第二章

在Canvas上繪圖

想要有效的使用HTML5 Canvas做二維空間的繪圖，上色和形狀的變化需要有強大的功能。而其內建的形狀是相當有限的，但透過*paths*即可使用一系列的線段來畫出我們想要的形狀，這將在本書第33頁的"使用Paths來畫線"此章節做討論。

很多線上的網站都有討論到HTML5 Canvas API。W3C網站對Canvas 2D Drawing API也有詳細的說明，而且會隨時修正有關它的內容與功能；您可進入以下的網站參考：*http://dev.w3.org/html5/canvas-api/canvas-2d-api.html*。

很可惜的是，這個線上的參考網站並沒有提供使用這個API的範列說明。我們並不是將所有的功能簡單的列印出來就好，而是花了很多的時間來建立範例，並對我們有提到的功能做深入的探討與說明。

本章的基本檔案設定

既然我們要透過Drawing API來做繪圖的處理，在本章節裡的所有範例程式就必須使用相同的檔案設定，設定的方式如下。我們所建立的範例都將以此為基礎，你只能對`drawScreen()`函數中的內容做更改而已：

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Ch2BaseFile - Template For Chapter 2 Examples</title>
<script src="modernizr-1.6.min.js"></script>
<script type="text/javascript">
window.addEventListener('load', eventWindowLoaded, false);
function eventWindowLoaded() {

   canvasApp();
}

function canvasSupport () {
     return Modernizr.canvas;
}

function canvasApp(){

if (!canvasSupport()) {
        return;
    }else{
     var theCanvas = document.getElementById("canvas");
     var context = theCanvas.getContext("2d");
   }

  drawScreen();

  function drawScreen() {
     //在這裡做改變
     context.fillStyle = '#aaaaaa';
     context.fillRect(0, 0, 200, 200);
     context.fillStyle  = '#000000';
     context.font = '20px _sans';
     context.textBaseline = 'top';
     context.fillText  ("Canvas!", 0, 0);

  }
}

</script>
</head>
<body>
<div style="position: absolute; top: 50px; left: 50px;">
<canvas id="canvas" width="500" height="500">
 Your browser does not support HTML5 Canvas.
</canvas>
</div>
</body>
</html>
```

基本的長方形

讓我們來繼續看在Canvas上繪製最原始、內建的幾何圖形－長方形。在Canvas裡，有三種不同的方式可以畫出長方形：分別是以填上色彩的方式、用筆畫或者是用清除的方式。其實我們也可以使用paths來建立長方形（或其他的形狀），這些將在下一章節做說明。

首先，讓我們先來看這三種動作的API函數：

`fillRect（x,y,width,height）`

以x,y位置為起點，繪出指定的寬和長且填上色彩的長方形。

`strokeRect（x,y,width,height）`

從x,y位置開始畫出指定寬和長度的一個長方形輪廓。這就是使用目前`strokeStyle`（筆的樣式）、`lineWidth`（線寬）、`lineJoin`（兩條連續直線末端的接合樣式），以及`miterlimit`（兩個線段所建立之角的大小）的設定。

`clearRect（x,y,width,height）`

從x,y位置開始清除指定的寬與長之特定區域並使其全為透明（使用透明的黑色）。

在要使用這三個函數中的任何一個之前，我們需要設定等一下要繪圖時所使用的填色（fill）或筆（stroke）的樣式。

要設定這些樣式最基本的方式就是將顏色的值使用24位元十六進位的方式來表示。這是我們第一個要展示的範例：

```
context.fillStyle = '#000000';
context.strokeStyle = '#ff00ff';
```

在範例2-1裡，將黑色設定為要填滿的顏色，將紫色設定為筆的顏色；結果如圖2-1。

範例2-1 基本的長方形

```
function drawScreen() {
     context.fillStyle  = '#000000';
     context.strokeStyle = '#ff00ff';
     context.lineWidth  = 2;
     context.fillRect(10,10,40,40);
     context.strokeRect(0, 0,60,60);
     context.clearRect(20,20,20,20);

}
```

圖2-1 基本的長方形

Canvas的狀態

當我們在Canvas context上繪圖時，我們可以使用所謂的繪圖狀態（drawing states）的堆疊：這些狀態儲存的資料是關於Canvas context在任何一個時間裡的狀態。這裡有每一個儲存在堆疊裡的狀態資料列表：

- 轉換矩陣的資訊，例如：使用`context.rotate()`及`context.setTransform()`方法來做轉動（rotations）或移動（translations）
- 目前的剪裁區域
- canvas目前的屬性值，例如（並非全部）：
  - globalAlpha
  - globalCompositeOperation
  - strokeStyle
  - textAlign, textBaseline
  - lineCap, lineJoin, lineWidth, miterLimit
  - fillStyle
  - font
  - shadowBlur, shadowColor, shadowOffsetX, and shadowOffsetY

稍後我們就會討論到這些狀態了。

不屬於狀態的部份有哪些？

目前在Canvas context上操作的path（稍後會討論到）及圖像（請看第四章）並不屬於儲存狀態裡的一部份。這個重要的功能可讓我們在canvas上繪圖，以及將各別的物件動起來。在第43頁的"將Canvas做簡單的轉換"章節裡使用了Canvas state，對目前正被建立與繪圖的形狀做改變，至於其他的canvas則不會更動到。

如何儲存與還原Canvas State？

儲存目前堆疊的狀態：

```
context.save()
```

使用"popping"還原回canvas儲存在堆疊中的最後一個狀態：

```
context.restore()
```

使用Paths來畫線

我們可以在canvas上使用Paths method繪製任意的形狀：path是由一連串的點所構成的，而這些點之間就可以畫成線。一個Canvas context只會有一個"current" path，當`context.save()` method被呼叫時，它並沒有被儲存在目前的繪圖狀態裡。

paths的context是一個非常重要的觀念，因為它可以讓你只改變目前在canvas上的path。

Path的開始與結束

path的開始函數為`beginPath()`，結束函數為`closePath()`。當你將path裡的兩個點連接起來時，它就是*subpath*。如果最後一個點和第一個點連接起來時，就是所謂的"closed"。

目前的轉換陣列將會影響到在此path上繪製的每一樣東西，我們將會在下面的章節中討論到轉換（transformation），如果我們不想將任何的改變反應至path上，就要將轉換矩陣（transformation matrix）都設為一樣（或重新設定）。

真正的開始繪圖囉

最簡單的path就是用一連串的`moveTo()`與`lineTo()`指令來完成，如下方的範例2-2：

範例2-2 一個簡單的線條

```
function drawScreen() {
   context.strokeStyle  = "black"; //需為列表中可用的顏色
   context.lineWidth  = 10;
   context.lineCap  = 'square';
   context.beginPath();
   context.moveTo(20, 0);
   context.lineTo(100, 0);
   context.stroke();
   context.closePath();

