用 HTML Canvas 创建一个图片浏览器

颜 林
2009 年 1 月 10 日发布

HTML Canvas 介绍

HTML5 是目前正在讨论的新一代 HTML 标准，它代表了现在 Web 领域的最新的发展方向。在 HTML5 标准中，加入了新的多样的内容描述标签，直接支持表单验证，视频和音频标签，网页元素的拖拽，离线存储，工作线程等等。当然，其中一个最令人激动的新特性就是新的标签类型 Canvas，开发人员可以通过该标签，在网页上直接用脚本进行绘图，产生各种 2D 渲染的效果。所以有人预言，HTML5 将是 Flash 和 Silverlight 的“杀手”。从 Firefox 1.5 开始就已经支持 Canvas，Safari 也是很早就开始支持 Canvas。新的浏览器比如 Chrome 也是从一开始就支持。但遗憾的是，到目前为止，IE 一直不支持该标准。

下面内容将通过如何用 Canvas 来制作一个图片浏览器的具体实例，来说明 Canvas 的各种 API，如何使用这些 API 以及如何应用到工程中去。本文将首先介绍如何创建图片浏览器的网页和 JavaScript 类，介绍整体界面的设计，然后介绍如何用 Canvas 的 API 来绘制 2D 图形，然后介绍如何在 Canvas 上加载和绘制图像，接下来本例会在图片浏览器中加入其他基于 Canvas 的效果，最后是总结和展望。

创建图片浏览器框架

创建文件

首先我们创建一个新的 html 文件 thumbnail.html，加入如清单 1 所示的内容：

清单 1.thumbnail.html

```
<!DOCTYPE HTML>
<html>
<head>
 <title>Canvas Based Thumbnail</title>
 <style type="text/css">
   body {
   background: black;
   color: white;
  font: 24pt Baskerville, Times, Times New
Roman, serif;
   padding: 0;
   margin: 0;
   overflow: hidden;
   }
 </style>
 <script type="text/javascript"
src="thumbnail.js"></script>
</head>
<body>
 <canvas id="canvas"></canvas>
</body>
</html>
```

这里我们可以看到，canvas 是 html 的一个新的标签，其用法和其他标签一样，只不过它的高和宽有独立的属性而不是在 css 定义的。如果我们要设置一个 Canvas 区域的宽高，必须定义为 <canvas width="100" height="100"> 而不能是 <canvas style="width:100,height:100">。在上面的 html 文件中我们没有直接定义 Canvas 区域的大小，而是在 JavaScript 中动态定义，下面将要详细说明。

现在我们创建一个新的 JavaScript 文件 thumbnail.js 来在 Canvas 中绘制图像，我们设计一个 thumbnail 类，该类可以处理用户事件，绘制图形，显示图像。然后在 window.onload 事件中加载该类，代码如清单 2 所示：

清单 2 .thumbnail.js

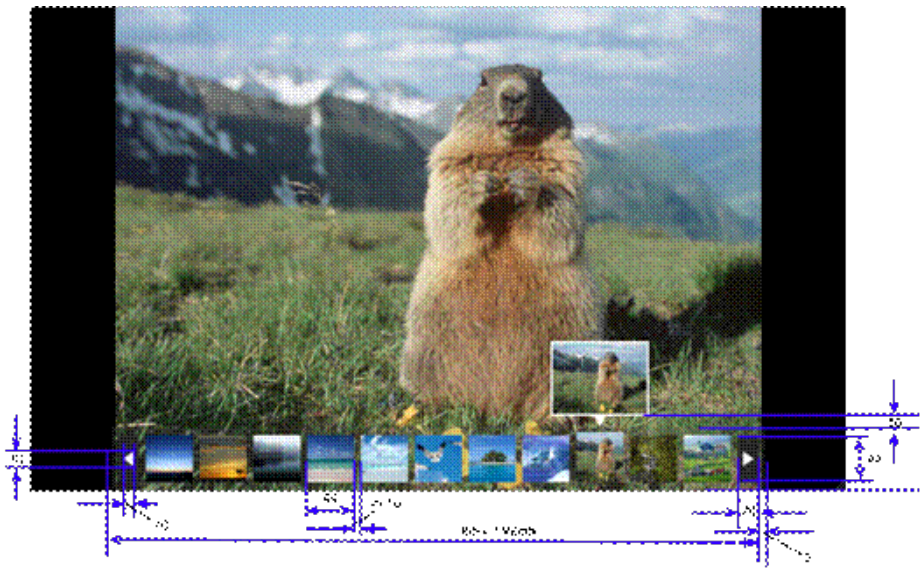
```
function thumbnail() {
   this.load = function()
   }
}

window.onload = function() {
   thumb = new thumbnail();
   thumb.load();
}
```

