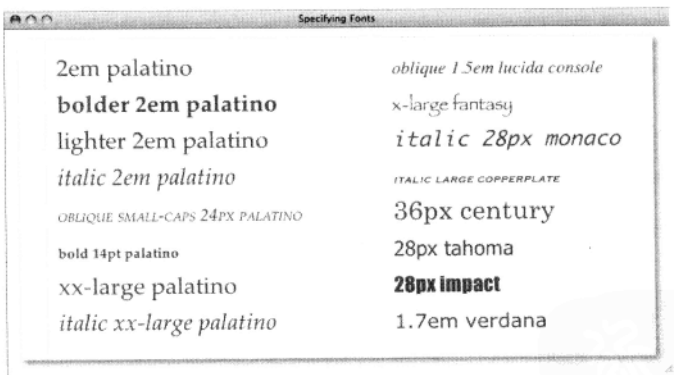


图 3-4 以各种指定的字体来绘制文本

左边这一列字符串所使用的字型都是 Palatino 字体集的变种，而右边一列则演示了一些“网页安全字型”（web-safe font）的绘制效果。

该应用程序内所用的所有字型都是网页安全的，这些字体本身并没有什么危险性。“网页安全”这个叫法所包含的意思是，这些字型已经被广泛地使用在 Windows、Mac 与 Linux 系统之中。因为它们的应用范围较广，所以你可以理所当然地认为这些字型在三大主流操作系统的所有浏览

⊖ 使用px、cm等单位的固定量。——译者注

⊜ 依照父元素的字型大小百分比。——译者注

器上都能够被正确地渲染。

图 3-4 中所示应用程序的代码列在了程序清单 3-3 之中。

程序清单 3-3　设置 font 属性

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    LEFT_COLUMN_FONTS = [
       '2em palatino',                      'bolder 2em palatino',
       'lighter 2em palatino',              'italic 2em palatino',
       'oblique small-caps 24px palatino',  'bold 14pt palatino',
       'xx-large palatino',                 'italic xx-large palatino'
    ],
    RIGHT_COLUMN_FONTS = [
       'oblique 1.5em lucida console',      'x-large fantasy',
       'italic 28px monaco',                'italic large copperplate',
       '36px century',                      '28px tahoma',
       '28px impact',                       '1.7em verdana'
    ],

    LEFT_COLUMN_X = 25,
    RIGHT_COLUMN_X = 425,
    DELTA_Y = 50,
    TOP_Y = 50,
    y = 0;

context.fillStyle = 'blue';

LEFT_COLUMN_FONTS.forEach( function (font) {
   context.font = font;
   context.fillText(font, LEFT_COLUMN_X, y +=DELTA_Y);
});

y = 0;

RIGHT_COLUMN_FONTS.forEach( function (font) {
   context.font = font;
   context.fillText(font, RIGHT_COLUMN_X, y +=DELTA_Y);
});
```

程序清单 3-3 之中的代码，先设置了 font 属性的值，然后再将其打印到屏幕上。请注意，该应用程序在每次循环中都会将赋予 font 属性的字符串照原样打印到屏幕上，这么做完全说得通。

然而，如果 font 属性的取值无效的话，那么浏览器就不会修改该属性的值，而会保持其原有值不变。比方说，你在指定 font-style 或 font-family 等分量值的时候弄错了顺序，或是将一个非法值指定给了 font-style 分量。

提示：通过 CSS3 与 Canvas 来指定字型属性时的区别

绘制环境对象的 font 属性也支持 CSS3 格式的字型语法，除了样式表（stylesheet）语法所特有的属性，例如 inherit 或 initial 等。如果你不巧刚好用到了 inherit 或 initial 的话，那么浏览器在执行到那行代码时会悄然地失败，并不抛出任何异常，同时也不会将该值设定给 font 属性。

通过 Canvas 来设置字型属性与通过 CSS3 来设置相比，还有一个区别：在 Canvas 中设置 line-height 属性时，浏览器将忽略其值，因为规范要求浏览器必须将该值设置为 normal。

提示：网页安全字体列表

在以下这些网址中都可以查到网页安全字体的列表：

http://www.speaking-in-styles.com/web-typography/Web-Safe-Fonts

http://www.codestyle.org/css/font-family/sampler-CombinedResultsFull.shtml

http://www.apaddedcell.com/web-fonts

3.3 文本的定位

在学过了如何对文本进行描边与填充，以及如何设置字型之后，我们再来看看怎么在 canvas 中对文本进行定位。

3.3.1 水平与垂直定位

当在 canvas 之中使用 strokeText() 或 fillText() 绘制文本时，需要指定所绘文本的 X 与 Y 坐标，然而，浏览器具体会将文本绘制在何处，则要看 textAlign 与 textBaseline 这两个绘图环境对象的属性。图 3-5 中的应用程序演示了使用这些属性的各种取值组合来绘制文本时的效果。

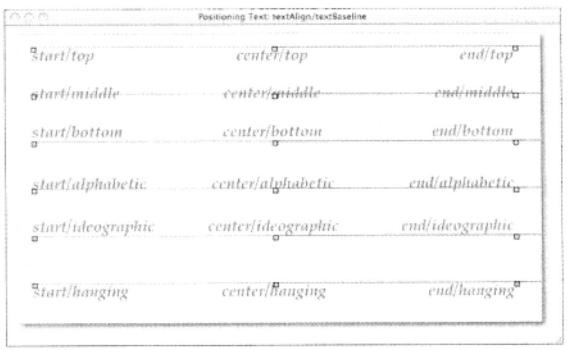


图 3-5 使用各种“对齐方式 / 基线”组合来绘制文本：这两个属性的默认值分别是 start 与 alphabetic

图 3-5 中的那个实心矩形表示应用程序传递给 fillText() 方法的 X 与 Y 坐标。图中所显示的每个字符串都表示了一种 textAlign 与 textBaseline 属性值的组合。

textAlign 属性可以取的值有：

- start
- center
- end
- left
- right

textAlign 属性的默认值是 start，当 canvas 元素的 dir 属性是 ltr 时，也就是说浏览器是按照由左至右的方向来显示文本时，left 的效果与 start 相同，而 right 的效果则与 end 相同。同理，如果 dir 属性的值是 rtl，也就是说浏览器是从右至左来显示文本的，那么 right 的效果则与 start 一

致，而 left 则与 end 一致。图 3-5 中所示的应用程序是按照从左至右的方向来显示文本的。

textBaseline 属性可以取的值有：

- top
- bottom
- middle
- alphabetic
- ideographic
- hanging

textBaseline 属性的默认值是 alphabetic，该值用于绘制由基于拉丁字母的语言所组成的字符串。ideographic 值则用于绘制日文或中文字符串，hanging 值用于绘制各种印度语字符串。top、bottom 与 middle 这三个值与特定的语言不相关，它们代表文本周围的边界框之内的某个位置，这个边界框也叫做“字符方框”（em square）。

图 3-5 所示应用程序的代码列在了程序清单 3-4 之中。表 3-2 总结了绘图环境对象中 textAlign 及 textBaseline 属性的用法[⊖]。

程序清单 3-4　利用 textAlign 与 textBaseline 属性来定位文本

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    fontHeight = 24,
    alignValues = ['start', 'center', 'end'],
    baselineValues = ['top', 'middle', 'bottom',
                      'alphabetic', 'ideographic','hanging'],
    x, y;

//Functions.........................................................

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for more information
}

function drawTextMarker() {
   context.fillStyle = 'yellow';
   context.fillRect  (x, y, 7, 7);
   context.strokeRect(x, y, 7, 7);
}

function drawText(text, textAlign, textBaseline) {
   if(textAlign) context.textAlign = textAlign;
   if(textBaseline) context.textBaseline =textBaseline;

   context.fillStyle = 'cornflowerblue';
   context.fillText(text, x, y);
}

function drawTextLine() {
   context.strokeStyle = 'gray';
   context.beginPath();
   context.moveTo(x, y);
```

⊖ textBaseline属性的各个取值所表示的意义，可以参阅Canvas规范中的图解，其网址是：http://images.whatwg.org/baselines.png。——译者注

```
    context.lineTo(x + 738, y);
    context.stroke();
}

//Initialization.....................................................

context.font = 'oblique normal bold 24px palatino';

drawGrid('lightgray', 10, 10);

for (var align=0; align < alignValues.length; ++align) {
    for (var baseline=0; baseline <baselineValues.length; ++baseline) {
        x = 20 + align*fontHeight*15;
        y = 20 + baseline*fontHeight*3;

        drawText(alignValues[align] + '/' +baselineValues[baseline],
                 alignValues[align],baselineValues[baseline]);

        drawTextMarker();
        drawTextLine();
    }
}
```

表 3-2 文本的对齐属性与基线属性

属　性	描　述
textAlign	该属性决定了文本在水平方向上的对齐方式。有效取值是：start、left、center、right 及 end。默认值是 start
textBaseline	该属性决定了文本在垂直方向上的对齐方式。有效取值是：top、bottom、middle、alphabetic、ideographic 及 hanging。默认值是 alphabetic

提示：字符方框

在数字时代以前使用印刷机印刷字符时，某个字型的磅值[⊖]被定义为刻有这种字符的金属活字版上每个字块的高度。这种金属活字块如下图所示：

活字字块的高度叫做磅值，而字母 M 的宽度则在习惯上被称为“字符方框”（em square）值。

然而，随着时间的推移，“字符方框”这个术语的含义也在逐渐演化，它也涵盖了那些不包含字母“M”的语言。现在，“字符方框”一般指的是某个字型的字符高度。

3.3.2 将文本居中

可以通过上一小节中所讨论的 textAlign 与 textBaseline 属性来将文本居中于某个点。图 3-6 所示的应用程序将文本置于 canvas 的中心。

程序清单 3-5 节选了该应用程序的部分 JavaScript 代码。

⊖ point size，也叫点、磅或级，是印刷上所使用的一种长度单位。72 点相当于 1 英寸。详情参见：http://zh.wikipedia.org/zh-cn/点_(印刷)。——译者注

程序清单 3-5　将文本居中于某个点

```
function drawText() {
   context.fillStyle = 'blue';
   context.strokeStyle = 'yellow';

   context.fillText(text, canvas.width/2,canvas.height/2);
   context.strokeText(text, canvas.width/2,canvas.height/2);
}

context.textAlign = 'center';
context.textBaseline = 'middle';
```

该应用程序分别将 textAlign 与 textBaseline 属性的值设置为 center 与 middle，然后对文本进行描边与填充。

应用程序的代码对文本进行填充与描边操作时，向 fillText() 与 strokeText() 方法所传入的绘制位置就是 canvas 的中心点。由于该应用程序已经分别将 textAlign 与 textBaseline 属性设置为 center 与 middle 了，所以绘制出来的文本就会位于 canvas 的中心，如图 3-6 所示。

图 3-6　将文本绘制在 canvas 的中心

3.3.3　文本的度量

只要你做的事情与文本有关，你就得设法获取某个字符串的像素宽度与高度。比方说，图 3-7 所演示的这个带有光标的简单文本编辑器，它就必须要知道应该把光标绘制在 canvas 中的哪个位置上，因而还要知道文本的尺寸。

The cursor is at the end of the line.

图 3-7　将光标绘制在计算好的像素位置上

要将光标放在这行文本的末尾，就必须先算出文本的宽度。

Canvas 绘图环境对象提供了一个名为 measureText() 的方法，用以度量某个字符串的像素宽度。该方法所返回的 TextMetrics 对象中包含了一个名为 width 的属性，这个属性就是字符串的宽度。在 3.4.2 小节中，我们将会使用 measureText() 方法来计算某行文本的宽度，计算方式如程序清单 3-6 所示。

程序清单 3-6　计算文本行的宽度

```
TextLine = function (x, y) {
   this.text = '';
```

```
    ...
};

TextLine.prototype = {
    getWidth: function(context) {
        return context.measureText(this.text).width;
    },
    ...
};
```

表 3-3 总结了 measureText() 方法的用法。

表 3-3 measureText() 方法的用法

方 法	描 述
TextMetrics measureText(DOMString text)	返回一个 TextMetrics 对象，该对象所包含的 width 属性代表传入文本的像素宽度。该宽度是基于当前的字型而计算出来的，在写作本书时，它是唯一一个可以从 TextMetrics 对象中获取的度量值

警告：在调用 measureText() 方法之前先设置好字型

在使用 measureText() 方法时，常见的错误就是在调用完该方法之后，才去设置字型。请注意，measureText() 方法是根据当前的字型来计算字符串宽度的，因此，如果你在调用 measureText() 方法之后才去改变字型，那么该方法所返回的宽度并不能反映出以那种字型来度量的实际文本宽度。

警告：文本宽度的度量值是不够精确的

只能够通过 measureText() 方法所返回的 TextMetrics 对象之中的 width 属性来获取任意字符串的像素宽度。但是，TextMetrics 对象却并未（至少在本书出版时）提供对应的 height 属性用以获取高度。而且，Canvas 规范中的一句话又使得文本尺寸的度量变得更为复杂：

使用 fillText() 与 strokeText() 方法来渲染字形（Glyph）时，绘制范围有可能会超出由字型大小所定义的范围框（也就是“字符方框”，em square），绘制的宽度也有可能会超出由 measureText() 方法所返回的宽度（也就是文本宽度）。

上文所引述的这段规范表明：由 measureText() 方法所返回的文本宽度并不精确。在通常情况下，就算该值不精确也无关紧要，不过，有的时候我们必须要获取精确的文本宽度。读者将会在 3.4.2 小节中看到这种情况。

3.3.4 绘制坐标轴旁边的文本标签

在 2.8.3 小节中，读者已经学会了如何绘制坐标轴，那么本小节我们就来给坐标轴加上文本标签，如图 3-8 所示。

程序清单 3-7 节选了图 3-8 所示应用程序的一部分 JavaScript 代码。

绘制垂直与水平坐标轴的文本标签是非常简单的。应用程序根据坐标轴的位置、坐标轴上每条刻度线的长度，以及坐标轴与文本标签之间的距离，来决定每个文本标签的绘制坐标。

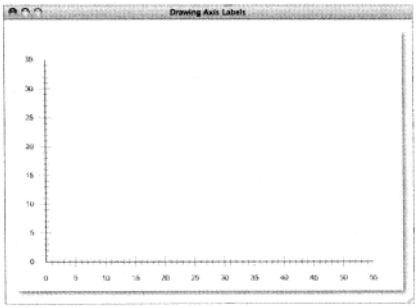


图 3-8　绘制带有文本标签的坐标轴

程序清单 3-7　为坐标轴加上文本标签

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    HORIZONTAL_AXIS_MARGIN = 50,
    VERTICAL_AXIS_MARGIN = 50,

    AXIS_ORIGIN = { x: HORIZONTAL_AXIS_MARGIN,
                    y:canvas.height-VERTICAL_AXIS_MARGIN },

    AXIS_TOP   = VERTICAL_AXIS_MARGIN,
    AXIS_RIGHT = canvas.width-HORIZONTAL_AXIS_MARGIN,

    HORIZONTAL_TICK_SPACING = 10,
    VERTICAL_TICK_SPACING = 10,

    AXIS_WIDTH  = AXIS_RIGHT - AXIS_ORIGIN.x,
    AXIS_HEIGHT = AXIS_ORIGIN.y - AXIS_TOP,

    NUM_VERTICAL_TICKS   = AXIS_HEIGHT /VERTICAL_TICK_SPACING,
    NUM_HORIZONTAL_TICKS = AXIS_WIDTH /HORIZONTAL_TICK_SPACING,

    TICK_WIDTH = 10,

    SPACE_BETWEEN_LABELS_AND_AXIS = 20;

//Functions.........................................................

function drawAxes() {
   context.save();
   context.lineWidth = 1.0;
   context.fillStyle = 'rgba(100,140,230,0.8)';
   context.strokeStyle = 'navy';

   drawHorizontalAxis();
   drawVerticalAxis();

   context.lineWidth = 0.5;
   context.strokeStyle = 'navy';

   context.strokeStyle = 'darkred';
   drawVerticalAxisTicks();
   drawHorizontalAxisTicks();
```

```
   context.restore();
}

// Axis drawing methods omitted for brevity. See Example 2.14
// for a complete listing
...

function drawAxisLabels() {
   context.fillStyle = 'blue';
   drawHorizontalAxisLabels();
   drawVerticalAxisLabels();
}

function drawHorizontalAxisLabels() {
   context.textAlign = 'center';
   context.textBaseline = 'top';

   for (var i=0; i <= NUM_HORIZONTAL_TICKS; ++i) {
      if (i % 5 === 0) {
         context.fillText(i,
            AXIS_ORIGIN.x + i *HORIZONTAL_TICK_SPACING,
            AXIS_ORIGIN.y +SPACE_BETWEEN_LABELS_AND_AXIS);
      }
   }
}

function drawVerticalAxisLabels() {
   context.textAlign = 'right';
   context.textBaseline = 'middle';

   for (var i=0; i <= NUM_VERTICAL_TICKS; ++i) {
      if (i % 5 === 0) {
         context.fillText(i,
                  AXIS_ORIGIN.x -SPACE_BETWEEN_LABELS_AND_AXIS,
                  AXIS_ORIGIN.y - i *VERTICAL_TICK_SPACING);
      }
   }
}

function drawGrid(color, stepx, stepy) {
   // Listing omitted for brevity. See Example 2.13
   // for more information
}

//Initialization.....................................................
context.font = '13px Arial';

drawGrid('lightgray', 10, 10);

context.shadowColor = 'rgba(100,140,230,0.8)';
context.shadowOffsetX = 3;
context.shadowOffsetY = 3;
context.shadowBlur = 5;

drawAxes();
drawAxisLabels();
```

请注意，应用程序会设置 textAlign 与 textBaseline 的值。在绘制水平坐标轴的文本标签时，应用程序将这两个属性分别设置为 center 和 top，在绘制垂直坐标轴时，则设置为 right 与 middle，如图 3-9 所示。

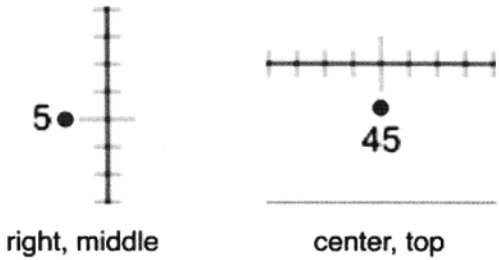


图 3-9　绘制坐标轴旁边的文本标签时所使用的对齐与基线属性

3.3.5　绘制数值仪表盘周围的文本标签

正如 3.3.4 小节讲的那样，为水平与垂直坐标轴画上文本标签是件很容易的事情。然而，如果要在圆弧或圆形周围添加文本标签，则稍具挑战性，因为我们得引入一些三角函数的计算。

图 3-10 所示的应用程序，绘制了一个表示圆周角度的仪表盘。

图 3-10　给数值仪表盘的周围加上文本标签

在绘制每个文本标签时，应用程序都要计算出它的位置，如图 3-11 中的圆点所示。应用程序的代码先把 textAlign 属性设置为 center，textBaseline 属性设置为 middle，然后再将每个标签绘制到其相应的位置之上。

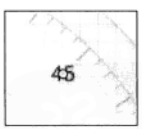


图 3-11　将仪表盘周围的文本标签绘制在合适的位置上，textAlign='center'，textBaseline='middle'

程序清单 3-8 列出了图 3-10 所示应用程序的 JavaScript 选段。

程序清单 3-8　绘制数值仪表盘周围的文本标签

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    DEGREE_ANNOTATIONS_FILL_STYLE ='rgba(0,0,230,0.9)',
    DEGREE_ANNOTATIONS_TEXT_SIZE = 12;

//Functions.....................................................
```

```
function drawDegreeAnnotations() {
   var radius = circle.radius + DEGREE_DIAL_MARGIN;

   context.save();
   context.fillStyle = DEGREE_ANNOTATIONS_FILL_STYLE;
   context.font = DEGREE_ANNOTATIONS_TEXT_SIZE + 'px Helvetica';

   for (var angle=0; angle < 2*Math.PI; angle +=Math.PI/8) {
      context.beginPath();

      context.fillText((angle * 180 /Math.PI).toFixed(0),
         circle.x + Math.cos(angle) * (radius -TICK_WIDTH*2),
         circle.y - Math.sin(angle) * (radius -TICK_WIDTH*2));
   }
   context.restore();
}

//Initialization.............................................
...

context.textAlign =  'center';
context.textBaseline = 'middle';

drawGrid('lightgray', 10, 10);
drawDial();
```

请注意程序清单 3-8 之中对 fillText() 方法的调用。应用程序传递给该方法的最后两个参数分别表示文本的 X 与 Y 坐标。还有一种办法，就是先将坐标系平移到想要绘制文本的位置，然后再以 (0, 0) 点为参数进行填充，其代码如下所示：

```
function drawDegreeAnnotations() {
   ...
   for (var angle=0; angle < 2*Math.PI; angle +=Math.PI/8) {
      ...
      context.translate(
         circle.x + Math.cos(angle) * (radius -TICK_WIDTH*2),
         circle.y - Math.sin(angle) * (radius -TICK_WIDTH*2));
      context.fillText((angle * 180 /Math.PI).toFixed(0), 0, 0);
   }
}
```

在下一小节中，我们将对绘图环境对象的坐标系进行平移，以便在圆弧周围绘制文本。

3.3.6 在圆弧周围绘制文本

图 3-12 所示应用程序可以将文本绘制在某段圆弧周围，该应用程序的代码列在程序清单 3-9 之中，这段代码通过如下步骤来绘制文本：

（1）计算圆弧周围每个字符的绘制坐标。

（2）将坐标系平移至绘制字符的地点。

（3）将坐标系旋转 π/2 [⊖] – angle 度。

（4）对字符进行描边或填充操作（有时也会同时进行两种操作）

drawCircularText() 函数实现了上述四个步骤。请注意，应用程序必须先调用 translate() 方法，然后才能再调用 rotate() 方法。如果你先将绘制环境对象的坐标系按照默认的原点坐标进行旋转，那么将会导致截然不同（而且莫名其妙）的绘制效果。

⊖ 原书此处为π，但从源代码来看，应是π/2，故这里采用π/2。——译者注

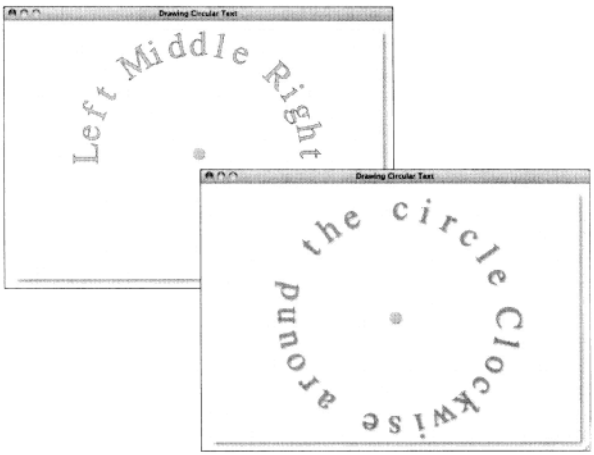


图 3-12　在圆弧周围绘制经过旋转的文本

程序清单 3-9　沿着圆弧绘制文本

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...
    TEXT_FILL_STYLE = 'rgba(100,130,240,0.5)',
    TEXT_STROKE_STYLE = 'rgba(200,0,0,0.7)',
    TEXT_SIZE = 64,

    circle = { x: canvas.width/2,
               y: canvas.height/2,
               radius: 200
             };

//Functions.............................................................

function drawCircularText(string, startAngle,endAngle) {
   var radius = circle.radius,
       angleDecrement = (startAngle -endAngle)/(string.length-1),
       angle = parseFloat(startAngle),
       index = 0,
       character;

   context.save();

   context.fillStyle = TEXT_FILL_STYLE;
   context.strokeStyle = TEXT_STROKE_STYLE;
   context.font = TEXT_SIZE + 'px Lucida Sans';

   while (index < string.length) {
      character = string.charAt(index);

      context.save();
      context.beginPath();

      context.translate(circle.x + Math.cos(angle) *radius,
                        circle.y - Math.sin(angle) *radius);
```

```
        context.rotate(Math.PI/2 - angle);

        context.fillText(character, 0, 0);
        context.strokeText(character, 0, 0);

        angle -= angleDecrement;
        index++;

        context.restore();
    }
    context.restore();
}

//Initialization.....................................................

context.textAlign = 'center';
context.textBaseline = 'middle';
...

drawCircularText("Clockwise around the circle",Math.PI*2, Math.PI/8);
```

既然读者已经很好地掌握了如何在 canvas 之中绘制文本，那么接下来我们就用学到的知识来实现一个简单的文本编辑器。

3.4 实现文本编辑控件

虽说 Canvas 并未提供一些复杂的文本编辑功能，诸如光标、文本选择、剪切、复制、粘贴等等，但是，它却提供了丰富的图形处理能力，让我们可以借此来实现这些功能。本章将会利用 Canvas 的 API 来研究如何实现一个简单的文本编辑器。

我们将从一个简单的文本输入光标开始，最终做好一个可以书写多个带光标的段落，并且可以支持多行文本的编辑器。

3.4.1 指示文本输入位置的光标

我们先来绘制一个简单的文本输入光标。这种光标只需绘制上去即可，不需要擦除，效果如图 3-13 所示。

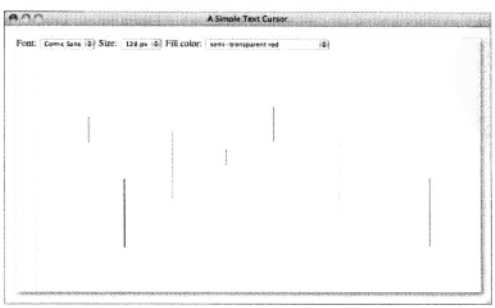


图 3-13 指示文本输入位置的光标

图 3-13 所示应用程序在用户每次点击鼠标时都会绘制一个光标。在程序清单 3-10 所列出的 TextCursor 对象最初版本的实现代码中，并没有 erase() 方法，所以，虽说图 3-13 之中的应用程

序看起来有很多光标，然而实际上该应用程序并不支持多个光标。只不过是因为每次用户点击鼠标时，应用程序都会重新配置绘制参数并重绘光标，同时又不将原有的光标擦除，所以才有了这样的绘制效果。

应用程序代码将页面顶部的 HTML 控件与 Canvas 绘图环境对象中的 font 及 fillColor 属性相连接，在绘制光标时，也会用到这些属性。因此，光标的绘制属性将取决于用户在页面顶部的 HTML 控件中所选的值。

程序清单 3-10　指示文本输入位置的光标

```
TextCursor = function (width, fillStyle) {
   this.fillStyle = fillStyle || 'rgba(0,0,0,0.5)';
   this.width     = width || 2;
   this.left      = 0;
   this.top       = 0;
};

TextCursor.prototype = {
   getHeight: function (context) {
      var h = context.measureText('W').width;
      return h + h/6;
   },

   createPath: function (context) {
      context.beginPath();
      context.rect(this.left, this.top,
                   this.width,this.getHeight(context));
   },

   draw: function (context, left, bottom) {
      context.save();

      this.left = left;
      this.top = bottom - this.getHeight(context);

      this.createPath(context);

      context.fillStyle = this.fillStyle;
      context.fill();

      context.restore();
   },
};
```

TextCursor 对象是一个简单的填充矩形，它的高度是根据绘图环境对象当前的字型属性而算出来的。然而回想一下前面讲过的内容：绘图环境的 measureText() 方法所返回的 TextMetric 对象中，唯一的度量值 width，就是你传给该方法的字符串所占据的宽度。所以，我们先要计算字母“M”的宽度，然后再将其乘以 1 又 1/6，以此值来作为文本光标的高度。

TextCursor 对象是通过一个单独的文件来实现的，它的名字叫做 text.js。本范例程序的 HTML 代码列在了程序清单 3-11 之中，这段代码包含了那个 JavaScript 程序文件。

程序清单 3-11　简单的文本输入光标（HTML 代码）

```
<!DOCTYPE html>
<html>
   <head>
```

```
        <title>A Simple Text Cursor</title>
        ...
    </head>

    <body>
        <canvas id='canvas' width='780' height='440'>
            Canvas not supported
        </canvas>
        ...

        <script src='text.js'></script>
        <script src='example.js'></script>
    </body>
</html>
```

程序清单 3-12 节选了图 3-13 所示应用程序的一部分 JavaScript 代码。该应用程序创建了一个 TextCursor 对象，它会将光标绘制在用户点击鼠标时的位置上。

程序清单 3-12　简单的文本输入光标（JavaScript 代码选段）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    cursor = new TextCursor();

function moveCursor(loc) {
   cursor.draw(context, loc.x, loc.y);
}

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(e);
   moveCursor(loc);
};
...
```

我们得承认，上面所讨论的这段代码绘制出来的文本光标并没有什么精彩之处，所以，为了让它变得更有趣一些，咱们再来编写一个 erase() 方法吧。

提示：计算文本高度所用的经验法则

measureText() 所返回的 TextMetrics 对象之中，唯一包含的度量值就是你传递给该方法的字符串宽度。这意味着你必须自己来计算字符串的高度。幸好对于大多数字型来说，将字母“M”的宽度再稍微增加一点儿，就可以得出近似的文本高度了。

3.4.1.1　光标的擦除

刚才那一小节在讲述文本光标时，回避了一个实现光标所遇到的最为复杂的问题，那就是如何擦除光标。在绘制光标时，你只是希望将它临时绘制在屏幕上，所以，必须有办法将其擦掉才行。

Canvas 提供了很多种用于绘制临时内容的方式，在 2.8.4 小节中，读者已经看到了如何绘制橡皮筋式辅助线，来帮助用户互动式地创建线段。在那个例子中，我们在绘制橡皮筋式辅助线之前，先把整个绘图表面保存起来，然后再通过恢复绘图表面来擦除上次的线段。

光标的擦除也可以采用类似的方式。在绘制光标之前的某个时刻，我们先调用 getImageData() 方法来抓取当前 canvas 的快照。在将光标绘制到 canvas 之后，我们把光标所在位置的那一块矩形快照内容复制回 canvas，这样的话，就可以实现擦除光标的效果了。

TextCursor.erase() 方法接受一个名为 imageData 的参数，该方法会将光标所在位置的那块矩形图像从快照中复制到 canvas 之上。程序清单 3-13 列出了 TextCursor 对象中 erase() 方法的实现代码。

程序清单 3-13　带有 erase() 函数的 TextCursor 对象

```
TextCursor.prototype = {
   ...

   erase: function (context, imageData) {
      context.putImageData(imageData, 0, 0,
                           this.left,  this.top,
                           this.width,  this.getHeight(context));
   }
};
```

TextCursor 对象的 erase() 方法假定传给它的 imageData 参数表示整个 canvas。为了擦除光标，该方法调用了绘图环境对象的 putImageData() 方法。在 2.8.4 小节中，我们也调用过这个方法，那时传入的 3 个参数分别代表快照图像本身，以及要将其恢复到 canvas 时所用的 X、Y 坐标。此处，我们又增加了用以表示快照复制范围的 4 个参数，只有它们所定义的那个矩形区域之中的内容，才会被复制到 canvas。

此处的重点是如何运用 putImageData() 方法来擦除光标，而不是深入研究该方法的细节。在第 4 章中，我们会详细讲述 getImageData() 方法与 putImageData() 方法。现在，大家只需要明白程序清单 3-13 之中的 erase() 方法会将快照中的某个特定矩形区域范围内的图像复制回 canvas 即可。快照图像数据是应用程序通过调用 getImageData() 方法而得到的。这段代码列在了程序清单 3-14 之中。

程序清单 3-14　光标的擦除

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    drawingSurfaceImageData,
    cursor = new TextCursor();

// Drawing surface.............................................

function saveDrawingSurface() {
   drawingSurfaceImageData = context.getImageData(0,0,
                             canvas.width,
                             canvas.height);
}

//Text........................................................
...

function moveCursor(loc) {
   cursor.erase(context, drawingSurfaceImageData);
   cursor.draw(context, loc.x, loc.y);
}

// Event handlers..............................................
...

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(e);
   moveCursor(loc);
```

```
};

//Initialization.............................................
...

drawGrid(GRID_STROKE_STYLE,
         GRID_HORIZONTAL_SPACING,
         GRID_VERTICAL_SPACING);

saveDrawingSurface();
```

程序清单 3-14 之中的这段应用程序代码，一开始先在背景之中绘制了网格线，然后调用 getImageData() 方法将绘图表面保存起来。当用户点击鼠标时，应用程序会将上一个位置的光标擦除，并将光标重绘在用户这一次点击鼠标的地方。

3.4.1.2 光标的闪烁效果

学会了如何擦除光标之后，我们很容易就能编写出像程序清单 3-15 这样实现光标闪烁效果的代码来。

程序清单 3-15 之中的应用程序代码创建了光标，并通过 blinkCursor() 函数使其闪烁。应用程序每秒钟都会将光标擦除 1 次，并且在擦除之后等待 300ms，再将其绘制出来。这意味着在每 1 秒钟的时间段内，光标有 700ms 是可见的。

程序清单 3-15 光标的闪烁效果

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    blinkingInterval,
    BLINK_ON = 500,
    BLINK_OFF = 500,

    cursor = new TextCursor();

//Functions.........................................................

function blinkCursor(loc) {
   blinkingInterval = setInterval( function (e) {
      cursor.erase(context, drawingSurfaceImageData);

      setTimeout( function (e) {
         cursor.draw(context, cursor.left,
                     cursor.top +cursor.getHeight(context));
      }, BLINK_OFF);
   }, BLINK_ON + BLINK_OFF);
}

function moveCursor(loc) {
   cursor.erase(context, drawingSurfaceImageData);
   cursor.draw(context, loc.x, loc.y);

   if (!blinkingInterval)
      blinkCursor(loc);
}

// Event handlers...................................................
canvas.onmousedown = function (e) {
```

```
    var loc = windowToCanvas(e);
    moveCursor(loc);
};
...
```

用户第一次点击鼠标时，应用程序调用 blinkCursor() 方法，该方法会使光标持续不停地闪烁。如果不想让它继续闪烁，比如，要将其彻底隐藏，那么调用 clearInverval() 方法就可以了。

我们已经做好了一个能够跟随鼠标出现的闪烁光标，现在利用它来向 canvas 之中插入一些文本。

3.4.2　在 Canvas 中编辑文本

图 3-14 所示应用程序可以让用户向 canvas 中输入文本行，它会将用户输入的字符追加至当前行的末尾，并让光标在用户打字过程中一直向后移动，始终位于文本的尾端。

可以按 Backspace 键来删除当前这一行的最后一个字符。如果用户在 canvas 中再次点击鼠标，那么应用程序就会结束当前这一行的输入，转而将光标移动到点击鼠标的地方，并开始新的一行。

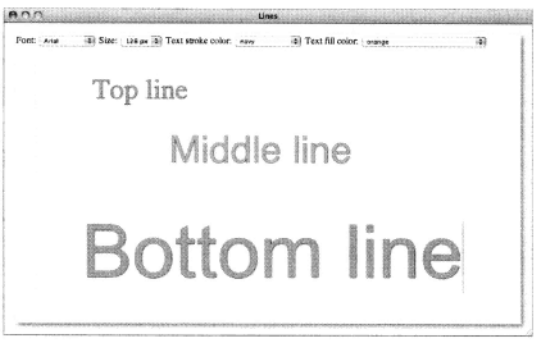


图 3-14　在 canvas 中实现单行文本的输入

在图 3-14 所示应用程序的代码中，实现了一个名叫 TextLine 的对象，其代码如程序清单 3-16 所示。

程序清单 3-16　TextLine 对象

```
// Constructor.........................................................

TextLine = function (x, y) {
    this.text = '';
    this.left = x;
    this.bottom = y;
    this.caret = 0;
};
// Prototype...........................................................

TextLine.prototype = {
    insert: function (text) {
        this.text = this.text.substr(0, this.caret) + text +
                    this.text.substr(this.caret);
        this.caret += text.length;
    },

    removeCharacterBeforeCaret: function () {
```

```
        if (this.caret === 0)
           return;

        this.text = this.text.substring(0, this.caret-1) +
                    this.text.substring(this.caret);

        this.caret--;
     },

     getWidth: function(context) {
        return context.measureText(this.text).width;
     },

     getHeight: function (context) {
        var h = context.measureText('W').width;
        return h + h/6;
     },

     draw: function(context) {
        context.save();
        context.textAlign = 'start';
        context.textBaseline = 'bottom';

        context.strokeText(this.text,  this.left, this.bottom);
        context.fillText(this.text,  this.left, this.bottom);

        context.restore();
     },

     erase: function (context, imageData) {
        context.putImageData(imageData, 0, 0);
     }
  };
```

每个 TextLine 对象都含有如下信息：一个字符串，该字符串在 canvas 中的位置，以及用户当前向字符串中插入文本的位置。插入文本的位置又叫做 caret。TextLine 对象中的 insert()、draw()、erase() 分别用于向当前 caret 所指的位置插入文本、绘制字符串及擦除文本，而 getWidth() 与 getHeight() 方法则可以分别获取文本行的宽度与高度。

图 3-14 所示应用程序其对 TextLine 对象的创建与操作，完全位于事件处理器之中。这段代码如程序清单 3-17 所示。

程序清单 3-17　文本行的绘制

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    fontSelect = document.getElementById('fontSelect'),
    sizeSelect = document.getElementById('sizeSelect'),
    strokeStyleSelect = document.getElementById('strokeStyleSelect'),
    fillStyleSelect = document.getElementById('fillStyleSelect'),

    GRID_STROKE_STYLE = 'lightgray',
    GRID_HORIZONTAL_SPACING = 10,
    GRID_VERTICAL_SPACING = 10,

    cursor = new TextCursor(),

    line,

    blinkingInterval,
```

```
    BLINK_TIME = 1000,
    BLINK_OFF = 300;

// General-purpose functions.................................

function drawBackground() { // Ruled paper
   // Listing omitted for brevity. See Example 3.2
   // for a complete listing.
}

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  / bbox.width),
            y: y - bbox.top  * (canvas.height / bbox.height)
          };
}

// Drawing surface...........................................

function saveDrawingSurface() {
   drawingSurfaceImageData = context.getImageData(0, 0,
                                                  canvas.width,
                                                  canvas.height);
}

// Text......................................................

function setFont() {
   context.font = sizeSelect.value + 'px ' + fontSelect.value;
}

function blinkCursor(x, y) {
   clearInterval(blinkingInterval);
   blinkingInterval = setInterval( function (e) {
      cursor.erase(context, drawingSurfaceImageData);

      setTimeout( function (e) {
         if (cursor.left == x &&
            cursor.top + cursor.getHeight(context) == y) {
            cursor.draw(context, x, y);
         }
      }, 300);
   }, 1000);
}

function moveCursor(x, y) {
   cursor.erase(context, drawingSurfaceImageData);
   saveDrawingSurface();
   context.putImageData(drawingSurfaceImageData, 0, 0);

   cursor.draw(context, x, y);
   blinkCursor(x, y);
}

// Event handlers............................................

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(e.clientX, e.clientY),
       fontHeight = context.measureText('W').width;

   fontHeight += fontHeight/6;
```

```
    line = new TextLine(loc.x, loc.y);
    moveCursor(loc.x, loc.y);
};

fillStyleSelect.onchange = function (e) {
    cursor.fillStyle = fillStyleSelect.value;
    context.fillStyle = fillStyleSelect.value;
}

strokeStyleSelect.onchange = function (e) {
    cursor.strokeStyle = strokeStyleSelect.value;
    context.strokeStyle = strokeStyleSelect.value;
}

// Key event handlers.........................................

document.onkeydown = function (e) {
    if (e.keyCode === 8 || e.keyCode === 13) {
        // The call to e.preventDefault() suppresses the browser's
        // subsequent call to document.onkeypress(), so
        // only suppress that call for Backspace and Enter.
        e.preventDefault();
    }

    if (e.keyCode === 8) { // Backspace
        context.save();

        line.erase(context, drawingSurfaceImageData);
        line.removeCharacterBeforeCaret();

        moveCursor(line.left + line.getWidth(context), line.bottom);

        line.draw(context);
        context.restore();
    }
}

document.onkeypress = function (e) {
    var key = String.fromCharCode(e.which);

    if (e.keyCode !== 8 && !e.ctrlKey && !e.metaKey) {
      e.preventDefault(); // No further browser processing

      context.save();

      line.erase(context, drawingSurfaceImageData);
      line.insert(key);

      moveCursor(line.left + line.getWidth(context), line.bottom);

      context.shadowColor = 'rgba(0,0,0,0.5)';
      context.shadowOffsetX = 1;
      context.shadowOffsetY = 1;
      context.shadowBlur = 2;

      line.draw(context);
      context.restore();
    }
}

// Initialization............................................
```

```
fontSelect.onchange = setFont;
sizeSelect.onchange = setFont;

cursor.fillStyle = fillStyleSelect.value;
cursor.strokeStyle = strokeStyleSelect.value;

context.fillStyle = fillStyleSelect.value;
context.strokeStyle = strokeStyleSelect.value;
context.lineWidth = 2.0;

setFont();
drawBackground();
saveDrawingSurface();
```

当用户点击鼠标之后，应用程序就会创建一个新的 TextLine 对象，并将该文本行对象与光标的位置都移动到点击鼠标的地方。

如果应用程序检测到了按键被按下的事件，那么它就会先检查按下的键是不是 Backspace。如果是，就擦除该行文本，将插入符号[⊖]前面的那个字符删除，重新调整光标位置，并重绘文本。 如果用户按下的键不是 Backspace，而且也没有按住 Ctrl 或 Meta 键不松开，那么，在稍后浏览器调用应用程序的 onkeypress() 方法时，用户所输入的字符就会被插入到这行文本之中了。

读者在学会了如何实现简单的单行文本控制之后，就可以将它扩展为多行文本输入，并以此来实现文本段的编辑。

警告：要想擦除文本，必须替换掉整个 Canvas

在本书 3.3.3 小节中，我们引用过 Canvas 规范中的这句话：

使用 fillText() 与 strokeText() 方法来渲染字形时，绘制范围有可能会超出由字型大小所定义的范围框（也就是“字符方框”，em square），绘制的宽度也有可能会超出由 measureText() 方法所返回的宽度（也就是文本宽度）……

该规范继续说：

……如果要渲染或移除文本，则需要注意：必须将剪辑区域所覆盖的整个 canvas 范围内的图像都替换掉，而不能仅仅替换根据文本宽度与字符方框高度所确定的那个范围框中的图像。

这最后一句话的意思是，本小节所实现的 TextLine 对象在 erase() 方法之中，应该将整个 canvas（被剪辑区域所覆盖的那一部分）的内容都擦除。这与 3.4.1.1 小节中的方式不同，在那一小节之中，为了擦除光标，我们只恢复了光标所在范围框内的一小块图像而已。

3.4.3 文本段的编辑

在上一小节之中，读者看到了如何实现闪烁的光标，如何编辑文本行。本小节我们则要实现一个 Paragraph 对象。每个文本段对象都含有一个由 TextLine 对象所构成的数组，同时还保存了一个指向当前用户所编辑文本行的引用。此外，文本段对象还存放了一个与用户编辑位置相同步的光标。用户可以通过图 3-15 所示的应用程序来编辑文本段。

文本段在逻辑上连接了其所包含的文本行。举例来说，当光标位于某行的最左端时，如果用户按下 Backspace 键，那么文本段对象则会将光标后边的文本，连同光标本身，上移一行，

⊖ caret，中文译为“脱字符号”或“插入符号”，在本章这个文本编辑器范例程序的语境中，它与“光标”（cursor）这个词的意义大致相同，在不致混淆的情况下，一律按习惯译为“光标”。——译者注

如图 3-16 所示。与此相似，如果用户在编辑段落中的某行文本时按下了 Enter 键，那么文本段对象则会创建一个新的文本行，并将其插入光标所在位置的下一行。

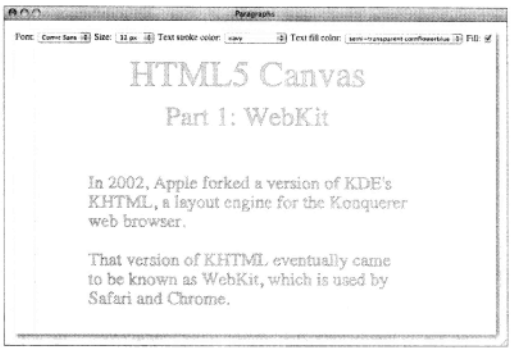


图 3-15　文本段的编辑

图 3-15 与图 3-16 之中的应用程序，使用了 Paragraph 对象所提供的一些重要方法：

- isPointInside()：如果给定的点位于段落内，则返回 true。
- moveCursorCloseTo()：给定一组 X 与 Y 坐标，将光标移动到距其最近的位置。
- addLine()：向文本段对象中增加一个 TextLine 对象。
- backspace()：在当前光标处执行退格操作。
- newline()：在当前光标处执行新行操作。
- insert()：在当前光标处插入文本。

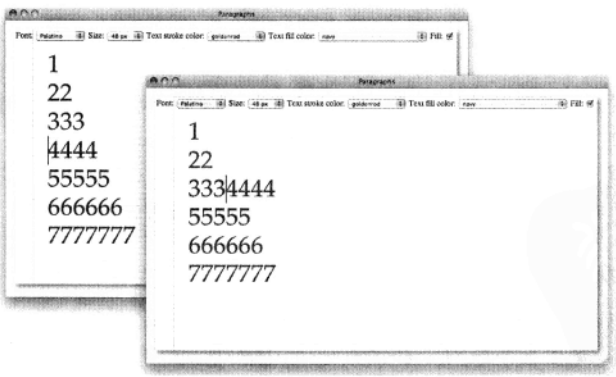


图 3-16　上方的图：按下 Backspace 键之前的文本段；下方的图：按下 Backspace 键之后的文本段

程序清单 3-18 节选了一部分图 3-15 之中所示应用程序的 JavaScript 代码。请注意看这段应用程序代码是如何使用上述 Paragraph 对象之中的方法的。

程序清单 3-18　Paragraph 对象的使用

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...
    cursor = new TextCursor(),
    paragraph;
...

function drawBackground() {
  // Listing omitted for brevity, see Example 3.1
}

// Drawing surface.................................................

function saveDrawingSurface() {
   drawingSurfaceImageData = context.getImageData(0, 0,
                                canvas.width, canvas.height);
}
...

// Event handlers..................................................

canvas.onmousedown = function (e) {
   var loc = windowToCanvas(canvas, e.clientX, e.clientY),
       fontHeight,
       line;

   cursor.erase(context, drawingSurfaceImageData);
   saveDrawingSurface();

   if (paragraph && paragraph.isPointInside(loc)) {
      paragraph.moveCursorCloseTo(loc.x, loc.y);
   }
   else {
      fontHeight = context.measureText('W').width,
      fontHeight += fontHeight/6;
      paragraph = new Paragraph(context, loc.x, loc.y - fontHeight,
                                drawingSurfaceImageData, cursor);
      paragraph.addLine(new TextLine(loc.x, loc.y));
   }
};
...

// Key event handlers..............................................

document.onkeydown = function (e) {
   if (e.keyCode === 8 || e.keyCode === 13) {
      // The call to e.preventDefault() suppresses the browser's
      // subsequent call to document.onkeypress(), so
      // only suppress that call for Backspace and Enter.
      e.preventDefault();
   }
   if (e.keyCode === 8) {  // Backspace
      paragraph.backspace();
   }
   else if (e.keyCode === 13) { // Enter
      paragraph.newline();
   }
}

document.onkeypress = function (e) {
```

```
    var key = String.fromCharCode(e.which);

    // Only process if user is editing text and they aren't
    // holding down the Ctrl or Meta keys.

    if (e.keyCode !== 8 && !e.ctrlKey && !e.metaKey) {
      e.preventDefault(); // No further browser processing

      context.fillStyle = fillStyleSelect.value;
      context.strokeStyle = strokeStyleSelect.value;

      paragraph.insert(key);
    }
}

// Initialization.....................................................
...

cursor.fillStyle = fillStyleSelect.value;
cursor.strokeStyle = strokeStyleSelect.value;

context.lineWidth = 2.0;
setFont();

drawBackground();
saveDrawingSurface();
```

如果用户在编辑某个文本段时，在该段落内部某处点击鼠标，那么 onmousedown() 事件处理器的代码就会调用文本段对象的 moveCursorCloseTo() 方法，将光标移动到距离按下鼠标处最近的地方。

如果点击鼠标时用户没有在编辑某个文本段，或者点击鼠标的地方位于当前活动的文本段之外，那么应用程序则会将绘图表面保存起来，然后创建一个新的文本段对象，并向该对象之中加入一个文本行对象。Paragraph 对象的构造器函数接受 4 个参数，它们分别是：指向绘图环境对象的引用、文本段的位置、绘图表面的图像数据，以及光标。

应用程序在 onkeydown() 事件处理器中处理 Backspace 与 Enter 键，它们分别会调用文本段对象的 backspace() 与 newline() 方法。

应用程序的 onkeypress() 事件处理器通过调用 paragraph 对象的 insert() 方法来向文本段中插入字符。

在程序清单 3-19 之中，我们将会看到 Paragraph 对象的代码。不过，在看那段代码之前，我们先来看看 Paragraph 对象是如何执行一些常用任务的。

3.4.3.1 创建与初始化文本段对象

图 3-15 所示应用程序采用如下代码来创建文本段对象：

```
var cursor = new TextCursor(),
    paragraph = new Paragraph(context, loc.x, loc.y - fontHeight,
                              drawingSurfaceImageData, cursor);
```

然后，应用程序向文本段对象之中加入了一个 TextLine 对象：

```
paragraph.addLine(new TextLine(loc.x, loc.y));
```

Paragraph 对象的构造器函数是这个样子的：

```
Paragraph = function (context, left, top, imageData, cursor) {
   this.context = context;
```

```
    this.drawingSurface = imageData;
    this.left = left;
    this.top = top;
    this.lines =[]
    this.activeLine = undefined;
    this.cursor = cursor;
    this.blinkingInterval = undefined;
};
```

Paragraph 对象之中包含了指向 Canvas 绘图环境对象的引用、在建立该对象时 canvas 绘图表面之中的图像数据、一个由 TextLine 对象所构成的数组，以及光标对象。Paragraph 对象也会记录自身的位置，以及用户当前正在编辑的 TextLine 对象。

Paragraph 对象的 addLine() 方法会向 TextLine 对象数组中加入一个 TextLine 对象，然后设置当前文本段的活动文本行，并将光标移动到新文本的开头：

```
Paragraph.prototype = {
    addLine: function (line) {
        this.lines.push(line);
        this.activeLine = line;
        this.moveCursor(line.left, line.bottom);
    },
    ...
}
```

请注意这段代码之中的 moveCursor() 方法。接下来，我们就看看这个方法是如何实现的。

3.4.3.2 根据鼠标点击位置来重新定位文本光标

文本段对象提供了一个名为 moveCursor() 的方法，可以将光标移动到 canvas 之中的特定位置上：

```
moveCursor: function (x, y) {
    this.cursor.erase(this.context, this.drawingSurface);
    this.cursor.draw(this.context, x, y);
    this.blinkCursor(x, y);
},
```

moveCursor() 方法将光标从当前位置擦掉，然后在新的位置重新将其绘制出来。然后，moveCursor() 方法会调用 blinkCursor() 方法。

Paragraph 对象通过 Paragraph.moveCursorCloseTo() 方法来将光标移动至它所包含的字符之间，该方法会将光标放置在距离给定的 canvas 位置最近的两个字符中间。moveCursorCloseTo() 方法的实现代码如下：

```
moveCursorCloseTo: function (x, y) {
    var line = this.getLine(y);

    if (line) {
        line.caret = this.getColumn(line, x);
        this.activeLine = line;
        this.moveCursor(line.getCaretX(context), line.bottom);
    }
},
getLine: function (y) {
    var line;

    for (i=0; i < this.lines.length; ++i) {
        line = this.lines [i];
        if (y > line.bottom - line.getHeight(context) &&
            y < line.bottom) {
            return line;
        }
```

```
    }
    return undefined;
},
```

3.4.3.3 向段落中插入文本

Paragraph 对象所提供的 insert() 方法用来向文本段中插入文本：

```
insert: function (text) {
    var t = this.activeLine.text.substring(0, this.activeLine.caret),
        w = this.context.measureText(t).width;

    this.activeLine.erase(this.context, this.drawingSurface);
    this.activeLine.insert(text);
    this.moveCursor(this.activeLine.left + w, this.activeLine.bottom);
    this.activeLine.draw(this.context);
  }
```

insert() 方法会将当前活动的文本行擦除，然后将文本插入该行，最后再重新绘制这一行内容。此方法也会把光标移动到行内插入文本的地方。回想一下前面讲过的内容：文本行的 erase() 方法会将整个 canvas 的内容都恢复到创建段落之前的样子，在效果上也就等同于擦除了整个段落。

3.4.3.4 向段落中插入新行

如果用户在编辑某一段时按下了 Enter 键，那么图 3-15 所示的应用程序就会调用文本段对象的 newline() 方法，该方法代码如下所示：

```
newline: function () {
    var textBeforeCursor =
            this.activeLine.text.substring(0, this.activeLine.caret),
        textAfterCursor =
            this.activeLine.text.substring(this.activeLine.caret),
        height = this.context.measureText('W').width +
                 this.context.measureText('W').width/6,
        bottom = this.activeLine.bottom + height,
        activeIndex,
        line;

    // Erase paragraph and set active line's text

    this.erase(this.context, this.drawingSurface);
    this.activeLine.text = textBeforeCursor;

    // Create a new line that contains the text after the cursor

    line = new TextLine(this.activeLine.left, bottom);
    line.insert(textAfterCursor);

    // Splice in new line, set active line, and reset caret

    activeIndex = this.lines.indexOf(this.activeLine);

    this.lines.splice(activeIndex+1, 0, line);
    this.activeLine = line;
    this.activeLine.caret = 0;

    // Starting at the new line, loop over remaining lines

    activeIndex = this.lines.indexOf(this.activeLine);

    for(var i=activeIndex+1; i < this.lines.length; ++i) {
        line = this.lines[i];
        line.bottom += height; // Move line down one row
    }

    this.draw();
    this.cursor.draw(this.context, this.activeLine.left,
                     this.activeLine.bottom);
},
```

newline() 方法擦除了光标与其所在的段落本身，创建了一个新的 TextLine 对象，并将它插入 paragraph 对象中所含的 TextLine 数组中。然后，newline() 方法会遍历所有位于新创建的文本行之下的那些文本行，并将其各自下移一行。最后，newline() 方法将更新之后的文本段与光标重新绘制出来。

3.4.3.5 处理段落编辑中的退格操作

文本段对象通过调用 backspace() 方法来处理用户所按下的 Backspace 键，该方法的代码如下所示：

```
backspace: function () {
   var lastActiveLine,
       activeIndex,
       t, w;

   this.context.save();
   if (this.activeLine.caret === 0) {
      if ( ! this.activeLineIsTopLine(); {
         this.erase();
         this.moveUpOneLine();
         this.draw();
      }
   }
   else {   //Active line has text
      this.context.fillStyle = fillStyleSelect.value;
      this.context.strokeStyle = strokeStyleSelect.value;
      this.activeLine.erase(this.context, drawingSurfaceImageData);
      this.activeLine.removeCharacterBeforeCaret();

      t = this.activeLine.text.slice(0, this.activeLine.caret);
      w = this.context.measureText(t).width;

      this.moveCursor(this.activeLine.left + w,
                      this.activeLine.bottom);
      this.activeLine.draw(this.context);
  }
  context.restore();
}
```

backspace() 方法首先检查当前情况是否满足这两个条件：光标位于当前文本行的最左端；当前行不是文本段中的第一行。如果同时符合这两个条件，那么该方法就将整个文本段都擦掉，并将当前文本行之下的各行依次上移一行，并重新绘制修改之后的文本段。否则，backspace() 方法就会将光标之前的那个字符删去，并重新绘制当前这行文本。

程序清单 3-19 完整地列出了 Paragraph 对象的代码。

程序清单 3-19　Paragraph 对象的代码

```
// Constructor.........................................................

Paragraph = function (context, left, top, imageData, cursor) {
   this.context = context;
   this.drawingSurface = imageData;
   this.left = left;
   this.top = top;
   this.lines = [];
   this.activeLine = undefined;
   this.cursor = cursor;
   this.blinkingInterval = undefined;
};

// Prototype...........................................................

Paragraph.prototype = {
```

```
isPointInside: function (loc) {
   var c = this.context;

   c.beginPath();
   c.rect(this.left, this.top,
          this.getWidth(), this.getHeight());

   return c.isPointInPath(loc.x, loc.y);
},

getHeight: function () {
   var h = 0;

   this.lines.forEach( function (line) {
      h += line.getHeight(this.context);
   });

   return h;
},

getWidth: function () {
   var w = 0,
       widest = 0;

   this.lines.forEach( function (line) {
        w = line.getWidth(this.context);
        if (w > widest) {
           widest = w;
        }
   });

   return widest;
},

draw: function () {
   this.lines.forEach( function (line) {
      line.draw(this.context);
   });
},

erase: function (context, imageData) {
   context.putImageData(imageData, 0, 0);
},

addLine: function (line) {
   this.lines.push(line);
   this.activeLine = line;
   this.moveCursor(line.left, line.bottom);
},

insert: function (text) {
  this.erase(this.context, this.drawingSurface);
  this.activeLine.insert(text);

  var t =  this.activeLine.text.substring(0, this.activeLine.caret),
      w =  this.context.measureText(t).width;

  this.moveCursor(this.activeLine.left + w,
                  this.activeLine.bottom);

  this.draw(this.context);
```

```
},

blinkCursor: function (x, y) {
   var self = this,
       BLINK_OUT = 200,
       BLINK_INTERVAL = 900;

   this.blinkingInterval = setInterval( function (e) {
      cursor.erase(context, self.drawingSurface);

      setTimeout( function (e) {
         cursor.draw(context, cursor.left,
                     cursor.top + cursor.getHeight(context));
      }, BLINK_OUT);
   }, BLINK_INTERVAL);
},

moveCursorCloseTo: function (x, y) {
   var line = this.getLine(y);

   if (line) {
      line.caret = this.getColumn(line, x);
      this.activeLine = line;
      this.moveCursor(line.getCaretX(context), line.bottom);
   }
},

moveCursor: function (x, y) {
    this.cursor.erase(this.context, this.drawingSurface);
    this.cursor.draw(this.context, x, y);

    if ( ! this.blinkingInterval)
       this.blinkCursor(x, y);
},

moveLinesDown: function (start) {
   for (var i=start; i < this.lines.length; ++i) {
      line = this.lines [i];
      line.bottom += line.getHeight(this.context);
   }
},

newline: function () {
   var textBeforeCursor =
          this.activeLine.text.substring(0, this.activeLine.caret),
       textAfterCursor =
          this.activeLine.text.substring(this.activeLine.caret),
       height = this.context.measureText('W').width +
                this.context.measureText('W').width/6,

       bottom = this.activeLine.bottom + height,
       activeIndex,
       line;

   // Erase paragraph and set active line's text

   this.erase(this.context, this.drawingSurface);
   this.activeLine.text = textBeforeCursor;

   // Create a new line that contains the text after the cursor

   line = new TextLine(this.activeLine.left, bottom);
```

```
      line.insert(textAfterCursor);

      // Splice in new line, set active line, and reset caret

      activeIndex = this.lines.indexOf(this.activeLine);
      this.lines.splice(activeIndex+1, 0, line);

      this.activeLine = line;
      this.activeLine.caret = 0;

      // Starting at the new line, loop over remaining lines

      activeIndex = this.lines.indexOf(this.activeLine);

      for(var i=activeIndex+1; i < this.lines.length; ++i) {
         line = this.lines [i];
         line.bottom += height;  // Move line down one row
      }

      this.draw();
      this.cursor.draw(this.context, this.activeLine.left,
                       this.activeLine.bottom);
   },

   getLine: function (y) {
      var line;
      for (i=0; i < this.lines.length; ++i) {
         line = this .lines[i];
         if (y > line.bottom - line.getHeight(context) &&
            y < line.bottom) {
            return line;
         }
      }
      return undefined;
   },

      getColumn: function (line, x) {
         var found = false,
             before,
             after,
             closest,
             tmpLine,
             column;

         tmpLine = new TextLine(line.left, line.bottom);
         tmpLine.insert(line.text);

         while ( ! found && tmpLine.text.length > 0) {
            before = tmpLine.left + tmpLine.getWidth(context);
            tmpLine.removeLastCharacter();
            after = tmpLine.left + tmpLine.getWidth(context);

            if (after < x) {
               closest = x - after < before - x ? after : before;
               column = closest === before ?
                        tmpLine.text.length + 1 : tmpLine.text.length;
               found = true;
            }
         }
         return column;
      },
```

```
activeLineIsOutOfText: function () {
   return this.activeLine.text.length === 0;
},

activeLineIsTopLine: function () {
   return this.lines[0] === this.activeLine;
},

moveUpOneLine: function () {
   var lastActiveText, line, before, after;

   lastActiveLine = this.activeLine;
   lastActiveText = '' + lastActiveLine.text;

   activeIndex = this.lines.indexOf(this.activeLine);
   this.activeLine = this.lines [activeIndex - 1];
   this.activeLine.caret = this.activeLine.text.length;

   this.lines.splice(activeIndex, 1);

   this.moveCursor(
      this.activeLine.left + this.activeLine.getWidth(this.context),
      this.activeLine.bottom);

   this.activeLine.text += lastActiveText;

   for (var i=activeIndex; i < this.lines.length; ++i) {
      line = this.lines[i];
      line.bottom -= line.getHeight(this.context);
   }
},

backspace: function () {
   var lastActiveLine,
       activeIndex,
       t, w;

   this.context.save();

   if (this.activeLine.caret === 0) {
      if ( ! this.activeLineIsTopLine()) {
         this.erase(this.context, this.drawingSurface);
         this.moveUpOneLine();
         this.draw();
      }
   }

   else {  // Active line has text
      this.context.fillStyle = fillStyleSelect.value;
      this.context.strokeStyle = strokeStyleSelect.value;

      this.erase(this.context, this.drawingSurface);
      this.activeLine.removeCharacterBeforeCaret();

      t = this.activeLine.text.slice(0, this.activeLine.caret),
      w = this.context.measureText(t).width;

      this.moveCursor(this.activeLine.left + w,
                      this.activeLine.bottom);

      this.draw(this.context);
```

```
            context.restore();
        }
    }
};
```

提示：WHATWG 的 Canvas 规范最佳实践：开发者不要自己去实现文本控件

WHATWG 的 Canvas 规范之中有一小段是用于讲述最佳实践的。其中一条建议：开发者不要自己去实现 Canvas 元素内的文本编辑控件，而是应该将 HTML 的 input 或 textarea 元素与 HTML5 中的 contenteditable 属性结合起来使用。（而在 W3C 版本的 Canvas 规范中有关 2d 绘图环境的描述里，则没有这样的注释）（最新版的 W3C Canvas 规范之中已经包含这段建议了，请参见：http://www.w3.org/TR/2dcontext/#best-practices。——译者注）

为什么不应该自己去实现文本控件呢？因为根据 WHATWG 所定规范的说法，实现文本控件所需的工作量太大了。如果想要实现一个有用的文本编辑控件，开发者必须实现诸如复制与粘贴、拖放、文本选择、文本滚动等功能，因为默认的 canvas 元素并不包含以上这些功能。

不过，我们不能仅仅因为 WHATWG 所定的规范不鼓励开发者自行实现文本编辑控件，就盲目地听从他们的建议。实际上，有些人已经开始编写自己的文本编辑控件了，比如 Bespin 编辑器（该编辑器所在项目曾用名“Mozilla Skywriter”，现已更名为 ACE，是一款为“Cloud9 IDE”云端集成开发环境所使用的开源网页编辑器，其官方网站是：http://ace.ajax.org/。——译者注）就是这样。就像对待其他最佳实践所采取的态度那样，我们对于这一条建议，也应该有保留地接受才对。开发者需要自己来判断是否值得去亲自开发文本控件。我们要注意的问题是，canvas 元素本身对于文本的支持极为有限，但是亲自实现文本控件却又要增加额外的工作量。

3.5 总结

Canvas 提供了一些可用来操作文本的基本功能，然而，它并不支持一些复杂的文本操作，诸如沿着圆弧绘制文本或编辑文本行等等。

在本章中，读者学到了如何利用 Canvas 所提供的这套极为有限的文本 API 来实现一些复杂的文本处理功能，包括沿着某段圆弧绘制文本。读者也了解到如何在坐标轴及数据仪表盘上绘制文本标签，如何设置对齐方式、字型等文本绘制属性。

本章后半部分向读者演示了如何实现文本控件。我们一开始先实现了一个用于插入文本的光标，然后讲解了如何实现可以编辑的文本行，最后告诉大家如何实现文本段的编辑。如果你想要实现自己的文本控件，那么这些范例程序中所编写的对象就是个良好的起点。

下一章我们将研究如何在 canvas 之中显示并操作图像。

第4章
图像与视频

HTML5 的 Canvas 元素提供了极为丰富的图像支持。开发者可以选择绘制某幅图像的全部或某个部分，可以在绘制的时候缩放或保持原样，可以将图片绘制在 canvas 中的任何地方，也可以操作每个像素的颜色及透明度。而且，如果我们将这些图像操作同剪切区域、离屏 canvas 等其他方面的 Canvas API 结合起来，那么就可以创建出很棒的图形效果来，比如制作动画与多人游戏，实现数据可视化（data visualization），或是模拟粒子物理学（particle physics）等。

图 4-1 中的这个放大镜程序就演示了一些我们可以利用 Canvas 的图形处理功能而制作出来的显示效果。当用户拖动放大镜时，应用程序就会对放大镜下方的像素进行放大，并将其绘制到 canvas 之中放大镜镜片的范围内。

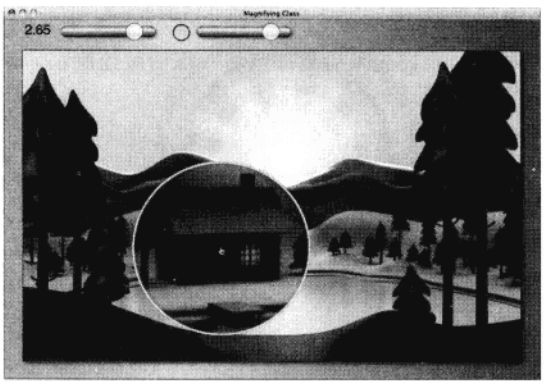


图 4-1　利用像素放大及剪辑区域技术来实现的放大镜程序

Canvas 的绘图环境对象提供了如下 4 个用于绘制及操作图像的方法：

- drawImage()
- getImageData()
- putImageData()
- createImageData()

大家可以想到的是，drawImage() 方法应该能把某幅图像绘制到 canvas 之中，然而大家也许猜不到，这个方法还能把另外一个 canvas 的内容或者某个视频的其中一帧绘制到当前 canvas 之中。这听起来是个非常振奋人心的功能[⊖]。

⊖ 原文为“That's a large can of whoopass”，意为“那感觉像是喝了一大罐WhoopAss一样”。WhoopAss是由“琼斯苏打水”（Jones Soda）公司出品的一款能量型饮料，详情参见：http://en.wikipedia.org/wiki/Jones_Soda#WhoopAss_and_Jones_Energy。——译者注

与图像数据有关的那两个方法，可以让开发者获取并操作图像之中的某个单独像素。getImageData() 方法用于获取图像中的底层像素，而 putImageData() 方法则可以将修改后的像素值放回到图像中去。这样一来，我们就可以总结出一套用于操作像素数据的方式了，虽说未必适用于所有情况，不过我们还是会在 4.5.1 小节之中讲讲它的。

我们也可以用 createImageData() 方法来创建一个表示空白图像的数据对象。可以将以 CSS 像素为单位的宽度与高度值传给该方法，作为图像数据的大小，也可以向它传递一个已经存在的 ImageData 对象，这样的话，该方法所返回的空白 ImageData 对象将会具有与传入对象相同的宽度和高度。

4.1 图像的绘制

drawImage() 方法可以将一幅图像的整体或某个部分绘制到 canvas 内的任何位置上，并且允许开发者在绘制过程中对图像进行缩放。也可以将图像绘制在离屏 canvas 中，这样的话，就可以对图像进行一些有技巧性的处理了，例如实现图像查看器，或是将某幅图像淡出至 canvas 中。在本章中我们将讨论离屏 canvas 的多种用途。

4.1.1 在 Canvas 之中绘制图像

通过图 4-2 所示应用程序，我们先来看看如何将一幅图像绘制在 canvas 中。程序清单 4-1 列出了该范例程序的代码。

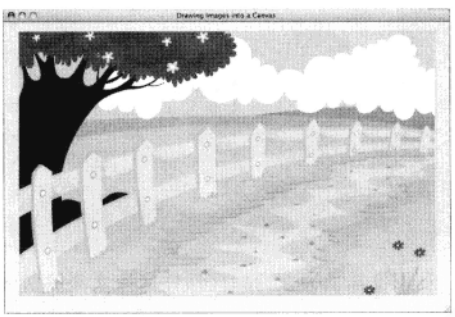


图 4-2 在 canvas 中绘制图像

程序清单 4-1 图像的绘制

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    image = new Image();

image.src = 'fence.png';
image.onload = function(e) {
   context.drawImage(image, 0, 0);
};
```

程序清单 4-1 的代码首先创建了一幅图像，设置了它的数据源，然后等待浏览器加载图片，在图片加载完成后，将其绘制于 canvas 的左上角。

这就是 drawImage() 最简单的用法了。采用这种方式，可以把一整张未经缩放的图像绘制到

canvas 之中，该方式的唯一缺点则是你必须等待图像加载完毕之后才能对其进行绘制。如果在图片尚未完成加载时就进行绘制，那么根据 Canvas 规范，drawImage() 方法的执行会失败，而且没有任何提示。

警告：在图像未被加载之前不得对其进行绘制

使用 drawImage() 方法可以将图像绘制到 canvas 之中，不过，如果在绘制时图像尚未完全加载，那么 drawImage() 方法则什么都不会做。在使用 drawImage() 方法时，务必保证所绘图像已经加载好了，通常我们会将其放在 onload 回调函数中以确保这一点。

警告：Canvas 规范与浏览器实现之间是存在差别的

根据 Canvas 规范所述，如果使用 drawImage() 方法来绘制尚未加载的图片，则该方法的执行会在没有任何提示的情况下失败。然而，很多浏览器却会通过抛出异常来指示这一情况。读者可以访问如下网址来确认各个浏览器的具体做法：http://bit.ly/ilW6ET。

一般来说，大家要记住：浏览器并不总是完全遵从 Canvas 规范。所以，很有必要写一组测试套件来测试一下各个浏览器对于 Canvas 规范的遵守程度。更多信息请参见：http://w3c-test.org/。

小技巧：图像的绘制效果受制于阴影、剪辑区域、图像合成等属性

drawImage() 方法在绘制图像时不会考虑当前路径，然而，它却会将 globalAlpha 设置、阴影效果、剪辑区域，以及全局图像合成操作符等属性运用到图像的绘制之中。

小技巧：图像的加载

很多应用程序在开始运行之前需要加载大量的图片，游戏程序就是一个典型的例子。在本书 9.1.2 小节中，你将会看到如何在加载多张图像时显示一个用以指示加载进度的滚动条。

读者已经学会了如何将图像绘制到 canvas 之中，接下来，我们就详细地讲讲 drawImage() 方法。

4.1.2 drawImage() 方法的用法

drawImage() 方法的用法如图 4-3 所示。

drawImage() 方法会将一幅图像绘制到一个 canvas 之中，所绘的图像叫做“源图像”（source image），而绘制到的地方则叫做“目标 canvas”（destination canvas）。在图 4-3 中，以字母“s”开头的变量名代表源图像，而以字母“d”开头的变量则用于指示目标 canvas。drawImage() 方法可以接受以下 3 套参数：

- drawImage(image, dx, dy)
- drawImage(image, dx, dy, dw, dh)
- drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)

在上述三种情况下，第一个参数都是 HTMLImageElement 类型的图像对象，不过，它也可以是一个 HTMLCanvasElement 类型的 canvas 对象，或 HTMLVideoElement 类型的视频对象。所以，开发者也可以将 canvas 或视频对象当成图像来用，这样一来，便催生了诸如视频编辑器这样的一大批应用程序。

上述 drawImage() 方法的第一种用法，会将整幅图像绘制在目标 canvas 中的指定位置上。

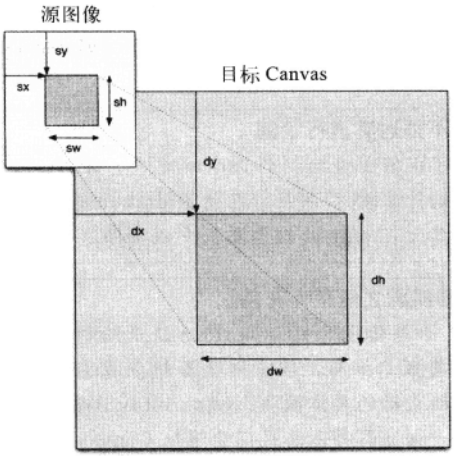


图 4-3 drawImage() 方法可以将整张图像或其中的一部分绘制到 canvas 之中，绘制时可以选择进行缩放或保持原样

drawImage() 方法的第二种用法，会将图像完整地绘制到指定的位置上，然而，在绘制时会根据目标区域的宽度与高度进行缩放。

drawImage() 方法的第三种用法则可以将整幅图像或其一部分绘制到目标 canvas 的指定位置上，而且在绘制时会根据目标区域的宽度与高度对图像进行缩放。

表 4-1 总结了 drawImage() 方法的用法。

表 4-1 drawImage() 方法的用法

方　法	描　述
drawImage(HTMLImageElement image, double sx, double sy, double sw, double sh, double dx, double dy, double dw, double dh);	将图像绘制到 canvas 之中。如果该图像参数是一个 HTMLVideoElement 类型的视频对象，那么 drawImage() 方法就会将视频的当前帧绘制出来。这个图像参数也可以是另外一个 HTMLCanvasElement 的 canvas 对象 向 canvas 之中绘制的图像可以是一整幅，也可以是它的一部分，并且在绘制时有可能要对图像进行缩放。源图像中需要被绘制的区域是通过 sx、sy、sw 与 sh 参数来确定的，浏览器会根据 dw 与 dh 参数对所绘内容进行缩放。在该方法的各种用法中，只有前三个参数是必需的

小技巧：图像、canvas 及视频都可以绘制到 canvas 之中

drawImage() 方法的使用很灵活：可以将一幅图像、一个 canvas 对象或一个视频帧的整体或某个部分绘制到 canvas 中。在绘制这些图像、canvas 对象或视频帧的时候，可以任意指定其绘制位置及缩放比例。

4.2 图像的缩放

读者已经学会了如何使用 drawImage() 方法将一幅未经缩放的图像绘制到 canvas 之中。现在我们就来看看如何用该方法在绘制图像的时候进行缩放，如图 4-4 所示。

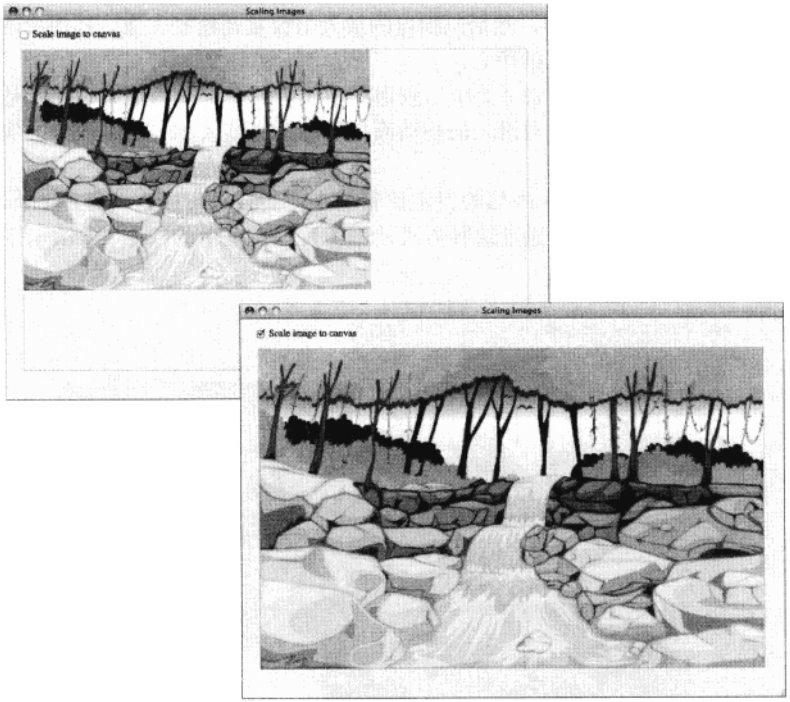


图 4-4 图像的缩放

图 4-4 上方的图像其尺寸比 canvas 小，然而，当用户选中复选框之后，应用程序则会重新绘制该图，将其放大，以符合 canvas 的尺寸。此时的绘制效果如图 4-4 下方的图像所示。

图 4-4 所示应用程序之中用于绘制图像的方法，其代码列在了程序清单 4-2 之中。

程序清单 4-2 图像的缩放

```
function drawImage() {
    context.clearRect(0, 0, canvas.width, canvas.height);

    if (scaleCheckbox.checked) {
        context.drawImage(image, 0, 0, canvas.width, canvas.height);
    }
    else {
        context.drawImage(image, 0, 0);
    }
}
```

如果用户选中了复选框，那么该函数就会在绘制时将图像缩放至与 canvas 相同的大小。否则，它就直接绘制未经缩放的图像。在这两种情况下，函数都会把图像绘制在 canvas 的 (0,0) 坐标处。

在 Canvas 边界之外绘制图像

程序清单 4-2 所列的应用程序将图像绘制在了 canvas 中的 (0,0) 这个点上，不过也可以通过指定非零的坐标值来将图像绘制在 canvas 中的任意位置上，图 4-5 所示应用程序演示了这种用法。

该应用程序提供了一个滑动条，可以让用户来调整图像的放大倍数。当用户拖动滑动条时，应用程序会清除 canvas 之中的内容，然后以指定的放大倍数重新绘制图像。还需要注意的是，该应用程序总是把图像绘制在 canvas 的中心。

除了会对图像进行缩放之外，图 4-5 所示应用程序还有一个有趣的地方，那就是左上角用于表示放大倍数的文字，会随着用户对滑块的拖动而持续地改变其大小。这种效果的实现代码列在了程序清单 4-5 之中。

不仅可以将图片绘制在 canvas 内部的指定位置之上，而且也可以将其绘制到 canvas 的范围之外。图 4-5 所示的应用程序正是通过这种方式使得图片保持居中的，图 4-6 演示了该应用程序的绘制原理。

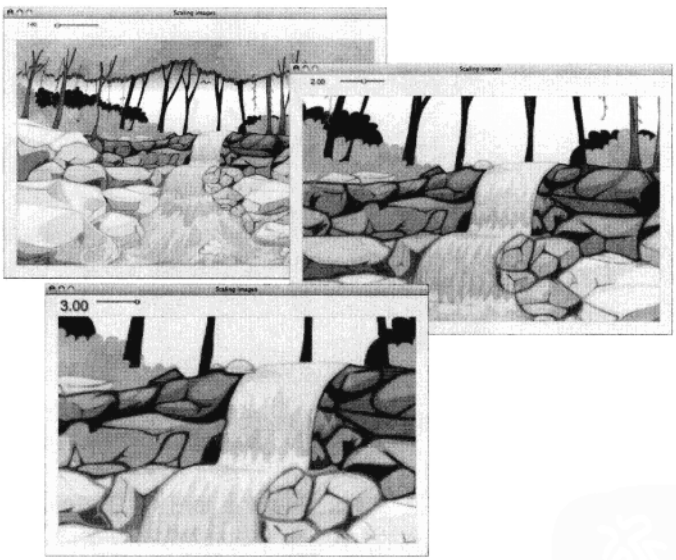


图 4-5　将缩放后的图片绘制于 Canvas 中央

图 4-6 演示了图 4-5 中的应用程序在以 2.0 的放大倍数来绘制图像时是如何运作的。为了将图像中位于 canvas 之内的部分同位于其外的部分区分开，图 4-6 使用与原图相同的颜色来表示绘制在 canvas 之内的这部分图像，而以褪色效果来表示位于 canvas 边界以外的那部分图像。

该应用程序将所绘图像的宽度与高度分别乘以用户选定的放大倍数，然后据此来计算图像左上角的绘制坐标。这段代码如程序清单 4-3 所示。

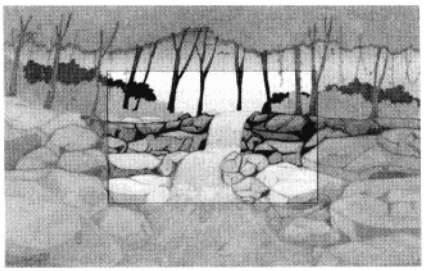


图 4-6　图 4.5 中的应用程序所绘制的完整图像，图像中位于 canvas 范围之外的部分以深色表示

程序清单 4-3　将缩放后的图像绘制于 Canvas 的中心

```
function drawImage() {
    var w = canvas.width,
        h = canvas.height,
        sw = w * scale,
        sh = h * scale;

    context.clearRect(0, 0, w, h);
    context.drawImage(image, -sw/2 + w/2, -sh/2 + h/2, sw, sh);
}
```

图 4-5 所示应用程序的 HTML 代码列在了程序清单 4-4 之中，其 JavaScript 代码列在了程序清单 4-5 之中。

程序清单 4-4　图像的缩放（HTML 代码）

```
<!DOCTYPE html>
<html>
    <head>
        <title>Scaling images</title>

        <style>
            body {
                background: rgba(100, 145, 250, 0.3);
            }

            #scaleSlider {
                vertical-align: 10px;
                width: 100px;
                margin-left: 90px;
            }

            #canvas {
                margin: 10px 20px 0px 20px;
                border: thin solid #aaaaaa;
                cursor: crosshair;
            }

            #controls {
                margin-left: 15px;
                padding: 0;
            }

            #scaleOutput {
```

```
                position: absolute;
                width: 60px;
                height: 30px;
                margin-left: 10px;
                vertical-align: center;
                text-align: center;
                color: blue;
                font: 18px Arial;
                text-shadow: 2px 2px 4px rgba(100, 140, 250, 0.8);
            }

        </style>
    </head>
    <body>
        <div id='controls'>
            <output id='scaleOutput'>1.0</output>
            <input id='scaleSlider' type='range'
                   min='1' max='3.0' step='0.01' value='1.0'/>
        </div>

        <canvas id='canvas' width='800' height='520'>
            Canvas not supported
        </canvas>

        <script src='example.js'></script>
    </body>
</html>
```

程序清单 4-5　图像的缩放（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    image = new Image(),

    scaleSlider = document.getElementById('scaleSlider'),
    scale = 1.0,
    MINIMUM_SCALE = 1.0,
    MAXIMUM_SCALE = 3.0;

// Functions.........................................................

function drawImage() {
    var w = canvas.width,
        h = canvas.height,
        sw = w * scale,
        sh = h * scale;

    context.clearRect(0, 0, canvas.width, canvas.height);
    context.drawImage(image, -sw/2 + w/2, -sh/2 + h/2, sw, sh);
}

function drawScaleText(value) {
    var text = parseFloat(value).toFixed(2);
    var percent = parseFloat(value - MINIMUM_SCALE) /
                  parseFloat(MAXIMUM_SCALE - MINIMUM_SCALE);

    scaleOutput.innerText = text;
    percent = percent < 0.35 ? 0.35 : percent;
    scaleOutput.style.fontSize = percent*MAXIMUM_SCALE/1.5 + 'em';
}
```

```
// Event handlers.....................................................

scaleSlider.onchange = function(e) {
    scale = e.target.value;

    if (scale < MINIMUM_SCALE) scale = MINIMUM_SCALE;
    else if (scale > MAXIMUM_SCALE) scale = MAXIMUM_SCALE;

    drawScaleText(scale);
    drawImage();
};

// Initialization....................................................

context.fillStyle = 'cornflowerblue';
context.strokeStyle = 'yellow';
context.shadowColor = 'rgba(50, 50, 50, 1.0)';
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur = 10;

image.src = 'waterfall.png';

image.onload = function(e) {
    drawImage();
    drawScaleText(scaleSlider.value);
};
```

小技巧：可以在 canvas 范围之外绘制图像

图像可以绘制在 canvas 之内，也可以绘制在它之外。比方说，程序清单 4-5 中所列的应用程序代码，在放大倍数大于 1.0 的情况下，就会把图像的绘制点指定到 canvas 外面去。

如果你向 canvas 之中绘制的图像有一部分会落在 canvas 的范围之外，那么浏览器就会将 canvas 范围外的那部分图像忽略。

可以在 canvas 范围之外进行绘制，这是一项重要的功能。例如在实现 5.7 节之中所讲述的背景图片滚动效果时，我们将把图像绘制在 canvas 范围外，并且通过平移 canvas 的坐标系来让背景中的某一部分内容显示在当前视窗范围内。

小技巧：进行页面渲染时不应首先考虑 Canvas 元素

程序清单 4-5 所列的应用程序代码，会在用户调整滑动条时，根据放大倍数来缩放屏幕左上角所显示的读数。读者也许立刻就能想到：可以用 canvas 元素来实现放大倍数的显示效果，我们可以在 canvas 之中调用 fillText() 方法来绘制数值，并且根据滑动条所指定的放大倍数对 canvas 进行缩放。然而，Canvas 规范中却是这么说的：

如果有一个更为合适的元素可以使用，那么开发者就不应该考虑在 HTML 文档中使用 canvas 元素。比如，用 canvas 元素来渲染页面标题，就是一个不恰当的做法。

对于程序清单 4-5 所展示的这个应用程序来说，使用 output 元素比使用 canvas 元素更为合适，而且实现起来也更加简单。

4.3 将一个 Canvas 绘制到另一个 Canvas 之中

图 4-7 所示应用程序将一幅图像绘制在 canvas 之中，然后在图像上方绘制了一些用做水印的文本。

图 4-7 水印效果的绘制

当用户使用左上角的滑动条来调整放大倍数时，应用程序会同时对图像与文本进行缩放。可以先把图像与文本按照滑动条所定的比例进行放大，将其绘制到一个离屏的 canvas 之中，然后再把离屏 canvas 之中的内容复制到屏幕上正在显示的这个 canvas 里面。然而，像本例这种情况，严格来讲并不是非得使用离屏 canvas 才行，因为 drawImage() 方法也可以将 canvas 的内容绘制到自己身上，方法如下：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    scaleWidth = ..., // Calculate scales for width and height
    scaleHeight = ...;

...
context.drawImage(canvas, 0, 0, scaleWidth, scaleHeight);
...
```

上述代码会将缩放之后的 canvas 内容绘制到本 canvas 之中。当用户调整放大倍数之后，应用程序会清除掉 canvas 之中的现有内容，把图像缩放到与 canvas 相同的宽度与高度，并将其绘制到 canvas 之中，然后再于 canvas 图像的上方绘制水印。

然而，用户最终将会看到的并不是处于该状态的 canvas 内容，因为，应用程序在执行完上述操作之后，会立刻将 canvas 之中的内容按照用户所指定的放大倍数绘制到自身。这样所造成的效果是：不仅仅是图像，就连水印也随之一并被放大了。

虽说将 canvas 绘制到自身的确很方便，不过在本例中，这么做的效率并不高。每当用户改动了放大倍数之后，应用程序都要将图像和水印绘制到 canvas 之中，然后再根据指定的缩放比例，将放大后的 canvas 重新绘制一遍。这意味每当放大倍数改变时，应用程序都要把所有的内容绘制两次才行。该程序的全部 JavaScript 代码如程序清单 4-6 所示。

程序清单 4-6 水印的绘制（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    image = new Image(),

    scaleOutput = document.getElementById('scaleOutput');
    scaleSlider = document.getElementById('scaleSlider'),
    scale = scaleSlider.value,
    scale = 1.0,

    MINIMUM_SCALE = 1.0,
    MAXIMUM_SCALE = 3.0;
// Functions.............................................................
```

```
function drawScaled() {
   var w = canvas.width,
       h = canvas.height,
       sw = w * scale,
       sh = h * scale;

   // Clear the canvas, and draw the image scaled to canvas size

   context.clearRect(0, 0, canvas.width, canvas.height);
   context.drawImage(image, 0, 0, canvas.width, canvas.height);

   // Draw the watermark on top of the image

   drawWatermark();

   // Finally, draw the canvas scaled according to the current
   // scale, back into itself. Note that the source and
   // destination canvases are the same canvas.

   context.drawImage(canvas, 0, 0, canvas.width, canvas.height,
                     -sw/2 + w/2, -sh/2 + h/2, sw, sh);
}

function drawScaleText(value) {
   var text = parseFloat(value).toFixed(2);
   var percent = parseFloat(value - MINIMUM_SCALE) /
                 parseFloat(MAXIMUM_SCALE - MINIMUM_SCALE);

   scaleOutput.innerText = text;
   percent = percent < 0.35 ? 0.35 : percent;
   scaleOutput.style.fontSize = percent*MAXIMUM_SCALE/1.5 + 'em';
}

function drawWatermark() {
   var lineOne = 'Copyright',
       lineTwo = 'Acme Inc.',
       textMetrics,
       FONT_HEIGHT = 128;

   context.save();
   context.font = FONT_HEIGHT + 'px Arial';

   textMetrics = context.measureText(lineOne);

   context.globalAlpha = 0.6;
   context.translate(canvas.width/2,
                     canvas.height/2-FONT_HEIGHT/2);

   context.fillText(lineOne, -textMetrics.width/2, 0);
   context.strokeText(lineOne, -textMetrics.width/2, 0);

   textMetrics = context.measureText(lineTwo);
   context.fillText(lineTwo, -textMetrics.width/2, FONT_HEIGHT);
   context.strokeText(lineTwo, -textMetrics.width/2, FONT_HEIGHT);

   context.restore();
}

// Event handlers.....................................................

scaleSlider.onchange = function(e) {
   scale = e.target.value;

   if (scale < MINIMUM_SCALE) scale = MINIMUM_SCALE;
   else if (scale > MAXIMUM_SCALE) scale = MAXIMUM_SCALE;

   drawScaled();
   drawScaleText(scale);
}

// Initialization....................................................

context.fillStyle     = 'cornflowerblue';
context.strokeStyle   = 'yellow';
context.shadowColor   = 'rgba(50, 50, 50, 1.0)';
context.shadowOffsetX = 5;
```

```
context.shadowOffsetY = 5;
context.shadowBlur    = 10;

var glassSize = 150;
var scale = 1.0;

image.src = 'lonelybeach.png';
image.onload = function(e) {
   context.drawImage(image, 0, 0, canvas.width, canvas.height);
   drawWatermark();
   drawScaleText(scaleSlider.value);
};
```

尽管采用离屏 canvas 技术需要多写一点代码，不过在本例这种情况下，做这些努力还是值得的，因为这么做可以极大地提高绘制效率。接下来我们就看看如何采用离屏 canvas 技术，来重新实现这个绘制水印的范例应用程序。

小技巧：可以将 canvas 的内容绘制到其自身，不过要小心

drawImage() 方法可以将一个 canvas 的内容绘制到另外一个 canvas 之上。也可以将 canvas 的内容绘制到其自身。在某些情况下，这么做确实很有用，比如程序清单 4-6 所展示的应用程序代码就利用这种方法来放大 canvas 之中的内容。然而，这种做法的绘制效率并不是很高，因为浏览器必须创建一个起中介作用的离屏 canvas 来存储放大之后的 canvas 图像。

4.4 离屏 canvas

离屏 canvas 经常用来存放临时性的图像信息，这在很多情况下都是非常有用的。举例来说，图 4-1 中所示的放大镜程序，就使用了一个离屏 canvas 来存储放大之后的那部分屏幕图像，然后再将离屏 canvas 里面的内容复制到正在显示的 canvas 之中。

图 4-8 演示了离屏 canvas 的另一种用法。在本例这种情况下，离屏 canvas 里面存储的是未经放大的图像与水印。当用户拖动滑动条时，应用程序会将离屏 canvas 里面的内容复制到正在显示的 canvas 之中，在这个过程中也会根据用户指定的数值对所绘内容进行放大。

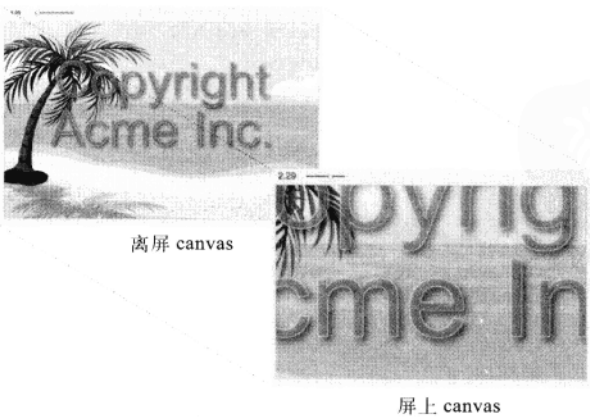


图 4-8 离屏 canvas

要使用离屏 canvas，通常得遵循如下四个步骤：

1. 创建用做离屏 canvas 的元素。
2. 设置离屏 canvas 的宽度与高度。
3. 在离屏 canvas 之中进行绘制。
4. 将离屏 canvas 的全部或一部分内容复制到正在显示的 canvas 之中。

