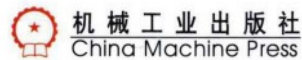
lufylegend的原理与使用

以实例为向导，详细讲解射击游戏、物理游戏、网络游戏等各类游戏的开发思路和技巧

张路斌 著

HTML5 Canvas Games Development by Example

HTML 5 Canvas 游戏开发实战



51CTO.com
技术成就梦想

Canvas是HTML5的新元素之一，用于在网页上绘制图形，相当于在HTML中嵌入了一张画布，方便直接在HTML上进行图形操作，具有极大的应用价值。如果我们掌握了使用Canvas绘图的方法，进行游戏开发时定可事半功倍。本书将会循序渐进地揭开Canvas的面纱，帮助读者快速入门，掌握Canvas的用法。书中不仅介绍了HTML5 Canvas的基础API，还重点阐述了如何在JavaScript中运用面向对象编程思想进行游戏开发。同时，本书还结合大量实例，详细讲解了如何使用HTML5 Canvas来制作各种常见类型的游戏，比如“剪刀石头布”、“俄罗斯方块”、“是男人就下一百层”等，在介绍每个游戏开发的过程中，都会涵盖游戏分析、开发过程、源码解析和小结等相关内容，帮助读者快速深入地了解每种类型游戏开发的完整步骤，让读者彻底掌握各种类型游戏的开发思路和设计技巧。

本书共包含四大部分内容：

- 第一部分（第1章）是准备工作，介绍一些HTML5的发展历史和开发环境的搭建。
- 第二部分（第2～4章）是基础知识介绍，详细讲解HTML5 Canvas的基础知识以及开源库件lufylegend的使用方法。
- 第三部分（第5～10章）是开发实战篇，通过实例为大家讲解如何进行HTML5的游戏开发，其中的游戏示例涉及各种常见的游戏类型，包括休闲、射击、Box2d及网游等多个领域。
- 第四部分（第11章）是提高篇，运用数据来分析如何提高程序的效率问题。

上架指导：计算机/程序设计/Web开发
ISBN 978-7-111-41912-9
定价：69.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

HTML5 Canvas Games Development by Example

HTML 5 Canvas
游戏开发实战

张路斌 著

TP312 HT
94

图书在版编目（CIP）数据

HTML5 Canvas 游戏开发实战 / 张路斌著．—北京：机械工业出版社，2013.4
（实战系列）

ISBN 978-7-111-41912-9

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

中国版本图书馆 CIP 数据核字（2013）第 057804 号

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

本书主要讲解使用HTML5 Canvas来开发和设计各类常见游戏的思路和技巧，在介绍HTML5 Canvas相关特性的同时，还通过游戏开发实例深入剖析了其内在原理，让读者不仅知其然，而且知其所以然。在本书中，除了介绍了HTML5 Canvas的基础API之外，还重点阐述了如何在JavaScript中运用面向对象的编程思想来进行游戏开发。

本书在介绍每个游戏开发的过程时，都会包括游戏分析、开发过程、代码解析和小结等相关内容，以帮助读者了解每种类型游戏开发的详细步骤，让读者彻底掌握各种类型游戏的开发思想。最后，还通过数据对比分析，指导读者提升程序的性能，写出高效的代码，从而开发出运行流畅的游戏。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：孙海亮
藁城市京瑞印刷有限公司印刷
2013年4月第1版第1次印刷
186mm × 240 mm • 21 印张
标准书号：ISBN 978-7-111-41912-9
定价：69.00 元

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

前　言

为什么要写这本书

并非计算机专业的我，却最终走上了编程之路，并写了这样一本书，为什么呢？其实一切都是因为和游戏结了缘。

小时候我非常喜欢玩游戏，为了玩游戏和小伙伴们干过不少调皮捣蛋的事情。初中为了得到自己的第一台游戏机，和父亲打赌，破天荒拿了全班第一名。当然有了游戏机的相伴，从那以后就再也没有拿过第一名了。因为数学上较有优势，所以读大学时选择的是数学专业，没有选读计算机让后来做开发的我多少感到有些遗憾。和其他人一样，大学是真正改变我人生的时期，第一次有了电脑，第一次从室友嘴里得知 QQ 为何物（这个大土包子），第一次接触了网游（还是大土包子），那时候非常痴迷于光荣公司的《三国志英杰传》和《三国志曹操传》，并且通过学习，试着制作了我人生中的第一款游戏《杨家将传奇》，在同类游戏中小有名气。如果你也喜欢玩《三国志曹操传》的 MOD，也经历过 2003 年到 2007 年那段《三国志曹操传》MOD 最鼎盛的时期，那么你也许会认识我，那时候我有个网名叫“回眸 75 度”。那时候制作的游戏因为不涉及编程，所以还不能叫做开发。大学毕业后，我开始思考自己未来的路，虽然没有任何编程基础，但是因大学时期游戏的制作，令我坚信游戏开发是我向往的职业。也是因为喜欢日本的游戏，最终来到了日本做软件开发，虽然并非全职的游戏开发者，但是我一直都在关注并学习着游戏的开发，不曾间断，一直至今。

我是一个喜欢不断学习新知识的人，所以当 HTML5 作为一个新技术出现的时候，我没有理由不去了解它。由于对 JavaScript 有一定的基础，所以我在学习 HTML5 的 Canvas 时，上手非常快。出于对 ActionScript 的喜爱，我一开始便试着在 JavaScript 中模仿 ActionScript 的 API 来做开发，并且在博客上发表了《用仿 ActionScript 的语法来编写 HTML5》系列文章，这便是最初的 lufylegend 开源库件的构建过程。当我把自己研究的类库整合到一起后，发现它使用起来十分方便，使用它来开发游戏可以节约大量的开发时间，于是我将其分享到了网上供大家免费使用，希望可以给相关开发者提供便利。

我刚开始接触 HTML5 是在 2011 年，那时候正是 HTML5 炒得最火的时候，网上到处都

是关于 HTML5 替代 Flash 的言论。我从来不认为 Flash 会因为 HTML5 而消失，但是我相信 HTML5 代表着互联网未来发展的方向。虽然 HTML5 并不像传言那样“一次编写，到处运行”，但是你只需要略加修改和调整就能让你的应用适用于另一个平台。

网页游戏有着不可替代的优势，而 HTML5 无疑是未来网页游戏的首选技术之一，特别是在移动领域 HTML5 是唯一的工具，至少现在它是你唯一的选择。虽然 HTML5 还很年轻，还有很多不成熟的地方，但是它正在不断地完善，相信它不会让我们等太久。所以学习这门新技术非常有必要。我非常希望能将自己的学习心得分享给大家，因此写了这本书，希望这本书能够将你带入 HTML5 的游戏世界里。

读者对象

本书主要适合于以下读者：

- 对 HTML5 开发感兴趣的人。本书对 HTML5 Canvas 进行了详细的介绍，想要学习 HTML5 开发的话绝对值得一看。
- 对游戏开发感兴趣的人。本书介绍了多个游戏实例的开发，在如何构建游戏方面对游戏开发人员来说有一定的借鉴作用。
- Flash 开发者。由于 JavaScript 和 ActionScript 具有一定的相似性，本书会模仿 ActionScript 的语法来进行游戏的构建和开发，如果你是一个 Flash 开发者，那么我相信本书会带你快速进入 HTML5 的世界。

如何阅读本书

本书从认识 HTML5 和 JavaScript 的面向对象开始展开，如果你从来没有接触过 HTML5 开发，那么请从第 1 章准备工作开始学习；如果你对 HTML5 有一定的了解，但不够全面，那么请从第 2 章 Canvas 基本功能开始系统学习；如果你已经全面掌握了 Canvas 的 API，那么可以从第 4 章 lufylegend 开源库件开始阅读，了解了 lufylegend 开源库件的运行原理之后再正式开始游戏的实战篇学习。

全书共包括四大部分，一共 11 章。

第一部分（第 1 章）是准备工作篇，介绍一些 HTML5 的发展历史和开发环境的搭建。

第二部分（第 2 ~ 4 章）是基础知识篇，详细讲解 HTML5 Canvas 的基础知识以及开源库件 lufylegend 的使用方法。

第三部分（第 5 ~ 10 章）是开发实战篇，通过实例为大家讲解如何使用 lufylegend 开源库件来进行 HTML5 的游戏开发，游戏实例涉及各种常见的游戏类型，包括休闲、射击、Box2d 及网游等多个领域。

第四部分（第 11 章）是技能提高篇，介绍如何运用数据来分析提高程序的效率。

PDG

勘误和支持

为了避免错误的发生，我已经将书中的源码进行了多次测试，但是由于本人的知识水平有限，加之编写的时间也很仓促，所以无法保证本书是100%正确的，书中也许会出现一些错误或者不准确的地方，恳请读者批评指正，以便再版时消除这些错误。如果您在阅读本书的过程中发现了不妥之处，欢迎您将错误信息发送到我的Web网站上，网址为http://lufylegend.com/book/view/1。本书源码下载地址为http://www.hzbook.com。我会及时阅读您提供的勘误信息，并将修改后的内容发布到网站上。

当然，也欢迎您发送邮件至我的邮箱lufy.legend@gmail.com，期待着您宝贵的意见和真挚反馈。

致谢

感谢我的父母，是你们给了我生命，给了我一颗不断学习的心。

感谢无所不知的网络，让我可以随时查阅资料。感谢无私地将自己的心得分享到网络上的coder们，是你们解决了我很多技术上的疑问，使我不断成长。

感谢机械工业出版社华章公司的杨绣国编辑，感谢你在百忙之中如此细心地审阅此书，是你的耐心和帮助引导我顺利完成了全部书稿。

最后我还要特别感谢软件工程师孙颖，是你跟我一起探讨了书中的内容，你的智慧与创意性的思维给了我无限的灵感。

谨以此书，献给众多热爱HTML5的朋友们。

张路斌（lufy）

目　　录

前　　言

第一部分　准备工作篇

第 1 章　准备工作 / 2

1.1　HTML5 介绍 / 2
　　1.1.1　什么是 HTML5 / 2
　　1.1.2　HTML5 的新特性 / 2
1.2　Canvas 简介 / 5
　　1.2.1　Canvas 标签的历史 / 5
　　1.2.2　Canvas 的定义和用法 / 6
　　1.2.3　如何使用 Canvas 来绘图 / 6
　　1.2.4　Canvas 的限制 / 7
1.3　开发与运行环境的准备 / 7
　　1.3.1　浏览器的支持 / 7
　　1.3.2　准备一个本地的服务器 / 8
1.4　开发工具的选择 / 8
1.5　测试与上传代码 / 12
1.6　JavaScript 中的面向对象 / 13
　　1.6.1　类 / 13
　　1.6.2　静态类 / 16
　　1.6.3　继承 / 16
1.7　小结 / 17

第二部分　基础知识篇

第 2 章　Canvas 基本功能 / 20

2.1　绘制基本图形 / 20
2.1.1　画线 / 20
2.1.2　画矩形 / 22
2.1.3　画圆 / 24
2.1.4　画圆角矩形 / 26
2.1.5　擦除 Canvas 画板 / 27
2.2　绘制复杂图形 / 28
2.2.1　画曲线 / 28
2.2.2　利用 clip 在指定区域绘图 / 30
2.2.3　绘制自定义图形 / 31
2.3　绘制文本 / 32
2.3.1　绘制文字 / 32
2.3.2　文字设置 / 33
2.3.3　文字的对齐方式 / 38
2.4　图片操作 / 41
2.4.1　利用 drawImage 绘制图片 / 41
2.4.2　利用 getImageData 和 putImageData 绘制图片 / 45
2.4.3　利用 createImageData 新建像素 / 47
2.5　小结 / 49

第 3 章　Canvas 高级功能 / 50

3.1　变形 / 50
3.1.1　放大与缩小 / 50
3.1.2　平移 / 53
3.1.3　旋转 / 54
3.1.4　利用 transform 矩阵实现多样化的变形 / 56
3.2　图形的渲染 / 65
3.2.1　绘制颜色渐变效果的图形 / 65
3.2.2　颜色合成之 globalCompositeOperation 属性 / 67
3.2.3　颜色反转 / 69
3.2.4　灰度控制 / 70
3.2.5　阴影效果 / 71

3.3 自定义画板 / 72
3.3.1 画板的建立 / 72
3.3.2 Canvas 画布的导出功能 / 79
3.4 小结 / 81

第 4 章 lufylegend 开源库件 / 82

4.1 lufylegend 库件简介 / 82
4.1.1 工作原理 / 82
4.1.2 库件使用流程 / 83
4.2 图片的加载与显示 / 84
4.2.1 图片显示举例 / 84
4.2.2 LBitmapData 对象 / 86
4.2.3 LBitmap 对象 / 87
4.3 层的概念 / 88
4.4 使用 LGraphics 对象绘图 / 90
4.4.1 绘制矩形 / 90
4.4.2 绘制圆 / 91
4.4.3 绘制任意多边形 / 92
4.4.4 使用 Canvas 的原始绘图函数进行绘图 / 93
4.4.5 使用 LSprite 对象进行绘图 / 94
4.4.6 使用 LGraphics 对象绘制图片 / 95
4.5 文本 / 101
4.5.1 文本属性 / 101
4.5.2 输入框 / 102
4.6 事件 / 103
4.6.1 鼠标事件 / 103
4.6.2 循环事件 / 104
4.6.3 键盘事件 / 105
4.7 按钮 / 106
4.8 动画 / 108
4.9 小结 / 113

第三部分 开发实战篇

第 5 章 从简单做起——“石头剪子布”游戏 / 116

5.1 游戏分析 / 116

5.2 必要的 JavaScript 知识 / 117
5.2.1 随机数 / 117
5.2.2 条件分支 / 117
5.3 分层实现 / 117
5.4 各个层的基本功能 / 119
5.4.1 基本画面显示 / 119
5.4.2 结果层的显示 / 126
5.4.3 控制层的显示 / 127
5.5 出拳 / 129
5.6 结果判定 / 131
5.7 小结 / 137

第 6 章 开发“俄罗斯方块”游戏 / 138

6.1 游戏分析 / 138
6.2 必要的 JavaScript 知识 / 138
6.3 游戏标题画面显示 / 139
6.4 向游戏里添加方块 / 141
6.5 控制方块的移动 / 152
6.5.1 键盘事件 / 152
6.5.2 触屏事件 / 155
6.6 方块的消除和得分的显示 / 157
6.7 小结 / 160

第 7 章 开发“是男人就下一百层”游戏 / 161

7.1 游戏分析 / 161
7.2 游戏标题画面显示 / 161
7.3 读取图片与背景显示 / 162
7.4 添加一个静止的地板 / 167
7.5 添加游戏主角 / 170
7.5.1 让游戏主角出现在画面上 / 170
7.5.2 通过键盘事件来控制游戏主角的移动 / 177
7.5.3 通过触屏事件来控制游戏主角的移动 / 178
7.6 添加多种多样的地板 / 179
7.6.1 会消失的地板 / 179
7.6.2 带刺的地板 / 181

7.6.3 带有弹性的地板 / 182
7.6.4 向左和向右移动的地板 / 184
7.7 游戏数据的显示 / 187
7.8 游戏结束与重开 / 190
7.9 小结 / 192

第 8 章 开发射击类游戏 / 193

8.1 游戏分析 / 193
8.2 添加一架可控飞机 / 194
8.2.1 添加一个飞机类 / 194
8.2.2 可控飞机类 / 197
8.3 为飞机添加多样化的子弹 / 203
8.3.1 建立一个子弹类 / 203
8.3.2 单发子弹 / 205
8.3.3 多发子弹 / 207
8.3.4 环形子弹 / 208
8.3.5 反向子弹 / 209
8.4 添加敌机 / 209
8.4.1 建立一个敌机类 / 210
8.4.2 建立一个敌机 Boss 类 / 214
8.5 碰撞检测 / 217
8.5.1 飞机与子弹的碰撞 / 217
8.5.2 我机与敌机的碰撞 / 220
8.6 子弹的变更 / 221
8.6.1 建立一个弹药类 / 222
8.6.2 弹药与我机的碰撞 / 223
8.7 飞机生命值的显示 / 225
8.8 游戏胜利与失败判定 / 226
8.9 小结 / 228

第 9 章 开发物理游戏 / 229

9.1 Box2D 简介 / 229
9.2 Box2dWeb 在 lufylegend 库件中的使用 / 229
9.3 创建各种各样的物体 / 234

9.3.1 矩形物体 / 234
9.3.2 圆形物体 / 237
9.3.3 多边形物体 / 239
9.4 响应鼠标拖拽物体 / 242
9.5 关节（Joint）/ 243
9.5.1 距离关节（b2DistanceJointDef）/ 243
9.5.2 旋转关节（b2RevoluteJointDef）/ 245
9.5.3 滑轮关节（b2PulleyJointDef）/ 247
9.5.4 移动关节（b2PrismaticJoint）/ 248
9.5.5 齿轮关节（b2GearJoint）/ 250
9.5.6 悬挂关节（b2LineJoint）/ 252
9.5.7 焊接关节（b2WeldJoint）/ 253
9.5.8 鼠标关节（Mouse Joint）/ 254
9.6 力 / 254
9.7 碰撞检测 / 256
9.8 镜头移动 / 260
9.9 做一个简单的物理游戏 / 263
9.10 小结 / 267

第 10 章 开发网络游戏 / 268

10.1 HTTP 通信 / 268
10.1.1 如何实现 HTTP 通信 / 268
10.1.2 HTTP 通信的弊端 / 275
10.2 Socket 通信 / 275
10.2.1 区分 Socket 通信和 HTTP 通信 / 276
10.2.2 服务器端 / 276
10.2.3 客户端 / 281
10.3 利用 WebSocket 实现简单的聊天室 / 283
10.4 做一款多人在线的坦克大战 / 293
10.4.1 服务器 / 293
10.4.2 客户端 / 293
10.5 小结 / 307

第四部分　技能提高篇

第 11 章　提高效率的分析 / 310

11.1　绘图时使用小数的影响 / 310
11.2　drawImage 和 putImageData 的效率比较 / 311
11.3　区域更新和图片大小对绘图效率的影响 / 311
11.4　图片格式对绘图效率的影响 / 313
11.5　优化代码以提高整体效率 / 314
11.5.1　使用位运算 / 314
11.5.2　少用 Math 静态类 / 316
11.5.3　优化算法 / 319
11.6　小结 / 322

第一部分

准备工作篇

❑ 第 1 章　准备工作

第1章　准备工作

作为本书的第一章，我们先来学习一下什么是HTML5、什么是Canvas元素、HTML5的开发与运行环境，以及如何选择它的开发工具等基础知识。对于游戏开发来说，如果不以面向对象为基础，那么开发思路就会不够清晰，代码也难以做到工整，可读性差，会给后期维护带来很大困难。所以在进入游戏开发之前，本章还会对JavaScript的面向对象编程进行简要的讲解。

1.1　HTML5介绍

自HTML5问世以来，越来越多的人开始关注它，打开搜索网页输入关键字“HTML5”，会发现与之相关的信息铺天盖地。可是，也有不少人谈及的HTML5只是古老的DHTML或Ajax，并非真正的HTML5。那么，到底什么是HTML5呢？下面我们就来认识一下，了解它具有哪些新特性。

1.1.1　什么是HTML5

HTML是Hyper Text Markup Language的简称，它是一种用于描述网页文档的标记语言，而HTML5则是这种标记语言的新标准。我们生活在一个网络信息时代，如何改良作为网页标记语言的HTML，自然成为开发者关注的重点内容之一。

自1993年6月HTML的第一版草案发布，到1999年12月24日HTML 4.01的发布，HTML一直在不断更新。但是HTML4并没有给HTML带来太大的突破，随着网络的迅速发展，它渐渐满足不了网络应用的需求了。2000年1月26日，可扩展超文本置标语言（eXtensible Hyper Text Markup Language），即XHTML出现了。XHTML的表现方式与HTML类似，不过语法上更加严格。因为XHTML更加注重页面规范和可用性，所以W3C执意发展XHTML。但是因为种种原因XHTML的进展非常缓慢，最主要是因为XHTML2不兼容以往任何一个版本的HTML。在这种情况下，HTML5出现了。

HTML5草案的前身名为Web Applications 1.0，于2004年由WHATWG提出，2007年W3C接纳了这种标准，并成立了新的HTML工作团队。HTML5的第一份正式草案于2008年1月22日公布。HTML5是W3C与WHATWG合作的结果，它成为HTML、XHTML以及HTML DOM的新标准。

1.1.2　HTML5的新特性

HTML5有很多令人心动的特性和新功能，比如，强化了Web网页的表现性能，增加了

本地数据库等 Web 应用的功能，以及图像操作等。

HTML5 在图像上引入了 Canvas 标签，通过 Canvas，用户可以动态生成各种图形图像、图表以及动画，而不再依赖于 Flash、silverlight 等插件了。

另外，HTML5 在地理位置操作上引入了 Geolocation API，其特点在于：

- 本身不去获取用户的位置，而是通过第三方接口来获取，例如 IP、GPS、WIFI 等方式。
- 用户可以随时开启和关闭，在被程序调用时也会首先征得用户同意，保证了用户的隐私。

同时，HTML5 还在数据储存上增加了本地数据库，可以使用 WebSQL 来储存数据，并且引入了 web storage API 实现了离线缓存功能，以此替代了 cookies，使得数据保存空间更大、更安全。

下面我们简单地举几个例子，来说明一下 HTML5 的优越之处。

1. 使用 video 标签播放动画

代码清单 1-1 中的 HTML 代码实现了播放动画功能。

代码清单 1-1

```
<video width="640" height="360" preload="auto" poster="hoge.png" controls autoplay>
<!-- 针对播放 webm 格式动画的浏览器 -->
<source src="hoge.webm" type='video/webm; codecs="vp8, vorbis"'>
<!-- 针对播放 ogv 格式动画的浏览器 ->
<source src="hoge.ogv" type='video/ogg; codecs="theora, vorbis"'>
<!-- 针对播放 mp4 格式动画的浏览器 -->
<source src="hoge.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
<!-- 当浏览器无法使用 video 标签的时候 -->
<p> 无法播放动画。<a href="hoge.html"> 推荐环境请看这里。</a></p>
</video>
```

2. 使用 audio 标签播放音频

代码清单 1-2 中的 HTML 代码可以实现音频播放功能。

代码清单 1-2

```
<audio controls loop>
<!-- 针对播放 ogg 格式音频的浏览器 -->
<source src="hoge.ogg">
<!-- 针对播放 wav 格式音频的浏览器 -->
<source src="hoge.wav">
<!-- 针对播放 mp3 格式音频的浏览器 -->
<source src="hoge.mp3">
<!-- 当浏览器无法使用 audio 标签的时候 -->
<p> 无法播放音频。<a href="hoge.html"> 推荐环境请看这里。</a></p>
</audio>
```

3. 使用 Canvas 标签绘制图形

代码清单 1-3 中的 HTML 代码可以实现绘制一个矩形的操作。

代码清单 1-3

```
<canvas id="canvas" width="640" height="360"></canvas>
<script>
// 获取 context 对象
var canvas = document.getElementById('canvas');
if(canvas.getContext){
var context = canvas.getContext('2d');
  // 设置颜色
context.fillStyle = 'rgb(255,0,0)';
  // 从坐标 (20,30) 开始，画一个 64×36 大小的矩形
context.fillRect(20,30,64,36);
}
</script>
```

4. 轻松取得当前的位置

代码清单 1-4 中的 JavaScript 代码可以获取当前位置的纬度和经度。

代码清单 1-4

```
<script>
window.addEventListener('load'. function () {
// 判断可否使用 geolocation
if(navigator.geolocation){
// 定期获取所在地
navigator.geolocation.watchPosition(update);
}
}, false);

// 取得位置并表示
function update(position){
// 纬度
var lat = position.coords.latitude;
// 经度
var lng = position.coords.longitude;
// 把纬度和经度显示出来
document.write(' 纬度：'+lat+', 经度：'+lng);
}
</script>
```

5. 将大量的数据保存在客户端

代码清单 1-5 中的 JavaScript 代码使用 LocalStorage 来保存大量的数据。

代码清单 1-5

```
<script>
```

```
//用 localStorage 来保存数据
localStorage.key = '想要保存的值';
//将 localStorage 中的值取出来
var hoge = localStorage.key;
//“想要保存的值”在页面上显示
document.write(hoge);
</script>
```

6. form 的强化

代码清单 1-6 中是 form 的几个比较常用的功能。

代码清单 1-6

```
<!-- 验证用户输入格式是否正确，只需要改变 type 的类型即可 -->
<input name="email" type="email">
<!-- 对于必须输入的项目，只需给 input 标签加上 require 属性即可 -->
<input name="text" type="text" require>
<!-- 当失去焦点的时候给出相应的提示，只需给 input 标签加上 placeholder 属性即可 -->
<input name="text" type="text" placeholder="例：姓名">
```

7. 全新的标签属性

在 HTML5 中取消了一些过时的 HTML4 标记，其中包括纯粹显示效果的标记，如 <font> 和 <center>，它们已经被 CSS 取代了。HTML5 吸取了针对 XHTML2 的一些建议，加强了一些用来改善文档结构的功能，如引入新的 HTML 标签 header、footer、dialog、aside、figure 等，使开发者能够更加容易地创建文档，以前开发者在实现这些功能时一般都是使用 div。另外，它还取消了一部分旧标签，如字体设置 font、居中设置 center 等。一小部分标签的含义也有所改变，如粗体样式 b 和斜体样式 i 标签虽然仍然保留，但它们的意义已经和以前有所不同，现在这些标签的意义只是为了将一段文字标识出来。

以上就是 HTML5 的一些新特性。需要注意的是，虽然 HTML5 已被 W3C 接纳，但现在还只是草案，在正式版发布之前，它的样式仍可能会有所变更。

1.2 Canvas 简介

Canvas 元素是 HTML5 的新元素之一，用于在网页上绘制图形，相当于在 HTML 中嵌入了一张画布，这样就可以直接在 HTML 上进行图形操作了，所以它具有极大的应用价值。Canvas 元素本身是没有绘图能力的，它需要借助 JavaScript 来实现绘图功能。

1.2.1 Canvas 标签的历史

Canvas 这个 HTML 元素是为客户端矢量图形而设计的。它自己没有行为，只是把一个绘图 API 展现给了客户端 JavaScript，以使脚本能够把想绘制的东西都绘制到一块画布上。

<canvas> 标记由 Apple 在 Safari 1.3 Web 浏览器中引入。对 HTML 进行这一扩展的原因

在于，希望 HTML 在 Safari 中的绘图能力也能为 Mac OS X 桌面的 Dashboard 组件所使用，并且 Apple 希望有一种方式在 Dashboard 中支持脚本化的图形。

Firefox 1.5 和 Opera 9 都跟随了 Safari 的引领，这两个浏览器都支持 <canvas> 标记。

可以在 IE 中使用 <canvas> 标记，并在 IE 的 VML 支持基础上用开源的 JavaScript 代码（由 Google 发起）来构建具有兼容性的画布。

<canvas> 的标准化工作正在由一个 Web 浏览器厂商的非正式协会进行推动。目前 <canvas> 已经成为 HTML5 草案中一个正式的标签。

1.2.2 Canvas 的定义和用法

使用 Canvas 标签，只需要向 HTML5 里添加 Canvas 元素即可，代码如下：

```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

1.2.3 如何使用 Canvas 来绘图

前面已经提到，Canvas 元素本身是没有绘图能力的，所有的绘制工作必须在 JavaScript 内部完成。下面来看看如何使用 Canvas 来绘图。

代码清单 1-7 所示代码可以实现绘制一个矩形。

代码清单 1-7

```
onload = function() {
    draw();
};
function draw() {
    /* 使用 id 来寻找 Canvas 元素 */
    var canvas = document.getElementById('canvassample');
    /* 验证 Canvas 元素是否存在，以及浏览器是否支持 Canvas 元素 */
    if ( ! canvas || ! canvas.getContext ) return false;  /* 创建 context 对象 */
    var ctx = canvas.getContext('2d');
    /* 画一个红色矩形 */
    cxt.fillStyle="#FF0000";
    cxt.fillRect(0,0,150,75);
}
```

下面来分析一下以上代码。要使用 Canvas 元素，首先要根据 id 或 name，将 Canvas 对象取出来，上面的代码使用的是 getElementById 方法，如果给 Canvas 标签加入了 name 属性，那么也可以使用 getElementByTagName 来获取 Canvas 对象。

另外，要使用 Canvas 元素，必须先判断这个元素是否存在及用户所使用的浏览器是否支持此元素。如果无法使用 Canvas 元素，那么下面做的所有事情都是无意义的。

上面的代码在使用 getContext 方法时，传递了一个“2d”参数，这就可以得到二维的 context 对象以实现二维图形的描画。试想一下，如果将来 Canvas 可以描画三维图形，或许也可以使用“3d”参数。但是目前还只能使用“2d”作为参数。

在上面的例子中，采用 fillStyle 方法将画笔颜色设置为红色。另外，用 fillRect 方法规定了图形的形状、位置和尺寸。更多 Canvas 的使用方法，将会在第 2 章之后进行详细介绍。

1.2.4 Canvas 的限制

使用 Canvas 可拥有多种绘制路径、矩形、圆形、字符及添加图像的方法。但是绘制的图形是静止的，如果要让所画的图形动起来，则需要画出每一帧的图形，然后再连接起来。这些都会在后面的章节做详细的介绍。

虽然 Canvas 可以对图形进行一系列操作，但是效率和普及性都存在一定的问题，对于一些相对复杂的图形和动画等，目前来看，还是 Flash 更胜一筹。

另外，Canvas 是 HTML5 的新属性，在使用时需要考虑用户的浏览器和使用环境。

Canvas 目前只是一张二维画布，如果要想实现三维效果，需要借助第三方类库，如 three.js 或者 Papervision3D 等。

1.3 开发与运行环境的准备

HTML5 有很多新的特性，各个浏览器对这些特性的支持度也都不一样。因为本书介绍的是基于 Canvas 的开发，所以在这里只介绍对 Canvas 的支持情况。

1.3.1 浏览器的支持

各种主流浏览器对 Canvas 的支持情况如表 1-1 所示。

表 1-1 浏览器对 Canvas 的支持度一览表

浏览器（版本）	是否支持 Canvas 元素
IE8	×
IE9	○
Firefox 3.6	○
Chrome 10.0	○
Safari 5.0	○
Opera 11.0	○

可以看到，支持 Canvas 的浏览器还是比较多的。但是在 IE 浏览器中，目前只有 IE9 以及更高版本才可以使用 HTML5 的 Canvas 标签。如果使用 IE8 或更低版本的 IE 浏览器，需要引入 Google 发布的开源类库 ExplorerCanvas 才可以，代码如下所示：

```
<! -- [if IE]><script src="excanvas.js"></script><![endif]-->
```

ExplorerCanvas 的下载地址为：http://code.google.com/p/explorercanvas/downloads/list。另外需要说明的是，低版本 IE 浏览器虽然引入开源类库 ExplorerCanvas 可以使用 Canvas，但是在功能上会有很多限制，如无法使用 fillText 方法等。

1.3.2 准备一个本地的服务器

虽然 HTML5 与 JavaScript 的开发都不需要用到专门的服务器，但是由于开发习惯的问题，笔者建议还是使用一个本地的服务器。

对于 Mac 来说，苹果系统本身自带本地服务器；对于 Windows 来说，则推荐使用 XAMPP（Apache+MySQL+PHP+PERL），它是一个功能强大的服务器系统开发套件，可以在 Windows、Linux、Solaris 等操作系统下安装使用，支持多语言（如英文、简体中文、繁体中文、韩文、俄文、日文等）。

XAMPP 的官方网址为：http://www.apachefriends.org/。

下载安装后，打开 xampp 文件夹并启动 xampp-control.exe 文件，然后单击 Apache 右侧的 Start 按钮启动 Apache，Apache 启动后状态变为“Running”，如图 1-1 所示。

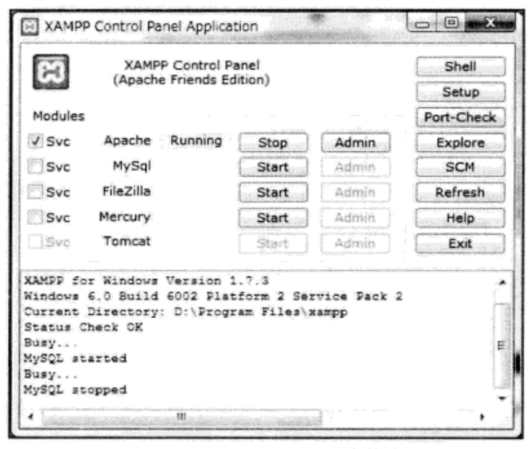


图 1-1 XAMPP 启动后状态

然后，将用户开发的文件或工程放到 XAMPP 安装文件夹下的 htdocs 文件夹下，这样就可以通过 http://localhost/ 文件名或 http://127.0.0.1/ 文件名来访问它了。

1.4 开发工具的选择

对于 HTML5 的开发来说，在开发工具方面没有特殊的要求，甚至可以用文本文档来编写代码。但是好的开发工具可以减少开发 Bug，提高开发效率。对于网页设计者来说，Dreamweaver 应该是最佳的选择，但由于 Dreamweaver 没有调试功能，而且 JavaScript 的自动提示功能又不太好用，作为以代码编写为主的开发者来说，这是难以接受的。

所以在这里推荐另一款开发工具，那就是 Eclipse，它是完全免费的，而且也带有比较好的 JavaScript 提示功能。

在下载 Eclipse 之前，需要先下载 JDK，来支持 Eclipse 的运行。JDK 的下载地址为：http://www.oracle.com/technetwork/java/javase/downloads/index.html

下载时，进入页面后，选择需要的 JDK，如图 1-2 所示。

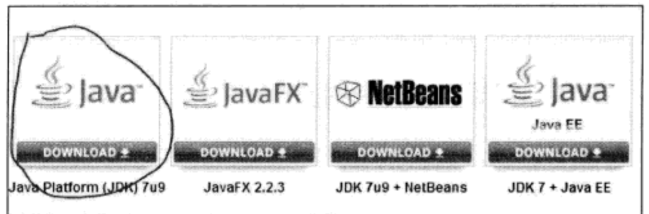


图 1-2　JDK 下载界面

单击 JDK 图标后，进入下载页面，SUN 公司为不同的系统提供了不同版本的 JDK，如图 1-3 所示。单击需要的版本进行下载，并安装。

Linux x86	120.63 MB	jdk-7u9-linux-i586.rpm
Linux x86	92.85 MB	jdk-7u9-linux-i586.tar.gz
Linux x64	118.82 MB	jdk-7u9-linux-x64.rpm
Linux x64	91.59 MB	jdk-7u9-linux-x64.tar.gz
Mac OS X	143.47 MB	jdk-7u9-macosx-x64.dmg
Solaris x86	135.14 MB	jdk-7u9-solaris-i586.tar.Z
Solaris x86	91.51 MB	jdk-7u9-solaris-i586.tar.gz
Solaris SPARC	135.7 MB	jdk-7u9-solaris-sparc.tar.Z
Solaris SPARC	95.15 MB	jdk-7u9-solaris-sparc.tar.gz
Solaris SPARC 64-bit	22.8 MB	jdk-7u9-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.51 MB	jdk-7u9-solaris-sparcv9.tar.gz
Solaris x64	22.48 MB	jdk-7u9-solaris-x64.tar.Z
Solaris x64	14.94 MB	jdk-7u9-solaris-x64.tar.gz
Windows x86	88.35 MB	jdk-7u9-windows-i586.exe
Windows x64	90.03 MB	jdk-7u9-windows-x64.exe

图 1-3　不同操作系统下的 JDK 下载选项

安装了 JDK 之后，就可以开始下载 Eclipse 了。Eclipse 的官方下载地址为：
http://www.eclipse.org/downloads/
下载界面如图 1-4 所示。

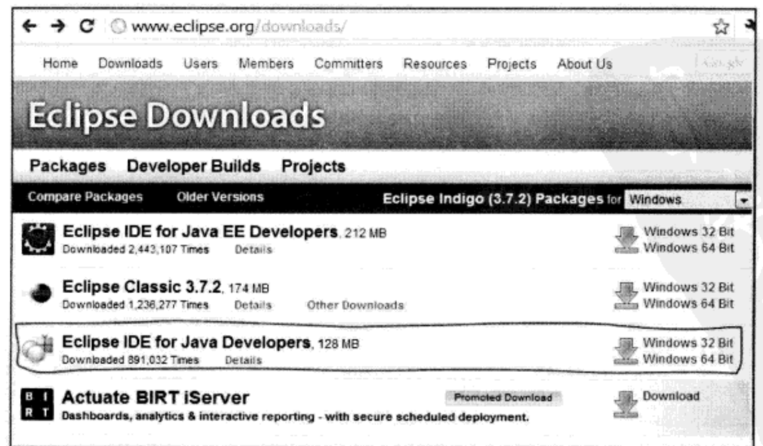


图 1-4　Eclipse 下载界面

下载后直接解压，运行 eclipse.exe 即可。

接下来开始安装 HTML5 环境。打开 Help 菜单中的 Install New Software 选项，如图 1-5 所示。

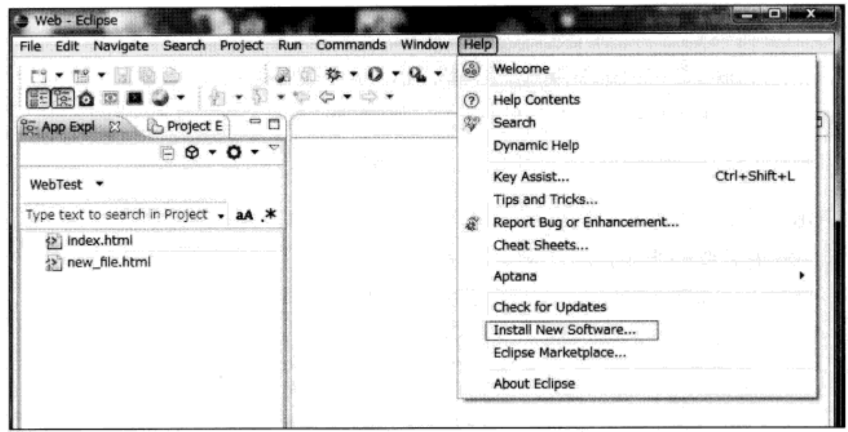


图 1-5 安装新组件操作

在弹出的窗口中，在 Work with 文本框中输入 http://download.aptana.com/studio3/plugin/install，如图 1-6 所示。

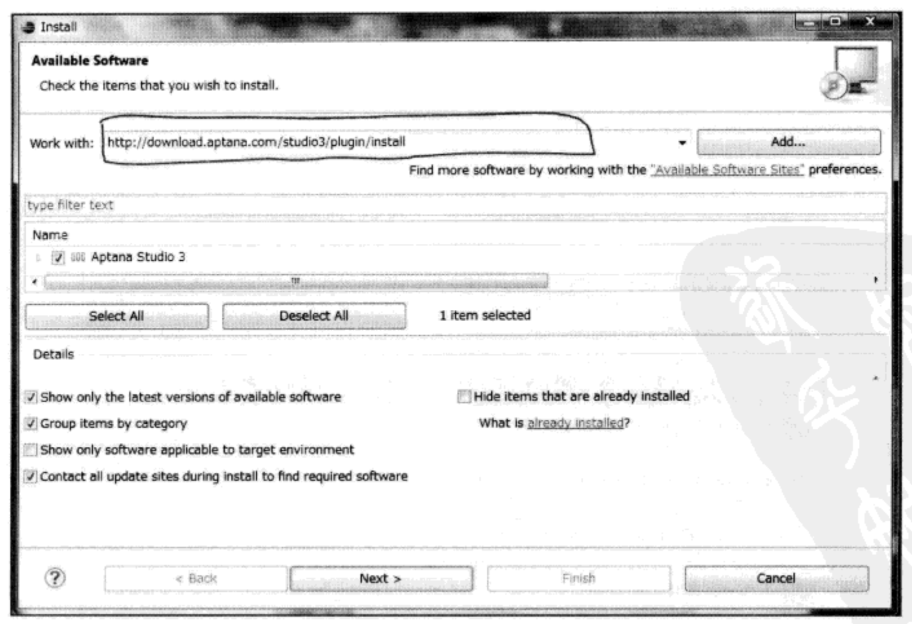


图 1-6 在文本框中输入 Aptana 下载地址

单击 Add 按钮，出现安装 Aptana Studio 3 界面，安装后重新启动 Eclipse。

在重新启动了 Eclipse 后，单击 Eclipse 左上角的 File 菜单，依次单击 New → Web Project，新建一个 Web Project，如图 1-7 所示。

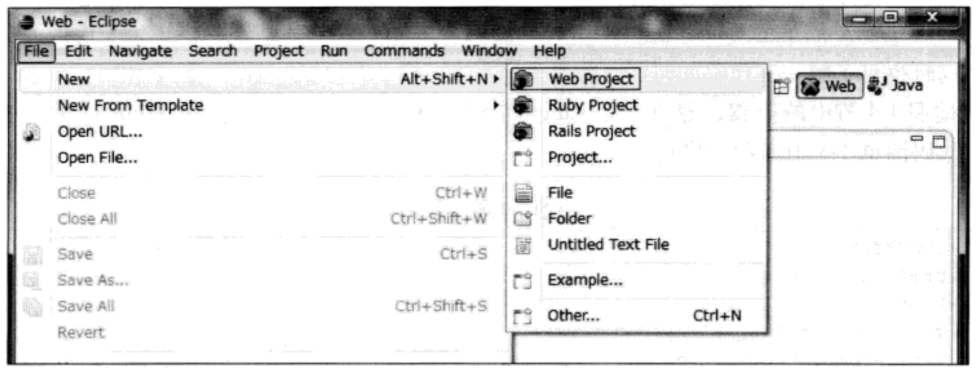


图 1-7　新建 Web Project

然后在 New From Template 菜单中，依次单击 HTML → HTML5 Template，新建一个 HTML 文件，如图 1-8 所示。

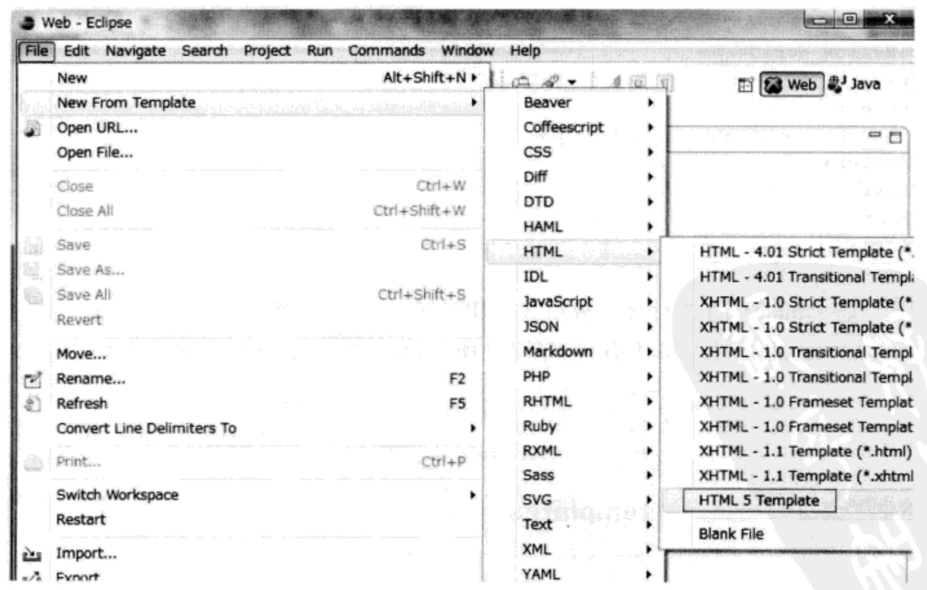


图 1-8　新建 HTML 文件

至此，整个准备工作就完成了。

1.5 测试与上传代码

在上传工程代码的时候，需要将工程里包含的 HTML、图片及 JavaScript 等所有文件上传至服务器，并保证文件结构不变。

现在来写个简单的页面，测试一下我们的服务器和开发工具是否已经正确无误地安装完毕，同时也可了解一下代码如何在服务器上运行。

按照 1.4 节中的步骤，建立一个 sample1.5.1 工程，在工程中新建一个 index.html 文件，输入代码清单 1-8 中所示的代码。

代码清单 1-8

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>templates</title>
</head>
<body>
<div>
<header>
<h1>templates</h1>
</header>
<nav>
<p>Hellow World</p>
</nav>
<footer>
<p>&copy; Copyright  by lufy</p>
</footer>
</div>
</body>
</html>
```

然后，将 sample1.5.1 文件夹存放到 XAMPP 的 htdocs 文件夹里。打开浏览器，输入网址：http://localhost/sample1.5.1/。如果看到如图 1-9 所示的测试结果，就说明开发环境安装成功了。

图 1-9 代码清单 1-8 的测试结果

1.6 JavaScript 中的面向对象

在本章开始的时候就已经说了，对于游戏开发来说，使用面向对象的方法进行编程是很有必要的。所以在介绍游戏开发之前，先来了解一下如何使用 JavaScript 进行面向对象的编程。可以说，JavaScript 是一种基于对象的语言，但是，它又不是一种真正的面向对象的编程语言，因为它的语法中不存在 class（类）。本节将分析和解决如何在 JavaScript 中实现封装和继承等面向对象的问题。

1.6.1 类

JavaScript 对象很抽象，所以下面将以实际的例子来解释如何定义一个简单的类。以下是一个没有任何属性和方法的类的定义：

```
function MyClass(){};
```

你可能会想，这不就是个简单函数的声明吗？没错，这个函数就是一个类的定义的实现。如何使用这个类呢？看下面的代码：

```
var cls1 = new MyClass();
```

这样，利用 new 就可以生成 MyClass 的一个实例了。所以，在 JavaScript 中，可以说函数就是类，类就是函数。

我们知道，一个实例的封装包含属性和方法的封装。那么如何实现呢？接着看下面的代码：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
};
var cls1 = new MyClass("lufy",10);
alert(cls1.name + ":" + cls1.age);//[lufy:10]
```

从上面的代码可以看出，在函数内使用 this 就能给函数本身增加属性值。而在上面的代码中就给 MyClass 函数增加了 name 和 age 两个属性。

同样，还可以利用 this 给这个类增加一个 toString 方法，代码如下：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
    this.toString() = function(){
            alert(this.name + ":" + this.age);
    };
};
var cls1 = new MyClass("lufy",10);
cls1.toString();//[lufy:10]
```

经过测试可以发现，我们已经成功地给 MyClass 增加了 toString 方法。另外，也可以用以下代码来添加方法：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
};
var cls1 = new MyClass("lufy",10);
cls1.toString() = function(){
    alert(this.name + ":" + this.age);
};
cls1.toString();//[lufy:10]
```

虽然这样也能给这个类添加一个方法，但需要注意的是，这种方式只是给 cls1 这个实例增加了方法，并未给 MyClass 本身增加方法。比如，下面的代码会出错：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
};
var cls1 = new MyClass("lufy",10);
cls1.toString() = function(){
    alert(this.name + ":" + this.age);
};
cls1.toString();//[lufy:10]
var cls2 = new MyClass("legend",12);
cls2.toString();// 报错
```

出错的原因是 cls2 内并不存在 toString 方法。可见想要通过这种方式来给一个类的本身增加方法是行不通的。要想给 MyClass 类的本身增加方法，需要将方法定义在 MyClass 这个函数的内部，这样的话，每声明一个新的实例，就会将 MyClass 本身复制一遍。但是，如果 MyClass 类里包含十几个或几十个方法，那么每次都得把这些方法复制一遍，这显然不是最优的做法。

既然不能将一个类（函数）所包含的方法都定义在函数的内部，那么，如何来给一个类添加方法呢？这就需要用到函数的 prototype 属性了。每一个函数都会包含一个 prototype 属性，这个属性指向了一个 prototype 对象，我们可以指定函数对应的 prototype 对象。如果不指定，则函数的 prototype 属性将指向一个默认的 prototype 对象，并且此默认 prototype 对象的 constructor 属性又会指向该函数。

当用构造函数创建一个新的对象时，新的对象会获取构造函数的 prototype 属性所指向的 prototype 对象的所有属性和方法，这样一来，构造函数对应的 prototype 对象所做的任何操作都会反映到它所生成的对象上，所有的这些对象将共享与构造函数对应的 prototype 对象的属性和方法。

虽然新创建的对象可以使用它的构造函数所指向的 prototype 对象的属性和方法，但不能像构造函数那样直接调用 prototype 对象（对象没有 prototype 属性）。

简而言之，就是如果我们使用函数的 prototype 对象来给函数添加方法，那么在创建一个新的对象的时候，并不会复制这个函数的所有方法，而是指向了这个函数的所有方法。

具体的实现方法参看下面的代码：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
};
MyClass.prototype.toString = function(){
    alert(this.name + ":" + this.age);
}
var cls1 = new MyClass("lufy",10);
cls1.toString();//[lufy:10]
var cls2 = new MyClass("legend",12);
cls2.toString();//[legend:12]
```

对于 prototype 对象来说，由于存在的是指向的关系，所以避免了不必要的浪费，如图 1-10 所示。

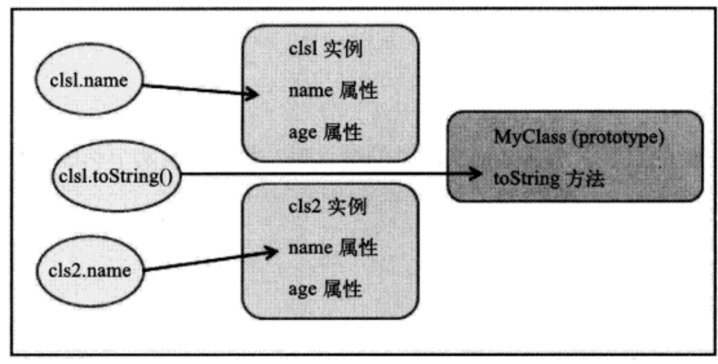


图 1-10　prototype 属性说明图

如果要加入多个方法，也可以使用下面的方式：

```
function MyClass(name,age){
    this.name = name;
    this.age = age;
};
MyClass.prototype = {
    toString:function(){
            alert(this.name + ":" + this.age);
    },
    sayHellow:function(){
            alert(this.name + ", 你好 !");
    }
};
```

```
var cls1 = new MyClass("lufy",10);
cls1.toString();//[lufy:10]
cls1.sayHellow ();//[lufy,你好 !]
```

这就是 JavaScript 中给类添加方法的实现，它是利用 prototype 来实现封装的。

1.6.2 静态类

那么，JavaScript 中的静态类又是如何实现的呢？其实下面的函数本身就可以当作静态类来用：

```
var StaticClass = function(){};
StaticClass.name = "StaticName";
StaticClass.Sum = function(value1,value2){
return value1 + value2;
};
alert(StaticClass.name);//[StaticName]
alert(StaticClass.Sum(1,3));//[4]
```

这样，在使用静态类的时候，就无须创建新的实例了，可直接用“类名 + 点 + 属性或方法”的方式。

1.6.3 继承

上面只讲了类的封装，那么如何实现类的继承呢？如有如下两个构造函数：

```
function PeopleClass(){
    this.type = "人";
};
PeopleClass.prototype = {
    getType:function(){
           alert("这是一个人");
    }
};
function StudentClass(name,sex){
    this.name = name;
    this.sex = sex;
};
```

如何让“学生”对象来继承“人”对象呢？可使用 apply 方法将父对象的构造函数绑定在子对象上，代码如下：

```
function PeopleClass(){
    this.type = "人";
};
PeopleClass.prototype = {
    getType:function(){
           alert("这是一个人");
    }
};
```

```
function StudentClass(name,sex){
    PeopleClass.apply(this, arguments);
    this.name = name;
    this.sex = sex;
};
var stu = new StudentClass("lufy","男");
alert(stu.type);//[人]
```

从运行结果来看，StudentClass 继承了 PeopleClass 的属性“人”。

而方法的继承，只要循环使用父对象的 prototype 进行复制，即可达到继承的目的。具体方法如下：

```
function PeopleClass(){
    this.type = "人";
};
PeopleClass.prototype = {
    getType:function(){
            alert("这是一个人");
    }
};
function StudentClass(name,sex){
    PeopleClass.apply(this, arguments);
    var prop;
    for(prop in PeopleClass.prototype){
            var proto = this.constructor.prototype;
            if(!proto[prop]){
                    proto[prop] = PeopleClass.prototype[prop];
            }
            proto[prop]["super"] = PeopleClass.prototype;
    }
    this.name = name;
    this.sex = sex;
};
var stu = new StudentClass("lufy","男");
alert(stu.type);//[人]
stu.getType();//[这是一个人]
```

以上，就是 JavaScript 中继承的实现。了解了这些知识后，就可以在 JavaScript 中使用面向对象的方法了，这对于提高开发效率是很有帮助的。

1.7 小结

本章主要带大家认识一下 HTML5，了解一下 HTML5 的背景，并且做了一些准备工作，包括开发环境的搭建和 JavaScript 面向对象知识的介绍。下一章我们将进入正题，从 HTML5 的 Canvas 标签的基础属性开始，一步步地学习如何开发各种类型的游戏。

第二部分

基础知识篇

- ❑ 第 2 章　Canvas 基本功能
- ❑ 第 3 章　Canvas 高级功能
- ❑ 第 4 章　lufylegend 开源库件

第 2 章　Canvas 基本功能

本章主要讲解 HTML5 Canvas 的基本功能，利用 Canvas 的 API，展示一些基本图形的绘制及操作方法，包括画线、画图、文字操作及图片操作等。

2.1　绘制基本图形

所谓基本图形，就是指线、矩形、圆等最简单的图形，任何复杂的图形都是由这些简单的图形组合而成的。我们首先来了解一下这些简单图形的绘制方法。

2.1.1　画线

你可能是第一次接触 Canvas 绘图。首先，我们通过绘制一个简单的直线来学习 Canvas 的功能。其代码如代码清单 2-1 所示。

代码清单　2-1

```
<!DOCTYPE HTML>
<html>
<body>
<canvas id="myCanvas" width="200" height="100">
你的浏览器不支持 HTML5
</canvas>
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 10;
ctx.strokeStyle = "red";
ctx.beginPath();
ctx.moveTo(10,10);
ctx.lineTo(150,50);
ctx.stroke();
</script>
</body>
</html>
```

运行后的效果图如图 2-1 所示。

下面来解释一下代码清单 2-1 中的代码。

```
<canvas id="myCanvas" width="200" height="100">
你的浏览器不支持 HTML5
</canvas>
```

PDG

这是在 HTML 中嵌入 Canvas 标签。Canvas 标签内部可以添加文字或 HTML 代码，如果浏览不支持 Canvas 标签，那么浏览器会自动跳过 Canvas 标签而运行 Canvas 内部的 HTML 代码。

```
var c=document.getElementById("myCanvas");
```

以上代码是获取 HTML 中的 Canvas 标签。

```
var ctx=c.getContext("2d");
```

这里返回一个用来绘制环境类型的环境。它返回的是一个 CanvasRenderingContext2D 对象，该对象实现了一个画布所使用的大多数方法。目前，Canvas 只支持二维环境，所以参数只能是“2d”。当然，将来也可能会支持三维。

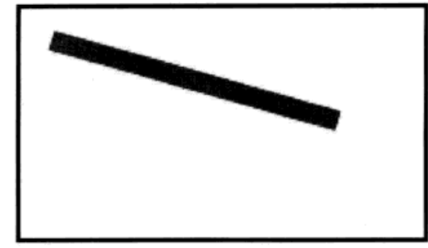


图 2-1　代码清单 2-1 运行后的效果图

```
ctx.lineWidth = 10;
```

以上代码用来设置线条宽度。

```
ctx.strokeStyle = "red";
```

此处设置画笔颜色为红色，这里的颜色值可以是英文字母，也可以直接使用颜色的 RGB 值，如红色为“#ff0000”，黑色为 rgb(0,0,0) 等。

```
ctx.beginPath();
```

以上代码创建一个新的路径。

```
ctx.moveTo(10,10);
```

以上代码将画笔光标位置移动到坐标 (10,10) 处。

```
ctx.lineTo(150,50);
```

以上代码从当前坐标开始移动画笔到坐标 (150,50) 处，绘制一条直线。

```
ctx.stroke();
```

上面代码表示开始绘制定义好的路径。

以上过程其实和我们在纸上画一条线是同样的道理，首先我们要选择一种颜色及线条的粗细，然后用画笔从一个点开始画到另一个点，这样就可以画出一条线了。

在画线的时候，也可以使用 lineCap 来定义线帽的样式，如在代码清单 2-2 中，分别使用了 lineCap 的 3 种样式。

代码清单　2-2

```
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 10;
```

```
ctx.strokeStyle = "red";

ctx.lineCap="butt";
ctx.beginPath();
ctx.moveTo(10,10);
ctx.lineTo(150,10);
ctx.stroke();

ctx.lineCap="round";
ctx.beginPath();
ctx.moveTo(10,40);
ctx.lineTo(150,40);
ctx.stroke();

ctx.lineCap="square";
ctx.beginPath();
ctx.moveTo(10,70);
ctx.lineTo(150,70);
ctx.stroke();
```

运行代码，可以看到 3 种不同的线帽，如图 2-2 所示。

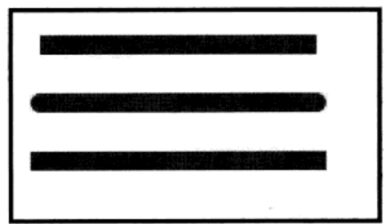


图 2-2 代码清单 2-2 运行后的效果图

2.1.2 画矩形

下面来看看如何画一个矩形，其代码如代码清单 2-3 所示。

代码清单 2-3

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 5;
ctx.strokeStyle = "red";
ctx.beginPath();
ctx.strokeRect(10,10,70,40);
</script>
```

运行效果如图 2-3 所示。

PDG

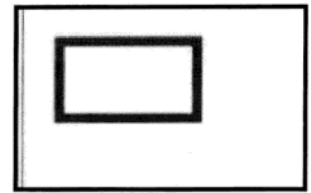


图 2-3　代码清单 2-3 运行后的效果图

在 Canvas 里，用 strokeRect 函数来绘制一个矩形，它需要 4 个参数，分别是：起点坐标 x 和坐标 y、矩形长、矩形宽。

也可以用下面代码来替换 strokeRect 函数，它可以实现同样的功能。

```
ctx.rect(10,10,70,40);
ctx.stroke();
```

如果要绘制一个实心的矩形，可以用 fillRect 函数，如代码清单 2-4 所示。

代码清单　2-4

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle = "red";
ctx.beginPath();
ctx.fillRect(10,10,70,40);
</script>
```

运行效果如图 2-4 所示。

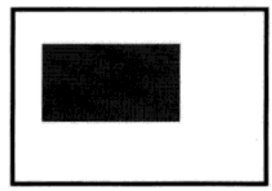


图 2-4　代码清单 2-4 运行后的效果图

fillRect 函数同样需要 4 个参数，分别是：起点的坐标 x 和坐标 y、矩形长、矩形宽。与 strokeRect 函数不同的是，画实心图形的时候，用 fillStyle 来定义图形的颜色。

当然，这里也可用另一种实现方法绘制矩形，代码如下：

```
ctx.rect(10,10,70,40);
ctx.fill();
```

PDG

2.1.3 画圆

圆其实就是一个 360 度的圆弧。在 Canvas 中，可使用 arc 函数来画一个圆弧。先看代码清单 2-5 所示代码。

代码清单 2-5

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 5;
ctx.strokeStyle = "red";
ctx.beginPath();
ctx.arc(100,100,70,0,130*Math.PI/180,true);
ctx.stroke();
</script>
```

运行效果如图 2-5 所示。

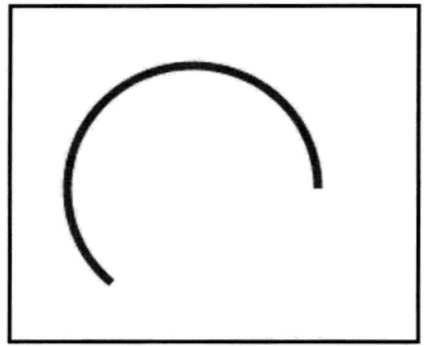


图 2-5 代码清单 2-5 运行后的效果图

arc 函数的 6 个参数分别是：圆弧中心的坐标 x 和坐标 y、圆弧半径、起始角度、终止角度、是否逆时针。需要解释的是，第 4 个和第 5 个参数需要传入的是圆弧的弧度，如要画 30 度的角，需要将其转化成弧度 30*Math.PI/180；第 6 个参数用来控制圆弧是顺时针旋转还是逆时针旋转。

和画矩形一样，同样可以用 fill 函数来画一个实心的圆弧，如代码清单 2-6 所示。

代码清单 2-6

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 5;
ctx.fillStyle = "red";
ctx.beginPath();
ctx.arc(100,100,70,0,130*Math.PI/180,true);
```

```
ctx.fill();
</script>
```

运行效果如图 2-6 所示。

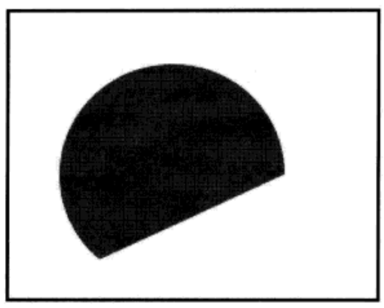


图 2-6　代码清单 2-6 运行后的效果图

画圆时，只需要让起始角度和终止角度之差为 360 度即可，具体代码如代码清单 2-7 所示。

代码清单　2-7

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.lineWidth = 5;
ctx.fillStyle = "red";
ctx.beginPath();
ctx.arc(100,100,70,0,360*Math.PI/180,true);
ctx.fill();
</script>
```

运行效果如图 2-7 所示。

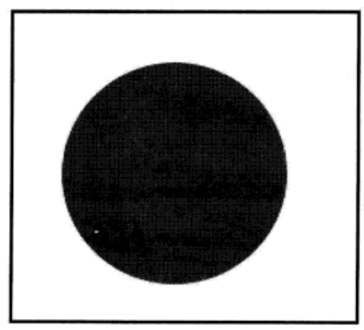


图 2-7　代码清单 2-7 运行后的效果图

2.1.4 画圆角矩形

Canvas 中没有直接画圆角矩形的 API，但是我们可以用 arcTo 函数来完成圆角的绘制，然后结合直线绘制，就可以完成圆角矩形的绘制了。在绘制圆角矩形之前，我们先来绘制一个圆角，如代码清单 2-8 所示。

代码清单 2-8

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.moveTo(20,20);
ctx.lineTo(70,20);
ctx.arcTo(120,30,120,70,50);
ctx.lineTo(120,120);
ctx.stroke(); </script>
```

运行效果如图 2-8 所示。

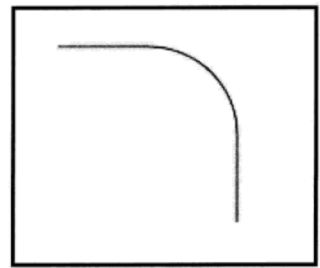


图 2-8 代码清单 2-8 运行后的效果图

在代码清单 2-8 中，arcTo 函数是用来为当前的子路径添加一条圆弧的，它需要 5 个参数，分别是：点 P1 的坐标 x 和坐标 y、点 P2 的坐标 x 和坐标 y、圆弧的半径 radius。该圆弧有一个点与当前位置到 P1 的线段相切，还有一个点和从 P1 到 P2 的线段相切。这两个切点就是圆弧的起点和终点，圆弧绘制的方向就是连接这两个点的最短圆弧的方向。

在很多常见的应用中，圆弧开始于当前位置而结束于 P2，但情况并不总是这样。如果当前的位置和圆弧的起点不同，这个方法将会添加一条从当前位置到圆弧起点的直线，而且总是将当前位置设置为圆弧的终点。

有了对 arcTo 函数的了解，画圆角矩形就简单多了，如代码清单 2-9 所示。

代码清单 2-9

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
```

```
ctx.moveTo(40,20);
ctx.lineTo(100,20);
ctx.arcTo(120,20,120,40,20);
ctx.lineTo(120,70);
ctx.arcTo(120,90,100,90,20);
ctx.lineTo(40,90);
ctx.arcTo(20,90,20,70,20);
ctx.lineTo(20,40);
ctx.arcTo(20,20,40,20,20);
ctx.stroke();
</script>
```

运行效果如图 2-9 所示。

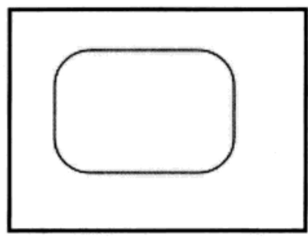


图 2-9　代码清单 2-9 运行后的效果图

2.1.5　擦除 Canvas 画板

擦除 Canvas 画板上的内容需要用到 clearRect 函数，此函数可以擦除一个矩形区域。它需要 4 个参数：起点的坐标 x 和坐标 y，擦除区域的长和宽。其用法如代码清单 2-10 所示。

代码清单　2-10

```
<canvas id="myCanvas" width="200" height="100">
你的浏览器不支持 HTML5
</canvas>
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle = "red";
ctx.beginPath();
ctx.fillRect(10,10,200,100);
ctx.clearRect(30,30,50,50);
</script>
```

上面的代码先绘制了一个红色的实心矩形，然后在红色矩形中间擦除了一个 50 × 50 的小矩形，效果如图 2-10 所示。

PDG

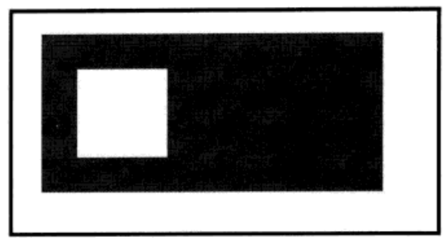


图 2-10 代码清单 2-10 运行后的效果图

2.2 绘制复杂图形

前面了解了基本图形的绘制方法，下面我们来看看复杂图形该如何绘制。

2.2.1 画曲线

贝塞尔曲线，又称贝兹曲线或贝济埃曲线，是应用于二维图形应用程序的数学曲线。在这里我们不研究贝塞尔曲线的原理，主要介绍在 Canvas 中如何绘制它。

1. 二次贝塞尔曲线

二次贝塞尔曲线有一个控制点。在 Canvas 中用 quadraticCurveTo(cpx,cpy,x,y) 函数来绘制二次贝塞尔曲线，cpx、cpy 表示控制点的坐标；x、y 表示终点坐标。

它的绘制如代码清单 2-11 所示。

代码清单 2-11

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.moveTo(100,100);
ctx.quadraticCurveTo(20,50,200,20);
ctx.stroke();
</script>
```

绘制效果如图 2-11 所示。

图 2-11 代码清单 2-11 运行后的效果图

图 2-12 所示是它的原理图。

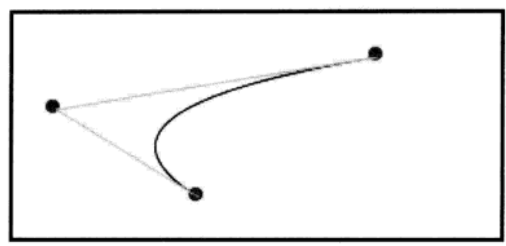


图 2-12　代码清单 2-11 运行的原理图

2. 三次贝塞尔曲线

三次贝塞尔曲线与二次贝塞尔曲线的区别在于，三次贝塞尔曲线有两个控制点，如代码清单 2-12 所示。

代码清单　2-12

```
<script type="text/javascript">
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.moveTo(68, 130);
var cX1 = 20;
var cY1 = 10;
var cX2 = 268;
var cY2 = 10;
var endX = 268;
var endY = 170;
ctx.bezierCurveTo(cX1, cY1, cX2,cY2, endX, endY);
ctx.stroke();
</script>
```

绘制效果如图 2-13 所示。

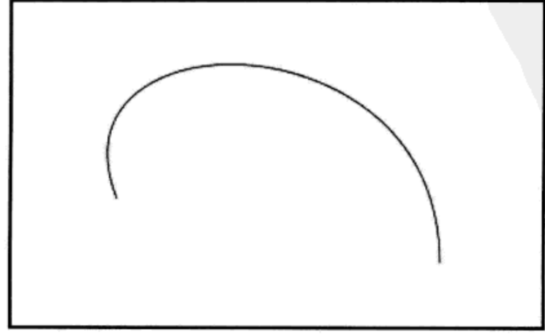


图 2-13　代码清单 2-12 运行后的效果图

PDG

图 2-14 所示是它的原理图。

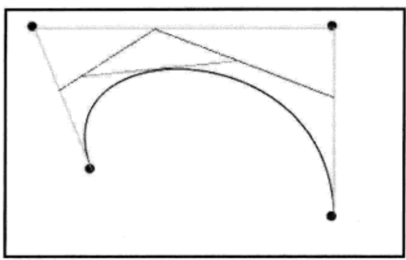


图 2-14 代码清单 2-12 运行的原理图

2.2.2 利用 clip 在指定区域绘图

clip 函数使用当前路径作为连续绘制操作的剪切区域。我们可以把它看作一扇窗户，无论在画板上绘制了多大的图形，最后看到的图形都只能由 clip 这扇窗户来决定。

为了更容易理解，我们看代码清单 2-13。

代码清单 2-13

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.arc(100,100,40,0,360*Math.PI/180,true);
ctx.clip();
ctx.beginPath();
//设定颜色
ctx.fillStyle="lightblue";
//绘制矩形
ctx.fillRect(0,0,300,150);
</script>
```

在这里，我们首先用 arc 画了一个圆，然后加入 clip 函数，最后又画了一个矩形，最后的效果如图 2-15 所示。

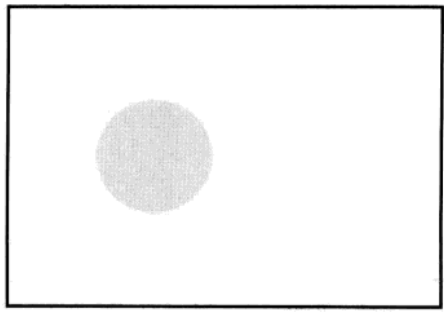


图 2-15 代码清单 2-13 运行后的效果图

可以看到，虽然我们画了一个矩形，但是最后出现的图形里并没有矩形。这是因为首先绘制的是一个圆，然后 clip 函数将当前的这个圆作为绘制操作的区域，所以之后画出的图形只能显示在这个区域内。也就是说，即使一间房子再大，可是它的窗户很小，最后我们透过这扇窗户能够看到的空间也就只有窗户这么大。所以代码清单中画出的大矩形只能显示出图 2-15 所示的效果了。