}
```

圖2-2顯示了此範例的輸出結果。

圖2-2 一個簡單的線條

範例2-2從20,0的位置開始至100,0的位置，簡單的繪出了10個像素寬的水平線（或stroke）。

我們也加入了lineCap和strokeStyle屬性，在我們要進入到更進階的繪圖功能之前，來簡單的看一下用在線條上的各個屬性。

lineCap屬性

context.lineCap。lineCap是指結束線條的可用端點樣式。它有下列三種值：

`butt`

預設值：扁平的線條端點。

`round`

圓形的線條端點。

`square`

方形的線條端點。

lineJoin屬性

context.lineJoin。lineJoin是指兩條線的末端接合角，這就稱為*join*。而三角形就是由這種接合角所構成的，只需設定基本的lineJoin屬性即可。

`miter`

預設值：斜接接合角；miterLimit是允許最大的斜接線寬度（預設值為10）。

`bevel`

斜面接合角。

`round`

圓形接合角。

`lineWidth`

lineWidth（預設值＝1.0）線的厚度。

strokeStyle

strokeStyle定義了線條或形狀要使用的顏色或樣式（如同在範例2-2簡單的長方形中所見到的）。

繪製更進階的線條範例

範例2-3將會顯示這三個屬性的動作：結果將繪製出如圖2-3的樣子。在canvas上畫線時你會發現有些奇怪的現像，這些我們都將一一說明。

範例2-3 Line Cap和join

```
function drawScreen() {

      // Sample 1: round end, bevel join, at top left of canvas
      context.strokeStyle  = "black"; //需為列表中可用的顏色
      context.lineWidth  = 10;
      context.lineJoin  = 'bevel';
      context.lineCap  = 'round';
      context.beginPath();
      context.moveTo(0, 0);
      context.lineTo(25, 0);
      context.lineTo(25,25);
      context.stroke();
      context.closePath();

      // Sample 2: round end, bevel join, not at top or left of canvas
      context.beginPath();
      context.moveTo(10, 50);
      context.lineTo(35, 50);
      context.lineTo(35,75);
      context.stroke();
      context.closePath();

      // Sample 3: flat end, round join, not at top or left of canvas
      context.lineJoin  = 'round';
      context.lineCap = 'butt';
      context.beginPath();
      context.moveTo(10, 100);
      context.lineTo(35, 100);
      context.lineTo(35,125);
      context.stroke();
      context.closePath();

   }
```

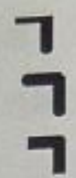


圖2-3 Line cap和join

這三個線與接角的範例足以說明在canvas上用這幾種屬性的組合所畫出的線條。

第一個範例試圖在canvas的左上角開始畫圖，結果圖像有點怪怪的；Canvas paths從開始位置像素的中心處同時向x與y方向向外繪製，就是因為這個原因，範例1中的頂線看起來比我們指定的10個像素的厚度還來得小；此外，水平線結束的"round"似乎也沒看見，因為這兩個都跑到螢幕坐標的"負值"區域；而且lineJoin的"bevel"角線根本就沒有畫出來。

第二個範例針對範例一的問題做了改進；這樣就畫得出完整厚度的水平線，以及圓形的線條端點和兩線交接的斜切面了。

第三個範例則是將lineCap改成預設的"butt"，並將lineJoin換成"round"（圓角弧度）。

進階的Path methods

接下來我們要看一些關於path更深入的methods，包括了弧形（arc）與曲線（curve），將此二種結合起來可以繪製更複雜的圖像。

弧形（Arcs）

在canvas上有4種函數可供我們來繪製弧形與曲線，光一個弧形就可以完成一個圓或圓形的任何一部份。

context.arc()

```
context.arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

x、y值是在定義圓心；radius是指我們所畫的圓之半徑；startAngle與endAngle是指弧度不是角度；anticlockwise則定義了弧形的方向，它的值有true或false。

舉例來說，如果我們要畫一個圓心在100,100半徑為20的圓，如圖2-4所示，我們就可以使用下方drawScreen()函數內的程式碼：

```
context.arc(100, 100, 20, (Math.PI/180)*0, (Math.PI/180)*360, false);
```

範例2-4說明了繪製一個圓的程式碼。

範例2-4 圓弧

```
function drawScreen() {

      context.beginPath();
      context.strokeStyle = "black";
      context.lineWidth = 5;
      context.arc(100, 100, 20, (Math.PI/180)*0, (Math.PI/180)*360, false);

      //完整的圓
      context.stroke();
      context.closePath();

}
```

圖2-4 一個基本的圓弧

要注意的是我們必須將起始角度（0）及終止角度（360）分別乘上（Math.PI/180）以轉換成弧度；藉由使用起始角度0終止角度360我們就可以畫一個圓了。

我們也可以將起始角度與終止角度不設成0與360這樣可畫出圓的一部份。程式碼中的drawScreen()就可以畫出一個順時針的方向的1/4圓，如圖2-5所示：

```
context.arc(100, 200, 20, (Math.PI/180)*0, (Math.PI/180)*90, false);
```

圖2-5 一個1/4的圓弧

如果我們要畫一個0-90度角以外的圓，如圖2-6，可以將anticlockwise參數設定為true：

```
context.arc(100, 200, 20, (Math.PI/180)*0, (Math.PI/180)*90, true);
```

圖2-6 一個3/4的圓弧

context.artTo()

```
context.arcTo(x1, y1, x2, y2, radius)
```

arcTo method只在最新的瀏覽器上才有支援的功能－或許是因為它的功能arc()函數就可做到了吧。它會從目前path的位置畫一條直線至座標點（x1,y1）之處，然後從該點繼續以radius給的半徑畫弧至座標點（x2,y2）之處。

只有在current path至少還有一個subpath的情況下，context.arcTo() method才可運作。所以，一開始我們要先從（0,0）的位置至（100,200）的位置畫一條線，接下來才可以畫一個小圓弧，看起來有點像彎曲的衣架（找不到更適合的形容了），如圖2-7：

```
context.moveTo(0,0);
context.lineTo(100, 200);
context.arcTo(350,350,100,100,20);
```

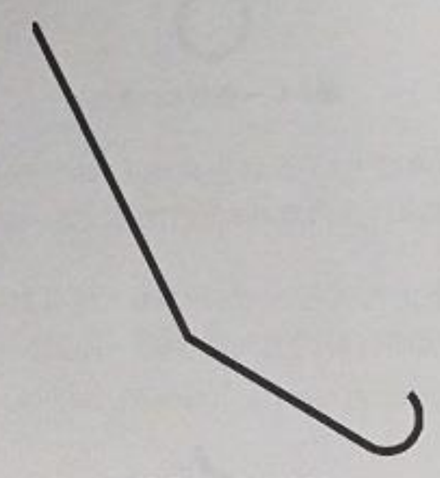


圖2-7 arcTo()範例

貝茲曲線（Bezier Curves）

貝茲曲線在三次方與二次方程式上比起arcs來得更有彈性，

- context.bezierCurveTo(*cp1x, cp1y, cp2x, cp2y, x, y*)
- context.quadraticCurveTo(*cpx, cpy, x, y*)

貝茲曲線是透過一個"起始點"和"終止點"來為2D的空間做定義，再加上一或二個"control"點，用以定義如何在canvas上建構出曲線。一般的三次方程式貝茲曲線使用2個點，而二次方程式則使用1個點。

二次方程式的版本如圖2-8所示，這是最簡單的，只需要一個終止點（最後一個點），以及在空間中的單一一點也就是控制點（control point）（第一個點）。

```
context.moveTo(0,0);
context.quadraticCurveTo(100,25,0,50);
```

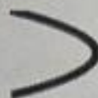


圖2-8 一個簡單的二次方程式貝茲曲線

這條曲線從0,0開始至0,50的位置做結束，並在100,25這一點開始建立弧形，這個點大概就在此弧形的垂直中心之處。此控制點100的值是用來拉出弧形，使其成一個被拉長的曲線。

