什么是 canvas?

HTML5 <canvas> 元素用于图形的绘制，通过脚本 (通常是 JavaScript)来完成.

<canvas> 标签只是图形容器，您必须使用脚本来绘制图形。

你可以通过多种方法使用 canvas 绘制路径,线段、圆、圆弧，字符以及添加图像。

浏览器支持

Internet Explorer 9、Firefox、Opera、Chrome 和 Safari 支持 <canvas> 标签的属性及方法。

注意: Internet Explorer 8 及更早的 IE 版本不支持 <canvas> 元素。

Canvas 绘制

创建一个画布（Canvas）

一个画布在网页中是一个矩形框，通过 <canvas> 元素来绘制.

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

使用 JavaScript 矩形

```
var c=document.getElementById("myCanvas");  //首先，找到 <canvas> 元素

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

//创建 context 对象，getContext() 方法返回一个用于在画布上绘图的环境
```

//返回值
一个 CanvasRenderingContext2D 对象，使用它可以绘制到 Canvas 元素中。该对象实现了一个画布所使用的大多数方法

```
//下面的两行代码绘制一个红色的矩形

ctx.fillStyle="#FF0000";

ctx.fillRect(0,0,150,75);
```

getContext("2d") 对象是内建的 HTML5 对象，拥有多种绘制路径、矩形、圆形、字符以及添加图像的方法。

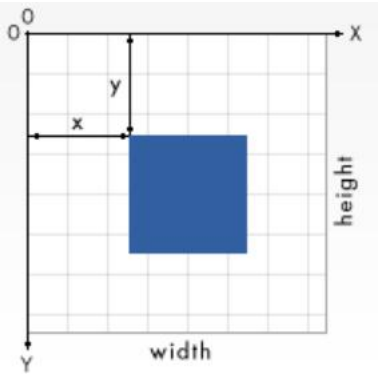
Canvas 坐标

canvas 是一个二维网格。

canvas 的左上角坐标为 (0,0)

上面的 fillRect 方法拥有参数 (0,0,150,75)。

意思是：在画布上绘制 150x75 的矩形，从左上角开始 (0,0)。



translate(x, y)
translate 方法接受两个参数。*x* 是左右偏移量，*y* 是上下偏移量，

如果不使用 translate 方法，那么所有矩形都将被绘制在相同的位置（0,0）。translate 方法同时让我们可以任意放置这些图案，而不需要在 fillRect() 方法中手工调整坐标值，既好理解也方便使用

ctx.translate(75,75);

添加样式和颜色

如果我们想要给图形上色，有两个重要的属性可以做到：fillStyle 和 strokeStyle。
fillStyle = color
设置图形的填充颜色。

strokeStyle = color
设置图形轮廓的颜色。

color 可以是表示 CSS 颜色值的字符串，渐变对象或者图案对象。我们迟些再回头探讨渐变和图案对象。默认情况下，

线条和填充颜色都是黑色（CSS 颜色值 #000000）。
注意: 一旦您设置了 strokeStyle 或者 fillStyle 的值，那么这个新值就会成为新绘制的图形的默认值。如果你要给每个图形上不同的颜色，你需要重新设置 fillStyle 或 strokeStyle 的值

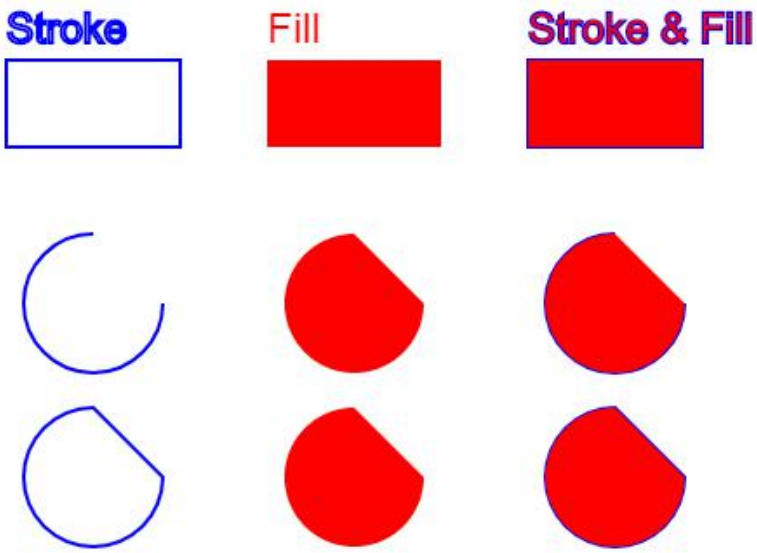
```
// 这些 fillStyle 的值均为 '橙色'
ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
ctx.fillStyle = "rgb(255,165,0)";
ctx.fillStyle = "rgba(255,165,0,1)";
```

