第 2 章

Canvas API

2

在本章中，我们将探索如何使用 HTML5 的 Canvas API。Canvas API 很酷，可以通过它来动态生成和展示图形、图表、图像以及动画。本章将使用渲染 API（rendering API）的基本功能来创建一幅可以放大缩小并自适应浏览器环境的图。还会演示如何基于用户输入来动态创建图像，生成热点图。当然，我们也会提醒你在使用 HTML5 Canvas 时需要注意的问题，并且分享解决这些问题的方法。

本章只涉及了最基本的图形知识，因此，你大可不必担心学不会而跳过本章。来吧，让我们一起来感受 HTML5 中这个强大的特性吧。

2.1 HTML5 Canvas 概述

关于 HTML5 Canvas API 完全可以写一本书（还不会是一本很薄的书）。由于只有一章的篇幅，所以我们将讨论 API 中那些我们认为是最常用的功能。

2.1.1 历史

Canvas 的概念最初是由苹果公司提出的，用于在 Mac OS X WebKit 中创建控制板部件（dashboard widget）。在 Canvas 出现之前，开发人员若要在浏览器中使用绘图 API，只能使用 Adobe 的 Flash 和 SVG（Scalable Vector Graphics，可伸缩矢量图形）插件，或者只有 IE 才支持的 VML（Vector Markup Language，矢量标记语言），以及其他一些稀奇古怪的 JavaScript 技巧。

假设我们要在没有 `canvas` 元素的条件下绘制一条对角线——听起来似乎很简单，但实际上如果没有一套二维绘图 API 的话，这会是一项相当复杂的工作。HTML5 Canvas 能够提供这样的功能，对浏览器端来说此功能非常有用，因此 Canvas 被纳入了 HTML5 规范。

起初，苹果公司曾暗示可能会为 WHATWG（Web Hypertext Application Technology Working Group，Web 超文本应用技术工作组）草案中的 Canvas 规范申请知识产权，这在当时引起了一些

Web 标准化追随者的关注。不过，苹果公司最终还是按照 W3C 的免版税专利权许可条款公开了其专利。

SVG 和 CANVAS 对比

“ Canvas 本质上是一个位图画布，其上绘制的图形是不可缩放的，不能像 SVG 图像那样可以被放大缩小。此外，用 Canvas 绘制出来的对象不属于页面 DOM 结构或者任何命名空间——这点被认为是一个缺陷。SVG 图像却可以在不同的分辨率下流畅地缩放，并且支持点击检测（能检测到鼠标点击了图像上的哪个点）。

既然如此，为什么 WHATWG 的 HTML5 规范不使用 SVG 呢？尽管 Canvas 有明显的不足，但 HTML Canvas API 有两方面优势可以弥补：首先，不需要将所绘制图像中的每个图元当做对象存储，因此执行性能非常好；其次，在其他编程语言现有的优秀二维绘图 API 的基础上实现 Canvas API 相对来说比较简单。毕竟，二鸟在林不如一鸟在手。”

——Peter

2.1.2 canvas 是什么

在网页上使用 `canvas` 元素时，它会创建一块矩形区域。默认情况下该矩形区域宽为 300 像素，高为 150 像素，但也可以自定义具体的大小或者设置 `canvas` 元素的其他特性。代码清单 2-1 是可放到 HTML 页面中的最基本的 `canvas` 元素。

代码清单 2-1 基本的 `canvas` 元素

```
<canvas></canvas>
```

在页面中加入了 canvas 元素后，我们便可以通过 JavaScript 来自由地控制它。可以在其中添加图片、线条以及文字，也可以在里面绘图，甚至还可以加入高级动画。

大多数主流操作系统和框架支持的二维绘制操作，HTML5 Canvas API 都支持。如果你在近年来曾经有过二维图像编程的经验，那么会对 HTML5 Canvas API 感觉非常顺手，因为这个 API 就是参照既有系统设计的。如果没有这方面经验，则会发现与这么多年来一直使用的图片加 CSS 开发 Web 图形的方式比起来，Canvas 的渲染系统有多么强大。

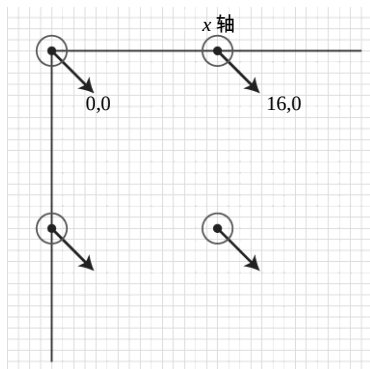

图 2-1　canvas 中的 *x*、*y* 坐标

使用 canvas 编程，首先要获取其上下文（context）。接着在上下文中执行动作，最后将这些动作应用到上下文中。可以将 canvas 的这种编辑方式想象成为数据库事务：开发人员先发起一个事务，然后执行某些操作，最后提交事务。

2.1.3　**canvas** 坐标

如图 2-1 所示，canvas 中的坐标是从左上角开始的，*x* 轴沿着水平方向（按像素）向右延伸，*y* 轴沿垂直方向向下延伸。左上角坐标为 x=0，y=0 的点称作原点。

2.1.4　什么情况下不用 **canvas**

尽管 canvas 元素功能非常强大，用处也很多，但在某些情况下，如果其他元素已经够用了，就不应该再使用 canvas 元素。例如，用 canvas 元素在 HTML 页面中动态绘制所有不同的标题，就不如直接使用标题样式标签（H1、H2 等），它们所实现的效果是一样的。

2.1.5 替代内容

访问页面的时候，如果浏览器不支持 canvas 元素，或者不支持 HTML5 Canvas API 中的某些特性，那么开发人员最好提供一份替代代码（2.1.7 节中的表 2-1 详细介绍了浏览器对 canvas 的支持情况）。例如，开发人员可以通过一张替代图片或者一些说明性的文字告诉访问者，使用最新的浏览器可以获得更佳的浏览效果。代码清单 2-2 展示了如何在 canvas 中指定替代文本，当浏览器不支持 canvas 的时候会显示这些替代内容。

代码清单 2-2 在 canvas 元素中使用替代内容

```
<canvas>
  Update your browser to enjoy canvas!
</canvas>
```

除了上面代码中的文本外，同样还可以使用图片，不论是文本还是图片都会在浏览器不支持 canvas 元素的情况下显示出来。

canvas 元素的可访问性怎么样

“ 提供替代图像或替代文本引出了可访问性这个话题——很遗憾，这是 HTML5 Canvas 规范中明显的缺陷。例如，没有一种原生方法能够自动为已插入到 canvas 中的图片生成用于替换的文字说明。同样，也没有原生方法可以生成替代文字以匹配由 Canvas Text API 动态生成的文字。在写本书的时候，暂时还没有其他方法可以处理 canvas 中动态生成的内容，不过已经有工作组开始着手这方面的设计了。让我们一起期待吧。”

——Peter

2.1.6　CSS 和 **canvas**

同大多数 HTML 元素一样，canvas 元素也可以通过应用 CSS 的方式来增加边框，设置内边距、外边距等，而且一些 CSS 属性还可以被 canvas 内的元素继承。比如字体样式，在 canvas 内添加的文字，其样式默认同 canvas 元素本身是一样的。

此外，在 canvas 中为 context 设置属性同样要遵从 CSS 语法。例如，对 context 应用颜色和字体样式，跟在任何 HTML 和 CSS 文档中使用的语法完全一样。

2.1.7　浏览器对 HTML5 Canvas 的支持

除了 Internet Explorer 以外，其他所有浏览器现在都提供对 HTML5 Canvas 的支持。不过，随后我们会列出一部分还没有被普遍支持的规范，Canvas Text API 就是其中之一，但是作为一个整体，HTML5 Canvas 规范已经非常成熟，不会有特别大的改动了。从表 2-1 中可以看到，写本书的时候，已经有很多浏览器支持 HTML5 Canvas 了。

表2-1　浏览器对HTML5 Canvas的支持

浏 览 器	支持情况
Chrome	从1.0版本开始支持
Firefox	从1.5版本开始支持
Internet Explorer	不支持
Opera	从9.0版本开始支持
Safari	从1.3版本开始支持

从上面的表格中可以看出，在所有浏览器中，只有 Internet Explorer 不支持 HTML5 Canvas。如果需要在 Internet Explorer 中使用 canvas，可以选择使用名为 explorercanvas 的开源项目（http://code.google.com/p/explorercanvas）。使用 explorercanvas 时，需要先判断当前浏览器是否是

Internet Explorer，如果是则在页面中嵌入 `script` 标签来加载 explorercanvas。写法如下：

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


开发者迫切希望 Canvas 可以受到广泛支持，因此不断有项目启动来尝试解决老浏览器或者非标准浏览器不支持 Canvas 的问题。微软已经宣布 Internet Explorer 9 将支持 `canvas`，因此，所有主流浏览器都支持 `canvas` 已经指日可待了。

由于各家浏览器对 `canvas` 的支持程度有差异，所以最好在使用 API 之前，先测试一下 HTML5 Canvas 是否被支持。2.2.1 节会讲解怎样通过代码来检测浏览器支持 Canvas 的情况。