程序清单 4-7 中所列的代码演示了上述步骤。

程序清单 4-7　使用离屏 canvas 的步骤

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),
...
// Set the offscreen canvas's size to match the onscreen canvas
offscreenCanvas.width = canvas.width;
offscreenCanvas.height = canvas.height;
...
// Draw into the offscreen context
offscreenContext.drawImage(anImage, 0, 0);
...
// Draw the offscreen context into the onscreen canvas
context.drawImage(offscreenCanvas, 0, 0,
                  offscreenCanvas.width, offscreenCanvas.height);
```

可以用如下代码来创建离屏 canvas：var offscreenCanvas = document. createElement('canvas');。这一行代码会创建一个不从属于任何 DOM [⊖]元素的 canvas 对象，因此该 canvas 是不可见的，这也就是它为何被叫做“离屏”（offscreen）canvas 的原因。

在默认情况下，这个离屏 canvas 的大小与 canvas 的默认值一样，是 300 像素宽，150 像素高。这样的尺寸一般来说并不符合开发者要实现的功能，所以需要重新调整它的大小。

在创建离屏 canvas 并调整好它的大小之后，通常我们就会在其上进行绘制，然后将离屏 canvas 的一部分或全部内容复制到屏幕 canvas 之中。

程序清单 4-8 列出了图 4-8 所示应用程序的代码。

程序清单 4-8　离屏 canvas 的使用

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),

    image = new Image(),

    scaleOutput = document.getElementById('scaleOutput'),
    canvasRadio = document.getElementById('canvasRadio'),
    imageRadio = document.getElementById('imageRadio'),

    scale = scaleSlider.value,
```

⊖ Document Object Model，简称DOM，是W3C组织推荐的处理可扩展置标语言的标准编程接口。详情参见：http://www.w3.org/DOM/。——译者注

```
    scale = 1.0,
    MINIMUM_SCALE = 1.0,
    MAXIMUM_SCALE = 3.0;

// Functions.........................................................

function drawScaled() {
   var w = canvas.width,
       h = canvas.height,
       sw = w * scale,
       sh = h * scale;

   context.drawImage(offscreenCanvas, 0, 0,
                     offscreenCanvas.width, offscreenCanvas.height,
                     -sw/2 + w/2, -sh/2 + h/2, sw, sh);
}

function drawScaleText(value) {
   var text = parseFloat(value).toFixed(2);
   var percent = parseFloat(value - MINIMUM_SCALE) /
                 parseFloat(MAXIMUM_SCALE - MINIMUM_SCALE);

   scaleOutput.innerText = text;
   percent = percent < 0.35 ? 0.35 : percent;
   scaleOutput.style.fontSize = percent*MAXIMUM_SCALE/1.5 + 'em';
}

function drawWatermark(context) {
   var lineOne = 'Copyright',
       lineTwo = 'Acme, Inc.',
       textMetrics = null,
       FONT_HEIGHT = 128;

   context.save();
   context.fillStyle = 'rgba(100,140,230,0.5);';
   context.strokeStyle = 'yellow';
   context.shadowColor = 'rgba(50, 50, 50, 1.0)';
   context.shadowOffsetX = 5;
   context.shadowOffsetY = 5;
   context.shadowBlur = 10;

   context.font = FONT_HEIGHT + 'px Arial';
   textMetrics = context.measureText(lineOne);
   context.translate(canvas.width/2, canvas.height/2);
   context.fillText(lineOne, -textMetrics.width/2, 0);
   context.strokeText(lineOne, -textMetrics.width/2, 0);

   textMetrics = context.measureText(lineTwo);
   context.fillText(lineTwo, -textMetrics.width/2, FONT_HEIGHT);
   context.strokeText(lineTwo, -textMetrics.width/2, FONT_HEIGHT);
   context.restore();
}

// Event handlers....................................................

scaleSlider.onchange = function(e) {
   scale = e.target.value;

   if (scale < MINIMUM_SCALE) scale = MINIMUM_SCALE;
   else if (scale > MAXIMUM_SCALE) scale = MAXIMUM_SCALE;
   drawScaled();
   drawScaleText(scale);
}

// Initialization...................................................

offscreenCanvas.width = canvas.width;
```

```
offscreenCanvas.height = canvas.height;

image.src = 'lonelybeach.png';
image.onload = function(e) {
   context.drawImage(image, 0, 0, canvas.width, canvas.height);
   offscreenContext.drawImage(image, 0, 0,
                              canvas.width, canvas.height);
   drawWatermark(context);
   drawWatermark(offscreenContext);
   drawScaleText(scaleSlider.value);
};
```

小技巧：使用离屏 canvas 来提高绘图效率

离屏 canvas 会占据一定的内存，不过它们可以显著地提高绘图效率。

请注意看程序清单 4-8 之中 drawScaled() 方法的绘制效率比程序清单 4-6 之中的方法提高了多少。程序清单 4-8 之中的这个应用程序，是借助离屏 canvas 来绘制的，而程序清单 4-6 之中的那个程序，则必须先清除当前 canvas，然后在其上绘制图像及水印，最后再把绘制好的内容复制到 canvas 自身。

讲完了如何绘制并缩放图像，如何将其绘制到离屏 canvas 等这些技术之后，咱们再来看看怎么操作图像中的单个像素。

4.5 操作图像的像素

getImageData() 与 putImageData() 这两个方法分别用来获取图像的像素信息，以及向图像中插入像素。与此同时，如果有需要，也可以修改像素的值，所以说，这两个方法能够让开发者对图像之中的像素进行任何可以想见的操作。

4.5.1 获取图像数据

咱们先来讲一个常见的例子：如何实现用橡皮筋式选取框来选中 canvas 的某个区域，如图 4-9 所示。

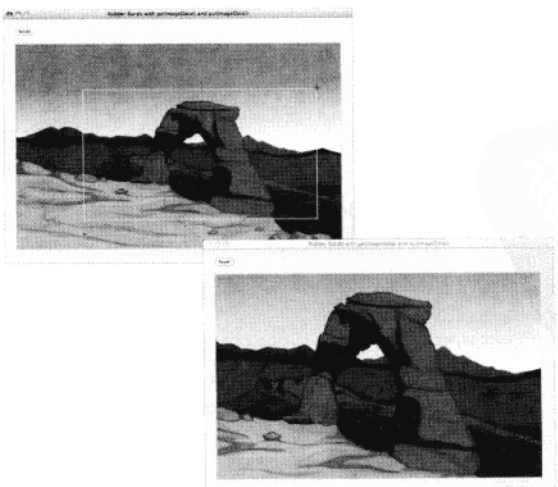


图 4-9 用橡皮筋式选取框来框选 canvas 之中的某个区域

在图 4-9 之中，用户使用橡皮筋式选取框选中了 canvas 里的某个区域，然后应用程序把选中的区域拉伸至与 canvas 同等长宽，并将其绘制出来。

每当用户拖动鼠标时，应用程序都会计算出选取框的大小，捕捉在该范围框之内的像素，然后再将橡皮筋式选取框绘制出来。下一次用户再拖动鼠标时，应用程序就会把上次拖动时捕捉的图像恢复到屏幕上，这样也就擦除了橡皮筋式选取框，从而可以开始新的框选操作了。

图 4-9 以及程序清单 4-9 之中所述的范例应用程序，其实并没有操作图像的像素，它只不过是在用户拖动橡皮筋式选取框时，对图像像素进行了简单的捕捉和恢复操作而已。

请注意，在 rubberbandEnd() 方法中，应用程序的代码调用了接受 9 个参数的 drawImage() 方法，对图像进行缩放，并将用户选择的那部分内容绘制出来。

还要请大家注意一个问题，那就是用户很容易会拖拽出一个宽度或高度为 0 的橡皮筋式选取框来。根据 Canvas 规范，如果指定给 getImageData() 方法的宽度或高度值是 0，那么该方法必须抛出异常。我们可以在程序清单 4-9 之中看到应用程序如何处理这种情况：此时程序并不会捕捉当前橡皮筋式选取框内的图像数据，直到用户下次移动鼠标时再去判断。这样的话，应用程序也就不用擦除上一次的选取框了，将它留在屏幕上就好。

鉴于 getImageData() 方法可能会抛出异常，所以 rubberbandStretch() 方法需要确保在调用 getImageData() 方法时不会发生宽度或高度为 0 的情况。实际上，rubberbandStretch() 方法会依据绘图环境对象的线宽属性进行判断，它只有在选取框足够大，以致能够容纳橡皮筋式框选线的情况下，才会将上一次捕捉的图像数据恢复到屏幕，并重新绘制当前的矩形选框。

程序清单 4-9　利用 getImageData() 与 putImageData() 方法来实现橡皮筋式选取框

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    resetButton = document.getElementById('resetButton'),

    image = new Image(),
    imageData,

    mousedown = {},
    rubberbandRectangle = {},
    dragging = false;

// Functions.......................................................

function windowToCanvas(canvas, x, y) {
    var canvasRectangle = canvas.getBoundingClientRect();
    return { x:x - canvasRectangle.left,
             y:y - canvasRectangle.top };
}

function captureRubberbandPixels() {
    imageData = context.getImageData(rubberbandRectangle.left,
                                     rubberbandRectangle.top,
                                     rubberbandRectangle.width,
                                     rubberbandRectangle.height);
}

function restoreRubberbandPixels() {
    context.putImageData(imageData, rubberbandRectangle.left,
```

```
        rubberbandRectangle.top);
}

function drawRubberband() {
    context.strokeRect(rubberbandRectangle.left + context.lineWidth,
                       rubberbandRectangle.top + context.lineWidth,
                       rubberbandRectangle.width - 2 * context.lineWidth,
                       rubberbandRectangle.height - 2 * context.lineWidth);
}

function setRubberbandRectangle(x, y) {
    rubberbandRectangle.left = Math.min(x, mousedown.x);
    rubberbandRectangle.top = Math.min(y, mousedown.y);
    rubberbandRectangle.width = Math.abs(x - mousedown.x),
    rubberbandRectangle.height = Math.abs(y - mousedown.y);
}

function updateRubberband() {
    captureRubberbandPixels();
    drawRubberband();
}

function rubberbandStart(x, y) {
    mousedown.x = x;
    mousedown.y = y;

    rubberbandRectangle.left = mousedown.x;
    rubberbandRectangle.top = mousedown.y;

    dragging = true;
}

function rubberbandStretch(x, y) {
    if (rubberbandRectangle.width > 2 * context.lineWidth &&
        rubberbandRectangle.height > 2 * context.lineWidth) {
        if (imageData !== undefined) {
            restoreRubberbandPixels();
        }
    }

    setRubberbandRectangle(x, y);

    if (rubberbandRectangle.width > 2 * context.lineWidth &&
        rubberbandRectangle.height > 2 * context.lineWidth) {
        updateRubberband();
    }
}

function rubberbandEnd() {
// Draw and scale image to the onscreen canvas.
    context.drawImage(canvas,
                      rubberbandRectangle.left + context.lineWidth * 2,
                      rubberbandRectangle.top + context.lineWidth * 2,
                      rubberbandRectangle.width - 4 * context.lineWidth,
                      rubberbandRectangle.height - 4 * context.lineWidth,
                      0, 0, canvas.width, canvas.height);
    dragging = false;
    imageData = undefined;
}

// Event handlers.....................................................
```

```
canvas.onmousedown = function (e) {
    var loc = windowToCanvas(canvas, e.clientX, e.clientY);
    e.preventDefault();
    rubberbandStart(loc.x, loc.y);
};

canvas.onmousemove = function (e) {
    var loc;

    if (dragging) {
        loc = windowToCanvas(canvas, e.clientX, e.clientY);
        rubberbandStretch(loc.x, loc.y);
    }
};

canvas.onmouseup = function (e) {
    rubberbandEnd();
};

// Initialization.....................................................

image.src = 'arch.png';
image.onload = function () {
    context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

resetButton.onclick = function (e) {
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

context.strokeStyle = 'navy';
context.lineWidth = 1.0;
```

小技巧：实现橡皮筋式选取框的各种方式

在本书 1.8.1 小节中，我们看到应用程序使一个带有可视边框的空白 div 元素浮动在 canvas 之上，以此做出橡皮筋式选取框的效果。当用户拖动鼠标时，应用程序会调整该 div 元素的大小，这样的话，就实现了一个可以伸缩的橡皮筋式选取框。

本小节所讲的这种实现方式，比起刚才那种略微复杂些，效率也稍差。不过，利用 canvas 自身机制来实现橡皮筋式选取框，可以让我们在实现的时候加入其他的效果，例如 4.11 节就会演示如何修改所选像素的透明度。

4.5.1.1 ImageData 对象

在 4.5.1 小节中，应用程序在实现橡皮筋式选取框的时候，调用 getImageData() 方法获取了一个指向 ImageData 对象的引用。稍后，该程序会将此对象传给 putImageData() 方法，以擦除上次所绘的框选线。

getImageData() 方法所返回的 ImageData 对象包含下列三个属性：

- width：以设备像素（device pixel）为单位的图像数据宽度。
- height：以设备像素为单位的图像数据高度。
- data：包含各个设备像素数值的数组。

width 与 height 均是只读的无符号长整数。data 属性所含的每个数组元素，均表示图像数据中的相应像素值，每个值中所含的颜色分量，都是含有 8 个二进制位的整数。在 4.5.2 小节中，我们将详细讲解 ImageData 对象。

提示：设备像素与 CSS 像素的对比

为了使所绘图像更加逼真，浏览器可能会用多个设备像素来表现一个 CSS 像素。比如，有一个边长为 200 像素的 canvas，那么它总共就含有 40000 个 CSS 像素。然而，若是浏览器在横竖两个方向上都用两个设备像素来表示每个 CSS 像素的话，那么总共就会有 160000（400× 400）个设备像素。可以通过 ImageData 对象的 width 与 height 属性来查看 canvas 所含设备像素的数量。

4.5.1.2 结合使用 putImageData 方法与脏矩形技术对图像数据进行局部渲染

每当用户拖动鼠标时，4.5.1 小节中所述应用程序就会调用 putImageData() 方法来擦除上一次绘制的选取框，然后，在绘制当前的橡皮筋式选取框之前，先行调用 getImageData() 方法，把由鼠标当前位置所确定的矩形框内的图像存储起来。

这种实现方式是可行的，不过，它有个缺点：getImageData() 方法的运行速度比较慢，每次用户移动鼠标时，应用程序都要调用该方法。在大多数情况下，canvas 的运行速度都很快，持续调用 getImageData() 对它没什么影响，不过，如果应用程序运行在诸如手机或平板电脑这样的低配置设备上，那么对 getImageData() 方法的调用就有可能成为性能瓶颈。

有种更有效率的实现方式：只在每次检测到鼠标按下事件时，才调用一次 getImageData() 方法，用以捕捉 canvas 内的全部像素。然后，每次在处理鼠标移动事件时，则调用 putImageData() 方法，仅仅将框选矩形所占据的那一小部分图像数据复制到 canvas。这种实现方式显著地降低了 getImageData() 方法的调用次数。

这种更为高效的实现方式，使用了 4 个可选参数来调用 putImageData() 方法，这 4 个参数所确定的“脏矩形”（dirty rectangle），指的是浏览器将要复制到 canvas 之中的那部分图像数据所占据的区域。该方法的调用格式是：putImageData(HTMLImage, dx, dy, dirtyX, dirtyY, dirtyWidth, dirtyHeight) 。

图 4-10 演示了这个 7 参数版本的 putImageData() 方法是如何将图像的某一部分数据复制到 canvas 的。

dx 与 dy 参数分别表示 putImageData() 方法所绘制的图像距离 canvas 左上角的 X、Y 偏移量。浏览器将会把所绘图像数据的左上角置于该偏移位置，并根据此偏移量来计算图形数据中的脏矩形所对应的 canvas 坐标。

putImageData() 方法的最后 4 个参数代表以设备像素为单位的脏矩形。当浏览器将脏矩形复制到 canvas 时，它会将设备像素转换为 CSS 像素，如图 4-10 所示。

我们可以修改 4.5.1 小节中的应用程序，让它在每次处理鼠标按下事件时，将所有 canvas 的像素都捕捉到 ImageData 对象中，然后在稍后用户拖动鼠标时，则仅仅将橡皮筋式选取框所占据的那一区域从 ImageData 复制到 canvas 之上。程序清单 4-10 列出了对 captureRubberbandPixels() 及 restoreRubberbandPixels() 所做的改动。

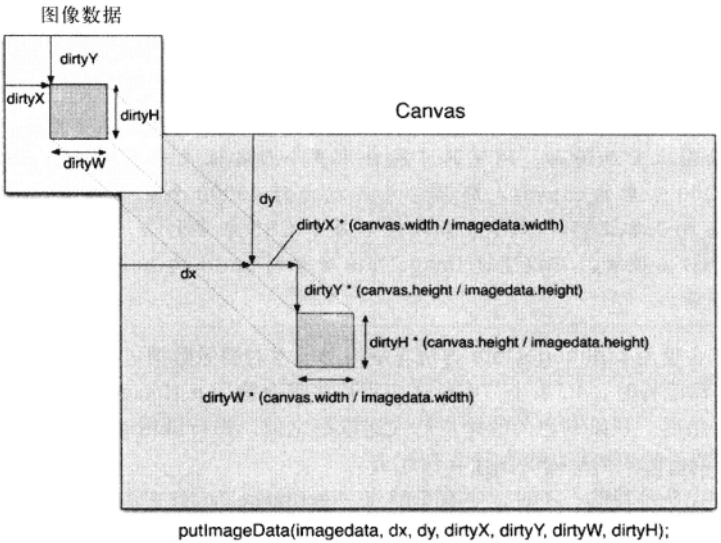


图 4-10 用 putImageData() 方法来复制脏矩形中的图像

程序清单 4-10 捕捉 canvas 之中的像素

```
function captureRubberbandPixels() {
    // Capture the entire canvas
    imageData = context.getImageData(0, 0, canvas.width, canvas.height);
}

function restoreRubberbandPixels() {
    var deviceWidthOverCSSPixels = imageData.width / canvas.width,
        deviceHeightOverCSSPixels = imageData.height / canvas.height;

    // Put data for the rubberband rectangle, scaled to device pixels

    context.putImageData(imageData, 0, 0,
        rubberbandRectangle.left,
        rubberbandRectangle.top,
        rubberbandRectangle.width * deviceWidthOverCSSPixels,
        rubberbandRectangle.height * deviceHeightOverCSSPixels);
}
```

表 4-2 总结了 getImageData() 与 putImageData() 方法的用法。

小技巧：putImageData() 方法不受全局设置的影响

在使用 putImageData() 方法向 canvas 之中绘制图像数据时，诸如 globalAlpha 与 globalCompositeOperation 这样的全局 canvas 属性值，不会影响到所绘的图像。浏览器也不会在绘制时运用图像合成、透明混合[⊖]或阴影等效果。 drawImage() 方法与之相反，它会受到上述所有全局属性的影响。

⊖ alpha blending，指的是将两个透明色混合成一个颜色的操作，详情参见：http://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending。——译者注

表 4-2 CanvasRenderingContext2D 对象中用于操作图像的方法

方法	描述
getImageData(in double sx, in double sy, in double sw, in double sh)	返回一个 ImageData 对象，该对象所含的数组具有 4×w×h 个整数值，w 和 h 表示以设备像素为单位的图像宽度与高度。可以通过 width 与 height 属性来访问该 ImageData 对象的宽度与高度。 ImageData 对象所含的 data 数组中，每 4 个整数值代表 1 个像素，这些整数分别表示红、绿、蓝颜色分量，以及代表透明度的 alpha 值。 请注意，该方法所返回 ImageData 对象的宽度并不总是等于传递给它的宽度参数值。这是因为前者是以设备像素为单位的，而后者则是以 CSS 像素为单位的
putImageData(in ImageData imagedata, in double dx, in double dy, in optional double dirtyX, in double dirtyY, in double dirtyWidth, in double dirtyHeight)	将图像数据绘制在 canvas 的 (dx, dy) 坐标处，该坐标是以 CSS 像素为单位的。后面 4 个参数所指定的脏矩形表示浏览器将会把这个矩形范围内的图像数据复制到屏幕 canvas。指定该矩形时需以设备像素为单位

小技巧：putImageData() 方法的可选参数

putImageData() 方法的后 4 个参数表示图像数据中的一块脏矩形区域。设置这个脏矩形是表明它所对应的那块 canvas 图像已经被修改了，稍后我们需要将该矩形内的图像重新复制到 canvas 之中的某个位置上。

这 4 个参数是可选的，如果不指定它们，则会使用默认值。下面这个列表总结了这些参数的意义，并列出了各自的默认值：

- 所绘脏矩形距离整幅图像数据左上角的水平偏移量，该值以设备像素为单位，默认值是 0。
- 所绘脏矩形距离整幅图像数据左上角的垂直偏移量，该值以设备像素为单位，默认值是 0。
- 以设备像素为单位的脏矩形宽度，默认值为整幅图像的宽度。
- 以设备像素为单位的脏矩形高度，默认值为整幅图像的高度。

警告：putImageData() 方法的参数同时使用了设备像素及 CSS 像素这两种计量单位

当调用接受 7 个参数的 putImageData() 方法时，必须同时指定所绘图像距离 canvas 左上角的偏移量（通过第 2 个与第 3 个参数传入），以及图像数据中需要绘制的那块脏矩形区域（通过后 4 个参数传入）。

在指定 canvas 偏移量时，需要以 CSS 像素为单位，然而在指定图像数据中的脏矩形区域时，则需要以设备像素为单位。如果不慎使用了同一种计量单位来指定这两组参数，那么 putImageData() 方法也许就达不到你所期望的运行效果了。

4.5.2 修改图像数据

在讲完如何使用 getImageData() 与 putImageData() 方法来存储并获取图像数据之后，我们现在来看看如何修改这些数据。

图 4-11 所示的橡皮筋式选取框程序，修改了选框内所有像素的透明度。

为了临时提升橡皮筋式选取框内的像素透明度，图 4-11 所示应用程序使用了两个与 canvas 大小相同的 ImageData 对象。

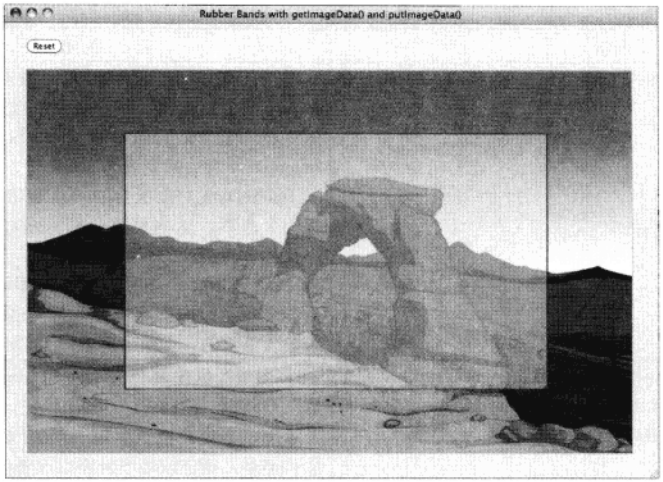


图 4-11　带有透明度特效的橡皮筋式选取框

其中一个 ImageData 对象用来保存用户上次按下鼠标时的 canvas 快照，另一个 ImageData 对象则含有对这份快照的拷贝，不同的是，其中每个像素的透明度都是原有快照的两倍，效果如图 4-12 所示。

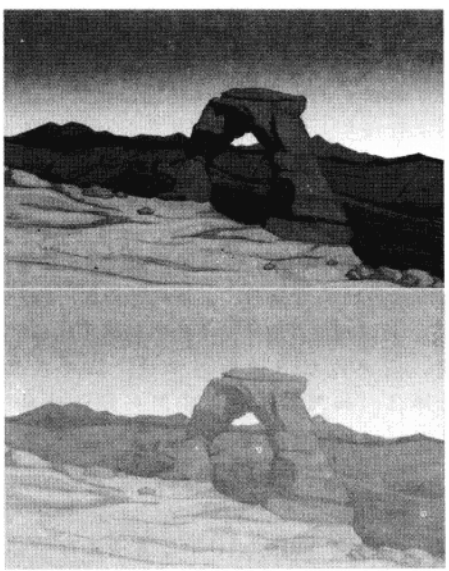


图 4-12　应用程序所使用的两个 ImageData 对象：顶部的用于绘制背景，底部的用于绘制橡皮筋式选取框

当用户拖动鼠标时，应用程序做了如下三件事：

（1）将整个背景快照（图 4-12 顶部）的图像恢复到 canvas 之中，以擦除上一次所绘的橡皮筋式选取框。

（2）从高透明度的快照（图 4-12 底部）中将当前选框内的图像复制到屏幕 canvas 之中。

（3）对橡皮筋式选取框进行描边。

4.5.2.1 使用 createImageData() 方法创建 ImageData 对象

图 4-11 所示应用程序在启动时，会调用 createImageData() 方法来创建 ImageData 对象。稍后当用户按下鼠标时，程序则会调用 captureCanvasPixels() 方法来初始化这个 ImageData 对象，该方法的代码列在了程序清单 4-11 之中。

当检测到了鼠标按下事件时，应用程序会调用 getImageData() 方法来抓取 canvas 之中的所有像素，然后再调用 copyCanvasPixels() 方法将这些像素复制到前面已经分配好的那个 imageDataCopy 对象中，程序在复制时会把每个像素的透明度加倍。于是，在每次触发完鼠标按下事件之后，应用程序都会将 canvas 图像数据以及另外一份透明度更高的快照保存起来。

程序清单 4-11 创建并初始化 ImageData 对象

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    image = new Image(),
    imageData,
    imageDataCopy = context.createImageData(canvas.width, canvas.height),
    ...

// Functions.........................................................

...
function copyCanvasPixels() {
    // Copy imageData into imageDataCopy, doubling the transparency
    // of each pixel in the array.
}

function captureCanvasPixels() {
    imageData = context.getImageData(0, 0, canvas.width, canvas.height);
    copyCanvasPixels();
}
...

function rubberbandStart(x, y) {
    ...
    captureCanvasPixels();
}

// Event handlers....................................................

canvas.onmousedown = function(e) {
    var loc = windowToCanvas(canvas, e.clientX, e.clientY);
    e.preventDefault();
    rubberbandStart(loc.x, loc.y);
};
...
```

程序清单 4-11 之中的 copyCanvasPixels() 方法，在复制 canvas 像素的同时对其进行了修改，使得创建出来的 ImageData 具有更高的透明度。下一小节将会详述此方法。

4.5.2.1.1 ImageData 对象中的数组

ImageData 对象中的 data 属性指向一个包含 8 位二进制整数的数组，这些整数的值位于 0 ～ 255 之间，分别表示一个像素的红、绿、蓝及透明度分量，如图 4-13 所示。

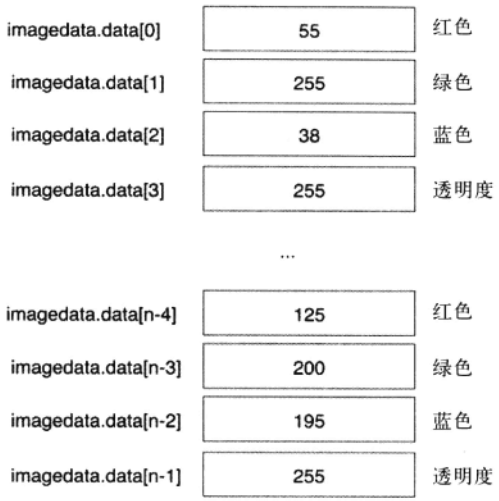


图 4-13　长度为 n 的图像数据数组

图 4-11 所示应用程序会将整个 canvas 之中的图像数据复制到另外一个 ImageData 对象中，复制时所调用的函数如下：

```
function copyCanvasPixels() {
    var i=0;

    // Copy red, green, and blue components of the first pixel

    for (i=0; i < 3; i++) {
        imageDataCopy.data[i] = imageData.data[i];
    }

    // Starting with the alpha component of the first pixel,
    // copy imageData, but make the copy more transparent

    for (i=3; i < imageData.data.length - 4; i+=4) {
        imageDataCopy.data[i] = imageData.data[i] / 2;  //Alpha
        imageDataCopy.data[i+1] = imageData.data[i+1];  //Red
        imageDataCopy.data[i+2] = imageData.data[i+2];  //Green
        imageDataCopy.data[i+3] = imageData.data[i+3];  //Blue
    }
}
```

上述代码把每个像素的红、绿、蓝分量及透明度都复制到 imageDataCopy 对象中，并将透明度加倍。这段代码遍历整个数组，在每次遍历时，都会处理 4 个整数，并在循环体内将数组中的这 4 个整数值复制到 imageDataCopy 对象的对应数组元素中，在复制的时候，会把 alpha 值减为原来的一半。

程序清单 4-12 列出了图 4-11 所示应用程序的全部 JavaScript 代码。

程序清单 4-12　能够修改图像数据的橡皮筋式选取框

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
```

```
    resetButton = document.getElementById('resetButton'),

    image = new Image(),
    imageData,
    imageDataCopy = context.createImageData(canvas.width, canvas.height),

    mousedown = {},
    rubberbandRectangle = {},
    dragging = false;

// Functions.....................................................
function windowToCanvas(canvas, x, y) {
    var canvasRectangle = canvas.getBoundingClientRect();
    return { x : x - canvasRectangle.left,
             y : y - canvasRectangle.top};
}
function copyCanvasPixels() {
    var i = 0;

    // Copy red, green, and blue components of the first pixel
    for ( i = 0; i < 3; i++) {
        imageDataCopy.data[i] = imageData.data[i];
    }

    // Starting with the alpha component of the first pixel,
    // copy imageData, and make the copy more transparent
    for ( i = 3; i < imageData.data.length - 4; i += 4) {
        imageDataCopy.data[i] = imageData.data[i] / 2;       // Alpha
        imageDataCopy.data[i + 1] = imageData.data[i + 1];  // Red
        imageDataCopy.data[i + 2] = imageData.data[i + 2];  // Green
        imageDataCopy.data[i + 3] = imageData.data[i + 3];  // Blue
    }
}
function captureCanvasPixels() {
    imageData = context.getImageData(0, 0, canvas.width, canvas.height);
    copyCanvasPixels();
}
function restoreRubberbandPixels() {
    var deviceWidthOverCSSPixels = imageData.width / canvas.width,
        deviceHeightOverCSSPixels = imageData.height / canvas.height;

    // Restore the canvas to what it looked like when the mouse went down

    context.putImageData(imageData, 0, 0);

    // Put the more transparent image data into the rubberband rectangle

    context.putImageData(imageDataCopy, 0, 0,

        rubberbandRectangle.left + context.lineWidth,
        rubberbandRectangle.top + context.lineWidth,

        (rubberbandRectangle.width - 2 * context.lineWidth) * deviceWidthOverCSSPixels,
        (rubberbandRectangle.height - 2 * context.lineWidth) * deviceHeightOverCSSPixels);
}
function setRubberbandRectangle(x, y) {
    rubberbandRectangle.left = Math.min(x, mousedown.x);
    rubberbandRectangle.top = Math.min(y, mousedown.y);
    rubberbandRectangle.width = Math.abs(x - mousedown.x),
    rubberbandRectangle.height = Math.abs(y - mousedown.y);
}
```

```
function drawRubberband() {
    context.strokeRect( rubberbandRectangle.left + context.lineWidth,
                        rubberbandRectangle.top + context.lineWidth,
                        rubberbandRectangle.width - 2 * context.lineWidth,
                        rubberbandRectangle.height - 2 * context.lineWidth);
}
function rubberbandStart(x, y) {
    mousedown.x = x;
    mousedown.y = y;

    rubberbandRectangle.left = mousedown.x;
    rubberbandRectangle.top = mousedown.y;
    rubberbandRectangle.width = 0;
    rubberbandRectangle.height = 0;

    dragging = true;

    captureCanvasPixels();
}
function rubberbandStretch(x, y) {
    if (rubberbandRectangle.width > 2 * context.lineWidth &&
        rubberbandRectangle.height > 2 * context.lineWidth) {
        if (imageData !== undefined) {
            restoreRubberbandPixels();
        }
    }

    setRubberbandRectangle(x, y);

    if (rubberbandRectangle.width > 2 * context.lineWidth &&
        rubberbandRectangle.height > 2 * context.lineWidth) {
        drawRubberband();
    }
}
function rubberbandEnd() {
    context.putImageData(imageData, 0, 0);

    // Draw the canvas back into itself, scaling along the way
    context.drawImage(canvas,
                      rubberbandRectangle.left + context.lineWidth * 2,
                      rubberbandRectangle.top + context.lineWidth * 2,
                      rubberbandRectangle.width - 4 * context.lineWidth,
                      rubberbandRectangle.height - 4 * context.lineWidth,
                      0, 0, canvas.width, canvas.height);
    dragging = false;
    imageData = undefined;
}

// Event handlers.....................................................

canvas.onmousedown = function(e) {
    var loc = windowToCanvas(canvas, e.clientX, e.clientY);
    e.preventDefault();
    rubberbandStart(loc.x, loc.y);
};
canvas.onmousemove = function(e) {
    var loc;
    if (dragging) {
        loc = windowToCanvas(canvas, e.clientX, e.clientY);
        rubberbandStretch(loc.x, loc.y);
    }
```

```
};
canvas.onmouseup = function(e) {
    rubberbandEnd();
};

// Initialization.....................................................

image.src = 'arch.png';
image.onload = function () {
    context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

resetButton.onclick = function(e) {
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.drawImage(image, 0, 0, canvas.width, canvas.height);
};

context.strokeStyle = 'navy';
context.lineWidth = 1.0;
```

提示：Canvas 规范已更新，ImageData 对象将使用 ArrayBuffer 来存放数据

读者在本小节中已经学到，如何访问以 8 位二进制整数来存放其红、绿、蓝及 Alpha 分量的图像像素数组。在 getImageData() 方法所返回的 ImageData 对象中，data 属性就是指向该数组的引用。

本书付印时，W3C 组织在规范书中将该引用的类型修改为 TypedArray。此类型是一个用于反映数组内容的视图，它的设计用意是：同样一份数据缓冲区（data buffer）中的内容可以用不同的格式来读取[⊖]。

从技术上来说，ImageData 对象所含的像素数据必须以 ArrayBuffer 的形式来存放，这样它才能被 Uint8ClampedArray 数组视图所包装[⊜]。关于 TypedArray 的更多信息，请参阅：https://developer.mozilla.org/en/JavaScript_typed_arrays。

在实际应用中，Canvas 规范书的这条修订并不会迫使你重写现有代码，因为我们仍然可以将 ImageData 对象中的像素数据当做一个普通的数组来用。只不过，从底层实现的角度来看，这种以 ArrayBuffer 形式存放的数组用起来更为灵活，也更加高效。

4.5.2.2 图像数据的遍历方式

假设有如下代码：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    imagedata = context.getImageData(0,0,canvas.width,canvas.height),
    data = imagedata.data,
    length = imagedata.data.length,
    width = imagedata.width,
    index = 0,
    value;
```

那么，我们就可以使用下列方式来遍历 ImageData 之中的图像数据。

遍历每个像素：

```
for (var index=0; index < length; ++i) {
```

⊖ TypedArray的官方文档位于：http://www.khronos.org/registry/typedarray/specs/latest/。——译者注

⊜ Uint8ClampedArray是众多TypedArray数组视图格式中的一种，此外还有Int32Array、Int16Array等格式。——译者注

```
    value = data[index];
}
```

反向遍历每个像素：

```
index = length-1;
while (index >= 0) {
    value = data[index];
    index--;
}
```

只处理 alpha 值，不修改红、绿、蓝分量：

```
for(index=3; index < length-4; index+=4) {
    data[index] = ...; // Alpha
}
```

只处理红、绿、蓝分量，不修改 alpha 值：

```
for(index=0; index < length-4; index+=4) {
    data[index] = ...;      // Red
    data[index+1] = ...;    // Green
    data[index+2] = ...;    // Blue
}
```

本书 4.9 节将详细讲述图像数据的遍历方式及其性能。

4.5.2.3 图像滤镜

在学会了如何操作图像中的单个像素之后，我们来讲讲图像滤镜（image filtering）的实现。图 4-14 展示了两种滤镜，分别是负片滤镜（negative filter）与黑白滤镜（black-and-white filter），程序清单 4-13 与程序清单 4-14 列出了它们各自的代码。

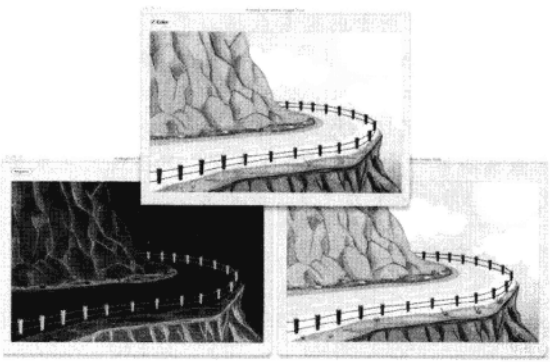


图 4-14　顶部：原始图像；底部：运用了负片滤镜与黑白滤镜后的图像

负片滤镜与黑白滤镜都会对图像数据进行遍历，每个循环会处理 4 个整数值，所以说，每次循环结束后，数组下标总是会指向某个像素的红色分量值。在循环体中，滤镜的代码会修改每个像素的红、绿、蓝分量值。滤镜算法不会改变像素的 alpha 值。

负片滤镜会从 255 之中减去每个像素的红、绿、蓝分量值，再将差值设置回去，这样也就等于“反转”了该像素的颜色。

黑白滤镜会计算出每个像素红、绿、蓝分量值的平均值，然后将三个分量都设置为这一均值，于是，就把图像由彩色变成了黑白。

程序清单 4-13　负片滤镜

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    negativeButton = document.getElementById('negativeButton');

negativeButton.onclick = function() {
    var imagedata =
            context.getImageData(0, 0, canvas.width, canvas.height),
        data = imagedata.data;

    for(i=0; i <= data.length - 4; i+=4) {
        data[i] = 255 - data[i]
        data[i + 1] = 255 - data[i + 1];
        data[i + 2] = 255 - data[i + 2];
    }
    context.putImageData(imagedata, 0, 0);
};

image.src = 'curved-road.png';
image.onload = function() {
    context.drawImage(  image, 0, 0, image.width, image.height, 0, 0,
                        context.canvas.width, context.canvas.height);
};
```

程序清单 4-14　黑白滤镜

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    drawInColorToggleCheckbox = document.getElementById('drawInColorToggleCheckbox');

function drawInBlackAndWhite() {
    var data = undefined,
        i = 0;

    imagedata = context.getImageData(0, 0, canvas.width, canvas.height);
    data = imagedata.data;

    for(i=0; i < data.length - 4; i+=4) {
        average = (data[i] + data[i+1] + data[i+2]) / 3;
        data[i] = average;
        data[i+1] = average;
        data[i+2] = average;
    }
    context.putImageData(imagedata, 0, 0);
}
function drawInColor() {
    context.drawImage(image, 0, 0,
        image.width, image.height, 0, 0,
        context.canvas.width, context.canvas.height);
}

colorToggleCheckbox.onclick = function() {
    if (colorToggleCheckbox.checked) {
        drawInColor();
    }
    else {
        drawInBlackAndWhite();
    }
```

```
};

image.src = 'curved-road.png';
image.onload = function() {
    drawInColor();
};
```

4.5.2.4 再谈设备像素与 CSS 像素的区别

某些滤镜，比如图 4-15 所示的浮雕滤镜（embossing filter），在计算滤镜效果时需要获取图像数据的宽度。举例来说，如果滤镜需要根据一个简单的等式来计算某个像素值，而该等式要用到当前像素右方以及下一行的对应像素值，那么此时必须知道图像的宽度[⊖]，才能计算出下一行的那个像素在数组中的位置。

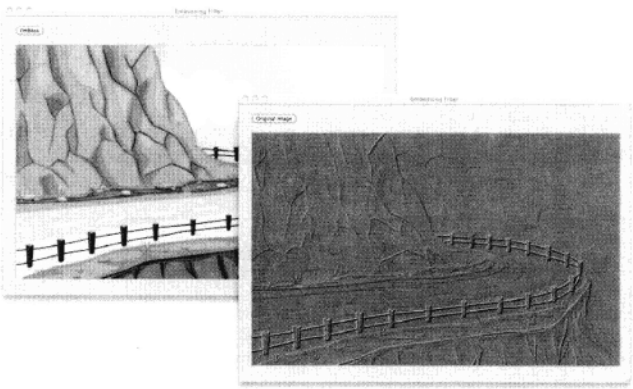


图 4-15 浮雕滤镜

图 4-15 所示应用程序的核心代码如下：

```
function emboss() {
    var imagedata, data, length, width;

    imagedata = context.getImageData(0, 0, canvas.width, canvas.height);
    data = imagedata.data;
    width = imagedata.width;
    length = data.length;

    for (i=0; i < length; i++) {
        if ((i+1) % 4 !== 0) {

            // Use imagedata.width instead of the width you pass
            // to getImageData(). Most of the time the two values
            // are the same, but if the browser uses multiple device
            // pixels per CSS pixel, only imagedata.width represents
            // the true width of the image data.
            data[i] =   255 / 2                  // Average value
                      + 2 * data[i]              // Current pixel
                      - data[i + 4]              // Next pixel
                      - data[i + width * 4];     // Pixel underneath
```

⊖ 本小节的标题是想提醒大家，在与图像数据有关的度量值中，应该使用设备像素而非CSS像素。作者在emboss()代码的注释中也强调了这一点。——译者注

```
        }
    }
    context.putImageData(imagedata, 0, 0);
}
```

上述代码会将所有像素都染成泥灰色，并且使用一种名为“边缘检测”[⊖]的技术使得位于颜色边界处的像素灰度更浓。这里所说的颜色边界是指像素颜色值发生突变的地方。实现边缘检测技术所用的算法需要计算当前像素值，及其右方与下方像素的值。

不过上述代码并未考虑边界条件（boundary condition）。例如，所有位于最后一行的像素，它的下方都不会再有其他像素了，而每行最右方的像素，它的右方也不会有别的像素出现。程序清单 4-15 列出了图 4-15 所示应用程序的 emboss() 函数，这段 JavaScript 代码就考虑到了刚才说的边界条件。

程序清单 4-15　浮雕滤镜

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    embossButton = document.getElementById('embossButton'),
    embossed = false;

// Functions. ..........................................................

function emboss() {
    var imagedata, data, length, width, index=3;

    imagedata = context.getImageData(0, 0, canvas.width, canvas.height);
    data = imagedata.data;
    width = imagedata.width;
    length = data.length;

    for (i=0; i < length; i++) { // Loop through every pixel

        // If we won't overrun the bounds of the array

        if (i <= length-width*4) {

            // If it's not an alpha

            if ((i+1) % 4 !== 0) {

                // If it's the last pixel in the row, there is no pixel
                // to the right, so copy previous pixel's values.

                if ((i+4) % (width*4) == 0) {
                    data[i] = data[i-4];
                    data[i+1] = data[i-3];
                    data[i+2] = data[i-2];
                    data[i+3] = data[i-1];
                    i+=4;
                }
                else { // Not the last pixel in the row
                    data[i] = 255/2                 // Average value
                              + 2*data[i]           // Current pixel
                              - data[i+4]           // Next pixel
```

⊖ 英文为Edge Detection，是一种基础的图像处理技术，其目的是标识数字图像中亮度变化明显的点。详情参见：http://zh.wikipedia.org/zh-cn/边缘检测。——译者注

```
                                - data[i+width*4];  // Pixel underneath
                }
            }
        }
        else { // Last row, no pixels underneath, so copy pixel above
            if ((i+1) % 4 !== 0) {
                data[i] = data[i-width*4];
            }
        }
    }
    context.putImageData(imagedata, 0, 0);
}

function drawOriginalImage() {
    context.drawImage(image, 0, 0,
                      image.width, image.height,
                      0, 0, canvas.width, canvas.height);
}

embossButton.onclick = function() {
    if (embossed) {
        embossButton.value = 'Emboss';
        drawOriginalImage();
        embossed = false;
    }
    else {
        embossButton.value = 'Original image';
        emboss();
        embossed = true;
    }
};

// Initialization.....................................................

image.src = 'curved-road.png';
image.onload = function() {
    drawOriginalImage();
};
```

本章讲的图像操作所采用的都是小图像，因而不会引发性能问题。然而，如果在相对大一些的图像上运用复杂的算法，那么当然不希望看到浏览器由于执行运算而陷入无响应状态。接下来我们就谈谈如何应对这种状况。

4.5.2.5 用工作线程处理图像

我们在处理图像时很有可能遭遇性能瓶颈，比如在一个配置不高的手机上处理大幅图像时。如果程序出现了性能问题，那么可以考虑将图像处理的任务交由工作线程（Web Worker）来做。

浏览器在执行 JavaScript 代码时，使用的是主线程，这意味着某些较为耗时的脚本可能会让应用程序的响应变得很迟钝[⊖]。幸好我们可以在 HTML5 开发中使用工作线程技术将某些代码放在主线程之外执行。图 4-16 所示的应用程序会对图像运用墨镜滤镜（sunglass filter）效果，它把实际的图像处理放在工作线程中执行。程序清单 4-16 列出了该应用程序的代码。

当主线程执行到 sunglassFilter = new Worker('sunglassFilter.js') 这一行语句时，会创建一个工作线程。传递给 Worker 构造器的文件名表示工作线程将要执行的 JavaScript 程序文件。

⊖ 原文为sluggish，其含义就是我们俗称的“应用程序很卡”、“运行不流畅”。——译者注

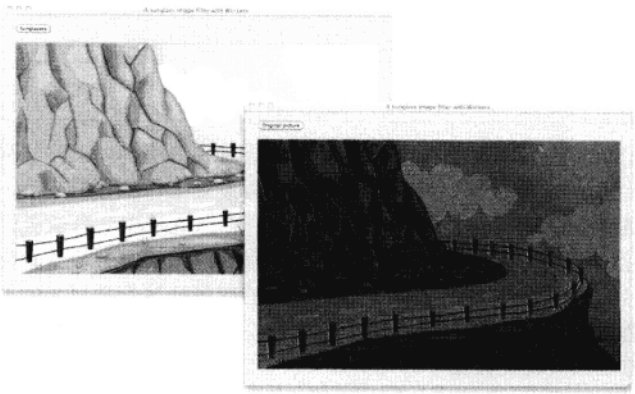


图 4-16　墨镜滤镜

程序清单 4-16　主线程

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    sunglassButton = document.getElementById('sunglassButton'),
    sunglassesOn = false,
    sunglassFilter = new Worker('sunglassFilter.js');

// Functions.........................................................

function putSunglassesOn() {
    sunglassFilter.postMessage(
        context.getImageData(0, 0, canvas.width, canvas.height));

    sunglassFilter.onmessage = function (event) {
        context.putImageData(event.data, 0, 0);
    };
}

function drawOriginalImage() {
    context.drawImage(image, 0, 0,
                      image.width, image.height, 0, 0,
                      canvas.width, canvas.height);
}

// Event handlers....................................................

sunglassButton.onclick = function() {
    if (sunglassesOn) {
        sunglassButton.value = 'Sunglasses';
        drawOriginalImage();
        sunglassesOn = false;
    }
    else {
        sunglassButton.value = 'Original picture';
        putSunglassesOn();
        sunglassesOn = true;
    }
};
```

```
// Initialization.....................................................
image.src = 'curved-road.png';
image.onload = function() {
    drawOriginalImage();
};
```

主线程通过 putSunglassesOn() 方法与工作线程通信。该方法会向工作线程投递一条消息，消息的内容是 canvas 之中的图像数据。然后，该方法还会设置工作线程的 onmessage 属性。当工作线程处理完图像中的像素之后，它会向自己投递一条消息，这时浏览器就会调用工作线程对象的 onmessage() 回调方法。在本例中，此方法会把工作线程处理过的图像数据重新绘制到 canvas 中。

程序清单 4-17 列出了工作线程的代码。它所实现的图像滤镜会把像素的颜色变深，同时增加其对比度。工作线程处理完传递给它的图像数据之后，会把修改后的数据投递给自己，这样主线程就可以通过回调方法来接收此数据了。

程序清单 4-17　sunglassFilter.js 之中的工作线程代码

```
onmessage = function(event) {
    var imagedata = event.data,
        data = imagedata.data,
        length = data.length,
        width = imagedata.width;

    for (i = 0; i < length; ++i) {
        if ((i+1) % 4 != 0) {
            if ((i+4) % (width*4) == 0) {// Last pixel in a row
                data[i] = data[i - 4];
                data[i+1] = data[i-3];
                data[i+2] = data[i-2];
                data[i+3] = data[i-1];
                i += 4;
            }
                else { data[i] = 2 * data[i] - data[i + 4] - 0.5 * data[i + 4];
            }
        }
    }

    postMessage(imagedata);
};
```

刚才我们说过，工作线程的作用就是将耗时的代码放在其中执行，从而使浏览器能够及时响应用户的操作。不过，对于图像处理来说，工作线程还有一个有用的地方：它们可以把图像处理所用的算法封装起来，便于以后复用。实际上，我们稍后就会讲到如何复用程序清单 4-17 之中的代码。

4.6　结合剪辑区域来绘制图像

图 4-17 所示应用程序将图 4-16 之中的那个程序变得更加直观，更加合乎逻辑。

程序清单 4-18 列出了这个运用墨镜效果的滤镜程序所用的代码，它用到了工作线程、图像处理、离屏 canvas、剪辑区域，以及 Canvas 绘图 API 等技术。在宏观层面上，可以将其工作原理总结如下：

```
var sunglassFilter = new Worker('sunglassFilter.js');
...
imagedata = context.getImageData(0, 0, canvas.width, canvas.height);
```

```
sunglassFilter.postMessage(imagedata);

sunglassFilter.onmessage = function(event) {
   offscreenContext.putImageData(event.data, 0, 0);
   drawLenses(leftLensLocation, rightLensLocation);
   drawWire(center);
   drawConnectors(center);
};
...
```

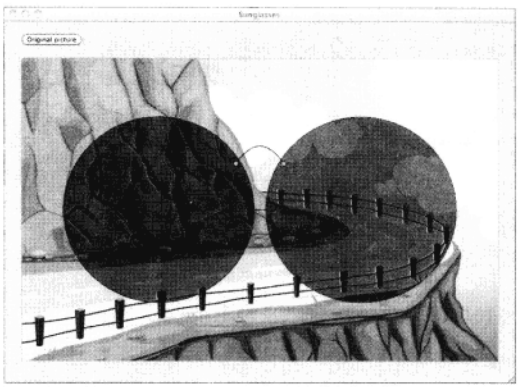


图 4-17　演示墨镜滤镜效果的程序

应用程序会从 canvas 中收集图像信息，然后将其投递给程序清单 4-17 之中的工作线程进行处理。

工作线程会对图像信息进行处理，使每个像素的颜色变深，并增高其对比度。处理完毕后，它会将数据投递出去，这又导致浏览器回调工作线程的 onmessage() 方法。大家在上面这段代码中已经看到，此方法会将修改后的像素复制到离屏 canvas 之中，然后绘制镜片、连接镜片的金属线以及固定金属线的连接头。

drawLenses() 方法将当前绘图环境的状态保存起来，并创建一段新的路径。然后，它向路径中增加两个表示镜片的圆形，并将该路径设置为剪辑区域，再把离屏 canvas 的内容绘制到屏幕 canvas 之中。由于剪辑区域被设定为这两个圆形所在的范围，所以在离屏 canvas 中，只有此范围内的图像才会被绘制到屏幕之上。最后，drawLenses() 方法将绘图环境对象的状态复原，这样的话，剪辑区域又会恢复到调用 context.clip() 方法之前的样子。

程序清单 4-18 之中的 drawWire() 与 drawConnectors() 方法，会利用 Canvas 的绘图 API 分别绘制出镜片连接线以及固定连线所用的连接头。

程序清单 4-18　综合运用了图像处理、离屏 canvas 以及剪辑区域等技术的墨镜效果演示程序

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),

    sunglassButton = document.getElementById('sunglassButton'),
```

```
    sunglassesOn = false,
    sunglassFilter = new Worker('sunglassFilter.js'),

    LENS_RADIUS = canvas.width/5;

// Functions.......................................................

function drawLenses(leftLensLocation, rightLensLocation) {
   context.save();
   context.beginPath();

   context.arc(leftLensLocation.x, leftLensLocation.y,
               LENS_RADIUS, 0, Math.PI*2, false);
   context.stroke();

   moveTo(rightLensLocation.x, rightLensLocation.y);

   context.arc(rightLensLocation.x, rightLensLocation.y,
               LENS_RADIUS, 0, Math.PI*2, false);
   context.stroke();

   context.clip();

   context.drawImage(offscreenCanvas, 0, 0,
                     canvas.width, canvas.height);
   context.restore();
}

function drawWire(center) {
   context.beginPath();
   context.moveTo(center.x - LENS_RADIUS/4, center.y - LENS_RADIUS/2);

   context.quadraticCurveTo(center.x, center.y - LENS_RADIUS+20,
                            center.x + LENS_RADIUS/4,
                            center.y - LENS_RADIUS/2);
   context.stroke();
}

function drawConnectors(center) {
   context.beginPath();

   context.fillStyle = 'silver';
   context.strokeStyle = 'rgba(0,0,0,0.4)';
   context.lineWidth = 2;

   context.arc(center.x - LENS_RADIUS/4, center.y - LENS_RADIUS/2,
               4, 0, Math.PI*2, false);
   context.fill();
   context.stroke();

   context.beginPath();
   context.arc(center.x + LENS_RADIUS/4, center.y - LENS_RADIUS/2,
               4, 0, Math.PI*2, false);
   context.fill();
   context.stroke();
}

function putSunglassesOn() {
   var imagedata,
       center = {
         x: canvas.width/2,
```

```
          y: canvas.height/2
        },
        leftLensLocation = {
          x: center.x - LENS_RADIUS - 10,
          y: center.y
        },
        rightLensLocation = {
          x: center.x + LENS_RADIUS + 10,
          y: center.y
        },

    imagedata = context.getImageData(0, 0,
                                     canvas.width, canvas.height);

    sunglassFilter.postMessage(imagedata);

    sunglassFilter.onmessage = function(event) {
       offscreenContext.putImageData(event.data, 0, 0);
       drawLenses(leftLensLocation, rightLensLocation);
       drawWire(center);
       drawConnectors(center);
    };
}

function drawOriginalImage() {
   context.drawImage(image, 0, 0, image.width, image.height,
                     0, 0, canvas.width, canvas.height);
}

// Event handlers.....................................................

sunglassButton.onclick = function() {
   if (sunglassesOn) {
      sunglassButton.value = 'Sunglasses';
      drawOriginalImage();
      sunglassesOn = false;
   }
   else {
      sunglassButton.value = 'Original picture';
      putSunglassesOn();
      sunglassesOn = true;
   }
};

offscreenCanvas.width = canvas.width;
offscreenCanvas.height = canvas.height;

// Initialization....................................................

image.src = 'curved-road.png';
image.onload = function() {
   drawOriginalImage();
};
```

读者已经学会了如何通过 Canvas 的 API 来处理图像，接下来我们看看怎么将图像制成动画。

4.7 以图像制作动画

在某段时间内持续向一幅图片运用滤镜，就可以实现动画效果了。例如，图 4-18 所示应用程

序就会让图片渐渐淡出。

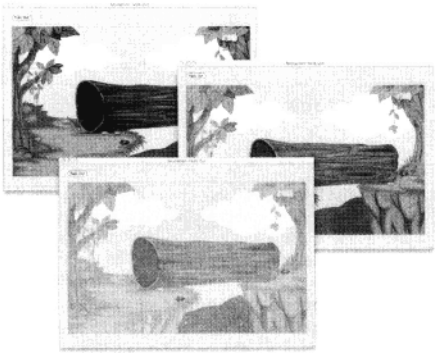


图 4-18 将图像从 canvas 中淡出

该应用程序通过 setInterval() 来持续地降低每个像素的 alpha 值，直到图像从视窗中淡出。

用户点击“Fade Out”按钮后，应用程序即开始播放共 25 帧的动画。动画中每幅画面的播放速率都是 60 帧 / 秒，所以整个动画持续大约 1/2 秒。

淡出动画效果的难点在于，每个像素起初的 alpha 值各不相同，因此，在每一帧中，应用程序都必须根据其初始值来降低每个像素的 alpha 值。为了便于执行这种“动态降低 alpha 值”（variable-alpha-channel reduction）的算法，应用程序把 getImageData() 方法返回的所有原始图像像素数据保存起来，在其后的每一帧动画之中，程序都会根据每个像素的初始值来决定当前这一步要减少的 alpha 值。

程序清单 4-19 列出了图 4-18 所示应用程序的代码。

程序清单 4-19 将图像从 canvas 中淡出

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    fadeButton = document.getElementById('fadeButton'),
    originalImageData = null,
    interval = null;

// Functions.............................................................

function increaseTransparency(imagedata, steps) {
   var alpha, currentAlpha, step, length = imagedata.data.length;

   for (var i=3; i < length; i+=4) { // For every alpha component
      alpha = originalImageData.data[i];

      if (alpha > 0 && imagedata.data[i] > 0) { // Not transparent yet
         currentAlpha = imagedata.data[i];
         step = Math.ceil(alpha/steps);

         if (currentAlpha - step > 0) { // Not too close to the end
            imagedata.data[i] -= step;  // Increase transparency
         }
         else {
            imagedata.data[i] = 0; // End: totally transparent
```

```
            }
        }
    }
}

function fadeOut(context, imagedata, x, y,
                 steps, millisecondsPerStep) {
   var frame = 0,
       length = imagedata.data.length;

   interval = setInterval(function () { // Once every millisecondsPerStep
      frame++;

      if (frame > steps) { // Animation is over
         clearInterval(interval); // End animation
         animationComplete();     // Put picture back in 1s
      }
      else {
        increaseTransparency(imagedata, steps);
         context.putImageData(imagedata, x, y);
      }
   }, millisecondsPerStep);
}

// Animation.......................................................

function animationComplete() {
   setTimeout(function() {
      context.drawImage(image, 0, 0, canvas.width, canvas.height);
   }, 1000);
}

// Event handlers..................................................

fadeButton.onclick = function() {
   fadeOut(context,
      context.getImageData(0, 0, canvas.width, canvas.height),
      0, 0, 20, 1000/60);
};

// Initialization..................................................

image.src = 'log-crossing.png';
image.onload = function() {
   context.drawImage(image, 0, 0, canvas.width, canvas.height);
   originalImageData = context.getImageData(0, 0,
                           canvas.width, canvas.height);
};
```

小技巧：有更为简便的办法可实现图像淡出效果

图 4-18 之中的应用程序是通过修改每个像素的 alpha 值来达到图像淡出效果的。像往常一样，还有很多种方式也可以在 canvas 之中做出同样的效果来。例如，可以在绘制每帧动画之前，先修改绘图环境对象的 globalAlpha 值，然后再绘制图像，这样也能实现淡出效果。

用离屏 canvas 制作动画

图 4-18 所示应用程序通过持续地增加每个像素的透明度来实现图像的淡出效果。图像之

中每个像素的初始透明度可能互不相同，所以，当程序将图像绘制到 canvas 之后，它就会调用 getImageData() 方法捕获图像中所有像素的值。在稍后绘制动画的每一帧时，应用程序都会根据存放在图像数据中的像素初始透明度（以 alpha 值的形式表示），来算出当前这一帧应该把每个像素的 alpha 值分别降低多少。

在制作图像的淡入效果时，也可以使用同样的算法。先将图像的像素值保存到一份快照中，然后根据像素的初始 alpha 值，计算出在动画的每一帧中各个像素的 alpha 值增量。但是，在播放淡入动画时，图像一开始是不会显示出来的，所以不能直接从屏幕 canvas 之中捕获其像素值。

为了在显示图像之前能够捕获其像素，图 4-19 所示应用程序先把图像绘制到离屏 canvas 之中，然后从该 canvas 里捕获像素值。

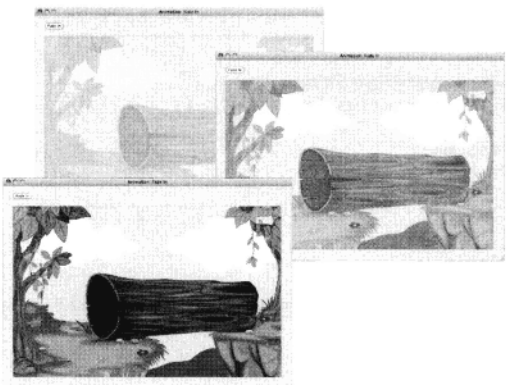


图 4-19　将图像淡入至 canvas 中

程序清单 4-20 列出了图 4-19 所示应用程序的全部代码。

程序清单 4-20　将图像淡入至 canvas 中

```
var image = new Image(),
    canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),
    fadeButton = document.getElementById('fadeButton'),
    imagedata,
    imagedataOffscreen,
    interval = null;

// Functions.....................................................

function increaseTransparency(imagedata, steps) {
   var alpha,
       currentAlpha,
       step,
       length = imagedata.data.length;

   for (var i=3; i < length; i+=4) { // For every alpha component
      alpha = imagedataOffscreen.data[i];

      if (alpha > 0) {
```

```
            currentAlpha = imagedata.data[i];
            step = Math.ceil(alpha/steps);

            if (currentAlpha + step <= alpha) { // Not at original alpha yet
               imagedata.data[i] += step; // Increase transparency
             }
             else {
               imagedata.data[i] = alpha; // End: original transparency
             }
         }
      }
   }

   function fadeIn(context, imagedata, steps, millisecondsPerStep) {
      var frame = 0;

      for (var i=3; i < imagedata.data.length; i+=4) { // For every alpha
          imagedata.data[i] = 0;
      }

      interval = setInterval(function () { // Every millisecondsPerStep
         frame++;

         if (frame > steps) {
            clearInterval(interval);
         }
         else {
           increaseTransparency(imagedata, steps);
            context.putImageData(imagedata, 0, 0);
         }
      }, millisecondsPerStep);
   }

   // Animation.............................................................

   function animationComplete() {
         setTimeout(function() {
            context.clearRect(0, 0, canvas.width, canvas.height);
         }, 1000);
      }

   // Event handlers........................................................

   fadeButton.onclick = function() {
      imagedataOffscreen = offscreenContext.getImageData(0, 0,
                              canvas.width, canvas.height);

      fadeIn(context,
             offscreenContext.getImageData(0, 0,
                              canvas.width, canvas.height),
             50,
             1000 / 60);
   };

   // Initialization........................................................

   image.src = 'log-crossing.png';
   image.onload = function() {
      offscreenCanvas.width = canvas.width;
      offscreenCanvas.height = canvas.height;
      offscreenContext.drawImage(image,0,0);
   };
```

4.8 图像绘制的安全问题

图像经常会带来安全隐患。比如，你可能想限制他人随意存取你发表在社交网络上的图片，或是某公司需要对其产品原型图保密，又或者某位从政人员需要对某些图片加密等。

所以，基于安全考量，HTML5 Canvas 规范允许绘制不属于自己的（也就是其他域中的）图像，然而，你不能通过 Canvas API 保存或修改其他域中的图像。

Canvas 绘图安全机制的原理如下：

每个 canvas 都有一个名为 origin-clean 的标志位，它的初始值是 true。如果使用 drawImage() 绘制了一幅其他域中的图像，那么 origin-clean 的值就会被设置为 false。与此类似，如果用 drawImage() 将另一个 origin-clean 标志为 false 的 canvas 绘制到当前的 canvas 中，那么它的 origin-clean 标志也会被设置为 false。

就其本身而言，将 canvas 的 origin-clean 设置为 false，并不会立刻导致诸如抛出异常这样的反应来。不过，如果在 origin-clean 标志为 false 的 canvas 上调用 toDataURL() 或 getImageData() 方法，那么此时浏览器则会抛出 SECURITY_ERR 异常。

浏览器会将用户的文件系统与运行应用程序的环境视为两个不同的域。所以，在默认情况下，你不能保存或修改文件系统中的图像。然而，这项安全措施在软件开发阶段很碍事，所以，很多浏览器都提供了临时绕过它的办法。比如说，可以通过在命令行中指定“--allow-file-access-from-files”参数来启动 Chrome 浏览器。这个参数会让浏览器绕开绘图安全机制，使我们可以保存或修改其他域中的图像。使用火狐浏览器时，可以调用如下函数：

```
netscape.security.PrivilegeManager.enablePrivilege(
                                        "UniversalBrowserRead");
```

如果以“--allow-file-access-from-files”作为命令行参数来启动 Chrome 浏览器，那么运行在其中的所有应用程序就可以保存或修改其他域中的图像了。然而，如果是在 Firefox 浏览器中通过调用 PrivilegeManager 的 enablePrivilege() 来绕过安全机制的话，那么只能在调用 enablePrivilege() 的这个方法内部保存或修改其他域中的图像。

小技巧：如何运行本书的范例程序

可以在 corehtml5canvas.com 网站下载本书所有范例程序的代码，也可以直接在网站上运行它们。如果选择先将代码下载到本机磁盘中，然后再运行，那么请注意：必须采用本节中所讲的其中一种方式来放宽浏览器对于跨域图像操作的限制，否则就无法运行本书中的某些范例程序，因为它们要使用 toDataURL() 与 getImageData() 来创建或修改图像。

4.9 性能

在进行图像处理时，亟须考虑性能问题。本节将会研究 jsperf.com 网站上的三个性能测试，它们所测试的内容如下：

- 遍历图像数据。
- 对比 drawImage() 与 putImageData() 的绘图效率。
- 使用 drawImage() 来绘制 canvas，而非普通的图像。
- 在使用 drawImage() 绘图时缩放图像。

还是要提醒大家，任何性能测试的结果都有可能因为运行时机或浏览器的差异而产生巨大变

化。所以说，性能测试只能作为编码时的参考，而不能作为其基本原则。我们应该经常去 jsperf.com 网站看看这些性能测试在当前状况下的表现。

4.9.1 对比 drawImage(HTMLImage)、drawImage(HTMLCanvas) 与 putImageData() 的绘图效率

drawImage() 与 putImageData() 都可以将图像绘制到 canvas 之中。在写作本书时，drawImage() 要比 putImageData() 快很多。

除了绘制性能高这个优势之外，drawImage() 还有一项 putImageData() 所不具备的功能：它可以将某个 canvas 绘制到另外一个 canvas 之中。正如本小节中的测试结果所示，绘制 canvas 的速度通常并不会比绘制图像慢很多。

- drawImage() 优于 putImageData()。
- 绘制 canvas 通常与绘制图像一样快。

以下是设置性能测试所用的代码：

```
<canvas width=364 height=126 id="c1"></canvas>
<canvas width=364 height=126 id="c2"></canvas>
<img src='...'/>

<script>
   var c1 = document.getElementById('c1').getContext('2d');
   var c2 = document.getElementById('c2').getContext('2d');
   var c2_c = document.getElementById('c2');
   var img = document.getElementById('imgd');
   c1.drawImage(img, 0, 0);
   var imgData = c1.getImageData(0, 0, parseInt(img.width),
                                       parseInt(img.height));
   function execute(drawMethod) {
      for(var i=0; i< 100; i++) {
         drawMethod(i);
      }
   }
</script>
```

这段测试代码创建了两个 canvas 与一个 Image，并将它绘制到其中一个 canvas 之上，然后获取了该 canvas 的图像数据，通过 imgData 变量来引用这份数据。这段设置代码还实现了一个 execute 函数，用以调用三个受测方法。

图 4-20 列出了测试用例[⊖]及其运行结果。

从测试用例的运行结果中可以看出，putImageData() 几乎总是要比 drawImage() 慢，而且要慢很多。所以说，在同等条件下，总是应该优先考虑使用 drawImage()。

4.9.2 在 Canvas 中绘制另一个 Canvas 与绘制普通图像之间的对比；在绘制时缩放图像与保持原样之间的对比

在 4.3 节中我们讲过如何将 canvas 绘制到其自身。在绘制时还可以对 canvas 的图像进行缩放。正如测试结果所示，将 canvas 绘制到其自身是一项耗时的操作，如果在绘制时还要进行缩放，则更加耗时。

- 将 canvas 绘制到其自身很耗时。
- 绘制 canvas 时对其缩放也很耗时。

⊖ 该测试的执行页面是：http://jsperf.com/canvas-drawimage-vs-putimagedata/21。——译者注

	Test	Ops/sec

DrawImage	`function d(i) {`
	`  c1.drawImage(img, 0, 0);`
	`}`
	`execute(d)`
		ready

PutPixel	`function p(i) {`
	`  c2.putImageData(imgData, 0, 0);`
	`}`
	`execute(p)`
		ready

DrawImage(canvas)	`function p(i) {`
	`  c1.drawImage(c2_c, 0, 0);`
	`}`
	`execute(p)`
		ready

Compare results of other browsers

Browserscope

UserAgent	DrawImage	DrawImage canvas	PutPixel	# Tests
Chrome 11.0.696	146	141	33	1
Chrome 12.0.742	193	198	37	12
Chrome 13.0.782	237	247	37	4
Chrome 14.0.791	99	143	109	1
Chrome 14.0.814	92	141	104	2
Firefox 5.0	533	537	16	8
iPad 4.3	148	143	18	6
iPad 4.3.3	22	16	18	2
Opera 11.50	224	137	50	1
Safari	29	16	2	1

Browserscope thinks you are using **Chrome 14.0.814**. No?

图 4-20　drawImage(HTMLImage)、drawImage(HTMLCanvas) 与 putImageData() 的绘图效率对比；分数越高，绘图效率越好

这个简单的性能测试[⊖]，其代码如下：

```
<script>
   var c = document.createElement('canvas');
   c.width = 256;
   c.height = 256;

   var ctx = c.getContext('2d');
   ctx.clearRect(0,0,c.width,c.height);

   var img = new Image(),
   img.src = c.toDataURL();
</script>
```

这段测试代码先创建了一个 canvas 元素，并将一部分像素重置为全透明的黑色[⊜]。然后又创建了一个 Image 图像，并将其 src 属性设置为表示 canvas 元素的图像数据。

图 4-21 展示了此测试用例的运行结果。

4.9.3　遍历图像数据

就其本质而言，操作图像数据是一件与性能紧密相关的事情。遍历含有大量数据的数组是非

⊖ 该测试的执行页面是：http://jsperf.com/canvas-drawimage-draw-canvas-into-itself。——译者注

⊜ 原文用的是clear black，Canvas规范中用的是transparent black，意思都是指全透明的黑色，该色用数值可表示为0x00000000，其Alpha值与红、绿、蓝分量皆为0。——译者注

常耗时的。所幸，在操作 canvas 图像数组时，有很多办法可以提高程序运行效率：

- 避免在循环体内直接访问对象的属性，而是应该将其存放于局部变量中。
- 应该用循环计数器来遍历完整的像素，而非像素分量。
- 逆向遍历与移位（bit-shifting）技巧的效果并不好。
- 不要频繁地调用 getImageData() 来获取少量数据。

	Test	Ops/sec
Copy, unscaled	ctx.drawImage(ctx.canvas, 0, 0);	ready
Copy, scaled	ctx.drawImage(ctx.canvas, 0, 0, ctx.canvas.width*2, ctx.canvas.height*2);	ready
Draw image	ctx.drawImage(img, 0, 0);	ready

You can edit these tests or add even more tests to this page by appending /edit to the URL.

Compare results of other browsers

Browserscope

UserAgent	Copy scaled	Copy unscaled	Draw image	# Tests
Chrome 14.0.814	292	1,270	8,863	1
Firefox 5.0.1	352	660	12,292	1
iPad 4.3.2	180	639	1,427	1
Safari 5.0.1	380	772	11,641	1

Browserscope thinks you are using Chrome 14.0.814 No?

图 4-21　将 canvas 绘制到其自身：分数越高，绘图效率越好

现在我们就来看看 jsPerf 网站上的这个性能测试都采用了哪些遍历图像数据的方式。首先列出设置代码：

```
var canvas = document.createElement('canvas');
canvas.width = 256;
canvas.height = 256;

var ctx = canvas.getContext('2d');
ctx.fillRect(0, 0, 256, 256);

var id = ctx.getImageData(0, 0, 256, 256);
var pixels = id.data;
var length = pixels.length;
var width = id.width;
var height = id.height;
```

上述代码首先创建了一个 canvas 元素，设置了它的宽度与高度，并将其所有像素重置为全透明的黑色。然后，这段代码调用 getImageData() 方法来获取指向 canvas 图像数据的引用。最后，将数组长度及图像数据宽度等属性保存到局部变量中。

图 4-22 与图 4-23 展示了各种测试用例的运行结果。

4.9.3.1　避免在循环体内直接访问对象属性，而应将其存于局部变量中

在图 4-22 中，前 4 个测试用例持续对比了两种访问图像数据的方式。一种是直接使用 width 与 height 这样的对象属性，另一种则是先将其存于局部变量中，然后再使用它。

这些测试也比较了使用单层循环与双层循环遍历图像数据时的速度。

如果使用单层循环来遍历图像数据，那么直接访问对象属性与将其存于局部变量相比，速度

上没有差别。然而，如果用双层循环来遍历，那么通过局部变量来访问就要比直接访问对象属性快得多。因此，把需要访问的对象属性存于局部变量中，似乎是个好办法。

	Test	Ops/sec

property accesses 2d	`for (var y = 0; y < id.height; y++) {`
	`  for (var x = 0; x < id.width; x++) {`
	`  var off = (y * id.width + x) * 4;`
	`  id.data[off] += 10;`
	`  id.data[off + 1] += 20;`
	`  id.data[off + 2] += 30;`
	`  id.data[off + 3] += 40;`
	`  }`
	`}`
		ready

property accesses 1d	`for (var i = 0; i < id.data.length; i += 4) {`
	`  id.data[i] += 10;`
	`  id.data[i + 1] += 20;`
	`  id.data[i + 2] += 30;`
	`  id.data[i + 3] += 40;`
	`}`
		ready

local variables 2d	`for (var y = 0; y < height; y++) {`
	`  for (var x = 0; x < width; x++) {`
	`  var off = (y * width + x) * 4;`
	`  pixels[off] += 10;`
	`  pixels[off + 1] += 20;`
	`  pixels[off + 2] += 30;`
	`  pixels[off + 3] += 40;`
	`  }`
	`}`
		ready

local variables 1d	`for (var i = 0; i < length; i += 4) {`
	`  pixels[i] += 10;`
	`  pixels[i + 1] += 20;`
	`  pixels[i + 2] += 30;`
	`  pixels[i + 3] += 40;`
	`}`
		ready

local variables 1d hack one	`for (var i = -1; i < length;) {`
	`  pixels[++i] += 10;`
	`  pixels[++i] += 20;`
	`  pixels[++i] += 30;`
	`  pixels[++i] += 40;`
	`}`
		ready

local variables 1d hack two	`var i = -1;`
	`while (i < length) {`
	`  pixels[++i] += 10;`
	`  pixels[++i] += 20;`
	`  pixels[++i] += 30;`
	`  pixels[++i] += 40;`
	`}`
		ready

local variables 1d hack three	`var i = length;`
	`while (i >= 0) {`
	`  pixels[--i] += 40;`
	`  pixels[--i] += 30;`
	`  pixels[--i] += 20;`
	`  pixels[--i] += 10;`
	`}`
		ready

图 4-22　图像数据的遍历（这一组测试用例的运行页面是：http://bit.ly/novcmK）

4.9.3.2　应该使用循环计数器来遍历完整的像素，而非像素分量

前面讲过，图像数据中的每个像素都是以 4 个 8 位二进制整数来保存的，它们分别表示像素的红、绿、蓝及 Alpha 分量，每个分量的取值范围是 0 ～ 255。我们再回忆一下，浏览器为了让图像显示得更清晰，有可能将单个的 CSS 像素使用多个设备像素来表示。

如果要按照像素分量来遍历的话，那么循环计数器[㊀]的步进次数将会是像素个数的 4 倍。因此最好是每次循环体都以同样的计数值为基准来遍历一个完整的像素[㊁]，而不要每次只遍历一个像素分量[㊂]。正是由于这个原因，所以在很多测试用例中，循环计数器每次的步进都是 4。

读者可能会觉得连续四次对循环下标进行自增运算，容易大幅降低运行效率。这么想也许是

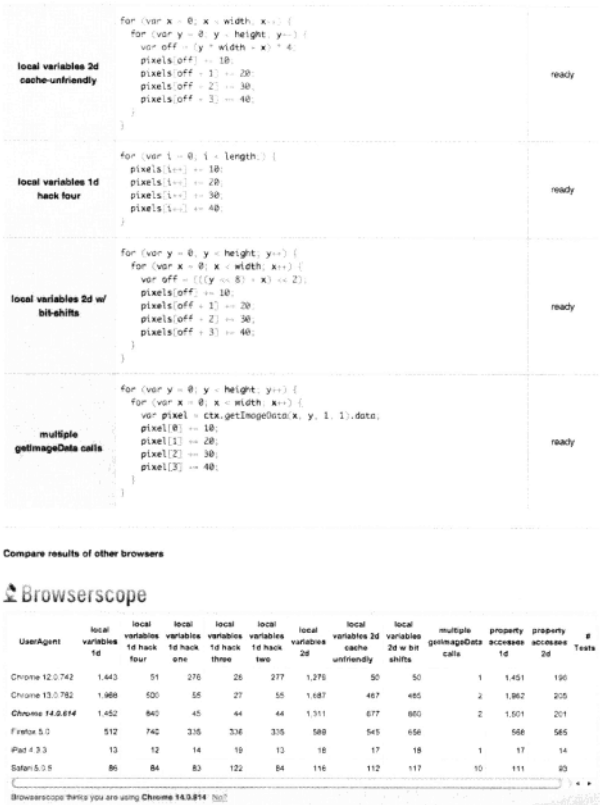
㊀ 这里的循环计数器是指在测试用例的循环语句中常用的数组下标“i”。——译者注

㊁ 例如以i、i+1、i+2、i+3这样的形式来遍历。——译者注

㊂ 例如以连续4次++i的形式来遍历。——译者注

对的。图 4-23 所示测试结果证实了由此引发的性能下降，不过 iPad 与 Safari 浏览器例外。

在图 4-22 之中，“local variable 1d”与“local variable 1d hack one”这两个测试用例分别对比了“以循环计数值为基准遍历一个完整的像素”与“用循环计数器来遍历每个像素分量”这两种算法的性能。我们会发现，以完整的像素为单位进行遍历通常比按照像素分量遍历要快，而且有时会快很多。



local variables 2d cache-unfriendly	`for (var x = 0; x < width; x++) { for (var y = 0; y < height; y++) { var off = (y * width + x) * 4; pixels[off] += 10; pixels[off + 1] += 20; pixels[off + 2] += 30; pixels[off + 3] += 40; } }`	ready
local variables 1d hack four	`for (var i = 0; i < length;) { pixels[i++] += 10; pixels[i++] += 20; pixels[i++] += 30; pixels[i++] += 40; }`	ready
local variables 2d w/ bit-shifts	`for (var y = 0; y < height; y++) { for (var x = 0; x < width; x++) { var off = (((y << 8) + x) << 2); pixels[off] += 10; pixels[off + 1] += 20; pixels[off + 2] += 30; pixels[off + 3] += 40; } }`	ready
multiple getImageData calls	`for (var y = 0; y < height; y++) { for (var x = 0; x < width; x++) { var pixel = ctx.getImageData(x, y, 1, 1).data; pixel[0] += 10; pixel[1] += 20; pixel[2] += 30; pixel[3] += 40; } }`	ready

Compare results of other browsers

Browserscope

UserAgent	local variables 1d	local variables 1d hack four	local variables 1d hack one	local variables 1d hack three	local variables 1d hack two	local variables 2d	local variables 2d cache unfriendly	local variables 2d w bit shifts	multiple getImageData calls	property accesses 1d	property accesses 2d	# Tests
Chrome 12.0.742	1,443	51	276	26	277	1,278	50	50	1	1,451	196	
Chrome 13.0.782	1,988	500	55	27	55	1,687	467	485	2	1,962	205	
Chrome 14.0.814	1,452	840	45	44	44	1,311	677	860	2	1,501	201	
Firefox 5.0	512	742	335	336	335	589	545	656		568	565	
iPad 4.3.3	13	12	14	19	13	18	17	18	1	17	14	
Safari 5.0.5	86	84	83	122	84	116	112	117	10	111	93	

Browserscope thinks you are using **Chrome 14.0.814** No?

