移动开发经典丛书

Professional HTML5 Mobile Game Development

HTML5 移动游戏开发高级编程

[美] Pascal Rettig 著
叶 斌 译

清华大学出版社

畅游潜力无限的HTML5移动游戏世界

热切期望进入如火如荼的移动游戏世界？《HTML5移动游戏开发高级编程》将助你实现梦想。这本精品书籍面向有兴趣为所有移动和触摸屏设备创建游戏的开发人员，以你现有的HTML5和JavaScript知识为基础，分步讲解如何使用HTML5构建单玩家和多玩家移动游戏。本书涵盖构建HTML5游戏的标准模式、构建方法的选择(CSS3、SVG或画布)以及流行的游戏引擎和框架等主题。最重要的是，你可修改和扩展本书提供的6个基础游戏的代码，最终开发出自己的游戏。

主要内容

- 阐释如何择机选用三种主要方法(CSS3、SVG或画布)之一来构建HTML游戏
- 介绍使用HTML5构建实时多玩家游戏的标准模式
- 讲述JavaScript游戏开发基础知识
- 分步讲解如何创建2D平台动作游戏以及构建非传统多人游戏界面
- 介绍各种移动增强功能，如地理定位、设备方向、加速和声音等
- 提供将HTML5游戏打包以便将其发布到应用商店的建议

作者简介

Pascal Rettig经营着网络咨询公司Cykod；Cykod成立于2006年，总部设在波士顿，主营业务是在线交互应用。Pascal也是GamesForLanguage的CTO，他组织成立了波士顿HTML5游戏开发研讨会，同时担任*UX Magazine*游戏版块的特约编辑。

源代码下载及技术支持

http://www.wrox.com

http://www.tupwk.com.cn/downpage

wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

清华大学出版社数字出版网站
WQBook 文泉局
www.wqbook.com

Wrox
An Imprint of
WILEY

ISBN 978-7-302-35631-8

9 787302 356318 >

定价：68.00元

移动开发经典丛书

HTML5 移动游戏开发高级编程

[美] Pascal Rettig 著

叶 斌 译

清华大学出版社

北 京

Pascal Rettig
Professional HTML5 Mobile Game Development
EISBN：978-1-118-30132-6

图书在版编目(CIP)数据
HTML5 移动游戏开发高级编程/(美) 瑞特格(Rettig, P.) 著；叶斌 译. —北京：清华大学出版社，2014
(移动开发经典丛书)
书名原文：Professional HTML5 Mobile Game Development
ISBN 978-7-302-35631-8

Ⅰ. ①H…　Ⅱ. ①瑞…　②叶…　Ⅲ. ①超文本标记语言—游戏程序—程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 046666 号

责任编辑：王　军　韩宏志
装帧设计：牛静敏
责任校对：成凤进
责任印制：何　芊

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　张：33　　字　数：803 千字
版　　次：2014 年 4 月第 1 版　　印　次：2014 年 4 月第1 次印刷
印　　数：1～4000
定　　价：68.00 元

产品编号：053104-01

作 者 简 介

Pascal Rettig 在孩童时代就迷恋上编程，开始编程时只有 7 岁，那时的他已经可以在 Apple II 上编写 BASIC 游戏了。Pascal 拥有麻省理工学院(Massachusetts Institute of Technology)的理学学士学位，并在 2002 年获得了麻省理工学院计算机科学和电子工程方面的工程硕士学位。自 1995 年以来，他一直在研究和开发各种 Web 技术。2011 年，Pascal 构建了基于 HTML5 游戏的语言学习系统 GamesForLanguage.com，他目前是交互式网络公司 Cykod 的合伙人。在波士顿，他每月组织一次 HTML5 游戏开发研讨会，这是美国国内历史最悠久的 HTML5 Game Development 月度研讨会之一；与此同时，他还管理着 HTML5 Game Development 社区的新闻网站 html5gamedevelopment.org。

技术编辑简介

Chris Ullman 是 MIG 的一位资深软件开发人员，致力于.NET 方面的开发；同时，他也是一位技术编辑/作家，就像是茶壶中泡久的茶袋一样，多年来他一直沉浸在与 Web 相关的各种技术中。Chris 具有计算机科学背景，所以在 ASP 的全盛时期(1997 年)，他倾向于选择 MS 解决方案。从 Wrox Press 出版社的《ASP 指南》开始，他编辑或参与编写了 30 多本书籍，其中最引人瞩目的是作为 Wrox 最畅销的 *Beginning ASP/ASP.NET 1.x/2* 系列图书的第一作者。目前，他闲居在康沃尔的荒野之上，过着与计算机技术无关的日子：跑步、创作音乐，还要和妻子 Kate 一起让三个打闹不已的孩子安静下来。

致　谢

我要感谢我的妻子 Martha，她不仅容忍我把所有空闲时间用来撰写此书(同时还要打理两家刚起步的公司)，而且慷慨地帮忙设计了本书用到的所有定制的游戏图片，确保读者可以欣赏到专业水准的艺术作品。

还要感谢家人对我的这份努力的支持，尽管这段时间我费尽心思让自己与世隔绝，但他们仍继续接受我作为家庭的一员。

特别感谢本书编辑 Carol Long、Jennifer Lynn 和 San Dee Phillips，感谢他们帮助我这个新手走完了把一些代码页变成一本有内聚力的图书的过程；同时还要感谢技术编辑 Chris Ullman，感谢他竭尽所能，以求确保此书尽量做到完美无缺。

最后感谢波士顿 HTML5 Game Development 社区，该社区是一个十分了不起的技术社区，正是置身于这样有上进心、这样聪明的一群人中，我才能做到不断学习、保持热情，并继续去研究一些新的项目。

前　言

自从社交游戏开始把游戏带给大众，帮助把这一曾经的亚文化变成一种面向大众的主流现象之后，游戏界和万维网就在彼此碰撞中发展着。再在其中投入移动设备，你会看到一种大众现象骤然出现，随着人们手中持有的设备越来越多，这一大众现象也变得越来越重要。

例如，截至撰写本书之时，一个在网络上大获成功的故事就是关于游戏开发厂商 Rovio 的，这一“愤怒的小鸟”游戏系列创造者的估值约 80 亿美元，几乎与手机制造巨头诺基亚(Nokia)价值相当。现在，人们花在手机和平板电脑上的时间比以往任何时候都要多，游戏以及社交网络占用了这段时间中的相当高的比例，智能手机和平板电脑显著取代了任天堂和索尼的专用移动游戏设备。借助 HTML5，游戏开发者现在拥有了这样的技术能力，即通过单一代码库能影响到更多的人，比以往任何时候能想象得到的都要多。

HTML5 移动游戏开发目前还是一项新技术，人们还不知道该如何看待这一技术，这很像是 2008 年时的智能手机游戏，苹果公司的应用商店(App Store)就是在这一年推出的。不过，一些重量级的组织已经加入进来，力保 HTML5 游戏取得成功。其中 Facebook 在 2012 年 5 月推出了它的应用中心(App Center)，把基于 HTML5 的 Web 应用变成了移动设备上的一等公民，它正在研究一些移动设备上的货币化手段，以求不再受制于苹果公司这种从其应用商店的应用内购买中抽取 30%手续费的做法。类似地，诸如 AT&T 一类的运营商也把 Web 应用看成一种从 Google 和苹果公司那里夺回失去收入的一种手段。

然而，在 HTML5 的游戏开发宏图中，一切都不容乐观。不同设备有着不同的功能、性能水准和屏幕分辨率，在 HTML5 移动游戏开发这一危险水域航行需要小心把握航向，而这正是本书能发挥作用的地方。本书旨在提供一个使用 HTML5 构建移动游戏的实用路线图，内容涵盖了媒介的可能性和局限性。若说 HTML5 桌面游戏的开发仍处于起步阶段，那么 HTML5 移动游戏的开发就还处于萌芽状态。成就一番伟业的可能性触手可及，但媒介的首记扣杀是否成功仍有待观察。

在早期阶段就涉猎某项技术可带来显著好处，使用新技术的幸事之一是噪音水平最低，相比其他已被接受的媒介，制造轰动所需的代价更少。HTML5 游戏，特别是移动设备上的这些游戏，其预算仅为普通 PC 和控制台游戏所需的数百万美元的很少一部分。然而，由于万维网的病毒式扩散本质，它们却有机会在瞬间创造出巨大的销量。HTML5 移动游戏有着更大的爆炸式增长可能性，因为它们能够借助链接实现实时共享，不需要接受者从应用商店下载一个可能并不适用于所持设备的应用。

本书将开启一段旅程，带领你畅游 HTML5 移动游戏开发这一激动人心的领域所呈现的可能性世界，我希望你扬帆起航，向这个世界进发。

本书读者对象

本书是为任何想要使用基于标准的无插件技术在浏览器中构建交互式游戏的读者准备的，它把重点放在移动游戏开发上，因为与诸如 Flash 之类存在竞争的 Web 技术相比，这是 HTML5 的优势所在，不过，你构建出来的游戏同样可在桌面浏览器中运行。

开发 HTML5 移动游戏需要通过一系列不同的媒介使用一些跨学科技能，若要正确无误地做到这一点，你需要对 JavaScript 语言有基本的了解，因为你将要最大限度地利用 JavaScript 在浏览器中构建游戏。本书不会从头讲解 JavaScript 知识，而是基于你所掌握的 JavaScript 知识快速构建游戏。

若不是天天都使用 JavaScript，那么你可能会发现有些地方的代码很难理解，但并非就完全没有希望了——要快速掌握 JavaScript，Gouglas Crockford 撰写的 *JavaScript: The Good Parts*(O'Reilly 出版，2008 年)可以帮助你熟悉该门语言，这本书只有区区 180 页，却产生了重大影响。在遇见书中提到的某些可能不太熟悉的技术时，你还可把此书当成参考资料。

若你是一位桌面游戏开发者，熟悉 C++更甚于 JavaScript，那么理解书中所谈内容是没有问题的，不过同样要说明的是，因为相比于 C++，JavaScript(尽管有着类 C 的语法)与 Lisp 有着更多的共性，所以你可能也会希望查阅一下 Crockford 的这本书。JavaScript 是弱类型、可变方法绑定的，对闭包的支持可能会带来一些难以理解的地方。

使用 Flash 构建游戏出身的 ActionScript 开发者应该会有宾至如归的感觉，唯一的主要障碍是 HTML5 游戏开发的连贯性还不如 Flash。务必密切留意第 7 章，因为该章内容说明了如何检查和调试 JavaScript，这样在游戏出现问题时，你就不会感到束手无策了。一些浏览器内置了非常强大的脚本调试器，所以你应该不会太过怀念 Flash IDE。

本书内容

本书谈论的是使用 HTML5 创建游戏，这些游戏运行在诸如 iOS 设备和 Android 一类兼容 HTML5 的智能手机上。Windows Phone 7.5 支持画布，因此在某些实例中也是游戏开发针对的目标，但因为它的画布性能受限，不支持基于标准的多点触控事件，所以在某些情况下，只能对 Windows 手机提供有限的支持。

倘若针对的目标是 iOS 5.0 或更高版本的移动 Safari 以及 Android 4.0 或更高版本的 Android Chrome，那么你能做到最大限度地利用本书，因为这些设备都拥有最快的 JavaScript 引擎和硬件加速的画布支持。许多游戏可以运行在较旧版本的 Android 上，但性能会受到限制。

本书的组织方式

本书由 8 部分组成，每一部分都带着某种专门目的来传授移动 HTML5 游戏的开发知识。

第 I 部分：“HTML5 潜力初探”通过三章内容传授如何从头构建可运行在任何支持画布的设备上的 HTML5 移动游戏，展示了这样的一种基本事实，即在不必加入任何外部库的情况下，能够利用哪些资源来快速建立和运行游戏。

第 II 部分：“移动 HTML5”回退一步，详细讨论移动设备上的 HTML5 的情况，同时还讨论了两个库——jQuery 和 Underscore.js——本书余下部分内容将用它们来构建游戏。

第III部分：“JavaScript 游戏开发基础”首先详细讲解如何检查和调试游戏，以及如何通过命令行使用 Node.js 运行 JavaScript；接着讨论了从头构建可重用 HTML5 游戏引擎的过程，展示了把代码结构化和组织成一些连贯模块的做法。

第IV部分：“使用 CSS3 和 SVG 构建游戏”绕开画布，展示如何使用其他两种技术——CSS3 和可伸缩矢量图(Scalable Vector Graphics，SVG)——在移动设备上构建游戏。第 14 章还介绍了很受欢迎的 JavaScript 物理引擎 Box2D。

第 V 部分：“HTML5 画布”首先详细讨论 canvas 标签，继而构建一个触摸友好的 2D 平台游戏和一个针对该平台游戏的用来构建关卡的关卡编辑器。

第VI部分：“多人游戏”展示如何使用 WebSocket 创建能够在多个玩家之间以异步实时方式提供有意义的交互游戏。

第VII部分：“移动增强”探讨如何使用其他一些 HTML5 系列 API 来增强游戏，内容涵盖地理定位和设备方向，以及移动设备上的 HTML5 声音状态。

第VIII部分：“游戏引擎和应用商店”研究一些可用的 HTML5 游戏引擎的前景——其中既有商用的也有开源的——并帮助你确定合适的引擎，这部分内容还涵盖了一些支持把硬件加速的 HTML5 游戏发布到本地移动应用商店中的新兴技术。

本书用到的软件产品和工具

本书例子运行在 Windows、OSX 或 Linux 的现代桌面浏览器中，术语“现代桌面浏览器”指的是 Internet Explorer 9 或以上版本，以及 Safari、Firefox 或 Chrome 的最新版本。

要在移动设备上运行例子，你需要运行 iOS 5.0 或更高版本的 iOS 设备，或是运行 Android 4.0 或更高版本的 Android 设备，这样能获得最佳效果。许多例子可在 Android 2.2 或更高版本上工作，但性能可能会有所限制。

若在 Mac 上运行，可借助可随同 XCode 一起安装的 iOS 模拟器来运行其中的一些例子。遗憾的是，目前的 Android 模拟器速度太慢，还不能作为很好运行 HTML5 游戏的测试床。

一些约定

为了帮助你最大限度地从书中汲取知识，以及进一步理解上下文信息，我们使用了一些贯穿全书的约定做法。

警告：像这样有着提醒字样的方框中存放的是一些与上下内容有直接关系的信息，这些信息很重要，需要牢记。

提示：以类似如此的方式列出一些注释、提示、建议和技巧。

补充栏

以与此类似的方式放置一些与当前讨论有关的解说。

本书以两种不同风格显示代码：

- 使用没有突出处理的 monofont 字体类型来显示大部分的代码例子。
- 使用粗体来强调当前上下文中特别重要的代码。

源代码

在读者学习本书中的示例时，可以手动输入所有代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/或 www.tupwk.com.cn/downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击 Download Code 链接，就可以获得所有的源代码。既可以选择下载一个大的包含本书所有代码的 ZIP 文件，也可以只下载某个章节中的代码。

注意：由于许多图书的标题都很类似，因此按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-1-118-30132-6。

在下载代码后，只需用解压缩软件对其进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。记住，可以使用书中列出的程序清单的编号容易地找到所要寻找的代码，如“程序清单 0-1”。

当为大多数可下载的源代码文件命名时，我们会使用这些清单中的数值。对于那些很少的没有用它自己的清单数值命名的程序清单，它们都与文件名匹配，所以很容易就可以在下载的源代码文件中找到它们。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/ booklist.shtml。

如果在 Book Errata 页面上没有看到你找出的错误，请进入 www.worx.com/contact/techsupport.shtml，填写表单，发电子邮件，我们就会检查你的信息，如果是正确的，就在本书的勘误表中粘贴一个消息，我们将在本书的后续版本中采用。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于传送与 Wrox 图书相关的信息和相关技术，与其他读者和技术用户交流。该论坛提供了订阅功能，当论坛上有新帖子时，会给你发送你选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 http://p2p.wrox.com 上有许多不同的论坛，帮助读者阅读本书，在读者开发自己的应用程序时，也可以从这个论坛中获益。要加入这个论坛，必须执行下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读其内容，单击 Agree 按钮。

(3) 提供加入论坛所需的信息及愿意提供的可选信息，单击 Submit 按钮。

(4) 然后就可以收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

提示：不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，才能发送自己的信息。

加入论坛后，就可以发送新信息，回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作原理，以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

目　录

第 I 部分

HTML5潜力初探

- 第 1 章：先飞后走，先难后易
- 第 2 章：从玩具到游戏
- 第 3 章：试飞结束，向移动进发

第 1 章

先飞后走，先难后易

本章提要

- 创建游戏的 HTML5 脚本
- 加载并在画布上绘制图像
- 设置游戏的结构
- 创建动画背景
- 监听用户的输入

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 1 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

1.1 引言

一直以来，游戏都是一种把技术运用到极致的媒介，本书延续了这一光荣传统，采用一些 Web 核心技术——HTML、CSS 和 JavaScript——并最大限度地挖掘了这些技术的功能和性能。作为一种游戏媒介，HTML5 仅用很短的时间就在功能方面取得了长足的进步，许多人相信，未来几年内，浏览器游戏将会是游戏的主要分发机制之一。

尽管最初目的并非用作创建游戏的环境，但 HTML5 实际上是一个高水准的、非常适合这项工作的环境。因此，不必立刻构建一个引擎来把所有样板代码抽离，你也能直接创造出一些好东西来，这就是你将要做的：在 HTML5 上从头开始构建一个一次性游戏——一个名为 Alien Invasion 的纵向卷轴 2D 太空射击类游戏。

1.2　用 500 行代码构建一个完整游戏

为了证明使用 HTML5 构建游戏是多么容易，前三章构建出来的这个游戏最终包含了不到 500 行的代码，且完全不使用任何库。

1.2.1　了解游戏

Alien Invasion 是一个纵向卷轴的 2D 射击类游戏，秉承了游戏“1942”的精髓(但是在太空中)，或可把它看成 Galaga 的一个简化版本。玩家控制出现在屏幕底部的飞船，操作飞船垂直飞过无边无际的太空领域，同时保卫地球，抵抗成群入侵的外星人。

在移动设备上玩游戏时，用户是通过显示在屏幕左下角的左右箭头进行控制的，发射(Fire)按钮在右侧。在桌面上玩游戏时，用户可以使用键盘的箭头键来控制飞行和使用空格键进行射击。

为弥补移动设备屏幕大小各有不同这一不足，游戏会调整游戏区域，始终按照设备大小来运行游戏。在桌面上，游戏会被放在浏览器页面中间的一个矩形区域中运行。

1.2.2　结构化游戏

几乎每个这种类型的游戏都包含了几块相同的内容：一些资产的加载、一个标题画面、一些精灵、用户输入、碰撞检测以及一个把这几块内容整合在一起的游戏循环。

该游戏尽可能少使用规范的结构，一种替代构建显式类的做法是利用 JavaScript 的动态类型(1.5.1 节将探讨更多关于这方面的内容)。诸如 C、C++和 Java 的一类语言被称为“强类型”的，这是因为你需要明确指出传递给方法的参数的类型，这意味着在希望给同一个方法传递不同类型的对象时，你需要显式地定义一些基类和接口。JavaScript 是弱(或动态)类型的，因为该语言不强制参数的类型，这意味着对象的定义更加松散，可按需给每个对象添加方法，不必构建一大堆的基类或接口。

图像资产的处理极为简单，先加载一幅图像，接着调用一个把所有图像精灵都放在一张 PNG 图像中的“精灵表(sprite sheet)”，然后在完成图像加载之后执行回调。另外，该游戏还提供了把精灵绘制到画布上的方法。

标题画面为主标题渲染精灵，显示移动的星空，这是与主游戏的背景相同的动画星空。

游戏循环也很简单，你有一个可被当成当前场景对待的对象，可以告诉该场景更新自身，然后绘制自身。这是一种简单的抽象，可用于标题画面和游戏的结束画面，也可用于游戏的主体部分。

用户输入方面，可以使用几个事件监听器来监听键盘的输入，以及使用画布上的几个“区”来检测触摸输入，可使用 HTML 标准方法 addEventListener 来同时支持这两种做法。

最后，就碰撞检测而言，把难实现的东西都排除出去，仅通过循环遍历每个对象的边框来检测碰撞。这是实现碰撞检测的一种缓慢且初级的做法，但实现起来简单，只要待检查的精灵数量不多，这种做法的效果还算不错。

1.2.3　最终实现的游戏

若想感受一下游戏出来之后的样子，可看一看图 1-1，也可以使用桌面浏览器和手边的任何一种移动设备来访问 http://cykod.github.com/AlienInvasion/。该游戏应该能够运行在任何支持 HTML5 画布的智能手机上；不过，在 Ice Cream Sandwich 之前的 Android 版本上(即 Android 4.0 之前的版本上)，画布的性能表现不佳。

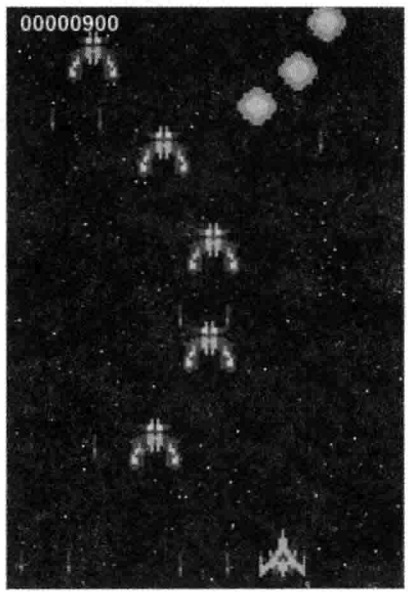


图 1-1　最终的游戏界面

现在，是时候开始动手编写游戏了。

1.3　添加 HTML 和 CSS 样板代码

HTML5 文件的主要样板代码包含的内容极少，在一个位居页面中间的 container(容器)内部放置一个<canvas>元素，这样就得到了一个有效的 HTML 文件，如代码清单 1-1 所示：

代码清单 1-1：游戏的 HTML 样板代码

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8"/>
<title>Alien Invasion</title>
  <link rel="stylesheet" href="base.css" type="text/css" />
</head>
<body>
  <div id='container'>
    <canvas id='game' width='320' height='480'></canvas>
  </div>
```

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

到目前为止，仅有的两个外部文件中的一个是 base.css，这是一个外部样式表，以及一个尚不存在的文件 game.js，该文件将用来存放游戏的 JavaScript 代码。将代码清单 1-1 中的 HTML 代码存放到新目录下一个名为 index.html 的文件中。

base.css 需要包含两个独立的部分，第一部分是 CSS 重置，CSS 重置确保所有元素在所有浏览器中看上去都是一样的，任何元素自有的风格和内边距都被删除。要实现这一点，重置部分要把所有元素的大小设置为 100%(16 像素字体)，并删除所有内边距、边框和外边距。使用的重置部分是大名鼎鼎的 Eric Meyer 重置：http://meyerweb.com/eric/tools/css/reset/。

把其中的 CSS 代码一字不差地复制到 base.css 顶部即可。

接下来，需要将两种额外的样式添加到该 CSS 文件中，如代码清单 1-2 所示。

代码清单 1-2：基本画布和容器样式

```
/* Center the container */
#container {
  padding-top:50px;
  margin:0 auto;
  width:480px;
}
/* Give canvas a background */
canvas {
  background-color: black;
}
```

第一个容器样式为容器指定到页面顶端的较小内边距(padding)，在页面的中间位置居中放置容器的内容。第二个样式为画布(canvas)元素指定黑色背景。

1.4　画布入门

注意一下放在页面 HTML 代码中部的 canvas 标签(如代码清单 1-1 所示)：

```
<canvas id='game' width='320' height='480'></canvas>
```

这就是游戏所有动作发生的地方——在如此一个不起眼的标签内，竟然能够制造出这么多令人兴奋不已的东西。

除了 width 和 height 之外，该标签还有 id，以方便引用。与大多数 HTML 元素不同，一般来说，永远不要把 CSS 的 width 和 height 加到 canvas 元素上，这些样式虽调整了画布的可见大小，但不会影响画布的像素尺寸，这是通过元素的 width 和 height 进行控制的，大部分情况下你应让它们保持原样。

1.4.1　访问上下文

在能够在画布上绘制任何东西之前，需要提取 canvas 元素的上下文。该上下文是一个对象，实际上，你就是通过该对象(而非 canvas 元素自身)进行 API 调用的。就 2D 画布游戏而言，可以提取出 2D 上下文，如代码清单 1-3 所示。

代码清单 1-3：访问渲染上下文

```
var canvas = document.getElementById('game');

var ctx = canvas.getContext && canvas.getContext('2d');
if(!ctx) {
   // No 2d context available, let the user know
   alert('Please upgrade your browser');
} else {
  startGame();
}
function startGame() {
 // Let's get to work
}
```

首先，从文档中抓取元素，最初的这几章使用内置的浏览器方法进行所有的 DOM(Document Object Model，文档对象模型)交互；后面将介绍如何使用 jQuery 以更简洁的方式完成同样的操作。

接着，调用 canvas 元素的 getContext 方法，其中的短路运算符(&&)提供了保护，这样就不会调用不存在的方法。该运算被用在接下来的 if 语句中，以防访问页面的浏览器不支持 canvas 元素。这种情况下，始终要“失败得有声响”，这样玩家才能正确地了解到该责怪自己的浏览器而非你的代码。“失败得有声响”意味着不会将错误隐藏在 JavaScript 控制台中，不会被“失败得无声息”的白屏取代，游戏会显式弹出消息，告知用户出了某些问题。

桌面浏览器(除 Internet Explorer 之外)都提供了一个 3D WebGL 驱动的渲染上下文，不过它称为 glcanvas，且在本书写作之时只可用在 Nokia 移动设备上。WebGL 是另一个独立于 HTML5 的标准，允许你在浏览器中使用硬件加速的 3D 图形。

将代码清单 1-3 中的代码保存成名为 game.js 的文件，从现在开始，可以开始使用 canvas 元素设计游戏了。

1.4.2　在画布上绘制

这一初始教程不会使用任何基于矢量的绘图例程，不过为了在屏幕上快速显示一些东西，可在页面上绘制一个矩形。修改 game.js 文件中的 startGame 方法，内容如下：

```
function startGame() {
  ctx.fillStyle = "#FFFF00";
  ctx.fillRect(50,100,380,400);
}
```

为了绘制一个实心的矩形，代码使用了 ctx 对象的 fillRect 方法，不过需要先设置一种填充风格。可将标准的 CSS 颜色表示当成字符串传递给 fillStyle，这些表示可以是十六进制颜色、RGB 三元组或 RGBA 四元组。

为在已有矩形之上叠放一个半透明矩形，可添加以下代码：

```
function startGame() {
  ctx.fillStyle = "#FFFF00";
  ctx.fillRect(50,100,380,400);
  // Second, semi-transparent blue rectangle
  ctx.fillStyle = "rgba(0,0,128,0.5);";
  ctx.fillRect(0,50,380,400);
}
```

若在加入上述代码之后重新加载 index.html 文件，就会看到一个美观的蓝色大矩形正好处在黑色画布的中间位置。

1.4.3　绘制图像

Alien Invasion 是一款老派的纵向卷轴 2D 射击类游戏，用到了样子复古的位图图形。幸运的是，画布提供了一个名为 drawImage 的简单方法，该方法有着几种不同的调用形式，这取决于你是想绘制完整图像还是仅绘制部分图像。

这一过程的唯一复杂之处在于，要绘制这些图形，游戏需要首先加载图像。但这不是什么大不了的事情，因为浏览器在加载图像方面是一把好手，只不过需要以异步方式进行加载，所以你需要等待一个回调，该回调的作用就是告诉你图像已准备就绪。

确保已将 sprites.png 文件从本书第 1 章的资产目录复制到当前游戏的 images/目录下，然后将代码清单 1-4 中的代码添加到 startGame 函数的末尾处。

代码清单 1-4：使用画布绘制图像(canvas/game.js)

```
function startGame() {
  ctx.fillStyle = "#FFFF00";
  ctx.fillRect(50,100,380,400);

  // Second, semi-transparent blue rectangle
  ctx.fillStyle = "rgba(0,0,128,0.8);";
  ctx.fillRect(25,50,380,400);

  var img = new Image();
  img.onload = function() {
    ctx.drawImage(img,100,100);
  }
  img.src = 'images/sprites.png';
}
```

若重新加载页面，现在应会看到精灵表被叠放在矩形的上方。若查阅本章代码中的 canvas/game.js 文件来了解完整的代码，你会看到，在试图把图像绘制到上下文中之前，代

码会先等待 onload 回调，然后在设置回调之后设置 src。这一顺序非常重要，因为若掉转了这两行代码的顺序，图像就会被缓存；若图像被缓存，Internet Explorer 就不会触发 onload 回调。可在图 1-2 中看到绘制结果，需要承认，结果是毫无美感的。

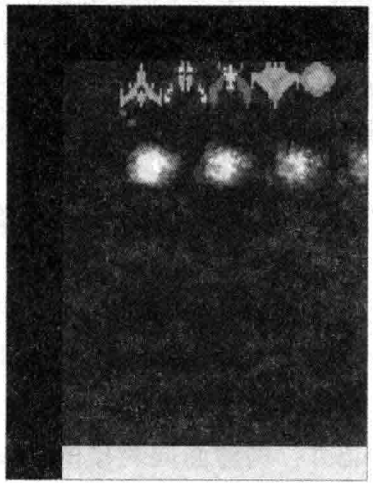


图 1-2　精灵表和绘制的矩形

第一个例子使用最简单的 drawImage 方法——接收一幅图像及 x 和 y 坐标作为参数，然后在画布上绘制出整幅图像。

修改 drawImage 所在行的代码，改后内容如下：

```
var img = new Image();
img.onload = function() {
  ctx.drawImage(img,100,100,200,200);
}
img.src = 'images/sprites.png';
```

现在图像被缩小，尺寸为另外传入的参数所指定的目标宽度和高度。这是 drawImage 的第二种调用形式，这种形式支持你按任何尺寸放大或缩小图像。

不过，drawImage 的最后一种调用形式才是位图游戏最常用的形式，这也是一种最复杂的形式，总共要接收 9 个参数：

```
drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)
```

这种形式使得你能使用参数 sx、sy、sWidth 和 sHeight 在图像中指定源矩形，以及使用参数 dx、dy、dWidth 和 dHeight 在画布上指定目标矩形。你可能已经知道，要从精灵表的某个精灵中抽取单独的帧，这就是你应该使用的格式。现在通过修改 drawImage 的调用来试用一下这种格式：

```
var img = new Image();
img.onload = function() {
  ctx.drawImage(img,18,0,18,25,100,100,18,25);
}
img.src = 'images/sprites.png';
```

若重新加载页面，现在会看到画布上只有玩家飞船的一个实例。到目前为止，一切进展顺利，下一节开始打造真正的游戏结构。

即时模式和保留模式

画布是一种以通常所说的即时模式(Immediate mode)来创建游戏的工具，在使用画布时，所有需要做的就是把像素绘制到页面上。画布不知道任何到处飞行的飞船和导弹一类的玩意儿，它所关心的一切就是像素，在帧与帧的切换之间，大部分的画布游戏都会完全清空画布，在更新后的位置重绘每样东西。

使用 DOM 创建游戏则与此相反，使用 DOM 就相当于以保留模式(Retained mode)创建

游戏，因为浏览器会自动记录下“场景图”，这一场景图跟踪对象的位置和层次。你不必从头绘制每帧图像，只需调整那些发生了变化的元素就可以了，浏览器会负责正确渲染每样东西。至于哪种模式更好，怎么说呢？这要视你的游戏而定，可参阅第 12 章中的讨论，了解何时该使用哪种模式。

1.5　创建游戏的结构

到目前为止，构建好的代码已可充当一种很好的手段，用来练习将要用到的画布功能。不过这些代码需要有个重组的过程，这样才能变成有用的游戏结构。现在回退一步，了解一些可用来整合游戏的模式。

1.5.1　构建面向对象的 JavaScript

JavaScript 是一种面向对象(Object-Oriented，OO)的语言，因此，在 JavaScript 中，包括字符串、数组、函数等在内的大部分元素都是对象。当然，这里的对象指的是 OO 意义上的对象。

但是，这并非意味着 JavaScript 拥有面向对象编程(Object-Oriented Programming，OOP)的全部特性。首先，它没有传统的继承模型；其次，它没有标准的构造机制，作为替代，它依赖于构造函数或对象字面量。

JavaScript 没有采用传统的继承做法，它实现了原型继承(prototypical inheritance)，这意味着可为一组子孙对象创建一个代表原型或蓝本的对象，该组子孙对象完全共享同样的基本功能。

传统继承和原型继承

如今用到的大多数流行的面向对象语言，包括 Java 和 C++在内，依赖的都是传统继承，这意味着对象行为是通过创建显式的类并实例化这些类的对象来定义的。JavaScript 有着一种基于原型概念定义类的较不固定的做法，这意味着先创建一个按照所希望方式行事的真实对象，然后再通过该对象创建子对象。

因为方法就是普通的 JavaScript 对象，所以在许多时候，开发者也通过简单复制其他一些对象的特性来假造 Java 风格的接口或多继承。这种灵活性未必是问题，相反，这意味着在如何创建对象以及如何为特定用例挑选最好的方法方面，开发者有着非常多的选择。

Alien Invasion 结合原型继承使用构造函数，这样做是有道理的。使用对象原型能够把对象创建的速度提升 50 倍，而且节省了内存的使用。但这种做法也会受到更多的限制，因为不能使用闭包(Closure)访问和保护数据。闭包是 JavaScript 的一项功能，该功能允许将方法中的变量保存起来以备后用，甚至在方法执行完毕之后还可使用。

第 9 章将更详细地讨论对象创建的模式，不过，就目前而言，只要明白使用不同方法是有意为之的就可以了。

1.5.2　利用鸭子类型

有一个很著名的说法是这样的：若它走起来像一只鸭子，叫起来像一只鸭子，那么它一定是一只鸭子。在使用强类型的语言编程时，毫无疑问它是一只鸭子——它必须是 Duck 类的一个实例；或者，若使用 Java 编程，就要实现 iDuck 接口。

而在 JavaScript 这一动态类型语言中，参数和引用是不做类型检查的，这意味着可把任何类型的对象当作参数传递给任何函数，该函数乐意把该对象当成其所期待的任何对象类型对待，直至有问题出现。

这种灵活性既是好事也是坏事，如果最终出现一些神秘莫测的错误消息或在运行时发生错误，这是坏事；在借助这一灵活性来保持浅继承树但又能够共享代码时，这是好事。这种基于对象的外部接口(而非它们的类型)来使用对象的概念称为鸭子类型(duck typing)。

Alien Invasion 在游戏界面和精灵这两个地方用到了这一概念，该游戏把任何响应 step() 和 draw()方法调用的事物都当成游戏界面对象或有效的精灵对待。把鸭子类型用于游戏界面，这使得 Alien Invasion 能够把标题画面和游戏中的界面当成同样的对象类型看待，简化了关卡和标题画面之间的切换。同样，把鸭子类型用于精灵意味着游戏能够灵活决定往游戏面板中添加的内容，其中包括玩家、敌人、炮弹和 HUD 元素等。HUD 是抬头显示设备(Head Up Display)的简称，这个术语常指位于游戏屏幕上方的元素，如剩余的生命条数和玩家得分等。

1.5.3　创建三个基本对象

该游戏需要用到三个几乎一直存在的基本对象：一个把所有东西捆绑在一起的 Game 对象，一个加载和绘制精灵的 SpriteSheet 对象，以及一个显示、更新精灵元素和处理精灵元素碰撞的 GameBoard 对象。该游戏还需要一大群不同的精灵，如玩家、敌方飞船、导弹及诸如得分和剩余生命条数一类的 HUD 对象等，稍后将介绍这些精灵。

1.6　加载精灵表

你已经看到，大部分的代码都需要加载精灵表并在页面上显示精灵。现在剩下要做的就是把这些功能提取出来放入一个包中，功能的一个增强之处是加入精灵名称和位置的映射表，这能把在屏幕上绘制精灵变得更容易一些。第二个增强之处是封装 onload 回调功能，对任何调用类隐藏细节。

代码清单 1-5 显示了完整的类。

代码清单 1-5：SpriteSheet 类

```
var SpriteSheet = new function() {
  this.map = { };
  this.load = function(spriteData,callback) {
    this.map = spriteData;
```

```
    this.image = new Image();
    this.image.onload = callback;
    this.image.src = 'images/sprites.png';
  };
  this.draw = function(ctx,sprite,x,y,frame) {
    var s = this.map[sprite];
    if(!frame) frame = 0;
    ctx.drawImage(this.image,
                  s.sx + frame * s.w,
                  s.sy,
                  s.w, s.h,
                  x,   y,
                  s.w, s.h);
  };
}
```

尽管这个类很短，只有两个方法，但需要注意的事情还不少。首先，因为只能有一个 SpriteSheet 对象，所以要使用以下语句来创建对象：

```
new function() { ... }
```

该语句把构造函数和 new 操作符放在同一行中，确保该类永远只会有一个实例被创建。

接着将两个参数传给构造函数，第一个参数 spriteData 把链接了精灵矩形和名称的精灵数据传递进来；第二个参数 callback 把图像 onload 方法的回调函数传递进来。

第二个方法 draw 是类的主力，因为是它真正把精灵绘制到上下文中。它接收的参数包括上下文、指定 spriteData 映射表中的精灵名称的字符串、绘制精灵的 x 和 y 位置，以及为具有多帧的精灵提供的一个可选的帧(frame)。

draw 方法使用这些参数来查找映射表中的 spriteData，以取得精灵的源位置以及精灵的宽度和高度(对于这个简单的 SpriteSheet 类来说，精灵的每一帧都被认为具有相同的尺寸并处在同一行上)。它使用这些信息计算出更复杂的 drawImage 方法要用到的参数，本章之前的 1.4.3 节中的内容已讨论过这一更复杂的 drawImage 方法。

尽管这些代码被设计成一次性的，且仅适用于这一特定的游戏，但你还是需要把诸如精灵数据和关卡一类的游戏数据与游戏引擎独立开来，这样更便于分块测试和构建。

把 SpriteSheet 添加到一个名为 engine.js 的新文件的项部，并用以下代码替换 game.js 中的 startGame 函数：

```
function startGame() {
  SpriteSheet.load({
    ship: { sx: 0, sy: 0, w: 18, h: 35, frames: 3 }
  },function() {
    SpriteSheet.draw(ctx,"ship",0,0);
    SpriteSheet.draw(ctx,"ship",100,50);
    SpriteSheet.draw(ctx,"ship",150,100,1);
  });
}
```

这里的 StartGame 函数调用 SpriteSheet.load 并传入几个精灵的详细信息，接着，在回调函数中(在加载 images/sprites.png 文件之后)测试绘制函数，在画布上绘制三个精灵。

修改 index.html 文件底部的内容，首先加载 engine.js，然后加载 game.js：

```
<body>
  <div id='container'>
    <canvas id='game' width='480' height='600'></canvas>
  </div>
  <script src='engine.js'></script>
  <script src='game.js'></script>
</body>
```

可找出本章代码中的 sprite_sheet/index.html 文件，了解一下上述例子的暂时形式是什么样子的。

现在，游戏可以在页面上绘制精灵了，可通过设置主游戏对象把其余一切串联起来。

1.7　创建 Game 对象

主游戏对象是个一次性对象，被命名为 Game 是顺理成章的事情。它的主要目的是初始化 Alien Invasion 的游戏引擎并运行游戏循环，以及提供一种机制来改变所显示的主场景。

因为 Alien Invasion 没有输入子系统，所以 Game 类还要负责设置键盘和触摸输入的监听器。开始时监听器只处理键盘输入，下一章再添加触摸输入。

现在，游戏已具雏形，这时进行一些额外的考虑很有必要。所以不要在获取代码时就随意地执行代码，一般来说，一种合理做法是在初始化游戏之前先等待页面下载完毕。Game 类已考虑到这一点，在启动游戏之前首先监听窗口的 load 事件。

Game 类的代码将被添加到 engine.js 文件的顶部。

1.7.1　实现 Game 对象

现在开始分步解说构成 Game 对象的 40 多行代码，一次讲解代码的一部分(可在本章代码文件 game_class/engine.js 的顶部查看完整的代码清单)。作为一次性的类实例，该类开始的一句与 SpriteSheet 很相似：

```
var Game = new function() {
```

接下来是初始化例程，在调用该例程时，用到的参数包括为画布元素填写的 ID、传递给 SpriteSheet 的精灵数据，以及在游戏做好启动准备时调用的回调。

```
// Game Initialization
  this.initialize = function(canvasElementId,sprite_data,callback) {

    this.canvas = document.getElementById(canvasElementId);
    this.width = this.canvas.width;
```

```
    this.height= this.canvas.height;

    // Set up the rendering context
    this.ctx = this.canvas.getContext && this.canvas.getContext('2d');

    if(!this.ctx) { return alert("Please upgrade your browser to play"); }

    // Set up input
    this.setupInput();

    // Start the game loop
    this.loop();

    // Load the sprite sheet and pass forward the callback.
    SpriteSheet.load(sprite_data,callback);
  };
```

通过本章之前的介绍，这段代码的大部分内容应不算陌生，其中抓取画布元素和检查 2d 上下文的那部分代码简单明了；然后所做的就是调用 setupInput()，该方法在接下来的内容中讨论；最后，游戏循环开始，将精灵表的数据传给 SpriteSheet.load。

接下来设置输入：

```
// Handle Input
var KEY_CODES = { 37:'left', 39:'right', 32 :'fire' };
this.keys = {};
this.setupInput = function() {
  window.addEventListener('keydown',function(e) {
    if(KEY_CODES[event.keyCode]) {
     Game.keys[KEY_CODES[event.keyCode]] = true;
     e.preventDefault();
    }
  },false);
  window.addEventListener('keyup',function(e) {
    if(KEY_CODES[event.keyCode]) {
     Game.keys[KEY_CODES[event.keyCode]] = false;
     e.preventDefault();
    }
  },false);
}
```

这个代码块的要点是为你所关心的那些按键添加 keydown 和 keyup 事件的监听器，特别是左箭头、右箭头和空格键。就这些事件而言，监听器把数值型的键值(Keycode)转换成一个更友好的标识符，并更新一个名为 Game.keys 的哈希，以此来描述用户输入的当前状态。玩家使用 Game.keys 哈希来控制飞船，就游戏使用的按键而言，事件处理程序还调用 e.preventDefault()方法，该方法防止浏览器执行任何响应按键的默认行为(对于箭头键和空格键来说，浏览器通常会试图滚动页面)。

关于上述事件处理程序的代码，还需要说明的一点是：它使用了 W3C 事件模型的

addEventListener 方法，Chrome、Safari 和 Firefox 浏览器的当前版本都提供了对这一代码的支持，但 Internet Explorer (IE)只在版本 9 及更高版本中提供支持。这不是什么大问题，因为在任何情况下，IE9 之前的版本都不支持画布，不过若你希望添加对较旧版本浏览器的兼容性，就需要多加小心(从第 9 章开始构建的引擎使用 jQuery 的 on 方法支持与浏览器无关的简单事件附加)。

Game 类的最后一部分内容相对短一些：

```
  // Game Loop
  var boards = [];
  this.loop = function() {
    var dt = 30/1000;
    for(var i=0, len = boards.length;i<len;i++) {
      if(boards[i]) {
        boards[i].step(dt);
        boards[i] && boards[i].draw(Game.ctx);
      }
    }
    setTimeout(Game.loop,30);
  };

  // Change an active game board
  this.setBoard = function(num,board) { boards[num] = board; };
};
```

其中 boards 数组中存放的是已更新并已绘制到画布上的游戏的各块内容，一个可能的面板(board)例子是背景或标题画面(下一章会为精灵的处理专门创建一个面板)。Game.loop 函数循环遍历所有面板，检查每个下标位置是否有面板。若有，则使用大致已逝去的秒数来调用面板的 step 方法，接着调用面板的 draw 方法，传入渲染上下文作为参数。对于 draw 调用而言，step 调用有可能已经删除了面板，所以要使用 boards[i] &&再次检查面板是否存在，以免代码崩溃。最后，setTimeout 被用于 loop 函数，以确保循环每 30 毫秒运行一次。使用 setTimout(而非 setInterval)确保定时器事件在游戏速度下降时不做备份，这种备份会导致奇怪的类似错位的行为。因为 setTimeout 不保留被调用函数的上下文，所以需要使用 Game.loop 这样的写法，即要显式地引用 Game 对象而非使用 this 这一关键字。

定时器方法

在 JavaScript 定时器方面，游戏开发能用到的不仅是 setTimeout 或 setInterval，甚至还有更多。第 9 章将讨论 requestAnimationFrame 方法，该方法使得浏览器能够同步游戏的调用和屏幕的更新。此外，通过传入固定值来硬编码时长，这通常是一种糟糕的做法，因为根据浏览器性能的不同，可能需要以不同时间间隔来调用定时器，不过对于此类简单游戏来说，这样做应该没问题。

因为各面板是按从下标 0 开始直至最大下标这样的顺序放置的，所以应把背景面板(比如下一节中的星空)添加至较低的下标位置，而诸如得分和 HUD 一类被加在末端的元素则

应最后绘制。

最后定义的是 Game 对象唯一的、在游戏期间被定期调用的方法 Game.setBoard，该方法所要做的就是设置 loop 方法用到的一个游戏面板。它用来切换活动的 GameBoard，GameBoard 用于标题画面及游戏的主体部分。

1.7.2　重构游戏代码

在浏览器中构建游戏时，应该持续关注正在构建的代码的结构。JavaScript 是一种非常灵活的语言，没有什么准则可用来规范游戏的结构化做法，构建好的东西可能很快就分崩离析。本书常用的一种模式会向你展示如何先快速简单地使用一个 API 或一项技术，然后花一些时间把代码结构化成库或模块。

game.js 中用于在屏幕上显示精灵的初始代码将被替换成能够完成同样任务的结构化代码，这种以某种方式结构化的代码能用在更复杂的游戏中。

更新 game.js，在代码中使用 Game 类，删除 game.js 中的所有内容，然后添加代码清单 1-6 所示的代码。

代码清单 1-6：重构后的 game.js 方法(game_class/game.js)

```
var sprites = {
 ship: { sx: 0, sy: 0, w: 18, h: 35, frames: 3 }
};
var startGame = function() {
   SpriteSheet.draw(Game.ctx,"ship",100,100,1);
}
window.addEventListener("load", function() {
  Game.initialize("game",sprites,startGame);
});
```

这段代码所做的就是设置一些可用的精灵、创建哑函数 startGame，该函数先在画布上绘制一艘飞船，以确保一切运行正常；然后监听窗口对象的加载事件，用适当的参数调用 Game.initialize 函数。

重新加载 index.html 文件(或运行代码例子 game_class/index.html)，你会看到一艘飞船孤零零地出现在画布元素上。

1.8　添加滚动背景

你正迫不及待地等待着一些比样板设置代码更有趣的事情出现，对吗？这里就有个好消息：从现在开始，事情会变得越来越有趣。我们先把动画星空添加到页面上，赋予游戏某些类似太空的效果。

可采用多种方法创建滚动的星空，不过在这个例子中，你需要留意绘制到屏幕上的对象的数目，因为每帧绘制太多精灵会降低游戏在移动设备上的运行速度。一种解决方法是

创建画布的离屏缓冲区，在缓冲区中随机绘制一堆星星，然后简单地绘制慢慢向下移过画布的星空。你只能用到有限几个不同的移动星星层，不过对于一个复古的射击类游戏来说，这一效果已经足够好了。

变幻莫测的 HTML5 性能

性能问题并不简单，HTML5 的真理之一是，在没有试过的情况下，你永远也不知道哪种做法的性能更好。在决定采用哪种方法来实现某个功能时，最好的选择是直接找出第一手资料，即对方法进行测试！通过页面 http://jsperf.com/prerendered-starfield，可以了解到与不同数量的星星和星空的不同绘制方法所对应的性能，要测试你的直觉正确与否，JSPerf.com 是一个非常不错的地方。要查看星空测试的结果，向下滚动页面，然后单击 Run Tests 按钮了解不同运行过程的性能。这个例子的答案没有表现得千篇一律，至少在撰写本书之时，大多数桌面在绘制各颗星星时表现更好，而 iOS 移动设备在绘制离屏缓冲区时性能更佳。由于在不久的将来，画布将全面获得更好的硬件加速，因此，一种看起非常可能的情况是，在未来数月乃至数年内，(如本节所介绍的)填充率受限的离屏缓冲区的速度将大幅提升。

现在，在把类当成整体对待之前，先对代码做一些必要的分块解说(若想提前查看效果，可跳至本节结尾处查看完整的类)。

StarField 类需要完成的事情主要有三项，第一项是创建离屏画布，这实际上相当简单，因为画布就是一个普通的 DOM 元素，具有两个特性 width 和 height，可以用与其他任何 DOM 元素都相同的方式进行创建：

```
var stars = document.createElement("canvas");
stars.width = Game.width;
stars.height = Game.height;
var starCtx = stars.getContext("2d");
```

因为星空需要拥有与游戏画布同等大小的尺寸，所以可通过抽取在 Game.initialize 方法中设置的 width 和 height 属性来设置星空的大小。

创建画布后，可以开始在画布上绘制星星或矩形。要实现这一点，最简单的做法是为需要绘制的每颗星星调用一次 fillRect。一个 for 循环加上 Math.random()的使用，可以生成随机的 x 和 y 位置，任务就此完成：

```
starCtx.fillStyle = "#FFF";
starCtx.globalAlpha = opacity;
for(var i=0;i<numStars;i++) {
  starCtx.fillRect(Math.floor(Math.random()*stars.width),
                   Math.floor(Math.random()*stars.height),
                   2,
                   2);
}
```

上述代码中唯一没有提及的部分是 globalAlpha 属性，该属性设置 canvas 元素的不透

明度。因为有多层星星以不同的速度移动，所以要获得好的效果，就得让移动较慢的星星比移动较快的那些亮度稍暗一些，以此来模拟它们的逐渐远去效果。

接下来是 draw 方法，Starfield 需要绘制整个 canvas 元素，其中包含了游戏画布上的星星；然而，因为它要不断地滚动，所以需要绘制两次：一次绘制上半部，一次绘制下半部。该方法用到了星空的偏移量(offset)，这是零到游戏高度之间的一个数值，方法使用该数值首先将任何已经移出游戏底部的星空部分绘制回顶部，然后再绘制底部。

```
this.draw = function(ctx) {
    var intOffset = Math.floor(offset);
    var remaining = stars.height - intOffset;
    if(intOffset > 0) {
      ctx.drawImage(stars,
                0, remaining,
                stars.width, intOffset,
                0, 0,
                stars.width, intOffset);
    }
    if(remaining > 0) {
      ctx.drawImage(stars,
              0, 0,
              stars.width, remaining,
              0, intOffset,
              stars.width, remaining);
    }
  }
```

这段代码看起来略显混乱，因为用到 9 个参数版本的 drawImage 方法来绘制各分段，但实际上它只将星空划分成了上半部和下半部，然后在游戏画布的底部绘制星空的上半部，以及在画布顶部绘制星空的下半部。

代码清单 1-7 给出了 Starfield 类的完整代码，这部分代码应放在 game.js 文件中。

代码清单 1-7：Starfield(starfield/game.js)

```
var Starfield = function(speed,opacity,numStars,clear) {

  // Set up the offscreen canvas
  var stars = document.createElement("canvas");
  stars.width = Game.width;
  stars.height = Game.height;

  var starCtx = stars.getContext("2d");
  var offset = 0;

  // If the clear option is set,
  // make the background black instead of transparent
  if(clear) {
    starCtx.fillStyle = "#000";
```

```
    starCtx.fillRect(0,0,stars.width,stars.height);
  }
  // Now draw a bunch of random 2 pixel
  // rectangles onto the offscreen canvas
  starCtx.fillStyle = "#FFF";
  starCtx.globalAlpha = opacity;
  for(var i=0;i<numStars;i++) {
    starCtx.fillRect(Math.floor(Math.random()*stars.width),
                     Math.floor(Math.random()*stars.height),
                     2,
                     2);
  }
  // This method is called every frame
  // to draw the starfield onto the canvas
  this.draw = function(ctx) {
    var intOffset = Math.floor(offset);
    var remaining = stars.height - intOffset;
    // Draw the top half of the starfield
    if(intOffset > 0) {
      ctx.drawImage(stars,
                0, remaining,
                stars.width, intOffset,
                0, 0,
                stars.width, intOffset);
    }
    // Draw the bottom half of the starfield
    if(remaining > 0) {
      ctx.drawImage(stars,
                0, 0,
                stars.width, remaining,
                0, intOffset,
                stars.width, remaining);
    }
  }
  // This method is called to update
  // the starfield
  this.step = function(dt) {
    offset += dt * speed;
    offset = offset % stars.height;
  }
}
```

上述代码中，仅有两部分内容尚未讨论，其中末尾处的 step 函数每隔几十毫秒就调用一次，该函数需要做的就是基于流逝的时间和速度更新 offset 变量，然后使用取模(%)运算符来确保 offset 的值位于零和 Starfield 的高度之间。

此外，还有一个条件检查 clear 参数是否已经设置，该参数的作用是使用黑色填充星星的第一层(后面的各层必须是透明的，这样它们才能正确地互相覆盖)。这种做法省却了在帧与帧之间显式地清除画布的必要性，节省了一些处理时间。

要查看星空的作用情况，需要修改 game.js 中的 startGame 函数，添加一些星空。修改该函数，通过如下设置，加入三个具有各种不透明度的星空：

```
var startGame = function() {
  Game.setBoard(0,new Starfield(20,0.4,100,true))
  Game.setBoard(1,new Starfield(50,0.6,100))
  Game.setBoard(2,new Starfield(100,1.0,50));
}
```

其中只有第一个星空把 clear 参数设置为 true，每个星空都比前一个的速度更高而且具有更高的不透明度，这赋予了星星处于远近不同距离中、正在不停掠过的效果。

1.9　插入标题画面

动画星空尽管不错，但还不算是一个游戏。对于游戏来说，若要开始打造前面提到的那些游戏元素，首先要做的事情之一是显示一个标题画面，向用户展示他们可以玩的东西。

Alien Invasion 的标题画面没什么特别之处——就是一个文本标题和一个副标题而已。所以，一个通用的 GameScreen 类加上放在屏幕中间位置的标题和副标题就足以满足需要了。

在画布上绘制文本

在画布上绘制文本很简单，允许使用任何已加载到页面上的字体。这一灵活性意味着可以使用任何标准的 Web 安全字体，以及任何已通过@font-face 加载到页面上的字体。

在声明@font-face 时要小心一些，因为根据必须支持的浏览器的不同，需要提供 4 种不同的文件格式。幸运的是，若你不打算在本地安装这些文件，而是把它们当成诸如免费的 Google Web 字体之类的在线服务的话，那么所用到的就是一个链接的样式表而已(可在 www.google.com/webfonts 上浏览可免费使用的 Google Web 字体)。

就 Alien Invasion 而言，Bangers 字体赋予了游戏一种很好的复古风格，很有电影“人体入侵者(Invasion of the Body Snatchers)”的感觉。把下面一行代码加入到 HTML(而非 JavaScript)脚本中，放在 bass.css 链接标签之后：

```
<head>
  <meta charset="UTF-8"/>
  <title>Alien Invasion</title>
  <link rel="stylesheet" href="base.css" type="text/css" />
  <link href='http://fonts.googleapis.com/css?family=Bangers'
   rel='stylesheet' type='text/css'>
</head>
```

下一步，游戏需要 TitleScreen 类在屏幕中间显示一些文本。要实现这一点，必须使用一个新的、之前尚未讨论过的画布方法—— fillText，以及两个新的画布属性—— font 和 textAlign。

当前使用的字体则通过给 context.font 传入一个 CSS 样式进行设置，例如：

```
ctx.font = "bold 25px Arial";
```

这一声明把 measureText 和 fillText 当前都要用到的字体设置为 25 像素高、粗体，使用 Arial 字体系列。

为确保字体水平居中在某个特定位置，需要把 context.textAlign 属性设置为 center。

```
ctx.textAlign = "center";
```

在计算文本位置并设置适当的字体风格后，就可以使用 fillText 在画布上绘制实心文本了：

```
fillText(string, x, y);
```

fillText 接收的参数有要绘制的字符串以及左上角的 x 和 y 位置。

有了这些文本绘制方法，现在就拥有了绘制标题画面的工具，该标题画面显示一个标题和一个副标题，并在用户按下发射键时调用一个可选的回调。

代码清单 1-8 给出了完成这部分工作的代码，将 TitleScreen 类添加到 engine.js 文件的底部。

代码清单 1-8：TitleScreen(titlescreen/engine.js)

```
var TitleScreen = function TitleScreen(title,subtitle,callback) {
  this.step = function(dt) {
    if(Game.keys['fire'] && callback) callback();
  };
  this.draw = function(ctx) {
    ctx.fillStyle = "#FFFFFF";
    ctx.textAlign = "center";

    ctx.font = "bold 40px bangers";
    ctx.fillText(title,Game.width/2,Game.height/2);

    ctx.font = "bold 20px bangers";
    ctx.fillText(subtitle,Game.width/2,Game.height/2 + 40);

};
```

与 Starfield 对象类似，TitleScreen 定义了 step 和 draw 方法。step 方法只有一项任务，那就是检查发射键是否被按下。若是，则调用传递进来的回调函数。

draw 方法真正完成了大部分工作。首先，它设置了一个将被标题和副标题使用的 fillStyle(白色)，然后设置了标题的字体。可通过把 x 移至画布一半宽度的位置在水平方向上居中显示标题，接着要做的就是使用这一计算好的 x 坐标和画布高度的一半来调用 fillText。

为了绘制副标题，方法使用一种新的字体来重复一遍前面提到的计算，然后在垂直位

置上偏移 40 像素，把副标题放在标题的下方。

现在需要把标题画面作为背景星空之上的新面板添加到页面上，修改 startGame 方法，如下所示，加入名为 playGame 的新回调函数：

```
var startGame = function() {
  Game.setBoard(0,new Starfield(20,0.4,100,true))
  Game.setBoard(1,new Starfield(50,0.6,100))
  Game.setBoard(2,new Starfield(100,1.0,50));
  Game.setBoard(3,new TitleScreen("Alien Invasion",
                                  "Press space to start playing",
                                  playGame));
}

var playGame = function() {
  Game.setBoard(3,new TitleScreen("Alien Invasion", "Game Started..."));
}
```

若重新加载浏览器，你应会看到标题画面，在按下空格键之后，标题画面应把副标题更新成“Game Started”。在下一节中，playGame 函数的内容会被真正开始游戏过程的代码替代。

1.10　添加主角

把 Alien Invasion 变成一款真正可玩的游戏的第一步是添加一艘由玩家控制的飞船，这是你加入游戏的第一个精灵。在下一章中，你将创建 GameBoard 类来同时管理正常游戏过程中出现在页面上的诸多精灵，不过就目前来讲，开始时使用一个精灵就足够了。

1.10.1　创建 PlayerShip 对象

第一步是创建一艘飞船并把它绘制到页面上，打开文件 game.js，然后把 PlayerShip 类添加到文件末尾处：

```
var PlayerShip = function() {
  this.w =  SpriteSheet.map['ship'].w;
  this.h =  SpriteSheet.map['ship'].h;
  this.x = Game.width/2 - this.w / 2;
  this.y = Game.height - 10 - this.h;
  this.vx = 0;
  this.step = function(dt) {
    // TODO - added the next section
}
  this.draw = function(ctx) {
    SpriteSheet.draw(ctx,'ship',this.x,this.y,1);
  }
}
```

与游戏画面类似，该精灵有着同样的两个外部方法：step 和 draw。保持接口的一致可让精灵和游戏画面有着最大程度的可互交换性。在初始化精灵时，通过设置几个变量来指定精灵在页面上的位置及精灵的高度和宽度(下一章将使用位置及高度和宽度来做一些简单的边框碰撞检测)。

从精灵表中抽取精灵的宽度和高度，虽然可以在这个地方硬编码宽度和高度，但是使用来自精灵表的尺寸就意味着，要改变尺寸的话，只需修改一个地方的代码即可。

接下来，把 playGame 函数修改成如下内容：

```
var playGame = function() {
  Game.setBoard(3,new PlayerShip());
}
```

若重新载入 index.html 文件并按下空格键，你就可以看到玩家飞船悬停在页面底部。

1.10.2　处理用户输入

接下来的任务是接受用户输入，允许玩家在游戏画面上来回移动飞船，这一过程在 PlayerShip 内部的 step 函数中实现。

step 函数包含三个主要部分，第一部分检查用户输入，更新飞船的移动方向；第二部分基于方向更新 x 坐标；最后，函数需要检查更新后的 x 位置是否位于屏幕界面内。用以下代码替换前面 step 方法中的 TODO 注释：

```
this.step = function(dt) {
  this.maxVel = 200;
  this.step = function(dt) {
    if(Game.keys['left']) { this.vx = -this.maxVel; }
    else if(Game.keys['right']) { this.vx = this.maxVel; }
    else { this.vx = 0; }

    this.x += this.vx * dt;

    if(this.x < 0) { this.x = 0; }
    else if(this.x > Game.width - this.w) {
      this.x = Game.width - this.w;
    }
  }
}
```

方法的第一部分检查 Game.keys 映射表，了解用户当前是否按下了左箭头或右箭头键，若是，则把速度设置成正确的正值或负值。代码的第二部分简单使用当前速度乘以自上次更新后过去的几分之一秒所得的值来更新 x 的位置。最后，方法查看 x 的位置是否超出了屏幕的左侧(小于零)或右侧(大于屏幕的宽度减去飞船的宽度)，若这两个条件之一为真，则 x 的值被修改成一个处于这一范围之内的值。

1.11　小结

到目前为止，你已经了解到如何搭建 HTML5 游戏的框架和运行游戏，这其中包括了加载精灵表、在画布上进行绘制、加入视差背景以及接收用户输入等。现在，可以通过浏览 player/index.html 文件来启动游戏，使用箭头键左右控制飞船的移动。祝贺你！在创建自己的第一个可运行 HTML5 游戏的过程中，你已经迈出了非常成功的一步。下一章基于这些初始代码继续构建游戏，为游戏添加敌人、关卡及其他一些内容。第 3 章通过加入移动支持来完成这个初始游戏的开发。

第 2 章

从玩具到游戏

本章提要

- 探讨场景管理
- 添加炮弹和敌方飞船
- 使用碰撞检测
- 制造爆炸

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 2 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

2.1 引言

在第 1 章中，你搭建了自己的第一款 HTML5 移动游戏的框架，实现了飞船在屏幕四周飞动。但到目前为止，所构建出来的东西与其说是游戏，倒不如说是玩具。要把它变成游戏，你还需要添加一些敌方飞船，并设置各种游戏元素，这样才能让敌我双方交战。

2.2 创建 GameBoard 对象

把 Alien Invasion 变成游戏的第一步是添加一种机制来同时处理页面上的一群精灵。目前的 Game 对象可以处理一摞面板(board)，不过这些面板彼此之间的行为是完全独立的。

此外，尽管 Game 对象提供了一种切换面板进出的机制，但在把任意数量的精灵添加至页面这方面，并未提供什么简易的做法。所以，我们要把 GameBoard 对象加进来。

2.2.1　了解 GameBoard 对象

GameBord 对象的作用更像是跳棋游戏中的游戏棋盘，它提供放置所有组件的场所，然后指明它们的移动方式，本节内容分类列出该对象的一些职责。GameBoard 的职责可分为 4 种不同类型：

- 负责保存一个对象列表，以及把精灵添加到列表中及从列表中删除精灵。
- 此外，它还需要遍历该对象列表。
- 需要以与之前的面板相同的方式来进行响应，它必须拥有 step 和 draw 函数，这两个函数会调用对象列表中每个对象的相应方法。
- 需要检测对象之间的碰撞。

接下来的几节内容将详细讲解 GameBoard 对象的各个部分，该对象的行为类似于简单的场景图。关于场景图，第 12 章中进行了详细讨论。GameBoard 类将被添加至 engine.js 文件的末尾处。

2.2.2　添加和删除对象

GameBoard 类的第一种也是最重要的一种职责是把游戏中的对象记录在案。记录对象列表的最简单做法就是使用数组，这个例子使用了名为 objects 的数组。

对 GameBoard 类的讨论是逐段进行的，但完整代码已被放到 engine.js 文件的末尾处：

```
var GameBoard = function() {
  var board = this;
  // The current list of objects
  this.objects = [];
  this.cnt = [];
```

代码中的这个数组就是添加和删除出现在游戏中的那些对象的地方。

下一步需要为该类提供添加对象的功能，这再简单不过了，把对象压入 objects 列表的末端，这项工作几乎就算完成了：

```
  // Add a new object to the object list
  this.add = function(obj) {
    obj.board=this;
    this.objects.push(obj);
    this.cnt[obj.type] = (this.cnt[obj.type] || 0) + 1;
    return obj;
  };
```

不过，对于一个要与其他对象进行交互的对象来说，它需要访问其所属的面板。为此，在 GameBoard.add 被调用时，面板为对象设置了名为 board 的属性。现在，对象可以通过访问面板来添加其他一些对象，如炮弹或爆炸对象的，或是在死去时删除自身。

此外，面板还必须保存计数，用于记录在某一给定时间不同类型的活动对象的数目。所以，该函数的倒数第二行代码使用布尔或运算符(||)在需要时把计数初始化为零，然后通过加 1 递增计数。只有到了本章后面的内容中，才会为对象指定类型，所以这是一行有点前瞻性的代码。

接下来是删除，这一过程比起初设想的要复杂一些，因为对象可能想要删除自身，或是删除 GameBoard 遍历对象列表这一步骤途中涉及的其他对象。没有考虑周全的实现会试图更新 GameBoard.objects，但是，因为 GameBoard 可能正处于遍历所有对象的过程中，所以在遍历的中途阶段对它们进行更改可能会给遍历代码带来一些问题。

一种可选做法是在开始每帧时创建对象列表的一个副本，但这样会给每一帧的绘制带来一些花销。最好首先在一个单独的数组中标记出要删除的对象，然后在遍历完所有对象之后再从对象列表中删除它们。以下是 GameBoard 使用的解决方案：

```
// Mark an object for removal
  this.remove = function(obj) {
    var wasStillAlive = this.removed.indexOf(obj) != -1;
    if(wasStillAlive) { this.removed.push(obj); }
    return wasStillAlive;
  };

  // Reset the list of removed objects
  this.resetRemoved = function() { this.removed = []; }

  // Remove objects marked for removal from the list
  this.finalizeRemoved = function() {
    for(var i=0,len=this.removed.length;i<len;i++) {
      var idx = this.objects.indexOf(this.removed[i]);
      if(idx != -1) {
        this.cnt[this.removed[i].type]--;
        this.objects.splice(idx,1);
      }
    }
  }
```

每次执行 step 时，首先会调用 resetRemoved，该方法用来重置要删除对象的列表。删除方法(remove)首先检查某个对象是否已被删除，然后，只有在对象未被放在要删除对象列表中时，才把它加到该列表中；接下来，若对象已被添加，则返回 true，若对象已死，则返回 false。遍历完所有对象后，会调用 finalizeRemoved 方法，该方法使用 Array.indexOf 查找对象列表中已被删除的对象，然后使用 Array.splice 方法从该列表中删去这些对象。从列表中删除后，对象实际上就相当于已死亡，因为它的 step 和 draw 方法将不会再被调用。

2.2.3　遍历对象列表

因为 GameBoard 所做的大部分工作是遍历对象列表，所以使用一两个辅助方法来简化这一做法也是合情合理的事情。这里需要的方法主要有两个，第一个是简单的遍历方法

iterate，该方法调用对象列表中的每个对象的同一个函数，这对 step 和 draw 方法很有用。第二个方法 detect 返回首个能让被传进去的函数返回 true 值的对象，该方法简化了碰撞检测。下面列出这两个方法。

首先是 iterate：

```
// Call the same method on all current objects
this.iterate = function(funcName) {
  var args = Array.prototype.slice.call(arguments,1);
  for(var i=0,len=this.objects.length;i<len;i++) {
    var obj = this.objects[i];
    obj[funcName].apply(obj,args)
  }
};
```

虽然主要作用仅是遍历 this.objects，不过这个方法的确用到了一两个有趣的 javaScript 功能。

方法的第一行代码就是一种为人熟知的 JavaScript 黑客手法，其中的 arguments 对象在每个方法调用中都是可用的，它包含了被传入到方法中的参数的列表，可在接受各种不同数量的参数的方法中使用。arguments 的行为在许多方面类似数组，但它不是真正的数组。这很可惜，因为在这个例子中，你想要做的是取出除了第一个参数之外的所有参数，第一个参数是 funcName，这样就可以把它们继续传递给每个对象的被调用函数。arguments 没有分片方法，不过，因为 JavaScript 支持使用 call 或 apply 来获得方法并把它们应用在任何对象上，所以以下这行代码：

```
var args = Array.prototype.slice.call(arguments,1);
```

所做的正是这样的事情，它把 arguments 对象变成从第二个元素开始的一个真正的数组。在循环内部，代码使用方括号运算符来查找对象属性中的方法，然后调用 apply 来使用传入的任何参数调用该方法。

接下来是检测方法，该方法将用在后面的碰撞检测中。它的工作是针对某个面板的所有对象运行同一个函数，然后返回能够让该函数返回 true 值的第一个对象。理论上，这种做法似乎没有太大用处，但若需要基于某些特定参数来进行碰撞检测或是查找某个具体对象，那么该检测方法就会很有用。

```
// Find the first object for which func is true
this.detect = function(func) {
  for(var i = 0,val=null, len=this.objects.length; i < len; i++) {
    if(func.call(this.objects[i])) return this.objects[i];
  }
  return false;
};
```

detect 方法包含了一个遍历对象的循环和一个对被传递进来的函数的调用，该调用使用的参数是被当成 this 上下文传入的对象。若该函数返回 true 值，则该对象被返回；否则，

在遍历完所有要进行比较的对象之后，检测方法返回 false 值。

2.2.4　定义面板的方法

接下来是两个标准的面板函数：step 和 draw。这两个方法的用法已经定义，它们自身的定义则很简单：

```
// Call step on all objects and then delete
// any objects that have been marked for removal
this.step = function(dt) {
  this.resetRemoved();
  this.iterate('step',dt);
  this.finalizeRemoved();
};

// Draw all the objects
this.draw= function(ctx) {
  this.iterate('draw',ctx);
};
```

step 和 draw 都使用 iterate 方法来调用列表中每个对象的某个具体命名函数，其中的 step 还保证被删除项列表的重置和最终删除。

2.2.5　处理碰撞

GameBoard 职责范围内的最后一项功能是处理碰撞。Alien Invasion 使用一种简化的碰撞模型，该模型把面板上的每个精灵都简化成一个简单的矩形边框，若两个不同对象的边框有重叠，则视为这两个精灵撞到了一起。因为除了宽度和高度之外，每个精灵还有 x 和 y 位置，所以精灵的边框很容易计算出来。

注意：边框是覆盖整个对象的最小矩形，使用边框(而非多边形或确切的像素数据)来进行碰撞检测，计算速度会更快一些，但准确性相应降低一些。

GameBoard 使用两个函数来处理碰撞检测，第一个函数 overlap 简单检查两个对象的边框的覆盖情况，若它们相交，则返回 true 值。进行这一检测的最简单做法很巧妙，不用检查一个对象是否进入了另一个对象所在的位置，只需检查一个对象是否并未进入另一个对象所在的位置，然后对结果取反就可以了。

```
this.overlap = function(o1,o2) {
   return !((o1.y+o1.h-1<o2.y) || (o1.y>o2.y+o2.h-1) ||
            (o1.x+o1.w-1<o2.x) || (o1.x>o2.x+o2.w-1));
};
```

这段代码所做的事情就是，比较对象 1 的下边框和对象 2 的上边框所在位置，查看对

象 1 是否位于对象 2 的上方；接着，比较对象 1 的上边框和对象 2 的下边框所在位置，如此一直比较完所有相应的边框。若这其中的任何一项比较值为真，就可以知道对象 1 并未与对象 2 有重叠的地方，然后通过简单地对这一检测结果取反，就能断定两个对象是否有重叠。

有了这个用来判断重叠的函数，检查一个对象和列表中其他所有对象的碰撞情况就变成了一件很容易的事情。

```
this.collide = function(obj,type) {
  return this.detect(function() {
    if(obj != this) {
     var col = (!type || this.type & type) && board.overlap(obj,this)
     return col ? this : false;
    }
  });
};
```

collide 使用 detect 函数来匹配传进来的对象和其他所有对象，然后返回第一个能够让 overlap 函数返回 true 值的对象。这其中的唯一复杂之处是支持一个可选的类型参数，这一复杂之处背后的想法是，不同类型的对象只应与某些对象碰撞，比如敌方飞船就不应和自己一方的飞船碰撞，但它们应可以与玩家和玩家发射的导弹碰撞。通过执行按位与(AND)运算，在不必因查找数组或哈希表而降低速度的情况下，代码就能完成针对多种对象类型的碰撞检查。不过这里需要说明一点，即每种不同类型必须是 2 的幂，这样才能避免不同类型的互相覆盖。

例如，若一些类型的定义如下：

```
var OBJECT_PLAYER = 1,
    OBJECT_PLAYER_PROJECTILE = 2,
    OBJECT_ENEMY = 4,
    OBJECT_ENEMY_PROJECTILE = 8;
```

那么敌方飞船就可以通过两种类型的按位或(OR)运算来检查是否与玩家或玩家发射的导弹发生了碰撞：

```
board.collide(enemy, OBJECT_PLAYER | OBJECT_PLAYER_PROJECTILE)
```

对象也可以被赋予多种类型，这种情况下，collide 函数依然能按计划正常工作。

有了这一函数，GameBoard 类就算完成了，可参阅本章的代码文件 gameboard/engine.js 了解该对象的完整版本。

2.2.6　将 GameBoard 添加到游戏中

随着 GameBoard 类的完成，下一步就是把它添加到游戏中，快速修改一下 game.js 中的 playGame 函数就能做到这一点：

```
var playGame = function() {
```

```
  var board = new GameBoard();
  board.add(new PlayerShip());
  Game.setBoard(3,board);
}
```

重新载入 index.html 文件，你所看到的游戏行为应与第 1 章结束时的一模一样。所有已做的事情就是使用 GameBoard 来负责飞船精灵的管理。这仍达不到完善的地步，因为到目前为止，游戏还没有很好地利用 GameBoard 类，这是因为游戏中只有一个精灵，这一点在下一节会有所改进。

2.3　发射导弹

现在，飞船只能在屏幕上左右来回飞动，是时候让玩家参与进来了。接下来要做的事情是绑定空格键，让它发射一对炮弹。

2.3.1　添加炮弹精灵

赋予玩家一些破坏性能力的第一步是为玩家的导弹对象创建蓝本，无论何时，只要玩家按下发射键，该对象就会被添加到游戏中，出现在玩家所在的位置。

PlayShip 对象不会使用对象原型来创建方法，因为一般来说，在游戏中，某个时刻只会有一位玩家存在，所以没必要优化对象的创建速度或是内存占用量。相反，在整个游戏通关过程中，将会有许多 PlayerMissile 被添加到游戏中，所以正确的做法是确保它们能被快速创建并且占用较少的内存(JavaScript 垃圾收集器可能会在游戏性能方面导致一些可以察觉的短暂停顿，所以简化它的工作会是你的最大兴趣之一)。基于 PlayerMissile 对象的创建频率，对象原型的使用变得意义重大了起来。为对象原型创建的函数只需创建一次，之后它们就会被保存到内存中。

把以下突出显示的文本添加到 game.js 顶部，插入导弹的精灵定义(别忘了前一行的逗号)：

```
var sprites = {
  ship: { sx: 0, sy: 0, w: 37, h: 42, frames: 1 },
  missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 }
};
```

接着，把完整的 PlayerMissile 对象(见代码清单 2-1)追加到 game.js 的末尾处。

代码清单 2-1：PlayerMissile 对象

```
var PlayerMissile = function(x,y) {
  this.w = SpriteSheet.map['missile'].w;
  this.h = SpriteSheet.map['missile'].h;
  // Center the missile on x
  this.x = x - this.w/2;
  // Use the passed in y as the bottom of the missile
```

```
  this.y = y - this.h;
  this.vy = -700;
};

PlayerMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  if(this.y < -this.h) { this.board.remove(this); }
};

PlayerMissile.prototype.draw = function(ctx)  {
  SpriteSheet.draw(ctx,'missile',this.x,this.y);
};
```

一开始用到的 PlayMissile 类的初始版本只有区区的 14 行代码，且大部分内容都是之前见过的样板代码。其中的构造函数简单设置精灵的几个属性，从 SpriteSheet 中抽取宽度和高度。因为玩家是从炮塔所在位置纵向向上发射导弹的，所以构造函数使用传入的 y 坐标作为导弹的底部位置，通过减去导弹的高度来确定导弹的起始 y 坐标。此外，函数还通过减去精灵宽度的一半在传入的 x 坐标处居中显示导弹。

如前所述，step 和 draw 方法被放在原型中创建，这样更高效。因为玩家的导弹在屏幕上只会垂直向上移动，所以 step 函数只需调整 y 属性，以及查看导弹是否已在 y 方向上完全移出屏幕。若导弹已经移出屏幕超过一个身高(一个导弹的高度，也即 this.y < -this.h)，它就会从面板中删除自身。

最后是 draw 方法，该方法仅使用 SpriteSheet 对象在导弹的 x 和 y 位置绘制导弹精灵。

2.3.2　连接导弹和玩家

若要真正把导弹放在屏幕上显示，PlayerShip 需要有所更新，以便能够响应发射键以及把两枚导弹添加到屏幕上。之所以是两枚，是因为飞船有两个炮塔，一个炮塔发射一枚导弹。另外，还需要加入重装弹的时长来限制导弹的发射速度。

为插入这一限制，你必须添加一个名为 reload 的新属性，该属性表示在下一对导弹能被发射之前的剩余时间。另外，还需添加另一个名为 reloadTime 的属性，该属性表示完整的重装弹时长。请把以下两行初始化代码添加到 PlayerShip 构造函数的顶部：

```
var PlayerShip = function() {
  this.w =  SpriteSheet.map['ship'].w;
  this.h =  SpriteSheet.map['ship'].h;
  this.x = Game.width / 2 - this.w / 2;
  this.y = Game.height - 10 - this.h;
  this.vx = 0;
  this.reloadTime = 0.25;  // Quarter second reload
  this.reload = this.reloadTime;
```

其中 reload 的值被设置成 reloadTime，这是为了避免玩家在按下发射键开始游戏时立刻发射导弹。

接下来，把 step 方法修改成如下内容：

```
this.step = function(dt) {
  if(Game.keys['left']) { this.vx = -this.maxVel; }
  else if(Game.keys['right']) { this.vx = this.maxVel; }
  else { this.vx = 0; }
  this.x += this.vx * dt;

  if(this.x < 0) { this.x = 0; }
  else if(this.x > Game.width - this.w) {
    this.x = Game.width - this.w
  }

  this.reload-=dt;
  if(Game.keys['fire'] && this.reload < 0) {
    Game.keys['fire'] = false;
    this.reload = this.reloadTime;
    this.board.add(new PlayerMissile(this.x,this.y+this.h/2));
    this.board.add(new PlayerMissile(this.x+this.w,this.y+this.h/2));
  }
}
```

这段代码在飞船的左右两侧添加两枚新的玩家导弹，前提是玩家按下了发射键且当时并未处在重装弹过程中。导弹的发射简单体现为把导弹添加到面板的正确位置，reload 属性也会被重置成 reloadTime，以此在导弹的两次发射之间加入延时。为了确保玩家必须首先按下再松开空格键来进行发射，而不能仅是按住发射键不放，该键被设置成 false 值(这不太能达到预期效果，因为 keydown 事件会被重复触发)。

重新加载游戏(或访问 http://mh5gd.com/ch2/missiles/)，测试一下导弹的发射情况。可调整 reloadTime，看看游戏使用不同速度发射导弹的效果如何。

2.4　添加敌方飞船

若是没有敌人，太空射击游戏就没有什么好玩的了，所以下一步你要通过创建 Enemy 精灵类来把一些敌方飞船加到游戏中。尽管会存在多种类型的敌方飞船，但它们都用同一个类进行表示，只是通过不同的图像和移动模板加以区分。

2.4.1　计算敌方飞船的移动

敌方飞船的移动方式使用一个包含了几个可插入参数的公式来定义，这种做法既能让敌方飞船表现出相对复杂的行为，又不必编写大量代码。该公式设置敌方飞船自被加入到面板中以来某个给定时刻的移动速度：

```
vx = A + B * sin(C * t + D)
vy = E + F * sin(G * t + H)
```

其中从 A 到 H 的所有字母代表了一些常量，可别让这些公式吓到了，它们所表达的意思就是敌方飞船的速度基于一个常量加上一个周期性重复的值(使用一个正弦函数来支持

周期性的值)。诸如此类公式的使用使得游戏能够添加一些以有趣模式在屏幕各处旋转移动的敌方飞船，这给游戏添加了一些活力，这种活力是一堆以直线飞入屏幕的敌方飞船所不能带来的。正弦和余弦函数常用于游戏的动画开发，因为它们提供了一种平滑的运动过渡机制。请参阅表 2-1 中的说明，来了解 A～H 之间的每个参数给敌方飞船的移动所带来的影响。

表 2-1　参数说明

参　数	说　明
A	水平速度常量
B	水平正弦速度的强度
C	水平正弦速度的周期
D	水平正弦速度的时移
E	垂直速度常量
F	垂直正弦速度的强度
G	垂直正弦速度的周期

注意：就该例而言，使用二次方程(a + bx + cx*x)创造出来的抛物线也是可用的，只可惜抛物线不提供周期性的行为，所以在这种情况下不是特别有用。

各种不同速度值的组合产生了各种不同的行为，若把 B 和 F 设置为零，那么敌方飞船的飞行线路就是直的，因为两个方向上的正弦部分都为零。若把 F 和 A 设置为零，则敌方飞船在 y 方向上以不变速度飞行，但在 x 方向上则是平滑地来回移动。

在 2.7.1 节中，你将通过设置各种参数变化来创建各种不同的敌方飞船。

在游戏产品中，若不打算由自己来操心数学计算的处理，那么可以考虑使用诸如 TweenJS(www.createjs.com/TweenJS)之类的补间引擎(tweening engine)，这类引擎能够以多种有趣的方式处理对象从一个位置到另一个位置的平滑移动。

2.4.2　构造 Enemy 对象

可以通过蓝本来创建敌方飞船，蓝本设置了所使用的精灵图像、初始的位置和移动常量 A～H 的值。构造函数还支持传入重写对象来覆盖默认的蓝本设置。

与 PlayerMissile 非常类似，Enemy 对象把方法添加到原型中，以此来提高对象的创建速度，以及减少内存占用量。

最初的 Enemy 版本看起来非常类似之前构造的两个精灵类(PlayerShip 和 PlayerMissile)，它有一个如代码清单 2-2 所示的初始化一些状态的构造函数、一个更新位置并检查精灵是

否出界的 step 方法，以及一个渲染精灵的 draw 函数。因为需要复制蓝本和所有重写参数，而且需要设置速度公式的参数，所以构造函数比之前的那些更复杂一些。

JavaScript 没有内置的方法可用来轻松复制另一个对象的特性，所以需要循环遍历特性来完成这一工作。为了免去让蓝本设置 A～H 之间每个参数的必要，这些参数中的每一个都被初始化成零。

代码清单 2-2：Enemy 的构造函数

```
var Enemy = function(blueprint,override) {
  var baseParameters =  { A: 0, B: 0, C: 0, D: 0,
                          E: 0, F: 0, G: 0, H: 0 }
  // Set all the base parameters to 0
  for(var prop in baseParameters) {
    this[prop] = baseParameters[prop];
  }
  // Copy of all the attributes from the blueprint
  for(prop in blueprint) {
    this[prop] = blueprint[prop];
  }
  // Copy of all the attributes from the override, if present
  if(override) {
    for(prop in override) {
      this[prop] = override[prop];
    }
  }
  this.w = SpriteSheet.map[this.sprite].w;
  this.h = SpriteSheet.map[this.sprite].h;
  this.t = 0;
}
```

该构造函数首先把三组对象复制到 this 对象中，它们分别是基础参数(base parameter)、蓝本(blueprint))和重写内容(override)。因为根据蓝本的不同，敌方飞船可能会使用不同的精灵，所以接下来的 width 和 height 基于对象的 sprite 属性进行设置。最后，t 参数被初始化成 0，用于记录该精灵已经存活了多长时间。

若重复编写同样的代码让你感到厌烦，别担心！本章后面的 2.5 节会对之加以清理。

2.4.3　移动和绘制 Enemy 对象

敌方飞船的 step 函数(见代码清单 2-3)应该基于前面提到的公式来更新速度，this.t 属性需要递增 dt 值来记录精灵的存活时间。接下来，可将本章前面提到的公式直接插入 step 函数来计算 x 和 y 方向的速度，通过 x 和 y 方向的速度来更新 x 和 y 的位置。最后，精灵需要检查它自己是否已出了左边界或右边界，若是，则敌方飞船从页面上删除自身。

代码清单 2-3：Enemy 对象的 step 和 draw 方法

```
Enemy.prototype.step = function(dt) {
```

```
  this.t += dt;
  this.vx = this.A + this.B * Math.sin(this.C * this.t + this.D);
  this.vy = this.E + this.F * Math.sin(this.G * this.t + this.H);
  this.x += this.vx * dt;
  this.y += this.vy * dt;
  if(this.y > Game.height ||
     this.x < -this.w ||
     this.x > Game.width) {
       this.board.remove(this);
  }
}

Enemy.prototype.draw = function(ctx) {
  SpriteSheet.draw(ctx,this.sprite,this.x,this.y);
}
```

draw 函数几乎是 PlayerMissile 对象的 draw 函数的翻版，唯一的不同之处是它必须在一个名为 sprite 的属性中找出要绘制的精灵。

2.4.4　将敌方飞船添加到面板上

现在，你要把一些初始的敌方飞船精灵添加到 game.js 文件的顶部，同时添加一个简单的敌方飞船蓝本，这是一种能够从页面顶部向下飞的敌方飞船：

```
var sprites = {
 ship: { sx: 0, sy: 0, w: 37, h: 42, frames: 1 },
 missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 },
 enemy_purple: { sx: 37, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_bee: { sx: 79, sy: 0, w: 37, h: 43, frames: 1 },
 enemy_ship: { sx: 116, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_circle: { sx: 158, sy: 0, w: 32, h: 33, frames: 1 }
};

var enemies = {
  basic: { x: 100, y: -50, sprite: 'enemy_purple', B: 100, C: 2 , E: 100 }
};
```

接下来修改 playGame，把两艘敌方飞船添加到页面顶部：

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new Enemy(enemies.basic));
  board.add(new Enemy(enemies.basic, { x: 200 }));
  board.add(new PlayerShip());
  Game.setBoard(3,board);
}
```

这两行代码使用 enemies 对象作为敌方飞船的蓝本，把向页面添加敌方飞船的工作简化成使用该蓝本来调用 new Enemy()方法。为让第二艘敌方飞船出现在第一艘敌方飞船的右侧，代码给构造函数传入一个把 x 设置成 200 的重写对象。

重新载入文件，在游戏启动时，你应会看到两个坏家伙左右摆动着飞到了屏幕的下方，然后消失在屏幕底部。此外，还可以访问一下 http://mh5gd.com/ch2/enemies，看看这些代码带来的效果。这些敌方飞船还未进行任何碰撞检测，所以它们还没有与玩家交手。

基本类型的敌方飞船只定义了三个飞船移动参数：B(水平正弦移动)、C(水平正弦周期)和 E(垂直不变移动)。可使用这些参数来影响移动，例如，递增 C 值就会提高敌方飞船来回移动的频率。

2.5　重构精灵类

到目前为止，游戏已有三个不同的精灵类，它们都用到了许多同样的样板代码，这意味着是时候应用三次法则(Rule of Three)了。

维基百科这样介绍这一法则：

三次法则是代码重构的一条经验法则，涉及当代码片段出现重复时，如何决定是否用一个新的子程序替代它的标准。三次法则的要求是，允许按需直接复制粘贴代码一次，但如果相同的代码片段重复出现三次以上，将其提取出来做成一个子程序就势在必行。马丁·福勒在《重构》一书中介绍了三次法则，并认为这一法则由 Don Roberts 提出。

https://zh.wikipedia.org/wiki/三次法则_(程序设计)

尽管 Alien Invasion 是一个一次性的游戏引擎，它的目标也不是最终成为一个通用的引擎，但是，如果有必要清理任何有蔓延趋势的重复，并把游戏变得更便于修改和扩展，花点儿时间来重构代码还是有利无害的，

没有人一开始就能写出完美的代码，特别是在原型化和尝试新功能阶段。不过，在代码可用之后，开发期间疏于对代码进行重构和清理会导致一些“技术债务(technical debt)”，项目的技术债务越多，修改功能和添加新功能就会变成一件越痛苦的事情。重构可以通过删除无用的代码、减少代码冗余和清理抽象来清除技术债务，所做的这一切未必能让你的游戏更出色，但能让你的游戏开发者生涯变得更美好。

在 Alien Invasion 中，造成三个精灵类代码冗余的元凶是样板式的设置代码和 draw 方法，该方法在三个类中都是一样的。现在是时候把它们抽取出来，放到一个名为 Sprite 的基对象中了，该对象可根据一组设置参数及一个要到用的精灵来处理初始化。在 Enemy 构造函数的内部，有三个循环用来将一个对象的内容复制到另一个中，这也是一个适合重构的地方。

若你没有使用 JavaScript 写过大量的原型继承，那么它的语法看起来可能会有点奇怪。因为 JavaScript 没有类的概念，所以不要定义类来表示被继承的属性，你要创建一个原型对象，当某个参数并未在实际对象中定义时，JavaScript 就会到这个原型中查找。

2.5.1　创建一个通用的 Sprite 类

在本节中，你将创建一个被其他所有精灵继承的 Sprite 对象。打开 engine.js 文件，添加如代码清单 2-4 所示的代码。

代码清单 2-4：通用的 Sprite 对象

```
var Sprite = function() { }

Sprite.prototype.setup = function(sprite,props) {
  this.sprite = sprite;
  this.merge(props);
  this.frame = this.frame || 0;
  this.w =  SpriteSheet.map[sprite].w;
  this.h =  SpriteSheet.map[sprite].h;
}

Sprite.prototype.merge = function(props) {
  if(props) {
    for(var prop in props) {
      this[prop] = props[prop];
    }
  }
}
Sprite.prototype.draw = function(ctx) {
  SpriteSheet.draw(ctx,this.sprite,this.x,this.y,this.frame);
}
```

这部分代码被放到 engine.js 文件中是因为，相比于游戏特定的代码，它是一段通用的引擎代码。构造函数内容为空是因为每个精灵都有自己的构造函数，对于每个子精灵对象的定义而言，Sprite 对象只被创建一次。JavaScript 的构造函数与其他诸如 C++之类 OO 语言的构造函数的工作方式不同，为解决这个问题，你需要在子对象中显式地调用一个单独的 setup 方法。

这个 setup 方法接收的参数包括 SpriteSheet 中的精灵的名称和一个属性对象，精灵被保存在对象的内部，接着属性被复制到 Sprite 中，宽度和高度也在此处设置。

因为属性复制是如此常见的一项需求，所以 Sprite 还定义了 merge 方法来专门完成这项工作，该方法被用在 setup 方法中。

最后是 draw 方法，到目前为止，该方法在每个精灵中几乎都是一样的，所以可在这里定义一次，然后在其他每个精灵中就都可以使用了。

2.5.2　重构 PlayShip

有了 Sprite 类，现在可以通过重构 PlayerShip 对象来简化它的设置了。代码清单 2-5 使用粗体标出了新的代码。

代码清单 2-5：重构后的 PlayerShip

```
var PlayerShip = function() {
  this.setup('ship', { vx: 0, frame: 1, reloadTime: 0.25, maxVel: 200 });

  this.reload = this.reloadTime;
  this.x = Game.width/2 - this.w / 2;
  this.y = Game.height - 10 - this.h;

  this.step = function(dt) {
    if(Game.keys['left']) { this.vx = -this.maxVel; }
    else if(Game.keys['right']) { this.vx = this.maxVel; }
    else { this.vx = 0; }
    this.x += this.vx * dt;
    if(this.x < 0) { this.x = 0; }
    else if(this.x > Game.width - this.w) {
      this.x = Game.width - this.w
    }
    this.reload-=dt;
    if(Game.keys['fire'] && this.reload < 0) {
      this.reload = this.reloadTime;
      this.board.add(new PlayerMissile(this.x,this.y+this.h/2));
      this.board.add(new PlayerMissile(this.x+this.w,this.y+this.h/2));
    }
  }
}

PlayerShip.prototype = new Sprite();
```

在构造函数的开头调用 setup 方法，这样就去掉了一些样板代码。对象的一些属性在调用 setup 时进行设置，但另一些则是在其后进行设置，因为它们依赖于其他属性值，如对象的宽度和高度，这些属性直到 setup 被调用后才可用。接下来的另一项重构是删除 draw 方法，因为该方法现在由 Sprite 掌管。

最后，紧跟在 PlayerShip 构造函数的定义后面，加上真正设置 PlayerShip 原型的代码。

2.5.3　重构 PlayerMissile

PlayerMissile 对象已经很简洁了，但重构有助于进一步简化它。重构之后的内容见代码清单 2-6。

代码清单 2-6：重构之后的 PlayerMissile

```
var PlayerMissile = function(x,y) {
  this.setup('missile',{ vy: -700 });
  this.x = x - this.w/2;
  this.y = y - this.h;
};
```

```
PlayerMissile.prototype = new Sprite();

PlayerMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  if(this.y < -this.h) { this.board.remove(this); }
};
```

构造方法仍然需要显式地设置 x 和 y 的位置，因为这些位置依赖于精灵的宽度和高度(直至 setup 被调用之后这些属性才是可用的)。setup 方法没有受到重构的影响，draw 方法现在由 Sprite 掌管，所以可删除该方法。

2.5.4　重构 Enemy

Enemy 对象从重构获得的益处最大，特别是它的构造方法。该方法不再使用几个循环把参数复制到对象中，几个 merge 调用把该方法简化成三行代码，见代码清单 2-7。

代码清单 2-7：重构后的 Enemy 对象(部分代码)

```
var Enemy = function(blueprint,override) {
  this.merge(this.baseParameters);
  this.setup(blueprint.sprite,blueprint);
  this.merge(override);
}
Enemy.prototype = new Sprite();
Enemy.prototype.baseParameters = { A: 0, B: 0, C: 0, D: 0,
                                   E: 0, F: 0, G: 0, H: 0,
                                   t: 0 };
```

其中的 step 方法未受影响(所以未放在代码清单 2-7 中显示)，draw 方法已被删除。可以注意到，merge 被显式调用来合并 baseParameters 和 override 参数集，预定义的 baseParameters 对象也从构造函数中抽取出来放到原型中。这虽然不是什么巨大优化，但却避免了在每次创建一个新的 Enemy 时，仅是为了被复制到该对象中，就需要重新创建一个静态的 baseParameters 对象。因为这里并未打算让 baseParameters 成为可修改的，所以只要存在该对象的一个副本就可以了。

2.6　处理碰撞

一个完整的 Alien Invasion 慢慢浮现出来了，现在它有了玩家、导弹和在屏幕上四处飞动的敌方飞船。只可惜，各部分之间彼此还没有交火，像保护地球免受毁灭的射击类游戏应该做的那样：炸掉对方。

不过好消息是，处理碰撞这种苦活已经被完成了大半，GameBoard 对象已经知道如何取得两个对象并查明它们是否有重叠，而且也已知道如何确定某个对象是否与其他所有某种特定类型的对象发生了碰撞。现在需要做的就是把正确的调用加入到这些碰撞函数中来。

就碰撞而言，Alien Invasion 可以使用两种机制。第一种机制主动在每个对象的 step 函

数中进行检查，查看该对象与其交互的其他所有对象的碰撞情况。第二种机制提供通用的碰撞阶段，在这个阶段中，对象在击中彼此时触发碰撞事件。前一种机制的实现较为简单，但后一种提供了更好的综合性能，且能够被更好地优化。Alien Invasion 决定走简单路线，不过第 18 章构建的平台动作游戏则使用了后一种更复杂的机制。

2.6.1　添加对象类型

为了确保对象只与那些让彼此的碰撞有意义的对象碰撞，需要为对象指定类型，这一点在本章开头已做讨论，但尚未在游戏中实现。实现的第一步是确定游戏拥有的各种对象类型，然后添加一些常量，这样就不必在代码中使用那些魔法数字了。

将代码清单 2-8 中的代码添加到 game.js 顶部，这段代码定义了 5 种不同的对象类型。

代码清单 2-8：对象的类型

```
var OBJECT_PLAYER = 1,
    OBJECT_PLAYER_PROJECTILE = 2,
    OBJECT_ENEMY = 4,
    OBJECT_ENEMY_PROJECTILE = 8,
    OBJECT_POWERUP = 16;
```

注意：代码清单 2-8 给出的这些类型值中的每一个都是 2 的幂，这是一种效率优化，这样就能够使用早前讨论过的按位逻辑运算。

接下来，在 game.js 中的三行 Sprite 原型赋值代码后面的适当位置，分别添加一行代码，用于设置每个 Sprite 的类型：

```
PlayerShip.prototype = new Sprite();
PlayerShip.prototype.type = OBJECT_PLAYER;
...
PlayerMissile.prototype = new Sprite();
PlayerMissile.prototype.type = OBJECT_PLAYER_PROJECTILE;
...
Enemy.prototype = new Sprite();
Enemy.prototype.type = OBJECT_ENEMY;
```

现在每个对象都有了可用于碰撞检测的类型。

2.6.2　让导弹和敌方飞船碰撞

为避免多余的工作，对象不会与每一种它们可能会碰上的对象都进行碰撞检测，对象只针对那些它们真正“希望”击中的对象进行检查。这意味着 PlayerMissile 对象会检查它们与 Enemy 对象的碰撞情况，但 Enemy 对象不会检查它们与 PlayerMissile 对象的碰撞情况，这样做能让计算量减少一些。

现在对象可能会被击中，它们需要使用一个方法来处理被击中时应该发生的事情。首先将一个方法添加到 Sprite 中，以便在任意对象被击中时将对象删除，该方法可在将来某个时候被各种继承自 Sprite 的对象重写。

把以下函数添加到 engine.js 的末尾处，放在剩余的 Sprite 对象定义部分的后面：

```
Sprite.prototype.hit = function(damage) {
  this.board.remove(this);
}
```

hit 方法的初始版本只是把对象从面板中删除，完全不考虑对象被毁坏的程度。

把一个破坏(damage)值添加到 PlayerMissile 构造函数中：

```
var PlayerMissile = function(x,y) {
  this.setup('missile',{ vy: -700, damage: 10 });
  this.x = x - this.w/2;
  this.y = y - this.h;
};
```

接下来打开 game.js，修改 PlayerMissile 的 step 方法，加入碰撞检测：

```
PlayerMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  var collision = this.board.collide(this,OBJECT_ENEMY);
  if(collision) {
    collision.hit(this.damage);
    this.board.remove(this);
  } else if(this.y < -this.h) {
      this.board.remove(this);
  }
};
```

导弹查看自己是否与任何 OBJECT_ENEMY 类型的对象发生了碰撞，接着调用其碰上的任何一个对象的 hit 方法。然后，它从面板中删除自身，因为它的任务已经完成。

启动游戏，现在你应能够射下两艘在屏幕中飞下来的敌方飞船。

2.6.3　让敌方飞船和玩家碰撞

为了战斗的公平起见，敌方飞船同样需要具备在接触到玩家时能把玩家击毁的能力。

把基本上相同的一段代码添加到 Enemy 的 step 方法中，允许 Enemy 击毁玩家。把 step 方法修改成如下内容：

```
Enemy.prototype.step = function(dt) {
  this.t += dt;
  this.vx = this.A + this.B * Math.sin(this.C * this.t + this.D);
  this.vy = this.E + this.F * Math.sin(this.G * this.t + this.H);
  this.x += this.vx * dt;
  this.y += this.vy * dt;
```

```
    var collision = this.board.collide(this,OBJECT_PLAYER);
    if(collision) {
      collision.hit(this.damage);
      this.board.remove(this);
    }

  if(this.y > Game.height ||
      this.x < -this.w ||
      this.x > Game.width) {
        this.board.remove(this);
    }
  }
```

除了使用 OBJECT_PLAYER 对象类型来调用 collide 之外，这段代码与添加到 PlayerMissile 对象中的代码是一样的。

完成这些修改后，启动游戏，让玩家被一艘敌方飞船击毁。

2.6.4　制造爆炸

到目前为止，碰撞已经收到了正确的效果；不过要是能制造一种更夸张的助兴效果，那就更好了。sprites.png 文件拥有很不错的爆炸动画，可达到此目的，动画中的爆炸图像是利用 http://www.positech.co.uk/上的爆炸生成器生成的。

把爆炸的精灵定义添加到 game.js 文件顶部：

```
var sprites = {
 ship: { sx: 0, sy: 0, w: 37, h: 42, frames: 1 },
 missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 },
 enemy_purple: { sx: 37, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_bee: { sx: 79, sy: 0, w: 37, h: 43, frames: 1 },
 enemy_ship: { sx: 116, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_circle: { sx: 158, sy: 0, w: 32, h: 33, frames: 1 },
 explosion: { sx: 0, sy: 64, w: 64, h: 64, frames: 12 }
};
```

现在将一些 health(健康)数据添加到基本类敌方飞船的蓝本中：

```
var enemies = {
  basic: { x: 100, y: -50, sprite: 'enemy_purple',
           B: 100, C: 4, E: 100, health: 20 }
};
```

接下来，需要重写 Enemy 对象继承自 Sprite 的默认 hit 方法，该方法需要降低 Enemy 的健康程度，所以要检查 Enemy 是否已经用完了健康值。若是，则以 Enemy 的中心为添加位置，在 GameBoard 中添加爆炸，如代码清单 2-9 所示。

代码清单 2-9：Enemy 的 hit 方法

```
Enemy.prototype.hit = function(damage) {
  this.health -= damage;
```

```
    if(this.health <=0) {
      if(this.board.remove(this)) {
        this.board.add(new Explosion(this.x + this.w/2,
                                     this.y + this.h/2));
      }
    }
  }
```

最后构建 Explosion 类，该类是一个简单的精灵，在被添加到页面上时，它只需快速播放一遍自己的各帧，然后从面板中删除自身就可以了。类的内容详见代码清单 2-10。

代码清单 2-10：Explosion 对象

```
var Explosion = function(centerX,centerY) {
  this.setup('explosion', { frame: 0 });
  this.x = centerX - this.w/2;
  this.y = centerY - this.h/2;
  this.subFrame = 0;
};

Explosion.prototype = new Sprite();

Explosion.prototype.step = function(dt) {
  this.frame = Math.floor(this.subFrame++ / 3);
  if(this.subFrame >= 36) {
    this.board.remove(this);
  }
};
```

Explosion 的构造方法接收传入的位置 centerX 和 centerY 作为参数，通过把精灵向左和向上分别移动其自身宽度和高度一半的距离来调整 x 和 y 的位置。step 方法不必关心每一爆炸帧的移动，它只须通过更新 subFrame 属性来遍历爆炸动画的每帧内容。爆炸动画的每帧都被当成三个游戏帧播放，目的是让爆炸过程持续得更久一些。当爆炸的所有 36 个子帧(subFrame)都已播放完毕(实际上是 12 帧)时，Explosion 就从面板中删除自身。

重新加载游戏，尝试击毁从屏幕中飞下来的敌方飞船。现在击毁一艘敌方飞船需要两枚导弹，不过敌方飞船应会在一场激烈的爆炸中被炸得粉身碎骨。

2.7　描述关卡

现在，Alien Invasion 已经具备了可玩游戏所需的全部机制，唯一缺少的就是一个把关卡数据和把敌方飞船添加到屏幕上的机制整合起来的部件。

2.7.1　设置敌方飞船

可为敌方飞船制造出无数种移动变化，但就此游戏而言，一开始会使用各种敌方飞船

精灵来设置 5 种不同类型的敌方飞船行为。可使用已定义好的类型，若愿意的话，也可以添加更多类型。可制造出其他许多变化，不过这 5 种类型就已经是很好的开始了。用代码清单 2-11 中的内容替换 game.js 顶部的 enemies 定义。

代码清单 2-11：定义敌方飞船

```
var enemies = {
  straight: { x: 0,   y: -50, sprite: 'enemy_ship', health: 10,
              E: 100 },
  ltr:      { x: 0,   y: -100, sprite: 'enemy_purple', health: 10,
              B: 200, C: 1, E: 200  },
  circle:   { x: 400,   y: -50, sprite: 'enemy_circle', health: 10,
              A: 0,  B: -200, C: 1, E: 20, F: 200, G: 1, H: Math.PI/2 },
  wiggle:   { x: 100, y: -50, sprite: 'enemy_bee', health: 20,
              B: 100, C: 4, E: 100 },
  step:     { x: 0,   y: -50, sprite: 'enemy_circle', health: 10,
              B: 300, C: 1.5, E: 60 }
};
```

只需在移动参数上有所变化，敌方飞船就会拥有非常不一样的移动风格。直线类(straight)敌方飞船只用到垂直速度参数 E，所以它以恒定的速率向下移动。

ltr 类(left-to-right，从左到右的英文缩写)敌方飞船有恒定的垂直速度，但另一方面，正弦水平速度(参数 B 和 C)赋予了它一种从左至右平滑扫动的移动风格。

盘旋类(circle)敌方飞船在两个方向上的基本运动都是正弦运动，但在 Y 方向使用参数 H 添加了一段时移，赋予飞船一种环形移动风格。

摆动类(wiggle)敌方飞船和阶梯类(step)敌方飞船使用了同一组参数，仅参数值有所不同。摆动类敌方飞船有着更小的 B 值和更大的 C、E 值，所以它只能像蛇一样蜿蜒行至屏幕下方，而阶梯类敌方飞船具有更大的 B 值和更小的 C、E 值，这使得它能够在整个屏幕上慢慢来回滑动至页面底部。

2.7.2　设置关卡数据

既然已知道 Alien Invasion 的各个游戏关卡就是填充一串串相同类型的敌方飞船，那么下一步就是找出一种好的机制以一种紧凑方式编码关卡数据。在找出这样一种机制后，就可以回过头来弄清楚，要在页面上不停生成这些敌方飞船，关卡对象需要完成哪些工作。先从希望如何使用一段代码开始考虑，再回头考虑代码的实现，这是一种很好的做法，这样最终能得到便于使用的代码。实现代码的工作量可能因此有所增加，但从长远看你会更乐于如此。

启动这一过程应该要做的第一件事是在一个数组中编码每艘敌方飞船的起始位置和每种敌方飞船类型，因为游戏的一关可能会有好几百艘敌方飞船，所以这很快就会变成一项费时费力的工作。一种更好的选择是把每串敌方飞船都编码成单个具有开始时间、结束时间和每艘飞船延迟时间的条目。这样，每串敌方飞船就以一种简洁的方式编码到关卡数

据中，可以看一下定义，很好地理解一下其中发生的事情。

添加第一关的关卡数据，将代码清单 2-12 中的代码插入到 game.js 的顶部。

代码清单 2-12：关卡数据

```
var level1 = [
 // Start,    End, Gap,  Type,   Override
  [ 0,       4000, 500, 'step' ],
  [ 6000,   13000, 800, 'ltr' ],
  [ 12000,  16000, 400, 'circle' ],
  [ 18200,  20000, 500, 'straight', { x: 150 } ],
  [ 18200,  20000, 500, 'straight', { x: 100 } ],
  [ 18400,  20000, 500, 'straight', { x: 200 } ],
  [ 22000,  25000, 400, 'wiggle', { x: 300 }],
  [ 22000,  25000, 400, 'wiggle', { x: 200 }]
];
```

其中每个条目都给出了以毫秒为单位的开始时间、结束时间以及每两艘敌方飞船之间出现的时间间隔，后面接着是敌方飞船的类型和任何重写参数。

2.7.3　加载和结束一关游戏

定义 level 类将如何使用关卡数据仅等于成功了一半，另一半是要确定 PlayGame 方法如何使用 Level 对象来启动游戏。最简单的解决方案就是创建另一个类似精灵的对象，然后把该对象添加到游戏面板中，让它按照正确的时间间隔来源源不断生成敌方飞船。在 Level 生成所有敌方飞船之后，它就可以通过回调来表明过关成功。

同样是逆向的工作方式，在真正实现之前，先编写使用 Level 对象的代码，使用代码清单 2-13 中所示的 playGame 方法来替换现有的那个，同时加入 winGame 和 loseGame 这两个新方法。

代码清单 2-13：修改游戏的初始化方法

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  board.add(new Level(level1,winGame));
  Game.setBoard(3,board);
}
var winGame = function() {
  Game.setBoard(3,new TitleScreen("You win!",
                                  "Press fire to play again",
                                  playGame));
}
var loseGame = function() {
  Game.setBoard(3,new TitleScreen("You lose!",
                                  "Press fire to play again",
                                  playGame));
}
```

添加关卡成了小事一桩，只需将一个新的 Level 精灵添加到面板中，并传入关卡数据 level1 和成功回调 winGame 即可。

winGame 方法仅是重用 TitleScreen 对象来显示一条成功消息和一条告知玩家他们可以重玩该游戏的消息。

loseGame 方法的作用与 winGame 方法相同，只是显示的消息少了祝贺之意。到目前为止，loseGame 尚未在某处被调用，不过可通过在 PlayShip 对象中添加定制的 hit 方法来对这一问题加以改正。将以下定义添加到 game.js 中，放在 PlayerShip 方法余下内容的后面(务必把它添加在设置原型的语句的后面)：

```
PlayerShip.prototype.hit = function(damage) {
  if(this.board.remove(this)) {
    loseGame();
  }
}
```

在消亡时，PlayShip 并未发生爆炸，这仅是为了简单起见。不过，可加入爆炸场面，并添加回调至爆炸步骤结尾处，在 PlayShip 已经完全炸毁之后才显示 loseGame 界面。

2.7.4　实现 Level 对象

现在剩下的工作就是实现 Level 对象了，该对象的职责已由关卡数据以及 playGame 和 winGame 方法的设置做法所定义。Level 对象只有两个方法：一个是构造函数，该函数复制关卡数据供对象自身使用(和修改)；另一个是 step 方法，该方法遍历关卡数据并在需要时把敌方飞船添加到面板中。

将代码清单 2-14 中显示的构造函数添加到 engine.js 的末尾处。

代码清单 2-14：Level 对象的构造函数

```
var Level = function(levelData,callback) {
  this.levelData = [];
  for(var i =0; i<levelData.length; i++) {
    this.levelData.push(Object.create(levelData[i]));
  }
  this.t = 0;
  this.callback = callback;
}
```

该构造函数的主要职责是深度复制作为参数传进来的关卡数据，复制数据是必需的，因为随着一关游戏的向前推进，方法会修改关卡数据，又因为在 JavaScript 中，对象是通过引用传递的，若玩家打算重玩一次某关游戏，那么这种修改就会妨碍到关卡的重用。

复制比表面上看起来要复杂，因为 JavaScript 没有内置的用于深度复制数组(Array)内部对象列表的机制。解决这一问题的做法是遍历关卡数据数组中的每个条目，使用现有的数据作为原型，调用内置的 Object.create 方法创建一个新对象，然后把这一新对象压入到一个新数组中。

接下来是 Level 对象的主要部分：step 方法。尽管 Level 不是一个标准的精灵(Sprite)，但它打算假扮精灵，并通过响应 step 和 draw 方法来让自己像精灵一样行事。如代码清单 2-15 所示，step 方法负责记录当前时间，并按时间顺序让敌方飞船降临页面之上。

代码清单 2-15：Level 对象的 step 方法

```
Level.prototype.step = function(dt) {
  var idx = 0, remove = [], curShip = null;

 // Update the current time offset
  this.t += dt * 1000;

  //  Example levelData
  //   Start, End,  Gap, Type,   Override
  // [[ 0,     4000, 500, 'step', { x: 100 } ]
  while((curShip = this.levelData[idx]) &&
        (curShip[0] < this.t + 2000)) {
    // Check if past the end time
    if(this.t > curShip[1]) {
      // If so, remove the entry
      remove.push(curShip);
    } else if(curShip[0] < this.t) {
      // Get the enemy definition blueprint
      var enemy = enemies[curShip[3]],
          override = curShip[4];

      // Add a new enemy with the blueprint and override
      this.board.add(new Enemy(enemy,override));

      // Increment the start time by the gap
      curShip[0] += curShip[2];
    }
    idx++;
  }
  // Remove any objects from the levelData that have passed
  for(var i=0,len=remove.length;i<len;i++) {
    var idx = this.levelData.indexOf(remove[i]);
    if(idx != -1) this.levelData.splice(idx,1);
  }

  // If there are no more enemies on the board or in
  // levelData, this level is done
  if(this.levelData.length == 0 && this.board.cnt[OBJECT_ENEMY] == 0) {
    if(this.callback) this.callback();
  }
}

// Dummy method, doesn't draw anything
Level.prototype.draw = function(ctx) { }
```

这是一个复杂方法，该方法可以分成三个主要部分：

- 第一部分使用 while 语句来遍历 levelData 数组元素的开头部分内容，直至遇到任何活动的飞船(这样做就免去了遍历 levelData 数组中每个元素的必要性)。代码检查关卡数据的每行内容，看看关卡存在的时长是否已超过了该行内容中的终值(行内容数组的第二个元素)。若是，则把该元素添加至一个要从 levelData 数组中删除的元素列表中；若不是，则取出该敌方飞船的蓝本和重写参数，将一艘新的敌方飞船添加到面板中。然后，给开始值(行内容数组的第一个元素)增加生成两艘飞船的间隔时间。这种修改开始时间的做法能够让 step 方法在无需任何附加逻辑的情况下把一连串的敌方飞船添加到页面上。
- step 方法的第二部分内容看起来应与 GameBoard 对象类似，这部分代码所要做的就是找出 levelData 中所有已被添加到删除列表中的条目，然后把它们从数组中剔除，这很像是 GameBoard 中 finalizeRemoved 方法所做的事情。
- 最后一部分内容包含了两个判断条件，其中一个检查 levelData 中是否还有将要出现的敌方飞船，另一个检查面板中的敌方飞船数目是否为零。若这两个条件同时成立，那么游戏的这一关就被认为已经结束，若已设置回调的话就调用回调。这种做法能够让关卡知道自己何时已经结束。

最后，Level 对象还需要 draw 方法，这样它就能很好地与 GameBoad 一起配合使用，不过该方法仅是存根(stub)而已，实际上什么事情也不做。

启动已经加入了所有 Level 部件的游戏，现在应能看到游戏的盛大场面和各种敌方飞船的光辉形象了。

2.8　小结

祝贺你！你拿到的不过是游戏的存根——孤零零的一艘在空荡荡的太空中飞来飞去的飞船——却把它变成了一个可玩的游戏，一个有着一波波来袭的敌人，有着胜利和失败画面的游戏。

不过，你可能已经注意到了一个小问题，那就是，到目前为止——它还不是移动的。下一章会纠正这一问题，到时你将添加一些触摸控件和对尺寸调整的支持。最后的一些收尾工作，比如得分等，能够把 Alien Invasion 变成一个精美、耐玩，同时也可在桌面上运行的移动游戏。

第 3 章

试飞结束，向移动进发

本章提要

- 探讨场景管理
- 添加炮弹和敌方飞船
- 使用碰撞检测
- 制造爆炸

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 3 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

3.1 引言

HTML5 大肆宣扬的一个优势是它对移动设备的支持，尽管本书的书名如此，但这却是到目前为止，书中游戏所忽略的一个方面，本章要做的就是纠正这一问题。给 Alien Invasion 添加对移动设备的支持，这意味着需要识别在触摸设备上玩游戏这种情况并做出正确响应，在这个例子中，则意味着添加触摸控件以及根据设备调整游戏界面尺寸。

3.2 添加触摸控件

自 2007 年 iPhone 进入市场以来，移动输入设备的发展方向就已经明朗化：触摸屏是

赢家。要把 Alien Invasion 打造成一款在移动设备和平板设备上可玩的游戏，它就必须是仅使用屏幕作为输入设备也是可玩的。

3.2.1　绘制控件

为了使在移动设备上玩游戏成为可能，一种常见的解决方案是在屏幕上添加一些可视触摸控件，这些控件可由位于页面底部的三个方形按钮组成：左移飞船的左箭头、右移飞船的右箭头，以及发射按钮“A”。

要加入这些控件，游戏需要在其他所有面板之上再添加一个新的游戏面板。因为是在其他所有游戏部件之后被渲染，所以控件始终处在页面的最上层位置。

游戏需要处理不同的屏幕分辨率，因此，游戏不会绘制固定大小的输入方块(根据设备的不同，最终得到的方块可能会过大或过小)，而会基于游戏界面的宽度调整方块的尺寸。根据一些非正式测试可知，把游戏的界面宽度分成 5 个区就足以获得很好的效果了，按钮要大到能够方便地单击，但又不能过多占用屏幕的实际使用面积。

第一步工作是把这些控件添加到页面上，打开 engine.js 文件，把以下对象(如代码清单 3-1 所示)添加到文件末尾处。

代码清单 3-1：Alien Invasion 的 TouchControls 对象

```
var TouchControls = function() {
  var gutterWidth = 10;
  var unitWidth = Game.width/5;
  var blockWidth = unitWidth-gutterWidth;

  this.drawSquare = function(ctx,x,y,txt,on) {
    ctx.globalAlpha = on ? 0.9 : 0.6;
    ctx.fillStyle =  "#CCC";
    ctx.fillRect(x,y,blockWidth,blockWidth);

    ctx.fillStyle = "#FFF";
    ctx.textAlign = "center";
    ctx.globalAlpha = 1.0;
    ctx.font = "bold " + (3*unitWidth/4) + "px arial";

    ctx.fillText(txt,
               x+blockWidth/2,
               y+3*blockWidth/4+5);
  };


  this.draw = function(ctx) {
    ctx.save();
    var yLoc = Game.height - unitWidth;

    this.drawSquare(ctx,gutterWidth,yLoc,
                 "\u25C0", Game.keys['left']);
```

```
    this.drawSquare(ctx,unitWidth + gutterWidth,yLoc,
                    "\u25B6",Game.keys['right']);

    this.drawSquare(ctx,4*unitWidth,yLoc,"A",Game.keys['fire']);
    ctx.restore();
  };

  this.step = function(dt) { };
};
```

该对象基于游戏界面的宽度设置一些值，这些值将用来绘制一些对象。每个方块的宽度被设置成游戏界面的 1/5 宽减去按钮分隔槽的 10 个像素宽。

对于每帧而言，对象的 draw 方法都会被调用，该方法转而调用内部方法 drawSquare，drawSquare 在指定位置绘制一个显示文本的矩形。代码使用 Unicode UTF-8 符号表示左右箭头，而非绘制三角形，字符\u25C0 和\u25B6 分别代表向左和向右的三角形。

注意：在 JavaScript 中，Unicode 字符可通过在字母或符号的 UTF-8 编码前面加上反斜杠 u(\u)前缀来表示。

draw 方法使用 2D 画布上下文的 save 和 restore 方法，以防不透明度和字体的改变对其他任何画布调用造成影响。

实际上，drawSquare 方法完成了大部分的工作，它接收 x 和 y 位置、被绘制在按钮上的文本等参数，确定按钮当前是否被按下，然后绘制实心的矩形和文本来创建按钮。按钮状态用来设置按钮背景的不透明度，方法根据状态的不同，使用 globalAlpha 属性设置不同的不透明度，这样玩家就可以看到自己何时按下了按钮。

要真正显示这一面板，所需要做的就是在初始化时把它添加到 Game 对象中。修改 engine.js 中的 Game.initialize 方法，将 setBoard 调用及两个由 TouchControls 使用的属性添加到 Game 中：

```
// Game Initialization
  this.initialize = function(canvasElementId,sprite_data,callback) {
    ...
    this.setupInput();
    this.setBoard(4,new TouchControls());
    this.loop();
    SpriteSheet.load(sprite_data,callback);
  };

  // Game Initialization
  this.initialize = function(canvasElementId,sprite_data,callback) {
    this.canvas = document.getElementById(canvasElementId);
```

```
    this.width = this.canvas.width;
    this.height= this.canvas.height;

    this.ctx = this.canvas.getContext && this.canvas.getContext('2d');
    if(!this.ctx) { return alert("Please upgrade your browser to play"); }

    this.setupInput();

    this.loop();

    SpriteSheet.load(sprite_data,callback);
};
```

触摸控件尚未能控制玩家飞船，不过若使用键盘，你应可看到按钮会变亮来响应控制，就仿佛它们已被按下似的。

3.2.2　响应触摸事件

若要让这些方块和触摸事件合作无间，游戏需要被设置成能够监听一组新的浏览器事件：touchstart、touchmove 和 touchend。你之前很有可能已经见过一些浏览器事件，比如说人人知道的 click 事件等，但这些新事件只有在非 Windows 触摸设备上才是可用的，它们的特别之处在于不仅包含了事件的详细信息，还包含了当前发生在设备上的其他所有触摸的详细信息。这些附加事件的详细信息被放在事件对象内部的三个数组中，如表 3-1 所示。

表 3-1　触摸事件的属性

事 件 属 性	说　明
event.touches	当前发生在设备上的所有触摸
event.targetTouches	发生在作为触发事件对象的同一个 DOM 上的所有触摸
event.changedTouches	在这一事件中发生了变化的所有触摸

游戏既会用到 targetTouches 数组，也会用到 changedTouches 数组，由此来获得良好效果。

targetTouches 用于左边控制移动的两个按钮，你希望用户能够按下这两个按钮之一来进行向左或向右移动。所以，每发生一个触摸事件，游戏就会查看当前是否有触摸击中了这两个按钮中的任一个。若是，则标记该按钮被按下，即使触发事件的触摸并未击中任何按钮，游戏也会进行此番检查。

就发射炮弹而言，作为一种设计选择，游戏要求玩家每次在想要发射导弹时都要按下发射按钮(一直按住发射键不放的不计在内)。基于此原因，只有用户实际上是在上一步时按住了发射键，游戏才会认为发射键已被按下。

将代码清单 3-2 中的这段代码添加到 TouchControls 类中，放在表示结束的花括号之前。

代码清单 3-2：TouchControls 的触摸跟踪

```
var TouchControls = function() {
  ...

  this.step = function(dt) { };

  this.trackTouch = function(e) {
    var touch, x;
    e.preventDefault();
    Game.keys['left'] = false;
    Game.keys['right'] = false;
    for(var i=0;i<e.targetTouches.length;i++) {
      touch = e.targetTouches[i];
      x = touch.pageX / Game.canvasMultiplier - Game.canvas.offsetLeft;
      if(x < unitWidth) {
        Game.keys['left'] = true;
      }
      if(x > unitWidth && x < 2*unitWidth) {
        Game.keys['right'] = true;
      }
    }
    if(e.type == 'touchstart' || e.type == 'touchend') {
      for(i=0;i<e.changedTouches.length;i++) {
        touch = e.changedTouches[i];
        x = touch.pageX / Game.canvasMultiplier - Game.canvas.offsetLeft;
        if(x > 4 * unitWidth) {
          Game.keys['fire'] = (e.type == 'touchstart');
        }
      }
    }
  };