2.2 使用 HTML5 Canvas API

本节将深入探讨 HTML5 Canvas API。为此，我们将使用各种 HTML5 Canvas API 创建一幅类似于 LOGO 的图像，图像是森林场景，有树，还有适合长跑比赛的美丽跑道。虽然这个示例从平面设计的角度来看毫无竞争力，但却可以合理演示 HTML5 Canvas 的各种功能。

2.2.1 检测浏览器支持情况

在创建 HTML5 `canvas` 元素之前，首先要确保浏览器能够支持它。如果不支持，你就要为那些古董级浏览器提供一些替代文字。代码清单 2-3 就是检测浏览器支持情况的一种方法。

代码清单 2-3 检测浏览器支持情况

```
try {
  document.createElement("canvas").getContext("2d");
  document.getElementById("support").innerHTML =
    "HTML5 Canvas is supported in your browser.";
} catch (e) {
  document.getElementById("support").innerHTML = "HTML5 Canvas is not supported ↵
```

```
                                                    in your browser.";
}
```

上面的代码试图创建一个 canvas 对象，并且获取其上下文。如果发生错误，则可以捕获错误，进而得知该浏览器不支持 canvas。页面中预先放入了 ID 为 support 的元素，通过以适当的信息更新该元素的内容，可以反映出浏览器的支持情况。

以上示例代码能判断浏览器是否支持 canvas 元素，但不会判断具体支持 canvas 的哪些特性。写本书的时候，示例中使用的 API 已经很稳定并且各浏览器也都提供了很好的支持，所以通常不必担心这个问题。

此外，希望开发人员能够像代码清单 2-3 一样为 canvas 元素提供备用显示内容。

2.2.2　在页面中加入 canvas

在 HTML 页面中插入 canvas 元素非常直观。代码清单 2-4 就是一段可以被插入到 HTML 页面中的 canvas 代码。

代码清单 2-4　canvas 元素

```
<canvas height="200" width="200"></canvas>
```

以上代码会在页面上显示出一块 200×200 像素的“隐藏”区域。假如要为其增加一个边框，可以像代码清单 2-5 中的代码一样，用标准 CSS 边框属性来设置。

代码清单 2-5　带实心边框的 canvas 元素

```
<canvas id="diagonal" style="border: 1px solid;" width="200" height="200">
</canvas>
```

注意，上面的代码中增加了一个值为“diagonal”的 ID 特性，这么做的意义在于以后的开

发过程中可以通过 ID 来快速找到该 canvas 元素。对于任何 canvas 对象来说，ID 特性都是特别重要的，因为对 canvas 元素的所有操作都是通过脚本代码控制的，没有 ID 的话，想要找到要操作的 canvas 元素会很难。

代码清单 2-5 在浏览器中的执行效果如图 2-2 所示。

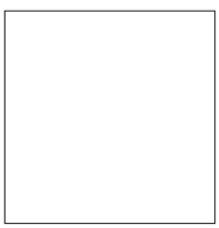

图 2-2 HTML 页面中的简单 canvas 元素

看起来好像没什么，但是就像那些艺术家说的，一张白纸可以画出最新最美的图画。现在，就让我们在这张“白纸”上作画吧。前面说过，在没有 HTML5 Canvas 的情况下，很难在页面上绘制一条对角线。现在我们来看看，有了 Canvas 以后，同样的事情会有多么简单。从代码清单 2-6 中可以看到，基于上面绘制的画布，仅仅使用几行代码就可以画出一条对角线。

代码清单 2-6 在 canvas 中绘制一条对角线

```
<script>
  function drawDiagonal() {
    // 取得 canvas 元素及其绘图上下文
    var canvas = document.getElementById('diagonal');
    var context = canvas.getContext('2d');

    //用绝对坐标来创建一条路径
    context.beginPath();
    context.moveTo(70, 140);
    context.lineTo(140, 70);

    // 将这条线绘制到 canvas 上
    context.stroke();
  }
```

```
    window.addEventListener("load", drawDiagonal, true);
</script>
```

仔细看一下上面这段绘制对角线的 JavaScript 代码。虽然简单，它却展示出了使用 HTML5 Canvas API 的重要流程。

首先通过引用特定的 canvas ID 值来获取对 canvas 对象的访问权。这段代码中 ID 就是 diagonal。接着定义一个 context 变量，调用 canvas 对象的 getContext 方法，并传入希望使用的 canvas 类型。代码清单中通过传入“2d”来获取一个二维上下文，这也是到目前为止唯一可用的上下文。

提示　规范未来的某个版本中可能会增加对三维上下文的支持。

接下来，基于这个上下文执行画线的操作。在代码清单中，调用了三个方法——beginPath、moveTo 和 lineTo，传入了这条线的起点和终点的坐标。

方法 moveTo 和 lineTo 实际上并不画线，而是在结束 canvas 操作的时候，通过调用 context.stroke()方法完成线条的绘制。图 2-3 显示了绘制结果。

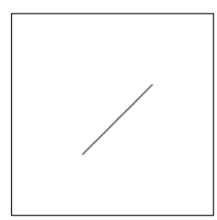

图 2-3　canvas 中的对角线

成功了！虽然从这条简单的线段怎么也想象不到最新最美的图画，不过与以前的拉伸图像、怪异的 CSS 和 DOM 对象以及其他怪异的实现形式相比，使用基本的 HTML 技术在任意两点间

绘制一条线段已经是非常大的进步了。从现在开始，就把那些怪异的做法永远忘掉吧。

从上面的代码清单中可以看出，canvas 中所有的操作都是通过上下文对象来完成的。在以后的 canvas 编程中也一样，因为所有涉及视觉输出效果的功能都只能通过上下文对象而不是画布对象来使用。这种设计使 canvas 拥有了良好的可扩展性，基于从其中抽象出的上下文类型，canvas 将来可以支持多种绘制模型。虽然本章经常提到对 canvas 采取什么样的操作，但读者应该明白，我们实际操作的是画布所提供的上下文对象。

如前面示例演示的那样，对上下文的很多操作都不会立即反映到页面上。beginPath、moveTo 以及 lineTo 这些函数都不会直接修改 canvas 的展示结果。canvas 中很多用于设置样式和外观的函数也同样不会直接修改显示结果。只有当对路径应用绘制（stroke）或填充（fill）方法时，结果才会显示出来。否则，只有在显示图像、显示文本或者绘制、填充和清除矩形框的时候，canvas 才会马上更新。

2.2.3 变换

现在我们探讨一下在 canvas 上绘制图像的另一种方式——使用变换（transformation）。接下来的代码清单显示结果跟上面是一样的，只是绘制对角线的代码不一样。这个简单示例可能会让你误认为使用变换增加了不必要的复杂性。事实并非如此，其实变换是实现复杂 canvas 操作的最好方式。在后面的示例中将会看到，我们使用了大量的变换，而这对熟悉 HTML5 Canvas API 的复杂功能是至关重要的。

也许了解变换最简单的方法（至少这种方法不涉及大量的数学公式，也不需手足并用地去解释）就是把它当成是介于开发人员发出的指令和 canvas 显示结果之间的一个修正层（modification layer）。不管在开发中是否使用变换，修正层始终都是存在的。

修正——在绘制系统中的说法是变换——在应用的时候可以被顺序应用、组合或者随意修改。每个绘制操作的结果显示在 canvas 上之前都要经过修正层去做修正。虽然这么做增加了额外的复杂性，但却为绘制系统添加了更为强大的功能，可以像目前主流图像编辑工具那样支持实时图像处理，所以 API 中这部分内容的复杂性是必要的。

不在代码中调用变换函数并不意味着可以提升 canvas 的性能。canvas 在执行的时候，变换会被呈现引擎隐式调用，这与开发人员是否直接调用无关。在接触最基本的绘制操作之前，提前了解系统背后的原理至关重要。

关于可重用代码有一条重要的建议：一般绘制都应从原点（坐标系中的 0,0 点）开始，应用变换（缩放、平移、旋转等），然后不断修改代码直至达到希望的效果。如图 2-4 所示。