2.2.3　绘制自定义图形

所谓自定义图形，是指将上面讲到的各种方法结合起来，实现一些特殊的图形。比如代码清单 2-14。

代码清单　2-14

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.moveTo(100, 150);
ctx.bezierCurveTo(50, 100, 100, 0, 150, 50);
ctx.bezierCurveTo(200, 0, 250, 100, 200, 150);
ctx.bezierCurveTo(250, 200, 200, 300, 150, 250);
ctx.bezierCurveTo( 100, 300, 50, 200,100, 150);
ctx.closePath();
ctx.moveTo(100, 150);
ctx.lineTo(150, 50);
ctx.lineTo(200, 150);
ctx.lineTo(150, 250);
ctx.lineTo(100, 150);
ctx.lineWidth = 5;
ctx.strokeStyle = "#ff0000";
ctx.stroke();
</script>
```

上面代码绘制的自定义图形如图 2-16 所示。

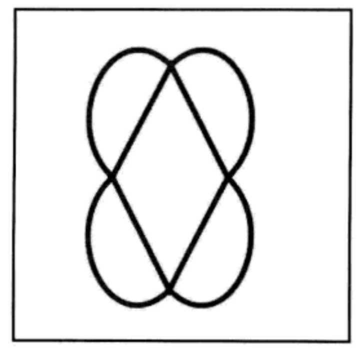


图 2-16　代码清单 2-14 运行后的效果图

2.3 绘制文本

文本在任何应用里都是必不可少的，可是，在 Canvas 的 API 中，只能显示文字，无法直接绘制一个输入框。所以，当需要显示输入框的时候需要使用 HTML 中的文本框来代替。下面来讲一下如何显示各种各样的文字。

2.3.1 绘制文字

绘制文字有 fillText 和 strokeText 两种方法，下面分别说明。

1. 使用 fillText 绘制文字

fillText(text,x,y,maxWidth) 函数很简单，它的 4 个参数分别是：文本字符串、坐标 x 和坐标 y、文本宽，其中第 4 个参数可以省去。当第 4 个参数省去的时候，文本宽度会自动设定为整个文本的宽度。

具体使用方法如代码清单 2-15 所示。

代码清单 2-15

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
//设定文字大小和字体
ctx.font="30px Arial";
//设定文字内容
ctx.fillText("Hello World",100,50);
</script>
```

运行效果如图 2-17 所示。

Hello World

图 2-17 代码清单 2-15 运行后的效果图

如果给 fillText 设定了第 4 个参数，那么整个文本的宽度就会发生变化，比如：

```
ctx.fillText("Hello World",100,50,50);
```

运行效果如图 2-18 所示。

Hello World

图 2-18 上述代码运行后的效果图

2. 使用 strokeText 绘制文字

使用 strokeText(text,x,y,maxWidth) 函数同样需要 4 个参数，它的用法与 fillText 函数完全相同，具体实现如代码清单 2-16 所示。

代码清单　2-16

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
// 设定文字大小和字体
ctx.font="30px Arial";
// 描画文字
ctx.strokeText ("Hello World",100,50);
</script>
```

运行效果如图 2-19 所示。

Hello World

图 2-19　代码清单 2-16 运行后的效果图

从效果图上我们不难看出 strokeText 与 fillText 的区别，strokeText 相当于线，而 fillText 相当于实心图形，这和之前画图形的绘制方法是相似的。

2.3.2　文字设置

上一节我们只是简单地讲解了如何显示一个文本，但是，一个文本可以有多种字体格式，各种文字也有大有小，有粗有细，本节会针对文字的设置做详细的说明。

1. 文字大小

可能大家已经发现，在上一节的各代码中都用到了 font 这个参数，如下面一行代码：

```
ctx.font="30px Arial";
```

它表示文字大小为 30，字体为 Arial。

下面我们来了解如何设定多种不同的文字大小，具体如代码清单 2-17 所示。

代码清单　2-17

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.beginPath();
```

```
// 设定文字大小为 30px
ctx.font="30px Arial";
ctx.fillText("Hello World",100,50);

ctx.beginPath();
// 设定文字大小为 50px
ctx.font="50px Arial";
ctx.fillText("Hello World",100,150);

ctx.beginPath();
// 设定文字大小为 100px
ctx.font="70px Arial";
ctx.fillText("Hello World",100,250);
</script>
```

运行效果如图 2-20 所示。

Hello World

Hello World

Hello World

图 2-20 代码清单 2-17 运行后的效果图

2. 文字字体

下面我们试着改变 font 中的字体，看看具体表现如何，如代码清单 2-18 所示。

代码清单 2-18

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.beginPath();
// 设定文字字体为 Arial
ctx.font="30px Arial";
ctx.fillText("Hello World (Arial)",50,50);
```

PDG

```
ctx.beginPath();
//设定文字字体为 Verdana
ctx.font="30px Verdana";
ctx.fillText("Hello World (Verdana)",50,100);

ctx.beginPath();
//设定文字字体为 Times New Roman
ctx.font="30px Times New Roman";
ctx.fillText("Hello World (Times New Roman)",50,150);

ctx.beginPath();
//设定文字字体为 Courier New
ctx.font="30px Courier New";
ctx.fillText("Hello World (Courier New)",50,200);

ctx.beginPath();
//设定文字字体为 serif
ctx.font="30px serif";
ctx.fillText("Hello World (serif)",50,250);

ctx.beginPath();
//设定文字字体为 sans-serif
ctx.font="30px sans-serif";
ctx.fillText("Hello World (sans-serif)",50,300);
</script>
```

运行效果如图 2-21 所示。

Hello World (Arial)

Hello World (Verdana)

Hello World (Times New Roman)

Hello World (Courier New)

Hello World (serif)

Hello World (sans-serif)

图 2-21 代码清单 2-18 运行后的效果图

可以看到，经过简单的设置就实现了不同的字体效果。

PDG

3. 文字粗体效果

和 CSS 一样，我们同样可以在 Canvas 中设置文字的 font-weight 属性，来给文字设置粗体效果。它同样可通过 font 来设置，使用方法如下：

```
ctx.font='normal 30px Arial';
```

font-weight 的值可以是 normal（正常）、bold（粗体）、bolder（更粗）、lighter（更细），还可以通过数字直接来设置，如下所示：

```
ctx.font='300 30px Arial';
```

下面设置不同的 font-weight，来对比一下效果，如代码清单 2-19 所示。

代码清单 2-19

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.beginPath();
//设定 font-weight 为 normal
ctx.font='normal 30px Arial';
ctx.fillText("Hello World (normal)",50,50);

ctx.beginPath();
//设定 font-weight 为 bold
ctx.font='bold 30px Arial';
ctx.fillText("Hello World (bold)",50,90);

ctx.beginPath();
//设定 font-weight 为 bolder
ctx.font='bolder 30px Arial';
ctx.fillText("Hello World (bolder)",50,130);

ctx.beginPath();
//设定 font-weight 为 lighter
ctx.font='lighter 30px Arial';
ctx.fillText("Hello World (lighter)",50,170);

ctx.beginPath();
//设定 font-weight 为 100
ctx.font='100 30px Arial';
ctx.fillText("Hello World (100)",50,210);

ctx.beginPath();
//设定 font-weight 为 600
ctx.font='600 30px Arial';
ctx.fillText("Hello World (600)",50,250);

ctx.beginPath();
```

```
// 设定 font-weight 为 900
ctx.font='900 30px Arial';
ctx.fillText("Hello World (900)",50,290);
</script>
```

运行效果如图 2-22 所示。

Hello World (normal)
Hello World (bold)
Hello World (bolder)
Hello World (lighter)
Hello World (100)
Hello World (600)
Hello World (900)

图 2-22 代码清单 2-19 运行后的效果图

可以看到，文字的粗细根据 font-weight 的值不同，效 果也都有所不同。

4. 文字斜体效果

有时候我们需要用斜体的方式来显示文字，Canvas 中的文字也有 font-style 属性，它也可通过设置 font 来处理，使用的方法如下 ：

```
ctx.font='italic 30px Arial';
```

font-style 的值可以设置为 italic，也可以设置为 oblique，它们都表示斜体，如代码清单 2-20 所示。

代码清单 2-20

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.beginPath();
// 设定 font-weight 为 normal
ctx.font='normal 30px Arial';
ctx.fillText("Hello World (normal)",50,50);

ctx.beginPath();
```

```
// 设定 font-style 为 italic
ctx.font='italic 30px Arial';
ctx.fillText("Hello World (italic)",50,90);

ctx.beginPath();
// 设定 font-style 为 oblique
ctx.font='oblique 30px Arial';
ctx.fillText("Hello World (oblique)",50,130);
</script>
```

运行效果如图 2-23 所示。

Hello World (normal)
Hello World (italic)
Hello World (oblique)

图 2-23 代码清单 2-20 运行后的效果图

2.3.3 文字的对齐方式

Canvas 中的文字通过 textAlign 和 textBaseline 来实现文字的对齐。textAlign 是水平方向的文字对齐，它的值包括 center、end、left、right、start。textBaseline 是竖直方向的文字对齐，它的值包括 alphabetic、bottom、hanging、ideographic、middle、top。

首先看水平方向的对齐。为了看出不同的对齐方式之间的区别，我们在文字坐标位置画一条竖线，如代码清单 2-21 所示。

代码清单 2-21

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.moveTo(160,0);
ctx.lineTo(160,300);
ctx.stroke();

ctx.beginPath();
ctx.textAlign='start';
ctx.font='30px Arial';
ctx.fillText("Hello World",160,50);

ctx.beginPath();
```

```
ctx.textAlign='end';
ctx.font='30px Arial';
ctx.fillText("Hello World",160,100);

ctx.beginPath();
ctx.textAlign='left';
ctx.font='30px Arial';
ctx.fillText("Hello World",160,150);

ctx.beginPath();
ctx.textAlign='center';
ctx.font='30px Arial';
ctx.fillText("Hello World",160,200);

ctx.beginPath();
ctx.textAlign='right';
ctx.font='30px Arial';
ctx.fillText("Hello World",160,250);
</script>
```

运行效果如图 2-24 所示。

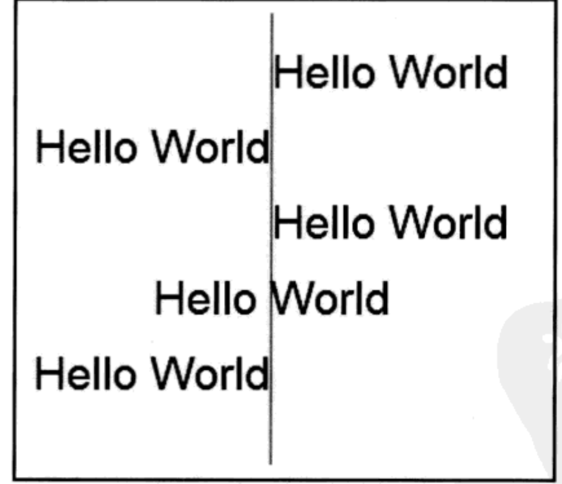


图 2-24　代码清单 2-21 运行后的效果图

可以看到，start 与 left 相同，表示文字从左侧开始对齐；end 与 right 相同，表示文字从右侧开始对齐，center 表示文字居中。

接下来看竖直方向的对齐。同样，为了看出不同的对齐方式之间的区别，我们在文字坐标位置画一条横线，如代码清单 2-22 所示。

代码清单 2-22

```
<script type="text/javascript">
var c=document.getElementById('myCanvas');
var ctx=c.getContext('2d');

ctx.textBaseline='alphabetic';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,50);
ctx.moveTo(0,50);
ctx.lineTo(250,50);
ctx.stroke();

ctx.textBaseline='bottom';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,100);
ctx.moveTo(0,100);
ctx.lineTo(250,100);
ctx.stroke();

ctx.textBaseline='hanging';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,150);
ctx.moveTo(0,150);
ctx.lineTo(250,150);
ctx.stroke();

ctx.textBaseline='ideographic';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,200);
ctx.moveTo(0,200);
ctx.lineTo(250,200);
ctx.stroke();

ctx.textBaseline='middle';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,250);
ctx.moveTo(0,250);
ctx.lineTo(250,250);
ctx.stroke();

ctx.textBaseline='top';
ctx.font='30px Arial';
ctx.fillText('Hello World',50,300);
ctx.moveTo(0,300);
ctx.lineTo(250,300);
ctx.stroke();
</script>
```

运行效果如图 2-25 所示。

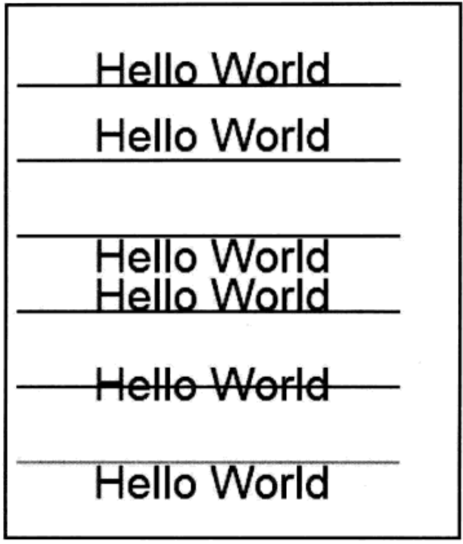


图 2-25　代码清单 2-22 运行后的效果图

图 2-25 所示很清晰地体现了竖直方向的各个对齐方式的区别。

2.4　图片操作

无论我们开发的是应用程序还是游戏软件，都是离不开图片，没有图片就无法让整个页面漂亮起来。开发游戏的时候，游戏中的地图、背景、人物、物品等都是由图片组成的，所以图片的显示和操作非常重要。Canvas 中提供了 drawImage 函数和 putImageData 函数来绘制图片，在本节中将一一讲解。

2.4.1　利用 drawImage 绘制图片

drawImage 函数有 3 种函数原型，其语法如下：

```
drawImage(image, dx, dy);
drawImage(image, dx, dy, dw, dh);
drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh);
```

第一个参数 image 是要绘制的对象，这个参数可以是 HTMLImageElement、HTMLCanvasElement 或者 HTMLVideoElement，dx、dy 是 image 在 Canvas 中定位的坐标值，dw、dh 表示 image 在 Canvas 中即将绘制区域（相对 dx 和 dy 坐标的偏移量）的宽度和高度值，sx、sy 是 image 所要绘制的起始位置，sw、sh 表示 image 所要绘制区域（相对 image 的 sx 和 sy 坐标的偏移量）的宽度和高度值。

关于第一个参数，我们先使用 HTML 的 <img> 标签来得到将要绘制的图片的数据。首

先准备一张图片 face.jpg，然后看代码清单 2-23。

代码清单 2-23

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>
img 标签 <br />
<img id="face" src="face.jpg" alt="The Face" width="240" height="240" /><br />
canvas 画板 <br />
<canvas id="myCanvas" width="500" height="350">
你的浏览器不支持 HTML5
</canvas>
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var img=document.getElementById("face");
ctx.drawImage(img,10,10);
</script>
</body>
</html>
```

代码解析

首先在 html 中加入 <img> 标签。

```
<img id="face" src="face.jpg" alt="The Face" width="240" height="240" /><br />
```

然后在 Canvas 中通过 <img> 标签的 id 取得图片数据。

```
var img=document.getElementById("face");
```

最后用 drawImage 函数将图片绘制到画板上。

```
ctx.drawImage(img,10,10);
```

运行效果如图 2-26 所示。

上面的代码是通过 <img> 标签来获取的，我们也可以通过 JavaScript 的 Image 对象来获取。具体使用方法如代码清单 2-24 所示。

代码清单 2-24

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
var img=document.getElementById("face");
img.onload = function(){
   ctx.drawImage(img,10,10);
```

```
};
</script>
```

图 2-26 代码清单 2-23 运行后的效果图

代码解析

首先建立 Image 对象。

```
var image = new Image();
```

然后通过设置 src 属性，来载入图片。

```
image.src = "face.jpg";
```

接着添加 onload 事件侦听，当图片载入完成时将其绘制到画板上。

```
image.onload = function(){
   ctx.drawImage(image,10,10);
};
```

运行效果如图 2-27 所示。

图 2-27 代码清单 2-24 运行后的效果图

下面具体看一下 drawImage 函数的 3 种函数原型的用法与区别，如代码清单 2-25 所示。

代码清单 2-25

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
var image = new Image();
image.src = "face.jpg";
image.onload = function(){
   ctx.drawImage(image,10,10);
   ctx.drawImage(image,260,10,100,100);
   ctx.drawImage(image,50,50,100,100,260,130,100,100);
};
</script>
```

代码解析

下面的代码表示从坐标 (10,10) 开始绘制整张图片。

```
ctx.drawImage(image,10,10);
```

下面的代码表示从坐标 (260,10) 开始绘制整张图片到长 100、宽 100 的矩形区域内。

```
ctx.drawImage(image,260,10,100,100);
```

下面的代码表示截取图片从 (50,50) 到 (100,100) 的部分，从坐标 (260,130) 开始绘制，放到长 100、宽 100 的矩形区域内。

```
ctx.drawImage(image,50,50,100,100,260,130,100,100);
```

运行效果如图 2-28 所示。

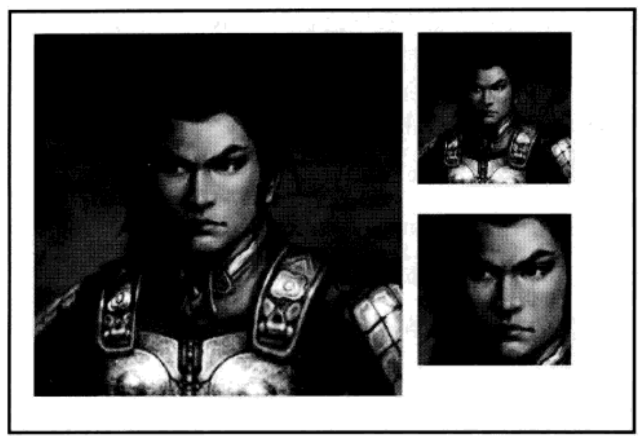


图 2-28　代码清单 2-25 运行后的效果图

2.4.2　利用 getImageData 和 putImageData 绘制图片

将图片绘制到 Canvas 画板上还有另一种方法，就是使用 putImageData(imgdata,dx,dy,sx,sy,sw,sh) 函数。putImageData 函数有 7 个参数，其中 imgdata 为像素数据，dx、dy 是绘制图片的定位坐标值，sx、sy 是 imgdata 所要绘制图片的起始位置，sw、sh 是 imgdata 所要绘制区域（相对 imgdata 的 sx 和 sy 坐标的偏移量）的宽度和高度值。值得一提的是，这里面第 4 个参数以及其后的所有参数都可以省略，如果这些参数都省略了，则表示绘制整个 imgdata。

在使用 putImageData 函数前，需要先用 getImageData(x,y,w,h) 函数得到像素数据，这里指的是从 Canvas 画板上取得所选区域的像素数据，它的 4 个参数分别是选择区域起点的坐标 x 和坐标 y，选择区域的长和宽。putImageData(imgdata,dx,dy,x,y,w,h) 函数则表示将所得到的像素数据描画到 Canvas 画板上形成图形。

下面是 putImageData 函数的使用过程，如代码清单 2-26 所示。

代码清单　2-26

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
var image = new Image();
image.src = "face.jpg";
image.onload = function(){
    ctx.drawImage(image,10,10);
    var imgData=ctx.getImageData(50,50,200,200);
    ctx.putImageData(imgData,10,260);
     ctx.putImageData(imgData,200,260,50,50,100,100);
};
</script>
```

PDG

代码解析

图片数据读取完成后，首先将图片数据绘制到 Canvas 画板上。

```
ctx.drawImage(image,10,10);
```

然后用 getImageData 函数从画板上取得像素数据。

```
var imgData=ctx.getImageData(50,50,200,200);
```

将所取得的整个像素数据画到 Canvas 画板上。

```
ctx.putImageData(imgData,10,260);
```

将所取得的像素数据的一部分画到 Canvas 画板上。

```
ctx.putImageData(imgData,200,260,50,50,100,100);
```

运行效果如图 2-29 所示。

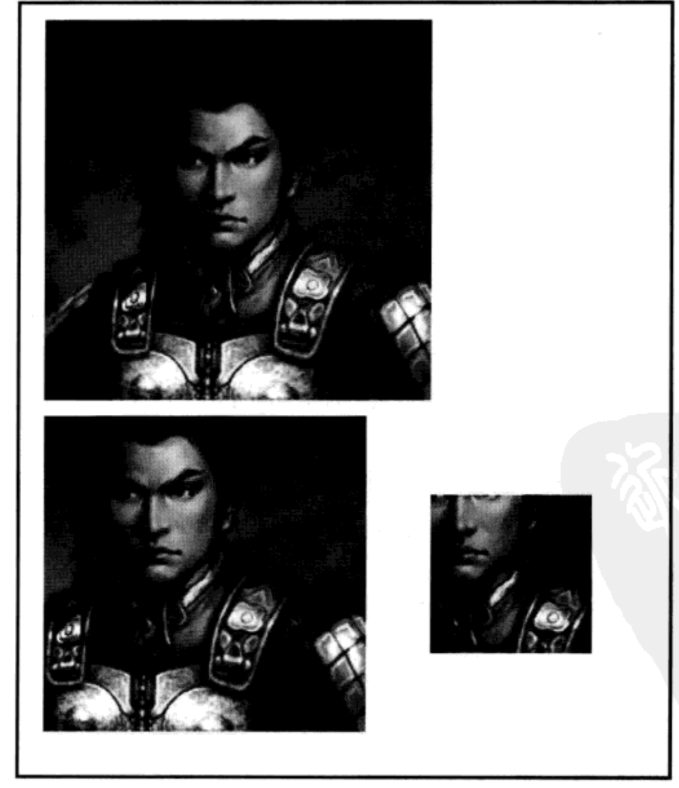


图 2-29 代码清单 2-26 运行后的效果图

PDG

注意　代码清单 2-26 中使用了 getImageData 函数获取图片数据，此函数在 Google Chrome 等浏览器中会涉及跨域问题，所以无法直接在浏览器中浏览，必须通过服务器来访问。我们在第 1 章配置环境的时候已经安装了本地服务器，参照 1.5 节的内容，将本地的代码文件放到本地服务器上，就可以看到测试结果了。以后凡是用到 getImageData 函数的地方，一定要使用此方法进行测试，此处不赘述。

2.4.3　利用 createImageData 新建像素

createImageData 函数有两种函数原型，其语法分别如下所示：

```
createImageData(sw, sh);
```

其一，返回指定大小的 imageData 对象。

```
createImageData(imagedata);
```

其二，返回与指定对象相同大小的 imageData 对象。

首先需要知道的是，通过 createImageData 返回的是一个空的 imageData 对象，必须要针对其像素进行赋值才能显示到 Canvas 画板上。下面我们通过例子来看看这两种原型的使用方法与区别，如代码清单 2-27 所示。

代码清单　2-27

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
var image = new Image();
image.src = "face.jpg";
image.onload = function(){
    ctx.drawImage(image,10,10);
    var imgData=ctx.getImageData(50,50,200,200);

    var imgData01=ctx.createImageData(imgData);
    for (i=0; i<imgData01.width*imgData01.height*4;i+=4){
            imgData01.data[i+0]=255;
            imgData01.data[i+1]=0;
            imgData01.data[i+2]=0;
            imgData01.data[i+3]=255;
    }
    ctx.putImageData(imgData01,10,260);

    var imgData02=ctx.createImageData(100,100);
    for (i=0; i<imgData02.width*imgData02.height*4;i+=4){
            imgData02.data[i+0]=255;
            imgData02.data[i+1]=0;
            imgData02.data[i+2]=0;
            imgData02.data[i+3]=155;
    }
```

```
    ctx.putImageData(imgData02,220,260);
};
</script>
```

代码解析

图片数据读取完成后，首先将图片数据绘制到 Canvas 画板上。

```
ctx.drawImage(image,10,10);
```

然后用 getImageData 函数从画板上取得像素数据。

```
var imgData=ctx.getImageData(50,50,200,200);
```

使用 createImageData 返回与 imgData 相同大小的 ImageData 对象。

```
var imgData01=ctx.createImageData(imgData);
```

下面 imgData01 进行赋值。

```
for (i=0; i<imgData01.width*imgData01.height*4;i+=4){
     imgData01.data[i+0]=255;
     imgData01.data[i+1]=0;
     imgData01.data[i+2]=0;
     imgData01.data[i+3]=255;
}
```

利用 putImageData 将 imgData01 画到 Canvas 画板上。

```
ctx.putImageData(imgData01,10,260);
```

使用 createImageData 返回一个大小为 100 × 100 的 ImageData 对象。

```
var imgData02=ctx.createImageData(100,100);
```

对 imgData02 进行赋值。

```
for (i=0; i<imgData02.width*imgData02.height*4;i+=4){
    imgData02.data[i+0]=255;
    imgData02.data[i+1]=0;
    imgData02.data[i+2]=0;
    imgData02.data[i+3]=155;
}
```

利用 putImageData 将 imgData02 画到 Canvas 画板上。

```
ctx.putImageData(imgData02,220,260);
```

运行效果如图 2-30 所示。

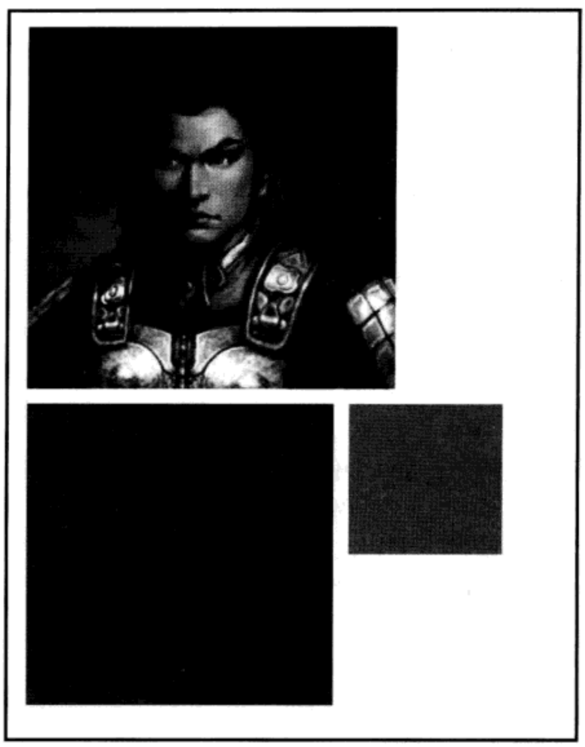


图 2-30 代码清单 2-27 运行后的效果图

2.5 小结

本章主要介绍了如何直接使用 Canvas 的绘图 API 进行基本图形、文字及图片的绘制。这些都属于最基本的图形绘制，因为它们不包括任何变形。但是这些看似简单的图形却是一切高级图形的基础，无论多复杂的图形都是由这些基本图形通过组合、变形等而得到的。至于如何将这些基本的图形通过变形及色彩的渲染等方法绘制出特殊的效果来，将会在下一章做详细的讲解。

第 3 章　Canvas 高级功能

在进行游戏开发的时候，离不开图形的变化，比如放大、缩小、旋转、平移等功能都是游戏开发中常用的变形手段。同时，为了使得游戏画面更加丰富多彩，还需要设置界面的颜色来实现更加精美的画面。本章将通过介绍 Canvas 的变形、色彩调整等高级功能来实现这些奇妙的变化。

3.1　变形

默认情况下，一个画布的坐标空间会使用画布的左上角 (0,0) 作为原点，x 值向右增加，y 值向下增加。这个坐标空间中的单位通常会被转换为像素，然后，可通过转换坐标空间在绘图过程中实现移动、缩放或旋转等操作。这些操作是通过 translate()、scale() 和 rotate() 等方法来实现的，它们会对画布的变换矩阵产生影响。

3.1.1　放大与缩小

我们用 scale 函数来实现图形的放大和缩小。

其函数原型如下：

```
scale(x, y);
```

其中，第一个参数 x 表示在 x 轴进行缩放，即水平缩放，第二个参数 y 表示在 y 轴上进行缩放，即在竖直方向上进行缩放。

具体实现方法如代码清单 3-1 所示。

代码清单　3-1

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(10,10,150,100);

ctx.scale(3,3);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(10,10,150,100);
</script>
```

运行后的效果如图 3-1 所示。

下面解释一下代码清单 3-1 的代码。首先，以页面上的坐标 (10,10) 为起点开始画了一个宽 150、高 100 的黑色矩形，如下所示：

```
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(10,10,150,100);
```

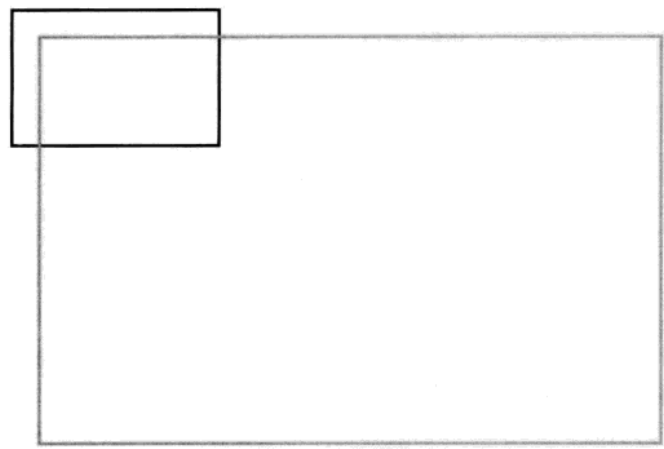


图 3-1　放大功能运行效果

然后，沿着 x 轴和 y 轴方向将其值放大 3 倍，并用同样的代码重新画一个灰色的矩形，如下所示：

```
ctx.scale(3,3);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(10,10,150,100);
```

从运行效果图可以看到，虽然用了同样的代码，但是第二次画的灰色矩形被放大了 3 倍，并且起始坐标值也一起被放大了 3 倍。

上述代码清单 3-1 实现了放大功能，那么现在来看看缩小功能如何实现，如代码清单 3-2 所示。

代码清单　3-2

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(50,50,150,100);

ctx.scale(0.5,0.5);
ctx.beginPath();
```

PDG

```
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(50,50,150,100);
</script>
```

运行后的效果如图 3-2 所示。

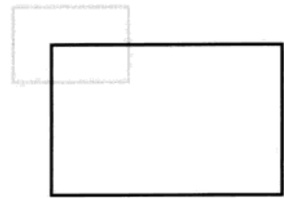


图 3-2 缩小功能运行效果

在代码清单 3-2 中，同样先画出了一个黑色矩形，然后沿着 x 轴和 y 轴方向上分别将其值缩小到了 0.5 倍，并用同样的代码重新画了一个灰色的矩形。

从运行效果图可以看到，虽然用了同样的代码，但是第二次画的灰色矩形被缩小为原来的 0.5 倍了，并且起始坐标值也一起缩小了 0.5 倍。

另外还需要了解的是，使用 scale 函数时，如果将参数设置为负数，还可以使图形翻转，比如代码清单 3-3 就实现了竖直方向的翻转。

代码清单 3-3

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>
<canvas id="myCanvas" width="500" height="350" style="background-color: #cccccc;">
你的浏览器不支持 HTML5
</canvas>
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var image = new Image();
image.src = "face.jpg";
image.onload = function(){
   ctx.drawImage(image,10,10);
   ctx.scale(1,-1);
   ctx.drawImage(image,250,-250);
};
</script>
</body>
</html>
```

运行效果如图 3-3 所示。

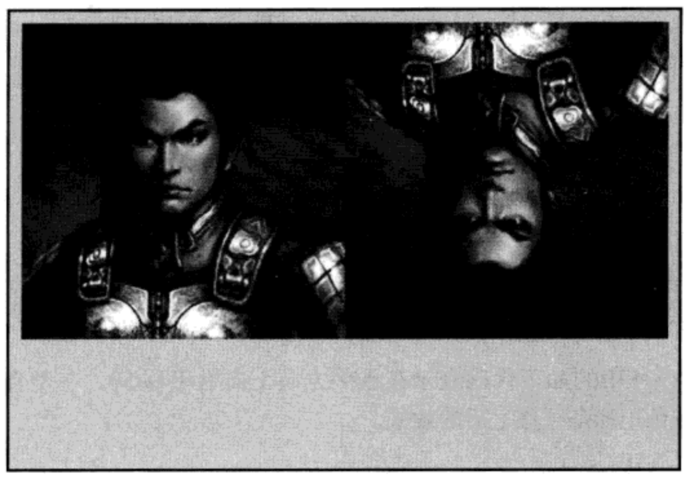


图 3-3　利用 scale 属性实现翻转

3.1.2　平移

使用 translate 函数可以实现对图形进行平移的功能。

其函数原型如下：

```
translate (x, y);
```

其中，第一个参数 x 表示在 x 轴进行平移，即在水平方向上平移，第二个参数 y 表示在 y 轴上进行缩放，即在竖直方向上平移。

具体的实现方法如代码清单 3-4 所示。

代码清单　3-4

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(10,10,150,100);

ctx.translate(50,100);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(10,10,150,100);
</script>
```

运行效果如图 3-4 所示。

PDG

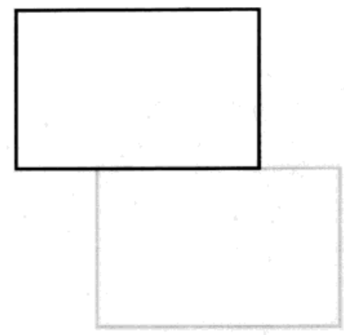


图 3-4 平移效果

在代码清单 3-4 中的如下代码表示在水平方向上向右平移 50，在竖直方向上向下平移 100，图 3-4 中灰色矩形是平移之后的效果。

```
ctx.translate(50,100);
```

3.1.3 旋转

利用 rotate 函数可以实现图形的旋转功能。

其函数原型如下 ：

```
rotate (angle);
```

这里需要注意的是，传入 rotate 里的参数 angle 是弧度而不是角度。如果角度为 angle，那么换算为弧度就是 angle*Math.PI/180，具体实现方法如代码清单 3-5 所示。

代码清单 3-5

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(200,50,100,50);

ctx.rotate(45*Math.PI/180);
ctx.beginPath();
ctx.strokeStyle = "#ff0000";
ctx.strokeRect(200,50,100,50);
</script>
```

运行效果如图 3-5 所示。

在图 3-5 中灰色矩形是旋转 45 度后的图形，可以看到用 rotate 来实现旋转时，是以 Canvas 的起始坐标 (0,0) 为中心进行旋转的。如果要想让图形以自己为中心旋转，则需要使用 3.1.2 节中的 translate 函数，如代码清单 3-6 所示。

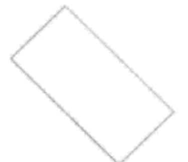


图 3-5 旋转效果

代码清单 3-6

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(200,50,100,50);

ctx.translate(250,75);
ctx.rotate(45*Math.PI/180);
ctx.translate(-250, -75);
ctx.beginPath();
ctx.strokeStyle = "#ff0000";
ctx.strokeRect(200,50,100,50);
</script>
```

运行效果如图 3-6 所示。

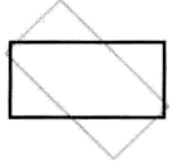


图 3-6 以自己为中心旋转的效果

在代码清单 3-6 中，下面的代码表示先将 Canvas 的起始坐标向右移 250，向下移 75，即移至所画矩形的中心处；然后开始旋转 45 度；接着再将 Canvas 的起始坐标向左移 250，向上移 75，即移回到原来位置，这样就完成了图形以自己为中心的旋转。

```
ctx.translate(250,75);
```

```
ctx.rotate(45*Math.PI/180);
ctx.translate(-250, -75);
```

3.1.4 利用 transform 矩阵实现多样化的变形

上面分别讲了缩放、平移以及旋转的实现方法。其实所有这些变形都是可以通过变形矩阵 transform 来实现的。先来看看 transform 函数的定义。

transform 函数的原型如下：

```
transform (a,b,c,d,e,f);
```

该函数的各个变量对应以下变换矩阵中相应位置的参数。

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$

下面来说明如何使用这个变换矩阵来实现上面的各种变形。

1. 缩放

假设原始坐标为 (x,y)，缩放后的坐标为 (x1,y1)，缩放的倍数分别为 a 和 d，那么就有下列公式：

$$x1=a*x$$
$$y1=d*y$$

因此，我们得到了如下矩阵公式。

$$\begin{pmatrix} x1 \\ y1 \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

这样就可以用 transform (a,0,0,d,0,0) 来替换 scale(a,d) 了，如代码清单 3-7 所示。

代码清单 3-7

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(10,10,150,100);

ctx.transform(3,0,0,3,0,0);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(10,10,150,100);
</script>
```

运行效果如图 3-7 所示。

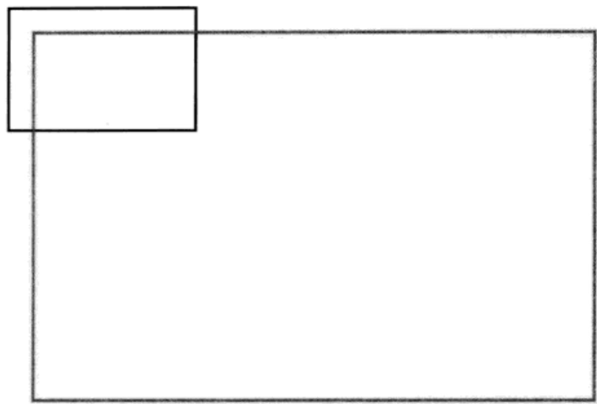


图 3-7　放大效果

可以看到，这里用代码清单 3-7 实现了和代码清单 3-1 一样的功能。

2. 平移

同样，假设原始坐标为 (x,y)，平移后的坐标为 (x1,y1)，在 x 轴和 y 轴的平移量分别为 e 和 f，那么就有下列公式：

$$x1=x+e$$
$$y1=y+f$$

因此，我们得到如下矩阵公式。

$$\begin{pmatrix} x1 \\ y1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & e \\ 0 & 1 & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

这样就可以用 transform (1,0,0,1,e,f) 来替换 translate (e,f) 了，如代码清单 3-8 所示。

代码清单　3-8

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(10,10,150,100);

ctx.transform (1,0,0,1,50,100);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(10,10,150,100);
</script>
```

运行效果如图 3-8 所示。

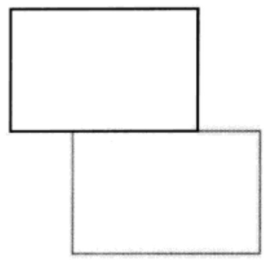


图 3-8 平移效果

可以看到，这里用代码清单 3-8 实现了和代码清单 3-4 一样的功能。

3. 旋转

旋转对应的矩阵公式如下。

$$\begin{pmatrix} x1 \\ y1 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

现在可以用 transform ($\cos\theta$, $\sin\theta$, $-\sin\theta$, $\cos\theta$,0,0) 来替换 rotate (θ) 了，如代码清单 3-9 所示。

代码清单 3-9

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(200,50,100,50);

ctx.transform (Math.cos(45*Math.PI/180),
    Math.sin(45*Math.PI/180),
    -Math.sin(45*Math.PI/180),
    Math.cos(45*Math.PI/180),0,0);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(200,50,100,50);
</script>
</body>
```

运行效果如图 3-9 所示。

可以看到，这里用代码清单 3-9 实现了和代码清单 3-5 一样的功能。

这样，我们就用 transform 完成了所有变形。另外，变换矩阵也可以通过 setTransform 函数来实现，setTransform 的参数与 transform 一样，不同的是，setTransform 函数是先消去之前的 transform 变换，然后重新进行变换的。为了区分 transform 和 setTransform，我们看看

PDG

下面的例子，如代码清单 3-10 所示。

图 3-9　旋转效果

代码清单　3-10

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(200,50,100,50);

ctx.transform (Math.cos(5*Math.PI/180),
    Math.sin(5*Math.PI/180),
    -Math.sin(5*Math.PI/180),
    Math.cos(5*Math.PI/180),0,0);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(200,50,100,50);

ctx.transform (Math.cos(10*Math.PI/180),
    Math.sin(10*Math.PI/180),
    -Math.sin(10*Math.PI/180),
    Math.cos(10*Math.PI/180),0,0);
ctx.beginPath();
ctx.strokeStyle = "#999999";
ctx.strokeRect(200,50,100,50);
</script>
```

运行效果如图 3-10 所示。

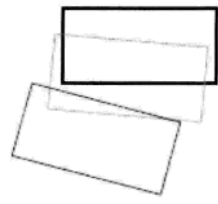


图 3-10　使用 transform 实现旋转效果

在代码清单 3-10 中，首先旋转 5 度，然后再旋转 10 度，所以第二次旋转相当于旋转了 15 度。

下面再看看代码清单 3-11 中 setTransform 的使用方法。

代码清单 3-11

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.beginPath();
ctx.strokeStyle = "#000000";
ctx.strokeRect(200,50,100,50);

ctx.transform (Math.cos(5*Math.PI/180),
    Math.sin(5*Math.PI/180),
    -Math.sin(5*Math.PI/180),
    Math.cos(5*Math.PI/180),0,0);
ctx.beginPath();
ctx.strokeStyle = "#cccccc";
ctx.strokeRect(200,50,100,50);

ctx.setTransform (Math.cos(10*Math.PI/180),
    Math.sin(10*Math.PI/180),
    -Math.sin(10*Math.PI/180),
    Math.cos(10*Math.PI/180),0,0);
ctx.beginPath();
ctx.strokeStyle = "#999999";
ctx.strokeRect(200,50,100,50);
</script>
```

运行效果如图 3-11 所示。

可以看到，第二次的旋转没有在第一次旋转的基础上进行，因为用 setTransform 实现旋转效果的时候，已经清除了上一个 transform，所以只旋转了 10 度。

4. 倾斜

接着来看一下如何实现图像的倾斜效果。首先看图 3-12。

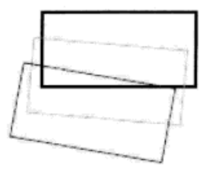


图 3-11 使用 setTransform 实现旋转效果

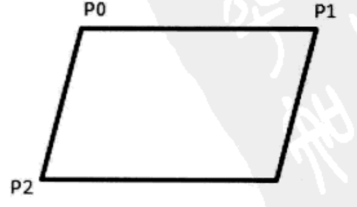


图 3-12 倾斜效果 1

图中的 3 个点 p0、p1、p2 遵循以下矩形公式。

$$\begin{pmatrix} (p1.x-p0.x)/width & (p2.x-p0.x)/height & p0.x \\ (p1.y-p0.y)/width & (p2.y-p0.y)/height & p0.y \\ 0 & 0 & 1 \end{pmatrix}$$

将上面的矩形公式带入 setTransform 函数中，如代码清单 3-12 所示。

代码清单　3-12

```
<script type="text/javascript">
var c=document.getElementById('myCanvas');
var ctx=c.getContext('2d');
ctx.setTransform(1,10/150,-40/100,1,40,10);
ctx.rect(50,50,150,100);
ctx.stroke();
</script>
```

运行效果如图 3-13 所示。

上述倾斜效果也可以根据另外 3 个点来实现，看图 3-14。

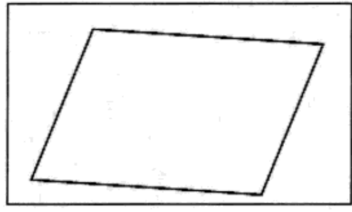


图 3-13 使用 setTransform 函数实现倾斜效果

图 3-14　倾斜效果 2

图中的 3 个点 p1、p2、p3 遵循以下矩形公式。

$$\begin{pmatrix} (p3.x-p2.x)/width & (p3.x-p1.x)/height & p2.x \\ (p3.y-p2.y)/width & (p3.y-p1.y)/height & p2.y \\ 0 & 0 & 1 \end{pmatrix}$$

将上面的矩形公式代入 setTransform 函数中，如代码清单 3-13 所示。

代码清单　3-13

```
<script type="text/javascript">
var c=document.getElementById('myCanvas');
var ctx=c.getContext('2d');
ctx.setTransform(130/150,-20/150,-20/100,80/100,0,0);
ctx.rect(50,50,150,100);
ctx.stroke();
</script>
```

运行效果如图 3-15 所示。

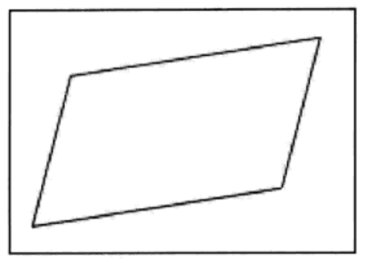


图 3-15 使用 setTransform 函数实现倾斜效果

5. 图片的扭曲效果

根据以上所介绍的内容，已经可以做到倾斜、旋转、平移和缩放 4 种基本变形，如图 3-16 所示。

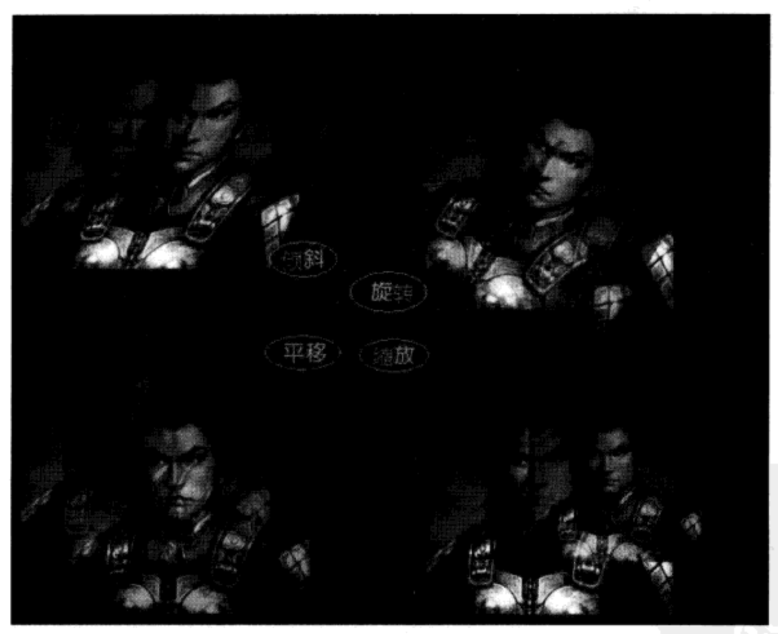


图 3-16 倾斜、旋转、平移和缩放 4 种变形

但是，对于非上述 4 种基本变形的特殊变形，就无法直接实现了。比如图 3-17 中的扭曲变形效果。

这时候，就需要通过多种变形组合来实现。先对图 3-17 进行分解，如图 3-18 所示。

此时分解出的两个图形可分别看作是两个基本变形的一部分，如图 3-19 所示。

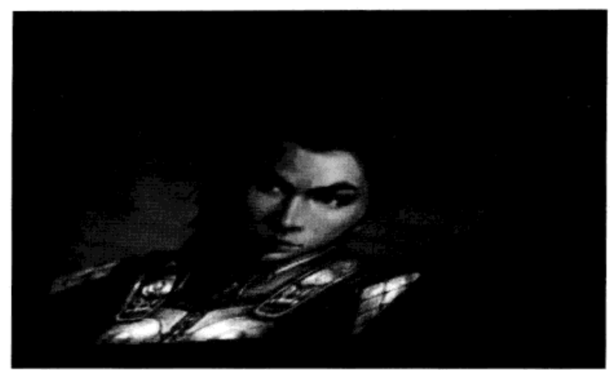


图 3-17　特殊变形

图 3-18　图 3-17 的分解图

图 3-19　分解出的两个图形

这样一来就简单了，只需要实现两个倾斜的变形，然后利用 clip 函数将图片的一部分绘制出来，就可以完成图 3-17 中的效果了，如代码清单 3-14 所示。

代码清单 3-14

```
<script type="text/javascript">
var c=document.getElementById('myCanvas');
var ctx=c.getContext('2d');
var img = new Image();
img.src="face.jpg";
img.onload = function(){
    ctx.save();
    ctx.beginPath();
    ctx.moveTo(80,0);
    ctx.lineTo(320,40);
    ctx.lineTo(0,200);
    ctx.closePath();
    ctx.clip();
    ctx.setTransform((320-80)/240,40/240,-80/240,200/240,80,0);
    ctx.drawImage(img,0,0);
    ctx.restore();

    ctx.save();
    ctx.beginPath();
    ctx.moveTo(320,40);
    ctx.lineTo(0,200);
    ctx.lineTo(200,150);
    ctx.closePath();
    ctx.clip();
    ctx.setTransform(200/240,(150-200)/240,(200-320)/240,(150-40)/240,0,200);
    ctx.drawImage(img,0,0-240);
    ctx.restore();
};
</script>
```

运行上面的代码，就可以得到图 3-17 的效果了。

解释一下代码清单 3-14。首先看一下左半边图形的绘制。

```
ctx.beginPath();
ctx.moveTo(80,0);
ctx.lineTo(320,40);
ctx.lineTo(0,200);
ctx.closePath();
ctx.clip();
```

上面这段代码是以 (80,0)、(320,40) 和 (0,200) 3 个点为顶点绘制一个三角形，然后利用 clip 函数将这个三角形作为绘图的可视区域。

```
ctx.setTransform(200/240,(150-200)/240,(200-320)/240,(150-40)/240,0,200);
```

上面的代码是以刚才的三角形的 3 个顶点来进行倾斜变形的，这个变形的原理已经在代码清单 3-12 中讲过了。

```
ctx.drawImage(img,0,0);
```

上面的代码是绘制图片，因为绘图的可视区域只是一个三角形，所以绘制完的图片只有一部分。

右半边图形的绘制思路和左半边是一样的。同时，两次绘图都加上了 save() 函数和 restore() 函数，这是为了让两次变形和绘图互不干涉。

3.2 图形的渲染

Canvas 提供了很多对颜色进行操作的 API，可实现渐变、反色等效果。这一节中将举例说明如何实现这些效果。

3.2.1 绘制颜色渐变效果的图形

颜色的渐变分为线性渐变和径向渐变，下面分别进行说明。

1. 线性渐变

使用 createLinearGradient 函数和 addColorStop 函数可以实现线性渐变功能。

createLinearGradient 函数的原型如下：

```
createLinearGradient(x1,y1,x2,y2)
```

其中的 4 个参数分别是渐变的出发点坐标 (x1，y1) 与终点坐标 (x2，y2)。

addColorStop 函数的原型如下：

```
addColorStop (position,color)
```

其中，position 参数必须是一个 0.0 到 1.0 之间的数值，表示渐变中颜色地点的相对地位；color 参数表示渐变的颜色。

它的绘制如代码清单 3-15 所示。

代码清单 3-15

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var grd=ctx.createLinearGradient(0,0,200,0);
grd.addColorStop(0.2,"#00ff00");
grd.addColorStop(0.8,"#ff0000");
ctx.fillStyle=grd;
ctx.fillRect(0,0,200,100);
</script>
```

绘制效果如图 3-20 所示。

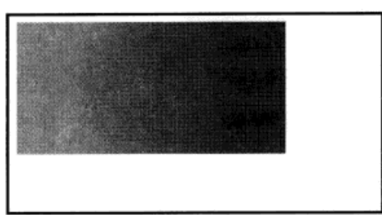


图 3-20 线性渐变效果

2. 径向渐变

使用 createRadialGradient 函数和 addColorStop 函数可以实现径向渐变功能。

createRadialGradient 函数的原型如下：

```
createRadialGradient (x0,y0,r0,x1,y1,r1)
```

其中，参数 x0、y0 为开始圆的圆心坐标，r0 为开始圆的直径；x1、y1 为结束圆的圆心坐标，r1 为结束圆的直径。

createRadialGradient 函数的使用方法如代码清单 3-16 所示。

代码清单 3-16

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var grd=ctx.createRadialGradient(100,100,10,100,100,50);
grd.addColorStop(0,"#00ff00");
grd.addColorStop(1,"#ff0000");
ctx.fillStyle=grd;
ctx.fillRect(0,0,200,200);
</script>
```

运行效果如图 3-21 所示。

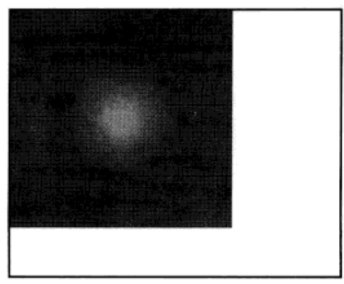


图 3-21 径向渐变效果

3.2.2 颜色合成之 globalCompositeOperation 属性

globalCompositeOperation 属性说明了绘制到画布上的颜色是如何与画布上已有的颜色组合起来的。下面列出了其中可能要设置的值以及它们的含义。这些值中的 source 一词指的是将要绘制到画布上的颜色，而 destination 指的是画布上已经存在的颜色，其默认值是 source-over。

- copy：只绘制新图形，删除其他所有内容。
- darker：在图形重叠的地方，其颜色由两个颜色值相减后决定。
- destination-atop：画布上已有的内容只会在它和新图形重叠的地方保留。新图形绘制于内容之后。
- destination-in：在新图形及画布上已有图形重叠的地方，画布上已有内容都保留。所有其他内容均为透明的。
- destination-out：在画布上已有内容和新图形不重叠的地方，已有内容保留。所有其他内容均为透明的。
- destination-over：新图形绘制于画布上已有内容的后面。
- lighter：在图形重叠的地方，其颜色由两种颜色值的加值来决定。
- source-atop：只有在新图形和画布上已有内容重叠的地方才绘制新图形。
- source-in：在新图形以及画布上已有内容重叠的地方才绘制新图形。所有其他内容均为透明的。
- ource-out：只有在和画布上已有图形不重叠的地方才绘制新图形。
- source-over：新图形绘制于画布上已有图形的顶部。这是默认的设置。
- xor：在重叠和正常绘制的其他地方，图形都成为透明的。

为了方便大家理解，我们来看看相关代码，如代码清单 3-17 所示。

代码清单 3-17

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.fillStyle="#00ff00";
ctx.fillRect(10,10,50,50);
ctx.globalCompositeOperation="source-over";
ctx.beginPath();
ctx.fillStyle="#ff0000";
ctx.arc(50,50,30,0,2*Math.PI);
ctx.fill();
</script>
```

运行效果如图 3-22 所示。

代码清单 3-14 中设置了 globalCompositeOperation 的属性为 source-over，表示新图形绘

制于已有图形的顶部。

其他属性实现的效果如图 3-23 至图 3-33 所示。

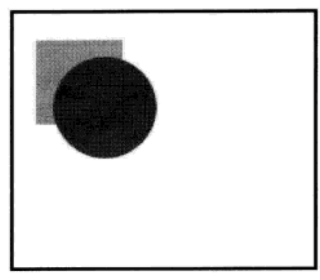


图 3-22 source-over 效果

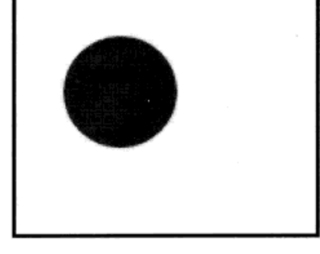


图 3-23 copy 属性实现的效果

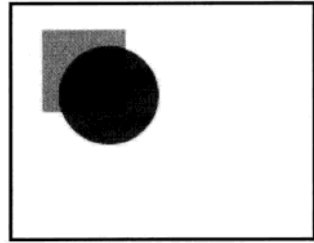


图 3-24 darker 属性实现的效果

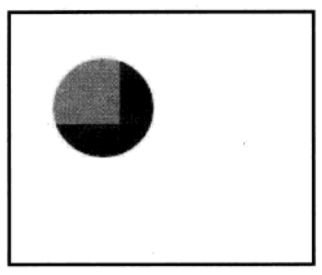


图 3-25 destination-atop 属性实现的效果

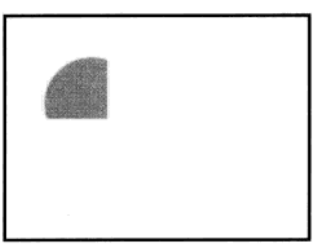


图 3-26 destination-in 属性实现的效果

图 3-27 destination-out 属性实现的效果

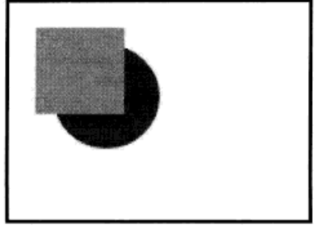


图 3-28 destination-over 属性实现的效果

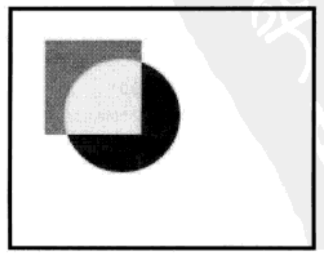


图 3-29 lighter 属性实现的效果

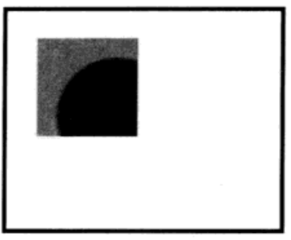


图 3-30 source-atop 属性实现的效果

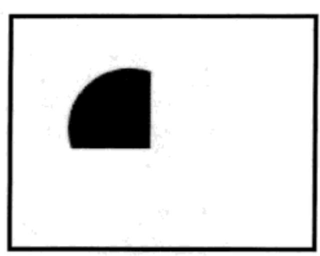


图 3-31 source-in 属性实现的效果

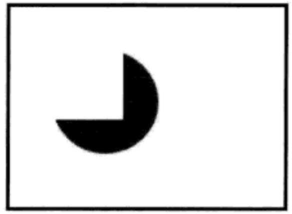


图 3-32 source-out 属性实现的效果

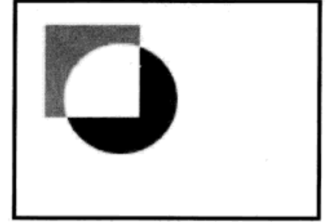


图 3-33 xor 属性实现的效果

以上就是 globalCompositeOperation 所有属性的效果图。

3.2.3 颜色反转

所谓颜色反转，就是对图形的每个像素进行颜色取反，如代码清单 3-18 所示。

代码清单 3-18

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var image = new Image();
image.src = "face.jpg";
image.onload = function(){
    ctx.drawImage(image,0,0);

    var imgdata = ctx.getImageData(0,0,250,250);
    var pixels = imgdata.data;
    // 遍历每个像素并对 RGB 值进行取反
    for (var i=0, n=pixels.length; i<n; i+= 4){
    pixels[i] = 255-pixels[i];
    pixels[i+1] = 255-pixels[i+1];
    pixels[i+2] = 255-pixels[i+2];
    }
    // 在指定位置进行像素重绘
    ctx.putImageData(imgdata, 250, 0);
};
</script>
```

运行效果如图 3-34 所示。

图 3-34 颜色反转效果

3.2.4 灰度控制

在游戏的制作过程中，有时候需要将图片变为灰色，其做法与 3.2.3 节所介绍的内容相似。首先对图形的每个像素进行灰度计算，如代码清单 3-19 所示。

代码清单 3-19

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

var image = new Image();
image.src = "face.jpg";
image.onload = function(){
    ctx.drawImage(image,0,0);

    var imgdata = ctx.getImageData(0,0,250,250);
    var pixels = imgdata.data;
     // 遍历每个像素并对 RGB 值进行取反
    for (var i=0, n=pixels.length; i<n; i+= 4){
          var grayscale = pixels[i] * .3 +
                 pixels[i+1] * .59 + pixels[i+2] * .11;
          pixels[i  ] = grayscale;     // red
          pixels[i+1] = grayscale;     // green
          pixels[i+2] = grayscale;     // blue
    }
    // 在指定位置进行像素重绘
    ctx.putImageData(imgdata, 250, 0);
};
</script>
```

运行效果如图 3-35 所示。

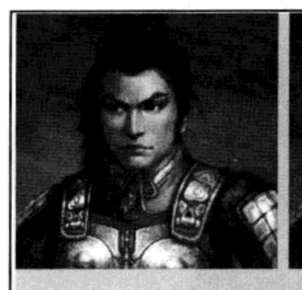


图 3-35　灰度效果

3.2.5　阴影效果

Canvas API 中包含了自动为你所绘制的任何图形添加下拉阴影的属性。阴影的颜色可用 shadowColor 属性来指定，并且可以通过 shadowOffsetX 和 shadowOffsetY 属性来改变。另外，应用到阴影边缘的羽化量也可以使用 shadowBlur 属性来设置。代码清单 3-20 给图片加上了红色阴影。

代码清单　3-20

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.shadowColor="#ff0000";
ctx.shadowBlur=10;
ctx.shadowOffsetX=20;
ctx.shadowOffsetY=30;

var image = new Image();
image.src = "face.jpg";
image.onload = function(){
   ctx.drawImage(image,0,0);

};
</script>
```

运行效果如图 3-36 所示。

在代码清单 3-20 中，下列代码表示设定阴影的颜色为红色。

```
ctx.shadowColor="#ff0000";
```

下列代码指定羽化阴影的程度为 10。

```
ctx.shadowBlur=10;
```

下列代码指定阴影的水平偏移量和垂直偏移量。

```
ctx.shadowOffsetX=20;
ctx.shadowOffsetY=30;
```

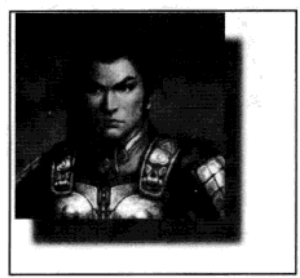


图 3-36 阴影效果

以上几个属性中从字面上不太容易理解的是羽化程度 shadowBlur。代码清单 3-21 将阴影的羽化程度设为了 100，这样大家就可以看到区别了。

代码清单 3-21

```
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.shadowColor="#ff0000";
ctx.shadowBlur=100;
ctx.shadowOffsetX=20;
ctx.shadowOffsetY=30;

var image = new Image();
image.src = "face.jpg";
image.onload = function(){
    ctx.drawImage(image,0,0);

};
</script>
```

运行效果如图 3-37 所示。

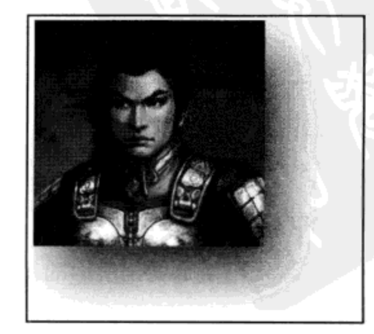


图 3-37 改变阴影羽化程度后的效果

3.3 自定义画板

前面的章节已经将 Canvas 的 API 大致介绍完毕了，下面我们来制作一个自定义画板，进一步熟悉一下这些 API 的用法。

3.3.1 画板的建立

建立一个画板的步骤如下：

（1）当鼠标按下的时候，开始描画，此处需要加入鼠标按下事件。

（2）当鼠标弹起的时候，结束描画，此处需要加入鼠标弹起事件。

（3）在鼠标按下并且移动的时候，在鼠标经过的路径上画线，此处需要加入鼠标移动事件。

代码清单 3-22 实现了建立一个简单画板的功能。

代码清单 3-22

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>

<canvas id="canvas" width="600" height="300""></canvas><br>

<script type="text/javascript">
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
//画一个黑色矩形
ctx.fillStyle="black";
ctx.fillRect(0,0,600,300);
//按下标记
var onoff = false;
var oldx = -10;
var oldy = -10;
//设置颜色
var linecolor = "white";
//设置线宽
var linw = 4;
//添加鼠标移动事件
canvas.addEventListener("mousemove",draw,true);
//添加鼠标按下事件
canvas.addEventListener("mousedown",down,false);
//添加鼠标弹起事件
canvas.addEventListener("mouseup",up,false);
function down(event){
    onoff = true;
    oldx = event.pageX-10;
    oldy = event.pageY-10;
}
function up(){
    onoff = false;
}
function draw(event){
    if(onoff == true){
            var newx = event.pageX-10;
            var newy = event.pageY-10;
```

```
            ctx.beginPath();
            ctx.moveTo(oldx,oldy);
            ctx.lineTo(newx,newy);
            ctx.strokeStyle=linecolor;
            ctx.lineWidth=linw;
            ctx.lineCap="round";
            ctx.stroke();

            oldx = newx;
            oldy = newy;
        };
};
</script>

</body>
</html>
```

运行效果如图 3-38 所示。

图 3-38 代码清单 3-22 的运行效果

代码清单 3-22 建立了一个黑色画板，当鼠标在画板上移动时，随鼠标的移动会画出白色线条。

在代码清单 3-22 中，下列代码画了一个黑色矩形区域，作为画板。

```
//画一个黑色矩形
ctx.fillStyle="black";
ctx.fillRect(0,0,600,300);
```

下面建立了 3 个变量，变量 onoff 用来控制鼠标是否按下，只有当鼠标按下的时候才会开始绘图。变量 oldx、oldy 表示鼠标发生移动前的坐标。

```
//按下标记
var onoff = false;
var oldx = -10;
var oldy = -10;
```

下面设置画笔的颜色为白色，线宽为 4。

```
//设置颜色
var linecolor = "white";
//设置线宽
var linw = 4;
```

下面给 Canvas 添加了鼠标按下侦听事件，当鼠标按下的时候，会调用 down 函数。

```
//添加鼠标按下事件
canvas.addEventListener("mousedown",down,false);
```

下面给 Canvas 添加了鼠标弹起侦听事件，当鼠标弹起的时候，会调用 up 函数。

```
//添加鼠标弹起事件
canvas.addEventListener("mouseup",up,false);
```

下面给 Canvas 添加了鼠标移动侦听事件，当鼠标在 Canvas 上移动的时候，会持续调用 draw 函数。

```
//添加鼠标移动事件
canvas.addEventListener("mousemove",draw,true);
```

下面看看 up、down、draw 3 个函数中的内容。

down 函数是在鼠标按下的时候调用的。当调用 down 函数的时候，会将 onoff 变量设置为 true，表示开始绘图，并给 oldx、oldy 赋予鼠标当前位置的坐标值。

```
function down(event){
    onoff = true;
    oldx = event.pageX-10;
    oldy = event.pageY-10;
}
```

up 函数是在鼠标弹起的时候调用的。当调用 up 函数的时候，将 onoff 变量设置为 false，表示结束绘图。

```
function up(){
    onoff = false;
}
```

draw 函数是在鼠标发生移动的时候不断持续调用的。当调用 draw 函数的时候，首先判断 onoff 变量的值，即判断鼠标是否处于按下状态，如果鼠标处于按下状态，则开始画线。

```
function draw(event){
    if(onoff == true){
        var newx = event.pageX-10;
        var newy = event.pageY-10;

        ctx.beginPath();
        ctx.moveTo(oldx,oldy);
```

```
            ctx.lineTo(newx,newy);
            ctx.strokeStyle=linecolor;
            ctx.lineWidth=linw;
            ctx.lineCap="round";
            ctx.stroke();

            oldx = newx;
            oldy = newy;
        };
    };
```

每次画线时，需要确定线条的起始位置和结束位置，线条的起始位置就是坐标(oldx,oldy)，然后把当前鼠标位置作为线条的结束位置，代码如下所示：

```
var newx = event.pageX-10;
var newy = event.pageY-10;
```

接着，利用 moveTo 和 lineTo 画线，代码如下所示：

```
ctx.beginPath();
ctx.moveTo(oldx,oldy);
ctx.lineTo(newx,newy);
ctx.strokeStyle=linecolor;
ctx.lineWidth=linw;
ctx.lineCap="round";
ctx.stroke();
```

上面的代码是画一条从坐标 (oldx,oldy) 到坐标 (newx,newy) 的线段，并设置了线条的颜色、宽度和线帽的类型。

在此次绘制结束后，新的鼠标位置将作为下一次画线的起始位置，代码如下所示：

```
oldx = newx;
oldy = newy;
```

上面的代码已经实现了一个最简单的画板功能。下面将其再完善一下，即加入按钮操作改变画笔颜色和线条宽度的功能。完整的代码如代码清单 3-23 所示。

代码清单 3-23

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>

<canvas id="canvas" width="600" height="300""></canvas><br>
<button style="width:80px;background-color:yellow;" onclick='linecolor =
"yellow";'>YELLOW</button>
<button style="width:80px;background-color:red;" onclick='linecolor =
```

```
"red";'>RED</button>
    <button style="width:80px;background-color:blue;" onclick='linecolor = "blue";'>
BLUE</button>
    <button style="width:80px;background-color:green;" onclick='linecolor = "green";'>
GREEN</button>
    <button style="width:80px;background-color:white;" onclick='linecolor = "white";'>
WHITE</button>
    <button style="width:80px;background-color:black;color:white;" onclick='linecolor
= "black";'>BLACK</button>

    <br>
    <button style="width:80px;background-color:white;" onclick="linw = 4;">4px</button>
    <button style="width:80px;background-color:white;" onclick="linw = 8;">8px</button>
    <button style="width:80px;background-color:white;" onclick="linw = 16;">16px</button>

    <script type="text/javascript">
    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    //画一个黑色矩形
    ctx.fillStyle="black";
    ctx.fillRect(0,0,600,300);
    //按下标记
    var onoff = false;
    var oldx = -10;
    var oldy = -10;
    //设置颜色
    var linecolor = "white";
    //设置线宽
    var linw = 4;
    //添加鼠标移动事件
    canvas.addEventListener("mousemove",draw,true);
    //添加鼠标按下事件
    canvas.addEventListener("mousedown",down,false);
    //添加鼠标弹起事件
    canvas.addEventListener("mouseup",up,false);
    function down(event){
        onoff = true;
        oldx = event.pageX-10;
        oldy = event.pageY-10;
    }
    function up(){
        onoff = false;
    }
    function draw(event){
        if(onoff == true){
            var newx = event.pageX-10;
            var newy = event.pageY-10;

            ctx.beginPath();
            ctx.moveTo(oldx,oldy);
            ctx.lineTo(newx,newy);
```

```
            ctx.strokeStyle=linecolor;
            ctx.lineWidth=linw;
            ctx.lineCap="round";
            ctx.stroke();

            oldx = newx;
            oldy = newy;
        };
    };

    </script>

    </body>
    </html>
```

运行效果如图 3-39 所示。

图 3-39 代码清单 3-23 的运行效果

在代码清单 3-23 中：

下面的代码加入了 6 个按钮，并加入了单击事件，当单击不同按钮的时候，就会相应地改变画笔的颜色。

```
    <button style="width:80px;background-color:yellow;" onclick='linecolor =
"yellow";'>YELLOW</button>
    <button style="width:80px;background-color:red;" onclick='linecolor =
"red";'>RED</button>
    <button style="width:80px;background-color:blue;" onclick='linecolor =
"blue";'>BLUE</button>
    <button style="width:80px;background-color:green;" onclick='linecolor =
"green";'>GREEN</button>
    <button style="width:80px;background-color:white;" onclick='linecolor =
"white";'>WHITE</button>
    <button style="width:80px;background-color:black;color:white;" onclick='linecolor
= "black";'>BLACK</button>
```