图 4-23　各种图像数据遍历方式的运行效率统计；分数越高，效率越好

4.9.3.3　逆向遍历与移位技巧的效果不佳

长久以来，一直盛行某些说法，认为在 JavaScript 中，逆向遍历数组与通过移位操作来计算数组偏移量能够提高代码执行速度。根据图 4-23 所列测试结果，我们看到移位技巧的执行效率与普通的方法相比并没有明显差别，而在某些情况下，逆向遍历却降低了程序的性能。逆向遍历（名为“local variable 1d hack three”的测试用例）在 Chrome 浏览器上运行得极慢，在 Firefox 浏览器中与普通方法速度相同，在 Safari 与 iPad 上则比正向遍历（名为“local variable 1d hack one”的测试用例）稍快一点。也就是说，采用这两种技巧的效果并不太好。

在决定采用逆向遍历或移位技巧之前，最好先做一下性能测试，但愿你能找到比这两种办法效率更高的手段来提升程序性能。

4.9.3.4 不要频繁地调用 getImageData() 来获取少量数据

与一般情况下使用 getImageData() 来获取全部图像数据相比，图 4-23 之中最后一个测试用例所演示的用法则是频繁地调用该方法来获取图像数据之中的每一个像素。

尽管不敢武断地给出结论，不过我们可以相当有把握地说，getImageData() 方法很耗时，像上述测试用例这样，“对图像数据中的每个像素，都调用一次 getImageData()”的做法，你应该极力避免。

4.10 放大镜

图 4-24 演示了本章开头所说的放大镜应用程序。可以拖动它来放大显示图像中的各个部分，也可以拖动程度顶部的滑动条来调整放大镜的大小及放大倍数。

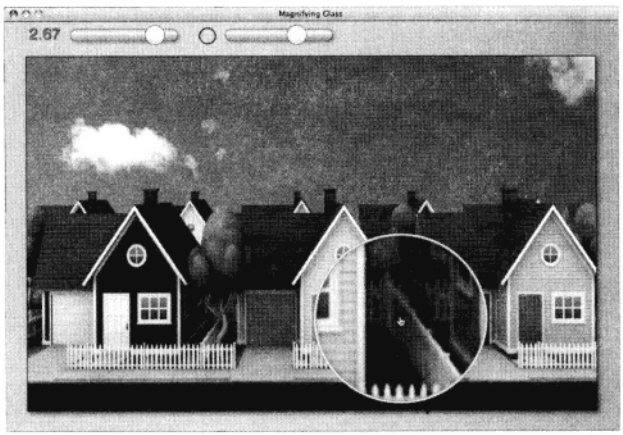


图 4-24 放大镜

该程序的工作原理如下：

当用户拖动鼠标时，程序捕获放大镜最小外接矩形范围内的像素。

然后，程序将剪辑区域设置为放大镜所在范围，并将刚捕获的图像绘制到 canvas 自身，在绘制时调用接受 9 个参数的 drawImage() 方法来放大图像。

除了绘制被放大镜放大的图像之外，应用程序还会擦除用户拖动之前的那个放大镜图像。每当用户拖动放大镜时，应用程序就会调用 putImageData() 方法把上一次移动鼠标时用 getImageData() 所捕获的背景图像重新恢复到 canvas 中。

所以，每次用户拖动放大镜时，应用程序都会按如下步骤行事：

（1）调用 putImageData() 方法，将上一次放大镜所在位置的背景图像恢复到 canvas 中。

（2）调用 getImageData() 方法，捕获放大镜当前位置下面的像素数据。

（3）将剪辑区域设定为放大镜所在范围。

（4）调用 drawImage() 方法，将放大后的图像绘制到 canvas 中。

（5）绘制放大镜的镜片。

以下是应用程序处理鼠标移动事件所用的代码：

```
canvas.onmousemove = function (e) {
   if (dragging) {
      eraseMagnifyingGlass();
      drawMagnifyingGlass(windowToCanvas(e.clientX, e.clientY));
   }
};
```

eraseMagnifyingGlass() 方法执行了上述步骤中的第 1 步。

```
function eraseMagnifyingGlass() { // Called when the mouse moves
   if (imageData != null) {
      context.putImageData(imageData,
         magnifyRectangle.x, magnifyRectangle.y);
   }
}
```

应用程序第一次调用 eraseMagnifyingGlass() 方法时，并没有需要擦除的内容，所以首先用 imageData != null; 语句判断是不是这种情况，如果不是，那么程序就调用 putImageData() 方法，将上次所画的放大镜图像擦除。

在擦掉了放大镜图像之后，应用程序的鼠标移动事件处理器会调用 drawMagnifyingGlass() 方法，它的实现代码如下：

```
function drawMagnifyingGlass(mouse) {
   var scaledMagnifyRectangle = null;

   magnifyingGlassX = mouse.x;
   magnifyingGlassY = mouse.y;

   calculateMagnifyRectangle(mouse);

   imageData = context.getImageData(magnifyRectangle.x,
                                    magnifyRectangle.y,
                                    magnifyRectangle.width,
                                    magnifyRectangle.height);
   context.save();

   scaledMagnifyRectangle = {
      width:  magnifyRectangle.width  * magnificationScale,
      height: magnifyRectangle.height * magnificationScale
   };

   setClip();

   context.drawImage(canvas,
      magnifyRectangle.x, magnifyRectangle.y,
      magnifyRectangle.width, magnifyRectangle.height,

      magnifyRectangle.x + magnifyRectangle.width/2 -
         scaledMagnifyRectangle.width/2,

      magnifyRectangle.y + magnifyRectangle.height/2 -
         scaledMagnifyRectangle.height/2,

      scaledMagnifyRectangle.width,
      scaledMagnifyRectangle.height);

   context.restore();

   drawMagnifyingGlassCircle(mouse);
}

function setClip()  {
   context.beginPath();
   context.arc(magnifyingGlassX, magnifyingGlassY,
               magnifyingGlassRadius, 0, Math.PI*2, false);

   context.clip();
}
```

drawMagnifyingGlass() 函数计算出当前放大镜所在位置的最小外接矩形，并捕获该范围内的像素，以便下次应用程序在绘制新的放大镜图像时可以用它擦掉旧的图像。

然后，应用程序算出放大之后的图像宽度与高度，并将剪辑区域设置为放大镜所在的范围。

最后，drawMagnifyingGlass() 方法将 canvas 绘制到其自身，并在绘制时放大图像。图 4-25 演示了应用程序是如何通过 drawImage() 方法来绘制图像的。

图 4-25　将放大之后的图像复制到放大镜所在的范围。顶部截图：未设置剪辑区域时的绘制效果；底部截图：设置了剪辑区域之后的绘制效果

图 4-25 顶部的截图是注释掉 setClip() 语句时的绘制效果。如果不设置剪辑区域，那么所有放大之后的图像就都会被 drawMagnifyingGlass() 函数之内的 drawImage() 方法绘制出来。

底部的截图则展示了恢复对 setClip() 的调用后所产生的绘制效果。设置了剪辑区域后，只有在放大镜所在范围内才会显示放大之后的图像。

提示：放大镜程序中的滑动条

放大镜应用程序在页面顶部放置了两个滑动条，用户可以借此来调整放大倍数与放大镜的大小。这些滑动条属于自定义控件，它们各自都是用一个独立的 canvas 来实现的，在第 10 章中将会讲到这部分内容。

4.10.1　使用离屏 canvas

4.10 节所讲到的实现方式是将 canvas 图像绘制到其自身，并在绘制时放大。除了这种办法以外，还可以将放大后的图像先绘制到离屏 canvas 中，稍后再绘制到屏幕 canvas 中，如程序清单 4-21 所示。

程序清单 4-21　使用离屏 canvas 来实现放大镜程序

```
var ...
    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d');
...
```

```
function drawMagnifyingGlass(mouse) {
   var scaledMagnifyRectangle = null;

   magnifyingGlassX = mouse.x;
   magnifyingGlassY = mouse.y;

   calculateMagnifyRectangle(mouse);

   imageData = context.getImageData(magnifyRectangle.x,
                                    magnifyRectangle.y,
                                    magnifyRectangle.width,
                                    magnifyRectangle.height);
   context.save();

   scaledMagnifyRectangle = {
      width:  magnifyRectangle.width  * magnificationScale,
      height: magnifyRectangle.height * magnificationScale
   };

  setClip();

    offscreenContext.drawImage(canvas,
       magnifyRectangle.x, magnifyRectangle.y,
       magnifyRectangle.width, magnifyRectangle.height,
       0, 0,
       scaledMagnifyRectangle.width,
       scaledMagnifyRectangle.height);

    context.drawImage(offscreenCanvas, 0, 0,
       scaledMagnifyRectangle.width,
       scaledMagnifyRectangle.height,

       magnifyRectangle.x + magnifyRectangle.width/2 -
          scaledMagnifyRectangle.width/2,

       magnifyRectangle.y + magnifyRectangle.height/2 -
          scaledMagnifyRectangle.height/2,

       scaledMagnifyRectangle.width,
       scaledMagnifyRectangle.height);

   context.restore();

   drawMagnifyingGlassCircle(mouse);
}
```

对于这个放大镜程序来说，将 canvas 直接绘制到其自身，要比使用离屏 canvas 的性能稍好一些。

4.10.2 接受用户从文件系统中拖放进来的图像

放大镜程序使用了 HTML5 的拖放（Drag and Drop）与文件系统（FileSystem）API[⊖]，使用户可以将图像由桌面拖放到应用程序中。图 4-26 上方的截图演示了该应用程序可以接受用户从桌面拖到浏览器中的图像，而下方的截图则展示了图像被拖放进来之后，程序会将其显示出来。

⊖ Drag and Drop API请参见http://dev.w3.org/html5/spec/dnd.html，FileSystem API的官方页面是http://www.w3.org/TR/file-system-api/。——译者注

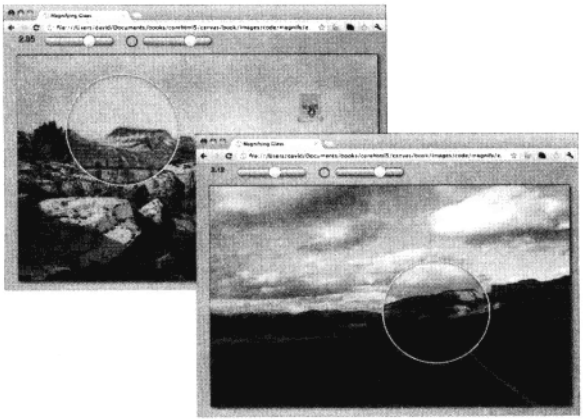


图 4-26 顶部截图：用户正在将图像拖放到浏览器中（注意右上角指示拖放动作的鼠标指针）；底部截图：应用程序将拖放进来的图像显示出来

在本书编写时，Chrome 浏览器是唯一支持文件系统 API 的浏览器。程序清单 4-22 展示了放大镜程序对该 API 的使用方法。

放大镜程序实现了 Drag Enter 及 Drag Over 事件的处理器，并在其代码中阻止浏览器对这两种事件作出默认的处理。程序还在 Drag Enter 事件处理器中告知浏览器，本程序支持拖放操作。

该应用程序在 Drop 事件处理器中通过 FileSystem API 在文件系统上申请了 5MB 空间，然后创建了一幅图像文件，并把通过文件系统获取的 URL 设置为 image 对象的 src 属性。

程序清单 4-22 FileSystem API 的用法

```
canvas.addEventListener('dragenter', function (e) {
   e.preventDefault();
   e.dataTransfer.effectAllowed = 'copy';
}, false);

canvas.addEventListener('dragover', function (e) {
   e.preventDefault();
}, false);

window.requestFileSystem =
          window.requestFileSystem || window.webkitRequestFileSystem;

canvas.addEventListener('drop', function (e) {
   var file = e.dataTransfer.files[0];

   window.requestFileSystem(window.TEMPORARY, 5*1024*1024,
      function (fs) {
         fs.root.getFile(file.name, {create: true},
            function (fileEntry) {
               fileEntry.createWriter( function (writer) {
                  writer.write(file);
               });
               image.src = fileEntry.toURL();
            },
            function (e) {
```

```
                alert(e.code);
            }
        );
    },

    function (e) {
        alert(e.code);
    }
  );
}, false);
```

4.11 视频处理

视频网站是个很赚钱的生意。2006 年，Google 曾以 16.5 亿美元收购了 YouTube 公司，目前，Google 宣称全世界超过 20% 的互联网流量都是由访问 YouTube 带来的。在这个曾经被 Flash 垄断的领域里，越来越多的视频网站现在已经倾向于使用 HTML5 技术了。

HTML5 提供的 video 元素可以控制视频文件的播放，而且 Canvas API 也允许开发者在播放视频时以逐帧的方式处理其内容。

在 4.1.2 小节中曾经讲过，drawImage() 方法除了能绘制图像，还可以把视频文件的某一帧绘制到 canvas 中，像是这样：

```
var video = document.getElementById('video'); // A <video> element
...

context.drawImage(video, 0, 0); // Draw video frame
```

上述代码中，调用 drawImage() 方法所用的 video 变量，其类型是 HTMLVideoElement。有了这种将视频文件的某一帧绘制到 canvas 中的功能，我们就可以结合 video 与 canvas 元素来做即时视频处理了。4.11.3 小节将会讲述具体做法。

4.11.1 视频格式

表 4-3 总结了本书付印时所广泛使用的三种视频格式。

表 4-3 浏览器对各种视频格式的支持情况

格式	最早支持该格式的浏览器版本
H.264 (MPEG-4)	IE9.0，Chrome 3.0(即将移除)[⊖]，Safari 3.1
Ogg Theora[⊜]	Firefox 3.5，Chrome 3.0，Opera 10.5
VP8 (WebM)	Firefox 4.0，Chrome 6.0，Opera 10.6

要注意的是，以上三种格式中没有哪一种是所有浏览器都支持的。由于此种限制，为了确保所有浏览器都能播放视频，我们需要指定多种格式的文件。在 video 元素中嵌入 source 元素，就可以支持多种格式了，其代码如下：

⊖ Google曾于2011年1月宣布将移除对该格式的支持，但截至2012年8月，Chrome浏览器仍然支持此格式。详情参见：http://blog.chromium.org/2011/01/html-video-codec-support-in-chrome.html。——译者注

⊜ Ogg Theora是由Xiph.Org基金会开发的有损图像压缩技术，它是VP3编码器经过开源后派生而来的，目标是达成比MPEG-4 Part 2更好的编码效率。该基金会还开发了著名的声音编码技术Vorbis，以及多媒体容器文件格式Ogg。详情参见：http://zh.wikipedia.org/zh-cn/Theora。——译者注

```
<video>
  <source src='video.ogg'/>
  <source src='video.mp4'/>
</video>
```

提示：视频格式简史

HTML5 规范起初将 Ogg Theora 定为标准的视频格式，因为它是免费而且开源的，还因为规范制定者认为使用单一的视频格式要比支持多种格式更好。Mozilla 与 Opera 都大力支持 Ogg Theora 格式。

然而，像 Apple 与 Nokia 这样的公司则担心专利问题（参见下一个提示框），而且 Apple 认为在规范书中指明视频格式并不好。

于是，重写后的规范书移除了对 Ogg Theora 格式的强制支持。随后，Google 在 2010 年收购了 On2 公司[㊀]，取得其 VP8 视频格式的所有权，并以 BSD 风格的使用条款将它发布，这是一种“不可撤回的免费专利”条款[㊁]。2011 年 1 月，Google 曾宣布，Chrome 浏览器将不再提供对 MPEG-4 格式的原生支持。

提示：“潜水艇专利”与“专利伏击”

在 1995 年之前，美国的专利条款以专利的公布日期（the date of issuance）而非原始申请日期（original filing date）作为计算其年限的法律依据。虽说费用有些昂贵，但是专利申请者可以尽量拖延专利的公布日期。这样的专利就叫做“潜水艇专利”（submarine patent）。

一旦有某个大公司侵犯了这种潜水艇专利，那么申请人就不再请求延期，而是将其公布出来，这样的话，原本不知情的公司就会受到专利持有人的控告，可能会因此被索赔一大笔钱。

潜水艇专利只是软件专利领域中隐晦的一角。还有一种叫做“专利伏击”（patent ambush）的策略，指的是某个即将申请专利的公司派其员工参加软件标准化组织，然后诱导该组织制定出侵犯专利的标准来。

尽管 Ogg 格式并未侵犯任何已知的专利技术，但是 Apple 和 Nokia 都反对采用此格式，其中一部分原因就是担心受到潜水艇专利和专利伏击的影响。

视频格式的转换

鉴于三种主流视频格式都未获得全部浏览器的支持，为保持各浏览器平台之间的可移植性，需要为同一份视频文件提供不同的编码格式。由于有了这个要求，所以我们迟早会碰到在各种视频格式之间进行转换的问题。

视频格式的转换有很多种办法，用图 4-27 所示的 Firefox 浏览器插件[㊂]就可以转换格式。

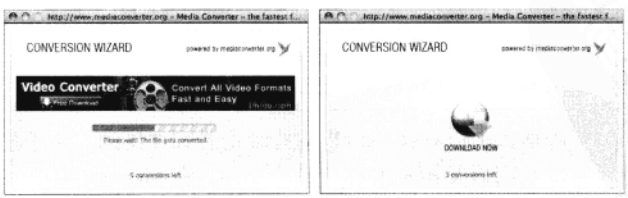


图 4-27 用 Firefox 插件来转换视频格式

㊀ 全称On2 Technologies，是美国一家视频压缩科技公司，以开发 TrueMotion S、TrueMotion 2、VP3、VP4、VP5、TrueMotion VP6、TrueMotion VP7 以及 VP8等产品闻名。——译者注）

㊁ irrevocable free patent，该条款详见：http://www.webmproject.org/license/software/。——译者注

㊂ 该插件的网址是：https://addons.mozilla.org/zh-CN/firefox/addon/media-converter/。——译者注

4.11.2 在 Canvas 中播放视频

我们研究 Canvas 视频功能的最终目标是为了实现即时视频处理。然而我们第 1 步要做的则是先用 canvas 来播放一段视频。图 4-28 所示应用程序可以把不可见的 video 元素中所含视频文件的每一帧，绘制到可见的 canvas 元素中，并在绘制时将其缩放至与 canvas 相同的大小。

图 4-28 在 canvas 中播放视频文件

程序清单 4-23 列出了图 4-28 所示应用程序的 HTML 代码。

请注意运用在 video 元素上的 CSS 代码，其中的 display 属性使得该元素不可见。

应用程序先播放不可见的 video 元素之中的视频文件，然后以播放动画循环的形式将其绘制出来。每次循环都利用 requestNextAnimationFrame() 函数将视频中的当前帧绘制到 canvas 中，以此来实现视频播放功能。本书 5.1.3 小节将会讲到动画循环所用到的这个 Polyfill 函数。该应用程序的 JavaScript 代码列在了程序清单 4-24 之中。

加载完视频文件后，应用程序就开始播放视频，并调用 requestNextAnimationFrame() 函数来启动动画循环。当浏览器准备好了下一帧将要绘制的动画时，它就会调用 animate() 函数。如果此时视频还未播放完，那么它就把当前帧绘制到 canvas 之中，然后再次调用 requestNextAnimationFrame() 函数，使动画继续播放下去。如果视频已经播放完毕，那么 animate() 函数就不再调用 requestNextAnimationFrame() 函数了，于是动画也就停止了。

程序清单 4-23 在 canvas 中播放视频文件（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Video</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            background: #ffffff;
            border: thin solid darkgray;
         }

         #video {
            display: none;
```

```
        }
      </style>
    </head>

  <body>
    <video  id='video' poster>
       <source src='dog-stealing.mp4'/>
       <source src='dog-stealing.ogg'/>
    </video>

    <canvas id='canvas' width='720' height='405'>
      Canvas not supported
    </canvas>

    <script src='requestNextAnimationFrame.js'></script>
    <script src='example.js'></script>
  </body>
</html>
```

请注意，JavaScript 代码中使用了接受 5 个参数的 drawImage() 方法，用它来把视频中的每一帧都缩放至与 canvas 相同的尺寸。本书 4.1.2 小节曾讲过该方法。

程序清单 4-24　在 canvas 中播放视频文件（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    video = document.getElementById('video');

function animate() {
   if (!video.ended) {
     context.drawImage(video, 0, 0, canvas.width, canvas.height);
     window.requestNextAnimationFrame(animate);
   }
}

video.onload = function (e) {
   video.play();
   window.requestNextAnimationFrame(animate);
};
```

我们已经学会了如何捕获视频中的每一帧并把它显示到 canvas 中，接下来再看看怎么在显示每一帧之前对其图像进行处理。

4.11.3　视频处理

图 4-29 所示应用程序与前一小节中讨论的程序类似，也是把一个不可见的 video 元素之中的视频绘制到一个可见的 canvas 元素中。在播放时，该程序还可以根据用户的选择，先对视频中的帧进行处理，然后再将其显示到 canvas 中。

应用程序提供了两个复选框，用于控制视频的颜色及方向，还提供了一个“Play”按钮，用户可以通过它来播放视频。

图 4-29 所示应用程序的代码列在了程序清单 4-25 之中。

该程序的 JavaScript 代码列在程序清单 4-26 之中，这段代码实现了一个会持续运行的动画循环，它将不可见的 video 元素之中的视频文件逐帧绘制到可见的 canvas 元素中。

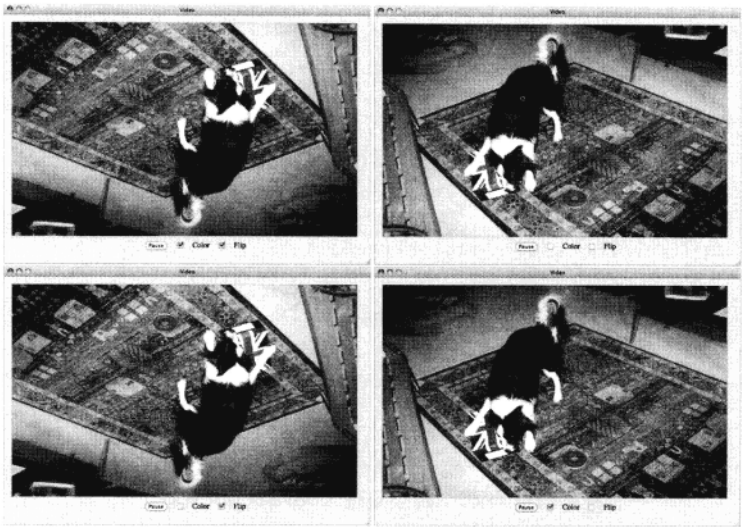


图 4-29　对视频进行图像处理

当浏览器准备好下一帧动画时，它就会调用 nextVideoFrame() 函数，所有视频处理都会在该函数中进行。如果视频已经播放完了，那么它就把按钮的文本设置成“Play”，并且不再调用 requestNextAnimationFrame() 函数，于是动画就不会继续播放了。

假如视频还未播完，那么 nextVideoFrame() 就会将当前帧绘制到离屏 canvas 中，然后根据用户的选择来决定是否要对该帧进行黑白及翻转处理。做完这些后，程序就将离屏 canvas 中的帧图像绘制到屏幕 canvas 之上。

程序清单 4-25　视频处理（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Video</title>

      <style>
         body {
            background: #dddddd;
         }

         .floatingControls {
            position: absolute;
            left: 175px;
            top: 300px;
         }

         #canvas {
            background: #ffffff;
            border: thin solid #aaaaaa;
         }

         #video {
            display: none;
```

```
        }
      </style>
    </head>

    <body>
      <video id='video' controls src='dog-stealing.mp4'></video>

      <canvas id='canvas' width='480' height='270'>
        Canvas not supported
      </canvas>

      <div id='controls' class='floatingControls'>
         <input id='controlButton' type='button' value='Play'/>
         <input id='colorCheckbox' type='checkbox' checked> Color
         <input id='flipCheckbox' type='checkbox'> Flip
      </div>

      <script src='requestNextAnimationFrame.js'></script>
      <script src='example.js'></script>
  </body>
</html>
```

程序清单 4-26　视频处理（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),
    context = canvas.getContext('2d'),
    video = document.getElementById('video'),
    controlButton = document.getElementById('controlButton'),
    flipCheckbox = document.getElementById('flipCheckbox'),
    colorCheckbox = document.getElementById('colorCheckbox'),
    imageData,
    poster = new Image();

// Functions.........................................................

function removeColor() {
   var data,
       width,
       average;

   imageData = offscreenContext.getImageData(0, 0,
                 offscreenCanvas.width, offscreenCanvas.height);
   data = imageData.data;
   width = data.width;

   for (i=0; i < data.length-4; i +=  4) {
      average = (data[i] + data[i+1] + data[i+2]) / 3;
      data[i]   = average;
      data[i+1] = average;
      data[i+2] = average;
   }

   offscreenContext.putImageData(imageData, 0, 0);
}

function  drawFlipped() {
   context.save();

   context.translate(canvas.width/2, canvas.height/2);
```

```
    context.rotate(Math.PI);
    context.translate(-canvas.width/2, -canvas.height/2);
    context.drawImage(offscreenCanvas, 0, 0);

    context.restore();
}

function nextVideoFrame() {
    if (video.ended) {
        controlButton.value = 'Play';
    }
    else {
      offscreenContext.drawImage(video, 0, 0);

      if (!colorCheckbox.checked)
         removeColor();

      if (flipCheckbox.checked)
         drawFlipped();
      else
        context.drawImage(offscreenCanvas, 0, 0);

      requestNextAnimationFrame(nextVideoFrame);
    }
}

function startPlaying() {
    requestNextAnimationFrame(nextVideoFrame);
    video.play();
}

function stopPlaying() {
    video.pause();
}

// Event handlers.....................................................

controlButton.onclick = function(e) {
    if (controlButton.value === 'Play') {
        startPlaying();
        controlButton.value = 'Pause';
    }
    else {
        stopPlaying();
        controlButton.value = 'Play';
    }
};

poster.onload = function() {
    context.drawImage(poster, 0, 0);
};

// Initialization....................................................

poster.src = 'dog-stealing-poster.png';

offscreenCanvas.width = canvas.width;
offscreenCanvas.height = canvas.height;
```

4.12 总结

Canvas API 将很多重要的功能都封装在 Canvas 绘图环境对象的 4 个方法中：drawImage() 可以将另一个 canvas 的内容、视频文件的帧或是图像绘制到当前 canvas 中；getImageData() 用来捕获 canvas 中某个矩形区域内的像素；putImageData() 可以修改 canvas 中某个矩形区域内的像素值；createImageData() 则用于创建一个包含像素颜色值的空数组。

用这 4 个方法就可以实现许多复杂的图像操作，例如图像滤镜或者放大镜。

本章讲述了图像的绘制与缩放、图像滤镜的实现、离屏 canvas 中的图像处理以及工作线程的运用。此外，还讲解了如何将图像操作与诸如剪辑区域及 Canvas 绘图 API 等其他 HTML5 Canvas 技术结合起来使用，如何在图像处理中使用例如工作线程这样位于 Canvas 范围之外的 HTML5 技术。将图像滤镜算法封装在工作线程中，可以减轻浏览器主 UI 线程的工作量，并使得这些工作线程在很多场合下都能被重用。读者还在本章中学到了一些提升绘图及图像操作性能的技巧。

然后，我们讲述了放大镜程序的实现，该程序用到了本章所讲的许多知识。该范例也展示了拖放 API 与文件系统 API 的用法，它们使得程序能够接受用户从桌面拖放到浏览器中的图像。

本章最后讲解了如何利用 video 及 canvas 元素来进行视频处理。处理过程中也用到了动画播放逻辑及 drawImage() 方法。

下一章将研究 Canvas 动画的制作方法，我们看看怎么让这些图像和图形动起来。

第5章 动　画

人们是很喜欢动画的，我们眼中的日常生活，就是以一幅永不止歇的动画来呈现的。所以说，动画是一种很自然、很直观的传播媒介。

动画也是个巨大的产业，从广告业到电子游戏业，有大量的资金都用在了动画制作上，动画在这些产业中扮演着重要的角色。不仅如此，动画的制作过程本身也和编写软件一样有趣。

基于 Flash 的动画曾经风靡网络，不过，这种情况正在改变，新兴的 HTML5 Canvas 技术正在抢夺 Flash 动画的垄断地位。你也许会觉得很惊讶，Canvas 并未对动画技术提供明确的支持。Canvas 所提供的底层图像处理能力完全可以用来创建动画中的帧，然而有关动画循环本身的标准则定义在另外一份 W3C 规范书中。本章讲解如何把动画循环与 Canvas 的图形 API 结合起来使用。

本章第 1 节讲述了用于实现动画循环的各种方式，从传统的 window.setTimeout() 到更新、更强大的 window.requestAnimationFrame()。5.1.3 小节是该节的重点，这一小节所讲的动画循环代码，可以移植到各个浏览器平台中使用，它是利用 window. requestAnimationFrame() 来实现的。

讲完动画循环之后，接下来关注如何将动画做得更平滑。读者将看到几种用于恢复动画背景的方式，以及它们各自对绘制性能的影响。读者还将学到如何实现基于时间的运动，如何实现滚动的背景，以及如何利用视差来模拟三维动画。

5.1　动画循环

就其本身而言，在 Canvas 中实现动画效果很简单：只需要在播放动画时持续更新并绘制就行了。例如，图 5-1 中的动画程序就在持续地绘制三个圆盘。

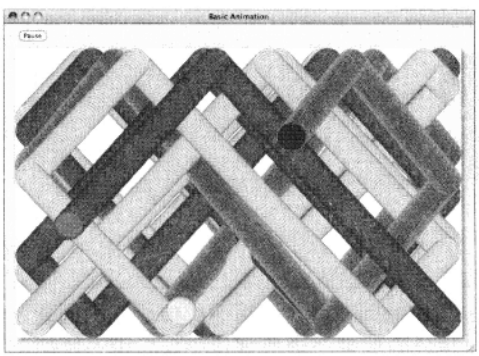


图 5-1　基本动画效果

这种持续的更新与重绘就叫做“动画循环”（animation loop），它是所有动画的核心逻辑。我

们看看它的工作原理吧。

动画是一种持续的循环，但是我们却无法实现这种持续循环，至少我们还不能用运行于浏览器中的 JavaScript 代码来实现传统意义上的持续循环。比如说，程序清单 5-1 所列出的这段 JavaScript 代码，在语法上是正确的，但是无论用多么强大的浏览器去运行它，都会导致浏览器失去响应。

程序清单 5-1 中的 while 循环实际上是个死循环（endless loop）。由于浏览器是在主线程中执行 JavaScript 代码的，所以这种死循环将会使浏览器无法响应用户输入，而且也无法绘制出动画效果。要实现动画效果，必须让浏览器每隔一小段时间就有一个喘息的机会，不能持续地执行死循环。

程序清单 5-1　会导致浏览器死锁的代码，不能用它实现动画

```
function animate() {
   // Update and draw animation objects
}

while(true) { // Locks up the browser: don't do this
   animate();
}
```

为了让浏览器在循环执行过程中得以喘息，可以使用 window.setInterval() 或 window.setTimeout() 来执行循环，它们的用法分别列在程序清单 5-2 与程序清单 5-3 之中。

程序清单 5-2　使用 setInterval() 来实现动画循环

```
function animate() {
  // Update and draw animation objects
}
...

// Start the animation at 60 frames/second

setInterval(animate, 1000 / 60);
```

程序清单 5-3　使用 setTimeout() 来实现动画循环[⊖]

```
// Approximating setInterval() with setTimeout()

function animate() {
   var start = +new Date(),
       finish;

   // Update and draw animation objects

   finish = +new Date();

   setTimeout(animate, (1000 / 60) - (finish - start));
}
...

animate(); // Start the animation
```

setInterval() 方法每隔一定的时间，就会调用一次传给它的函数，然而 setTimeout() 方法则只会在到达指定时间点时，将传给它的那个函数调用一次。由于这两个方法在调用机制上的差别，

⊖ 代码中使用“+new Date()”，是为了通过“一元加”操作符来将变量类型由Date对象转化为Number数值，以便于在调用setTimeout()时使用。——译者注

所以使用 setInterval() 来实现动画循环时，只需调用一次就够了，而如果用 setTimeout() 来做，则需要持续地调用它。请注意，调用 setTimeout() 时，必须明确告诉浏览器下一次执行动画循环的时间。所以，我们每次调用它时，都必须将下次执行动画循环的时间点计算出来。与此相反，如果使用 setInterval() 方法，则只需要指定一次调用间隔即可。

虽说 setTimeout() 与 setInterval() 方法有很多用途，然而它们并不是专门用来实现动画的。实现动画循环的首选方式，是使用 W3C 标准中名为 requestAnimationFrame() 的方法。接下来咱们就看看它的用法。

警告：不要用 window.setInterval() 或 window.setTimeout() 来做动画

我们必须要认识到，window.setInterval() 与 window.setTimeout() 并不能提供制作动画所需的精确计时机制。它们只是让应用程序能在某个大致时间点上运行代码的通用方法而已。

比如，根据 HTML5 规范，浏览器为了节省电力消耗，可以拉长执行代码的间隔时间。以浏览器术语来说，就是“强制规定时间间隔的下限”（clamping the timeout interval）。况且浏览器也确实是在这么做。举例来说，在写作本书时，Firefox 浏览器在单次调用 setTimeout() 时，所允许的最小时间间隔是 10 毫秒，而后续调用的最小间隔则是 5 毫秒。这也就是说，如果你以“3 毫秒”为参数来调用 setTimeout() 方法的话，浏览器会根据规则认定此参数无效，迫使调用者必须等待 10 毫秒才行。

提示：不应主动命令浏览器何时去绘制下一帧动画，这应由浏览器来通知你

虽说 setTimeout() 与 setInterval() 的时间间隔机制并不精确，不过在调用它们时，开发者会主动告知其绘制下一帧动画的时间。

然而，调用者其实并不知道绘制下一帧动画的最佳时机，你很可能根本不了解浏览器绘制动画的内部机制。相反，浏览器肯定比调用者更了解绘制下一帧动画的最佳时机。

所以我们不要像调用 setTimeout() 和 setInterval() 方法时那样，主动命令浏览器何时去绘制下一帧动画，而应该让浏览器在它觉得可以绘制下一帧动画时通知你。我们用 requestAnimationFrame() 方法来实现此功能。

5.1.1　通过 requestAnimationFrame() 方法让浏览器来自行决定帧速率

使用 window.setInterval() 或 window.setTimeout() 制作出的动画，其效果可能并不如预期般流畅，而且还可能会占用额外的资源。这是因为 setInterval() 与 setTimeout() 方法具有如下特征：

- 它们都是通用方法，并不是专为制作动画而用。
- 即使向其传递以毫秒为单位的参数值，它们也达不到毫秒级的精确性。
- 没有对调用动画循环的机制作优化。
- 不考虑绘制动画的最佳时机，而只会一味地以某个大致的时间间隔来调用动画循环。

使用 window.setInterval() 与 window.setTimeout() 方法来制作动画，其根本错误在于它们的抽象程度不符合要求（wrong level of abstraction）。我们想让浏览器执行的是一套可以控制各种细节的动画 API，它能够实现诸如“最优帧速率”（optimal frame rate）、“选择绘制下一帧的最佳时机”等功能。由于 window.setInterval() 与 window.setTimeout() 这两个方法并非专为动画而设，所以如果要使用它们的话，那么这些具体细节就必须留待开发者来完成。

所幸浏览器的开发者们已经意识到，他们需要支持动画功能，并且提供了名为requestAnimationFrame() 的方法。开发者应该用它来制作动画，其用法如程序清单 5-4 所示。

程序清单 5-4 使用 window.requestAnimationFrame() 方法制作动画

```
function animate(time) {
   // Update and draw animation objects

   requestAnimationFrame(animate); // Sustain the animation
}
...

requestAnimationFrame(animate); // Start the animation
```

想要播放动画时，就调用 requestAnimationFrame() 方法，并将指向动画播放函数的引用传递给它。浏览器在决定绘制第一帧动画时，就会调用这个函数。通常在动画播放函数中，我们也会根据情况再次调用 requestAnimationFrame() 方法，使动画循环持续地执行下去。

请注意，与 window.setTimeout() 及 window.setInterval() 方法不同，requestAnimationFrame() 不需要调用者指定帧速率，相反，浏览器会自行决定最佳的帧速率。

W3C 也提供了 cancelRequestAnimationFrame() 方法，用于取消回调函数。requestAnimationFrame() 方法会返回一个 long 型的对象，用做标识回调函数身份的句柄（handle）。以后若要取消回调函数的执行，则可将其传给 cancelRequestAnimationFrame() 方法。

表 5-1 总结了 requestAnimationFrame() 与 cancelRequestAnimationFrame() 方法的用法。

警告：各浏览器对 requestAnimationFrame() 方法的特定实现

requestAnimationFrame() 方法定义在 W3C 的“基于脚本动画的定时控制”（Timing control for script-based animations）这一规范书中，详情参阅：http://webstuff.nfshost.com/anim-timing/Overview.html[⊖]。）

本书付印时，此规范书出现在 HTML5 规范中的时间还不太长，所以当时各个浏览器都仅支持其自己对 requestAnimationFrame() 方法的特定实现。下表总结了这些方法。

浏览器	方　法
Chrome 10	window.webkitRequestAnimationFrame(FrameRequestCallback callback, Element element)
Firefox (Gecko) 4.0 (2.0)	window.mozRequestAnimationFrame(FrameRequestCallback callback)
Internet Explorer 10, Platform Preview 2	Window.msRequestAnimationFrame(FrameRequestCallback callback)

Chrome 与 IE 浏览器还分别提供了 webkitCancelAnimationFrame() 与 msCancelAnimationFrame() 方法，用于取消动画回调函数的执行。

由于 W3C 的 requestAnimationFrame() 与 cancelRequestAnimationFrame() 方法尚未获得广泛支持[⊖]，所以不要直接调用这些规范中定义的标准方法，而应该通过“polyfill 方法”间接地调用。5.1.3 小节将讲述如何实现这种可以兼容多个浏览器的“polyfill 方法”。

⊖ 该规范的官方网址是：http://www.w3.org/TR/animation-timing/。——译者注

⊖ 各大浏览器的最新版本已经提供了对动画API的支持，详情参阅：http://html5test.com/compare/feature/animation-requestAnimationFrame.html。——译者注

提示：requestAnimationFrame() 方法传递给回调函数的时间值

一般来说，动画都是基于时间的，所以 requestAnimationFrame() 方法在回调动画函数时，会传递给它一个时间值，该值表示从 1970 年 1 月 1 日到当前所经过的毫秒数。

提示：若你对 requestAnimationFrame() 方法的细节不感兴趣，可以跳至后续小节

如果读者对 requestAnimationFrame() 方法背后的故事及各浏览器对该方法的特定实现细节不感兴趣，那么请直接跳至 5.1.3 小节，那里将会讲述如何以“polyfill 式方法”来间接地实现 requestAnimationFrame() 的功能。

表 5-1　W3C 标准中的 requestAnimationFrame() 与 cancelRequestAnimationFrame() 方法

方　法	描　述
long window.requestAnimationFrame(FrameRequestCallback callback)	请求浏览器在绘制下一帧动画时调用指定的回调函数。若要取消回调，可将该方法所返回的句柄传递给 cancelRequestAnimationFrame() 方法
void window.cancelRequestAnimationFrame(long handle)	将原来以 requestAnimationFrame() 方法所注册的回调函数取消执行。必须在浏览器还未执行回调函数时才能调用此方法

5.1.1.1　Firefox 浏览器对 requestAnimationFrame() 功能的实现

Firefox 浏览器在 4.0 版本中首次提供了该浏览器特定的 requestAnimationFrame() 方法变种，叫做 mozRequestAnimationFrame()。此方法的使用与 requestAnimationFrame() 一样，如程序清单 5-5 所示。

程序清单 5-5　在 Firefox 浏览器中反复执行动画回调函数

```
function animate(time) {
   // Update and draw animation objects

   window.mozRequestAnimationFrame(animate);
}

window.mozRequestAnimationFrame(animate);
```

Firefox 所实现的 mozRequestAnimationFrame() 方法遵循如下规则：

- Firefox 调用动画回调函数的最大频率是每秒钟 60 次。
- 当动画所在分页不可见时，Firefox 对动画回调函数的调用频率不超过每秒 1 次。
- Firefox 调用回调函数的速度不会高于渲染页面的速度。

与 requestAnimationFrame() 方法一样，Firefox 也会将绘制下一帧动画的时间传递给动画回调函数。

警告：Firefox 4.0 版本的 window.mozRequestAnimationFrame() 方法有 bug

Firefox 4.0 版本的一个 bug 导致使用 window.mozRequestAnimationFrame() 所做的动画，其帧速率大约只能达到每秒 30 ～ 40 帧。该 bug 在 Firefox 5.0 版本中已修复。如果你想让动画运行在 Firefox 4.0 浏览器上，请不要使用 mozRequestAnimationFrame()。

表 5-2 列出了 mozRequestAnimationFrame() 方法的用法。

表 5-2 Firefox 浏览器提供的 mozRequestAnimationFrame() 方法

方 法	描 述
window.mozRequestAnimationFrame (FrameRequestCallback)	请求浏览器在绘制下一帧动画时调用指定的回调函数。该回调参数是可选的，因为 Firefox 允许开发者通过向 window 对象增加事件监听器来指定回调函数

5.1.1.2 Chrome 浏览器对 requestAnimationFrame() 功能的实现

Chrome 浏览器也采用了与 Firefox 浏览器相同的回调模型，它将这个函数叫做 window.webkitRequestAnimationFrame()。该函数的使用与 Firefox 所提供的 window.mozRequestAnimationFrame() 是一样的，如程序清单 5-6 所示。

Chrome 浏览器在调用动画回调函数时所遵循的规则与 Firefox 基本一致：

- Chrome 调用动画回调函数的最大频率是每秒钟 60 次。
- 当动画所在分页不可见时，Chrome 就不再调用动画回调函数了，直到该分页再次可见为止。
- Chrome 调用回调函数的速度不会高于渲染页面的速度。

程序清单 5-6 在 Chrome 浏览器中反复执行动画回调函数

```
function animate(time) {
   if (time == undefined)
      time = +new Date();

   // Update and draw animation objects

   window.webkitRequestAnimationFrame(animate);
}
...

window.webkitRequestAnimationFrame(animate);
```

表 5-3 总结了 webkitRequestAnimationFrame() 与 webkitCancelAnimationFrame() 方法的用法。

表 5-3 Chrome 浏览器提供的 webkitRequestAnimationFrame() 方法

方 法	描 述
long window.webkitRequestAnimationFrame(FrameRequestCallback callback, optional Element element)	请求浏览器在绘制下一帧动画时，调用指定的回调函数。若要取消下一次的动画帧绘制，则该函数所返回的句柄传递给 webkitRequestAnimationFrame() 方法 这个可选的 element 参数，没有出现在 Firefox 版本的实现中，它用来指定播放动画的元素。如果该元素不可见，那么 Chrome 就不会调用回调函数
void window.webkitRequestAnimationFrame(long handle)	该方法用于取消早前通过 webkitRequestAnimationFrame() 方法所注册的回调函数。必须在浏览器执行回调函数之前调用此方法

提示：如何使用 webkitRequestAnimationFrame() 方法的可选 element 参数

webkitRequestAnimationFrame() 的第二个参数是可选的，它是一个指向 HTML 元素的引用。如果该元素不可见，那么 webkitRequestAnimationFrame() 方法就不会调用动画回调函数了。通常情况下，可以将运行动画效果的 canvas 元素引用传递给 webkitRequestAnimationFrame()。

警告：Chrome 10 版本的浏览器传递给回调函数的 time 参数值有 bug

与 Firefox 浏览器一样，Chrome 也会将绘制下一帧动画的时间传递给动画回调函数。然而，在首次实现 webkitRequestAnimationFrame() 方法的 Chrome 10 版本中，浏览器没有将绘制的时间传递给动画回调函数，所以导致回调函数中的 time 变量的值成了 undefined。在这种情况下，可以像程序清单 5-6 那样自行对该参数赋值。

5.1.2　Internet Explorer 浏览器对 requestAnimationFrame() 功能的实现

从 Internet Explorer 10、Platform Preview 2 版本开始，Microsoft 提供了两个和 W3C 标准相似的方法，分别叫做 msRequestAnimationFrame() 与 msCancelRequestAnimationFrame()。程序清单 5-7 演示了 msRequestAnimationFrame() 方法的用法。

程序清单 5-7　在 Internet Explorer 浏览器中反复执行动画回调函数

```
function animate(time) {
   // Update and draw animation objects

   window.msRequestAnimationFrame(animate);
}
window.msRequestAnimationFrame(animate);
```

5.1.3　可移植于各浏览器平台的动画循环逻辑

在 5.1.1 小节中我们说过，开发者可以使用，而且也应该使用 W3C 标准所定义的 requestAnimationFrame() 方法来制作动画。

然而，在所有浏览器都支持该方法之前，你需要通过每个浏览器各自提供的特定方法来实现动画播放功能，如果浏览器连这样的方法都没有提供的话，那么就需要写一段默认的实现代码，让浏览器能借此来播放动画。

我们来研究一种可以在各浏览器之间移植的动画循环逻辑：该方案如果发现当前浏览器支持 W3C 标准的实现方法，那么就使用它，否则，就使用浏览器专属的实现方式。如果浏览器既不支持 W3C 的标准实现，又不提供专属的实现方式，那么该方案就会借助 setTimeout() 方法来实现一段每秒 60 帧的动画播放代码。

这个可移植的动画解决方案叫做 window.requestNextAnimationFrame()（注意，此方法名比 W3C 标准的方法名多了一个“Next”），它的用法与 webkitRequestAnimationFrame()、mozRequestAnimationFrame() 及 requestAnimationFrame() 方法相似：

```
function animate(time) {
   // Update and draw animation objects

   window.requestNextAnimationFrame(animate);
}

window.requestNextAnimationFrame(animate);
```

我们先试试能不能按照如下方式来实现这个 requestNextAnimationFrame() 方法：

```
window.requestNextAnimationFrame =
   (function () {
      return window.requestAnimationFrame    ||
         window.webkitRequestAnimationFrame ||
         window.mozRequestAnimationFrame    ||
```

```
            window.msRequestAnimationFrame      ||

            function (callback, element) { // Assume element is visible
               var self = this,
                   start,
                   finish;

               window.setTimeout( function () {
                  start = +new Date();
                  callback(start);
                  finish = +new Date();

                  self.timeout = 1000 / 60 - (finish - start);

               }, self.timeout);
            };
         }
      )
   ();
```

上述代码将一个函数对象的值赋给了 window 对象的 requestNextAnimationFrame 属性。这个函数对象是对另外一个匿名函数表达式[㊀] 进行估值操作[㊁] 所产生的返回值，该技巧在 JavaScript 语言编程中叫做“自执行函数”[㊂]。

如果可以使用 W3C 标准的方法或是浏览器专属的实现方式，那么上述代码就将这个标准的或专属的实现函数赋值给 window.requestNextAnimationFrame()。

如果当前浏览器既不支持 W3C 标准，又不提供专属实现，那么就利用 window.setTimeout() 方法实现一段代码，让浏览器能够以大约每秒 60 帧的速度来播放动画，然后再将其赋值给 code.requestNextAnimationFrame() 使用。

上面这种 window.requestNextAnimationFrame() 的实现方式在大多数情况下都可以正常运行，不过它存在两个问题：

（1）如果在 Chrome 10 版本的浏览器中运行，则 animate() 函数被回调时所传入的 time 参数值会是 undefined。假如你要实现 5.6 节中所介绍的“基于时间的运动”，那么在更新动画帧的回调函数中所出现的 undefined 值，将会干扰动画的播放。

（2）我们在 5.1.1.1 小节中提过，Firefox 4.0 版本的浏览器所实现的 window.mozRequestAnimationFrame() 方法有个 bug，导致大多数动画的帧速率被局限在每秒 30 ～ 40 帧的范围内，这样的话，很多动画播放起来会慢得让人难以忍受。如果打算在 Firefox 4.0 上播放动画，那就必须解决该问题。

程序清单 5-8 列出了最终版本的 window.requestNextAnimationFrame() 方法，这次所写的代码解决了上述两个问题。如果动画在 Chrome 10 版本的浏览器中播放，那么我们会把 window.webkitRequestAnimationFrame() 封装在另外一个函数中，使得动画更新函数在被回调的时候，能够获得有效的 time 参数值。如果动画运行在 Firefox 4.0 版本的浏览器中，那么这段代码会用默认的 setTimeout() 实现方式来替换掉浏览器所提供的 window.mozRequestAnimationFrame() 方法。

㊀ 指的是从代码第二行开始，到倒数第二行为止，以“(function () {...})”形式所定义的匿名函数表达式。——译者注

㊁ 代码最后一行的“()”表示对匿名函数表达式进行估值操作。——译者注

㊂ 该技巧的具体细节可参看：http://stackoverflow.com/questions/1634268/explain-javascripts-encapsulated-anonymous-function-syntax。——译者注

提示："Polyfill 式方法"

Polyfill 这个词是由 polymorphically 与 backfill 这两个词合成的。与面向对象语言中的"多态"（polymorphism）相似，"polyfill 式方法"[⊖]在运行时会根据具体情况来执行不同的代码。这种"Polyfill 式方法"也可以给尚未支持某个特定规范的浏览器提供一份预备的功能代码（backfill functionality）。比方说，本小节所说的 requestNextAnimationFrame() 就是一个"polyfill 式方法"，它可以根据浏览器对 requestAnimationFrame() 的支持情况，在程序运行时自行决定将要执行的代码。如果当前浏览器不支持"基于脚本动画的时间控制"规范，那么它将采用借助 setTimeout() 方法所实现的那段预备代码来绘制动画。

"Polyfill 式方法"代表了一种重要的思维方式，它与传统的想法不同：过去开发跨平台软件时，我们的程序只会考虑那些各平台都支持的功能[⊜]，而现在这种"Polyfill 式方法"，则会优先考虑利用每个平台所提供的先进特性，仅在该平台不支持这种特性时，才会使用预备代码以应对需求。

程序清单 5-8　用于实现 W3C 规范所定义 requestAnimationFrame() 功能的"Polyfill 式方法"

```
window.requestNextAnimationFrame =
   (function () {
      var originalWebkitMethod,
          wrapper = undefined,
          callback = undefined,
          geckoVersion = 0,
          userAgent = navigator.userAgent,
          index = 0,
          self = this;

      // Workaround for Chrome 10 bug where Chrome
      // does not pass the time to the animation function

      if (window.webkitRequestAnimationFrame) {
          // Define the wrapper

          wrapper = function (time) {
            if (time === undefined) {
               time = +new Date();
            }
            self.callback(time);
          };

          // Make the switch

         originalWebkitMethod = window.webkitRequestAnimationFrame;

         window.webkitRequestAnimationFrame =
         function (callback, element) {
            self.callback = callback;

            // Browser calls wrapper; wrapper calls callback

            originalWebkitMethod(wrapper, element);
         }
      }

      // Workaround for Gecko 2.0, which has a bug in
      // mozRequestAnimationFrame() that restricts animations
      // to 30-40 fps.
```

⊖ polyfill method，此词尚未有中文译名，其大意为"多平台智能式适配方法"。——译者注

⊜ programming to the lowest common denominator，字面意思为"针对最小公分母来编程"，它用于比喻"在编程时只支持各平台的共性而放弃其特性"的做法。——译者注

```
        if (window.mozRequestAnimationFrame) {
           // Check the Gecko version. Gecko is used by browsers
           // other than Firefox. Gecko 2.0 corresponds to
           // Firefox 4.0.
           index = userAgent.indexOf('rv:');

           if (userAgent.indexOf('Gecko') !=  -1) {
              geckoVersion = userAgent.substr(index + 3, 3);

              if (geckoVersion === '2.0') {
                 // Forces the return statement to fall through
                 // to the setTimeout() function.

                 window.mozRequestAnimationFrame = undefined;
              }
           }
        }

        return window.requestAnimationFrame   ||
           window.webkitRequestAnimationFrame ||
           window.mozRequestAnimationFrame    ||
           window.oRequestAnimationFrame      ||
           window.msRequestAnimationFrame     ||

           function (callback, element) {
              var start,
                  finish;

              window.setTimeout( function () {
                 start = +new Date();
                 callback(start);
                 finish = +new Date();

                 self.timeout = 1000 / 60 - (finish - start);

              }, self.timeout);
           };
        }
     )
  ();
```

图 5-2 所示应用程序使用了程序清单 5-8 中名为 requestNextAnimationFrame() 的函数。

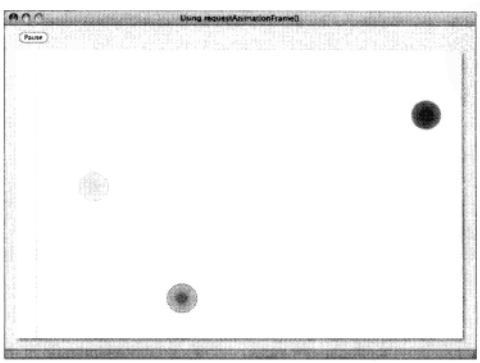


图 5-2　使用名为 requestNextAnimationFrame() 的“polyfill 式方法”来绘制动画

图 5-2 中应用程序绘制了三个运动的圆盘，它与图 5-1 所示应用程序类似，不过，图 5-2 这个程序在绘制每帧动画之前会先把背景擦掉。

程序清单 5-9 与程序清单 5-10 分别列出了图 5-2 所示应用程序的 HTML 与 JavaScript 代码。请注意，HTML 页面的代码引入了一个名叫 requestNextAnimationFrame.js 的 JavaScript 程序文件，该文件包含程序清单 5-8 所列“polyfill 式方法”requestNextAnimationFrame() 的实现代码。

应用程序的大部分代码都用来定义及绘制圆盘，所有与动画有关的代码，都被封装在 animate() 方法以及程序清单尾部“Animate”按钮的 onclick 事件处理器之中。

当用户点击“Animate”按钮时，click 事件处理器就会调用 requestNextAnimationFrame() 方法来播放动画，在调用时还会将指向 animate() 函数的引用传递给它。而这个 animate() 函数，则会先把 canvas 的图像擦除，然后绘制下一帧动画，最后再次调用 requestNextAnimationFrame() 方法，使动画能够继续播放下去。

程序清单 5-9　使用 requestAnimationFrame() 方法来绘制动画（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Using requestAnimationFrame()</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            background: #ffffff;
            cursor: pointer;
            margin-left: 10px;
            margin-top: 10px;
            -webkit-box-shadow: 3px 3px 6px rgba(0,0,0,0.5);
            -moz-box-shadow: 3px 3px 6px rgba(0,0,0,0.5);
            box-shadow: 3px 3px 6px rgba(0,0,0,0.5);
         }

         #controls  {
            margin-top: 10px;
            margin-left: 15px;
         }
      </style>
   </head>

   <body>
      <div id='controls'>
         <input id='animateButton' type='button' value='Animate'/>
      </div>

      <canvas id='canvas' width='750' height='500'>
         Canvas not supported
      </canvas>

      <script  src='requestNextAnimationFrame.js'></script>
      <script  src='example.js'></script>
   </body>
</html>
```

程序清单 5-10 使用 requestAnimationFrame() 方法来绘制动画（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    paused = true,
    discs = [
      {
         x: 150,
         y: 250,
         lastX: 150,
         lastY: 250,
         velocityX: -3.2,
         velocityY: 3.5,
         radius: 25,
         innerColor: 'rgba(255,255,0,1)',
         middleColor: 'rgba(255,255,0,0.7)',
         outerColor: 'rgba(255,255,0,0.5)',
         strokeStyle: 'gray',
      },

      {
         x: 50,
         y: 150,
         lastX: 50,
         lastY: 150,
         velocityX: 2.2,
         velocityY: 2.5,
         radius: 25,
         innerColor: 'rgba(100,145,230,1.0)',
         middleColor: 'rgba(100,145,230,0.7)',
         outerColor: 'rgba(100,145,230,0.5)',
         strokeStyle: 'blue'
      },

      {
         x: 150,
         y: 75,
         lastX: 150,
         lastY: 75,
         velocityX: 1.2,
         velocityY: 1.5,
         radius: 25,
         innerColor: 'rgba(255,0,0,1.0)',
         middleColor: 'rgba(255,0,0,0.7)',
         outerColor: 'rgba(255,0,0,0.5)',
         strokeStyle:'orange'
      },
   ],

       numDiscs = discs.length,
       animateButton = document.getElementById('animateButton');

    // Functions.......................................................

    function drawBackground() {
       // Listing omitted for brevity See Example 3.1
       // for a complete listing
    }

    function update() {
       var disc = null;
```

```
        for(var i=0; i < numDiscs; ++i)  {
           disc = discs[i];

           if (disc.x + disc.velocityX + disc.radius >
                  context.canvas.width ||
                  disc.x + disc.velocityX - disc.radius < 0)
              disc.velocityX = -disc.velocityX;

           if (disc.y + disc.velocityY + disc.radius >
                  context.canvas.height ||
                  disc.y + disc.velocityY - disc.radius < 0)
              disc.velocityY= -disc.velocityY;

           disc.x += disc.velocityX;
           disc.y += disc.velocityY;
        }
     }

     function draw() {
        var disc = discs[i];

        for(var i=0; i < numDiscs; ++i) {
           disc = discs[i];

           gradient = context.createRadialGradient(disc.x, disc.y, 0,
                              disc.x, disc.y, disc.radius);
           gradient.addColorStop(0.3, disc.innerColor);
           gradient.addColorStop(0.5, disc.middleColor);
           gradient.addColorStop(1.0, disc.outerColor);

        context.save();
        context.beginPath();
        context.arc(disc.x, disc.y, disc.radius, 0, Math.PI*2, false);
        context.fillStyle = gradient;
        context.strokeStyle = disc.strokeStyle;
        context.fill();
        context.stroke();
        context.restore();
     }
}

// Animation.......................................................

function animate(time) {
    if (!paused) {
       context.clearRect(0,0,canvas.width,canvas.height);
       drawBackground();
       update();
       draw();

       window.requestNextAnimationFrame(animate);
    }
}

// Event handlers..................................................

animateButton.onclick = function (e) {
    paused = paused ? false : true;
    if (paused) {
```

```
        animateButton.value = 'Animate';
    }
    else {
        window.requestNextAnimationFrame(animate);
        animateButton.value = 'Pause';
    }
};

// Initialization.....................................................

context.font = '48px Helvetica';
```

5.2 帧速率的计算

动画是由一系列叫做“帧”（frame）的图像组成的，这些图像的显示频率就叫做“帧速率”（frame rate）。通常来说，有必要计算一下帧速率。比如，在实现 5.6 节中所述“基于时间的运动”效果时，可能会用到动画的帧速率，或是有时为了保证动画能够播放得足够流畅，我们也需要知道动画的帧速率。

图 5-3 所示应用程序可以算出动画的帧速率，并将它显示在 canvas 之中。

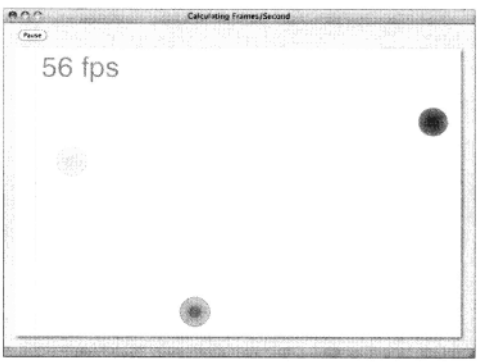


图 5-3 计算动画每秒钟播放的帧数

用于计算帧速率的代码列在了程序清单 5-11 之中，该清单还列出了应用程序实现动画循环的代码。该应用程序使用了一个简单的等式，根据当前帧距离上一帧的时间，计算出动画每秒钟播放的帧数（frame per second，简称 fps）。

应用程序的代码将上次绘制动画帧的时间从当前时间中减去，得到了这两帧动画的时间差，然后再用 1000 除以这个以毫秒为单位的时间差，于是就得出了动画每秒钟播放的帧数，也就是其帧速率。

程序清单 5-11 帧速率的计算

```
var lastTime = 0;

function calculateFps() {
    var now = (+new Date),
        fps = 1000 / (now - lastTime);
    lastTime = now;
```

```
    return fps;
}

function animate(time)  {
    eraseBackground();
    drawBackground();
    update();
    draw();

    context.fillStyle = 'cornflowerblue';
    context.fillText(calculateFps().toFixed() + ' fps', 20, 60);

    window.requestNextAnimationFrame(animate);
}

window.requestNextAnimationFrame(animate);
```

提示："3D Monster Maze"游戏的运行速度是每秒 6 帧

个人电脑上的首款三维第一人称射击游戏（3D first-person shooter game）是 1981 年发布的"3D Monster Maze"[㊀]，该游戏运行于 Sinclair ZX81 平台[㊁]。它的运行速度大约为每秒 6 帧。

5.3　以不同的帧速率来执行各种任务

很多动画程序在播放动画时还要执行其他任务。比方说，在播放动画时可能还要显示剧情文本，播放音乐，或是更新游戏分数。此类任务大都不需要以每秒钟 60 帧的速度执行，所以说，要学会把不同的任务安排在不同的帧速率上执行，这一点很重要。

程序清单 5-12 列出了范例程序的动画循环，该循环会根据上一次更新 fps 数值的时间来判断当前是否已经过了一秒钟，如果是的话，那么就再次更新 fps 数值[㊂]。

程序清单 5-12　以不同的帧速率来执行不同的任务

```
var lastFpsUpdateTime = 0,
    lastFpsUpdate = 0;

function animate(time) {
    var fps = 0;

    if (time == undefined) {
        time = +new Date;
    }

    if (!paused) {
        eraseBackground();
        drawBackground();
        update(time);
        draw();

        fps = calculateFps();

        // Once per second, update the frame rate
```

㊀ 游戏名称意为"三维怪物迷宫"，详情参见：http://en.wikipedia.org/wiki/3D_Monster_Maze。——译者注

㊁ 这款家用电脑的详情请参见：http://en.wikipedia.org/wiki/Sinclair_ZX81。——译者注

㊂ 这也就意味着，不论动画的帧速率是多少，"更新动画帧速率数值"的这个任务，其运行速度总是每秒1帧，不必与动画播放的速度一致。——译者注

```
        if (now - lastFpsUpdateTime > 1000) {
           lastFpsUpdateTime = now;
           lastFpsUpdate = fps;
        }

        context.fillStyle = 'cornflowerblue';
        context.fillText(lastFpsUpdate.toFixed() + ' fps', 50, 48);
    }
}
```

5.4 恢复动画背景

实现动画效果所用的大多数技术都比较简单，例如借助 requestAnimationFrame()，每隔一段时间就调用一次自定义的 animate() 方法，还有就是根据动画内容计算出运动物体下一次的位置，并将其绘制到新坐标处，这个做起来也很容易。绘制动画时具有挑战性的环节在于如何处理背景。从本质上讲，无外乎这三种办法：

- 将所有内容都擦除，并重新绘制。
- 仅重绘内容发生变化的那部分区域。
- 从离屏缓冲区中将内容发生变化的那部分背景图像复制到屏幕上。

将所有内容都擦除并重新绘制，这是最直截了当的办法。如果仅重绘内容发生变化的区域，那么还是得擦除并重画背景，只不过执行操作的范围仅局限于屏幕上图像有变化的那块区域。最后，我们还有第三种选择，那就是从离屏缓冲区中将内容发生变化的那部分背景图像复制到屏幕上（这也叫做“图块复制”，blitting）。

“擦除全部内容并重绘”这个办法，在前面的章节中已经讲过了，现在咱们专门来研究怎样借助剪辑区域及“图块复制”技术来实现动画。

小技巧：是否要把每帧动画的所有内容都重绘一遍？

虽说听上去有些反常，但有时把每帧动画的所有内容都重绘一遍，反倒可以获得最佳性能。通常来说，如果背景很简单，而且你要绘制的运动物体也比较简单的话，那么，将所有内容都擦掉并且重新绘制，也许是个好办法。

5.4.1 利用剪辑区域来处理动画背景

如果背景图像很简单，那么先擦掉背景然后再重绘下一帧动画这个办法效果还不错。不过，如果背景很复杂，那么在画每帧之前都重绘背景，其耗时就太长了。在这种情况下，你可以考虑将重绘操作限定在 canvas 中的某个特定区域内。

在 2.15 节中讲过，利用剪辑区域，可以将所有的绘制操作都限定在由某条路径所定义的范围内。设置好剪辑区域后，接下来执行的绘制命令就只会在该区域内生效。

图 5-4 所示动画程序，在绘制圆盘的同时，利用剪辑区域来恢复背景图像。通常我们会在程序刚启动时绘制整个背景图像，然后在绘制圆盘动画的同时，利用剪辑区域来恢复被上一帧动画所破坏的背景。不过在本例中，应用程序一开始并未绘制背景图，这样的话，你就能看到该程序是如何利用剪辑区域技术来填充圆盘在播放上一帧动画时所占据的那块背景了。

下面列出了利用剪辑区域技术来恢复上一帧动画所占背景图像的执行步骤。

（1）调用 context.save()，保存屏幕 canvas 的状态。

（2）通过调用 beginPath() 来开始一段新的路径。
（3）在 context 对象上调用 arc()、rect() 等方法来设置路径。
（4）调用 context.clip() 方法，将当前路径设置为屏幕 canvas 的剪辑区域。
（5）擦除屏幕 canvas 中的图像（实际上只会擦除剪辑区域所在的这一块范围）。
（6）将背景图像绘制到屏幕 canvas 中（绘制操作实际上只会影响剪辑区域所在范围）。
（7）恢复屏幕 canvas 的状态参数，该操作主要是为了重置剪辑区域。

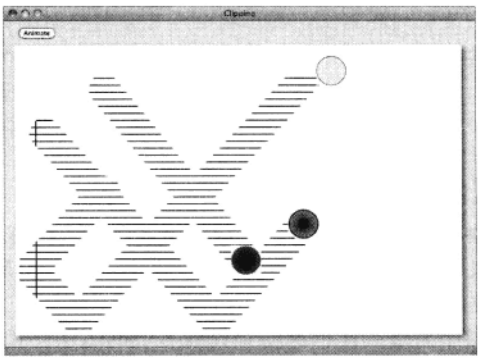


图 5-4　利用剪辑区域技术来恢复动画背景

图 5-4 所示应用程序通过如下代码实现了上述步骤：

```
function draw() {
      var numDiscs,
          disc,
          i;

      for(i=0; i < numDiscs; ++i) {
         drawDiscBackground(discs[i]);
      }
      ...

      for(i=0; i < numDiscs; ++i) {
         drawDisc(discs[i]);
      }
      ...
 }
function drawDiscBackground(disc) {
    context.save();

    context.beginPath();
    context.arc(disc.lastX, disc.lastY,
                disc.radius+1, 0, Math.PI*2, false);
    context.clip();

    eraseBackground();
    drawBackground();

    context.restore();
 }
```

draw() 方法会将圆盘上一次所占据位置的背景图像恢复，并重新将其绘制在当前位置上。

drawDiscBackground() 方法在绘制圆盘上一帧所据位置的背景图时，先将剪辑区域设置为描述圆盘在上一帧时所占位置的那条路径，然后再于该剪辑区域内擦除并重绘背景。然后，此方法会恢复绘图环境对象的状态参数，于是也就将剪辑区域复原了。如果不恢复剪辑区域的话，那么后续对 context.clip() 方法的调用就会将剪辑区域设置为当前剪辑区域和当前路径的交集部分，这会导致剪辑区域迅速变小，最后消失。这样的话，所有与图形有关的操作就都不起作用了。

小技巧：利用剪辑区域有时能提高绘制速度，有时则不能

将背景的一小部分区域绘制到屏幕上，要比把全部背景都绘制上去要快得多。因此，如果所绘物体不多，那么利用剪辑区域技术要比直接重绘全部背景要好。然而，如果动画中的物体变得很多，那么这种做法也会相应地增加每帧动画需要绘制背景的次数。多次背景绘制操作所耗的时间会令你吃不消，利用剪辑区域技术绘制少量物体时所带来的性能优势将不复存在。

5.4.2 利用图块复制技术来处理动画背景

在前一小节中，我们讲解了如何利用剪辑区域技术来避免在每帧动画时都要重绘整个背景。虽说剪辑区域可以将绘制操作局限在 canvas 中的某个范围内执行，但是你毕竟还是得在播放每帧动画时都执行一遍重绘全部背景的操作。

另外一种办法则是将整个背景一次性地绘制到离屏 canvas 中，稍后从离屏 canvas 中只将修复动画背景所需的那一块图像复制到屏幕上即可。

将离屏 canvas 中的背景图块复制到屏幕上，同样需要按照上一小节所讲的 7 个步骤来做。但是第 6 步有所不同，原来的操作是将背景绘制到由剪辑区域所定义的那一小块范围内，而现在则改为将修复背景所需图块从离屏 canvas 中复制到屏幕上。其代码如下：

```
function drawDiscBackground(context, disc) {
    var x = disc.lastX,
        y = disc.lastY,
        r = disc.radius,
        w = r*2,
        h = r*2;

    context.save();

    context.beginPath();
    context.arc(x, y, r+1, 0, Math.PI*2, false);
    context.clip();

    context.clearRect(0, 0, canvas.width, canvas.height);
    context.drawImage(offscreenCanvas,
                      x-r, y-r, w, h, x-r, y-r, w, h);
    context.restore();
}
```

小技巧：剪辑区域和图块复制技术的优点与缺点

剪辑区域和图块复制技术都可以修复被上一帧动画所破坏的那部分背景，而不需要重绘整个背景图像。使用剪辑区域是为了在待修复的那块区域内重绘背景，而图块复制则是将待修复的那块图像从离屏 canvas 中拷贝到屏幕上。一般来说，图块复制要比使用剪辑区域的速度快，然而它需要一个离屏 canvas，这会占据更多的内存。

5.5 利用双缓冲技术绘制动画

到目前为止，本章所用的动画逻辑都是下面这个样子的：

```
var canvas = document.getElementById('canvas'),
   context = canvas.getContext('2d'),
    ...
function animate(time) {
    context.clearRect(0, 0, canvas.width, canvas.height);
   // Update and draw animation objects...
   requestNextAnimationFrame(time); // Keep the animation going
}
   requestNextAnimationFrame(time); // Start the animation
```

上述代码首先清除 canvas，然后绘制下一帧动画。假如动画是单缓冲的（single buffered），那么就意味着其内容会被立刻绘制到屏幕 canvas 中。这样的话，擦除背景的那一瞬间所造成的空白可能会使动画看起来有些闪烁。

防止闪烁的一种办法就是使用双缓冲（double buffering）。如果用双缓冲，那么就不是将动画内容直接绘制到屏幕 canvas 中了，而是先将所有东西都绘制到离屏 canvas 里面，然后把该 canvas 的全部内容一次性地复制到屏幕 canvas 中。程序清单 5-13 演示了这种做法。

程序清单 5-13　利用双缓冲技术绘制动画

```
// For illustration only. Do not do this.

var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    // Create an offscreen canvas

    offscreenCanvas = document.createElement('canvas'),
    offscreenContext = offscreenCanvas.getContext('2d'),
    ...
offscreenCanvas.width = canvas.width;
offscreenCanvas.height = canvas.height;

function animate(now)  {
   offscreenContext.clearRect(
      0, 0, offscreenCanvas.width, offscreenCanvas.height);

      // Update and draw animation objects into the offscreen canvas...

      // Clear the onscreen canvas and draw the offscreen
      // into the onscreen canvas

      context.clearRect(0, 0, canvas.width, canvas.height);
      context.drawImage(offscreenCanvas, 0, 0);
}
```

双缓冲技术可以有效地消除动画绘制时的闪烁，所以浏览器会自动采用双缓冲来实现 canvas 元素。开发者并不需要自己来实现它。而且如果按照程序清单 5-13 所示方法来手工实现双缓冲的话，反倒会降低动画的绘制效率。在绘制每帧动画时，都花时间把离屏缓冲 canvas 的内容复制到屏幕上，这样做是毫无益处的，因为浏览器已经内建了对双缓冲技术的支持。

如果你曾经在调试器中单步执行过与 Canvas 有关的代码，那么可能会怀疑浏览器到底有没

有自动运用双缓冲技术来绘制 canvas 元素。毕竟我们在调试器中单步执行代码的时候，对 Canvas API 的调用是立刻就生效的。然而，调试器是在另外一个线程中运行的，所以，尽管看上去调用 Canvas API 似乎都能立即生效，但实际上，它们还是通过双缓冲机制来运作的。

可以通过如下代码来验证，浏览器确实是使用双缓冲技术来绘制 canvas 元素的：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    sum = 0,
    ...

function animate(now) {
   eraseBackground(); // Erase the onscreen canvas

   // Erased background, starting busy work

   for (var i=0; i < 500000; ++i)  {
      sum += i;
   }

   // Done with busy work

   drawBackground();  // Draw the background into onscreen canvas
   draw();            // Draw animation objects into onscreen canvas
   requestNextAnimationFrame(time); // Keep the animation going
}

requestNextAnimationFrame(time); // Start the animation
```

上述代码在擦除 canvas 背景之后，执行了一个用于模拟繁忙操作任务的循环。

如果 canvas 没有使用双缓冲，那么就意味着对 Canvas API 的调用会立即生效。这样的话，上述代码将会频繁地显示出空白的 canvas 画面来。因为它在擦除 canvas 之后，执行了一项繁忙的任务，而这项任务所花的时间是我们可以察觉到的。在花时间执行完任务后，这段代码继续绘制背景及圆盘，并重复整个循环过程。然而实际上，刚才假设的情况并未发生，我们根本就没有看到空白的 canvas 画面。擦除 canvas 的操作不会立即生效，因为这个调用与其余对 Canvas API 的调用一样，都会被浏览器纳入双缓冲机制中。所以说，浏览器并没有在执行那项耗时的任务之前就擦除背景图像，而稍后它将离屏缓冲区的内容复制到屏幕上时，才算真正执行了擦除操作。

警告：浏览器会自动对 Canvas 使用双缓冲，无需手工干预

各大浏览器厂商在实现 canvas 元素的时候，都使用了双缓冲机制，所以开发者自己再去实现它就会适得其反。如果手工实现它，那么将离屏缓冲区复制到屏幕上的这个操作反而会降低性能。由于浏览器已经在 canvas 中内建了双缓冲机制，所以开发者自己再去实现一遍，是没有任何好处的。

这并不是说我们不应该使用多缓冲区技术了。在 5.4.2 小节中已经演示过，将离屏背景图缓冲区的某个图块复制到屏幕上，可以提高绘制复杂背景图像的效率。然而，传统的双缓冲技术，也就是那种“将所有内容都绘制到离屏 canvas 中，稍后再将其复制到屏幕上”的办法，对于 canvas 元素来说，不但无益，反而有害。

5.6 基于时间的运动

假设在多人射击游戏中有两个玩家各自沿着某条走廊前行，如果保持行进速度不变，那么他们将同时到达走廊的交汇处。若是由于某玩家的电脑配置比另一人好得多，而导致其电脑播放游

戏动画的速度更快，那么玩家们就不会再和比自己电脑配置高的人玩了，因为他们总是能提前到达目的地。

不论底层的帧速率如何，动画都应以稳定的速度播放才对。正如图 5-5 与图 5-6 所演示的那样，降低动画的帧速率其实并不难，同时播放多个动画无疑会降低它们的运行速度。

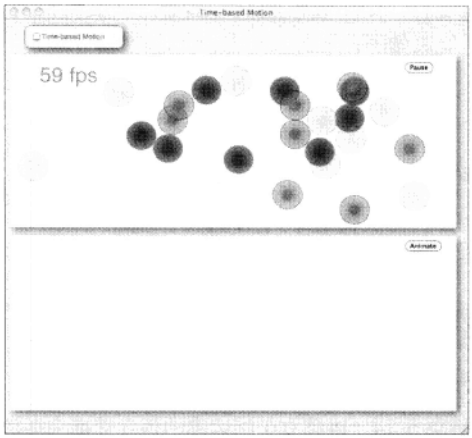


图 5-5　一幅动画正在以每秒 60 帧的速度播放

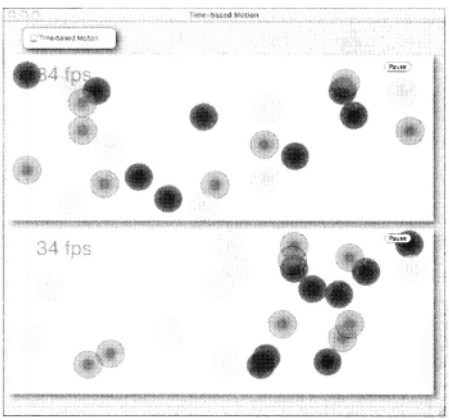


图 5-6　同时运行多个动画会降低帧速率

图 5-5 所示应用程序有两种模式。如果顶部的复选框被选中，那么该程序将会以基于时间的运动来控制动画播放速度，也就是说圆盘每秒钟所移动的像素数是固定的。若是未选中此复选框，那么应用程序就不会使用基于时间的运动来控制动画了，此时圆盘的运动速度将随着动画的帧速率一起变化。

使用“基于时间的运动”来控制动画，令所有圆盘均以相同速度来移动，有时候看上去效果并不好。如果动画每秒钟播放大约 30 帧，那么它就赶不上显示器的刷新频率了，因此动画的某些帧就会被省略（这种现象又叫做“掉帧”、“丢帧”，dropping frame）。用户会看到圆盘从一个地方

突然跳到另外一个地方，整个动画的播放不够连贯。

然而，如果以每秒 30 帧的速度来运行动画，那么不管使不使用“基于时间的运动”，都会发生掉帧现象，因而动画看上去总是断断续续的。所以说，在其他因素都相同的情况下，最好是用“基于时间的运动”使动画在所有情况下都能以相同的速度播放。

想让动画以稳定的速度运行，而不受帧速率的影响，那就要根据物体的速度计算出它在两帧之间所移动的像素数。计算公式如下：

$$\frac{像素}{帧} = \frac{像素}{秒} \times \frac{帧}{秒}$$

该公式也可写作如下形式：

$$\frac{像素}{帧} = \frac{像素}{秒} \times \frac{秒}{帧}$$

上述公式可以算出物体每一帧应该移动多少像素。如果用户启用了“基于时间的运动”模式，那么图 5-5 中的应用程序就会使用该公式算出在每一帧动画之中，每个圆盘所移动的像素数。这段代码列在程序清单 5-14 之中。

程序清单 5-14　以独立于帧速率的恒定速度来播放动画

```
function updateTimeBased(time) {
   var disc = null,
       elapsedTime = time - lastTime,

   for(var i=0; i < discs.length;  ++i) {
      disc = discs[i];
      deltaX = disc.velocityX

      deltaX = disc.velocityX * (elapsedTime / 1000);
      deltaY = disc.velocityY * (elapsedTime / 1000);

      if (disc.x + deltaX + disc.radius > topContext.canvas.width ||
             disc.x + deltaX - disc.radius < 0) {
         disc.velocityX = -disc.velocityX;
         deltaX = -deltaX;
      }

      if (disc.y + deltaY + disc.radius > topContext.canvas.height ||
             disc.y + deltaY - disc.radius < 0) {
         disc.velocityY= -disc.velocityY;
         deltaY =  -deltaY;
      }

      disc.x = disc.x + deltaX;
      disc.y = disc.y + deltaY;

      lastTime = time;
   }
}
```

应用程序的代码将圆盘每秒移动的像素数，也就是它的速度，乘以每一帧持续的秒数[⊖]，就得出了圆盘在每帧所移动的像素数了。无论帧速率如何，圆盘都会以这个速度在每帧中走过相同的像素数。

⊖ “每一帧持续的秒数”也就是“每秒所播帧数”（fps）的倒数。——译者注

5.7　背景的滚动

读者已经在本章中看到，播放动画时如何才能不影响到物体后面的静态背景。很多动画的背景本身也在移动，例如，图 5-7 中所示的应用程序，即以流动的云彩作为背景。你也可以实现一幅动画，将其用做某款横向卷轴（side-scroller）电子游戏的背景。

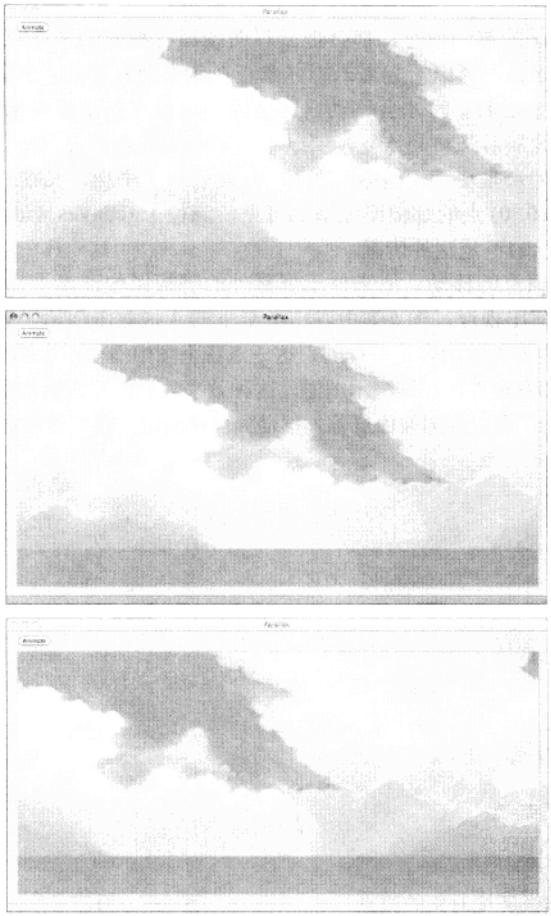


图 5-7　模拟流动的云彩：由上到下三张图演示了云彩从右至左的移动过程

图 5-7 所示应用程序通过移动 canvas 绘图环境对象的原点坐标来实现背景滚动效果，其代码如下：

```
var SKY_VELOCITY = 30, // 30 pixels/second
    skyOffset = 0;     // Translate by this offset
    ...

function draw() {
   skyOffset = skyOffset < canvas.width ?
```

```
                skyOffset + SKY_VELOCITY/fps : 0;

    context.save();
    context.translate(-skyOffset, 0);
    context.drawImage(sky, 0, 0);
    context.drawImage(sky, sky.width, 0);
    context.restore();
}
```

应用程序在绘制动画的每一帧时，都会把云彩图像画在相同的坐标点上，然而由于程序平移了绘图环境对象的原点坐标，所以看起来云彩好像正在从右向左移动，其效果如图 5-7 所示。

坐标系的原点只在绘制云彩时才会移动，这是因为该程序在平移坐标系之前先将绘图环境对象的状态保存起来，在绘制后又将其恢复，所以不会影响到其余部分。

该程序把表示天空的那张图画了两遍，在 (0, 0) 点绘制了一次，又在 (canvas.width, 0) 点绘制了一次。一开始，位于 (0, 0) 点的那幅图像全部可见，而位于 (canvas.width, 0) 点的图像则完全不可见。图 5-8 顶部所演示的正是这种情况。

随着绘图环境对象原点的移动，原来位于屏幕外的那幅图像渐渐滚动至屏幕内，而起初位于屏幕内的那幅图则渐渐移出视野。图 5-8 中，由上至下的 4 张截图演示了这个过程。

其实天空图像左右边界处的内容是一样的，这一点在上图中看得不太明显。图 5-9 展示了左方图像的右边界与右方图像的左边界，在该演示图中，这种效果我们就可以看得很清楚了。由于背景图左右边界处的内容是相同的，所以当图像由屏幕外移动到屏幕内时，我们就不会感觉到图像的断裂了。

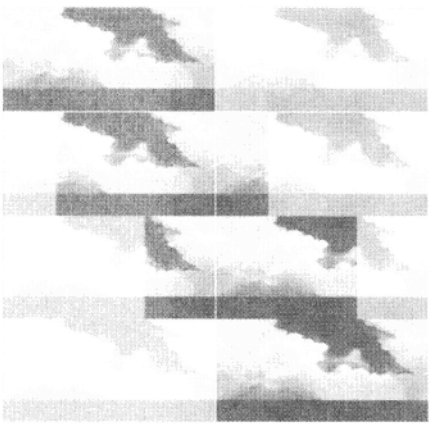


图 5-8　通过移动绘图环境对象的原点坐标来实现背景的滚动

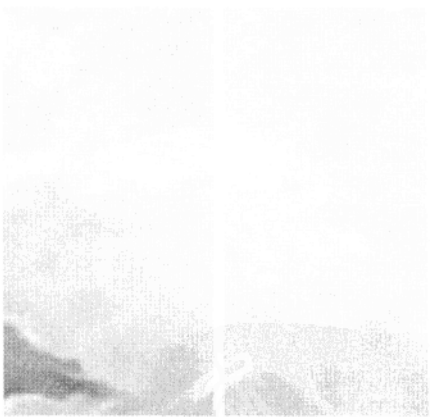


图 5-9　相邻背景图的右边界与左边界，其内容必须一致

在本例中，起初位于屏幕外的那张背景图像，与一开始就显示在屏幕中的那张图是一样的。然而，不是非得这样才行，只要相邻两张（或多张）图像的交界处内容相同就可以[⊖]。只要图像边界处能吻合，那么不管背景图的其余内容怎么变化，我们都可以实现一幅平滑滚动的背景。

图 5-7 所示应用程序的 HTML 代码列在程序清单 5-15 中，其 JavaScript 代码列在程序清单 5-16 中。

⊖ 除上述要求外，还需要让最后一张图像的右边界能够与第一张的左边界相接。——译者注

程序清单 5-15　背景的移动（HTML 代码）

```
<!DOCTYPE html>
   <head>
     <title>Scrolling Backgrounds</title>

      <style>
         body {
            background: #dddddd;
         }

         #canvas {
            position: absolute;
            top: 30px;
            left: 10px;
            background: #ffffff;
            cursor: crosshair;
            margin-left: 10px;
            margin-top: 10px;
            -webkit-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
            -moz-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
            box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
         }

         input {
            margin-left: 15px;
         }

      </style>
   </head>

  <body>
    <canvas id='canvas' width='1024' height='512'>
      Canvas not supported
  </canvas>

  <input id='animateButton' type='button' value='Animate'/>

  <script src='requestNextAnimationFrame.js'></script>
  <script src='example.js'></script>
</body>
</html>
```

程序清单 5-16　背景的滚动（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    controls = document.getElementById('controls'),
    animateButton = document.getElementById('animateButton'),
    sky = new Image(),

    paused = true,
    lastTime = 0,
    fps = 0,

    skyOffset = 0,
    SKY_VELOCITY = 30; // 30 pixels/second

// Functions.......................................................

function erase() {
```

```
    context.clearRect(0, 0, canvas.width, canvas.height);
}

    function draw() {
       context.save();

       skyOffset = skyOffset < canvas.width ?
                   skyOffset + SKY_VELOCITY/fps : 0;

       context.save();
       context.translate(-skyOffset, 0);

       context.drawImage(sky, 0, 0);
       context.drawImage(sky, sky.width-2, 0);

       context.restore();
    }

    function calculateFps(now) {
       var fps = 1000 / (now - lastTime);
       lastTime = now;
       return fps;
    }

    function animate(now) {
       if (now === undefined) {
          now = +new Date;
       }

       fps = calculateFps(now);

       if (!paused) {
          erase();
          draw();
       }

       requestNextAnimationFrame(animate);
    }

    // Event handlers.................................................

    animateButton.onclick = function (e) {
       paused = paused ? false : true;
       if (paused) {
           animateButton.value = 'Animate';
       }
       else {
          animateButton.value = 'Pause';
       }
    };

// Initialization.................................................

canvas.width = canvas.width;
canvas.height = canvas.height;

sky.src = 'sky.png';
sky.onload = function (e) {
   draw();
};

requestNextAnimationFrame(animate);
```

5.8　视差动画

没人知道鸟儿在行走时为何要一上一下地点头，不过有种流行的说法就是，这样做可以让其视野具有立体感。垂直方向的快速运动几乎同时带来两种略微不同的视角，以此产生名为“运动视差”（motion parallax）的效果，使小鸟可以感知物体的深度。

幸好人眼是以“重叠视野”（overlapping vision）来观察外物的，所以我们能自行感知视差及深度，并不需要以那种搞怪的方式去观察物体。从不同的位置，以不同的视线来观察同一个物体时，就会产生视差。这也是远处物体看上去好像比近处物体移动速度慢的原因。

动画制作者让各个动画图层以不同的速度滚动，这样就实现了视差效果。比方说，图 5-10 之中的动画就有 4 个图层，图 5-11 将它们分解展示出来。在这幅动画中，蓝天和白云比其他物体距离观察者更远，所以天空图层从右至左的滚动速度非常慢。接下来，离观察者稍微近一些的则是背景中的小树，它们比天空图层移动得稍快些，但是不如前面的大树速度快，因为大树距离观察者更近。最后，位于最前方的草地图层其滚动速度比动画中其他所有物体都要快很多。

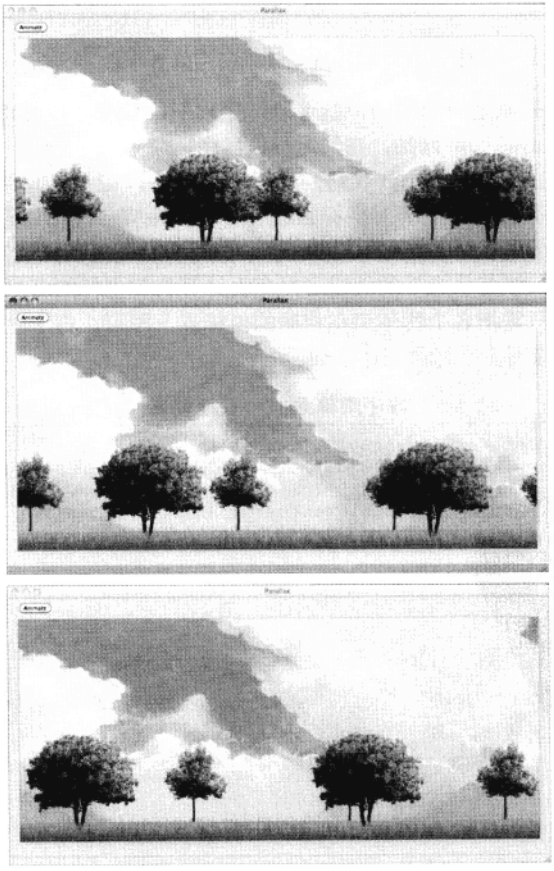


图 5-10　以视差模拟三维效果：由上至下三张图中，前方大树以更快的速度从右至左移动，超越了后方的小树

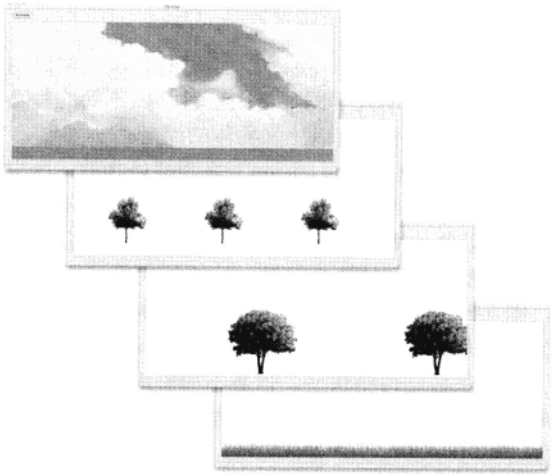


图 5-11 制作视差动画所用的各个图层

这四个图层以不同的速度一起滚动，就可以产生出三维效果了。然而本书无法展示此效果，请读者访问 corehtml5canvas.com 网站，在线运行该程序。

图 5-10 所示应用程序的 JavaScript 代码列在程序清单 5-17 之中。请注意其中的 draw() 方法，该方法先计算出绘制四个图层时需要对坐标系进行平移的距离，然后保存绘图环境对象的状态，平移坐标原点，并将每个图层之中的物体绘制出来，最后再恢复绘图环境对象。这个“保存状态、平移坐标系、绘制物体、恢复状态”的流程可以免去计算物体坐标的麻烦，不管图层在当前这一帧滚动至何处，我们都只需把各个物体画在相同的坐标点上即可。由于坐标系在各帧之间发生了平移，所以画出来的图像看上去好像正在移动。

程序清单 5-17 视差动画

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    controls = document.getElementById('controls'),
    animateButton = document.getElementById('animateButton'),

    tree = new Image(),
    nearTree = new Image(),
    grass = new Image(),
    grass2 = new Image(),
    sky = new Image(),

    paused = true,
    lastTime = 0,
    lastFpsUpdate = { time: 0, value: 0 },
    fps=60,

    skyOffset = 0,
    grassOffset = 0,
    treeOffset = 0,
    nearTreeOffset = 0,

    TREE_VELOCITY = 20,
```

```
    FAST_TREE_VELOCITY = 40,
    SKY_VELOCITY = 8,
    GRASS_VELOCITY = 75;

// Functions.......................................................

function erase() {
    context.clearRect(0, 0, canvas.width, canvas.height);
}

function draw() {
    context.save();

    skyOffset = skyOffset < canvas.width ?
                skyOffset + SKY_VELOCITY/fps : 0;

    grassOffset = grassOffset < canvas.width ?
                  grassOffset + GRASS_VELOCITY/fps : 0;

    treeOffset = treeOffset < canvas.width ?
                 treeOffset + TREE_VELOCITY/fps : 0;

    nearTreeOffset = nearTreeOffset < canvas.width ?
                     nearTreeOffset + FAST_TREE_VELOCITY/fps : 0;

    context.save();
    context.translate(-skyOffset, 0);
    context.drawImage(sky, 0, 0);
    context.drawImage(sky, sky.width-2, 0);
    context.restore();

    context.save();
    context.translate(-treeOffset, 0);
    context.drawImage(tree, 100, 240);
    context.drawImage(tree, 1100, 240);
    context.drawImage(tree, 400, 240);
    context.drawImage(tree, 1400, 240);
    context.drawImage(tree, 700, 240);
    context.drawImage(tree, 1700, 240);
    context.restore();

    context.save();
    context.translate(-nearTreeOffset, 0);
    context.drawImage(nearTree, 250, 220);
    context.drawImage(nearTree, 1250, 220);
    context.drawImage(nearTree, 800, 220);
    context.drawImage(nearTree, 1800, 220);
    context.restore();

    context.save();
    context.translate(-grassOffset, 0);

    context.drawImage(grass, 0,
                      canvas.height-grass.height);
    context.drawImage(grass, grass.width-5,
                      canvas.height-grass.height);
    context.drawImage(grass2, 0,
                      canvas.height-grass2.height);
    context.drawImage(grass2, grass2.width,
                      canvas.height-grass2.height);
    context.restore();
}
```

```
function calculateFps(now) {
    var fps = 1000 / (now - lastTime);
    lastTime = now;
    return fps;
}

function animate(now) {
    if (now === undefined) {
        now = +new Date;
    }

    fps = calculateFps(now);

    if (!paused) {
        erase();
        draw();
    }

    requestNextAnimationFrame(animate);
}

// Event handlers.....................................................

animateButton.onclick = function (e) {
    paused = paused ? false : true;
    if (paused) {
    animateButton.value = 'Animate';
    }
    else {
    animateButton.value = 'Pause';
    }
};

// Initialization.....................................................

context.font = '48px Helvetica';

tree.src = 'smalltree.png';
nearTree.src = 'tree-twotrunks.png';
grass.src = 'grass.png';
grass2.src= 'grass2.png';
sky.src = 'sky.png';
sky.onload = function (e) {
    draw();
};

requestNextAnimationFrame(animate);
```

5.9 用户手势

有些动画可以自行播放，而另一些则需要用户参与互动。在桌面电脑与移动设备上，用户通常会以鼠标或手指触摸来与动画互动，这些方式就叫做用户手势。

图 5-12 之中的应用程序与 4.10 节所展示的那个放大镜程序相同，不过，在这个版本的程序中，用户可以通过快速拖拽并松开鼠标的方式把放大镜“扔”出去。放大镜被扔出去之后，就会沿着刚才拖拽鼠标的方向继续移动，其速度与用户拖拽它的力度有关。若是碰到了 canvas 的边缘，就会被弹回，并继续沿反方向前行。图 5-12 展示了放大镜被用户扔出之后的运动效果。

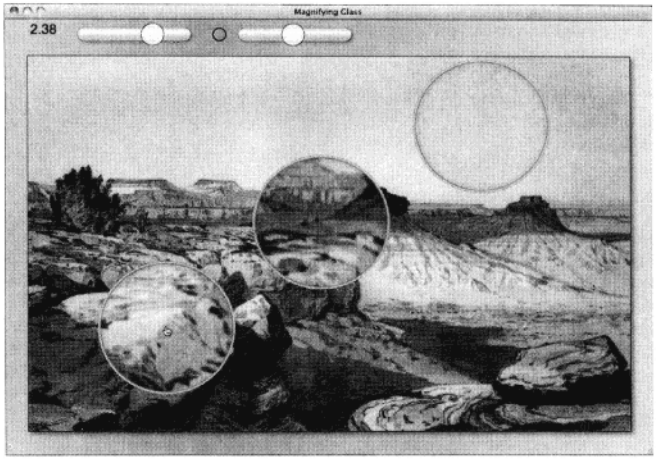


图 5-12　用户手势动画效果演示：正在由左下向右上移动的放大镜

程序清单 5-18 列出了图 5-12 所示应用程序的部分代码。这些代码用于实现用户扔出放大镜的手势。

用户拖拽鼠标时，程序会记录鼠标按下及松开的时间与位置，等到表示拖拽过程结束的那个鼠标松开事件发生后，应用程序的 didThrow() 方法会使用一个简单的等式计算出在本次用户手势中，鼠标光标的移动速度。如果该速度足够快，那么程序就判定用户已经扔出了放大镜，并于随后开始播放其运动动画。

程序清单 5-18　根据用户手势来播放动画（代码节选）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    animating = false,
    dragging = false,
    mousedown = null,
    mouseup = null;

// Functions.......................................................

function didThrow() {
    var elapsedTime = mouseup.time - mousedown.time;
    var elapsedMotion = Math.abs(mouseup.x - mousedown.x) +
                        Math.abs(mouseup.y - mousedown.y);
    return (elapsedMotion / elapsedTime * 10) > 3;
}

// Event handlers..................................................

canvas.onmousedown = function (e) {
    var mouse = windowToCanvas(e.clientX, e.clientY);
    mousedown = { x: mouse.x, y: mouse.y, time: (new Date).getTime() };
    e.preventDefault(e);
    if (animating) { // Stop the current animation
```

```
            animating = false;
            clearInterval(animationLoop);
            eraseMagnifyingGlass();
        }
        else { // Start dragging
            dragging = true;
            context.save();
        }
    };

    canvas.onmousemove = function (e) {
        if (dragging) {
            eraseMagnifyingGlass();
            drawMagnifyingGlass(
                windowToCanvas(e.clientX, e.clientY));
        }
    };

    canvas.onmouseup = function (e) {
        var mouse = windowToCanvas(canvas, e.clientX, e.clientY);
        mouseup = { x: mouse.x, y: mouse.y, time: (new Date).getTime() };

        if (dragging) {
            if (didThrow()) {
                velocityX = (mouseup.x-mousedown.x)/100;
                velocityY = (mouseup.y-mousedown.y)/100;
                animate(mouse, { vx: velocityX, vy: velocityY });
            }
            else {
                eraseMagnifyingGlass();
            }
        }
        dragging = false;
    };
```

5.10 定时动画

本章到现在为止所讲的动画都是持续运行的，然而，许多动画却只需要在某个时间段内运行即可。这一小节就要讲述如何通过秒表让动画在不同的时间段内运行，读者还可以学到怎么将秒表逻辑封装到简单的 Animation 对象中。

5.10.1 秒表

图 5-13 所示应用程序模拟了一块秒表，用户可以在程序的文本框中输入倒计时的秒数，然后按下“Start”按钮来启动它。启动后，其指针会随着时间的流逝平缓地绕回 0 点。

图 5-13 中的程序用到了名为 Stopwatch 的对象，该对象有如下方法：

- void start()
- void stop()
- Number getElapsedTime()
- Boolean isRunning()
- void reset()

用户可以随时启动或停止秒表，也可以看到它当前的倒计时读数。如果这个数字在不停地减

少，则说明秒表正在倒计时。若要将其复位，则先停止秒表，然后在倒计时框输入 0，最后再按下“Start”按钮即可。Stopwatch 对象的实现代码列在了程序清单 5-19 之中。

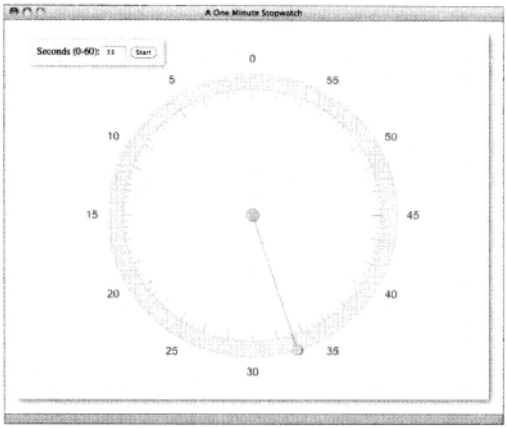


图 5-13　秒表

程序清单 5-20 列出了图 5-13 所示应用程序的 JavaScript 代码选段。

程序中的“Start”按钮有两个用途：既可以启动秒表，又可以将其停止。如果在显示“Start”时按下该按钮，应用程序则会启动秒表，并将按钮文本改为“Stop”，同时禁用倒计时输入框，然后请求浏览器在执行下一帧动画逻辑时开始绘制秒表的倒计时效果。

如果在按钮显示为“Stop”时按下了它，那么程序则会停止秒表，并将按钮文本改为“Start”，同时启用倒计时输入框。请注意，秒表停止之后，应用程序就不会再调用 requestNextAnimationFrame() 方法了，如此一来，倒计时动画也就随着秒表一起停止了。

程序清单 5-19　Stopwatch 对象的实现代码

```
// Stopwatch.............................................................
//
// Like the real thing, you can start and stop a stopwatch, and you
// can find out the elapsed time the stopwatch has been running.
// After you stop a stopwatch, its getElapsedTime() method returns
// the elapsed time between the start and stop.
//
// Stopwatches are used primarily for timing animations.

// Constructor...........................................................

Stopwatch = function (){ };

// Prototype.............................................................

Stopwatch.prototype = {
    startTime: 0,
    running: false,
    elapsed: undefined,

    start: function () {
        this.startTime = +new Date();
```

```
        this.elapsedTime = undefined;
        this.running = true;
    },

    stop: function () {
        this.elapsed = (+new Date()) - this.startTime;
        this.running = false;
    },

    getElapsedTime: function () {
        if (this.running) {
            return (+new Date()) - this.startTime;
        }
        else {
            return this.elapsed;
        }
    },

    isRunning: function() {
        return this.running;
    },

    reset: function() {
        this.elapsed = 0;
    }
};
```

程序清单 5-20　使用 stopwatch 对象来实现秒表

```
var stopwatch = new Stopwatch(),
    secondsInput = document.getElementById('secondsInput'),
    startStopButton = document.getElementById('startStopButton');
...

startStopButton.onclick = function (e) {
    var value = startStopButton.value;
    if (value === 'Start') {
        stopwatch.start();
        startStopButton.value = 'Stop';
        requestNextAnimationFrame(animate);
        secondsInput.disabled = true;
    }
    else {
        stopwatch.stop();
        timerSetting = parseFloat(secondsInput.value);
        startStopButton.value = 'Start';
        secondsInput.disabled = false;
    }
    stopwatch.reset();
};

function animate() {
    if (stopwatch.isRunning() &&
        stopwatch.getElapsedTime() > timerSetting*1000) {

        // Animation is over

        stopwatch.stop();
        startStopButton.value = 'Start';
        secondsInput.disabled = false;
        secondsInput.value = 0;
```

```
        }
        else if (stopwatch.isRunning()) { // Animation is running
            redraw();
            requestNextAnimationFrame(animate);
        }
    }
```

只要秒表运行的时间未达到用户设置的倒计时值，那么 animate() 函数就重绘秒表，并通过 requestNextAnimationFrame() 方法请求浏览器稍后再次调用自己。若是秒表运行的时间超出了用户设定的倒计时秒数，那么程序就不再运行秒表的倒计时逻辑及其动画效果了。

5.10.2　动画计时器

程序清单 5-20 向大家展示了如何用秒表来控制动画的播放时间。这么做的确很有用，不过我们不妨在更高的抽象层面上实现一个 AnimationTimer 对象，用它来控制动画的播放，则会更加方便。其代码如程序清单 5-21 所示。

程序清单 5-21　动画计时器的实现代码

```
// Constructor.........................................................

AnimationTimer = function (duration){
    this.duration = duration;
};

// Prototype...........................................................

AnimationTimer.prototype = {
    duration: undefined,
    stopwatch: new Stopwatch(),

    start: function () {
        this.stopwatch.start();
    },

    stop: function () {
        this.stopwatch.stop();
    },

    getElapsedTime: function () {
        var elapsedTime = this.stopwatch.getElapsedTime();

        if (!this.stopwatch.running)
            return undefined;
        else
            return elapsedTime;
    },

    isRunning: function() {
        return this.stopwatch.isRunning();
    },

    isOver: function () {
        return this.stopwatch.getElapsedTime() > this.duration;
    },
};
```

