第14章 HTML5 API

HTML5是用于创建网页的超文本标记语言的最新版本。目前最新的版本号为5.1，于2016年11月成为了W3C的推荐标准。HTML5给规范添加了大量的新功能。还超越了实际的标记语言，带来了很多相关的技术，比如CSS和JavaScript。我们在第10章介绍表单验证的时候，已经看到了一些新的表单控件以及校验表单的API。在本章，我们将继续来看一下HTML5中有哪些其他的API。

本章我们将学习如下内容：

- data-属性介绍
- Web Storage存储数据
- Geolocation API 获取位置信息
- Web Worker
- 插入音频和视频
- 使用canvas进行绘图
- HTML5中的拖放
- Notification API创建用户通知

14-1 data- 属性

data-属性是一种会被浏览器忽略的自定义属性，使用它可以将数据很轻松地嵌入到网页中。它们是页面私有的，所以不被外部服务使用，唯一的用途就是被JavaScript程序所用。这意味着data-属性非常适合添加用作钩子的数据，程序可以利用该钩子访问页面上特定元素的相关信息。

data-属性的属性名可以由开发者来决定，不过必须使用如下的格式：

- 以data-开头。
- 只包含小写字母，数字，连字符，点，冒号或者下划线。
- 包含一个可选的字符串值

示例如下：

```
data-powers = 'flight superSpeed'
data-rating = '5'
data-dropdown
data-user = 'XJ'
data-max-length = '32'
```

包含在属性中的信息可以用于识别特殊的元素。例如，所有带有一个`data-dropdown`属性的元素可以被识别为下拉菜单。该属性的值还可以被用于过滤不同的元素。例如，我们可以找到所有`data-rating`值为3的元素。

每个元素都有一个dataset属性可以用来访问它包含的data-属性。示例如下：

```
<body>
    <div id="test" data-power="high speed">
        this is a test
    </div>
    <script>
        let one = document.getElementById("test");
        let dataContent = one.dataset.power;
        console.log(dataContent);//high speed
    </script>
</body>
```

注意这里需要将前缀的data-去掉。要访问该属性，使用了dataset.powers，就好像powers是dataset的一个属性一样。如果data-属性的名称包含连字符，那么需要使用驼峰命名法，示例如下：

```
<body>
    <div id="test" data-superman-power="strong and speed">
        this is a test
    </div>
    <script>
        let one = document.getElementById("test");
        let dataContent = one.dataset.supermanPower;
        console.log(dataContent);//strong and speed
    </script>
</body>
```

如果遇到老的浏览器不支持data-属性，那么可以考虑使用DOM里面的`getAttribute()`方法来获取

```
let dataContent = one.getAttribute("data-power");
```

data-属性提供了一种将数据直接添加到HTML标记中的便捷方式，从而实现了更丰富的用户体验。

14-2 Web Storage

Web Storage(Web存储)提供了一种方式，可以让Web页面实现在客户端浏览器中意键值对的形式在本地保存数据。在了解HTML5的Web存储之前，我们先来看一下前面我们所介绍过的使用Cookies存储机制的优缺点。

14-2-1 Cookie存储机制的优缺点

在讲解BOM一章时，我们有介绍过Cookie。Cookie是HTML4中在客户端存储简单用户信息的一种方式，它使用文本来存储信息，当有应用程序使用Cookie时，服务器端就会发送Cookie到客户端，客户端浏览器将会保存该Cookie信息。下一次页面请求时，客户端浏览器就会把Cookie发送到服务器。Cookie最典型的应用是用来保存用户信息，用户设置和密码记忆等。

使用Cookie有其优点，也有其缺陷，其优点主要表现在以下几个方面。

- 简单易用
- 浏览器负责发送数据
- 浏览器自动管理不同站点的Cookie。

但是经过长期的使用，Cookie的缺点也渐渐暴露了出来，主要有以下几点。

- 因为使用的是简单的文本存储数据，所以Cookie的安全性很差。Cookie保存在客户端浏览器，很容易被黑客窃取。
- Cookie中存储的数据容量有限，其上限为4KB。
- 存储Cookie的数量有限，多数浏览器的上限为30或50个。
- 如果浏览器的安全配置为最高级别，那么Cookie则会失效。
- Cookie不适合大量数据的存储，因为Cookie会由每个对服务器的请求来传递，从而造成Cookie速度缓慢效率低下。

14-2-2 为什么要用 Web Storage

HTML5的Web Storage提供了两种在客户端存储数据的方法，
即 `localStorage` 和 `sessionStorage` 。

localStorage

localStorage是一种没有时间限制的数据存储方式，可以将数据保存在客户端的硬盘或其他存储器，存储时间可以是一天，两天，几周或几年。浏览器的关闭并不意味着数据也会随之消失。当再次打开浏览器时，我们依然可以访问这些数据。localStorage用于持久化的本地存储，除非主

动删除数据，否则数据是永远不会过期的。

sessionStorage

sessionStorage指的是针对一个session的数据存储，即将数据保存在session对象中。Web中的session指的是用户在浏览某个网站时，从进入网站到关闭浏览器所经过的这段时间，可以称为用户和浏览器进行交互的"会话时间"。session对象用来保存这个时间段内所有要保存的数据。当用户关闭浏览器后，这些数据会被删除。
sessionStorage用于本地存储一个会话(session)中的数据，这些数据只有在同一个会话中的页面才能访问，并且当会话结束后数据也随之销毁。因此sessionStorage不是一种持久化的本地存储，仅仅是会话级别的存储。

从上面的介绍我们可以看出，localStorage可以永久保存数据，而sessionStorage只能暂时保存数据，这是两者之间的重要区别，在具体使用时应该注意。

Web Storage存储机制比传统的Cookie更加强大，弥补了Cookie诸多缺点，主要在以下几个方面做了加强：

- Web Storage提供了易于使用的API接口，只需要设置键值对即可使用，简单方便
- 在存储容量方面可以根据用户分配的磁盘配额进行存储，能够在每个用户域存储5MB～10MB的内容。用户不仅可以存储session，还可以存储许多信息，如设置偏好，本地化的数据和离线数据等。
- Web Storage还提供了使用JavaScript编程的接口，开发者可以使用JavaScript客户端脚本实现许多以前只能在服务器端才能完成的工作。

14-2-3 使用 Web Storage

如果浏览器支持Web Storage API，那么window对象中就会有localStorage和sessionStorage这个属性。这个属性是一个原生对象，带有很多用于存储数据的属性和方法。信息是以键值对的形式存储，值只能是字符串。

存储示例：我们可以通过`setItem()`方法来存储一个值，然后通过`getItem()`方法来获取值

在1.html里面设置localStorage的值

```
<body>
    <script>
        localStorage.setItem("name","xiejie");
        console.log("信息已保存!");
    </script>
</body>
```

接下来在2.html里面来获取已经设置了的值

```
<body>
    <script>
        let name = localStorage.getItem("name");
        console.log(`姓名为${name}`);//姓名为xiejie
    </script>
</body>
```

这里，我们存储的值是存放于本地的，所以没有时间的限制，即使我们关闭浏览器，设置了的值也依然存在。除了上面使用 `getItem()` 和 `setItem()` 来存取数据以外，我们还可以直接进行如下的存值和取值操作，就好像是给一个属性赋值一样，如下：

在1.html里面设置localStorage的值

```
<body>
    <script>
        localStorage.setItem("name","xiejie");
        localStorage.age = 18;
        console.log("信息已保存!");
    </script>
</body>
```

接下来在2.html里面来获取已经设置了的值

```
<body>
    <script>
        let name = localStorage.getItem("name");
        let age = localStorage.age;
        console.log(`姓名为${name},年龄为${age}`);
        //姓名为xiejie,年龄为18
    </script>
</body>
```

如果要删除本地存储中的一个条目，可以使用 `removeItem()` 方法或者直接使用 `delete` 运算符，如下：

```
localStorage.removeItem("name");
//或者
delete localStorage.name
```

如果要彻底删除本地存储中所有的东西，可以使用`clear()`方法，如下：

```
localStorage.clear()
```

14-2-4 storage事件

每次将一个值存储到本地存储时，就会触发一个storage事件。由事件监听器发送给回调函数的事件对象有几个自动填充的属性如下：

- key：告诉我们被修改的条目的键
- newValue：告诉我们被修改后的新值
- oldValue：告诉我们修改前的值
- storageArea：告诉我们是本地存储还是会话存储
- url：被改变键的文档地址

需要注意的是，这个事件只在同一域下的任何窗口或者标签上触发，并且只在被存储的条目改变时触发。

示例如下：这里我们需要打开服务器进行演示，本地文件无法触发storage事件
1.html代码如下：在该页面下我们设置了两个localStorage并存储了两个值

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <script>
        localStorage.name = "xiejie";
        localStorage.age = 20;
        console.log("信息已经设置!");
    </script>
</body>
</html>
```

2.html代码如下：在该页面中我们安装了一个storage的事件监听器，安装之后只要是同一域下面的其他storage值发生改变，该页面下面的storage事件就会被触发

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <script>
        let name = localStorage.getItem("name");
        let age = localStorage.age;
        console.log(`姓名为${name},年龄为${age}`);
        window.addEventListener("storage",(e)=>{
            console.log(`${e.key}从${e.oldValue}修改为${e.newValue}`);
            console.log(`这是一个${e.storageArea}存储`);
            console.log(`被改变的url为${e.url}`);
        },true);
    </script>
</body>
</html>
```

接下来我们来改变1.html里面localStorage.name的值

```
localStorage.name = "谢杰";
```

之后我们就会看到安装在2.html页面里面的storage事件就会被触发，显示出如下效果：

```
name从xiejie修改为谢杰
这是一个[object Storage]存储
被改变的url为http://localhost/1.html
```

最后，需要说明一下的就是虽然Web Storage只能存储字符串看起来限制很大，但是通过使用JSON，我们可以在本地存储中存储任何的JSON对象，如下：

```
let xiejie = {"name":"xiejie","age":18};
localStorage.setItem("info",JSON.stringify(xiejie));
```

这段代码会将xiejie这个对象存储为以info作为键的一个JSON字符串，当我们要将其取出来使用的时候，只需要再使用`JSON.parse`方法将其转换回JSON对象即可

```
xiejie = JSON.parse(localStorage.info);
```

14-3 Geolocation

HTML5 Geolocation API是HTML5新增的地理位置应用程序接口，它提供了一个可以准确感知浏览器用户当前位置的方法。如果浏览器支持，且设备具有定位功能，就能够直接使用这组API来获取当前位置的信息。该Geolocation API可以应用于移动设备中的地理定位。允许用户在Web应用程序中共享位置信息，使其能够享受位置感知的服务。

HTML5的Geolocation API的使用方法也非常简单。请求一个位置信息，如果用户同意，浏览器就会返回相应的位置信息，该位置信息是通过支持HTML5地理定位功能的底层设备来提供的，例如笔记本电脑或者手机。位置信息由纬度，经度坐标和一些其他元数据组成。有了这些位置信息我们就可以构建位置感知类的应用程序。

14-3-1 为什么要学习Geolocation

设想这样一个场景：有一个Web应用程序，它可以向用户提供附近不远处某商店运动鞋的打折优惠信息。使用HTML5 Geolocation API，就可以请求其他用户共享他们的位置，如果他们同意，应用程序就可以提供相关信息，告诉用户去附近哪家商店可以挑选到打折的鞋子。

HTML5 Geolocation技术另一个应用场景就是构建计算行走(跑步)路程的应用程序。我们可以在开始跑步时通过手机浏览器启动应用程序的记录功能。在用户移动过程中，应用程序会记录已跑过的距离，还可以把跑步过程对应的坐标显示在地图上，甚至可以显示出海拔信息。如果用户正在和其他选手一起参加跑步比赛，那么应用程序还可以显示其对手的位置。

诸如此类的应用，现在已经很多很多了。

14-3-2 位置信息的表示方式

位置信息主要由一对纬度和经度坐标组成，例如：

```
Latitude:39.17222,Longitude:-120.13778
```

在这里，纬度(距离赤道以北或以南的数值表示)是39.17222，经度(距离英国格林威治以东或以西的数值表示)是120.13778，经纬度坐标的表示可以使用以下两种方式

- 十进制格式，如120.13778
- DMS角度格式，如39度20分

HTML5 Geolocation API返回坐标的格式为十进制格式。除了经纬度的坐标，HTML5 Geolocation还提供位置坐标的准确度，并提供其他一些元数据，具体情况取决于浏览器所在的

硬件设备。这些元数据包括海拔，海拔准确度，行驶方向和速度等。如果这些元数据不存在则返回null。

14-3-3 使用Geolocation API

`geolocation`是`navigator`对象的一个属性。这个属性有一个方法`getCurrentPosition()`，该方法会返回一个position对象给指定的回调函数。

传给youAreHere()函数的position对象有一个coords属性，该属性又有latitude和longitude属性，二者一起给出了设备的坐标。这些坐标随后可以与其他应用程序或者Web服务(比如地图服务)结合使用，以获取用户的准确位置。示例如下：

```
<body>
    <script>
        let youAreHere = function(position){
            console.log(`纬度为：${position.coords.latitude}`);
            console.log(`经度为：${position.coords.longitude}`);
        }
        navigator.geolocation.getCurrentPosition(youAreHere);
    </script>
</body>
```

position对象还有几个其他的属性，可以用来查找设备的位置和移动信息：

- position.speed属性：返回设备的地面速度(米/秒)
- position.altitude属性：返回设备的海拔估算值
- position.heading属性：返回设备正在移动的方向
- position.timestamp属性：返回位置信息被记录的时间

position对象还有计算测量精度的属性。例如position.accuracy属性以米为单位返回经度和纬度的精确度。返回值越低，测量结果的精确度就越高，position.altitudeAccuracy属性就是这种情况，将返回以米为单位的海拔属性的精度。

此外，`geolocation`对象还有一个`watchPosition()`方法，该方法在每次设备的位置更新时，会调用一个回调函数。这个方法返回一个ID，这个ID可以用于引用被观察的位置：

```
let id = navigator.geolocation.watchPosition(youAreHere);
```

`clearWatch()`方法可以用于阻止调用回调，使用作为观察用的id来作为参数

```
navigator.geolocation.clearWatch(id);
```

HTML5 Geolocation API为网站或者应用程序添加基于位置的信息提供了有用的接口，但是大部分接口都需要谷歌地图来提供服务支持。由于国内特殊的环境，所以更多时候是利用百度地图提供的API来获取位置信息。所以Geolocation作为一个了解即可。

14-4 Web Worker

在运行大型，复杂的JavaScript脚本的时候经常会出现浏览器假死的现象，那么能不能让这些代码在后台运行，或者让JavaScript函数在多个进程中同时运行呢？HTML5的Web Worker正是为了解决这个问题而出现的。HTML5的Web Worker可以让Web应用程序具备后台的处理能力。它支持多线程处理功能，因此可以充分利用多核CPU带来的优势，将耗时长的任务分配给HTML5的Web Worker运行，从而避免了有时页面反应迟缓，甚至假死的现象。