  Game.canvas.addEventListener('touchstart',this.trackTouch,true);
  Game.canvas.addEventListener('touchmove',this.trackTouch,true);
  Game.canvas.addEventListener('touchend',this.trackTouch,true);
  Game.playerOffset = unitWidth + 20;
};
```

在前面的描述中，控件被称为“按钮”，这是基于它们的绘制方式来说的。但若检查一下击中检测代码，就会发现它们是被当成列来进行检测的，只有击中的 x 位置被用来确定用户是否按下了某个按钮。这种做法用在许多移动应用商店游戏中，且效果很好，这是因为玩家在试图按下按钮时，触摸的位置可能不会太准确，他们可能会把手垂直放在设备上，这样感觉舒服一些。

该段代码将一个名为 trackTouch 的方法加入到 TouchControls 中，该方法可作为任何触发事件的通用处理程序使用。trackTouch 所做的第一件事情是调用 e.preventDefault()，该方法摒除任何可能会与该事件有关联的现有行为，包括滚动、单击、缩放等，这样做防止了

页面在用户与画布元素进行交互时展示任何默认行为(至少在 iOS 上是这样。截止撰写本书之时，在 Android 上依然能够通过多点触控触发滚动和缩放，希望这个问题很快能得以修正)。

接下来，trackTouch 方法把左右键的值都设成 false，这样做是因为，若发现有触摸击中按钮，这两个按键之一就会被设回 true 值。以这种方式来执行检测能够让用户在两个按钮之间来回滑动他们的手指，或在不失去节奏的情况下交换手指。对于任何触摸来说，游戏都会查看它们是否落在画布左边头两个单元中，若是，则把这些触摸和左右移动按钮对应起来。

对于发射导弹来说，方法只检查了 changedTouches，如前所述，这是为了强制玩家反复按下发射按钮，只有这样才能接连不断地发射导弹。游戏检测任何触摸是否落在右边的最后一个单元中，若是，则根据事件是 touchstart 还是 touchend 把发射键的值相应地设成 true 或 false。

最后，名为 playerOffset 的变量的值被设置成 unitWidth + 20，这样做的目的是，若屏幕显示了触摸控件，就把玩家飞船向屏幕上方移动；但若是在桌面浏览器中玩游戏，就把玩家飞船置于屏幕的底部。

为了让这一做法生效，PlayShip 需要使用新的 y 位置进行初始化，修改 game.js 中的 PlayerShip，修改后的内容如下粗体代码所示：

```
var PlayerShip = function() {
  this.setup('ship', { vx: 0, reloadTime: 0.25, maxVel: 200 });
  this.reload = this.reloadTime;
  this.x = Game.width/2 - this.w / 2;
  this.y = Game.height - Game.playerOffset - this.h;
  ...
```

现在，玩家飞船被放在距屏幕底部一定距离的位置，这个距离对设备来说是刚好的，这样就可以防止控制按钮遮盖游戏内容。

3.2.3　在移动设备上测试

要在真实移动设备上测试该游戏，你需要使用 Web 服务器来运行游戏。你可在开发机上安装服务器，也可以将代码部署到 Web 主机上，不过这两种方法都有点超出了本书的讨论范围。

注意：在互联网上，可找到许多质量参差不齐的托管公司。一般来说，若刚起步，那么 DreamHost(http://dreamhost.com)是一个可接受的选择。

从长期来看，你需要在不必部署的情况下测试游戏，这样既能做到对游戏进行修改和快速测试，而又不会被任何的中间步骤拖慢开发进程。若使用 Windows，你最有可能做的

是安装 IIS，这取决于你的 Windows 版本，不过这一过程可能涉及 IIS 的配置，除非已能轻松自如地使用 Windows 配置任务，否则你可能会希望使用接下来的某种可选做法。若使用的是 Mac，可通过 System Preferences 中的 Share 部分访问 Web Sharing，在 Linux 上则可以安装 Apache。

若觉得使用完全配置版的 Web 服务器压力太大，那么可在 http://www.wampserver.com/en/ 上了解一下 WAMP。作为一个项目，WAMP 旨在给你提供一台 Windows 上的零配置 Apache 服务器。而对于其他一些平台来说，OS X 上的 Web Sharing 和 Linux 上的 Apache 原生包通常是一种更好的选择。就一些简单需求而言，还可以试用一下 http://code.google.com/p/mongoose/ 上的 mongoose，这是一台要放在你想要提供的目录中执行的 Web 服务器。

假设你是通过 Wi-Fi 来连接网络的，你的开发机和移动设备都位于同一网络中，那么在安装配置了 Web 服务器并把文件放到正确位置后，你现在应能够通过移动设备访问开发机了(这取决于服务器和配置)。

查找机器在局域网中的 IP 地址，这一地址很可能与机器处在 Web 环境中时所看到的公共 IP 地址是不同的，因为大部分 Wi-Fi 网络都位于某个路由器的后面。要在 Windows 上找出你的 IP 地址，最简单的方法是启动命令提示符(Command Prompt)程序(通常放在附件(Accessories)中)，然后输入 ipconfig。

除了其他一些杂乱的内容之外，你应会看到一串类似 xxx.xxx.xxx.xxx 这样的数字(通常是类似 192.168.0.50 这样的内容)。你可能会看到其他一两个以 1 结尾的 IP 地址，如 192.168.0.1，这是网关地址而非你的计算机地址。

在 Mac 或 Linux 上，启动终端(Terminal)，输入 ifconfig。

在 Linux 上，你可能需要输入 sudo ifconfig。同样，你也会在一堆信息中看到一串类似 IP 地址的数字。

有了 IP 地址和文档根目录(Web 服务器提供文件的目录)下的游戏路径，现在你应该可以通过输入该 IP 地址后面加上路径在移动设备上启动游戏了。另外，你也可以通过 http://mh5gd.com/ch3/touch/这一地址来运行到目前为止的游戏版本。

若在移动设备上运行游戏，你立刻就会注意到一个问题：虽然游戏是可玩的，但默认的游戏界面很小，且画布元素又覆盖了触摸事件，这意味着进行放大操作会很困难(可以通过捏拉画布周边的空白处来进行缩放)。下一节会纠正这一问题。

3.3　最大化游戏界面

移动设备的屏幕实际使用面积对移动游戏来说尤其可贵，你最不想做的事情就是因为没有最大化游戏界面而浪费了一些实际使用面积。

3.3.1　设置视口

第一步，告诉浏览器你不想让用户缩放页面，这可以通过在 HTML 中设置视口标签

<meta>来实现。这个视口标签最开始是 iOS 特有的功能，但后来扩展到 Android。在 HTML 文档的<head>部分添加以下代码：

```
  <meta name="viewport" content="width=device-width, user-scalable=0,
minimum-scale=1.0, maximum-scale=1.0"/>
```

该标签告诉浏览器，把页面的宽度设置成实际设备的像素宽，且不允许用户进行缩放。第 6 章将深入探讨这个标签。

若重新加载页面，你会发现游戏界面被放大了一些，但仍未准确吻合页面的宽度。

3.3.2　调整画布尺寸

要解决界面大小问题，让游戏的界面设置尽可能迎合页面的尺寸，那你还得多采取几个步骤才行。由于各种移动设备的独特之处，实现这一点比表面上看起来的要复杂一些。第 6 章会深入讨论这一问题，以下只给出基本的伪代码：

```
Check if browser has support for touch events
(检查浏览器是否支持触摸事件)

Exit early if screen is larger than a max size or no touch support
(若屏幕大于最大尺寸或没有触摸支持，提前退出)

Check if the user is in landscape mode,
(检查用户是否处于横屏模式下)

    if so, ask them to rotate the browser
    (若是，要求他们旋转浏览器)

Resize container to be larger than the page
to allow removal of address bar
(将容器的尺寸调整为大于页面，以便删除地址栏)

Scroll window slightly to force removal of address bar.
(稍微滚动窗口，借此强制删除地址栏)

Set the container size to match the window size
(将容器的尺寸设置成与窗口大小相吻合)

Check if you're on a larger device (like a tablet)
(检查游戏是否在(平板电脑一类的)较大设备上运行)
    if so, set the view size to be twice
    the pixel size for performance
    (若是，把视图的大小设置成性能像素大小的两倍)
If not,
(否则)
    set canvas to match the size of the window.
    (将画布设置成与窗口大小相吻合)

Finally, set the canvas to absolute position
```

```
in the top left of the window
```
（最后，将画布设置成使用绝对定位，把画布的绝对位置设置成窗口的左上角）

接下来看一下真正的代码，将代码清单 3-3 中的方法添加到 engine.js 文件中的 Game 对象的定义中，放在返回语句之前。

代码清单 3-3：setupMobile

```
this.setupMobile = function() {

  var container = document.getElementById("container"),
      hasTouch =  !!('ontouchstart' in window),
      w = window.innerWidth, h = window.innerHeight;

  if(hasTouch) { mobile = true; }

  if(screen.width >= 1280 || !hasTouch) { return false; }

  if(w > h) {
    alert("Please rotate the device and then click OK");
    w = window.innerWidth; h = window.innerHeight;
  }

  container.style.height = h*2 + "px";
  window.scrollTo(0,1);
  h = window.innerHeight + 2;

  container.style.height = h + "px";
  container.style.width = w + "px";
  container.style.padding = 0;

  if(h >= this.canvas.height * 1.75 ||
     swx >= this.canvas.height * 1.75) {
    this.canvasMultiplier = 2;
    this.canvas.width = w / 2;
    this.canvas.height = h / 2;
    this.canvas.style.width = w + "px";
    this.canvas.style.height = h + "px";
  } else {
    this.canvas.width = w;
    this.canvas.height = h;
  }
  this.canvas.style.position='absolute';
  this.canvas.style.left="0px";
  this.canvas.style.top="0px";
};
```

其中的 innerWidth 和 innerHeight 需要被检查多次，这是因为在该方法的调用过程中，在用户旋转设备后，以及在调用 window.scrollTo 来删除地址栏之后，窗口的大小都会发生

变化。

方法还用到了一种技巧，即画布元素的 CSS 尺寸可独立于它的(使用标签的 width 和 height 特性指定的)像素尺寸进行设置，这使得你能放大元素的可视大小而又不必填充更多像素。这种做法的缺点是，像素实际上会被放大 4 倍，这使得游戏看起来有点像素化。不过在诸如 Alien Invasion 一类的复古射击类游戏中，这倒不是什么大问题，只要有所留意就行了。

接下来，对 setupMobile 的调用必须在 Game.initialize 中进行。此外，只要设备支持触摸控件，游戏就应该把它们添加到页面上。修改 Game.initialize 方法，改后的内容如下：

```
// Game Initialization
this.initialize = function(canvasElementId,sprite_data,callback) {
  this.canvas = document.getElementById(canvasElementId);

  this.playerOffset = 10;
  this.canvasMultiplier= 1;
  this.setupMobile();

  this.width = this.canvas.width;
  this.height= this.canvas.height;

  this.ctx = this.canvas.getContext &&
           this.canvas.getContext('2d');

  if(!this.ctx) {
    return alert("Please upgrade your browser to play");
  }
  this.setupInput();
  if(this.mobile) {
    this.setBoard(4,new TouchControls());
  }

  this.loop();

SpriteSheet.load(sprite_data,callback);
  };
```

有了对 this.mobile 的检查，游戏仅在设备支持触摸的情况才下加入可视触摸控件并绑定触摸事件。

3.3.3　添加到 iOS 主屏幕

如何达到 HTML5 游戏的至高境界：全屏运行？你需要用到最后一组 meta 标签，这些代码只可用在 iOS 设备、iPad、iPhone 或 iPod Touch 上。

将以下两个<meta>标签添加到<head>中，放在视口声明的下面：

```
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
```

现在重新加载游戏，然后单击按钮把它添加到主屏幕中。除了在页面的顶部会出现一条细细的状态栏之外，你现在可以全屏运行游戏了(可参阅第 6 章了解这些标签的完整说明)。此外，你还可通过 http://mh5gd.com/ch3/resize 运行该游戏的最新版本。

3.4　添加得分

显然，这个游戏还缺少了一样东西：用户可以借此来向他们的朋友吹嘘的记分系统。这是一个可以快速纠正的问题，只需在游戏中添加一个新的游戏面板即可。

将代码清单 3-4 中的内容添加到 engine.js 的末尾处。

代码清单 3-4：GamePoints

```
var GamePoints = function() {
  Game.points = 0;
  var pointsLength = 8;
  this.draw = function(ctx) {
    ctx.save();

    ctx.font = "bold 18px arial";
    ctx.fillStyle= "#FFFFFF";

    var txt = "" + Game.points;

    var i = pointsLength - txt.length, zeros = "";
    while(i-- > 0) { zeros += "0"; }

    ctx.fillText(zeros + txt,10,20);

    ctx.restore();
  }
  this.step = function(dt) { }
}
```

该对象的存在目标只有一个：在游戏的左上角处绘制得分。游戏的当前得分被直接存放在 Game 对象的名为 points 的属性中。每当创建一个新的 GamePoints 对象时，游戏就假设这是一个新开始的游戏，然后把得分重置成 0。

就每一帧而言，游戏都会抓取当前得分，然后用一些前导零来填充得分，这样得分的位数始终是 pointsLength。然后，游戏调用 fillText 将得分绘制到屏幕上。

为将得分显示在页面上，游戏需要创建一个 GamePoints 对象，打开 game.js，将初始化程序添加到第 5 块面板上：

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  board.add(new Level(level1,winGame));
```

```
  Game.setBoard(3,board);
  Game.setBoard(5,new GamePoints(0));
};
```

之所以选用第 5 块面板，是因为第 4 块面板刚在上一节中被 TouchControls 使用了。

现在，若重新加载游戏，你会看到得分出现在页面的左上角，只可惜一直停留在零分处。因为玩家每次击毁一艘敌方飞船时都应该获得一些分数，所以最简单的做法是往 Enemy.hit 方法中加入一些逻辑。

在 game.js 中修改该方法，修改后的内容如下：

```
Enemy.prototype.hit = function(damage) {

  this.health -= damage;

  if(this.health <=0) {
    if(this.board.remove(this)) {
      Game.points += this.points || 100;
      this.board.add(new Explosion(this.x + this.w/2,
                                   this.y + this.h/2));
    }
  }
};
```

得分是基于每艘敌方飞船来增加的，但若敌方飞船没有设置得分属性，那么默认的分值就是 100。可以修改 enemies 蓝图，使分值基于敌方类型而变动。

重新加载游戏，现在你应能积累分数了。此外，也可以通过访问 http://mh5gd.com/ch3/score 运行游戏的最新版本。

3.5　使之成为公平的战斗

现在你将着手实现 Alien Invasion 的最后一项增强功能，那就是赋予敌方飞船一点回击的火力。

沿袭 PlayerMissile 的做法，游戏需要使用一个 EnemyMissile 对象来表示敌方飞船发射的炮弹。将代码清单 3-5 中的代码添加到 game.js 的末尾处，用以创建 EnemyMissile。

代码清单 3-5：EnemyMissile 对象

```
var EnemyMissile = function(x,y) {
  this.setup('enemy_missile',{ vy: 200, damage: 10 });
  this.x = x - this.w/2;
  this.y = y;
};

EnemyMissile.prototype = new Sprite();
EnemyMissile.prototype.type = OBJECT_ENEMY_PROJECTILE;
```

```
EnemyMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  var collision = this.board.collide(this,OBJECT_PLAYER)
  if(collision) {
    collision.hit(this.damage);
    this.board.remove(this);
  } else if(this.y > Game.height) {
      this.board.remove(this);
  }
};
```

EnemyMissile 与 PlayerMisisle 非常像一对孪生子，但前者是邪恶的那一个；它具有不同的垂直方向、不同的类型、不同的碰撞类型，以及不同的检查飞离面板的方式。这两者的功能是一样的，只是作用方向相反而已。

要在页面上显示 EnemyMissile 对象，Enemy 的 step 函数需要按照随机的时间间隔发射一些导弹。作为一种附加的复杂情况，一些敌方飞船会同时发射两枚导弹，这很像是玩家飞船的做法，而另一些就只能沿着中间直线向下发射一枚导弹。

敌方飞船导弹的精灵 enemy_missile 也是需要定义的内容，所以要把该条目添加到 game.js 头部的 sprites 列表中：

```
var sprites = {
 ship: { sx: 0, sy: 0, w: 37, h: 42, frames: 1 },
 missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 },
 enemy_purple: { sx: 37, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_bee: { sx: 79, sy: 0, w: 37, h: 43, frames: 1 },
 enemy_ship: { sx: 116, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_circle: { sx: 158, sy: 0, w: 32, h: 33, frames: 1 },
 explosion: { sx: 0, sy: 64, w: 64, h: 64, frames: 12 },
 enemy_missile: { sx: 9, sy: 42, w: 3, h: 20, frame: 1 }
};
```

修改 Enemy 对象，如以下代码的突出部分所示，加入导弹的发射能力属性：

```
Enemy.prototype = new Sprite();
Enemy.prototype.type = OBJECT_ENEMY;

Enemy.prototype.baseParameters =
           { A: 0, B: 0, C: 0, D: 0,
             E: 0, F: 0, G: 0, H: 0,
             t: 0, firePercentage: 0.01,
             reloadTime: 0.75, reload: 0 };

Enemy.prototype.step = function(dt) {
  this.t += dt;

  this.vx = this.A +
            this.B * Math.sin(this.C * this.t + this.D);
```

```
    this.vy = this.E +
            this.F * Math.sin(this.G * this.t + this.H);

    this.x += this.vx * dt;
    this.y += this.vy * dt;

    var collision = this.board.collide(this,OBJECT_PLAYER);
    if(collision) {
      collision.hit(this.damage);
      this.board.remove(this);
    }

    if(this.reload <= 0 &&
       Math.random() < this.firePercentage) {
      this.reload = this.reloadTime;
      if(this.missiles == 2) {
        this.board.add(
          new EnemyMissile(this.x+this.w-2,this.y+this.h/2)
        );
        this.board.add(
          new EnemyMissile(this.x+2,this.y+this.h/2)
        );
      } else {
        this.board.add(
          new EnemyMissile(this.x+this.w/2,this.y+this.h)
        );
      }

    }
    this.reload-=dt;

    if(this.y > Game.height ||
       this.x < -this.w ||
       this.x > Game.width) {
        this.board.remove(this);
    }
};
```

这一改变首先影响到 baseParameters，敌方飞船需要加入两个额外的属性默认值来控制发射的可能性和可能的发射速度，这两个属性分别是 firePercentage 和 reloadTime。firePercentage 是一个用来检查随机数的数值，若随机数小于 firePercentage，那么敌方飞船就发射一枚或多枚导弹。因为该方法在每步进一帧时都会被调用，所以 firePercentage 必须是一个相对较小的数值，这样才能防止敌方飞船不断射击。

接下来是 reloadTime 和 reload，这两个属性的作用与 PlayerShip 中的相应属性完全一样，都用来防止导弹被快速连续地发射出去。

除了是基于 Enemy 配置的导弹数目(1 或 2)这点之外，实际发射导弹的代码也与玩家飞船对象的对应代码相同，该部分代码需要检查是该从敌方飞船的中间位置送出一枚发射

的导弹呢，还是从敌方飞船的左右两侧送出两枚发射的导弹。与 PlayerShip 类似，Enemy 也需要防止导弹的快速连续发射，为阻止这种情况的发生，Enemy 需要检查炮弹的重装时间，只有在武器上膛后，Enemy 才会检查随机生成的数值，看看自己是否应该发射导弹。

若加载该游戏，你应会如预期所料的那样，看到敌方飞船发射导弹；不过，所有敌方飞船都以同样的频率来发射导弹，且一次只发射一枚导弹。

为调整发射频率，你需要修改放在 game.js 顶部的敌方飞船蓝本。修改 enemies 数组，让 ltr 类和 wiggle 类敌方飞船每次发射两枚导弹(以配合它们的精灵图像)，且把 straight 类和 wiggle 类敌方飞船的 firePercentage 值降至 0.001，以此防止它们一次发射太多的导弹，修改后的内容如下所示：

```
var enemies = {
  straight: { x: 0,   y: -50, sprite: 'enemy_ship', health: 10,
              E: 100, firePercentage: 0.001 },
  ltr:   { x: 0,   y: -100, sprite: 'enemy_purple', health: 10,
           B: 75, C: 1, E: 100, missiles: 2  },
  circle: { x: 250,   y: -50, sprite: 'enemy_circle', health: 10,
            A: 0,  B: -100, C: 1, E: 20,
            F: 100, G: 1, H: Math.PI/2 },
  wiggle: { x: 100, y: -50, sprite: 'enemy_bee', health: 20,
            B: 50, C: 4, E: 100, firePercentage: 0.001,
            missiles: 2 },
  step:   { x: 0,   y: -50, sprite: 'enemy_circle', health: 10,
            B: 150, C: 1.2, E: 75 }
};
```

接下来，重新加载游戏，现在出现在你面前的应该是一个具有活跃敌人的可玩游戏了。另外，也可以通过访问 http://mh5gd.com/ch3/fair/来运行该游戏。

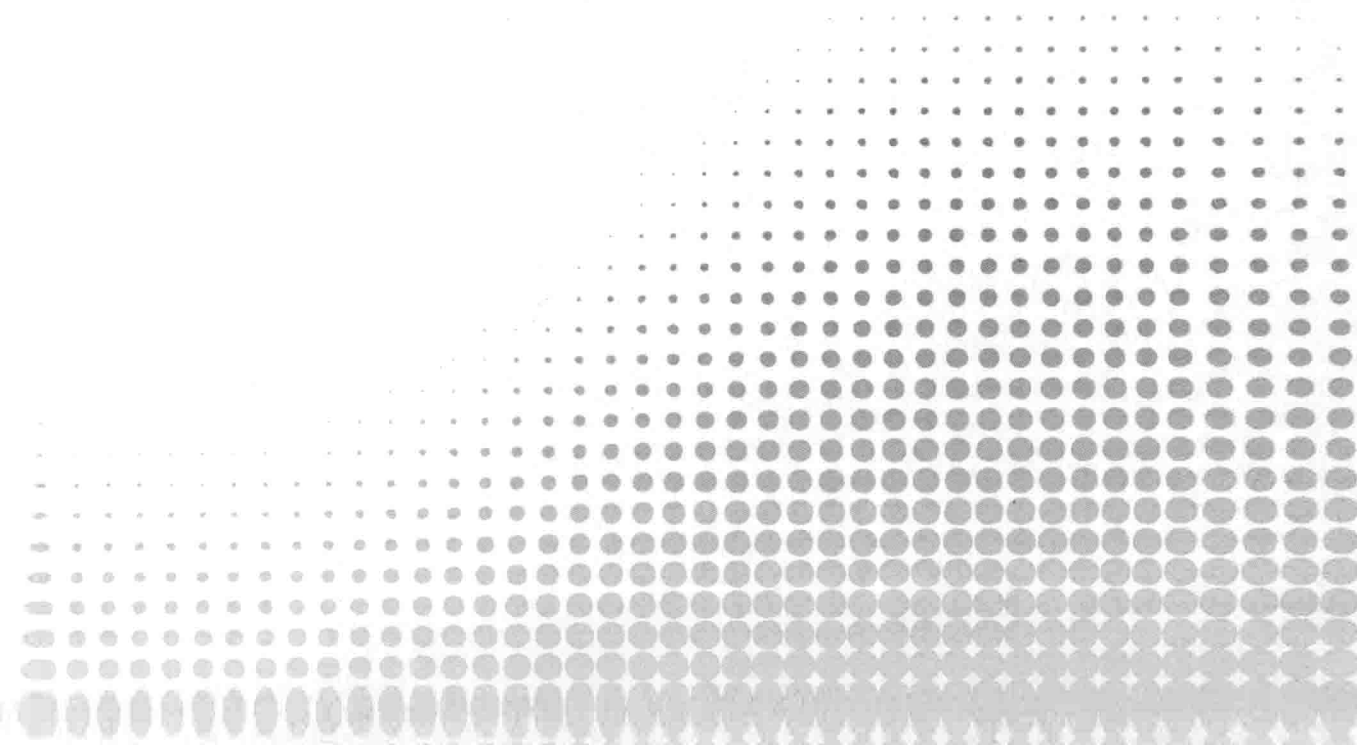
3.6　小结

从最初在画布上留下几道痕迹，到构建一个功能齐全的移动射击类游戏，整个过程发展迅速，令人目不暇接，这真是旋风式的前三章。作为演示游戏，Alien Invasion 还算不错，但作为完备的太空类射击游戏，它需要补充的功能还有许多，这其中包括高分排名、飞船动画、音效、更长且变化多端的关卡、新的敌方飞船及挑战大反派等。好消息是，可以分支和增强 Github 上的代码(https://github.com/cykod/AlienInvasion)，若有依照书中内容练习，那么你已逐行逐句地了解了该游戏的代码，并应已完全弄懂了游戏的运作机制；可通过阅读 readme 文件来了解他人已经实现的功能。

第 II 部分

移动HTML5

- 第 4 章：移动设备上的 HTML5
- 第 5 章：了解一些有用的库
- 第 6 章：成为一个良好的移动市民



第 4 章

移动设备上的 HTML5

本章提要

- 熟悉 HTML5 背后的历史
- 了解功能检测和渐进增强
- 将 HTML5 用于游戏开发
- 将 HTML5 用于移动设备
- 了解移动浏览器的状况

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 4 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

4.1 引言

HTML5 是一项卓越的技术，等到 2020 年在所有浏览器中获得全面实现时，它将会帮你领回干洗衣物、收拾公寓、遛狗，还能给世界带来和平。实际上，这些事情它一样都不会做，这些只是关于该标准的夸大之辞。不过有一样事情 HTML5 是保证会做到的，那就是：把这个世界变得更加有趣。

4.2　HTML5 的发展简史

从技术上讲，在万维网联盟(World Wide Web Consortium)将要制定和推广的一长串标准中，HTML5 就是接下来要制定和推广的标准。万维网联盟(在线访问地址 www.w3.org)，也称为 W3C，是一家标准机构，主要负责 Web 的标准化，以便不同的内容生产商和浏览器制造商能构建出包括 HTML5 在内的可正确互操作的技术。

4.2.1　了解 HTML5“不同寻常”的成长历程

HTML5 开启生命之旅的方式与它的前辈有所不同，它并非是作为 W3C 的心血结晶开始存在的。相反，从许多方面来看，HTML5 都是为反抗 W3C 当时推出的标准 XHTML 2.0 而催生的结果。

XHTML 2.0 有许多优点，若真的获得成功，它会把 Web 变成一个更加和谐的地方，而非今天的这幅充斥着坏标记(bad markup)的景象。但它具有一个致命缺陷：不向后兼容，这使得该标准实际上成为一个无望成功的计划。不向后兼容意味着若拥有一个完全有效的 HTML 4 站点，那么你得把它全部丢弃，重新从头开始编写代码，这样才能把它变成一个 XHTML 2.0 兼容的网站。此外，回溯 2004 年，当时 W3C 革新 Web 的速度慢得像蜗牛，在 2000 年，HTML 4.01 就已获发布，这是对 HTML4 标准的最后一次更新。

作为对 XHTML 2.0 标准的失望之情的响应，一个被称为 Web 超文本应用技术工作组(Web Hypertext Application Technology Working Group，WHATWG)的分支机构在 2004 年成立。WHATWG 由来自 Apple、Mozilla 和 Opera 的成员组成，它立即着手制定一个新的标准。到 2007 年，WHATWG 在过去三年制订的这个已经取得一些显著进展的标准被当成名为 HTML5 的新标准的起点提交给了 W3C，以便与 XHTML 2 同步发展。

W3C 做出了让步，开始制定 HTML5 标准，在 2008 年初推出了第一个工作草案。此外，在 2009 年，W3C 接受了 XHTML 已成为无用标准这一事实，解散了制定 XHTML 2.0 规范的工作组，放弃了该规范。

4.2.2　期待 HTML6？HTML7？不，仅 HTML5 足矣

2011 年 1 月，WHATWG 的 HTML5 规范的编辑 Ian Hickson 宣布，工作组将把 HTML5 规范当成一部“活文档”对待，规范将会被无限期持续更新和维护下去。W3C 仍会发布 HTML5 的官方快照，但随着新技术建议的出现，文档本身将会被继续更新。

这对开发者意味着什么呢？从好的方面看，许多非常酷的技术会逐渐从想法成为单个浏览器的实现，进而得到良好支持和写入规范——所需时间仅是这一过程过去要花费时间的几分之一。作为一个开发者，你将收获许多好东西，这些东西能够把构建应用变成一件乐事。从坏的方面看，进化管道会催生许多技术，如对摄像头的直接访问和对游戏手柄的支持等，但浏览器的碎片化会再次成为一个大问题，而且需要竭尽全力不停追赶正在发生的一切。

4.2.3　关于规范

关于 HTML5 规范，非常令人满意的一件事情是它是可读懂的。尽管这是一部针对浏览器实现的文档，但它也是一部对 Web 开发者极有用的文档，无论何时需要弄清楚一些东西的权威性，这部文档都值得你查一查。许多时候，相比于通过 Google 搜索答案的一般做法，该规范能更高效地回答你的问题，若你尚未读过该规范，可快速访问一下 HTML 规范的永久性站点“Living Standard”，这一站点的永久访问地址是 www.whatwg.org/html。

下一次若要查找一些内容，如 drawImage 的某种变体的参数，那么可以花一点时间来熟悉一下该规范的格式和内容的组织方式，因为它提供了最简单的方法来查找 HTML5 的具体功能，只有在遇到非标准实现或尚未成为规范的功能时，你才应该从别处入手。

4.2.4　区分 HTML5 家族和 HTML5

术语 HTML5 在 2010 年 4 月前后开始为人们所认识，当时史蒂夫·乔布斯写下了他的那封很不怎么样的信“Thoughts on Flash(关于 Flash 的一些思考)”(www.apple.com/hotnews/thoughts-on-flash)，从那时起，HTML5 就被当成许多不同标准的总术语(umbrella term)来使用，这些标准中的一些在某个时期曾是 HTML5 的组成部分，后来又被分离了出去，而其他一些则自始至终都是自有的标准。

HTML5 通常包含的一些标准如下：

- SVG
- CSS3
- WebGL
- Web Workers
- Web Storage
- Web SQL Database
- Web Sockets
- Geolocation
- Microdata
- Device API
- File API

使用 HTML5 这一总术语来包含所有这些标准，这是一种技术性错误，不过倒也相当方便。HTML5 是一个人们都知道和理解的行业用语，对于大多数开发者来说，它仅意味着在无插件的情况下在浏览器中构建一些本地化的东西，我认为这样理解是没问题的。

何时引用的是 HTML5 规范的某个特定部分，何时引用的是另一个规范，本书在这一点上是明确无误的，不过大部分情况下，本书引用的都是 HTML5 已包括在内的一系列技术。

4.3　恰当地使用 HTML5

如前所述，由于机缘的巧合，在 Web 的历史上，Web 开发者第一次拥有了一个 W3C 批准的，与日常开发实情和期望同步的官方规范(好吧，现在还是个“工作草案”)。对于因 Web 发展停滞不前而渴望新技术已久的 Web 开发者来说，当前的这一复兴活动给人的感觉像是到处翻找零碎的面包屑未果后，突然有人给递上了一块大蛋糕。

在过去几年中，被添加到 Web 开发者工具箱中的好东西数量多得惊人。即使是 Microsoft 这种深陷专有技术泥潭的传统公司也参加了进来，展开了一系列的推动工作，支持基于标准的不使用专属插件的 Web 开发。

4.3.1　尝试 HTML5

HTML5 是一个真正的浏览器驱动的规范，因为浏览器制造商是推动标准向前发展的力量，这意味着许多功能一开始以非兼容实现方式出现在一个或多个浏览器中，但随着浏览器制造商在细节和实现上达成一致而最终被统一。

这种从最初实现到标准的进化的最显著例子是 CSS3 厂商前缀(如前所述，从技术角度看，CSS3 并非 HTML5 的组成部分)。在场景突然需要新的 CSS3 功能时，你通常需要编写多行可能仅是厂商前缀稍有不同的代码。例如，在 2010 年时，为给容器添加下拉阴影，你不得不编写下面这样的代码：

```
-moz-box-shadow:5px 5px 10px #000;;
-webkit-box-shadow:5px 5px 10px #000;;
-ms-box-shadow:5px 5px 10px #000;;
-o-box-shadow:5px 5px 10px #000;;
 box-shadow:5px 5px 10px #000;;
```

这其中包括 4 个厂商特定的版本和一个基于标准的前瞻性版本。

时至 2011 年，以如下方式简单编写代码已经成为可能：

```
box-shadow:5px 5px 10px #000;
```

虽然跟踪变化的速率是一项艰巨任务，但作为一个开发者，可以同时得到两方面的好处：不仅可立刻使用全新工具(不过要谨慎使用，特别是在标准尚未完成时)，而且功能的标准化在数月内就能实现，不再需要等待数年的时间。

4.3.2　嗅探浏览器

在 Web 开发的艰苦旧时代，你可能会看到以下这样的代码散落在 HTML 文档的<head>标签段中：

警告：以下代码是一个反例，告诉我们什么不该做，所以请不要使用这两个例子中的任一种写法。

```
<!--[if IE 6]>
 <link rel='stylesheet' href='ie6.css' type='text/css'/>
<![endif]-->
<!--[if IE 7]>
 <link rel='stylesheet' href='ie7.css' type='text/css'/>
<![endif]-->
```

或者，在某些情况下你可能会这样写：

```
function isIE() {
 if(navigator.userAgent.match(/MSIE (\d+\.\d+);/i)) {
   isVersion = new Number(RegExp.$1);
   return true;
 } else {
   return false;
 }
}

if(isIE()) {
  if(ieVersion == 6) { /* IE6 only Code */ }
  else if(ieVersion == 7) { /* IE7 only Code */ }
}
```

这两段代码中的第一段称为条件注释(conditional comments)(IE 独有的功能)，而第二段则称为 UA 嗅探(UA sniffing)，因为该段代码试图通过浏览器提供的 userAgent 信息串来破译所使用的浏览器。

尽管在 2007 年时，使用这两段代码来确定应用程序应有的行为可能是一个可行的解决方案，在当时要兼容的三个浏览器版本——IE6、IE7 和 Firefox 2——已占据了近 95%的市场份额，但当前的这一轮浏览器大战已显著改变了这一格局，如图 4-1 所示，其中的数据源于 statcounter.com。

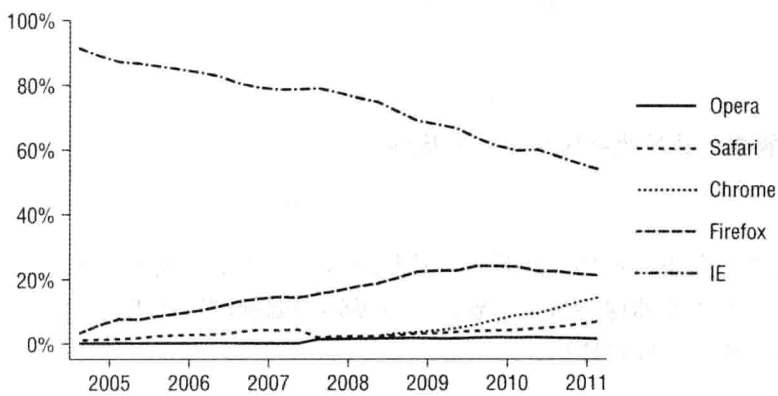


图 4-1　桌面浏览器的使用份额

截止撰写本书之时，桌面 IE 所占据的浏览器份额已经下滑至刚超过 50%，而且要由 4 个不同的版本——IE6、IE7、IE8 和 IE9 来瓜分，与此同时，IE10 的发布也近在眼前。Chrome

的市场份额则在上升，目前已达两位数；而 Firefox，虽然小有下滑，但依然超过 20%。Safari 处在缓慢的稳步上升趋势中，若苹果公司继续保持强劲的发展势头，它可能会达到两位数。除了 IE，其他浏览器的大部分用户都很有可能会使用当前的版本或是前两个版本之一，这是因为这些浏览器现在都在积极推动自动更新这种做法。IE 也推出了自动更新，但较旧操作系统的用户所能使用的浏览器受到了限制(Windows 2000 的最高版本是 IE6，Windows XP 的最高版本是 IE8，Windows Vista 的最高版本是 IE9)。

尽管可以自动更新，但考虑到所有浏览器仍在使用旧版本，所以你现在看到的就是，有超过 15 种桌面浏览器和版本的组合需要支持。更糟的是，本书要讨论的主要对象可不是桌面浏览器，对吧？在移动设备方面，在美国占据主导地位的浏览器竞争对手是 iOS 和 Android，与此同时，Firefox、Windows Phone 7 (WP7)、Opera 和 Blackberry 也有获胜的可能。诸多不同的设备和屏幕尺寸、正在兴起的平板电脑、Android 的各个分支，把这些东西都加进来，你的脑袋应该开始有些晕了吧。

也许你现在就可以开始编写一些条件注释以及进行浏览器嗅探，但这一过程永远不会完结，因为随时随地都会有新的设备被发布到市场中来。

4.3.3　确定功能而非浏览器

不过，这其中也不全是悲观和沮丧，若方法得当，你是可以开发出既适用于现在也适用于将来的应用的。这一方法的最重要之处就在于这一想法，即测试浏览器的功能而非试图去识别浏览器本身。

比如，一种检查支持的初始简单做法(这在 HTML5 出现之前常会见到)是搜索互联网来查明哪些浏览器支持 Canvas 标签，然后对版本进行匹配。你可能试过写一些如下代码：

```
var canvasSupported = (isIE() && ieVersion >= 9) ||
                      (isFirefox() && ffVersion >= 1.5) ||
                      (isOpera() && oVersion >= 9) ||
                       ... And so on ..
```

警告：这段代码同样是一个反例。

正如你可能已发现的那样，这是一种糟糕的做法。不足之处在于需要对大家正在使用的所有不同类型的浏览器保持最新了解，并且要知道哪种浏览器支持哪些功能，这简直就是在制造一场灾难。若仅这样写：

```
function isCanvasSupported() {
   // Directly check if the browser knows about the canvas element
}
```

那么日子就要好过多了。实际上，可以就这样写，可以只针对 HTML5 中那些你所关心的每项功能这样编写代码。

其中最常见的方法是尝试创建一个元素并检查它是否支持某个方法；又或者，若功能并非针对 DOM 元素，那么可以仅检查浏览器是否拥有必需的方法。以下为每种情况列举一个例子：

```
function isCanvasSupported(){
  // First create a <canvas> element
  var elem = document.createElement('canvas');

  // Next make sure it supports getContext and can return a 2D context
  return !!(elem.getContext && elem.getContext('2d'));
}

function isGeolocationSupported() {
  // Check if the geolocation object is defined
  return navigator.geolocation;
}
```

现在，在你打算去查阅规范和为每项想要支持的功能编写函数之前，我要告诉你，已有隐身在 Modernizr(www.modernizr.com)背后的好心人帮你完成了所有的这些繁重工作，你要付出的代价就是加载一个以压缩方式提供的大小不超过 7KB 的脚本。如果想进一步压缩该文件的大小，可在自定义下载时只选择所需的部分。

Modernizr 能够帮助你把功能检测代码简化成如下写法：

```
if(Modernizr.canvas) {
  // Do something with canvas
}
```

Modernizr 提供了 40 多项对下一代功能是否得到支持的检查，并会经常增加一些检查，因此，它应该是你的必备资源。

4.3.4　渐进增强

现在，你已能够做到确切知道运行代码的浏览器支持哪些功能，那么如何才能尽量利用这些信息呢？答案是：“要视情况而定。”

某些情况下，缺乏对某个特定功能的支持就相当于成功无望。例如，若游戏依赖于画布，那么缺乏对画布的支持就意味着游戏不可能运行。在其他一些情况下，某个功能的缺失可能会降低用户的体验，但应该不会妨碍到用户玩游戏。

现实世界中的渐进增强

我所在的公司曾为 GamesForLanguage.com 构建了一个名为 SpaceWords 的游戏，该游戏让多个玩家使用它们的智能手机作为游戏的控制器。若浏览器支持方位事件，游戏就会使用这些事件作为移动输入，允许玩家通过旋转手机来控制飞船；若智能手机不支持这些事件，游戏就使用触摸事件；最后，若手机不支持触摸事件，游戏就使用普通的 MouseDown 事件。

在以上每种情况中，能用上更多的功能就意味着可以给玩家带来更好的体验，同时又不会把那些使用功能较少的设备的玩家完全拒之游戏门外。渐进增强(progressive enhancement)是一个行业用语，意思是从功能的基础级开始，然后只为支持额外功能的那些设备加入这些功能。

在一些最好的情况下，同一份代码既可用于最小支持也可用于增强的功能，你只需在其中包含一些用于增强体验的额外代码即可。由于大部分开发者都在一些功能最好的设备上进行开发和测试，这意味着相对于使用两个完全不同的代码库，你更有可能获得可在功能较少的设备上正常工作的代码。

4.3.5　弥补差距的腻子脚本

上一节谈到，某些情况下，某一特定功能的缺失如何会让成功无望，实际上，事情并非完全如此，因为某些时候你可能会找到一个腻子脚本来填补这一缺失。

腻子脚本(polyfill)就是一大块代码，这些代码把不被浏览器原生支持的功能反向移植回这些浏览器中。对于移动设备来说，这可能不会是个问题，因为只要使用的是新近推出的智能手机，那么它的移动栈(mobile stack)一般来说就都是良好的。

不过，一旦离开 iOS、Android 和 WP7 领域，踏入黑莓(Blackberry)和功能性手机的地界，你所获得的支持就会减少。在这些情况下，诸如用于本地存储和 CSS3 功能的一类腻子脚本就会变得非常有用。

检测功能(而非浏览器)的好处也再次凸显出来，因为真正的腻子脚本会暴露与原生功能一样的接口，这意味着你通常可以把一个腻子脚本添加到文档顶部，为较旧的浏览器无偿添加一些增强的功能。

例如，CSS3 Pie(http://css3pie.com/)就提供了一个腻子脚本把圆角、渐变和下拉阴影一类的 CSS3 装饰添加至较旧版本的 IE 中。

Modernizr在它的维基(wiki)页面上提供了一个内容非常丰富的腻子脚本列表：https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills，鉴于其兴趣所在，这毫不奇怪。

4.4　从游戏角度考虑 HTML5

在本章中，你已经了解了许多关于 HTML5 的内容，但对于 HTML5 如何帮助你在移动设备上构建游戏所知不多。本节讨论一些使用 HTML5 构建游戏的主要做法，下一节则会谈及一些可让游戏变得更有趣的移动友好功能。现在，先从 HTML5 角度简单了解一下三种可用来构建游戏的做法(后续各章将更详细地讨论这些做法中的每一种，第 12 章解释应为特定游戏挑选哪一种做法)。

4.4.1　画布

画布(Canvas)虽在第 15 章中才会被详细讨论，但在本书各处都被用到。若论游戏，从许多方面来说，画布都是现在的 HTML5 标准中最有吸引力的功能，它赋予了开发者访问可在其中绘制图像和图形的高速帧缓冲区的能力。

本书关于画布的大多数讨论指的都是 2D 画布，因为这是唯一在当前移动设备上获得很好支持的画布上下文，但也会有一些简单的讨论是关于 WebGL 上下文及对 3D 画布的支持的，3D 画布已做好准备，随时都有可能出现在移动设备上(也许等到本书出现在你手中时，3D 画布也已出现在移动设备上了)。图 4-2 展示的是在第 18 章中创建的画布平台动作游戏。

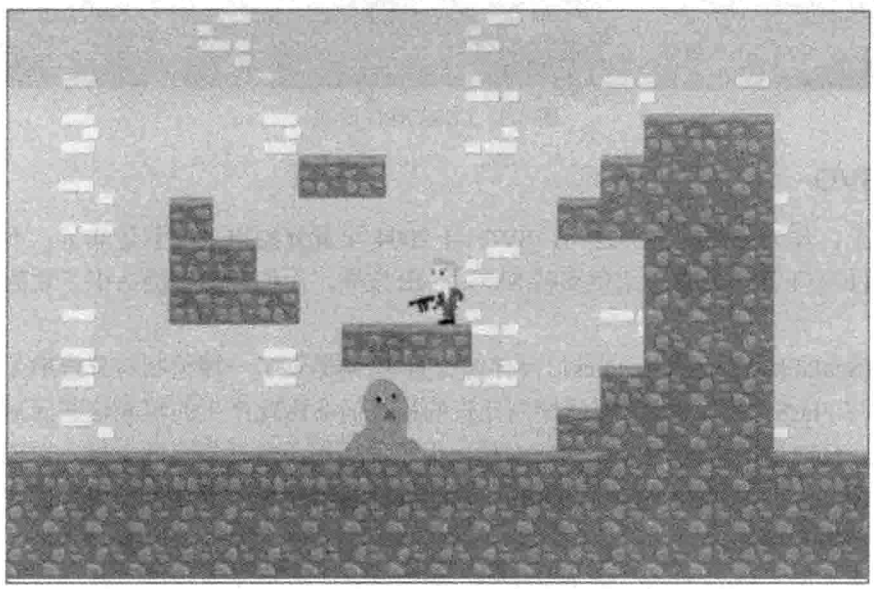


图 4-2　移动设备上的 2D 平台动作游戏

4.4.2　CSS3/DOM

正如你将在第 12 章中见到的那样，CSS3 为传统的 HTML DOM 元素增添了许多强大的功能，完全可被看作一种构建游戏的可行技术。对于使用触摸界面(也即移动设备)的游戏来说，使用 CSS3 和 DOM 所获得的速度可能会更快一些，因为浏览器自动处理了许多单击检测和交互。此外，移动 WebKit 支持硬件加速的变形和动画，这在许多情况下提升了画布的性能。图 4-3 展示的是在第 13 章中构建的 CSS3 RPG 游戏。

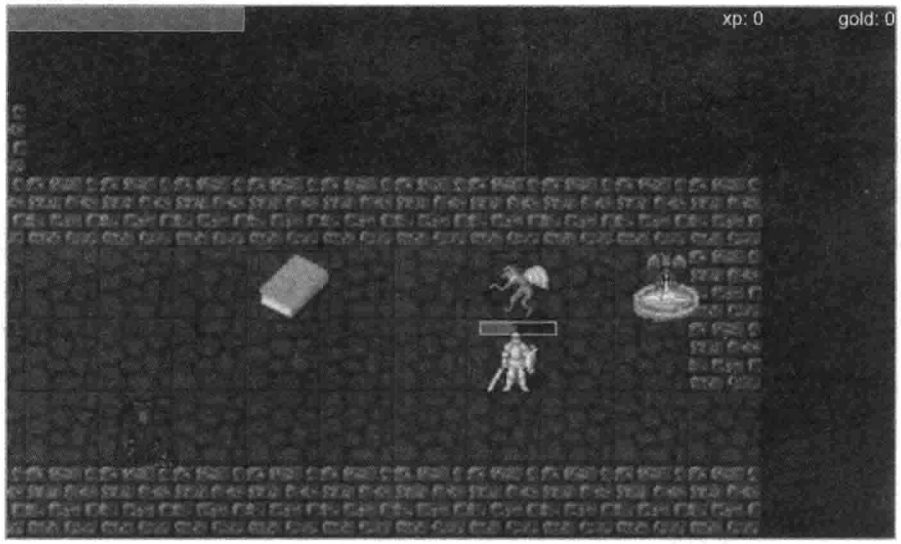


图 4-3　CSS3 RPG 游戏

4.4.3　SVG

实际上，作为家族的害群之马，SVG 自 2004 年就开始出来招惹是非了。不过，随着 IE9 在 2011 年 4 月作为首个原生支持 SVG 的 IE 发布，人们开始慢慢认识了它的好处和潜在用途。

SVG(Scalable Vector Graphics，可伸缩矢量图形)提供了一种绘制与分辨率无关的图形的方式，与 Flash 相似，SVG 存储绘制元素的指令而非由此产生的组成这些元素的像素。这意味着，只要使用正确，SVG 能够创建出很小的适用于移动设备的文件。它还能生成自己的场景图——非常类似于标准 DOM——在 WebKit 移动设备上获得了很好的支持而且在触摸游戏中很有用处。

在许多方面，SVG 结合了画布和 CSS3 两者之长，它的主要问题是性能有所欠缺，所以，若你正在构建的是动作游戏，那么 SVG 可能不是一种选择。图 4-4 展示的是在第 14 章中构建的 SVG 游戏。

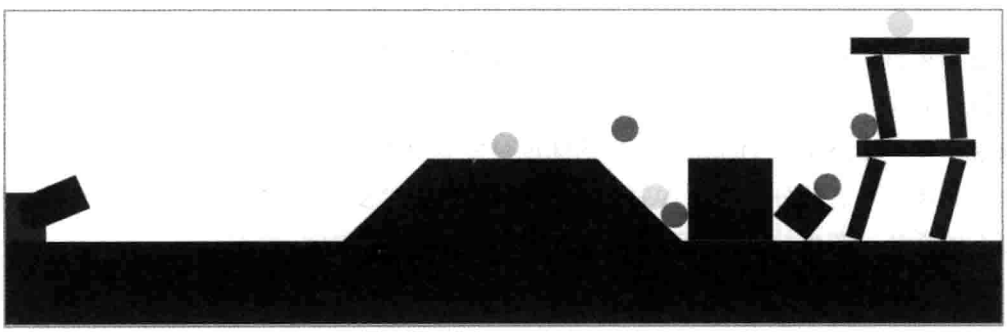


图 4-4　SVG 大炮射击游戏

4.5　从移动角度考虑 HTML5

现如今常听到“移动先行(mobile first)”这样的说法，这一措词的意思是把移动设备当成主要的考虑目标，之后再考虑对其他一些浏览器的支持。这可能在移动浏览器方面体现得不是很明显，截至 2012 年，移动浏览器的总使用率勉强达到两位数。不过，这不仅是一个数字游戏。首先考虑移动设备意味着先从一些重大的制约因素入手，然后由此找出解决之道。

移动设备具有很小的屏幕、不同的屏幕尺寸和纵横比，它们有着有限的处理能力和带宽，内存通常也很有限。所有这些约束都会强制开发者去优化自己的 Web 应用——把它变得更具适应性，让它拥有更快的加载速度和更好的性能。毫不奇怪，所有这些事情都能给用户和桌面浏览器带来更好的体验。Web 开发者一直都有点懒，他们在连接到互联网的 30 英寸显示器上查看项目，在 4 核的多线程处理器上运行代码。休闲类网页游戏目标人群中的相当一部分很可能不会使用同样的硬件规格，把网站的移动用途作为主要因素加以考虑有助于你远离这样的一种态度。

4.5.1　了解一些新的 API

作为 HTML5 捆绑在一起的规范家族是一个很大的家族，从游戏的角度来看，有些部分是你不能不关心的。语义方面的进步、诸如<aside>一类的标签和对微数据的支持实际上会有助于构建出成功的游戏吗？也许不会，但许多令人兴奋的应用编程接口(Application Programming Interfaces，API)是你应该关心的，从游戏的角度看，这其中最令人感兴趣的那些如表 4-1 所示。

表 4-1　一些新的 HTML5 API

API 名 称	说　明
Touch(触摸)	大多数移动设备都支持触摸事件，这些事件的行为与它们的等价鼠标事件非常类似，但会增加一些触摸跟踪和多点触控信息，这意味着可以一次跟踪多个触摸，对于需要拥有屏幕控件的动作游戏来说，这一点很重要
Orientation 和 Acceleration (方向和加速)	支持在用户来回移动手机时检测方向变化和加速事件
Application Cache(应用缓存)	允许离线缓冲游戏所需的资产，这意味着可以把游戏配置成在无任何互联网访问的情况下启动
Offline Storage(离线存储)	提供本地存储游戏状态的能力，这意味着你不必经常把每样东西保存到服务器上，可以在不需要访问服务器的情况下，让玩家从他们上次停下的地方重返游戏
Geolocation(地理定位)	具有检测玩家在现实世界中的地理位置的能力

4.5.2　即将登场的 WebAPI

尽管一些可用的 API 提供了访问移动设备上的许多功能的能力，但截止撰写本书之时，一些显著的差距依然存在。Mozilla 的 WebAPI 项目旨在填补这些差距，提供一些一般来说只提供给本地应用的访问硬件和操作系统资源的能力，像访问摄像头、文件系统和震动马达这类很酷的技术，就有可能会促成一些可围绕其来构建游戏的有趣功能。Mozilla 的维基站点 https://wiki.mozilla.org/WebAPI 记录了 WebAPI 的进展情况。

4.6　调查移动浏览器的前景

在 iPhone 和 Mobile Safari 在 2007 年问世之前，手机上的网页浏览状况可谓凄惨，只有 Opera 提供移动浏览体验，而且并未将移动浏览看成有限的菜单驱动体验。在这之后的几年中，移动浏览器的发展异常迅猛，其功能已变得更为丰富，这些浏览器上的 JavaScript 引擎已经变得更好用。

能像在桌面浏览器上那样在手机上玩同样的 Web 游戏？伴随着每一个新的 HTML5 游戏出场，这一在 5 年前不可想象的事情正在慢慢成为一种现实。尽管硬件很重要，但与桌面的情况非常类似，运行在目标设备上的浏览器是可用功能最重要的仲裁者。

4.6.1　WebKit：市场霸主

好消息是与桌面不同，两个占主导地位的智能手机平台——iOS 和 Android——都有着非常出色的移动浏览器，更妙的是，它们共享 WebKit 引擎，这意味着可以期待这两个浏览器之间有一组相近的功能和相近的渲染引擎。WebKit 并非唯一在用的移动 HTML5 浏览器，但根据 www.statcounter.com 提供的资料，截至 2011 年 12 月，WebKit 浏览器的确占有了逾 80%的美国移动通信量。

WebKit 一开始是作为开源的 Konquerer 网页浏览器内部的 KHTML 渲染引擎和 KDE JavaScript 引擎(KJS)存在的。苹果公司采用(并分支)KHTML 和 KJS，并把它们重命名为 WebCore 和 JavaScriptCore，但仍置于 WebKit 的庇佑之下。

由于 KHTML 和 KJS(也即 WebCore 和 JavaScriptCore)最初是在 GNU 的宽通用公共许可证(Lesser General Public License，LGPL)下发布的，包括苹果公司在内的公司在发布源自这些技术的产品时，都需要把对 WebKit 做出修改的源代码发布回开源社区。这意味着因为该项目一直保持开源，包括 Google、Nokia、Blackberry、Amazon 和 HP 在内的许多移动设备厂商已经将其用作移动浏览器的基础，并一直在把他们的修改发布回社区，这一做法全面提升了移动浏览的品质。

如前所述，这两个占主导地位的、拥有支持 HTML5 的浏览器的智能手机平台——iOS 和 Android——都使用基于 WebKit 的浏览器作为默认的设备浏览器。不过，Android 并未使用 JavaScriptCore，而是使用 Google 的 V8 JavaScript 引擎来执行 JavaScript。

WebKit 也用在 Safari 这一苹果公司的默认浏览器和 Google 的 Chrome 中，因此，在进

行移动开发时，最好使用 Chrome 或 Safari，比起使用 Firefox 或 IE 来，这样做能获得更接近主流移动市场的结果。由于 Chrome 中的开发工具是同类中最好的，因此可考虑使用 Chrome 作为主要的开发浏览器。

一个世界级浏览器引擎的背后是 80%的市场份额，尽管对于开发者来说，这可能梦想成真，但实情却是，并非所有的 WebKit 浏览器都是一样的。Quirksmode.org 在比较了各种移动 WebKit 版本之间的差异后宣布：

不存在“移动设备上的 WebKit”这回事。

可在 Quirksmode.org 网站上查看完整的报告：www.quirksmode.org/webkit_mobile.html。

4.6.2　Opera：依然在埋头苦干

Opera 很早就进入了移动领域，在 2000 年推出了 Opera Mobile。2005 年发布的 Opera Mini 是移动设备上的第一个实用浏览器，该浏览器使用一个中间服务器来处理和压缩请求，从而提高了当时的低带宽和低功率设备的传送速度。Opera 后来已经失去了其在美国的市场主导地位，但在世界范围内，Opera 的所有产品仍然占据着 24.5%的市场份额，这使得它成为世界上最流行的移动浏览器。不过这一情况可能不会持续太久。

4.6.3　Firefox：Mozilla 的移动产品

Mozilla 的 Android 移动浏览器 Fennec(苹果公司禁止在 iPhone 或 iPod 上安装别的浏览器)非常先进，它支持许多大家都想要的 HTML5 功能，其中包括诸如多点触控和方向支持等一些 Android 和一些默认的基于 WebKit 的浏览器所缺少的功能。

不过，你可能对 Firefox 在桌面领域的发展道路有所感触，作为互联网的监察机构，Mozilla 在社区中起到了重要作用，它确保普通网民的关注不会被淹没在主导网络的企业利益的喧嚣声中，拥有一个移动设备上的产品确保了没有接受 Mozilla 建议的下一代设备不会被创造出来。

4.6.4　WP7 上的 Internet Explorer 9

尽管 Windows Phone 7 在发布时获得了普遍好评，但它还未占据显著的市场份额。7.5 版本使用与桌面 IE9 相同的渲染引擎 Trident 5.0，支持 SVG、HTML5、CSS3 和提升 JavaScript 性能的 JIT 编译。不过，与桌面类似，IE 有时会坚持自己的那一套。所以，如果希望支持 IE9(在应该要这样做时)，那么在开发过程的早期，你至少应该在模拟器上运行应用，并尽早及经常在真实的硬件上测试应用。

4.6.5　平板电脑

在进行任何关于手机的讨论时，都不要忘了平板电脑：一个正好落在传统移动设备和桌面之间的正在快速成长的细分市场。平板电脑有着自己的问题，一般来说，与桌面相比，它们的动力不足，却比手机有着更高的屏幕分辨率，这意味着游戏要在不必使用硬件进行

备份的情况下填充许多像素。好消息是，在大多数流行的iOS和Android平板设备上，WebKit同样占据了浏览器领域的主导地位。针对这些平板设备，你可应用同样的规则，即测试其对不同功能和设备屏幕尺寸的支持。

Android 带来了一项特别的挑战，因为与 iOS 的模拟器相比，它的模拟器很慢，所以要高效地测试种类极其繁多的 Android 设备可不是一件容易的事情。

4.7　小结

移动浏览仍处在不断发展之中，无论从技术角度还是从用户角度看都是如此。移动浏览器领域仍在以可与桌面领域相媲美(若不是更快的话)的速度不断发展着，到本书上架待售之日，任何关于哪些 HTML5 功能在哪些移动浏览器中获得了支持的具体说法都有可能已过时。幸运的是，你已知道不要绑定到某些特定的浏览器上，而要与这些浏览器自我宣称的那些功能绑定起来。

永远都不要依赖图表中的信息来了解可在每种设备上启用哪些游戏功能；始终要溯本求源，直接检查或使用 Modernizr 来检查浏览器的功能。只有当需要了解应花时间来把哪些功能添加到游戏中，以及是哪部分浏览器实现了这些功能时，你才应该查看浏览器的功能表。若想获取一些很好的概述信息，你可访问诸如 http://mobilehtml5.org 之类的站点，不过最好的做法始终是在浏览器中进行尝试。

第 5 章

了解一些有用的库

本章提要

- 了解 jQuery
- 了解回调和事件
- 使用 Underscore.js 实用工具库

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 5 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

5.1 引言

在 Ajax 出现之前，阻止 JavaScript 成为交互式跨浏览器应用开发的可行平台的最大障碍是各种浏览器之间的细微差别。跨各种浏览器的各式各样功能的不一致实现(Internet Explorer 的罪过尤其深重)意味着：要与网页交互，开发者需要了解每个浏览器的底细，然后通过修改代码来处理不同的非兼容实现。这一困难加上操纵页面元素和发起异步 Web 调用的繁冗，难怪相对于使用 Java 和 Flash 所达成的事情，人们会将 JavaScript 看成不太实用的东西。但随着一些把开发者经验变得更趋一致以及把 JavaScript 代码变得更简洁的库开始获得广泛使用，这一观点也开始有所改变。

5.2　了解 JavaScript 库

jQuery 以迅猛的速度成为最受欢迎的 JavaScript 库，已被全世界超过 40%的网站所使用(参见 http://w3techs.com/technologies/overview/javascript_library/all)。它获得普及的原因主要有两个：擅长本职工作和名称空间的友好性，这意味着你从不必担心 jQuery 会影响到其他代码。你将在本章中了解 jQuery，然后在本书各处使用它。

你还将了解到另一个较小的名为 Underscore.js 的库，鉴于 jQuery 主要关注 DOM 元素的操纵和发起 Ajax 调用，Underscore.js 提供了一些实用函数，这些函数把 JavaScript 变成了一种对开发者更友好的语言，特别是在函数式编程方面。自称为 JavaScript 的实用工具腰带，Underscore 是这样描述自己的："……是搭配 jQuery 这件礼服的领带。"它用到了许多与 jQuery 一样的惯例做法，而且也是名称空间友好的。

需要用到库吗？

从技术角度看，在没有使用任何类型的 JavaScript 库的情况下，你也能够勉强应付。没有什么你打算使用 jQuery 或 Underscore 来实现的事情是直接使用 JavaScript 做不到的，库只不过是减少了许多痛苦而已。若使用 JavaScript，你最终得到的是更长更难以理解的代码，并且需要编写更多分支语句来处理不同的浏览器实现。

5.3　从 jQuery 谈起

若已经熟悉 jQuery，那么你可能会想跳过这一节。不过，你至少应快速翻阅一下其中的内容，因为本节是从一个游戏特定的角度来讨论 jQuery 的。

5.3.1　将 jQuery 添加到页面

jQuery 库由单个 JavaScript 文件构成，要把 jQuery 加载到页面中，需要借助<script>标签来加载该文件。有两种可选做法可用来实现这一点：可以直接下载 jQuery，或通过内容分发网络(Content Delivery Network，CDN)来加载。

直接加载 jQuery 意味着你拥有了该文件和页面的完全控制权，这既是好事也是坏事。jQuery 是一个大型的库，若未以(精简和压缩的)正确方式提供，那它可是一头巨兽。jQuery 1.7 的加载大小约为 250KB，不过，若以压缩和精简方式提供，则只有 33KB。若直接在 Web 服务器上提供该库，需要确保自己提供的是精简板，且已正确设置了它的缓存头，这样代码才不会在随后重载时被发送(请参阅下一章以了解更多关于缓存头的详细信息)。

使用 CDN 的另一大优势是，大部分 CDN 在全球各地都有边缘节点位置(edge-location)，这意味着无论游戏玩家住在世界上的哪个地方，他们的附近都会有一台与互联网有着高速连接的 CDN 服务器。在通过普通的 Web 服务器提供文件时，这可不是始终都能保证做到的事情。

使用 CDN 的最后一个优势是，鉴于 jQuery 的普及，很可能访问者的浏览器中已经缓存了 jQuery 的一个 CDN 版本，而这就意味不需要发送任何数据。

要通过 Google CDN 来提供 jQuery，所需要做的就是把一个<script>标签添加到页面中：

```
<script
src='http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js'>
</script>
```

jQuery 网站(http://docs.jquery.com/Downloading_jQuery#CDN_Hosted_jQuery)上有许多可选的下载项，但托管在 Google 上的这个是目前最流行的。要在本地提供该库，需要从 http://jquery.com 下载该文件，然后把它固定放在某个可以访问到的地方：

```
<script src='js/jquery.min.js'></script>
```

至于应该把<script>加在页面上的哪个地方，这完全取决于你自己，唯一的限制是应该把它放在对 jQuery 对象的首次引用之前。大多数使用 jQuery 来增强页面的网站都建议把它放在页面的底部，置于</body>标签之前。对于游戏来说，这不是什么重要的事情，因为在 jQuery 和其他 JavaScript 脚本完成加载之前，页面上不会有任何可见的东西存在。

5.3.2　了解$操作符

所有被加载到页面中的 JavaScript 都共享一个全局名称空间，这意味着若某个 JavaScript 文件定义了一个顶层变量，那么该变量可被后继加载的 JavaScript 文件访问和覆盖。

有时这是一件好事，例如在编写游戏时，把代码分成几个文件存放之后，仍能让它们合在一起工作，这真的很不错。不过，在编写库时，若能做到不必担心库中的每个方法与游戏中的方法发生冲突的话，那就太好了。例如，你可能希望将 hide()方法添加到游戏对象中，但又不能让该方法与 DOM 操纵库提供的 hide()方法发生冲突。

jQuery 解决这一问题的做法是把它所要做的每件事情都封装到一个名为 jQuery 的对象中，这样的话，若希望隐藏页面上一个 ID 为“"my-object-id”的元素(且并不存在其他任何 jQuery)，那么可以这样写：

```
var elem = document.getElementById("my-object-id");
jQuery(elem).hide();
```

jQuery 还藏了额外的几招，首先，可使用$对象来代替输入整个词 jQuery，以下代码与之前所写的代码是完全等同的：

```
var elem = document.getElementById("my-object-id");
$(elem).hide();
```

即便如此，你仍未见识到 jQuery 最强大的部分，那就是 CSS 选择器。相对于把一个真正的 DOM 对象传递给 jQuery，可仅传递一个代表了 CSS 选择器的字符串，jQuery 会找出一个或多个与选择器相匹配的对象，这样你就可以把上述代码简化成一行：

```
$("#my-object-id").hide();
```

若还记得之前用过的 CSS，那么其中那些在某项内容前面加上一个井号(#)前缀的做法就意味着你的目标是 ID 属性。

jQuery 还支持一些更复杂的选择器，例如，可以这样写：

```
$("#my-form > input[type=checkbox]:checked").hide();
```

该行代码将会隐藏所有当前被选中的，直接位于 ID 为 my-form 的元素内部的多选框。

5.3.3　操纵 DOM

jQuery 擅长做的事大多都与“操纵 DOM”有关，“操纵 DOM”意指修改网页的结构和内容。术语 DOM 所代表的意思是文档对象模型(Document Object Model)，该模型是网页的编程接口。

jQuery 的 hide()方法仅是 jQuery 提供来操纵 DOM 的众多函数中的一个，其中少数的方法(attr、css、animate、val、html 和 text)拥有两种形式：“读取器(getter)”形式和“设置器(setter)”形式。读取器形式通常不接收任何参数，或只接收一个字符串参数，并会返回一个值；设置器形式通常会接收一两个参数：接收一个属性和一个值，或接收一个对象字面量来一次设置多个值。以下是一些使用了各种不同形式的例子：

```
// Return the href of the first link on the page
var myHref= $("a").attr('href');

// Set the href for all links on the page
$("a").attr("href","http://www.google.com");

// Set the href and target for all links on the page
$("a").attr({ href: "http://www.google.com",
              target: "_blank" });

// Return the HTML in the first paragraph element
var myHTML = $("p").html();

// Set the HTML on all paragraph elements
var myHTML = $("p").html("Lorem ipsum dolor sic amet");
```

读取器版本只返回与集合匹配的第一个元素的适当值，但设置器却为所有匹配的元素设置值。

以下列表给出了一些可供使用的最常见 jQuery 方法及说明。

- $(selector).show()/$(selector).hide()：在页面上显示和隐藏匹配的元素。
- $(selector).addClass(name)/$(selector).removeClass(name)：添加或删除某个元素的 CSS 类。
- $(selector).empty()：清空某个元素的所有内容。
- $(selector).val(newValue)/$(selector).val()：设置或返回某个输入元素的值。

- $(selector).attr(attributeName)/$(selector).attr(attributeName,value)/$(selector).attr(attributeHash)：检索或设置某个对象的任意特性。
- $(selector).css(propertyName)/$(selector).css(propertyName,value)/$(selector).css(propertyHash)：检索或设置某个 DOM 对象的任意 CSS 样式。
- $(selector).animate(propertyHash)：随着时间的推移来改变 CSS 样式。
- $(selector).fadeIn()/$(selector).fadeOut()：把某个元素淡入或淡出视图。
- $(selector).append(content)/$(content).appendTo(selector)：把某些内容添加到某个容器的末尾处。
- $(selector).html()/$(selector).html(newHtml)：直接获取或设置某个元素的 HTML。

养成阅读文档的习惯

本节仅触及了一小部分可用的方法，要养成访问 jQuery 文档站点 http://api.jquery.com 的习惯，因为对于你来说，越是容易从文档中找到做法，在构建动态 Web 界面时事情就会变得越容易。

在第 12 章中，在使用 DOM 元素构建角色扮演游戏(RPG)时，将重温这其中的许多 jQuery 方法。

使用诸如 Zepto.js 一类的 jQuery 替代实现

jQuery 的最大优势是它统一了跨 IE6 之后所有不同浏览器的编程体验，然而，它的最大优势也带来了两个很大的弱点：庞大且速度可能会变慢。Zepto.js 是一个由 JavaScript 高手 Thomas Fuchs 创建的库，其目的是提供一个有着类 jQuery 语法但又不像 jQuery 那么臃肿的库。

基本上可以这样说，Zepto.js 是一个侧重于 Internet Explorer (IE)之外的其他现代浏览器的简易替代，因为它有着清晰的目标，那就是支持基于 Webkit 的浏览器和 Firefox 浏览器，目前经缩减后的 JavaScript 代码不到 6KB，几乎在所有移动设备上都能工作得很好(WP7 除外)。Zepto.js 不能用在 IE 上，包括在 IE9 上也是如此，所以，若桌面浏览器也是游戏的目标，jQuery 仍然是更好的选择。

若希望保证加载的速度及降低文件的大小，但仍然能获得 jQuery 所带来的便利，那么可通过站点 http://zeptojs.com/了解一下 Zepto.js。

5.3.4　创建回调

对于从其他一些语言转向 JavaScript 的开发者来说，他们通常不太能接受的一个最显著的 JavaScript 特色就是函数被当成一等对象对待的这种想法。尽管大多数的其他语言都支持回调机制，但无处不在的回调以及将函数作为参数传递的做法还是让 JavaScript 显得有些与众不同。

如你所知，JavaScript 中的函数是使用 function()关键字来创建的，函数可以有名称，例如：

```
function sayPhrase() { ... }
```

也可以是匿名的：

```
function() { ... }
```

匿名函数能带来什么好处呢？如前所述，它们能够被放在参数中进行传递，也可被当成普通对象对待。若希望保存匿名函数以备后用，可以这样写：

```
var sayPhrase = function() { ... }
```

这种做法还证明了这一观点，即函数可被当成普通对象对待，可像任何其他值一样被赋值给变量(本书在大部分情况下给出的都是后一种形式，目的就是为了表明函数和其他任何对象是一样的，都可被传递)。

要调用 sayPhrase，需要把一对括号附加在名称的后面。不过，若不想调用该函数，只想把它作为回调传给另一个函数，就要去掉括号。

```
sayPhrase(); // Call sayPhrase

// Pass sayPhrase as a callback to another function
otherFunction(sayPhrase);
```

在了解到函数在 jQuery 中作为一等对象存在之后，要谈到的第二个微妙之处则是与调用函数相关的。如你所知，JavaScript 是一种面向对象(Object-Oriented，OO)的语言，虽然不是典型的那种。对象则是一些结合了数据和用来与这些数据交互的方法的数据结构。

在 JavaScript 中，一种标准的、基于对象的方法调用如下所示：

```
bobObj.sayPhrase();
```

在上述代码中，bobObj 是一个包含 sayPhrase 方法的对象。与大多数 OO 语言类似，可使用一个特殊关键字来引用当前对象。不过与大多数其他语言不同，在 JavaScript 中，当前对象(通常称为上下文(context))的可塑性比其他语言中的要强得多，例如：

```
var bobObj = {
  phrase: "Yay!",
  sayPhrase: function() { alert(this.phrase); }
 };

// Will pop up an alert with "Yay!"
bobObj.sayPhrase();

// Will pop up an alert with "undefined" after 100ms
setTimeout(bobObj.sayPhrase, 100);
```

在第一个例子中，一切都按预期工作，this 这一关键字被绑定到了你所预期的对象 bobObj 上；然而，在第二个例子中，函数 sayPhrase 的上下文丢失了，因为用到的是回调中的上下文。可以采用两种方法来解决这一问题，第一种是不要使用 this 关键字，直接引

用对象：

```
bobObj = {
   phrase: "Yay!",
   sayPhrase: function() { alert(bobObj.phrase); }
  };
// Works correctly
setTimeout(bobObj.sayPhrase, 100);
```

对于那些一次性对象来说，这种做法是没有问题的。但对象具有许多实例的情况更为常见，所以显式地引用对象是不可能的(如第 9 章中所讨论的那样，也可使用在对象创建期间实现的设计模式来避开这一问题)。

第二种解决该问题的做法是使用 jQuery 的 proxy 函数来永久性地绑定上下文：

```
bobObj = {
   phrase: "Yay!",
   sayPhrase: function() { alert(this.phrase); }
  };
// Create a proxied function
var proxiedFunc = $.proxy(bobObj.sayPhrase, bobObj);
// Will pop up an alert with "Yay!" after 100ms
setTimeout(proxiedFunc, 100);
```

从现在开始，任何时候调用 proxiedFunc，this 关键字都会被绑定到 bobObj 上。

Underscore.js(稍后讨论)也提供了一个实现了同样事情的便利方法。若使用 Underscore，setTimeout 可改写成：

```
var proxiedFunc = _.bind(bobObj.sayPhrase, bobObj)
setTimeout(proxiedFunc);
```

在 HTML5 游戏开发中，了解回调的这些微妙之处是很重要的，因为回调会被经常用到。至关重要的一点是，要确保自己知道如何传递函数，以及能够识别出任何给定情况下的 this 上下文。

5.3.5　绑定事件

回调的一种常见用途是绑定事件处理程序，从 1.7 版本开始，jQuery 使用$(selector).on 来提供一套附加事件的统一做法。因为要处理的事项很多，所以$(selector).on 具有一个用到了一些可选参数的非常复杂的定义，其中最常见的形式如下：

```
jQuery.on( events [, selector] [, data], handler )
```

这其中只有两个参数是必需的，那就是 events 和 handler。events 参数是一个由逗号分隔的事件名称组成的字符串，不过最常见的情况是只有一个名称。以下是一个最简单的例子，该例调用$(selector).on 来把一个单击(click)事件与一个 ID 为 start-button 的链接绑定起来：

```
$("a#start-button").on("click",function(event) {
  alert('Starting Game!');
});
```

可以注意到，标准选择器 $(" a#start-button ")后面紧跟着的是要绑定的事件 click，然后才是回调。

文档准备就绪这一特殊情况

你得留心才能确保代码的运行时机准确无误，若试图运行的 JavaScript 引用了尚未被加载的 DOM 元素，就有可能会给自己带来一些麻烦。通常情况下，你会希望等到整个页面都完成加载之后才运行 JavaScript，这一过程处在 window.onload 事件发生之前(在所有资产和图像完成加载之后，会触发该事件)。可使用标准的事件语法：$(document).on(" ready " ,function () { ... })来等待该事件，不过因为这一写法如此常见，所以 jQuery 提供了一种快捷写法，仅需把一个函数传递给 jQuery 运算符即可，即$(function() { ... })。

某些情况下，需要阻止事件处理程序采取默认的行为。在上述例子中，页面在默认情况下会转向你刚单击的链接所指向的 href 目标，但这可能是你所不希望发生的。这种情况下，需要告诉该事件你不希望默认行为发生，可以通过调用被当作参数传递进来的事件对象的 preventDefault 方法来实现这一点：

```
$("a#start-button").on("click",function(event) {
 event.preventDefault();
 alert('Starting Game!');
});
```

event.preventDefault 最常被一些具有默认行为的 HTML 元素使用，比如链接、输入元素和表单等，或被一些具有键盘事件的 HTML 元素使用，在这些事件中，你不希望页面的滚动受到影响。

若在这之后你希望去掉所有已被绑定到 start-button 上的单击事件，那么可以调用 $(selector).off，如下所示：

```
$("a#start-button").off("click");
```

这一调用去掉该按钮上的所有单击事件处理程序，若只希望去掉某个特定的处理程序，需要把该处理程序作为第二个参数传递进去。

这里继续讨论上一节提到的回调中的上下文，jQuery 有意把事件回调中的 this 对象改成触发了该事件的 DOM 元素，若需要访问回调外部的上下文，就需要对 jQuery 的这一行为有所准备才行。

例如，若希望在 Start 按钮(start-button)被单击后隐藏该按钮，你可这样写：

```
$("a#start-button").on("click",function(event) {
 event.preventDefault();
 alert('Starting Game!');
```

```
 $(this).hide();
});
```

需要使用$(this)把 this 对象封装在 jQuery 对象选择器中。

JavaScript 也拥有事件委托能力，这意味着可将元素和其子元素上的事件绑定在一起，这对于移动游戏开发非常有用，因为移动游戏往往拥有大量被频繁地添加到页面上和从页面上删除的元素，而与页面的交互则需要借助触摸事件进行。

把事件和这些元素中的每一个分别绑定在一起，这不仅耗费时间，而且速度会很慢。若改用事件委托这种做法，那么可以只绑定到容器元素上，但在需要时仍可接收所有事件。

为列举一个具体例子来说明这种做法的有用之处，创建了一个名为 Block Clicker 的简单游戏，该游戏的代码如代码清单 5-1 所示。游戏的目标是在时间用完之前，尽可能多地单击出现在页面上的方块，若玩家单击了 20 个方块中的 15 个，就赢得了游戏，否则就输掉了游戏。

代码清单 5-1：binding.html——一个简单的形状单击游戏

```
var width=$(window).width(), height=$(window).height(),
    countdown = 20, countup = 0;

var nextElement = function() {
  if(countdown == 0) {
    gameOver();
    return;
  }
  var x=Math.random()*(width - 50),
      y=Math.random()*(height - 50);
  $("<div>").css({
      position:'absolute',
      left: x,   top: y,
      width: 50, height: 50,
      backgroundColor: 'red'
    }).appendTo("#container");
  countdown--;
}

var gameOver = function() {
  // Stop additional nextElement calls from firing
  clearInterval(timer);
  if(countup > 15) {
    alert("You won!");
  } else {
    alert("You lost!");
  }
}
var timer = setInterval(nextElement,500);
$("#container").on('mousedown','div',function(e) {
    countup++;
```

```
        $(this).fadeOut();
    });
```

这段代码首先使用 jQuery 获取窗口的尺寸，然后定义两个变量来分别存放剩下要显示的方块数和用户已单击的方块数。

接下来，代码定义了 nextElement 函数，该函数每被调用一次，就有一个方块被添加到页面中。该函数首先通过查看 countdown 变量来检查游戏是否已结束，若是则调用 gameOver 方法；若游戏尚未结束，就生成随机的 x 和 y 位置，然后创建、样式化和在屏幕的相应位置显示一个新的 50×50 像素的红色方块。

GameOver 方法首先清除间隔定时器，这样就不会再有元素被添加到页面上，然后弹出一条提醒消息，告诉玩家他们是赢是输。

接下来是一个指向 setInverval 的调用，确保每隔 500 毫秒就有一个新的方块被创建。

最后，借助于$(selector).on 调用，游戏捕获了#container 元素内部的所有 的所有 mousedown 事件，即便对于尚未创建的那些元素来说也是如此。在玩家单击 时，countup 变量被递增，然后该元素淡出界面。在这个例子中，mousedown 事件用来替代 click 事件，这是因为 click 事件要求在被单击的同一个元素上释放鼠标，这会减慢游戏的速度。

注意：在 Block Clicker 游戏中，用户可通过连续快速单击正在淡出的元素来作弊，你能想出一种防止这种情况的办法吗？

5.3.6　发起 Ajax 调用

到目前为止，只有一个方面的 jQuery 功能被详细讨论到，那就是 DOM 操作，但该库所提供的有用技巧并不止这些。jQuery 还提供了一个简单一致的接口来向后台的 Web 服务发起 Ajax 调用，使用 Ajax 允许游戏在不必重新加载整个页面的情况下传递 Web 服务器端数据。在发起 Ajax 调用时，需要记住的一件事情是它们被定义成异步的，这意味着你不能确定调用将在何时完成。

5.3.7　调用远程服务器

说到发起 Ajax 调用，jQuery 提供了一个名为$.Ajax 的方法来规范全部的调用。若查看一下 http://api.jquery.com/jQuery.ajax/上的$.Ajax 文档，你会发现，该方法采用 30 多个不同的选项来配置将被发起的请求的各个部分。

在仅是希望发送或抓取一些数据时，这种级别的可配置性有可能太过于强大了。好在 jQuery 提供了几个快捷方法，例如，假设你希望加载一个存放关卡数据的 JSON 文件，那么可以简单这样写：

```
$.getJSON("level1.js",function(levelData) {
    // Do something with your levelData
});
```

其中的内幕是，jQuery 负责创建一个 XMLHttpRequest 对象(真正执行调用的浏览器对象)，注册 onreadystatechange 回调，并在解析返回数据和使用 levelData 调用回调之前先检查适当的返回状态。即使没弄清上述句子也别担心，它主要表明的含义是，直接处理 Ajax 调用的细节是一件相当复杂的事情，所以 jQuery 在这些调用的外围提供了一层非常好用的封装器，这意味着可以专注于自己的游戏而非传输机制。

$.getJSON 还可用来发起 JSONp 请求，这是一种绕过同一域这一限制的变通做法，该限制通常会妨碍到 Ajax 调用。要使用 JSONp，只需在请求的 URL 后面简单加上 callback=? 参数即可(前提是远程服务器支持 JSONp)。

$.getJSON 只是所提供的辅助方法之一，其他一些可用的方法如下所示，这里只给出了这些方法的常见形式：

- $.get(url,[data,], successCallback(data))：正如你可能已料到的那样，该方法发起一个 get 请求并返回数据，在加载 HTML 或其他一些你希望在放到页面上之前先进行处理的数据格式(如文本文件)时，可使用该方法。
- $(selector).load(url)：这是一个很便捷的方法，该方法把一个 Ajax get 请求的响应加载到任何与选择器匹配的元素中，可直接用它把服务器的内容(比如说前 10 列表或是制作人员表)快速加载到页面上。
- $.getScript(url)：发起一个 get 请求，但会把响应当成 JavaScript 处理。在希望服务器直接生成由客户端执行的 JavaScript 时，这种做法很有用。$.getScript 之所以有用，还因为它能够加载任何域中的数据，而其他任何除了 getJSON 之外的 Ajax 调用都要求针对同一个域发起请求。
- $.post(url, [data,], success(data))：使用可选的数据(data)向 URL 指向的地址发起一个 post 请求，post 请求的执行通常是为给服务器发送大量数据，也可用于提交表单数据。

要了解更多关于 jQuery 提供的所有这些 Ajax 方法的细节，请查阅完整的在线文档 http://api.jquery.com/category/ajax/。

5.3.8　使用 Deferred

Ajax 在异步方面的问题之一是，在试图一次做多件事情时，你可能无法确定哪一件事情会先完成，这其中所涉及的逻辑有些繁杂，需要用到许多状态变量或是一大堆嵌套的回调。幸运的是，从 jQuery 1.5 开始，所有 jQuery Ajax 方法都会返回一个称为 Deferred 的对象，若使用得当，该对象可以极大简化你的回调代码。

比如你希望加载三个独立的 JSON 数据文件，然后在完成加载后执行一些操作，有了 Deferred，你就可以这样写：

```
$.when($.getJSON('level1.json'),
       $.getJSON('enemies.json'),
       $.getJSON('player.json'))
   .then(function(level,enemies,player) {
```

```
    // We know all three files have loaded
}).fail(function() {
    // One or more files failed to load
});
```

你不必担心每个单独的调用是成功还是失败，或是它们的响应顺序如何。相反，你把它们封装起来，放在一个很友好的包中，在所有的这些远程调用都完成之后，就会收到一个回调(可以在 http://api.jquery.com/category/deferred-object/上找到关于 Deferred 的文档说明)。

5.4 使用 Underscore.js

尽管jQuery提供了许多可让编写JavaScript的工作变得更加轻松的实用方法，但jQuery把重点放在修改DOM和提供简化的Ajax支持上，它并未提供多少用于其他目的的实用方法(虽说如此，jQuery确也提供了一些这类方法，可参阅http://api.jquery.com/category/utilities/了解一些例子)。

幸运的是，有一个名为Underscore.js的库正是为此目的而创建的，Underscore是一个小型的库(精简和压缩后只有不到4KB)，提供了约60个方法，这些方法能把JavaScript变得更便于理解和更加简洁。

5.4.1 访问 Underscore

Underscore.js以单一JavaScript文件的方式被包含到项目中，这与jQuery类似，另一点与jQuery类似的是，作者决定不污染现有的JavaScript名称空间，而是把所有方法都封装起来，放到一个以下划线字符“_”来标识的函数的内部。

可采用两种方式来调用Underscore的方法，它们分别是函数式风格和面向对象风格，以下为每种风格列举一个例子：

```
_.isString(myVar);
_(myVar).isString();
```

函数式方法直接调用Underscore对象的函数，而OO方法先针对目标调用Underscore(与jQuery使用选择器的做法相似)，然后调用结果对象的方法。

5.4.2 使用集合

Underscore.js中的大多数方法都以使用集合为目标，这些集合是数组或对象。因为游戏开发要做的大部分工作是操纵诸如精灵一类的对象列表，所以这些方法用起来很方便。比如你有一个名为sprites的对象数组，你希望依次调用每个对象的update()方法，那么可以编写一个for循环：

```
for(var i=0,len=sprites.length;i<len;i++) {
  sprites[i].update();
}
```

另一种选择是使用 Underscore.js，那么写法就变成：

```
_(sprites).invoke('update');
```

后一种写法更简短，且代码的意图更清晰，唯一的缺点是调用 Underscore 方法而非仅编写自己的循环增加了一些开销。不过，在大部分情况下，这一开销是可以忽略不计的，且更少更简洁的代码这一好处值得你做出这样的取舍。

以下代码清单记录了一些最常用也是最有用的方法：

- _.each(list, callback, [context])：把列表(list)中的每个元素当成回调(callback)的参数依次调用回调，在支持 forEach 的浏览器中，该方法使用原生的 forEach 语句。需要注意的是，_.each 接收一个额外的 context 对象作为参数，该对象被绑定到回调内部的 this 上。
- _.map(list, callback, [context])：该方法与_.each 类似，除了接收回调(callback)的返回值，然后返回一个新数组这一点之外。
- _.find(list, callback, [context])：返回列表(list)中第一个让回调函数(callback)返回真值的项，就基于一些任意的条件函数查找列表中的元素这一目的而言，该方法很有用处。
- _.filter(list, callback, [context])：该方法与_.find 类似，除了返回的是让回调返回真值的所有项组成的数组这一点之外。
- _.without(array, [*values])：返回一个新的没有包含任何已被删除值(values)的实例的数组，这很有用，例如，可用于删除列表中的已死精灵。
- _.uniq(array, [isSorted], [iterator])：_.uniq 返回已删除所有重复元素的数组的一个副本，若数组刚好处于有序状态，可把 true 值传给 isSorted 以便提高性能。

总之，这些方法把对象列表的使用变成一件更简单明了的事情。

5.4.3　使用实用函数

此外，Underscore 还提供了许多通用的实用函数，这些函数能把编码工作变得更为容易。

- _.bindAll(object, [*methodNames])：修改任何到 object 对象的 methodNames 方法的调用，这样上下文就始终都是 object。例如，这意味着可以把方法传递给 jQuery 的事件处理函数，而又不必考虑 this 这一上下文。
- _.keys(object) / _.values(object)：返回某个对象的所有键和值。
- _.extend(destination, *sources)：把源对象列表(sources)中的所有属性复制到目标(destination)中，覆盖任何现有属性。
- _.is[ObjectType]：Underscore.js 提供了许多很好的.is[ObjectType]形式的方法来检查传入对象的类型，这非常有用，因为 JavaScript 不提供内置的类型检查方法。Underscore 提供的这一类方法有_.isEqual、_.isEmpty、_.isElement、_.isArray、_.isFunction、_.isString、_.isNumber、_.isBoolean、_.isNaN、_.isNull 和_.isUndefined

等，所有这些_.is[ObjectType]方法在某种程度上都是不言自明的：每个方法要做的事情都是检查被传进来的对象，然后返回 true 或 false 值。它们提供了一种检查 JavaScript 对象类型的简便做法，这对于检查参数和伪造多态性都很有用处，这里的伪造多态性是指依赖被传到方法中的参数来让方法执行不同的行为。

- _.uniqueId([prefix])：为客户端的 DOM 元素或模型生成全局性的唯一标识符，这很有用处，因为你通常会希望给添加到页面上的元素加上唯一的 ID 特性。

本书中经常用到这些实用方法，特别是_.extend。

5.4.4　链式调用 Underscore 方法

因为 Underscore 在集合方面具有如此不错的作为，所以要尽量简化在一行中处理多个 Underscore 方法调用的语法。

以下是这一问题的一个例子，若你希望从 enemy 类别的所有精灵中找出最大的 y 值，那么可以这样写：

```
_(_(_(sprites)
      .filter(function(s) { return s.category == "enemy"; }))
      .pluck('y'))
      .max();
```

若能弄清楚所有这些嵌套的_ (..)调用，那说明你运气还不错，因为这可不是什么特别容易读懂的内容。出于这一原因，Underscore 提供了一种称为链接(chaining)的机制。

```
_.chain(sprites)
      .filter(function(s) { return s.category == "enemy"; })
      .pluck('y')
      .max().value();
```

若希望在一行代码中把多个 Underscore 函数链接起来，可调用_.chain()，在完成链接后，再调用.value()。

5.5　小结

两个很有用的 JavaScript 库——jQuery 和 Underscore——能够帮助你编写出更为简洁的代码，这其中既不会有针对每种浏览器的检查，也不会有样板代码存在，这样的代码更便于理解和维护。

不使用库无疑也是一种选择(毕竟 Alien Invasion 的构建就未用到库)，但把 jQuery 和 Underscore 都加进来所付出的代价也还不到 40KB，这大约就是一幅很小的 JPG 图像的大小。所以，要好好利用他人的辛勤劳动所换来的成果，把跨浏览器的开发变成一件更为容易的事情。

第 6 章

成为一个良好的移动市民

本章提要

- 最大化游戏界面
- 利用 iOS 的功能
- 处理有限的带宽
- 使用应用缓存

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 6 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

6.1 引言

你能够克服在移动设备上进行开发所带来的诸多挑战，前提是要事先做一点规划，以及要拥有一点关于目标平台局限性方面的知识。事情之所以变得困难是因为，在没有进行充分的事先考虑或在没有考虑移动设备所带来的局限性的情况下，硬把桌面游戏塞到移动设备中。本章为你准备了一些关于移动设备独特性的内容，若要成功开发和发布移动设备上的 HTML5 游戏，这些独特之处是需要了解的。

6.2 响应设备的能力

在移动设备上开发游戏的主要挑战之一是，如何最大化小型设备的屏幕实际使用面

积，同时还要支持这些设备的各种屏幕分辨率和纵横比。与诸如苹果公司的 App Store、DS 或 PSP 一类的单一平台移动设备不同，HTML5 移动游戏不仅需要处理数量众多的不同设备(这些设备都有各自的分辨率和纵横比)，还需要处理可以横屏模式或竖屏模式玩游戏的这种可能性。

6.2.1　最大化实际使用面积

当玩家在诸如移动设备屏幕一类很小的屏幕上玩游戏时，你所能做的最好事情就是确保游戏已占据了可用的屏幕实际使用面积的整个高度和宽度。

添加一个启动步骤

若游戏应被嵌入到页面的其他内容中，那么你可能会希望给游戏添加一个“启动”步骤，这样就可以在开始加载游戏之前等待一个来自用户的动作。

不过，对桌面的考虑则有所不同，总之，与屏幕大小相比，你倒是会希望限制游戏界面的大小。例如，就目前这一代浏览器和硬件而言，在一个 24 英寸的桌面显示器上最大化界面很可能导致游戏速度过慢，且对于基于位图的游戏来说，这也会把图片变得太过像素化。另一种做法是大幅增加游戏在桌面上的可视区域，但这有可能带来其他一些问题。一般来说，你也希望，无论是在桌面上还是在移动设备上，游戏的运行都能获得相类似的效果。

大多数引擎拿出的解决方案是，当游戏在移动设备上运行时，把游戏界面最大化至某一尺寸，但当用户在桌面上玩游戏时，则把它放在一个固定大小的容器中运行。

理想情况下，你用来开发游戏的方法应是与确切的尺寸和纵横比无关的。这说起来容易做起来难，其中的决定性因素是游戏的类型。对于平台动作游戏或是角色扮演游戏(RPG)来说，你应以这样一种方式来构建游戏，即根据屏幕的尺寸和面积来变换屏幕上的可视关卡的数目。若你拥有的是一块任何时候都需要完整显示在屏幕上的固定游戏区域，那事情就更难办了，这种情况下，需要确保自己已做到在保证纵横比不变的同时最大化了可玩区域的大小。

使用固定的纵横比可能导致一些不太理想的情况出现，因为许多设备都有着非常不同的竖屏和横屏模式的纵横比。一般来说，若需要保证游戏区域有一个固定不变的纵横比，那么你只能优化一种视图——或竖屏或横屏——然后或要求用户旋转设备(在 HTML5 中，你无法锁定屏幕旋转)，或接受游戏要在一个比理想情况要小的方框中运行这一事实。

6.2.2　调整出合适的画布尺寸

在本节中，你将看到一些处理屏幕调整的代码，其中的每个例子都以一些 HTML 样板代码和一个居中放在页面上的 480x480 像素的<canvas>元素作为开始：

```
<!DOCTYPE HTML>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8"/>
  <title>Page Resize</title>
  <link rel="stylesheet" href="lib/base.css" type="text/css" />
  <script src='lib/jquery.min.js'></script>
</head>
<body>
  <div id='container'>
    <canvas id='game' width='480' height='480'></canvas>
  </div>
</body>
</html>
```

除了库 jQuery，这里列出的唯一外部资源是一个名为 base.css 的样式表，与前一章的情况相同，该 base.css 样式表只包含了 Meyer 重置和一两个游戏特定的样式，在重置的后面加上以下两个样式：

```
#container {
  padding-top:50px;
  margin:0 auto;
  width:480px;
}
canvas {
  background-color:green;
}
```

若在桌面浏览器上打开这一代码文件，你会看到一个很不错的绿色的<canvas>元素居中显示在页面上。

接下来，将以下代码添加到结束标签</body>之前：

```
<script>
// Wait for the document.ready callback
$(function() {
  var maxWidth = 480;
  var maxHeight = 440;
  var handleResize = function() {
    // Get the window width and height
    var w = window.innerWidth ||
            window.document.documentElement.clientWidth ||
            window.document.body.clientWidth;
    var h = window.innerHeight ||
            window.document.documentElement.clientHeight ||
            window.document.body.clientHeight;
    if(w < maxWidth) {
     $("#container").css('width','auto');
     $("#game").css({position: 'absolute',
                     top: 0, left: 0, zIndex: 10000 })
               .attr({width: w, height: Math.min(h,maxHeight) });
    }
  }
```

```
  handleResize();
});
</script>
```

这段在文档准备就绪时运行的代码将以窗口宽度为基础，或把画布放在页面中间，或把画布的位置修改成绝对位置，让它占据整个浏览器大小。你可通过运行本章代码中的resize.html 文件来理解这段代码。

令人遗憾的是，jQuery 并未提供一种适用于所有浏览器的解决方法，根据浏览器的不同，用来确定客户端窗口的无滚动条高度和宽度的代码是不同的。在这件事上，Internet Explorer(IE)再次成为制造问题的浏览器，所以，若你打算忽略 IE 9 之前的 IE 版本，那么高度和宽度的计算代码就可以缩减成如下内容：

```
// Get the window width and height
var w = window.innerWidth, h = window.innerHeight;
```

在基于画布的游戏中，这样写通常是没问题的；对于可能要支持较旧 IE 版本的基于 DOM 的游戏来说，你应该使用例子中的完整字符串，但因为这一例子使用的是<canvas>，所以本章余下例子都会使用以上较为简短的字符串。

6.3　处理浏览器的尺寸调整、滚动和缩放

仅因为玩家按某种界面大小在浏览器中启动游戏并不意味着游戏会一直以这种方式运行，用户可能调整桌面浏览器的大小，或可能旋转他们的移动设备以获得更好的视觉效果。大部分支持 HTML5 的移动设备也支持用户捏拉缩放页面，在开发移动游戏时，需要考虑所有这些行为。

6.3.1　处理尺寸调整

即使不打算调整游戏在桌面上运行时的界面尺寸，你也应该考虑调整游戏以适应移动设备的屏幕大小，因为玩家很可能旋转设备以获得更好的游戏视觉效果。

每次调整浏览器大小时，用来调整画布元素大小的代码就是类似代码清单 6-1 中所给出的代码。因为没有游戏被附到这一例子上，所以调用游戏代码以让游戏知道自己已被调整大小的代码(即// Game.resize (newDim);)已被注释掉。可通过运行本章代码中的resize.html 文件来理解这一例子。

代码清单 6-1：resize.html——一块自我调整大小的画布

```
<script>
// Wait for the document.ready callback
$(function() {
  var maxWidth = 480;
  var maxHeight = 440;
  var initialWidth = $("#game").attr('width');
```

```
  var initialHeight = $("#game").attr('height');
  var handleResize = function() {
    // Get the window width and height
    var w = window.innerWidth, h = window.innerHeight,
        newDim;
    if(w <= maxWidth) {
     newDim = { width: Math.min(w,maxWidth),
                height: Math.min(h,maxHeight) };
     $("#game").css({position:'absolute', left:0, top:0 });
     $("#container").css('width','auto');
    } else {
     newDim = { width: initialWidth,
                height: initialHeight };
     $("#game").css('position','relative');
     $("#container").css('width',maxWidth);
    }
    $("#game").attr(newDim)
    // Let the game know the page has resized.
    // Game.resize(newDim);
  }
  $(window).bind('resize',handleResize);
  handleResize();
});
</script>
```

更新后的代码现在用到一个 else 条件语句，目的是允许游戏在全占页面或居中显示这两种状态之间切换。为实现这一点，代码把画布的初始宽度和高度存储起来，若窗口宽度大于预定的最大宽度，则把该元素改回使用相对位置和初始大小。可以通过调整浏览器窗口的大小来试用这一功能。

对窗口的 resize 事件的绑定确保了在每次调整浏览器尺寸时，都会调用 handleResize ( ) 方法。

你还需要考虑尺寸调整的另一个方面，因为游戏已使用一个特定尺寸进行初始化，所以在游戏的过程中，它必定会调整自己的界面大小。至于能用一种多容易的做法来实现这一点，这取决于所创建的游戏，但这是一件你从一开始就需要考虑的事情，要尽早做出支持或不支持的决定。尺寸调整涉及哪些内容与游戏密切相关，平台动作游戏可能只需显示或多或少的周边区域，纸牌游戏则需要缩放整个视图。

6.3.2　防止滚动和缩放

为弥补有限的屏幕大小，在移动浏览器中浏览网页时会涉及大量的滚动，对于那些没有建立移动版本的网站来说，这通常还涉及捏拉缩放，但在正常游戏期间允许这两种行为中的任一种都会带来灾难性后果。

若在移动设备上加载上一节中的 resize.html 文件，然后在画布区域滑动手指，你会发现，整个页面的表现正符合你对一般网页可能有的猜测：页面在滚动；若双击绿色的画布区域，它会缩小。

防止这一现象出现的解决方法很简单：绑定 touchMove 事件并调用 event.preventDefault()。把以下代码添加到上一节的尺寸调整代码的末尾处(仍放在 jQuery document.ready 一段中)：

```
$(document).on("touchmove",function(event) {
  event.preventDefault();
});
```

现在，重载页面后再试一次，若你试图上下左右滚动页面，页面应该会在原位置上停止不动。若不想表现得那么贪心(比如说，像你曾负责创建的那些交互式广告那样到处污染侧边栏)，那么可以把这一处理程序仅限于用在画布对象上：

```
$("#game").on("touchmove",function(event) {
  event.preventDefault();
});
```

这段代码能让玩家操纵页面的其余部分，放大游戏，然后可以在不用担心滚动和缩放的情况下玩游戏。

游戏内的滚动和缩放

若是希望玩家滚动、捏拉或是缩放的话那该怎么办呢？大多数情况下，你应重新创建游戏内部的行为而非使用浏览器的内置行为。第 10 章将介绍的一些触摸事件提供了一种跟踪多点触控的机制，该机制能够让你跟踪诸如捏拉缩放一类更高层面的行为。

6.3.3　设置视口

防止缩放和滚动并无问题，但若在 iPhone 上加载该页面，你首先会注意到没有一种浏览器的大小调整方案是起作用的，绿色的画布一直停留在被缩小了的状态。

之所以这样是因为，除非显式地告诉浏览器要把页面设置成多大，否则它会以缩小了的状态开始显示，就如同你正在浏览的是一个普通网。若要修正这一问题，需要将一个特殊的 meta 标签添加到头段(head)以设置视口(页面的可视区域)。在游戏中，最常见的视口设置做法是把视口设成 100%的设备分辨率，并且完全不允许用户进行缩放。将以下 meta 标签添加到文档的头段中：

```
<meta name="viewport" content="width=device-width, initial-scale=1.0,
maximum-scale=1.0, user-scalable=no">
```

在移动设备上重新加载页面，你应会看到绿色的画布被放大至占满了整个页面。尽管视口一开始是 iOS 仅有的功能，但现在，在 Android 和 Mozilla 的 Fennec 移动浏览器上都获得了支持。

谎言、该死的谎言和像素

在 iPhone4 之前的旧时好时光中，像素就是像素，这是 Web 开发者绝不会怀疑的事情。但随着 iPhone4 的 Retina 显示屏的引入，一夜之间这突然变得不再正确了。这是因为，成千上万的网站都是精确按着 iPhone 竖屏模式的 320 像素来开发的，若是它们被缩小至正常

大小的四分之一，那么看起来就会显得很可笑，所以，苹果公司引入了 CSS 像素的概念。iPhone4 及后来者仍假装拥有的是 320x480 的分辨率，但实际上它们的分辨率是该分辨率的两倍。若要利用这一分辨率，你可能需要攻克一些难关，第 15 章在深入探讨画布时会谈到这些难关。就目前而言，大多数设备要解决的麻烦事是以某种速度来填充像素，好让游戏开发者开心地接受 320x480，所以，这不是需要立刻用上的东西。

6.3.4　去除地址栏

可以使用另一个技巧来获取更多一点的页面实际使用面积，那就是去除 iOS 设备上的地址栏，可使用在页面完成加载之后稍稍滚动页面的招术来实现这一点：

```
window.scrollTo(0,1)
```

只有在页面内容长于一整页时，这一招才奏效；更糟的是，地址栏的移出还会影响所获取的页面 innerHeight。

这会带来一点小问题，因为你希望把画布的大小调整成占据整个页面，但在完成这一占满页面的调整之后，页面的大小有可能会发生改变。要解决这一问题，可以简单地把容器元素的高度设置成一个比没有地址栏情况下的最终高度还要大的已知值，然后滚动窗口来重新计算 innerHeight。修改 handleResize 方法开始部分的内容，修改后的内容如下所示：

```
var touchDevice = !!('ontouchstart' in document);

 var handleResize = function() {
   var w = window.innerWidth, h = window.innerHeight, newDim;

   // Make sure the content is bigger than the page.
   if(w <= maxWidth && touchDevice) {
     $("#container").css({height: h * 2});
   }
   window.scrollTo(0,1);

   // Get the height again, scrollTo may have changed the innerHeight
   h = window.innerHeight;
```

只应在支持触摸事件的浏览器中使用这一调整容器大小的招术，因为在其他浏览器(比如说一般的桌面浏览器)中，这就是一次正常的滚动，加入一大堆多余的空格只会带出害得页面到处滚动的不必要的滚动条。出于这一原因，touchDevice 是设成 true 值还是 false 值完全取决于 document 对象是否支持 ontouchstart 事件。

关于尺寸调整，最后还有一个更微妙的地方需要处理。在设备从垂直方向转向水平方向(竖屏转至横屏)时，iOS 在目前并不会触发一个尺寸调整事件，需要绑定的是一个不同的事件：orientationchange。可以使用之前的 touchDevice 检查来决定监听哪一个事件，使用以下代码来替换掉绑定到 handleResizes 上的上述尺寸调整(resize)事件：

```
var resizeEvent = touchDevice ? 'orientationchange' : 'resize';
```

```
$(window).on(resizeEvent,handleResize);
```

现在，这段代码会决定是监听 resize 事件还是 orientationchange 事件。

代码清单 6-2 给出了完整的代码以供参考，其中包含了之前介绍的所有移动设备特定的调整。

代码清单 6-2：addressbar.html——一块调整大小以适应移动设备的画布

```
<script>
// Wait for document ready callback
$(function() {
  var maxWidth = 480;
  var maxHeight = 480;
var initialWidth = $("#game").attr('width');
  var initialHeight = $("#game").attr('height');
  var touchDevice = 'ontouchstart' in document;
  var handleResize = function() {
    var w = window.innerWidth, h = window.innerHeight, newDim;
    // Make sure the content is bigger than the page.
    if(w <= maxWidth && touchDevice) {
      $("#container").css({height: h * 2});
    }
    window.scrollTo(0,1);
    // Get the height again, scrollTo may have changed the innerHeight
    h = window.innerHeight;
    if(w <= maxWidth) {
     newDim = { width: Math.min(w,maxWidth),
                height: Math.min(h,maxHeight) };
     $("#game").css({position:'absolute', left:0, top:0 });
     $("#container").css("width","auto");
    } else {
     newDim = { width: initialWidth,
                height: initialHeight };
     $("#game").css('position','relative');
     $("#container").css('width', maxWidth);
    }
    $("#game").attr(newDim)
    // Let the game know the page has resized.
    // Game.resize(newDim);
};
  var resizeEvent = touchDevice ? 'orientationchange' : 'resize';
  $(window).on(resizeEvent,handleResize);

  $(document).on("touchmove",function(event) {
    event.preventDefault();
  });
  handleResize();
});
</script>
```

以这种显露方式把这段代码放在这里响应文档准备就绪事件，这不会是一种长期可行的解决方案，上述代码将会被纳入第 9 章构建的移动引擎 Quitnus 中。

6.4　配置 iOS 主屏幕应用

此外，你还需要多添加几段代码到游戏中，目的是让大家把游戏保存到主屏幕上。首先要添加的是一个指明游戏是“web-app-capable(Web 应用可行的)”的 meta 标签，无论应用是否把自身标记成是 web-app-capable 的，用户都可以把游戏保存至主屏幕，但若如此显式设置该标记，浏览器就会自动以全屏模式加载应用，页面不会显示地址栏或底部的按钮栏。

6.4.1　把游戏变成 Web 应用可行的

若要把游戏变成 Web 应用可行的，需要把以下 meta 标签添加到文档的<head>中：

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

第二个要添加的 meta 标签可让应用在启动时看起来更像是一个普通应用，Mobile Safari 默认使用浅灰色的状态栏，不过可以通过添加另一个 meta 标签把它换成黑色的状态栏：

```
<meta name="apple-mobile-web-app-status-bar-style" content="black" />
```

该 meta 标签的 content 值被设置成 default 选项意味着保留灰色的状态栏；上面说到的“black”选项可把它变成标准的黑色状态栏，应用在界面上显示状态栏时，用到的就是这种状态栏；black-translucent 选项则会把内容向上推至页面顶端，但会让状态栏半透明地覆盖在内容之上。对于大多数情况来说，black 是最好的选项，除非你希望让玩家在整个 480x320 像素区域中玩游戏，这种情况下，可使用 black-translucent 选项。

6.4.2　添加启动画面

当应用被放在主屏幕上时，苹果公司额外提供了一种可选做法来提升启动体验，那就是在设备开始运行应用时显示启动画面。可将一个<link>标签添加到<head>段中指定这一启动画面，首先，创建一个竖屏方向的 320x460 像素图像，然后添加一个 link 标签来链接该图像：

```
<link rel="apple-touch-startup-image" href="/path/to/320x460-startup-image.png">
```

若希望支持更多的而不仅是低分辨率的 iPhone 版本，那么可以为 iPhone 和 iPhone4+的 Retina 显示屏及 iPad 添加一整套竖屏和横屏模式的启动画面，做法是把适当的媒介(media)查询添加到每个 link 标签中：

```
<!-- 320x460 for iPhone before iPhone 4 and iPod Touch -->
<link rel="apple-touch-startup-image" media="(max-device-width: 480px) and
```

```
 not (-webkit-min-device-pixel-ratio: 2)" href="/path/to/320x460-startup-
image.png" />

<!-- 640x920 for Retina display on iPhone 4 and above-->
<link rel="apple-touch-startup-image" media="(max-device-width: 480px) and
 (-webkit-min-device-pixel-ratio: 2)" href="/path/to/640x920-startup-
image.png" />

<!-- 768x1004 for iPad in Portrait mode -->
<link rel="apple-touch-startup-image" media="(max-device-width: 1024px) and
 (orientation: portrait)" href="/path/to/768x1004-startup-image.png" />

<!-- 1024x748 for iPad in Landscape mode. Image should be rotated 90
degrees clockwise. -->
<link rel="apple-touch-startup-image" media="(max-device-width: 1024px) and
(orientation: landscape)" href="/path/to/1024x748-startup-image.png" />
```

警告：请确保自己使用的是如上注释中描述的具有确切大小和转向的图像。

若没有指定启动画面，默认情况下，用户看到的是游戏在上一次结束时的最后界面截图。

6.4.3　配置主屏幕图标

正如有人说过，细节决定成败。可加到游戏上让它看起来更像一个真正的本地应用的最后点睛之笔是，添加<link>标签来指定一个定制的主屏幕图标，简单地说，可以通过指定一个使用了“apple-touch-icon”关系的<link>标签来添加该图标，该关系指向一个 57x57 像素的 PNG 图像：

```
<link rel="apple-touch-icon" href="/path/to/57x57-icon-image.png" />
```

该图像不应该拥有任何标准的 iOS 图标装饰，而应该只是一个正方形的图标图像。若使用的是这一版本的 link 标签，那么 iOS 可以为图标补上圆角和高光效果。

不过，若使用这一版本，那么留给 Android 用户的就是一种欠佳的体验，因为 Android 不会为图像添加额外的增强效果。要解决这一问题，可使用一种已装饰了高光风格的“precomposed(预先构成的)”图标，做法是指定：

```
<link rel="apple-touch-icon-precomposed" href="/path/to/57x57-precomposedicon-
image.png" />
```

若还要顾及具有 Retina 显示屏的 iPad 和 iPhone，那添加一组三个显式设置了尺寸(size)的图标就能达到这一效果：

```
<!-- 72x72 for iPad -->
<link rel="apple-touch-icon-precomposed" sizes="72x72"
href="/path/to/72x72-icon-image.png" />

<!-- 114x114 for Retina Display on iPhone 4 and up -->
<link rel="apple-touch-icon-precomposed" sizes="114x114"
href="/path/to/114x114-icon-image.png" />

<!-- 57x57 for iPhone pre iPhone 4 and iPod Touch, Android 2.1+ -->
<link rel="apple-touch-icon-precomposed" sizes="57x57"
href="/path/to/57x57-icon-image.png" />
```

与 favicon.ico 文件类似，iOS 可在站点的根目录下查找其名称含有类似“apple-touch-icon.png”字样的文件，然后自动使用它们，不过你应该明确告诉设备哪些图标是可用的。

6.5　考虑移动设备的性能

桌面浏览器的发展已经到达了这样的一个阶段，即使用 HTML5 来构建任何类型的简单游戏都已是可实现的目标，你不需要费什么劲就能让一个 2D 平台动作游戏或是纵向卷轴的射击类游戏流畅地运行。

在移动设备上，性能却是另一回事了。若希望给用户带来一种流畅的体验，那么从一开始就需要考虑性能。为了对移动设备的性能局限性有一个体会，现在来了解一下 MarBook pro、iPhone 和 iPad 之间的各种简单渲染测试的比较(参见图 6-1)。

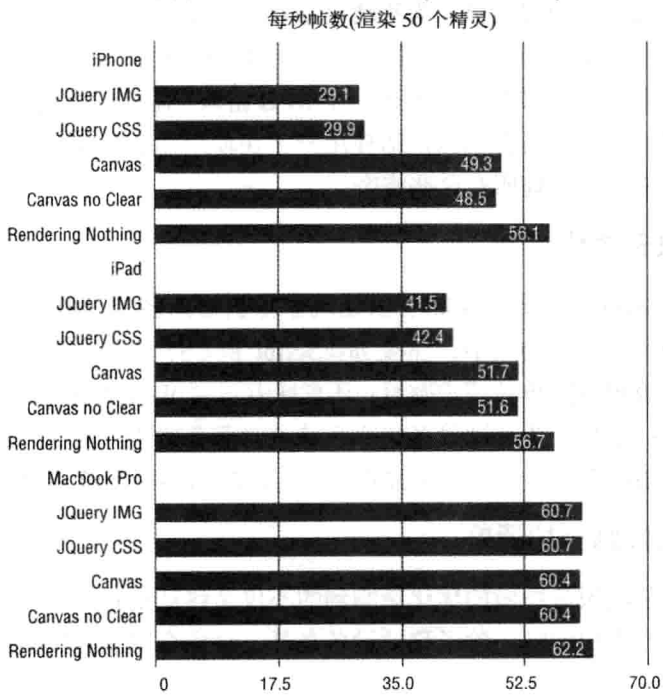


图 6-1　移动设备上的 HTML5 渲染方法的比较

可通过访问 http://cykod.github.com/mobile-html5-tests 来运行这些测试，这些测试的功能包括了创建各种数目的图像精灵，以及在一块 320x320 的面板上垂直向下移动它们。图 6-1 给出了渲染 50 个精灵的结果，测试运行了五种渲染做法，第一种使用基于<img>标签的精灵，第二种使用 CSS 的基于 background-image(背景图像)的精灵，第三种使用 CSS 精灵但用了-webkit-transform 而非只是设置左上角，第四和第五种方法都使用了画布，但第五种没有清除整块画布。

在桌面上，如何渲染精灵或是渲染多少精灵都无关紧要，帧率(frame rate)维持在每秒 60 帧左右。在移动设备上，情况则完全不同。在使用 IMG 或 CSS 精灵时，iPad2 和 iPhone4 上的帧率立刻有所下降，而 iPad2，得益于其更强一些的处理器，在使用画布精灵时，能做到稍更长时间地保持帧率不变，但在渲染 100 个基于图像的移动精灵时，这两种设备上的每秒帧数都明显下降。

虽然 100 个精灵可能看起来数目很大，但若考虑到炮弹、粒子效果、背景图案和其他一些动画，那么对于一个简单游戏来说，这并非一个不合理的数目。就目前而言，这些测试结果意味着在对游戏做出一些决定时，需要对移动平台有所了解，这算是个坏消息。好消息是，移动设备的性能一直在提升，且对于部署到移动设备上来说，除了使用 Mobile Safari 之外，还有其他一些选择。

6.6　适应有限的带宽和存储

鉴于 WiFi 在美国的使用优势，为移动设备设计游戏并不一定意味着玩家在玩游戏时使用的是 3G 网络，但这确已成了一个你需加以考虑的可能性。

若使用成百上千 MB 字节的资产创建一个 RPG 游戏，需要一个递增式的加载系统，这样的系统不会试图一次下载所有东西，即使用户正在使用 4G 或 WiFi，移动浏览器也不会有缓存空间或内存来高效处理所有这些资产。

6.6.1　为移动设备优化

为移动设备优化意味什么？意味着以让玩家尽快进入游戏这样一种方式包装和交付游戏；意味着限制所使用库的大小、缩减 JavaScript 和 CSS 资产，以及使用精灵表来限制设备必须发起的下载图像的单独请求数目；还意味着设置 Web 服务器，提供已经压缩的资产以减少带宽成本；最后，为移动设备优化意味着对服务器进行配置，让它提供正确的缓存头来确保资产已被缓存在设备上，无需在玩家每次玩游戏时下载这些资产。

6.6.2　移动设备好则一切皆好

好消息是，为移动设备所做的优化影响到的不仅是移动设备，把游戏的加载速度变得更快，让游戏的运行变得顺畅，除了移动设备之外，这也有可能会给桌面带来一种全方位的更佳体验。

许多时候，鉴于目前桌面浏览器的速度，遵循最佳做法好像是不太值得的事情，因为

优势看起来一直在上升。实际上，这种态度比乍看起来更能给游戏带来危害，作为游戏和 Web 开发者，你可能有机会使用较新的、高速连接到互联网的机器，但全国和全世界各地的潜在游戏用户则不太可能都这么幸运。

一些玩家可能使用拨号服务(这种方式支持的速度要比 3G 慢许多)，他们还指望着在老早就该退休的电脑上玩游戏呢。因为 Web 属于休闲游戏领域，所以你必须顾及更广泛的硬件平台，使用移动设备作为基线是一个良好开端，这可确保你进行了必要的优化。

6.6.3　缩减 JavaScript

HTML5 游戏的大小主要取决于 JavaScript 和图像文件，已写好的 JavaScript 文件应包含大量有用的注释，同时还有良好的缩进、描述性的变量和函数名，以及较宽的间距等。

仅是因为以这种方式来编写文件，并不意味着需要把以这种方式编写的文件提供给用户。现有的一些压缩工具可用来读入你的 JavaScript，剥离其中的注释和空格，转换各部分内容，大大降低最终文件的大小。例如 jQuery 1.7.1，原来大小几乎已达 250K，但在缩减之后大约只剩 93K。

若要帮助缩减代码，可以做两件事情，首先把代码包装在一个匿名函数中，以此来把它置于全局范围之外。通常你会看到如下写法的代码：

```
(function(window) {
  // Bunch of stuff defined.
  window.exportedFunction = function() { .. }
})(window);
```

这里，所有代码都被包装起来放到一个匿名函数中，然后在被传进来的 window 类中显式设置其所有要导出的内容，这种做法明确了哪些变量是全局的。关于这一模式有多种变体，但关键是全局名称空间中不会充斥着大量无用的东西。

第二件可做到缩减代码的事情是，确保使用 var 关键字来创建不需要在包装函数外部使用的局部变量，以这种方式使用局部变量允许缩减工具自动把这些变量的名称改成诸如 a 和 b 一类的短名，这会带来更好的文件压缩效果。

可用的 JavaScript 缩减工具不少，其中最受欢迎的三个分别是 Yahoo 的 YUI 压缩器(http://developer.yahoo.com/yui/compressor/)、Google 的 Closure 编译器(http://code.google.com/closure/compiler/)和 Uglify.js(https://github.com/mishoo/UglifyJS)。本书使用 Uglify.js，因为它很受欢迎，且是用 JavaScript 编写的。

要运行 Uglify.js，需要用到某种类型的 JavaScript 命令行环境。第 9 章会讨论 Node.js 这一用于运行 JavaScript 的命令行和服务器端框架的搭建和运行，在此之前，可以通过把一些代码传递给 Uglify.js 的在线版本 http://marijnhaverbeke.nl/uglifyjs 来获得并使用 Uglify.js 的输出。

6.6.4　设置正确的头域内容

你可能会做的最糟糕的事情之一是，在玩家再次访问游戏时，强制他们重新下载那些

并未改动的大型资产文件。把过期时间这一头域内容设置成将来的某个时间点，这意味着告知浏览器，它可以缓存任何自己喜欢的资产。

允许浏览器缓存哪些文件才有意义呢？是几乎所有的资产文件，这其中包括图像、音效和关卡数据等，若不是在服务器端动态生成的话，那么它们都是应该可缓存的东西。

简单来说，若使用 Apache，那么可添加一个指令，把过期时间设成将来的某个时间点：

```
<Directory /path/to/asset/files>
  ExpiresDefault "access plus 10 years"
</Directory>
```

这一指令要求你预先安装了 Apache 模块 mod_expires。

不过，若对游戏做了改动，需要确保能够更新游戏资产。一种确保在缓存打开时始终能够提供最新资产的做法是，把文件的秒格式的最后修改时间(亦称为 mtime)追加在 URL 的后面，如下所示：

```
<script src='js/game.js?1326075236'></script>
```

若 HTML 文件以动态方式提供，那么这是很容易办到的，若经由静态文件提供，那么可使用构建脚本来硬编码一个修改日期。

就动态加载的资产而言，你可使用一个类似的方法来实现这件事情。一般情况下，对小游戏来说，把一个自动获得的全局版本号附加在任何被加载资产的后面是最简单的做法。

最后，确保自己在开发期间关闭了缓存。只因浏览器提供的是代码的旧版本，结果害得自己穷追不舍一个已经改正的错误，再没有什么比这让人更窝火的了。

确保资产提供了正确的缓存头，到此事情才解决了一半。另一半是要确保以压缩(gzip)方式把受益于压缩的资产提供给支持压缩的浏览器(就此时而言，是指每一种应考虑在其中运行 HTML5 游戏的浏览器，甚至包括古老的 IE6 在内)。

若使用的是 Apache，那么这个过程由 mod_deflate 模块来负责。请确保 mod_deflate 已被启用，然后把以下代码添加到虚拟主机或.htaccess 文件中：

```
AddOutputFilterByType DEFLATE application/javascript application/
x-javascript text/html text/plain text/xml text/css
```

现在，可在网络上提供已经压缩的 JavaScript、HTML、CSS、纯文本和 XML 文件了。

6.6.5　经由 CDN 提供

要让资产真正做到快速下载，实际上，最好放弃亲自提供资产的做法，直接通过内容交付网络(content delivery network)，也即 CDN 来提供它们，CDN 旨在通过遍布全国和全球的边缘节点位置来快速提供文件。就近提供文件意味着请求将获得更快的响应(需要较短的来回传递包的时间)，不过最大的优势体现在，一般来说，因为 CDN 已为快速提供文件进行了优化，所以它们拥有能实现这一目标的基础设施和互联网肥管。

Amazon.com 的 Cloudfront 是最受欢迎的 CDN 之一，比较容易上手而且要价便宜，除

非你累积的带宽有显著上升。Cloudfront 使用的是 Amazon.com 的 S3 云存储服务，需要配置 Cloudfront，从特定的 S3 桶抽取文件，不过自此之后，任何上传到 S3 的文件就都是可通过 Cloudfront 获得的。

要注册使用 S3 和 Cloudfront，需要先在站点 http://aws.amazon.com/上注册一个账户；接着，可以通过该站点启动一个 AWS 管理控制台，然后通过该控制台进到 S3 选项卡中(这时可能会提示你先注册服务)并单击 Create Bucket 按钮，接着输入一个唯一的桶名，桶名需是跨所有 S3 桶唯一的，所以起名时得要有点创意才行。

接下来，需要单击 Cloudfront 选项卡，创建一个新的分发(distribution)。单击 Create Distribution 按钮，选择你刚创建好的桶。可按下 Continue 按钮走完余下的几个界面，直至分发准备就绪。创建了分发后，它可能要花上五分钟的时间来进行设置，不过，可以在属性中看到分发的域名。任何上传到桶中的文件都可以在该域名下被访问到，且访问速度极快。

可使用管理控制台将一些文件手动复制到 S3 上，不过，你也可以使用一些工具和库来帮助自己完成这件事情，如 s3sync：https://github.com/ms4720/s3sync。

非图像和脚本资源(即诸如.json 关卡数据和 CSV 文件一类的数据资源)一般经由 Ajax 加载，Ajax 有一个同源策略，这一同源策略要求你加载资产所经由的域/子域、协议和端口要和加载主 HTML 脚本经由的域/子域、协议和端口相同。对大多数资产来说，这并非是不可通融的，因为图像、音频、视频和 JavaScript 文件都能顺利地加载，只是要记住有这样一个策略就是了。

6.7　借助应用缓存的完全离线运行

关于成为良好移动市民所应做的每件事情，本章都已谈及，但仍漏掉了一样内容，那也是网页应用的圣杯(Holy Grail)：允许用户在无任何互联网连接的情况下玩游戏。只要配置得当，那些已把游戏保存到主屏幕上的用户能够在搭乘地铁时启动游戏，尽情击落各种外星人。将这一功能添加到游戏的秘诀在于把游戏配置成已正确使用了应用缓存(Application Cache)的，应用缓存是定义在离线 Web 应用(Offline Web Applications)之下的一个 HTML 标准。

6.7.1　创建代码清单文件

在把应用变成离线可用的必需举措中，关键在于修改页面开头的<html>标签，把 HTML 页面和一个代码清单(Manifest)文件关联起来，做法如下：

```
<html lang="en" manifest="/manifest.appcache">
  .. Rest of your HTML ..
</html>
```

实际的代码清单文件名称由你来决定，不过，已经商定的文件后缀名是.appcache，且

该文件需要作为 text/cache-manifest 这种 MIME 类型(mime-type)提供，Apache 一般不会预先配置这一 MIME 类型。可将以下代码添加到 Apache 配置中或.htaccess 文件中，确保在提供该文件时指定了正确的 MIME 类型：

```
AddType text/cache-manifest .appcache
```

一般来说，你应该在头域中显式地覆盖 manifest.appcache 文件的过期时间，以防该文件被缓存。因为 Apache 已启用了 mod_expires，所以可在配置文件或.htaccess 文件中添加以下声明来实现这一点：

```
ExpiresByType text/cache-manifest "access plus 0 minutes"
```

接下来，需要编写真正的缓存代码清单文件，这实际上是一个相当简单的文本文档，该文档以大写的两个词 CACHE MANIFEST 作为开始，后面跟上至多三个不同部分的内容。