代码清单 2-7 展示了如何使用最简单变换方法——translate 函数。

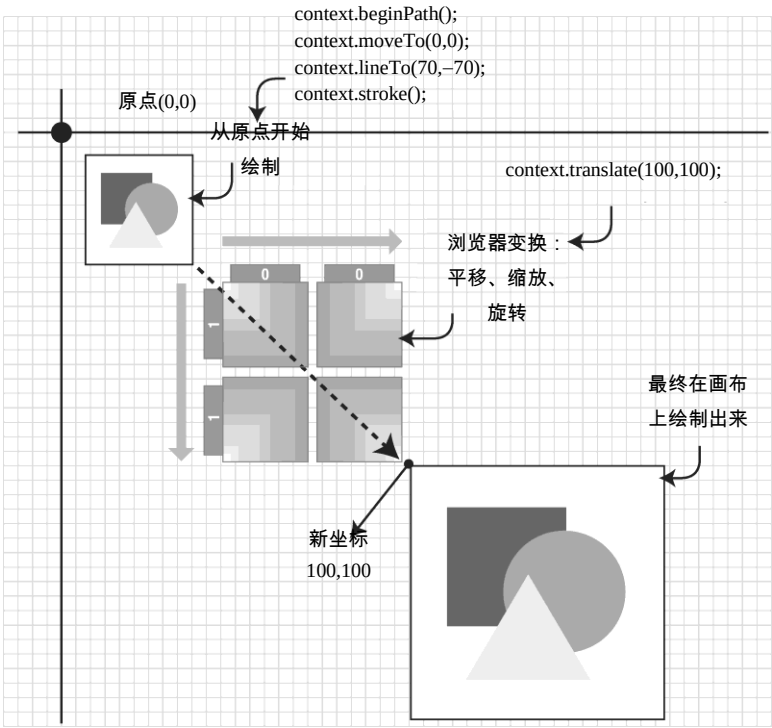

图 2-4　基于原点绘制和变换的示意图

代码清单 2-7　用变换的方式在 canvas 上绘制对角线

```
<script>
  function drawDiagonal() {
    var canvas = document.getElementById('diagonal');
    var context = canvas.getContext('2d');

    // 保存当前绘图状态
    context.save();

    // 向右下方移动绘图上下文
    context.translate(70, 140);

    // 以原点为起点，绘制与前面相同的线段
    context.beginPath();
    context.moveTo(0, 0);
    context.lineTo(70, -70);
    context.stroke();

    // 恢复原有的绘图状态
```

```
    context.restore();
  }

  window.addEventListener("load", drawDiagonal, true);
</script>
```

我们详细研究一下上面这段通过平移方式绘制对角线的 JavaScript 代码。

(1) 首先，通过 ID 找到并访问 canvas 对象。(ID 是 diagonal。)

(2) 接着通过调用 canvas 对象的 getContext 函数获取上下文对象。

(3) 接下来，保存尚未修改的 context，这样即使进行了绘制和变换操作，也可以恢复到初始状态。如果不保存，那么在进行了平移和缩放等操作以后，其影响会带到后续的操作中，而这不一定是我们所希望的。在变换前保存 context 状态可以方便以后恢复。

(4) 下一步是在 context 中调用 translate 函数。通过这个操作，当平移行为发生的时候，我们提供的变换坐标会被加到结果坐标 (对角线) 上，结果就是将要绘制的对角线移动到了新的位置上。不过，对角线呈现在 canvas 上是在绘制操作结束之后。

(5) 应用平移后，就可以使用普通的绘制操作来画对角线了。代码清单中调用了三个函数来绘制对角线——beginPath、moveTo 以及 lineTo。绘制的起点是原点 (0,0)，而非坐标点 (70,140)。

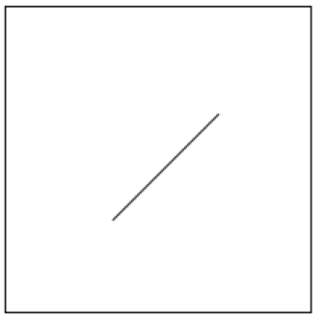

图 2-5 canvas 上平移过的对角线

(6) 在线条勾画出来之后，可以通过调用 context.stroke() 函数将其显示在 canvas 上。

(7) 最后，恢复 context 至原始状态，这样后续的 canvas 操作就不会被刚才的平移操作影

响了。图 2-5 显示了用这段代码绘制的对角线。

虽然新绘制的对角线看起来跟前面的一模一样，但这次绘制使用了强大的变换功能。学习完本章接下来的内容，就会明白变换的强大之处。

2.2.4　路径

关于绘制线条，我们还能提供很多有创意的方法。不过，现在应该进一步学习稍复杂点的图形：路径。HTML5 Canvas API 中的路径代表你希望呈现的任何形状。本章对角线示例就是一条路径，你可能已经注意到了，代码中调用 `beginPath` 就说明是要开始绘制路径了。实际上，路径可以要多复杂有多复杂：多条线、曲线段，甚至是子路径。如果想在 `canvas` 上绘制任意形状，那么你需要重点关注路径 API。

按照惯例，不论开始绘制何种图形，第一个需要调用的就是 `beginPath`。这个简单的函数不带任何参数，它用来通知 `canvas` 将要开始绘制一个新的图形了。对于 `canvas` 来说，`beginPath` 函数最大的用处是 `canvas` 需要据此来计算图形的内部和外部范围，以便完成后续的描边和填充。

路径会跟踪当前坐标，默认值是原点。`canvas` 本身也跟踪当前坐标，不过可以通过绘制代码来修改。

调用了 `beginPath` 之后，就可以使用 `context` 的各种方法来绘制想要的形状了。到目前为止，我们已经用到了几个简单的 `context` 路径函数。

- `moveTo(x, y)`：不绘制，只是将当前位置移动到新的目标坐标`(x,y)`。

- lineTo(x, y)：不仅将当前位置移动到新的目标坐标(x,y)，而且在两个坐标之间画一条直线。

简而言之，上面两个函数的区别在于：moveTo 就像是提起画笔，移动到新位置，而 lineTo 告诉 canvas 用画笔从纸上的旧坐标画条直线到新坐标。不过，再次提醒一下，不管调用它们哪一个，都不会真正画出图形，因为我们还没有调用 stroke 或者 fill 函数。目前，我们只是在定义路径的位置，以便后面绘制时使用。

下一个特殊的路径函数叫做 closePath。这个函数的行为同 lineTo 很像，唯一的差别在于 closePath 会将路径的起始坐标自动作为目标坐标。closePath 还会通知 canvas 当前绘制的图形已经闭合或者形成了完全封闭的区域，这对将来的填充和描边都非常有用。

此时，可以在已有的路径中继续创建其他的子路径，或者随时调用 beginPath 重新绘制新路径并完全清除之前的所有路径。

跟了解所有复杂系统一样，最好的方式还是实践。现在，我们先不管那些线条的例子，使用 HTML5 Canvas API 开始创建一个新场景——带有长跑跑道的树林。权且把这个图案当成是我们长跑比赛的标志吧。同其他的画图方式一样，我们将从基本元素开始。在这幅图中松树的树冠最简单。代码清单 2-8 演示了如何在 canvas 上绘制一颗松树的树冠。

代码清单 2-8　用于绘制树冠轮廓的函数

```
function createCanopyPath(context) {
  // 绘制树冠
  context.beginPath();

  context.moveTo(-25, -50);
```

```
  context.lineTo(-10, -80);
  context.lineTo(-20, -80);
  context.lineTo(-5, -110);
  context.lineTo(-15, -110);

  // 树的顶点
  context.lineTo(0, -140);

  context.lineTo(15, -110);
  context.lineTo(5, -110);
  context.lineTo(20, -80);
  context.lineTo(10, -80);
  context.lineTo(25, -50);

  // 连接起点，闭合路径
  context.closePath();
}
```

从上面的代码中可以看到，我们用到的仍然是前面用过的移动和画线命令，只不过调用次数多了一些。这些线条表现的是树冠的轮廓，最后我们闭合了路径。我们为这棵树的底部留出了足够的空间，后面几节将在这里的空白处画上树干。代码清单 2-9 演示如何使用树冠绘制函数将树的简单轮廓呈现到 canvas 上。

代码清单 2-9　在 canvas 上画树的函数

```
function drawTrails() {
  var canvas = document.getElementById('trails');
  var context = canvas.getContext('2d');

  context.save();
  context.translate(130, 250);

  // 创建表现树冠的路径
  createCanopyPath(context);

  // 绘制当前路径
  context.stroke();
  context.restore();
}
```

这段代码中所有的调用想必大家已经很熟悉了。先获取 canvas 的上下文对象，保存以便后续使用，将当前位置变换到新位置，画树冠，绘制到 canvas 上，最后恢复上下文的初始状态。

图 2-6 展示了我们的绘画技艺，一条简单的闭合路径表现了树冠。以后我们会详细扩展这段代码，现在算是一个好的开始。

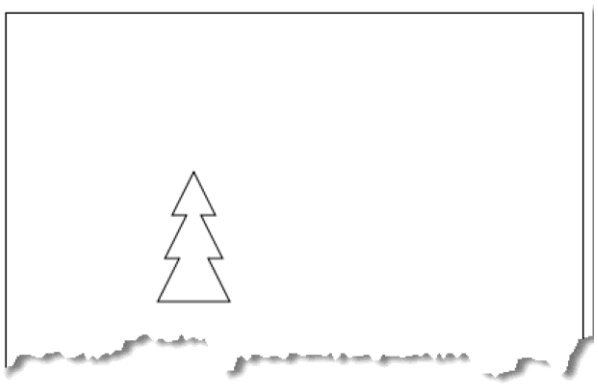

图 2-6 表现树冠的简单路径

2.2.5 描边样式

如果开发人员只能绘制直线，而且只能使用黑色，HTML5 Canvas API 就不会如此强大和流行。下面我们就使用描边样式让树冠看起来更像是树。代码清单 2-10 展示了一些基本命令，其功能是通过修改 `context` 的属性，让绘制的图形更好看。