AnimationTimer 对象是在上一小节所讲的 Stopwatch 对象外围包裹了一小层代码。其大多数功能都是直接代理给 stopwatch 对象来完成的，只是新增了一个名为 isOver() 的方法。该方法告

诉调用者动画的播放时间是否超出了其预定时长。对于已经播放完毕的动画，开发者通常需要手动将其停止，因为它们并不会自动停下来。

AnimationTimer 对象的用法与 stopwatch 一样，此外它还多了 isOver() 方法可供调用。需要注意的是，AnimationTimer 对象实际上并不播放动画，它只是构建了一个计时器，用以将“时间”这个因素抽象出来。在 7.2 节中大家会更为深入地理解这一点，那时我们将会扩展该对象，用它来实现各式各样的非线性时间轴扭曲（time warp）操作，这其中包括了缓动（easing）及弹簧（elasticity）效果等非线性的移动方式。

5.11 动画制作的最佳指导原则

在制作动画时，请牢记下列指导原则：

- 使用类似 requestNextAnimationFrame() 这样的“polyfill 式”方法来保持浏览器兼容性。
- 将业务逻辑的更新与动画的绘制分开。
- 使用“基于时间的运动”来协调动画的播放速度。
- 用剪辑区域或图块复制技术将复杂的背景图像恢复到屏幕上。
- 必要时可使用一个或多个离屏缓冲区以提升背景的绘制速度。
- 不要手工实现传统的双缓冲算法：浏览器会自动实现它的。
- 不要通过 CSS 指定阴影及圆角效果。
- 不要在 Canvas 中进行带有阴影效果的绘制操作。
- 不要在播放动画时分配内存。
- 使用性能调试及时间轴工具来监控并改善动画的绘制效率。

制作动画时应该使用 requestNextAnimationFrame() 这样的“polyfill 式”函数，此种函数要能够应对本章所述 Firefox 4.0 及 Chrome 10 浏览器特有的 bug 才行。requestNextAnimationFrame() 比 setTimeout() 或 setInterval() 要好，因为它是专门为了制作动画而编写的。

将运动物体的更新逻辑与这些物体的绘制分开来做也是个好办法，因为如果在绘制的同时修改运行物体的属性，则有可能干扰动画的播放。

制作者还应该使用“基于时间的运动”算法来确保动画在不同情况下都能以相同的速度运行，不受底层帧速率的影响。对于大多数动画来说，哪怕应用程序运行得再慢，我们都要保证动画能够以平稳的速度播放，在制作游戏时尤其要注意这一点。运用本章所学知识，开发者可以实现“基于时间的运动”算法，让动画播放速度不受帧速率的影响。至少对于简单的运动方式而言，这种算法是很容易实现的。

请记住，浏览器会自动对 Canvas 元素运用双缓冲机制。由于这个原因，所以开发者没有必要自己再去实现它。而离屏缓冲区则是很有用的，我们经常在程序中使用一个或更多的离屏缓冲区，尤其在绘制复杂动画背景时更是如此。在其余因素都相同的情况下，从离屏缓冲区中复制图像到屏幕上，与直接重绘每帧动画的背景相比，绘制效率要更高一些。

不论是使用 CSS 代码来指定效果还是直接修改 Canvas 元素属性，阴影与渐变都会降低绘图效率，在移动设备上尤为明显。如果应用程序运行得很慢，那么一定要在启用阴影与渐变效果和不启用效果的情况下分别测试，以确定是否由于启用了阴影与渐变效果而导致程序变慢。

最后要说的是，在播放动画时，应该避免分配内存。因为此时浏览器不会运行垃圾收集器

(garbage collector)，至少可以说，它此时执行垃圾回收的频率非常低。开发者也可以使用性能分析及时间轴工具来寻找性能瓶颈。

5.12　总结

在本章中，读者学会了如何用 Canvas 来实现动画。大家也看到了，虽说 setTimeout() 与 setInterval() 也可以制作动画，不过我们还是应该首选 requestAnimationFrame() 或与其功能相同的浏览器特定方法。

接下来我们又讲了“基于时间的运动”，它可以使动画以恒定的速度运行，不受底层帧速率的影响。

然后，大家学到了如何制作滚动的动画背景，以及如何利用“近处的物体看上去比远处的物体移动速度快”这一原理来实现视差动画，模拟三维效果。

最后，我们讲了如何通过检测用户手势来控制动画的播放，并总结了一些制作 Canvas 动画的最佳指导原则，以此来结束本章。在下一章中，我们要花一点儿时间，把本章所学的知识封装起来，便于以后复用，这样在每次制作动画时就不需要再从头开始编写代码了。

第6章
精　　灵

上一章讲述了 canvas 动画的实现方式。读者学会了如何使用 requestAnimationFrame() 方法实现平滑的动画效果，学会了如何利用剪辑区域和离屏 canvas 技术绘制动画背景，还学会了如何利用“基于时间的运动”来控制动画播放速度，以及如何使用秒表与动画计时器来实现定时动画。

在制作基于 Canvas 的动画时，最好能够将这些基本功能都封装为 JavaScript 语言的对象，这样的话，每次实现动画时就不用再从头开始编写代码了。本章就来研究“精灵”（sprite）的实现方式，它是一种可以集成入动画之中的图形对象。你会看到如何在不影响动画背景的情况下移动精灵，并赋予它们各种行为（behavior）。比方说，可以给“小球”对象添加“弹起”行为，给“炸弹”对象添加“爆炸”行为。这些行为可以是无限重复的，也可以只在某一段时间内发生，这两种实现方式读者都可以在本章中学到。

精灵对象也可以随着时间来改变其样貌，这样就可以模拟出爆炸等效果来。读者将会看到“精灵动画制作器”（sprite animator）的用法，这种对象可以周期性地改变精灵的外观。

提示：精灵的历史

“精灵”一词本来指的是希腊神话中的“Fairy”（小仙子、妖精）。其作为一个计算机词汇，最早是由德州仪器（Texas Instruments）9918(A) 型“视频显示处理器”（video display processor）的实现者所创造的。精灵可以通过软件或硬件的方式实现，比如，1985 年上市的 Commodore Amiga 电脑，就支持 8 个以硬件方式实现的精灵。与精灵历史有关的详细信息，请参见以下维基百科页面：http://en.wikipedia.org/wiki/Sprite_(computer_graphics)。

提示：精灵并非 Canvas API 的一部分

Canvas API 并未直接提供对精灵的支持，然而，它提供了实现精灵所需的全部图形处理功能。本章所讲的精灵、绘制器（painter）、动画制作器（animator）等等对象，虽说不是 Canvas API 的一部分，但却都是从它衍生而来的。

有无数种方式都可以用来实现本章所述的精灵、精灵动作以及精灵动画。开发者可以直接使用本章的代码，也可以根据各自需要来修改它们，还可以依据本章所讲的概念来自行编写精灵的实现代码。

提示：设计模式、精灵行为以及动画制作器

本章将会实现三种设计模式：策略模式（Strategy）、命令模式（Command）与享元模式（Flyweight）。策略模式用于将精灵与绘制器解耦（decouple），命令模式用于实现精灵的动作，而享元模式则可以让我们用一个实例来表示多个精灵。

本章还实现了开源项目中的两个概念。第一个概念叫做“行为”，它来源于 Android 平台一款知名的开源游戏：Replica Island。第二个概念叫做“精灵动画制作器”，它的最终实现会在第 7 章

中讲到。此概念源自一个知名的动画底层程序库：Animator.js。

Replica Island 及 Animator.js 项目的更多信息，请分别参阅下列网址：http://bit.ly/kNzDVc，http://bit.ly/krLIo6。

6.1 精灵概述

要制作一个有用的精灵对象，必须让开发者能把它们绘制出来，能够将其放置于动画中的指定位置，并且能以给定的速度将其从一个地方移动到另一个地方。这些精灵对象或许还能接受调用者的命令，来执行某些特定的动作，例如下落、弹起、飞行、爆炸，以及与其他精灵碰撞等等。表 6-1 列出了 Sprite 对象的属性。

painter 属性是一个指向 Painter 对象的引用，该对象使用 paint(sprite, context) 方法来绘制精灵。behaviors 属性指向一个对象数组，数组中的每个对象都会以 execute(sprite, context, time) 方法来对精灵进行某种形式的操作。程序清单 6-1 列出了 Sprite 对象的实现代码。

精灵对象有两个方法：paint() 与 update()。update() 方法用于执行每个精灵的行为，执行的顺序就是这些行为被加入精灵之中的顺序。paint() 方法则将精灵的绘制代理给绘制器来做，不过仅仅在精灵确实有绘制器并且可见时，此方法才会生效。

表 6-1　Sprite 对象的属性

属性	描　述
top	精灵左上角（upper left-hand corner，简称 ulhc）的 Y 坐标
left	精灵左上角的 X 坐标
width	精灵的宽度
height	精灵的高度
velocityX	精灵的水平速度
volocityY	精灵的垂直速度
behaviors	一个包含精灵行为对象的数组，在精灵执行更新逻辑时，该数组中的各行为对象都会被运用于此精灵
painter	用于绘制此精灵对象的绘制器
visible	表示此精灵是否可见的 boolean 标志
animating	表示此精灵是否正在执行动画效果的 boolean 标志

Sprite 构造器接受三个参数：精灵的名称、绘制器及行为数组。

程序清单 6-1　Sprite 对象的代码

```
// Constructor.........................................................

var Sprite = function (name, painter, behaviors) {
   if (name !== undefined)       this.name = name;
   if (painter !== undefined)    this.painter = painter;

   this.top = 0;
   this.left = 0;
   this.width = 10;
   this.height = 10;
   this.velocityX = 0;
   this.velocityY = 0;
```

```
    this.visible = true;
    this.animating = false;
    this.behaviors = behaviors || [];

    return  this;
};
// Prototype.......................................................
Sprite.prototype = {
    paint: function (context) {
        if (this.painter !== undefined && this.visible) {
            this.painter.paint(this, context);
        }
    },

    update: function (context, time) {
        for (var i = 0; i < this.behaviors.length; ++i) {
            this.behaviors[i].execute(this, context, time);
        }
    }
};
```

既然大家已经知道了精灵的实现方式，那么咱们就看看如何使用它吧。图 6-1 所示应用程序就展示了一个简单的精灵对象。

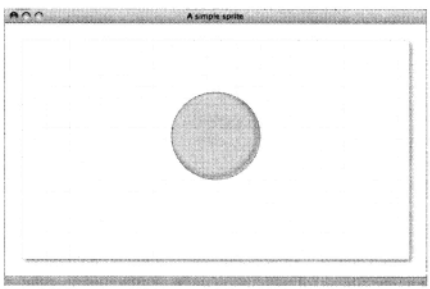


图 6-1　简单的精灵演示程序

图 6-1 所示应用程序的 JavaScript 代码列在了程序清单 6-2 中。

程序清单 6-2　简单的精灵演示程序（JavaScript 代码）

```
var context = document.getElementById('canvas').getContext('2d'),
    RADIUS = 75,
    ball = new Sprite('ball',
        {
            paint: function(sprite, context) {
                context.beginPath();
                context.arc(sprite.left + sprite.width/2,
                            sprite.top + sprite.height/2,
                            RADIUS, 0, Math.PI*2, false);
                context.clip();

                context.shadowColor = 'rgb(0,0,0)';
                context.shadowOffsetX = -4;
                context.shadowOffsetY = -4;
                context.shadowBlur = 8;

                context.lineWidth = 2;
                context.strokeStyle = 'rgb(100,100,195)';
```

```
                context.fillStyle = 'rgba(30,144,255,0.15)';
                context.fill();
                context.stroke();
            }
        }
    );

function drawGrid(color, stepx, stepy) {
   // Draws a grid. See Section 2.8.2
   // for a full listing
}

drawGrid('lightgray', 10, 10);
ball.left = 320;
ball.top = 160;
ball.paint(context);
```

程序清单 6-2 之中的这段代码，创建了一个名为 ball 的精灵对象，并在创建该对象时指定了一个自定义的绘制函数来绘制这个小球。ball 对象的实现代码比较简单，但它显得很乏味，因为该对象并未绑定任何行为。在 6.3 节中，我们将学习如何将行为绑定到精灵对象之上。

6.2　精灵绘制器

Sprite 对象与绘制其内容的绘制器对象之间是解耦的（decoupled）。如此一来，就可以在程序运行时为精灵对象动态地设定绘制器了，这极大地提升了程序的灵活度。比方说，可以实现一个精灵动画制作器，它每隔一段时间就将精灵的绘制器交换一次。实际上，本书 6.4 节就会讲到这种动画制作器的实现方式。

Painter 对象只需实现如下这个方法即可：void paint(sprite, context)。所有 Painter 对象都可被归纳为以下三类：

- 描边及填充绘制器
- 图像绘制器
- 精灵表绘制器

描边及填充绘制器使用 Canvas 的图形 API 来绘制精灵，而图像绘制器则用于绘制图像。最后，精灵表绘制器用于绘制精灵表中的单个精灵。我们来看看每一种绘制器的用法。

提示：绘制器与策略模式

精灵对象不需要自己完成绘制，相反，它会将绘制操作代理给另外一个对象来做。从本质上讲，Painter 对象就是一些可以互相交换着使用的绘制算法，在程序运行时，开发者可以将其设定给精灵对象，这种特色表明，绘制器就是策略模式的一个实际用例。策略模式的详细信息请参见：http://bit.ly/k94Fro。

提示：sprites.js 程序文件

Sprite 对象的实现代码，连同与精灵有关的 ImagePainter 等各对象的代码，都放在名为 sprites.js 的文件之中。本章的所有范例程序，都会在 HTML 页面中引入这个文件。

6.2.1　描边与填充绘制器

描边与填充绘制器（stroke and fill painter）会调用包括 stroke() 与 fill() 在内的 Canvas 图形

函数来绘制精灵。例如，图 6-2 所展示的这个时钟，它的三个指针就是使用精灵来表示的。

首先，应用程序创建了一个用于绘制精灵的对象，然后将这个 Painter 对象传给了 Sprite 构造器：

```
var ballPainter = {
   paint: function (sprite, context) {
      var x = sprite.left + sprite.width/2,
          y = sprite.top  + sprite.height/2,
          ...
          radius = sprite.width/2;

      context.save();
      context.beginPath();
      context.arc(x, y, radius, 0, Math.PI*2, false);
      context.clip();

      // Continue drawing the sprite...

      context.restore();
   }
},
...
ball = new Sprite('ball', ballPainter);
```

接下来，应用程序使用名为 ball 的精灵对象来绘制时钟指针，其代码如下：

```
function drawHand(loc, isHour) {
   // Move ball to the appropriate location
   ...
   ball.paint(context);
}

function drawHands() {
   var date = new Date(),
       hour = date.getHours();

    // Seconds

    ball.width = 20;
    ball.height = 20;
    drawHand(date.getSeconds(), false);

    // Minutes

    hour = hour > 12 ? hour - 12 : hour;
    ball.width = 35;
    ball.height = 35;
    drawHand(date.getMinutes(), false);

   // Hours

   ball.width = 50;
   ball.height = 50;
   drawHand(hour*5 + (date.getMinutes()/60)*5, true);

   // Centerpiece

   ball.width = 10;
   ball.height = 10;
   ball.left = canvas.width/2 - ball.width/2;
   ball.top = canvas.height/2 - ball.height/2;
   ballPainter.paint(ball, context);
}
```

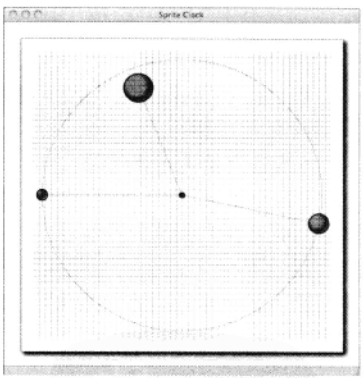


图 6-2　使用精灵绘制的时钟，其指示的时刻为 11 时 17 分 45 秒

图 6-2 所示应用程序的全部 JavaScript 代码都列在了程序清单 6-3 之中。

程序清单 6-3　使用精灵绘制的时钟（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    CLOCK_RADIUS = canvas.width/2 - 15,
    HOUR_HAND_TRUNCATION = 35,

    // Painter.......................................................

    ballPainter = {
      paint: function (sprite, context) {
         var x = sprite.left + sprite.width/2,
             y = sprite.top + sprite.height/2,
             width = sprite.width,
             height = sprite.height,
             radius = sprite.width/2;

         context.save();
         context.beginPath();
         context.arc(x, y, radius, 0, Math.PI*2, false);
         context.clip();

         context.shadowColor = 'rgb(0,0,0)';
         context.shadowOffsetX = -4;
         context.shadowOffsetY = -4;
         context.shadowBlur =  8;

         context.fillStyle = 'rgba(218,165,32,0.1)';
         context.fill();

         context.lineWidth = 2;
         context.strokeStyle = 'rgb(100,100,195)';
         context.stroke();

         context.restore();
       }
    },

    // Sprite........................................................

    ball = new Sprite('ball', ballPainter);

// Functions.........................................................

function drawGrid(color, stepx, stepy) {
   // Omitted for brevity. See Example 2.13
   // for a complete listing.
   ...
}

function drawHand(loc, isHour) {
   var angle = (Math.PI*2) * (loc/60) - Math.PI/2,
       handRadius = isHour ? CLOCK_RADIUS - HOUR_HAND_TRUNCATION
                           : CLOCK_RADIUS,
       lineEnd = {
          x: canvas.width/2 +
             Math.cos(angle)*(handRadius - ball.width/2),
          y: canvas.height/2 +
```

```
                Math.sin(angle)*(handRadius - ball.width/2)
        };

    context.beginPath();
    context.moveTo(canvas.width/2, canvas.height/2);
    context.lineTo(lineEnd.x, lineEnd.y);
    context.stroke();

    ball.left = canvas.width/2  +
                Math.cos(angle)*handRadius - ball.width/2;

    ball.top = canvas.height/2 +
               Math.sin(angle)*handRadius - ball.height/2;

    ball.paint(context);
}

function drawClock() {
    drawClockFace();
    drawHands();
}

function drawHands() {
    var date = new Date(),
        hour = date.getHours();

    ball.width = 20;
    ball.height = 20;
    drawHand(date.getSeconds(), false);

    hour = hour > 12 ? hour - 12 : hour;
    ball.width = 35;
    ball.height = 35;
    drawHand(date.getMinutes(), false);

    ball.width = 50;
    ball.height = 50;
    drawHand(hour*5 + (date.getMinutes()/60)*5);

    ball.width = 10;
    ball.height = 10;
    ball.left = canvas.width/2 - ball.width/2;
    ball.top = canvas.height/2 - ball.height/2;
    ballPainter.paint(ball, context);
}

function drawClockFace() {
     context.beginPath();
     context.arc(canvas.width/2, canvas.height/2,
                 CLOCK_RADIUS, 0, Math.PI*2, false);

    context.save();
    context.strokeStyle = 'rgba(0,0,0,0.2)';
    context.stroke();
    context.restore();
}

// Animation.........................................................

function animate() {
    context.clearRect(0, 0, canvas.width, canvas.height);
```

```
    drawGrid('lightgray', 10, 10);
    drawClock();

    window.requestNextAnimationFrame(animate);
}

// Initialization.....................................................

context.lineWidth = 0.5;
context.strokeStyle = 'rgba(0,0,0,0.2)';
context.shadowColor = 'rgba(0,0,0,0.5)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur = 4;
context.stroke();

window.requestNextAnimationFrame(animate);

drawGrid('lightgray', 10, 10);
```

学会了描边及填充绘制器之后，咱们来看看如何实现图像绘制器（image painter）。

小技巧：享元精灵

虽说图 6-2 中的应用程序看上去需要 4 个精灵对象才能绘制出来，可实际上只用了 1 个。从程序清单 6-3 中得知，应用程序在绘制 3 个指针以及表盘中心的那个枢轴时，所用的是同一个精灵对象。

使用一个对象来表示多个概念，这就是享元模式。它减少了需要创建的对象数，从而降低了内存占用量，这对于动画和电子游戏的制作来说，是极为重要的。

小技巧：使用 window.requestNextAnimationFrame() 来控制动画播放

图 6-2 所示应用程序使用了 5.1.3 小节中所讲的“polyfill 式方法”：window.requestNext-Animation-Frame()。该方法的实现代码位于 requestNextAnimationFrame.js 文件中，应用程序在 HTML 页面的代码中引入了此文件。

6.2.2　图像绘制器

图像绘制器对象含有一个指向图像对象的引用，它会将此图像绘制到经由 paint() 方法所传入的绘图环境对象之上。图像绘制器实现起来很简单，如程序清单 6-4 所示。

程序清单 6-4　供精灵对象所用的图像绘制器

```
var ImagePainter = function (imageUrl) {
    this.image = new Image();
    this.image.src = imageUrl;
};

ImagePainter.prototype = {
    paint: function (sprite, context) {
        if (this.image.complete) {
            context.drawImage(this.image, sprite.left, sprite.top,
                              sprite.width, sprite.height);
        }
    }
};
```

在创建图像绘制器时，需要将指向图像 URL 的引用传给 ImagePainter 构造器。只有当图像完全载入之后，图像绘制器的 paint() 方法才会将其绘制出来。

图 6-3 中的这个精灵是用图像绘制器来构建的。

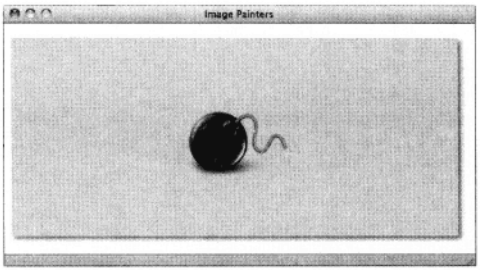


图 6-3　使用图像绘制器构建的精灵对象

程序清单 6-5 列出了图 6-3 所示应用程序的 JavaScript 代码。该应用程序演示了一幅简单的动画，它在反复地绘制一个含有炸弹图像的精灵。

程序清单 6-5　使用图像绘制器来构建精灵对象（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    bomb = new Sprite('bomb', new ImagePainter('bomb.png')),
    BOMB_LEFT = 220,
    BOMB_TOP = 80,
    BOMB_WIDTH = 180,
    BOMB_HEIGHT = 130;

function animate() {
   context.clearRect(0, 0, canvas.width, canvas.height);
   bomb.paint(context);
   window.requestNextAnimationFrame(animate);
}

bomb.left = BOMB_LEFT;
bomb.top = BOMB_TOP;
bomb.width = BOMB_WIDTH;
bomb.height = BOMB_HEIGHT;

window.requestNextAnimationFrame(animate);
```

警告：图像的加载

请注意，程序清单 6-5 中的程序并不仅仅是把精灵绘制完一次就停下了，它要反复地绘制这个精灵对象，因为该精灵是要用在动画效果之中的。虽说程序清单 6-5 所演示的动画碰巧不怎么有趣，不过它依然算得上是一个动画程序。

由于图像绘制器专门负责反复绘制精灵对象所用的图像，所以它们并不负责图像的加载。如果在调用图像绘制器的 draw() 方法时，图像尚未载入，那么该方法不会执行任何操作。要是几毫秒之后图像已经加载好了，那么再次调用该方法时它就会将精灵图像绘制出来，如果到时图像仍然没有完成加载，那么动画循环逻辑就会继续执行下去，等到图像加载完毕，精灵才会显示出来。

这种有些漠视开发者需求的图像加载策略并不能应对所有情况。比方说，有时候我们必须在开始绘制之前就要预先把全部图像都加载好。在 9.1.2 小节中，我们将制作一个图像加载器来满足这种需要。

6.2.3 精灵表绘制器

为了节省磁盘空间、减少下载次数，如果用于制作动画的精灵对象其每帧所用的图像都比较小，那么就可以把它们都放在一张图片中，如图 6-4 所示。这张包含动画每一帧图像的图片，就叫做精灵表（sprite sheet）。

图 6-4　精灵表

在绘制动画的某一帧时，我们从精灵表中将该帧图像所对应的矩形区域复制到屏幕上，即可将其显示出来了。

从一张图片中复制多个矩形区域中的图像，要比直接复制多张图像到屏幕上快得多，而且，将许多小图像存放在一个图形文件中，可以极大地减少应用程序所发送的 HTTP 请求数。所以说，不论从哪方面看，使用精灵表来制作动画都是个不错的选择。

精灵表绘制器（sprite sheet painter）会把精灵表中表示当前动画帧的那个单元格画出来。绘制器对象中还有一个数组索引，该数组中的元素对应于精灵表中每个单元格的信息，调用 advance() 方法可以将索引值加 1。程序清单 6-6 列出了精灵表绘制器的代码。

程序清单 6-6　精灵表绘制器

```
SpriteSheetPainter = function (cells) {
   this.cells = cells || [];
   this.cellIndex = 0;
};

SpriteSheetPainter.prototype = {
   advance: function () {
      if (this.cellIndex == this.cells.length-1) {
          this.cellIndex = 0;
      }
      else {
         this.cellIndex++;
      }
   },

   paint: function (sprite, context) {
      var cell = this.cells[this.cellIndex];
      context.drawImage(spritesheet, cell.x, cell.y, cell.w, cell.h,
                        sprite.left, sprite.top, cell.w, cell.h);
   }
};
```

图 6-5 所示应用程序在页面上方显示出一张简单的精灵表，同时在页面下方使用该精灵表中的图像来绘制精灵动画。应用程序使用精灵表绘制器来将精灵表中的单元格绘制到屏幕上。

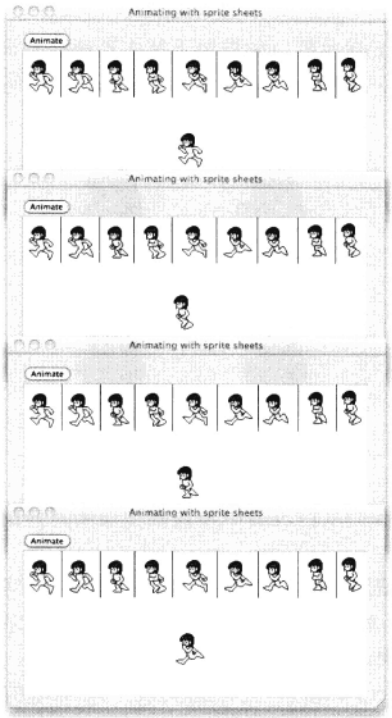


图 6-5 用精灵表制作动画

图 6-5 所示应用程序的代码列在了程序清单 6-7 之中。该程序将动画的帧速率设置为每秒 10 帧，因为在每秒 60 帧的情况下，9 帧的动画会以极快的速度飞逝而过。该程序还使用了 9 参数版本的 drawImage() 方法，将精灵表中需要显示的那块矩形区域绘制到 canvas 之上。关于 drawImage() 方法的更多信息，请参阅 4.1.2 小节。

程序清单 6-7 使用精灵表绘制器制作动画（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    animateButton = document.getElementById('animateButton'),
    spritesheet = new Image(),
    runnerCells = [
        { left: 0,   top: 0, width: 47, height: 64 },
        { left: 55,  top: 0, width: 44, height: 64 },
        { left: 107, top: 0, width: 39, height: 64 },
        { left: 150, top: 0, width: 46, height: 64 },
        { left: 208, top: 0, width: 49, height: 64 },
        { left: 265, top: 0, width: 46, height: 64 },
        { left: 320, top: 0, width: 42, height: 64 },
        { left: 380, top: 0, width: 35, height: 64 },
        { left: 425, top: 0, width: 35, height: 64 },
    ],
    sprite = new Sprite('runner', new SpriteSheetPainter(runnerCells)),
```

```
    interval,
    lastAdvance = 0,
    paused = false,
    PAGEFLIP_INTERVAL = 100;

// Functions.......................................................

function drawBackground() {
   var STEP_Y = 12,
       i = context.canvas.height;

   while(i < STEP_Y*4) {
      context.beginPath();
      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();
      i -= STEP_Y;
  }
}

function pauseAnimation() {
   animateButton.value = 'Animate';
   paused = true;
}

function startAnimation() {
   animateButton.value = 'Pause';
   paused = false;
   lastAdvance = +new Date();
   window.requestNextAnimationFrame(animate);
}

// Event handlers..................................................

animateButton.onclick = function (e) {
   if (animateButton.value === 'Animate') startAnimation();
   else                                   pauseAnimation();
};

// Animation.......................................................

function animate(time) {
   if ( ! paused) {
      context.clearRect(0, 0, canvas.width, canvas.height);
      drawBackground();
      context.drawImage(spritesheet, 0, 0);

      sprite.paint(context);

      if (time - lastAdvance > PAGEFLIP_INTERVAL) {
         sprite.painter.advance();
         lastAdvance = time;
      }
      window.requestNextAnimationFrame(animate);
   }
}

// Initialization.................................................

spritesheet.src = 'running-sprite-sheet.png';
spritesheet.onload = function(e) {
   context.drawImage(spritesheet, 0, 0);
};

sprite.left = 200;
sprite.top = 100;
context.strokeStyle = 'lightgray';
context.lineWidth = 0.5;

drawBackground();
```

该应用程序的 animate() 方法先清除 canvas，绘制背景及页面上方的精灵表，然后再绘制精灵。

绘制好精灵之后，animate() 方法会判断当前这一帧的持续时间是否已经超过了 PAGEFLIP_INTERVAL 毫秒，如果是的话，那么就切换到下一帧。最后，应用程序向 window 对象请求在浏览器绘制下一帧动画时再次调用 animate() 方法。这样的话，稍后该方法就会被执行，而新一轮的动画循环逻辑也会就此开始。

6.3 精灵对象的行为

在学会了精灵的绘制之后，我们继续来研究如何为精灵对象增加行为，使其能够像人一样执行各种动作。

其实，只要实现了 execute(sprite, context, time) 方法的对象，都可以叫做“行为”。该方法一般会以某种方式来修改精灵的属性，比如移动其位置，或是修改其外观。

精灵含有一个行为对象数组，它的 update() 方法会遍历该数组，使每个行为对象都得以执行一次。这样的话，我们就可以把行为封装为对象，在程序运行的时候将它添加到多个精灵之中。程序清单 6-8 之中的应用程序实现了一个行为对象，该对象将“精灵原地跑步”这个动作封装起来。图 6-5 展示了此动作的运行效果。

程序清单 6-8　原地跑步的精灵

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...
    runInPlace = {
       lastAdvance: 0,
       PAGEFLIP_INTERVAL: 1000,
       execute: function (sprite, context, now) {
          if (now - this.lastAdvance > this.PAGEFLIP_INTERVAL) {
              sprite.painter.advance();
              this.lastAdvance = now;
          }
       }
    },
    sprite = new Sprite('runner',
             new SpriteSheetPainter(runnerCells), [ runInPlace ]);
...
function animate(time) {
   context.clearRect(0, 0, canvas.width, canvas.height);
   drawBackground();
   context.drawImage(spritesheet, 0, 0);
   sprite.update(context, time);
   sprite.paint(context);
   window.requestNextAnimationFrame(animate);
}
...
```

程序清单 6-8 所创建的 runInPlace 对象有一个名为 execute() 的方法，这意味着它可以当成行为对象来用。应用程序在创建精灵对象时，把只包含 runInPlace 这一个对象的行为数组，传递给精灵的构造器。接下来，动画循环会持续地调用精灵对象的 update() 方法，而该方法又会调用 runInPlace 对象的 execute() 方法，所以精灵就会原地跑步了。

提示：行为对象运用了命令模式

行为对象能够将某种命令封装起来，它是命令模式的一个应用实例。行为对象可以被执行，也可以被存放在某个队列之中，比如精灵对象所含的行为数组就是如此。有关命令模式的更多信息，请参阅：http://bit.ly/lhIa5q。

6.3.1　将多个行为组合起来

精灵含有一个行为对象数组，所以开发者可以根据需要向任何精灵对象之中添加任意数量的行为对象。精灵的 update() 方法会从数组中的第一个行为对象开始，一直遍历到最后一个对象，依次调用其 execute() 方法。

图 6-6 所示应用程序把 6.3 节讲的“原地跑步”行为与一个“让精灵从右至左移动”的行为结合起来使用。这样产生的效果就是，精灵会从屏幕右侧跑至屏幕左侧。

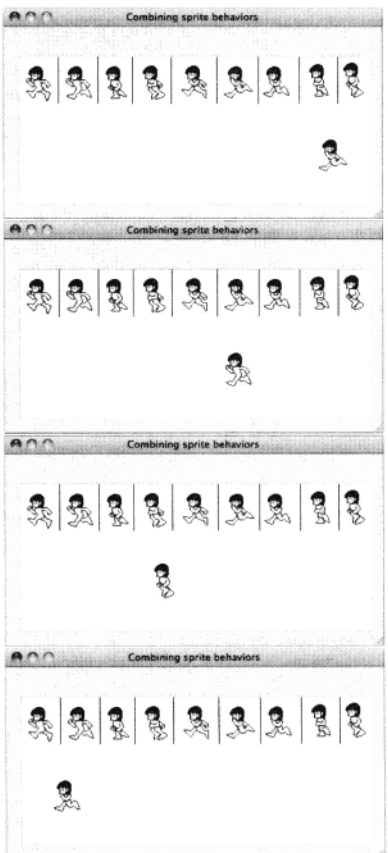


图 6-6　将多个行为对象联合起来使用，使精灵从屏幕右方跑至左方

该程序创建精灵对象所用的代码如下：

```
sprite = new Sprite('runner',
                    new SpriteSheetPainter(runnerCells),
                    [ runInPlace, moveLeftToRight ]),
```

精灵可以有任意多个行为对象，开发者在程序运行时可以直接操作 behaviors 数组来增加及移除行为对象。比如说，我们可以很容易地修改图 6-6 所示应用程序的精灵对象，向其增加一个由用户输入来触发的跳跃行为。

图 6-6 所示应用程序的 JavaScript 代码，列在了程序清单 6-9 之中。

程序清单 6-9　将多个行为联合起来使用（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    spritesheet = new Image(),
    runnerCells = [
        { left:  0,   top:  0, width:  47, height:  64 },
        { left:  55,  top:  0, width:  44, height:  64 },
        { left:  107, top:  0, width:  39, height:  64 },
        { left:  150, top:  0, width:  46, height:  64 },
        { left:  208, top:  0, width:  49, height:  64 },
        { left:  265, top:  0, width:  46, height:  64 },
        { left:  320, top:  0, width:  42, height:  64 },
        { left:  380, top:  0, width:  35, height:  64 },
        { left:  425, top:  0, width:  35, height:  64 },
    ];

    // Behaviors.................................................

    runInPlace = {
       lastAdvance: 0,
       PAGEFLIP_INTERVAL: 100,

       execute: function (sprite, context, time) {
          if (time - this.lastAdvance > this.PAGEFLIP_INTERVAL) {
             sprite.painter.advance();
             this.lastAdvance = time;
          }
       }
    },

    moveLeftToRight = {
       lastMove: 0,

       execute: function (sprite, context, time) {
         if (this.lastMove !== 0) {
            sprite.left -= sprite.velocityX *
                           ((time - this.lastMove) / 1000);

            if (sprite.left < 0) {
               sprite.left = canvas.width;
            }
         }
         this.lastMove = time;
       }
    },

    // Sprite....................................................

    sprite = new Sprite('runner', new SpriteSheetPainter(runnerCells),
                        [ runInPlace, moveLeftToRight ]);
```

```
// Functions.....................................................

function drawBackground() {
   var STEP_Y = 12,
       i = context.canvas.height;

   while(i > STEP_Y*4) {
      context.beginPath();

      context.moveTo(0, i);
      context.lineTo(context.canvas.width, i);
      context.stroke();

      i -= STEP_Y;
   }
}

// Animation....................................................

function animate(time) {
   context.clearRect(0,0,canvas.width,canvas.height);
   drawBackground();

   context.drawImage(spritesheet, 0, 0);

   sprite.update(context, time);
   sprite.paint(context);

   window.requestNextAnimationFrame(animate);
}

// Initialization...............................................

spritesheet.src = 'running-sprite-sheet.png';

spritesheet.onload = function(e) {
   context.drawImage(spritesheet, 0, 0);
};

sprite.velocityX = 50; // pixels/second
sprite.left = 200;
sprite.top = 100;

context.strokeStyle = 'lightgray';
context.lineWidth = 0.5;

window.requestNextAnimationFrame(animate);
```

6.3.2　限时触发的行为

行为对象一旦被加入精灵之中，它就会随着 update() 方法而执行。一般来说，动画循环会反复地调用 update() 方法以实现动画效果。实际上，只要某个行为对象被加入到精灵之中，那么精灵就会持续地表现出该行为，直到有人把它从精灵的行为数组中移除为止。

然而，有时候你只想让某个行为在一个特定的时间段内执行。比如说，你想让某个正在移动的物体在接受用户输入之后的那一小段时间内，表现出被推动的效果来。

图 6-7 所示应用程序使用了限时触发行为（timed behavior）对象，此种对象所封装的行为只会在给定的时间段内表现出来。每当用户点击左箭头或右箭头时，应用程序都将使小球朝着对应的方向移动 200 毫秒。

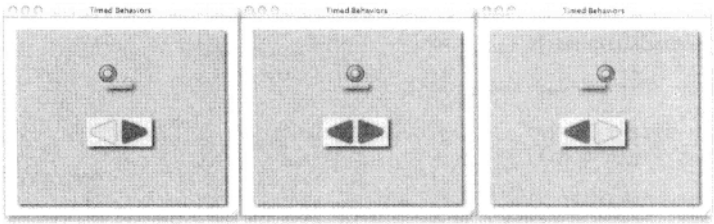


图 6-7 限时触发的行为

由于小球正在进行“基于时间的运动”，所以其速度应以“每秒所移动的像素数”来定义，确切地说，它的移动速度是每秒 110 像素。因此，200 毫秒恰好能移动 22 个像素（200 毫秒是五分之一秒，用其乘以 110 即得 22）。

小球下方的支架宽度为 44 像素，所以当它位于支架中央时，无论按下左箭头还是右箭头，都会使它滚动到支架边缘，如图 6-7 所示。如果用户将小球从支架上推落，那么程序则会重新把它放到正中位置。

当用户点击某个箭头后，程序就会启动动画计时器（详情参见本书 5.10.2 小节），并设置表示方向的标志位。然后，小球中的 moveBall 行为对象就会令其开始移动。这段代码如下：

```
var ANIMATION_DURATION = 200,
    pushAnimationTimer = new AnimationTimer(ANIMATION_DURATION),

    moveBall = {
       execute: function (sprite, context, time) {
           if (pushAnimationTimer.isRunning()) {
              if (arrow === LEFT) ball.left -= ball.velocityX / fps;
              else                ball.left += ball.velocityX / fps;

              if (isBallOnLedge()) {
                 if (pushAnimationTimer.isOver()) {
                    pushAnimationTimer.stop();
                 }
              }
              else {
                pushAnimationTimer.stop();
                ball.left = LEDGE_LEFT + LEDGE_WIDTH/2 - BALL_RADIUS;
                ball.top = LEDGE_TOP - BALL_RADIUS*2;
              }
           }
       }
     },

    ball = new Sprite('ball', painter, [ moveBall ]);
    ...
```

如果动画计时器正在运行，那么 moveBall 行为对象就根据用户所点击的箭头，令小球左移或右移。

程序为了实现“基于时间的运动”效果，需要将小球的速度（单位是每秒移动的像素数）除以动画的帧速率（单位是每秒播放的帧数），这样就可以得到小球在当前这一帧所移动的像素数了。欲详细了解“基于时间的运动”，请参阅本书 5.6 节。

在调整完小球的位置后，moveBall 行为对象的代码将会检查小球是否还在支架上。如果它还在，并且移动时间已经超过 200 毫秒了，那么就将动画计时器停止。若是它已经从支架上掉落，那么就停止动画计时器，并将小球重新放回支架中央。

在学会了限时行为对象的实现方式后，咱们来看看如何把这个概念封装成“精灵动画制作器”（sprite animator），使其适用范围更广。

6.4　精灵动画制作器

在制作精灵动画时，我们不仅经常要将精灵从一个地方移动到另一个地方，而且还会频繁地轮换精灵的图像，如图 6-8 所示。

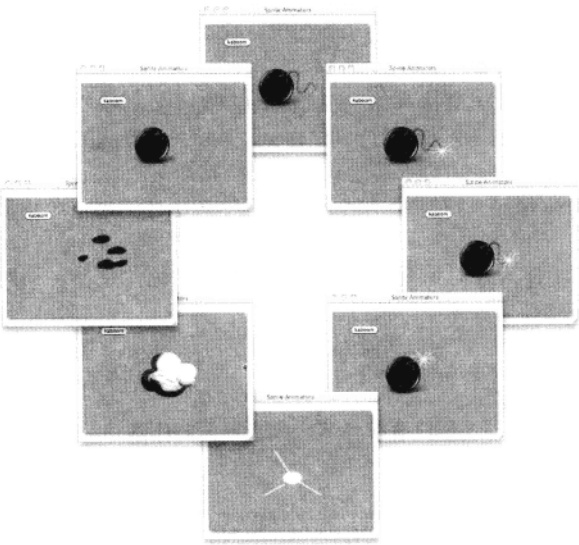


图 6-8　用两个精灵动画制作器实现的动画，其中一个表现引线燃烧阶段，另一个表现爆炸效果。从正上方起，按顺时针方向，依次为：点燃引线、炸弹爆炸、重新出现

图 6-8 所示应用程序包含一个按钮与一个精灵对象。如果点击按钮，那么应用程序就会轮换精灵的图像以做出动画效果，让它看上去像是一颗引线正在燃烧的炸弹。

当引线全部燃尽时，应用程序会使用另外一组图像序列来制作动画，将精灵显示为一个正在爆炸的炸弹。

爆炸效果播放完毕后，程序会分两步将精灵复原：首先绘制一颗没有引线的炸弹，紧接着为其配上一条引线。

图 6-9 与图 6-10 分别列出了引线燃烧阶段与爆炸阶段所使用的动画图像。

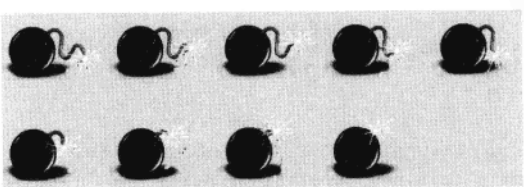


图 6-9　引线燃烧动画所用的各单元格图像（按从左至右、从上至下的顺序播放）

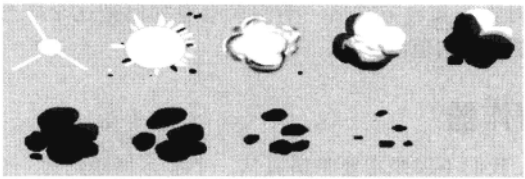


图 6-10 爆炸动画所用的各单元格图像（按从左至右、从上至下的顺序播放）

精灵动画制作器对象用于控制精灵的动画图像。程序清单 6-10 列出了 SpriteAnimator 对象的代码。

SpriteAnimator 对象含有一个精灵绘制器数组（“精灵绘制器”这个概念在本书 6.2 节中讲过），数组中的每个元素都是一个实现了 paint(sprite, context) 方法的对象，这些对象都可以绘制精灵。每个精灵对象都有一个专门负责其绘制的精灵绘制器。

精灵动画制作器对象每隔一段时间，就会从数组中按次序选出一个绘制器对象，并用其绘制某个精灵。

在创建 SpriteAnimator 对象时，要将精灵绘制器数组传给构造器，同时还可以根据需要传入一个回调函数，用于在动画播放完毕时执行。

调用 SpriteAnimator.start() 就可以开始播放动画了。该方法需要知道播放动画效果的精灵对象，以及动画持续的毫秒数。

程序清单 6-10 精灵动画制作器

```
// Constructor.......................................................

var SpriteAnimator = function (painters, elapsedCallback) {
     this.painters = painters || [];
     this.elapsedCallback = elapsedCallback;
     this.duration = 1000;
     this.startTime = 0;
     this.index = 0;
};

// Prototype........................................................

SpriteAnimator.prototype = {
   end: function (sprite, originalPainter) {
      sprite.animating = false;
      if (this.elapsedCallback) this.elapsedCallback(sprite);
      else                      sprite.painter = originalPainter;
   },

    start: function (sprite, duration) {
       var endTime = +new Date() + duration,
           period = duration / (this.painters.length),
           animator = this,
           originalPainter = sprite.painter,
           lastUpdate = 0;

      this.index = 0;
      sprite.animating = true;
      sprite.painter = this.painters[this.index];

      requestNextAnimationFrame( function spriteAnimatorAnimate(time) {
```

```
            if (time < endTime) {
               if ((time - lastUpdate) > period) {
                  sprite.painter = animator.painters[++animator.index];
                  lastUpdate = time;
               }
               requestNextAnimationFrame(spriteAnimatorAnimate);
            }
            else {
               animator.end(sprite, originalPainter);
            }
         });
      },
   };
```

为了播放动画效果，SpriteAnimator 对象的 start() 方法需要将动画持续时间与当前时间相加，以算出动画的停止时间。然后，还要算出动画的“周期”（period），也就是分配给每张动画图像的显示时间。

最后，SpriteAnimator.start() 方法调用 window.requestNextAnimationFame() 方法（5.1.3 小节所讲的这个“polyfill 式方法”，间接地实现了 HTML5 标准所要求的 requestAnimationFrame() 功能），并将负责更新精灵绘制器的函数对象传给它。如果在创建 SpriteAnimator 对象时指定了回调函数，那么 SpriteAnimator 对象就会在动画播放完毕时调用它。若是没有指定的话，那么该对象则会把调用 start() 方法之前精灵原有的那个绘制器对象复原。

程序清单 6-11 列出了图 6-8 所示应用程序的代码。

程序清单 6-11　精灵动画制作器的用法

```
var canvas = document.getElementById('canvas'),
     context = canvas.getContext('2d'),
     explosionButton = document.getElementById('explosionButton'),

     BOMB_LEFT = 100,
     BOMB_TOP = 80,
     BOMB_WIDTH = 180,
     BOMB_HEIGHT = 130,

     NUM_EXPLOSION_PAINTERS = 9,
     NUM_FUSE_PAINTERS = 9,

     // Painters.......................................................

     bombPainter = new ImagePainter('bomb.png'),
     bombNoFusePainter = new ImagePainter('bomb-no-fuse.png'),
     fuseBurningPainters = [],
     explosionPainters = [],

     // Animators......................................................

     fuseBurningAnimator = new SpriteAnimator(
        fuseBurningPainters,
        function () { bomb.painter = bombNoFusePainter; });

     explosionAnimator = new SpriteAnimator(
        explosionPainters,
        function () { bomb.painter = bombNoFusePainter; });

     // Bomb...........................................................

     bomb = new Sprite('bomb', bombPainter),
```

```
// Functions.......................................................

function resetBombNoFuse() {
    bomb.painter = bombNoFusePainter;
}

// Event handlers..................................................

explosionButton.onclick = function (e) {
   if (bomb.animating) // Not now...
      return;

   // Burn fuse for 2 seconds

   fuseBurningAnimator.start(bomb, 2000);

  // Wait for 3 seconds, then explode for 1 second

  setTimeout(function () {
     explosionAnimator.start(bomb, 1000);

     // Wait for 2 seconds, then reset to the original bomb image

     setTimeout(function () {
         bomb.painter = bombPainter;
    },  2000);
   }, 3000);
};

// Animation.......................................................

function animate(now) {
   context.clearRect(0, 0, canvas.width, canvas.height);
   bomb.paint(context);
   window.requestNextAnimationFrame(animate);
}

// Initialization.................................................

bomb.left = BOMB_LEFT;
bomb.top = BOMB_TOP;

bomb.width = BOMB_WIDTH;
bomb.height = BOMB_HEIGHT;

for (var i=0; i < NUM_FUSE_PAINTERS; ++i) {
   fuseBurningPainters.push(
      new ImagePainter('fuse-0' + i + '.png'));
}

for (var i=0; i < NUM_EXPLOSION_PAINTERS; ++i) {
   explosionPainters.push(
      new ImagePainter('explosion-0' + i + '.png'));
}

window.requestNextAnimationFrame(animate);
```

该程序创建了两个精灵动画制作器：fuseBurningAnimator 与 explosionAnimator。它们起初都有一个用于存放绘制器对象的空数组。稍后应用程序会分别用适当的图像绘制器对象来初始化这两个数组。

应用程序的大部分逻辑都包含在按钮点击事件处理器的代码中。如果炸弹精灵已经在播放动画效果了，那么该处理器直接返回，否则，它就让 fuseBurningAnimator 对象启动，并播放为时两秒的动画。当引线燃尽时，fuseBurningAnimator 对象的回调函数会将炸弹精灵的绘制器对象设置为 bombNoFusePainter，这个绘制器会把没有引线的那张炸弹图片绘制出来。

程序接下来延时 1 秒，然后播放为期两秒的炸弹爆炸动画。爆炸动画播放完毕后，explosionAnimator 对象也会把炸弹精灵的绘制器对象设置成 bombNoFusePainter，用以绘制不含引线的炸弹图片。

最终，在炸弹爆炸完 1 秒后，应用程序将炸弹精灵的绘制器对象设置为 bombPainter，此对象会把有引线但尚未点燃的那张炸弹图片绘制在屏幕上。

提示：SpriteAnimator 也可制作非线性动画

程序清单 6-10 中的 SpriteAnimator 对象会用动画的时长除以绘制器的个数，计算出每个绘制器所占据的绘制周期。对于线性动画的制作来说，这么做可以保证动画能以平稳的速度向前播放，这正是我们想要的效果。然而，还有许多动画是非线性的。比如，图 6-9 中的引线是匀速燃烧的。然而我们希望当引线变短时，它的燃烧速度看上去能比原来更快一些，所以，引线燃烧动画的播放速度应该随着进度而逐渐增高才是。

我们会在第 7 章中研究如何以非线性的方式来处理精灵的移动及动画播放。

6.5　基于精灵的动画循环

我们偶尔会像本章所讲的这样，直接绘制精灵，然而大多数情况下，开发者都是用一个基于精灵的可以复用动画循环来实现绘制的：

```
var sprites = [ new Sprite(...), ... ], // An array of sprites
    context = ...;
...
function animate(time) {
   var i;
   ...
   context.clearRect(0, 0, context.canvas.width,
                           context.canvas.height);
   drawBackground();

   for (i=0; i < sprites.length; ++i) {
      sprites.update(context, time);
   }

   for (i=0; i < sprites.length; ++i) {
      sprites.paint(context);
   }
   ...
   window.requestNextAnimationFrame(animate);
}
```

上述动画循环代码会将精灵数组遍历两次，首先更新每个精灵的逻辑，然后将它们绘制出来。

将更新与绘制分开，是刻意而为的。因为某个精灵对象的更新可能会影响到其他对象。比方说，如果你正在更新逻辑的这个精灵对象与另一个精灵相碰撞，那么这两个精灵很有可能都会由于碰撞而改变其位置。

如果将更新与绘制交错执行，那么可能会出现这种情况：精灵先被画在了某个坐标上，然后

由于另一个精灵的逻辑更新而导致其位置发生改变，如此一来，原有的那个精灵就会显示在错误的地方了。由于精灵之间可能存在着相互关联，所以必须先将所有精灵的更新逻辑都运行完毕，然后才能绘制它们。

6.6 总结

精灵是制作绚丽动画的关键要素之一。读者在本章中学到了如何封装精灵对象，如何使用精灵绘制器和精灵行为对象，以及如何将精灵的动画效果提取为可以复用的精灵动画制作器。上述对象都使得开发者能够在更高的抽象层面上编程，从而极大地简化了需要编写的代码。

下一章我们将会讲述动画制作的另一个关键要素：物理效果。

第7章
物理效果

物理效果，或与其类似的东西，通常在基于精灵的动画中发挥着一定的作用，在游戏中则更是如此。本章将会讲述在动画中广泛使用的几种基本物理效果：

- 重力
- 非线性运动
- 非线性动画

从“刺猬索尼克”[㊀]到“割绳子”[㊁]，电子游戏中的重力效果无所不在。我们先来看看这三种与重力有关的常见效果：下落、抛射体弹道运动[㊂]以及有重力参与的协运动（harmonic motion）。

协运动为弹簧、钟摆等这样的运动方式提供了数学模型，以计算物体距离标准位置的偏移量。在介绍完协运动之后，我们就来研究如何用“时间轴扭曲”技术来实现缓入、缓出、振荡及弹跳等效果。

咱们先来研究最弱的宇宙基本力，也就是重力。

提示：最弱的宇宙基本力

在迄今为止所发现的宇宙基本力中，重力是最弱的一种。已知的四种宇宙基本力（fundamental force）由强至弱分别是：强核作用力（strong nuclear）、电磁力（electromagnetic）、弱核作用力（weak nuclear）及重力（gravity）。

强核作用力用于维持原子核在原子中的稳定性，它是迄今为止最强的宇宙基本力。电磁力是像磁铁吸引物体时的那种力，它的强度大约是强核作用力的百分之一。弱核作用力是一种会引发放射性衰变（radioactive decay）的力，其强度约为电磁力的一千亿分之一。而重力的强度大概是电磁力的 1 000 000 000 000 000 000 000 000 000 000 000 000 分之一（10 的负 36 次方）。

7.1 重力

光是个奇妙的东西，它同时具有粒子性与波动性[㊃]。尽管某些实验证明[㊄]中微子[㊅]的速度比光还要快，不过在理论上，它的速度仍然是宇宙间的极限速度。水也是天赐的万灵药，它是生命之

㊀ Sonic The Hedgehog，又名“刺猬音速小子”，是由世嘉（Sega）公司推出的一系列以其吉祥物“索尼克”（Sonic）为主角的电子游戏。1991年发行首部作品后，又接连有续作登场。详情参见：http://zh.wikipedia.org/wiki/索尼克系列游戏列表。——译者注

㊁ Cut the Rope，又名“怪物吃糖果”，是ZeptoLab公司开发的一款益智游戏，于2010年由Chillingo公司发行。详情参见：http://zh.wikipedia.org/zh-cn/怪物吃糖果。此游戏有以HTML5技术开发的版本，网址是：http://www.cuttherope.ie/。——译者注

㊂ 原文为projectile trajectory，可称为“抛体运动”、“抛射体运动”，也俗称“抛物线运动”。——译者注

㊃ 又叫波粒二相性，详情参阅：http://zh.wikipedia.org/wiki/波粒二相性。——译者注

㊄ 实验详情参阅：http://zh.wikipedia.org/zh-cn/中微子#.E9.80.9F.E5.BA.A6。——译者注

㊅ Neutrino，字面意思为“微小的电中性粒子”，又译作微中子，是轻子的一种。有实验表明，中微子确实有微小的质量，但并不为零。详情参见：http://zh.wikipedia.org/zh-cn/中微子。——译者注

源，是世间少有的能够在冻结状态下膨胀其体积的物质。像光和水一样，重力也是宇宙间最为神奇的事情之一。就连金属物靠近小磁铁时受到的微弱磁力，都要比重力强好多好多倍，然而宇宙间若是没了重力，那么所有东西都不复存在了。

在现实世界中，地球表面附近的所有物体在下坠时，其加速度都是 $9.81m/s^2$，也可以表示为 $32ft/s^2$。要模拟重力加速度效果，你必须让精灵也以相同的加速度下落才行。这听起来挺容易的，在大多数情况下也确实如此。

7.1.1 物体的下落

图 7-1 所示应用程序模拟了物体的下落效果。小球一开始位于平台中部，而当用户连续多次按下左箭头后，它就会从平台掉落并下坠至 Canvas 范围外。在下落过程中，应用程序使用等式 7.1 所示的这个简单方程[⊖]来计算小球的垂直速度。

$$v_y = gt$$

等式 7.1　根据移动时间 t 与重力加速度常数 g 来计算落体的垂直速度

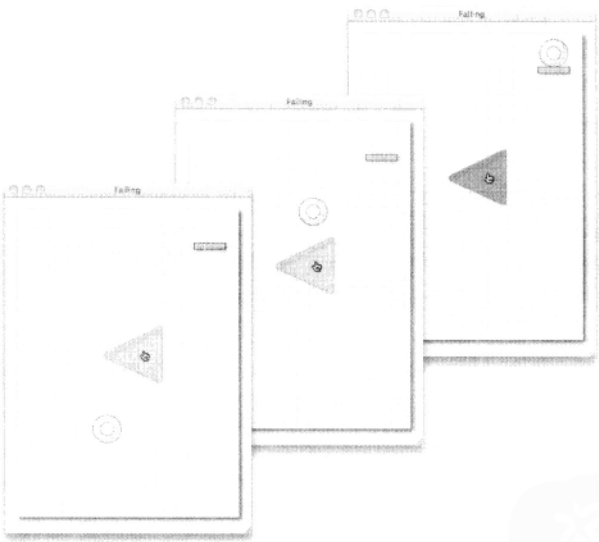


图 7-1　从平台上掉落的物体

将等式 7.1 转写为 JavaScript 代码不难，然而，难点在于重力加速度常数是按照米或英尺来度量的，并不是以像素为单位的，所以模拟重力效果时必须将米转换为像素。程序清单 7-1 列出了图 7-1 所示应用程序的实现代码。

程序先创建了名为 ball 的精灵，并向其中添加了一个用于移动其位置的行为对象。此外还创建了一个动画计时器，用以追踪动画播放时间。本书 5.10.2 小节曾详述了动画计时器的实现及用法。

⊖ 落体的速度计算公式，请参阅维基百科：http://bit.ly/jURRlf。

程序清单 7-1 物体的下落动画

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    ...

    GRAVITY_FORCE = 9.81,            // 9.81 m/s/s
    PLATFORM_HEIGHT_IN_METERS = 10, // 10 meters

    pixelsPerMeter = (canvas.height - LEDGE_TOP) /
                     PLATFORM_HEIGHT_IN_METERS,
    ...

    //moveBall is a behavior -- an object with an
    //execute(sprite, context, time) method -- that's
    //attached to the ball. When the application calls
    //ball.update(), the ball sprite executes the moveBall
    //behavior. The sprite passed to execute() is the ball.

    moveBall = {
        execute: function (sprite, context, time) {
            ...
            if (fallingAnimationTimer.isRunning()) { // Ball is falling

                // Reposition the ball at a steady pixels/second rate

                sprite.top += sprite.velocityY / fps;

                // Recalculate the ball's velocity

                sprite.velocityY = GRAVITY_FORCE *
                    (fallingAnimationTimer.getElapsedTime()/1000) *
                    pixelsPerMeter;

                if (sprite.top > canvas.height) {
                    stopFalling();
                }
            }
        }
    },
    function stopFalling() {
        fallingAnimationTimer.stop();
        ...

        ball.left = LEDGE_LEFT + LEDGE_WIDTH/2 - BALL_RADIUS;
        ball.top = LEDGE_TOP - BALL_RADIUS*2;

        ball.velocityY = 0;
    },

    ...

    // Create the animation timer and the ball sprite.

    fallingAnimationTimer = new AnimationTimer(),

    ball = new Sprite(
        'ball', // Name
        { paint: function(sprite, context) { ... } }, // Painter
        [ moveBall ]), // Behaviors
    ...
}
```

程序将平台到 canvas 底部的距离假定为 10 米，然后计算出这段距离所占据的像素数，根据这两个数值得出每米所对应的屏幕像素个数。稍后可以凭此值将小球速度从每秒移动的米数转换为每秒移动的像素数。

如果动画计时器正在运行，那么就表明小球已经开始下落了，此时 moveBall 行为对象的 execute() 方法就会用如下语句持续地更新小球位置：

```
ball.top += ball.velocityY / fps;
```

将速度（每米移动的像素数）除以动画帧速率（每秒播放的帧数），就可以得出小球在当前这一帧动画中所移动的像素数。接下来，该方法会用如下语句重新计算小球的速度：

```
ball.velocityY = GRAVITY_FORCE *
    (fallingTimer.getElapsedTime()/1000) * pixelsPerMeter;
```

上述代码将重力加速度常数（9.81m/s^2）乘以小球下落的秒数，在计算时，加速度常数单位中作为分母的“秒”，可以和下落时间中的“秒”互相消去（本书 1.11.4 小节讲述了如何根据计量单位来推导等式），于是最后所得到的速度值，其单位就是每秒所移动的米数。而应用程序中，定义小球每秒移动速度所用的单位是像素，所以，还要将这个值再乘以 pixelsPerMeter，才能把它换算成每秒所移动的像素数。

提示：用精灵对象来简化物理效果的演示代码

本章的范例程序利用了第 6 章所实现的精灵对象，这样的话，我们就能把精力集中到范例中的物理效果之上，而不会分心于动画的实现细节。与图 7-1 中的应用程序一样，本章范例代码的重点是实现诸如程序清单 7-1 中 moveBall 那样的精灵行为对象。

提示：摩擦力

一个物体假如总是位于地表附近，那么，它就一直会被指向地心的重力所吸引。

与此类似，假如物体总是在某个均匀表面上移动的话，那么摩擦力也是会一直存在的。然而，与总是指向地心的重力不同，摩擦力的方向与物体移动方向是相反的。

大家在程序清单 7-1 中可以看到，如何根据重力加速度来计算精灵当前的移动速度。同理，我们也可以根据摩擦力对物体的影响来修正其速度。9.3.3 小节将会讲述如何在弹珠台游戏中将摩擦力因素考虑进来。

7.1.2 抛射体弹道运动

上一小节讲了根据重力加速度来修正物体的垂直下落速度，这一小节我们再把水平方向的运动也考虑进来，这样的话就可以模拟抛射体的弹道了。

图 7-2 所示应用程序是一个小游戏：玩家要将小球投至桶中。用户在移动鼠标时，应用程序总是会画出一条连接小球中心与鼠标光标的导线，用以表示投掷小球的角度及速度。导线越长，投掷小球的力度就越大。

该应用程序除了会随着用户鼠标的移动而持续绘制导线以外，它还会不停地更新信息面板上显示的发射速度与角度值。

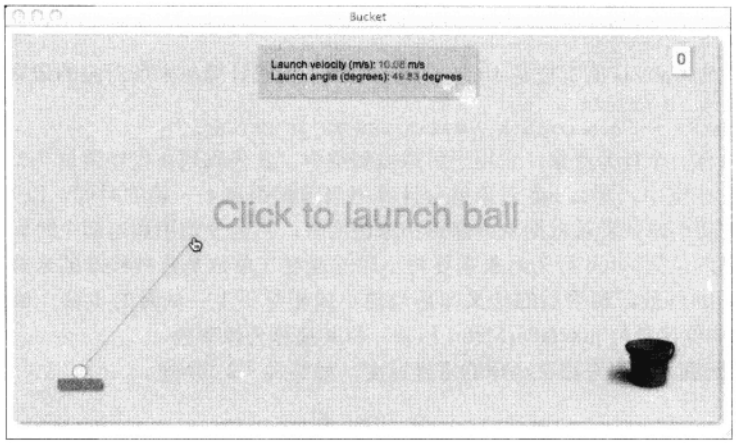


图 7-2　将小球投至桶中的游戏

程序右上角显示玩家当前的分数。如果小球在未离开 canvas 区域的情况下被投入桶中，那么玩家得两分，如图 7-3 上方截图所示。要是像底部截图所描述的那样，小球先被抛离 canvas 区域，然后又落入桶中，那么玩家则得三分。

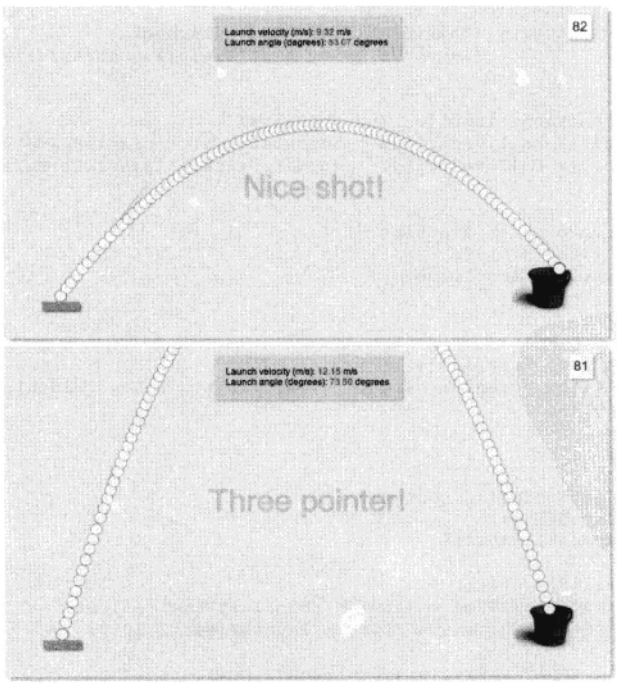


图 7-3　两分球（顶部截图）与三分球（底部截图）的判定标准示意图

该程序通过如下语句来创建表示小球的精灵对象：

```
ball = new Sprite('ball', ballPainter, [ lob ]),
```

这个游戏假定 Canvas 的宽度是 10 米，然后用如下代码计算每米所对应的像素数：

```
ARENA_LENGTH_IN_METERS = 10,
pixelsPerMeter = canvas.width / ARENA_LENGTH_IN_METERS,
```

小球精灵只有一个行为对象，它用于将球抛到空中，其代码列在程序清单 7-2 里面。

如果小球正在空中，那么 lob 对象就会计算出当前帧距离上一帧的时间，以及小球的飞行总时长。稍后在更新小球位置及根据重力修正小球速度时，将会分别用到这两个时间值。

要更新小球的位置，lob 行为对象需要将小球的速度（单位是每秒移动的米数，m/s）乘以当前帧距离上一帧的秒数。相乘后的结果即是当前小球相对于上一帧时的位移，单位是米。最后，lob 对象需要将该位移量与 pixelsPerMeter 相乘，将其换算为像素数。

lob 对象根据重力因素来修正小球的垂直速度，如等式 7.2 [⊖]所述。

$$v_y = v_{y0} - gt$$

等式 7.2　抛体垂直速度的修正公式

程序清单 7-2　用来封装小球抛掷动作的行为对象

```
lob = {
    lastTime: 0,
    GRAVITY_FORCE: 9.81, // m/s/s

    applyGravity: function (elapsed) {
        ball.velocityY = (this.GRAVITY_FORCE * elapsed) -
                         (launchVelocity * Math.sin(launchAngle));
    },

    updateBallPosition: function (updateDelta) {
        ball.left+= ball.velocityX * (updateDelta) * pixelsPerMeter;
        ball.top += ball.velocityY * (updateDelta) * pixelsPerMeter;
    }

    checkForThreePointer: function () {
        if (ball.top < 0) {
            threePointer = true;
        }
    },

    checkBallBounds: function () {
        if (ball.top > canvas.height || ball.left > canvas.width){
            reset();
        }
    },

    execute: function(ball, context, time) {
        var updateDelta,
            elapsedFlightTime;

        if (ballInFlight) {
            elapsedFrameTime = (time - this.lastTime) / 1000;
            elapsedFlightTime = (time - launchTime) / 1000;

            this.applyGravity(elapsedFlightTime);
```

⊖ 抛射体的弹道计算公式详见：http://bit.ly/lwNcox。

```
            this.updateBallPosition(elapsedFrameTime);
            this.checkForThreePointer();
            this.checkBallBounds();
        }
        this.lastTime = time;
    }
},
```

请大家注意：用于计算落体垂直速度的等式 7.1，与根据重力修正抛体垂直速度的等式 7.2 很相似。其区别在于，后者要将抛体的初始垂直速度（vy0）考虑进来。

程序清单 7-3 与程序清单 7-4 分别列出了图 7-2 所示应用程序的 HTML 代码与 JavaScript 代码。这个小球投掷游戏，除了能够演示抛射体的运动轨迹外，还具备了很多专业级游戏所含的要素，诸如计分牌（scoreboard）、游戏信息面板，以及显示用户得分的专属画面等等。

程序清单 7-3　小球投掷游戏的 HTML 代码

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bucket</title>

        <style>
            output {
                color: blue;
            }

            .floatingControls {
                background: rgba(0,0,0,0.1);
                border: thin solid skyblue;
                -webkit-box-shadow: rgba(0,0,0,0.3) 2px 2px 4px;
                -moz-box-shadow: rgba(100,140,230,0.5) 2px 2px 6px;
                box-shadow: rgba(100,140,230,0.5) 2px 2px 6px;
                padding: 15px;
                font: 12px Arial;
            }

            #canvas {
                background: skyblue;
                -webkit-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
                -moz-box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
                box-shadow: 4px 4px 8px rgba(0,0,0,0.5);
                cursor: pointer;
            }

            #scoreboard {
                background: rgba(255,255,255,0.5);
                position: absolute;
                left: 755px;
                top: 20px;
                color: blue;
                font-size: 18px;
                padding: 5px;
            }

            #controls {
                position: absolute;
                left: 285px;
                top: 20px;
            }
```

```
        </style>
    </head>
    <body>
        <canvas id='canvas' width='800' height='450'>
            Canvas not supported
        </canvas>
        <div id='scoreboard' class='floatingControls'>0</div>

        <div id='controls' class='floatingControls'>
            Launch velocity (m/s):
            <output id='launchVelocityOutput'></output> m/s<br/>

            Launch angle (degrees):
            <output id='launchAngleOutput'></output> degrees<br/>
        </div>

        <script src = 'requestNextAnimationFrame.js'></script>
        <script src = 'sprites.js'></script>
        <script src = 'example.js'></script>
    </body>
</html>
```

除了 canvas 元素外，该游戏的 HTML 代码又创建了两个 DIV 元素，分别是 scoreboard 和 controls。程序使用 CSS 代码将 scoreboard 放在 canvas 的右上角，将 controls 放在 canvas 的正上方。

以上 HTML 代码还引入了三个 JavaScript 文件。首先引入的 requestNextAnimationFrame.js 文件含有“polyfill 式”方法 requestNextAnimationFrame() 的实现代码，本书 5.1.3 小节曾讲述过此方法。接下来引入的 sprites.js 文件则包含第 6 章所述精灵对象的实现代码。最后，HTML 代码将该程序自身所用的 JavaScript 代码也引了进来，这段代码如程序清单 7-4 所示。

提示：“polyfill 式方法” requestNextAnimationFrame()

本章范例程序用到了 5.1.3 小节所述名为 requestNextAnimationFrame() 的“polyfill 式”方法，该方法能够实现 HTML5 规范书中所要求的 requestAnimationFrame() 功能，而且其操作流程与规范中定义的标准方法一致，都会在浏览器将要绘制下一帧动画时，预先通知经由参数传进来的那个回调函数对象。

程序清单 7-4　小球投掷游戏的 JavaScript 代码

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    scoreboard = document.getElementById('scoreboard'),
    launchAngleOutput = document.getElementById('launchAngleOutput'),
    launchVelocityOutput =
        document.getElementById('launchVelocityOutput'),

    elapsedTime = undefined,
    launchTime = undefined,

    score = 0,
    lastScore = 0,
    lastMouse = { left: 0, top: 0 },

    threePointer = false,
    needInstructions = true,

    LAUNCHPAD_X = 50,
```

```
LAUNCHPAD_Y = context.canvas.height-50,
LAUNCHPAD_WIDTH = 50,
LAUNCHPAD_HEIGHT = 12,
BALL_RADIUS = 8,
ARENA_LENGTH_IN_METERS = 10,
INITIAL_LAUNCH_ANGLE = Math.PI/4,

launchAngle = INITIAL_LAUNCH_ANGLE,
pixelsPerMeter = canvas.width / ARENA_LENGTH_IN_METERS,

// Launch pad.................................................

launchPadPainter = {
    LAUNCHPAD_FILL_STYLE: 'rgb(100,140,230)',

    paint: function (ledge, context) {
        context.save();
        context.fillStyle = this.LAUNCHPAD_FILL_STYLE;
        context.fillRect(LAUNCHPAD_X, LAUNCHPAD_Y,
                         LAUNCHPAD_WIDTH, LAUNCHPAD_HEIGHT);
        context.restore();
    }
},
launchPad = new Sprite('launchPad', launchPadPainter),

// Ball.......................................................

ballPainter = {
    BALL_FILL_STYLE: 'rgb(255,255,0)',
    BALL_STROKE_STYLE: 'rgb(0,0,0,0.4)',

    paint: function (ball,context) {
        context.save();
        context.shadowColor= undefined;
        context.lineWidth =2;
        context.fillStyle =this.BALL_FILL_STYLE;
        context.strokeStyle= this.BALL_STROKE_STYLE;

        context.beginPath();
        context.arc(ball.left, ball.top,
                    ball.radius, 0, Math.PI*2, false);

        context.clip();
        context.fill();
        context.stroke();
        context.restore();
    }
 },
// Lob behavior...............................................

lob = {
    lastTime: 0,
    GRAVITY_FORCE: 9.81, // m/s/s

    applyGravity: function (elapsed) {
        ball.velocityY = (this.GRAVITY_FORCE * elapsed) -
                         (launchVelocity * Math.sin(launchAngle));
    },

    updateBallPosition: function (updateDelta) {
        ball.left +=
```

```
            ball.velocityX * (updateDelta) * pixelsPerMeter;
        ball.top +=
            ball.velocityY * (updateDelta) * pixelsPerMeter;
    },

    checkForThreePointer: function () {
        if (ball.top < 0) {
            threePointer = true;
        }
    },

    checkBallBounds: function () {
        if (ball.top > canvas.height || ball.left > canvas.width){
            reset();
        }
    },
    execute: function (ball, context, time) {
        var updateDelta,
            elapsedFlightTime;

        if (ballInFlight) {
            elapsedFrameTime = (time - this.lastTime)/1000;
            elapsedFlightTime = (time - launchTime)/1000;

            this.applyGravity(elapsedFlightTime);
            this.updateBallPosition(elapsedFrameTime);
            this.checkForThreePointer();
            this.checkBallBounds();
        }
        this.lastTime = time;
    }
},

ball = new Sprite('ball', ballPainter, [ lob ]),
ballInFlight = false,

// Bucket.......................................................
catchBall = {
    ballInBucket: function() {
        return ball.left > bucket.left + bucket.width/2 &&
               ball.left < bucket.left + bucket.width   &&
               ball.top > bucket.top && ball.top <
               bucket.top + bucket.height/3;
    },

    adjustScore: function() {
        if (threePointer) lastScore = 3;
        else              lastScore = 2;

        score += lastScore;
        scoreboard.innerText = score;
    },

    execute: function (bucket, context, time) {
        if (ballInFlight && this.ballInBucket()) {
            reset();
            this.adjustScore();
        }
    }
},

BUCKET_X = 668,
```

```
    BUCKET_Y = canvas.height - 100,
    bucketImage = new Image(),

    bucket = new Sprite('bucket',
        {
            paint: function (sprite, context) {
            context.drawImage(bucketImage, BUCKET_X, BUCKET_Y);
            }
        },
            [ catchBall ]
    );

// Functions.........................................................

function windowToCanvas(x, y) {
    var bbox = canvas.getBoundingClientRect();

    return { x: x - bbox.left * (canvas.width / bbox.width),
             y: y - bbox.top * (canvas.height / bbox.height)
           };
}

function reset() {
    ball.left = LAUNCHPAD_X + LAUNCHPAD_WIDTH/2;
    ball.top = LAUNCHPAD_Y - ball.height/2;
    ball.velocityX = 0;
    ball.velocityY = 0;
    ballInFlight = false;
    needInstructions = false;
    lastScore = 0;
}
function showText(text) {
    var metrics;

    context.font = '42px Helvetica';
    metrics = context.measureText(text);

    context.save();
    context.shadowColor = undefined;
    context.strokeStyle = 'rgb(80,120,210)';
    context.fillStyle = 'rgba(100,140,230,0.5)';

    context.fillText(text,
                     canvas.width/2 - metrics.width/2,
                     canvas.height/2);

    context.strokeText(text,
                       canvas.width/2 - metrics.width/2,
                       canvas.height/2);
    context.restore();
}

function drawGuidewire() {
    context.moveTo(ball.left, ball.top);
    context.lineTo(lastMouse.left, lastMouse.top);
    context.stroke();
};

function updateBackgroundText() {
    if (lastScore == 3)          showText('Three pointer!');
    else if (lastScore == 2)     showText('Nice shot!');
    else if (needInstructions)   showText('Click to launch ball');
};
```

```
function resetScoreLater() {
    setTimeout(function () {
        lastScore = 0;
    }, 1000);
};

function updateSprites(time) {
    bucket.update(context, time);
    launchPad.update(context, time);
    ball.update(context, time);
}

function paintSprites() {
    launchPad.paint(context);
    bucket.paint(context);
    ball.paint(context);
}

// Event handlers.................................................

canvas.onmousedown = function(e) {
    var rect;

    e.preventDefault();

    if ( ! ballInFlight) {
        ball.velocityX = launchVelocity * Math.cos(launchAngle);
        ball.velocityY = launchVelocity * Math.sin(launchAngle);
        ballInFlight = true;
        threePointer = false;
        launchTime = +new Date();
    }
};

canvas.onmousemove = function (e) {
    var rect;

    e.preventDefault();

    if ( ! ballInFlight) {
        loc = windowToCanvas(e.clientX, e.clientY);
        lastMouse.left = loc.x;
        lastMouse.top = loc.y;

        deltaX = Math.abs(lastMouse.left - ball.left);
        deltaY = Math.abs(lastMouse.top - ball.top);

        launchAngle =
            Math.atan(parseFloat(deltaY) / parseFloat(deltaX));

        launchVelocity =
            4 * deltaY / Math.sin (launchAngle) / pixelsPerMeter;

        launchVelocityOutput.innerText = launchVelocity.toFixed(2);
        launchAngleOutput.innerText =
            (launchAngle * 180/Math.PI).toFixed(2);
    }
};

// Animation loop.................................................

function animate(time) {
```

```
    elapsedTime = (time - launchTime) / 1000;
    context.clearRect(0, 0, canvas.width, canvas.height);

    if (!ballInFlight) {
        drawGuidewire();
        updateBackgroundText();

        if (lastScore !== 0) { // Just scored
            resetScoreLater();
        }
    }

    updateSprites(time);
    paintSprites();

    window.requestNextAnimationFrame(animate);
}

// Initialization.....................................................

ball.width = BALL_RADIUS*2;
ball.height = ball.width;
ball.left = LAUNCHPAD_X + LAUNCHPAD_WIDTH/2;
ball.top = LAUNCHPAD_Y - ball.height/2;
ball.radius = BALL_RADIUS;

context.lineWidth = 0.5;
context.strokeStyle = 'rgba(0,0,0,0.5)';
context.shadowColor = 'rgba(0,0,0,0.5)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur = 4;
context.stroke();

bucketImage.src = 'bucket.png';
bucketImage.onload = function (e) {
    bucket.left = BUCKET_X;
    bucket.top = BUCKET_Y;
    bucket.width = bucketImage.width;
    bucket.height = bucketImage.height;
};

window.requestNextAnimationFrame(animate);
```

7.1.3 钟摆运动

在重力这一节的最后，我们来讲讲如何模拟受重力影响的钟摆运动效果。

上一小节中，那个简单的公式可以算出落体或抛射体的速度。该公式是线性的，也就是说物体的移动速度是与时间成正比的。

然而，钟摆是非线性的运动系统。所以在计算其速度时，不能像原来那样简单地用移动时间乘以重力加速度常数，而是要用等式 7.3 所述公式，根据钟摆动画的持续时间来计算其角度[⊖]。

$$\theta=\theta_0\times\cos(\sqrt{g/1}\times t)$$

等式 7.3　简单钟摆运动的角度计算公式

⊖ 有关钟摆运动的计算公式，请参阅维基百科：http://bit.ly/mvpGu7。

在等式 7.3 中，θ 表示钟摆当前的角度，θ_0 是钟摆的初始角度，g 表示重力加速度，l 表示摆杆长度。值得注意的是，公式中并没有将悬在钟摆下端的物体重量考虑在内，它的重量并不影响钟摆的角度。

图 7-4 所示应用程序模拟了钟摆运动效果。

该程序首先创建了代表钟摆的精灵对象：

```
pendulum = new Sprite('pendulum', pendulumPainter, [ swing ]),
```

这个钟摆精灵只有一个表示摆动行为的 swing 对象，该对象使用等式 7.3 来计算钟摆的角度，代码如下：

```
swing = {
    GRAVITY_FORCE : 32, // 32 ft/s/s,
    ROD_LENGTH : 0.8,   // 0.8 ft

    execute : function(pendulum, context, time) {
        pendulum.angle =
            pendulum.initialAngle * Math.cos(
                Math.sqrt(this.GRAVITY_FORCE / this.ROD_LENGTH) *
                elapsedTime);

        pendulum.weightX =
            pendulum.x + Math.sin(pendulum.angle) * pendulum.rodLength;

        pendulum.weightY =
            pendulum.y + Math.cos(pendulum.angle) * pendulum.rodLength;
    }
};
```

得知钟摆的角度后，swing 行为对象就可以据此算出摆锤的位置了。

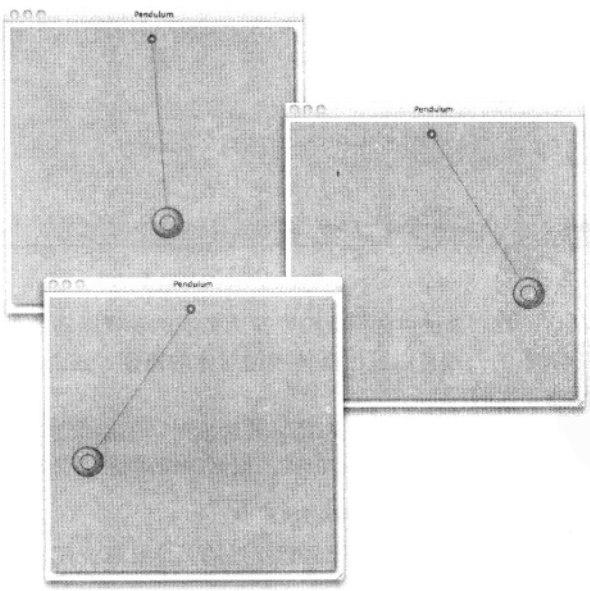


图 7-4 非线性的钟摆运动

图 7-4 所示应用程序的 JavaScript 代码列在程序清单 7-5 之中。

程序清单 7-5　钟摆运动

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    elapsedTime = undefined,
    startTime = undefined,

    PIVOT_Y = 20,
    PIVOT_RADIUS = 7,
    WEIGHT_RADIUS = 25,
    INITIAL_ANGLE = Math.PI/5,
    ROD_LENGTH_IN_PIXELS = 300,

    // Pendulum painter.............................................

pendulumPainter = {
    PIVOT_FILL_STYLE:   'rgba(0,0,0,0.2)',
    WEIGHT_SHADOW_COLOR:'rgb(0,0,0)',
    PIVOT_SHADOW_COLOR: 'rgb(255,255,0)',
    STROKE_COLOR:       'rgb(100,100,195)',

    paint: function (pendulum, context) {
        this.drawPivot(pendulum);
        this.drawRod(pendulum);
        this.drawWeight(pendulum, context);
    },

    drawWeight: function (pendulum, context) {
        context.save();
        context.beginPath();
        context.arc(pendulum.weightX, pendulum.weightY,
                    pendulum.weightRadius, 0, Math.PI*2, false);
        context.clip();

        context.shadowColor = this.WEIGHT_SHADOW_COLOR;
        context.shadowOffsetX = -4;
        context.shadowOffsetY = -4;
        context.shadowBlur = 8;
        context.lineWidth = 2;

        context.strokeStyle = this.STROKE_COLOR;
        context.stroke();
        context.beginPath();

        context.arc(pendulum.weightX, pendulum.weightY,
                    pendulum.weightRadius/2, 0, Math.PI*2, false);
        context.clip();

        context.shadowColor = this.PIVOT_SHADOW_COLOR;
        context.shadowOffsetX = -4;
        context.shadowOffsetY = -4;
        context.shadowBlur = 8;

        context.stroke();
         context.restore();
    },

    drawPivot: function (pendulum) {
        context.save();
        context.beginPath();
        context.shadowColor = undefined;
        context.fillStyle = 'white';
        context.arc(pendulum.x + pendulum.pivotRadius,
                    pendulum.y, pendulum.pivotRadius/2,
                    0, Math.PI*2, false);
```

```
        context.fill();
        context.stroke();

        context.beginPath();
        context.fillStyle = this.PIVOT_FILL_STYLE;
        context.arc(pendulum.x + pendulum.pivotRadius,
                    pendulum.y, pendulum.pivotRadius,
                    0, Math.PI*2, false);
        context.fill();
        context.stroke();
        context.restore();
        },

        drawRod: function (pendulum) {
            context.beginPath();

            context.moveTo(
                pendulum.x + pendulum.pivotRadius +
                pendulum.pivotRadius*Math.sin(pendulum.angle),

                pendulum.y + pendulum.pivotRadius *
                    Math.cos(pendulum.angle)
            );

            context.lineTo(
                pendulum.weightX -
                    pendulum.weightRadius*Math.sin(pendulum.angle),

                pendulum.weightY -
                    pendulum.weightRadius*Math.cos(pendulum.angle)
            );

            context.stroke();
        }
        },

        // Swing behavior.............................................

         swing = {
            //For a gravity force of 32 ft/s/s, and a rod
            //length of 0.8 ft (about 10 inches), the time period
            //for the pendulum is about one second. Make the rod
            //longer for a longer time period.

            GRAVITY_FORCE: 32,  // 32 ft/s/s,
            ROD_LENGTH: 0.8,    // 0.8 ft

            execute: function(pendulum, context, time) {
                pendulum.angle = pendulum.initialAngle * Math.cos(
                    Math.sqrt(this.GRAVITY_FORCE/this.ROD_LENGTH) *
                    elapsedTime);

                pendulum.weightX = pendulum.x +
                    Math.sin(pendulum.angle) * pendulum.rodLength;

                pendulum.weightY = pendulum.y +
                    Math.cos(pendulum.angle) * pendulum.rodLength;
            }
        };

        // Pendlulum..................................................

        pendulum = new Sprite('pendulum', pendulumPainter, [ swing ]);

    // Animation loop.................................................

    function animate(time) {
        elapsedTime = (time - startTime) / 1000;
        context.clearRect(0, 0, canvas.width, canvas.height);
        pendulum.update(context, time);
        pendulum.paint(context);
```

```
    window.requestNextAnimationFrame(animate);
}

// Initialization.................................................

pendulum.x = canvas.width/2;
pendulum.y = PIVOT_Y;
pendulum.weightRadius =WEIGHT_RADIUS;
pendulum.pivotRadius =PIVOT_RADIUS;
pendulum.initialAngle =INITIAL_ANGLE;
pendulum.angle=INITIAL_ANGLE;
pendulum.rodLength=ROD_LENGTH_IN_PIXELS;

context.lineWidth = 0.5;
context.strokeStyle = 'rgba(0,0,0,0.5)';
context.shadowColor = 'rgba(0,0,0,0.5)';
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.shadowBlur = 4;
context.stroke();

startTime = + new Date();
animate(startTime);
```

7.2 时间轴扭曲

7.1.3 小节讲了一个用于计算钟摆角度的非线性公式。实际生活中，非线性的运动系统很普遍，例如弹簧、钟摆、弹跳的小球等等，都是如此。所以，在动画中模拟非线性的运动系统是一项重要的技术。

7.1.3 小节中所说的非线性，指的是钟摆的运动方式，除此以外，动画的其他方面也可以用非线性的方式呈现出来。例如，我们要模拟某人脸红的表情，那么就要描绘出面部突然泛红而后又淡淡褪去的过程。在这种情况下，以非线性方式变化的因素是颜色，而非运动物体的位置。

从本质上讲，我们要描述的是那些与时间呈非线性变化的属性，不论它指的是位置、颜色，还是其他特征。这样一来，我们要统一处理的问题就应该是时间，而不是发生变化的那些个属性。

5.10.2 小节中实现了一个简单的 AnimationTimer 对象，它可以用来控制动画播放。其用法如下：

```
var ANIMATION_DURATION = 1000, // One second

    // Create an animation timer
    animationTimer = new AnimationTimer(ANIMATION_DURATION);
    ...

function animate() {
    var elapsed;
    ...
    if ( ! animationTimer.isOver()) {

            // Update the animation, based on the
            // animation timer's elapsed time

            updateAnimation(animationTimer.getElapsedTime());
            ...
    }

    // Keep the animation going

    requestNextAnimationFrame(animate);
}
...
animationTimer.start();              // Start the animation timer
requestNextAnimationFrame(animate); // Start the animation
```

创建好动画计时器之后，就启动它。在动画播放完毕之前，可以周期性地获取动画当前的播放时间，并据此来更新相应的动画内容。

动画计时器并无特殊之处，它们就是一种能将动画播放时间封装起来的对象，我们可以用其判断动画是否结束。

动画计时器真正派上用场的地方在于：你可以写一份 AnimationTimer.getElapsedTime() 方法的实现代码，让其返回一个与实际播放时间不同的值。这样做就可以实现“时间轴扭曲”（warp time）效果了。

例如，我们可以实现一个 getElapsedTime() 方法，一开始，让它返回大大低于实际播放时间的值。然后随着动画的播放，我们稳步地减少该方法返回值与实际播放时间的差量。如此一来，动画在时间轴上起初会播放地非常缓慢，随后则逐渐提速。

这种“起初缓慢，逐渐提速”的算法就叫做缓入效果（ease-in effect），如图 7-5 所示。

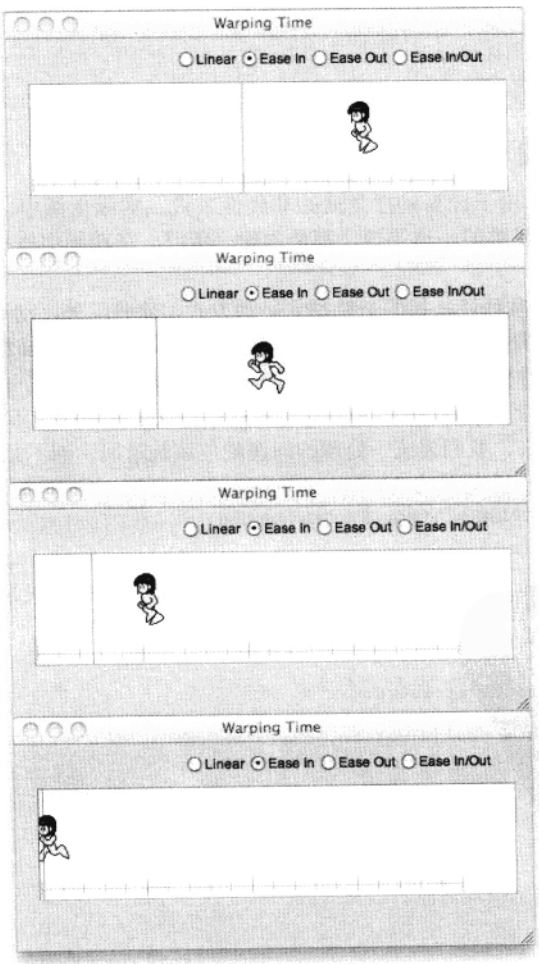


图 7-5 缓入运动效果

图 7-5 所示应用程序中的精灵对象，会在一系列图像之中轮换地选择一张来显示，这样看上去就会有跑动的效果了。此外，程序还会让精灵从右至左移动，而不是让它在原地跑步。

应用程序根据动画的实际播放时间来移动那条垂直的线段，使用经由 AnimationTimer.getElapsedTime() 方法扭曲过的动画播放时间值来计算精灵当前位置。

对于图 7-5 所示的缓入运动效果来说，一开始精灵会落在标准时间线的后方，然而随着应用程序的执行，二者的间距也将逐渐缩小，到动画播放完毕时，两者就会重合起来。

将图 7-5 所示缓入运动与图 7-6 所示的线性运动做个比较，可以看出，精灵在进行线性运动时，总是以恒定的速度随着标准时间线一起移动。

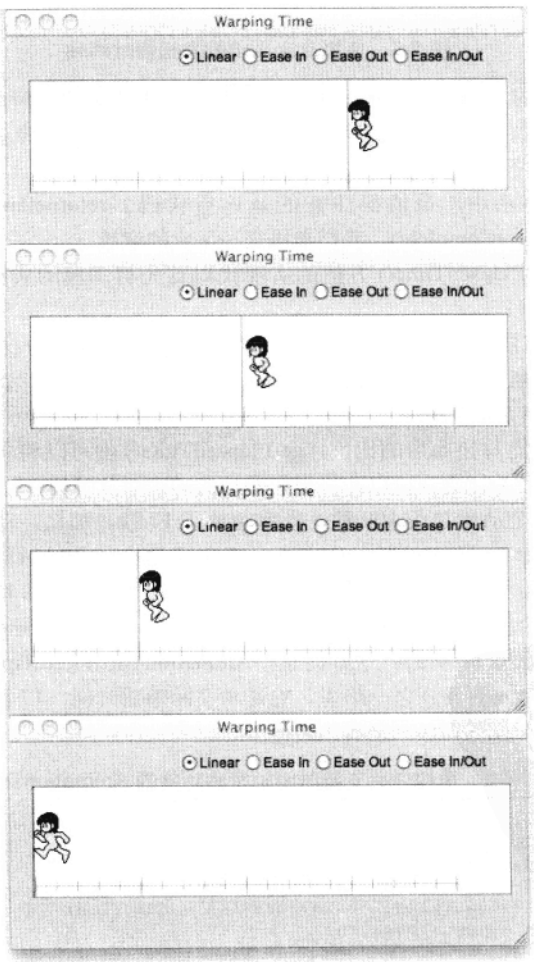


图 7-6 线性运动效果

除了图 7-5 所示的缓入效果之外，我们还可以用时间轴扭曲技术创建出很多效果来，常见的

有缓出（ease out）、缓入缓出（ease in/out）、弹簧运动（elastic）、弹跳运动（bounce）等。7.4 节将会讲述如何实现图 7-5 所示应用程序的各种运动效果。

要扭曲时间轴，就得按照每种特效所需的时间值来修改 AnimationTimer. getElapsedTime() 方法的行为。其中一种做法是：由开发者提供一份时间扭曲函数，作为 AnimationTimer. getElapsedTime() 返回动画播放时间值的依据，如图 7-7 所示。

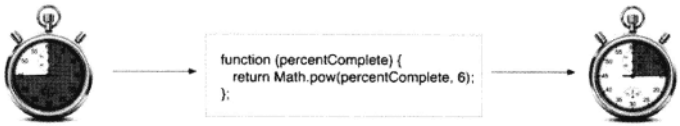


图 7-7 使用自定义函数来扭曲时间轴

程序清单 7-6 实现的 AnimationTimer 对象可以让开发者指定扭曲时间轴所用的函数。当调用 getElapsedTime() 函数时，动画计时器对象会将实际的动画播放时间传递给开发者自定义的时间扭曲函数。

getElapsedTime() 方法中，最值得注意的是这行代码：returnelapsedTime * (this.timeWarp (percentComplete) / percentComplete)。我们来研究一下它的意思。

AnimationTimer.getElapsedTime() 方法的实现代码首先将动画的实际播放时间除以总长度，算出当前播放进度百分比。

getElapsedTime() 方法把介于 0.0 ～ 1.0 之间的动画播放进度百分比值，传给开发者在创建 AnimationTimer 对象时所提供的时间扭曲函数。该函数通常会返回一个扭曲后的播放进度百分比，它的值也位于 0.0 ～ 1.0 之间。

有了实际播放百分比与扭曲后的比值，getElapsedTime() 就可以将实际动画播放时间乘以扭曲后的比值与实际播放百分比之商，以此得出扭曲后的动画播放时间，并将其返回。

举例来说，假设某个动画总长 100 秒，当前播放了 17 秒，那么，动画的实际播放百分比就是 17%。AnimationTimer.getElapsedTime() 方法会将 0.17 这个值传给时间扭曲函数。假如这个扭曲函数将传入的参数值取平方后返回，那么返回值就是 0.0289，这意味着 17% 的播放进度在扭曲后就变成了 2.89%，这个扭曲后的动画播放百分比（2.89%）还不及实际播放进度（17%）的五分之一（0.0289/0.17=0.17，0.17<0.2）。这样的话，AnimationTimer.getElapsedTime() 方法就会返回 2.89，该值比实际播放时间的五分之一略少，它是将实际播放时间（17 秒）乘以“扭曲后的比值（2.89%）与实际播放百分比（17%）之商”而得出的。

程序清单 7-6 重构之后支持时间轴扭曲功能的 AnimationTimer 对象

```
// Constructor.......................................................