- CACHE：应被缓存的文件。
- NETWORK：这些资源只应在连线时才是可用的。
- FALLBACK：这些资源在设备离线时应有一个可用的后备版本。通过一个在线版本后跟一个离线版本来指定 FALLBACK 资源。
- CACHE 和 NETWORK 各含一个文件或路径列表(允许使用通配符)

若在设备连线时加载了一个已被缓存的页面，浏览器会发出获取代码清单文件的请求。若该代码清单文件已发生改变，则重新下载所有文件；若该代码清单文件尚未变动(或已被正常缓存在浏览器中)，则不会重新加载已缓存的文件。因为你可能会经常在无需修改 Manifest 文件中的资产列表的情况下更新一些资产，所以，最常见的修改 Manifest 文件的做法是在注释中放入一个版本号。

比如说，你有一个经由/myGame 目录中的 index.html 加载的游戏，该游戏有一些静态资产被放在/myGame/images 和/myGame/js 目录中。然后，假设它有一个加载自/myGame/high-scores.php 的高分榜和一个加载自/myGame/ads.php 的广告，那么，可以设置一个如下的缓存代码清单文件：

```
CACHE MANIFEST
# Version: 1
# Remember to update the version whenever you change a file

CACHE
# Cache the game index.html file and all assets
/myGame/index.html
/myGame/js/*
/myGame/images/*

NETWORK
# Always try to pull the high scores from the network
/myGame/high-scores.php

FALLBACK
```

```
# Fallback to a static ad if user is not connected
/myGame/ads.php /myGame/static-ad.html
```

注意：具有 manifest 声明的 HTML 文件默认会被自动缓存，不需要列在代码清单文件中，但显式说明会将其缓存也无妨。

在玩家玩过游戏之后，下一次等他们再调出该游戏时，游戏就会从应用缓存中提取所有资产。若玩家在线，那么按下/myGame/ads.php 下载的就是真实的广告(或是该文件中存放的任意内容)，但在离线时，按下该文件加载的是应用缓存中的/myGame/static-ad.html。若你按下放在/myGame/high-scores.php 文件中的高分榜，那么无论设备是否连线，浏览器都会尝试发出请求。

6.7.2　检查浏览器是否在线

你可能希望游戏根据是否在线来采取不同的做法，Mobile Safari 和 Android 浏览器可使用 navigator.onLine 标志来检查浏览器是否认为自己在线，做法如下：

```
if(navigator.onLine) {
 // do something when online
} else {
 // Fallback
}
```

不过，这仅是一点辅助措施，而且事情也远没那么简单。仅因为 navigator.onLine 返回真值并不意味着你真能通过网络访问数据，浏览器可能是接上了没有连通互联网的 WiFi，或连接质量很糟，实际上并不能够下载任何数据。原则上，即使设备给人一种在线的假象，你也始终要捕捉任何网络错误。

在桌面上，基于同样的原因，navigator.onLine 也已信誉破产，Chrome 开发者甚至把这一错误标记成“WONT FIX(不做修正)”。

6.7.3　监听更高级的行为

本节仅对应用缓存做了简单介绍，HTML5 规范为应用缓存定义了八个不同的事件，因此，一个能做到即时更新缓存的完整健壮的实现还需要做大量的工作来处理各种不同的情况。好消息是，大多数情况下，你所需要关心的就是缓存文件以备离线之用。

6.7.4　最后的警告

测试和调试应用缓存有可能是个痛苦的过程，特别是在你已如所建议的那样把缓存头打开之后。最简单的测试方法是关闭缓存头、确认浏览器通过访问服务器获取每个文件、加入代码清单文件、禁用 WiFi 或以太网，你很快就能够确定游戏在离线情况下是如何行事的。当游戏在桌面浏览器上的运行没有问题之后，试一下在处于飞行模式中的移动设备

上重新加载该游戏，看看你所做的努力是否生效。

作为再次警告之言——在开发期间，不要启用缓存头和应用缓存。若常需要耗费精力弄清楚浏览器使用的到底是新的还是缓存的资源，那么你可能很快就会崩溃。

6.8　小结

成为一个良好的移动市民还真费了不少功夫！好在很大一部分的繁重工作都是公式化的，在完成设置之后就不必理会了。这一说法适用于移动设备的 meta 标签、缓存头和应用缓存；但也有不适用的地方，那就是确保游戏可在目标移动设备的约束条件之内正常运行，在这方面，需要花一些心思，做一些规划，且很有可能要创建一些原型，之后才能确定自己能够在当前这一代硬件上走多远。

第Ⅲ部分

JavaScript游戏开发基础

- 第 7 章：了解 HTML5 游戏开发环境
- 第 8 章：在命令行上运行 JavaScript
- 第 9 章：自建 Quintus 引擎(1)
- 第 10 章：自建 Quintus 引擎(2)
- 第 11 章：自建 Quintus 引擎(3)

第 7 章

了解 HTML5 游戏开发环境

本章提要

- 选择开发环境
- 探讨 Chrome 开发者工具
- 调试 JavaScript
- 改进和优化游戏
- 在移动设备上调试

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 7 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

7.1 引言

过去十年中，在配上一个很好的文本编辑器后，浏览器已发展成为了一个出色的开发环境，可用于构建、调试和优化 Web 游戏。要寻找一种 HTML5 IDE，那么不必舍近求远，你每天用来在网上冲浪的浏览器就是你要寻找的目标。

虽然几乎所有的浏览器都有着很不错的调试环境，但本书会专门讨论 Chrome 开发者工具(Chrome Developer Tools)。Chrome 在所有平台(Windows、OS X 和 Linux)上都是可用的，它提供了一个最新的 WebKit 浏览器，该浏览器与大多数移动设备上的 WebKit 浏览器在许多方面保持了一致。

7.2 选择编辑器

在能够把代码组织起来放在浏览器中运行之前，需要先使用某种编辑器来编写这些代码。使用哪种文本编辑器或开发环境这完全取决于你自己，可以走 IDE 类路线，使用诸如 WebStorm、Aptana、Netbeans 甚至 Visual Studio 一类成熟的开发环境；又或，像许多开发者一样，仅用一个很好的文本编辑器就能满足自己的需求。在 PC 上，Notepad++是一个很受欢迎的选择；在 Mac 上，TextMate 或 MacVim(若你喜欢新鲜事物的话)都是不错的选择；在 Linux 上，Emacs 或 gVIM 都能用来完成编码所需的工作。至于 Dreamweaver，那就别去碰了，因为它的工作更多侧重于编写 HTML 而非 JavaScript，这对于编写代码来说，可谓弊大于利。

7.3 探讨 Chrome 开发者工具

Chrome 在所有平台上都是可用的，它有着首屈一指的开发工具。在用它来访问页面时，可以查看几乎所有与页面相关的信息，可以通过控制台执行任意 JavaScript。Safari 有着一套近乎相同的工具(它们共享相同的代码库)，然而，Safari 在 Linux 上是不可用的，且在开发者当中，它的受欢迎程度也没有 Chrome 那么高。

7.3.1 激活开发者工具

与 Firebug 不同，Chrome 开发者工具已预先与 Chrome 安装在一起了，只需打开就可以使用。要访问工具包，可以使用鼠标单击浏览器右上角的扳手形状的菜单，然后选择 Developer Tools。在 PC 或 Linux 上，还可按下 Ctrl+Shift+I 来打开它们，在 Mac 上，起作用的按键是 Command+Option+I。

7.3.2 审查元素

开发者工具中的第一个选项卡是 Element 选项卡(见图 7-1)，该选项卡可让你查看文档对象模型(Document Object Model，DOM)的当前状态，这与通过 View Source 菜单看到的 HTML 代码是不同的，因为该菜单显示的是从服务器加载的 HTML 代码，但 JavaScript 有可能已经修改了 DOM。在左边的窗格中，可以上下浏览 DOM，可以随意打开和关闭每个元素块，可通过双击属性来修改它们。还可通过右键单击(在 Mac 上是按下 Ctrl 键单击)做进一步修改，如编辑节点(Node)的 HTML 代码，或是添加或删除元素等。

要审查页面上的某个具体的 DOM 元素，还可右击(在 Mac 上是按下 Ctrl 键单击)页面上的任何 DOM 元素，然后选择 Inspect Element 菜单。完成这一操作之后，左边窗格会给出你所单击的元素在页面 HTML 代码中的位置，右边窗格则显示了该元素的所有属性，其中最突出的细节是 CSS 样式，自上而下，窗格逐一列出了从最具体化的到最通用的样式，并划掉那些已被更具体的样式覆盖了的样式。

图 7-1　审查元素

可以通过单击具体样式边上的多选框来启用或禁用相应的样式，还可以通过单击属性或值来修改现有的样式，或是通过单击样式下面的结束花括号来添加一个新的样式，而删除属性名也就意味着从该样式中完全删除该属性。

要查看应用了所有样式的最终结果，打开 Styles 选项卡上面的 Computed Style 选项卡。这个选项卡很有用，例如，在元素以百分比来设置大小时，若想弄清楚诸如元素的像素宽一类的基于像素的值，就可以使用它。

处在 Styles 选项卡下面的是 Metrics 选项卡(见图 7-2)，该选项卡给出了当前元素的盒模型(box model)表示。

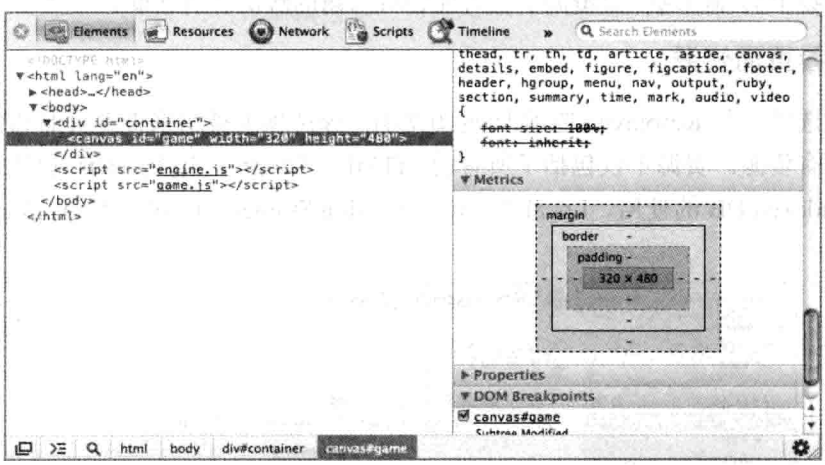


图 7-2　Metrics 选项卡中的元素盒模型

处在 Metrics 选项卡下面的是 Properties 选项卡(参见图 7-2 的底部)，若打开该选项卡，它会显示被选中元素的所有属性及对象继承的属性。可采用与修改 CSS 样式同样的做法来修改这些属性，即双击属性的值然后修改它，你还可以通过删除属性名来删除属性，不过一些必需的属性是不能被删除或修改的。

Properties 选项卡下面是 DOM Breakpoints 选项卡，若在跟踪被添加到页面或从页面中删除的元素时进展不顺，那么可以针对 DOM 的修改添加一些断点(见图 7-3)，这些断点会在适当时暂停 JavaScript 的执行。

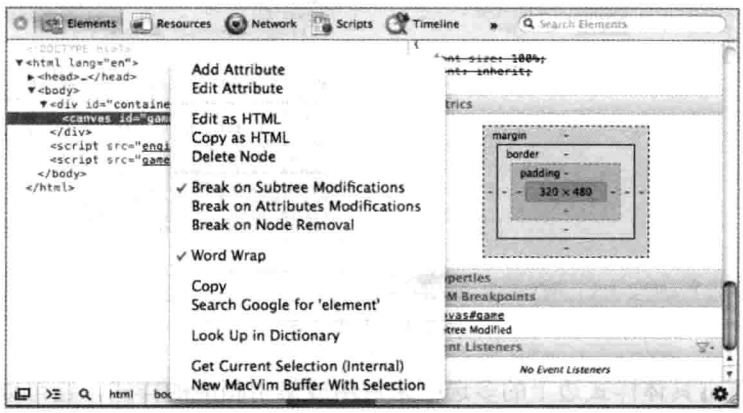


图 7-3　添加一个 DOM 断点

位于右侧的最后一个选项卡是 Event Listeners 选项卡，该选项卡给出了应用在该 DOM 元素上的事件。

在基于画布的游戏中，审查和修改元素及查看样式并不如在一般应用中那么重要，但若使用 CSS3 或 SVG 构建游戏，就需要了解应用在每个 DOM 元素上的具体样式及样式的层次结构。你可能会经常遇到这样一种情况，即无法把某个样式应用到某个对象上，这时候，了解是哪个具体的选择器覆盖了它可能会对问题的解决有所帮助。

7.3.3　查看页面资源

第二个选项卡是 Resources 选项卡(见图 7-4)，该选项卡被用来查看页面和任何内联框架用到的所有资源。资源不仅包括了所有的 HTML、Scripts、Stylesheets 和艺术资产，还包括诸如 Indexed DB 的使用、Local Storage、Session Storage、Cookie 和 Application Cache 等内容。

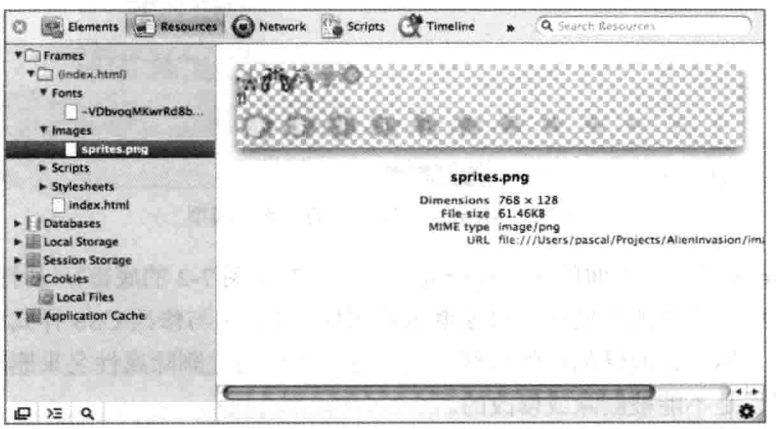


图 7-4　Resources 选项卡

这一选项卡极其有用，在需要查看游戏的本地存储或应用缓存的当前状态时尤其如此。可添加、修改或删除 Local Storage 中的键和值。与 Elements 选项卡类似，在 Resources 选项卡中的许多地方，你都可以添加、编辑和修改各种条目。编辑一般通过双击来触发，添加则是通过按下屏幕下方的加号或单击空行来触发。可以通过突出显示要删除的元素然后按下屏幕底部的 X 符号删除它们，或是右键单击(在 Mac 上是按下 Ctrl 键单击)要删除的元素然后选中 Delete 菜单删除它们。你不能使用开发者工具来修改页面的资产和应用缓存，但可以很容易地修改界面上的其他元素。

该选项卡最常见的用途是检查 Cookie、本地存储和应用缓存，因为这些元素可能较难调试。特别是查看应用缓存(见图 7-5)能够帮助你了解游戏是否如预期般进行了缓存。

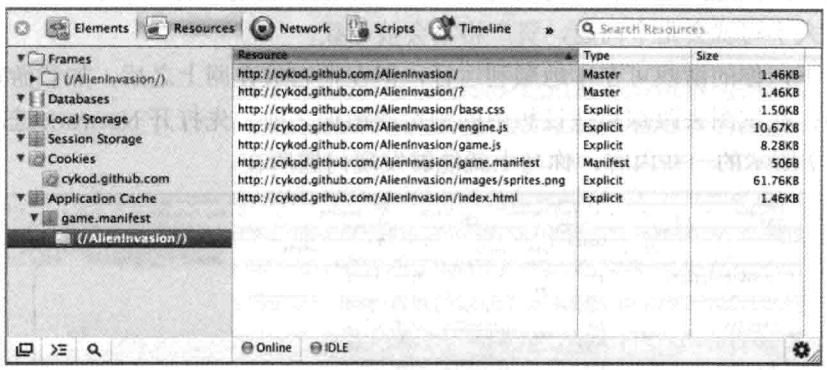


图 7-5　查看应用缓存

7.3.4　跟踪网络传输

接下来是 Netwok 选项卡(见图 7-6)，正如你很有可能已料到的那样，该选项卡跟踪游戏发出的所有网络请求，其中包括了下载 HTML、JavaScript 和各种资产，以及发起 Ajax 请求和建立 WebSocket 等。

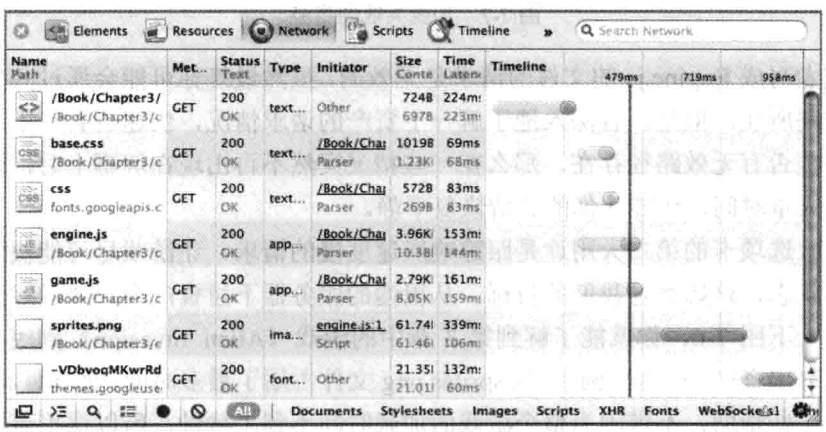


图 7-6　Network 选项卡

一般来说，游戏问题可以追溯到 JavaScript 错误或资源加载问题。资源加载问题，比

如说为 JavaScript 文件或资产指定了错误路径等，这类问题发生的频繁程度远比你想象的要高，且令人头疼不已，因为这不会是你第一时间想要尝试调试的内容。

快速查看一下 Network 选项卡能够帮助你解决许多资源加载问题，它给出了你想要加载的每项资源的实际响应状态，并突出显示被加载文件的任何问题。只有打开该选项卡后才能捕获资源的使用情况，所以，你可能需要重载页面来查看所发出的请求。

这其中一个常见的问题是大小写问题，在使用 file://这样的 URL 地址来在本地测试游戏时，文件名的大小写是没有影响的。若你有一个名为 engine.js 的文件，你试图在一个脚本文件中使用 Engine.js 这一名称加载它，那么浏览器很顺利就加载了该文件，好像没有什么错误发生。但在把游戏部署到网站后，除非 Web 服务器是一台安装了 Windows 的机器，否则的话，大小写就会是个问题，资产将不会被加载。

想象一下，你的游戏可在本地顺利运行，但在把它放在网上之后，加载游戏得到的却是空白界面。在试图去跟踪那些设想中的浏览器错误之前，先打开 Network 选项卡，若看到诸如图 7-7 所示的一些内容，你马上就能够发现问题所在。

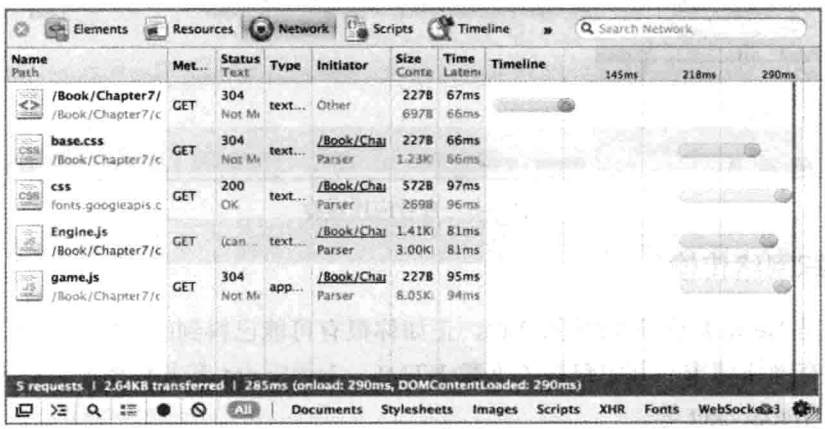


图 7-7　加载失败的资源

对误被大写成 Engine.js 的文件的请求是无效的，虽然最终你可能会通过其他一些方法找出这一问题所在，但是，若深入地了解一下资产的请求情况，快速查看一下 Network 选项卡，检查是否有无效路径存在，那么在一些精灵突然不再出现在屏幕上时，这一做法能够为你省下大量时间，还能让你的心情恢复平静。

Network 选项卡的第二大用途是跟踪响应速度慢的请求，让游戏尽可能快地完成加载以进入可玩状态，这始终是我们的目标，从很慢的服务器下拉资产会导致游戏速度显著放缓。若参考一下图 7-6，你就能了解到第 3 章中的游戏《Alien Invasion》到底花费了多长时间来加载每项资产。在这个例子中，sprites.png 文件占用了最多时间，把文件移放到 CDN 上或会带来一些帮助，不过因为整个游戏的加载时间不到 1 秒钟，所以这里可能不需要用到太多优化。在越是大型的、用到大量加载自不同服务器的资产的游戏中，你有越多优化工作需要做。

单击选项卡中的单项请求时，选项卡会显示该请求的所有详细信息和服务器的响应。

若代码通过 Ajax 调用来回应服务器，那么该选项卡所起的作用就更显得无可替代，因为可以跟踪参数和服务器的响应。截至撰写本书之时为止，开发者工具对 Websocket 的支持没有达到可查看被来回传送的数据的程度，所以，你得通过日志记下任何需要查看的数据。

7.4　调试 JavaScript

因为 HTML5 游戏非常倚重 JavaScript，所以在有出错之处或是出现意料之外的行为时，你通常希望审查正在运行的游戏。幸运的是，开发者工具提供了一个一流的调试环境，允许你查看对象、函数和值，以及在指定的地方暂停游戏，查看游戏的确切状态。

7.4.1　查看 Console 选项卡

在出现问题时，你首先应查看的地方是 Console 选项卡。该选项卡会就任何在游戏运行期间出现的 JavaScript 错误发出警告信息，错误用红色高亮标出，你会收到出现错误的文件的名称和错误所在行代码的行号(见图 7-8)。

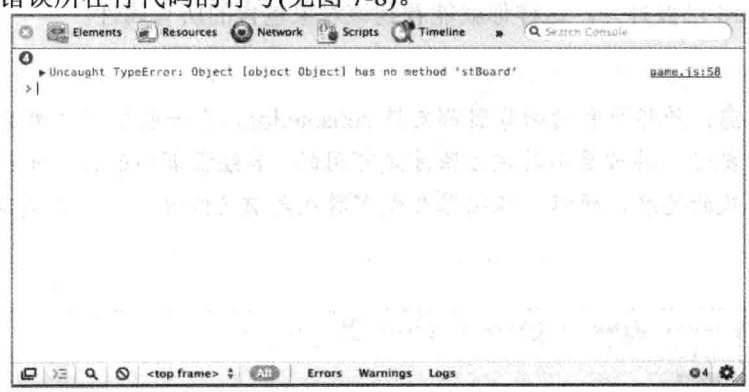


图 7-8　控制台中显示的 JavaScript 错误

单击右侧的文件名，可打开出现问题的文件，同时突出显示错行代码(见图 7-9)。可单击错误左边的小箭头来打开回调，该操作显示出到达当前行代码需要嵌套调用的所有函数。

图 7-9　出错代码行处显示的一个 JavaScript 错误

除了突出显示错误之外，还可以使用 console.log 方法把消息和数据记录下来，放到控制台中显示。若你记录了一个字符串，控制台会把它当作成字符串显示，同时显示调用 console.log 的那行代码的行号。若记录下的是比字符串更复杂的一些内容，那么控制台会显示整个对象，你可通过单击条目旁边的箭头来审查该对象。

例如，可将以下代码添加至第 3 章的 game.js 文件的 playGame 方法中：

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  board.add(new Level(level1,winGame));
  console.log("Logging board");
  console.log(board);
  Game.setBoard(3,board);
  Game.setBoard(5,new GamePoints(0));
};
```

启动游戏后，控制台中的内容看起来与图 7-10 类似。在图 7-10 中，通过单击箭头，第二个条目 board 已被打开，这样你就能看到该对象包含的所有属性。

注意：并非所有的浏览器都支持 console.log，在一些较旧的浏览器上，只有在开发者工具被启用时该方法才是可用的。若经常调用的话，该方法有可能降低游戏的速度，所以，请确保在发布游戏之前已经删除了所有的调用。

图 7-10　在控制台中显示对象日志

最后一点，也可能是最重要的一点，那就是在运行游戏时可以使用控制台来执行任意 JavaScript 并检查对象。

在按下回车键之后，任何被输入到控制台中的内容都会获得执行，这意味着若将一个 JavaScript 函数添加到游戏中，那么可以通过控制台来使用它。对于诸如打开开发者模式、

添加健康值以把游戏变得更便于测试，以及跳至游戏的任意地方等这类事情来说，这是非常有用的。以游戏 Alien Invasion 为例，可通过在控制台执行适当的方法(见图 7-11)来测试游戏的开始或显示“你输了”的界面，这通常是在被敌方飞船击中之后显示的画面。

图 7-11　在控制台中运行命令

任何被输入到控制台中的全局变量或对象都可变成可单击的对象，就如同你在游戏中某处针对它们调用了 console.log。所以，若需要查看某个对象的状态，只需在控制台中输入它、打开它，然后就可以单击想要查看的属性来了解正在发生的事情。

7.4.2　运用 Script 选项卡

若希望查看游戏加载的 JavaScript，或需要运行单步调试器来深入分析代码，那么此时就是打开 Script 选项卡的时候了。

默认情况下，Script 选项卡显示的是第一个包含 JavaScript 的文件，不过可以通过单击选项卡左上角的文件名称(见图 7-12)来打开任意包含了 JavaScript 的文件。

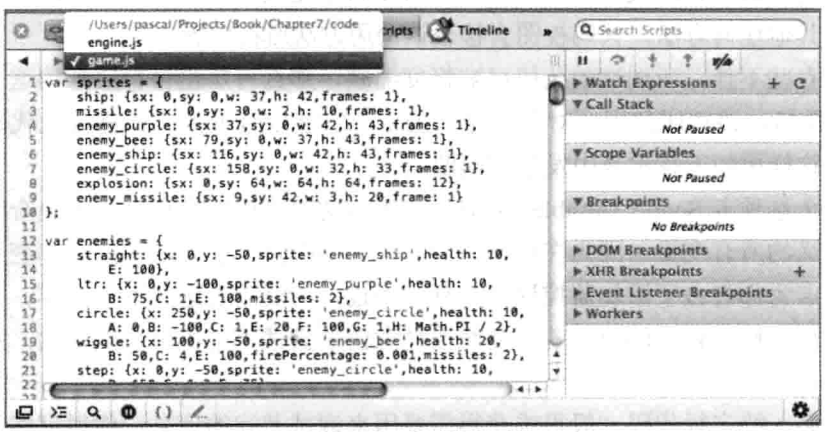


图 7-12　从 Script 选项卡选择不同文件

一般来说，在感兴趣的地方给代码加上断点之后，你就可以开始调试了。断点会告诉浏览器暂停游戏的执行，并将控制权转交给调试器。可通过单击脚本文件左边的编号列在

文件的适当行上添加断点，这会给该行打上一个标志，下次游戏运行到该行代码时，所有执行就会停止(见图 7-13)。

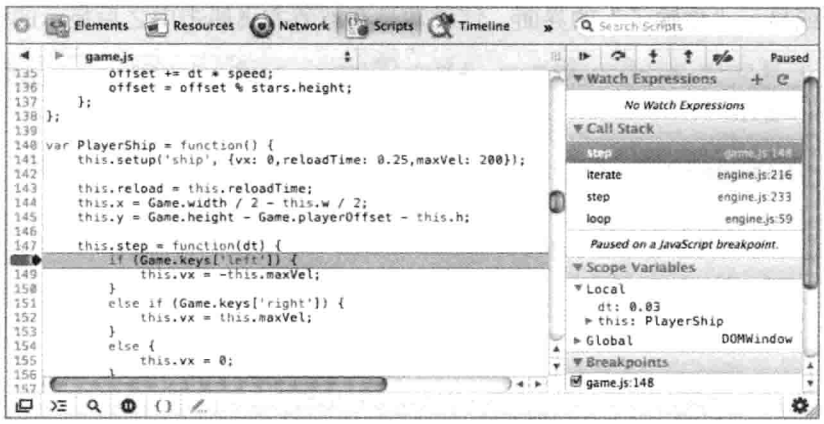


图 7-13　Script 选项卡中的断点

此时，右侧的选项卡中塞满了大量有用的信息，前面三个选项卡——Watch Expressions、Call Stack 和 Scope Variables——一般来说是最常用到的。

Watch Expressions 选项卡默认是空的，可在其中添加表达式，然后在浏览器当前停在的作用域中计算表达式。若某个具体变量的值对于搞清楚正在发生的事情很重要，那么可以把它添加到这个地方以便查看。还可以添加任意表达式，如计算等，做法是单击位于选项卡头部的加号小按钮，然后输入表达式。因为表达式是在当前上下文中求值的，所以可在表达式中使用局部变量和 this 对象。让鼠标悬停在某个监控上，按下减号就可以删除它。对于诸如对象和数组之类复杂的值而言，可单击小箭头来打开被监控元素的详细信息。

第二个选项卡 Call Stack 给出了你当前处在函数调用链中的位置。一般来说，你所调试的问题不与导致错误的代码有关，而往往与使用了无效参数来调用它的代码有关。单击调用栈中的其他任何方法，代码视图会移至该调用方法所在位置。

若运气有些不佳，所调试的代码已被极度缩减，那么可单击窗口底部的这个很有用的 Prettyprint 按钮，该按钮使用一对花括号来表示，它能以合理的方式格式化代码窗口中的所有代码，这样更便于找出其中发生的问题。

最后一个选项卡 Scope Variables 类似于一个自动的监控变量集合，该集合给出了当前作用域中定义的所有变量。因为在许多时候，你关心的是当前方法中的局部变量的值，所以这一选项卡为你省去了添加监控这一步骤。不过，与监控表达式不太一样，可通过单击变量的值来真正修改它们的值，若需要修改任意值来尝试不同值带来的影响，那么这一功能就很适用。

只查看某一特定行代码一般很难获得需要用来调试游戏的信息，你很可能需要以细微的控制步骤来走完程序，找出发生冲突的确切位置。要实现这一点，可使用位于选项卡右上方的一小排控制按钮(见图 7-14)。

图 7-14　脚本调试控制按钮

第一个按钮的作用是暂停脚本的执行，或在暂停之后恢复脚本的执行。若遇到设有断点的脚本行，执行就会暂停，单击该按钮可重启脚本的执行，直至遇到下一个断点。下一个按钮 Step Over Next Function Call 用来逐行执行代码，调试器不会再深入到栈的下一层，只在当前一行上执行完所有代码，然后步入下一行。接下来的两个按钮 Step into Next Function Call 和 Step out of Current Function Call 能让你更好地控制在代码中推进的方式。若希望在栈中进一步下移，那么按下前一个按钮，若希望一直执行代码直至当前方法返回，那么按下后一个按钮。最后，在找出发生的问题之后，可以按下最后一个按钮在启用和禁用所有断点之间做一个切换。在需要运行游戏片刻以达成某种条件时，关闭断点是很有用的；之后，就可以按下该按钮切换回启用断点状态，从而让代码准确地暂停在自己需要的地方。

7.5　分析和优化代码

HTML5 画布在桌面上的性能已到达这样的一个阶段，即在创建简单的 2D 游戏时，在大多数浏览器上你都不必过多关注性能问题(2010 年时的情况并非如此，当时画布的实现要慢很多)。移动设备却是另一番情景，任何你能做来优化游戏的事情都极可能会带来好处，比如说带来更流畅的游戏体验、让更广范围的设备拥有可接受的帧率等。

开发者工具配备了三种不同的工具来帮助你尽可能地提升游戏的性能，这里先从最重要的工具说起。

7.5.1　运行性能分析

对代码进行性能分析意味着跟踪游戏执行每个函数调用所用的时间，通过记录下每个函数调用的开始和结束时间，有多种方法可用来在代码中实现这一点。不过很幸运，开发者工具已经自带了一种非常简单的实现做法，你只需点按钮即可，不必修改代码。

要为游戏的执行进行性能分析，打开开发者工具中的 Profiles 选项卡，确保 Collect JavaScript CPU profile 一项已被选中，然后单击 Start 按钮。接着玩一会游戏，然后单击 Stop 按钮。也可以通过单击那个小小的 Record 按钮(灰色圆形)来实现这一点。

你所得的结果看上去与图 7-15 中的内容相似，结果详细列出了游戏中的每个方法调用、游戏花在每个方法上的时间(self 列)，以及整个方法占用的时间，这其中包括了对其他任何方法的调用(total 列)。看起来，似乎时间大部分都被花在了一个被称作“(program)”的神秘项上，实际上，此行代表了浏览器，以画布游戏为例，这通常是代表没有被游戏用到的额外处理器周期。不过，若开发的是 CSS3 或 SVG 游戏，那么实际上你可能不会看到这么多的空闲周期(可借助 Timelines 选项卡的功能来优化这一点)，因为浏览器可能需要做大量工作来处理动画和过渡。

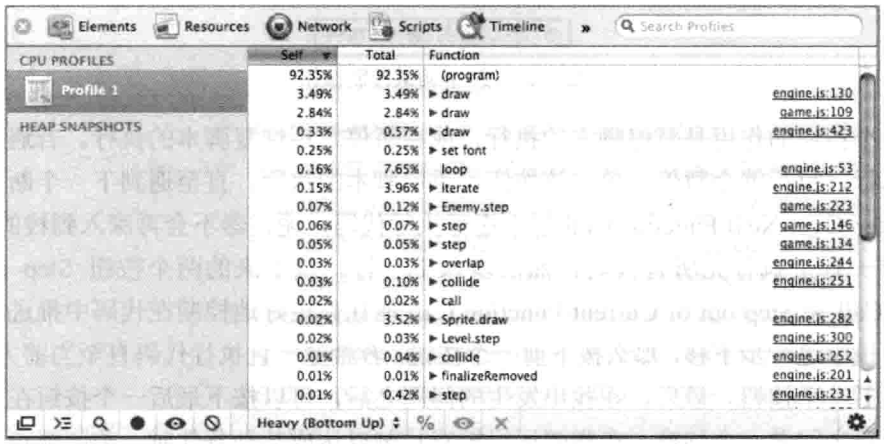


图 7-15　在 Chrome 浏览器中对游戏 Alien Invasion 所做的性能分析

图 7-15 给出了游戏 Alien Invasion 的细分列表，可以从中看到各种 draw 方法占用了大量时间。遗憾的是，这不是一个适合优化的地方，因为这些 draw 方法都很简单，仅是调用画布的 draw 方法而已。不过这意味着，找出一种减少 draw 调用次数的做法有可能就是一种优化游戏的做法。

一个你可能在最初认为会有很大机会改善性能的地方是碰撞方法，因为它是一个相当基础的方法，不过在花上数小时优化一个例程之前，还是先确保它真能让事情有所改变。用在游戏上的全部时间所占百分比是 100%～92.35%[花在“(program)”块中的时间所占百分比]，相当于 7.65%。用在碰撞上的全部时间所占百分比只有 0.10%，这意味着 collide 方法只占用了总游戏执行时间的 0.10%/7.65%，即 1.3%(0.013)。可通过减少 50%的执行次数来优化该方法(已是很大程度的优化)，不过这仅会带来 0.65%的执行速度的提升。这样的改进程度用户未必能留意到，也许并不值得花时间去优化它。

在进行优化时，需要找出哪些地方是值得花时间去优化的。在这个例子中，虽然 collide 方法使用了一个原生算法，很容易对它进行优化，但却可能不值得花费这个时间。在其他一些情况下，比如说在屏幕上有着许多不同对象在相互碰撞时，这会是一个更好的优化目标。

另一个因素是，虽然大多数的移动浏览器共享 WebKit 内核，但 Android 和 iOS 用到了不同的 JavaScript 引擎，这意味着 Chrome 开发者工具虽然能够很好地标识出，在 Android 上，哪些地方会存在性能问题，但对于 iOS 来说，它不一定同样是一个很好的指示器。这种情况下，你可能需要启动桌面上的 Safari，打开其中的开发者工具(该工具与 Chrome 的几乎是一样的，因为它们共享相同的代码库)和性能分析。图 7-16 显示的是在 Safari 中运行类似的性能分析基准得到的结果。

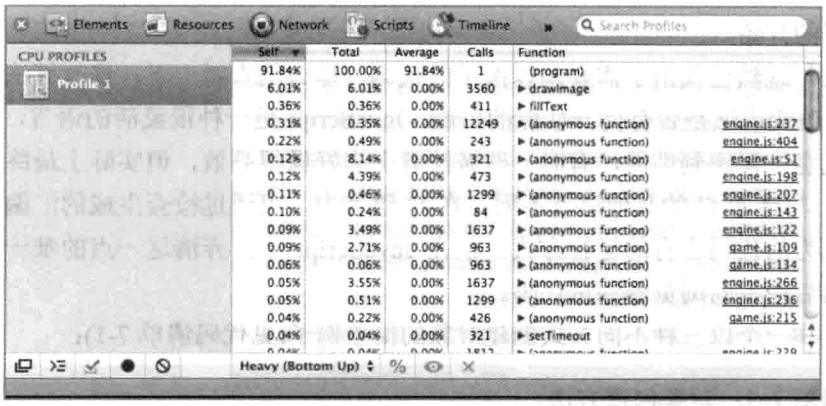


图 7-16　在 Safari 中对游戏 Alien Invasion 所做的一个性能分析

Safari 中用到的 JavsSript 引擎在处理匿名函数方面比不上 Chrome 的 V8 JavaScript 引擎，但它确实带来了一个额外的好处，那就是它能够让你深入内部，获得原生方法调用的时间花费情况。因此，若代码在 iOS 上运行不畅，需要加以优化，那么，即使不需要名称以备之后引用，也应为匿名方法加上名称(function collideCallback ( ) { .. }而非 function() { .. })。

在 Chrome 中，Profiles 选项卡还有另外两个性能分析快照：CSS 选择器性能分析和 Heap 性能分析。在 HTML5 游戏开发中，前者可能不会太有用，因为大多数的查找一般通过 ID 进行，不过堆快照也许能帮得上忙。若遇到内存问题，你最可能遇上的是内存泄漏，导致这一问题的原因是未删除所有指向已不再用的对象的引用，那么堆快照的获取能够帮助你了解某时刻 DOM 和 JavaScript 所占用内存的大小。

启动游戏并小玩一段时间，然后获取一个堆快照，你应就能从中检测出任何异常。例如，若你玩了一段时间的游戏，获取了一个快照，然后通过单击井号(#)按照降序排序对象的数目，结果发现游戏中有 2000 个精灵对象在到处游逛，那么一种很有可能的情况是，在精灵死去之后你并未正确删除它们。此外，你还应留意大小激增超预期的对象，因为这在很大程度上意味着它们所保持的数据超出了预期。

另外的两个选项卡 Timelines 和 Auditing 可能会给调试网页性能问题带来一些帮助，但在游戏优化方面的用处不大，所以这里就不予讨论了。

7.5.2　真正进行游戏优化

有了这些任你支配的工具，下一个最有可能出现在你脑海中的问题是：“如何真正地去优化游戏？”可能正如你所料，答案是：“视情况而定，”这取决于你使用的浏览器和想要做的事情。

优化工作的第一步是要弄清你可能从一些优化中收获哪些益处，如前所述，性能分析就是一个很好的着手点。CPU 数据汇总和堆快照都能够帮助你确认，哪些地方可能应该是你努力的目标——前者是因为它能够告诉你，哪些代码占用了大量时间，后者是因为它能够让你了解到，哪些类型的对象是你正在大量创建的。对数以千计存在的对象进行优化

是一个很好的开始。

接下来，既然已找出了需要施以援手的代码，那么就查看一下，修改算法是否能带来帮助，或者改变语法是否有提升性能的可能。JavaScript 是一种很灵活的语言，有时这种灵活性却会无意中损害到性能，有时一些做法看上去好像很高效，但实际上最终要消耗一些 CPU 周期。与诸如 C 的其他语言不同，在 C 语言中，可通过检查生成的汇编代码来了解编译器所认为的代码应有的运行方式，但在 JavaScript 中，弄清这一点的唯一做法就是编写测试，然后跨不同浏览器测试代码。

现在，举一个以三种不同方式创建对象的简单例子(见代码清单 7-1)：

代码清单 7-1：对象创建方法

```
var Obj1 = function() {}
Obj1.prototype.yay = function(x) {};
Obj1.prototype.boo = function(y) {};

var Obj2 = function() {
 this.yay = function(x) {};
 this.boo = function(y) {};
}

var Obj3 = function() {
  function yay(x) {};
  function boo(y) {};
  return {
   yay: yay,
   boo: boo
  }
}
```

这三种对象创建模式中的每一种都会生成一个行为完全相同的对象，但在 Chrome 18 中，其中最快的方法(Obj1，使用原型)比最慢的方法(Obj2，使用赋给 this 的匿名方法)速度快了 25 倍还多。要运行该测试，可访问 http://jsperf.com/object-creation-tests。

现在，在一台大约 2010 年出品的 MacBook Pro 上，第二种方法仍然录得每秒钟 170 万个对象，所以，除非你要创建非常多的对象，否则在一般游戏中，不同方法之间的差别不会特别明显。这正体现了问题的核心所在：务必先进行性能分析，然后测试自己的直觉，确保花费在优化上的时间是有效的。

诸如 http://jsperf.com 之类的站点把测试设置变成了一件很容易的事情，无论你想要优化什么它都可以帮你测试，并且能够帮助你快速判断自己的直觉是否正确，以及是否值得花费精力来获得潜在的速度提升。

7.6　在移动设备上调试

在桌面上进行调试是可以的，但因为本书是一本介绍移动游戏的书，所以，在真实的硬件之上进行调试的做法可谓无可替代。遗憾的是，除了 Android 上的 Chrome 外，没有其他内置方式可用来调试你的页面。

若你正在使用 Android 的 4.0(Ice Cream Sandwich，冰淇淋三明治)或更新版本，并已在设备上安装 Chrome，在桌面上安装 Android 开发工具包(Android SDK)，那么可以依照 Google 文档中描述的做法来启动远程调试(Remote Debugging)，文档的访问地址是 http://code.google.com/chrome/mobile/docs/debugging.html。

对于 iOS 来说，事情就没那么方便了。若只是为了监控 JavaScript 错误，可打开调试控制台，打开控制台的做法是依次选中 Settings | Safari | Advanced。现在，任何被加载的页面都会在其顶部显示 Safari 的 Debug 控制台；若存在错误，那么这些错误会被标出，可单击这一选项卡来查看一些更详细的内容。在希望深入跟踪一些难以调试的平台特定问题时，这一控制台很有可能无法满足你的需求。好消息是，存在一个名为 Weinre 的工具，该工具是 Apache 的 cordova 项目的一部分，用来添加基本的远程调试功能，它的访问地址是: https://github.com/apache/incubator-cordova-weinre。

Weinre 的工作方式是在某台计算机上运行一个基于 Java 的服务器，然后让移动设备通过包含适当的 script 标签来连接该服务器。在连接上后，你就可以访问其行为类似于“开发者工具”的选项卡的一个有限子集(见图 7-17)。

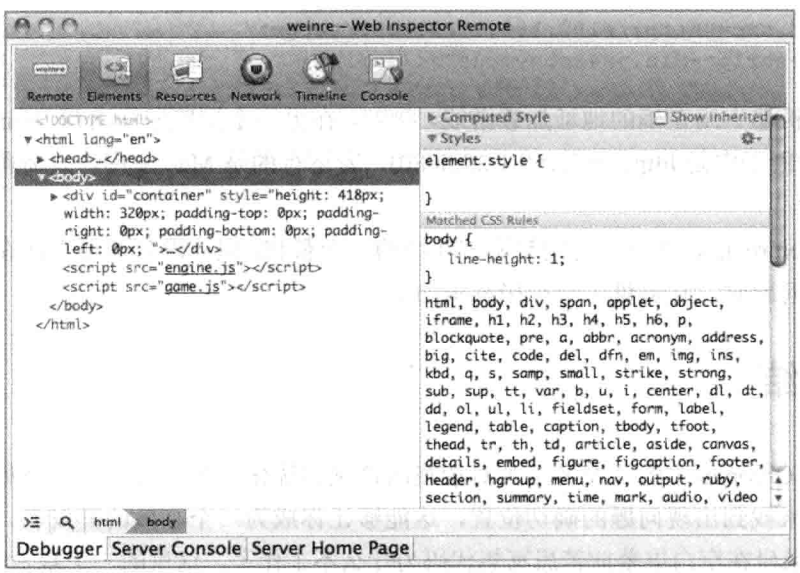


图 7-17　Weinre 远程审查器

因为 Weinre 仅是你在移动设备上加载的一个脚本，所以你不能访问脚本调试器或 Profiles 选项卡，但可在一个 JavaScript 控制台中审查元素和执行代码。要开始使用 Weinre，

需要直接下载 Java 的 JAR 文件，又或者，若是在 Mac 上，则可以下载 Mac 包。

若已经下载 JAR 文件，那么在已解压的 Weinre 目录下执行以下命令来运行它：

```
java -jar weinre.jar ~DHboundHost -all-
```

若下载的是 OS X 包，那么需要在主目录中创建一个名为.weinre(开头的一点很重要，别忘加上)的目录，然后使用以下内容创建一个名为 server.properties 的文件：

```
boundHost: -all-
```

接下来，双击 Weinre 应用启动它。

这两种情况下，boundHost 选项都是必需的，这样移动设备才能访问 Weinre 服务器。

警告：使用被设置成-all-的 boundHost 选项运行 Weinre 具有潜在的安全风险，你只应在可信任的局域网上这样做。

为使用 Weinre 所做的正常设置如下：把开发机设置成一个 Web 服务器，同时在上面运行 Weinre 的 Java 服务器；获取机器在局域网中的 IP 地址(见第 6 章中的介绍)，然后在游戏的 HTML 中硬编码指向该机器的 script 标签。例如，若你的开发机的 IP 地址被设成 192.168.1.20，且 Weinre 运行在端口 8080 上(这是默认端口)，那么将以下代码添加至游戏的 HTML 文件中：

```
<script src="http://192.168.1.20:8080/target/
target-script-min.js#anonymous"></script>
```

在你的机器上用正确的地址加载游戏，接着，在另一个浏览器中访问 Weinre 服务器的地址(在上述例子中是 http://192.168.1.20:8080)，若运行的是 Mac 应用，只需单击其他选项卡之一即可。

尽管 Weinre 并非桌面上的完整调试环境的一个替代，但在遇到需要直接在移动设备上进行调试的问题时，它提供了一种极有用的工具。

7.7 小结

内置在 Chrome 中的开发者工具对游戏的调试和优化非常有帮助，知道如何使用浏览器中的工具来找到出现问题的确切位置，这能够让你成为一个高效的开发者，因为从今之后你再不用呆坐在空白屏幕前苦思冥想代码为什么不工作了。这里的一个基本经验法则是，在代码失灵时，一开始先查看 Network 选项卡，接着加入 console.log 语句，然后进入全面的单步调试阶段。在移动设备上，可选的做法更有限，不过诸如 Weinre 一类工具的使用意味着有更多的信息可用来调试游戏。

第 8 章

在命令行上运行 JavaScript

本章提要

- 安装服务器端 JavaScript 环境
- 了解 Node.js
- 安装和使用 Node 模块
- 编写自己的 Node 命令行模块

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 8 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

8.1 引言

贯穿其整个历史，JavaScript 始终作为占据主导地位的浏览器语言在使用，虽然它曾进军服务器端，这要回溯至 Netscape 在 1996 年推出 LiveWire 之时，但服务器端的 JavaScript 似乎从未赢得太多关注。到 2010 年，这一情况开始有所改变，这是紧跟在 2009 年 11 月 JSConf 大会展示了 Node.js 之后发生的事情，突然之间——这几乎无法解释——人们对独立于浏览器运行 JavaScript 这样的做法产生了兴趣，在此后的几年中，JavaScript 已经变成了一种完全可接受的服务器端和命令行语言。本章将向你展示如何通过命令行安装、运行和编写用于支持游戏的脚本。

8.2 了解 Node.js

虽然还有其他一些服务器端的 JavaScript 环境可用，但到目前为止，Node.js(现在一般称为 Node)是最受欢迎的，且其提供了最好的跨平台支持(Windows、OS X 和 Linux)。基于这一普及性，Node 成了本书要讨论的 JavaScript 环境。

Node 的核心思想之一是，所有 I/O(输入/输出)都应该是非阻塞的，这意味着任何时候服务器在等待一些数据或一些输入时，它都不应该妨碍其他代码的执行。Node 处理这一问题的做法是使用回调(callback)，当阻塞的资源变为可用之后，回调就会自动执行，这通常称为事件式编程(evented programming)。这是一种 JavaScript 程序员很熟悉的编程模式，因为大部分用户界面(包括网页)都采用了这种编程做法：使用回调来添加事件监听器，在事件被触发时，回调就会被调用。

在 Node 中，编写非阻塞式代码是基本要求，因为 Node 是单线程的，这意味着在某个给定时刻只有一个线程在执行。若代码随意暂停执行，那么整个服务器就会停止接收请求，这看起来像是缺点，但以单线程模式运作，Node 可以在不必占用大量内存的情况下处理数目庞大的并发请求。采用单线程模式也意味 Node 不需要在线程之间切换上下文，这带来了性能的提升。结合这一点和底层的 JavaScript 引擎 V8(驱动 Chrome 的引擎)的速度，你就可以明白大家的兴奋点所在：一种多年来被大家贬为玩具的脚本语言突然间在基准方面打败了 Python、PHP 和 Ruby 之类服务器端语言的中坚成员。

服务器端 JavaScript 在浏览器外部运行，你希望这样做可能出于两大原因。第一是想要编写一个能够处理 Web 请求的服务器，第二是想要编写能够自动执行某些任务的命令行脚本，本书对这两种用途都进行了讨论。

本章讨论使用 Node 编写 JavaScript 命令行脚本，这些脚本要做的是诸如诊断代码以及包装和缩减 JavaScript 一类事情，第 18 章和第 19 章将介绍使用 Node 来构建游戏服务器，这是事件式单线程服务器擅长的工作。

8.3 安装 Node

随着 Node 0.6 的发布，Windows 上的简易安装已成现实。在这之前，需要安装诸如 Cygwin 一类 UNIX 风格的 POSIX 环境来实现 Node 的运行。从 0.6 版开始，Node 自带了 Windows 和 OS X 平台的安装包，可从 http://nodejs.org 上下载。

可运行安装包并按照提示来搭建 Node 的运行环境，除非使用的是 Windows，在这种情况下你没有其他选择。但安装程序并非安装 Node 的理想方法，因为它不会在安装的同时设置好开发环境，没有这样的开发环境，你就不能安装一些使用了本地化的 C 或 C++代码的模块。出于这一原因，应遵从以下这些特定平台的安装说明。

8.3.1　在 Windows 上安装 Node

在 Windows 上，当前唯一可做的选择是通过包来安装 Node 或使用 Visual Studio 进行编译。遗憾的是，在安装一些包含了需要编译的本地化源代码的模块时(如后续章节中用到的 node-canvas 模块)，你会遇到一些困难。

截至撰写本书时为止，node-canvas 模块还不存在最新的、预先构建好的 Windows 库，所以，要跟随下一节中使用了 node-canvas 模块的教程来练习，可从 www.vmware.com/products/player 下载 VMWare 的 VMPlayer 软件。VMPlayer 是一个免费软件，它能让你在 Windows 上运行一台虚拟的 Linux 计算机，可从 www.thoughtpolice.co.uk/vmware 下载一个 Linux 映像。

找到有着最大版本号(截至撰写本书之时为止是 11.10)的 Ubuntu Desktop 镜像，下载它，然后通过 VMPlayer 运行它。在 VMPlayer 中搭建并运行该虚拟机后，就可以通过菜单 Applications | Accessories | Terminal 启动命令行，从这个地方开始，按照 Linux 的安装说明来继续安装过程。要将文件复制到 Linux 虚拟机中，只需把它从你的桌面上拖放到虚拟机中，虚拟机就会把它添加到桌面上。

8.3.2　在 OS X 上安装 Node

在 OS X 上，你同样需要一个构建环境来安装本地化的模块，这意味着要安装 XCode 或安装 XCode 的命令行工具。若不打算使用 XCode IDE，那么可以免去安装 IDE 的麻烦，只安装免费的命令行工具即可，https://developer.apple.com/downloads 上提供了该工具的免费下载。

如果尚未用来访问该页面的免费开发者 ID 账号，那么需要创建一个。在设置了构建环境(XCode 或上述下载)之后，若尚未安装 Homebrew，就安装它，Homebrew 提供了一种用于安装各种包的隔离环境，可按照 http://mxcl.github.com/homebrew 上的提示来下载 Homebrew。在安装 Homebrew 之后，可以通过终端(Terminal)来运行以下命令：

```
brew install node
```

这一行命令代码的作用是安装一个最新版本的 Node，这样你就算搭好了 Node 的包管理器 npm 的运行环境了。

8.3.3　在 Linux 上安装 Node

在 Linux 上，事情一般较为简单一些，若已提供预先安装好的包管理器，那么可以使用它来完成环境的搭建工作。在 Ubuntu 上，应该运行以下命令：

```
sudo apt-get install node
```

若包管理器没有最新版本，那么可从 http://nodejs.org 下载源代码包，对它进行解压，然后就可以开始运行标准的构建和安装命令了：

```
./configure
```

```
make
make install
```

你可能需要以 root 用户身份运行最后一行命令，以便能以全局模式安装 Node。

8.3.4 追踪最新版的 Node

Node 是一个飞速发展的项目，若希望跟上先进技术的步伐，可从 GitHub 资源库下载它的通常被称为 HEAD 的最新版本，下载地址是 https://github.com/joyent/node.git。

作为开源软件的一种常见情况，在生产项目中你应谨慎使用 Node 的最新版本，锁定一个编号版本通常会更安全一些，除非你迫切需要最新版本中的某个功能或是对某个错误的修正，而且正在开发的是一个较大型的、不会很快发布的项目。

8.4 安装和使用 Node 模块

在安装了 Node 后，你就可以任意使用人们打包提供的数百个模块了。要实现这一点，使用 npm，即 Node 包管理器(node package manager)，该管理器提供一种自动化方式来下载和安装库和实用程序。以前，npm 和 Node 是分开安装的，但现在它们已被打包在一起。若尚未安装 npm，请确保自己运行的是一个较新版本的 Node。

若尚未安装 npm，你仍可按 http://npmjs.org/上的指示来安装它，不过在此之前，先确认一下自己安装的 Node 最新的。

8.4.1 安装模块

默认情况下，npm 把包本地安装在当前目录下的名为 node_modules 目录下。在构建服务器端应用时，这具有重大意义，这样你就可以做到准确控制所使用的库版本。不过，在通过命令行使用 Node 时，你通常会希望以全局模式安装某些模块，这样无论当前处在哪个目录下，你都能访问到模块的执行文件。

要以全局模式安装某个模块，使用--global 选项，接下来要安装的 jshint 是一个特别有用的模块，该模块派生自 Douglas Crockford 的 JSLint 工具，它解析 JavaScript 代码并反馈哪些部分需要调整。

要安装该模块，通过命令行运行以下命令：

```
npm install --global jshint
```

该命令安装 jshint 模块并把模块的可执行文件设置成是可通过命令行访问的。

8.4.2 诊断代码

在安装了 jshint 这一 Node 模块后，现在可通过命令行运行 jshint 对代码执行一个快速的语法检查。JSHint 可以发现诸如漏写分号一类的错误或是一些奇怪的结构，这些结构不一定会妨碍代码的运行，但可能导致难以发现的错误。

要针对某个文件运行 jshint，在命令行上运行该命令，并在命令的后面带上该文件的名称，例如：

```
jshint engine.js
```

jshint 生成一个描述性的警告列表，其中包括了行号和列号，这些内容有助于你提升自己编写的 JavaScript 的质量。

8.4.3　缩减代码

在部署游戏以供玩家玩乐之时，你会希望游戏有着尽可能快的加载速度。一种缩短加载时间的做法是保证在网络上传输的内容尽可能少，可以运行众多的 JavaScript 缩减工具之一来处理自己的 JavaScript 代码，正是为了降低文件的大小，这些工具才被编写出来。

JavaScript 缩减工具接收 JavaScript，删除其中的空格，然后重写并缩短变量和参数的名称，以此来显著降低代码文件的大小。以第 3 章中的游戏 Alien Invasion 为例，通过运行 uglify-js 这一缩减工具来处理 engine.js 和 game.js，代码文件的大小从 19K 降至 11K 多一些，压缩率超过 40%。

要安装 uglify-js，通过 npm 以全局模式安装该模块：

```
npm install --global uglify-js
```

uglify-js 的执行文件只接收一个文件，所以若希望把多个文件合并成一个(你应该这样做)，需要单独使用命令把它们串接起来，在 Windows 上，可运行以下命令：

```
type file1.js file2.js > all.js
uglify-js all.js > all.min.js
```

在 OS X 和 Linux 上，运行以下命令：

```
cat file1.js file2.js | uglify-js > all.min.js
```

现在你得到一个文件——all.min.js——该文件包含了所有被传进来的文件的代码，这些代码文件被缩减成单个易于提供的文件。

减少所提供文件的数量也能够加快游戏的加载速度，因为浏览器必须发起的每个独立请求都会占用一些额外时间，在移动设备上尤其如此。就一个产品级游戏而言，在部署游戏之前，应该编写一个外壳脚本，这样就能运行该脚本从而以一种自动化方式实现这一点。

8.5　创建自己的脚本

尽管已有数百个 Node 模块写成在用，但在构建游戏时，你会有一些特定需求，在这方面，一些简单的服务器端脚本能够带来帮助。就你可使用的脚本语言来说，存在多种选择，这其中包括了 Bash、Script、Python、Ruby 和 PHP 等，不过，因为你的游戏将是 JavaScript 类型的游戏，游戏的库将用 JavaScript 来编写，所以同样使用 JavaScript 来编写命令行脚本

会合理一些。

为让你获取一些构建 Node 模块的经验，本节将详细讲解构建脚本的过程，该脚本通过一个图像文件目录生成精灵表和一些相应的 JSON。图 8-1 给出了一个示例性的输出图像，其中显示了按行排列的精灵。

图 8-1　生成的精灵表

这一过程中唯一存在的一个小问题是，要在 Windows 上编译本节中用到的 node-canvas 模块不太容易，因为它依赖于本地化的 C(要绕开这一问题，参见之前章节介绍的在 Windows 上搭建 Linux 虚拟机运行环境的做法)。

8.5.1　创建 package.json 文件

首先为要编写的脚本创建一个名为 spriter 的新目录，然后在该目录下创建并打开一个名为 package.json 的文件，package.json 是一个 npm 用来获取模块及其依赖信息的文件。在 package.json 文件中填充代码清单 8-1 所示的内容，并在适当的位置上换上你的姓名和 E-mail 地址。

代码清单 8-1：package.json 文件

```
{
  "name": "Spriter",
  "description": "A Sprite Map generator",
  "author": "Your Name <youremail@domain.com>",
  "version": "0.0.1",
  "dependencies": {
    "canvas" : "0.10.2",
    "futures": "2.3.1"
  },
  "bin": "./bin/spriter",
  "main": "./spriter.js"
}
```

其中的 name、description、author 和 version 等字段的含义基本上不言自明，若你打算把自己的模块发布到 npm 上，那么 version 字段在更新模块时会起到十分重要的作用。dependencies 字段是一个哈希，其中存放了本模块所依赖的其他模块及其应该安装的版本；bin 字段稍后会用到，届时该模块会被链接以允许你在任何目录下以命令行方式调用这一脚本；最后是 main 参数，该参数指明哪个文件中存放了用来处理导出的主脚本，bin 和 main 现在还不是必需的，但后续章节中有需要用到它们的地方。

这段代码用到了一个小巧的服务器端画布模块 canvas，该模块提供了一个可以通过服务器端 Node 代码来使用的画布 2D API，它依赖于两个需要你来安装的、名称分别为 cairo

和 pixman 的图形库。此外，代码还用到了 futures 模块，该模块提供了 Promise 和 Deferred 功能。若你不熟悉 Promise 和 Deferred 也不必担心，稍后会谈及这部分内容。

在 Linux 上，可安装 ibcairo2-dev(Debian 和 Ubuntu)或 cairo-devel(Fedora 和 openSUSE)包，在 Ubuntu 上——或在你运行的 Ubuntu 虚拟机上——这意味着通过命令行运行以下命令：

```
sudo apt-get install libcairo2-dev
```

在 OS X 上，通过 Homebrew 来安装 cairo，使用以下命令：

```
brew install cairo pixman
```

在完成 cairo 库的安装后，在 spriter 目录中运行以下命令来安装所有依赖：

```
npm install
```

该命令下载画布模块依赖并将其安装到 node_modules 子目录中，接下来就应该构建 Node 模块脚本了。

8.5.2　使用服务器端画布

首先需要测试服务器端画布的功能，这是为了确保它的可用性，测试做法是以普通的绘制内容为例，在画布上绘制两个重叠的矩形。

在编写 Web 服务器时，Node 中的文件 I/O 通常以异步方式实现，不过，在编写命令行脚本时，可稍放宽回调模式的要求，使用方法的 Sync 版本，这种版本以同步方式完成工作。

Node 提供了一个名为 fs.writeFilSync 的方法，该方法接收一个文件名和一个缓冲区作为参数，然后把缓冲区中的内容写入文件中。node-canvas 模块有一个名为 canvas.toBuffer () 的方法，该方法能够生成画布的缓冲区，可以在回调中以异步方式使用 canvas.toBuffer ()，不过，在这个例子中可以使用同步版本。

在 spriter 目录中创建一个名为 spriter.js 的文件并打开它，然后将代码清单 8-2 中的代码输入其中。

代码清单 8-2：Spriter.js 中的样板代码

```
var fs = require('fs'),
    Canvas = require('canvas'),
    canvas = new Canvas(200,200),
    ctx = canvas.getContext('2d');
ctx.fillStyle = "#CCC";
ctx.fillRect(0,0,100,100);
ctx.fillStyle = "#C00";
ctx.fillRect(50,50,100,100);
fs.writeFileSync("./sprites.png",canvas.toBuffer());
```

这段代码的绝大部分都是标准的画布代码，与你在浏览器中编写的类似，唯一不同的

地方是最初的 require 语句和用于输出文件的方法调用。

代码调用 require("..")加载通过 npm 安装的模块，然后把返回值赋给变量以备使用。fs 模块是一个内置的库，它提供了 Node 中的基本文件系统访问。

上一节中安装的 canvas 模块提供一些功能来模仿客户端的画布对象，可以通过调用以下语句来创建新的画布对象：

```
new Canvas(width,height)
```

代码清单 8-2 创建了一个 200x200 像素的画布，然后以与客户端相同的做法来检索上下文对象。

最后，在进行了几个简单的绘制调用之后，代码调用以下方法把画布输出到一个.png 文件中：

```
fs.writeFileSync("./sprites.png",canvas.toBuffer());
```

该行代码创建一个缓冲区对象，然后将缓冲区中的内容写到第一个参数指定的文件中。

要测试这一脚本，可以运行以下命令：

```
node ./spriter.js
```

该行命令应会在同一目录下生成一个名为 sprites.png 的文件，文件中的内容是两个重叠的矩形。若遇到任何错误，再次确认已正确安装了 canvas 模块。

8.5.3 创建可重用的脚本

要把 spriter 脚本变成既能为其他模块所用，又能通过命令行使用，需要对 spriter.js 文件做几处改动，以及要把必需的 spriter 脚本添加到 bin 目录中。

Node 提供了一个名为 exports 的对象，任何时候需要一个文件，require()会返回一个 exports 对象。若不打算返回一个对象，那么你也可以覆盖返回的对象，做法是把 module.exports 设置成希望返回的任何内容。在 spriter 这个例子中，你只需公开一个函数，该函数在精灵文件中创建精灵。

重新编写 spriter.js 文件，修改后的文件包含如代码清单 8-3 所示的内容。

代码清单 8-3：一个导出的 spriter.js 文件

```
var fs = require('fs'),
    Canvas = require('canvas');

function spriter() {
  var canvas = new Canvas(200,200),
      ctx = canvas.getContext('2d');
  ctx.fillStyle = "#CCC";
  ctx.fillRect(0,0,100,100);
  ctx.fillStyle = "#C00";
  ctx.fillRect(50,50,100,100);
  fs.writeFileSync("./sprites.png",canvas.toBuffer());
```

```
}

// Make the spriter method available
module.exports = spriter;
```

现在，功能被包装在了一个可供外部调用的函数中。

接下来，在模块的 bin 子目录下创建一个名为 spriter 的文件(没有扩展名)并打开它，然后将代码清单 8-4 所示代码添加到该文件中。

代码清单 8-4：　bin/spriter 中的命令行脚本

```
#!/usr/bin/env node
var spriter = require('../spriter');
spriter();
```

该脚本的唯一目的是加载你刚才编写的模块，然后调用 spriter 函数。

需要把该文件变成可执行的，做法是打开文件权限的可执行位，在 spriter 目录下通过命令行运行以下命令：

```
chmod a+x bin/spriter
```

下一步，可使用 npm link 命令把 bin 目录中的文件变成整个系统可用的，同时自己仍能修改其中的代码。在 spriter 目录下运行以下命令：

```
npm link
```

现在，在系统中的任何地方，都可以通过命令行运行 spriter 命令，创建这个(没有什么用的)sprites.png 文件。在下一节中，你将把 spriter 变成一个有用的精灵地图生成器。

8.6　编写一个精灵地图生成器

在弄清组建一个 Node 模块的逻辑流程之后，接下来就是把模块变成真正可用的，该模块的目标是根据一个图像文件目录生成精灵地图(sprite map)的 PNG 文件和相应的 JSON 文件。在 HTML5 游戏开发中，精灵地图很有用，因为你不会希望通过加载数百个独立的图像文件来处理动画，所以，如你所见，情形正好相反，你只需加载一个或几个已在其中放置了许多图像的精灵表文件即可。

依照以下步骤来实现脚本的目标：

(1) 接收一个图像文件目录作为参数，其中的图像文件按数字顺序编号(即 ship01.png、ship02.png、...、enemy01.png、enemy02.png、...)。

(2) 输出一个精灵地图，其中的每一行图像都对应了一个按顺序编号的文件列表。

(3) 输出一个详细说明了每个精灵的像素位置和帧数的 JSON 文件，这些精灵可被加载到接下来的几章将要构建的游戏引擎中。

接下来的几节将把该脚本的各部分组合起来。

8.6.1 使用 Futures 模块

使用 node-canvas 模块加载图像的做法与在客户端加载它们的做法相同：设置 src 属性，然后等待一个 onload 事件。因为这是一个异步事件，所以从理论上来说，图像有可能不按照顺序加载(或完全不加载)，跟踪所有这些请求需要编写一些内务(housekeeping)代码或大量的嵌套调用。

幸运的是，我们可以使用 Promise 模式，该模式封装了这种以串行方式处理将来事件的做法，在第 5 章讨论 Deferred 时，你已了解了一个关于该模式的例子。Node 有几个不同模块都提供了 Promise 和 Deferred 对象(与 jQuery 中的类似)，其中最好用的是 Futures 模块，该模块提供了一堆不同类型的对象来简化工作。因为在 Node 和浏览器中都是可用的，所以该模块绝对值得你花时间去了解一番。

spriter 脚本使用的是 Futures 中的 Join 模块，该模块提供了一种方式来轻松加载一串异步事件，只有在所有事件都完成之后才触发回调。

要创建一个新的 Join 对象，只需调用以下方法：

```
var join = Join();
```

可以调用以下方法把一个新的 Promise 对象添加该对象中：

```
var promiseFunction = join.add();
```

把所有 Promise 对象添加到 join 中后，就可以调用以下方法：

```
join.when(function(args) {
  /* Triggered when all promises have been called */
}
```

例如，代码清单 8-5 中的代码展示了如何使用 Futures 模块来加载两个图像：

代码清单 8-5：使用 Join 的例子

```
var join = Join(),
    img1 = new Image(),
    img2 = new Image();
img1.onload = join.add();
img1.src = "images/image1.png";
img2.onload = join.add();
img2.src = "images/image2/png";
join.when(function() {
  // Both images have been loaded
});
```

这里还需要编写一些记账代码来跟踪两个图像可能的非顺序加载，其中的唯一紧要之处是在完成所有图像加载之后对 join.when 的调用。代码清单 8-5 给出的是基本代码，可对之加以扩充，以支持在传给 spriter 脚本使用的目录中放置任意数量的图像。

8.6.2　自上而下进行编码

现在是时候是使用生成精灵地图的真正代码来替换掉 spriter.js 中的基础级画布演示代码了，在这个例子中，自上而下的编码方式可能是最简单的，先从生成精灵地图这一最高层面的方法开始，然后编写使其工作所需的辅助方法。

从高层角度看，spriter 需要加载一个放有图像的目录，并要依照文件名称来分行排序图像；然后，它需要基于每个精灵的宽度、每行图像的数目和每行的高度计算其需要创建的画布的尺寸，spriter 假设所有行中的所有精灵都具有相同的尺寸(这是游戏引擎的一个需求，所以是合理的假设)。接下来，它需要真正把每个图像绘制到画布上并生成 JSON；最后需要做的是，生成 sprites.png 和 sprites.json 文件。

首先重写 spriter 方法，打开 spriter.js，用代码清单 8-6 中的代码替代掉该文件中的所有代码。

代码清单 8-6：重写 spriter 方法

```
function spriter(directory) {
  var files = fs.readdirSync(directory),
      rowData = {},
      // Load all the images
      join = loadImages(directory,files,rowData);

  // Wait for the all images to load
  join.when(function() {
    // Get the dimensions of the output sprite map
    var dimensions = calculateSize(rowData),
        canvas = new Canvas(dimensions.width, dimensions.height);

    // Draw the images to the canvas and return the JSON data
    var jsonOutput = drawImages(rowData,canvas);

    // Write out both the sprites.png and sprites.json files
    fs.writeFileSync("./sprites.png",canvas.toBuffer());
    fs.writeFileSync("./sprites.json",JSON.stringify(jsonOutput));
    util.print("Wrote sprites.png and sprites.json\n");
  });

  // Make the spriter method available
  module.exports = spriter;
```

现在讲解一下上述代码，文件在开始部分加进来的 Node 模块现在又多了几个，这些模块都是需要用到的，其中包括了之前提到的 Futures 模块。出于更便于访问的目的，代码还把几个对象(Canvas.Image 和 Futures.join)从现有模块中提取出来，放到了顶层变量中。

接着，spriter 函数接收一个将会通过 bin/spriter 脚本传递进来的目录名称，该函数使用一个对 fs.readdirSync 的快速调用来载入该目录下的所有文件；然后，这些文件会被传递给一个目前尚未编写的 loadImages 方法，该方法返回一个 join 对象，在所有图像完成加载之

后，该对象的回调就会被触发。此外，loadImages 方法还要填充 rowData 对象，这是一个把精灵名称和组成一行的精灵图像列表映射起来的对象。

接下来，在触发 join.when 回调之后，代码调用了一个也是目前尚未编写的名为 calculateSize 的方法，该方法通过图像的行数计算出画布的总体尺寸，然后创建这一尺寸的画布对象 canvas。最后，在画布上进行真正的绘制，这部分工作由最后一个需要你来编写的方法 drawImages 完成。

现在，canvas 对象包含了已渲染的精灵地图，jsonOutput 变量中存有指明了每个精灵位置的精灵数据，所以，剩下要做的就是把两个文件写到硬盘中。要把 jsonOutput(这是一个 JavaScript 对象)转换成字符串，可调用内置的 JSON.stringify 方法。

若是在编写客户端代码，那么你可能需要把所有这些代码都包装在闭包中，以此来防止名称空间被污染，但因为 Node 为每个文件赋予了各自的作用域，所以这种做法不是必需的。

8.6.3　加载图像

现在可以专注于编写 loadImages 方法了，该方法接收的参数包括目录、文件列表和一个需要使用图像列表进行填充的行数据结构。

该方法的工作是遍历列表中的每个文件，将其添加到适当的行中，然后创建一个 Image 对象，接着开始加载图像，将 onload 方法绑定到 Join 对象上是为能按照之前介绍的做法来处理异步加载。

loadImages 方法使用正则表达式从文件名中提取名称和文件编号，从而支持按行索引和排序。将代码清单 8-7 中的代码添加到 spriter.js 的末尾处。

代码清单 8-7：loadImages 方法

```
function loadImages(directory,files,rowData) {
  var fileRegex = /^(.*?)([0-9]+)\.[a-zA-Z]{3}$/,
      join = Join();

  for(var i=0;i<files.length;i++) {
    (function(file) {
      var results = file.match(fileRegex),
          img = new Image();

      if(results) {
        var rowName = results[1],
            fileNum = parseInt(results[2],10);

        img.onload = join.add();
        img.onerror = function() {
          util.print("Error loading: " + file + "\n"); process.exit(1);
        }

        img.src = directory.replace(/\/$/,"") + "/" + file;
```

```
        rowData[rowName] = rowData[rowName] || [];
        rowData[rowName].push([fileNum,img]);
      }
    })(files[i]);
  }

  return join;

}
```

loadImages 有几处代码颇令人感兴趣，第一处是存放在 fileRegex 中的正则表达式，该表达式使用两个捕获组来抓取行名称和文件编号。spriter 假设每个文件都使用了 filename0000.ext 这一名称格式，其中最后的 0000 表示精灵的行编号。在生成图像列表时，输出文件常用到这种做法。

正则表达式的第一个捕获组(捕获组是一些被保留下来的值，创建方式是使用一对括号把正则表达式的一部分围括起来)如下：

```
(.*?)
```

.*意味着匹配任何字符，不过在末尾处加一个问号意味着把匹配器设成非贪婪模式，即它可以匹配任意字符直至遇到匹配分组。

第二个捕获组——([0-9]+)——匹配任意数字组合，其中包括 1、001 和 9999 等。

若字符串不匹配，那么对 String.match (regexp)的调用返回 null，若正则表达式匹配，则返回任意组的匹配值。代码清单 8-7 中的代码在 results 变量中存储与正则表达式相匹配的结果，并从捕获组中提取出 rowName 和 fileNum 值。

代码的第二个有趣之处是 for 循环中的匿名函数，你可能以前遇见过这一模式，该模式称为立即调用的函数表达式(immediately invoked function expression，IIFE)。在 JavaScript 中，该模式非常有用，因为它创建了自己的作用域，这使得你能够保存变量以备之后在回调中使用，在这个例子中，使用它是因为 onerror 回调需要提醒用户哪个文件存在问题。

在不使用 IIFE 的情况下，以下的 onerror 回调：

```
img.onerror = function() {
        util.print("Error loading: " + file + "\n"); process.exit(1);
      }
```

仅输出最后被赋给 file 变量的值，这有可能会导致极大的误解。通过使用 IIFE，匿名函数会创建一个闭包，这意味着 file 变量会被保存下来。

onload 方法被之前提到的对 join.add ()的调用所替代，方法返回了变量 join，因为 spriter 要使用它来指明所有图像在何时完成加载。

loadImages 创建了主要的数据结构 rowData，该对象被当作参数传递进来，不过因为 JavaScript 中的对象是通过引用传递的，所以，该方法对 rowData 对象所做的任何改动在调用方法中都是可访问到的。在这个例子中，该方法使用某种数据结构填充 rowData 对象，

这一结构看起来如下所示：

```
{
  'sprite_one': [ [ 2, Image ],
                  [ 1, Image ],
                 ... ],
  'sprite_two': [ [ 1, Image ],
                  [ 2, Image ],
                  ... ]
}
```

该对象的每个键都对应了一个条目数组，数组把精灵编号和实际的 Image 对象链接起来，这些图像可能是也可能不是按正确顺序排序的，这取决于操作系统返回文件列表的方式(这些文件不一定是已排序的)。

8.6.4 计算画布的尺寸

接下来是计算画布尺寸的方法，该方法的任务是把每行图像的高度相加，找出最大行的宽度，使用它作为最终的精灵地图图像的宽度，使用最大行是因为图像的尺寸必须是方形的。将代码清单 8-8 中的代码添加到 spriter.js 的末尾处。

代码清单 8-8：在 calculateSize 中计算图像尺寸

```
var maxSpriteWidth = 1024;

function calculateSize(rowData) {
  var maxWidth = 0,
      totalHeight = 0;

  for(var spriteName in rowData) {
    // Order by ascending number
    var row = rowData[spriteName],
    firstImage = row[0][1],
    width = firstImage.width * row.length,
    rows = 1;

    if(width > maxSpriteWidth) {
      rows = Math.ceil(width / maxSpriteWidth);
      width = maxSpriteWidth;
    }

    maxWidth = Math.max(width,maxWidth);
    totalHeight += firstImage.height * rows;
  }

  return { width: maxWidth, height: totalHeight };
}
```

这一方法相当简单，先遍历各行，从每行中取出第一个图像(应该还记得，行中每个图

像应该具有相同的大小)，然后使用图像的高作为行高，使用图像的宽与图像数量的乘积作为行宽。此外，它还使用 Math.max 来找出最大行宽，以此作为图像的最终宽度。接下来，它检查最终合成的图像的宽度是否大于最大允许的精灵宽度，若是则计算精灵的行数，并把最终精灵图像的宽度设成最大宽度。最后，它返回一个存放了计算好的宽度和高度的对象。

8.6.5　在服务器端画布上绘制图像

参照代码清单 8-6，现在剩下要做的事情就是编写 drawImages 方法，该方法接收行数据和已创建的画布作为参数，绘制 rowData 中的图像，然后把将被游戏引擎使用的 jsonOutput 作为输出返回。

这一方法实际上比预想中的简单，因为把图像绘制到画布上就是到 drawImage 的一个调用而已。你唯一需要留意的是，要按照图像的索引来排序每行数据，从而防止精灵以错误的顺序出现在画布上。将代码清单 8-9 中的代码添加到 spriter.js 的末尾处。

代码清单 8-9：创建 drawImages

```
function drawImages(rowData,canvas) {
   var ctx = canvas.getContext('2d'),
      curY = 0,
      jsonOutput = {};

   for(var spriteName in rowData) {
    // Order by ascending number
     var row = rowData[spriteName].sort(function(a,b) {
                                    return a[0] - b[0];
                                   }),
       firstImage = row[0][1],
       imageWidth = firstImage.width,
       rowHeight = firstImage.height,
       rowWidth = Math.min(imageWidth * row.length, maxSpriteWidth),
       cols = Math.floor(rowWidth / imageWidth),
       rows = Math.ceil(row.length / cols);

     jsonOutput[spriteName] = { sx: 0, sy: curY, cols: cols,
                           tilew: imageWidth, tileh: rowHeight,
                           frames: row.length };

     for(var i =0;i<rows;i++) {
       for(var k=0;k<cols;k++) {
         if(row[k+i*cols]) {
           ctx.drawImage(row[k + i*cols ][1],k*imageWidth,curY);
         }
       }
       curY += rowHeight;
     }
   }
```

```
    return jsonOutput;

}
```

drawImages 方法接受行数据和画布作为参数，然后循环遍历每一行。就每行而言，它会使用一个方法来调用 JavaScript 的 sort 方法，该方法根据数组中第一个元素排序图像；然后，它从每行中取出第一个图像来计算行的高度和每一行的宽度。

有了这些信息之后，它就可以根据每一帧的宽和高及画布当前的 y 位置(存储在 curY 中)为该行创建 jsonOutput 条目了。

接着，代码遍历行中的每个图像，根据该精灵的行数和列数，在正确的 x 和 y 位置上绘制它，之后更新 curY，从而保证该精灵的每一行和每个精灵都被绘制在正确的 y 位置上。

8.6.6 更新和运行脚本

随着 drawImages 的编写结束，现在的 spriter.js 就算大功告成了；不过，对 bin/ spriter 目录下的脚本文件需要进行一些更新，以便能够将传递给它的目录继续传递下去。修改 bin/ spriter，使其中的内容与代码清单 8-10 中的代码保持一致。

代码清单 8-10：更新后的脚本

```
#!/usr/bin/env node
var spriter = require('../spriter');
spriter(process.argv[2]);
```

其中的唯一改动之处是加入了 process.argv[2]，该变量把脚本名称之后的第一个参数传给 spriter 函数。

现在可通过命令行来运行 spriter 命令了，传入图像目录名称，脚本应会在你运行脚本的目录下输出 sprites.json 和 sprites.png 这两个文件。

若这是一个要通过 npm 提供的生产就绪脚本，你可能希望能将更多选项传给脚本，借此控制输出文件和用来匹配文件的正则表达式。所有这些代码在某种程度上都是样板代码，可通过现有的一些模块来自行解决这一问题。大多数时候，在为自己编写脚本时，你会极度关注如何自动完成构建和部署过程，所以，这里把处理众多不同选项的需求留作一道习题，交由你来完成。

在模块能够正常使用后，可创建该目录的一个 tarball 包，然后使用 npm install 命令并传入包名，以便在不同计算机上安装该模块。要了解更多信息，可参阅 npmjs.org 上的安装说明 http://npmjs.org/doc/install.html。

若希望把模块发布到 npm 的包列表中，以便能够通过 npm install 使用名称直接安装该模块，可参阅 http://npmjs.org/doc/publish.html 上的 npm publish 命令文档了解相关信息(需要首先使用 npm adduser 创建一个用户账号)。

8.7　小结

现在，你已经搭建好了 Node.js 的运行环境，安装了几个模块并使用其中的脚本诊断和缩减了自己的代码。你还编写了自己的模块和脚本来生成精灵地图和相应的 JSON 文件，在后面的章节中，可以使用这些文件，不必再手动创建精灵地图和逐个计算精灵和帧的位置。第 19 章和第 21 章将再次谈到 Node，届时它将会被用作编写多玩家游戏的 Web 服务器，这可是 Node 最擅长的用例。

第 9 章

自建 Quintus 引擎(1)

本章提要

- 设计和创建 Quintus API
- 创建高效的游戏循环
- 将传统继承添加至 JavaScript
- 构建事件系统
- 创建组件系统

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 9 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

9.1 引言

本章涵盖了 Quintus 这个 HTML5 移动友好游戏引擎最初的自举开发过程，这一可重用的开发者友好引擎将用在本书其余各部分中使用。本章介绍该引擎的基本结缔组织，下一章讨论资产的加载和渲染，以及用户输入的处理。

JavaScript 原本并非一种打算用于游戏开发的语言，但自从被当成一门语言，从最初主要用于动态选中和取消选中多选框起，它已走过了一条漫长的发展道路，现在的 JavaScript 非常适于当成一种交互式游戏开发语言使用。

9.2 创建可重用 HTML5 引擎的框架

虽然第 1 到第 3 章构建的游戏是一个完全可接受的一次性游戏，但它的代码相当脆弱，而且是特定于游戏自身的。在接下来的几章中，你将组建一个更通用的引擎，该引擎支持更好的游戏间代码重用。

该引擎将被命名成 Quintus，Quintus 在拉丁文中的意思是“第五”(这含义似乎不错)，它将被用于本书余下游戏的构建。如第 6 章中提到的，该引擎有两大依赖：jQuery 和 Underscore.js；你已知道，使用 HTML5 可在无任何依赖的情况下构建游戏；不过，若这样做的话，则意味着本书要包含更多非游戏相关的代码，以及只能将较少的时间用在实际的游戏开发上。

在使用 JavaScript 构建游戏时，你不需要推倒重构那些用于游戏开发的传统模式。作为一种语言，JavaScript 有其怪癖之处，但它极具可塑性，可对之加以塑造来迎合你最喜欢的编程风格。这并非意味着你能够以任何方式构建出高性能的游戏，某些开发风格较其他的更能与 JavaScript 的异步、单线程特性相适应。

9.2.1 设计基本的引擎 API

引擎的核心部分被放在单个文件中，该文件被命名为 quintus.js。要让引擎有所作为，需要往其中加入其他一些模块，这些模块提供了诸如渲染和输入之类的功能。Quintus 引擎具有几样特定的需求：

(1) 需要在同一页面上运行多个引擎实例，该需求确保引擎作为独立单元出现，不会干扰自身或页面的其他组成部分。

(2) 在可能的情况下，引擎应提供合理的选项默认值，以此免去搭建运行环境所需的大量配置。

(3) 引擎应足够灵活，对于简单的例子和较复杂的游戏来说都是可用的，且应支持不同的渲染引擎(比如 Canvas、CSS、SVG 和可能的 WebGL 等)。

从这些需求开始，引擎的构建工作以逆向方式展开，代码清单 9-1 展示了一种简单的 API 用法，说明如何使用 API 编写一个基本的动画例子。

代码清单 9-1：一个简单的 Quintus API 例子

```
var MyExample = Quintus();
MyExample.load('assetName.png',function() {
  var object = new MyExample.CanvasSprite({
                    asset: 'assetName.png', x: 0, y: 0
                });

  object.update = function(dt) {
    // Code to update the object
  };
  MyExample.gameLoop(function(dt) {
```

```
    this.clear();

    object.update(dt);
    object.render(this.ctx);
  });

});
```

这个简单的例子加载一项资产并创建一个对象，然后更新对象并在屏幕上绘制该对象。

对于一个有限制的例子来说，这可能已经足够了，但功能较全面一些的用例还要求有额外的一些舞台(stage)和场景(scene)功能，这些功能自动处理多个对象的更新和渲染，这种情况下，引擎会接管游戏循环的处理。一个功能稍齐全的游戏可能如代码清单 9-2 所示。

代码清单 9-2：一个较复杂的 API 例子

```
var MyGame = Quintus()
             .include("Input,Sprites,Scenes")
             .setup();

var spriteType1 = MyGame.CanvasSprite.extend({
  // Overrides for this type of object
});

var spriteType2 = MyGame.CanvasSprite.extend({
  // Overrides for this type of object
});

MyGame.load([ 'asset1.png', 'asset2.png', 'sprites.json'],function() {

  var scene1 = new MyGame.Scene(function(stage) {
    stage.add(new MyGame.SpriteType1({ ... Options .. }));
    stage.add(new MyGame.SpriteType2({ ... Options .. }));
  });

  MyGame.stageScene(scene1);
});
```

在这个例子中，游戏使用 Quintus.Input、Quintus.Sprites 和 Quintus.Scenes 等扩展模块来扩充基本的 Quintus 功能，然后创建两个可重用的精灵类型。该游戏加载了多项资产，其中包括一个精灵表；然后，在这些资产加载完毕后，它创建一个新的场景对象(scene1)；最后，它调用展现场景的方法(stageScene)，该方法启动基于场景的游戏循环(若循环尚未启动的话)并处理场景的更新和渲染。

9.2.2　着手编写引擎代码

既已定义出了开发者友好的引擎 API，现在打开 quintus.js，开始编写一些初始代码来

充当所有引擎代码的效仿基础。Quintus 借鉴了 jQuery 的做法，使用单个方法作为工厂方法，并将一个容器对象用作引擎的扩展。

要使用 Quintus 创建新游戏，只需调用 Quintus 方法，然后借助附加功能来使用和扩展这一对象。因为单个页面上可能存在多个 Quintus 实例，所以需要在每个实例中为引擎加载所有扩展。

要实现这一目标，可创建一个名为 Quintus 的函数，该函数创建、扩展并在最后返回一个新对象。代码清单 9-3 给出了该引擎的最初结构。

代码清单 9-3：基本的引擎结构

```
var Quintus = function(opts) {
  var Q = {};

// Some base options to be filled in later
  Q.options = {
    // TODO: set some sensible defaults
  };
  if(opts) { _(Q.options).extend(opts); }


  Q._normalizeArg = function(arg) {
    if(_.isString(arg)) {
      arg = arg.replace(/\s+/g,'').split(",");
    }
    if(!_.isArray(arg)) {
      arg = [ arg ];
    }
    return arg;
  };

 // Shortcut to extend Quintus with new functionality
  // binding the methods to Q
  Q.extend = function(obj) {
    _(Q).extend(obj);
    return Q;
  };

  // Syntax for including other modules into quintus
  Q.include = function(mod) {
    _.each(Q._normalizeArg(mod),function(m) {
      m = Quintus[m] || m;
      m(Q);
    });
    return Q;
  };

  // TODO: Additional Quintus Code goes here
```

```
    return Q;
}
```

这段初始代码提供了模块化架构的基础，引擎的剩余部分将基于这一架构构建。这段代码主要用于创建一个 options 对象，然后使用通过 opts 传进来的任意附加选项扩展该对象。

在接下来的代码中，可使用_normalizeArg 方法接收一个被传进来的串，这是一个用逗号分隔的名称串，然后把它转换成一个去除了所有空格的名称数组。该方法为你带来了一些方便，比如说，它使得你能以"sword, shield, health"方式而非["sword", "shield", "health"]方式进行编写。若你传给该方法的是一个数组，那么该元素数组会被直接使用，方法不对其做任何转换。使用_normalizeArg 的目的是防止传入一个包含列表。

Q.include 和 Q.extend 方法的作用是用 Sprites 和 Scenes 一类的附加模块来扩展 Quintus 的功能。为支持链式调用，它们仍旧返回 Q 变量。

Quintus 是一个方法，该方法接收一个可选的选项哈希作为参数，并返回一个具有基本功能的引擎实例。

9.3　添加游戏循环

正如你已知道的那样，游戏从帧到帧之间的切换，这一过程的真正执行是由游戏循环来主导的，循环负责更新游戏的状态，然后在屏幕上渲染游戏的当前帧。浏览器中的主要渲染和 JavaScript 引擎同在一个单线程中运行，这意味着你不能像在有着真正多线程的环境中那样，使用单一紧凑循环作为游戏循环；相反，如你在第 1 章所见，游戏循环必须使用定时器来实现，定时器释放 JavaScript 代码的控制权，把控制权交回给浏览器，这样浏览器才能更新页面和处理输入事件。

9.3.1　构建更好的游戏循环定时器

长时间以来，游戏循环定时器的构建都是使用一直存在于浏览器中的 setTimeout 和 setInterval 函数来实现的，这种做法的效果某种程度上可让人接受，但也存在一些缺点。

第一个缺点在速度较慢的计算机和浏览器上表现得尤其突出，与具有处理能力的浏览器相比，游戏可能尝试在这些计算机和浏览器上进行更频繁的游戏更新，最终导致速度明显下降。第二个缺点是，甚至在浏览器激活不同的标签页时，游戏也会继续全速运行，这既拖慢了计算机的速度，也害得 JavaScript 游戏恶名远扬。

从 2011 年开始，浏览器开始加入了对 requestAnimationFrame API 的支持，该 API 允许浏览器基于其更新屏幕的实际速度来控制游戏循环的调用频率。自该 API 首次推出以来，requestAnimationFrame 规范已趋稳定，所以，现在该 API 在支持该规范的各个浏览器是一致的。就那些不支持该 API 的浏览器而言，Paul Irish(在互联网上的其他一些人的帮助下)开发了一个腻子脚本(polyfill)，可在必要时把 requestAnimationFrame 支持反向移植回所有

使用 setTimeout 的浏览器。关于这一主题，你可参阅 Paul 在 2011 年撰写的博客文章，访问地址是 http://paulirish.com/2011/requestanimationframe-for-smart-animating/。

将代码清单 9-4 中的代码添加到 quintus.js 文件的头部(置于文件顶端的 Quintus 定义的外部)，这段代码可通过 windows 对象把一个一致的 requestAnimationFrame 方法暴露给所有浏览器。

代码清单 9-4：requestAnimationFrame 的腻子脚本

```
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x)
{
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] ||
window[vendors[x]+'CancelRequestAnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime +
timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
}());
```

如你所见，如果 requestAnimationFrame 的无厂商前缀的版本是不可用的，代码会遍历每个潜在的厂商特定前缀，然后把带有前缀的版本用作无前缀版本。若该做法失败，代码使用 setTimeout 和 cancelTimeout 来近似模拟 requestAnimationFrame 和 cancelAnimationFrame。因为不受浏览器的原生支持，这些腻子脚本方法同样存在之前提到的那些缺点，但它们已是所能用到的最好做法了。

9.3.2 将已优化的游戏循环添加到 Quintus

既已为已经优化的定时器方法创建了一个一致的腻子脚本，下一步要做的就是创建游戏循环本身。传统的游戏循环使用两大块代码来执行每一帧，它们分别是更新代码和渲染代码。更新部分负责推进游戏逻辑在一小段时间内的发展，处理任何的用户输入、移动和

对象间碰撞，以及把每个游戏对象更新成一致的状态。

然后，游戏需要把自身渲染到屏幕上，至于以什么样的步骤进行渲染，这取决于游戏的构建方式。就基于画布的游戏而言，通常你需要清除整块画布，重新在页面上绘制所有必需的精灵。而对于 CSS 和 SVG 游戏来说，只要正确更新了页面上的对象的属性，那么工作实际上就算完成了——浏览器会负责对象的移动和更新。

了解到这方面的知识后，就可以把游戏循环方法添加到 Quintus 中了，之外还可添加暂停游戏和取消暂停游戏的方法 pauseGame 和 unpauseGame。打开 quintus.js，将代码清单 9-5 中的代码添加到该文件中，置于最后的 return 语句之前。

代码清单 9-5：添加游戏循环

```
Q.gameLoop = function(callback) {
  Q.lastGameLoopFrame = new Date().getTime();

  Q.gameLoopCallbackWrapper = function(now) {
    Q.loop = requestAnimationFrame(Q.gameLoopCallbackWrapper);
    var dt = now - Q.lastGameLoopFrame;
    if(dt > 100) { dt = 100; }
    callback.apply(Q,[dt / 1000]);
    Q.lastGameLoopFrame = now;
  };

  requestAnimationFrame(Q.gameLoopCallbackWrapper);
};

Q.pauseGame = function() {
  if(Q.loop) {
    cancelAnimationFrame(Q.loop);
  }
  Q.loop = null;
};

Q.unpauseGame = function() {
  if(!Q.loop) {
    Q.lastGameLoopFrame = new Date().getTime();
    Q.loop = requestAnimationFrame(Q.gameLoopCallbackWrapper);
  }
}
```

Q.gameLoop 方法接收一个回调，该回调预期接收一个 dt 参数，该参数表示与上一帧之间的以秒为单位的时间差(这是一个很小的值，接近于 1 秒的 1/60)，Q.gameLoop 方法把该回调包装在一个方法中，该方法计算出上一帧时间与被传入 requestAnimationFrame 回调中的当前时间的差值。这个经过包装的回调被保存在 Q.gameLoopCallbackWrapper 中，Q.pauseGame 和 Q.unpauseGame 使用它来启动和停止游戏定时器。

9.3.3 测试游戏循环

现在，Quintus 已拥有足够的功能，至少可把它放在页面上，测试一下它的游戏循环以及 pauseGame 和 unpauseGame 代码。

创建一个名为 gameloop_test.html 的新文件并打开它，然后将代码清单 9-6 中的 HTML 填写到该文件中，确保所依赖的 jQuery 和 underscore.js 与此阶段的 quintus.js 代码位于相同的目录中。

代码清单 9-6：游戏循环测试 gameloop_test.html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title></title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
  </head>
  <body>
    <div id='timer'>0</div>
    <div id='fps'>0</div>
    <button id='pause'>Pause</button>
    <button id='unpause'>Unpause</button>
    <script>
      var TimerTest = Quintus();

      var totalFrames = 0,
          totalTime = 0;

      TimerTest.gameLoop(function(dt) {
          totalTime += dt;
          totalFrames += 1;
          $("#timer").text(Math.round(totalTime * 1000) + " MS");
          $("#fps").text(Math.round(totalFrames / totalTime) + " FPS");
      });

      $("#pause").on('click',TimerTest.pauseGame);
      $("#unpause").on('click',TimerTest.unpauseGame);
    </script>
  </body>
</html>
```

加载该页面应能执行游戏循环函数，并能使用两个按钮来暂停和取消暂停游戏。

这一练习中的游戏循环用于在两个全局变量中记录下总用时和总帧数，然后使用 jQuery 更新两个 ，以毫秒为单位显示循环已经运行的时间和动画每秒绘制的帧数。

该段代码使用 jQuery.fn.on 方法把 Q.pauseGame 和 Q.unpauseGame 方法绑定到按钮单

击事件上。Quintus 代码的构建做法带来了一个好的副作用，那就是 quintus.js 没有一处引用了 this 对象，代码始终通过局部变量 Q 来引用 Quintus 对象，该变量被绑定在一个闭包中。这意味着 JavaScript 最棘手的一个地方，即在任意给定时刻知道 this 指向什么对象这件事不会影响到 Quintus 的主体代码。因此，可将诸如 Q.pauseGame 和 Q.unpauseGame 一类的 Quintus 方法传入回调中，不必担心如何把它们绑定到它们的对象上。

在引入精灵时，这种类型的绑定就不能采用了，因为如第 2 章所述，在这种情况下，你应使用原型属性来节省内存和创建时间；不过，因为要创建的 Quintus 引擎实例只有为数不多的几个(通常每个页面只有一个)，所以可采用这种对象创建方法来把工作变得更容易一些。

9.4　添加继承

在过去，对象继承的做法被广泛用于大多数游戏引擎中。例如，动画精灵构建自活动精灵，活动精灵又构建自一些基础对象(Base Object)。这种继承层次结构的发展，部分是因为诸如 C++一类语言的静态特性，这种特性适于使用继承类和虚函数来同等对待不同类型的对象。

9.4.1　在游戏引擎中使用继承

随着游戏引擎在规模方面的增长，人们意识到静态的类层次结构很快会变得臃肿不堪。就算可以在类中使用一些浅层次结构来让对象共享相同的基本功能，但人为地创建一个很深的层次结构通常没有太大意义。

以一个射击类游戏为例，该游戏拥有一些可供玩家选择的不同武器，假设你有以下三种武器：

- 撬棍：只可用来击打他人，不配弹药。
- 手枪：可用作投射类武器，也可用作击打他人的近身类武器，配有数量有限的射击弹药。
- 手榴弹：只可用作投射类武器，不过能带来一定范围的杀伤效果。

即使是在这个简单例子中，要设计出一个既支持代码重用，同时要避免武器背负它们不需要的功能的层次结构也是不容易的。你可能会倾向于创建以下的基类集合：

```
Weapon
  MeleeWeapon
    RangedWeapon
      AreaDamageWeapon
```

虽然这不够完美——例如，手榴弹会继承自 AreaDamageWeapon，但需要重写所有的近身类武器的功能并禁用它——不过乍一看，这至少像是一个可行的基础。然而，继续往其中添加一些新的武器类型的话，层次结构很快就变得臃肿有余。试想一下，该如何处理

附带了榴弹发射器的步枪呢？哪种武器是有射程、可用于近战且有区域杀伤力的呢？诸如地雷一类的东西可被埋设在某个地方，在踩中时会带来区域性的杀伤效果(很像是一种近身类武器)，这类武器也是一个问题。在你开始过五关斩六将，努力维持类层次结构的正常使用时，很明显是时候转用别的做法了。

这一别的做法被称为组件/实体模型(component/entity model)，它的理念是，你不必通过定义一个线性的类层次结构来建立起所想要的功能层次，相反，你只需定义一些互不知道彼此存在的、只涉及自身业务的松耦合组件即可。例如，在之前的武器定义这一例子中，可以定义近身攻击(Melee Attacks)、远程攻击(Ranged Attacks)和区域破坏攻击(Area Damage Attacks)等几种独立组件，然后把这些组件的触发和不同的发射输入绑定在一起(例如，若武器已装备且发射按钮 1 被按下，则发起近身攻击，若发射按钮 2 被按下则发起远程攻击)。

组件的唯一缺点是它们往往会变得数量巨大，在创建对象时，你得加上一长串组件，以及要应对不知道哪些基础级功能获得所有对象支持这样的情形，这些事情可能会带来一些挑战。此外，组件也往往是独立的，这意味着在确实需要它们之间进行交互时，你会遇到另外一些问题。

为此，Quintus 同时支持继承和组件，采取了一种介于这两者之间的折中做法，这使得你既能在一些地方合理使用继承，又可在需要更多灵活性时使用组件。为了支持前者，引擎需要把一个传统继承模型添加到 JavaScript 中；为了支持后者，它需要增加一个组件系统，以及要拥有对事件的底层支持，这样才能做到尽可能支持组件之间的解耦。

9.4.2 将传统继承添加至 JavaScript

JavaScript 不存在受限于静态类型接口这样的问题，相反，它通常会利用鸭子类型的概念，这一之前已提到过的鸭子类型的中心思想是，对象的类型是无关紧要的，真正重要的是对象所响应的属性和方法。

JavaScript 确实支持一个原型继承模型，该模型启用了一种更传统的继承类型。使用开箱即用的原型继承会带来三个主要问题：它不支持通过超类型的方法调用继承得来的功能、创建子对象的过程稍有杂凑之感，以及无法继承构造函数。

可使用多种方法来解决这些问题，也可把一个更加传统的类层次结构添加到 JavaScript 中。一个最受欢迎的类层次结构是 jQuery 的创建者 John Resig 的 Simple JavaScript 继承，其灵感来自 base2 和另一个名为 Prototype.js 的 JavaScript 库。按照 John 在其网站 http://ejohn.org/blog/simple-javascript-inheritance/上的原始描述，这是一块开源代码，在 MIT 协议下进行发布。

将代码清单 9-7 中的代码添加到 quintus.js 顶部的某处，置于 Quintus 的构造函数方法的外部(靠近文件的顶端，紧跟在 requestAnimationFrame 代码的后面，置于任意花括号的外部就可以了)。

代码清单 9-7：Simple JavaScript 继承

```
/* Simple JavaScript Inheritance
```

```
 * By John Resig http://ejohn.org/
 * MIT Licensed.
 */
// Inspired by base2 and Prototype
(function(){
  var initializing = false,
      fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/;
  // The base Class implementation (does nothing)
  this.Class = function(){};

  // Create a new Class that inherits from this class
  Class.extend = function(prop) {
    var _super = this.prototype;

    // Instantiate a base class (but only create the instance,
    // don't run the init constructor)
    initializing = true;
    var prototype = new this();
    initializing = false;

    // Copy the properties over onto the new prototype
    for (var name in prop) {
      // Check if we're overwriting an existing function
      prototype[name] = typeof prop[name] == "function" &&
        typeof _super[name] == "function" &&
              fnTest.test(prop[name]) ?
        (function(name, fn){
          return function() {
            var tmp = this._super;

            // Add a new ._super() method that is the same method
            // but on the super-class
            this._super = _super[name];

            // The method only need to be bound temporarily, so we
            // remove it when we're done executing
            var ret = fn.apply(this, arguments);
            this._super = tmp;

            return ret;
          };
        })(name, prop[name]) :
        prop[name];
    }

    // The dummy class constructor
    function Class() {
      // All construction is actually done in the init method
      if ( !initializing && this.init )
        this.init.apply(this, arguments);
```

```
    }

    // Populate our constructed prototype object
    Class.prototype = prototype;

    // Enforce the constructor to be what we expect
    Class.prototype.constructor = Class;
    // And make this class extendable
    Class.extend = arguments.callee;

    return Class;
  };
})();
```

这段代码的主旨是允许使用 extend 方法从现有类中派生出新类，继承得来的对象可像父对象那样共享相同的实例方法，可以通过子方法的 this._ super ()调用父方法。这一特殊情况由代码中部的循环进行处理，该循环并非仅是一味地拷贝整个方法，它会检查父类的现有方法，然后创建一个包装器函数，在调用期间，该函数临时将 this._super 方法设置成父类的定义：

```
// Copy the properties over onto the new prototype
for (var name in prop) {

  // Check if we're overwriting an existing function
  prototype[name] = typeof prop[name] == "function" &&
    typeof _super[name] == "function" && fnTest.test(prop[name]) ?
    (function(name, fn){
      return function() {
        var tmp = this._super;

        // Add a new ._super() method that is the same method
        // but on the super-class
        this._super = _super[name];

        // The method only need to be bound temporarily, so we
        // remove it when we're done executing
        var ret = fn.apply(this, arguments);
        this._super = tmp;

        return ret;
      };
    })(name, prop[name]) :
    prop[name];
}
```

上述代码检查属性是否已存在于超类中，若是，则创建一个函数，在再次调用新方法之前，该函数会先对 this ._ super 执行一个临时的重新赋值。若该方法不存在，代码仅对属性赋值，不会加入任何额外开销。

Class 代码还加入了一个构造函数，该函数自动调用对象的 init()方法，还支持初始化函数的链式调用：

```
// The dummy class constructor
function Class() {
  // All construction is actually done in the init method
  if ( !initializing && this.init )
    this.init.apply(this, arguments);
}
```

最后把 extend 方法添加到类中，这样类就可被进一步子类化：

```
// And make this class extendable
Class.extend = arguments.callee;
```

调用 arguments.callee 时，会返回被调用的方法(在该例中是 extend)，然后把该方法赋给所返回的 Class 对象的属性 extend，以此来支持进一步的子类化。

9.4.3　运用 Class 的功能

要增加对使用该代码的感性认识，可试着在浏览器的控制台中使用该功能，代码如下：

```
var Person = Class.extend({
  init: function() { console.log('Created Person'); },
  speak: function() { console.log('Person Speaking:'); }
});

var p = new Person();
// Logs: Created Person

p.speak();
// Logs: Person Speaking:

var Guy = Person.extend({
  init: function() { this._super(); console.log('Created Guy'); },
  speak: function() { this._super(); console.log("I'm a Guy!"); }
});

var bob = new Guy();
// Logs: Created Person
//       Created Guy

bob.speak();
// Logs: Person Speaking
//       I'm a Guy!

// Girl doesn't call the super method
var Girl = Person.extend({
  init: function() { console.log('Created Girl'); },
  speak: function() { console.log("I'm a Girl!"); }
```

```
});

var jill = new Girl();
// Logs: Created Girl

jill.speak();
// Logs: I'm a Girl!
```

如你所见，类功能使得以一种清晰的方式创建并扩展类成为可能，这些子类确保超类的方法能在适当时候被调用。

```
bob instanceof Person;   // true
bob instanceof Guy;      // true
bob instanceof Girl;     // false
jill instanceof Person;  // true
jill instanceof Guy;     // false
jill instanceof Girl;    // true
```

该功能也简化了必要时对于对象类型的使用，可以注意到，bob 和 jill 都如预期般响应了 instanceof 运算符。

9.5 支持事件

在游戏引擎中添加对事件的支持，这更便于避免引擎的不同部分过度紧密耦合。这意味着，在不必了解与游戏进行通信的对象的任何信息的情况下，游戏的一部分和其他部分之间能够就事件和行为进行沟通。

在把一些组件添加到这一混搭环境中后，它甚至允许精灵在不必了解构成自身的所有组件的情况下与自身通信。精灵的某个物理组件可能会触发一个碰撞事件，而两个负责监听该事件的组件则可以分别处理适当音响效果和动画效果的触发过程。

9.5.1 设计事件 API

Quintus 使用了一个名为 Evented 的基类，这是任何需要订阅和触发事件的对象的起点。像往常一样，你必须先考虑 API，然后围绕该 API 来构建代码。

给定一个 player 精灵和一个 scene 对象，现在来讲解一个事件功能例子：

```
// Play the intro animation on the player
// when the scene starts
scene.bind('start',player,function() {
  this.showIntro();
});

// Bind a method on player using the method name
scene.bind('finish',player,'showFinal');

// Trigger the start event on the scene
```

```
scene.trigger('start');

// Unbind the player from the start event
scene.unbind('start',player);

// Release the player from listening
// to all events (such as if it's blown up)
player.debind();
```

该 API 提供了一种方法来绑定、触发和解除绑定事件，以及从所有事件中释放对象(如从游戏中删除对象时)，这样已被销毁的精灵就不会继续响应事件了。

9.5.2　编写 Evented 类

如你所见，Evented 基类需要支持四个方法：bind、unbind、trigger 和 debind。打开 quintus.js，将代码清单 9-8 中的 Q.Evented 定义添加至 gameLoop 代码的后面(但要置于最后的 return 语句之前)。在接下来的几节中，你将依次填写其中的每个方法调用。

代码清单 9-8：Evented 的代码大纲

```
Q.Evented = Class.extend({
  bind: function(event,target,callback) {
    // TODO: Fill in bind code
  },

  trigger: function(event,data) {
    // TODO: Fill in trigger code
  },

  unbind: function(event,target,callback) {
    // TODO: Fill in unbind code
  },

  debind: function() {
   // TODO Fill in the debind code
  }
});
```

为保证 Evented 类的子类化过程的简单性，该类未使用充当构造函数的 init 方法，但会在必要时即时初始化任何对象。

9.5.3　填写 Evented 方法

先来分析 bind 方法，该方法的任务是把监听器绑定到某个特定事件上以及设定目标事件触发时的回调。目标是一个可选参数，该参数提供了回调的上下文，并且允许回调被删除，在调用目标的 debind 以防过时事件出现时就可以删除回调。将代码清单 9-9 中的代码填写到 bind 方法中。

代码清单 9-9：Evented 的 bind 方法

```
bind: function(event,target,callback) {
  // Handle the case where there is no target provided
  if(!callback) {
    callback = target;
    target = null;
  }
  // Handle case for callback that is a string
  if(_.isString(callback)) {
    callback = target[callback];
  }

  this.listeners = this.listeners || {};
  this.listeners[event] = this.listeners[event] || [];
  this.listeners[event].push([ target || this, callback]);
  if(target) {
    if(!target.binds) { target.binds = []; }
    target.binds.push([this,event,callback]);
  }
},
```

就基本面来说，bind 方法相对简单：唯一紧要之处是中间部分把监听器添加到对象中的三行代码，该对象名为 this.listeners，使用事件名称作为键。每个监听器包含一个两元素数组，该数组由一个上下文对象和回调本身构成。这段代码首先检查 this.listeners 数组是否存在，因为 Evented 没有用来充当构造函数的 init 方法。

剩余代码贯彻了 Quintus 的开发者友好目标，支持多种不同的输入格式，bind 方法可通过三种不同方式进行调用：

```
scene.bind('start',function() { ... });
scene.bind('start',player,function() { ... });
scene.bind('start',player,'methodName');
```

在不必考虑上下文时，或在对象被从游戏中删除后不必考虑对象自身的解除绑定时，可使用第一种签名(signature)。后面两种提供了相同的结果，不过一种接收目标对象的方法属性的名称字符串作为参数，另一种接收方法本身作为参数。

接下来考虑 trigger 方法，这是四个方法中最简单的一个。使用代码清单 9-10 中的代码填写该方法(别忘了用逗号分割这四个方法的定义)。

代码清单 9-10 Evented 的 trigger 方法

```
trigger: function(event,data) {
  if(this.listeners && this.listeners[event]) {
    for(var i=0,len = this.listeners[event].length;i<len;i++) {
      var listener = this.listeners[event][i];
      listener[1].call(listener[0],data);
    }
```

```
    }
  },
```

trigger 仅查看是否有监听器在监听该特定事件，若是，则遍历每个监听器，使用所提供的上下文调用回调。因为每个监听器由一个存放了上下文和回调的数组构成，所以实际发起调用的代码如下：

```
listener[1].call(listener[0],data);
```

鉴于 JavaScript 中的上下文和 this 对象这些概念的灵活性，你应始终明确指出方法调用的上下文。

因为事件已被绑定，是可触发的，所以在对象已被销毁或不再需要触发对象的某些特定事件时，可解除这些事件的绑定。将代码清单 9-11 中的代码填写到 unbind 方法中。

代码清单 9-11：Evented 的 unbind 方法

```
unbind: function(event,target,callback) {
  if(!target) {
    if(this.listeners[event]) {
      delete this.listeners[event];
    }
  } else {
    var l = this.listeners && this.listeners[event];
    if(l) {
      for(var i = l.length-1;i>=0;i--) {
        if(l[i][0] == target) {
          if(!callback || callback == l[i][1]) {
            this.listeners[event].splice(i,1);
          }
        }
      }
    }
  }
},
```

unbind 方法可以接收一个、两个或三个参数，每一种参数形式都比前一种提供了更具体的被删除事件信息。第一种情况没有提供目标，对象通过简单地删除 listeners 对象的键来删除事件的整个监听器列表。

第二种和第三种情况只解除绑定了该事件所有可能的监听器中的一个或几个。实际上，该方法需要遍历每个监听器，然后用内置的 Array.splice 方法来删除那些被解除绑定的监听器。遍历数组的同时从数组中删除元素是项较为棘手的工作，因为若按正常方式遍历数组然后修改它的长度，那么最终可能遇到问题。一个绕开这一问题的做法是从数组末端往回遍历至开始处，若需要删除某个元素，那么该操作也不会影响数组中排在它前面的元素的索引值。

Evented 的最后一个方法是 debind 方法，该方法从所有注册了某个对象的监听器中删

除该对象。unbind 用来删除某个对象的一些监听器，而 debind 则用来在销毁对象时删除其所有的监听器，以防发生内存泄漏和意外行为。将代码清单 9-12 中的代码填写到 debind 方法中。

代码清单 9-12：Evented 的 debind 方法

```
debind: function() {
  if(this.binds) {
    for(var i=0,len=this.binds.length;i<len;i++) {
      var boundEvent = this.binds[i],
          source = boundEvent[0],
          event = boundEvent[1];
      source.unbind(event,this);
    }
  }
}
```

这段代码遍历 this.binds 数组中的每个元素，然后调用 unbind 来删除它们。

9.6 支持组件

需要用来自建 Quintus 的最后一部分核心功能是加入组件支持，如前面 9.4.1 一节所述，组件简化了小块可重用功能的创建，这些可重用功能可经组合和匹配来满足各种精灵和对象的需要。

第 26 章将介绍的 Crafty.js 是一个成熟的、很受欢迎的 HTML5 游戏引擎，该引擎完全基于组件-实体架构，它也是 Quintus 组件方法的灵感来源。

9.6.1 设计组件 API

一如往常，先考虑 API 及你打算如何在游戏中使用组件。精灵组件的增加和删除操作必须简捷，它们应是可通过对象访问的，但又不能过分污染对象的名称空间。代码清单 9-13 给出的例子说明了如何定义和使用组件。

代码清单 9-13：设想的组件系统

```
var exGame = Quintus();
var player = new exGame.GameObject();
exGame.register('sword',{
 added: function() {
  // When whatever we are registered with triggers
  // a fire event, call the attack method
  this.entity.bind('fire',this,'attack');
 },
 attack: function() {
  // Code to attack
 },
```

```
// Methods copied directly over to the entity
extend: {
  attack: function() {
   this.sword.attack();
  }
}

});
// Add the sword component
player.add('sword');

// Calls attack via event
player.trigger('fire');

// Call attack directly from extended event
player.attack();

// Remove the sword component
player.del('sword');

// Should cause an error
player.attack();
```

Quintus 中的组件系统需要注册组件、添加组件、删除组件，还允许使用其他一些方法来扩充基本精灵。

9.6.2　实现组件系统

组件系统的真正实现分为三个独立部分，第一部分是用来注册组件的注册方法，注册功能由 Q.register 方法处理，在该方法的内部，它创建了一个新的组件类并把它存储在 Q.components 中，以类的名称为索引。

照常将代码清单 9-14 中的代码添加到 quintus.js 的末尾处，放在最后的 return 语句之前。

代码清单 9-14：基本的 Quintus 组件功能

```
Q.components = {};

Q.register = function(name,methods) {
  methods.name = name;
  Q.components[name] = Q.Component.extend(methods);
};
```

这段代码在 Q.components 对象中注册组件，并用传进来的方法扩展 Q.Component 类。下一步创建 Q.Component 类，该类处理一些繁重工作，包括把自身添加到对象中以及从对象中删除自身。

将代码清单 9-15 中的代码添加到刚才添加的代码之后，用以创建 Q.Component 类。

代码清单 9-15：Q.Component 类

```
Q.Component = Q.Evented.extend({
    init: function(entity) {
      this.entity = entity;
      if(this.extend) _.extend(entity,this.extend);
      entity[this.name] = this;
      entity.activeComponents.push(this.name);
      if(this.added) this.added();
    },

    destroy: function() {
      if(this.extend) {
        var extensions = _.keys(this.extend);
        for(var i=0,len=extensions.length;i<len;i++) {
          delete this.entity[extensions[i]];
        }
      }
      delete this.entity[this.name];
      var idx = this.entity.activeComponents.indexOf(this.name);
      if(idx != -1) {
        this.entity.activeComponents.splice(idx,1);
      }
      this.debind();
      if(this.destroyed) this.destroyed();
    }
  });
```

这一组件基类只负责两件主要事情：处理被添加至某个实体中及被从某个实体中删除的事务。将组件添加到某个实体中时(这在充当构造函数的 init 中实现)，组件完成了以下 5 件事情：

(1) 设置一个属性以便能往回引用该实体。

(2) 使用来自其 extend 特性的新属性来扩充实体。

(3) 将自身作为属性添加至实体中，将其名称为属性的名称(所以，如 sword 组件可由 entity.sword 访问)。

(4) 它还把自身添加至实体的活动组件列表中。

(5) 它调用组件的 added 方法来设置诸如监听器一类的任何后初始化(post-initialization)需求。

Q.Component 类继承自 Q. Evented 对象，所以它可以绑定也可以被绑定。

在组件被销毁时，它需要做一些相反的事情，这就等于要从实体中删除所有的扩展，做法是删除与扩展对象的键匹配的属性；接着，它需要从实体中删除以组件命名的属性，以及从活动组件列表中删除该组件条目；最后，它调用 debind 来删除已绑定的所有事件处理程序，若存在自定义的 destroyed()处理程序，它还需要调用该处理程序。

组件系统的最后一块内容是 Q.GameObject 类，该类继承自 Q.Evented，负责添加和删

除组件。Q.GameObject 类是所有活动的游戏对象的基类，诸如精灵一类的对象都继承自该类。

将代码清单 9-16 中的 Q.GameObject 定义添加到 quintus.js 的末尾处，置于最后的返回语句之前。

代码清单 9-16：Q.GameObject 的定义

```
Q.GameObject = Q.Evented.extend({
    has: function(component) {
      return this[component] ? true : false;
    },
    add: function(components) {
      components = Q._normalizeArg(components);
      if(!this.activeComponents) { this.activeComponents = []; }
      for(var i=0,len=components.length;i<len;i++) {
        var name = components[i],
            comp = Q.components[name];
        if(!this.has(name) && comp) {
          var c = new comp(this);
          this.trigger('addComponent',c);
        }
      }
      return this;
    },
    del: function(components) {
      components = Q._normalizeArg(components);
      for(var i=0,len=components.length;i<len;i++) {
        var name = components[i];
        if(name && this.has(name)) {
          this.trigger('delComponent',this[name]);
          this[name].destroy();
        }
      }
      return this;
    },

    destroy: function() {
      if(this.destroyed) { return; }
      this.debind();
      if(this.parent && this.parent.remove) {
        this.parent.remove(this);
      }
      this.trigger('removed');
      this.destroyed = true;
    }
  });
```

这一 Q.GameObject 基类同样继承自 Q.Evented，支持事件的监听和触发。该类的代码包含了 4 个主要方法：add、has、del 和 destroy，前三个分别用来添加、检查和删除对象的

组件；最后一个方法 destroy 用来删除对象自身。

先说一下 has 方法，该方法检查 Q. GameObject 对象是否已拥有某个特定组件，做法是检查对象是否有一个同名属性。这种做法存在一些风险，因为这依赖于开发者对组件名称和扩展属性的谨慎使用，若存在名称冲突，那么不管情况如何，肯定会有其他一些问题随之发生。

接下来是 add 和 del 方法，这两个方法增加和删除 Q.GameObject 对象的组件，它们几乎为彼此的镜像。add 方法遍历所有要添加的组件，在 Q.components 中查找它们，然后创建新的组件对象；del 方法做相反的事情，遍历要删除的组件，并为每个组件调用其 remove 方法。

这两个方法都使用了一些附加逻辑来防止开发者多次添加或删除同一组件，它们也都把组件实例本身作为数据参数进行传递，以此来触发一个事件。

destroy 方法调用对象的 debind 方法，若对象有父类的话，则从父类中删除它；接着，它触发一个 removed 事件来允许任何组件进行必要的清除工作；此外，它还添加了一个 destroyed 属性来防止 destroy 方法被多次调用。

9.7 小结

现在，你已经拥有了一些用来创建可重用的 HTML5 游戏引擎的构建块，你创建了最初的游戏容器对象、游戏循环以及一些使用事件和组件的基类。这些工作将给后面以一种模块化方式构建引擎的剩余部分带来极大帮助，有了这样一个坚实基础，下一章接着讨论一些游戏元素到页面的加载，以及一个可重用用户输入处理系统的构建。

第10章

自建 Quintus 引擎(2)

本章提要

- 捕捉输入
- 绘制屏幕控件
- 加载资产

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 10 章的下载压缩中，代码文件的名称分别依照本章各处使用的文件名称命名。

10.1 引言

上一章中已经为可重用引擎埋下了种子，本章将继续充实这一基础，使用类在屏幕上绘制控件、接受用户的输入以及加载资产。

10.2 访问游戏容器元素

就游戏而言，要在屏幕上渲染任何东西，它必须有一个可供其在上面进行绘制的对象。对于画布游戏来说，该对象就是画布(Canvas)元素；而对于其他类型的游戏来说，它或是一个普通的 或是一个 SVG 元素。为此，需要为 Quintus 创建一个灵活的设置(setup)方法，该方法通过传入的 ID 抓取一个容器，或在必要时从头创建一个元素。若元素支持

上下文，该设置方法还会从元素中提取上下文。

将代码清单 10-1 中的代码添加到第 9 章的 quintus.js 的末尾处，置于最后的 return Q 语句之前。

代码清单 10-1：Quintus 的 setup 和 clear 方法

```
Q.setup = function(id, options) {
  var touchDevice = 'ontouchstart' in document;
  options = options || {};
  id = id || "quintus";
  Q.el = $(_.isString(id) ? "#" + id : id);

  if(Q.el.length === 0) {
    Q.el = $("<canvas width='320' height='420'></canvas>")
            .attr('id',id).appendTo("body");
  }

  var maxWidth = options.maxWidth || 5000,
      maxHeight = options.maxHeight || 5000,
      resampleWidth = options.resampleWidth,
      resampleHeight = options.resampleHeight;

  if(options.maximize) {
    $("html, body").css({ padding:0, margin: 0 });
    var w = Math.min(window.innerWidth,maxWidth);
    var h = Math.min(window.innerHeight - 5,maxHeight)

    if(touchDevice) {
      Q.el.css({height: h * 2});
      window.scrollTo(0,1);

      w = Math.min(window.innerWidth,maxWidth);
      h = Math.min(window.innerHeight - 5,maxHeight);
    }

    if(((resampleWidth && w > resampleWidth) ||
        (resampleHeight && h > resampleHeight)) &&
       touchDevice) {
      Q.el.css({  width:w, height:h })
          .attr({ width:w/2, height:h/2 });
    } else {
      Q.el.css({  width:w, height:h })
          .attr({ width:w, height:h });
    }

  }

  Q.wrapper = Q.el
              .wrap("<div id='" + id + "_container'/>")
```

```
                .parent()
                .css({ width: Q.el.width(),
                       margin: '0 auto' });

  Q.el.css('position','relative');

  Q.ctx = Q.el[0].getContext &&
          Q.el[0].getContext("2d");


  Q.width =  paraseInt(Q.el.attr('width'),10);
  Q.height = parseInt(Q.el.attr('height'),10);

  $(window).bind('orientationchange',function() {
    setTimeout(function() { window.scrollTo(0,1); }, 0);
  });

  return Q;
};


Q.clear = function() {
  Q.ctx.clearRect(0,0,Q.el[0].width,Q.el[0].height);
};
```

setup 方法的代码执行的是一项简单任务，但其中也有些复杂之处，因为它要尽力顾及不同的使用模式，即可分别以无参数形式、元素的 id 或是元素自身作为参数对其加以调用。若无 id 被传入，引擎默认使用 quintus 这一字符串作为 id；若无法找到一个元素，它创建一个新的<canvas>元素。接着，它创建一个用来在页面上居中显示元素的包装器元素，该包装器元素的默认大小是 320x420，足以填满 iPhone 的屏幕。最后，它检查对象是否有一个 getContext 方法，若找到一个，则提取出 2d 上下文并把它保存在 Q.ctx 中以备后用。

该方法还接受一个选项哈希作为参数，该哈希使用一个可选的 maximize 选项来重新调整主元素的大小，以让它与屏幕相匹配。此外，针对触摸设备，第 6 章中介绍的去除地址栏的屏幕最大化技术也已被添加到这里。在进行最大化时，方法还接受可选的 maxWidth、maxHeight、resampleWidth 和 resampleHeight 等参数，前两个参数设置游戏的最大宽度和高度，后两个参数设置可选的宽度和高度，移动设备根据这两个参数来折半重新采样。