代码定义了一个初始化函数 load，并且声明了 thumbnail 类，这样我们就可以在 thumbnail 类中添加代码，在 Canvas 上绘制各种图形以及图像了。

设计界面

我们为这个图片浏览页面设计这样一种界面，图片通过缩放占满全部网页空间，在中间下方，绘制一个导航栏，显示缩略图，当点击缩略图时，该图片显示到网页中。同时，如果鼠标悬停在某个缩放图上，则显示一个大一点的预览图用来供用户预览。在导航栏的左右两边，添加 2 个按钮，用于翻页显示上一页和下一页的缩略图。对导航栏上所有控件的尺寸大小和位置如图 1 所示的方案。这样，我们就可以按照该方案在 Canvas 上绘制这些控件了。

图 1. 界面设计



用 Canvas 绘制图形

绘制导航框

首先我们绘制如图 2 所示的一个导航栏。在左右两边各有一个按钮，按钮上显示一个三角形的指示图形。当鼠标放到一个按钮上时，按钮的背景色能变成高亮的颜色，显示当前选中的按钮。

图 2. 导航框

清单 3 显示了绘制图 2 所示的导航栏的代码片段。

清单 3 . 绘制导航框代码

```
function thumbnail() {
   const NAVPANEL_COLOR = 'rgba(100, 100, 100,
0.2)';    // 导航栏背景色
   const NAVBUTTON_BACKGROUND = 'rgb(40, 40,
40)';  // 导航栏中 button 的背景色
   const NAVBUTTON_COLOR = 'rgb(255, 255,
255)';   //button 的前景色
   const NAVBUTTON_HL_COLOR = 'rgb(100, 100,
100)';   //button 高亮时的前景色

   var canvas =
document.getElementById('canvas');   // 获得
canvas 对象
   var context = canvas.getContext('2d');
// 获得上下文对象

   // 绘制左边 button
   function paintLeftButton(navRect, color) {
       //left button
       lButtonRect = {
           x: navRect.x + NAVBUTTON_XOFFSET,
           y: navRect.y + NAVBUTTON_YOFFSET,
           width: NAVBUTTON_WIDTH,
           height: navRect.height -
NAVBUTTON_YOFFSET * 2
       }

       context.save();
       context.fillStyle = color;
       context.fillRect(lButtonRect.x,
lButtonRect.y,
lButtonRect.width, lButtonRect.height);

       //left arrow
       context.save();
       context.fillStyle = NAVBUTTON_COLOR;
       context.beginPath();
       context.moveTo(lButtonRect.x +
NAVBUTTON_ARROW_XOFFSET,
lButtonRect.y + lButtonRect.height/2);
       context.lineTo(lButtonRect.x +
lButtonRect.width - NAVBUTTON_ARROW_XOFFSET,
```

```
lButtonRect.y + NAVBUTTON_ARROW_YOFFSET);
        context.lineTo(lButtonRect.x +
lButtonRect.width - NAVBUTTON_ARROW_XOFFSET,
lButtonRect.y + lButtonRect.height -
NAVBUTTON_ARROW_YOFFSET);
        context.lineTo(lButtonRect.x +
NAVBUTTON_ARROW_XOFFSET,
lButtonRect.y + lButtonRect.height/2);
        context.closePath();
        context.fill();
        context.restore();

        context.restore();
    }
```

如上所述，在页面 html 中我们声明了 <canvas> 元素。当需要在 <canvas> 区域绘制图形时，我们需要获得绘制图形的上下文对象，在一个上下文对象中，保存了当前的初始坐标位置，颜色，风格等等信息。这里我们通过如清单 4 的语句获取 Canvas 的 2 维绘图的上下文对象。