14-4-1 Web Worker介绍

1. Web Worker概述

在Web应用程序中，Web Worker是一项后台处理技术。在此之前，JavaScript创建的Web应用程序中，因为所有的处理都是在单线程内执行的，所以如果脚本所需运行时间太长，程序界面就会长时间处于停止状态，甚至当等待时间超出一定的限度，浏览器就会提示脚本运行时间过长需要中断正在执行的处理。

为了解决这个问题，HTML5新增加了一个Web Worker API。使用这个API，用户可以很容易的创建在后台运行的线程，这个线程被称之为Worker。如果将可能耗费较长时间的处理交给后台来执行，则对用户在前台页面中执行的操作没有影响。

Web Worker的优点如下：

- 加载一个JavaScript文件，进行大量的复杂计算，而不挂起主进程，并通过 `postMessage` 和 `onMessage` 进行通行。
- 可以在Worker中通过 `importScripts(url)` 方法来加载JavaScript脚本文件。
- 可以使用 `setTimeout()` ， `clearTimeout()` ， `setInterval()` 和 `clearInterval()` 。
- 可以使用 `XMLHttpRequest` 进行异步请求。
- 可以访问 `navigator` 的部分属性。
- 可以使用JavaScript核心对象。

除了上述的优点，Web Worker本身也是具有一定局限性的，具体如下：

- 不能跨域加载JavaScript
- Worker内代码不能访问DOM
- 各个浏览器对Worker的实现还没有完全支持，不是每个浏览器都支持所有的新特性。
- 使用Web Worker加载数据没有JSONP和Ajax加载数据高效。

2. 浏览器支持情况

各个浏览器对Web Worker的支持情况不太一样，代表列出了目前主流浏览器对Web Worker的支持情况：

浏览器	描述
IE	不支持
Firefox	3.5及以上的版本支持
Opera	10.6及以上的版本支持
Chrome	3.0及以上的版本支持
Safari	4.0及以上的版本支持

在调用Web Worker API之前，应该确认一下当前的浏览器是否支持Web Worker。如果不支持，可以提供一些备用信息，提醒用户使用最新的浏览器。例如，可以使用下面的代码来检测浏览器的支持情况：

```
<body>
    <script>
        if(window.Worker)
        {
            console.log("您的浏览器支持Web Worker");
        }
    </script>
</body>
```

如果所使用的浏览器支持Web Worker，则Worker将会作为window对象的一个属性存在，所以我们只需要检测window.Worker是否存在即可判断支持情况。

3. Web Worker成员

在开始使用Web Worker之前，我们先来看一下使用Worker时会遇到的属性和方法，如下：

- self：self关键值用来表示本线程范围内的作用域。
- postMessage()：向创建线程的源窗口发送信息。
- onMessage：获取接收消息的事件句柄。
- importScripts(urls)：导入其他JavaScript脚本文件。参数为该脚本文件的URL地址，可以导入多个脚本文件。导入的脚本文件必须与使用该线程文件的页面在同一个域中，并在同一个端口中。

```
importScripts("worker1.js","worker2.js","worker3.js");
```

- navigator对象：有`appName`，`platform`，`userAgent`和`appVersion`属性，它们可以用来标识浏览器。
- sessionStorage/localStorage：在线程中可以使用Web Storage。
- XMLHttpRequest：在线程中可以嵌套线程。
- setTimeout()/setInterval()：在线程中可以实现定时处理。
- close：结束本线程。
- eval()，isNaN()，ecape()等：可以使用JavaScript的核心函数。
- object：可以创建和使用本地对象。
- WebSockets：可以使用Web Sockets API向服务器发送和接收消息。

14-4-2 Web Worker的使用

Web Worker的使用方法非常简单，只需要创建一个Web Worker对象，并传入希望执行的JavaScript文件即可。之后在页面中设置一个事件监听器，用来监听由Web Worker对象发来的消息以及错误信息。如果想要在页面与Web Worker之间建立通信，数据需要通过`postMessage()`函数来进行传递。

首先我们开始第一步，创建Web Worker。步骤十分简单，只要在Worker类的构造器中，将需要在后台线程中执行的脚本文件的URL地址作为参数传入，就可以创建Worker对象，如下：

```
let worker = new Worker("./worker.js");
```

> 注意：在后台线程中是不能访问页面或者窗口对象的，此时如果在后台线程的脚本文件中使用window
> 或者document对象，则会引发错误。

这里传入的JavaScript的URL可以是相对或者绝对路径，只要是相同的协议，主机和端口即可。如果想获取Worker进程的返回值，可以通过`onmessage`属性来绑定一个事件处理程序。如下：

```
<body>
    <script>
       let worker = new Worker("./worker.js");
       worker.onmessage = function(){
           console.log("the message is back!");
       }
    </script>
</body>
```

这里第一行代码用来创建和运行Worker进程，第2行设置Worker的`onmessage`属性用来绑定事

件处理程序，当Worker的`postMessage()`方法被调用时，这个被绑定的程序就会被触发。

使用Worker对象的`postMessage()`方法可以给后台线程发送消息。发送的消息可以是文本数据，也可以是任何JavaScript对象，需要通过JSON对象的`stringify`方法将其转换成文本数据。

```
worker.postMessage(message);
```

通过获取Worker对象的onmessage事件句柄以及Worker对象的`postMessage()`方法就可以实现线程内部消息的接收和发送。

Web Worker不能自行终止，但是能够被启用它们的页面所终止。调用`terminate()`函数可以终止后台进程。被终止的Web Workers将不再响应任何消息或者执行任何其他运算。终止之后，Worker不能被重新启动，但是可以使用同样的URL创建一个新的Worker。

最后，我们来书写一个完整的例子来演示Worker的工作机制，如下：

首先，是我们的1.html页面代码

```
<body>
    <p>计数：<output id="result"></output></p>
    <button onclick="startWorker()">开始工作</button>
    <button onclick="stopWorker()">停止工作</button>
    <p><strong>注意：</strong>IE9及其以下版本浏览器不支持 Web Workers.</p>
    <script>
        let w;
        //开始Worker的代码
        function startWorker()
        {
            //判断浏览器是否支持
            if (typeof (Worker) !== "undefined")
            {
                if (typeof (w) == "undefined")
                {
                    w = new Worker("1.js");
                }
                w.onmessage = function (event)
                {
                    document.getElementById("result").innerHTML = event.data
;
                };
            }
            else {
                document.getElementById("result").innerHTML = "你的浏览器不支持
";
```

```
            }
        }
        //停止Worker的代码
        function stopWorker()
        {
            w.terminate();
            w = undefined;
        }
    </script>
</body>
```

接下来我们运行在后台的Worker的代码，如下：

```
let i = 0;
function timedCount()
{
    i = i + 1;
    //每次得到的结果都通过postMessage方法返回给前台
    postMessage(i);
    setTimeout("timedCount()", 500);
}
timedCount();
```

效果：

计数：8

开始工作 停止工作

注意： Internet Explorer 9 及更早 IE 版本浏览器不支持 Web Workers.

上面的代码，当用户点击"开始工作"时，会创建一个Web Worker在后台进行计数。每次计的数都会通过postMessage方法返回给前台。当用户点击"停止工作"时，则会terminate()方法来终止Web Worker的运行。

14-5 多媒体

在HTML5之前，要在浏览器中显示音频和视频是非常困难的，经常我们需要使用类似于Flash这一类的插件才能够实现。不过，从HTML5开始引入了`<audio>`和`video`这两个标签可以实现在网页中快速插入音频和视频，同时我们还可以使用JavaScript来控制多媒体的播放。

14-5-1 引入多媒体

引入多媒体的方式如下：

```
<body>
    <audio src="./test.mp3" controls>
        您的浏览器不支持audio标签
    </audio>
</body>
```

上面的代码我们就在页面里面引入了一个音频标签。对于支持该标签的浏览器，会正常显示出音频的控制面板(注意必须要书写controls属性)，而对于不支持该标签的浏览器，则会显示出中间的那句"您的浏览器不支持audio标签"

- src:多媒体文件的路径
- controls:显示控制面板

为了做到浏览器兼容，我们可以将多媒体文件的路径写在`<source>`标签里面。然后可以对应对应多个多媒体文件。浏览器会自动选择第一个可以识别的格式，示例如下：

```
<body>
    <audio controls>
        <source src="./test.mp3">
        <source src="./test.ogg">
        您的浏览器不支持audio标签
    </audio>
</body>
```

如果要插入视频，那么只需要将上面的`<audio>`标签换成`<video>`标签即可。

14-5-2 相关属性和方法

- muted属性 用来设置当前媒体是否开静音，如果设置为true代表开启静音，false为不开启

```
<body>
    <audio controls muted=true>
        <source src="./test.mp3">
        <source src="./test.ogg">
        您的浏览器不支持audio标签
    </audio>
</body>
```

- autobuffer属性 设置为true后将实现自动缓冲

```
<body>
    <audio controls autobuffer=true>
        <source src="./test.mp3">
        <source src="./test.ogg">
        您的浏览器不支持audio标签
    </audio>
</body>
```

- autoplay属性 设置为自动播放

```
<body>
    <audio controls autoplay=true>
        <source src="./test.mp3">
        <source src="./test.ogg">
        您的浏览器不支持audio标签
    </audio>
</body>
```

- loop属性 设置后将会重复播放

```
<body>
    <audio controls loop=true>
        <source src="./test.mp3">
        <source src="./test.ogg">
        您的浏览器不支持audio标签
    </audio>
</body>
```

- poster属性 设置视频的封面，没有播放时可以显示一张图片(注意此属性只有`video`标签拥有，`audio`无此属性)

```
<body>
    <video controls poster="./1.jpg">
        <source src="./test.mp4">
        <source src="./test.ogv">
        <source src="./test.swf">
        您的浏览器不支持video标签
    </video>
</body>
```

- width和height属性 用于设置视频的宽度和高度

我们可以通过DOM的方法之一来引用到页面里面的`<audio>`或者`<video>`标签，如下：

```
const video = document.getElementsByTagName("video")[0];
```

引用成功之后我们就可以在此元素上面对多媒体进行控制，例如上面我们所介绍的属性都是可以通过在引用后的元素上面来进行设置的，如下：

```
video.muted = true;//设置为静音
video.loop = true;//设置循环播放
```

还可以在引用了元素的基础上进行一些其他的设置，如下：

- volune属性 用于设置音量

```
video.volume = 0.9;
```

- currentTime属性 用于获取多媒体当前所播放的具体时间

```
video.currentTime += 10;//视频播放快进10秒
```

- playbackRate属性 该属性用于设置快进或者快退速度的数字值。值为1时表示以正常速度播放

```
video.playbackRate = 8;// 以8被速度快进
```

- duration属性 用于查看该多媒体的持续时间，也就是时长为多久

```
video.duration
```

音频和视频也有很多可以触发的事件，包括：

- `play` 事件，用于开始播放多媒体
- `pause` 事件，用于暂停多媒体
- `volumechange` 事件，用于调整多媒体的音量
- `loadedmetadata` 事件，在多媒体的所有元数据被加载时会触发该事件

这些事件可以让我们响应用户和视频的所有的交互。例如，如下的事件监听器可以检测用户是否暂停了视频的播放：

```
video.addEventLister('pause',()=>{
    console.log('视频已经暂停');
},false);
```

当然，除了上面所介绍的属性和方法以外，HTML5中还提供了其他的API来让开发者对多媒体文件进行控制。完整的文档可以参见如下地址：
https://www.w3.org/2010/05/video/mediaevents.html

14-6 canvas画图

14-6-1 认识canvas元素

在HTML5中引入了`canvas`，`canvas`元素是网页上一个矩形的单元，可以用JavaScript在上面绘画。控制其每一个像素。`canvas`拥有多种绘制路径、矩形、圆形、字符以及添加图像的方法。
`canvas`主要应用于如下的领域：

- 游戏:canvas在基于Web的图像显示方面比Flash更加立体、更加精巧，canvas游戏在流畅度和跨平台方面更牛。
- 可视化数据(数据图表话)，比如: 百度的`echart`、`d3.js`、`three.js`
- banner广告:Flash曾经辉煌的时代，智能手机还未曾出现。现在以及未来的智能机时代，HTML5技术能够在banner广告上发挥巨大作用，用canvas实现动态的广告效果再合适不过。
- 未来
  模拟器:无论从视觉效果还是核心功能方面来说，模拟器产品可以完全由JavaScript来实现。
  远程计算机控制:canvas可以让开发者更好实现基于Web的数据传输，构建一个完美的可视化控制界面。
  图形编辑器:Photoshop图形编辑器将能够100%基于Web实现。

1. 添加canvas元素

要添加canvas的方法很简单，就是书写一个canvas标签。注意这是一个双标签，默认什么都不会显示，如果想要看到画布，可以添加一个border

```
<body>
    <canvas width="400" height="300" style="border:1px solid">
    </canvas>
</body>
```

2. 如何绘制图像

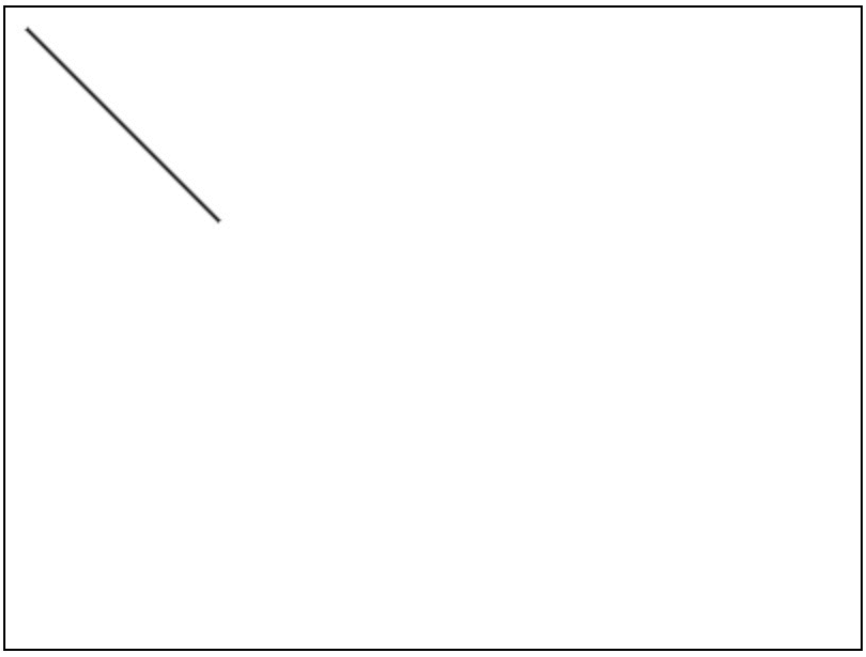
canvas本身不能进行绘制图形，绘制图形的工作都是交给JavaScript来完成的。绘制图形的步骤大致分为如下4个步骤：

1.添加canvas元素
2.在JS中获取canvas元素
3.获取到上下文，创建context对象
画布上下文是一个包含用于在画布上画画的所有方法的对象
4.使用JS来进行绘制