代码清单 2-10 使用描边样式

```
// 加宽线条
context.lineWidth = 4;

// 平滑路径的接合点
context.lineJoin = 'round';

// 将颜色改成棕色
context.strokeStyle = '#663300';

// 最后，绘制树冠
context.stroke();
```

设置上面的这些属性可以改变以后将要绘制的图形外观，这个外观起码可以保持到我们将

context 恢复到上一个状态。

首先，我们将线条宽度加粗到 4 像素。

接着，我们将 lineJoin 属性设置为 round，这是修改当前形状中线段的连接方式，让拐角变得更圆滑；也可以把 lineJoin 属性设置成 bevel 或者 miter（相应的 context.miterLimit 值也需要调整）来变换拐角样式。

最后，通过 strokeStyle 属性改变了线条的颜色。在这个例子中，我们使用了 CSS 值来设置颜色，不过在后面几节中，我们将看到 strokeStyle 的值还可以用于生成特殊效果的图案或者渐变色。

还有一个没有用到的属性——lineCap，可以把它的值设置为 butt、square 或者 round，以此来指定线条末端的样式。哦，示例中的线是闭合的，没有端点。图 2-7 就是我们加工过的树冠，与之前扁平的黑线相比，现在是一条更粗、更平滑的棕色线条。

图 2-7　为树冠应用了描边样式

2.2.6　填充样式

正如你所期望的那样，能影响 canvas 的图形外观的并非只有描边，另一个常用于修改图形的方法是指定如何填充其路径和子路径。从代码清单 2-11 中可以看到，用宜人的绿色填充树冠有多么简单。

代码清单 2-11 使用填充样式

```
// 将填充色设置为绿色并填充树冠
context.fillStyle = '#339900';
context.fill();
```

首先，我们将 fillStyle 属性设置成合适的颜色。(在后面，我们将看到还可以使用渐变色或者图案填充。) 然后，只要调用 context 的 fill 函数就可以让 canvas 对当前图形中所有的闭合路径内部的像素点进行填充。结果如图 2-8 所示。

图 2-8 填充后的树冠

由于我们是先描边后填充，因此填充会覆盖一部分描边路径。我们示例中的路径是 4 像素宽，这个宽度是沿路径线居中对齐的，而填充是把路径轮廓内部所有像素全部填充，所以会覆盖描边

路径的一半。如果希望看到完整的描边路径，可以在绘制路径（调用 `context.stroke()`）之前填充（调用 `context.fill()`）。

2.2.7　填充矩形区域

每棵树都有一个强壮的树干。我们在原始图形中为树干预留了足够的空间。从代码清单 2-12 中可以看到，通过 `fillRect` 函数可以画出树干。

代码清单 2-12　调用 `fillRect` 函数

```
// 将填充色设为棕色
context.fillStyle = '#663300';

// 填充用作树干的矩形区域
context.fillRect(-5, -50, 10, 50);
```

在上面的代码中，再次将棕色作为填充颜色。不过跟上次不一样的是，我们不用 `lineTo` 功能显式画树干的边角，而是使用 `fillRect` 一步到位画出整个树干。调用 `fillRect` 并设置 *x*、*y* 两个位置参数和宽度、高度两个大小参数，随后，Canvas 会马上使用当前的样式进行填充。

虽然示例中没有用到，但与之相关的函数还有 `strokeRect` 和 `clearRect`。`strokeRect` 的作用是基于给出的位置和坐标画出矩形的轮廓，`clearRect` 的作用是清除矩形区域内的所有内容并将它恢复到初始状态，即透明色。

canvas 动画

“ 在 HTML5 Canvas API 中，`canvas` 的清除矩形功能是创建动画和游戏的核心功能。通过反复绘制和清除 `canvas` 片段，就可能实现动画效果，互联网上有很多这样的例子。但是，如果希

望创建运行起来比较流畅的动画，就需要使用剪裁（clipping）[①]功能了，有可能还需要二次缓存 canvas，以便最小化由于频繁的清除动作而导致的画面闪烁。本书中不会专门讲解动画，我们鼓励大家自己探索学习。”

——Brain[②]

图 2-9 显示的是基于树冠图形添加的、一次填充的树干。

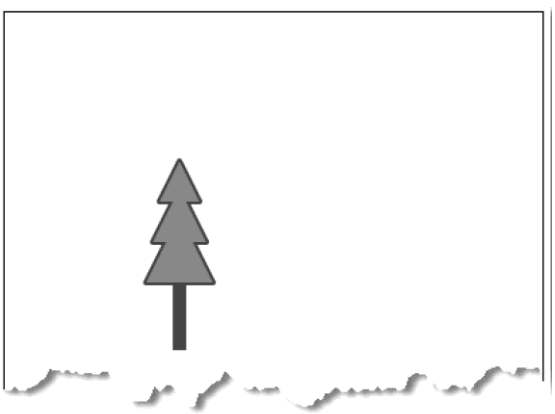

图 2-9 带有矩形树干的树

2.2.8 绘制曲线

这个世界，特别是自然界，并不是只有直线和矩形。canvas 提供了一系列绘制曲线的函数。我们将用最简单的曲线函数——二次曲线，来绘制我们的林荫小路。代码清单 2-13 演示了如何添加两条二次曲线。

① 剪裁（clipping）：本节没有介绍剪裁功能，此功能常用于创建动画，作者本意是希望读者能够自己探索相关的知识领域。——译者注

② Brain，本书的作者之一。——译者注

代码清单 2-13　绘制曲线

```
// 保存 canvas 的状态并绘制路径
context.save();

context.translate(-10, 350);
context.beginPath();

// 第一条曲线向右上方弯曲
context.moveTo(0, 0);
context.quadraticCurveTo(170, -50, 260, -190);

// 第二条曲线向右下方弯曲
context.quadraticCurveTo(310, -250, 410,-250);

// 使用棕色的粗线条来绘制路径
context.strokeStyle = '#663300';
context.lineWidth = 20;
context.stroke();

// 恢复之前的 canvas 状态
context.restore();
```

跟以前一样，第一步要做的事情是保存当前 canvas 的 context 状态，因为我们即将变换坐标系并修改轮廓设置。要画林荫小路，首先要把坐标恢复到修正层的原点，向右上角画一条曲线。

从图 2-10 中可以看到，quadraticCurveTo 函数绘制曲线的起点是当前坐标，带有两组（x,y）参数。第二组是指曲线的终点。第一组代表控制点（control point）。所谓的控制点位于曲线的旁边（不是曲线之上），其作用相当于对曲线产生一个拉力。通过调整控制点的位置，就可以改变曲线的曲率。在右上方再画一条一样的曲线，以形成一条路。然后，像之前描边树冠一样把这条路绘制到 canvas 上（只是线条更粗了）。

HTML5 Canvas API 的其他曲线功能还涉及 bezierCurveTo、arcTo 和 arc 函数。这些函

数通过多种控制点（如半径、角度等）让曲线更具可塑性。图 2-11 显示了绘制在 canvas 上的两条曲线，看起来就像是穿过树林的小路一样。

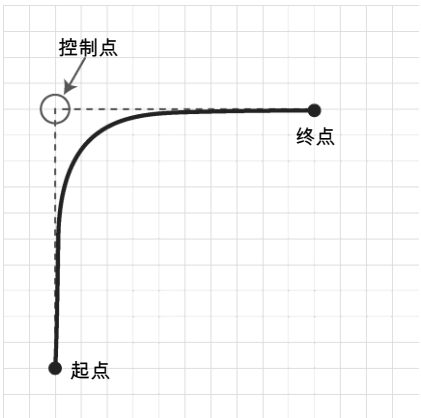

图 2-10 曲线的起点、终点和控制点

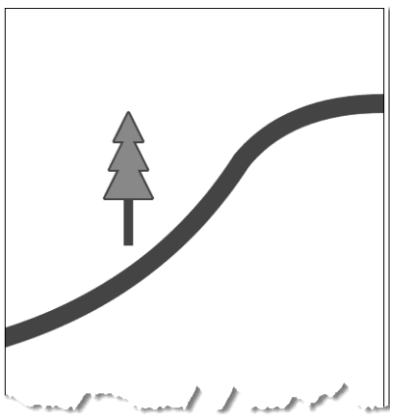

图 2-11 组成小路的曲线

2.2.9 在 canvas 中插入图片

在 canvas 中显示图片非常简单。可以通过修正层为图片添加印章、拉伸图片或者修改图片等，并且图片通常会成为 canvas 上的焦点。用 HTML5 Canvas API 内置的几个简单命令可以轻松地为 canvas 添加图片内容。

不过，图片增加了 canvas 操作的复杂度：必须等到图片完全加载后才能对其进行操作。浏览器通常会在页面脚本执行的同时异步加载图片。如果试图在图片未完全加载之前就将其呈现到 canvas 上，那么 canvas 将不会显示任何图片。因此，开发人员要特别注意，在呈现之前，应确保图片已经加载完毕。

我们的示例将加载一张树皮纹理的图片作为树干以供 canvas 使用。为保证在呈现之前图片

已完全加载，我们提供了回调，即仅当图像加载完成时才执行后续代码，如代码清单 2-14 所示。

代码清单 2-14　加载图像

```
// 加载图片 bark.jpg
var bark = new Image();
bark.src = "bark.jpg";

// 图片加载完成后，将其显示在 canvas 上
bark.onload = function () {
  drawTrails();
}
```