而三次方程式的貝茲曲線提供了更多的選擇，因為有兩個控制點可使用。如圖2-9所顯示的"S"曲線就是一種非常容易製作出的曲線：

```
context.moveTo(150,0);
context.bezierCurveTo(0,125,300,175,150,300);
```

圖2-9 兩個控制點的貝茲曲線

Canvas剪裁區域

將save()與restore()函數和Canvas裁剪區域結合在一起就可以限制path與subpath的繪圖範圍。要這麼做，首先要設定rect()為我們想要繪圖區域的長方形，然後再呼叫clip()函數。clip()函數設定了要剪裁的方形區域，此方形區域可用rect() method定義；現在，無論我們在目前的context下怎麼樣的畫，它只會顯示此區域內部份的圖樣而已。在繪圖的過程中你可以把它想像成遮罩（mask）。範例2-5顯示了它的運作方式，至於產生的結果正如圖2-10所示。

範例2-5 Canvas的裁剪區域

```
function drawScreen() {

    //在螢幕上繪製一個大框框
    context.fillStyle = "black";
    context.fillRect(10, 10, 200, 200);
    context.save();
    context.beginPath();
    //將canvas從0,0處開始裁剪成50x50的大小
    context.rect(0, 0, 50, 50);
    context.clip();

    //紅色的圓
    context.beginPath();
    context.strokeStyle = "red"; //需在有效的顏色列表中
    context.lineWidth = 5;
    context.arc(100, 100, 100, (Math.PI/180)*0, (Math.PI/180)*360, false);
    //完整的圓
    context.stroke();
    context.closePath();

    context.restore();

    //重新裁剪成完整的canvas
    context.beginPath();
    context.rect(0, 0, 500, 500);
    context.clip();

    //繪製一條不可被裁剪的藍色線條
    context.beginPath();
    context.strokeStyle = "blue"; //需在有效的顏色列表中
    context.lineWidth = 5;
    context.arc(100, 100, 50, (Math.PI/180)*0, (Math.PI/180)*360, false);
    //完整的圓
    context.stroke();
    context.closePath();

}
```

圖2-10 Canvas的裁剪區域

範例2-5在Canvas上先畫了一個200x200的黑色方形，接下來設定了Canvas的裁剪區域為rect（0,0,50,50），clip()就會依據以上所規定的樣子來剪裁。當我們在畫紅色的圓形時，在此方形內我們只看到了一部份的圓；最後，我們又將裁剪的區域改成rect（0,0,500,500）並畫上一個藍色的圓，此時，完整的圓形都可以在canvas上看到了。

其他的Canvas methods都可以和剪裁區域一起使用，最明顯的是arc()函數：

arc(float x, float y, float radius, float startAngle, *float* endAngle, *boolean* anticlockwise)

可以用它來建立一個圓形的剪裁區域以取代矩形。

在Canvas上做組合

組合是指我們能對在canvas上所繪製的物件做多精細的控制其透明度與分層，有兩個屬性可供我們控制Canvas的組合作業：globalAlpha與globalCompositeOperation。

globalAlpha

globalAlpha的預設值為1.0（完全不透明），它的值可以從0.0到1.0；此屬性必須在畫出形狀之前就先設定好。

globalCompositeOperation

globalCompositeOperation是在控制形狀如何畫到目前的Canvas圖像中，由其是在globalAlpha的值有任何的變更及圖像有任何的改變之後（下一章節第43頁"將Canvas做簡單的轉換"，將會有更多的說明）。

在下方的列表中，"source"是我們要在canvas上畫的形狀（原圖形），"destination"則是指目前在canvas上顯示的圖像（新圖形）。

copy

只保留新圖形，其餘都不要。

destination-atop

原圖形與新圖形重疊的部份予以保留，並顯示完整的新圖形，其餘的部份則是透明。

destination-in

原有圖形與新圖形重疊之處才會畫出來，其餘則是透明。

destination-out

原有圖形與新圖形不重疊部份才會畫出來，其餘則是透明。

destination-over

會在原圖形之後方繪製新圖形，也就是原圖形會蓋住新圖形。

lighter

原圖形與新圖形重疊的部份會做加色（打光）處理，1.0是其值的極限。

source-atop

新圖形與原圖形重疊之處會被畫出來，並且蓋在原圖形上面，而新圖形沒有重疊部份不會畫出。

source-in

新圖形僅出現與原圖形重疊的部份，其餘都透明。

source-out

顯示新圖形，但只有新圖形與原圖形不重疊的部份才會被畫出來。

source-over

（預設值），在原有圖形上畫上新的圖像，也就是新圖形會覆蓋在原有的圖形上面。

xor

重疊的部份變透明。

範例2-6在說明這些值如何影響著在canvas上所畫出的圖形，結果如圖2-11所示：

範例2-6 Canvas組合範例

```
function drawScreen() {

      //在螢幕上繪製一個大框框
      context.fillStyle = "black"; //
      context.fillRect(10, 10, 200, 200);

      //讓globalCompositeOperation保持現狀
      //繪製一個紅色的正方形
      context.fillStyle = "red";
      context.fillRect(1, 1, 50, 50);

      //將globalCompositeOperation設定為source-over
```

```
    context.globalCompositeOperation = "source-over";
    //旁邊再畫一個紅色的正方形
    context.fillRect(60, 1, 50, 50);
    //將其設定為destination-atop
    context.globalCompositeOperation = "destination-atop";
    context.fillRect(1, 60, 50, 50);

    //設定globalAlpha
    context.globalAlpha = .5;

    //將其設定為source-atop
    context.globalCompositeOperation = "source-atop";
    context.fillRect(60, 60, 50, 50);

}
```

圖2-11 Canvas組合範例

在這個範例裡，你可以看到我們在大玩Canvas的globalCompositeOperation與globalAlpha這兩個屬性。當我們將值設定為"source-over"時，基本上就是將globalCompositeOperation的值重設回預設值；然後我們產生了一些紅色的正方形，以顯示各種不同方式的組合。使用destination-atop要注意一下，它會將新的圖形放到目前Canvas圖像的下方；而globalAlpha屬性只會對在設定好這個值之後所畫的圖才會有所影響。這意味著，下一個要畫的圖形其透明度我們跟本就不需要用到save()與restore()將Canvas狀態給儲存起來。

在下一章節裡，我們將要看看會影響整個canvas的一些轉換（transformation）：如果我們只想改變最新的圖形，就必須使用save()與restore()函數。

將Canvas做簡單的轉換

在canvas上轉換，是指所繪製形狀其物理屬性的數學調整，最常使用的兩個形狀變換是縮放（scale）與旋轉（rotate），這也就是我們這一章所要講的重點。

運用在所有轉換（transformations）背後的原理就是一個數學矩陣的運作；很幸運地，在使用簡單的Canvas轉換時，你並不需要去瞭解它；而這裡我們所要討論的是如何透過改變Canvas的屬性來做到rotation、translation及scale的轉換。

旋轉（rotation）與移動（translate）轉換

一個面向左邊的canvas物件，可以說它的旋轉角度為0（如果一個物件有面向的問題，這個就很重要；否則我們會以此為指標）。因此，如果我們畫一個正方形（四邊都等長），除了一個面向左邊的平面之外，它一開始並沒有初始的面向。我們來畫一個正方形來參考一下：

```
//繪製一個紅色的正方形
context.fillStyle = "red";
context.fillRect(100,100,50,50);
```