下面的代码加入了 3 个按钮，并加入了单击事件，当单击不同按钮的时候，就会相应地改变线条的宽度。

```
<button style="width:80px;background-color:white;" onclick="linw = 4;">4px</button>
<button style="width:80px;background-color:white;" onclick="linw = 8;">8px</button>
<button style="width:80px;background-color:white;" onclick="linw = 16;">16px</button>
```

3.3.2 Canvas 画布的导出功能

在 3.3.1 节中我们建立了一个画板，这一节再来给画板添加图片导出功能，即复制 Canvas 画板上的图像，使其保存为图片格式。

要将 Canvas 画板保存为图片格式，只需要使用下面的方法即可：

```
canvas.toDataURL("image/png");
```

现在可在页面上新建一个 <img >标签，然后将复制的 Canvas 内容用 <img> 表示出来。完整代码如代码清单 3-24 所示。

代码清单 3-24

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>
<canvas id="canvas" width="600" height="300""></canvas><br>
<button style="width:80px;background-color:yellow;" onclick='linecolor =
"yellow";'>YELLOW</button>
<button style="width:80px;background-color:red;" onclick='linecolor =
"red";'>RED</button>
<button style="width:80px;background-color:blue;" onclick='linecolor =
"blue";'>BLUE</button>
<button style="width:80px;background-color:green;" onclick='linecolor =
"green";'>GREEN</button>
<button style="width:80px;background-color:white;" onclick='linecolor =
"white";'>WHITE</button>
<button style="width:80px;background-color:black;color:white;" onclick='linecolor
= "black";'>BLACK</button>

<br>
<button style="width:80px;background-color:white;" onclick="linw = 4;">4px</button>
<button style="width:80px;background-color:white;" onclick="linw = 8;">8px</button>
<button style="width:80px;background-color:white;" onclick="linw = 16;">16px</button>
<br>
<button style="width:80px;background-color:pink;" onclick="copyimage();">EXPORT</button>
<br>
<img src="" id="image_png"  width="600" height="300">
<script type="text/javascript">
var canvas = document.getElementById("canvas");
```

```
var ctx = canvas.getContext("2d");
//画一个黑色矩形
ctx.fillStyle="black";
ctx.fillRect(0,0,600,300);
//按下标记
var onoff = false;
var oldx = -10;
var oldy = -10;
//设置颜色
var linecolor = "white";
//设置线宽
var linw = 4;
//添加鼠标移动事件
canvas.addEventListener("mousemove",draw,true);
//添加鼠标按下事件
canvas.addEventListener("mousedown",down,false);
//添加鼠标弹起事件
canvas.addEventListener("mouseup",up,false);
function down(event){
    onoff = true;
    oldx = event.pageX-10;
    oldy = event.pageY-10;
}
function up(){
    onoff = false;
}
function draw(event){
    if(onoff == true){
        var newx = event.pageX-10;
        var newy = event.pageY-10;

        ctx.beginPath();
        ctx.moveTo(oldx,oldy);
        ctx.lineTo(newx,newy);
        ctx.strokeStyle=linecolor;
        ctx.lineWidth=linw;
        ctx.lineCap="round";
        ctx.stroke();

        oldx = newx;
        oldy = newy;
    };
};
function copyimage(event){
    var img_png_src = canvas.toDataURL("image/png");
    document.getElementById("image_png").src = img_png_src;
}
</script>

</body>
</html>
```

运行效果如图 3-40 所示。

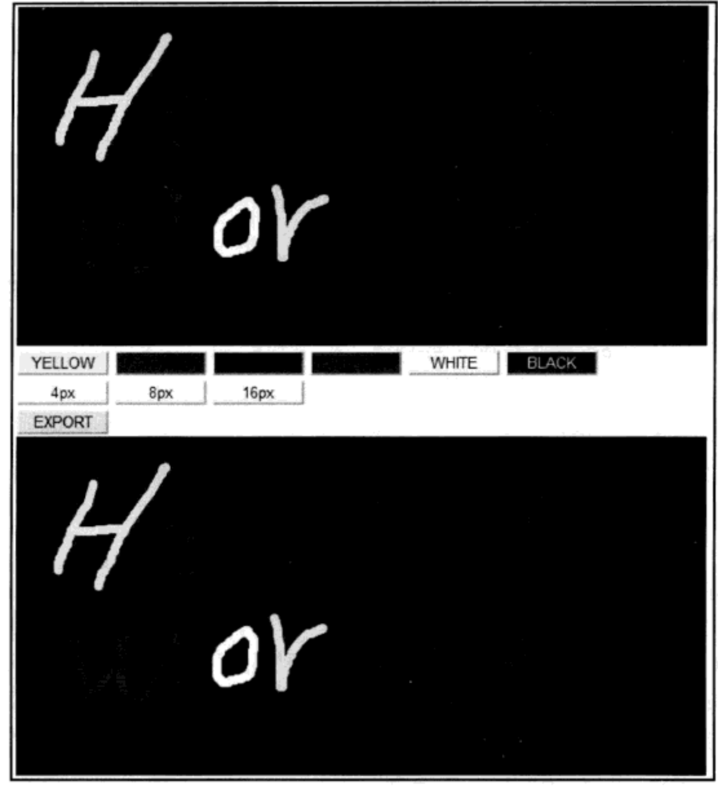


图 3-40　代码清单 3-24 的运行效果

如果想要将图片保存为图片文件，还需要借助 PHP 或 ASP 等工具，这里就不做讨论了。

3.4　小结

本章主要首先介绍了 Canvas 中的缩放、平移、旋转和倾斜等基本变形功能的实现，并且利用这些基本变形的组合操作实现了图片的扭曲；接着介绍了色彩的渲染；最后运用之前所介绍的内容制作了一个简单的自定义画板。到此 Canvas 的绘图相关 API 就基本上介绍完了。只要运用好这些 API，就可以制作出更多更复杂的效果。但是直接运用这些 API 来开发游戏是相当不方便的，一些很简单的功能可能都需要大量的代码来实现，比如本章的图片扭曲功能，必须根据图片扭曲后的顶点坐标去计算相应的参数，而制作游戏时可能会有更为复杂的图形要处理。所以从下一章开始将会介绍一款 HTML5 的开源库件 lufylegend，并介绍如何使用它来进行游戏的开发。

第 4 章　lufylegend 开源库件

虽然各大浏览器都支持 HTML5，但是我们在使用 HTML5 开发游戏的时候，依然面临着几个大问题。第一，各浏览器对 JavaScript 和 HTML 的解析是不一致的，例如在 IE 中鼠标点击时的 offsetX 属性在 Firefox 中则必须用 layerX 来代替，这就使得我们在开发游戏的时候必须考虑代码在各个浏览器中的兼容性问题，否则游戏将无法在所有的浏览器上都正常运行。第二，手机浏览器和 PC 浏览器也就是区别的，例如手机浏览器支持 touch 事件，但是不支持 mouse 事件，而 PC 浏览器则正好相反。第三，JavaScript 的并非面向对象编程，直接影响了代码的易读性。以上这些只是利用 HTML5 开发游戏或应用时必然会遇到的一部分问题，为了解决这些问题，最快捷的方法就是利用框架开发，因为一般来说，框架中会提供一些解决这类问题的方法。本书会详细介绍如何利用 HTML5 的开源框架 lufylegend 来进行游戏开发。

4.1　lufylegend 库件简介

lufylegend 库件是一个 HTML5 开源框架，它利用了 JavaScript 和 ActionScript 的相似性，用仿 ActionScript 3.0 的语法对 HTML5 Canvas 的 API 进行了封装，这不但实现了面向对象开发，而且使得 HTML5 的开发像使用 Flash 一样便利。lufylegend.js 支持 Google Chrome、Firefox、Opera、IE9、IOS、Android 等多种常用环境，而且实现了根据手机浏览器和 PC 浏览器的不同而自动对 mouse 事件和 touch 事件进行切换等的功能，这样就可以减少开发者在解决浏览器兼容方面问题时花费的工作量，从而可更专注于游戏设计方面的开发。

4.1.1　工作原理

lufylegend 库件封装了 Canvas 绘图的所有 API，它利用 JavaScript 的 setInterval 函数，对 Canvas 画板进行周期性重绘。每次对 Canvas 画板进行重绘的时候，首先要使用 clearRect 方法清空整个画板，然后再调用需要进行重绘的各个 API 函数，这样就可达到重新绘制所有图形的目的。

lufylegend 库件在进行初始化工作的时候，会根据浏览器的不同来判断需要加载的是 mouse 事件还是 touch 事件。在利用 lufylegend 库件加载点击事件的时候，只是将被加载的事件加入了框架预先准备的数组当中，当点击事件发生的时候才会调用这个数组里的事件，这样做的好处是，虽然被加载的点击事件只是初始化时的一个，但被储存在数组中的事件可以是多数个，这样就可以给多个对象加载点击事件，而 lufylegend 库件会根据玩家点击对象的不同自动调用相应的事件。

本书的源码中附带有 lufylegend 库件的 1.5.1 版，用户也可以从 lufylegend 库件的官网上下载其最新版，下载网址为：

http://lufylegend.com/lufylegend

lufylegend 库件有原版和 min 版两个版本，原版便于用户阅读和学习源码，min 版是去除了空格和回车等占位符号的压缩版本。

4.1.2　库件使用流程

lufylegend 库件的使用方法很简单，主要分为以下几个步骤：

（1）引入 lufylegend 库件。

（2）在 HTML 中创建一个 <div> 标签。

（3）使用 init 函数进行初始化工作。

代码清单 4-1 是 lufylegend 库件的使用流程。

代码清单　4-1

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8" />
    <script type="text/javascript" src="../../lufylegend-1.5.1.min.js"></script>
</head>
<body>
<div id="mylegend">loading...</div>
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
    alert("lufylegend start");
}
</script>
</body>
</html>
```

运行效果如图 4-1 所示。

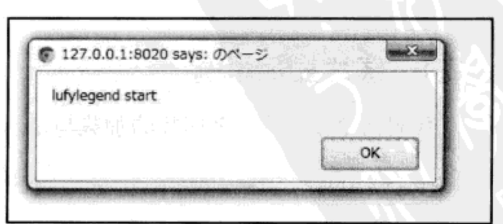


图 4-1　lufylegend 库件的运行效果

代码解析

首先引入 lufylegend 库件，下面的代码是引入库件的压缩版 1.4.0。

```
<script type="text/javascript"
src="../../lufylegend-1.5.1.min.js"></script>
```

要注意的是，引入的路径一定要正确。然后在 HTML 中创建一个 <div> 标签。

```
<div id="mylegend">loading...</div>
```

接下来是使用库件内置的 init 方法进行初始化。init 函数的原型如下所示：

```
init(speed,divid,width,height,completeFunc,type)
```

参数：

- speed：游戏速度设定。
- divid：传入一个 div 的 ID，库件初始化的时候，会将 Canvas 加入到此 div 内部。
- width：游戏界面宽。
- height：游戏界面高。
- completeFunc：游戏初始化完成后，调用此函数。

在下面的代码中，50 表示将游戏的速度设定为 50；mylegend 是之前创建的 <div> 标签的 ID；500 和 350 分别表示要创建的 Canvas 画板的长和宽；main 是初始化库件结束后要调用的函数的名称。

```
init(50,"mylegend",500,350,main);
```

在初始化结束后，会调用下面的 main 函数。

```
function main(){
    alert("lufylegend start");
}
```

上面的代码用于验证库件的引入和初始化工作是否成功，如果弹出了 lufylegend start 对话框，则表示库件的初始化工作正常。

为了使大家更容易理解 lufylegend 库件，接下来对它的原理做更详细的介绍。

4.2 图片的加载与显示

使用 lufylegend 库件显示图片时，可分为以下几个步骤：

(1) 使用 Lloader 类加载图片数据。

(2) 将读取完的图片数据保存到 LbitmapData 中。

(3) 利用 Lbitmap 将图片显示到画板上。

4.2.1 图片显示举例

代码清单 4-2 是一个图片的加载与显示过程。

代码清单 4-2

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8" />
    <script type="text/javascript" src="../../lufylegend-1.5.1.min.js"></script>
</head>
```

PDG

```
<body>
<div id="mylegend">loading...</div>
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var bitmap = new LBitmap(bitmapdata);
    addChild(bitmap);
}
</script>
</body>
</html>
```

运行效果如图 4-2 所示。

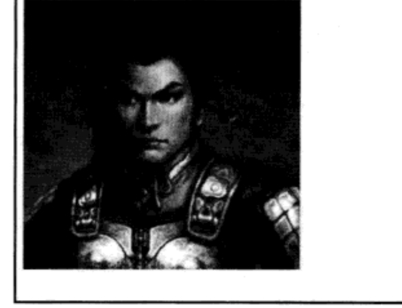


图 4-2 lufylegend 库件显示图片的运行效果

代码解析

下面这段代码表示读取图片：

```
loader = new LLoader();
loader.addEventListener(LEvent.COMPLETE,
loadBitmapdata);
loader.load("face.jpg","bitmapData");
```

这跟下面代码的功能一样，其实就是给 Image 加了一个 onload 事件。

```
var image = new Image();
image.onload = function(){
};
```

在图片读取完成后会调用 loadBitmapdata 函数，而此时的 loader.content 就是一个 Image。

```
var bitmapdata = new LBitmapData(loader.content);
```

上面的代码会新建一个 LBitmapData 对象，并将已读取完的 Image 作为参数传给新建的 LBitmapData 对象。

LBitmapData 是 lufylegend 库件中的一个类，它只是用来保存和读取 Image 对象的，如果要将图片显示到 Canvas 画板上，则需要用到 LBitmap。来看下面的代码：

```
var bitmap = new LBitmap(bitmapdata);
addChild(bitmap);
```

这里新建了一个 LBitmap 对象，并将上面新建的 LBitmapData 对象作为参数传给了新建

的 LBitmap 对象。LBitmap 的功能是将 Image 对象显示到 Canvas 画板上。

addChild 函数是将对象添加到 Canvas 画板上，被 addChild 函数添加的对象会按照添加的先后顺序依次显示出来。

可以看到，用 lufylegend 库件显示图片，主要用到了 LBitmapData 和 LBitmap 两个对象。下面来细说一下这两个对象。

4.2.2 LBitmapData 对象

首先来看 LBitmapData 对象，它的构造器如下：

```
LBitmapData(image,x,y,width,height)
```

参数：

image:Image 对象。

- x:Image 可视范围 x 坐标。
- y:Image 可视范围 y 坐标。
- width:Image 可视范围宽。
- height:Image 可视范围高。

除了第一个参数外，其余 4 个参数都可以控制图片显示的范围，如图 4-3 所示。

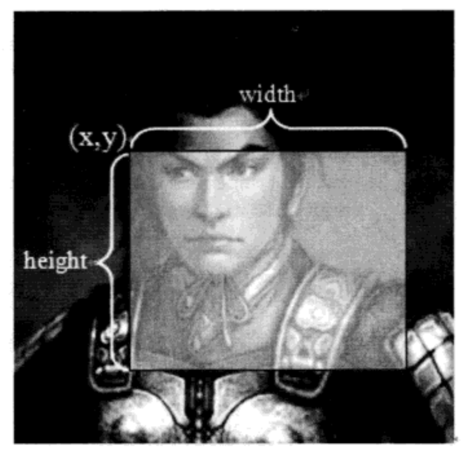


图 4-3 4 个参数控制图片显示范围

在代码清单 4-3 中设置了 LBitmapData 的显示范围。

代码清单 4-3

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata =
            new LBitmapData(loader.content,50,50,150,150);
    var bitmap = new LBitmap(bitmapdata);
    addChild(bitmap);
}
</script>
```

运行效果如图 4-4 所示。

将图 4-4 与代码清单 4-2 的运行效果图 4-2 做一下对照会发现，如果这 4 个参数省略的话，则会显示整张图片。

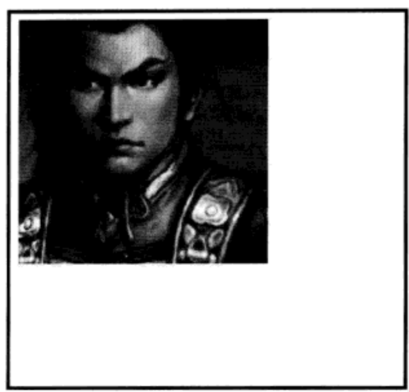


图 4-4　显示范围运行效果

4.2.3　LBitmap 对象

再来看 LBitmap 对象，LBitmap 不但能将图片显示到 Canvas 画板上，还可以控制图片的各种属性，如坐标 (x,y)、透明度 (alpha)、旋转 (rotate)、缩放 (scaleX, scaleY) 等。在代码清单 4-4 中，则设置了图片的旋转和透明度。

代码清单　4-4

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var bitmap = new LBitmap(bitmapdata);
    //图片坐标
    bitmap.x = 50;
    bitmap.y = 50;
    //图片旋转60度
    bitmap.rotate = 60;
    //图片透明度设置为0.4
    bitmap.alpha = 0.4;
    addChild(bitmap);
}
</script>
```

运行效果如图 4-5 所示。

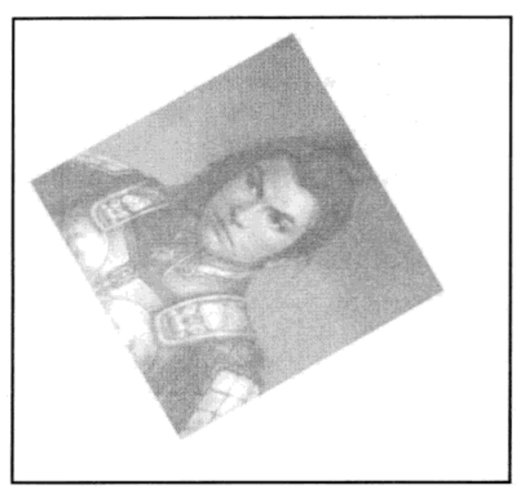


图 4-5 设置图片的旋转和透明度

4.3 层的概念

在游戏的世界里，我们可以看到各种地图及各种游戏人物，还能看到人物在地图上行走、对话等。无论是地图还是人物，其实都是图片的图像处理与显示的结果，让不同的图像按照先后顺序显示到屏幕上，我们就会看到不同的游戏界面。也就是说，这些图像显示的先后顺序以及位置决定了游戏的界面。

在图 4-6 中，LayerA 层是最下层，LayerB 层和 LayerC 层是第二层，LayerD 层在最上层。当它们分别出现在游戏界面上的时候，我们首先看到的是最上层的 LayerD 层，然后是 LayerB 层和 LayerC 层，最后看到的是 LayerA 层。

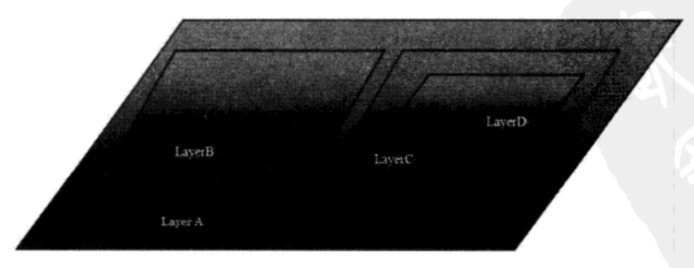


图 4-6 层的分布

所以，如果把不同的图像放到不同的层，就能决定这些图像在游戏画面里出现的顺序。

lufylegend.js 库件为 Canvas 实现了层的概念，它就是 LSprite 对象。它的用法很简单，如下面的代码实现了给游戏加入一个层的功能。

```
var layer = new LSprite();
addChild(layer);
```

有了层的概念，那么就可以将代码清单 4-1 调整为代码清单 4-5 了。

代码清单　4-5

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var bitmap = new LBitmap(bitmapdata);
    //加入层 LSprite
    var layer = new LSprite();
    addChild(layer);
    layer.addChild(bitmap);
}
</script>
```

运行效果如图 4-7 所示。

代码解析

下面的代码将 layer 层添加到画板上，然后再将 bitmap 添加到 layer 层。

```
var layer = new LSprite();
addChild(layer);
layer.addChild(bitmap);
```

LSprite 对象和 LBitmap 对象一样，也有坐标 (x,y)、透明度 (alpha)、旋转 (rotate)、缩放 (scaleX, scaleY) 等属性，不过它控制的是整个层的属性。如代码清单 4-6 使用了 LSprite 对象的旋转属性。

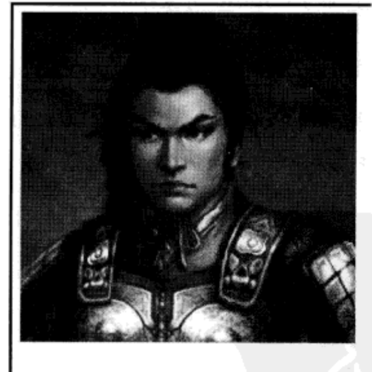


图 4-7　有了层的概念后的图片效果

代码清单　4-6

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
```

```
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var bitmap = new LBitmap(bitmapdata);
    //加入层 LSprite
    var layer = new LSprite();
    addChild(layer);
    layer.addChild(bitmap);
    layer.x = 50;
    layer.y = 50;
    layer.rotate = 60;
}
</script>
```

运行效果如图 4-8 所示。

图 4-8 使用了 LSprite 对象的旋转属性

4.4 使用 LGraphics 对象绘图

LGraphics 是 lufylegend 库件中的一个绘图类，它内置了一些函数以简化绘图。它可以单独使用，也可以与 LSprite 对象配合使用。

4.4.1 绘制矩形

首先，利用 LGraphics 对象中的 drawRect 函数来绘制矩形，代码如下所示：

```
drawRect (thickness,lineColor,pointArray,isfill,color)
```

参数：

- thickness：边框线宽。
- lineColor：边框颜色。
- pointArray：矩形范围数组 [坐标 x, 坐标 y, 长 , 宽]。
- isfill：是否填充矩形。
- color：填充颜色。

代码清单 4-7 是 drawRect 的一个使用示例。

代码清单 4-7

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
```

```
    var graphics = new LGraphics();
    addChild(graphics);
    graphics.drawRect(1,'#000000',[50,50,100,100]);
    graphics.drawRect(1,'#000000',
            [170,50,100,100],true,'#cccccc');
}
</script>
```

运行效果如图 4-9 所示。

图 4-9 drawRect 的使用效果

代码解析

下面代码建立了一个绘图对象，并将其加载到 Canvas 画板上。

```
var graphics = new LGraphics();
addChild(graphics);
```

下面表示画一个空心矩形。

```
graphics.drawRect(1,'#000000',[50,50,100,100]);
```

下面表示画一个填充矩形。

```
graphics.drawRect(1,'#000000',[170,50,100,100],true,'#cccccc');
```

4.4.2 绘制圆

利用 LGraphics 对象中的 drawArc 函数来绘制圆时，其代码如下所示：

```
drawArc (thickness,lineColor,pointArray,isfill,color)
```

参数：

- ❑ thickness：边框线宽。
- ❑ lineColor：边框颜色。
- ❑ pointArray：圆形参数数组 [坐标 x, 坐标 y, 半径，起始角度，结束角度，顺时针或逆时针]。
- ❑ isfill：是否填充。
- ❑ color：填充颜色。

代码清单 4-8 是 drawArc 的一个使用示例。

代码清单 4-8

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
    var graphics = new LGraphics();
    addChild(graphics);
    graphics.drawArc(1,'#000000',[60,60,50,0,360*Math.PI/180]);
    graphics.drawArc(1,'#000000',
            [180,60,50,0,360*Math.PI/180],true,'#cccccc');
```

```
}
</script>
```

运行效果如图 4-10 所示。

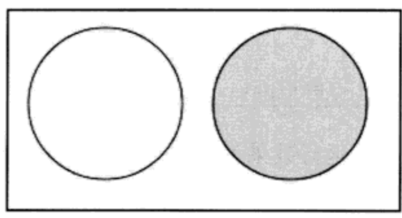


图 4-10 drawArc 的使用效果

代码解析

下面代码建立了一个绘图对象，并将其加载到 Canvas 画板上。

```
var graphics = new LGraphics();
addChild(graphics);
```

下面表示画一个空心圆。

```
graphics.drawArc(1,'#000000',[60,60,50,0,360*Math.PI/180]);
```

下面表示画一个填充圆。

```
graphics.drawArc(1,'#000000',
          [180,60,50,0,360*Math.PI/180],true,'#cccccc');
```

4.4.3 绘制任意多边形

LGraphics 对象除了可以画矩形和圆之外，还可以使用其中的 drawVertices 函数，根据坐标顶点数组来绘制图形，代码如下所示：

```
drawVertices (thickness,lineColor, vertices,isfill,color)
```

参数：

- ❑ thickness：边框线宽。
- ❑ lineColor：边框颜色。
- ❑ vertices：顶点数组 [[顶点 1],[顶点 2],[顶点 3]......]。
- ❑ isfill：是否填充。
- ❑ color：填充颜色。

需要注意的是，传入 drawVertices 的顶点数组中的顶点个数必须大于等于 3 个。代码清单 4-9 是 drawVertices 的一个使用示例。

代码清单 4-9

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
   var graphics = new LGraphics();
   addChild(graphics);

   graphics.drawVertices(1,'#000000',
   [[50,20],[80,20],[100,50],[80,80],[50,80],[30,50]]);
```

```
    graphics.drawVertices(1,'#000000',
    [[150,20],[180,20],[200,50],[180,80],[150,80],[130,50]],
    true,'#cccccc');
}
</script>
```

运行效果如图 4-11 所示。

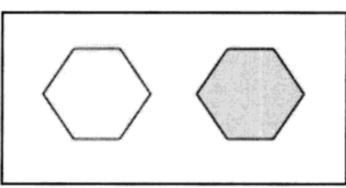


图 4-11　drawVertices 的使用效果

代码解析

下面代码建立了一个绘图对象，并将其加载到 Canvas 画板上。

```
var graphics = new LGraphics();
addChild(graphics);
```

下面表示画一个空心六角形。

```
graphics.drawVertices(1,'#000000',
    [[50,20],[80,20],[100,50],[80,80],[50,80],[30,50]]);
```

下面表示画一个填充六角形。

```
graphics.drawVertices(1,'#000000',
    [[150,20],[180,20],[200,50],[180,80],[150,80],[130,50]],
    true,'#cccccc');
```

4.4.4　使用 Canvas 的原始绘图函数进行绘图

LGraphics 对象可以使用 Canvas 的原始绘图函数进行绘图。代码清单 4-10 说明了如何在 LGraphics 对象中使用 Canvas 的原始绘图函数。

代码清单　4-10

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
    var graphics = new LGraphics();
    addChild(graphics);
    graphics.add(function(coodx,coody){
            LGlobal.canvas.strokeStyle = "#000000";
            LGlobal.canvas.moveTo(20,20);
            LGlobal.canvas.lineTo(200,200);
            LGlobal.canvas.stroke();
    });
}
</script>
```

运行效果如图 4-12 所示。

代码解析

下面的代码将建立一个绘图对象，并将其加载到 Canvas 画板上。

```
var graphics = new LGraphics();
addChild(graphics);
```

以下代码调用 LGraphics 对象中的 add 函数，传入绘图函数。

```
graphics.add(function(coodx,coody){
            LGlobal.canvas.strokeStyle = "#000000";
            LGlobal.canvas.moveTo(20,20);
            LGlobal.canvas.lineTo(200,200);
            LGlobal.canvas.stroke();
});
```

因为写在 fun 内部的 JavaScript 代码都会被运行，所以绘制图形的代码可以写在它里面。这里用到的 LGlobal.canvas 其实就是 getContext("2d") 所返回的对象。下面是 add 函数的原型：

```
add(fun)
```

参数：fun 即绘图 Function。

图 4-12　在 lufylegend 库件中使用原始绘图函数

4.4.5　使用 LSprite 对象进行绘图

上面介绍了独立使用 LGraphics 对象来绘图的方法。由于每个 LSprite 对象都包含一个 LGraphics 对象，所以上面的绘图都可以使用 LSprite 对象中的 graphics 来实现。比如代码清单 4-7 中画矩形的代码也可以写成代码清单 4-11。

代码清单　4-11

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
   var layer = new LSprite();
   addChild(layer);
   layer.graphics.drawRect(1,'#000000',[50,50,100,100]);
   layer.graphics.drawRect(1,'#000000',
                  [170,50,100,100],true,'#cccccc');
}
</script>
```

运行效果如图 4-13 所示，可以看到运行效果是一样的。

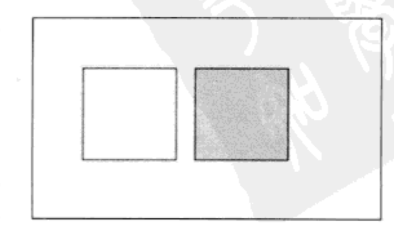


图 4-13　使用 LSprite 对象绘图效果

4.4.6　使用 LGraphics 对象绘制图片

使用 LGraphics 对象也可以直接绘制图片，主要是结合 LGraphics 对象的 beginBitmapFill 函数来实现的。下面来看看这个函数的强大功能。

PDG

1. 绘制一个圆形区域的图片

LGraphics 对象的 beginBitmapFill 函数可以和 drawArc 函数结合起来使用。具体实现看代码清单 4-12。

代码清单　4-12

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var backLayer = new LSprite();
    addChild(backLayer);
    backLayer.graphics.beginBitmapFill(bitmapdata);
    backLayer.graphics.drawArc(1,"#000",
                      [110,80,50,0,Math.PI*2]);
}
</script>
```

运行上面代码可以得到如图 4-14 所示效果。

代码解析

首先，利用 beginBitmapFill 函数将 LBitmapData 对象储存到 LGraphics 对象当中，代码如下：

```
backLayer.graphics.beginBitmapFill(bitmapdata);
```

图 4-14　使用 LGraphics 对象绘制圆形区域图片

然后再利用 drawArc 函数，绘制一个圆形区域。那么 LGraphics 对象会直接将储存在自己内部的 LBitmapData 对象透过这个圆形区域显示出来，代码如下：

```
backLayer.graphics.drawArc(1,"#000",[110,80,50,0,Math.PI*2]);
```

2. 绘制一个矩形区域的图片

LGraphics 对象的 beginBitmapFill 函数也可以和 drawRect 函数结合起来使用。具体实现看代码清单 4-13。

代码清单　4-13

```
<script type="text/javascript">
var loader;
```

```
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var backLayer = new LSprite();
    addChild(backLayer);
    backLayer.graphics.beginBitmapFill(bitmapdata);
    backLayer.graphics.drawRect(1,"#000",[80,50,70,100]);
}
</script>
```

运行上面代码可以得到如图 4-15 所示的效果。

图 4-15 使用 LGraphics 对象绘制矩形区域图片

代码解析

与绘制圆形区域图片一样，首先利用 beginBitmapFill 函数将 LBitmapData 对象储存到 LGraphics 对象当中，然后再利用 drawRect 函数，绘制一个矩形区域。那么 LGraphics 对象会直接将储存在自己内部的 LBitmapData 对象透过这个矩形区域显示出来，代码如下所示：

```
backLayer.graphics.drawRect(1,"#000",[80,50,70,100]);
```

3. 绘制一个多边形区域的图片

LGraphics 对象的 beginBitmapFill 函数还可以和 drawVertices 函数结合起来，绘制多边形区域的图形。具体实现看代码清单 4-14。

代码清单 4-14

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var backLayer = new LSprite();
    addChild(backLayer);
    backLayer.graphics.beginBitmapFill(bitmapdata);
    backLayer.graphics.drawVertices(1,"#000",
```

```
                        [[120,50],[100,200],[200,150]]);
}
</script>
```

运行上面代码可以得到如图 4-16 所示的效果。

代码解析

与绘制圆形和矩形区域图片一样，首先利用 beginBitmapFill 函数将 LBitmapData 对象储存到 LGraphics 对象当中，然后再利用 drawVertices 函数绘制一个多边形区域。那么 LGraphics 对象会直接将储存在自己内部的 LBitmapData 对象透过这个多边形区域显示出来，代码如下所示：

```
backLayer.graphics.drawVertices(1,"#000",
                        [[120,50],[100,200],[200,150]]);
```

图 4-16　使用 LGraphics 对象绘制多边形区域图片

4. 绘制一个扭曲的图片

在第 3 章中介绍了如何通过变形来绘制一种特殊效果的扭曲图片。那么如何利用 lufylegend 库件来实现这一功能呢？利用 LGraphics 对象的 beginBitmapFill 函数结合 drawTriangles 函数，就可以实现任意扭曲图形的绘制。先来看一下 drawTriangles 函数的定义：

```
drawTriangles(vertices, indices, uvtData,thickness,color)
```

参数：

- vertices：由数字构成的矢量，其中的每一对数字将被视为一个坐标位置（x, y 对）。该参数是必需要有的。
- indices：由整数或索引构成的矢量，其中每 3 个索引定义一个三角形。如果 indexes 参数为 null，则每 3 个顶点（vertices 矢量中的 6 对 x,y）定义一个三角形。否则，每个索引将引用一个顶点，即 vertices 矢量中的一对数字。例如，indexes[1] 引用 (vertices[2], vertices[3])。
- uvtData：由用于应用纹理映射的标准坐标构成的矢量。每个坐标引用用于填充的位图上的一个点。每个顶点必须具有一个 UV 或一个 UVT 坐标。对于 UV 坐标，(0,0) 是位图的左上角，(1,1) 是位图的右下角。
- thickness：分割完的三角形边框线宽。
- color：分割完的三角形边框颜色。

直接看上面的文字，恐怕不太容易理解，下面举几个例子说明。最后两个参数比较简单，先来说说这两个参数。图 4-17 是对最后两个参数（线宽设置为 2、颜色为白色）进行设置的效果图。

可以看到，图 4-17 实际上是由一些三角形拼接而成的，这就是 drawTriangles 函数的原理。它将一张图分解成 N 个小三角形，然后将这些小三角形进行变形、移动等，最后再将这些小三角形拼接成一张完整的图片，从而实现了图片的扭曲效果。

接下来介绍其他 3 个参数的用法。

第一个参数 vertices 其实就是定义每个顶点的坐标。这几个顶点的顺序如图 4-18 所示。

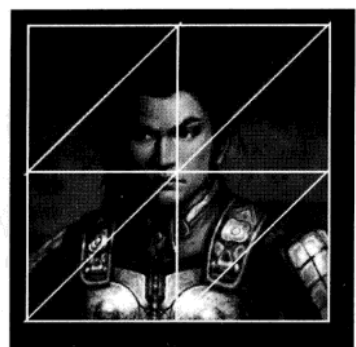


图 4-17 drawTriangles 函数绘制图片效果

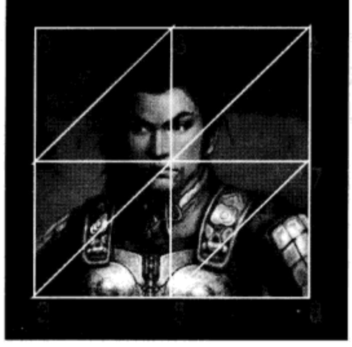


图 4-18 顶点顺序示意图

vertices 参数中储存的就是图 4-18 中 9 个顶点的坐标。如果图片的大小是 240 × 240，并且图中的三角形是等分的，那么代码如下：

```
vertices = new Array();
vertices.push(0, 0);
vertices.push(0, 120);
vertices.push(0, 240);
vertices.push(120, 0);
vertices.push(120, 120);
vertices.push(120, 240);
vertices.push(240, 0);
vertices.push(240, 120);
vertices.push(240, 240);
```

第二个参数 indices 表示定义三角形，数组 vertices 中每 3 个顶点可以组成一个三角形，indices 就是用来定义这些三角形的。这些三角形的顶点顺序是有规定的，其实从前面的图 4-17 和图 4-18 中可以看到，每两个三角形是一个矩形，定义三角形的时候，要以这些矩形的 4 个顶点为基准。三角形的顶点顺序分别是（左上，右上，左下）和（右上，左下，右下），如图 4-19 所示。

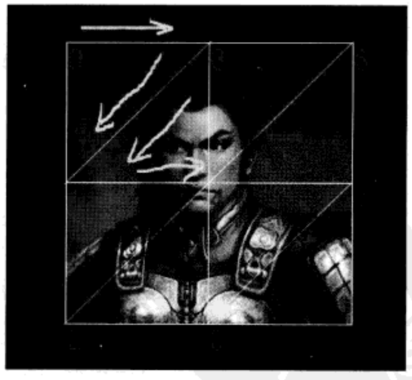


图 4-19 三角形顶点顺序示意图

对应图 4-19 中的三角形，代码如下：

```
indices = new Array();
indices.push(0, 3, 1);
indices.push(3, 1, 4);
indices.push(1, 4, 2);
indices.push(4, 2, 5);
```

```
indices.push(3, 6, 4);
indices.push(6, 4, 7);
indices.push(4, 7, 5);
indices.push(7, 5, 8);
```

第三个参数 uvtData 表示定义上面的每个顶点相对于整张图片的比例。如图 4-19 中 9 个顶点的坐标，它们相对于原图片中的位置分别为图 4-20 所示的位置。

对应图 4-20 中的坐标，转换成代码如下：

```
uvtData = new Array();
uvtData.push(0, 0);
uvtData.push(0, 0.5);
uvtData.push(0, 1);
uvtData.push(0.5, 0);
uvtData.push(0.5, 0.5);
uvtData.push(0.5, 1);
uvtData.push(1, 0);
uvtData.push(1, 0.5);
uvtData.push(1, 1);
```

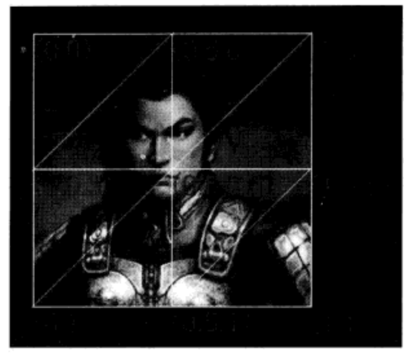


图 4-20 顶点位置示意图

有了上面这些参数的定义，再通过 LSprite 对象的 graphics 属性的 beginBitmapFill 和 drawTriangles 两个函数，就可以绘制多样化的图形了。

如果在 vertices 参数中定义的坐标位置就是原图片中所对应的位置，那么图片是没有什么变化的。下面，改变这些坐标的位置，如代码清单 4-15 所示。

代码清单 4-15

```
<script type="text/javascript">
var loader;
init(50,"mylegend",500,350,main);

function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("face.jpg","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content);
    var backLayer = new LSprite();
    backLayer.x=100;
    addChild(backLayer);

    vertices = new Array();
    vertices.push(0, 0);
    vertices.push(0-50, 120);// 将原坐标的 x 坐标左移 50
    vertices.push(0, 240);
    vertices.push(120, 0);
    vertices.push(120, 120);
```

```
    vertices.push(120, 240);
    vertices.push(240, 0);
    vertices.push(240+50, 120);// 将原坐标的 x 坐标右移 50
    vertices.push(240, 240);

    indices = new Array();
    indices.push(0, 3, 1);
    indices.push(3, 1, 4);
    indices.push(1, 4, 2);
    indices.push(4, 2, 5);
    indices.push(3, 6, 4);
    indices.push(6, 4, 7);
    indices.push(4, 7, 5);
    indices.push(7, 5, 8);

    uvtData = new Array();
    uvtData.push(0, 0);
    uvtData.push(0, 0.5);
    uvtData.push(0, 1);
    uvtData.push(0.5, 0);
    uvtData.push(0.5, 0.5);
    uvtData.push(0.5, 1);
    uvtData.push(1, 0);
    uvtData.push(1, 0.5);
    uvtData.push(1, 1);

    backLayer.graphics.beginBitmapFill(bitmapdata);
    backLayer.graphics.drawTriangles(vertices, indices, uvtData);
}
</script>
```

运行上面代码可以得到如图 4-21 所示的效果。

图 4-21 代码清单 4-15 运行效果

可以看到，在绘制图片过程中，只是改变了一下 drawTriangles 函数的参数 vertices 中对应位置的坐标，就实现了如图 4-21 所示的变形。

在上面的例子中，只是对图片进行了小部分的分解，在实际应用中可以将一张图片分解成几十个甚至几百个小三角形，从而实现更复杂的变形，如图 4-22 和图 4-23 所示的效果。

大家可通过本书源码包中 other 文件夹里的 drawTriangles-01.html 和 drawTriangles-02.html 查看图 4-22 和图 4-23 的实现代码。

图 4-22 drawTriangles 函数实现图片扭曲效果

图 4-23 drawTriangles 函数实现 3D 效果

4.5 文本

LTextField 对象是 lufylegend 库件中专门用于显示文本信息的一个对象。下面的代码只要调用 LTextField 的构造函数就可以显示文本框了。

```
var field = new LTextField ( );
```

4.5.1 文本属性

创建的文本框对象不会自动加入可视化对象列表中，这就意味它不会被显示出来，只有手动调用 addChild( ) 方法添加它才能显示。代码清单 4-16 创建了 LTextField 对象并显示之。

代码清单 4-16

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
   var layer = new LSprite();
   addChild(layer);
   var field = new LTextField();
   field.text = "Hello World!";
   layer.addChild(field);
}
</script>
```

运行效果如图 4-24 所示。

Hello World!

图 4-24 创建 LTextField 对象并显示的效果

LTextField 对象可以改变文本的各种属性，代码清单 4-17 就通过它对文本做了一些修饰。

PDG

代码清单 4-17

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
   var layer = new LSprite();
   addChild(layer);
   var field = new LTextField();
   field.x = 50;
   field.y = 50;
   field.text = "Hello World!";
   field.size = 25;
   field.color = "#333333";
   field.weight = "bolder";
   layer.addChild(field);
}
</script>
```

运行效果如图 4-25 所示。

Hello World!

图 4-25 对文本修饰的效果

代码解析

下面代码新建了一个 LTextField 对象，并设置其坐标为(50,50)。

```
var field = new LTextField();
field.x = 50;
field.y = 50;
```

下面设置文字大小为 25。

```
field.size = 25;
```

下面设置文字的颜色为 #333333。

```
field.color = "#333333";
```

下面设置文字显示为粗体。

```
field.weight = "bolder";
```

4.5.2 输入框

LTextField 还可以将文本变为一个输入框，只需要将文本的 texttype 属性设置为 LTextFieldType.INPUT 即可。具体用法如代码清单 4-18 所示。

代码清单 4-18

```
<script type="text/javascript">
init(50,"mylegend",500,350,main);
function main(){
   var layer = new LSprite();
   addChild(layer);
```

```
    var field = new LTextField();
    field.x = 50;
    field.y = 50;
    field.setType(LTextFieldType.INPUT);
    layer.addChild(field);
}
</script>
```

运行效果如图 4-26 所示。

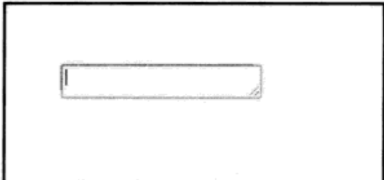


图 4-26 绘制输入框效果

代码解析

可使用 LTextField 对象的 setType 函数，设置 texttype 属性为 LTextFieldType.INPUT，将其变成一个输入框。代码如下所示：

```
field.setType(LTextFieldType.INPUT);
```

4.6 事件

lufylegend 库件里有各种事件，使用 addEventListener 可以为各种事件添加侦听。

4.6.1 鼠标事件

鼠标事件分为鼠标按下（LMouseEvent.MOUSE_DOWN）、鼠标弹起（LMouseEvent.MOUSE_UP）和鼠标移动（LMouseEvent.MOUSE_MOVE)3 个事件。

代码清单 4-19 说明了鼠标事件的用法。

代码清单 4-19

```
<script type="text/javascript">
init(50,"mylegend",300,300,main);
var field;
function main(){
    var layer = new LSprite();
    layer.graphics.drawRect(1,'#cccccc',[0,0,300,300],true,'#cccccc');
    addChild(layer);
    field = new LTextField();
    field.text = "Wait Click!";
    layer.addChild(field);
    layer.addEventListener(LMouseEvent.MOUSE_DOWN,downshow);
    layer.addEventListener(LMouseEvent.MOUSE_UP,upshow);
}
function downshow(event){
    field.text = "Mouse Down!";
}
function upshow(event){
    field.text = "Mouse Up!";
}
</script>
```

运行效果如图 4-27 所示。

代码解析

下面代码表示建立一个文本。

```
field = new LTextField();
```

下面给 layer 对象加载鼠标按下和鼠标弹起事件侦听。

```
layer.addEventListener(LMouseEvent.MOUSE_
DOWN,downshow);
layer.addEventListener(LMouseEvent.MOUSE_
UP,upshow);
```

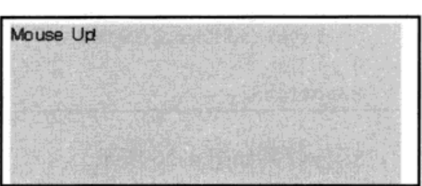


图 4-27 鼠标事件的用法

以下两个函数分别用来侦听鼠标按下和鼠标弹起事件，当鼠标按下的时候 field 文本的显示值为“Mouse Down!”，当鼠标弹起的时候 field 文本的显示值为“Mouse Up!”。

```
function downshow(event){
   field.text = "Mouse Down!";
}
function upshow(event){
   field.text = "Mouse Up!";
}
```

注意 在手机上 MOUSE_DOWN、MOUSE_UP 和 MOUSE_MOVE 三个事件是用 TOUCH_START、TOUCH_END 和 TOUCH_MOVE 来代替的。但是在 lufylegend 库件里不需要有这些区分，无论是进行 PC 开发还是手机开发，都是用 MOUSE_DOWN、MOUSE_UP 和 MOUSE_MOVE 三个事件，因为 lufylegend 库件会根据运行环境自动转换相应的事件。

4.6.2 循环事件

如果想重复执行某段代码，那么就需要用到循环事件的侦听，循环事件就是指按照指定间隔时间不断重复地广播某事件，有的地方解释为以帧频不断触发脚本，本书把它叫作循环事件。只要给某一对象添加此侦听事件，就可以达到循环的目的。在 lufylegend 库件里可使用 LEvent.ENTER_FRAME 来添加侦听循环事件。

具体用法如代码清单 4-20 所示。

代码清单 4-20

```
<script type="text/javascript">
init(50,"mylegend",300,300,main);
var field;
function main(){
   var layer = new LSprite();
   layer.graphics.drawRect(1,'#cccccc',[0,0,300,300],true,'#cccccc');
```

```
    addChild(layer);
    field = new LTextField();
    field.text = "0";
    layer.addChild(field);
    layer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
function onframe(){
    field.text = parseInt(field.text) + 1;
}
</script>
```

运行效果如图 4-28 所示。

将会看到，屏幕上的文本会由 0 开始不断自增。

218

图 4-28　循环事件运行效果

代码解析

下面代码建立了一个文本，并将其显示值设置为 0。

```
field = new LTextField();
field.text = "0";
```

下面给 layer 对象加载循环事件侦听。

```
layer.addEventListener(LEvent.ENTER_FRAME,onframe);
```

下面是循环事件的回调函数，它将当前 field 的显示值转换为整形数据，并加上 1，然后重新复制给 field，这样就实现了 field 的值由 0 开始自增。

```
function onframe(){
    field.text = parseInt(field.text) + 1;
}
```

4.6.3　键盘事件

在 lufylegend 库 件 里 用 LKeyboardEvent.KEY_DOWN、LKeyboardEvent.KEY _UP 和 LKeyboardEvent.KEY_PRESS 来侦听键盘事件。

由于键盘事件需要加载到 window 上，所以加载的时候与前面讲述的方法会稍微有些变化，具体做法看代码清单 4-21。

代码清单　4-21

```
<script type="text/javascript">
init(50,"mylegend",300,300,main);
var field;
function main(){
    var layer = new LSprite();
    layer.graphics.drawRect(1,'#cccccc',
                  [0,0,300,300],true,'#cccccc');
    addChild(layer);
    field = new LTextField();
```

```
    field.text = "Wait Click!";
    layer.addChild(field);
    LEvent.addEventListener(LGlobal.window,
                    LKeyboardEvent.KEY_DOWN,downshow);
    LEvent.addEventListener(LGlobal.window,
                    LKeyboardEvent.KEY_UP,upshow);
}
function downshow(event){
    field.text = event.keyCode + " Down!";
}
function upshow(event){
    field.text = event.keyCode + " Up!";
}
</script>
```

运行效果如图 4-29 所示。

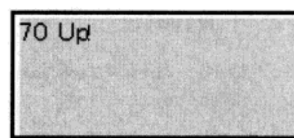


图 4-29 键盘事件运行效果

代码解析

与前面讲述的方法不同的是，这里是使用 LEvent.addEventListener 来加载键盘事件的，其中的 LGlobal.window 就是 window 对象。所以键盘事件是加载到 window 对象上的，这样就能侦听整个浏览器窗口。

```
LEvent.addEventListener(LGlobal.window,
                LKeyboardEvent.KEY_DOWN,downshow);
LEvent.addEventListener(LGlobal.window,
                LKeyboardEvent.KEY_UP,upshow);
```

下面这两个函数分别用来侦听键盘上的某个键按下和弹起事件，event.keyCode 可以得到被按下的键的键值。当键盘上的某个键被按下的时候，field 文本的显示值为“键值 Down!”，当该键弹起的时候 field 文本的显示值为“键值 Up!”。键盘上每个键所对应的键值都是不同的，所以我们玩游戏的时候就可以根据这些键值的不同来判断按下了哪个键，从而做出不同的指令，如向前走、向上跳等。

```
function downshow(event){
    field.text = event.keyCode + " Down!";
}
function upshow(event){
    field.text = event.keyCode + " Up!";
}
```

4.7 按钮

为了简化游戏开发，lufylegend 内置了 LButton 类来添加按钮。其原型如下：

```
LButton(DisplayObject_up,DisplayObject_over)
```

PDG

参数：

- DisplayObject_up：代表按钮默认 up 状态的对象，当鼠标不在按钮上，按钮就处于这个状态。
- DisplayObject_over：当鼠标移动到按钮上时按钮的状态，鼠标离开时按钮又回到 up 状态。

传入按钮的这两个状态对象，可以是 LSprite 对象，也可以是 LBitmap 对象。代码清单 4-22 是使用 LButton 的一个简单应用示例。

代码清单 4-22

```
<script type="text/javascript">
init(50,"mylegend",300,300,main);
var loader,bitmapup,bitmapover,field;
function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadUp);
    loader.load("up.png","bitmapData");
}
function loadUp(event){
   bitmapup = new LBitmap(new LBitmapData(loader.content));
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadOver);
    loader.load("over.png","bitmapData");
}
function loadOver(){
   bitmapover = new LBitmap(new LBitmapData(loader.content));
   var layer = new LSprite();
   addChild(layer);
   field = new LTextField();
   field.text = "Wait Click!";
   layer.addChild(field);
   var testButton = new LButton(bitmapup,bitmapover);
   testButton.y = 50;
   layer.addChild(testButton);
   testButton.addEventListener(LMouseEvent.MOUSE_DOWN,
                              downshow);
}
function downshow(event){
   field.text = "testButton Click!";
}
</script>
```

运行效果如图 4-30 所示。

图 4-30 添加按钮效果

代码解析

下面读取了两张图片，准备接下来当作按钮的两个状态。

```
function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadUp);
```

```
    loader.load("up.png","bitmapData");
}
function loadUp(event){
    bitmapup = new LBitmap(new LBitmapData(loader.content));
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadOver);
    loader.load("over.png","bitmapData");
}
```

下面建了一个 LButton 按钮对象，并将 bitmapup、bitmapover 作为参数传给了这个按钮。

```
var testButton = new LButton(bitmapup,bitmapover);
testButton.y = 50;
layer.addChild(testButton);
```

LButton 继承自 LSprite 类，所以它有 LSprite 类的所有属性和方法，这样就可以给按钮添加鼠标点击事件了。代码如下：

```
testButton.addEventListener(LMouseEvent.MOUSE_DOWN,
                                    downshow);
```

在上面的测试中，当单击按钮的时候，文本就会发生变化。

4.8 动画

动画在游戏中是最常见的，最简单的动画为人的行走。可以说，动画是游戏最基本的组成部分。在 lufylegend 库件中利用 LAnimation 类和循环事件，可以很轻松地实现一组动画的播放。下面举例说明具体做法，这里将以一个人向 4 个方向行走的动画为例。

首先要准备一张图片，这张图片包含了某个人物的几组动作，如图 4-31 所示。

图 4-31 准备的图片

图 4-31 可以分成 4 行 4 列，共 16 个小图片，每个小图片代表人物的一个动作。如果把这些小图片每一行的 4 个小图片顺序播放，那么就会形成一组动画。LAnimation 类实际上就是利用这些小图片不同的坐标位置，并让这些小图片逐个显示以形成动画的。下面是 LAnimation 的构造器：

```
LAnimation(layer,data,list)
```

参数：

- ❑ layer：LSprite 对象。
- ❑ data：LBitmapData 对象。
- ❑ list：一个存储坐标的二维数组。

根据 LAnimation 的构造器可以得知，我们需要准备一个二维的坐标数组，也就是上面所说的 16 个小图片的坐标。这里可以用 LGlobal.divideCoordinate 函数来实现。其原理如下：

```
LGlobal.divideCoordinate(width,height,row,col)
```

参数：

width：宽。

height：高。

row：行数。

col：列数。

LGlobal.divideCoordinate 函数会将传入的宽和高按照行数和列数进行拆分计算，从而得到一个二维数组。如图 4-31 中的图片宽 256px，高 256px，那么进行的拆分的代码如下：

```
LGlobal.divideCoordinate(256,256,4,4)
```

返回值

[[{x:0,y:0},{x:64,y:0},{x:128,y:0},{x:192,y:0}],
[{x:0,y:64},{x:64,y:64},{x:128,y:64},{x:192,y:64}],
[{x:0,y:128},{x:64,y:128},{x:128,y:128},{x:192,y:128}],
[{x:0,y:192},{x:64,y:192},{x:128,y:192},{x:192,y:192}]]

返回的二维数组中每个元素中的坐标对应图 4-31 中每个相应位置的小图片的坐标值。LAnimation 类的具体用法如代码清单 4-23 所示。

代码清单 4-23

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8" />
    <script type="text/javascript"
            src="../../lufylegend-1.5.1.min.js"></script>
</head>
<body>
<div id="mylegend">loading...</div>
<script type="text/javascript">
var loader,anime,layer;
init(200,"mylegend",500,350,main);
function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("chara.png","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content,0,0,64,64);
    var list = LGlobal.divideCoordinate(256,256,4,4);
    // 加入层 LSprite
```

```
    layer = new LSprite();
    addChild(layer);
    anime = new LAnimation(layer,bitmapdata,list);
    layer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
function onframe(){
    anime.onframe();
}
</script>
</body>
</html>
```

运行效果如图 4-32 所示。

可以看到画面上的人物已经动起来了，实际上就是将图 4-31 中的第一行小图片逐个循环播放起来。

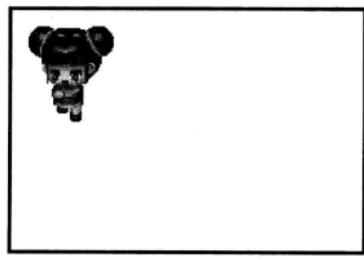


图 4-32 使用 LAnimation 类的效果

代码解析

下面的代码将读取完的 Image 保存到 LBitmapData 里，显示范围为 (0,0,64,64)。

```
var bitmapdata = new LBitmapData(loader.content,
0,0,64,64);
```

下面建了一个保存有坐标位置的二维数组。

```
var list = LGlobal.divideCoordinate(256,256,4,4);
```

下面建了一个 LSprite 层，并将其加载到 Canvas 上。

```
layer = new LSprite();
addChild(layer);
```

下面建了一个 LAnimation 类，并将上面准备好的 layer、bitmapdata、list 作为参数传入 LAnimation 对象中。

```
anime = new LAnimation(layer,bitmapdata,list);
```

下面给 layer 层添加循环事件侦听。

```
layer.addEventListener(LEvent.ENTER_FRAME,onframe);
```

循环事件的回调函数调用 LAnimation 类的 onframe 函数，来实现动画的顺序循环播放。

```
function onframe(){
    anime.onframe();
}
```

LAnimation 类的 onframe 函数的功能是将所播放图片的列号加 1，如果循环 onframe 函数，就变成了动画。但是目前只是实现了第一行图片的循环播放，如果要实现所有图片的循环播放，则需要用到 LAnimation 类的 setAction 函数。其函数原型如下所示：

```
setAction(rowIndex,colIndex)
```

参数：

rowIndex：数组行号。

colIndex：数组列号。

使用 setAction 函数可以改变 LAnimation 类所播放图片的行号和列号，如果只需要改变播放的行号，那么第二个参数可以省略。代码清单 4-24 说明了 setAction 函数的详细用法。

代码清单 4-24

```
<script type="text/javascript">
var loader,anime,layer;
init(200,"mylegend",500,350,main);
function main(){
    loader = new LLoader();
    loader.addEventListener(LEvent.COMPLETE,loadBitmapdata);
    loader.load("chara.png","bitmapData");
}
function loadBitmapdata(event){
    var bitmapdata = new LBitmapData(loader.content,0,0,64,64);
    var list = LGlobal.divideCoordinate(256,256,4,4);
    //加入层 LSprite
    layer = new LSprite();
    addChild(layer);
    anime = new LAnimation(layer,bitmapdata,list);
    layer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
function onframe(){
    var action = anime.getAction();
    switch(action[0]){
        case 0:
            layer.y += 5;
            if(layer.y >= 200){
                anime.setAction(2);
            }
            break;
        case 1:
            layer.x -= 5;
            if(layer.x <= 0){
                anime.setAction(0);
            }
            break;
        case 2:
            layer.x += 5;
            if(layer.x >= 200){
                anime.setAction(3);
            }
            break;
        case 3:
```

```
                layer.y -= 5;
                if(layer.y <= 0){
                    anime.setAction(1);
                }
                break;
        }
        anime.onframe();
    }
</script>
```

运行效果如图 4-33 所示。

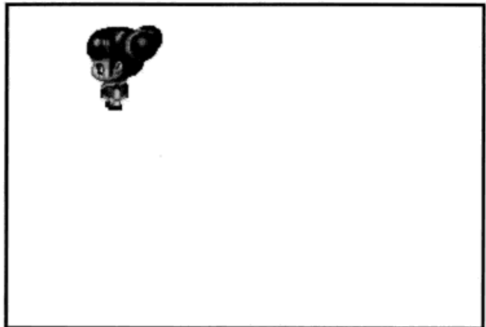


图 4-33 使用 LAnimation 类的循环播放效果

可以看到画面上的人物已经开始绕着 4 个方向走动起来了。

代码解析

使用 LAnimation 类的 getAction 函数取得 anime 对象当前所播放动画的行号和列号，其返回值为数组类型 [行号 , 列号]。代码如下 :

```
var action = anime.getAction();
```

下面代码利用 switch 对当前所播放动画的行号进行了区别处理，[0,1,2,3] 这 4 个行号在图 4-31 中分别代表下、左、右、上 4 个方向，然后在 4 个方向上改变坐标值进行相应的移动，并且根据所移动到达的位置来改变移动的方向。

```
switch(action[0]){
        case 0:
            layer.y += 5;
            if(layer.y >= 200){
                anime.setAction(2);
            }
            break;
        case 1:
            layer.x -= 5;
            if(layer.x <= 0){
                anime.setAction(0);
            }
            break;
        case 2:
            layer.x += 5;
            if(layer.x >= 200){
                anime.setAction(3);
            }
            break;
        case 3:
            layer.y -= 5;
            if(layer.y <= 0){
```

```
                anime.setAction(1);
            }
            break;
    }
```

循环事件的回调函数调用 LAnimation 类的 onframe 函数，来实现动画的顺序循环播放。

```
anime.onframe();
```

4.9　小结

本章带领大家认识了开源库件 lufylegend 的工作原理，并且详细介绍了该库件的几个核心类的功能和使用方法。对照一下第 2 章和第 3 章的内容，就会发现 lufylegend 的便利之处。其实 lufylegend 的优势主要在游戏开发上，从下一章开始会带大家一起学习游戏开发的相关知识。

PDG

第三部分

开发实战篇

- 第 5 章　从简单做起—“石头剪子布”游戏
- 第 6 章　开发“俄罗斯方块”游戏
- 第 7 章　开发“是男人就下一百层”游戏
- 第 8 章　开发射击类游戏
- 第 9 章　开发物理游戏
- 第 10 章　开发网络游戏

第5章　从简单做起——“石头剪子布”游戏

从这一章开始，将从实战出发，以示例为基础，讲解如何使用 lufylegend 库件进行 HTML 5 的游戏开发。由于这是本书的第一个游戏，所以选择比较简单的游戏。本章将介绍如何做一款猜拳游戏——石头剪子布。

5.1　游戏分析

首先来分析一下一个猜拳游戏需要用到哪些东西。

如图 5-1 所示是预计开发的游戏画面。

图 5-1　猜拳游戏界面

需要用到的要素大约有下面几种：

- ❑ 图片描画

从图 5-1 中可以得知，需要将石头、剪刀和布这 3 张图片绘制到画面上。

- ❑ 图形绘制

图 5-1 中的背景以及边框，都需要由图形来描画。

- ❑ 文字绘制

图 5-1 中统计次数等文字的显示，则用到了文字绘制方式。

- ❑ 鼠标的点击

当玩家出拳的时候，通过点击 3 个不同的按钮来选择出拳。

- ❑ 电脑 AI

所需的电脑 AI 很简单，只要让电脑随机出拳即可。

- ❑ 条件分支与判断

胜负的判定需要用条件分支来显示相应的结果。

5.2 必要的 JavaScript 知识

根据 5.1 节的分析，本章会用到以下几个 JavaScript 的知识点。

5.2.1 随机数

JavaScript 中生成随机数用到的是 Math.random() 函数，它可以随机生成 0 到 1 之间的一个小数。如果需要随机生成一个 5 到 10 之间的整数，可以使用如下方法：

```
Math.floor(5 + 5*Math.random());
```

5.2.2 条件分支

JavaScript 中条件分支有两种方式：if...else if...else... 和 switch。

if...else if...else... 语句的用法如下所示：

```
if(条件1){
    条件1成立时执行代码
}else if(条件2){
    条件2成立时执行代码
}else{
    条件1和条件2都不成立时执行代码
}
```

switch 语句的用法如下所示：

```
switch(变量){
   case 值1:
            执行代码块1
            break;
   case 值2:
            执行代码块2
            break;
   default:
            当变量既不是值1也不是值2的时候执行的代码块
}
```

5.3 分层实现

现在开始一步步地来实现这个简单的小游戏。

首先，将游戏做一下层次划分。从图 5-1 来看，可以将整个游戏界面作为一个层，然后将选择出拳部分和结果显示部分分离成为另外两个层。这样整个游戏界面大致可以分为 3 个层。先来将这 3 个部分分别显示到画面上，如代码清单 5-1 所示。

代码清单 5-1

```
init(50,"mylegend",800,400,main);
var backLayer,
```

PDG

```
    resultLayer,
    clickLayer;
    function main(){
       backLayer = new LSprite();
       addChild(backLayer);
       gameInit();
    }
    function gameInit(){
       //添加游戏界面背景
       backLayer.graphics.drawRect(10,'#008800',[0,0,LGlobal.width,LGlobal.height],
true,'#000000');
       //结果显示层初始化
       initResultLayer();
       //操作层初始化
       initClickLayer();
    }
    function initResultLayer(){
       resultLayer = new LSprite();
       resultLayer.graphics.drawRect(4,'#ff8800',[0,0,150,110],true,'#ffffff');
       resultLayer.x = 10;
       resultLayer.y = 100;
       backLayer.addChild(resultLayer);
    }
    function initClickLayer(){
       clickLayer = new LSprite();
       clickLayer.graphics.drawRect(4,'#ff8800',[0,0,300,110],true,'#ffffff');
       clickLayer.x = 250;
       clickLayer.y = 275;
       backLayer.addChild(clickLayer);
    }
```

运行效果如图 5-2 所示。

图 5-2 中的黑色背景层为 backLayer 层，resultLayer 层和 clickLayer 层如图 5-3 所示。

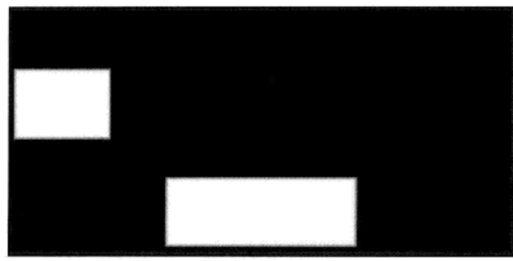


图 5-2　代码清单 5-1 的运行效果

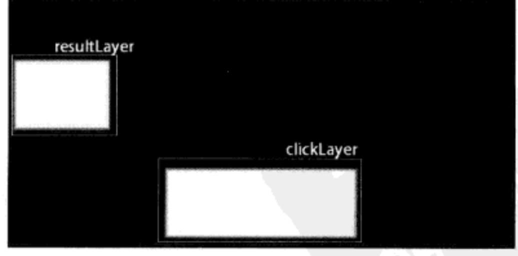


图 5-3　图 5-2 的说明图

代码解析

下面的代码表示新建 3 个层变量，分别代表 3 个显示层。

```
var backLayer,
```

```
resultLayer,
clickLayer;
```

初始化层 backLayer，将其作为背景层显示出来。

```
backLayer = new LSprite();
addChild(backLayer);
```

在 backLayer 背景层上绘制一个黑色矩形，其中的 LGlobal.width 和 LGlobal.height 是 lufylegend 库件中的两个变量，而 LGlobal.width 的值则是 Canvas 对象的宽，LGlobal.height 的值是 Canvas 对象的高。

```
//添加游戏界面背景
   backLayer.graphics.drawRect(10,'#008800',
           [0,0,LGlobal.width,LGlobal.height],true,'#000000');
```

初始化 resultLayer 层，将其作为结果表示层，并在层上绘制一个白色矩形，当作结果表示层的背景色。

```
function initResultLayer(){
   resultLayer = new LSprite();
   resultLayer.graphics.drawRect(4,'#ff8800',
                               [0,0,150,110],true,'#ffffff');
   resultLayer.x = 10;
   resultLayer.y = 100;
   backLayer.addChild(resultLayer);
}
```

初始化 clickLayer 层，并将其作为控制点击层，同时在层上绘制一个白色矩形，当作控制点击层的背景色。

```
function initClickLayer(){
   clickLayer = new LSprite();
   clickLayer.graphics.drawRect(4,'#ff8800',[0,0,300,110],true,'#ffffff');
   clickLayer.x = 250;
   clickLayer.y = 275;
   backLayer.addChild(clickLayer);
}
```

5.4　各个层的基本功能

5.3 节已经简单地划分了层次，并将 3 个层次分别显示到了游戏界面上。本节将逐个实现各个层的功能。

5.4.1　基本画面显示

为了将己方出拳和电脑出拳的图片清楚地显示到画面上，需要将下面几张准备好的图片（如图 5-4 ～图 5-7 所示）读到游戏里面，然后按照需求显示到画面上。

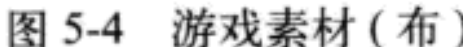


图 5-4 游戏素材(布)

图 5-5 游戏素材(剪刀)

图 5-6 游戏素材(石头)

石头·剪子·布

图 5-7 游戏素材(游戏标题)

关于图片的读取方法，在第 4 章已经讲过。本次使用 lufylegend 库件中另一种比较方便的图片读取方式，即利用静态类 LLoadManage 的 load 函数来一次性读取多张图片。此函数的原函数如下：

```
LLoadManage.load ($list,$onupdate,$oncomplete);
```

参数：

- $list：要读取的图片数组。
- $onupdate：每读取数组中一张图片后调用的函数，可以为空。
- $oncomplete：读取完数组中所有图片后调用的函数。

需要注意的是参数 $list，这个图片数组的格式必须是 lufylegend 库件所规定的格式，其具体用法见代码清单 5-2，此清单修改了代码清单 5-1 中变量设定部分和 main 函数中的代码。

代码清单 5-2

```
init(50,"mylegend",800,400,main);
var loadingLayer,
backLayer,
resultLayer,
clickLayer;
var imglist = {};
var imgData = new Array(
            {name:"title",path:"../images/title.png"},
            {name:"shitou",path:"../images/shitou.png"},
            {name:"jiandao",path:"../images/jiandao.png"},
            {name:"bu",path:"../images/bu.png"});
function main(){
    backLayer = new LSprite();
    addChild(backLayer);
    loadingLayer = new LoadingSample3();
    backLayer.addChild(loadingLayer);
```

```
    LLoadManage.load(
            imgData,
            function(progress){
                    loadingLayer.setProgress(progress);
            },
            function(result){
                    imglist = result;
                    backLayer.removeChild(loadingLayer);
                    loadingLayer = null;
                    gameInit();
            }
    );
}
```

图片读取过程中的显示效果如图 5-8 所示。

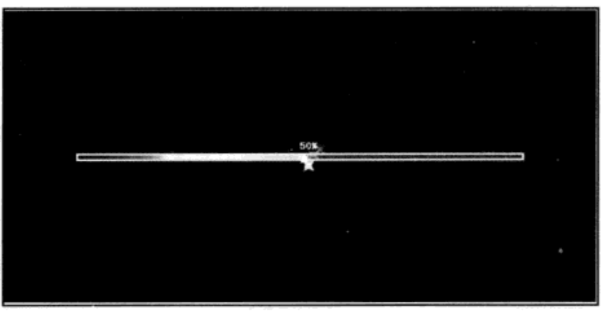


图 5-8　图片读取过程中的显示效果

代码解析

```
   var imglist = {};
var imgData = new Array(
          {name:"title",path:"../images/title.png"},
          {name:"shitou",path:"../images/shitou.png"},
          {name:"jiandao",path:"../images/jiandao.png"},
          {name:"bu",path:"../images/bu.png"});
```

上面代码中的变量 imgData 中含有需要读取的图片的路径，变量 imglist 是用来保存读取后的图片数据的数组，保存方式是以 key=>value 的形式，其中的 key 就是 imgData 中对应的 name 值。