该方法仍返回 Q 对象，允许必要时链式发起进一步的 Quintus 调用。

clear 方法较为简单，仅是清除整块画布。

要试用这个简单功能，创建一个名为 canvas_test.html 的 HTML 文件并打开它，将代码清单 10-2 中的 HTML 代码放入其中。

代码清单 10-2：画布测试例子

```
<!DOCTYPE HTML>
<html lang="en">
```

```
  <head>
    <title>Canvas Test</title>
    <meta charset="UTF-8">
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
  </head>
  <body>
    <script>
      var Q = Quintus().setup();
      Q.el.css('backgroundColor','red');
    </script>
  </body>
</html>
```

该例子仅创建一个新的 Quintus 实例，然后把创建好的画布元素的背景颜色改成红色，以此来指明元素在页面上的位置。若在浏览器中加载该例子，你应会看到一个红色、瘦长的画布元素从上到下出现在浏览器的中间位置上。若把设置行代码修改成如下内容

```
var q = Quintus().setup("quintus", { maximize: true });
```

那么你应既能够在桌面上也能够在移动设备上最大化这块红色区域。

10.3 捕捉用户输入

实际上，你不太可能去制造这样的游戏，即游戏没有给玩家提供任何与之进行交互的方式。一如第 1 到第 3 章构建的游戏 Alien Invasion，遵照与其相同的模式，Quintus 处理来自键盘和触摸界面的用户输入。在开发过程中以及玩家在桌面上玩游戏时，支持键盘输入能带来很大的帮助。

10.3.1 创建输入子系统

实现输入的最简单做法是直接把某些动作(例如按下右箭头按键)绑定到游戏的某种行为上，例如向右移动玩家这样的行为。不过，在将更多输入选项添加到游戏中之后，这种机制所存在的问题就会暴露出来。

在游戏需要既能在移动设备上又能在桌面上运行的情况下，你至少得准备两套输入机制：键盘或鼠标，以及触摸。从真正生成输入和给引擎提供一致接口的那部分代码中抽离游戏的逻辑，这种做法能把游戏开发变得更容易一些。

Quintus 使用一个名为 Quintus.Input 的模块来处理输入，保证了输入代码与 Quintus 引擎其余部分之间的独立，这有助于防止依赖性，以及更便于在必要时替换不同的输入引擎。

输入模块最终支持五种不同的输入机制：

- 键盘(桌面输入)
- 鼠标(桌面输入)

- 直接操纵(触摸输入)
- 游戏小键盘(keypad)(触摸输入)
- 游戏手柄(joypad)(触摸输入)

很显然，键盘和鼠标都不必额外说明；直接操纵指的是直接移动屏幕上的东西的能力(回顾一下游戏愤怒小鸟(Angry Birds)中的拉开并弹出弹弓这一操作)。小键盘输入是指屏幕按钮的使用，这与游戏 Alien Invasion 中使用的按钮是一样的；游戏手柄输入指的是那种很小的、模拟风格的触摸板，其用法类似于一个四向控制器，既可当成(提供了强度和角度的)模拟控制器使用，也可当成(提供了四个离散方向的移动的)数字控制器使用。

本章将构建键盘、小键盘和游戏手柄的输入，第 14 章会谈及直接操纵和鼠标输入。

10.3.2　自建输入模块

输入模块的主要目标是绑定某种输入动作，这或是键盘键的按下，或是屏幕小键盘按钮的按压，或是特定于诸如 left(向左)或 right(向右)一类动作的游戏手柄移动。动作的触发方式应是无关紧要的，游戏看到的应是同一种输入流。

游戏以两种方式接收输入，第一种是查找 Q.inputs 对象：

```
Q.inputs['fire'] // true if fire is being held down
```

每个被绑定的动作都有一个对象键，若按键当前被按下，则相应对象键的值被设成 true。这对移动来说非常有用，因为你会希望在绘制每帧时查看用户是否在移动，若是，则适当地更新他们的位置。

动作的触发是通过特定按压实现的，这并非仅是按住按钮不放这么简单。输入模块继承了 Evented 来支持监听器的绑定。例如：

```
Q.input.bind('fire',function() {
  console.log("Fire pressed");
});

Q.input.bind('fireUp',function() {
  console.log("Fire released");
});
```

由于同时支持两种方法，所以能够给游戏的步进(step)代码带来一些帮助，这样的话，这部分代码就不必在绘制每帧时都要轮询每个输入来检查变化。

每种输入方法都可对被触发事件进行配置，不过，一些默认配置已可处理大多数常见情况，所以不需要完成太多配置。

若要开始编写该模块，在 quintus.js 所在目录下创建一个新的名为 quintus_input.js 的文件并打开它，然后将代码清单 10-3 中的代码加入其中。

代码清单 10-3：基本的 Quintus.Input 模块

```
Quintus.Input = function(Q) {
```

```
var KEY_NAMES = { LEFT: 37, RIGHT: 39, SPACE: 32,
                  UP: 38, DOWN: 40,
                  Z: 90, X: 88
                };

var DEFAULT_KEYS = { LEFT: 'left', RIGHT: 'right',
                     UP: 'up',     DOWN: 'down',
                     SPACE: 'fire',
                     Z: 'fire',
                     X: 'action' };

var DEFAULT_TOUCH_CONTROLS  = [ ['left','<' ],
                            ['right','>' ],
                            [],
                            ['action','b'],
                            ['fire', 'a' ]];

// Clockwise from midnight (a la CSS)
var DEFAULT_JOYPAD_INPUTS =  [ 'up','right','down','left'];

Q.inputs = {};
Q.joypad = {};

var hasTouch =  !!('ontouchstart' in window);

Q.InputSystem = Q.Evented.extend({
 // TODO: Fill in Input System code
});

Q.input = new Q.InputSystem();

};
```

如你所见，这段代码设置了四个用于输入模块的常量风格变量，第一个变量KEY_NAMES 是一个便捷的数组，该数组把输入键码和更加开发者友好的名称一一对应起来。代码清单 10-3 只为箭头键、空格键及 Z 和 X 按键定义了码值，因为这些是本书用到的按键，可以根据需要来添加另外一些键码。

其余三个数组定义了键盘、小键盘触摸控件和游戏手柄默认的输入动作绑定。

接下来是 Q.inputs 和 Q.joypad 对象，这两个对象保存动作输入的当前状态，它们被初始化成空对象；另外还有一个布尔变量 hasTouch，该变量用来检查浏览器是否支持触摸事件。

最后被添加到代码中的是 InputSystem 类的一个存根，该类继承自 Evented，所以，如代码清单所示，对象可以绑定到输入事件。

10.3.3　处理键盘事件

键盘事件是最便于处理的部分，这部分代码与第 1 章中的代码十分相似。可用代码清单 10-4 中的代码填写代码清单 10-3 中 Q.InputSystem 的 TODO 部分。

代码清单 10-4：键盘事件

```
Q.InputSystem = Q.Evented.extend({
  keys: {},
  keypad: {},
  keyboardEnabled: false,
  touchEnabled: false,
  joypadEnabled: false,

  bindKey: function(key,name) {
    Q.input.keys[KEY_NAMES[key] || key] = name;
  },

  keyboardControls: function(keys) {
    keys = keys || DEFAULT_KEYS;
    _(keys).each(function(name,key) {
     this.bindKey(key,name);
    },Q.input);
    this.enableKeyboard();
  },

  enableKeyboard: function() {
    if(this.keyboardEnabled) return false;

    // Make selectable and remove an :focus outline
    Q.el.attr('tabindex',0).css('outline',0);

    Q.el.keydown(function(e) {
      if(Q.input.keys[e.keyCode]) {
        var actionName = Q.input.keys[e.keyCode];
        Q.inputs[actionName] = true;
        Q.input.trigger(actionName);
        Q.input.trigger('keydown',e.keyCode);
      }
      e.preventDefault();
    });

    Q.el.keyup(function(e) {
      if(Q.input.keys[e.keyCode]) {
        var actionName = Q.input.keys[e.keyCode];
        Q.inputs[actionName] = false;
        Q.input.trigger(actionName + "Up");
        Q.input.trigger('keyup',e.keyCode);
      }
      e.preventDefault();
```

```
    });
    this.keyboardEnabled = true;
  },
```

键盘控件包括三个方法：

- bindKey 负责通过在 Q.input.keys 对象中设置一个值将键码(被按下的键的数字标识，而非它的 ASCII 表示)绑定到特定动作上，它接受 KEY_NAMES 数组中的键名(如 LEFT)或直接键码。
- keyboardControls 的实际作用是启用游戏的键盘控件，它针对每个键调用 bindKey 方法，以此来绑定所有传进来的键(或使用代码清单 10-4 中定义的 DEFAULT_KEYS)；然后调用 enableKeyboard，该方法实现浏览器事件的绑定。
- enableKeyboard 定义元素的 keydown 和 keyup 事件，为将<canvas>元素变成可选的又要禁用任何的轮廓焦点，方法首先要添加 tabindex 属性，然后把轮廓(outline)样式设为 0。

接下来，keydown 事件查找与键码相关的动作，若存在，则把该输入动作设成 true。它还触发一个与输入动作同名的事件，然后通过调用 e.preventDefault()来防止发生任何默认的浏览器行为。

keyup 事件所做的事情刚好相反，它将动作设成 false，并触发一个动作名称加上“Up”后缀之后与其名称匹配的事件。

10.3.4 添加小键盘控件

接下来是小键盘控件，这也是一个已在第 3 章中构建的版本。这些控件的目标是把一些不同按钮添加到屏幕底部。按钮比键盘事件要难处理一些，这是因为你不能直接把 touchstart 和 touchend 事件与小键盘的按压直接对应起来。若这样处理，用户就无法如他们经常会做的那样在屏幕上滑动手指了。

相反，Quintus 利用了每个触摸事件所包含的当前所有屏幕触摸及任何已发生改变的触摸，每次只要有事件发生，它就遍历这些触摸事件，且不管发生的是什么事件，它都会更新按下的动作。

虽然这一机制需要用到更多内务代码在键被按下或释放时触发事件，但你会得到这样的一种行为，即它既模拟了键的按压，又允许用户滑动他们的手指。

引擎假设控件的绘制横跨了屏幕的底部，应是覆盖了整个设备的。因此，使用控件数组中的条目个数来计算控件的尺寸，数组有五个条目意味着每个控件可以占据五分之一屏幕，但要减去控件间的边距大小。该控件数组允许有空白条目，这些条目代表不应触发事件的空余位置。

将代码清单 10-5 中的代码添加到 Q.InputSystem 定义的内部，放在键函数的后面，即置于之前添加的代码清单 10-4 的代码之后。

代码清单 10-5：触摸控件

```
touchLocation: function(touch) {
  var el = Q.el,
      pageX = touch.pageX,
      pageY = touch.pageY,
      pos = el.offset(),
      touchX = (el.attr('width') || Q.width) *
               (pageX - pos.left) / el.width(),
      touchY = (el.attr('height')||Q.height) *
               (pageY - pos.top) / el.height();
  return { x: touchX, y: touchY };
},

touchControls: function(opts) {
  if(this.touchEnabled) return false;
  if(!hasTouch) return false;

  Q.input.keypad = opts = _({
    left: 0,
    gutter:10,
    controls: DEFAULT_TOUCH_CONTROLS,
    width: Q.el.attr('width') || Q.width,
    bottom: Q.el.attr('height') || Q.height
  }).extend(opts||{});

  opts.unit = (opts.width / opts.controls.length);
  opts.size = opts.unit - 2 * opts.gutter;

  function getKey(touch) {
    var pos = Q.input.touchLocation(touch);
    for(var i=0,len=opts.controls.length;i<len;i++) {
      if(pos.x < opts.unit * (i+1)) {
        return opts.controls[i][0];
      }
    }
  }

  function touchDispatch(event) {
    var elemPos = Q.el.position(),
        wasOn = {},
        i, len, tch, key, actionName;

    // Reset all the actions bound to controls
    // but keep track of all the actions that were on
    for(i=0,len = opts.controls.length;i<len;i++) {
      actionName = opts.controls[i][0];
      if(Q.inputs[actionName]) { wasOn[actionName] = true; }
      Q.inputs[actionName] = false;
    }
```

```
    for(i=0,len=event.touches.length;i<len;i++) {
      tch = event.touches[i];
      key = getKey(tch);

      if(key) {
        // Mark this input as on
        Q.inputs[key] = true;

        // Either trigger a new action
        // or remove from wasOn list
        if(!wasOn[key]) {
          Q.input.trigger(key);
        } else {
          delete wasOn[key];
        }
      }
    }

    // Any remaining were on the last frame
    // and need to trigger an up action
    for(actionName in wasOn) {
      Q.input.trigger(actionName + "Up");
    }

    return null;
  }

  Q.el.on('touchstart touchend touchmove touchcancel',function(e) {
    touchDispatch(e.originalEvent);
    e.preventDefault();
  });

  this.touchEnabled = true;
},

disableTouchControls: function() {
  Q.el.off('touchstart touchend touchmove touchcancel');
  this.touchEnabled = false;
},
```

代码清单 10-5 中的代码定义了三个顶层方法：touchLocation、touchControls 和 disableTouchControls，touchLocation 用来找出<canvas>元素(或非<canvas>元素)上的正确像素位置，即使画布已被重新调整成一种不同于其像素尺寸的尺寸(这可通过 CSS 样式实现)也可实现这种查找。为将触摸代码用在基于 DOM 的游戏上，代码在主元素 Q.el 没有 width 特性时(因为在基于 DOM 的游戏中该特性不存在)使用 Q.width 变量。从性能角度看，这段代码可通过缓存 jQuery 调用(而非在每次调用时计算它们)来加以改进，不过为了保持代码的简单性和在事件更换方面的弹性，这里把这一改进留作一道练习题，由你来完成。

作为主要的设置方法，touchControls 在其内部进一步定义了另外两个方法，它们完成大部分处理小键盘事件的实际工作。

(1) getKey 调用 touchLocation 获得画布上的像素位置，然后返回对应于该元素的小键盘按钮(若有的话)。

(2) touchDispatch 在每次有触摸事件发生时被调用，该方法主要由三个循环组成：

- 第一个循环将 Q.inputs 数组中的所有条目的值设成 false，目的是取消所有小键盘绑定的输入的标志，同时在 wasOn 对象中记下所有设置了标志的输入。
- 第二个循环遍历设备上当前发生所有的触摸，然后把它们映射成小键盘的按压，并设置适当的输入标志。若之前尚未设置输入标志，该循环触发 Q.input 的事件；否则，从 wasOn 数组中删除该动作。
- 最后一个循环遍历所有在 touchDispatch 被调用之前设置了标志的输入，然后触发适当的按钮松开事件。

touchControls 的最后一部分绑定所有触摸事件——touchstart、touchend、touchmove 和 touchcancel——来调用 touchDispatch。

最后一个方法 disableTouchControls 简单地删除元素的事件处理程序，以此来禁用屏幕小键盘；若刚好设置了游戏手柄，它也会把游戏手柄标识为不可用的。

10.3.5　添加游戏手柄控件

最后要支持的输入机制是游戏手柄(joypad)，该机制也是最复杂的一种，这主要是因为它需要既像一个模拟器又像一个数字小键盘那样工作，数字小键盘把小键盘的位置映射成动作，以此让它兼容键盘和小键盘输入。

游戏手柄背后的思想是，在游戏手柄区域的任何位置，用户都可以初始化一个触摸来设置控件的中心位置，之后，游戏手柄应相对于该位置来检测移动。同样，游戏手柄在大小、颜色和透明度，以及在它所触发的动作方面是可配置的；不过，为了简化最常见使用情况的初始化，代码为大部分选项设置了默认值。此外，游戏手柄还会触发与键盘和小键盘一样的默认动作，即向上、右、左和下移动。

与小键盘不同，游戏手柄需要区别对待每个触摸事件。touchstart 事件用来启动手柄和设置手柄的中心位置，捕捉触摸的标识，这样从这个时候开始，就只有该触摸可用来调整游戏手柄。touchmove 事件用来真正移动游戏手柄的中心位置，最后，touchend 事件删除手柄。

将代码清单 10-6 中的代码添加到 Q.InputSystem 定义内部的小键盘函数之后，即置于代码清单 10-5 中的代码之后。

代码清单 10-6：游戏手柄控件

```
joypadControls: function(opts) {
  if(this.joypadEnabled) return false;
  if(!hasTouch) return false;
  var joypad = Q.joypad = _.defaults(opts || {},{
```

```
    size: 50,
    trigger: 20,
    center: 25,
    color: "#CCC",
    background: "#000",
    alpha: 0.5,
    zone: (Q.el.attr('width')||Q.width) / 2,
    joypadTouch: null,
    inputs: DEFAULT_JOYPAD_INPUTS,
    triggers: []
  });

  Q.el.on('touchstart',function(e) {
    if(joypad.joypadTouch === null) {
      var evt = e.originalEvent,
      touch = evt.changedTouches[0],
      loc = Q.input.touchLocation(touch);

      if(loc.x < joypad.zone) {
        joypad.joypadTouch = touch.identifier;
        joypad.centerX = loc.x;
        joypad.centerY = loc.y;
        joypad.x = null;
        joypad.y = null;
      }
    }
  });

  Q.el.on('touchmove',function(e) {
    if(joypad.joypadTouch !== null) {
      var evt = e.originalEvent;

      for(var i=0,len=evt.changedTouches.length;i<len;i++) {
        var touch = evt.changedTouches[i];

        if(touch.identifier === joypad.joypadTouch) {
          var loc = Q.input.touchLocation(touch),
              dx = loc.x - joypad.centerX,
              dy = loc.y - joypad.centerY,
              dist = Math.sqrt(dx * dx + dy * dy),
              overage = Math.max(1,dist / joypad.size),
              ang =  Math.atan2(dx,dy);

          if(overage > 1) {
            dx /= overage;
            dy /= overage;
            dist /= overage;
          }

          var triggers = [
```

```
            dy < -joypad.trigger,
            dx > joypad.trigger,
            dy > joypad.trigger,
            dx < -joypad.trigger
          ];

          for(var k=0;k<triggers.length;k++) {
            var actionName = joypad.inputs[k];
            if(triggers[k]) {
              Q.inputs[actionName] = true;

              if(!joypad.triggers[k]) {
                Q.input.trigger(actionName);
              }
            } else {
              Q.inputs[actionName] = false;
              if(joypad.triggers[k]) {
                Q.input.trigger(actionName + "Up");
              }
            }
          }

          _.extend(joypad, {
            dx: dx, dy: dy,
            x: joypad.centerX + dx,
            y: joypad.centerY + dy,
            dist: dist,
            ang: ang,
            triggers: triggers
          });

          break;
        }
      }
    }
    e.preventDefault();

});

Q.el.on('touchend touchcancel',function(e) {
    var evt = e.originalEvent;

    if(joypad.joypadTouch !== null) {
      for(var i=0,len=evt.changedTouches.length;i<len;i++) {
      var touch = evt.changedTouches[i];
        if(touch.identifier === joypad.joypadTouch) {
          for(var k=0;k<joypad.triggers.length;k++) {
            var actionName = joypad.inputs[k];
            Q.inputs[actionName] = false;
          }
```

```
                joypad.joypadTouch = null;
                break;
              }
            }
          }
          e.preventDefault();
        });

        this.joypadEnabled = true;
      },
```

与小键盘类似，主要方法 joypadControls 先为游戏手柄设置配置，然后设置一些事件处理程序。因为游戏可能需要访问诸如移动强度和角度之类的游戏手柄模拟信息，所以，为便于访问，手柄信息存放在 Q.joypad(而非 Q.input.joypad)中。

第一个处理程序 touchstart 负责识别激活手柄的触摸，其做法是先通过检查 joypad.joypadTouch 被设置成 null 来确定手柄尚未激活(注意句中的三个等号：===，这是必需的，因为触摸标识有可能为 0，若允许类型强制的话，此时的标识与 null 进行比较会返回 true)。接下来，该方法检查触摸是否发生在游戏手柄区域内，该区域默认为游戏区的左侧，若是，方法捕获标识及触摸的中心位置，然后将内圆位置设置成 null 来防止任何的初始移动。

touchmove 处理程序完成了大量的游戏手柄计算工作，该处理程序首先通过检查 joypad.joypadTouch 未被设成 null 来确定手柄已被激活；接着，它通过检查所有已发生改变的触摸来查找与存储在 joypad.joypadTouch 中的标识匹配的触摸，若找到，则计算该触摸的中心位置。为防止中心位置落到游戏手柄边界之外，中心位置被限制成不超出游戏手柄的范围。游戏手柄的总距离和角度也被计算出来，目的是允许游戏提取手柄方向和强度方面的模拟信息。

接下来，touchmove 处理程序检查四种动作触发器的每一种，看看游戏手柄是否发生移动，移动距离是否远得足以算作一次触发移动。若有新的触发器被激活或某个旧的触发器不再是活动的，处理函数就触发相应的事件，然后更新游戏手柄的当前状态。

最后，touchend 处理程序检查发生改变的触摸中是否存在与游戏手柄标识相匹配的触摸，若有，则将游戏手柄设置成不可用。

10.3.6 绘制屏幕输入

现在，Quintus.Input 模块已经具备了需要用来捕获用户输入的各种控件，它还缺少的是一种把这些控件绘制到屏幕上的做法。小键盘的按钮被绘制成一系列带有文本的方块(参见第 3 章)，游戏手柄则被绘制成一对同心圆。将代码清单 10-7 中的四个方法添加到 joypadControls 方法的后面。

代码清单 10-7：绘制屏幕输入

```
drawButtons: function() {
  var keypad = Q.input.keypad,
      ctx = Q.ctx;
```

```
    ctx.save();
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";

    for(var i=0;i<keypad.controls.length;i++) {
      var control = keypad.controls[i];

      if(control[0]) {
        ctx.font = "bold " + (keypad.size/2) + "px arial";
        var x = i * keypad.unit + keypad.gutter,
            y = keypad.bottom - keypad.unit,
            key = Q.inputs[control[0]]

        ctx.fillStyle = keypad.color || "#FFFFFF";
        ctx.globalAlpha = key ? 1.0 : 0.5;
        ctx.fillRect(x,y,keypad.size,keypad.size);

        ctx.fillStyle = keypad.text || "#000000";
        ctx.fillText(control[1],
                     x+keypad.size/2,
                     y+keypad.size/2);
      }
    }

    ctx.restore();
  },

  drawCircle: function(x,y,color,size) {
    var ctx = Q.ctx,
        joypad = Q.joypad;

    ctx.save();
    ctx.beginPath();
    ctx.globalAlpha=joypad.alpha;
    ctx.fillStyle = color;
    ctx.arc(x, y, size, 0, Math.PI*2, true);
    ctx.closePath();
    ctx.fill();
    ctx.restore();
  },

  drawJoypad: function() {
    var joypad = Q.joypad;
    if(joypad.joypadTouch !== null) {
      Q.input.drawCircle(joypad.centerX,
                         joypad.centerY,
                         joypad.background,
                         joypad.size);
```

```
    if(joypad.x !== null) {
      Q.input.drawCircle(joypad.x,
                         joypad.y,
                         joypad.color,
                         joypad.center);
    }
  }

},

drawCanvas: function() {
  if(this.touchEnabled) {
    this.drawButtons();
  }

  if(this.joypadEnabled) {
    this.drawJoypad();
  }
}
```

drawCanvas 方法是其中的主要方法，游戏的每帧都需要调用它来绘制控件，但它只可用于基于画布的游戏。该方法调用两个辅助方法：drawButtons 和 drawJoypad。drawJoypad 进而又调用一个名为 drawCircle 的辅助方法来绘制游戏手柄圆的外圆和内圆。

drawButtons 遍历每个控件并为其绘制一个方形，且在方形上绘制用来标识按钮的文本字符。它还检查每个小键盘按钮的状态，这样就能以高亮方式绘制当前被按下的按钮。该方法使用画布的 textAlign 和 textBaseline 特性在按钮上居中显示文本，这些内容将在第 15 章中详细讨论。fillText 方法的速度可能很慢，所以在生产环境中你极可能希望用图像替代文本按钮。

drawCircle 使用 arc 函数加上一个 fill 函数来绘制圆，drawJoypad 调用该方法两次：一次绘制外圈较大的圆，一次绘制中间的圆。可参阅第 15 章深入了解画布 API 的说明。

10.3.7 完善和测试输入

现在，Quintus.Input 模块的功能已经编写完成，唯一还可添加的内容是一点作为辅助的胶水代码——帮助用户快速搭建运行环境——以及一个用来测试整块代码的 HTML 文件。

基于这样的一种假设，即除了后备的桌面键盘控件之外，输入系统的最常见用法是一个有着 a 和 b 两个按钮的双向小键盘，或是一个有着 a 和 b 两个按钮的四向游戏手柄。可通过将一个简单的辅助方法添加到顶层的 Quintus 模块来实现这样的设置。将代码清单 10-8 中的代码添加至 Q.input 定义的后面。

代码清单 10-8：控件的辅助方法

```
Q.input = new Q.InputSystem();
```

```
  Q.controls = function(joypad) {
    Q.input.keyboardControls();

    if(joypad) {
      Q.input.touchControls({
        controls: [ [],[],[],['action','b'],['fire','a']]
      });
      Q.input.joypadControls();
    } else {
      Q.input.touchControls();
    }

    return Q;
  };
};
```

这段代码只接收一个参数，一个关于是要打开游戏手柄还是仅使用小键盘的布尔类型选项。

接下来要做的事情是测试这一代码，创建一个名为 input_test.html 的文件，将代码清单 10-9 中的代码填写到该文件中。

代码清单 10-9：input_test.html 输入测试

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Input Test</title>
    <meta name='viewport' content='width=device-width, user-scalable=no'>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_input.js'></script>
  </head>
  <body>
    <script>
      var Q = Quintus()
              .include("Input")
              .setup("quintus",{ "maximize": true })
              .controls(true);
      Q.input.bind('fire',function() {
        console.log('fire!');
      });
      Q.input.bind('fireUp',function() {
        console.log('fire up');
      });
      Q.gameLoop(function() {
        Q.clear();
        Q.input.drawCanvas();
```

```
    });
    Q.el.css('backgroundColor','#666');
  </script>
 </body>
</html>
```

你会看到，这段代码首先完成 Quintus 的设置步骤，其中包括了你刚编写的 Quintus.Input 模块；然后打开游戏手柄控件。此外，它还绑定了两个事件处理程序，目的是测试被触发的事件。可以试着绑定其他的一些输入，包括向左、右、上和下移动的输入等。在移动设备上，若打开了控制台，你应会看到控制台日志。

在桌面和 WP7 上，你不会看到任何控件，因为触摸事件不获支持，不过发射键这一触发器在桌面上是可用的。

应把屏幕的左侧用作游戏手柄，使用右侧来按压按钮，如图 10-1 所示。

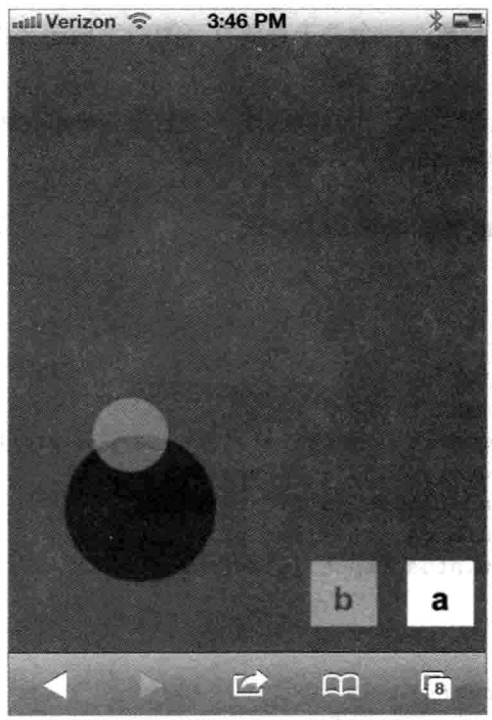


图 10-1　游戏手柄和按钮输入

10.4　加载资产

除非正在构建的是一个完全不同一般的游戏，否则你总需要在某些时刻将一些资产加载到游戏中。资产包括图像、音频、精灵和关卡数据，以及其他任何游戏需要用来运行的、被保存在单独文件中的东西。

从某种意义上说，HTML5 会自动处理资产的加载。若创建了一个 Image 对象、设置

了来源，然后绘制它，那么浏览器很愿意遵从你的要求，不会抛出任何错误消息。不过，在图像完全被下载之前，它不会在屏幕上绘制该图像，这意味着游戏元素是突然“弹出”到页面上的。一种更好的策略是，先显示一个加载画面，在此期间预先加载所有的资产，等到所有资产都准备就绪之后才开始游戏。以下代码给出了一个例子，说明应如何使用资产加载功能：

```
var Q = Quintus().setup();
Q.load(['sprites.png','correct.ogg'],function() {
  alert('Everything loaded!');
});
```

为将加载方法变得更灵活，Q.load 会被设计成以一种弹性方式接收参数，既可以接收单个资产名称字符串或一个资产数组，也可以接收一个映射了资产名称和要加载文件的名称的对象。

本书已有两次处理图像的经验，第一次在第 1 章中，然后是第 8 章，这将是最后一次，你将为 Quintus 构建一个通用的资产加载器，该加载器会在本书余下的内容中被重复使用。

10.4.1　定义资产类型

为明确如何加载某个特定文件，引擎需要拥有一个可转换成特定资产类型的文件扩展名列表，这其实是一个很简短的列表，因为引擎目前只关心图像和音频文件。Quintus 还应根据文件名返回资产类型，这意味着引擎需要使用一个方法来提取扩展名，并在资产类型列表中查找类型。

要实现这一点，再次打开 quintus.js，将代码清单 10-10 中的代码添加到最后的 return 语句之前。

代码清单 10-10：资产类型功能

```
// Augmentable list of asset types
Q.assetTypes = {
  // Image Assets
  png: 'Image', jpg: 'Image', gif: 'Image', jpeg: 'Image',
  // Audio Assets
  ogg: 'Audio', wav: 'Audio', m4a: 'Audio', mp3: 'Audio'
};

// Determine the type of an asset with a lookup table
Q.assetType = function(asset) {
  // Determine the lowercase extension of the file
  var fileExt = _(asset.split(".")).last().toLowerCase();

  // Lookup the asset in the assetTypes hash, or return other
  return Q.assetTypes[fileExt] || 'Other';
};
```

第一块代码定义了一个对象，该对象把小写的文件扩展名映射到大写的资产类型上，

因为被定义成 Q 的一个公共对象，所以该对象很容易被扩展，这样就能处理更多的资产类型。

方法 Q.assetType 所做的工作是把文件名转换成一个小写的扩展名，然后在 Q.assetTypes 对象中查找资产类型。该方法使用一行代码从文件名中提取扩展名，其借助句点符号来分割文件名，把分割结果放到一个数组中，然后提取该数组的最后一个元素作为扩展名，并将其转换成小写；最后，方法返回找到的值，对于其他类型的文件来说，则返回一个特殊的 Other 类型。

10.4.2 加载特定资产

接下来的任务是为不同的资产类型编写方法，通过创建 Image 类型和 Audio 类型的对象来加载相应类型的资产，Other 类型的资产则仅通过 jQuery 的 Ajax get 请求进行加载。这里所做的假设是，若开发者希望加载一些类型随机的文件，那么在你提供了数据的情况下，方法可以推断出如何处理这些数据。

将代码清单 10-11 中的三个加载方法添加到 quintus.js 的末尾处，同样是放在最后的 return 语句之前。

代码清单 10-11：资产加载方法

```
// Loader for Images
Q.loadAssetImage = function(key,src,callback,errorCallback) {
  var img = new Image();
  $(img).on('load',function() {  callback(key,img); });
  $(img).on('error',errorCallback);
  img.src = Q.options.imagePath + src;
};


Q.audioMimeTypes = { mp3: 'audio/mpeg',
                     ogg: 'audio/ogg; codecs="vorbis"',
                     m4a: 'audio/m4a',
                     wav: 'audio/wav' };

// Loader for  Audio
Q.loadAssetAudio = function(key,src,callback,errorCallback) {
  if(!document.createElement("audio").play || !Q.options.sound) {
    callback(key,null);
    return;
  }

  var snd = new Audio(),
      baseName = Q._removeExtension(src),
      extension = null,
      filename = null;

  // Find a supported type
```

```
  extension =
    _(Q.options.audioSupported)
    .detect(function(extension) {
      return snd.canPlayType(Q.audioMimeTypes[extension]) ?
                              extension : null;
  });

  // No supported audio = trigger ok callback anyway
  if(!extension) {
    callback(key,null);
    return;
  }

  // If sound is turned off,
  // call the callback immediately
  $(snd).on('error',errorCallback);
  $(snd).on('canplaythrough',function() {
    callback(key,snd);
  });
  snd.src = Q.options.audioPath + baseName + "." + extension;
  snd.load();
  return snd;
};

// Loader for other file types, just store the data
// returned from an ajax call
Q.loadAssetOther = function(key,src,callback,errorCallback) {
  $.get(Q.options.dataPath + src,function(data) {
    callback(key,data);
  }).fail(errorCallback);
};
```

这三个加载器方法中的每一个用到的资产类型不同，但都执行了同样的任务，也即它们提供了一个一致的回调调用，该回调使用被传入的键和正在处理的被加载对象。此外，该段代码还定义了一个从文件名中删除扩展名的辅助方法。

Q.loadAssetImage 创建一个 Image 对象，并把回调和加载事件方法关联起来。Q.loadAssetOther 仅使用 jQuery 的 get 方法把数据加载到缓冲区中，并在加载完毕后触发回调。

最后要说的是 Q.loadAssetAudio 方法，该方法创建一个 Audio 元素，然后确定 Audio 标签是否可用或声音是否已正常打开。在不支持音频标签或声音被关闭的情况下，方法立刻调用回调并返回。

Audio 加载器还需要做一些额外的工作来确保浏览器支持音频文件的加载，为实现这一点，它保存了一个名为 Q.audioMimeTypes 的哈希，该哈希把文件扩展名映射到音频的 MIME 类型上。有了扩展名对应的 MIME 类型，它就可以使用该 MIME 类型来调用 Audio.canPlayType，以此来检查受支持类型的文件是不是可播放的。

有必要对所有这些文件类型都执行一个播放检测，因为没有哪种音频格式获得了所有

浏览器的支持，所以，确保提供尽可能多的音频文件格式来满足受支持浏览器的需要的责任就落到开发者的头上，一般来说，这意味着要把音频资产变成.mp3 或.ogg 这种可用的格式。为减轻开发者的负担，上述代码清单中的代码剥除了所有音频文件的扩展名，然后尽力去找出这样的一个文件，即该文件的格式既是开发者已在 Q.options.audioSupported 中表明支持的，也是获得浏览器支持的。

若没有提供受支持的音频，引擎会表现得好像一切依然顺利，然后让用户在没有音响效果的情况下玩游戏。这意味着若开发者只想支持一种音频格式，如.mp3，那么他们可以这么做，因为游戏仍可以运行，只不过是没有音效罢了。

上述每个方法还会接收一个 errorCallback 回调参数，若在加载资产过程中出现问题，则触发该回调。

10.4.3 完善加载器

为了完善基本的加载器功能，引擎需要在某个地方记录下资产，此外，还需要一个方法来完成同步资产加载的繁重工作。

不过，在到达这一步之前，需要首先将一些默认值添加到引擎中，以此简化资产加载的过程，并提供一个游戏支持的音频格式列表。在接近 Quintus 定义顶部的地方，你已经定义了一个可存放引擎任何默认值的选项哈希(options)，更新该哈希，使其与代码清单 10-12 中的代码保持一致。

代码清单 10-12：Quintus 的默认选项

```
var Quintus = function(opts) {
  var Q = {};
  Q.options = {
    imagePath: "images/",
    audioPath: "audio/",
    dataPath:  "data/",
    audioSupported: [ 'mp3','ogg' ],
    sound: true
  };
  if(opts) { _(Q.options).extend(opts); }
```

资产被保存在系统的一个简单对象中，通过名称进行索引。要在哈希中进行查找，只需要调用 Q.asset(name)即可。实际加载列表中的资产的繁重工作由 Q.load 方法来处理，它的任务是接收编程者传递进来的任何关于要加载资产的内容(这可能是一个字符串、一个数组或一个对象)，把内容转换成一个一致的数据结构，然后遍历每个对象，调用适当的加载器方法(方法定义如代码清单 10-11 所示)。

将代码清单 10-13 中的代码添加到 quintus.js，照常放在最后的 return 语句之前。

代码清单 10-13：资产加载

```
  // Return a name without an extension
```

```
Q._removeExtension = function(filename) {
  return filename.replace(/\.(\w{3,4})$/,"");
};

// Asset hash storing any loaded assets
Q.assets = {};


// Getter method to return an asset
Q.asset = function(name) {
  return Q.assets[name];
};

// Load assets, and call our callback when done
Q.load = function(assets,callback,options) {
  var assetObj = {};

  // Make sure we have an options hash to work with
  if(!options) { options = {}; }

  // Get our progressCallback if we have one
  var progressCallback = options.progressCallback;

  var errors = false,
      errorCallback = function(itm) {
        errors = true;
        (options.errorCallback ||
         function(itm) { alert("Error Loading: " + itm ); })(itm);
      };

  // If the user passed in an array, convert it
  // to a hash with lookups by filename
  if(_.isArray(assets)) {
    _.each(assets,function(itm) {
      if(_.isObject(itm)) {
        _.extend(assetObj,itm);
      } else {
        assetObj[itm] = itm;
      }
    });
  } else if(_.isString(assets)) {
    // Turn assets into an object if it's a string
    assetObj[assets] = assets;
  } else {
    // Otherwise just use the assets as is
    assetObj = assets;
  }

  // Find the # of assets we're loading
  var assetsTotal = _(assetObj).keys().length,
```

```
    assetsRemaining = assetsTotal;

  // Closure'd per-asset callback gets called
  // each time an asset is successfully loadded
  var loadedCallback = function(key,obj) {
    if(errors) return;

    // Add the object to our asset list
    Q.assets[key] = obj;

    // We've got one less asset to load
    assetsRemaining--;

    // Update our progress if we have it
    if(progressCallback) {
      progressCallback(assetsTotal - assetsRemaining,assetsTotal);
    }

    // If we're out of assets, call our full callback
    // if there is one
    if(assetsRemaining === 0 && callback) {
      // if we haven't set up our canvas element yet,
      // assume we're using a canvas with id 'quintus'
      callback.apply(Q);
    }
  };

  // Now actually load each asset
  _.each(assetObj,function(itm,key) {

    // Determine the type of the asset
    var assetType = Q.assetType(itm);

    // If we already have the asset loaded,
    // don't load it again
    if(Q.assets[key]) {
      loadedCallback(key,Q.assets[key]);
    } else {
      // Call the appropriate loader function
      // passing in our per-asset callback
      // Dropping our asset by name into Q.assets
      Q["loadAsset" + assetType](key,itm,
                                 loadedCallback,
                                 function() { errorCallback(itm); });
    }
  });

};
```

第一个定义是资产哈希(assets)，该哈希被定义成 Q 的一个具有简单的 Q.asset 读取器

(getter)方法的空对象，该方法仅查找 assets 哈希的键。定义一个读取器方法而非直接公开 Q.assets 哈希，这种做法允许他人重写该方法，比如说，重写该方法来创建一些动态生成的资产。

load 方法是用来加载资产的主要方法，它的功能可分为三个部分：

- 第一部分把开发者想要加载的任何资产转换成一种一致的格式，该格式看起来类似如下：

```
{ "assetname.ext" : "assetname.ext",
  "assetname2.ext" : "assetname2.ext" }
```

- 接下来的部分定义 loadedCallback，该方法被当成回调方法传递给每一个资产类型特定的加载方法，它记录下剩余资产的数目，并最终在所有资产都被正确加载之后触发最后的回调。若有任何错误记录在案，该回调会提早返回，防止游戏试图启用无效的资产。若已经提供了进度回调(如在加载界面上显示进度条)，那么该回调在每次加载一个新的资产时都会调用进度回调。
- 最后一部分遍历资产哈希中的每个元素，确定正确的资产类型，然后把它分配给相应的加载器函数。

这一资产加载方法允许开发者基于文件扩展名创建新的资产类型，并能在必要时添加新的加载器方法。

10.4.4　添加预加载支持

虽然 Q.load 方法已完成了大部分的繁重工作，不过引擎还可进一步简化开发者的工作，做法是提供预加载支持，允许开发者在发起真正的加载调用之前把资产标识成预加载的。在组装场景或在由不同模块负责确定需要加载哪些资产时，这种做法很有用处。

将代码清单 10-14 中的代码添加到 quintus.js 的末尾处，放在最后的 return 语句之前。

代码清单 10-14 预加载支持

```
// Array to store any assets that need to be
  // preloaded
  Q.preloads = [];

  // Let us gather assets to load
  // and then preload them all at the same time
  Q.preload = function(arg,options) {
    if(_(arg).isFunction()) {
      Q.load(_(Q.preloads).uniq(),arg,options);
      Q.preloads = [];
    } else {
      Q.preloads = Q.preloads.concat(arg);
    }
  };
```

这段预加载代码提供了仅由一个名为 Q.preload 的方法构成的简单接口，该方法接收一

个回调方法或者一或多个需要加载的资产作为参数，它检查参数并确定，是将更多项目添加到预加载列表中还是通过调用 Q.load 来真正完成加载。代码简单调用 Array.concat 函数把项目添加到预加载列表中，该函数可能将一个元素添加至数组的末尾处，而如果被传进来的参数也是一个数组，那么它会把两个数组连接成一个数组。

以下是预加载代码的一些可能用法：

```
var Q = Quintus({ audioSupported: [ 'wav','ogg']}).setup();

Q.preload('sprites.png');
Q.preload(['fire.mp3','explosion.mp3']);
Q.preload(function() {
  alert("All loaded!");
});
```

把生成加载资源列表的调用和加载调用分开，在组件负责跟踪其所需资源的情况下，这种做法提供了更大的灵活性。可研究一下本章代码中的 asset_test.html 文件，这是一个真正可运行的例子，该例子加载了两个文件。

10.5 小结

现在，你拥有了 HTML5 游戏引擎所需的其他一些粘合模块，即设置、输入和资产模块等。对于游戏来说，若希望在屏幕上显示任意内容，可视容器的设置至关重要；在移动设备上实现输入则要比桌面上的来得复杂一些，因为任何类型的输入都需要使用诸如按钮或游戏手柄一类的可视元素来显示给用户；至于资产加载，这虽然不是什么很吸引人的主题，但做到正确实现还是很重要的，没有什么比得上在游戏过程中屏幕上突然弹出一堆元素更能给游戏的专业水准带来杀伤力的了。

第11章

自建 Quintus 引擎(3)

本章提要

- 创建精灵表
- 添加精灵
- 构建 Stage 类
- 用 Quintus 构建一个简单游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 11 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

11.1 引言

前两章介绍了关于构建 Quintus 引擎的内容，为构建游戏奠定了基础。本章通过添加对精灵和舞台的支持来充实引擎的功能。精灵是引擎用来在屏幕上显示图形和元素的主要可视游戏对象(GameObject)，精灵使用一些基本资产或精灵表(SpriteSheet)把自身绘制到游戏中；场景(Scene)则提供了一种方式把诸如关卡一类的游戏的某个离散部分包装成一个既好用又可重用的包；最后，作为游戏状态的主要容器，舞台(Stage)负责跟踪 GameObject 列表，以及确保它们在每帧中的更新和绘制。

11.2　定义精灵表

在着手处理精灵之前，引擎打算首先添加对精灵表的支持。如你在之前的章节中所见，精灵表支持在单个图像中存储任意数目的图像，这样做的目的是加快游戏的加载过程，以及把动画变得更便于使用。精灵表功能被嵌入到了第 8 章的代码中，这些代码通过命令行生成精灵表，这样就不必再费时费力地计算精灵的 x 和 y 坐标位置了。

11.2.1　创建 SpriteSheet 类

在这个例子中，一个 SpriteSheet 对象只引用同一精灵的一组大小相似的帧，一个单次加载的图像资产可被编译成若干 SpriteSheet 对象。

Quintus 中的精灵功能将自成一个模块。创建一个名为 quintus_sprites.js 的文件并打开它，将代码清单 11-1 中的代码放到该文件中，以此来定义 SpriteSheet 类的功能。

代码清单 11-1：Q. SpriteSheet 类

```
Quintus.Sprites = function(Q) {

  // Create a new sprite sheet
  // Options:
  //  tilew - tile width
  //  tileh - tile height
  //  w     - width of the sprite block
  //  h     - height of the sprite block
  //  sx    - start x
  //  sy    - start y
  //  cols  - number of columns per row
  Q.SpriteSheet = Class.extend({
    init: function(name, asset,options) {
      _.extend(this,{
        name: name,
        asset: asset,
        w: Q.asset(asset).width,
        h: Q.asset(asset).height,
        tilew: 64,
        tileh: 64,
        sx: 0,
        sy: 0
        },options);
      this.cols = this.cols ||
                  Math.floor(this.w / this.tilew);
    },

    fx: function(frame) {
      return (frame % this.cols) * this.tilew + this.sx;
    },
```

```
    fy: function(frame) {
      return Math.floor(frame / this.cols) * this.tileh + this.sy;
    },

    draw: function(ctx, x, y, frame) {
      if(!ctx) { ctx = Q.ctx; }
      ctx.drawImage(Q.asset(this.asset),
                    this.fx(frame),this.fy(frame),
                    this.tilew, this.tileh,
                    Math.floor(x),Math.floor(y),
                    this.tilew, this.tileh);

    }

  });

  return Q;
};
```

SpiteSheet 类较为简短，只用了不到 40 行代码。充当构造函数的 init 方法什么也不做，仅根据传入的资产对象和一组选项来初始化对象的初始值。除构造函数外，该类只包含了三个方法，它们分别是计算帧在精灵表中的 x 和 y 位置的 fx 和 fy 方法，以及 draw 方法，该方法在被传入的上下文的 x 和 y 位置上绘制 SpriteSheet 的某一特定帧。

fy(frame y 的缩写)的计算方法是，首先使用 Math.floor 来获取被传进来的特定帧所在的行，然后使用每一区块(tile)的高度乘以行数。fx 的计算方法是，使用取模运算符找出横向索引每行的区块数，然后把得到的区块数乘以每个区块的宽度，由此得到最后的结果。因为缺乏对亚像素渲染的支持(在移动设备上尤其如此)，draw 方法还使用 Math.floor 把任何传入的 x 和 y 值转换成整数。

这一类定义的最终效果是，给定一个 SpriteSheet 对象，可在画布的 x 和 y 位置上快速绘制出一个特定帧。

11.2.2　跟踪和加载精灵表

创建可以绘制单独帧的精灵表，这在实现引擎对精灵表支持的道路上，你已经前进了一大步；不过，Quintus 还需要一个中央机制来编译和跟踪精灵表，简化精灵表的引用和查找。将代码清单 11-2 中的代码添加到 quintus_sprites.js 的末尾处，置于最后的结束花括号之前。

代码清单 11-2：加载和跟踪精灵表

```
Q.sheets = {};
Q.sheet = function(name,asset,options) {
  if(asset) {
    Q.sheets[name] = new Q.SpriteSheet(name,asset,options);
  } else {
    return Q.sheets[name];
```

```
  }
};

Q.compileSheets = function(imageAsset,spriteDataAsset) {
  var data = Q.asset(spriteDataAsset);
  _(data).each(function(spriteData,name) {
    Q.sheet(name,imageAsset,spriteData);
  });
};
```

其中的 Q.sheets 对象提供了一个存储精灵表的中央位置。

Q.sheet 方法承担了双重责任，既是一个设置器(setter)方法，也是一个读取器(getter)方法，它根据名称返回一个精灵表，或创建一个新的 SpriteSheet。把一个方法用于多种目的(在使用提供了原生支持的语言来实现时，这种做法称为方法重载(method overloading))，这有助于把需要开发者记住的方法名称的个数保持在一个较可管理的数量级上。你可看到，Q.sheet 检查用户是否传入了一个资产，若是，则创建一个新的 SpriteSheet 对象，通过调用 Q.asset(asset)在 Quintus 引擎中查找该资产，并继续往下传递任何附加的选项；若没有资产被传进来，则仅是通过名称查找一个精灵表并返回该精灵表。

代码清单 11-2 中给出的最后一个方法 Q.compileSheets 结合资产名称和 JSON 格式的精灵数据资产，通过由第 8 章的 spriter 生成器生成的数据自动生成一个或多个精灵表。

11.2.3 测试 SpriteSheet 类

为测试精灵表的功能，加载一个包含了一些精灵的页面，然后播放一两个动画。创建一个新的名为 spritesheet_test.html 的 HTML 文档，将代码清单 11-3 中的代码填写到其中。

代码清单 11-3：测试 SpriteSheet 类

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Sprite Test</title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_sprites.js'></script>
  </head>
  <body>
    <script>
      var Q = Quintus().include('Sprites').setup();
      Q.load(['sprites.png','sprites.json'],function() {
        Q.compileSheets('sprites.png','sprites.json');

        var slowDown = 4,
            frame1 = 0,
            frame2 = 0;
```

```
        Q.gameLoop(function() {
          Q.clear();

          var sheet1 = Q.sheet('man');
          sheet1.draw(Q.ctx,50,50,Math.floor(frame1/slowDown));
          frame1 = (frame1+1) % (sheet1.frames  * slowDown);

          var sheet2 = Q.sheet('blob');
          sheet2.draw(Q.ctx,150,50,Math.floor(frame2/slowDown));
          frame2 = (frame2+1) % (sheet2.frames * slowDown);
        });
      });
    </script>
  </body>
</html>
```

要运行该例子，确保你拥有一个存放了文件 sprites.png 的 images/目录和一个存放了文件 sprites.json 的 data/目录。此外，你可能还需要通过服务器(或 localhost)运行代码，目的是避开浏览器针对经由 Ajax 加载本地文件(在这个例子中是 sprites.json)的安全保护。

通过分析上述代码，你可看到，例子一开始首先加载并编译精灵列表，之后再运行一个简单的游戏循环来绘制两个具有不同名称的精灵的一帧，该循环通过两个计数器——frame1 和 frame2——来遍历每一帧，并通过一个 slowDown 因子来放缓代码的速度，避免动画播放得过快。

最终产生的效果是，页面的画布上有两个动画在不停回放。

11.3　添加精灵

定义了精灵表后，下一个任务是创建精灵对象。因为精灵一般对于 Quintus 和游戏来说都很重要，所以真正精灵类的出场可能会令人稍感意外：它的实现仅用了不到 40 行代码，且并未使用太多的内置功能。代码之所以不多，是因为大部分的重要功能都已构建完毕，这些功能驻留在其他地方：事件由 Evented 类处理，组件由 GameObject 类处理，图形由资产和 SpriteSheet 类处理等。

这意味着留给精灵类完成的工作就是把这各个部分捆绑在一起，组成一个包。本章构建的精灵类将被命名为 Sprite，第 13 章将讨论一个继承自 Sprite 的类 DomSprite，这是一个在使用 HTML 和 CSS3 构建游戏时会用到的类。

11.3.1　编写 Sprite 类

将代码清单 11-4 中的精灵代码添加到 quintus_sprites.js 文件中，置于最后的结束花括号之前。

代码清单 11-4：Sprite 类

```
// Properties:
//   x
//   y
//   z - sort order
//   sheet or asset
//   frame
Q.Sprite = Q.GameObject.extend({
  init: function(props) {
    this.p = _({
      x: 0,
      y: 0,
      z: 0,
      frame: 0,
      type: 0
      }).extend(props||{});
    if((!this.p.w || !this.p.h)) {
      if(this.asset()) {
        this.p.w = this.p.w || this.asset().width;
        this.p.h = this.p.h || this.asset().height;
      } else if(this.sheet()) {
        this.p.w = this.p.w || this.sheet().tilew;
        this.p.h = this.p.h || this.sheet().tileh;
      }
    }
    this.p.id = this.p.id || _.uniqueId();
  },

  asset: function() {
    return Q.asset(this.p.asset);
  },

  sheet: function() {
    return Q.sheet(this.p.sheet);
  },

  draw: function(ctx) {
    if(!ctx) { ctx = Q.ctx; }
    var p = this.p;
    if(p.sheet) {
      this.sheet().draw(ctx, p.x, p.y, p.frame);
    } else if(p.asset) {
      ctx.drawImage(Q.asset(p.asset),
                    Math.floor(p.x),
                    Math.floor(p.y));
    }
    this.trigger('draw',ctx);
  },
```

```
    step: function(dt) {
      this.trigger('step',dt);
    }
  });
```

如你所见，精灵基类的代码并没有多少，这是有意为之的。精灵基类被设计得尽量精简，依赖派生出来的类和组件来实现任何特定功能。

其中最长的方法是充当构造函数的 init 方法，该方法确保对象具有一个有效的属性对象 p，并从被分配给对象的资产或 SpriteSheet 对象中提取对象的宽度和高度。此外，它还通过 underscore.js 生成一个全局唯一的 ID，从而赋予每个精灵自身一个唯一标识。

接下来，Sprite 类定义了两个读取器方法——asset 和 sheet——在适用的情况下提取所分配的资产或 SpriteSheet 对象。同样，使用一个读取器方法意味着子级类(甚至组件)可重写该方法。

draw 方法负责把精灵资产真正绘制到画布上，它使用一些条件代码检查单个资产或 SpriteSheet 对象，而且可同时处理这两者。draw 方法是一个可被子级类重写的候选方法，这些子类具有更复杂的或嵌套的绘制功能。此外，该方法还会触发一个 draw 事件，以防组件需要完成一些额外的绘制工作。与 SpriteSheet 一样，因为对亚像素渲染的支持各异，draw 方法还使用 Math.floor 把任何传入的 x 和 y 值转换成整数。

step 方法仅是一个存根，它为所有监听组件触发一个 step 事件。

11.3.2　引用精灵、属性和资产

你在第 10 章中已看到，引擎把所有已加载资产存放在一个哈希中，以便能通过名称便捷地引用资产。如你刚刚所见，Q.sheets 对象中的 SpriteSheet 的引用方式与此类似。

这是有意为之的，因为我们的目的是让精灵通过名称(而非通过传入精灵表自身实例)来引用资产和精灵表，这样做是因为， Quintus 的主要目标之一是把精灵变成可序列化的，这意味着它们的当前状态可被写到硬盘上和本地存储中，或是可通过网络发送，然后在网络管道另一端完整无缺地重组出来。

要实现这一目标，引擎必须确保只有诸如字符串和数值一类的简单类型被用作属性类型，以及那些可序列化的属性要与其他属性和方法分开存放。你可看到，精灵类在属性 p 之下保存了一个单独的属性哈希，这一做法使对象属性的访问变得更加复杂，但也带来了额外的好处，那就是把状态的全部内容与对象的所有其他属性分隔开来。

因为为每个精灵指定一个名称有些多余，所以每个精灵是通过存储在 p.id 中的唯一标识符进行识别的。这种做法允许其他对象通过精灵对象的标识符(而非通过到精灵对象的实际引用)来存储和传递精灵对象，从而简化了垃圾收集和网络同步。

11.3.3　运用 Sprite 对象

画布的 Sprite 对象已准备就绪，现在是时候去快速实现一个示范性的游戏来展示 Sprite 类的用法了。本章构建一个简单的打砖块风格(breakout-style)游戏，首先在游戏底部添加一

个可由用户控制的球拍，图 11-1 呈现了到本章结束时游戏所具备的外观。

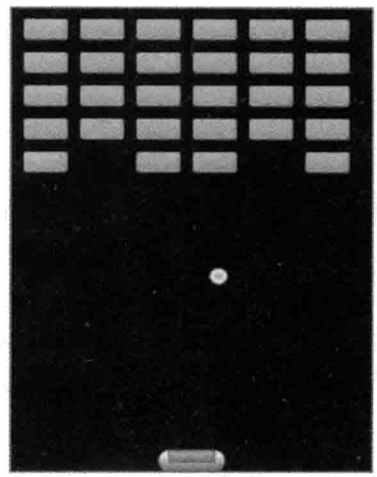


图 11-1　最终的游戏画面

新建并打开一个名为 blockbreak.html 的文件，将代码清单 11-5 中的 HTML 样板代码置于其中，以此作为页面的起始代码。还需要把上一章中的 quintus.js 和 quintus_input.js 文件，及其所依赖的 jquery.min.js 和 underscore.js 文件包含进来。

代码清单 11-5：blockbreak.html 的样板代码

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Block Break</title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_input.js'></script>
    <script src='quintus_sprites.js'></script>
    <script src='blockbreak.js'></script>
    <style>
      body { padding:0px; margin:0px; }
      canvas { background-color:black; }
    </style>
  </head>
  <body>
  </body>
</html>
```

接下来创建代码清单 11-5 中引用的 blockbreak.js 文件，输入代码清单 11-6 中的代码，实现一个简单的可运行的球拍程序。要运行该游戏，需要通过本地主机上的服务器来运行

它，因为游戏需要加载精灵文件。此外，还需要把作为游戏资产的 blockbreak.png 文件和 blockbreak.json 文件分别复制到游戏的 images/子目录和 data/子目录中，这样才能把资产变成可为游戏所用的。

代码清单 11-6：blockbreak.js

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites')
                      .setup();

  Q.input.keyboardControls();
  Q.input.touchControls({
            controls:  [ ['left','<' ],[],[],[],['right','>' ] ]
          });

  Q.Paddle = Q.Sprite.extend({
    init: function() {
      this._super({
        sheet: 'paddle',
        speed: 200,
        x: 0
      });
      this.p.x = Q.width/2 - this.p.w/2;
      this.p.y = Q.height - this.p.h;
      if(Q.input.keypad.size) {
        this.p.y -= Q.input.keypad.size + this.p.h;
      }
    },

    step: function(dt) {
      if(Q.inputs['left']) {
        this.p.x -= dt * this.p.speed;
      } else if(Q.inputs['right']) {
        this.p.x += dt * this.p.speed;
      }
      if(this.p.x < 0) {
        this.p.x = 0;
      } else if(this.p.x > Q.width - this.p.w) {
        this.p.x = Q.width - this.p.w;
      }
      this._super(dt);
    }
  });

  Q.Ball = Q.Sprite.extend({
    init: function() {
      this._super({
        sheet: 'ball',
```

```
      speed: 200,
      dx: 1,
      dy: -1,
    });
    this.p.y = Q.height / 2 - this.p.h;
    this.p.x = Q.width / 2 + this.p.w / 2;
  },

  step: function(dt) {
    var p = this.p;

    p.x += p.dx * p.speed * dt;
    p.y += p.dy * p.speed * dt;

    if(p.x < 0) {
      p.x = 0;
      p.dx = 1;
    } else if(p.x > Q.width - p.w) {
      p.dx = -1;
      p.x = Q.width - p.w;
    }

    if(p.y < 0) {
      p.y = 0;
      p.dy = 1;
    } else if(p.y > Q.height - p.h) {
      p.dy = -1;
      p.y = Q.height- p.h;
    }
    this._super(dt);


  }


});

Q.load(['blockbreak.png','blockbreak.json'], function() {
  Q.compileSheets('blockbreak.png','blockbreak.json');

  var paddle = new Q.Paddle();
  var ball = new Q.Ball();

  Q.gameLoop(function(dt) {
    Q.clear();

    paddle.step(dt);
    paddle.draw();

    ball.step(dt);
```

```
        ball.draw();

        Q.input.drawCanvas();
      });

    });

});
```

代码的设置和加载部分应会给人以熟悉的感觉。因为游戏现在通过一个单独的 JavaScript 文件进行加载，加载过程不再内嵌在文档中，所以整个文件就是一个包装器，体现为 jQuery 的文档就绪事件的一个回调：$(function( ) { .. })。接下来，代码创建游戏对象 Q 并进行设置，因为代码被包装在一个闭包中，所以变量 Q 也被添加到了 window 对象中，这样万一希望直接使用游戏的 API，可通过控制台使用 window.Q 访问它。游戏还设置了默认的键盘控件，还设置了一些自定义的 touchControls，目的是删除该游戏不必用到的 a 和 b 按钮。

接下来是 Q.Paddle 精灵，该精灵继承自 Sprite，init 方法使用该对象的一些初始属性来调用 Sprite 的构造函数 init，其中包括了将用来绘制对象的 SpriteSheet 的名称。init 方法做了一点点运算工作，计算出居中显示球拍的 x 位置和对象的 y 位置，因为在桌面计算机上，球拍可被置于页面底部，但在触摸设备上，它应向上稍做移动，为小键盘控件留下相应空间。

然后，球拍的 step 方法重写继承自 Sprite 的默认的 step 方法，使用 Q.inputs 对象来确定球拍是否应向左或向右移动。

Q.Ball 精灵也被添加了进来，该精灵仅在游戏区域四处弹跳，在遇到墙壁时反转方向。就目前来讲，它尚未与球拍进行交互。step 方法使用球的速度和方向来更新球在每帧中的位置。

最后一步，游戏运行 gameLoop，该方法清除画布、步进游戏、绘制球拍，然后绘制输入元素。

运行该例子，应能向左和向右移动球拍。

11.4　使用场景设置舞台

研究一下上一节例子中的 gameLoop，很容易就会发现，若每次步进都需要单独地更新和绘制每个对象，那么游戏的实际代码可能变得粗笨异常；在加入碰撞检测之后，事情有可能骤然间变得复杂起来。如你在第 2 章的 GameBoard 对象中所见，这里的解决方案基于这样的想法，即使用一个对象管理众多精灵的更新和绘制。Quintus 将这一对象称为 Stage(舞台)，还会额外加入一个 Scene(场景)对象概念，该对象用来设置某个特定舞台的舞台对象。Scene 的作用之一是简化关卡的设置，然后在关卡之间进行切换。

11.4.1 创建 Quintus.Scenes 模块

为着手实现场景功能，Quintus 添加一个名为 Quintus.Scenes 的新模块来包含类 Q.Stage 和 Q.Scene。实际上，Q.Scene 对象极其简单，它唯一的目的就是包装一个函数，该函数设置被传入的舞台对象。

Q.Stage 稍复杂些，不过看起来与第 2 章的 GameBoard 类似，只是增加了其他一些事件功能。

创建一个新的名为 quintus_scenes.js 的 JavaScript 文件，将代码清单 11-7 中的代码置于其中。

代码清单 11-7：Scenes 的功能

```
Quintus.Scenes = function(Q) {

  Q.scenes = {};
  Q.stages = [];

  Q.Scene = Class.extend({
    init: function(sceneFunc,opts) {
      this.opts = opts || {};
      this.sceneFunc = sceneFunc;
    }
  });

  // Set up or return a new scene
  Q.scene = function(name,sceneObj) {
    if(!sceneObj) {
      return Q.scenes[name];
    } else {
      Q.scenes[name] = sceneObj;
      return sceneObj;
    }
  };
};
```

场景的功能很简洁，包含了一个名为 Q.Scene 的类，该类有一个简单目标，那就是捕获一个回调方法和一个可选的选项哈希。然后，它使用一个 Q.scene 方法来同时充当读取器方法(在传入一个参数时)和设置器方法(在传入两个参数时)。

这一场景功能背后的设想是，你希望以独立方式来设置游戏的某一关或某一段内容，所以它放入一个独立的方法中来简化不同场景之间的切换。

11.4.2 编写 Stage 类

Q.Stage 类负责记录一个精灵列表并让精灵更新和渲染自身。与类 Q.Scene 相比，它的代码量要大许多，但所做的工作也要多许多。其大部分代码应与第 2 章中的类的代码看起来十分类似。

将代码清单 11-8 中的 Stage 类的定义添加到 Quintus.Scenes 模块的末尾处，置于最后的结束花括号之前。

代码清单 11-8：overlap 方法和 Stage 类

```
Q.overlap = function(o1,o2) {
  return !((o1.p.y+o1.p.h-1<o2.p.y) || (o1.p.y>o2.p.y+o2.p.h-1) ||
           (o1.p.x+o1.p.w-1<o2.p.x) || (o1.p.x>o2.p.x+o2.p.w-1));
};

Q.Stage = Q.GameObject.extend({
  // Should know whether or not the stage is paused
  defaults: {
    sort: false
  },

  init: function(scene) {
    this.scene = scene;
    this.items = [];
    this.index = {};
    this.removeList = [];
    if(scene) {
      this.options = _(this.defaults).clone();
      _(this.options).extend(scene.opts);
      scene.sceneFunc(this);
    }
    if(this.options.sort && !_.isFunction(this.options.sort)) {
       this.options.sort = function(a,b) { return a.p.z - b.p.z; };
    }
  },

  each: function(callback) {
    for(var i=0,len=this.items.length;i<len;i++) {
     callback.call(this.items[i],arguments[1],arguments[2]);
    }
  },

  eachInvoke: function(funcName) {
    for(var i=0,len=this.items.length;i<len;i++) {
      this.items[i][funcName].call(
        this.items[i],arguments[1],arguments[2]
      );
    }
  },

  detect: function(func) {
    for(var i = 0,val=null, len=this.items.length; i < len; i++) {
      if(func.call(this.items[i],arguments[1],arguments[2])) {
```

```
      return this.items[i];
    }
  }
  return false;
},

insert: function(itm) {
  this.items.push(itm);
  itm.parent = this;
  if(itm.p) {
    this.index[itm.p.id] = itm;
  }
  this.trigger('inserted',itm);
  itm.trigger('inserted',this);
  return itm;
},

remove: function(itm) {
  this.removeList.push(itm);
},

forceRemove: function(itm) {
  var idx = _(this.items).indexOf(itm);
  if(idx != -1) {
    this.items.splice(idx,1);
    if(itm.destroy) itm.destroy();
    if(itm.p.id) {
      delete this.index[itm.p.id];
    }
    this.trigger('removed',itm);
  }
},

pause: function() {
  this.paused = true;
},

unpause: function() {
  this.paused = false;
},

_hitTest: function(obj,type) {
  if(obj != this) {
    var col = (!type || this.p.type & type) && Q.overlap(obj,this);
    return col ? this : false;
  }
},
```

```
  collide: function(obj,type) {
    return this.detect(this._hitTest,obj,type);
  },

  step:function(dt) {
    if(this.paused) { return false; }

    this.trigger("prestep",dt);
    this.eachInvoke("step",dt);
    this.trigger("step",dt);

    if(this.removeList.length > 0) {
      for(var i=0,len=this.removeList.length;i<len;i++) {
       this.forceRemove(this.removeList[i]);
      }
      this.removeList.length = 0;
    }
  },

  draw: function(ctx) {
    if(this.options.sort) {
      this.items.sort(this.options.sort);
    }
    this.trigger("predraw",ctx);
    this.eachInvoke("draw",ctx);
    this.trigger("draw",ctx);
  }
});
```

Q.Stage 对象继承自 Q.GameObject，正如你可能已经想到的那样，这一做法允许它绑定和触发事件，以及通过组件加入其他一些行为。其中的 init 方法设置对象的初始属性，若有场景对象被传入，则执行 scene 方法。Quintus 舞台对象还支持 z 顺序这一概念，允许对象在被渲染之前先进行排序。

接下来的三个方法——each、eachInvoke 和 detect——与第 2 章中介绍的相同，它们用作帮助器方法，目的是把项目列表操作的执行变得更容易一些。each 和 eachInvoke 之间的区别在于，前者接收一个真正的回调方法作为参数，后者则调用一个被当成对象属性存储起来的方法。你会发现，这两个方法都是使用两个参数来调用 Function.call，而非调用 Function.apply，后一种调用可以支持作为数组传入的任意数目的参数。之所以做出这样的选择，是因为调用 Function.apply 需要创建一个新的数组对象，对于一些被频繁调用的方法来说，这是任何高质量的 HTML5 引擎都应该尽量避免的做法，目的是减少由垃圾收集器带来的运行不畅现象。

注意：作为一个 HTML5 游戏程序员，你只有一个真正的敌人，那就是垃圾收集器。JavaScript 是一种支持垃圾收集的语言，这意味着开发者不必关心内存的分配和释放。不过，垃圾收集器的缺点是，每隔一定时间，JavaScript 就需要通过运行垃圾收集器来清理不再被使用的那部分内存，收集器的运行可能会占用一些时间，有时超过 100 毫秒，这会给游戏的运行带来一个明显的阻滞。任何 JavaScript 引擎的目标都应该是在一个正常帧的更新渲染过程中尽可能少创建对象，并在需要创建精灵之类真正的游戏对象时保存对象的创建。

添加和删除对象功能的做法也与第 2 章中的如出一辙，insert、remove 和 forceRemove 这三个方法所执行的任务与以往的并无不同，仅是增加了对对象的 destroy 方法的调用和一些触发事件；此外，方法还会额外处理对象的 id 索引。还记得第 10 章中的 Q.GameObject 的代码吗？destroy 方法还调用了 remove，该调用可能导致一个无限递归循环，这就是添加 GameObject.destroyed 属性的原因所在，该属性允许你通过调用 GameObject.destroy()或 Stage.remove(object)删除对象，无论采用哪种做法，引擎都能做出正确反应。

为了把通过对象 id 查找对象的做法变得更容易一些，stage 对象既保存了已排序的项目数组，也保存了一个被当成哈希使用的对象，该哈希对象以对象的 id 为键来映射对象。

碰撞方法也与在第 2 章中所见到的类似，overlap 方法被提取了出来，被当作一个方法直接置于 Q 中，出于作用域方面的考虑，这样做可把该方法变得更易于获取。实现边框检测的主要方法——Stage.collide——调用一个名为_hitTest 的辅助方法，该方法用到了 Q.collide 方法。_hitTest 的定义可被当成一个匿名函数直接嵌在 Q.collide 内部，但这会导致函数定义在每次调用时都被解析，引擎的速度会因此受到一些影响，还会增加需要收集的垃圾数量。

最后，step 和 draw 方法遍历列表中的每个对象，调用每个对象的适当方法，以及在这之前和之后触发一些事件。此外，draw 方法还调用了一个可选的排序函数来确保对象按照正确的 z 顺序绘制。

11.4.3 丰富场景功能

场景所需的最后一部分代码是一些用于游戏的展现、清除、暂停、取消暂停和循环的辅助方法，这些实用方法被直接添加至 Q 对象实例中。

将代码清单 11-9 中的代码添加到 Quintus.Scenes 模块的末尾处，置于最后的结束花括号之前。

代码清单 11-9：场景和舞台的一些实用方法

```
Q.activeStage = 0;
```

```
Q.stage = function(num) {
  // Use activeStage is num is undefined
  num = (num === void 0) ? Q.activeStage : num;
  return Q.stages[num];
};

Q.stageScene = function(scene,num,stageClass) {
  stageClass = stageClass || Q.Stage;
  if(_(scene).isString()) {
    scene = Q.scene(scene);
  }

  num = num || 0;

  if(Q.stages[num]) {
    Q.stages[num].destroy();
  }

  Q.stages[num] = new stageClass(scene);

  if(!Q.loop) {
    Q.gameLoop(Q.stageGameLoop);
  }
};

Q.stageGameLoop = function(dt) {
  if(Q.ctx) { Q.clear(); }

  for(var i =0,len=Q.stages.length;i<len;i++) {
    Q.activeStage = i;
    var stage = Q.stage();
    if(stage) {
      stage.step(dt);
      stage.draw(Q.ctx);
    }
  }

  Q.activeStage = 0;

  if(Q.input && Q.ctx) { Q.input.drawCanvas(Q.ctx); }
};

Q.clearStage = function(num) {
  if(Q.stages[num]) {
    Q.stages[num].destroy();
    Q.stages[num] = null;
  }
};

Q.clearStages = function() {
```

```
    for(var i=0,len=Q.stages.length;i<len;i++) {
      if(Q.stages[i]) { Q.stages[i].destroy(); }
    }
    Q.stages.length = 0;
  };
```

与 Q.scene 相似，Q.stage 这一辅助方法返回一个特定的舞台对象，它的复杂程度有所增加，这体现在引擎保存了一个 activeStage 变量，该变量代表当前正在步入和绘制的舞台，这样便于精灵和引擎的其他部分引用这一活动的舞台对象。

与既充当读取器又充当设置器的 Q.scene 不同，为将一个新的舞台添加到游戏中，引擎提供了一个名为 Q.stageScene 的不同方法来接收并在舞台上展现一个场景。该方法可分别使用 0 到 3 个参数进行调用，第一种情况下，没有参数被传入，方法创建一个新的空舞台对象，并把它放到 stages 数组的首个槽位上。若传进来三个参数，则意味着由开发者控制要展现的场景，即要使用的槽位和舞台类。因为 Q.Stage 是一个普通的可扩展类，所以，希望扩展默认舞台功能(比如说添加一些更先进的碰撞检测算法)的模块可以这么做，并可使用该舞台类调用 Q.stageScene。

实际上，Q.stageScene 的代码很简单，若有字符串被传入，它就查找一个场景对象；若槽位中存在一个舞台对象，它就销毁该对象；然后在正确槽位上创建一个新的舞台对象；最后，查看游戏循环是否已被启动，若还没有，使用一个专门的 Q.stageGameLoop 方法(接下来定义)来运行循环，该方法确保每个场景的所有活动步进(step)都会被更新和渲染，这意味着在首次调用 Q.stageScene 时，引擎负责自动启动相应的游戏循环。

如刚才所介绍，Q.stageGameLoop 方法被传入到 Q.gameLoop 中，若存在上下文，该方法清除上下文，然后遍历 Q.stages 数组中的每个舞台对象(该数组有可能不是每个索引都有对应的值存在，故要小心行事，确保数组的索引对应了一个有效的舞台对象)。然后，它调用每个舞台对象的 step 和 draw 方法，设置 Q.activeStage 属性，这样该舞台上的任何精灵都能够调用 Q.stage()来获取当前的活动舞台对象(这样做的原因有很多，比如它们需要使用该舞台对象进行碰撞检测)。

注意：你可能想知道，既然仅一个舞台对象就能轻松支持所需数目的对象，引擎为什么还要耗费额外精力来支持多个活动舞台对象的同时存在呢？虽然还存在其他的一些原因，但这样做主要是为了便于在当前游戏之上添加一些分层游戏界面。例如，若正在开发一个 RPG 游戏，那么你可能会希望轻松做到在游戏界面之上弹出一个库存画面并暂停游戏。通过往引擎中加入一些对这两种功能的支持，在不必把太多复杂性带入引擎的情况下，引擎就能把这类功能接下来的编程工作变得更容易一些。

11.5　完成 Blockbreak 游戏的编写

在确定余下的这些场景功能之后，现在可以完成 Blockbreak 游戏的编写了。首先打开 blockbreak.html 文件，将刚创建的 quintus_scenes.js 文件添加到开始之处的 script 标签中：

```
<script src='jquery.min.js'></script>
<script src='underscore.js'></script>
<script src='quintus.js'></script>
<script src='quintus_input.js'></script>
<script src='quintus_sprites.js'></script>
<script src='quintus_scenes.js'></script>
<script src='blockbreak.js'></script>
```

接着需要在最初的设置代码中添加 Scenes 模块，把 blockbreak.js 开始处的 include 调用更新为以下内容：

```
$(function() {
  var Q = window.Q = Quintus()
                     .include('Input,Sprites,Scenes')
                     .setup();
  Q.input.keyboardControls()
  Q.input.touchControls({
           controls:  [ ['left','<' ],[],[],[],['right','>' ]]
         });
```

接下来，修改位于文件末尾处的 Q.load 回调内部的代码，该回调使用场景和舞台功能来启动游戏：

```
Q.load(['blockbreak.png','blockbreak.json'], function() {
  Q.compileSheets('blockbreak.png','blockbreak.json');
  Q.scene('game',new Q.Scene(function(stage) {
    stage.insert(new Q.Paddle());
    stage.insert(new Q.Ball());
  }));
  Q.stageScene('game');
});
```

可以看到，真正的代码被缩减成一个场景定义和该场景的舞台展现。现在，游戏的重置变得很简单，仅需调用 Q.stageScene('game')即可。重载游戏页面，确保每样东西仍起到与之前完全相同的作用。

既然样板代码的修改已不成问题，接下来首先添加对碰撞的支持，检测球和其可能接触到的其他任何物体之间的碰撞。在 blockbreak.js 文件的 Q.Ball 定义中，将一些碰撞检测代码添加到定义顶部，并添加一些代码在球掉到屏幕底部之外时重置游戏。为保持代码的简单性，Blockbreak 只设置了一个游戏关卡，且玩家只有一次生命。

```
step: function(dt) {
  var p = this.p;
  var hit = Q.stage().collide(this);
```

```
    if(hit) {
      if(hit instanceof Q.Paddle) {
        p.dy = -1;
      } else {
        hit.trigger('collision',this);
      }
    }

    p.x += p.dx * p.speed * dt;
    p.y += p.dy * p.speed * dt;

    if(p.x < 0) {
      p.x = 0;
      p.dx = 1;
    } else if(p.x > Q.width - p.w) {
      p.dx = -1;
      p.x = Q.width - p.w;
    }
    if(p.y < 0) {
      p.y = 0;
      p.dy = 1;
    } else if(p.y > Q.height) {
      Q.stageScene('game');
    }
  }
```

这段代码实现了球拍和球之间的交互，若没有击中球，让它掉到了屏幕底部之外，那么游戏将重新开始。虽然在一个空屏幕上到处追逐一个球可能让你觉得乐趣无穷，不过现在还是要在这一令人兴奋的空间中加入一些有棱有角的砖块。

砖块类不会有什么特别之处，其唯一明确拥有的功能就是在球触发了碰撞事件时额外执行一些代码。将代码清单 11-10 中的代码添加到 blockbreak.js 中，放在 Q.load 语句之前：

代码清单 11-10：Blockbreak 的 Q.Block 类

```
Q.Block = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props).extend({ sheet: 'block'}));
    this.bind('collision',function(ball) {
      this.destroy();
      ball.p.dy *= -1;
      Q.stage().trigger('removeBlock');
    });
  }
});
```

回顾一下在这两段代码之前对 Q.Ball 类所做的修改，你会发现球在触碰到任何非球拍物体时就已经触发了 collision 回调。砖块对象监听该碰撞回调、删除自身、反转球的纵向前进方向，它还触发一个 removeBlock 舞台事件。

接下来，需要修改 blockbreak.js 末尾处的那部分真正实现了加载的代码，将一些砖块添加到画面上，以及在所有砖块都被击中之后执行一些操作。这些操作很简单，仅是一个把游戏重置为启动的动作。修改以下突出显示的代码，完成 Blockbreak 游戏的编写：

```
Q.load(['blockbreak.png','blockbreak.json'], function() {
    Q.compileSheets('blockbreak.png','blockbreak.json');
    Q.scene('game',new Q.Scene(function(stage) {
      stage.add(new Q.Paddle());
      stage.add(new Q.Ball());

      var blockCount=0;
      for(var x=0;x<6;x++) {
        for(var y=0;y<5;y++) {
          stage.insert(new Q.Block({ x: x*50+10, y: y*30+10 }));
          blockCount++;
        }
      }
      stage.bind('removeBlock',function() {
        blockCount--;
        if(blockCount == 0) {
          Q.stageScene('game');
        }
      });
    }));

    Q.stageScene('game');
  });
```

可以注意到，代码使用变量 blockCount 来跟踪尚未被击毁的砖块的数目。为了设置砖块，scene 方法使用一些硬编码值来遍历变量 x 和 y 的坐标值，然后把这些元素添加到页面上。在接收到一个 removeBlock 事件时，它递减计数器，直至没有砖块剩余，然后直接重启游戏。

在 Blockbreak 游戏的开发进程上，本书只打算实现到这一步，因为该游戏的主要目的是作为一个例子，说明如何使用场景和精灵的功能来构建游戏——只用了不到 100 行代码，该游戏就完成了它的工作。若想成为一个完整的游戏，它还有一段路要走，因为：碰撞检测尚未考虑一些侧面碰撞；球拍位置并没有控制弹跳；且游戏没有提供多条玩家生命、没有显示得分，也没有弹出欢迎和游戏结束画面等。此外，加入一些增强能力的物件(power-up)也不错，不过这里把这些都留作练习来由你完成。

11.6　小结

现在，你已经完成了 Quintus 的初始功能，在其中加入了精灵、精灵地图、场景和舞台。该引擎现在已是一个完整的引擎，如本章在结束时构建出来的简单游戏 Blockbreak 所示，它已足以用来构建画布游戏。在接下来的几章中，你将会看到，在添加几个扩展之后，该引擎可用来构建 CSS3 和 SVG 游戏。

第Ⅳ部分

使用CSS3和SVG构建游戏

- 第 12 章：使用 CSS3 构建游戏
- 第 13 章：制作一个 CSS3 RPG 游戏
- 第 14 章：使用 SVG 和物理引擎构建游戏

第12章

使用 CSS3 构建游戏

本章提要

- 决定使用场景图
- 添加 DOM 功能至 Quintus
- 运行一个基于 DOM 版本的示例游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 12 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

12.1 引言

本章绕开使用画布构建游戏的做法，展示如何使用 DOM 元素和 CSS3 技术构建游戏，以及如何做到在移动设备上仍然保持良好性能。本章把对 DOM 的支持添加到了 Quintus 引擎，下一章以本章的功能为基础，构建一个 nethack 风格的简单地牢勇士(dungeon crawler)游戏的开头部分。

12.2 选定场景图

在开始深入研究如何使用 CSS3 构建游戏之前，我们首先要回答这样一个问题，即在什么情况下使用这三种可用的 HTML5 家族技术——画布(Canvas)、CSS3 和 SVG 之一来构

建游戏才是合理的？答案是，技术的采用很大程度上取决于三个因素：目标受众和设备、交互方法，以及性能需求。

12.2.1　目标受众

就第一个因素目标受众而言，你应该要注意的一点是，画布和 SVG 仅在桌面的 IE9 及更新版本中获得了原生支持，这意味着若你希望针对较旧版本的 IE 浏览器开发游戏，那么这两种技术都得排除在外。反过来，在移动设备上，画布元素仅在安装了 iOS 5 及更新版本、Android Chrome 或 WP 7.5 的设备上才是硬件加速的，一些较旧的智能手机无法填充太多的像素(在任何合理的画布分辨率下，全屏重绘都有可能发生溢出)，所以，若目标是这类设备，需要谨慎考虑技术的选择。

12.2.2　交互方法

游戏的类型及其所需的交互方法应能驱使你做出技术的选用决定。<canvas>标签是很强大，但并未以原生方式提供一个场景图，场景图(scene graph)是当前场景中每个元素的状态的一个层次化表示。对于 CSS3(DOM)和 SVG 来说，你拥有的是嵌套的多组元素，这些元素有着一些不相关的、可修改的表示(如改变某个 DOM 元素的 left 或 top 位置)。而对于画布来说，你所拥有的就是页面上的那一大堆像素。

比如，你绘制了一辆车并希望移动这辆车，若使用 CSS3 或 SVG 实现，那么可以找出在已放在场景图进行描述的车对象，通过修改它的 CSS 属性(变形、left 和 top)来移动它。无论向何处移动它，它都会一直停在那里，直至你再次移动它。若该车有一些车轮子对象，那么这些车轮会自动与车一起移动。

对于画布来说，你仅拥有车的一种表示方法，那就是被绘制在画布上的一堆像素。若试图移动它，最终得到的结果就是把画布上的一些像素从一个地方移到了另一个地方。为了移动车，需要清除画布并在正确的新位置上重绘每样东西，或通过绘制背景来擦除车的图像，然后在一个不同的位置上绘制车。若车有移动的车轮，那么需要绘制车，然后还要在车轮每次转动时绘制车轮。尽管画布赋予你绘制游戏中的每个像素的控制权，但它要求开发者或游戏引擎完成更多工作。

场景图还有一个好处，因为场景图知道元素的位置，所以浏览器可以处理触摸和鼠标事件，并可把事件路由给正确的对象。用画布实现游戏，需要编写逻辑来选择和锁定特定对象。

所有这些关于场景图的讨论都意味着，若你的游戏借助于直接交互来运作——换言之，用户与屏幕上的元素直接交互——那么使用一种拥有场景图的技术能把编程工作变得更容易一些。例如，在一个纸牌游戏中，用户单击且有可能会拖曳纸牌，那么这样的纸牌游戏就可受益于场景图，而一个仅使用左右箭头和跳跃按钮作为其控件的动作游戏则不会。

12.2.3　性能需求

最后一项考虑是性能，如前所述，所构建游戏的类型及其性能需求会影响到你的技术

决定。CSS3 和 SVG 往往能在较多设备上有着更好的表现，前提是一次不会移动过多元素。若在任何给定时刻都仅移动少数几个元素，那么 CSS3 表现尤其出色，它拥有硬件加速支持的过渡。有了硬件加速，对象会以一种高帧率平滑移动，你不必担心它们的动画问题。

另一方面，就快速滚动的平台动作游戏而言，在支持硬件加速的画布的浏览器中，相比于使用 CSS3，硬件加速的画布通常会表现得更好。

12.3　实现 DOM 支持

Quintus 可通过一个名为 Quintus.DOM 的模块把对 DOM 的支持添加到引擎中，该模块创建两个分别与 Sprite 和 Stage 类等价的基于 DOM 的类，所以这两个类被命名成 DOMSprite 和 DOMStage 也是意料之中的事。此外，setup 方法也有一个基于 DOM 的被命名为 setupDOM 的等价方法。

12.3.1　考虑 DOM 的特性

在 API 层面，Quintus 的 DOM 类的行为与它们的画布等价类很相似，但在内部，因为 DOM 提供了一个持久的场景图，所以类的行为差别很大。DOMStage 的步进(step)方法依然遍历每个精灵，但 DOMSprite 的步进(step)方法增加了额外的职责，那就是更新文档中用来表示精灵的元素。另外，因为浏览器负责实际的元素绘制工作，所以 draw 方法只包含了一个事件触发(trigger)调用。

精灵会被当作 元素添加到页面中，该 元素设置了宽度和高度，设置了背景图像，并使用通过元素的精灵地图计算得出的偏移位置来调整背景图像，所以帧的修改就相当于移动背景的位置。

接下来确定精灵的位置，这乍一看可能很容易，只需使用 left 和 top 这两个传统的 CSS 属性及绝对定位进行设置就可以了，但要想获取最佳性能，你得使用新的 CSS3 属性 transform(变形)，该属性受益于硬件加速的渲染。

因为 transform 属性前面要加上一个特定于常见宿主厂商的前缀，所以引擎查看浏览器的支持，并为精灵的定位生成一个方法，该方法在必要时可回退成提供 left 和 top 支持。

最后还有一个动画问题，在渲染每一帧时以手动方式动态显示一堆 DOM 元素，要做到这一点当然没问题，但在浏览器中，特别是在移动设备上，大规模使用这一做法所带来的工作量相当繁重。幸运的是，借助于过渡(transition)和关键帧动画(key frame animation)，CSS3 加入了对动画的支持。在这个例子中，过渡的使用有着很大意义，因为游戏可以只更新对象状态一次，然后依赖浏览器把属性从一个值渐变成另一个值。

遗憾的是，CSS3 的过渡和 transform 属性一样，也身陷厂商前缀困境，所以，你同样需要使用一个方法来检测出最好的做法，目的是为所有浏览器都提供支持。

12.3.2　自建 Quintus 的 DOM 模块

着手实现 Quintus.DOM 模块的第一步是创建一个新的名为 quintus_dom.js 的文件并打

开它，在其中输入代码清单 12-1 中的代码。

代码清单 12-1：自建的 Quintus DOM 模块

```
Quintus.DOM = function(Q) {
 Q.setupDOM = function(id,options) {
    options = options || {};
    id = id || "quintus";
    Q.el = $(_.isString(id) ? "#" + id : id);
    if(Q.el.length === 0) {
      Q.el = $("<div>")
                .attr('id',id)
                .css({width: 320, height:420 })
                .appendTo("body");
    }
    if(options.maximize) {
      var w = $(window).width();
      var h = $(window).height();
      Q.el.css({width:w,height:h});
    }
   Q.wrapper = Q.el
                .wrap("<div id='" + id + "_container'/>")
                .parent()
                .css({ width: Q.el.width(),
                       height: Q.el.height(),
                       margin: '0 auto' });
    Q.el.css({ position:'relative', overflow: 'hidden' });
    Q.width = Q.el.width();
    Q.height = Q.el.height();
    setTimeout(function() { window.scrollTo(0,1); }, 0);
    $(window).bind('orientationchange',function() {
      setTimeout(function() { window.scrollTo(0,1); }, 0);
    });
    return Q;
  };
};
```

这段代码创建了最初的模块包装器方法，此外，还创建了 setupDOM 方法，这是与基于画布的游戏的 Q.setup ()方法等价的方法，它使用一个现有的 DOM 元素，或是创建一个新的元素作为游戏的包装器。若 maximize 选项被传入，则方法重设容器的尺寸以适应屏幕的大小。接着，方法创建一个包装器容器来包含该元素，以便能在页面上居中显示该元素。它还设置了该元素的定位方式，允许该元素内部的元素在必要时采用绝对定位，它还把 overflow 属性设成 hidden，目的是防止游戏中的任何元素出现在游戏容器的外部。

12.3.3　创建一致的移动方法

在加入 DOMSprite 和 DOMStage 类来真正把一些东西绘制到屏幕上之前，我们需要先解决一个问题，那就是找出一个一致的定位方法。

这里的想法是，找出所涉及的浏览器所支持的表现最佳的方法，然后把该方法绑定到一个一致的方法名称上，这样余下的 DOM 支持就不必去了解元素确切的定位做法。CSS3 定义了对变形属性的支持，这取决于浏览器是否支持 translate(..)和 translate3d(..)这两个值，相比于使用传统的 left 和 top 属性，这些值能做到更有效地移动元素，尤其是 translate3d，能带来硬件加速的变形被应用于 DOM 元素这样的效果。

注意：使用变形的缺点是，不能保证存在某个变形属性是不带厂商前缀的，所以，在试图找出所支持的最好定位方法时，需要考虑每一个厂商前缀。若 translate3d 不被支持，则使用 translate，否则就采用简单老旧的 top 和 left 定位做法。

代码首先会检测 CSS3 变形是否获得了支持，若是，则检查 translate3d(该值触发 WebKit 的硬件加速支持)是否可用，若是，则调用一个名为 translate3DBuilder 的方法，该方法返回另一个使用恰当前缀进行了定制的方法。借助于闭包的强大功能，JavaScript 把创建返回方法的方法变成了一件很容易的事情。若 translate3d 不被支持，则调用 translateBuilder 返回一个执行非 3D 变形的方法。

将代码清单 12-2 中的代码添加到 quintus_dom.js 的末尾处，置于最后的结束花括号之前。

代码清单 12-2：检查移动支持

```
(function() {
  function translateBuilder(attribute) {
    return function(dom,x,y) {
      dom.style[attribute] =
      "translate(" + Math.floor(x) + "px," +
      Math.floor(y) + "px)";
    };
  }
  function translate3DBuilder(attribute) {
    return function(dom,x,y) {
      dom.style[attribute] =
      "translate3d(" + Math.floor(x) + "px," +
      Math.floor(y) + "px,0px)";
    };
  }
  function scaleBuilder(attribute) {
    return function(dom,scale) {
      dom.style[attribute + 'Origin'] = "0% 0%";
      dom.style[attribute] = "scale(" + scale + ")";
    };
  }
```

```
  function fallbackTranslate(dom,x,y) {
    dom.style.left = x + "px";
    dom.style.top = y + "px";
  }
  var has3d =  ('WebKitCSSMatrix' in window &&
                'm11' in new WebKitCSSMatrix());
  var dummyStyle = $("<div>")[0].style;
  var transformMethods = ['transform',
                          'webkitTransform',
                          'MozTransform',
                          'msTransform' ];
  for(var i=0;i<transformMethods.length;i++) {
    var transformName = transformMethods[i];
    if(!_.isUndefined(dummyStyle[transformName])) {
      if(has3d) {
        Q.positionDOM = translate3DBuilder(transformName);
      } else {
        Q.positionDOM = translateBuilder(transformName);
      }
      Q.scaleDOM = scaleBuilder(transformName);
      break;
    }
  }
  Q.positionDOM = Q.positionDOM || fallbackTranslate;
  Q.scaleDOM = Q.scaleDOM || function(scale) {};
})();
```

可以注意到，为防止函数污染 Quintus.DOM 主名称空间，整个表达式被包装在一个被立刻调用的函数表达式(IIFE，Immediately-Invoked Function Expression)中。这一做法使得整个段代码最终只会产生两个被添加到 Q 中的定义：Q.positionDOM 和 Q.scaleDOM。

这一代码清单的第一部分由三个返回方法的方法构成，若之前尚未见过许多此类做法，那么对你而言，这可能是一个相对难以理解的概念，所以，我们现在来深入了解一下其中的一个方法：

```
function translateBuilder(attribute) {
  return function(dom,x,y) {
    dom.style[attribute] =
    "translate(" + Math.floor(x) + "px," +
    Math.floor(y) + "px)";
  };
}
```

可以注意到，整个 translateBuilder 由一条返回一个函数的 return 语句构成。被返回的函数用到了被传入到最初方法中的 attribute 参数，这种做法在 JavaScript 中是允许的，因为该门语言支持闭包，闭包把函数的定义和最初定义函数时的作用域绑定在一起。在这之后，被返回的方法可用在代码库的任何地方，且在其被调用时可跟踪到特性(attribute)之前所绑定的值。

在定义了各种绑定方法后，代码创建一个 元素，并检查该元素的 style 特性，看看元素是否支持加上了各种不同厂商前缀的变形特性。

此外，它还进行了一个特定于 WebKit 的 3D 检查，若你想要一个更一般化的 translate3d 检查，那么可研究一下 Modernizr。因为大部分移动浏览器都是基于 WebKit 的，所以特定于 WebKit 的检查显然已满足了多数需求。

这段代码还创建了一个 Q.scaleDOM 方法，该方法用来执行一个缩放变形，因为缩放变形没有必需的 3D 等价方法，所以这一缩放方法的创建较为简单。

12.3.4　创建一致的过渡方法

在创建了一种一致的方式来尽可能高效地移动 DOM 元素之后，需要把整件事情再做一遍，这次的目的是创建一种简单做法来为支持过渡的浏览器添加过渡支持。

将代码清单 12-3 中的代码添加到 quintus_dom.js 的末尾处，置于最后的结束花括号之前。

代码清单 12-3：检查过渡支持

```
(function() {
  function transitionBuilder(attribute,prefix){
   return function(dom,property,sec,easing) {
     easing = easing || "";
     if(property == 'transform') {
       property = prefix + property;
     }
     sec = sec || "1s";
     dom.style[attribute] = property + " " + sec + " " + easing;
   };
 }
 // Dummy method
 function fallbackTransition() { }
 var dummyStyle = $("<div>")[0].style;
 var transitionMethods = ['transition',
                          'webkitTransition',
                          'MozTransition',
                          'msTransition' ];
 var prefixNames = [ '', '-webkit-', '-moz-', '-ms-' ];
 for(var i=0;i<transitionMethods.length;i++) {
   var transitionName = transitionMethods[i];
   var prefixName = prefixNames[i];
   if(!_.isUndefined(dummyStyle[transitionName])) {
     Q.transitionDOM = transitionBuilder(transitionName,prefixName);
     break;
   }
 }
 Q.transitionDOM = Q.transitionDOM || fallbackTransition;
})();
```

与上一节中的代码一样，这段代码遵循同一种模式，只是它的目标是创建一个能让开发者以一种一致的方式给属性添加过渡的方法。在这个例子中，若不存在内置的支持，则回退方法什么也没做。游戏依旧按预期运行，不过所有的过渡都是瞬间完成而非动画式的。

12.3.5　实现 DOM 精灵类

接下来是与画布的 Sprite 类等价的 DOM 类，该类实际上派生于 Q.Sprite 这一基类(故 Quintus.DOM 模块必须晚于 Quintus.Sprites 模块加载)。

如前所述，DOM 和画布精灵之间的主要区别在于，DOM 精灵自身不需要关心绘制，但它需要确保自己保持了与 DOM 元素属性的同步。

将代码清单 12-4 中的代码添加到 quintus_dom.js 的末尾处，置于最后的结束花括号之前。

代码清单 12-4：DOMSprite 类

```
Q.DOMSprite = Q.Sprite.extend({
  init: function(props) {
    this._super(props);
    this.el = $("<div>").css({
      width: this.p.w,
      height: this.p.h,
      zIndex: this.p.z || 0,
      position: 'absolute'
    });
    this.dom = this.el[0];
    this.rp = {};
    this.setImage();
    this.setTransform();
  },

  setImage: function() {
    var asset;
    if(this.sheet()) {
      asset = Q.asset(this.sheet().asset);
    } else {
      asset = this.asset();
    }
    if(asset) {
      this.dom.style.backgroundImage = "url(" + asset.src + ")";
    }
  },

  setTransform: function() {
    var p = this.p;
    var rp = this.rp;
    if(rp.frame !== p.frame) {
      if(p.sheet) {
        this.dom.style.backgroundPosition =
```

```
            (-this.sheet().fx(p.frame)) + "px " +
            (-this.sheet().fy(p.frame)) + "px";
        } else {
          this.dom.style.backgroundPosition = "0px 0px";
        }
        rp.frame = p.frame;
      }
      if(rp.x !== p.x || rp.y !== p.y) {
        Q.positionDOM(this.dom,p.x,p.y);
        rp.x = p.x;
        rp.y = p.y;
      }
    },

    hide: function() {
      this.dom.style.display = 'none';
    },

    show: function() {
      this.dom.style.display = 'block';
    },

    draw: function(ctx) {
      this.trigger('draw');
    },

    step: function(dt) {
      this.trigger('step',dt);
      this.setTransform();
    },

    destroy: function() {
      if(this.destroyed) return false;
      this._super();
      this.el.remove();
    }
  });
```

其中 init 方法负责创建实际的 DOM 元素 ，该元素包含了背景图像。它使用 jQuery 创建 并设置对象的尺寸和 zIndex。对于任意 DOM 操作来说，jQuery 的使用都会带来一点开销，所以，该方法还为那些可能每帧都会执行到的操作提取出了实际的 DOM 对象，并将其保存到 this.dom 中。

接下来创建一个名为 rp 的对象，该对象用来在 DOM 的一些真正属性被设置时存储这些属性。对 DOM 对象进行修改某种程度上在性能方面是代价高昂的，所以，取代每帧都更新这些属性的做法，精灵的 step 方法比较其属性哈希 p 和 rp 中的值，然后只在存有差异时才更新 DOM 对象。最后，init 方法调用 this.setImage()，该方法设置 div 的 backgroundImage；还调用 setTransform()，该方法设置元素在容器中的位置及背景图像的位

置(该位置对应于精灵地图中的帧)。

setImage 方法很简单，因为它所做的就是从精灵表或从资产中提取出背景图像来设置 backgroundImage 属性。

setTransform 较复杂些，你可看到，如前所述，它检查 p 和 rp 这两个对象之间在帧及 x 和 y 属性方面的差异，若帧需要更新，它就使用精灵表的辅助方法 fx 和 fy 计算位置，若该精灵对象并未附有一个精灵表，则将位置设成 0。

就位置而言，之前创建的 Q.positionDOM 方法用来以任何一种浏览器支持的最好做法设置位置。

因为 rp 对象被初始化成一个空对象，所以在首次运行 setTransform 时，帧和位置一定会被设置。

show 和 hide 方法把元素的 display 属性调整成 none 或 block，这会分别导致元素在页面上的隐藏或显示效果。

因为浏览器负责真正的对象绘制工作，所以 draw 方法仅是一个存根，触发一个 draw 事件。类似地，step 方法触发一个 step 事件，但此后它还调用了 setTransform，以防任何定位特性被修改。

最后，destroy 方法需要清除 DOM 对象及其保存的内部记录，所以，在让继承得来的方法完成它的工作之后，它调用 jQuery 的 remove()方法从页面中删除元素。

现在，Q.DOMSprite 类拥有了一个与 Q.Sprite 兼容的接口，不过，若试图使用标准的 Q.Stage 对象来跟踪 DOMSprite，你会大失所望，因为实际情况是，不会有任何东西出现在屏幕上。

12.3.6　创建 DOM 舞台类

为了能让 DOMSprite 对象正常发挥作用，我们需要把它们添加到一个有着自己的容器 DOM 元素以及知道如何把 DOM 元素添加到页面上的舞台对象中。因为那些可用于<canvas>标签的 CSS 缩放技巧不能被用在 DOM 元素上，所以在必要时引擎必须采用一种不同的缩放内容的机制。幸运的是，用来移动内容的 CSS3 变形式样同样支持缩放值。为便于在舞台对象中分开处理缩放和移动(在下一章的例子中，舞台对象会跟踪玩家)，DOMStage 类创建了一个独立的、用于缩放视图的包装器元素。

因为 Sprite 和 Stage 功能存放在不同的模块中，所以 DOMStage 类执行了一个检查，以防有人试图在没有使用场景和舞台模块的情况下使用 DOMSprite 构建游戏。

将代码清单 12-5 中的代码添加到 quintus_dom.js 文件的末尾处，同之前一样，置于最后的结束花括号之前。这段代码补全了基本的 DOM 精灵功能。

代码清单 12-5：DOMStage 类

```
if(Q.Stage) {
  Q.DOMStage = Q.Stage.extend({
    init: function(scene) {
      this.el = $("<div>").css({
```

```
        top:0,
        position:'relative'
      }).appendTo(Q.el);
      this.dom = this.el[0];
      this.wrapper = this.el.wrap('<div>').parent().css({
        position:'absolute',
        left:0,
        top:0
      });
      this.scale = 1;
      this.wrapper_dom = this.wrapper[0];
      this._super(scene);
    },

    insert: function(itm) {
      if(itm.dom) { this.dom.appendChild(itm.dom); };
      return this._super(itm);
    },

    destroy: function() {
      this.wrapper.remove();
      this._super();
    },

    rescale: function(scale) {
      this.scale = scale;
      Q.scaleDOM(this.wrapper_dom,scale);
    },

    centerOn: function(x,y) {
      this.x = Q.width/2/this.scale -  x;
      this.y = Q.height/2/this.scale - y;
      Q.positionDOM(this.dom,this.x,this.y);
    }
  });
}
```

Q.DOMStage 类继承了基本的画布舞台类，故正常的 Q.Stage 类中的所有方法，包括暂停和取消暂停等方法都是可用的。init 方法的任务是创建充当容器的 DOM 元素和被用来缩放容器的包装器元素，在完成该任务后，它仅调用父类的 init 方法来处理余下的事情。

类似地，insert 和 destroy 方法调用超类的相应方法，此外，insert 还把已添加的精灵的 DOM 元素追加到舞台容器元素中，destroy 方法还要确保从页面中删除 wrapper 元素，而这会删除该元素的所有子元素。

rescale 方法是新加入的方法，它使用本章之前创建的 Q.scaleDOM 方法来重新调整包装器的大小。与之前章节使用 CSS 缩放画布的作用一样，该方法将用来填充诸如平板电脑之类的较大设备的屏幕。centerOn 方法根据当前的缩放情况来重新定位舞台，还被用作跟踪玩家的摄像头。

12.3.7　替换画布的等价类

在编写完 Q.setupDOM 以及 Q.DOMSprite 和 Q.DOMStage 类后，基本功能的实现也就完成了。不过，DOM 等价类使用起来有些麻烦，例如，要使用 Q.DOMStage 类展现一个场景，需要重写 stageClass，即要写成

```
Q.stageScene(sceneObj,0,Q.DOMStage)
```

而非只写成

```
Q.stageScene(sceneObj);
```

若编写的是基于 DOM 的游戏，这会给代码带一些不必要的噪音，所以，为了简化代码，添加一个方法，该方法使用基于 DOM 的方法和类替换掉画布的等价方法和类。这样，可以在进行设置之前先调用该方法，目的是使 DOM 游戏的编写工作变得更容易一些。

与之前一样，将代码清单 12-6 中的代码添加到 quintus_dom.js 的末尾处。

代码清单 12-6：Q.domOnly 方法

```
Q.domOnly = function() {
  Q.Stage = Q.DOMStage;
  Q.setup = Q.setupDOM;
  Q.Sprite = Q.DOMSprite;
  return Q;
};
```

把对这一方法的调用链接到游戏开始之处的设置中，简化游戏从画布到 DOM 的转换。

12.3.8　测试 DOM 功能

在深入研究 CSS3 nethack 风格游戏的构建之前，先把上一章中那个简单的 Blockbreak 游戏转换成一个基于 DOM 的游戏，以此来测试一下 DOM 功能。

打开上一章的 blockbreak.html 文件(或把代码复制到一个新目录中)，然后添加一个<script>标签来加载刚才所写的 quintus_dom.js 文件。此外，你还需要把 style 标签修改成引用# quintus 而非仅引用 canvas 元素：

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Block Break</title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_input.js'></script>
    <script src='quintus_sprites.js'></script>
```

```
    <script src='quintus_scenes.js'></script>
    <script src='quintus_dom.js'></script>
    <script src='blockbreak.js'></script>
    <style>
      body { padding:0px; margin:0px; }
      #quintus { background-color:black; }
    </style>
  </head>
  <body>
  </body>
</html>
```

接下来，打开 blockbreak.js 文件，修改初始的设置调用，加入 DOM 模块并链入一个到 domOnly ()的调用。

```
$(function() {
  var Q = window.Q = Quintus()
                  .include('Input,Sprites,Scenes,DOM')
                  .domOnly()
                  .setup();
  Q.input.keyboardControls()
```

因为 domOnly()方法使用基于 DOM 的类替换掉了所有的画布等价类，所以代码无需再做其他改动。

在浏览器中启动游戏，照例，需要通过本地主机(localhost)来运行它，因为它要加载 JSON 数据。

除了移动设备屏幕底部的移动控件外，你看到的游戏与上一章中的没有不同之处。控制球拍的热点依然起作用，但因为没有编写代码来显示按钮，所以这些按钮没有出现在移动设备的屏幕上。

若在 Chrome 中审查页面，你会发现，所有的砖块及球拍和球实际上都是 DOM 元素，现在，你实现了一个可在 IE6 上运行的 HTML5 游戏！

12.4　小结

在本章中，你了解了如何借助性能优化的变形和过渡，使用 DOM 元素来构建游戏。此外，你还把对 DOM 元素的支持添加到 Quintus 引擎中，仅使用几行代码就把游戏例子 Blockbreak 转换成了一个基于 DOM 的游戏。

第13章

制作一个 CSS3 RPG 游戏

本章提要

- 创建滚动的区块地图
- 构建一个 RPG 游戏
- 添加敌人和增强能力的物件

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 13 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

13.1 引言

本章运用第 12 章中基于 DOM 的代码来构建一个简单的 nethack 风格的 RPG 游戏，该游戏要求提供区块背景(tiled background)支持，所以，引擎还要在下一节中添加一个名为 DOMTileMap 的类，该类正是为这一目的量身定做的。

13.2 创建滚动的区块地图

为创建一个 nethack 风格的游戏，引擎需要采用一种有效方式将一个 2D 区块地图(tile map)添加到游戏中。一种简化做法是仅在每个位置上加上一个绝对定位的精灵，这样做是可行的；不过，随着地图变得越来越大，这一做法会降低浏览器的速度，最后浏览器会变

得极其缓慢。所以，与之相反，你应该创建单个大型的精灵，该精灵可像一个单元那样四处移动，且可被当成单个元素对待。

13.2.1　了解性能问题

若你采用的是一个中等大小的地图，该地图可能有 50 个区块(tile)高和 50 个区块宽，那么这一地图最终会产生 2500 个精灵，每个精灵在每一帧中都要执行步进(step)方法；此外，每次对某个元素进行修改时，浏览器都需要重绘容器，这会显著降低持续更新的帧率。若你并未创建一个更高效的碰撞检测机制来替换遍历每个潜在对象这种做法，那么每次遍历就都需要测试每个移动的精灵。所有这些可能性导致的结果是，要不就得使用非常小的地图，要不就得接受糟糕的性能。

一种更好的解决方案是，创建单个区块地图精灵，该精灵包含了所有区块，这些区块可被当成单个实体步进和移动。因为各区块是不单独移动的，所以碰撞检测很简单，只需用位置除以每一区块的尺寸，得到某一区块位置，然后对这一区块位置进行检查即可。

13.2.2　实现 DOM 区块地图类

出于所有这些原因，引擎添加了一个名为 Q.DOMTileMap 的类，该类封装了所有这些功能。RPG 游戏的各个关卡都继承自该类，目的是加入其他一些游戏特定的功能。

为提高浏览器的性能，区块地图中的每一区块都被当成一个浮动的 DOM 元素添加进来。只要包含区块的 的宽度和高度设置无误，所有浮动元素最终都会出现在正确的位置上。

为了防止用户一下看到地牢的全貌，区块地图还支持个别区块的显示和隐藏(随着玩家的移动，区块背景会被一一揭示出来)。因为把显示方法设置成 none 会导致所有浮动区块的位移，所以该类把这一做法替换成仅是切换元素的可视性(visibility)属性。每次只要浏览器访问和影响 DOM，这些操作都会带来性能损失，所以该区块地图类在一个数据结构中记录下来哪些区块被显示和哪些被隐藏，仅在绝对必要时才修改 DOM 元素。

代码清单 13-1 给出了 Q.DOMTileMap 类的代码，把它添加到 quintus_dom.js 的末尾处，与之前一样，置于最后的结束花括号之前。

代码清单 13-1：Q.DOMTileMap 类

```
Q.DOMTileMap = Q.DOMSprite.extend({
  // Expects a sprite sheet, along with cols and rows properties
  init:function(props) {
    var sheet = Q.sheet(props.sheet);
    this._super(_(props).extend({
      w: props.cols * sheet.tilew,
      h: props.rows * sheet.tileh,
      tilew: sheet.tilew,
      tileh: sheet.tileh
    }));
    this.shown = [];
```

```
    this.domTiles = [];
  },

  setImage: function() { },

  setup: function(tiles,hide) {
    this.tiles = tiles;
    for(var y=0,height=tiles.length;y<height;y++) {
      this.domTiles.push([]);
      this.shown.push([]);
      for(var x=0,width=tiles[0].length;x<width;x++) {
        var domTile = this._addTile(tiles[y][x]);
        if(hide) { domTile.style.visibility = 'hidden'; }
        this.shown.push(hide ? false : true);
        this.domTiles[y].push(domTile);
      }
    }
  },

  _addTile: function(frame) {
    var p = this.p;
    var div = document.createElement('div');
    div.style.width = p.tilew + "px";
    div.style.height = p.tileh + "px";
    div.style.styleFloat = div.style.cssFloat = 'left';
    this._setTile(div,frame);
    this.dom.appendChild(div);
    return div;
  },

  _setTile: function(dom,frame) {
    var asset = Q.asset(this.sheet().asset);
    dom.style.backgroundImage = "url(" + asset.src + ")";
    dom.style.backgroundPosition = (-this.sheet().fx(frame)) +"px "
+ (-this.sheet().fy(frame)) + "px";
  },

  validTile: function(x,y) {
    return (y >= 0 && y < this.p.rows) &&
           (x >= 0 && x < this.p.cols);
  },

  get: function(x,y) { return this.validTile(x,y) ?
                          this.tiles[y][x] : null; },

  getDom: function(x,y) { return this.validTile(x,y) ?
                                  this.domTiles[y][x] : null; },
  set: function(x,y,frame) {
    if(!this.validTile(x,y)) return;
    this.tiles[y][x] = frame;
```

```
    var domTile = this.getDom(x,y);
    this._setFile(domTile,frame);
  },

  show: function(x,y) {
    if(!this.validTile(x,y)) return;
    if(this.shown[y][x]) return;
    this.getDom(x,y).style.visibility = 'visible';
    this.shown[y][x] = true;
  },

  hide: function(x,y) {
    if(!this.validTile(x,y)) return;
    if(!this.shown[y][x]) return;
    this.getDom(x,y).style.visibility = 'hidden';
    this.shown[y][x] = false;
  }
});
```

该类有些复杂，所以将它分成三大块。第一块是 init 方法，该方法设置区块地图的属性。

```
Q.DOMTileMap = Q.DOMSprite.extend({
  init:function(props) {
    var sheet = Q.sheet(props.sheet);
    this._super(_(props).extend({
      w: props.cols * sheet.tilew,
      h: props.rows * sheet.tileh,
      tilew: sheet.tilew,
      tileh: sheet.tileh
    }));
    this.shown = [];
    this.domTiles = [];
  },
  setImage: function() { },
```

init 方法从传入的属性中提取精灵表及其行数、列数，以及一些区块数据等，然后使用这些数据来计算精灵的宽度和高度。它调用 this._super ()方法让 DOMSprite 类完成初始化和实际 DOM 元素的创建，DOMSprite 的 init 方法还会调用 setImage 来设置精灵的背景图像，不过因为 DOMTileMap 元素不需要背景图像，所以该方法被重写成一个空的存根方法。

接下来是三个用来接收一个二维的区块帧数组作为参数并创建区块地图的方法：

```
  setup: function(tiles,hide) {
    this.tiles = tiles;
    for(var y=0,height=tiles.length;y<height;y++) {
      this.domTiles.push([]);
      this.shown.push([]);
      for(var x=0,width=tiles[0].length;x<width;x++) {
```

```
        var domTile = this._addTile(tiles[y][x]);
        if(hide) { domTile.style.visibility = 'hidden'; }
        this.shown.push(hide ? false : true);
        this.domTiles[y].push(domTile);
      }
    }
  },

  _addTile: function(frame) {
    var p = this.p;
    var div = document.createElement('div');
    div.style.width = p.tilew + "px";
    div.style.height = p.tileh + "px";
    div.style.styleFloat = div.style.cssFloat = 'left';
    this._setTile(div,frame);
    this.dom.appendChild(div);
    return div;
  },

  _setTile: function(dom,frame) {
    var asset = Q.asset(this.sheet().asset);
    dom.style.backgroundImage = "url(" + asset.src + ")";
    dom.style.backgroundPosition = (-this.sheet().fx(frame)) +"px " +
                                   (-this.sheet().fy(frame)) + "px";
  },
```

setup 方法接收一个二维数组，调用内部的辅助方法_addTile 创建 DOM 元素，并使用适当的值来更新 domTiles 和 shown 数组。domTiles 数组是一个与 tiles 数组相同的二维数组，唯一的不同之处是它指向实际的 DOM 元素，这样这些 DOM 元素就可被操纵。shown 数组是一个存放布尔值的二维数组，它记录下哪些区块是可见的，哪些是被隐藏起来的。

_addTile 方法接收一个帧为参数，并返回一个被设置成该帧的 DOM 元素。因为要创建的 DOM 元素有许多，所以引擎采用原生的 document.createElement 方法而非通常所用的 jQuery 方法来尽量赢得一些速度优势。float 属性的设置也有些棘手，因为在使用 JavaScript 访问该属性时，不同浏览器的引用方式不同。该方法没有试图找出正确的引用方式，而是采取了一种对两个选项都进行设置的快捷做法。此外，它还调用_setTile，以此作为一种快捷方式，基于帧正确设置背景图像和背景图像的位置。

该类的最后一块内容检索并更新区块地图中的区块：

```
validTile: function(x,y) {
  return (y >= 0 && y < this.p.rows) &&
         (x >= 0 && x < this.p.cols);
},
get: function(x,y) { return this.validTile(x,y) ?
                            this.tiles[y][x] : null; },
getDom: function(x,y) { return this.validTile(x,y) ?
                               this.domTiles[y][x] : null; },
set: function(x,y,frame) {
```

```
    var domTile = this.getDom(x,y);
    if(!domTile) return;
    this.tiles[y][x] = frame;
    this._setFile(domTile,frame);
  },

  show: function(x,y) {
    var domTile = this.getDom(x,y);
    if(!domTile) return;
    if(this.shown[y][x]) return;
    domTile.style.visibility = 'visible';
    this.shown[y][x] = true;
  },

  hide: function(x,y) {
    var domTile = this.getDom(x,y);
    if(!domTile) return;
    if(!this.shown[y][x]) return;
    domTile.style.visibility = 'hidden';
    this.shown[y][x] = false;
  }
```

为了保持各个游戏代码的简单性，在游戏调用上述任一区块操纵例程时，若将无效的区块位置数据传入例程中，引擎应在处理失败后保持沉默，这样做的目的是允许游戏在不必进行边界检查的情况下隐藏或显示地图外部的区块。为方便地做到这一点，validTile 方法根据之前传入的行列范围来检查被传入的 x 和 y 位置，若元素超出边界，则返回 false 值。

get 和 getDOM 方法采用这一做法来防止对 tiles 和 domTiles 数组进行不正确索引并因此导致异常。set 方法的作用是允许游戏更新某一特定区块的帧，这可用来实现动画或修改区块地图的状态(例如，在某扇门被打开时)。show 和 hide 方法切换单一区块的可视性。

13.3　构建 RPG 游戏

编写完所有这些代码后，现在是时候把注意力转向真正的 RPG 游戏的构建上来了，这可是本章标题的卖点。这里的基本设计方案是，游戏加载一个包含了关卡的 ASCII 地图的文本文件，地图中有散落各处的怪物和战利品；然后，游戏把该文本文件转换成一个区块地图和一组玩家将与其交互的精灵。

13.3.1　创建 HTML 文件

一如往常，首先创建必需的 HTML 包装器文件来存放游戏，创建一个新的名为 rpg.html 的文件，在其中输入代码清单 13-2 中的代码。

代码清单 13-2：RPG 游戏的包装器文件

```
<!DOCTYPE HTML>
```

```
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
minimum-scale=1.0, maximum-scale=1.0"/>
    <title>RPG</title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_input.js'></script>
    <script src='quintus_sprites.js'></script>
    <script src='quintus_scenes.js'></script>
    <script src='quintus_dom.js'></script>
    <script src='rpg.js'></script>
    <style>
      * { padding:0px; margin:0px; }
      #quintus { background-color:black; }
    </style>
  </head>
  <body>
  </body>
</html>
```

同之前一样，该文件几乎是空的，仅包含了几个重置样式和一些加载游戏的 script 标签。

13.3.2　设置游戏

为启动游戏，接下来为游戏创建一个基本结构，该结构设置窗口界面并加载一些艺术资产。

不必重造 nethack(也即 rogue-like)区块集(tileset)，因为一些热心人已在互联网上发布了一些属于公共域的区块集：http://rltiles.sourceforge.net/，可使用它们来构建游戏。

本章的 RPG 游戏使用三个来自 RLTiles 的文件搭建运行环境，为了很好地融入到游戏中，这些区块需要执行一些背景消除工作，除此之外应该说它们的效果还是很不错的。本章已在文件的 images/目录下准备好了所需的图像文件，每个图像文件都包含了一大组 32x32 像素的图像。目前这一游戏尚未打算好好利用其中的大多数区块，而仅是出于视觉效果，从中随机抽取一些敌人和物件。

有了这三个文件，现在可以启动游戏的开发了，创建之前的 HTML 包装器文件用到的 rpg.js 文件，将代码清单 13-3 中的样板代码置于其中。另外，在游戏的 data/子目录下有一个名为 level1.txt 的关卡数据文本文件，游戏的运行还需要用到该文件。在目前，该文件中的内容无关紧要，可从本章的文件资产中复制一个，或创建一个并把它保存成空文件。

代码清单 13-3：启动 RPG 游戏

```
$(function() {
  var Q = window.Q = Quintus()
```

```
                    .include('Input,Sprites,Scenes,DOM')
                    .domOnly()
                    .setup('quintus',{ maximize: true });
var tileSize = 32;
var TILE = {
  WALL: 10,
  FLOOR: 30,
  STAIRS: 45
};
var impassableTiles = {
  10: true
};
Q.input.keyboardControls();
Q.input.joypadControls({zone: Q.width});
Q.load(['characters.png',
        'dungeon.png',
        'items.png',
        'level1.txt'], function() {

  Q.sheet('characters', 'characters.png',
          { tilew: tileSize, tileh: tileSize });

  Q.sheet('tiles', 'dungeon.png',
          { tilew: tileSize, tileh: tileSize });

  Q.sheet('items', 'items.png',
          { tilew: tileSize, tileh: tileSize });

  Q.scene('level1',new Q.Scene(function(stage) {
    if(Q.width > 600 || Q.height > 600) {
      stage.rescale(2);
    }
    alert("Loaded!");

  }));
  Q.stageScene('level1');
});
});
```

这段代码设置 Q 对象，接着设置几个之后会用到的全局变量，然后设置默认的键盘控件和一个占整屏宽度的游戏手柄。

接下来，代码加载四个资产——三个图像和一个关卡数据文本文件——并在加载完毕后设置三个精灵表。

接着，代码创建一个新的名为 level1 的场景，现在，该场景所做的事情不多，仅是在浏览器的宽度或高度大于 600 的情况下重新调整游戏画面大小。这一做法能够让 iPad 和桌面浏览器得到一个放大了的游戏视图(就支持变形的浏览器而言，在较旧版本的浏览器中，所有内容看起来都显得更小一些)。

最后，游戏调用 Q.stageScene ("level1")把第一个场景加载到游戏中，若一切按计划顺利进行，你应会看到一个写着“Loaded!”的提示框出现在页面上。

13.3.3　添加区块地图

现在是时候把一些区块添加到面板上了，为此，游戏通过继承 DOMTileMap 类创建了一个类，该类能够接收关卡数据文本文件资产为参数，并能够把它转换成一些 DOMTileMap 类能够用得着的内容。

为了简化关卡的创建，关卡格式使用某种 ASCII 文件表示，其中 X 代表墙，点号(.)代表走廊和房间，诸如怪物和宝物一类的附加精灵则用其他一些字母标出。游戏能够根据这些数据算出地图的宽度和高度。见图 13-1，这是格式的一个例子，说明了关卡看起来的可能样子。

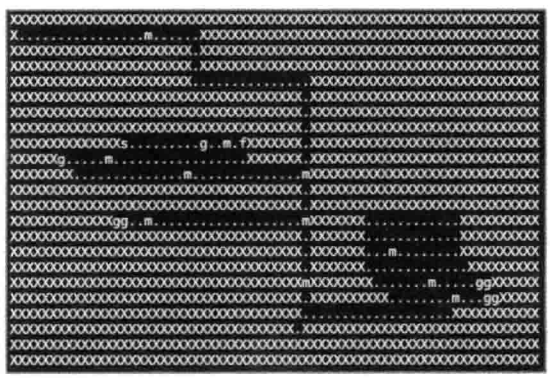


图 13-1　关卡文本文件

再次打开 rpg.js 文件，将代码清单 13-4 中的 Q.Level 类定义添加到文件中，置于 Q.load 方法之前。

代码清单 13-4：Q.Level 类

```
Q.Level = Q.DOMTileMap.extend({
  legend: {
    "X": "wall",
    ".": "floor"
  },

  init:function(asset,stage) {
    this.stage = stage;
    this.level = [];
    this.sprites = [];
    var data = Q.asset(asset);
    this.extra = [];
    _.each(data.split("\n"),function(row) {
      var columns = row.split("");
      if(columns.length > 1) {
        this.level.push(columns);
```

```
        this.sprites.push([]);
      }
    },this);

    this._super({
      cols:this.level[0].length,
      rows:this.level.length,
      sheet: 'tiles'
    });

    var tiles =[];
    for(var y=0;y<this.level.length;y++) {
      tiles[y] = [];
      for(var x =0;x<this.level[0].length;x++) {
        var square = this.level[y][x],
            frame = null,
            method = this.legend[square] || "wall";

        frame = this[method](x*tileSize,y*tileSize);
        tiles[y].push(frame);
      }
    }
    this.setup(tiles,false);
  },

  insert: function(sprite) {
    this.stage.insert(sprite);
    this.sprites[sprite.p.tileY][sprite.p.tileX] = sprite;
    return sprite;
  },

  wall: function(x,y) { return TILE.WALL; },
  floor: function(x,y) { return TILE.FLOOR; }
});
```

现在该类主要用于修改 init 方法，让该方法接收一个资产和一个舞台对象作为参数，然后查明 legend 属性中的每种区块的处理方式，并使用该方式来设置区块地图。目前该方法只支持两种区块类型：floor 和 wall；每种类型仅用来控制区块的外观。将来该类还会添加一些功能来记录每一区块位置上的精灵，所以，作为这一功能的先期准备，init 方法使用与主区块数据同样的行数来创建 this.sprites 数组。此外，该类还提供了一个把精灵插入到舞台对象中的辅助方法，该方法把这些精灵添加到 this.sprites 数组中。

要测试该类，删除提示框，把 level1 场景的 Q.scene 定义内部的代码修改成如下内容：