示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid"></canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        //使用JS来进行绘制
        context.beginPath();//开始路径
        context.moveTo(10,10);
        context.lineTo(100,100);
        context.closePath();//闭合路径
        context.stroke();
    </script>
</body>
```

效果：

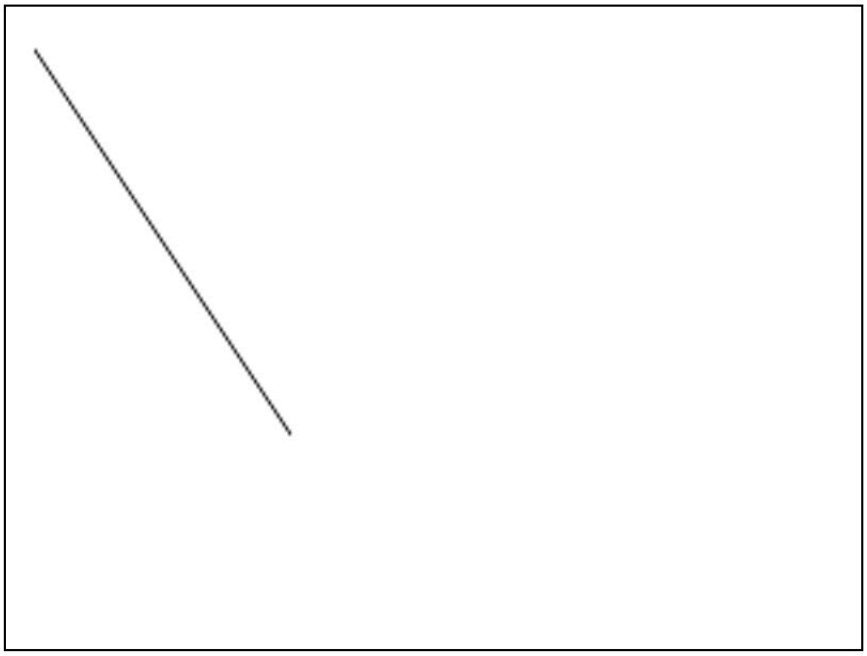


这里需要提一个关于Canvas设置宽高的注意事项。在Canvas中默认有一个宽高，值为300*150。如果我们在 `<style>` 里面再设置Canvas的宽高的话，此时会有一个拉伸效果，如

下：

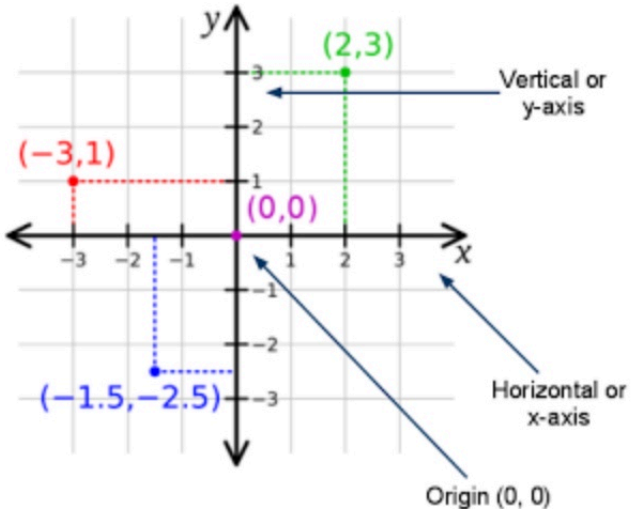
```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        canvas{
            width: 400px;
            height: 300px;
            border: 1px solid;
        }
    </style>
</head>

<body>
    <!-- 添加canvas元素 -->
    <canvas></canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        //使用JS来进行绘制
        context.beginPath();//开始路径
        context.moveTo(10, 10);
        context.lineTo(100, 100);
        context.closePath();//闭合路径
        context.stroke();
    </script>
</body>
```

效果：canvas的宽高同样是400*300，但是由于我们是在`<style>`里面设置的宽高，此时会对canvas进行一个拉伸，如下图



3. 关于坐标

canvas里面坐标的计算和平时我们在数学里面接触的x轴，y轴的计算方式一样，如下图：



4. 查看浏览器是否支持canvas

并不是所有的浏览器都支持canvas，我们可以通过下面这个最简单的方式来检测浏览器是否支持canvas。对于不支持canvas的浏览器，会将里面的话直接输出出来，如下：

```
<canvas width="400" height="300" style="border:1px solid">
    您的浏览器不支持canvas元素，请升级您的浏览器
</canvas>
```

还有一种方法，就是利用JS里面检测getContext属性是否存在，如果存在就是支持，不存在就是不支持。示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        if(canvas.getContext)
        {
            console.log("支持");
        }
        else{
            console.log("不支持");
        }
    </script>
</body>
```

14-6-2 绘制简单图形

1. 绘制直线

绘制直线用到的方法有3个：`moveTo()`、`lineTo()`、`stoke()`

`moveTo()`:相当于将画笔移动到某一个坐标

`lineTo()`:勾勒一条线，但是这只是草稿

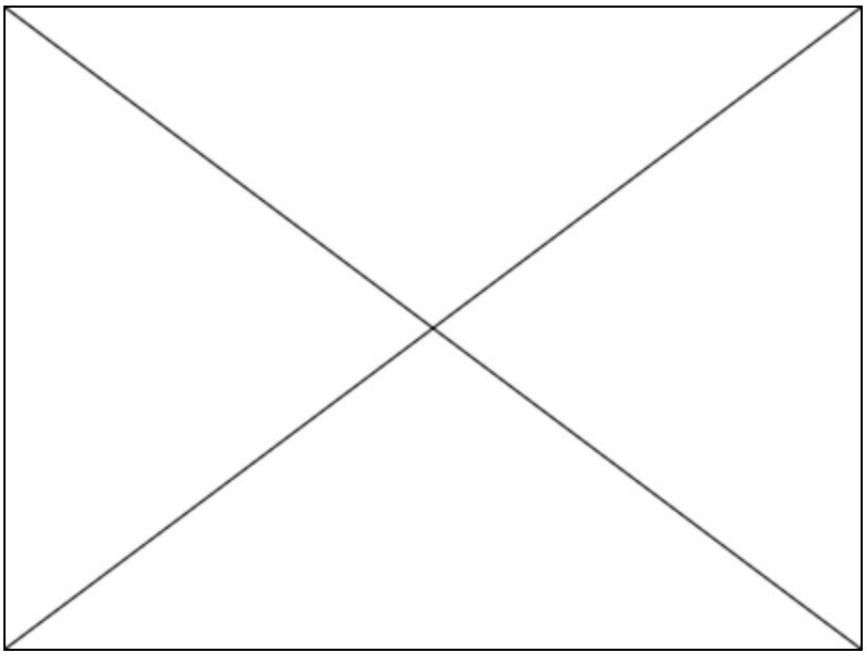
`stoke()`:根据勾勒的线描边

具体示例：画出两条对角线

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.moveTo(0,0);
        context.lineTo(400,300);
        context.moveTo(400,0);
        context.lineTo(0,300);
        context.stroke();
```

```
    </script>
</body>
```

效果：

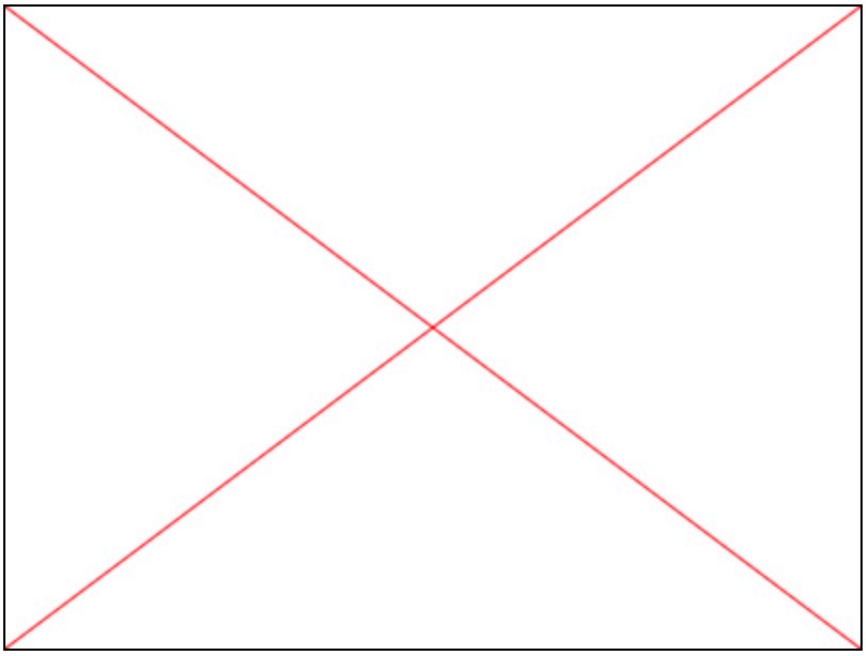


2. 绘制矩形

在绘制矩形之前，我们先来看一下在canvas中如何设置颜色。通过storkeStyle属性可以指定勾勒图形时的颜色，而fillStyle则是用于指定填充图形时的颜色。颜色的取值和CSS里面一样。示例如下：我们在绘制上面的对角线之前改变一下颜色：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.moveTo(0,0);
        context.lineTo(400,300);
        context.moveTo(400,0);
        context.lineTo(0,300);
        context.strokeStyle = "#ff0000";//设置勾勒颜色
        context.stroke();
    </script>
</body>
```

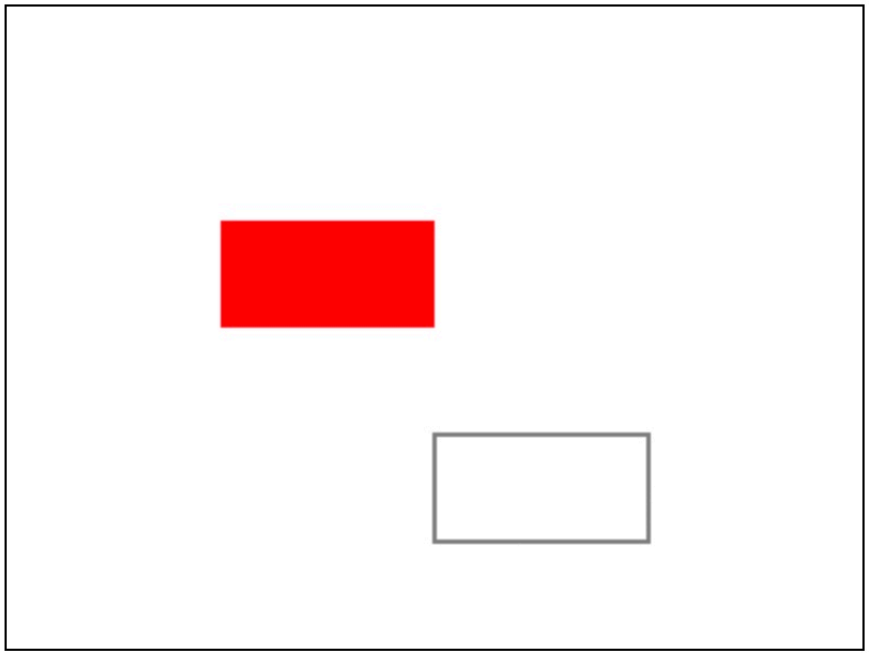
效果：



接下来我们来绘制矩形。在canvas里面绘制矩形的方法有两个：`fillRect()`和`strokeRect()`。这两个方法所接受的参数值一样分别为x坐标，y坐标，矩形长和矩形宽，区别在于前面的方法是填充矩形，后面的是空心矩形

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.fillStyle = "#ff0000";//设置填充颜色
        context.fillRect(100,100,100,50);//绘制填充矩形
        context.strokeRect(200,200,100,50);//绘制空心矩形
    </script>
</body>
```

效果：



3. 绘制圆形

使用arc()方法可以在画布上绘制一个圆形，该方法接收6个参数，分别如下表所示：

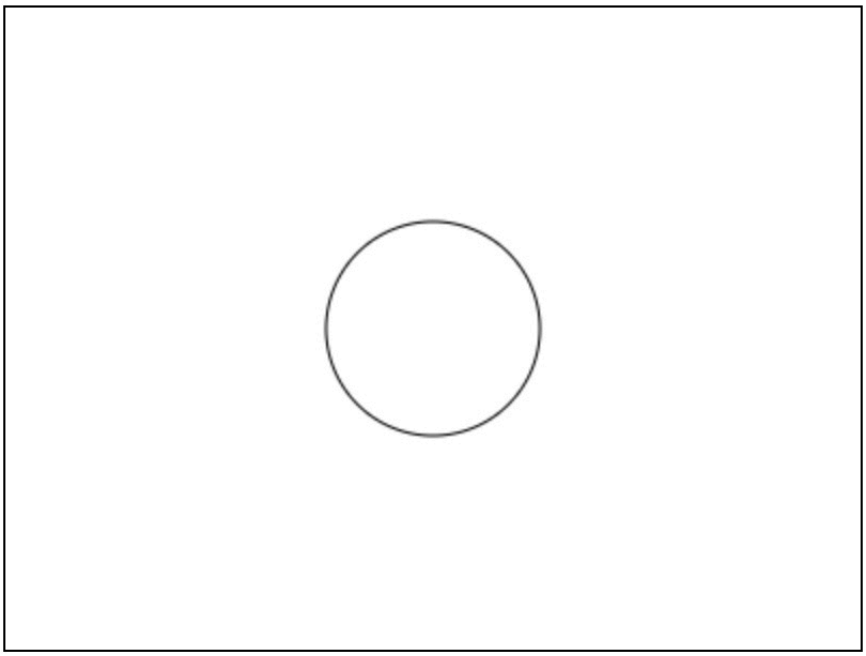
参数	描述
x	圆的中心的x坐标
y	圆的中心的y坐标
r	圆的半径
sAngle	起始角度，以弧度计算，3点钟位置是0度
eAngle	结束角度，以弧度计算
counterclockwise	可选，规定应该是顺时针绘制还是逆时针绘制。false=顺时针，true=逆时针

角度转弧度的公式为：`π/180×角度`
所以如果我们要绘制圆形，那么起始角度为0，结束角度为360度，也就是Math.PI*2就表示整圆，如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
```

```
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.arc(200,150,50,0,Math.PI*2,false);
        context.stroke();
    </script>
</body>
```