清单 4. 获得绘图上下文

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

获得上下文对象之后，我们就可以通过 Canvas 提供的 API 来进行绘画了。在清单 5 中，我们使用了矩形的绘制函数来绘制整个导航栏背景和左右两个按钮。所下所示：

清单 5. 矩形绘制函数

```
fillRect(x,y,width,height)：  绘制一个填充的矩形
strokeRect(x,y,width,height)：  给一个矩形描边
clearRect(x,y,width,height)：  清除该矩形内所有内容使之透明
```

例如，我们要绘制一个简单的矩形可以用如清单 6 所示代码：

清单 6. 绘制简单矩形

```
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');
context.fillStyle = 'black';
context.fillRect(0, 0, 50, 50);
context.clearRect(0, 0, 20, 20);
context.strokeRect(0, 0, 20, 20);
```

我们绘制该导航框时，需要在左右两边各绘制一个三角形，对于除了矩形以外的所有多边形，必须得通过路径来绘制，常用的路径相关函数有：

清单 7. 绘制路径函数

```
beginPath(): 开始一段路径
closePath(): 结束一段路径
moveTo(x,y)  : 移动起始点到某点
lineTo(x,y) :  绘制线段到目标点
```

这样，我们在绘制三角形的时候，只需要确定三个顶点的坐标，就可以通过 lineTo 函数绘制三条线段，但是，我们还需要一个函数在该三角形区域内填充颜色，这样需要用到填充和描边的函数和样式：

清单 8. 填充和描边样式

```
fillStyle = color :  设置填充颜色
storkeStyle = color : 设置描变颜色
```

这里 color 值可以是标准的 CSS 颜色值，还可以通过 rgba 函数设置透明度。我们可以如下设置：

清单 9. 填充样式举例

```
context.fillStyle = "white";
context.strokeStyle = "#FFA500";
context.fillStyle = "rgb(255,165,0)";
context.fillStyle = "rgba(255,165,0,1)";
```

同样，当需要填充颜色样式或者描边时，有如下函数：

清单 10. 填充和描边函数

```
stroke() : 按照当前描边样式描边当前路径
fill() : 按照当前填充样式填充路径所描述的形状
```

这样，用上述几个函数，我们绘制一个三角形时，可以用如下语句：

清单 11. 绘制三角形代码

```
var canvas =
document.getElementById('canvas');
var context = canvas.getContext('2d');
context.fillStyle = 'black';
context.beginPath();
context.moveTo(0,0);
context.lineTo(10,0);
context.lineTo(10,10);
context.lineTo(0,0);
context.closePath();
context.fill();
```

在清单 3 中，我们还声明了一些常量来定义导航栏的各种控件的大小，其中长度值都是以像素为单位的。这样我们绘制了整个导航栏，但我们现在需要当鼠标放到按钮上时，按钮的前景色能够高亮，显示当前选中的按钮。这就需要我们在代码中响应用户事件，并进行不同类型的绘制。

响应用户事件

响应用户事件和普通的 DOM 编程类似，如清单 12 所示：

清单 12. 响应鼠标移动时间

```
var lastMousePos;    // 当前鼠标位置
this.load = function() {
    //event binding
    canvas.onmousemove = onMouseMove;
}
function onMouseMove(event) {
    lastMousePos = {x:event.clientX,
y:event.clientY};
    paint();
}
function pointIsInRect(point, rect) {
    return (rect.x < point.x && point.x <
rect.x + rect.width &&
            rect.y < point.y && point.y <
rect.y + rect.height);
}
function paint() {
    context.clearRect(0, 0, canvas.width,
canvas.height);
    var paintInfo = {inLeftBtn:false,
inRightBtn:false}

    if (lastMousePos && navRect && lButtonRect
&& rButtonRect) {
        if (pointIsInRect(lastMousePos,
navRect)) {
            paintInfo.inLeftBtn =
pointIsInRect(lastMousePos, lButtonRect);
            paintInfo.inRightBtn =
pointIsInRect(lastMousePos, rButtonRect);
```

```
            }
        }
        paintNavigator(paintInfo);
    }
```