```
Q.scene('level1',new Q.Scene(function(stage) {
    if(Q.width > 600 || Q.height > 600) {
      stage.rescale(2);
    }
    stage.level = stage.insert(
```

```
                    new Q.Level("level1.txt",stage)
                  );
});
Q.stageScene('level1');
```

若加载游戏，你应会看到关卡已被渲染到屏幕上。因为 level1.txt 文件经由 Ajax 进行加载，所以你必须确保自己是通过本地主机(localhost)而非使用 file://这样的 URL 地址来加载页面。

13.3.4　创建一些有用的组件

区块环境中的精灵需要表现出与 2D 平台动作游戏中的精灵不同的行为，它们应以 tileSize 大小的递增距离来在面板上四处移动，避免穿墙或是穿越彼此的行为，并且要保持关卡精灵数组的更新，因为它们会在地牢中四处走动。

以一种可重用方式把这些功能封装起来，这是一个很不错的选择。一种做法是创建一个 TileSprite 基类，然后让所有精灵都继承该类，但若希望在其他地方重用精灵，你可能会发现这种做法很繁琐。另一种处理这一问题的方法是创建一个组件，该组件为所有精灵增加区块感知的定位和移动。我们选择后一种做法。

该组件挂接 step 事件，查看精灵的 dx 和 dy 属性(direction x 和 direction y 的缩写)，确定精灵是否试图朝任一方向移动，若是，则检查是否有其他区块或精灵挡在前面，否则移动该精灵；若有其他精灵挡在前面，则触发一个事件并传入被撞上的精灵，以此来让该精灵知道它碰到了一些东西。

再次打开 rpg.js 文件，将代码清单 13-5 中的 tiled 组件定义添加到 Q.Level 定义的前面。

代码清单 13-5：tiled 组件

```
Q.register('tiled', {
  added:function() {
    var p = this.entity.p;
    _(p).extend({
      wait: 0,
      delay: 0.15,
      tileX: Math.floor(p.x / tileSize),
      tileY: Math.floor(p.y / tileSize),
      dx: 0,
      dy: 0
    });
    this.direction = {};
    this.entity.bind('step',this,'move');
    this.entity.bind('removed',this,'removed');
  },

  move: function(dt) {
    var p =this.entity.p,
        stage = this.entity.parent;
```

```
    if(p.wait <= 0) {
      var destX = p.tileX, destY = p.tileY;

      if(p.attacking) {
         this.entity.trigger('attack',this.direction);
      } else if(p.dx || p.dy) {
        if(p.dx > 0) { destX += 1; }
        else if(p.dx < 0) { destX -= 1 };
        if(p.dy > 0) { destY += 1; }
        else if(p.dy < 0) { destY -= 1; }

        if(!impassableTiles[stage.level.get(destX,destY)]) {
          var sprite = stage.level.sprites[destY][destX];
          this.direction.dx = destX - p.tileX;
          this.direction.dy = destY - p.tileY;
          this.direction.sprite = sprite;
          if(!sprite) {
            this.moved(destX,destY);
            this.setPosition();
            p.wait = p.delay;
          } else {
            p.wait = p.delay * 2;
          }
          this.entity.trigger(sprite ? 'hit' : 'moved',
                              this.direction);
        }
      }
    } else {
      p.wait -= dt;
    }
  },
  setPosition: function() {
    var p =this.entity.p;
    p.x = p.tileX * tileSize;
    p.y = p.tileY * tileSize;
  },
  moved: function(destX,destY) {
    var stage = this.entity.parent;
    var p =this.entity.p;
    stage.level.sprites[p.tileY][p.tileX] = null;
    p.tileX = destX;
    p.tileY = destY;
    stage.level.sprites[p.tileY][p.tileX] = this.entity;
  },
  removed: function() {
    var stage = this.entity.parent;
    var p =this.entity.p;
    stage.level.sprites[p.tileY][p.tileX] = null;
  }
});
```

这是本书用到的首批重要的 Quintus 组件之一，所以该组件很值得我们深入研究一番，了解一下它都实现了哪些功能。

不知你是否还记得，组件开始之处的 added()方法在组件最初被添加到游戏对象中时被调用。通常情况下，该方法主要完成两件事情：扩展游戏对象的属性哈希和绑定一些对象事件。在此处，组件加入了一些属性来获取当前区块位置、移动延迟和移动方向；接着，它绑定了 step 和 removed 事件。

step 事件处理程序对应于 move 方法，该方法是其中最复杂的一个，若对象在两次步进之间没有等待，那么它负责移动对象(等待情况被记录在 wait 属性中)。此外，它还检查攻击(attacking)属性，该属性在对象攻击另一个对象时被用到。

最后执行的主要检查首先确定目标 x 和 y 位置，接着查看是否有任何不可通过的区块挡在前面，然后查看是否有精灵挡在前面，若没有精灵，则使用两个辅助方法——moved 和 setPosition——移动对象，并重置延迟，以防对象移动得过快。若并非不可通过的区块挡路，则重置延迟，并把精灵添加到方向(direction)对象中。最后，触发一个 hit 事件或一个 moved 事件，并把方向对象中的数据作为参数传入。

方向对象是与每个被触发事件一起传递的事件对象，出于节省内存的目的，它在多个调用之间被重用。

辅助方法 setPosition 用于基于区块位置更新对象的 x 和 y 位置，方法 moved 保持关卡的 sprites 数组的同步，目的是简化区块层面的碰撞检查。

趁现在还处在创建组件的状态中，还需要往代码库中快速添加另一个名为 transition 的组件。将代码清单 13-6 中的代码添加到 tiled 组件定义的后面。

代码清单 13-6：transition 组件

```
Q.register('transition', {
  added: function() {
    Q.transitionDOM(this.entity.dom,'transform','0.25s');
  }
});
```

这个简单的组件仅是增加对变形的过渡支持，在整个区块上的对象被移动时支持平滑移动。

接下来的这个组件，你应该直接把它添加到 transition 组件的后面，是一个跟踪用户在关卡中的位置的摄像头(camera)组件。该组件的实现很简单，仅需绑定玩家的 moved 事件并告知舞台对象在玩家移动时居中显示玩家就可以了。组件的实现代码如代码清单 13-7 所示。

代码清单 13-7：camera 组件

```
Q.register('camera', {
  added: function() {
    this.entity.bind('moved',this,'track');
```

```
  },
  track: function() {
    var p = this.entity.p,
        stage = this.entity.parent;
    stage.centerOn(p.x, p.y);
  }
});
```

该组件能从实体中提取舞台对象，实体把舞台对象视作自己的父级；然后，组件简单调用 centerOn 方法来调整视图。

现在到了最后一个组件，该组件需要提取用户的输入，可将该组件添加到 camera 组件定义的后面。该组件被命名为 player_input，它仅查看输入并设置(之前用在 tiled 组件中的)p.dx 和 p.dy 变量，目的是指明玩家试图移动的方向。代码清单 13-8 列出了该组件的代码。

代码清单 13-8：玩家输入组件

```
Q.register('player_input', {
  added: function() {
    this.entity.bind('step',this,'input');
  },
  input: function() {
    var p = this.entity.p;
    if(Q.inputs['left']) { p.dx = -1 }
    else if(Q.inputs['right']) { p.dx = 1;}
    else { p.dx = 0;}
    if(Q.inputs['up']) { p.dy = -1 }
    else if(Q.inputs['down']) { p.dy = 1;}
    else { p.dy = 0;}
  }
});
```

因为输入系统已被分离，所以无论输入来自游戏手柄还是键盘，player_input 组件都不必关心输入的来源。

13.3.5　添加玩家

在编写完这些组件的编写后，接下来就是添加玩家类了。该玩家类将代表在游戏中四处移动的玩家，并封装玩家的所有功能。你所需要做的就是子类化 Q.Sprite 类，在构造函数中设置一些基本属性，以及加入一些上一节中构建的组件。

将代码清单 13-9 所示的 Q.Player 类添加到代码文件中，放在之前定义的组件之后。

代码清单 13-9：Player 类

```
Q.Player = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'characters',
```

```
        frame: 65,
        wait: 0,
        z: 10,
        attack: 5,
        health: 40,
        maxHealth: 40,
        gold: 0,
        xp: 0
      }).extend(props));
      this.add('player_input, tiled, camera, transition');
    }
  });
```

这个玩家类定义了一些不会被立刻用到的初始属性，如 health、maxHealth、gold 和 xp 等，但这些属性在本章后面的内容中都会用到。不过，如你所见，不必创建很深的类层次结构，组件的使用使得更便于将可重用功能块添加到精灵对象中。

为在屏幕上显示玩家，在代码文件末尾处的 level1 场景的内部，把玩家对象加入到舞台对象中，同时把 transition 组件也添加到舞台对象中，这样舞台对象就可以流畅地跟踪玩家。

```
Q.scene('level1',new Q.Scene(function(stage) {
 if(Q.width > 600 || Q.height > 600) {
   stage.rescale(2);
 }
 stage.level = stage.insert(
                 new Q.Level("level1.txt",stage)
               );
 stage.add('transition');
 var player = stage.insert(new Q.Player({ x: 1 * tileSize,
                                          y: 1 * tileSize }));
 player.camera.track();
 player.bind('removed',stage,function() {
  Q.stageScene('level1');
 });
}));
Q.stageScene('level1');
```

在把这几部分代码都添加到代码文件后，加载 rpg.html，玩家应能够响应键盘的箭头键或是游戏手柄，在舞台上四处移动(游戏手柄实际上是不可见的，不过若四处拖动手指，玩家角色会做出响应)。

13.3.6　添加迷雾、敌人和战利品

除了最不愿意承担风险的冒险家，对于其他任何人来说，在一个空荡荡的地牢中到处转悠不见得有多好玩。为了把事件变得更有趣一些，现在是时候加入一些要与之打斗的敌人和一些可收归己有的战利品了。

第一步是为所需的每种不同类型的对象都添加一个精灵类，该例创建了三种不同类型

的对象：

- 玩家攻击的敌人
- 玩家捡到的战利品
- 供玩家补充健康值的健康泉

这三种对象的类都是简短的精灵类，都拥有一个可以指示对象如何与玩家交互的 interact 方法。

将代码清单 13-10 所示的三种精灵类型添加到 rpg.js 中，置于 Q.Player 类的后面。

代码清单 13-10：Enemy、Fountain 和 Loot 类

```
Q.Enemy = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'characters',
      z: 10,
      health: 10,
      maxHealth: 10,
      damage: 5,
      xp: 100
    }).extend(props));
    this.add('tiled, transition');
    this.bind('interact',this,'interact');
    this.hide();
  },

  interact: function(data) {
    this.p.health -= data.damage;
    if(this.p.health <= 0) {
      this.destroy();
      data.source.trigger('xp',this.p.xp);
    } else {
      var damage = Math.round(Math.random() * this.p.damage);
      data.source.trigger('interact',
                          { source: this, damage: damage });
    }
    this.trigger('health',this);
  }
});

Q.Fountain= Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'tiles',
      frame: 71,
      z: 10,
      power: 10
    }).extend(props));
    this.add('tiled');
```

```
    this.bind('interact',this,'interact');
    this.hide();
  },

  interact: function(data) {
    data.source.trigger('heal',{ amount: this.p.power });
  }
});

Q.Loot = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'items',
      frame: Math.floor(Math.random() * 30 * 9) + 150,
      z: 10,
      gold: Math.floor(Math.random() * 100)
    }).extend(props));
    this.add('tiled');
    this.bind('interact',this,'interact');
    this.hide();
  },
  interact: function(data) {
    data.source.trigger('gold',this.p.gold);
    this.destroy();
  }
});
```

在每个类中，精灵都包含一个充当构造函数的 init 方法，该方法设置对象的属性并绑定之前提到的 interact 方法。因为每种精灵都可以在玩家靠近时选择取消隐藏自身，所以在开始时它们都是被隐藏起来的。

每种元素的 interact 方法中都有一些有趣行为发生，以 Q.Enemy 类为例，当用户与敌人交互时，会认为用户在攻击敌人(这看起来是合理的，难道你通常会向敌人问路吗？)该方法根据传入的数值来降低敌人的健康值，并根据健康值或让敌人直接死去，或让敌人回击玩家。若敌人已死，则触发玩家的一个 xp(experience point，经验值)事件，该事件可用来升级玩家。此外，这一攻击交互还会触发敌人的一个 health 事件，下一节将使用这一事件来更新血槽(health bar)。

Q.Fountain 类的 interact 方法调用玩家的 heal 方法来恢复玩家的一些健康值。最后是 Q.Loot 类，该类触发玩家的一个 gold 事件。

通过在每个类中绑定事件而非调用目的方的特定方法，源方和接受方得以解耦，这意味着一方不必了另一方的任何事情，彼此的依赖性得以降低。此外，这还意味着组件可以轻松通过挂接系统来把其他一些功能添加到核心的精灵行为中。

就与其他元素交互的玩家类而言，该类需要被扩展成可处理碰撞和攻击事件，此外，它还需要使用一个 interact 方法来处理被敌人攻击的情况。

修改 Q.Player 类，修改后的代码如代码清单 13-11 所示。

代码清单 13-11：修改后的玩家精灵

```
Q.Player = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'characters',
      frame: 65,
      wait: 0,
      z: 10,
      attack: 5,
      health: 40,
      maxHealth: 40,
      gold: 0,
      xp: 0
    }).extend(props));
    this.add('player_input, tiled, camera, transition');
    this.bind('hit',this,'collision');
    this.bind('attack',this,'attack');
    this.bind('interact',this,'interact');
    this.bind('heal',this,'heal');
  },

  collision: function(data) {
    this.p.x += data.dx * tileSize/2;
    this.p.y += data.dy * tileSize/2;
    this.p.attacking = true;
  },

  attack: function(data) {
    var damage = Math.round(Math.random() * this.p.attack);
    data.sprite.trigger('interact',
                        { source: this, damage: damage });
    this.p.attacking = false;
    this.tiled.setPosition();
  },

  interact: function(data) {
    this.p.health -= data.damage;
    if(this.p.health <= 0) {
      this.destroy();
    }
    this.trigger('health');
  },

  heal: function(data) {
    this.p.health += data.amount;
    if(this.p.health > this.p.maxHealth) {
      this.p.health = this.p.maxHealth;
    }
```

```
      this.trigger('health');
    }
  });
```

Player 类在遇上某个精灵时会发送一个初始的碰撞事件(这由之前介绍的 tiled 组件处理)。它对该事件的回应是，移动半个区块远的距离，进入到所涉及的区块中，并把 attacking 属性设置成 true 值。然后，经过一个很短的时间，tiled 组件发送一个 attack 事件。接收到该事件后，Player 类根据 attack 属性计算出一个随机的伤害值，然后将 interact 事件发送给它所碰到的任何精灵。若该精灵是一个敌人，则该敌人承受这一伤害，结果是死亡或者反击，若是反击，则触发玩家的一个 interact 事件。

heal 方法则与此相反，它根据一个事先设置的数值来提升玩家的健康值。heal 和 interact 这两个方法最后都会触发一个 health 事件来指明玩家的健康值已发生了变化。

13.3.7　使用精灵扩展区块地图

在更新了玩家类并创建了其他一些精灵类之后，剩下要做的事情就是更新 Q.Level 类，把一些精灵添加到面板上，置于所需的地方。另外，目前大型浏览器的视口过大，玩家能看到的地牢面积过多。一种更好的选择是缩窄视口，这样就可以只在玩家接近地图区块时才把它们暴露出来。不知你是否还记得，Q.DOMTileMap 类有一个打开或关闭个别区块可视性的选项，现在，可随着玩家的四处走动，使用该属性来慢慢暴露地牢的面貌。

要把其他一些精灵放入到区块地图中，拥有 legend 属性的 Q.Level 类需要加以扩展，这样才能提供一些新的不同类型区块的创建方法。这些新方法除了返回适当的区块外，每个方法还会把一个对象插入舞台对象中。将代码清单 13-12 中突出显示的那部分代码添加到 Q.Level 类中。

代码清单 13-12：最终的 Level 类

```
Q.Level = Q.DOMTileMap.extend({
  legend: {
    "X": "wall",
    ".": "floor",
    "m": "monster",
    "f": "fountain",
    "d": "door",
    "g": "gold",
    "s": "stairs"
  },
  init:function(asset,stage) {
    this.stage = stage;
    this.level = [];
    this.sprites = [];
    var data = Q.asset(asset);
    this.extra = [];
    _.each(data.split("\n"),function(row) {
      var columns = row.split("");
```

```
    if(columns.length > 1) {
      this.level.push(columns);
      this.sprites.push([]);
    }
  },this);

  this._super({
    cols:this.level[0].length,
    rows:this.level.length,
    sheet: 'tiles'
  })

  var tiles =[];
  for(var y=0;y<this.level.length;y++) {
    tiles[y] = [];
    for(var x =0;x<this.level[0].length;x++) {
      var square = this.level[y][x],
          frame = null,
          method = this.legend[square] || "wall";

      frame = this[method](x*tileSize,y*tileSize);
      tiles[y].push(frame);
    }
  }
  this.setup(tiles,true);
},

insert: function(sprite) {
  this.stage.insert(sprite);
  this.sprites[sprite.p.tileY][sprite.p.tileX] = sprite;
  return sprite;
},

unfog: function(x,y) {
  for(var sx=x-2,ex=x+2;sx<=ex;sx++) {
    for(var sy=y-2,ey=y+2;sy<=ey;sy++) {
      this.show(sx,sy);
      if(this.validTile(sx,sy) && this.sprites[sy][sx]) {
        this.sprites[sy][sx].show();
      }
    }
  }
},
wall: function(x,y) { return TILE.WALL; },

floor: function(x,y) { return TILE.FLOOR; },

stairs: function(x,y) {
  this.startX = x;
  this.startY = y;
  return TILE.STAIRS;
```

```
    },

    gold: function(x,y) {
      this.insert(new Q.Loot({ x:x, y:y }));
      return TILE.FLOOR;
    },

    fountain: function(x,y) {
      this.insert(new Q.Fountain({ x:x, y:y }));
      return TILE.FLOOR;
    },

    monster: function(x,y) {
      var frame = Math.floor(Math.random()*64);
      this.insert(new Q.Enemy({ x:x, y:y, frame:frame }));
      return TILE.FLOOR;
    }

  });
```

虽然这段代码很长，但每个精灵方法都是一样的，它仅创建一个所需类型的精灵并将其添加到舞台对象中，然后返回位于该精灵下的地板区块。其中的 stairs 精灵比较特殊，因为它标记了玩家在开始本关游戏时所处的位置，这一开始位置存储在 startX 和 startY 属性中。

值得一提的还有 unfog 方法，它负责在玩家周围划出一个包含了区块和精灵的四方区域，然后随着玩家的走近，通过取消隐藏区块和精灵的做法来把它们变成可见的。这一做法允许关卡随着玩家的走动来慢慢暴露自己的布局。为让它发挥作用，该方法需要在 camera 组件中触发，所以，现在要把以下突出显示的代码行添加到该组件中：

```
Q.register('camera', {
    added: function() {
      this.entity.bind('moved',this,'track');
    },
    track: function() {
      var p = this.entity.p,
          stage = this.entity.parent;
      stage.centerOn(p.x, p.y);
      stage.level.unfog(p.tileX,p.tileY);
    }
  });
```

接下来需要更新创建玩家的地点，使用关卡的 startX 和 StartY 位置创建玩家。把 level1 场景的创建方法改成如下内容：

```
  Q.scene('level1',new Q.Scene(function(stage) {
    if(Q.width > 600 || Q.height > 600) {
      stage.rescale(2);
    }
```

```
    stage.level = stage.insert(new Q.Level("level1.txt",stage));
    stage.add('transition');
    var player = stage.insert(new Q.Player({ x: stage.level.startX ,
                                             y: stage.level.startY }));
    player.camera.track();
    player.bind('removed',stage,function() {
      Q.stageScene('level1');
    });
  }));
```

完成这两处修改之后，现在可以做到在地牢中四处走动，攻击随机出现的怪物和捡获战利品了。虽然这些功能已可用，但存在一个重要的问题，那就是你并不知道每个敌人还剩有多少健康值，以及自己收获了多少金币和经验值(xp)。作为游戏的收尾，下一节将改进这一问题。

13.3.8　添加血槽和 HUD

为了完成这个简单的 RPG 游戏，接下来加入一些可见的反馈，这些反馈显示敌人所剩的健康值及玩家在健康值、金币和 xp 收获方面的当前状况。

构建 DOM 游戏的好处之一是，把新的持久元素添加到游戏中很容易。以将要添加的血槽为例，一组简单的 CSS 长方形应就能应对一切。

为将血槽变成可重用的，我们打算把它建成一个组件，该组件可被添加到精灵对象中，并可通过监听精灵对象到处放置的 health 事件进行更新。

将代码清单 13-13 中的 healthbar 组件添加到 rpg.js 中的其他组件所在位置，放在 player_input 组件定义之后。

代码清单 13-13：healthbar 组件

```
Q.register('healthbar', {
  added: function() {
    this.entity.bind('health',this,'update');

    this.bg = $("<div>").appendTo(this.entity.dom).css({
      width: "100%",
      height: 5,
      position: 'absolute',
      bottom: -6,
      left: 0,
      backgroundColor: "#000",
      border: "1px solid #999"
    }).hide();

    this.bar = $("<div>").appendTo(this.entity.dom).css({
      width: "100%",
      height: 5,
      position: 'absolute',
      bottom: -5,
```

```
      left: 1,
      backgroundColor: "#F00"
    }).hide();

    Q.transitionDOM(this.bar[0],'width');

  },

  large: function() {
    this.bg.css({ height: 20, bottom: -1 }).show();
    this.bar.css({ height: 20, bottom: 0 }).show();
    return this;
  },

  update: function(sprite) {
    this.bar.show();
    this.bg.show();
    var p = sprite.p;
    var width = Math.round(p.health / p.maxHealth * 100);
    this.bar.css('width',width + "%");
  }
});
```

你可看到，该组件创建了两个 元素，再就是一个 update 方法，该方法基于所涉及精灵剩余的健康值来设置里面的那个 的宽度。两个 一开始都被隐藏了起来，这是为了保持画面的简洁，除非所保持的健康值已小于满值，否则精灵一般不会显示血槽。为了给血槽的失血过程加上动画效果，组件针对血槽调用了 Q.transitionDOM 方法，给它的宽度(width)属性加上了一个过渡。

为查看血槽的实际应用情况，把 healthbar 组件添加到 Q.Enemy 精灵中，如以下突出部分代码所示：

```
Q.Enemy = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      sheet: 'characters',
      z: 10,
      health: 10,
      maxHealth: 10,
      damage: 5,
      xp: 100
    }).extend(props));
    this.add('tiled, transition, healthbar');
    this.bind('interact',this,'interact');
    this.hide();
  },
```

现在，若在地牢中走动并攻击所遇到的敌人，那么你应会看，随着你对它们的攻击，在死亡之前，它们的血槽中的血会逐渐减少。

最后所需的一个功能是 HUD(Head Up Display，抬头显示设备)，HUD 用来显示玩家的健康值及沿途收获的金币和 xp 值。为实现该功能，游戏重用你刚创建的血槽(放大了其外观尺寸，不知你是否还记得之前代码中的 large 方法)，并添加了一个新的名为 Q.Stat 的精灵来在屏幕上显示统计数据。

此外，游戏还会启用另一个充当 HUD 元素容器的舞台对象，这样第一个舞台对象就可以在必要时四处移动来跟踪玩家。

同样，因为该游戏使用 DOM 元素，所以把文本精灵添加到游戏中是很简单的事情，仅需设置精灵的 的内容就可以了。至于玩家的血槽，我们创建一个设置了大小的假精灵来骗过引擎，然后调用 healthbar 组件的 large 方法来放大血槽的尺寸。

将代码清单 13-14 所示的这两个精灵的定义添加到 rpg.js 中，置于 Q.load 调用的前面。

代码清单 13-14：Stat 精灵

```
Q.Stat = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props).extend({
      w: 100, h: 20, z: 100
    }));

    this.el.css({color: 'white', fontFamily: 'arial' })
           .text(this.p.text + ": 0");
  },

  update: function(data) {
    this.el.text(this.p.text + ": " + data.amount);
  }
});

Q.PlayerHealth = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props).extend({
      w: Q.width / 4, h: 20, z: 100
    }));
    this.add('healthbar');
    this.healthbar.large();
  },
  update: function(sprite) {
    this.trigger('health',sprite);
  }
});
```

如你所见，这两个精灵类很简单。第一个设置了元素的一些 CSS 样式，然后使用 jQuery 的 text 方法来设置 div 的内容。第二个 Q.PlayerHealth 精灵类使了一点小手段，重用了上一节中的 healthbar 组件，不过，它调用辅助方法 large 来重设血槽的大小，把尺寸改大一些。然后，通过触发一个 health 事件，它把所接收到的任何更新事件都传递给 healthbar 组件。

现在到了压轴部分：游戏需要设置一个名为 hud 的新场景，该场景创建 HUD 元素并

绑定更新(update)方法，这样这些元素才能被更新。接着，level1 场景需要在自己被创建时启动 hud 场景，并把玩家的适当事件与 hud 舞台触发的事件绑定起来。

这一切听起来要比实际情况复杂许多，按照代码清单 13-15 中突出显示的内容来修改 rpg.js 结尾处的那部分代码，应就能获得想要的效果。

代码清单 13-15：HUD 场景

```
Q.scene('hud',new Q.Scene(function(stage) {
  var health, gold, xp;
  health = stage.insert(new Q.PlayerHealth({ x: 0, y: 10 }));
  stage.bind('health',health,'update');
  gold = stage.insert(new Q.Stat({
                              text: "gold", x: Q.width-100, y: 10
                          }));

  stage.bind('gold',gold,'update');
  xp = stage.insert(new Q.Stat({ text: "xp", x: Q.width-200, y: 10 }));
  stage.bind('xp',xp,'update');
}));

Q.scene('level1',new Q.Scene(function(stage) {
  Q.stageScene('hud',1);
  if(Q.width > 600 || Q.height > 600) {
    stage.rescale(2);
  }
  stage.level = stage.insert(new Q.Level("level1.txt",stage));
  stage.add('transition');
  var player = stage.insert(new Q.Player({ x: stage.level.startX ,
                                           y: stage.level.startY }));
  player.camera.track();
  player.bind('removed',stage,function() {
    Q.stageScene('level1');
  });

  player.bind('health',stage,function() {
    Q.stage(1).trigger('health',player);
  });

  Q.stage(1).trigger('health',player);
  player.bind('gold',stage,function(amount) {
    player.p.gold += amount;
    Q.stage(1).trigger('gold',{ amount: player.p.gold });
  });

  player.bind('xp',stage,function(amount) {
    player.p.xp += amount;
    Q.stage(1).trigger('xp',{ amount: player.p.xp });
  });

}));
```

通过调用 Q.stage(1)方法来触发调用，level1 场景触发了 hud 场景的事件。hud 场景创建了所有三个界面元素，然后把它们的 update 方法绑定到舞台对象的事件上。

把 hud 舞台和关卡主舞台彼此分开，这样一来，在多个关卡中重用 hud，以及保持各个代码块的短小，这些都变成了容易做到的事情。

若重载该游戏，现在你应该能累积金币和攻击随机出现(不过都是同等难度)的敌人精灵，见图 13-2。

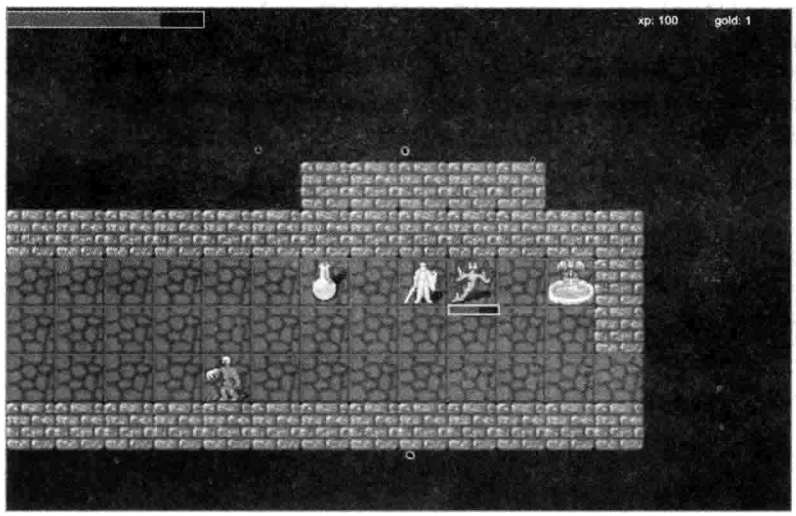


图 13-2　在 iPad 上显示的最终游戏画面

在编写了几百行代码后，现在你已构建出了一个 nethack 风格、基于 CSS3 的 RPG 游戏主体框架，如图 13-2 所示。此外，还可以添加许多功能来丰富该游戏，其中包括敌人的移动、敌人对应的不同强度和赏金、随机的关卡和路径发现方式、物品栏，以及一个出色的 nethack 风格 RPG 游戏应具备的其他所有机关等。若希望继续深入研究该游戏，那么作为遵循 MIT 协议的开源代码，这一游戏也已放到 GitHub 上供你下载。

13.4　小结

在本章中，你使用 CSS 和 DOM 元素构建了一个游戏，不过，还有更多的诸如动画和 3D 变形一类的 CSS3 功能尚未被谈及，这些功能本身就能填满另一本书。若你正在探索更多可用 CSS3 来实现的事情，可以查看一下参考文献部分提到的一些资源。虽然新的 CSS3 功能很简洁，但随着硬件加速的画布出现在越来越多的设备和浏览器中，且 CSS 的主要优势之一又是向后兼容，所以，依赖于最先进的 CSS 功能从无到有构建一个游戏可能已不是最好的做法。本章向你展示了如何构建一个既能为较新桌面和移动浏览器提供非常顺畅的动画体验，但即使是放回桌面的 IE6 中也能顺利运行的游戏。

第14章

使用 SVG 和物理引擎构建游戏

本章提要

- 了解可缩放矢量图(SVG)
- 通过 JavaScript 操纵 SVG
- 创建 SVG 精灵
- 实现一个物理引擎
- 添加敌人和增强能力的物品

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 14 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

14.1 引言

SVG(scalable vector graphics，可缩放矢量图形)是 HTML5 所拥有的一种最可能与 Flash 直接抗衡的技术。SVG 提供了绘制矢量图形的能力，可通过缩放、旋转和变形图形来迎合你的心意；与此同时，它还提供了一个可与其交互的场景图，这意味着 SVG 元素可接收鼠标和触摸事件。本章使用 SVG 和一个(借助 ActionScript)从 C++移植到 JavaScript 上的、名为 Box2D 的 2D 物理引擎来共同创建一个物理模拟场所和大炮射击游戏。

14.2　了解一些 SVG 基础知识

SVG 的际遇有些离奇，它很早就问世了(1999 年)，却从未获得重用。原因之一是因为以前的 Internet Explorer 不支持该标准，而是选用了自有的矢量标记语言(Vector Markup Language，VML)来担当同一任务。在 IE 9 发布之前，使用 SVG 就意味着完全割舍 IE 用户，或必须使用诸如 Raphael.js 一类既支持 SVG 也支持 VML 的库。

随着 IE 9 的发布，由于有着以 HTML5 认可的方式构建矢量图形的能力，SVG 的前景开始出现了转机。移动设备上的 Safari 和 Android 3.0 及更新版本，以及所有桌面浏览器的最新版本都提供了非常出色的 SVG 支持，所以，只要你觉得舍弃较旧版本的 Internet Explorer 和 Android 用户是可以接受的，那么构建一个依赖于 SVG 的游戏就是一种可行的选择。不过，它在性能方面的表现仍有待改进，所以，在确认把 SVG 用作一项游戏技术之前，先在目标平台上执行一些测试会好一些。

14.2.1　在页面上显示 SVG

SVG 是一种基于 XML 的标记语言，提供了许多的矢量原语支持，这其中包括文本、矩形、圆形、椭圆和任意路径等。这些原语可使用不同的笔画(stroke)和填充(fill)，包括模式(pattern)和渐变(gradient)填充等。SVG 还支持一些高级功能，如剪裁、掩蔽、合成和动画。

浏览器提供了许多种把 SVG 放到页面上的做法，包括作为<img>标签的源、通过<embed>标签链接、通过<object>标签链接，以及使用<iframe>嵌入，或直接把<svg>标签放到页面中。拥有太多选择也是一种困惑，不过实际上你不必了解所有这些做法。下面列出三个不同用例，每个用例都使用了一种推荐的嵌入机制：

- 第一种，若你只想用一种简单方式把一个外部 SVG 文档放到页面上，那么使用<img>标签。你不能使用脚本操纵该标签，4.0 版本之前的 Firefox 用户也没有使用这种做法的福气(目前这大约相当于 4%的 Firefox 用户，且这一数目会越来越少)，不过这是把 SVG 文档放到页面上的最简单做法，你仅需把它当成图像使用即可。

```
<img src='mydocument.svg' />
```

- 第二种，若有一个外部 SVG 文档，你希望通过脚本操纵它并与它交互，那么使用<embed>标签，该标签支持文档访问和事件处理程序的添加，格式如下：

```
<embed src="mydocument.svg" type="image/svg+xml" />
```

- 第三种，从游戏开发的角度看，最常见的用法是直接把 SVG 文档嵌入到页面中：

```
<svg id="mysvg" xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="20" y="20" width="50" height="50" fill="black" />
</svg>
```

通常情况下，应以一个空的<svg>标签开头(与<canvas>标签类似)，然后动态加入所有对象，这就是本章扩展 Quintus 引擎以支持 SVG 的做法。其中要留意的一件事情是，标签包含了 version 特性和 xmlns(XML namespace 的缩写)特性。

名称空间很重要，因为你把一个不同类型的文档嵌入到了 HTML 中，需要告诉浏览器如何处理它。浏览器是很聪明的，在这种情况下，它能够渲染没有指定名称空间的文档；不过，试图通过 JavaScript 创建没有名称空间的元素，这样的做法却不会成功。

14.2.2　了解基本的 SVG 元素

如你所见，简单地说，SVG 文档就是一些可被直接嵌入到页面中的简单 XML 文档，也可通过 URL 加载 SVG 文档，也可以把本地计算机中的 SVG 文档直接加载到浏览器中。

代码清单 14-1 给出了一个内嵌在 HTML5 文档中的简单手写 SVG 文件，需要使用 images/目录中的 penguin.png 文件来让它正常工作。该文件的输出如图 14-1 所示。

代码清单 14-1：一个简单的 SVG 文件

```
<!DOCTYPE HTML>
<html lang="en">
<head>
     <meta charset="UTF-8">
     <title>A Birdhouse</title>
</head>
<body>
<svg id="mysvg" xmlns="http://www.w3.org/2000/svg" version="1.1"
     width="400" height="400"
     viewBox="0 0 150 150" >
  <g transform="rotate(-5,75,75)">
    <rect id="rect" x="50" y="50" rx="5" ry="5"
          width="50" height="50" fill="black" />
    <circle id="circle" cx="75" cy="75" r="10" fill="#CCC"/>
    <path d="M 50 50 L 75 50 L 75 0 z"
          fill="black" stroke="#CCC" stroke-width="2"/>
    <polygon points="75,50 100,50 75,0"
          fill="#CCC" stroke="black" stroke-width="2"/>
  </g>

  <image xlink:href='images/penguin.png'
     width='32' height='32'
     transform="translate(75,75)"
     onclick="alert('Penguin Click');"Click
     ontouchstart="alert('Penguin Touch');" />
  <text x="75" y="125" text-anchor="middle"
        font-family="Verdana" font-size="10" fill="black" >
    A tilted birdhouse
  </text>
</svg>
```

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

图 14-1　一个 SVG 页面例子

如前所述，SVG 提供了许多不同的绘制原语。与画布不同，在创建某个元素之后，该元素会像 DOM 元素那样，在场景图中占据一个位置，可以通过修改元素的 x 和 y 属性或添加一个 transform 属性来移动元素。SVG 元素还可被附上事件处理程序，若在桌面上单击企鹅图像，可触发 onclick 事件的处理程序；若在 WebKit 设备上触摸它，则会触发一个“Penguin Touch”提示框的弹出。

每种原语元素还都有一组特定于所涉及元素的属性，虽然仅一章的篇幅无法对 SVG 规范所有细节进行全面概述，不过接下来的几节会讨论之前用到的几个原语元素的一些详细信息。若希望了解关于 SVG 的更多内容，请查阅 www.w3.org/TR/SVG/上的规范。

注意：截至撰写本书时为止，Android 设备尚不支持把 ontouchstart 事件处理程序当成特性添加的做法，但在借助 addEventListener 手动进行添加后，它们是可正常工作的。

<svg>

前面已对基本的<svg>容器元素作了简单介绍，不过，它还有另外几个在本章后面起到重要作用的属性(SVG 中的特性区分大小写，所以务必与文中给出的大小写保持一致)。其中的 width 和 height 特性的含义不言自明，因为它们定义的是 HTML 文档内部的元素的宽度和高度。ViewBox 特性(其中的“B”是大写)很有趣，它定义 SVG 文档在 SVG 容器内部的可视部分。viewBox 接收四个整型参数(用空格而非逗号隔开)：

```
<svg width='WIDTH' height='HEIGHT' viewBox="X Y WIDTH HEIGHT"> ... </svg>
```

如你所料，从游戏的角度看，viewBox 可被当成摄像头用在游戏中。若把 viewBox 的 WIDTH 和 HEIGHT 设置成小于<svg>元素的 WIDTH 和 HEIGHT，那么 SVG 容器中的对

象就会被放大显示；若把 WIDTH 和 HEIGHT 设置成比容器大，那么元素就会被缩小；而设置 X 和 Y 参数实质上就是在页面上平移摄像头。

若<svg>元素的纵横比和 viewBox 无法对齐，那么内容在容器内部的显示方式就取决于 preserveAspectRatio 特性的值。默认选项会确保内容被按比例缩小，这样整个 viewBox 在容器内部就都是可见的，内容在 x 和 y 方向上都是居中显示。preserveAspectRatio 有许多可配置的值，所以若需控制元素的显示方式，可通过 w3.org 网站的页面 www.w3.org/TR/SVG/coords.html#PreserveAspectRatioAttribute 来了解这些可用选项的完整说明。

大部分情况下，游戏会设置一个与元素的纵横比保持一致的视口，所以你一般不必重新配置 preserveAspectRatio。

SVG 文档必须有一个顶层的<svg>标签，但它也可以拥有子<svg>标签，这些标签可被单独变形和移动。

<rect>

<rect>标签定义一个矩形或一个正方形，它接收 width 和 height 参数来定义矩形大小，以及接收可选的 rx 和 ry 参数来给矩形加上边框圆角。x 和 y 位置指的是矩形的左上角。

<circle>和<ellipse>

<circle>标签定义一个圆形，它接收一个 cx 和一个 cy 参数，这两个参数定义圆的圆心，然后是一个 r 参数，该参数定义半径。若不想要一个对称的圆，你也可以使用<ellipse>，除了圆心参数之外，它还使用 rx 和 ry 来定义 x 和 y 半径。

<path>

<path>标签是一个非常有用的标签，该标签就相当于 SVG 的瑞士军刀，可用来绘制任意路径和形状。它的 d 特性设置用来绘制对象的路径数据，路径数据是一种命令后面跟参数形式的字符串，以下是前面给出的绘制三角形的例子：

```
<path d="M 50 50 L 75 50 L 75 0 Z" fill="black"
      stroke="#CCC" stroke-width="2"/>
```

其中的<path>标签用到了三个命令：

- M：移动到绝对位置
- L：连线到绝对位置
- Z：关闭路径

使用小写的 m 或 l 则意味着在其之后提供的点位置是相对前一个命令而言的，并非一个绝对位置(除了初始的 m，在这种情况下，该命令后面的点位置无论如何都会被解释成一个绝对位置)。这些命令绘制直线，但也可以分别使用 C 和 S 或 Q 和 T 命令绘制三次贝塞尔曲线和二次贝塞尔曲线。不过，手工绘制贝塞尔曲线可不是什么好玩的事情，所以，你可能更希望使用诸如 Adobe Illustrator 或是开源的 Inkscape 一类的程序来生成路径，这两个

程序都导出到 SVG。

同样，欲了解更多详细信息，w3.org 规范文档也提供了一个完整的资料来源：www.w3.org/TR/SVG/paths.html#DAttribute。

<polygon>和<polyline>

<polygon>和<polyline>元素很像是仅限于直线的<path>元素版本，这两个元素都接收一个 points 特性，该特性定义了一组构成形状的点。<polygon>元素是一些闭合的形状，最后一个点会自动连接回第一个点，<polyline>元素就只可设置笔画，不可填充。

```
<polygon points="75,50 100,50 75,0"
         fill="#CCC" stroke="black" stroke-width="2"/>
```

如你所见，每个点都被定义成一组逗号分隔的 x 和 y 值，点和点之间则以空格分开。

<image>

可将一些图像内嵌到 SVG 中，但这里没有什么魔法，基于位图的图像不会因此变成了基于矢量的，如果它们被过于放大就会露出原形。如之前的例子所示，<image>标签的写法如下：

```
<image xlink:href='images/penguin.png' width='32' height='32' />
```

<image>标签需要用一个 xlink:href 特性来定义与 DOM 标签<img>的 src 特性等价的内容，然后显式说明 width 和 height。若不提供宽度和高度，元素默认为 0×0。

<text>

如你所料，也可以在 SVG 内部绘制文本。可使用 font-family 和 font-size——名称与 CSS 中的相同——设置字体和尺寸，以及使用 x 和 y 或是变形来定位文本。实际上，文本被放在文本标签的内部，如下例所示：

```
<text x="75" y="125" text-anchor="middle"
       font-family="Verdana" font-size="10" fill="black" >
   A tilted birdhouse
 </text>
```

通过使用 text-anchor 特性及其可能的值：start、middle 或 end，这些值分别对应左对齐、中间对齐和右对齐的文本，可以控制文本的位置，这一位置是相对于所提供的 x 和 y 坐标而言的。

在<text>元素的内部，可使用<tspan>标签为各段文本加上不同的样式和设定不同的位置，例如：

```
<text x="75" y="125" font-family="Verdana" font-size="10" fill="black" >
   A tilted <tspan fill="red">birdhouse</tspan>
</text>
```

该例子的输出效果是，单词 *birdhouse* 的字体被设置成红色。

若希望获得动态的多行文本，SVG 并非实现这一做法的地方，需要使用<tspan>及做了适当调整的 x 和 y 位置来显式拆分文本，或使用多个<text>元素。可考虑把 DOM 元素置于 SVG 之上，然后使用 zIndex 来控制层次顺序。

<g>

<g>是本节讨论的最后一个标签，该标签用来分组元素，这样这些元素就可以被当成一个单元进行样式化和变形。以下内容来自之前的例子：

```
<g transform="rotate(-5,75,75)">
  <rect id="rect" x="50" y="50" rx="5" ry="5"
        width="50" height="50" fill="black" />
  <circle id="circle" cx="75" cy="75" r="10" fill="#CCC"/>
  <path d="M 50 50 L 75 50 L 75 0 z"
        fill="black" stroke="#CCC" stroke-width="2"/>
  <polygon points="75,50 100,50 75,0"
        fill="#CCC" stroke="black" stroke-width="2"/>
</g>
```

<g>标签内部的所有元素都围绕 75,75 这一中心旋转-5 度，组的使用允许你定义可轻松实现动画的复杂子形状。

14.2.3　变形 SVG 元素

之前讨论的所有元素都有一个名为 transform 的特性，该特性使得 SVG 元素能够实现任意的定位、旋转和缩放，这与 CSS3 提供的 transform 特性非常类似。

transform 属性接收一个变形列表，然后按照顺序逐个实施。表 14-1 给出了可用的变形。

表 14-1　变形属性

变　形	说　明
translate(tx,ty)	将元素向右下角移动 tx 和 ty 个单位，若 ty 被省略，则假设为 0
scale(sx,sy)	在水平方向和垂直方向分别缩放元素 sx 倍和 sy 倍，若 sy 被省略，则假设它的值与 sx 相同。sx 和 sy 的值可大于 1 也可小于 1，大于 1 则放大元素，小于 1 则缩小元素
rotate(angle) rotate(angle,cx,cy)	以 angle 为度数来旋转元素，若提供了 cx 和 cy 有提供，则元素以该点为中心进行旋转；若未提供，则以(0,0)为中心进行旋转。提供 cx 和 cy 的做法相当于 translate(cx,cy) rotate(angle) translate(-cx,-cy)这一调用的快捷方式
skewX(angle) skewY(angle)	较少用于游戏，该属性实现一个水平或垂直方向的倾斜
matrix(a,b,c,d,e,f)	该属性允许你使用一个 3×3 矩阵执行一个任意的 2D 变形，在二维空间中，矩阵的一行始终是 0,0,1，所有只有六个值 a-f 需要指定。上述每种变形都可用矩阵形式来表示，所以，显式设置矩阵提供了一种实施任意变形的快捷做法

14.2.4　应用笔画和填充

之前给出的所有矢量元素都可以指定一种笔画(stroke)，笔画定义元素轮廓的绘制方式；以及指定一种填充，填充定义对象内部看起来的样子(<polyline>没有内部的说法，所以填充不适用于该元素)。

有许多可用的笔画和填充属性，表 14-2 列出了其中最常用的那些属性。

表 14-2　笔画和填充属性

属　性	说　明
stroke	某种用来绘制轮廓的颜色，或指向某种用来绘制轮廓的渐变或模式的引用
stroke-width	轮廓线的宽度
stroke-linejoin	可将值设为 miter、round、bevel 和 inherit 之一，该属性定义了不同线段之间的连接方式，miter 是默认值，定义了一个锐角
stroke-opacity	介于 0～1 之间的一个值，定义轮廓的不透明度
fill	某种用来绘制填充的颜色，或指向某种用来绘制填充的渐变或模式的引用
fill-opacity	介于 0～1 之间的一个值，定义填充的不透明度

注意：此外，还存在其他一些用于创建虚线、微调线连接及控制填充算法的属性，这些属性可能不常被用到，不过可以访问页面 www.w3.org/TR/SVG/painting.html 了解一下规范中的所有这些可用选项。

可使用简单的 CSS 颜色定义 stoke 和 fill 属性，不过它们也可以使用渐变。SVG 支持两种类型的渐变：线性的和径向的。线性渐变定义代表渐变起点和终点的 x1、y1 和 x2、y2 属性，径向渐变除了定义 r 半径之外，还定义分别代表外圆圆心和渐变焦点的 cx、cy 和 fx、fy。然后，这两种类型的渐变都会使用颜色标签<stop>来表示颜色从开始到结束的具体百分比。

渐变在<defs>段内完成创建，并使用 id 特性进行标识。在这之后它们可被引用，引用方式是在 URL 值内部使用前面附加了一个井号的 id。

接下来使用一个例子来说明所有这些做法，这样会更容易理解一些。代码清单 14-2 定义了两个渐变：一个线性的和一个径向的，然后把它们分别应用在两个等大的方形上，结果如图 14-2 所示。

代码清单 14-2：SVG 渐变

```
<!DOCTYPE HTML>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <title>SVG Gradients</title>
</head>
<body>
<svg id="mysvg" xmlns="http://www.w3.org/2000/svg"
    version="1.1" width="800" height="800" >
  <defs>
    <linearGradient id="linear-test"
                    x1="1" y1="0" x2="0" y2="0">
      <stop offset="5%" stop-color="black" />
      <stop offset="55%" stop-color="white" />
      <stop offset="95%" stop-color="black" />
    </linearGradient>
    <radialGradient id="radial-test"
                    cx="0" cy="0" r="1" fx="0" fy="0">
      <stop offset="5%" stop-color="black" />
      <stop offset="55%" stop-color="white" />
      <stop offset="95%" stop-color="black" />
    </radialGradient>
  </defs>
  <rect x="0" y="50" width="375" height="375" fill="url(#linear-test)" />
  <rect x="400" y="50" width="375" height="375" fill="url(#radial-test)" />
  </svg>
</body>
</html>
```

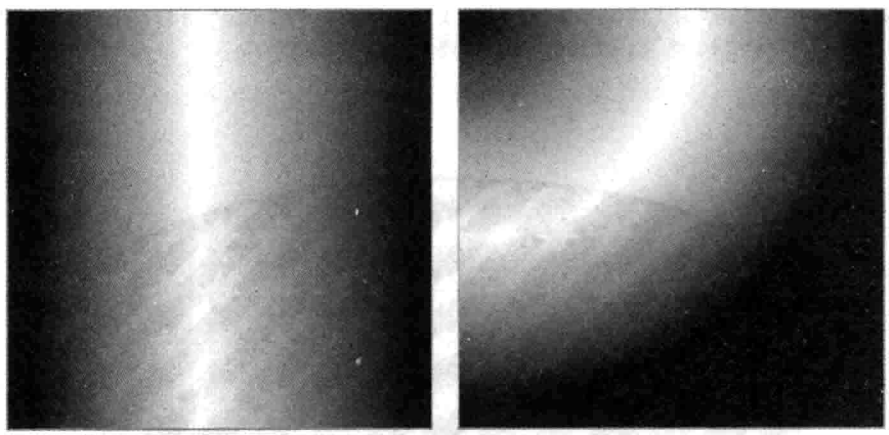


图 14-2　SVG 渐变

这两类渐变还都支持一个 gradientUnits 特性，该特性确定 x1、y1、x2、y2 和 cx、cy、r、fx、fy 特性所采用的单位，默认情况下使用 objectBoundingBox 值，这意味着所有的值应处于 0～1 范围内。若使用的是另一选项 userSpaceOnUse，则值的设置范围将与画布上的元素的相同。

SVG 还支持模式填充，这使得你能够定义一组 SVG 元素来充当可被重复使用的填充模式。模式使用一个带有 id 的<pattern>元素定义，该元素接收 width 和 height 特性，此外还接收一个 viewBox 特性，viewBox 的作用方式与<svg>元素中的相同。在<pattern>元素内

部，可以绘制一些构成该模式的 SVG 元素。之后，填充操作需要在填充特性中引用 id，这与渐变中的做法是一样的。同样，举例说明可让这些做法显得更易于理解一些，代码清单 14-3 给出了一个使用圆形和对角 polyline 创建的模式，在把这一模式应用到一个椭圆上后，得到的结果如图 14-3 所示。

代码清单 14-3：一个 SVG 模式例子

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>SVG Patterns</title>
</head>
<body>
<svg id="mysvg" xmlns="http://www.w3.org/2000/svg" version="1.1"
width="800" height="800" >
 <defs>
   <pattern id="pattern-test" patternUnits="userSpaceOnUse"
            x="0" y="0" width="50" height="50"
            viewBox="0 0 10 10" >
     <circle cx="5" cy="5" r="5" fill="black" />
     <polyline points="0,0 10,10" stroke="white" stroke-width="2"/>
   </pattern>
  </defs>
  <ellipse fill="url(#pattern-test)" stroke="black" stroke-width="5"
           cx="400" cy="200" rx="350" ry="150" />
</svg>
</body>
</html>
```

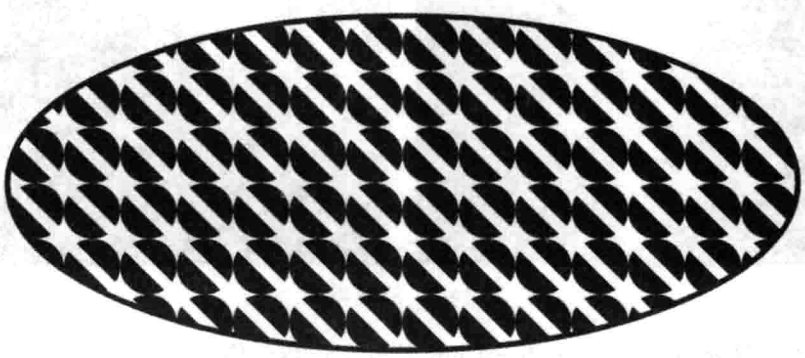


图 14-3　SVG 模式

同样，完整的规范中还记载了一些很少用到的选项，你可了解一下这些本书未涵盖的内容。

14.2.5　超越基础

SVG 是一个庞大的规范，它还包含了其他许多方面的内容，这其中包括 Animation(动

画)、Filters(过滤器)、Clipping(剪裁)、Masking(屏蔽)和 Compositing(合成)等，这些都是本章不会用到的功能。不过，在许多情况下，这些功能在游戏开发中可能是很有用处的，所以，若有兴趣了解更多这方面的内容，你应花点时间读完整个 SVG 规范：www.w3.org/TR/SVG。

若尚未看过该规范，那么你应该熟悉一下它的内容，因为溯本求源通常是快速获取答案的最佳做法(往往比 Google 搜索更快捷)。

14.3　通过 JavaScript 使用 SVG

如你所料，SVG 元素有一个通过 JavaScript 公开的接口，该接口使得你能够操纵任意 SVG 属性。但实现这一功能的方法与普通 DOM 元素的有所不同，所以 jQuery 无法以一贯方式加以处理。好在用于扩展 jQuery 的插件中有提供 SVG 支持的插件，比如 Keith Wood 的 jQuery SVG：http://keith-wood.name/svg.html。

不过，本节不打算引入另一个依赖，而探讨如何直接添加和操纵 SVG 文档。这样做的目的是确保在下一节将 SVG 支持添加到 Quintus 引擎中时，引擎不会因过多的抽象层而在性能方面陷入困境(在直接使用 DOM 方法存在性能优势时，上一章的 CSS3 RPG 游戏同样采用了这样的做法)。

14.3.1　创建 SVG 元素

在不使用 jQuery 的情况下创建新 DOM 节点的一般机制是使用 document.createElement 方法，但使用这一方法来在页面上创建一个<svg>元素却不太奏效。它会把一个名为<svg>的元素添加到页面中，但该元素表现得像一个普通 DOM 元素，并不拥有任何类 SVG 的属性。

要动态创建一个 SVG 元素，需要使用鲜为人知的 document.createElementNS 方法，该方法在指定的名称空间中创建一个元素。

就 SVG 而言，名称空间应定义成：

```
http://www.w3.org/2000/svg
```

要创建 SVG 元素，需要调用 createElementNS 并传入 SVG 的名称空间作为参数：

```
var SVGNS = "http://www.w3.org/2000/svg";
var svg = document.createElementNS(SVGNS,"svg");
```

该元素现在是一个正常的 SVG 元素了，它会把自身内部的元素渲染成适当的 SVG 元素。

若要创建其他 SVG 元素，那么继续使用这一创建模式。例如，要创建一个<rect>元素，同样需要使用 createElementNS：

```
var SVGNS = "http://www.w3.org/2000/svg";
var rect = document.createElementNS(SVGNS,"rect");
```

可使用标准的 appendChild()命令将<rect>添加到<svg>元素中：

```
svg.appendChild(rect);
```

这一命令把子元素(在这个例子中是<rect>)添加到 SVG 容器的结尾处，因为 SVG 没有 zIndex 这一概念，所以 SVG 元素在容器中的顺序实际上是很重要的，因为后来添加的元素会被绘制在前面加入的元素之上。

14.3.2　设置和读取 SVG 特性

有了一个<svg>容器并掌握了在该容器中创建元素的能力，现在你可能会认为，设置这些元素的特性与设置 DOM 元素的特性一样，都是很容易的事情：

```
//无效的做法
rect.width = 500;
```

但情况并非如此。直接设置 SVG 元素特性的这种做法不起作用。要设置属性，需要使用 setAttribute 或 setAttributeNS 方法，这两种方法在没有名称空间的情况下(或在某个具体给定的名称空间中)设置命名属性。

大部分 svg 元素特性并未处于某个名称空间中，所以，对于 width、height 这一类特性来说，使用 setAttribute 是没问题的。例如，要设置某个 rect 对象的宽度，可以这样写：

```
// This will work
rect.setAttribute('width',500);
```

不过，一些元素确实拥有位于某个名称空间中的特性，一个例子是之前提到的<image>标签的 xlink :href 特性：

```
<image xlink:href='images/penguin.png' width='32' height='32' />
```

要设置该特性，需要使用 setAttributeNS，并提供正确的名称空间：

```
image.setAttributeNS("http://www.w3.org/1999/xlink","href","image.png");
```

有一组对应的方法——getAttribute 和 getAttributeNS——可用来充当特定特性的读取器，例如，要检索某个 rect 对象的 width，可以这样写：

```
var width = rect.getAttribute('width');
```

既然已掌握了创建和操纵 SVG 元素的能力，现在是时候使用 Quintus 引擎扩展来增加对 SVG 元素的支持了。

14.4　将 SVG 支持添加到 Quintus

把 SVG 元素添加到引擎中的做法与添加 DOM 元素的做法非常相似：先添加一个定制的 Q.setupSVG 方法来设置一个<svg>元素，然后创建一个定制的 Q.SVGSprite 类，该类知道如何创建一个相应的 SVG 元素。

使用 SVG 元素创建游戏的主要复杂之处在于，若允许元素由可以随机角度旋转的任意多边形构成，那么碰撞检测将会变得很棘手。幸而，这不是一个必须由引擎解决的问题，因为在下一节中添加进来的 2D 物理引擎 Box2dWeb 将负责处理碰撞的细节。

14.4.1　创建 SVG 模块

既然已掌握了如何通过 JavaScript 与 SVG 进行交互的知识，现在是时候创建 Quintus 的 SVG 模块了。创建并打开一个新的名为 quintus_svg.js 文件，输入代码清单 14-4 中的代码。

代码清单 14-4：基本的 Quintus.SVG 模块

```
Quintus.SVG = function(Q) {
  var SVG_NS ="http://www.w3.org/2000/svg";
  Q.setupSVG = function(id,options) {
    options = options || {};
    id = id || "quintus";
    Q.svg = $(_.isString(id) ? "#" + id : id)[0];
    if(!Q.svg) {
      Q.svg = document.createElementNS(SVG_NS,'svg');
      Q.svg.setAttribute('width',320);
      Q.svg.setAttribute('height',420);
      document.body.appendChild(Q.svg);
    }

    if(options.maximize) {
      var w = $(window).width()-1;
      var h = $(window).height()-10;
      Q.svg.setAttribute('width',w);
      Q.svg.setAttribute('height',h);
    }

    Q.width = Q.svg.getAttribute('width');
    Q.height = Q.svg.getAttribute('height');
    Q.wrapper = $(Q.svg)
                .wrap("<div id='" + id + "_container'/>")
                .parent()
                .css({ width: Q.width,
                       height: Q.height,
                       margin: '0 auto' });
```

```
    setTimeout(function() { window.scrollTo(0,1); }, 0);
    $(window).bind('orientationchange',function() {
      setTimeout(function() { window.scrollTo(0,1); }, 0);
    });
    return Q;
  };
};
```

Q.setupSVG 方法遵循了许多与之前的 Q.setup 和 Q.setupDOM 方法相同的模式，主要的不同之处在于，该方法需要使用 document.createElementNS 方法和 setAttribute 来创建和修改元素，而非使用一贯可靠的 jQuery 或直接设置对象的属性。

文件在开头处设置了 SVG 名称空间以备后用，在创建 SVG 元素后，该元素仍会像普通 DOM 元素那样行事，故 Q.wrapper 元素的创建可依照普通的创建方式进行。

14.4.2　添加 SVG 精灵

接下来是 SVG 精灵类，该类与 DOMSprite 类共享许多相同的做法：它必须真正创建一个浏览器元素，并通过设置该元素的属性来移动它。这里的不同之处在于，所创建元素的类型依赖于 SVGSprite 被设置成的形状。

再次打开 quintus_svg.js，将代码清单 14-5 中的代码添加到该文件的末尾处，置于最后的结束花括号之前。

代码清单 14-5：Q.SVGSprite 类

```
Q.SVGSprite = Q.Sprite.extend({
    init: function(props) {
      this._super(_(props).defaults({
        shape: 'block',
        color: 'black',
        angle: 0,
        active: true,
        cx: 0,
        cy: 0
      }));
      this.createShape();
      this.svg.sprite = this;
      this.rp = {};
      this.setTransform();
    },

    set: function(attr) {
      _.each(attr,function(value,key) {
        this.svg.setAttribute(key,value);
      },this);
    },

    createShape: function() {
      var p = this.p;
```

```
  switch(p.shape) {
    case 'block':
      this.svg = document.createElementNS(SVG_NS,'rect');
      _.extend(p,{ cx: p.w/2, cy: p.h/2 });
      this.set({ width: p.w, height: p.h });
      break;
    case 'circle':
      this.svg = document.createElementNS(SVG_NS,'circle');
      this.set({ r: p.r, cx: 0, cy: 0 });
      break;
    case 'polygon':
      this.svg = document.createElementNS(SVG_NS,'polygon');
      var pts = _.map(p.points,
                  function(pt) {
                    return pt[0] + "," + pt[1];
                  }).join(" ");
      this.set({ points: pts });
      break;

  }
  this.set({ fill: p.color });
  if(p.outline) {
    this.set({
      stroke: p.outline,
      "stroke-width": p.outlineWidth || 1
    });
  }
},

setTransform: function() {
  var p = this.p;
  var rp = this.rp;
  if(rp.x !== p.x ||
     rp.y !== p.y ||
     rp.angle !== p.angle ) {
    var transform = "translate(" + (p.x - p.cx) + "," +
                                   (p.y - p.cy) + ") " +
                    "rotate(" + p.angle +
                            "," + p.cx +
                            "," + p.cy +
                            ")";
    this.svg.setAttribute('transform',transform);
    rp.angle = p.angle;
    rp.x = p.x;
    rp.y = p.y;
  }
},

draw: function(ctx) {
  this.trigger('draw');
```

```
  },

  step: function(dt) {
    this.trigger('step',dt);
    this.setTransform();
  },

  destroy: function() {
    if(this.destroyed) return false;
    this._super();
    this.parent.svg.removeChild(this.svg);
  }
});
```

Q.SVGSprite 的 init 方法没有任何特别之处，仅是设置默认对象的 shape(形状)和 color(颜色)的默认值；它然后调用 createShape 来创建元素；最后，它调用 setTransform 来设置 SVG 元素的变形属性。

set 方法是一个辅助方法，它接受哈希属性作为参数，并使用 setAttribute 来设置每个属性。虽然这会带来一定花销，可能不应该用在每次都会执行的步进过程中，但它确实提供了一种便利做法，即以一种类 jQuery 方式一次设置多个属性。

createShape 方法也许是其中最令人感兴趣的，它的主体部分是一个 switch 语句，该语句查看 p.shape 属性然后创建适当类型的 SVG 元素。

对于 block 形状，它创建一个<rect>元素并设置该元素的 width 和 height 属性；对于 circle 形状，它创建一个<circle>元素并设置半径，因为元素使用 transform 属性来实现移动，所以它把圆心的 x 和 y 位置设置成 0；最后是 polygon 形状，对于该形状，它需要创建一个点字符串，并将其传给 points 特性。

在完成元素的创建之后，createShape 方法查看 fill 和 outline 属性，来设置填充和笔画。

同样，setTransform 方法与 Q.DOMSprite 中的同名方法看起来很相像，它的主要工作是查找自上一帧以来是否有任何属性特性被修改，然后适当地更新 SVG 元素的变形。为了实现这一点，它首先制作一个 translate 变形，随后是一个被设置成以对象为中心进行旋转的 rotate 变形，这里的顺序很重要，因为交换这两者的顺序会导致一个以点(0,0)为中心的旋转，这会给后面的实现带来各种问题。

14.4.3　创建 SVG 舞台类

引擎所需的最后一部分 SVG 功能是创建一个 SVG 感知的舞台类，该类可充当 Q.VGSprite 的容器。

为把不同的舞台分隔开来，SVGStage 类创建了自己的子<svg>元素，这些子元素位于主 SVG 元素的内部。此外，该类还公开了一些方法来移动视口。代码清单 14-6 给出了 Q.SVGStage 的代码，这些代码应被添加至 quintus_svg.js 的末尾处，放在最后的结束花括号之前。

代码清单 14-6：Q.SVGStage

```
Q.SVGStage = Q.Stage.extend({
    init: function(scene) {
      this.svg = document.createElementNS(SVG_NS,'svg');
      this.svg.setAttribute('width',Q.width);
      this.svg.setAttribute('height',Q.height);
      Q.svg.appendChild(this.svg);

      this.viewBox = { x: 0, y: 0, w: Q.width, h: Q.height };
      this._super(scene);
    },

    insert: function(itm) {
      if(itm.svg) { this.svg.appendChild(itm.svg); }
      return this._super(itm);
    },

    destroy: function() {
      Q.svg.removeChild(this.svg);
      this._super();
    },

    viewport: function(w,h) {
      this.viewBox.w = w;
      this.viewBox.h = h;
      if(this.viewBox.cx || this.viewBox.cy) {
        this.centerOn(this.viewBox.cx,
                      this.viewBox.cy);
      } else {
        this.setViewBox();
      }
    },

    centerOn: function(x,y) {
      this.viewBox.cx = x;
      this.viewBox.cy = y;
      this.viewBox.x = x - this.viewBox.w/2;
      this.viewBox.y = y - this.viewBox.h/2;
      this.setViewBox();
    },

    setViewBox: function() {
      this.svg.setAttribute('viewBox',
                            this.viewBox.x + " " + this.viewBox.y + " " +
                            this.viewBox.w + " " + this.viewBox.h);
    },

    browserToWorld: function(x,y) {
      var m = this.svg.getScreenCTM();
```

```
    var p = this.svg.createSVGPoint();
    p.x = x; p.y = y;
    return p.matrixTransform(m.inverse());
  }
});

Q.svgOnly = function() {
  Q.Stage = Q.SVGStage;
  Q.setup = Q.setupSVG;
  Q.Sprite = Q.SVGSprite;
  return Q;
};
```

此外，代码清单 14-6 还包含了一个 Q.svgOnly 方法，该方法与上一章中的 Q.domOnly 方法类似，为了更便于访问，它使用 SVG 类来替换与它们等价的非 SVG 类。

init、insert 和 destroy 方法看上去应都与 Q.DOMStage 中的那些相类似。init 方法负责创建<svg>子元素并把它添加到主 Q.svg 对象中；insert 方法扩充了父类方法，调用 appendChild 把元素添加到舞台的<svg>标签中。最后是 destroy 方法，该方法确保在舞台对象被删除时清除舞台的<svg>标签。

更令人感兴趣的是 viewport、centerOn 和 setViewBox 这几个方法，这些方法允许你把舞台对象的<svg>元素的 viewBox 当成摄像头使用，调用 centerOn 实现平移，以及调用 viewport 设置 viewBox 的宽度和高度，以此来实现放大和缩小。viewport 方法足够智能，能够检查用户之前是否已经调用了 centerOn，若是，则在重置 viewBox 之后使用存储起来的 cx 和 cy 坐标来重新居中显示画面。

最后值得一提的是 browserToWorld 方法，为了确定用户触摸或单击了 SVG 环境中的哪些位置，需要把事件的像素位置转换成该环境内的相应位置。更麻烦的是，因为 viewBox 已被设置，所以，SVG 元素只能根据 viewBox 的纵横比而非屏幕的纵横比来进行局部放大(试想一下已转成横屏的设备上的一个呈竖屏形状的 viewBox——要算出 viewBox 的像素尺寸得花点心思)。

所有这些复杂性都意味着，根据事件在屏幕上的像素位置来推算出事件的 SVG 位置是一件极困难的事情。幸而，SVG 规范提供了一个能够让你省却大半麻烦的方法 getScreenCTM，该方法返回一个到另一方的转换矩阵，即从 SVG 单位到屏幕单位的转换。不过，只要借助一点矩阵数学，调用 m.inverse()方法，你就可以使用该矩阵的逆矩阵来实现从屏幕到 SVG 这个方向的单位转换。

14.4.4　测试 SVG 类

在编写完所有这些 SVG 引擎代码后，现在是时候用 SVG 把一些东西渲染到屏幕上了。一如往常，首先创建一个模板式的 HTML 文件来加载必要的 JavaScript 文件，创建一个名为 cannon.html 的文件，将代码清单 14-7 中的代码添加到其中。

代码清单 14-7：cannon.html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
         content="width=device-width, user-scalable=0,
                  minimum-scale=1.0, maximum-scale=1.0"/>
    <title>SVG Test</title>
    <script src='jquery.min.js'></script>
    <script src='underscore.js'></script>
    <script src='quintus.js'></script>
    <script src='quintus_input.js'></script>
    <script src='quintus_sprites.js'></script>
    <script src='quintus_scenes.js'></script>
    <script src='quintus_svg.js'></script>
    <script src='cannon.js'></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

接下来，创建一个被上述文件引用的名为 cannon.js 的文件，该文件最终存放的是本章稍后创建的 Box2D 驱动的大炮射击游戏，不过就目前而言，它仅是存放一些简单的 SVG 测试代码。

将代码清单 14-8 中的代码添加到 cannon.js 这一新文件中，这段代码的目标是测试获支持的三种不同的形状——方块、圆形和多边形——并测试 browserToWorld 方法，确保可把对象添加到用户所触摸的屏幕位置上。

代码清单 14-8：最初的 cannon.js 文件

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites,Scenes,SVG')
                      .svgOnly()
                      .setup('quintus',{ maximize: true });

  Q.scene('level',new Q.Scene(function(stage) {

     stage.insert(new Q.Sprite({
      x: 100, y: 250, w: 500, h: 50
    }));

    stage.insert(new Q.Sprite({
      w: 30, h:20, x: 0, y: 100
```

```
    }));

    stage.insert(new Q.Sprite({
      r: 30, x: 50, y: 100, shape:'circle'
    }));

    stage.insert(new Q.Sprite({
      x: 120, y: 100, shape: 'polygon', color: "red",
      points: [[ 0, 0 ], [ 100, 0 ],[ 120, 25],[ 50, 50]]
    }));

    stage.viewport(400,400);
    stage.centerOn(100,200);

    $(Q.wrapper).on('touchstart',function(e) {
      var touch = e.originalEvent.changedTouches[0];
      if(touch.target.sprite) {
        touch.target.sprite.destroy();
      } else {
        var point = stage.browserToWorld(touch.pageX,touch.pageY);
        var box = stage.insert(new Q.Sprite({
                    x: point.x, y: point.y, w: 20, h: 20
        }));
      }
      e.preventDefault();
    });
  }));
  Q.stageScene("level");
});
```

其中的大部分代码看上去应有些眼熟才是，最开始的 Quintus()调用发起了链式调用，先把一些模块包含进来，然后调用 svgOnly 来替换画布精灵，最后调用 setup 并传入 options 把 SVG 元素最大化成页面的尺寸。

scene 方法设置一个名为 level 的场景，该场景把一些精灵添加到页面上。不过，这些精灵并未像往常一样拥有资产或精灵表，它们只定义了一个 shape 属性(或使用默认的 block 形状)，此外就是宽度和高度，或是半径(就圆形而言)。对于 polygon 精灵来说则是一个点集合，该集合定义了一组构成所需形状的点。

舞台设置了一个较小的视口，它的中心点也被调成落在第一个较大的方块上。若希望查看这一做法带来的效果，则可以把这些行的代码注释掉，然后重新加载页面。

最后，剩余的那部分代码检查页面上任何地方新发生的触摸，若找到，代码首先检查目标元素是否有一个 sprite 属性，若有，则销毁该精灵；若目标元素没有 sprite 属性，则代码使用 stage.browserToWorld 方法把第一个发生变化的触摸转换成 SVG 元素内部的一个点，然后在该位置上添加一个新的方块精灵。

这段代码的最终效果是，若你触摸了页面上已存在的对象，则代码会删除该对象；若触摸的是页面上的一个空白位置，则代码添加一个新方块。

现在，在受支持的移动浏览器(iOS 或 Android3+)上打开页面，试一下这些操作，你应会看到三种不同类型的精灵：方块、圆形和多边形。此外，你还应能够通过单击现有的精灵来删除它们，以及通过单击一个空白位置来添加新的精灵。

SVG 元素的命中区域很精确：除非你直接触摸了元素的内部，否则它不会消失(这里的命中区域不是正方形的)。

14.5　使用 Box2D 添加物理支持

在空间中放上一堆静止不动的元素，这不会让人觉得有多好玩。为了让事情变得更具互动性，以及免去试图推算任意凸多边形的碰撞所带来的麻烦，Quintus 增加了对一个名为 Box2D 的著名 2D 物理引擎的 JavaScript 移植版的支持，该引擎可从 http://box2d.org/下载。

Box2D 由 Erin Catto 创建，使用 C++实现；不过，为支持 Flash 开发者使用 Box2D，几个很有创新精神的家伙手动创建了一个名为 Box2DFlash 的 ActionScript 3.0 移植版本(可从 http://box2dflash.sourceforge.net/下载)。而另一些喜欢捣鼓新玩意的人，他们利用 ActionScript 和 JavaScript 之间的相似性，创建了一个 ActionScript 到 JavaScript 的转换器，该转换器可将 ActionScript 代码转换成 JavaScript。这样一说，你该明白了吧？

目前最便于使用也是最新的 JavaScript 移植版是 Box2dWeb，可从 Google Code 站点 http://code.google.com/p/box2dweb/下载。

Box2dWeb 并非完全适合 JavaScript 环境的需要，因为在每一帧的渲染过程中它都会创建许多对象，所以有可能会给 JavaScript 的垃圾收集器带来一些问题，不过它的效果出奇地好，而且把它和已经写好的 SVG 代码整合起来也是很简单的事情。

Box2dWeb 没有提供完整的文档，但这是因为它与 Box2DFlash 共享了相同的定义，后者已提供了很出色的 API 文档：www.box2dflash.org/docs/2.1a/reference/。

本章中的 SVG 代码的具体实现采用了引入物理引擎这样的做法，不过这样做的好处显而易见，因为画布游戏也可能需要借助物理引擎来实现功能。出于这一原因，Quintus 的物理功能被创建成一组组件而非继承自 SVGSprite 和 SVGStage 的类。在不增加对旋转精灵的支持的情况下，基本的画布 Sprite 类和默认的 Stage 类不会用到物理组件，不过这是一些很容易添加到基类中的功能。

14.5.1　了解物理引擎

在深入研究物理引擎的整合细节之前，需要先了解一下物理引擎实际上做了哪些事情。

首先，之所以把物理引擎集成到系统中是因为，现实世界中的对象交互是相当复杂的，若希望准确模拟一个球飞入到 2D 空间的一堆方块中的行为，该行为会导致由这些方块堆起来的塔的崩塌、方块的翻滚及彼此间的倾靠，若要模拟这些，你得费一番周折才行。

第一项挑战是各方块和球之间的精确到像素级的碰撞计算，由于方块会以各种角度进

行旋转，所以之前各章中的简单边框碰撞检测做法用在这里就显得捉襟见肘了。

第二项挑战是对碰撞发生后的行为的准确模拟：两个方块会弹开对方吗？方块会在地面上滑动吗？或是因为摩擦而停止运动？在某个任意多边形被击中时，它应如何滚动？

物理引擎会为你解决这些问题，你的工作是定义形状和构成模拟的物体(body)的物理属性。此后，可以把这些细节都交由物理引擎处理，要求它以 1/60 秒的速度来推进模拟并告知对象的新角度和位置。物理引擎能够根据对象的速度和任何作用在对象上的力(包括重力)来更新对象，并负责正确处理所有碰撞和交互。

14.5.2　实现 world 组件

与其他许多物理引擎类似，Box2D 依赖于一个核心对象 world，该对象充当被添加到游戏中的所有物理物体的容器。因为这非常类似于 Quintus 中的 stage 对象，所以把 world 作为一个组件添加到舞台对象中也就是顺理成章的事了。

设置和模拟 Box2D 环境的方法调用相当简单，你只需创建一个新的 Box2D.Dynamics.b2World 对象并传入该环境下的重力即可。然后，就每一帧而言，需要调用该环境的 step 方法，并传入自上一次调用 step 以来已经流逝的时间及速度和位置的迭代次数作为参数。运行更多次迭代则意味着模拟每次步进的幅度更小，这样模拟效果会更好且更稳定(即对象不会飞离或跌穿其他对象)，但也会花费更多的渲染时间。

Box2D 提供了一个非常全面的 API，本书不会深入阐述这一 API，因此，引擎仅是精心挑选出了一个有限的功能子集，能做到让对象在屏幕上以一种赏心悦目的方式飞来飞去就可以了。

world 组件只需发起几个调用就能搭建及运行 Box2D 环境，为让对象和碰撞回访引擎，组件还需要添加一个监听器，任何时候只要有碰撞发生，就会触发组件的回调。

为着手实现 Quintus.Physics 模块，创建一个新的名为 quintus_physics.js 的文件，将代码清单 14-9 中的代码添加到其中。

代码清单 14-9：Quintus.Physics 的自建

```
Quintus.Physics = function(Q) {
  var B2d = Q.B2d = {
      World: Box2D.Dynamics.b2World,
      Vec: Box2D.Common.Math.b2Vec2,
      BodyDef: Box2D.Dynamics.b2BodyDef,
      Body: Box2D.Dynamics.b2Body,
      FixtureDef: Box2D.Dynamics.b2FixtureDef,
      Fixture: Box2D.Dynamics.b2Fixture,
      PolygonShape: Box2D.Collision.Shapes.b2PolygonShape,
      CircleShape: Box2D.Collision.Shapes.b2CircleShape,
      Listener:  Box2D.Dynamics.b2ContactListener
    };

  var defOpts = Q.PhysicsDefaults = {
    gravityX: 0,
```

```
  gravityY: 9.8,
  scale: 30,
  velocityIterations: 8,
  positionIterations: 3
};

Q.register('world',{
  added: function() {
    this.opts = _(defOpts).clone();
    this._gravity = new B2d.Vec(this.opts.gravityX,
                                this.opts.gravityY);
    this._world = new B2d.World(this._gravity, true);
    _.bindAll(this,"beginContact","endContact","postSolve");

    this._listener = new B2d.Listener();
    this._listener.BeginContact = this.beginContact;
    this._listener.EndContact = this.endContact;
    this._listener.PostSolve = this.postSolve;
    this._world.SetContactListener(this._listener);

    this.col = {};
    this.scale = this.opts.scale;
    this.entity.bind('step',this,'boxStep');
  },

  setCollisionData: function(contact,impulse) {
    var spriteA = contact.GetFixtureA().GetBody().GetUserData(),
        spriteB = contact.GetFixtureB().GetBody().GetUserData();

    this.col["a"] = spriteA;
    this.col["b"] = spriteB;
    this.col["impulse"] = impulse;
    this.col["sprite"] = null;
  },

  beginContact: function(contact) {
    this.setCollisionData(contact,null);
    this.col.a.trigger("contact",this.col.b);
    this.col.b.trigger("contact",this.col.a);
    this.entity.trigger("contact",this.col);
  },

  endContact: function(contact) {
    this.setCollisionData(contact,null);
    this.col.a.trigger("endContact",this.col.b);
    this.col.b.trigger("endContact",this.col.a);
    this.entity.trigger("endContact",this.col);
  },

  postSolve: function(contact, impulse) {
```

```
        this.setCollisionData(contact,impulse);
        this.col["sprite"] = this.col.b;
        this.col.a.trigger("impulse",this.col);
        this.col["sprite"] = this.col.a;
        this.col.b.trigger("impulse",this.col);
        this.entity.trigger("impulse",this.col);
      },

      createBody: function(def) {
        return this._world.CreateBody(def);
      },

      destroyBody: function(body) {
        return this._world.DestroyBody(body);
      },

      boxStep: function(dt) {
        if(dt > 1/20) { dt = 1/20; }
        this._world.Step(dt,
                         this.opts.velocityIterations,
                         this.opts.positionIterations);
      }
    });
  };
```

作为例子，Box2D.Dynamics.b2World 是 Box2D 所定义的那些进行了很好名称空间规划的对象之一，这一名称空间的唯一问题是给输入增加了一点负担。出于这一原因，一种常见做法是在作用域内创建一组较短的类名，Quintus 同样遵循这一模式，把一些元素添加到了一个 B2d 对象中。

接下来，该模块定义了一些可用来创建和运行该环境的默认值，这些值包括重力的 x 分量和 y 分量、缩放倍数，以及要运行的速度和位置迭代的次数计数器等。

缩放选项是一个有趣的选项，虽然你可能会认为对象的大小应无关紧要，不过在使用一种较小的标度来衡量对象时，比如说是在 1～10(而非 100～1000)这个范围内，Box2D 的效果更好些。这意味着一般的像素标度不太适合作为 Box2D 的首选对象标度，这就是为什么缩放选项要设置默认值的原因。它是一个除数，用来把对象尺寸从像素规模缩减到一个更小的范围内。若把尺寸单位看成一米，所涉及对象的尺寸范围为一到几米，那么就 Box2D 的计算而言，你能够击中甜区(sweet spot)。

接下来是 added 方法，它的主要工作就是创建环境。实际上，这比听上去要容易许多，所需要做的就是通过选项哈希创建一个重力矢量，然后创建一个新的 B2d.World 对象。有了该对象，可以开始把一些实体添加到 Box2D 中，并且可以模拟真实世界的情况了。

唯一的问题在于，你没有任何关于某个对象何时与另一个对象发生碰撞的信息，对象仍不会发生重叠，因为是 Box2D 处理这部分内容，但你会因此缺乏任何关于交互的有意义的信息，这些信息是构建游戏所必需的(难道你希望子弹仅是从敌人身上弹开吗？)。为了

解决这一问题，Box2D 提供了设置接触监听器(contact listener)的能力。world 组件绑定了这一功能，然后通过触发 contact、endContact 和 impulse 事件把它所接收到任何碰撞转发给精灵，这些事件分别对应于 Box2D 的 BeginContact、EndContact 和 PostSolve 回调，前面两个——BeginContact 和 EndContact——分别在两个实体开始触碰和停止触碰时被调用。

最后一个 PostSolve 在每次有其他物体带来冲量(impulse)时都会被调用，这可能会是频繁发生的情况(试想一下球从方块堆积而成的小山上滚下来)，所以，你必须小心处理，保证 impulse 事件处理程序的快速执行。此外，为了保持内存的低使用率，所有的碰撞处理程序还会在所有事件之间重用相同的 col 对象，并会使用辅助方法 setCollisionData 来填充回调的数据。

每个回调都会触发三个事件，每个对象一个，然后是舞台对象一个。这种做法允许你在必要时在舞台对象内部集中进行碰撞检测，以此替代每个对象的碰撞检测。

设置完接触监听器后，added 方法把自身绑定到舞台对象的 step 方法上，目的是触发 boxStep 回调。boxStep 负责使用被传入到回调中的 dt 以及速度和位置的迭代次数，以正确的速率逐步更新环境。

唯一的其他两个方法——createBody 和 destroyBody——充当 Box2D 环境的同名方法的代理。

14.5.3　实现 physics 组件

在完成了 world 组件的构建之后，下一个就是把物理支持添加到精灵中的组件了。毫无意外地，该组件被命名为 physics，它的工作是在被添加时创建一个与精灵的大小和形状相同的 Box2D 物体、基于 Box2D 的模拟更新精灵的位置，以及在精灵被删除时删除该物体。

Box2D 支持两种类型的对象：静态的和动态的，静态对象除了等待动态对象撞向它们之外不做任何事情，它们给处理器带来的压力要小许多，因为它们不必在每次步进时都进行更新。组件支持基于 type 属性来创建这两类对象，默认情况下创建的是动态对象。

该组件的复杂性主要体现在 insert 方法中，该方法在精灵被添加到舞台对象中之后被调用，它负责创建被添加到 Box2D 环境中的物体对象。物体拥有许多属性，其中包括了位置和物体为静态或动态的说明。在创建物体之前，这些属性需要设置一些初始值。

不过，这些属性并不会告诉 Box2D 任何关于形状的信息，这项工作由一个或多个被添加到物体中的夹具(fixture)负责。每个夹具必须是一个凸面体(这意味着它没有凹陷部分)，虽然 Box2D 支持多个夹具，但 physics 组件仅支持一个。夹具也具有诸如对象的密度(density)、摩擦力(friction)和回复力(restitution)(弹力)一类的详细信息。

为将 physics 组件放入引擎中，将代码清单 14-10 中的代码添加到 quintus_physics.js 的末尾处，置于最后的结束花括号之前。

代码清单 14-10：physics 组件

```
var entityDefaults = Q.PhysicsEntityDefaults = {
```

```
  density: 1,
  friction: 1,
  restitution: .1
};

Q.register('physics',{
  added: function() {
    if(this.entity.parent) {
      this.inserted();
    } else {
      this.entity.bind('inserted',this,'inserted');
    }
    this.entity.bind('step',this,'step');
    this.entity.bind('removed',this,'removed');
  },

  position: function(x,y) {
    var stage = this.entity.parent;
    this._body.SetAwake(true);
    this._body.SetPosition(new B2d.Vec(x / stage.world.scale,
                                       y / stage.world.scale));
  },

  angle: function(angle) {
    this._body.SetAngle(angle / 180 * Math.PI);
  },

  velocity: function(x,y) {
    var stage = this.entity.parent;
    this._body.SetAwake(true);
    this._body.SetLinearVelocity(new B2d.Vec(x / stage.world.scale,
                                             y / stage.world.scale));
  },

  inserted: function() {
    var entity = this.entity,
        stage = entity.parent,
        scale = stage.world.scale,
        p = entity.p,
        ops = entityDefaults,
        def = this._def = new B2d.BodyDef,
        fixtureDef = this._fixture = new B2d.FixtureDef;

    def.position.x = p.x / scale;
    def.position.y = p.y / scale;
    def.type = p.type == 'static' ?
               B2d.Body.b2_staticBody :
               B2d.Body.b2_dynamicBody;
    def.active = true;
```

```
    this._body = stage.world.createBody(def);
    this._body.SetUserData(entity);
    fixtureDef.density = p.density || ops.density;
    fixtureDef.friction = p.friction || ops.friction;
    fixtureDef.restitution = p.restitution || ops.restitution;

    switch(p.shape) {
      case "block":
        fixtureDef.shape = new B2d.PolygonShape;
        fixtureDef.shape.SetAsBox(p.w/2/scale, p.h/2/scale);
        break;
      case "circle":
        fixtureDef.shape = new B2d.CircleShape(p.r/scale);
        break;
      case "polygon":
        fixtureDef.shape = new B2d.PolygonShape;
        var pointsObj = _.map(p.points,function(pt) {
          return { x: pt[0] / scale, y: pt[1] / scale };
        });
        fixtureDef.shape.SetAsArray(pointsObj, p.points.length);
        break;
    }

    this._body.CreateFixture(fixtureDef);
    this._body._bbid = p.id;
  },
  removed: function() {
    var entity = this.entity,
        stage = entity.parent;
    stage.world.destroyBody(this._body);
  },
  step: function() {
    var p = this.entity.p,
        stage = this.entity.parent,
        pos = this._body.GetPosition(),
        angle = this._body.GetAngle();
    p.x = pos.x * stage.world.scale;
    p.y = pos.y * stage.world.scale;
    p.angle = angle / Math.PI * 180;
  }
});
```

将该组件添加到精灵中后，它首先检查精灵是否已被添加到某个舞台对象中，若是，则立刻调用 inserted 方法，否则它会等到 inserted 事件被触发才调用该方法。

如你所见，inserted 方法占据了 physics 组件代码的很大一块，该方法创建 Box2D 物体，基于对象的形状设置夹具的定义，以及创建物体的夹具。

其中的两个辅助方法——position 和 velocity——允许你设置精灵的物体的这些属性。这两个方法还会调用物体的 SetAwake(true)。为节省 CPU 周期，经过一段时间后，Box2D

就会把那些没有移动和导致碰撞的对象置于一种“睡眠”状态。为确保这些对象在被人为移动时能被“唤醒”并再次开始响应作用力，需要手动调用 SetAwake(一般来说，任何时候只要有对象涉及碰撞，Box2D 都会自动为你处理调用事宜)。

removed 方法仅确保一件事，那就是除了要从屏幕上删除精灵之外，Box2D 环境中的物体也要被销毁。

最后是 step 方法，该方法在每次步进之后被调用，鉴于之前讨论过的缩放属性，该方法负责把 Box2D 中的物体的位置和角度转换回精灵的位置。

14.5.4　将物理支持添加到例子中

编写完 physics 组件后，现在是时候见识一下 Box2D 的作用了。要做到这一点相当简单，仅需把 world 组件添加到舞台对象中，以及把 physics 组件添加到每个精灵对象中即可。首先打开文件 cannon.html，把 Box2dWeb 和 quintus_physics.js 文件添加到该 HTML 文件中(需要通过本章代码获取 Box2dWeb-2.1.a.3.js 的一个副本，或下载一个自己的版本)，如下所示：

```
<script src='quintus_sprites.js'></script>
<script src='quintus_scenes.js'></script>
<script src='quintus_svg.js'></script>
<script src='Box2dWeb-2.1.a.3.js'></script>
<script src='quintus_physics.js'></script>
<script src='cannon.js'></script>
```

更新 cannon.js 中突出显示部分的代码，如代码清单 14-11 所示，把物理支持添加到该 SVG 例子中。

代码清单 14-11：把物理支持添加到 SVG 例子中

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites,Scenes,SVG,Physics')
                      .svgOnly()
                      .setup('quintus',{ maximize: true });

  Q.scene('level',new Q.Scene(function(stage) {


    stage.insert(new Q.Sprite({
      x: 100, y: 250, w: 500, h: 50, type:"static"
    }));
    stage.insert(new Q.Sprite({
      w: 30, h:20, x: 0, y: 100
    }));
    stage.insert(new Q.Sprite({
      r: 30, x: 50, y: 100, shape:'circle'
    }));
```

```
stage.insert(new Q.Sprite({
  x: 120, y: 100, shape: 'polygon', color: "red",
  points: [[ 0, 0 ], [ 100, 0 ],[ 120, 25],[ 50, 50]]
}));

stage.add("world");
stage.each(function() { this.add("physics"); });

stage.viewport(400,400);
stage.centerOn(100,200);
$(Q.wrapper).on('touchstart',function(e) {
  var touch = e.originalEvent.changedTouches[0];
  if(touch.target.sprite) {
    touch.target.sprite.destroy();
  } else {
    var point = stage.browserToWorld(touch.pageX,touch.pageY);
    var box = stage.insert(new Q.Sprite({
      x: point.x, y: point.y, w: 20, h: 20
    }));
    box.add("physics");
    box.bind("contact",function(sprite) {
      sprite.set({fill:"blue"});
    });
  }
  e.preventDefault();
});
```

此外，还需要把第一个方块的类型更新成 static，这样才能让它充当其他对象的平台。最后，为在触摸屏幕空白处时会被添加的方块元素增加一个 contact 事件监听器，目的是把任何它们触碰到的对象变成蓝色，结果如图 14-4 所示。

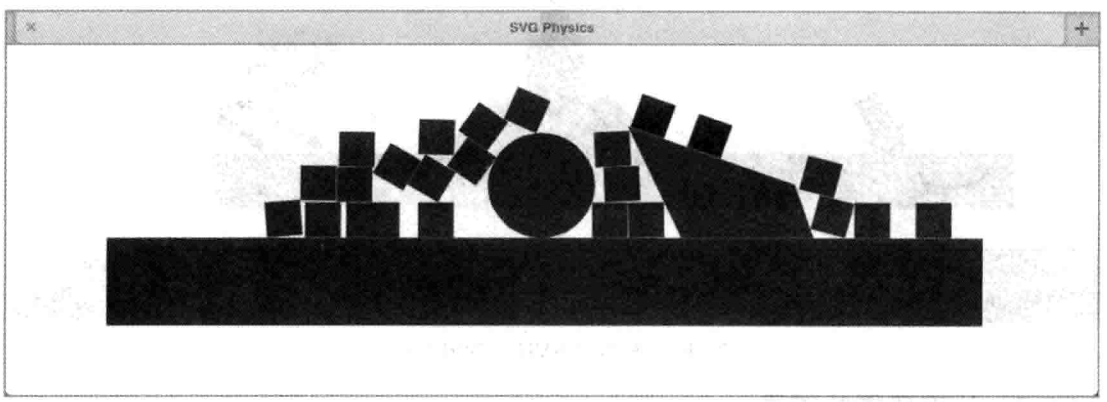


图 14-4　把 Box2D 的物理支持应用到 SVG 上

14.6　创建一个大炮射击游戏

在把物理和 SVG 支持添加到 Quintus 中后，你已经具备了用来构建一个基于物理的简单撞击类游戏所需的一切部件(Box2D 可以驱动 Angry Birds(愤怒的小鸟)这款游戏的 2D 物理引擎哦)。

基于物理的游戏的有趣之处在于，作为开发者，不必做太多工作，就能让游戏具备基本功能并能运作起来，因为物理引擎为你处理了大量的事情。但另一方面，若要挖掘游戏的趣味性以及保证游戏的良好效果，基于物理的游戏需要进行大量参数方面的调整才行。

14.6.1　设计游戏

借助本章构建的 physics 组件，大炮射击游戏背后的想法很简单：向页面四周扔出一些圆形的物体，设法击中一些小的圆形目标。

该游戏使用触摸位置来控制大炮的角度，任何时候只要松开触点就发射一颗炮弹(在桌面上则使用 mousemove 和 mouseup 事件)。

大炮是一个未启用 physics 组件的多边形精灵，所以它可根据需要进行移动和调整。它所发射的 Q.CannonBall 精灵则启用了 physics 组件，所以这些精灵能够与其他任何东西发生碰撞。

最后是 Q.Target 对象，该对象其实就是一个启用了 physics 组件的粉色小球。它监听自己的 contact 事件并检查自己所触碰到的对象是否为 Q.CannonBall 对象，若是，它就可以销毁自身并更新页面剩余目标的数目；若该数目减至 0 值，它就重启这一关游戏，在一个真正的游戏中，这就应是进入下一关游戏的时候了。见图 14-5，这是游戏过程中的一个屏幕快照。

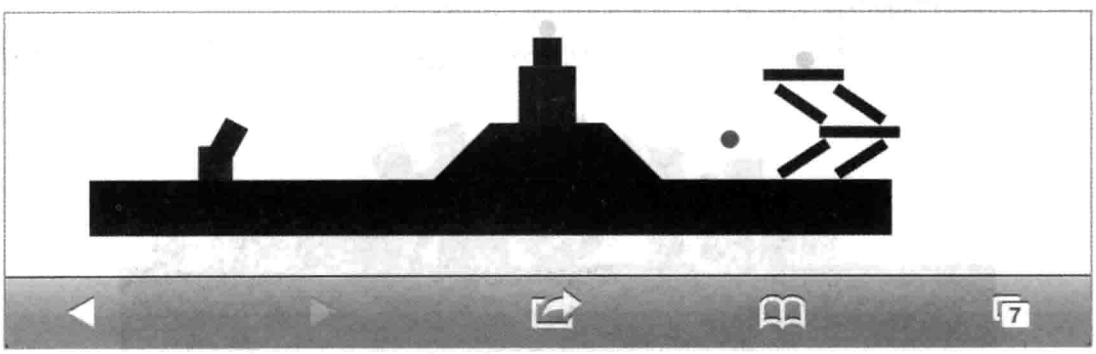


图 14-5　最终的 SVG 大炮射击游戏

14.6.2　构建所需的精灵

现在是时候编写最终版本的游戏代码了，打开 cannon.js，删掉现有的那些代码。第一部分需要加入的代码是上一节中介绍的三个精灵类：Q.CannonBall、Q.Cannon 和 Q.Target，将代码清单 14-12 中代码添加到 cannon.js 文件的顶部，放在开始之处的 Quintus 设置代码

的后面。

代码清单 14-12：大炮射击游戏的精灵

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites,Scenes,SVG,Physics')
                      .svgOnly()
                      .setup('quintus',{ maximize: true });


  Q.CannonBall = Q.Sprite.extend({
    init: function(props) {
      this._super({
        shape: 'circle',
        color: 'red',
        r: 8,
        restitution: 0.5,
        density: 4,
        x: props.dx * 50 + 10,
        y: props.dy * 50 + 210,
        seconds: 5
      });
      this.add('physics');
      this.bind('step',this,'countdown');
    },

    countdown: function(dt) {
      this.p.seconds -= dt;
      if(this.p.seconds < 0) {
        this.destroy();
      } else if(this.p.seconds < 1) {
        this.set({ "fill-opacity": this.p.seconds });
      }
    }
  });

  Q.Cannon = Q.Sprite.extend({
    init: function(props) {
      this._super({
        shape:'polygon',
        color: 'black',
        points: [[ 0,0 ], [0,-5], [5,-10], [8, -11], [40, -11],
                 [ 40, 11], [8, 11], [5, 10], [0, 5] ],
        x: 10,
        y: 210
      });
    },

    fire: function() {
```

```
      var dx = Math.cos(this.p.angle / 180 * Math.PI),
          dy = Math.sin(this.p.angle / 180 * Math.PI),
          ball = new Q.CannonBall({ dx: dx, dy: dy, angle: this.p.angle });
      Q.stage().insert(ball);
      ball.physics.velocity(dx*400,dy*400);
    }
  });

  var targetCount = 0;
  Q.Target = Q.Sprite.extend({
    init: function(props) {
      this._super(_.extend(props,{
        shape: 'circle',
        color: 'pink',
        r: 8
      }));
      targetCount++;
      this.add('physics');
      this.bind('contact',this,'checkHit');
    },

    checkHit: function(sprite) {
      if(sprite instanceof Q.CannonBall) {
        targetCount--;
        this.parent.remove(this);
        if(targetCount == 0) { Q.stageScene('level'); }
      }
    }
  });
});
```

正如所料，三个精灵类的代码都很简短，其中大部分都是属性设置代码。

此外，代表从大炮中发射出来的炮弹的 Q.CannonBall 类只有一个方法 countdown，该方法确保炮弹在发射五秒钟之后被从页面上删除，在炮弹的生存期只剩下不到 1 秒钟时，它还要设置炮弹的淡出效果。该类使用大炮的基座位置和传入的 dx 和 dy 值来计算炮弹的初始位置，dx 和 dy 值则是通过大炮发射炮弹时的角度计算得出的。

Q.Cannon 类被定义成一个类似大炮的多边形，如前所述，它不需要加入 physics 组件，因为它并不参与任何碰撞，但通过设置它的角度，你可对它加以控制。因此，该类中的 points 是以这样一种方式进行设置的，即在旋转该多边形时，围绕着大炮的基座进行旋转，因为基座被设置成点(0,0)。唯一被加入到该类中的方法是 fire 方法，该方法使用角度及 sin 和 cos 方法计算大炮顶端的位置。

不知你是否还记得高中几何，cos 返回 0～1 之间的一个值，该值代表斜边长度为 1 的直角三角形的水平分量，sin 实现同样的事情，代表的是垂直分量。这意味着大炮的顶点，所以炮弹发射点的计算方法是，用大炮的长度加上大概为炮弹半径大小的空间得到的和分别乘以 dx 和 dy，而这就是 Q. CannonBall 类所做的事情。

然后，fire 方法把新创建的炮弹对象插入到舞台对象中，然后使用计算得出的 dx 和 dy 值给它指定一个速度，这样炮弹就可以朝基于大炮角度的正确方向飞出去。

最后是仅创建了一个小粉球的 Q.Target 对象，该对象监听一个 contact 事件并检查击中自身的是否为炮弹，若是，则从父对象中删除自身并检查是否还有剩余的目标，若没有剩余目标则重启这一关游戏。因为 contact 事件会在环境的步进循环过程中发生，所以精灵需要小心行事，不能立刻销毁自身(若它调用 this.destroy()就会发生这样的事情)。相反，它要调用父对象的 remove 方法，该方法记录下该次步进结束时要删除的精灵。

14.6.3　收集用户输入并完成游戏编写

游戏的最后一部分代码简单通过抓取用户的输入来控制大炮的角度，并以一种有趣的阵型在页面上设置一些方块和目标。

这部分代码使用一个监听器来移动大炮的角度，角度的变化依赖于移动设备上的 touchstart 和 touchmove 事件及桌面上的 mousemove 事件。然后，在用户抬起手指或松开鼠标时，游戏以事先计算好的角度发射炮弹。

接下来，场景设置了一些由用户来撞倒的不同方块和几个供用户瞄准的目标，将代码清单 14-13 中的代码添加到 cannon.js 的末尾处，置于最后的花括号之前。

代码清单 14-13：剩余部分的大炮射击游戏代码

```
$(Q.wrapper).on('touchstart touchmove mousemove',function(e) {
  var stage = Q.stage(0),
      cannon = stage.cannon,
      touch = e.originalEvent.changedTouches ?
              e.originalEvent.changedTouches[0] : e,
      point = stage.browserToWorld(touch.pageX,touch.pageY);

  var angle = Math.atan2(point.y - cannon.p.y,
                         point.x - cannon.p.x);
  cannon.p.angle = angle * 180 / Math.PI;
  e.preventDefault();
});

$(Q.wrapper).on('touchend mouseup',function(e) {
  Q.stage(0).cannon.fire();
  e.preventDefault();
});

Q.scene('level',new Q.Scene(function(stage) {
  targetCount = 0;
  stage.add("world");
  stage.insert(new Q.Sprite({
    x: 250, y: 250, w: 700, h: 50, type:"static"
  }))

  stage.insert(new Q.Sprite({ w: 10, h:50, x: 500, y: 200 }));
```

```
    stage.insert(new Q.Sprite({ w: 10, h:50, x: 550, y: 200 }));
    stage.insert(new Q.Sprite({ w: 70, h:10, x: 525, y: 170 }));
    stage.insert(new Q.Sprite({ w: 10, h:50, x: 500, y: 130 }));
    stage.insert(new Q.Sprite({ w: 10, h:50, x: 550, y: 130 }));
    stage.insert(new Q.Sprite({ w: 70, h:10, x: 525, y: 110 }));

    stage.insert(new Q.Sprite({
      points: [[ 0,0 ], [ 50, -50 ],[150, -50],[200,0]],
      x: 200,
      y: 225,
      type:'static',
      shape: 'polygon'
    }));

    stage.insert(new Q.Sprite({ w: 50, h:50, x: 300, y: 150 }));
    stage.insert(new Q.Sprite({ w: 25, h:25, x: 300, y: 115 }));

    stage.each(function() { this.add("physics"); });

    stage.insert(new Q.Target({ x: 525, y: 90 }));
    stage.insert(new Q.Target({ x: 300, y: 90 }));
    stage.insert(new Q.Sprite({ w: 30, h:30, x: 10, y: 210,
                                color: 'blue' }));

    stage.cannon = stage.insert(new Q.Cannon());
    stage.viewport(600,400);
    stage.centerOn(300,100);

  }));
  Q.stageScene("level");
```

如你所见，与之前的例子相比，这里没有太多新东西。touchstart 处理程序用到了一点数学，使用 atan2(这在计算游戏手柄的角度时已讨论过)来计算大炮的角度，不过除此之外，这些事件处理程序并无任何其他难解之处。它们被放在关卡定义的外部，这是因为关卡有可能会被重置(又或你希望定义多个关卡)，且作为取消绑定和重新绑定处理程序这一做法的代替，只要处理程序不需要用到场景定义方法中的任何局部变量，那么同一个处理程序就可被一次次地重复使用。为解决这一问题，cannon 对象被当成舞台对象的属性存储起来。

只要你愿意，还可以将许多功能添加到该游戏中，其中包括多个关卡、得分，以及对用户可发射炮弹数量的限制等。此外，还可以使用 impulse 处理程序来跟踪与目标接触所产生的作用力，并只有在目标受到足够作用力的影响时(因此妨碍到缓慢滚动的炮弹制造破坏)才删除它们，

14.7　小结

你学习了许多基础知识，了解了如何使用 SVG 创建一个用到任意形状的游戏，以及

如何把这些形状和一个 2D 物理引擎关联起来。此外，Box2D 还有许多细节，其中包括连接器(joint)和冲量等，这些都是本章尚未讨论的，不过，它的基本功能应已足够用来构建简单的游戏(搞不好就是下一个 Angry Birds 呢！)。从游戏开发的角度来看，SVG 依然是一个没有获得太多关注的规范，一旦性能达到一个跨设备可接受的水平，鉴于 SVG 所提供的使用浏览器自带的场景图绘制任意矢量元素这一灵活性，你可能就会见到更多使用 SVG 来制作的游戏。此外，对诸如过滤器和动画之类高级功能的支持也正变得越来越好，所以，期望在不久的将来能看到许多更酷的 SVG 演示例子出现。

第Ⅴ部分

HTML5画布

- 第 15 章：了解 HTML5 的杰出画布
- 第 16 章：实现动画
- 第 17 章：运用像素
- 第 18 章：创建一个 2D 平台动作游戏
- 第 19 章：构建一个画布编辑器

第15章

了解 HTML5 的杰出画布

本章提要

- 创建画布元素并设定画布元素的尺寸
- 通过画布创建图像
- 绘制图像、文本和路径
- 创建渐变和模式
- 使用变形
- 了解其他一些画布效果

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 15 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

15.1 引言

本章深入探讨画布(Canvas)API，虽然在上两章中，你已了解了一些可使用画布之外的其他技术构建 HTML5 游戏的做法，但从游戏的角度看，画布仍是用来构建游戏的最灵活技术。之前的一些章节展示了如何使用画布和点阵式精灵表来制作游戏，但实际上，画布提供了多种绘制图像的做法，它拥有一整套矢量绘制方法，除了绘制形状和曲线之外，还支持文本、变形和各种各样的合成模式。

15.2 画布标签入门

你已经看到，把<canvas>元素放到页面上是很简单的事情，只需使用 width 和 height 特性将该元素添加到 HTML 文档中即可：

```
<canvas id="mycanvas" width='640' height='480'></canvas>
```

该语句在页面上创建一个 640 像素宽、480 像素高的元素，默认情况下，<canvas>元素的 CSS 高度和高度被设置成与这一像素宽度和高度相同的值。

15.2.1 了解 CSS 和像素尺寸

不过，CSS 尺寸和像素尺寸未必是相同的，可以使用完全独立于像素宽度和高度的 CSS 宽度和高度(该宽度和高度决定元素在页面上的大小)。为清楚地说明这一点，代码清单 15-1 给出了一些代码，这些代码在四个有着不同像素尺寸的画布元素的每个像素位置上都放上一个随机着色、1 像素大小的矩形。

代码清单 15-1：验证 CSS 尺寸和像素尺寸

```
<script src='jquery.min.js'></script>
<style>
 canvas { width: 200px; height: 200px; }
</style>

<canvas width="2" height="2"></canvas>
<canvas width="10" height="10"></canvas>
<canvas width="50" height="50"></canvas>
<canvas width="10" height="100"></canvas>

<script>
  $("canvas").each(function() {
    var ctx = this.getContext("2d");
    for(var y=0,h=this.height;y<h;y++) {
      for(var x=0,w=this.width;x<w;x++) {
        var r = Math.floor(Math.random()*255),
            g = Math.floor(Math.random()*255),
            b = Math.floor(Math.random()*255);
        ctx.fillStyle = "rgb(" + r + "," + g + "," + b + ")";
        ctx.fillRect(x,y,1,1);
      }
    }
  });
</script>
```

图 15-1 展示了这一代码的运行结果，可以注意到，虽然页面上的所有画布元素都被设置成了相同的大小，但它们各自有不同的像素密度。

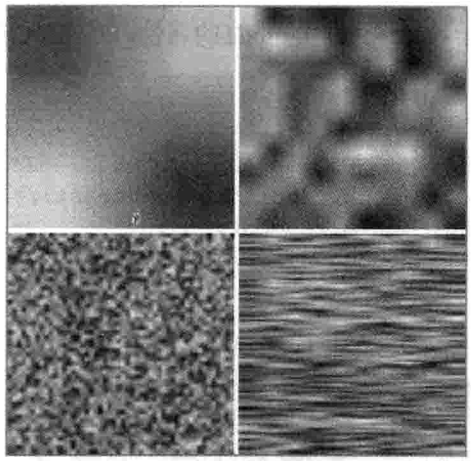


图 15-1　画布的 CSS 尺寸和像素尺寸

此外，你可能也已注意到了发生在前两个元素中的情况，当放大画布元素后，浏览器努力平滑生成图像在像素间的过渡。默认情况下，所有现代浏览器都会把双三次(bicubic)(IE 浏览器)或双线性(bilinear)(其他所有浏览器)上采样算法(upsampling algorithm)应用到图像上，这样的话，在以比自身的像素尺寸更大的尺寸进行显示时，图像看起来就会更平滑些。这一采样规则同样被应用在画布标签上，所以，若你设置的元素宽度和高度大于像素尺寸，就会发生一些上采样。

注意：双三次和双线性上采样是两种以一定大小的图像为处理源的算法，目的是在图像被放大时美化图像的观感。在没有任何上采样的情况下，放大后的图像会显露出由较大像素构成的“锯齿”边缘。这两种算法都用到了一种颜色插值方法，所以放大后的图像看上去很平滑，不会产生锯齿现象，但这是以所付出的处理时间和最终图像所丢失的细节为代价的。在保留细节方面，双三次算法通常比双线性算法做得更好，不过有时可能会导致奇异的“光晕”效应。

通常情况下，这样做是没问题的，但若创建的是一个 8 位的复古游戏，那么你可能更愿意要一个比较清晰的像素化外观。为支持这一点，一些浏览器提供一个名为 image-rendering 的 CSS 属性，该属性在重采样算法方面贡献了一点点控制权。截至撰写本书之时为止，该属性只有在 Firefox 和 Internet Explorer 上才是可用的，不过 WebKit 已经把一个补丁合并到了自身的每日构建中，所以，此刻在你阅读本书时，Safari、iOS、Chrome 和 Android Chrome 应都已提供了对这一属性的支持。这一属性的目的是强制浏览器使用更快的最邻近算法(nearestneighbor algorithm)，该算法不在像素之间做任何插值。只有 Microsoft 允许显式设置所用的算法，其他浏览器则赋予了该样式不同的名称。

为了能让这一属性起到作用，你得在厂商前缀这个地方耽搁点时间，Microsoft 仍然我

行我素，使用了一个名为-ms-interpolation-mode 的属性，以下的 CSS 样式应该不会在 WebKit 添加支持之后变成过时的：

```
canvas {
  image-rendering: -moz-crisp-edges;          /* Firefox 6.0+ */
  image-rendering: -webkit-optimize-contrast;/* Webkit */
  image-rendering: optimize-contrast;         /* Standards compliant */
  -ms-interpolation-mode: nearest-neighbor;   /* MS Specific extension*/
}
```

图 15-2 展示了这一属性在 Firefox 11 中所带来的外观效果，边缘变得更加清晰。

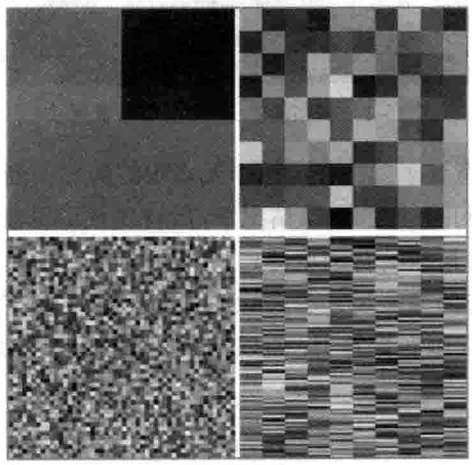


图 15-2 使用清晰边缘重新调整画布的大小

如第 6 章所述，在诸如 iPhone 4、iPad 3 和 Galaxy Nexus 一类高分辨率设备上，CSS 像素与显示设备的像素并非一一对应。这意味着若让画布元素保留它的默认分辨率，不设置 CSS 宽度，那么你就没有最大限度地利用显示设备的能力，举例来说，这意味着被绘制到画布元素上的文本是不清晰的。

为解决这一问题，可以检查窗口提供的一个名为 window.devicePixelRatio 的属性，该属性可以告诉你设备与 CSS 像素之间的比例倍数。借助这一倍数，可放大画布元素的像素尺寸，又可以保证 CSS 尺寸不变。通过加入上下文的一个 scale 方法，游戏再不必关心<canvas>元素是否已被重新调整大小，可继续使用 CSS 像素来定位游戏中的元素。代码清单 15-2 说明了实现方式。

代码清单 15-2：高分辨率设备的缩放调整

```
var $canvas = $("#mycanvas"),
    Canvas = $canvas[0],
    ctx = canvas.getContext("2d");

if (window.devicePixelRatio) {
  var pixelWidth = canvas.width,
      pixelHeight = canvas.height;
```

```
    canvas.width = pixelWidth * window.devicePixelRatio;
    canvas.height = pixelHeight * window.devicePixelRatio;

    $canvas.css({ width: pixelWidth, height: pixelHeight });
    ctx.scale(window.devicePixelRatio, window.devicePixelRatio);
  }
```

这段代码首先提取<canvas>元素的宽度和高度属性的原始值，接着使用 devicePixelRatio 来放大像素宽度和高度。然后，它把 CSS 宽度和高度设置成原始尺寸，这样画布就不会被重新设置大小。最后，它使用 scale 方法(本章后面介绍关于变形的更多内容)来放大所有的上下文调用。

15.2.2　提取渲染上下文

你可能会使用<canvas>标签来实现的大部分有趣事情都是借助于该元素的上下文实现的，正如你已在本书许多地方见到的那样，这一上下文是通过以下调用获得的：

```
var ctx = canvas.getContext("2d");
```

2D 是目前所有现代浏览器都可以支持的上下文，所有绘制调用一直都在该上下文(而非画布元素)中执行。

若浏览器不支持画布，getContext 方法就不会出现在画布元素中。也正如你之前在本书中所见，通过检查 getContext 方法是否存在于某个新建的<canvas>元素中，可以判断浏览器是否支持该标签：

```
var hasCanvas = document.createElement("canvas").getContext ? true : false;
```

此外，也可以检查该方法是否存在于页面上某个现有的<canvas>元素中。

本章自始至终用变量 ctx 来引用一个任意的 2D 渲染上下文，但毫无疑问，可为上下文指定任何变量，也可以拥有页面上多个画布元素的多个上下文。

注意：此外，还存在一个 webgl 上下文(根据浏览器的不同，有时也作为 experimental-webgl 提供)，该上下文公开 WebGL 的渲染 API，不过因为大部分移动设备都不支持 WebGL，所以本书不打算讨论这部分内容。

15.2.3　通过画布创建图像

其他唯一一个获得跨浏览器良好支持的方法是 canvas.toDataURL 方法，该方法返回一个代表了画布当前状态的快照图像的数据 URL。这一图像可生成一个<img>标签，或是可被保存到服务器上。该方法接收一个可选参数，参数指明要保存文件的类型为“image/ png”或“image/jpeg”(Chrome 还支持一种称为“image/ webp”的新图像类型)。若未传入该参

数，方法默认生成一个 png 文件。对于 JPEG 和 webp，你还可以传入第二个可选的质量参数。

要通过画布标签生成图像，可以这样写：

```
// Generate a PNG image
png = canvas.toDataURL();
png = canvas.toDataURL("image/png");
// Generate a JPG with quality 0.8
jpg = canvas.toDataURL("image/jpeg", 0,8);
```

可通过运行代码清单 15-3 中的代码来测试 toDataURL 方法，在每次单击或触摸画布时，这段代码都会抓取一个快照。

代码清单 15-3：to-data-url.html

```
<script src='jquery.min.js'></script>
<canvas id="mycanvas", width="400" height="400"></canvas>
<div id='snapshots'></div>

<script>
  var canvas = $("#mycanvas")[0],
      ctx = canvas.getContext("2d");
  function randInt(max) {
    return Math.floor(Math.random() * max);
  }
  function drawRandomRectangle() {
    var r = randInt(255), g = randInt(255), b = randInt(255),
        s = randInt(100), x = randInt(400), y = randInt(400);
    ctx.fillStyle = "rgb(" + r + "," + g + "," + b + ")";
    ctx.fillRect(x,y,s,s);
  }
  setInterval(drawRandomRectangle,50);
  $(canvas).on("click touchstart",function(e) {
    var url = canvas.toDataURL("image/png");
    $("<img>").({ src: url, width: 100, height:100 }).prependTo("#snapshots");
    e.preventDefault();
  });
</script>
```

这段代码创建一个画布元素，然后每隔 50 毫秒就把一个有着随机颜色和大小的正方形添加到页面上。单击或触摸画布元素所触发的事件调用 toDataURL 方法，创建一个新的<img>标签，把 src 特性设置成由 toDataURL 方法返回的 URL，然后把该<img>标签前置到一个名为 snapshots 的 中。每次单击都会生成一个新图像，因为<img>的宽度被设置成 100，所以，借助自己的每次单击，可看到随着时间的逝去画布发生了什么样的变化。

此外，W3C 规范还定义了一个输出 File 对象的 toBlob 方法，该方法节省了一些内存，因为文件可被写到硬盘中，且这样更便于使用。遗憾的是，截至撰写本书之时为止，该方法尚未在任何浏览器中获得了实现，所以，你应避免使用它(Firefox 定义了一个 mozGetAsFile

方法，不过这是非标准的，且使用的是一种不同语法)。

15.3　在画布上进行绘制

画布上下文提供了许多不同做法在<canvas>元素上进行绘制，其中四个主要方法分别用来绘制矩形、路径、文本和图像(该上下文还提供了一些直接修改像素数据的方法，这会在第 17 章中进行介绍)。除了图像，其他绘制方法还可以采用 stroke(笔画)方式进行绘制，意即只有轮廓会被绘制，或以 fill(填充)方式进行绘制，意即内里部分会被绘制。与 SVG 类似，画布也支持多种不同的线连接样式和端帽，同样与 SVG 类似的是，画布也使用渐变和模式来实现笔画和填充。

15.3.1　设置填充和笔画样式

画布上下文分别把当前笔画和填充样式的状态存放在 strokeStyle 和 fillStyle 属性中，这两个属性都是可读写的。你可为笔画和填充设置的最简单值是 CSS 颜色值，这些颜色值可以是形如“#F00”或“#FF0000”一类带井号前缀的普通十六进制颜色字符串，或是“rgb(255,0,255)”形式的 RGB 三元组字符串，也可以是类似“red”这样的命名颜色。

以下是一些例子：

```
ctx.fillStyle = "teal";

ctx.strokeStyle = "rgb(128,64,64)";

ctx.fillStyle = "#FF0000";

console.log(ctx.fillStyle);
// Logs the last value "#FF0000"

console.log(ctx.strokeStyle);
// Logs the strokeStyle converted hexadecimal as "#804040"
```

也可以使用渐变和模式来设置 fillStyle 和 strokeStyle，画布支持两种类型的渐变：线性的和径向的。通过调用以下上下文方法，可以实现这两种渐变的创建：

```
var linearGradient = ctx.createLinearGradient(x0, y0, x1, y1);
var radialGradient = ctx.createRadialGradient(x0, y0, r0, x1, y1, r1);
```

线性渐变从(x0,y0)开始，到(x1,y1)结束。它们会被绘制成一条无限宽的带，这条带垂直于由传入的点所连成的直线。

径向渐变生成一个锥形，所创建的锥形位于由传入的参数所定义的两个圆形之间，处在这两个圆形外部的区域是透明的。若希望只创建一个单环渐变，可以把第一个半径设置成 0，且把两个点(x0,y0)和(x1,y1)设置成相同的点。

在创建渐变后，需要给它加上颜色起止点(color stop)，这些颜色起止点定义颜色在从

渐变的起点到终点这一线路中所占据的具体百分比。要添加一个起止点，需要调用渐变的addColorStop 方法并传入一个介于 0～1 之间代表起止点位置的数值和一个颜色参数：

```
gradient.addColorStop(position, color);
```

例如，要创建一个到达中间位置时从黑变白然后变黑的渐变，可以这样写：

```
linearGradient.addColorStop(0,"#000");
linearGradient.addColorStop(0.5,"#FFF");
linearGradient.addColorStop(1,"#000");
```

可以把任何有效的 CSS 颜色定义当成第二个参数传递给 addColorStop。

最后一种可以创建的笔画和填充样式是模式，模式(pattern)是用来重复填满填充区域的图像，可通过调用以下方法创建：

```
var pattern = ctx.createPattern(sourceImage,repeatString);
```

sourceImage 可以是一个<img>元素，这包括使用 new Image()、<video>元素和<canvas>元素创建的对象；repeatString 是以下这些值："repeat"、"repeat-x"、"repeat-y"或"no-repeat"之一，你应该能认出来，这些字符串与那些用来设置 CSS 中的 backgroundImage 的重复值是一样的，它们分别与在两个方向上重复、仅水平方向重复、仅垂直方向重复和不重复这四种模式对应。

代码清单 15-4 给出的例子创建了两种类型的渐变，此外，它还通过离屏<canvas>元素生成了一个简单的模式，结果如图 15-3 所示。

代码清单 15-4：创建画布的渐变

```
<script src='jquery.min.js'></script>
<style> canvas { background-color:black; } </style>

<canvas id="mycanvas", width="600" height="400"></canvas>
<script>
  var canvas = $("#mycanvas")[0],
      ctx = canvas.getContext("2d"),
      width = canvas.width,
      height = canvas.height;
  var linearGradient = ctx.createLinearGradient(0,0,100,300),
      radialGradient = ctx.createRadialGradient(300,200,0,
                                                300,300,200);
  linearGradient.addColorStop(0,"#000");
  linearGradient.addColorStop(0.5,"#FFF");
  linearGradient.addColorStop(1,"#000");

  radialGradient.addColorStop(0,"#000");
  radialGradient.addColorStop(0.5,"#FFF");
  radialGradient.addColorStop(1,"#000");
```

```
    ctx.fillStyle = linearGradient;
    ctx.fillRect(0,0,200,400);
    ctx.fillStyle = radialGradient;
    ctx.fillRect(200,0,200,400);

    var patternCanvas = $("<canvas width='20' height='20'>")[0],
        patternCtx = patternCanvas.getContext("2d")

    patternCtx.fillStyle = "#777";
    patternCtx.fillRect(0,0,10,10);

    patternCtx.fillStyle = "#FFF";
    patternCtx.fillRect(10,10,10,10);

    ctx.fillStyle = ctx.createPattern(patternCanvas,"repeat");
    ctx.fillRect(400,0,200,400);
</script>
```

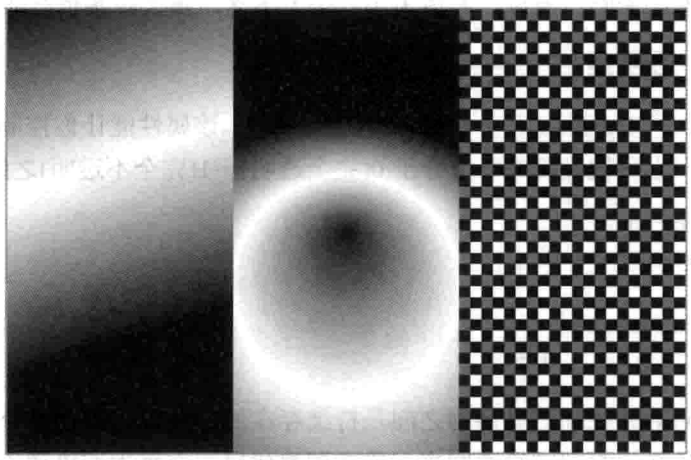


图 15-3　画布的渐变

这段代码创建一个 600 像素宽、400 像素高的画布，并各用一种不同的填充来绘制三个矩形。第一个包含了一个线性渐变，第二个是径向渐变，第三个是一个简单的通过离屏的 20×20 画布元素创建的模式填充。

这些渐变使用相对于整个画布(而非相对于个别矩形)的位置进行创建，这意味着移动 fillRect 会导致矩形渐变的位移。要把渐变用在精灵上，需要使用“使用画布变形矩阵”一节中介绍的平移、旋转和缩放方法来移动包含了渐变的元素，而不是在具体的画布位置上绘制它们。

15.3.2　设置笔画细节

虽然渐变和模式既可用在笔画上也可用在填充上，不过一些笔画特定的设置能够让你

控制定义了笔画的直线的画法，这些属性包括以下这些：

```
// Sets the line width in current units (default 1)
ctx.lineWidth = width;

// Sets the cap style on the end of lines
// possible values are "butt", "round", "square" (default "butt")
ctx.lineCap = "butt";

// Sets the style at corners between two lines
// possible values are "round", "bevel", "miter" (default "miter")
ctx.lineJoin = "miter";