現在，若想要將canvas旋轉45度角，我們需要做兩次簡單的步驟：第一，將目前的Canvas transformation一直設定為 "identity"（或是"reset"）矩陣：

```
context.setTransform(1,0,0,1,0,0);
```

因為Canvas使用弧度來做轉換，而不是角度，所以需要將45度角轉換成弧度：

```
var angleInRadians = 45 * Math.PI / 180;
context.rotate(angleInRadians);
```

第一課：要先呼叫setTransform()或其他的轉換函數，轉換才會對所繪製的形狀與線條生效。

如果你完全照著這些程式碼來操作，將會產生一個有趣的結果...就是"什麼都沒有"！這是因為setTransform()函數必須先做設定，這樣在Canvas上繪製形狀才會有效。首先，先畫一個正方形，然後再設定其轉換的屬性；這麼做對剛剛畫好的正方形而言並不會有任何的改變。範例2-7是個正確的程式碼，它產生了如我們所預期的結果，圖解說明如圖2-12。

範例2-7 簡單的旋轉變換

```
function drawScreen() {

    //繪製一個紅色的正方形
    context.setTransform(1,0,0,1,0,0);
    var angleInRadians = 45 * Math.PI / 180;
    context.rotate(angleInRadians);
    context.fillStyle = "red";
    context.fillRect(100,100,50,50);

  }
```

圖2-12 簡單的旋轉變換

這次我們得到了一個結果，但它可能和你預期的有所不同。這紅色的方形是被旋轉了，但看起來canvas也一起被旋轉了。整個canvas其實是不該被旋轉的，應該是在`context.rotate()`函數被呼叫後，只有繪圖的部份被旋轉而已。所以，為什麼我們的正方形不但旋轉而且還從螢幕的左邊移了出去呢？旋轉一開始是設成"nontranslated" 0，造成正方形從整個canvas的最左上角0 的位置處開始旋轉。

範例2-8提供了一個稍微不同的腳本：首先，先畫一個黑色的方形，然後為其設定旋轉的轉換，最後再畫一個紅色的方形，結果請看圖2-13。

範例2-8 旋轉與Canvas的狀態

```
function drawScreen() {

     //繪製一個黑色的正方形
     context.fillStyle = "black";
     context.fillRect(20,20,25,25);

     //繪製一個紅色的正方形
     context.setTransform(1,0,0,1,0,0);
     var angleInRadians = 45 * Math.PI / 180;
     context.rotate(angleInRadians);
     context.fillStyle = "red";
     context.fillRect(100,100,50,50);

  }
```

圖2-13 旋轉與Canvas的狀態

小黑正方形並沒有被旋轉的函數所影響，所以在這裡我們就可以知道，只有在context.rotate()函數被呼叫後所畫的圖才會被影響到。

同樣的，紅色的方形又再次的從左邊移開了。再次的說明一下，會發生這樣的情形是因為canvas並不知道在做旋轉時要以什麼當做依據。在沒有真正的旋轉依據時，位置0,0的這個點就會被設定為其旋轉的依據點：context.rotate()函數會在位置0,0的四週做旋轉，所產生的結果就是我們準備在下一個課要說的。

第二課：我們必須將原始點（point of origin）"移動"到我們要繪製形狀的中央，這樣圖形在旋轉時就會以自己的中心點為依據開始旋轉。

我們來改變一下範例2-8，將紅色的方形在目前的位置上旋轉45度角。

首先，在呼叫fillRect()函數時，將要傳入的參數設成幾個變數並指定它們的值，如此我們便可以控制紅色方形的屬性；雖然這不是必要的，但會讓程式碼更容易閱讀與修改：

```
var x = 100;
var y = 100;
var width = 50;
var height = 50;
```

接下來，呼叫context.translate()函數，我們必須將canvas的原始依據點改為我們要旋轉的紅色方形的中心。這個函數會將canvas的依據點移到我們所接受的x與y位置。而現在紅色方形的中心就是我們的物件左上角x位置（100）加上紅色方形寬度的一半。使用我們為紅色方形所建立的變數，就會看起來像這樣：

```
x+0.5*width
```

再來，我們還需找出移動時的依據點y的位置，這次我們使用的是圖形左上角y的值，以及方形的高度：

```
y+.05*height
```

translate()函數看起來會像這樣：

```
context.translate(x+.05*width, y+.05*height)
```

現在，我們已經將canvas移動到正確的點了，可以準備旋轉囉。這部份的程式碼並不需要改變：

```
context.rotate(angleInRadians);
```

最後，要開始繪圖囉！我們不能直接使用範例2-8裡的值，因為canvas的依據點已經被我們移到將要畫出紅色方形的中心點位置了。你可以將125,125看成開始繪圖的起始點。之所

以x為125是因為方形的左上角為100，再加上寬度的一半（25）；y的值為125也是同樣的道理；translate() method就是在做這件事。

我們需要在正確的x與y座標上開始繪圖；我們還要將繪圖物件的x座標減去該物件寬度的一半，y座標減去該物件高度的一半：

```
context.fillRect(-0.5*width,-0.5*height, width, height);
```

為什麼要這樣做呢？圖2-14說明了這樣的情況：

想一想，我們要在左上角開始畫一個方形時，如果我們的原始點在125,125，實際上左上角的位置是100,100；但，如果我們將canvas的原始點從125,125換成了0,0，在沒有移動canvas的情況下開始畫方形，其實是從canvas -25,-25的位置開始畫。

這樣就會迫使我們在0,0的座標上畫出我們的方形，而不是125,125的座標。所以，當我們真正在畫方形時，就是用如圖2-15所示的這個座標為依據。

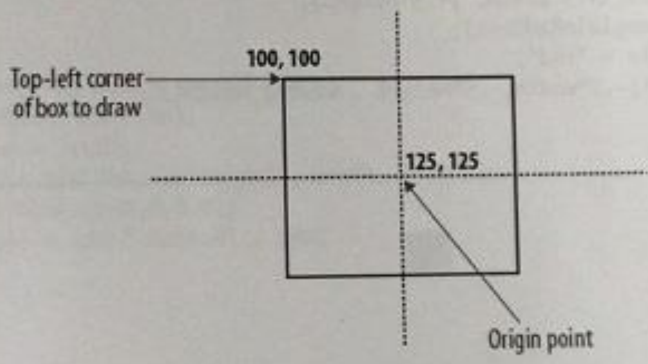


圖2-14 新的移動點

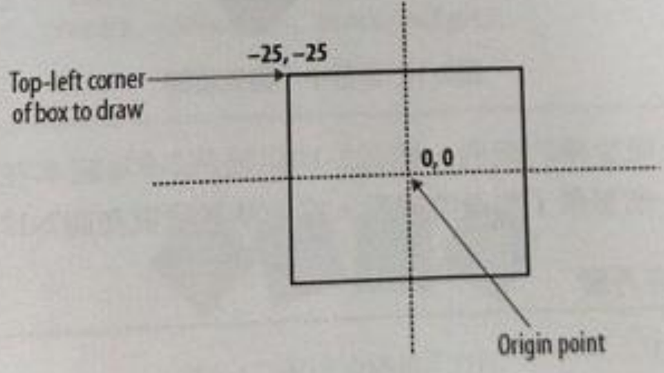


圖2-15 使用移動點來畫圖

總而言之，要將原始依據點改成我們要畫的圖形的中心點，這樣才可依著該點旋轉。而在畫方形時，我們要讓程式碼的行為看起來像（125,125）那樣，實際上卻是（0,0）。如果我們沒有移動原始依據點，依舊可以將（125,125）做為方形的中心點（如圖2-14）。範例2-9說明它將如何運作，而其結果將如圖2-16所示。