效果：

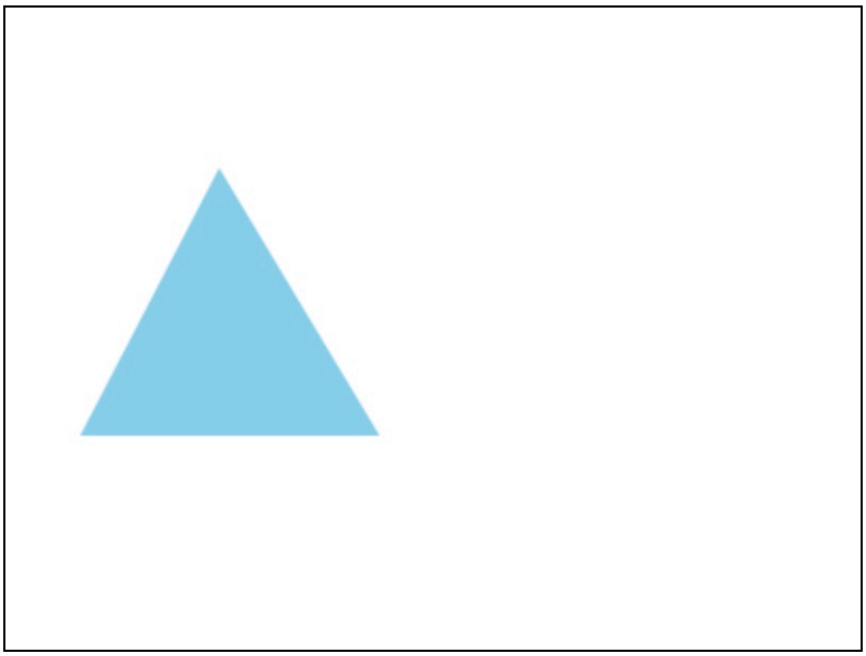


4. 绘制三角形

绘制三角形，其实就是绘制线条，所以我们用绘制线条的方式就可以绘制出一个三角形，如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.moveTo(100,75);
        context.lineTo(175,200);
        context.lineTo(35,200);
        context.fillStyle = "skyblue";
        context.fill();//填充方法会自动闭合路径
    </script>
</body>
```

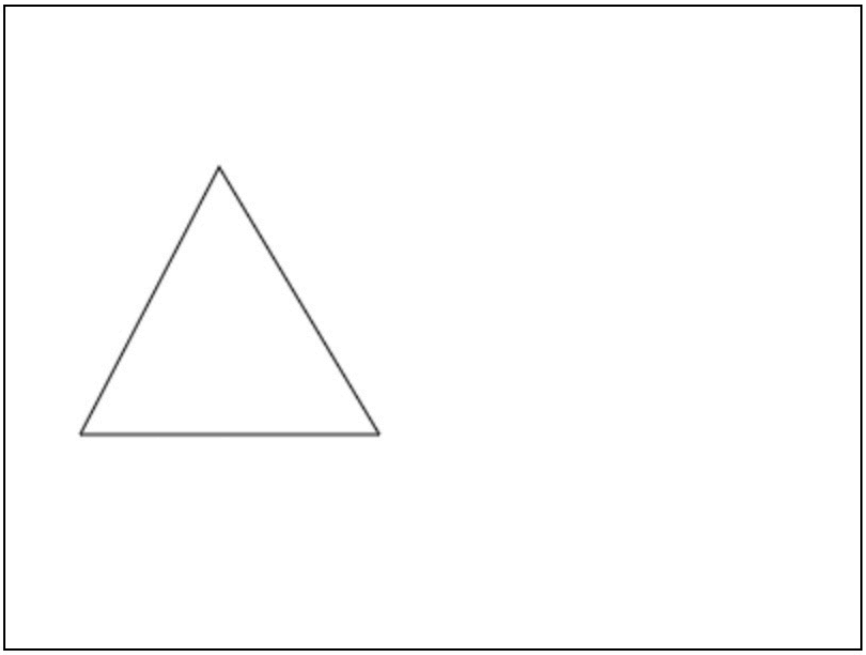
效果：



如果是使用的`stroke()`方法绘制空心三角形，则需要提前使用`closePath()`方法将路径闭合一下

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.moveTo(100,75);
        context.lineTo(175,200);
        context.lineTo(35,200);
        context.closePath();//闭合路径
        context.stroke();
    </script>
</body>
```

效果：



5. 绘制曲线

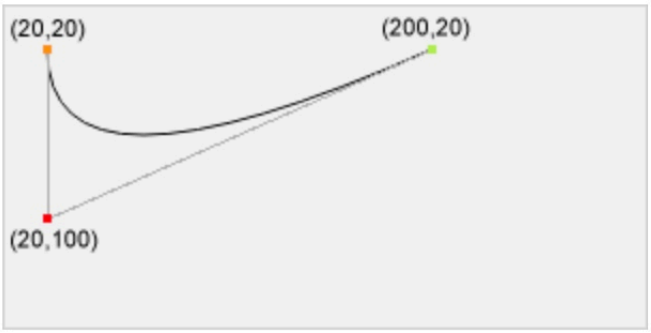
相对于绘制直线，矩形，圆形等简单图形而言，绘制曲线比较具有挑战性，但是一旦掌握了其原理，就能创建出许多复杂的图形。贝塞尔曲线在计算机图形学中的作用至关重要，其应用也非常广泛，如在一些数学软件，矢量绘图软件和三维动画软件中，经常会见到贝塞尔曲线，主要用于数值分析领域或产品设计和动画制作领域。这里我们将介绍如何在canvas中绘制贝塞尔曲线，包括二次方曲线和三次方曲线。

其实主要就是对应的两个方法。绘制二次方贝塞尔曲线的语法如下：

```
context.quadraticCurveTo(cp1x,cp1y,x,y)
```

`quadraticCurveTo()`方法用于绘制二次贝塞尔曲线。二次贝塞尔曲线需要两个点。第一个点是用于二次贝塞尔计算中的控制点，第二个点是曲线的结束点。也就是说cp1x和cp1y是控制点的坐标，x和y是终点的坐标。

曲线的开始点是当前路径中最后一个点。如果路径不存在，那么需要使用`beginPath()`和`moveTo()`方法来定义开始点。如下图：

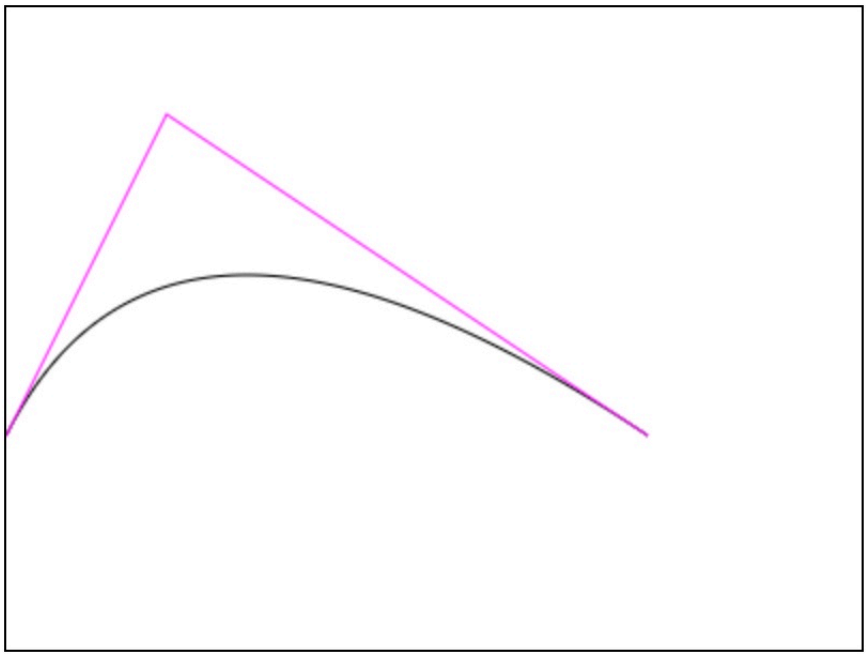


- 开始点： moveTo(**20,20**)
- 控制点： quadraticCurveTo(**20,100**,200,20)
- 结束点： quadraticCurveTo(20,100,**200,20**)

下面是该方法的一个示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.strokeStyle = "dark";
        context.beginPath();
        context.moveTo(0,200); // 这是起点位置
        context.quadraticCurveTo(75,50,300,200); // 第一个点是控制点 第二个是终点
        context.stroke();
        // 下面绘制的直线用于表示上面曲线的控制点和控制线，控制点坐标即两直线的交点(75,5
0)
        context.strokeStyle = "#ff00ff";
        context.beginPath();
        context.moveTo(75,50);
        context.lineTo(0,200);
        context.moveTo(75,50);
        context.lineTo(300,200);
        context.stroke();
    </script>
</body>
```

效果如下：



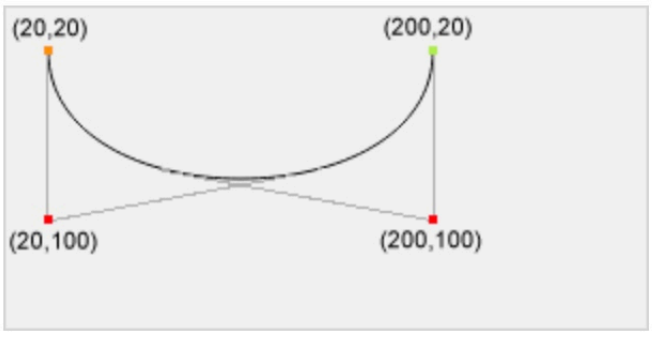
在上面的示例中，我们通过 `context.quadraticCurveTo` 绘制出来的曲线即二次方贝塞尔曲线，两条直线为控制线，两直线的交点即曲线的控制点。

绘制三次方贝塞尔曲线的语法如下：

```
context.bezierCurveTo(cpl1x,cp2y,cp2x,cp2y,x,y);
```

`bezierCurveTo()` 方法用于绘制三次贝塞尔曲线。三次贝塞尔曲线需要三个点。前两个点是用于三次贝塞尔计算中的控制点，第三个点是曲线的结束点。换句话说，这里的参数cp1x和cp1y代表的是第一个控制点的坐标，cp2x和cp2y代表的是第二个控制点的坐标，x和y是终点坐标。

曲线的开始点是当前路径中最后一个点。如果路径不存在，那么需要使用 `beginPath()` 和 `moveTo()` 方法来定义开始点。如下图所示：

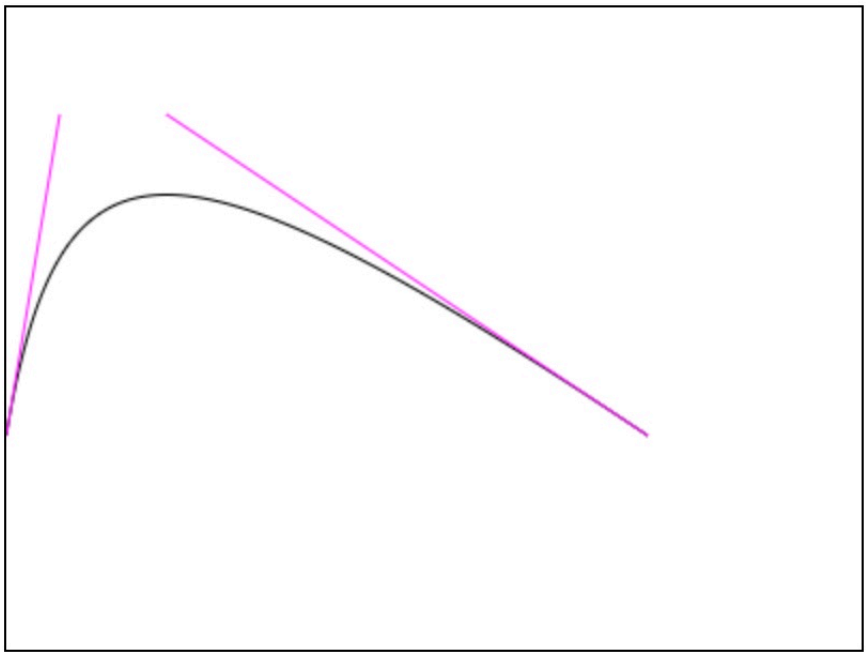


- 开始点： moveTo(**20,20**)
  - 控制点 1： bezierCurveTo(**20,100**,200,100,200,20)
    - 控制点 2： bezierCurveTo(20,100,**200,100**,200,20)
      - 结束点： bezierCurveTo(20,100,200,100,**200,20**)

下面是该方法的一个示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.strokeStyle = "dark";
        context.beginPath();
        context.moveTo(0,200);
        context.bezierCurveTo(25,50,75,50,300,200);
        context.stroke();
        // 下面绘制的直线用于表示上面曲线的控制点和控制线，控制点坐标即两直线的交点(75,5
0)
        context.strokeStyle = "#ff00ff";
        context.beginPath();
        context.moveTo(25,50);
        context.lineTo(0,200);
        context.moveTo(75,50);
        context.lineTo(300,200);
        context.stroke();
    </script>
</body>
```

效果：



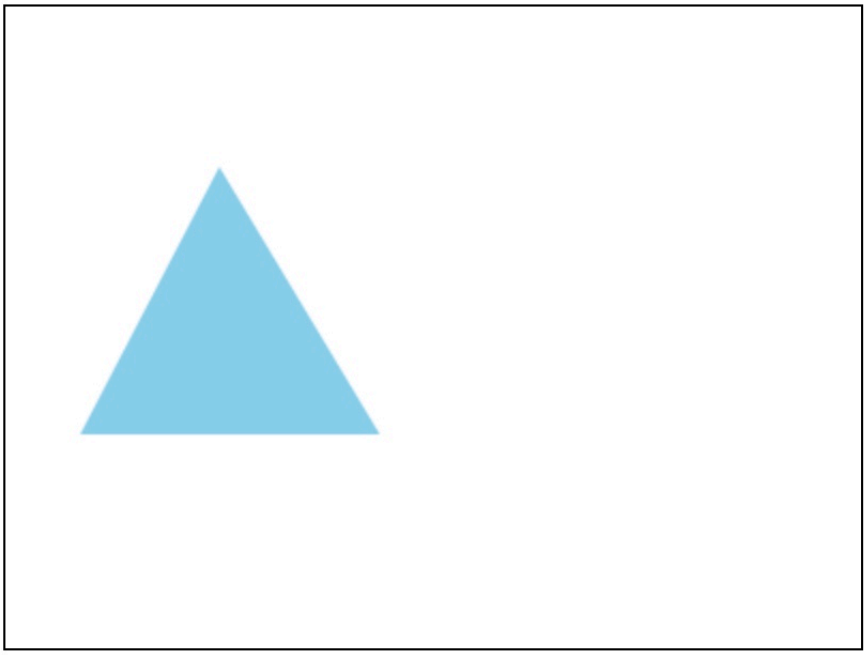
在上面的示例中，我们通过`context.bezierCurveTo`方法绘制出来的曲线即三次方贝塞尔曲线，两条直线为控制线，两直线上方的端点即为曲线的控制点。