从上面的代码中可以看到，我们为 bark.jpg 图片添加了 `onload` 处理函数，以保证仅在图像加载完成时才调用主 `drawTrails` 函数。这样做可以保证后续的调用能够把图片正常显示出来，如代码清单 2-15 所示。

代码清单 2-15　在 `canvas` 上显示图像

```
// 用背景图案填充作为树干的矩形
context.drawImage(bark, -5, -50, 10, 50);
```

在这段代码里，我们用纹理贴图替换了之前调用 `fillRect` 函数的填充来作为新的树干。尽管替换的动作很小，但 `canvas` 上显示出来的树干更有质感。注意，在 `drawImage` 函数中，除了图片本身外，还指定了 `x`、`y`、`width` 和 `height` 参数。这些参数会对贴图进行调整以适应预定的 10×50 像素树干区域。我们还可以把原图的尺寸传进来，以便在裁切区域内对图片进行更多控制。

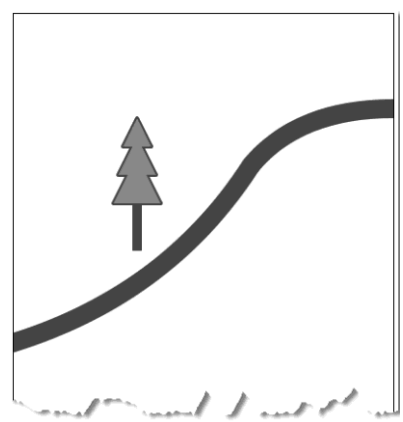

图 2-12　使用了树干贴图的树

在图 2-12 中可以看到，同之前用矩形填充的方式相比，树干的变化不大。

2.2.10 渐变

对树干还是不满意？其实我也是。我们使用另一种可以让树干变得稍微好看点的绘制方法：渐变。渐变是指在颜色集上使用逐步抽样算法，并将结果应用于描边样式和填充样式中。使用渐变需要三个步骤：

(1) 创建渐变对象；

(2) 为渐变对象设置颜色，指明过渡方式；

(3) 在 `context` 上为填充样式或者描边样式设置渐变。

可以将渐变看做是颜色沿着一条线进行缓慢地变化。例如，如果为渐变对象提供了 A、B 两个点，不论是绘制还是填充，只要从 A 移动到 B，都会带来颜色的变化。

要设置显示哪种颜色，在渐变对象上使用 `addColorStop` 函数即可。这个函数允许指定两个参数：颜色和偏移量。颜色参数是指开发人员希望在偏移位置描边或填充时所使用的颜色。偏移量是一个 0.0 到 1.0 之间的数值，代表沿着渐变线渐变的距离有多远。

假如要建立一个从点(0,0)到点(0,100)的渐变，并指定在 0.0 偏移位置使用白色，在 1.0 偏移位置使用黑色。当使用绘制或者填充的动作从(0,0)画到(0,100)后，就可以看到颜色从白色（起始位置）渐渐转变成了黑色（终止位置）。

除了可以变换成其他颜色外，还可以为颜色设置 alpha 值（例如透明），并且 alpha 值也是可以变化的。为了达到这样的效果，需要使用颜色值的另一种表示方法，例如内置 alpha 组件的 CSS `rgba` 函数。

下面我们通过示例来详细了解如何使用两个渐变来填充（相应的函数为 `fillRect`）矩形区域，并形成最终的树干，见代码清单 2-16。

代码清单 2-16　使用渐变

```
// 创建用作树干纹理的三阶水平渐变
var trunkGradient = context.createLinearGradient(-5, -50, 5, -50);

// 树干的左侧边缘是一般程度的棕色
trunkGradient.addColorStop(0, '#663300');

// 树干中间偏左的位置颜色要淡一些
trunkGradient.addColorStop(0.4, '#996600');

// 树干右侧边缘的颜色要深一些
trunkGradient.addColorStop(1, '#552200');

// 使用渐变色填充树干
context.fillStyle = trunkGradient;
context.fillRect(-5, -50, 10, 50);

// 接下来，创建垂直渐变，以用作树冠在树干上投影
var canopyShadow = context.createLinearGradient(0, -50, 0, 0);

// 投影渐变的起点是透明度设为 50%的黑色
canopyShadow.addColorStop(0, 'rgba(0, 0, 0, 0.5)');

// 方向垂直向下，渐变色在很短的距离内迅速渐变至完全透明，这段长度之外的树干上没有投影
canopyShadow.addColorStop(0.2, 'rgba(0, 0, 0, 0.0)');

// 在树干上填充投影渐变
context.fillStyle = canopyShadow;
context.fillRect(-5, -50, 10, 50);
```

如图 2-13 所示，使用了两个渐变后，最终绘制出来的树干有了平滑的光照效果。现在，树干看起来更平滑，同时树干上也有了轻微的阴影效果。我们把这幅图保存起来吧。

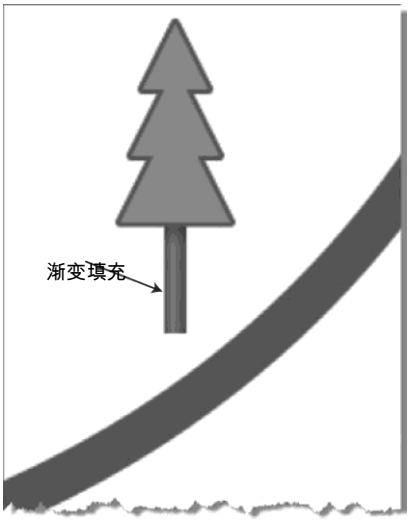

图 2-13 有渐变树干的树

除了我们刚才用到的线性渐变以外，HTML5 Canvas API 还支持放射性渐变，所谓放射性渐变就是颜色会介于两个指定圆间的锥形区域平滑变化。放射性渐变和线性渐变使用的颜色终止点是一样的，不过参数如代码清单 2-17 所示。

代码清单 2-17 使用放射性渐变的示例

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

代码中，前三个参数代表以(x0,y0)为圆心，r0 为半径的圆，后三个参数代表以(x1,y1)为圆心，r1 为半径的另一个圆。渐变会在两个圆中间的区域出现。

2.2.11 背景图

直接绘制图像有很多用处，但在某些情况下，像 CSS 那样使用图片作为背景也非常有用。我们已经了解了如何使用加粗的颜色描边和填充。在描边和填充的时候，HTML5 Canvas API 还

支持图片平铺。

现在我们把林荫小路变得崎岖一点。这次不再对曲线跑道进行描边，而是使用背景图片填充的方法。为了达到预想的效果，我们将已经作废的树干图片（我们已经有了“渐变”树干）替换成砾石图片。我们将调用 createPattern 函数来替代之前的 drawImage 函数，如代码清单 2-18 所示。

代码清单 2-18　使用背景图片

```
// 加载砾石背景图
var gravel = new Image();
gravel.src = "gravel.jpg";
gravel.onload = function () {
    drawTrails();
}

// 用背景图替代棕色粗线条
context.strokeStyle = context.createPattern(gravel, 'repeat');
context.lineWidth = 20;
context.stroke();
```

从上面的代码中可以看到，绘制的时候还是使用 stroke()函数，只不过这次我们先设置了 context 上的 strokeStyle 属性，把调用 context.createPattern 的返回值赋给该属性。再次强调一下，图片必须提前加载完毕，以便 canvas 执行后续操作。context.createPattern 的第二个参数是重复性标记，可以在表 2-2 中选择合适的值。

表2-2　重复性参数

平铺方式	意　义
repeat	（默认值）图片会在两个方向平铺
repeat-x	横向平铺
repeat-y	纵向平铺
no-repeat	图片只显示一次，不平铺

图 2-14 显示了应用背景图片方式画出的小路。

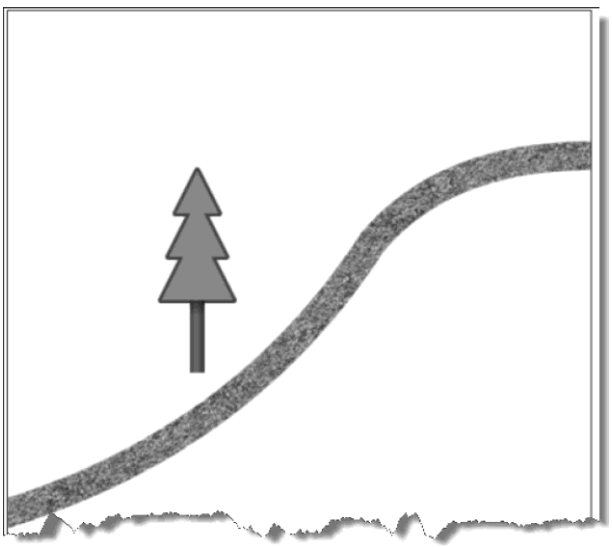

图 2-14 使用平铺图片作为背景的小路

2.2.12 缩放 canvas 对象

树林里怎么可能只有一棵树呢？现在我们来解决这个问题。为简单起见，我们计划把示例代码中用于绘制树的操作独立出来，当做一个单独的例程，称为 `drawTree`，见代码清单 2-19。