这样我们就绘制了一个完整的导航栏，它能够响应鼠标移动事件，并高亮当前选中的按钮。下面我们需要加载和显示图片，这就需要用到 Canvas 的绘制图像函数。

用 Canvas 绘制图像

加载和显示图像

加载和显示图像的代码片段如清单 13 所示：

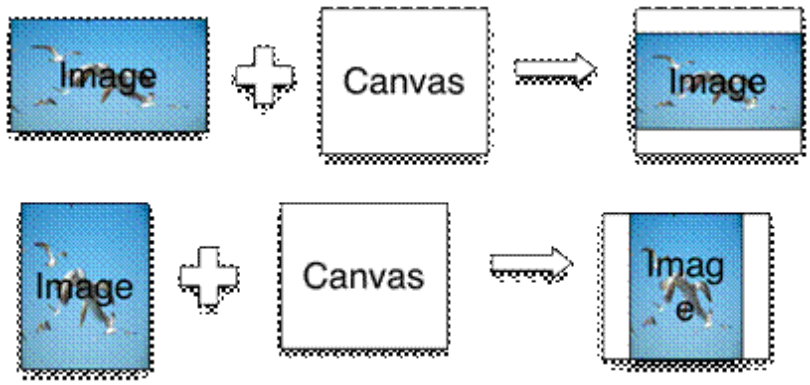
清单 13. 加载和显示图像

```
    const PAINT_INTERVAL = 20;  // 循环间隔
    const PAINT_SLOW_INTERVAL = 20000;
    const IDLE_TIME_OUT = 3000;  // 空闲超时时间

    // 定义全部图片 URL 数组，在本例中，所有图片保存在和网页同目录中
    var imageLocations = [
    '2006109173628.jpg',
    '2007310132939.jpg',
    '200733094828-1.jpg'
    ];
   // 加载图片
    function loadImages() {
        var total = imageLocations.length;
        var imageCounter = 0;
        var onLoad = function(err, msg) {
            if (err) {
                console.log(msg);
            }
            imageCounter++;
            if (imageCounter == total) {
                loadedImages = true;
            }
        }

        for (var i = 0; i < imageLocations.length; i++) {
            var img = new Image();
            img.onload = function() { onLoad(false); };
            img.onerror = function() { onLoad(true, e);};
            img.src = imageLocations[i];
            images[i] = img;
        }
```

```
    }
    // 绘制图片
    function paintImage(index) {
        if (!loadedImages)
            return;
        var image = images[index];
        var screen_h = canvas.height;
        var screen_w = canvas.width;
        var ratio =
getScaleRatio({width:image.width,
height:image.height},
{width:screen_w, height:screen_h});
    var img_h = image.height * ratio;
    var img_w = image.width * ratio;

    context.drawImage(image, (screen_w -
img_w)/2, (screen_h - img_h)/2, img_w, img_h);
    }
```

在清单 13 的代码中，我们更新了主绘制函数 paint，加入了 paintImage 函数，在 paintImage 函数中，利用 Canvas 的 drawImage 函数，在整个 Canvas 区域，尽可能大地缩放图片并显示在 Canvas 中，其最佳缩放比例如图 3 所示：

图 3. 最佳缩放比例示例



这里缩放比例是通过本例所定义的函数 getScaleRatio 来获得的，其详细代码见附件。这样我们可以在 Canvas 上绘制图像，绘制图像的函数定义如下：

清单 14. 绘制图像函数

```
drawImage(image, x, y)    image 为一个图像或者
Canvas 对象，x，y 为图片所要放至位置的左上角坐标
```

但该函数还无法满足我们的要求，我们需要缩放图片到一个最佳大小，这就需要 Canvas 绘制图片函数的另外一种形式：

清单 15. 绘制图像函数 2

```
drawImage(image, x, y, width, height)
width, height 为图像在目标 Canvas 上的大小
```

该函数将图片缩放到 width 和 height 所指定的大小并显示出来。我们通过函数 getScaleRatio 来计算最佳缩放大小，然后就可以通过如清单 15 所示来绘制最佳大小的图片。

绘制图片需要传入一个 image 对象，它一般是一个图片或者 Canvas 对象。也就是说你可以从一个 URL 中下载图片显示在 Canvas 中，也可以在一个 Canvas 中显示另外一个 Canvas 中绘制的图形。通过如清单 16 所示的代码来加载图片：