範例2-9 依著中心點做旋轉

```
function drawScreen() {

      //繪製一個黑色正方形
      context.fillStyle = "black";
      context.fillRect(20,20 ,25,25);

      //繪製一個紅色正方形
      context.setTransform(1,0,0,1,0,0);
      var angleInRadians = 45 * Math.PI / 180;
      var x = 100;
      var y = 100;
      var width = 50;
      var height = 50;
      context.translate(x+.5*width, y+.5*height);
      context.rotate(angleInRadians);
      context.fillStyle = "red";
      context.fillRect(-.5*width,-.5*height , width, height);

   }
```

圖2-16 依著中心點做旋轉

讓我們再來看最後一個旋轉的範例，範例2-10以範例2-9為範本在canvas上又增加了4個40*40的正方形，每一個都做了些微的轉動。它的結果呈現在圖2-17中。

範例2-10 多個旋轉的正方形

```
function drawScreen() {

      //繪製一個紅色正方形
      context.setTransform(1,0,0,1,0,0);
      var angleInRadians = 45 * Math.PI / 180;
```

```
    var x = 50;
    var y = 100;
    var width = 40;
    var height = 40;
    context.translate(x+.5*width, y+.5*height);
    context.rotate(angleInRadians);
    context.fillStyle = "red";
    context.fillRect(-.5*width,-.5*height , width, height);
    context.setTransform(1,0,0,1,0,0);
    var angleInRadians = 75 * Math.PI / 180;
    var x = 100;
    var y = 100;
    var width = 40;
    var height = 40;
    context.translate(x+.5*width, y+.5*height);
    context.rotate(angleInRadians);
    context.fillStyle = "red";
    context.fillRect(-.5*width,-.5*height , width, height);

    context.setTransform(1,0,0,1,0,0);
    var angleInRadians = 90 * Math.PI / 180;
    var x = 150;
    var y = 100;
    var width = 40;
    var height = 40;
    context.translate(x+.5*width, y+.5*height);
    context.rotate(angleInRadians);
    context.fillStyle = "red";
    context.fillRect(-.5*width,-.5*height , width, height);
    context.setTransform(1,0,0,1,0,0);
    var angleInRadians = 120 * Math.PI / 180;
    var x = 200;
    var y = 100;
    var width = 40;
    var height = 40;
    context.translate(x+.5*width, y+.5*height);
    context.rotate(angleInRadians);
    context.fillStyle = "red";
    context.fillRect(-.5*width,-.5*height , width, height);

}
```

圖2-17 多個旋轉的正方形

接下來，將要研究的是縮放的轉換。

縮放的轉換

context.scale()函數有2個參數：第一個是x軸的縮放屬性，第二個則是y軸的縮放屬性；它們的值為1時，就表示是該物件正常的大小，因此，如果想要將該物件放大兩倍時，就必須將x與y的值都設定為2。在drawScreen()中放入下方的程式碼就會產生如圖2-18的紅色正方形：

```
context.setTransform(1,0,0,1,0,0);
context.scale(2,2);
context.fillStyle = "red";
context.fillRect(100,100 ,50,50);
```

圖2-18 一個放大的正方形

如果將這幾行的程式碼做個測試，你將會發現它的運作方式和旋轉（rotation）有點類似。我們並沒有為放大了2倍的正方形移動它的縮放原始點，而是將canvas左上角做為原始點；結果就使得紅色方形離得更遠了。我們想要的是紅色方形保持在原來的地方，且以自己的中心點來做縮放。要這麼做，需在縮放比例之前，將原始點移至方形的中心，並以此中心來繪製方形（正如我們在範例2-9所做的那樣）。範例2-11產生的結果顯示在圖2-19中。

範例2-11 從中心點開始放大

```
function drawScreen() {

    //繪製一個紅色正方形
    context.setTransform(1,0,0,1,0,0);
    var x = 100;
    var y = 100;
    var width = 50;
    var height = 50;
    context.translate(x+.5*width, y+.5*height);
    context.scale(2,2);
    context.fillStyle = "red";
    context.fillRect(-.5*width,-.5*height , width, height);

}
```

圖2-19 從中心點開始放大

將縮放與旋轉轉換合併

如果我們想要對一個物件同時做縮放與旋轉，Canvas transformations可以很容易的將兩個結合，以達到所想要的結果（如圖2-20所示）。來看看範例2-12，依著前一個範例來更改，我們該如何使用scale(2,2)及rotate(angleInRadians)將其結合起來。

範例2-12 合併縮放與旋轉

```
function drawScreen() {
      context.setTransform(1,0,0,1,0,0);
      var angleInRadians = 45 * Math.PI / 180;
      var x = 100;
      var y = 100;
      var width = 50;
      var height = 50;
      context.translate(x+.5*width, y+.5*height);
      context.scale(2,2);
      context.rotate(angleInRadians);
      context.fillStyle = "red";
      context.fillRect(-.5*width,-.5*height , width, height);

   }
```

圖2-20 縮放與旋轉的合併

範例2-13也是一個將旋轉與縮放合併的例子，不過這次使用長方形。圖2-21就是它所呈現的結果。

範例2-13 縮放與旋轉一個長方形的物件

```
function drawScreen() {

     //繪製一個紅色長方形
     context.setTransform(1,0,0,1,0,0);
     var angleInRadians = 90 * Math.PI / 180;
     var x = 100;
     var y = 100;
     var width = 100;
     var height = 50;
     context.translate(x+.5*width, y+.5*height);
     context.rotate(angleInRadians);
     context.scale(2,2);
     context.fillStyle = "red";
     context.fillRect(-.5*width,-.5*height , width, height);

  }
```

圖2-21 長方形物件的旋轉與縮放

> **為每個形狀找中心點**
>
> 在canvas上所繪製的長方形或任何形狀的旋轉或縮放行為都和正方形很像。其實，只要我們很確定在做旋轉、縮放，或同時做旋轉與縮放之前移動原始點至圖形的中心點，我們就可以看到所期望的結果了。要記住，任何形狀"中心點"的x值是該形狀寬度的一半，y值是該形狀高度的一半。當我們試著要找這個中心點時，就需要使用邊框原理（*bounding box theory*）。
>
> 圖2-22說明了此原理。即使該形狀不是一個正方形，我們還是可以找出一個包括了該物件所有點的邊框。圖2-22是一個還像方形的方形，此原理也適用於長方形的邊框。

