# HTML5

### HTML5 表单元素

#### datalist 元素

datalist 元素用于为文本框提供一个可供选择的列表，使用 datalist 元素来为表单小部件提供建议的、自动完成的值，并使用一些 option 子元素来指定要显示的值。然后使用 list 属性将数据列表绑定到一个文本域(通常是一个 <input> 元素)。

一旦数据列表与表单小部件相关联，它的选项用于自动完成用户输入的文本。通常，这是作为一个下拉框向用户展示的，在输入框中输入可能匹配的内容。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<label for="myColor">What's your favorite color?</label>

<input type="text" name="myColor" id="myColor" list="mySuggestion" />

<datalist id="mySuggestion">

<option>black</option>

<option>blue</option>

<option>green</option>

<option>red</option>

<option>white</option>

<option>yellow</option>

</datalist>

</body>

</html>

也可以写成下面这样：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<label for="myColor">What's your favorite color?</label>

<input type="text" name="myColor" id="myColor" list="mySuggestion" />

<datalist id="mySuggestion">

<option value="black"> </option>

<option value="blue"> </option>

<option value="green"> </option>

<option value="red"> </option>

<option value="white"> </option>

<option value="yellow"> </option>

</datalist>

</body>

</html>

运行效果为：



HTML5 表单新增的属性

**autocomplete 属性**

autocomplete 属性规定表单是否应该启用自动完成功能：自动完成允许浏览器预测对字段的输入，当用户在字段开始键入时，浏览器基于之前键入过的值，应该显示出在字段中填写的选项。当 autocomplete 属性值为 on 时表示启用自动完成功能，为 off 时表示关闭。autocomplete 属性适用于 <form>，以及下面的 <input> 类型：text, search, url, telephone, email, password, datepickers, range 以及 color。

例子：

<body>

<form action="#" method="get" autocomplete="on">

Name:<input type="text" name="name" /><br />

E-mail: <input type="email" name="email" autocomplete="off" /><br />

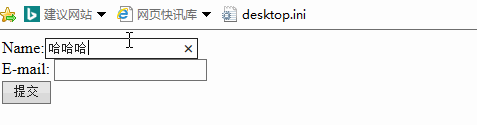
<input type="submit" value="提交" />

</form>

</body>



运行效果为：



**autofocus 属性**

autofocus 属性规定在页面加载时，域自动地获得焦点。适用于所有 <input> 标签的类型。

例子：

<body>

<form action="#" method="get">

Name：<input type="text" name="user\_name" autofocus="autofocus" /><br />

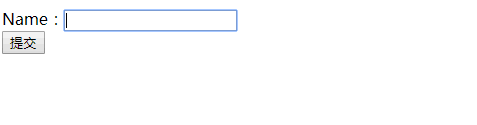
<input type="submit" value="提交" />

</form>

</body>



运行效果为：



注：大家可以把上面代码中 autofocus 属性删掉对比看看效果。

**form 属性**

form 属性规定输入域所属的一个或多个表单。form 属性适用于所有 <input> 标签的类型。form 属性必须引用所属表单的 id。

例子：

<body>

<form action="#" method="get" id="user\_form">

First name:<input type="text" name="fname" />

<input type="submit" value="提交" />

</form>

<p>

下面的输入域在 form 元素之外，但仍然是表单的一部分， 也就是说提交按钮会把

first name 和 last name的值都提交。

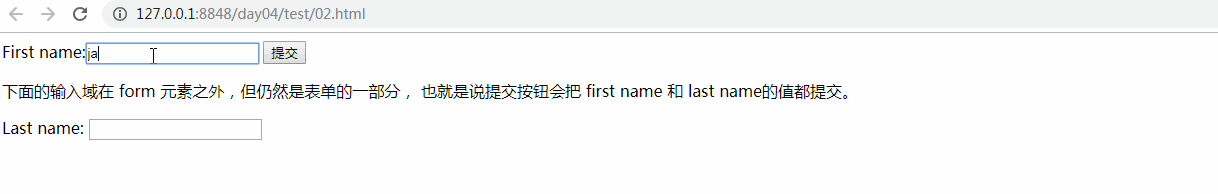
</p>

Last name: <input type="text" name="lname" form="user\_form" />

</body>



运行效果为：



注：如需引用一个以上的表单，请使用空格分隔的列表。

**multiple 属性**

multiple 属性规定输入域中可选择多个值，适用于以下类型的 <input> 标签：email 和 file。

例子：

<body>

<form action="#" method="get" id="user\_form">

选择文件: <input type="file" name="file" multiple="multiple" />

<input type="submit" value="提交" />

</form>

</body>



**novalidate 属性**

novalidate 属性规定在提交表单时不应该验证 form 或 input 域。适用于 <form>，以及下面的 <input> 类型：text, search, url, telephone, email, password, datepickers, range 以及 color。

例子：

<body>

<form action="#" method="get" id="user\_form" novalidate="true">

E-mail: <input type="email" name="user\_email" />

<input type="submit" value="提交" />

</form>

</body>



**pattern 属性**

pattern 属性规定用于验证 input 域的模式（pattern）。模式（pattern） 是正则表达式。pattern 属性适用于以下类型的 <input> 标签：text, search, url, telephone, email 以及 password。

例子：

<body>

<form action="#" method="get">

Name:

<input type="text" name="name" pattern="[A-z]{4}" title="请输入四个字母" />

<input type="submit" value="提交" />

</form>

</body>



运行效果为：



注：想要了解更多正则表达式的语法，请访问[正则表达式手册](http://tool.oschina.net/uploads/apidocs/jquery/regexp.html)。

**placeholder 属性**

placeholder 属性提供一种提示（hint），描述输入域所期待的值。适用于以下类型的 <input> 标签：text, search, url, telephone, email 以及 password。提示（hint）会在输入域为空时显示出现，会在输入域获得焦点时消失。

例子：

<body>

<form action="#" method="get">

<input type="search" name="user\_search" placeholder="实验楼" />

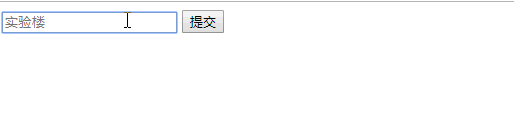
<input type="submit" value="提交" />

</form>

</body>



运行效果为：



**required 属性**

required 属性规定必须在提交之前填写输入域（不能为空）。适用于以下类型的 <input> 标签：text, search, url, telephone, email, password, date pickers, number, checkbox, radio 以及 file。

例子：

<body>

<form action="#" method="get">

<input type="search" name="user\_search" required="required" />

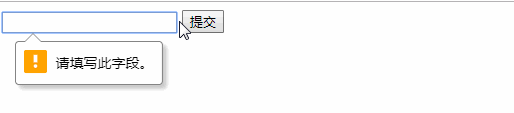
<input type="submit" value="提交" />

</form>

</body>



运行效果为：



### HTML5 输入类型

#### Input 类型 - email

email 类型用于应该包含 e-mail 地址的输入域。在提交表单时，会自动验证 email 域的值。

例子：

<body>

<form action="#" method="get">

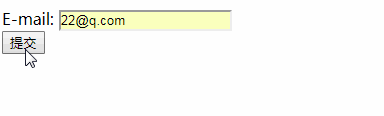
E-mail: <input type="email" name="user\_email" /><br />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



#### Input 类型 - url

url 类型用于应该包含 URL 地址的输入域。在提交表单时，会自动验证 url 域的值。

例子：

<body>

<form action="#" method="get">

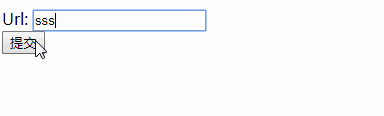
Url: <input type="url" name="user\_url" /><br />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



#### Input 类型 - number

number 类型用于应该包含数值的输入域。属性 max 设定允许输入的最大值，属性 min 设定允许输入的最小值，属性 value 设定默认值，属性 step 设定合法的数字间隔（比如 step 的值为 2，则合法的数字为 -2,0,2,4 等）。

例子：

<body>

<form action="#" method="get">

number：<input

type="number"

name="points"

min="0"

max="10"

step="2"

value="2"

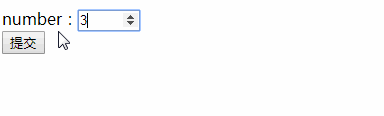
/><br />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



#### Input 类型 - range

range 类型用于应该包含一定范围内数字值的输入域。range 类型显示为滑动条。同样的 range 也有 max，min，value，step 属性与上面所讲的 number 中的一致。

例子：

<body>

<form action="#" method="get">

range:

<input

type="range"

name="points"

min="1"

max="10"

value="2"

step="2"

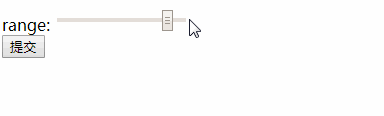
/><br />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



#### Input 类型 - Date Pickers（日期选择器）

HTML5 拥有多个可供选取日期和时间的新输入类型：

* date - 选取日、月、年
* month - 选取月、年
* week - 选取周和年
* time - 选取时间（小时和分钟）
* datetime - 选取时间、日、月、年（UTC 时间）**注意：此类型已被弃用，目前大多数浏览器都不再支持。**
* datetime-local - 选取时间、日、月、年（本地时间）

例子：

<body>

<form action="#" method="get">

Date: <input type="date" name="user\_date" /><br />

<input type="submit" value="提交" />

<br /><br />

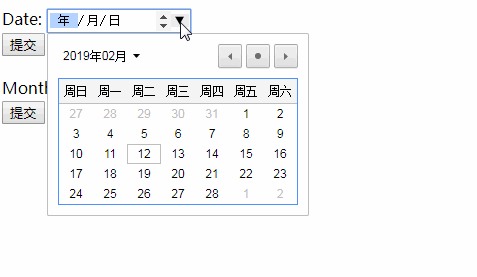
Month: <input type="month" name="user\_month" /><br />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



#### Input 类型 - color

color 类型用于选择颜色。

例子：

<body>

<form action="#" method="get">

<input type="color" name="colordemo" />

<input type="submit" value="提交" />

</form>

</body>

运行效果为：



### HTML5 视频介绍

HTML5 规定了一种通过 video 元素来包含视频的标准方法。

视频格式和浏览器支持如下所示：

| **格式** | **浏览器** |
| --- | --- |
| .ogg | FireFox 3.5+ ，chrome 5.0+ ，Opera 10.5+ |
| .mp4/H.264 | Safari 3.0+ ，chrome 5.0+ ，IE 9.0+ |
| .webm | FireFox 4.0+ ，chrome 6.0+ ，Opera10.6+ |

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<video width="320" height="240" controls="controls">

<source

src="https://labfile.oss.aliyuncs.com/courses/1248/video.ogg"

type="video/ogg"

/>

<source

src="https://labfile.oss.aliyuncs.com/courses/1248/video.mp4"

type="video/mp4"

/>

你的浏览器不支持video元素

</video>

</body>

</html>

注：<video> 与 </video> 之间插入的内容是供不支持 video 元素的浏览器显示的。video 元素允许多个 source 元素，source 元素可以链接不同的视频文件，浏览器将使用第一个可识别的格式。

引入单个文件也可以这样写：

<body>

<video

src="https://labfile.oss.aliyuncs.com/courses/1248/video.ogg"

width="320"

height="240"

controls="controls"

>

你的浏览器不支持video元素

</video>

</body>

video 标签的属性如下所示：



这些属性设置都很简单，大家可以自行尝试体验一番。

**字幕的简单使用**

使用常用的 WebVtt 字幕格式，在 <video> 中使用 <track> 元素引入字幕。例如：

<track

src="https://labfile.oss.aliyuncs.com/courses/1248/video\_ch.vtt"

srclang="zh"

kind="subtitles"

label="中文"

default

/>

<track

src="https://labfile.oss.aliyuncs.com/courses/1248/video\_en.vtt"

srclang="en"

kind="subtitles"

label="English"

/>

track 元素属性说明：

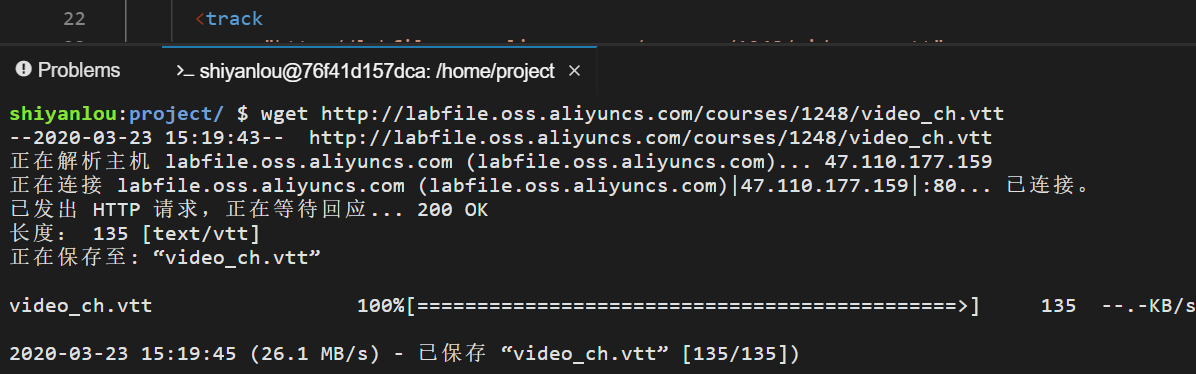
* src：指定资源 url。
* srclang：资源的语言，例如：中文 zh，英文 en。
* kind：默认值是 subtitles 字幕，captions 标题、音效及其他音频信息，descriptions 视频的文本描述，chapters 章节导航，metadata 元数据。
* label：选择字幕时候出现的文字。
* default：default 指的是默认会显示的字幕。例如两个 <track> 元素，如果都没有 default 属性，那都不显示，需要用户手动调出。另外，default 只能出现在一个 <track> 元素上。

例子：

首先我们在终端中分别输入以下命令，获取所需字幕文件：

wget https://labfile.oss.aliyuncs.com/courses/1248/video\_ch.vtt

wget https://labfile.oss.aliyuncs.com/courses/1248/video\_en.vtt



<body>

<video

src="https://labfile.oss.aliyuncs.com/courses/1248/video.ogg"

width="320"

height="240"

controls="controls"

>

你的浏览器不支持video元素

<track

src="video\_ch.vtt"

srclang="en"

kind="subtitles"

label="中文"

default

/>

<track src="video\_en.vtt" srclang="en" kind="subtitles" label="English" />

</video>

</body>

video\_ch.vtt 中的内容为：

WEBVTT 00:00:01.000 --> 00:00:04.000 不准在课堂上打游戏 00:00:05.000 -->

00:00:09.000 好好学习，天天向上！

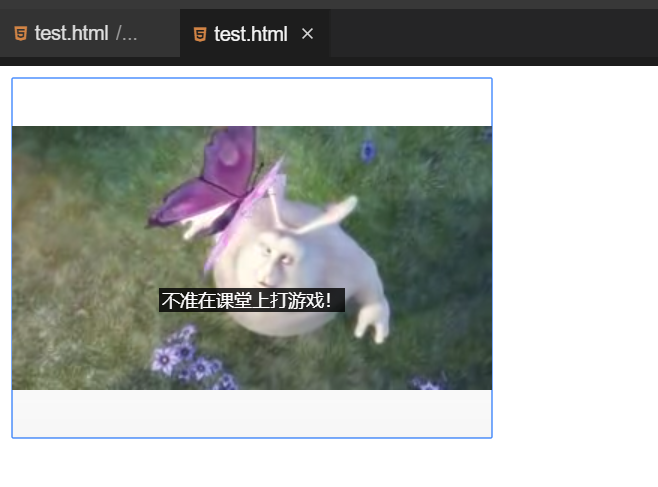
video\_en.vtt 中的内容为：

WEBVTT 00:00:01.000 --> 00:00:04.000 Don't play games in class！ 00:00:05.000

--> 00:00:09.000 Study hard and make progress every day！



运行效果如下所示：



通过上面的案例，我们了解了简单的视频制作以及为视频添加字幕和切换字幕。

### HTML5 音频介绍

HTML5 规定了一种通过 audio 元素来包含音频的标准方法。

音频格式和浏览器支持如下所示：

| **格式** | **浏览器** |
| --- | --- |
| .ogg | FireFox 3.5+，chrome 3.0+，Opera 10.5+ |
| .mp3 | Safari 3.0+，chrome 3.0+，IE 9.0+ |
| .wav | FireFox 3.5+，Safari 3.0+，Opera10.5+ |

例子：

<body>

<audio controls="controls">

<source

src="https://labfile.oss.aliyuncs.com/courses/1248/video.ogg"

type="audio/ogg"

/>

<source

src="https://labfile.oss.aliyuncs.com/courses/1248/phone.mp3"

type="audio/mpeg"

/>

你的浏览器不支持audio元素

</audio>

</body>

注：<audio> 与 </audio> 之间插入的内容是供不支持 audio 元素的浏览器显示的。audio 元素允许多个 source 元素，source 元素可以链接不同的音频文件，浏览器将使用第一个可识别的格式。

引入单个文件也可以这样写：

<body>

<audio

src="https://labfile.oss.aliyuncs.com/courses/1248/phone.mp3"

controls="controls"

>

你的浏览器不支持video元素

</audio>

</body>

audio 标签的属性如下所示：

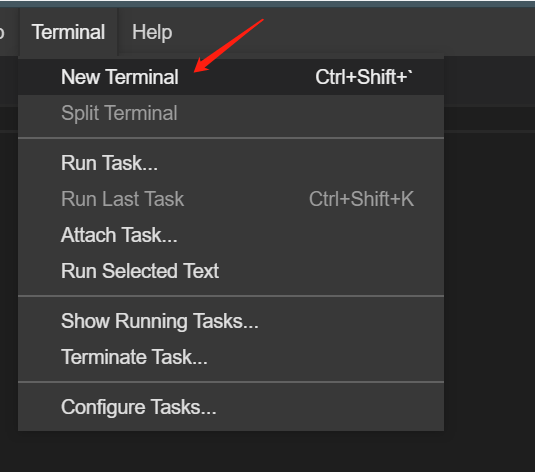


### HTML5 拖放概述

拖放（Drag 和 drop）是 HTML5 标准的组成部分，任何元素都能够拖放，从字面意思上来理解，拖放就是抓取对象后拖到另一个位置上。

注：img元素和 a 元素（必须指定 href）默认允许拖放。

打开 Terminal 终端：



在 Terminal 中输入以下命令获取本节使用到的图片：

wget https://labfile.oss.aliyuncs.com/courses/1248/drag-image.png

图片的下载位置，请注意在后续 html 文件所在同目录下。

#### 浏览器支持

Internet Explorer 9、Firefox、Opera 12、Chrome 以及 Safari 5 支持拖放。

**注释：**在 Safari 5.1.2 中不支持拖放。

### HTML5 拖放使用

注意：之后的学习会涉及 JavaScript 的基础用法，如果有不明白的，可以到实验楼[JavaScript 基础教程](https://www.lanqiao.cn/courses/1238)进行学习。 先来看个例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

#div1 {

width: 200px;

height: 100px;

padding: 10px;

border: 1px solid red;

}

</style>

<script type="text/javascript">

function allowDrop(ev) {

ev.preventDefault();

}

function drag(ev) {

ev.dataTransfer.setData('Text', ev.target.id);

}

function drop(ev) {

ev.preventDefault();

var data = ev.dataTransfer.getData('Text');

ev.target.appendChild(document.getElementById(data));

}

</script>

</head>

<body>

<p>请把图片拖放到矩形中：</p>

<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>

<br />

<img

id="drag1"

src="drag-image.png"

draggable="true"

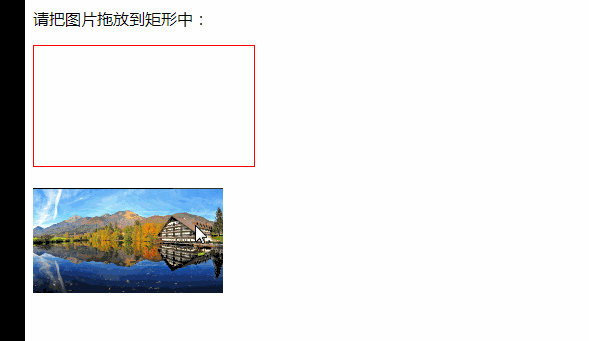
ondragstart="drag(event)"

/>

</body>

</html>

运行效果为：



下面将为大家一一介绍拖放是如何实现的。

#### 确定什么是可拖动的

为了使元素可拖放，首先把 draggable 属性设置为 true，再加上全局事件处理函数 ondragstart，如下所示：

<img draggable="true" ondragstart="drag(event)" />

#### 定义拖动数据

每个 drag event 都有一个 dataTransfer 属性保保存事件的数据。这个属性（ DataTransfer 对象）也有管理拖动数据的方法。setData() 方法添加一个项目的拖拽数据，如下面的示例代码所示：

function drag(ev) {

ev.dataTransfer.setData('Text', ev.target.id);

}

在这个例子中数据类型是 "Text"，值是可拖动元素的 id ("drag1")。

#### 定义一个放置区

ondragover 事件规定在何处放置被拖动的数据。默认地，无法将数据/元素放置到其他元素中。如果需要设置允许放置，我们必须阻止对元素的默认处理方式,如下所示：

function allowDrop(ev) {

ev.preventDefault();

}

#### 进行放置

当放置被拖数据时，会发生 drop 事件。如下所示：

function drop(ev) {

//调用 preventDefault() 来避免浏览器对数据的默认处理

ev.preventDefault();

//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。

var data = ev.dataTransfer.getData('Text');

//被拖数据是被拖元素的 id ("drag1"),把被拖元素追加到放置元素（目标元素）中

ev.target.appendChild(document.getElementById(data));

}

上面只是简单的讲解了一个图片拖放的案例，想要了解更多拖放的知识，请访问：[MDN HTML 拖放 API](https://developer.mozilla.org/zh-CN/docs/Web/API/HTML_Drag_and_Drop_API#%E6%8E%A5%E5%8F%A3)

## Canvas 元素

canvas 元素的外观和 img 元素相似，但是没有 img 元素的 src 属性和 alt 属性。canvas 的 width 属性和 height 属性用来设置画布的宽和高，单位是 px。默认的画布的高度是 150px，宽度是 300px。

<body>

<canvas

id="myCanvas"

width="200"

height="100"

style="border:2px solid #000000;"

>

</canvas>

</body>

注意: 默认情况下 <canvas> 元素没有边框和内容，标签通常需要指定一个 id 属性 (脚本中经常引用)。

### 绘制简单图形

canvas 元素本身并不能实现图形绘制，所有的绘制工作必须要和 JavaScript 脚本结合起来。首先，给 canvas 元素添加一个 id 属性，这样能够让我们在 JavaScript 脚本中通过 id 属性找到对应的 canvas 元素。

<canvas

id="myCanvas"

width="200"

height="100"

style="border:2px solid #000000;"

>

<!-- 添加id属性值为myCanvas -->

</canvas>

添加了 id 属性后，找到对应的 canvas 元素：

var myCanvas = document.getElementById('myCanvas');

// 通过document.getElementById来找到id为myCanvas的元素

然后通过 canvas 元素的 getContext()方法获取上下文，即创建 Context 对象，以获取允许进行绘制的 2D 环境。

var ctx = myCanvas.getContext('2d');

最后通过 Context 对象的相关方法完成绘制，比如：fillStyle()等。

ctx.fillStyle = 'red';

//设置矩形的位置和尺寸（位置从 左上角原点坐标开始，尺寸为100\*100的矩形）

ctx.fillRect(0, 0, 100, 100);

整体的代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" style="width:200;height:100">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

var myCanvas = document.getElementById('mycanvas');

var ctx = myCanvas.getContext('2d');

//设置颜色

ctx.fillStyle = 'red';

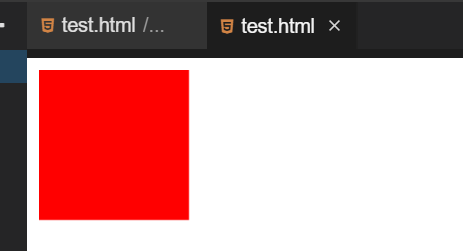
//设置矩形的位置和尺寸（位置从 左上角原点坐标开始，尺寸为100\*100的矩形）

ctx.fillRect(0, 0, 100, 100);

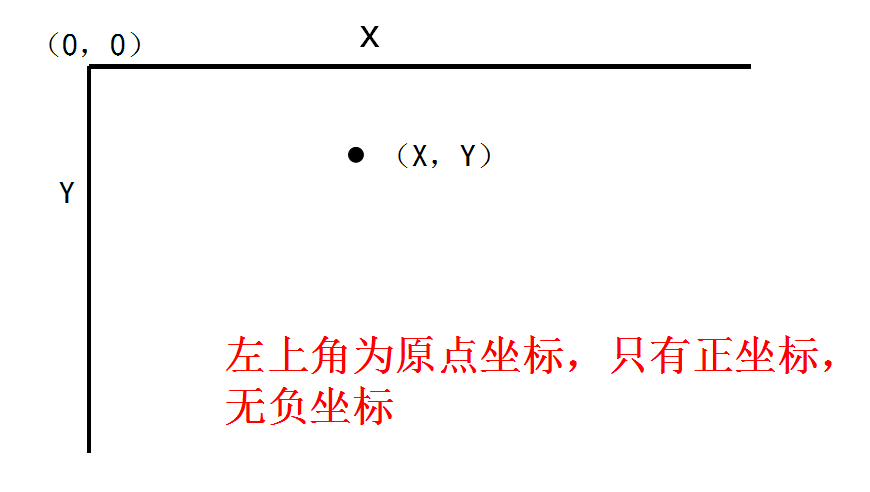
</script>

</body>

</html>



**注意：**进行绘制时，需要指定确定的坐标位置，坐标原点(0,0)位于 canvas 的左上角，x 轴水平方向向右延伸，y 轴垂直向下延伸，如图：



### 直线绘制

* strokeStyle：设置或返回笔的颜色、渐变或模式。默认值为：#000000。
* lineWidth：设置或返回当前的线条宽度 ，以像素计。
* beginPath()：起始一条路径，或重置当前路径。
* closePath()：创建从当前点回到起始点的路径。
* moveTo()：把路径移动到画布中的指定点，不创建线条。
* lineTo()：添加一个新点，然后在画布中创建从该点到最后指定点的线条。
* stroke()：绘制已定义的路径。

绘制一条直线例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

// 开始路径绘制

ctx.beginPath();

// 设置路径起点，坐标为(20,20)

ctx.moveTo(20, 20);

// 添加一个(200,200)的新点

ctx.lineTo(200, 200);

// 设置线宽

ctx.lineWidth = 2.0;

// 设置线的颜色

ctx.strokeStyle = '#CC0000';

// 绘制已定义的路径

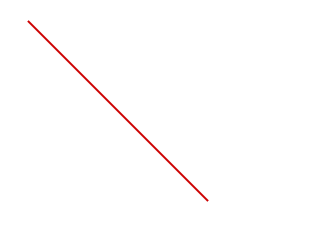
ctx.stroke();

</script>

</body>

</html>

运行效果为：



绘制三角形例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

// 开始路径绘制

ctx.beginPath();

// 设置路径起点，坐标为(20,20)

ctx.moveTo(20, 20);

// 添加一个(200,200)的新点

ctx.lineTo(200, 200);

// 添加一个(400,20)的新点

ctx.lineTo(400, 20);

//创建从当前点回到起始点的路径

ctx.closePath();

// 设置线宽

ctx.lineWidth = 2.0;

// 设置线的颜色

ctx.strokeStyle = '#CC0000';

// 绘制已定义的路径

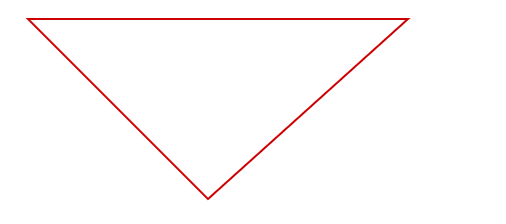
ctx.stroke();