清单 16. 加载图片代码

```
var onLoad = function(err, msg) {
    if (err)   console.log(msg);
}
var img = new Image();
img.onload = function() { onLoad(false); };
img.onerror = function() { onLoad(true, e);};
img.src = ‘ myImage.png ’ ;  // 设置源路径
```

在整个程序中，我们利用了 setInterval 函数加入了一个定时器来触发主循环，用于不断循环等待全部图片加载。当等待时间超过一个阀值之后，主循环进入 idle 状态，该循环不仅能够用于等待全部图片加载，也可以用于绘制动画效果，我们在后面将会讲到如何利用该主循环来制作动态效果。

绘制缩略图

下一步需要在导航栏中绘制每个图片的缩略图，该缩略图必须按照最优的大小和间隔排列在导航栏中，同时缩略图必须经过裁剪，获得最优的显示区域。整体效果如图 4 所示：

图 4. 缩略图效果

实现代码片段如清单 17 所示：

清单 17. 缩略图代码

```
   const HL_OFFSET = 3;
   const THUMBNAIL_LENGTH = NAVPANEL_HEIGHT -
NAVBUTTON_YOFFSET*2;   // 缩略图显示区域的高度
   const MIN_THUMBNAIL_LENGTH = 10;  // 最小缩
略图间隔

   var currentImage = 0;   // 当前图片序号
   var firstImageIndex = 0;  // 当前缩略图中第一
张图片序号
   var thumbNailCount = 0;  // 当前显示的缩略图数
   var maxThumbNailCount = 0;  // 最大能够显示的
缩略图数
   // 绘制缩略图
   function paintThumbNails(inThumbIndex) {
       if (!loadedImages)
           return;

       if(inThumbIndex != null) {
           inThumbIndex -= firstImageIndex;
       } else {
           inThumbIndex = -1;
       }

       var thumbnail_length = rButtonRect.x -
lButtonRect.x - lButtonRect.width;
       maxThumbNailCount =
Math.ceil(thumbnail_length /
THUMBNAIL_LENGTH);
       var offset = (thumbnail_length -
THUMBNAIL_LENGTH * maxThumbNailCount) /
(maxThumbNailCount + 1);
       if (offset < MIN_THUMBNAIL_LENGTH) {
           maxThumbNailCount =
Math.ceil(thumbnail_length/ (THUMBNAIL_LENGTH
+
MIN_THUMBNAIL_LENGTH));
           offset = (thumbnail_length -
THUMBNAIL_LENGTH * maxThumbNailCount) /
(maxThumbNailCount + 1);
       }

       thumbNailCount = maxThumbNailCount >
```

```
imageCount - firstImageIndex?
imageCount - firstImageIndex:
maxThumbNailCount;

       imageRects = new Array(thumbNailCount);

       for (var i = 0; i < thumbNailCount;
i++) {
           image = images[i+firstImageIndex];
           context.save();
           var x = lButtonRect.x +
lButtonRect.width +
(offset+THUMBNAIL_LENGTH)*i;
           srcRect =
getSlicingSrcRect({width:image.width,
height:image.height},
{width:THUMBNAIL_LENGTH, height:
THUMBNAIL_LENGTH});
           imageRects[i] = {
               image:image,
               rect: {
                   x:x+offset,
                   y:inThumbIndex == i?
navRect.y+NAVBUTTON_YOFFSET-HL_OFFSET:
navRect.y+NAVBUTTON_YOFFSET,
                   height: THUMBNAIL_LENGTH,
                   width: THUMBNAIL_LENGTH
               }
           }

           context.translate(x, navRect.y);
           context.drawImage(image, srcRect.x,
srcRect.y,
srcRect.width, srcRect.height,
offset, imageRects[i].rect.y - navRect.y,
THUMBNAIL_LENGTH, THUMBNAIL_LENGTH);
           context.restore();
       }
   }
```

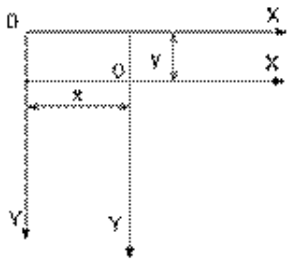
清单 17 的代码使用了 Canvas 中坐标转换的方法来绘制每张缩略图。转换坐标函数如清单 18 所示：