```
loadingLayer = new LoadingSample3();
backLayer.addChild(loadingLayer);
```

上面代码的作用是加载进度条，图 5-8 中之所以会出现进度条，就是因为加载了 LoadingSample3 对象。在 lufylegend 库件中一共有 3 个进度条显示对象，分别是 LoadingSample1、LoadingSample2 和 LoadingSample3，这 3 个不同的对象都继承自 LSprite，分别可以显示 3 种不同的进度条样式。本次使用的是第三种进度条对象 LoadingSample3，另外两种效果会在后面

的章节中出现。

```
LLoadManage.load(
        imgData,
        function(progress){
                loadingLayer.setProgress(progress);
        },
        function(result){
                imglist = result;
                backLayer.removeChild(loadingLayer);
                loadingLayer = null;
                gameInit();
        }
  );
```

上面代码是利用静态类 LLoadManage 的 load 函数来读取数组中的图片。第一个参数 imgData 就是要读取的图片数组；第二个参数是 function 类型，是在每读取一张图片后调用的函数，其中的 progress 指的是读取完的图片个数占整个图片数组长度的比例，将这个比例值通过 setProgress 传给 LoadingSample3 对象，就实现了动态的进度条；第三个参数也是 function 类型，是在读取完所有图片后调用的函数，该函数被调用时，会执行该函数内的所有代码。上面代码的功能是，首先将读取完图片后的结果集赋值给变量 imglist，然后移除画面上的进度条对象，最后调用 gameInit，进行游戏下一步的操作。

通过代码清单 5-2 已经将需要的图片都读取到了游戏里，接下来则要修改 gameInit 函数，添加相应的变量，如代码清单 5-3 所示。

代码清单 5-3

```
init(50,"mylegend",800,400,main);
var loadingLayer,
backLayer,
resultLayer,
clickLayer,
selfBitmap,
enemyBitmap;
var imglist = {};
var imgData = new Array(
         {name:"title",path:"../images/title.png"},
         {name:"shitou",path:"../images/shitou.png"},
         {name:"jiandao",path:"../images/jiandao.png"},
         {name:"bu",path:"../images/bu.png"});
var showList = new Array();
function main(){
   backLayer = new LSprite();
   addChild(backLayer);
   loadingLayer = new LoadingSample3();
   backLayer.addChild(loadingLayer);
   LLoadManage.load(
         imgData,
```

```
                function(progress){
                        loadingLayer.setProgress(progress);
                },
                function(result){
                        imglist = result;
                        backLayer.removeChild(loadingLayer);
                        loadingLayer = null;
                        gameInit();
                }
        );
    }
    function gameInit(){
        showList.push(new LBitmapData(imglist["shitou"]));
        showList.push(new LBitmapData(imglist["jiandao"]));
        showList.push(new LBitmapData(imglist["bu"]));
        //添加游戏界面背景
        backLayer.graphics.drawRect(10,'#008800',[0,0,LGlobal.width,LGlobal.height],
true,'#000000');
        //显示游戏标题
        var titleBitmap = new LBitmap(new LBitmapData(imglist["title"]));
        titleBitmap.x = (LGlobal.width - titleBitmap.width)/2;
        titleBitmap.y = 10;
        backLayer.addChild(titleBitmap);
        //玩家方出拳图片
        selfBitmap = new LBitmap(showList[0]);
        selfBitmap.x = 400 - selfBitmap.width - 50;
        selfBitmap.y = 130;
        backLayer.addChild(selfBitmap);
        //电脑方出拳图片
        enemyBitmap = new LBitmap(showList[0]);
        enemyBitmap.x = 400 + 50;
        enemyBitmap.y = 130;
        backLayer.addChild(enemyBitmap);
        //玩家、电脑名称设定
        var nameText;
        nameText = new LTextField();
        nameText.text = "玩家";
        nameText.weight = "bolder";
        nameText.color = "#ffffff";
        nameText.size = 24;
        nameText.x = selfBitmap.x +
                (selfBitmap.width - nameText.getWidth())/2;
        nameText.y = 95;
        backLayer.addChild(nameText);
        nameText = new LTextField();
        nameText.text = "电脑";
        nameText.weight = "bolder";
        nameText.color = "#ffffff";
        nameText.size = 24;
        nameText.x = enemyBitmap.x +
                (enemyBitmap.width - nameText.getWidth())/2;
```

```
        nameText.y = 95;
        backLayer.addChild(nameText);
        //结果显示层初始化
        initResultLayer();
        //操作层初始化
        initClickLayer();
    }
    function initResultLayer(){
        resultLayer = new LSprite();
        resultLayer.graphics.drawRect(4,'#ff8800',
                                [0,0,150,110],true,'#ffffff');
        resultLayer.x = 10;
        resultLayer.y = 100;
        backLayer.addChild(resultLayer);
    }
    function initClickLayer(){
        clickLayer = new LSprite();
        clickLayer.graphics.drawRect(4,'#ff8800',
                                [0,0,300,110],true,'#ffffff');
        clickLayer.x = 250;
        clickLayer.y = 275;
        backLayer.addChild(clickLayer);
    }
```

运行效果如图 5-9 所示。

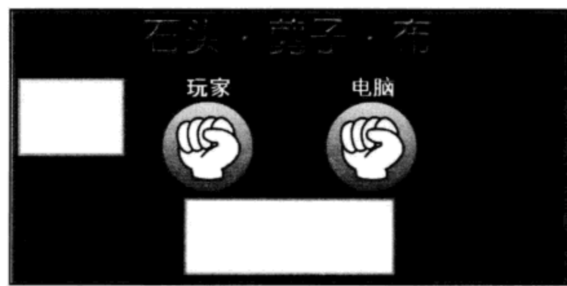


图 5-9 代码清单 5-3 的运行效果

代码解析

```
var loadingLayer,
backLayer,
resultLayer,
clickLayer,
selfBitmap,
enemyBitmap;
```

从上面的代码中可以看到，在变量部分增加了 selfBitmap 和 enemyBitmap 变量，用于显示己方和电脑的出拳结果。

```
showList.push(new LBitmapData(imglist["shitou"]));
```

```
showList.push(new LBitmapData(imglist["jiandao"]));
showList.push(new LBitmapData(imglist["bu"]));
```

上面代码是为了避免反复新建 LBitmapData 对象，所以将新建的 3 个 LBitmapData 对象添加到数组 showList 中。imglist 是利用 LLoadManage 读取图片后返回的结果集。所以这 3 个 LBitmapData 对象中分别保存了石头、剪刀、布 3 张图片的 Image 对象。

```
//显示游戏标题
var titleBitmap = new LBitmap(new LBitmapData(imglist["title"]));
titleBitmap.x = (LGlobal.width - titleBitmap.width)/2;
titleBitmap.y = 10;
backLayer.addChild(titleBitmap);
```

上面的代码是将游戏标题图片添加到 backLayer 层上，并设定图片显示的位置。

```
//玩家方出拳图片
selfBitmap = new LBitmap(showList[0]);
selfBitmap.x = 400 - selfBitmap.width - 50;
selfBitmap.y = 130;
backLayer.addChild(selfBitmap);
//电脑方出拳图片
enemyBitmap = new LBitmap(showList[0]);
enemyBitmap.x = 400 + 50;
enemyBitmap.y = 130;
backLayer.addChild(enemyBitmap);
```

上面的代码是添加玩家和电脑出拳的图片，并设定图片显示的位置。

```
//玩家、电脑名称设定
var nameText;
nameText = new LTextField();
nameText.text = "玩家";
nameText.weight = "bolder";
nameText.color = "#ffffff";
nameText.size = 24;
nameText.x = selfBitmap.x +
          (selfBitmap.width - nameText.getWidth())/2;
nameText.y = 95;
backLayer.addChild(nameText);
nameText = new LTextField();
nameText.text = "电脑";
nameText.weight = "bolder";
nameText.color = "#ffffff";
nameText.size = 24;
nameText.x = enemyBitmap.x +
          (enemyBitmap.width - nameText.getWidth())/2;
nameText.y = 95;
backLayer.addChild(nameText);
```

上面的代码是利用 LTextField 对象来显示玩家和电脑的名称，其中用到了 LTextField 的 getWidth 函数，它的作用是返回 LTextField 对象的文字列的宽度。

5.4.2 结果层的显示

观察图 5-1 可以看到，在结果层中记录着胜利、失败和平局等各种结果的次数。代码清单 5-4 中的变量部分则为相应的结果定义了变量。

代码清单 5-4

```
var loadingLayer,
backLayer,
resultLayer,
clickLayer,
selfBitmap,
enemyBitmap,
selfTextAll,
selfTextWin,
selfTextLoss,
selfTextDraw,
win = 0,
loss = 0,
draw = 0;
```

其中，变量 win、loss、draw 分别代表胜利次数、失败次数和平局次数；变量 selfTextAll、selfTextWin、selfTextLoss 和 selfTextDraw 是 LTextField 对象，用来显示各种结果值。具体看代码清单 5-5。

代码清单 5-5

```
function initResultLayer(){
    resultLayer = new LSprite();
    resultLayer.graphics.drawRect(4,'#ff8800',[0,0,150,110],true,'#ffffff');
    resultLayer.x = 10;
    resultLayer.y = 100;
    backLayer.addChild(resultLayer);
    selfTextAll = new LTextField();
    selfTextAll.text = " 猜拳次数 : 0";
    selfTextAll.weight = "bolder";
    selfTextAll.x = 10;
    selfTextAll.y = 20;
    resultLayer.addChild(selfTextAll);
    selfTextWin = new LTextField();
    selfTextWin.text = " 胜利次数 : 0";
    selfTextWin.weight = "bolder";
    selfTextWin.x = 10;
    selfTextWin.y = 40;
    resultLayer.addChild(selfTextWin);
    selfTextLoss = new LTextField();
    selfTextLoss.text = " 失败次数 : 0";
    selfTextLoss.weight = "bolder";
    selfTextLoss.x = 10;
    selfTextLoss.y = 60;
```

```
        resultLayer.addChild(selfTextLoss);
        selfTextDraw = new LTextField();
        selfTextDraw.text = "平局次数 : 0";
        selfTextDraw.weight = "bolder";
        selfTextDraw.x = 10;
        selfTextDraw.y = 80;
        resultLayer.addChild(selfTextDraw);
    }
```

上面的代码中，实例化了 selfTextAll、selfTextWin、selfTextLoss 和 selfTextDraw 四个对象，并显示到结果层上。初始状态下各结果的次数都是 0，每次出拳后，这些结果值会重新进行计算后再显示出来。计算方法在 5.5 节中详细说明。

代码清单 5-5 的运行效果如图 5-10 所示。

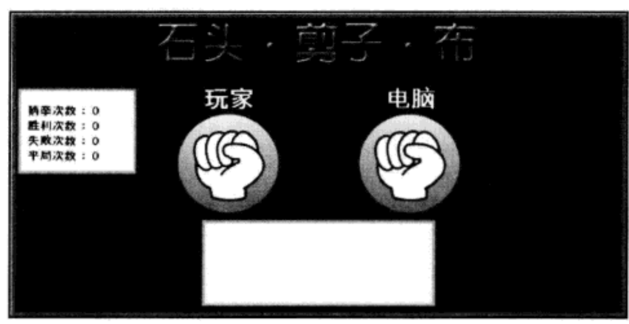


图 5-10　代码 5-5 运行效果

5.4.3　控制层的显示

控制层中包含了石头、剪刀、布 3 个按钮。代码清单 5-6 是为控制层添加这 3 个按钮的过程。

代码清单　5-6

```
function initClickLayer(){
    clickLayer = new LSprite();
    clickLayer.graphics.drawRect(4,'#ff8800',[0,0,300,110],true,'#ffffff');
    var msgText = new LTextField();
    msgText.text = "请出拳：";
    msgText.weight = "bolder";
    msgText.x = 10;
    msgText.y = 10;
    clickLayer.addChild(msgText);
    var btnShitou = getButton("shitou");
    btnShitou.x = 30;
    btnShitou.y = 35;
    clickLayer.addChild(btnShitou);
```

```
    var btnJiandao = getButton("jiandao");
    btnJiandao.x = 115;
    btnJiandao.y = 35;
    clickLayer.addChild(btnJiandao);
    var btnBu = getButton("bu");
    btnBu.x = 200;
    btnBu.y = 35;
    clickLayer.addChild(btnBu);
    clickLayer.x = 250;
    clickLayer.y = 275;
    backLayer.addChild(clickLayer);
}
function getButton(value){
    var btnUp = new LBitmap(new LBitmapData(imglist[value]));
    btnUp.scaleX = 0.5;
    btnUp.scaleY = 0.5;
    var btnOver = new LBitmap(new LBitmapData(imglist[value]));
    btnOver.scaleX = 0.5;
    btnOver.scaleY = 0.5;
    btnOver.x = 2;
    btnOver.y = 2;
    var btn = new LButton(btnUp,btnOver);
    btn.name = value;
    return btn;
}
```

运行效果如图 5-11 所示。

图 5-11 代码清单 5-6 的运行效果

代码解析

```
function getButton(value){
    var btnUp = new LBitmap(new LBitmapData(imglist[value]));
    btnUp.scaleX = 0.5;
    btnUp.scaleY = 0.5;
    var btnOver = new LBitmap(new LBitmapData(imglist[value]));
    btnOver.scaleX = 0.5;
```

```
    btnOver.scaleY = 0.5;
    btnOver.x = 2;
    btnOver.y = 2;
    var btn = new LButton(btnUp,btnOver);
    btn.name = value;
    return btn;
}
```

上面代码是建立一个 getButton 函数，此函数根据传入的参数来建立一个按钮，并且将传入的参数设置为按钮的名称及样式。

```
var btnShitou = getButton("shitou");
btnShitou.x = 30;
btnShitou.y = 35;
clickLayer.addChild(btnShitou);
var btnJiandao = getButton("jiandao");
btnJiandao.x = 115;
btnJiandao.y = 35;
clickLayer.addChild(btnJiandao);
var btnBu = getButton("bu");
btnBu.x = 200;
btnBu.y = 35;
clickLayer.addChild(btnBu);
clickLayer.x = 250;
clickLayer.y = 275;
backLayer.addChild(clickLayer);
```

上面的代码是调用 getButton 函数，通过传入 shitou、jiandao、bu 三个参数来建立石头、剪刀、布 3 个按钮。

5.5　出拳

说到出拳，就是通过单击石头、剪刀、布 3 个按钮来决定自己的出拳结果，对方的出拳则使用 JavaScript 的随机函数来随机进行。5.4 节已经为游戏添加了石头、剪子、布 3 个按钮，代码清单 5-7 则给这 3 个按钮添加鼠标点击事件。

代码清单　5-7

```
function initClickLayer(){
    clickLayer = new LSprite();
    clickLayer.graphics.drawRect(4,'#ff8800',[0,0,300,110],true,'#ffffff');
    var msgText = new LTextField();
    msgText.text = "请出拳：";
    msgText.weight = "bolder";
    msgText.x = 10;
    msgText.y = 10;
    clickLayer.addChild(msgText);
    var btnShitou = getButton("shitou");
```

```
    btnShitou.x = 30;
    btnShitou.y = 35;
    clickLayer.addChild(btnShitou);
    btnShitou.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    var btnJiandao = getButton("jiandao");
    btnJiandao.x = 115;
    btnJiandao.y = 35;
    clickLayer.addChild(btnJiandao);
    btnJiandao.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    var btnBu = getButton("bu");
    btnBu.x = 200;
    btnBu.y = 35;
    clickLayer.addChild(btnBu);
    btnBu.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    clickLayer.x = 250;
    clickLayer.y = 275;
    backLayer.addChild(clickLayer);
}
function onclick(event,display){
    var selfValue,enemyValue;
    if(display.name == "shitou"){
            selfValue = 0;
    }else if(display.name == "jiandao"){
            selfValue = 1;
    }else if(display.name == "bu"){
            selfValue = 2;
    }
    enemyValue = Math.floor(Math.random()*3);
    selfBitmap.bitmapData = showList[selfValue];
    enemyBitmap.bitmapData = showList[enemyValue];
}
```

运行上面的代码，出现的画面和图 5-11 是一样的，但是增加了点击效果。

代码解析

```
var btnShitou = getButton("shitou");
btnShitou.x = 30;
btnShitou.y = 35;
clickLayer.addChild(btnShitou);
btnShitou.addEventListener(LMouseEvent.MOUSE_UP,onclick);
var btnJiandao = getButton("jiandao");
btnJiandao.x = 115;
btnJiandao.y = 35;
clickLayer.addChild(btnJiandao);
btnJiandao.addEventListener(LMouseEvent.MOUSE_UP,onclick);
var btnBu = getButton("bu");
btnBu.x = 200;
btnBu.y = 35;
clickLayer.addChild(btnBu);
btnBu.addEventListener(LMouseEvent.MOUSE_UP,onclick);
```

```
clickLayer.x = 250;
clickLayer.y = 275;
backLayer.addChild(clickLayer);
```

上面的代码是在建立了 3 个按钮之后，分别给这 3 个按钮加上 MOUSE_UP 事件。当鼠标单击按钮并弹起的时候，就会调用 onclick 函数。

```
function onclick(event,display){
   var selfValue,enemyValue;
   if(display.name == "shitou"){
           selfValue = 0;
   }else if(display.name == "jiandao"){
           selfValue = 1;
   }else if(display.name == "bu"){
           selfValue = 2;
   }
   enemyValue = Math.floor(Math.random()*3);
   selfBitmap.bitmapData = showList[selfValue];
   enemyBitmap.bitmapData = showList[enemyValue];
}
```

上面代码新建了 selfValue、enemyValue 两个变量，selfValue 用来表示玩家的出拳结果，enemyValue 用来表示电脑的出拳结果。selfValue 的值由所点击的按钮来决定，enemyValue 的值通过随机函数 Math.random() 来决定。最后将 selfBitmap 和 enemyBitmap 中的 LBitmapData 值改为相应的数据。因为 showList 中保存的图片依次为石头、剪刀和布 3 张图片，对应在 showList 数组中的索引值则分别是 0、1、2。

5.6　结果判定

为了方便判定胜负结果，首先来准备一个结果判定数组。代码如下：

```
var checkList = [
[0,1,-1],
[-1,0,1],
[1,-1,0]
];
```

	石头	剪刀	布
石头	平	胜	负
剪刀	负	平	胜
布	胜	负	平

图 5-12　结果判定表

上面的结果判定数组，分别对应着图 5-12 中的值，其中行对应玩家出拳，列对应电脑出拳。

代码清单 5-8 是胜负判断的实现。

代码清单　5-8

```
function onclick(event,display){
    var selfValue,enemyValue;
    if(display.name == "shitou"){
```

```
            selfValue = 0;
    }else if(display.name == "jiandao"){
            selfValue = 1;
    }else if(display.name == "bu"){
            selfValue = 2;
    }
    enemyValue = Math.floor(Math.random()*3);
    selfBitmap.bitmapData = showList[selfValue];
    enemyBitmap.bitmapData = showList[enemyValue];
    var result = checkList[selfValue][enemyValue];
    if(result == -1){
            loss += 1;
    }else if(result == 1){
            win += 1;
    }else{
            draw += 1;
    }
    selfTextWin.text = "胜利次数 : " + win;
    selfTextLoss.text = "失败次数 : " + loss;
    selfTextDraw.text = "平局次数 : " +  draw;
    selfTextAll.text = "猜拳次数 : " + (win + loss + draw);
}
```

运行效果如图 5-13 所示。

图 5-13 代码清单 5-8 的运行效果

代码解析

```
var result = checkList[selfValue][enemyValue];
```

上面的代码通过 selfValue 和 enemyValue 来得到胜负的结果，selfValue 对应着图 5-12 结果判定表中的行，enemyValue 对应着图 5-12 结果判定表中的列，这样就能得到表中对应的胜负值了。

```
if(result == -1){
   loss += 1;
}else if(result == 1){
   win += 1;
```

```
}else{
    draw += 1;
}
```

上面的代码是根据胜负结果，来分别增加胜利、失败和平局的次数。

```
selfTextWin.text = "胜利次数 : " + win;
selfTextLoss.text = "失败次数 : " + loss;
selfTextDraw.text = "平局次数 : " +  draw;
selfTextAll.text = "猜拳次数 : " + (win + loss + draw);
```

上面代码用于当胜负结果决定后，将胜利、失败、平局和猜拳次数显示到结果层上。猜拳次数等于胜利、失败和平局次数的总和。

这样，这个小游戏的开发就完成了。代码清单 5-9 是本游戏的完整代码。

代码清单　5-9

```
init(50,"mylegend",800,400,main);
var loadingLayer,
backLayer,
resultLayer,
clickLayer,
selfBitmap,
enemyBitmap,
selfTextAll,
selfTextWin,
selfTextLoss,
selfTextDraw,
win = 0,
loss = 0,
draw = 0;
var imglist = {};
var imgData = new Array(
            {name:"title",path:"../images/title.png"},
            {name:"shitou",path:"../images/shitou.png"},
            {name:"jiandao",path:"../images/jiandao.png"},
            {name:"bu",path:"../images/bu.png"}
            );
var checkList = [
[0,1,-1],
[-1,0,1],
[1,-1,0]
];
var showList = new Array();
function main(){
    LGlobal.setDebug(true);
    backLayer = new LSprite();
    addChild(backLayer);
    loadingLayer = new LoadingSample3();
    backLayer.addChild(loadingLayer);
```

```
        LLoadManage.load(
                imgData,
                function(progress){
                    loadingLayer.setProgress(progress);
                },
                function(result){
                    imglist = result;
                    backLayer.removeChild(loadingLayer);
                    loadingLayer = null;
                    gameInit();
                }
        );
    }
    function gameInit(){
        showList.push(new LBitmapData(imglist["shitou"]));
        showList.push(new LBitmapData(imglist["jiandao"]));
        showList.push(new LBitmapData(imglist["bu"]));
        //添加游戏界面背景
        backLayer.graphics.drawRect(10,'#008800',[0,0,LGlobal.width,LGlobal.height],
true,'#000000');

        //显示游戏标题
        var titleBitmap = new LBitmap(new LBitmapData(imglist["title"]));
        titleBitmap.x = (LGlobal.width - titleBitmap.width)/2;
        titleBitmap.y = 10;
        backLayer.addChild(titleBitmap);
        //玩家方出拳图片
        selfBitmap = new LBitmap(showList[0]);
        selfBitmap.x = 400 - selfBitmap.width - 50;
        selfBitmap.y = 130;
        backLayer.addChild(selfBitmap);
        //电脑方出拳图片
        enemyBitmap = new LBitmap(showList[0]);
        enemyBitmap.x = 400 + 50;
        enemyBitmap.y = 130;
        backLayer.addChild(enemyBitmap);
        //玩家、电脑名称设定
        var nameText;
        nameText = new LTextField();
        nameText.text = "玩家";
        nameText.weight = "bolder";
        nameText.color = "#ffffff";
        nameText.size = 24;
        nameText.x = selfBitmap.x + (selfBitmap.width - nameText.getWidth())/2;
        nameText.y = 95;
        backLayer.addChild(nameText);
        nameText = new LTextField();
        nameText.text = "电脑";
        nameText.weight = "bolder";
        nameText.color = "#ffffff";
        nameText.size = 24;
```

PDG

```
    nameText.x = enemyBitmap.x + (enemyBitmap.width - nameText.getWidth())/2;
    nameText.y = 95;
    backLayer.addChild(nameText);

    //结果显示层初始化
    initResultLayer();
    //操作层初始化
    initClickLayer();
}
function initResultLayer(){
    resultLayer = new LSprite();
    resultLayer.graphics.drawRect(4,'#ff8800',[0,0,150,110],true,'#ffffff');
    resultLayer.x = 10;
    resultLayer.y = 100;
    backLayer.addChild(resultLayer);
    selfTextAll = new LTextField();
    selfTextAll.text = "猜拳次数 : 0";
    selfTextAll.weight = "bolder";
    selfTextAll.x = 10;
    selfTextAll.y = 20;
    resultLayer.addChild(selfTextAll);
    selfTextWin = new LTextField();
    selfTextWin.text = "胜利次数 : 0";
    selfTextWin.weight = "bolder";
    selfTextWin.x = 10;
    selfTextWin.y = 40;
    resultLayer.addChild(selfTextWin);
    selfTextLoss = new LTextField();
    selfTextLoss.text = "失败次数 : 0";
    selfTextLoss.weight = "bolder";
    selfTextLoss.x = 10;
    selfTextLoss.y = 60;
    resultLayer.addChild(selfTextLoss);
    selfTextDraw = new LTextField();
    selfTextDraw.text = "平局次数 : 0";
    selfTextDraw.weight = "bolder";
    selfTextDraw.x = 10;
    selfTextDraw.y = 80;
    resultLayer.addChild(selfTextDraw);
}
function initClickLayer(){
    clickLayer = new LSprite();
    clickLayer.graphics.drawRect(4,'#ff8800',[0,0,300,110],true,'#ffffff');
    var msgText = new LTextField();
    msgText.text = "请出拳:";
    msgText.weight = "bolder";
    msgText.x = 10;
    msgText.y = 10;
    clickLayer.addChild(msgText);
    var btnShitou = getButton("shitou");
    btnShitou.x = 30;
```

```
    btnShitou.y = 35;
    clickLayer.addChild(btnShitou);
    btnShitou.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    var btnJiandao = getButton("jiandao");
    btnJiandao.x = 115;
    btnJiandao.y = 35;
    clickLayer.addChild(btnJiandao);
    btnJiandao.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    var btnBu = getButton("bu");
    btnBu.x = 200;
    btnBu.y = 35;
    clickLayer.addChild(btnBu);
    btnBu.addEventListener(LMouseEvent.MOUSE_UP,onclick);
    clickLayer.x = 250;
    clickLayer.y = 275;
    backLayer.addChild(clickLayer);
}
function onclick(event,display){
    var selfValue,enemyValue;
    if(display.name == "shitou"){
            selfValue = 0;
    }else if(display.name == "jiandao"){
            selfValue = 1;
    }else if(display.name == "bu"){
            selfValue = 2;
    }
    enemyValue = Math.floor(Math.random()*3);
    selfBitmap.bitmapData = showList[selfValue];
    enemyBitmap.bitmapData = showList[enemyValue];
    var result = checkList[selfValue][enemyValue];
    if(result == -1){
            loss += 1;
    }else if(result == 1){
            win += 1;
    }else{
            draw += 1;
    }
    selfTextWin.text = "胜利次数 : " + win;
    selfTextLoss.text = "失败次数 : " + loss;
    selfTextDraw.text = "平局次数 : " +  draw;
    selfTextAll.text = "猜拳次数 : " + (win + loss + draw);
}
function getButton(value){
    var btnUp = new LBitmap(new LBitmapData(imglist[value]));
    btnUp.scaleX = 0.5;
    btnUp.scaleY = 0.5;
    var btnOver = new LBitmap(new LBitmapData(imglist[value]));
    btnOver.scaleX = 0.5;
    btnOver.scaleY = 0.5;
    btnOver.x = 2;
    btnOver.y = 2;
```

```
        var btn = new LButton(btnUp,btnOver);
        btn.name = value;
        return btn;
    }
```

5.7　小结

本章通过一个简单的小游戏，让大家熟悉了LBitmapData、LBitmap、LSprite、LTextField等几个类的实际应用，掌握了如何使用lufylegend库件里的鼠标事件。这些都是lufylegend库件里最基本的组成元素，也是一个游戏最基本的组成元素。运用好这些基本元素，就可以制作出各种类型的游戏以及应用。本章的内容较为简单，并没有涉及时间轴循环，因为游戏中的所有事物，只有在触发点击事件的时候，才会发生变化。下一章会带大家认识一下lufylegend库件里的一个非常实用的功能——循环事件。

第6章　开发“俄罗斯方块”游戏

上一章中，介绍了 lufylegend 库件中基本元素的使用。本章主要讲解循环播放事件、键盘事件和触屏事件的应用。

“俄罗斯方块”（Tetris）是由俄罗斯人阿列克谢·帕基特诺夫于 1984 年发明的，之后提供给了各个游戏开发公司。由于其上手简单，内容新颖，趣味性强，在各个平台迅速传播，风靡全球。一直到现在这款游戏仍然拥有很高的人气，并且不断地衍生出各种新的版本和玩法。下面就通过俄罗斯方块这款游戏的制作来学习一下上述几个事件的应用。

6.1　游戏分析

首先来分析一下所需素材和要素。准备开发的游戏画面如图 6-1 所示。

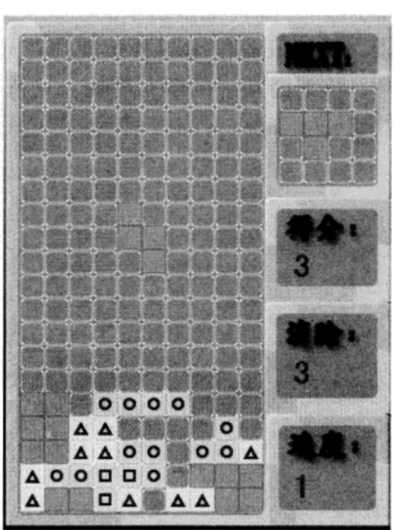


图 6-1　游戏画面预览图

需要掌握的要素有下面几点：

- 图片描画。图 6-1 中的背景和方块都是由图片组成的。
- 文字绘制。图 6-1 中的得分、消除层数和速度的显示都是由文字的绘制。
- 循环播放事件。实现方块的自动下落。
- 键盘事件。对方块的移动、变形等进行控制。
- 触屏事件。为了让玩家在手机上也可以体验这个游戏，还需要加入触屏事件，以便通过滑动屏幕来实现与键盘一样的功能。
- 游戏层次划分。在游戏的制作中，游戏的层次被分为背景层、进度条显示层、方块绘制层、方块预览层 4 层。其中，背景层用来显示背景图片，进度条显示层在读取图片的时候用来显示读取进度，方块绘制层用来显示游戏中的方块，方块预览层则用来显示预览栏中的预览方块。

6.2　必要的 JavaScript 知识

这款游戏的重点在于二维数组的应用，通过建立二维坐标数组，可实现方块的移动、变形以及判定。

PDG

二维数组又称为矩阵，在任意一门程序设计语言中都是必不可少的。JavaScript 中关于二维数组的定义如下所示：

```
var a = new Array(
            new Array(0,0),
            new Array(1,1)
);
```

或者直接用中括号来定义：

```
var a = [[0,0],[1,1]];
```

其实 JavaScript 中的二维数组就是将 N 个一维数组存储在另一个一维数组中。可以通过下面的代码来取得二维数组中的某个元素。

```
a[x][y]
```

因为 a[x] 表示数组 a 中的第 x 个元素，所以 a[x][y] 就表示 a[x] 这个数组的第 y 个元素。二维数组的应用很广泛，如游戏中的地图、网格等都需要由二维数组来实现。

6.3 游戏标题画面显示

先来利用 LSprite 对象的 graphics 属性和 LTextField 对象来制作一个游戏标题画面，如代码清单 6-1 所示。

代码清单 6-1

```
//声明变量
//进度条显示层，背景层，方块绘制层，方块预览层
var loadingLayer,backLayer,graphicsMap,nextLayer;
function main(){
            //背景层初始化
            backLayer = new LSprite();
            //在背景层上绘制黑色背景
            backLayer.graphics.drawRect(1,"#000000",[0,0,320,480],true,"#000000");
            //背景显示
            addChild(backLayer);
            gameInit();
}
//读取完所有图片，进行游戏标题画面的初始化工作
function gameInit(){
            //显示游戏标题
            var title = new LTextField();
            title.x = 50;
            title.y = 100;
            title.size = 30;
            title.color = "#ffffff";
            title.text = "俄罗斯方块";
            backLayer.addChild(title);
            //显示说明文字
```

```
            backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
            var txtClick = new LTextField();
            txtClick.size = 18;
            txtClick.color = "#ffffff";
            txtClick.text = "点击页面开始游戏";
            txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
            txtClick.y = 245;
            backLayer.addChild(txtClick);
}
```

运行效果如图 6-2 所示。

图 6-2 代码清单 6-1 的运行效果

代码解析

```
//进度条显示层，背景层，方块绘制层，方块预览层
var loadingLayer,backLayer,graphicsMap,nextLayer;
```

上述代码将新建 4 个层变量。

```
function main(){
            //背景层初始化
            backLayer = new LSprite();
            //在背景层上绘制黑色背景
            backLayer.graphics.drawRect(1,
"#000000",[0,0,320,480],true,"#000000");
            //背景显示
            addChild(backLayer);
            gameInit();
}
```

上述代码初始化背景层 backLayer，并使用 drawRect 方法绘制一个黑色矩形框作为游戏背景。

```
//显示游戏标题
var title = new LTextField();
title.x = 50;
title.y = 100;
title.size = 30;
title.color = "#ffffff";
title.text = "俄罗斯方块";
backLayer.addChild(title);
```

以上代码用于在 backLayer 背景层上添加一个 LTextField 对象，其中的文字设置为“俄罗斯方块”，它被作为游戏的标题。

```
//显示说明文
backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
var txtClick = new LTextField();
txtClick.size = 18;
txtClick.color = "#ffffff";
txtClick.text = "点击页面开始游戏";
```

```
txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
txtClick.y = 245;
backLayer.addChild(txtClick);
```

上述代码也会在 backLayer 背景层上添加一个 LTextField 对象，不过文字设置为“点击页面开始游戏”，将其作为游戏的操作说明。

6.4　向游戏里添加方块

为了使游戏中的方块多样化，准备 4 种不同颜色的方块图片，如图 6-3 ～图 6-6 所示。

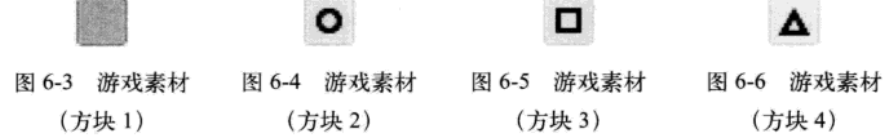


图 6-3　游戏素材（方块 1）　图 6-4　游戏素材（方块 2）　图 6-5　游戏素材（方块 3）　图 6-6　游戏素材（方块 4）

在显示方块之前，首先给游戏添加一张背景图片，为了尽量简化游戏的开发过程，我们将背景图片做成如图 6-7 所示的样式。

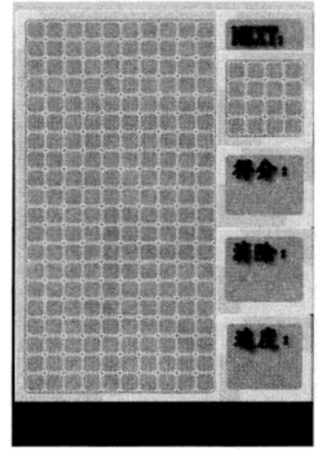


图 6-7　游戏素材（背景图片）

首先利用 LLoadManage 将上述图片全都读取至游戏中，如代码清单 6-2 所示。

代码清单　6-2

```
//声明变量
//进度条显示层，背景层，方块绘制层，方块预览层
var loadingLayer,backLayer,graphicsMap,nextLayer;
var imglist = {};
var imgData = new Array(
{name:"backImage",path:"./images/backImage.png"},
```

PDG

```
{name:"r1",path:"./images/r1.png"},
{name:"r2",path:"./images/r2.png"},
{name:"r3",path:"./images/r3.png"},
{name:"r4",path:"./images/r4.png"}
);
function main(){
            //背景层初始化
            backLayer = new LSprite();
            //在背景层上绘制黑色背景
            backLayer.graphics.drawRect(1,"#000000",
                            [0,0,320,480],true,"#000000");
            //背景显示
            addChild(backLayer);
            //进度条读取层初始化
            loadingLayer = new LoadingSample1();
            //进度条读取层显示
            backLayer.addChild(loadingLayer);
            //利用LLoadManage类，读取所有图片，并显示进度条进程
            LLoadManage.load(
                   imgData,
                   function(progress){
            loadingLayer.setProgress(progress);
                   },
                   gameInit
            );
}
//读取完所有图片，进行游戏标题画面的初始化工作
function gameInit(result){
            //取得图片读取结果
            imglist = result;
            //移除进度条层
            backLayer.removeChild(loadingLayer);
            loadingLayer = null;
            //显示游戏标题
            var title = new LTextField();
            title.x = 50;
            title.y = 100;
            title.size = 30;
            title.color = "#ffffff";
            title.text = "俄罗斯方块";
            backLayer.addChild(title);
            //显示说明文字
            backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
            var txtClick = new LTextField();
            txtClick.size = 18;
            txtClick.color = "#ffffff";
            txtClick.text = "点击页面开始游戏";
            txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
            txtClick.y = 245;
            backLayer.addChild(txtClick);
            //添加点击事件，点击画面则游戏开始
```

```
            backLayer.addEventListener(LMouseEvent.MOUSE_UP,gameToStart);
}
//游戏画面初始化
function gameToStart(){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();
            //背景图片显示
            var bitmap = new LBitmap(new LBitmapData(imglist["backImage"]));
            backLayer.addChild(bitmap);
}
```

其运行效果和图 6-7 是一样的。

代码解析

```
var imglist = {};
var imgData = new Array(
{name:"backImage",path:"./images/backImage.png"},
{name:"r1",path:"./images/r1.png"},
{name:"r2",path:"./images/r2.png"},
{name:"r3",path:"./images/r3.png"},
{name:"r4",path:"./images/r4.png"}
);
```

上面代码是为 LLoadManage 读取图片做准备的，它将所需读取的图片路径储存到数组中。

```
//进度条读取层初始化
loadingLayer = new LoadingSample1();
//进度条读取层显示
backLayer.addChild(loadingLayer);
//利用 LLoadManage 类，读取所有图片，并显示进度条进程
LLoadManage.load(
            imgData,
            function(progress){
                    loadingLayer.setProgress(progress);
            },
            gameInit
);
```

上面代码首先添加了进度条，然后利用 LLoadManage 来读取图片，读取完图片后调用 gameInit 函数，开始进行游戏标题画面的初始化工作。在上一章中曾提到过，lufylegend 库件中有 3 个进度条读取类，并且使用了 LoadingSample3 进度条来显示对象，本游戏使用了另一个进度条显示对象 LoadingSample1，读取过程中的效果如图 6-8 所示。

图 6-8　进度条效果

下面看代码清单 6-3 中的 gameInit 函数。

代码清单 6-3

```
// 读取完所有图片，进行游戏标题画面的初始化工作
function gameInit(result){
                //取得图片读取结果
                imglist = result;
                //移除进度条层
                backLayer.removeChild(loadingLayer);
                loadingLayer = null;
                //显示游戏标题
                var title = new LTextField();
                title.x = 50;
                title.y = 100;
                title.size = 30;
                title.color = "#ffffff";
                title.text = " 俄罗斯方块 ";
                backLayer.addChild(title);
                //显示说明文字
                backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
                var txtClick = new LTextField();
                txtClick.size = 18;
                txtClick.color = "#ffffff";
                txtClick.text = " 点击页面开始游戏 ";
                txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
                txtClick.y = 245;
                backLayer.addChild(txtClick);
                //添加点击事件，点击画面则游戏开始
                backLayer.addEventListener(LMouseEvent.MOUSE_UP,gameToStart);
}
```

将此处的 gameInit 函数与代码清单 6-1 做对比，可以得知，代码清单 6-2 首先将读取完的图片数据保存到 imglist 里，然后给 backLayer 层增加鼠标点击事件。当点击游戏画面的时候，将调用下面的 gameToStart 函数 ：

```
//游戏画面初始化
function gameToStart(){
                //背景层清空
                backLayer.die();
                backLayer.removeAllChild();
                //背景图片显示
                var bitmap = new LBitmap(
                       new LBitmapData(imglist["backImage"]));
                backLayer.addChild(bitmap);
}
```

当点击游戏画面的时候，首先将背景层清空，然后添加背景图片。LSprite 的 die 函数用于移除所有的事件侦听，removeAllChild 函数则会移除所有子对象。

接下来显示方块。为了显示方块，先要建立一个方块类 Box，如代码清单 6-4 所示。

代码清单　6-4

```
    function Box(){
                var self = this;
                self.box1=[[0,0,0,0],
                          [0,0,0,0],
                          [1,1,1,1],
                          [0,0,0,0]];
                self.box2=[[0,0,0,0],
                          [0,1,1,0],
                          [0,1,1,0],
                          [0,0,0,0]];
                self.box3=[[0,0,0,0],
                          [1,1,1,0],
                          [0,1,0,0],
                          [0,0,0,0]];
                self.box4=[[0,1,1,0],
                          [0,1,0,0],
                          [0,1,0,0],
                          [0,0,0,0]];
                self.box5=[[0,1,1,0],
                          [0,0,1,0],
                          [0,0,1,0],
                          [0,0,0,0]];
                self.box6=[[0,0,0,0],
                          [0,1,0,0],
                          [0,1,1,0],
                          [0,0,1,0]];
                self.box7=[[0,0,0,0],
                          [0,0,1,0],
                          [0,1,1,0],
                          [0,1,0,0]];
                self.box=[self.box1,self.box2,self.box3,self.box4,self.box5,self.
box6,self.box7];
    }
    Box.prototype = {
                getBox:function (){
                        var self = this;
                        var num=7*Math.random();
                        var index=parseInt(num);
                        var result = [];
                        var colorIndex = 1 + Math.floor(Math.random()*4);
                        var i,j;
                        for(i=0;i<4;i++){
                               var child = [];
                               for(j=0;j<4;j++){
                                    child[j] = self.box[index][i][j]*colorIndex;
                               }
                               result[i] = child;
                        }
                        return result;
                }
    }
```

代码解析

首先在构造器内建立 7 个二维数组，分别对应 7 种方块，如图 6-9 所示。

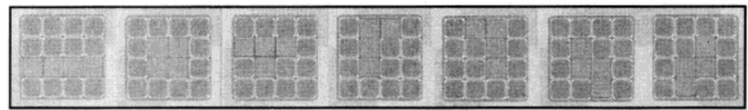


图 6-9 游戏中的 7 种方块

Box 类的 getBox 方法用于随机返回一个方块。colorIndex 代表方块的颜色，这个颜色是随机生成的。随机生成的方块数组的元素值为 0 或者 1，0 表示所处位置为空，1 则表示所处位置被砖块填充。然后将这个数组中每个元素的值与 colorIndex 相乘，所得的积将组成一个新的数组作为返回值。

有了 Box 类，就可以新增一个 BOX 变量了，如下所示：

```
//方块类变量，用于生成新的方块
var BOX;
```

在游戏开始的时候，将初始化一个 Box 类，用来生成新的方块，如下所示：

```
//方块类变量初始化
BOX = new Box();
```

为了实现整个画面中所有方块的绘制，我们将图 6-7 中左边的网格区域设置为一个二维数组，二维数组中每个元素的值都可以代表网格中是否有方块，以及方块的颜色。

首先，使用下面的代码来初始化二维数组的值，代表开始时画面上没有方块。

```
//方块坐标数组初始化
map=[
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0]
                ];
```

然后为了让上面的二维数组中每一个对应位置都能显示一个小方块，要新建一个二维数组 nodeList，并为其赋值，如代码清单 6-5 所示。

代码清单　6-5

```
//方块数据数组初始化
nodeList = [];
var i,j,nArr,bitmap;
for(i=0;i<map.length;i++){
            nArr = [];
            for(j=0;j<map[0].length;j++){
                  bitmap = new LBitmap(bitmapdataList[0]);
                  bitmap.x = bitmap.getWidth()*j+START_X1;
                  bitmap.y = bitmap.getHeight()*i+START_Y1;
                  graphicsMap.addChild(bitmap);
                  nArr[j] = {
                        "index":-1,
                        "value":0,
                        "bitmap":bitmap};
            }
            nodeList[i] = nArr;
}
```

这样就为 map 数组中每个元素的对应位置添加了一个 LBitmap 对象，用来显示一个方块的图片。其中用到了 bitmapdataList 数组，它里面储存了预先准备好的 4 种不同的方块图片的 LBitmapData 数据。代码如下：

```
//将方块的图片数据保存到数组内
            bitmapdataList = [
                  new LBitmapData (imglist["r1"]),
                  new LBitmapData (imglist["r2"]),
                  new LBitmapData (imglist["r3"]),
                  new LBitmapData(imglist["r4"])
            ];
```

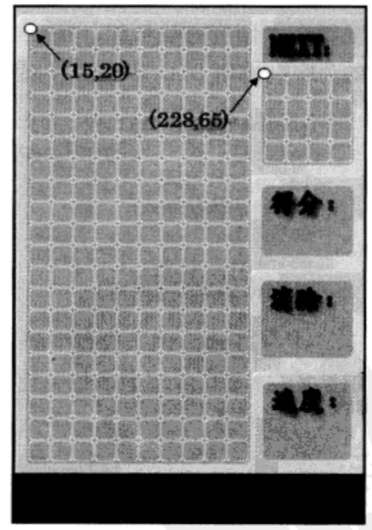


图 6-10　两个网格的起始坐标

另外代码清单 6-5 中用到了 START_X1 和 START_Y1 两个变量，它们表示网格的起始坐标。图 6-10 中分别标注了两个网格的起始坐标。

因为每个方块都是先在预览栏中出现的，所以就先来看看如何预览方块，如代码清单 6-6 所示。

代码清单　6-6

```
//获取下一方块
function getNewBox(){
            if (nextBox==null){
                  nextBox=BOX.getBox();
```

```
                }
                nowBox=nextBox;
                pointBox.x=3;
                pointBox.y=-4;
                nextBox=BOX.getBox();

                nextLayer.removeAllChild();
                var i,j,bitmap;
                for(i=0;i<nextBox.length;i++){
                        for(j=0;j<nextBox[0].length;j++){
                                if(nextBox[i][j] == 0){
                                        continue;
                                }
                                bitmap = new LBitmap(bitmapdataList[nextBox[i][j] - 1]);
                                bitmap.x = bitmap.getWidth()*j+START_X2;
                                bitmap.y = bitmap.getHeight()*i+START_Y2;
                                nextLayer.addChild(bitmap);
                        }
                }
}
```

代码解析

getNewBox 函数的功能是将预览方块 nextBox 的值赋给当前下落的方块 nowBox，然后重新获取预览方块数组。所以给当前下落方块赋值之前，首先必须判断 nextBox 是否为空，如为空则使用 BOX.getBox() 来取得一个方块数组。上述过程的具体实现如下所示：

```
if (nextBox==null){
                nextBox=BOX.getBox();
}
nowBox=nextBox;
pointBox.x=3;
pointBox.y=-4;
nextBox=BOX.getBox();
```

其中的 pointBox 代表当前下落方块的坐标值。通过上面代码得到预览方块后，将其显示到预览层 nextLayer 上，相关代码如下所示：

```
nextLayer.removeAllChild();
                var i,j,bitmap;
                for(i=0;i<nextBox.length;i++){
                        for(j=0;j<nextBox[0].length;j++){
                                if(nextBox[i][j] == 0){
                                        continue;
                                }
                                bitmap = new LBitmap(bitmapdataList[nextBox[i][j] - 1]);
                                bitmap.x = bitmap.getWidth()*j+START_X2;
                                bitmap.y = bitmap.getHeight()*i+START_Y2;
                                nextLayer.addChild(bitmap);
                        }
```

```
            }
}
```

其中 START_X2 和 START_Y2 是预览网格的起始坐标。

为了将方块添加到游戏界面上，给游戏增加如函数：

```
/添加方块
function plusBox(){
            var i,j;
            for(i=0;i<nowBox.length;i++){
                for(j=0;j<nowBox[i].length;j++){
                    if(i+pointBox.y < 0 || i+pointBox.y >= map.length ||
j+pointBox.x < 0 || j+pointBox.x >= map[0].length){
                        continue;
                    }
                    map[i+pointBox.y][j+pointBox.x]=nowBox[i][j]+map
[i+pointBox.y][j+pointBox.x];
                    nodeList[i+pointBox.y][j+pointBox.x]["index"] = map
[i+pointBox.y][j+pointBox.x] - 1;
                }
            }
}
```

因为 map 数组和 nodeList 数组分别代表网格数组的值和它们对应位置所显示的图片，所以改变它们的值就可以使网格发生变化。这样，将当前下落方块数组 nowBox 中储存的相应值通过计算添加给 map 和 nodeList 两个二维数组后，就能将当前图片绘制到网格上了。

游戏中的方块是不断下落的，所以游戏每次循环的时候，首先要将当前方块从网格中移除，然后再将当前方块下移一个网格，重新调用 plusBox()，这样就可以实现让方块不断下落这个功能了。为了将当前方块从网格中移除，需给游戏添加如下方法：

```
//移除方块
function minusBox(){
            var i,j;
            for(i=0;i<nowBox.length;i++){
                for(j=0;j<nowBox[i].length;j++){
                    if(i+pointBox.y < 0 || i+pointBox.y >= map.length ||
j+pointBox.x < 0 || j+pointBox.x >= map[0].length){
                        continue;
                    }
                    map[i+pointBox.y][j+pointBox.x]=map[i+pointBox.y]
[j+pointBox.x]-nowBox[i][j];
                    nodeList[i+pointBox.y][j+pointBox.x]["index"] = map
[i+pointBox.y][j+pointBox.x] - 1;
                }
            }
}
```

minusBox 函数的功能和 plusBox() 正好相反，它是将 nowBox 数组的值从 map 数组和 nodeList 数组中去除掉。

游戏中的方块不断下落，当一个方块落到另一个方块上的时候，它就无法再继续往下落了。要实现此功能，游戏每次循环的时候，必须在方块下落之前做一下判断，代码如下所示：

```
//判断是否可移动
function checkPlus(nx,ny){
    var i,j;
    //循环 nowBox 数组的每个元素
    for(i=0;i<nowBox.length;i++){
        for(j=0;j<nowBox[i].length;j++){
            if(i+pointBox.y + ny < 0){
                //判断网格还为落入网格范围内
                continue;
            }else if(i+pointBox.y + ny >= map.length || j+pointBox.x + nx < 0 ||
j+pointBox.x + nx >= map[0].length){
                //判断网格超出网格范围
                if(nowBox[i][j] == 0){
                //判断网格为空则继续判断
                    continue;
                }else{
                //判断网格不为空则代表无法移动
                    return false;
                }
            }
            if(nowBox[i][j] > 0 && map[i+pointBox.y + ny][j+pointBox.x + nx] > 0){
                //判断网格的位置有方块，而将要移动到此位置的当前方块也不为空，则代表无法移动
                return false;
            }
        }
    }
    return true;
}
```

checkPlus(nx,ny) 函数有两个参数，分别代表了当前方块在 x 轴和 y 轴移动的偏移量。上面的代码详细判断了方块移动时会出现的各种情况，如果返回值为 false，则代表再移动就会移出网格或者出现重叠方块。

接下来给游戏添加循环播放事件，代码如下所示：

```
//添加循环播放事件侦听
backLayer.addEventListener(LEvent.ENTER_FRAME, onframe);
```

下面是上面代码中的 onframe 函数。

```
//循环播放
function onframe(){
    //首先，将当前下落方块移出画面
    minusBox();
    if(speedIndex++ > speed){
        speedIndex = 0;
        if (checkPlus(0,1)){
```

```
                //可以下移，则将方块坐标下移一位
                pointBox.y++;
            }else{
                //无法下移
                plusBox();
                if(pointBox.y < 0){
                    //如果当前方块的坐标小于零，则游戏结束
                    gameOver();
                    return;
                }
                //取得新方块
                getNewBox();
            }
        }
        plusBox();
        drawMap();
    }
```

通过向 checkPlus 函数传入 (0,1)，来判断方块是否可以下移一位，如果可以则下移一位，如果不可以则判断游戏是否结束，如未结束则取得新的方块继续下落，如结束则显示游戏结束画面。其中 drawMap() 的功能是绘制网格中的所有小方块，具体代码如下所示：

```
    //绘制所有方块
    function drawMap(){
        var i,j,boxl = 15;
        for(i=0;i<map.length;i++){
            for(j=0;j<map[0].length;j++){
                if(nodeList[i][j]["index"] >= 0){
                    nodeList[i][j]["bitmap"].bitmapData = bitmapdataList[nodeList[i][j]["index"]];
                }else{
                    nodeList[i][j]["bitmap"].
bitmapData = null;
                }
            }
        }
    }
```

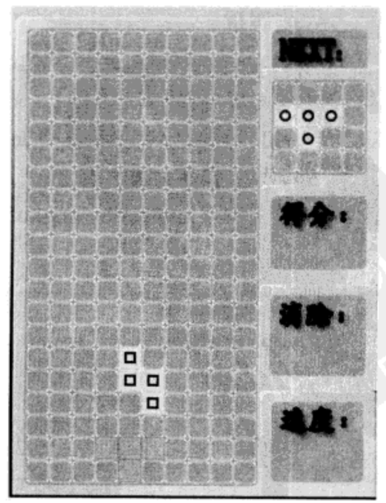


图 6-11　游戏效果图

上面代码通过判断 nodeList 中对应位置上的 index 值来决定是绘制相应的方块，还是将对应位置上的图片移出画面。运行游戏后得到的效果如图 6-11 所示。

最后，游戏结束画面的代码如下所示：

```
    //游戏结束
    function gameOver(){
                backLayer.die();
                var txt = new LTextField();
                txt.color = "#ff0000";
                txt.size = 40;
                txt.text = "游戏结束";
```

```
                txt.x = (LGlobal.width - txt.getWidth())*0.5;
                txt.y = 200;
                backLayer.addChild(txt);
    }
```

运行游戏后将会得到图 6-12 所示的效果图。

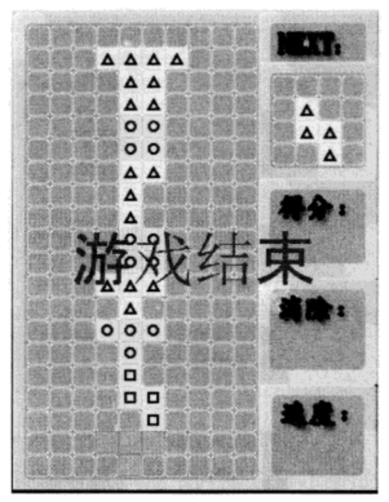


图 6-12 游戏结束画面

6.5 控制方块的移动

使用 HTML5 开发游戏的优点就在于跨平台，为了使游戏能够同时运行在电脑和智能手机上，必须同时给游戏添加键盘控制和触屏控制。在电脑上运行的时候，可以通过键盘的上下左右来控制方块的左右移动和变形，而在手机上运行时则可以通过手指在屏幕上向不同方向的滑动来实现和键盘相同的功能。

6.5.1 键盘事件

键盘事件很简单，如代码清单 6-7 所示。

代码清单 6-7

```
    if(!LGlobal.canTouch){
                // 在 PC 上运行的时候，添加键盘事件【上 下 左 右】
                LEvent.addEventListener(LGlobal.window,LKeyboardEvent.KEY_DOWN,onkeydown);
                LEvent.addEventListener(LGlobal.window,LKeyboardEvent.KEY_UP,onkeyup);
    }
```

代码解析

LGlobal.canTouch 是 lufylegend 库件中用来判断是否可以触屏的变量，如果它为 true 则代表可以触屏（如智能手机），如果它为 false 则代表是在电脑上。上面的代码表示当玩家环境为电脑的时候，要为游戏增加键盘事件。因为键盘事件必须加载到 window 窗口上，所以加载方式和鼠标事件是不同的，这里使用了 LEvent.addEventListener 来加载键盘事件。

下面来实现 onkeydown 和 onkeyup 这两个函数，如代码清单 6-8 所示。

代码清单 6-8

```
    // 键盘按下事件
    function onkeydown(event){
                if(myKey.keyControl != null)return;
                if(event.keyCode == 37){//left
                        myKey.keyControl = "left";
                }else if(event.keyCode == 38){//up
                        myKey.keyControl = "up";
                }else if(event.keyCode == 39){//right
```

```
                    myKey.keyControl = "right";
                }else if(event.keyCode == 40){//down
                    myKey.keyControl = "down";
                }
}
//键盘弹起事件
function onkeyup(event){
                myKey.keyControl = null;
                myKey.stepindex = 0;
}
```

在 onkeydown 函数中，键盘上每个键的 keyCode 都是不同的，所以可以通过 keyCode 值来判断玩家按下了键盘上的哪个键，如果按下了上、下、左、右 4 个方向键中的任何一个，则会将 myKey.keyControl 的值设置为所按键的值。在 onkeyup 函数中将 myKey.keyControl 的值清空，代表动作结束，如代码清单 6-9 所示。

代码清单 6-9

```
//循环播放
function onframe(){
    //首先，将当前下落方块移出画面
    minusBox();
    if(myKey.keyControl != null && myKey.stepindex-- < 0){
        myKey.stepindex = myKey.step;
        switch(myKey.keyControl){
            case "left":
                if(checkPlus(-1,0)){
                    pointBox.x -= 1;
                }
                break;
            case "right":
                if(checkPlus(1,0)){
                    pointBox.x += 1;
                }
                break;
            case "down":
                if(checkPlus(0,1)){
                    pointBox.y += 1;
                }
                break;
            case "up":
                changeBox();
                break;
        }
    }
    if(speedIndex++ > speed){
        speedIndex = 0;
        if (checkPlus(0,1)){
            pointBox.y++;
        }else{
```

```
                plusBox();
                if(pointBox.y < 0){
                    gameOver();
                    return;
                }
                getNewBox();
            }
        }
        plusBox();
        drawMap();
    }
```

分析上面的代码可知，当 myKey.keyControl 不为空的时候，当前方块的坐标 pointBox 会根据所按下的方向键来试着做相应的移动，当通过 checkPlus 判断出相应的方向可以移动时，坐标就会向相应的方向移动一个坐标。其中 myKey.stepindex 和 myKey.step 用来控制移动的频率。

当按下方向键“上”的时候，调用的方法是 changeBox，即让方块做一次旋转，实现代码如代码清单 6-10 所示。

代码清单 6-10

```
    //方块变形
    function changeBox(){
                var saveBox = nowBox;
                nowBox = [
                [0,0,0,0],
                [0,0,0,0],
                [0,0,0,0],
                [0,0,0,0]
                ];
                var i,j;
                for(i=0;i<saveBox.length;i++){
                        for(j=0;j<saveBox[1].length;j++){
                              nowBox[i][j]=saveBox[(3-j)][i];
                        }
                }
                if (!checkPlus(0,0)){
                        nowBox = saveBox;
                }
    }
```

上面代码所做的旋转如图 6-13 所示。

旋转后调用 checkPlus 函数并通过传入 (0,0) 参数来判断当前位置是否有其他与之相重叠的方块，如果有与它相重叠的方块就证明不可以变换，则将方块返回到未变化前的形状。

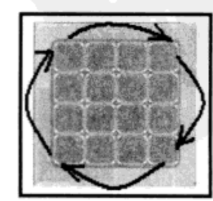


图 6-13 方块旋转示意图

现在通过键盘的上、下、左、右 4 个方向键，已经可以顺利操作当前落下的方块了。

6.5.2 触屏事件

触屏事件就是玩家使用智能手机玩游戏的时候，用手指触摸手机屏幕而触发的事件，如代码清单 6-11 所示。

代码清单 6-11

```
//添加鼠标按下、鼠标弹起和鼠标移动事件
backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,touchDown);
backLayer.addEventListener(LMouseEvent.MOUSE_UP,touchUp);
backLayer.addEventListener(LMouseEvent.MOUSE_MOVE,touchMove);
```

上面代码中虽然加入的是鼠标事件，但是因为在 lufylegend.js 库件中会根据游戏的运行环境进行鼠标事件和触屏事件的自动转换，所以鼠标事件就是触屏事件，而这里所说的鼠标在智能手机上就变成了手指。下面来逐一实现 touchDown、touchUp 和 touchMove 这 3 个函数，如代码清单 6-12 所示。

代码清单 6-12

```
//鼠标按下
function touchDown(event){
            myKey.isTouchDown = true;
            myKey.touchX = Math.floor(event.selfX / 20);
            myKey.touchY = Math.floor(event.selfY / 20);
            myKey.touchMove = false;
            myKey.keyControl = null;
}
//鼠标弹起
function touchUp(event){
            myKey.isTouchDown = false;
            if(!myKey.touchMove)myKey.keyControl = "up";
}
//鼠标移动
function touchMove(event){
            if(!myKey.isTouchDown)return;
            var mx = Math.floor(event.selfX / 20);
            if(myKey.touchX == 0){
                  myKey.touchX = mx;
                  myKey.touchY = Math.floor(event.selfY / 20);
            }
            if(mx > myKey.touchX){
                  myKey.keyControl = "right";
            }else if(mx < myKey.touchX){
                  myKey.keyControl = "left";
            }
            if(Math.floor(event.selfY / 20) > myKey.touchY){
                  myKey.keyControl = "down";
            }
}
```

通过上面代码可以看到，每当鼠标按下的时候，就会将 myKey.isTouchDown 设置为 true，表示鼠标已按下。这时会将 myKey.keyControl 设置为空，将 myKey.touchMove 设置为 false，相当于对 myKey 做了一个初始化，与此同时还记录下了点击位置的坐标。

当鼠标弹起的时候，将 myKey.isTouchDown 设置为 false，表示鼠标已抬起，并且如果鼠标没有发生过移动，则将方块变形。

最复杂的要数鼠标移动事件。通过将当前鼠标位置和记录下的鼠标位置进行对比，来判断是否移动和向哪个方向移动。

下面来修改一下游戏的 onframe 函数，以便加入通过触屏触发的事件，如代码清单 6-13 所示。

代码清单 6-13

```
//循环播放
function onframe(){
    //首先，将当前下落方块移出画面
    minusBox();
    if(myKey.keyControl != null && myKey.stepindex-- < 0){
        myKey.stepindex = myKey.step;
        switch(myKey.keyControl){
            case "left":
                if(checkPlus(-1,0)){
                    pointBox.x -= 1;
                    if(LGlobal.canTouch){
                        myKey.keyControl = null;
                        myKey.touchMove = true;
                        myKey.touchX = 0;
                    }
                }
                break;
            case "right":
                if(checkPlus(1,0)){
                    pointBox.x += 1;
                    if(LGlobal.canTouch){
                        myKey.keyControl = null;
                        myKey.touchMove = true;
                        myKey.touchX = 0;
                    }
                }
                break;
            case "down":
                if(checkPlus(0,1)){
                    pointBox.y += 1;
                    if(LGlobal.canTouch){
                        myKey.keyControl = null;
                        myKey.touchMove = true;
                        myKey.touchY = 0;
                    }
                }
```

```
                break;
            case "up":
                changeBox();
                if(LGlobal.canTouch){
                    myKey.keyControl = null;
                    myKey.stepindex = 0;
                }
                break;
        }
    }
    if(speedIndex++ > speed){
        speedIndex = 0;
        if (checkPlus(0,1)){
            pointBox.y++;
        }else{
            plusBox();
            if(pointBox.y < 0){
                gameOver();
                return;
            }
            getNewBox();
        }
    }
    plusBox();
    drawMap();
}
```

可以看到，如果是手机操作的，每次方块发生旋转或者移动时，都会为相应的动作做一些初始化工作。如果是向左或向右滑动屏幕，则将 myKey.touchMove 设置为 true，表明正在滑动屏幕，同时为了正确判断下一次的触屏位置，将 myKey.touchX 设置为 0。如果是向下滑动屏幕，也会将 myKey.touchMove 设置为 true，不同的是将 myKey.touchY 设置为 0，因为向下滑动屏幕只跟 Y 坐标有关系。另外无论向哪个方向滑动屏幕，为了避免同一动作的重复执行，都会将 myKey.keyControl 的值设置为空。好了，如果你有手机的话，可以用手机测试一下，已经可以通过滑动屏幕来控制游戏了。

6.6　方块的消除和得分的显示

方块的消除是指当方块满一整行的时候，将这一整行消除，具体算法如代码清单 6-14 所示。

代码清单　6-14

```
// 消除指定层的方块
function moveLine(line){
                var i;
                for(i=line;i>1 ;i--){
```

```
                        for(j=0;j<map[0].length;j++){
                            map[i][j]=map[i-1][j];
nodeList[i][j].index=nodeList[i-1][j].index;
                        }
                    }
                    for(j=0;j<map[0].length;j++){
                        map[0][j]=0;
                        nodeList[0][j].index=-1;
                    }
}
//消除可消除的方块
function removeBox(){
                    var i,j,count = 0;
                    for(i=pointBox.y;i<(pointBox.y+4);i++){
                        if(i < 0 || i >= map.length)continue;
                        for(j=0;j<map[0].length;j++){
                            if(map[i][j]==0){
                                break;
                            }
                            if(j==map[0].length - 1){
                                moveLine(i);
                                count++;
                            }
                        }
                    }
}
```

代码解析

因为每次消除的只能是刚刚落下的方块所在行，所以只需要从代表当前下落方块的坐标pointBox开始算起即可，如果有一整行全部为方块的话，就调用moveLine函数，来消除这一整行的方块。在moveLine函数中会利用改变二维数组的元素值来实现方块的消除。

运行上述代码，可以发现游戏已经有了消除行的功能了。

下面要做的工作就是将游戏所得分数等显示到游戏界面上，并且每消除100层，游戏速度就加快一点。首先来显示得分等信息，如代码清单6-15所示。

代码清单 6-15

```
//得分显示
                pointText = new LTextField();
                pointText.x = 240;
                pointText.y = 200;
                pointText.size = 20;
                backLayer.addChild(pointText);
                //消除层数显示
                delText = new LTextField();
                delText.x = 240;
                delText.y = 290;
                delText.size = 20;
```

```
            backLayer.addChild(delText);
            //速度显示
            speedText = new LTextField();
            speedText.x = 240;
            speedText.y = 385;
            speedText.size = 20;
            backLayer.addChild(speedText);
```

可以看到，利用 LTextField 对象可以轻松地实现文字的显示。为了时刻显示正确信息，游戏中可能会有多个地方需要刷新这几个 LTextField 对象的值，所以建立如下函数来统一操作。

```
//游戏得分、消除层数以及游戏速度显示
function showText(){
            pointText.text = point;
            delText.text = del;
            speedText.text = speedMax - speed + 1;
}
```

在每次消除层的时候，都需要调用 showText 函数来刷新数据。

```
//消除可消除的方块
function removeBox(){
    var i,j,count = 0;
    for(i=pointBox.y;i<(pointBox.y+4);i++){
        if(i < 0 || i >= map.length)continue;
        for(j=0;j<map[0].length;j++){
            if(map[i][j]==0){
                break;
            }
            if(j==map[0].length - 1){
                moveLine(i);
                count++;
            }
        }
    }
    if(count == 0)return;
    del += count;
    if(count == 1){
        point += 1;
    }else if(count == 2){
        point += 3;
    }else if(count == 3){
        point += 6;
    }else if(count == 4){
        point += 10;
    }
    if(speed > 1 && del / 100 >= (speedMax - speed + 1)){
        speed--;
    }
```

PDG

```
    showText();
}
```

上面代码通过消除的层数来计算新的消除层数 del 和得分 point，然后每消除 100 层将速度值提高一个单位。最后的效果图如图 6-1 所示。

这样俄罗斯方块这款游戏就制作完成了，大家可以通过随书附赠的第 6 章源代码，查看一下这个游戏的完整代码。

6.7 小结

通过本章内容的学习，大家应该进一步熟悉了 LBitmapData、LBitmap、LSprite、LTextField 等几个类的使用方法。循环播放事件可以为游戏添加时间轴的概念，没有它的话游戏中的一切都只能是静止的，所以说几乎所有的游戏和应用都会用到它。鼠标事件和键盘事件是玩家控制游戏的手段，所以必须熟练掌握这几个事件的用法，学会如何获取鼠标的位置坐标和键盘中被按下的按钮。在接下来的几章里会带大家继续熟悉这些类和事件的使用方法。

PDG

第 7 章　开发“是男人就下一百层”游戏

上一章中，通过制作俄罗斯方块这款小游戏，为大家介绍了 lufylegend 库件中的循环播放事件、键盘事件和触屏事件的应用，本章将为大家讲解如何制作一款 2D 卷轴游戏。

卷轴游戏是因为游戏的背景看起来像是卷轴在滚动而得名，“是男人就下一百层”即为一款 2D 卷轴游戏。游戏中玩家的任务是让主角持续下落，其过程中会遇到各种麻烦，或地板消失，或地板带刺致使血量降低，可以让玩家挑战一下自己的极限。下面就来看一下如何实现这类游戏的制作。

7.1　游戏分析

首先来分析一下所需素材和要素。准备开发的游戏画面如图 7-1 所示。

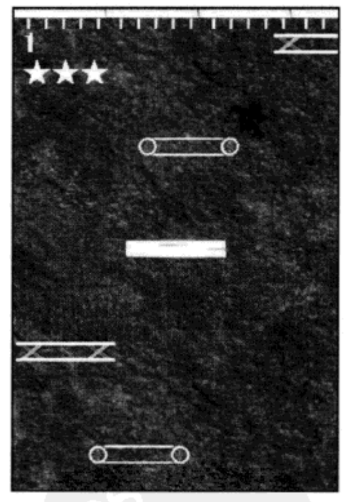


图 7-1　游戏画面预览图

要制作此游戏，需要用到的要素如下：

- 图片描画

图 7-1 中的背景和方块都是由图片组成的。

- 文字绘制

图 7-1 中左上角的得分部分是由文字的绘制。

- 循环播放事件

屏幕的卷轴移动和游戏主角的下坠都需要循环播放事件来完成。

- 键盘事件

游戏主角的左右移动等动作由键盘来控制。

- 触屏事件

加入触屏控制，让玩家在手机上也同样可以体验这个游戏。

- 游戏层次划分

在本游戏制作中，可以将游戏层次划分为：进度条显示层、背景层、人物层、障碍层。进度条显示层用来显示图片读取时的进度，背景层用来显示不断卷动的背景图片，人物层用来显示游戏中的主角，障碍层用来显示不断出现的各种地板。

7.2　游戏标题画面显示

和上一章一样，先利用 LSprite 对象的 graphics 属性和 LTextField 对象制作一个游戏标

题画面，如代码清单 7-1 所示。

代码清单 7-1