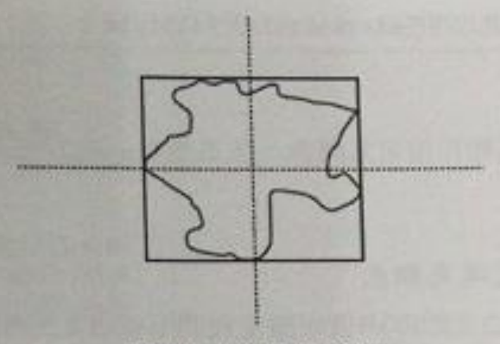


圖2-22 複雜圖形之邊界

漸層顏色之物件

本章我們透過基本與複雜的形狀結構很快的瞭解了顏色與填色的樣式，不過現在，我們要開始對它做更深入的討論了。除了這些簡單的填色之外，我們還可以使用不同的漸層樣式；此外，Canvas也提供了點陣圖像的填色method（見第四章）。

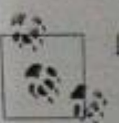
設定基本的填充顏色

Canvas的fillStyle屬性是用來設定canvas上的圖形其基本顏色。在稍早的章節裡已經使用過fillStyle來指定圖形的顏色了；範例是這樣子的：

```
context.fillStyle = "red";
```

以下是在HTML4規範裡所允許的字串值顏色；截至目前為止，HTML5對於顏色的規範並沒有新增其他的顏色，因此，這些顏色在HTML5裡一樣可以正常的運作：

```
黑色 = #000000
綠色 = #008000
銀色 = #C0C0C0
石灰色 = #00FF00
灰色 = #808080
黃褐色 = #808000
白色 = #FFFFFF
黃色 = #FFFF00
棕色 = #800000
深藍色 = #000080
紅色 = #FF0000
藍色 = #0000FF
紫色 = #800080
深青色 = #008080
紫紅色 = #FF00FF
淺綠色 = #00FFFF
```



以上所有的顏色都可用於strokeStyle及fillStyle。

當然，用顏色的名字來為物件指定其顏色，並不是唯一的方法；以下還有其他幾種可行的方案：

使用rgb() *method*來設定填充顏色

rgb() method使用了24位元的RGB值來指定我們所要填充的顏色：

```
context.fillStyle = rgb(255,0,0);
```

這個和前面使用紅色字串值來指定顏色的結果是一樣的。

使用十六進位字串來設定填充顏色

我們也可以使用十六進位的字串來設定fillStyle的顏色：

```
context.fillStyle = "#ff0000";
```

使用rgba() *method*來設定填充顏色

rgba() mtehod允許使用32位元來指定填充的顏色，而最後的8位元代表alpha值：

```
context.fillStyle = rgba(255,0,0,1);
```

alpha值從1（不透明）到0（透明）。

使用漸層色

在canvas上有兩種方法可以做出漸層的感覺：線性與放射狀（徑向）。**線性漸層**可以是垂直、水平或對角線的樣式；而**放射狀漸層**的各樣式都是從中心點以圓的方式做放射狀而出。來看看每一個範例吧！

線性漸層

線性漸層有三種基本的形式：水平、垂直與對角線。透過設定**色彩停止點**（*color stops*）來控制填色物件的漸層變化。

線性水平漸層 範例2-14是一個簡單的水平漸層，結果如圖2-23。

範例2-14 水平線性漸層

```
function drawScreen() {

      //水平漸層的值必須設定為0
      var gr = context.createLinearGradient(0, 0, 100, 0);

      //增加色彩停止點
      gr.addColorStop(0,'rgb(255,0,0)');
      gr.addColorStop(.5,'rgb(0,255,0)');
      gr.addColorStop(1,'rgb(255,0,0)');

      //使用fillStyle設定漸層
      context.fillStyle = gr;
      context.fillRect(0, 0,100,100);

   }
```

圖2-23 水平線性漸層

為了要繪製水平漸層，我們必須建立一個變數（gr）來參考到新的漸層，這是我們所設定的樣子：

```
var gr = context.createLinearGradient(0,0,100,0);
```

在`createLinearGradient` method裡的四個參數分別是漸層的起點－左上角的*x,y*座標，以及漸層的終點－右下角的*x,y*座標；範例中漸層的起點是`0,0`，終點是`100,0`。要注意的是當我們在繪水平漸層時，兩個*y*座標的值都會是`0`。而在繪垂直漸層時則剛好相反。

一但定義好了漸層的大小之後，接下來就是要加入色彩停止點（color stops）並設定其2個參數值：第一個參數必須是介於0.0和1.0之間的數值，定義在漸層中色彩之間的相對位置，第二個參數則是要填上的顏色。

```
gr.addColorStop(0,'rgb(255,0,0)');
gr.addColorStop(.5,'rgb(0,255,0)');
gr.addColorStop(1,'rgb(255,0,0)');
```

所以，在範例2-14裡，我們設定了紅色在`0`的位置，綠色在中間`0.5`的地方，另一個紅色則是在1的位置；這將會對等的畫出紅到綠再到紅的漸層。

接下來，要使用context.fillStyle來設定漸層的樣式：

```
context.fillStyle = gr;
```

最後，在canvas上繪一個長方形：

```
context.fillRect(0, 0, 100, 100);
```

值得注意的是，我們所畫出來的長方形就是漸層的實際大小，但我們還是可以透過以下的方式來改變長方形的大小：

```
context.fillRect(0, 100, 50, 100);
context.fillRect(0, 200, 200, 100);
```

範例2-15是在範例2-14中多加了這兩行，它的結果就會如圖2-24所示。漸層色會填滿可用的空間，而超出漸層色所定義的區域則會使用最後的一個顏色來填滿。

圖2-24 多重物件上的線形水平漸層

範例2-15 多重物件上的漸層

```
function drawScreen() {

   var gr = context.createLinearGradient(0, 0, 100, 0);

   //增加色彩停止點
   gr.addColorStop(0,'rgb(255,0,0)');
   gr.addColorStop(.5,'rgb(0,255,0)');
   gr.addColorStop(1,'rgb(255,0,0)');

   //使用fillStyle設定漸層
   context.fillStyle = gr;
   context.fillRect(0, 0, 100, 100);
   context.fillRect(0, 100, 50, 100);
   context.fillRect(0, 200, 200, 100);

   }
```

使用水平漸層於圖形之輪廓 漸層可使用於任何的形狀上－即使是形狀的輪廓，也是可以的。範例2-16是從範例2-15而來，只是用strokeRect來取代填滿顏色的長方形；圖2-25呈現出非常不一樣的結果。

範例2-16 輪廓的水平漸層

```
function drawScreen() {

    var gr = context.createLinearGradient(0, 0, 100, 0);

    //增加色彩停止點
    gr.addColorStop(0,'rgb(255,0,0)');
    gr.addColorStop(.5,'rgb(0,255,0)');
    gr.addColorStop(1,'rgb(255,0,0)');
    //使用fillStyle設定漸層
    context.strokeStyle = gr;
    context.strokeRect(0, 0, 100, 100);
    context.strokeRect(0, 100, 50, 100);
    context.strokeRect(0, 200, 200, 100);

}
```