6. 清空画布

使用 `clearRect()` 方法可以清除某个矩形范围内的图形，语法如下：
`clearRect(x坐标，y坐标，矩形长，矩形宽)`
具体示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <div>
        <button onclick="clearTest()">清除</button>
    </div>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let clearTest = function(){
            context.clearRect(0,0,400,300);
        }
        context.moveTo(100,75);
        context.lineTo(175,200);
        context.lineTo(35,200);
        context.fillStyle = "skyblue";
        context.fill();
    </script>
</body>
```

效果：点击清除按钮后画布会被清空



清除

14-6-3 图像的变换

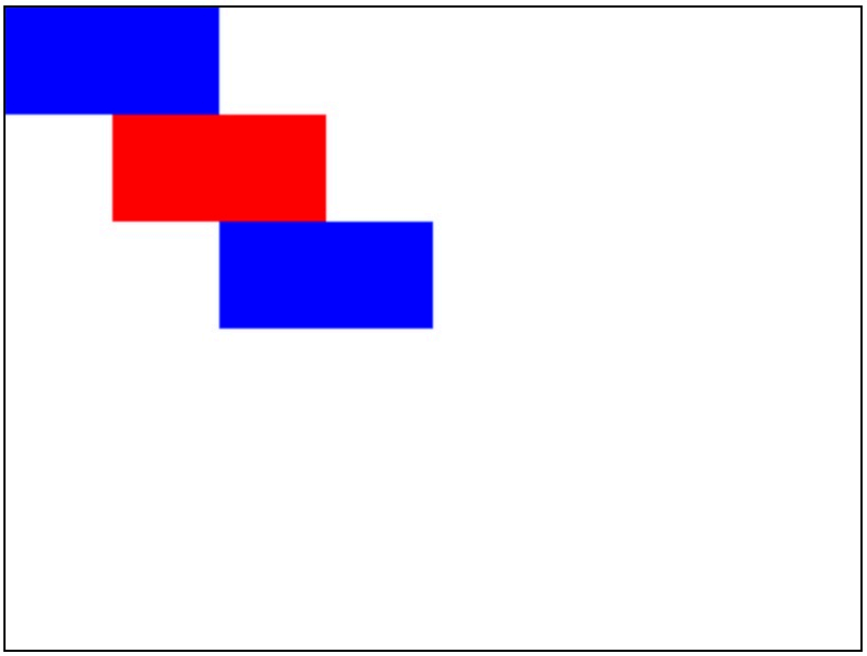
1. 保存和恢复状态

在介绍图像的变换之前，我们先来看一下在canvas里面如何保存和恢复canvas的状态。在canvas里面可以使用 `save()` 和 `restore()` 方法来保存和恢复canvas的状态。注意，这里的状态是指的一些设置信息，例如当前的颜色设置，应用变形等，恢复也是恢复一些设置。已经画好了的图像是不会有影响的。

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.fillStyle = "blue";
        context.fillRect(0,0,100,50);
        context.save();//保存的是填充颜色的设置信息
        context.fillStyle = "red";
        context.fillRect(50,50,100,50);
        context.restore();//恢复填充颜色的设置信息
        context.fillRect(100,100,100,50);
    </script>
```

```
</body>
```

效果：

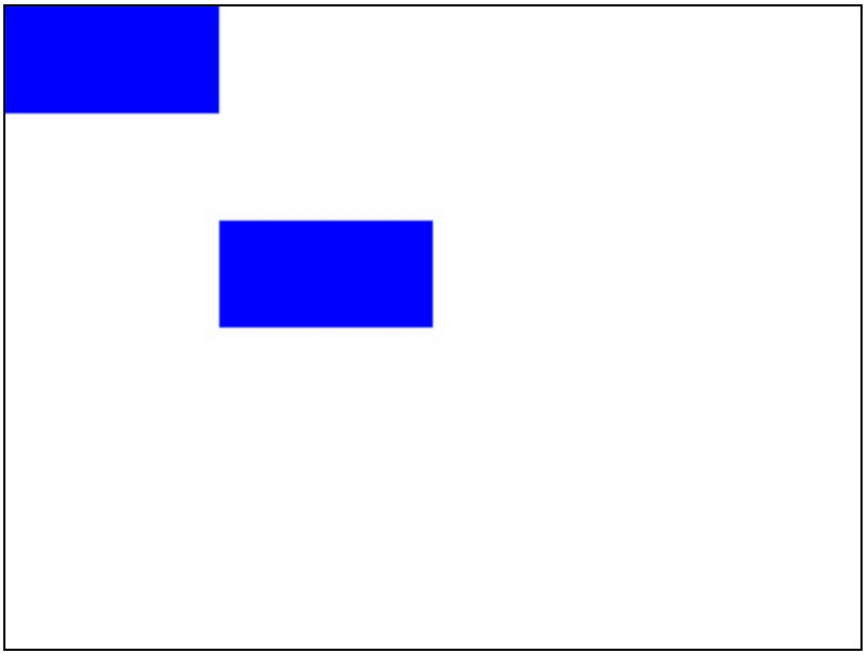


2. 移动坐标空间

在canvas里面画布的坐标空间默认是以画布左上角的(0,0)为原点，但是我们可以通过`translate()`方法来移动坐标空间，示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.fillStyle = "blue";
        context.fillRect(0,0,100,50);
        context.translate(100,100);//移动坐标原点至100x100区域
        context.fillRect(0,0,100,50);
    </script>
</body>
```

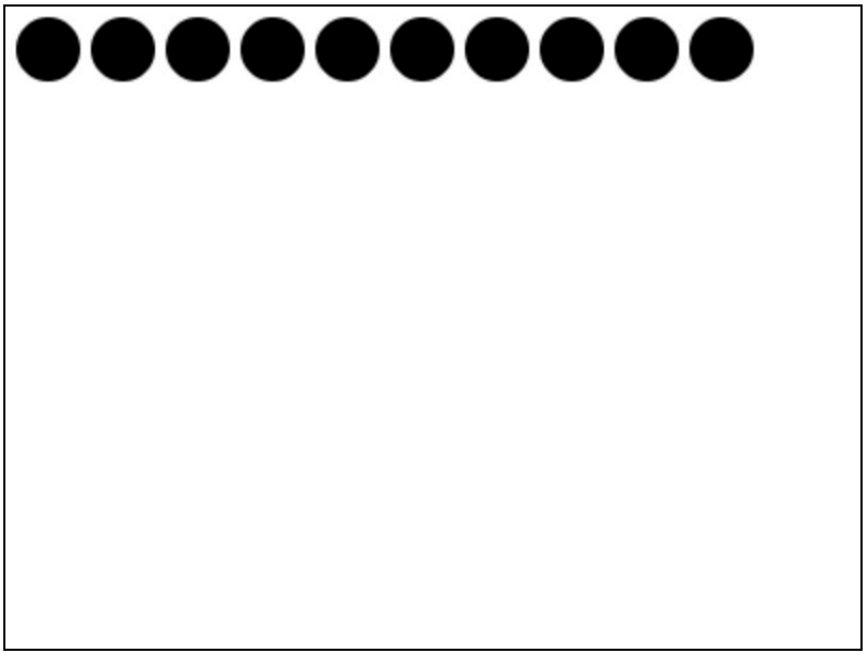
效果：



这里我们可以利用该方法在画布上绘制10个小球

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let i = 1;
        while(i<=10)
        {
            context.beginPath();
            context.arc(20,20,15,0,Math.PI*2,false);
            context.fill();
            context.closePath();
            context.translate(35,0);
            i++;
        }
    </script>
</body>
```

效果：



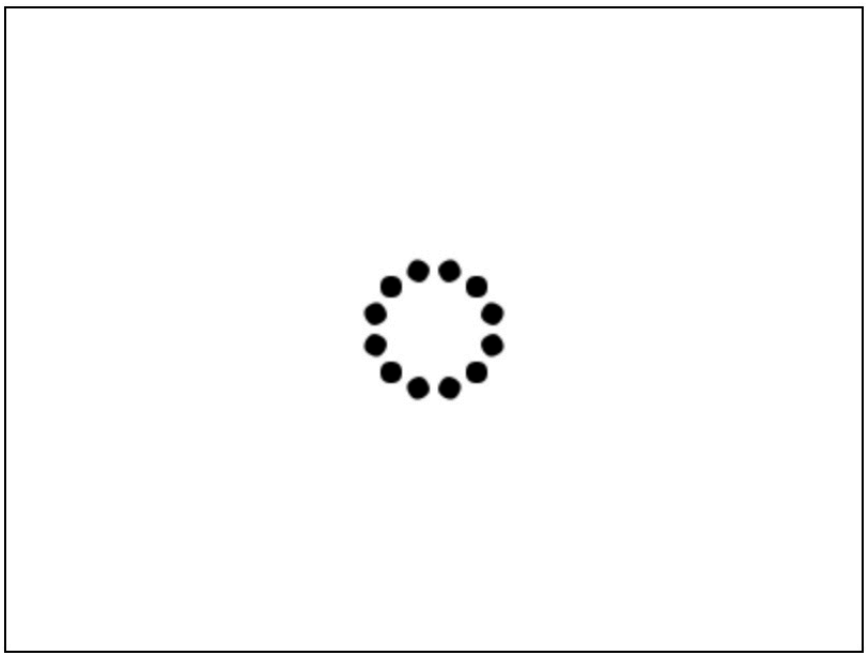
这里需要注意一定要书写 `beginPath()` 和 `closePath()` 来开启和闭合路径，否则会一直在一个路径上面进行绘制。

3. 旋转坐标空间

除了移动坐标空间，还可以旋转坐标空间，通过 `rotate()` 方法。旋转的单位也是以弧度为单位，所以需要将角度换算为弧度。

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let i = 1;
        context.translate(200,150);//先将画布的中心点移到中间
        while(i<=12)
        {
            context.beginPath();
            context.arc(20,20,5,0,Math.PI*2,false);
            context.fill();
            context.closePath();
            context.rotate(Math.PI*30/180);//每次旋转30度
            i++;
        }
    </script>
</body>
```

效果：



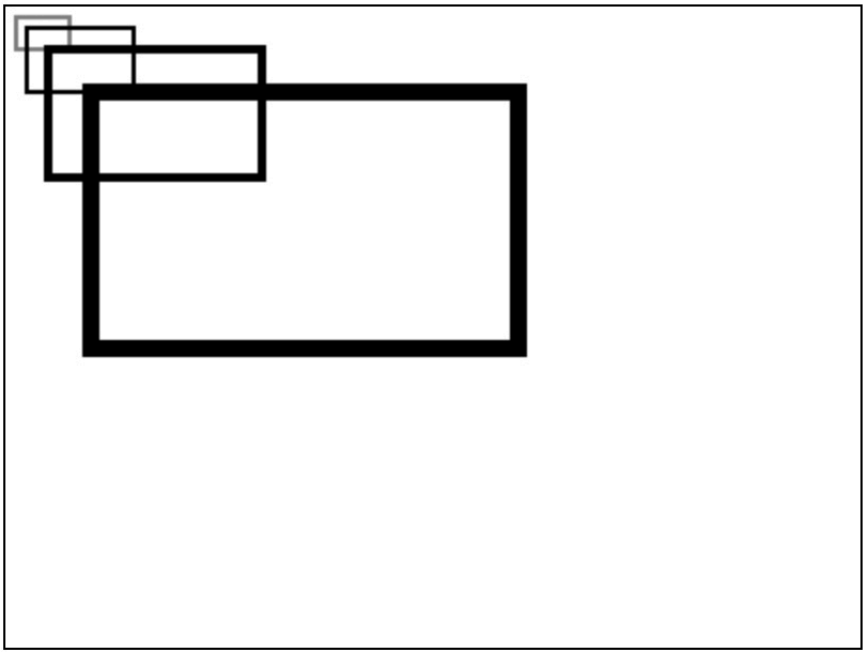
4. 缩放图形

语法为：`context.scale(x,y)`，所代表的含义为小于1为缩小，大于1为放大。x为水平方向的缩放倍数，y为竖直方向的缩放倍数。

具体示例：绘制一个矩形，放大到200%，再次绘制矩形，放大到200%，重复4次

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let i = 1;
        while(i<=4)
        {
            context.strokeRect(5,5,25,15);
            context.scale(2,2);
            i++;
        }
    </script>
</body>
```

效果：



14-6-4 图像的组合和裁切

当两个或两个以上的图形存在重叠区域时，默认情况下一个图形画在前一个图像之上。通过指定图像 `globalCompositeOperation` 属性的值可以改变图形的绘制顺序或绘制方式， `globalAlpha` 可以指定图形的透明度。
`globalCompositeOperation` 属性设置或返回如何将一个源(新的)图像绘制到目标(已有)的图像上。

- 源图像为打算放置到画布上的绘图。
- 目标图像为已经放置在画布上的绘图。

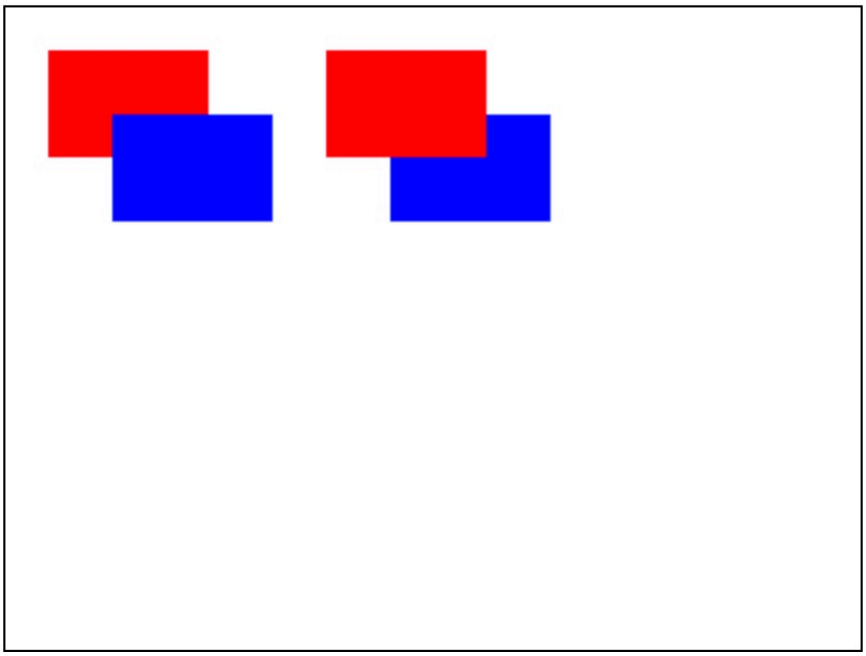
其中 `ctx.globalCompositeOperation = 'source-over'` 为默认设置

示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.fillStyle = "red";//设置填充颜色为红色
        context.fillRect(20,20,75,50);//画出红色矩形
```

