```
    // 绘制红色矩形
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    // 绘制半透明蓝色矩形
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);

    // 在前两个矩形重叠的区域擦除一个矩形区域
    context.clearRect(40, 40, 10, 10);
}
```

以上代码在两个矩形重叠的区域上擦除了一个小矩形，图 18-3 展示了结果。

图　18-3

18.3.3　绘制路径

2D 绘图上下文支持很多在画布上绘制路径的方法。通过路径可以创建复杂的形状和线条。要绘制路径，必须首先调用 `beginPath()` 方法以表示要开始绘制新路径。然后，再调用下列方法来绘制路径。

- `arc(x, y, radius, startAngle, endAngle, counterclockwise)`：以坐标 `(x, y)` 为圆心，以 `radius` 为半径绘制一条弧线，起始角度为 `startAngle`，结束角度为 `endAngle`（都是弧度）。最后一个参数 `counterclockwise` 表示是否逆时针计算起始角度和结束角度（默认为顺时针）。
- `arcTo(x1, y1, x2, y2, radius)`：以给定半径 `radius`，经由 `(x1, y1)` 绘制一条从上一点到 `(x2, y2)` 的弧线。
- `bezierCurveTo(c1x, c1y, c2x, c2y, x, y)`：以 `(c1x, c1y)` 和 `(c2x, c2y)` 为控制点，绘制一条从上一点到 `(x, y)` 的弧线（三次贝塞尔曲线）。
- `lineTo(x, y)`：绘制一条从上一点到 `(x, y)` 的直线。
- `moveTo(x, y)`：不绘制线条，只把绘制光标移动到 `(x, y)`。
- `quadraticCurveTo(cx, cy, x, y)`：以 `(cx, cy)` 为控制点，绘制一条从上一点到 `(x, y)` 的弧线（二次贝塞尔曲线）。
- `rect(x, y, width, height)`：以给定宽度和高度在坐标点 `(x, y)` 绘制一个矩形。这个方法与 `strokeRect()` 和 `fillRect()` 的区别在于，它创建的是一条路径，而不是独立的图形。

创建路径之后，可以使用 `closePath()` 方法绘制一条返回起点的线。如果路径已经完成，则既可以指定 `fillStyle` 属性并调用 `fill()` 方法来填充路径，也可以指定 `strokeStyle` 属性并调用 `stroke()` 方法来描画路径，还可以调用 `clip()` 方法基于已有路径创建一个新剪切区域。

下面这个例子使用前面提到的方法绘制了一个不带数字的表盘：

```
let drawing = document.getElementById("drawing");

// 确保浏览器支持<canvas>
if (drawing.getContext) {
  let context = drawing.getContext("2d");

  // 创建路径
  context.beginPath();

  // 绘制外圆
  context.arc(100, 100, 99, 0, 2 * Math.PI, false);

  // 绘制内圆
  context.moveTo(194, 100);
  context.arc(100, 100, 94, 0, 2 * Math.PI, false);

  // 绘制分针
  context.moveTo(100, 100);
  context.lineTo(100, 15);

  // 绘制时针
  context.moveTo(100, 100);
  context.lineTo(35, 100);

  // 描画路径
  context.stroke();
}
```

这个例子使用 `arc()`绘制了两个圆形，一个外圆和一个内圆，以构成表盘的边框。外圆半径 99 像素，原点为(100,100)，也就是画布的中心。要绘制完整的圆形，必须从 0 弧度绘制到 2π 弧度（使用数学常量 `Math.PI`）。而在绘制内圆之前，必须先把路径移动到内圆上的一点，以避免绘制出多余的线条。第二次调用 `arc()`时使用了稍小一些的半径，以呈现边框效果。然后，再组合运用 `moveTo()`和 `lineTo()`分别绘制分针和时针。最后一步是调用 `stroke()`，得到如图 18-4 所示的图像。

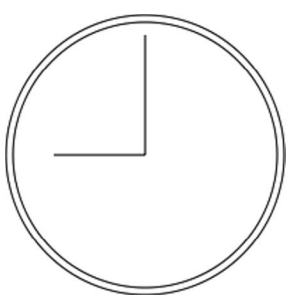

图 18-4

路径是 2D 上下文的主要绘制机制，为绘制结果提供了很多控制。因为路径经常被使用，所以也有一个 `isPointInPath()`方法，接收 *x* 轴和 *y* 轴坐标作为参数。这个方法用于确定指定的点是否在路径上，可以在关闭路径前随时调用，比如：

```
if (context.isPointInPath(100, 100)) {
  alert("Point (100, 100) is in the path.");
}
```