```
//声明变量
//进度条显示层，背景层，方块绘制层，方块预览层
var loadingLayer,backLayer,graphicsMap,nextLayer;
function main(){
        //背景层初始化
        backLayer = new LSprite();
        //在背景层上绘制黑色背景
        backLayer.graphics.drawRect(1,"#000000",[0,0,320,480],true,"#000000");
        //背景显示
        addChild(backLayer);
        gameInit();
}
//读取完所有图片，进行游戏标题画面的初始化工作
function gameInit(){
        //显示游戏标题
        var title = new LTextField();
        title.x = 50;
        title.y = 100;
        title.size = 30;
        title.color = "#ffffff";
        title.text = "是男人就下100层";
        backLayer.addChild(title);
        //显示说明文字
        backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
        var txtClick = new LTextField();
        txtClick.size = 18;
        txtClick.color = "#ffffff";
        txtClick.text = "点击页面开始游戏";
        txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
        txtClick.y = 245;
        backLayer.addChild(txtClick);
}
```

运行效果如图 7-2 所示。

上面这段代码除了标题说明文字外，其余内容与第 6 章的代码清单 6-1 完全相同，所以看代码清单 6-1 中的代码解析说明即可。

图 7-2 代码清单 7-1 效果图

7.3 读取图片与背景显示

为了使游戏中的背景图片可以连贯地卷动显示，就需要准备一个可以上下衔接的图片，如图 7-3 所示。

这张图片的特别之处在于它上下可以衔接成一张图片，如果将两张同样的图片对接，就会出现如图 7-4 所示的效果。

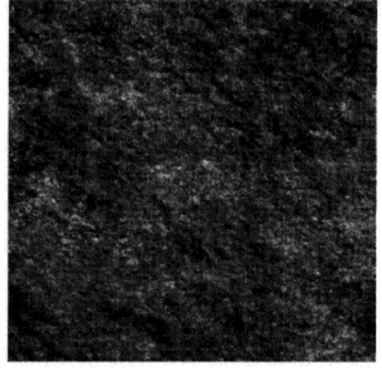


图 7-3　游戏素材（背景图片）

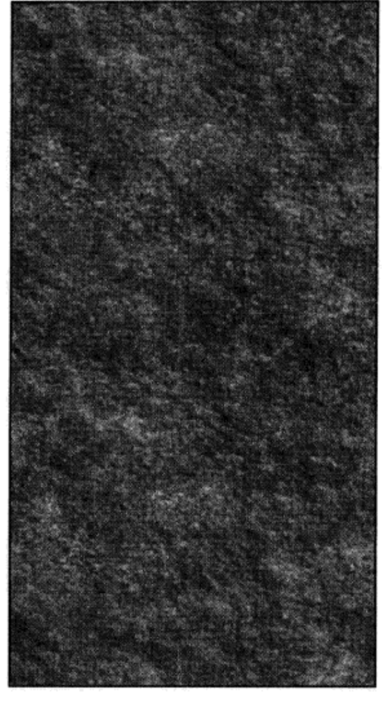


图 7-4　游戏素材对接后的效果图

准备好图片后，我们依然利用 LLoadManage 将图片读取进游戏中，如代码清单 7-2 所示。

代码清单　7-2

```
//声明变量
//游戏主层，进度条显示层，背景层，障碍层
var backLayer,loadingLayer,background,stageLayer;
var imglist = {};
var imgData = new Array(
{name:"back",path:"./images/back.png"}
);
function main(){
            //游戏主层初始化
            backLayer = new LSprite();
            //在主层上绘制黑色背景
            backLayer.graphics.drawRect(1,"#000000",
                        [0,0,320,480],true,"#000000");
            //背景显示
            addChild(backLayer);
            //进度条读取层初始化
            loadingLayer = new LoadingSample2(50);
            //进度条读取层显示
            backLayer.addChild(loadingLayer);
            //利用 LLoadManage 类，读取所有图片，并显示进度条进程
```

PDG

```
            LLoadManage.load(
                    imgData,
                    function(progress){
            loadingLayer.setProgress(progress);
                    },
                    gameInit
            );
}
//读取完所有图片，进行游戏标题画面的初始化工作
function gameInit(result){
            //取得图片读取结果
            imglist = result;
            //移除进度条层
            backLayer.removeChild(loadingLayer);
            loadingLayer = null;
            //显示游戏标题
            var title = new LTextField();
            title.x = 50;
            title.y = 100;
            title.size = 30;
            title.color = "#ffffff";
            title.text = "是男人就下100层";
            backLayer.addChild(title);
            //显示说明文字
            backLayer.graphics.drawRect(1,"#ffffff",[50,240,220,40]);
            var txtClick = new LTextField();
            txtClick.size = 18;
            txtClick.color = "#ffffff";
            txtClick.text = "点击页面开始游戏";
            txtClick.x = (LGlobal.width - txtClick.getWidth())/2;
            txtClick.y = 245;
            backLayer.addChild(txtClick);
            //添加点击事件，点击画面则游戏开始
            backLayer.addEventListener(LMouseEvent.MOUSE_UP,gameStart);
}
//游戏画面初始化
function gameStart(){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();
            //背景图片显示
}
```

代码解析

```
var imglist = {};
var imgData = new Array(
{name:"back",path:"./images/back.png"}
);
```

PDG

上面代码是为 LLoadManage 读取图片做准备的，将所需读取的图片路径储存到数组中。

```
//进度条读取层初始化
loadingLayer = new LoadingSample2(50);
//进度条读取层显示
backLayer.addChild(loadingLayer);
//利用 LLoadManage 类，读取所有图片，并显示进度条进程
LLoadManage.load(
            imgData,
            function(progress){
                   loadingLayer.setProgress(progress);
            },
            gameInit
);
```

上面代码首先添加了进度条，然后利用 LLoadManage 来读取图片。读取完图片后将调用 gameInit 函数。在前几章中，已经使用过了 lufylegend 库件中 3 个进度条读取类中的两个，本游戏将使用第 3 种进度条显示对象 LoadingSample2。传入 LoadingSample2 构造器的参数 50 是画面中所显示的文字的大小，如果没有传递参数的话，则文字的大小将为默认大小。读取过程中的效果如图 7-5 所示。

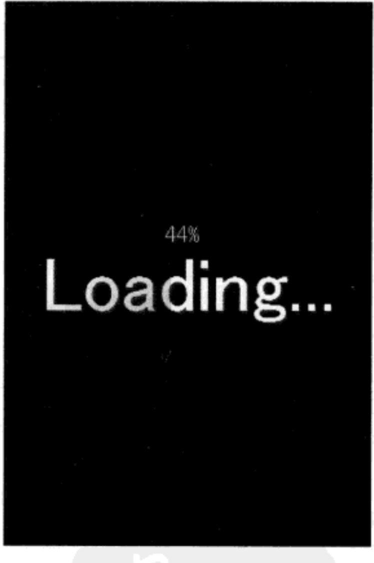


图 7-5　进度条效果图

将代码清单 7-2 中的 gameInit 函数与代码清单 7-1 中的 gameInit 函数进行比较，可以得知，代码清单 7-2 首先会将读取完的图片数据保存到 imglist 里，并且在最后给 backLayer 层增加鼠标点击事件。当点击游戏画面的时候，将调用下面的 gameStart 函数：

```
//游戏画面初始化
function gameStart(){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();
            //背景图片显示
}
```

当点击游戏画面的时候，首先要将背景层清空，然后添加背景图片。LSprite 的 die 函数表示移除所有的事件侦听，removeAllChild 函数表示移除所有子对象。

接下来看看如何来显示卷轴背景。首先新建一个 Background 类，如代码清单 7-3 所示。

代码清单　7-3

```
function Background(){
            base(this,LSprite,[]);
            var self = this;
```

```
            self.bitmapData = new LBitmapData(imglist["back"]);
            self.bitmap1 = new LBitmap(self.bitmapData);
            self.addChild(self.bitmap1);
            self.bitmap2 = new LBitmap(self.bitmapData);
            self.bitmap2.y = self.bitmap1.getHeight();
            self.addChild(self.bitmap2);
            self.bitmap3 = new LBitmap(self.bitmapData);
            self.bitmap3.y = self.bitmap1.getHeight()*2;
            self.addChild(self.bitmap3);
}
Background.prototype.run = function(){
            var self = this;
            self.bitmap1.y -= STAGE_STEP;
            self.bitmap2.y -= STAGE_STEP;
            self.bitmap3.y -= STAGE_STEP;
            if(self.bitmap1.y < -self.bitmap1.getHeight()){
                   self.bitmap1.y = self.bitmap2.y;
                   self.bitmap2.y = self.bitmap1.y + self.bitmap1.getHeight();
                   self.bitmap3.y = self.bitmap1.y + self.bitmap1.getHeight()*2;
            }
}
```

代码解析

首先在构造器内建立 3 个 LBitmap 对象，并依次进行显示。因为背景图片是可以无缝衔接的，所以显示到画面上就好像一张图片一样。

Background 类的 run 函数是将 Background 对象中的 3 个子图片向上移动一个 STAGE_STEP 步长，这个步长会在定义部分提前定义好相应的值，待第一个 LBitmap 对象移出屏幕后，就会把第二个 LBitmap 对象的坐标赋值给第一个 LBitmap 对象，然后再重新计算其他两个 LBitmap 对象的坐标。这样做是为了让游戏背景始终由这 3 个 LBitmap 对象来显示。

有了 Background 类，添加背景就方便了，如代码清单 7-4 所示。

代码清单 7-4

```
//游戏画面初始化
function gameStart(restart){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();

            background = new Background();
            backLayer.addChild(background);
            backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
function onframe(){
            background.run();
}
```

代码运行效果如图 7-6 所示。

代码解析

首先实例化背景层，代码如下所示：

```
background = new Background();
backLayer.addChild(background);
```

然后添加循环播放侦听。

```
backLayer.addEventListener(LEvent.ENTER_
FRAME,onframe);
```

在循环播放侦听函数中不断调用 Background 类的 run 函数，以达到背景卷动效果，代码如下所示：

```
function onframe(){
    background.run();
}
```

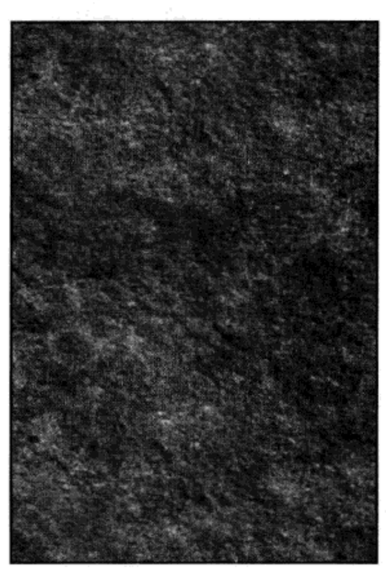


图 7-6　背景卷轴效果图

7.4　添加一个静止的地板

之所以先添加地板，后添加游戏主角，是因为游戏主角在游戏中是不断下落的，如果没有地板，游戏主角就无法在游戏画面上停留。

从图 7-1 中可以看到，在这个游戏中，有各种各样的地板，这些地板有一些共同的属性，比如它们都在不停地向上移动。为了实现这些共同的属性，我们先来建立一个 Floor 类，作为所有地板的父类，这个父类里包含所有地板的公共部分，如代码清单 7-5 所示。

代码清单　7-5

```
function Floor(){
            base(this,LSprite,[]);
            var self = this;
            self.hy = 0;
            self.setView();
}
Floor.prototype.setView = function(){}
Floor.prototype.onframe = function (){
            var self = this;
            self.y -= STAGE_STEP;
};
Floor.prototype.hitRun = function (){};
```

代码解析

在构造器里，首先用 base 函数实现了对 LSprite 类的继承。这里需要注意，游戏主角不一定正好站在地板的上方，还有可能站在地板中间等地方，对比效果如图 7-7 和图 7-8 所示。

图 7-7 人物在地板正上方效果图

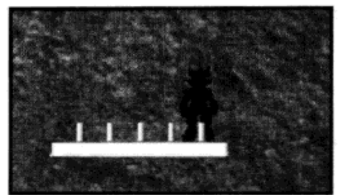


图 7-8 人物在地板中间效果图

因此在代码中加入了变量 hy，它是用来控制游戏主角相对于地板的位置的，后面会详细说明。

每个地板都有一个皮肤，一种样式。下面的 setView 函数用于给地板设定皮肤。

```
Floor.prototype.setView = function(){}
```

因为每个地板的皮肤不同，所以这个函数会在每个子类中进行重写。

下面的 onframe 函数实现了每调用一次 onframe，地板就向上移动一个 STAGE_STEP 长度。

```
Floor.prototype.onframe = function (){
            var self = this;
            self.y -= STAGE_STEP;
};
```

当游戏主角落到每个地板上时，会根据地板的不同而有不同的表现，比如强制左移，强制右移，或者向上跳起等。下面的 hitRun 函数是为了实现这些不同的表现而建立的。

图 7-9 游戏素材（地板 1）

```
Floor.prototype.hitRun = function (){};
```

这个函数与 setView 一样，都需要在每个子类中进行重写。

下面准备第一个地板的图片，如图 7-9 所示。

然后建立一个继承自 Floor 类的子类 Floor01，代码如下所示：

```
function Floor01(){
            base(this,Floor,[]);
}
Floor01.prototype.setView = function(){
            var self = this;
            self.bitmap = new LBitmap(new LBitmapData(imglist["floor0"]));
            self.addChild(self.bitmap);
}
```

代码解析

首先用 base 函数来实现类的继承，然后重写 setView 函数，以给 Floor01 类设定皮肤。

```
var self = this;
self.bitmap = new LBitmap(new LBitmapData(imglist["floor0"]));
```

```
self.addChild(self.bitmap);
```

因为在 Floor01 类中，只是设定了地板的皮肤，所以这个地板相当于一个静止的地板。在实现了 Floor01 类之后，接下来把它添加到画面上。

先将地板层进行实例化，代码如下：

```
function stageInit(){
            stageLayer = new LSprite();
            backLayer.addChild(stageLayer);
}
```

然后建立 addStage 函数，添加一个 Floor01 地板。

```
function addStage(){
            var mstage;
            mstage = new Floor01();
            mstage.y = 480;
            mstage.x = Math.random()*280;
            stageLayer.addChild(mstage);
}
```

上面的代码很简单，就是新建一个 Floor01 对象，随机设定它的坐标，并将其添加到地板层上。

接着，修改 gameStart 和 stageInit 函数，让地板对象不断地添加到屏幕上，代码如下所示：

```
function gameStart(){
           //背景层清空
           backLayer.die();
           backLayer.removeAllChild();

           background = new Background();
           backLayer.addChild(background);

           stageInit();

           backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
function onframe(){
           background.run();
           if(stageSpeed-- < 0){
                  stageSpeed = 100;
                  addStage();
           }
           var key = null;
           for(key in stageLayer.childList){
                  var _child = stageLayer.childList[key];
                  _child.onframe();
           }
}
```

代码解析

在 gameStart 函数中添加了对地板初始化函数 stageInit 的调用。

下面的代码使用 stageSpeed 来控制添加地板的快慢。对于代码中的 stageSpeed 变量的值，循环函数 onframe 每执行一次就会减去 1，当 stageSpeed 变量的值小于 0 的时候，又会被重新设定为 100，并且每调用一次 addStage 函数就加入一块地板，也就是说循环函数每执行 100 次会向屏幕上添加一块地板。

```
if(stageSpeed-- < 0){
            stageSpeed = 100;
            addStage();
}
```

然后，下面的代码将循环移动屏幕上的所有地板，实现所有的地板都不断地向上移动的效果。

```
var key = null;
for(key in stageLayer.childList){
            var _child = stageLayer.childList[key];
            _child.onframe();
}
```

最后，运行代码得到的效果如图 7-10 所示。

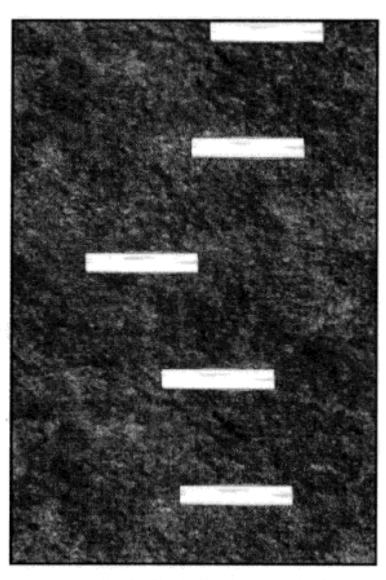


图 7-10 添加静止的地板后的效果图

7.5 添加游戏主角

本游戏依然利用 HTML5 的跨平台特点，分别给游戏添加键盘事件和触屏事件，以使游戏在电脑上和智能手机上都能运行。

7.5.1 让游戏主角出现在画面上

新建一个 Chara 类，来实现对游戏主角的控制，如代码清单 7-6 所示。

代码清单 7-6

```
function Chara(){
            base(this,LSprite,[]);
            var self = this;
            self.moveType = null;
            self.hp = 3;
            self.maxHp = 3;
            self.hpCtrl = 0;
            self.isJump = true;
            self.index = 0;
            self.speed = 0;
            self._charaOld = 0;
```

```
            var list = LGlobal.divideCoordinate(960,50,1,24);
            var data = new LBitmapData(imglist["hero"],0,0,40,50);
            self.anime = new LAnimation(self,data,[
                   [list[0][0]],
                   [list[0][1]],
                   [list[0][2],list[0][3],list[0][4],list[0][5],list[0][6],
list[0][7],list[0][8],list[0][9],list[0][10],list[0][11],list[0][12]],
                   [list[0][13],list[0][14],list[0][15],list[0][16],list[0][17],
list[0][18],list[0][19],list[0][20],list[0][21],list[0][22],list[0][23]]
            ]);
    }

    Chara.prototype.onframe = function (){
            var self = this;
            self._charaOld = self.y;
            self.y += self.speed;
            self.speed += g;
            if(self.speed>20)self.speed=20;
            if(self.y > LGlobal.height){
                   self.hp = 0;
            }
            if(self.moveType == "left"){
                   self.x -= MOVE_STEP;
            }else if(self.moveType == "right"){
                   self.x += MOVE_STEP;
            }
            if(self.x  < -10){
                   self.x = -10;
            }else if(self.x > LGlobal.width - 30){
                   self.x = LGlobal.width - 30;
            }
            if(self.index-- > 0){
                   return;
            }
            self.index = 10;
            self.anime.onframe();
    };
    Chara.prototype.changeAction = function(){
            var self = this;
            if(self.moveType == "left"){
                   hero.anime.setAction(3);
            }else if(self.moveType == "right"){
                   hero.anime.setAction(2);
            }else if(hero.isJump){
                   hero.anime.setAction(1,0);
            }else{
                   hero.anime.setAction(0,0);
            }
    }
```

代码解析

先来解释一下构造器的各个属性。

```
self.moveType = null;
```

moveType 用来控制游戏主角是左移还是右移。

```
self.hp = 3;
self.maxHp = 3;
```

hp 表示游戏主角的当前血量，mapHp 表示游戏主角的最大血量。

```
self.hpCtrl = 0;
```

当游戏主角的血量降低之后，用 hpCtrl 来控制血量恢复的速度，后面会有更详细的介绍。

```
self.isJump = true;
```

isJump 用来表示游戏主角的当前是否处于跳跃状态。

```
self.index = 0;
```

index 用来控制游戏主角动作变换的快慢，在 Chara 类的 onframe 函数中会用到。

```
self.speed = 0;
```

speed 表示游戏主角下落的速度。

```
self._charaOld = 0;
```

_charaOld 用来记录游戏主角每次下落之前的 y 坐标。

```
var list = LGlobal.divideCoordinate(960,50,1,24);
var data = new LBitmapData(imglist["hero"],0,0,40,50);
self.anime = new LAnimation(self,data,[
            [list[0][0]],
            [list[0][1]],
            [list[0][2],list[0][3],list[0][4],list[0][5],list[0][6],list[0][7],
list[0][8],list[0][9],list[0][10],list[0][11],list[0][12]],
            [list[0][13],list[0][14],list[0][15],list[0][16],list[0][17],list[0][18],
list[0][19],list[0][20],list[0][21],list[0][22],list[0][23]]
            ]);
```

要了解上面的代码，首先来看一下游戏主角的图片，如图 7-11 所示。

图 7-11 游戏素材（游戏主角）

首先用 divideCoordinate 切割图片，取得游戏主角每个动作图片的坐标。将这些动作图片的坐标数组中的坐标元素重新排列组合，组合成一个新的二维数组，分别表示游戏主角的

站立、跳跃、左移和右移 4 个动作。

接下来看 Chara 类的 onframe 函数，它在游戏每次循环时都会被调用。

```
self._charaOld = self.y;
self.y += self.speed;
```

上面的代码首先记录下游戏主角的 y 坐标，然后根据游戏主角的下落速度让游戏主角下落一次。

```
self.speed += g;
if(self.speed>20)self.speed=20;
```

g 表示加速度，根据加速度改变游戏主角的下落速度。当下落速度大于 20 的时候，就将速度设置为 20，从而限制主角下落速度的最大值，以免下落速度过大，在与地板做碰撞检测的时候会漏掉一部分地板。

```
if(self.y > LGlobal.height){
        self.hp = 0;
}
```

当游戏主角掉落到屏幕之外时，直接将血量降为 0。

```
if(self.moveType == "left"){
        self.x -= MOVE_STEP;
}else if(self.moveType == "right"){
        self.x += MOVE_STEP;
}
```

根据游戏主角的 moveType 属性，来控制游戏主角是左移还是右移一个 MOVE_STEP 单位。

```
if(self.x  < -10){
        self.x = -10;
}else if(self.x > LGlobal.width - 30){
        self.x = LGlobal.width - 30;
}
```

控制游戏主角的 x 坐标，防止游戏主角移出游戏的屏幕之外。

```
if(self.index-- > 0){
        return;
}
self.index = 10;
self.anime.onframe();
```

用 index 属性来控制游戏主角动作的变换快慢。这里设置为 10 表示游戏的循环函数每执行 10 次，主角的动作变换一次。

最后再来看 Chara 类的 changeAction 函数，此函数用于控制游戏主角的动作。

```
Chara.prototype.changeAction = function(){
```

```
            var self = this;
            if(self.moveType == "left"){
                    hero.anime.setAction(3);
            }else if(self.moveType == "right"){
                    hero.anime.setAction(2);
            }else if(hero.isJump){
                    hero.anime.setAction(1,0);
            }else{
                    hero.anime.setAction(0,0);
            }
    }
```

上面的代码首先根据 moveType 来设定游戏主角的动作是左移还是右移，当既不是左移也不是右移的时候，则判断游戏主角的状态是不是跳跃状态，是则设定动作为跳跃，否则设定动作为站立。

接下来对游戏 main.js 文件中的 onframe 做一些修改，如代码清单 7-7 所示。

代码清单 7-7

```
    function onframe(){
            background.run();
            if(stageSpeed-- < 0){
                    stageSpeed = 100;
                    addStage();
            }
            var key = null,found = false;
            hero.isJump = true;
            for(key in stageLayer.childList){
                    var _child = stageLayer.childList[key];
                    if(_child.y < -_child.getHeight()){
                            stageLayer.removeChild(_child);
                    }

                    if(!found &&
                       hero.x + 30 >= _child.x && hero.x <= _child.x + 90 &&
                       hero.y + 50 >= _child.y+_child.hy && hero._charaOld + 50 <=
_child.y+_child.hy+1){
                                    hero.isJump = false;
                                    hero.changeAction();
                                    _child.child = hero;
                                    hero.speed = 0;
                                    hero.y = _child.y - 49 + _child.hy;
                                    _child.hitRun();
                                    found = true;
                    }else{
                            _child.child = null;
                    }
                    _child.onframe();
            }
            if(hero.isJump)hero.anime.setAction(1,0);
```

```
            if(hero){
                hero.onframe();
                if(hero.hp <= 0){
                    backLayer.removeChild(hero);
                    hero = null;
                    gameOver();
                }
            }
}
function gameOver(){
}
```

代码解析

可以看到，在 onframe 中循环了所有的地板，并依次进行了判断和处理。下面进一步详细解说。

```
var key = null,found = false;
hero.isJump = true;
```

上面代码建立相应的变量，found 用来表示游戏主角是否已经落到了地板上，并且将游戏主角的状态设置为跳跃。而在后面的循环判断中对游戏主角当前的动作进行判断，如果游戏主角已经落到了地板的上面，且处在跳跃状态，则会解除跳跃状态。

在分析地板的相应判断之前，先来看下面的代码：

```
if(hero.isJump)hero.anime.setAction(1,0);
if(hero){
            hero.onframe();
            if(hero.hp <= 0){
                backLayer.removeChild(hero);
                hero = null;
                gameOver();
            }
}
```

上面代码表示，当游戏主角处于跳跃状态时，将它的动作设置为跳跃。另外，当游戏主角不为空时，持续调用它的 onframe 函数，来实现移动、下落、动作变换等相应的工作。最后当主角的血量减为 0 时，游戏结束。

下面来看一看循环每个地板后的处理。

```
var _child = stageLayer.childList[key];
if(_child.y < -_child.getHeight()){
            stageLayer.removeChild(_child);
}
```

上面的代码首先取出当前地板，并且判断当前地板是否已经处于屏幕之外，如果在屏幕之外则将其移除。

```
if(!found &&
```

```
    hero.x + 30 >= _child.x && hero.x <= _child.x + 90 &&
    hero.y + 50 >= _child.y+_child.hy &&
hero._charaOld + 50 <= _child.y+_child.hy+1){
        hero.isJump = false;
        hero.changeAction();
        _child.child = hero;
        hero.speed = 0;
        hero.y = _child.y - 49 + _child.hy;
        _child.hitRun();
        found = true;
}else{
        _child.child = null;
}
```

上面代码首先判断游戏主角是否已经站在地板上了，因为 _charaOld 代表游戏主角每次下落前的 y 坐标，所以如果出现了如图 7-12 所示的情况，则表示游戏主角已经站在了地板的上面。

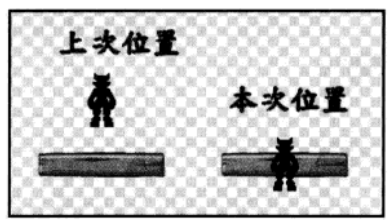


图 7-12 是否站在地板上判断示意图

如果游戏主角已经站在了地板的上面，则进行下列处理：

```
hero.isJump = false;
hero.changeAction();
_child.child = hero;
hero.speed = 0;
hero.y = _child.y - 49 + _child.hy;
_child.hitRun();
found = true;
```

上面的代码进行了下面几个处理：

- 解除游戏主角的跳跃状态。
- 调用 changeAction 函数，改变游戏主角的动作。
- 将当前地板的 child 设置为游戏的主角。
- 设置游戏主角的下落速度 speed 为 0。
- 修改游戏主角的 y 坐标，使其正好站在游戏地板的上面。如果地板的 hy 属性不是 0，则将位置进行调整，这就会出现如图 7-8 所示的情况。
- 调用地板的 hitRun 函数，根据地板的种类，进行不同的处理。
- 修改 found，表示游戏人物已经站到了地板的上面。

如果游戏主角没有站在当前地板的上面，则将它的 child 设置为空。

```
_child.child = null;
```

大家可能不明白上面所提到的地板的 child 属性的作用，下面就来解释一下。修改 Floor 类的 onframe 函数，代码如下所示：

```
Floor.prototype.onframe = function (){
            var self = this;
            self.y -= STAGE_STEP;
            if(self.child){
                   self.child.y -= STAGE_STEP;
            }
};
```

可以看到，当地板的 child 不为空时，会让游戏主角上升一个 STAGE_STEP 步长，使得游戏主角和地板一起上升。

接下来，修改 gameStart 函数，把游戏人物添加到游戏中来。

```
//游戏画面初始化
function gameStart(restart){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();

            background = new Background();
            backLayer.addChild(background);

            hero = new Chara();
            hero.x = 140;
            hero.y = 100;
            hero.hp = hero.maxHp;
            backLayer.addChild(hero);

            stageInit();

            backLayer.addEventListener(LEvent.
ENTER_FRAME,onframe);
}
```

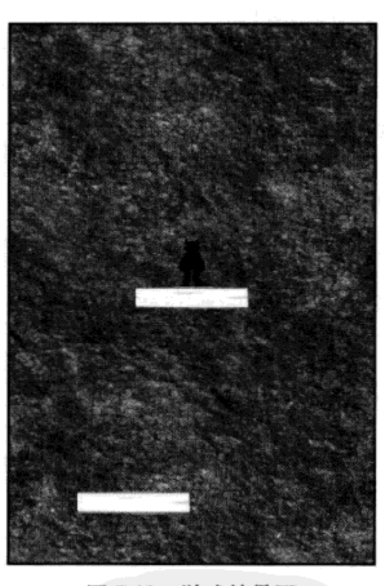


图 7-13　游戏效果图

最后，运行上面的代码，效果如图 7-13 所示。

7.5.2　通过键盘事件来控制游戏主角的移动

通过键盘控制游戏主角的移动时，要给游戏添加键盘事件，代码如下所示：

```
if (!LGlobal.canTouch){
            LEvent.addEventListener(window,LKeyboardEvent.KEY_DOWN,down);
            LEvent.addEventListener(window,LKeyboardEvent.KEY_UP,up);
            }
```

LGlobal.canTouch 是为了判断当前浏览器是电脑还是智能手机，即判断是否可以触屏。上面的代码表示当游戏运行环境为电脑的时候，为游戏增加键盘事件。

下面来实现 down 和 up 这两个函数，代码如下所示：

```
function up(event){
```

```
            hero.moveType = null;
            hero.changeAction();
}
function down(event){
            if(hero.moveType)return;
            if(event.keyCode == 37){
                  hero.moveType = "left";
            }else if(event.keyCode == 39){
                  hero.moveType = "right";
            }
            hero.changeAction();
}
```

代码解析

keyCode 等于 37 表示左方向键，keyCode 等于 39 表示右方向键，down 函数表示当键盘按键按下的时候，根据 keyCode 的不同而改变游戏主角的移动方向；up 函数表示当键盘按键弹起的时候，将游戏主角的动作设置为站立。

运行游戏，效果如图 7-14 所示。

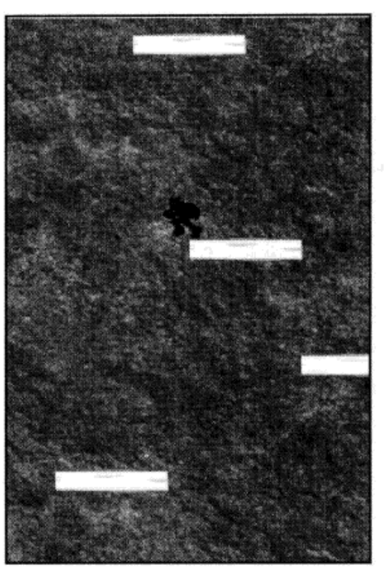


图 7-14 键盘控制游戏主角移动效果图

7.5.3 通过触屏事件来控制游戏主角的移动

下面再来看看如何用触屏事件来实现游戏主角移动的控制。为游戏添加触屏事件，代码如下所示：

```
//鼠标按下事件
backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,mousedown);
//鼠标弹起事件
backLayer.addEventListener(LMouseEvent.MOUSE_UP,mouseup);
```

下面来逐一实现上面代码中的这几个函数，如下所示：

```
function mouseup(event){
            hero.moveType = null;
            hero.changeAction();
}
function mousedown(event){
            if (event.offsetX <= LGlobal.width*0.5){
                  down({keyCode:37});
            }else{
                  down({keyCode:39});
            }
}
```

上面代码在调用 down 函数时传入参数 {keyCode:37}，是在模拟一个键盘事件传入 down 函数，因为在上一节中，已经实现了键盘控制人物移动的功能，所以这个地方模拟一个键盘

事件（相当于将触屏事件转换成了键盘事件）即可，且触屏弹起事件与键盘的按键弹起事件功能完全相同。

下面的代码表示在触屏按下事件中，将所点击的屏幕位置与屏幕的中间位置做对比。

```
if (event.offsetX <= LGlobal.width*0.5)
```

运行代码，触摸屏幕，得到的游戏效果图与图 7-14 相同。

7.6　添加多种多样的地板

到目前为止，已经实现了游戏的基本功能，只不过目前所有出现在游戏中的地板都是静止的。下面来向游戏中添加多种多样的地板。

7.6.1　会消失的地板

会消失的地板指的是当游戏主角站到这种地板上时，稍微停留一段时间，地板便会逐渐消失。下面建立一个继承自 Floor 的 Floor02 类来实现这种地板，如代码清单 7-8 所示。

代码清单　7-8

```
function Floor02(){
            base(this,Floor,[]);
            var self = this;
            self.ctrlIndex = 0;
}
Floor02.prototype.setView = function(){
            var self = this;
            self.bitmap = new LBitmap(new LBitmapData(imglist["floor1"],0,0,100,20));
            self.addChild(self.bitmap);
}
Floor02.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.ctrlIndex++;
            if(self.ctrlIndex >= 40){
                   self.parent.removeChild(this);
}else if(self.ctrlIndex == 20){
            self.bitmap.bitmapData.setCoordinate(100,0);
            }
}
```

代码解析

图 7-15　游戏素材（会消失的地板）

首先在构造器中实现对 Floor 类的继承，然后在 setView 函数中设定该地板对象自己的皮肤，看图 7-15 中的素材图片。

下面的代码中在新建 LBitmapData 对象的时候传入了所有参数，以控制 LBitmapData 对

象的大小。因为图 7-15 中的图片大小是 200×20，所以这次新建的 LBitmapData 对象相当于将图 7-15 中的左边图片显示到画面上。

```
Floor02.prototype.setView = function(){
            var self = this;
            self.bitmap = new LBitmap(new LBitmapData(imglist["floor1"],0,0,100,20));
            self.addChild(self.bitmap);
}
```

下面具体看一下 Floor02 类的 hitRun 函数。其中，callParent 函数表示调用父类的函数，它需要两个参数，第一个参数是所调用的父类函数的函数名，第二个参数是固定值 arguments。调用父类的 hitRun 是为了在运行自己本身的 hitRun 函数时，首先运行一次父类的 hitRun 函数。

```
self.callParent("hitRun",arguments);
```

下面的代码是通过 ctrlIndex 的值来控制地板的状态。

```
self.ctrlIndex++;
if(self.ctrlIndex >= 40){
            self.parent.removeChild(this);
}else if(self.ctrlIndex == 20){
            self.bitmap.bitmapData.setCoordinate(100,0);
}
```

当游戏主角站在当前地板上的时候 hitRun 函数才会被调用，所以上面的代码表达的意思是当游戏主角站在当前地板上的时候，地板的 ctrlIndex 值开始自增计算，当自增 20 次后，将地板的图片改变，改变图片用的是 LBitmapData 类的 setCoordinate 函数，这个函数可以用来改变 LBitmapData 类中图片的显示区域。上面的 setCoordinate(100,0) 表示将显示区域由图 7-15 的左边图片变为右边图片。若游戏的主角继续站在地板上，地板的 ctrlIndex 值自增 40 次之后，则该地板从画面上消失。

有了 Floor02 类之后，再来修改 addStage 函数，代码如下所示：

```
function addStage(){
            var mstage;
            var index = Math.random() * 2;
            if(index < 1){
                    mstage = new Floor02();
            }else if(index < 2){
                    mstage = new Floor01();
            }
            mstage.y = 480;
            mstage.x = Math.random()*280;
            stageLayer.addChild(mstage);
}
```

代码解析

上面代码很简单，就是利用随机函数来实现随机加入 Floor02 地板。

游戏效果如图 7-16 所示。

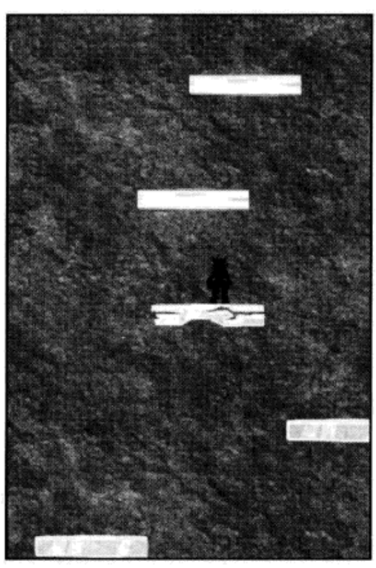


图 7-16 会消失的地板运行效果图

7.6.2 带刺的地板

带刺的地板是指当游戏主角站到这种地板上的时候，会被刺伤，所以血量会减去一个单位。下面建立一个继承自 Floor 的类 Floor03 来实现这种地板，如代码清单 7-9 所示。

代码清单 7-9

```
function Floor03(){
        base(this,Floor,[]);
        this.hit = false;
        this.hy = 10;
}
Floor03.prototype.setView = function(){
        var self = this;
        self.bitmap = new LBitmap(new LBitmapData(imglist["floor3"]));
        self.addChild(self.bitmap);
}
Floor03.prototype.hitRun = function (){
        var self = this;
        self.callParent("hitRun",arguments);
        if(self.hit)return;
        self.hit = true;
        self.child.hp -= 1;
}
```

PDG

代码解析

首先在构造器中实现对 Floor 类的继承，然后在 setView 函数中设定自己的皮肤，如图 7-17 所示。

在构造器里设定了 hy 属性为 10，表示游戏主角站在地板上的时候，相对于地板的位置为 10，效果如图 7-8 所示。

图 7-17 游戏素材（带刺的地板）

下面再来看看 hitRun 函数。

```
Floor03.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            if(self.hit)return;
            self.hit = true;
            self.child.hp -= 1;
}
```

上面的代码说明，当游戏主角站在当前地板上时，血量减去一个单位。

接着，修改 addStage 函数，代码如下所示：

```
function addStage(){
            var mstage;
            var index = Math.random() * 3;
            if(index < 1){
                    mstage = new Floor03();
            }else if(index < 2){
                    mstage = new Floor02();
            }else if(index < 3){
                    mstage = new Floor01();
            }
            mstage.y = 480;
            mstage.x = Math.random()*280;
            stageLayer.addChild(mstage);
}
```

游戏效果如图 7-18 所示。

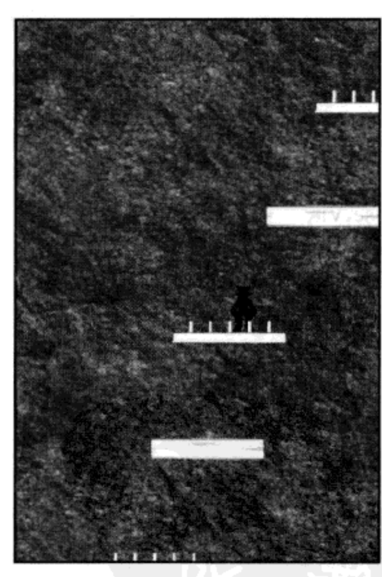


图 7-18 带刺地板运行效果图

7.6.3 带有弹性的地板

带有弹性的地板是指当游戏主角站到这种地板上的时候，会被弹起来。下面建立一个继承自 Floor 的类 Floor04 来实现这种地板，如代码清单 7-10 所示。

代码清单 7-10

```
function Floor04(){
            base(this,Floor,[]);
            var self = this;
            self.ctrlIndex = 0;
            this.hy = 8;
```

```
}
Floor04.prototype.setView = function(){
            var self = this;
            self.bitmap = new LBitmap(new LBitmapData(imglist["floor2"],0,0,100,20));
            self.addChild(self.bitmap);
}
Floor04.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.ctrlIndex = 0;
            self.child.y -= self.hy;
            self.child.speed = -4;
            self.child.isJump = true;
            self.child = null;
            self.bitmap.bitmapData.setCoordinate(100,0);
}
Floor04.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            self.ctrlIndex++;
            if(self.ctrlIndex == 20)self.bitmap.bitmapData.setCoordinate(0,0);
}
```

代码解析

图 7-19　游戏素材（带有弹性的地板）

首先在构造器中实现对 Floor 类的继承，然后在 setView 函数中设定自己的皮肤，如图 7-19 所示。

在构造器里设定了 hy 属性为 8，表示游戏主角站在地板上时，相对于地板的位置为 8。而 setView 函数和 7.1.1 节中的功能一样，都是实现素材图片的左边图片。

下面再来看看 hitRun 函数。

```
Floor04.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.ctrlIndex = 0;
            self.child.y -= self.hy;
            self.child.speed = -4;
            self.child.isJump = true;
            self.child = null;
            self.bitmap.bitmapData.setCoordinate(100,0);
}
```

上面代码表示，首先改变游戏主角的 y 坐标，将游戏主角移到地板之外，然后将它的速度设定为 -4，速度是负数的时候表示向上跳跃，并且设置主角为跳跃状态。最后通过 setCoordinate 函数，将地板素材的显示区域改为图 7-19 右边的图片。

继续来看 onframe 函数，代码如下所示：

```
Floor04.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            self.ctrlIndex++;
            if(self.ctrlIndex == 20)
            self.bitmap.bitmapData.setCoordinate(0,0);
}
```

上面代码表示，使用地板的 ctrlIndex 属性来恢复弹性地板的状态，重新将地板素材的显示区域改为图 7-19 左边的图片。

最后，修改 addStage 函数，代码如下所示：

```
function addStage(){
            var mstage;
            var index = Math.random() * 4;
            if(index < 1){
                  mstage = new Floor04();
            }else if(index < 2){
                  mstage = new Floor03();
            }else if(index < 3){
                  mstage = new Floor02();
            }else if(index < 4){
                  mstage = new Floor01();
            }
            mstage.y = 480;
            mstage.x = Math.random()*280;
            stageLayer.addChild(mstage);
}
```

游戏效果如图 7-20 所示。

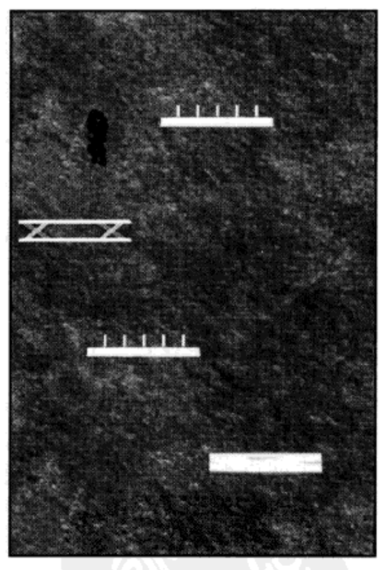


图 7-20 带有弹性地板运行效果图

7.6.4 向左和向右移动的地板

向左和向右移动的地板是指当游戏主角站到这种地板上的时候，地板会强制游戏主角向相应的方向移动。首先来实现向右移动的地板，下面建立一个继承自 Floor 的类 Floor05 实现这种地板，如代码清单 7-11 所示。

代码清单 7-11

```
function Floor05(){
            base(this,Floor,[]);
}
Floor05.prototype.setView = function(){
            var self = this;
            self.graphics.drawRect(1,"#cccccc",[10,2,80,16]);
            self.wheelLeft = new LBitmap(new LBitmapData(imglist["wheel"]));
            self.addChild(self.wheelLeft);
            self.wheelRight = new LBitmap(new LBitmapData(imglist["wheel"]));
```

```
            self.wheelRight.x = 100 - self.wheelRight.getWidth()
            self.addChild(self.wheelRight);
}
Floor05.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            self.wheelLeft.rotate += 2;
            self.wheelRight.rotate += 2;
}
Floor05.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.child.x += (MOVE_STEP-1);
}
```

代码解析

首先在构造器中实现对 Floor 类的继承，然后在 setView 函数中设定该地板对象自己的皮肤，如图 7-21 所示。

图 7-21　游戏素材（齿轮）

下面主要来看一下 setView 函数。

```
Floor05.prototype.setView = function(){
            var self = this;
            self.graphics.drawRect(1,"#cccccc",[10,2,80,16]);
            self.wheelLeft = new LBitmap(new LBitmapData(imglist["wheel"]));
            self.addChild(self.wheelLeft);
            self.wheelRight = new LBitmap(new LBitmapData(imglist["wheel"]));
            self.wheelRight.x = 100 - self.wheelRight.getWidth()
            self.addChild(self.wheelRight);
}
```

上面代码表示，首先画了一个矩形，然后在矩形的两边分别加入两个齿轮，形成一个传送带的样式。然后在下面的 onframe 函数中不断地改变两个齿轮的角度，从而实现转动的效果。

```
Floor05.prototype.onframe = function (){
          var self = this;
          self.callParent("onframe",arguments);
          self.wheelLeft.rotate += 2;
          self.wheelRight.rotate += 2;
}
```

接着，在 hitRun 函数中持续改变游戏主角的 x 坐标，实现强制游戏主角向右移动的效果。

```
Floor05.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.child.x += (MOVE_STEP-1);
}
```

上面的代码已经完成了可以强制游戏主角向右移动的地板，而强制游戏主角向左移动的地板和它的原理完全相同，代码也几乎完全相同。下面建立一个继承自 Floor 的类 Floor06 来实现这种地板，如代码清单 7-12 所示。

代码清单 7-12

```
function Floor06(){
            base(this,Floor,[]);
}
Floor06.prototype.setView = function(){
            var self = this;
            self.graphics.drawRect(1,"#cccccc",[10,2,80,16]);
            self.wheelLeft = new LBitmap(new LBitmapData(imglist["wheel"]));
            self.addChild(self.wheelLeft);
            self.wheelRight = new LBitmap(new LBitmapData(imglist["wheel"]));
            self.wheelRight.x = 100 - self.wheelRight.getWidth()
            self.addChild(self.wheelRight);
}
Floor06.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            self.wheelLeft.rotate -= 2;
            self.wheelRight.rotate -= 2;
}
Floor06.prototype.hitRun = function (){
            var self = this;
            self.callParent("hitRun",arguments);
            self.child.x -= (MOVE_STEP-1);
}
```

代码解析

对照一下代码清单 7-11 中的代码，就可以发现，不同之处如下：

- ❑ onframe 函数中齿轮的旋转方向不同。
- ❑ hitRun 函数中强制游戏主角移动的方向不同。

最后，修改 addStage 函数，代码如下所示：

```
function addStage(){
            var mstage;
            var index = Math.random() * 6;
            if(index < 1){
                   mstage = new Floor06();
            }else if(index < 2){
                   mstage = new Floor05();
            }else if(index < 3){
                   mstage = new Floor04();
            }else if(index < 4){
                   mstage = new Floor03();
            }else if(index < 5){
```

PDG

```
            mstage = new Floor02();
        }else if(index < 6){
            mstage = new Floor01();
        }
        mstage.y = 480;
        mstage.x = Math.random()*280;
        stageLayer.addChild(mstage);
}
```

游戏效果如图 7-22 所示。

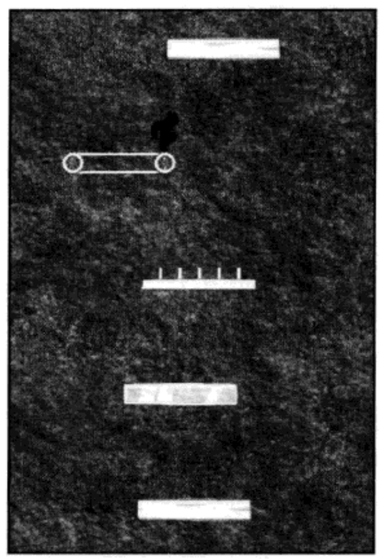


图 7-22 移动地板运行效果图

7.7 游戏数据的显示

游戏的操作部分已经全都完成了，下面将游戏的数据部分显示到画面上。游戏的数据包括游戏主角的血量和游戏主角总共下降的层数。

首先建立几个变量来表示这些数据。

```
layers = 0,layersText,hpText;
```

layers 表示游戏主角下降的层数，layersText 用来将层数的数值显示到画面上，hpText 用来将游戏主角的血量显示到画面上。

接着建立 showInit 函数，来进行文字显示的初始化，代码如下所示：

```
function showInit(){
        layersText = new LTextField();
```

```
            layersText.x = 10;
            layersText.y = 20;
            layersText.size = 20;
            layersText.weight = "bolder";
            layersText.color = "#ffff00";
            backLayer.addChild(layersText);

            hpText = new LTextField();
            hpText.x = 10;
            hpText.y = 50;
            hpText.size = 20;
            hpText.weight = "bolder";
            hpText.color = "#ffffff";
            backLayer.addChild(hpText);
}
```

上面的代码简单地设置了文字的颜色和大小，然后通过下面的 showView 将游戏数据显示到游戏画面上。

```
function showView(){
            layersText.text = (layers / LGlobal.height) >>> 0;
            hpText.text = hero.hp==3?"★ ★ ★":hero.hp==2?"★ ★":hero.
hp==1?"★":"";
}
```

上面的代码需要解释的是，(layers / LGlobal.height) >>> 0; 表示用位移计算进行取整，这个运算也可以用 Math.floor() 函数来替换。

完成了上面的工作，游戏基本上就完成了。但是仔细观察图 7-1 这个游戏的预览图，你会发现在游戏顶部还有一排刺，当游戏主角碰到顶层的刺时，他的血量应该减去一个单位。

所以还要建立顶层初始化函数，进行顶层的初始化，代码如下所示：

```
function wallInit(){
            var bitmapDataUp = new LBitmapData(imglist["floor3"]);
            var bitmapUp;
            bitmapUp = new LBitmap(bitmapDataUp);
            bitmapUp.rotate = 180;
            addChild(bitmapUp);
            bitmapUp = new LBitmap(bitmapDataUp);
            bitmapUp.x = 90;
            bitmapUp.rotate = 180;
            addChild(bitmapUp);
            bitmapUp = new LBitmap(bitmapDataUp);
            bitmapUp.x = 90*2;
            bitmapUp.rotate = 180;
            addChild(bitmapUp);
            bitmapUp = new LBitmap(bitmapDataUp);
            bitmapUp.x = 90*3;
            bitmapUp.rotate = 180;
```

```
            addChild(bitmapUp);

    }
```

接下来需要在游戏主角接触到顶部的时候，将他的血量减去一个单位。在前面所有的代码中，改动的游戏主角 y 坐标的地方一共有两处，下面还要对这两处地方进行修改。

第一个地方是，弹性地板将游戏主角弹起时，要在主角跳跃上升的过程中，判断一下是否它接触到了顶层的刺。修改 Chara 类的 onframe 函数，代码如下所示：

```
Chara.prototype.onframe = function (){
        var self = this;
        self._charaOld = self.y;
        self.y += self.speed;
        self.speed += g;
        if(self.speed>20)self.speed=20;
        if(self.y > LGlobal.height){
            self.hp = 0;
        }else if(self.y < 10){
            self.hp--;
            self.y += 20;
            if(self.speed < 0)self.speed=0;
        }
        if(self.moveType == "left"){
            self.x -= MOVE_STEP;
        }else if(self.moveType == "right"){
            self.x += MOVE_STEP;
        }
        if(self.x  < -10){
            self.x = -10;
        }else if(self.x > LGlobal.width - 30){
            self.x = LGlobal.width - 30;
        }
        self.addHp();
        if(self.index-- > 0){
            return;
        }
        self.index = 10;
        self.anime.onframe();
};
```

当游戏主角的 y 坐标小于 10 的时候表示已经接触到了画面顶部的刺，将血量减去一个单位后，将主角向下移动一定距离，使其离开顶层一段距离，并且如果当前主角的速度为负值，即正在向上运动的时候，将它的速度设为 0，以免重复减血。

第二个地方是当游戏主角随着地板一起上升的时候，需要进行相应的判断。修改所有地板的父类 Floor 的 onframe 函数，代码如下所示：

```
Floor.prototype.onframe = function (){
        var self = this;
```

```
            self.y -= STAGE_STEP;
            if(self.child){
                   self.child.y -= STAGE_STEP;
                   if(self.child.y < 10){
                          self.child.hp--;
                          self.child.y += 20;
                          self.child = null;
                   }
            }
};
```

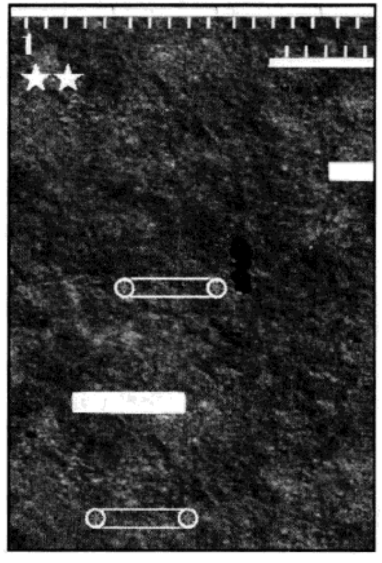


图 7-23 显示数据运行效果图

和第一种情况一样，当游戏主角的 y 坐标小于 10 的时候表示已经接触到了画面顶部的刺，将血量减去一个单位后，将主角向下移动一定距离，使其离开顶层一段距离，以免重复减血。

运行上面的代码，得到运行效果如图 7-23 所示。

7.8 游戏结束与重开

在代码清单 7-7 中，有个空的 gameOver 函数，表示游戏结束。

下面来具体实现 gameOver 函数的内容，如代码清单 7-13 所示。

代码清单 7-13

```
function gameOver(){
        backLayer.die();
        var overLayer = new LSprite();
        backLayer.addChild(overLayer);
        overLayer.graphics.drawRect(4,'#ff8800',[0,0,200,100],true,'#ffffff');
        overLayer.x = (LGlobal.width - overLayer.getWidth())*0.5;
        overLayer.y = (LGlobal.height - overLayer.getHeight())*0.5;
        var txt;
        txt = new LTextField();
        txt.text = "成绩："+layersText.text;
        txt.x = 20;
        txt.y = 20;
        overLayer.addChild(txt);
        txt = new LTextField();
        txt.text = "继续加油！！";
        txt.x = 20;
        txt.y = 50;
        overLayer.addChild(txt);
        backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,function(event){
              gameStart(true);
        });
}
```

代码解析

下面的代码首先将加载到 backLayer 层的事件清除。

```
backLayer.die();
```

接着，新建一个 overLayer 层来实现游戏结果的显示。

```
var overLayer = new LSprite();
backLayer.addChild(overLayer);
```

然后通过一些 LTextField 对象，将最终的游戏结果显示到 overLayer 结果层上。最后，给游戏加上一个点击事件，从而让游戏可以重新开始。

```
backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,
            function(event){
                   gameStart(true);
            });
```

因为键盘事件只需要加载一遍，所以要修改一下 gameStart 游戏初始化函数，代码如下所示：

```
function gameStart(restart){
            //背景层清空
            backLayer.die();
            backLayer.removeAllChild();
            background = new Background();
            backLayer.addChild(background);
            hero = new Chara();
            hero.x = 140;
            hero.y = 100;
            hero.hp = hero.maxHp;
            backLayer.addChild(hero);
            stageInit();
            showInit();
            wallInit();
            backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
            backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,mousedown);
            backLayer.addEventListener(LMouseEvent.MOUSE_UP,mouseup);
            if(!LGlobal.canTouch && !restart){
                   LEvent.addEventListener(window,
                          LKeyboardEvent.KEY_DOWN,down);
                   LEvent.addEventListener(window,
                          LKeyboardEvent.KEY_UP,up);
            }
}
```

上面的代码表明，只有当传入 gameStart 的参数为 false 的时候才加载键盘事件。因为在代码清单 7-13 中，传入 gameStart 的参数为 true，所以控制了键盘事件只会加载一遍。

运行上面的代码将得到如图 7-24 所示的游戏效果。

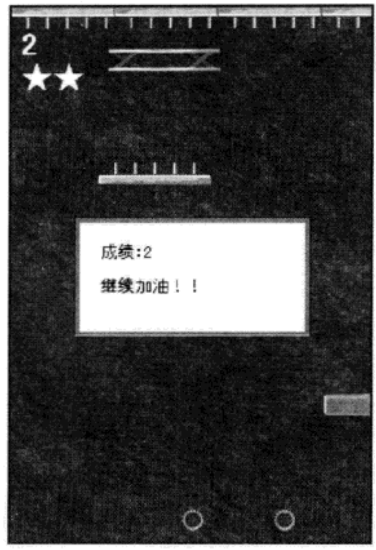


图 7-24 游戏结束效果

7.9 小结

本章除了对之前的一些知识内容进行巩固之外，还带大家认识了如何在游戏中实现卷轴滚动效果。这个游戏虽然简单，但是里面有弹跳、坠落、重力等内容，这些同时也是一般横板过关游戏的组成部分，相信会对想做横板过关游戏的朋友起到一定的借鉴作用。另外，本章进一步让大家了解了如何分别通过触屏事件和键盘事件来控制游戏，实现真正意义上的HTML5 游戏的多平台化。

PDG

第 8 章　开发射击类游戏

本章将讲解如何制作一款简单的射击类游戏。射击类游戏一般分为弹幕类射击游戏和卷轴类射击游戏。弹幕类射击游戏是指大量敌弹以一定的算法有规则地射出并移动，在屏幕上形成“幕状”；卷轴类射击游戏是指背景看起来像是卷轴在滚动一样的射击游戏。上一章我们已经对卷轴游戏做了简单的介绍，此类游戏根据卷轴的方向不同又分为横向和纵向卷轴射击游戏。无论哪类射击游戏，一般都是玩家操控一架飞机，朝敌机射出大量子弹并且不断躲避敌人的子弹。通常，在射击类游戏中，我们会将弹幕类和卷轴类射击游戏结合起来使用，比如“雷电”、“虫姬”等非常经典的射击游戏，它们都包含了弹幕类和卷轴类两种射击游戏的特点，所以这类游戏又称为卷轴弹幕类射击游戏。

接下来，就通过一款简单的射击类游戏的制作，来了解一下此类游戏的制作原理。

8.1　游戏分析

首先，来分析一下所需素材和要素。如图 8-1 所示是预计开发的游戏画面。

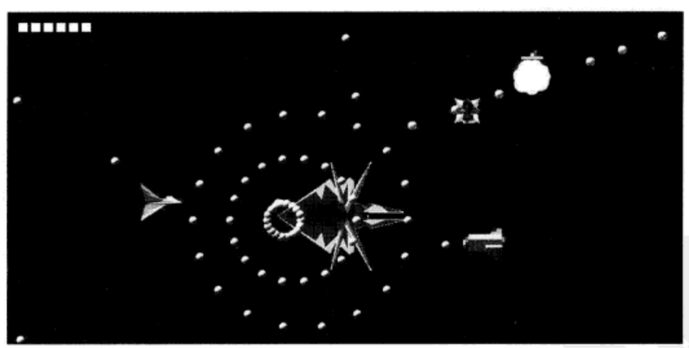


图 8-1　游戏画面预览图

需要掌握的要素有下面几点：

❑ 图片描画

图 8-1 中的飞机和子弹等都是由图片组成的。

❑ 碰撞检测

检测游戏中飞机是否被子弹击中、敌机和我机之间是否发生碰撞。

❑ 循环播放事件

游戏中子弹和飞机的移动都是通过循环播放事件来完成的。

❑ 触屏事件

加入触屏控制，可控制飞机的移动和子弹的发射。

❑ 游戏层次划分

本游戏可以将游戏层次划分为进度条显示层、游戏层、飞机层、子弹层、弹药层和文字层 6 层。进度条显示层用来显示图片读取时的进度。游戏层用来控制整个游戏的显示状态，其他各层都会添加到游戏层上面。飞机层用来显示我方和敌方的飞机。子弹层用来显示游戏中飞机发射的子弹。弹药层用来显示浮动的弹药，我方飞机拿到弹药之后，会改变所发射的子弹种类。

8.2 添加一架可控飞机

因为图片读取、进度条显示等操作在前面的章节已经都有了详细的介绍，所以这部分内容不再重复，直接从添加飞机开始讲起。

8.2.1 添加一个飞机类

通过分析，可以发现无论是敌机还是我机，它们都有一些共同的属性，比如它们都有皮肤，都可以发射子弹等。所以我们需要一个共用的飞机类来控制这些共同的属性。

代码清单 8-1 即为一个完整的飞机类。

代码清单 8-1

```
/**
 * 飞机类
 * */
function Plain(x,y,shootX,shootY,bitmapData,hp){
            base(this,LSprite,[]);
            var self = this;
            //飞机出现位置
            self.x = x;
            self.y = y;
            //炮口相对飞机的相对位置
            self.shootX = shootX;
            self.shootY = shootY;
            //是否射击炮弹
            self.canshoot=false;
            //自动移动控制
            self.move=[0,0];
            //飞机自动移动时的移动速度
            self.speed=1;
            //飞机生命值
            self.hp = hp;
            //飞机是否死亡
            self.isdie=false;
            //将飞机显示到画面上
```

```
            self.bitmap = new LBitmap(bitmapData);
            self.addChild(self.bitmap);
}

/**
 * 循环
 * */
Plain.prototype.onframe = function (){
            var self = this;
            //移动
            self.x += self.move[0]*self.speed;
            self.y += self.move[1]*self.speed;
            //射击
            if(self.canshoot)self.shoot();
};
/**
 * 射击
 * */
Plain.prototype.shoot = function (){};
```

代码解析

首先通过向构造器传入相关的参数，来对飞机的一些属性初始化。

```
Plain(x,y,shootX,shootY,bitmapData,hp)
```

下面详细介绍一下这些属性。

```
//飞机出现位置
self.x = x;
self.y = y;
```

构造器的第 1 个和第 2 个参数用来设定飞机出现的位置坐标，即将飞机添加到游戏画面上时的坐标。

```
//炮口相对飞机的相对位置
self.shootX = shootX;
self.shootY = shootY;
```

构造器的第 3 个和第 4 个参数用来设定炮口相对于飞机的位置，因为飞机实际上是由一张图片构成的，而炮口的位置又是由这张图片来决定的（如敌机的炮口在图片的左边，而我机的炮口在图片的右边），这里是为了控制炮口在游戏中的位置。

```
//是否射击炮弹
self.canshoot=false;
```

这个属性用来控制飞机是否可以发射炮弹，初始化的时候默认不可发射炮弹，当它的值是 true 的时候则表示可以发射炮弹。

```
//自动移动控制
```

```
self.move=[0,0];
```

我机是由玩家来控制的，但是敌机是不受玩家控制、自动移动的，上面这个属性就是用来控制敌机自动移动的。

```
//飞机自动移动时的移动速度
self.speed=1;
```

以上属性用来控制飞机自动移动时的移动速度，值越大飞机移动的速度就越快。

```
//飞机生命值
self.hp = hp;
```

以上属性用来控制飞机的生命值，当生命值变为0的时候，代表飞机已经死亡，可以从屏幕上移除。这个属性由构造器的第6个参数来设定。

```
//飞机是否死亡
self.isdie=false;
```

飞机是否从屏幕上移除，不仅由它的生命值来决定，比如敌机在屏幕上移动时，如果移动到游戏画面之外，也应该将它从屏幕上移除。所以当需要将生命值不为0的飞机从屏幕上移除的时候，就可以将上面这个属性设置为true。

```
//将飞机显示到画面上
self.bitmap = new LBitmap(bitmapData);
self.addChild(self.bitmap);
```

通过构造器传入的第5个参数bitmapData的作用是新建一个LBitmap对象，从而将飞机显示到游戏画面上。

接下来看一下onframe循环函数，代码如下所示：

```
//移动
self.x += self.move[0]*self.speed;
self.y += self.move[1]*self.speed;
```

它通过move和speed属性来控制飞机的移动。

```
//射击
if(self.canshoot)self.shoot();
```

在循环函数中，通过canshoot属性来控制子弹的发射。当canshoot的值是true时，就会不停地调用shoot()函数发射子弹。

```
/**
 * 射击
 * */
Plain.prototype.shoot = function (){};
```

这是飞机的shoot函数，具体如何发射会在后面介绍。

8.2.2 可控飞机类

有了 8.2.1 节的飞机类 Plain，如果再建一个继承自 Plain 的类，它就拥有前面飞机类的所有属性了。

代码清单 8-2 是一个我机类 Player 的完整代码。

代码清单 8-2

```
/**
 * 我机类
 * */
function Player(x,y,shootX,shootY,bitmapData,hp){
    base(this,Plain,[x,y,shootX,shootY,bitmapData,hp]);
    var self = this;
    self.belong = "self";
    self.downX = self.downY = 0;
}
/**
 * 循环
 * */
Player.prototype.onframe = function (){
    var self = this;
    self.callParent("onframe",arguments);
};
```

代码解析

首先要向构造器中传入相关的参数，这是父类构造器初始化时所必须具有的属性。

```
Player (x,y,shootX,shootY,bitmapData,hp)
```

Player 类的构造器中有一些单独的设置，下面来详细介绍。

```
base(this,Plain,[x,y,shootX,shootY,bitmapData,hp]);
```

上面代码调用 base 函数并传入参数，实现对 Plain 类的继承。

```
self.belong = "self";
```

这里为了区分敌机和我机，设置 Player 类的 belong 属性为 self，即我机。

```
self.downX = self.downY = 0;
```

上面这两个属性用来记录鼠标按下时飞机的位置，在控制飞机移动的时候会用到，后面会详细说明。

```
/**
 * 循环
 * */
Player.prototype.onframe = function (){
    var self = this;
    self.callParent("onframe",arguments);
};
```

这里通过调用 callParent 方法来调用父类的 onframe 方法。

接下来看代码清单 8-3，它用来将飞机添加到游戏屏幕上，并且会通过鼠标事件来控制飞机的移动。

代码清单 8-3

```
/**
 * Main
 * */
//设定游戏速度、屏幕大小、回调函数
init(20,"mylegend",800,400,main);

/**层变量*/
//显示进度条所用层
var loadingLayer;
//游戏层、飞机层
var gameLayer,plainLayer;
//图片path数组
var imgData = new Array(
    {name:"player",path:"./images/player.png"}
);
//读取完的图片数组
var imglist;
var player;
var mouseStartX,mouseStartY,mouseNowX,mouseNowY;
var MOVE_STEP = 5;
function main(){
    loadingLayer = new LoadingSample3();
    addChild(loadingLayer);
    LLoadManage.load(
        imgData,
        function(progress){
            loadingLayer.setProgress(progress);
        },
        gameInit
    );
}
function gameInit(result){
    imglist = result;
    removeChild(loadingLayer);
    loadingLayer = null;

    //游戏层初始化
    gameLayer = new LSprite();
    addChild(gameLayer);
    gameLayer.graphics.drawRect(1,"#000000",
            [0,0,800,400],true,"#000000");

    plainLayer = new LSprite();
    gameLayer.addChild(plainLayer);
```

```
    var bitmapData = new LBitmapData(imglist["player"]);
    player = new Player(100,150,
            bitmapData.width, bitmapData.height*0.5,
            bitmapData,30);
    plainLayer.addChild(player);

    gameLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
    gameLayer.addEventListener(LMouseEvent.MOUSE_DOWN,ondown);
    gameLayer.addEventListener(LMouseEvent.MOUSE_MOVE,onmove);
    gameLayer.addEventListener(LMouseEvent.MOUSE_UP,onup);
}
/**
 * 循环
 * */
function onframe(){
    var key;
    for(key in plainLayer.childList){
        plainLayer.childList[key].onframe();
    }

    if(!player.canshoot)return;
    if(player.x - player.downX > mouseNowX - mouseStartX){
        player.x -= MOVE_STEP;
        if(player.x - player.downX < mouseNowX - mouseStartX){
            player.x = mouseNowX - mouseStartX + player.downX;
        }
    }else if(player.x - player.downX < mouseNowX - mouseStartX){
        player.x += MOVE_STEP;
        if(player.x - player.downX > mouseNowX - mouseStartX){
            player.x = mouseNowX - mouseStartX + player.downX;
        }
    }
    if(player.y - player.downY > mouseNowY - mouseStartY){
        player.y -= MOVE_STEP;
        if(player.y - player.downY < mouseNowY - mouseStartY){
            player.y = mouseNowY - mouseStartY + player.downY;
        }
    }else if(player.y - player.downY < mouseNowY - mouseStartY){
        player.y += MOVE_STEP;
        if(player.y - player.downY > mouseNowY - mouseStartY){
            player.y = mouseNowY - mouseStartY + player.downY;
        }
    }
    if(player.x < 0){
        player.x = 0;
        setCoordinate(mouseNowX,mouseNowY);
    }else if(player.x + player.getWidth() > LGlobal.width){
        player.x = LGlobal.width - player.getWidth();
        setCoordinate(mouseNowX,mouseNowY);
    }
```

```
    if(player.y < 0){
        player.y = 0;
        setCoordinate(mouseNowX,mouseNowY);
    }else if(player.y + player.getHeight() > LGlobal.height){
        player.y = LGlobal.height - player.getHeight();
        setCoordinate(mouseNowX,mouseNowY);
    }
}

function ondown(event){
    player.canshoot=true;
    setCoordinate(event.offsetX,event.offsetY);
}
function setCoordinate(x,y){
    mouseStartX = mouseNowX = x;
    mouseStartY = mouseNowY = y;
    player.downX = player.x;
    player.downY = player.y;
}
function onmove(event){
    if(!player.canshoot)return;
    mouseNowX=event.offsetX;
    mouseNowY = event.offsetY;
}
function onup(event){
    player.canshoot=false;
}
```

代码解析

首先看一下游戏的初始化函数 gameInit。

```
//游戏层初始化
gameLayer = new LSprite();
addChild(gameLayer);
gameLayer.graphics.drawRect(1,"#000000",
                    [0,0,800,400],true,"#000000");
```

以上代码是游戏层初始化，并且在游戏层上画一个黑色的矩形，作为游戏的背景。

```
plainLayer = new LSprite();
gameLayer.addChild(plainLayer);
```

以上代码用于初始化飞机层。

```
var bitmapData = new LBitmapData(imglist["player"]);
player = new Player(100,150,
            bitmapData.width, bitmapData.height*0.5, bitmapData,30);
plainLayer.addChild(player);
```

上面代码在坐标 (100,150) 位置上初始化一架飞机，并将其添加到飞机层上。

```
gameLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
gameLayer.addEventListener(LMouseEvent.MOUSE_DOWN,ondown);
gameLayer.addEventListener(LMouseEvent.MOUSE_UP,onup);
```

以上给游戏添加了 4 个相关的事件，即循环事件 onframe、鼠标按下事件 ondown、鼠标滑动事件 onmove、鼠标弹起事件 onup。在具体看这 4 个事件之前，先来看下面的函数 setCoordinate。

```
function setCoordinate(x,y){
            mouseStartX = mouseNowX = x;
            mouseStartY = mouseNowY = y;
            player.downX = player.x;
            player.downY = player.y;
}
```

上面这个函数用来记录鼠标按下时鼠标的当前坐标和飞机的当前坐标，以便为控制飞机移动做准备。

```
function ondown(event){
            player.canshoot=true;
            setCoordinate(event.offsetX,event.offsetY);
}
```

当鼠标按下时调用 ondown 函数，将我机 player 的 canshoot 属性设置为 true，表示开始射击。这个属性也用来控制我机是否可以移动，并且调用 setCoordinate 函数记录下鼠标和飞机当时的坐标。

```
function onup(event){
            player.canshoot=false;
}
```

当鼠标弹起时调用 onup 函数，将我机 player 的 canshoot 属性设置为 false，表示停止射击和移动。

```
function onmove(event){
            if(!player.canshoot)return;
            mouseNowX=event.offsetX;
            mouseNowY = event.offsetY;
}
```

当鼠标在屏幕上移动时调用 onmove 函数，它首先判断我机 player 的 canshoot 属性，如果 canshoot 的值为 false，则此行以下的代码不再执行；如果是 true，表示鼠标处于按下状态，则记录下鼠标当时的坐标。

最后来看一下循环函数 onframe() 中的代码。