</script>

</body>

</html>

运行效果为：



### 矩形绘制

#### rect() 方法介绍

使用 rect() 方法创建矩形。语法为：

ctx.rect(x,y,width,height);

参数说明：

* x 表示矩形左上角的 x 坐标。
* y 表示矩形左上角的 y 坐标。
* width 表示矩形的宽度，以像素计。
* height 表示矩形的高度，以像素计。

注：使用 stroke() 或 fill() 方法在画布上实际地绘制矩形。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制矩形

ctx.rect(10, 10, 100, 200);

//绘制已定义的路径

ctx.stroke();

</script>

</body>

</html>

运行效果为：



#### strokeRect() 方法介绍

使用 strokeRect() 方法绘制矩形（不填色）。笔触的默认颜色是黑色。语法为：

ctx.strokeRect(x, y, width, height);

注：参数与 rect() 方法一致，唯一的区别是这里不需要再加一句 stroke() 或 fill() 方法。无法填色。

前面绘制矩形的例子也可以这样写：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制矩形

ctx.strokeRect(10, 10, 100, 200);

</script>

</body>

</html>

#### fillRect() 方法介绍

使用 fillRect() 方法创建实心矩形。语法为：

ctx.fillRect(x, y, width, height);

注：参数说明与前面一致，默认的填充颜色为黑色，可以使用 fillStyle 属性来设置用于填充绘图的颜色、渐变或模式。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制矩形

ctx.fillRect(10, 10, 100, 200);

</script>

</body>

</html>

运行效果为：



#### clearRect() 方法介绍

使用 clearRect() 方法清空给定矩形内的指定像素。语法为：

ctx.clearRect(x, y, width, height);

注：参数说明与前面一致。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制矩形

ctx.fillRect(10, 10, 100, 200);

//清空指定像素

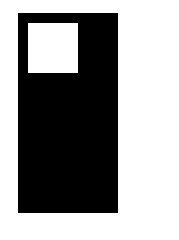
ctx.clearRect(20, 20, 50, 50);

</script>

</body>

</html>

运行效果为：



### 圆和扇形的绘制

使用 arc() 方法绘制圆或者椭圆。语法为：

ctx.arc(x, y, r, sAngle, eAngle, counterclockwise);



参数说明：

* x 表示圆的中心的 x 坐标。
* y 表示圆的中心的 y 坐标。
* r 表示圆的半径。
* sAngle 表示起始角，以弧度计（特别需要注意的是弧的圆形的三点钟位置是 0 度而不是常规以为的 90 度）。
* eAngle 表示结束角，以弧度计。
* counterclockwise 表示绘制圆的方向，值为 false 表示顺时针，为 true 表示逆时针。是一个可选值，默认值是 false。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//开始路径绘制

ctx.beginPath();

//绘制圆

ctx.arc(100, 75, 50, 0, 2 \* Math.PI);

//绘制已定义的路径

ctx.stroke();

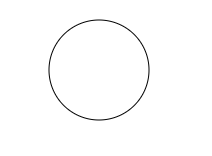
</script>

</body>

</html>



运行效果为：



当然绘制扇形也很简单，只需要给定角度值小于 2 \* Math.PI 再闭合一下就可以了，来看看例子。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//开始路径绘制

ctx.beginPath();

//绘制圆

ctx.arc(100, 75, 50, 0, 0.5 \* Math.PI);

//闭合

ctx.moveTo(100, 125);

ctx.lineTo(100, 75);

ctx.lineTo(150, 75);

//绘制已定义的路径

ctx.stroke();

</script>

</body>

</html>



运行效果为：



### 填充和渐变

使用 fillStyle 属性，设置或返回用于填充绘画的颜色、渐变或模式。语法为：

ctx.fillStyle = color | gradient | pattern;

参数说明：

* color 表示绘图填充的颜色。默认值是 #000000。
* gradient 表示用于填充绘图的渐变对象（线性或放射性）。
* pattern 表示用于填充绘图的 pattern 对象。

例子：

绘制实心矩形，填充颜色为红色。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//填充为红色

ctx.fillStyle = 'red';

//绘制实心矩形

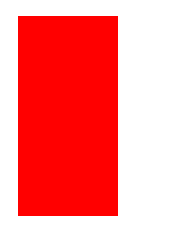
ctx.fillRect(10, 10, 100, 200);

</script>

</body>

</html>

运行效果为：



#### 渐变

使用 createLinearGradient() 方法创建线性渐变。语法为：

ctx.createLinearGradient(x0, y0, x1, y1);

参数说明：

* x0 表示渐变开始点的 x 坐标。
* y0 表示渐变开始点的 y 坐标。
* x1 表示渐变结束点的 x 坐标。
* y1 表示渐变结束点的 y 坐标。

使用 addColorStop() 方法规定渐变对象中的颜色和停止位置。语法为：

gradient.addColorStop(stop, color);

参数说明：

* stop 表示渐变中开始与结束之间的位置。是介于 0.0 与 1.0 之间的值。
* color 表示在结束位置显示的 CSS 颜色值。

注：addColorStop() 方法与 createLinearGradient() 或 createRadialGradient() 一起使用。我们可以多次调用 addColorStop() 方法来改变渐变。如果我们不对 gradient 对象使用该方法，那么渐变将不可见。为了获得可见的渐变，至少需要创建一个色标。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//设置渐变色

var gradient = ctx.createLinearGradient(0, 0, 170, 0);

gradient.addColorStop(0, 'red');

gradient.addColorStop('0.2', 'orange');

gradient.addColorStop('0.5', 'yellow');

gradient.addColorStop('0.7', 'green');

gradient.addColorStop(1, 'blue');

//填充色为渐变色

ctx.fillStyle = gradient;

//绘制实心矩形

ctx.fillRect(10, 10, 100, 200);

</script>

</body>

</html>

运行效果为：



使用 createRadialGradient() 方法创建放射状/环形的渐变。语法为：

ctx.createRadialGradient(x0, y0, r0, x1, y1, r1);

参数说明：

* x0 表示渐变的开始圆的 x 坐标。
* y0 表示渐变的开始圆的 y 坐标。
* r0 表示开始圆的半径。
* x1 表示渐变的结束圆的 x 坐标。
* y1 表示渐变的结束圆的 y 坐标。
* r1 表示结束圆的半径。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//设置渐变色

var gradient = ctx.createRadialGradient(75, 50, 5, 90, 60, 100);

gradient.addColorStop(0, 'red');

gradient.addColorStop('0.2', 'orange');

gradient.addColorStop('0.5', 'yellow');

gradient.addColorStop('0.7', 'green');

gradient.addColorStop(1, 'blue');

//填充色为渐变色

ctx.fillStyle = gradient;

//绘制实心矩形

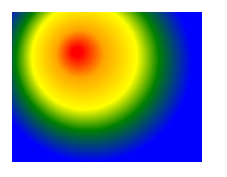
ctx.fillRect(10, 10, 190, 150);

</script>

</body>

</html>

运行效果为：



#### fill() 方法

使用 fill() 方法填充当前的图像（路径）。默认颜色是黑色。填充另一种颜色/渐变使用 fillStyle 属性。

语法为：

ctx.fill();

注：如果路径未关闭，那么 fill() 方法会从路径结束点到开始点之间添加一条线，以关闭该路径，然后填充该路径。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制矩形

ctx.rect(20, 20, 150, 100);

ctx.fillStyle = 'red';

ctx.fill();

</script>

</body>

</html>

运行效果为：



### 文字绘制

#### fillText() 方法

使用 fillText() 方法在在画布上绘制实心文本。语法为：

ctx.fillText(text, x, y, maxWidth);

参数说明：

* text 规定在画布上输出的文本。
* x 表示开始绘制文本的 x 坐标位置（相对于画布）。
* y 表示开始绘制文本的 y 坐标位置（相对于画布）。
* maxWidth 表示允许的最大文本宽度，以像素计。可选。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//绘制实心文本

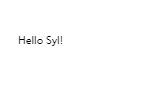
ctx.fillText('Hello Syl!', 10, 50);

</script>

</body>

</html>

运行效果为：



#### strokeText() 方法

使用 strokeText() 方法在画布上绘制空心文本。语法为：

ctx.strokeText(text, x, y, maxWidth);

注：参数说明与 fillText() 方法一致。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//设置字体样式

ctx.font = '50px Georgia';

//绘制空心文本

ctx.strokeText('Hello Syl!', 10, 50);

</script>

</body>

</html>

运行效果为：



#### font 属性

使用 font 属性设置或返回画布上文本内容的当前字体属性。font 属性使用的语法与 CSS font 属性相同。

#### textAlign 属性

使用 textAlign 属性设置或返回文本内容的当前对齐方式。语法为：

ctx.textAlign = 'center|end|left|right|start';

参数说明：

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

注：使用 fillText() 或 strokeText() 方法在实际地在画布上绘制并定位文本。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="520px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//画一条线便于展示

ctx.strokeStyle = 'blue';

ctx.moveTo(200, 20);

ctx.lineTo(200, 180);

ctx.stroke();

//设置字体样式

ctx.font = '20px Georgia';

//值为start的情况

ctx.textAlign = 'start';

ctx.strokeText('Hello Syl!', 200, 20);

//值为center的情况

ctx.textAlign = 'center';

ctx.strokeText('Hello Syl!', 200, 60);

//值为end的情况

ctx.textAlign = 'end';

ctx.strokeText('Hello Syl!', 200, 100);

//值为left的情况

ctx.textAlign = 'left';

ctx.strokeText('Hello Syl!', 200, 140);

//值为right的情况

ctx.textAlign = 'right';

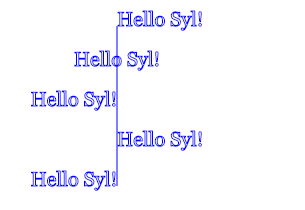
ctx.strokeText('Hello Syl!', 200, 180);

</script>

</body>

</html>

运行效果为：



#### textBaseline 属性

textBaseline 属性设置或返回在绘制文本时的当前文本基线。语法为：

ctx.textBaseline = 'alphabetic|top|hanging|middle|ideographic|bottom';

参数说明：

* alphabetic 表示文本基线是普通的字母基线。默认值。
* top 表示文本基线是 em 方框的顶端。
* hanging 表示文本基线是悬挂基线。
* middle 表示文本基线是 em 方框的正中。
* ideographic 表示文本基线是表意基线。
* bottom 表示文本基线是 em 方框的底端。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="1314px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//画一条线便于展示

ctx.strokeStyle = 'blue';

ctx.moveTo(20, 100);

ctx.lineTo(1314, 100);

ctx.stroke();

//设置字体样式

ctx.font = '30px Georgia';

//值为alphabetic的情况

ctx.textBaseline = 'alphabetic';

ctx.fillText('Hello Syl!', 20, 100);

//值为top的情况

ctx.textBaseline = 'top';

ctx.fillText('Hello Syl!', 220, 100);

//值为hanging的情况

ctx.textBaseline = 'hanging';

ctx.fillText('Hello Syl!', 420, 100);

//值为middle的情况

ctx.textBaseline = 'middle';

ctx.fillText('Hello Syl!', 620, 100);

//值为ideographic的情况

ctx.textBaseline = 'ideographic';

ctx.fillText('Hello Syl!', 820, 100);

//值为bottom的情况

ctx.textBaseline = 'bottom';

ctx.fillText('Hello Syl!', 1020, 100);

</script>

</body>

</html>

运行效果为：



### 图片绘制

在 Terminal 输入以下命令获取图片绘制用到的图片：

wget https://labfile.oss.aliyuncs.com/courses/1248/a.png

使用 drawImage() 方法在画布上绘制图像、画布或视频。drawImage() 方法也能够绘制图像的某些部分，或增加或减少图像的尺寸。

canvas 绘制图片的基本格式为：

//创建一个图片对象

var image = new Image();

//设置图片的路径

image.src = 'xxx';

//当图片加载完成后

image.onload = function () {

//绘制图片

ctx.drawImage();

};

语法 1，在画布上定位图像：

ctx.drawImage(img, x, y);

语法 2，在画布上定位图像，并规定图像的宽度和高度：

ctx.drawImage(img, x, y, width, height);

语法 3，剪切图像，并在画布上定位被剪切的部分：

ctx.drawImage(img, sx, sy, swidth, sheight, x, y, width, height);

参数说明：

* img 规定要使用的图像、画布或视频。
* sx 表示开始剪切的 x 坐标位置。可选值。
* sy 表示开始剪切的 y 坐标位置。可选值。
* swidth 表示被剪切图像的宽度。可选值。
* sheight 表示被剪切图像的高度。可选值。
* x 表示在画布上放置图像的 x 坐标位置。
* y 表示在画布上放置图像的 y 坐标位置。
* width 表示要使用的图像的宽度（伸展或缩小图像）。可选值。
* height 表示要使用的图像的高度，（伸展或缩小图像）。可选值。

例子 1，在画布上定位图像并作出一个立体的效果：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="1314px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//创建一张图片

var image = new Image();

//设置图片的路径

image.src = 'a.png';

//当图片加载完成后

image.onload = function () {

//输出5张照片

for (var i = 0; i < 5; i++) {

//参数：（1）绘制的图片 （2）坐标x，（3）坐标y

ctx.drawImage(image, 100 + i \* 80, 100 + i \* 80);

}

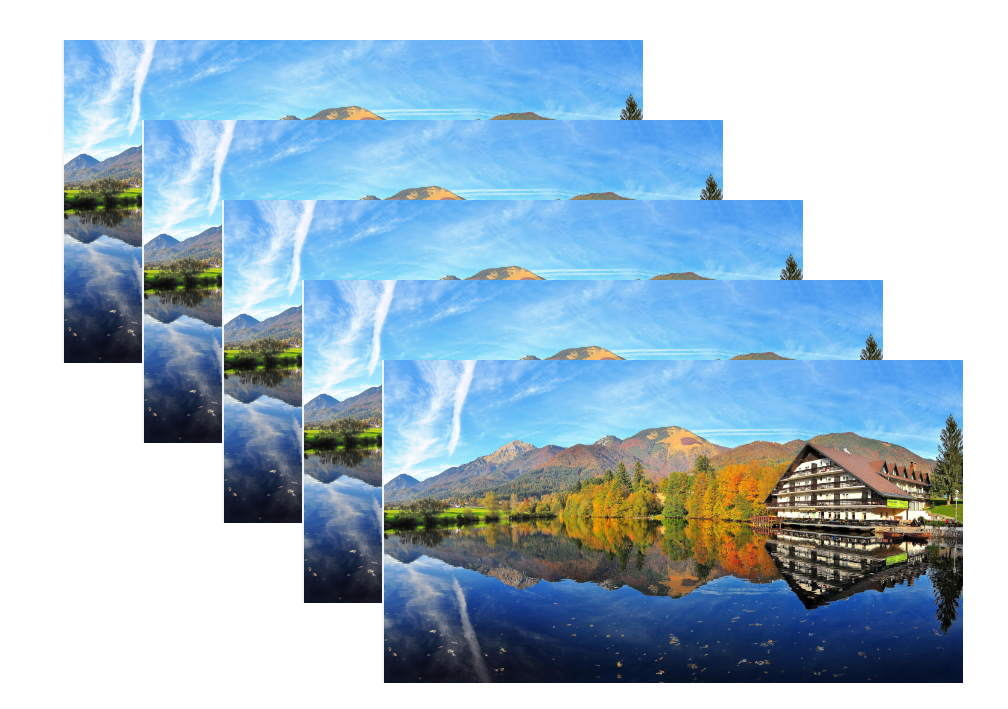
};

</script>

</body>

</html>

运行效果为：



例子 2，在画布上定位图像，并规定图像的宽度和高度：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="1314px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//创建一张图片

var image = new Image();

//设置图片的路径

image.src = 'a.png';

//当图片加载完成后

image.onload = function () {

//绘制图片

ctx.drawImage(image, 100, 100, 150, 150);

};

</script>

</body>

</html>

运行效果为：



例子 3，剪切图像，并在画布上定位被剪切的部分：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<!--添加canvas元素，设置画布的大小-->

<canvas id="mycanvas" width="1314px" height="1314px">

对不起，你的浏览器不支持canvas

</canvas>

<script type="text/javascript">

//获取canvas元素

var myCanvas = document.getElementById('mycanvas');

//获取Context上下文

var ctx = myCanvas.getContext('2d');

//创建一张图片

var image = new Image();

//设置图片的路径

image.src = 'a.png';

//当图片加载完成后

image.onload = function () {

//绘制图片

ctx.drawImage(image, 100, 100, 150, 150, 150, 150, 150, 150);

};

</script>

</body>

</html>

运行效果为：



## Web Storage

### Web Storage 概述

早些时候，本地存储使用的是 cookie。但是 Web 存储需要更加的安全与快速。这些数据不会被保存在服务器上，但是这些数据只用于用户请求网站数据上。它也可以存储大量的数据，而不影响网站的性能。

HTML5 定义了本地存储规范 Web Storage，提供了两种存储类型 API：localStorage 和 sessionStorage。

先说说 cookie：

1. 大小的限制：cookie 的大小被限制在 4KB。
2. 带宽的限制：只要涉及 cookie 的请求，cookie 数据都会在服务器和浏览器间来回传送。这样无论访问哪个页面，cookie 数据都会消耗网络的带宽。
3. cookie 会频繁的在网络中传送，而且数据在网络中是可见的，因此在不加密的情况下 ，是有安全风险的。

Web Storage 数据存储机制相比于 Cookie 有明显的优势：

* 存储空间的大小一般为 5~10MB，与具体浏览器有关。
* 存储内容仅仅存储在本地客户端，不会被发送到服务器。
* 提供了更丰富、更易用的接口、操作更方便。

### localStorage 方法

特点：持久化的本地存储，除非主动手动删除，否则数据一直不会过期。

#### 方法：

**1.设置**

setItem(key,value)：在本地客户端存储一个字符串类型的数据。

setItem.key=value：也可以像这样直接存储。



保存数据：

localStorage.setItem(key, value);



保存添加数据例子：

//方法1向本地存储中添加一个名为name,值为"syl"的key-value对象

localStorage.setItem('name', 'syl');

//方法2

localStorage['price'] = 1314;

//方法3

localStorage.amount = 520;



注：使用 setItem 方法保存数据时，将第一个参数 key 指定为键名，将第二个参数 value 指定为键值，保存时不允许保存相同的键名，保存后可以修改键值，但不允许改键名（只能重新取键名，然后再保存键值）。

**2.获取**

getItem(key)：读取已存储在本地的数据，获取键值。

localStorage.key：也可以像这样直接获取值。



读取数据：

localStorage.getItem(key);



注：使用 getItem 方法读取数据时，将参数指定为键名，返回键值。

**3.删除**

removeItem(key)：移除已存储在本地数据，通过键名作为参数删除数据。

localStorage.clear()：也可以一次性清除



删除单个数据：

localStorage.removeItem(key);



注：通过 key 删除本地数据。

例子：

<body>

<h1>简单Web留言本</h1>

<textarea id="memo" cols="60" rows="6"></textarea><br />

<input type="button" value="新增留言" onclick="saveStorage('memo');" />

<input type="button" value="清空数据" onclick="clearStorage();" />

<input

type="button"

value="清空最后一个数据"

onclick="clearsingleStorage();"

/>

<hr />

<p id="msg"></p>

<script type="text/javascript">

//savaStorage是一个新增留言的函数

function saveStorage(id) {

//获取textarea的value值

var data = document.getElementById(id).value;

//获取当前时间

var time = new Date().toUTCString();

//将当前时间作为键名，textarea的value值（也就是用户输入的值）的值作为键值

localStorage.setItem(time, data);

//显示留言

showMsg('msg');

}

//showMsg是一个显示留言的函数

function showMsg(id) {

var result = '<table border="1">';

//遍历本地储存数据

for (var i = 0; i < localStorage.length; i++) {

//获取key值

var key = localStorage.key(i);

//获取value值

var value = localStorage.getItem(key);

//显示数据

result += '<tr><td>' + value + '</td><td>' + key + '</td></tr>';

}

result += '</table>';

var target = document.getElementById(id);

target.innerHTML = result;

}

//显示留言

showMsg('msg');

//clearStorage是一个清空留言的函数

function clearStorage() {

//清空数据

localStorage.clear();

//显示留言

showMsg('msg');

}

//clearsingleStorage是一个删除单个数据的函数

function clearsingleStorage() {

localStorage.removeItem(localStorage.key(localStorage.length - 1));

//显示留言

showMsg('msg');

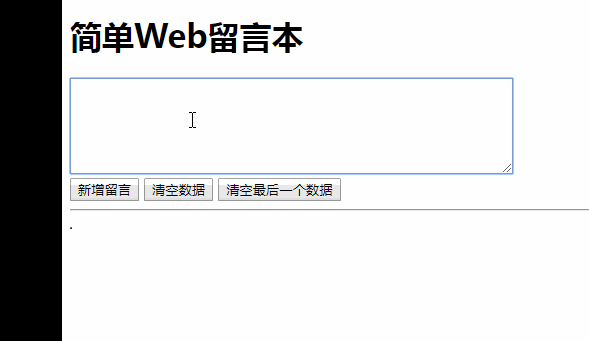
}

</script>

</body>



运行效果如下所示：



### sessionStorage 方法

直接关闭页面，再重新运行查看一下效果。

sessionStorage 方法将数据保存在 session 对象中，所谓 session 直译过来就是会话，再通俗一点讲就是指用户在浏览某个网站时，从进入网站到关闭浏览器的这段时间，session 对象可以用来保存在这段时间内所要求保存的任何数据。我们称之为会话级别的本地存储。

注：当浏览器窗口被关闭时，session 对象保存的数据会被删除。

sessionStorage 方法的使用与 localStorage 方法的使用类似，这里不在累赘：

例子：

<body>

<script type="text/javascript">

if (sessionStorage.pagecount) {

sessionStorage.pagecount = Number(sessionStorage.pagecount) + 1;

} else {

sessionStorage.pagecount = 1;

}

document.write('你刷新了本页面 ' + sessionStorage.pagecount + ' 次');

</script>

<p>刷新页面看看效果。</p>

<p>关闭浏览器再运行看看效果</p>

</body>

注：上面的案例 sessionStorage 改成 localStorage 也就可以简单的记录自己的网站的浏览人数了，当然这只是我们自己写简单网站，自己看自己浏览次数的使用方法。请大家自行尝试一下 sessionStorage 保存

## HTML5 文件上传

### HTML5 文件上传概述

在之前我们操作本地文件都是使用 flash、silverlight 或者第三方的 activeX 插件等技术，由于使用了这些技术后就很难进行跨平台、或者跨浏览器、跨设备等情况下实现统一的表现，从另外一个角度来说就是让我们的 web 应用依赖了第三方的插件，而不是很独立。 在 HTML5 标准中，默认提供了操作文件的 API 让这一切直接标准化。有了操作文件的 API，让我们的 Web 应用可以很轻松的通过 JS 来控制文件的读取、写入、文件夹、文件等一系列的操作。

### 如何实现文件上传

在 HTML4 标准中文件上传控件只接受一个文件，而在新标准中，只需要设置 multiple，就支持多文件上传。按住 Ctrl 或者 Shift 即可选择多个文件。

<input type="file" class="file" multiple />

#### 获取文件信息

选中文件通过 HTMLInputElement.files 属性返回，返回值是一个 FileList 对象，这个对象是一个包含了许多 File 文件的列表。比如我们首先运行上面例子的代码，然后随便上传一个文件（这里上传的是一个名为 700.png 的图片），然后按 F12 进入控制台，输入代码:

file.files;

运行效果为：



从运行效果我们可以看出每个 File 对象包含了以下信息：

* lastModified 表示 UNIX timestamp 形式的最后修改时间。
* lastModifiedDate 表示 Date 形式的最后修改时间。
* name 表示文件名。只读字符串，只包含文件名称，不包含任何路径。
* size 表示文件的字节大小，只读的 64 位整数。
* type 表示文件的 MIME 类型。当类型不确定时为 ""。
* webkitRelativePath 此处为空，当在 input 上加 webkitdirectory 属性时，用户可选择文件夹，此时 webkitRelativePath 表示文件夹中文件的相对路径。

注：FileList 对象由 DOM 提供，列出了所有用户选择的文件，每一个代表了一个 File 对象。你可以通过检查文件列表的 length 属性决定用户可以选则多少文件。各个 File 对象可以方便地通过访问文件列表来获取，像访问数组那样。

#### 限制文件的上传类型

如果我们需要限制用户上传文件的类型，比如某处我们只希望能够上传图片，那么我们可以使用 input 的 accept 属性，accept 属性接受一个逗号分隔的 MIME 类型字符串。比如：

<!-- 表示只接受 png 图片 -->

accept="image/png" 或 accept=".png"

<!-- 表示接受PNG/JPEG 文件. -->

accept="image/png, image/jpeg" 或 accept=".png, .jpg, .jpeg"

<!-- 表示接受任何图片文件类型. -->

accept="image/\*"

<!-- 表示接受任何 MS Doc 文件类型. -->

accept=".doc,.docx,.xml,application/msword,application/vnd.openxmlformats-officedocument.wordprocessingml.document"

例子：

<form>

<div>

<label for="profile\_pic">上传图片</label>

<input

type="file"

id="profile\_pic"

name="profile\_pic"

accept=".jpg, .jpeg, .png"

/>

</div>

<div>

<button>Submit</button>

</div>

</form>

### 文件读取

以上 File 对象只获取到了对文件的描述信息，但没有获得文件中的数据。HTML5 中提供了 FileReader 对象允许 Web 应用程序异步读取存储在用户计算机上的文件（或原始数据缓冲区）的内容。

首先创建一个 FileReader 实例：

var reader = new FileReader();



#### 属性

FileReader.error 属性表示在读取文件时发生的错误，只读。语法为：

var error = instanceOfFileReader.error;



FileReader.readyState 属性表示 FileReader 状态的数字，只读。取值如下：

| **常量名** | **值** | **描述** |
| --- | --- | --- |
| EMPTY | 0 | 还没有加载任何数据. |
| LOADING | 1 | 数据正在被加载. |
| DONE | 2 | 已完成全部的读取请求. |

语法为：

var state = instanceOfFileReader.readyState;



FileReader.result 属性表示文件的内容。该属性仅在读取操作完成后才有效，数据的格式取决于使用哪个方法来启动读取操作，只读。

#### 事件处理

| **FileReader.onabort** | **处理abort事件。该事件在读取操作被中断时触发。** |
| --- | --- |
| FileReader.onerror | 处理error事件。该事件在读取操作发生错误时触发。 |
| FileReader.onload | 处理load事件。该事件在读取操作完成时触发。 |
| FileReader.onloadstart | 处理loadstart事件。该事件在读取操作开始时触发 |
| FileReader.onloadend | 处理loadend事件。该事件在读取操作结束时（要么成功，要么失败）触发。 |
| FileReader.onprogress | 处理progress 事件。该事件在读取 Blob 时触发 |

#### 方法

| **FileReader.abort()** | **中止读取操作。在返回时，readyState属性为DONE。** |
| --- | --- |
| FileReader.readAsArrayBuffer() | 开始读取指定的  Blob 中的内容, 一旦完成, result 属性中保存的将是被读取文件的  ArrayBuffer  数据对象 。 |
| FileReader.readAsDataURL() | 开始读取指定的 Blob 中的内容。一旦完成，result 属性中将包含一个 data: URL 格式的字符串以表示所读取文件的内容。 |
| FileReader.readAsText() | 开始读取指定的 Blob 中的内容。一旦完成，result 属性中将包含一个字符串以表示所读取的文件内容。 |

我们常用的是上传一个图片并显示出来和上传文本显示文本，因为这里只对 FileReader.readAsDataURL() 和 FileReader.readAsText() 方法进行举例说明，想要了解更多的知识可以访问 [MDN FileReader](https://developer.mozilla.org/zh-CN/docs/Web/API/FileReader)