清单 18. 转换坐标函数

```
translate(x, y)  x 为横轴偏移方向大小，y 为纵轴方向偏移大小
```

其原理如图 5 所示：

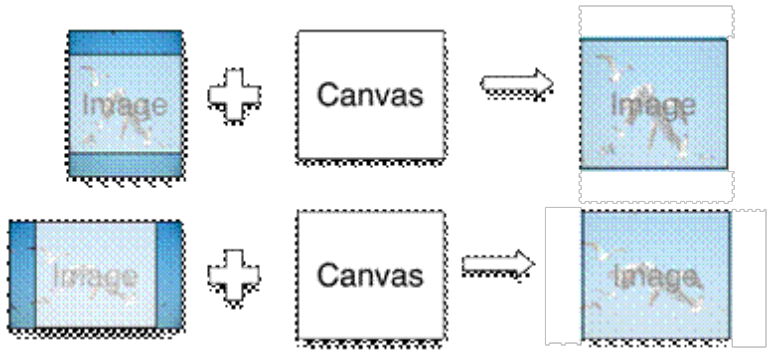
图 5. 转换坐标



Canvas 绘图的坐标系和大部分操作系统绘图的坐标系一致，都是左上角为原点，向右为 x 方向，向下为 y 方向。从图 5 中我们看出，新的坐标原点平移到了 (x,y) 位置，后面的 Canvas 绘图函数都是以新的原点为基准绘图。清单 17 在绘制每张缩略图时，首先转换原点到缩略图的左上角，然后在固定的 x 和 y 坐标位置显示图片，将这个过程做成一个循环，就绘制了所有等间距的缩略图。

将图片显示到缩略图中，我们还需要把图片缩放到其中较短的一边能够和缩略图的边长重合，同时截去超出缩略图大小的图片部分，从而达到最优的显示缩略图的效果。其示意图如图 6 所示。

图 6. 截取最佳图片部分



为了获得这种最优的缩略图显示效果，我们需要获得如下信息：1. 原图中应该截取哪些部分图片；2 · 缩放多大的比例到目标区域中。本例定义了函数 getSlicingSrcRect 实现了这个功能，它返回一个 rect 对象，包括了应该截取原图的哪些区域，其详细代码见附件。但还需要一个函数来将这个截取的图片部分缩放到目标区域中，这就用到了 Canvas 绘制图像函数 drawImage 的另外一种形式：

清单 19. 绘制图像函数 3

```
drawImage(image, sx, sy, sWidth, sHeight, dx,
dy, dWidth, dHeight)
sx, sy, sWidth, sHeight 为原图中的需要截取的区域，
dx, dy, dWidth, dHeight 为目标区域的位置大小
```

该函数截取原图片的部分区域，然后缩放显示到目标区域中。我们利用这个函数，就能够实现截取最佳区域以显示在缩略图中的效果。

在绘制缩略图我们还实现了一个小技巧：缩略图大小是固定的，但之间的间距是动态调整的，当缩略图之间的间距小于一个阀值的时候，我们强制最小间隔不小于阀值，详细代码请看清单 17。

响应点击事件

显示缩略图以后，我们需要响应点击事件，即能够点击缩略图，显示所对应的图片。同时，我们还需要点击左右两边的按钮，能够实现缩略图的翻页。这是通过清单 20 所示的代码实现的：