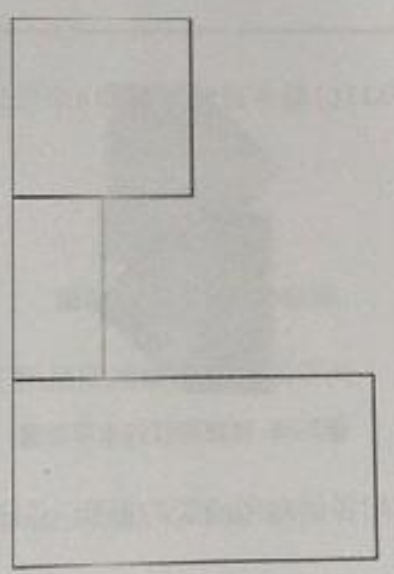


圖2-25 輪廓的水平漸層

使用水平漸層於複雜的形狀 我們也可以在封閉式的形狀中使用線性漸層，如範例2-17所示；它是一個終點又回到和起點同一個點上的封閉形狀。

範例2-17 複雜形狀的水平漸層

```
function drawScreen() {

    var gr = context.createLinearGradient(0, 0, 100, 0);

    //增加色彩停止點
    gr.addColorStop(0,'rgb(255,0,0)');
    gr.addColorStop(.5,'rgb(0,255,0)');
    gr.addColorStop(1,'rgb(255,0,0)');

    //使用fillStyle設定漸層
    context.fillStyle = gr;
    context.beginPath();
    context.moveTo(0,0);
    context.lineTo(50,0);
    context.lineTo(100,50);
    context.lineTo(50,100);
    context.lineTo(0,100);
    context.lineTo(0,0);
    context.stroke();
    context.fill();
    context.closePath();

}
```

此範例中，使用了`context.fill()`指令及`fillStyle`來完成填色的效果，結果正如圖2-26所示：

圖2-26 複雜形狀的水平漸層

圖2-26顯示了使用我們所建立的各個點所繪製的形狀，只要所有的點之間連成的形狀是封閉的，就可以畫出漸層的效果。

垂直漸層 垂直漸層和水平漸層繪製的方式非常的類似：不同之處是y的值不可以為0，而兩個x的值都必須為0。範例2-18是從範例2-17修改而來，只是將水平漸層改為垂直漸層，其結果如圖2-27所示。

範例2-18 垂直漸層

```
function drawScreen() {

     var gr = context.createLinearGradient(0, 0, 0, 100);

     // 增加色彩停止點
     gr.addColorStop(0,'rgb(255,0,0)');
     gr.addColorStop(.5,'rgb(0,255,0)');
     gr.addColorStop(1,'rgb(255,0,0)');

     // 使用fillStyle設定漸層
     context.fillStyle = gr;
     context.beginPath();
     context.moveTo(0,0);
     context.lineTo(50,0);
     context.lineTo(100,50);
     context.lineTo(50,100);
     context.lineTo(0,100);
     context.lineTo(0,0);
     //context.stroke();
     context.fill();
     context.closePath();

  }
```

圖2-27 垂直漸層的範例

範例2-18與範例2-17唯一的不同點是使用漸層的方式不同。

水平漸層（範例2-17）

```
var gr = context.createLinearGradient(0, 0, 100, 0);
```

垂直漸層（範例2-18）

```
var gr = context.createLinearGradient(0, 0, 0, 100);
```

繪製輪廓的規則和水平漸層一樣，範例2-19是從範例2-18而來，它使用繪製輪廓的方式來取代在物件上填滿顏色，結果請參考圖2-28。

範例2-19 垂直漸層的輪廓

```
function drawScreen() {

      var gr = context.createLinearGradient(0, 0, 0, 100);

      //增加色彩停止點
      gr.addColorStop(0,'rgb(255,0,0)');
      gr.addColorStop(.5,'rgb(0,255,0)');
      gr.addColorStop(1,'rgb(255,0,0)');

      //使用fillStyle設定漸層
      context.strokeStyle = gr;
      context.beginPath();
      context.moveTo(0,0);
      context.lineTo(50,0);
      context.lineTo(100,50);
      context.lineTo(50,100);
      context.lineTo(0,100);
      context.lineTo(0,0);
      context.stroke();
      context.closePath();

   }
```

對角線漸層 可以很容易的透過createLinearGradient()函數的第二個x與y參數來畫出對角線漸層樣式：

```
var gr= context.createLinearGradient(0, 0, 100, 100);
```

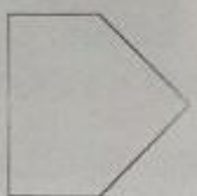


圖2-28 垂直漸層的輪廓

要畫出如圖2-29這麼完美的對角線漸層，需填色的正方形與對角線漸層樣式的大小必須一樣大，範例2-20就是它的程式碼。

範例2-20 對角線漸層

```
function drawScreen() {

      var gr = context.createLinearGradient(0, 0, 100, 100);

      //增加色彩停止點
```

```
    gr.addColorStop(0,'rgb(255,0,0)');
    gr.addColorStop(.5,'rgb(0,255,0)');
    gr.addColorStop(1,'rgb(255,0,0)');

    //使用fillStyle設定漸層
    context.fillStyle = gr;
    context.beginPath();
    context.moveTo(0,0);
    context.fillRect(0,0,100,100)
    context.closePath();

}
```

圖2-29 對角線漸層的範例

放射狀漸層

放射狀漸層和線性漸層定義的過程其實非常的相似，雖然放射狀漸層使用了6個參數來做初始化的動作，不像線性漸層只使用了4個參數，但它們都是靠著彩色停止點（color stops）來改變顏色的。

這六個參數分別是用來定義中心點與兩個圓的半徑，第一個是"起始"的圓，第二個是"終止"的圓，來看一下範例吧：

```
var gr = context.createRadialGradient(50,50,25,50,50,100);
```

第一個圓的中心點在50,50的座標上，且半徑為25；第二個的中心點也在50,50的座標上，半徑為100；這將會產生2個同心圓。

使用和線性漸層一樣的方式來設定彩色停止點（color stops）：

```
gr.addColorStop(0,'rgb(255,0,0)');
gr.addColorStop(.5,'rgb(0,255,0)');
gr.addColorStop(1,'rgb(255,0,0)');
```

範例2-21將這些都放在一起，它所畫出來的樣子就會像圖2-30那樣。

範例2-21 一個簡單的放射狀漸層

```
function drawScreen() {

    var gr = context.createRadialGradient(50,50,25,50,50,100);

    //增加色彩停止點
    gr.addColorStop(0,'rgb(255,0,0)');
    gr.addColorStop(.5,'rgb(0,255,0)');
    gr.addColorStop(1,'rgb(255,0,0)');

    //使用fillStyle設定漸層
    context.fillStyle = gr;
    context.fillRect(0, 0, 200, 200);

}
```

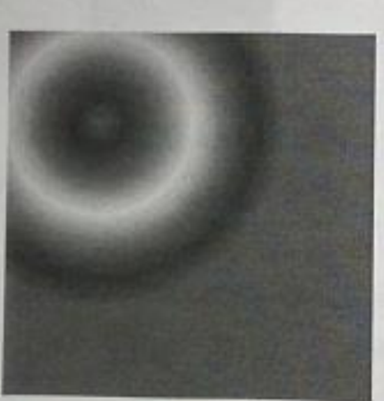


圖2-30 一個簡單的放射狀漸層

範例2-22將第二個圓的圓心做了點偏移，所產生後的結果如圖2-31所示。

範例2-22 複雜的放射狀漸層

```
function drawScreen() {

    var gr = context.createRadialGradient(50,50,25,100,100,100);

    //增加色彩停止點
    gr.addColorStop(0,'rgb(255,0,0)');
    gr.addColorStop(.5,'rgb(0,255,0)');
    gr.addColorStop(1,'rgb(255,0,0)');

    //使用fillStyle設定漸層
    context.fillStyle = gr;
    context.fillRect(0, 0, 200, 200);

}
```