请上传 .txt 、.jpg 、.png 文件。

例子：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>文件读取</title>

<script type="text/javascript">

//检查浏览器是否支持FileReader

if (typeof FileReader == 'undified') {

alert('您老的浏览器不行了！');

}

function showDataByURL() {

var resultFile = document.getElementById('fileDemo').files[0];

if (resultFile) {

//创建一个 FileReader 实例

var reader = new FileReader();

//读取文件数据

reader.readAsDataURL(resultFile);

/\*读取的过程就相当于 加载过程 \*/

/\*读取完毕 预览 \*/

reader.onload = function (e) {

var urlData = this.result;

document.getElementById('result').innerHTML +=

"<img src='" + urlData + "' alt='" + resultFile.name + "' />";

};

}

}

function showDataByText() {

var resultFile = document.getElementById('fileDemo').files[0];

if (resultFile) {

var reader = new FileReader();

reader.readAsText(resultFile, 'gb2312');

reader.onload = function (e) {

var urlData = this.result;

document.getElementById('result').innerHTML += urlData;

};

}

}

</script>

</head>

<body>

<input type="file" name="fileDemo" id="fileDemo" multiple="multiple" />

<input

type="button"

value="显示图片"

id="readAsDataURL"

onclick="showDataByURL();"

/>

<input

type="button"

value="显示文本"

id="readAsText"

onclick="showDataByText();"

/>

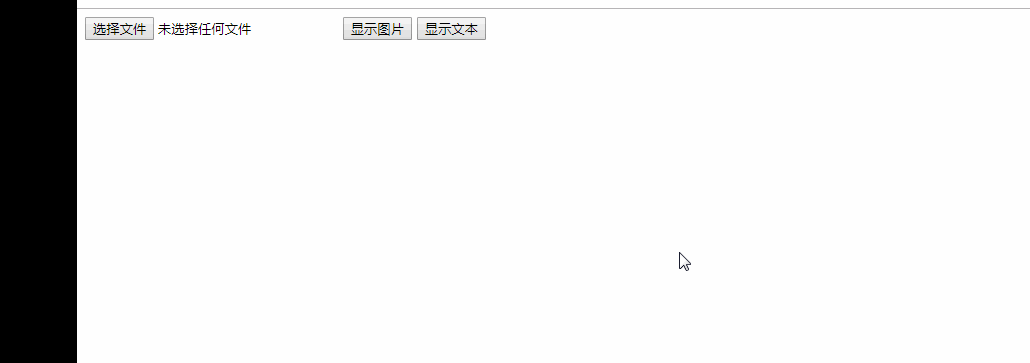
<div id="result"></div>

</body>

</html>



运行效果为：



# JavaScript

## 什么是 JavaScript

JavaScript，通常缩写为 JS，是一种高级的，解释执行的编程语言。JavaScript 是一门基于原型、函数先行的语言，是一门多范式的语言，它支持面向对象编程，命令式编程，以及函数式编程。它提供语法来操控文本、数组、日期以及正则表达式等，不支持 I/O，比如网络、存储和图形等，但这些都可以由它的宿主环境提供支持。它已经由 ECMA（欧洲计算机制造商协会）通过 ECMAScript 实现语言的标准化。它被世界上的绝大多数网站所使用，也被世界主流浏览器（Chrome、IE、Firefox、Safari、Opera）支持。

虽然 JavaScript 与 Java 这门语言不管是在名字上，或是在语法上都有很多相似性，但这两门编程语言从设计之初就有很大的不同，JavaScript 的语言设计主要受到了 Self（一种基于原型的编程语言）和 Scheme（一门函数式编程语言）的影响。在语法结构上它又与 C 语言有很多相似，例如 if 条件语句、while 循环、switch 语句、do-while 循环等。

在客户端，JavaScript 在传统意义上被实现为一种解释语言，但在最近，它已经可以被即时编译（JIT）执行。随着最新的 HTML5 和 CSS3 语言标准的推行，它还可用于游戏、桌面和移动应用程序的开发和在服务器端网络环境运行，如 Node.js。

注：定义来自于维基百科。

**JavaScript 的组成**

* ECMAScript：JavaScript 的语法标准。
* DOM：JavaScript 操作网页上的元素的 API。
* BOM：JavaScript 操作浏览器的部分功能的 API。

**JavaScript 的特点**

* 可以使用任何文本编辑工具编写，然后使用浏览器就可以执行程序。
* 是一种解释型脚本语言：代码不进行预编译，从上往下逐行执行，不需要进行严格的变量声明。
* 主要用来向 HTML 页面添加交互行为。

## JavaScript 范例

实验需新建一个 test.html 文件，用于编写代码。后续的例子中，将不再提醒建立文件，大家根据个人需求自行创建对应的 html 文件，并完成代码练习：

由于实验楼使用是 WebIDE 的在线环境，所以有不熟悉对同学请阅读下：[实验楼 WebIDE 使用指南](https://www.lanqiao.cn/library/shiyanlou-docs/feature/webide)，前端开发 部分的内容。

首先来看看范例代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

console.log('hello world');

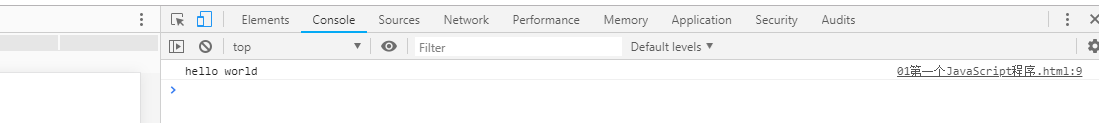
</script>

</body>

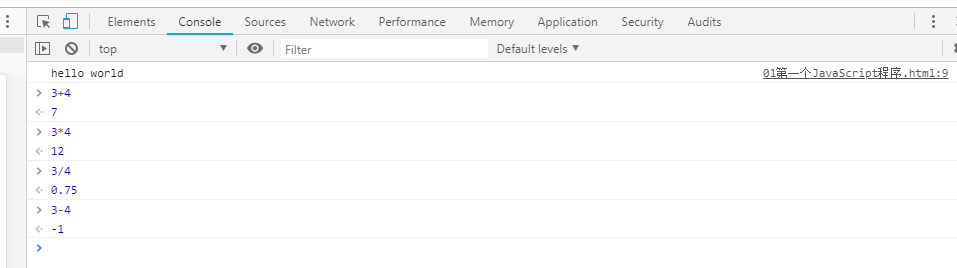
</html>



上述代码的含义就是在我们的控制台打印一句话：hello world。首先教大家怎么查看：将上述代码复制到一个 html 文件中，然后在浏览器中运行，点击 F12，再点击控制台上的 Console，即可查看。如下图所示：



值得注意的是我们可以直接在控制台上输入 JavaScript 代码，然后点击 enter 让其执行。如下图所示我们执行几行简单的加减乘除：



通过上面的代码，我们可以看出 JavaScript 代码是放在 <script>……</script> 标签里，而包含 JavaScript 代码的 script 标签，我们可以放在 <body>……</body> 标签里，也可以放在 <head>……</head> 标签里。比如上述范例也可以这样写：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<script>

console.log('hello world');

</script>

</head>

<body></body>

</html>



执行结果没有什么区别，不同的是执行顺序，简单的来说，放在前面的会先执行。此外，和 CSS 引入相类似，JavaScript 也可以通过外部引入。首先我们需要创建一个扩展名为 .js 的文件，然后在 html 页面中引入它。同样的拿上述范例来修改，我们首先创建一个叫 test.js，名字可以自己取，但是扩展名一定要是 .js，只有这样才能够识别包含 JavaScript 代码的文件，然后在里面写上我们的 JavaScript 代码：

console.log('hello world');



在 html 文件中写上如下代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script src="test.js"></script>

</body>

</html>

值得注意的是 test.js 文件要和你的 html 文件在同一目录下才能用上面的方式引用，否则的话需要使用绝对路径来引入 js 文件，具体引入需要根据实际情况灵活运用了。前两种方式都是直接把 JavaScript 代码放在 HTML 中，在页面加载的同时，那些 JavaScript 的代码就被解析了。而把 JavaScript 代码放在外部文件中，只有在事件被触发，需要该段 JavaScript 代码时，才调用执行。这样做有个好处，当页面比较复杂的时候，把大量的 JavaScript 代码放到外部文件，只有在需要的时候才执行，那么会明显地加快页面加载速度，而且实现结构化分离，也便于我们维护自己的代码，所以建议大家养成外部引入的方式来写我们的 JavaScript 代码。

## 变量

**变量是什么**

在计算机中，数据都存在内存中。而一个变量，就是一个用于存放数值的容器，每个变量存放的数值是可变的，每个变量都有其独有的名字，每个变量都占有一段内存。

注：变量不是数值本身，变量仅仅是一个用于储存数值的容器。

**声明变量**

通过 var 关键字来声明变量，比如：

var name = '实验楼';



上述代码声明了一个名为 name 的变量，并赋值为“实验楼”。注意此处的等于符号（=）为赋值符号，不是我们传统意义上理解的等号。

变量的命名规则如下：

* 变量名必须以字母、下划线 “\_”、美元符号 “$” 开头，不能以数字开头。
* 变量可以包含字母、数字、下划线和美元符号。
* 不能使用 JavaScript 中的关键字做为变量名。
* 变量名不能有空格。
* 变量名对大小写敏感，比如：name 和 Name 就是两个完全不同的变量。

另外在 JavaScript 中，变量也可以不作声明，而在使用时再根据数据的类型来确其变量的类型，如：

x = 50; // 变量 x 为整数



**变量类型**

* Number：你可以在变量中存储数字，不论这些数字是 10（整数），或者是 3.1415926（浮点数）。

var x1 = 10;

var x2 = 3.1415926;



* String：存储字符（比如 "shiyanlou"）的变量，字符串可以是引号中的任意文本，你可以使用单引号或双引号，也可以在字符串中使用引号，只要不匹配包围字符串的引号即可：

var carname = 'shiyanlou';

var carname = 'shiyanlou';

var answer = "I Love 'shiyanlou'";

var answer = 'I Love "shiyanlou"';



* Boolean：布尔类型的值有两种：true 和 false。通常被用于在适当的代码之后，测试条件是否成立，后续会讲到。
* Array：数组是一个单个对象，其中包含很多值，方括号括起来，并用逗号分隔。后续我们将会对数值进行详细的讲解，此处看两个简单的数值例子：

var myNameArray = ['Tom', 'Bob', 'Jim'];

var myNumberArray = [10, 15, 20];



* Object：对象类型。同样的我们会在后续的课程中详细讲解什么是对象，此处先看一个简单的例子：

var student = { name: 'Tom', age: 18 };



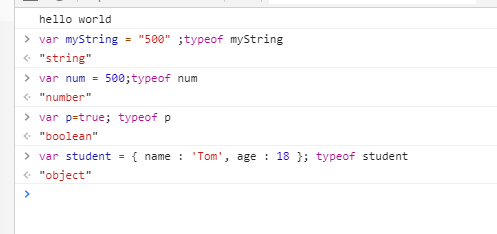
**动态类型**

JavaScript 是一种“动态类型语言”，这意味着不同于其他一些语言（如 C、Java），你不需要指定变量将包含什么数据类型（例如 number 或 string），全部用 var 关键字声明就是了。比如如果你声明一个变量并给它一个带引号的值，浏览器就会知道它是一个字符串：

var myString = 'Hello';



值得注意的就是引号中如果是一个数字，它依然是 string 类型的。我们可以在控制台中通过 typeof 函数，来查看我们声明的变量是什么类型的。



**注释**

单行注释：用来描述下面一个或多行代码的作用。单行注释快捷键：Ctrl + /。

// 这是一个变量

var name = 'zhangsan';



多行注释：用来注释多条代码。多行注释快捷键：Ctrl + Shift + /。

/\*

var name = "zhangsan";

var age = 18;

console.log(name, age);

\*/

## 数组

### 数组概述

通过我们前面所学的我们可以知道如果我们要保存一个数据，我们可以用 var 关键字去定义一个变量，但是如果数据很多呢？难道我需要一个个去定义？比如一个班上很多人，我们 var name1 = "张三"; var name2 = "李四"; ... 一直这样定义下去？显然这是不科学的，而这也由此引出了我们的数组。首先我们来看看维基百科对于数组的定义：

在计算机科学中，数组数据结构（英语：array data structure），简称数组（英语：Array），是由相同类型的元素（element）的集合所组成的数据结构，分配一块连续的内存来存储。利用元素的索引（index）可以计算出该元素对应的存储地址。

最简单的数据结构类型是一维数组。例如，索引为 0 到 9 的 32 位整数数组，可作为在存储器地址 2000，2004，2008，...2036 中，存储 10 个变量，因此索引为 i 的元素即在存储器中的 2000+4×i 地址。数组第一个元素的存储器地址称为第一地址或基础地址。

#### 数组语法

var myarray = new Array(1, 2, 3, 4, 5); // 创建数组同时赋值

// or

var myarray = [1, 2, 3, 4, 5]; // 直接输入一个数组（称“字面量数组”）



注：我们可以用上述两种方法创建数组，myarray 指的是定义的数组名，可以自己取，数组储存的数据可以是任何类型（数字，字符，布尔值等）。每一个值都有一个索引值，从 0 开始。比如：

var color = ['red', 'green', 'blue', 'yellow'];

color[0]; // returns "red"

color[1]; // returns "green"

color[2]; // returns "blue"

color[3]; // returns "yellow"

color[4]; // returns undefined



#### 多维数组

多维数组就是数组中还包含数组，我们可以一层一层的来看。比如：

var student = [

['张三', '男', '18'],

['李四', '女', '20'],

];

student[0][2]; // returns "18"

### 操作数组

#### 修改数组

修改数组中的元素内容也很简单，直接为它提供新值就可以了。比如：

var color = ['red', 'green', 'blue', 'yellow'];

color[0] = 'black';

color; // returns ["black", "green", "blue", "yellow"]



#### 获取数组长度

同样的我们使用 length 来获取数组的长度。比如：

var color = ['red', 'green', 'blue', 'yellow'];

color.length; // returns 4



#### 数组和字符串之间的转换

通过 split() 方法，将字符串转换为数组。比如：

'1:2:3:4'.split(':'); // returns ["1", "2", "3", "4"]

'|a|b|c'.split('|'); // returns ["", "a", "b", "c"]



相反的我们通过 join() 方法将数组转换为字符串。比如：

['1', '2', '3', '4'].join(':'); // returns "1:2:3:4"

['', 'a', 'b', 'c'].join('|'); // returns "|a|b|c"



注：我们同样可以使用 toString() 方法将数组转换为字符串，但是 join() 方法可以指定不同的分隔符，而 toString() 方法只能是逗号。

#### 添加和删除数组项

在数组尾部添加一个或多个元素，使用 push() 方法。比如：

var arr = ['1', '2', '3', '4'];

arr.push('5', '6');

arr; // returns ["1", "2", "3", "4", "5", "6"]

使用 pop() 方法将删除数组的最后一个元素，把数组长度减 1，并且返回它删除的元素的值。如果数组已经为空，则 pop() 不改变数组，然后返回 undefined 值。比如：

var arr = ['1', '2', '3', '4'];

arr.pop(); // returns 4

arr; // returns ["1", "2", "3"]

var arr1 = [];

arr1.pop(); // returns undefined

arr1; // returns []

unshift() 和 shift() 从功能上与 push() 和 pop() 完全相同，只是它们分别作用于数组的开始，而不是结尾。大家可以直接在控制台自行尝试一番。

注：数组的方法其实还有很多，我们将在后续的课程中一一为大家讲解。

## Null 和 Undefined

null 和 undefined 都表示无，但是也有一些区别。现在控制台上执行以下语句：



这里需要说明的是：== 是相等操作符，比较值是否相等，如果相等输出为 true，否则为 false。=== 是全等操作符，比较值和类型是否都相等，如果都相等输出为 true，否则为 false。通过我们在控制台中的实践可以发现，null 和 undefined 的值不等于 0，它们的值相等，但是类型不相等。undefined 表示所有没有赋值变量的默认值，而 null 则表示一个变量不再指向任何对象地址。

## 字符串

在前面的变量章节中，我们已经简单讲过字符串的基础知识，这里我们再拓展一下。我们前面讲过我们可以使用单引号或双引号，也可以在字符串中使用引号，只要不匹配包围字符串的引号即可。比如：

var carname = 'shiyanlou';

var carname = 'shiyanlou';

var answer = "I Love 'shiyanlou'";

var answer = 'I Love "shiyanlou"';



下面的代码将会出现错误，因为它会混淆浏览器和字符串的结束位置:

var x1 = 'I've got no right to take my place...';



聪明的你可能会觉得这样不行，我们就换种方法，比如：

var x1 = 'I have got no right to take my place...';



或者：

var x1 = "I've got no right to take my place...";



没错这样做都是可行的方法，但是其实我们还有另外一种方法，使用转义符号。转义字符意味着我们对它们做一些事情，以确保它们在文本中被认可，而不是代码的一部分。在 JavaScript 中，我们通过在字符之前放一个反斜杠来实现这一点。试试这个:

var x1 = 'I\'ve got no right to take my place...';



常用的转义符：



**连接字符串**

我们通过 “+” 连接字符串，比如在控制台中输入下面的代码：

var one = 'Hello,jack.';

var two = "I'm rose";

result = one + two;



最后控制台显示结果为："Hello,jack.I'm rose"。

如果我们在控制台输入的是 "jack"+18 思考一下会报错吗？要不再自己动手实践看看结果？结果是：浏览器很聪明的把数字转换成了字符串，最后结果为 "jack18"。另外输入 "20"+"19"，最后的结果也是字符串 "2019"。

**字符串转换**

我们可以通过 toString() 方法把数字转换成字符串。

var myNum = 123;

var myString = myNum.toString();

typeof myString;



也可以通过 Number() 对象把传递给它的字符串类型的数字转换为数字。

var myString = '123';

var myNum = Number(myString);

typeof myNum;



注：如果传递的不是纯数字的字符串，则返回的不是数字，而是 NaN（not a number）。

**获取字符串的长度**

通过 length 属性获取字符串的长度，结果返回一个数字。比如：

var myString = 'hello world ';

myString.length;



上述代码在控制台中运行的结果为：12（除了字母还有两个空格）。

如果你要查找这个字符串的第一个字符，你可以这么做：

myString[0];



同样的你也可以查找到其他的字符，比如第五个字符，就是 myString[4]，这是因为电脑是从 0 开始，而不是 1，因此我们都要执行减一操作。

**在字符串中查找子字符串并提取它**

1. 有时候我们需要判断一个较长的字符串里面是否存在一个我们指定的较小的字符串，就比如我们要查找一段话里面是否包含一个词或者一个字，这个时候我们可以使用 indexof() 方法来完成，更详细的语法为：

str.indexOf(searchValue, fromIndex);



str 指的是我们需要查的较长的字符串，searchValue 表示我们指定的较小的字符串，fromIndex 表示调用该方法的字符串中开始查找的位置，是一个可选的任意整数值，也可以不写，默认是 0 表示从头开始查，fromIndex < 0 和 fromIndex = 0 是一样的效果，表示从头开始查找整个字符串。如果 fromIndex >= str.length，则该方法的返回值为 -1。这里有个特殊的情况：就是如果被查找的字符串（searchValue）是一个空字符串，那么当 fromIndex <= 0 时返回 0，0 < fromIndex <= str.length 时返回 fromIndex，fromIndex > str.length 时返回 str.length。这样说你可能不太明白，我们来实践一下看看实际效果：

'Blue Sky'.indexOf('Blue'); // returns 0

'Blue Sky'.indexOf('Ble'); // returns -1

'Blue Sky'.indexOf('Sky', 0); // returns 5

'Blue Sky'.indexOf('Sky', -1); // returns 5

'Blue Sky'.indexOf('Sky', 5); // returns 5

'Blue Sky'.indexOf('Sky', 9); // returns -1

'Blue Sky'.indexOf('', 0); // returns 0

'Blue Sky'.indexOf('', 5); // returns 5

'Blue Sky'.indexOf('', 9); // returns 8



注：返回值指的是指定值第一次出现的索引，如果没有找到返回 -1。indexOf() 方法区分大小写，比如：

'Blue Sky'.indexOf('blue'); // returns -1

'Blue Sky'.indexOf('Blue'); // returns 0



1. 当你知道字符串中的子字符串开始的位置，以及想要结束的字符时，slice() 方法可以用来提取它。比如：

'Blue Sky'.slice(0, 3); // returns "Blu"



注：slice(strat，end)，第一个参数 start 是开始提取的字符位置，第二个参数 end 是提取的最后一个字符的后一个位置。所以提取从第一个位置开始，直到但不包括最后一个位置。另外第二个参数也可以不写，不写代表某个字符之后提取字符串中的所有剩余字符。比如：

'Blue Sky'.slice(2); // returns "ue Sky"



**转换大小写**

字符串方法 toLowerCase() 和 toUpperCase() 字符串并将所有字符分别转换为小写或大写。比如：

var string = 'I like study';

string.toLowerCase(); // returns "i like study"

string.toUpperCase(); // returns "I LIKE STUDY"



**替换字符串的某部分**

可以使用 replace() 方法将字符串中的一个子字符串替换为另一个子字符串。比如：

var string = 'I like study';

string.replace('study', 'sleep'); // returns "I like sleep"

注意这样只能替换第一个出现的字符串，如果字符串是类似 I like study study，那么第二个 study 不会被替换。

此时可以使用全局替换方法。

var string = 'I like study study';

string.replace(/study/g, 'sleep');

注：字符串的操作方法其实还有很多，我们将在后续的课程中再为大家作深入讲解。

### 转换成字符串

1. 几乎每个值都有 toString() 方法。比如在控制台输入以下代码：

var num = 8;

var numString = num.toString();

numString; // returns "8"

var result = true;

var resultString = result.toString();

resultString; // returns "true"



数值类型的 toString()，可以携带一个参数，输出对应进制的值。比如：

var num = 16;

console.log(num.toString()); // "16" 默认是10进制

console.log(num.toString(10)); // "16"

console.log(num.toString(2)); // "10000"

console.log(num.toString(8)); // "20"

console.log(num.toString(16)); // "10"



1. String() 函数。比如在控制台中输入以下代码：

var num1 = 8;

var num2 = String(num1);

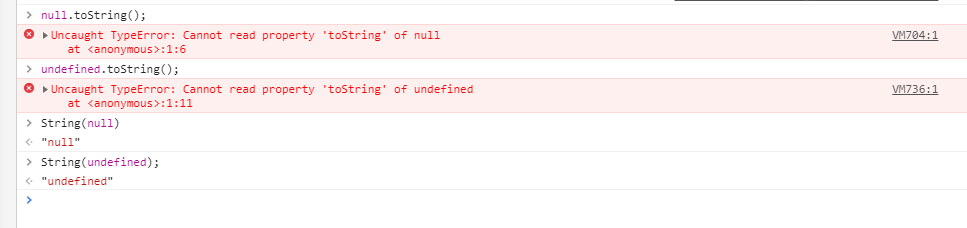
num2; // returns "8"

var result = true;

var result1 = String(result);

result1; // returns "true"

注：因为有的值没有 toString() 方法，所以需要用 String()，比如 null 和 undefined。如下所示：



1. 使用拼接字符串。比如在控制台中输入以下代码：

var num1 = 16;

var num2 = num1 + '';

num2; // returns "16"

### 转换成数值类型

1. Number() 可以把任意值转换成数值，如果要转换的字符串中有一个不是数值的字符，返回 NaN（not a number）。比如在控制台中依次输入以下代码：

var num1 = Number(true);

num1; // true 返回 1，false 返回 0

var num2 = Number(undefined);

num2; // 返回 NaN

var num3 = Number(null);

num3; // 返回 0

var num4 = Number('syl');

num4; // 返回 NaN

var num5 = Number(' ');

num5; // 如果是空字符串返回 0

var num6 = Number(123);

num6; // 返回123，如果是数字型的字符，返回数字

var num7 = Number('123abc');

num7; // 返回 NaN



1. parseInt() 把字符串转换成整数。比如在控制台中依次输入以下代码：

var num1 = parseInt('12.3abc');

num1; // 返回 12，如果第一个字符是数字会解析知道遇到非数字结束，只取整，不是约等于

var num2 = parseInt('abc123');

num2; // 返回 NaN，如果第一个字符不是数字或者符号就返回 NaN

var num3 = parseInt('');

num3; // 空字符串返回 NaN，但是 Number("")返回 0

var num4 = parseInt('520');

num4; // 返回 520

var num5 = parseInt('0xA');

num5; // 返回 10



另外值得注意的是，parseInt() 可以传递两个参数，第一个参数是要转换的字符串，第二个参数是要转换的进制。大家可以自行尝试一下。

1. parseFloat() 把字符串转换成浮点数。写法和 parseInt() 相似，主要有以下几个不同点：

* parseFloat 不支持第二个参数，只能解析 10 进制数。
* 如果解析的内容里只有整数，解析成整数。

例子：

parseFloat('10'); // returns 10

parseFloat('10.000'); // returns 10

parseFloat('10.01'); // returns 10.01

parseFloat('10'); // returns 10

parseFloat('10 hours'); // returns 10

parseFloat('aaa 10'); // returns NaN

1. 执行减 0 操作。比如：

var n = '10';

var m = n - 0;

m; // returns 10

值得注意的是，如果该字符串不是纯粹的数字字符串，那么它执行减 0 操作后，虽然变成了一个数字类型，但是返回值为 NaN。比如：

var n = 'abc';

var m = n - 0;

m; // returns NaN

typeof m; // returns "number"

### 转换成布尔类型

1. 使用 Boolean() 函数。比如：

Boolean(123); // returns true

Boolean('abc'); // returns true

Boolean(null); // returns false

Boolean(undefined); // returns false

Boolean(NaN); // returns false

Boolean(0); // returns false

Boolean(''); // returns false

Boolean(false); // returns false

Boolean('false'); // returns true

Boolean(-1); // returns true

1. 流程控制语句会把后面的值隐式转换为布尔类型。比如：

var message;

if (message) {

//会自动把 message 转换成 false，最后打印结果为：我很好

console.log('你好啊');

} else {

console.log('我很好');

}

## 函数

### 创建函数

#### 函数声明创建函数

语法为：

function functionName(parameters) {

// 执行的代码

}



例子：

function f(a, b) {

console.log(a + b);

} // 创建一个名为 f 的函数，它有两个形参 a，b

f(2, 3); // 调用函数 f，传入实参 2 和 3，最终运行结果为在控制台上打印出 5



#### 函数表达式创建函数

JavaScript 函数可以通过一个表达式定义。函数表达式可以存储在变量中。

语法为：

var functionName = function (parameters) {

// 执行的代码

};



把函数声明创建函数的例子改写为用函数表达式创建函数：

var f = function (a, b) {

console.log(a + b);

};

f(2, 3);



#### 函数声明和函数表达式的区别

* 函数声明

// 此处的代码执行没有问题，JavaScript 解析器首先会把当前作用域的函数声明提前到整个作用域的最前面。

f(2, 3);

function f(a, b) {

console.log(a + b);

}



* 函数表达式

// 报错：f is not a function

// 这是因为函数表达式，如同定义其它基本类型的变量一样，只在执行到某一句时也会对其进行解析

f(2, 3);

var f = function (a, b) {

console.log(a + b);

};



#### 函数的参数

* 形参：function f(a, b){return a + b;} // a, b 是形参，占位用，函数定义时形参无值。
* 实参：当我们调用上面的函数时比如 f(2, 3);其中 2 和 3 就是实参，会传递给 a 和 b，最后函数中执行的语句就变成了：return 2 + 3;。

注：在 JavaScript 中，实参个数和形参个数可以不相等。

#### 在 JavaScript 中没有重载

function f(a, b) {

return a + b;

}

function f(a, b, c) {

return a + b + c;

}

var result = f(5, 6);

result; // returns NaN



上述代码中三个参数的 f 把两个参数的 f 覆盖，调用的是三个参数的 f，最后执行结果为 NaN，而不是 11。

#### 在 JavaScript 中函数的返回值

* 如果函数中没有 return 语句，那么函数默认的返回值是：undefined。
* 如果函数中有 return 语句，那么跟着 return 后面的值就是函数的返回值。
* 如果函数中有 return 语句，但是 return 后面没有任何值，那么函数的返回值也是：undefined。
* 函数在执行 return 语句后会停止并立即退出，也就是说 return 语句执行之后，剩下的代码都不会再执行了。
* 当函数外部需要使用函数内部的值的时候，我们不能直接给予，需要通过 return 返回。比如：

var f = function (a, b) {

a + b;

};

console.log(f(2, 3)); // 结果为 undefined



var f = function (a, b) {

return a + b;

};

console.log(f(2, 3)); // 结果为 5



#### 匿名函数

匿名函数就是没有命名的函数，一般用在绑定事件的时候。语法为：

function(){

// 执行的代码

}



例子：

var myButton = document.querySelector('button');

myButton.onclick = function () {

alert('hello');

};

注：将匿名函数分配为变量的值，也就是我们前面所讲的函数表达式创建函数。一般来说，创建功能，我们使用函数声明来创建函数。使用匿名函数来运行负载的代码以响应事件触发（如点击按钮），使用事件处理程序。

#### 自调用函数

匿名函数不能通过直接调用来执行，因此可以通过匿名函数的自调用的方式来执行。比如：

(function () {

alert('hello');

})();

### call、apply、bind

在学习 call()、apply()、bind() 方法之前，我们先来复习一下 this 的指向问题，我们前面说过一个口诀：谁调用 this，它就指向谁。让我们先来看一个例子：

function foods() {}

foods.prototype = {

price: '￥15',

say: function () {

console.log('My price is ' + this.price);

},

};

var apple = new foods();

apple.say(); // My price is ￥15

var orange = new foods();

orange.say(); // My price is ￥15



也就是说上述例子调用 say() 方法，最后打印的结果都是一样的，但是如果我们想打印橘子的价钱是 10 元呢？又不想重新定义 say() 方法。JavaScript 为我们专门提供了一些函数方法用来帮我们更优雅的处理函数内部 this 指向问题。这就是接下来我们要学习的 call()、apply()、bind() 三个函数方法。

#### call

call() 方法调用一个函数, 其具有一个指定的 this 值和分别地提供的参数（参数的列表）。语法为：

fun.call(thisArg, arg1, arg2, ...)