AnimationTimer = function (duration, timeWarp) {
    if (timeWarp !== undefined) this.timeWarp = timeWarp;
    if (duration !== undefined) this.duration = duration;
    this.stopwatch = new Stopwatch();

};

// Prototype........................................................

AnimationTimer.prototype = {
```

```
    start: function () {
        this.stopwatch.start();
    },

    stop: function () {
        this.stopwatch.stop();
    },

    getElapsedTime: function () {
        var elapsedTime = this.stopwatch.getElapsedTime(),
            percentComplete = elapsedTime / this.duration;

        if (!this.stopwatch.running)     return undefined;
        if (this.timeWarp == undefined) return elapsedTime;

        return elapsedTime *
            (this.timeWarp(percentComplete) / percentComplete);
    },

    isRunning: function() {
        return this.stopwatch.running;
    },

    isOver: function () {
        return this.stopwatch.getElapsedTime() > this.duration;
    },
};
```

这个带有时间轴扭曲功能的动画计时器对象，其用法如下：

```
var ANIMATION_DURATION = 1000, // One second
    animation = new AnimationTimer(ANIMATION_DURATION,
        function (percentComplete) {
            return Math.pow(percentComplete, 2);
        });
    ...

    function animate() { // Repeatedly called from animation loop
        ...
        if ( ! animation.isOver()) {
            elapsed = animation.getElapsedTime();

            // Update the animation, based on the elapsed time
            update(elapsed);
        }
        ...
    }
    ...
```

在上述程序清单中，时间轴扭曲函数会将实际播放百分比取平方，这样就可以实现刚才所说的那种缓入效果了。当参数值很小时，平方后的值也很小，而随着参数值的提升，其平方值也会越来越接近原值。比如说 0.2 的平方是 0.04，它与 0.2 相比，显得非常小，而 0.9 的平方则是 0.81，该值已经与 0.9 十分接近了。运用这种时间扭曲函数，就可以让动画一开始沿着时间轴缓缓播放，然后逐渐地提速。

7.3 时间轴扭曲函数

程序清单 7-6 列出的 AnimationTimer 对象中有一些内建的时间扭曲函数，它们列在程序清单

7-7 之中。

开发者可以使用程序清单 7-7 列出的那些方法来实现时间轴扭曲效果。例如，若要实现缓入效果，则可以编写如下代码：

```
var ANIMATION_DURATION = 1000, // One second
    animation = new AnimationTimer(ANIMATION_DURATION,
                                   AnimationTimer.makeEaseIn(1));
    ...
```

在上述程序清单中，传递给 AnimationTimer.makeEaseIn() 方法的参数用来控制扭曲效果的强度，它的意义将会在下一节中演示。本节我们先来逐个看看程序清单 7-7 所实现的这些时间轴扭曲效果函数。

程序清单 7-7　时间轴扭曲函数

```
var DEFAULT_ELASTIC_PASSES = 3;

AnimationTimer.makeEaseIn = function (strength) {
    return function (percentComplete) {
        return Math.pow(percentComplete, strength*2);
    };
};

AnimationTimer.makeEaseOut = function (strength) {
    return function (percentComplete) {
        return 1 - Math.pow(1 - percentComplete, strength*2);
    };
};

AnimationTimer.makeEaseInOut = function () {
    return function (percentComplete) {
        return percentComplete - Math.sin(percentComplete*2*Math.PI) /
                (2*Math.PI);
    };
};

AnimationTimer.makeElastic = function (passes) {
    passes = passes || DEFAULT_ELASTIC_PASSES;
    return function (percentComplete) {
        return ((1-Math.cos(percentComplete * Math.PI * passes)) *
                (1 - percentComplete)) + percentComplete;
    };
};

AnimationTimer.makeBounce = function (bounces) {
    var fn = AnimationTimer.makeElastic(bounces);
    return function (percentComplete) {
        percentComplete = fn(percentComplete);
        return percentComplete <= 1 ? percentComplete : 2-percentComplete;
    };
};

AnimationTimer.makeLinear = function () {
    return function (percentComplete) {
        return percentComplete;
    };
};
```

提示：渐变动画

读者也可能在 Flash 或 CSS3 中见过本章所讲的这种时间轴扭曲技术，不过名字叫做“渐变动画”[⊖]。在这种动画中，只要定义好关键帧就行了，Flash 或 CSS3 会根据开发者提供的时间扭曲函数（补间函数）来自动生成相邻两个关键帧之间的过渡帧。

Canvas 并未提供 CSS3 与 Flash 所支持的这种高级抽象功能，它并不能制作原生的渐变动画。所以，在本章中我们将自己实现渐变动画。

7.4 时间轴扭曲运动

读者已经学会了如何使用动画计时器与时间轴扭曲函数来控制动画时间轴的流动，下面咱们用图 7-8 所示应用程序，来演示一些经典的动作渐变[⊜]动画。在该程序中，用户可以使用不同的时间轴扭曲函数以影响小球的运动方式。

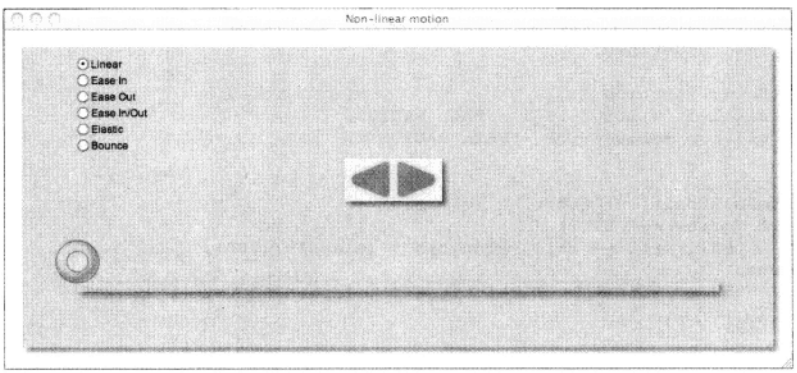


图 7-8 演示各种动作渐变算法

本程序支持如下时间轴扭曲函数：

- 线性运动：以恒定速度移动。
- 缓入运动：起初缓慢移动，逐渐提速。
- 缓出运动：起初快速移动，逐渐减速。
- 缓入缓出运动：起初缓慢移动，逐渐提速，再逐渐减速。
- 弹簧运动：围绕一个点振荡。
- 弹跳运动：在某点附近反复弹跳。

在用户选定某个时间轴扭曲函数并按下其中某个箭头按钮之后，应用程序就会用选定的函数来影响小球的运动。程序清单 7-8 演示了应用程序移动小球所用的代码。

该范例程序创建了 6 个单选按钮，它们分别对应于程序清单 7-7 所列的 6 个时间轴扭曲函数。程序将动画时长定为 3.6 秒，并用此值来创建动画计时器对象。由于在创建时并未指定时间轴扭曲函数，所以计时器将使用默认的线性函数。

⊖ Tweening、Inbetweening，又叫“补间动画”——译者注

⊜ motion tweening，又称“运动内插”、“动作补间”。——译者注

接下来，程序创建了名为 ball 的精灵，该对象含有名为 moveBall 的行为对象。如果动画计时器正在运转，那么行为对象的 execute() 方法机会根据计时器返回的动画运行时间来移动小球。

程序清单 7-8　以各种运动方式移动小球

```
var linear=AnimationTimer.makeLinear(),
    easeIn=AnimationTimer.makeEaseIn(1),
    easeOut=AnimationTimer.makeEaseOut(1),
    easeInOut=AnimationTimer.makeEaseInOut(1),
    elastic=AnimationTimer.makeElastic(5),
    bounce=AnimationTimer.makeBounce(5),

    PUSH_ANIMATION_DURATION = 3600,

    pushAnimationTimer = new AnimationTimer(PUSH_ANIMATION_DURATION),
    ...

// Move ball behavior.........................................

moveBall = {
    lastTime: undefined,

    resetBall: function () {
        ball.left = LEDGE_LEFT - BALL_RADIUS;
        ball.top = LEDGE_TOP - BALL_RADIUS*2;
    },

    updateBallPosition: function (elapsed) {
        if (arrow === LEFT)
            ball.left -= ball.velocityX * (elapsed/1000);
        else
            ball.left += ball.velocityX * (elapsed/1000);
    },

    execute: function (ball, context, time) {
        if (pushAnimationTimer.isRunning()) {
            var animationElapsed = pushAnimationTimer.getElapsedTime(),
                elapsed;

            if (this.lastTime !== undefined) {
                elapsed = animationElapsed - this.lastTime;

                this.updateBallPosition(elapsed);

                if (isBallOnLedge()) {
                    if (pushAnimationTimer.isOver()) {
                        pushAnimationTimer.stop();
                    }
                }
                else { // Ball fell off the ledge
                    pushAnimationTimer.stop();
                    this.resetBall();
                }
            }
        }
        this.lastTime = animationElapsed;
    }
},
// Ball sprite...................................................
ball = new Sprite('ball', ..., [ moveBall ]);
```

请注意，行为对象的 execute() 方法并不知道动画计时器所采取的时间轴扭曲操作，该方法只是从计时器获取动画播放时间，并据此来计算小球的当前位置。

当用户选中某个单选按钮后，应用程序会为动画监听器设置适当的时间扭曲函数。这段代码如下：

```
var linearRadioButton   =document.getElementById('linearRadioButton'),
    easeInRadioButton   =document.getElementById('easeInRadioButton'),
    easeOutRadioButton  =document.getElementById('easeOutRadioButton'),
    easeInOutRadioButton=document.getElementById('easeInOutRadioButton'),
    elasticRadioButton  =document.getElementById('elasticRadioButton'),
    bounceRadioButton   =document.getElementById('bounceRadioButton');

linearRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = linear;
};

easeInRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = easeIn;
};

easeOutRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = easeOut;
};

easeInOutRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = easeInOut;
};

elasticRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = elastic;
};

bounceRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = bounce;
};

linearRadioButton.onchange = function (e) {
    pushAnimationTimer.timeWarp = linear;
};
```

正如上述各个事件监听器的代码所示，要想运用时间轴扭曲技术，只需要为动画计时器对象设置好相应的时间轴扭曲函数即可，此后计时器就会用设置好的函数来播放动画了。

下面几个小节将简述 AnimationTimer 对象所支持的各种时间轴扭曲算法。

7.4.1 没有加速度的线性运动

根据牛顿力学原理，移动物体在不受空气阻力与摩擦力影响，也不和其他物体碰撞的情况下，将以当前的速度与方向持续运动下去。这种没有加速度的运动，就叫做线性运动（linear motion），如图 7-9 所示。

在图 7-9 中，小球以恒定的速度从左至右移动。描述线性运动的数学公式很简单，动画的实际播放时间就是时间轴扭曲函数所要返回的值，如等式 7.4 所示。

$$f(x) = x$$

等式 7.4　线性运动所用的时间轴扭曲函数

实现等式 7.4 所用的函数也很简单：

```
function (percentComplete) {
    return percentComplete;
};
```

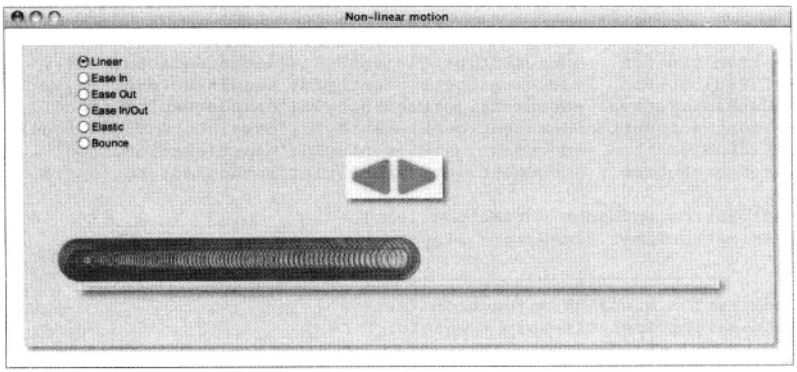


图 7-9 线性运动（图中的每个小球代表动画中的一帧）

如果用横轴刻度表示动画的实际播放百分比，用纵轴刻度表示经过扭曲函数处理之后的播放百分比，那么线性运动所用的时间轴扭曲函数，其图像就是一条呈 45 度角的直线，如图 7-10 所示。

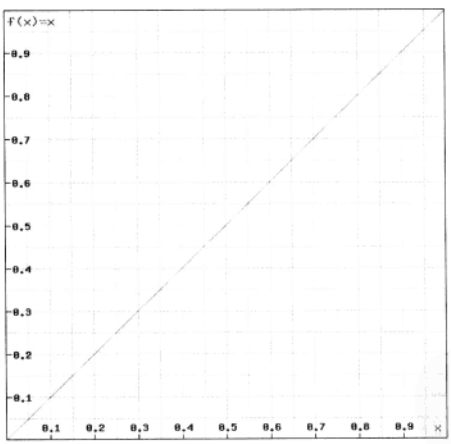


图 7-10 线性运动所用的时间轴扭曲函数图像

7.4.2 逐渐加速的缓入运动

在现实生活中，很多物体都不会以某个恒定的速度持续地移动下去。在很多情况下，都是起初缓慢地移动，然后加速。比如短跑运动员从静止状态突然加速冲出人群，或潜水者从没有垂直速度的状态加速扎入水中。

在动画制作的术语中，这种逐渐加速的运动过程叫做“缓入”（ease in），如图 7-11 所示。

缓入运动所用的时间扭曲函数是一个幂函数（power function）。等式 7.5 中所用的指数是 2，然而不一定非要用这个值，你可以用 4 或 5 做指数，那样做会让缓入效果更为夸张。

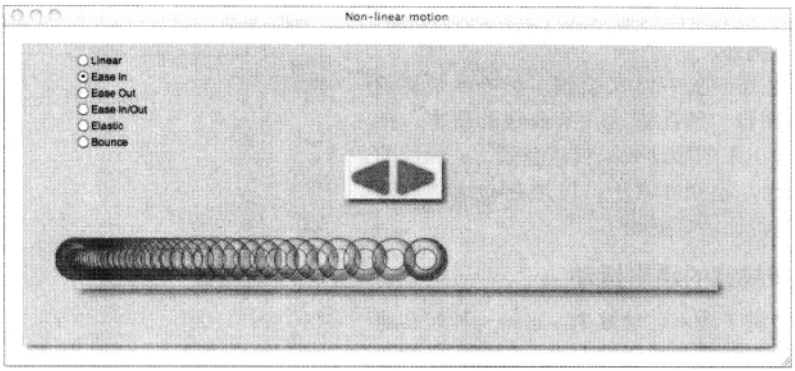


图 7-11　缓入运动

$$f(x) = x^2$$

等式 7.5　缓入运动所用的时间轴扭曲函数

等式 7.5 可以用如下 JavaScript 代码来实现：

```
function (percentComplete) {
    return Math.pow(percentComplete, 2);
};
```

图 7-12 描绘了上述 JavaScript 函数的图像。

在图 7-12 中，横轴上的刻度表示动画的实际播放百分比，而纵轴上的刻度，则是被播放动画时所用的时间轴扭曲函数处理之后的值。在本范例程序中，该值就是 AnimationTimer.getElapsedTime() 方法的返回值。

线性运动所用的时间轴扭曲函数，其图像作为参考线也画在了图 7-12 之中。图中的曲线就是刚才那个平方函数的图像，这条曲线上的点，其纵坐标代表扭曲后的时间值，而横坐标则表示实际的播放时间。举例来说，横轴上的 0.5 表示动画此时已经播放完全长的一半了，而纵轴上对应的刻度却是 0.25，这说明它看上去好像只播完了前四分之一。

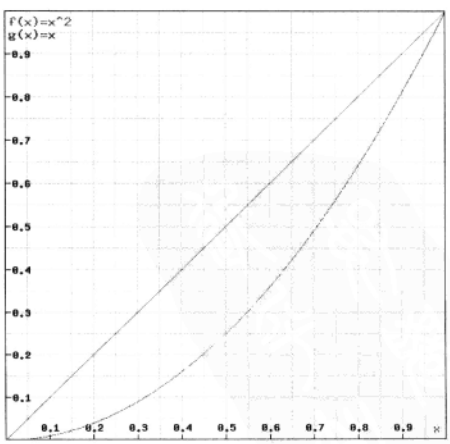


图 7-12　缓入运动所用的时间轴扭曲函数图像

图 7-12 中的两个坐标轴都表示动画的播放百分比，而直线与曲线上面各点的斜率则表示动画播放速度。图 7-12 中的那条直线，其斜率是恒定的，所以凡是用它来做时间扭曲函数的那些动画，其播放速度也是稳定不变的。然而那条曲线的斜率却在持续地变化：一开始斜率很小，所以动画在这段时间内就播放地很缓慢，沿着 x 轴从左向右移动，我们就会发现曲线上各点的斜率也在逐渐增大，这意味着动画会越播越快。该函数图像之所以是条曲线，就是因为它上面各个点的斜率不同。

图 7-12 所示的幂函数曲线在许多系统中都会用到，诸如弹簧运动、经济学分析等等，制作动画时当然也不例外。

除了 x^2 之外，图 7-13 又绘制了另外两个幂函数 x^3 与 x^4 的图像。请注意这几条曲线的斜率。指数越高，所做出来的缓动效果就越夸张。x^4 函数的曲线与 x^2 相比，起初非常平，但是在动画播放的后半段，就显得比后者更陡了。

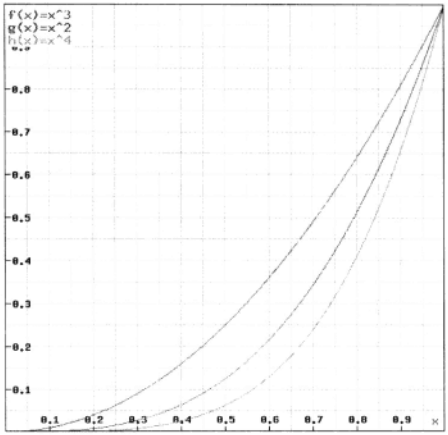


图 7.13 缓入运动所用的幂函数图像

7.4.3 逐渐减速的缓出运动

上一小节讲了缓入运动效果，这是一种起初速度很慢，而后逐渐提速的运动方式。与之相反的运动效果叫做缓出（ease out）运动，起初物体移动得很快，然后逐渐减速，如图 7-14 所示。

制作缓出运动效果所用的公式如等式 7.6 所述，图 7-15 画出了等式所对应函数的图像。函数曲线起初的斜率很陡，但稍后就逐渐降低了，最后趋近于 0。因此，动画的播放速度一开始非常快，然后就逐渐放慢了。

$$f(x) = 1 - (1 - x)^2$$

等式 7.6 缓出运动所用的时间轴扭曲函数

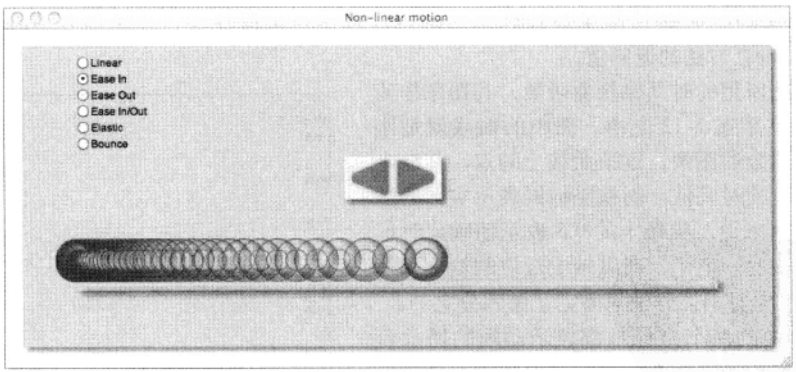


图 7-14 缓出运动

等式 7.6 可用如下 JavaScript 代码实现：

```
function (percentComplete) {
    return 1 - Math.pow(1 - percentComplete, 2);
}
```

和讲解缓入效果时所用的方式相同，我们也在图 7-16 中画出制作缓出效果所用的三次和四次函数曲线。与缓入运动类似，函数的次数越高，做出来的效果就越明显。

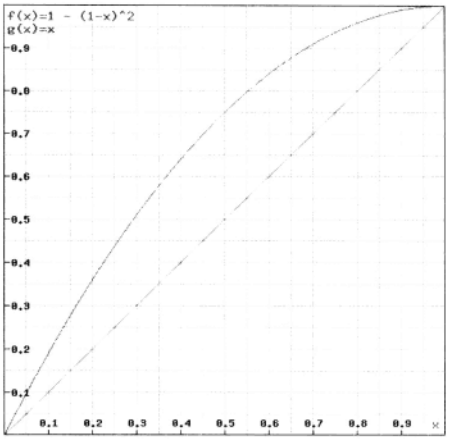


图 7-15　缓出运动所用的时间扭曲函数图像

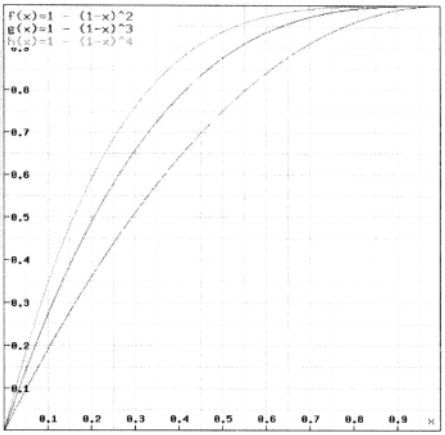


图 7-16　缓出运动所用的函数图像

7.4.4　缓入缓出运动

我们再来思考一下原来举过的那个短跑运动员起跑的例子。运动员在起跑后会逐渐达到其最大速度，然后到了某个时刻，其速度又会逐渐放缓，直至停下。这种运动类型结合了缓入运动与缓出运动，其效果如图 7-17 所示。

从本质上讲，缓入缓出运动是一种周期运动，所以我们可以用正弦波来表示它。等式 7.7 列出了制作该运动效果所用的时间轴扭曲函数，图 7-18 画出了该函数的图像。

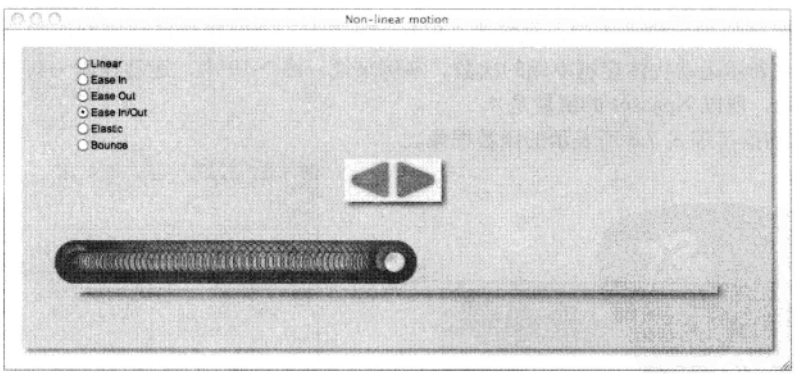


图 7-17　先进行缓入运动，然后进行缓出运动

$$f(x) = x - \sin(x \times 2\pi) / (2\pi)$$

等式 7.7　缓入缓出运动所用的时间轴扭曲函数

等式 7.7 所代表的函数图像画在了图 7-18 之中。

以下 JavaScript 代码实现了等式 7.7 所列公式：

```
function (percentComplete) {
    return percentComplete
```

```
                - Math.sin(percentComplete*2*Math.PI) / (2*Math.PI);
};
```

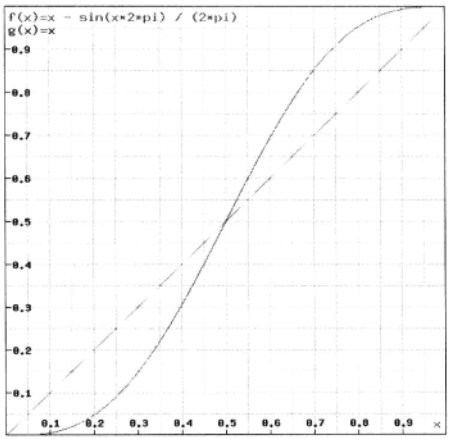


图 7-18　缓入缓出运动所用的时间轴扭曲函数图像

7.4.5　弹簧运动与弹跳运动

弹簧运动与弹跳运动是另外两种常见的特效，图 7-19 与图 7-21 分别描绘了它们的效果。

等式 7.8 列出了实现弹簧运动所用的公式。

$$f(x) = (1-\cos(x \times \text{Npasses} \times \pi) \times (1-x)) + x$$

等式 7.8　弹簧运动所用的时间轴扭曲函数

与本节前面的那些等式不同，在等式 7.8 中，除了表示动画播放进度百分比的 x 之外，还有一个变量，它表示运动物体穿越中轴的次数。举例来说，图 7-19 中，运动物体一共完成了 4 次穿越中轴的运动，所以 Npasses 的值就是 4。

图 7-20 描绘了等式 7.8 所表示的函数图像。

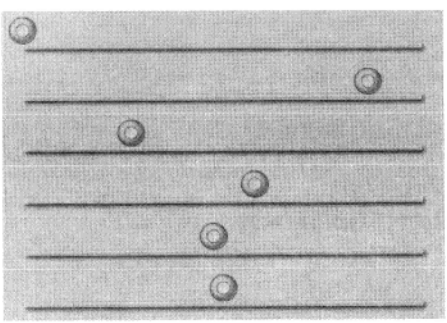


图 7-19　弹簧运动（从上到下依次演示了运动全过程）

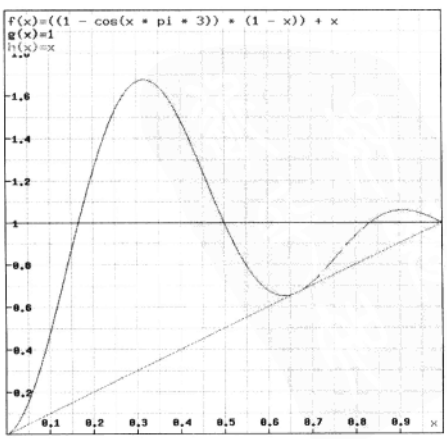


图 7-20　弹簧运动所用的时间轴扭曲函数图像

与刚才的公式一样，它的函数图像也同本节前面的那些图像不同，因为这次纵轴的跨度是2.0而非1.0，而且函数曲线上的某些点，其纵坐标也超过了1.0。

在实际运用中，如果使用纵轴刻度所示的值来播放动画，那么物体就会超越它在动画结束时所应止步的位置，随着运动的进行，物体在超越应有的终止点后又会朝反方向移动，再次穿越中轴，如此循环往复。在图7-20中，由于函数所用的Npasses值是3，所以运动物体会3次穿越“y=1.0”这条中轴线。

弹跳运动与弹簧运动类似，其效果如图7-21所示。一开始小球由平台中央向左移动，在到达平台最左端后又被弹回。

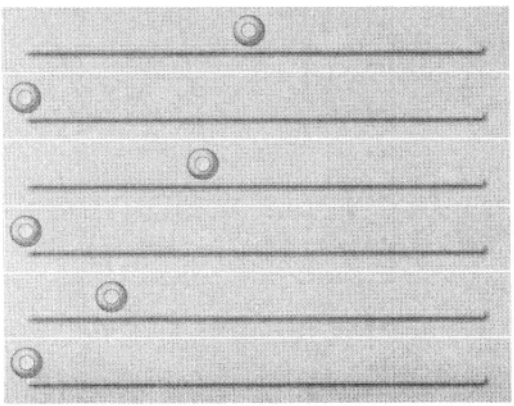


图7-21 弹跳运动

要制作弹跳效果，需要用到两个公式。等式7.9所列的公式，与实现弹簧运动所用的等式7.8相同。如果物体当前被弹起的次数是偶数，则用此公式，否则就用等式7.10所列的那个公式，两式中的$N_{bounces}$均表示运动物体被弹起的总次数[⊖]。

$$f(x) = (1 - \cos(x \times N_{bounces} \times \pi) \times (1-x)) + x$$

等式7.9 弹跳运动所用的时间轴扭曲函数，适用于物体当前被弹起的次数为偶数的情况

$$f(x) = 2 - (((1-\cos(x \times \pi \times N_{bounces})) \times (1-x)) + x)$$

等式7.10 弹跳运动所用的时间轴扭曲函数，适用于物体当前被弹起的次数为奇数的情况

图7-22画出了等式7.9与等式7.10所表示的函数图像。背后的小图与前面的大图都将两个函数图像画了出来，然而大图中仅包含弹跳运动算法真正用到的那部分图像。

本节讲解了如何扭曲时间轴，以及如何利用扭曲后的播放时间值来影响动画中物体的运动方式。不过，时间轴扭曲技术也可以用来修改除物体运动方式之外的其他属性。例如，我们可以用缓入、缓出等效果来修改动画帧的播放速度，让动画中精灵的动作切换过程也展现出加速或减速

⊖ 此处原书想要表述的意思是：先按照实现弹簧运动所用的那个公式（即等式7.9）来计算，若结果大于1.0，那么再根据等式7.10对其进行修正，使其小于1.0。比方说，如果按等式7.9计算出来的结果是1.4，那么由于该值大于1.0，所以必须要根据等式7.10对其进行调整，用2减去它，得到0.6，即为最终结果。因为物体在弹跳运动时不可能超过它最终应该停止的那个位置，所以要对大于1.0的值进行修正。为了简洁起见，翻译时在不改变作者原意的情况下，重新组织了这段文字。——译者注

的效果来。下一节就要研究这个问题。

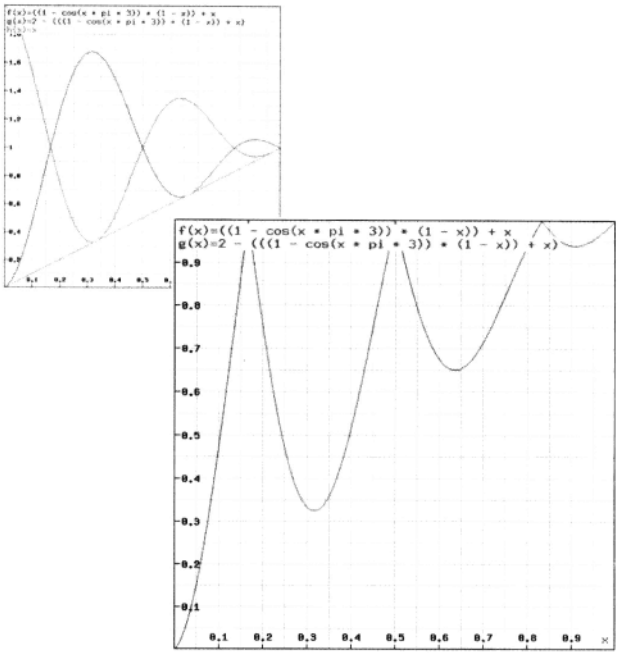


图 7-22　弹跳运动所用的时间轴扭曲函数图像

7.5　以扭曲后的帧速率播放动画

读者在前面已经学会了如何扭曲时间轴，以及如何以之影响物体的运动方式。现在我们讲讲怎样用该技术来修改其他基于时间的属性。

图 7-23 所示应用程序会根据用户所选的时间轴扭曲算法来播放动画，这些算法在页面顶部以单选按钮的形式表示。在图 7-23 中，当前用户选择了缓出效果，这将导致精灵对象的运动被扭曲后的时间值所影响。在动画的前四分之三时间段内，它会一直领先于那条垂直线段所标识的标准位置，然而，在动画的最后四分之一时段里，被扭曲的时间值将会导致精灵的运动速度显著降低，在动画播放完毕的那一刻，标准时间线就会追上逐渐减速的精灵。

图 7-23 之中的标准时间线与精灵，分别表示了动画的真实播放时间以及扭曲后的播放时间。有一个问题没有在图 7-23 之中演示出来，那就是：精灵的帧速率也会随着它的移动速度而改变。当我们选择缓出效果时，在动画的前四分之三时间段内，精灵会以很快的速度在各个动画帧之间轮换，看上去好像一直在标准时间线的前方疯狂地跑动。然而，在动画的后四分之一，精灵的速度与动画换帧的速率都会急剧下降，于是标准时间线就逐渐赶了上来，并在动画播放完毕时追上精灵。

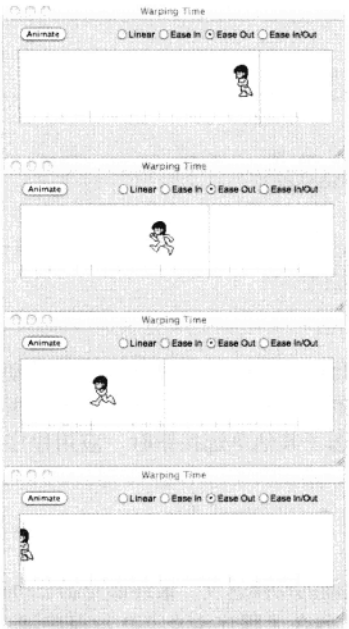


图 7-23　用时间轴扭曲技术来影响精灵的运动及动画的换帧速度

该程序所创建的精灵含有两个行为对象，分别是 moveRightToLeft 与 runInPlace：

```
sprite = new Sprite('runner',
                    new SpriteSheetPainter(runnerCells),
                    [ moveRightToLeft, runInPlace ]);
```

如果精灵绘制器上一次换帧的时间距离现在已经超过 0.1 秒，那么 runInPlace.execute() 方法就会命令精灵绘制器对象切换至下一帧，并且重新设定用于保存上次换帧时间的 lastAdvance 变量。

```
runInPlace = {
   execute: function() {
      var elapsed = animationTimer.getElapsedTime();

      if (lastAdvance === 0) {  // Skip first time
         lastAdvance = elapsed;
      }
      else if (lastAdvance !== 0 &&
               elapsed - lastAdvance > PAGEFLIP_INTERVAL) {
         sprite.painter.advance();
         lastAdvance = elapsed;
      }
   }
},
```

moveRightToLeft.execute() 方法将精灵的速度乘以前后两次调用该方法的时间差，以此得出精灵本次所应移动的距离。

```
moveRightToLeft = {
   lastMove: 0,
      reset: function () {
```

```
            this.lastMove = 0;
        },

        execute: function(sprite, context, time) {
        var elapsed = animationTimer.getElapsedTime(),
            advanceElapsed = elapsed - this.lastMove;

        if (this.lastMove === 0) { // Skip first time
            this.lastMove = elapsed;
        }
        else {
            sprite.left -= (advanceElapsed / 1000) * sprite.velocityX;
            this.lastMove = elapsed;
        }
    }
},
```

如果动画计时器对象的 getElaspsedTime() 方法所返回的是动画实际播放时间，那么精灵的运动速度与换帧速率都将是个恒定的值。实际上，一开始启动应用程序时，默认选择的线性算法其效果就是如此。然而，当用户点选了其他单选按钮时，应用程序就会更换动画计时器对象的时间轴扭曲函数了。比如说：

```
easeInRadioButton.onchange = function (e) {
    animationTimer.timeWarp = AnimationTimer.makeEaseIn(1);
};
```

精灵的运动速度与它在各帧之间的切换速度，最终都受制于动画计时器对象的时间轴扭曲函数。

图 7-23 所示应用程序的代码列在了程序清单 7-9 与程序清单 7-10 之中。请注意，程序在绘制精灵对象的每个动画帧时，使用了本书 6.2.3 小节所讲的精灵表绘制器。

程序清单 7-9　用时间轴扭曲技术来影响精灵的运动及动画的换帧速度 (HTML 代码)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Warping Time</title>

        <style>
            body {
                background: #cdcdcd;
            }

            .controls {
                position: absolute;
                left: 150px;
                top: 10px;
                font: 12px Arial;
            }

            #canvas {
                position: absolute;
                left: 0px;
                top: 20px;
                margin: 20px;
                border: thin inset rgba(100,150,230,0.8);
                background: #efefef;
            }

            #animateButton {
                margin-left: 15px;
```

```
            margin-bottom: 10px;
         }
      </style>
   </head>

   <body>
      <input id='animateButton' type='button' value='Animate'/>

      <canvas id='canvas' width='420' height='100'>
         Canvas not supported
      </canvas>

      <div id='motionControls' class='controls'>
         <div id='motionRadios'>
            <input type='radio' name='motion'
                   id='linearRadio' checked/>Linear

            <input type='radio' name='motion'
                   id='easeInRadio'/>Ease In

            <input type='radio' name='motion'
                   id='easeOutRadio'/>Ease Out

            <input type='radio' name='motion'
                   id='easeInOutRadio'/>Ease In/Out
         </div>
      </div>

      <script src='stopwatch.js'></script>
      <script src='animationTimer.js'></script>
      <script src='requestNextAnimationFrame.js'></script>
      <script src='sprites.js'></script>
      <script src='example.js'></script>
   </body>
</html>
```

程序清单 7-10　用时间轴扭曲技术来影响精灵的运动及动画的换帧速度 (JavaScript 代码)

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),

    linearRadio = document.getElementById('linearRadio'),
    easeInRadio = document.getElementById('easeInRadio'),
    easeOutRadio = document.getElementById('easeOutRadio'),
    easeInOutRadio = document.getElementById('easeInOutRadio'),

    animateButton = document.getElementById('animateButton'),
    spritesheet = new Image(),

    runnerCells = [
      { left: 0,   top: 0, width: 47, height: 64 },
      { left: 55,  top: 0, width: 44, height: 64 },
      { left: 107, top: 0, width: 39, height: 64 },
      { left: 152, top: 0, width: 46, height: 64 },
      { left: 208, top: 0, width: 49, height: 64 },
      { left: 265, top: 0, width: 46, height: 64 },
      { left: 320, top: 0, width: 42, height: 64 },
      { left: 380, top: 0, width: 35, height: 64 },
      { left: 425, top: 0, width: 35, height: 64 },
    ],

    interval,
```

```
    lastAdvance = 0.0,

    SPRITE_LEFT = canvas.width - runnerCells[0].width;
    SPRITE_TOP = 10,

    PAGEFLIP_INTERVAL = 100,
    ANIMATION_DURATION = 3900,

    animationTimer = new AnimationTimer(ANIMATION_DURATION,
                            AnimationTimer.makeLinear(1)),

    LEFT = 1.5,
    RIGHT = canvas.width - runnerCells[0].width,
    BASELINE = canvas.height - 9.5,
    TICK_HEIGHT = 8.5,
    WIDTH = RIGHT-LEFT,

    runInPlace = {
       execute: function() {
          var elapsed = animationTimer.getElapsedTime();

          if (lastAdvance === 0) {  // Skip first time
             lastAdvance = elapsed;
          }
          else if (lastAdvance !== 0 &&
                   elapsed - lastAdvance > PAGEFLIP_INTERVAL) {
             sprite.painter.advance();
             lastAdvance = elapsed;
          }
       }
    },

    moveRightToLeft = {
       lastMove: 0,
          reset: function () {
              this.lastMove = 0;
          },

          execute: function(sprite, context, time) {
          var elapsed = animationTimer.getElapsedTime(),
              advanceElapsed = elapsed - this.lastMove;

          if (this.lastMove === 0) { // Skip first time
             this.lastMove = elapsed;
          }
          else {
             sprite.left -= (advanceElapsed / 1000) * sprite.velocityX;
             this.lastMove = elapsed;
          }
       }
    },

    sprite = new Sprite('runner',
                        new SpriteSheetPainter(runnerCells),
                        [ moveRightToLeft, runInPlace ]);

// Functions.......................................................

function endAnimation() {
   animateButton.value = 'Animate';
   animateButton.style.display = 'inline';
   animationTimer.stop();
```

```
    lastAdvance = 0;
    sprite.painter.cellIndex = 0;
    sprite.left = SPRITE_LEFT;
    animationTimer.reset();
    moveRightToLeft.reset();
}

function startAnimation() {
    animationTimer.start();
    animateButton.style.display = 'none';
    window.requestNextAnimationFrame(animate);
}

function drawAxis() {
    context.lineWidth = 0.5;
    context.strokeStyle = 'cornflowerblue';

    context.moveTo(LEFT, BASELINE);
    context.lineTo(RIGHT, BASELINE);
    context.stroke();

    for (var i=0; i <= WIDTH; i+=WIDTH/20) {
        context.beginPath();
        context.moveTo(LEFT + i, BASELINE-TICK_HEIGHT/2);
        context.lineTo(LEFT + i, BASELINE+TICK_HEIGHT/2);
        context.stroke();
    }

    for (i=0; i < WIDTH; i+=WIDTH/4) {
        context.beginPath();
        context.moveTo(LEFT + i, BASELINE-TICK_HEIGHT);
        context.lineTo(LEFT + i, BASELINE+TICK_HEIGHT);
        context.stroke();
    }

    context.beginPath();
    context.moveTo(RIGHT, BASELINE-TICK_HEIGHT);
    context.lineTo(RIGHT, BASELINE+TICK_HEIGHT);
    context.stroke();
}

function drawTimeline() {
    var realElapsed = animationTimer.getRealElapsedTime(),
        realPercent = realElapsed / ANIMATION_DURATION;

    context.lineWidth = 0.5;
    context.strokeStyle = 'rgba(0,0,255,0.5)';

    context.beginPath();

    context.moveTo(WIDTH - realPercent*(WIDTH), 0);
    context.lineTo(WIDTH - realPercent*(WIDTH), canvas.height);
    context.stroke();
}

// Event handlers.....................................................

animateButton.onclick = function (e) {
    if (animateButton.value === 'Animate') startAnimation();
    else                                   endAnimation();
};
```

```
linearRadio.onclick = function (e) {
   animationTimer.timeWarp = AnimationTimer.makeLinear(1);
};

easeInRadio.onclick = function (e) {
   animationTimer.timeWarp = AnimationTimer.makeEaseIn(1);
};

easeOutRadio.onclick = function (e) {

   animationTimer.timeWarp = AnimationTimer.makeEaseOut(1);
};

easeInOutRadio.onclick = function (e) {

   animationTimer.timeWarp = AnimationTimer.makeEaseInOut();
};

// Animation.......................................................

function animate(time) {
   if (animationTimer.isRunning()) {
      elapsed = animationTimer.getElapsedTime();

      context.clearRect(0, 0, canvas.width, canvas.height);
      sprite.update(context, time);
      sprite.paint(context);

      drawTimeline();
      drawAxis();

      if (animationTimer.isOver()) {
         endAnimation();
      }
      window.requestNextAnimationFrame(animate);
   }
}

// Initialization..................................................

spritesheet.src = 'running-sprite-sheet.png';
sprite.left = SPRITE_LEFT;
sprite.top = SPRITE_TOP;
sprite.velocityX = 100; // pixels/second

drawAxis();

spritesheet.onload = function () {
   sprite.paint(context);
};
```

7.6 总结

本章讲述了在制作动画与游戏时所用的基本物理效果。首先讲的是如何针对受重力影响的运动进行建模，这些运动包括了落体运动、在空中飞行的抛射体运动以及钟摆等非线性运动。

大多数运动都是非线性的，比如加速驶离红绿灯的汽车、弹跳的小球等，所以我们研究了如何利用时间轴扭曲技术来创建缓入、缓出等效果。学会了时间轴扭曲技术后，我们就可以用它来调整一些受时间因素制约的属性了，比如移动速度及动画帧速率等。正如本章最后那个例子所演示的，时间轴扭曲技术不仅可以修改精灵的移动速度，而且可以改变精灵对象在各个动画帧之间切换的速率。

下一章将会讲述与碰撞检测有关的物理学知识。

第8章
碰撞检测

在许多动画与绝大部分游戏中，都会用到各种形式的碰撞检测，读者将在本章中学到如何实现碰撞检测。有些检测技术比较简单，例如外接图形判别法与光线投射法（Ray Casting），还有一些则稍微复杂，例如任意多边形、圆形、图像与精灵之间的碰撞检测。

本章大部分内容都在讲解“分离轴定理”[⊖]的运用，该定理的判断准确度很高，而且适用范围广泛，可以用于检测二维平面及三维空间中的多边形碰撞。读者将会学到如何在 Canvas 中运用 SAT 来检测多边形碰撞，以及如何将其推广至圆形、图像与精灵之间的碰撞检测中。

在本章的最后，我们要讲一下运用 SAT 所产生的副产品：最小平移向量（minimum translation vector，简称 MTV）。这个值恰好就是将两个物体由碰撞变为不碰撞时，所需移动的最小距离。运用该值，我们可以使两个相互碰撞的物体分离，也可以使两个尚未碰撞的物体粘在一起，还可以让一个物体从另一个物体表面弹开。

8.1 外接图形判别法

在二维平面中执行碰撞检测时，通常会根据物体外接图形的面积[⊜]来判定（在三维空间中则是根据体积），所以咱们就先通过一些例子来看看如何采用这个方法来检测碰撞。

8.1.1 外接矩形判别法

外接矩形经常被当作物体的轮廓来参与碰撞检测，此时它也称作“边界框”（bounding box）。图 8-1 所示应用程序演示了此种判别方法。用户可以用左右箭头来移动一开始出现在上方平台中的那个小球。如果小球从平台左侧落下，并碰到了下方的平台，那么它就会停在该平台上。

程序使用下面这个方法来判断小球是否碰到了底部的平台：

```
ballWillHitLedge: function (ledge) {
   var ballRight = ball.left + ball.width,
       ledgeRight = ledge.left + ledge.width,
       ballBottom = ball.top + ball.height,
       nextBallBottomEstimate = ballBottom + ball.velocityY / fps;

   return ballRight > ledge.left &&
          ball.left < ledgeRight &&
          ballBottom < ledge.top &&
          nextBallBottomEstimate > ledge.top;
}
```

ballWillHitLedge() 方法首先算出小球底部的位置，然后根据它现在的移动速度与当前动画的帧速率，估算出小球在下一帧动画中的位置。

⊖ separating axis theorem，简称SAT，又叫“超平面分离定理”（Hyperplane separation theorem），详情参见：http://en.wikipedia.org/wiki/Separating_axis_theorem。——译者注

⊜ 这里所说的“面积”，意思是看两个物体的表面是否发生重合，并不强调具体面积值的计算。——译者注

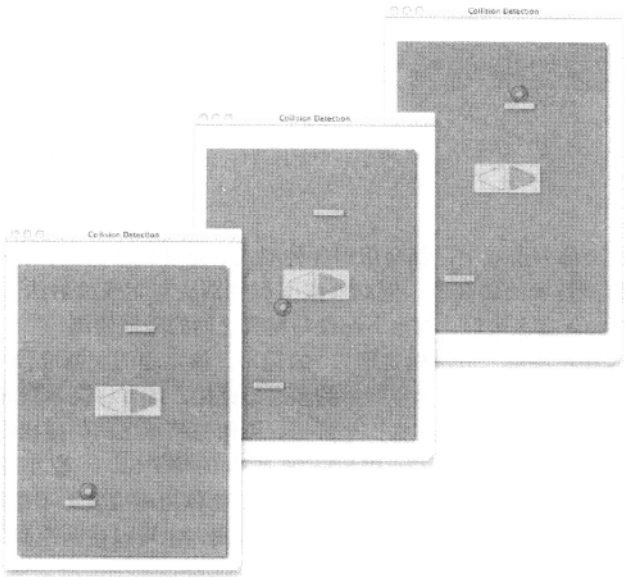


图 8-1 根据外接矩形来检测碰撞

如果小球当前位于平台上方，而动画播至下一帧时就将和平台相碰，那么 ballWillHitLedge() 方法则返回 true。

除了采用边界框来判断碰撞之外，图 8-1 所示应用程序还演示了“事前碰撞检测法”(priori collision detection)，它可以提前探知是否会发生碰撞。也可以使用“事后碰撞检测法”(posteriori collision detection)，此种方式在事发之后才能检测到碰撞。下一小节我们将会讲述如何实现这种“事后碰撞检测法”。

提示：事前碰撞检测法可能会失效

在上个示例中，应用程序通过计算小球在下一帧动画中的位置来检测碰撞。这种方法容易出错，因为它是根据当前帧速率来估算的，而帧速率如果突然改变，那么估算出的结果可能就不准了。

假设当前小球在平台上方，并且应用程序预判在下一帧动画时它不会和平台相碰，但事实上两者却恰恰相碰了，那么程序就漏判了这次碰撞。

事前碰撞检测法的一个缺点就是判断的准确度不够高。

8.1.2 外接圆判别法

有些时候，采用外接圆来检测碰撞比用外接矩形来检测，要更加准确一些。图 8-2 演示了如何用这种检测方式来判断小球是否落入桶中。如果小球与桶上方的那个圆形相碰，那么应用程序就知道小球将会落入桶中。

从图 8-2 中可以看出，检测两个圆形是否碰撞是很简单的。如果两个圆心之间的距离小于两圆半径之和，那么两者就发生了碰撞。很多人喜欢采用外接圆判别法，就是因为其算法简单易行。

在图 8-3 之中，应用程序采用图 8-2 所示判别法来判断小球是否会落入桶中。

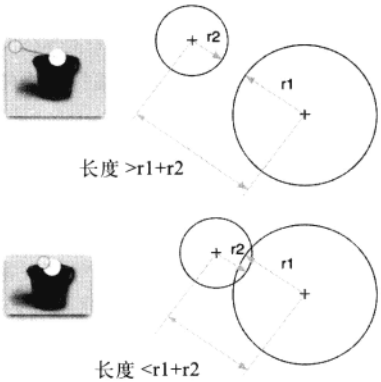


图 8-2　检测两个圆形是否碰撞：看圆心间距是否小于半径之和

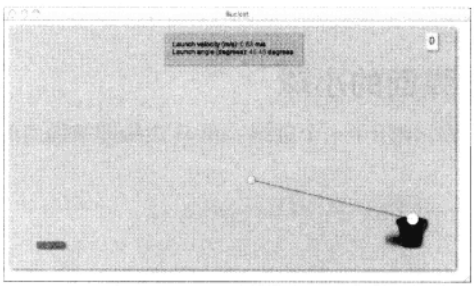


图 8-3　使用外接圆判别法检测碰撞

该程序采用如下方法来判断小球会不会落在桶里：

```
isBallInBucket: function() {
   var ballCenter = { x: ball.left + BALL_RADIUS,
                      y: ball.top  + BALL_RADIUS
   },
   distance = Math.sqrt(
      Math.pow(bucketHitCenter.x - ballCenter.x, 2) +
      Math.pow(bucketHitCenter.y - ballCenter.y, 2));
   return distance < BALL_RADIUS + bucketHitRadius;
}
```

在上述方法中，bucketHitCenter 对象中含有判定得分所用圆形[⊖]的圆心坐标，该方法的代码利用毕达哥拉斯定理[⊜]，按照等式 8.1 所述公式，来计算小球的圆心与判定圆的圆心之间的距离。

$$c=\sqrt{a^2+b^2}$$

等式 8.1　两点间距离计算公式

图 8-3 所示应用程序使用的是“事后碰撞检测法”，只有在确实发生了碰撞之后，该方法才能检测到结果。

⊖ 即图中那个白色的圆形。——译者注

⊜ 原文为Pythagorean theorem，中文叫勾股定理。——译者注

与图 8-1 所示应用程序一样，图 8-2 之中程序所用的这套碰撞检测法也不完美。如果小球移动得太快，那么它就会在相邻两个动画帧之间快速穿越判定得分所用的那个圆形，导致应用程序漏判。要解决这个问题，最简单的办法就是限制动画里那些小物体的移动速度。另外，本章稍后就将告诉大家一些更为精确的碰撞检测方式。

小技巧：如何在“事前碰撞检测法”与“事后碰撞检测法”中作出选择

“事前碰撞检测法”是在可能发生碰撞之前提前判定，而“事后碰撞检测法”则是在发生了碰撞之后再做判断。

由于“事前碰撞检测法”需要估算物体将来的位置，所以该算法必须知道所有可能影响物体位置的因素，诸如速度等等，这样才能够执行运算。而与此相反，“事后碰撞检测法”并不需要这样的计算过程，它只需要判断两个物体是否已经发生了碰撞。

但是，采用“事后碰撞检测法”时，必须要进行一些碰撞后的处理工作[⊖]，这些工作通常和“事前碰撞检测法”中预估物体未来的运动位置所做的运算一样复杂。所以说，在这两种碰撞检测法中，并没有哪一种比另外一种更为简单。

8.2 碰到墙壁即被弹回的小球

图 8-4 所示的动画应用程序演示了一个碰到 canvas 边缘即被弹回的小球。

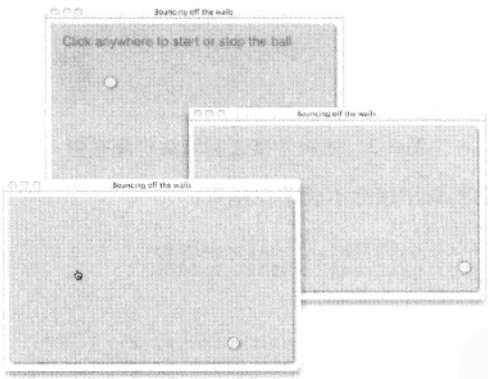


图 8-4 碰到墙壁即被弹回的小球

如果采用边界框判别法，那么根据物体的当前位置与移动速度，我们很容易就可以让小球在碰到 canvas 边缘时被反弹回来。这段代码如程序清单 8-1 所示。

程序清单 8-1 碰到墙壁即被弹回的小球

```
handleEdgeCollisions: function() {
   var bbox = getBoundingBox(ball),
       right = bbox.left + bbox.width,
       bottom = bbox.top + bbox.height;

   if (right > canvas.width || bbox.left <  0) {
```

⊖ 比如将已经碰撞的两个物体拉开，恢复到未发生碰撞时的样子。——译者注

```
        velocityX = -velocityX;

        if (right > canvas.width) {
           ball.left -= right-canvas.width;
        }

        if (bbox.left < 0)  {
           ball.left -= bbox.left;
        }
     }
        if (bottom > canvas.height || bbox.top < 0) {
           velocityY = -velocityY;

           if (bottom > canvas.height) {
              ball.top -= bottom-canvas.height;
           }
           if (bbox.top < 0) {
              ball.top -= bbox.top;
           }
        }
     };
```

程序清单 8-1 先获取了小球边界框的位置，然后检测它是否已经移出了 canvas 的范围。如果是的话，那么该方法就将小球的水平速度或垂直速度设置为原值的相反数，并再次更新小球的位置，使其重新位于 canvas 范围之内。

读者在掌握了如何使用外接图形判别法来实现碰撞检测之后，咱们再来讲一种更为严谨的算法，那就是光线投射法（ray casting）。

8.3 光线投射法

除了用外接图形来判定之外，我们也可以使用比它更为精确的光线投射法来检测碰撞。图 8-5 所示应用程序就使用光线投射法来判断小球是否落入桶中。这个程序很简单，它的左方是小球发射台，右方是小桶。当用户点击鼠标时，程序将以小球与鼠标位置的连线为发射方向，将小球抛出去，抛射的力度与连线的长度成正比。

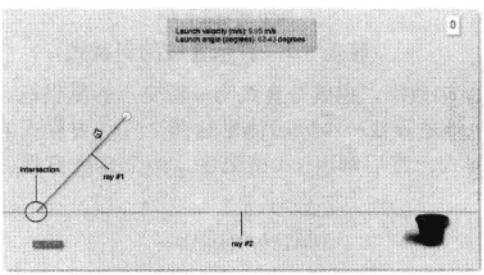


图 8-5　用光线投射法检测碰撞

光线投射法非常简单：画一条与物体的速度向量（velocity vector）相重合的线，然后再从另一个待检测物体出发，绘制第二条线，根据两条线的交点位置来判定是否发生碰撞。

图 8-5 所示应用程序绘制了一条与小球速度向量相重合的线，在图中以 #1 表示，同时该程序又绘制了一条被标注为 #2 的线，然后计算这两条线的交点。

在小球的飞行过程中，程序不断地擦除并重绘从小球到 1、2 号线交点处的连线。图 8-6 演示了这种绘制效果。

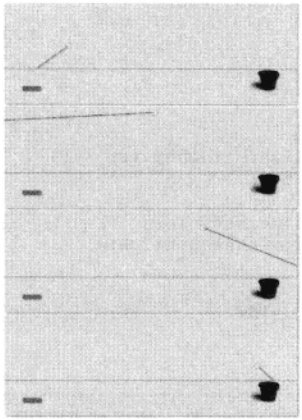


图 8-6　程序不断地重绘以小球为起点的射线

如果以下两个条件都满足，那么应用程序就会判定小球已经落入桶中：

- 1 号线与 2 号线的交点在桶口的左右边沿之间。
- 小球位于 2 号线下方。

在图 8-7 中，最右方的截图展示了同时满足上述两个条件时的情况。

图 8-7　光线投射法的详细判断过程：最右方的情况即视为发生碰撞，同时玩家得分

要实现光线投射法，咱们得先从等式 8.2 开始，这是用以表示直线的斜截式[⊖]。

$$y = mx + b$$

等式 8.2　表示直线的斜截式

在等式 8.2 中，b 表示 y 轴截距，也就是直线与 y 轴交点的纵坐标。

寻找两条线的交点，也就是寻找一个同时满足这两个直线方程式的点，因此，我们令 1 号线的方程式等于 2 号线的方程式。然后解出 x（请记住，两个直线方程中的 x_1 与 x_2 是相同的）。等式 8.3 演示了求解过程。

$$mx_1+b_1=mx_2+b_2$$
$$mx_1-mx_2=b_2-b_1$$
$$x(m_1-m_2)=b_2-bx_1$$
$$x=(b_2-b_1)\ /\ (m_1-m_2)$$

等式 8.3　求解两线交点的推导过程

⊖ slope–intercept equation，用斜率与y轴截距来表示直线的等式，简称斜截式。——译者注

解出了 x 的值后，可将其带入任意一条线的斜截式方程中，并解出 y 值。程序清单 8-2 的代码描述了这个计算过程，并演示了图 8-5 中的应用程序是如何实现碰撞检测的。

程序清单 8-2　一个可以使用光线投射法来进行碰撞检测的对象

```
catchBall = {
   intersectionPoint: { x: 0, y: 0 },

   isBallInBucket: function() {  // A posteriori
      if (lastBallPosition.left === ball.left ||
         lastBallPosition.top === ball.top)  {
         return;
      }
      // (x1, y1) = Last ball position
      // (x2, y2) = Current ball position
      // (x3, y3) = Bucket left
      // (x4, y4) = Bucket right

      var x1 = lastBallPosition.left,
          y1 = lastBallPosition.top,
          x2 = ball.left,
          y2 = ball.top,
          x3 = BUCKET_LEFT + BUCKET_WIDTH/4,
          y3 = BUCKET_TOP,
          x4 = BUCKET_LEFT + BUCKET_WIDTH,
          y4 = y3,

          // m1 = slope of (x1, y1) to (x2, y2)

          m1 = (ball.top - lastBallPosition.top) /
               (ball.left - lastBallPosition.left),

          // m2 = slope of (x3, y3) to (x4, y4)

          m2 = (y4 - y3) / (x4 - x3), // Zero, but calculate
                                      // anyway for illustration
          // b1 = y-intercept for (x1, y1) to (x2, y2)

          b1 = y1 - m1*x1,

          // b2 = y-intercept for (x3, y3) to (x4, y4)

          b2 = y3 - m2*x3;

      this.intersectionPoint.x = (b2 - b1) / (m1 - m2);
      this.intersectionPoint.y = m1*this.intersectionPoint.x + b1;

      return this.intersectionPoint.x > x3 &&
             this.intersectionPoint.x < x4 &&
             ball.top + ball.height > y3 &&
             ball.left + ball.width < x4;
   }
};
```

警告：如何处理水平线与垂直线

isBallInBucket() 方法完全忽略小球做水平运动与垂直运动时的情况。这是因为在做水平运动时，代表小球运动向量的那条线，其斜率为 0，而做垂直运动时，其运动向量的斜率为无穷大。这两个值都不适用于该方法所要采用的光线投射法。

如果开发者要在代码中使用斜截式，那么就需要对水平线与垂直线做特殊处理。

算法微调

光线投射法比外接图形判别法更加精确。不管图 8-5 中的小球移动速度多么快，应用程序总是能够检测到碰撞。不过，与所有检测碰撞的办法一样，光线投射法也不完美，图 8-8 演示了这种边界情况[⊖]。

图8-8 边界情况

在图 8-8 中，代表小球运动向量的那条线与经过小桶上沿的那条线相交，其交点刚好位于桶口的右边沿，而小球则位于横线之下。在这种情况下，刚才的算法会被判定小球与小桶发生了碰撞，然而实际上小球却并未落入桶中。

为了应对这种边界情况，我们需要将程序清单 8-2 中的 return 语句修改如下：

```
return intersectionPoint.x > x3 &&
       intersectionPoint.x < x4 &&
       ball.top + ball.height > y3 &&
       ball.left + ball.width < x4;
```

在讲完了这些碰撞检测所用的简单技巧后，我们来研究如何使用分离轴定理与最小平移向量技术来判断多边形、圆形、图像及精灵之间的碰撞。

8.4 分离轴定理（SAT）与最小平移向量（MTV）

到此为止，我们已经讲解了如何使用外接矩形与外接圆这样的外接图形来实现简单的碰撞检测，读者也学会了怎样使用光线投射法来判断碰撞。这些技巧能够适用于许多情况，而且实现起来也相对简单些，然而，它们却并不适用于检测任意多边形之间的碰撞。

本章剩下的部分将会告诉大家如何采用基于分割轴定理的算法来实现更为精确的碰撞检测。分割轴定理可以检测多边形之间的碰撞，不过也适用于圆形、图像及精灵。

读者还将看到如何利用分割轴定理来计算处理碰撞时所用到的最小平移向量。

提示：分离轴定理只适用于凸多边形

分离轴定理只适用于凸多边形，也就是所有内角均小于 180 度的多边形。凸多边形的各个顶点都是由多边形的中心向外延伸的，比如矩形、三角形、正方形等。而只要有一个角大于 180°，那么这个多边形就有了凹陷（dent），也就成了凹多边形，比如吃豆人[⊜]的形状就是如此。不能使用分离轴定理检测凹多边形之间的碰撞。

8.4.1 使用分割轴定理检测碰撞

图 8-9 演示了位于 Canvas 坐标系统中的两个多边形。右侧截图就是两者发生碰撞时的情形。

从概念上来说，分离轴定理很好理解，可以用通俗的语言将其数学模型表述如下：把受测的两个多边形置于一堵墙前面，用光线照射它们，然后根据其阴影部分是否相交来判断二者有没有相撞，如图 8-10 与图 8-11 所示。

⊖ edge case一词在软件工程中，指的是输入值恰好位于某个合法取值范围的边界处或边界外沿时的情况。例如某程序所接收的合法输入值范围是0～100之间的整数，那么当输入值为0、100、−1、101时，即可称之为边界情况，也叫边界条件。详情参阅：http://en.wikipedia.org/wiki/Edge_case及http://en.wikipedia.org/wiki/Boundary_case。——译者注

⊜ Pac-Man，一款电子游戏的主角，其形状为缺了一角的薄饼。详情参阅：http://zh.wikipedia.org/zh-cn/吃豆人。——译者注

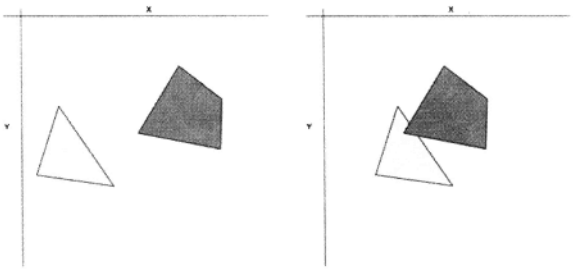


图 8-9　Canvas 中的两个多边形，右侧截图是两者碰撞时的情形

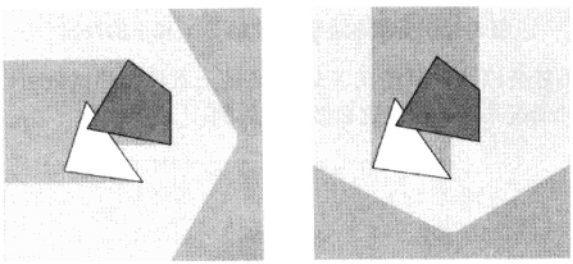


图 8-10　向受测多边形照射光线，看其阴影部分是否互相分离

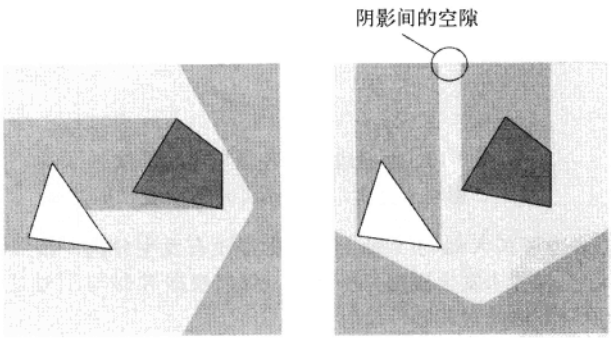


图 8-11　如果发现两者在某个方向上的阴影相互分离，那么它们就没有碰撞

上文提到的阴影部分，在数学上叫做“投影”，而那堵墙，则叫做“轴”。图 8-12 画出了图 8-10 与图 8-11 之中的那两个多边形、坐标系的 X 轴与 Y 轴，以及多边形在坐标轴上的投影。

只要它们在任何一条轴上的投影不重叠，那么就说明两者没有相碰。在图 8-12 的左侧截图中，两个多边形在 X 轴上的投影相互分离，这说明二者并未碰撞。而在右侧截图中，两个多边形在所有轴上的投影都有重叠部分，所以说二者发生了碰撞。只要能在任何一条轴上找到互相分离的投影，就说明两个多边形没有发生碰撞，而它们一旦发生了碰撞，那么肯定在所有的轴上都找不到互相分离的投影。

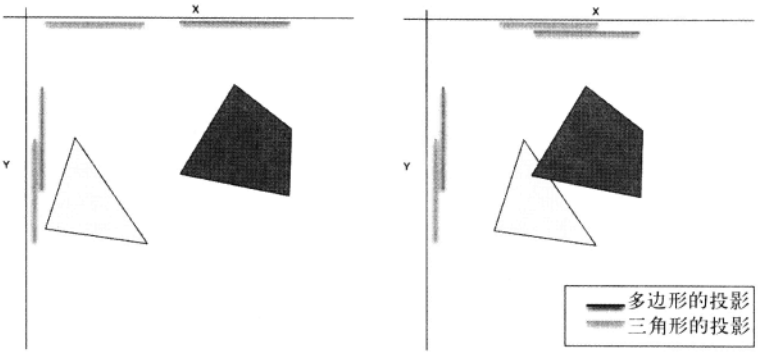


图 8-12　受测多边形在 X 轴与 Y 轴上的投影

看了图 8-12，你可能会以为，只要在 X 轴与 Y 轴上做投影测试就可以判断是否碰撞了。但是，从图 8-13 所演示的情况来看，仅仅这么做是不够的。

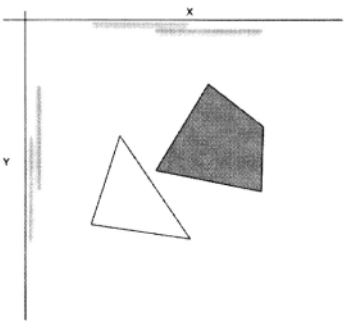


图 8-13　两多边形在 X 轴与 Y 轴的投影均未分离，然而两者并不相碰

图 8-13 中的两个多边形在 X 轴与 Y 轴上的投影都没有发生分离，按照刚才的想法，它们应该已经相撞了，但实际上却并未发生碰撞。所以说，仅仅根据 X 轴与 Y 轴这两个方向的投影来判定碰撞是不够的，必须全方位地对其进行投射才行。

图 8-14 绘制出了运用分离轴定理进行判断时所需考虑的全部投影轴。

用分离轴定理检测碰撞时，必须像图 8-14 这样，将受测多边形投射在各个投影轴上，直至找到相互分离的投影，我们才能说这两个多边形没有发生碰撞。

投影轴的数量等同于多边形的边数。比如说，在图 8-14 中，受测的三角形有 3 条边，而四边形有 4 条边，所以总共有 7 个投影轴。

由于投影轴的数量与每个多边形的边数有关，所以可能需要在很多轴上进行投影测试，从性能的角度看，这种算法也许比较耗时。不过，只要在任意一条轴上找到相互分离的投影，那么根据轴分割定理，我们就可以立刻结束检测过程，判定两者并未发生碰撞。

图 8-15 中的这两个多边形，在每条轴上的投影都未分离，故而可以判定它们发生了碰撞。

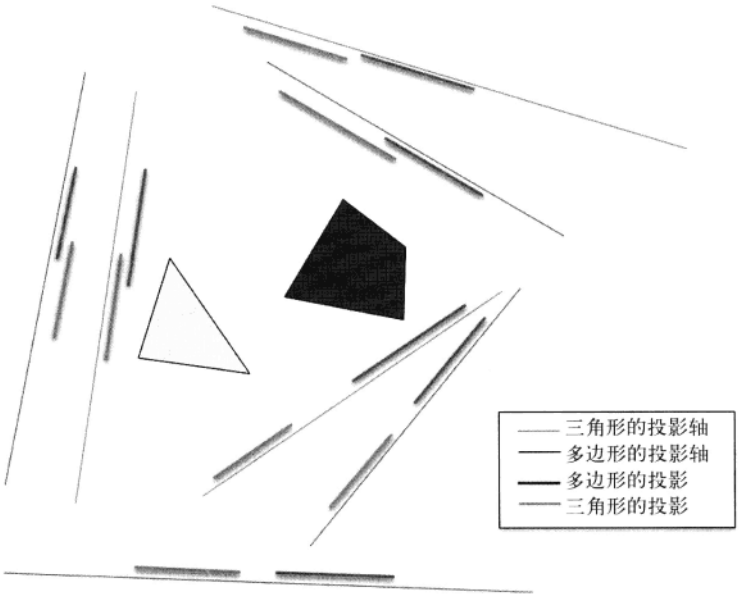


图 8-14　分离轴定理需要将受测多边形投射在各投影轴上，直至找到相互分离的投影

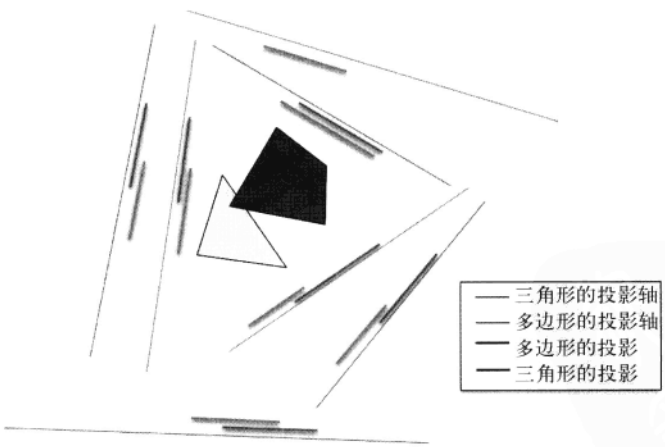


图 8-15　两个相互碰撞的受测多边形在每条轴上的投影均未分离

从更高的抽象层次上看，可以把使用轴分离定理判断两个多边形是否相撞的过程，用下列伪代码表示出来：

```
// Returns true if the polygon1 and polygon2 have collided

function polygonsCollide(polygon1, polygon2) {
```

```
    var axes, projection1, projection2;

    axes = polygon1.getAxes();
    axes.push(polygon2.getAxes()); // axes is an axis array

    for (each axis in axes) {
       projection1 = polygon1.project(axis);
       projection2 = polygon2.project(axis);

       if ( ! projection1.overlaps(projection2))
          return false; // Separation means no collision
    }
    return true; // No separation on any axis means collision
}
```

在将上述伪代码转写为实际程序之前，我们得先来思考这几个问题：

- 如何确定多边形的各个投影轴？
- 如何将多边形投射到某条分离轴上？
- 如何检测两段投影是否发生重叠？

下面数个小节将回答这些问题，并告诉大家如何在 Canvas 中用分离轴定理检测碰撞。

8.4.1.1 投影轴

在用分离轴定理检测多边形碰撞时，必须能够根据某个给定的多边形平面（polygon face），找到它的所有投影轴。图 8-16 演示了怎样根据多边形的任意一条边来确定其投影轴。

在图 8-16 中，我们使用一条从 p1 指向 p2 的向量来表示多边形的某个边，这个向量叫做边缘向量（edge vector）。

在分离轴定理中，还需确定一条垂直于边缘向量的法向量（normal vector），我们叫它“边缘法向量”（edge normal vector）。

在图 8-16 之中，投影轴位于受测多边形的右下方。其实这条轴的位置无所谓，因为无论它在哪里，其长度都是无限的，故而多边形在该轴上的投影也都是一样的。这条轴的方向才是关键。

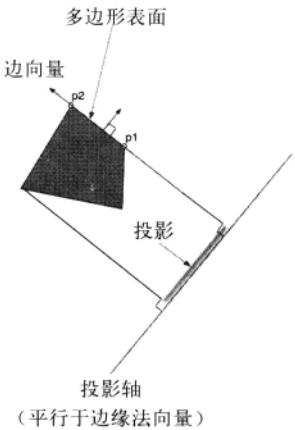


图 8-16 将多边形投射到与其某个边垂直的投影轴上

给定 p1、p2 两个点，可以用下面这段代码来创建一条垂直于边缘向量的投影轴：

```
// Getting an axis to project onto. That axis
// is normal to the edge from p1 to p2

var v1 = new Vector(p1.x, p1.y);
    v2 = new Vector(p2.x, p2.y);
    axis = v1.edge(v2).normal();
```

程序清单 8-3 列出了上段代码中 Vector 对象的实现代码。

计算向量的长度要用到勾股定理，也就是直角三角形斜边的平方等于两直角边的平方和。

向量之间可以进行加减法，也可以相乘。两向量的纯量乘积又叫做点积（dot product），因为这个乘积可以用一个点来表示。要想求得两个向量之间的边缘向量，可以把二者相减。

程序清单 8-3　Vector 对象

```
// Constructor.......................................................

var Vector = function(x, y) {
      this.x = x;
      this.y = y;
};

// Prototype........................................................

Vector.prototype = {
   getMagnitude: function () {
      return Math.sqrt(Math.pow(this.x, 2) +
                       Math.pow(this.y, 2));
   },

   add: function (vector) {
      var v = new Vector();
      v.x = this.x + vector.x;
      v.y = this.y + vector.y;
      return v;
   },

   subtract: function (vector) {
      var v = new Vector();
      v.x = this.x - vector.x;
      v.y = this.y - vector.y;
      return v;
   },

   dotProduct: function (vector) {
      return this.x * vector.x +
             this.y * vector.y;
   },

   edge: function (vector) {
      return  this.subtract(vector);
   },
   perpendicular: function () {
      var v = new Vector();
      v.x = this.y;
      v.y = 0-this.x;
      return v;
   },
   normalize: function () {
      var v = new Vector(0, 0),
          m = this.getMagnitude();

      if (m != 0) {
         v.x = this.x / m;
         v.y = this.y / m;
      }
      return v;
   },

   normal: function  () {
      var p = this.perpendicular();
      return p.normalize();
   }
};
```

在 vector 对象上调用 perpendicular() 方法，就可以得到与之垂直的向量。如果调用的是 normal() 方法，那么就可以得到的正规化（normalize）之后的垂直向量。所谓“正规化”，就是将向量的长度缩减为 1，因此，正规化之后的向量又叫做单位向量（unit vector）。

由于单位向量的长度总是 1，所以主要用它来指示方向。例如前段代码中的那个 axis 变量，就是为了指示投影轴的方向而设的。

现在给定多边形上相邻的两个顶点，我们就可以据此来创建一条用于标识投影轴的向量了。接下去要做的，就是把两个受测多边形的每一条边所对应的投影轴都确定下来，并且检测多边形在每条轴上的投影是否分离。如果找到了两个相互分离的投影，那就说明两者未碰撞，否则，我们就判定这两个多边形发生了碰撞。在这一系列操作，需要用到“投影”（Projection）这个概念。

8.4.1.2 投影

正如程序清单 8-4 所示，这个 Projection 对象很简单：用某条轴上的最小值与最大值即可表示一段投影。Projection 对象可以告诉开发者它是否与另外一段投影发生重叠，这就是该对象的全部功能。

我们已经写好 Vector 与 Projection 对象，而且还能根据多边形的某个边来创建一条投影轴，那么现在就该研究如何运用分离轴定理实现多边形与其他形状的碰撞检测了。

程序清单 8-4 Projection 对象

```
var Projection = function (min, max) {
    this.min = min;
    this.max = max;
};

Projection.prototype = {
   overlaps: function (projection) {
      return this.max > projection.min && projection.max > this.min;
   }
};
```

8.4.1.3 形状与多边形

本章的最终目标是要用分离轴定理来实现多边形、圆形、图像及精灵的碰撞检测。要做到这一点，首先得实现一个含有如下方法的 Shape 对象用以表示某种形状：

- boolean collidesWith(anotherShape)
- Vector[] getAxes()
- boolean separationOnAxes(axes, anotherShape)
- Projection project(axis)

collidesWith() 方法的用法如下：

```
if (shape1.collidesWith(shape2)) {
   ...
}
```

在上述代码中，如果 shape1 与 shape2 相碰，那么 collidesWith() 方法就会返回 true。

图 8-17 画出了受测多边形的所有投影轴，以及它在各轴上的投影。在图 8-16 中我们说过，每条投影轴都平行于一条和多边形某边相垂直的法向量。

Shape.getAxes() 方法返回一个向量数组，其中每个元素都表示一条投影轴，而 Shape.project(axis) 方法则返回该多边形在某条轴上的投影。

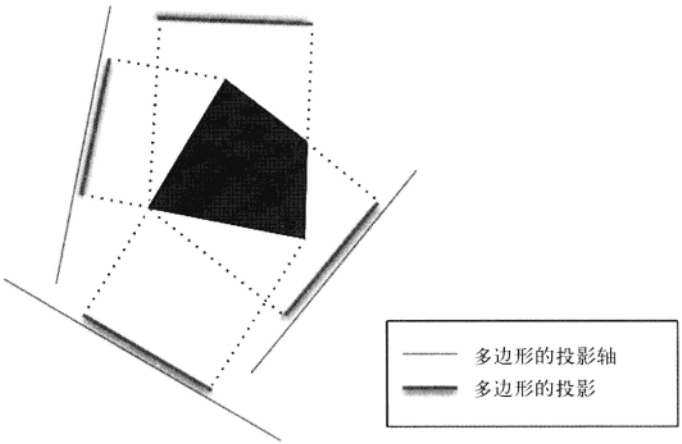


图 8-17　Shape.getAxes() 方法返回多边形的所有投影轴，而 Shape.project() 方法则返回它在某条轴上的投影

程序清单 8-5 列出了上述 4 个 Shape 对象方法的实现代码。

程序清单 8-5　Shape 对象中用于碰撞检测的方法

```
Shape.prototype  = {
   ...
   // This shape collides with otherShape if there is no separation
   // along either of the shape's axes.

   collidesWith: function (otherShape) {
      var axes = this.getAxes().concat(otherShape.getAxes());
      return !this.separationOnAxes(axes, otherShape);
   },

   // Is there separation between this shape and
   // otherShape along any of the specified axes?

   separationOnAxes: function (axes, otherShape) {
      for (var i=0; i < axes.length; ++i) {

         axis = axes[i];
         projection1 = otherShape.project(axis);
         projection2 = this.project(axis);
         if (! projection1.overlaps(projection2)) {
            return true;
         }
      }
      return false;
   },

   // Get this shape's axes, to be used for collision detection by SAT

   getAxes: function () {
      throw 'getAxes() not implemented';
   },
   ...

   // Project this shape onto the specified axis

   project: function (axis) {
```

```
        throw 'project(axis) not implemented';
    }
};
```

collidesWith() 方法在每个 Shape 对象上调用 getAxes() 方法，然后将获取到的投影轴传递给 separationOnAxes() 方法，后者会将两个受测对象分别投射到每条投影轴之上，只要在某条轴上发现了相互分离的投影，那么该方法就立刻返回 true，反之，如果在所有的投影轴上都找不到相互分离的投影，那么它就返回 false。

多边形与圆形的 getAxes() 及 project() 方法各不相同，所以我们得分别在 Polygon 和 Circle 对象中实现它们。先看 Polygon 对象的方法实现代码：

```
// Polygons have an array of points

var Polygon = function () {
   this.points =  [];
   ...
};
...

// Polygons are shapes

Polygon.prototype = new Shape();
...

// Projects each point in the polygon onto the
// specified axis and then returns a projection
// with the minimum and maximum of those projected points.

Polygon.prototype.project = function (axis) {
   var scalars = [],
       v = new Vector();

   this.points.forEach( function (point) {
      v.x = point.x;
      v.y = point.y;
      scalars.push(v.dotProduct(axis));
   });

   return new Projection(Math.min.apply(Math, scalars),
                         Math.max.apply(Math, scalars));
};

// Returns all of the polygon's axes needed for
// collision detection testing with SAT

Polygon.prototype.getAxes = function () {
   var v1 = new Vector(),
       v2 = new Vector(),
       axes = [];

   for (var i=0; i < this.points.length-1; i++) {
      v1.x = this.points[i].x;
      v1.y = this.points[i].y;

      v2.x = this.points[i+1].x;
      v2.y = this.points[i+1].y;

      axes.push(v1.edge(v2).normal());
   }

   return axes;
};
```

现在的 Polygon 对象在 Shape 对象的基础上又实现了 project() 与 getAxes() 方法，于是我们可以在碰撞检测算法中使用它了。

Circle 对象的 project() 与 getAxes() 方法的实现，将在 8.4.1.5 小节中讲解。

程序清单 8-6 与程序清单 8-7 分别列出了 Shape 对象与 Polygon 对象的全部代码。

程序清单 8-6　Shape 对象

```
// Constructor.......................................................

var Shape = function () {
   this.x = undefined;
   this.y = undefined;
   this.strokeStyle = 'rgba(255, 253, 208, 0.9)';
   this.fillStyle = 'rgba(147, 197, 114, 0.8)';
};

// Prototype........................................................

Shape.prototype = {
   // Collision detection methods....................................

   collidesWith: function (shape) {
      var axes = this.getAxes().concat(shape.getAxes());
      return !this.separationOnAxes(axes, shape);
   },

   separationOnAxes: function (axes, shape) {
      for (var i=0; i < axes.length; ++i)  {
         axis = axes[i];
         projection1 = shape.project(axis);
         projection2 = this.project(axis);

         if (! projection1.overlaps(projection2)) {
            return  true; // Don't have to test remaining axes
         }
      }
      return false;
   },

   project: function (axis) {
      throw 'project(axis) not implemented';
   },

   getAxes: function () {
      throw 'getAxes() not implemented';
   },

   move: function (dx, dy) {
      throw 'move(dx, dy) not implemented';
   },

   // Drawing methods...............................................

   createPath: function (context) {
      throw 'createPath(context) not implemented';
   },

   fill: function (context) {
      context.save();
      context.fillStyle = this.fillStyle;
```

```
        this.createPath(context);
        context.fill();
        context.restore();
    },

    stroke: function (context) {
        context.save();
        context.strokeStyle = this.strokeStyle;
        this.createPath(context);
        context.stroke();
        context.restore();
    },

    isPointInPath: function (context, x, y) {
        this.createPath(context);
        return context.isPointInPath(x, y);
    },
};
```

程序清单 8-7　Polygon 对象（附 Point 对象）

```
// Constructor.......................................................

var Point = function (x, y) {
    this.x = x;
    this.y = y;
};

var Polygon = function () {
    this.points = [];
    this.strokeStyle = 'blue';
    this.fillStyle = 'white';
};

// Prototype........................................................