圖2-31 複雜的放射狀漸層

和線形漸層一樣，我們也可以將複雜的形狀做放射狀的漸層。範例2-23使用了弧形來做放射狀的漸層，其結果將會如圖2-32所示。

範例2-23 圓形內的放射狀漸層樣式

```
function drawScreen() {

      var gr = context.createRadialGradient(50,50,25,100,100,100);

      //增加色彩停止點
      gr.addColorStop(0,'rgb(255,0,0)');
      gr.addColorStop(.5,'rgb(0,255,0)');
      gr.addColorStop(1,'rgb(255,0,0)');

      //使用fillStyle設定漸層
      context.fillStyle = gr;
      context.arc(100, 100, 100, (Math.PI/180)*0, (Math.PI/180)*360, false);
      context.fill();

   }
```

範例2-23的放射狀漸層是從範例2-22修改而來，只是將長方形改成了圓形；因此背景的紅色方形都沒有出現。

同樣的，也可以將放射狀漸層應用到形狀的輪廓上，其結果將如範例2-24及圖2-33所示。

圖2-32 圓形內的放射狀漸層樣式

範例2-24 圓形輪廓上的漸層樣式

```
function drawScreen() {

      var gr = context.createRadialGradient(50,50,25,100,100,100);

      //增加色彩停止點
      gr.addColorStop(0,'rgb(255,0,0)');
      gr.addColorStop(.5,'rgb(0,255,0)');
      gr.addColorStop(1,'rgb(255,0,0)');

      //使用fillStyle設定漸層
      context.strokeStyle = gr;
      context.arc(100, 100, 50, (Math.PI/180)*0, (Math.PI/180)*360, false)
      context.stroke();

   }
```

圖2-33 圓形輪廓上的漸層樣式

範例2-24建立了一個比範例2-23還小的一個圓，如此放射狀漸層就會表現在圓形的輪廓上。

圖形

在第四章我們會在canvas上使用影像圖案；在此，要很快的看一下如何在我們所繪製的圖形上貼上影像圖案。

要將圖案填滿必須使用`createPattern()`函數，它有二個參數：第一個是影像物件的instance，第二個參數是一個字串，它是用來定義在要顯示的形狀裡該圖案要如何不斷的被重覆顯示出來。我們可以使用一個載入的影像檔案，或者用另一個完整的canvas來當作填充的樣示以畫出圖形。

目前有4種將圖案填滿的方式：

- `repeat`
- `repeat-x`
- `repeat-y`
- `no-repeat`

目前各個瀏覽器對此四種樣式支援的程度都不同，不過標準的standard是最常見的。現在就讓我們來看看這四種不同樣式所呈現出的結果吧！

圖2-34顯示了一個簡單的圖案，我們要使用它來做功能的測試；它是一個在透明背景上20x20大小的綠色圓形，我們將它命名為*fill_20x20.gif*。

圖2-34 使用了名為fill_20x20.gif的圖案做填充的效果

範例2-25使用第一個repeat字串來做測試，首先建立一個方形，然後使用這個綠色的圓來做填充的效果，結果如圖2-35所示：

範例2-25 使用repeat來將image檔案做填充的效果

```
function drawScreen() {

      var fillImg = new Image();
      fillImg.src = 'fill_20x20.gif';
      fillImg.onload = function(){

         var fillPattern = context.createPattern(fillImg,'repeat');
         context.fillStyle = fillPattern;
         context.fillRect(0,0,200,200);

      }

   }
```

在圖像還沒完全被載入前，最好不要使用Image instance；這在第四章將會有更詳細的說明，但現在，先簡單的建立一個onload的事件處理函數，當Image要被使用到時，就必須呼叫此函數。repeat字串可以將200x200的正方形做完整的填滿效果，接下來要看看程式碼裡其他的重複字串方式是如何呈現出結果的（範例2-26），圖例請參考圖2-36至2-38。

圖2-35 repeat填充的範例

範例2-26 使用no-repeat、repeat-x及repeat-y字串

```
function drawScreen() {


    var fillImg = new Image();
    fillImg.src = 'fill_20x20.gif';

    fillImg.onload = function(){

      var fillPattern1 = context.createPattern(fillImg,'no-repeat');
      var fillPattern2 = context.createPattern(fillImg,'repeat-x');
      var fillPattern3 = context.createPattern(fillImg,'repeat-y');

      context.fillStyle = fillPattern1;
      context.fillRect(0,0,100,100);

      context.fillStyle = fillPattern2;
      context.fillRect(0,110,100,100);

      context.fillStyle = fillPattern3;
      context.fillRect(0,220,100,100);

    }

}
```

這些圖案在不同的瀏覽器會有些許的差異。

當使用repeat-x與repeat-y的重複方式時，只有FireFox會將所有有意義的東西都呈現出來。我們將在第四章討論到更多的填充範例，以及圖像的各種使用方式。

圖2-36 在Safari中使用no-repeat、repeat-x、repeat-y

圖2-37 在FireFox中使用no-repeat、repeat-x、repeat-y

圖2-38 在Chrome中使用no-repeat、repeat-x、repeat-y

在Canvas上建立圖形陰影

Canvas上有4個參數可為我們所繪製的圖形加上陰影的效果，這個功能在有支援HTML5的瀏覽器上並非都可行。

要呈現陰影的效果必須設定Canvas上的4個屬性：

- shadowOffsetX
- shadowOffsetY
- shadowBlur
- shadowColor

shadowOffsetX與shadowOffsetY的值可以是正數值也可以是負數值；負值表示陰影會往上或往左延伸，shadowBlur屬性是用來設定陰影的模糊程度，這三個參數並不受轉換矩陣所影響；shadowColor可以是HTML4所規定的任何彩色字串－rgb()或rgba()－或是十六進位的字串值。

範例2-27與圖2-39是各種不同圖形配上不同的陰影設定所呈現出的結果。

圖2-39 為圖形物件增加陰影效果

範例2-27 為圖形物件增加陰影效果

```
function drawScreen() {

    context.fillStyle = 'red';

    context.shadowOffsetX = 4;
    context.shadowOffsetY = 4;
    context.shadowColor = 'black';
    context.shadowBlur = 4;
    context.fillRect(10,10,100,100);

    context.shadowOffsetX = -4;
    context.shadowOffsetY = -4;
    context.shadowColor = 'black';
    context.shadowBlur = 4;
    context.fillRect(150,10,100,100);

    context.shadowOffsetX = 10;
    context.shadowOffsetY = 10;
    context.shadowColor = 'rgb(100,100,100)';
    context.shadowBlur = 8;
    context.arc(200, 300, 100, (Math.PI/180)*0, (Math.PI/180)*360, false)
    context.fill();
}
```

如我們所看到的，如果隨著`shadowBlur`的值來調整`shadowOffset`值，就會呈現各種不同的陰影效果。同樣的，我們也可以為複雜的形狀、線條與弧形來繪製陰影。

接下來呢?

本章我們花了大部份的時間在墊定基礎，介紹了建構一個簡單與複雜形狀的方式，以及如何在canvas上繪出這些圖形並將它們做各種不同的轉變。也討論了如何為這些圖形做組合、旋轉、縮放、移動、填充及陰影的效果。但我們只是剛剛開始探索著HTML5 Canvas而已；下個章節，我們將要來看看在Canvas上如何建立並控制文字物件。