2D 上下文的路径 API 非常可靠，可用于创建涉及各种填充样式、描述样式等的复杂图像。

18.3.4　绘制文本

文本和图像混合也是常见的绘制需求，因此2D绘图上下文还提供了绘制文本的方法，即 fillText() 和 strokeText()。这两个方法都接收 4 个参数：要绘制的字符串、*x* 坐标、*y* 坐标和可选的最大像素宽度。而且，这两个方法最终绘制的结果都取决于以下 3 个属性。

- font：以 CSS 语法指定的字体样式、大小、字体族等，比如"10px Arial"。
- textAlign：指定文本的对齐方式，可能的值包括"start"、"end"、"left"、"right"和"center"。推荐使用"start"和"end"，不使用"left"和"right"，因为前者无论在从左到右书写的语言还是从右到左书写的语言中含义都更明确。
- textBaseLine：指定文本的基线，可能的值包括"top"、"hanging"、"middle"、"alphabetic"、"ideographic"和"bottom"。

这些属性都有相应的默认值，因此没必要每次绘制文本时都设置它们。fillText()方法使用 fillStyle 属性绘制文本，而 strokeText()方法使用 strokeStyle 属性。通常，fillText()方法是使用最多的，因为它模拟了在网页中渲染文本。例如，下面的例子会在前一节示例的表盘顶部绘制数字“12”：

```
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);
```

结果就得到了如图 18-5 所示的图像。

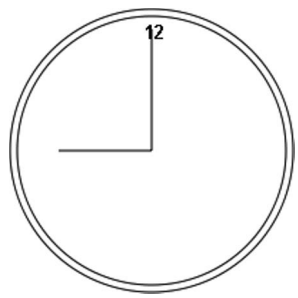

图　18-5

因为把 textAlign 设置为了"center"，把 textBaseline 设置为了"middle"，所以(100, 20)表示文本水平和垂直中心点的坐标。如果 textAlign 是"start"，那么 *x* 坐标在从左到右书写的语言中表示文本的左侧坐标，而"end"会让 *x* 坐标在从左到右书写的语言中表示文本的右侧坐标。例如：

```
// 正常
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);
// 与开头对齐
context.textAlign = "start";
context.fillText("12", 100, 40);
// 与末尾对齐
context.textAlign = "end";
context.fillText("12", 100, 60);
```

字符串"12"被绘制了3次，每次使用的坐标都一样，但textAlign值不同。为了让每个字符串不至于重叠，每次绘制的*y*坐标都会设置得大一些。结果就是如图18-6所示的图像。

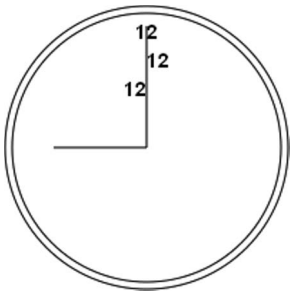

图 18-6

因为表盘中垂直的线条是居中的，所以文本的对齐方式就一目了然了。类似地，通过修改textBaseline属性，可以改变文本的垂直对齐方式。比如，设置为"top"意味着*y*坐标表示文本顶部，"bottom"表示文本底部，"hanging"、"alphabetic"和"ideographic"分别引用字体中特定的基准点。

由于绘制文本很复杂，特别是想把文本绘制到特定区域的时候，因此2D上下文提供了用于辅助确定文本大小的measureText()方法。这个方法接收一个参数，即要绘制的文本，然后返回一个TextMetrics对象。这个返回的对象目前只有一个属性width，不过将来应该会增加更多度量指标。

measureText()方法使用font、textAlign和textBaseline属性当前的值计算绘制指定文本后的大小。例如，假设要把文本"Hello world!"放到一个140像素宽的矩形中，可以使用以下代码，从100像素的字体大小开始计算，不断递减，直到文本大小合适：

```
let fontSize = 100;
context.font = fontSize + "px Arial";
while(context.measureText("Hello world!").width > 140) {
  fontSize--;
  context.font = fontSize + "px Arial";
}
context.fillText("Hello world!", 10, 10);
context.fillText("Font size is " + fontSize + "px", 10, 50);
```

fillText()和strokeText()方法还有第四个参数，即文本的最大宽度。这个参数是可选的（Firefox 4是第一个实现它的浏览器），如果调用fillText()和strokeText()时提供了此参数，但要绘制的字符串超出了最大宽度限制，则文本会以正确的字符高度绘制，这时字符会被水平压缩，以达到限定宽度。图18-7展示了这个参数的效果。