清单 20．响应鼠标点击事件

```
    // 加入了鼠标点击事件的响应
    this.load = function() {
        //event binding
        canvas.onclick = onMouseClick;
        canvas.onmousemove = onMouseMove;
}
// 鼠标点击事件处理
    function onMouseClick(event) {
        point = {x: event.clientX,
y:event.clientY};
        lastMousePos = point;

        if (pointIsInRect(point, lButtonRect))
{
            nextPane(true);
        } else if (pointIsInRect(point,
rButtonRect)) {
            nextPane(false);
        } else {
            var selectedIndex =
findSelectImageIndex(point);
            if (selectedIndex != -1) {
                selectImage(selectedIndex);
            }
        }
        updateIdleTime();
    }
    // 返回所点击的缩略图序号，如果没有点击缩略图则返回
-1
    function findSelectImageIndex(point) {
        for(var i = 0; i < imageRects.length;
i++) {
            if (pointIsInRect(point,
```

```
imageRects[i].rect))
                return i +
firstImageIndex;
}
        return -1;
    }
    // 将当前图片序号设为 index，重画
    function selectImage(index) {
        currentImage = index;
        paint();
    }
    // 将缩略图翻页，更新缩略图中第一张图片的序号
    function nextPane(previous) {
        if (previous) {
            firstImageIndex = firstImageIndex -
maxThumbNailCount < 0?
0 : firstImageIndex - maxThumbNailCount;
        } else {
            firstImageIndex = firstImageIndex +
maxThumbNailCount*2 - 1 > imageCount - 1?
(imageCount - maxThumbNailCount > 0?
imageCount - maxThumbNailCount: 0) :
firstImageIndex + maxThumbNailCount;

        }
        currentImage = (firstImageIndex <=
currentImage &&
currentImage <= firstImageIndex +
maxThumbNailCount)? currentImage :
firstImageIndex;
        paint();
    }
```

这里我们通过 2 个变量 firstImageIndex 和 currentImage 来控制缩略图和当前图片的显示，并能够根据鼠标点击来改变当前选中的图片。

加入其他效果

根据当前窗口大小调整 Canvas 大小

当浏览器的大小改变的时候，我们的图片浏览器就会由于没能重画导致部分区域无法显示。我们需要根据浏览器当前页面大小来动态定义整个图片浏览器的大小，从而能够调整整个图片浏览器的最佳大小。代码如清单 21 所示：

清单 21 .Resize 支持

```
    this.load = function() {
        //resize
        resize();
```

```
        window.onresize = resize;

        //event binding
        canvas.onclick = onMouseClick;
        canvas.onmousemove = onMouseMove;

        loadImages();

        startLoop();
        updateIdleTime();
    }
    function resize() {
        var size = getScreenSize();
        canvas.width = size.width;
        canvas.height = size.height;
        paint();
    }

    function getScreenSize() {
        return { width:
document.documentElement.clientWidth,
height:
document.documentElement.clientHeight};
    }
```

这里代码响应了 window 对象的 onresize 事件，从而能够响应整个浏览器页面大小改变的事件，通过 document.documentElement.clientWidth 和 document.documentElement.clientHeight 这两个 DOM 属性，我们获得了当前页面显示范围大小，从而能够动态调整 Canvas 的大小到最佳位置。

显示缩略图预览

我们还需要实现这种效果：当鼠标放置在某个缩略图上方时，能够显示一个缩略图预览界面，其效果如图 7 所示：

图 7 . 缩略图预览

实现代码如清单 22 所示：

清单 22 . 缩略图预览代码

```
    const ARROW_HEIGHT = 10;  // 下方三角形的高度
    const BORDER_WRAPPER = 2;   // 边缘白框的厚度
    // 绘制预览图
    function paintHighLightImage(srcRect, imageRe
      var ratio = imageRect.image.width == srcRec

      THUMBNAIL_LENGTH/imageRect.image.width:THUM
      ratio *= 1.5;

        var destRect = {
            x:imageRect.rect.x + imageRect.rect.w
imageRect.image.width*ratio/2,
            y:navRect.y - ARROW_HEIGHT - BORDER_W
imageRect.image.height*ratio,
            width: imageRect.image.width * ratio,
            height: imageRect.image.height * rati
        }

        var wrapperRect = {
            x: destRect.x - BORDER_WRAPPER,
            y: destRect.y - BORDER_WRAPPER,
            width: destRect.width + BORDER_WRAPPE
            height: destRect.height + BORDER_WRAP
        }

        var arrowWidth = ARROW_HEIGHT * Math.tan(

        context.save();
        context.fillStyle = 'white';
        context.translate(wrapperRect.x, wrapperR
        context.beginPath();
        context.moveTo(0, 0);
        context.lineTo(wrapperRect.width, 0);
        context.lineTo(wrapperRect.width, wrapper
        context.lineTo(wrapperRect.width/2 + arro
        context.lineTo(wrapperRect.width/2, wrapp
        context.lineTo(wrapperRect.width/2 - arro
        context.lineTo(0, wrapperRect.height);
        context.lineTo(0, 0);
        context.closePath();
        context.fill();
        context.drawImage(imageRect.image, BORDER
destRect.width, destRect.height);
        context.restore();
    }
```