```
        context.fillStyle = "blue";//设置填充颜色为蓝色
        context.fillRect(50,50,75,50);//画出蓝色矩形
        context.fillStyle = "red";//重新将填充颜色改为红色
        context.fillRect(150,20,75,50);//画出红色矩形
        context.fillStyle = "blue";//将填充颜色修改为蓝色
        //设置在源图像的上方显示目标图像
        context.globalCompositeOperation = "destination-over";
        context.fillRect(180,50,75,50);//画出蓝色矩形
    </script>
</body>
```

效果：



关于`globalCompositeOperation`其他的取值具体如下表：

值	描述
source-over	默认。在目标图像上显示源图像。
source-atop	在目标图像顶部显示源图像。源图像位于目标图像之外的部分是不可见的。
source-in	在目标图像中显示源图像。只有目标图像内的源图像部分会显示，目标图像是透明的。
source-out	在目标图像之外显示源图像。只会显示目标图像之外源图像部分，目标图像是透明的。
destination-over	在源图像上方显示目标图像。

destination-atop	在源图像顶部显示目标图像。源图像之外的目标图像部分不会被显示。
destination-in	在源图像中显示目标图像。只有源图像内的目标图像部分会被显示，源图像是透明的。
destination-out	在源图像外显示目标图像。只有源图像外的目标图像部分会被显示，源图像是透明的。
lighter	显示源图像 + 目标图像。
copy	显示源图像。忽略目标图像。
xor	使用异或操作对源图像与目标图像进行组合。

14-6-5 颜色和样式选项

1. 应用不同线条

通过下面的属性值，可以为线条应用不同的线型，分别为线条的粗细，端点样式，两线段连接处样式和绘制交点方式

- lineWidth = value
- lineCap = type
- lineJoin = type
- miterLimit = value

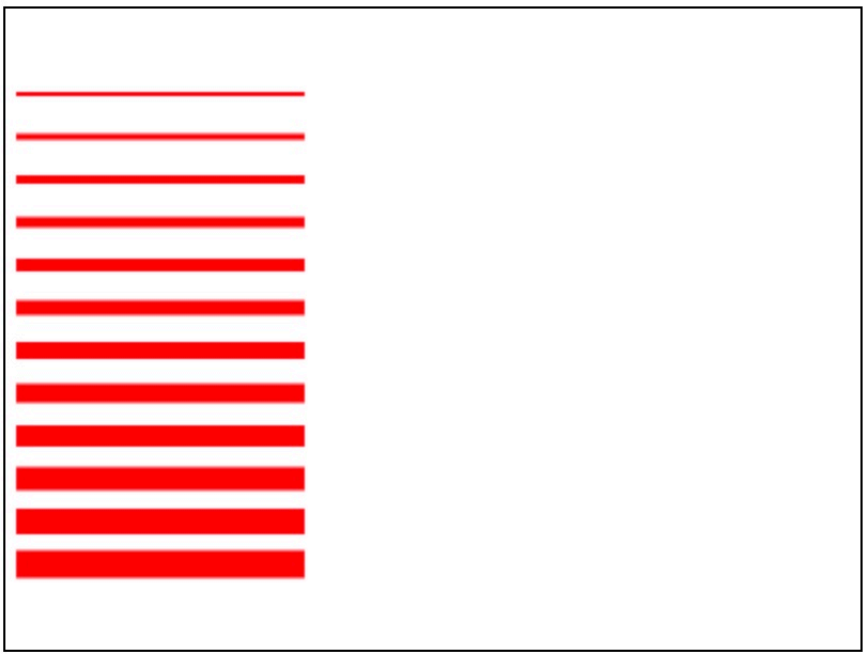
line-Width = value

设置线条的粗细，默认值为1.0
具体示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let i = 1;
        for(let j=0;j<=11;j++)
```

```
        {
            context.strokeStyle = "red";
            i++;
            context.lineWidth = i;//设置画笔的宽度为i，i每次会自增1
            context.beginPath();
            context.moveTo(5,i*20);
            context.lineTo(140,i*20);
            context.stroke();
            context.closePath();
        }
    </script>
</body>
```

效果：



lineCap

设置端点样式，值有3种，分别是butt，round和square，如下表：

值	描述
butt	默认。向线条的每个末端添加平直的边缘
round	向线条的每个末端添加圆形线帽
square	向线条的每个末端添加正方形线帽

具体示例：

```
<body>
    <!-- 添加canvas元素 -->
```

```
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let lineCap = ["butt","round","square"];
        //绘制参考线
        context.strokeStyle = "red";
        context.beginPath();
        //绘制左参考线
        context.moveTo(30,10);
        context.lineTo(30,150);
        //绘制右参考线
        context.moveTo(180,10);
        context.lineTo(180,150);
        context.stroke();
        //开始绘制3条直线
        context.strokeStyle = "blue";
        context.lineWidth = 20;
        for(let i=0;i<lineCap.length;i++)
        {
            context.lineCap = lineCap[i];
            context.beginPath();
            context.moveTo(30,30+i*50);
            context.lineTo(180,30+i*50);
            context.stroke();
        }
    </script>
</body>
```

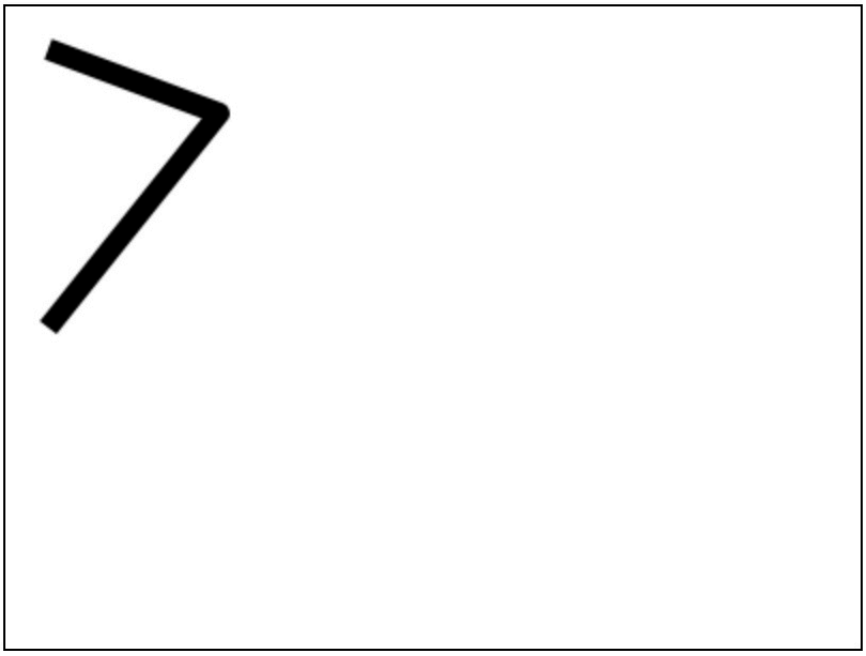
效果：

lineJoin

设置连接处样式，有3个属性值可选，分别是round，bevel和miter，默认值为miter
具体示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.beginPath();
        context.lineWidth = 10;
        context.lineJoin = "round";
        context.moveTo(20,20);
        context.lineTo(100,50);
        context.lineTo(20,150);
        context.stroke();
    </script>
</body>
```

效果：



miterLimit

用于设置或返回最大斜接长度。
斜接长度指的是在两条线交汇处内角和外角之间的距离。

图解 1　　图解 2　　图解 3

注意：只有当 lineJoin 属性为 "miter" 时，miterLimit 才有效。
边角的角度越小，斜接长度就会越大。
为了避免斜接长度过长，我们可以使用 miterLimit 属性。
如果斜接长度超过 miterLimit 的值，边角会以 lineJoin 的 "bevel" 类型来显示（图解 3）：

图解 1　　图解 2　　图解 3

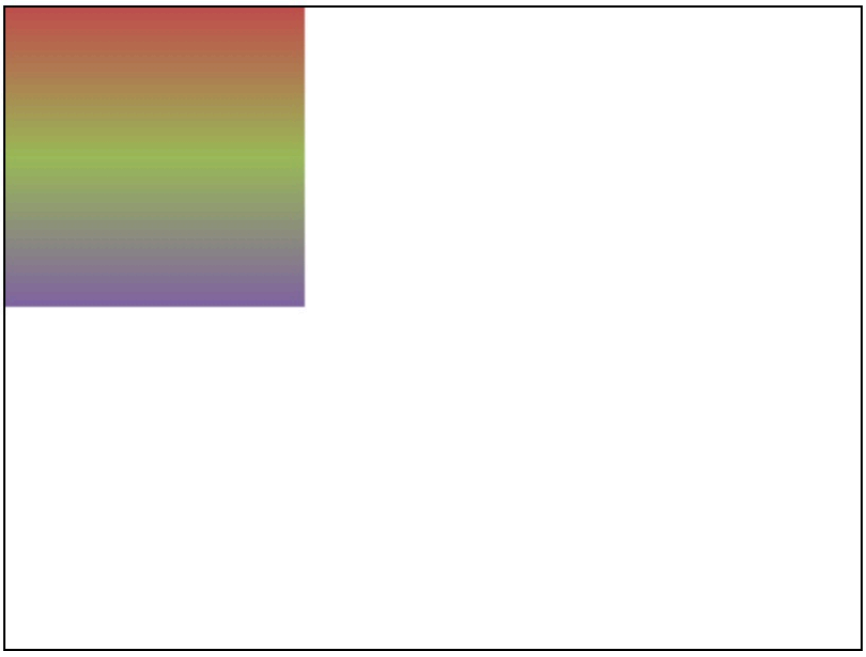
2. 绘制线性渐变

语法：

`context.createLinearGradient(x1,y1,x2,y2)`:x1,y1代表起点。x2,y2代表终点
`addColorStop(position,color)`:position为颜色位置，取值范围0-1，color为具体颜色
具体示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.beginPath();
        //指定渐变区域
        let grad = context.createLinearGradient(0,0,0,140);
        //添加几个渐变色
        grad.addColorStop(0,'rgb(192,80,77)');
        grad.addColorStop(0.5,'rgb(155,187,89)');
        grad.addColorStop(1,'rgb(128,100,162)');
        //将这个渐变设置为填充颜色
        context.fillStyle = grad;
        //绘制矩形
        context.rect(0,0,140,140);
        context.fill();
    </script>
</body>
```

效果：



如果要设置水平方向的渐变，只需要修改终点坐标即可

```
let grad = context.createLinearGradient(0,0,140,0);
```

倾斜的线形渐变

```
let grad = context.createLinearGradient(0,0,140,140);
```

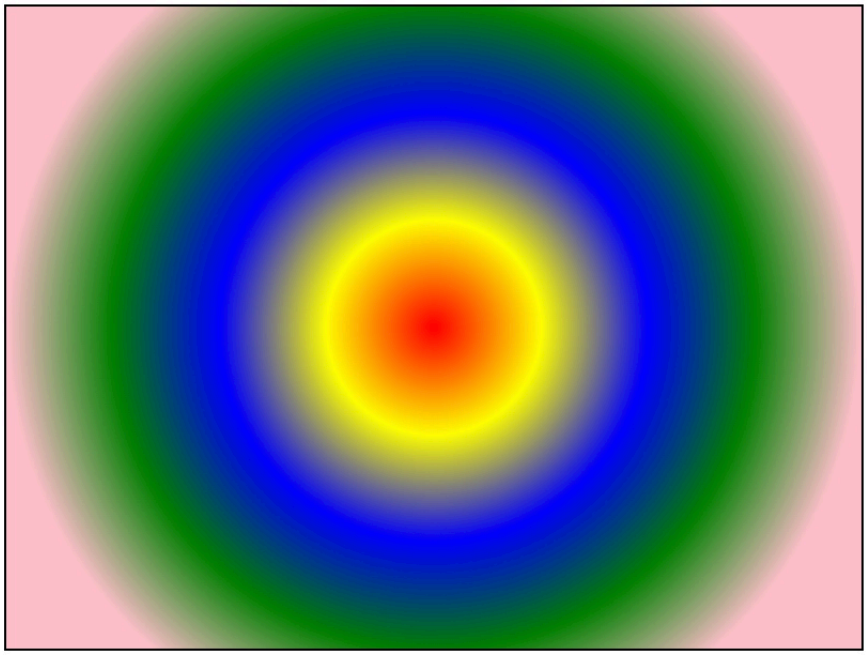
3. 绘制径向渐变

语法：context.createRadialGradient(x1,y1,r1,x2,y2,r2)
x1,y1,r1定义了一个以x1,y1为原点，半径为r1的圆
x2,y2,r2定义了一个以x2,y2为原点，半径为r2的圆

具体示例：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        context.beginPath();
        //添加径向渐变
        let radialGrad = context.createRadialGradient(200,150,0,200,150,200)
;
        //为径向渐变添加颜色
        radialGrad.addColorStop(0.0,"red");
        radialGrad.addColorStop(0.25,"yellow");
        radialGrad.addColorStop(0.5,"blue");
        radialGrad.addColorStop(0.75,"green");
        radialGrad.addColorStop(1.0,"pink");
        //将渐变设置为填充颜色
        context.fillStyle = radialGrad;
        context.fillRect(0,0,400,300);
    </script>
</body>
```

效果：



4. 绘制图案

语法：context.createPattern(image,type)
type的取值repeat，repeat-x，repeat-y和no-repeat

示例代码如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            //创建img对象
            let img = new Image();
            //指定图片的路径，图片处于当前目录下面
            img.src = "1.JPG";
            img.onload = function(){
                let ptn = context.createPattern(img,"repeat-y");
                context.fillStyle = ptn;//指定填充风格
                context.fillRect(0,0,400,300);//进行填充
            }
        }
        draw();
    </script>
```

```
</body>
```