注：

* thisArg 指的是在 fun 函数中指定的 this 的值。如果指定了 null 或者 undefined 则内部 this 指向 window，同时值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的自动包装对象。是一个可选项。
* arg1, arg2, ...指定的参数列表。也是可选项。
* 使用调用者提供的 this 值和参数调用该函数的返回值。若该方法没有返回值，则返回 undefined。
* call() 允许为不同的对象分配和调用属于一个对象的函数/方法。
* call() 提供新的 this 值给当前调用的函数/方法。你可以使用 call() 来实现继承：写一个方法，然后让另外一个新的对象来继承它（而不是在新对象中再写一次这个方法）。

1. 使用 call() 方法调用函数并且指定上下文的 this。前面的例子可以改写成：
2. function foods() {}
3. foods.prototype = {
4. price: '￥15',
5. say: function () {
6. console.log('My price is ' + this.price);
7. },
8. };
9. var apple = new foods();
10. orange = {
11. price: '￥10',
12. };
13. apple.say.call(orange); // My price is ￥10



1. 在一个子构造函数中，你可以通过调用父构造函数的 call() 方法来实现继承。在控制台输入如下代码：
2. function Father(name, age) {
3. this.name = name;
4. this.age = age;
5. }
6. function Son(name, age) {
7. Father.call(this, name, age);
8. this.hobby = 'study';
9. }
10. var S1 = new Son('zhangsan', 18);
11. S1; // Son {name: "zhangsan", age: 18, hobby: "study"}



#### apply

apply() 方法与 call() 方法类似，唯一的区别是 call() 方法接受的是参数，apply() 方法接受的是数组。语法为：

fun.apply(thisArg, [argsArray]);

1. 使用 apply() 方法将数组添加到另一个数组。

例子：

var array = ['a', 'b', 'c'];

var nums = [1, 2, 3];

array.push.apply(array, nums);

array; // ["a", "b", "c", 1, 2, 3]



注：concat() 方法连接数组，不会改变原数组，而是创建一个新数组。而使用 push() 是接受可变数量的参数的方式来添加元素。使用 apply() 则可以连接两个数组。

1. 使用 apply() 方法和内置函数。

例子：

var numbers = [7, 10, 2, 1, 11, 9];

var max = Math.max.apply(null, numbers);

max; // 11



注：直接使用 max() 方法的写法为：Math.max(7, 10, 2, 1, 11, 9);

#### bind

bind() 方法创建一个新的函数（称为绑定函数），在调用时设置 this 关键字为提供的值。并在调用新函数时，将给定参数列表作为原函数的参数序列的前若干项。语法为：

fun.bind(thisArg[, arg1[, arg2[, ...]]])

注：参数 thisArg：当绑定函数被调用时，该参数会作为原函数运行时的 this 指向。当使用 new 操作符调用绑定函数时，该参数无效。参数：arg1，arg2，...表示当目标函数被调用时，预先添加到绑定函数的参数列表中的参数。

我们创建一个简单的绑定函数例子：

var bin = function () {

console.log(this.x);

};

var foo = {

x: 10,

};

bin(); // undefined

var func = bin.bind(foo); // 创建一个新函数把 'this' 绑定到 foo 对象

func(); // 10



我们再来看一个例子：

this.num = 6;

var test = {

num: 66,

getNum: function () {

return this.num;

},

};

test.getNum(); // 返回 66

var newTest = test.getNum;

newTest(); // 返回 6, 在这种情况下，"this"指向全局作用域

// 创建一个新函数，将"this"绑定到 test 对象

var bindgetNum = newTest.bind(test);

bindgetNum(); // 返回 66

var newTest = test.getNum;

newTest();

// 上面这两行代码其实相当于：

var newTest(){

return this.num;

}

// 所以 this 指向的是全局作用域，返回 6。

### 递归

在程序中，递归就是函数自己直接或者间接的调用自己。

例子：计算 1 到 10 之间的整数相加的和：

function foo(n) {

if (n == 0) {

return 0;

} // 临界条件

else {

return n + foo(n - 1);

}

}

var a = foo(10);

a; // 55

注：一定要写临界条件，不然程序无法结束并且会报错。

### 作用域

作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。简单来说，作用域的值就是作用范围，也就是说一个变量或函数在什么地方可以使用，在什么地方不能使用。

#### 块级作用域

在 JavaScript 中是没有块级作用域的。比如：

{

var num = 123;

{

console.log(num);

}

}

console.log(num);



上面的例子并不会报错，而是打印两次 123，但是在其他编程语言中（C#、C、JAVA）会报错，这是因为在 JavaScript 中是没有块级作用域。也就是说，使用 {} 标记出来的代码块中声明的变量 num，是可以被 {} 外面访问到的。

#### 函数作用域

JavaScript 的函数作用域是指在函数内声明的所有变量在函数体内始终是可见的，不涉及赋值。来看个例子：

function test() {

var num = 123;

console.log(num);

if (2 == 3) {

var k = 5;

for (var i = 0; i < 10; i++) {}

console.log(i);

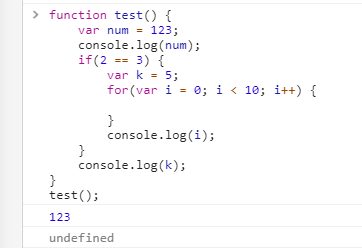
}

console.log(k); // 不会报错，而是显示 undefined

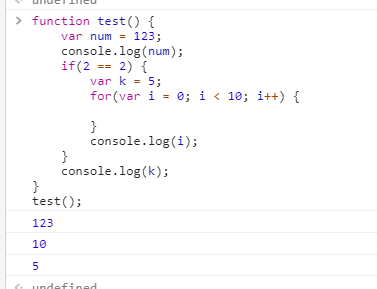
}

test();

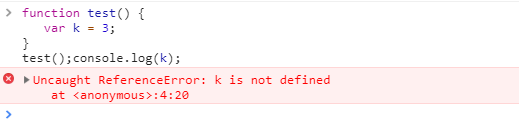




我们把上面的代码 if 语句中的条件改为 2 == 2 来看看效果：



我们再来对比一下如果没有在函数体内会是什么效果：



#### 全局作用域

全局作用域也就是说什么地方都能够访问到。比如我们不用 var 关键字，直接声明变量的话，那这个变量就是全局变量，它的作用域就是全局作用域。使用 window 全局对象来声明，全局对象的属性也是全局变量。另外在所有的函数外部用 var 声明的变量也是全局变量，这是因为内层作用域可以访问外层作用域。

注：

* 内层作用域可以访问外层作用域，反之不行。
* 整个代码结构中只有函数可以限定作用域。
* 如果当前作用规则中有名字了，就不考虑外面的同名变量。
* 作用域规则首先使用提升规则分析。

#### 变量名提升

JavaScript 是解释型的语言，但是它并不是真的在运行的时候完完全全的逐句的往下解析执行。我们来看个例子：

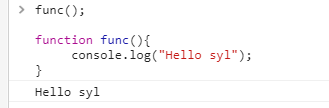
func();

function func() {

console.log('Hello syl');

}

在控制台中运行效果为：



这说明了它并不是完全的逐句往下解析的，否则是会报错的。显然，在执行 func() 之前，引擎就已经解析到了 function func(){}，发生了变量名提升。那么变量名提升是在什么时候发生的呢？JavaScript 引擎在对 JavaScript 代码进行解释执行之前，会对 JavaScript 代码进行预解析，在预解析阶段，会将以关键字 var 和 function 开头的语句块提前进行处理。当变量和函数的声明处在作用域比较靠后的位置的时候，变量和函数的声明会被提升到作用域的开头。也就是说上面的代码，我们可以理解为：

function func() {

console.log('Hello syl');

}

func();

再来看看变量声明的例子：

console.log(num);

var num = 10;



结果可能有些出人意料，但是我们这里说的提示，是声明的提升，也就是说上面的代码，我们可以理解为：

var num; // 这里是声明

console.log(num); // 变量声明之后并未有初始化和赋值操作，所以这里是 undefined

num = 10; // 最终打印结果为 10

下面再来看几个复杂一点的例子。

函数同名的时候：

func();

function func() {

console.log('Hello syl');

}

func();

function func() {

console.log('hi syl');

} // 最终结果打印了两次 hi syl

上面代码相当于：

function func() {

console.log('Hello syl');

}

function func() {

console.log('hi syl');

}

func();

func();

函数变量同名的时候：

console.log(foo);

function foo() {}

var foo = 6;

当出现变量声明和函数同名的时候，只会对函数声明进行提升，变量会被忽略。所以上面的代码相当于：

function foo() {}

console.log(foo);

foo = 6;

我们再来看一种：

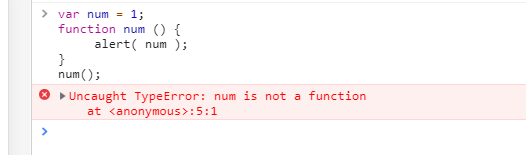
var num = 1;

function num() {

alert(num);

}

num();



上面的代码相当于：

function num() {

alert(num);

}

num = 1;

num();

下面我们来看一个思考题：

var num = 3;

function foo() {

console.log(num);

var num = 4;

console.log(num);

}

foo();

注：大家可以先思考一下，然后根据我们前面所学的变量名提升，试着写一下等价的代码，最后再复制上述代码到控制台中运行看看结果。

上面的代码相当于：

var num = 3;

function foo() {

var num; // 在函数顶部声明了局部变量，覆盖了函数体外同名的全局变量

console.log(num); // 变量存在，但是它的值为 undefined

num = 4; // 将其初始化赋值。

console.log(num); //打印我们期望的值 4

}

### 闭包

闭包是指函数可以使用函数之外定义的变量。

#### 简单的闭包

在 JavaScript 中，使用全局变量是一个简单的闭包实例。比如：

var num = 3;

function foo() {

console.log(num);

}

foo(); //打印 3

#### 复杂的闭包

function f1() {

var num1 = 6;

function f2() {

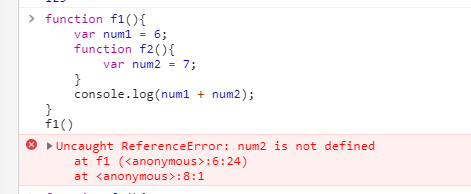
var num2 = 7;

}

console.log(num1 + num2);

}

f1();



在上述代码中函数 f2 能够访问到它外层的变量 num1，但是 f1 是不能访问 f2 中的变量num2，因此我们可以把 num2 作为 f2 的返回值，再把 f2 作为返回值就可以访问到了。

function f1() {

var num1 = 6;

function f2() {

var num2 = 7;

return num2;

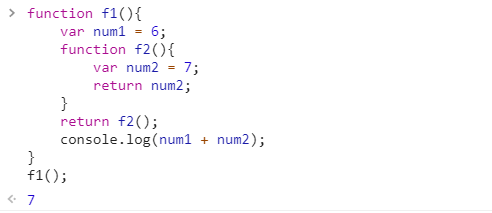
}

return f2();

console.log(num1 + num2);

}

f1();



### arguments 对象

在函数代码中，使用特殊对象 arguments，无需明确指出参数名，我们就能访问它们。第一个参数是 arguments[0]，第二个参数是 arguments[1]，以此类推。比如：

function foo() {

console.log(arguments[0]);

console.log(arguments[1]);

}

foo(2, 3); // 打印 2 3

还可以用 arguments 对象检测函数的参数个数，引用属性 arguments.length 即可。来看一个遍历参数求和的例子：

function add() {

var sum = 0;

for (var i = 0; i < arguments.length; i++) {

sum += arguments[i];

}

return sum;

}

add(); // 0

add(1); // 1

add(1, 2); // 3

add(1, 2, 3); // 6

### Function 对象

用 Function() 对象创建函数的语法如下：

var function\_name = new Function(arg1, arg2, ..., argN, function\_body)

注：每个参数都必须是字符串，function\_body 是函数主体，也就是要执行的代码。

例子：

var add = new Function('a', 'b', 'console.log(a+b);');

add(2, 5); // 打印 7

再看一个例子:

var add = new Function('a', 'b', 'console.log(a+b);');

var doAdd = add;

doAdd(2, 5); // 打印 7

add(2, 5); // 打印 7

在上述例子中，变量 add 被定义为函数，然后 doAdd 被声明为指向同一个函数的指针。用这两个变量都可以执行该函数的代码，并输出相同的结果。因此，函数名只是指向函数的变量，那么我们可以把函数作为参数传递给另一个函数，比如下面的例子：

function addF(foo, b, c) {

foo(b, c);

}

var add = new Function('a', 'b', 'console.log(a+b);');

addF(add, 2, 5); // 打印 7

#### Function 对象的 length 属性

函数属于引用类型，所以它们也有属性和方法。length 属性声明了函数期望的参数个数。

例子：

var add = new Function('a', 'b', 'console.log(a+b);');

console.log(add.length); // 打印 2

#### Function 对象的方法

Function() 对象也有与所有对象共享的 valueOf() 方法和 toString() 方法。这两个方法返回的都是函数的源代码。

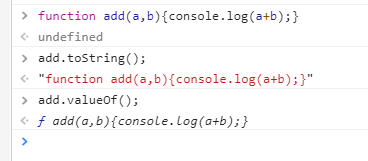
例子：

var add = new Function('a', 'b', 'console.log(a+b);');

add.valueOf();

add.toString();





### 挑战：制作直角三角形

任务要求：自定义一个函数，提示用户输入一个正整数。如果用户输入的非正整数，提示用户输入错误，并返回输入界面让用户输入。直到用户输入一个正确的正整数后，在页面打印出一个由 \* 组成的直角三角形。第一行打印一个 \* ，第二行打印两个 \* ，以此类推下去，最后一行的 \* 个数为用户输入的正整数。比如我们输入 9，最后打效果如下：



参考源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

function star(i) {

for (var j = 1; j <= i; j++) {

for (var k = 1; k <= j; k++) {

document.write('\*');

}

document.write('<br>');

}

}

do {

var n = prompt('请输入一个正整数');

if (Number(n) > 0 && parseInt(n) == parseFloat(n)) {

star(n);

} else {

alert('输入错误，请输入一个正整数');

}

} while (!(Number(n) > 0 && parseInt(n) == parseFloat(n)));

</script>

</body>

</html>

## JSON

JSON（JavaScript Object Notation，JavaScript 对象表示法）是一种由道格拉斯·克罗克福特构想和设计、轻量级的数据交换语言，该语言以易于让人阅读的文字为基础，用来传输由属性值或者序列性的值组成的数据对象。尽管 JSON 是 JavaScript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。

JSON 数据格式与语言无关，脱胎于 JavaScript，但当前很多编程语言都支持 JSON 格式数据的生成和解析。JSON 的官方 MIME 类型是 application/json，文件扩展名是 .json。

注：定义来自维基百科。

下面给出两个简单的 json 示例：

{

"name": "zhangsan",

"age": 18,

"gender": "male"

}



{

"students": [

{ "firstName": "san", "lastName": "zhang" },

{ "firstName": "si", "lastName": "li" },

{ "firstName": "wu", "lastName": "wang" }

]

}

特别需要注意的是：

* JSON 是一种纯数据格式,它只包含属性，没有方法。
* JSON 的属性必须通过双引号引起来。
* JSON 要求两头有 {} 来使其合法。
* 可以把 JavaScript 对象原原本本的写入 JSON 数据，比如：字符串，数字，数组，布尔还有其它的字面值对象。

### 常用内置对象介绍

前面在我们讲数组的方法和字符串的操作的时候，已经讲解了一部分内置对象的内容。本节，我们再来系统的学习一下一些常用的内置对象。

#### Array 对象

1. Array 对象的常用属性：length，获取数组的长度。
2. Array 对象的常用方法：

* concat() 方法用于连接两个或多个数组，并返回结果。语法为：

arrayObject.concat(arrayX, arrayX, ..., arrayX);



例子：

var a = [1, 2, 3];

var b = [4, 5, 6];

var c = ['one', 'two', 'three'];

console.log(a.concat(b, c)); // 打印结果为：[1, 2, 3, 4, 5, 6, "one", "two", "three"]



* join() 方法，将数组转换成字符串。（数组章节有详细介绍，这里不过多的赘述，下面的类似情况同样处理，大家看到这个方法，首先回想一下我们前面所学的知识，如有遗忘，再回去看一看加深记忆）。
* pop() 方法，删除并返回数组的最后一个元素。
* push() 方法，向数组的末尾添加一个或更多元素，并返回新的长度。
* reverse() 方法，颠倒数组的顺序。比如：

var a = [1, 2, 3, 4];

a.reverse();

console.log(a); // a 数组变成：[4, 3, 2, 1]。



* shift() 方法，删除并返回数组的第一个元素。
* unshift() 方法，向数组的开头添加一个或更多元素，并返回新的长度。
* slice() 方法，从某个已有的数组返回选定的元素。语法为：

arrayObject.slice(start, end);

// strat 值是必需的，规定从何处开始选取

// end 值可选，规定从何处结束选取，如果没有设置，默认为从 start 开始选取到数组后面的所有元素



在控制台中输入以下代码：

var a = [1, 2, 3, 4, 5, 6];

a.slice(2, 5); // 结果为[3, 4, 5]，另外需要注意的是该方法不会修改数组，只是返回一个子数组，a 数组还是 [1, 2, 3, 4, 5, 6]



* splice() 方法，删除或替换当前数组的某些项目。语法为：

arrayObject.splice(start, deleteCount, options);

// start 值是必需的，规定删除或替换项目的位置

// deleteCount 值是必需的，规定要删除的项目数量，如果设置为 0，则不会删除项目

// options 值是可选的，规定要替换的新项目

// 和 slice() 方法不同的是 splice() 方法会修改数组



在控制台中输入以下代码：

var a = [1, 2, 3, 4, 5, 6];

a.splice(2, 2, 'abc');

a; // 最终 a 数组变成了[1, 2, "abc", 5, 6]



* sort() 方法，将数组进行排序。语法为：

arrayObject.sort(sortby);

// sortby 是可选的，规定排序顺序，必需是函数。如果没有参数的话，将会按照字母顺序进行排序，更准确的说是按照字符编码（可自行百度了解）的顺序进行排序。如果想按照其他标准进行排序，则需要提供比较函数



例子：

<script>

var arr1 = ["a", "z", "k", "w", "x"];

document.write(arr1 + "<br />");

document.write(arr1.sort()+ "<br />" + "<br />");

var arr2 = [11,55,22,44,66,33];

document.write(arr2 + "<br />");

document.write(arr2.sort() + "<br />" + "<br />");

var arr3 = [1,22,44,6,55,5,2,4,66];

document.write(arr3 + "<br />");

document.write(arr3.sort() + "<br />" + "<br />");

function sortNum1(a, b){

return a - b; // 从小到大排序

}

var arr4 = [1,22,44,6,55,5,2,4,66];

document.write(arr4 + "<br />");

document.write(arr4.sort(sortNum1) + "<br />" + "<br />");

function sortNum2(a,b){

return b - a; // 从大到小排序

}

var arr4 = [1,22,44,6,55,5,2,4,66];

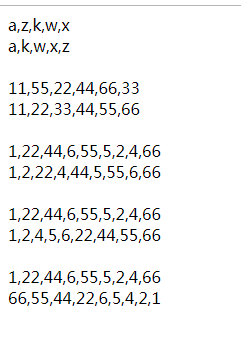
document.write(arr4 + "<br />");

document.write(arr4.sort(sortNum2) + "<br />" + "<br />");

</script>



运行结果为：



* toString() 方法，把数组转换为字符串，并返回结果。

#### String 对象

1. String 对象的常用属性：length，获取字符串的长度。
2. String 对象的常用方法：

* charAt() 方法，获取指定位置处字符。语法为：

stringObject.charAt(index);

// 字符串中第一个字符的下标是 0。如果参数 index 不在 0 与 string.length 之间，该方法将返回一个空字符串



例子：

var str = 'Hello world!';

document.write(str.charAt(2));

// 以上代码输出为 l



* charCodeAt() 方法，获取指定位置处字符的 Unicode 编码。语法为：

stringObject.charCodeAt(index);

// 字符串中第一个字符的下标是 0。如果 index 是负数，或大于等于字符串的长度，则 charCodeAt() 返回 NaN



例子：

var str = 'Hello world!';

document.write(str.charCodeAt(2));

// 以上代码输出为 l08



* concat() 方法，连接字符串，等效于 “+”，“+” 更常用。与数组中的 concat() 方法相似。
* slice() 方法，提取字符串的片断，并在新的字符串中返回被提取的部分（字符串章节有详细介绍，这里不过多的赘述，下面的类似情况同样处理）。
* indexOf() 方法，检索字符串。
* toString() 方法，返回字符串。
* toLowerCase() 方法，把字符串转换为小写。
* toUpperCase() 方法，把字符串转换为大写。
* replace() 方法，替换字符串中的某部分。
* split() 方法，把字符串分割为字符串数组。

#### Date 对象

Date 对象方法：

* Date()：返回当日的日期和时间（输出的是中国标准时间）。
* getDate()：从 Date 对象返回一个月中的某一天 (1 ~ 31)。
* getDay()：从 Date 对象返回一周中的某一天 (0 ~ 6)。
* getMonth()：从 Date 对象返回月份 (0 ~ 11)。
* getFullYear()：从 Date 对象以四位数字返回年份。
* getHours()：返回 Date 对象的小时 (0 ~ 23)。
* getMinutes()：返回 Date 对象的分钟 (0 ~ 59)。
* getSeconds()：返回 Date 对象的秒数 (0 ~ 59)。
* getMilliseconds()：返回 Date 对象的毫秒(0 ~ 999)。

#### Math 对象

1. Math 对象的常用属性：

* E ：返回常数 e (2.718281828...)。
* LN2 ：返回 2 的自然对数 (ln 2)。
* LN10 ：返回 10 的自然对数 (ln 10)。
* LOG2E ：返回以 2 为底的 e 的对数 (log2e)。
* LOG10E ：返回以 10 为底的 e 的对数 (log10e)。
* PI ：返回 π（3.1415926535...)。
* SQRT1\_2 ：返回 1/2 的平方根。
* SQRT2 ：返回 2 的平方根。

1. Math 对象的常用方法：

* abs(x) ：返回 x 的绝对值。
* round(x) ：返回 x 四舍五入后的值。
* sqrt(x) ：返回 x 的平方根。
* ceil(x) ：返回大于等于 x 的最小整数。
* floor(x) ：返回小于等于 x 的最大整数。
* sin(x) ：返回 x 的正弦。
* cos(x) ：返回 x 的余弦。
* tan(x) ：返回 x 的正切。
* acos(x) ：返回 x 的反余弦值（余弦值等于 x 的角度），用弧度表示。
* asin(x) ：返回 x 的反正弦值。
* atan(x) ：返回 x 的反正切值。
* exp(x) ：返回 e 的 x 次幂 (e^x)。
* pow(n, m) ：返回 n 的 m 次幂 (nm)。
* log(x) ：返回 x 的自然对数 (ln x)。
* max(a, b) ：返回 a, b 中较大的数。
* min(a, b) ：返回 a, b 中较小的数。
* random() ：返回大于 0 小于 1 的一个随机数。

### 创建对象和访问对象

创建对象的方式有很多，下面为大家讲解几种常用的方式。

#### 通过对象字面量来创建

var student = {

name: 'zhangsan',

age: 18,

gender: 'male',

sayHi: function () {

console.log('hi,my name is ' + this.name);

},

};

把上面的代码复制到控制台中，然后尝试依次输入下面的代码看看效果：

student.name;

student.age;

student.gender; // 调用对象的属性

student.sayHi(); // 调用对象的方法

通过上面的例子你会发现对象的属性和方法通过 "." 来访问。

#### 通过 new Object() 创建对象

var student = new Object();

(student.name = 'zhangsan'),

(student.age = 18),

(student.gender = 'male'),

(student.sayHi = function () {

console.log('hi,my name is ' + this.name);

});

#### 通过工厂函数创建对象

function createStudent(name, age, gender) {

var student = new Object();

student.name = name;

student.age = age;

student.gender = gender;

student.sayHi = function () {

console.log('hi,my name is ' + this.name);

};

return student;

}

var s1 = createStudent('zhangsan', 18, 'male');

#### 自定义构造函数

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

this.sayHi = function () {

console.log('hi,my name is ' + this.name);

};

}

var s1 = new Student('zhangsan', 18, 'male');

#### new 关键字

构造函数，是一种特殊的函数。主要用来在创建对象时初始化对象，即为对象成员变量赋初始值，总与 new 运算符一起使用在创建对象的语句中。这里有需要特别注意的几点：

* 构造函数用于创建一类对象，首字母要大写。
* 内部使用 this 关键字给对象添加成员。
* 使用 new 关键字调用对象构造函数。

#### this 详解

在 JavaScript 中，我们经常会使用到 this 关键字，那么 this 到底指向什么呢？这里有一个口诀：谁调用 this，它就是谁。大家也可以从前面的例子中细细体会一下。

* 函数在定义的时候 this 是不确定的，只有在调用的时候才可以确定。
* 一般函数直接执行，内部 this 指向全局 window。比如：

function test() {

console.log(this);

}

test(); // window.test();

* 函数作为一个对象的方法，被该对象所调用，那么 this 指向的是该对象。
* 构造函数中的 this，始终是 new 的当前对象。

#### 遍历对象的属性

通过 for...in 语句用于遍历数组或者对象的属性，对数组或者对象的属性进行循环操作。比如：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

var student = {

name: 'zhangsan',

age: 18,

gender: 'male',

};

for (var key in student) {

console.log(key);

console.log(student[key]);