在函数 paintHighLightImage 中大量使用了 Canvas 的路径绘图函数来绘制这个底部为三角形箭头，上部为矩形的形状。感兴趣的读者可以研究这些 Canvas 绘图函数的使用。

自动隐藏

最后我们在加入一个动态的效果：当鼠标不再移动超过一定时刻的时候，导航栏能够自动隐藏。其代码如清单 23 所示：

清单 23 . 自动隐藏代码

```
// 加入了自动隐藏导航栏的功能
   function paint() {
       context.clearRect(0, 0, canvas.width,
canvas.height);
       paintImage(currentImage);
       var paintInfo = {inLeftBtn:false,
inRightBtn:false, inThumbIndex: null}

       if (lastMousePos && navRect &&
lButtonRect && rButtonRect) {
           if (pointIsInRect(lastMousePos,
navRect)) {
               paintInfo.inLeftBtn =
pointIsInRect(lastMousePos, lButtonRect);
               paintInfo.inRightBtn =
pointIsInRect(lastMousePos, rButtonRect);
               if (!paintInfo.inLeftBtn &&
!paintInfo.inRightBtn) {
                   var index =
findSelectImageIndex(lastMousePos);
                   if (index != -1) {
                       paintInfo.inThumbIndex
= index;
                   }
               }
           }
       }
       if(idleTime && getTime() - idleTime <=
IDLE_TIME_OUT) {
           paintNavigator(paintInfo);
       }
   }
```

当空闲时间超过阀值时，导航栏能够自动隐藏，这样浏览图片更加方便。

最终效果

在合并了上述所有清单代码之后，我们在浏览器上就可以看到如图 8 所示的效果。

图 8 . 完整的图片浏览器效果

完整的代码请看附件。运行代码需要 Firefox 1.5, Chrome 1, Safari 3 以上版本的浏览器。

总结及展望

本文用图片浏览器的例子来说明 Canvas 的各种函数的使用。该例子也只是一个简单的 demo，并未涉及更为高级的 Canvas 使用，例如旋转坐标转换，绘制曲线，组合图形，渐变色彩等等。该例子也可以进一步改进，加入更多动态效果并提高效率。本文就不一一叙述。

从上述例子我们也能看到，Canvas 作为 HTML 5 新的元素，其绘图功能已经很接近操作系统的渲染函数。Canvas 元素可以进行矢量绘图也可以进行位图的绘制，在不久的将来，Canvas 还能利用 WebGL 技术支持 3D 绘图，这为未来的的网页游戏制作和更为丰富的 Web 用户体验提供了便利。在最新的 Google Wave 平台中，就已经大量使用了 Canvas 技术来渲染用户界面。我相信，在不久的将来，Canvas 能够大量被广大网页设计师和架构师所使用，并进一步被得到完善和加强。

下载资源

- 本文相关源代码 (samplecode.zip | 2176)

相关主题

- 参看系列文章“Canvas Tutorial”，了解更多 Canvas 绘图细节。
- 如果对 Canvas 标准感兴趣可以参看完整的 Canvas API 标准。
- 关于 Cufon 的完整 API 参考，读者可见 http://wiki.github.com/sorccu/cufon/api。
- MDC网站上有丰富的Canvas资料。
- 本例中的UI设计是部分参考了Flickr的界面，但它是用Flash实现的，而我们用纯HTML实现的。
- developerWorks上有一系列文章关于HTML的未来:《HTML 的未来》
- developerWorks 技术活动和网络广播：随时关注 developerWorks 技术活动和网络广播。
- developerWorks Web development 专区：通过专门关于 Web 技术的文章和教程，扩展您在网站开发方面的技能。

评论

☐ 有新评论时提醒我

添加或订阅评论，请先登录或注册。

最新的 历史 热门

L.EC 2017 年 07 月 25 日

麻烦您可以发一下这个附件 真的下载不了 735366835@qq.com

yikelibm 2011 年 11 月 21 日

为什么下载不了，出现404错误，谁有的发给我啊，402486893@qq.com，急需啊