Polygon.prototype = new Shape();

Polygon.prototype.getAxes = function () {
    var v1 = new Vector(),
        v2 = new Vector(),
        axes = [];

    for (var i=0; i < this.points.length-1; i++) {
        v1.x = this.points[i].x;
        v1.y = this.points[i].y;

        v2.x = this.points[i+1].x;
        v2.y = this.points[i+1].y;

        axes.push(v1.edge(v2).normal());
    }

    v1.x = this.points[this.points.length-1].x;
    v1.y = this.points[this.points.length-1].y;

    v2.x = this.points[0].x;
    v2.y = this.points[0].y;

    axes.push(v1.edge(v2).normal());

    return axes;
```

```
};

Polygon.prototype.project = function (axis) {
   var scalars = [],
       v = new Vector();

   this.points.forEach( function (point) {
      v.x = point.x;
      v.y = point.y;
      scalars.push(v.dotProduct(axis));
   });

   return new Projection(Math.min.apply(Math, scalars),
                         Math.max.apply(Math, scalars));
};

Polygon.prototype.addPoint = function (x, y) {
   this.points.push(new Point(x,y));
};

Polygon.prototype.createPath = function (context) {
   if (this.points.length === 0)
      return;

   context.beginPath();
   context.moveTo(this.points[0].x,
                  this.points[0].y);

   for (var i=0; i < this.points.length; ++i) {
      context.lineTo(this.points[i].x,
                     this.points[i].y);
   }

   context.closePath();
};

Polygon.prototype.move = function (dx, dy) {
   for (var i=0, point; i < this.points.length; ++i) {
      point = this.points[i];
      point.x += dx;
      point.y += dy;
   }
};
```

提示：以多边形对象创建的路径会自动闭合

请注意程序清单 8.7 中的 Polygon.createPath() 方法，它在返回前会调用 Canvas 绘图环境对象的 closePath() 方法，自动将路径闭合。

之所以要这么做，是基于两个原因。首先，不用为了闭合路径而去手工安插一个与多边形第一个顶点相重合的点。其次，使用刚才那个办法来闭合路径，本身就不明智。在渲染时，会多画出来一条连接第一个顶点与最后一个顶点（该点与前者重合）的连线。

8.4.1.4 多边形之间的碰撞

我们已经将运用分离轴定理来检测碰撞所需的全部组件都准备好了，现在就来看看图 8-18 中的这个应用程序是如何检测多边形碰撞的。

该程序创建了三个可以拖动的多边形。如果把一个多边形拖放到另一个之上，那么程序将在

Canvas 左上角显示“collision”字样，文字颜色与碰撞前静止的那个多边形相同。

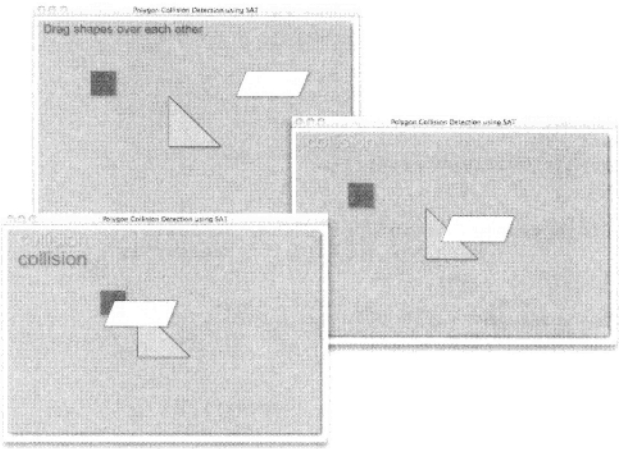


图 8-18 检测多边形之间的碰撞

程序中检测碰撞的代码如下：

```
function detectCollisions() {
   var textY = 30,
       numShapes = shapes.length,
       shape,
       i;
   if (shapeBeingDragged) {
      for(i = 0; i < numShapes; ++i) {
         shape = shapes[i];

         if (shape !== shapeBeingDragged) {
            if (shapeBeingDragged.collidesWith(shape)) {
               context.fillStyle = shape.fillStyle;
               context.fillText('collision', 20, textY);
               textY += 40;
            }
         }
      }
   }
}
```

如果用户正在拖动某个图形，那么程序就把该多边形的 shapeBeingDragged 变量设为 true。在检测碰撞时，detectCollisions() 方法会先检查这个变量是不是 true。如果是，那么就表示当前这个图形正在被拖动。

假如 shapeBeingDragged 变量是 true，那么 detectCollisions() 方法就会遍历所有图形对象，检测当前正在拖动的图形有没有和它们相碰撞。

程序清单 8-8 列出了本程序的全部 JavaScript 代码。

程序清单 8-8 多边形之间的碰撞检测（JavaScript 代码）

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    shapes = [],
```

```
    polygonPoints = [
      // The paths described by these point arrays
      // are open. They are explicitly closed by
      // Polygon.createPath()

       [ new Point(250, 150), new Point(250, 250),
         new Point(350, 250) ],

       [ new Point(100, 100), new Point(100, 150),
         new Point(150, 150), new Point(150, 100) ],

       [ new Point(400, 100), new Point(380, 150),
         new Point(500, 150), new Point(520, 100) ]
    ],

    polygonStrokeStyles = [ 'blue', 'yellow', 'red'],
    polygonFillStyles   = [ 'rgba(255,255,0,0.7)',
                            'rgba(100,140,230,0.6)',
                            'rgba(255,255,255,0.8)' ],

    mousedown = { x: 0, y: 0 },
    lastdrag = { x: 0, y: 0 },
    shapeBeingDragged = undefined;

// Functions.......................................................

function windowToCanvas(x, y) {
   var bbox = canvas.getBoundingClientRect();
   return { x: x - bbox.left * (canvas.width  / bbox.width),
            y: y - bbox.top  * (canvas.height / bbox.height)
          };
}
function drawShapes() {
   shapes.forEach( function (shape) {
      shape.stroke(context);
      shape.fill(context);
   });
}

function detectCollisions() {
   var textY = 30,
       numShapes = shapes.length,
       shape,
       i;

   if (shapeBeingDragged) {
      for(i = 0; i < numShapes; ++i) {
         shape = shapes[i];

         if (shape !== shapeBeingDragged) {
            if (shapeBeingDragged.collidesWith(shape))      {
               context.fillStyle = shape.fillStyle;
               context.fillText('collision', 20, textY);
               textY += 40;
            }
         }
      }
   }
}

// Event handlers.................................................
```

```
canvas.onmousedown = function  (e) {
   var location = windowToCanvas(e.clientX, e.clientY);

   shapes.forEach( function (shape) {
      if (shape.isPointInPath(context, location.x, location.y)) {
         shapeBeingDragged = shape;
         mousedown.x = location.x;
         mousedown.y = location.y;
         lastdrag.x = location.x;
         lastdrag.y = location.y;
      }
   });
};

canvas.onmousemove = function (e) {
   var location,
       dragVector;

      if (shapeBeingDragged !== undefined) {
         location = windowToCanvas(e.clientX, e.clientYe);
         dragVector = { x: location.x - lastdrag.x,
                        y: location.y - lastdrag.y
                      };

         shapeBeingDragged.move(dragVector.x, dragVector.y);

         lastdrag.x = location.x;
         lastdrag.y = location.y;

         context.clearRect(0, 0, canvas.width, canvas.height);
         drawShapes();
         detectCollisions();
      }
   };

   canvas.onmouseup = function (e) {
      shapeBeingDragged = undefined;
   };

  // Initialization.....................................................

  for (var i=0; i < polygonPoints.length; ++i) {
     var polygon = new Polygon(),
         points = polygonPoints[i];

     polygon.strokeStyle = polygonStrokeStyles[i];
     polygon.fillStyle = polygonFillStyles[i];

     points.forEach( function (point) {
        polygon.addPoint(point.x, point.y);
     });

     shapes.push(polygon);
  }

  context.shadowColor = 'rgba(100,140,255,0.5)';
  context.shadowBlur = 4;
  context.shadowOffsetX = 2;
  context.shadowOffsetY = 2;
  context.font = '38px Arial';

  drawShapes();
```

```
context.save();
context.fillStyle = 'cornflowerblue';
context.font = '24px Arial';
context.fillText('Drag shapes over each other', 10, 25);
context.restore();
```

在学会了使用分离轴定理来检测多边形之间的碰撞后，我们再将该算法推广到圆形与多边形的之间碰撞检测。

8.4.1.5 圆形与多边形之间的碰撞检测

在上文中读者已经看到了，使用分离轴定理来检测两个多边形之间的碰撞时，要分别将每个多边形投射到所有的投影轴上，然后寻找两多边形在每条轴上的投影是否互相分离。每条投影轴都对应于多边形的某一条边。

然而如果要将分离轴定理运用于圆形，就会发现一个问题：圆可以近似地看成一个有无数条边的正多边形，而我们不可能按照这些边一一进行投影与测试。我们只需要将圆形投射到一条投影轴上即可，这条轴就是圆心与距其最近的多边形顶点之间的连线，如图 8-19 所示。

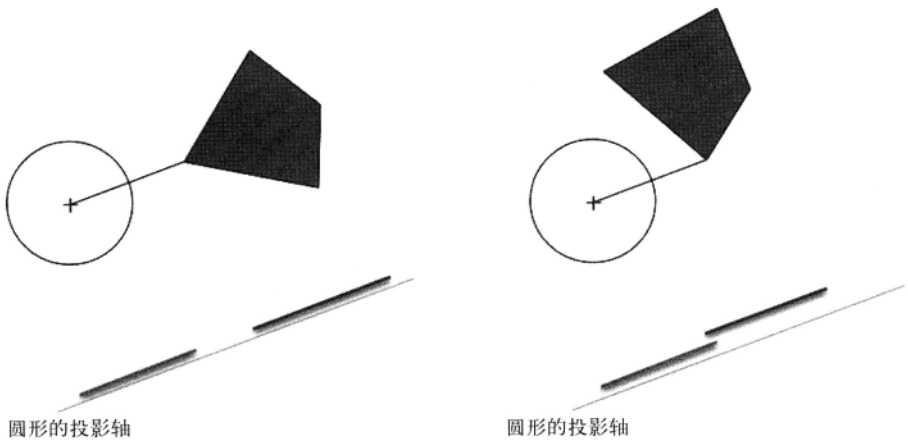


图 8-19　检测多边形与圆形之间的碰撞

在图 8-19 右侧的截图之中，圆形与多边形并未发生碰撞，然而它们在这条轴上的投影却有了重叠。这说明只在圆形的投影轴上进行投射是不够的，还需要将这两个受测物体投射到多边形的各条投影轴上才行。若是我们将图 8-19 中的两个受测物体投射到圆形与多边形的全部投影轴上，那么就可以在其中发现相互分离的投影，如图 8-20 所示。

程序清单 8-9 列出了 Circle 对象的代码。首先要注意的是，该对象的 getAxes() 方法返回 undefined，这是由于单凭圆形自身是无法确定投影轴的。要将图 8-19 中的那条投影轴确定下来，还得知道多边形的位置。

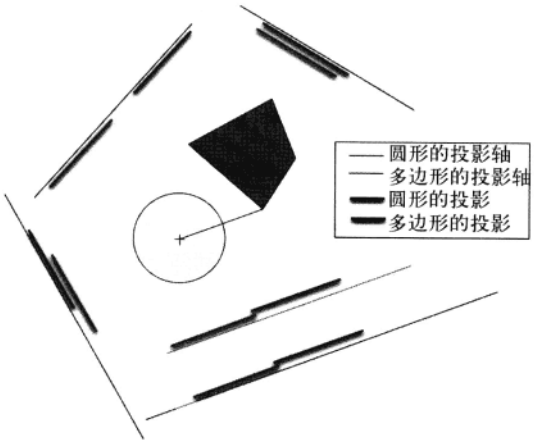


图 8-20 若圆形与多边形未发生碰撞，则必然能在某条轴上找到相互分离的投影

程序清单 8-9 Circle 对象

```
// Constructor.......................................................

var Circle = function (x, y, radius) {
   this.x = x;
   this.y = y;
   this.radius = radius;
   this.strokeStyle = 'rgba(255, 253, 208, 0.9)';
   this.fillStyle = 'rgba(147, 197, 114, 0.8)';
}

// Prototype........................................................

Circle.prototype = new Shape();

Circle.prototype.collidesWith = function (shape) {
   var point, length, min=10000, v1, v2,
       edge, perpendicular, normal,
       axes = shape.getAxes(), distance;

   if (axes === undefined) { // Circle
      distance = Math.sqrt(Math.pow(shape.x - this.x, 2) +
                           Math.pow(shape.y - this.y, 2));

      return distance < Math.abs(this.radius + shape.radius);
   }
   else  { // Polygon
      return polygonCollidesWithCircle(shape, this);
   }
};

Circle.prototype.getAxes = function () {
   return undefined; // Circles have an infinite number of axes
};

Circle.prototype.project = function (axis) {
```

```
    var scalars = [],
        point = new Point(this.x, this.y);
        dotProduct = new Vector(point).dotProduct(axis);

    scalars.push(dotProduct);
    scalars.push(dotProduct + this.radius);
    scalars.push(dotProduct - this.radius);

    return new Projection(Math.min.apply(Math, scalars),
                          Math.max.apply(Math, scalars));
};
Circle.prototype.move = function (dx, dy) {
    this.x += dx;
    this.y += dy;
};
Circle.prototype.createPath = function (context) {
    context.beginPath();
    context.arc(this.x, this.y, this.radius, 0, Math.PI*2, false);
};
```

Circle 对象也实现了 collidesWith() 方法。该方法先在传入图形参数的上调用 getAxes() 方法，看看返回的投射轴是不是 undefined，如果是，那说明传入的对象也是个圆形，于是 Circle.collidesWith() 方法就使用 8.1.2 小节中讲过的办法来检测两个圆之间的碰撞。

如果在传递给 Circle.collidesWith() 方法的参数之上调用 getAxes()，能够返回投影轴的话，则说明该参数是个多边形，于是 Circle.collidesWith() 方法就调用一个名为 polygonCollidesWithCircle() 的方法来检测碰撞。此方法的代码列在程序清单 8-10 之中。

程序清单 8-10　检测多边形与圆形之间的碰撞

```
function getPolygonPointClosestToCircle(polygon, circle) {
    var min = 10000,
        length,
        testPoint,
        closestPoint;

    for (var i=0; i < polygon.points.length; ++i) {
        testPoint = polygon.points[i];
        length = Math.sqrt(Math.pow(testPoint.x - circle.x, 2),
                           Math.pow(testPoint.y - circle.y, 2));
        if (length < min) {
            min = length;
            closestPoint = testPoint;
        }
    }

    return closestPoint;
};

function polygonCollidesWithCircle (polygon, circle) {
    var min=10000, v1, v2,
        edge, perpendicular, normal,
        axes = polygon.getAxes(),
        closestPoint = getPolygonPointClosestToCircle(polygon, circle);

    v1 = new Vector(new Point(circle.x, circle.y));
    v2 = new Vector(new Point(closestPoint.x, closestPoint.y));

    axes.push(v1.subtract(v2).normalize());

    return !polygon.separationOnAxes(axes, circle);
};
```

在程序清单 8-10 之中，如果传递给 polygonCollidesWithCircle() 方法的多边形与圆形对象发生了碰撞，那么该方法就返回 true。此方法会创建一条投影轴，它连接圆心与距其最近的多边形顶点。这条轴与多边形对象所返回的其他投影轴一起被用于碰撞检测，该方法会检查受测图形在这些轴上面的投影有没有相互分离。

虽说现在的 Circle 对象可以检测与多边形对象的碰撞，不过反过来还不行：Polygon 对象目前还不能检测它与圆形之间的碰撞。要解决这个问题，我们可以按照程序清单 8-11 所示，重构 Polygon.collidesWith() 方法。

程序清单 8-11　重构之后的 Polygon.collidesWith() 方法

```
Polygon.prototype.collidesWith = function (shape) {
   var axes = shape.getAxes();

   if (axes === undefined) {
      return polygonCollidesWithCircle(this, shape);
   }
   else {
      axes.concat(this.getAxes());
      return !this.separationOnAxes(axes, shape);
   }
};
...
```

图 8-21 所示的应用程序可根据分离轴定理来检测多边形与圆形之间的碰撞。

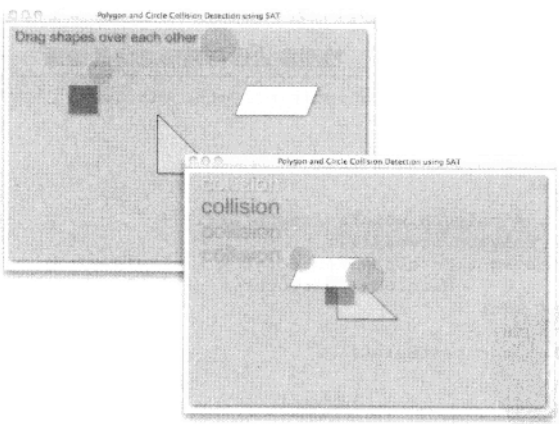


图 8-21　检测多边形与圆形之间的碰撞

图 8-21 所示应用程序的代码与图 8-18 中那个检测多边形碰撞的程序相似，后者的代码列在程序清单 8-8 之中。二者的区别在于，图 8-21 中的应用程序又多创建了两个圆形，并把它们添加到程序的 shapes 数组中。

```
...
circle1 = new Circle(150, 75, 20);
circle2 = new Circle(350, 25, 30);
...
shapes.push(circle1);
```

```
shapes.push(circle2);
...
```

读者已经学到了如何使用分离轴定理来检测多边形与圆形的碰撞，接下来我们再看看如何用该算法检测图像与精灵的碰撞。

8.4.1.6 图像与精灵的碰撞检测

能够检测诸如多边形与圆形这样任意图形之间的碰撞是很重要的，不过，还有一项同样重要的技术要掌握，那就是检测图像与精灵之间的碰撞。

图 8-22 所示的应用程序，显示了数个多边形、一幅网球图像，以及一个高尔夫球精灵。所有这些物体都可以拖动，应用程序会在用户拖动的过程中检测当前物体是否与程序中的其他物体相碰。

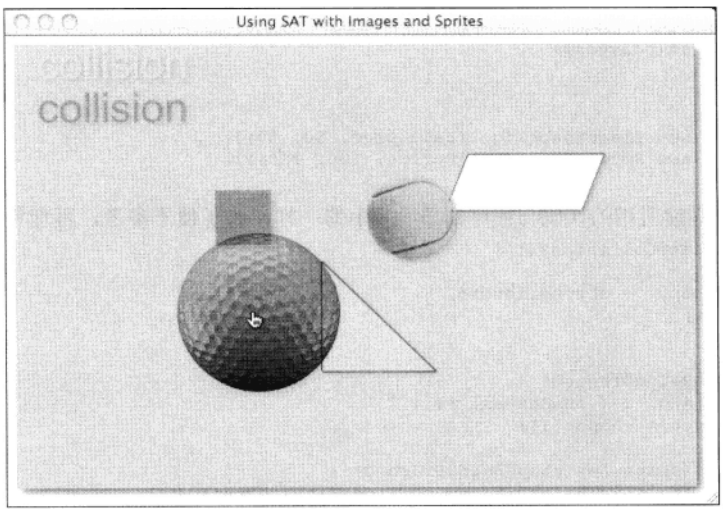


图 8-22 运用分离轴定理检测图像与精灵的碰撞

该应用程序创建了三个 Polygon 对象、一个 ImageShape 对象，以及一个 SpriteShape 对象，并将上述 5 个对象都加入 shapes 数组。这段代码如下：

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    shapes = [],

    ballSprite = new Sprite('ball',
                            new ImagePainter('tennis-ball.png')),

    polygonPoints = [
      [ new Point(250, 150), new Point(250, 250),
        new Point(350, 250), new Point(250, 150) ],

      [ new Point(100, 100), new Point(100, 150),
        new Point(150, 150), new Point(150, 100),
        new Point(100, 100) ],

      [ new Point(400, 100), new Point(380, 150),
        new Point(500, 150), new Point(520, 100),
```

```
        new Point(400, 100) ]
    ],

    polygonStrokeStyles =  [ 'blue', 'yellow', 'red'],
    polygonFillStyles   =  [ 'rgba(255,255,0,0.7)',
                             'rgba(100,140,230,0.6)',
                             'rgba(255,255,255,0.8)' ];

for (var i=0; i < polygonPoints.length; ++i) {
   var polygon = new Polygon(),
       points = polygonPoints[i];

   polygon.strokeStyle = polygonStrokeStyles[i];
   polygon.fillStyle = polygonFillStyles[i];

   points.forEach( function (point) {
      polygon.addPoint(point.x, point.y);
   });

   shapes.push(polygon);
}
...

shapes.push(new ImageShape('golfball.png', 50, 50));
shapes.push(new SpriteShape(ballSprite, 100, 100));
...
```

本程序检测碰撞所用的代码与程序清单 8-8 中的一样，为了便于参考，再度将其列出：

```
function detectCollisions() {
   var textY = 30,
       numShapes = shapes.length,
       shape,
       i;

   if (shapeBeingDragged) {
      for(i = 0; i < numShapes; ++i) {
         shape = shapes[i];

         if (shape !== shapeBeingDragged) {
            if (shapeBeingDragged.collidesWith(shape)) {
               context.fillStyle = shape.fillStyle;
               context.fillText('collision', 20, textY);
               textY += 40;
            }
         }
      }
   }
}
```

程序清单 8-12 与程序清单 8-13 分别列出了 ImageShape 与 SpriteShape 对象的代码。

程序清单 8-12　ImageShape 对象

```
// Constructor.......................................................

var ImageShape = function(imageSource, x, y, w, h) {
   var self = this;

   this.image = new Image();
   this.imageLoaded = false;
   this.points = [ new Point(x,y) ];
   this.x = x;
   this.y = y;

   this.image.src = imageSource;
```

```
   this.image.addEventListener('load', function (e) {
      self.setPolygonPoints();
      self.imageLoaded = true;
   }, false);
}
// Prototype.......................................................

ImageShape.prototype = new Polygon();

ImageShape.prototype.fill = function (context) { }; // Nothing to do

ImageShape.prototype.setPolygonPoints = function() {
   this.points.push(new Point(this.x + this.image.width, this.y));
   this.points.push(new Point(this.x + this.image.width,
                              this.y + this.image.height));
   this.points.push(new Point(this.x, this.y + this.image.height));
};

ImageShape.prototype.drawImage = function (context) {
   context.drawImage(this.image, this.points[0].x, this.points[0].y);
};
ImageShape.prototype.stroke = function (context) {
   var self = this;

   if (this.imageLoaded)  {
      context.drawImage(this.image,
                        this.points[0].x, this.points[0].y);
   }
   else {
      this.image.addEventListener('load', function (e) {
         self.drawImage(context);
      }, false);
   }
};
```

ImageShape 与 SpriteShape 都是 Polygon 对象，它们分别代表图像与精灵外围的边界框。你可以根据某幅图像或某个精灵来创建相应的 ImageShape 或 SpriteShape 对象，用以检测它们之间的碰撞。

程序清单 8-13　SpriteShape 对象

```
// Constructor.....................................................
var SpriteShape = function (sprite, x, y) {
   this.sprite = sprite;
   this.x = x;
   this.y = y;
   sprite.left = x;
   sprite.top = y;
   this.setPolygonPoints();
};

// Prototype.......................................................
SpriteShape.prototype = new Polygon();
SpriteShape.prototype.move = function (dx, dy) {
   var point, x;
   for(var i=0; i < this.points.length; ++i) {
      point = this.points[i];
      point.x += dx;
```

```
        point.y += dy;
    }
    this.sprite.left = this.points[0].x;
    this.sprite.top = this.points[0].y;
};

SpriteShape.prototype.fill = function (context) { };

SpriteShape.prototype.setPolygonPoints = function() {
    this.points.push(new Point(this.x, this.y));
    this.points.push(new Point(this.x + this.sprite.width, this.y));
    this.points.push(new Point(this.x + this.sprite.width,
                               this.y + this.sprite.height));
    this.points.push(new Point(this.x, this.y + this.sprite.height));
};

SpriteShape.prototype.stroke = function (context) {
    this.sprite.paint(context);
};
```

8.4.2 根据最小平移向量应对碰撞

在检测到多边形、圆形、图像与精灵之间的碰撞后，我们需要应对它。

通常来说，如果碰撞之后，相撞的双方依然存在，那么就需要将二者分开。分开之后，可以使原来相撞的两个物体彼此弹开，也可以让它们粘在一起，还可以根据具体需要来实现其他行为。不过首先要做的，还是将二者分开，这就要用到最小平移向量（Minimum Translation Vector，MTV）了。

8.4.2.1 最小平移向量

最小平移向量是指，如果要让某个物体不再与另外一个物体相撞，所需移动的最小距离。图 8-23 用两个相互碰撞的多边形来说明最小平移向量的概念。

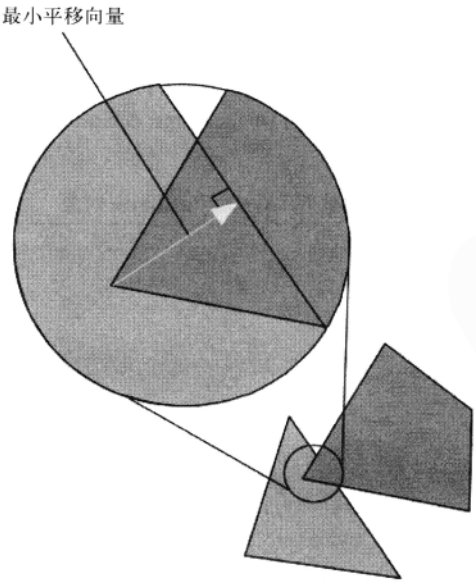


图 8-23 两个相互碰撞的多边形之间的最小平移向量

程序清单 8-14 列出了一段简单的 JavaScript 代码，我们可以用它来实现最小平移向量。

程序清单 8-14　最小平移向量

```
var MinimumTranslationVector = function (axis, overlap) {
   this.axis = axis;        // axis is a vector
   this.overlap = overlap; // overlap is a scalar (single value)
};
```

MinimumTranslationVector 对象的 axis 属性，是一个指示方向的单位向量，而它的 overlap 属性，则表示两图形在该方向上发生重叠的部分所占据的长度。

在使用分离轴定理判断受测图形在各条投射轴上的投影是否互相分离时，我们可以顺便计算出最小平移向量。回忆一下程序清单 8-5 中的 Shape.separationOnAxes() 方法，它在投影轴数组中迭代，将受测图形分别投射到每条轴上，并且查看投射后的阴影是否发生了分离，那段代码如下：

```
Shape.prototype = {
   ...
   separationOnAxes: function (axes, shape) {
     for (var i=0; i < axes.length;  ++i) {
        axis = axes[i];
        projection1 = shape.project(axis);
        projection2 = this.project(axis);
        if (! projection1.overlaps(projection2)) {
           return true;
        }
     }
     return false;
   }
   ...
}
```

separationOnAxes() 方法所返回的 boolean 值，表示是否在某条轴上找到了相互分离的投影。

程序清单 8-15 列出了另外一种实现 separationOnAxes() 方法的办法，这种办法同时还能算出最小平移向量。新的方法更名为 minimumTranslationVector，并且不再返回 boolean 值，而是返回一个 MinimumTranslationVector 对象。

与 separationOnAxes() 一样，minimumTranslationVector() 方法也会将受测图形投射到每条轴上，并检测其投影是否互相分离。二者的区别在于，minimumTranslationVector() 方法还会记录下受测图形重叠部分长度最小的那条轴。

程序清单 8-15　Shape.minimumTranslationVector(axes, shape) 方法的代码

```
Shape.prototype = {
...
   minimumTranslationVector: function (axes, shape) {
      var minimumOverlap = 100000,
          overlap,
          axisWithSmallestOverlap;

      for (var i=0; i < axes.length; ++i) {
         axis = axes[i];
         projection1 = shape.project(axis);
         projection2 = this.project(axis);
         overlap = projection1.overlap(projection2);

         if (overlap === 0) {
            return { axis: undefined, // No collision
```

```
                    overlap: 0
                 };
            }
            else {
               if (overlap < minimumOverlap) {
                  minimumOverlap = overlap;
                  axisWithSmallestOverlap = axis;
               }
            }
         }
         return { axis: axisWithSmallestOverlap, // Collision
                  overlap: minimumOverlap
                };
      }
      ...
   }
```

如果在某条轴上找到了相互分离的投影，则说明未发生碰撞，也就不存在最小平移向量。在这种情况下，minimumTranslationVector() 方法所返回的 MinimumTranslationVector 对象中，axis 的值为 undefined，而 overlap 的值为 0。否则，此方法返回的 MinimumTranslationVector 对象，就会包含重叠部分长度最小的那条轴，以及该轴上重叠部分的长度。

程序清单 8-16 中的方法都利用了 Shape.minimumTranslationVector() 所提供的功能，它们不仅可以检测碰撞，而且还能返回指向最小平移向量的引用。

程序清单 8-16　检测碰撞并计算最小平移向量

```
// Collision between two polygons

function polygonCollidesWithPolygon (p1, p2) {
   var mtv1 = p1.minimumTranslationVector(p1.getAxes(), p2),
       mtv2 = p1.minimumTranslationVector(p2.getAxes(), p2);

   if (mtv1.overlap === 0 && mtv2.overlap === 0)
      return { axis: undefined, overlap: 0 };
   else
      return mtv1.overlap < mtv2.overlap ? mtv1 : mtv2;
}

// Collision between two circles

function circleCollidesWithCircle (c1, c2) {
   var distance = Math.sqrt( Math.pow(c2.x - c1.x, 2) +
                             Math.pow(c2.y - c1.y, 2)),
       overlap = Math.abs(c1.radius + c2.radius) - distance;
   return overlap < 0 ?

      new MinimumTranslationVector(undefined, 0) :
      new MinimumTranslationVector(undefined, overlap);
}

// Collision between a polygon and a circle

function polygonCollidesWithCircle (polygon, circle) {
   var axes = polygon.getAxes(),
       closestPoint = getPolygonPointClosestToCircle(polygon, circle);

   axes.push(getCircleAxis(circle, polygon, closestPoint));

   return polygon.minimumTranslationVector(axes, circle);
}
```

程序清单 8-17 列出了重构之后的 collidesWith() 方法，它可以检测圆形与多边形之间的碰撞。重构后的方法用到了程序清单 8-15 所列的 minimumTranslationVector() 函数。

在学会了检测碰撞以及计算最小平移向量的方法之后，接下来看看如何利用好这个向量。给定两个相互碰撞的图形以及它们之间的最小平移向量，我们就可以像程序清单 8-18 这样将其分离开。

程序清单 8-17　重构之后的 collidesWith() 方法

```
// Circles.............................................................

Circle.prototype.collidesWith = function (shape) {
   if (shape.radius === undefined) {
      return polygonCollidesWithCircle(shape, this);
   }
   else {
      return circleCollidesWithCircle(this, shape);
   }
};

// Polygons............................................................

Polygon.prototype.collidesWith = function (shape) {
   if (shape.radius !== undefined) {
      return polygonCollidesWithCircle(this, shape);
   }
   else {
      return polygonCollidesWithPolygon(this, shape);
   }
};
```

程序清单 8-18 所列的 separate() 方法可以用于多边形及圆形。正如 8.4.1.5 小节所述，针对圆形进行碰撞检测之后，所返回的最小平移向量，其 axis 属性总是 undefined。如果发生了这种情况，那么 separate() 方法就会创建一条表示速度方向的单位向量，并将其当做 axis 属性来用。

因为圆形的移动位置是由其速度决定的，所以使用代表速度的单位向量就可以计算出将圆形从碰撞中分离所需要移动的距离。虽说沿着 separate() 方法算出的那条轴来平移，所要移动的距离不见得就是最小的，不过，该距离已经充分接近理论上的最小值了。尽管比理想情况下的移动距离要稍微远一点，然而根据这个平移向量来移动，我们还是能够将圆形从碰撞中分离开的。

最小平移向量的基本用法就是将两个相互碰撞的物体分开。除此以外，它还有另外两个用途：可以让两个物体粘连在一起，也可以让它们彼此弹开。

程序清单 8-18　将两个相互碰撞的图形分离

```
// Move the shape that's moving (shapeMoving) out of collision

function separate(shapeMoving, mtv) {
   var dx,
       dy,
       velocityMagnitude,
       point;

   if (mtv.axis === undefined) { // circle
      point = new Point();
      velocityMagnitude = Math.sqrt(Math.pow(velocity.x, 2) +
                                    Math.pow(velocity.y, 2));

      point.x = velocity.x / velocityMagnitude;
```

```
        point.y = velocity.y / velocityMagnitude;

        mtv.axis = new Vector(point);
    }

    dy = mtv.axis.y * mtv.overlap;
    dx = mtv.axis.x * mtv.overlap

    if ((dx < 0 && velocity.x < 0) || // Don't move in same direction
        (dx > 0 && velocity.x > 0)) {
        dx = -dx;
    }

    if ((dy < 0 && velocity.y < 0) || // Don't move in same direction
        (dy > 0 && velocity.y > 0)) {
        dy = -dy;
    }

    shapeMoving.move(dx, dy);
}
```

8.4.2.2 利用最小平移向量使两个物体粘在一起

图 8-24 所示应用程序含有几个圆形及多边形。点击某个图形后，应用程序就让它运动起来。图形在碰到 Canvas 的边界后会被弹回，并继续运动，直到它与另外某个图形相撞。

一旦发生了碰撞，应用程序就会让正在移动的这个图形停下来，并且在 0.5 秒后，将这两个碰撞的物体分开。由于刚才移动的那个图形现在停下来了，所以这两个物体看上去好像是粘在了一起，如图 8-24 下方截图所示。

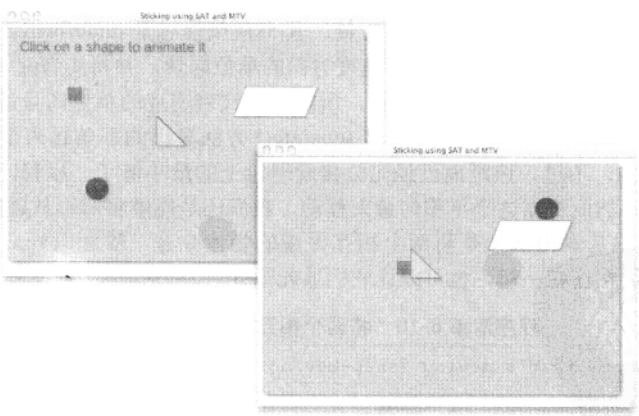


图 8-24 利用最小平移向量使两个物体粘在一起

程序清单 8-19 列出了图 8-24 所示应用程序的 JavaScript 代码。该程序使用 5.1.3 小节中讲的 window.requestNextAnimationFrame() 方法来播放动画，并且用到了本章讲过的那几个图形对象。

如果程序清单 8-19 所列的 detectCollisions() 函数检测到了碰撞，那么它就调用 stick() 方法，在调用时会将图形对象的 collidesWith() 方法所返回的最小平移向量作为参数传入。

stick() 函数首先检查 mtv.axis 属性的值是不是 undefined。如果是，那么正在移动的这个物体

就是一个圆形，stick() 函数会将最小平移向量的 axis 属性修改表示圆形移动速度的单位向量。

随后，stick() 函数会计算出与其他图形相撞的这个物体在 X 轴和 Y 轴方向上所需移动的距离，并且在 500 毫秒之后将它从另一个物体身上移开。

程序清单 8-19　利用最小平移向量使两个物体粘在一起

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d'),
    shapes = [],
    polygonPoints = [
       [ new Point(250, 150), new Point(250, 200),
         new Point(300, 200) ],

       [ new Point(100, 100), new Point(100, 125),
         new Point(125, 125), new Point(125, 100) ],

       [ new Point(400, 100), new Point(380, 150),
         new Point(500, 150), new Point(520, 100) ],
    ],

    polygonStrokeStyles = [ 'blue', 'yellow', 'red'],
    polygonFillStyles   = [ 'rgba(255,255,0,0.7)',
                            'rgba(100,140,230,0.6)',
                            'rgba(255,255,255,0.8)' ],
    shapeMoving = undefined,
    c1 = new Circle(150, 275, 20),
    c2 = new Circle(350, 350, 30),

    lastTime = undefined,
    velocity = { x: 350, y: 190 },
    lastVelocity = { x: 350, y: 190 },
    STICK_DELAY = 500,
    stuck = false;
    showInstructions = true;

// Functions.......................................................

function windowToCanvas (e) {
   var x = e.x || e.clientX,
       y = e.y || e.clientY,
       bbox = canvas.getBoundingClientRect();

   return { x: x - bbox.left * (canvas.width / bbox.width),
            y: y - bbox.top  * (canvas.height / bbox.height)
          };
};

function drawShapes() {
   shapes.forEach( function (shape) {
      shape.stroke(context);
      shape.fill(context);
   });
}
function stick(mtv) {
   var dx,
       dy,
       velocityMagnitude,
       point;

   if (mtv.axis === undefined) { // The moving object is a circle.
```

```
   point = new Point();
   velocityMagnitude = Math.sqrt(Math.pow(velocity.x, 2) +
                                 Math.pow(velocity.y, 2));

   // Point the MTV axis in the direction of the circle's velocity.

   point.x = velocity.x / velocityMagnitude;
   point.y = velocity.y / velocityMagnitude;

   mtv.axis = new Vector(point);
}

// Calculate delta X and delta Y. The mtv.axis is a unit vector
// indicating direction, and the overlap is the magnitude of
// the translation vector.

dx = mtv.axis.x * mtv.overlap;
dy = mtv.axis.y * mtv.overlap;

// If deltas and velocity are in the same direction,
// turn deltas around.

if ((dx < 0 && velocity.x < 0) || (dx > 0 && velocity.x > 0))
   dx = -dx;

if ((dy < 0 && velocity.y < 0) || (dy > 0 && velocity.y > 0))
   dy = -dy;

// In STICK_DELAY (500) ms, move the moving shape out of collision

setTimeout(function  ()  {
   shapeMoving.move(dx, dy);
}, STICK_DELAY);

// Reset pertinent variables

   lastVelocity.x = velocity.x;
   lastVelocity.y = velocity.y;
   velocity.x = velocity.y = 0;

   // Don't stick again before STICK_DELAY expires
   stuck = true;
}

function collisionDetected(mtv) {
   return mtv.axis != undefined || mtv.overlap !== 0;
}

function detectCollisions() {
   var textY = 30, bbox, mtv;

   if (shapeMoving) {
      shapes.forEach( function (shape) {
         if (shape !== shapeMoving) {
            mtv = shapeMoving.collidesWith(shape);

            if (collisionDetected(mtv)) {
               if (!stuck)
                  stick(mtv);
            }
         }
```

```
        });

        bbox = shapeMoving.boundingBox();
        if (bbox.left + bbox.width > canvas.width || bbox.left < 0) {
           velocity.x = -velocity.x;
        }
        if (bbox.top + bbox.height > canvas.height || bbox.top < 0) {
           velocity.y = -velocity.y;
        }
     }
};

// Event handlers.....................................................

canvas.onmousedown = function (e) {
    var location = windowToCanvas (e);

    if (showInstructions)
       showInstructions = false;

    velocity.x = lastVelocity.x;
    velocity.y = lastVelocity.y;

   shapeMoving = undefined;
   stuck = false;

   shapes.forEach( function (shape) {
      if (shape.isPointInPath(context, location.x, location.y))  {
         shapeMoving = shape;
      }
   });
};

// Animation..........................................................

function animate(time) {
   var elapsedTime, deltaX;

   if (lastTime === 0) {
      if (time !== undefined)
         lastTime = time;

      window.requestNextAnimationFrame(animate);
      return;
   }

   context.clearRect(0, 0, canvas.width, canvas.height);

   if (shapeMoving !== undefined) {
      elapsedTime = parseFloat(time - lastTime) / 1000;
      shapeMoving.move(velocity.x * elapsedTime,
                       velocity.y * elapsedTime);
   }

   detectCollisions();
   drawShapes();
   lastTime = time;

   if (showInstructions) {
      context.fillStyle = 'cornflowerblue';
      context.font = '24px Arial';
```

```
        context.fillText('Click on a shape to animate it', 20, 40);
    }
    window.requestNextAnimationFrame(animate);
};

// Initialization.....................................................

for (var i=0; i < polygonPoints.length; ++i) {
    var polygon = new Polygon(),
        points = polygonPoints[i];

    polygon.strokeStyle = polygonStrokeStyles[i];
    polygon.fillStyle = polygonFillStyles[i];

    points.forEach( function  (point) {
        polygon.addPoint(point.x, point.y);
    });

    shapes.push(polygon);
}

c1.fillStyle = 'rgba(200, 50, 50, 0.5)';

shapes.push(c1);
shapes.push(c2);

context.shadowColor = 'rgba(100,140,255,0.5)';
context.shadowBlur = 4;
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.font = '38px Arial';

window.requestNextAnimationFrame(animate);
```

8.4.2.3 利用最小平移向量实现反弹效果

图 8-25 所示应用程序含有数个图形，在用户点击某个图形之后，该图形就开始运动。如果它碰到了 canvas 边界或是其他图形，那么就会弹回，并继续运动下去。

要让某个图形被另外一个图形弹回来，我们得根据碰撞边界的法向量对反弹前的速度做反射，以求出反弹后的速度，如图 8-26 所示。

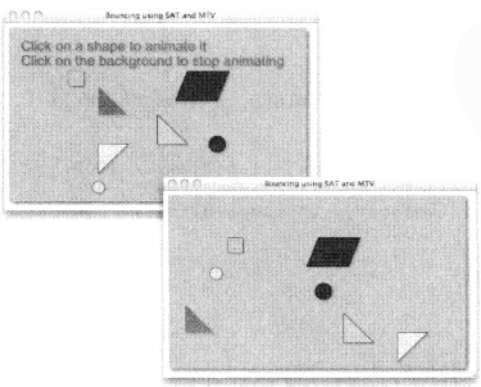


图 8-25 利用最小平移向量实现反弹效果

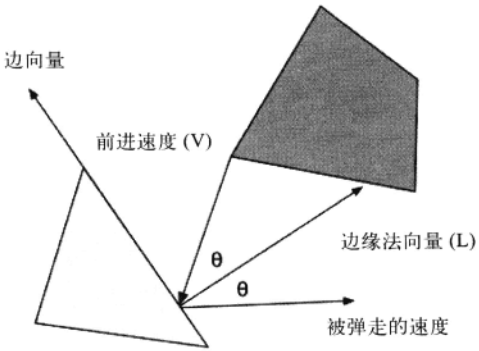


图 8-26　求速度向量的反弹向量

为了将一个图形从另一个图形的某条边上弹回，可以利用等式 8.4，求出某个向量关于某条轴的反射向量来。在本例这种情况下，等式中的向量是反射前的速度向量，而那条轴则是边缘法向量，或是代表碰撞边界的边缘向量。

$$\theta_{反弹后}=2\times(V\cdot L)/(L\cdot L)L-V$$

等式 8.4　求向量 V 对于向量 L 的反射向量

图 8-25 中的应用程序与 8.4.2.2 小节中的那个程序有很多共同点，有鉴于此，程序清单 8-20 仅仅列出了与两图形反弹效果有关的那部分代码，其中 bounce() 方法实现了等式 8.4 所列的公式。

程序清单 8-20　利用最小平移向量实现反弹效果

```
function detectCollisions() {
   if (shapeMoving) {
      handleShapeCollisions();
      handleEdgeCollisions();
   }
};

function handleShapeCollisions() {
   var mtv;

   shapes.forEach( function (shape) {
      if (shape !== shapeMoving) {
          mtv = shapeMoving.collidesWith(shape);
          if (collisionDetected(mtv)) {
             bounce(mtv, shapeMoving, shape);
          }
      }
   });
}

function collisionDetected(mtv) {
   return mtv.axis != undefined || mtv.overlap !== 0;
}

function separate(mtv) {
   var dx, dy, velocityMagnitude, point;

   if (mtv.axis === undefined) {
```

```
        point = new Point();
        velocityMagnitude = Math.sqrt(Math.pow(velocity.x, 2) +
                                      Math.pow(velocity.y, 2));

        point.x = velocity.x / velocityMagnitude;
        point.y = velocity.y / velocityMagnitude;

        mtv.axis = new Vector(point);
    }

    dy = mtv.axis.y * mtv.overlap;
    dx = mtv.axis.x * mtv.overlap

        if((dx < 0 && velocity.x < 0) ||
           (dx > 0 && velocity.x > 0)) {
          dx = -dx;
        }

        if ((dy < 0 && velocity.y < 0) ||
            (dy > 0 && velocity.y > 0)) {
           dy = -dy;
        }

        shapeMoving.move(dx, dy);
}

function checkMTVAxisDirection(mtv, collider, collidee) {
    var centroid1, centroid2, centroidVector, centroidUnitVector;

    if (mtv.axis === undefined)
        return;

    centroid1 = new Vector(collider.centroid()),
    centroid2 = new Vector(collidee.centroid()),
    centroidVector = centroid2.subtract(centroid1),
    centroidUnitVector = (new Vector(centroidVector)).normalize();

    if (centroidUnitVector.dotProduct(mtv.axis) > 0) {
        mtv.axis.x = -mtv.axis.x;
        mtv.axis.y = -mtv.axis.y;
    }
};

function bounce(mtv, collider, collidee) {
    var dotProductRatio, vdotl, ldotl, point,
         velocityVector = new Vector(new Point(velocity.x, velocity.y)),
         velocityUnitVector = velocityVector.normalize(),
         velocityVectorMagnitude = velocityVector.getMagnitude(),
         perpendicular;

    if (shapeMoving) {
        checkMTVAxisDirection(mtv, collider, collidee)

        point = new Point();

        if (mtv.axis !== undefined) {
            perpendicular = mtv.axis.perpendicular();
        }
        else {
            perpendicular = new Vector(new Point(-velocityUnitVector.y,
                                                 velocityUnitVector.x));
```

```
        }

        vdotl = velocityUnitVector.dotProduct(perpendicular);
        ldotl = perpendicular.dotProduct(perpendicular);
        dotProductRatio = vdotl / ldotl;

        point.x = 2 * dotProductRatio * perpendicular.x -
                  velocityUnitVector.x;

        point.y = 2 * dotProductRatio * perpendicular.y -
                  velocityUnitVector.y;

        separate(mtv);

        velocity.x = point.x * velocityVectorMagnitude;
        velocity.y = point.y * velocityVectorMagnitude;
    }
}
```

8.5 总结

碰撞检测是一个范围深广的主题，实际上整本书都在讨论它。在本章中，读者学到了一些比较容易实现的检测碰撞方法，包括外接矩形判别法、外接圆形判别法以及光线投射法。

然而，这一章的大部分内容都在讲述如何运用分离轴定理以及与之相关的最小平移向量。将这两者结合起来，可以实现出一种优质的碰撞检测算法，该算法几乎能满足所有对检测碰撞的需求。

在下一章中，我们会将本章所学的许多知识，与前面各章中讲解的内容结合起来，以实现一款基于 Canvas 的游戏。

第9章
游戏开发

在可以用电脑实现出来的东西中，游戏开发可以算是最有趣的事情之一啦，不过这并不等于说它很简单。你需要掌握诸如代数、三角函数、矢量运算等基本的数学知识，而且还得处理一些相当复杂的问题，比如实现动画及检测碰撞。不过话说回来，对于软件开发者来讲，能把自己对游戏的构想展现在屏幕上，这比其他东西所带来的满足感要大得多。

所幸本书前面数章已经讲解了制作游戏所要用到的数学运算、动画制作与碰撞检测等技术。处理完了这些难题之后，现在是时候来享受游戏开发的乐趣了。

本章内容可分为三节：

- 游戏引擎
- 游戏原型
- 弹珠台游戏

9.1 节讲了一个简单的游戏引擎，它大约有 450 行 JavaScript 代码。该引擎提供了一些制作游戏所用的基本功能，例如基于时间的运动效果、游戏暂停以及高分榜，等等。程序清单 9-9 将会列出这个引擎的全部代码。

9.2 节将制作一个极简的游戏原型[⊖]，它包含了游戏中需要实现的各种基础功能，但却并不包括具体的游戏逻辑。读者可以把它理解为游戏领域的“Hello, World”程序[⊜]。

9.3 节将会制作一个精美的弹珠台游戏，它要用到游戏引擎以及前面数章中所讲的很多技术。

9.1 游戏引擎

本章所讲的游戏引擎支持下列功能：

- 实现游戏循环：start()
- 绘制精灵：addSprite()、getSprite()
- 支持基于时间的运动：pixelsPerFrame()
- 调用回调方法：startAnimate()、paintUnderSprites()、paintOverSprites()、endAnimate()
- 暂停游戏：togglePaused()
- 处理按键：addKeyListener()
- 同时播放多个声音：canPlaySound()、playSound()

⊖ 原文为ungame，意为“不是游戏的游戏”，也就是一套不含游戏本身的逻辑，但却提供了其基本组件的演示框架。——译者注

⊜ “Hello, World”程序是指在计算机屏幕上输出“Hello, World!”（意为“世界，你好！”）这行字符串的程序。一般来说，这是每一种编程语言中最基本、最简单的程序，也经常是初学者所编写的第一个程序。它还可以用来确定该语言的编译器、程序开发环境，以及运行环境是否已经安装妥当。详情参见：http://zh.wikipedia.org/zh-cn/Hello_World。——译者注

- 记录帧速率：fps
- 记录游戏时间：gameTime
- 维护高分榜：setHighScore()、getHighScores()、clearHighScores()

上述功能列表也提到了 GameEngine 对象中与每个功能相对应的属性及方法。比如说，我们可以用 addSprite() 方法将精灵对象加入游戏引擎，也可以用 getSprite() 方法获取一个指向该精灵对象的引用。

从本质上讲，游戏引擎是通过 5.1.3 小节所讲的 window.requestNextAnimationFrame() 方法来实现游戏循环的，而此方法间接实现了 window.requestAnimationFrame() 方法的功能。游戏引擎在游戏循环中提供了一些注入功能代码所用的回调函数。可以注入功能代码的时机包括：每帧动画开始播放之前、游戏引擎绘制精灵之前和之后，以及动画播放完毕后。

引擎中有个简单的方法，叫做 pixelsPerFrame()，如果告诉它某物体每秒钟移动的像素数，那么它就可以返回该物体在当前这帧动画中移动了多少个像素。

开发者还可以查看当前的帧速率以及游戏时间。这里的游戏时间是指游戏程序运行的总时间减去暂停的时间。togglePaused() 方法可以用来暂停及恢复游戏。

引擎还支持高分榜、按键处理、声音播放等基本功能。

程序清单 9-1 列出了利用游戏引擎来制作游戏所需的基本流程：先创建游戏引擎对象以及精灵对象，然后将精灵加入引擎中，并且实现播放动画效果所用的回调方法，最后启动游戏引擎。

程序清单 9-1　利用游戏引擎来制作游戏所需的基本流程

```
// Create the game
var game = new Game('nameOfYourGame', 'canvasElementId'),

     // Create some sprites, and add them to the
     // game with game.addSprite()

     s1 = new Sprite(...),
     s2 = new Sprite(...); // s2 will be drawn on top of s1

game.addSprite(s1);
game.addSprite(s2);

// Implement animation callbacks

game.paintUnderSprites = function () {
   drawBackground(); // Implement this
    // Paint under sprites...
};

game.paintOverSprites = function () {
   // Paint over sprites...
};

game.startAnimate = function () {
   // Things to do at the beginning of the animation frame
};

game.endAnimate = function () {
   // Things to do at the end of the animation frame
};

// Start the game
game.start();
```

创建 Game 对象时，需要指定游戏名称以及游戏所在 canvas 元素的标识符。在将高分榜保存至本地存储空间时，游戏引擎会用到刚才提供的游戏名称。

在下面数小节中，我们将详述引擎的各项功能。

9.1.1 游戏循环

游戏循环遵循如下步骤：

（1）如果游戏暂停了，那么就跳过以下各步骤，并在 100 毫秒后再次执行游戏循环。

（2）更新帧速率。

（3）设置游戏时间。

（4）清除屏幕内容。

（5）在播放动画前，调用名为 startAnimate() 的回调方法。

（6）绘制精灵背后的内容。

（7）更新精灵。

（8）绘制精灵。

（9）绘制精灵前方的内容。

（10）在动画播放完毕后，调用名为 endAnimate() 的回调方法。

（11）请求浏览器播放下一帧动画。

游戏引擎的 togglePaused() 方法可以暂停并继续游戏。当引擎启动时，游戏并不处于暂停状态，所以第一次调用 togglePaused() 方法时，游戏将会暂停，再度调用时游戏又会继续运行。

如果游戏暂停了，那么游戏循环就不再做其他事情了，它只会请求浏览器在 100 毫秒之后安排该循环再度执行。许多电子游戏的帧速率是每秒 60 帧，也就是两帧之间延迟 16 毫秒。而该引擎此时所用的时间间隔却比 16 毫秒长很多，这意味着游戏引擎在暂停状态的 CPU 使用率会比平时低。

游戏中诸如运动等很多因素都受帧速率的影响，所以，如果游戏没有暂停，那么游戏循环首先要做的就是更新帧速率及游戏时间。然后，循环会清除屏幕内容，以便开始绘制下一帧动画。

在将屏幕清除干净后，引擎会调用名为 startAnimate() 及 paintUnderSprites() 的回调方法。前者负责每帧动画播放前的准备工作，比方说，许多游戏都会在 startAnimate() 方法中检测碰撞。paintUnderSprites() 方法主要用于绘制背景，有时也用它来绘制游戏场景中的部分内容。

绘制完精灵背后的内容，游戏循环将画出所有可见的精灵对象，然后，还会调用名为 paintOverSprites() 的回调方法，开发者可于此处绘制叠放在精灵前方的内容。

最后，游戏循环将调用名为 endAnimation() 的回调方法，并且使用 5.1.3 小节中名为 window.requestNextAnimationFrame() 的“Polyfill”式方法来请求浏览器绘制下一帧动画。

程序清单 9-2 列出了实现上述步骤所用的代码。

程序清单 9-2　游戏循环

```
var Game = function (gameName, canvasId) {
   var canvas = document.getElementById(canvasId),
       self = this; // Used by key event handlers below

   // General

   this.context = canvas.getContext('2d');
   this.sprites = [];
   ...
```

```
    // Time

    this.startTime = 0;
    this.lastTime = 0;
    this.gameTime = 0;
    this.fps = 0;
    this.STARTING_FPS = 60;

    this.paused = false;
    this.startedPauseAt = 0;
    this.PAUSE_TIMEOUT = 100;
    ...

    return this;
};

// Game methods.....................................................

Game.prototype = {
    ...

   // Game loop.....................................................

   start: function () {
      var self = this;                 // The this variable is the game
      this.startTime = getTimeNow();   // Record game's startTime

      Starts the animation

      window.requestNextAnimationFrame(
         function (time) {
            // The this variable in this function is the window.

            self.animate.call(self, time); // self is the game
         });
      },

       // Drives the game's animation. This method is called by the
       // browser when it's time for the next animation frame.

       animate: function (time) {
          var self = this;

          if (this.paused) {
             // In PAUSE_TIMEOUT (100) ms, call this method again to see
             // if the game is still paused. There's no need to check
             // more frequently.

             setTimeout( function () {
                self.animate.call(self, time);
             }, this.PAUSE_TIMEOUT);
          }

          else {                        // Game is not paused
             this.tick(time);           // Update fps, game time
             this.clearScreen();        // Prepare for next frame

             this.startAnimate(time);   // Override as you wish
             this.paintUnderSprites();  // Override as you wish

             this.updateSprites(time); // Invoke sprite behaviors
```

```
        this.paintSprites(time);  // Paint sprites in the canvas

        this.paintOverSprites();  // Override as you wish
        this.endAnimate();        // Override as you wish

        // Call this method again when it's time for
        // the next animation frame

        window.requestNextAnimationFrame(
           function (time) {
              self.animate.call(self, time);
           });
     }
  },

  // Update the frame rate, game time, and the last time the
  // application drew an animation frame.

  tick: function (time) {
     this.updateFrameRate(time);
     this.gameTime = (getTimeNow()) - this.startTime;
     this.lastTime = time;
  },

// Update the frame rate, based on the amount of time it took
// for the last animation frame only.

updateFrameRate: function (time) {
   if (this.lastTime === 0) this.fps = this.STARTING_FPS;
   else                     this.fps = 1000 / (time - this.lastTime);
},

// Clear the entire canvas.

clearScreen: function () {
   this.context.clearRect(0, 0,
      this.context.canvas.width, this.context.canvas.height);
},

// Update all sprites. The sprite update() method invokes all
// of a sprite's behaviors.

updateSprites: function (time) {
   for(var i=0; i < this.sprites.length; ++i) {
      var sprite = this.sprites[i];
      sprite.update(this.context, time);
   };
},

// Paint all visible sprites.

paintSprites: function (time) {
  for(var i=0; i < this.sprites.length; ++i) {
     var sprite = this.sprites[i];
     if (sprite.visible)
        sprite.paint(this.context);
  };
},
...
// Override the following methods as desired. animate() calls
```

```
    // the methods in the order they are listed.
    startAnimate:      function (time) { },
    paintUnderSprites: function ()     { },
    paintOverSprites:  function ()     { },
    endAnimate:        function ()     { }
};
```

我们可以像这样来创建 Game 对象并启动游戏：

```
var game = new Game('gameName', 'canvasId');
...
game.start();
```

调用了游戏引擎对象的 start() 方法后，该方法代码中的 this 变量就是 game 对象自身，这正是我们想要的。

然而，当通过 requestNextAnimationFrame() 函数向浏览器请求播放下一帧动画时，在 start() 方法所提交的那个函数内部，this 变量却是指向 window 对象的。如果 start() 方法在使用 requestNextAnimationFrame() 函数向浏览器提交请求时，用的是 this 变量，也就是以 this.animate(time) 的形式来编码的话，那么执行代码的时候可能会在 window 对象上调用并不存在的 animate() 方法。

表 9-1　游戏引擎对象中与游戏循环有关的方法

方法	描述
start()	设置游戏启动时间，并请求浏览器绘制下一帧动画，以此开始游戏
animate(time)	实现游戏循环
tick(time)	在播放每帧动画前更新帧速率及游戏时间
updateFrameRate(time)	更新游戏当前的帧速率
cleanScreen()	使用 context.clearRect() 方法来清除屏幕
updateSprites(time)	更新所有精灵对象
paintSprites(time)	绘制所有可见的精灵
startAnimate()	游戏引擎在播放每帧动画前都会调用此回调方法。默认情况下它不做任何事情，留待开发者在其中实现游戏逻辑
paintUnderSprites(time)	游戏引擎在绘制精灵前会调用此回调方法。默认情况下它不做任何事情，留待开发者在其中实现游戏逻辑
paintOverSprites(time)	游戏引擎绘制完精灵后会调用此回调方法。默认情况下它不做任何事情，留待开发者在其中实现游戏逻辑
endAnimate()	游戏引擎绘制完当前动画帧之后，回调用此回调方法。默认情况下它不做任何事情，留待开发者在其中实现游戏逻辑

游戏引擎的 start() 方法向 requestNextAnimationFrame() 传递了一个函数，并利用 JavaScript 语言内建的 call() 方法[⊖]来确保该函数中的 this 变量引用的是 game 对象而非 window 对象。一开始执行 start() 方法时，this 变量是指向 game 对象的，所以 start() 方法将它保存在名为 self 的变量中。稍后调用 call() 时，start() 方法会将先前保存的 self 变量传进去。

引擎的 animate() 方法实现了本节开头所述的 11 个步骤。该方法在稍后调用 requestNextAnimationFrame() 方法时会使用与 start() 方法一样的手段，来保证调用时所引用的是 game 对象

⊖ call()方法的用法请参考：https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/Function/call。——译者注

而非 window 对象。

表 9-1 列出了游戏引擎对象中与游戏循环有关的方法。

9.1.1.1 暂停

游戏引擎对象中有一个名为 paused 的属性，用来指示游戏当前是否处于暂停状态。如果游戏暂停了，那么引擎就不再执行游戏循环了，所以如果游戏引擎的 paused 属性是 true，那么游戏中什么事情都不会发生。

程序清单 9-3 演示了如何暂停并恢复游戏。

如果游戏暂停了，那么 animate() 方法会调用 setTimeout()，请求浏览器在大约 100 毫秒后再次调用 animate() 方法。因为游戏处于暂停状态的情况不会频繁出现，所以此时我们不需借助 requestNextAnimationFrame()，直接使用较为简便的 setTimeout() 方法即可。

使用 togglePaused() 方法暂停游戏时，该方法会记录下当前时间，在稍后恢复游戏时，需要据此来调整游戏运行的总时长。

在用 togglePaused() 将游戏从暂停状态恢复时，该方法会把游戏暂停的时长从游戏开始时间中减去。这样的话，游戏就会很精确地从原来暂停处继续执行下去，而不会出现有可能非常明显的时间跳跃现象。注意，此时游戏引擎对象的 startTime 属性也许不能表示游戏启动的实际时间了，因为该属性在游戏从暂停状态恢复时已经被调整过了。如果基于某种原因，你需要知道游戏启动的精确时间，那么可以自己记录这个值。

程序清单 9-3　在暂停状态与运行状态间切换

```
var Game = function (gameName, canvasId) {
   var canvas = document.getElementById(canvasId),
       self = this; // Used by key event handlers below
   ...
   this.startTime = 0;
   this.lastTime = 0;

   this.paused = false;
   this.startedPauseAt = 0;
   this.PAUSE_TIMEOUT = 100;
   ...
   return this;
};

// Game methods.......................................................

Game.prototype = {

   start: function () {
      this.startTime = getTimeNow(); // Record game's startTime
      ...
      window.requestNextAnimationFrame(
         function (time) {
            self.animate.call(self, time); // self is the game
         });
   },

   animate: function (time) {
      var self = this;

      if (this.paused) {
          // After PAUSE_TIMEOUT (100) ms, call this method again
```

```
            // to see if the game is still paused. There's no need to
            // check more frequently.

            setTimeout( function () {
               self.animate.call(self, time); // self is the game
            }, this.PAUSE_TIMEOUT); // PAUSE_TIMEOUT is 100 ms
         }
         else { // Game is not paused
            // Paint the next animation frame
            ...
            window.requestNextAnimationFrame(
               function (time) {
                  self.animate.call(self, time);
               });
         }
      },
      togglePaused: function () {
         var now = getTimeNow();

         this.paused = !this.paused;

         if (this.paused) {
            this.startedPauseAt = now;
         }
         else { // Not paused
            // Adjust start time, so game starts where it left off when
            // the user paused it.

            this.startTime = this.startTime + now - this.startedPauseAt;
            this.lastTime = now;
         }
      },
   };
```

在了解到游戏引擎是如何通过控制时间来暂停并恢复游戏之后，咱们看看怎样获知某物体在当前动画帧中所应移动的距离。

9.1.1.2 基于时间的运动

5.6 节已经讲了实现基于时间的运动所带来的好处。游戏引擎通过一个简单但又重要的方法来实现基于时间的运动，它叫做 pixelPerFrame()，其代码如程序清单 9-4 所示。

程序清单 9-4 游戏引擎实现基于时间的运动所用的方法

```
   pixelsPerFrame: function (time, velocity) {
      // This method returns the amount of pixels an object should move
      // for the current animation frame, given the current time and
      // the object's velocity. Velocity is measured in pixels/second.
      //
      // Note: (pixels/second) * (second/frame) = pixels/second:

      return velocity / game.fps;
   },
```

pixelsPerFrame() 方法以当前时间与物体移动速度作为参数，速度的单位是每秒钟移动的像素数。该方法的返回值表示在目前速度下，物体于当前帧中所应移动的像素数。

一般来说，在引擎对象的 startAnimate() 或 endAnimate() 回调方法中，以及精灵对象的 update() 方法中，都有可能会用到 pixelsPerFrame() 方法。

9.1.2 加载图像

许多游戏都要使用大量的图像，而且大多数游戏都会在启动时加载它们。图像的加载要耗费一定时间，所以说游戏最好能在加载图像时将相关信息显示给用户。

开发者可以用本章所讲的游戏引擎载入多张图像，并且能够获知任意时刻引擎所加载的图像数。程序清单 9-5 列出了游戏引擎中与图像加载有关的代码。下列三个方法可用于加载图像与追踪图像加载的进度：

- queueImage(imageUrl)：将图像放入加载队列中。
- loadImages()：开发者需要持续调用该方法，直到其返回 100 为止（方法的返回值表示图像加载完成百分比）。
- getImage(imageUrl)：返回图像对象。只有在 loadImages() 返回 100 之后，才可以调用该方法。

可以先用 queueImage() 方法把需要加载的图片排入队列，然后持续调用 loadImages() 方法，直到其返回 0 为止。该方法的返回值也可以用来更新游戏界面，以告诉玩家当前的图片加载进度，这段代码如下所示：

```
var game = new Game('gameName', 'canvasId');
...

game.queueImage('images/image1.png');
game.queueImage('images/image2.png');
...

interval = setInterval( function (e) {
   loadingPercentComplete = game.loadImages();

   if (loadingPercentComplete === 100) {
      clearInterval(interval);

       // Done loading images, update user interface accordingly
   }
   progressbar.draw(loadingPercentComplete);
}, 16);
```

要知道，有些图像也许会加载失败。我们可以通过 imagesFailedToLoad 属性来确认是不是所有图像都加载好了，该属性表示未完成加载的图像数。然而，不论是否有无法加载的图像，loadImages() 在处理完全部图像之后都会返回 100（加载完成百分比）。

程序清单 9-5　加载图像

```
var getTimeNow = function () {
   return +new Date();
};

var Game = function (gameName, canvasId) {
   var canvas = document.getElementById(canvasId),
   ...

   // Image loading

   this.imageLoadingProgressCallback;
   this.images = {};
   this.imageUrls = [];
   this.imagesLoaded = 0;
   this.imagesFailedToLoad = 0;
   this.imagesIndex = 0;
```

```
    ...

    return this;
};

// Game methods.....................................................

Game.prototype = {
    // Given a URL, return the associated image

    getImage: function (imageUrl) {
        return this.images[imageUrl];
    },

    // This method is called by loadImage() when
    // an image loads successfully.

    imageLoadedCallback: function (e) {
        this.imagesLoaded++;
    },

    // This method is called by loadImage() when
    // an image does not load successfully.

    imageLoadErrorCallback: function (e) {
        this.imagesFailedToLoad++;
    },

    // Loads a particular image

    loadImage: function (imageUrl) {
        var image = new Image(),
            self = this;

        image.src = imageUrl;

        image.addEventListener('load',
            function (e) {
                self.imageLoadedCallback(e);
            });

        image.addEventListener('error',
            function (e) {
                self.imageLoadErrorCallback(e);
            });

        this.images[imageUrl] = image;
    },

    // You call this method repeatedly to load images that have been
    // queued (by calling queueImage()). This method returns the
    // percent of the game's images that have been processed. When
    // the method returns 100, all images are loaded, and you can
    // quit calling this method.

    loadImages: function () {

        // If there are images left to load

        if (this.imagesIndex < this.imageUrls.length) {
            this.loadImage(this.imageUrls[this.imagesIndex]);
```

```
            this.imagesIndex++;
        }

        // Return the percent complete

        return (this.imagesLoaded + this.imagesFailedToLoad) /
               this.imageUrls.length * 100;
    },

    // Call this method to add an image to the queue. The image
    // will be loaded by loadImages().

    queueImage: function (imageUrl) {
        this.imageUrls.push(imageUrl);
    },
    ...
};
```

每次调用游戏引擎对象的 queueImage() 方法时，传入的图像 URL 地址就会被加入一个数组中。其后每次调用 loadImages() 时，该方法会加载此数组中的下一幅图像，并且返回目前的总加载进度。

请注意，loadImage() 方法也使用了与游戏引擎的 start() 及 animate() 方法相同的技巧，以确保在注册用于处理图像加载完毕及图像加载错误事件的回调方法时，这些方法引用的是游戏引擎对象而非 window 对象。

9.1.3 同时播放多个声音

游戏中通常需要同时播放多个声音，比如，游戏在播放背景音乐的同时，可能还要播放音效。如程序清单 9-6 所示，我们的游戏引擎也支持同时播放多个声音。

具有 canPlay...() 形式的数个方法可用于查询浏览器是否能够播放某种特定格式的声音文件，而 playSound() 方法则用来播放声音。

Game 对象的构造器创建了 10 个 Audio 对象，并将其加入一个数组中。调用 playSound() 方法时，游戏引擎会找出第一个未被占用的声道，并用它来播放指定的声音。

调用 playSound() 方法时，需要用相应的 audio 元素标识符作为其参数。此方法会在第一个未被占用的声道上播放 audio 元素中的音源。

程序清单 9-6　引擎对声音播放的支持

```
var Game = function (gameName, canvasId) {
    ...

    this.soundOn = true;
    this.soundChannels = [];
    this.audio = new Audio();
    this.NUM_SOUND_CHANNELS = 10;

    for (var i=0; i < this.NUM_SOUND_CHANNELS; ++i) {
        var audio = new Audio();
        this.soundChannels.push(audio);
    }

    ...

    return this;
```

```
};

Game.prototype = {

   canPlayOggVorbis: function () {
      return "" != this.audio.canPlayType('audio/ogg; codecs="vorbis"');
   },

   canPlayMp4: function () {
      return "" != this.audio.canPlayType('audio/mp4');
   },

   getAvailableSoundChannel: function () {
      var audio;

      for (var i=0; i < this.NUM_SOUND_CHANNELS; ++i) {
         audio = this.soundChannels[i];
         if (audio.played && audio.played.length > 0) {
            if (audio.ended)
                return audio;
         }
         else {
            if (!audio.ended)
               return audio;
         }
      }
      return undefined; // All tracks in use
   },

   playSound: function (id) {
      var track = this.getAvailableSoundChannel(),
          element = document.getElementById(id);

      if (track && element) {
         track.src = element.src === '' ?
                     element.currentSrc : element.src;
         track.load();
         track.play();
      }
   },
};
```

9.1.4 键盘事件

许多游戏都需要玩家通过键盘进行操作。游戏引擎提供了注册按键事件监听器的功能，如程序清单 9-7 所示。

addKeyListener() 方法可以用来向游戏中注册按键监听器。传递给该方法的参数对象必须具有名为 key 及 listener 的属性。这两个属性分别表示需要监听的按键以及处理按键事件所用的回调方法。比如，当玩家按下了正在被监听的按键时，游戏引擎会调用开发者所注册的回调方法。

程序清单 9-7　引擎对注册按键监听器以及按类型过滤按键事件的支持

```
var Game = function (gameName, canvasId) {
   var canvas = document.getElementById(canvasId);
   ...
   this.keyListeners = [];
   ...
};
```

```
Game.prototype = {

   // Key listeners.................................................

   addKeyListener: function (keyAndListener) {
      game.keyListeners.push(keyAndListener);
   },

   findKeyListener: function (key) {
      var listener = undefined;

      game.keyListeners.forEach(function (keyAndListener) {
         var currentKey = keyAndListener.key;
         if (currentKey === key) {
            listener = keyAndListener.listener;
         }
      });
      return listener;
    },

    keyPressed: function (e) {
       var listener = undefined,
           key = undefined;

       switch (e.keyCode) {
          // Add more keys as needed
          case 32: key = 'space';       break;
          case 83: key = 's';           break;
          case 80: key = 'p';           break;
          case 37: key = 'left arrow';  break;
          case 39: key = 'right arrow'; break;
          case 38: key = 'up arrow';    break;
          case 40: key = 'down arrow';  break;
       }

       listener = game.findKeyListener(key);
       if (listener) { // Listener is a function
          listener();  // Invoke the listener function
       }
     },
  };
```

在默认情况下，游戏引擎可以监听程序清单 9-7 中所列的 space（空格键）、s 等按键。要加入对其他按键的支持也很简单，以下网址列出了 JavaScript 所用的键值码：http://bit.ly/tvU2NS。

9.1.5 高分榜

游戏引擎将高分榜数组以 JSON 格式[⊖]存放在本地存储空间中，如程序清单 9-8 所示。

程序清单 9-8 游戏引擎对高分榜功能的支持

```
var Game = function (gameName, canvasId) {
   var canvas = document.getElementById(canvasId);
   ...
   this.HIGH_SCORES_SUFFIX = '_highscores';
   ...
};
```

⊖ JavaScript Object Notation，简称JSON，意为JavaScript对象表示法，是一种以文字为基础且易于阅读的轻量级数据交换格式，详情参见：http://zh.wikipedia.org/zh-cn/JSON。——译者注

```
Game.prototype = {

   // High scores.....................................................

   getHighScores: function () {
      var key = game.gameName + game.HIGH_SCORES_SUFFIX,
          highScoresString = localStorage[key];

      if (highScoresString == undefined) {
         localStorage[key] = JSON.stringify([]);
      }
      return JSON.parse(localStorage[key]);
   },

   setHighScore: function (highScore) {
      var key = game.gameName + game.HIGH_SCORES_SUFFIX,
          highScoresString = localStorage[key];

      highScores.unshift(highScore);
      localStorage[key] = JSON.stringify(highScores);
   },

   clearHighScores: function () {
      localStorage[game.gameName + game.HIGH_SCORES_SUFFIX] =
         JSON.stringify([]);
   },
};
```

getHighScores() 方法将游戏名称加上“_highscores”后缀，并以此字符串作为键值，将游戏高分榜存放在本地存储空间中。

setHighScore() 方法将高分榜数据从本地存储空间载入一个列表中，然后将当前最高分置于列表顶部，最后把列表存回本地存储空间。

clearHighScores() 方法可以将本地存储空间中存放的高分榜数组清空。

表 9-2 总结了程序清单 9-8 中所列的方法。

表 9-2　游戏引擎对象中与高分榜有关的方法

方法	描述
setHighScore(highScore)	将 highScore 参数所指的高分加入本地存储空间里的高分列表中
getHighScores()	返回本地存储空间中保存的游戏高分列表
clearHighScores()	清空本地存储空间中的游戏高分数据

9.1.6　游戏引擎源代码

程序清单 9-9 列出了游戏引擎的源代码。

程序清单 9-9　游戏引擎的源代码（gameEngine.js）

```
var getTimeNow = function () {
   return +new Date();
};

// Game.............................................................

// This game engine implements a game loop that draws sprites.
//
```

```
// The game engine also has support for:
//
// Time-based motion (game.pixelsPerFrame())
// Pause (game.togglePaused())
// High scores (game.[get][clear]HighScores(), game.setHighScore())
// Sound (game.canPlaySound(), game.playSound())
// Accessing frame rate (game.fps)
// Accessing game time (game.gameTime)
// Key processing (game.addKeyListener())
//

// The game engine's animate() method invokes the following methods,
// in the order listed:
//
//    game.startAnimate()
//    game.paintUnderSprites()
//    game.paintOverSprites()
//    game.endAnimate()
//
// Those four methods are implemented by the game engine to do nothing.
// You override those methods to make the game come alive.

var Game = function (gameName, canvasId) {
  var canvas = document.getElementById(canvasId),
      self = this; // Used by key event handlers below

  // General

  this.context = canvas.getContext('2d');
  this.gameName = gameName;
  this.sprites = [];
  this.keyListeners = [];

  // High scores

  this.HIGH_SCORES_SUFFIX = '_highscores';

  // Image loading

  this.imageLoadingProgressCallback;
  this.images = {};
  this.imageUrls = [];
  this.imagesLoaded = 0;
  this.imagesFailedToLoad = 0;
  this.imagesIndex = 0;

  // Time

  this.startTime = 0;
  this.lastTime = 0;
  this.gameTime = 0;
  this.fps = 0;
  this.STARTING_FPS = 60;

  this.paused = false;
  this.startedPauseAt = 0;
  this.PAUSE_TIMEOUT = 100;

  // Sound

  this.soundOn = true;
  this.soundChannels = [];
  this.audio = new Audio();
  this.NUM_SOUND_CHANNELS = 10;

  for (var i=0; i < this.NUM_SOUND_CHANNELS; ++i) {
```

```
      var audio = new Audio();
      this.soundChannels.push(audio);
   }

   // The this object in the following event handlers is the
   // DOM window, which is why the functions call
   // self.keyPressed() instead of this.keyPressed(e).

   window.onkeypress = function (e) { self.keyPressed(e)  };
   window.onkeydown  = function (e) { self.keyPressed(e); };

   return this;
};

// Game methods.....................................................

Game.prototype = {
   // Given a URL, return the associated image

   getImage: function (imageUrl) {
      return this.images[imageUrl];
   },

   // This method is called by loadImage() when
   // an image loads successfully.

   imageLoadedCallback: function (e) {
      this.imagesLoaded++;
   },

   // This method is called by loadImage() when
   // an image does not load successfully.

   imageLoadErrorCallback: function (e) {
      this.imagesFailedToLoad++;
   },

   // Loads a particular image

   loadImage: function (imageUrl) {
      var image = new Image(),
          self = this;

      image.src = imageUrl;

      image.addEventListener('load',
         function (e) {
            self.imageLoadedCallback(e);
         });

      image.addEventListener('error',
         function (e) {
            self.imageLoadErrorCallback(e);
         });

      this.images[imageUrl] = image;
   },

   // You call this method repeatedly to load images that have been
   // queued (by calling queueImage()). This method returns the
   // percent of the game's images that have been processed. When
   // the method returns 100, all images are loaded, and you can
   // quit calling this method.

   loadImages: function () {
```

```
    // If there are images left to load

    if (this.imagesIndex < this.imageUrls.length) {
       this.loadImage(this.imageUrls[this.imagesIndex]);
       this.imagesIndex++;
    }

    // Return the percent complete

    return (this.imagesLoaded + this.imagesFailedToLoad) /
            this.imageUrls.length * 100;
 },

 // Call this method to add an image to the queue. The image
 // will be loaded by loadImages().

 queueImage: function (imageUrl) {
    this.imageUrls.push(imageUrl);
 },

 // Game loop.......................................................

 // Starts the animation by calling window.requestNextAnimationFrame().
 //
 // window.requestNextAnimationFrame() is a polyfill method
 // implemented in requestNextAnimationFrame.js. You pass
 // requestNextAnimationFrame() a reference to a function
 // that the browser calls when it's time to draw the next
 // animation frame.
 //
 // When it's time to draw the next animation frame, the
 // browser invokes the function that you pass to
 // requestNextAnimationFrame(). Because that function is
 // invoked by the browser (the window object, to be more exact),
 // the this variable in that function will be the window object.
 // We want the this variable to be the game instead, so we use
 // JavaScript's built-in call() function to call the function,
 // with the game specified as the this variable.

 start: function () {
    var self = this;                // The this variable is the game
    this.startTime = getTimeNow(); // Record game's startTime

    window.requestNextAnimationFrame(
       function (time) {
          // The this variable in this function is the window,
          // not the game, which is why we do not simply
          // do this: animate.call(time).

          self.animate.call(self, time); // self is the game
       });
    },

    // Drives the game's animation. This method is called by the
    // browser when it's time for the next animation frame.
    //
    // If the game is paused, animate() reschedules another call to
    // animate() in PAUSE_TIMEOUT (100) ms.
    //
    // If the game is not paused, animate() paints the next animation
    // frame and reschedules another call to animate() when it's time
    // to draw the next animation frame.
    //
    // The implementations of this.startAnimate(),
    // this.paintUnderSprites(), this.paintOverSprites(), and
```

```
// this.endAnimate() do nothing. You override those methods to
// create the animation frame.

animate: function (time) {
   var self = this;

   if (this.paused) {
      // In PAUSE_TIMEOUT (100) ms, call this method again to see
      // if the game is still paused. There's no need to check
      // more frequently.

      setTimeout( function () {
         self.animate.call(self, time);
      }, this.PAUSE_TIMEOUT);
   }
   else {                        // Game is not paused
      this.tick(time);           // Update fps, game time
      this.clearScreen();        // Prepare for the next frame

      this.startAnimate(time);   // Override as you wish
      this.paintUnderSprites(); // Override as you wish

      this.updateSprites(time); // Invoke sprite behaviors
      this.paintSprites(time);  // Paint sprites in the canvas

      this.paintOverSprites();  // Override as you wish
      this.endAnimate();        // Override as you wish

      // Keep the animation going.

      window.requestNextAnimationFrame(
         function (time) {
            self.animate.call(self, time);
         });
   }
},

// Update the frame rate, game time, and the last time the
// application drew an animation frame.

tick: function (time) {
   this.updateFrameRate(time);
   this.gameTime = (getTimeNow()) - this.startTime;
   this.lastTime = time;
},

// Update the frame rate, based on the amount of time it took
// for the last animation frame only.

updateFrameRate: function (time) {
   if (this.lastTime === 0) this.fps = this.STARTING_FPS;
   else                     this.fps = 1000 / (time - this.lastTime);
},

// Clear the entire canvas.

clearScreen: function () {
   this.context.clearRect(0, 0,
      this.context.canvas.width, this.context.canvas.height);
},

// Update all sprites. The sprite update() method invokes all
// of a sprite's behaviors.

updateSprites: function (time) {
```

```
        for(var i=0; i < this.sprites.length; ++i) {
           var sprite = this.sprites[i];
           sprite.update(this.context, time);
        };
     },

     // Paint all visible sprites.

     paintSprites: function (time) {
        for(var i=0; i < this.sprites.length; ++i) {
           var sprite = this.sprites[i];
           if (sprite.visible)
              sprite.paint(this.context);
        };
     },

     // Toggle the paused state of the game. If, after
     // toggling, the paused state is unpaused, the
     // application subtracts the time spent during
     // the pause from the game's start time. That
     // means the game picks up where it left off,
     // without a potentially large jump in time.

     togglePaused: function () {
        var now = getTimeNow();

        this.paused = !this.paused;

        if (this.paused) {
           this.startedPauseAt = now;
        }

      else { // Not paused
         // Adjust start time, so game starts where it left off when
         // the user paused it.

        this.startTime = this.startTime + now - this.startedPauseAt;
        this.lastTime = now;
     }
  },

  // Given a velocity of some object, calculate the number of pixels
  // to move that object for the current frame.

  pixelsPerFrame: function (time, velocity) {
      // Sprites move a certain amount of pixels per frame
      // (pixels/frame). This methods returns the amount of
      // pixels a sprite should move for a given frame. Sprite
      // velocity is measured in pixels/second,
      // so: (pixels/second) * (second/frame) = pixels/frame:

      return velocity / this.fps; // pixels/frame
  },

  // High scores.................................................

  // Returns an array of high scores from local storage.

  getHighScores: function () {
     var key = this.gameName + this.HIGH_SCORES_SUFFIX,
         highScoresString = localStorage[key];

     if (highScoresString == undefined) {
        localStorage[key] = JSON.stringify([]);
     }
     return JSON.parse(localStorage[key]);
```

```
},

// Sets the high score in local storage.

setHighScore: function (highScore) {
   var key = this.gameName + this.HIGH_SCORES_SUFFIX,
       highScoresString = localStorage[key];

   highScores.unshift(highScore);
   localStorage[key] = JSON.stringify(highScores);
},

   // Removes the high scores from local storage.

   clearHighScores: function () {
      localStorage[this.gameName + this.HIGH_SCORES_SUFFIX] =
      JSON.stringify([]);
   },

   // Key listeners.................................................

   // Add a (key, listener) pair to the keyListeners array.

   addKeyListener: function (keyAndListener) {
      this.keyListeners.push(keyAndListener);
   },

   // Given a key, return the associated listener.

   findKeyListener: function (key) {
      var listener = undefined;

      for(var i=0; i < this.keyListeners.length; ++i) {
         var keyAndListener = this.keyListeners[i],
             currentKey = keyAndListener.key;
         if (currentKey === key) {
            listener = keyAndListener.listener;
         }
      };
      return listener;
   },

   // This method is the callback for key down and key press events.

   keyPressed: function (e) {
      var listener = undefined,
          key = undefined;

      switch (e.keyCode) {
         // Add more keys as needed

         case 32: key = 'space';          break;
         case 68: key = 'd';              break;
         case 75: key = 'k';              break;
         case 83: key = 's';              break;
         case 80: key = 'p';              break;
         case 37: key = 'left arrow';     break;
         case 39: key = 'right arrow';    break;
         case 38: key = 'up arrow';       break;
         case 40: key = 'down arrow';     break;
      }
      listener = this.findKeyListener(key);
      if (listener) { // Listener is a function
         listener();  // Invoke the listener function
      }
```

```
    },

    // Sound.......................................................

    // Returns true if the browser can play sounds in ogg file format.

    canPlayOggVorbis: function () {
       return "" != this.audio.canPlayType('audio/ogg; codecs="vorbis"');
    },

    // Returns true if the browser can play sounds in mp3 file format.

    canPlayMp3: function () {
       return "" != this.audio.canPlayType('audio/mpeg');
    },

    // Returns the first sound available channel.

    getAvailableSoundChannel: function () {
       var audio;

       for (var i=0; i < this.NUM_SOUND_CHANNELS; ++i) {
          audio = this.soundChannels[i];
          if (audio.played && audio.played.length > 0) {
             if (audio.ended)
                return audio;
          }
          else {
             if (!audio.ended)
                return audio;
          }
       }
       return undefined; // All channels in use
    },

    // Given an identifier, play the associated sound.

playSound: function (id) {
    var channel = this.getAvailableSoundChannel(),
        element = document.getElementById(id);

    if (channel && element) {
       channel.src = element.src === '' ?
                     element.currentSrc : element.src;
       channel.load();
       channel.play();
    }
},

// Sprites.....................................................

// Add a sprite to the game. The game engine will update the sprite
// and paint it (if it's visible) in the animate() method.

addSprite: function (sprite) {
    this.sprites.push(sprite);
},

// It's probably a good idea not to access sprites directly,
// because it's better to write generalized code that deals with
// all sprites, so this method should be used sparingly.

getSprite: function (name) {
    for(i in this.sprites) {
       if (this.sprites[i].name === name)
```

```
            return this.sprites[i];
        }
        return null;
    },

    // The following methods, which do nothing, are called by animate()
    // in the order they are listed. Override them as you wish.

    startAnimate:       function (time) { },
    paintUnderSprites:  function ()     { },
    paintOverSprites:   function ()     { },
    endAnimate:         function ()     { }
};
```

看过了引擎的实现方式之后，我们来讲讲如何用它制作游戏。

9.2 游戏原型

本节将通过游戏原型来演示本章开头所述的游戏引擎的用法，该原型如图 9-1 所示。

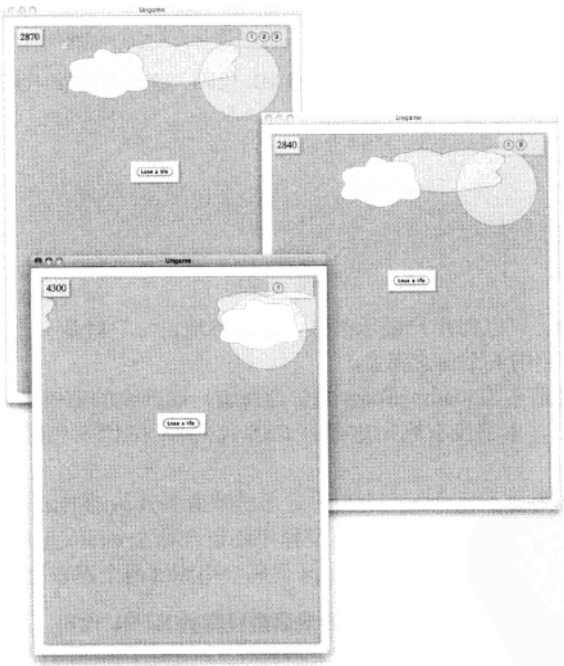


图 9-1 运行游戏原型

“不死族”[⊖]的生物并未真正死亡，同理，我们这个“不是游戏的游戏原型”（ungame）实际上也算不得真正的游戏。它并不是为了玩，而是想要演示如何用游戏引擎来制作游戏。

⊖ undead，又称不死生物、亡灵族、死灵族，是指肉体已经死亡却还能活动的怪物，通常被认为是遗留人间的魂魄和具备自我意识的尸体。它们以不同型态出现在各地文化的传说和小说里，常见于奇幻小说、恐怖小说和角色扮演游戏。详情参见：http://zh.wikipedia.org/zh-cn/不死生物。——译者注

程序开始运行后，背景就会持续地滚动。如果点击“Lose a life”按钮，玩家就会失去一条生命。玩家一开始有三条生命，游戏画面右上方的信息面板中也会显示与当前生命数相对应的图标。

这个游戏原型程序包含了大多数游戏所共有的特性，如下所列：

- 资源加载画面
- 游戏资源管理
- 声音播放
- 具有视差效果的滚动背景
- 生命数量显示
- 高分榜
- 按键处理
- 暂停功能与自动暂停机制
- 对游戏结束的流程处理

我们首先来看看这个游戏原型程序的 HTML 代码。

9.2.1 游戏原型程序的 HTML 代码

程序清单 9-10 列出了这个游戏原型程序的 HTML 代码。这段 HTML 代码定义了如下 DIV 元素：

- loadingToast
- scoreToast
- pausedToast
- gameOverToast
- highScoreToast
- loseLifeToast

“信息弹窗”（toast）可以向用户展示消息，更通俗地说，它就是一个对话框。上文已经列出了这个游戏原型程序所用的 6 个信息弹窗。

游戏启动时，只显示名为 loadingToast 的信息弹窗，而其余的则被游戏原型程序的 CSS 隐藏起来了（为了简洁起见，本书没有列出这部分 CSS 代码），在 CSS 代码中，它们的 display 属性都被设置为 none。

游戏原型程序用了两个 canvas，一个用于显示游戏背景及滚动的云彩，另一个用于在游戏画面右上方显示玩家当前的生命数。该游戏原型程序还包含两个 audio 元素，浏览器在启动游戏的时候就会将其加载。游戏原型程序的 JavaScript 代码会用到这两个声音元素。

程序清单 9-10　游戏原型程序的 HTML 代码

```
<!DOCTYPE html>
<html>
   <head>
      <title>Ungame</title>
      <link rel="stylesheet" type="text/css" href="ungame.css"/>
   </head>

   <body>
      <!-- Game canvas.............................................. -->

      <canvas id="gameCanvas" width="550" height="750">
         Canvas not supported
```

```
</canvas>

<!-- Loading Toast........................................... -->

<div id='loadingToast' class='toast'>
   <span id='loadingToastTitle' class='title'>The Ungame</span>

   <span id='loadingToastBlurb' class='blurb'>
      <p>This game is an ungame, sort of like the undead:
         The undead are not really dead, and this is not really
         a game; however, it implements essential functionality
         pertient to most games.</p>

      <p>The ungame comes with:</p>

      <ul>
         <li>This loading screen</li>
         <li>Asset management</li>
         <li>Music and Sounds</li>
         <li>A scrolling background with parallax</li>
         <li>Lives indicator (upper right corner)</li>
         <li>Score indicator (appears when the ungame starts)</li>
         <li>High score functionality</li>
         <li>Key processing (including throttling)</li>
         <li>Pause (press 'p' key once the ungame starts)</li>
         <li>Auto-Pause (when the window loses focus)</li>
      </ul>

      <p>The ungame is implemented with a
         simple game engine (~200 lines of JavaScript).</p>
   </span>

   <span id='loadButtonSpan'>
      <input type='button' id='loadButton' value='Load Game...'
             autofocus='true'/>
      <span id='loadingMessage'>Loading...</span>
   </span>

   <div id='progressDiv'></div>
</div>

<!-- Scores................................................ -->

<div id='scoreToast' class='toast'></div>

<!-- Lives................................................. -->

<canvas id='livesCanvas' width='90' height='40'>
   Canvas not supported
</canvas>

<!-- Paused................................................ -->

<div id='pausedToast' class='toast'>
  <p class='title' style='margin-left: 45px;'>Paused</p>
  <p>Click anywhere to start</p>
</div>

<!-- Game Over............................................. -->

<div id='gameOverToast' class='toast'>
```

```
        <p class='title'>Game Over</p><br/>
        <p><input id='clearHighScoresCheckbox' type='checkbox'/>
           clear high scores</p>
        <input id='newGameButton' type='button' value='new game'
              autofocus='true'/>
    </div>

    <!-- High scores.............................................. -->

    <p id='highScoreParagraph'></p>

    <div id='highScoreToast' width='400' style='display: none'>
      <p class='title'>High score!</p>

      <p>What's your name?</p>

      <input id='nameInput' type='text' autofocus='true'>
      <input id='addMyScoreButton' type='button' value='add my score'
             disabled='true'>

       <input id='newGameFromHighScoresButton' type='button'
              value='new game'>

       <p class='title' id='previousHighScoresTitle' display='none'>
         Previous High Scores
       </p>

       <!-- The following ordered list is populated
                      by JavaScript in ungame.js -->
       <ol id='highScoreList'></ol>
     </div>

     <!-- Lose Life............................................... -->

     <div id='loseLifeToast' class='toast'>
        <input id='loseLifeButton' type='button' value='Lose a life'
               autofocus='true'/>
     </div>

     <!-- Sounds................................................... -->

     <audio id='pop' preload='auto'>
        <source src='sounds/pop.ogg' type='audio/ogg'>
        <source src='sounds/pop.mp3' type='audio/mp3'>
     </audio>

     <audio id='whoosh' preload='auto'>
        <source src='sounds/whoosh.ogg' type='audio/ogg'>
        <source src='sounds/whoosh.mp3' type='audio/mp3'>
     </audio>

    <script src = 'requestNextAnimationFrame.js'></script>
    <script src = 'progressbar.js'></script>
    <script src = 'gameEngine.js'></script>
    <script src = 'ungame.js'></script>
  </body>
</html>
```

接下来，我们看看这个原型程序的游戏循环。

9.2.2 原型程序的游戏循环

游戏原型程序创建了一个Game对象，并且重新实现了它的paintUnderSprites()与paintOverSprites()方法。虽说该原型程序并没有精灵对象，不过游戏引擎依然会调用paintUnderSprites()与paintOverSprites()方法。程序清单9-11列出了这两个方法的代码。

程序清单9-11 paintUnderSprites()与paintOverSprites()方法

```
var game = new Game('ungame', 'gameCanvas'),
    ...

game.paintOverSprites = function () {
   paintNearCloud(game.context, 120, 20);
   paintNearCloud(game.context,game.context.canvas.width+120, 20);
};

game.paintUnderSprites = function () {

   // Background erased by game engine's clearScreen()

   if (!gameOver && livesLeft === 0) {
      over();
   }
   else {
      paintSun(game.context);
      paintFarCloud(game.context, 20, 20);
      paintFarCloud(game.context,game.context.canvas.width+20, 20);

      if (!gameOver) {
         updateScore();
      }

      updateLivesDisplay();
   }
};
...
game.start();
```

paintUnderSprites()方法绘制太阳及远处稍大的那团云彩，此时如果游戏尚未结束的话，那么它还会更新分数及玩家生命数的显示信息。paintOverSprites()方法绘制近处稍小的那团云彩。这两个方法都会用固定的坐标值将云彩绘制两遍。正如程序清单9-12所示，游戏原型程序平移绘图环境对象的坐标系，使云彩看起来正在从左向右移动。

游戏原型程序在绘制每帧动画时，都会调用scrollBackground()函数，该函数会将绘图环境对象的坐标系平移一小段距离。如果平移量达到或超过了canvas的宽度，那么scrollBackground()函数就会将其重新调整为小于canvas宽度的值[⊖]，这样看上去背景就好像一直在滚动似的。

程序清单9-12 背景的滚动

```
var game = new Game('ungame', 'gameCanvas'),
    ...
// Scrolling the background.........................................
```

⊖ 此操作是通过求余数运算，即%运算符来实现的。请参见程序清单9-12中的这行代码："translateOffset = (translateOffset + translateDelta) %game.canvas.width;"。——译者注

```
    translateDelta = 0.025,
    translateOffset = 0,

    scrollBackground = function () {
        translateOffset = (translateOffset + translateDelta) %
                          game.canvas.width;
        game.context.translate(-translateOffset, 0);
    },

    // Paint Methods.....................................................

    paintClouds = function (context) {
        paintFarCloud(game.context, 0, 20);
        paintNearCloud(game.context, game.context.canvas.width + 120, 20);
    },

    paintSun = function (context) {
        ...
    },

    paintFarCloud = function (context, x, y) {
        context.save();
        scrollBackground();

        // Paint far cloud with quadratic curves...

        context.restore();
    },

    paintNearCloud = function (context, x, y) {
        context.save();
        scrollBackground();
        scrollBackground();

        // Paint near cloud with quadratic curves...

        context.restore();
    },
```

提示：游戏原型程序中的视差动画

游戏原型的 scrollBackground() 函数被 paintFarCloud() 方法调用了 1 次，而被 paintNearCloud() 方法调用了 2 次。这样的话，近处云朵的移动速度就是远处的 2 倍，从而产生了一种和缓的视差效果。本书 5.8 节详述了视差动画效果的制作方式。

9.2.3 游戏原型程序的加载画面

游戏原型启动之后，会显示如图 9-2 左上方截图所示的加载画面，其中含有对该原型程序的简短描述，还有一个加载游戏所用的按钮。用户点击此按钮后，原型程序会用一个进度条取代按钮，并开始载入游戏所用的资源。

程序清单 9-13 列出了处理“Load Game”按钮的 onclick 事件所用的代码。

游戏原型程序本身并没有使用任何图像资源，太阳和云朵都是用代码直接绘制出来的。不过，为了演示资源的加载，该程序载入了 12 张图像，并使用进度条来表示加载进度。9.1.2 小节曾详述了图像的加载，本书 10.2 节则会讲述进度条的制作。

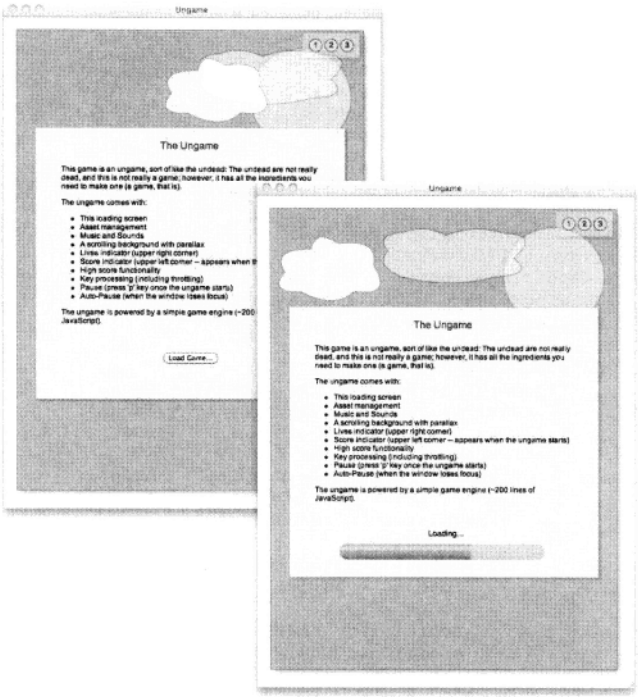


图 9-2　加载画面

游戏引擎加载完图像之后，程序清单 9-13 中的 onclick 事件处理器将会通过 window.setTimeout() 函数将加载画面逐段隐藏。首先让进度条消失，其次是简介文字，最后将显示加载信息所用的弹窗也隐藏起来。