代码清单 2-19 绘制树对象的函数

```
// 创建树对象绘制函数，以便重用
function drawTree(context) {
  var trunkGradient = context.createLinearGradient(-5, -50, 5, -50);
  trunkGradient.addColorStop(0, '#663300');
  trunkGradient.addColorStop(0.4, '#996600');
  trunkGradient.addColorStop(1, '#552200');
  context.fillStyle = trunkGradient;
  context.fillRect(-5, -50, 10, 50);

  var canopyShadow = context.createLinearGradient(0, -50, 0, 0);
  canopyShadow.addColorStop(0, 'rgba(0, 0, 0, 0.5)');
  canopyShadow.addColorStop(0.2, 'rgba(0, 0, 0, 0.0)');
  context.fillStyle = canopyShadow;
```

```
    context.fillRect(-5, -50, 10, 50);

    createCanopyPath(context);
    context.lineWidth = 4;
    context.lineJoin = 'round';
    context.strokeStyle = '#663300';
    context.stroke();

    context.fillStyle = '#339900';
    context.fill();
}
```

可以看到，drawTree 函数包括了之前绘制树冠、树干和树干渐变的所有代码。为了在新的位置画出大一点的树，我们将使用另一种变换方式——缩放函数 context.scale，如代码清单 2-20 所示。

代码清单 2-20　绘制树对象

```
// 在(130,250)的位置绘制第一棵树
context.save();
context.translate(130, 250);
drawTree(context);
context.restore();

// 在(260,500)的位置绘制第二棵树
context.save();
context.translate(260, 500);

// 将第二棵树的宽高分别放大至原来的 2 倍
context.scale(2, 2);
drawTree(context);
context.restore();
```

scale 函数带有两个参数来分别代表在 *x*、*y* 两个维度的值。每个参数在 canvas 显示图像的时候，向其传递在本方向轴上图像要放大（或者缩小）的量。如果 *x* 值为 2，就代表所绘制图像中全部元素都会变成两倍宽，如果 *y* 值为 0.5，绘制出来的图像全部元素都会变成之前的一半高。使用这些函数，就可以很方便的在 canvas 上创建出新的树，如图 2-15 所示。

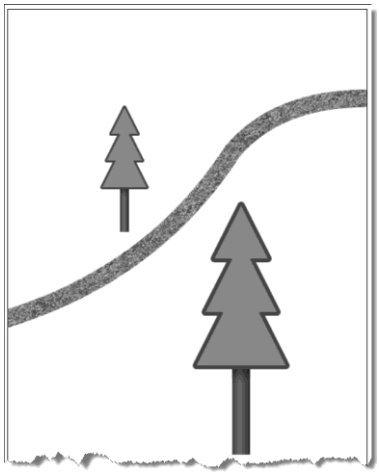

图 2-15 放大后的树

始终在原点执行图形和路径的变换操作

“ 示例中演示了为什么要在原点执行图形和路径的变换操作，执行完后再统一平移。理由就是缩放（scale）和旋转（rotate）等变换操作都是针对原点进行的。

如果对一个不在原点的图形进行旋转变换，那么 rotate 变换函数会将图形绕着原点旋转而不是在原地旋转。与之类似，如果进行缩放操作时没有将图形放置到合适的坐标上，那么所有路径坐标都会被同时缩放。取决于缩放比例的大小，新的坐标可能会全部超出 canvas 范围，进而给开发人员带来困惑，为什么我的缩放操作会把图像删了？”

——Brian

2.2.13 Canvas 变换

变换操作并不限于缩放和平移，我们可以使用函数 context.rotate(angle) 来旋转图像，

甚至可以直接修改底层变换矩阵以完成一些高级操作，如剪裁图像的绘制路径。如果想旋转图像，只需执行代码清单 2-21 所示的一系列操作即可。

代码清单 2-21　旋转图像

```
context.save();

// 旋转角度参数以弧度为单位
context.rotate(1.57);
context.drawImage(myImage, 0, 0, 100, 100);

context.restore();
```

在代码清单 2-22 中，我们将演示如何对路径坐标进行随意变换，以从根本上改变现有树的路径显示，并最终创建一个阴影效果。

代码清单 2-22　一种变换的使用方法

```
// 创建用于填充树干的三阶水平渐变色
// 保存 canvas 的当前状态
context.save();

// X 值随着 Y 值的增加而增加，借助拉伸变换，可以创建一棵用作阴影的倾斜的树
context.transform(1, 0, -0.5, 1, 0, 0);

// 在 Y 轴方向，将阴影的高度压缩为原来的 60%
context.scale(1, 0.6);

// 使用透明度为 20%的黑色填充树干
context.fillStyle = 'rgba(0, 0, 0, 0.2)';
context.fillRect(-5, -50, 10, 50);

// 使用已有的阴影效果重新绘制树
createCanopyPath(context);
context.fill();

// 恢复之前的 canvas 状态
context.restore();
```

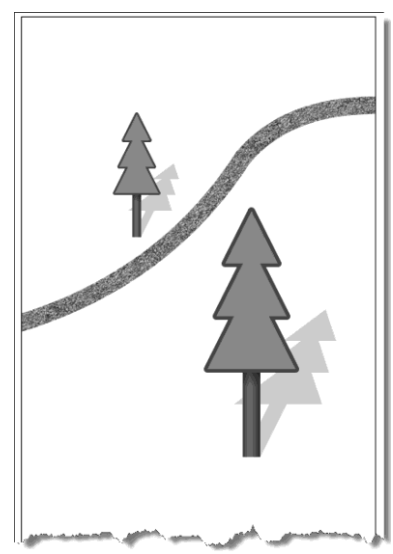

图 2-16　带有变换阴影的树

你可以像上面那样直接修改context变换矩阵，前提是要熟悉二维绘图系统中的矩阵变换。分析这种变换背后的数

学含义，可以看出我们通过调整与Y轴值相对应的参数改变了X轴的值，这样做目的是为了拉伸出一棵灰色的树做阴影。接下来，我们按照60%的比例将剪裁出的树缩小到了合适的尺寸。

注意，剪裁过的“阴影”树会先被显示出来，这样一来，真正的树就会按照Z轴顺序（canvas中对象的重叠顺序）显示在阴影的上面。此外，树影的填充用到了CSS的RGBA特性，通过特性我们将透明度值设为正常情况下的20%。至此，带有半透明效果的树影就做好了。将其应用于已经缩放过的树上，效果如图2-16所示。

2.2.14 Canvas 文本

在作品即将完成之际，我们要在图像的上部添加一个别致的标题，以此来向大家演示 HTML5 Canvas API 强大的文本功能。需要特别注意的是，操作 canvas 文本的方式与操作其他路径对象的方式相同：可以描绘文本轮廓和填充文本内部；同时，所有能够应用于其他图形的变换和样式都能用于文本。

context 对象的文本绘制功能由两个函数组成：

- fillText(text,x, y,maxwidth)
- strokeText(text,x,y,maxwidth)

两个函数的参数完全相同，必选参数包括文本参数以及用于指定文本位置的坐标参数。maxwidth 是可选参数，用于限制字体大小，它会将文本字体强制收缩到指定尺寸。此外，还有一个 measureText 函数可供使用，该函数会返回一个度量对象，其中包含了在当前 context 环境下指定文本的实际显示宽度。

为了保证文本在各浏览器下都能正常显示，Canvas API 为 context 提供了类似于 CSS 的属

性，以此来保证实际显示效果的高度可配置。如表 2-3。

表2-3　文本呈现相关的**context**属性

属　性	值	备　注
font	CSS字体字符串	例如 :italic Arial ,scan-serif
textAlign	start、end、left、right、center	默认是start
textBaseline	top、hanging、middle、alphabetic、ideographic、bottom	默认是alphabetic

对上面这些 context 属性赋值能够改变 context，而访问 context 属性可以查询到其当前值。在代码清单 2-23 中，我们首先创建了一段使用 Impact 字体的大字号文本，然后使用已有的树皮图片作为背景进行填充。为了将文本置于 canvas 的上方并居中，我们定义了最大宽度和 center（居中）对齐方式。

代码清单 2-23　使用 canvas 文本

```
// 在 canvas 上绘制标题文本
context.save();

// 字号为 60px，字体为 impact
context.font = "60px impact";

// 将文本填充为棕色
context.fillStyle = '#996600';
// 将文本设为居中对齐
context.textAlign = 'center';

// 在 canvas 顶部中央的位置，以大字体的形式显示文本
context.fillText('Happy Trails!', 200, 60, 400);
context.restore();
```