```
var key;
for(key in plainLayer.childList){
            plainLayer.childList[key].onframe();
}
```

要循环飞机层上的所有飞机，须调用飞机的循环函数 onframe。不过，现在飞机层上只有一架飞机，以后添加敌机时也会循环所有的敌机。

```
if(!player.canshoot)return;
```

上面代码用于判断我机 player 的 canshoot 属性，当 canshoot 的值为 false，表示鼠标处于弹起状态，则不进行下面的处理；如果是 true，开始操控飞机移动。

```
if(player.x - player.downX > mouseNowX - mouseStartX){
            player.x -= MOVE_STEP;
            if(player.x - player.downX < mouseNowX - mouseStartX){
                  player.x = mouseNowX - mouseStartX + player.downX;
            }
}else if(player.x - player.downX < mouseNowX - mouseStartX){
            player.x += MOVE_STEP;
            if(player.x - player.downX > mouseNowX - mouseStartX){
                  player.x = mouseNowX - mouseStartX + player.downX;
            }
}
if(player.y - player.downY > mouseNowY - mouseStartY){
            player.y -= MOVE_STEP;
            if(player.y - player.downY < mouseNowY - mouseStartY){
                  player.y = mouseNowY - mouseStartY + player.downY;
            }
}else if(player.y - player.downY < mouseNowY - mouseStartY){
            player.y += MOVE_STEP;
            if(player.y - player.downY > mouseNowY - mouseStartY){
                  player.y = mouseNowY - mouseStartY + player.downY;
            }
}
```

前面提到，当鼠标按下时，已经通过 setCoordinate 函数分别记录下了鼠标的坐标（mouseStartX, mouseStartY）和飞机的坐标（downX, downY），上面的代码将鼠标的当前坐标和鼠标按下时的坐标（mouseStartX, mouseStartY）进行比较，来计算飞机相对于（downX, downY）的坐标值（x', y'），然后改变飞机现在的坐标值，向坐标（x', y'）移动。

```
if(player.x < 0){
            player.x = 0;
            setCoordinate(mouseNowX,mouseNowY);
}else if(player.x + player.getWidth() > LGlobal.width){
            player.x = LGlobal.width - player.getWidth();
            setCoordinate(mouseNowX,mouseNowY);
}
if(player.y < 0){
            player.y = 0;
            setCoordinate(mouseNowX,mouseNowY);
}else if(player.y + player.getHeight() > LGlobal.height){
            player.y = LGlobal.height - player.getHeight();
            setCoordinate(mouseNowX,mouseNowY);
}
```

上面代码表示当飞机移到屏幕外面的时候，停止移动飞机。

运行代码，会得到如图 8-2 所示的效果。

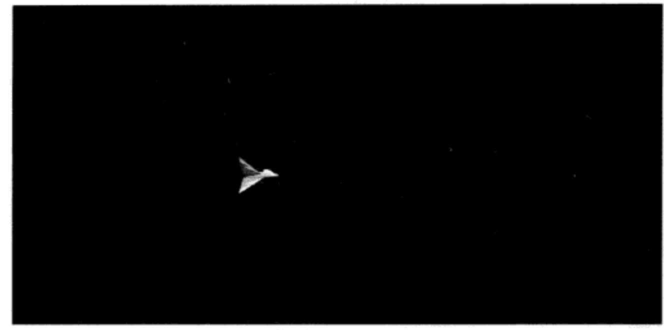


图 8-2　添加飞机运行效果图

通过在屏幕上滑动鼠标，已经可以控制飞机的移动了。

8.3　为飞机添加多样化的子弹

射击游戏的灵魂在于多种多样的子弹，本节就来为飞机添加多样化的子弹。

8.3.1　建立一个子弹类

为了便于控制子弹，首先建立一个子弹类，如代码清单 8-4 所示。

代码清单　8-4

```
/**
 * 子弹类
 * */
function Bullet(params){
            base(this,LSprite,[]);
            var self = this;
            //出现位置
            self.x = params.x;
            self.y = params.y;
            //xy 轴速度
            self.xspeed = params.xspeed;
            self.yspeed = params.yspeed;
            self.belong = params.belong;
            self.isdie = false;
            //子弹图片
            self.bitmap = new LBitmap(params.bitmapData);
            //显示
            self.addChild(self.bitmap);
}

/**
```

```
  * 循环
  * */
Bullet.prototype.onframe = function (){
        var self = this;
        //子弹移动
        self.x += self.xspeed;
        self.y += self.yspeed;
        //子弹位置检测
        if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y >
LGlobal.height){
            //从屏幕移除
            bulletLayer.removeChild(self);
        }
};
```

代码解析

首先看一下子弹类的构造器 Bullet。

```
base(this,LSprite,[]);
```

上面代码实现了对 LSprite 类的继承。

下面再来设定子弹在屏幕上出现的位置坐标。

```
//出现位置
self.x = params.x;
self.y = params.y;
```

然后给子弹设置一个速度，来控制子弹移动。

```
//xy 轴速度
self.xspeed = params.xspeed;
self.yspeed = params.yspeed;
```

接着给子弹设置属性，这是为了区别子弹是由敌机还是我机发射出来的。

```
self.belong = params.belong;
```

下面代码用于确定子弹是否可以从屏幕上移除。

```
self.isdie = false;
```

下面添加子弹图片，用来显示子弹。

```
//子弹图片
self.bitmap = new LBitmap(params.bitmapData);
//显示
self.addChild(self.bitmap);
```

下面来看一下子弹类的循环函数 onframe。

```
//子弹移动
```

```
self.x += self.xspeed;
self.y += self.yspeed;
```

上面代码根据子弹在 x 轴和 y 轴上的速度，来改变子弹的坐标，从而实现子弹的移动。以下代码用于判断子弹的位置，当子弹移动到游戏屏幕之外的时候，将子弹移除。

```
//子弹位置检测
if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y > LGlobal.height){
            //从屏幕移除
            bulletLayer.removeChild(self);
}
```

8.3.2 单发子弹

为了让飞机能够发射子弹，首先给飞机类 Plain 添加一个 setBullet 方法，用来给飞机设定子弹，代码如下所示：

```
Plain.prototype.setBullet = function (bulletIndex){
             this.bullet=bulletIndex;
};
```

然后修改飞机类 Plain 的射击函数 shoot，如代码清单 8-5 所示。

代码清单 8-5

```
/**
 * 射击
 * */
Plain.prototype.shoot = function (){
    var self = this;
    var bullet = bulletList[self.bullet];
    if(self.shoopIndex++ < bullet.step)return;
    self.shoopIndex=0;
    //开始发射
    for(i=0;i<bullet.count;i++){
        //发射角度
        var angle = i*bullet.angle + bullet.startAngle;
        //子弹xy轴速度
        xspeed = bullet.speed*Math.cos(angle * Math.PI / 180);
        yspeed = bullet.speed*Math.sin(angle * Math.PI / 180);
        var params = {
            bitmapData:self.bulletBitmapData,
            x:self.x+self.shootX,
            y:self.y+self.shootY,
            xspeed:xspeed,
            yspeed:yspeed,
            belong:self.belong};
        //子弹实例化
        obj = new Bullet(params);
        //显示
```

```
            bulletLayer.addChild(obj);
        }
    };
```

代码解析

```
var bullet = bulletList[self.bullet];
```

上面代码用于取得子弹类型，其中 bulletList 是预先在 Main 类中准备好的一个子弹类型数组，它里面包含了子弹必需的属性，具体在后面解释。

```
if(self.shoopIndex++ < bullet.step)return;
self.shoopIndex=0;
```

以上代码用来控制子弹发射的频率快慢。

```
for(i=0;i<bullet.count;i++){
```

在上面的 for 循环中，bullet.count 是子弹每次发射的数量。

```
//发射角度
var angle = i*bullet.angle + bullet.startAngle;
```

以上代码用于计算子弹发射时相对于水平线的角度。

```
//子弹 xy 轴速度
xspeed = bullet.speed*Math.cos(angle * Math.PI / 180);
yspeed = bullet.speed*Math.sin(angle * Math.PI / 180);
```

上面代码计算子弹在 x 轴和 y 轴上的移动速度。

```
var params = {
            bitmapData:self.bulletBitmapData,
            x:self.x+self.shootX,
            y:self.y+self.shootY,
            xspeed:xspeed,
            yspeed:yspeed,
            belong:self.belong};
//子弹实例化
obj = new Bullet(params);
//显示
bulletLayer.addChild(obj);
```

这里实例化一个子弹，并将其添加到子弹层 bulletLayer 上。

再来看看如何修改 Player 类的构造器，给我机类设定子弹图片，代码如下所示：

```
/**
 * 我机类
 * */
function Player(x,y,shootX,shootY,bitmapData,hp){
    base(this,Plain,[x,y,shootX,shootY,bitmapData,hp]);
```

```
    var self = this;
    self.belong = "self";
    self.downX = self.downY = 0;
    self.bulletBitmapData=new LBitmapData(imglist["bullet01"]);
}
```

下面修改 Main.js，添加子弹数组 bulletList，代码如下所示：

```
var bulletList = new Array(
            {startAngle:0,angle:0,step:10,speed:5,count:1}//1 发
);
```

在数组的子对象中，属性依次为：子弹射击的初始角度，每次发射多颗子弹时每颗子弹的相对角度，子弹的发射频率，子弹的速度，每次发射子弹的数量。上面的代码表示每次发射一发子弹，子弹的初始角度为 0，速度为 5，子弹的发射频率为 10。

在实例化飞机的时候，可调用飞机类的 setBullet 函数，为飞机添加子弹，代码如下所示：

```
player = new Player(100,150,
            bitmapData.width,bitmapData.height*0.5,
            bitmapData,30);
plainLayer.addChild(player);
player.setBullet(0);
```

运行代码，得到的效果如图 8-3 所示。

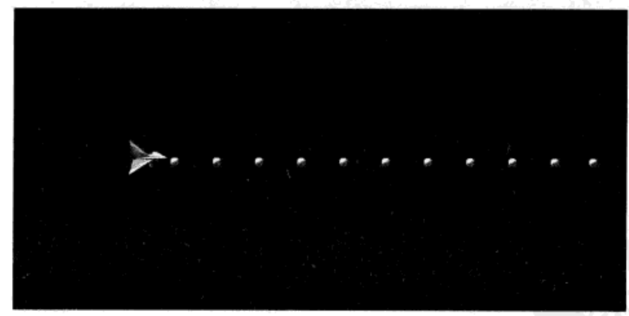


图 8-3　发射子弹运行效果图

8.3.3　多发子弹

有了上面的子弹数组，发射多个子弹也比较简单了，只需修改子弹数组 bulletList 即可，代码如下所示：

```
var bulletList = new Array(
            {startAngle:0,angle:0,step:10,speed:5,count:1},//1 发
            {startAngle:-20,angle:20,step:10,speed:5,count:3},//3 发
            {startAngle:-40,angle:20,step:10,speed:5,count:5} //5 发
);
```

数组的第 2 个元素表示子弹发射的初始角度为 -20，而每颗子弹的相对角度为 20，这样每次发射 3 颗子弹的话，子弹在屏幕上就会形成对称的图形。每次发射 5 颗子弹的原理也一样。在飞机初始化时，给飞机设定不同的子弹，运行代码得到的效果如图 8-4 和图 8-5 所示。

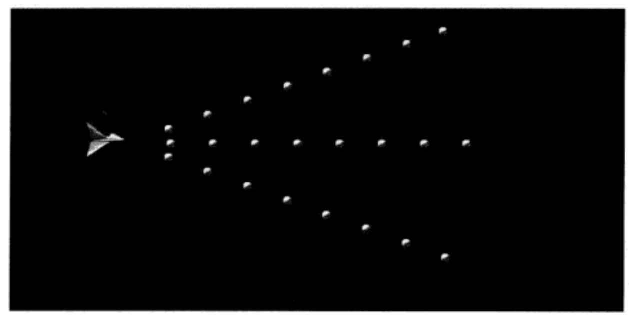


图 8-4　3 颗子弹运行效果图

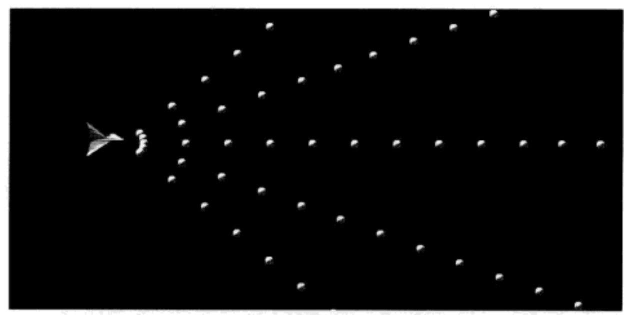


图 8-5　5 颗子弹运行效果图

8.3.4 环形子弹

根据 8.3.3 节介绍的子弹发射原理计算子弹的角度，并修改子弹数组 bulletList，即可得到环形子弹，代码如下所示：

```
var bulletList = new Array(
            {startAngle:0,angle:0,step:10,speed:5,count:1},//1 发
            {startAngle:-20,angle:20,step:10,speed:5,count:3},//3 发
            {startAngle:-40,angle:20,step:10,speed:5,count:5} ,//5 发
            {startAngle:0,angle:20,step:10,speed:5,count:18}// 环发
);
```

因为子弹的初始角度为 0，每颗子弹的相对角度为 20，这样一次性发射 18 颗子弹就会形成一个环形，运行代码得到的效果如图 8-6 所示。

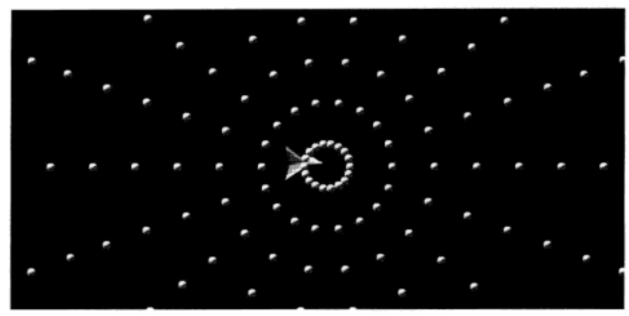


图 8-6　环形子弹运行效果图

8.3.5　反向子弹

要让子弹反向发射，主要是要改变子弹发射的初始角度，通过修改子弹数组 bulletList 可以实现，代码如下所示：

```
var bulletList = new Array(
            {startAngle:0,angle:20,step:10,speed:5,count:1},//1 发
            {startAngle:-20,angle:20,step:10,speed:5,count:3},//3 发
            {startAngle:-40,angle:20,step:10,speed:5,count:5},//5 发
            {startAngle:0,angle:20,step:10,speed:5,count:18},// 环发
            {startAngle:180,angle:20,step:50,speed:5,count:1},//1 发
            {startAngle:160,angle:20,step:50,speed:5,count:3},//3 发
            {startAngle:140,angle:20,step:50,speed:5,count:5}//5 发
);
```

上面子弹数组中最后 3 个元素的子弹初始角度都超过 90 度，在屏幕上的效果就是反白发射，运行代码得到的效果如图 8-7 所示。

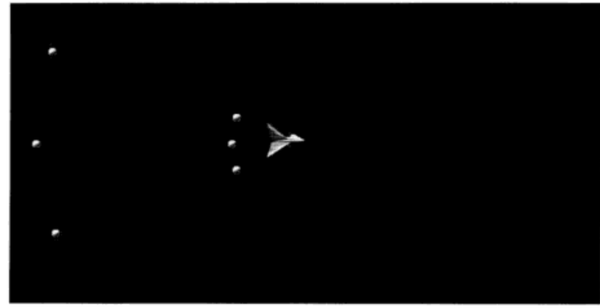


图 8-7　反向子弹运行效果图

8.4　添加敌机

前面已经给游戏添加了我方飞机，并且实现了飞机发射子弹的控制方法，下面将给游戏

添加敌方飞机，做好战斗准备。

8.4.1 建立一个敌机类

首先建立一个敌机类，如代码清单 8-6 所示。

代码清单 8-6

```
/**
 * 敌机类
 * */
function Enemy(x,y,shootX,shootY,bitmapData,hp){
            base(this,Plain,[x,y,shootX,shootY,bitmapData,hp]);
            var self = this;
            self.belong = "enemy";
            self.bulletBitmapData=new LBitmapData(imglist["bullet02"]);
}

/**
 * 循环
 * */
Enemy.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            var isOut = false;
            if(self.x < -self.getWidth() || self.x > LGlobal.width ||
                   self.y < -self.getHeight() || self.y > LGlobal.height){
                   isOut = true;
            }
            if(isOut)self.whenOut();
            if(self.isdie || self.hp <= 0){
                   plainLayer.removeChild(self);
            }
};

Enemy.prototype.whenOut = function (){
            var self = this;
            if(self.move.length > 0)self.move.splice(0,2);
            if(self.move.length == 0)self.isdie = true;
};
```

代码解析

因为已经建好了飞机类 Plain，所以只需要让 Enemy 类继承自 Plain 类，它就能够拥有飞机的属性——可以飞，可以发射子弹。

先来看一下构造器中的代码，如下所示：

```
base(this,Plain,[x,y,shootX,shootY,bitmapData,hp]);
```

以上代码是让 Enemy 类继承自 Plain 类。

在代码清单 8-2 中，为了区分敌机和我机，设置 Player 类的所属为 self。下面代码设置 Enemy 类的所属为 enemy，这样这两个所属值就分别代表了我机和敌机。

```
self.belong = "enemy";
```

以下代码与代码清单 8-2 中的作用相同，即给敌机设定子弹图片。可以使用相同的子弹图片，也可以使用不同的子弹图片，从而区分我机和敌机射出的子弹。

```
self.bulletBitmapData=new LBitmapData(imglist["bullet02"]);
```

再来看敌机的循环函数 onframe。

```
self.callParent("onframe",arguments);
```

这里调用父类，也就是 Plain 类的循环函数 onframe，来完成父类的相关动作。

下面代码根据敌机的坐标来判断敌机是否移动出游戏屏幕。

```
var isOut = false;
if(self.x < -self.getWidth() || self.x > LGlobal.width ||
            self.y < -self.getHeight() || self.y > LGlobal.height){
            isOut = true;
}
```

如果敌机移出了游戏屏幕，则调用敌机的 whenOut 函数来执行一些相关的处理，代码如下所示：

```
if(isOut)self.whenOut();
```

当敌机死亡后，则将敌机从 plainLayer 层中移除。

```
if(self.isdie || self.hp <= 0){
            plainLayer.removeChild(self);
}
```

下面是敌机的 whenOut 函数。

```
Enemy.prototype.whenOut = function (){
            var self = this;
            if(self.move.length > 0)self.move.splice(0,2);
            if(self.move.length == 0)self.isdie = true;
};
```

从代码清单 8-1 中得知，飞机的自动移动是由它的 move 属性来控制的，这个 move 属性的值是一个数组，数组的前两个元素决定了飞机自动移动的 x 轴和 y 轴的速度。所以如果想在飞机移动的过程中控制它，只要改变 move 数组中前两个元素的值就可以了。上面的 whenOut 函数的功能是，当敌机移出游戏屏幕时，将敌机 move 数组中的前两个元素删除。如果改变后的 move 数组的长度是 0，则将敌机的 isdie 属性设置为 true，表示可以从画面中移除；如果不为 0，则敌机就会根据 move 数组中前两个新元素的值来进行移动。这样也就实现了利用敌机的 move 数组来控制敌机移动的效果。

接着修改 Main.js，添加敌机以及控制敌机出现的事件。这里添加相关变量 ctrlList，代码如下所示：

```
var ctrlIndex = 0;
var ctrlList=[
{"frames":10,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":15,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":20,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":25,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":30,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false}
];
var frame = 0;
var frames = 0;
```

ctrlList 数组中的元素是游戏中会出现在画面中的敌机，其中 frames 用来控制敌机出现的时间，bullet 表示敌机的子弹在子弹数组中的序号，move 就是上面提到的控制敌机移动的数组，img 是敌机的图片，x 表示敌机出现时的 x 轴坐标，y 表示敌机出现时的 y 轴坐标，hp 表示敌机的生命值，最后的 isboss 用来判断出现的敌机是否是敌方的 Boss 飞机。

然后给 Main.js 添加 setObject 函数，用于添加敌机，如代码清单 8-7 所示。

代码清单 8-7

```
function setObject(){
            if(frame++ < 10)return;
            frame = 0;
            frames++;
            var ctrlObject = ctrlList[ctrlIndex];
            if(ctrlObject["frames"] == frames){
                   ctrlIndex++;

                   bitmapData = new LBitmapData(imglist[ctrlObject.img]);
                   var enemy;
                   if(ctrlObject.isboss){
                          enemy = new Boss(ctrlObject.x,ctrlObject.y,0,
bitmapData.height*0.5,bitmapData,ctrlObject["hp"]);
                   }else{
                          enemy = new Enemy(ctrlObject.x,ctrlObject.y,0,
bitmapData.height*0.5,bitmapData,ctrlObject["hp"]);
                   }
                   plainLayer.addChild(enemy);
                   enemy.setBullet(ctrlObject.bullet);
                   enemy.move = ctrlObject.move;
                   enemy.canshoot=true;
            }
}
```

代码解析

```
if(frame++ < 10)return;
```

```
frame = 0;
frames++;
```

上面的代码用来控制添加敌机的频率。

```
var ctrlObject = ctrlList[ctrlIndex]
```

上面代码表示取得要添加的敌机的属性，ctrlIndex 表示要添加的敌机在 ctrlList 数组中的序号。

```
if(ctrlObject["frames"] == frames)
```

上面判定表示，只有当 frames 与当前取得的敌机出现的时间相一致，才能进行添加敌机的操作。

```
ctrlIndex++;
```

因为 ctrlList 数组中的敌机是按照顺序依次添加的，所以每添加一架敌机，就需要将 ctrlIndex 的值加 1，让它指向 ctrlList 数组的下一个元素。

```
bitmapData = new LBitmapData(imglist[ctrlObject.img]);
var enemy;
enemy = new Enemy(ctrlObject.x,ctrlObject.y,
        0, bitmapData.height*0.5,bitmapData,ctrlObject["hp"]);
plainLayer.addChild(enemy);
```

上面代码表示，根据取得的敌机属性，向 plainLayer 层上添加一架敌机。

```
enemy.setBullet(ctrlObject.bullet);
enemy.move = ctrlObject.move;
enemy.canshoot=true;
```

上面代码给添加的敌机设定子弹和移动数组，并且设定敌机的 canshoot 属性为 true，表示可以进行射击。

最后在 Main.js 的循环函数 onframe 中调用 setObject() 函数进行添加敌机的操作，然后运行代码，得到的效果如图 8-8 所示。

图 8-8 添加敌机运行效果图

8.4.2 建立一个敌机 Boss 类

敌方的 Boss 不同于一般的敌机，通常生命值都比较高，发射的子弹也比较难于躲避。代码清单 8-8 建立了一个敌机的 Boss。

代码清单 8-8

```
/**
 * 敌机 Boss 类
 * */
function Boss(x,y,shootX,shootY,bitmapData,hp){
            base(this,Enemy,[x,y,shootX,shootY,bitmapData,hp]);
            this.shootIndex = 0;
}

/**
 * 循环
 * */
Boss.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            if(self.isdie || self.hp <= 0)gameClear();
};
Boss.prototype.whenOut = function (){
            var self = this;
            if(self.x < 400){
                    self.move[0]=1;
                    self.move[1]=Math.random()>0.5?1:-1;
            }else{
                    self.move[0]=-1;
                    self.move[1]=Math.random()>0.5?1:-1;
            }
};
/**
 * 射击
 * */
Boss.prototype.shoot = function (){
            var self = this;
            self.shootIndex++;
            if(self.shootIndex>100 && self.shootIndex < 150){
                    return;
            }else if(self.shootIndex >= 150){
                    self.shootIndex = 0;
            }
            self.callParent("shoot",arguments);
};
```

代码解析

可以看到，这个敌机 Boss 继承于 Enemy，所以它不但拥有飞机 Plain 的所有属性，同时

还具有一般敌机 Enemy 的所有属性。

敌机 Boss 还有一些特殊的设定，下面来分析一下。首先看循环函数 onframe。

```
/**
 * 循环
 * */
Boss.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);
            if(self.isdie || self.hp <= 0)gameClear();
};
```

由于本章制作的射击游戏只是为了讲解射击游戏的制作原理，所以只设定一个关卡，当敌机 Boss 死亡时，则游戏通关。在敌机 Boss 的生命值为 0 后，则会调用 Main.js 的 gameClear 函数，显示游戏通关。

再来看敌机 Boss 的 whenOut 函数。

```
Boss.prototype.whenOut = function (){
            var self = this;
            if(self.x < 400){
                   self.move[0]=1;
                   self.move[1]=Math.random()>0.5?1:-1;
            }else{
                   self.move[0]=-1;
                   self.move[1]=Math.random()>0.5?1:-1;
            }
};
```

敌机在 move 数组的长度变为 0 之后，就会从游戏画面上移除。而从上面的代码可以看到，敌机 Boss 移出屏幕的时候，会对 move 数组的前两个元素的值进行重新复制，而不是删除前两个元素，所以敌机 Boss 永远不会移出游戏画面，除非它的生命值 hp 的值变为 0。

再来看敌机 Boss 的 shoot 函数。

```
/**
 * 射击
 * */
Boss.prototype.shoot = function (){
            var self = this;
            self.shootIndex++;
            if(self.shootIndex>100 && self.shootIndex < 150){
                   return;
            }else if(self.shootIndex >= 150){
                   self.shootIndex = 0;
            }
            self.callParent("shoot",arguments);
};
```

因为为敌机 Boss 设定的子弹种类是每次发射 18 颗的环形子弹，所以如果连续发射，我

机将无法躲避，所以在敌机 Boss 发射子弹的过程中要加入一定时间的间隔。

为了让敌机 Boss 出现，修改敌机数组 ctrlList，代码如下所示：

```
var ctrlList=[
{"frames":10,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":15,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":20,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":25,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":30,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":50,"bullet":5,"move":[0,-1,-1,1],img:"enemy2",x:600,y:400,hp:5,isboss:false},
{"frames":55,"bullet":5,"move":[0,-1,-1,1],img:"enemy2",x:600,y:400,hp:5,isboss:false},
{"frames":60,"bullet":5,"move":[0,-1,-1,1],img:"enemy2",x:600,y:400,hp:5,isboss:false},
{"frames":65,"bullet":5,"move":[0,-1,-1,1],img:"enemy2",x:600,y:400,hp:5,isboss:false},
{"frames":70,"bullet":5,"move":[0,-1,-1,1],img:"enemy2",x:600,y:400,hp:5,isboss:false},
{"frames":90,"bullet":4,"move":[-1,-1,-1,1],img:"enemy1",x:800,y:400,hp:3,isboss:false},
{"frames":95,"bullet":4,"move":[-1,-1,-1,1],img:"enemy1",x:800,y:400,hp:3,isboss:false},
{"frames":100,"bullet":4,"move":[-1,-1,-1,1],img:"enemy1",x:800,y:400,hp:3,isboss:false},
{"frames":105,"bullet":4,"move":[-1,-1,-1,1],img:"enemy1",x:800,y:400,hp:3,isboss:false},
{"frames":110,"bullet":4,"move":[-1,-1,-1,1],img:"enemy1",x:800,y:400,hp:3,isboss:false},
{"frames":130,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":135,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":140,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":145,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":150,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":180,"bullet":3,"move":[-1,0],img:"enemy3",x:800,y:180,hp:100,isboss:true},
{"frames":200,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":220,"bullet":5,"move":[0,1,-1,-1],img:"enemy2",x:600,y:0,hp:5,isboss:false},
{"frames":230,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false},
{"frames":250,"bullet":4,"move":[-1,1,-1,-1],img:"enemy1",x:800,y:0,hp:3,isboss:false}
];
```

可以看到第 21 个元素的 isboss 属性为 true，表示加入的敌机是敌方的特殊飞机 Boss。

因为 setObject 函数中并没有对 isboss 进行判断，所以修改 setObject 函数中添加敌机部分的代码，如下所示：

```
var enemy;
if(ctrlObject.isboss){
            enemy = new Boss(ctrlObject.x,ctrlObject.y,
            0,bitmapData.height*0.5,bitmapData,ctrlObject["hp"]);
}else{
            enemy = new Enemy(ctrlObject.x,ctrlObject.y,
            0,bitmapData.height*0.5,bitmapData,ctrlObject["hp"]);
}
```

这里通过 isboss 来判断添加普通敌机 Enemy 还是添加敌机 Boss。

运行代码，效果如图 8-9 所示。

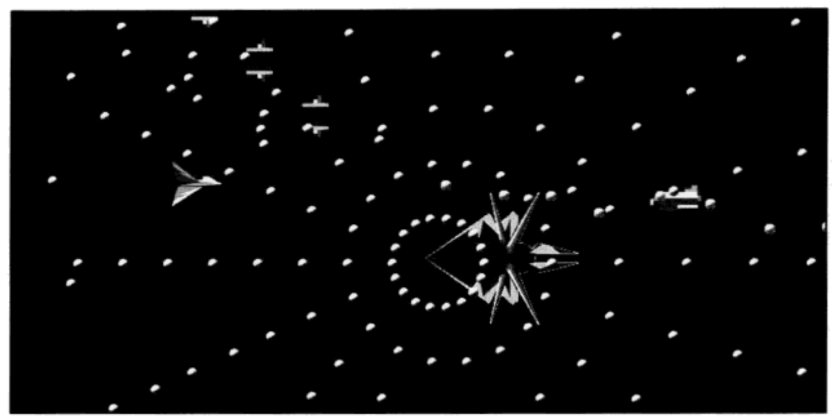


图 8-9　添加 Boss 敌机效果图

8.5　碰撞检测

目前已经为游戏添加了各种飞机和子弹，但是飞机与飞机之间、飞机与子弹之间并没有发生碰撞。本节主要来实现这些碰撞的检测，显示子弹击中飞机的效果。

8.5.1　飞机与子弹的碰撞

要检测屏幕上所有子弹是否与飞机发生碰撞，就需要对子弹进行一一检测。由于每颗子弹都会持续调用子弹的循环函数，因此在子弹的循环函数中进行碰撞检测是最合理的做法。代码清单 8-9 对子弹类进行了相关修改。

代码清单　8-9

```
/**
 * 子弹类
 * */
function Bullet(params){
    base(this,LSprite,[]);
    var self = this;
    //出现位置
    self.x = params.x;
    self.y = params.y;
    //xy 轴速度
    self.xspeed = params.xspeed;
    self.yspeed = params.yspeed;
    self.belong = params.belong;
    self.isdie = false;
    //子弹图片
    self.bitmap = new LBitmap(params.bitmapData);
    //显示
```

```
        self.addChild(self.bitmap);
    }

    /**
     * 循环
     * */
    Bullet.prototype.onframe = function (){
        var self = this;
        if(self.isdie){
            self.removeRun();
            return;
        }
        //子弹移动
        self.x += self.xspeed;
        self.y += self.yspeed;
        //子弹位置检测
        if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y > LGlobal.height){
            //从屏幕移除
            bulletLayer.removeChild(self);
        }
        var key,plain;
        if(self.belong == player.belong){
            for(key in plainLayer.childList){
                plain = plainLayer.childList[key];
                if(player.objectindex != plain.objectindex && LGlobal.hitTestArc(self,plain)){
                    plain.hp--;
                    self.isdie=true;
                    self.bitmap.bitmapData = new LBitmapData(imglist["remove"]);
                    self.bitmap.x = -self.bitmap.getWidth() * 0.5;
                    self.bitmap.y = -self.bitmap.getHeight() * 0.5;
                }
            }
        }else{
            if(LGlobal.hitTestArc(self,player)){
                player.hp--;
                self.isdie=true;
                self.bitmap.bitmapData = new LBitmapData(imglist["remove"]);
            }
        }
    };

    Bullet.prototype.removeRun = function (){
        var self = this;
        if(self.alpha <= 0){
            bulletLayer.removeChild(self);
            return;
        }
        self.bitmap.scaleX+=0.1;
        self.bitmap.scaleY+=0.1;
        self.bitmap.x = -self.bitmap.getWidth() * 0.5;
        self.bitmap.y = -self.bitmap.getHeight() * 0.5;
```

```
        self.alpha-=0.1;
    }
```

代码解析

先来对比一下代码清单 8-9 和代码清单 8-4 中子弹的循环函数 onframe，有一些不同之处。代码清单 8-9 中追加了两部分内容，下面就解释一下追加部分代码的含义。

追加部分一：

```
if(self.isdie){
            self.removeRun();
            return;
}
```

代码清单 8-9 中的循环函数一开始就加入了 isdie 的判断，如果 isdie 的值是 true，则调用子弹的 removeRun() 函数（后面会对 removeRun() 函数的功能进行详细解释）。

追加部分二：

```
var key,plain;
if(self.belong == player.belong){
    for(key in gameLayer.childList){
        plain = gameLayer.childList[key];
        if(player.objectindex != plain.objectindex && LGlobal.hitTestArc(self,plain)){
            plain.hp--;
            self.isdie=true;
            self.bitmap.bitmapData = new LBitmapData(imglist["remove"]);
            self.bitmap.x = -self.bitmap.getWidth() * 0.5;
            self.bitmap.y = -self.bitmap.getHeight() * 0.5;
        }
    }
}else{
    if(LGlobal.hitTestArc(self,player)){
        player.hp--;
        self.isdie=true;
        self.bitmap.bitmapData = new LBitmapData(imglist["remove"]);
        self.bitmap.x = -self.bitmap.getWidth() * 0.5;
        self.bitmap.y = -self.bitmap.getHeight() * 0.5;
    }
}
```

上面的代码是子弹与飞机之间碰撞的检测。首先根据子弹的属性，来判断应该与我机还是敌机进行碰撞检测。无论是哪种检测，都会用到 LGlobal.hitTestArc 函数，这个函数可以用来检测两个对象是否发生了圆形碰撞。

在 lufylegend 库件中，有矩形碰撞和圆形碰撞两种检测。矩形碰撞可以使用 LGlobal.hitTestRect，这个碰撞是检测由两个对象的长和宽所决定的矩形是否发生重叠，如有重叠则代表发生了碰撞。圆形碰撞可以使用 LGlobal.hitTestArc，这个碰撞检测是检测以两个对象的宽为直径所决定的圆是否发生了重叠，如有重叠则表示发生了碰撞。

代码清单 8-9 中使用圆形碰撞进行碰撞检测，如果发生碰撞，则将与子弹发生碰撞的飞机的生命值 hp 减去 1，并将子弹的 isdie 属性设置为 true。

因为子弹与飞机发生碰撞后会发生爆炸，所以代码清单 8-9 中没有将子弹直接移除，而是将它的图片改为爆炸状态的图片了。

最后，来看一下子弹类的 removeRun 函数。

```
Bullet.prototype.removeRun = function (){
            var self = this;
            if(self.alpha <= 0){
                   bulletLayer.removeChild(self);
                   return;
            }
            self.bitmap.scaleX+=0.1;
            self.bitmap.scaleY+=0.1;
            self.bitmap.x = -self.bitmap.getWidth() * 0.5;
            self.bitmap.y = -self.bitmap.getHeight() * 0.5;
            self.alpha-=0.1;
}
```

图 8-10 游戏素材（爆炸效果）

removeRun 函数主要实现子弹的爆炸效果，爆炸图片如图 8-10 所示。

removeRun 函数会将爆炸图片逐渐变大，并且逐渐变淡，当图片的透明度变为 0 的时候，将爆炸图从子弹层 bulletLayer 中移除。

运行代码，得到的效果如图 8-11 所示。

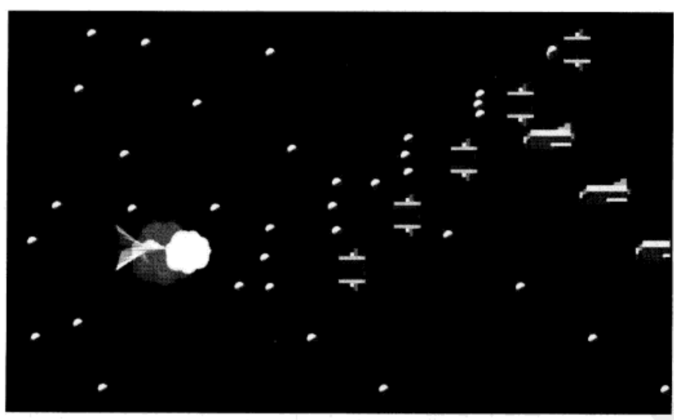


图 8-11 子弹与飞机碰撞效果图

8.5.2 我机与敌机的碰撞

由于我机需要与屏幕上所有的敌机进行碰撞检测，因此可以在敌机的循环函数 onframe 中进行碰撞检测。代码清单 8-10 修改了敌机的循环函数。

代码清单 8-10

```
/**
 * 循环
 * */
Enemy.prototype.onframe = function (){
	var self = this;
	self.callParent("onframe",arguments);
	var isOut = false;
	if(self.x < -self.getWidth() || self.x > LGlobal.width ||
		self.y < -self.getHeight() || self.y > LGlobal.height){
		isOut = true;
	}
	if(isOut)self.whenOut();
	if(self.isdie || self.hp <= 0){
		plainLayer.removeChild(self);
	}else if(LGlobal.hitTestArc(self,player)){
		player.hp--;
		if(player.x < self.x){
			player.x -= player.getWidth();
		}else{
			player.x += player.getWidth();
		}
	}
};
```

代码解析

与代码清单 8-6 中的循环函数对比可以看到，代码清单 8-10 在函数的最后添加了如下的碰撞检测代码。

```
if(self.isdie || self.hp <= 0){
	plainLayer.removeChild(self);
}else if(LGlobal.hitTestArc(self,player)){
	player.hp--;
	if(player.x < self.x){
		player.x -= player.getWidth();
	}else{
		player.x += player.getWidth();
	}
}
```

上面代码表示，如果敌机没有死亡，则与我机进行碰撞检测。如果与我机发生碰撞，则将我机的生命值减去 1，然后我机退后或者向前一个步长来避开连续进行碰撞。

运行代码，将会发现敌机和我机之间已经可以发生碰撞了。

8.6 子弹的变更

在射击游戏中，玩家的飞机所发射出的子弹，通常不是固定的一种，玩家拾起不同的弹

药，就会变换不同的子弹，这样游戏才显得更加精彩。

8.6.1 建立一个弹药类

弹药同样有自己的皮肤，可以在游戏画面中移动，所以也需要构建一个弹药类来管理弹药，如代码清单 8-11 所示。

代码清单 8-11

```
/**
 * 弹药类
 * */
function BulletCtrl(params){
            base(this,LSprite,[]);
            var self = this;
            //出现位置
            self.x = params.x;
            self.y = params.y;
            //xy 轴速度
            self.xspeed = params.xspeed;
            self.yspeed = params.yspeed;
            self.bulletIndex = params.bulletIndex;
            self.bitmap = new LBitmap(params.bitmapData);
            //显示
            self.addChild(self.bitmap);
}

/**
 * 循环
 * */
BulletCtrl.prototype.onframe = function (){
            var self = this;
            //移动
            self.x += self.xspeed;
            self.y += self.yspeed;
            //位置检测
            if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y >
LGlobal.height){
                    //从屏幕移除
                    bulletCtrlLayer.removeChild(self);
            }

};
```

代码解析

先来看弹药类的构造器。

```
base(this,LSprite,[]);
```

它实现了对 LSprite 类的继承。

以下代码设定了弹药在屏幕中出现时的坐标位置。

```
//出现位置
self.x = params.x;
self.y = params.y;
```

以下代码设定了弹药在 x 轴和 y 轴移动的速度。

```
//xy 轴速度
self.xspeed = params.xspeed;
self.yspeed = params.yspeed;
```

下面代码设定弹药的子弹序号，当我机与弹药发生碰撞的时候，我机的子弹会根据这个序号而变化。

```
self.bulletIndex = params.bulletIndex;
```

下面设定弹药的皮肤，即给弹药添加一张图片。

```
self.bitmap = new LBitmap(params.bitmapData);
//显示
self.addChild(self.bitmap);
```

下面是弹药的循环函数 onframe。

```
//移动
self.x += self.xspeed;
self.y += self.yspeed;
```

这里表示让弹药根据在 x 轴和 y 轴移动的速度进行移动。

下面代码表示弹药移动到屏幕之外后，将其从画面中移除。

```
//位置检测
if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y > LGlobal.height){
            //从屏幕移除
            bulletLayer.removeChild(self);
}
```

8.6.2　弹药与我机的碰撞

8.6.1 节中已添加了一个弹药类，下面就将弹药添加到游戏中。修改 Main.js 的 setObject 函数，如代码清单 8-12 所示。

代码清单　8-12

```
function setObject(){
    if(frame++ < 10)return;
    frame = 0;
    frames++;
    if(frames % 50 == 0){
        var bulletIndex = Math.random()>0.5?1:2;
```

```
            var obj = new BulletCtrl({
                x:790,y:100+Math.random()*200,xspeed:-1,yspeed:0,
                bulletIndex:bulletIndex,bitmapData:new LBitmapData(imglist["item"
+bulletIndex])
            });
            bulletCtrlLayer.addChild(obj);
        }
        var ctrlObject = ctrlList[ctrlIndex];
        if(ctrlObject["frames"] == frames){
            ctrlIndex++;

            bitmapData = new LBitmapData(imglist[ctrlObject.img]);
            var enemy;
            if(ctrlObject.isboss){
                enemy = new Boss(ctrlObject.x,ctrlObject.y,0,bitmapData.height*0.5,
bitmapData,ctrlObject["hp"]);
            }else{
                 enemy = new Enemy(ctrlObject.x,ctrlObject.y,0,bitmapData.height*0.5,
bitmapData,ctrlObject["hp"]);
            }
            plainLayer.addChild(enemy);
            enemy.setBullet(ctrlObject.bullet);
            enemy.move = ctrlObject.move;
            enemy.canshoot=true;
        }
    }
```

代码解析

与代码清单 8-7 做对比，可以发现代码清单 8-12 增加了如下代码：

```
if(frames % 50 == 0){
            var bulletIndex = Math.random()>0.5?1:2;
            var obj = new BulletCtrl({
                    x:790,y:100+Math.random()*200,xspeed:-1,yspeed:0,
                    bulletIndex:bulletIndex,bitmapData:new
LBitmapData(imglist["item"+bulletIndex])
            });
            bulletCtrlLayer.addChild(obj);
}
```

上面代码说明，frames 每运行 50 个单位，就往游戏的 bulletCtrlLayer 层上加入一个弹药。弹药的子弹序号随机设定为 1 或者 2。

弹药的运行效果如图 8-1 所示，图中的数字 2 标识的就是一个弹药。

但是目前，弹药还不能和我机发生碰撞，所以需要在弹药类的循环函数中加入与我机的碰撞检测，代码如下所示：

```
if(LGlobal.hitTestArc(self,player)){
            player.setBullet(self.bulletIndex);
```

```
            self.parent.removeChild(self);
}
```

这段代码表示，使用 LGlobal.hitTestArc 函数进行碰撞检测，当与我机发生碰撞时，将我机的子弹序号设定为弹药中所保存的子弹序号，然后将弹药从游戏画面中移除。添加碰撞检测后的循环函数如下所示：

```
/**
 * 循环
 * */
BulletCtrl.prototype.onframe = function (){
            var self = this;
            //移动
            self.x += self.xspeed;
            self.y += self.yspeed;
            //位置检测
            if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y > 
LGlobal.height){
                        //从屏幕移除
                        bulletCtrlLayer.removeChild(self);
            }
            if(LGlobal.hitTestArc(self,player)){
                        player.setBullet(self.bulletIndex);
                        self.parent.removeChild(self);
            }
};
```

运行代码，可以发现当我机与弹药发生碰撞时，我机所发射的子弹会根据弹药中的子弹序号而发生变化。

8.7　飞机生命值的显示

游戏中的飞机是有生命值的，但是在目前的画面上，我机的生命值是看不到的。下面就将我机的生命值显示到游戏的画面上。

首先，在游戏进行初始化的时候，在初始化函数 gameInit 中加入以下代码：

```
hpText = new LTextField();
hpText.color="#ffffff";
hpText.x = hpText.y = 10;
textLayer.addChild(hpText);
```

上面代码表示，新建一个 LTextField 对象，将其添加到游戏的 textLayer 层上。因为游戏的背景是黑色的，所以这里将文字的颜色设定为白色。

然后在 Main.js 中添加 showText 函数。

```
function showText(){
            hpText.text = "";
```

```
            for(var i=0;i<player.hp;i++){
                  hpText.text += " ■ ";
            }
}
```

上面的代码将 hpText 所表示的文字设定为连续的小方块，并且这些小方块的数量是由我机的生命值 hp 来决定的，也就是说小方块的数量就代表了我机的生命值。

最后，在 Main.js 中的循环函数中调用 showText 函数，这样就实现了我机生命值的即时表示方法。

运行代码，可以得到如图 8-1 所示的效果，图中左上角的小方块就是飞机的生命值。

8.8 游戏胜利与失败判定

当我机的生命值 hp 降为 0 的时候，游戏应该结束。所以在我机 Player 类的循环函数中还应该添加对生命值的判定，代码如下所示：

```
/**
 * 循环
 * */
Player.prototype.onframe = function (){
            var self = this;
            self.callParent("onframe",arguments);

            if(self.hp <= 0){
                  gameOver();
            }
};
```

上面代码表示，在飞机生命值降为 0 之后，调用 gameOver 函数，表示游戏结束。

下面来具体实现 gameOver 函数的内容，如代码清单 8-13 所示。

代码清单 8-13

```
function gameOver(){
            gameLayer.die();
            gameLayer.graphics.drawRect(1,"#000000",[0,0,800,400],true,"#000000");
            var overLayer = new LSprite();
            gameLayer.addChild(overLayer);
            overLayer.graphics.drawRect(4,'#ff8800',[0,0,200,100],true,'#ffffff');
            overLayer.x = (LGlobal.width - overLayer.getWidth())*0.5;
            overLayer.y = (LGlobal.height - overLayer.getHeight())*0.5;

            txt = new LTextField();
            txt.text = " 游戏结束！！ ";
            txt.size = 20;
            txt.x = 20;
            txt.y = 40;
```

```
            overLayer.addChild(txt);
    }
```

代码解析

```
gameLayer.die();
```

上面的代码首先会将加载到 gameLayer 层的所有事件清除。

```
gameLayer.graphics.drawRect(1,"#000000",[0,0,800,400],true,"#000000");
var overLayer = new LSprite();
gameLayer.addChild(overLayer);
overLayer.graphics.drawRect(4,'#ff8800',[0,0,200,100],true,'#ffffff');
overLayer.x = (LGlobal.width - overLayer.getWidth())*0.5;
overLayer.y = (LGlobal.height - overLayer.getHeight())*0.5;
```

以上代码表示在游戏画面上画出一个小的矩形窗体，用来显示游戏结果。

```
txt = new LTextField();
txt.text = "游戏结束！！";
txt.size = 20;
txt.x = 20;
txt.y = 40;
overLayer.addChild(txt);
```

这里表示在小的矩形窗体上添加 LTextField 对象，显示“游戏结束”字样。

当我机生命值降为 0 的时候，得到的游戏效果如图 8-12 所示。

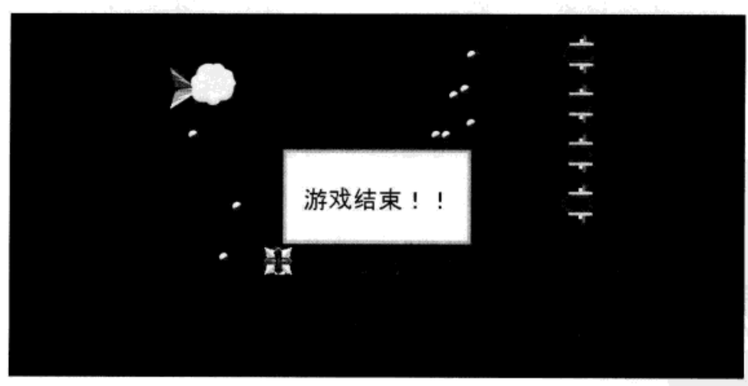


图 8-12 游戏结束效果图

除了游戏结束画面外，还有游戏通关画面。从代码清单 8-8 中可以得出，当敌机 Boss 被击毙之后，会调用 gameClear 函数。gameClear 函数的具体内容如代码清单 8-14 所示。

代码清单 8-14

```
function gameClear(){
        gameLayer.die();
```

```
        gameLayer.graphics.drawRect(1,"#000000",[0,0,800,400],true,"#000000");
        var overLayer = new LSprite();
        gameLayer.addChild(overLayer);
        overLayer.graphics.drawRect(4,'#ff8800',[0,0,200,100],true,'#ffffff');
        overLayer.x = (LGlobal.width - overLayer.getWidth())*0.5;
        overLayer.y = (LGlobal.height - overLayer.getHeight())*0.5;

        txt = new LTextField();
        txt.text = "游戏通关！！";
        txt.size = 20;
        txt.x = 20;
        txt.y = 40;
        overLayer.addChild(txt);
}
```

上面代码的功能和代码清单 8-13 完全相同，只是显示的文字内容不同。gameClear 函数表示结果为游戏通关。

运行代码，当消灭了敌机 Boss 的时候，得到的游戏效果图如图 8-13 所示。

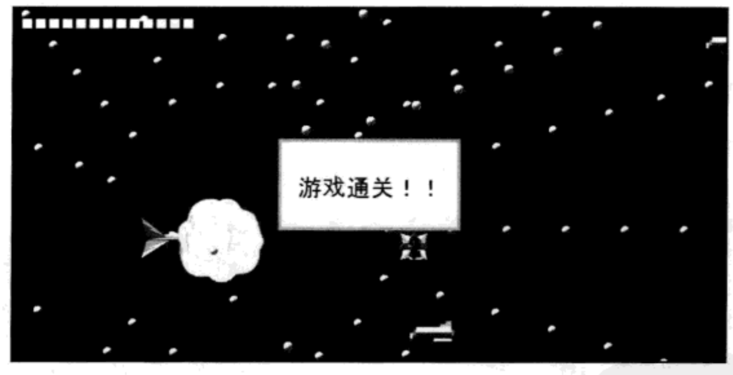


图 8-13　游戏通关效果图

8.9　小结

通过本章内容，大家应该对射击游戏的原理以及游戏中物体的碰撞有了简单的了解。由于本章主要是为了讲解此类游戏的原理，所以本次所制作的游戏只有一个关卡。如果想制作完整的射击游戏，还需要对敌机数组和子弹数组等相关内容进行一些扩展，为游戏添加更多种变化的子弹，为敌机实现更复杂的移动规律和行动 AI，这样游戏才能更有趣。

第 9 章　开发物理游戏

物理引擎是编写计算机程序来模拟牛顿力学的模型，主要是通过为刚性物体赋予真实的物理属性的方式来计算物体的运动、旋转和碰撞反应，从而模拟真实的物理世界。物理引擎的一个重要应用就是游戏开发。物理引擎有很多，本章主要介绍 Box2D。

9.1　Box2D 简介

Box2D 是暴雪软件原理工程师 Erin Catto 为在 2006 年召开的中国游戏开发者大会上做物理学演示而设计的。最初是用 C++ 编写的，后来衍生出了 flash、Java 和 Object-C 等多种版本。

Box2D 提供了矩形、圆形及多边形等几何形状的物体仿真，并且可以为这些物体添加密度、摩擦力、弹力等属性，还可用接头连接不同的形状，甚至可以包括关节、马达和滑轮，从而更真实地在计算机上模拟了真实的物理世界。

2008 年 5 月 10 日，日本开发者あんどうやすし试着用 JavaScript 移植 Box2D，2008 年 5 月 17 日，他发布了 JavaScript 版的 Box2D 物理引擎，并将其命名为 Box2DJS。但目前 Box2DJS 已经停止更新，最新 JavaScript 版本的 Box2D 引擎是由 Uli.Hecht 在 2011 年 6 月发布的 Box2dWeb-2.1a。

本章主要介绍一下如何在 lufylegend 库件中结合 Box2dWeb 来制作物理游戏。Box2dWeb 的下载地址如下，http://code.google.com/p/box2dweb/downloads/list。

9.2　Box2dWeb 在 lufylegend 库件中的使用

在 Box2d 的物理世界里，b2World 被称为一个世界，在这个世界里有各种物体（2BodyDef）和关节（b2JointDef），每个物体都有自己的形状（2ShapeDef），这些被添加到物理世界中的物体都遵循牛顿运动定律。

那么，为什么要使用 lufylegend 库件来操作 Box2dWeb 呢？

首先来看一个例子，如图 9-1 所示是 Box2dWeb 下载包中的 demo。

代码清单 9-1 是上面 demo 的源代码。

PDG

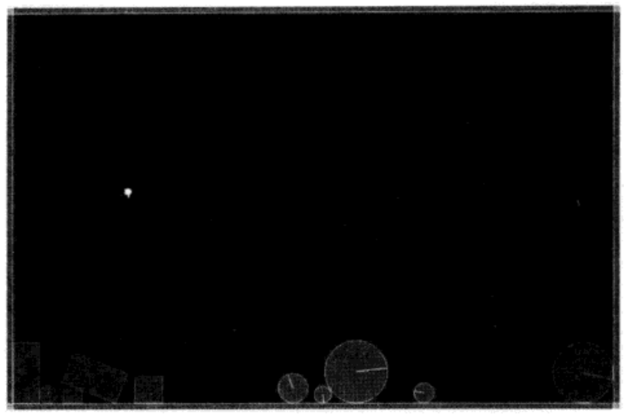


图 9-1 Box2dWeb 的 demo 效果

代码清单 9-1

```
function init() {
        var b2Vec2 = Box2D.Common.Math.b2Vec2,
        b2AABB = Box2D.Collision.b2AABB,
        b2BodyDef = Box2D.Dynamics.b2BodyDef,
        b2Body = Box2D.Dynamics.b2Body,
        b2FixtureDef = Box2D.Dynamics.b2FixtureDef,
        b2Fixture = Box2D.Dynamics.b2Fixture,
        b2World = Box2D.Dynamics.b2World,
        b2MassData = Box2D.Collision.Shapes.b2MassData,
        b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape,
        b2CircleShape = Box2D.Collision.Shapes.b2CircleShape,
        b2DebugDraw = Box2D.Dynamics.b2DebugDraw,
        b2MouseJointDef = Box2D.Dynamics.Joints.b2MouseJointDef;
        var world = new b2World(new b2Vec2(0, 10), //gravity
        true                     //allow sleep
        );
        var fixDef = new b2FixtureDef;
        fixDef.density = 1.0;
        fixDef.friction = 0.5;
        fixDef.restitution = 0.2;
        var bodyDef = new b2BodyDef;
        //create ground
        bodyDef.type = b2Body.b2_staticBody;
        fixDef.shape = new b2PolygonShape;
        fixDef.shape.SetAsBox(20, 2);
        bodyDef.position.Set(10, 400 / 30 + 1.8);
        world.CreateBody(bodyDef).CreateFixture(fixDef);
        bodyDef.position.Set(10, -1.8);
        world.CreateBody(bodyDef).CreateFixture(fixDef);
        fixDef.shape.SetAsBox(2, 14);
        bodyDef.position.Set(-1.8, 13);
        world.CreateBody(bodyDef).CreateFixture(fixDef);
```

```
        bodyDef.position.Set(21.8, 13);
        world.CreateBody(bodyDef).CreateFixture(fixDef);
        //create some objects
        bodyDef.type = b2Body.b2_dynamicBody;
        for(var i = 0; i < 10; ++i) {
                if(Math.random() > 0.5) {
                        fixDef.shape = new b2PolygonShape;
                        fixDef.shape.SetAsBox(Math.random() + 0.1//half width,
                        Math.random() + 0.1 //half height
                        );
                } else {
                        fixDef.shape = new b2CircleShape(Math.random() + 0.1
//radius

                        );
                }
                bodyDef.position.x = Math.random() * 10;
                bodyDef.position.y = Math.random() * 10;
                world.CreateBody(bodyDef).CreateFixture(fixDef);
        }
        //setup debug draw
        var debugDraw = new b2DebugDraw();
        debugDraw.SetSprite(document.getElementById("canvas").getContext("2d"));
        debugDraw.SetDrawScale(30.0);
        debugDraw.SetFillAlpha(0.5);
        debugDraw.SetLineThickness(1.0);
        debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);
        world.SetDebugDraw(debugDraw);
        window.setInterval(update, 1000 / 60);
        //mouse
        var mouseX, mouseY, mousePVec, isMouseDown, selectedBody, mouseJoint;
        var canvasPosition = getElementPosition(document.getElementById("canvas"));
        document.addEventListener("mousedown", function(e) {
                isMouseDown = true;
                handleMouseMove(e);
                document.addEventListener("mousemove", handleMouseMove, true);
        }, true);
        document.addEventListener("mouseup", function() {
                document.removeEventListener("mousemove", handleMouseMove, true);
                isMouseDown = false;
                mouseX = undefined;
                mouseY = undefined;
        }, true);
        function handleMouseMove(e) {
                mouseX = (e.clientX - canvasPosition.x) / 30;
                mouseY = (e.clientY - canvasPosition.y) / 30;
        };
        function getBodyAtMouse() {
                mousePVec = new b2Vec2(mouseX, mouseY);
                var aabb = new b2AABB();
                aabb.lowerBound.Set(mouseX - 0.001, mouseY - 0.001);
                aabb.upperBound.Set(mouseX + 0.001, mouseY + 0.001);
```

```
                // Query the world for overlapping shapes.
                selectedBody = null;
                world.QueryAABB(getBodyCB, aabb);
                return selectedBody;
        }
        function getBodyCB(fixture) {
                if(fixture.GetBody().GetType() != b2Body.b2_staticBody) {
                        if(fixture.GetShape().TestPoint(fixture.GetBody().
GetTransform(), mousePVec)) {
                         selectedBody = fixture.GetBody();
                         return false;
                        }
                }
                return true;
        }
        //update
        function update() {
                if(isMouseDown && (!mouseJoint)) {
                        var body = getBodyAtMouse();
                        if(body) {
                                var md = new b2MouseJointDef();
                                md.bodyA = world.GetGroundBody();
                                md.bodyB = body;
                                md.target.Set(mouseX, mouseY);
                                md.collideConnected = true;
                                md.maxForce = 300.0 * body.GetMass();
                                mouseJoint = world.CreateJoint(md);
                                body.SetAwake(true);
                        }
                }
                if(mouseJoint) {
                        if(isMouseDown) {
                                mouseJoint.SetTarget(new b2Vec2(mouseX,
mouseY));
                        } else {
                                world.DestroyJoint(mouseJoint);
                                mouseJoint = null;
                        }
                }
                world.Step(1 / 60, 10, 10);
                world.DrawDebugData();
                world.ClearForces();
        };
        //helpers
        //http://js-tut.aardon.de/js-tut/tutorial/position.html
        function getElementPosition(element) {
                var elem = element, tagname = "", x = 0, y = 0;
                while(( typeof (elem) == "object") && ( typeof (elem.
tagName) != "undefined")) {
                        y += elem.offsetTop;
                        x += elem.offsetLeft;
```

```
                        tagname = elem.tagName.toUpperCase();
                        if(tagname == "BODY")
                                elem = 0;
                        if( typeof (elem) == "object") {
                                if( typeof (elem.offsetParent) == "object")
                                        elem = elem.offsetParent;
                        }
                }
                return {
                        x : x, y : y
                };
        }
};
```

大家可能不理解上面的代码，有一个大致印象即可。为了形成对比，接着来看一下使用 lufylegend 库件后的代码清单 9-2。

代码清单　9-2

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Box2dWeb Demo</title>
<script type="text/javascript"
            src="../../Box2dWeb-2.1.a.3.min.js"></script>
<script type="text/javascript"
            src="../../lufylegend-1.5.0.js"></script>
</head>
<body onload='init(10,"mylegend",600,400,main,LEvent.INIT)'>
<div id="mylegend">loading......</div>
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            backLayer.graphics.drawRect(0,"#000000",[0, 0, 600, 400],false);
            addChild(backLayer);
            gameInit();
}
function gameInit(){
            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            backLayer.addChild(cLayer);
            var shapeArray = [
                        [[0,0],[10,0],[10,400],[0,400]],
                        [[0,0],[600,0],[600,10],[0,10]],
                        [[590,0],[600,0],[600,400],[590,400]],
                        [[0,390],[600,390],[600,400],[0,400]]
            ];
```

```
                cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);

                for(var i=0;i<10;i++){
                        add();
                }
}
function add(){
                cLayer = new LSprite();
        cLayer.x = Math.random()*10*LGlobal.box2d.drawScale;
        cLayer.y = Math.random()*10*LGlobal.box2d.drawScale;
                backLayer.addChild(cLayer);
        var r = (Math.random() + 0.1)*LGlobal.box2d.drawScale;
        if(Math.random() > 0.5) {
                cLayer.addBodyCircle(r,0,0,1,.5,.4,.5);
        }else{
                cLayer.addBodyPolygon(r,r,1,5,.4,.2);
        }
cLayer.setBodyMouseJoint(true);
}
</script>
</body>
</html>
```

我们将代码清单 9-1 和代码清单 9-2 进行对比，可以看到，使用 lufylegend 库件后，实现同样的功能，代码清单 9-2 中的代码更为简洁。这也是我们使用 lufylegend 库件来操作 Box2dWeb 的主要原因。下面将详细介绍如何将 lufylegend 库件与 Box2dWeb 结合起来，一步步地创建一个仿真的物理世界。

9.3 创建各种各样的物体

要在 lufylegend 库件中使用 Box2dWeb 首先需要对 Box2d 进行初始化。初始化很简单，只需要加入下面一行代码即可。

```
LGlobal.box2d = new LBox2d();
```

在 Box2D 中最基本的对象叫做刚体（rigid body），刚体表示一块十分坚硬的物质，它上面的任何两点之间的距离都是完全不变的，它们就像“钻石”那样“坚硬”。下面主要介绍如何使用 lufylegend 库件对 Box2d 的刚体进行创建和操作。为了加以区分，在后面的讨论中，我们用物体（body）来代替刚体，实际上这里所说的物体就是刚体。在真实的世界里，存在着各种形状的物体，圆的、方的、大的、小的等，现在就来看看如何在程序中实现各种各样的物体。

9.3.1 矩形物体

通过 addBodyPolygon 函数可以为 LSprite 对象添加一个矩形的物体。

函数原型：

```
addBodyPolygon(width,height,type,density,friction,restitution);
```

参数含义：

- width 表示矩形物体的宽；
- height 表示矩形物体的高；
- type 表示静态或动态；
- density 表示密度；
- friction 表示摩擦；
- restitution 表示弹力。

代码清单 9-3 利用 addBodyPolygon 函数来创建一个物体。

代码清单　9-3

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            //打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            //加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(100,60,1,5,0.4,0.2);
}
</script>
```

运行效果如图 9-2 所示。

代码解析

```
LGlobal.setDebug(true);
```

这里表示首先打开 debug 模式，图 9-1 和图 9-2 中的图形都是在 debug 模式下才会显示的，如果关闭 debug 模式，这些图形是不会显示的。

图 9-2　创建矩形物体

接着对 Box2d 进行初始化，代码如下所示：

```
LGlobal.box2d = new LBox2d();
```

然后加入一个静态的矩形物体作为地板，代码如下所示：

```
cLayer = new LSprite();
cLayer.x = 300;
cLayer.y = 390;
backLayer.addChild(cLayer);
cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
```

上面的代码使用 addBodyPolygon 函数给 LSprite 对象加入了一个矩形物体，并且将函数的第三个参数设置为了 0，表示物体为静态，这样加入的物体就是静止不动的了。

最后加入一个动态的矩形物体，代码如下所示：

```
cLayer = new LSprite();
cLayer.x = 300;
cLayer.y = 100;
backLayer.addChild(cLayer);
cLayer.addBodyPolygon(100,60,1,5,0.4,0.2);
```

上面代码在坐标 (300,100) 的位置上加入了一个动态的矩形物体，因为前面已经加入了一个静态的矩形物体作为地板，所以这个动态的物体会掉落到地板上，如果没有这个地板，那么动态的物体就会一直做自由落体运动而持续下落。

Box2d 中的物体分为两种，一种是静态，另一种是动态。静态的物体是固定的，它们之间不会发生碰撞。动态物体的运动和碰撞是完全遵循牛顿定律的，也就是说只要给它们设置合适的质量、摩擦和弹力等参数，剩下的事情都可以交给“牛顿”来完成了。

利用 lufylegend 库件给物体加上一个形状或者说给物体加上一个皮肤是很简单的，只要给 LSprite 对象加上一张图片就可以了，如代码清单 9-4 所示。

代码清单 9-4

```
<script type="text/javascript">
var backLayer,cLayer,bitmap;
var imglist = {};
var imgData = new Array(
                        {name:"face",path:"./face.jpg"}
                        );
function main(){
            //打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            backLayer.graphics.drawRect(0,"#000000",[0, 0, 600, 400],false);
            addChild(backLayer);
            LLoadManage.load(imgData,null,gameInit);
}
function gameInit(result){
            imglist = result;
```

```
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            //加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            bitmap = new LBitmap(new LBitmapData(imglist["face"],50,50,100,120));
            cLayer.addChild(bitmap);
            cLayer.addBodyPolygon(100,120,1,5,0.4,0.2);
}
</script>
```

运行效果如图 9-3 所示。

图 9-3　给矩形物体加上皮肤

代码解析

对比一下代码清单 9-3，可以看出加入物体时的主要区别如下：

```
bitmap = new LBitmap(new LBitmapData(imglist["face"],50,50,100,120));
cLayer.addChild(bitmap);
cLayer.addBodyPolygon(100,120,1,5,0.4,0.2);
```

上面的代码给 LSprite 对象加上了一个 LBitmap 对象，因为 LBitmap 对象中储存的是图片数据，所以这也就相当于给 LSprite 加上了一张图片。然后使用 addBodyPolygon 函数将 LSprite 对象变为动态物体，这样就创建了一个带有皮肤的矩形物体了。

9.3.2　圆形物体

通过 addBodyCircle 函数可以为 LSprite 对象添加一个圆形的物体。

函数原型：

```
addBodyCircle(radius,cx,cy,type,density,friction,restitution);
```

参数含义：

- radius 表示圆形物体的半径；

- cx 表示圆形物体圆心的 x 坐标；
- cy 表示圆形物体圆心的 y 坐标；
- type 表示静态或动态；
- density 表示密度；
- friction 表示摩擦；
- restitution 表示弹力。

代码清单 9-5 利用 addBodyCircle 函数创建一个物体。

代码清单 9-5

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            //打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            //加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的圆形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            cLayer.addBodyCircle(50,0,0,1,5,0.4,0.2);
}
</script>
```

运行效果如图 9-4 所示。

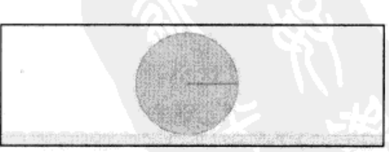


图 9-4 创建圆形物体

代码解析

代码清单 9-4 和代码清单 9-2 相比较，除了最后加入物体部分不一样以外，其余完全一样。

```
cLayer.addBodyCircle(50,0,0,1,5,0.4,0.2);
```

上面代码使用 addBodyCircle 函数，给 LSprite 对象加入了一个动态的圆形物体。

接着，利用 LSprite 对象给圆形物体加上皮肤，如代码清单 9-6 所示。

代码清单 9-6

```
<script type="text/javascript">
```

```
var backLayer,cLayer,bitmap;
var imglist = {};
var imgData = new Array(
                        {name:"face",path:"./face.jpg"}
                        );
function main(){
            //打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            backLayer.graphics.drawRect(0,"#000000",[0, 0, 600, 400],false);
            addChild(backLayer);
            LLoadManage.load(imgData,null,gameInit);
}
function gameInit(result){
            imglist = result;
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            //加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,0,0,0.5);
            //加入一个动态的圆形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            bitmap = new LBitmapData(imglist["face"],50,50,190,190);
            cLayer.graphics.beginBitmapFill(bitmap);
            cLayer.graphics.drawArc(1,"#000000",[50,50,50,0,2*Math.PI],true);
            cLayer.addBodyCircle(50,0,0,1,5,0.4,0.5);
}
</script>
```

运行效果如图 9-5 所示。

代码解析

```
    bitmap = new LBitmapData(imglist["face"]);
    cLayer.graphics.beginBitmapFill(bitmap);
    cLayer.graphics.drawArc(1,"#000000",[50,50,50,
0,2*Math.PI],true);
    cLayer.addBodyCircle(50,0,0,1,5,0.4,0.5);
```

图 9-5　给圆形物体加上皮肤

上面的代码主要是利用了 LGraphics 的 beginBitmapFill 方法，在 LSprite 对象上加上了一个圆形图片作为物体的皮肤。

9.3.3　多边形物体

addBodyVertices 函数可以通过顶点坐标数组的方式为 LSprite 对象添加一个任意的多边

形物体。下面我们就来认识一下该函数。

函数原型：

```
addBodyVertices(vertices,cx,cy,type,density,friction,restitution);
```

参数含义：

- vertices 表示多边形的顶点数组；
- cx 表示多边形物体圆心的 x 坐标；
- cy 表示多边形物体圆心的 y 坐标；
- type 表示静态或动态；
- density 表示密度；
- friction 表示摩擦；
- restitution 表示弹力。

代码清单 9-7 利用 addBodyVertices 函数创建一个多边形物体。

代码清单 9-7

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            // 打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            // 加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            // 加入一个动态的多边形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            var shapeArray = [
                [[0,25],[25,0],[150,25],[150,75],[25,100],[0,75]]
            ];
            cLayer.addBodyVertices(shapeArray,25,75,1,.5,.4,.5);
}
</script>
```

运行效果如图 9-6 所示。

代码解析

将代码清单 9-6 与之前的代码清单 9-2 和代码清单 9-4 进行对比，你会发现在加入物体

部分是有区别的，看下面的代码。

```
var shapeArray = [
    [[0,25],[25,0],[150,25],[150,75],[25,100],[0,75]]
];
cLayer.addBodyVertices(shapeArray,25,75,1,.5,.4,.5);
```

在上面的代码中，使用 addBodyVertices 函数给 LSprite 对象加入了一个动态的多边形物体。

图 9-6　创建多边形物体

与矩形物体和圆形物体一样，也可以利用 LSprite 对象给多边形物体加上皮肤，看下面代码清单 9-8。

代码清单　9-8