// Sets the max size of a miter join in current units
// Prevents mitered corners from getting too large at small angles.
ctx.miterLimit = 10;
```

关于笔画的细节，没有太多要说的，除了一点，那就是 lineWidth 和 miterLimit 属性会受到当前变形状态的影响。所以，若放大了一个上下文，那么线条也会随之变宽。

15.3.3 调整不透明度

此外，渲染上下文还提供了一个 globalAlpha 属性，该属性能让你控制其所渲染的任意对象的不透明度，它可被设置成一个介于 0(完全透明)～1(完全不透明)之间的值。

```
// Fully opaque
ctx.globalAlpha = 1;

// 50% transparent
ctx.globalAlpha = 0.5;
```

该属性在不同的路径和渲染调用之间是持续存在的，所以，若在某个地方对其进行了修改，那么过后务必记得把它给改回来，或者调整代码，在每次渲染调用之前先把它设置成 1。

15.3.4 绘制矩形

画布所支持的最简单绘制方法是用来绘制任意尺寸矩形的方法，它提供了三个方法，分别用来清除矩形、创建一个实心矩形和创建一个矩形边框：

```
// Clear the specified rectangle,
// setting each pixel to black and transparent
ctx.clearRect(x, y, w, h)

// Create a filled-in rectangle using the current fillStyle
ctx.fillRect(x, y, w, h)
```

```
// Create a rectangle outline using the current strokeStyle
ctx.strokeRect(x, y, w, h)
```

这些方法的执行速度通常都很快(在使用渐变或模式填充时除外)，clearRect 常用来在两帧之间清除画布。

15.3.5　绘制图像

你已经在第 1 章中了解了三个 drawImage 方法，这里再次给出，以供参考：

```
// Draw an image at x,y at its full size
ctx.drawImage(image, x, y)

// Draw an image at x,y rescaled to width w and height h
ctx.drawImage(image, x, y, w, h)

// Draw the portion of the image defined by the rectangle sx,sy and sw,sh
// at x,y with width w and height h
ctx.drawImage(image, sx, sy, sw, sh, x, y, w, h)
```

第一种版本在画布的某个位置上按照图像的尺寸来绘制完整图像。

第二种版本绘制一个尺寸已被重新调整为 w 宽和 h 高的完整图像，若画布已使用 CSS 进行缩放，或已经应用了变形，那么这可能并不意味着是 w 像素宽和 h 像素高。例如，如前所述，若出于 Retina iOS 设备的缘故，画布已被缩小一半，那么你可能会希望加载两倍于分辨率的图像，然后以一半宽度和高度来绘制它们，这样就能获得最佳视觉效果。

第三种版本就是用在 Quintus 代码中的那个版本，Quintus 使用该版本从精灵表中提取出图像的某个部分并把它绘制到画布上。

各版本中的图像(image)参数可以是一个 Image 对象(相当于<img>这一 DOM 元素)、另一个<canvas>元素或是一个<video>元素。

15.3.6　绘制路径

路径是画布提供的最复杂绘制工具，也是最强大的工具。它们能够让你在画布上绘制任意形状和曲线。在完成了路径的定义后，可以调用 stroke 来绘制路径轮廓，或调用 fill 来绘制路径所形成的实心形状。若路径是未关闭的，那么在你调用 fill 时路径会被隐式关闭。

除了调用 stroke 和 fill 外，还可以通过调用 clip 来使用现有路径定义一个剪裁区域。这可把之后的绘制命令局限在之前已绘制好的路径上，直至调用 restore 为止。HTML5 规范定义了一个名为 resetClip 的方法，但截至撰写本书之时为止，该方法尚未在任何浏览器中获得了很好实现。

使用点以及点之间的连接段来定义路径，这些段可以是直线、圆弧或曲线。每个路径都由一个或多个子路径构成，这些子路径既可是闭合的(意味着最后一个点连接到了第一个点上)也可是开放的。

要创建一个路径，先调用 ctx.beginPath()，接着是任意数目的路径命令，然后调用 ctx.fill()或 ctx.stroke()。若希望创建多个子路径，还可以调用 ctx.closePath()来关闭子路径，同时隐式创建一个新路径，该路径将上一路径的最后一个点作为起点。此外，还可以调用 ctx.moveTo(x, y)来移动下一个命令的起始点，若前面的路径已经有多个点，则隐式创建一个新的子路径；不过调用 moveTo 不会隐式关闭前一个路径。在调用 stroke 或 fill 时，当前路径中的所有子路径都会受影响。

画布提供了 7 种不同的用于绘制各种路径段的命令，这些命令可混搭使用。以下给出了每种命令的详细说明：

- ctx.lineTo(x, y)：在(x,y)处添加一个新的点，并用一条直线把该点和前一个点连接起来。
- ctx.quadraticCurveTo(cpx, cpy, x, y)：在(x,y)处添加一个新点，并使用一条控制点为(cpx,cpy)的二次贝塞尔曲线来把该点和前一个点连接起来。
- ctx.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)：将一个新点(x,y)添加到子路径中，并使用由控制点(cp1x,cp1y)和(cp2x,cp2y)定义的三次贝塞尔曲线来把该点和前一个点连接起来。
- ctx.arcTo(x1, y1, x2, y2, radiusX, [, radiusY, rotation])：使用所定义的半径在(x1,y1)和(x2,y2)之间添加一条弧线，此外，它还使用一条直线把(x1,y1)和位于子路径中的前一个点连接起来。若提供了 radiusY 和 rotation，arcTo 绘制某个椭圆的部分弧线，该椭圆从 x 轴正方向开始逆时针旋转 rotation 弧度。
- ctx.arc(x, y, radius, startAngle , endAngle [, anticlockwise])：该命令绘制一段以(x,y)为传入半径(radius)的起点，分别以 startAngle 和 endAngle 为起始角和结束角(定义成弧度)的圆弧。若逆时针方向(anticlockwise)的值被设为 true，则圆弧将按逆时针方向绘制。
- ctx.ellipse (x, y, radiusX , radiusY , rotation, startAngle , endAngle, anticlockwise)：该命令绘制某个椭圆位于 startAngle 和 endAngle 之间的部分弧线，该椭圆以 x 和 y 为给定半径的起点，它的长半轴从 x 轴正向开始逆时针旋转 rotation 弧度。该命令会把所绘制椭圆弧的起始点和位于子路径中的前一个点连接起来。
- ctx.rect (x, y, w, h)：该命令绘制一个新的矩形子路径并关闭该子路径，子路径由定义矩形四角的四个点构成。该命令与 fillRect 和 strokeRect 类似，不过它生成的是一个子路径。

如你所见，可以采用许多方法来绘制矢量路径，不过，在画布上完成绘制后，路径就会被转换成像素数据，路径细节就会丢失。因此，规范已做了一些更新，允许创建一个新的 path 对象，通过调用 ctx.stroke(path)和 ctx.fill (path)，你可重用该对象。不过截至撰写本书之时为止，尚未有任何浏览器在画布元素中加入了对该对象的支持。

注意：贝塞尔曲线是一些常用在计算机图形中的参数曲线，它们使用一个起点和一个终点以及一个(二次曲线)或两个(三次曲线)控制点进行定义。这些控制点定义曲线在起点和终点之间的大小和形状。

在计算二次贝塞尔曲线在某个特定时间的点的公式中，时间 t 的取值介于 0 ~ 1 之间，该公式可被写成一个二次矢量方程：

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_c + t^2 P_1$$

或可用 JavaScript 表示成：

```
var x = (1-t)*(1-t)*p0.x + 2*t*(1-t)*pc.x + t*t*p1.x
var y = (1-t)*(1-t)*p0.y + 2*t*(1-t)*pc.y + t*t*p1.y
```

其中，已知起点和终点是 p0 和 p1，控制点是 pc。三次贝塞尔公式则更加复杂，幸而，浏览器会为你处理这些曲线的绘制事宜。若想感受一下控制点是如何影响弧线的，你可加载包含在本章代码中的 bezier.html 文件试一试。

15.3.7　在画布上渲染文本

如你在第 1 章的标题画面中所见，画布还具有在画布元素上渲染文本的能力。与绘制矩形类似，画布提供两个方法来渲染文本：

```
ctx.fillText(str, x, y);
    ctx.strokeText(str, x, y);
```

第一个方法 fillText 使用实心字符在位置(x,y)上绘制一个文本字符串(就是一般文本)，而第二个方法 strokeText 则只绘制轮廓。默认情况下，文本是左对齐的(至少那些从左向右书写的语言是这样)，不过，可以使用 ctx.textAlign 属性的以下任意值来修改对齐方式：

```
ctx.textAlign = "left";   // Left aligned from x,y
  ctx.textAlign = "right";  // Right aligned from x,y
  ctx.textAlign = "center"; // Centered on x,y
  ctx.textAlign = "start";  // Same as "left" in left-to-right languages
  ctx.textAlign = "end";  // Same as "right" in left-to-right languages
```

默认值是 start，不过若要在某个具体位置上水平居中显示文本，可使用 center。

垂直对齐则通过 ctx.textBaseLine 属性来控制，通过把这一属性设置成不同的值，可以控制文本相对于传入的 y 值的摆放位置：

```
  ctx.textBaseline = "top";     // text baseline is top of the em square
  ctx.textBaseline = "middle";  // text baseline is middle of the em square
  ctx.textBaseline = "alphabetic"; // text is on normal alphabetic baseline
  ctx.textBaseline = "bottom";  // text baseline is bottom of the em square
```

截至撰写本书之时为止，尚有其他两个选项未获得支持(ideographic 和 hanging)。textBaseLine 的默认值是 alphabetic，在定位文本时，该值有时可能不太好用，把 textBaseLine 设置成 top 或 bottom 可能更容易把文本准确置于你所希望的地方。

最后一个属性是 ctx.font，该属性使得你能使用 CSS 风格的字体字符串来设置字体。这个字符串可包含任何内容，从仅是一个包含了大小和字体类型的声明到一个包含了完整的样式、变形、加粗、大小和类型的声明都可以。若传入无效的字体声明，那么赋值会失败但不会报错。

```
ctx.font = "20px Arial";    // Set the font to 20px

// Font set to italic, bold 40px Lobster, line height isn't actually used
ctx.font = "italic normal bold 40px/20px Lobster";

// Setting just the size or family isn't valid and is ignored
ctx.font = "40px";  // INVALID
ctx.font = "Arial"; // INVALID

// Still returns "italic normal bold 40px/20px Lobster"
console.log(ctx.font);
```

任何可在页面上使用的字体都可在画布中使用，这意味着任何通过@font-face 加载的字体都是可随意使用的。

此外，还可以使用 ctx.measureText 方法来测量文本字符串的宽度，它返回一个 TextMetrics 对象，这一来自 HTML5 规范的对象给人的感觉是应该包含了许多有趣的属性，比如说水平的和垂直的边界框，以及以 em 为单位的高度信息等。但实际上，浏览器只实现了所渲染文本的宽度属性。

```
var m = ctx.measureText("This is some text");
  // Width of "This is some text" with the current font
  console.log(m.width);
```

也许以后还会出现更多属性，不过现在的主要用例是预先算出某段文本的宽度，以备定位或检查单击时使用。

15.4 使用画布变形矩阵

虽然之前提到过，但本书尚未详细介绍过画布提供的变形矩阵。这一变形矩阵使得你能够平移、旋转和缩放任何被绘制在画布上的元素，包括图像和路径等。

这一矩阵的作用方式与第 14 章中的 SVG 变形的作用方式是一样的，不过除此之外，画布还提供了一种轻松保存和恢复矩阵状态的做法，目的是支持绘制命令的简单嵌套(SVG 不存在这一问题，因为 DOM 中的元素是彼此上下嵌套的)。

15.4.1　了解基本的变形

与 SVG 和 CSS3 类似，画布提供了标准的基本 2D 变形：平移、缩放和旋转。在实施某个变形之后，它就会作用在你所绘制的每样东西上，直至你改变它为止。

```
ctx.translate(x,y);

ctx.scale(sx,sy);

// Rotate takes an angle in radians
ctx.rotate(angle * Math.PI / 180);
```

若需要实施一个不是由内置的平移、缩放或旋转来处理的自定义变形(如倾斜)，你还可以直接使用一些矩阵值来调用 ctx.transform：

```
ctx.transform(a, b, c, d, e, f);
```

在调用任何上述方法时，在内部，浏览器都会创建一个要执行所需变形的转换矩阵，然后将其与当前变形矩阵(current transformation matrix)相乘。这意味着操作顺序很重要——这通常也是大家不太容易理解的地方。

最好按照全局到局部这样的顺序来实施变形，这意味着若希望把某个对象移到画布的某个位置上，并让它旋转某个角度显示，那么你首先应实施较为全局性的变形(平移)，然后实施较为局部的变形(旋转)，若按照相反的顺序实施，就意味着旋转是一个全局性的变形，所以，元素应该围绕它的平移来旋转。

若元素需要围绕它的中心旋转且元素并非以(0,0)为中心，那么你也需要用平移到中心的变形和从中心移回的变形来前后括住该旋转。

例如，若要围绕某个矩形的中心旋转该矩形，然后把它平移到画布的某个位置上。那么要按照相反的顺序来实施这些变形(若从后向前阅读以下代码会更便于理解一些)。

```
// Move the object to the correct spot
ctx.translate(250,200);

// Uncenter the element back to its original spot
ctx.translate(50,50);

// Rotate it
ctx.rotate(45 * Math.PI / 180);

// Center it
ctx.translate(-50,-50);

// Draw it
ctx.fillRect(0,0,100,100);
```

若修改任何这些变形的顺序(除了前面两个平移，它们是可交换顺序的)，你最终会得到一个改变了位置的矩形，虽然你原本打算改变的是旋转角度。

15.4.2 保存、恢复和重置变形矩阵

因为嵌套变形是一种常见的需求，所以画布上下文提供了两种简便方法来保存和恢复变形矩阵的状态：

```
ctx.save(); // Save the state
  ctx.translate(...);
  ctx.scale(...);
  ctx.rotate(...);
   ...
ctx.restore(); // Restore the matrix
```

使用 save 和 restore 能够让你做到在不影响任何其他可能依赖于变形矩阵的代码的情况下，实施任意数量的子变形，然后恢复状态。

此外，还可以使用 setTransform 把变形矩阵重置回一个未知状态：

```
ctx.setTransform (a, b, c, d, e, f)
```

若需要把矩阵重置成单位矩阵(该矩阵不实现任何变形)，可以运行

```
ctx.setTransform(1,0,0,1,0,0);
```

必须小心使用变形的完全重置，因为这可能会导致异常结果，例如，如前所示，若已为 Retina 图形调整画布的尺寸，那么重置就会导致意外情况出现。

15.4.3 绘制雪花

为了彻底展示变形的强大功能，接下来构建一个递归式的随机“雪花生成器”，该生成器生成一些很有趣的类似分形的递归图案。

这里的想法是，每个雪花都可以定义成具有多个朝不同方向展开的分支，然后每个分支又都有一组更小的子分支在某个角度范围内展开，如此下去，直至达到所设定的层数限制为止。因为递归的每个步骤都是相同的，所以可以只使用一个调用了自身的方法。这些嵌套调用的唯一不同之处在于变形矩阵的状态是已经设置好的，这样子分支就会被绘制到父分支的下面。

图 15-4 展示了几近无数可能的雪花中的一个比较有趣的输出，雪花生成器的代码如代码清单 15-5 所示。

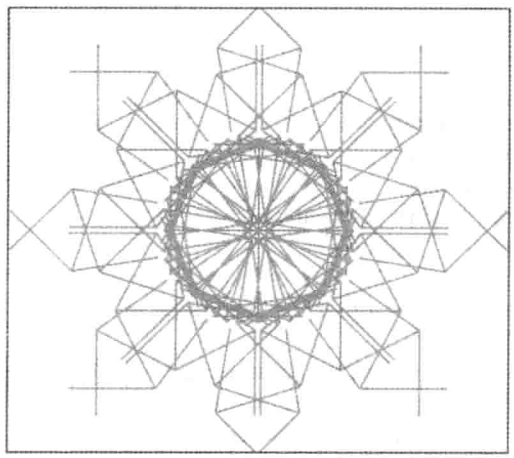


图 15-4　输出的雪花图案

代码清单 15-5：随机生成雪花

```
<script src='jquery.min.js'></script>
<style> canvas { background-color:white; } </style>

<canvas id="mycanvas", width="600" height="400"></canvas>

<script>
  var canvas = $("#mycanvas")[0],
      ctx = canvas.getContext("2d");

  function randInt(max) {
    return Math.floor(Math.random() * max);
  }
  function randomSnowflake() {
    var rootBranches = randInt(8)+1,
        childBranches = randInt(8)+2,
        childSpread = Math.random()*0.5 + 0.5,
        size = 50 + randInt(50),
        level = randInt(4)+1,
        distance = Math.random()*0.5 + 0.5;

    function drawSnowflake(branches,spread,level) {
      var angle;
      for(var i=0;i<branches;i++) {
        if(spread == 1) {
          // Don't overlap branches of we are rotating fully
          angle = Math.PI * 2 * spread * (-0.5 + i/branches);
        } else {
          angle = Math.PI * 2 * spread * (-0.5 + i/(branches-1));
        }

        ctx.save();
```

```
        // Rotate to point straight up for this branch
        ctx.rotate(angle);

        // Draw this branch
        ctx.beginPath();
        ctx.moveTo(0,0);
        ctx.lineTo(0,size*distance);
        ctx.stroke();

        // Draw child branches if necessary
        if(level > 0) {
          // Move to the end of the branch and scale down
          ctx.translate(0,size*distance);
          ctx.scale(distance,distance);
          drawSnowflake(childBranches,childSpread,level-1);
        }
        ctx.restore();
      }
    }

    ctx.clearRect(0,0,600,400);
    ctx.save();

    // Generate a random color
    var r = randInt(255), g = randInt(255), b = randInt(255);
    ctx.strokeStyle = "rgb(" + r + "," + g + "," + b + ")";
    ctx.lineWidth = 2;

    // Center the initial branches
    ctx.translate(300,200);

    drawSnowflake(rootBranches,1,level);
    ctx.restore();
  }
  randomSnowflake();
  $(canvas).on('click',randomSnowflake);
</script>
```

外部方法 randomSnowflake 随机生成一些属性值，分别设置分支数、展开角度、大小、递归层数和随机的雪花颜色等。接着，它调用递归方法 drawSnowflake，该方法为每个已旋转到正确角度的分支绘制一条线。然后，它查看是否还有子层要绘制，若是，则设置子分支的变形矩阵，然后使用一些已更新的参数来再次调用自身。

因为所有绘制调用都放在 ctx.save()和 ctx.restore()调用之间，所以，每个分支都可以把它的变形矩阵传给它的子分支而不会影响其他任何分支。

每次在你单击画布时，就会生成一朵新的雪花。仅绘制一些线条和改变几个参数就能制造出如此多种可能性，嵌套变形的强大可见一斑。

15.5　应用画布效果

作为本章的收尾，画布还提供了其他两种值得一提的效果：阴影和合成效果。

15.5.1　添加阴影

画布上下文提供了一种方式来给任何已绘制的元素加上投影，这些元素包括文本、矩形、路径和图像等。上下文使用一组四个属性对其加以控制：

```
// CSS color shadow, accepts RGBA, RGB, hexadecimal
ctx.shadowColor = "rgba(255,255,255,0.5)";

ctx.shadowOffsetX = 4;    // horizontal shadow offset
ctx.shadowOffsetY = 4;    // vertical shadow offset
ctx.shadowBlur = 10;      // distance for shadow to fade out
```

可使用阴影来生成普通的投影效果，做法是使用较暗的 shadowColor 值；又或者可通过把阴影的偏移值设置为零并使用一种较亮的颜色来制造一种微妙的光晕效果。

警告：相比于没有阴影的元素，带阴影的元素的绘制显然更耗费处理器资源，所以要慎用阴影，可考虑把效果预渲染到离屏画布缓冲区中。

15.5.2　使用合成效果

画布上下文还提供了一个名为 globalCompositeOperation 的属性，该属性控制画布合成现有内容和新绘制到画布上的元素的方式。规范为这一属性定义了 11 个不同的可能值，默认值是 source-over，即把新绘制的元素置于已有内容之上。

遗憾的是，截至撰写本书之时为止，对这些有趣的合成操作的跨浏览器一致性支持还少得可怜，所以，这一属性目前还处在跟踪进展状况而非投入使用阶段。

作为直接从规范中摘录的内容，表 15-1 列出了每种操作的预期结果。

表 15-1　合成操作

操　作	说　明
source-atop	A 在 B 之上。在任何两个图像都不透明的地方显示源图像(A)，在任何目标图像(B)不透明但源图像透明的地方显示目标图像，其他地方都透明显示
source-in	A 在 B 中。在任何源图像(A)和目标图像(B)都不透明的地方显示源图像，其他地方都透明显示
source-out	A 在 B 之外。在任何源图像(A)不透明且目标图像(B)透明的地方显示源图像，其他地方都透明显示

(续表)

操　作	说　明
source-over(默认)	A 覆盖 B。在任何源图像(A)不透明的地方显示源图像，其他地方显示目标图像(B)
destination-atop	B 在 A 之上。与 source-atop 的做法相同，只是要在做法描述中互换源图像(A)和目标图像(B)的位置
destination-in	B 在 A 中。与 source_in 的做法相同，只是互换了源图像(A)和目标图像(B)的使用
destination-out	B 在 A 之外。与 souce-out 的做法相同，只是互换了源图像(A)和目标图像(B)的使用
destination-over	B 覆盖 A。与 source-over 的做法相同，只是互换了源图像(A)和目标图像(B)的使用
lighter	A 加上 B。显示源图像(A)和目标图像的(B)的叠加效果，颜色值最大只能到达255(100%)
copy	A(B 被忽略)。显示源图像(A)而非目标图像(B)
xor	A 异或 B。源图像(A)和目标图像(B)之间的一个异或运算

很遗憾，Safari 是未能正确处理其中少数几个操作的浏览器之一，这其中包括了对 source-in、source-out、destination-in、destination-atop、copy 的不正确渲染。

Firefox 则未能正确处理 copy 操作，不过桌面 Chrome 和 Android Chrome 的最新版本就都能正确实施所有的操作。可以通过图 15-5 了解各种操作的视觉效果，也可通过运行本章代码中的 composition.html 文件来试用一下这些操作。

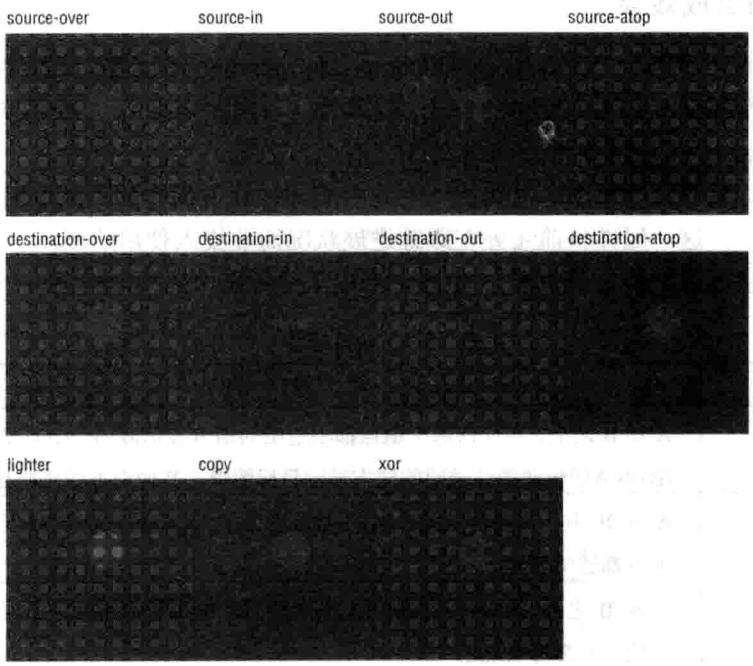


图 15-5　合成操作的效果

截至撰写本书之时为止，只有 source-over(默认)、source-atop、destination-over、destination-out 和 lighter 等操作获得了所有浏览器的良好支持。

15.6　小结

现在，你已经了解到更多关于画布元素 API 的使用信息，其中包括如何使用 2D 画布，如何充分发挥它的能力，而这又包括了像素尺寸的特殊性，以及 2D 上下文的各种渲染功能等。此外，你还了解画布的各种矢量绘制功能，以及如何设置渐变和模式填充。最后，你了解了如何使用诸如阴影和合成效果的画布效果。

第16章

实现动画

本章提要

- 定义动画 API
- 构建动画系统
- 创建基于画布的视口
- 创建视差背景

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 16 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

16.1 引言

第 11 章中的精灵表支持允许使用了精灵表的 Sprite 通过修改精灵的帧属性来播放动画，虽然对于用到单一动画的简单游戏来说，这一方法是可行的，但一些较复杂的游戏(如第 18 章中构建的平台动作游戏)需要一个更健壮的系统来处理动画。为此，本章构建了一个支持较复杂行为的动画系统，此外，本章还研究了构建及动态显示基于画布的视差滚动背景所需的一些技术。

16.2 构建动画地图

一个健壮的 Quintus 动画系统应该能够实现两大目标，第一个目标是借助名称来简化

动画的触发，并且不必关心动画的播放速度或播放帧。第二个目标是把动画系统和实体的事件挂接起来，这样动画就能触发事件，从而更方便地设定动作和行为的发生时机。

16.2.1 确定动画 API

首先考虑一下一个好的动画系统 API 看起来的可能样子。要考虑的内容主要有两方面，第一个是定义动画的方法，其次是播放动画的方式。第一个方面很简单：需要一种方式来设置组成动画的帧，此外，还要设置动画的其他所有细节。代码清单 16-1 展示了这部分 API 的工作方式。

代码清单 16-1：动画 API

```
Q.animations('player', {
  run_right:  { frames: _.range(0,10) },
  run_left:   { frames: _.range(10,20) },
  stand:      { frames: _.range(30,25), rate: 1/5 },
  fire:       { frames: _.range(25,30), loop: false, rate: 1/30 },
  die:        { frames: _.range(30,45), rate: 1/5, next: 'dead' },
  dead:       { frames: [ 45 ] }
});
```

在上述代码中，Q.animations 创建了一个名为 player 的精灵动画地图，并传入一个定义了帧和其他所有动画细节的哈希，这些细节包括动画是否不应循环、是否有一个对速率的重写，以及是否在当前动画完成之后播放另一个动画。

作为传入诸如[0 , 1, 2, 3, 4, 5, 6]一类较长的帧数组的快捷方式，通过调用_.range(0,7)，可使用下划线快捷方法_.range 完成同样的操作(注意，_.range 也会生成一个长数组，不过第二个数字参数不包括在内)。

在上例中，玩家拥有左跑动和右跑动的循环动画，以及一个以较慢速率播放的原地站立循环动画，然后是以较快速率播放且不循环的发射动画，再接着是一个垂死动画，在完成自身的播放后，该动画自动播放一个 1 帧的死亡动画。

定义完动画后，现在是时候找出播放它们的做法了。因为播放(play)常是一个与动画联系在一起的术语，所以一个简单的播放方法接收要播放动画的名称和一个可选的优先级值为参数。把优先级添加到播放方法中，这种做法允许具有较高优先级的动画(如攻击)覆盖具有较低优先级的动画(如跑动或走动)。

代码清单 16-2 展示了在加入动画驱动后，玩家精灵看起来的可能样子。

代码清单 16-2：动画化的玩家精灵

```
Q.sheet('player_animations', 'dummy.png',{ tilew: 96, tileh: 96});

Q.Player = Q.Sprite.extend({
  init:function(props) {
    this._super(_(props).extend({
      sprite: 'player',
```

```
      sheet: 'player_animations',
      rate: 1/15
    }));

    this.add('animation');
    this.bind('animEnd.fire',this,function() { console.log("Fired!"); });
    this.bind('animLoop.run_right',this,function() {
      console.log("run right");
    });

    this.bind('animLoop.run_left',this,function() {
      console.log("run left"); }
    );
    Q.input.bind('fire',this,"fire");
  },

  fire: function() {
    this.play('fire',1);
  },

  step: function(dt) {
    if(Q.inputs['right']) {
      this.play('run_right');
    } else if(Q.inputs['left']) {
      this.play('run_left');
    } else {
      this.play('stand');
    }
    this._super(dt);
  }
});
```

玩家类的 init 方法定义了一个 rate 属性，该属性设置精灵动画的默认播放速度，它还定义了一个指明控制动画的精灵的 sprite 属性和一个绑定玩家和精灵表的标准 sheet 属性。接着，方法把 animation 组件加进来，该组件会把 play 方法添加到玩家对象中。然后，方法定义几个回调，在特定动画完成播放或是已经完成帧的单次遍历之后，相应的回调就会被调用。前者可用来触发其他动作，后者则可用来触发周期性的行为，比如说玩家在奔跑时歇口气，或是在水下时耗尽了氧气，需要慢下来换口气。

把精灵动画地图和精灵表分开存放，这种做法简化了两者的更换。例如，可拥有这样的一些精灵表，这些精灵表与动画帧匹配，但代表了不同的角色。

step 方法被设置成根据用户的移动动作来以最低默认优先级播放特定动画，而 fire 方法则以一个较高优先级(1)来播放发射动画，这样发射动画就可以优先于移动动画播放。因为发射动作被设置成一个非循环的动画，所以在该动画播放完毕后，移动动画会继续播放。

16.2.2 编写动画模块

在定义了动画 API 后，现在可以实现该模块了。该模块被命名为 Quintus.Anim，除了

包含几个定义和检索动画的辅助方法之外，该模块还包含了一个可被添加到 Sprite 对象(实际上，任何拥有一个 frame 属性和一个 step 方法的对象都可以)中的 animation 组件。

该动画组件使用一个调用了 animation.play 的 play 方法来扩展 Sprite 并设置动画，此外，它还绑定了 step 事件，目的是更新当前帧并在需要时触发一些事件。

该模块的代码如代码清单 16-3 所示，这些代码应被添加到一个名为 quintus_anim.js 的新文件中。

代码清单 16-3：Quintus.Anim 模块

```
Quintus.Anim = function(Q) {
  Q._animations = {};
  Q.animations = function(sprite,animations) {
    if(!Q._animations[sprite]) Q._animations[sprite] = {};
    _.extend(Q._animations[sprite],animations);
  };

  Q.animation = function(sprite,name) {
    return Q._animations[sprite] && Q._animations[sprite][name];
  };

  Q.register('animation',{
    added: function() {
      var p = this.entity.p;
      p.animation = null;
      p.animationPriority = -1;
      p.animationFrame = 0;
      p.animationTime = 0;
      this.entity.bind("step",this,"step");
    },
    extend: {
      play: function(name,priority) {
        this.animation.play(name,priority);
      }
    },
    step: function(dt) {
      var entity = this.entity,
          p = entity.p;
      if(p.animation) {
        var anim = Q.animation(p.sprite,p.animation),
            rate = anim.rate || p.rate,
            stepped = 0;
        p.animationTime += dt;
        if(p.animationChanged) {
          p.animationChanged = false;
        } else {
          p.animationTime += dt;
          if(p.animationTime > rate) {
            stepped = Math.floor(p.animationTime / rate);
```

```
            p.animationTime -= stepped * rate;
            p.animationFrame += stepped;
          }
        }
        if(stepped > 0) {
          if(p.animationFrame >= anim.frames.length) {
            if(anim.loop === false || anim.next) {
              p.animationFrame = anim.frames.length - 1;
              entity.trigger('animEnd');
              entity.trigger('animEnd.' + p.animation);
              p.animation = null;
              p.animationPriority = -1;
              if(anim.trigger) {
                entity.trigger(anim.trigger,anim.triggerData)
              }
              if(anim.next) { this.play(anim.next,anim.nextPriority); }
              return;
            } else {
              entity.trigger('animLoop');
              entity.trigger('animLoop.' + p.animation);
              p.animationFrame = p.animationFrame % anim.frames.length;
            }
          }
          entity.trigger("animFrame");
        }
        p.sheet = anim.sheet || p.sheet;
        p.frame = anim.frames[p.animationFrame];
      }
    },

    play: function(name,priority) {
      var entity = this.entity,
          p = entity.p;
      priority = priority || 0;
      if(name != p.animation && priority >= p.animationPriority) {
        p.animation = name;
        p.animationChanged = true;
        p.animationTime = 0;
        p.animationFrame = 0;
        p.animationPriority = priority;
        entity.trigger('anim');
        entity.trigger('anim.' + p.animation);
      }
    }

  });
};
```

如你所见，该组件的主体部分是负责更新 frame 的 step 方法，若流逝时间已足够长，该方法就更新 frame，然后，它会触发一些事件。不过，在讨论这部分内容之前，我们现

在先来逐段讲解这一大块代码。

首先是三个声明，这些使得你能设置和检索动画，动画存放在 Q._ animations 属性这一嵌套哈希中。使用精灵名称和一组 animation 来调用 Q.animations，你就可以把这些动画添加为该精灵表的可用动画。使用精灵名称和动画名称调用 Q.animation 则返回该动画的详细信息。

每个动画都需要一个帧数组，不过为增加灵活性，还需要支持其他一些属性，这些属性能把各个动画的定制和动画间的链接变得更容易一些：

```
{
  frames: [0,1,2,3],   /* An array of frames in the sheet */
                       /* can be created with _.range(0,3) */
  /* Optional Parameters */
  sheet: 'sheetName',  /* An override for the sheet, if used,
                       must be on all animations */
  loop: false,         /* Loop animation (default: true) */
  rate: 1/30,          /* Frame rate override for this animation */
  next: 'animName',    /* Animation to auto-play after this one */
  trigger: 'event',    /* Custom event to trigger when done */
  trigerData: { .. }   /* Optional custom trigger data */
}
```

animation 组件只在实体的接口中添加了一个被公开的方法 play，如前所述，该方法接收一个动画名称和一个可选的优先级为参数。

被扩展到实体中的 play 方法仅是组件直接创建的方法(在末尾处列出)的一个代理，该方法首先检查你是否真要更换动画，因为若调用打算播放的动画与已在播放的动画是同一个动画，那么它会继续播放该动画。接下来，若动画已被更改，且传入的优先级高于当前正在播放的动画，那么动画的属性就会更新，而且会触发两个 anim 事件：一个是一般的 anim 事件；另一个是特定于将被播放动画的 anim 事件。

step 方法完成了大部分工作，它首先检查是否存在要播放的动画，若是，就接管对精灵的 frame 属性的控制。

它然后查看这是不是某个新动画的第一帧，若不是，就基于速率(这是来自精灵的速率或是每个动画都可以重新设置的速率)更新动画时间，并根据时间步来使用所需的帧数推进动画。

接下来，它查看动画是否已更新自身的这一帧，若是，则还要多做一些工作。首先，代码检查当前帧是否已是动画的最后一帧，若是，则再查看这是否为只循环一次的动画，或者该动画是否设置了下一个要播的动画。若这两个条件之一成立，为安全起见，方法把该帧设置成动画的最后一帧，接着触发两个 animEnd 事件。然后，它把动画(animation)和 animationPriority 都重置成默认值。最后，它触发一个自定义事件或播放下一个动画，前提是相应的属性已经设置。

若这是一个循环动画，那么方法触发 animLoop 事件，发出单次循环已经播放完毕的信号，然后使用取模运算符来确保 animationFrame 的值位于帧数范围内。

最后，在其他所有事情都完成后，方法设置精灵的 frame 属性，并选择在动画设置了 sheet 属性的情况下设置精灵的动画精灵表。

16.2.3　测试动画

为测试动画功能，接下来创建一个能够在舞台上闲逛的简单动画精灵。首先创建一个名为 animation.html 的 HTML 文件，然后将代码清单 16-4 中的代码输入其中。

代码清单 16-4：动画的自建 html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
user-scalable=0, minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Animation</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>
    <script src='js/quintus_anim.js'></script>
    <script src='animation.js'></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

这就是基本的 Quintus 自建代码，只不过加入了新的 quintus_anim.js 模块而已。为将 jQuery 和 underscore 这两个依赖库还有那些一步步构建出来的 Quintus 代码与所有的例子代码分开存放，从本章开始，引擎代码被放到一个单独的 js/目录中。

接下来创建前面提到的 animation.js 文件，然后将代码清单 16-5 中的代码放入其中。这段代码定义了本章开头所介绍的用户控制的玩家角色的一个精简版本，该角色能够四处走动，且能够触发一个发射动画。要运行这一代码，需要用到分别位于本章代码的 images/和 data/子文件夹中的一些图像和 sprites.json 文件。此外，因为经由 Ajax 加载 sprites.json 文件的缘故，你还需要通过本地主机(localhost)来启动该例。

代码清单 16-5：一个基本的行走演示

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites,Scenes,Anim')
```

```
                    .setup('quintus', { maximize: true })
                    .controls()
Q.Player = Q.Sprite.extend({
  init:function(props) {
    this._super(_(props).extend({
      sheet: 'man',
      sprite: 'player',
      rate: 1/15,
      speed: 700
    }));
    this.add('animation');
    this.bind('animEnd.fire',this,function() {
                                    console.log("Fired!");
                                   });
    this.bind('animLoop.run_right',this,function() {
                                        console.log("right");
                                       });
    this.bind('animLoop.run_left',this,function() {
                                       console.log("left");
                                      });
    Q.input.bind('fire',this,"fire");
  },
  fire: function() {
    this.play('fire',1);
  },
  step: function(dt) {
    var p = this.p;
    if(p.animation != 'fire') {
      if(Q.inputs['right']) {
        this.play('run_right');
        p.x += p.speed * dt;
      } else if(Q.inputs['left']) {
        this.play('run_left');
        p.x -= p.speed * dt;
      } else {
        this.play('stand');
      }
    }
    this._super(dt);
  }
});

Q.Block = Q.Sprite.extend({
  init:function(props) {
    this._super(_(props).extend({ sheet: 'woodbox' }));
  }
});

Q.scene('level',new Q.Scene(function(stage) {
  stage.insert(new Q.Player({ x:100, y:50, z:2 }));
```

```
    stage.insert(new Q.Block({ x:800, y:160, z:1 }));
    stage.insert(new Q.Block({ x:550, y:160, z:1 }));
  }, { sort: true }));

  Q.load(['sprites.png','sprites.json',,'background-floor.png',
          'background-wall.png'],function() {
    Q.compileSheets('sprites.png','sprites.json');
    Q.animations('player', {
      run_right: { frames: _.range(7,-1,-1), rate: 1/10},
      run_left: { frames: _.range(0,8), rate:1/10 },
      fire: { frames: [8,9,10,8], next: 'stand', rate: 1/30 },
      stand: { frames: [8], rate: 1/5 }
    });
    Q.stageScene("level");
  });
});
```

其中的玩家类(Player)在许多方面与本章开头的玩家类例子是一样的，它定义了一个由玩家控制、基于玩家所做的动作来播放动画的 Q.Player 精灵。此外，它还在动画完成时所触发的几个事件中输出了一些日志。实际上，在真正游戏中，这些事件可用来触发子弹的发射或消耗用户精力。

障碍块类(Block)从精灵表中提取一个子项，目的是显示一个板条箱。接下来，代码定义场景“level”，作为参考，该场景设置了一个 Player 对象和两个障碍块。因为玩家应该位于其他任何东西之前，所以还需要给舞台增加一个 sort 选项。

最后，Q.load 方法加载资产、编译精灵表，然后创建动画。若希望把动画和游戏逻辑分开，或自动生成它们，可使用.json 文件来轻松加载它们(稍后会用到其中的背景图像)。

若在浏览器或移动设备中启动这一例子，你可做到让画面中的人在相对于两个障碍块的静态白色背景区域中来回走动，如图 16-1 所示。

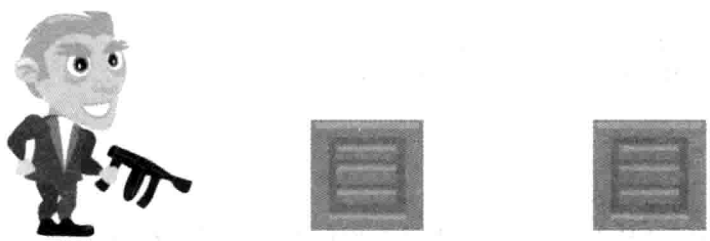


图 16-1　走动的人

16.3　添加画布视口

SVG 改善了引擎的应用，可做到轻松添加摄像头；内置的 viewport 特性使得你能控制 SVG 元素的视口，并在实践中把它当摄像头使用。

不过，借助画布标签的变形，使用一个可命名为 viewport 的组件，你可轻松把这一功能添加到舞台中。viewport 组件可使用几个方法来支持游戏的视口居中调整，以及添加要

跟踪的精灵，这意味着任何时候都要在屏幕上居中显示该精灵。

为更新视口的位置，该组件绑定了 step 事件；此外，组件还绑定了 predraw 和 draw 事件，目的是根据视口的状态把所有的渲染调用包装在适当的保存、变形和恢复中。

将代码清单 16-6 中的 viewport 组件代码添加到 quintus.anim.js 文件中，置于最后的结束花括号之前。

代码清单 16-6：viewport 组件

```
Q.register('viewport',{
  added: function() {
    this.entity.bind('predraw',this,'predraw');
    this.entity.bind('draw',this,'postdraw');
    this.x = 0,
    this.y = 0;
    this.centerX = Q.width/2;
    this.centerY = Q.height/2;
    this.scale = 1;
  },

  extend: {
    follow: function(sprite) {
      this.unbind('step',this.viewport);
      this.viewport.following = sprite;
      this.bind('step',this.viewport,'follow');
      this.viewport.follow();
    },

    unfollow: function() {
      this.unbind('step',this.viewport);
    },

    centerOn: function(x,y) {
      this.viewport.centerOn(x,y);
    }
  },

  follow: function() {
    this.centerOn(this.following.p.x + this.following.p.w/2,
                  this.following.p.y + this.following.p.h/2);
  },

  centerOn: function(x,y) {
    this.centerX = x;
    this.centerY = y;
    this.x = this.centerX - Q.width / 2 / this.scale;
    this.y = this.centerY - Q.height / 2 / this.scale;
  },

  predraw: function() {
```

```
    Q.ctx.save();
    Q.ctx.translate(Q.width/2,Q.height/2);
    Q.ctx.scale(this.scale,this.scale);
    Q.ctx.translate(-this.centerX, -this.centerY);
  },

  postdraw: function() {
    Q.ctx.restore();
  }
});
```

除了设置所需的视口参数外，开头的 added 方法并没有做太多事情。该方法设置了初始的 centerX 和 centerY 位置、一个用来控制游戏应如何放大精灵的缩放倍数和一些事件，以及用来确定窗口左上角位置的 x 和 y 坐标值。

接着，代码使用三个方法来扩展舞台，它们分别是 follow、unfollow 和 centerOn。这些方法使得开发者能够要求视图跟踪某个特定精灵的位置、解除对该精灵的跟踪，以及能够手动把视口对准某个特定的像素位置。follow 和 unfollow 简单绑定和解除绑定一个事件处理程序，该程序在每次步进时调用组件的 centerOn 方法。此外，实体的 centerOn 方法也仅是组件方法的一个代理。

接下来是组件的 centerOn 方法，该方法设置 centerX 和 centerY，然后通过中心点、画布的宽度和缩放倍数来计算 x 和 y 位置。

predraw 方法完成了所有设置视口变形的工作，它保存当前变形，在窗口中居中上下文，完成所有必要的缩放调整，然后以所想要的中心点的值为距离，移动窗口的左上角。

postdraw 方法通过简单调用上下文的 restore 方法来撤消所有变形。

要测试这一方法，修改 animation.js 中 Q.scene 部分的代码，修改后的内容如以下突出部分所示：

```
Q.scene('level',new Q.Scene(function(stage) {
  var player = stage.insert(new Q.Player({ x:100, y:50, z:2 }));
  stage.insert(new Q.Block({ x:800, y:160, z:1 }));
  stage.insert(new Q.Block({ x:550, y:160, z:1 }));

  stage.add('viewport');
  stage.follow(player);
  Q.input.bind('action',stage,function() {
    stage.viewport.scale = stage.viewport.scale == 1 ? 0.5 : 1;
  });
}, { sort: true }));
```

修改后的代码把一个视口添加到了舞台中，并把它的跟踪对象设置成玩家；然后，代码添加一个事件处理程序，允许用户通过按压动作按键来使用不同缩放比例玩游戏(在移动设备上是 b 按钮)。

16.4 实现视差效果

“视差滚动”(parallax scolling)是一种用来在 2D 卷轴游戏中赋予画面深度的技术，实现方法是让不同的背景层以不同速度滚动。例如，若让天空层以比山脉层更慢的速度进行滚动，那么这一做法能从简单层面赋予画面一种天空远离群山的效果。

为把这一功能放入引擎中，引擎需要增加一个名为 Repeater 的新精灵。该精灵与刚定义的 viewport 组件共同支持一些外加的背景元素。它的工作原理是，在 x 和 y 方向上，或只在一个方向上重复自身，且保持在屏幕上的位置的对齐。对于那些只在单个方向上重复背景的横向卷轴或纵向卷轴游戏来说，一个方向的重复是很有用的。

将代码清单 16-7 中的 Repeater 精灵添加到 quintus.anim.js 的末尾处。

代码清单 16-7：Repeater 精灵

```
Q.Repeater = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props).defaults({
      speedX: 1,
      speedY: 1,
      repeatY: true,
      repeatX: true
    }));
    this.p.repeatW = this.p.repeatW || this.p.w;
    this.p.repeatH = this.p.repeatH || this.p.h;
  },

  draw: function(ctx) {
    var p = this.p,
        asset = this.asset(),
        sheet = this.sheet(),
        scale = this.parent.viewport.scale,
        viewX = this.parent.viewport.x,
        viewY = this.parent.viewport.y,
        offsetX = p.x + viewX * this.p.speedX,
        offsetY = p.y + viewY * this.p.speedY,
        curX, curY, startX;
    if(p.repeatX) {
      curX = Math.floor(-offsetX % p.repeatW);
      if(curX > 0) { curX -= p.repeatW; }
    } else {
      curX = p.x - viewX;
    }
    if(p.repeatY) {
      curY = Math.floor(-offsetY % p.repeatH);
      if(curY > 0) { curY -= p.repeatH; }
    } else {
      curY = p.y - viewY;
```

```
      }
      startX = curX;
      while(curY < Q.height / scale) {
        curX = startX;
        while(curX < Q.width / scale) {
          if(sheet) {
            sheet.draw(ctx,curX + viewX, curY + viewY,p.frame);
          } else {
            ctx.drawImage(asset,curX + viewX, curY + viewY);
          }
          curX += p.repeatW;
          if(!p.repeatX) { break; }
        }
        curY += p.repeatH;
        if(!p.repeatY) { break; }
      }
    }
  });
```

同往常一样，init 方法仅设置一些初始的默认值，它还默认把重复的宽度和高度设成图像或资产的尺寸，这样在默认情况下，区块就能做到完美重复。

draw 方法较为复杂，它需要计算每一重复区块的偏移量，上述代码采用了简单的做法。该类没有计算用来绘制每个角落的部分图像的确切尺寸，而在必要时重新绘制区块。所以，关于边角的部分区块的渲染处理，这里把这些边缘情况的优化代码留作一道习题，由你来完成。

一些复杂度的增加是因为，背景有可能只需在垂直或水平方向重复，若元素未被设置成在某个方向上重复，那么做法就不是使用取模运算符来计算偏移量，而是把区块的位置设成精灵的 x 或 y 位置减去视图的位置。

另一方面，若区块在某个方向上重复，那么首先得使用精灵的位置计算偏移量，且要把视图的位置乘以滚动速度。

最后，绘制循环遍历每个方向，从画布左侧之前开始直到画布右侧之后，以及从画布顶部之上开始直到画布底部之下，若 Repeater 对象在任一方向上的重复已被关闭，那么相应的循环在走完一次之后就退出。

要使用 Repeater，把两个滚动背景添加到 level 场景中：

```
Q.scene('level',new Q.Scene(function(stage) {
    stage.insert(new Q.Repeater({ asset: 'background-wall.png',
                                  speedX: 0.50, repeatY: false, y:-225 }));
    stage.insert(new Q.Repeater({ asset: 'background-floor.png',
                                  speedX: 1.0, repeatY: false,  y:260}));
    var player = stage.insert(new Q.Player({ x:100, y:50, z:2 }));
    stage.insert(new Q.Block({ x:800, y:160, z:1 }));
    stage.insert(new Q.Block({ x:550, y:160, z:1 }));
    stage.add('viewport');
    stage.follow(player);
```

```
    Q.input.bind('action',stage,function() {
      stage.viewport.scale = stage.viewport.scale == 1 ? 0.25 : 1;
    });
  }, { sort: true }));
```

最终效果看起来应与图 16-2 所示的画面类似，具体取决于设备或浏览器的尺寸。

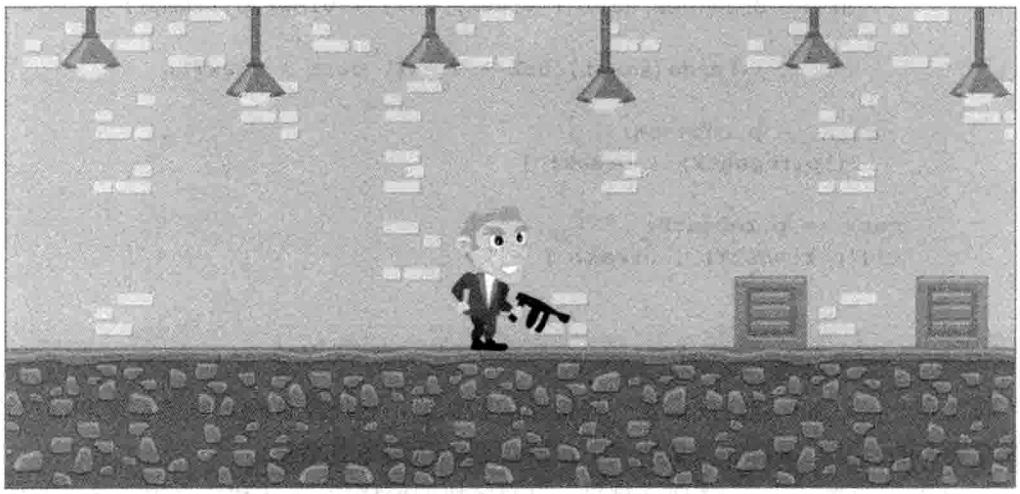


图 16-2 最终效果

16.5 小结

本章构建了一个简单动画系统，该系统允许你使用已安排好时间的命名动画来控制角色的动画。把在给定时间要播放角色的哪一帧之类的细节抽离，这样更便于你给角色添加较复杂的行为。此外，作为第 18 章的平台动作游戏的一个准备，本章还讲解了跟踪玩家的动画摄像头的添加以及对滚动视差背景的支持。

第17章

运用像素

本章提要

- 回顾 2D 物理学
- 从图像和画布中读取像素数据
- 将像素写回画布

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 17 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

17.1 引言

新 HTML5 画布标签的一个被大肆宣扬的功能是直接访问像素数据的能力，到目前为止，你还没有机会体验像素的一些玩法，不过在本章中，你会了解到一些可用来直接审查和操纵像素的手段。作为使用像素数据的一个实际应用，你将构建一个“登陆者”(Lander)风格的游戏，该游戏涉及借助推进器一阵阵小的爆发在地图上飞动飞船(参见图 17-1)。不过在此之前，本章先稍稍偏离这一主题，简单回顾一下 2D 物理学，由此获取一些构建 Lander 游戏的基础知识。

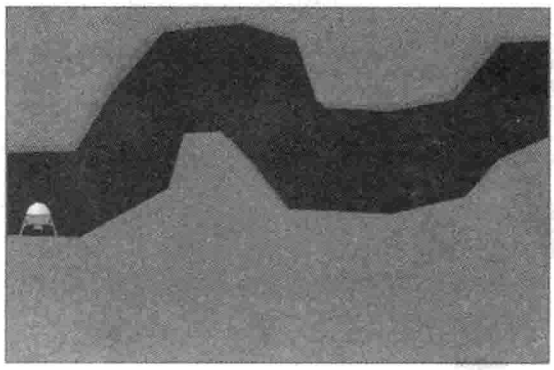


图 17-1　最终的 Lander 游戏画面

17.2　回顾 2D 物理学

在第 14 章中，你用到了一些物理学知识，当时是使用 Box2D 的 JavafScript 移植版来实现物理过程和碰撞检测，只不过，物理库为你处理了所要实现事情的细节，你并没有真正深入研究个中原理。在想要一个全面的物理模拟而又不必关心一些细枝末节时，这样做是没问题的，但一般情况下，你只会用到一些基本的 2D 运动，所以，一个完整的物理库就有些多余了。就 Lander 游戏而言，你想要的是精确到像素级别的碰撞检测，这样的碰撞检测会给传统的物理引擎带来很大的压力，因为从理论上讲，该引擎中的每个像素都必须被建模成一个模拟对象。

高等数学

警告：本节讨论一些高中水平的代数和基础微积分。若已隔了一段时间，那么这应该是一个温故知新的好机会；若看上去有些难，努力去读懂它。游戏编程往往涉及大量数学，越是先进的技术就需要用到越复杂的东西，所以从长远看，能让自己做到顺利读懂那些时不时会遇到的公式是很值得的。

17.2.1　了解力、质量和加速度

对于那些学过高中物理的读者来说，以下公式应不陌生：

公式 1：$f=m\times a$ 或 $a=f/m$

公式 2：$v=v_0+a\times dt$

公式 3：$p=p_0+v\times dt$

这些是基本的二维动力学方程，这些方程定义了物体的加速度是物体质量和施加在其之上的力的一个函数，以及刚体的位置与它的起始位置、速度和加速度这三者的组合关系。

公式 1 说明了物体的加速度等于作用在物体上的力除以物体质量，公式 2 计算当前速度 v，前提是已知初始速度 v_0 和通过公式 1 计算出来的加速度，符号 *dt* 代表时间的一个瞬

间增量。最后，公式 3 根据初始位置和通过公式 2 计算得出的当前速度计算物体的位置。

假设力和由此得出的加速度是一个常量，借助于一点微积分知识，你能把后面两个公式合成一个公式：

$$公式\ 4：p = p_0 + v_0 \times t + ½ \times a \times t^2$$

这一公式告诉你，拥有恒定加速度的物体的位置可由该物体的初始位置 P_0、初始速度 v_0 和恒定加速度 a 的一个函数来确定。对于任意被代入该公式的以秒为单位的 t 值，你都可以计算出相应的位置。有没有什么关于恒定加速度的好例子呢？嗯，重力就是一个，重力可被建模成一个值为 9.8 m/s ^{2} 的恒定作用力。

17.2.2　为炮弹建模

鉴于上述公式，可以轻松为被发射到空中的炮弹建模，因为它的垂直加速度仅受重力(一个常量)制约，而它的水平加速度为 0(这刚好也是一个常量)。因为还没有创建任何类型的矢量类，所以处理 2D 位置的最简单做法就是每一帧都计算该公式两次，一次用于 x 方向，一次用于 y 方向。

就 x 方向而言，可以通过完全去掉 x 的加速度部分来进一步简化公式，不过出于完整性考虑，这部分还是被保留了下来。

代码清单 17-1 采用了上述公式，并使用 Quintus 来运行一个把炮弹发射到空中的简单模拟。可修改任何初始值，看看它们是如何影响行为的。

代码清单 17-1：使用闭合形式解为炮弹建模

```
var Q = Quintus().include("Sprites").setup()

Q.load(['cannonball.png','cannonball2.png'],function() {

  var ball1 = new Q.Sprite({
    asset: 'cannonball.png',
    x0:    0,   // Initial X position
    vx0:  20,   // X velocity
    ax:    0,   // X acceleration
    y0:  380,   // Initial Y position
    vy0:-100,   // Y Velocity
    ay:   20,   // Constant Y acceleration
    t:     0    // Starting time
  });

  ball1.step = function(dt) {
    var p = this.p;
    p.t += dt;

    p.x = p.x0 + p.vx0 * p.t + 0.5 * p.ax * (p.t * p.t);
    p.y = p.y0 + p.vy0 * p.t + 0.5 * p.ay * (p.t * p.t);
  }
```

```
    Q.gameLoop(function(dt) {
      Q.clear();

      ball1.step(dt);
      ball1.draw(Q.ctx);
    });

});
```

其中很大一部分代码的工作是设置炮弹的位置和速度的初始值，实际位置的计算则只用到了更新函数末尾处的两行代码，这两行代码完全照搬公式 4，一个用于 x 方向，一个用于 y 方向。最后显式调用 gameLoop 函数，该函数先清除画布，然后更新炮弹的位置并调用绘制方法。若希望显式跟踪炮弹的路径，可以删除对 this.clear()的调用，这样炮弹就会留下它的运动轨迹。

17.2.3　换成迭代解

公式 4 是一个闭合形式解，只要在这一个公式中代入 t，你就可以求出炮弹在任一时间点上的确切位置。之所以能做到这一点是因为，你已模型化了这样的一个简单物体。一旦遇上较复杂的情况，比如说物体涉及与其他物体的交互或涉及用户的输入，那么不从 MIT(麻省理工学院)雇几个数学家来，估计你是找不出一个闭合形式解了。

相反，需要找回最初那组微分公式(公式 1 到公式 3)，然后把它们变成一些比较计算机友好的东西。实现这一点的最简单做法是使用一种“离散集成技术”，离散集成意味着作为使用微积分来确定一个准确解的替代，可以使用一个小的真正 *dt* 值来求出近似解。最常用的离散集成方法刚好也是最简单的方法，被称为向前欧拉法(Forward Euler)，以著名的瑞士数学家 Leonhard Euler 的名字命名。这一方法可被很好地转换成计算机模拟，这意味着可以假设在任意一小段时间中，任何可能会发生改变的值(如加速度或速度等)都是恒定的。速度和位置公式现在变成了以下这个样子：

$$\text{公式 5：} v = v_{t-1} + a_t \times \mathrm{d}t$$

$$\text{公式 6：} p = p_{t-1} + v_t \times \mathrm{d}t$$

在上述公式中，*dt* 不再是一个瞬间增量，而是一段很短暂但可以衡量的时间。在游戏中，它大约为 1/30 秒，也即动画的一帧。

这些公式不再使用 v_0 和 p_0(时间为零时的速度和位置)这两个符号，而代以 v_{t-1} 和 p_{t-1}(上一次模拟步进时的速度和位置)，这意味着仅是代入 *t* 的一个值已不能获得炮弹在任意给定时间点的位置，相反，你采用的是增量式计算，只能基于前一个位置计算出当前位置。

对于大部分情况来说，这一局限性是有好处的，因为通常你只关心游戏的当前状态，不过，出于回放或时间旅行的目的(比如在 *Braid* 之类的游戏中)，记录之前发生的事情也会是计划之一，所以，你必须多做一些额外的工作，在某处记录下游戏的历史。

现在，是时候通过同时运行闭合形式解的例子和迭代解的例子来比较一下这两者了。创建第二个名为 ball2 的 Sprite 对象，使用前面介绍的向前欧拉法来更新其位置。将代码清单 17-2 中所示的 ball2 代码添加到 ball1 的 step 方法的后面。

代码清单 17-2：使用迭代解为炮弹建模

```
var ball2 = new Q.Sprite({
  asset: 'cannonball2.png',
  x:    0,
  vx:   20,
  ax:    0,
  y:   380,
  vy: -100,
  ay:   20
});

ball2.step = function(dt) {
  var p = this.p;

  p.vy += p.ay * dt;

  p.x += p.vx * dt;
  p.y += p.vy * dt;
}
```

为让第二个炮弹出现在画面上，需要更新 gameLoop，以对两个炮弹都进行更新，如下所示：

```
Q.gameLoop(function(dt) {
    this.clear();

    ball1.step(dt);
    ball1.draw(this.ctx);

    ball2.step(dt);
    ball2.draw(this.ctx);
});
```

因为第二个模拟使用了运动公式的一个近似解，所以预计你会看到一些错误缓慢发生，这表现为两条运动轨迹的偏离，不过，它所产生的实际效果与闭合形式解并没有明显差别。这是一个好现象，因为这意味着所使用的近似值并未对真正解的行为改变太多。

17.2.4　抽取可重用类

现在是时候把在第 11 章中为 Quintus 创建的 Sprite 对象拿出来，使用一个已修改的 step 方法来扩展它了，这样做是为了抽离你的运动方法。你唯一需要做的修改是更新 init 方法，

把速度和加速度初始化为零，然后把新对象的更新代码复制到新对象中(见代码清单 17-3)。把 MovingSprite 类添加到 quintus_sprites.js 末尾处，置于最后的结束花括号之前。

代码清单 17-3：Quintus 的 MovingSprite 类

```
Q.MovingSprite = Q.Sprite.extend({
  init: function(props) {
    this._super(_({
      vx: 0,
      vy: 0,
      ax: 0,
      ay: 0
    }).extend(props));
  },

  step: function(dt) {
    var p = this.p;

    p.vx += p.ax * dt;
    p.vy += p.ay * dt;

    p.x += p.vx * dt;
    p.y += p.vy * dt;

    this._super(dt);
  }
});
```

MovingSprite 类把初始速度和加速度添加到了基本属性中，然后修改 step 方法，让它运行一个迭代解。

17.3 实现 Lander 游戏

现在，好好来利用一下新创建的前向欧拉精灵类，构建一个基于物理学的简单游戏，在该游戏中，你驾驶一台月球登陆器在洞穴中探险。为什么这会是物理细节的一个很好利用呢？嗯，与其他游戏不同，一般来说，在这些游戏中，你会直接控制游戏主角的速度，但在登陆者风格的游戏中，你只控制加速度，这意味着移动需要加以提前计划。可实施三种控制：左推动、右推动和上推动。需要很小心地安排登陆器的攀升，因为若用了太大的上升推力，那么你就只能眼睁睁地看着自己的登陆器撞向岩洞顶部了。

17.3.1 自建游戏

创建一个名为 lander.html 的新文件，将代码清单 17-4 中的代码添加到其中，创建游戏的基本框架并设置游戏。

代码清单 17-4：lander.html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Simple Cannon example</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>

    <meta name="viewport" content="width=device-width,user-scalable=no">
    <style>
      body { padding:0px; margin:0px; }
      #quintus { background-color:#CCC; }
    </style>
  </head>
  <body>
    <canvas id='quintus' width='480' height='320'></canvas>

    <script>
      var Q = Quintus().include("Input,Sprites,Scenes")
                       .setup()
                       .controls();


    </script>
  </body>
</html>
```

因为这一登陆者游戏不会用到太多代码，所以整个游戏都在 lander.html 这一个文件中构建。

17.3.2　构建飞船

现在创建一个通过键盘控制且只使用推动力控制的飞船精灵基类，扩展 MovingSprite 类，加入一个新的 step 方法，该方法接收用户输入并算出新位置。

为了赋予游戏一点活力，对飞船进行一些设置，让飞船在被施加了垂直推力时显示一个推进器图像(见代码清单 17-5)，此外，还要添加一些约束来防止登陆器飞离屏幕，若登陆器飞出了画布的边界之外，就让它停下来。

把这部分代码添加到 lander.html 中，置于</script>这一结束标签之前。

代码清单 17-5：飞船基类

```
Q.Ship = Q.MovingSprite.extend({
  step: function(dt) {
    var p = this.p;

    // Set our horizontal force
    p.fx = 0;
    if(Q.inputs['left']) { p.fx -= p.thrustX; }
    if(Q.inputs['right']) { p.fx += p.thrustX; }

    // Set our vertical force
    if(Q.inputs['fire']) {
      p.fy = -p.thrustY;
      p.asset = "lander_thrust.png";
      } else {
      p.fy = 0;
      p.asset = "lander.png";
    }

    // Calculate our y and x acceleration
    p.ay = p.gravity + p.fy / p.m;
    p.ax = p.fx / p.m;

    // Let our super-class update our x and y
    this._super(dt);

    // Force our lander to stay in our box
    // and zero out our velocity when we hit a wall
    if(p.y < 0) { p.y = 0;  p.vy = 0; }
    if(p.y > Q.height- p.h) { p.y = Q.height - p.h; p.vy = 0; }
    if(p.x < 0) { p.x = 0; p.vx = 0; }
    if(p.x > Q.width - p.w) { p.x = Q.width - p.w; p.vx = 0; }
  }
});
```

Ship 类扩展上一节中的 MovingSprite 类，重写 step 方法，支持玩家输入以及其他一些行为。该更新方法包含了几部分不同的内容，第一部分通过查看玩家输入来计算作用在飞船上的力，若按下的是右或左按键则添加一个水平方向的力，若按下的是空格键则添加一个垂直方向的力。此外，该类还会更换精灵的资产，在飞船加速向上时显示 lander_thrust.png 图形。接着，方法通过这些作用力、一个重力常量和飞船的质量属性来计算当前加速度，在更新加速度后，方法调用父类方法来更新速度和位置。最后，方法执行必要的边界检查，确保飞船停留在屏幕上。

接下来把 Ship 类真正加入到游戏中，先是例行公事的标准加载和舞台设置细节，之后创建一个背景精灵和一个新的飞船对象 ship，并初始化对象的 x 和 y 位置，以及它的质量

和重力。将代码清单 17-6 中的代码添加到 lander.html 的末尾处，置于结束标签</script>之前。

代码清单 17-6：Lander 游戏的基本代码

```
Q.load(['lander.png','background.png',
        'lander_thrust.png','map.png'], function() {

  Q.scene("level",new Q.Scene(function(stage) {
    stage.insert(new Q.Sprite({ asset: "background.png" }));
    stage.insert(new Q.Sprite({ asset: "map.png" }));

    var ship = stage.insert(new Q.Ship({
      asset: 'lander.png',
      x:       10,  // X Position
      y:       170, // Y Position
      gravity: 20,  // Gravity
      m:       1,   // Ship's Mass
      thrustX: 40,  // Horizontal Thrust
      thrustY: 80,  // Vertical Thrust
    }));

  }));

  Q.stageScene("level");
});
```

启动代码，或运行本章代码中的 lander_basice.html 例子，然后使用左按键、右按键和空格键来移动飞船。可修改这一简单登陆器的质量和重力选项，感受一下对这些变量的调整会如何影响飞船的移动。

尽管背景已被加载进来，但飞船尚未与其交互，仅是从上面飞过，这一问题将在下一节得以纠正。

17.3.3　精确到像素级

在实现简单飞船在屏幕上的漂移之后，现在可以开始把精力放在最大的一个尚未考虑的要素上了，这就是洞穴的岩壁。

虽然可按前面第 14 章的 SVG 练习中的做法来构建游戏，只实现对象到对象的碰撞检测，不过，利用画布标签访问位图的像素数据的能力，实现登陆器位图和关卡位图之间的像素到像素的碰撞检测，这样的做法则显得更合理一些。

可直接查看用来绘制整个游戏的 canvas 标签，但随着画布被绘上多层内容和飞船，在这之后你只有依靠一些技巧(比如说，通过查找某种特有的颜色)，才能分辨出哪部分是岩壁哪部分是飞船了。为将游戏变得更灵活一些，可以把背景重绘到一个离屏画布中，然后从中提取出像素数据。为促成这一灵活性，可将一个新方法添加到 Quintus 的核心代码中，该方法返回某个已被加载图像的像素数据。

这里利用了 getImageData 方法(见 W3C 规范 www.w3.org/TR/2dcontext/#pixel-manipulation)，该方法返回画布对象的图像数据。

运行例子的那些麻烦事

就通过本地文件系统(也即通过以 file://开头的 URL)访问像素数据而言，不同的浏览器存在着不同的限制。若在运行例子时出现了问题(可通过查看 JavaScript 控制台找出错误)，那么就像之前为那些加载了 JSON 数据的例子所做的那样，通过本地服务器来运行它们。

若拥有一个图像而非一个画布对象，以背景图像为例，那么可以使用 jQuery 来创建一个新的画布元素，然后把该背景图像绘制到这一画布元素上。然后，如代码清单 17-7 所示，可通过调用 getImageData 并传入所需部分图像的坐标(如整个画布)从画布上下文中返回图像的像素数据。

代码清单 17-7：返回图像的像素数据

```
Q.imageData = function(img) {
  var canvas = $("<canvas>").attr({
                    width: img.width,
                    height: img.height })[0];

  var ctx = canvas.getContext("2d");
  ctx.drawImage(img,0,0);

  return ctx.getImageData(0,0,img.width,img.height);
}
```

检索图像数据某种程度上是一种代价高昂的操作，所以，除非是在更新图像，否则最好只这样做一次，然后缓存结果。

将代码清单 17-7 中的代码添加到 quintus.js 的末尾处，置于最后的 return Q 语句之前。

17.3.4 运用 ImageData 对象

现在你已经有了一个 ImageData 对象，如何来确定某个具体的 x 和 y 位置上的像素是“可碰撞”的呢？嗯，这一任务相当于找出某个具体像素的颜色值，若要实现这一点，需要用到一些简单的数学知识。ImageData 对象拥有宽度和高度特性，不过信息的主体部分则位于 ImageData.data 中，这是一个存放了真正的像素数据的一维数组，数据格式为 4 字节的 RGBA(红色、绿色、蓝色和透明度)数据块，这意味着该数组的每四个元素一起构成画布的 1 个像素。在这四个元素中，每个元素都代表了一个 0～255 之间的数字，可以检查组成该像素的红色、绿色、蓝色和透明度的值。

如图 17-2 所示，透明度值确定了像素的不透明度，若值为 0 则意味着像素是完全透明的，而值为 255 则表示像素是完全不透明的。可使用透明度值来确定图像中的该特定像素位置上是否有东西存在。

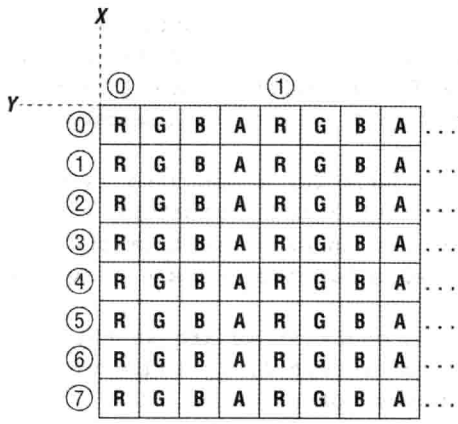


图 17-2　透明度值和透明度

为获取图像的某个具体像素的数据，需要确定该像素从数组开始处算起的元素编号。因为每个像素由四个元素组成，所以数据每行的长度是图像像素宽度的四倍。为了得到行偏移量，需要把每行的长度乘以 y 位置，为了索引到元素在行中的正确编号上，使用 x 的值并把它乘以 4。接下来，因为你想要的是该像素的透明度值，所以再给结果加上 3，这一计算过程最终体现为以下代码：

```
alpha = imageData.data[y*4*imageData.width + x*4 + 3];
```

明白了吗？很好，因为你实际上需要在一个二维循环的每次迭代中做两次这样的计算，你将遍历小登陆器的所有像素，把飞船的每个非透明像素与同一位置的背景像素做比较。

为让内循环变得稍微简单一点，可预先计算出 Ship 对象相对于背景像素数据的起始位置。代码实现了这一点，如代码清单 17-8 所示，并把计算结果存放在变量 bgOffset 中。把下面的 checkCollision 方法添加到 Ship 类中。

代码清单 17-8：检查飞船和背景之间的碰撞

```
checkCollision: function() {
  var bgData = Q.backgroundPixels;

  // Get a integer based position from our
  // x and y values
  var bgx = Math.floor(this.p.x);
  var bgy = Math.floor(this.p.y);

  // Calculate the initial offset into our background
  var bgOffset = bgx * 4 + bgy * bgData.width * 4 + 3;

  // Pull out our data easy access
  var pixels = this.imageData.data;
  var bgPixels = bgData.data;
```

```
for(var sy=0;sy < this.imageData.height;sy++) {
  for(var sx=0;sx < this.imageData.width;sx++) {
    // Check for an existing pixel on our ship
    if(pixels[sx*4 + sy * this.imageData.width * 4 + 3]) {

      // Then check for a matching existing pixel
      // on the background starting from our bgOffest
      // and then indexing in from there
      if(bgPixels[bgOffset + sx*4 + sy * bgData.width * 4]) {

        // Next check if we are at the bottom of the lander
        // if so return 1, to indicate that we might be landing
        // instead of crashing
        if(sy > this.imageData.height - 2) {
          return 1;
          } else {
          // Otherwise return 2 and...Boom!
          return 2;
        }
      }
    }
  }
}
return 0;
}
```

碰撞代码通过检查碰撞是否发生在对象 ship 的底部来尝试区分“着陆”和“撞毁”两种情况，若是，方法返回 1；否则即碰撞发生在对象的其他地方，方法返回 2，指明此时该把飞船炸毁。

现在，需要更新 Ship 类，在 step 函数中检查碰撞。把以下代码添加到 step 方法的起始处，在飞船已毁时停止飞船的移动：

```
step: function(dt) {
  if ( this.dead ) return;
```

接着，在这同一函数的末尾处，加入对 checkCollision 方法的一个调用，并适当处理响应：

```
this._super(dt);

var col;
if(col = this.checkCollision()) {
  if(col == 1 && Math.abs(p.vy) < 30) {
    if(p.vy > 0) {
      p.vy = 0;
    }
    } else {
    this.dead=true;
  }
}
```

需要验证 checkCollision 的返回值，若返回 1，那说明撞到的是登陆器的底部，若移动速度缓慢，那么可以着陆，否则就把 Ship 对象标记为已毁。

接下来，需要修改游戏本身的代码，提取出飞船要与其进行比较的 backgroundPixels。如下所示，把突出显示的代码行添加到 lander.html 的场景定义中：

```
Q.load(['lander.png','background.png',
        'lander_thrust.png','map.png'], function() {

  Q.scene("level",new Q.Scene(function(stage) {
    stage.insert(new Q.Sprite({ asset: "background.png" }));
    stage.insert(new Q.Sprite({ asset: "map.png" }));

    var ship = stage.insert(new Q.Ship({
      asset: 'lander.png',
      x:       10,       // X Position
      y:       230,      // Y Position
      gravity: 20,       // Gravity
      m:       1,        // Ship's Mass
      thrustX: 40,       // Horizontal Thrust
      thrustY: 80,       // Vertical Thrust
    }));

    Q.backgroundPixels = Q.imageData(Q.asset('map.png'));
    ship.imageData = Q.imageData(Q.asset('lander.png'));

  }));

  Q.stageScene("level");
});
```

祝贺你！现在你已经实现了背景和飞船之间的精确到像素级的碰撞，所以，创建新关卡变得很容易，只需创建一个新的关卡图像即可。若希望看看实现的效果，可运行本章代码中的 lander_collision.html 文件。在飞船撞毁后，需要按下浏览器的 Reload 按钮来重新启动游戏。

虽然代码清单 17-7 中的代码是可用的，但有待优化。若这是一个游戏产品，那么你应该深入研究一下，找出一种最好做法来优化 checkCollision 代码的内循环部分。一种临时的优化做法是修改循环变量，每次递增一个数值而非 1，以免在查找像素数据时需要执行乘法运算(相比于加法，执行乘法耗费的 CPU 时间更多)。这里把这一优化留作习题，你可试着自己来解答。

getImageData 的限制

在通过 getImageData 使用画布时，有几个与此相关的限制是你应该知道的。若把来自另一个域的图像放入到画布中，那么画布就成为了“被污染”的，你会因此而不能使用该

画布的 getImageData 方法。设置这样的一个限制是为了防止访问可能包含了个人信息的其他网站图像，因为图像的加载用到了被发送到该域的所有 cookie。

17.3.5 制造爆炸

现在，在 Lander 的当前版本中，你还没有为飞船的毁灭制造一种生动效果——飞船仅是停止不动而已。可借助一些爆炸像素来加以改进，创建一个新类，该类接收登陆器的像素，然后在登陆器被摧毁时炸开这些像素，这相当于实现了一个简化版的基于像素的粒子引擎。

1. 添加 Explosion 类

首先，需要创建一个新的名为 Explosion 的类，虽然它的行为与你所熟悉的精灵相似，因为它拥有一个 step 和一个 draw 方法，不过可以让它直接继承自基类，因为你不需要任何现有的功能。

init 方法将接收一些信息，这些信息是关于将被粒子化的登陆器的——具体为它的位置、速度和图像数据——然后创建一系列粒子，每个粒子有各自的位置和速度，代表所输入的登陆器图像的一个像素。出于性能原因，你不会把登陆器的每个像素都转换成粒子，相反，你仅需每隔三个像素就取出一个加以粒子化，创建一些 3x3 像素大小的粒子。这是一个很随意的决定——你可通过修改 init 代码来生成不同大小的粒子，感受一下性能随之发生的变化。代码清单 17-9 给出的是添加了注释的 init 代码。

代码清单 17-9：Explosion 类的 init 方法

```
Q.Explosion = Q.GameObject.extend({
  init: function(x,y,vx,vy,imgData) {

    // Set up a container for our pixels
    this.particles = []

    // Grab the lander's image data
    var landerData = imgData.data;


    // Create a 3x3 pixel-data
    // image data container to use for blitting down the road
    this.pixelData = Q.ctx.createImageData(3,3);
    this.drawPixel = this.pixelData.data;

    // Pixels are going to be exploding out from
    // the center of the lander
    var centerX = imgData.width / 2;
    var centerY = imgData.height / 2;

    // Loop over each fourth pixel of the lander image
    for(var sy=0;sy < imgData.height;sy+=4) {
```

```
    for(var sx=0;sx < imgData.width;sx+=4) {

      // Offset into the 1 dimension pixel data array
      var loc = sx*4 + sy * imgData.width * 4;

      // If there's a lander pixel here
      if(landerData[loc + 3]) {

        // Get the direction of the pixel from center
        var distX = sx - centerX;
        var distY = sy - centerY;

        // Add a new particle
        this.particles.push({
          x: x + sx,  // starting position x
          y: y + sy,  // starting position y
          lifetime: 5,  // remaining lifetime
          r: landerData[loc] + 20,  // make it a little redder
          g: landerData[loc+1],
          b: landerData[loc+2],
          a: landerData[loc+3],
          // For particle velocity, use the ship's
          // velocity, plus a random direction emanating
          // from the center of the ship
          vx: vx/6 +  distX * 5 *(Math.random()+0.5),
          vy: vy/6 + distY * 5 * (Math.random()+0.5)
        });
      }
    }
  }
},
```

在 init 方法运行后，Explosion 对象就拥有了一组与飞船像素的原始颜色相似的像素，这些像素可向外散开并可独立移动。

2. 绘制像素

创建粒子后，需要实现余下的两个精灵函数：step 和 draw(见代码清单 17-10)，这样才算是完成了 Q.Explosion 类的编写。对于 step 来说，该函数需要给爆炸粒子加上重力的影响，然后使用前向欧拉法来步进每个粒子。draw 函数所做的事情会更令人感兴趣一些，在上述的初始化函数中，你创建了一个 3x3 像素的 imageData 对象供 draw 函数使用。需要使用每个粒子的颜色来填充这一 9 像素的 imageData 对象，然后使用画布的 putImageData 方法(见 www.w3.org/TR/2dcontext/#pixel-manipulation)将其绘制到画布上。

代码清单 17-10：更新和绘制爆炸粒子

```
step: function(dt) {
 for(var i =0,len=this.particles.length;i<len;i++) {
```

```
    var v = this.particles[i];
    if(v.lifetime > 0) {
      v.vy += 20 * dt;
      v.x += dt * v.vx;
      v.y += dt * v.vy;
      v.lifetime -= dt
    }
    if(v.lifetime <= 0) { Q.stageScene('level'); return; }

  }

},
draw: function(ctx) {
  for(var i=0,len=this.particles.length;i<len;i++) {
    var v = this.particles[i];

    if(v.lifetime > 0) {
      for(var l=0;l<36;l+=4) {
        this.drawPixel[l+0] = v.r;
        this.drawPixel[l+1] = v.g;
        this.drawPixel[l+2] = v.b;
        this.drawPixel[l+3] = v.a;
      }

      ctx.putImageData(this.pixelData,v.x,v.y);
    }
  }

}
});
```

为填充这一 9 像素的 imageData 对象，需要遍历其中所有 9 个像素。因为 imageData.data 是一个一维数组，所以可使用单个循环把数据复制到这一数组中。可优化绘制函数中的这一循环，做法是把变量 i 的递增值设为 4 而非 1，每次访问像素数据的四个元素，避免在循环中用到乘法。

putImageData 是一个十分有趣的方法，因为它照字面实现了图像数据到画布的位块传输——换句话说，忽略任意透明度，每个像素都被位对位(bit-for-bit)地复制了过来。一般来说，这意味着这不是一个很好的合成工具，不过在这个例子中，因为你仅是绘制不透明的方块，所以它很好地贯彻了自己的服务宗旨，并带来了额外好处：速度够快。若希望直接操纵多层的像素数据，需要使用 putImageData 把数据放到一个离屏画布中，然后使用 drawImage 把图像绘制到活动缓冲区中，同时将 globalAlpha 设成一个小于 1 的数值。

实际上，调用 createImageData 是一个(相对)缓慢的过程，所以，把 3x3 方块重用于每个粒子可提升性能。

这是实现爆炸效果的唯一做法吗？绝对不是，至少还存在两种你可采用的方法。第一种可选做法是简单在页面上绘制具有正确颜色的 3 像素矩形，另一种是使用原始图像并代

以绘制图像的个别像素方块。目前讨论的这一方法仅是作为一个例子，向你展示如何使用putImageData。

为让粒子发挥作用，需要修改Ship类，在飞船炸毁时创建一个新的爆炸对象。修改飞船的step方法，如以下突出部分代码所示：

```
var col;
if(col = this.checkCollision()) {
  if(col == 1 && Math.abs(p.vy) < 30) {
    if(p.vy > 0) {
      p.vy = 0;
    }
  } else {
    this.parent.insert(
      new Q.Explosion(p.x,p.y,p.vx,p.vy,this.imageData)
    );
    this.parent.remove(this);
    this.dead=true;
  }
}
```

3. 加入粒子和岩壁的碰撞

就你的目的而言，这一Lander游戏已基本完成，要说缺少的也就是对爆炸的一个改进。虽然爆炸对象已经给出了你所想要的效果，不过它们尚未与背景交互。因为已访问过像素数据，所以你完全可以通过执行一个快速的逐一像素检查来实现像素级的粒子碰撞。如代码清单17-11所示，可使用一个比登陆器碰撞检测代码更简单的版本来更新Explosion.step方法，把这些岩壁都纳入考虑范围。为简化碰撞检测，假设每个粒子都只有1个像素大小，因为同时移动所有这些粒子会导致速度变慢，在移动设备上更是如此。

代码清单17-11：让粒子和岩壁产生交互

```
step: function(dt) {

 var bgData = Q.backgroundPixels;
 var pixels = bgData.data;

 for(var i =0,len=this.particles.length;i<len;i++) {
   var v = this.particles[i];
   if(v.lifetime > 0) {

     var oldx = v.x, oldy = v.y;
     v.vy += 20 * dt;
     v.x += dt * v.vx;
     v.y += dt * v.vy;
     var loc = Math.floor(v.x)*4 + Math.floor(v.y) * bgData.width * 4;
     if(pixels[loc + 3]) {
       v.x = oldx;
```

```
        v.y = oldy;
        v.vy *= -0.2;
        v.vx *= -0.2;
      }
      v.lifetime-=dt;
    }
    if(v.lifetime <= 0) { Q.stageScene('level'); return; }

  }

},
```

为确定在像素数组中的偏移位置(存放在 loc 变量中)，需要使用 Math.floor 方法。之所以这样做是因为粒子的位置是使用浮点数表示的，这些浮点数不一定刚好落在整数边界上。为了用来索引数组，这些数值需转换成整数，Math.floor 实现了这一点，它接收一个任意浮点数，切除所有小数部分后返回一个整数。

若运行该游戏，或运行本章代码中的 lander_explosion.html 文件，那么你现在应能看到一艘在撞毁时会炸得粉碎的飞船。

17.4 小结

画布把直接操纵像素数据变成了一种可能，虽然这是一种不常用的功能，但对于各种需要像素级别的碰撞或画布后处理(post-processing)的情况来说，该功能很有用处。不过，在使用逐一像素处理这样的做法时，用量方面需要谨慎把握，因为许多移动设备的 CPU 功率要比桌面 CPU 小得多，所以你得小心，别让它们负荷过重。

第18章

创建一个 2D 平台动作游戏

本章提要

- 创建区块层
- 优化区块渲染
- 增加 2D 平台动作游戏的碰撞检测
- 构建一个平台动作游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 18 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

18.1 引言

在成长于任天堂娱乐系统年代的开发者(以及那些像我一样能够也会经常在脑海中哼唱超级马里奥兄弟的 8 位起始和弦的人)的心中，2D 平台动作游戏(2D platformer)始终占据着特殊位置。因为拥有非常配合屏幕小键盘的简单控制，而且很容易上手，所以 2D 平台动作游戏作为一种受欢迎的移动游戏类型被保留了下来。本章构建一个简单的 2D 平台动作游戏的一些基本元素，这其中用到了第 16 章中的动画工具，并增加了一些平台动作游戏特定的区块(tile)和碰撞。

18.2 创建区块层

至今一直在用的这种简单的碰撞检测方案存在一个重大缺陷：碰撞不会随着碰撞项的增多而进行规模调整。不知你是否还记得到目前为止 Quintus 实现碰撞的做法(不包括 Box2D 的使用在内)，它把每个精灵和舞台上的其他所有精灵进行比较。

虽然对于舞台来说，这样做是没问题的，舞台中的精灵和可能发生的碰撞的数量都很有限，但对于平台动作游戏来说——此类游戏可能有着接连不断的关卡，每一关都可能存在着成千上万的精灵要与其碰撞的区块——这种做法可能很快就会变得难以进行。为了解决这一问题，引擎需要支持碰撞层的概念。确定某个精灵在任意给定位置上与哪一区块发生了交互是很简单的事情，因为区块在帧到帧之间并未发生移动，它们被放在固定的位置上。

此外，因为关卡有可能变大，你得只绘制关卡在页面上的可视部分，所以，优化要绘制的内容是区块层(tile layer)的另一项职责。

18.2.1 编写 TileLayer 类

为将对平台动作游戏的支持添加到引擎中，现增加另一个 Quintus 模块：Quintus.Platformer。该模块包括了一个用到 2d 模块的 Q.TileLayer 类，以及一个被优化来使用 Q.TileLayer 的特殊舞台。最初尚未优化的 TileLayer 很简单，它的任务是加载区块并在每一帧中绘制所有区块。

创建一个名为 quintus_platformer.js 的 JavaScript 新文件，将代码清单 18-1 的代码加入其中。

代码清单 18-1：基本的 TileLayer 类

```
Quintus.Platformer = function(Q) {

 Q.TileLayer = Q.Sprite.extend({
   init: function(props) {
     this._super(_(props).defaults({
       tileW: 32,
       tileH: 32,
       blockTileW: 10,
       blockTileH: 10,
       type: 1
     }));
     if(this.p.dataAsset) {
       this.load(this.p.dataAsset);
     }
     this.blocks = [];
     this.p.blockW = this.p.tileW * this.p.blockTileW;
     this.p.blockH = this.p.tileH * this.p.blockTileH;
     this.colBounds = {};
```

```
    this.directions = [ 'top','left','right','bottom'];
  },

  load: function(dataAsset) {
    var data = _.isString(dataAsset) ? Q.asset(dataAsset) : dataAsset;
    this.p.tiles = data;
    this.p.rows = data.length;
    this.p.cols = data[0].length;
    this.p.w = this.p.rows * this.p.tileH;
    this.p.h = this.p.cols * this.p.tileW;
  },

  setTile: function(x,y,tile) {
    var p = this.p,
        blockX = Math.floor(x/p.blockTileW),
        blockY = Math.floor(y/p.blockTileH);

    if(blockX >= 0 && blockY >= 0 &&
       blockX < this.p.cols &&
       blockY <  this.p.cols) {
      this.p.tiles[y][x] = tile;
      if(this.blocks[blockY]) {
        this.blocks[blockY][blockX] = null;
      }
    }
  },

  draw: function(ctx) {
    var p = this.p,
        tiles = p.tiles,
        sheet = this.sheet();
    for(var y=0;y < p.rows;y++) {
      if(tiles[y]) {
        for(var x =0;x < p.cols;x++) {
          if(tiles[y][x]) {
            sheet.draw(ctx,
                       x*p.tileW + p.x,
                       y*p.tileH + p.y,
                       tiles[y][x]);
          }
        }
      }
    }
  }
});
};
```

TileLayer 精灵还不用做太多事情，init 方法一如既往地先设置几个属性，若有 dataAsset 属性被传入，就调用 load 方法。此外，它还设置了几个将会在下一节中用到的属性。

setTile 方法用来在事后修改地图上某个位置的区块，其中一些代码会清除被预先渲染的组块(block)，等下一节加入预渲染之后，这部分代码的作用就便于理解了。

为了限制游戏每帧需要绘制的区块数，TileLayer 做了优化，把区块构成的组块预先渲染到离屏画布元素中。位于 init 方法开头的另一部分初始代码对此进行了设置，为本章后面“优化绘制”一节中加入的代码预先算出了每个组块的大小(以像素为单位)。

load 方法仅载入一些数组中的某个数组，这些数组定义了所有位置上的区块的精灵表的帧，在载入后，方法通过该数组计算 TileLayer 的大小。

最后，draw 方法重载默认的 Sprite 类 draw 方法，遍历区块的每一行，并在相应位置上绘制那些具有非零值的区块。

18.2.2 试用 TileLayer 代码

为测试这一代码，你需要用到标准的 HTML 自建文件。创建一个名为 platform.html 的新文件，将代码清单 18-2 的代码输入其中。

代码清单 18-2： 自建文件 platform.html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
 user-scalable=0, minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Platformer</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>
    <script src='js/quintus_anim.js'></script>
    <script src='js/quintus_platformer.js'></script>
    <script src='platform.js'></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

这是标准的 Quintus 代码，这段代码把所需的依赖和以前已写好的模块，以及你刚开始编写的新模块 quintus_platformer.js 都加了进来。

接下来，创建被加到上述文件中的 platform.js 文件，然后输入本章剩余部分将要构建的平台动作游戏的起始阶段代码。可注意到，该段代码包含了额外的一些加载和动画代码，

这些代码不会被马上用到。platform.js 的初始代码如代码清单 18-3 所示。

代码清单 18-3：平台动作游戏的初始代码

```
$(function() {
  var Q = window.Q = Quintus()
                  .include('Input,Sprites,Scenes,Anim,Platformer')
                  .setup('quintus', { maximize: true })
                  .controls();

  Q.scene('level',new Q.Scene(function(stage) {

    Q.compileSheets('sprites.png','sprites.json');

    stage.insert(new Q.Repeater({ asset: 'background-wall.png',
                                  speedX: 0.50, y:-225, z:0 }));
    var tiles = stage.insert(new Q.TileLayer({ sheet: 'block',
                                               x: -100, y: -100,
                                               tileW: 32,
                                               tileH: 32,
                                               blockTileW: 10,
                                               blockTileH: 10,
                                               dataAsset: 'level.json',
                                               z:1 }));
    stage.add('viewport');
    stage.centerOn(0,0);
    Q.input.bind('right',function() {
      stage.viewport.centerOn(stage.viewport.centerX + 64,
                              stage.viewport.centerY)
    });
    Q.input.bind('left',function() {
      stage.viewport.centerOn(stage.viewport.centerX - 64,
                              stage.viewport.centerY)
    });
  }, { sort: true }));

  Q.load(['background-wall.png','sprites.png',
          'sprites.json','level.json'],function() {

    Q.stageScene("level");
  });
});
```

这段代码看起来应与第 15 章中的代码类似，只是增加了一些动画和已加载的资产。为了让例子正常工作，你需要用到本章代码中的一些资产，务必把其中的图像文件置于 images/目录中，并把.json 文件置于 data/目录中。

在这段代码中，舞台绑定了向左和向右移动的动作，允许你移动视口，所以，你应该在桌面浏览器中使用箭头按键，或在移动设备上使用小键盘来滚动背景。

18.2.3 优化绘制

既有小的区块又有大型的关卡，每一帧需要绘制的区块数量很快就能将性能置于死地。一种临时的优化做法是只绘制那些真正出现在屏幕上的区块，但即便如此，要填满iPhone 的 480x320 屏幕还是需要 150 个 32x32 区块，这对于每帧的绘制来说依然数量太多。一种更好的解决方案是预先渲染区块的组块(毕竟区块不会在每一帧中都发生变化)，然后在正确位置上绘制较少组块。

为将这部分代码添加到 TileLayer 类中，用代码清单 18-4 中的代码来替换该类中的 draw 方法。

代码清单 18-4：区块的预渲染

```
prerenderBlock: function(blockX,blockY) {
  var p = this.p,
      tiles = p.tiles,
      sheet = this.sheet(),
      blockOffsetX = blockX*p.blockTileW,
      blockOffsetY = blockY*p.blockTileH;

  if(blockOffsetX < 0 || blockOffsetX >= this.p.cols ||
     blockOffsetY < 0 || blockOffsetY >= this.p.rows) {
       return;
  }

  var canvas = document.createElement('canvas'),
      ctx = canvas.getContext('2d');

  canvas.width = p.blockW;
  canvas.height= p.blockH;
  this.blocks[blockY] = this.blocks[blockY] || {};
  this.blocks[blockY][blockX] = canvas;

  for(var y=0;y<p.blockTileH;y++) {
    if(tiles[y+blockOffsetY]) {
      for(var x=0;x<p.blockTileW;x++) {
        if(tiles[y+blockOffsetY][x+blockOffsetX]) {
          sheet.draw(ctx,
                     x*p.tileW,
                     y*p.tileH,
                     tiles[y+blockOffsetY][x+blockOffsetX]);
        }
      }
    }
  }
},

drawBlock: function(ctx, blockX, blockY) {
  var p = this.p,
```

```
        startX = Math.floor(blockX * p.blockW + p.x),
        startY = Math.floor(blockY * p.blockH + p.y);

    if(!this.blocks[blockY] || !this.blocks[blockY][blockX]) {
      this.prerenderBlock(blockX,blockY);
    }

    if(this.blocks[blockY]  && this.blocks[blockY][blockX]) {
      ctx.drawImage(this.blocks[blockY][blockX],startX,startY);
    }
  },

  draw: function(ctx) {
    var p = this.p,
        viewport = this.parent.viewport,
        viewW = Q.width / viewport.scale,
        viewH = Q.height / viewport.scale,
        startBlockX = Math.floor((viewport.x - p.x) / p.blockW),
        startBlockY = Math.floor((viewport.y - p.y) / p.blockH),
        endBlockX = Math.floor((viewport.x + viewW - p.x) / p.blockW),
        endBlockY = Math.floor((viewport.y + viewH - p.y) / p.blockH);

    for(var y=startBlockY;y<=endBlockY;y++) {
      for(var x=startBlockX;x<=endBlockX;x++) {
        this.drawBlock(ctx,x,y);
      }
    }
  }
```

渲染代码被分成三个独立方法，第一个版本的 draw 方法被换掉，新的 draw 方法计算每个方向上的开始和结束组块，然后为每个组块调用辅助方法 drawBlock。

drawBlock 方法接收区块位置，将其转换成像素位置来计算 startX 和 startY 变量的值。接着检查离屏画布是否已被创建，若尚未创建，则调用 prerenderBlock 方法来创建它。然后，它使用标准的画布方法 drawImage 把离屏画布绘制到屏幕上，drawImage 方法接受一个画布元素作为它的第一个参数(标准的图像对象也可以)。

最后是 prerenderBlock 方法，该方法创建一个尺寸为组块大小的离屏画布，然后绘制组块中的每一个区块。之后，它把画布保存在 this.blocks 属性中以备将来重用。

18.3　处理平台动作游戏的碰撞

如前所述，平台动作游戏的碰撞检测需要予以特别关注。正如之前已介绍过的那样，鉴于关卡的规模，首要问题是精灵和区块间的碰撞需要高度优化；其次要求，碰撞检测应是稳定和相当准确的。因为精灵的大部分时间都花在走动上，嗯，是走在平台上，所以，

引擎应该为这种情况做一些优化。此外，精灵还会跑会跳，通常会制造一些事端；在触碰到物体时，精灵需要基于冲撞的方向做出反馈。

构建一个计算现实情况中的碰撞并对碰撞做出响应的物理引擎，这既很困难也很耗费处理器资源。一种较简单的解决方案是构建一个精灵应如何回应碰撞的简化模型，这样的模型更便于实现，且较少占用处理器。

借鉴旧时的传统平台动作游戏的做法，可将精灵视为一些代表了对象范围的点的一个刚性集合。若某个点还被贴上了 top、left、right、bottom 一类的位置标签，那么如果该点与其他物体发生碰撞的话，确定精灵应有的反应就是一件很容易的事情。若精灵的顶部撞上了某个对象，那么引擎可以向下移动精灵，直至它不再触碰到该对象为止。这一处理同样适用于其他所有方向，见图 18-1。

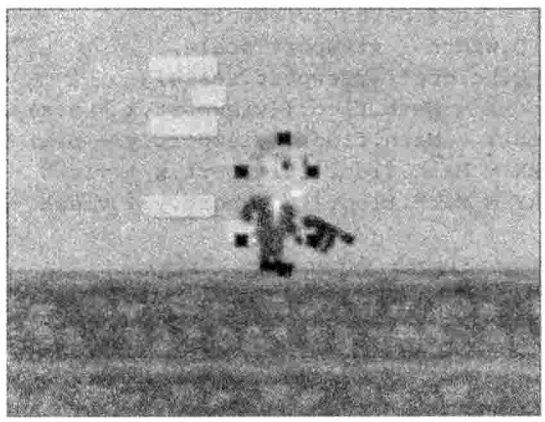


图 18-1 碰撞检测

借助这一算法，任意有形精灵与环境中的区块之间的碰撞检测都可被简化为针对方块来检查点，这是一种成本极低的计算。

18.3.1 添加 2d 组件

添加平台动作游戏碰撞的第一步是创建一个组件，如前所述，精灵可以把该组件加入自身，实现 2D 碰撞检测。所以，该组件也被顺理成章地命名为“2d”，它的任务是支持对碰撞点进行设置；此外，它还借用上一章中的简单的物理代码来支持精灵对重力和速度的回应。

将代码清单 18-5 中的 2d 组件代码添加到 quintus_platformer.js 的末尾处，置于最后的结束花括号之前。

代码清单 18-5：2d 组件

```
Q.gravityY = 9.8*100;
Q.gravityX = 0;
Q.dx = 0.05;

Q.register('2d',{
```

```
  added: function() {
    var entity = this.entity;
    _(entity.p).defaults({
      vx: 0,
      vy: 0,
      ax: 0,
      ay: 0,
      gravity: 1,
      collisionMask: 1
    });
    entity.bind('step',this,"step");
    if(Q.debug) {
      entity.bind('draw',this,'debugDraw');
    }
  },

extend: {
    collisionPoints: function(points) {
      var p = this.p, w = p.w, h = p.h;
      if(!points) {
        p.col = {
          top:   [ [w/2, 0]],
          left:  [ [0, h/3], [0, 2*h/3]],
          bottom:[ [w/2, h]],
          right: [ [w, h/3], [w, 2*h/3]]
        }
      } else {
        p.col = points;
      }
    }
  },

  step: function(dt) {
    var p = this.entity.p,
        dtStep = dt;

    while(dtStep > 0) {
      dt = Math.min(1/30,dtStep);
      // Updated based on the velocity and acceleration
      p.vx += p.ax * dt + Q.gravityX * dt * p.gravity;
      p.vy += p.ay * dt + Q.gravityY * dt * p.gravity;
      p.x += p.vx * dt;
      p.y += p.vy * dt;
      this.entity.parent.collide(this.entity);
      dtStep -= 1/30;
    }
  },
  debugDraw: function(ctx) {
    var p = this.entity.p;
    ctx.save();
```

```
        ctx.fillStyle = "black";
        if(p.col) {
          _.each(p.col,function(points,dir) {
            for(var i=0;i<points.length;i++) {
              ctx.fillRect(p.x + points[i][0] - 2,
                           p.y + points[i][1] - 2,
                           4,4);
            }
          });
        }
        ctx.restore();
      }
    });
```

组件的 added 方法额外加入了一些速度和加速度属性，以及一个倍数，用来指明精灵对重力的响应强度。此外，它还设置了可确定精灵应主动与哪些对象碰撞的碰撞掩码。然后，它绑定 step 事件，在每次步进时更新精灵。作为一项便利措施，若 Q.debug 属性被设成打开标志，它就添加一个到 draw 事件的可选绑定。

2d 组件直接把 collisionPoints 方法加入到了精灵中，让精灵把碰撞点设置成点的一个数组哈希。若没有传入点，该方法创建一些默认的点，这些点划定的范围与精灵的边界框同样大小。

为了基于加速度和速度更新精灵的位置，step 方法用到了类似上一章中的公式。之后，它调用父舞台对象的 collide 方法，该方法的职责是，任何时候只要精灵与别的东西发生碰撞，它就把精灵置于区块对象所在区域外并发送回调。

不过，这一 step 方法有一个与众不同之处，那就是它给 dt 加上了一个 1/30 秒的上限，这样在每次调用时它都有移动，且可通过遍历这些更小的时间步来防止任何精灵移动过快。之所以这样做是因为，精灵不能过深地嵌入到对象中，碰撞机制依赖于这一点，否则就可能触发错误的碰撞点——因为 HTML5 游戏仍偶有可能因垃圾收集而运行不畅。

最后是 debugDraw 方法，若在 added 中被设置了打开标志，该方法会在每个碰撞点的位置上绘制一个小矩形，以供每帧之后的调试用。

18.3.2 计算平台动作游戏的碰撞

计算与精灵的碰撞点发生碰撞的任务落在了上一节创建的 TileLayer 类的身上，它的职责是针对可能发生的区块碰撞来检查精灵的每个点，然后返回碰撞及如何加以改正的信息。

碰撞的计算方法很简单：把每个点的位置除以每个区块的尺寸，查找区块数组，看看是否有区块存在，若有则使用该点的类型来确定应朝哪个方向推动精灵，从而避免碰撞继续发生。

将代码清单 18-6 中的三个方法添加到 quintus_platformer.js 中，放在 Q.TileLayer 类的 prerenderBlock 方法之前，请确保方法的结束逗号已正确对齐。

代码清单 18-6：TileLayer 中的碰撞点检测

```
checkBounds: function(pos,col,start) {
  start = start || 0;
  for(var i=0;i<4;i++) {
    var dir = this.directions[(i+start)%4];
    var result = this.checkPoints(pos,col[dir],dir);
    if(result) {
      result.start = i+1;
      return result;
    }
  }
  return false;
},

checkPoints: function(pos,pts,which) {
  for(var i=0,len=pts.length;i<len;i++) {
    var result = this.checkPoint(pos.x+pts[i][0],
                                 pos.y+pts[i][1],which);
    if(result) {
      result.point = pts[i];
      return result;
    }
  }
  return false;
},

checkPoint: function(x,y,which) {
  var p = this.p,
      tileX = Math.floor((x - p.x) / p.tileW),
      tileY = Math.floor((y - p.y) / p.tileH);

  if(p.tiles[tileY] && p.tiles[tileY][tileX] > 0) {
    this.colBounds.tile = p.tiles[tileY][tileX];
    this.colBounds.direction = which;
    switch(which) {
      case 'top':
        this.colBounds.destX = x;
        this.colBounds.destY = (tileY+1)*p.tileH + p.y + Q.dx;
        break;
      case 'bottom':
        this.colBounds.destX = x;
        this.colBounds.destY = tileY*p.tileH + p.y - Q.dx;
        break;
      case 'left':
        this.colBounds.destX = (tileX+1)*p.tileW + p.x + Q.dx;
        this.colBounds.destY = y;
        break;
      case 'right':
        this.colBounds.destX = tileX*p.tileW + p.x - Q.dx;
```

```
            this.colBounds.destY = y;
            break;
          }
          return this.colBounds;
        }
        return false;
      },
```

其中的主方法 checkBounds 接收一个位置对象和一个碰撞点对象，并通过调用 checkPoints 来针对点对应的区块检查每个点，然后返回所找到的第一个碰撞，否则就返回 false 值。为防止精灵被困在角落中然后慢慢穿过墙壁这样的情况出现，start 参数循环取值检查的起始边。

checkPoints 方法检查点的一个数组，它的做法是遍历数组并调用 checkPoint(单数形式)，然后同样是返回导致碰撞的第一个结果。

最后，真正出力干活的 checkPoint 方法计算点对应的区块位置，若该位置上存在区块，它就填充 colBound 对象，为了节省垃圾收集时间，这是一个在碰撞之间被重用的对象。因为知道导致碰撞的点的类型，所以该方法能够按需要把该点向区块之上、之下或两侧移动，以免两者发生碰撞。

18.3.3 使用 PlatformStage 拼接

为连接 2d 组件和 TileLayer，游戏需要创建一个专用的舞台对象。该舞台对象的任务是针对 TileLayer 来检测对象的碰撞，以及使用一个更简单的边界框来检测与其他任何精灵的碰撞，若有碰撞发生，它调整精灵的位置，然后根据所检测到的碰撞，在必要时调用这些对象的一些事件。

这里的 Q.PlatformStage 类的任务就是处理这些不同的部件，虽然这一舞台类可能把多个区块层放到屏幕上，但它只能把一个区块层用于碰撞，这一区块层称为 collisionLayer。若拥有与 collisionLayer 匹配的 collisionMask，精灵就会执行区块碰撞的处理过程。

此外，该舞台还会执行精灵之间的普通边界框检测。

将代码清单 18-7 中的 Q.PlatformStage 代码添加到 quintus_platformer.js 的末尾处，完成该文件的功能。

代码清单 18-7：PlatformStage 类

```
Q.PlatformStage = Q.Stage.extend({
  collisionLayer: function(layer) {
    this.collision = this.insert(layer);
  },

  _tileCollision: function(obj,start) {
    if(obj.p.col) {
      var result = this.collision.checkBounds(obj.p,obj.p.col,start);
      if(result) {
        return result;
```

```
      }
    }
    return false;
  },

  _hitTest: function(obj,collision) {
    if(obj != this && this != collision &&
       this.p.type && (this.p.type & obj.p.collisionMask )) {
      var col = Q.overlap(obj,this);
      return col ? this : false;
    }
    return false;
  },

  collide: function(obj) {
    var col;
    if(obj.p.collisionMask & this.collision.p.type) {
      while(col = this._tileCollision(obj,col ? col.start : 0)) {
        if(col) {
          var destX = col.destX - col.point[0],
              destY = col.destY - col.point[1];
          obj.p.x = destX;
          obj.p.y = destY;
          if(col.direction == 'top' || col.direction == 'bottom') {
            obj.p.vy = 0;
          } else {
            obj.p.vx = 0;
          }
          obj.trigger('hit',this.collision);
          obj.trigger('hit.tile',col);
        }
      }
    }
    col = this.detect(this._hitTest,obj,this.collision);
    if(col) {
      obj.trigger('hit',col);
      obj.trigger('hit.sprite',col);
    }
  }
});
```

其中的设置器方法 collisionLayer 设置一个应会处理碰撞的 TileLayer 对象。

两个辅助方法——_tileCollision 和_hitTest——分别针对区块层和针对其他精灵来检测对象。若被检测对象有一个碰撞点属性，_tileCollision 调用上一节中的 checkBounds 方法，若没有，就返回 false 值。_hitTest 仅是舞台的标准方法的一个稍加修改版，唯一增加的内容是碰撞(collision)参数，该参数用来检查所针对的精灵是否为碰撞精灵，若是，则把它当成单独处理的区块碰撞而略过。

最后是在每次步进时由所有 2d 精灵调用的 collide 方法，该方法做了大量工作。首先，

它使用一个按位与(bitwise AND)来检查该精灵是否会与碰撞层发生碰撞，若是，它遍历所有可能的区块碰撞，然后根据碰撞告知的做法，基于碰撞返回的 destX 和 destY 参数(由 TileLayer 的 checkPoint 方法返回)来调整精灵的位置。此外，方法还重置精灵的碰撞方向的速度(垂直或水平方向)，每个碰撞也会触发精灵的一个 hit 事件并把碰撞对象(TileLayer 对象)作为数据传入，以及触发一个更具体的 hit.tile 事件，并给该事件传入碰撞自身的信息，这些信息允许精灵在碰撞方向上做出响应。

在处理完区块碰撞后，精灵使用前面提到的_hitTest 方法来调用 detect 方法，遍历其他所有精灵，若有碰撞发生，则触发 hit 事件。

18.4 构建游戏

所有部件都已准备就绪，现在是时候把精力放在游戏上了。本节构建的这个简单的平台动作游戏用到了第 16 章中的人物角色一个缩小版，此外，它还把一些黏糊糊的团状怪物用作敌人。该游戏只有三种精灵：玩家、子弹和团状怪物。

18.4.1 自建游戏

由外而内，在填写必需的精灵类之前，先来搭建游戏的框架。打开 platform.js，使用代码清单 18-8 中代码替换其中的内容。

代码清单 18-8：平台动作游戏的代码

```
$(function() {
  var Q = window.Q = Quintus()
                  .include('Input,Sprites,Scenes,Anim,Platformer')
                  .setup('quintus', { maximize: true })
                  .controls();

  Q.Enemy = Q.Sprite.extend({
    // TODO
  });
  Q.Player = Q.Sprite.extend({
     // TODO
  });
  Q.Bullet = Q.Sprite.extend({
     // TODO
  });

  Q.scene('level',new Q.Scene(function(stage) {
    stage.insert(new Q.Repeater({ asset: 'background-wall.png',
                                  speedX: 0.50, y:-225, z:0 }));
    var tiles = stage.insert(new Q.TileLayer({ sheet: 'block',
                                               x: -100, y: -100,
                                               tileW: 32,
                                               tileH: 32,
```

```
                                        dataAsset: 'level.json',
                                        z:1 }));
    stage.collisionLayer(tiles);
    var player = stage.insert(new Q.Player({ x:100, y:0,
                                      z:3, sheet: 'man' }));

    stage.insert(new Q.Enemy({ x:400, y:0, z:3 }));
    stage.insert(new Q.Enemy({ x:600, y:0, z:3 }));
    stage.insert(new Q.Enemy({ x:1200, y:100, z:3 }));
    stage.insert(new Q.Enemy({ x:1600, y:0, z:3 }));

    stage.add('viewport');
    stage.follow(player);
  }, { sort: true }));

  Q.load(['sprites.png','sprites.json',
          'background-wall.png','level.json'],function() {
    Q.compileSheets('sprites.png','sprites.json');

    Q.animations('player', {
      run_right: { frames: _.range(7,-1,-1), rate: 1/15},
      run_left: { frames: _.range(19,11,-1), rate:1/15 },
      fire_right: { frames: [9,10,10], next: 'stand_right', rate: 1/30 },
      fire_left: { frames: [20,21,21], next: 'stand_left', rate: 1/30 },
      stand_right: { frames: [8], rate: 1/5 },
      stand_left: { frames: [20], rate: 1/5 },
      fall_right: { frames: [2], loop: false },
      fall_left: { frames: [14], loop: false }
    });

    Q.animations('blob', {
      run_right: { frames: _.range(0,2), rate: 1/5 },
      run_left: { frames: _.range(2,4), rate: 1/5 }
    });
    Q.stageScene("level",0,Q.PlatformStage);
  });
});
```

这段代码设置引擎和场景，然后加载一些资产并设置一些动画。所有的这些设置部分看上去都应与之前各章及本章之前的例子十分类似。

这一次，玩家在每个方向上都会用到一些动画，为邪恶的团状怪物角色设计的第二组简单动画也被添加进来。

下面将依次加入三个精灵。

18.4.2　创建敌人

敌人精灵(亦称“团状怪物”)就是一个在平台上来回移动的精灵，并且会在碰到墙壁时改变方向。为此，它需要监听 **hit.tile** 事件，在该事件被触发时反转方向。此外，它还需

要在触碰玩家时给玩家造成一定伤害，这可以通过监听 hit.sprite 事件来实现。

使用代码清单 18-9 中的代码替换掉 Q.Enemy 类的存根。

代码清单 18-9：Q.Enemy 类

```
Q.Enemy = Q.Sprite.extend({
  init:function(props) {
    this._super(_(props).extend({
      sheet: 'blob',
      sprite: 'blob',
      rate: 1/15,
      type: 2,
      collisionMask: 5,
      health: 50,
      speed: 100,
      direction: 'left'
    }));
    this.bind('damage',this,'damage');
    this.bind('hit.tile',this,'changeDirection');
    this.bind('hit.sprite',this,'hurtPlayer');
    this.add('animation, 2d')
        .collisionPoints()
  },

  changeDirection: function(col) {
    if(col.direction == 'left') {
     this.p.direction = 'right';
    } else if(col.direction == 'right') {
      this.p.direction = 'left';
    }
  },

  hurtPlayer: function(col) {
    if(col.p.x < this.p.x) {
      col.p.x -= 10;
      col.damage(5);
    } else {
      col.p.x += 10;
      col.damage(5);
    }
  },

  damage: function(amount) {
    this.p.health -= amount;
    if(this.p.health <= 0) {
      this.destroy();
    }
  },

  step: function(dt) {
```

```
    var p = this.p;
    if(p.direction == 'right') {
      this.play('run_right');
      p.vx = p.speed;
    } else {
      this.play('run_left');
      p.vx = -p.speed;
    }
    this._super(dt);
  }
});
```

照例，init 方法设置属性并绑定事件，此外，它还把 animation 和 2d 组件添加到精灵中，并把默认的碰撞点用于精灵。在这个例子中，type 和 collisionMask 很重要，因为 type 被设置为 2——这样子弹就能够区分敌人和玩家——以及 collisionMask 被设置为 5——这样敌人就能与玩家(类型 4)和区块(类型 1)相碰撞。

changeDirection 方法被调用来响应 hit.tile 事件，碰撞的详细信息被传入该事件中，其中包括碰撞发生的位置；团状怪物的移动方向与碰撞方向相反。

hurtPlayer 在团状怪物每次遇到另一个精灵时被调用，因为游戏中唯一的另一个与团状怪物的 collisionMask 匹配的精灵是玩家，所以团状怪物知道可以伤害他们。在子弹遇到团状怪物时，情况则刚好相反，这种情况下，会调用团状怪物的 damage，团状怪物的健康值被减少，直至少于 0 值，然后团状怪物会销毁自身。

因为团状怪物只能向左或向右移动，所以 step 方法很简单，它检查自身当前的移动方向，设置正确方向的速度，然后播放适当的动画。

18.4.3　添加子弹

为除掉团状怪物，玩家需要一些火力。子弹是一个很简单的精灵，甚至不需要使用精灵表来绘制自身，只需代以绘制一个很小的矩形即可。

使用代码清单 18-10 中的代码来替换掉子弹类的存根。

代码清单 18-10：子弹精灵

```
Q.Bullet = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props).extend({ w:4, h:2,
                                  gravity:0, collisionMask:3 }));
    this.add('2d')
    this.collisionPoints();
    this.bind('hit.tile',this,'remove');
    this.bind('hit.sprite',this,'damage');
  },

  remove: function() {
    this.destroy();
  },
```

```
  damage: function(obj) {
    obj.trigger('damage',10);
    this.destroy();
  },

  draw: function(ctx) {
    var p = this.p;
    ctx.fillStyle = "#000";
    ctx.fillRect(p.x,p.y,p.w,p.h);
  }
});
```

子弹类的代码不多，它把 gravity 属性设为 0，避免重力影响子弹。此外，在碰到区块以及给任何碰到的精灵造成伤害之后，它就会销毁自身。最后，它重写 draw 方法，仅绘制一个很小的黑色矩形来代表子弹。

同样，type 和 collisionMask 发挥了它们的作用，这样子弹就只与精灵和敌人发生碰撞。

18.4.4 创建玩家

最后是玩家类，玩家有一些额外的复杂性，因为玩家需要移动、跳跃和发射子弹。

一个始终需要予以关注的问题是跳跃问题，角色应只有站在坚实的地面上时才可以跳跃，所以，玩家需要记录下所站的位置。一种实现做法是记录上一次与底部(bottom)的点放生的碰撞，然后只有在上一次的底部碰撞发生在刚过去的几帧中时才允许跳跃，这就是这里用到的技术。

为同步子弹发射动画和子弹真正射出去的时间，玩家可以先监听指明子弹动画已经播放完毕的事件，之后再真正把子弹精灵添加到舞台中。

将代码清单 18-11 中的代码添加进来，加入玩家精灵，完成平台动作游戏的编写。

代码清单 18-11：玩家类

```
Q.Player = Q.Sprite.extend({
  init:function(props) {
    this._super(_(props).extend({
      sheet: 'man',
      sprite: 'player',
      rate: 1/15,
      speed: 250,
      standing: 3,
      type: 4,
      health: 100,
      collisionMask: 1,
      direction: 'right'
    }));

    this.add('animation, 2d')
        .collisionPoints({
```

```
        top: [[ 20, 3]],
        left: [[ 5,15], [ 5,40]],
        bottom: [[ 20,51 ]],
        right: [[ 30,15], [ 30,40]]
      });

  this.bind('animEnd.fire_right',this,"launchBullet");
  this.bind('animEnd.fire_left',this,"launchBullet");
  this.bind('hit.tile',this,'tile');
  Q.input.bind('fire',this,"fire");
  Q.input.bind('action',this,"jump");
},

fire: function() {
  this.play('fire_' + this.p.direction,2);
},

damage: function(amount) {
  this.p.health -= amount;
  if(this.p.health < 0) {
    Q.stageScene("level",0,Q.PlatformStage);
  }
},
launchBullet: function() {
  var p = this.p,
      vx = p.direction == 'right' ? 500 : -500,
      x = p.direction == 'right' ? (p.x + p.w) : p.x;
  this.parent.insert(new Q.Bullet({ x: x, y: p.y + p.h/2, vx: vx }));
},

jump: function() {
  if(this.p.standing >= 0) {
    this.p.vy = -this.p.speed * 1.4;
    this.p.standing = -1;
  }
},

tile: function(collision) {
  if(collision.direction == 'bottom') {
    this.p.standing = 5;
  }
},

step: function(dt) {
  var p = this.p;
  if(p.animation == 'fire_right' || p.animation == 'fire_left') {
    if(this.p.standing > 0) {
      this.p.vx = 0;
    }
  } else {
```

```
      if(this.p.standing < 0) {
        if(p.vx) {
          p.direction = p.vx > 0 ? 'right' : 'left';
        }
        this.play('fall_' + p.direction,1);
      }
      if(Q.inputs['right']) {
        this.play('run_right');
        p.vx = p.speed;
        p.direction = 'right';
      } else if(Q.inputs['left']) {
        this.play('run_left');
        p.vx = -p.speed;
        p.direction = 'left';
      } else {
        p.vx = 0;
        this.play('stand_' + p.direction);
      }
      this.p.standing~DH;
    }
    this._super(dt);
  }
});
```

相比于 Bullet 和 Enemy，Player 精灵需要处理的事情更多。init 方法为 collisionPoints 设置一组定制的点，从而支持与区块之间的更精确交互。此外，它还绑定了一些碰撞、跳跃和发射子弹的事件。

fire 方法会在玩家发射子弹时被调用，它以一个较高的优先级来播放 fire_right 和 fire_left 动画，以防这两个动画被其他任何动画覆盖；damage 回调则会在团状怪物击中玩家时被调用。在整个发射动画播放完毕后，动画引擎触发一个事件，通知玩家是时候真正把 Bullet 精灵绘制到画布上。

tile 方法用来记录与区块发生的碰撞，这样做的唯一目的是跟踪玩家是否站在区块上。它所计算的 standing 属性被 jump 方法用来把玩家弹射到空中，但只有在玩家当前是站在坚实地面上时才会这样做。

最后，最复杂的 step 方法负责播放正确的动画，并随着玩家在屏幕上的跑动来更新玩家的速度。

该方法考虑了几种不同的状态：发射子弹、下落、跑动和站立不动，第一种状态被认为是最重要的，可覆盖其他状态。下落动画是次重要的状态，因为若玩家正在跳跃，他的脚就不应在空中走动。接下来，若玩家按下了右或左按键，方法就播放相应方向的跑动动画。若以上这些条件无一为真，玩家就站立不动，面向玩家松手时的方向。

现在，你应能够在 level.json 所定义的很短的关卡中走动、跳跃和发射子弹了。在图 18-2 中，可以看到运行在 iPhone 上的最终游戏画面。

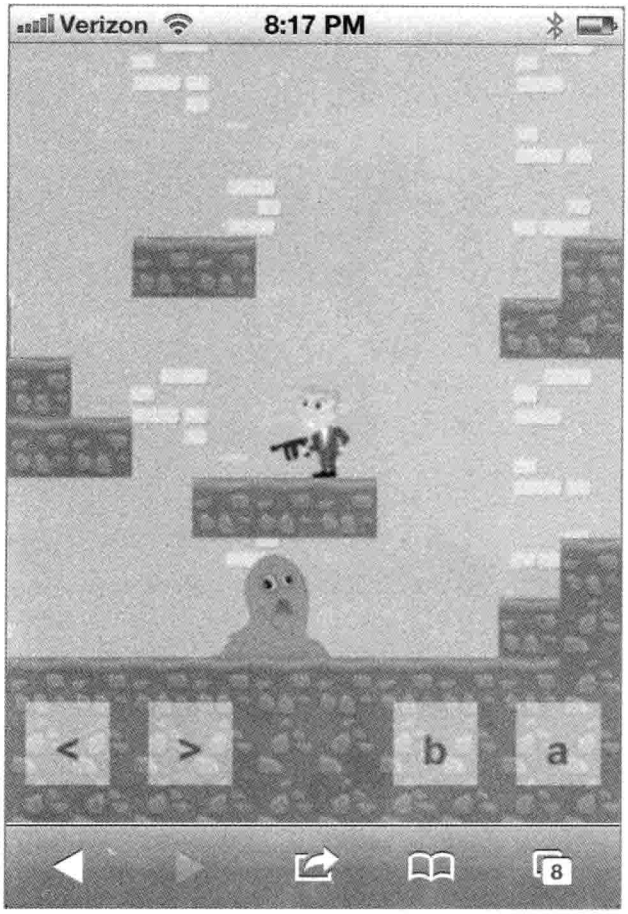


图 18-2　最终的游戏画面

下一章将向你展示如何使用一个关卡编辑器来编辑需要用来创建关卡但却不容易读写的.json 文件，从而为玩家构建更具娱乐性的游戏场所。

18.5　小结

现在，你已经把画布和动画的代码连同一些新的区块和碰撞检测整合在一起，构建了一个简单平台动作游戏。若要置于游戏产品中，那么引擎还存在许多亟待优化和增强的地方，这包括增加用来实现更精确精灵碰撞的多边形到多边形的碰撞，以及使用四叉树或基于区块的系统来尽量减少精灵碰撞计算等。

第19章

构建一个画布编辑器

本章提要

- 使用 Node.js 提供游戏服务
- 创建一个触摸友好的编辑器
- 保存关卡数据

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 19 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

19.1 引言

以上一章中构建的 2D 平台动作游戏为基础，本章构建一个编辑器，该编辑器使得你能够编辑关卡，修改组成关卡的区块数据，并且能够保存这些被修改的关卡。为了支持关卡数据的保存，需要把游戏封装在一个 Node.js 应用之内，该应用能将保存的游戏数据输出到文件中。

19.2 使用 Node.js 提供游戏服务

在应用能够处理关卡数据的保存需求之前，游戏需要通过静态 Web 服务器以外的另一种方式加以提供。其中的一种解决方案是编写一个简单的 PHP 脚本，该脚本接收数据并将

数据保存到硬盘的文件中。不过，因为本书与 JavaScript 相关，而且在接下来的几章中，你将使用 Node.js 来构建一个多玩家游戏，所以，最好使用 Node 来构建编辑器，以便从中获取更多的 Node 经验。

19.2.1 创建 package.json 文件

与第 8 章中的精灵表创建器应用类似，编辑器需要用 package.json 文件来让 Node 知道一些关于应用的细节，其中包括依赖等。

为编辑器创建名为 editor 的新目录，接着使用代码清单 19-1 中的内容来创建名为 package.json 的文件。

代码清单 19-1：package.json 文件