结果如图 2-17 所示，林间小路的美景马上就增添了几分欢快的味道。

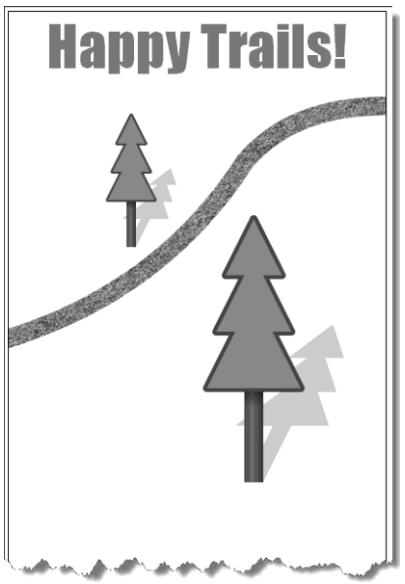

图 2-17 使用背景图片填充的文本

2.2.15 应用阴影

最后，我们将使用内置的 Canvas Shadow API 为文本添加模糊阴影效果。虽然我们能够通过 HTML5 Canvas API 将阴影效果应用于之前执行的任何操作中，但与很多图形效果的应用类似，阴影效果的使用也要把握好“ 度”。

可以通过几种全局 `context` 属性来控制阴影，见表 2-4。

表2-4 阴影属性

属 性	值	备 注
`shadowColor`	任何CSS中的颜色值	可以使用透明度（`alpha`）
`ShadowOffsetX`	像素值	值为正数，向右移动阴影；值为负数，向左移动阴影
`shadowOffsetY`	像素值	值为正数，向下移动阴影；值为负数，向上移动阴影
`shadowBlur`	高斯模糊值	值越大，阴影边缘越模糊

`shadowColor` 或者其他任意一项属性的值被赋为非默认值时，路径、文本和图片上的阴影

效果就会被触发。代码清单 2-24 显示了如何为文本添加阴影效果。

代码清单 2-24　应用阴影效果

```
// 设置文字阴影的颜色为黑色，透明度为 20%
context.shadowColor = 'rgba(0, 0, 0, 0.2)';

// 将阴影向右移动 15px，向上移动 10px
context.shadowOffsetX = 15;
context.shadowOffsetY = -10;

// 轻微模糊阴影
context.shadowBlur = 2;
```

执行上述代码后，canvas 渲染器会自动应用阴影效果，直到恢复 canvas 状态或者重置阴影属性。添加阴影后的效果如图 2-18 所示。

图 2-18　有阴影效果的标题

如你所见，由 CSS 生成的阴影只有位置上的变化，而无法与变换生成的阴影（树影）保持同步。为了一致起见，在 canvas 上绘制阴影时，应该尽量只用一种方法。

2.2.16　像素数据

Canvas API 最有用的特性之一是允许开发人员直接访问 canvas 底层像素数据。这种数据访问是双向的：一方面，可以以数值数组形式获取像素数据；另一方面，可以修改数组的值以将其应用于 canvas。实际上，放弃本章之前讨论的渲染调用，也可以通过直接调用像素数据的相关方法来控制 canvas。这要归功于 context API 内置的三个函数。

第一个是 context.getImageData(sx, sy, sw, sh)。这个函数返回当前 canvas 状态并以数值数组的方式显示。具体来说，返回的对象包括三个属性。

- width：每行有多少个像素。
- height：每列有多少个像素。
- data：一维数组，存有从 canvas 获取的每个像素的 RGBA 值。该数组为每个像素保存了四个值——红、绿、蓝和 alpha 透明度。每个值都在 0 到 255 之间。因此，canvas 上的每个像素在这个数组中就变成了四个整数值。数组的填充顺序是从左到右，从上到下（也就是先第一行再第二行，依此类推），如图 2-19 所示。

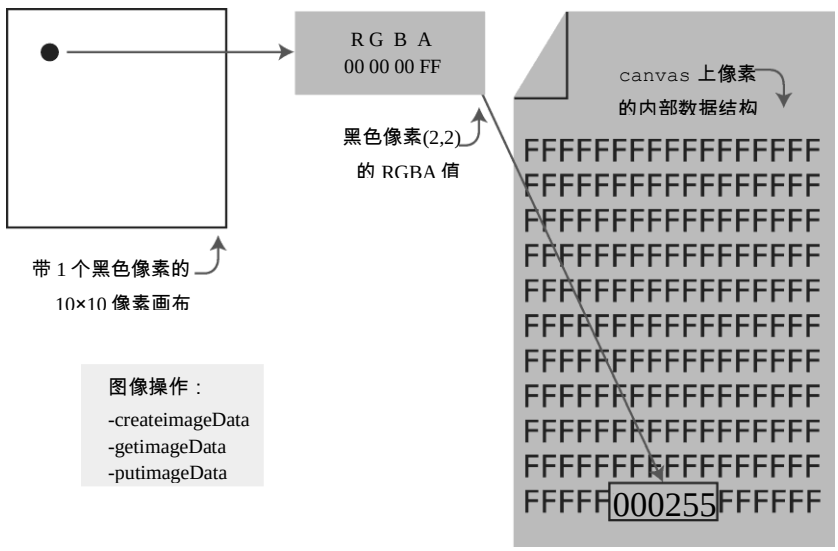

图 2-19　像素数据及其内部数据结构

getImageData 函数有四个参数，该函数只返回这四个参数所限定的区域内的数据。只有被 x、y、width 和 height 四个参数框定的矩形区域内的 canvas 上的像素才会被取到，因此要想获取所有像素数据，就需要这样传入参数：getImageData(0, 0, canvas.width, canvas.height)。

因为每个像素由四个图像数据表示，所以要计算指定的像素点对应的值是什么就有点头疼。不要紧，下面有公式。

在给定了 width 和 height 的 canvas 上，在坐标(x ,y)上的像素的构成如下。

- 红色部分：((width * y) + x) * 4
- 绿色部分：((width * y) + x) * 4 + 1
- 蓝色部分：((width * y) + x) * 4 + 2

- 透明度部分：((width * y) + x) * 4 + 3

一旦可以通过像素数据的方式访问对象，就可以通过数学方式轻松修改数组中的像素值，因为这些值都是从 0 到 255 的简单数字。修改了任何像素的红、绿、蓝和 alpha 值之后，可以通过第二个函数来更新 canvas 上的显示，那就是 context.putImageData(imagedata, dx, dy)。

putImageData 允许开发人员传入一组图像数据，其格式与最初从 canvas 上获取来的是一样的。这个函数使用起来非常方便，因为可以直接用从 canvas 上获取数据加以修改然后返回。一旦这个函数被调用，所有新传入的图像数据值就会立即在 canvas 上更新显示出来。dx 和 dy 参数可以用来指定偏移量，如果使用，则该函数就会跳到指定的 canvas 位置去更新显示传进来的像素数据。

最后，如果想预先生成一组空的 canvas 数据，则可调用 context.createImageData(sw, sh)，这个函数可以创建一组图像数据并绑定在 canvas 对象上。这组数据可以像先前那样处理，只是在获取 canves 数据时，这组图像数据不一定会反映 canvas 的当前状态。

2.2.17 Canvas 的安全机制

上面讨论了直接操纵像素数据的方法，在这里有必要重点提醒一下，大多数开发者都会合法使用像素数据操作。尽管如此，还是会有人出于某些邪恶的目的利用这种从 canvas 直接获取并且修改数据的能力。出于这个原因，origin-clean canvas 的概念应运而生，换句话说，如果 canvas 中的图片并非来自包含它的页面所在的域，页面中的脚本将不能取得其中的数据。

如图 2-20 所示，如果来自 http://www.example.com 的页面包含 canvas 元素，那么页面中的

代码完全有可能在 canvas 里面呈现来自 http://www.remote.com 的图片。毕竟在任何 Web 页面中显示其他远程网站的图片都是完全可接受的。

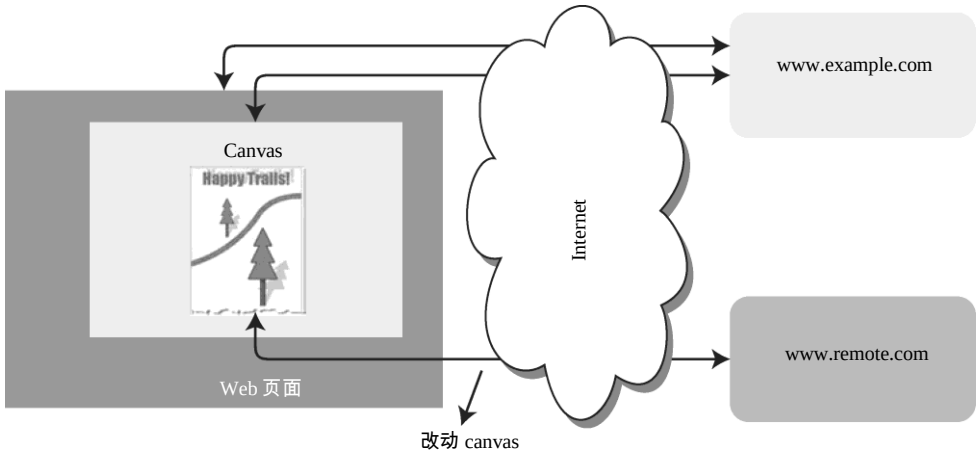

图 2-20　本地和远程图像来源

然而，在没有 Canvas API 以前，无法使用编程的方式获取下载图片的像素信息。来自其他网站的私有图片可以显示在本地，但无法被读取或者复制。如果允许脚本读取本地之外的图像数据，那么整个网络中的用户照片以及其他敏感的在线图片文档将被“ 无限制地共享”。

