Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

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

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
Copyright © 2012 by Pascal Rettig
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2013-4898

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

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：成风进
责任印制：

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：
装 订 者：
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：

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。

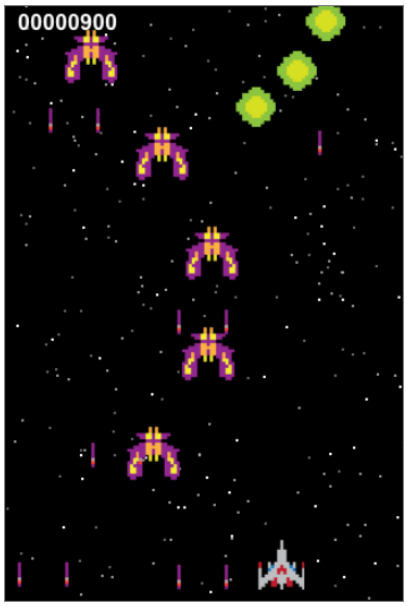

图 1-1　最终的游戏界面

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。

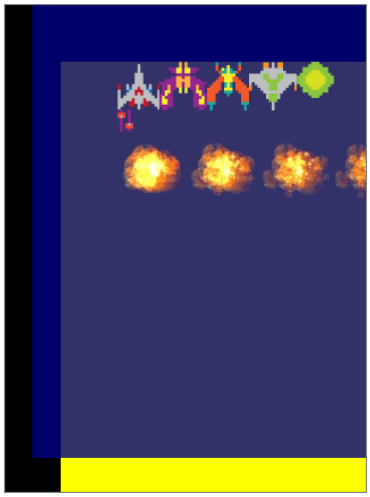

图 1-2　精灵表和绘制的矩形

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 的新文件的顶部，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。