```
{
    "name": "platformer-editor"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "jade": ">= 0.0.1"
    , "underscore": "1.3.3"
  }
}
```

该应用有三项依赖，分别是你十分熟悉的 Underscore.js、一个名为 express 的 Node.js 应用框架和该框架的依赖 jade。

19.2.2 设置 Node 以提供静态资产

Node.js 提供了用于处理 Web 请求的最基本功能，要让它完成诸如提供静态文件之类的事情，需要加入一个模块。就其唯一目标为提供静态文件来说，存在许多可供你使用的不同模块，其中包括 node-static 和 node-paperboy 等；不过，因为这一应用的目标不仅是提供文件，所以加入一个除了处理静态文件之外还能处理其他任务的功能更全面的框架，这样更合理一些。出于这一考虑，这里使用的框架是一个名为 express 的 Node 模块：http://expressjs.com/。

express 提供了许多不同的功能，包括视图、缓存、路由、会话和静态文件等。本章只用到 express 功能的一个很小的子集，但相比于使用没有模块支持的 Node.js，它仍能减轻你的工作。你将通过 npm 安装 express，所以不必担心它的下载问题。

在刚为应用创建的 editor 目录下创建一个名为 app.js 的文件，填入代码清单 19-2 中的内容，这是一个简单的 express 样板应用。

代码清单 19-2：express 样板应用

```
var express = require('express'),
    fs = require('fs'),
```

```
    _ = require('underscore');

var app = module.exports = express.createServer();

// Configuration
app.configure(function(){
  app.use(express.bodyParser());
  app.use(express.static(__dirname + '/public'));
});
app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});
app.configure('production', function(){
  app.use(express.errorHandler());
});

// Start the server on port 3000
app.listen(3000, function(){
  console.log("Express server listening on port %d in %s mode",
app.address().port, app.settings.env);
});
```

该应用首先加载几个依赖，把服务器设置成使用 express.bodyParser 解析传入的 POST 消息，并使用 express.static 把 public 目录设置成提供静态资产的地方。接着，它根据服务器是运行在开发(Development)模式还是生产(Production)模式来设置几个错误处理程序(默认是开发模式)。最后，它在端口 3000 上启动服务器。

要试运行这一应用，把上一章中的所有代码、图像、js 以及数据目录都复制到一个比 app.js 文件低一层的名为 public 的子目录下，并把文件 platform.html 改名为 index.html。

然后，需要运行 npm 来安装必需的模块，通过命令提示符窗口，在 package.json 和 app.js 文件所在的目录下运行以下命令：

```
npm install
```

该命令创建所需的 node_modules 目录并安装依赖。

应通过运行以下命令来运行应用：

```
node app.js
```

该命令在端口 3000 上启动服务器，允许你在浏览器中通过访问 http://localhost:3000/ 来运行上一章的平台动作游戏。若找出了 IP 地址，那么你也可以像在计算机上那样，在同一 Wi-Fi 网络中通过移动设备来玩该游戏。

19.3　创建编辑器

编辑器由一个位于现有的平台动作游戏代码之上的编辑工具层构成，通过最大限度地

利用现有的平台动作游戏代码，编辑器只需加入移动视图和修改区块的功能。

19.3.1 修改平台动作游戏代码

为将编辑器代码加入到现有的 platform.js 文件中而同时尽量减少改动，可打开 public/ 目录中的 index.html 文件，把即将创建的 quintus_editor.js 文件加入其中，如代码清单 19-3 所示。

代码清单 19-3：修改后的 index.html 文件

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
          minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Platformer</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>
    <script src='js/quintus_anim.js'></script>
    <script src='js/quintus_platformer.js'></script>
    <script src='js/quintus_editor.js'></script>
    <script src='platform.js'></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

接下来，打开同一目录中的 platform.js 文件，在该文件的开头加入 Editor 模块并删除控件调用(稍后由编辑器启用控件)。

```
$(function() {
  var Q = window.Q
        = Quintus()
          .include('Input,Sprites,Scenes,Anim,Platformer,Editor')
          .setup('quintus', { maximize: true });
```

下一步，为了支持编辑器和不同关卡的加载，需要对同一文件末尾处的内容做一些修改，使用一个正则表达式来有选择地把不同关卡文件加载到游戏中，并通过设置把编辑器加到游戏中。

需要修改的各行代码如下所示(第一块代码应添加到 Q.scene 调用之前，余下的改动都

位于 Q.scene 回调的内部)：

```
var match = window.location.search.match(/level=([^\&]+)/),
    levelFile = 'level.json';
if(match) {
  levelFile = match[1] + '.json';
}

Q.scene('level',new Q.Scene(function(stage) {
  stage.insert(new Q.Repeater({ asset: 'background-wall.png',
                                speedX: 0.50, y:-225, z:0 }));
  var tiles = stage.insert(new Q.TileLayer({ sheet: 'block',
                                             x: -100, y: -100,
                                             tileW: 32,
                                             tileH: 32,
                                             dataAsset: levelFile,
                                             z:1 }));
  stage.collisionLayer(tiles);
  var player = stage.insert(new Q.Player({ x:100, y:0, z:3 }));
  stage.insert(new Q.Enemy({ x:400, y:0, z:3 }));
  stage.insert(new Q.Enemy({ x:600, y:0, z:3 }));
  stage.insert(new Q.Enemy({ x:1200, y:100, z:3 }));
  stage.insert(new Q.Enemy({ x:1600, y:0, z:3 }));
  stage.add('viewport');
  stage.follow(player);

  stage.add('editor');
  stage.editor.setFile(levelFile);
  stage.bind('reset',function() {
    Q.stageScene("level",0,Q.PlatformStage);
  });

}, { sort: true }));
Q.load(['sprites.png','sprites.json',
        'background-wall.png',levelFile],function() {

  Q.compileSheets('sprites.png','sprites.json');
  Q.animations('player', {
    run_right: { frames: _.range(7,-1,-1), rate: 1/15},
    run_left: { frames: _.range(19,11,-1), rate:1/15 },
    fire_right: { frames: [9,10,10], next: 'stand_right', rate: 1/30 },
    fire_left: { frames: [20,21,21], next: 'stand_left', rate: 1/30 },
    stand_right: { frames: [8], rate: 1/5 },
    stand_left: { frames: [20], rate: 1/5 },
    fall_right: { frames: [2], loop: false },
    fall_left: { frames: [14], loop: false }
  });
  Q.animations('blob', {
    run_right: { frames: _.range(0,2), rate: 1/5 },
    run_left: { frames: _.range(2,4), rate: 1/5 }
```

```
    });
    Q.stageScene("level",0,Q.PlatformStage);

});
```

其中的主要变化是增加了 levelFile 变量，该变量存放被加载到编辑器中的关卡文件的名称。这可通过把一个名为 level 的参数添加到 URL 的末尾处来实现，例如：

```
http://localhost:3000/?level=level2
```

这一请求将尝试加载 data/level2.json 而非默认文件 data/level.json。

这里标出的第二处改动是把 editor 组件添加到了舞台中，该组件尚未编写，所以现在运行代码会导致错误，不过下一节会完成这一工作。

此外，这部分代码还将一个名为 reset 的新事件绑定到了舞台对象上，编辑器使用该事件来告诉游戏重启自身(游戏仅重载所需的场景来实现这一点)。

19.3.2 创建编辑器模块

Quintus.Editor 模块仅包括了一个组件，名为 editor，该组件可被添加到 Stage 对象中，它创建一些工具按钮来让用户移动游戏、涂绘区块、擦除区块、选择区块、放大缩小，以及最后把关卡存回到服务器上。

编辑器的首个版本把一些按钮添加到屏幕上，并支持你在各种工具间进行选择。

创建文件 js/quintus_editor.js，将代码清单 19-4 中的代码加入其中，实现一个可运行的编辑器。

代码清单 19-4：基本的编辑器模块

```
Quintus.Editor = function(Q) {

  Q.register('editor',{

    added: function() {
      var stage = this.entity;
      stage.pause();
      $("#quintus-editor").remove();
      this.controls = $("<div id='quintus-editor'>")
          .appendTo(Q.wrapper)
          .css({position:"absolute",top:0, zIndex: 100});
      _.bindAll(this);

      this.buttons = {
        move: this.button("move",this.move),
        paint: this.button("paint",this.paint),
        erase: this.button("erase",this.erase),
        select: this.button("tile",this.tile),
        play: this.button("play",this.play),
        out: this.button("-",this.out),
```

```
    in: this.button("+",this.in),
    save: this.button("save",this.save)
  };

  this.select('move');
  this.activeTile = 1;

  Q.el.on('touchstart mousedown',this.touch);
  Q.el.on('touchmove mousemove',this.drag);
  Q.el.on('touchend mouseup',this.release);
},

setFile: function(levelFile) {
  this.levelFile = levelFile;
},

button: function(text,callback) {
  var elem = $("<div>")
             .text(text)
             .css({float:'left',
                   margin: "10px 5px",
                   padding:"15px 5px",
                   backgroundColor:'#DDD',
                   width:35,
                   textAlign:'center',
                   fontSize: "14px",
                   cursor:'pointer',
                   fontFamily: 'Arial',
                   fontWeight:'bold',
                   boxShadow: "2px 2px 5px #999",
                   borderRadius: "5px",
                  color:"black"})
              .appendTo(this.controls);

  elem.on('mousedown touchstart',callback);
  return elem;
},

select: function(button) {
  if(this.selected) {
    this.buttons[this.selected].css('backgroundColor','#DDD');
  }
  this.selected = button;
  if(this.buttons[this.selected]) {
    this.buttons[this.selected].css('backgroundColor','#FFF');
  }
},

move: function() {
  this.select('move');
```

```
    },

    paint: function() {
      this.select('paint');
    },

    erase: function() {
      this.select('erase');
    },

    play: function(e) {
      if(this.playing) {
        this.buttons['play'].text('Play');
        Q.input.disableTouchControls();
        this.entity.trigger('reset');
        this.select();
      } else {
        Q.el.off('touchstart mousedown',this.touch);
        Q.el.off('touchmove mousemove',this.drag);
        Q.el.off('touchend mouseup',this.release);
        this.select('play');
        this.buttons['play'].text('reset');
        this.playing = true;
        this.entity.unpause();
        Q.controls();
      }
      e.preventDefault();
    },

    out: function() {
      this.entity.viewport.scale /= 1.5;
      this.entity.viewport.recenter();
    },

    in: function() {
      this.entity.viewport.scale *= 1.5;
      this.entity.viewport.recenter();
    },
  });
};
```

如你所知，在将组件被添加到某个对象后，就会立即调用 added 方法。该方法所做的第一件事情是暂停舞台对象，这样 step 方法就不再被调用，所有精灵都会停在原地不动。接着，它设置编辑器的容器，同时还删除任何有着相同 ID 的元素，这在重置编辑器时很有用。然后，编辑器调用_bindAll 方法，该方法把对象的所有方法都和对象自身绑定起来，防止 jQuery 的事件回调在用到 this 对象状态时出现混乱。该方法设置了一些被添加到屏幕顶部的按钮，这些按钮都用到了 button 方法，这一方法的定义被放在其他几个方法之后。接下来，added 方法预选择了 move 工具，把 activeTile(被用来涂绘的区块)设置为第一个精

灵，并把一些事件处理程序绑定到触摸事件和等价的鼠标事件上。

touch、drag 和 release 方法直到下一节才会被添加进来，若如这里所做的那样把 null 值传给 on 方法，jQuery 不会引发异常。

setFile 方法只是保存关卡文件的名称以备将来使用。

button 方法创建一个矩形状的 DOM 元素，使用一些样式把它的外观变得类似按钮。然后，它把该元素添加到 added 方法创建的容器中，这些按钮用来控制编辑器和选择当前激活的工具。

select 按钮用来高亮显示当前激活的工具，虽然某些按钮在你单击它们时就会有所动作，不过前面三个按钮——move、paint 和 erase——被用作在用户单击或触摸画布时控制所发生事情的工具，这三个按钮各自的回调方法的任务仅是调用 select 方法来设置当前的工具。

可使用 play 按钮在 Editing 模式和激活游戏这两者之间进行切换，这样就可以对关卡进行测试。在首次按下该按钮时，编辑器关闭所有的绑定事件，选择 play 按钮，然后取消暂停舞台对象，当再次按下时，它触发 reset 事件，舞台对象使用该事件来重置场景。

最后两个方法——out 和 in——修改视口来进行缩小或放大。

若该代码运行正确，编辑器被放到了浏览器的顶部，那么你应可在前面三个工具之间进行选择，并可使用+和-按钮来放大和缩小，以及按下 Play 按钮开始游戏。目前这些工具还什么都不做，不过下一节会改进这一问题。

19.3.3　添加触摸和鼠标事件

为了把这些代码变成编辑器，需要编写三个缺失的事件方法——touch、drag 和 release——查看当前的工具并采取适当的行动。这三个方法的代码及两个辅助方法 tilePos 和 tool 的代码如代码清单 19-5 所示，这些代码应被添加到 quintus_editor.js 的末尾处，置于 editor 组件定义的内部。

代码清单 19-5：画布的事件方法

```
touch: function(e) {
  var touch = e.originalEvent.changedTouches ?
            e.originalEvent.changedTouches[0] : e,
      stage = this.entity;
  this.start = { pageX: touch.pageX, pageY: touch.pageY };
  this.viewportX = stage.viewport.centerX;
  this.viewportY = stage.viewport.centerY;
  this.tool(touch);
  e.preventDefault();
},

drag: function(e) {
  var touch = e.originalEvent.changedTouches ?
            e.originalEvent.changedTouches[0] : e,
      stage = this.entity;
```

```
    if(this.start) {
      this.tool(touch);
    }
    e.preventDefault();
  },

  release: function(e) {
    this.start= null;
  },

  tilePos: function(x,y) {
    var canvasPos = $(Q.el).offset(),
        canvasX = (x - canvasPos.left) / Q.el.width() * Q.width,
        canvasY = (y - canvasPos.top) / Q.el.height() * Q.height,
        viewport = this.entity.viewport,
        tileLayer = this.entity.collision,
        tileX = Math.floor( (canvasX / viewport.scale +
                             viewport.x - tileLayer.p.x)
                           / tileLayer.p.tileW),
        tileY = Math.floor( (canvasY / viewport.scale +
                             viewport.y - tileLayer.p.y)
                           / tileLayer.p.tileH);
    return { x: tileX, y: tileY };
  },

  tool: function(touch) {
    var stage = this.entity,
        viewport = stage.viewport;
    switch(this.selected) {
      case 'move':
        stage.centerOn(this.viewportX +
                       (this.start.pageX - touch.pageX)
                        / viewport.scale,
                      this.viewportY +
                       (this.start.pageY - touch.pageY)
                        / viewport.scale);
        break;
      case 'paint':
        var tile = this.tilePos(touch.pageX, touch.pageY);
        stage.collision.setTile(tile.x,tile.y,this.activeTile);
        break;
      case 'erase':
        var tile = this.tilePos(touch.pageX, touch.pageY);
        stage.collision.setTile(tile.x,tile.y,0);
        break;
    }
  },
```

touch 方法被 touchstart 和 mousedown 事件调用，该方法提取鼠标事件数据或第一个变动触摸的数据，然后保存事件的起始位置及原始的视口中心位置。之后，它调用 tool 方

法，该方法基于当前所选的工具来完成实际工作。

drag 方法被 touchmove 和 mousemove 事件调用，该方法首先检查当前是否存在编辑器正在跟踪的行为，若是，则调用 tool 方法来采取任何必要的行动。

最后是 release 方法，该方法仅把 start 属性设置为 null，因为该属性被 drag 用来判断自身是否应该做一些事情，这实际上是停止工具的作用。

tilePos 方法的任务稍有棘手，要基于页面上的 x 和 y 位置来找出确切的区块位置。它的计算方法如下：首先确定在画布元素上的像素位置(存放在 canvasX 和 canvasY 中)，然后使用视口缩放比例、视口位置和区块层偏移来计算出准确偏移位置，然后把该位置除以每个区块的大小，从而得到区块的位置(存放在 tileX 和 tileY 中)。

tool 方法查看当前被激活的工具，若 move 被选中，舞台对象根据用户拖动手指或移动鼠标的距离来重设中心位置。若 paint 被选中，它把当前位置的区块设为激活区块；若 erase 被选中，则清除该区块。

若在浏览器或移动设备中加载编辑器，你应能移动、涂绘和擦除区块。图 19-1 展示的是 iPhone 上以横屏模式显示的编辑器看起来的样子。

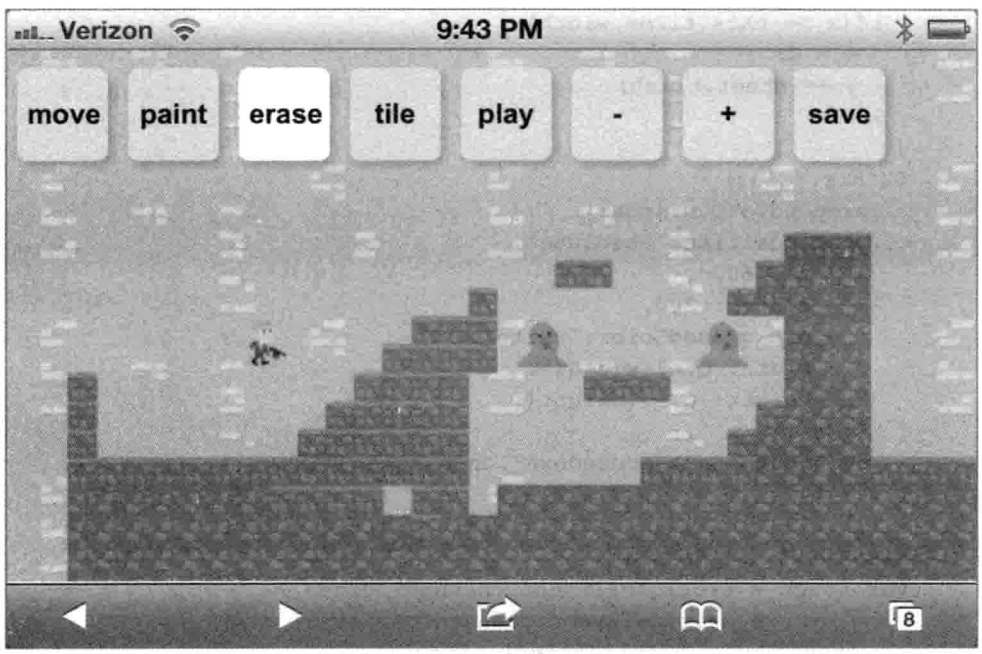


图 19-1　iPhone 上的编辑器界面

19.3.4　选择区块

现在只剩下两部分功能尚未实现：选择要涂绘的区块的能力以及把关卡保存回服务器上的代码。本节添加第一部分功能：区块选择。选择区块的机制使得 tile 按钮能够弹出精灵表中所有区块的一幅图像，并且允许用户选择要绘制的区块。为此，需要创建一个画布元素来存放图像，然后把该画布的事件转换成区块选择。

为加入区块功能，将代码清单 19-6 中的代码添加到 quintus_editor.js 中，放在组件定义的末尾处。

代码清单 19-6：区块功能

```
tile: function() {
  if(!this.tiles) this.setupTiles();
  $(this.tiles).show();
},

setupTiles: function() {
  var sheet = this.entity.collision.sheet();
  this.tiles = document.createElement("canvas");
  this.tiles.width = Q.el.width();
  this.tiles.height = Q.el.height();
  var x = 0, y = 0, ctx = this.tiles.getContext('2d');
  for(var i=0;i<sheet.frames;i++) {
    sheet.draw(ctx, x, y, i);
    x += sheet.tilew;
    if(x >= this.tiles.width) {
      x = 0;
      y += sheet.tileh;
    }
  }
  $(this.tiles)
    .prependTo(Q.wrapper)
    .css({position:'absolute',
          top:60,
          zIndex: 200,
          backgroundColor:'white',
          width: Q.el.width(),
          height: Q.el.height()
        })
    .on('touchstart mousedown',this.selectTile);
},

selectTile: function(e) {
  var touch = e.originalEvent.changedTouches ?
            e.originalEvent.changedTouches[0] : e,
      canvasPos = $(this.tiles).offset(),
      canvasX = (touch.pageX - canvasPos.left),
      canvasY = (touch.pageY - canvasPos.top),
      tileLayer = this.entity.collision,
      sheet = tileLayer.sheet(),
      tileX = Math.floor(canvasX / sheet.tilew),
      tileY = Math.floor(canvasY / sheet.tileh),
      frame = tileX + tileY *
            Math.floor(this.tiles.width / sheet.tilew);
   $(this.tiles).hide();
   if(frame <= sheet.frames) {
```

```
        this.activeTile = frame;
      }
      e.preventDefault();
    },
```

tile 是主要的工具方法，它的任务是先检查区块画布是否已设置，若还没有就调用 setupTiles 方法来生成它。之后，它仅显示该元素来阻塞屏幕的其余处理，等待选择区块这一元素事件的发生。

setupTiles 方法创建一个画布元素，在上面绘制区块地图精灵表中的每个区块，再把该元素添加到编辑器的控件 DOM 元素 controls 中。然后，它添加一个事件处理程序，允许用户通过单击或触摸区块来选择区块。

最后是 selectTile 方法，该方法执行一点数学运算，根据区块的尺寸和画布的尺寸来算出帧数，从而找出想要的区块帧。然后，它检查该帧是否为一个有效帧，若是则设置 activeTile。

在浏览器中运行该编辑器，现在你应能选择不同区块来构造页面上的元素了。

19.4　添加关卡保存支持

编辑器的最后一项任务是支持把已修改的关卡保存回服务器中这一功能。在客户端，如代码清单 19-7 所示，实现这一功能的代码很简单。照例，这段代码应放在同一位置，即 quintus_editor.js 中组件定义的末尾处。

代码清单 19-7：编辑器的保存方法

```
save: function() {
  var levelName = prompt("Level Name?",this.levelFile);
  if(levelName) {
    $.post('/save',{ tiles: this.entity.collision.p.tiles,
                     level: levelName });
  }
}
```

这段代码弹出浏览器提示框，要求用户输入文件名称，然后使用$.post 把该文件名称连同碰撞层的 tile 属性中的区块数据一起直接给服务器发送回去。

在服务器端，事情几乎同样简单，做法是使用 express 的语法把路由添加到服务器中，将代码清单 19-8 中的代码添加到 app.js 的末尾处。

代码清单 19-8：服务器端的保存方法

```
app.post('/save', function(req, res){
  var data = _(req.body.tiles).map(function(row) {
    return _(row).map(function(tile) { return Number(tile); });
  });
  fs.writeFile("public/data/" + req.body.level,
```

```
                JSON.stringify(data));
    res.send(201);
});
```

这段代码简单定义到/save 的一个路由，该路由提取被送进来的数据，把 tiles 数据转换成一些数值，然后把它们写到文件中。默认情况下，被正确送递进来的区块数据以数组的形式存在，不过数组中的每个元素都是一个字符串，所以需要把它转换成数值。

写入文件的过程很简单，只需调用 fs.writeFile 并传入文件名称和被转换成 JSON 字符串的关卡数据即可，关卡数据的转换使用 JSON.stringify 实现。

安全警告

这段代码不应被用作部署到公开的 Web 服务器上的内容，而应只是你可在内部用来定制关卡的工具。因为这一代码允许用户写入任意命名的文件，所以它可能用来改写一些数据文件或操作系统文件。最终用户友好的产品编辑器应检查所传入的文件名称的有效性，或把数据保存到数据库而非文件系统中。

若重新启动服务器并重新加载编辑器，你现在拥有的就应该是一个基本可用的编辑器了，可以使用它来创建和编辑关卡区块数据。不过除了区块数据之外，本章尚未讨论其他一些用来添加和操纵精灵的具体信息，比如玩家和团状怪物的位置等。这会是一些游戏依赖信息，不过基本做法是相同的：增加一个界面来添加和操纵元素，然后提供一个序列化方法把数据保存回服务器中。

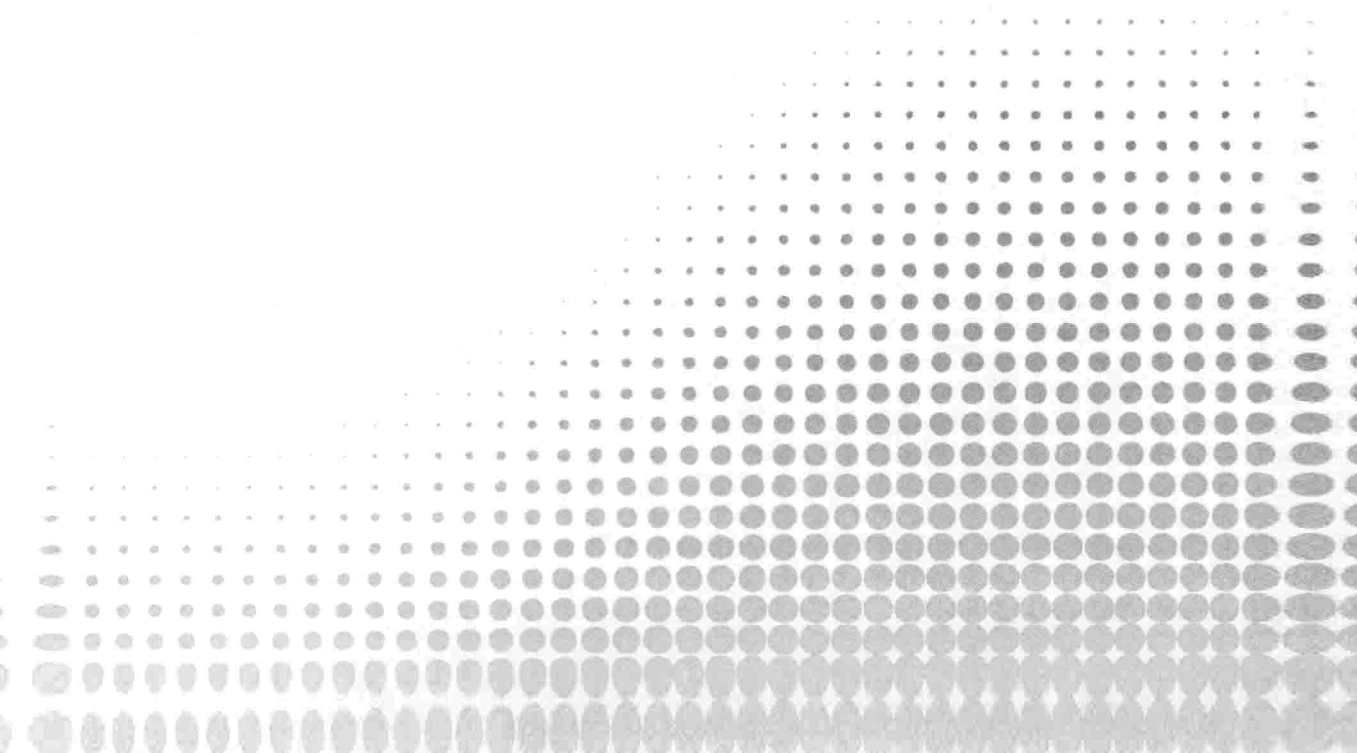
19.5 小结

本章详细阐述了把交互式的游戏区块编辑器添加到上一章的平台动作游戏中的步骤，向你展示了如何添加工具系统，以及如何让用户选择、涂绘和擦除区块。此外，它还增加了把关卡数据保存回服务器中的功能支持，与手工编辑 JSON 数据文件的做法相比，这一功能把创建和编辑关卡变成了一件较容易的事情。在下一章中，你将继续使用 Node.js 为多玩家游戏构建一些更复杂的功能。

第Ⅵ部分

多 人 游 戏

- 第 20 章：构建在线社交游戏
- 第 21 章：实现实时交互
- 第 22 章：构建非传统风格的游戏



第20章

构建在线社交游戏

本章提要

- 了解基于 HTTP 的多玩家游戏
- 使用 Node.js 构建一个简单的社交游戏
- 集成 Facebook
- 连接 NoSQL 数据库
- 部署游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 20 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

20.1 引言

到目前为止，本书构建的游戏提供的都是单玩家体验，但事情并非一定如此：随着 HTML5 驱动的设备访问 Internet 的便利化，任何这样的设备几乎都可做到随时访问 Internet，因此，可将游戏和某个中央数据库连接起来，把它变成一个无论是在桌面 Facebook 中还是移动设备的 Facebook 中都可玩的游戏。本章创建一个简单的、可从 Facebook 内部运行的游戏。

20.2 了解基于 HTTP 的多玩家游戏

由客户端发出整页请求或 Ajax 请求来更新自身，这就是你可构建的最基本的多玩家游戏类型。这种类型的游戏依赖客户端向服务器端发送和请求信息，若说这看起来像是典型的网页，那确实没错。这种做法的缺点是服务器端无法逆转方向从而在有事发生时立刻通知客户端；优点是可以使用标准的 Web 架构和服务器来构建和扩充游戏。

大多数小型网站具有类似的架构：使用脚本语言(PHP、Ruby、Python 或 JavaScript)编写的服务器端代码读写诸如数据库一类的持久层中的数据，除去少量会话数据外，请求之间不存在信息共享，所有信息都被存放在数据库中(或是其他诸如键-值存储一类的持久层中，如 Redis)。保持架构简单就意味着可以继续加入 Web 服务器来处理不断增长的负荷，唯一需要担心出现扩充问题的地方是数据库层。

这种类型的架构在过去几十年中一直很好地服务于网络，只要使用得当，它就是一个非常适合用来构建多玩家游戏的架构。不过，对于需要在不同玩家之间直接进行大量交互的游戏来说，这并不是一种非常好的架构，因为服务器端不能立刻向一个玩家通报另一个玩家正在做的事情。

定期向服务器端请求信息的行为被称为轮询(polling)，因为客户端会定期轮询服务器端以求获得新的信息。不断执行 Ajax 轮询可赋予多玩家游戏一种伪实时的表象，这种情况下，即使是在玩家并未显式地采取行动时，也会有一些事情发生。

就轮询做法而言，最好的多玩家游戏类型是这样的，游戏中每个玩家的行为几乎都是自发的，在多玩家方面，游戏不会涉及太多玩家之间的交互。这包括了这样的一些游戏，在这些游戏中，玩家主要是自己在玩游戏，但有一个中央服务器会在服务器端运行许多游戏逻辑来保证玩家的诚实。在老一辈的社交游戏中，一些游戏，如 Mob Wars，就非常符合此类特点。

第 21 章中讨论的另一种可选做法是使用基于套接口的技术，如 Websocket、Flash 套接口，或是诸如长轮询一类的伪套接口变通方案等，游戏使用这些技术来支持服务器端发起请求，该章内容深入解释了这些术语和概念。

20.3 设计一个简单的社交游戏

了解如何创建基于 HTTP 的多玩家游戏的最好方式就是构建一个。可构建许多复杂的社交游戏，不过，构建一个简单游戏来充当例子，说明如何连接多玩家游戏的各个部件，这应足以帮助你着手去创建一些更复杂的东西。

你将构建的游戏是一个基于 Ian Bogost 的恶搞版社交游戏 Cow Clicker 的简单游戏，正如你所猜想的那样，Cow Clicker 的游戏要点是每隔一定时间单击一次自己的奶牛并由此获得分数。本章构建的是一个类似的游戏，不过做了一点很刺激的改变：你要单击的是团状

怪物而非奶牛。这里所说的团状怪物就是第 18 章所构建的平台动作游戏中的团状怪物敌人。

虽然这一游戏相当简单，但它需要用到典型的基于 HTTP 的多玩家游戏的所有部件：运行游戏的服务器、登录用户的认证系统、存放用户进程的数据库，以及驻留在服务器端、用来控制玩家可采取的行动的游戏逻辑。

这里选用的服务器是一个运行了 Express 框架的简单 Node.js 应用，作为一个社交游戏，它使用 Facebook 作为认证系统，允许用户在不必输入 e-mail 地址或创建密码的情况下登录。为了存储进程，游戏连接了一个名为 MongoDB 的 NoSQL 数据库。服务器会被设置成经过一定时间后才允许玩家再次单击他们的团状怪物，以此来防止他们的积分过快增加。

20.4　集成 Facebook

要创建一个把 Facebook 用于身份验证的游戏，需要创建一个 Facebook 应用来生成一个应用 ID(Application ID)和应用密码(App secret)。

20.4.1　生成 Facebook 应用

为了创建应用，请确保自己已登录到 Facebook 中，然后访问 https://developers.facebook.com，单击页面顶部的 Apps(应用)按钮，然后单击+ Create New App(创建新应用)按钮，输入 Blob Clicker 作为应用的名称，如图 20-1 所示，保留 App Namespace 为空。

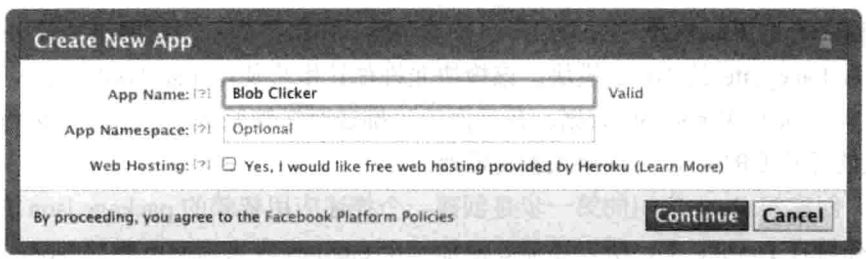


图 20-1　创建一个新应用

若尚未验证账号，那么你还需要通过一个安全检查。

接下来，你会看到 App ID(应用 ID)、App Secret(应用密码)和一个用来配置应用的选项界面。出于测试目的，先通过 localost 运行应用。为设置这一运行方式，在字段 App Domains(应用域)中填入域名称 localhost，接着向下拉动页面，单击 Website with Facebook Login 选项，在字段 Site URL(站点 URL)中输入 URL 地址 http://localhost:3000 /(别忘了最后的斜杠“/”)。图 20-1 展示了这一界面应有的样子。

在页面底部，单击 Save Changes 按钮。不要把自己的应用密码告知其他人(图 20-2 中的密码已被重置)。应用 ID 是用来标识你的应用的公开信息。

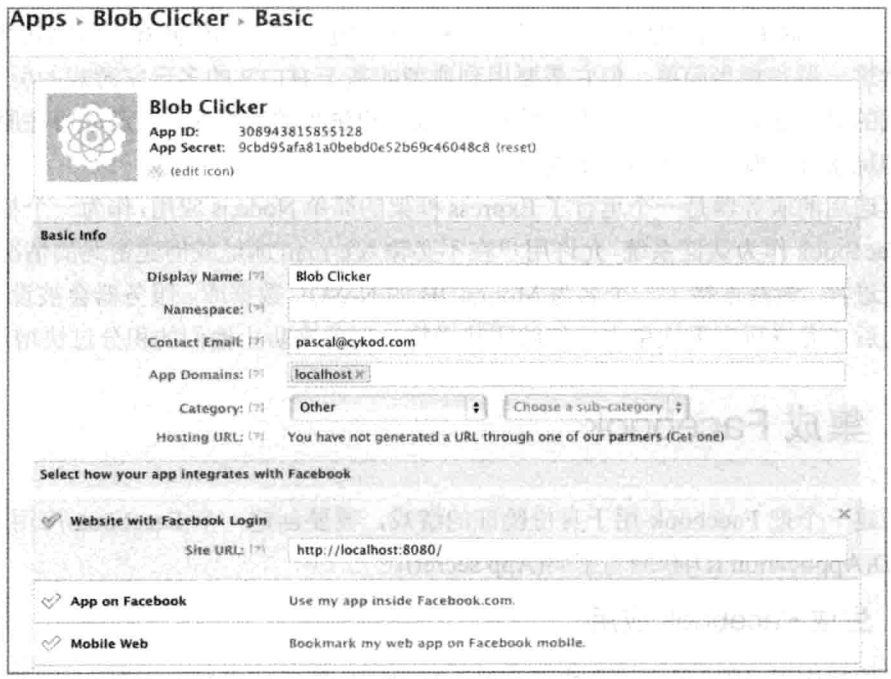


图 20-2 基本应用信息

20.4.2 创建 Node.js 服务器

为创建一个与 Facebook 集成的 Node.js 服务器，你将用到一个由网络托管公司 Heroku 创建的名为 Faceplate 的 Node 模块。该模块允许你让用户使用 Facebook 登录，以及通过 Facebook API 读写 Facebook 数据。在后面的“推送至托管服务”一节中，你将生成一个 Heroku 上的托管 URL，以允许他人参与游戏。

照例，创建 Node.js 应用的第一步是创建一个描述应用依赖的 package.json 文件，因为该应用被托管在 Heroku 上，所以还需要添加一个 engines 段来指明所需的 Node.js 版本。

创建一个名为 blob_clicker 的新目录，然后使用代码清单 20-1 中的内容在该目录下创建一个新的 package.json 文件。

代码清单 20-1：Blob Clicker 的 package.json 文件

```
{
  "name": "blob-clicker",
  "version": "0.0.1",
  "private": true,
  "engines": {
    "node": "0.6.11",
    "npm":  "1.1.1"
  },
  "dependencies": {
    "express": "2.5.8",
    "ejs":      "0.4.3",
```

```
    "faceplate": "0.0.4",
    "mongodb": "1.0.2"
  }
}
```

该游戏的依赖有 Express、前面提到的 Faceplate，以及一个名为 ejs(嵌入式 JavaScript)的模块，这是一个简单的模板系统，支持使用直接内嵌到视图中的服务器端 JavaScript 来创建视图；最后还有 mongodb，这是一个在本章后面用来连接 MongoDB 数据库的模块。

下一步，创建一个可存放服务器基本存根的 web.js 文件，然后输入代码清单 20-2 中的代码。

代码清单 20-2：应用存根 web.js

```
var express = require('express');

var fbId = process.env.FACEBOOK_APP_ID || "YOUR FACEBOOK APP ID",
    fbSecret = process.env.FACEBOOK_SECRET || "YOUR FACEBOOK SECRET",
    sessionSecret = process.env.SESSION_SECRET || "A RANDOM STRING",
    port = process.env.PORT || 3000;

var app = express.createServer(
  express.logger(),
  express.static(__dirname + '/public'),
  express.bodyParser(),
  express.cookieParser(),
  express.session({ secret: sessionSecret }),
  require('faceplate').middleware({
    app_id: fbId,
    secret: fbSecret,
    scope:  'email'
  })
);

app.listen(port);

function login_page(req,res) {
  if(req.facebook.token) {
    req.facebook.me(function(user) {
      req.session.user_id = user.id;
      req.session.user_name = user.name;
      res.redirect('/game');
    });
  } else {
    req.facebook.app(function(app) {
      res.render('login.ejs', {
        layout:    false,
        req:       req,
        app:       app
      });
```

```
    });
  }
}

app.get('/',login_page);
app.post('/',login_page);

function authenticated(method) {
  return function(req,res) {
    if (req.session.user_id) {
      method(req,res);
    } else {
      res.redirect('/');
    }
  }
}
app.get('/game',authenticated(function(req,res) {
  res.end("You are: " + req.session.user_name );
}));
```

这一服务器包含了三大块代码，第一块设置一些默认值，以及把应用设置成一个Express驱动的服务器；第二块处理主页，允许Facebook登录；最后一块，游戏使用authenticated方法和/game也来证实你已登录，然后开始真正的游戏进程。

文件开头的内容需要使用你的Facebook应用ID、Facebook应用密码和一个用来编码会话的随机字符串加以修改。你应修改随机串以防用户修改他们的会话数据，不过也可以保留原样直至部署时再进行修改。代码中的布尔或||运算符用来检查环境变量，以求先从中取得相应的值。Heroku使用环境变量来存放配置数据，这样就更容易为开发和生产环境提供不同的值。

接着是Express应用的创建，该应用设置了服务器所需的各部分内容，包括日志、提供文件的公共目录、请求体和cookie解析器、跟踪登录者的会话，以及用于Facebook的OAuth2身份验证的faceplate中间件等。该应用被告知监听某个特定的端口，这或是默认的3000端口，或是一个由托管服务器设置的端口。

login_page方法用作应用的主页，它的任务是检查用户是否有一个来自Facebook的有效会话令牌，若有则把用户的Facebook ID和名称保存到会话中，然后重定向至/game页面；若没有则渲染login.ejs页面，该页面显示登录按钮以让玩家登录。

login_page既绑定了GET方法也绑定了POST方法，在访问页面时默认使用的是GET方法，不过POST方法会用来把关于用户的数据发送回服务器端。

在画布应用中，若用户已经登录，Facebook直接使用签名请求来连接应用的画布URL。

authenticated方法是一个元方法(meta-method)，它接收一个方法并使用一个查看用户是否已经登录的检查把该方法围括起来，若尚未登录则把用户重定向回主页，以便让他们完成登录。

最后是/game路径，该路径用到authenticated方法，并在该方法中使用用户名向他们

显示一条消息。这仅是一个用来证实身份验证已正确工作的存根，“完成 Blob Clicker 的编写”一节会把游戏逻辑填写到其中。

20.4.3　添加登录视图

基本的 Facebook 集成工作已算完成，还需要做的就是添加视图文件 login.ejs 和一个名为 channel.html 的特殊文件，该文件帮助 Facebook 处理 Internet Explorer 中的跨域问题。

在 blob_ clicker 目录下创建一个新的名为 views 的目录，然后将代码清单 20-3 中的代码放到该目录下一个名为 login.ejs 的文件中。

代码清单 20-3：视图文件 login.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Login to Blob Clicker</title>
    <script src="//code.jquery.com/jquery-1.7.2.min.js"></script>
    <meta name="viewport" content="width=device-width,➥
user-scalable=0, minimum-scale=1.0, maximum-scale=1.0"/>
    <link href='/style.css' rel='stylesheet' type='text/css' />
  </head>
  <body>
    <div id="fb-root"></div>
    <script type="text/javascript">
      window.fbAsyncInit = function() {
        FB.init({
          appId      : '<%= app.id %>',
          channelUrl : '//<%= req.headers['host'] %>/channel.html',
          status     : true,
          xfbml      : true
        });

        FB.Event.subscribe('auth.login', function(response) {
          $("#login").hide();
          $("#signed_request").val(response.authResponse.signedRequest);
          $("#signed_form").submit();
        });
        FB.Canvas.setAutoGrow();
      };

      $(function() {
          window.scrollTo(0,10);
          $("html").on("touchmove",function(e) { e.preventDefault(); });
      });

      // Load the SDK Asynchronously
```

```
    (function(d, s, id) {
      var js, fjs = d.getElementsByTagName(s)[0];
      if (d.getElementById(id)) return;
      js = d.createElement(s); js.id = id;
      js.src = "//connect.facebook.net/en_US/all.js";
      fjs.parentNode.insertBefore(js, fjs);
    }(document, 'script', 'facebook-jssdk'));
  </script>

  <form action='/' id='signed_form' method='post'>
    <input type='hidden' name='signed_request'
          id='signed_request' value=''/>
  </form>
  <div id='title-screen'>
    <div id='login'>
      <div class="fb-login-button" data-scope="email"></div>
    </div>
  </div>
 </body>
</html>
```

该视图文件所包含的无非是一些 JavaScript 脚本，这些脚本设置 Facebook 的 JavaScript SDK、显示一个登录按钮，以及在用户登录时把 signed_request 提交给页面。

该文件引用了一个 style.css 文件，你可在本章的代码中找到该文件及其中用到的三个图像文件：blog.png、title.png 和 interior.png。把文件 style.css 放到 blob_clicker 目录下新建的目录 public/中，并把图像文件放到 public/images/下面。若你不使用这些文件，应用仍然能够运行，只不过看起来像是一些未经样式化的文本。加载了图像和样式的登录界面如图 20-3 所示。

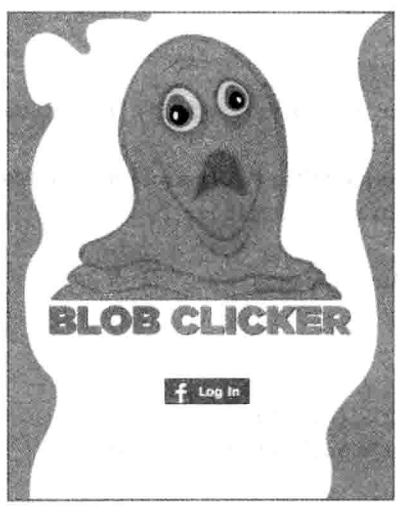


图 20-3　Blob Clicker 的登录界面

在完成 JavaScript 的加载和初始化之后，Facebook 会调用一个名为 fbAsyncInit 的方法，这意味着一旦进入 fbAsyncInit 内部，就可确定 Facebook 的 SDK 是可用的了。为了设置 SDK，你首先需要使用自己创建的应用 ID 来调用 FB.init，这里使用 ejs 标签<%= app.id %>来自动传入该 ID，此外，还需要向 FB.init 传入特殊文件 channel.html 的 URL，这需要用到一个借助 HTTP 请求头的 host 字段生成的绝对 URL，事情因此变得有些复杂。

接着，应用订阅 Facebook SDK 的 auth.login 事件，在用户登录或是已登录时，会触发该事件。为把签名请求(signed-request)传给服务器端，代码用到了一个它要以 POST 方式传送的包含了隐藏字段的表单。服务器端据此取出参数 signed-request，然后执行必需的用户登录逻辑。

此外，FB.init 还接受一个 cookie 选项，该选项使用签名请求来设置会话 cookie，该 cookie 会自动把签名请求传送给服务器端。但 faceplate 中间件会解析每个 HTTP 请求的这一签名请求，这会导致一个单独的到 Facebook API 的服务器端 HTTP 请求，从而减慢游戏的速度。因此，采用只在登录时设置签名请求的做法，游戏可做到更加迅捷。

接下来，加入 Facebook 所提供的用来异步加载 Facebook JavaScript SDK 的代码。

最后，使用正确的样式类来加入一个 标签，把 Facebook 的登录按钮添加到页面上。

```
<div class="fb-login-button" data-scope="email"></div>
```

剩下要做的是，为不支持跨域通信的浏览器创建 channel.html 文件，在应用的下面创建一个名为 public/的目录，然后将代码清单 20-4 中的代码输入该公共目录下名为 channel.html 的文件中。

代码清单 20-4：channel.html

```
<script src="http://connect.facebook.net/en_US/all.js"></script>
```

如前所述，该文件对于 Internet Explorer 和一些旧浏览器来说是必需的，所以，若未正确设置，那么你可能要等到错误报告出来之后才会发现。

20.4.4　测试 Facebook 身份验证

所有部件都已准备就绪，现在应可以测试 Facebook 提供的身份验证了。你可通过输入 node web.js 来运行自己的 Node.js 应用，这会在端口 3000 上启动服务器。现在，你可在桌面浏览器中输入地址 http://localhost:3000 来访问该服务器。

若一切正常运行，你应能看到标题画面和一个很小但可单击的 Facebook 登录按钮。单击该按钮会弹出一个 OAuth 登录窗口，如图 20-4 所示。

图 20-4 Blog Clicker 的 OAuth 登录窗口

单击 Log in with Facebook 按钮，该操作把你带回自己的应用中，它把窗口重定向到 /game 页面上，并显示你的 Facebook 账户名。

20.5 连接数据库

为了记录用户单击团状怪物的次数，以及防止用户过于频繁地单击，需要用到一个存储机制。如前所述，这里选用的机制是很受欢迎的 NoSQL 数据库 MongoDB。

20.5.1 在 Windows 上安装 MongoDB

要安装 MongoDB，可从网站 www.mongodb.org/downloads 下载一个最新版本。

推荐的 Windows 安装方法是这样的，首先需要根据自己运行的是 32 位的还是 64 位的 Windows，下载一个正确的版本；在完成文件的下载之后，把文件夹解压到某个位置上(C:\Program Files\ 就是一个合适的地方)。你可能会希望修改文件夹的名称 mongo-xxxxxxx(其中的 xxxxxxx 是所下载版本的版本号)，那么改成 mongo 即可。

文件夹下载完毕后，需要创建一个数据目录，MongoDB 默认使用目录 C:\data\db，但它不会自动创建这一目录。启动 CMD 终端窗口，运行以下命令在资源管理器中创建目录：

```
C:\> mkdir \data
C:\> mkdir \data\db
```

在创建该文件夹之后，可使用 CD 命令进入 mongo\bin 目录：

```
C:\> cd \Program Files\mongo\bin
```

可在此运行 mongod 命令来启动数据库服务器：

```
C:\Program Files\mongo> mongod
```

至此，服务器已处运行状态(需要保持命令窗口的打开)。欲了解最新的安装指南，请参阅 Mongodb 网站上的 Windows Quickstart 教程：www.mongodb.org/display/DOCS/

Quickstart+Windows。

也许你厌烦了通过命令行来运行 Mongo，那么可以把它设置成以服务方式运行，可以通过在线文档 www.mongodb.org/display/DOCS/Windows+Service 了解更多的使用说明。

20.5.2　在 OS X 上安装 MongoDB

在 OS X 上，若依照第 8 章中的说明，可使用 Homebrew 来安装 MongoDB：

```
$ brew install mongo
```

这一命令应会安装 MongoDB 并启动服务器，若尚未安装 Homebrew，那么需要下载该软件并依照网站上的快速入门指示来安装该软件。

20.5.3　在 Linux 上安装 MongoDB

在 Linux 上，可以使用包管理器来安装最新的版本。在 Ubuntu 或 Debian 上，可运行以下命令：

```
$ apt-get install mongodb
```

该命令应会安装 MongoDB，且会为你启动服务器。

20.5.4　通过命令行连接 MongoDB

若一切按计划进行，那么在 OS X 和 Linux 上，可以在任意目录下运行 Mongo 来加载交互式外壳程序。在 Windows 上，需要在解压目录下的 bin 目录下运行 mongo.exe 命令。

你应会看到交互式外壳程序的界面，且可在其中的>提示符后输入 mongo 命令。MongoDB 外壳程序是一个基于 JavaScript 的外壳，这使得它很适用于本书到目前为止所做的一切(MongoDB 还支持一种基于 JavaScript 的、用于复杂查询的 MapReduce 查询语言，但这要比本章谈及的内容更复杂一些)。

MongoDB 是一个面向文档的存储数据库，这意味着可以把任意文档和数据存储到其中，而不必像在诸如 MySQL、PostgreSQL 或 SQL Server 一类传统的关系数据库系统(RDBMS)中所做的那样，显式地定制数据库的模式。

紧跟在这段内容后面给出的是交互式外壳的一个会话例子，可通过在提示符(>)后输入这些命令来理解其中的内容。这一会话展示了如何切换数据库，以及如何插入和查询数据。与 RDBMS 类似，MongoDB 支持单例安装拥有多个不同数据库。不过，MongDB 中没有表的说法，它把与表等价的结构称为集合(collection)。以下会话先切换到一个名为 blob 的数据库上，接着把几条记录插入到一个名为 clicks 的集合中，然后把这些记录给查询出来。任何时候，你都不需要明确定义数据库 blob 或集合 clicks，可以直接使用它们，MongoDB 会在需要时创建它们。

```
> use blob
switched to db blob
> db.clicks.save({ user: "Tester", clicks: 5 })
```

```
> db.clicks.save({ user: "Tester 2", clicks: 15 })
> db.clicks.save({ user: "Tester 3", clicks: 10 })
> db.clicks.find()
{ "_id" : ObjectId("4fb945ec8137643c2ac40e085e"), ➥
"user" : "Tester", "clicks" : 5 }
{ "_id" : ObjectId("4fb945c28f0137643c2ac40e085f"), ➥
"user" : "Tester 2", "clicks" : 15 }
{ "_id" : ObjectId("4fb946508137643c2ac40e0860"), ➥
"user" : "Tester 3", "clicks" : 10 }
>
> db.clicks.find({user:"Tester"})
{ "_id" : ObjectId("4fb945ec8137643c2ac40e085e"), ➥
"user" : "Tester", "clicks" : 5 }
>
> db.clicks.findOne({user:"Tester 2"})
{
      "_id" : ObjectId("4fb945c28f0137643c2ac40e085f"),
      "user" : "Tester 2",
      "clicks" : 15
}
>
> db.clicks.find().sort({ clicks: -1 }).limit(2)
{ "_id" : ObjectId("4fb945c28f0137643c2ac40e085f"), ➥
"user" : "Tester 2", "clicks" : 15 }
{ "_id" : ObjectId("4fb946508137643c2ac40e0860"), ➥
"user" : "Tester 3", "clicks" : 10 }
> db.clicks.drop();
true
> exit
bye
```

在 MongoDB 中创建记录很简单，只需使用 JavaScript 对象中的模型数据来调用 db.collectionName.save ({..})即可。

集合的查询通过调用 db.collectionName.find()方法实现，该方法返回所有对象；另外也可使用你想要匹配的 JavaScript 对象属性来调用该方法，例如，为了查找所有其 user 属性被设置成 Tester 的用户，上述例子用到了以下命令：

```
db.clicks.find({user:"Tester"})
```

若只查找一个对象，那么可使用 db.collectionName.findOne ({..})。此外，如你所料，你还可以通过调用 sort 和 limit 来排序和限定结果。最后是 db.collectionName.drop()，该方法可销毁整个集合。

注意：就能使用 MongDB 来做的事情而言，这一例子只不过触及皮毛。可在 MongoDB 的官方使用手册 http://docs.mongodb.org/manual/中找到关于它的更多信息。

MongoDB 提供了很丰富的接口来查询文档(包括深入查询嵌套文档)，不过，就本章的 Blob Clicker 游戏而言，这里给出的一些基本用法已能满足你的需求。

20.5.5　将 MongoDB 集成到游戏

因为 Node.js 以异步方式工作，所以，与诸如数据库一类的任何外部资源交互往往都会用到大量的回调，因而导致重度的嵌套，而这有可能令人难以跟踪应用的逻辑。

在大型应用中，解决这一问题的方案是使用诸如约定(promise)模式的技术，这在第 8 章中已介绍过。在 Blob Clicker 这个例子中，另一种解决方案是把数据库代码和其他代码分隔开来，这样更便于跟踪实际的页面流程。

为了把数据库支持加入到游戏中，用代码清单 20-5 中的代码替换掉 web.js 文件末尾处的哑命令 app.get ("/game"...)。这段代码设置三个数据库方法——fetchUser、clickUser 和 topTen——之后这些方法会被应用的路由使用。

代码清单 20-5：Blob Clicker 数据库和路由代码

```
var clickTime = 5000,
    dbMethods = {};

require("mongodb").connect(process.env.MONGOHQ_URL ||
                           "mongodb://localhost/blob_clicker",
                           {}, function(error,db) {
  db.collection('users', function(err, collection){
    dbMethods.fetchUser = function(session,callback) {
      collection.findOne({ user_id: session.user_id },
                         function(error,user) {
        if(!user) {
          user = {
            user_id: session.user_id,
            name: session.user_name,
            clicks: 0,
            next_click: new Date().getTime()
          }
        }
        callback(user);
      });
    };

    dbMethods.clickUser = function(user, callback) {
      var now = new Date().getTime();
      if(user.next_click <= now) {
        user.clicks += 1;
        user.next_click = now + clickTime;
        collection.save(user, function() { callback(user); });
      } else {
        callback(false);
      }
```

```
    };

    dbMethods.topTen = function(callback) {
      collection.find().sort({ clicks: -1 })
                       .limit(10)
                       .toArray(function(error,results) {
        var output = [];
        for(var i in results) {
          output.push([ results[i].name, results[i].clicks ]);
        }
        callback(output);
      });
    };
  });
});

app.get('/game',authenticated(function(req,res) {
  dbMethods.fetchUser(req.session,function(user) {
    var now = new Date().getTime(),
        nextClick = (user.next_click - now)/1000;
    res.render('game.ejs', {
        layout:    false,
        req:       req,
        user:      user,
        nextClick: nextClick
       });
});
}));

app.post("/click",authenticated(function(req,res) {
  dbMethods.fetchUser(req.session,function(user) {
    dbMethods.clickUser(user,function(clicked) {
      if(clicked) {
        var now = new Date().getTime(),
            nextClick = (user.next_click - now)/1000;
        res.json({ clicked: true, user: clicked, nextClick: nextClick });
      } else {
        res.json({ clicked: false })
      }
    });
  });
}));

app.get('/top-ten',authenticated(function(req,res) {
  dbMethods.topTen(function(results) {
    res.json({ users: results });
  });
}));
```

开始处的 var 声明设置 clickTime 变量，该变量控制玩家在单击之前需要等待的时间，

以及一个 dbMethods 对象，该对象将会被填入数据库方法。

接着，应用加入 mongodb 驱动程序并连接数据库；同样，若存在可用的环境变量就使用该变量，否则使用本地数据库 blob_clicker。

接下来，系统连接 users 集合并定义了三个数据库方法：fetchUser、clickUser 和 topTen。第一个方法使用 collection.findOne 来通过用户的 Facebook ID 查找用户，若有用户被找到，方法返回该用户；否则，方法设置一个可代用的新对象。之后，该用户对象被传给回调。系统依赖于用户的 Facebook ID，以此作为用来查找用户的唯一标识。

第二个方法 clickUser 接收一个用户对象并查看该对象是否可再次单击，若是，它更新单击次数和下一次的单击时间并保存该对象，在完成保存之后，它把该对象返回给回调；若还不能单击，则使用 false 值来调用回调。

第三个方法 topTen 使用 find、sort 和 limit 方法来返回排名前十的单击者列表，并按照 clicks 字段的降序排序该列表。MongoDB 的 collection 方法支持多个查询方法的链式调用，最后一个 toArray 调用返回结果数组。

勿忘使用索引

前面给出的代码并未创建任何被称作集合索引的结构，虽然在用户量不大时这一代码运行得还算可以，但这种做法意味着 MongoDB 需要遍历集合中的每个文档，通过用户的 ID 来查找用户。为了提高这一查询和其他查询的速度，需要告知 MongoDB 如何为集合创建索引，这与 RDBMS 中的做法非常类似。通过 mongo 外壳程序，可运行以下命令：

```
db.users.ensureIndex({user_id:1});
```

该命令确保 users 集合的 user_id 字段拥有一个索引，可参阅 www.mongodb.org/display/DOCS/Indexes 了解更多细节。

紧跟在数据库方法之后的是三个游戏路由的定义，第一个路由/game 将响应包装在前面提到的 authenticated 方法中，以此来确保用户已通过身份验证，然后，它从数据库中提取用户对象，计算下一次的单击时间并渲染 game.ejs 视图。

click 方法是应用以 POST 方式提交用户单击的地方，该方法先从数据库中提取用户对象，然后尝试单击用户。若单击成功则返回 clicked:true、更新后的用户对象(其中包含了单击次数)和重新计算的下一次单击时间；否则返回 clicked:false。该方法使用 Express 的 res.json 为客户端返回 JSON 格式的数据，jQuery 能够轻松解析和处理这一格式。

最后是 top-ten 方法，该方法仅是以 JSON 格式返回排名前十的单击者数据。

20.6　完成 Blob Clicker 的编写

为了完成 Blob Clicker 游戏的编写，最后还需要一个 game.ejs 文件，该文件包含了游戏的代码。因为这个游戏例子极其简单，所以除了加入 Quintus 引擎外，代码仅是使用几

个 jQuery 调用来更新页面。

将代码清单 20-6 中的代码添加到 views/目录下一个名为 game.ejs 的新文件中。

代码清单 20-6：view/game.ejs 文件

```
<!DOCTYPE html>
<html>
  <head>
    <title>Blob Clicker</title>
    <script src="//code.jquery.com/jquery-1.7.2.min.js"></script>
    <link href='/style.css' rel='stylesheet' type='text/css' />
    <meta name="viewport" content="width=device-width, ➥
user-scalable=0, minimum-scale=1.0, maximum-scale=1.0"/>
  </head>
  <body>
    <div id="fb-root"></div>
    <div id="main-screen">
      <h1><%= user.name %></h1>
      <div id='blob'>Click Me</div>
      <div id='clicks'><%= user.clicks %></div>
      <div id='show-top-ten'>See Top Ten</div>
      <div id='hide-top-ten'>Back to game</div>
      <ol id='top-ten'></ol>
    </div>
    <script>
      $(function() {
        var nextClick = <%= nextClick %>,
            clickTimer = null;

        function updateNextClick() {
          $("#blob").text(nextClick >= 0 ?
                          Math.ceil(nextClick) + " seconds" :
                          "Click Now");
        }

        function setClickTimer() {
          clearInterval(clickTimer);
          clickTimer = setInterval(function() {
            nextClick--;
            updateNextClick();
          },1000);
        }

        updateNextClick();
        setClickTimer();

       $("#blob").on("click",function() {
$.post("/click",function(data) {
          if(data.clicked) {
            $("#clicks").text(data.user.clicks);
```

```
            nextClick = data.nextClick;
            updateNextClick();
            setClickTimer();
          }
        });
      });

      $("#show-top-ten").on("click",function() {
        $("#show-top-ten,#blob").hide();
        $.get("/top-ten",function(data) {
          $("#hide-top-ten,#top-ten").show();
          $("#top-ten").empty();
          $(data.users).each(function(idx) {
            $("#top-ten").append("<li> " + this[0] + ": " + this[1]);
          });
        });
      });

      $("#hide-top-ten").on("click",function() {
        $("#show-top-ten, #blob").show();
        $("#hide-top-ten, #top-ten").hide();
      });

      window.scrollTo(0,10);
      $("html").on("touchmove",function(e) { e.preventDefault(); });
    });

</script>
</body>
</html>
```

为简洁起见，这一 80 行的文件包含了游戏客户端的所有部件(样式除外)，如你之前所见，较大的游戏会分成多个单独的 JS 文件存放。

该文件一开始先设置一些包含了游戏可视部件的 HTML 元素，同样，若是载入一些样式和图像，游戏的外观会更吸引人一些；否则就只能看到一些文本。图 20-5 展示的是最终实现的游戏画面。

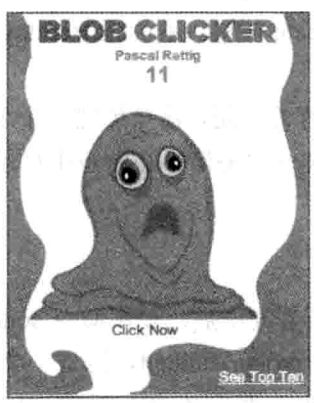


图 20-5　最终的 Blob Clicker 游戏画面

游戏的第一块代码定义了一些方法来更新倒数计时器，以让用户知道他们何时可以再次单击团状怪物。这部分代码用到了 setInterval，该方法虽然有损于动画，但很适用于倒数计时器，因为它会每秒自动触发一次。

接下来是单击处理程序，该程序处理实际的团状怪物单击操作，它把一个 POST 请求发送给/click，若用户抢先单击了怪物，那么回送的响应是一个 JSON 对象{ clicked : false }；若用户在适当时单击了怪物，则服务器端以总单击数和下一次单击时间为详细信息作答。

然后单击页面底部的 top-ten 链接，这一操作从服务器端拉取排名前十的用户列表的 JSON 数据，然后创建一些列表项来显示列表中的内容。

最后是隐藏前十列表，这一操作在页面上切换显示团状怪物和从服务器端拿到的前十列表。

在构建完 game.ejs 后，可以本地运行应用、登录和单击团状怪物。

20.7 推送至托管服务

独自一人玩 Facebook 游戏没有多大乐趣，所以，为了让其他玩家参与该游戏，你应把它推送到一个托管服务上。幸而，Facebook 提供了非常简单的做法，可让你轻松把游戏部署到托管服务上。

首先访问 Facebook 应用的 Basic 设置页面，然后在 Hosting URL 一行中单击 Get One 链接，如图 20-6 所示。

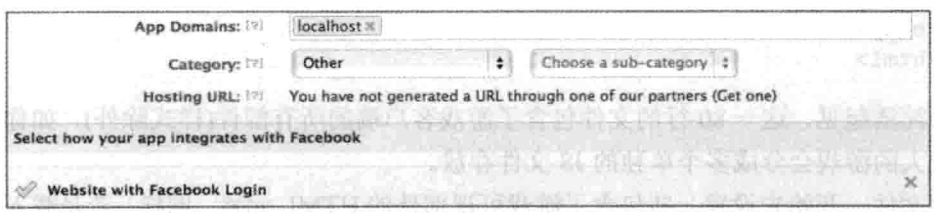


图 20-6 设置一个托管 URL

单击这一链接使你能在 Heroku 托管服务上创建一个应用，单击 Next 按钮，然后选择 Node.js 作为环境，然后单击 Create 按钮。若尚无 Heroku 托管账号，需要按照提示来设置一个。

为了告知 Heroku 你想要运行的应用，需要创建一个名为 Procfile 的文件，该文件告诉平台运行什么命令来启动你的 Web 服务器。在游戏主目录下创建 Procfile 文件，把下一行内容放入其中：

```
web: node web.js
```

该行内容告诉 Heroku，在部署完毕后，使用命令 node web.js 来运行 Web 服务器。

要使用 Heroku，需要安装 Heroku Toolbelt，这是一个具有命令行界面的程序，可让你使用 Git(该工具也会随同 Toolbelt 一起安装)把应用推送到 Heroku 上。要安装 Toolbelt，访

问 https://toolbelt.herokuapp.com/，从上面下载适合所用平台(这应可预先选择)的包并进行安装。

在完成 Toolbelt 的安装后，需要登录到 Heroku 中。打开一个外壳提示符窗口，运行以下命令：

```
heroku login
```

输入用来登录 Heroku 的电子邮件地址和密码，此外，若没有 SSH 密钥，那么命令可能会提示你创建一个新的 SSH 密钥(若不知道 SSH 密钥为何也勿担心，按照提示去做就是了)。

在登录后，需要创建一个新的 Git 存储库、提交应用、添加一个远程存储库、添加 MongoDB 支持，然后推送该存储库。虽然这看起来有些复杂，但每步只用到一个命令，因此很快就可以完成。

完成这些工作需要用到的唯一信息是你刚创建的应用的名称，该名称有着类似 severe-mountain-1301 这样的格式(Heroku 以形容词-名称-数字这一格式来创建名称)。

在你的游戏所在目录下，输入以下命令行中的命令，使用 Heroku 自动创建的应用名称(可在 Heroku 账户信息中找到)来替换 severe-mountain-1301：

```
git init
git commit . -m "Initial Commit"
git remote add heroku git@heroku.com:severe-mountain-1301.git
heroku addons:add mongohq:free
git push -f heroku master
```

最后一行命令需要花费一点时间，因为它会把你的游戏发送到云端部署，其中的-f 标志代表 *force*，之所以使用这一标志是因为 Facebook 创建了一个你应要覆盖的默认应用，不过自此之后，大部分情况下，你都不会再用到-f 标志了。

关于 Git 版本控制

Git 是一个非常流行的开源版本控制系统，若尚未用过 Git，那么最好从此处 http://git-scm.com/book 入手。你可能曾访问过 http//github.com，这是一个面向开源项目的免费 Git 托管服务。Heroku 使用 Git 来处理部署，这样就很容易做到与工作流集成，但可能会给 Git 新手带来一些混淆。

接下来，你必须更新 Facebook 应用的一些细节，以便与托管应用的 URL 保持一致。回到 Facebook 应用的 Basic 设置页面中，把 App Domains 和 Site URL 修改成应用的 URL。单击打开 App on Facebook 和 Mobile Web，其中应已预先填入了托管 URL，如图 20-7 所示。

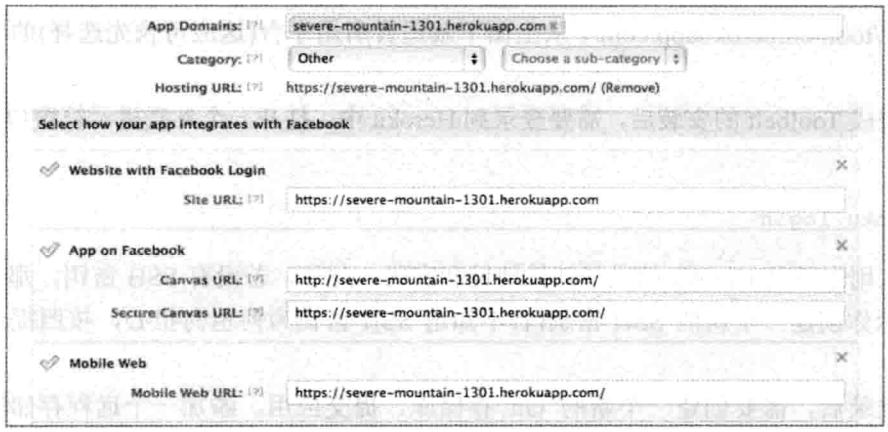


图 20-7 在 Facebook 应用中设置托管 URL

在完成了应用部署和 Facebook 应用更新后，现在可以通过 Site URL 以及在 Facebook 画布中展示 Blob Clicker 的光辉形象了。若要查看 Facebook 画布 URL，单击页面顶部的 Apps，该操作返回应用的摘要页面，可从中找出在 Facebook 内部运行游戏的 Canvas Page URL。

20.8 小结

本章讲述了如何创建一个简单的社交游戏并把它部署到网络上。虽然这个游戏——Blob Clicker——并不具有革新性，但可以使用实现它所需的一些功能部件，比如说使用 Facebook 身份验证、数据库连接和网络部署来构建一个功能齐全的社交游戏。

第21章

实现实时交互

本章提要

- 了解 WebSocket
- 使用 Socket.io 创建一个支持 WebSocket 的服务器
- 构建一个实时的多玩家游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 21 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

21.1 引言

如上一章所示，可使用一种标准的 HTTP 架构来构建多玩家社交游戏。不过，对于所能构建的游戏类型来说，在不采用各种黑客手段的情况下，若想让服务器向客户端推送数据，那么这种架构会存在一些局限性。WebSocket 以原生方式给浏览器植入了一种基于套接口的实时双向对话机制，为这一问题提供了一种解决方案。本章探讨使用一个名为 Socket.io 的 Node.js 库来构建实时游戏，该库支持 WebSocket 和一些回退机制。

21.2 了解 WebSocket

WebSocket 提供了一个浏览器原生 API，该 API 支持通过创建到服务器端的套接口

(socket)连接来提供实时、双向的信道，该信道可用来在客户端到服务器端以及服务器端到客户端这两个方向上传递消息。

在网络编程中，TCP 套接口是一个常见概念；网络上的所有 HTTP 通信都是经由套接口传输的。从 HTML5 游戏的角度看，其过程就是浏览器打开一个到服务器端的套接口、发起一个 HTTP 资源请求、等待资源完成下载，之后再关闭套接口。在关闭套接口后，发送其他任意数据请求都会再次打开一个新的套接口。此外，若服务器有一些需要告知客户端的内容，那么它需要等待，直到客户端请求新的资源时它才能发送数据。

在 WebSocket 之前，一种被大量使用的解决方案是长轮询。长轮询(long polling)意指客户端打开一个到服务器端的请求，服务器在不能写入数据时会一直保持请求的打开，直到有信息告知客户端为止。在有了一些需要发往客户端的数据后，它把数据写入套接口，然后关闭它，把它当成一次正常的请求。另一方面，客户端处理从服务器端发送过来的数据，然后立即打开一个新请求以等待更多数据到来。这一机制使得服务器端能够向客户端发送数据；不过，为两个方向上推送的每块数据都创建一个新的套件口，由此产生的开销会带来性能方面的损失。

Flash 提供 socket 支持已有些年头，所以，另一种变通做法是通过已加载的 SWF 来使用 Flash 套接口，SWF 通过 Flash 到 JavaScript 的通信公开了一个接口。不过，Flash 套接口的一个问题是，为了也在同一服务器端提供普通的 HTTP 请求，它们需要存在于其他端口而非正常的 HTTP 80 端口和 HTTPS 443 端口之上；所以，基于这些特定端口构建起来的互联网基础设施(防火墙、代理等)需要更新以支持所有的 Web 浏览器和服务器，其中的许多浏览器和服务器可能位于家庭或公司的防火墙后面，而这些防火墙会限制对非标准端口的访问。

在 2009 年，作为浏览器缺乏固有套接口及端口问题的一个解决方案，WebSocket 规范(www.w3.org/TR/websockets)被正式提出。WebSocket 背后的想法是，使用一种服务器端和客户端双方都需要理解的握手技术把标准的 HTTP 套接口升级成 WebSocket。之后，该套接口保持打开，允许在客户端和服务器端之间进行双向的全双工通信。

HTML5 应用的 WebSocket 使用能力发展得有些缓慢，其主要原因在于规范有一个演进过程，这意味着不同浏览器会支持不同的规范版本，而且，一些与代理的缓存定位相关的安全性问题曾导致 WebSocket 支持被从 Firefox 中删除，直至该问题被修正。

好消息是，该问题现已被修正，除 IE9 之外，所有目前这一代浏览器都已启用了 WebSocket 的某个版本。坏消息是，因为代理、缓存和 IE9，你还不能在没有提供某种回退支持的情况下使用标准的 WebSocket。出于这一原因，作为直接使用 WebSocket 的一种替代，本章会占用大部分篇幅来介绍一个名为 Socket.io 的 Node.js 库，不管是原生 WebSocket 还是某一受支持的回退机制获得了支持，该库都会提供一个一致的客户端和服务器端 API。

21.3　在浏览器中使用原生 WebSocket

受支持的浏览器所提供的原生 WebSocket API 小巧简洁，但除了收发服务器端的文本之外，它并未做太多事情，所以，额外的兼容性和回退问题便把这一 API 的直接使用变成了一件烦人的事情。

假设你有一个支持 WebSocket 的服务器端，那么借助于以下代码，可以在浏览器中使用一个新的 WebSocket 对象来打开一个连接：

```
var socket = new WebSocket("ws://servername.com/socket-resource");
```

与 http://这一 URL 前缀等价的 WebSocket 前缀是 ws://，安全 WebSocket 则有一个与 https://等价的 wss://前缀。

该套接口对象有四个用来监听套接口事件的回调：

```
socket.onopen = function(){
  // Socket has been opened
};
socket.onmessage = function(event) {
  // Message data in event.data
};
socket.onclose = function() {
  // WebSocket has been closed
};
socket.onerror = function(event) {
  // Error triggered
};
```

要通过套接口发送数据，调用 socket.send：

```
socket.send (message);
```

该方法把消息串发送给服务器端。有一件需要记住的与 WebSocket 有关的事情是，所有来回发送的数据都采用字符串格式，所以，使用某种机制来编码和解码这些串的工作就要靠你自己来完成了(JSON 是一种公认的且很受欢迎的选择)。

websocket.org 网站没有仅是简单设置一个服务器来测试原生的 WebSocket，相反，它供了一个回显服务器，可以使用它来测试自己编写的客户端 WebSocket 代码。

创建一个新的名为 echo.html 的文件，将代码清单 21-1 中的代码加入其中。

代码清单 21-1：回显服务器的一个简单客户端

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="http://ajax.googleapis.com/➥
```

```
ajax/libs/jquery/1.7.2/jquery.min.js" ></script>
  <title>WebSocket Test</title>
</head>
<body>
<script>
  var echoURI = "ws://echo.websocket.org/";
  var socket;

  $(function() {
    socket = new WebSocket(echoURI);
    socket.onopen = function() {
      $("#output").append("<div>WebSocket Opened</div>");
    };

    socket.onclose = function() {
      $("#output").append("<div>WebSocket Opened</div>");
    };

    socket.onmessage = function(event) {
      $("#output").append("<div>WebSocket Message:" +
                          event.data + "</div>");
    };

    socket.onerror = function() {
      $("#output").append("<div>WebSocket Error</div>");
    }
    $("#send").on("click",function() {
      var value = $("#message").val();
      $("#output").append("<div>Sending: " + value + "</div>");
      socket.send(value);
      $("#message").val("");
    });
  });

</script>
<input type='text' id='message'/>
<button id='send'>Send</button>
<div id="output"></div>
</body>
</html>
```

在一个支持 WebSocket 的浏览器中加载该文件，之后可向服务器端发送消息，服务器会马上向你回显消息。

这段代码使用回显服务器的 URL 设置了一个简单套接口，然后添加一些回调，只要四个基本事件之一被触发，这些回调就会把一条消息添加到 ID 为 output 的 中。

此外，它还给 Send 按钮添加了一个单击事件处理程序，目的是发送输入的任意信息。因为它所连接的服务器是一个回显服务器，所以任何被发送的消息都应会触发一条返回消息。

上述代码涵盖了客户端 WebSocket 的使用，在服务器端，需要一个库来处理生存期较长的请求，这意味着使用标准的 PHP 或 Ruby on Rails 框架之类的做法行不通。幸而，你很熟悉的 Node 对于处理大量并发连接很在行。

如引言一节所述，本章会介绍一个 WebSocket 之上的名为 Socket.io 的服务器端抽象。若希望直接使用 WebSocket，那么可以了解一下 Gihub 上提供的 Node 模块 ws：https://github.com/einaros/ws。

该模块使得你能够使用几行代码在 Node.js 中创建一个 WebSocket 服务器，其语法与浏览器端的 WebSocket API 类似。

21.4　使用 Socket.io：支持回退的 WebSocket

若希望创建一个实时游戏而又无需受浏览器兼容性和回退这类烦心事的困扰，那么有许多库可提供帮助，不过其中一个最受欢迎、用法最简单的库是 http://socket.io 上提供的 Socket.io。

Socket.io 是一个抽象了客户端和服务器端的 WebSocket 及多种受支持回退的 Node 库，它还提供了通过套接口透明地发送 JSON 数据的能力，还添加了对任意数量的自定义事件的支持。此外，Socket.io 还可很好地集成到 Express 中，这意味着可以使用单个应用来提供 HTTP 方法、WebSocket 和静态文件。加之对心跳、超时和断开连接的支持，显而易见，使用构建在 WebSocket 之上的库(而非直接使用 WebSocket)能把编程工作变得容易许多。

为在使用 Socket.io 构建游戏之前先熟悉一下该库，你将构建一个简单的多用户涂鸦应用，该应用支持大家以一种实时方式涂改彼此的画作。

21.4.1　创建涂鸦应用的服务器端

在服务器端，Socket.io 通过监听 connection 事件开展工作。这些事件使用一个 socket 对象触发回调；然后，可以附上其他一些监听器来监听诸如 disconnect 一类的标准事件和自定义名称的事件。

要发送数据，可使用事件的名称和任何需要随之传送的数据来调用 socket.emit，可通过调用 socket.broadcast.emit 把事件发送给除 socket 自身之外的其他所有套接口。

为了创建涂鸦应用，首先要创建一个说明依赖的 package.json 文件。创建一个新的名为 scribble 的项目目录，然后将代码清单 21-2 中的 package.json 文件添加到其中。

代码清单 21-2：涂鸦应用的 package.json

```
{
   "name": "scribbler"
 , "version": "0.0.1"
 , "private": true
 , "dependencies": {
    "express": "2.5.8",
```

```
    "socket.io": "0.9.6"
  }
}
```

现在，通过命令行在该目录下运行 npm install 来抓取依赖——Socket.io 有几个。

接下来创建 app.js 文件，然后将代码清单 21-3 中的代码加入其中。

代码清单 21-3：涂鸦应用的 app.js

```
var express = require('express'),
    app = express.createServer(),
    io = require('socket.io').listen(app);

app.configure(function(){
  app.use(express.static(__dirname + '/public'));
});

app.listen(3000);

// Clear the board every 60 seconds
setInterval(function() {
  io.sockets.emit('clear');
},60000);

io.sockets.on('connection', function (socket) {
  socket.on('paint',function(data) {
    socket.broadcast.emit('paint', data);
  });
  socket.on('disconnect', function () {
   console.log("Someone disconnected");
  });
});
```

如代码清单 20-3 所示，设置和运行 Socket.io 服务器所需的代码可谓非常之少。在创建 Express 服务器之后，为了把 Socket.io 附加到该服务器上，你只需调用 listen：

```
io = require('socket.io').listen(app);
```

余下代码设置 Express、配置静态目录以及绑定端口，这都与你之前见到的做法是一样的。

因为有一大堆的随机用户，涂鸦内容很有能会变得杂乱不堪，所以服务器毫不客气，每隔 60 秒钟就会清除一次画板，其做法是发送一个 clear 消息告诉客户端清除它们的涂鸦区域。若目标是所有套接口，那么可调用 io.socket.emit，该方法接收一个事件名称和一个可选的数据对象。

为响应新连接进来的套接口，需要把 connecttion 事件绑定到套接口列表 io.sockets 上，该事件使用这一客户端连接的 socket 对象触发它的回调。

可以把该套接口对象存放起来以供后面引用，也可以把其他一些事件绑定到该对象

上。在这个例子中，被绑定到客户端的 paint 事件在客户端每绘制一行时都会被触发，服务器端通过调用 socket.broadcast.emit 方法来响应该事件，该方法的作用与 io.sockets.emit 类似，除了一点，那就是它会跳过那个调用自己的套接口对象。

21.4.2　添加涂鸦应用的客户端

要完成这一涂鸦应用，还需要添加应用的客户端。该应用使用 Quintus 主要是为了省去设置及最大化画布的步骤。在应用的下面创建一个 public/子文件夹，然后使用代码清单 21-4 中的内容创建一个 index.html 文件。

代码清单 21-4：涂鸦应用的 index.html 文件

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0, ➥
 minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Scribble</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='scribble.js'></script>
    <script src="/socket.io/socket.io.js"></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

需要在 public/下创建一个 public/js/子目录，把前面代码中列出的三个依赖：jquery.min.js、underscore.js 和 quintus.js 放在该目录中，可以在本章的代码中找到这些文件。

需要注意的是，这段代码的结尾处有一个特殊的脚本标签：

```
<script src="/socket.io/socket.io.js"></script>
```

这是一个由 Socket.io 创建的路径，它提供了一些便利性。首先，它设置默认与被请求文件相同的 WebSocket 路径和服务器，这可以简化连接的建立，而且还可以防止出现任何需要在开发和生产环境之间选择正确地址的问题。此外，它还自动确定使用哪一种传输机制：是直接使用 WebSocket 还是使用回退之一。

接下来，在 public/目录中创建代码清单 21-4 中用到的 scribble.js 文件，将代码清单 21-5 中的代码加入其中。

代码清单 21-5：scribble.js

```
$(function() {
  var Q = Quintus().setup('quintus', { maximize: true }),
      socket = io.connect(),
      start = {},
      move = {};
  function getTouch(e) {
    var touch = e.originalEvent.changedTouches ?
                e.originalEvent.changedTouches[0] : e,
        canvasPos = Q.el.offset(),
        canvasX = (touch.pageX - canvasPos.left) / Q.el.width() * Q.width,
        canvasY = (touch.pageY - canvasPos.top) / Q.el.height() * Q.height;
    e.preventDefault();
    return { x: canvasX, y: canvasY };
  }

  function drawLine(from,to) {
    Q.ctx.strokeStyle= "#000";
    Q.ctx.beginPath();
    Q.ctx.moveTo(from.x,from.y);
    Q.ctx.lineTo(to.x,to.y);
    Q.ctx.stroke();
  }

  Q.el.on('touchstart mousedown',function(e) {
    start = getTouch(e);
  });

  Q.el.on('touchmove mousemove',function(e) {
    if(!start.x) return;
    move = getTouch(e);
    drawLine(start,move);
    socket.emit("paint",{ start: start, move: move });
    start = move;
  });

  Q.el.on('touchend mouseup mouseleave',function(e) {
    start.x = null;
  });

  socket.on("connect",function() {
    console.log("Connected");
  });

  socket.on("paint",function(data) {
    drawLine(data.start,data.move);
  });

  socket.on("clear",function(data) {
```

```
    Q.ctx.clearRect(0,0,Q.width,Q.height);
  });

});
```

如你所见，这段代码用到了 Quintus，但仅是用来设置画布、调整画布的大小以及把画布变成在 Q.ctx 属性中可用的。

创建到服务器端的套接口连接很简单，只需调用：

```
socket = io.connect();
```

因为放置 socket.io.js 文件的服务器与套接口所连接的是同一台服务器，所以你不必提供要连接的 URI 和端口(若需要连接到不同的服务器上，你也可以提供这些内容)。

接着是两个辅助方法 getTouch 和 drawLine，getTouch 从事件发生位置中提取画布像素位置，drawLine 在画布的两点间绘制的一条新线。

接下来，涂鸦应用定义三个用来跟踪触摸、移动和松开动作的事件处理程序。在发生触摸或鼠标单击时，应用标记线的开始位置，在这一最初触摸之后移动鼠标或移动手指时，应用仅是绘制一条从开始位置到当前位置的直线。

为让其他用户看到你绘制的直线，除了绘制该直线外，应用还会调用 socket.emit 将一个 paint 事件发送回服务器端。如你所见，客户端 API 与服务器端 API 很类似。

最后，若松开手指或鼠标，那么应用会停止绘制。余下三个监听器被绑定到了套接口上，第一个 connect 事件只是向控制台输出套接口已连接这样的日志内容，用在这里是为了说明内置事件和自定义事件的处理方式是相同的。

第一个自定义事件 paint 基于服务器端传送的数据绘制一条直线，不知你是否还记得，服务器端只是转述它所收到的任何 paint 事件，把这些事件发送给其他所有的客户端。

clear 事件则由服务器端每隔 60 秒发送一次，该事件告诉应用清除整个画布。

这一入门例子应足以帮助你着手使用 Socket.io，不过，下一节会继续向你展示如何构建一个简单的乒乓球游戏，在这个游戏中，两个玩家彼此之间可以来回击球。

21.5　用 Socket.io 构建一个多人乒乓球游戏

基于 WebSocket 技术的使用开启了包括实时动作游戏在内的多玩家游戏的许多不同可能性，为了通过实践来了解这一点，你将构建一个两玩家的乒乓球游戏，让玩家跨屏幕来回击球。

在该游戏中，两个玩家在各自的设备上模拟整个游戏，不过，其中一个玩家会充当“主控方”，另一个充当“从属方”。主控方控制球的真实位置，并定期向从属方发送更新，从属方则通过更新球的位置来反映游戏的真实状态。

21.5.1 处理延时

多玩家实时游戏存在的问题之一是延时问题，根据网络速度及服务器和玩家之间的距离不同，超过 100 毫秒的延时和丢包现象在移动设备上都很常见。这意味着，为保证动作的连续性，需要实现某种把延迟考虑在内的预测建模。

乒乓球游戏通过计算一个“延迟”来处理这一问题，这里的延迟指的是游戏从一个玩家处为另一个玩家取得一个数据包所花费的时间。它使用这一延迟来计算，从数据开始离开一个玩家到到达另一玩家处为止的这段时间中，球或另一个玩家应已移动的距离。

因为游戏在两个客户端都得模拟球的路径，所以每个玩家都应会看到球的平滑更新，不过，若一个设备上的球与另外一个设备上的失去同步，那么从属玩家偶尔会看到一个闪现的线路修正。

如你所见，在玩游戏时，这一做法的成功程度取决于浏览器和连接。截至撰写本书之时止，移动设备在 WebSocket 支持方面仍存在一段距离，所以，相比于多玩家动作游戏，半实时的游戏有可能是一种更好的选择。

21.5.2 防止作弊

在多玩家 HTML5 游戏中，只有一种方式可用来防止作弊，即除了用户的输入之外，绝不要相信客户端告知你的任何事情。这意味着服务器端需要模拟整个游戏，然后只接收用户的输入来更新游戏状态。

例如，客户端不会说：“我的球拍的 x 位置是 200，”相反，客户端会告诉服务器：“我希望向右移动球拍，”服务器端会据此适当地更新球拍位置，然后使用更新后的球拍位置来更新所有的客户端。这样做就意味着客户端永远都不能要求服务器端在游戏中做一些非物理上可行的事情，因为游戏只会计算那些有意义的游戏模拟。

举个例子，在乒乓球游戏中，玩家不会突然从屏幕左侧移到屏幕右侧，然而，若服务器端就这么把玩家的球拍位置当成一种事实接受，那么这种情况就有可能发生。

在服务器端进行处理的缺点是，这意味着与客户端承担所有模拟责任这种做法相比，你给处理器带来了更大的负荷。

为了保证事情的简单性，本章中的乒乓球游戏不会在服务器端模拟一切，不过，因为该游戏自始至终都使用 JavaScript，所以你没理由不会在客户端运行与服务器端同样可信的游戏代码。

21.5.3 部署实时应用

遗憾的是，上一章中使用的托管服务器 Heroku 不支持 WebSocket，原因是它们无法通过 Heroku 的各种缓存和代理层。因为 WebSocket 才刚开始慢慢渗透到整个网络中，所以你会在其他一些主机托管平台中遇到同样的问题。Nodejitsu 是一个 Node.js 托管平台，支持原生的 WebSocket。

若希望绕过这一问题，强制使用长轮询来替代 WebSocket(这对性能是一个打击)，那么

可修改服务器端代码来强制使用长轮询，使用以下代码来限制所采用的传输方式(还需要设置最长持续时间为 10 秒来防止 Heroku 超时)：

```
io.configure(function () {
  io.set("transports", ["xhr-polling"]);
  io.set("polling duration", 10);
});
```

另一种选择是把 Node.js 部署到自己的服务器或 VPS(Virtual Private Server，虚拟专用服务器)上，虽然走完一些步骤来实现这一做法已不在本书讨论范围之内，不过，有了亚马逊每月费用 14 美元的微实例，从预算角度来看，这并非遥不可及的事情。

21.5.4　创建自动匹配的服务器端

两玩家多人游戏的一个很重要的环节是，需要两个玩家来玩游戏，这带来了一个匹配问题：如何配对玩家？

正如你可能曾经历过的那样，在多玩家游戏中，玩家匹配可能会变得很麻烦，需要在游戏大厅中显示反应游戏进展情况的统计数据和一些图像，以及一些在技能水平或其他一些特性方面相匹配的玩家。不过，这里的乒乓球游戏不会构建一个需要前端和 UI 的大厅，而是试着创建尽可能多的双玩家游戏，这意味着首先访问站点的玩家会一直等到第二个玩家的到来。若有玩家离开，下一个要加入的玩家就连同剩下的玩家重新加入游戏。

服务器端的另一个主要任务是简单地在玩家之间来回传递消息，保持游戏的同步。此外，服务器端还要把来自一个客户端的 delay 消息弹送给另一个客户端，然后再回送，目的是跟踪两个客户端之间的时间延迟。

创建一个名为 pong 的新目录，接着先创建 package.json 文件，其内容与涂鸦例子 Scribble 目录下的同名文件类似，如代码清单 21-6 所示。

代码清单 21-6：乒乓球游戏的 package.json 文件

```
{
    "name": "multi-player-pong"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "socket.io": "0.9.6"
  }
}
```

接下来使用代码清单 21-7 中的内容在同一目录下创建 app.js 文件。同样，这段代码设置一个 Express 服务器，与涂鸦例子中的做法一样，它把 Socket.io 和应用连接起来，监听套接口连接。

代码清单 21-7：乒乓球游戏的服务器端代码 app.js

```
var express = require('express'),
    app = module.exports = express.createServer(),
    io = require('socket.io').listen(app);
app.configure(function(){
  app.use(express.bodyParser());
  app.use(express.static(__dirname + '/public'));
});

app.listen(3000);

var games = [];

io.sockets.on('connection',function(socket) {
  var game = null;
  for(var i=0;i<games.length;i++) {
    if(games[i].length < 2) {
      game = i;
    }
  }
  if(game === null) {
    games.push([])
    game = games.length-1;
  }
  games[game].push(socket);
  socket.set('game',game);
  if(games[game].length == 2) {
    games[game][0].set('partner',socket);
    games[game][1].set('partner',games[game][0]);
    games[game][0].emit('master');
    games[game][1].emit('slave');
  }

  socket.on('delay',function(data) {
    socket.get('partner',function(err,partner) {
      if(partner) {
        data.steps += 1;
        partner.emit('delay',data);
      }
    });
  });

  socket.on('move',function(data) {
    socket.get('partner',function(err,partner) {
      if(partner) {
        partner.volatile.emit('move',data);
      }
    });
  });
```

```
  socket.on('ball',function(data) {
    socket.get('partner',function(err,partner) {
      if(partner) {
        partner.volatile.emit('ball',data);
      }
    });
  });

  socket.on('disconnect',function() {

    socket.get('partner',function(err,partner) {
      if(partner) {
        partner.emit('end');
        partner.set("partner",null);
      }
    });

    socket.get('game',function(err,game) {
      var idx = games[game].indexOf(socket);
      if(idx!=-1) games[game].splice(idx, 1);
    });
  });
});
```

现在，通过命令行在该目录下运行 npm install，抓取依赖。

在有新的连接进来时，服务器找出 games 数组中的第一个其条目少于两个的条目，该数组存放了多组一起配对玩游戏的玩家。若找不到少于两个玩家的条目，它把一个新的条目添加到数组中，作为新游戏的入口。在这两种情况下，它都会把条目在 games 数组中的索引保存起来，用于跟踪与套接口关联的游戏。

利用 Socket.io 的功能，游戏通过 socket.set 和 socket.get 把一些附加数据和套接口关联起来。

若在把套接口添加到当前游戏中后，当前游戏就集齐了两名玩家，那么服务器会把其中一位设置为主控方(控制球的标准模拟的客户端)，并把另一名设置为从属方。此外，它还把一项命名为 partner(拍档)的数据添加到每个套接口中，用以映射该套接口的配对套接口。

若是把这一例子实现成一个更完整的大厅风格游戏，那么可为每个游戏创建一个用来存放该游戏玩家代码清单的对象，这样很容易就能做到给该游戏中的所有玩家发送消息。

在接收到一个用来测定数据包往返时间的 delay 事件时，服务器端把该事件传递给套接口的拍档，并递增步数。在接收到已倒换了 3 次的 delay 事件后，套接口就知道数据包已经完成了一个完整的往返，所以，它把从一个客户端到另一个客户端的时间估算为总延迟时间的一半。

主要的数据事件 move 和 ball 仅把一个套接口的数据传给它的拍档，两者都以 volatile(可

丢弃)方式发出数据，这意味着数据是时间相关的，在客户端采用长轮询的情况下，若客户端正处在两个请求之间，那么数据可能会被忽略。在数据随事件被频繁传送时，这种事情就会发生；在玩家连续接收到一堆数据包时，相比于让球“快进”，不如丢掉一些包。

最后，在断开连接时，套接口给自己的拍档发送一个事件，然后从游戏列表中删除自身。这意味着若玩家之一还逗留在游戏中，那么下一个要加入的玩家就会获得匹配。

21.5.5　构建乒乓球游戏的前端

两玩家的乒乓球游戏的前端包括了 public/目录下用来加载所有必需依赖的基本 HTML 文件、存放了依赖和 Quintus 引擎的 js/目录，以及游戏文件 pong.js。

基本 HTML 文件如代码清单 21-8 所示，除了加入一个黑色边框来标示桌面上的画布之外，文件内容没有太多意外之处。

代码清单 21-8：乒乓球游戏的 index.html 文件

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
minimum-scale=1.0, maximum-scale=1.0"/>
    <title>Pong</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>
    <script src='pong.js'></script>
    <script src="/socket.io/socket.io.js"></script>
    <style>
      * { padding:0px; margin:0px; }
      canvas { border:1px solid black; }
    </style>
  </head>
  <body>
  </body>
</html>
```

请确保已把 jQuery、underscore 和所有必需的 quintus*.js 文件放到了 public/js 目录下，否则游戏将无法运行。

接下来是 pong.js 文件中的游戏，代码清单 21-9 给出了文件的内容，其中大部分代码应与第 11 章中的击砖游戏看上去类似。如下所示，处理 Socket.io 的那部分代码已被高亮标出(不过，余下代码与 blockkbreak.js 中的内容大不相同，所以不能简单把高亮部分代码加入到之前的文件中)：

代码清单 21-9：pong.js

```
$(function() {
  var Q = window.Q = Quintus()
                  .include('Input,Sprites,Scenes')
                  .setup('quintus');
  var socket = io.connect();
  Q.input.keyboardControls()
  Q.input.touchControls({
          controls:  [ ['left','<' ],[],[],[],['right','>' ]]
        });
  var gameType = null, delay = 0;
  Q.Paddle = Q.Sprite.extend({
    init: function(props) {
      this._super(_(props).defaults({
        w: 60, h: 20,
        speed: 200,
        direction: null
      }));
    },
    step: function(dt) {
      dt += this.p.paddleDelay / 1000;
      this.p.paddleDelay = 0;
      if(this.p.direction == 'left') {
        this.p.x -= this.p.speed * dt;
      } else if(this.p.direction == 'right') {
        this.p.x += this.p.speed * dt;
      }
      if(this.p.x < 0) { this.p.x = 0; }
      if(this.p.x > Q.width - this.p.w) { this.p.x = Q.width - this.p.w; }
    },
    draw: function(ctx) {
      ctx.fillStyle = "black";
      ctx.fillRect(Math.floor(this.p.x),
                   Math.floor(this.p.y),
                   this.p.w,this.p.h);
    }
  });

  Q.PlayerPaddle = Q.Paddle.extend({
    step: function(dt) {
      var lastDirection = this.p.direction;
      this.p.direction = null;
      if(Q.inputs['left']) {
        this.p.direction = 'left';
      } else if(Q.inputs['right']) {
        this.p.direction = 'right';
      }
      this._super(dt);
      if(lastDirection != this.p.direction) {
```

```
      socket.emit("move",[this.p.direction,this.p.x]);
    }
  }
});

Q.EnemyPaddle = Q.Paddle.extend({
  init: function(props) {
    this._super(props);
    var self = this, p = this.p;
    socket.on("move",function(data) {
      p.direction = data[0];
      p.x = data[1];
      self.step(delay/1000);
    });
  }
});

Q.Ball = Q.Sprite.extend({
  init: function(props) {
    this._super(_(props||{}).defaults({
      x: 200, y: 100,
      w: 10,  h: 10,
      dx: -1, dy: -1,
      speed: 100,
      ballRate: 0.5,
      ballSend: 0.5
    }));
    var self = this, p = this.p;
    if(gameType == 'slave') {
      socket.on("ball",function(pos) {
        p.x = pos.x;
        p.y = pos.y;
        p.dx = pos.dx;
        p.dy = pos.dy;
        self.step(delay/1000);
      });
    }
  },
  step: function(dt) {
    var p = this.p;
    var hit = Q.stage().collide(this);
    if(hit) {
      p.dy = hit.p.y < 100 ? 1 : -1;
    }
    p.x += p.dx * p.speed * dt;
    p.y += p.dy * p.speed * dt;
    var maxX = Q.width - p.w;
    if(p.x < 0) { p.x = 0; p.dx = 1; }
    else if(p.x > maxX) { p.dx = -1; p.x = maxX; }
```

```
    if(p.y < 0 || p.y > Q.height) {
      p.x = 200; p.y = 100;
      p.dy *= -1;
    }
    if(gameType == 'master') {
      p.ballSend -= dt;
      if(p.ballSend < 0) {
        socket.emit("ball", { x: p.x, y: p.y, dx: p.dx, dy: p.dy });
        p.ballSend += p.ballRate;
      }
    }
  },
  draw: function(ctx) {
    ctx.fillStyle = "black";
    ctx.beginPath();
    ctx.arc(this.p.x + this.p.w/2,
            this.p.y + this.p.h/2,
            this.p.w/2,0,Math.PI*2);
    ctx.fill();
  }
});

Q.scene('game',new Q.Scene(function(stage) {
  if(gameType == 'master') {
    stage.insert(new Q.PlayerPaddle({ x:0, y: 40}));
    stage.insert(new Q.EnemyPaddle({ x:0, y: Q.height - 100}));
  } else if(gameType == 'slave') {
    stage.insert(new Q.EnemyPaddle({ x:0, y: 40}));
    stage.insert(new Q.PlayerPaddle({ x:0, y: Q.height - 100}));
  }
  stage.insert(new Q.Ball());
}));

socket.on("master",function() {
  gameType = 'master';
  Q.stageScene("game");
});

socket.on("slave",function() {
  gameType = 'slave';
  Q.stageScene("game");
});

socket.on("end",function() {
  Q.clearStage(0);
});

socket.on('delay',function(data) {
  if(data.steps == 3) {
    // delay 1/2 of the round trip time
```

```
      delay = (new Date().getTime() - data.timer)/2;
      if(delay > 50) {
        delay = 50;
      }
    } else {
      data.steps += 1;
      socket.emit('delay',data);
    }
  });

  setInterval(function() {
    socket.emit('delay',{ steps: 0, timer: new Date().getTime() });
  },2000);
});
```

图 21-1 展示的是在两个相邻桌面浏览器中运行的最终游戏的画面。

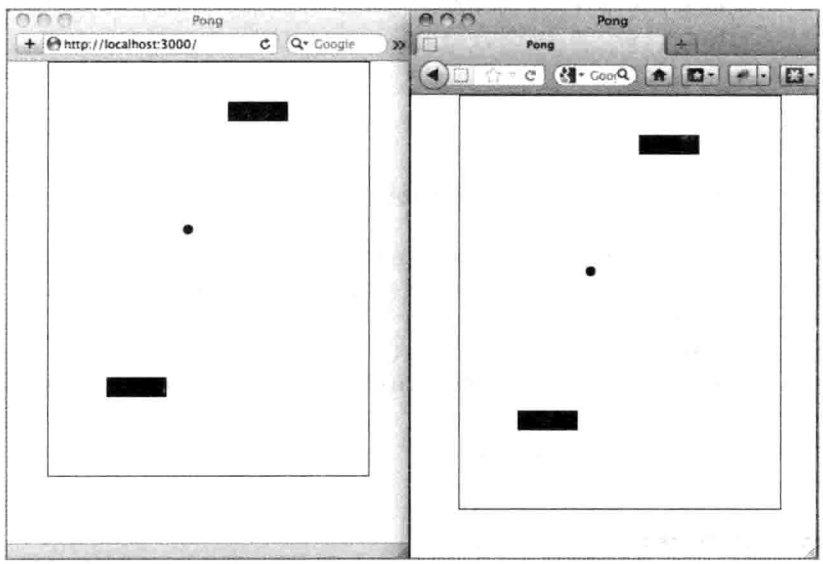


图 21-1 Safari 和 Firefox 之间的乒乓游戏

如前所述，虽然在一个步骤中看到这么多代码，不过其中的大部分应与击砖游戏的代码看上去类似。在一开始的引擎设置代码之后，游戏定义了三个球拍类。

第一个是 Q.Paddle，该类简单定义了一个根据方向属性来左右移动的球拍。此外，它还通过重写 draw 方法来绘制一个简单的黑色矩形。

第二个球拍类继承自 Q.Paddle，定义的是玩家的球拍：Q.PlayerPaddle。该类只负责通过重写 step 方法来基于玩家输入控制 direction 属性；此外，每次只要方向发生变化，它就向服务器端发出一个 move 事件，事件数据包括了球拍的方向和当前的 x 位置。

第三个球拍类是 Q.EnemyPaddle，该类代表的是与玩家一起玩游戏的拍档玩家的球拍。它只对基类 Q.Paddle 做了一处重写，那就是通过绑定套接口来设置球拍的方向和位置，而这就是 Q.PlayerPaddle 传送给应用服务器后又经应用服务器传送过来的 move 消息。

Q.Ball 类用来表示球，它表现出了许多与击砖游戏中的球相似之处。其间的主要不同之处在于，这里的球的属性或时而被服务器更新，或被定期发送给服务器端，借以传送给另一个玩家。

尽管两个游戏客户端从理论上来说都是在以相同的速度模拟游戏，不过，时钟、网络延迟和图形能力的不同就意味着两个客户端彼此之间会慢慢失去同步。为防止出现这一问题，球的位置和方向都会以 ballRate 速率从客户端传送给服务器端，在这个例子中，该速率被设置成每半秒钟一次。

在从属方收到指明更新后的球位置的 ball 消息时，回调更新球的位置和方向，之后以两个客户端之间的网络延迟为时长，手动调用 step 方法来推进球。

在完成精灵的定义后，还有另一要处理的复杂之处，那就是根据玩家是主控方还是从属方来设置场景。代码通过查看 gameType 这一全局变量来实现这一点，若该客户端是主控方，就把玩家置于顶部而把对手置于底部，若是从属方，则对调双方位置。之后，代码仅是插入一个球并让它飞起来。

余下各块代码是一些套接口事件，首先是 slave 和 master 事件，不知你是否还记得服务器端是如何通过指明玩家是 master 或 slave 来设置和开始游戏的，游戏开始之后，这两个事件就会在舞台上呈现 game 场景。接下来是 end 事件，该事件在另一个玩家断开连接时被调用，它仅是通过清除舞台来停止游戏。

比较复杂的是 delay 事件，借助于代码末端最后一个对 setInterval 的调用，该事件每两秒钟被发送一次，setInterval 使用 steps 变量和以毫秒为单位的当前时间来发送一条消息。如你已经在服务器端代码中见到的那样，作为一种衡量端到端之间的包发送延迟的方式，该消息会从另一个客户端弹送回原来的客户端。客户端查看 steps 变量，若它的值等于 3，就可以确定消息已走完了全程。

在回收到该消息之后，原来的客户端提取最新的以毫秒为单位的时间，并使用这一当前时间减去其最初在消息中发送的时间来计算出往返延迟，然后把该数值除以二，由此获得从一个客户端经由服务器端向另一个客户端发送消息所耗费的时间。

把所有部件组合到一起，现在你拥有了这样的一个游戏，在该游戏中，借助一种简单的客户端预测形式来尽量保持彼此之间的同步，两个玩家能够直接与对方玩游戏。

21.6　小结

本章探讨了如何使用 WebSocket 和一个名为 Socket.io 的库来构建多玩家能够直接交互的实时游戏，展示了如何把与本书早前的单玩家游戏相似的代码修改成通过中央服务器传递消息的多玩家游戏。利用本章给出的这些代码，你应能够构建出在多玩家之间准即时交互的实时和半实时游戏。

第22章

构建非传统风格的游戏

本章提要

- 创建一个 Twitter 应用
- 连接 Twitter API
- 构建一个基于 Twitter API 的游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 22 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

22.1 引言

HTML5 游戏开发者有机会突破 Web 游戏惯有的界限，把游戏从 Adobe Flash 的硬性方框中解放出来，让它们扩展到页面的其余部分和网络的其余地方。不仅仅是简单的游戏化，HTML5 游戏还有机会从标准游戏类型中分化出来，进入其他一些领域和媒介。一种可实现这一点的做法是利用其他服务和网站充当媒介，在其之上运行游戏。本章就把 Twitter 用作一个协作版猜词游戏 Hangman 的媒介，Twitter 有着移动友好的网站和客户端。

22.2 创建一个 Twitter 应用

为与 Twitter 交互，需要创建一个 Twitter 应用。这种情况下，若已拥有一个 Twitter 账

户，那么为了避免对 Twitter API 的测试骚扰到你的粉丝，你可能希望再创建一个新的 Twitter 账户。

可通过访问 https://twitter.com/signup 来注册一个 Twitter 账户并选择一个唯一的新名称。本章构建的游戏是一个简单的 Hangman 游戏，在完成注册并确认了自己的账户后，访问 https://dev.twitter.com。这是 Twitter 的开发者站点，在这里你可查阅文档和创建 Twitter 应用，需要使用刚才创建的账户重新登录一次。

在登录后，让鼠标悬停在右上角的账户名称上并单击 My Applications。在应用页面，单击 Create New Application 按钮(见图 22-1)。

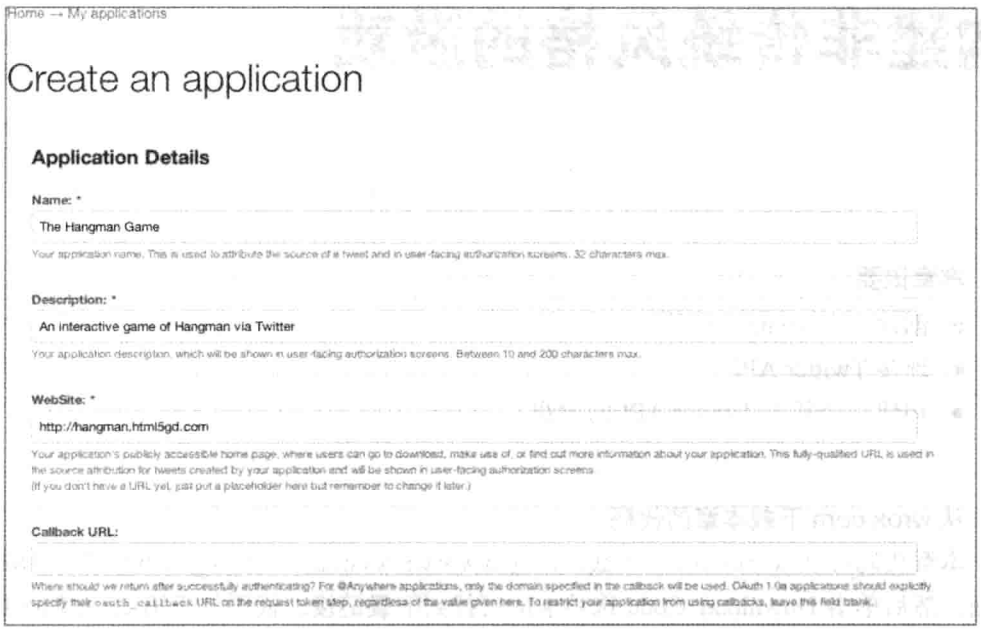


图 22-1 应用的设置界面

填写前面三个必填字段：Name、Description 和 WebSite，也可能要借助一点创意来构想一个唯一名称。接受使用条款，填写验证码，然后提交表单。接着，应用的设置界面就会出现在窗口中。

因为游戏会发送推文(tweet)，所以需要修改应用的许可权限。单击 Settings 选项卡，然后向下滚动页面至 Application Type 段，把类型从 Read Only 改成 Read and Write，然后单击 Update 按钮。

为更便于着手使用 API，你无需走普通的 OAuth 流程(如在第 20 章中所见)，Twitter 提供了一种机制来获取访问令牌(access token)。单击 Details 选项卡，滚动页面至底部，单击 Create My Access Tocken 按钮，这一操作生成一个可以直接使用的访问令牌(你可能需要重新加载该页面，因为 Twitter 对页面信息的更新有时会有点慢)。

保持该页面的打开，因为在下一节中，需要用到其中的四项信息，它们分别是：Consumer Key、Consumer Secret、Access Token Key 和 Access Token Secret。

22.3　将 Node 应用连接至 Twitter

为了通过 Node 连接 Twitter，这里用到了一个名为 ntwitter 的非常好用的模块(该模块是 node-twitter 的一个分支，node-twitter 的灵感则来源于 twitter-node)。ntwitter 模块简化了与 Twitter 的交互，不过，更重要的是，它支持 Twitter 的信息流 API，这意味着可以实时获取发送给自己的推文，不必定期轮询 Twitter。

22.3.1　发送第一条推文

虽然你是通过 NPM 来安装模块的，不过 ntwitter 模块的源代码被托管在 GitHub 上，若需参考，可访问 https://github.com/AvianFlu/ntwitter。

首先，为应用创建一个名为 hangman 的新目录，然后把照例要用到的 package.json 文件放到该目录下，把 ntwitter 作为一项依赖填入其中(见代码清单 22-1)。

代码清单 22-1：package.json

```
{
    "name": "twitter-hangman"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "ntwitter": "0.3.0"
  }
}
```

运行 npm install 来安装 ntwitter 这一依赖。

接下来，尝试通过 API 发送推文；创建一个新的 app.js 文件并打开它，把代码清单 22-2 中的代码填入其中，需要使用上一节中的值来替换掉大写的密钥和密码配置值。最后，运行代码，做法是在命令行中运行命令 node app.js。

代码清单 22-2：发送第一条推文的 app.js

```
var twitter = require("ntwitter");
var client = new twitter({
  consumer_key: "YOUR_CONSUMER_KEY",
  consumer_secret: "YOUR_CONSUMER_SECRET",
  access_token_key: "YOUR_ACCESS_TOKEN_KEY",
  access_token_secret: "YOUR_ACCESS_TOKEN_SECRET"
});

client.verifyCredentials(function (err, data) {
  if(err) {
    console.log("Unable to connect to twitter, please verify config");
  } else {
    client.updateStatus("Hello Twitter!", function (err, data) {
      if(!err) {
```

```
        console.log(data);
      } else {
        console.log(err);
      }
    });
  }
});
```

该段代码先请求加载 ntwitter 模块，然后，使用你在设置应用时创建的凭据来创建一个客户端 client。若这是一个可能会连接多个用户的应用，那么你应通过 OAuth 身份验证流程来获取用户的访问令牌信息。不过，在这个例子中，你使用 Twitter 提供的功能，通过 Web 界面生成这些信息。

接下来，代码调用 verifyCredentials 来确保你输入的配置变量的正确性，这一检查不是必需的，但可以帮助你调试问题或无效的密钥。

最后，代码调用 updateStatus 来发送一条推文，Twitter 回送该条推文的全部数据细节。

若一切按计划顺利进行，你应能在时间线(Timeline)中看到该条推文。若单击展开链接，应会看到来源是你的应用，如图 22-2 所示。

图 22-2 第一条推文

若尝试再次运行该程序，Twitter 会返回一条错误消息，因为它不允许内容完全相同的两条推文被发送多次。

22.3.2 监听用户的信息流

发送推文仅算是完成了一半工作，要对参与游戏的玩家做出响应，需要监听传入的消息。

Twitter 提供了两个用于监听传入消息的 API：一个依赖轮询的 REST API 和一个实时推送消息的信息流 API。就诸如 Hangman 一类用户可能要通过向游戏账户发送推文来参与的游戏而言，信息流 API 能够实现更高的性能。

Twitter 信息流 API 的文档可通过访问 https://dev.twitter.com/docs/streaming-apis/streams/user 获得。

要通过信息流 API 监听与某个特定用户名相关的推文，可使用 ntwitter 的 client.stream 方法。因为用户有可能并未关注参与 Hangman 游戏的人，所以游戏还需要监听未被关注人

的回复。为实现这一点，replies:all 选项必须作为一个选项被传递进去。

使用代码清单 22-3 中突出显示的内容替换掉 app.js 文件后半部的代码，目的是监听那些发向你的账户的推文。需要用自己创建的账户名来替换掉 accountName 属性的值。

代码清单 22-3：读取用户的信息流

```
var twitter = require("ntwitter");
var client = new twitter({
  consumer_key: "YOUR_CONSUMER_KEY",
  consumer_secret: "YOUR_CONSUMER_SECRET",
  access_token_key: "YOUR_ACCESS_TOKEN_KEY",
  access_token_secret: "YOUR_ACCESS_TOKEN_SECRET"
});

var accountName = 'hangmangame';
client.stream('user', { track:accountName ,replies:'all' },
function(stream) {
  stream.on('data', function (data) {
    console.log("****************");
    console.log(data);
    console.log("****************");
  });
  stream.on('end', function (response) {
    // Need to reconnect
  });
  stream.on('destroy', function (response) {
    // Need to reconnect
  });
});
```

若通过命令 node app.js 运行该文件，首先你应会看一个填充了该账户的朋友账户 ID 的列表，之后接口会坐等用户的活动(你的开发账户可能还没有关注别人或被别人关注)。

若向该账户(本例中是@hangmangame)发送推文或是账户自身发送推文，那么这些推文都会被输出到控制台上。

22.4　随机生成单词

为了玩 Hangman 游戏，应用需要访问一个游戏用到的单词表。幸而，网络上有许多提供单词表的地方，获取词表的一个最佳站点是 http://wordlist.sourceforge.net/。在其无数可用的词表中，就 Hangman 的目的而言，最好的词表是 12dicts，该词表包含了一个近似美国英语常见核心词汇的代码清单。

可通过地址 http://downloads.sourceforge.net/wordlist/12dicts-5.0.zip 下载原始的 12dicts 源文件，不过本章代码已经包含了一个名为 words.txt 的文件，该文件存放了最常见单词的一个代码清单，并对该代码清单略加修改，调整了一些行的结尾。12dicts 文件使用

AGID(Automatically Generated Inflection Database，自动生成的变形库)词表作为来源，所以AGID 的许可协议被包含在了本章的下载中，该词表可被免费使用，但在分发时必须包含版权声明。

为了指出存在某种特殊性的单词，words.txt 使用标点符号在一些词的末端做了标记，随机生成单词的代码会被设置成忽略这些单词。

随机生成单词的基本机制是把 words.txt 文件加载到一个很大的数组中，然后从数组中随机挑选一个子项，直至找到一个只包含字母字符的单词。

在本章的后面，这部分代码会被纳入到游戏的主体结构中，不过，若要体验一下随机生成单词的做法，可使用代码清单 22-4 中的代码创建一个名为 word.js 的文件，这段代码会把 10 个随机生成的单词输出到控制台上。

代码清单 22-4：随机生成 10 个单词

```
var fs = require('fs'),
    words = fs.readFileSync('words.txt').toString().split("\n");

function randomWord() {
  var word;
  do {
    word = words[Math.floor(Math.random()*words.length)];
  } while(!word.match(/^\w+$/) || word.length < 5)
  return word;
}

for(var i=0;i<10;i++) {
  console.log(randomWord());
}
```

可以看到，这一代码所做的第一件事是加载文件 words.txt 并使用换行符分割文件。然后，randomWord 方法从单词表中随机挑选一个单词，接着使用一个只包含单词字符(\w)和至少包含五个字母的正则表达式来检查该单词。

每次使用命令 node word.js 运行这一文件都应会在控制台上输出 10 个随机生成的单词，words.txt 文件中存放了超过 32 000 个单词，所以应有足够多的变化。

22.5 创建 Twitter 上的 Hangman 游戏

创建游戏所需的库已准备就绪，现在是时候深入研究真正的游戏代码了。这里的主要做法是，使用一个 Hangman 游戏的填空位置作为内容递送一条推文，然后响应来自用户的猜测所缺少字母的推文。应用向任何给自己发送了推文的用户做出回应，告知用户字母出现的次数，并发送推文更新填字板的状态。

该游戏的完整代码如代码清单 22-5 所示，用代码清单 22-5 中的代码替换 app.js 中的

所有内容，如之前所做的那样，需要替换掉 accountName 和 Twitter 配置变量的值。

代码清单 22-5：Twitter 上的 Hangman 游戏

```
var twitter = require("ntwitter"),
    fs = require('fs'),
    words = fs.readFileSync('words.txt').toString().split("\n");

var client = new twitter({
  consumer_key: "YOUR_CONSUMER_KEY",
  consumer_secret: "YOUR_CONSUMER_SECRET",
  access_token_key: "YOUR_ACCESS_TOKEN_KEY",
  access_token_secret: "YOUR_ACCESS_TOKEN_SECRET"
});
var accountName = "hangmanword";
function randomWord() {
  var word;
  do {
    word = words[Math.floor(Math.random()*words.length)];
  } while(!word.match(/^\w+$/) || word.length < 5)
  return word;
}
var Hangman = function(accountName,client) {
  var self = this;
  this.gameNumber = 0;
  var hangman = "__O-[-<";
  this.newWord = function() {
    this.word = randomWord();
    this.currentWord = this.word.split("");
    this.currentGuesses = [];
    this.guesses = [];
    this.lettersRemaining = this.currentWord.length;
    this.guessesRemaining = 5;
    this.gameNumber++;
    this.sendGameUpdate();
    console.log("\n");
    console.log("New Word:" + this.word);
  };
  this.sendTweet = function(status) {
    client.updateStatus(status,function(err,data) {
      if(!err) {
        console.log("Sent Tweet:" + status);
      } else {
        console.log("Error Sending Tweet:" + status +
                    "\nError:" + err);
      }
    });
  };
  this.sendGameUpdate = function() {
    var status = "Game " + this.gameNumber + ": " +
```

```
            hangman.substring(0,hangman.length - this.guessesRemaining ) +
            " Word:";
  for(var i=0;i<this.currentWord.length;i++) {
    if(this.currentGuesses[i]) {
      status += " " + this.currentWord[i];
    } else {
      status += " _"
    }
  }
  this.sendTweet(status);
};
this.sendExistingGuess = function(tweeter,guess) {
  this.sendTweet("@" + tweeter + ' Sorry someone has already guessed "' +
              guess + '"');
};
this.sendIncorrect = function(tweeter,guess) {
  var extra = this.guessesRemaining <= 0 ?
            " - Game Over (Word was " + this.word + ")" : ""
  this.sendTweet("@" + tweeter + ' sorry there are no ' +
              guess + "'s in game " + this.gameNumber + extra);
};
this.sendCorrect = function(tweeter,guess,correct) {
  var extra = this.lettersRemaining == 0 ?
            " - Congratulations you win!" : ""
  this.sendTweet("@" + tweeter + ' yes, ' + guess + " appears " +
              correct + (correct > 1 ? " times" : " time") +
              " in game " + this.gameNumber + extra);
};
this.handleGuess = function(tweet) {
  if(!tweet.text) return;
  var guess = tweet.text.replace(/[^a-z]/gi,""),
      tweeter = tweet.user.screen_name,
      correct = 0;
  try {
    if(tweet.text.indexOf("@" + accountName) === 0) {
      guess = guess[guess.length-1].toLowerCase();
      if(this.guesses.indexOf(guess) != -1) {
        return this.sendExistingGuess(tweeter,guess);
      }
      this.guesses.push(guess);
      for(var letter=0;letter < this.currentWord.length;letter++) {
        if(this.currentWord[letter].toLowerCase() == guess) {
          correct++;
          this.lettersRemaining~DH;
          this.currentGuesses[letter] = true;
        }
      }
      if(correct > 0) {
        this.sendCorrect(tweeter,guess,correct);
        if(this.lettersRemaining == 0) {
```

```
            this.newWord();
          } else {
            this.sendGameUpdate();
          }
        } else {
          this.guessesRemaining~DH;
          this.sendIncorrect(tweeter,guess);
          if(this.guessesRemaining > 0) {
            this.sendGameUpdate();
          } else {
            setTimeout(function() { self.newWord(); }, 2000);
          }
        }
      }
    } catch(e) {
      console.log("Error:" + e.toString());
    }
  };

  this.connect = function() {
    client.stream('user',
                  { track:accountName ,replies:'all' },
                  function(stream) {
      stream.on('data', function (data) {
        setTimeout(function() {
          self.handleGuess(data);
        },1);
      });

      stream.on('end', function (response) {
        self.connect();
      });

      stream.on('error', function (response) {
        console.log("Error");
      });

      stream.on('destroy', function (response) {
        self.connect();
      });
    });
  };
  this.newWord();
  this.connect();
};
var hangman = new Hangman(accountName,client);
```

游戏的主要代码被放在一个 Hangman 对象中，该对象跟踪并记录下当前游戏的状态。要开始一个新游戏，对象调用 newWord 方法，该方法随机抓取一个单词，然后初始化要猜测的内容并把游戏递送到 Twitter 上。接下来是 sendTweet 方法，该方法只是调用客户端的

updateStatus 发送一条推文，然后捕获所发送的推文或是发生的任何错误并把它们输出到控制台上。

sendGameUpdate 方法以如下形式输出一条推文：

```
Game 1: __O-[- Word: _ _ _ m _ _ _
```

该条推文给出当前游戏的编号、在人被吊死前剩下的猜测次数(以一种简陋的 ASCII 渲染效果表示)，以及已被找出的字母。

接下来的三个方法用来响应参与游戏的人，这些方法向玩家发送消息，告知其上一次的猜测情况。

游戏代码的主体部分是 handleGuss 方法，该方法接收一条推文，先进行检查以确保该推文是发给游戏账户的，然后，它提取推文的最后一个字母字符，把该字符当成猜测的字符。

这一做法允许用户按照以下这些方式发送 tweet 消息：

```
@hangmangame is there an a?
@hangmangame how about a b?
@hangmangame c
```

所有上述例子都是有效的，允许在发送消息时带点变化，这很有必要，因为 Twitter 不允许连续投送两条相同的推文。

在抽取出猜测字母后，游戏检查猜测字母是否已被试过，若是则告知玩家；若这不是一次重复的猜测，那么游戏会检查该字母出现的次数和剩余的要猜测的字母个数，然后基于用户是否正确猜出了单词来给用户发送一条推文。

若玩家猜出了单词或用完了猜测次数，那么游戏会告知用户，然后再发出一个新的单词。图 22-3 给出了一个游戏例子。

图 22-3 一个游戏例子

22.6　小结

本章展示了如何在 Twitter API 之上构建游戏，结合 Twitter 提供的推送通知，本章实现了一个玩家可通过智能手机参与的异步 Hangman 游戏。除了在 Twitter 上构建游戏外，还可以把自己的视野扩展到可用来构建游戏的各种网络部件上。可供利用的网络和移动可访问的交互媒介数量庞大，其中包括 SMS、Photo API 和搜索 API 等，可以创造性地混搭使用所有这些媒介，创建一些具有革新性的游戏。

第Ⅶ部分

移 动 增 强

- 第 23 章：通过地理位置定位
- 第 24 章：查询设备的方向和加速
- 第 25 章：播放音效：移动设备的罩门

第23章

通过地理位置定位

本章提要

- 使用地理位置定位用户
- 绘制静态地图
- 绘制交互式地图

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 23 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