为了避免如此，在 getImageData 函数被调用的时候，如果 canvas 中的图像来自其他域，就会抛出安全异常。这样的话，只要不获取显示着其他域中图片的 canvas 的数据，那么就可以随意呈现这些远程图片。在开发的过程中要注意这个限制条件，使用安全的渲染方式。

2.3　使用 HTML5 Canvas 创建应用

使用 Canvas API 可以创建许多种应用：图形、图表、图片编辑等，然而最奇妙的一个应用是修改或者覆盖已有内容。最流行的覆盖图被称为热点图。虽然热点图听起来是度量温度的意思，

不过这里的热度可以用于任何可测量的活动。地图上活跃程度高的部分使用暖色标记(例如红色、黄色或白色), 活跃程度低的部分不显示颜色变化，或者显示浅浅的黑色或灰色。

举个例子，热点图可以用在城市地图上来标记交通路况，或者在世界地图上显示风暴的活动情况。在 HTML5 中这些应用都非常容易实现，只需要将 `canvas` 叠放在地图上显示即可。实际上就是用 `canvas` 覆盖地图，然后再基于相应的活动数据绘制出不同的热度级别。

现在，我们使用已经学过的 Canvas API 知识来绘制一个简单的热点图。这个示例中，热度数据不是来源于外部，而是来源于我们的鼠标在地图上的移动情况。鼠标移动到某个区域，会使这个区域的“ 热度” 增加。将鼠标放在特定区域不动会让该区域“ 温度” 迅速增长至极限。为了示范，我们将在一个“ 难以名状” 的地图上进行热点图的覆盖演示 (见图 2-21)。

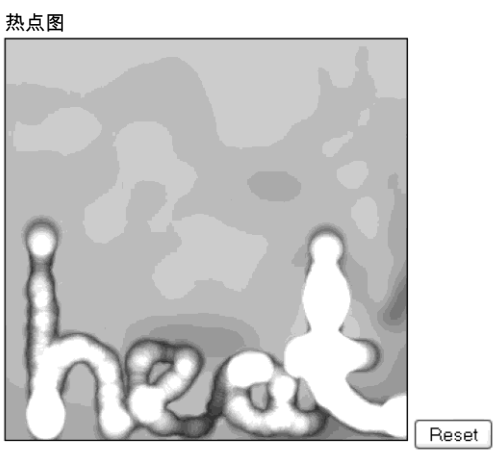

图 2-21 热点图应用

上面看到的是热点图应用的最终效果，下面我们深入分析实现代码。和往常一样，你可以在

线查看或下载示例的源代码。

这份源码中，我们从 HTML 元素开始分析。为了演示效果，这个 HTML 中只包含了标题（Heatmap）、画布和按钮（Reset，用来复位热点图）。canvas 上显示的背景图片文件是 mapbg.jpg，通过代码清单 2-25 中的 CSS 代码应用到 canvas 中。

代码清单 2-25　热点图的 canvas 元素

```
<style type="text/css">
  #heatmap {
      background-image: url("mapbg.jpg");
  }
</style>

<h2>Heatmap </h2>
<canvas id="heatmap" class="clear" style="border: 1px solid ; " height="300"□↵
 width="300"> </canvas>
<button id="resetButton">Reset</button>
```

我们还声明了一些变量，然后对其进行了初始化，以备后用。

```
var points = {};
var SCALE = 3;
var x = -1;
var y = -1;
```

接下来，为了支持全局绘制操作，我们将为 canvas 设置一个高透明值，并且设置为混合模式，让新的绘制操作点亮底层的像素而不是替换它们。

然后，如代码清单 2-26 所示，我们会设置 addToPoint 函数，在鼠标移动的时候或者每隔 1/10 s 的时间调用它以改变显示效果。

代码清单 2-26　loadDemo 函数

```
function loadDemo() {
  document.getElementById("resetButton").onclick = reset;
```

```
  canvas = document.getElementById("heatmap");
  context = canvas.getContext('2d');
  context.globalAlpha = 0.2;
  context.globalCompositeOperation = "lighter"

function sample() {
  if (x != -1) {
    addToPoint(x,y)
  }
  setTimeout(sample, 100);
}

canvas.onmousemove = function(e) {
  x = e.clientX - e.target.offsetLeft;
  y = e.clientY - e.target.offsetTop;
  addToPoint(x,y);
}

  sample();
}
```

使用 canvas 的 clearRect 函数，就可以让用户在点击 Reset 按钮的时候，将整个 canvas 区域清空并重置回原始状态（见代码清单 2-27）。

代码清单 2-27 reset 函数

```
function reset() {
  points = {};
  context.clearRect(0,0,300,300);
  x = -1;
  y = -1;
}
```

接下来我们建立一张颜色查找表，以便在 canvas 上执行绘制操作的时候使用。代码清单 2-28 中列出了颜色亮度由低到高的范围，不同的颜色值会被用来代表各种不同的热度。intensity 的值越大，返回的颜色越亮。

代码清单 2-28 getColor 函数

```
function getColor(intensity) {
  var colors = ["#072933", "#2E4045", "#8C593B", "#B2814E", "#FAC268", "#FAD237"];
```

```
    return colors[Math.floor(intensity/2)];
  }
```

不管什么时候，只要鼠标移过或者悬停在 canvas 的某个区域，就会有一个点被绘制出来。鼠标在特定区域中停留的时间越长，这个点就越大（同时越亮）。像代码清单 2-29 中所示，使用 context.arc 函数根据特定的半径值绘制圆，通过传到 getColor 函数中的半径值来判断，半径越大画出的圆越亮、颜色越热。

代码清单 2-29　drawPoint 函数

```
function drawPoint(x, y, radius) {
  context.fillStyle = getColor(radius);
  radius = Math.sqrt(radius)*6;

  context.beginPath();
  context.arc(x, y, radius, 0, Math.PI*2, true);

  context.closePath();
  context.fill();
}
```

在 addToPoint 函数中（每次鼠标移动或者悬停的时候都会调用这个函数），canvas 特定点上的热度值会升高并保存下来。代码清单 2-30 中显示了最高的热度值是 10。给定像素点的当前热度值一旦被检测到，那么相应的像素以及相关的热度、半径值就会被传递到 drawPoint 函数中。

代码清单 2-30　addToPoint 函数

```
function addToPoint(x, y) {
  x = Math.floor(x/SCALE);
  y = Math.floor(y/SCALE);

  if (!points[[x,y]]) {
    points[[x,y]] = 1;
  } else if (points[[x,y]]==10) {
    return
```

```
    } else {
      points[[x,y]]++;
    }
    drawPoint(x*SCALE,y*SCALE, points[[x,y]]);
  }
```


最后，还注册了一个 loadDemo 函数，用来在窗口加载完毕的时候调用。

```
window.addEventListener("load", loadDemo, true);
```

总而言之，这些 100 行左右的代码可以向大家证明：使用 HTML5 Canvas API，在短短的时间内，不用任何插件或者外部技术就可以实现非常高级的功能。另外，我们身边的各种数据源又是无穷无尽的，既然将它们可视化如此简单方便，那我们还等什么呢？

进阶功能之全页玻璃窗

在前面的示例中，我们看到了如何把 canvas 应用于图片上。其实，我们还可以把 canvas 应用于整个浏览器窗口或者其中的一部分之上——这种技术通常被称作“玻璃窗”（glass pane）。在 Web 页面中放置玻璃窗后，我们可以做很多之前意想不到的事情。

例如，可以编写函数来获取页面中所有 DOM 元素的绝对位置，然后创建循序渐进的帮助功能，从而引导 Web 应用的用户，一步一步地教他们学会操作。

另外，可以借助 canvas 玻璃窗并利用鼠标事件让用户在 Web 页面上绘制反馈。不过，使用此功能时，请记住以下三点。

- 需要将 canvas 的 CSS 属性 position 设置为 absolute，并且指定 canvas 的位置、宽度和高度。如果没有明确的宽度和高度值，那么 canvas 将保持默认尺寸——0 像素。
- 别忘了将 canvas 的 CSS 属性 z-index 的值设置得大一些，使其能够盖在所有显示内容

的上面。如果 canvas 被其他内容覆盖在最下面，就毫无用武之地了。

- 设置 canvas 玻璃窗会阻塞后续的事件访问 ,因此需要提醒开发人员 ,不需要时要记得 关窗”。

2.4　小结

到目前为止，我们看到了 HTML5 Canvas API 提供的强大功能，利用它可以直接修改 Web 应用的外观，而不必再像以前那样借助于各种第三方技术。HTML5 Canvas API 还可以通过自由组合图像、渐变和复杂路径等方式来创建你能想到的几乎所有效果。需要注意的是，绘制工作通常应以原点为起点，在展现图像之前要先完成加载，而在使用外部来源的图片时则要留心。如果能学会驾驭 canvas，那么你就能在网页上创建出前所未见的应用。