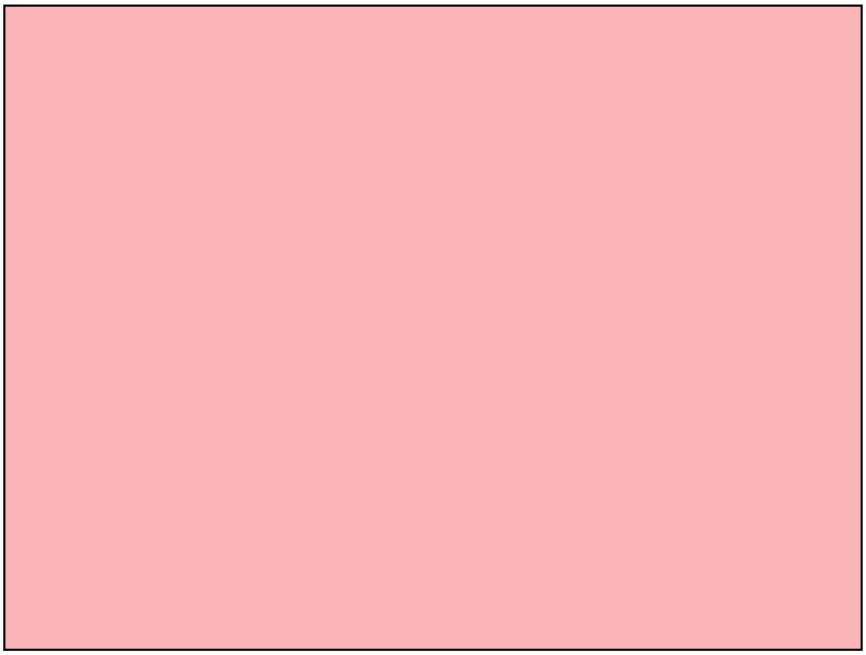
效果：

5. 绘制图形的透明度

使用rgba可以创建透明色，其中a就代表透明度，取值范围为0-1，示例代码如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
                //指定颜色
                context.fillStyle = "rgba(255,0,0,0.3)";
                //开始填充
                context.fillRect(0,0,400,300);
        }
        draw();
    </script>
</body>
```

效果：



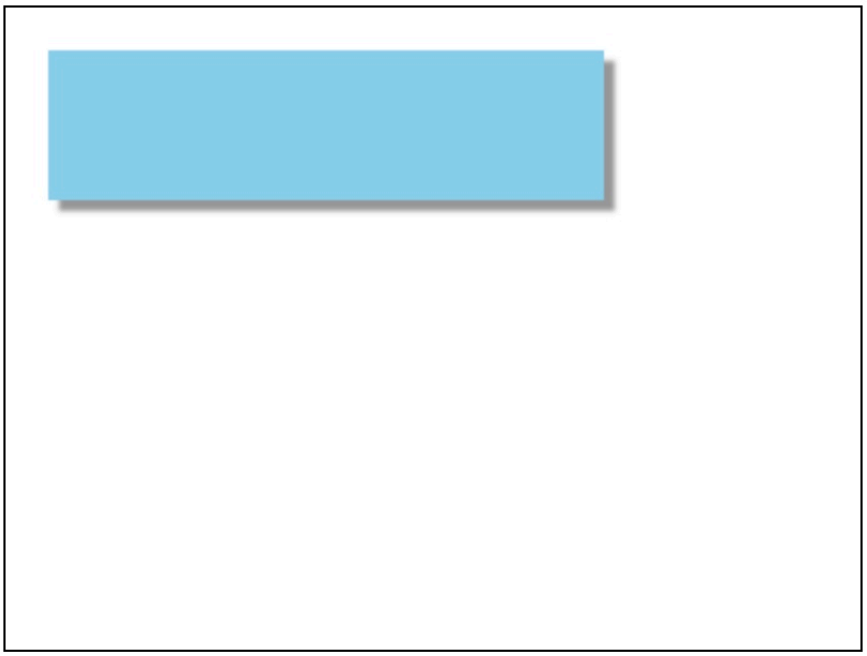
6. 创建阴影

语法基本上和CSS3里面创建阴影的方式相同，如下：
context.shadowOffsetX：阴影水平偏移
context.shadowOffsetY：阴影垂直偏移
context.shadowBlur：阴影模糊度
context.shadowColor：阴影颜色

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            //设置阴影相关属性
            context.shadowOffsetX = 5;
            context.shadowOffsetY = 5;
            context.shadowBlur = 2;
            context.shadowColor = "rgba(0,0,0,0.4)";
            //绘制矩形，矩形已经有了所设置的阴影，如果有文字，则文字也会有所设置的阴影
            context.fillStyle = "skyblue";
            context.fillRect(20,20,260,70);
        }
        draw();
    </script>
```

```
</body>
```

效果：



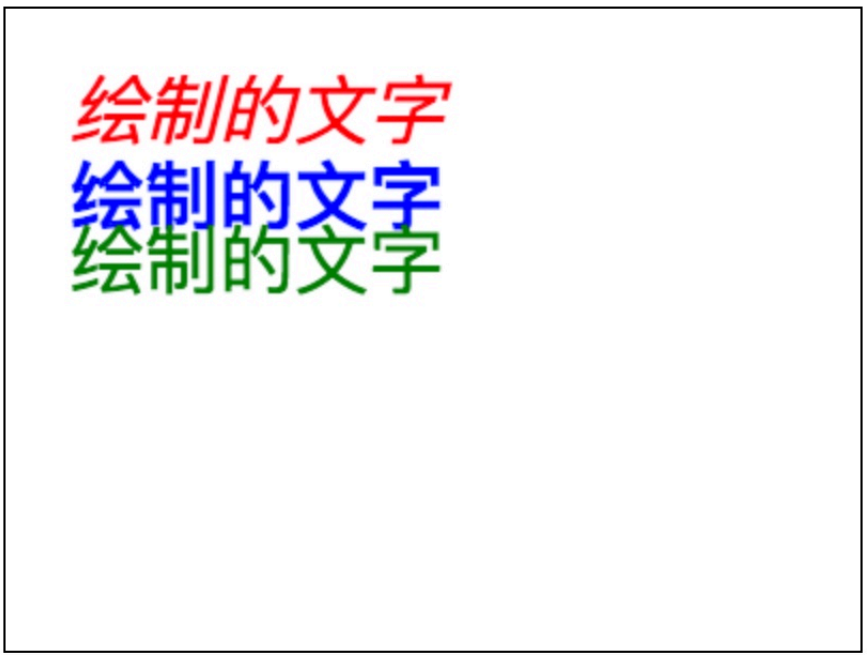
14-6-6 绘制文字

可以向canvas里面绘制文字
语法：content.fillText(text，x，y，[maxWidth])
fillText(text,x,y,maxWidth)：在画布上绘制"被填充的"文本
text：规定在画布上输出的文本。
x：开始绘制文本的 x 坐标位置（相对于画布）。
y：开始绘制文本的 y 坐标位置（相对于画布）。
maxWidth：可选。允许的最大文本宽度，以像素计。

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            //绘制第一段文字
            context.font = "italic 35px 黑体";
            context.fillStyle = "red";
            context.fillText("绘制的文字",30,60,200);
```

```
            //绘制第二段文字
            context.font = "bold 35px 宋体";
            context.fillStyle = "blue";
            context.fillText("绘制的文字",30,100);
            //绘制第三段文字
            context.font = "35px 微软雅黑";
            context.fillStyle = "green";
            context.fillText("绘制的文字",30,130);
        }
        draw();
    </script>
</body>
```

效果：



1. 文字相关属性

font：设置或返回文本内容当前的字体属性。

语法	默认值
context.font = "italic small-caps bold 12px arial"	10px sans-serif

其中`font`里面可以设置的属性如下：

值	描述
font-style	规定字体样式。可能的值：normal，italic，oblique
font-variant	规定字体变体。可能的值：normal，small-caps

font-weight	规定字体的粗细。可能的值：normal，bold，bolder，lighter，100-900
font-size/line-height	规定字号和行高，以像素计。
font-family	规定字体系列。
caption	使用标题控件的字体(比如按钮、下拉列表等)。
icon	使用用于标记图标的字体。
menu	使用用于菜单中的字体(下拉列表和菜单列表)。
message-box	使用用于对话框中的字体。
small-caption	使用用于标记小型控件的字体。
status-bar	使用用于窗口状态栏中的字体。

关于文字对齐的属性为`textAlign`，该属性可以设置或返回文本内容的当前对齐方式，可选属性值如下：

值	描述
start	默认。文本在指定的位置开始。
end	文本在指定的位置结束。
center	文本的中心被放置在指定的位置。
left	文本左对齐。
right	文本右对齐。

文字的基线的属性为`textBaseline`，该属性可以设置或返回在绘制文本时使用的当前文本基线，可选属性如下：

值	描述
alphabetic	默认。文本基线是普通的字母基线。
top	文本基线是 em 方框的顶端
hanging	文本基线是悬挂基线。
middle	文本基线是 em 方框的正中。

ideographic	文本基线是表意基线。
bottom	文本基线是 em 方框的底端。

2. 绘制轮廓文字

语法如下：
strokeText(text,x,y,maxWidth)：在画布上绘制文本(无填充)
text：规定在画布上输出的文本。
x：开始绘制文本的 x 坐标位置(相对于画布)。
y：开始绘制文本的 y 坐标位置(相对于画布)。
maxWidth：可选。允许的最大文本宽度，以像素计。

示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            //绘制第一段文字
            context.font = "italic 35px 黑体";
            context.fillStyle = "red";
            context.strokeText("绘制的文字",30,60,200);
            //绘制第二段文字
            context.font = "bold 35px 宋体";
            context.fillStyle = "blue";
            context.strokeText("绘制的文字",30,100);
            //绘制第三段文字
            context.font = "35px 微软雅黑";
            context.fillStyle = "green";
            context.strokeText("绘制的文字",30,130);
        }
        draw();
    </script>
</body>
```

效果：

3. 测量文字宽度

语法如下：
measureText()：返回包含指定文本宽度的对象。
context.measureText(text).width：其中text为要测量的文本。

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            context.font = "40px 微软雅黑";//设置字体样式
            context.textAlign = "center";//设置文本内容对齐方式
            context.textBaseline = "middle";//设置当前文本的基线
            //创建渐变颜色
            let grd = context.createLinearGradient(100,50,350,50);
            grd.addColorStop(0,"red");
            grd.addColorStop(0.3,"blue");
            grd.addColorStop(0.5,"green");
            grd.addColorStop(0.7,"yellow");
            grd.addColorStop(1,"black");
            context.fillStyle = grd;//将填充色设置为填充色
            let txt = "this is a test";//设置文本
            //计算文本宽度
```

```
            let length = Math.round(context.measureText(txt).width);
            //书写文字
            context.fillText(txt,200,100);
            context.fillText(`width:${length}`,200,150);
        }
        draw();
    </script>
</body>
```

效果：

14-6-7 操作和使用图像

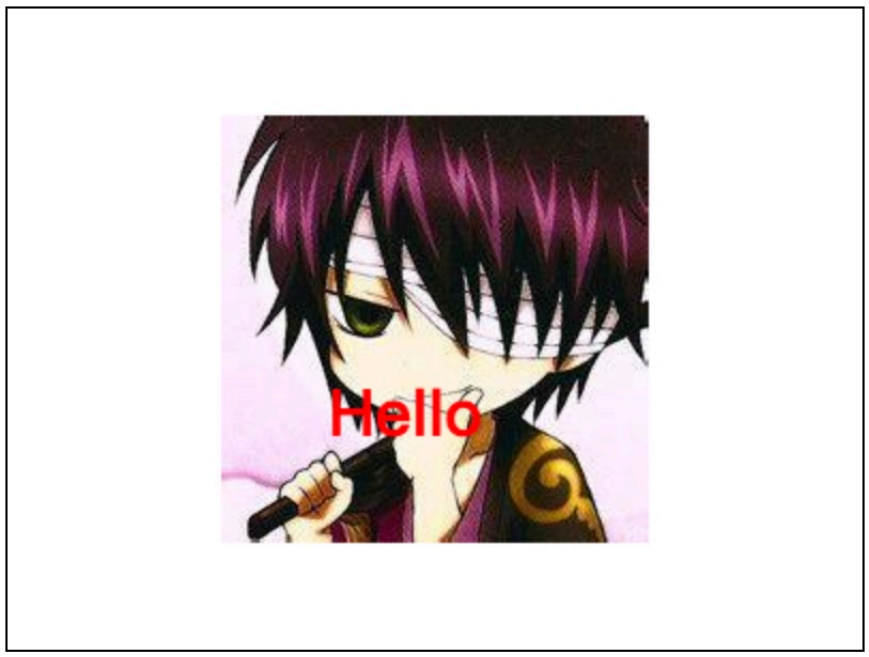
在canvas里面还可以引入图像，所使用到的方法为`draeImage()`方法，语法如下：
context.drawImage(image,x,y)

示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
```

```
            let img = new Image();
            img.src = "./1.jpg";
            img.onload = function(){
                context.drawImage(img,100,50);//载入图片的位置
                context.font = "bold 30px 微软雅黑";//设置字体属性
                context.fillStyle = "red";//设置字体颜色
                context.fillText("Hello",150,200);//绘制文字
            }
        }
        draw();
    </script>
</body>
```

效果：

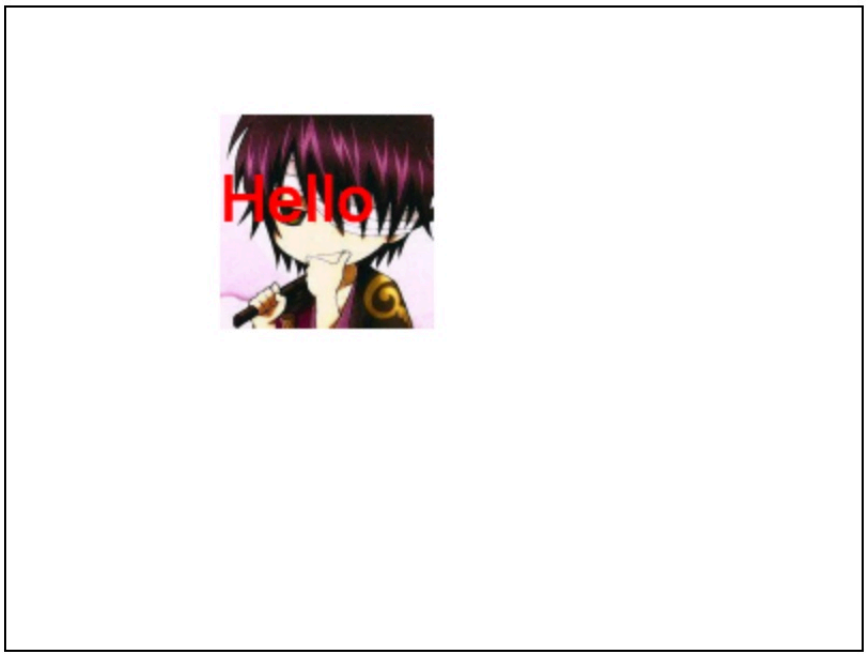


除了引入图像，`draeImage()`方法还可以改变引入图像的大小，语法如下：
context.drawImage(image,x,y,width,height)
width和height分别是用来指定图像的宽度和高度

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            let img = new Image();
```

```
            img.src = "./1.jpg";
            img.onload = function(){
                context.drawImage(img,100,50,100,100);//载入图片的位置
                context.font = "bold 30px 微软雅黑";//设置字体属性
                context.fillStyle = "red";//设置字体颜色
                context.fillText("Hello",100,100);//绘制文字
            }
        }
        draw();
    </script>
</body>
```

效果：



除了上面所提到的这两个使用以外，`draeImage()`方法还可以创建图像的切片，这个时候又需要提供新的参数，该方法完整的参数列表如下：

context.drawImage(image,sx,sy,sw,sh,dx,dy,dw,dh)

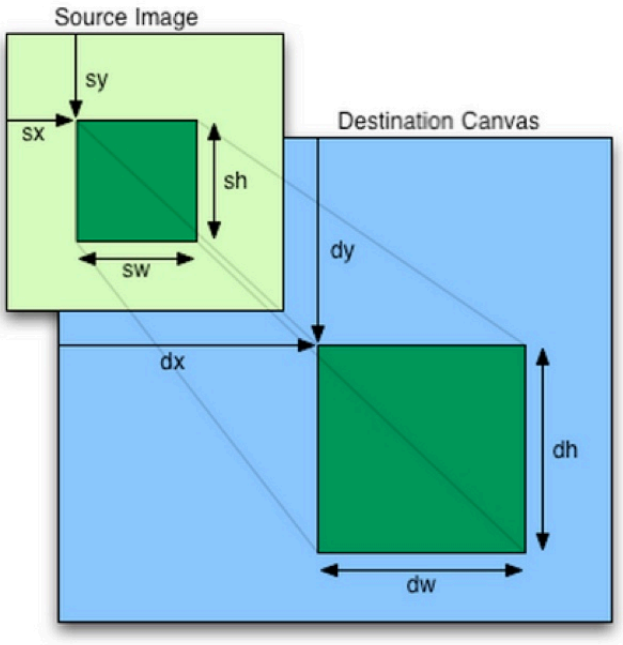
sx和sy：是源图像被切割区域的起始坐标

sw和sh：是源图像被切下来的宽度和高度

dx和dy：是被切割下来的图像放入canvas的起始坐标

dw和dh：是切割下来的图像的宽度和高度

如下图：



示例如下：