Canvas 绘图环境中有些属于立即绘制图形方法，有些绘图方法是基于路径的

立即绘制图形方法仅有两个 strokeRect(),fillRect(),虽然 strokezText(),fillText()方法也是立即绘制的，但是文本不算是图形。

基于路径的绘制系统

使用这些绘制系统时，你需要先定义一个路径，然后再对其进行描边或填充，也可以描边加填充这样图形才能显示出来。
Canvas 中的三种绘制方式：



绘制一条线段

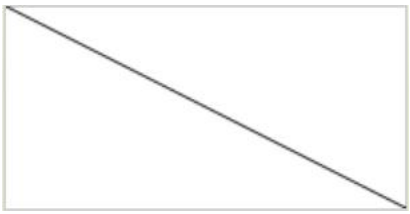
在 Canvas 上画线，我们将使用以下两种方法： 绘制直线，需要用到的方法 lineTo()。

moveTo(x,y) 定义线条开始坐标

lineTo(x,y) 定义线条结束坐标,绘制一条从当前位置到指定 x 以及 y 位置的直线

该方法有两个参数：x 以及 y，代表坐标系中直线结束的点。开始点和之前的绘制路径有关，之前路径的结束点就是接下来的开始点，等等。。。开始点也可以通过 moveTo()函数改变。

定义开始坐标(0,0), 和结束坐标 (200,100)。然后使用 stroke() 方法来绘制线条:



```
//画一条线
ctx.beginPath();
ctx.moveTo(305,30);
ctx.lineTo(400,100);
ctx.strokeStyle = 'red';
ctx.stroke();
```

基于路径的绘制方法，必须要描边或者填充。所以要想看到结果，我们必须还要使用 **stroke()**方法

beginPath()与 closePath()

从上面 canvas 中的三种绘制方式中我们可以看出，第二行的弧形路径是开放路径，最后一行的弧形是封闭路径。那么封闭的路径是怎么实现的呢?

下面我们来看看 canvas 中路径绘制中两个比较重要的方法

- beginPath(): 清除当前所有子路径，以此来重置当前路径，重新规划一条路径。
- closePath(): 用于封闭某段开放路径。不是必需的，如果图形是已经闭合了的，即当前点为开始点，该函数什么也不做。

先绘制出一条折线

```
function drawLine(){
    cxt.strokeStyle = 'green';
    cxt.lineWidth = 2;
    cxt.moveTo(50, 50);
    cxt.lineTo(50, 150);
    cxt.lineTo(150, 150);
    cxt.stroke();
}
```

修改上面例子中的代码在代码中添加 beginPath()与 closePath()方法

```
function drawLine(){
    //描边三角形
    cxt.strokeStyle = 'green';
    cxt.lineWidth = 2;
    cxt.beginPath();
    cxt.moveTo(50, 50);
    cxt.lineTo(50, 150);
    cxt.stroke();
    cxt.beginPath();
    cxt.lineTo(150, 150);
```

```
    cxt.lineTo(150, 250);
    cxt.stroke();
    cxt.closePath();
}
```

注意：当你调用 fill()函数时，所有没有闭合的形状都会自动闭合，所以此时 closePath()函数不是必须的。

但是调用 stroke()：如果你在 stroke()方法之前只用 closePath()会形成闭合路径，如果在 stroke()方法之后调用 closePath()方法，此时图形已经绘制完成，当前的绘制路径已经关闭，所以 closePath()方法不起作用。