}

</script>

</body>

</html>

注：key 是一个变量，这个变量中存储的是该对象的所有的属性的名字。

#### 删除对象的属性

使用 delete 删除对象的属性。比如在控制台中输入以下代码：

var student = {

name: 'zhangsan',

age: 18,

gender: 'male',

};

student.name; // zhangsan

delete student.name;

student.name; // undefined

### 挑战：内置对象使用案例

1. 写一个函数，格式化日期对象，最终输入形式为：yyyy/MM/dd HH:mm:ss。比如：

1

参考源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

function formatDate(d) {

// 如果 d 不是日期对象的实例，返回空

if (!d instanceof Date) {

return;

}

var year = d.getFullYear(),

month = d.getMonth() + 1,

date = d.getDate(),

hour = d.getHours(),

minute = d.getMinutes(),

second = d.getSeconds();

month = month < 10 ? '0' + month : month;

date = date < 10 ? '0' + date : date;

hour = hour < 10 ? '0' + hour : hour;

minute = minute < 10 ? '0' + minute : minute;

second = second < 10 ? '0' + second : second;

return (

year +

'/' +

month +

'/' +

date +

' ' +

hour +

':' +

minute +

':' +

second

);

}

console.log(formatDate(new Date()));

</script>

</body>

</html>



1. 写一个函数去掉数组中的重复元素。比如有一个数组 array = ["x", "c", "a", "b", "c", "b", "c"]。

参考源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script type="text/javascript">

var array = ['x', 'c', 'a', 'b', 'c', 'b', 'c'];

function clearArray() {

var o = {};

for (var i = 0; i < array.length; i++) {

var item = array[i];

if (o[item]) {

o[item]++;

} else {

o[item] = 1;

}

}

var tmpArray = [];

for (var key in o) {

if (o[key] == 1) {

tmpArray.push(key);

} else {

if (tmpArray.indexOf(key) == -1) {

tmpArray.push(key);

}

}

}

return tmpArray;

}

console.log(clearArray(array));

</script>

</body>

</html>



1. 写一个函数判断一个字符串中出现次数最多的字符，并统计这个字符出现的次数。比如有这样一个字符串："daskljahiagnaknscsjdwfanalvnahdpwod"。

参考源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script type="text/javascript">

var str = 'daskljahiagnaknscsjdwfanalvnahdpwod';

function max() {

var o = {};

for (var i = 0; i < str.length; i++) {

var item = str.charAt(i);

if (o[item]) {

o[item]++;

} else {

o[item] = 1;

}

}

var max = 0;

for (var key in o) {

if (max < o[key]) {

max = o[key];

var tmp;

tmp = key;

}

}

console.log('出现次数最多的字符是:' + tmp + '出现次数为：' + max);

}

max(str);

</script>

</body>

</html>

## Web Api

浏览器是一个封装的较为完善的软件，它给我们提供了操作浏览器功能和页面元素的接口。在本节中，我们将对这些接口进行学习。

#### 知识点

* API 概念
* BOM 简介
* BOM 的顶级对象 window
* DOM 简介
* DOM HTML
* DOM CSS
* DOM 节点
* DOM 事件

#### API

API（Application Programming Interface，应用程序编程接口）："计算机操作系统"（Operating system）或"程序库"提供给应用程序调用使用的代码。其主要目的是让应用程序开发人员得以调用一组例程功能，而无须考虑其底层的源代码为何、或理解其内部工作机制的细节。API 本身是抽象的，它仅定义了一个接口，而不涉及应用程序在实际实现过程中的具体操作。

注：定义来自维基百科。

#### Web API

Web API 是浏览器提供的一套操作浏览器功能和页面元素的 API（BOM 和 DOM）。

### BOM 简介

浏览器对象模型（Browser Object Model (BOM)）指的是由 Web 浏览器暴露的所有对象组成的表示模型。BOM 与 DOM（Document Object Model，文档对象模型）不同，其既没有标准的实现，也没有严格的定义，所以浏览器厂商可以自由地实现 BOM。

作为显示文档的窗口，浏览器程序将其视为对象的分层集合。当浏览器分析文档时，它将创建一个对象的集合，以定义文档，并详细说明它应如何显示。浏览器创建的对象称为文档对象，它是浏览器使用的更大的对象集合的一部分。此浏览器对象集合统称为浏览器对象模型或 BOM。

BOM 层次结构的顶层是窗口对象，它包含有关显示文档的窗口的信息。某些窗口对象本身就是描述文档和相关信息的对象。

注：定义来自维基百科。

### BOM 的顶级对象 window 以及常用操作方法

window 是浏览器的顶级对象，当调用 window 下的属性和方法时，可以省略 window。

#### 对话框

* alert()：显示带有一段消息和一个确认按钮的警告框。
* prompt()：显示可提示用户输入的对话框。
* confirm()：显示带有一段消息以及确认按钮和取消按钮的对话框。

#### 页面加载事件

* onload

window.onload = function () {

// 当页面加载完成执行

// 当页面完全加载所有内容（包括图像、脚本文件、CSS 文件等）执行

};

* onunload

window.onunload = function () {

// 当用户退出页面时执行

};

#### 浏览器尺寸

var width = window.innerWidth;

document.documentElement.clientWidth;

document.body.clientWidth;

var height = window.innerHeight;

document.documentElement.clientHeight;

document.body.clientHeight;

上述代码可以获取所有浏览器的宽高（不包括工具栏/滚动条）。

#### 定时器

* setTimeout() 方法在指定的毫秒数到达之后执行指定的函数，只执行一次。clearTimeout() 方法取消由 setTimeout() 方法设置的 timeout。

// 创建一个定时器，2000毫秒后执行，返回定时器的标示

var timerId = setTimeout(function () {

console.log('Hello shiyanlou');

}, 2000);

// 取消定时器的执行

clearTimeout(timerId);

* setInterval() 方法设置定时调用的函数也就是可以按照给定的时间（单位毫秒）周期调用函数，clearInterval() 方法取消由 setInterval() 方法设置的 timeout。

// 创建一个定时器，每隔 2 秒调用一次

var timerId = setInterval(function () {

var date = new Date();

console.log(date.toLocaleTimeString());

}, 2000);

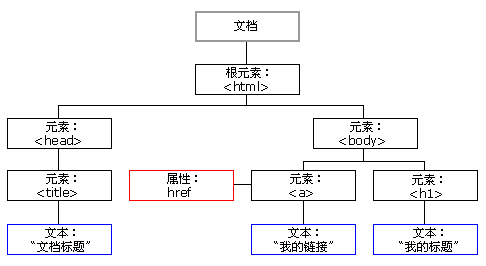
// 取消定时器的执行

clearInterval(timerId);

注：BOM 的操作方法还有很多，但是一般情况下我们常用的就是上面所介绍的。有兴趣的可以自行百度了解 BOM 的更多操作方法和介绍。

### DOM 简介

文档对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展标志语言的标准编程接口。DOM 定义了访问 HTML 和 XML 文档的标准。我们这里主要学习 HTML DOM。DOM 可以把 HTML 看做是文档树，通过 DOM 提供的 API 可以对树上的节点进行操作。下面我们来看一下 W3C 上的 DOM 树：



### DOM HTML

DOM 能够操作 HTML 的内容。

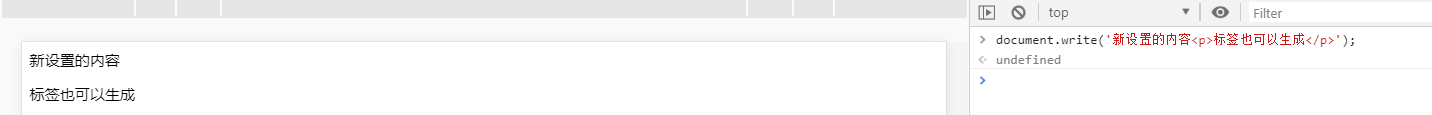
#### 改变 HTML 输出流

在 JavaScript 中，使用 document.write() 可用于直接向 HTML 输出流写内容。比如：

document.write('新设置的内容<p>标签也可以生成</p>');



在控制台中复制上述代码运行后：



#### 改变 HTML 内容

使用 innerHTML 属性改变 HTML 内容。比如修改 p 标签中的内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<p id="p1">Hello World!</p>

<script>

document.getElementById('p1').innerHTML = 'Hello 实验楼';

</script>

</body>

</html>

#### 改变 HTML 属性

语法：

document.getElementById(id).attribute = new value();

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<img id="image" src="https://www.baidu.com/img/bd\_logo1.png" />

<script>

document.getElementById('image').src =

'https://static.shiyanlou.com/img/shiyanlou\_logo.svg';

</script>

</body>

</html>

注：上述例子将显示实验楼的 logo 图片。

### DOM CSS

DOM 能够改变 HTML 元素的样式。语法为：

document.getElementById(id).style.property = new style();

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<p id="syl" style="color: red;">实验楼</p>

<script>

document.getElementById('syl').style.color = 'green';

</script>

</body>

</html>

注：在上述例子中，p 标签中实验楼的颜色本来为红色，但是通过 DOM 方法，最后将其改变成了绿色。运行上述代码，最终的效果是显示一个颜色为绿色的实验楼文本。

### DOM 节点

根据 W3C 的 HTML DOM 标准，HTML 文档中的所有内容都是节点：整个文档就是一个文档节点，而每一个 HTML 标签都是一个元素节点。HTML 标签中的文本则是文本节点，HTML 标签的属性是属性节点，一切都是节点。



### DOM 节点的操作

#### 获取节点

要操作节点，首先我们要找到节点。主要有以下三种办法：

1. 通过 ID 找到 HTML 元素：使用方法 getElementById() 通过元素的 ID 而选取元素，比如：

document.getElementById('demo'); // 假定已经有一个 ID 名为 demo 的标签，可以这样来获取它

1. 通过标签名找到 HTML 元素：使用方法 getElementsByTagName() 来选取元素，如果有多个同类型标签，那么我们可以通过下标来确认，比如：

<html>

<body>

<input type="text" />

<input type="text" />

<script>

document.getElementsByTagName('input')[0].value = 'hello'; // 下标为 [0] 表示选取第 1 个 input 标签

document.getElementsByTagName('input')[1].value = 'shiyanlou'; // 下标为 [1] 表示选取第 2 个 input 标签

</script>

</body>

</html>

1. 通过类名来找到 HTML 元素：使用方法 getElementsByClassName() 通过元素的类名来选取元素。比如：

document.getElementsByClassName('name'); // 返回包含 class = "name" 的所有元素的一个列表。

#### DOM 节点之间的关系

DOM 的节点并不是孤立的，我们从 DOM 树中也可以看出，节点与节点之间存在着相对的关系，就如同一个家族一样，有父辈，有兄弟，有儿子等等。下面我们来看一下都有哪些节点：

| **父节点** | **兄弟节点** | **子节点** | **所有子节点** |
| --- | --- | --- | --- |
| parentNode | nextSibling | firstChild | childNodes |
|  | nextElementSibling | firstElementChild | children |
|  | previousSibling | lastChild |  |
|  | previousElementSibling | lastElementChild |  |

例子：

<html>

<head>

<title>DOM 节点演示</title>

</head>

<body>

<h1>我是h1标签</h1>

<p>我是p标签</p>

</body>

</html>

上面的例子中：

* <html>节点没有父节点，它是根节点。
* <head> 和 <body> 的父节点是 <html> 节点。
* 文本节点 我是 p 标签 的父节点是 <p> 节点。
* <html> 节点有两个子节点：<head> 和 <body>。
* <h1> 节点和 <p> 节点是兄弟节点，同时也是 <body> 的子节点。

需要注意以下几点：

* childNodes：它是标准属性，它返回指定元素的子元素集合，包括 HTML 节点，所有属性，文本节点。
* children：非标准属性，它返回指定元素的子元素集合。但它只返回 HTML 节点，甚至不返回文本节点。
* nextSibling 和 previousSibling 获取的是节点，获取元素对应的属性是 nextElementSibling 和 previousElementSibling。
* nextElementSibling 和 previousElementSibling 有兼容性问题，IE9 以后才支持。

#### DOM 节点的操作

1. 创建节点
   * 创建元素节点：使用 createElement() 方法。比如：
2. var par = document.createElement('p');
   * 创建属性节点：使用 createAttribute() 方法。
   * 创建文本节点：使用 createTextNode() 方法。
3. 插入子节点
   * appendChild () 方法向节点添加最后一个子节点。
   * insertBefore (插入的新的子节点，指定的子节点) 方法在指定的子节点前面插入新的子节点。如果第二个参数没写或者为 null，则默认插入到后面。
4. 删除节点：使用 removeChild() 方法。写法为：

父节点.removeChild(子节点);

node.parentNode.removeChild(node); // 如果不知道父节点是什么，可以这样写

1. 替换子节点：使用 replaceChild() 方法。语法为：

node.replaceChild(newnode, oldnode);

1. 设置节点的属性：
   * 获取：getAttribute(name)
   * 设置：setAttribute(name, value)
   * 删除：removeAttribute(name)

### DOM 事件

#### 事件的定义

在什么时候执行什么事。

#### 事件三要素

事件由：事件源 + 事件类型 + 事件处理程序组成。

* 事件源：触发事件的元素。
* 事件类型：事件的触发方式（比如鼠标点击或键盘点击）。
* 事件处理程序：事件触发后要执行的代码（函数形式，匿名函数）。

#### 常用的事件

| **事件名** | **说明** |
| --- | --- |
| onclick | 鼠标单击 |
| ondblclick | 鼠标双击 |
| onkeyup | 按下并释放键盘上的一个键时触发 |
| onchange | 文本内容或下拉菜单中的选项发生改变 |
| onfocus | 获得焦点，表示文本框等获得鼠标光标。 |
| onblur | 失去焦点，表示文本框等失去鼠标光标。 |
| onmouseover | 鼠标悬停，即鼠标停留在图片等的上方 |
| onmouseout | 鼠标移出，即离开图片等所在的区域 |
| onload | 网页文档加载事件 |
| onunload | 关闭网页时 |
| onsubmit | 表单提交事件 |
| onreset | 重置表单时 |

例子 1 ：鼠标单击事件：

<p onclick="this.innerHTML = '我爱学习，身体好好!'">请点击该文本</p>

例子 2 ：鼠标双击事件：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<h1 ondblclick="changetext(this)">请点击该文本</h1>

<script>

function changetext(id) {

id.innerHTML = '我爱学习，身体棒棒!';

}

</script>

</body>

</html>

例子 3 ：鼠标移除悬停：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<div

onmouseover="mOver(this)"

onmouseout="mOut(this)"

style="background-color:deepskyblue;width:200px;height:100px;"

>

把鼠标移到上面

</div>

<script>

function mOver(obj) {

obj.innerHTML = '你把鼠标移到了上面 ';

}

function mOut(obj) {

obj.innerHTML = '你把鼠标移开了';

}

</script>

</body>

</html>

注：大家可以把上述例子运行一下感受一下事件的魅力，并且可以自己尝试着写一些其他的简单事件。

### 挑战：导航栏样式切换

通过鼠标点击更改导航栏的样式，来看看最终的效果：



参考源码：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Title</title>

<style>

#list li {

list-style-type: none;

width: 100px;

height: 50px;

line-height: 50px;

background-color: beige;

text-align: center;

float: left;

}

#list li.current {

background-color: red;

}

#list li a {

text-decoration: none;

}

</style>

</head>

<body>

<div id="menu">

<ul id="list">

<li class="current">

<a href="javascript:void(0)">首页</a>

</li>

<li>

<a href="javascript:void(0)">HTML</a>

</li>

<li>

<a href="javascript:void(0)">CSS</a>

</li>

<li>

<a href="javascript:void(0)">JavaScript</a>

</li>

<li>

<a href="javascript:void(0)">关于</a>

</li>

<li>

<a href="javascript:void(0)">帮助</a>

</li>

</ul>

</div>

<script>

// 获取所有的 li 标签

var liObjs = document.getElementById('list').getElementsByTagName('li');

// 循环遍历，找到每个 li 中的 a，注册点击事件

for (var i = 0; i < liObjs.length; i++) {

// 每个 li 中的 a

var aObj = liObjs[i].firstElementChild;

aObj.onclick = function () {

// 把这个 a 所在的 li 的所有的兄弟元素的类样式全部移除

for (var j = 0; j < liObjs.length; j++) {

liObjs[j].removeAttribute('class');

}

//当前点击的 a 的父级元素 li(点击的这个 a 所在的父级元素 li)，设置背景颜色

this.parentNode.className = 'current';

};

}

</script>

</body>

</html>

## 值类型和引用类型

#### 值类型

值类型又叫基本数据类型，在 JavaScript 中值类型有以下五种：

* 数值类型
* 布尔类型
* undefined
* null
* 字符串

值类型存储在栈（stack）中，它们的值直接存储在变量访问的位置。比如：

var num = 18;

var flag = true;

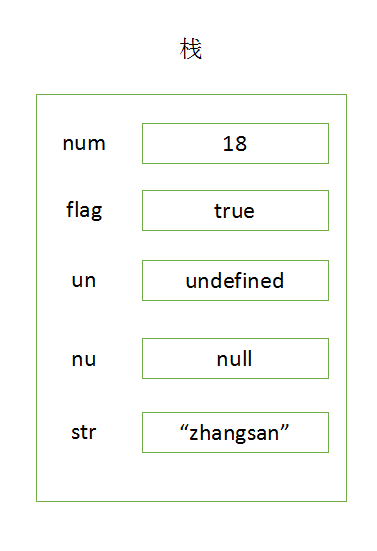
var un = undefined;

var nu = null;

var str = 'zhangsan';



上面定义的这些值类型的数据在内存中的存储如下图所示：



#### 引用类型

引用类型又叫复合数据类型，在 JavaScript 中引用类型有以下三种：

* 对象
* 数组
* 函数

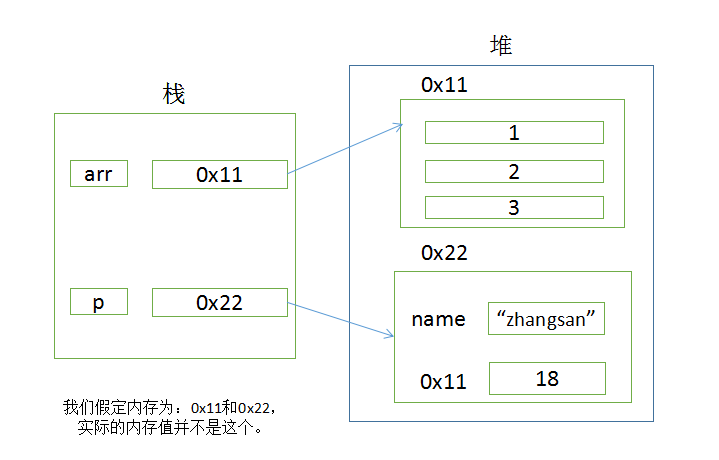
引用类型存储在堆中，也就是说存储在变量处的值是一个指针，指向存储对象的内存处。比如：

var arr = [1, 2, 3];

var p = { name: '张三', age: 18 };



上面定义的这些引用类型的数据在内存中的存储如下图所示：



### 值类型和引用类型的特征

#### 值类型的特征

* 值类型的值是不可变的，不可变是指值类型指向的空间不可变。比如：

var a = 2;

a = a + 2;

console.log(a); // 打印结果为 4。



在上述例子中，a 变量指向的值变了，但是 2 的内存没有变。

* 按值传递的变量之间互不影响。比如：

var a = 1;

var b = a;

a = a + 2;

console.log(a, b); // 打印结果为 3, 1



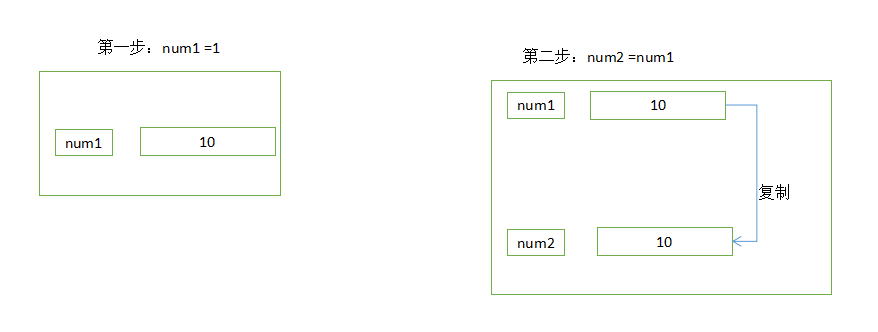
* 值类型赋值，直接将值赋值一份。比如：

var num1 = 10;

var num2 = num1;



上述代码在内存中的体现为：



* 当参数为值类型的时候，函数内和函数外的两个变量完全不同，仅仅只是存的值一样而已，修改时互不影响。比如：

function foo(num) {

num = num + 1;

}

var a = 1;

foo(a);

console.log(a); // 打印结果为 1



#### 引用类型的特征

* 引用类型赋值，是将地址复制一份。比如：

var p = { name: 'zhangsan', age: 18 };

var p1 = p;



上述代码在内存中的体现为：



再来看一段代码：

var p = { name: '张三', age: 18 };

var p1 = p;

console.log(p.name); // 打印结果为张三

console.log(p1.name); // 打印结果为张三

p.name = '李四';

console.log(p.name); // 打印结果为李四

console.log(p1.name); // 打印结果为李四



* 当参数为引用类型的时候，函数内和函数外的两个变量不同，但是共同指向同一个对象，在函数内修改对象数据时会影响外部。比如：

function foo(o) {

o.age = o.age + 1;

}

var p = { name: 'zhangsan', age: 18 };

foo(p);

console.log(p.age); // 打印结果为 19。

注：其实我们可以这样理解。引用类型中的地址是一把钥匙，钥匙指向的是宝藏，复制一把钥匙后，两把钥匙能打开的是同一个宝藏。

## 调试工具的使用

在编写 JavaScript 代码时，如果遇见错误，首先在浏览器中运行我们的代码，然后 F12，查看错误信息。比如运行以下代码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

alerttt('hello');

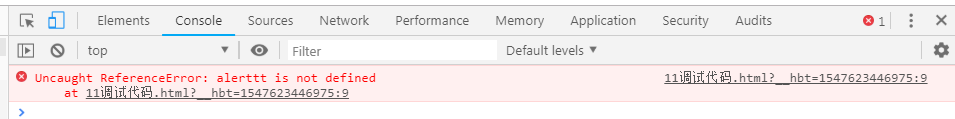
</script>

</body>

</html>



首先查看控制台信息，在控制台中会有报错提示，一般看这个就能知道问题是什么了。



在 Sources 中能够清楚的看到哪一行的代码出问题了，会有很明显地提醒。



如果我们想知道变量的值，在调试的时候，可以加一句 console.log(变量) 语句来打印出来，然后在控制台中看。console.log() 语句也是我们编程中经常需要使用的，因为有时候，我们也不能直观的一下就知道传递进来的值到底是什么，可能需要看半天的逻辑然后计算半天。直接打印出来的方式也有利于帮助我们判断，不过一般情况下大家记得调试完，要把这行语句注释掉或者删掉。

#### 设置断点，逐步执行

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<script>

for (var i = 1; i <= 100; i++) {

for (var j = 1; j <= 10; i++) {

j = i \* j;

console.log(j);

}

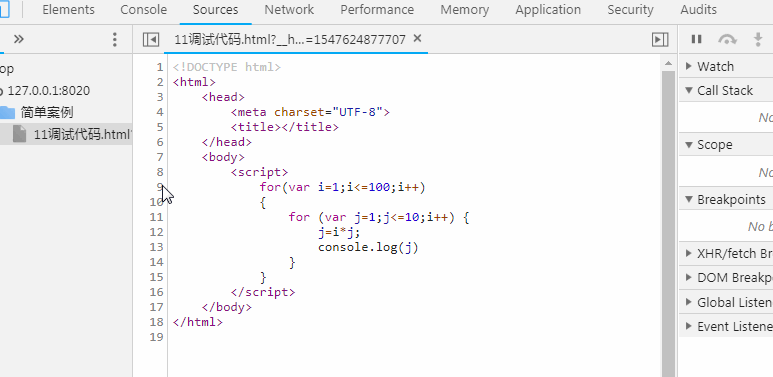
}

</script>

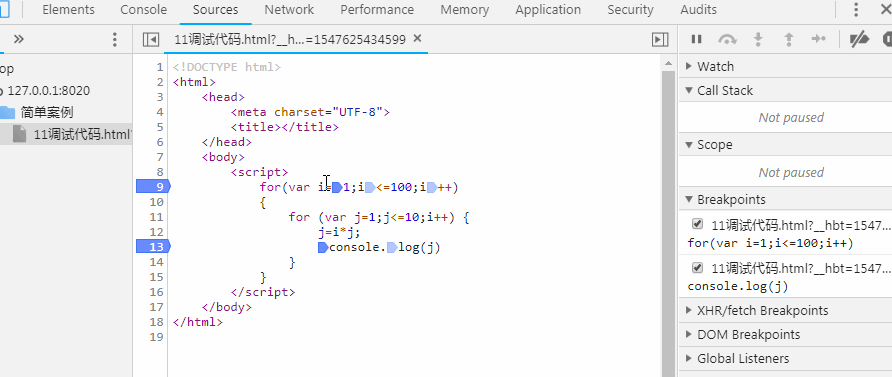
</body>

</html>

首先运行上述代码，点击 F12 进行调试界面，点击 Sources，设置断点的方法很简单，直接左键点击代码旁边的数字行数。如下所示：



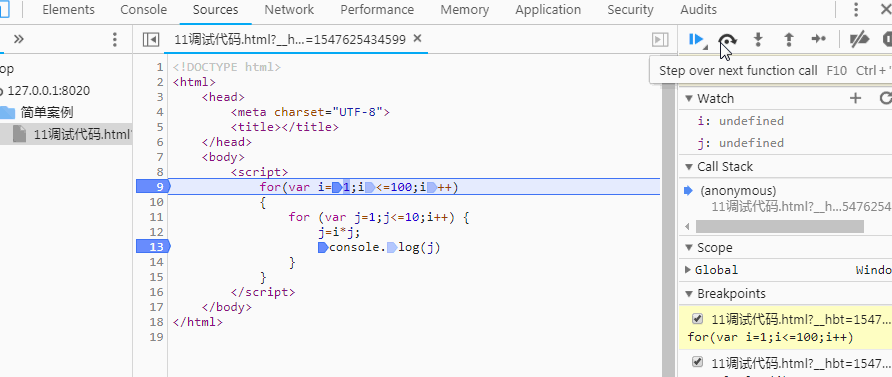
将变量添加到 Watch 窗口，实时查看它的值的变化，如下所示：



准备工作做好之后我们还需要点击一下刷新，如下所示，红色箭头所指的地方点击一下：

1

然后就可以通过点击运行按钮，逐行运行我们设置断点时的代码，并在 Watch 中查看变量值的变化，如下所示：



## 异常处理

在我们实际编程过程中，经常会遇到各种各样的错误，有可能是语法错误，也可能是拼写错误，也可能是浏览器的兼容性问题或者其它莫名其妙的问题。当出现错误时，JavaScript 引擎通常会停止，并生成一个错误信息。那我们应该怎么来调试我们的代码呢？

#### 异常捕获

我们使用 try-catch 语句开捕获异常，语法为：

try {

// 这里写可能出现异常的代码

} catch (err) {

// 在这里写，出现异常后的处理代码

}



例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<input type="button" value="点击一下" onclick="message()" />

<script>

var txt = '';

function message() {

try {

alertt('我爱学习，身体好好'); // 故意写错 alert

} catch (err) {

txt = 'There was an error on this page.\n\n';

txt += 'Error description: ' + err.message + '\n\n';

txt += 'Click OK to continue.\n\n';

alert(txt);

}

}

</script>

</body>

</html>



需要注意以下几点：

* 语句 try 和 catch 是成对出现的。
* 如果在 try 中出现了错误，try 里面出现错误的语句后面的代码都不再执行，直接跳转到 catch 中，catch 处理错误信息，然后再执行后面的代码。
* 如果 try 中没有出现错误，则不会执行 catch 中的代码，执行完 try 中的代码后直接执行后面的代码。
* 通过 try-catch 语句进行异常捕获之后，代码将会继续执行，而不会中断。

#### throw 语句

通过 throw 语句，我们可以创建自定义错误。throw 语句常常和 try catch 语句一起使用。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

</head>

<body>

<p>请输入 0 到 100 之间的数字：</p>

<input id="demo" type="text" />

<button type="button" onclick="myFunction()">测试输入值</button>

<p id="throwText"></p>

<script>

function myFunction() {

try {

var x = document.getElementById('demo').value;

if (x == '') throw '您输入的值为空';

if (isNaN(x)) throw '您输入的不是数字';

if (x > 100) throw '您输入的值太大';

if (x < 0) throw '您输入的值太小';

} catch (err) {

var y = document.getElementById('throwText');

y.innerHTML = '错误提示：' + err + '。';

}

}

</script>

</body>

</html>

## 面向对象

### 面向对象编程举例

下面我们通过一个例子，来感受一下什么叫面向对象编程。比如我们设置页面中的 div 标签 和 p 标签的背景色为 color。如果按照我们前面所需我们可能会这样写：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div,

p {

width: 200px;

height: 100px;

}

</style>

</head>

<body>

<div>你好吗？</div>

<p>我很好</p>

<div>测试一下嘛</div>

<p>好的啊</p>

<script>

var divs = document.getElementsByTagName('div');

for (var i = 0; i < divs.length; i++) {

divs[i].style.backgroundColor = 'red';

}

var ps = document.getElementsByTagName('p');

for (var j = 0; j < ps.length; j++) {

ps[j].style.backgroundColor = 'red';

}

</script>

</body>

</html>



是不是觉得有点麻烦？好像有重复的？有的人可能会想到用函数来封装一下相同的代码:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div,

p {

width: 200px;

height: 100px;

}

</style>

</head>

<body>

<div>你好吗？</div>

<p>我很好</p>

<div>测试一下嘛</div>

<p>好的啊</p>

<script>

function getTagname(tagName) {

return document.getElementsByTagName(tagName);

}

function setStyle(arr) {

for (var i = 0; i < arr.length; i++) {

arr[i].style.backgroundColor = 'red';

}

}

var divs = getTagname('div');

setStyle(divs);

var ps = getTagname('p');

setStyle(ps);

</script>

</body>

</html>



我们再来看看使用面向对象的方式：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div,

p {

width: 200px;

height: 100px;

}

</style>

</head>

<body>

<div>你好吗？</div>

<p>我很好</p>

<div>测试一下嘛</div>

<p>好的啊</p>

<script>

var test = {

getEle: {

// 实际上本案例只需要写 tag，但是为了体现面向对象的思想，我们把获取节点的三种方式都写出来

tag: function (tagName) {

return document.getElementsByTagName(tagName);

},

id: function (idName) {

return document.getElementById(idName);

},

class: function (className) {

return document.getElementsByClassName(className);

},

},

// 实际上本案例只需要写 setStyle，同样的为了体现面向对象编程的思想，我们可以设置添加移除修改样式的函数。

setCss: {

setStyle: function (arr) {

for (var i = 0; i < arr.length; i++) {

arr[i].style.backgroundColor = 'red';

}

},

updateCss: function () {},

deleteCss: function () {},

// ...