程序清单 9-13　加载画面

```
loadButton.onclick = function(e) {
    var interval,
        loadingPercentComplete = 0;

    e.preventDefault();

    progressDiv.style.display = 'block';
    loadButton.style.display = 'none';

    loadingMessage.style.display = 'block';
    progressDiv.appendChild(progressbar.domElement);

    // The following images are not used. The ungame loads
    // to illustrate loading images at the beginning of a game.

    game.queueImage('images/image1.png');
    game.queueImage('images/image2.png');
    game.queueImage('images/image3.png');
    game.queueImage('images/image4.png');
    game.queueImage('images/image5.png');
```

```
    game.queueImage('images/image6.png');
    game.queueImage('images/image7.png');
    game.queueImage('images/image8.png');
    game.queueImage('images/image9.png');
    game.queueImage('images/image10.png');
    game.queueImage('images/image11.png');
    game.queueImage('images/image12.png');

    interval = setInterval(function(e) {
        loadingPercentComplete = game.loadImages();

        if (loadingPercentComplete === 100) {
            clearInterval(interval);
            setTimeout(function(e) {
                loadingMessage.style.display = 'none';
                progressDiv.style.display = 'none';

                setTimeout(function(e) {
                    loadingToastBlurb.style.display = 'none';
                    loadingToastTitle.style.display = 'none';

                    setTimeout(function(e) {
                        loadingToast.style.display = 'none';
                        loseLifeToast.style.display = 'block';
                        game.playSound('sounds/pop');

                        setTimeout(function(e) {
                            loading = false;
                            score = 10;
                            scoreToast.innerText = '10';
                            scoreToast.style.display = 'inline';
                            game.playSound('pop');
                        }, 1000);
                    }, 500);
                }, 500);
            }, 500);
        }
        progressbar.draw(loadingPercentComplete);
    }, 16);
};
// Start game.........................................................
game.start();
```

9.2.4 暂停画面

9.1.1.1 小节讲述了如何使用游戏引擎来暂停并恢复游戏。如果玩家暂停了游戏，那么屏幕上会显示出一个相应的信息弹窗，如图 9-3 所示。

程序清单 9-14 列出了游戏原型程序所用的 togglePaused() 方法，它会调用游戏引擎对象中的同名方法，如果游戏被暂停了，那么还将显示一个信息弹窗，用以表示游戏已经暂停。用户单击这个弹窗即可继续游戏，此外也可以通过键盘上的 P 键来暂停及恢复游戏。9.2.5 小节将会详述游戏中的按键处理。

程序清单 9-14　暂停画面

```
var game = new Game('ungame', 'gameCanvas'),
...

pausedToast = document.getElementById('pausedToast'),
...
```

```
// Paused.............................................................

togglePaused = function () {
    game.togglePaused();
    pausedToast.style.display = game.paused ? 'inline' : 'none';
},

pausedToast.onclick = function (e) {
    pausedToast.style.display = 'none';
    togglePaused();
},
```

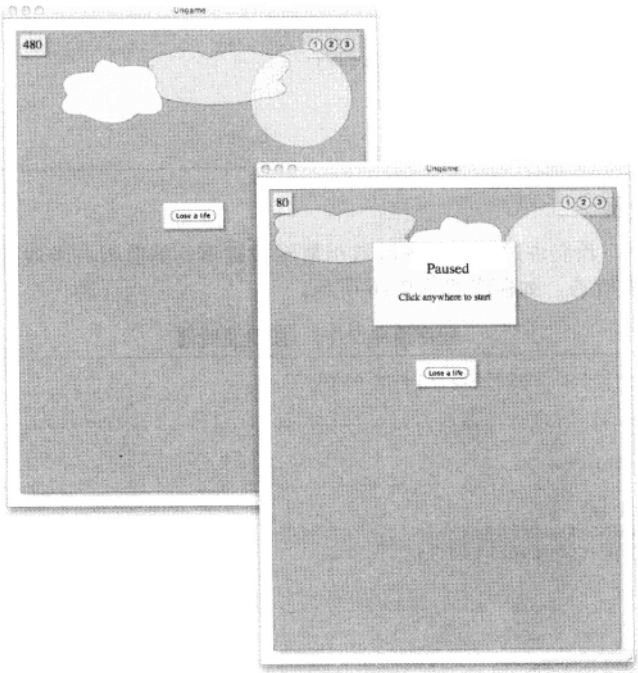


图 9-3 暂停画面

自动暂停

5.1.3 小节讲过，制作动画时应该使用 window.requestAnimationFrame() 方法。简单地说，这么做的原因是，浏览器比开发者更清楚何时应该绘制下一帧动画。

此外，如果用户打开了一个新的浏览器分页，或者将焦点移到了其他窗口中，那么 window.requestAnimationFrame() 就会极大地降低所绘动画的帧速率。浏览器通过限制帧速率，可以节省 CPU 资源及电池消耗。

然而，动画帧速率降低之后，可能会产生副作用：过低的帧速率会影响到碰撞检测算法。因此，当用户打开新的浏览器分页或切换到另一个窗口时，最好不要让浏览器自动降低动画的帧速率。

我们无法阻止浏览器自动降低动画的帧速率，不过我们可以修改自己所做的游戏，让它在当

前窗口失去焦点时自动暂停。当游戏所在窗口重新获得焦点后，我们再让游戏继续运行，或是提供某种机制让用户自行恢复游戏。

程序清单 9-15 列出了在游戏原型程序中实现自动暂停功能所用的代码。

程序清单 9-15　自动暂停功能的实现代码

```
var game = new Game('ungame', 'gameCanvas'),
...
window.onblur = function windowOnBlur() {
    if (!gameOver && !game.paused) {
        togglePaused();
    }
},

window.onfocus = function windowOnFocus() {
    if (game.paused) {
        togglePaused();
    }
},
```

9.2.5　按键监听器

在 9.1.4 小节中，我们讲过如何实现游戏引擎对按键事件的监听。游戏原型程序还实现了一个监听 P 键的按键监听器，如程序清单 9-16 所示。

程序清单 9-16　按键监听器

```
var game = new Game('ungame', 'gameCanvas'),
    ...
// Key listeners.....................................................

game.addKeyListener(
    {
        key: 'p',
        listener: function () {
            game.togglePaused();
        }
    }
);
...
```

当用户按下 P 键后，游戏原型程序会切换游戏的暂停状态。

我们已经知道了游戏原型程序是如何使用游戏引擎来控制游戏运行的，现在来看看怎么利用游戏引擎来处理游戏结束时的情况。

9.2.6　游戏结束及高分榜

几乎每个游戏都有某种记录最高得分的机制。如果玩家在游戏结束时创造了新的最高分，那么游戏将会显示这个分数，并给玩家提供一个保留高分记录的机会。

我们的游戏原型程序也有高分榜功能，图 9-4 左上方的截图展示了该程序的高分信息显示面板。

如果游戏结束时玩家没有达成最高分，那么显示画面则如图 9-4 底部的截图所示。与大部分游戏不同，这个原型程序可以让玩家在重新开始游戏之前先清空现有的高分榜。

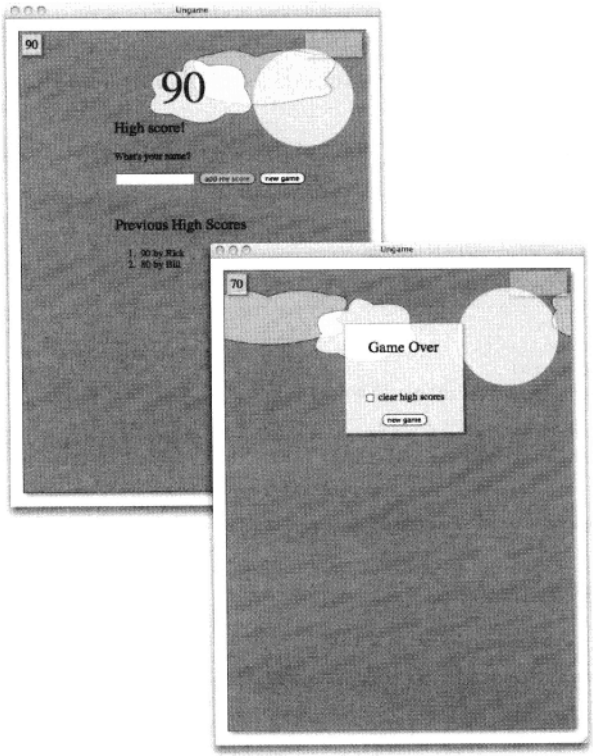


图 9-4 游戏原型程序的高分榜

图 9-5 中标出了与高分显示面板有关的 HTML 元素名称。程序清单 9-17 的代码使用与这些名称对应的 HTML 元素来实现游戏原型程序的高分榜功能。

请注意，程序清单 9-17 所列的代码都是与用户界面有关的，至于在本地存储空间中保存高分榜数据等底层任务则由游戏引擎来完成。我们在 9.1.5 小节中曾经讲过这部分内容。

程序清单 9-17 高分榜

```
var game = new Game('ungame', 'gameCanvas'),
    ...
    score = 0,
    lastScore = 0,
    lastScoreUpdate = undefined,

    // High score.....................................................

    HIGH_SCORES_DISPLAYED = 10,

    highScoreToast = document.getElementById('highScoreToast'),
    highScoreParagraph = document.getElementById('highScoreParagraph'),
    highScoreList = document.getElementById('highScoreList'),
    nameInput = document.getElementById('nameInput'),
    addMyScoreButton = document.getElementById('addMyScoreButton'),
    newGameButton = document.getElementById('newGameButton'),
```

```
previousHighScoresTitle =
    document.getElementById('previousHighScoresTitle'),

newGameFromHighScoresButton =
    document.getElementById('newGameFromHighScoresButton'),

clearHighScoresCheckbox =
    document.getElementById('clearHighScoresCheckbox'),

// Game over.....................................................

gameOverToast = document.getElementById('gameOverToast'),
gameOver = false,

// Game over.....................................................

over = function () {
    var highScore;
    highScores = game.getHighScores();

    if (highScores.length == 0 || score > highScores[0].score) {
        showHighScores();
    }
    else {
        gameOverToast.style.display = 'inline';
    }
    gameOver = true;
    lastScore = score;
    score = 0;
};

// High scores...................................................

// Change game display to show high scores when
// player bests the high score.

showHighScores = function () {
    highScoreParagraph.style.display = 'inline';
    highScoreParagraph.innerText = score;
    highScoreToast.style.display = 'inline';
    updateHighScoreList();
};

// The game shows the list of high scores in
// an ordered list. This method creates that
// list element and populates it with the
// current high scores.

updateHighScoreList = function () {
    var el,
        highScores = game.getHighScores(),
        length = highScores.length,
        highScore,
        listParent = highScoreList.parentNode;

    listParent.removeChild(highScoreList);
    highScoreList = document.createElement('ol');
    highScoreList.id = 'highScoreList'; // So CSS takes effect
    listParent.appendChild(highScoreList);

    if (length > 0) {
```

```
            previousHighScoresTitle.style.display = 'block';

            length = length > 10 ? 10 : length;

            for (var i=0; i < length; ++i) {

                highScore = highScores[i];
                el = document.createElement('li');
                el.innerText = highScore.score +
                               ' by ' + highScore.name;
                highScoreList.appendChild(el);
            }
        }
        else {
            previousHighScoresTitle.style.display = 'none';
        }
    }
```

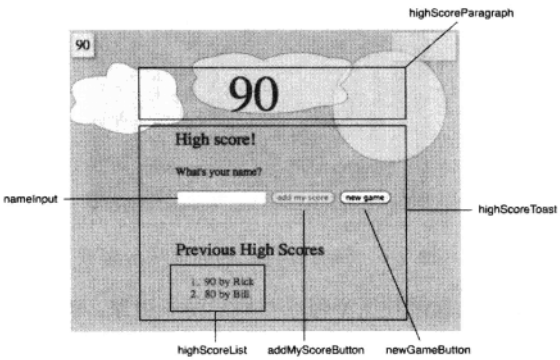


图 9-5　高分信息显示面板

读者已经学到了如何实现一个简单而功能丰富的游戏引擎，并且知道怎样用它做出一个极简的游戏原型程序，现在我们该来制作一款精美的游戏了。

9.3　弹珠台游戏

本章最后将实现如图 9-6 所示的弹珠台游戏，它用到了本章开头所讲的游戏引擎。

要实现这个弹珠台游戏，开发者需要应对下列挑战：

- 在考虑到现实中的重力及摩擦力等因素的情况下，对弹珠的运动方式建模。
- 实现弹珠台底部左右两个弹板的非线性运动。
- 检测碰撞。有时需要检测高速运动的弹珠 与其他物体的碰撞。
- 检测弹珠与弹板之间的碰撞，这两者可能会同时移动。
- 检测弹珠与弹珠台圆顶之间的碰撞，并在碰撞之后改变弹珠的移动方式。

除了前面这一串令人生畏的需求之外，我们还要考虑诸如资源加载、按键处理、高分榜等一般游戏都有的功能。好在 9.2 节中讲过的那个游戏原型程序已经实现了这些细节，我们可以基于该原型程序来制作弹珠台游戏，这样就能专注于实现那些与弹珠本身有关的业务逻辑了。

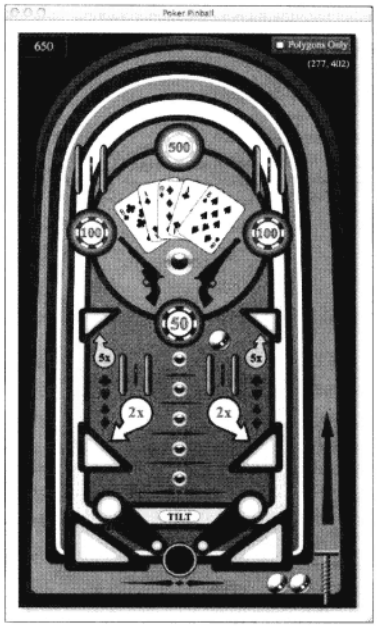


图 9-6 带有扑克元素的弹珠台游戏

弹珠台游戏的制作不仅要依靠游戏原型程序与底层游戏引擎，而且还会用到第 4 章至第 8 章中所讲的大量知识。

咱们先来看看弹珠台游戏的游戏循环吧。

提示：弹珠台游戏的源代码

弹珠台游戏的实现过程相当冗长，差不多有 1500 行代码，如果全部列出来的话，需要三十多页，所以本书略去了这部分源代码。接下来的数小节将讲述游戏实现过程中的几个重要问题，并且只会列出与其主题有关的代码。弹珠台游戏以及本书其他范例程序的完整源代码，都可以在 corehtml5canvas.com 网站中下载。

9.3.1 游戏循环弹珠

程序清单 9-18 列出了弹珠台游戏中的游戏循环所用的代码。

回想一下，我们在 9.1 节中介绍游戏引擎时曾说过，游戏引擎对象已经实现好了一个游戏循环，并在其中提供了 4 个可供注入功能代码的回调方法：

- startAnimate()
- paintUnderSprites()
- paintOverSprites()
- endAnimate()

程序清单 9-18　弹珠台游戏的游戏循环

```
var game = new Game('pinball', 'gameCanvas'),
    ... // Declarations omitted for brevity

game.startAnimate = function () {
    var collisionOccurred;

    if (loading || game.paused || launching)
        return;

    if (!gameOver && livesLeft === 0) {
        over();
        return;
    }
    if (ballOutOfPlay) {
        ballOutOfPlay = false;
        prepareForLaunch();
        brieflyShowTryAgainImage(2000);
        livesLeft--;
        return;
    }

    adjustRightFlipperCollisionPolygon();
    adjustLeftFlipperCollisionPolygon();

    collisionOccurred = detectCollisions();

    if (!collisionOccurred && applyGravityAndFriction) {
        applyFrictionAndGravity(); // Modifies ball velocity
    }
};

game.paintUnderSprites = function () {
    if (loading)
        return;

    updateLeftFlipper();
    updateRightFlipper();

    if (showPolygonsOnly) {
        drawCollisionShapes();
    }
    else {
        if (!showingHighScores) {
            game.context.drawImage(backgroundImage,0,0);

            drawLitBumper();
            if (showTryAgain) {
                brieflyShowTryAgainImage(2000); // Show image for 2 seconds
            }

            paintLeftFlipper();
            paintRightFlipper();

            for (var i=0; i < livesLeft-1; ++i) {
                drawExtraBall(i);
            }
        }
    }
};
```

弹珠台游戏实现了 startAnimate() 与 paintUnderSprites()，这两个方法在上段代码中的出现顺序正是它们被游戏引擎所调用的顺序。

游戏引擎在绘制新的动画帧之前，会调用 startAnimate() 方法。当游戏处于结束、加载、暂停或发射弹珠等状态时，该方法不做任何事情。如果游戏不在上述状态中，那么这个方法就检查弹珠是否处于死球[⊖]状态，并据此采取相应的措施。

然后，startAnimate() 方法调整左右两个弹板的碰撞多边形（如果它们正在运动的话），并且调用 detectCollisions() 方法来检测并应对碰撞。最后，要是没有碰撞发生，并且当前状况应该考虑重力及摩擦力因素的话（正在发射弹珠时是不用考虑这两个因素的），那么该方法就会根据这两个力修正弹珠的移动速度。9.3.3 小节与 9.3.6 小节分别详述了如何实现摩擦力与重力对弹珠运动方式的影响，以及如何检测弹珠与其他物体的碰撞。

弹珠游戏的 paintUnderSprites() 方法绘制背景以及其余未发射的弹珠，并且如果发现当前弹珠与某个挡板相撞，那么就将这个挡板点亮。

paintUnderSprites() 方法也会更新并绘制左右两个弹板。如果玩家正在操控它们的话，那么 updateLeftFlipper() 与 updateRightFlipper() 方法就会调整对应弹板的角度。

最后要注意的是，如果 showTryAgain 属性为 true，那么 startAnimate() 方法还会调用一个名为 brieflyShowTryAgainImage() 的方法。当弹珠处于图 9-7 所示的死球状态时，brieflyShow-TryAgainImage() 方法会显示一幅含有“Try Again”字样的图像，提示玩家用下一颗弹珠再玩一次。

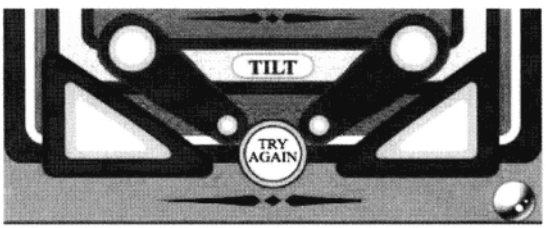


图 9-7 当弹珠落到弹珠台底部时，提示玩家再试一次

9.3.2 弹珠精灵

弹珠台游戏中只用到了两个精灵：弹珠以及发射弹珠的发射器。弹珠精灵的实现代码列在了程序清单 9-19 之中。

程序清单 9-19 弹珠精灵的代码

```
var game = new Game('pinball', 'gameCanvas'),
    ... // Declarations omitted for brevity

    lastBallPosition = new Point(),

    ballMover = {
        execute: function (sprite, context, time) {
            if (!game.paused && !loading) {
                lastBallPosition.x = sprite.left;
                lastBallPosition.y = sprite.top;

                if ( !launching && sprite.left < ACTUATOR_LEFT &&
                     (sprite.top > FLIPPER_BOTTOM || sprite.top < 0)) {
                    ballOutOfPlay = true;
                }
```

⊖ 原文是“out of play”，意为足球术语中的“死球”一词，详情参见：https://en.wikipedia.org/wiki/Ball_in_and_out_of_play，在本游戏中是指弹珠落入弹珠台底部，也就是位于有效游戏区域之外。——译者注

```
                sprite.left += game.pixelsPerFrame(time, sprite.velocityX);
                sprite.top += game.pixelsPerFrame(time, sprite.velocityY);
            }
        },
    },

    ballSprite = new Sprite('ball',
                            new ImagePainter('images/ball.png'),
                            [ ballMover ]),
    ...
```

表示弹珠的精灵对象是用 ImagePainter 来创建的，该绘制器对象负责将传给其构造器的 URL 所对应的图像画出来。本书第 6 章曾详述了精灵与图像绘制器对象的用法。

在弹珠对象中，最有趣的部分要数它的行为了。ball 的行为是由 ballMover 对象来实现的，该对象的 execute() 方法先将弹珠当前的位置记录到 lastBallPosition 之中，然后再移动它。在检测碰撞时，游戏会用刚才记录好的 lastBallPosition 作为位移向量。

需要注意的是，ballMover.execute() 方法会根据游戏引擎对象的 pixelsPerFrame() 方法来算出弹珠在 X 及 Y 方向上所移动的像素数。如果弹珠处于死球状态，那么 ballMover 对象就将 ballOutOfPlay 属性设为 true，这样的话，游戏引擎下一次调用 startAnimate() 方法时，就会把弹珠重新放在发射平台上了。

还有一点值得说明，那就是 ballMover 对象本身并不直接根据重力与摩擦力去修正弹珠的移动速度，而是将这项任务留给 applyFrictionAndGravity() 方法来完成。我们稍后就会讲到这个方法。

9.3.3 重力与摩擦力

回忆一下，在 9.3.1 小节中我们说过，游戏引擎在绘制每一帧动画之前，都要调用弹珠台游戏的 startAnimate() 方法，而该方法则通过 applyFrictionAndGravity() 来调整弹珠的移动速度。这段代码如下所示：

```
if (!collisionOccurred && applyGravityAndFriction) {
    applyFrictionAndGravity(parseFloat(time - game.lastTime));
}
```

startAnimate() 方法把当前帧与上一帧的时间差，以毫秒为单位，传给了程序清单 9-20 所列的 applyFrictionAndGravity() 方法。

程序清单 9-20　根据重力与摩擦力修正弹珠的移动速度

```
applyFrictionAndGravity = function (time) {
    var lastElapsedTime = time / 1000,
        gravityVelocityIncrease = GRAVITY * seconds * 0.5;

    if (Math.abs(ballSprite.velocityX) > MIN_BALL_VELOCITY) {
        ballSprite.velocityX *= Math.pow(0.2, lastElapsedTime);
    }

    ballSprite.velocityY += gravityVelocityIncrease *
        parseFloat(game.context.canvas.height / GAME_HEIGHT_IN_METERS);
},
```

根据给定的毫秒数，applyFrictionAndGravity() 方法就可以算出从播放上一帧动画到当前帧这段时间内，重力与摩擦力对弹珠速度所产生的影响。

为了表现摩擦力所带来的效果，每过一秒钟，applyFrictionAndGravity() 方法就将弹珠的速度

设为原来的 50%。速度降低的百分比是根据经验得来的，弹珠在桌面上滚动时，它的速度大致上就是按照这个值逐步减缓的。

applyFrictionAndGravity() 方法也会逐渐增加弹珠的垂直速度，以反映重力对它的影响。首先将当前帧距离上一帧的秒数乘以重力加速度常数（9.8m/s2），然后再将结果乘以 0.1，这样就算出了弹珠的垂直速度增量。由于弹珠台的摆放位置与水平面大致平行，倾斜角度很小，这样就极大地减少了重力对弹珠速度的影响，所以要将算出的结果乘以 0.1。

弹珠的移动实现起来还相对容易一些，不过弹板的移动可就是另外一回事了。接下来我们就讲讲这个问题。

9.3.4 弹板的移动

弹珠台的弹板是以非线性的方式移动的。当弹板升起时，它的速度会迅速增大，然而随着高度的增加，其速度则逐步衰减。在现实生活中，弹珠台的弹板不可能全速运动，当它到了最高位置之后，就会立刻停止。同理，在弹板下降的过程中，它的速度也会因重力的影响而逐渐增大。

读者也许看出来了，弹板在升起与落下时的这两种非线性的运动方式，正是 7.2 节中讲过的缓出运动（ease out）与缓入运动（ease in）。在那一章里，我们把相应的时间扭曲函数绑定到简单的动画计时器对象身上，以此来实现缓入与缓出运动。所以，如果打算用计时器对象来控制动画的运动（5.6 节中说过，开发者的确应该这么做），那么把时间扭曲函数绑定到动画计时器对象上，就可以实现出非线性的运动效果了。

程序清单 9-21 展示了弹珠台游戏如何用动画计时器对象来控制其左弹板的移动。应用程序创建了两个计时器对象，分别用于弹板的上升与下落。前者的持续时间为 25 毫秒，而后者则是 175 毫秒。这意味着弹板上升时的速度要比下落时快一些。

请注意程序清单 9-21 中设置弹板角度的那几行代码。应用程序会向计时器对象查询弹板持续运动的时间，有了这个值，我们就可以分别实现出弹板在上升与下落过程中所做的缓出与缓入运动效果了。

程序清单 9-21 还用到了左侧弹板旋转轴的位置及其偏移量，图 9-8 描述了这两个值所表示的意义。

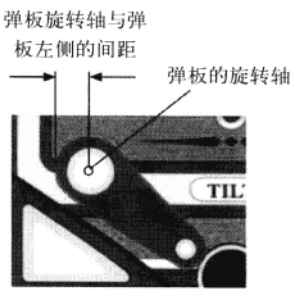


图 9-8 弹板的旋转轴及其偏移量

程序清单 9-21 弹板的运动

```
var game = new Game('pinball', 'gameCanvas'),
    ... // Some declarations omitted for brevity

    FLIPPER_RISE_DURATION = 25,      // Milliseconds
    FLIPPER_FALL_DURATION = 175,     // Milliseconds
    MAX_FLIPPER_ANGLE = Math.PI/4,   // 45 degrees
    ...

    leftFlipperRiseTimer =
        new AnimationTimer(FLIPPER_RISE_DURATION,
                           AnimationTimer.makeEaseOut(3)),
    leftFlipperFallTimer =
        new AnimationTimer(FLIPPER_FALL_DURATION,
                           AnimationTimer.makeEaseIn(3)),

    leftFlipperAngle = 0,
```

```
    ...

function updateLeftFlipper() {
    if (leftFlipperRiseTimer.isRunning()) {          // Flipper is rising
        if (leftFlipperRiseTimer.isOver()) {         // Finished rising
            leftFlipperRiseTimer.stop();             // Stop rise timer
            leftFlipperAngle = MAX_FLIPPER_ANGLE;    // Set flipper angle
            leftFlipperFallTimer.start();            // Start falling
        }
        else {                                       // Flipper is still rising
            leftFlipperAngle =
                MAX_FLIPPER_ANGLE / FLIPPER_RISE_DURATION *
                leftFlipperRiseTimer.getElapsedTime();
        }
    }
       else if (leftFlipperFallTimer.isRunning()) { // Flipper is falling
        if (leftFlipperFallTimer.isOver()) {         // Finished falling
            leftFlipperFallTimer.stop();             // Stop fall timer
            leftFlipperAngle = 0;                    // Set flipper angle
        }
       else {                                        // Flipper is still falling
            leftFlipperAngle = MAX_FLIPPER_ANGLE -
                MAX_FLIPPER_ANGLE / FLIPPER_FALL_DURATION *
                leftFlipperFallTimer.getElapsedTime();
        }
    }
};
function paintLeftFlipper() {
    if (leftFlipperRiseTimer.isRunning() ||
        leftFlipperFallTimer.isRunning()) {
        game.context.save();
        game.context.translate(LEFT_FLIPPER_PIVOT_X,
                               LEFT_FLIPPER_PIVOT_Y);

        game.context.rotate(-leftFlipperAngle);

                game.context.drawImage(game.getImage('images/leftFlipper.png'),
                               -LEFT_FLIPPER_PIVOT_OFFSET_X,
                               -LEFT_FLIPPER_PIVOT_OFFSET_Y);
        game.context.restore();
    }
    else {
        game.context.drawImage(game.getImage('images/leftFlipper.png'),
            LEFT_FLIPPER_PIVOT_X - LEFT_FLIPPER_PIVOT_OFFSET_X,
            LEFT_FLIPPER_PIVOT_Y - LEFT_FLIPPER_PIVOT_OFFSET_Y);
    }
};
```

9.3.5 处理键盘事件

弹珠台游戏需要处理如下键盘事件：

- 按下 K 键时，右侧弹板弹起，同时播放弹板弹起的音效。
- 按下 D 键时，左侧弹板弹起，同时播放弹板弹起的音效。
- P 键切换游戏的暂停状态。
- ↑键使发射器的弹簧弹起。
- ↓键压下发射器的弹簧。
- 空格键发射弹珠。

利用游戏引擎所提供的机制，我们可以按照程序清单 9-22 所述方式来实现弹珠台游戏的各个按键监听器。

程序清单 9-22 弹珠台游戏的按键监听器

```
var game = new Game('pinball', 'gameCanvas'),
    ...

lastKeyListenerTime = 0,    // For throttling

game.addKeyListener(
    {
        key: 'p',
        listener: function () {
            togglePaused();
        }
    }
);

game.addKeyListener(
    {
        key: 'k',
        listener: function () {
            if ( !launching && !gameOver) {
                rightFlipperAngle = 0;
                rightFlipperRiseTimer.start();
                game.playSound('flipper');
            }
        }
    }
);

game.addKeyListener(
    {
        key: 'd',
        listener: function () {
            if ( !launching && !gameOver) {
                leftFlipperAngle = 0;
                leftFlipperRiseTimer.start();
                game.playSound('flipper');
            }
        }
    }
);

game.addKeyListener(
    {
        key: 'up arrow',
        listener: function () {
            var now;

            if (!launching || launchStep === 1)
                return;

            now = +new Date();

            if (now - lastKeyListenerTime > 80) { // Throttle
                lastKeyListenerTime = now;

                launchStep--;

                ballSprite.top = BALL_LAUNCH_TOP + (launchStep-1) * 9;

                actuatorSprite.painter.image =
                    launchImages[launchStep-1];
```

```
                adjustActuatorPlatformShape();
            }
        }
    }
);

game.addKeyListener(
    {
        key: 'down arrow',
        listener: function () {
            var now;

            if (!launching || launchStep === LAUNCH_STEPS)
                return;

            now = +new Date();

            if (now - lastKeyListenerTime > 80) { // Throttle
                lastKeyListenerTime = now;
                launchStep++;
                actuatorSprite.painter.image = launchImages[launchStep-1];
                ballSprite.top = BALL_LAUNCH_TOP + (launchStep-1) * 9;
                adjustActuatorPlatformShape();
            }
        }
    }
);

game.addKeyListener(
    {
        key: 'space',
        listener: function () {
            if (!launching && ballSprite.left === BALL_LAUNCH_LEFT &&
                ballSprite.velocityY === 0) {
                launching = true;
                ballSprite.velocityY = 0;
                applyGravityAndFriction = false;
                launchStep = 1;
            }
            if (launching) {
                ballSprite.velocityY = -300 * launchStep;
                launching = false;
                launchStep = 1;

                setTimeout( function (e) {
                    actuatorSprite.painter.image = launchImages[0];
                    adjustActuatorPlatformShape();
                }, 50);

                setTimeout( function (e) {
                    applyGravityAndFriction = true;
                    adjustRightBoundaryAfterLaunch();
                }, 2000);
            }
        }
    }
);
```

P 键所对应的按键监听器，会调用弹珠台游戏的 togglePaused() 方法来切换游戏的暂停状态，而该方法又会调用游戏引擎对象的同名方法，并且显示一个表示游戏暂停的信息弹窗，如图 9-9 所示。

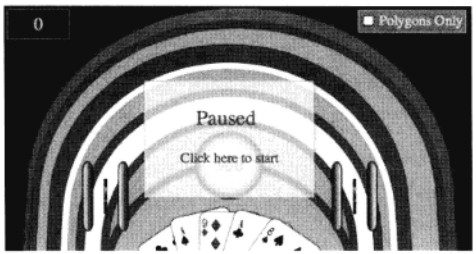


图 9-9 弹珠台游戏的暂停画面截图

用于控制左右弹板的 D 键与 K 键所对应的按键监听器对象，将弹板的角度设为 0，同时启动弹板上升动画所用的计时器对象，并播放“啷”的一声，用以表示弹板弹起时的音效。

↑与 ↓键分别用于抬升及下压弹珠发射器。弹珠台游戏会控制这两个按键事件的响应频率，每隔 80 毫秒才会处理一次，于是当玩家持续按下↑或 ↓键不松时，我们就能控制住弹珠发射器抬升或下压的速率了。

最后，如果检测到玩家按下了空格键，那么就将弹珠从发射器上弹出去。50 毫秒之后，该键的处理器对象会把重力与摩擦力的影响屏蔽掉，以便弹珠可以平滑地沿着弹珠台的右边界发射出去。两秒钟后，按键处理器又会恢复重力与摩擦力的影响。

9.3.6 碰撞检测

弹珠台游戏主要运用 8.4 节所述的分离轴定理（SAT）来实现事后碰撞检测。对于移动速度缓慢且面积相对大一些的多边形，该定理的检测效果很好，不过，对于面积小而且移动迅速的物体，其效果就不那么理想了。鉴于分离轴定理检测法有此缺陷，所以我们在判断弹珠与正在移动的弹板之间的碰撞时，会辅以光线投射法，以提高判断精度。

首先我们来看弹珠台游戏如何用分离轴定理来检测小球与弹板之外的其他物体之间的碰撞。

9.3.6.1 用分离轴定理检测碰撞

由于弹珠台游戏要使用分离轴定理来检测碰撞，因此我们要创建一些多边形，用它们来表示那些能与弹珠发生碰撞的物体，如图 9-10 所示。

如果选中了弹珠台游戏右上方的“Polygons Only”复选框，那么游戏就只会把碰撞检测所用的多边形以及信息显示面板绘制出来。

在这种情况下，只有碰撞检测所用的多边形才会被画出来，于是弹珠台游戏就变成一个功能演示程序了。对于玩家来说，只画出碰撞检测用的多边形，的确不怎么有趣，不过这对开发者来说却非常有用，因为可以由此看出碰撞发生时的具体情况。

在勾选了“Polygons Only”复选框后，弹珠台游戏使用如下方法来绘制检测碰撞所用的多边形：

```
function drawCollisionShapes() {
    var centroid;

    shapes.forEach( function (shape) {
        shape.stroke(game.context);
        game.context.beginPath();
        centroid = shape.centroid();
        game.context.arc(centroid.x, centroid.y, 1.5, 0,
                         Math.PI*2, false);
```

```
        game.context.stroke();
    });
}
```

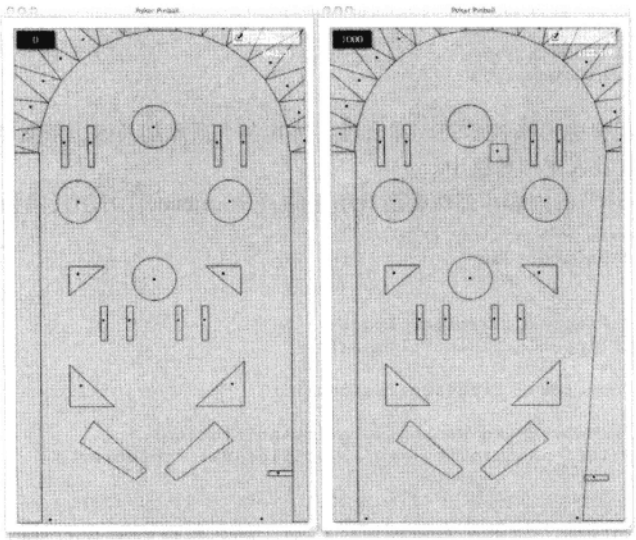


图 9-10　在弹珠台游戏中检测碰撞所用的多边形。左图为弹珠发射前的情况，右图为发射后的情况

上述方法先调用 shape 对象的 stroke() 方法来绘制多边形，然后绘制一个半径为 1.5 像素的小圆，用以表示多边形中心[⊖]的大致位置。

为了检测碰撞，对于每一个可能和弹珠相撞的物体，弹珠台程序都会为其创建一个圆形或多边形，这段代码如下所示：

```
var game = new Game('pinball', 'gameCanvas'),
    ...
    // Collision Detection.........................................

    shapes = [],
    ...

    fiveHundredBumper = new Circle(256, 187, 40),
    oneHundredBumperRight = new Circle(395, 328, 40),
    oneHundredBumperLeft = new Circle(116, 328, 40),
    fiftyBumper = new Circle(255, 474, 40),

    leftBoundary = new Polygon(),
    rightBoundary = new Polygon(),
    ...

leftBoundary.points.push(new Point(45, 235));
leftBoundary.points.push(new Point(45, game.context.canvas.height));
leftBoundary.points.push(new Point(-450, game.context.canvas.height));
leftBoundary.points.push(new Point(-450, 235));
leftBoundary.points.push(new Point(45, 235));
```

⊖ centroid，又叫“几何中心”、“重心”，详情参见：https://zh.wikipedia.org/zh-cn/几何中心。——译者注

```
rightBoundary.points.push(new Point(508, 235));
rightBoundary.points.push(new Point(508, game.context.canvas.height));
rightBoundary.points.push(new Point(508*2, game.context.canvas.height));
rightBoundary.points.push(new Point(508*2, 235))
rightBoundary.points.push(new Point(508, 235));
...
shapes.push(leftBoundary);
shapes.push(rightBoundary);
...
```

如果某物体是用多边形来表示的，那么还要将其各个顶点加入多边形对象中。此后，程序会把创建好的图形对象放入图形数组中。

回想一下 9.3.1 小节的内容，弹珠台游戏所用的 startAnimate() 方法代码如下：

```
game.startAnimate = function () {
    var collisionOccurred;
    ...

    adjustRightFlipperCollisionPolygon();
    adjustLeftFlipperCollisionPolygon();

    collisionOccurred = detectCollisions();

    if (!collisionOccurred && applyGravityAndFriction) {
        applyFrictionAndGravity(); // Modifies ball velocity
    }
};
```

在绘制每一帧动画前，如果左右弹板正在移动，那么弹珠台游戏的 startAnimate() 方法就会调整碰撞检测所用的多边形对象。此后，调用 game 对象的 detectCollisions() 方法来检测并应对可能发生的碰撞。该方法的代码列在程序清单 9-23 之中。

为了使用分离轴定理来判断弹珠是否与其他物体发生碰撞，detectCollisions() 方法会把除了弹珠之外的所有 shape 对象一一传递给弹珠对象的 collidesWith() 方法，同时传入的还有弹珠的位移向量。

在 8.4.2 小节中讲过，弹珠对象的 collidesWith() 方法所返回的 MinimumTranslationVector 对象，表示要将弹珠恢复到未发生碰撞时的状态，所需移动的最小距离。弹珠台游戏的 detectCollisons() 方法把这个位移向量传递给 game 对象的 bounce() 方法，让弹珠从与之相撞的物体表面上反弹回来。

因为 ballShape 对象的 collidesWith() 方法已经将分离轴定理检测法实现好了，所以如果弹板是不动的，那么通过该方法很容易就可以检测到弹珠与弹板之间的碰撞。然而，由于弹珠的移动速度很快（每秒大约 400 像素），而且弹板在移动的时候，其角速度也不低，所以分离轴定理检测法可能会漏判弹珠与正在移动的弹板之间的碰撞。因此，game 对象的 detectCollisions() 方法还要针对每个弹板调用 detectFlipperCollision() 方法，使用光线投射法来检测小球与弹板的碰撞。有关 detectFlipperCollision() 方法的更多信息，请参阅程序清单 9-26。

程序清单 9-23　使用分离轴定理来检测碰撞

```
function collisionDetected(mtv) {
    return mtv.axis !== undefined && mtv.overlap !== 0;
};

function detectCollisions() {
    var mtv, shape, displacement, position, lastPosition;
```

```
if (!launching && !loading && !game.paused) {
    ballShape.x = ballSprite.left;
    ballShape.y = ballSprite.top;
    ballShape.points = [];
    ballShape.setPolygonPoints();

    position = new Vector(new Point(ballSprite.left,
                                    ballSprite.top));

    lastPosition = new Vector(new Point(lastBallPosition.x,
                                        lastBallPosition.y));

    displacement = position.subtract(lastPosition);

    for (var i=0; i < shapes.length; ++i) {
        shape = shapes[i];

        if (shape !== ballShape) {
            mtv = ballShape.collidesWith(shape, displacement);
            if (collisionDetected(mtv)) {
                updateScore(shape);

                setTimeout ( function (e) {
                    bumperLit = undefined;
                }, 100);

                if (shape === twoXBumperLeft        ||
                    shape === twoXBumperRight       ||
                    shape === fiveXBumperRight      ||
                    shape === fiveXBumperLeft       ||
                    shape === fiftyBumper           ||
                    shape === oneHundredBumperLeft  ||
                    shape === oneHundredBumperRight ||
                    shape === fiveHundredBumper) {
                    game.playSound('bumper');
                    bounce(mtv, shape, 4.5);
                    bumperLit = shape;
                    return true;
                }
                else if (shape === rightFlipperShape) {
                    if (rightFlipperAngle === 0) {
                        bounce(mtv, shape, 1 + rightFlipperAngle);
                        return true;
                    }
                }
                else if (shape === leftFlipperShape) {
                    if (leftFlipperAngle === 0) {
                        bounce(mtv, shape, 1 + leftFlipperAngle);
                        return true;
                    }
                }
                else if (shape === actuatorPlatformShape) {
                    bounce(mtv, shape, 0.2);
                    return true;
                }
                else {
                    bounce(mtv, shape, 0.96);
                    return true;
                }
            }
        }
    }
```

```
        detectFlipperCollision(LEFT_FLIPPER);
        detectFlipperCollision(RIGHT_FLIPPER);

        return flipperCollisionDetected;
    }
    return false;
}
```

不过，现在我们关注的还是 bounce() 方法，它可以让弹珠从与之发生碰撞的物体上弹回来。该方法及其辅助方法的代码，都列在了程序清单 9-24 之中。

bounce() 方法会根据弹珠的移动速度创建一条表示速度的单位向量，它还要创建一条与最小平移向量相垂直的向量。我们最终要根据这个垂直向量算出入射的单位向量所对应的反射向量。然而，这样的垂直向量却有两条，所以游戏必须决定到底依据哪一条来计算反射向量。

程序清单 9-24　将弹珠从碰撞物体上弹回

```
function clampBallVelocity() {
    if (ballSprite.velocityX > MAX_BALL_VELOCITY)
        ballSprite.velocityX = MAX_BALL_VELOCITY;
    else if (ballSprite.velocityX < -MAX_BALL_VELOCITY)
        ballSprite.velocityX = -MAX_BALL_VELOCITY;

    if(ballSprite.velocityY > MAX_BALL_VELOCITY)
        ballSprite.velocityY = MAX_BALL_VELOCITY;
    else if (ballSprite.velocityY < -MAX_BALL_VELOCITY)
        ballSprite.velocityY = -MAX_BALL_VELOCITY;
};

function separate(mtv) {
    var dx, dy, velocityMagnitude, point, theta=0,
        velocityVector = new Vector(new Point(ballSprite.velocityX,
                                              ballSprite.velocityY)),
        velocityUnitVector = velocityVector.normalize();

    if (mtv.axis.x === 0) {
        theta = Math.PI/2;
    }
    else {
        theta = Math.atan(mtv.axis.y / mtv.axis.x);
    }

    dy = mtv.overlap * Math.sin(theta);
    dx = mtv.overlap * Math.cos(theta);

    if (mtv.axis.x < 0 && dx > 0 || mtv.axis.x > 0 && dx < 0) dx = -dx;
    if (mtv.axis.y < 0 && dy > 0 || mtv.axis.y > 0 && dy < 0) dy = -dy;

    ballSprite.left += dx;
    ballSprite.top += dy;
}

function checkMTVAxisDirection(mtv, shape) {
    var flipOrNot,
        centroid1 = new Vector(ballShape.centroid()),
        centroid2 = new Vector(shape.centroid()),
        centroidVector = centroid2.subtract(centroid1),
        centroidUnitVector = (new Vector(centroidVector)).normalize();

    if (centroidUnitVector.dotProduct(mtv.axis) > 0) {
```

```
        mtv.axis.x = -mtv.axis.x;
        mtv.axis.y = -mtv.axis.y;
    }
}
function bounce(mtv, shape, bounceCoefficient) {
    var velocityVector = new Vector(new Point(ballSprite.velocityX,
                                              ballSprite.velocityY)),
        velocityUnitVector = velocityVector.normalize(),
        velocityVectorMagnitude = velocityVector.getMagnitude(),
        reflectAxis, point;

    checkMTVAxisDirection(mtv, shape);

    if (!loading && !game.paused) {
        if (mtv.axis !== undefined) {
            reflectAxis = mtv.axis.perpendicular();
        }

        separate(mtv);

        point = velocityUnitVector.reflect(reflectAxis);

                if (shape === leftFlipperShape || shape === rightFlipperShape) {
            if (velocityVectorMagnitude < MIN_BALL_VELOCITY_OFF_FLIPPERS)
                velocityVectorMagnitude = MIN_BALL_VELOCITY_OFF_FLIPPERS;
        }

        ballSprite.velocityX = point.x * velocityVectorMagnitude *
                               bounceCoefficient;

        ballSprite.velocityY = point.y * velocityVectorMagnitude *
                               bounceCoefficient;

        clampBallVelocity();
    }
}
```

checkMTVAxisDirection() 方法负责决定到底依据哪一条向量来反射弹珠的速度向量，该方法先创建一条向量，它从弹珠的中心出发，指向与之相撞的另一个物体的中心。然后，此方法会求出这个向量与最小平移向量之间的点积。如果其点积大于 0，则说明两个向量之间的夹角为锐角，这也就意味着二者大致位于同一个方向。

我们要让弹珠远离碰撞物体的中心，所以，如果点积大于 0，那么我们要就掉转最小平移向量的方向，这样弹珠才能远离与之相撞的物体。

bounce() 方法在决定好了反射弹珠速度向量时所依据的反射轴之后，就会调用 separate() 方法，使用最小平移向量将弹珠与碰撞物体分开。

接下来，bounce() 方法将设定弹珠被弹回之后的速度。然而，在设定弹珠速度时，它还需要对两种情况做出调整。首先，如果弹珠与弹板相撞时的移动速度很低，那么该方法就要将弹珠的速度调高到预先设定好的某个最小值。这样做是为了确保弹板总是能把弹珠打到一定距离之外，避免弹珠因为反弹速度过低而在弹板上反复跳跃的情况。最后，该方法要将弹珠的速度限定在某个最大值之内，否则，过高的移动速度可能会使玩家根本看不清弹珠，而且还会导致碰撞检测算法发生漏判。

9.3.6.2 弹珠与弹珠台圆顶的碰撞

正如图 9-11 所示，弹珠台游戏在实现碰撞检测算法时，会把弹珠台的圆顶视为多个三角形，以此来检测弹珠与圆顶之间的碰撞。

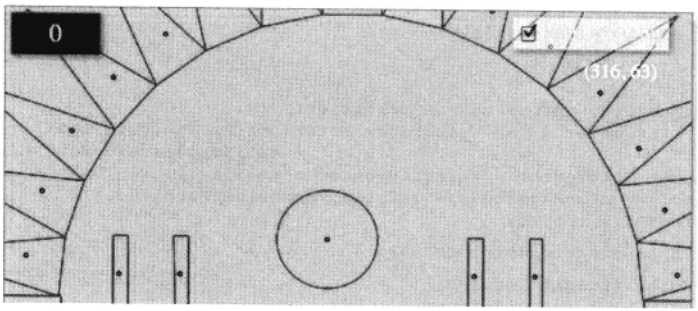


图 9-11 弹珠台的圆顶是由数个三角形拼接而成的

弹珠台游戏用如下列码来创建表示弹珠台圆顶的那些三角形：

```
var DOME_SIDES = 15,
    DOME_X = 275,
    DOME_Y = 235,
    DOME_RADIUS = 232,
    domePolygons = createDomePolygons(DOME_X, DOME_Y,
                                      DOME_RADIUS, DOME_SIDES);

domePolygons.forEach( function (polygon) {
    shapes.push(polygon);
});
```

游戏程序通过 createDomePolygons() 方法来创建三角形，并将每个创建好的三角形对象都加入检测碰撞所用的图形数组中。程序清单 9-25 列出了该方法的代码。

程序清单 9-25 所列代码很简单，它是以循环的方式来创建这 15 个三角形的。每次循环都会创建一个表示三角形的 Polygon 对象，然后算出此三角形的顶点，并将创建好的三角形加入一个数组。循环结束后，将包含三角形对象的数组返回。

这段代码中有个地方很有意思，那就是 midPointRadius，此方法需要用它的值计算出每个三角形中距离圆顶表面最远的那个顶点。该值要取得足够大，这样才能让每个三角形的中心离圆顶表面远一些。如果它的值比较小，例如我们用 radius*1.05 来取代 radius*1.5，那么创建出来的三角形就会变成图 9-12 中的样子。

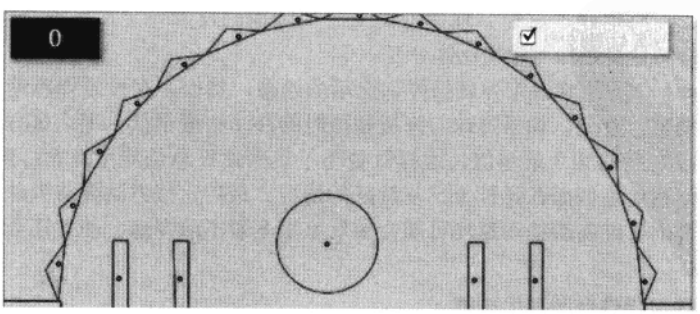


图 9-12 如果搭建圆顶所用的三角形太小，那么会影响碰撞检测的精确度

由于弹珠台游戏采用的是事后碰撞检测法，所以只有等碰撞发生了之后，我们才能检测到它。

这样的话，弹珠的中心有可能会在游戏尚未检测到碰撞之前，率先越过三角形的中心。如果发生了此种情况，那么算法中用于计算弹珠如何从三角形表面反弹回来的那个 centroidVector 向量的方向就会出错，而由它算出的弹珠反弹速度，其方向也是错的。该速度的方向本来是应该远离三角形的，而现在却变成了朝向三角形一方，于是弹珠就会粘在圆顶表面。为了保证弹珠不会粘在圆顶上，所以我们必须将三角形做得高一些，这样其中心才能远离圆顶表面，如图 9-11 所示。

程序清单 9-25　创建拼接圆顶所用的 Polygon 对象

```
function createDomePolygons(centerX, centerY, radius, sides) {
    var polygon,
        polygons = [],
        startTheta = 0,
        endTheta,
        midPointTheta,
        thetaDelta = Math.PI/sides,
        midPointRadius = radius*1.5;

    for (var i=0; i < sides; ++i) {
        polygon = new Polygon();

        endTheta = startTheta + thetaDelta;
        midPointTheta = startTheta + (endTheta - startTheta)/2;

        polygon.points.push(
            new Point(centerX + radius * Math.cos(startTheta),
                      centerY - radius * Math.sin(startTheta)));

        polygon.points.push(
            new Point(centerX + midPointRadius * Math.cos(midPointTheta),
                      centerY - midPointRadius * Math.sin(midPointTheta)));

        polygon.points.push(
            new Point(centerX + radius * Math.cos(endTheta),
                      centerY - radius * Math.sin(endTheta)));

        polygon.points.push(
            new Point(centerX + radius * Math.cos(startTheta),
                      centerY - radius * Math.sin(startTheta)));

        polygons.push(polygon);

        startTheta += thetaDelta;
    }
    return polygons;
}
```

9.3.6.3　检测弹珠与弹板之间的碰撞

如果弹板是静止的，那么弹珠台游戏就会使用分离轴定理来检测弹珠与弹板之间的碰撞。然而，如果弹板在弹起的同时，弹珠也在高速运动，那么分离轴定理就有可能漏判弹珠与正在弹起的弹板之间的碰撞。

鉴于分离轴定理在检测高速移动的小物体时，效果不甚理想，所以弹珠台游戏在采用该算法检测碰撞的同时，还要辅以 8.3 节中介绍过的光线投射法。程序清单 9-26 列出了 detectFlipperCollision() 方法的代码，如果检测不到弹珠与静止物体之间的碰撞，那么 detectCollisions() 方法就会调用此方法来检测弹珠与弹板之间的碰撞。

程序清单 9-26 检测弹珠与弹板之间的碰撞

```
function detectFlipperCollision(flipper) {
    var v1, v2, l1, l2, surface, ip, bbox = {}, riseTimer;

    bbox.top = 725;
    bbox.bottom = 850;

    if (flipper === LEFT_FLIPPER) {
        v1 = new Vector(leftFlipperBaselineShape.points[0].rotate(
                        LEFT_FLIPPER_ROTATION_POINT,
                        leftFlipperAngle));

        v2 = new Vector(leftFlipperBaselineShape.points[1].rotate(
                        LEFT_FLIPPER_ROTATION_POINT,
                        leftFlipperAngle));
        bbox.left = 170;
        bbox.right = 265;
        riseTimer = leftFlipperRiseTimer;
    }
    else if (flipper === RIGHT_FLIPPER) {
        v1 = new Vector(rightFlipperBaselineShape.points[0].rotate(
                        RIGHT_FLIPPER_ROTATION_POINT,
                        rightFlipperAngle));

        v2 = new Vector(rightFlipperBaselineShape.points[1].rotate(
                        RIGHT_FLIPPER_ROTATION_POINT,
                        rightFlipperAngle));

        bbox.left = 245;
        bbox.right = 400;
        riseTimer = rightFlipperRiseTimer;
    }
    if ( ! flipperCollisionDetected && riseTimer.isRunning() &&
        ballSprite.top + ballSprite.height > bbox.top &&
        ballSprite.left < bbox.right) {
        surface = v2.subtract(v1);
        l1 = new Line(new Point(ballSprite.left, ballSprite.top),
                      lastBallPosition),
        l2 = new Line(new Point(v2.x, v2.y), new Point(v1.x, v1.y)),
        ip = l1.intersectionPoint(l2);

        if (ip.x > bbox.left && ip.x < bbox.right) {
            reflectVelocityAroundVector(surface.perpendicular());

            ballSprite.velocityX = ballSprite.velocityX * 3.5;
            ballSprite.velocityY = ballSprite.velocityY * 3.5;

            if (ballSprite.velocityY > 0)
                ballSprite.velocityY = -ballSprite.velocityY;

            if (flipper === LEFT_FLIPPER && ballSprite.velocityX < 0)
                ballSprite.velocityX = -ballSprite.velocityX;
            else if (flipper === RIGHT_FLIPPER &&
                     ballSprite.velocityX > 0)
                ballSprite.velocityX = -ballSprite.velocityX;
        }
    }
}
```

detectFlipperCollision() 方法先创建两条向量，其中一条由原点指向弹板表面的第一个点，另

一条由原点指向弹板表面的第二个点。该方法随后用第二个向量减去第一个，于是就得出了一条沿着弹板边缘而延伸的向量。

这个方法还创建了两条线段，其中一条将弹珠在上一帧的位置与其当位置连接起来，而另外一条则紧贴着弹板的边缘。创建好了这两条线段后，该方法判断它们是否相交。

最后，如果弹珠离弹板足够近，同时两条线段的交点位于弹板的左右边沿之间，那么就说明两者已经相撞，于是该方法就会将弹珠的速度调整为相应的值。

9.4 总结

在前面数章中，我们已经准备好了大量的知识，于是本章就利用这些知识来向读者演示如何以 Canvas 实现一款 HTML5 游戏。

一开始，我们先实现了一个简单的游戏引擎，它大约有 450 行代码。尽管很小，但是创建游戏所需的基本组件却都已经做好了，包括游戏循环，以及对基于时间的运动、游戏暂停、高分记录、声音播放等功能的支持，此外也实现了按键监听器。

在有了这个简单而功能丰富的游戏引擎之后，我们又把注意力转移到了游戏原型程序的制作上。这个“Hello, World”式的游戏原型程序主要是为了演示如何实现游戏中的各个环节，它本身并没有与实际游戏内容相关的逻辑。

本章最后实现了一款精美的弹珠台游戏。游戏制作过程中需要应对一些复杂的游戏设计问题，诸如圆周运动、对重力与摩擦力建模，以及使用分离轴定理与光线投射法来检测碰撞，等等。

第10章
自定义控件

在本书中，读者已经学到了如何将一个或多个 canvas 元素与诸如文本输入框、按钮等标准的 HTML 控件相结合，来制作基于 Canvas 的应用程序。标准的 HTML 控件对于很多基于 Canvas 的应用程序来说足够用了，不过，仍然有一些应用程序需要使用自定义控件，尤其是当找不到合适的标准 HTML 控件，或者所用的标准 HTML 控件并未被所有浏览器支持的时候，更是如此。还有一种情况也得自己来制作控件，那就是你要求自己的控件在所有支持 HTML5 的浏览器中都必须具备相同的外观。

本章我们将从头开始，来实现如下 4 个控件：

- 圆角矩形（rounded rectangle）
- 进度条（progress bar）
- 滑动条（slider）
- 图像查看器（image panner）

本章所实现的控件都会执行下列操作：

- 将该控件所用的代码放在名为 COREHTML5 的全局对象中。
- 使用 draw() 方法将自身绘制到 canvas 中。
- 将 canvas 置于 DIV 元素内，使开发者可以通过 domElement 属性来访问此 DIV。
- 实现 appendTo(element) 方法，该方法可将控件对应的 DOM 元素加入某个 HTML 元素之中，并且会根据外围元素的大小来调整 DOM 元素与配套 canvas 的尺寸。

一般来说，不应把新创建的 JavaScript 对象置于全局命名空间之中。为了遵从此惯例，我们把本章所做的控件都放在一个叫做 COREHTML5 的对象中。开发者必须通过 COREHTML5 对象才能实例化某个控件对象。比方说，如果要创建 10.1 节中所说的圆角矩形，那么就应该使用如下代码：

```
roundedRectangle = new COREHTML5.RoundedRectangle(
                        'rgba(0,0,0,0.2)', 'darkgoldenrod', 90, 25);
```

COREHTML5 对象也是一个命名空间。其他人所实现的全局对象，有可能会与你所实现的那些对象重名，从而覆写了你定义好的对象。适当地使用命名空间，可以减少发生这种情况的可能性。比方说，其他人也极有可能实现一个名叫 Slider 的对象，不过他们几乎不可能将其放到名为 COREHTML5 的全局对象中，所以 COREHTML5.Slider 对象的重名几率很小。

由于本章所讲的控件都要基于 Canvas 来实现，所以它们都得创建一个 canvas 元素，随后将自己绘制到这个 canvas 上。用于绘制控件的 draw() 方法，接受一个名为 context 的可选参数，该参数必须是某个 Canvas 的绘图环境对象。如果指定了这个 context 参数，那么控件的 draw() 方法就会将自己绘制到那个绘图环境对象所对应的 Canvas 之上。要是没有指定 context 参数，那么 draw() 方法则会将控件绘制到与自己内部的绘图环境对象所对应的 Canvas 上。

为了使用本章所实现的控件，开发者必须先创建好控件实例，然后将控件的 DOM 元素加入 DOM 结构树中的某个 HTML 元素之中。这个操作可以通过控件的 appendTo(element) 方法来完

成。开发者也可以通过控件的 domElement 属性来访问其 DOM 元素，这意味着我们可以用代码来控制 DOM 元素的属性，像是这样：

```
roundedRectangle.domElement.style.position = 'absolute';
roundedRectangle.domElement.style.top = '50px';
roundedRectangle.domElement.style.left = '50px';
```

此外，开发者也可以在 CSS 中创建好某个 class，然后用程序代码将该 class 应用到控件的 DOM 元素上：

```
roundedRectangle.domElement.className = 'customRectangle';
```

本章的自定义控件所具备的上述特性，都是经由表 10-1 中所列的方法而实现出来的。本章的所有自定义控件都实现了这些方法。

表 10-1　自定义控件所支持的方法

方　法	描　述
appendTo(element)	将控件的 DOM 元素加入 element 参数所代表的元素之中，并且根据外围的 DOM 元素大小来调整控件的 DOM 元素与配套 canvas 的尺寸
createDOMElement()	创建控件的 DOM 元素
createCanvas()	创建控件的 canvas 元素
draw(context)	绘制此控件。context 参数是可选的，如果指定了该参数，那么控件则会被绘制到其所对应的 Canvas 之中，否则就绘制到自己内部的 Canvas 中
erase()	擦除控件的 canvas
resize(width, height)	重新调整控件 canvas 的大小

提示：将 DIV 元素暴露给开发者

本章所讲的控件都会创建一个 canvas 元素与一个 DIV 元素，将 canvas 元素加入 DIV 之中，并且通过 domElement 属性将 DIV 暴露给开发者。这样开发者就可以将该 DIV 元素加入 DOM 结构树的任意元素中了。

自定义控件还实现了一个名为 appendTo() 的方法，该方法可以将 DIV 元素加入另一个元素之中，并调整 DIV 与 canvas 元素的尺寸，使其符合那个外围元素的大小。

10.1　圆角矩形控件

图 10-1 之中的圆角矩形控件是本章的 4 个控件之中最简单的一个，进度条与滑动条控件也要用到它。

图 10-1 所示的应用程序含有一个圆角矩形控件，并提供了两个控制矩形宽度与高度的滑动条。用户拖动滑动条时，矩形将会持续地改变其大小，以符合滑动条所指示的尺寸。

程序清单 10-1 列出了程序所用的 HTML 代码，这段代码创建了两个滑动条控件，以及一个名为 roundedRectangleDiv 的 DIV 元素。

程序清单 10-1　演示圆角矩形控件的应用程序所用的 HTML 代码

```
<!DOCTYPE html>
   <head>
     <title>Rounded Rectangles</title>
      <style>
```

```
        body {
           background: bisque;
        }

        #roundedRectangleDiv {
           position: absolute;
           left: 50px;
           top: 70px;
           width: 450px;
           height: 80px;
        }

        .range {
           vertical-align: -5px;
        }
        #controls {
           color: blue;
           margin-top: 20px;
           margin-left: 65px;
        }

        #widthRangeDiv {
           margin-right: 30px;
           display: inline;
        }
     </style>
  </head>

  <body>
     <div id='controls'>
        <div id='widthRangeDiv'>
          Width:  <input id='widthRange' class='range'  type='range'
                         minimum='5' maximum='100'/>
        </div>

        Height: <input id='heightRange' class='range'  type='range'
                        minimum='5' maximum='100'/>
     </div>

     <div  id='roundedRectangleDiv'></div>

     <script  src='roundedRectangle.js'></script>
     <script  src='example.js'></script>
  </body>
</html>
```

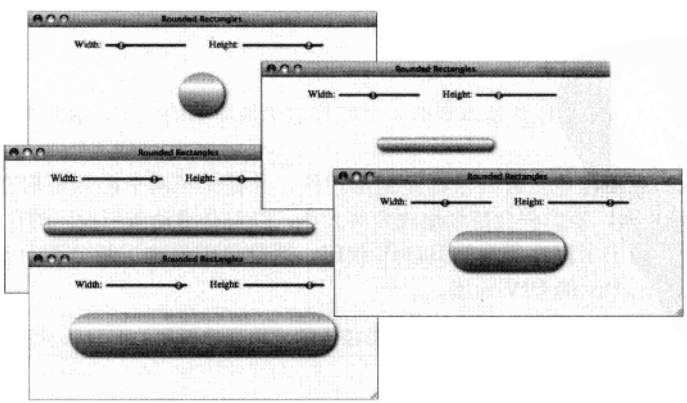


图 10-1　圆角矩形控件

程序清单 10-2 所列的 JavaScript 代码，创建了一个圆角矩形，并使用该矩形的 appendTo() 方法将该矩形控件的 DIV 元素添加到 roundedRectangleDiv 之中。此外，应用程序还向滑动条控件注册了处理数值改动所用的事件处理器，当用户变更了滑动条的值之后，事件处理器要调整矩形控件的大小，并将其重绘。

应用程序调用接受 4 个参数的 COREHTML5.RoundedRectangle 构造器来创建圆角矩形控件。程序清单 10-3 列出了该构造器的代码。前两个参数代表控件在绘制圆角矩形时所采用的描边及填充方式，而后两个参数则表示圆角矩形的相对高度与宽度。这两个值是相对于它所在的 DOM 元素来说的，它们表示圆角矩形在水平及垂直方向上与其外围 DOM 元素高度及宽度的百分比，可以用 0 ～ 1.0 之间的值来表示，也可以取 1 ～ 100 之间的值。图 10-1 所示应用程序在启动时会根据滑动条的初始值来设置这些参数，图 10-2 展示的就是应用程序以初始配置值运行的情况。

程序清单 10-2　演示圆角矩形控件的应用程序所用的 JavaScript 代码

```
var widthRange = document.getElementById('widthRange'),
    heightRange = document.getElementById('heightRange'),
    roundedRectangle = new COREHTML5.RoundedRectangle(
                            'rgba(0,0,0,0.2)', 'darkgoldenrod',
                            widthRange.value, heightRange.value);

// Event handlers.....................................................

function resize() {
   roundedRectangle.horizontalSizePercent = widthRange.value/100;
   roundedRectangle.verticalSizePercent   = heightRange.value/100;

   roundedRectangle.resize(roundedRectangle.domElement.offsetWidth,
                           roundedRectangle.domElement.offsetHeight);

   roundedRectangle.erase();
   roundedRectangle.draw();
}

// Initialization....................................................

widthRange.onchange  = resize;
heightRange.onchange = resize;

roundedRectangle.appendTo(
   document.getElementById('roundedRectangleDiv'));

roundedRectangle.draw();
```

程序清单 10-3　圆角矩形控件对象的代码

```
var COREHTML5 = COREHTML5 ||  {};

// Constructor.......................................................

COREHTML5.RoundedRectangle = function(strokeStyle, fillStyle,
                                      horizontalSizePercent,
                                      verticalSizePercent) {
   this.strokeStyle = strokeStyle ? strokeStyle : 'gray';
   this.fillStyle   = fillStyle   ? fillStyle   : 'skyblue';

   horizontalSizePercent = horizontalSizePercent || 100;
   verticalSizePercent   = verticalSizePercent   || 100;

   this.SHADOW_COLOR = 'rgba(100,100,100,0.8)';
   this.SHADOW_OFFSET_X = 3;
```

```
    this.SHADOW_OFFSET_Y = 3;
    this.SHADOW_BLUR = 3;

    this.setSizePercents(horizontalSizePercent, verticalSizePercent);
    this.createCanvas();
    this.createDOMElement();

    return  this;
}

// Prototype.......................................................

COREHTML5.RoundedRectangle.prototype  =  {

    // General functions.............................................

    createCanvas: function () {
        var canvas = document.createElement('canvas');
        this.context = canvas.getContext('2d');
        return canvas;
    },

    createDOMElement: function () {
        this.domElement = document.createElement('div');
        this.domElement.appendChild(this.context.canvas);
    },

    appendTo: function (element) {
        element.appendChild(this.domElement);
        this.domElement.style.width = element.offsetWidth +  'px';
        this.domElement.style.height = element.offsetHeight +  'px';
        this.resize(element.offsetWidth, element.offsetHeight);
    },

    resize: function (width, height)  {
        this.HORIZONTAL_MARGIN = (width - width *
                                  this.horizontalSizePercent)/2;
        this.VERTICAL_MARGIN   = (height - height *
                                  this.verticalSizePercent)/2;

        this.cornerRadius =  (this.context.canvas.height/2  -
                              2*this.VERTICAL_MARGIN)/2;

        this.top    = this.VERTICAL_MARGIN;
        this.left   = this.HORIZONTAL_MARGIN;
        this.right  =  this.left + width  - 2*this.HORIZONTAL_MARGIN;
        this.bottom = this.top  + height - 2*this.VERTICAL_MARGIN;

        this.context.canvas.width = width;
        this.context.canvas.height = height;
    },

    setSizePercents: function (h, v) {
        // horizontalSizePercent and verticalSizePercent
        // represent the size of the rounded rectangle in terms
        // of horizontal and vertical percents of the rectangle's
        // enclosing DOM element.

        this.horizontalSizePercent = h > 1 ? h/100 : h;
        this.verticalSizePercent   = v > 1 ? v/100 : v;
    },
```

```
// Drawing functions.............................................

fill: function  () {
   var radius =  (this.bottom - this.top)  / 2;

   this.context.save();
   this.context.shadowColor   = this.SHADOW_COLOR;
   this.context.shadowOffsetX = this.SHADOW_OFFSET_X;
   this.context.shadowOffsetY = this.SHADOW_OFFSET_Y;
   this.context.shadowBlur = 6;

   this.context.beginPath();

   this.context.moveTo(this.left + radius,  this.top);

   this.context.arcTo(this.right, this.top,
                      this.right, this.bottom, radius);

   this.context.arcTo(this.right, this.bottom,
                      this.left, this.bottom, radius);

   this.context.arcTo(this.left, this.bottom,
                      this.left, this.top, radius);

   this.context.arcTo(this.left, this.top,
                      this.right, this.top, radius);

   this.context.closePath();

   this.context.fillStyle = this.fillStyle;
   this.context.fill();
   this.context.shadowColor = undefined;
},

overlayGradient:  function  ()  {
   var gradient =
       this.context.createLinearGradient(this.left, this.top,
                                         this.left, this.bottom);

   gradient.addColorStop(0,    'rgba(255,255,255,0.4)');
   gradient.addColorStop(0.2,  'rgba(255,255,255,0.6)');
   gradient.addColorStop(0.25, 'rgba(255,255,255,0.7)');
   gradient.addColorStop(0.3,  'rgba(255,255,255,0.9)');
   gradient.addColorStop(0.40, 'rgba(255,255,255,0.7)');
   gradient.addColorStop(0.45, 'rgba(255,255,255,0.6)');
   gradient.addColorStop(0.60, 'rgba(255,255,255,0.4)');
   gradient.addColorStop(1,    'rgba(255,255,255,0.1)');

   this.context.fillStyle = gradient;
   this.context.fill();

   this.context.lineWidth =  0.4;
   this.context.strokeStyle =  this.strokeStyle;
   this.context.stroke();

   this.context.restore();
},

draw: function (context) {
   var originalContext;

   if (context) {
```

```
            originalContext = this.context;
            this.context = context;
         }

         this.fill();
         this.overlayGradient();

         if (context) {
            this.context = originalContext;
         }
      },

      erase: function()  {
         // Erase the entire canvas

         this.context.clearRect(0,  0,  this.context.canvas.width,
                                        this.context.canvas.height);
      }
   };
```

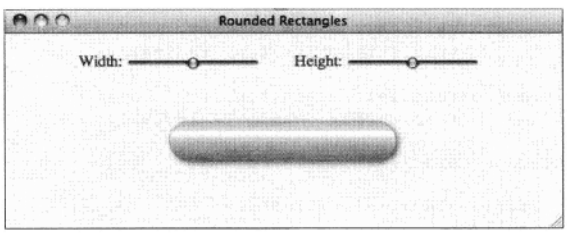


图 10-2　以初始配置值运行的应用程序

请注意程序清单 10-3 中的第一行代码：var COREHTML5 = COREHTML5 || {};。如果不存在名为 COREHTML5 的全局对象，那么这行代码就将其创建出来。此后，程序清单 10-3 中的代码又在全局的 COREHTML5 对象中添加了一个作为构造器而用的 RoundedRectangle 函数，并且定义了 RoundedRectangle 所对应的原型对象（prototype object）。

COREHTML5.RoundedRectangle 之中的方法分为两类。第一类是诸如 appendTo()、resize() 这样的通用函数，它们可以操作控件的 canvas 及 DIV 元素。第二类则用于绘制圆角矩形。在阅读程序清单时，尤其要注意 appendTo() 与 draw() 方法。

appendTo() 方法可以将圆角矩形控件之中的 DOM 元素加入指定的 HTML 元素中。比方说，图 10-1 中的应用程序就把圆角矩形添加到了一个名为 roundedRectangleDiv 的 DIV 元素之中，如图 10-3 所示。

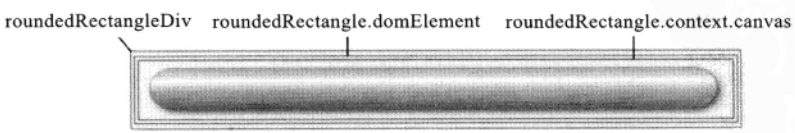


图 10-3　将圆角矩形加入 DIV 元素中（注意：图中三个元素大小相同）

此后，appendTo() 方法将圆角矩形 DOM 元素的大小设置成与上级 DOM 元素相同的值，并调用 resize() 方法，使控件中 canvas 元素的尺寸也与这两个 DOM 元素相符。举例来说，假设我们

将某个圆角矩形添加到一个 500 像素宽、400 像素高的 DIV 元素中，那么圆角矩形的 appendTo() 方法就会将矩形控件的 DOM 元素及 canvas 元素也设置成 500 像素宽、400 像素高。这么做有个前提，那就是矩形必须将外围元素的空间全部占满。回想一下 COREHTML5.RoundedRectangle 构造器的后两个参数，它们用来决定圆角矩形控件在上级元素中到底会占据多大的空间。

在绘制圆角矩形时，可以向 draw() 方法传入一个 Canvas 绘图环境对象，这样的话，圆角矩形就会被绘制到那个 Canvas 之中了。如果想把控件绘制到离屏 Canvas 中，那么这项功能就会很有用。实际上，在 10.2 节中，我们就是用这个办法来实现进度条控件的。

若是像图 10-1 所示应用程序这样，在绘制圆角矩形时，没有给 draw() 方法传入 Canvas 绘图环境对象，那么圆角矩形就会被绘制到自己内部的 Canvas 之中。

圆角矩形的绘制要分两步来做。首先，draw() 方法将圆角矩形填充成一个如图 10-4 上方截图所示的实心形状。然后，draw() 方法会在刚才画好的实心图形顶部叠加一道白色调的渐变色，用来模拟从顶部打下来的灯光，使得矩形表面看起来有一定的弧度，如图 10-4 下方截图所示。

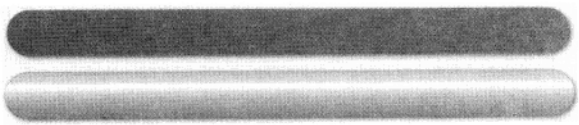


图 10-4　用半透明的渐变色叠加效果来绘制圆角矩形

小技巧：如何运用圆角矩形控件

本节所讲的圆角矩形控件有两种用途。一种情况是，它所封装的功能可能会被其他控件用到。比如说，本章要讲的进度条与滑动条控件都会用到圆角矩形。

还有一种用法，就是你可以利用圆角矩形控件来实现一些与之无关的自定义控件。我们可以保留 COREHTM5.RoundedRectangle 对象所提供的大部分通用功能，例如 appendTo()、resize()，等等，然后自己重新实现绘制代码。事实上，本章接下来要实现的那几个控件采用的正是这个办法。

10.2　进度条控件

上一节讲了如何用简单的圆角矩形来实现基于 Canvas 的控件，本节我们则要以使用了圆角矩形的进度条为例，来研究“复合控件”（composite control），也就是那种本身包含其他控件的控件。图 10-5 演示了正在向前走的进度条控件。

程序清单 10-4 列出了图 10-5 所示应用程序的 HTML 代码。

这段 HTML 代码创建了“Start”按钮和用以显示“Loading...”字样的 span 元素。正如图 10-5 最上方的截图所示，这个 span 元素起初是不可见的。在 HTML 代码中，还创建了一个名叫 progressbarDiv 的 DIV 元素。

接下来，我们在 HTML 代码中把创建 COREHTML5.Progressbar 及 COREHTML.Rounded-Rectangle 控件所用的 JavaScript 程序文件也加载进来。由于进度条控件要用到圆角矩形，所以为了使进度条控件所在的 JavaScript 程序文件能够正常运作，我们必须把圆角矩形所在的那个 JavaScript 文件一并引入才行。然后，还要在 HTML 中引入 requestNextAnimationFrame() 方法所用的 JavaScript 程序文件。本书 5.1.3 小节讲过这个制作动画所用的“polyfill 式方法”，此范例程

序将用它来制作进度条动画。

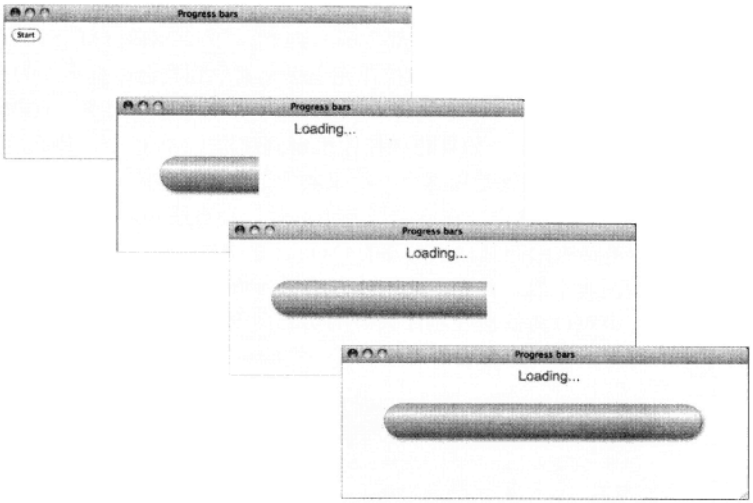


图 10-5 正在向前走的进度条控件

程序清单 10-5 列出了应用程序所用的 JavaScript 代码。

应用程序创建了一个以青色（teal）填充的进度条控件，并使用近乎半透明的黑色为其描边。这个进度条占据了外围 DOM 元素 90% 的宽度及 70% 的高度。此后，程序将进度条控件加入名为 progressbarDiv 的 DIV 元素中。

程序清单 10-4 进度条控件演示程序所用的 HTML 代码

```
<!DOCTYPE html>
   <head>
    <title>Progress bars</title>

      <style>
         body {
            background: linen;
         }

         #loadingSpan {
            font: 20px Arial;
            font-align: center;
            position: absolute;
            left: 250px;
            color: teal;
            text-shadow: 1px 1px rgba(0,0,0,0.1);
         }

         #progressbarDiv {
            position: absolute;
            left: 35px;
            top: 50px;
            width: 500px;
            height: 70px;
         }
```

```
      </style>
    </head>

  <body>
    <input type='button' id='startButton' value='Start'/>
    <span id='loadingSpan' style='display: none'>Loading...</span>

    <div id='progressbarDiv'></div>

    <script src='roundedRectangle.js'></script>
    <script src='progressbar.js'></script>
    <script src='requestNextAnimationFrame.js'></script>
    <script src='example.js'></script>
  </body>
</html>
```

应用程序也实现了 onclick 事件的处理，当用户按下 Start 按钮时，进度条动画就开始播放了。程序清单 10-6 列出了 COREHTML5.Progressbar 对象的代码。

程序清单 10-5　进度条控件演示程序所用的 JavaScript 代码

```
var startButton = document.getElementById('startButton'),
    loadingSpan = document.getElementById('loadingSpan'),
    progressbar = new COREHTML5.Progressbar('rgba(0,0,0,0.2)',
                                            'teal', 90, 70),
    percentComplete = 0;

// Event handlers.....................................................

startButton.onclick = function (e) {
   loadingSpan.style.display = 'inline';
   startButton.style.display = 'none';

   percentComplete += 1.0;

   if (percentComplete > 100) {
      percentComplete = 0;
      loadingSpan.style.display = 'none';
      startButton.style.display = 'inline';
   }
   else {
      progressbar.erase();
      progressbar.draw(percentComplete);
      requestNextAnimationFrame(startButton.onclick);
   }
};

// Initialization....................................................

progressbar.appendTo(document.getElementById('progressbarDiv'));
```

进度条控件会创建下面 4 样东西：

- 一个 COREHTML5.RoundedRectangle 对象
- 一个显示在屏幕上的 canvas
- 一个离屏 canvas
- 一个 DOM 元素（也就是 DIV 元素）

进度条的 appendTo() 方法先将圆角矩形画在离屏 canvas 之中，等到开发者调用进度条的 draw() 方法时，进度条控件会根据 percentComplete 属性将离屏 canvas 中相应的那部分图像复制

到屏幕 canvas 之中。

本章讲到这里，读者已经学会了如何实现简单的控件，还了解了怎样制作由其他控件组合而成的复合控件。接下来，我们将要制作一种更为复杂的控件，它可以处理发生在其上的用户事件。

程序清单 10-6　进度条控件的实现代码

```
var COREHTML5 = COREHTML5 || {};

// Constructor.................................................

COREHTML5.Progressbar = function(strokeStyle, fillStyle,
                                 horizontalSizePercent,
                                 verticalSizePercent) {
   this.trough = new COREHTML5.RoundedRectangle(strokeStyle,
                                                fillStyle,
                                                horizontalSizePercent,
                                                verticalSizePercent);
   this.SHADOW_COLOR = 'rgba(255,255,255,0.5)';
   this.SHADOW_BLUR = 3;
   this.SHADOW_OFFSET_X = 2;
   this.SHADOW_OFFSET_Y = 2;

   this.percentComplete = 0;
   this.createCanvases();
   this.createDOMElement();

   return  this;
}

// Prototype...................................................

   COREHTML5.Progressbar.prototype = {
      createDOMElement: function () {
         this.domElement = document.createElement('div');
         this.domElement.appendChild(this.context.canvas);
      },

      createCanvases: function () {
         this.context = document.createElement('canvas').
                        getContext('2d');
         this.offscreen = document.createElement('canvas').
                        getContext('2d');
      },

      appendTo: function (element) {
         element.appendChild(this.domElement);

         this.domElement.style.width  = element.offsetWidth  + 'px';
         this.domElement.style.height = element.offsetHeight + 'px';

         this.resize(); // Erases everything in the canvases

         this.trough.resize(element.offsetWidth, element.offsetHeight);
         this.trough.draw(this.offscreen);
      },

      setCanvasSize: function () {
         var domElementParent = this.domElement.parentNode;

         this.context.canvas.width = domElementParent.offsetWidth;
```

```
        this.context.canvas.height = domElementParent.offsetHeight;
    },

    resize: function () {
        var domElementParent = this.domElement.parentNode,
        w = domElementParent.offsetWidth,
        h = domElementParent.offsetHeight;

        this.setCanvasSize();

        this.context.canvas.width = w;
        this.context.canvas.height = h;

        this.offscreen.canvas.width = w;
        this.offscreen.canvas.height = h;
    },

    draw: function (percentComplete) {
        if (percentComplete >  0)  {

            // Copy the appropriate region of the foreground canvas
            // to the same region of the onscreen canvas

            this.context.drawImage(
                this.offscreen.canvas, 0, 0,
                this.offscreen.canvas.width*(percentComplete/100),
                this.offscreen.canvas.height,
                0, 0,
                this.offscreen.canvas.width*(percentComplete/100),
                this.offscreen.canvas.height);
        }
    },

    erase: function() {
        this.context.clearRect(0, 0,
                               this.context.canvas.width,
                               this.context.canvas.height);
    },
};
```

10.3 滑动条控件

许多控件都会把发生在控件上的事件推送给注册好的监听器。所以说，学会在基于 Canvas 的自定义控件中进行事件处理是一项重要的技能。本节我们就来讲解如何将事件处理机制纳入图 10-6 所示的滑动条控件之中。

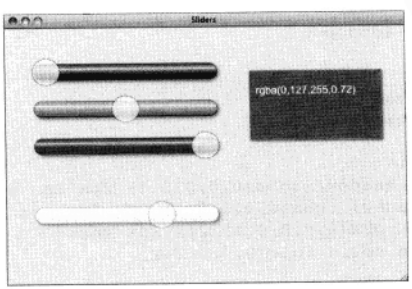


图 10-6　演示滑动条控件用法的应用程序

图 10-6 所示应用程序是一个简单的颜色选择器（color picker），其中前三个滑动条分别用于控制红、绿、蓝颜色分量，另外一个则可以调整颜色的不透明度（opacity）。当用户通过拖动滑块来调整控件值的时候，应用程序不仅会改变右方色标（color patch）之中的颜色，而且还会改变滑动条中那个圆角矩形的颜色。图 10-7 演示了如何在程序中通过调整滑动条来调配各种颜色。

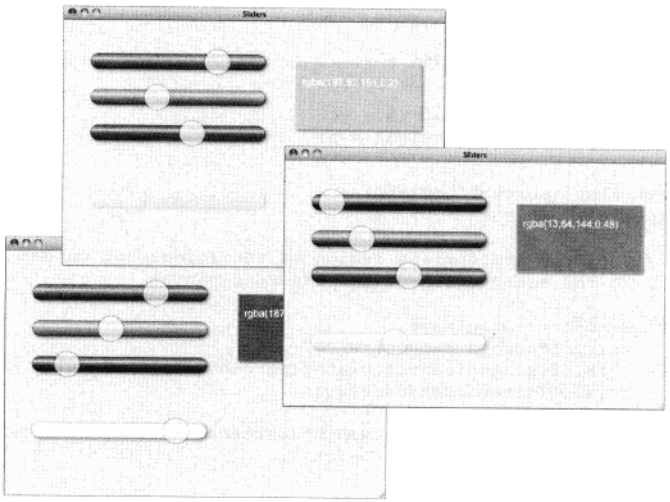


图 10-7　用滑动条控件调配各种颜色

程序清单 10-7 列出了图 10-6 所示应用程序的 HTML 代码。

这段 HTML 代码先把与每个滑动条对应的 DIV 元素都创建好，然后创建显示色标所用的 canvas。接下来，程序的 JavaScript 代码将创建 4 个滑动条控件，并把它们分别加入对应的 DIV 元素中。图 10-8 描述了与调整蓝色分量所用的滑动条控件相对应的元素结构。

程序清单 10-7　滑动条控件演示程序所用的 HTML 代码

```
<!DOCTYPE html>
   <head>
      <title>Sliders</title>

      <style>
         body {
            background: #dddddd;
         }

         #colorPatchCanvas {
            position: absolute;
            top: 75px;
            left: 410px;
            -webkit-box-shadow: rgba(0,0,0,0.5) 2px 2px 4px;
            -moz-box-shadow: rgba(0,0,0,0.5) 2px 2px 4px;
            box-shadow: rgba(0,0,0,0.5) 2px 2px 4px;
            border: thin solid rgba(0,0,0,0.2);
         }

         .slider {
```

```
            width: 324px;
            height: 50px;
         }

         #redSliderDiv {
            position: absolute;
            left: 40px;
            top: 50px;
         }

         #greenSliderDiv {
            position: absolute;
            left: 40px;
            top: 115px;
         }

         #blueSliderDiv {
            position: absolute;
            left: 40px;
            top: 180px;
         }

         #alphaSliderDiv {
            position: absolute;
            left: 40px;
            top: 300px;
         }
      </style>
   </head>

   <body>
      <div id='redSliderDiv'   class='slider'></div>
      <div id='greenSliderDiv' class='slider'></div>
      <div id='blueSliderDiv'  class='slider'></div>
      <div id='alphaSliderDiv' class='slider'></div>

      <canvas id='colorPatchCanvas' width='220' height='120'>
         Canvas not supported
      </canvas>

      <script src='roundedRectangle.js'></script>
      <script src='slider.js'></script>
      <script src='example.js'></script>
   </body>
</html>
```

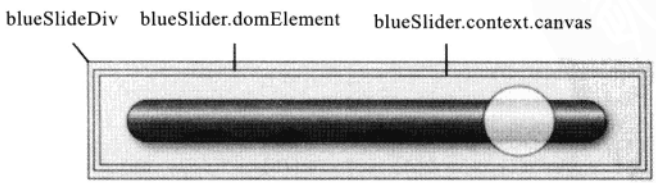


图 10-8　滑动条控件的元素结构：外围 DIV 元素（blueSliderDiv）包含控件本身的 DIV 元素（blueSlider.domElement），即该 DIV 元素中又嵌套了一个 HTML5 Canvas 元素

程序清单 10-8 列出了应用程序所用的 JavaScript 代码。

程序清单 10-8 滑动条控件演示程序所用的 JavaScript 代码

```
var colorPatchContext = document.getElementById('colorPatchCanvas').
                            getContext('2d'),

    redSlider   = new COREHTML5.Slider('rgb(0,0,0)',
                                       'rgba(255,0,0,0.8)', 0),

    blueSlider  = new COREHTML5.Slider('rgb(0,0,0)',
                                       'rgba(0,0,255,0.8)', 1.0),

    greenSlider = new COREHTML5.Slider('rgb(0,0,0)',
                                       'rgba(0,255,0,0.8)', 0.25),

    alphaSlider = new COREHTML5.Slider('rgb(0,0,0)',
                                       'rgba(255,255,255,0.8)', 0.5);

redSlider.appendTo('redSliderDiv');
blueSlider.appendTo('blueSliderDiv');
greenSlider.appendTo('greenSliderDiv');
alphaSlider.appendTo('alphaSliderDiv');

// Functions.......................................................

function updateColor() {
   var alpha = new Number((alphaSlider.knobPercent).toFixed(2));
   var color = 'rgba('
               + parseInt(redSlider.knobPercent * 255) + ','
               + parseInt(greenSlider.knobPercent * 255) + ','
               + parseInt(blueSlider.knobPercent * 255) + ','
               + alpha + ')';

   colorPatchContext.fillStyle = color;

   colorPatchContext.clearRect(0, 0, colorPatchContext.canvas.width,
                                     colorPatchContext.canvas.height);

   colorPatchContext.fillRect(0, 0, colorPatchContext.canvas.width,
                                    colorPatchContext.canvas.height);

   colorPatchContext.font = '18px Arial';
   colorPatchContext.fillStyle = 'white';
   colorPatchContext.fillText(color, 10, 40);

   alpha = (alpha + 0.2 > 1.0) ? 1.0 : alpha + 0.2;
   alphaSlider.opacity = alpha;
}

// Event handlers..................................................

redSlider.addChangeListener( function() {
   updateColor();
   redSlider.fillStyle = 'rgb(' +
      (redSlider.knobPercent * 255).toFixed(0) + ', 0, 0)';
});

greenSlider.addChangeListener( function() {
   updateColor();
   greenSlider.fillStyle = 'rgb(0, ' +
      (greenSlider.knobPercent * 255).toFixed(0) + ', 0)';
});

blueSlider.addChangeListener( function () {
```

```
    updateColor();
    blueSlider.fillStyle = 'rgb(0, 0, ' +
       (blueSlider.knobPercent * 255).toFixed(0) + ')';
});

alphaSlider.addChangeListener( function() {
    updateColor();
    alphaSlider.fillStyle = 'rgba(255, 255, 255, ' +
       (alphaSlider.knobPercent * 255).toFixed(0) + ')';

    alphaSlider.opacity = alphaSlider.knobPercent;
});

// Initialization.....................................................

redSlider.fillStyle   = 'rgb(' +
    (redSlider.knobPercent * 255).toFixed(0) + ', 0, 0)';

greenSlider.fillStyle = 'rgb(0, ' +
    (greenSlider.knobPercent * 255).toFixed(0) + ', 0)';

blueSlider.fillStyle  = 'rgb(0, 0, '  +
    (blueSlider.knobPercent * 255).toFixed(0) + ')';

alphaSlider.fillStyle =  'rgba(255, 255, 255, '  +
    (alphaSlider.knobPercent * 255).toFixed(0)  + ')';

alphaSlider.opacity = alphaSlider.knobPercent;

alphaSlider.draw();
redSlider.draw();
greenSlider.draw();
blueSlider.draw();
```

应用程序创建了 4 个滑动条控件，并且向每个控件都注册了一个用于监听数值改变的事件监听器。一旦滑块的数值发生变化，它就会在所有已注册的监听器身上回调 addChangeListener() 方法。应用程序所注册的这 4 个事件处理器，都会改变色标以及滑动条本身的颜色。

程序清单 10-9 列出了 COREHTML5.Slider 对象的代码。在阅读这段代码时，要着重研究事件处理器的实现方式，以及控件对数值改变事件的支持机制。该控件在其数值发生改变时会触发相关的事件，这正是它与进度条及圆角矩形控件的区别。