网格的绘制

既然我们已经明白了如何绘制真正的 1 像素的线段，那我们就开始绘制网格

```
function drawLine(stepx, stepy){
    cxt.lineWidth = 0.5;
    cxt.strokeStyle = 'green';
    //绘制竖线
    for(var i= stepx + 0.5; i< cxt.canvas.width; i+= stepx){
        cxt.beginPath();
        cxt.moveTo(i, 0);
        cxt.lineTo(i, cxt.canvas.height);
        cxt.stroke();
    }
    //绘制横线
    for(var i= stepy + 0.5; i< cxt.canvas.height; i+= stepy){
        cxt.beginPath();
        cxt.moveTo(0, i);
        cxt.lineTo(cxt.canvas.width, i);
        cxt.stroke();
    }
}
drawLine(10, 10);
```

绘制路径介绍

用 canvas 绘制简单的图形，有点像用一支笔在纸上画画。通过不停调整落笔的位置、画各种各样的线条来勾勒出画的框架，再用各种颜色对封闭区间填充、对线条描边。在 canvas 中绘图的过程基本和这差不多。

canvas 里有**路径**的概念。可以理解成通过画笔画出的任意线条，这些线条甚至不用相连。在没描边(stroke)或是填充（fill）之前，路径在 canvas 画布上是看不到的。

Context2D 提供了一系列方法来绘制路径

图形的基本元素是路径。路径是通过不同颜色和宽度的线段或曲线相连形成的不同形状的点的集合。一个路径，甚至一个子路径，都是闭合的。使用路径绘制图形需要一些额外的步骤。

- 首先，你需要创建路径起始点。
- 然后你使用画图命令去画出路径。
- 之后你把路径封闭。
- 一旦路径生成，你就能通过描边或填充路径区域来渲染图形。

```
function draw() {
 var canvas = document.getElementById('canvas');
 if (canvas.getContext){
 var ctx = canvas.getContext('2d');

    ctx.beginPath();
    ctx.arc(75,75,50,0,Math.PI*2,true); // 绘制
    ctx.moveTo(110,75);
    ctx.arc(75,75,35,0,Math.PI,false);   // 口(顺时针)
    ctx.moveTo(65,65);
    ctx.arc(60,65,5,0,Math.PI*2,true);  // 左眼
    ctx.moveTo(95,65);
    ctx.arc(90,65,5,0,Math.PI*2,true);  // 右眼
    ctx.stroke();
 }
}
```

以下是所要用到的函数：

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

第一步：生成路径的，用 **beginPath**()。本质上，路径是由很多子路径构成，这些子路径都是在一个列表中，所有的子路径（线、弧形、等等）构成图形。而每次这个方法调用之后，列表清空重置，然后我们就可以重新绘制新的图形。

beginPath()

新建一条路径，生成之后，图形绘制命令被指向到路径上生成路径。

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

确定是否需要移动笔触

moveTo 移动笔触

一个非常有用的函数，而这个函数实际上并不能画出任何东西，也是上面所描述的路径列表的一部分，这个函数就是 moveTo()。或者你可以想象一下在纸上作业，一支钢笔或者铅笔的笔尖从一个点到另一个点的移动过程。
moveTo(x, y)
将笔触移动到指定的坐标 x 以及 y 上。
当 canvas 初始化或者 beginPath()调用后，你通常会使用 moveTo()函数设置起点。我们也能够使用 moveTo()绘制一些不连续的路径。
第二：描边或填充路径区域来渲染图形

第三步就是调用函数指定绘制路径。

//可以绘制什么图形

直线， 矩形， 三角形， 圆弧，圆

```
//红色的矩形
ctx.fillStyle="#FF0000";
ctx.fillRect(0,0,150,75);
```

stroke()

通过线条来绘制图形轮廓。

fill()