},

};

var divs = test.getEle.tag('div');

test.setCss.setStyle(divs);

var ps = test.getEle.tag('p');

test.setCss.setStyle(ps);

</script>

</body>

</html>

### 构造函数

首先，我们来复习一下创建对象的方式。

1. 通过对象字面量来创建。

var student = {

name: 'zhangsan',

age: 18,

gender: 'male',

sayHi: function () {

console.log('hi,my name is ' + this.name);

},

};



1. 通过 new Object() 创建对象。

var student = new Object();

(student.name = 'zhangsan'),

(student.age = 18),

(student.gender = 'male'),

(student.sayHi = function () {

console.log('hi,my name is ' + this.name);

});



上面两种都是简单的创建对象的方式，但是如果有两个 student 实例对象呢？cv（Ctrl C + Ctrl V） 大法？分别命名一下 student1 和 student2？那如果是一个班的学生，n 个学生呢？显然如果这样做的话代码冗余率太高，是不可取的。我们也学过函数，所以简单方式的改进是：工厂函数。

1. 通过工厂函数来创建对象。

function createStudent(name, age, gender) {

var student = new Object();

student.name = name;

student.age = age;

student.gender = gender;

student.sayHi = function () {

console.log('hi,my name is ' + this.name);

};

return student;

}

var s1 = createStudent('zhangsan', 18, 'male');

var s2 = createStudent('lisi', 19, 'male');



这样封装代码确实解决了代码冗余的问题，但是每次调用函数 createStudent() 都会创建新函数 sayHi()，也就是说每个对象都有自己的 sayHi() 版本，而事实上，每个对象都共享一个函数。为了解决这个问题，我们引入面向对象编程里的一个重要概念：构造函数。

1. 通过构造函数来创建对象。

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

this.sayHi = function () {

console.log('hi,my name is ' + this.name);

};

}

var s1 = new Student('zhangsan', 18, 'male');

来看看构造函数与工厂函数的区别：

* 首先在构造函数内没有创建对象，而是使用 this 关键字，将属性和方法赋给了 this 对象。
* 构造函数内没有 return 语句，this 属性默认下是构造函数的返回值。
* 函数名使用的是大写的 Student。
* 用 new 运算符和类名 Student 创建对象。

### 构造函数存在的问题

构造函数虽然科学，但仍然存在一些问题。

我们使用前面的构造函数例子来讲解（修改了 sayHi() 方法）：

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

this.sayHi = function () {

console.log('hi');

};

}



首先我们创建两个实例化对象：

var s1 = new Student('zhangsan', 18, 'male');

s1.sayHi(); // 打印 hi

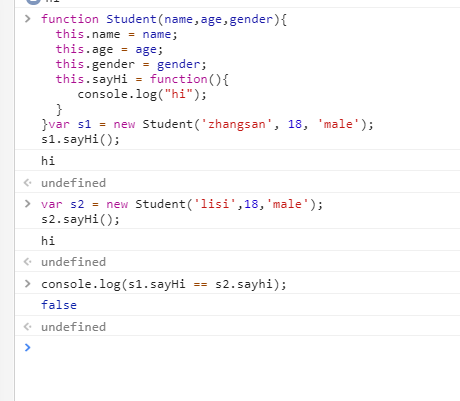
var s2 = new Student('lisi', 18, 'male');

s2.sayHi(); // 打印 hi

console.log(s1.sayHi == s2.sayhi); // 结果为 false



眼见为实，来看看效果：



由于每个对象都是由 new Student 创建出来的，因此每创建一个对象，函数 sayHi() 都会被重新创建一次，这个时候，每个对象都拥有一个独立的，但是功能完全相同的方法，这样势必会造成内存浪费。有的人可能会想，既然是一样的那我们就单独把它提出来，写一个函数，每次调用不就可以了吗？比如：

function sayHi() {

console.log('hi');

}

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

this.sayHi = sayHi;

}

var s1 = new Student('zhangsan', 18, 'male');

s1.sayHi(); // 打印 hi

var s2 = new Student('lisi', 18, 'male');

s2.sayHi(); // 打印 hi

console.log(s1.sayHi == s2.sayHi); // 结果为 true



但是这样做会导致全局变量增多，可能会引起命名冲突，代码结果混乱，维护困难。通过使用原型可以很好的解决这个问题。

### 原型

#### 原型：prototype

在 JavaScript 中，每一个函数都有一个 prototype 属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。我们来看看前面例子原型的写法：

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

}

Student.prototype.sayHi = function () {

console.log('hi');

};

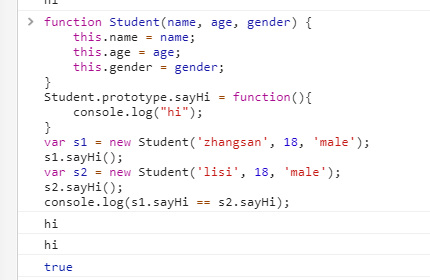
var s1 = new Student('zhangsan', 18, 'male');

s1.sayHi(); // 打印 hi

var s2 = new Student('lisi', 18, 'male');

s2.sayHi(); // 打印 hi

console.log(s1.sayHi == s2.sayHi); // 结果为 true



#### 构造函数、实例、原型三者之间的关系

我们之前提到过：每一个函数都有一个 prototype 属性，指向另一个对象。让我们用代码验证一下，在编辑器中输入以下代码：

<script type="text/javascript">

function F() {}

console.log(F.prototype);

</script>

上述代码在浏览器中打印结果为 Object，验证了我们所说的 prototype 属性，指向另一个对象。

构造函数的 prototype 对象默认都有一个 constructor 属性，指向 prototype 对象所在函数。在控制台中运行下面的代码：

function F() {}

console.log(F.prototype.constructor === F); // 结果为 ture

通过构造函数得到的实例对象内部会包含一个指向构造函数的 prototype 对象的指针 \_\_proto\_\_。\_\_proto\_\_ 属性最早是火狐浏览器引入的，用以通过实例对象来访问原型，这个属性在早期是非标准的属性。在控制台中运行下面的代码：

function F() {}

var a = new F();

console.log(a.\_\_proto\_\_ === F.prototype); // 结果为 true

实例对象可以直接访问原型对象成员，所有实例都直接或间接继承了原型对象的成员。

总结：每个构造函数都有一个原型对象，原型对象包含一个指向构造函数的指针 constructor，而实例都包含一个指向原型对象的内部指针\_\_proto\_\_。

#### 原型链

我们说过所有的对象都有原型，而原型也是对象，也就是说原型也有原型，那么如此下去，也就组成了我们的原型链。

#### 属性搜索原则

属性搜索原则，也就是属性的查找顺序，在访问对象的成员的时候，会遵循以下原则：

* 首先从对象实例本身开始找，如果找到了这个属性或者方法，则返回。
* 如果对象实例本身没有找到，就从它的原型中去找，如果找到了，则返回。
* 如果对象实例的原型中也没找到，则从它的原型的原型中去找，如果找到了，则返回。
* 一直按着原型链查找下去，找到就返回，如果在原型链的末端还没有找到的话，那么如果查找的是属性则返回 undefined，如果查找的是方法则返回 xxx is not a function。

#### 更简单的原型语法

在前面的例子中，我们是使用 xxx.prototype. 然后加上属性名或者方法名来写原型，但是每添加一个属性或者方法就写一次显得有点麻烦，因此我们可以用一个包含所有属性和方法的对象字面量来重写整个原型对象：

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

}

Student.prototype = {

hobby: 'study',

sayHi: function () {

console.log('hi');

},

};

var s1 = new Student('wangwu', 18, 'male');

console.log(Student.prototype.constructor === Student); // 结果为 false

但是这样写也有一个问题，那就是原型对象丢失了 constructor 成员。所以为了保持 constructor 成员的指向正确，建议的写法是：

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

}

Student.prototype = {

constructor: Student, // 手动将 constructor 指向正确的构造函数

hobby: 'study',

sayHi: function () {

console.log('hi');

},

};

var s1 = new Student('wangwu', 18, 'male');

console.log(Student.prototype.constructor === Student); // 结果为 true

#### 原型链继承

我们都听过这么一句话：子承父业。而在我们的 JavaScript 中也有继承，接下来我们会学习原型链继承。原型链继承的主要思想是利用原型让一个引用类型继承另外一个引用类型的属性和方法。

function Student(name, age, gender) {

this.name = name;

this.age = age;

this.gender = gender;

}

Student.prototype.sayHi = function () {

console.log('hi');

};

var s1 = new Student('zhangsan', 18, 'male');

s1.sayHi(); // 打印 hi

var s2 = new Student('lisi', 18, 'male');

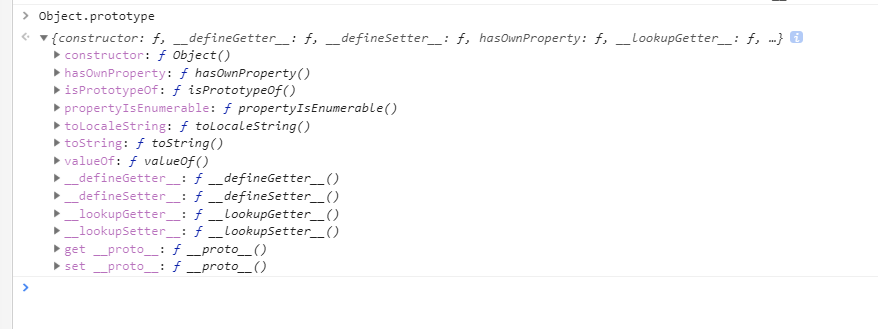
s2.sayHi(); // 打印 hi

上述例子中实例化对象 s1 和 s2 都继承了 sayHi() 方法。

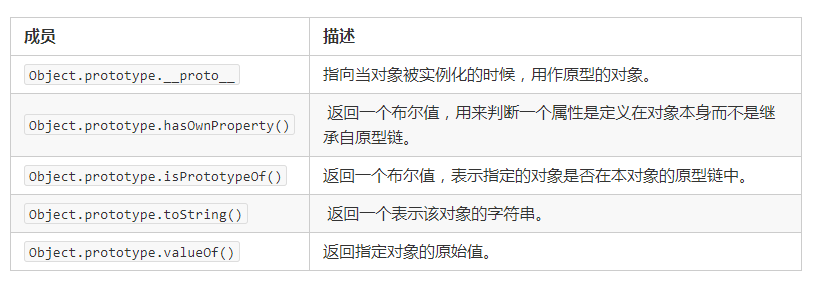
#### Object.prototype 成员介绍

在控制台中输入以下代码：

Object.prototype;



我们介绍常用的几个 Object.prototype 成员：



# CSS

## CSS 语法

CSS 的语法单元是样式，每个样式包含两部分内容：选择器和声明（或者称为规则）：

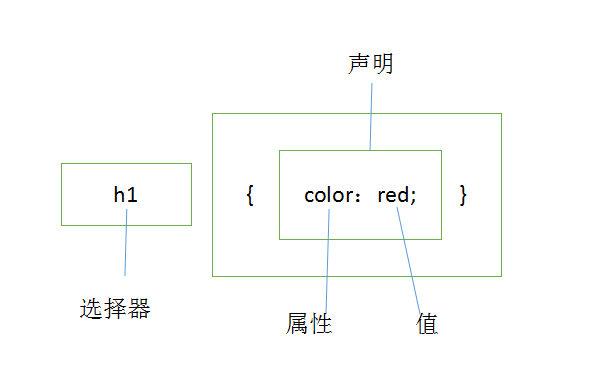
例子：

h1 {

color: red;

}

注：上述例子中 h1 是选择器，color 是属性，red 是属性值。



CSS 语法结构为：选择器{属性：值;属性：值;...}

* 选择器：通常是你需要设置样式的 HTML 元素。
* 属性：是你需要设置的样式属性，比如宽高，颜色，大小等。
* 属性值：你需要设置的样式属性的具体值，比如宽高具体是多少 px，颜色具体是什么颜色。

## CSS 数值与单位

你会在很多地方看到 CSS 单位中使用到数值。下面为大家介绍数值的最常用类别。

#### 颜色值

在网页中的颜色设置是非常重要，有字体颜色，背景颜色，边框颜色等，设置颜色的方法也有很多：

**1.英文命名颜色**

p {

color: pink;

}

/\*给p标签的字体颜色设置为粉色\*/

**2.RGB 颜色**

由 Red、Green、Blue 三种颜色的比例来配色，简称 RGB。

p {

color: rgb(154, 32, 432);

}

每一项的值可以是 0~255 之间的整数，也可以是 0~100% 的百分数，如：

p {

color: rgb(30%, 20%, 40%);

}

**3.十六进制颜色**

现在较为普遍的颜色使用法，其原理其实也是 RGB 设置，但是其每一项的值由 0~255 变成了十六进制 00-ff。

p {

color: #00eeff;

}

#### 长度值

长度单位总结了几种常用单位：px，em，%，这三种单位都是相对单位。

**1.像素**

px 像素（Pixel）。相对长度单位，像素 px 是相对于显示器屏幕分辨率而言的。

特点：

* IE 无法调整那些使用 px 作为单位的字体大小。
* 国外的大部分网站能够调整的原因在于其使用了 em 或 rem 作为字体单位。

**2.em**

就是本元素给定字体的 font-size 值，如果元素的 font-size 为 14px，那么 1em=14px。如果 font-size 为 18px，那么 1em=18px。

p {

font-size: 14px;

line-height: 2em;

}

/\*在这里的line-height行高被设置成了14\*2=28px\*/

**3.百分比**

p {

font-size: 14px;

line-height: 120%;

}

/\*行高为：14\*1.2=16.8\*/

## CSS 样式类型

CSS 样式类型分为三种：内联式、内嵌式、外链式。

#### 内联式(行内式)

通过标签的 style 属性来设置元素的样式，语法格式为：

<标签名 style="属性1:属性值1; 属性2:属性值2; ..."> 内容 </标签名>

例子：

<p style="color:blue;">哈喽，要保持微笑哦。</p>

优点：十分灵活，书写方便，权重高（后面会提到）。

缺点：只能操作某一个标签，没有实现样式和结构相分离。

#### 内嵌式（内部样式表）

将 CSS 代码集中写在 HTML 文档的 head 头部标签中，并且用 style 标签定义。语法格式为：

<head>

<style type="text/css">

选择器 {属性1:属性值1; 属性2:属性值2; ...}

</style>

</head>

例子：

<style type="text/css">

p{color:blue;}

</style>

优点：可以通过一条语句操作多个标签或类。

缺点：只能控制一个页面，没有彻底实现样式和结构分离。

#### 外链式（外部样式表）

将所有的样式放在一个或多个以 .CSS 为扩展名的外部样式表文件中，通过 link 标签将外部样式表文件链接到 HTML 文档中。语法格式为：

<head>

<link href="CSS文件的路径" rel="stylesheet" />

</head>

注：href 定义所链接外部样式表文件的 URL，可以是相对路径，也可以是绝对路径。rel 定义当前文档与被链接文档之间的关系，在这里需要指定为 “stylesheet”，表示被链接的文档是一个样式表文件。

例子：

首先我们要创建一个 css 文件，比如：main.css。然后在里面就能直接写 css 代码了。比如:

p {

color: blue;

}

其次我们在 html 文件中引入这个 css 文件。

<link rel="stylesheet" href="main.css" />

优点：一个单独的 CSS 文件，多个 HTML 文件可以引用一个 CSS 样式表文件。HTML 代码和 CSS 代码分离,要写什么就在哪个文件去找，修改方便。

注：这样写的前提是 HTML 页面与 CSS 文件在同一文件夹下。行内样式表一般写在标签头部，内嵌式样式表、外联式一般写在 <head></head> 标签内。为了编码规范，希望大家尽量使用外联式来写我们的 CSS 代码。

#### 三种方法的优先级

我们学了三种 CSS 样式类型，那么如果对于同一个元素，我们同时使用了三种方法设置了 CSS 样式，这个时候，哪个方法是有效的呢 ？

我们可以记住一个优先级：

内联式 > 嵌入式 > 外链式

但是“嵌入式 > 外链式” 有一个前提：嵌入式 CSS 样式的位置一定在外链式后面。

## 挑战：制作谷歌图标

通过我们前面所学的 CSS 内容，来制作一个谷歌图标的小案例。



源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<meta charset="utf-8" />

</head>

<body>

<p style="font-size:60px">

<span style="color:blue">G</span>

<span style="color:red">o</span>

<span style="color:orange">o</span>

<span style="color:blue">g</span>

<span style="color:green">l</span>

<span style="color:red">e</span>

</p>

</body>

</html>

注：span 是一个行内标签，它能够不换行显示。

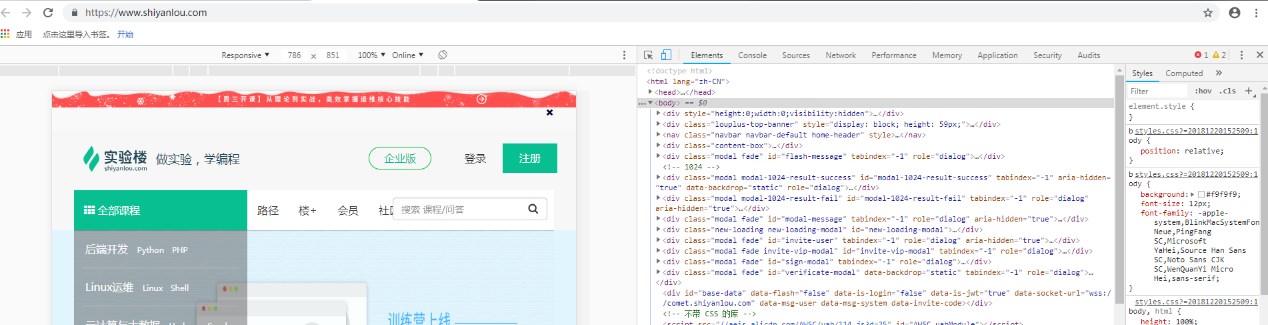
## 学习方法-通过浏览器进行 CSS 排错和调试代码

和 HTML 一样，在 CSS 中，如果选择器是无效的，那么它就不会选择任何东西，而整个规则也不会再做任何事情，浏览器只会继续执行下一个规则，而不是直接崩溃报错。

这使得就算我们的代码有错，但是依然能够运行显示给用户，只是显示的样式可能和我们预期的不一样。但是，当你想要去修改这个错误的时候，似乎会显得无从下手，因为我们甚至都不知道错在哪个地方，当代码量庞大的时候，一行一行去找显然是不可取的。

那么如何在浏览器中调试我们的 CSS 代码 或者查看别人网站的 CSS 代码呢？下面将为大家一一讲解。

* 首先在浏览器中运行我们的前端代码，或者直接进去别人的网站，此处以[实验楼](https://www.lanqiao.cn/)为例。
* 然后我们点击 F12 或者点击鼠标右键再点检查，进入调试模式。如下图所示：



* 最右边 styles 里面的代码就是我们的 CSS 代码了。
* 如果是自己的项目，可以直接在 styles 里面修改我们的 CSS 代码，使得样式调到我们所需要的，最后再复制调试好的代码，到我们的本地。大大减少了我们本地改一次 CSS 代码，运行一次的时间，提高编程效率。
* Elements 那一行最左边的pic这个图标用鼠标左键选中，直接点击页面，就能指定显示出那一块的样式。
* styles 里面特别要注意有感叹号或者代码上面有一条线划掉的。这说明这些样式未成功生效。

## 文字排版

**字体**

通过 font-family 属性设置字体种类，注意不要设置不常用的字体，因为如果用户本地电脑上如果没有安装你设置的字体，就会显示浏览器默认的字体。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

body {

font-family: 'yahei';

/\*微软雅黑\*/

}

</style>

</head>

<body>

<p>实验楼</p>

</body>

</html>

下面再给大家介绍几种几乎所有系统都能够支持的几种字体：Arial，Courier New，Georgia，Times New Roman，Trebuchet MS，Verdana。

**字体大小和颜色**

字体大小通过 font-size 属性设置。字体的常用单位是：px，em。字体颜色通过 color 属性来设置。

<style>

body {

font-family: 'yahei';

font-size: 40px;

color: red;

}

</style>

<body>

<p>实验楼</p>

</body>

注意：这里只展示 <style>标签和 <body> 标签的内容，其他部分未改变的不做展示，接下来的例子都是这样，不再做提示。

**字体样式-粗体**

通过 font-weight 属性设置字体的粗细。常用的有 normal：字体正常，bold：文字加粗。

<style>

body {

font-size: 40px;

color: red;

font-weight: bold;

}

</style>

<body>

<p>实验楼</p>

</body>

**字体样式-斜体**

通过 font-style 属性设置文字格式。

* normal: 将文本设置为普通字体 (将存在的斜体关闭)。
* italic: 如果当前字体的斜体版本可用，那么文本设置为斜体版本；如果不可用，那么会利用 oblique 状态来模拟 italics。
* oblique: 将文本设置为斜体字体的模拟版本，也就是将普通文本倾斜的样式应用到文本中。

body {

font-size: 40px;

color: red;

font-style: italic;

}

**字体装饰**

通过 text-decoration 属性设置或者取消字体上的文本装饰，我们一般会使用这个属性去掉链接上的默认下划线。可用值为：

* none: 取消已经存在的任何文本装饰。
* underline: 文本下划线。
* overline: 文本上划线。
* line-through: 穿过文本的线（删除线）。

body {

font-size: 40px;

color: red;

font-style: italic;

text-decoration: underline;

}

其他的属性可以自行动手尝试。

**段落排版**

**缩进**

段落缩进使用 text-indent 属性，中文文字中的段前习惯空两个文字的空白。

<style>

body {

font-size: 14px;

color: red;

font-style: italic;

text-decoration: underline;

text-indent: 2em;

}

</style>

<body>

<p>实验楼是一个很好的学习网站</p>

</body>

**行高**

可以使用 line-height 属性设置段落行高。

<style>

body {

font-size: 14px;

color: red;

}

</style>

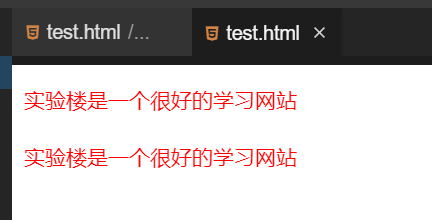
<body>

<p>实验楼是一个很好的学习网站</p>

<p style="line-height:30px">实验楼是一个很好的学习网站</p>

</body>

当两个段落一个设置行高，一个未设置行高时的效果：



大家可以不设置行高看看效果。

**中文字间距、字母间距**

可以使用 letter-spacing 属性设置文字间距或者字母间距。

<style>

body{

font-size:14px;

color:red;

letter-spacing:2em;

}

</style>

</head>

<body>

<p>实验楼是一个很好的学习网站</p>

</body>

**对齐**

如果想为块状元素中的文本、图片设置样式居中，可以使用text-align 属性。

<style>

body{

font-size:14px;

color:red;

text-align:center;

}

</style>

</head>

<body>

<p>实验楼是一个很好的学习网站</p>

</body>

同样你也可以使用 text-align:left 设置左对齐，使用 text-align:right 设置右对齐。

## 列表特定样式

* 符号样式：用 list-style-type 设置用于列表的项目符号的类型，例如无序列表的方形或圆形项目符号，或有序列表的数字，字母或罗马数字。常见的取值有：
  + disc 无序列表的默认值，实心圆。
  + circle 空心圆。
  + square 实心方块。
  + decimal 有序列表的默认值阿拉伯数字。
  + lower-alpha 小写英文字母。
  + upper-alpha 大写英文字母。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

ul {

list-style-type: circle;

}

ol {

list-style-type: lower-alpha;

}

</style>

</head>

<body>

<ul>

<li>111</li>

<li>222</li>

<li>333</li>

</ul>

<ol>

<li>555</li>

<li>666</li>

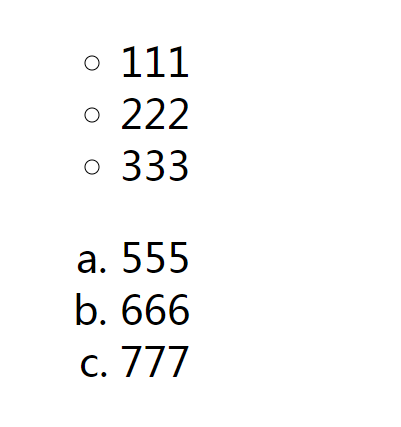
<li>777</li>

</ol>

</body>

</html>

效果为：



* 项目符号位置：通过 list-style-position 设置在每个项目开始之前，项目符号是出现在列表项内，还是出现在其外。默认值为 outside，这使项目符号位于列表项之外。如果值设置为 inside，项目条目则位于行内。我们把前面的例子设置为 inside 来看看效果,为了便于查看效果，我们为 ul 和 li 设置背景色。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

ul {

list-style-type: circle;

list-style-position: inside;

background-color: red;