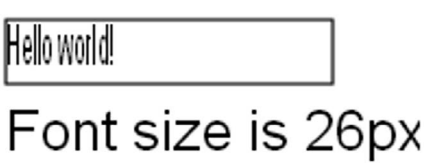

图 18-7

绘制文本是一项比较复杂的操作，因此支持<canvas>元素的浏览器不一定全部实现了相关的文本绘制API。

18.3.5　变换

上下文变换可以操作绘制在画布上的图像。2D 绘图上下文支持所有常见的绘制变换。在创建绘制上下文时，会以默认值初始化变换矩阵，从而让绘制操作如实应用到绘制结果上。对绘制上下文应用变换，可以导致以不同的变换矩阵应用绘制操作，从而产生不同的结果。

以下方法可用于改变绘制上下文的变换矩阵。

- `rotate(angle)`：围绕原点把图像旋转 `angle` 弧度。
- `scale(scaleX, scaleY)`：通过在 *x* 轴乘以 `scaleX`、在 *y* 轴乘以 `scaleY` 来缩放图像。`scaleX` 和 `scaleY` 的默认值都是 1.0。
- `translate(x, y)`：把原点移动到`(x, y)`。执行这个操作后，坐标(0, 0)就会变成`(x, y)`。
- `transform(m1_1, m1_2, m2_1, m2_2, dx, dy)`：像下面这样通过矩阵乘法直接修改矩阵。

```
m1_1 m1_2 dx
m2_1 m2_2 dy
0    0    1
```

- `setTransform(m1_1, m1_2, m2_1, m2_2, dx, dy)`：把矩阵重置为默认值，再以传入的参数调用 `transform()`。

变换可以简单，也可以复杂。例如，在前面绘制表盘的例子中，如果把坐标原点移动到表盘中心，那再绘制表针就非常简单了：

```
let drawing = document.getElementById("drawing");

// 确保浏览器支持<canvas>
if (drawing.getContext) {
  let context = drawing.getContext("2d");

  // 创建路径
  context.beginPath();

  // 绘制外圆
  context.arc(100, 100, 99, 0, 2 * Math.PI, false);

  // 绘制内圆
  context.moveTo(194, 100);
  context.arc(100, 100, 94, 0, 2 * Math.PI, false);

  // 移动原点到表盘中心
  context.translate(100, 100);

  // 绘制分针
  context.moveTo(0, 0);
  context.lineTo(0, -85);

  // 绘制时针
  context.moveTo(0, 0);
  context.lineTo(-65, 0);

  // 描画路径
  context.stroke();
}
```

把原点移动到(100, 100)，也就是表盘的中心后，要绘制表针只需简单的数学计算即可。这是因为所有计算都是基于(0, 0)，而不是(100, 100)了。当然，也可以使用 `rotate()`方法来转动表针：

```
let drawing = document.getElementById("drawing");

// 确保浏览器支持<canvas>
if (drawing.getContext) {
  let context = drawing.getContext("2d");

  // 创建路径
  context.beginPath();

  // 绘制外圆
  context.arc(100, 100, 99, 0, 2 * Math.PI, false);

  // 绘制内圆
  context.moveTo(194, 100);
  context.arc(100, 100, 94, 0, 2 * Math.PI, false);

  // 移动原点到表盘中心
  context.translate(100, 100);

  // 旋转表针
  context.rotate(1);

  // 绘制分针
  context.moveTo(0, 0);
  context.lineTo(0, -85);

  // 绘制时针
  context.moveTo(0, 0);
  context.lineTo(-65, 0);

  // 描画路径
  context.stroke();
}
```