23.1 引言

到目前为止，本书主要把“移动”这一单词视作一种尺寸形状(小屏幕)和一种输入机制(触摸屏)的代指。移动设备的第三个主要方面是，它们是便携式的，因此，用户的位置可被纳入游戏中，构成游戏的一个有趣方面。本章探讨对“地理定位(geolocation)”的设备位置测定的支持，地理定位是 HTML5 规范家族的规范之一；此外，本章还讨论如何在游戏中使用设备的位置。

23.2 地理定位入门

从技术角度看，地理定位支持并非 HTML5 的组成部分，而是一个单独存在的地理定

位 API 规范。该规范的最新发布版本被保存在 W3 网站上，可通过地址 www.w3.org/TR/geolocation-API/访问。

尽管本章从移动设备角度讨论地理定位，不过，该 API 在桌面浏览器中也是可用的。支持该 API 的桌面浏览器(IE9+和其他所有浏览器的最新版本)使用的是一种精确度较低的 IP 地址反向查找机制。

该 API 定义了两种抓取位置的机制：单次抓取和监视抓取。前一种类型在对地址有一次性需求时使用，后一种的工作方式类似于 setInterval，它会随着设备的移动重复调用回调。

因为通过网页获取用户的位置涉及隐私顾虑，所以，这两种机制都会向用户发送通知，赋予他们允许或拒绝抓取位置请求的权力。

23.3 一次性获取位置

获取用户位置是一件相当简单的事情，只需使用一个回调来调用 navigator.geolocation.getCurrentPosition 即可。该调用触发一个在浏览器顶部显示的通知，如图 23-1 所示，赋予用户允许或拒绝请求的权力。若提供了第二个回调，那么如果浏览器因请求被拒或其他错误不能获取位置，就会调用该回调。

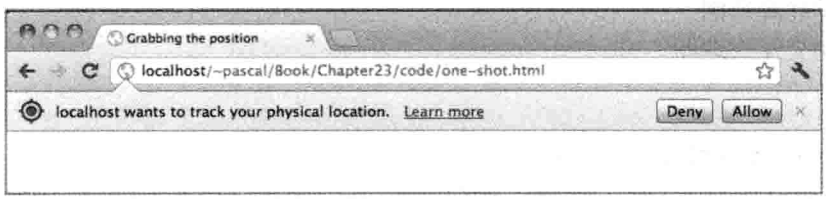


图 23-1 地理定位许可对话框

要在控制台中查看请求返回的数据，可将代码清单 23-1 中的代码输入到一个名为 position.html 的文件中，然后在桌面浏览器中加载该页面，同时打开 JavaScript 控制台。

代码清单 23-1：单次获取位置

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Grabbing the position</title>
</head>
<body>
  <script>
    function logPosition(position) {
      console.log(position);
    }
    function positionError(error) {
      console.log(error);
    }
```

```
    navigator.geolocation.getCurrentPosition(logPosition,positionError);
  </script>
</body>
</html>
```

检查控制台就能看到输出的请求结果。若你拒绝了请求，或系统无法找出你的位置，就会调用以 PositionError 对象为参数的 positionError 方法。若通过 file:// URL 运行该文件，Chrome 默认给出一个错误，所以你应通过 localhost(本地主机)来运行它。

若地理定位成功，就会调用以 Position 对象为参数的 logPosition 回调，该对象会被输出到控制台中，如图 23-2 所示。

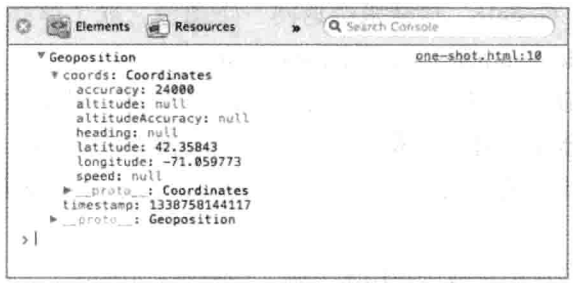


图 23-2　返回位置例子

大部分详细信息都被置于一个 coords 子对象中，该对象至少包含了纬度、经度和以米为单位的精度，此外，它可能还包含了其他一些数据。

以下是位置对象提供的所有域：

- Latitude(纬度)：纬度的最佳数值猜测
- Longitude(经度)：经度的最佳数值猜测
- Altitude(海拔)：海拔估计，若不做估计则为 null
- Accuracy(精度)：以米为单位的纬度和经度的准确度
- altitudeAccuracy(海拔精度)：以米为单位的海拔准确度或 null
- heading(朝向)：若速度大于零，则为以度为单位的方向，否则为 NaN
- speed(速度)：以米/每秒为单位的速度

如前所述，其中只有经度、纬度和精度是保证提供的。

此外，getCurrentPosition 还接受一个选项对象作为第三个参数，截至撰写本书之时止，这一对象提供了以下三个选项：

- enableHighAccuracy：提示精度很重要，这可能需要花费更长时间来生成位置，会消耗更多电池电量，但如有可能，结果也会更精确。
- Timeout：在超时之前等待位置返回的时长，以毫秒为单位。
- maximumAge：位置的最长存在期，以毫秒为单位，若大于零，则方法可以返回缓存位置。

默认情况下，enableHighAccuracy 被设置成 false；timeout 被设置成 0，这意味着永不会超时；maximumAge 也被设置成 0，这意味着不使用缓存数据。若希望快速获得位置信

息，那么把 maximumAge 设置成一个大于零的值。

23.4 在地图上标出位置

经度和纬度能告诉你的信息是有限的，所以，首先要做的事情是在地图上标出位置。为此，需要访问一个地图 API。Google Maps 是最受欢迎的地图 API 之一，它提供了两种不同的 API，一种是静态地图 API，另一种是你可能十分熟悉的传统交互式地图。

生成静态地图很容易，所涉及的就是以<img>标签的 src 为表达方式，发送一个有着适当形式的请求给 Google，该请求会返回一个图像。欲了解静态地图 API，可访问 https://developers.google.com/maps/documentation/staticmaps/。

若希望得到一张在某个具体位置上放置了标记的地图，那么需要使用输出图像的尺寸、标记和一个传感器选项值来生成一个 URL，传感器选项的作用是告知 Google 该应用是否使用传感器来确定位置(该字段是必填的)。此外，你还需要传送一个缩放值来控制生成图像的放大方式。

标记的定义方法是使用竖线字符(|)来分隔一些特性，例如：

```
markers=color:red||label:A|lat,long
```

因为这需要被编码进 URL 中，所以竖线字符被 URL 编码成字符串%7C。

代码清单 23-2 给出的代码把第一个例子修改成输出一张你所在位置的静态地图，各桌面浏览器的精确程度差别极大，所以，它可能只能显示一个与你的实际位置大致相近的位置。

代码清单 23-2：静态地图

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Grabbing the position</title>
  <script src='js/jquery.min.js'></script>
</head>
<body>
  <script>
    function logPosition(position) {
      var url = "http://maps.googleapis.com/maps/api/staticmap?" +
              "zoom=13&size=320x420&" +
              "markers=color:blue%7Clabel:S%7C" +
              position.coords.latitude + "," +
              position.coords.longitude + "&sensor=true";
      $("<img>").attr("src",url)
              .appendTo("body");
    }
    function positionError(error) {
```

```
    console.log(error);
  }
  navigator.geolocation.getCurrentPosition(logPosition,positionError,{
    enableHighAccuracy: true
  });
 </script>
</body>
</html>
```

该例子把 jQuery 用于 DOM 操纵，且把 enableHighAccuracy 设置成 true，由此来获得一个尽可能准确的位置。

23.5　监视位置随时间的变化

地理定位 API 另外提供了一个名为 watchPosition 的方法，该方法接收的参数与 getCurrentPosition 的相同。它的工作方式类似于 setInterval，它会返回一个稍后会被 clearWatch 用来清除监视的 ID。

若想了解走动情况，那么运行代码清单 23-3 中的代码，这一代码使用 watchPosition 把经度和纬度输出到移动浏览器的一个 中，你可从中了解这些数值发生了怎样的变化。

代码清单 23-3：监视位置的变化

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Watching the position</title>
  <script src='js/jquery.min.js'></script>
</head>
<body>
  <script>
    function logPosition(position) {
      $("#logs").prepend(position.coords.latitude + "," +
                         position.coords.latitude + "<br/>");
    }
    function positionError(error) {
      console.log(error);
    }
    var watchID = navigator.geolocation.watchPosition(
                                  logPosition,positionError,{
                                      enableHighAccuracy: true
                                  });
    setTimeout(function() {
      navigator.geolocation.clearWatch(watchID);
    },30000);
  </script>
  <div id='logs'></div>
```

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

你可看到，这一监视在 30 秒钟之后会被清除，以免系统不停更新位置。

注意：激活移动设备上的 GPS(这实际上就是你在启用 enableHighAccuracy 选项时要求要做的事情)会消耗电池电量，所以要为用户周全考虑，只有在必须更新地址时才使用监视。

23.6 绘制交互式地图

要绘制交互式地图，需要使用交互式的 Google Maps API。该 API 之前的版本都要求有一个 API 密钥，但当前的 v3 版本已不再需要。不过，若希望借助于这一 API 赚钱或是跟踪自己的使用情况，那么你必须通过 https://code.google.com/apis/console 获取一个 API 密钥。

Google 网站提供了完整的 Maps v3 API 文档，可通过地址 https://developers.google.com/maps/documentation/javascript/reference 访问。

虽然该 API 包罗万象，但它具有非常完善的文档，且你只会用到所有可用对象的一个很小的子集：Map、Marker 和 LatLng。Map 对象代表了整张地图；Marker 对象是可放在页面上的标记，就像你在静态地图中所做的那样；LatLng 则用来存放单个位置。

要创建地图，需要创建一个新的地图对象，并把一个用来填充的 DOM 元素传递给它，三个必填的选项是中心点、初始缩放级别和 mapTypeId。

中心点是一个 LatLng 对象，可通过传入两个分别代表纬度和经度的浮点数来创建该对象。缩放级别是一个介于 1～18 之间的数值，它控制地图的放大方式，不过有些区域无法达到 18 级(这通常是美国的农村地区和世界范围内的其他地区)。mapTypeId 对象为 google.maps.MapTypeId 类中的四个常量之一，每个常量分别代表了一种 Google Maps 支持的地图类型：HYBRID(混合)、ROADMAP(道路)、SATELLITE(卫星)和 TERRAIN(地形)。

若仍打算尝试走动的情况，那么运行代码清单 23-4 中代码来创建一个交互式地图，借助一个随着你的移动来更新自身的图钉，该地图将能跟踪你的位置。

代码清单 23-4：一个自动更新的交互式地图

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Interactive Map</title>
  <script src='js/jquery.min.js'></script>
  <meta name="viewport" content="width=device-width, user-scalable=0,
```

```
minimum-scale=1.0, maximum-scale=1.0"/>
  <script type="text/javascript"
src="https://maps.googleapis.com/maps/api/js?sensor=true"></script>
  <style> body { padding:0px; margin:0px; } </style>
</head>
<body>
  <script>
  $(function() {
    var map = null,
       marker = null;
    function createMap(latlng) {
      var mapOptions = {
        zoom: 18,
        center: latlng,
        mapTypeId: google.maps.MapTypeId.TERRAIN
      };
      var div = $("<div>").css({ width: "100%",
                               height:"100%",
                               position:"absolute"})
                          .appendTo("body")[0];

      map = new google.maps.Map(div, mapOptions);
      marker = new google.maps.Marker({
        position: latlng,
        map: map,
        title: "Me"
      });
    }
    function updateMap(pos) {
      var latlng = new google.maps.LatLng(pos.coords.latitude,
                                      pos.coords.longitude);
      if(!map) {
        createMap(latlng);
      } else {
        marker.setPosition(latlng);
      }
    }
    function positionError(error) {
      alert("Error tracking your position");
      navigator.geolocation.clearWatch(watchID);
    }
    var watchID = navigator.geolocation.watchPosition(
                               updateMap,positionError,{
                                 enableHighAccuracy: true
                               });
  });
  </script>
</body>
</html>
```

事件的循环始于对 watchPosition 的调用，每次只要位置发生变化，该方法就会触发一个到 updateMap 的调用。updateMap 创建一个新的 LatLng 对象，然后判断地图之前是否已被绘制，若没有则调用 createMap 来生成最初的地图；若已经绘制了地图，则调用标记的 setPosition 命令把标记更新到新的位置上。

23.7 计算两点间的距离

在开始使用地理定位来构建游戏时，首先要面对的一个困难是计算两对经纬度之间的距离。无论是检测玩家之间的邻近性还是到目标的距离，这都是你必须解决的问题。

在客户端，Google 的 Map v3 API 在 google.maps.geometry.spherical 下提供了一个名为 computeDistanceBetween 的静态方法，该方法接收两个 LatLng 对象并返回以米为单位的距离。

若手边还没有可用来便捷地完成这一计算的 API，那么可使用半正矢(Haversine)距离公式(http://en.wikipedia.org/wiki/Haversine_formula)来计算球面两点之间的距离。

就使用 JavaScript 实现的这一公式而言，存在许多可用资源，不过最好用的网络资源之一是 www.movable-type.co.uk/scripts/latlong.html，它提供了一个使用 JavaScript 来实现的简洁半正矢公式，该公式接收 lat1、lon1、lat2 和 lon2，并输出两点间以公里为单位的距离：

```
var R = 6371; // km
var dLat = (lat2-lat1).toRad();
var dLon = (lon2-lon1).toRad();
var lat1 = lat1.toRad();
var lat2 = lat2.toRad();

var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
        Math.sin(dLon/2) * Math.sin(dLon/2) * Math.cos(lat1) * Math.cos(lat2);
var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
var d = R * c;
```

可以把这一公式直接插入到代码的某个方法中，用它来计算两点之间的距离。

23.8 小结

本章展示了如何在浏览器中使用地理定位来生成可用来显示交互式地图的位置，有了这种跟踪位置以及显示和更新交互式地图的能力，许多增强现实游戏都可被构建在浏览器中，这其中包括了清道夫狩猎游戏、地理藏宝游戏、基于邻近性的游戏等等。本章把一些地理定位工具添加到了你的开发工具库中，但愿这一举措能为你开辟出一片突破了游戏惯有界线的 HTML5 游戏新天地。

第24章

查询设备的方向和加速

本章提要

- 了解屏幕方向
- 了解设备方向 API
- 运用设备方向
- 防止设备旋转

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 24 章的下载中，代码文件的名称分别依照本章各处使用的文件名称命名。

24.1 引言

使用移动设备的方向来控制游戏，这曾是智能手机游戏时代的一项惊人之举。随着浏览器对设备方向和加速的支持，作为 HTML5 游戏开发者，现在这一功能对你来说已是触手可及。本章探讨了 DeviceOrientation Event API(设备方向事件 API)，该 API 提供了两个十分有用的事件：设备方向和设备移动事件。此外，它还提供了第三个事件，即罗盘；不过，除非所实现的是需要精确方向数据的长运行期应用，否则，尽可以忽略此事件。

24.2 考查设备的方向

在探讨 DeviceOrientation Event API 之前，很值得做的一件事情是简单考查一下 window.

orientation 属性。该属性不会告知你设备确切的持握角度，不过，它可以告诉你一个指明了朝向的角度，该角度指示哪一个方向——竖屏或横屏——是设备屏幕的方向。

此外，还可以通过监听 orientationchange 事件来检测设备何时被旋转至一种不同的配置。要测试这一功能，把以下代码添加到任何加载了 jQuery 的页面中：

```
$( window ).on("orientationchange",function(e) {
  alert ( window.orientation );
});
```

根据设备的不同，随着设备的转动，每次在屏幕旋转时你都应会看到一个以 90 为增量递增的值，该值告知你设备当前的朝向角度。虽然看上去解码这一值是很简单的事情，但实则不然。

手机，特别是 iPhone 和 Galaxy Nexus，会把正常的竖屏位置看成是 window.orientation 值为 0 的情况。这两种手机都不支持倒转的竖屏模式，所以不存在 180 这一值。iPhone 有两种横屏方向：90 和-90，而 Galaxy Nexus 则把横屏方向视为 window.orientation 为 90 的情况。

对于平板电脑来说，这一情况就更混乱。Kindle Fire 和 iPad 都把正常的竖屏模式看作是方向值为 0 的情况，不过，它们还使用方向值 180 来支持手机并不支持的倒置竖屏模式。其他平板电脑，如 Android ASUS Transformer，则把横屏模式当成方向值为 0 的情况，其他情况则以此为基准做旋转。

不过，这些不同值并未与规范相悖，就 ASUS 平板电脑而言，把方向值 0 指为横屏模式是因为横屏被定义成平板电脑的“标准方向”。

对于设备来说，“标准方向”这一说法是一个非常重要的概念，因为 deviceorientation 事件也都与标准方向相关，且始终把标准方向假设成竖屏模式。

24.3 设备方向事件入门

若希望获得更多信息，而不仅是屏幕的竖屏或横屏方向，那么此时就要深入研究 deviceorientation 事件了，该事件给出了三个角度值，这些值精确指示了设备在 3D 空间中的持握方式，且事件会随着设备的调整而被高频触发。

与设备方向相关的规范是 DeviceOrientation 事件规范(DeviceOrientation Event Specification)，你可通过 www.w3.org/TR/orientation-event/获得该规范的最新版本。其最重要的事件是 deviceorientation，可使用该事件来确定设备的持握角度。

虽然这是一个用于移动设备的事件，不过桌面浏览器也已开始支持该事件(MacBook 几年前就已开始安装加速度计)，该事件在 Chrome 中是可用的。Firefox 也提供了支持，不过至少在 OS X 中，Firefox 并未触发该事件。截至撰写本书之时为止，桌面 IE、Safari 和 Opera 都还未提供支持。

24.3.1　检测和使用事件

要确定自己的浏览器是否支持该事件，可以检查该事件对象是否存在于 window 对象中。

```
if (window.DeviceOrientationEvent) {
    // Device orientation supported
}
```

Firefox 6 之前的版本支持一个非标准的 OrientationEvent 事件，不过自版本 6 之后它已支持标准事件。

接下来，要监听该事件，那么一如既往，可以使用 addEventListener 或 jQuery。唯一需要说明的是，若使用 jQuery，那么需要取出原始事件对象，通过该对象访问所关心的事件属性，因为 jQuery 没有把 deviceorientation 事件的属性复制到自己的统一事件对象中。

```
// Use either method
window.addEventListener("deviceorientation",function(eventData) {
  // Handle event
});

$(window).on("deviceorientation",function(e) {
 var eventData = e.originalEvent;
 // Handle event
});
```

在以上两个例子中，eventData 对象都存放了一些你所关心的属性。

24.3.2　了解事件数据

deviceorientation 事件使用一个包含了三个不同属性的对象来触发自己的回调，每个属性存放一个不同的绕轴旋转角度：alpha、beta 和 gamma。

- alpha - [0 - 360]：设备的朝向(可看成是北、南、东、西这样的方向)，可以通过把 360 减去 alpha 来判断罗盘的朝向。
- beta - [-180 -180]：设备的前后倾斜角度，beta 值为 0 意味着设备是平放的，beta 值为 90 则意味着它被垂直持握，顶端向上。
- gamma - [-90 -90]：设备的左右倾斜角度，gamma 值为 0 意味着设备是平放的，gamma 值为-90 则意味着设备向左垂直倾斜。

因为 alpha 依赖于用户面对的方向，所以，在游戏中，你一般只会用到 beta 和 gamma，因为无论玩家面朝哪个方向而坐，这两个属性都是可供使用的。

alpha 主要用在增强现实类装置中，如 Android 设备上(alpha 值并不是特别准确，不过这只是笔者的经验之谈，所以你的情况可能会有所不同)。

24.4 试用设备方向

要试用设备方向事件，可使用第 14 章中的 SVG 和物理代码构建一个简单的演示例子。图 24-1 呈现了例子的最终画面。

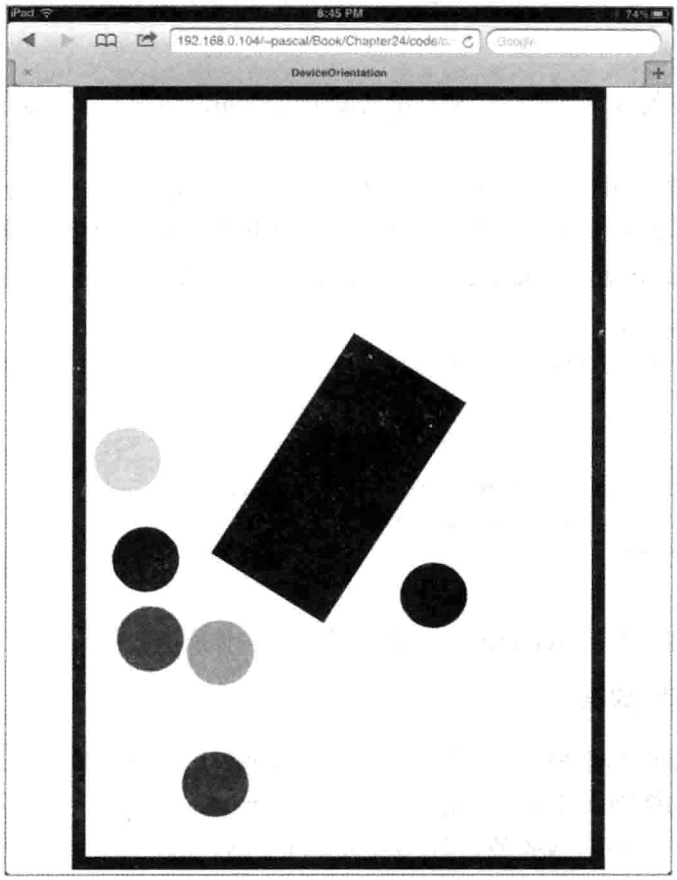


图 24-1 最终实现的设备方向例子

该演示例子包括了一组静态的墙，这些墙用来围住一组可对重力做出反应的球。重力始终与现实情况保持一致，不过随着你转动设备，重力会相对于设备发生改变，这会导致球在屏幕上四处飞动。

24.4.1 创建一个玩球的场所

首先，使用代码清单 24-1 中的内容创建一个名为 orient.html 的 HTML 新文件。除了第 14 章中的 Box2dWeb-2.1.a.3.js 之外，你还需要用到一些 quintus 文件和引擎的依赖。

代码清单 24-1：方向例子的 HTML 文件

```
<!DOCTYPE HTML>
<html lang="en">
```

```
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, user-scalable=0,
 minimum-scale=1.0, maximum-scale=1.0"/>
    <title>DeviceOrientation</title>
    <script src='js/jquery.min.js'></script>
    <script src='js/underscore.js'></script>
    <script src='js/Box2dWeb-2.1.a.3.js'></script>
    <script src='js/quintus.js'></script>
    <script src='js/quintus_input.js'></script>
    <script src='js/quintus_sprites.js'></script>
    <script src='js/quintus_scenes.js'></script>
    <script src='js/quintus_physics.js'></script>
    <script src='js/quintus_svg.js'></script>
    <script src='orient.js'></script>
    <style>
      * { padding:0px; margin:0px; }
    </style>
  </head>
  <body>
  </body>
</html>
```

接着，创建代码清单 24-1 中引用的 orient.js 文件，并用代码清单 24-2 中的内容填充它。

代码清单 24-2：orient.js

```
$(function() {
  var Q = window.Q = Quintus()
                      .include('Input,Sprites,Scenes,SVG,Physics')
                      .svgOnly()
                      .setup('quintus',{ maximize: true });
  Q.Ball = Q.Sprite.extend({
    init: function(props) {
      this._super(_(props).defaults({
        shape: 'circle',
        color: 'red',
        r: 25,
        restitution: 0.9,
        density: 4,
        seconds: 5
      }));
      this.add('physics');
    }
  });

  Q.scene('level',new Q.Scene(function(stage) {

    stage.add("world");

    // Create the walls
```

```
    stage.insert(new Q.Sprite({ x: 5, y: 300, w: 10, h: 600 }));
    stage.insert(new Q.Sprite({ x: 395, y: 300, w: 10, h: 600 }));
    stage.insert(new Q.Sprite({ x: 200, y: 5, w: 400, h: 10 }));
    stage.insert(new Q.Sprite({ x: 200, y: 595, w: 400, h: 10 }));

    // Add the center object
    var center = stage.insert(new Q.Sprite({
      x: 200, y: 300, w: 100, h: 200
    }));

    stage.each(function() {
      this.p.type = 'static';
      this.add("physics");
    });

    // Add the balls
    stage.insert(new Q.Ball({ x: 100, y: 50, color:"blue" }));
    stage.insert(new Q.Ball({ x: 200, y: 50, color:"pink" }));
    stage.insert(new Q.Ball({ x: 300, y: 50, color:"black" }));
    stage.insert(new Q.Ball({ x: 100, y: 150, color:"green" }));
    stage.insert(new Q.Ball({ x: 200, y: 150, color:"teal" }));
    stage.insert(new Q.Ball({ x: 300, y: 150, color:"orange" }));
    stage.viewport(400,600);
    stage.centerOn(200,300);

  }));

  Q.stageScene("level");
});
```

至本书此处，代码清单 24-2 中的代码看上去应很熟悉才对。

该段代码仅定义了一个可重用的精灵：Q.Ball，该精灵类定义了球的形状、大小和物理属性，并加入了 physics 组件，以便让球回应重力和其他对象。

墙壁精灵被创建为普通的 Q.Sprite 对象(请记住这些是 Q.SVGSprite 对象，只不过设置阶段的 svgOnly ()调用把 Q.SVGSprite 复制给 Q.Sprite)。之后，这些对象的类型被设置成 static，目的是把它们变成非移动的对象；接着 physics 组件也被添加到了这些对象中。

接下来，六个不同颜色的球被添加到了舞台中。

若在浏览器中加载这段代码，你应会看到六个正在垂直下落的球。

24.4.2 添加方向控制

为添加相对于设备的重力调整支持，这里需要加入一个 deviceorientation 事件处理程序，该程序提取旋转数据，在必要时调整 Box2D 环境中的重力矢量。

因为值为 0 的 beta 和 gamma 意味着设备是平放的，因此球不应移动，调整重力的最简易做法是把某个常量乘以 beta 的正弦，由此获得重力的 y 分量，以及乘以 alpha 的正弦，由此获得重力的 x 分量(从球的角度出发)。

为给体验注入更多刺激性，被存放在 center 变量中的中央块也会发生旋转，始终面向北方(或 alpha 值为 0)。

要尝试这一做法，如代码清单 24-3 所示，把其中突出显示部分代码添加到靠近 orient.js 文件底端的位置。

代码清单 24-3：方向事件处理程序

```
stage.viewport(400,600);
stage.centerOn(200,300);

if (window.DeviceOrientationEvent) {
  $(window).on("deviceorientation",function(e) {
    var eventData = e.originalEvent
        tiltLR = eventData.gamma,
        tiltFB = eventData.beta,
        direction = eventData.alpha;

    center.physics._body.SetAngle(direction * Math.PI / 180);

    var leanAngle = tiltLR * Math.PI / 180,
        tiltAngle = tiltFB * Math.PI /180,
        gravityX = 20 * Math.sin(leanAngle),
        gravityY = 20 * Math.sin(tiltAngle);
    stage.world._world.m_gravity.x = gravityX;
    stage.world._world.m_gravity.y = gravityY;
  });
}

}));
```

若同样在受支持移动设备的浏览器中，或在最新 MacBook 的 Chrome 中运行这一例子，那么可以与其中的球互动，通过轻微旋转设备来调整它们的移动(最多至方向翻转的位置，之后情况将变得不可预料，球不再遵循重力的作用)。

24.4.3　处理浏览器的旋转

翻转设备至方向发生改变的点会给前端带来一个与 HTML5 游戏中的设备方向相关的重要问题：作为 Web 开发者，你目前无法通过锁住屏幕显示来防止旋转。若用户手机的倾斜角度过大，那么他们最终会在横屏和竖屏之间进行切换，反之亦然。除了以这样一种方式来构建游戏，即防止用户以过大角度翻转设备之外，还没有一种完整方案可用来解决这一问题。

好消息是，存在一个 Screen Orientation API 规范：www.w3.org/TR/screen-orientation/，该规范包含了锁定屏幕的功能。坏消息是，截至撰写本书之时止，该规范只在 Firefox Mobile 中获得了实现。

在 Screen Orientation API 普及之前，一个部分解决方案是，任何时候只要有 orientationchange

事件发生，就检查 window.orientation 值。window.orientation 包含了以度为单位的设备默认持握位置的方向值。这实际上要比设想的复杂一些，因为如你在“考查设备的方向”一节所见，window.orientation 值并非是跨设备或平台一致的。

就处理非 Android 手机设备而言，很遗憾，目前做法似乎是把任一方向的横屏模式都视作 window.orientation 值为 90 这一情况，可将代码清单 24-4 中的代码添加到 orient.js 中，置于 Q.Scene 之前，其作用是基于 window.orientation 的值来变形容器。

代码清单 24-4：处理窗口的旋转

```
function rotateContainer() {
  $("#quintus_container")[0].style.webkitTransform =
                  "rotate(" + -1*window.orientation + "deg)";
}
rotateContainer();
$(window).on("orientationchange",rotateContainer);

Q.scene('level',new Q.Scene(function(stage) {
```

其中的 rotateContainer 方法是 WebKit 厂商前缀特定的，不过，通过处理各种旋转前缀，它可被扩展到其他浏览器中。该方法根据 window.orientation 的值来反向旋转容器元素，这样容器的角度就不会发生改变。

如前所述，，因为截至撰写本书之时止，Android 手机只提供一个方向值 90(绝不会有 -90)，所以，对于那些存在设备旋转-90 度这种情况的手机来说，这一修正做法并不起作用。

需要更多控制？

DeviceOrientation 事件规范提供了一个名为 devicemotion 的更复杂事件，该事件赋予设备移动更加细粒度的控制。它提供了一个 acceleration 和一个 accelerationIncludingGravity 子对象，这两个对象都拥有原始的 x、y 和 z 加速值。此外，它还包含一个 rotationRate 对象，该对象提供了一段给定时间内设备旋转的 alpha、beta 和 gamma 值。devicemotion 的应用难度更高一些，而且在大多数情况下，它并不会带来许多的附加价值；不过，若需要更细粒度的控制，你可通过前面提供的链接访问该规范，了解其中的详情。

24.5 小结

本章简要介绍了 window.orientation 属性值，此外，还介绍了使用 deviceorientation 事件把基于加速计的玩法添加到 HTML5 游戏中的详细过程。不过，该功能附带了一项警告说明，那就是设备的旋转会导致设备屏幕方向发生改变，从而无法很好配合 HTML5 游戏的需要。通过向浏览器开放设备的方向，现在，HTML5 游戏开发者有了更多工具，这些工具可用来构建用到了创造性输入机制的游戏。

第25章

播放音效：移动设备的罩门

本章提要

- 了解<audio>标签
- 创建一个桌面音效引擎
- 创建一个移动音效引擎

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 25 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

25.1 引言

除了给浏览器功能带来其他许多增强，HTML5 最后还兑现了将音频视为一等公民的承诺。尽管有其局限性，不过简单的<audio>标签可用在桌面 HTML5 游戏中播放音乐和一些花哨的声音效果。遗憾的是，移动设备上的 HTML5 音频并未获得足够重视；本章探讨当前移动设备上的 HTML5 音频的局限性，以及针对这些局限性的一些可能应变做法。

25.2 使用 audio 标签

如前所述，作为 HTML5 规范的核心组成部分，HTML5 定义了一个<audio>标签。该标签的主要设计目标是页面内的音频播放，不过，标签的灵活性意味着游戏开发者同样可

以赋予它播放游戏音效的新用途。

25.2.1 把 audio 标签用于简单播放

< audio >标签可用来创建一个页面音频播放器，仅需一个标签即可：

```
<audio src="music.mp3" controls/>
```

不过，从游戏开发角度看，更令人感兴趣的是，借助你曾在第 10 章中大致了解的 Audio 对象，<audio>标签还可以完全脱离于可视组件的方式进行创建。

```
var snd = new Audio();
snd.src = "music.mp3";
snd.addEventListener("canplaythrough",function() {
  snd.play();
});
snd.load();
```

上述例子创建一个新的音频对象，并把来源设置为文件 music.mp3；之后，一旦 canplaythrough 事件被触发就开始播放音乐。canplaythrough 的意思是音频文件的加载量已足以开始播放，若保持当前的加载速率，它会在播放到达文件结尾处之前完成加载。

25.2.2 处理不同的受支持格式

如第 10 章所述，不同浏览器支持不同的音频格式，皆因目前还没有哪一种格式获得了所有浏览器的支持。若要最大范围地涵盖这些浏览器，需要支持至少两种格式：.mp3 和.ogg，或者.mp3 和.wav。因为.ogg 采用的是可与.mp3 的小文件尺寸相媲美的有损压缩，所以相对于.wav 而言是一种更好的选择。

从标记角度来说，若希望支持这一格式，可使用不同的来源(source)标签表示，并依赖浏览器从它们支持的格式中选出第一项来源：

```
<audio controls>
  <source src="music.mp3" type="audio/mp3" />
  <source src="music.ogg" type="audio/ogg" />
</audio>
```

若使用 JavaScript 加载文件，可使用 Audio 对象的 canPlayType 方法来检查类型支持，从而决定加载哪个元素：

```
var snd = new Audio();

if(snd.canPlayType('audio/mpeg')) {
  snd.src = "music.mp3";
} else if(snd.canPlayType('audio/ogg; codecs="vorbis"') {
  snd.src = "music.ogg";
}

// ... load and play the sound
```

因为 audio/ogg 这一 MIME 类型是一个支持多种不同编解码器的容器，所以需要检查具体采用的编码(就音频而言通常是 vorbis)。

为什么不支持单一格式？

作为 Web 开发者，一件令人难以想象的郁闷之事是没有一种音频(就此而言，或是视频)格式在所有浏览器中获得了支持。造成这一现状的主要原因在于专利，无论是将音频数据编解码为 MP3 音频格式(这是到目前为止最为受欢迎的格式)还是执行相反过程，这都是一个受弗劳恩霍夫研究所(Fraunhofer Institute)拥有的专利技术保护的过程。出于这一原因，开源的 Firefox 浏览器选择不在浏览器中支持 MP3 文件格式，而是支持开放的 OGG 音频文件格式。

但其他诸如 Internet Explorer 和 Safari 一类浏览器并未支持 OGG，这其中的缘由含糊不清，有人猜测是因为，Microsoft 和苹果公司不希望这种被宣称从技术角度来看性能较差的开源格式“获胜”。人人都认定，第一个获得所有浏览器支持的文件格式将赢得这场格式之争，因为仅需支持一种格式是开发者很乐意见到的事情。

25.2.3　了解移动设备音频的局限性

在讨论了 HTML5 音频标签的基本情况之后，接下来要宣布一个好消息和一个坏消息。好消息是<autio>标签已获得了 iOS 和 Android 的一些当前版本的支持；而坏消息是好些实例提供的支持都是严重残缺不全的。

第一个问题是，iOS 对声音请求做了限制，声音只能通过用户发起的操作来加载和播放。此外，iOS 一次只能播放一个通道中的 HTML5 音频。在高于 2.3 的较新版本中，Android 所采取的限制措施没有这么苛刻，不过与加载和切换音频文件相关的延迟实质上意味着与 iOS 所实施的是同样的限制。

可以想象，这严重限制了移动设备的游戏音效功能，游戏在任何适当时候(如在炮弹击中敌人时)都需要播放声音效果，而不仅是响应用户的动作。

这些依然允许播放一些音频的权宜之计的功能相当有限，不过，借助“使用音效精灵”一节给出的音效精灵概念，从游戏角度来看，你仍可在移动设备上播放一定数量的音效。

在 iOS 6 中，所有这一切都会发生变化，iOS 6 将支持 Audio Data API 的一些实现。Android 也会在未来的某个时候提供对 Audio Data API 的支持，不过，截至撰写本书之时止，音效精灵是当前可采用的最好做法。

25.3　构建一个简单的桌面音效引擎

在深入探究一些必要的变通之术，以实现在移动设备上播放音效之前，先来了解在桌面上通过<audio>标签播放声音效果会涉及哪些事项，这是很值得一做的事情。

25.3.1 将 audio 标签用于游戏音效

从游戏角度看，Audio 对象存在的一个问题是，每个对象一次只能播放一个音效。这意味着若希望在几乎同一时刻播放同一音效两次(因为，比如说会存在两颗炮弹爆炸这种情况)，那么如果只使用一个音频元素，这种情况将无法实现。

作为消除这一局限性的一种方法，HTML5 游戏开发人员很快就发现，在某个音效被加载之后，若把它的来源赋予另一个不同的音频元素，该音效将不会被再次下载，而是几乎立刻就开始播放。这引出了游戏音效系统的一种设计做法，即把一些事先创建好的 Audio 对象用作播放声音效果的通道。

要实现这一做法，音效系统需要完成的一项工作是跟踪哪一个通道仍在播放音频，然后只把新的音效添加到那些并未正在播放的通道中。

25.3.2 添加一个简单的音效系统

在第 10 章中，资产加载器的代码已经提供了一些基于受支持格式来加载音频文件的功能。这意味着加载方面的代码已经构建完成，唯一还需要的是用来设置通道以及在适当情况下播放音效的代码。

这一例子把音效添加到第 11 章的 Block Break(击砖)游戏例子中。首先在 blockbreak.html 所在的目录中创建一个名为 quintus_audio.js 的新文件，将代码清单 25-1 中代码添加到其中。

代码清单 25-1：Quintus 的桌面音效系统

```
Quintus.Audio = function(Q) {
  Q.audio = {
    channels: [],
    channelMax:  Q.options.channelMax || 10,
    active: {}
  };
  // Dummy methods
  Q.play = function() {};
  Q.audioSprites = function() {};

  Q.enableSound = function() {
    var hasTouch =  !!('ontouchstart' in window);
    if(!hasTouch) {
      Q.audio.enableDesktopSound();
    } else {
      Q.audio.enableMobileSound();
    }
    return Q;
  };

  Q.audio.enableDesktopSound = function() {
    for (var i=0;i<Q.audio.channelMax;i++) {
Q.audio.channels[i] = {};
      Q.audio.channels[i]['channel'] = new Audio();
```

```
      Q.audio.channels[i]['finished'] = -1;
  }
    Q.play = function(s,debounce) {
      if(Q.audio.active[s]) return;
      if(debounce) {
        Q.audio.active[s] = true
        setTimeout(function() {
          delete Q.audio.active[s];
        },debounce);
      };

      for (var i=0;i<Q.audio.channels.length;i++) {
        var now = new Date();
        if (Q.audio.channels[i]['finished'] < now.getTime()) {
Q.audio.channels[i]['finished'] = now.getTime() +
                                    Q.asset(s).duration*1000;
          Q.audio.channels[i]['channel'].src = Q.asset(s).src;
          Q.audio.channels[i]['channel'].load();
          Q.audio.channels[i]['channel'].play();
          break;
        }
      }
    }
  }

  Q.audio.enableMobileSound = function() {
    // TODO: Add mobile support
  }
};
```

如你所见，至少就桌面而言，这一音效系统的代码是相当简短的，内容主要包括几个配置变量和哑方法，紧跟其后是 enableSound 方法，该方法检查代码是否运行在触摸设备上，然后确定加载哪个音效系统。若是桌面浏览器，代码调用 enableDesktopSound，该方法设置音频通道并把真正的 play 方法(实际播放音效的方法)添加到 Q 中。代码提供了 play 和 audioSprites 两个哑方法，这样若音效系统未被启用，游戏仍可调用这些方法。

设置音频通道的过程包括创建一个由成对的音效对象和 finished 属性构成的数组，finished 属性指明音效完成播放的时间。因为一开始没有正在播放的音效，所以所有通道的该属性都被初始化成-1。

接下来是真正的 play 方法，该方法的主要任务是查找一个开放通道，换句话说就是完成时间小于当前时间的通道；然后，从 Q.asset 中提取某个预先载入的音频文件的 src 值；再加载并播放该音频文件。为了稍加增强自身的功用性，该方法还接收一个防抖动时间(debounce time)作为参数，其作用是防止在大约为这一参数指定的毫秒时间内再次播放同一个音效。对于 play 方法可能会在很短时间内被重复调用，但只触发了一种音效的情况来说，这一做法很有用。

25.3.3 将音效添加到 Block Break 游戏

为将音响效果添加到 Block Break 游戏中，需要在 blockbreak.html 中加入对 quintus_audio.js 文件的引用，并对 blockbreak.js 做出三处改动。

首先打开 blockbreak.html，加入所需的<script>标签：

```
<script src='quintus.js'></script>
<script src='quintus_input.js'></script>
<script src='quintus_sprites.js'></script>
<script src='quintus_scenes.js'></script>
<script src='quintus_audio.js'></script>
<script src='blockbreak.js'></script>
```

接着打开 blockbreak.js，把 Audio 模块添加到文件顶部，并调用 enableSound()来启用音效系统。

```
$(function() {
  var Q = window.Q = Quintus()
                  .include('Input,Sprites,Scenes,Audio')
                  .setup()
                  .enableSound();
```

接下来修改 Q.Ball 中的 step 方法，在球击中球拍时播放 paddle.mp3，在球击中砖块时播放 block.mp3。

```
Q.Ball = Q.Sprite.extend({
   init: function() {
     this._super({
       sheet: 'ball',
       speed: 200,
       dx: 1,
       dy: -1,
     });
     this.p.y = Q.height / 2 - this.p.h;
     this.p.x = Q.width / 2 + this.p.w / 2;
   },
   step: function(dt) {
    var p = this.p;
    var hit = Q.stage().collide(this);
    if(hit) {
      if(hit instanceof Q.Paddle) {
        Q.play('paddle.mp3',500);
        p.dy = -1;
      } else {
        Q.play('block.mp3');
        hit.trigger('collision',this);
      }
    }
```

paddle.mp3 的播放进行了防抖动处理，每隔 500 毫秒才播放一次。这是因为球有可能会连续多帧与球拍发生重叠，所以音效播放需要进行防抖动处理，防止游戏每帧都试图播放一个新的音效。另一方面，block.mp3 文件则不需要防抖动处理，因为砖块在发生碰撞之后就会被删除。

最后修改 Q.load 的调用，加载 paddle.mp3 和 block.mp3：

```
Q.load(['blockbreak.png','blockbreak.json',
        'paddle.mp3','block.mp3'], function() {
  Q.compileSheets('blockbreak.png','blockbreak.json');
  Q.scene('game',new Q.Scene(function(stage) {
    stage.insert(new Q.Paddle());
    stage.insert(new Q.Ball());
```

接下来，确保你已拥有了所需的文件，即存放在 audio/目录中的.mp3 和.ogg 文件。虽然加载方法提到的是以.mp3 结尾的方法，但引擎会自动根据你所在平台提供的支持来替换扩展名。

若在桌面浏览器中启动该游戏，现在你应会在球击中砖块和球拍时听到简单的声音效果。

如上一节所述，对于移动设备来说，这一简单机制是无效的，在下一节中，你将为移动设备添加声音效果。

25.4　构建一个移动音效系统

我最愿意看到的事情就是本节中的代码变得过时，这是一种黑客式做法，要做的也不是什么游戏非做不可的事，仅是播放一些简单音效而已。遗憾的是，这也是当前 HTML5 游戏开发者在构建移动游戏时的唯一可选做法。这不是一种很好的解决方案：iOS 上的音效不能做到很好同步，且预先加载音效是不允许的；Android 具有类似的限制。在移动设备提供 Web Audio API 之前，移动设备上的音效支持十分有限。

25.4.1　使用音效精灵

那么，解决方案是什么呢？是音效精灵。与图像精灵表类似，音效精灵的作用方式是把多个音效置于单个音频文件中，使用静音片段把它们隔开。

若希望使用一个独立的库来实现这一功能，那么 GitHub 上的 Zynga 点唱机库 https://github.com/zynga/jukebox 是一个选择。这是一个在移动设备上播放音频的库，与你添加到 Quintus 中的代码有着同样的工作方式。点唱机库的设计目标是在尽可能多的设备上实现播放，它已测试了所有回退到 Android 1.6 中的做法，且包含了一种用于旧 Android 设备的 Flash 回退。代码清单 25-2 中的代码仅针对最新的 iOS 和 Android 设备。

为将音效精灵变成可用的，在用户与游戏首次交互时(比如说触摸标题画面)，音效系统开始预先加载一个音频。在音频可播放后，引擎开始播放该音频文件开始位置的音效，

这是一段只包含了静音的音效。然后，系统设置一个定时器，周而复始不停播放该音频文件开始部分的静音。

循环播放这一音效是因为，iOS 支持音频的自动链接：在某一音效结束后，它可在无用户交互的情况下开始播放下一音效。不过，在无其他音效正在播放时，它并不支持随意开始播放某一音效。

这意味着这一音效精灵的音频文件始终需要处于播放状态，即使要播放的仅是静音。

要触发一个真正的声音效果，系统快进至音频文件中放置该音效的点，然后设置一个定时器，只要该音效完成播放，就再播静音。因为这不是很精确的科学计算，所以两个音效之间至少要存在 1 秒钟的延迟，这很重要，这样才能确保不会有音效紧跟在该音效后面突然开始播放。

若想了解实现这一功能的代码看起来的样子，使用代码清单 25-2 中的代码填充 quintus_audio.js 的 enableMobileSound 方法存根。

代码清单 25-2：enableMobileSound 方法

```
Q.audio.enableMobileSound = function() {

  var isiOS = navigator.userAgent.match(/iPad|iPod|iPhone/i) != null;

  Q.audioSprites = function(asset) {
    if(_.isString(asset)) asset = Q.asset(asset);
    Q.audio.spriteFile = asset['resources'][0].replace(/\.[a-z]+$/,"");
    Q.audio.sprites = asset['spritemap'];
    Q.el.on("touchstart",Q.audio.start);
  }

  // Turn off normal sound loading and processing
  Q.options.sound = false;

  Q.audio.timer = function() {
    Q.audio.sheet.currentTime = 0;
    Q.audio.sheet.play();
    Q.audio.silenceTimer = setTimeout(Q.audio.timer,500);
  };

  Q.audio.start = function() {
    Q.audio.sheet = new Audio();
    Q.audio.sheet.preload = true;
    Q.audio.sheet.addEventListener("canplaythrough", function() {
      Q.audio.sheet.play();
      Q.audio.silenceTimer = setTimeout(Q.audio.timer,500);
    });

    var spriteFilename = Q.options.audioPath + Q.audio.spriteFile;
    if(isiOS) {
      Q.audio.sheet.src = spriteFilename + ".caf";
```

```
    } else {
      Q.audio.sheet.src = spriteFilename + ".mp3";
    }

    Q.audio.sheet.load();
    Q.el.off("touchstart",Q.audio.start);
  };

  Q.play = function(sound,debounce) {
    if(!Q.audio.sheet || !Q.audio.silenceTimer) return;
    if(Q.audio.activeSound) return;
    if(debounce) {
      Q.activeSound = true
      setTimeout(function() {
        Q.audio.activeSound = null;
      },debounce);
    }

    sound = sound.replace(/\.[a-z0-9]+$/,"");
    if(Q.audio.sprites && Q.audio.sprites[sound]) {
      var startTime = Q.audio.sprites[sound].start - 0.05,
          endDelay = Q.audio.sprites[sound].end - startTime;
      Q.audio.sheet.currentTime = startTime;
      Q.audio.sheet.play();
      clearTimeout(Q.audio.silenceTimer);
      Q.audio.silenceTimer = setTimeout(Q.audio.timer,
                                        endDelay*1000 + 500);
    }
  };
};
```

该方法首先使用 userAgent 的匹配方法来检查设备是否为 iOS 设备，这不是一种理想的方法(用户代理匹配从来都不是)，不过，这是为 iOS 加入一个特定于文件类型的变通做法的唯一方式。

接着是 Q.audioSprites 方法，该方法用来传入提供精灵位置和长度信息的 JSON 数据资产。它把要加载文件的名称和精灵存放在 Q.audio 对象中；此外，它还把第一个触摸事件绑定到 Q.el(画布元素)上，以此来启动音效系统，这就是让系统开始播放精灵音效的窍门所在。

接下来，它将 Q.options.sound 设置为 false，这就相当于告诉 Quintus 中的加载系统，无需在正常的 Q.load 过程中尝试加载任何音频文件。这一设置是必需的，目的是防止引擎尝试加载音频文件，因为由于移动设备对预加载的限制，这些文件永不会触发它们的 canplaythrough 回调。

Q.audio.timer 方法是默认的回调，该方法确保在没有其他音效被播放时，音频元素会继续循环播放精灵前 500 毫秒已知为静音的内容。

Q.audio.start 方法在第一个触摸事件被触发时被调用，因为是经由用户事件触发，所以

它能够设置和加载音频文件。在 canplaythrough 事件被触发时，它调用播放方法并启动静音定时器。

接下来，系统把音频文件加载到 Q.audio.sheet 这一音效精灵表中，在 iOS 设备上，虽然移动 Safari 支持一些不同的声音格式，但只有.caf 文件格式(在使用 IMA-ADPCM 编解码器编码时)可在不必使用 iTunes 实现真正播放的情况下进行本地播放。在移动 Safari 中使用 iTunes 播放音效会导致明显的延迟和声音的断续。

最后，Q.audio.start 关闭 touchstart 的回调，防止它被多次调用。

在采用与桌面相同的方式完成防抖动处理后，Q.play 方法自身的任务就是从 Q.audio.sprites 对象中找出精灵，首先，它删除所有音效文件名称的扩展名(与资产相比，精灵名称并不把文件扩展名包含在内，前者一般会这样做)，然后检查该音效是否存在。

在找到该音效后，游戏就可以计算出音效的开始时间和应有的播放时长。然后，它只是把音效精灵表的播放指针拉到音效在文件中的开始位置，然后再次调用 play 方法，强制它继续播放该音频文件。开始时间被轻微回调了 0.05 秒，因为若不这样做，一些较短的或立刻开始的音效可能被 Android 和 iOS 跳过。最后，它清除静音定时器的超时，并把它设置成在音效播放完毕后再触发。

25.4.2 生成精灵文件

若要生成游戏所需的音效合并文件和相应的 JSON 文件，可打开一个音频编辑程序，使用适当的静音间隔来手动放置一些声音效果，然后手动创建一个 JSON 文件。不过，随着事情的发展，这有可能成为一个维护噩梦。

幸而，用来为 Zynga 的点唱机生成合并文件和输出 JSON 的工具已被开发出来。因为之前添加的 Quintus 代码使用的是点唱机功能的一个子集，所以，可以使用该工具，获取工具的地址是 https://github.com/tonistiigi/audiosprite。

该库依赖 ffmpeg 这一工具，这是一个需要单独安装的工具。你可通过 ffmpeg 网站 http://ffmpeg.org/下载它，也可通过 OS X 上的 Homebrew(brew install ffmpeg)或是 Linux 上的包管理器安装它，Windows 用户则需要下载并运行安装程序。

在安装完 ffmpeg 之后，若要安装 audiosprite，可使用 NPM 并通过以下命令以全局方式进行安装：

```
npm installl -g audiosprite
```

该命令安装了一个名为 audiosprite 的命令，该命令用来生成合并在一起的音效精灵文件。要把 block.wav 和 paddle.wav 这两个文件合并到名为 audiosprites 的输出文件中，你可运行：

```
audiosprite --silence 1 --output audiosprites block.wav paddle.wav
```

该命令生成 audiosprites.caf、audiosprites.mp3 和 audiosprites.json 这三个文件(此外还有.ogg 和.m4a 文件)，其中 audiosprites.json 文件看起来与代码清单 25-3 中的内容相似。

代码清单 25-3：audiosprites.json 文件

```
{
  "resources": [
    "audiosprites.caf",
    "audiosprites.ac3",
    "audiosprites.mp3",
    "audiosprites.m4a",
    "audiosprites.ogg"
  ],
  "spritemap": {
    "silence": {
      "start": 0,
      "end": 1,
      "loop": true
    },
    "block": {
      "start": 2,
      "end": 2.03,
      "loop": false
    },
    "paddle": {
      "start": 4,
      "end": 4.04,
      "loop": false
    }
  },
  "autoplay": "silence"
}
```

因为 audiosprite 工具仅接收一个文件列表然后生成合并文件和 JSON 文件，所以很容易把它纳入构建流程中，这样就可以把音效精灵文件的生成变成一件不那么痛苦的事情。

25.4.3　将音效精灵添加到游戏

要将音效精灵添加到 Block Break 游戏中，仅需要对加载代码做出两处小改动，目的分别是加载 audiosprites.json 文件和告诉引擎使用该 JSON 文件作为音效精灵。修改 blockbreak.js 文件中的 Q.load 回调，修改后的内容如以下突出部分代码所示：

```
Q.load(['blockbreak.png','blockbreak.json','audiosprites.json',
        'paddle.mp3','block.mp3'], function() {
  Q.compileSheets('blockbreak.png','blockbreak.json');
  Q.audioSprites("audiosprites.json");

  Q.scene('game',new Q.Scene(function(stage) {
```

若在移动 iOS 或 Android 设备上运行该游戏，现在你应该会听到声音效果。在 iOS 上，声音会有些不连贯，同步效果也不是特别好，但很遗憾，在受限于媒介的情况下，这已经是所能达成的最佳结果了。

25.5 展望 HTML5 音频的未来

尽管移动设备上的 HTML5 音频目前处于一种特别糟糕的状态，不过桌面上的情况也好不到哪里去。就目前这一代浏览器而言，IE9 实际上在<audio>支持方面表现最佳。其他一些浏览器(特别是 Chrome)在 2011 年的表现实则有些退步，给<audio>标签的播放带来了一些新问题。这些问题现正慢慢被修复，从长期来看，HTML5 游戏音效的前景是光明的。

特别是，Web Audio API 给人留下了相当深刻的印象(可参阅 www.w3.org/TR/webaudio/上的规范来了解详情)不过，因为 Web Audio API 目前仅在桌面的 Chrome 和 Safari 中可用，跨浏览器支持的高性能底层音频 API 的出现还需要一段时间，所以，<audio>标签成为游戏引擎的选择构建块仍可能是时机尚未成熟的事情。

25.6 小结

本章把目光转向移动 HTML5 游戏不那么让人乐观的一面：音效。首先介绍 HTML5 <audio>标签的一些基本使用情况，然后话题一转，讨论如何把一个简单的桌面和移动设备共用音效引擎添加到 Quintus 引擎中。虽然移动浏览器中的声音功能是有限的，但只要熟知这些局限性就可以利用它，但只能是利用它来增强游戏体验而非把它当成移动游戏核心功能加以依赖。

第Ⅶ部分

游戏引擎和应用商店

- 第 26 章：使用 HTML5 游戏引擎
- 第 27 章：瞄准应用商店
- 第 28 章：挖掘下一个热点

第26章

使用 HTML5 游戏引擎

本章提要

- 回顾 HTML5 游戏引擎的历史
- 了解一些商用 HTML5 引擎
- 了解一些开源 HTML5 引擎

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 26 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

26.1 引言

HTML5 游戏的开发很容易上手，再加上 HTML5 作为快速构建游戏的媒介的日益普及，这最终导致了游戏引擎的大量涌现，这些引擎中既有开源的也有商用的，本章探讨其中一些最受欢迎的引擎。

26.2 回顾 HTML5 引擎的历史

虽然自 2004 年被苹果公司引入以增强 OS X 的 Dashboard 组件以来，画布元素就一直以某种形式存在着，但仅是从过去几年开始，人们才开始真正把 HTML5 看成一种游戏开发平台。造成这一现状的原因有许多，但 JavaScript 的性能局限性和高性能渲染技术的缺

乏让大部分理智的人无法萌生把 JavaScript 用于真正游戏开发的念头。

改变是从 2008 年开始的，为了应对开始出现在大量使用 Ajax 的应用(如 Google 的 Gmail JavaScript)的日益庞大的 Javascript 的处理需求，浏览器厂商开始重视性能问题。

为了帮助提高自身应用的速度，Goolge 发布了由 V8 JavaScript 引擎驱动的 Chrome 浏览器，Safari 和 Firefox 也都推出了性能大幅提升的新 JavaScript 引擎。这是一场全力以赴的 JavaScript 装备竞赛，人们开始慢慢看到，JavaScript 越来越像一门真正的语言，已可用于更大规模的应用开发。

开发者都有一种无法抑制的冲动，那就是使用他们涉及的任何媒介来编写游戏程序，JavaScript 也不例外。自 JavaScript 问世以来，使用它编写的简单游戏就随处可见，但在 2008 年，事情开始有了变化，一些很有想法的人，比如说 GameQuery(http://gamequeryjs.com/)背后的开发人员，他们开始搭建一些更大型、更复杂的游戏框架。

随着画布元素开始出现在越来越多的浏览器中，作为一种潜在的游戏开发工具，开发者把注意力转向了它。早期诸如 gameJS(http://gamejs.org/)之类的工具基于流行的 Python PyGame 库，往往会在画布之上提供一层薄薄的包装器。

在此期间，2010 年初出现了一个名为 Akihabara(www.kesiev.com/akihabara/)的游戏引擎。不过与其说它是一个完整的游戏引擎，不如说是一个松散的工具集合，Akihabara 的神奇之处在于，它几乎可运行在任何支持画布标签的平台上，这包括 iPhone、iPad、Android 和一些较新的桌面浏览器，以及任天堂 Wii 内的互联网频道(Internet Channel)。

正如 Akihabara 的意大利创建者所说的那样：

> 可从这里下载的 Akihabara 也是我的个人梦想，它是一组库、工具和配置预设，可帮助你使用 JavaScript 创建独立风格的、8/16 位时代的像素游戏，这些游戏无需任何 Flash 插件就可以运行在浏览器中；它只用到了 HTML5 功能的一个极小的子集，这一功能子集实际上在许多现代浏览器中都有提供。

以经典的独立风格低分辨率像素游戏为目标，Akihabara 击中了性能甜区，但同时显露了这样一种迹象，即在浏览器和设备最终赶上了 HTML5 游戏开发者的需要，能够为游戏提供一个高速、稳定的平台之后，游戏开发所能达成的状况。

自那以后，众多引擎纷纷登台亮相，其中既有商用的也有开源的，每一种都有自己的理念及支持平台和技术(画布、DOM 和 WebGL 等)。

JavaScript Wiki 上有一个专门用于介绍 HTML5 游戏引擎的页面：http://jswiki.org/game-engines.html，页面内容会被定期更新。

每种引擎都有着各自的目标市场和目标开发者人群，所以，选择一种符合自己需要的引擎是很重要的事情。

26.2.1　使用商用引擎

在一个完全标准和开放的平台上使用商业引擎，这似乎有些不协调，但使用商用引擎的好处明显大于开源引擎：有专门致力于引擎开发的团体，有更好更新的文档和教程(通常

如此)。

商用引擎最显而易见的缺点是它们需要付费；不过，目前一些广受欢迎的 HTML5 引擎的费用相对而言并不高(若希望有一个准确估计，那么大概相当于一两个控制台游戏的成本)，或者它们会在游戏赚钱之后才抽取收入的一定百分比，所以成本不应成为你最担心的问题。

商用引擎的主要缺点与一些被强加于开发和分发的限制相关，大多数 HTML5 引擎都是按开发者授权的，这意味着任何帮助你开发游戏的人也都需要持有许可证。其次，虽然 HTML5 的性质意味着从理论上来说，你始终可以对引擎的一些代码进行修改，但就依赖 IDE 来构建和导出游戏的引擎而言，你不太可能轻易做到修改其代码，相反，需要在引擎固有的功能范围内开展工作。

26.2.2　Impact.js

Impact 是目前最流行的商用 HTML5 引擎之一，是一位名为 Dominic Szablewski 的开发者开发的产品，也是最早一个赢得了广泛关注的商用 HTML5 游戏引擎，该引擎售价为每开发者 99 美元，可通过网站 http://impactjs.com 购买。

Impact 配备了一个名为 Weltmeister 的功能强大的关卡编辑器，该编辑器能够让你轻松创建分层的视差滚动区块地图，以及放置和编辑实体(游戏对象)。图 26-1 展示了 Weltmeister 的使用情况，这是一个随引擎一起提供的跑跳(jumpnrun)游戏演示。

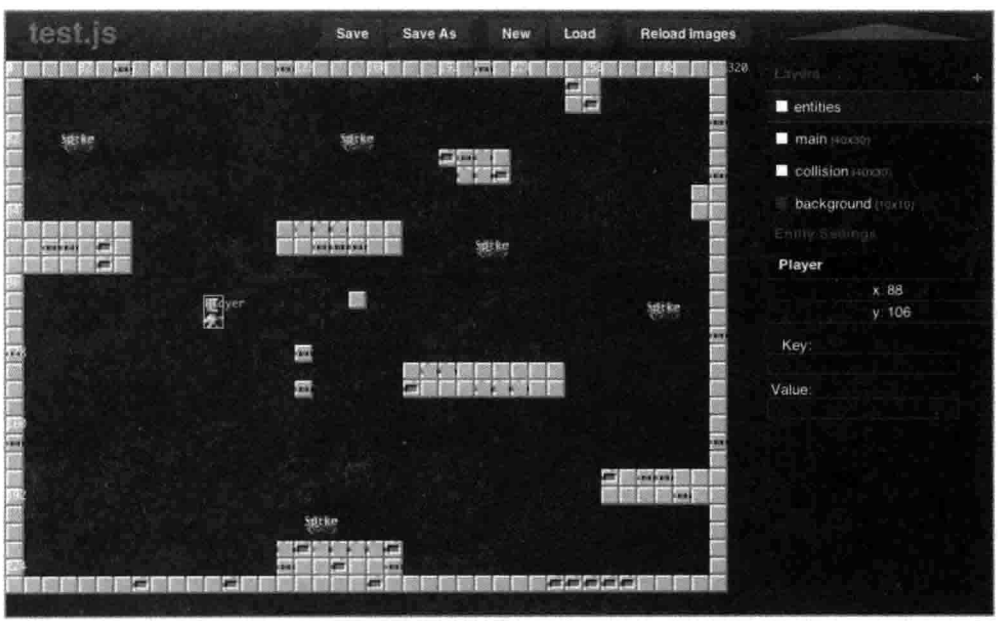


图 26-1　Impact.js 的 Weltmeister 关卡编辑器

Impact 是一个程序员游戏引擎，虽然 Weltmeister 是一个极好的关卡创建工具，但大部分的游戏都要通过在代码编辑器中编写代码和创建实体来构建。Impact 的设计基于一个经典的继承模型和一个模块系统，模块系统简化了对象间的依赖加载管理。Impact 提供了一

个打包系统，在做好游戏发布的准备之后，你可使用这一系统把游戏打包成单个用于发布的 JavaScript 文件，Impact 将这一过程称为“烘烤”(baking)。

正如你对 HTML5 游戏引擎所期待的那样，Impact 旨在以移动设备为运行游戏的平台。Szablewski 甚至创建了一种在 iOS 中构建和打包游戏的做法，使用硬件加速的 OpenGL 而非移动 Safari 的画布来运行图形。这一做法称为 iOSImpact，它支持在苹果公司的 App Store(应用商店)中发布完全用 HTML5 编写的 Impact 游戏。移动 HTML5 公司 AppMobi 已使用这一技术来构建它的开源项目 DirectCanvas，目的是允许其他游戏引擎在 iOS 上进行本地化发布，下一章会对此加以讨论。

作为一个游戏引擎，Impact 主要针对 2D 卷轴平台动作游戏进行了优化，不过理论上该引擎可用来创建任何类型的游戏。把它用于其他游戏类型仅意味着你最后要编写的代码会更多一些。只要不超出基于图像的精灵这一范围，只要你采用的是它所支持的基本图像精灵和区块地图，Impact 就能很好地抽离 HTML5 特定的渲染问题的细节。

26.2.3　Spaceport.io

Spaceport.io(http://spaceport.io)是一个混合引擎，支持在 HTML5(以及 Flash 和本地的 iOS 和 Android 应用)上运行的矢量图形。它基于 JavaScript，但拥有一个由 Flash 启发而来的 API，通过在转换器中运行.SWF 文件来加载其中的矢量资产。

若现有一些 ActionScript 3.0 游戏，那么可以通过转换器运行它们，这样基本可把游戏代码转换成 JavaScript 编写的代码，但一些额外的手动修改仍是必需的。

Spaceport.io 的很大一个卖点是它与 Flash 的 API 十分类似，这可以帮助许多基于 Flash 的游戏开发者实现一个飞跃，而且它在资产开发管道中提供了对 Flash 和矢量图形的支持。就当前状态而言，它在 HTML5 游戏开发方面的主要缺点之一是，动画的处理一般通过位图图形精灵表实现，这导致了单层、千篇一律的动画效果。

从授权角度看，一开始在开发阶段使用，Spaceport.io 是免费的，但在把游戏推出到市场后，引擎要求分享 10%的收入。

26.2.4　IDE 引擎

除了前面介绍的两种商用引擎之外，还有两种基于 IDE 的引擎(GameMaker HTML5 和 Scirra 的 Construct 2)可以输出移动可玩的 HTML5 游戏。这两个应用都需要下载和安装，之后就可在其中构建游戏。

GameMaker 拥有一种被称为 GameMaker Language(GML)的自定义脚本语言，该语言用来编写元素脚本。Construct 2 则宣称自身支持用户主要借助拖放操作来构建游戏，几乎不必编写代码。这两种引擎对 2D 卷轴动作游戏的支持都是最好的，但同样，它们几乎可用来构建任何你喜欢的游戏类型。

26.3　使用开源引擎

在开源方面，HTML5 引擎也如雨后春笋般涌现。虽然数量多得一本书的篇幅都无法悉数涵盖，但鉴于受欢迎程度和对移动 HTML5 游戏的支持，只有其中的几个值得在此一提。

26.3.1　Crafty.js

Crafty.js(http://craftyjs.com)是一个轻量级的 HTML5 引擎，完全基于组件和实体概念。该引擎有着非常小的空间占用率，经由缩减(Minified)和压缩(Gzipped)之后，无依赖版本的大小只有不到 90k。

使用方面，你不必定义类，只需创建实体并在其中添加组件即可。如在 Quintus 中所见，组件可增加一些额外功能，还可触发和响应事件。

此外，主对象 Crafty 的行为与 jQuery 对象类似，因为它也可用来查询拥有特定组件或组件组合的对象：

```
Crafty("Enemy");
// will return all entities with the Enemy component
```

随同 Crafty 发行的还有一些很有用的组件，其中包括了基本物理过程、基于多边形的碰撞检测、双向(平台动作游戏)、四向(纵向卷轴)和触摸控件，以及音效支持等。

可以创建新的组件，做法是调用 Crafty.c 并传入一个 init 方法和其他任何应被添加到实体中的方法。

组件拥有一个单独设立的站点：http://craftycomponents.com，该站点支持用户提交组件，这样就可以轻松做到通过网络直接加载组件。

若想了解 Crafty 的使用看起来是什么样子的，可研究一下代码清单 26-1 中的代码。这段代码在游戏区域中创建一个白色方框，并添加重力和支持跑跳的双向控件；然后，它在玩家的下方创建一个蓝色地板对象，用以支持玩家在上面跑动。

代码清单 26-1：一个使用 Crafty 创建的简单例子

```
<html>
<head>
 <script src="jquery.min.js"></script><script src="crafty-min.js"></script>
</head>
<body>
<script type="text/javascript">
$(function() {

  Crafty.init(640,480).canvas.init();
  Crafty.background("black");

  // Create the player object with some initial components
```

```
    var player = Crafty.e("2D, Canvas, Color, Player, Physics")
        .color("white")
        .attr({w:50, h:50, x:126, y:0});

    // You can also add additional components after the fact
    player.addComponent('Gravity').gravity("Floor");
    player.addComponent("Twoway").twoway(5,50);
    player.addComponent("Collision");

    var floor = Crafty.e("2D, Canvas, Color, Collision, Floor")
            .color("blue")
              .attr({h:30, w:400, x:0, y:380 })
});
</script>

</body>
</html>
```

如你所见，Crafty 能在不必创建复杂类层次结构的情况下完成游戏的编码工作。要运行该例子，你仅需把该段代码置于一个已加载了 crafy.js 库的 HTML 文件中即可。

由于使用组件-实体系统而具备的灵活性，Crafty 是一个适用于大多数游戏类型的通用引擎。若使用内置的组件，那么它能为除了 2D 平台动作游戏之外的其他所有游戏类型的开发提供最佳支持，对于 2D 平台动作游戏来说，它需要一些额外代码来处理与平台的正确交互。因为提供了一个先进的基于凸多边形的碰撞检测系统，它能够很好地用于诸如 2D RPG 游戏一类的垂直卷轴游戏的开发，这类游戏需要更多的碰撞处理，而不仅是简单的基于区块的玩法。

要了解更多关于 Crafty 的信息，可访问网站 http://craftyjs.com/。Crafty 采用双重授权模式，同时遵循 MIT 和 GPL 许可协议，这意味着可以使用该引擎构建任何类型的商用或开源游戏。

26.3.2　LimeJS

作为一个引擎，LimeJS 明确把自身描述成一个“用于为所有现代触摸屏和桌面浏览器构建快速且拥有本地化体验的游戏的 HTML5 游戏框架”。相对于 Crafty，LimeJS 是一个功能更全面的框架(这既是好事也是坏事)，它把 Google 的闭包库(https://code.google.com/p/closure-library/)用作依赖解决方案和自己的事件系统。此外，它还配备了 Google 的闭包编译器、Box2d JS 的一个闭包优化版本和 Closure Templates(闭包模板)。

正如你对这一规模的框架所预料的那样，在开始入手时，LimeJS 的使用要比 Crafty 复杂一些，后者仅涉及一个文件。

LimeJS 用到一个名为 Director 的主对象，该对象充当主协调对象，运行一个主定时循环。游戏每个单独的关卡或画面被称为一个 Scene(场景)，每个场景都有许多 Layer(层次)对象，每个 Layer 对象都可包含任意数量的 Node(节点)对象。诸如 Sprite(精灵)、Circle(圆)、Label(标签)和 Polygon(多边形)一类继承自 Node 的类就是真正被放到屏幕上的对象。

使用一个四级层次结构来把对象放到屏幕上显示，虽然这看起来有些复杂，不过，在构建游戏时，这其中的每一层都提供了一个很有用的抽象。

要了解使用 LimeJS 编写的简单例子看起来是什么样子的，可研究一下代码清单 26-2 中的代码。该例子在你触摸的屏幕位置上绘制一个圆形，并让该圆随着触摸或鼠标移动，在松手后，圆形会逐渐变大及显现。

LimeJS 无需任何额外代码就能很好地处理多触摸，所以，可以一次使用多个手指进行触摸和拖曳。

代码清单 26-2：一个 Lime.js 例子

```
goog.provide('movingballs');

goog.require('lime.Director');
goog.require('lime.Scene');
goog.require('lime.Circle');
goog.require('lime.animation.Spawn');
goog.require('lime.animation.FadeTo');
goog.require('lime.animation.ScaleTo');
goog.require('lime.animation.MoveTo');

movingballs.start = function(){
  var director = new lime.Director(document.body,1024,768),
      scene = new lime.Scene();
  director.makeMobileWebAppCapable();
  goog.events.listen(scene,['mousedown','touchstart'],function(e){
    var circle = new lime.Circle()
                     .setSize(50,50)
                     .setFill(Math.floor(Math.random()*255),
                              Math.floor(Math.random()*255),
                              Math.floor(Math.random()*255));
    scene.appendChild(circle);
    circle.setPosition(e.position.x,e.position.y)
          .setOpacity(0.5);

    e.swallow(['mousemove','touchmove'],function(e) {
      circle.runAction(
        new lime.animation.MoveTo(e.position)
                         .setEasing(lime.animation.Easing.LINEAR)
      );
    });

    e.swallow(['mouseup','touchend'],function(e){
      circle.runAction(new lime.animation.Spawn(
        new lime.animation.FadeTo(1),
        new lime.animation.ScaleTo(1.5)
      ));
    });
  });
```

```
    director.replaceScene(scene);
};
```

该例子是 helloworld 例子的一个修改版本，可通过命令行使用命令 bin/lime.py create helloworld 来生成一个 helloworld 例子。与 Crafty 相比，使用 Lime 搭建和运行例子的过程会更复杂一些，因为需要先下载 LimeJS，运行命令 bin/lime.py init 下载依赖，然后创建项目的目录。

如你所见，因为 Google 的闭包库用到了大量名称空间，所以事事都稍显繁琐，但也因为该库的功能强大，所以代码仍能保持相当的紧凑性。

有两件事 LimeJS 做得很好，值得在此一提。首先是它用来跟踪触摸的机制，e.swallow 代码很智能，只会作用在引发原始事件的同一触摸上，它能够在内部跟踪触摸标识符，无需你的过问。

LimeJS 提供的动画系统简化了即发即弃式(fire-and-forget)动画的设置(类似 jQuery 的做法)；此外，它还通过 lime.animations.Spawn 合并这些动画，让它们并发运行。

LimeJS 遵从 Apache License 许可协议，只要提供了适当的归属说明，你就能够把它用于任何个人或商业目的。要了解更多关于 LimeJS 的信息或下载该引擎，可访问 www.limejs.com。

26.3.3　EaselJS

EaselJS 是一个在构建 Microsoft 赞助的游戏 Pirates Love Daises(www.pirateslovedaisies.com)过程中被创造出来的框架，它处在一个有趣的位置上，因为它实际上并不是一个游戏引擎，而是一个用来把画布变得较容易使用的框架。它提供了一个场景图、一些基类和一些能把画布的使用变得更简单和更富有成效的实用方法。

若认为 EaselJS 的目标就是创建一个类似 Flash 那样的脚本环境，那么这种想法不算离谱。它提供了一个表现类似 Flash 舞台的 Stage 对象，一个行为很像 Flash 图形的 Shape 对象，以及一个与 Flash 的影片片段相似的用来跟踪动画帧的 MovieClip 对象。

与 Flash 类似，EaselJS 不会配备许多游戏特定的功能，所以诸如区块地图、物体的物理过程一类东西都需要你来添加。不过，因为有着如此之小、如此重点突出的框架 API，所以 EaselJS 很容易上手和使用。

此外，因为 EaselJS 被包装在单一的 JavaScript 库中，且是 CDN 托管的，所以可以轻松把 EaselJS 加入到自己的项目中，仅需在 HTML 中添加一个<script>标签即可：

```
<script src="http://code.createjs.com/easeljs-0.4.1.min.js"></script>
```

若想了解 EaselJS 代码看起来的样子，可参考代码清单 26-3。该例子添加一个旋转的、会在屏幕上窜下跳的弹跳球，在你单击或触摸该球时，它会被朝着一个新的随机方向抛出去。

虽然在这一例子中，EaselJS 看起来并没有像 LimeJS 那样提供了许多功能，但它确实做了不少工作。它使用 Graphics 对象随意绘制一个球形，并对球形执行精确到像素级的击中检测处理；它处理球形对象的缩放和旋转；此外，它还借助 Ticker 对象，按照具体所需

的 FPS(Frames Per Second，每秒帧数)来处理主循环的运行。

代码清单 26-3：EaselJS 例子

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
  <script src="http://code.createjs.com/easeljs-0.4.1.min.js"></script>
  <meta name='viewport' content='width=device-width, user-scalable=no'>
</head>
<body>
  <canvas id='canvas' width='320' height='480'></canvas>
  <script>
    var canvas, stage, graphic, ball;

    canvas = document.getElementById("canvas");
    stage = new Stage(canvas);
    Touch.enable(stage);

    graphic = new Graphics();
    graphic.setStrokeStyle(1);
    graphic.beginStroke(Graphics.getRGB(0,0,0));
    graphic.beginFill(Graphics.getRGB(255,0,0));
    graphic.drawCircle(0,0,25);
    graphic.lineTo(0,0,0,25);

    ball = new Shape(graphic);
    ball.x = 50;
    ball.y = 50;
    ball.vx = 100;
    ball.vy = 1000;
    ball.pulse = 0;

    ball.onPress = function() {
      var direction = Math.random()*Math.PI*2;
      ball.vx = Math.cos(direction) * 200;
      ball.vy = Math.sin(direction) * 200;
    }

    stage.addChild(ball);
    window.tick = function(dt) {
      var seconds  = dt / 1000;

      ball.vy += 50 * seconds; // Add some gravity
      ball.x += ball.vx * seconds;
      ball.y += ball.vy * seconds;
      ball.pulse += seconds;
      ball.scaleX = 1 + Math.sin(ball.pulse)/2;
```

```
        ball.scaleY = 1 + Math.sin(ball.pulse)/2;
        ball.rotation += ball.vx * seconds;

        if(ball.x + 25 > canvas.width) {
          ball.vx = -Math.abs(ball.vx);
        } else if(ball.x - 25 < 0) {
          ball.vx = Math.abs(ball.vx);
        }

        if(ball.y + 25 > canvas.height) {
          ball.vy = -Math.abs(ball.vy);
        } else if(ball.y - 25 < 0) {
          ball.vy = Math.abs(ball.vy);
        }

        stage.update();
      }
      Ticker.setFPS(60)
      Ticker.addListener(window);
    </script>
  </body>
  </html>
```

因为库可经由 CDN 加载，所以该代码清单给出的是一个完整例子。

可以设置舞台对象，做法是传入一个画布元素，然后针对舞台调用 Touch.enable 来让它处理触摸事件。

Graphics 对象支持创建用到任意矢量图片的对象，先创建这样的一个图形对象，接着把它添加到一个名为 ball 的 Shape 对象中。

因为 ball 是一个 JavaScript 对象，所以可以给该对象添加一些额外属性。在这个例子中，x 和 y 属性是 EaselJS 内置的，而 vx、vy 和 pulse 属性是自定义的。

ball 可以拥有诸如 onPress 一类的事件处理程序，任何时候只要 ball 对象被单击或被触摸，就会调用该事件处理程序。在这个例子中，onPress 方法仅是随机生成一个方向并调整球的速度以指向该方向。

球四处弹跳的真正逻辑被置于 window 对象的一个 tick 方法中，可以给 Ticker 对象添加任意数量的监听器，这是一个被 EaselJS 用作游戏循环的对象。该对象必须响应 tick 方法，这是每一帧都要调用的方法，自上次调用之后，随着一定毫秒时间的逝去，下一帧又会调用该方法。

tick 方法借助简单的牛顿物理学来移动球，你在第 17 章中已对这一做法有所了解；不过，该方法还设置了球上下跳动的幅度大小。

EaselJS 的发布遵循 MIT 许可协议，可不受限制地用于任何商业和开源项目。它是 http://createjs.com 上提供的 JavaScript 库套件的组成部分，该套件额外提供了一些在游戏中十分有用的功能，其中的 TweenJS 提供补间和动画支持，SoundJS 简化通过 JavaScript 使用声音的做法，PreloadJS 把加载和播放声音变成简单的事情，而 Zoe 则是一个从 Flash 的

SWF 文件中导出精灵表的工具。

26.4　小结

本章介绍了最受欢迎的一些支持移动设备的商用和开源 HTML5 游戏引擎，目的是让你从理念和编码的角度感受一下，在使用其中一些流行开源库开发游戏时，大概过程是什么样的。除了所介绍的这些，可供选择的引擎和库还有许多，且每一种都有着自己的理念和游戏目标。HTML5 游戏框架领域仍处于发展的初期阶段，在这一领域中，每天都会涌现许多令人兴奋的新事物。

第27章

瞄准应用商店

本章提要

- 为 Chrome Web Store 创建应用
- 使用 CocoonJS 实现本地化
- 使用 AppMobi 的 DirectCanvas 构建游戏

从 wrox.com 下载本章的代码

本章代码可从 wrox.com 上下载，访问 www.wrox.com/remtitle.cgi?isbn=9781118301326 页面，然后单击 Download Code 选项卡即可找到下载链接。代码位于第 27 章的下载压缩包中，代码文件的名称分别依照本章各处使用的文件名称命名。

27.1 引言

仅因为你的游戏是使用 HTML5 来开发的 Web 游戏并不意味着该游戏仅能存在于 Web 中，可使用多种方法打包游戏，这样就可把它变成各种应用商店适用的。对于游戏的桌面版本，本章将展示如何把游戏发布到 Chrome Web Store 中；对于应用移动版本的打包，本章将探讨两种技术：Ludei 的 CocoonJS 和 AppMobi 的 DirectCanvas。这两种技术都支持接收 HTML5 画布游戏然后把它打包成本地化应用，本地化应用会把一般的画布渲染调用替换成硬件加速的 OpenGL ES 调用，这样仅需很少的代码改动就能极大提升 HTML5 游戏的图形性能。之后，生成的应用可分发到各种移动应用商店中，包括苹果公司的 App Store 和 Google 的 Play 等。

27.2　为 Google 的 Chrome Web Store 打包应用

Google 的 Chrome Web Store 是一个免费和付费 Web 应用的在线商城，可通过 https://chrome.google.com/webstore/访问。

Google Chrome Web Store 中的应用被 Google 称为“可安装的 Web 应用”，简单地说，它们就是一般的 Web 应用，仅是已被配置成作为 Chrome 扩展程序使用，且可通过 Chrome Web Store 进行安装。

Web Store 支持两种不同的应用：托管应用和打包应用。托管应用(hosted app)就是普通的 Web 应用，仅是在被提交到 Web Store 时附加了一点额外的元数据。另一方面，打包应用(packaged app)则会被下载到用户的计算机中，且可在无需任何额外处理的情况下离线使用。此外，如果托管应用被配置成适当使用应用缓存代码清单(cache manifest)，那么它们也可以离线方式使用。

27.2.1　创建托管应用

为 Chrome 浏览器创建用于测试目的的托管应用很简单，仅需创建一个目录来存放必需的代码清单(manifest)和图标，然后把该目录当成一个未打包的扩展程序从 Chrome 内部进行加载即可。

为详细讲解所需的步骤，接下来要做的是把托管在 github 页面 http://cykod.github.com/AlienInvasion/上的 Alien Invasion 游戏变成一个托管应用。

首先创建一个名为 invasion-app 的文件夹；接着，使用代码清单 27-1 中的内容在该目录中创建一个名为 manifest.json 的文件。

代码清单 27-1：Invasion 应用的 manifest.json 文件

```
{
  "name": "Alien Invasion",
  "description": "Save the world, you know the drill...",
  "version": "1.0.0",
  "app": {
    "urls": [
      "*://cykod.github.com/AlienInvasion/"
    ],
    "launch": {
      "web_url": "http://cykod.github.com/AlienInvasion/"
    }
  },
  "icons": {
    "128": "invasion_128.png"
  },
  "offline_enabled": true,
  "permissions": [
  ]
}
```

该代码清单文件由几个重要部分组成，第一部分是一些基本信息域：name、description 和 version。这些域都会出现在扩展程序窗口中，向用户提供一些信息。接下来是 app(应用)信息，该信息告知 Chrome 哪些 urls(URL 地址)在代码清单末端列出的权限范围内应被允许访问(这一例子不必用到额外的权限)。此外，app 信息还包括了 launch(启动)信息，因为这是一个托管应用，所以需要包含一个说明从哪里启动应用的 web_url 域。

对于 icons 域，你只需指定一个 128 x 128 像素的图标。为了与其他图标的大小保持一致，它应只填充一个大约 96 像素的方块，然后在方块四周设置一个 16 像素宽的透明边框(要了解更多细节，可参阅 Google 的图像指引：https://developers.google.com/chrome/web-store/docs/images)。

接下来，若应用支持应用缓存，那么可将 offline_enabled 键的值设置成 true，这样，在浏览器处于离线状态时，启动画面上的游戏图标就不会变灰。

最后，可要求一些应用在默认情况下能获得的额外权限，这样应用就不必在每次启动时都要询问用户。这些可提供的权限包括 background、clipboardRead、clipboardWrite、geolocation、notifications 和 unlimitedStorage。除 background 之外，其他权限应都是不言自明的。

background 权限是一种特殊权限，它能够让你在后台继续运行代码，甚至在应用没有被激活或尚未启动时也是如此。对于多玩家游戏来说，这可能很有用处，即便用户当前并不在玩游戏，游戏也能够通知用户正发生在游戏世界中的各用户的行为。访问 https://developers.google.com/chrome/apps/docs/background，可以读到更多关于后台功能的深入详尽的文档。

现在，把图标文件 invasion_128.png 复制到代码清单文件所在目录下，大功即刻告成！

可测试这一托管应用，做法是把它加载成一个未打包的扩展程序。单击 Chrome 的 Wrench 菜单，然后选择 Tools 菜单项，之后单击 Extensions 菜单项。

若尚未处于开发者模式下，单击页面右上角的 Developer Mode 多选框；接着，单击 Load Unpacked Extension 按钮，然后找出你刚才创建的文件夹并单击 Select 按钮(你要选择的是一个文件夹而非文件，所以仅需单击文件夹然后单击 Select 按钮即可)。

若一切按计划顺利进行，那么你应可在扩展页面的顶部看到带有图标的 Alien Invasion 扩展程序，如图 27-1 所示。

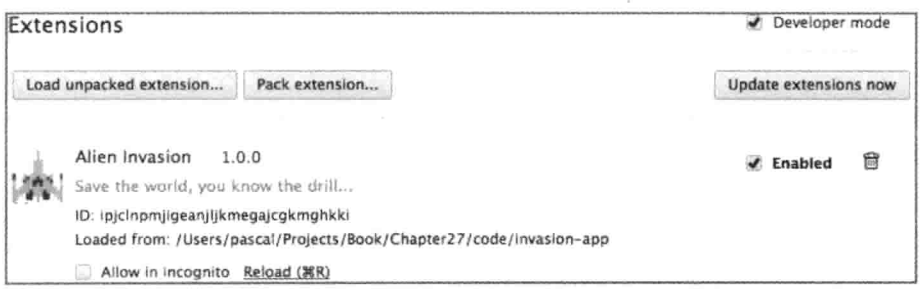


图 27-1　Alien Invasion 扩展程序

若创建一个新标签页，那么应会看到 Alien Invasion 图标出现在应用界面中，可以通过单击该图标直接跳转至游戏页面。

27.2.2　创建打包应用

托管应用和打包应用的区别仅在于打包应用在应用目录中包含了运行游戏必需的所有文件，并且是指向一个本地文件而非一个 web_url。

要创建 Alien Invasion 游戏的一个打包应用，需要创建一个新的目录，并把所有的 Alien Invasion 游戏文件都复制到其中。

接着创建一个 manifest.json 文件，如代码清单 27-2 所示，同样，把图标文件 invasion_128.png 连同一个充当应用收藏夹图标的 invasion_16.png 文件复制到该目录中。

代码清单 27-2：打包应用的 manifest.json 文件

```
{
  "name": "Alien Invasion Packaged",
  "description": "Save the world, you know the drill...",
  "version": "1.0.0",
  "app": {
    "launch": {
      "local_path": "index.html"
    }
  },
  "icons": {
    "128": "invasion_128.png",
    "16" : "invasion_16.png"
  }
}
```

如你所见，这一文件看起来与托管应用的类似，其主要不同在于 launch 部分，该部分用到了一个 local_path 而非一个 web_url。其他的唯一区别就在于应用标签页会使用一个 16x16 像素的图标来充当收藏夹图标(托管应用使用网站的收藏夹图标，所以该图像不是必需的)。

可通过单击 Chrome 扩展程序页面中的 Load Unpacked Extension 按钮来以同样方式加载该应用，然后在创建新的标签页时就可以通过应用启动界面来运行它。

27.2.3　发布应用

要发布应用，需要登录 Google 账户并访问 Chrome Web Store 的开发者信息中心：https://chrome.google.com/webstore/developer/dashboard。

然后，单击 Add New Item 链接，上传扩展程序目录中的.zip 文件。你既可上传托管应用也可上传打包应用，不过对于托管应用，需要使用 Google Webmaster 工具来证明自己是代码清单文件的 URL 段列出的所有域的拥有者。

在完成压缩文件的上传之后，你还有机会输入详细说明、上传图标和宣传图像、选择

分类和地区，以及挂接 Google Analytics。

在发布首个应用时，你还需要支付 5 美元的一次性费用，Google 借此来防止 SPAM 账户。此后，你的应用就会被发布到 Chrome 商店中，供世界各地数以百万计的 Chrome 用户轻松获取。

27.3　使用 CocoonJS 加速应用

CocoonJS 是一个由 Ludei 创建的本地化包装器，它使得你能够通过 HTML5 游戏创建本地化的 iOS 和 Android 应用。它所宣称的价值定位特别具吸引力：在无需对游戏做任何改动的情况下，你就可以把它包装成本地化应用，并且能够获得几个数量级的性能改进。

此外，还需要说明的是，CocoonJS 支持 HTML 的一个有限子集，主要通过公开一个使用 JavaScript 编写的 API 来供使用，这是一个模仿画布 API 的 API。CocoonJS 所支持功能的完整代码清单已在它的维基站点上列出，可通过访问 http://wiki.ludei.com/cocoonjs:featurelist 获得。

截至撰写本书之时为止，CocoonJS 对一些适用于游戏开发的元素，比如说画布、图像和声音等元素的 DOM 支持很有限。

27.3.1　准备把游戏载入 CocoonJS

不管 Ludei 如何宣称，根据 HTML5 游戏编写方式的不同，可能还是需要在对游戏做一些修改之后才能把它载入到 CocoonJS 中。首先要考虑的是把画布置于页面中的方式，CocoonJS 会解析你的 index.html 文件，但只是加载其中提到的 JavaScript 文件，这意味着需要确保自己是通过 JavaScript 而非 HTML 中的<canvas>标签生成<canvas>元素的。

CocoonJS 的<canvas>标签还支持一个特殊选项，可使用它把<canvas>元素放大至填满整个屏幕，同时仍保持它的纵横比不变。

要修改第 3 章中的 Alien Invasion 游戏以便能使用 CocoonJS，先修改 engine.js 中的 Game.initialize 方法，修改后的内容如代码清单 27-3 所示。

代码清单 27-3：修改后的 Game.initialize 方法

```
// Game Initialization
this.initialize = function(canvasElementId,sprite_data,callback) {

  this.canvas= document.createElement("canvas");
  // CocoonJS extension
  this.canvas.style.cssText="idtkscale:ScaleAspectFit;";
  this.canvas.width = 320;
  this.canvas.height = 480;
  document.body.appendChild(this.canvas);

  this.playerOffset = 10;
  this.canvasMultiplier= 1;
```

```
    this.mobile = true;

    this.width = this.canvas.width;
    this.height= this.canvas.height;

    this.loop();
    if(this.mobile) {
      this.setBoard(4,new TouchControls());
    }
    SpriteSheet.load(sprite_data,callback);
  };
```

该方法忽略传入的 canvasElementId，仅是创建一个新的画布元素，然后针对该元素调用 appendChild 方法，把它变成可见的。

此外，它还把特殊属性 idtkscale 的值设置为 ScaleAspectFit，目的是确保画布元素按照所希望的方式放大。

接下来，删除移动设置方法，代之以 his.mobile = true 语句，因为画布元素已不再需要进行尺寸调整。完成这一修改后，Alien Invasion 就做好了被包装到 CocoonJS 中的准备。

此外，你还应检查的另一事项是，确保自己使用 requestAnimationFrame 来处理动画，最确切地说是使用 webkitRequestAnimationFrame 这一厂商前缀版本，因为这是 CocoonJS 支持的版本。

要把这添加到 Alien Invasion 中，可修改 engine.js 中的 Game.loop 方法，修改后的内容如代码清单 27-4 所示。

代码清单 27-4：修改后的 Game.loop 方法

```
var lastTime = new Date().getTime();
var maxTime = 1/30;

// Game Loop
this.loop = function() {

  var curTime = new Date().getTime();
  webkitRequestAnimationFrame(Game.loop);
  var dt = (curTime - lastTime)/1000;
  if(dt > maxTime) { dt = maxTime; }

  for(var i=0,len = boards.length;i<len;i++) {
    if(boards[i]) {
      boards[i].step(dt);
      boards[i].draw(Game.ctx);
    }
  }

  lastTime = curTime;
};
```

这样做可以确保应用以设备可支持的最大速率进行刷新，从而赋予游戏最流畅的动画效果。

27.3.2　在 Android 上测试 CocoonJS

CocoonJS 在 Google Play 商城提供了一个名为 CocoonJS Launcher 的应用。

若打开该应用，你有两个可供选择的选项：第一个是查看一些预先创建的演示例子，第二个是启动自己的应用。要启动自己的应用，首先需要获取一个注册码。可以通过单击 Register 按钮、填写表单及确认电子邮件地址这一系列操作来获取注册码。

接下来，需要把游戏文件打包成.zip 文件，在创建.zip 文件时，确保并未把目录压缩进去而仅是压缩了其中的所有文件，因为 CocoonJS 需要在压缩文件的顶层查找一个 index.html 文件。

现在把该.zip 文件放在某个可在设备上通过 URL 对其加以访问的地方，若是在开发机上运行一个 Web 服务器，那么只要 Android 设备位于同一个 WiFi 网络中，就可以访问文件。否则，得把该文件上传到一个 Web 主机中。

进入 Android 版的 Launcher 应用界面，在 Zip URL 字段中输入.zip 文件的 URL，然后单击 Launch Current 按钮。在等待文件完成下载和启动后，你应会看到 Alien Invasion 游戏以一个提高了的帧率在手机上运行。Ludei 支持到 Android 2.2 的全面回退，所以，那些可能无法运行 HTML5 游戏的手机也可成为目标。

以 Galaxy Nexus 为例，在使用 CocoonJS 之后，游戏的帧率会从十几左右上升至大约 200FPS，所以，这带来了很显著的性能收益。

若要重启应用或加载一个新版本，需要显式地把它从运行应用列表中删除然后再次运行。

在 iPhone 上测试 CocoonJS

截至撰写本书之时为止，Ludei 只发布了它的 iOS SDK，Launcher App 尚未被批准在苹果公司的 App Store 上架。要在 iOS 上测试应用，需要通过一个复杂的过程来运行应用，该过程要求修改下载文件的 Bundle ID、在 developer.apple.com 上生成一个临时的预置描述文件(provisioning profile)，以及使用 XCode 创建一个.ipa 文件。待到本书付印时，这一过程很有可能已变得过时，所以这里不再做详细介绍，你应通过查看 Ludei 维基 http://wiki.ludei.com/cocoonjs:launcherios 上的最新详情来了解如何在 iOS 上搭建启动器运行环境。

27.3.3　构建云端应用

截至撰写本书之时为止，Ludei 尚未开放用于在云端构建大众消费的应用的云服务。等到它这样做了之后，你就可以在无须下载 XCode 或在机器上安装 Android SDK 的情况下构建本地化应用。

对于 iOS 开发而言，你仍需在 developer.apple.com 上创建一个账户，并需要通过 https://developer.apple.com/programs/ios/参加 99 美元一年的 iOS 开发者计划(iOS Developer Program)，这样才能在苹果公司的 App Store 中创建和分发应用。

27.4　使用 AppMobi 的 XDK 和 DirectCanvas 构建应用

作为 CocoonJS 的替代，AppMobi 也可用于把 HTML5 应用打包成可在 iOS 和 Android 上运行的本地化应用。AppMobi 在开源项目 PhoneGap 之上构建它的技术，该项目用到了一个使用原生设备 API 来增强自身的本地化 Web 浏览器组件，这些原生设备 API 包括音频和视频的录制，以及对设备中存储的联系人的访问等。

如上一章所述，借助 Impact.js 开发期间的 iOSImpact 创建成果，AppMobi 开发出了最初版本的 DirectCanvas，该技术使用一个自定义的 OpenGL ES 加速实现替换了标准的 <canvas>对象。

27.4.1　了解 DirectCanvas

DirectCanvas 的做法与 CocoonJS 稍有不同，它要求把游戏分成两部分：第一部分是一个连接主要的 PhoneGap Web 视图的初始 HTML 文件，Web 视图拥有一个正常的 DOM，可接受用户的输入。第二部分仅为 JavaScript，包含了在硬件加速的画布中进行绘制的游戏主体部分。不过，这第二部分不能通过绑定任何监听器来获取触摸输入，所以，需要使用一个 DirectCanvas 提供的桥接来手动把这些信息从第一部分传给第二部分。

27.4.2　安装 XDK

AppMobi XDK 是一个功能强大的 Java 应用，也是 Chrome 浏览器的扩展程序，它使得你能够在模拟器中构建和测试应用，然后把应用发送到云端进行构建，所有这一切都无需安装用来构建 iOS 和 Android 应用的开发工具的本地副本。可通过访问 AppMobile 的网站 www.appmobi.com/ ?q =node/154 了解更多信息。

要安装 XDK，请访问 Chrome Web Store(https://chrome.google.com/webstore)，然后搜索 AppMobi XDK；忽略其他搜索结果，仅安装 AppMobi HTML5 XDK。遵照安装提示，完成把扩展程序添加到 Chrome 中的过程。

XDK 的启动要求使用 AppMobi 账户登录，若没有，则按照界面上的说明创建一个新账户。

在成功进入 XDK 主窗口之后，根据所运行的例子和所选设备的不同，你应该会看到一个类似图 27-2 的画面。

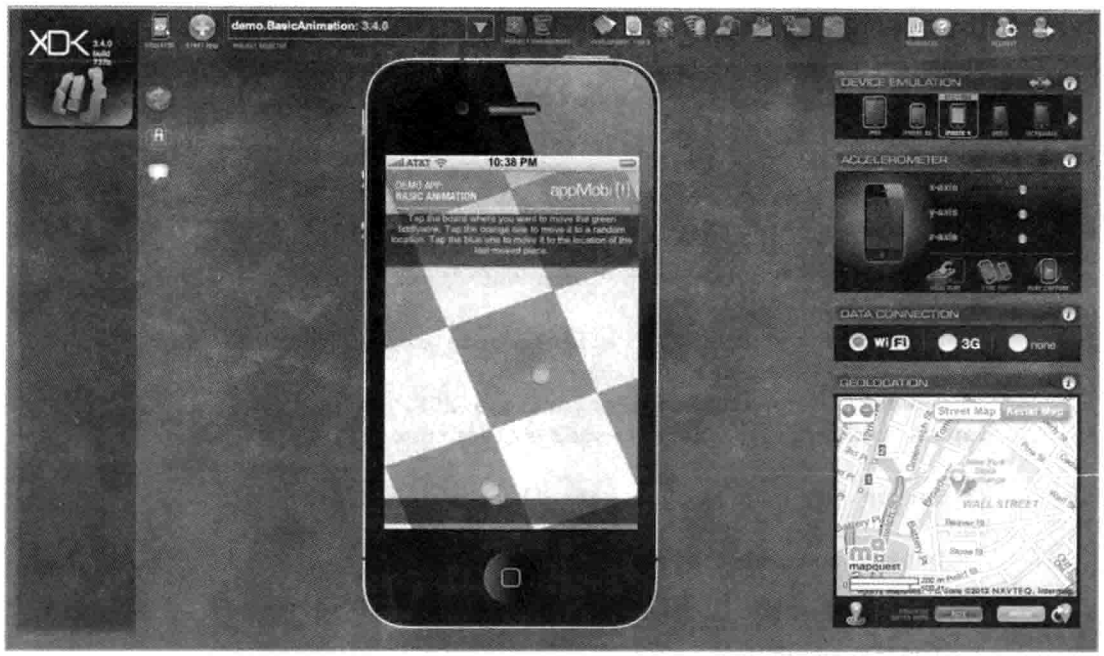


图 27-2　AppMobi XDK

XDK 支持以竖屏和横屏模式测试不同规格的设备，还支持测试地理定位和设备加速。不过这里提醒一句，用来运行所有例子的浏览器引擎仍是 Chrome 的 JavaScript 引擎和 WebKit 渲染器，所以这更多是一般行为和规格的模拟，而非设备特性或性能的真正模拟。

可以试运行各种随 XDK 发行的演示例子，做法是通过单击绿色的下拉箭头来选择不同的演示。

27.4.3　创建应用

要创建一个新应用，单击项目选择器下拉列表左边的那个大大的加号按钮；在下一个出现的界面中，选择 Client-Side 作为项目类型，然后单击 Next 按钮；接着给项目提供一个名称，比如说 alieninvasion，然后单击 Next 按钮；接着再次单击 Next 按钮跳过 API 注入界面，因为你不需要把任何这类服务用于该应用；最后，单击 Finish 按钮完成应用的创建并将其载入 XDK。

接下来，单击页面顶部的打开项目文件夹图标，该操作打开 AppMobi 希望你在其中放置游戏文件的位置。把第 3 章中的 Alien Invasion 游戏的最终代码复制到项目的新文件夹中，然后单击接近 XDK 左上角的 Reload App 按钮(该按钮看上去像是一个循环符号)。该应用会弹出一个警告，因为你删除了加载 XDK 的代码，不过若单击 OK 按钮，仍会加载游戏。

27.4.4　修改 Alien Invasion 以使用 DirectCanvas

为了让 Alien Invasion 使用 DirectCanvas，需要把用户输入从游戏主体中分离出来。

分离过程的第一步是，使用代码清单 27-5 中的代码替换游戏的 index.html 文件，这段

代码是 AppMobi 提供的样板 index.html 的一个修改版本。

代码清单 27-5：DirectCanvas 的 index.html 文件

```
<!DOCTYPE html>
<html>
<head>
<title>Alien Invasion</title>
<meta http-equiv="Content-type" content="text/html; charset=utf-8">
<style type="text/css">
    /* Prevent copy paste for all elements except text fields */
    * { -webkit-user-select:none;
        -webkit-tap-highlight-color:rgba(255, 255, 255, 0); }
    input, textarea { -webkit-user-select:text; }

    /* Set up the page with a default background image */
    body {
        background-color:#fff;
        color:#000;
        font-family:Arial;
        font-size:48pt;
        margin:0px;padding:0px;
        background-image:url('images/background.jpg');
    }
</style>
<script type="text/javascript" charset="utf-8" ➦
src="http://localhost:58888/_appMobi/appmobi.js"></script>
<script type="text/javascript" charset="utf-8" ➦
src="http://localhost:58888/_appMobi/xhr.js"></script>
<script type="text/javascript">

/* This code is used to run as soon as appMobi activates */
var onDeviceReady=function(){

    // Size the display to 320px by 480px
    AppMobi.display.useViewport(320,480)

    // hide splash screen
    AppMobi.device.hideSplashScreen();

    // Load files for Direct Canvas
    AppMobi.canvas.load("index.js");

    var keys = {}
    var trackTouch = function(e) {
      var touch, x;
      var gutterWidth = 10;
      var unitWidth = 320/5;
      var blockWidth = unitWidth-gutterWidth;
      e.preventDefault();
```

```
        keys['left'] = false;
        keys['right'] = false;
        for(var i=0;i<e.touches.length;i++) {
         touch = e.touches[i];
         x = touch.pageX;
         if(x < unitWidth) {
           keys['left'] = true;
         }
         if(x > unitWidth && x < 2*unitWidth) {
           keys['right'] = true;
         }
        }
        if(e.type == 'touchstart' || e.type == 'touchend') {
         for(i=0;i<e.changedTouches.length;i++) {
           touch = e.changedTouches[i];
           x = touch.pageX;
           if(x > 4 * unitWidth) {
             keys['fire'] = (e.type == 'touchstart');
           }
         }
        }
        AppMobi.canvas.execute('Game.setKeys('+ keys["left"] + ","
                                              + keys["right"] + ","
                                              + keys["fire"] + ")");
    };
    document.addEventListener('touchstart',trackTouch,false);
    document.addEventListener('touchmove',trackTouch,false);
    document.addEventListener('touchend',trackTouch,false);
    document.addEventListener('touchcancel',trackTouch,false);
};

document.addEventListener("appMobi.device.ready",onDeviceReady,false);

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

所有位于 onDeviceReady 方法之前的代码都是标准的 AppMobi 样板代码，这部分代码设置一些默认的样式并加载 AppMobi 的 JavaScript。

该应用设置一个适当尺寸的视口、隐藏启动画面，并把 index.js 文件加载到 DirectCanvas 中。因为 DirectCanvas 是一个完全不同于 index.html 的执行环境，没有变量或方法被渗漏出来，所以需要显式执行在其他上下文中运行的代码来往返运送信息。

游戏主体中的 touch 方法已从 engine.js 移至 index.html，这是因为所有的输入都需要集中放在 index.html 文件中，而这又是因为 DirectCanvas 的上下文无法接收任何的用户输入。

在 touch 方法的末尾处，可以看到一个用来把数据传递给画布上下文的调用：

```
AppMobi.canvas.execute('Game.setKeys(' + keys["left"] + ","
                                       + keys["right"] + ","
                                       + keys["fire"] + ")");
```

其中的 setKeys 方法是一个需要添加到 engine.js 中的新方法。

下一步，创建代码清单 27-5 中引用的 index.js 文件，该文件只有两行内容，如代码清单 27-6 所示，其作用是告诉文件，把 engine.js 和 game.js 文件加载到上下文中。

代码清单 27-6：index.js 文件

```
AppMobi.context.include( 'engine.js' );
AppMobi.context.include( 'game.js' );
```

现在剩下要做的就是修改 engine.js，让它使用 AppMobi 的 DirectCanvas，以及通过 setKeys 方法而非绑定事件来取得输入值。此外，你还应修改循环方法，这次同样是使用 requestAnimationFrame。最后，在所有一切都绘制完毕后，你还需要显式调用 context.present ()来把它们绘制到屏幕上。

修改 engine.js 顶部的内容，如代码清单 27-7 中突出显示部分代码所示，除了对初始化方法的一些修改和一个新的 setKey 方法之外，其中还加入了 requestAnimationFrame 的衬垫代码(shim)。

代码清单 27-7：为 DirectCanvas 所做的 engine.js 修改

```
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0;
        x < vendors.length && !window.requestAnimationFrame;
        ++x) {
        window.requestAnimationFrame =
                window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] ||
                window[vendors[x]+'CancelRequestAnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() {
                            callback(currTime + timeToCall);
                    }, timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
```

```
            clearTimeout(id);
        };
}());

var Game = new function() {
  var boards = [];
  // Game Initialization
  this.initialize = function(canvasElementId,sprite_data,callback) {

    var ctx = this.ctx = AppMobi.canvas.getContext("2d");

    this.ctx.width = 320;
    this.ctx.height = 480;

    this.playerOffset = 10;
    this.canvasMultiplier= 1;
    this.mobile = true;

    this.width = 320;
    this.height = 480;
    this.loop();
    this.setBoard(4,new TouchControls());

    SpriteSheet.load(sprite_data,callback);
  };

  this.keys = {};
  this.setKeys = function(l,r,fire) {
    Game.keys['left'] = l;
    Game.keys['right'] = r;
    Game.keys['fire'] = fire;
  };

  var lastTime = new Date().getTime();
  var maxTime = 1/30;
  // Game Loop
  this.loop = function() {
    var curTime = new Date().getTime();
    requestAnimationFrame(Game.loop);
    var dt = (curTime - lastTime)/1000;
    if(dt > maxTime) { dt = maxTime; }

    for(var i=0,len = boards.length;i<len;i++) {
      if(boards[i]) {
        boards[i].step(dt);
        boards[i].draw(Game.ctx);
      }
    }
    Game.ctx.present();
    lastTime = curTime;
  };
```

requestAnimationFrame 的衬垫代码与你在第 9 章中见到的是一样的。

在 Game.initialize 方法中，作为创建画布元素的替代，需要使用 DirectCanvas 从 AppMobi.canvas 元素中抓取上下文。此外，这一上下文还可被直接调整尺寸——一般来说你只能针对 canvas 元素这样做。Initialize 方法余下部分代码仅是默认当前设备为移动设备，并把一个宽度和高度硬编码到了游戏中。

setKeys 方法取代了之前存在的输入处理程序，并且基于 index.html 发送过来的信息，显式设置被激活的按键。

最后一个很重要的改动是，循环方法显式调用 Game.ctx.present ()来渲染屏幕。

27.4.5　在设备上测试应用

AppMobi 简化了移动设备上的游戏测试，它为 iOS 和 Android 提供了一个 AppLab 应用，可以通过各自相应的应用商店下载。在设备上完成了该应用的安装之后，就可启动它。AppLab 使得你能够运行那些已被同步到云端的应用(单击右上角的 My Apps 这一小按钮)。

要把应用同步到云端，单击 XDK 顶部工具栏中的 Test Anywhere 按钮。在完成应用的同步后，你应在设备上运行这一激活版本(要小心，因为截至撰写本书之时为止，DirectCanvas 的 Android 版本仍处于测试阶段，仍存在一些可视缺陷，到本书付印时，这些缺陷有望得到修复)。

在云端构建应用

若应用已经能够在 AppLab 中正常运行，那么是时候把它作为产品推出了。若要实现这一点，可单击 XDK 顶部工具栏中的 Build for App Store 按钮，为 iOS 或 Android 构建应用的过程涉及了一系列的步骤，AppMobi 会很好地引导你走完这一个过程。

为 iOS 构建应用的过程尤其痛苦，涉及了通过应用开发者门户 developer.apple.com 创建并下载证书和预置描述文件。每年需要缴纳 99 美元的会员费，加入 https://developer.apple.com/programs/ios/上的 iOS 开发者计划，这样才能完成这一过程。在做完这些事情之后，你会得到一个用于 iOS 的.ipa 文件或一个用于 Android 的.apk 文件，可以把该文件安装在设备上，这样就可以测试最终的应用。

27.5　小结

本章向你展示了如何以不同方式打包 HTML5 游戏，借以把它们部署到各个应用商店中，首先讨论托管应用和打包应用这两种 Chrome Web Store 应用的创建，接着研究了两种通过 HTML5 应用创建本地化移动游戏的平台：CocoonJS 和 AppMobi。现在，移动 HTML5 游戏不再仅是为浏览器构建，它们还可被本地化部署到几乎所有的 iOS 和 Android 设备上。如你将在下一章中见到的那样，许多很酷的新技术正在接连涌现，但本地化的性能和功能将始终至少略微领先于浏览器，所以，把 HTML5 游戏打包放到应用商店中，这一做法只会变得越来越普遍。

第28章

挖掘下一个热点

本章提要

- 介绍使用 WebGL 实现浏览器 3D
- 预览一些即将出现的 API
- 展望 WebAPI 提供的本地化支持的未来

28.1 引言

本书涵盖了许多方面的内容，但 HTML5 领域发展迅速，许多尚未在移动浏览器中获得普遍使用的尖端技术规范都值得纳入考虑范围，因为它们会扩充所能开发的移动 HTML5 游戏的类型，这包括在浏览器中通过 OpenGL ES 直接访问硬件加速的 3D，通过 Web Audio API 获得更好的声音支持，以及访问其他一些本地化的硬件功能等。本章将介绍这些最前沿的技术规范。

28.2 使用 WebGL 实现 3D

基于画布的游戏的最大缺点之一是,它们只能停留在二维的平面世界中。当然，可在 2D 画布之上构建自己的 3D 渲染和光栅处理引擎，不过性能可能不会令人满意。

幸而，所需的帮助很快就能到位。WebGL，这个 OpenGL ES 的 Web 版本，是一个在浏览器中提供硬件加速 3D 支持的规范。OpenGL ES 是 OpenGL for Embedded System(嵌入式系统的 OpenGL)的简写，它是桌面 OpenGL 标准的一个更小、更功耗友好的表亲。过去 20 年来，桌面 OpenGL 标准一直是各种类型的 3D 软件(包括游戏)的驱动力。

OpenGL ES 已是可通过 iOS 和 Android 上的本地化 API 访问的，你在应用商店中见到

的一些 3D 游戏都由它来驱动。通过 WebGL 公开一个基于 JavaScript 的 API，这意味着 HTML5 应用可在无插件的情况下在浏览器中创建精致的 3D 场景和游戏。要在受支持的浏览器中访问 WebGL 画布，你只需创建一个标准的画布元素，然后请求一个 WebGL 上下文而非一个标准的 2D 上下文即可。

注意：WebGL 是一个由 Khronos 小组提出的标准，可在 Khronos.org 网站上找到最新的标准细节：www.khronos.org/webgl/。

依赖一些使用一种基于 C 语言、名为 WebGL Shader Language(GLSL)的专用着色器语言编写的着色器程序，WebGL 提供了一个生成 3D 场景的底层 API。除非拥有编写 GPU 着色器程序的经验，否则在开始使用时，WebGL 可能会有点让人望而生畏。

幸而，一个名为 Three.js 的很受欢迎的库提供了一个高层抽象，该抽象层能让你着手构建浏览器中的 3D 又无需为着色器烦心。可通过访问 http://mrdoob.github.com/three.js/获取 Three.js，还可浏览该站点上的大量例子。

虽然 Three.js 简化了 WebGL 的使用，但 3D 编程仍有可能是相当复杂的。现有的一些项目能够把使用 Three.js 进行开发变成一件较容易的事，其中最受欢迎的项目是 tQuery：http://jeromeetienne.github.com/tquery/。

tQuery 的目标是在 Three.js 上提供一个类 jQuery 的接口，简化浏览器中的 3D 入门和使用。除了 WebGL 外，Three.js 还支持其他一些 3D 渲染器，这样一些简单的例子就可以运行在移动设备上，不过较为复杂的例子还需要使用 WebGL 渲染器。

从移动角度看，问题的主要症结在于只有一种移动浏览器：Opera 12 支持 WebGL 的一般性使用。在桌面上，除了 Internet Explorer 之外，其他所有浏览器的当前版本都支持 WebGL，这其中包括 Firefox、Chrome、Safari 和 Opera 等。遗憾的是，Microsoft 尚未承诺在未来的某一时间点提供 WebGL 支持，不过，鉴于 WebGL 在开发者中的受欢迎程度，对于 Microsoft 来说，若继续坚持这一立场，那么这看起来就有点像是浏览器的自杀行为。

虽然 WebGL 在移动 Safari 中尚未可用，但在 iAd 中已经被开启，所以，支持已是内置的，只是被禁用了而已。若拥有一个已成功越狱的 iPad，那么可找到一些资源来把 WebGL 支持打开。Android 方面，WebGL 出现在各种地方，如 Xperia 智能手机和 Xoom 平板电脑的演示等，但目前还没有在 Android Chrome 中获得支持的时间表。

28.3　使用 Web Audio API 获得更好的声音访问

如你在第 25 章中所见，在移动设备上，HTML5 的声音并未获得很好的支持。好消息是，一个名为 Web Audio API 的功能极其丰富的音频 API 已经进入了 Chrome 和 Safari 的桌面版本中。该 API 可能有点让人望而生畏，因为它处于底层却又功能丰富，是一种游戏开发者(而非 Web 开发者)更惯于使用的 API。要了解最新公布的规范，可访问 www.w3.org/

TR/webaudio。

若能获得更多浏览器的支持，Web Audio API 就能为 HTML5 游戏提供一个功能强大的音频层，该音频层支持通过 JavaScript 来实时创建、混录音效和音乐。

截至撰写本书时为止，苹果公司已宣布 iOS 6 将支持该音频 API，但支持的级别和对播放音频的限制仍有待确定。因为 Web Audio API 是一个 Google 率先在 Chrome 中实现的项目，所以该 API 似乎很可能在不久的将来出现在 Android Chrome 中。

在撰写本书之时，Microsoft、Mozilla 和 Opera 都还没有宣布支持该 API 的任何意向。

28.4　使用全屏 API 扩大游戏画面

由于移动设备提供的屏幕较小，所以，任何能最大化游戏的屏幕实际使用面积的做法都能增强游戏的可玩性。

在桌面上，Firefox、Chrome 和 Safari 都已增加了对全屏 API 的支持，该 API 能够让你指定一个应被全屏显示的 DOM 元素。目前，这一 API 的规范仍处在不断变化之中，http://dvcs.w3.org/hg/fullscreen/raw-file/tip/Overview.html 上提供了一个工作草案。

当前桌面浏览器仅以带厂商前缀方式提供了该 API，尚未有移动浏览器发表支持声明，不过苹果公司就已宣布 iOS 6 将拥有一个全屏的横屏模式。该模式不同于全屏 API，不过应仍可以给游戏开发者带来一些额外的屏幕空间。

28.5　使用屏幕方向 API 锁定设备屏幕

因为设备的转动在经过某个点之后，屏幕就会发生旋转，所以，如你在第 24 章中所见，游戏开发者利用 Device Orientation API 创造出来的一些有趣玩法会因此而大打折扣。

好消息是，存在这样的一个 W3C 规范，该规范既能让开发者更准确地捕捉屏幕方向的当前状态，又能让开发者把屏幕锁定在某个特定方向上。若想了解这一规范的工作草案，可访问 www.w3.org/TR/ screen-orientation/。

截至撰写本书之时为止，在移动设备方面，只有 Mozilla 的 Android 浏览器 Fennec 实现了该规范，不过在将来，Android Chrome 和移动 Safari 都很可能会提供对这一 API 的支持。

28.6　使用 WebBTC 添加实时通信

WebRTC 项目是一个开源项目，它的目标是通过 JavaScript API 把实时通信添加到 Web 浏览器中，这包括了通过中央服务器和通过点对点方式发起语音和视频通话的功能。对于双玩家游戏来说，该项技术实属锦上添花之作，它允许玩家在互相对战的同时(在无数其他用户中)保持通信。

该项目有一个网站，访问地址为 www.webrtc.org，此外，还可通过 www.w3.org/TR/webrtc/

访问该项目的规范草案。

目前，尚未有桌面浏览器提供对 WebRTC 的支持，不过，可通过开发版 Chrome 试用一个版本，可参阅 WebRTC 网站了解详情。

28.7　追踪其他即将出现的本地化功能

Mozilla WebAPI 项目的网络地址为 https://wiki.mozilla.org/WebAPI，该站点链接了其他一些正通过 JavaScript API 慢慢公开的设备原生功能。虽然其中的大部分功能都不会引起游戏开发者的兴趣，不过某些类似 Vibration API(www.w3.org/TR/vibration/)这样的功能可用来实现有趣的游戏反馈。

Mozilla 还有一个被称为 AreWeFunYet 的项目(网址为 www.arewefunyet.com)，该项目的设立是为了跟踪 Gecko 作为游戏平台的表现情况。虽然其中的许多细节都是 Mozilla 特定的，但该项目仍值得加以研究。

此外，W3C 还有一个游戏社区小组(Games Community Group)，它的网上站点是 www.w3.org/community/games/，该小组致力于“改善游戏开发者赖以创建游戏的开放式 Web 标准的质量”。若你对 HTML5 游戏开发的未来感兴趣，那么这是一个值得加入的小组。

28.8　小结

本章介绍了一些振奋人心的 API，虽然就目前而言，这些 API 还不能为 HTML5 移动游戏开发者所用，但在不久的将来，它们应会陆续出现在移动浏览器中。这些 API 将给 HTML5 游戏开发者带来一些工具，这些工具与本地化应用开发者手中的工具一样，可被用来以一种跨浏览器、多平台方式制作引人入胜的体验。

在展望了这些已出现在地平线上的功能之后，本书的 HTML5 移动游戏开发世界之旅也就到了尾声。从基础开始，搭建一个简单的基于画布的游戏，乃至从无到有，构建一个以移动为中心的完整的 HTML5 游戏引擎，本书伴随你走过了一段精彩的旅程。在此过程中，该游戏引擎被用来构建了许多不同的游戏和演示例子，这些例子用到了 CSS、SVG，最重要的是用到了画布。在学习过程中，本书还向你介绍了把类似 Underscore.js 和 jQuery 这样的库用于游戏开发的做法，以及如何使用 Node.js 在服务器端运行 JavaScript。此外，你还了解了如何把游戏变成离线可用的、如何把它与 NoSQL 数据库连接起来、如何使用 Socket.io 来让玩家互相对战，以及如何使用一个托管平台把游戏推向整个互联网。

作为一种技术，HTML5 赋予了你赢取数量空前的潜在玩家的机会，任何玩家都可随时通过受支持的移动设备或桌面浏览器访问你的游戏。现在，是时候把你所学到的东西用来构建下一个精彩游戏了——这将是一个能让全世界的人都参与进来，共同娱乐的游戏，这将是一个像可跨互联网数据管道流传的链接那样可快速传播的游戏。

附录 A

资　　源

本附录包含了各种资源，强烈建议你使用这些资源来进一步探索 HTML5 和 JavaScript。

HTML5 和 JavaScript 书籍

若想了解更多关于 HTML5 和 JavaSccript 方面的内容，建议阅读以下书籍：

- *JavaScript: The Good Parts*，作者 Douglas Crockford(O'Reilly 出版，2008 年)：一部开创性作品，为 JavaScript 作为一门“真正的语言”正名，并推出了一系列开发者最佳实践，是 JavaScript 开发者的必读之作。
- *JavaScript Patterns*，作者 Stoyan Stefanov(O'Reilly 出版，2010 年)：关于 JavaScript 的第二部优秀作品，该书进一步推进了这门语言的发展，并提供了许多能让 JavaScript 迎合自己愿望的不同做法。
- *Introducing HTML5 Game Development*，作者 Jesse Freeman(O'Reilly 出版，2012 年)：首批关于单个 HTML5 游戏引擎的书籍之一，Impact.js 的一个深入(但不失简洁)指南。
- *Foundation HTML5 Canvas: For Games and Entertainment*，作者 Robert Hawkes (friendsofED 出版，2011 年)：HTML5 画布的一个较为通俗全面的介绍，适合经验较少的开发者阅读。
- *Foundation HTML5 Animation with JavaScript*，作者 Billy Lamberta、Keith Peters(friendsofED 出版，2011 年)：本书使用 HTML5 画布让物体以一种有趣方式移动。
- *HTML5 Games Most Wanted: Build the Best HTML5 Games*，作者 Egor Kuryanovich 等(friendsofED 出版，2012 年)：一本 HTML5 开发手册，其中包含了许多不同作家关于各种 HTML5 游戏开发技术的文章。

- *Making Isometric Social Real-Time Games with HTML5, CSS3, and Java Script*，作者 Mario Andres Pagella(O'Reilly 出版，2011 年)：一本专注于某种游戏类型的小书，书中内容很好地涵盖了这类游戏，提供了用来构建一批 Facebook 社交游戏所需的代码。

网络资源

若想了解更多关于 HTML5 和 JavaScript 方面的内容，建议关注以下网络资源：

- www.html5gamedevelopment.org：一个由作者本人管理的网站，跟踪 HTML5 游戏开发的最新趋势。
- www.html5gamedevs.com：一个很活跃的聚合站点，发布最新的 HTML5 游戏和游戏开发新闻。
- www.badassjs.com：关于最前沿的 JavafScript 技术的第一方深度文章。
- www.creativejs.com：关于 JavaScript 新鲜酷辣一面的第一方深度报道。
- buildnewgames.com：Microsoft 赞助的网站，关于各种 HTML5 游戏开发主题的深度文章。
- javascriptweekly.com：这不是一个网站，而是一个每周一次的时事通讯，收集了一周以来的各类 JavaScript 新闻，是 JavaScript 开发者必读的内容。
- www.html5rocks.com：网上关于 HTML5 的最权威资源之一，每当有新的 API 问世，HTML5 Rocks 往往会第一时间发表关于该 API 的深度文章。
- developer.mozilla.org：作为了解最新 HTML5 技术的最佳资源，是 HTML5rocks 网站的竞争对手。
- www.lostdecadegames.com/lostcast：HTML5 游戏开发公司 Lost Decade Games 的一个双周播客站点，常有一些特邀嘉宾活跃于该 HTML5 游戏开发社区。
- www.chromeexperiments.com：若某项功能很酷且是使用 JavaScript 编写的，那么你很有可能会发现它是一个 Chrome 实验。若希望了解如何最大限度地利用浏览器，可访问这一站点。

[General Information]
书名=HTML5移动游戏开发高级编程
作者=（美）瑞特格著
丛书名=移动开发经典丛书
页数=508
SS号=13522632
出版日期=2014.04
出版社=北京：清华大学出版社
ISBN号=978-7-302-35631-8
中图法分类号=TP312
原书定价=68.00
参考文献格式=（美）瑞特格著.HTML5移动游戏开发高级编程.北京：清华大学出版社,2014.04.
内容提要=本书说明何时使用三种基本方法(CSS3、SVG或Canvas)来创建HTML游戏。介绍用HTML5创建实时多人游戏的标准模式。介绍JavaScript游戏开发的基本知识。创建2Dplatformer并构建非传统的多人用户界面。展示大量的移动新功能，如Geolocation，Deviceorientation，accelerations和声音等。展示如何将HTML5游戏放置到appstore中。