}

ol {

list-style-type: lower-alpha;

}

li {

background-color: gold;

}

</style>

</head>

<body>

<ul>

<li>111</li>

<li>222</li>

<li>333</li>

</ul>

<ol>

<li>555</li>

<li>666</li>

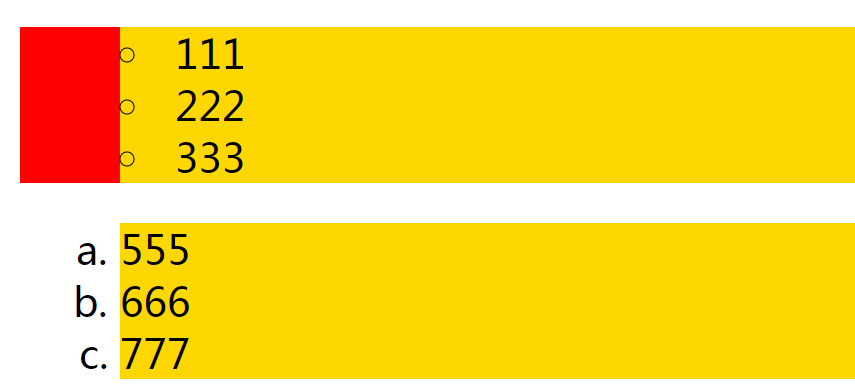
<li>777</li>

</ol>

</body>

</html>

效果为：



### 样式化链接

选择器中有以下几种伪类 :link、 :hover、 :active、 :visited， 我们也可以为伪类设置样式。

首先我们写一个链接，来看看它默认的一些样式：

<a href="#">我是链接</a>

效果图：



我们发现以下特点：

* 链接具有下划线。
* 未访问过的 (Unvisited) 的链接是蓝色的。
* 访问过的 (Visited) 的链接是紫色的。
* 悬停 (Hover) 在一个链接的时候鼠标的光标会变成一个小手的图标。
* 激活 (Active) 链接的时候会变成红色 (用鼠标左键点击链接)。

知道链接的默认样式后，我们同样可以使用 color 属性改变链接不同状态的颜色，使用 text-decoration 属性去掉链接默认的下划线。一般来说对于链接的样式，我们掌握这两个就够了。如果你有兴趣把自己的链接添加一些独特的样式，你可以自己尝试一下哟。

<style>

a:hover {

color: red;

text-decoration: none;

/\*鼠标经过，字体颜色变成红色，并且去掉下划线\*/

}

</style>

<body>

<a href="https://www.lanqiao.cn/">实验楼</a>

</body>

## Web 字体

对于字体的使用来说，有时候我们可能并不仅仅满足于使用自带的字体。而通过 Web 字体，我们可以使用更多的不同的文本样式。

首先我们通过 @font-face 指定要下载的字体文件。

@font-face {

font-family: 'Bitstream Vera Serif Bold';

src: url('http://developer.mozilla.org/@api/deki/files/2934/=VeraSeBd.ttf');

}

然后使用 @font-face 中指定的字体种类名称来将你的定制字体应用到你需要的地方，比如说：

body {

font-family: 'Bitstream Vera Serif Bold', serif;

}

完整的代码为：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

@font-face {

font-family: 'Bitstream Vera Serif Bold';

src: url('http://developer.mozilla.org/@api/deki/files/2934/=VeraSeBd.ttf');

}

body {

font-family: 'Bitstream Vera Serif Bold', serif;

}

</style>

</head>

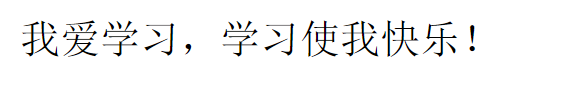
<body>

<div>我爱学习，学习使我快乐！</div>

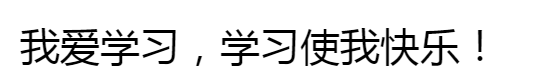
</body>

</html>

使用 Web 字体后在浏览器中运行的效果图为：



再来看看没有使用之前是什么样子的：



注：现在网上有很多免费可以使用的 web 字体，大家可以自行百度搜索下载使用在自己的网页当中。比如：[Google Web Fonts](https://fonts.google.com/)。

## CSS 样式化区块

### 元素分类

在 HTML 中的标签元素大体被分为三类：块级元素、行内元素、行内块元素。

常用的块级元素：

<div>

<p></p>

<h1>

...

<h6>

<ol>

<ul>

<table>

<address>

<blockquote>

<form></form>

</blockquote>

</address>

</table>

</ul>

</ol>

</h6>

</h1>

</div>

常见的行内元素：

<a>

<span>

<br />

<i>

<em>

<strong> <label></label></strong></em></i></span

></a>

常见的行内块元素：

<img /> <input />

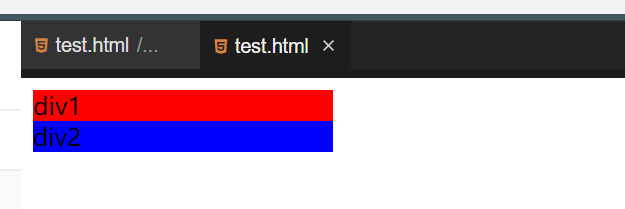
**块级元素**

块级元素，顾名思义，该元素呈现“块”状，所以它有自己的宽度和高度，也就是可以自定义 width 和 height，除此之外，块级元素独自占据一行高度（float 浮动除外），一般可以作为其他容器使用，可容纳块级元素和行内元素。块级元素有以下特点：

1. 每个块级元素都从新一行开始，并且其后的元素都另起一行。
2. 元素的高度、宽度、行高以及顶和底边间距都可以设置。
3. 元素宽度在不设置的情况下，是它本身父容器的 100%（和父元素宽度一致）。

<div style="background:red;width:200px;">div1</div>

<div style="background:blue;width:200px;">div2</div>



**行内元素**

行内元素不可以设置宽（width）和高（height），但可以与其他行内元素位于同一行，行内元素内一般不可以包含块级元素。行内元素的高度一般由元素内部的字体大小决定，宽度由内容的长度控制。行内元素有以下特点：

1. 和其他元素在一行上。
2. 元素的高度、宽度、行高以及顶和底边间距都不可以设置。
3. 元素的宽度就是它包含的文字或图片的宽度，不可改变。

<style>

a,

span {

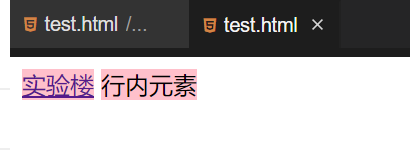
background: pink;

}

</style>

<a href="https://www.lanqiao.cn/">实验楼</a>

<span>行内元素</span>



**行内块元素**

行内块级元素，它既具有块级元素的特点，也有行内元素的特点，它可以自由设置元素宽度和高度，也可以在一行中放置多个行内块级元素。比如 input、img 就是行内块级元素，它可以设置高宽以及一行多个。具体特点如下：

1. 和其他元素在一行上。
2. 元素的高度、宽度、行高以及顶和底边间距都可以设置。

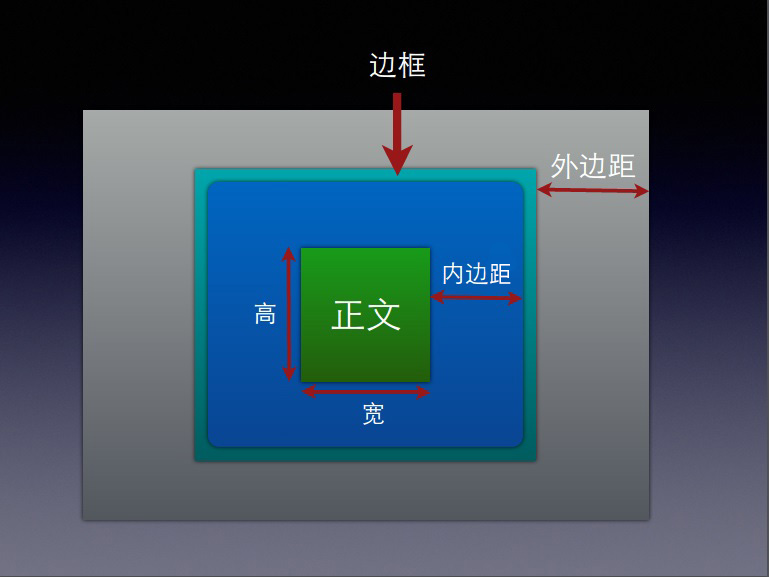
### 区块模型

区块模型也就是我们常说的盒子模型，而所谓盒子模型就是把 HTML 页面中的元素看作是一个矩形的盒子，也就是一个盛装内容的容器。

盒模型的宽度和高度和我们平常所说的物体的宽度和高度是不一样的。CSS 定义的宽和高，指的是填充以里的内容范围。

因此盒子的宽度=左外边距+左边框+左内边距+内容宽度+右内边距+右边框+右外边距。

每个矩形都由元素的内容（content）、内边距（padding）、边框（border）和外边距（margin）组成。我们用一张图来表示：



#### padding（内边距）

padding 内边距位于内容框的外边缘与边界的内边缘之间。padding-top:上内边距，padding-right:右内边距，padding-bottom:下内边距，padding-left:左内边距。设置内边距的写法：

* padding:5px 10px 15px 20px;表示上内边距是 5 px，右内边距是 10 px，下内边距是 15 px，左内边距是 20 px。
* padding:5px 10px 15px;表示上内边距是 5 px，右内边距和左内边距是 10 px，下内边距是 15 px。
* padding:5px 10px;表示上内边距和下内边距是 5 px，右内边距和左内边距是 10 px。
* padding:10px;表示所有 4 个内边距都是 10 px。

注：padding 属性接受长度值或百分比值，但不允许使用负值。如果使用百分比值，百分数值是相对于其父元素的 width 计算的。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

#box1 {

width: 200px;

height: 200px;

background-color: red;

padding: 20px 10px 10px 50px;

}

#box2 {

width: 200px;

height: 200px;

background-color: blue;

}

</style>

</head>

<body>

<div id="box1"></div>

<div id="box2"></div>

</body>

</html>



可以看到，同样是设置高宽为 200 像素的盒子，有 padding 的明显要大一些，padding 在边框里面。

#### border（边框）

元素的边框 (border) 是围绕元素内容和内边距的一条或多条线。你可以设置它的粗细，样式和颜色。

边框语法：

border: border-width||border-style||border-color;

例子：

div {

border: 2px solid red;

}

这条代码能给 div 设置边框粗细为 2px、样式为实线、颜色为红色的边框。

注意：border-style（边框样式）的常用样式有：dashed（虚线）、dotted（点线）、solid（实线）.

你可以按照上右下左的格式，分别写出元素的各个边框的属性。

例子：

div {

border-style: solid;

border-width: 15px 5px 15px 5px;

}

也可以简写为：

div {

border-style: solid;

border-width: 15px 5px;

}

注：和前面 padding 的简写方式一样，有遗忘的可以往前面看看哟。

也可以单独设置一个方向的边框：

div {

border-top: 1px solid red;

}

/\*只给上边框设置一像素的红色实线\*/

以此类推：

border-right：右边框

border-left：左边框

border-bottom：下边框

边框的颜色可以使用任何类型的颜色值，例如可以是命名颜色，也可以是十六进制和 RGB 值：

p {

border-style: dashed;

border-color: yellow rgb(25%, 35%, 45%) #909990 red;

}

注：默认的边框颜色是元素本身的前景色。如果没有为边框声明颜色，它将与元素的文本颜色相同。另一方面，如果元素没有任何文本，假设它是一个表格，其中只包含图像，那么该表的边框颜色就是其父元素的文本颜色（因为 color 可以继承）。这个父元素很可能是 body、div 或另一个 table。

#### margin(外边距)

外边距（margin）代表 CSS 框周围的外部区域，称为外边距。和 padding 类似，也有 margin-top、margin-right、margin-bottom 和 margin-left。写法仿照 padding。

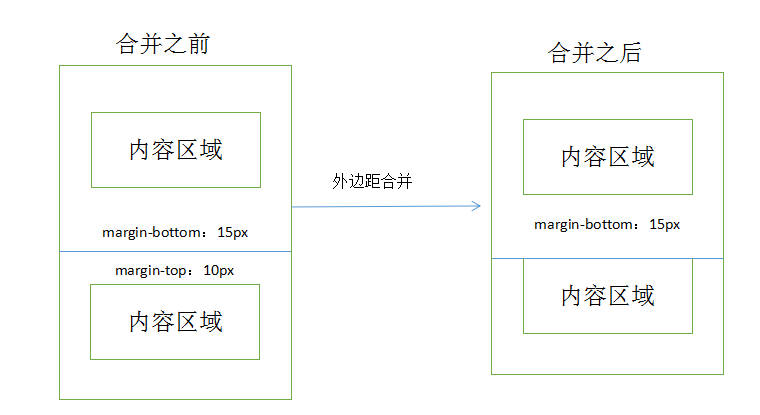
注：与 padding 不同的是 margin 可以是负值。

外边距合并问题：

简单地说，外边距合并指的是，当两个垂直外边距相遇时，它们将形成一个外边距。合并后的外边距的高度等于两个发生合并的外边距的高度中的较大者。

当一个元素出现在另一个元素上面时，第一个元素的下外边距与第二个元素的上外边距会发生合并。

外边距合并示例图：



#### 高级的框操作

溢流：当你使用绝对的值设置了一个框的大小（如，固定像素的宽/高），允许的大小可能不适合放置内容，这种情况下内容会从盒子溢流。我们使用 overflow 属性来控制这种情况的发生。常用的值：

* auto:当内容过多的时候溢流的内容被隐藏，然后出现滚动条，让我们滚动查看所有的内容。
* hidden: 当内容过多，溢流的内容被隐藏。
* visible: 当内容过多，溢流的内容被显示在盒子的外边。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

width: 500px;

height: 2.5em;

padding: 1em 1em 1em 1em;

border: 1px solid black;

}

.one {

overflow: auto;

}

.two {

overflow: hidden;

}

.three {

overflow: visible;

}

</style>

</head>

<body>

<p class="one">

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

</p>

<p class="two">

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

</p>

<p class="three">

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

我爱学习，学习使我快乐！我爱学习，学习使我快乐！

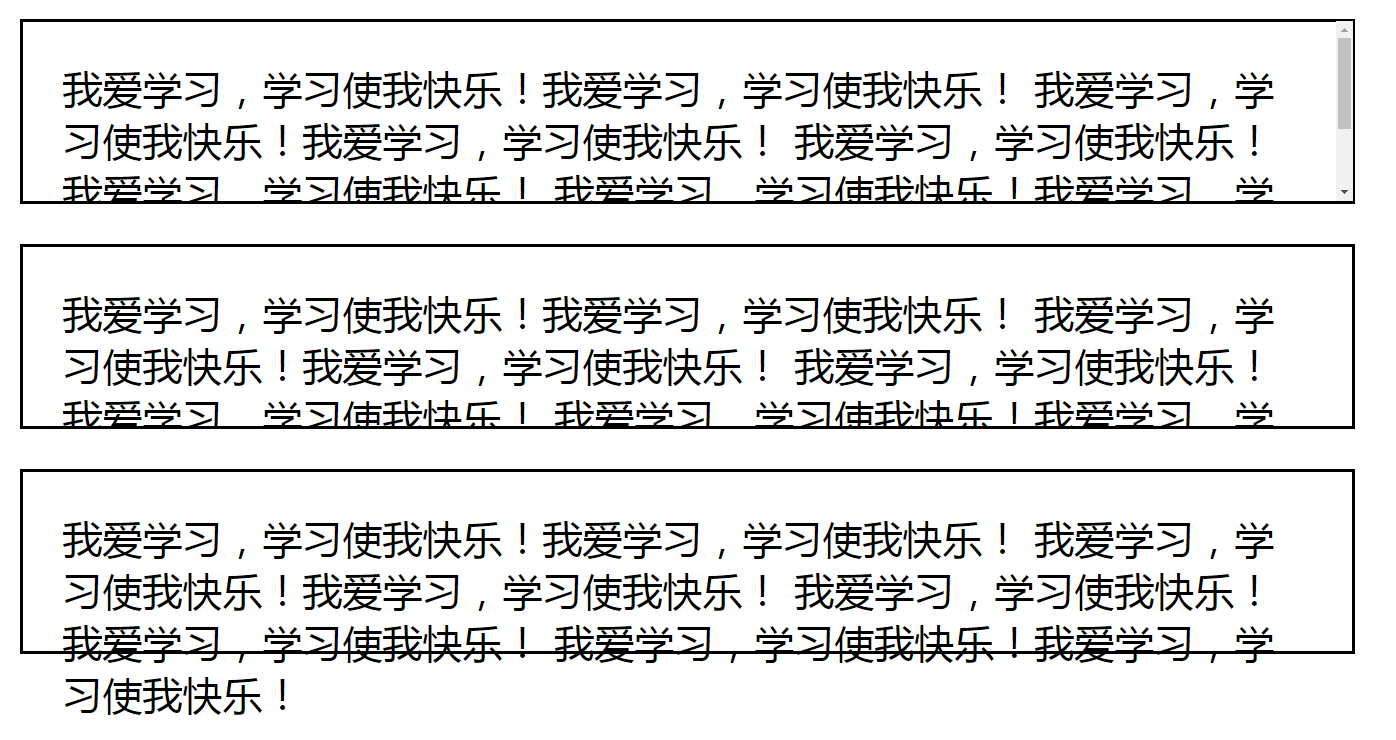
我爱学习，学习使我快乐！我爱学习，学习使我快乐！

</p>

</body>

</html>

在浏览器运行效果展示图：



### 样式化区块

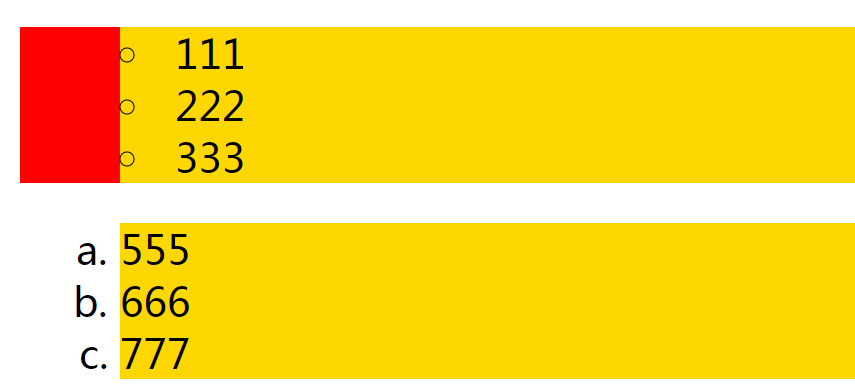
样式化区块其实就是对我们前面所学到的盒子模型，进行样式化的操作，包括操纵其背景颜色、图像、边框等。

#### 背景

元素的背景是指，在元素内容、内边距和边界下层的区域。而在我们前面学习样式化列表的时候已经接触过 background 属性了。



* 背景颜色：通过 background-color 属性设置，首先，大多数元素的背景色是 transparent（透明色），而不是 white（白色），设置了背景颜色后，元素的内容依然是可读的，很多时候，我们需要都需要设置背景色来帮助我们，比如前面介绍样式化列表的 list-style-position 属性 inside 值的效果就使用到了背景色。



注：源码在样式化列表章节，此处的图片是为了演示背景色的作用与重要性。

* 背景重复：background-repeat 允许您指定背景图像是如何重复的。默认值是 repeat，图像将在整个背景中水平和竖直地重复。但是大多数情况下我们是不需要重复的，因此经常在使用背景图片的时候设置为 no-repeat。此外还有两个值，repeat-x: 图像将在整个背景中水平地重复。repeat-y: 图像会在背景下垂直地重复。
* 背景位置：background-position 允许我们在背景中任意位置放置背景图像。除了上面图片所给的值，我们还可以自定义值，使用两个通过空格分格的值，指定图像的水平（x）和垂直（y）坐标，图像的左上角为原点（0,0），x 坐标从左到右，y 坐标从上到下。
* background-image 还有一组可用的值：颜色渐变。通过 linear-gradient() 函数设置，函数至少需要用逗号分隔的三个参数——背景中渐变的方向，开始的颜色和结尾的颜色。例如：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.box {

width: 100px;

height: 100px;

background-image: linear-gradient(to bottom, red, blue);

}

</style>

</head>

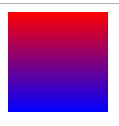
<body>

<div class="box"></div>

</body>

</html>

在浏览器中运行效果图为：



注：这个渐变将从上到下，从顶部的红色开始，然后平稳过渡到底部的蓝色。 渐变的方向可以通过关键字来指定方向 （to bottom，to right， to bottom right 等）或者使用角度值 (0 deg 相当于 to top，90 deg 相当于 to right，直到 360 deg，它再次相当于 to top ）。

#### 图片

我们在 HTML 模块中学习了 img 标签来添加图片，现在我们来了解它的一些常用的 CSS 属性。

* height：定义图片的高度。
* width：定义图片的宽度。
* border：定义图片周围的边框。

例子：

<img src="/实验楼.jpg" width="px" height="200px" border="1" />

注：上述例子中图片引用地址为相对地址

#### 表格

我们前面已经学习了如何制作表格，现在我们来讲讲表格的一些样式，让我们的表格更加美观。

* 表格边框：同样的我们使用 border 属性为表格设置边框。

例子：

table,

th,

td {

border: 2px solid red;

}

* 折叠边框：使用 border-collapse 属性设置是否将表格边框折叠为单一边框。

例子：

table {

border-collapse: collapse;

}

* 表格的宽度和高度：正如你所预料的一样，我们使用 width 和 height 属性设置表格的高度和宽度。

例子：

table {

width: 50%;

}

th {

height: 50px;

}

* 表格文本对齐的方式：通过 text-align 属性设置水平对齐方式，取值有 left（左对齐）right（右对齐）和 center（居中对齐），通过 vertical-align 属性 设置垂直对齐方式，取值有：top（顶端对齐）bottom（底部对齐）和 middle（居中对齐）。

例子：

td {

height: 50px;

vertical-align: top;

}

****

* 表格内边距：通过为 td 和 th 元素设置 padding 属性，控制表格中内容和边框的距离。

例子：

td {

padding: 30px;

}

****

* 表格颜色：通过 color 设置表格文本颜色，通过 background-color 设置表格背景颜色。

例子：

table,

td,

th {

border: 1px solid blue;

}

th {

background-color: red;

color: yellow;

}

****

* 表格标题位置：通过 caption-side 属性设置表格的标题位置，它的取值有：top（默认值。把表格标题定位在表格之上），bottom（把表格标题定位在表格之下）和 inherit（规定应该从父元素继承 caption-side 属性的值）。

表格综合案例：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

table,

td {

border: 1px solid blue;

}

td {

background-color: red;

color: yellow;

text-align: left;

vertical-align: bottom;

padding: 30px;

}

table {

width: 200px;

height: 100px;

border-collapse: collapse;

}

caption {

caption-side: bottom;

}

</style>

</head>

<body>

<table>

<caption>

我是标题

</caption>

<tr>

<td>one</td>

<td>two</td>

</tr>

<tr>

<td>three</td>

<td>four</td>

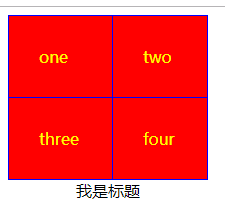
</tr>

</table>

</body>

</html>

在浏览器中的运行效果图为：



#### 轮廓

轮廓（outline）是绘制于元素周围的一条线，位于边框边缘的外围，可起到突出元素的作用。

* outline-color 设置轮廓的颜色。取值和其余颜色的取值一样。
* outline-style 设置轮廓的样式。取值如下：



* outline-width 设置轮廓的宽度。它的值有：
  + thin 规定细轮廓。
  + medium 默认取值规定中等的轮廓。
  + thick 规定粗的轮廓。
  + length 允许你自定义轮廓粗细的值。
  + inherit 规定应该从父元素继承轮廓宽度的设置。
* outline 上面三个属性的连写。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

p {

outline: red dotted thick;

width: 100px;

height: 50px;

}

</style>

</head>

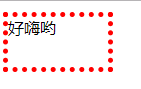
<body>

<p>好嗨哟</p>

</body>

</html>

在浏览器中运行的效果为：



#### 盒子阴影

通过 box-shadow 属性设置盒子阴影。box-shadow 有四个值:

* 第一个值是水平偏移量（水平阴影）：即向右的距离，阴影被从原始的框中偏移(如果值为负的话则为左)。
* 第二个值是垂直偏移量（垂直阴影）：即阴影从原始盒子中向下偏移的距离(或向上，如果值为负)。
* 第三个值是模糊半径（影子大小）：即在阴影中应用的模糊度。
* 第四个值是阴影的基本颜色。你可以使用任何长度和颜色单位来定义这些值。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

.shadow {

box-shadow: 5px 5px 5px red;

width: 200px;

height: 100px;

background-color: blue;

}

</style>

</head>

<body>

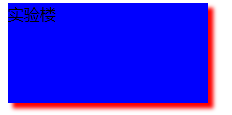
<div class="shadow">实验楼</div>

</body>

</html>

****

在浏览器中的运行效果为：



多个盒子阴影：使用逗号隔开。

例子：

box-shadow: 1px 1px 1px yellow, 2px 2px 1px yellow, 3px 3px 1px blue, 4px 4px

1px blue, 5px 5px 1px black, 6px 6px 1px black;

使用 inset 关键字，把它放在一个影子声明的开始，使它变成一个内部阴影，而不是一个外部阴影。比如我们为前面的例子加上 inset 关键字，最后的运行效果就变成下面这样了：



源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

.shadow {

box-shadow: inset 5px 5px 5px red;

width: 200px;

height: 100px;

background-color: blue;

}

</style>

</head>

<body>

<div class="shadow">实验楼</div>

</body>

</html>

#### 圆角边框

使用 border-radius 属性来创建圆角。

例子：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

.radius {

width: 200px;

height: 200px;

border: 1px;

background-color: red;

border-radius: 50%; /\*将正方形变成圆\*/

}

</style>

</head>

<body>

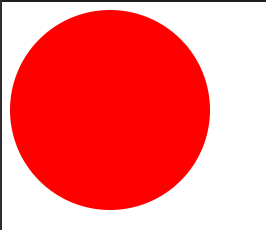
<div class="radius"></div>

</body>

</html>

注：border-radius 的值除了用百分号（%）还可以用 length，比如：border-radius:25px;。

效果图：



#### 挑战：做一个酷炫的盒子

要求：盒子的宽高设置一个合适的，盒子相对于浏览器水平居中，并且距离顶端设置合适的距离，文字相对于盒子水平垂直居中。文字的颜色和盒子颜色一样，增加一个文字阴影让它显示出来。设置圆角边框，设置一个 1 px 的实线边框，边框颜色比盒子略微深一点。设置盒子阴影，一个外部阴影，一个内部阴影。大致要求达到，其余的值设置可以自行调整，不一定非要与案例一样。完成后的大致效果如下：



源码：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.box {

width: 200px;

height: 100px;

color: red;

background-color: red;

text-align: center;

line-height: 100px;

margin: 100px auto;

text-shadow: 2px 2px 5px black;

border-radius: 20px;

border: 1px solid crimson;

box-shadow: 2px 2px 5px black, inset 2px 2px 5px black;

}

</style>

</head>

<body>

<div class="box">我爱学习，身体棒棒！</div>

</body>

</html>

注：为了能让图片看的更清楚，此处的图片是在浏览器中放大了的，上述代码实际运行的话会小很多。

下一步

## CSS 选择器

### 选择器分类

选择器主要可分为以下几类：

* 基础选择器：标签选择器，类选择器，id 选择器，通配符选择器
* 组合选择器：标签指定式选择器，后代选择器，并集选择器
* 属性选择器
* 伪类选择器

### 标签选择器

标签选择器也叫元素选择器。其实就是 html 代码中的标签，比如 <html>、<body>、<h1>、<p>、<img>。

语法格式为：

标签名 {

属性1: 属性值1;

属性2: 属性值2;

...;

}



给个例子，比如我们写个简单的 p 标签，里面随便写一句话，然后我们设置文字的颜色为红色。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: red;

}

</style>

</head>

<body>

<p>实验楼</p>

</body>

</html>

再来看看效果：



### 类选择器

类选择器，是对 HTML 标签中 class 属性进行选择。CSS 类选择器的选择符是 "."。

类选择器命名规范：不能是纯数字，不能是标签名，不建议使用汉字。一般是点+对应样式描述。

例如：.shadow。

同样的拿前面的例子来说，我们为 p 标签 加上一个 class 属性，然后通过 class 属性设置它的背景色为黄色。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: red;

}

.bgc {

background-color: yellow;

}

</style>

</head>

<body>

<p class="bgc">实验楼</p>

</body>

</html>



我们再来看看效果：



值得注意的是文档中的多个元素可以具有相同的类名，而单个元素可以有多个类名(以空格分开多个类名的形式书写)。样式显示效果跟 HTML 元素中的类名先后顺序没有关系，受 CSS 样式书写的上下顺序有关(具体在后面 CSS 层叠性和继承性中会讲到)。以下是一个简单的例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: red;

}

.bgc {

background-color: yellow;

}

.one {

font-size: large;

}

</style>

</head>

<body>

<p class="bgc">实验楼</p>

<p class="bgc one">我爱学习</p>

</body>

</html>

在浏览器中运行的效果为：



### id 选择器

id 是对 HTML 标签中 id 属性进行选择。id 选择器的选择符是 "#"。 任何元素都可以使用 id 属性设置唯一的 id 名称。这是选择单个元素的最有效的方式。

特别需要注意的是 id 选择器具有唯一性，也就是说同一个 HTML 页面不能有相同的 id 名称(使用多个相同的 id 选择器，浏览器不会报错但是不符合 W3C 标准)。

可以这样理解：一个人的名字（class 类名）可以有很多重名的，但是身份证号（id 名称）只能有一个唯一的，不能重复的。

语法格式为：

#id名 {

属性1: 属性值1;

属性2: 属性值2;

...;

}

我们为前面的例子设置一个 id 属性，通过 id 属性来设置高度。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: red;

}

.bgc {

background-color: yellow;

}

#height {

height: 100px;

}

</style>

</head>

<body>

<p class="bgc" id="height">实验楼</p>

</body>

</html>

在浏览器运行的效果图：



### 通配符选择器

通配符选择器用“\*”号表示，他是所有选择器中作用范围最广的，能匹配页面中所有的元素。其基本语法格式如下：

\*{

属性1：属性值1;

属性2：属性值2;

...