程序清单 10-9　滑动条控件的实现代码

```
var COREHTML5 = COREHTML5 || {};

// Constructor.......................................................

COREHTML5.Slider = function(strokeStyle, fillStyle,
                            knobPercent, hpercent, vpercent) {
    this.trough = new COREHTML5.RoundedRectangle(strokeStyle, fillStyle,
                        hpercent || 95,  // Horizontal size percent
                        vpercent || 55); // Vertical size percent

    this.knobPercent = knobPercent || 0;
    this.strokeStyle = strokeStyle ? strokeStyle : 'gray';
    this.fillStyle = fillStyle ? fillStyle : 'skyblue';

    this.SHADOW_COLOR = 'rgba(100,100,100,0.8)';
    this.SHADOW_OFFSET_X = 3;
```

```
    this.SHADOW_OFFSET_Y = 3;

    this.HORIZONTAL_MARGIN = 2 * this.SHADOW_OFFSET_X;
    this.VERTICAL_MARGIN   = 2 * this.SHADOW_OFFSET_Y;

    this.KNOB_SHADOW_COLOR    = 'yellow';
    this.KNOB_SHADOW_OFFSET_X = 1;
    this.KNOB_SHADOW_OFFSET_Y = 1;
    this.KNOB_SHADOW_BLUR     = 0;

    this.KNOB_FILL_STYLE   = 'rgba(255,255,255,0.45)';
    this.KNOB_STROKE_STYLE = 'rgba(0,0,150,0.45)';

    this.context = document.createElement('canvas').getContext('2d');
    this.changeEventListeners = [];

    this.createDOMElement();
    this.addMouseHandlers();

    return this;
}

// Prototype.........................................................

COREHTML5.Slider.prototype = {

    // General functions to override..................................

    createDOMElement: function () {
        this.domElement = document.createElement('div');
        this.domElement.appendChild(this.context.canvas);
    },

    appendTo: function (elementName) {
        document.getElementById(elementName).
            appendChild(this.domElement);

        this.setCanvasSize();
        this.resize();
    },

    setCanvasSize: function () {
        var domElementParent = this.domElement.parentNode;

        this.context.canvas.width = domElementParent.offsetWidth;
        this.context.canvas.height = domElementParent.offsetHeight;
    },

    resize: function() {
        this.cornerRadius = (this.context.canvas.height/2 -
                             2*this.VERTICAL_MARGIN)/2;

        this.top    = this.HORIZONTAL_MARGIN;
        this.left   = this.VERTICAL_MARGIN;

        this.right  = this.left + this.context.canvas.width -
                      2*this.HORIZONTAL_MARGIN;

        this.bottom = this.top + this.context.canvas.height -
                      2*this.VERTICAL_MARGIN;

        this.trough.resize(this.context.canvas.width,
```

```
                        this.context.canvas.height);

   this.knobRadius =  this.context.canvas.height/2 -
                      this.context.lineWidth*2;
},

// Event handlers.................................................

addMouseHandlers: function() {
   var slider = this; // Let DIV's event handlers access this object

   this.domElement.onmouseover = function(e) {
      slider.context.canvas.style.cursor = 'crosshair';
   };

   this.domElement.onmousedown = function(e) {
      var mouse = slider.windowToCanvas(e.clientX, e.clientY);

      e.preventDefault();

      if (slider.mouseInTrough(mouse) ||
          slider.mouseInKnob(mouse)) {

         slider.knobPercent = slider.knobPositionToPercent(mouse.x);
         slider.fireChangeEvent(e);
         slider.erase();
         slider.draw();
         slider.dragging = true;

      }
   };

   window.addEventListener('mousemove', function(e) {
      var mouse = null,
          percent = null;

      e.preventDefault();

      if (slider.dragging) {
         mouse = slider.windowToCanvas(e.clientX, e.clientY);
         percent = slider.knobPositionToPercent(mouse.x);

         if (percent >= 0 && percent <= 1.0) {
            slider.fireChangeEvent(e);
            slider.erase();
            slider.draw(percent);
         }
      }
   }, false);

   window.addEventListener('mouseup', function(e) {
      var mouse = null;

      e.preventDefault();

      if (slider.dragging) {
         slider.fireChangeEvent(e);
         slider.dragging = false;
      }
   }, false);
},
```

```
// Change events.................................................

fireChangeEvent: function(e) {
   for (var i=0; i < this.changeEventListeners.length; ++i) {
      this.changeEventListeners[i](e);
   }
},

addChangeListener: function (listenerFunction) {
   this.changeEventListeners.push(listenerFunction);
},

// Utility functions.............................................

mouseInKnob: function(mouse) {
   var position = this.knobPercentToPosition(this.knobPercent);
   this.context.beginPath();
   this.context.arc(position, this.context.canvas.height/2,
                    this.knobRadius, 0, Math.PI*2);

   return this.context.isPointInPath(mouse.x, mouse.y);
},

mouseInTrough: function(mouse) {
   this.context.beginPath();
   this.context.rect(this.left, 0,
      this.right - this.left, this.bottom);

   return this.context.isPointInPath(mouse.x, mouse.y);
},

windowToCanvas: function(x, y) {
   var bbox = this.context.canvas.getBoundingClientRect();

   return {
      x: x - bbox.left * (this.context.canvas.width  / bbox.width),
      y: y - bbox.top  * (this.context.canvas.height / bbox.height)
   };
},

knobPositionToPercent: function(position) {
   var troughWidth = this.right - this.left - 2*this.knobRadius;
   return (position - this.left - this.knobRadius)/ troughWidth;
},

knobPercentToPosition: function(percent) {
   if (percent > 1) percent = 1;
   if (percent < 0) percent = 0;
   var troughWidth = this.right - this.left - 2*this.knobRadius;
   return percent * troughWidth + this.left + this.knobRadius;
},

// Drawing functions.............................................

fillKnob: function (position) {
   this.context.save();

   this.context.shadowColor   = this.KNOB_SHADOW_COLOR;
   this.context.shadowOffsetX = this.KNOB_SHADOW_OFFSET_X;
   this.context.shadowOffsetY = this.KNOB_SHADOW_OFFSET_Y;
   this.context.shadowBlur    = this.KNOB_SHADOW_BLUR;
```

```
        this.context.beginPath();

        this.context.arc(position,
                         this.top + ((this.bottom - this.top) / 2),
                         this.knobRadius, 0, Math.PI*2, false);

        this.context.clip();

        this.context.fillStyle = this.KNOB_FILL_STYLE;
        this.context.fill();
        this.context.restore();
    },

    strokeKnob: function () {
        this.context.save();
        this.context.lineWidth = 1;
        this.context.strokeStyle = this.KNOB_STROKE_STYLE;
        this.context.stroke();
        this.context.restore();
    },

    drawKnob: function (percent) {
        if (percent < 0) percent = 0;
        if (percent > 1) percent = 1;

        this.knobPercent = percent;
        this.fillKnob(this.knobPercentToPosition(percent));
        this.strokeKnob();
    },

    drawTrough: function () {
        this.context.save();
        this.trough.fillStyle = this.fillStyle;
        this.trough.strokeStyle = this.strokeStyle;
        this.trough.draw(this.context);
        this.context.restore();
    },

    draw: function (percent) {
        this.context.globalAlpha = this.opacity;

        if (percent === undefined)  {
            percent = this.knobPercent;
        }

        this.drawTrough();
        this.drawKnob(percent);
    },

    erase: function() {
        this.context.clearRect(
            this.left - this.knobRadius, 0 - this.knobRadius,
            this.context.canvas.width  + 4*this.knobRadius,
            this.context.canvas.height + 3*this.knobRadius);
    }
};
```

与进度条一样，滑动条控件时也要创建并使用 COREHTML5.RoundedRectangle 对象。

当滑动条控件检测到鼠标按下事件时，它就会将存储在事件对象中的窗口坐标转换为 canvas 坐标，并判断光标是否在滑动条的滑槽或滑块（knob）之上。如果在的话，那么事件处理器就会

调整滑块的位置，使之符合鼠标事件对象所提供的 X 坐标。然后，事件处理器对象会触发数值改变事件（change event），重绘滑动条控件，并且设置一个标志位，用来表示用户正在拖动控件中的滑块。

在用户持续拖动鼠标的过程中，滑动条控件的鼠标事件处理器会不停地触发数值改变事件，并重绘滑动条控件。如果滑动条控件检测到了鼠标松开事件（mouse up event），那么它会最后一次触发数值改变事件，然后将 dragging 属性设为 false。

COREHTML5.Slider 对象的构造器创建了一个名为 changeEventListeners 的空数组。滑动条控件可通过 addChangeListener() 与 fireChangeEvent() 方法操作此数组的内容。前一个方法可以向滑动条控件中添加监听数值改变事件所用的函数，而后一个方法则用于将滑动条控件中所发生的数值改变事件通知给所有的监听器。

滑动条控件中另一个有趣的事情是它所实现的发光效果。如图 10-9 所示，滑动条的滑块看起来正在发光，并照亮了其下方的滑槽。为了制作滑动条控件的发光效果，我们用半透明的白色将滑块填充，同时绘制黄色阴影。具体情况请参照程序清单 10-9 中的 fillKnob() 方法。

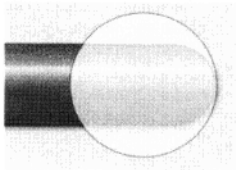


图 10-9　通过绘制阴影来制作发光效果

10.4　图像查看器控件

现在读者已经很好地掌握了如何实现基于 Canvas 的控件，在本章的最后，我们来看看怎样制作图 10-10 所示的图像查看器控件。它既用到了本章所学的知识，也用到了本书其他章节所讲的诸如绘制、阴影及图像操作等技巧。

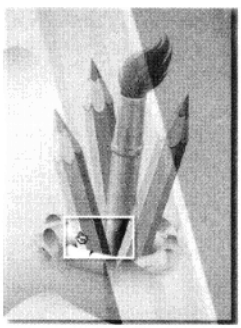


图 10-10　图像查看器控件

图像查看器控件会把一张非常大的图像按比例缩小后显示出来，然后提供一个可以拖动的视窗，如图 10-10 所示。如图 10-11 所示，当用户拖动应用程序中的视窗时，图像查看器控件就会把视窗中所对应的这部分图像放大，并绘制到与该控件相关的 canvas 中。图 10-11 中的应用程序演示了该控件的操作方式。

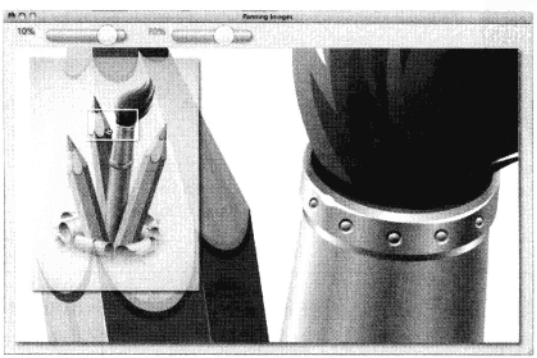


图 10-11　使用图像查看器控件查看大幅图像

在创建图像查看器控件时，可以按照下列方式指定与之关联的 canvas 及图像：

```
var pan = new COREHTML5.Pan(context.canvas, image);
```

如图 10-12 所示，用户可以拖动视窗来查看图像的不同部分。

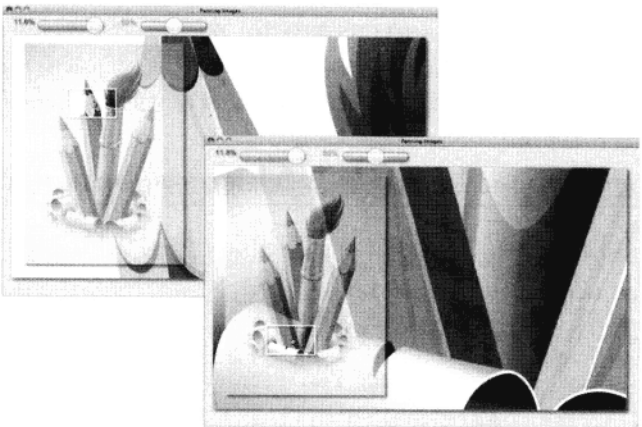


图 10-12　拖动图像视窗来查看图像的各个部分

程序顶部的滑动条可以调整图像查看器的大小与透明度，如图 10-13 所示。虽说如此，但是这些滑动条并不是图像查看器的一部分。图像查看器控件本身并不知道它们的存在，是应用程序将图像查看器与滑动条连接到了一起。

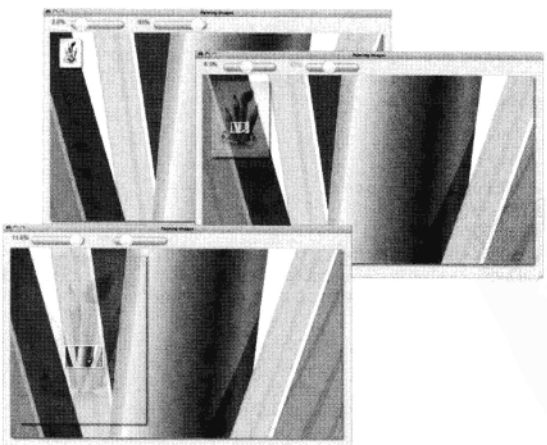


图 10-13　控制内嵌 canvas 的大小与透明度

图 10-11 所示应用程序的 HTML 代码列在了程序清单 10-10 之中。

程序清单 10-10 里面的 HTML 代码先创建了两个 DIV 元素，一个用于放置程序顶部的滑动条，另一个则用于容纳显示图像所用的 canvas 元素。接下来，我们将创建圆角矩形、滑动条与

图像查看器控件所用的 JavaScript 引入 HTML 之中。最后，再让 HTML 代码把程序本身所用的 JavaScript 文件包含进来即可。程序清单 10-11 列出了后者的代码。

这段 HTML 代码还创建了两个用于显示滑动条数值的 span 元素，应用程序的 JavaScript 代码负责设定这两个元素的值。

最后要注意的是，在 HTML 代码中的 CSS 部分中，声明了一个叫做 pan 的 class，然而，仔细研读这段 HTML 代码，我们就会发现并没有哪个元素声明过值为 pan 的 class 属性。其实该属性是留给应用程序的 JavaScript 代码来用的。在程序运行时，它的 JavaScript 代码会将图像查看器控件的 class 属性设置为 pan。

程序清单 10-10　图像查看器控件演示程序所用的 HTML 代码

```
<!DOCTYPE html>
<html>
    <head>
        <title> Panning Images</title>

        <style>
            body {
                background: rgba(100, 145, 250, 0.3);
            }

            #canvas {
                position: absolute;
                left: 0px;
                top: 50px;
                margin-left: 20px;
                margin-right: 0;
                margin-bottom: 20px;
                padding: 0;
                -webkit-box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                -moz-box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                border: 1px solid rgba(100,140,130,0.5);
                cursor: crosshair;
            }

            .pan {
                position: absolute;
                left: 50px;
                top: 70px;
                -webkit-box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                -moz-box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                box-shadow: rgba(60,60,70,0.7) 5px 5px 7px;
                cursor: pointer;
            }

            #sizeSliderDiv {
                position: absolute;
                left: 20px;
                top: -5px;
                margin-left: 10px;
                display: inline;
                width: 175px;
                height: 45px;
            }

            #alphaSliderDiv {
```

```
                position: absolute;
                left: 270px;
                top: -5px;
                margin-left: 10px;
                display: inline;
                width: 175px;
                height: 45px;
            }

            #controls {
                position: absolute;
                left: 10px;
                margin-left: 35px;
                margin-bottom: 25px;
            }

            #alphaSpan {
                position: absolute;
                left: 240px;
                vertical-align: center;
                color: rgb(80,100,190);
                font: 18px Arial;
                text-shadow: 2px 2px 4px rgba(100,140,250,0.8);
            }

            #sizeSpan {
                position: absolute;
                left: -20px;
                vertical-align: center;
                color: rgb(80,100,190);
                font: 18px Arial;
                text-shadow: 2px 2px 4px rgba(100,140,250,0.8);
            }
        </style>
    </head>

    <body id='body'>

        <div id='controls'>
            <span id='sizeSpan'>0</span>
            <div id='alphaSliderDiv'></div>

            <span id='alphaSpan'>0</span>
            <div id='sizeSliderDiv'></div>
        </div>

        <canvas id='canvas' width='1000' height='600'>
            Canvas not supported
        </canvas>

        <script src='roundedRectangle.js'></script>
        <script src='slider.js'></script>
        <script src='pan.js'></script>
        <script src='example.js'></script>
    </body>
</html>
```

程序清单 10-11　图像查看器控件演示程序所用的 JavaScript 代码

```
var context = document.getElementById('canvas').getContext('2d'),
    image = new Image(),
```

```
    alphaSpan = document.getElementById('alphaSpan'),
    sizeSpan = document.getElementById('sizeSpan'),

    sizeSlider = new COREHTML5.Slider('blue', 'cornflowerblue',
                                      0.85, // Knob percent
                                      90,   // Take up % of width
                                      50),  // Take up % of height

    alphaSlider = new COREHTML5.Slider('blue', 'cornflowerblue',
                                       0.50,// Knob percent
                                       90,  // Take up % of width
                                       50), // Take up % of height

    pan = new COREHTML5.Pan(context.canvas, image),
    e = pan.domElement,

    ALPHA_MAX = 1.0,
    SIZE_MAX = 12;

// Event handlers.....................................................
sizeSlider.addChangeListener(function (e) {
    var size = (parseFloat(sizeSlider.knobPercent) * 12);
    size = size < 2 ? 2 : size;
    sizeSpan.innerHTML = size.toFixed(1) + '%';

    pan.imageContext.setTransform(1,0,0,1,0,0); // Identity matrix
    pan.viewportPercent = size;

    pan.erase();
    pan.initialize();
    pan.draw();
});
alphaSlider.addChangeListener(function (e) {
    alphaSpan.innerHTML =
        parseFloat(alphaSlider.knobPercent * 100).toFixed(0) + '%';
    alphaSpan.style.opacity = parseFloat(alphaSlider.knobPercent);
    pan.panCanvasAlpha = alphaSlider.knobPercent;
    pan.erase();
    pan.draw();
});
// Initialization....................................................
image.src = 'pencilsAndBrush.jpg';
document.getElementById('body').appendChild(e);
e.className = 'pan';

alphaSlider.appendTo('alphaSliderDiv');
sizeSlider.appendTo('sizeSliderDiv');

pan.viewportPercent = sizeSlider.knobPercent * SIZE_MAX;
pan.panCanvasAlpha = alphaSlider.knobPercent * ALPHA_MAX;

sizeSpan.innerHTML = pan.viewportPercent.toFixed(0) + '%';
alphaSpan.innerHTML = (pan.panCanvasAlpha * 100).toFixed(0) + '%';

alphaSlider.draw();
sizeSlider.draw();
```

应用程序的 JavaScript 代码创建了两个滑动条控件，它们的描边属性均为 blue，填充样式均为 cornflowerblue。这两个控件的滑槽占据外围元素 90% 的宽度与 50% 的高度。应用程序启动时，把用于调整图像查看器尺寸的滑块位置设为滑动条全长的 85%（意思就是让滑块位于滑动

条中代表最小值的左端点与代表最大值的右端点之间，且与左端点之间的距离为滑动条全长的85%），而把用于调整控件 alpha 值的滑块位置设为滑动条全长的 50%。

应用程序向每个滑动条控件添加用于监听其数值改变的监听器对象。其中，监听 sizeSlider 控件的那个监听器，会根据滑动条的读数来相应地调整图像查看器控件的大小。该监听器对象还会设置 sizeSpan 对象的 innerHTML 属性，以反映滑动条的值。最后，监听器对象将现有的图像擦除，初始化并重新绘制图像。

监听 alphaSlider 控件的监听器则会根据滑动条的读数来调整图像查看器控件的透明度，并且设置 alphaSpan 对象的 innerHTML 属性，使之与滑动条控件的值相符。

程序清单 10-12 列出了 COREHTML5.Pan 对象的代码。

程序清单 10-12　图像查看器控件的代码

```
var COREHTML5 = COREHTML5 || { };

// Constructor.......................................................

COREHTML5.Pan = function(imageCanvas, image,
                         viewportPercent, panCanvasAlpha) {
    var self = this;

    // Store arguments in member variables

    this.imageCanvas = imageCanvas;
    this.image = image;
    this.viewportPercent = viewportPercent || 10;
    this.panCanvasAlpha = panCanvasAlpha || 0.5;

    // Get a reference to the image canvas's context
    // and create the pan canvas and the DOM element.
    // Put the pan canvas in the DOM element.

    this.imageContext = imageCanvas.getContext('2d');
    this.panCanvas = document.createElement('canvas');
    this.panContext = this.panCanvas.getContext('2d');

    this.domElement = document.createElement('div');
    this.domElement.appendChild(this.panCanvas);

    // If the image is not loaded, initialize when the image loads;
    // otherwise, initialize now.

    if (image.width == 0 || image.height == 0) { // Image not loaded
        image.onload = function(e) {
            self.initialize();
        };
    }
    else {
        this.initialize();
    }
    return this;
};

// Prototype........................................................

COREHTML5.Pan.prototype = {
    initialize: function () {
        var width = this.image.width * (this.viewportPercent/100),
```

```
        height = this.image.height * (this.viewportPercent/100);

    this.addEventHandlers();
    this.setupViewport (width, height);
    this.setupDOMElement(width, height);
    this.setupPanCanvas (width, height);
    this.draw();
},

setupPanCanvas: function (w, h){
    this.panCanvas.width = w;
    this.panCanvas.height = h;
},

setupDOMElement: function (w, h) {
    this.domElement.style.width = w + 'px';
    this.domElement.style.height = h + 'px';
    this.domElement.className = 'pan';
},

setupViewport: function (w, h) {
    this.viewportLocation = { x: 0, y: 0 };
    this.viewportSize = { width: 50, height: 50 };
    this.viewportLastLocation = { x: 0, y: 0 };

    this.viewportSize.width = this.imageCanvas.width *
                              this.viewportPercent/100;

    this.viewportSize.height = this.imageCanvas.height *
                               this.viewportPercent/100;
},

moveViewport: function(mouse, offset) {
    this.viewportLocation.x = mouse.x - offset.x;
    this.viewportLocation.y = mouse.y - offset.y;

    var delta = {
        x: this.viewportLastLocation.x - this.viewportLocation.x,
        y: this.viewportLastLocation.y - this.viewportLocation.y
    };

    this.imageContext.translate(
        delta.x * (this.image.width / this.panCanvas.width),
        delta.y * (this.image.height / this.panCanvas.height));

    this.viewportLastLocation.x = this.viewportLocation.x;
    this.viewportLastLocation.y = this.viewportLocation.y;
},

isPointInViewport: function (x, y) {
    this.panContext.beginPath();
    this.panContext.rect(this.viewportLocation.x,
                         this.viewportLocation.y,
                         this.viewportSize.width,
                         this.viewportSize.height);

    return this.panContext.isPointInPath(x, y);
},

addEventHandlers: function() {
    var pan = this;
```

```
        pan.domElement.onmousedown = function(e) {
            var mouse = pan.windowToCanvas(e.clientX, e.clientY),
                offset = null;

            e.preventDefault();

            if (pan.isPointInViewport(mouse.x, mouse.y)) {
                offset = { x: mouse.x - pan.viewportLocation.x,
                           y: mouse.y - pan.viewportLocation.y };

                pan.panCanvas.onmousemove = function(e) {
                    pan.erase();
                    pan.moveViewport(
                        pan.windowToCanvas(e.clientX, e.clientY), offset);

                    pan.draw();
                };

                pan.panCanvas.onmouseup = function(e) {
                    pan.panCanvas.onmousemove = undefined;
                    pan.panCanvas.onmouseup = undefined;
                };
            }
        };
    },

    erase: function() {
        this.panContext.clearRect(0, 0,
                                  this.panContext.canvas.width,
                                  this.panContext.canvas.height);
    },

    drawPanCanvas: function(alpha) {
        this.panContext.save();
        this.panContext.globalAlpha = alpha;
        this.panContext.drawImage(this.image,
                                  0, 0,
                                  this.image.width,
                                  this.image.height,
                                  0, 0,
                                  this.panCanvas.width,
                                  this.panCanvas.height);
        this.panContext.restore();
    },

    drawImageCanvas: function() {
        this.imageContext.drawImage(this.image,
                                    0, 0,
                                    this.image.width,
                                    this.image.height);
    },

    drawViewport: function () {
        this.panContext.shadowColor = 'rgba(0,0,0,0.4)';
        this.panContext.shadowOffsetX = 2;
        this.panContext.shadowOffsetY = 2;
        this.panContext.shadowBlur = 3;

        this.panContext.lineWidth = 3;
        this.panContext.strokeStyle = 'white';
        this.panContext.strokeRect(this.viewportLocation.x,
```

```
                                    this.viewportLocation.y,
                                    this.viewportSize.width,
                                    this.viewportSize.height);
    },

    clipToViewport: function() {
        this.panContext.beginPath();
        this.panContext.rect(this.viewportLocation.x,
                             this.viewportLocation.y,
                             this.viewportSize.width,
                             this.viewportSize.height);
        this.panContext.clip();
    },

    draw: function() {
        this.drawImageCanvas();
        this.drawPanCanvas(this.panCanvasAlpha);

        this.panContext.save();
        this.clipToViewport();
        this.drawPanCanvas(1.0);
        this.panContext.restore();

        this.drawViewport();
    },

    windowToCanvas: function(x, y) {
        var bbox = this.panCanvas.getBoundingClientRect();

        return {
            x: x - bbox.left * (this.panCanvas.width / bbox.width),
            y: y - bbox.top * (this.panCanvas.height / bbox.height)
        };
    },
};
```

10.5 总结

尽管标准的 HTML 控件对于许多应用程序来说，已经足够用了，不过我们仍然有很多充分的理由来自己实现基于 Canvas 的控件。比方说，我们要用到一种 HTML 标准并未提供的控件，或者想要让控件在所有浏览器中都具备相同的外观。

读者在本章中学习了如何实现基于 Canvas 的控件。在实现过程中，我们看到，先要创建 canvas 元素，然后将它包裹在一个 DIV 元素之中，最后再把这个 DIV 元素暴露给开发者。由于开发者能够获取此 DIV 元素，所以就可以把它加入 DOM 树状结构里的任何元素之中。

我们还学会了如何实现进度条与滑动条这样的复合控件，也就是那种使用其他控件制作出来的控件。最后，大家看到了怎样把事件处理机制集成到控件之中，以及如何将事件通知给已经注册的事件监听器对象。

在下一章，咱们将看看如何实现基于 Canvas 的移动网络应用程序。

第11章
移动平台开发

20 世纪 90 年代，Java 语言大受欢迎，因为开发者只需要编写一次代码就可以让应用程序在多个操作系统上运行。到了本世纪初，HTML5 标准也在移动开发领域的前沿做着相同的事情——让开发者编写的程序能够运行在桌面操作系统及诸多移动操作系统之上。

在本章中，读者将学到如何使基于 Canvas 的应用程序运行在移动设备之上。我们将着重介绍怎样让本书中的放大镜程序及绘图程序运行在 iPad 之上，使它们看上去与原生应用（native application）程序毫无二致。

4.10 节曾经讲过放大镜程序，图 11-1 演示了该程序在 Mac OS X 系统与 iPad 的 iOS5 系统中的运行情况。从视觉效果上说，除了标题栏之外，此程序在两个操作系统中的样子是相同的。由于应用程序同时支持对鼠标与触摸事件的处理，所以同一套代码可以运行在这两种操作系统之上。

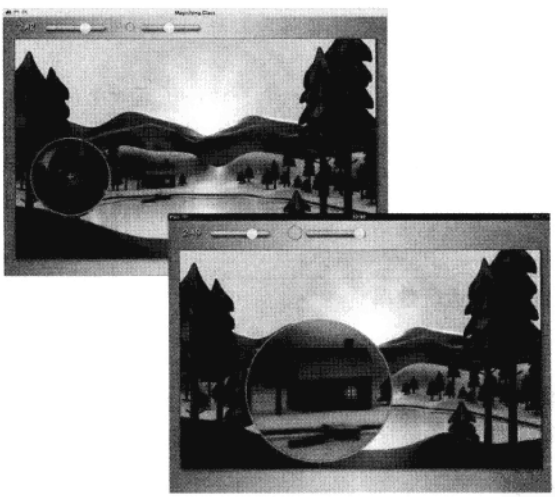


图 11-1　同一个应用程序在桌面操作系统（顶部截图）与 iPad 系统（底部截图）上的运行情况

为了演示如何让基于 Canvas 的应用程序可以运行在移动设备之上，本章将向读者讲述下列技巧：

- 通过指定名为 viewport 的 metatag [⊖]，我们可以根据特定的设备与显示方向来优化应用程序的视窗大小。

⊖ metatag，即meta element，用于在HTML语言中设置metadata（元数据），以<meta ...>的形式出现。中文可称为“元标签”，以这种形式所定义的元数据也叫“后设资料”、“后设数据”。详情参见：https://en.wikipedia.org/wiki/Meta_element。——译者注

- 使用 CSS 媒体特征查询（media query）技术，用适合移动设备的界面风格来显示应用程序。
- 以 JavaScript 语言编写监听器，用来侦测媒体特征的变更，以便在设备显示方向发生改变时，相应地调整应用程序的显示方式。
- 处理触摸事件。
- 禁用惯性滚动（inertial scrolling）功能。
- 阻止用户缩放应用程序，停用“DIV 闪烁”（DIV flashing）等特性。
- 实现“手指缩放”（pinch and zoom）功能。
- 在平板电脑上实现基于 Canvas 的虚拟键盘控件。

想要让基于 Canvas 的应用程序在 iOS5 系统中运行时看起来和原生应用程序一模一样，你还要学习以下知识：

- 创建应用程序图标及启动图像。
- 使用媒体特征查询技术来选择适当的应用程序图标及启动图像。
- 令 HTML5 应用程序直接以全屏模式运行，不显示任何浏览器饰件。
- 设置状态栏的背景色。

提示：HTML5 程序与原生应用程序的对比

在本书付印之时，很多人都在争论：HTML5 程序与原生应用程序相比，到底哪一个能在移动设备上胜出。读者通过学习本章的知识就能明白，我们可以把 HTML5 应用程序实现得同 iPad 上面的原生应用程序一模一样。

这种程序界面上的实现问题，只是争论的一个层面，还有一个问题则是如何使用诸如陀螺仪（gyroscope）与 GPS 定位等底层功能。HTML5 规范一直在这方面不断地更新，以求将这些功能都囊括进来。此外，开发者也可以通过类似 PhoneGap[⊖]这样的开发框架来使用这些功能。

总之，如果打算让应用程序支持多种机型及多个操作系统的话，那么与原生应用程序相比，使用 HTML5 技术可以极大地降低软件开发成本。

提示：Canvas 在移动设备上的运行效率

当笔者写作本书大部分内容之时，Canvas 在移动设备上的运行效率让人颇为失望。尽管在桌面操作系统之中，基于 Canvas 的动画可以流畅地播放，但是在许多移动设备上，却显得不够连贯。然而，在本书快要写完的时候，使用硬件加速来渲染 Canvas 的 iOS5 系统正式推出了。于是，在世界上最为流行的 iPhone 移动设备之中，HTML5 游戏的境遇彻底改变了。硬件加速功能极大地提升了 Canvas 的性能，因此开发者有可能用它制作出流畅的动画与电子游戏。

11.1 移动设备的视窗

由于移动设备的屏幕通常比桌面电脑的屏幕小，所以在分辨率相同的情况下，为了能让用户看清楚内容，移动浏览器只能把网页的某一部分显示出来，如图 11-2 所示。

⊖ 一套开源的移动平台开发框架，让开发者使用JavaScript、HTML5、CSS3等技术制作出能够运行于各个平台的程序，而不再需要针对每个特定的平台来编程。详情参见：https://en.wikipedia.org/wiki/PhoneGap。——译者注

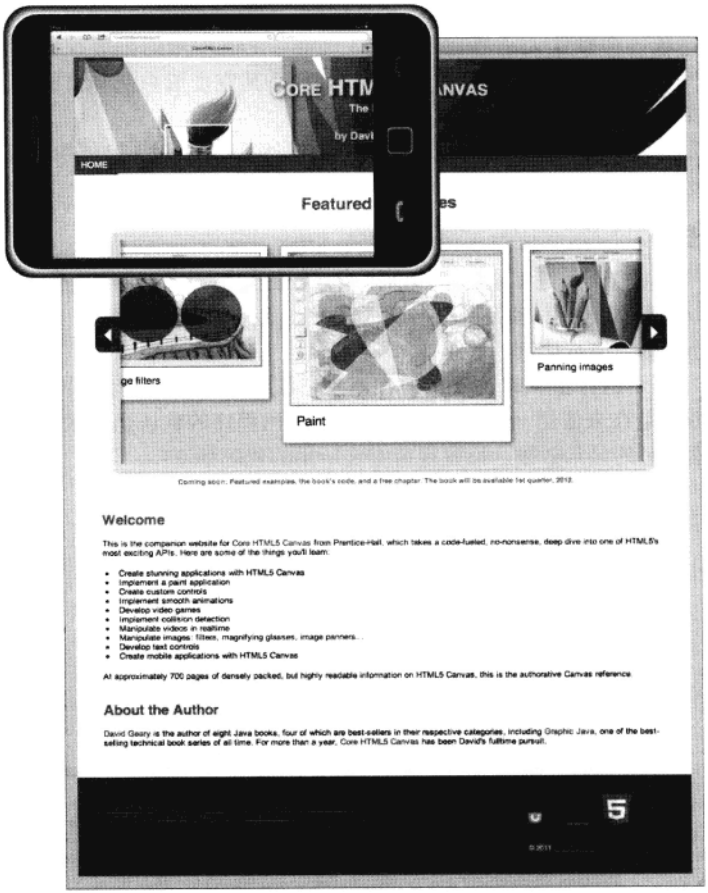


图 11-2 移动浏览器的视窗比桌面浏览器小

不过大部分移动浏览器一开始显示网页的时候，都不会像图 11-2 那样做。因为如果一开始就那样显示的话，那么几乎所有用户都会立刻将网页缩小，以使网页宽度符合屏幕宽度。所以大部分移动浏览器在刚开始显示网页的时候，都会将网页自动缩小，使其宽度刚好与屏幕相符，如图 11-3 所示。

图 11-3 大多数移动浏览器在刚加载完网页时的默认显示方式

包括 iPad 等平板电脑浏览器在内的移动浏览器是如何做到让网页一加载进来就能自动符合屏幕宽度的呢？理解这一点很重要。

移动浏览器一开始会把网页绘制到一个叫做“排版视窗”（layout viewport）的离屏视窗之中，以便页面内的 CSS 排版指令能够等比例地产生与桌面浏览器相似的显示效果。实际上，可以将图 11-2 之中的那幅网页理解

为“排版视窗”。移动设备的屏幕则被称为“可见视窗”（visible viewport）。

每个设备的排版视窗默认大小都是固定值。例如，iPad 所用的排版视窗为 980 像素宽，而 Android 设备的排版视窗则为 800 像素宽。但无论如何，其宽度值总是与桌面操作系统中一个典型的应用程序窗口相近。

网络浏览器先把页面渲染到离屏的排版视窗之中，然后再将该视窗中的内容复制到设备的可见视窗之中，在复制的同时还会进行缩放。

viewport 元标签

移动设备的可见视窗大小是固定的，然而排版视窗的大小却是可调整的。借助 viewport 元标签，我们不仅可以调整浏览器的排版视窗，还可以调整该视窗的各种属性，诸如用户是否可以缩放视窗，以及最小与最大缩放比例，等等。

图 11-4 展示了一个运行在 iPad 之上的应用程序。该程序只有一个宽 500 像素、高 50 像素的 DIV 元素。顶部截图是在未使用 viewport 元标签时，应用程序的默认运行状态。中间那张截图是当视窗为 500 像素宽时的显示效果。设置视窗宽度所用的代码如下：

```
<meta name='viewport' content='width=500'/>
```

底部截图显示的是当视窗宽度为 100 像素时应用程序的样子。

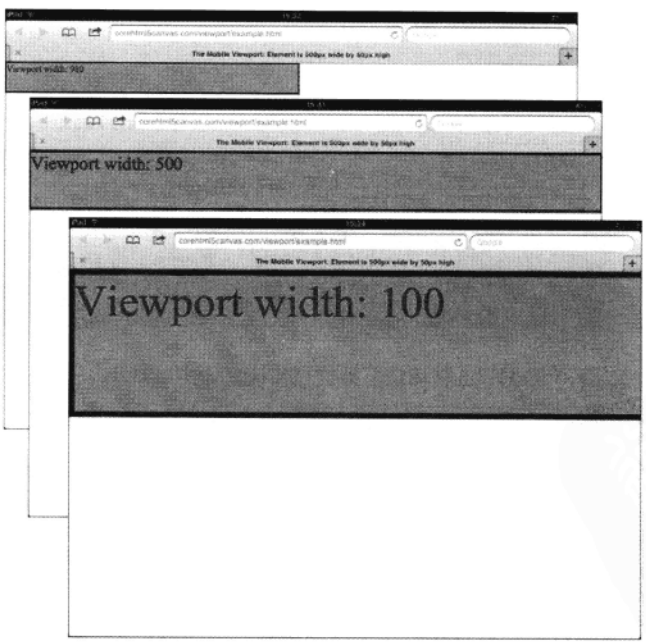


图 11-4　通过 viewport 元标签设置移动浏览器排版视窗的宽度

在图 11-4 顶部截图中，iPad 把网页渲染到宽 980 像素的排版视窗中，然后再对其缩放，使之符合可见视窗。在横屏模式（landscape mode）下，iPad 的宽度为 1040 像素，这与排版视窗的宽度（980 像素）相近，这两个视窗的尺寸大致相同。由于可见视窗与排版视窗几乎一样大，所以

占据排版视窗大约一半宽度的那个 500 像素宽的 DIV 元素，在横屏模式下也占了 iPad 显示屏一半左右的宽度。另外两张截图所演示的情况则与上述效果不同。

当排版视窗的宽度变小之后，在应用程序将 DIV 元素从较小的排版视窗绘制到固定大小的可见视窗时，就会把它放大。由于 DIV 元素的宽度与排版视窗一样，都是 500 像素，所以在放大之后，DIV 元素就会将屏幕的水平方向全部占满。如果排版视窗是 100 像素宽，那么浏览器就会把 500 像素宽的 DIV 元素填到 100 像素宽的视窗中，再将仅有 100 像素宽的内容放大到 1040 像素宽，于是这个 DIV 元素就会被放得相当大。

修改了排版视窗的宽度之后，浏览器会自动决定垂直方向的缩放比例，以维持 DIV 元素的宽度与高度之比不变。虽然我们通常改变的是排版视窗的宽度，不过也可以改变它的高度，如此一来，浏览器就会自动设置水平方向的缩放比例，这样还是能保持宽度与高度之比不变。

程序清单 11-1 列出了图 11-4 所示应用程序的 HTML 代码。

应用程序使用 viewport 元标签将排版视窗的宽度设为 500 像素。在图 11-4 的三张截图中，这个宽度值与 DIV 元素里面的文本都各不相同。

下面列举几种 viewport 元标签的用法：

```
<meta name="viewport" content="width=480"/>
<meta name="viewport" content="width=device-width,
                               initial-scale=1.0, user-scalable=yes"/>
<meta name="viewport"
      content="width=device-width, initial-scale=1.0,
               maximum-scale=1.0, user-scalable=no"/>
```

上面第一种用法，是用 viewport 元标签将排版视窗的宽度设为 480 像素。如果你要在宽屏幕设备上显示很窄的网站，那么可以将排版视窗的宽度写为某个硬编码（hardcode）的数值。例如，某个为 iPhone 设计的网站只有 480 像素宽，我们现在要让其显示在配有 retina 显示屏[⊖]的 iPhone 之上，那么可能就要像 11.4 中间的那张截图一样，让应用程序横向填满手机屏幕才好。此时就得把排版视窗的宽度设置成 480 像素，不然的话，网页就会像顶部那张截图一样，只占据一半屏幕宽度。

程序清单 11-1　通过 viewport 元标签调整排版视窗的宽度

```
<!DOCTYPE html>
<html>
   <head>
      <title>
         The Mobile Viewport: Element is 500px wide by 50px high
      </title>
      <meta  name='viewport'  content='width=500'/>
      <style>
         body {
            margin: 0px;
            padding: 0px;
         }
         #box {
            background: goldenrod;
            border: 2px solid navy;
```

⊖ 一种具备超高像素密度的液晶屏，最初可以将960×640的分辨率压缩到iPhone的3.5英寸屏幕内。详情参见：https://zh.wikipedia.org/zh-cn/Retina显示屏。——译者注

```
            color: blue;
            width: 500px;
            height: 50px;
        }
    </style>
</head>

<body>
    <div  id='box'>Viewport width: 500</div>
</body>
</html>
```

第二种使用 viewport 元标签的办法，将排版视窗的宽度设为 device-width，也就是设备的物理宽度，这样的话，无论是竖屏还是横屏，这个宽度值都不会变。对于那些本来就打算运行在桌面浏览器中的网站来说，由于其宽度通常与大多数手机一样（甚至更大），所以将排版视窗的宽度设置为 device-width 是个不错的办法。第二种用法也将初始缩放比例设为 1.0，并且允许用户缩放应用程序。

在 viewport 元标签的第三种用法中，排版视窗与初始缩放比例也被分别设置为 device-width 与 1.0，此外，我们还把最大缩放比例设置为 1.0，并禁止用户缩放应用程序。

表 11-1 列出了可以在 viewport 元标签的 content 属性中使用的属性值。

表 11-1　viewport 元标签的 content 属性值

content 属性	有效取值
width	1 ～ 10 000 之间的非负数，以像素为单位。也可以指定为代表设备物理宽度的 device-width。未知的关键字与取值都将视为 1 像素
height	1 ～ 10 000 之间的非负数，也可以设置表示设备物理高度的 device-height。与 width 属性一样，未知的关键字与取值都被视为 1 像素
initial-scale, minimum-scale, maximum-scale	0.1 ～ 10 之间的非负数。此外还可以设置为： • device-width 或 device-height，这两个值等同于 10。 • yes 与 no，分别表示 1 与 0.1
user-scalable	yes 与 no。yes 通常意味着用户可以用“手指缩放”操作以调整应用程序的缩放比例，而设置为 no 则禁止用户调整
target-densityDpi	70 ～ 400 之间的数值，用以表示 dpi（每英寸点数，dots per inch）。此外还可以设置为： • device-width 或 device-height，这两个值等同于 10。 • yes 和 no，分别表示 1 与 0.1。 虽说大多数浏览器也接受 target-densitydpi 这样的写法，不过规范中的第二个 d 是大写的

读者在很好地理解了移动设备的“视窗”（viewport）概念与它对应用程序排版的影响之后，我们来学习怎样根据不同的设备来选择与之对应的 CSS、程序启动图标与启动画面。

小技巧：将排版视窗的宽度设置为 device-width，而非具体数值

对于那些本来就是为桌面浏览器而设计的网络应用程序来说，应该将排版视窗的宽度设置为 device-width，而不要设置成具体数值。这样一来，离屏排版视窗的宽度就会与设备的宽度一样了。

小技巧：防止网页在 iPad 横屏时突然放大

如果在 viewport 元标签中把 initial-scale 设为 1.0，那么浏览器就会使网页在水平方向上与屏幕宽度相符，这通常是我们想要的效果。然而，移动 Safari 浏览器会照字面意思来对待“初始缩放比例”（initial-scale）这个属性，也就是说，它仅仅会在第一次加载页面时将缩放比例设置为该属性的值。如果将设备由竖屏模式旋转为横屏模式，那么移动 Safari 浏览器就会放大所显示页面的宽度，使之符合当前的设备宽度。在这种情况下，我们可以将 maximum-scale 属性设置为 1.0，这样就可以确保当用户把设备由竖屏模式旋转为横屏模式时，网页不会突然被撑大。

11.2 媒体特征查询技术

CSS3 标准新增的媒体特征查询（media query）技术，可以让应用程序根据运行它的设备来选择 CSS 及图像等资源。开发者也可以创建查询媒体特征所用的监听器，当浏览器侦测到设备显示方向等媒体特征发生改变时，会通知这些监听器。

11.2.1 媒体特征查询与 CSS

在 CSS 代码中，开发者可以使用 @media 来检测诸如显示方向或屏幕宽度等媒体特征（media feature），并根据检测结果为不同类型的设备声明不同的 CSS 规则。程序清单 11-2 展示了这种用法。

程序清单 11-2 通过媒体特征查询技术选择性地运用 CSS 规则

```
<!DOCTYPE html>
<html>
    <head>
        ...
        <style>
            ...

            @media all and (min-device-width: 481px) and
                (max-device-width: 1024px) and
                (orientation:portrait) {
                #controls {
                    ...
                }
                ...
            }

            @media all and (min-device-width: 481px)
                and (max-device-width: 1024px)
                and (orientation:landscape) {
                #controls {
                    ...
                }
                ...
            }
            ...
        </style>
    </head>

    <body>
    ...
    </body>
</html>
```

程序清单 11-2 的 CSS 规则代码通过 min-device-width、max-device-width 与 orientation 等媒体特征判断网页是运行在竖屏还是横屏的 iPad 之上，并据此运用相应的 CSS 效果。表 11-2 列出了全部媒体特征。

表 11-2 媒体特征

媒体特征	是否可以加 min-/max- 前缀	描述
width	是	视窗的宽度
height	是	视窗的高度
device-width	是	屏幕的宽度
device-height	是	屏幕的高度
orientation	否	可取 portrait 或 landscape
aspect-ratio	是	width 与 height 的比值
device-aspect-ratio	是	device-width 与 device-height 的比值
color	是	每个颜色分量所占的二进制位数
color-index	是	颜色查找表中的条目数量
monochrome	是	在单色缓冲区中，每个像素所占的二进制位数
resolution	是	设备的像素密度
scan	否	电视设备的扫描方式[⊖]
grid	否	如果值为 1，则表示 tty 终端等“基于网格的设备”（grid-based device）；若为 0，则代表像电脑显示器这样的“非网格设备”（non-grid device）

警告：不要使用媒体特征查询及 CSS 来修改 canvas 的大小

你有时可能想要使用媒体特征查询技术来检测设备的某些媒体特性，并据此修改 canvas 的大小。如果已经这样做了，那么请回想一下，本书 1.1.1 小节讲过，如果用 CSS 修改 canvas 的大小，那么修改的将是元素大小，而不是 canvas 绘图表面的大小。

如果想根据不同的设备特征来修改 canvas 的大小，那么应该用 JavaScript 语言实现一个监听器，让它监听媒体特征查询列表，等到其中的属性改变时，再据此调整 canvas 的大小。

11.2.2 用 JavaScript 程序应对媒体特征的变化

有时需要在程序运行时动态地处理媒体特征所发生的变化。例如，当显示方向等媒体特征改变时，我们想要修改一个或多个 canvas 的大小。在这种情况下，仅仅通过指定一些有条件的 CSS 规则来应对不同的屏幕方向是不够的，因为此时除了要用 CSS 修改 canvas 元素的大小之外，我们还必须以编程的方式修改其绘图表面的大小，使之与 canvas 元素的大小相符。本书 1.1.1 小节曾详述了在程序运行时修改 canvas 大小所带来的问题。

图 11-5 演示了运行在竖屏模式下的放大镜程序，当用户旋转设备方向时，应用程序的 canvas 也会随之改变其大小。

程序清单 11-3 列出了应用程序应对显示方向的改变所用的代码。首先检查 window.

⊖ 该值可以取progressive或interlace，分别代表逐行扫描及隔行扫描。详情参见：http://www.w3.org/TR/css3-mediaqueries/#scan。——译者注

matchMedia() 方法是否存在，如果存在，那么应用程序就创建一个媒体特征查询列表，该列表中只包含一个媒体查询条目，用于显示设备的方向。

接下来，应用程序实现一个监听器对象，用以检测媒体特征查询列表中的属性，当屏幕方向改变时，浏览器会将此事件通知给监听器。如果监听器侦测到媒体特征查询列表之中的属性已经改变，那么它就会调整应用程序之中的 canvas 以及放大镜的大小。

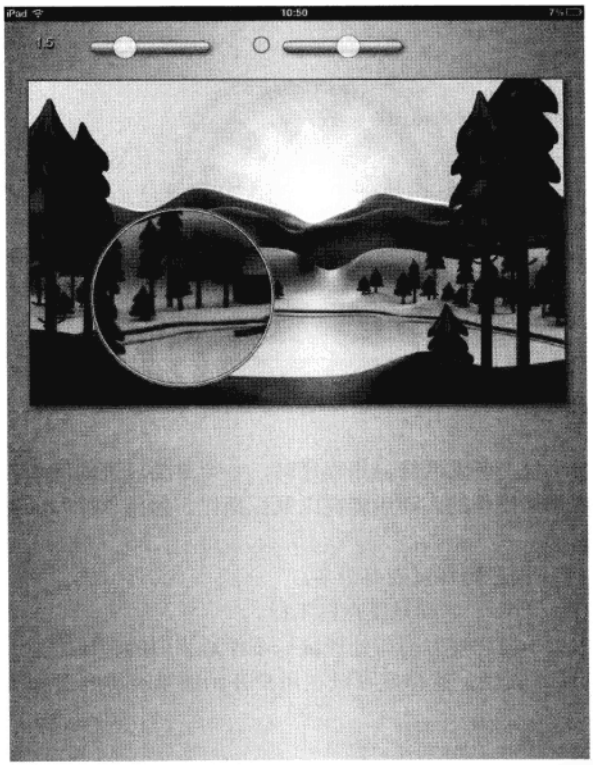


图 11-5　在 iPad 的屏幕显示方向改变时调整 canvas 的大小

程序清单 11-3　侦测媒体特征查询列表中的属性值改变所用的监听器对象

```
if (window.matchMedia && screen.width < 1024) {
    var m = window.matchMedia("(orientation:portrait)"),
        lw = 0,
        lh = 0,
        lr = 0;

    function listener (mql) {
        var cr = canvas.getBoundingClientRect();

        if (mql.matches) { // Portrait
            // Save landscape size to reset later
            lw = canvas.width;
            lh = canvas.height;
```

```
            lr = magnifyingGlassRadius;

            // Resize for portrait
            canvas.width = screen.width - 2*cr.left;
            canvas.height = canvas.width*canvasRatio;

            magnifyingGlassRadius *=
                (canvas.width + canvas.height) / (lw + lh);
        }
        else if (lw !== 0 && lh !== 0) { // Landscape
            // Reset landscape size
            canvas.width = lw;
            canvas.height = lh;

            magnifyingGlassRadius = lr;
        }

        // Setting canvas width and height resets and
        // erases the canvas, making a redraw necessary

        draw();
    }

    m.addListener(listener);
}
```

11.3 触摸事件

在移动平台与桌面平台上开发网络应用程序时，一个最为显著的区别就是，前者几乎都是通过手指或手写笔触摸屏幕来操作的，而后者则以鼠标操作。触摸事件与鼠标事件很相似，只有以下两个主要的不同点：

- 鼠标光标只有一个，而触摸点可能有很多。
- 鼠标光标可以悬停（hover），而触摸点则不行。

处理触摸事件的方式，在许多方面与处理鼠标事件是相似的。比方说，我们可以把某个函数赋给 canvas 的 ontouchstart 属性，这样就可以在用户开始触摸时得到通知了。增加监听器所用的代码如下：

```
canvas.ontouchstart = function (e) {
   alert('touch start');
};
```

你也可以调用 addEventListener() 方法来新增监听触摸开始事件所用的监听器：

```
canvas.addEventListener('touchstart', function (e) {
   alert('touch start');
});
```

表 11-3 列出了各种类型的触摸事件。

11.3.1 TouchEvent 对象

浏览器传递给触摸事件监听器的事件对象之中含有各种属性，表 11-4 列出了这些属性。

大部分情况下，只使用 touches 和 changedTouches 属性就够了。这两个属性值的类型都是 TouchList，该类型是一个由 Touch 对象组成的列表。接下来我们就看看这个列表。

表 11-3 触摸事件

事件类型	是否可以取消	是否走完整个冒泡式触发过程[⊖]	描述	浏览器对该事件的默认处理方式
touchstart	是	是	用户将某个触摸点置于触摸界面之上	未定义
touchmove	是	是	用户在触摸界面上移动触摸点	未定义
touchend	是	是	触摸点离开了触摸区域	根据具体情况而定，可能将其视为：mousemove、mousedown、mouseup、click
touchcancel	是	是	触摸点的触摸动作被打断，或是触摸点个数超出了设备所能处理的范围	未定义

表 11-4 TouchEvent 对象的属性

属性名	属性值的数据类型	描述
touches	TouchList	由正在界面上触摸的各个触摸点所组成的列表
changedTouches	TouchList	与上次触摸事件相比，发生改变的各个触摸点。对于 touchstart 事件来说，它表示那些刚刚被激活的触摸点；对于 touchmove 事件来说，表示那些位置发生了移动的触摸点；对于 touchend 与 touchcancel 事件来说，则表示那些不再停留于触摸界面之上的触摸点
targetTouches	TouchList	正在界面上触摸而且位于当前元素范围之内的那些触摸点。除非某个触摸点被拖到了元素范围之外，否则该列表就等同于 touches 列表
altKey、ctrlKey、metaKey、shiftKey	boolean	如果在触摸事件发生时，与之对应的按键（Alt、Ctrl、Meta 或 Shift）处于被按下的状态，那么其值就是 true。由于某些移动设备并没有物理键盘，所以这些属性的值可能是不确定的

11.3.2 TouchList 对象

TouchList 对象有两个属性：

- length

⊖ 原书写为Bubble，也就是Bubbling Phase，是一个与事件监听器触发顺序相关的术语。当某个元素与其外围元素均注册有同一种事件监听器时，如果发生了该类型的事件，那么就会有两种触发事件监听器的顺序。一种是捕获式触发（event capturing），也就先触发最外围元素的监听器，然后依次向内，直至发生事件的元素；另一种则是冒泡式触发（event bubbling），先触发最内层元素的事件监听器，然后依次向外，直至最外围的元素。W3C的事件模型则采用了折中的办法，先从外至内走完捕获式触发过程，然后再由内向外走完冒泡式触发过程（bubbling phase）。详情参见：https://en.wikipedia.org/wiki/DOM_events#Event_flow及http://www.quirksmode.org/js/events_order.html。——译者注

• Touch identifiedTouch(identifier)

给定某个 TouchList 对象，我们可以通过 length 属性获取列表中所含 Touch 对象的个数。其代码如下：

```
canvas.ontouchstart = function (e) {
   alert(e.touches.length + ' touches on the device');
};
```

我们可以像操作数组那样，访问 TouchList 之中的每一个 Touch 元素：

```
canvas.ontouchstart = function (e) {
   for(var i=0; i < e.touches.length;  ++i) {
      alert('Touch at: ' + e.touches[i].pageX + ',' +
                                e.touches[i].pageY);
   }
};
```

每个 Touch 对象都有一个独特的标识符，如果 TouchList 中存在具有特定标识符的 Touch 对象，那么 identifiedTouch() 方法就返回它。当你想要了解某一个触摸点是否参与了多个触摸事件时，该方法是很有用的。

11.3.3 Touch 对象

触摸事件监听器最终还是需要检查 Touch 对象本身的属性。表 11-5 列出了 Touch 对象的各个属性。

表 11-5 Touch 对象的属性

属性名	属性值的数据类型	描述
clientX	long	触摸点相对于视窗的 X 坐标。该值不将滚动条的宽度计算在内
clientY	long	触摸点相对于视窗的 Y 坐标。该值不将滚动条的高度计算在内
identifier	long	代表触摸点身份的独特标识符，同一个触摸点的身份标识符在不同的事件对象中保持不变
pageX	long	触摸点相对于视窗的 X 坐标。该值会将滚动条的宽度计算在内
pageY	long	触摸点相对于视窗的 Y 坐标。该值会将滚动条的高度计算在内
screenX	long	触摸点相对于屏幕的 X 坐标
screenY	long	触摸点相对于屏幕的 Y 坐标
target	EventTarget	触摸动作开始时，触摸点所在的元素。就算该点其后被拖出了初始元素，target 依然会指向一开始的那个元素

11.3.4 同时支持触摸事件与鼠标事件

虽说触摸事件与鼠标事件很相似，不过二者仍然需要分开处理。假如想让应用程序同时运行在桌面浏览器与手机浏览器之中，那么必须将触摸事件与鼠标事件等同对待，把事件处理逻辑封装在一系列方法之中，这些方法不需要知道待处理事件到底是鼠标事件还是触摸事件。程序清单 11-4 演示了这种事件处理策略。

程序清单 11-4 一个同时支持触摸事件与鼠标事件的事件处理模板

```
// Touch event handlers.....................................

canvas.ontouchstart = function (e) {
   e.preventDefault(e); // Optional
```

```
    mouseDownOrTouchStart(windowToCanvas(e.pageX, e.pageY));
};

canvas.ontouchmove = function (e) {
    e.preventDefault(e); // Optional
    mouseMoveOrTouchMove(windowToCanvas(e.pageX, e.pageX ));
};

canvas.ontouchend = function (e) {
    e.preventDefault(e); // Optional
    mouseUpOrTouchEnd(windowToCanvas(e.pageX, e.pageX));
};

// Mouse event handlers.....................................

canvas.onmousedown = function (e) {
    e.preventDefault(e); // Optional
    mouseDownOrTouchStart(windowToCanvas(e.clientX, e.clientY));
};

canvas.onmousemove = function (e) {
    e.preventDefault(e); // Optional
    mouseMoveOrTouchMove(windowToCanvas(e.clientX, e.clientY));
};

canvas.onmouseup = function (e) {
    e.preventDefault(e); // Optional
    mouseUpOrTouchEnd(windowToCanvas(e.clientX, e.clientY));
};

// General functions.........................................

function mouseDownOrTouchStart(location) {
      // IMPLEMENT
};

function mouseMoveOrTouchMove(location) {
      // IMPLEMENT
};

function mouseUpOrTouchEnd(location) {
      // IMPLEMENT
};
```

上述 JavaScript 代码实现了若干个非常短小的函数，用于处理触摸事件与鼠标事件。这些函数都直接将逻辑代理给一个不区分触摸事件与鼠标事件的函数，由后者负责统一处理。此外，事件处理器都需要调用事件对象的 preventDefault() 方法，告诉浏览器不要再继续处理用户的触摸手势及鼠标输入了，以免干扰程序的执行。

请注意，事件处理器使用 1.6.1.1 小节中讲过的 windowToCanvas() 方法来决定触摸点的位置。应用程序会把每个触摸事件的 pageX 与 pageY 属性传给该方法，传入的这两个参数分别将滚动条的宽度与高度计算在内。

小技巧：阻止浏览器执行滚动网页等动作

在事件对象上调用 preventDefault() 方法，即可阻止浏览器对该事件采取诸如滚动网页等默认的处理动作。此方法可以避免各种开发者不想看到的浏览器互动操作，例如缩放页面、偶然间选取了网页内容，以及 DIV 闪烁等。

11.3.5 手指缩放

实现“手指缩放”（pinch and zoom）操作所需的全部功能都可以在 HTML5 处理触摸事件的 API 中找到。程序清单 11-5 节选了放大镜程序的一部分代码，以此向大家展示如何让用户通过手指缩放操作调整放大镜的放大倍数。

对于类型为 touchstart 及 touchmove 的触摸事件，如果发现有两个点同时在触摸设备，而且它们之中至少有一个点的位置发生改变，那么我们就判定用户正在“捏”（pinch）屏幕。假如用户正通过“捏”屏幕来进行缩放操作，那么放大镜程序用于处理 touchstart 事件的方法就会计算出两个触摸点之间的距离，以及当前放大倍数与该距离的比值。

接下来，处理 touchmove 事件的方法也会计算当前两个触摸点之间的距离，并且将该值乘以刚才计算好的比值，这样就得出了新的放大倍数。

程序清单 11-5　手指缩放操作的实现代码

```
var magnificationScale = scaleOutput.innerHTML,
    pinchRatio,
    ...

function isPinching (e) {
   var changed = e.changedTouches.length,
       touching = e.touches.length;

   return changed === 1 || changed === 2 && touching === 2;
}

function isDragging (e) {
   var changed = e.changedTouches.length,
       touching = e.touches.length;

   return changed === 1 && touching === 1;
}

canvas.ontouchstart=  function (e) {
   var changed = e.changedTouches.length,
       touching = e.touches.length,
       distance;

   e.preventDefault(e);

   if (isDragging(e)) {
      mouseDownOrTouchStart(windowToCanvas(e.pageX, e.pageY));
   }
   else if (isPinching(e)) {
      var touch1 = e.touches.item(0),
          touch2 = e.touches.item(1),
          point1 = windowToCanvas(touch1.pageX, touch1.pageY),
          point2 = windowToCanvas(touch2.pageX, touch2.pageY);

      distance = Math.sqrt(Math.pow(point2.x - point1.x, 2) +
                           Math.pow(point2.x - point1.x, 2));
      pinchRatio = magnificationScale / distance;
   }
};

canvas.ontouchmove = function (e) {
   var changed =e.changedTouches.length,
       touching = e.touches.length,
       distance, touch1, touch2;

   e.preventDefault(e);
    if (isDragging(e)) {
```

```
        mouseMoveOrTouchMove(windowToCanvas(e.pageX, e.pageY));
    }
    else if (isPinching(e)) {
        var touch1 = e.touches.item(0),
            touch2 = e.touches.item(1),
            point1 = windowToCanvas(touch1.pageX, touch1.pageY),
            point2 = windowToCanvas(touch2.pageX, touch2.pageY),
            scale;

        distance = Math.sqrt(Math.pow(point2.x - point1.x, 2) +
                             Math.pow(point2.x - point1.x, 2));

        scale = pinchRatio * distance;

        if (scale > 1 && scale < 3) {
            magnificationScale =
                parseFloat(pinchRatio * distance).toFixed(2);

            draw();
        }
    }
};

canvas.ontouchend = function (e) {
    e.preventDefault(e);
    mouseUpOrTouchEnd(windowToCanvas(e.pageX, e.pageY));
};
```

11.4 iOS5

苹果公司投入了大量精力来提升 iOS5 的基础架构，以便更好地执行原生应用程序，不过他们也在努力让 HTML5 应用程序的运行效果与 iOS5 设备上的原生应用程序保持一致。本节我们就来研究如何让 HTML5 应用程序看上去和 iOS5 原生应用程序相仿。

提示：HTML5 程序在 iOS5 与 Android 平台之上的区别

苹果公司的 iOS5 平台可以让 HTML5 应用程序具备主屏幕启动图标及启动画面，并且能够以全屏模式运行。在本书付印时，Android 系统也支持在主屏幕上添加启动图标，但是尚不支持启动画面及全屏运行模式。

11.4.1 应用程序图标及启动画面

在 iOS5 系统中，想为 HTML5 应用程序指定图标及启动画面是很简单的，只需使用如下代码即可：

```
<link rel='apple-touch-startup-image'
      href='startup-iPad-landscape.png'/>

<link rel='apple-touch-icon-precomposed' sizes='72x72'
      href='icon-ipad.png'/>
```

在 iOS5 系统中指定图标及启动画面就是这么简单。在图 11-6 中，我们可以看到放大镜及绘图程序的启动图标，图 11-7 则展示了放大镜程序的启动画面。

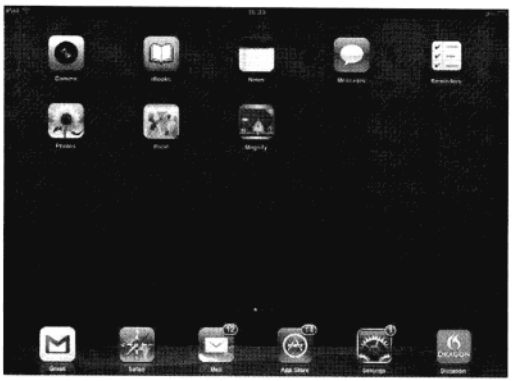


图 11-6 iOS5 系统的应用程序启动图标

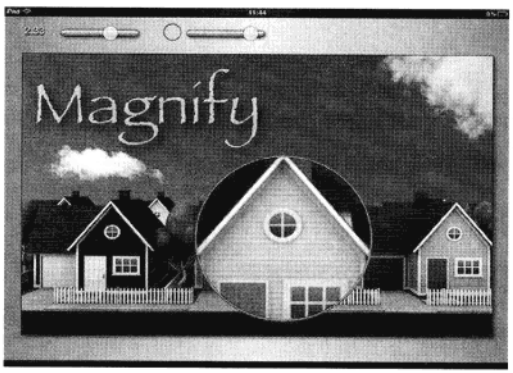


图 11-7 iOS5 系统的应用程序启动画面

提示：在 iOS5 系统上必须使用大小合适的图标

如果在 link 元素的 apple-touch-icon-precomposed 属性中设置了应用程序的主屏幕图标，却发现并没有显示出来，那么也许是图标尺寸需要调整。有关 iOS5 系统图标尺寸的详细信息，请参考：http://bit.ly/yNkfHy。

11.4.2 利用媒体特征查询技术设置 iOS5 系统的应用程序图标及启动画面

我们可以像程序清单 11-6 这样，将媒体特征查询技术与 iOS5 系统对图标及启动画面的支持结合起来，根据不同设备的媒体特征来配置应用程序。

程序清单 11-6 在 iOS5 系统中利用媒体特征查询技术设置应用程序的图标及启动画面

```
<!-- 320x460 for iPhone 3GS -->

<link rel='apple-touch-startup-image'
      media='(max-device-width: 480px)
                 and not (-webkit-min-device-pixel-ratio: 2)'
      href='startup-iphone.png' />
```

```
<!-- 640x920 for retina display -->

<link rel='apple-touch-startup-image'
      media='(max-device-width: 480px)
           and (-webkit-min-device-pixel-ratio: 2)'
      href='startup-iphone4.png' />

<!-- iPad Portrait 768x1004 -->

<link rel='apple-touch-startup-image'
      media='(min-device-width: 768px) and (orientation: portrait)'
      href='startup-iPad-portrait.png'/>

<!-- iPad Landscape 1024x748 -->

<link rel='apple-touch-startup-image'
      media='(min-device-width: 768px) and (orientation: landscape)'
      href='startup-iPad-landscape.png'/>

<link rel='apple-touch-icon-precomposed' sizes='72x72'
      href='icon-ipad.png'/>
```

11.4.3 以不带浏览器饰件的全屏模式运行应用程序

如图 11-8 所示，我们可以将某个 URL 作为图标，添加到 iPad 的主屏幕。

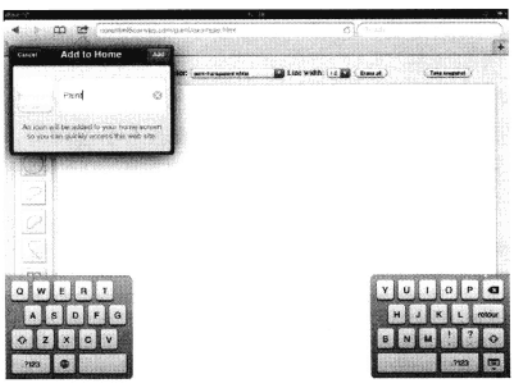


图 11-8 将绘图应用程序的 URL 作为图标，添加到主屏幕

如果我们在网络应用程序中添加如下元标签，那么当用户在主屏幕中启动应用程序之后，它就会以全屏模式运行。

```
<meta name='apple-mobile-web-app-capable' content='yes'/>
```

图 11-9 展示了在全屏模式下运行的绘图应用程序。

11.4.4 应用程序的状态栏

我们可以在 iOS5 系统中控制应用程序状态栏的外观，如图 11-10 所示。

使用如下元标签，即可设置状态栏的外观：

```
<meta name='apple-mobile-web-app-status-bar-style'
      content='black-translucent'/>
```

content 属性可以取如下三个值：

图 11-9　在 iPad 中以全屏模式运行的绘图应用程序

- default
- black
- black-translucent

default 与 black 模式下的颜色是一样的，都如图 11-10 顶部截图所示，而 black-translucent 则会让黑色的状态栏呈现半透明效果。

仔细观察图 11-10，我们就会发现，当状态栏的颜色被设置为 black-translucent 时，应用程序的控件与屏幕顶端之间的距离更近了。这是因为，当状态条不透明时，也就是其颜色为 black 或 default 时，iOS5 系统会让网页的顶端紧贴状态栏的底端。而当状态条半透明时，iOS5 系统则会让网页的顶端与状态栏的顶端对齐。

图 11-10　纯黑色与半透明黑色的状态栏

11.5　虚拟键盘

iPad 系统有一个内置的虚拟键盘，不过，只有在用户点击了文本输入框之后，它才会显示出来。

由于不是原生应用程序，所以我们没有办法在用户未向文本框中输入内容时显示虚拟键盘并捕捉其按键事件。实际上，包括这个绘图程序在内，本书中的全部范例程序都属于这种情况。用户选中代表文本输入模式的那个图标后，就可以点击画布中的任意位置，此时将出现文本输入光标。之后，如果出现键盘的话，我们就可以输入文字内容了，应用程序会将输入的文本显示出来，如图 11-11 所示。

图 11-11　在绘图应用程序中输入文本

只有在用户手动点击文本框并向其中输入内容时，iOS5 系统才会显示虚拟键盘，以编程的方式将焦点转移到文本框是不行的。所以说，许多程序员在开发这种需要以文本框之外的手段来输入文本的程序时，经常会卡住。然而，对于那些熟悉 Canvas 的开发者来说，则可以通过实现一个自制的虚拟键盘并将其显示出来。下面这一小节就会告诉大家怎样去实现它。

实现基于 Canvas 的虚拟键盘

本节我们将要实现一个基于 Canvas 的虚拟键盘，如图 11-12 所示。此键盘在任何设备上都能运作，开发者可以：

图 11-12　按键不透明的虚拟键盘在 iPad 上的运行效果

- 显示并隐藏键盘
- 让键盘以半透明色呈现

• 将键盘加入任何 DIV 元素中
• 把键盘同底层应用程序连接起来

此键盘自带如下功能：

• 支持鼠标与触摸事件。
• 当用户点击虚拟键盘时，自动激活并开始闪烁。
• 根据键盘外围的 DIV 元素来调整键盘自身及其按键的大小。
• 当用户激活某个按键时，通知相应的事件监听器。

通过实现虚拟键盘，我们复习一下本书前面所讲的内容：

• 1.8.1 小节，在 canvas 中使用不可见的 HTML 元素。
• 2.4 节，使用各种颜色与阴影效果来对图形进行填充及描边。
• 2.5 节，创建线性渐变色。
• 2.9.3 小节，实现圆角矩形。
• 第 10 章，实现各种自定义控件。
• 10.3 节，在自定义控件中支持事件监听器机制。
• 11.3.4 小节，统一处理鼠标与触摸事件。

您也可以使键盘以半透明色呈现，如此一来，所有按键也都变成半透明的了，如图 11-13 所示。

在绘图应用程序中，如果文本输入光标位于画布的上半区，那么应用程序就以不透明的方式显示键盘，否则，就将键盘变为半透明，以便用户能看见输入的文本。

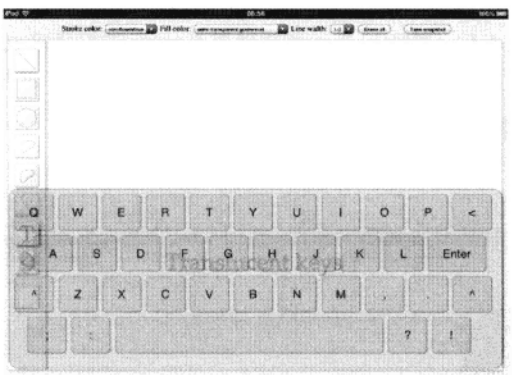


图 11-13 按键半透明的虚拟键盘在 iPad 上的运行效果

与第 10 章所讲的自定义控件一样，虚拟键盘也必须被加入某个 DOM 元素之中才能用，这个元素通常是一个 DIV。当键盘成为子元素时，它就会调整自身的大小，以符合外围的 DOM 元素，如图 11-14 所示。

在学习键盘的实现代码之前，我们先看看绘图程序如何使用键盘。程序清单 11-7 节选了绘图程序 HTML 代码中的一段。在这段 HTML 代码中，有一个 id 属性为 keyboard 的不可见 DIV 元素。由于 CSS 规则中将其高度设置为 0 像素，所以这个名叫 keyboard 的 DIV 元素一开始就是不可见的。

图 11-14　键盘会根据外围的 HTML 元素来调整自身大小

程序清单 11-7　绘图应用程序的 HTML 代码选段

```
<!DOCTYPE html>
<html>
    <head>
        ...
        <style>
            ...

            #keyboard {
                position: absolute;
                left: 25px;
                top: 0px;
                width: 1000px;
                height: 0px;

                background: rgba(129,129,138,0.4);

                -webkit-box-shadow: rgba(0,0,0,0.2) 3px 3px 4px;
                -moz-box-shadow: rgba(0,0,0,0.2) 3px 3px 4px;
                box-shadow: rgba(0,0,0,0.2) 3px 3px 4px;
            }

        </style>
    </head>

    <body>
        ...

        <canvas id='iconCanvas' width='75' height='685'>
            Canvas not supported
        </canvas>

        <canvas id='drawingCanvas' width='915' height='685'>
            Canvas not supported
        </canvas>

        <div id='keyboard'></div>
        ...

        <script src='keyboard.js'></script>
```

```
        <script src='example.js'></script>
    </body>
</html>
```

绘图应用程序的 JavaScript 代码会根据情况显示和隐藏虚拟键盘。程序清单 11-8 列出了这段 JavaScript 代码。

程序清单 11-8　绘图应用程序操控虚拟键盘所用的 JavaScript 代码

```
var keyboard = new COREHTML5.Keyboard();
...

// Keyboard. ..................................................

function showKeyboard() {
    var keyboardElement = document.getElementById('keyboard');

    keyboardElement.style.height = '370px';
    keyboardElement.style.top = '375px';
    keyboardElement.style.border = 'thin inset rgba(0,0,0,0.5)';
    keyboardElement.style.borderRadius = '20px';

    keyboard.resize(1000, 368);
    keyboard.translucent = mousedown.y > drawingCanvas.height/2;
    keyboard.draw();
}
function hideKeyboard() {
    var keyboardElement = document.getElementById('keyboard');

    keyboardElement.style.height = '0px';
    keyboardElement.style.top = '760px';
    keyboardElement.style.border = '';
    keyboardElement.style.borderRadius = '';

    keyboard.resize(1000, 0);
}
...

// Text.......................................................

function startDrawingText() {
    drawingText = true;
    currentText = '';
    drawTextCursor();
    showKeyboard();
}
...

// Event handling functions...................................

function mouseDownOrTouchStartInControlCanvas(loc) {
    if (drawingText) {
        drawingText = false;
        eraseTextCursor();
        hideKeyboard();
    }
    ...
};
...

// Initialization.............................................

keyboard.appendTo('keyboard');
...
```

绘图应用程序启动之后，会创建一个新的键盘对象，并将其加入名为 keyboard 的元素中。

绘图应用程序实现了 showKeyboard() 和 hideKeyboard() 方法，分别用于显示及隐藏键盘。这两个方法都会获取一个引用，此引用指向那个 id 属性为 keyboard 的 DIV 元素。通过调整 DIV 元素的 height 属性，我们就可以令键盘变得可见或不可见。

当用户要输入文本时，绘图应用程序就显示键盘。等到用户按下了工具栏上的某个图标之后，程序则会把键盘隐藏起来。

注意：本书所实现的键盘是不完备的

为了简洁起见，本章实现的这个虚拟键盘的功能并不完备。比如说，键盘上没有数字键，而且我们也没有提供数字输入机制。

我们制作这个键盘，是为了演示如何实现键盘控件，以及如何将它集成到 HTML5 应用程序中，使之能够运行在 iPad 这样的平板电脑上。你可以根据需求来随意扩充虚拟键盘的功能，并将其用在自己所开发的应用程序里。

在实现虚拟键盘的过程中，我们用到了两个 JavaScript 对象，它们是 Key 与 Keyboard。与第 10 章中所采取的办法一样，它们也被定义在名为 COREHTML5 的全局 JavaScript 对象之中，以避免名称冲突。

学会了键盘控件的用法之后，咱们来研究它的实现代码。首先看看按键是怎么实现的。

按键

在程序清单 11-9 所列的 Key 对象构造器中，我们可以看到，Key 对象是很简单的，它只有三个属性，分别代表按键上面所显示的文本、按键当前是否被选中，以及按键是否半透明。

程序清单 11-9　Key 对象的构造器

```
COREHTML5.Key = function (text) {
    this.text = text;
    this.selected = false;
    this.translucent = false;
}
```

Key 对象的方法是在其原型对象中实现的，程序清单 11-10 列出了这些方法。

Key 对象首先使用渐变色填充一个圆角矩形，然后在其正中央绘制代表该按键的文本。绘制工作是由 draw() 方法来完成的，该方法也创建了绘制按键所用的渐变色对象，而且还会设置好绘图环境对象的属性，以便用其绘制矩形及文本。利用 Key 对象的各个属性及方法，我们可以擦除、重绘按键，也可以让按键变得不透明或半透明，还可以选中此按键。被选中的按键的渐变色会比其他按键更深一些。

程序清单 11-10　Key 对象的各个方法

```
COREHTML5.Key.prototype = {
    createPath: function (context) {
        context.beginPath();

        if (this.width > 0)
            context.moveTo(this.left + this.cornerRadius, this.top);
        else
            context.moveTo(this.left - this.cornerRadius, this.top);

        context.arcTo(this.left + this.width, this.top,
```

```
                    this.left + this.width,
                    this.top + this.height,
                    this.cornerRadius);

    context.arcTo(this.left + this.width,
                    this.top + this.height,
                    this.left, this.top + this.height,
                    this.cornerRadius);

    context.arcTo(this.left, this.top + this.height,
                    this.left, this.top,
                    this.cornerRadius);

    if (this.width > 0) {
        context.arcTo(this.left, this.top,
                        this.left + this.cornerRadius, this.top,
                        this.cornerRadius);
    }
    else {
        context.arcTo(this.left, this.top,
                        this.left - this.cornerRadius, this.top,
                        this.cornerRadius);
    }
},

createKeyGradient: function (context) {
    var keyGradient = context.createLinearGradient(
                          this.left, this.top,
                          this.left, this.top + this.height);

    if (this.selected) {
        keyGradient.addColorStop(0, 'rgb(208,208,210)');
        keyGradient.addColorStop(1.0, 'rgb(162,162,166)');
    }
    else if (this.translucent) {
        keyGradient.
            addColorStop(0, 'rgba(298,298,300,0.20)');
        keyGradient.
            addColorStop(1.0, 'rgba(255,255,255,0.20)');
    }
    else {
        keyGradient.addColorStop(0, 'rgb(238,238,240)');
        keyGradient.addColorStop(1.0, 'rgb(192,192,196)');
    }

    return keyGradient;
},

setKeyProperties: function (context, keyGradient) {
    context.shadowColor = 'rgba(0,0,0,0.8)';
    context.shadowOffsetX = 1;
    context.shadowOffsetY = 1;
    context.shadowBlur = 1;

    context.lineWidth = 0.5;

    context.strokeStyle = 'rgba(0,0,0,0.7)';
    context.fillStyle = keyGradient;
},

setTextProperties: function (context) {
```

```
        context.shadowColor = undefined;
        context.shadowOffsetX = 0;

        context.font = '100 ' + this.height/3 + 'px Helvetica';
        context.fillStyle = 'rgba(0,0,0,0.4)';
        context.textAlign = 'center';
        context.textBaseline = 'middle';
    },

    draw: function (context) {
        var keyGradient = this.createKeyGradient(context);

        context.save();

        this.createPath(context);

        this.setKeyProperties(context, keyGradient);
        context.stroke();
        context.fill();

        this.setTextProperties(context);
        context.fillText(this.text, this.left + this.width/2,
                                    this.top + this.height/2);

        context.restore();
    },

    erase: function(context) {
        context.clearRect(this.left-2, this.top-2,
                          this.width+6, this.height+6);
    },

    redraw: function (context) {
        this.erase(context);
        this.draw(context);
    },

    toggleSelection: function (context) {
        this.selected = !this.selected;
    },

    isPointInKey: function (context, x, y) {
        this.createPath(context);
        return context.isPointInPath(x, y);
    },

    select: function () {
        this.selected = true;
    },

    deselect: function () {
        this.selected = false;
    },
}
```

Key 对象是写给 Keyboard 对象用的，所以接下来咱们就看看这个 Keyboard 对象。

Keyboard 对象

Keyboard 对象与第 10 章中定义的那些对象一样，都属于自定义控件。所以与那一章所讲的自定义控件一样，我们也可以创建 keyboard 对象并将其加入某个既有的 DOM 元素之中，这里的

DOM 元素通常是指 DIV。

程序清单 11-11 列出了 Keyboard 对象的构造器，其代码创建了一个 4 行 11 列的二维数组，其中的每个元素都是一个 Key 对象。随后，构造器创建了绘制虚拟键盘所用的 canvas，并将其加入 DOM 元素之中。

程序清单 11-11　Keyboard 对象的构造器

```
// Constructor.............................................................

COREHTML5.Keyboard = function() {
   var keyboard = this;

   this.keys = [
       [ new COREHTML5.Key('Q'), new COREHTML5.Key('W'),
         new COREHTML5.Key('E'), new COREHTML5.Key('R'),
         new COREHTML5.Key('T'), new COREHTML5.Key('Y'),
         new COREHTML5.Key('U'), new COREHTML5.Key('I'),
         new COREHTML5.Key('O'), new COREHTML5.Key('P'),
         new COREHTML5.Key('<') ],

       [ new COREHTML5.Key('A'), new COREHTML5.Key('S'),
         new COREHTML5.Key('D'), new COREHTML5.Key('F'),
         new COREHTML5.Key('G'), new COREHTML5.Key('H'),
         new COREHTML5.Key('J'), new COREHTML5.Key('K'),
         new COREHTML5.Key('L'), new COREHTML5.Key('Enter') ],

       [ new COREHTML5.Key('^'), new COREHTML5.Key('Z'),
         new COREHTML5.Key('X'), new COREHTML5.Key('C'),
         new COREHTML5.Key('V'), new COREHTML5.Key('B'),
         new COREHTML5.Key('N'), new COREHTML5.Key('M'),
         new COREHTML5.Key(','), new COREHTML5.Key('.'),
         new COREHTML5.Key('^') ],

       [ new COREHTML5.Key(';'), new COREHTML5.Key(':'),
         new COREHTML5.Key(' '), new COREHTML5.Key('?'),
         new COREHTML5.Key('!') ]
   ];

   this.createCanvas();
   this.createDOMElement();

   this.translucent = false;
   this.shifted = false;
   this.keyListenerFunctions = [];

   this.context.canvas.onmousedown = function (e) {
      keyboard.mouseDownOrTouchStart(keyboard.context,
         keyboard.windowToCanvas(keyboard.context.canvas,
                                 e.clientX, e.clientY));

      // prevents inadvertent selections on desktop

      e.preventDefault();
   };

   this.context.canvas.ontouchstart = function (e) {
      keyboard.mouseDownOrTouchStart(keyboard.context,
         keyboard.windowToCanvas(keyboard.context.canvas,
                                 e.touches[0].clientX,
                                 e.touches[0].clientY));
```

```
        e.preventDefault(); // prevents flashing on iPad
    };

    return this;
}
```

在默认情况下，键盘并不呈现半透明的效果，键盘上的各个按键也是不透明的，所以其translucent 属性为 false。而且，由于用户一开始并未激活 Shift 键，所以键盘对象的 shifted 属性默认也是 false。每个键盘对象刚被创建好的时候，都带有一个空数组，其中可以放入按键监听器对象，以便在用户激活键盘上的按键时，keyboard 对象能够通知给这些监听器。

Keyboard 对象构造器在这段代码的最后向 canvas 对象注册了两个事件处理器，一个用于处理鼠标按下事件，另一个则用于处理触摸开始事件。这两个事件处理器都会分别调用处理相应事件所需的方法。

程序清单 11-12 列出了 Keyboard 对象的各个方法。

keyboard 对象所拥有的方法可以按其内容分为 5 类：

- 大多数自定义控件都需要实现的通用方法。
- 用于绘制键盘及其按键的绘制方法。
- 处理诸如创建按键、激活按键等事务所用的按键方法。
- 用于支持监听器注册及通知机制的按键监听器方法。
- 可以同时处理鼠标与触摸事件的事件处理器。

程序清单 11-12　Keyboard 对象的各个方法

```
// Prototype.....................................................................

COREHTML5.Keyboard.prototype = {

    // General functions ..........................................................

    windowToCanvas: function (canvas, x, y) {
        var bbox = canvas.getBoundingClientRect();
        return { x: x - bbox.left * (canvas.width  / bbox.width),
                 y: y - bbox.top  * (canvas.height / bbox.height)
               };
    },

    createCanvas: function () {
        var canvas = document.createElement('canvas');
        this.context = canvas.getContext('2d');
    },

    createDOMElement: function () {
        this.domElement = document.createElement('div');
        this.domElement.appendChild(this.context.canvas);
    },

    appendTo: function (elementName) {
        var element = document.getElementById(elementName);

        element.appendChild(this.domElement);
        this.domElement.style.width = element.offsetWidth + 'px';
        this.domElement.style.height = element.offsetHeight + 'px';
        this.resize(element.offsetWidth, element.offsetHeight);
        this.createKeys();
```

```
},

resize: function (width, height) {
   this.domElement.style.width = width + 'px';
   this.domElement.style.height = height + 'px';

   this.context.canvas.width = width;
   this.context.canvas.height = height;
},

// Drawing Functions.......................................................

drawRoundedRect: function (context, cornerX, cornerY,
                           width, height, cornerRadius) {
   if (width > 0)
     this.context.moveTo(cornerX + cornerRadius, cornerY);
   else
     this.context.moveTo(cornerX - cornerRadius, cornerY);

   context.arcTo(cornerX + width, cornerY,
                 cornerX + width, cornerY + height,
                 cornerRadius);

   context.arcTo(cornerX + width, cornerY + height,
                 cornerX, cornerY + height,
                 cornerRadius);

   context.arcTo(cornerX, cornerY + height,
                 cornerX, cornerY,
                 cornerRadius);

   if (width > 0) {
      context.arcTo(cornerX, cornerY,
                    cornerX + cornerRadius, cornerY,
                    cornerRadius);
   }
   else {
         context.arcTo(cornerX, cornerY,
                    cornerX - cornerRadius, cornerY,
                    cornerRadius);
   }

   context.stroke();
   context.fill();
},

drawKeys: function () {
   for (var row=0; row < this.keys.length; ++row) {
      for (var col=0; col < this.keys[row].length; ++col) {
         key = this.keys[row][col];

         key.translucent = this.translucent;
         key.draw(this.context);
      }
   }
},

draw: function (context) {
   var originalContext, key;

   if (context) {
```

```
        originalContext = this.context;
        this.context = context;
    }

    this.context.save();
    this.drawKeys();

    if (context) {
        this.context = originalContext;
    }

    this.context.restore();
},

erase: function() {
    // Erase the entire canvas
    this.context.clearRect(0, 0, this.context.canvas.width,
                           this.context.canvas.height);
},

// Keys.....................................................................

adjustKeyPosition: function (key, keyTop, keyMargin,
                             keyWidth, spacebarPadding) {
    var key = this.keys[row][col],
        keyMargin = this.domElement.clientWidth /
                    (this.KEY_COLUMNS*8),
        keyWidth = ((this.domElement.clientWidth - 2*keyMargin) /
                    this.KEY_COLUMNS) - keyMargin,
        keyLeft = keyMargin + col * keyWidth + col * keyMargin;

    if (row === 1) keyLeft += keyWidth/2;
    if (row === 3) keyLeft += keyWidth/3;

    key.left = keyLeft + spacebarPadding;
    key.top = keyTop;
},

adjustKeySize: function (key, keyMargin, keyWidth, keyHeight) {
    if (key.text === 'Enter')  key.width = keyWidth * 1.5;
    else if (key.text === ' ') key.width = keyWidth * 7;
    else                       key.width = keyWidth;

    key.height = keyHeight;
    key.cornerRadius = 5;
},

createKeys: function() {
    var key,
        keyMargin,
        keyWidth,
        keyHeight,
        spacebarPadding = 0;

    for (row=0; row < this.keys.length; ++row) {
        for (col=0; col < this.keys[row].length; ++col) {
            key = this.keys[row][col];

            keyMargin = this.domElement.clientWidth /
                        (this.KEY_COLUMNS*8);
            keyWidth = ((this.domElement.clientWidth - 2*keyMargin) /
```

```
                    this.KEY_COLUMNS) - keyMargin;

            keyHeight = ((this.KEYBOARD_HEIGHT - 2*keyMargin) /
                         this.KEY_ROWS) - keyMargin;

            keyTop = keyMargin + row * keyHeight + row * keyMargin;

            this.adjustKeyPosition(key, keyTop, keyMargin,
                                   keyWidth, spacebarPadding);

            this.adjustKeySize(key, keyMargin, keyWidth, keyHeight);

            if (this.keys[row][col].text === ' ') {
               spacebarPadding = keyWidth*6; // pad from now on
            }
         }
      }
   },

   getKeyForLocation: function (context, loc) {
      var key;

      for (var row=0; row < this.keys.length; ++row) {
         for (var col=0; col < this.keys[row].length; ++col) {
            key = this.keys[row][col];

            if (key.isPointInKey(context, loc.x, loc.y)) {
              return key;
            }
         }
      }
      return null;
   },

   shiftKeyPressed: function (context) {
      for (var row=0; row < this.keys.length; ++row) {
         for (var col=0; col < this.keys[row].length; ++col) {
            nextKey = this.keys[row][col];

            if (nextKey.text === '^') {
               nextKey.toggleSelection();
               nextKey.redraw(context);
               this.shifted = nextKey.selected;
            }
         }
      }
   },

   activateKey: function (key, context) {
      key.select();
      setTimeout( function (e) {
                     key.deselect();
                     key.redraw(context);
                  }, 200);

      key.redraw(context);

      this.fireKeyEvent(key);
   },

   // Key listeners.............................................................
```

```
    addKeyListener: function (listenerFunction) {
       this.keyListenerFunctions.push(listenerFunction);
    },

    fireKeyEvent: function (key) {
       for (var i=0; i < this.keyListenerFunctions.length; ++i) {
          this.keyListenerFunctions[i](
              this.shifted ? key.text : key.text.toLowerCase());
       }
    },

    // Event handlers.............................................................

    mouseDownOrTouchStart: function (context, loc) {
       var key = this.getKeyForLocation(context, loc);

       if (key) {
          if (key.text === '^') {
             this.shiftKeyPressed(context);
          }
          else {
            if (this.shifted) this.activateKey(key, context);
            else              this.activateKey(key, context);
          }
       }
    }
};
```

代码中的这几个通用方法分别用于创建键盘所用的 canvas 以及 DOM 元素，并将该元素加入某个既有的 DOM 树中。Keyboard 对象的 appendTo() 方法会调用 resize() 方法，后者将调整绘制键盘所用的 canvas 以及键盘对象中 DOM 元素的尺寸，使之符合外围的 DOM 元素。有关这些通用方法的实现详情，请参考第 10 章所讲的其他自定义控件。

Keyboard 对象的绘制方法也很易懂。draw() 方法接受一个可选的 Canvas 绘图环境对象为其参数，如果调用时传入了此参数，那么 Keyboard 对象就将键盘图样绘制到该参数之中，否则就绘制到对象内部的 canvas 里面。不论哪种情况，键盘上的按键都会随同键盘一并画出。

Keyboard 对象之中的麻烦事都是由这几个按键方法来完成的，例如创建按键对象，根据键盘 canvas 中的某个坐标点位置返回与之对应的按键，以及激活某个按键，等等。

addKeyListener() 与 fireKeyEvent() 这两个按键监听器方法，分别负责向键盘控件注册按键监听器，以及将按键事件通知给已注册的监听器。

最后，mouseDownOrTouchStart() 方法负责应对鼠标按下及触摸开始事件，如果发现某个键被按下了，那么该方法就激活此按键。

11.6 总结

在本章中，读者学到了如何让基于 Canvas 的应用程序在移动设备上运行。一开始，我们讲解了移动设备之中的“视窗”（viewport）这一概念，以及与之相关的 viewport 元标签。通过元标签，我们可以配置离屏的排版视窗，以适应各种移动设备的浏览器。其后，我们学习了媒体特征查询技术，它可以让开发者为各种不同的设备定义不同的 CSS 规则。同时，我们也可以监听

“媒体特征查询列表”，在列表之中，一旦诸如屏幕显示方向这样的媒体特征发生了变化，那么以JavaScript 代码写成的监听器就可以据此做出相应的处理。

本章也告诉大家如何处理触摸事件，以及怎样利用触摸事件 API 来实现一个能运行在各手机浏览器之上的“手指缩放”操作。

然后，我们讲了一点题外话，也就是如何让 HTML5 应用程序与 iOS5 平台的原生应用程序高度相似。这部分内容包括了创建应用程序图标与启动图像，以及在不带浏览器饰件的情况下用全屏模式运行应用程序。

本章最后实现了一个虚拟键盘控件，在实现过程中，我们复习了本书前面所学的很多技术。

[General Information]
书名=HTML5 Canvas核心技术 图形、动画与游戏开发
作者=（美）基瑞著
页数=486
出版社=北京市：机械工业出版社
出版日期=2013.02
SS号=13260600
DX号=000011750854