通过填充路径的内容区域生成实心的图形。会自动调用 closePath 方法

就是闭合路径 **closePath**(),不是必需的。这个方法会通过绘制一条从当前点到开始点的直线来闭合图形。如果图形是已经闭合了的，即当前点为开始点，该函数什么也不做。

closePath()

closePath() 方法创建从当前点到开始点的路径。闭合路径之后图形绘制命令又重新指向到上下文中。

注意：当你调用 fill()函数时，所有没有闭合的形状都会自动闭合，所以你不需要调用 closePath()函数。但是调用 stroke()时不会自动闭合。

Canvas - 路径（画线）

面的例子绘制两个三角形，一个是填充的，另一个是描边的。

```
function draw() {
 var canvas = document.getElementById('canvas');
 var ctx = canvas.getContext('2d');

 // 填充三角形
ctx.beginPath();
ctx.strokeStyle = 'blue';
ctx.fillStyle="#FF6600";
ctx.moveTo(200,220);
ctx.lineTo(220,240);
ctx.lineTo(230,110);
ctx.fill();

 // 描边三角形
ctx.beginPath();
ctx.strokeStyle = 'red';
ctx.moveTo(225,225);
ctx.lineTo(235,245);
ctx.lineTo(245,225);
ctx.closePath();
ctx.stroke();

}
```

绘制文本

样式

font 当前我们用来绘制文本的样式. 这个字符串使用和 CSS font 属性相同的语法

textAlign : 文本对齐选项. 可选的值包括：start, end, left, right or center. 默认值是 start

textBaseline 基线对齐选项.

direction 文本方向。可能的值包括：ltr, rtl, inherit。默认值是 inherit

渲染文本

fillText(text, x, y [, maxWidth])

在指定的(x,y)位置填充指定的文本，绘制的最大宽度是可选的.

strokeText(text, x, y [, maxWidth])

在指定的(x,y)位置绘制文本边框，绘制的最大宽度是可选的.

文本用当前的填充方式被填充

画文本实例

```
function drawText() {
  var ctx = document.getElementById('canvas').getContext('2d');
   ctx.font = "48px serif";
   ctx.strokeText("Hello World",10,50);
   ctx.fillText("Hello world", 10, 50);
}
```

绘制矩形

我们回到矩形的绘制中。canvas 提供了三种方法绘制矩形：

fillRect(x, y, width, height)

绘制一个填充的矩形

strokeRect(x, y, width, height)

绘制一个矩形的边框

clearRect(x, y, width, height)

清除指定矩形区域，让清除部分完全透明。

上面提供的方法之中每一个都包含了相同的参数。x 与 y 指定了在 canvas 画布上所绘制的矩形的左上角（相对于原点）的坐标。width 和 height 设置矩形的尺寸。

矩形实例

```
//画一个填充的矩形 fillRect 和 矩形的边框 strokeRect
 function rect(){
     ctx.fillStyle="#ff6600";
      // 绘制一个填充的矩形
     ctx.fillRect(250,250,80,100);
     // 绘制一个矩形的边框
     ctx.strokeRect(320,120,80,100);
}
```

圆弧

绘制圆弧或者圆，我们使用 arc()方法。

arc 用法

arc(x, y, radius, startAngle, endAngle, anticlockwise)

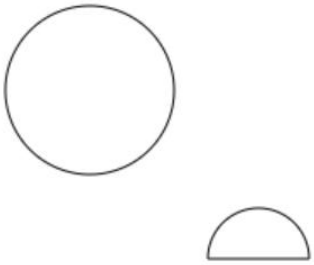
画一个以（x,y）为圆心的以 radius 为半径的圆弧（圆），从 startAngle 开始到 endAngle 结束，按照 anticlockwise 给定的方向（默认为顺时针）来生成。

方法有六个参数：
x,y 为绘制圆弧所在圆上的圆心坐标。
radius 为半径。
startAngle 以及 endAngle 参数用弧度定义了开始以及结束的弧度。
这些都是以 x 轴为基准。
参数 anticlockwise 为一个布尔值。为 true 时，是逆时针方向，否则顺时针方向。

注意：arc()函数中表示角的单位是弧度，不是角度。角度与弧度的 js 表达式：

弧度=(Math.PI/180)*角度。

实例方法



```
//各种圆弧
  function arcEx(){
          ctx.beginPath();
          ctx.translate(200,350);
           // ctx.scale(2,2);

           ctx.fillStyle="#000";
           ctx.arc(0,0,50,0,2*Math.PI,true);
          ctx.stroke();
          ctx.closePath();
          ctx.beginPath();

           ctx.arc(100,100,30,0,Math.PI,true);
           ctx.closePath();
           ctx.stroke();
    }
```