```
<script type="text/javascript">
var backLayer,cLayer,bitmap;
var imglist = {};
var imgData = new Array(
                        {name:"face",path:"./face.jpg"}
                        );
function main(){
            //打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            backLayer.graphics.drawRect(0,"#000000",[0, 0, 600, 400],false);
            addChild(backLayer);
            LLoadManage.load(imgData,null,gameInit);
}
function gameInit(result){
            imglist = result;
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            //加入一个静态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,0,0,0.5);
            //加入一个动态的多边形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            bitmap = new LBitmapData(imglist["face"],50,50,190,190);
            cLayer.graphics.beginBitmapFill(bitmap);

            var verticesArray= [
                [0,25],[25,0],[150,25],[150,75],[25,100],[0,75]
            ];
            cLayer.graphics.drawVertices(1,"#000000",verticesArray,true);
```

```
            var shapeArray = [
                [[0,25],[25,0],[150,25],[150,75],[25,100],[0,75]]
            ];
            cLayer.addBodyVertices(shapeArray,25,75,1,.5,.4,.5);
}
</script>
```

运行效果如图 9-7 所示。

代码解析

```
    bitmap = new LBitmapData(imglist["face"]);
    cLayer.graphics.beginBitmapFill(bitmap);
    cLayer.graphics.drawArc(1,"#000000",[50,50,
50,0,2*Math.PI],true);
    cLayer.addBodyCircle(50,0,0,1,5,0.4,0.5);
```

图 9-7 给多边形物体加上皮肤

上面的代码主要是利用 LGraphics 的 beginBitmapFill 方法在 LSprite 对象上加入了一个多边形图片来作为物体的皮肤。

9.4 响应鼠标拖拽物体

在 Box2d 中，如果想用鼠标拖拽物体，那就要通过 b2MouseJointDef 对象和鼠标的 mousemove 事件来完成。而且，开发者必须通过 mousedown 事件，在鼠标按下的时候，为程序加上 b2MouseJointDef 对象和鼠标的 mousemove 事件，并且还要再通过 mouseup 事件，在鼠标弹起的时候，移除被加载的 b2MouseJointDef 对象和鼠标的 mousemove 事件。

而使用 lufylegend.js 库件后，鼠标拖拽就变得非常简便了，只需要调用 LSprite 对象的 setBodyMouseJoint 方法即可。下面就来看看 setBodyMouseJoint 方法的具体含义。

函数原型：

```
setBodyMouseJoint (flag);
```

参数含义：flag 表示是否支持鼠标拖拽。

具体的使用方法如代码清单 9-9 所示。

代码清单 9-9

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            // 打开 debug 模式
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);
            //Box2d 初始化
            LGlobal.box2d = new LBox2d();
            // 加入一个静态的矩形物体
```

```
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的矩形物体
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 100;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(100,60,1,5,0.4,0.2);
            //鼠标拖拽有效
            cLayer.setBodyMouseJoint(true);
}
</script>
```

代码解析

对比一下代码清单 9-9 和代码清单 9-3 可以发现，代码清单 9-9 只多了如下一行代码：

```
cLayer.setBodyMouseJoint(true);
```

传入 setBodyMouseJoint 函数的参数为 true，表示鼠标拖拽有效。如果要解除物体的鼠标拖拽，可以再次调用 setBodyMouseJoint 函数，并传入参数 false 即可。

运行代码，会发现已经可以用鼠标拖拽物体进行移动了。

9.5　关节（Joint）

关节又叫连接器，是一种用于把两个或多个物体固定到一起的约束。Box2d 支持多种关节，比如：旋转、齿轮等，下面会一一介绍。

9.5.1　距离关节（b2DistanceJointDef）

距离关节是一个最简单的关节，它约束了两个物体之间的距离，两个物体之间的距离关节一旦建立，它们的距离就将会固定住。当拖拽其中一个物体时，另一个物体为了保持距离固定不变，也会跟着移动。代码清单 9-10 建立了一个简单的距离关节。

代码清单　9-10

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
```

```
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的矩形物体1
            box01 = new LSprite();
            box01.x = 200;
            box01.y = 100;
            backLayer.addChild(box01);
            box01.addBodyPolygon(100,60,1,5,0.4,0.2);
            box01.setBodyMouseJoint(true);
            //加入一个动态的矩形物体2
            box02 = new LSprite();
            box02.x = 400;
            box02.y = 100;
            backLayer.addChild(box02);
            box02.addBodyPolygon(100,60,1,5,0.4,0.2);
            box02.setBodyMouseJoint(true);
            //加入一个距离关节
            LGlobal.box2d.setDistanceJoint(box01.box2dBody, box02.box2dBody);
}
</script>
```

代码运行效果如图 9-8 所示。

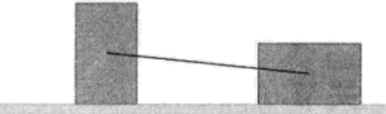


图 9-8 距离关节效果

代码解析

要给两个物体添加距离关节，只需要下面一行代码：

```
LGlobal.box2d.setDistanceJoint(box01.box2dBody, box02.box2dBody);
```

在这行代码中用到了 setDistanceJoint 函数，该函数的原型如下所示：

```
setDistanceJoint(b2BodyDefA, b2BodyDefB)
```

参数含义：

- ❑ b2BodyDefA 表示物体 A；
- ❑ b2BodyDefB 表示物体 B。

理解了 setDistanceJoint 函数，再来看看上面的那行代码，其中，box01 和 box02 是两个 LSprite 对象，lufylegend 库件通过 addBodyPolygon 将它们变成了矩形物体，然后通过调用 LSprite 对象的 box2dBody 属性，即可得到 Box2d 的一个物体（b2BodyDef），最后将保存在 box01 和 box02 中的 b2BodyDef 对象作为参数传给 setDistanceJoint 函数，就可以创建一个距离关节了。

9.5.2　旋转关节（b2RevoluteJointDef）

旋转关节可以强制两个物体共享一个描点，这样就能使它们进行相对旋转。代码清单 9-11 建立了一个简单的旋转关节。

代码清单　9-11

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的圆形物体 1
            box01 = new LSprite();
            box01.x = 250;
            box01.y = 200;
            backLayer.addChild(box01);
            box01.addBodyCircle(100,0,0,1,1,0.4,0.2);
            box01.setBodyMouseJoint(true);
            //加入一个静态的圆形物体 2
            box02 = new LSprite();
            box02.x = 250;
            box02.y = 150;
            backLayer.addChild(box02);
            box02.addBodyCircle(10,0,0,0,1,0.4,0.2);
            //加入一个旋转关节
            LGlobal.box2d.setRevoluteJoint(box01.box2dBody, box02.box2dBody);
}
</script>
```

上述代码的运行效果如图 9-9 所示。

代码解析

下面这行代码给两个物体添加了旋转关节。

```
LGlobal.box2d.setRevoluteJoint(box01.box2dBody,
box02.box2dBody);
```

其中用到了 setRevoluteJoint 函数，该函数的原型如下所示：

```
setRevoluteJoint (b2BodyDefA, b2BodyDefB, limits,
motors)
```

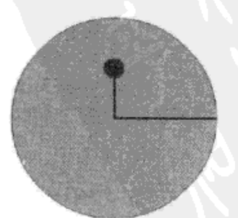


图 9-9　旋转关节效果

参数含义：

- b2BodyDefA 表示物体 A。
- b2BodyDefB 表示物体 B。
- limits 表示旋转角度限制数组，这个数组的内容是 [最小角度，最大角度]，在这里它可以限制旋转关节旋转的角度。
- motors 表示马达数组，这个数组的内容是 [力度，速度]，马达可以有很多用途，在这里它可以使关节自动进行旋转。

同样的，将保存在 box01 和 box02 中的 b2BodyDef 对象作为参数传给 setRevoluteJoint 函数，就可以创建一个旋转关节。但是上面的代码清单 9-11 省略了限制和马达，所以代码运行后的初始效果里，画面中的物体是静止的。那下面就来为其加入限制和马达，如代码清单 9-12 所示。

代码清单 9-12

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
        LGlobal.setDebug(true);
        backLayer = new LSprite();
        addChild(backLayer);

        LGlobal.box2d = new LBox2d();
        cLayer = new LSprite();
        cLayer.x = 300;
        cLayer.y = 390;
        backLayer.addChild(cLayer);
        cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
        //加入一个动态的圆形物体1
        box01 = new LSprite();
        box01.x = 250;
        box01.y = 200;
        backLayer.addChild(box01);
        box01.addBodyCircle(100,0,0,1,1,0.4,0.2);
        box01.setBodyMouseJoint(true);
        //加入一个静态的圆形物体2
        box02 = new LSprite();
        box02.x = 250;
        box02.y = 150;
        backLayer.addChild(box02);
        box02.addBodyCircle(10,0,0,0,1,0.4,0.2);
        //加入一个旋转关节
        LGlobal.box2d.setRevoluteJoint(box01.box2dBody, box02.box2dBody,
[-360,720*5],[450,2]);
}
</script>
```

其运行效果如图 9-10 所示。

代码解析

```
    LGlobal.box2d.setRevoluteJoint(box01.box2dBody,
box02.box2dBody ,[-360,720*5],[450,2]);
```

上面代码表示在添加旋转关节的时候，加入了限制和马达，这样画面中的物体就会自动发生旋转，而且因为有最大旋转角度的限制，所以当物体旋转了一段时间后就会停止下来。

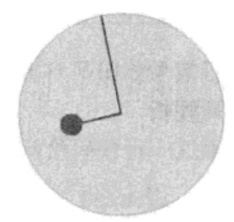


图 9-10　加入了限制和马达的旋转关节效果

9.5.3　滑轮关节（b2PulleyJointDef）

要使用滑轮关节，可以先创建一个滑轮，然后将两个物体通过一条“绳子”接通，使得当一个物体上升或者下降的时候，因为“绳子”的长短不变，另一个物体就会相应地下降或者上升。代码清单 9-13 建立了一个滑轮关节。

代码清单　9-13

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的圆形物体 1
            box01 = new LSprite();
            box01.x = 200;
            box01.y = 100;
            backLayer.addChild(box01);
            box01.addBodyCircle(20,0,0,1,1,0.5,0.6);
            box01.setBodyMouseJoint(true);
            //加入一个动态的圆形物体 2
            box02 = new LSprite();
            box02.x = 300;
            box02.y = 100;
            backLayer.addChild(box02);
            box02.addBodyCircle(20,0,0,1,1,0.4,0.2);
            box02.setBodyMouseJoint(true);
            //加入一个滑轮关节
            LGlobal.box2d.setPulleyJoint(box01.box2dBody, box02.box2dBody,[0,50,
```

```
300],[0,100,300],1.0);
    }
    </script>
```

运行效果如图 9-11 所示。

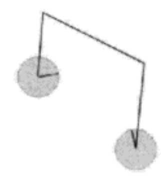
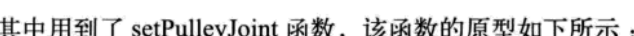


图 9-11 滑轮关节效果

代码解析

下面这一行代码给两个物体添加了滑轮关节。

```
    LGlobal.box2d.setPulleyJoint(box01.box2dBody, box02.
box2dBody, [0,50,300],[0,100,300],1.0);
```



其中用到了 setPulleyJoint 函数，该函数的原型如下所示：

```
    setPulleyJoint (b2BodyDefA, b2BodyDefB, anchorA,
anchorB, ratio)
```

参数含义：

- b2BodyDefA：表示物体 A。
- b2BodyDefB：表示物体 B。
- anchorA：表示物体 A 相关的控制参数数组，这个数组的内容是 [x,y,length]，使用 setPulleyJoint 建立滑轮关节的时候，会自动以物体本身的中心作为描点，anchorA 数组的前两个元素，决定了关节被建立时物体相对于这个描点的位置，anchorA 数组的最后一个元素，决定了左侧绳子的长度。
- anchorB：表示物体 B 相关的控制参数数组，该数组中各元素的含义同 anchorA。
- ratio：表示两边绳子的比例系数，比如在上面的例子中，如果将比例系数设置为 2，那么左边的物体上升 2 个单位的时候，右边物体只下降 1 个单位。

可以试着修改一下各个参数的值，看看会发生哪些变化。

9.5.4 移动关节（b2PrismaticJoint）

对于移动关节来说，它会有一个自由度，也就是说它限制了两个物体的移动范围，即只能沿指定轴相对移动。代码清单 9-14 建立了一个移动关节。

代码清单 9-14

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
```

PDG

```
                cLayer.y = 390;
                backLayer.addChild(cLayer);
                cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
                //加入一个静态的圆形物体 1
                box01 = new LSprite();
                box01.x = 200;
                box01.y = 100;
                backLayer.addChild(box01);
                box01.addBodyCircle(20,0,0,0,1,0.5,0.6);
                //加入一个动态的圆形物体 2
                box02 = new LSprite();
                box02.x = 200;
                box02.y = 100;
                backLayer.addChild(box02);
                box02.addBodyCircle(20,0,0,1,1,0.4,0.2);
                box02.setBodyMouseJoint(true);
                //加入一个移动关节
                LGlobal.box2d.setPrismaticJoint(box01.box2dBody, box02.box2dBody,
[0,1],[-5,2.5],[1,0]);
    }
    </script>
```

上述代码的运行效果如图 9-12 所示。

图 9-12　移动关节效果

代码解析

下面这行代码给两个物体添加了移动关节。

```
LGlobal.box2d.setPrismaticJoint(box01.box2dBody, box02.box2dBody,
                                        [0,1],[-5,2.5],[1,0]);
```

其中用到了 setPrismaticJoint 函数，该函数的原型如下所示 ：

```
setPrismaticJoint (b2BodyDefA, b2BodyDefB, vec, limits,motors)
```

参数含义 ：

- b2BodyDefA 表示物体 A。
- b2BodyDefB 表示物体 B。
- vec 表示物体 A 和物体 B 的相对移动方向，它是一个数组 [x,y]，设置不同的比例，可以建立不同方向上的移动关节。
- limits 表示移动的相对长度限制数组，这个数组的内容是 [正向最大长度 , 反向最大角度]，在这里它可以限制两个物体相对移动的最大长度。
- motors 表示马达数组，这个数组的内容是 [正向力度 , 反向力度]，这个马达可以给移动关节添加一个持续的力。比如在上面的例子中，如果将马达参数设置为 [0,10]，你会发现，物体不是向下移动了，而是向上移动，即使你用鼠标将物体拖拽到下面，它还是会因为马达的反向力度而向上移动。

9.5.5 齿轮关节（b2GearJoint）

使用Box2d可以模拟齿轮功能，这样就可以轻松地建立复杂的机械模型等。齿轮关节相对来说稍微复杂一些，因为它需要结合旋转关节和移动关节这两种关节。代码清单9-15建立了一个齿轮关节。

代码清单 9-15

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的圆形物体1
            box01 = new LSprite();
            box01.x = 200;
            box01.y = 200;
            backLayer.addChild(box01);
            box01.addBodyCircle(100,0,0,1,1,0.5,0.6);
            box01.setBodyMouseJoint(true);
            //加入一个静态的圆形物体2
            box02 = new LSprite();
            box02.x = 200;
            box02.y = 200;
            backLayer.addChild(box02);
            box02.addBodyCircle(20,0,0,0,1,0.4,0.2);
            //加入一个动态的圆形物体3
            box03 = new LSprite();
            box03.x = 500;
            box03.y = 200;
            backLayer.addChild(box03);
            box03.addBodyCircle(20,0,0,1,1,0.5,0.6);
            box03.setBodyMouseJoint(true);
            //在物体1和物体2之间建立旋转关节
            var revoluteJoint = LGlobal.box2d.setRevoluteJoint(box02.box2dBody,
box01.box2dBody);
            //在物体2和物体3之间建立移动关节
            var prismaticJoint = LGlobal.box2d.setPrismaticJoint(box03.box2dBody,
box02.box2dBody,[1,0],[-5,2.5],[1,0]);
            //利用建立好的旋转关节和移动关节，在物体1和物体3之间建立齿轮关节
            LGlobal.box2d.setGearJoint(box01.box2dBody, box03.
box2dBody,2,revolute-Joint,prismaticJoint);
```

```
}
</script>
```

上述代码的运行效果如图 9-13 所示。

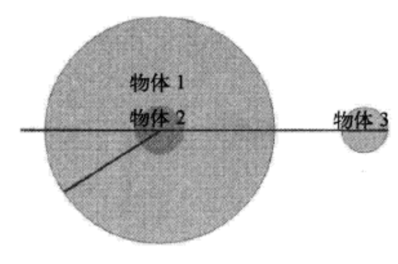


图 9-13　齿轮关节效果

代码解析

因为要用到旋转关节和移动关节，所以代码清单 9-15 在创建了 3 个物体之后，首先建立了就这两个关节，代码如下所示：

```
// 在物体 1 和物体 2 之间建立旋转关节
var revoluteJoint = LGlobal.box2d.setRevoluteJoint(box02.box2dBody, box01.
box2dBody);
// 在物体 2 和物体 3 之间建立移动关节
var prismaticJoint = LGlobal.box2d.setPrismaticJoint(box03.box2dBody,box02.
box2dBody,[1,0],[-5,2.5],[1,0]);
```

最后，利用建立好的旋转关节和移动关节，在物体 1 和物体 3 之间建立齿轮关节，代码如下所示：

```
LGlobal.box2d.setGearJoint(box01.box2dBody, box03.box2dBody,2,
                                   revoluteJoint,prismaticJoint);
```

这里用到了 setGearJoint 函数，该函数的原型如下所示：

```
setGearJoint (b2BodyDefA, b2BodyDefB,ratio, revoluteJointDef,prismaticJointDef)
```

参数含义：

- b2BodyDefA 表示物体 A。
- b2BodyDefB 表示物体 B。
- ratio 表示齿轮的比例系数，这个数值越小，物体 A 旋转 1 周使得物体 B 移动的距离也就越大，如果这个值设置得很大，那么物体 A 旋转几周才能使 B 移动一段很短的距离。

PDG

- revoluteJointDef 为齿轮关节中的物体 A 和轴心所建立的旋转关节。
- prismaticJointDef 为齿轮关节中的物体 B 和齿轮轴心所建立的移动关节。

9.5.6 悬挂关节（b2LineJoint）

悬挂关节类似于一个垂直的移动关节，它将一个物体悬挂到了另一物体上，如代码清单 9-16 所示。

代码清单 9-16

```
    <script type="text/javascript">
    var backLayer,cLayer;
    function main(){
                LGlobal.setDebug(true);
                backLayer = new LSprite();
                addChild(backLayer);

                LGlobal.box2d = new LBox2d();
                cLayer = new LSprite();
                cLayer.x = 300;
                cLayer.y = 390;
                backLayer.addChild(cLayer);
                cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
                //加入一个静态的圆形物体1
                box01 = new LSprite();
                box01.x = 200;
                box01.y = 100;
                backLayer.addChild(box01);
                box01.addBodyCircle(20,0,0,0,1,0.5,0.6);
                //加入一个动态的圆形物体2
                box02 = new LSprite();
                box02.x = 200;
                box02.y = 200;
                backLayer.addChild(box02);
                box02.addBodyCircle(20,0,0,1,1,0.4,0.2);
                box02.setBodyMouseJoint(true);
                //加入一个悬挂关节
                LGlobal.box2d.setLineJoint(box01.box2dBody, box02.box2dBody,[0,1],
[-2,2],[200,10]);
    }
    </script>
```

上述代码的运行效果如图 9-14 所示。

代码解析

图 9-14 和图 9-12 看似一样，但是图 9-12 中的物体是不可旋转的，而图 9-14 中下方的物体自身是可以旋转的。下面这一行代码用于给两个物体添加悬挂关节。

```
LGlobal.box2d.setLineJoint(box01.box2dBody, box02.box2dBody,
                                  [0,1],[-2,2],[200,10]);
```

其中用到了 setLineJoint 函数，该函数的原型如下所示：

```
setLineJoint (b2BodyDefA, b2BodyDefB, vec, limits,motors)
```

图 9-14　悬挂关节效果

参数含义：

- b2BodyDefA 表示物体 A。
- b2BodyDefB 表示物体 B。
- vec 表示物体 B 相对于悬挂点的移动方向，这个悬挂点就是物体 B 的初始位置，vec 是一个数组 [x,y]，设置不同的比例，可以建立不同方向上的悬挂关节；这和移动关节比较类似，大家可以试着改变这个参数的值，来体验一下它们的具体区别。
- limits 表示移动的相对长度限制数组，这个数组的内容是 [正向最大长度，反向最大角度]，它可以限制两个物体相对移动的最大长度。
- motors 表示马达数组，这个数组的内容是 [正向力度 , 反向力度]，这个马达可以给移动关节添加一个持续的力，比如在上面的例子中，如果将马达参数设置为 [0,10]，你会发现，物体不是向下移动了，而是向上移动，即使你用鼠标将物体拖拽到下面，它还是会因为马达的反向力度而再次向上移动。

9.5.7　焊接关节（b2WeldJoint）

焊接关节相当于捆绑，就是将两个物体牢牢地绑在一起，使其成为一个物体，如代码清单 9-17 所示。

代码清单　9-17

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入一个动态的圆形物体 1
            box01 = new LSprite();
            box01.x = 200;
            box01.y = 100;
            backLayer.addChild(box01);
            box01.addBodyCircle(50,0,0,1,1,0.4,0.2);
```

```
            box01.setBodyMouseJoint(true);
            //加入一个动态的圆形物体2
            box02 = new LSprite();
            box02.x = 250;
            box02.y = 100;
            backLayer.addChild(box02);
            box02.addBodyCircle(50,0,0,1,1,0.4,0.2);
            box02.setBodyMouseJoint(true);
            //加入一个焊接关节
            LGlobal.box2d.setWeldJoint(box01.box2dBody, box02.box2dBody);
}
</script>
```

上述代码的运行效果如图 9-15 所示。

代码解析

下面这行代码表示给两个物体添加焊接关节。

```
LGlobal.box2d.setWeldJoint(box01.box2dBody, box02.
box2dBody);
```

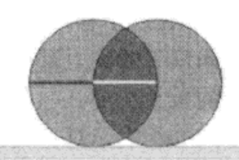


图 9-15 焊接关节效果

其中用到了 setWeldJoint 函数，该函数的原型如下所示：

```
setWeldJoint (b2BodyDefA, b2BodyDefB)
```

参数含义：

- b2BodyDefA 表示捆绑对象物体 A；
- b2BodyDefB 表示捆绑对象物体 B。

焊接关节在物理世界中也是比较常见的，比如，有时候我们可能需要创建一个比较复杂或者不规则形状的物体，这时无法用一个矩形或者圆形来表示，那就可能需要用到焊接关节了，通过焊接可用一些简单的物体组成自己需要的特殊形状。

9.5.8 鼠标关节（Mouse Joint）

鼠标关节是为了拖拽物体而设置的，这个在 9.4 节“响应鼠标拖拽物体”中已经进行了详细介绍，这里就不重复了。

9.6 力

虽然在 Box2d 所创建的物理世界中，一切物体的运动都是遵循牛顿定律的，它们之间的碰撞和运动都不用我们来控制，但是有时候我们需要改变它们的运动轨迹，这时候我们可以给某个物体加上一个力，从而改变它的运动。来看代码清单 9-18，在这个代码段中，每单击一次鼠标，就新建一个物体，并且给它加上一个冲力，把这个物体弹飞出去。

代码清单　9-18

```
<script type="text/javascript">
var backLayer,cLayer;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            backLayer.graphics.drawRect(1,"#000",[0,0,600,400]);
            addChild(backLayer);
            //加入地面
            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 300;
            cLayer.y = 390;
            backLayer.addChild(cLayer);
            cLayer.addBodyPolygon(600,10,0,5,0.4,0.2);
            //加入鼠标事件
            backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,createBox);
}
function createBox(event){
            //加入一个动态物体
            var box01 = new LSprite();
            box01.x = event.selfX;
            box01.y = event.selfY;
            backLayer.addChild(box01);
            box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
            //定义一个力的大小
            var force = 500;
            //定义一个方向
            var vec = new LGlobal.box2d.b2Vec2(force,-force);
            //给物体box01加上一个指定大小、指定方向的力
            box01.box2dBody.ApplyForce(vec, box01.box2dBody.GetWorldCenter());
}
</script>
```

上述代码的运行效果如图 9-16 所示。

图 9-16　加上力的运行效果

代码解析

为了不断加入新的物体，这里加入了鼠标事件，代码如下所示：

```
//加入鼠标事件
backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,createBox);
```

当单击鼠标的时候，会调用 createBox 函数。下面来看 createBox 函数中的代码段。

```
//加入一个动态物体
var box01 = new LSprite();
box01.x = event.selfX;
box01.y = event.selfY;
backLayer.addChild(box01);
box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
```

上面代码就是在鼠标单击的位置上加入一个动态的物体。

```
//定义一个力的大小
var force = 500;
//定义一个方向，方向的冲量大小由上面定义的 force 来控制
var vec = new LGlobal.box2d.b2Vec2(force,-force);
//给物体 box01 加上一个指定大小、指定方向的力
box01.box2dBody.ApplyForce(vec, box01.box2dBody.GetWorldCenter());
```

上面代码使用 ApplyForce 函数，给物体加上一个指定方向的冲力，使其获得一个初始速度，然后弹飞出去。其中的 box01.box2dBody.GetWorldCenter() 表示取得物体 box01 的中心。

ApplyForce 函数的原型如下所示：

```
ApplyForce (vec, point)
```

参数含义：

- vec 表示带有方向的冲量；
- point 表示施加力的着力点。

9.7 碰撞检测

在 Box2d 的世界里，物体是可以相互自由碰撞的，它们如何进行碰撞我们不必去管，但是在游戏中，有时候我们需要知道它们碰撞的力度，比如将一块石头砸向一块玻璃，玻璃是否被砸碎取决于石头和玻璃接触时碰撞力度的大小。在 lufylegend 中，可以使用 LGlobal.box2d.setEvent 来侦听碰撞事件。

LGlobal.box2d.setEvent 函数的原型如下所示：

```
LGlobal.box2d.setEvent (type,listener)
```

参数含义：

- type 事件类型包括 LEvent. PRE_SOLVE（碰撞前）、LEvent. BEGIN_CONTACT（碰撞开始）、LEvent.END_CONTACT（碰撞结束）、LEvent. POST_SOLVE（碰撞后）4 种类型。
- listener 是侦听函数，在上面 4 种碰撞类型中，只有 LEvent. POST_SOLVE（碰撞后）能够取得碰撞的力度。

具体用法如代码清单 9-19 所示。

代码清单 9-19

```
<script type="text/javascript">
var backLayer,cLayer,label;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 0;
            cLayer.y = 0;
            backLayer.addChild(cLayer);
            //通过顶点坐标数组，来加入上下左右 4 面墙
            var shapeArray = [
                [[0,0],[600,0],[600,10],[0,10]],
                [[600,0],[600,400],[590,400],[590,10]],
                [[600,400],[0,400],[0,390],[590,390]],
                [[0,0],[10,0],[10,400],[0,400]]
            ];
            cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);
            //加入一些矩形物体
            for(var i = 0;i<5;i++){
                        cLayer = new LSprite();
                cLayer.x = Math.random()*10*LGlobal.box2d.drawScale+10;
                cLayer.y = Math.random()*10*LGlobal.box2d.drawScale+10;
                        backLayer.addChild(cLayer);
                var w = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
                var h = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
                        cLayer.addBodyPolygon(w,h,1,5,0.4,0.2);
            }
            //加入一个圆形物体
            var box01 = new LSprite();
            box01.name = "mybox";
            box01.x = 250;
            box01.y = 200;
            backLayer.addChild(box01);
            box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
            //给圆形物体加上鼠标拖拽
            box01.setBodyMouseJoint(true);
            //为显示碰撞的力度，建立一个文本对象，显示力度数值
            label = new LTextField();
            backLayer.addChild(label);
            //加入碰撞侦听事件
            LGlobal.box2d.setEvent(LEvent.POST_SOLVE,postSolve);
}
//侦听函数
function postSolve(contact, impulse){
            var objA = contact.GetFixtureA().GetBody().GetUserData();
            var objB = contact.GetFixtureB().GetBody().GetUserData();
```

```
            if(objA.type == "LSprite" && objA.name == "mybox"){
                        label.text = impulse.normalImpulses[0];
            }else if(objB.type == "LSprite" && objB.name == "mybox"){
                        label.text = impulse.normalImpulses[0];
            }
}
</script>
```

运行效果如图 9-17 所示。

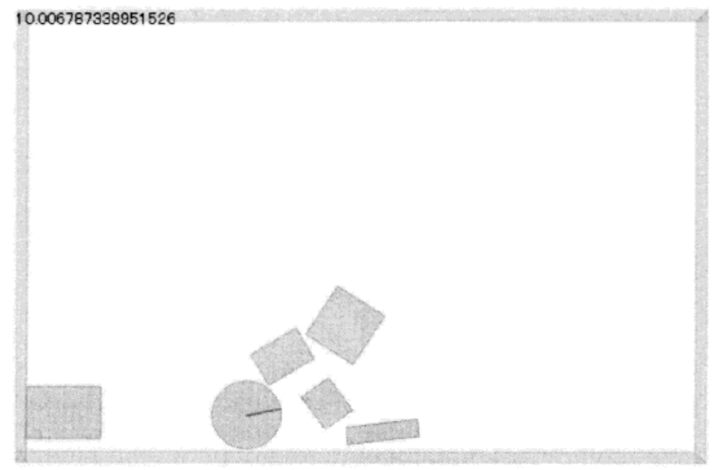


图 9-17 碰撞检测效果

代码解析

首先来建立上下左右 4 个墙壁，代码如下所示：

```
//通过顶点坐标数组，来加入上下左右4面墙
var shapeArray = [
    [[0,0],[600,0],[600,10],[0,10]],
    [[600,0],[600,400],[590,400],[590,10]],
    [[600,400],[0,400],[0,390],[590,390]],
    [[0,0],[10,0],[10,400],[0,400]]
];
cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);
```

在 9.3.3 节中已经介绍了如何利用顶点数组来添加多边形，这里就是利用顶点数组来同时添加 4 组矩形，从而实现添加一个矩形框的目的。

```
//加入一些矩形物体
for(var i = 0;i<5;i++){
            cLayer = new LSprite();
            cLayer.x = Math.random()*10*LGlobal.box2d.drawScale+10;
            cLayer.y = Math.random()*10*LGlobal.box2d.drawScale+10;
```

```
            backLayer.addChild(cLayer);
            var w = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
            var h = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
            cLayer.addBodyPolygon(w,h,1,5,0.4,0.2);
}
```

上面代码先加入一些矩形物体，以便在下面检测它们之间的碰撞。

```
//加入一个圆形物体
var box01 = new LSprite();
box01.name = "mybox";
box01.x = 250;
box01.y = 200;
backLayer.addChild(box01);
box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
//给圆形物体加上鼠标拖拽
box01.setBodyMouseJoint(true);
```

上述代码加入了一个圆形的物体，接下来要检测的就是与这个物体发生的碰撞。为了区别其他物体，这里将其 name 属性设置为 mybox。

```
//为显示碰撞的力度，建立一个文本对象
label = new LTextField();
backLayer.addChild(label);
```

为了清楚地知道碰撞的力度，建立了一个文本对象将碰撞的力度数值显示到画面上。

```
//加入碰撞侦听事件
LGlobal.box2d.setEvent(LEvent.POST_SOLVE,postSolve);
```

上面代码用 LGlobal.box2d.setEvent 添加了一个碰撞事件，为了取得碰撞的力度，碰撞类型选择的是 LEvent.POST_SOLVE（碰撞后）。

下面来看一下碰撞的侦听函数。

```
//侦听函数
function postSolve(contact, impulse){
            var objA = contact.GetFixtureA().GetBody().GetUserData();
            var objB = contact.GetFixtureB().GetBody().GetUserData();
            if(objA.type == "LSprite" && objA.name == "mybox"){
                        label.text = impulse.normalImpulses[0];
            }else if(objB.type == "LSprite" && objB.name == "mybox"){
                        label.text = impulse.normalImpulses[0];
            }
}
```

这个碰撞侦听函数有两个参数：contact 用来获取碰撞的物体，impulse 用来获取碰撞的力度。因为碰撞是发生在两个物体之间的，所以通过 contact 获取到的对象有两个，分别是 contact.GetFixtureA().GetBody() 和 contact.GetFixtureB().GetBody()。由于 lufylegend 库件通过 LSprite 对象在向物理世界里加入物体的时候，将本身赋值给了物体的 userdata 属性，所以通

过 GetUserData() 函数，可以从物体中取得相应的 LSprite 对象，然后通过获取的对象物体的 type 属性和 name 属性来判断，碰撞是否发生及碰撞的物体中是否有我们指定的物体，如果有，则通过 impulse 来获取碰撞力度，并将这个碰撞力度数值显示到画面上。

9.8 镜头移动

玩过愤怒的小鸟的读者应该知道，游戏中的镜头会始终跟随着发射出去的小鸟。但是在 Box2d 中，镜头默认是不会动的，就是说，当物体移动到镜头以外的时候，虽然物体依然存在，但是从画面上是看不到的。在实际应用中，我们通常会让镜头跟随着某一物体一起运动，当物体移动的时候，镜头也随之移动，这样才能始终让物体显示在画面上。具体做法如代码清单 9-20 所示。

代码清单 9-20

```
<script type="text/javascript">
var backLayer,cLayer,label;
function main(){
            LGlobal.setDebug(true);
            backLayer = new LSprite();
            addChild(backLayer);

            LGlobal.box2d = new LBox2d();
            cLayer = new LSprite();
            cLayer.x = 0;
            cLayer.y = 0;
            backLayer.addChild(cLayer);
            //通过顶点坐标数组，来加入上下左右4面墙
            var shapeArray = [
                [[0,0],[800,0],[800,10],[0,10]],
                [[800,0],[800,400],[790,400],[790,10]],
                [[800,400],[0,400],[0,390],[790,390]],
                [[0,0],[10,0],[10,400],[0,400]]
            ];
            cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);
            //加入一些矩形物体
            for(var i = 0;i<5;i++){
                        cLayer = new LSprite();
                cLayer.x = Math.random()*10*LGlobal.box2d.drawScale+10;
                cLayer.y = Math.random()*10*LGlobal.box2d.drawScale+10;
                        backLayer.addChild(cLayer);
                var w = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
                var h = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
                        cLayer.addBodyPolygon(w,h,1,5,0.4,0.2);
            }
            //加入一个圆形物体
            box01 = new LSprite();
            box01.x = 250;
```

```
            box01.y = 200;
            backLayer.addChild(box01);
            box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
            //给圆形物体加上鼠标拖拽
            box01.setBodyMouseJoint(true);
            //加入循环侦听事件
            backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
//循环函数
function onframe(){
            backLayer.x = LGlobal.width*0.5 - box01.x;
            if(backLayer.x > 0){
                        backLayer.x=0;
            }else if(backLayer.x < LGlobal.width - 1600){
                        backLayer.x = LGlobal.width - 1600;
            }
            LGlobal.box2d.synchronous();
}
</script>
```

上述代码的运行效果如图 9-18 所示。

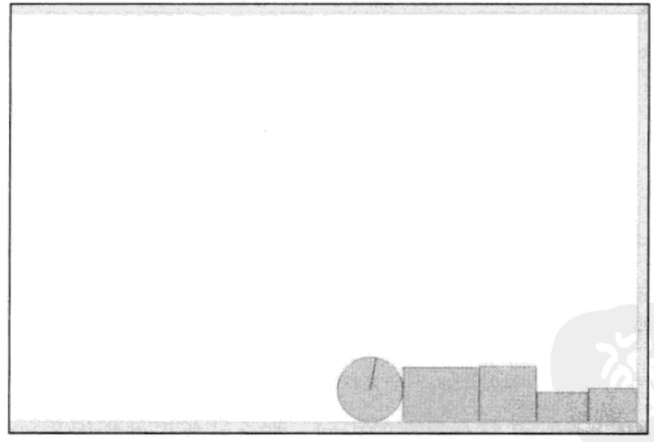


图 9-18 镜头移动效果

代码解析

首先来建立上下左右 4 面墙壁，代码如下所示：

```
//通过顶点坐标数组，来加入上下左右 4 面墙
var shapeArray = [
                [[0,0],[800,0],[800,10],[0,10]],
                [[800,0],[800,400],[790,400],[790,10]],
                [[800,400],[0,400],[0,390],[790,390]],
                [[0,0],[10,0],[10,400],[0,400]]
```

```
            ];
cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);
```

和代码清单 9-19 一样，这里利用顶点数组来同时添加四组矩形作为墙壁。为了实现镜头移动功能，墙壁的大小要大于窗口大小。

```
//加入一些矩形物体
for(var i = 0;i<5;i++){
            cLayer = new LSprite();
            cLayer.x = Math.random()*10*LGlobal.box2d.drawScale+10;
            cLayer.y = Math.random()*10*LGlobal.box2d.drawScale+10;
            backLayer.addChild(cLayer);
            var w = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
            var h = (Math.random()*2 + 0.5)*LGlobal.box2d.drawScale;
            cLayer.addBodyPolygon(w,h,1,5,0.4,0.2);
}
```

上面代码加入一些物体作为参照物，以方便确认镜头是否发生了移动。

```
//加入一个圆形物体
box01 = new LSprite();
box01.x = 250;
box01.y = 200;
backLayer.addChild(box01);
box01.addBodyCircle(30,0,0,1,1,0.5,0.6);
//给圆形物体加上鼠标拖拽
box01.setBodyMouseJoint(true);
```

上面代码加入了一个圆形物体，接下来的操作会让镜头始终跟随着这个圆形物体。

```
//加入循环侦听事件
backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
```

为了让镜头始终跟随一个物体，必须时刻计算这个物体的坐标，所以需要加入循环事件。下面代码是在循环侦听函数中所做的处理。

```
backLayer.x = LGlobal.width*0.5 - box01.x;
```

首先，假设物体 box01 在整个屏幕的中间，那么就要根据物体 box01 的 x 坐标来计算 box01 所在层 backLayer 的 x 坐标，代码如下所示：

```
if(backLayer.x > 0){
            backLayer.x=0;
}else if(backLayer.x < LGlobal.width - 1600){
            backLayer.x = LGlobal.width - 1600;
}
```

当层 backLayer 的 x 坐标超出屏幕的显示上限时，就重新设定层 backLayer 的 x 坐标。

```
LGlobal.box2d.synchronous();
```

在 LGlobal.box2d.synchronous 函数中，可以根据物体所在层的坐标，也就是 backLayer 的坐标，来重新计算 Box2d 世界中所有物体的坐标。这里为了将问题简化，只是让镜头的 x 坐标始终跟随 box01 的 x 坐标，在实际应用过程中，可能也需要进行 y 坐标跟随，不过实现方法是完全一样的。当计算好了物体所在层的 x 坐标和 y 坐标之后，只需要调用 LGlobal.box2d.synchronous 函数，就可以实现物理世界中其他所有坐标的重新计算与显示了。

9.9 做一个简单的物理游戏

有了前面对 Box2d 的介绍，下面来做一个非常简单的物理游戏，如代码清单 9-21 所示。

代码清单 9-21

```
<script type="text/javascript">
var backLayer,cLayer,enemy,gameOver=false;
var point = {x:100,y:200};
function main(){
            LGlobal.box2d = new LBox2d();
            backLayer = new LSprite();
            addChild(backLayer);
            backLayer.graphics.drawRect(1,"#CC3300",[0,0,600,10],true,"#CC3300");
            backLayer.graphics.drawRect(1,"#CC3300",[590,0,10,400],true,"#CC3300");
            backLayer.graphics.drawRect(1,"#CC3300",[0,390,600,10],true,"#CC3300");
            backLayer.graphics.drawRect(1,"#CC3300",[0,0,10,400],true,"#CC3300");
            backLayer.graphics.drawRect(1,"#CC3300",[220,200,10,200],true,"#CC3300");
            backLayer.graphics.drawRect(1,"#CC3300",[230,270,30,10],true,"#CC3300");

            cLayer = new LSprite();
            backLayer.addChild(cLayer);
            //通过顶点坐标数组，来加入上下左右 4 面墙
            var shapeArray = [
                [[0,0],[600,0],[600,10],[0,10]],
                [[600,0],[600,400],[590,400],[590,10]],
                [[600,400],[0,400],[0,390],[590,390]],
                [[0,0],[10,0],[10,400],[0,400]],
                [[220,200],[230,200],[230,400],[220,400]],
                [[230,270],[260,270],[260,280],[230,280]]
            ];
            cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);

            backLayer.graphics.drawArc(1,"#336699",[point.x,point.y,50,0,
2*Math.PI],true,"#336699");

            enemy = new LSprite();
            enemy.name = "enemy";
            enemy.x = 250;
            enemy.y = 240;
            backLayer.addChild(enemy);
            enemy.addBodyPolygon(20,20,1,5,0.4,0.2);
```

```
            enemy.graphics.drawRect(1,"#FF3399",[0,0,20,20],true,"#FF3399");
            //加入鼠标事件
            backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,createBox);
            //加入碰撞侦听事件
            LGlobal.box2d.setEvent(LEvent.POST_SOLVE,postSolve);
}
//侦听函数
function postSolve(contact, impulse){
            if(gameOver)return;
            var objA = contact.GetFixtureA().GetBody().GetUserData();
            var objB = contact.GetFixtureB().GetBody().GetUserData();
            if(objA.type == "LSprite" && objB.type == "LSprite"){
                  if((objA.name == "mybox" && objB.name == "enemy") ||
                        (objA.name == "enemy" && objB.name == "mybox")){
                        gameOver = true;
                        backLayer.removeChild(enemy);
                  }
            }
}

function createBox(event){
            if((event.offsetX - point.x)*(event.offsetX - point.x) +
                  (event.offsetY - point.y)*(event.offsetY - point.y) > 50*50)return;

            var box01 = new LSprite();
            box01.name = "mybox";
            box01.x = event.selfX;
            box01.y = event.selfY;
            backLayer.addChild(box01);
            box01.graphics.drawArc(1,"#000",[10,10,10,0,360*Math.PI/180],true,"#000");
            box01.addBodyCircle(10,0,0,1,1,0.5,0.6);
            var angle = Math.atan2(event.offsetY - point.y,event.offsetX - point.x);
            var force = 200;
            var vec = new LGlobal.box2d.b2Vec2(force*Math.cos(angle),force*Math.sin(angle));
            box01.box2dBody.ApplyForce(vec, box01.box2dBody.GetWorldCenter());
}
</script>
```

上述代码的运行效果如图 9-19 所示。

这是一个利用 Box2d 物理属性制作的一个非常简单的物理游戏，单击左边的圆形区域，会弹出简单的带弹性的小球，如果弹出的小球碰到了中间的小正方体，那么小正方体会消失。

代码解析

可以看到，代码清单 9-21 和前面代码的最大不同是，没有以下的 debug 设定：

```
LGlobal.setDebug(true);
```

因为在正式的游戏中是不需要开启 debug 模式的，所以这里省略了 debug 设定。

首先看看游戏初始化函数 main 中的代码。

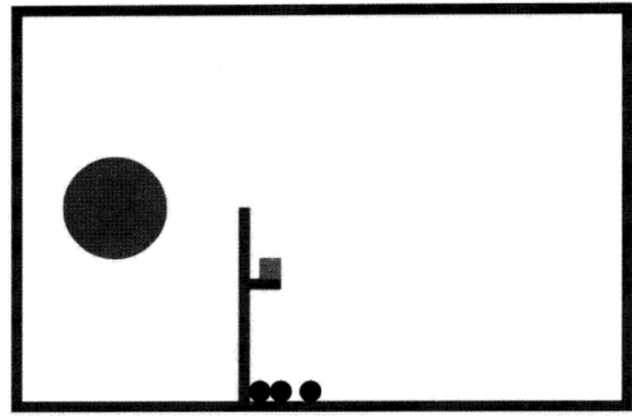


图 9-19　物理游戏效果

```
LGlobal.box2d = new LBox2d();
backLayer = new LSprite();
addChild(backLayer);
```

上面的代码对 Box2d 进行初始化，然后创建了一个 LSprite 对象，作为物理世界一个空间，之后所有的物体都会加到这个空间里。

下面代码建立一组墙壁。

```
backLayer.graphics.drawRect(1,"#CC3300",[0,0,600,10],true,"#CC3300");
backLayer.graphics.drawRect(1,"#CC3300",[590,0,10,400],true,"#CC3300");
backLayer.graphics.drawRect(1,"#CC3300",[0,390,600,10],true,"#CC3300");
backLayer.graphics.drawRect(1,"#CC3300",[0,0,10,400],true,"#CC3300");
backLayer.graphics.drawRect(1,"#CC3300",[220,200,10,200],true,"#CC3300");
backLayer.graphics.drawRect(1,"#CC3300",[230,270,30,10],true,"#CC3300");
cLayer = new LSprite();
backLayer.addChild(cLayer);
//通过顶点坐标数组，加入上下左右 4 面墙
var shapeArray = [
    [[0,0],[600,0],[600,10],[0,10]],
    [[600,0],[600,400],[590,400],[590,10]],
    [[600,400],[0,400],[0,390],[590,390]],
    [[0,0],[10,0],[10,400],[0,400]],
    [[220,200],[230,200],[230,400],[220,400]],
    [[230,270],[260,270],[260,280],[230,280]]
];
cLayer.addBodyVertices(shapeArray,0,0,0,.5,.4,.5);
```

下面的代码在 backLayer 层上画了一个圆，后面的处理会加入鼠标事件，当单击这个圆的时候，就发射小球。

```
backLayer.graphics.drawArc(1,"#336699",[point.x,point.y,50,0,2*Math.PI],true,
"#336699");
```

加入一些目标物体，这个游戏的目标就是让单击鼠标弹出的小球和目标物体接触。

```
enemy = new LSprite();
enemy.name = "enemy";
enemy.x = 250;
enemy.y = 240;
backLayer.addChild(enemy);
enemy.addBodyPolygon(20,20,1,5,0.4,0.2);
enemy.graphics.drawRect(1,"#FF3399",[0,0,20,20],true,"#FF3399");
```

下面代码加入必要的鼠标事件和碰撞侦听事件。

```
//加入鼠标事件
backLayer.addEventListener(LMouseEvent.MOUSE_DOWN,createBox);
//加入碰撞侦听事件
LGlobal.box2d.setEvent(LEvent.POST_SOLVE,postSolve);
```

上面的鼠标事件是用来加入和弹出小球的，而碰撞侦听事件用来检测弹出的小球和目标物体是否发生了碰撞。

碰撞侦听函数代码如下所示：

```
//侦听函数
function postSolve(contact, impulse){
            if(gameOver)return;
            var objA = contact.GetFixtureA().GetBody().GetUserData();
            var objB = contact.GetFixtureB().GetBody().GetUserData();
            if(objA.type == "LSprite" && objB.type == "LSprite"){
                    if((objA.name == "mybox" && objB.name == "enemy") ||
                            (objA.name == "enemy" && objB.name == "mybox")){
                            gameOver = true;
                            backLayer.removeChild(enemy);
                    }
            }
}
```

上面代码的作用是检测发生碰撞的两个物体是否是弹出的小球和目标物体，如果是，则移除目标物体。

鼠标侦听函数代码如下所示：

```
function createBox(event){
            if((event.offsetX - point.x)*(event.offsetX - point.x) +
                        (event.offsetY - point.y)*(event.offsetY - point.y) > 50*50)
return;

            var box01 = new LSprite();
            box01.name = "mybox";
            box01.x = event.selfX;
            box01.y = event.selfY;
            backLayer.addChild(box01);
            box01.graphics.drawArc(1,"#000",[10,10,10,0,360*Math.PI/180],
```

```
true,"#000");
                box01.addBodyCircle(10,0,0,1,1,0.5,0.6);
                var angle = Math.atan2(event.offsetY - point.y,event.offsetX - point.x);
                var force = 200;
                var vec = new LGlobal.box2d.b2Vec2(force*Math.cos(angle),force*Math.
sin(angle));
                box01.box2dBody.ApplyForce(vec, box01.box2dBody.GetWorldCenter());
    }
```

这里首先判断单击的位置是否是左边的圆形区域，根据点击的位置和左边的圆形区域的圆心来计算小球弹出时的角度，然后新加入一个小球，并将其弹飞出去。

以上就是利用 Box2d 来制作的一个非常简陋的小游戏，只要向物理世界里加入需要的物体和力，就可以生成一个生动的画面了。

9.10 小结

利用物理引擎来制作游戏的方便之处就在于只需要设计一个思路，然后按照思路，通过向物理世界中加入一些基本的物体，或者这些基本物体的组合体，并且给这些物体加上相应的力、摩擦等参数，引擎就会自动来完成接下来的工作。甚至不用了解引擎中的碰撞、摩擦等原理，因为它们在现实生活中确实存在，只要将真实生活中存在的画面再现到游戏中就可以了。而且，用 lufylegend 库件进一步简化了将真实生活中存在的画面再现到游戏中时稍显繁琐的过程，让开发变得更加简便容易，引擎使用起来更加得心应手。

PDG

第 10 章　开发网络游戏

网络游戏，英文名称为 Online Game，简称网游，也叫在线游戏，是利用计算机网络实现的多人同时在线娱乐交流的游戏。

网络游戏大致可以分为两类：

一类是利用网页浏览器实现的多人互动游戏，这类游戏无需下载游戏客户端，用户打开浏览器就可以进入游戏，所以很适合上班族玩家。这类游戏统称为网页游戏，又叫 Web 游戏、无端网游等。

另一类由游戏公司提供游戏客户端，玩家需要下载游戏客户端才能进入游戏。由于游戏素材全都存在本地客户端，因此游戏加载的速度要比网页游戏快得多。而大家所说的网络游戏也大多是指这一类的网络游戏。

另外，网络游戏的通信方式也有两种，一种是 HTTP 通信，另一种是 Socket 通信。接下来将详细介绍这两种通信方式的区别。

10.1　HTTP 通信

HTTP 协议即超文本传送协议（Hypertext Transfer Protocol），是 Web 联网的基础，是基于 TCP 协议的应用层协议。HTTP 连接采用的是“短连接”方式，先由客户端发送请求，得到服务器回送响应，然后释放连接，即为“短连接”，完成一次这个过程则称为“一次连接”。

网页游戏大多采用 HTTP 通信，因为玩家只是在刷新网页的时候才需要和服务器通信取得当前页面所需要的数据。

10.1.1　如何实现 HTTP 通信

JavaScript 与后台服务器进行通信用的技术是 AJAX，AJAX 不是一种新的编程语言，而是一种用于创建更好、更快、交互性更强的 Web 应用程序的技术。通过 AJAX，JavaScript 可以使用 XMLHttpRequest 对象直接与后台服务器进行通信，也就是说，可以在不刷新页面的情况下来实现与服务器交换数据。比如当一个网页只需要实现局部更新的时候，无需重载整个页面，只要通过 AJAX 来实现局部的更新即可，这样可以减少传输数据，提高网页的显示速度。

因为需要与服务器进行通信，所以实现 HTTP 通信需要解决服务器端和客户端两个问题。下面分别来看看如何实现。

1. 服务器端

服务器的开发，可以使用 Java、.net、PHP 等语言，这里使用 PHP 来构建一个简单的服务器。

首先新建一个 server.php 文件，代码清单 10-1 是 server.php 文件中的代码，它实现了数据的储存和读取。

代码清单 10-1

```
<?php
$name = $_POST["name"];
$mode= $_POST["mode"];
if($mode == "read"){
        $count = read_file($name);
        echo $count;
}else if($mode == "write"){
        $count = read_file($name);
        $message = $count + 1;
        $result = write_file($name,$message);
        echo $result;
}
function read_file($name){
        if(!file_exists("${name}.txt")){
                return 0;
        }
        $file = fopen("${name}.txt","r");
        $filemsg = "";
        while (!feof($file)) {
                $line = fgets($file);
                $filemsg .= $line;
        }
        fclose($file);
        return $filemsg;
}
function write_file($name,$message){
        $file = fopen("${name}.txt","w+");
        $filemsg = "ok";
        if(!fwrite($file,$message)){
                $filemsg = "error";
        }
        fclose($file);
        return $filemsg;
}
?>
```

上面的 PHP 代码其实实现的是记录用户访问某个文件的次数的功能。由于本书主要讲解使用 JavaScript 操作 Canvas 的相关知识，因此对于 PHP 的语法和用法等不详细解说。下面仅针对代码清单 10-1 做一些简单说明。

代码解析

```
<?php
```

PHP 代码以“<?”或“<?php”开始，以“?>”结束。

```
$name = $_POST["name"];
$mode= $_POST["mode"];
```

HTTP 协议定义了与服务器交互的不同方法，GET 和 POST 方法是其中最基本的两种方法。这里的例子使用 POST 方法进行通信。上面的代码表示获取从客户端传来的 name 和 mode 两个参数的值，在 PHP 中定义变量的形式为 $+ 变量名称。

```
if($mode == "read"){
        ......
}else if($mode == "write"){
        ......
}
```

上面是 PHP 中的 if 判断语句，根据变量 $mode 的值进行不同的处理。

当变量 $mode 的值是 read 的时候，将进行如下处理 ：

```
$count = read_file($name);
echo $count;
```

其中，变量 $name 表示文件的名称，这里通过 read_file 函数，获得文件 $name 的访问次数，然后使用 echo 将获得的访问次数输出给客户端。

当变量 $mode 的值是 write 的时候，进行如下处理 ：

```
$count = read_file($name);
$message = $count + 1;
$result = write_file($name,$message);
echo $result;
```

在上面的代码中，首先通过 read_file 函数获得文件 $name 的访问次数，接着将这个次数加 1，然后将计算完的次数通过 write_file 函数进行保存，最后使用 echo 将保存的结果（成功或失败）输出给客户端。

```
function read_file($name){
        if(!file_exists("${name}.txt")){
                return 0;
        }
        $file = fopen("${name}.txt","r");
        $filemsg = "";
        while (!feof($file)) {
                $line = fgets($file);
                $filemsg .= $line;
        }
        fclose($file);
        return $filemsg;
}
```

在上面的代码中，file_exists 函数是用来判断文件 $name 是否存在的，如果不存在则返回次数 0 ；如果存在则使用 fopen 函数打开文件，然后通过 fgets 函数取得保存在文件 $name

中的信息，也就是被保存的次数，最后返回取得的次数。

```
function write_file($name,$message){
    $file = fopen("${name}.txt","w+");
    $filemsg = "ok";
    if(!fwrite($file,$message)){
                $filemsg = "error";
    }
    fclose($file);
    return $filemsg;
}
```

上面使用 fopen 函数打开文件 $name，然后通过 fwrite 函数将变量 $message 中的值保存到文件 $name 中，最后处理结果。

2. 客户端

客户端是由 HTML 代码和 JavaScript 代码组成的，代码清单 10-2 是 HTML 文件中的代码。

代码清单 10-2

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>client</title>
<script type="text/javascript" src="ajax.js"></script>
</head>
<body onload="setClickNum()">
<header>
<h1> HTTP 通信测试 </h1>
</header>
<nav>
<p>
        <input type="button" value=" 按钮 " onclick="addClick()" />
</p>
<p>
按钮被按下次数：<div id="showNum">0</div>
</p>
</nav>
</body>
</html>
```

代码解析

上面的代码通过调用 setClickNum 函数，来显示按钮被按下的次数。当单击按钮的时候会通过 addChild 函数来将这个次数加 1 并保存。这个代码清单只是函数的调用，功能的实现部分都是通过 JavaScript 代码来完成的。

至于与服务器的通信，这里主要是应用 AJAX 技术来实现的，使用 AJAX 交互必须使用

XMLHttp 对象。XmlHttp 提供了一套客户端同 HTTP 服务器通信的协议，它是可以在 JavaScript、VbScript、JScript 等脚本语言中通过 HTTP 协议传送或接收 XML 及其他数据的 API。

XmlHttp 对象的属性如表 10-1 所示。

表 10-1 XmlHttp 对象的属性一览表

属性名	解释
onreadystatechange	指定当 readyState 属性改变时的事件处理句柄，只写
readyState	返回当前请求的状态，只读
responseBody	将回应信息正文以 unsigned byte 数组形式返回，只读
responseStream	以 Ado Stream 对象的形式返回响应信息，只读
responseText	将响应信息作为字符串返回，只读
responseXML	将响应信息格式化为 Xml Document 对象并返回，只读
status	返回当前请求的 HTTP 状态码，只读
statusText	返回当前请求的响应行状态，只读

再看 XmlHttp 对象的方法，如表 10-2 所示。

表 10-2 XmlHttp 对象的方法一览表

属性名	解释
abort	取消当前请求
getAllResponseHeaders	获取响应的所有 HTTP 头
getResponseHeader	从响应信息中获取指定的 HTTP 头
open	创建一个新的 HTTP 请求，并指定此请求的方法、URL 以及验证信息（用户名和密码）
send	发送请求到 HTTP 服务器并接收回应
setRequestHeader	单独指定请求的某个 HTTP 头

以上两个表介绍了 XmlHttp 对象的所有属性和方法，不用一一理解，因为有些属性一般是用不到的，有个大致印象就行。在代码清单 10-3 中将使用 XmlHttp 对象与服务器进行信息的交互。

代码清单 10-3

```
function setClickNum(){
var obj = document.getElementById("showNum");
var url = "./server.php";
var data = "mode=read&name=countnum";
var oncomplete = function(value){
obj.innerHTML = value;
}
getRequest(url,data,oncomplete);
}
function addClick(){
var url = "./server.php";
```

```
var data = "mode=write&name=countnum";
var oncomplete = function(value){
var obj = document.getElementById("showNum");
obj.innerHTML = parseInt(obj.innerHTML) + 1;
}
getRequest(url,data,oncomplete);
}
function getRequest(url, data, oncomplete) {
var ajax = GetHttpObject();
    if (!ajax) {
        alert("no");
    }
    ajax.open("POST", url, true);
ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
    ajax.onreadystatechange = function(){
if (ajax.readyState == 4 && ajax.status == 200){
if (ajax.responseText.length > 0){
oncomplete(ajax.responseText);
}
         }
};
    ajax.send(data);
}
function GetHttpObject(){
    if(typeof XMLHttpRequest != 'undefined'){
        return new XMLHttpRequest();
    }
    try{
        return new ActiveXObject("Msxml2.XMLHTTP");
    }catch (e){
        try{
            return new ActiveXObject("Microsoft.XMLHTTP");
        }catch (e) {}
    }
    return false;
}
```

代码解析

首先来看一看 GetHttpObject 函数。

```
function GetHttpObject(){
    if(typeof XMLHttpRequest != 'undefined'){
        return new XMLHttpRequest();
    }
    try{
        return new ActiveXObject("Msxml2.XMLHTTP");
    }catch (e){
        try{
            return new ActiveXObject("Microsoft.XMLHTTP");
        }catch (e) {}
```

```
    }
    return false;
}
```

现在的绝大多数浏览器都增加了对 XmlHttp 的支持，几乎所有浏览器均支持 XMLHttp-Request 对象，只有 IE5 和 IE6 使用 ActiveXObject 对象。GetHttpObject 函数实现了所有浏览器的兼容，当 XMLHttpRequest 对象不存在的时候，才会创建 ActiveXObject 对象。

```
function getRequest(url, data, oncomplete) {
var ajax = GetHttpObject();
    if (!ajax) {
        alert("no");
    }
    ajax.open("POST", url, true);
ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
    ajax.onreadystatechange = function(){
if (ajax.readyState == 4 && ajax.status == 200){
if (ajax.responseText.length > 0){
oncomplete(ajax.responseText);
}
         }
};
    ajax.send(data);
}
```

上面代码首先获取一个可以与服务器交互的对象，然后通过 POST 方法与服务器进行通信，其中变量 url 是服务器地址，变量 data 是传递给服务器的参数，变量 oncomplete 是与服务器通信完成后所调用的函数。从表 10-1 我们可以知道，readyState 表示返回当前请求的状态，它的值是 4 的时候表示通信完成。至于其他各个值的含义如表 10-3 所示。

表 10-3 readyState 对应含义

属性名	解释
0（未初始化）	对象已建立，但是尚未初始化（尚未调用 open 方法）
1（初始化）	对象已建立，尚未调用 send 方法
2（发送数据）	send 方法已调用，但是当前的状态及 HTTP 头未知
3（数据传送中）	已接收部分数据，因为响应及 HTTP 头不全，这时通过 responseBody 和 responseText 获取部分数据会出现错误
4（完成）	数据接收完毕，此时可以通过 responseBody 和 responseText 获取完整的回应数据

```
function setClickNum(){
var obj = document.getElementById("showNum");
var url = "./server.php";
var data = "mode=read&name=countnum";
var oncomplete = function(value){
obj.innerHTML = value;
}
getRequest(url,data,oncomplete);
}
```

上面代码中，setClickNum 函数表示通过与服务器进行通信得到按钮被单击的次数，然后将取得的结果显示到页面上。

```
function addClick(){
var url = "./server.php";
var data = "mode=write&name=countnum";
var oncomplete = function(value){
var obj = document.getElementById("showNum");
obj.innerHTML = parseInt(obj.innerHTML) + 1;
}
getRequest(url,data,oncomplete);
}
```

在上面的代码中，addClick 函数表示通过与服务器进行通信来保存按钮被单击的次数，然后将单击后的次数显示到页面上。

从代码清单 10-1 到代码清单 10-3 实现了一个简单的 http 通信。将书中的源码中的文件夹 10-1（包括文件夹里面的文件）复制到 xampp 的 htdocs 目录下，然后通过访问 http://localhost/10-1/http.html，可以得到如图 10-1 所示的运行效果。

HTTP 通信测试

按钮

按钮被按下次数:

2

图 10-1 HTTP 通信运行效果

10.1.2 HTTP 通信的弊端

因为 HTTP 通信只有在访问或刷新网页的时候，才能和服务器进行通信，所以当它想要处理即时通信的功能（比如 MSN、QQ 或者在线聊天室等）或者像 MMRPG 等的网络游戏时，就需要不断地向服务器发出数据请求，不断地进行通信，这种技术通常被称为“轮询”，但是这种方式会大量消耗服务器的带宽和资源。

所以，当客户端和服务器需要即时通信时，就需要用到另外一种技术了，即 Socket 通信。

10.2 Socket 通信

上面提到了使用“轮询”技术来实现即时通信时，会大量消耗服务器带宽和资源，针对这种情况，HTML5 中定义了 WebSocket 协议，能节省服务器资源和带宽，并实现实时通信。

什么是 Socket 呢？应用层通过传输层进行数据通信时，TCP 和 UDP 会遇到同时为多个应用程序进程提供并发服务的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP/IP 协议交互提供了称为套接字（Socket）的接口，从而区分不同应用程序进程间的网络通信和连接。Socket 接口是 TCP/IP 网络的 API，它定义了许多函数或例程，程序员可以用它们来开发 TCP/IP 网络上的应用程序。

网上对 Scoket 的定义有很多种说法，但是对于初学者来说，这些定义都太过于专业，下面做一个比喻，让大家更容易理解什么是 Socket。

Socket 和现实世界中的邮寄系统有些类似，在邮寄系统中，寄信的时候，寄信人会将信函投到邮箱中，然后通过邮局传送给收信人。寄信人可以同时给多个人寄信，收信人也可以同时收到很多封来信。Socket 通信也是如此，客户端向服务端发送信息，就相当于寄信这一过程，而服务端向客户端发送信息，就相当于邮局将信件传送给收信人。

下面的流程图很形象地说明了上面的通信过程。

寄信人→邮箱→邮局→邮递员→收件人

客户端→ Socket →服务器→ Socket →客户端

10.2.1 区分 Socket 通信和 HTTP 通信

为了让大家彻底明白什么是 Socket 通信，什么是 HTTP 通信，下面再举一个例子。

小明和小刚预约放学后一起回家，可是小刚的作业还没有做完，小明要等小刚做完作业后才能跟他一起回家。

如果是 HTTP 通信的话，那么小明要每隔一段时间（假设为 5 分钟）询问一下小刚。过程如下：

```
小明：小刚，你作业做完了吗？
小刚：还没有，你再稍等一会儿。
5 分钟后
小明：小刚，你作业做完了吗？
小刚：还没有，你再稍等一会儿。
5 分钟后
小明：小刚，你作业做完了吗？
小刚：还没有，你再稍等一会儿。
……
小明：小刚，你作业做完了吗？
小刚：终于做完了。
小明：我们一起回家吧。
```

如果是 Socket 通信的话，那么小明只需要等小刚做完作业后叫他就可以了。过程如下：

```
小明：小刚，你作业做完了告诉我一声。
……
小刚：小明，我的作业做完了。
小明：我们一起回家吧。
```

可以看到，使用 Socket 通信的时候，小明只需要等小刚作业做完后叫他就可以了。因为小刚作业做完了会主动告诉小明。而使用 HTTP 通信时，小明需要不停地问小刚作业是否已经做完，因为小刚只会在小明问自己的时候才会告诉小明作业做完与否。

10.2.2 服务器端

在 HTML5 中的 Socket 通信要用到 WebSocket，下面就来认识一下它。

1. WebSocket

WebSocket 是 Socket 中的一种，它是新加入 HTML5 中的一项功能，是一种浏览器与

服务器间进行双向通信的网络技术。在 WebSocket API 中，浏览器和服务器只需要做一个握手的动作，浏览器和服务器之间就形成了一条快速通道，两者之间就可以直接进行数据互传了。在此之前，要在网页上实现与服务器的数据互传，恐怕只能借助 Flash 或者 JavaApplet 等技术了。

2. Jetty

Jetty 是使用 Java 语言编写的一个开源的 servlet 容器，它为基于 Java 的 Web 内容（如 JSP 和 servlet）提供运行环境，可用于快速构建服务器。其实 WebSocket 服务器的构建可以使用 Java 或者 node.js 等多种语言。这里将使用 Jetty 这个开源容器来构建 WebSocket 服务器，这样我们就不需要从零开始构建服务器了。Jetty 的 API 以一组 JAR 包的形式发布，开发人员可以将 Jetty 容器实例化成一个对象，这样就可以迅速地为一些独立运行（stand-alone）的 Java 应用提供网络和 Web 连接。

本次使用的 Jetty 版本是 8.1.7，下载地址如下：

http://dist.codehaus.org/jetty/jetty-hightide-8.1.7/

如果用户使用的是 Windows 环境，只要先下载 jetty-hightide-8.1.7.v20120910.zip 文件，然后解压缩即可得到 Jetty 的 jar 类库。

3. 使用 Jetty 快速建立一个简单的服务器

下面详细说明如何在 Eclips 下使用 Jetty。

首先，使用 Eclips 创建一个 Java 工程，如图 10-2 所示。

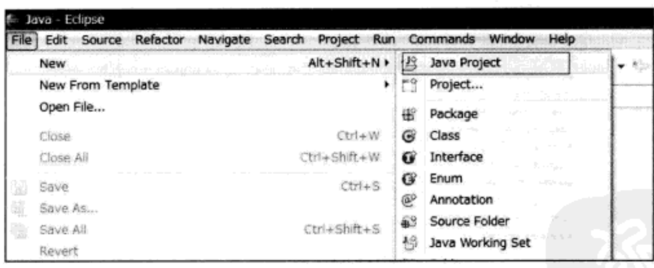


图 10-2　创建 Java 工程

然后给工程命名，可以起任意自己喜欢的名字，如图 10-3 所示。

输入工程名之后，单击“完成”按钮（Finish）。工程建立完之后，需要引进 Jetty 的库文件，鼠标右击新建的工程，选择“属性”（Properties）选项，如图 10-4 所示。

在弹出的“属性”界面中，在左边的菜单中选择 Java Build Path 选项，然后在弹出的菜单中选择 Libraries 选项卡，如图 10-5 所示。

单击右边的 Add External JARs 按钮，在弹出的选择框中，在解压缩的 Jetty 的 lib 文件夹中选择所有扩展名为 jar 的文件，如图 10-6 所示。

New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: websocket_sample

Use default location

Location: D:\eclipse-java-indigo-SR2-win32\eclipse\object\websocket_sample Browse...

JRE

Use an execution environment JRE: JavaSE-1.7

Use a project specific JRE: jre7

Use default JRE (currently 'jre7') Configure JREs...

Project layout

Use project folder as root for sources and class files

Create separate folders for sources and class files Configure default...