因为原点已经移动到表盘中心，所以旋转就是以该点为圆心的。这相当于把表针一头固定在表盘中心，然后向右拨了一个弧度。结果如图 18-8 所示。

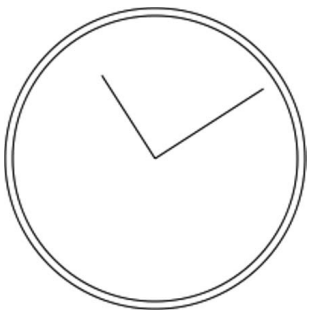

图 18-8

所有这些变换，包括 fillStyle 和 strokeStyle 属性，会一直保留在上下文中，直到再次修改它们。虽然没有办法明确地将所有值都重置为默认值，但有两个方法可以帮我们跟踪变化。如果想着什么时候再回到当前的属性和变换状态，可以调用 save()方法。调用这个方法后，所有这一时刻的设置会被放到一个暂存栈中。保存之后，可以继续修改上下文。而在需要恢复之前的上下文时，可以调用

restore()方法。这个方法会从暂存栈中取出并恢复之前保存的设置。多次调用 save()方法可以在暂存栈中存储多套设置，然后通过 restore()可以系统地恢复。下面来看一个例子：

```
context.fillStyle = "#ff0000";
context.save();

context.fillStyle = "#00ff00";
context.translate(100, 100);
context.save();

context.fillStyle = "#0000ff";
context.fillRect(0, 0, 100, 200);      // 在(100, 100)绘制蓝色矩形

context.restore();
context.fillRect(10, 10, 100, 200);    // 在(100, 100)绘制绿色矩形

context.restore();
context.fillRect(0, 0, 100, 200);      // 在(0, 0)绘制红色矩形
```

以上代码先将 fillStyle 设置为红色，然后调用 save()。接着，将 fillStyle 修改为绿色，坐标移动到(100, 100)，并再次调用 save()，保存设置。随后，将 fillStyle 属性设置为蓝色并绘制一个矩形。因为此时坐标被移动了，所以绘制矩形的坐标实际上是(100, 100)。在调用 restore()之后，fillStyle 恢复为绿色，因此这一次绘制的矩形是绿色的。而绘制矩形的坐标是(110, 110)，因为变换仍在起作用。再次调用 restore()之后，变换被移除，fillStyle 也恢复为红色。绘制最后一个矩形的坐标变成了(0, 0)。

注意，save()方法只保存应用到绘图上下文的设置和变换，不保存绘图上下文的内容。

18.3.6　绘制图像

2D 绘图上下文内置支持操作图像。如果想把现有图像绘制到画布上，可以使用 drawImage()方法。这个方法可以接收 3 组不同的参数，并产生不同的结果。最简单的调用是传入一个 HTML 的<img>元素，以及表示绘制目标的 *x* 和 *y* 坐标，结果是把图像绘制到指定位置。比如：

```
let image = document.images[0];
context.drawImage(image, 10, 10);
```

以上代码获取了文本中的第一个图像，然后在画布上的坐标(10, 10)处将它绘制了出来。绘制出来的图像与原来的图像一样大。如果想改变所绘制图像的大小，可以再传入另外两个参数：目标宽度和目标高度。这里的缩放只影响绘制的图像，不影响上下文的变换矩阵。比如下面的例子：

```
context.drawImage(image, 50, 10, 20, 30);
```

执行之后，图像会缩放到 20 像素宽、30 像素高。

还可以只把图像绘制到上下文中的一个区域。此时，需要给 drawImage()提供 9 个参数：要绘制的图像、源图像 *x* 坐标、源图像 *y* 坐标、源图像宽度、源图像高度、目标区域 *x* 坐标、目标区域 *y* 坐标、目标区域宽度和目标区域高度。这个重载后的 drawImage()方法可以实现最大限度的控制，比如：

```
context.drawImage(image, 0, 10, 50, 50, 0, 100, 40, 60);
```

最终，原始图像中只有一部分会绘制到画布上。这一部分从(0, 10)开始，50 像素宽、50 像素高。而绘制到画布上时，会从(0, 100)开始，变成 40 像素宽、60 像素高。

像这样可以实现如图 18-9 所示的有趣效果。

图 18-9

第一个参数除了可以是 HTML 的<img>元素，还可以是另一个<canvas>元素，这样就会把另一个画布的内容绘制到当前画布上。

结合其他一些方法，drawImage()方法可以方便地实现常见的图像操作。操作的结果可以使用 toDataURL()方法获取。不过有一种情况例外：如果绘制的图像来自其他域而非当前页面，则不能获取其数据。此时，调用 toDataURL()将抛出错误。比如，如果来自 www.example.com 的页面上绘制的是来自 www.wrox.com 的图像，则上下文就是“脏的”，获取数据时会抛出错误。