各种效果

调整坐标位置

translate(x, y)

translate 方法接受两个参数。*x* 是左右偏移量，*y* 是上下偏移量，

如果不使用 translate 方法，那么所有矩形都将被绘制在相同的位置（0,0）。translate 方法同时让我们可以任意放置这些图案，而不需要在 fillRect() 方法中手工调整坐标值，既好理解也方便使用

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

旋转 Rotating

它用于以原点为中心旋转 canvas

rotate(angle)

这个方法只接受一个参数：旋转的角度(angle)，它是顺时针方向的，以弧度为单位的值。

```
ctx.rotate(2* Math.PI/20);
```

```
var oCan = document.getElementById("canvas");
        var context = oCan.getContext("2d");
        context.translate(50,50);
        context.fillRect(10,10,60,60);
        // 前面的不会受影响

        context.rotate(30 * Math.PI / 180); //旋转画布 , 旋转完 后面所画的 图都会是旋转后的角度
        //后面绘制的内容会被影响
        context.strokeText("Hello World",100,0);
```

缩放

scale(x, y)

scale 方法可以缩放画布的水平和垂直的单位。两个参数都是实数，可以为负数，x 为水平缩放因子，y 为垂直缩放因子，如果比 1 小，会比缩放图形， 如果比 1 大会放大图形。默认值为 1， 为实际大小。

```
ctx.scale(2,2);
```

变形 Transforms

最后一个方法允许对变形矩阵直接修改。

transform(m11, m12, m21, m22, dx, dy)

这个方法是将当前的变形矩阵乘上一个基于自身参数的矩阵，在这里我们用下面的矩阵：

渐变 createLinearGradient

渐变可以填充在矩形, 圆形, 线条, 文本等等, 各种形状可以自己定义不同的颜色。

以下有两种不同的方式来设置 Canvas 渐变：

createLinearGradient(*x,y,x1,y1*) - 创建线性渐变

createRadialGradient(*x,y,r,x1,y1,r1*) - 创建一个径向/圆渐变

当我们使用渐变对象，必须使用两种或两种以上的停止颜色。

addColorStop()方法指定颜色停止，参数使用坐标来描述，可以是 0 至 1.

使用渐变，设置 fillStyle 或 strokeStyle 的值为 渐变，然后绘制形状，如矩形，文本，或一条线。

使用 createLinearGradient():

创建一个线性渐变。使用渐变填充矩形:

```
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d"); // 创建渐变
var grd=ctx.createLinearGradient(0,0,200,0);
grd.addColorStop(0,"red");
grd.addColorStop(1,"white"); // 填充渐变
ctx.fillStyle=grd;
ctx.fillRect(10,10,150,80);
```

预测量文本宽度

measureText()
将返回一个 TextMetrics 对象的宽度、所在像素，这些体现文本特性的属性

```
textwidth = ctx.measureText(text).width;
```

Canvas - 图像

把一幅图像放置到画布上, 使用以下方法:

引入图像到 canvas 里需要以下两步基本操作：

- 获得一个指向 HTMLImageElement 的对象或者另一个 canvas 元素的引用作为源，也可以通过提供一个 URL 的方式来使用图片
- 使用 drawImage()函数将图片绘制到画布上

HTMLImageElement
这些图片是由 Image()函数构造出来的，或者任何的<img>元素

HTMLVideoElement
用一个 HTML 的 <video>元素作为你的图片源，可以从视频中抓取当前帧作为一个图像

HTMLCanvasElement
可以使用另一个 <canvas> 元素作为你的图片源。

ImageBitmap
这是一个高性能的位图，可以低延迟地绘制，它可以从上述的所有源以及其它几种源中生成

由零开始创建图像

或者我们可以用脚本创建一个新的 HTMLImageElement 对象。要实现这个方法，我们可以使用很方便的 Image()构造函数。

当脚本执行后，图片开始装载。

若调用 drawImage 时，图片没装载完，那什么都不会发生（在一些旧的浏览器中可能会抛出异常）。因此你应该用 load 事件来保证不会在加载完毕之前使用这个图片：

```
function drawImg(){
    var img = new Image();   // 创建 img 元素
  img.src = 'myImage.png'; // 设置图片源地址
  img.onload = function(){
   // 执行 drawImage 语句
     ctx.drawImage(img,0,0);
```

```
  }
}
```

动画的基本步骤

setTimeout（setInterval）

动画原理

我们在浏览器中看到的图像是在以每秒 60 次的频率刷新的，由于刷新频率很高，因此你感觉不到它在刷新。而**动画本质就是要让人眼看到图像被刷新而引起变化的视觉效果，这个变化要以连贯的、平滑的方式进行过渡。** 那怎么样才能做到这种效果呢?