```
<body>
    <!-- 添加canvas元素 -->
    <canvas width="400" height="300" style="border:1px solid">
        您的浏览器不支持canvas元素，请升级您的浏览器
    </canvas>
    <script>
        //在JS中获取canvas元素
        let canvas = document.getElementsByTagName('canvas')[0];
        //获取到上下文，创建context对象
        let context = canvas.getContext("2d");
        let draw = function(){
            let img = new Image();
            img.src = "./1.jpg";
            img.onload = function(){
                context.drawImage(img,50,80,150,150,100,100,200,170);
            }
        }
        draw();
    </script>
</body>
```

效果：

14-7 HTML5 拖放

HTML5中提供了专门用于拖放的API。拖放也是网页中比较常见的功能之一。所谓拖放，即抓取对象以后拖到另一个位置。在HTML5中，拖放是标准的一部分，任何元素都能够拖放。

实际上，拖放这个词我们可以拆开来看，对应两个词。分别是拖拽和释放。拖拽对应英语单词Drag，而释放对应英语单词Drop。这是拖放这个行为发生时的两种状态。当鼠标点击源对象后一直移动对象不松手即为拖拽，一旦松手即为释放了。

14-7-1 源对象和目标对象

在讲解HTML5的拖放之前，有一个概念我们必须要弄清楚，那就是什么源对象和目标对象。

- 源对象：指的是我们鼠标点击的一个事物，这里可以是一张图片，一个div，或者一段文本等等。
- 目标对象：指的是我们拖动源对象后移动到一块区域，源对象可以进入这个区域，可以在这个区域上方悬停(未松手)，可以释松手释放将源对象放置此处(已松手)，也可以悬停后离开该区域。

具体如下图所示：

14-7-2 拖放相关API

除了上面解释的源对象和目标对象以外，其实还可以分出来一个过程对象，因为拖放事件由不同的元素产生。一个元素被拖放，他可能会经过很多个元素上，最终到达想要放置的元素内。所

以，除了被拖放的元素称为源对象，到达的元素称为目标对象以外，还可以将被经过的元素称为过程对象。不同的对象产生不同的拖放事件。

源对象：

- dragstart：源对象开始拖放。
- drag：源对象拖放过程中。(鼠标可能在移动也可能未移动)
- dragend：源对象拖放结束。

过程对象：

- dragenter：源对象开始进入过程对象范围内。
- dragover：源对象在过程对象范围内移动。
- dragleave：源对象离开过程对象的范围。

目标对象：

- drop：源对象被拖放到目标对象内。

14-7-3 DataTransfer对象

在所有拖放事件中提供了一个数据传递对象 `dataTransfer`，用于在源对象和目标对象间传递数据。接下来认识一下这个对象的方法和属性，来了解它是如何传递数据的。

setData()

该方法向 dataTransfer 对象中存入数据。接收两个参数，第一个表示要存入数据种类的字符串，现在支持有以下几种：

- text/plain：文本文字。
- text/html：HTML文字。
- text/xml：XML文字。
- text/uri-list：URL列表，每个URL为一行。

第二个参数为要存入的数据。例如：

```
event.dataTransfer.setData('text/plain','Hello World');
```

getData()

该方法从 `dataTransfer` 对象中读取数据。参数为在 `setData` 中指定的数据种类。例如：

```
event.dataTransfer.getData('text/plain');
```

clearData()

该方法清除`dataTransfer`对象中存放的数据。参数可选，为数据种类。若参数为空，则清空所有种类的数据。例如：

```
event.dataTransfer.clearData();
```

setDragImage()

该方法通过用img元素来设置拖放图标。接收三个参数，第一个为图标元素，第二个为图标元素离鼠标指针的X轴位移量，第三个为图标元素离鼠标指针的Y轴位移量。例如：

```
let source = document.getElementById('source'),
    icon = document.createElement('img');

icon.src = 'img.png';

source.addEventListener('dragstart',function(ev){
    ev.dataTransfer.setDragImage(icon,-10,-10)
},false)
```

effectAllowed 和 dropEffect 属性

这两个属性结合起来设置拖放的视觉效果。
值得注意的是：IE 不支持dataTransfer对象。

14-7-4 实现拖放排序

上面已经熟悉了拖放API的使用，我们来实现个简单的拖放排序，这也是在项目中比较常见的。
先来理一下思路：

- 将要拖动的元素的draggable属性设置为true
- 为要拖动的元素设置dragstart事件，具体的事件处理为：将该元素存储到dataTransfer里面
- 为目标对象设置dragover事件，具体的事件处理为：阻止默认行为
- 为目标对象设置drop事件，具体的事件处理为：阻止默认行为、将dataTransfer里面的数据取出，添加到目标对象里面

接下来我们来看一个拖放具体的示例，如下：

```
...
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        div{
            width: 200px;
            height: 200px;
            border: 3px solid;
            float: left;
            margin-right: 20px;
        }
    </style>
</head>
<body>
    <div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)">
        <img src="./1.jpg" draggable="true" ondragstart="drag(event)" id="dr
ag1">
    </div>
    <div id="div2" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
    <script>
        let drag = function(e){
            e.dataTransfer.setData("Text",e.target.id);
        }
        let allowDrop = function(e){
            e.preventDefault();
        }
        let drop = function(e){
            e.preventDefault();
            let data=e.dataTransfer.getData("Text");
            e.target.appendChild(document.getElementById(data));
        }
    </script>
</body>
...
```

效果：可以将图片进行左右的拖动

接下来我们来具体看一下这段代码。

首先，为了使元素可拖动，需要把`draggable`属性设置为true，然后在被拖动的源对象上面设置ondragstart事件，规定源对象开始被拖动触发什么事件，这里我们触发了drag事件。

drag事件要做的事儿，是通过`dataTransfer.setData()`方法来设置被拖数据的数据类型和值。这里我们设置的数据类型为Text，值是可拖动元素的id。

```
<img src="./1.jpg" draggable="true" ondragstart="drag(event)" id="drag1">
let drag = function(e){
    e.dataTransfer.setData("Text",e.target.id);
}
```

接下来，作为容器的两个div都设置了`ondragover`事件来规定在何处放置被拖动的数据。事件的处理函数调用的`preventDefault()`方法来阻止元素默认的处理方式(默认行为是以链接形式打开)

```
let allowDrop = function(e){
    e.preventDefault();
}
```

最后就是两个容器都设置了`ondrop`事件来规定图片拖动到div上方并且释放后要做的事儿。对应的事件处理函数要执行的操作

- 阻止浏览器默认行为
- 通过`dataTransfer.getData("Text")`方法获得被拖的数据。该方法将返回在`setData()`方法中设置为相同类型的任何数据。
- 通过DOM的API来添加相应的节点

```
let drop = function(e){
    e.preventDefault();
```

```
    let data=e.dataTransfer.getData("Text");
    e.target.appendChild(document.getElementById(data));
}
```

14-8 Notification

14-8-1 为什么需要Notification

传统的桌面通知可以写一个div放到页面右下角自动弹出来，并通过轮询等等其他方式去获取消息并推送给用户。但是这种方式有个弊端就是：当我在使用京东进行购物的时候，我是不知道人人网有消息推送过来给我的，而必须要等我把当前页面切到人人网才会知道有消息推送过来了。这种方式的消息推送是基于页面存活的。

所以我们需要这么一种策略：无论用户当前在看哪一个页面，只要有消息，都应该能推送给用户让他知道有新的消息来了。这就是Notification要解决的问题。

在以前，我们的通知实现主要是通过闪烁页面的标题内容来实现，实现原理其实很简单，就是定时器不断修改document.title的值。具体示例如下：

```
<body>
    <script>
        let titleInit = document.title, isShine = true;
        //定时器不断修改document.title的值
        setInterval(function(){
            let title = document.title;
            if (isShine == true)
            {
                if (/新/.test(title) == false)
                {
                    document.title = '【你有新消息】';
                }
                else{
                    document.title = '【        】';
                }
            }
            else{
                document.title = titleInit;
            }
        },100);
        //判断当前窗口是否处于焦点来决定还是否要闪烁
        window.onfocus = function(){
            isShine = false;
        };
        window.onblur = function(){
            isShine = true;
        };
    </script>
```

```
</body>
```

但是上面也有讲过，这种方式实现的通知非常依赖当前页面的张开。如果将浏览器最小化然后干其他的工作的话，就不会看到来自于浏览器的通知。

14-8-2 使用Notification

1. 创建notification

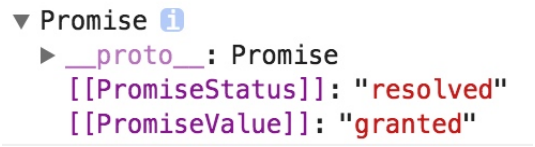
Notification API可以让我们使用系统通知来显示消息。这通常是屏幕角落的一个弹出消息，不过根据操作系统的不同而有所不同。使用系统通知的一个好处在于，即使调用它们的网页不是当前的标签页，消息通知依然能够显示。

在发送通知之前，我们需要得到用户的授权。通过全局对象的`Notification`的`requestPermission`方法来实现获取用户的授权。示例如下：

```
if(window.Notification)
{
    Notification.requestPermission();
}
```

这段代码会请求权限，询问用户是否允许显示通知

这个方法会返回一个promise。如下图：



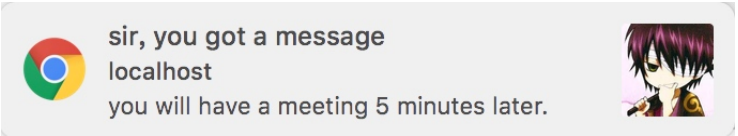
如果用户点击的是允许，那么该promise的值为`granted`。如果用户点击的是禁止，那么该promise的值为`denied`。这两个值可以在后面用于根据用户的选择来判断是否要显示桌面通知。

如果用户选择的是允许，那么我们就可以创建一个新的通知，如下面代码所示：

```
<body>
```

```
    <script>
        if(window.Notification)
        {
            Notification.requestPermission()
            .then((permission)=>{
                //如果用户选择的是允许
                if(Notification.permission==='granted'){
                    //通过new Notification来创建新的通知
                    new Notification("sir, you got a message", {
                        icon: './1.jpg',
                        body: 'you will have a meeting 5 minutes later.',
                    });
                }
            });
        }
    </script>
</body>
```

效果：当我们刷新页面之后，就会弹出相应的桌面通知，如下图



这里，构造函数的第一个参数是`sir, you got a message`是该通知的标题，第二个参数是一个JSON对象。在上面的例子中`you will have a meeting 5 minutes later.`是该通知的主体内容，而icon是显示一个图标。当然，除了这些属性以外，该JSON对象还有很多其他的属性可选，具体可以参考下面链接

https://developer.mozilla.org/zh-CN/docs/Web/API/notification

2. 延时的系统通知

前面介绍Notification的时候有介绍过，使用Notification最大的好处在于通知是系统级别的。所以即使调用它们的网页不是当前的标签页，消息通知依然能够显示。而上面的例子里面，我们一刷新页面就会弹出所创建的通知，无法看出Notification的优点。这一次我们来给系统通知一个延迟，然后将浏览器最小化之后，查看这个通知是否能够弹出来。示例代码如下：

```
<body>
    <script>
        window.setTimeout(function(){
            Notification.requestPermission()
            .then((permission)=>{
                if(Notification.permission==='granted'){
                    let notification = new Notification("sir, you got a mess
age",{
```

```
                        icon: './1.jpg',
                        body: 'you will have a meeting 5 minutes later.',
                    });
                }
            });
        },5000);
    </script>
</body>
```

在这个例子中我们设置了`setTimeout()`函数来延迟系统通知出现，让系统通知5秒以后再显示出来。然后我们将浏览器最小化后，系统通知依然可以在5秒钟后显示出来。

3. 事件处理

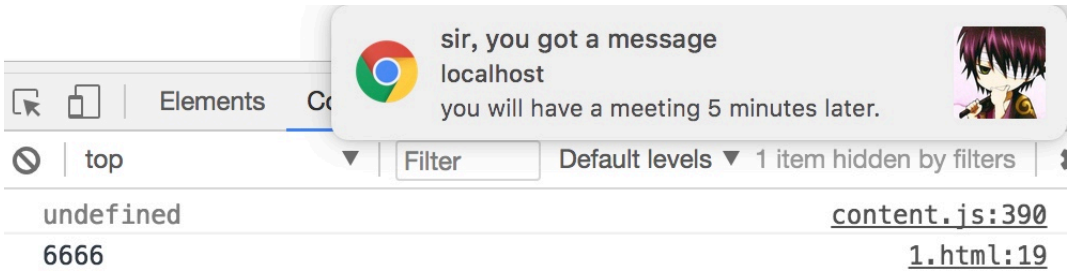
在Notification中可以使用的事件如下：

- Notification.onclick：处理`click`事件的处理。每当用户点击通知时被触发。
- Notification.onshow：处理`show`事件的处理。当通知显示的时候被触发。
- Notification.onerror：处理`error`事件的处理。每当通知遇到错误时被触发。
- Notification.onclose：处理`close`事件的处理。当用户关闭通知时被触发。

这里我们就演示一个点击事件的效果，示例代码如下：

```
<body>
    <script>
        if(window.Notification)
        {
            Notification.requestPermission()
            .then((permission)=>{
                if(Notification.permission==='granted'){
                    //将新建的通知存储在notification变量里面
                    let notification = new Notification("sir, you got a mess
age",{
                        icon: './1.jpg',
                        body: 'you will have a meeting 5 minutes later.',
                    });
                    //为notification添加点击事件
                    notification.addEventListener('click',function(){
                        console.log('6666');
                    })
                }
            });
        }
    </script>
</body>
```

效果：点击通知以后会触发事件



总结：

1. data-属性有助于将自定义的数据嵌入到网页中，然后利用JavaScript与HTML进行交互，增强用户体验。
2. Web Storage API允许键值类似于cookie的方式存储在用户的设备上，但是比cookie好的一点在于没有大小的限制。
3. Geolocation API可以访问到用户设备的地理坐标，但是由于国内特殊情况，一般要获取用户设备的地理坐标时，会采用百度地图的那一套API。
4. Web Worker API可用于在后台执行计算密集型的任务，这有助于避免网站变得无法响应。
5. 多媒体API允许使用JavaScript来控制音频和视频的播放。
6. canvas元素可以用来在网页上配合JavaScript动态地绘制图形与文本。
7. HTML5中拖放元素变为了标准的一部分，通过新提供的API可以对任何元素实现拖放。
8. Notification API允许在用户的系统上显示通知，即使调用它们的网页不是当前的标签页，消息通知依然能够显示。