18.3.7 阴影

2D 上下文可以根据以下属性的值自动为已有形状或路径生成阴影。

- shadowColor：CSS 颜色值，表示要绘制的阴影颜色，默认为黑色。
- shadowOffsetX：阴影相对于形状或路径的 *x* 坐标的偏移量，默认为 0。
- shadowOffsetY：阴影相对于形状或路径的 *y* 坐标的偏移量，默认为 0。
- shadowBlur：像素，表示阴影的模糊量。默认值为 0，表示不模糊。

这些属性都可以通过 context 对象读写。只要在绘制图形或路径前给这些属性设置好适当的值，阴影就会自动生成。比如：

```
let context = drawing.getContext("2d");
// 设置阴影
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur  = 4;
context.shadowColor   = "rgba(0, 0, 0, 0.5)";
// 绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
// 绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

这里两个矩形使用了相同的阴影样式，得到了如图 18-10 所示的结果。

图　18-10

18.3.8　渐变

渐变通过 CanvasGradient 的实例表示，在 2D 上下文中创建和修改都非常简单。要创建一个新的线性渐变，可以调用上下文的 createLinearGradient()方法。这个方法接收 4 个参数：起点 *x* 坐标、起点 *y* 坐标、终点 *x* 坐标和终点 *y* 坐标。调用之后，该方法会以指定大小创建一个新的 CanvasGradient 对象并返回实例。

有了 gradient 对象后，接下来要使用 addColorStop()方法为渐变指定色标。这个方法接收两个参数：色标位置和 CSS 颜色字符串。色标位置通过 0 ~ 1 范围内的值表示，0 是第一种颜色，1 是最后一种颜色。比如：

```
let gradient = context.createLinearGradient(30, 30, 70, 70);
gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
```

这个 gradient 对象现在表示的就是在画布上从(30, 30)到(70, 70)绘制一个渐变。渐变的起点颜色为白色，终点颜色为黑色。可以把这个对象赋给 fillStyle 或 strokeStyle 属性，从而以渐变填充或描画绘制的图形：

```
// 绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
// 绘制渐变矩形
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

为了让渐变覆盖整个矩形，而不只是其中一部分，两者的坐标必须搭配合适。以上代码将得到如图 18-11 所示的结果。

图　18-11

如果矩形没有绘制到渐变的范围内，则只会显示部分渐变。比如：

```
context.fillStyle = gradient;
context.fillRect(50, 50, 50, 50);
```

以上代码执行之后绘制的矩形只有左上角有一部分白色。这是因为矩形的起点在渐变的中间，此时颜色的过渡几乎要完成了。结果矩形大部分地方是黑色的，因为渐变不会重复。保持渐变与形状的一致非常重要，有时候可能需要写个函数计算相应的坐标。比如：

```
function createRectLinearGradient(context, x, y, width, height) {
  return context.createLinearGradient(x, y, x+width, y+height);
}
```

这个函数会基于起点的 x、y 坐标和传入的宽度、高度创建渐变对象，之后调用 `fillRect()`方法时可以使用相同的值：

```
let gradient = createRectLinearGradient(context, 30, 30, 50, 50);
gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
// 绘制渐变矩形
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```


计算坐标是使用画布时重要而复杂的问题。使用类似 `createRectLinearGradient()`这样的辅助函数能让计算坐标简单一些。

径向渐变（或放射性渐变）要使用 `createRadialGradient()`方法来创建。这个方法接收 6 个参数，分别对应两个圆形圆心的坐标和半径。前 3 个参数指定起点圆形中心的 x、y 坐标和半径，后 3 个参数指定终点圆形中心的 x、y 坐标和半径。在创建径向渐变时，可以把两个圆形想象成一个圆柱体的两个圆形表面。把一个表面定义得小一点，另一个定义得大一点，就会得到一个圆锥体。然后，通过移动两个圆形的圆心，就可以旋转这个圆锥体。

要创建起点圆心在形状中心并向外扩散的径向渐变，需要将两个圆形设置为同心圆。比如，要在前面例子中矩形的中心创建径向渐变，则渐变的两个圆形的圆心都必须设置为(55, 55)。这是因为矩形的起点是(30, 30)，终点是(80, 80)。代码如下：

```
let gradient = context.createRadialGradient(55, 55, 10, 55, 55, 30);
gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
// 绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
// 绘制渐变矩形
context.fillStyle = gradient;
context.fillRect(30, 30, 50, 50);
```

运行以上代码会得到如图 18-12 所示的效果。

图 18-12

因为创建起来要复杂一些，所以径向渐变比较难处理。不过，通常情况下，起点和终点的圆形都是同心圆，只要定义好圆心坐标，剩下的就是调整各自半径的问题了。