< Back Next > Finish Cancel

图 10-3 给工程命名

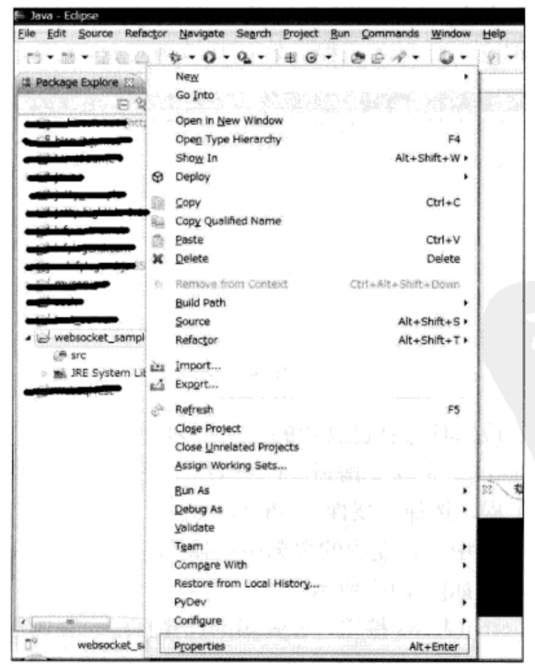


图 10-4 选择“属性”选项

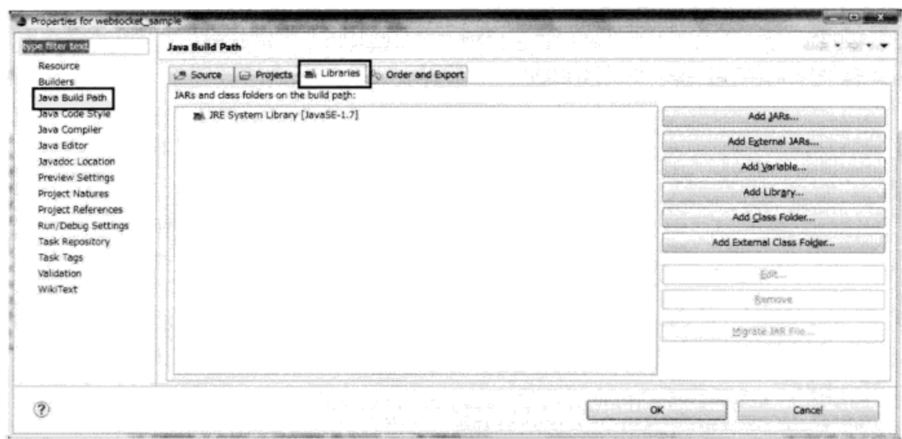


图 10-5　选择 Java Build Path 选项卡

图 10-6　选择 jar 文件

注意，图 10-6 是在日文系统下的图片，中文系统下按钮和列表等的名称和图 10-6 是不一样的。将所有类库引入完成后，其界面如图 10-7 所示。

最后单击 OK 按钮，准备工作就完成了。

将 Jetty 引入工程后，就可以开始建立 WebSocket 服务器了。打开本书源码中第 10 章 /10.2 文件夹，将里面的 SampleServer.java、SampleSocket.java、Script.java、User.java 这 4 个 .java 文件复制到刚才新建的工程里的 src 文件夹中，如图 10-8 所示。

复制完成之后，Eclips 的新建工程如图 10-9 所示。

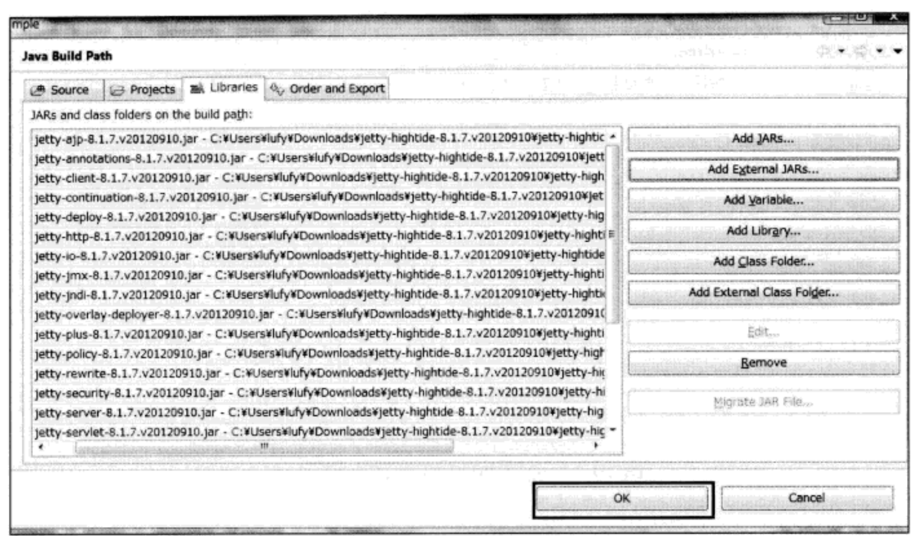


图 10-7 引入 Jetty 完成

图 10-8 复制 .java 文件

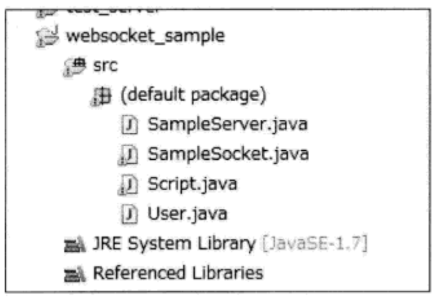


图 10-9 复制 .java 文件后工程文件夹内容

这 4 个 .java 文件是本书中 WebSocket 服务器的源代码，本书不涉及 Java 的内容讲解，大家可以查阅 Java 相关书籍进行学习。

4. 运行服务器

鼠标右击上面新建的 Java 工程中的 SampleServer.java 文件，在弹出的菜单中选择“运行”（Run As）选项，然后选择以 Java Application 方式运行，如图 10-10 所示。

单击“运行”后，如果得到如图 10-11 所示显示，表示服务器已经启动。

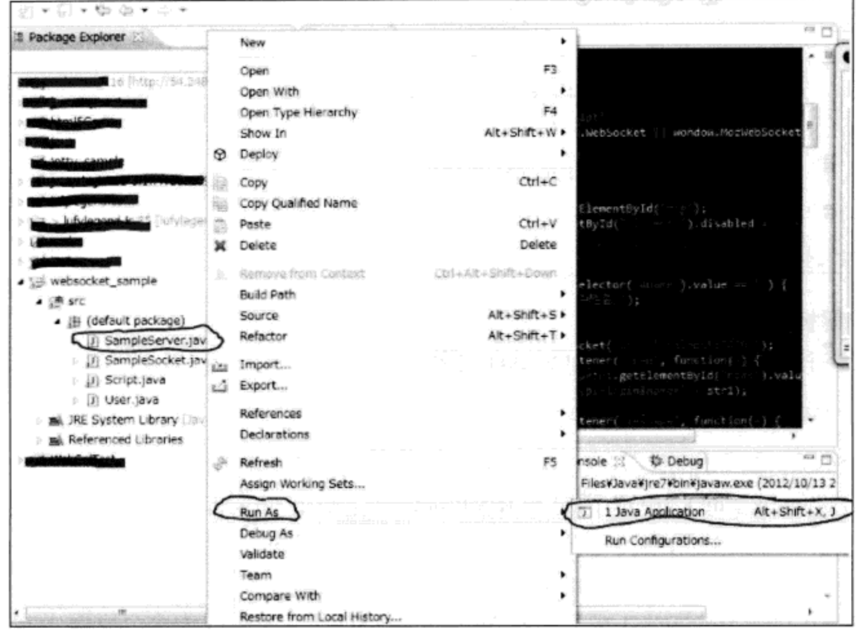


图 10-10　运行服务器菜单

Problems　Javadoc　Declaration　Console　Debug

SampleServer (1) [Java Application] C:¥Program Files¥Java¥jre7¥bin¥javaw.exe (2012/10/13 22:21:47)

2012-10-13 22:21:48.642:INFO:oejs.Server:jetty-8.1.7.v20120910
2012-10-13 22:21:48.704:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8800

图 10-11　服务器启动效果

10.2.3　客户端

在 HTML5 中建立 WebSocket 连接非常简单。代码清单 10-4 说明了如何建立一个 WebSocket 连接，并且为 WebSocket 添加了相应的侦听事件。

代码清单　10-4

```
<script type="text/javascript">
        var WebSocket = window.WebSocket || wondow.MozWebSocket;
        var socket = new WebSocket("ws://localhost:8800/");
        socket.addEventListener('open', function(e) {
                        // 代码 ......
        });
```

PDG

```
            socket.addEventListener('message', function(e) {
                        // 代码 ......
            });
            socket.addEventListener('error', function(e) {
                        // 代码 ......
            });
            socket.addEventListener('close', function(e) {
                        // 代码 ......
            });
</script>
```

代码解析

首先得到浏览器所支持的 WebSocket 对象。

```
var WebSocket = window.WebSocket || wondow.MozWebSocket;
```

WebSocket 是基于浏览器的，所以页面运行的当前浏览器必须支持 HTML5，并且支持 WebSocket。Google 的 Chrome 浏览器在 Chrome 5 之后的版本都支持 WebSocket，但由于 WebSocket 还没有最终版本，草案在不断更新，因此不同的版本会支持不同的草案。Apple 公司的 Safari 浏览器也支持 WebSocket。虽然 FireFox 也支持 WebSocket，但是在 FireFox 中，从版本 6 开始 WebSocket 被更名为 MozWebSocket 了。也是因为这个原因，上面的代码在得到 WebSocket 对象的时候，使用了 window.WebSocket 和 window.MozWebSocket 两种方式。

在得到了当前浏览器所支持的 WebSocket 对象后，就可以通过下面的代码来连接服务器了。

```
var socket = new WebSocket("ws://localhost:8800/");
```

因为在建立服务器的时候，端口设定成了 8800，所以这里需要使用 8800 端口来连接服务器。在建立 Socket 连接时，可以为这个连接添加侦听事件，代码如下所示：

```
socket.addEventListener('open', function(e) {
        // 代码 ......
});
```

上面的代码表示给 Socket 连接添加侦听 open 事件，并在接通服务器时运行方法中的内容。

```
socket.addEventListener('message', function(e) {
        // 代码 ......
});
```

上面的代码表示给 Socket 连接添加侦听 message 事件，并在客户端接收到服务器发来的消息时运行方法中的内容。

```
socket.addEventListener('error', function(e) {
```

```
        // 代码 ......
});
```

上面的代码表示给 Socket 连接添加侦听 error 事件，并在客户端与服务器连接过程中，发生异常时运行方法中的内容。

```
socket.addEventListener('close', function(e) {
        // 代码 ......
});
```

上面的代码表示给 Socket 连接添加侦听 close 事件，并在接客户端与服务器断开连接时运行方法中的内容。

这样，一个最简单的客户端就完成了。

10.3　利用 WebSocket 实现简单的聊天室

上面讲了 WebSocket 的服务器和客户端的实现方法，接下来试着利用这些方法来建立一个简单的聊天室。首先建一个代码客户端的页面，如代码清单 10-5 所示。

代码清单　10-5

```
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body onload="initial();">
    <header>
        <h1>WebSocket 聊天室 </h1>
    </header>
    <article>
        <input type="button" id="openbtn" onclick="doOpen();" value=" 连接服务器 " />
        <input type="button" id="closebtn" onclick="doClose();" value=" 断开连接 " />
        <hr>
        <form onsubmit="return doAction();">
            <table>
                <tr>
                    <th> 用户 :</th>
                    <td>
                        <input type="text" id="name" size="10">
                    </td>
                </tr>
                <tr>
                <th> 消息 :</th>
                    <td>
                        <select id="to">
                            <option value="all"> 所有人 </option>
                        </select>
                        <br />
```

```
                        <input type="text" id="message" size="40">
                        <input type="submit" value=" 发送 ">
                    </td>
                </tr>
            </table>
        </form>
        <hr>
        <ul id="msg"></ul>
    </article>
</body>
</html>
```

代码解析

代码清单 10-5 是一个 HTML 页面，其运行效果如图 10-12 所示。

```
<body onload="initial();">
```

上面代码表示当页面加载完成时触发 initial 方法。

图 10-12　聊天室 HTML 页面

```
<input type="button" onclick="doOpen();
" value=" 连接服务器 " />
<input type="button" onclick="doClose();" value=" 断开连接 " />
```

上面代码建立了两个按钮来控制服务器的连接和断开。当单击“连接服务器”按钮的时候，调用 doOpen 方法连接服务器，当单击“断开连接”按钮的时候，调用 doClose 方法将与服务器之间的连接断开。

```
<input type="text" id="name" size="10">
```

上面这个 text 文本框用来填写用户名称。

```
<select id="to">
<option value="all">所有人</option>
</select>
```

上面是一个下拉列表，用来显示已经进入聊天室的用户列表。

```
<input type="text" id="message" size="40">
```

上面这个 text 文本框用来输入聊天内容。

```
<input type="submit" value=" 发送 ">
```

上面是一个提交按钮，表示当单击这个按钮的时候，发送聊天内容。在 form 表单的标签中添加了 onsubmit 事件，代码如下所示：

```
<form onsubmit="return doAction();">
```

这个事件表示当 form 表单的提交事件触发时，调用 doAction 方法，向服务器发送信息。

HTML 代码用来显示页面，和服务器进行的通信都是由 JavaScript 代码完成的。下面开始写 JavaScript 代码，以完成与服务器之间的通信。在代码清单 10-5 中曾提到，当页面加载完成时会触发 initial 方法，下面就先来看一下 initial 方法。

```
function initial() {
    msg = document.getElementById("msg");
    document.getElementById("closebtn").disabled = true;
}
```

上面代码为了之后的操作方便，建立了一个变量 msg 来控制消息的显示。因为客户端与服务器的连接状态要么为连接，要么为断开，即“连接服务器”和“断开连接”这两个按钮应该只能有一个是可以被单击的，所以当页面加载完成时，首先将“断开连接”按钮变为不可用，如图 10-13 所示。

图 10-13 “断开连接”按钮不可用

当用户单击“连接服务器”按钮的时候，会调用 doOpen 方法来连接服务器。doOpen 方法的用法如代码清单 10-6 所示。

代码清单 10-6

```
function doOpen() {
    if(document.querySelector('#name').value == "") {
        alert('请输入用户姓名。');
        return;
    }
    socket = new WebSocket("ws://localhost:8800/");
    socket.addEventListener('open', function(e) {

    });
    socket.addEventListener('message', function(e) {

    });
    socket.addEventListener('error', function(e) {

    });
    socket.addEventListener('close', function(e) {

    });
    msg.innerHTML = "<li>连接服务器成功</li>" + msg.innerHTML;
    document.getElementById("name").disabled = true;
    document.getElementById("openbtn").disabled = true;
    document.getElementById("closebtn").disabled = false;
}
```

在前面的代码清单 10-4 中曾说明了如何建立一个 WebSocket 连接，代码清单 10-6 首先判断用户姓名是否输入，然后才进行连接。在服务器连接成功之后会将“连接服务器”和“断开连接”这两个按钮的状态进行互换，如图 10-14 所示。

图 10-14 “连接服务器”按钮不可用

上面的代码中，为 Socket 连接添加了 open、message、error 和 close 这 4 个侦听事件，分别在下面的情况下会触发：与服务器连接成功、接收服务器信息、发生异常、与服务器连接断开。下面依次来完成这 4 个侦听事件触发时所需要处理的内容。首先来看看 open 侦听事件，代码如下所示：

```
socket.addEventListener('open', function(e) {
    var str1 = document.getElementById("name").value;
    socket.send("type=login&name=" + str1);
});
```

在该侦听事件中，send 方法用来向 Socket 服务器发送信息，发送的内容是字符串。上面的代码表示当接通服务器时，将用户名称发送给服务器进行登录。本例中所用的所有和服务器进行通信的字符串，都是采用类似 URL 参数的形式来进行传送的，这些都是在服务器端已经设定好的，如果自己写服务器端的时候，可以自由地设定客户端与服务器之间通信时的字符串形式。

```
socket.addEventListener('error', function(e) {
    alert("error!!");
});
```

上面代码表示在客户端与服务器连接过程中，发生异常时弹出提示窗口，提示信息为“error!!”。

```
socket.addEventListener('close', function(e) {
    msg.innerHTML = "<li> 服务器切断 </li>" + msg.innerHTML;
    document.getElementById("name").disabled = false;
    document.getElementById("openbtn").disabled = false;
    document.getElementById("closebtn").disabled = true;
});
```

上面代码表示当接客户端与服务器断开时，显示信息“服务器切断”，并且交换“连接服务器”和“断开连接”这两个按钮的状态。

代码清单 10-7 可以说是这个聊天室的核心部分，它要处理从服务器传来的各种信息。

代码清单 10-7

```
socket.addEventListener('message', function(e) {
    var value = getScript(e.data);
    var text;
```

```
    switch(value["result"]) {
        case "error":
            text = "<li>" + value["error"] + "</li>" + msg.innerHTML;
            break;
        case "loginok":
            document.getElementById("name").value = value["name"];
            text = "<li>" + " 登录成功 " + "</li>" + msg.innerHTML;
            break;
        case "talk":
            text = "<li>" + value["msg"] + "</li>" + msg.innerHTML;
            break;
        case "removeuser":
            removeUser(value["name"]);
            break;
        case "setuserlist":
            removeAllUser();
            var list = value["list"].split(",");
            for(var i=0,l=list.length;i<l;i++){
                addUser(list[i]);
            }
            break;
    }
    if(text)msg.innerHTML = text;
});
```

代码解析

下面的代码用于分解字符串，其中 e.data 表示获取从服务器传来的字符串信息，getScript 函数表示将字符串进行分解。

```
var value = getScript(e.data);
```

其中，getScript 函数的具体内容如下所示：

```
function getScript(value) {
    var valueArray = value.split("&");
    var arr;
    var scriptObj = {};
    for(var i in valueArray) {
        arr = valueArray[i].split("=");
        scriptObj[arr[0]] = arr[1];
    }
    return scriptObj;
}
```

因为 getScript 中的字符串格式是以“name1=value1&name2=value2”的形式传入的，所以可以使用 & 和 = 将字符串一一分解成为数组，以便于操作。

将服务器传来的字符串信息进行分解后，用 switch 语句将信息进行分类并解析。

```
        case "error":
```

```
            text = "<li>" + value["error"] + "</li>" + msg.innerHTML;
            break;
```

当服务器传来的信息为“result=error&error= 错误信息”时，表示出现错误。上面的代码用于将错误信息显示出来。

```
        case "loginok":
            document.getElementById("name").value = value["name"];
            text = "<li>" + " 登录成功 " + "</li>" + msg.innerHTML;
            break;
```

当服务器传来的信息为“result=loginok&name= 姓名”时，表示登录成功，此时如果用户输入的名称与已经登录的用户重复，那么服务器会在用户名称上添加一个随机的数字，用于区分用户。上面的代码表示将登录成功的用户名重新设定，然后显示“登录成功”。

```
        case "talk":
            text = "<li>" + value["msg"] + "</li>" + msg.innerHTML;
            break;
```

当服务器传来的信息为“result=talk&msg= 聊天内容”时，表示有用户发送了聊天内容。上面的代码就表示将用户发送的聊天内容显示到页面上。

```
        case "setuserlist":
            removeAllUser();
            var list = value["list"].split(",");
            for(var i=0,l=list.length;i<l;i++){
                addUser(list[i]);
            }
            break;
```

当服务器传来的信息为“result=setuserlist&list= 用户 1, 用户 2, 用户 3”时，表示因为有用户进入或离开聊天室，而需要刷新本地用户列表。上面的代码用于重新设定用户列表。其中的 removeAllUser 函数用于移除用户列表中的所有用户，addUser 函数用于向用户列表中添加用户。

removeAllUser 函数的实现方法如下：

```
function removeAllUser() {
    var list = document.getElementById("to");
    for(var i = 1,l=list.options.length; i < l; i++) {
        list.options.remove(1);
    }
}
```

addUser 函数的实现方法如下：

```
function addUser(username) {
    var list = document.getElementById("to");
    if(!isExitUser(list, username)) {
        var item = new Option(username, username);
```

```
            list.options.add(item);
        }
    }
```

在 addUser 函数中的 isExitUser 函数用于判断用户是否存在于用户列表中，其实现方法如下：

```
function isExitUser(username) {
    var list = document.getElementById("to");
    var isExit = false;
    for(var i = 0; i < list.options.length; i++) {
        if(list.options[i].value == username) {
            isExit = true;
            break;
        }
    }
    return isExit;
}
```

下面接着对解析服务器信息部分进行讲解。

```
case "removeuser":
            removeUser(value["name"]);
            break;
```

当服务器传来的信息为“result=removeuser&name= 用户名称 " 时，表示有用户离开聊天室。上面的代码用于移除离开的用户。

以上就是处理服务器传送到客户端的信息的方法。下面接着对页面的 JavaScript 进行完善。

当用户单击“断开连接”按钮的时候，会调用 doClose 函数。下面是 doClose 函数的实现。

```
function doClose() {
    if(socket.readyState == WebSocket.OPEN) {
        socket.close();
    }
}
```

上面的代码是使用 Socket 的 close 方法来断开 Socket 连接的。

最后，来实现用来发送信息的 doAction 函数，代码如下：

```
function doAction() {
    if(socket.readyState == WebSocket.OPEN) {
        var to = document.getElementById("to").value;
        var msg = document.getElementById("message").value;
        socket.send("type=talk&target="+to+"&msg="+msg);
        document.getElementById("message").value = "";
    } else {
        alert('连接服务器失败。');
```

```
    }
    return false;
}
```

在 doAction 函数中，首先要判断一下 Socket 连接是否为打开，如果 Socket 连接的状态为打开，那么首先确定对谁发送信息，然后使用 Socket 连接的 send 函数来向服务器发送信息，发送的信息格式为“type=talk&target= 接收对象 &msg= 发送内容”。

代码清单 10-8 是上述聊天室页面的完整 JavaScript 代码。

代码清单 10-8

```
<script type="text/javascript">
var WebSocket = window.WebSocket || wondow.MozWebSocket;
var socket;
var msg;
function initial() {
    msg = document.getElementById("msg");
    document.getElementById("closebtn").disabled = true;
}
function doOpen() {
    if(document.querySelector('#name').value == "") {
        alert('请输入用户姓名。');
        return;
    }
    socket = new WebSocket("ws://localhost:8800/");
    socket.addEventListener('open', function(e) {
        var str1 = document.getElementById("name").value;
        socket.send("type=login&name=" + str1);
    });
    socket.addEventListener('message', function(e) {
        var value = getScript(e.data);
        var text;
        switch(value["result"]) {
            case "error":
                text = "<li>" + value["error"] + "</li>" + msg.innerHTML;
                break;
            break;
                case "loginok":
                document.getElementById("name").value = value["name"];
                text = "<li>" + "登录成功" + "</li>" + msg.innerHTML;
                break;
            case "talk":
                text = "<li>" + value["msg"] + "</li>" + msg.innerHTML;
                break;
            case "removeuser":
                removeUser(value["name"]);
                break;
            case "setuserlist":
                removeAllUser();
                var list = value["list"].split(",");
```

```
                for(var i=0,l=list.length;i<l;i++){
                    addUser(list[i]);
                }
                break;
        }
        if(text)msg.innerHTML = text;
    });
    socket.addEventListener('error', function(e) {
        alert("error!!");
    });
    socket.addEventListener('close', function(e) {
        msg.innerHTML = "<li>服务器切断</li>" + msg.innerHTML;
        document.getElementById("name").disabled = false;
        document.getElementById("openbtn").disabled = false;
        document.getElementById("closebtn").disabled = true;
    });
    msg.innerHTML = "<li>连接服务器成功</li>" + msg.innerHTML;
    document.getElementById("name").disabled = true;
    document.getElementById("openbtn").disabled = true;
    document.getElementById("closebtn").disabled = false;
}
function getScript(value) {
    var valueArray = value.split("&");
    var arr;
    var scriptObj = {};
    for(var i in valueArray) {
        arr = valueArray[i].split("=");
        scriptObj[arr[0]] = arr[1];
    }
    return scriptObj;
}
function doAction() {
    if(socket.readyState == WebSocket.OPEN) {
        var to = document.getElementById("to").value;
        var msg = document.getElementById("message").value;
        socket.send("type=talk&target="+to+"&msg="+msg);
        document.getElementById("message").value = "";
    } else {
        alert('连接服务器失败。');
    }
    return false;
}
function doClose() {
    if(socket.readyState == WebSocket.OPEN) {
        socket.close();
    }
}
function addUser(username) {
    var list = document.getElementById("to");
    if(!isExitUser(list, username)) {
        var item = new Option(username, username);
```

```
            list.options.add(item);
        }
    }
    function removeUser(username) {
        var list = document.getElementById("to");
        for(var i = 0; i < list.options.length; i++) {
            if(list.options[i].value == username) {
                list.options.remove(i);
                break;
            }
        }
    }
    function removeAllUser() {
        var list = document.getElementById("to");
        for(var i = 1,l=list.options.length; i < l; i++) {
            list.options.remove(1);
        }
    }
    function isExitUser(username) {
        var list = document.getElementById("to");
        var isExit = false;
        for(var i = 0; i < list.options.length; i++) {
            if(list.options[i].value == username) {
                isExit = true;
                break;
            }
        }
        return isExit;
    }
    </script>
```

现在，运行服务器，然后同时打开 3 个网页，输入不同的用户名连接服务器。当用户列表选择所有人的时候，表示发送的信息是所有人可见的，如图 10-15 所示。

图 10-15 聊天室中对所有人发言

可以看到，向所有人发送信息的时候，发送的信息所有用户都可以看到。如果在用户列表中选择个别用户进行发送，那么发送的信息只有指定用户才能看到，效果如图 10-16 所示。

图 10-16　聊天室中对指定用户发言

这样，利用 WebSocket 制作的一个简易的聊天室就完成了。

10.4　做一款多人在线的坦克大战

前面利用 WebSocket 制作了一个简易的聊天室，实现了客户端与服务器之间的即时通信。接下来我们试做一款多人在线的小游戏，来了解一下 WebSocket 在游戏方面的应用。

相信大家都玩过任天堂 FC 平台上的坦克大战，玩家控制一辆坦克，为了保卫基地不被摧毁而展开战斗。游戏中敌方的坦克是由电脑控制的，而我们现在要实现的是玩家与玩家之间的战斗。

10.4.1　服务器

本次依然使用前面所用到的 Java 服务器。有关本次游戏用到的坦克移动、子弹发射等功能，进行服务器端的开发时都已经设定好了，直接运行即可使用。

10.4.2　客户端

一款多人在线游戏应该让用户之间可以进行简单的交流，所以本次的制作保留了聊天的功能。下面的内容会在 10.3 节代码的基础上做进一步的修改，并且添加一些游戏的功能。

从代码清单 10-5 中我们知道，页面载入完成后会调用 initial() 函数，所以我们需要在 initial() 函数中进行 lufylegend 库件的初始化工作。为 initial 函数添加下面一行代码：

```
init(10, "mylegend", 600, 400, main, LEvent.INIT);
```

在 main 函数运行之前，初始化了一个 backLayer 对象，作为游戏界面的底层。然后新建

了一个 tanklist 数组，用来保存游戏中的坦克。main 函数中做一些简单的游戏准备工作，比如，将 backLayer 对象载入游戏，并画一个黑色边框，作为游戏的背景区域，最后给游戏添加循环事件侦听。相关代码如下所示：

```
var backLayer,tanklist=new Array();
function main() {
    addChild(backLayer);
    backLayer.graphics.drawRect(1, "#000", [0, 0, 600, 400]);
    backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
}
```

其中，onframe 函数是游戏的循环函数，内容很简单，就是循环所有坦克，调用每辆坦克的循环函数。具体如下所示：

```
function onframe(){
for(var key in tanklist){
tanklist[key].onframe();
}
}
```

玩家在接通服务器的时候，页面上必须有一辆自己的坦克，这样才能跟其他玩家进行战斗，所以下面要新建一个坦克类。代码清单 10-9 即为一个坦克对象的完整代码。

代码清单 10-9

```
function Tank(name,direction,color) {
    base(this, LSprite, []);
    var self = this;
    self.targetX = 0;
    self.targetY = 0;
    self.moveX = 0;
    self.moveY = 0;
    self.bulletlist = new Array();
    self.direction = "";
    self.name = name;
    self.color=color;
    switch(direction){
        case "up":
            self.changeUp();
            break;
        case "down":
            self.changeDown();
            break;
        case "left":
            self.changeLeft();
            break;
        case "right":
            self.changeRight();
            break;
    }
```

```
    self.setName();
}
Tank.prototype.setName = function() {
    var self = this;
    var nameText = new LTextField();
    nameText.color = self.color;
    nameText.text = self.name;
    nameText.x = (40 - nameText.getWidth())*0.5;
    nameText.y = self.y - 16;
    self.addChild(nameText);
}
Tank.prototype.changeUp = function() {
    var self = this;
    self.direction = "up";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000", [20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [30, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [18, 0, 4, 20],true,self.color);
}
Tank.prototype.changeDown = function() {
    var self = this;
    self.direction = "down";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [30, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [18, 20, 4, 20],true,self.color);
}
Tank.prototype.changeLeft = function() {
    var self = this;
    self.direction = "left";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 30, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 18, 20, 4],true,self.color);
}
Tank.prototype.changeRight = function() {
    var self = this;
    self.direction = "right";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 30, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [20, 18, 20, 4],true,self.color);
}
Tank.prototype.onframe = function() {
```

```
    var self = this,i,j,bullet,tank;
    self.move();
    self.setDirection();
    for(i=0;i<self.bulletlist.length;i++){
        bullet = self.bulletlist[i];
        bullet.onframe();
        if(bullet.isdie){
            self.bulletlist.splice(i--,1);
            backLayer.removeChild(bullet);
        }
    }
}
Tank.prototype.move = function() {
    var self = this;
    if(self.x == self.targetX && self.y == self.targetY)return;
    if(self.moveX != 0){
        self.x += self.moveX;
        if(self.x == self.targetX){
            self.moveX = 0;
            self.moveY = self.y > self.targetY ? -1 : 1;
        }
    }else if(self.moveY != 0){
        self.y += self.moveY;
        if(self.y == self.targetY){
            self.moveY = 0;
            self.moveX = self.x > self.targetX ? -1 : 1;
        }
    }else{
        if(self.x == self.targetX){
            self.moveY = self.y > self.targetY ? -1 : 1;
        }else if(self.y == self.targetY){
            self.moveX = self.x > self.targetX ? -1 : 1;
        }else if(Math.random() > 0.5){
            self.moveX = self.x > self.targetX ? -1 : 1;
        }else{
            self.moveY = self.y > self.targetY ? -1 : 1;
        }
    }
}
Tank.prototype.setDirection = function() {
    var self = this;
    if(self.x == self.targetX && self.y == self.targetY)return;
    if(self.moveX > 0){
        if(self.direction != "right")self.changeRight();
    }else if(self.moveX < 0){
        if(self.direction != "left")self.changeLeft();
    }else if(self.moveY > 0){
        if(self.direction != "down")self.changeDown();
    }else if(self.moveY < 0){
```

```
            if(self.direction != "up")self.changeUp();
        }
    }
```

代码解析

坦克类的构造器如下：

```
function Tank(name,direction,color)
```

构造器有 3 个参数，第一个参数是坦克的名字，也就是玩家的名字，第二个参数是坦克出现时的方向，第三个参数是坦克的颜色。

构造器中的代码如下：

```
base(this, LSprite, []);
```

这个已经多次提到，是继承 LSprite 类。

```
var self = this;
self.targetX = 0;
self.targetY = 0;
self.moveX = 0;
self.moveY = 0;
```

在上面的代码中，targetX 和 targetY 是移动目标的坐标，如果坦克的当前坐标和目标坐标不一致的话，会向着目标坐标进行移动，这样，在单击屏幕上的某个位置时，就会自动将这个位置的坐标设置成坦克的目标坐标，坦克则朝着这个位置进行移动。moveX 和 moveY 是在坦克移动中，用来控制坦克的移动方向的，这两个参数在坦克的 move 函数中会用到。

```
self.bulletlist = new Array();
```

上面代码表示坦克的子弹数组，每辆坦克发射出的子弹都会保存到自己的子弹数组中，用来控制子弹的移动和碰撞等。

```
   self.name = name;
    self.color=color;
    switch(direction){
        case "up":
            self.changeUp();
            break;
        case "down":
            self.changeDown();
            break;
        case "left":
            self.changeLeft();
            break;
        case "right":
            self.changeRight();
            break;
    }
```

上面的代码用于设定坦克的名称、颜色和方向。在设定方向的时候会根据方向来绘制一个坦克。changeUp、changeDown、changeLeft 和 changeRight 这 4 个方法分别是绘制 4 个方向的坦克。

```
self.setName();
```

以上这行代码表示调用 setName 函数，让坦克显示自己的名称。下面是 setName 函数的完整代码。

```
Tank.prototype.setName = function() {
var self = this;
var nameText = new LTextField();
    nameText.color = self.color;
    nameText.text = self.name;
    nameText.x = (40 - nameText.getWidth())*0.5;
    nameText.y = self.y - 16;
    self.addChild(nameText);
}
```

在 setName 函数中，通过新建一个 LTextField 对象来显示坦克的名称。

```
Tank.prototype.changeUp = function() {
    var self = this;
    self.direction = "up";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [30, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [18, 0, 4, 20],true,self.color);
}
Tank.prototype.changeDown = function() {
    var self = this;
    self.direction = "down";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [30, 0, 10, 40],true,self.color);
    self.graphics.drawRect(1, "#000", [18, 20, 4, 20],true,self.color);
}
Tank.prototype.changeLeft = function() {
    var self = this;
    self.direction = "left";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 30, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 18, 20, 4],true,self.color);
}
```

```
Tank.prototype.changeRight = function() {
    var self = this;
    self.direction = "right";
    self.graphics.clear();
    self.graphics.drawArc(1,"#000",[20,20,13,0,2*Math.PI],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 0, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [0, 30, 40, 10],true,self.color);
    self.graphics.drawRect(1, "#000", [20, 18, 20, 4],true,self.color);
}
```

上面代码中的 4 个函数，先给坦克设定方向，然后再利用坦克的 graphics 属性绘制相应方向上的坦克。

```
Tank.prototype.onframe = function() {
    var self = this,i,j,bullet,tank;
    self.move();
    self.setDirection();
    for(i=0;i<self.bulletlist.length;i++){
        bullet = self.bulletlist[i];
        bullet.onframe();
        if(bullet.isdie){
            self.bulletlist.splice(i--,1);
            backLayer.removeChild(bullet);
        }
    }
}
```

以上代码中的 onframe 函数是坦克的循环函数，它不断被调用。在 onframe 函数中又会通过不断调用 move 和 setDirection 函数，让坦克移动起来。接着循环子弹数组，调用每颗子弹的 onframe 函数，来控制子弹的移动，并且当子弹移出屏幕的时候将子弹移出子弹数组。

既然 onframe 函数中调用了 move 和 setDirection 函数，那么先来看看 move 函数，代码如下：

```
Tank.prototype.move = function() {
    var self = this;
    if(self.x == self.targetX && self.y == self.targetY)return;
    if(self.moveX != 0){
        self.x += self.moveX;
        if(self.x == self.targetX){
            self.moveX = 0;
            self.moveY = self.y > self.targetY ? -1 : 1;
        }
    }else if(self.moveY != 0){
        self.y += self.moveY;
        if(self.y == self.targetY){
            self.moveY = 0;
            self.moveX = self.x > self.targetX ? -1 : 1;
```

```
            }
        }else{
            if(self.x == self.targetX){
                self.moveY = self.y > self.targetY ? -1 : 1;
            }else if(self.y == self.targetY){
                self.moveX = self.x > self.targetX ? -1 : 1;
            }else if(Math.random() > 0.5){
                self.moveX = self.x > self.targetX ? -1 : 1;
            }else{
                self.moveY = self.y > self.targetY ? -1 : 1;
            }
        }
    }
```

在 move 函数中，先判断坦克当前的坐标（x, y）和坦克的目标坐标（targetX, targetY）是否相等，如果不相等则向目标坐标移动。

下面是 setDirection 函数。

```
Tank.prototype.setDirection = function() {
    var self = this;
    if(self.x == self.targetX && self.y == self.targetY)return;
    if(self.moveX > 0){
        if(self.direction != "right")self.changeRight();
    }else if(self.moveX < 0){
        if(self.direction != "left")self.changeLeft();
    }else if(self.moveY > 0){
        if(self.direction != "down")self.changeDown();
    }else if(self.moveY < 0){
        if(self.direction != "up")self.changeUp();
    }
}
```

在上面的代码中，setDirection 函数用于判断坦克当前移动的方向和坦克的面对朝向是否相同，如果不相同则通过调用 changeUp、changeDown、changeLeft 和 changeRight 这 4 个方法中的相应方法来变换坦克的方向。

每一辆坦克都能发射自己的子弹来攻击其他玩家。为了便于控制坦克子弹的移动和碰撞，需要建一个子弹类，如代码清单 10-10 所示。

代码清单 10-10

```
function Bullet(name,direction,color){
    base(this, LSprite, []);
    var self = this;
    self.isdie = false;
    self.name = name;
    switch(direction){
        case "up":
            self.mx = 0;
            self.my = -1;
```

```
            break;
        case "down":
            self.mx = 0;
            self.my = 1;
            break;
        case "left":
            self.mx = -1;
            self.my = 0;
            break;
        case "right":
            self.mx = 1;
            self.my = 0;
            break;
    }
    self.graphics.drawArc(1,"#000",[0,0,3,0,2*Math.PI],true,color);
}
Bullet.prototype.onframe = function() {
    var self = this;
    if(self.isdie)return;
    self.x += self.mx;
    self.y += self.my;
    if(self.x < 0 || self.x > LGlobal.width || self.y < 0 || self.y > LGlobal.height){
        self.isdie = true;
        return;
    }
    var tank;
    for(j=0;j<tanklist.length;j++){
        tank = tanklist[j];
        if(tank.name != self.name && LGlobal.hitTest(self,tank)){
            self.isdie = true;
            socket.send("type=kill&name="+tank.name);
            break;
        }
    }
}
```

代码解析

在子弹的构造器中同样有名称、方向和颜色 3 个参数，其中名称用来判定与子弹发生碰撞的坦克是否为子弹发射方，接着根据传来的方向，给子弹设定 x 轴和 y 轴上的移动速度，最后使用子弹的 graphics 属性绘制一个小球，表示子弹的形状。

子弹的循环函数 onframe 也是被不断调用的，每调用一次，就让子弹在相应的 x 轴或 y 轴上移动一个单位，直到子弹移出游戏界面，这时会将子弹设置为死亡。最后，循环坦克数组，判断子弹自身是否和某辆坦克发生了碰撞，如果发生了碰撞，则向服务器发送信息通知服务器。字符串“type=kill&name= 坦克名称”用于告诉服务器，某坦克被杀死了。

由于这是一款多人在线游戏，当玩家 A 的坦克发生了移动的时候，玩家 B 的界面上也能看到玩家 A 在移动。所以玩家做任何动作都要先通知服务器，然后服务器再通知所有在

线的玩家，从而使得所有玩家在界面上同时发生改变，这样就实现了不同玩家之间游戏的同步。比如，在单击屏幕的某个位置时，并不是立刻让坦克移动，而是将单击的坐标发送给服务器，然后服务器通知所有的玩家某坦克的目标以及发生的移动，各玩家在接收到服务器传来的信息后，通过解析这段信息来实施相应的指令。

先来看看玩家通过服务器传送信息的部分。在游戏中利用鼠标来控制坦克的行走，利用键盘的空格键来让坦克发射子弹，所以需要为游戏添加两个事件侦听，一个是鼠标事件，另一个是键盘事件。这时需要修改 main 函数，如下所示：

```
function main() {
    addChild(backLayer);
    backLayer.graphics.drawRect(1, "#000", [0, 0, 600, 400]);
    backLayer.addEventListener(LEvent.ENTER_FRAME,onframe);
    backLayer.addEventListener(LMouseEvent.MOUSE_UP,onup);
    LEvent.addEventListener(LGlobal.window,
          LKeyboardEvent.KEY_UP,onkeyup);
}
```

上面的代码分别给游戏添加了一个侦听鼠标弹起事件的函数 onup 和侦听键盘按键弹起的事件 onkeyup。先来看看 onup 函数。

```
function onup(event){
    socket.send("type=move&x=" +  event.selfX + "&y=" + event.selfY);
}
```

前面说过，玩家做任何一件事情，都是先通知服务器，所以当鼠标弹起的时候，向服务器发送信息“type=move&x= 移动目标的 x 坐标 &y= 移动目标的 y 坐标”。这条消息告诉服务器，玩家要进行移动了，并且发送了移动目的地的坐标。

下面是 onkeyup 函数的内容：

```
function onkeyup(event){
    if(event.keyCode != 32)return;
    socket.send("type=shoot&direction=" + getSelf().direction);
}
```

其中，keyCode 用来判定用户按下的是哪个键。onkeyup 函数处理的内容是，当键盘的空格键弹起的时候，向服务器发送信息“type=shoot&direction= 射击方向”。这条消息告诉服务器，玩家发射了子弹，并且告诉了子弹的发射方向。函数中 getSelf() 表示得到玩家控制的坦克对象，代码如下所示：

```
function getSelf(){
    if(mytank)return mytank;
    for(var key in tanklist){
        tank = tanklist[key];
        if(tank.name == document.getElementById("name").value){
            mytank=tank;
        }
```

```
    }
    return mytank;
}
```

上面的内容很简单，通过循环所有的坦克，来找到玩家自己控制的坦克，然后返回给函数调用端。

接着来看一下客户端如何接收服务器信息，并且如何通过解析这些信息来获得相应的游戏指令，如代码清单 10-11 所示。

代码清单 10-11

```
socket.addEventListener('message', function(e) {
    var value = getScript(e.data);
    var text;
    switch(value["result"]) {
        case "error":
            text = "<li>" + value["error"] + "</li>" + msg.innerHTML;
            break;
        case "loginok":
            document.getElementById("name").value = value["name"];
            text = "<li>" + " 登录成功 " + "</li>" + msg.innerHTML;
            var dirlist = ["up","down","left","right"];
            socket.send("type=addTank&x=" + Math.floor(LGlobal.width*Math.random()) +
                "&y=" + Math.floor(LGlobal.height*Math.random()) +
                "&direction=" + dirlist[Math.floor(4*Math.random())] +
                "&color="+randomColor());
            break;
        case "talk":
            text = "<li>" + value["msg"] + "</li>" + msg.innerHTML;
            break;
        case "removeuser":
            removeUser(value["name"]);
            break;
        case "setuserlist":
            removeAllUser();
            var list = value["list"].split(",");
            for(var i = 0, l = list.length; i < l; i++) {
                addUser(list[i]);
            }
            break;
        case "addTank":
            addTank(value["name"],
                parseInt(value["x"]),parseInt(value["y"]),
                value["direction"],value["color"]);
            break;
        case "move":
            move(value["name"],
                parseInt(value["x"]),parseInt(value["y"]));
            break;
        case "shoot":
```

```
            shoot(value["name"],value["direction"]);
            break;
        case "kill":
            kill(value["name"]);
            break;
    }
    if(text) msg.innerHTML = text;
});
```

代码解析

getScript 函数表示分解服务器字符串信息，这个在前面的聊天室中已经讲过。

下面依次讲解 switch 的每个消息分支。

```
case "error":
    text = "<li>" + value["error"] + "</li>" + msg.innerHTML;
    break;
```

上面代码表示当游戏信息解析发生错误的时候，显示错误提示信息。

```
case "loginok":
    document.getElementById("name").value = value["name"];
    text = "<li>" + " 登录成功 " + "</li>" + msg.innerHTML;
    var dirlist = ["up","down","left","right"];
    socket.send("type=addTank&x=" +
        Math.floor(LGlobal.width*Math.random()) +
        "&y=" + Math.floor(LGlobal.height*Math.random()) +
        "&direction=" + dirlist[Math.floor(4*Math.random())] +
        "&color="+randomColor());
    break;
```

在上面的代码中，当从服务器接收到字符串“result=loginok&name= 用户名”时，代表用户登录成功，这时候直接向服务器发送信息。“type=addTank&x= 坐标 x&y= 坐标 y&direction= 方向 &color= 颜色”表示为游戏添加一辆坦克，并且发送了坦克出现的坐标、方向和颜色。

```
case "talk":
case "removeuser":
case "setuserlist":
```

上面这 3 个分支直接应用了 10.3 节中聊天室的功能，不再重复。

```
case "addTank":
    addTank(value["name"],
        parseInt(value["x"]),parseInt(value["y"]),
        value["direction"],value["color"]);
    break;
```

在上面的代码中，当从服务器接收到字符串“result=addTank&name= 用户名 &x=x 坐标 &y=y 坐标 &direction= 方向 &color= 颜色”时，代表为游戏添加一辆坦克，调用 addTank 函

数来为游戏添加坦克。addTank 函数代码如下所示：

```
function addTank(name,x,y,direction,color){
    var tank = new Tank(name,direction,color);
    tank.x = x;
    tank.y = y;
    tank.targetX = x;
    tank.targetY = y;
    backLayer.addChild(tank);
    tanklist.push(tank);
}
```

在这里，addTank 函数的功能是新建一个坦克对象，并将其添加进 tanklist 数组。

```
case "move":
    move(value["name"],
        parseInt(value["x"]),parseInt(value["y"]));
    break;
```

在上面的代码中，当从服务器接收到字符串"result=move&name=用户名&x=x坐标&y=y坐标"时，表示某坦克发生移动，调用 move 函数来设定坦克的移动目的地。move 函数的代码如下所示：

```
function move(name,targetX,targetY){
    var tank;
    for(var key in tanklist){
        tank = tanklist[key];
        if(tank.name == name)break;
    }
    if(tank == null)return;
    tank.targetX = targetX;
    tank.targetY = targetY;
    tank.moveX=tank.moveY=0;
}
```

在这里，move 函数的功能首先是循环坦克数组，通过坦克名称来获得坦克对象，然后为坦克设定移动目的地的坐标。

```
case "shoot":
    shoot(value["name"],value["direction"]);
    break;
```

在上面的代码中，当从服务器接收到字符串"result=shoot&name=用户名&direction=方向"时，表示游戏中某坦克发射了一颗子弹，调用 shoot 函数来发射子弹。shoot 函数的代码如下所示：

```
function shoot(name,direction){
    var bullet,tank;
    for(var key in tanklist){
```

```
            tank = tanklist[key];
            if(tank.name == name)break;
        }
        if(tank == null)return;
        bullet = new Bullet(name,direction,tank.color);
        bullet.x = tank.x + 20;
        bullet.y = tank.y + 20;
        tank.bulletlist.push(bullet);
        backLayer.addChild(bullet);
    }
```

shoot 函数的功能首先是循环坦克数组，通过坦克名称来获得坦克对象，然后新建一个子弹对象，并且将子弹对象加入该坦克对象的子弹数组中。

```
case "kill":
    kill(value["name"]);
    break;
```

在上面的代码中，当从服务器接收到字符串“result=kill&name= 用户名”时，表示游戏中的某坦克死亡，调用 kill 函数移除坦克。kill 函数的代码如下所示：

```
function kill(name){
    if(name == getSelf().name){
        doClose();
    }else{
        var tank;
        for(var key in tanklist){
            tank = tanklist[key];
            if(tank.name == name){
                backLayer.removeChild(tank);
                break;
            }
        }

    }
}
```

kill 函数的功能首先是判断被杀死的坦克是不是自己，如果是自己，则游戏结束，断开与服务器的连接；如果不是自己，则循环坦克数组，通过坦克名称来获得坦克对象，然后将该坦克移出游戏。

以上就是整个游戏的全部功能，运行相关代码得到的效果如图 10-17 所示。

你可以打开本书源码中的文件夹 10-3，查看完整的游戏代码，启动服务器后，直接用浏览器打开 Tank.html 就可以了，也可以使用多个浏览器窗口扮演多个角色运行该游戏。

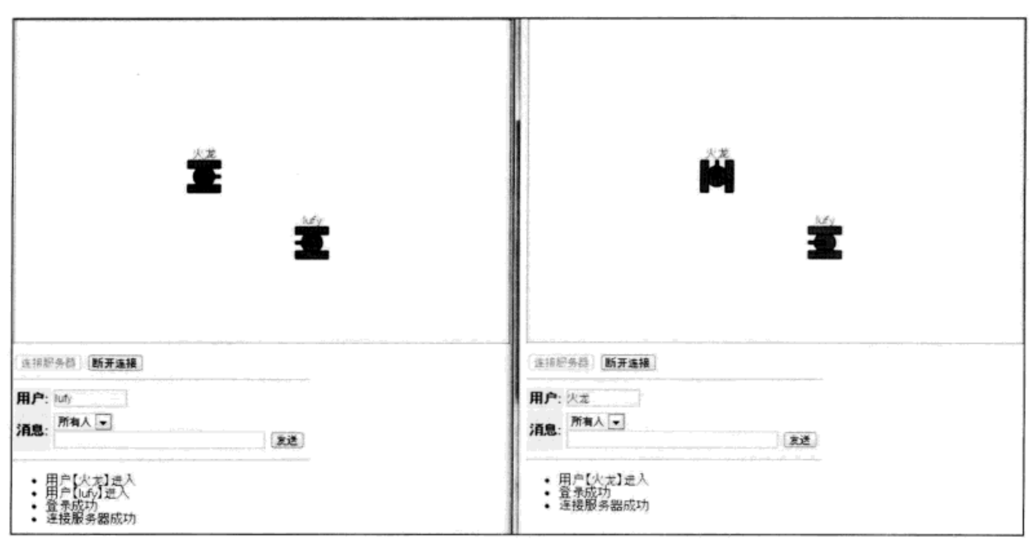


图 10-17　多人坦克大战游戏效果

10.5　小结

在本章介绍的聊天室和坦克大战案例中，客户端与服务器之间的通信所用的字符串信息以及信息发送的格式等都是已经在服务器上设定好了的。如果自己开发服务器，可以选择自己喜欢的通信方式，如 XML、JSON 等，自己设定这些信息。

另外，本章中的游戏案例只是进行了功能性的演示，帮助大家认识一下网络游戏的实现过程。真正开发网络游戏的时候，还需要考虑网络延时等多种因素，需要查阅一些相关的书籍，才能够开发出好的游戏。

PDG

第四部分

技能提高篇

❑ 第 11 章　提高效率的分析

第 11 章　提高效率的分析

本章将分别在 Chrome、Firefox、Safari、Opera 这 4 个浏览器上进行测试，用数据来详细分析一下如何提高 HTML5 的执行效率。

11.1　绘图时使用小数的影响

在绘制图片的时候，使用整数和小数作为坐标，其效率是有区别的。下面就来分别测试一下整数和小数在各个浏览器中的执行效率。先看代码清单 11-1。

代码清单　11-1

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img,20,20);
}
```

以上代码是使用整数坐标绘图 1000 次。再来看看代码清单 11-2。

代码清单　11-2

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img,310.7,10.6);
}
```

以上代码是使用小数坐标绘图 1000 次。

两次测试在各个浏览器中的执行结果如表 11-1 所示。

表 11-1　使用小数和整数作为坐标绘图时的测试结果

浏览器	整数执行时间 a（毫秒）	小数执行时间 b（毫秒）	b/a
Chrome	105	108	1
Firefox	115	2318	20.1
Safari	162	1613	10
Opera	63	1286	20.4

可以看到，除了 Chrome 以外，在其他几个浏览器中使用小数绘图都是非常慢的。所以在绘图时要避免使用小数，如果遇到小数坐标，要先将其转换成整数再进行绘图。关于如何高速地将小数转换成整数，在后面会进一步进行分析。

11.2　drawImage 和 putImageData 的效率比较

在 HTML5 中进行绘图时，有 drawImage 和 putImageData 两种方法。当然，它们应用的领域不同，现在只在效率上分析一下两个函数中哪个更快速。先来看代码清单 11-3。

代码清单　11-3

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img,20,20);
}
```

以上代码是使用 drawImage 函数绘图 1000 次。再来看看代码清单 11-4。

代码清单　11-4

```
var imgData=ctx.getImageData(20,20,240,240);
for (i=0; i < 1000; i++) {
        ctx.putImageData(imgData,300,20);
}
```

以上代码是使用 putImageData 函数绘图 1000 次。

两个函数绘制的是相同大小的图片，它们在各个浏览器中的执行结果如表 11-2 所示。

表 11-2　drawImage 和 putImageData 的效率测试结果

浏览器	drawImage 函数 执行时间 a（毫秒）	putImageData 函数 执行时间 b（毫秒）	b/a
Chrome	104	484	4.6
Firefox	95	482	5
Safari	141	110	0.8
Opera	62	219	3.5

可以看到，只有在 Safari 上 putImageData 函数的效率略高于 drawImage 函数，而在其他几个浏览器中，drawImage 函数的执行速度明显要比 putImageData 函数快很多。所以在开发过程中，如果这两个函数同时可用，应该选用效率较高的 drawImage 函数。

11.3　区域更新和图片大小对绘图效率的影响

在开发 HTML5 应用时，有时候画面很大，但是需要更新的只是画面中的一部分。如果只对画面的一部分进行更新的话，对提高整个程序的执行效率是否有帮助呢？

先来看看 drawImage 函数绘制区域大小对速度的影响，如代码清单 11-5 所示。

代码清单　11-5

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img,20,20);
}
```

以上是使用 drawImage 函数绘制一张 240×240 的大图片 1000 次。再来看看代码清单 11-6。

代码清单 11-6

```
for (i=0; i < 1000; i++) {
            ctx.drawImage(img,0,0,50,50,20,280,50,50);
};
```

以上是使用 drawImage 函数绘制大图片中 50×50 大小的那部分 1000 次。再看一个代码清单 11-7。

代码清单 11-7

```
for (i=0; i < 1000; i++) {
            ctx.drawImage(img_mini,300,280);
};
```

以上是使用 drawImage 函数直接绘制一张 50×50 的小图片 1000 次。

使用 drawImage 函数绘制大图、绘制大图中的一部分、绘制小图在各个浏览器中的执行结果如表 11-3 所示。

表 11-3 使用 drawImage 函数绘制不同大小图片的测试结果

浏览器	绘制大图 执行时间 a（毫秒）	绘制大图的一部分 执行时间 b（毫秒）	绘制小图 执行时间 b（毫秒）
Chrome	104	9	8
Firefox	95	15	15
Safari	141	26	24
Opera	62	11	6

可以看到，使用 drawImage 函数绘图的时候，绘制的区域越小，速度越快，而图片的大小对效率影响不大，也就是说我们可以准备一张大图，但是在实际绘图的时候只绘制其中的一部分。

再来看看使用 putImageData 函数绘制图片时区域大小对速度的影响。请看代码清单 11-8。

代码清单 11-8

```
var imgData=ctx.getImageData(20,20,240,240);
for (i=0; i < 1000; i++) {
            ctx.putImageData(imgData,300,20);
}
```

以上是使用 putImageData 函数绘制大小为 240×240 的图片 1000 次。再来看代码清单 11-9。

代码清单　11-9

```
var imgData=ctx.getImageData(20,20,50,50);
for (i=0; i < 1000; i++) {
        ctx.putImageData(imgData,200,280);
}
```

以上是使用 putImageData 函数绘制大小为 50×50 的图片 1000 次。

使用 putImageData 函数绘制大图和绘制小图在各个浏览器中的执行结果如表 11-4 所示。

表 11-4　使用 putImageData 函数进行不同大小图片的测试结果

浏览器	putImageData 绘制大小为 240×240 图片执行时间 a(毫秒)	putImageData 绘制大小为 50×50 图片执行时间 b(毫秒)
Chrome	484	22
Firefox	482	37
Safari	110	5
Opera	219	10

可以看到，putImageData 函数和 drawImage 函数一样，在使用它绘图时，绘制的区域越小，绘图的速度越高，图片大小对执行时间有直接的影响。

11.4　图片格式对绘图效率的影响

在进行游戏开发的时候，用到的图片会是多种多样的，图片的格式主要有 jpg、png、gif 和 bmp 这 4 种。下面来分析一下在 HTML5 中绘制这 4 种格式的图片时效率是否会有区别。先来看代码清单 11-10。

代码清单　11-10

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img_jpg,20,20);
}
```

以上是使用 jpg 格式的图片绘图 1000 次。再来看看代码清单 11-11。

代码清单　11-11

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img_png,280,20);
}
```

以上是使用 png 格式的图片绘图 1000 次。然后看代码清单 11-12。

代码清单　11-12

```
for (i=0; i < 1000; i++) {
        ctx.drawImage(img_gif,20,280);
}
```

以上是使用 gif 格式的图片绘图 1000 次。最后看代码清单 11-13。

代码清单 11-13

```
for (i=0; i < 1000; i++) {
            ctx.drawImage(img_bmp,280,280);
}
```

以上是使用 bmp 格式的图片绘图 1000 次。

表 11-5 是使用这 4 种格式的图片时的绘图速度。

表 11-5 使用不同图片格式进行绘图的速度测试结果

浏览器	jpg 格式 执行时间（毫秒）	png 格式 执行时间（毫秒）	gif 格式 执行时间（毫秒）	bmp 格式 执行时间（毫秒）
Chrome	801	774	782	768
Firefox	418	461	452	459
Safari	1318	854	1232	891
Opera	405	506	792	445

可以看到，各种浏览器在数据上是有区别的，就平均值和实用性来说，png 格式相对好一些。

11.5 优化代码以提高整体效率

前面对 HTML5 中的一些函数进行了效率分析，下面对 JavaScript 的代码进行分析，看看应如何优化代码来提高程序的效率。

11.5.1 使用位运算

在游戏开发中，各种运算都是不可缺少的。我们在做各种运算的时候，可选择的方法很多，那么选取哪种方法来进行运算更有效率呢？下面进行一些简单的测试。

1. 除法运算

除法运算是很常用的运算，比如将图片分为几等份。对于除法运算，通常的做法是使用“/”直接进行运算。来看看代码清单 11-14。

代码清单 11-14

```
for (i=0; i < 100000000; i++) {
            r = 321 / 16;
}
```

代码清单 11-14 是使用“/”进行除法运算 100000000 次。再看代码清单 11-15。

代码清单　11-15

```
for (i=0; i < 100000000; i++) {
        r = 321 >> 4;
}
```

代码清单 11-15 是使用位运算来代替除法进行运算 100000000 次。

两种除法运算的执行结果如表 11-6 所示。

表 11-6　使用除法运算与位运算做除法的测试结果

浏览器	除法运算 执行时间（毫秒）	位运算 执行时间（毫秒）
Chrome	560	480
Firefox	419	376
Safari	1063	1374
Opera	312	265

可以看到，除了 Safari 之外，其他几个浏览器中位运算的效率普遍高于直接进行除法运算。

2. 乘法运算

在程序中进行坐标计算等运算时一定会用到乘法运算，下面是对乘法运算的测试。先看代码清单 11-16。

代码清单　11-16

```
for (i=0; i < 100000000; i++) {
        r = 321 * 16;
}
```

代码清单 11-16 是使用 * 进行乘法运算 100000000 次。再看代码清单 11-17。

代码清单　11-17

```
for (i=0; i < 100000000; i++) {
        r = 321 << 4;
}
```

代码清单 11-17 是使用位运算来代替乘法进行运算 100000000 次。

两种乘法运算的执行结果如表 11-7 所示。

表 11-7　使用乘法运算与位运算做乘法的测试结果

浏览器	乘法运算 执行时间（毫秒）	位运算 执行时间（毫秒）
Chrome	540	471

PDG

（续）

浏览器	乘法运算 执行时间（毫秒）	位运算 执行时间（毫秒）
Firefox	425	371
Safari	1080	969
Opera	342	260

可以看到，4 个浏览器中位运算的效率都略高于直接进行乘法运算。

3. 求余运算

下面分析一下求余数运算。还是先来看代码清单 11-18。

代码清单 11-18

```
for (i=0; i < 100000000; i++) {
        r = 321 % 4;
}
```

代码清单 11-18 是使用 % 进行求余运算 100000000 次。再看代码清单 11-19。

代码清单 11-19

```
for (i=0; i < 100000000; i++) {
        r = 321 & 3;
}
```

代码清单 11-19 是使用位运算来代替求余进行运算 100000000 次。

两种求余运算的结果如表 11-8 所示。

表 11-8 使用求余运算与位运算进行求余运算的测试结果

浏览器	求余运算 执行时间（毫秒）	位运算 执行时间（毫秒）
Chrome	540	47
Firefox	396	376
Safari	1043	1060
Opera	314	263

可以看到，结果依然是除了 Safari 之外，其他几个浏览器中位运算的效率都高于直接进行求余运算。

11.5.2 少用 Math 静态类

为了方便进行各种运算，JavaScript 提供了 Math 静态类，里面包括了取整、求最大最小值、三角函数等各种函数，但是这些方式的效率并不是最快的。下面举几个例子来说明一下。

1. 取整

利用 Math 静态类做取整运算很简单，看代码清单 11-20。

代码清单　11-20

```
for (i=0; i < 100000000; i++) {
        r = Math.floor(1.3);
}
```

以上是使用 Math.floor 函数进行取整 100000000 次。再来看看代码清单 11-21。

代码清单　11-21

```
for (i=0; i < 100000000; i++) {
        r = 1.3 >> 0;
}
```

以上是使用位运算来取整 100000000 次。

表 11-9 是使用 Math.floor 函数和位运算进行取整的对比结果。

表 11-9　使用 Math.floor 函数和位运算进行取整的结果对比

浏览器	Math.floor 函数 执行时间（毫秒）	位运算 执行时间（毫秒）
Chrome	556	464
Firefox	476	379
Safari	7074	967
Opera	2426	262

可以看到，使用位运算来做取整运算能够大幅度地提高程序的效率。

2. 求最大值

利用 Math 静态类来求最大值时，需要使用它的 Math.max 函数，看代码清单 11-22。

代码清单　11-22

```
for (i=0; i < 100000000; i++) {
        r = Math.max(6,4);
}
```

代码清单 11-22 是使用 Math. max 函数求最大值 100000000 次。再来看看代码清单 11-23。

代码清单　11-23

```
for (i=0; i < 100000000; i++) {
        if(6>4){
                r = 6;
        }else{
                r = 4;
```

```
        }
}
```

代码清单 11-23 是直接使用 if 判断来求最大值 100000000 次。

表 11-10 是使用 Math. max 函数和 if 判断来求最大值的对比结果。

表 11-10 Math. max 函数和 if 判断来求最大值的结果对比

浏览器	Math.max 函数 执行时间（毫秒）	if 语句 执行时间（毫秒）
Chrome	848	453
Firefox	401	394
Safari	8059	982
Opera	8009	1720

可以看到，直接使用 if 判断来求最大值要比 Math. max 函数快很多。

3. 求幂

利用 Math 静态类来求幂时会使用它的 Math. pow 函数，看代码清单 11-24。

代码清单 11-24

```
for (i=0; i < 100000000; i++) {
        r = Math.pow(2,5);
}
```

代码清单 11-24 是使用 Math. pow 函数来求幂 100000000 次。再来看看代码清单 11-25。

代码清单 11-25

```
for (i=0; i < 100000000; i++) {
        r = 2 * 2 * 2 * 2 * 2;
}
```

代码清单 11-25 是采用直接运算的方式来求幂 100000000 次。

表 11-11 是使用 Math. pow 函数和直接求幂的对比结果。

表 11-11 Math. pow 函数和直接求幂的结果对比

浏览器	Math. pow 函数 执行时间（毫秒）	直接求幂 执行时间（毫秒）
Chrome	531	466
Firefox	5001	380
Safari	3069	984
Opera	22845	529

可以看到，直接求幂的效率要高于 Math.pow 函数。

11.5.3 优化算法

除了使用位运算等替换原始函数来提高效率外，最能影响执行速度的就是程序中的算法。比如在开发游戏的时候，经常要判断两个物体是否相遇，这时候可能要做两两的碰撞检测，下面通过测试来看看优化算法的重要性。

代码清单 11-26 中给出的是一种最简单的做法，直接循环游戏中的对象，从而进行两两的碰撞检测。

代码清单 11-26

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>碰撞检测 一般方法</title>
</head>
<body>
<div id="mylegend">loading......</div>
<script type="text/javascript" src="http://lufylegend.com/js/lufylegend-
1.5.0.js"></script>
<script type="text/javascript">
init(30,"mylegend",800,450,main,LEvent.INIT);
var back,query,list;
function main(){
    back = new LSprite();
    back.graphics.drawRect(1,"#000000",[0,0,800,450],true,"#cccccc");
    addChild(back);
    list = [];
    var child;
    for(var i=0;i< 200;i++){
        child = new LSprite();
        child.graphics.drawRect(1,"#000000",[0,0,20,20]);
        child.x = Math.random()*750;
        child.y = Math.random()*400;
        child.sx = 10 - Math.random()*20;
        child.sy = 10 - Math.random()*20;
        back.addChild(child);
        list.push(child);
    }
    back.addEventListener(LEvent.ENTER_FRAME, onframe);
}
function onframe(){
    var child,child2;

    var arr = [];
    for(var key=0; key < back.childList.length;key++){
        child = back.childList[key];
        child.x += child.sx;
        child.y += child.sy;
```

```
            if(child.x < 0 || child.x > LGlobal.width)child.sx *= -1;
            if(child.y < 0 || child.y > LGlobal.height)child.sy *= -1;

            child.graphics.clear();
            child.graphics.drawRect(1,"#000000",[0,0,20,20]);
            for (var i = key + 1; i < back.childList.length; i++) {
                child2 = back.childList[i];
                if(LGlobal.hitTest(child,child2)){
                    arr[key] = 1;
                    arr[i] = 1;
                }
            }
        }
        for (key in arr){
            child = back.childList[key];
            child.graphics.clear();
            child.graphics.drawRect(1,"#ff0000",[0,0,20,20]);
        }
    }
    </script>
    </body>
    </html>
```

在代码清单 11-26 的测试中，因为是两两碰撞，所以随着物体数量的增加，需要检测的碰撞次数是成积数增长的，这样的方法对于物体数量较少的检测来说是可以的，但是当屏幕上需要检测的物体数量比较多的时候，这种方式就无法胜任了。大家可以打开本书源码文件夹第 11 章 /11.5 里的 01.html 文件进行测试，该程序检测了 200 个物体进行两两碰撞的情况，对于一般配置的电脑来说，一定会出现卡顿的现象。

基于此，对于大量物体的两两碰撞检测，通常会使用“四叉树”算法来提高检测的效率。四叉树是一种树状结构，图 11-1 是四叉树的原理图。

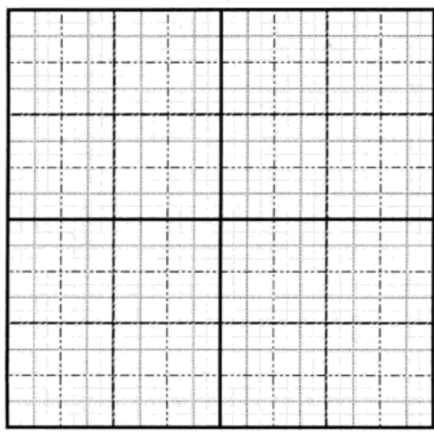


图 11-1 四叉树原理图

从图 11-1 中可以看到，首先将一个区域分成了四等份，然后每个区域再继续分成 4 个区域，这样检测的次数就会以 4 为倍数进行缩减。

代码清单 11-27 是使用四叉树进行两两碰撞的检测。

代码清单　11-27

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>碰撞检测 四叉树方法</title>
</head>
<body>
<div id="mylegend">loading......</div>
<script type="text/javascript" src="http://lufylegend.com/js/lufylegend-1.5.0.js">
</script>
<script type="text/javascript">
init(30,"mylegend",800,450,main,LEvent.INIT);
var back,query,list;
function main(){
    back = new LSprite();
    back.graphics.drawRect(1,"#000000",[0,0,800,450],true,"#cccccc");
    query = new LQuadTree(new LRectangle(0,0,800,450));
    query.createChildren(3);
    addChild(back);
    list = [];
    var child;
    for(var i=0;i< 400;i++){
        child = new LSprite();
        child.graphics.drawRect(1,"#000000",[0,0,20,20]);
        child.x = Math.random()*750;
        child.y = Math.random()*400;
        child.sx = 10 - Math.random()*20;
        child.sy = 10 - Math.random()*20;
        back.addChild(child);
        list.push(child);
        query.add(child,child.x,child.y);
    }
    back.addEventListener(LEvent.ENTER_FRAME, onframe);
}

function onframe(){
    var child,child2;
    for (var i = 0; i < list.length; i++) {
        child = list[i];
        child.x += child.sx;
        child.y += child.sy;
        if(child.x < 0 || child.x > LGlobal.width)child.sx *= -1;
        if(child.y < 0 || child.y > LGlobal.height)child.sy *= -1;
        query.remove(child);
        query.add(child,child.x,child.y);
    }
```

```
    var arr = [];
    for(var key=0; key < back.childList.length;key++){
        child = back.childList[key];

        child.graphics.clear();
        child.graphics.drawRect(1,"#000000",[0,0,20,20]);
        var queryArr = query.getDataInRect(new LRectangle(child.x - 20,child.y -
20,60,60));
        for (var i = 0; i < queryArr.length; i++) {
            child2 = queryArr[i];
            if(child.objectindex != child2.objectindex && LGlobal.hitTest(child,child2)){
                arr.push(key);
                break;
            }
        }
    }
    for (key in arr){
        child = back.childList[arr[key]];
        child.graphics.clear();
        child.graphics.drawRect(1,"#ff0000",[0,0,20,20]);
    }
}
</script>
</body>
</html>
```

在代码清单 11-27 的测试中，对 500 个物体进行了碰撞检测，其检测的效率要比代码清单 11-26 高很多，完全没有卡顿的现象。大家可以打开本书源码文件夹第 11 章 /11.5 里的 02.html 文件进行测试，看看四叉树算法的效率如何。

11.6 小结

本章进行测试的各个浏览器的版本如下：

Chrome v23.0.1243.2

Firefox 16.0.2

Safari 5.1.7

Opera 12.02

由于各个浏览器都在不断地提高自身的性能，所以不同版本浏览器的测试结果也可能会有很大差别。

影响程序效率的因素有很多，本章无法全部列举，只是总结了一小部分。在实际开发过程中，即使是相同的结果也要尝试使用多种方法来实现，才能从中筛选出最优的方法。只有不断地进行测试总结，才能不断地提升程序的性能，写出高效的代码，也才能开发出运行流畅的游戏。

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
ｉｎｆｏ：本信息由Ｏｎｌｙｏｗｎ终结版生成
图书名：ＨＴＭＬ　５　ＣＡＮＶＡＳ游戏开发实战
ＳＳ：１３２３６１４０
页码：３２２
本书总页数：３３６
ＢｏｏｋＵｒｌ：ｈｔｔｐ：／／ｂｏｏｋ．ｃｃｅｌｉｂ．ｃｏｍ／ｂｏｏｋＤｅｔａｉｌ．ｊｓｐ？ｄｘＮｕｍｂｅｒ＝０００
０１１７２９７２２＆ｄ＝４４５１５ＦＣＢＦ７７１４２４１２５５ＢＦ７０４Ｅ３５ＥＣ３００＆ｆｅｎｌｅｉ＝１８１７０４
０３０２＆ｓｗ＝ＨＴＭＬ５＋Ｃａｎｖａｓ％Ｄ３％ＣＥ％ＣＦ％Ｂ７％ＢＦ％ＡＡ％Ｂ７％Ａ２％ＣＡ％Ｂ５％Ｄ５％ＢＤ
Ｓｔｒ：／ｉｍｇ１５／３２７６Ｂ８５Ｅ８７７１９５１６７５６ＤＢＡ４８５３Ｄ０ＢＦＡ１Ｃ７Ｆ０６１１Ｂ８７ＢＦＤＣ６
２ＣＤ５０ＥＢ８２３６Ｄ８Ｆ２０２５４３７０Ｂ５１８ＥＤ０３Ｆ６３２７Ａ４９８１０ＤＤ３９６３５ＡＥ９２５１７９ＤＥ
ＣＡＣＤ２８Ｅ５２Ｄ８ＣＦ８７Ｆ７７８ＦＤＢ２０ＤＤ０Ａ９Ｄ３Ｂ６１９Ｆ１６６Ｃ３１Ｆ２Ｆ８Ａ１３Ｅ７９ＣＦＤＣ１Ｅ
Ｄ１９７９Ｂ９１０４８２Ｆ３７１６４９ＣＣ３６Ｂ６Ａ４２７Ｅ３１５０９Ｆ５３３Ｂ４８９３Ａ５５Ｃ４５Ｂ７Ｃ２Ｄ７０８
１Ｅ８Ｆ９Ａ５／ｂ４８／