刷新频率为 60Hz 的屏幕每 16.7ms 刷新一次，我们在屏幕每次刷新前，将图像的位置向左移动一个像素，即 1px。这样一来，屏幕每次刷出来的图像位置都比前一个要差 1px，因此你会看到图像在移动；由于我们人眼的视觉停留效应，当前位置的图像停留在大脑的印象还没消失，紧接着图像又被移到了下一个位置，因此你才会看到图像在流畅的移动，这就是视觉效果上形成的动画。

setTimeout（setInterval）的缺点

理解了上面的概念以后，我们不难发现，setTimeout 其实就是通过设置一个间隔时间来不断的改变图像的位置，从而达到动画效果的。但我们会发现，利用 seTimeout 实现的动画在某些低端机上会出现卡顿、抖动的现象。 这种现象的产生有两个原因：

setTimeout 的执行时间并不是确定的。在 Javascript 中， setTimeout 任务被放进了异步队列中，只有当主线程上的任务执行完以后，才会去检查该队列里的任务是否需要开始执行，因此 setTimeout 的实际执行时间一般要比其设定的时间晚一些。

刷新频率受屏幕分辨率和屏幕尺寸的影响，因此不同设备的屏幕刷新频率可能会不同，而 setTimeout 只能设置一个固定的时间间隔，这个时间不一定和屏幕的刷新时间相同。

以上两种情况都会导致 setTimeout 的执行步调和屏幕的刷新步调不一致，从而引起丢帧现象。

所以，使用 window.setInterval()或者 window.setTimeout()制作的动画，有如下缺点：

1.他们都是通用的方法，并不是专为绘制动画而用

2.即使向其专递以毫秒为单位的参数值，它们也达不到毫秒级的准确性

3.没有对调用动画循环的机制优化

4.不考虑绘制动画的最佳时机，而只会一味地以某个大致的时间间隔调用动画循环。

requestAnimationFrame

你可以通过以下的步骤来画出一帧:

- **清空 canvas**
  除非接下来要画的内容会完全充满 canvas （例如背景图），否则你需要清空所有。最简单的做法就是用 clearRect 方法。
- **保存 canvas 状态 save()**
  如果你要改变一些会改变 canvas 状态的设置（样式，变形之类的），又要在每画一帧之时都是原始状态的话，你需要先保存一下。
- **绘制动画图形（animated shapes)**
  这一步才是重绘动画帧。
- **恢复 canvas 状态**
  如果已经保存了 canvas 的状态，可以先恢复它，然后重绘下一帧

window.requestAnimationFrame() 告诉浏览器——你希望执行一个动画，并且要求浏览器在下次重绘之前调用指定的回调函数更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

```
window.requestAnimationFrame(callback);
```

简单来说，requestAnimationFrame 方法用于通知浏览器重采样动画。

当 requestAnimationFrame(callback)被调用时不会执行 callback，而是会将元组<handle,callback>插入到动画帧请求回调函数列表末尾（其中元组的 callback 就是传入

requestAnimationFrame 的回调函数），并且返回 handle 值，该值为浏览器定义的、大于 0 的整数，唯一标识了该回调函数在列表中位置。

每个回调函数都有一个布尔标识 cancelled，该标识初始值为 false，并且对外不可见。

浏览器在执行“采样所有动画”的任务时会遍历动画帧请求回调函数列表，判断每个元组的 callback 的 cancelled，如果为 false，则执行 callback。

cancelAnimationFrame 方法用于取消先前安排的一个动画帧更新的请求。

当调用 cancelAnimationFrame(handle)时，浏览器会设置该 handle 指向的回调函数的 cancelled 为 true。

无论该回调函数是否在动画帧请求回调函数列表中，它的 cancelled 都会被设置为 true。

如果该 handle 没有指向任何回调函数，则调用 cancelAnimationFrame 不会发生任何事情。

当页面可见并且动画帧请求回调函数列表不为空时，浏览器会定期地加入一个“采样所有动画”的任务到 UI 线程的队列中。

这个 API 的调用很简单，如下所示：

```
var progress = 0;//回调函数 function render() {
    progress += 1; //修改图像的位置

    if (progress < 100) {
        //在动画没有结束前，递归渲染
         window.requestAnimationFrame(render);
    }
}
//第一帧渲染
window.requestAnimationFrame(render);
```

cancelAnimationFrame 的使用方法也很简单

```
function a(time) {
        console.log("animation");
        id = window.requestAnimationFrame(a);
    }
    a();
window.cancelAnimationFrame(id);
```

与 setTimeout 相比，requestAnimationFrame 最大的优势是**由系统来决定回调函数的执行时机。**具体一点讲，如果屏幕刷新率是 60Hz,那么回调函数就每 16.7ms 被执行一次，如果刷新率是 75Hz，那么这个时间间隔就变成了 1000/75=13.3ms，换句话说就是，requestAnimationFrame 的步伐跟着系统的刷新步伐走。**它能保证回调函数在屏幕每一次的刷新间隔中只被执行一次**，这样就不会引起丢帧现象，也不会导致动画出现卡顿的问题。

requestAnimationFrame 还有以下两个优势

- **CPU 节能**：使用 setTimeout 实现的动画，当页面被隐藏或最小化时，setTimeout 仍然在后台执行动画任务，由于此时页面处于不可见或不可用状态，刷新动画是没有意义的，完全是浪费 CPU 资源。而 requestAnimationFrame 则完全不同，当页面处理未激活的状态下，该页面的屏幕刷新任务也会被系统暂停，因此跟着系统步伐走的 requestAnimationFrame 也会停止渲染，当页面被激活时，动画就从上次停留的地方继续执行，有效节省了 CPU 开销。
- **函数节流**：在高频率事件(resize,scroll 等)中，为了防止在一个刷新间隔内发生多次函数执行，使用 requestAnimationFrame 可保证每个刷新间隔内，函数只被执行一次，这样既能保证流畅性，也能更好的节省函数执行的开销。一个刷新间隔内函数执行多次时没有意义的，因为显示器每 16.7ms 刷新一次，多次绘制并不会在屏幕上体现出来。

浏览器兼容性

由于 requestAnimationFrame 目前还存在兼容性问题，而且不同的浏览器还需要带不同的前缀。因此需要通过优雅降级的方式对 requestAnimationFrame 进行封装，优先使用高级特性，然后再根据不同浏览器的情况进行回退，直止只能使用 setTimeout 的情况。

所以，如果想要简单的兼容，可以这样子：

```
window.requestAnimFrame = (function(){
  return  window.requestAnimationFrame       ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame    ||
          function( callback ){
            window.setTimeout(callback, 1000 / 60);
          };
})();
```

简单的应用

现在分别使用 setInterval、setTimeout 和 requestAnimationFrame 这三个方法制作一个简单的进制度效果

setInterval

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    clearInterval(timer);
    myDiv.style.width = '0';
    timer = setInterval(function(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
            myDiv.innerHTML =     parseInt(myDiv.style.width)/5 + '%';
        }else{
            clearInterval(timer);
        }
    },16);
}
</script>
```

setTimeout

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    clearTimeout(timer);
    myDiv.style.width = '0';
    timer = setTimeout(function fn(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
```

```
        myDiv.innerHTML =     parseInt(myDiv.style.width)/5 + '%';
        timer = setTimeout(fn,16);
    }else{
        clearTimeout(timer);
    }
},16);
}
</script>
```

requestAnimationFrame

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    myDiv.style.width = '0';
    cancelAnimationFrame(timer);
    timer = requestAnimationFrame(function fn(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
            myDiv.innerHTML =     parseInt(myDiv.style.width)/5 + '%';
            timer = requestAnimationFrame(fn);
        }else{
            cancelAnimationFrame(timer);
        }
    });
}
</script>
```