}

我们一般使用通配符选择器定义 CSS 样式，清除所有 HTML 标记的默认边距。

\* {

margin: 0; /\* 定义外边距\*/

padding: 0; /\* 定义内边距\*/

}

其他例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

\* {

color: red;

}

</style>

</head>

<body>

<p>实验楼</p>

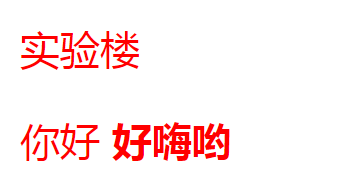
<span>你好</span>

<strong>好嗨哟</strong>

</body>

</html>

在浏览器中运行效果：



### 标签指定式选择器

标签指定式选择器又称交集选择器，由两个选择器构成，其中第一个必须是标签选择器，第二个为 class 选择器或 id 选择器。两个选择器之间不能有空格。简单记忆为：并且的意思。即...又...的意思，比如 p.one 选择的是类名为 one 的段落标签。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: red;

}

p.one {

color: blue;

}

p#two {

color: yellow;

}

</style>

</head>

<body>

<p>实验楼</p>

<p class="one">实验楼</p>

<p id="two">实验楼</p>

</body>

</html>

在浏览器中运行效果为：



### 后代选择器

后代选择器又叫包含选择器，用来选择元素或元素组的后代。其写法就是把外层标记写在前面，内层标记写在后面，中间用空格分隔。当标记发生嵌套时，内层标记就成为外层标记的后代。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.one #two p {

color: red;

}

</style>

</head>

<body>

<div class="one">

<div id="two">

<p>实验楼</p>

</div>

</div>

</body>

</html>

### 子代选择器

与后代选择器相比，子元素选择器只能选择作为某个元素的子元素。子选择器使用大于号作为子结合符。

例如你只想选择 p 元素的子元素 strong 元素，为其设置样式。你可以这么写：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

p > strong {

color: red;

}

</style>

</head>

<body>

<p>

<strong>实验楼</strong>

<strong>实验楼</strong>

</p>

<p>

<em>

<strong>实验楼</strong>

<strong>实验楼</strong>

</em>

</p>

</body>

</html>

在浏览器中运行后的效果为：



注：上述例子中，只有第一个 p 标签中的两个 strong 元素变成了红色，而第二个 p 标签中的 两个 strong 元素不受影响。而如果改成后代选择器，上诉例子中的 style 标签里面改成下面的代码，其余代码不变：

<style>

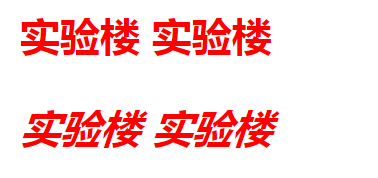
p strong {

color: red;

}

</style>

最终运行效果为：



注：从上述两个例子中我们可以看出，子代选择器只选择指定标签的第一代子元素或者理解为它的直接后代（只指定儿子，而不管孙子），后代选择器是作用于所有子后代元素（不只儿子，还可以选择到孙子，重孙子等等）。

### 并集选择器

并集选择器是各个选择器通过逗号连接而成的，任何形式的选择器（包括标签选择器、class 类选择器、id 选择器等），都可以作为并集选择器的一部分。如果某些选择器定义的样式完全相同，或部分相同，就可以利用并集选择器为它们定义相同的 CSS 样式。简单记忆为：和的意思。

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

p,

#one,

.two {

color: red;

}

</style>

</head>

<body>

<p>我爱学习</p>

<span id="one">我爱运动</span>

<strong class="two">我爱睡觉</strong>

</body>

</html>

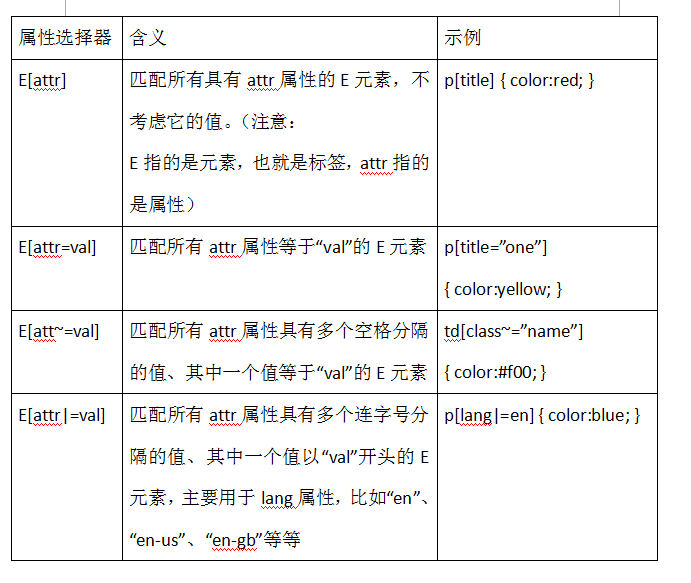
在浏览器中运行效果为：



注：上述例子表示 p 标签和 id 名为 one 的标签 和 class 名为 two 的标签的样式为：颜色为红色。

### 属性选择器

属性选择器是根据元素的属性及属性值来选择元素。常用的几种属性选择器：



例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: blue;

}

p[title='one'] {

color: red;

}

p[title~='two'] {

color: green;

}

p[title|='three'] {

color: pink;

}

</style>

</head>

<body>

<p title="five">我爱学习</p>

<p title="one">我爱运动</p>

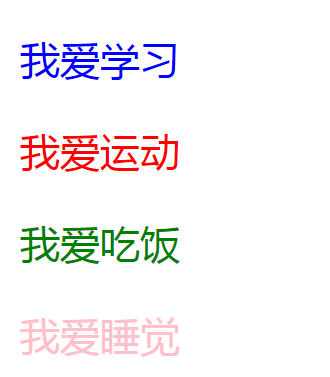
<p title="one two">我爱吃饭</p>

<p title="three-four">我爱睡觉</p>

</body>

</html>

在浏览器中的运行效果为：



### 伪类选择器

CSS 伪类用于向某些选择器添加特殊的效果。下面给大家介绍几个常用的伪类。

* :link 应用于未被访问过的链接。IE6 不兼容，解决此问题，直接使用 a 标签。
* :hover 应用于有鼠标指针悬停于其上的元素。在 IE6 只能应用于 a 连接，IE7+ 所有元素都兼容。
* :active 应用于被激活的元素，如被点击的链接、被按下的按钮等。
* :visited 应用于已经被访问过的链接。

例子：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

/\*a:link{

color: red;

}\*/

/\*链接默认状态\*/

a {

color: red;

text-decoration: none;

}

/\*链接访问之后的状态\*/

a:visited {

color: yellow;

}

/\*鼠标放到链接上显示的状态\*/

a:hover {

color: blue;

text-decoration: line-through;

}

/\*链接激活的状态\*/

a:active {

color: pink;

}

</style>

</head>

<body>

<a href="">实验楼</a>

</body>

</html>

注：我们可能会遇到被点击访问过的超链接不再具有 hover 和 active 了，解决办法是改变 CSS 属性的排列顺序，L-V-H-A（link-visited-hover-active）。您可以运行一下，然后更改 L-V-H-A 的顺序后，看看效果。

## CSS 层叠性和继承性

#### 层叠性

层叠性是指多种 CSS 样式的叠加。而最终什么选择器在层叠中胜出遵循下列规则。

* !important 具有最大优先级。也就是说任何情况下，只要定义了它,它的样式最终就会被应用。

例子：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

#syl {

color: yellow;

font-size: 10px;

}

.box {

color: green;

font-size: 60px;

}

div {

color: red !important;

font-size: 60px !important;

}

</style>

</head>

<body>

<div class="box" id="syl" style="font-size:12px; color:yellow;">实验楼</div>

</body>

</html>



在浏览器中运行效果为：



* 源代码出现次序。

例子：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

.box2 {

font-size: 100px;

color: blue;

}

.box {

font-size: 60px;

color: pink;

}

</style>

</head>

<body>

<div class="box box2 ">实验楼</div>

</body>

</html>

在浏览器中运行效果为：



注：这里的先后次序不是说 class 里面类名的先后次序，而是说定义样式的先后顺序。上述例子中先定义的 box2 类样式，再定义的 box 类样式，最后显示的是 box 类中的样式。

* 优先级：
  + 规则（1）：行内样式 > 页内样式 > 外部引用样式 > 浏览器默认样式。
  + 规则（2）：important > 内联 > id > 伪类|类 | 属性选择 > 标签 > 伪对象 > 通配符 > 继承。另外权重会叠加，所以为了便于理解，权重设置如下： - 1 表示标签选择器的权重；- 10 表示类选择器的权重；- 100 表示 id 选择器的权重；- 1000 表示内联样式的权重。

注：权重相同时，CSS 遵循就近原则。也就是说靠近元素的样式具有最大的优先级，或者说排在最后的样式优先级最大。所有都相同时，声明靠后的优先级大。

例子：

#header #left ul li.first a

权重为100+100+1+1+10+1=213

#### 继承性

继承性就是指我们写 CSS 样式表的时候子元素会继承父元素的某些样式，比如文本颜色和字号。下面我们来看个简单的例子，帮助我们理解：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

.box {

font-size: 15px;

color: red;

}

</style>

</head>

<body>

<div class="box">

<span>实验楼</span>

</div>

</body>

</html>

在浏览器中运行效果为：



注：想要设置一个可继承的属性，只需将它应用于父元素即可。所有字相关的都可以继承，比如：color、text-系列、font-系列、line-系列等。并不是所有的 CSS 属性都可以继承，例如：边框、外边距、内边距、背景、定位、元素宽高属性就不能继承。所有盒子相关的属性都不能继承例如：a 标签，h1 标签不会实现继承。

### 挑战：CSS 层叠性和继承性题

先看代码然后思考一下结果，最后再把代码复制到浏览器中运行看看是不是和你做的结果一样，如果不一致，仔细想想为什么，再去看看前面的内容复习一下吧。

第一题：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

#father #son {

color: red;

}

#father p.two {

color: yellow;

}

</style>

</head>

<body>

<div id="father" class="one">

<p id="son" class="two">请问这行字体是什么颜色的？</p>

</div>

</body>

</html>

第二题：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

p {

color: orange;

}

#father {

color: pink;

}

</style>

</head>

<body>

<div id="father">

<p>请问这行字体是什么颜色的？</p>

</div>

</body>

</html>

第三题：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div p {

color: red;

}

#father {

color: red;

}

p.two {

color: deeppink;

}

</style>

</head>

<body>

<div id="father" class="one">

<p class="two">请问这行字体是什么颜色的？</p>

</div>

</body>

</html>

第四题：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div div div div div div div div div div div div {

color: yellow;

}

.one {

color: blue;

}

</style>

</head>

<body>

<div>

<div>

<div>

<div>

<div>

<div>

<div>

<div>

<div>

<div>

<div>

<div class="one">请问这行文字是什么颜色的</div>

</div>

</div>

</div>

</div>

</div>

</div>

</div>

</div>

</div>

</div>

</div>

</body>

</html>

第五题：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.two {

color: red;

}

.one {

color: blue;

}

</style>

</head>

<body>

<div class="one two">请问这行字体是什么颜色的？</div>

</body>

</html>

第六题：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

#syl {

color: yellow;

}

.box {

color: green;

}

div {

color: red !important;

}

</style>

</head>

<body>

<div class="box" id="syl" style="color:yellow;">

请问这行字体是什么颜色的？

</div>

</body>

</html>

## CSS 布局模型

### 布局模型概述

布局模型与盒模型一样都是 CSS 最基本、 最核心的概念。但布局模型是建立在盒模型基础之上，又不同于我们常说的 CSS 布局样式或 CSS 布局模板。CSS 三种基本网页布局模型：流动模型（Flow）、浮动模型（Float）、层模型（Layer）。

### 流动模型

流动（Flow）是默认的网页布局模式。也就是说网页在默认状态下的 HTML 网页元素都是根据流动模型来分布网页内容的。 流动布局模型具有 2 个比较典型的特征：

1. 块状元素都会在所处的包含元素内，自上而下按顺序垂直延伸分布，因为在默认状态下，块级元素的宽度都为 100%，实际上，块状元素都会以行的形式占据位置（每一个便签都显示着自己本来默认的那个宽高）。
2. 在流动模型下，行内元素都会在所处的包含元素内从左到右水平分布显示（内联元素可不像块级元素独占一行的）。

### 浮动模型

块状元素都是独占一行，如果现在我们想让两个块状元素并排显示，怎么办呢？设置元素浮动就可以实现。 任何元素在默认情况下是不能浮动的，但可以用 CSS 定义为浮动，如 div、p、table、img 等元素都可以被定义为浮动。如下代码可以实现两个 div 元素一行显示。

<head>

<style>

div {

width: 200px;

height: 200px;

border: 2px blue solid;

float: left;

}

</style>

</head>

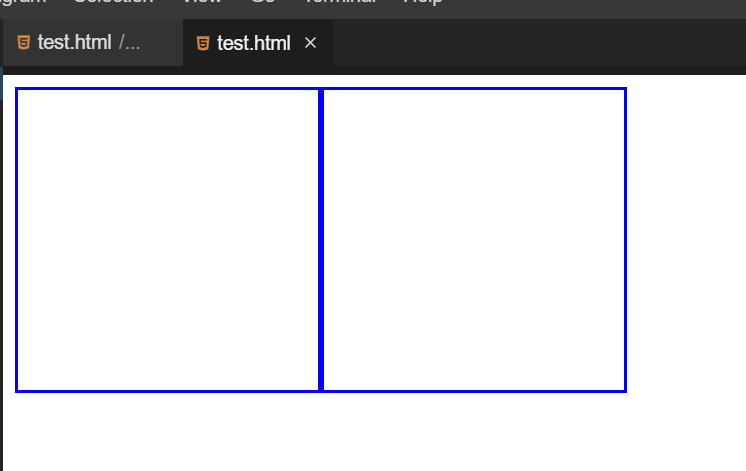
<body>

<div></div>

<div></div>

</body>





#### 脱标

特别需要注意的是，设置了浮动的元素，脱离标准流（脱标）。这样说可能不好理解，我们下面来一个具体的例子。

首先写三个不浮动的 div。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.box {

width: 600px;

height: 300px;

border: 2px dashed green;

}

.box1,

.box2,

.box3 {

width: 200px;

height: 100px;

}

.box1 {

background-color: red;

}

.box2 {

background-color: yellow;

}

.box3 {

background-color: blue;

}

</style>

</head>

<body>

<div class="box">

<div class="box1"></div>

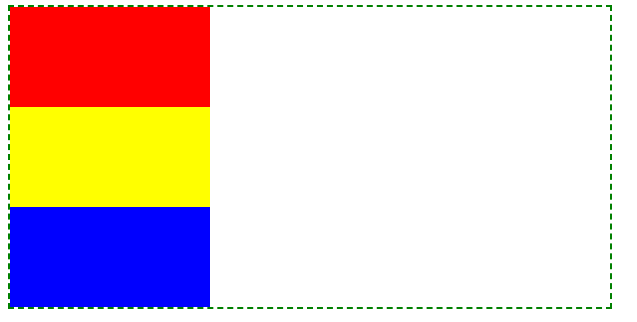
<div class="box2"></div>

<div class="box3"></div>

</div>

</body>

</html>



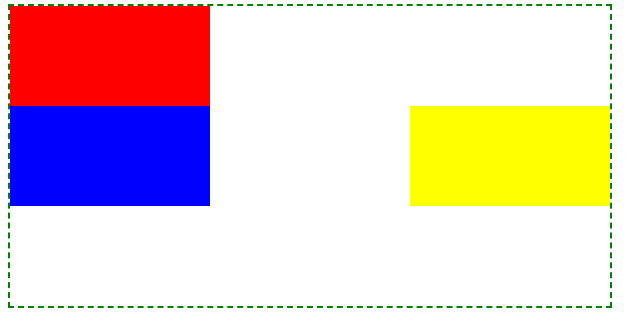
然后我们给 box2 盒子（黄色的盒子），设置右浮动来看看效果：

.box2 {

background-color: yellow;

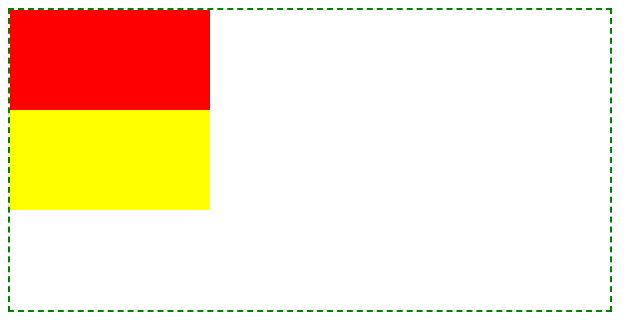
float: right;

}



结论：box2 盒子（黄色的盒子）脱离了文档流并且向右移动，直到它的右边缘碰到包含框的右边缘。而 box3 盒子（蓝色的盒子）则替代了原来 box2 的位置。

给 box2 盒子（黄色的盒子），设置左浮动来看看效果：



结论：box2 盒子（黄色的盒子）脱离了文档流并且左右移动，直到它的左边缘碰到包含框的左边缘。因为它不再处于文档流中，所以它不占据空间，实际上覆盖住了 box3 盒子，使 box3 盒子从视图中消失。我们可以理解为 box2 盒子像羽毛一样飘在 box3 盒子上面，挡住了 box3 盒子。

### 清除浮动

当父容器没有设置高度，里面的盒子没有设置浮动的情况下会将父容器的高度撑开。一旦父容器中的盒子设置浮动，脱离标准文档流，父容器立马没有高度，下面的盒子会跑到浮动的盒子下面。出现这种情况，我们需要清除浮动。

首先我们写一个示例，不设置浮动，第一个盒子的父元素不设置高度。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.box {

width: 600px;

border: 2px dashed green;

}

.box1,

.box2 {

width: 200px;

height: 100px;

}

.box1 {

background-color: red;

}

.box2 {

background-color: blue;

}

</style>

</head>

<body>

<div class="box">

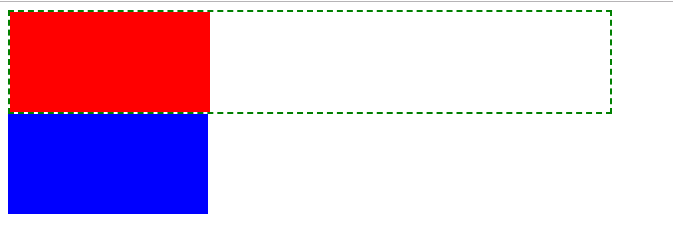
<div class="box1"></div>

</div>

<div class="box2"></div>

</body>

</html>



然后我们修改上面的例子，给 box1 盒子设置一个左浮动。再来看看效果。



我们发现 box2 盒子跑到 box1 盒子下面去了，这个时候我们就需要清除浮动带来的不利影响了。

清除浮动的方法有很多，这里给大家介绍一个最常见的用得最多的方法：使用伪元素来清除浮动，是在浮动元素的父元素上设置伪元素，此处是 box 上设置。

.clearfix:after {

content: '';

/\*设置内容为空\*/

height: 0;

/\*高度为0\*/

line-height: 0;

/\* 行高为0\*/

display: block;

/\*将文本转为块级元素\*/

visibility: hidden;

/\*将元素隐藏\*/

clear: both; /\*清除浮动\*/

}

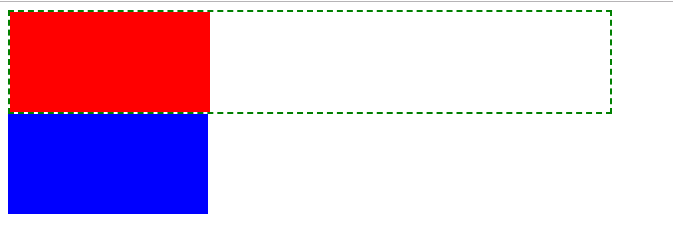
.clearfix {

zoom: 1;

/\*为了兼容IE\*/

}

清除浮动后的效果：



注：是在浮动元素的父级元素上设置伪元素，也就是说我们把 <class="box"> 改成 <class="box clearfix">，在其他地方使用的时候，也是只需要在标签中添加上一个 class 名 clearfix，并直接复制上述清除浮动的伪元素代码即可。当然我们这里还可以直接给父元素加上高度，来解决这个不利影响，不过一般不建议这么做，因为有时候我们也不知道高度具体是多少，容易出问题，所以还是建议用伪元素来清除浮动。

### 层模型

层模型有三种形式：相对定位、绝对定位、固定定位。

#### 层模型-绝对定位

绝对定位语法为：

position：absolute;

绝对定位有以下特点：

* 绝对定位以浏览器左上角为基准设置位置
* 当一个盒子包含在另一个盒子中，父盒子未设置定位，子盒子以浏览器左上角为基准设置位置； 当父盒子设置定位，子盒子以父盒子左上角为基准设置位置
* 绝对定位不占空间位置（类似于浮动）

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

div {

width: 200px;

height: 200px;

border: 5px red solid;

position: absolute;

left: 100px; /\*相对于浏览器向左偏移100像素\*/

top: 80px; /\*相对于浏览器向上偏移80像素\*/

}

</style>

</head>

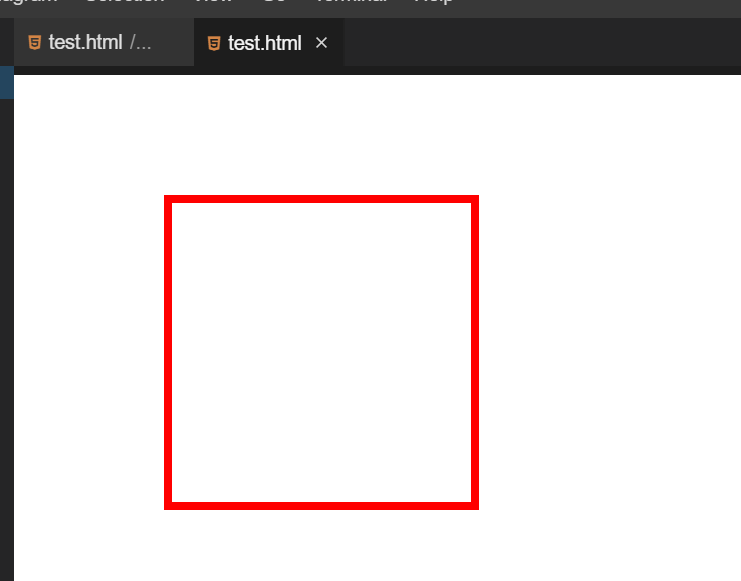
<body>

<div></div>

<!-- 相对于浏览器进行定位 -->

</body>

</html>



#### 相对定位

相对定位语法为：

position：relative;

相对定位有以下特点：

* 相对定位以元素自身的位置为基准设置位置
* 相对定位要占空间位置
* 一般子元素设置绝对定位，父元素设置相对定位（子绝父相）

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

.box1 {

width: 200px;

height: 100px;

position: relative;

border: 1px dashed green;

}

.box2 {

width: 100px;

height: 50px;

position: absolute;

border: 1px dashed blue;

top: 20px;

left: 20px;

}

</style>

</head>

<body>

<div class="box1">

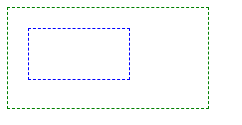
<div class="box2"></div>

</div>

</body>

</html>

效果图：



#### 固定定位

固定定位表现类似于绝对定位，只有一个主要区别：绝对定位固定元素是相对于 <html> 元素或其最近的定位祖先，而固定定位固定元素则是相对于浏览器视口本身。固定定位也不占空间位置，语法为：

position：fixed;

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style>

p {

position: fixed;

top: 200px;

left: 100px;

}

</style>

</head>

<body>

<p>实验楼</p>

</body>

</html>

注：复制这段代码到你的编辑器中运行，然后调整你浏览器的视口大小，你会发现，无论怎么变，p 标签所在的位置始终距离浏览器视口的左距离为 100px，上距离为 200px。

#### 介绍 top， bottom， left， right

使用 top， bottom， left 和 right 来精确指定要将定位元素移动到的位置，这些属性的值可以使用逻辑上期望的任何单位：px，cm，%等。也可以使用负值。

#### z-index

z-index 属性设置元素的堆叠顺序。拥有更高堆叠顺序的元素总是会处于堆叠顺序较低的元素的前面。用于处理当多个元素重叠的时候，哪些元素设置在其他元素的顶部。特别需要注意的是元素可拥有负的 z-index 属性值，属性值没有单位，只有数字。z-index 仅能在定位元素上奏效（例如 position:absolute;）给个例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title></title>

<style type="text/css">

.box {

position: absolute;

left: 50px;

top: 20px;

z-index: -1;

background-color: red;

width: 200px;

height: 200px;

}

.box1 {

position: absolute;

left: 80px;

top: 20px;

z-index: 2;

background-color: pink;

width: 200px;

height: 200px;

}

</style>

</head>

<body>

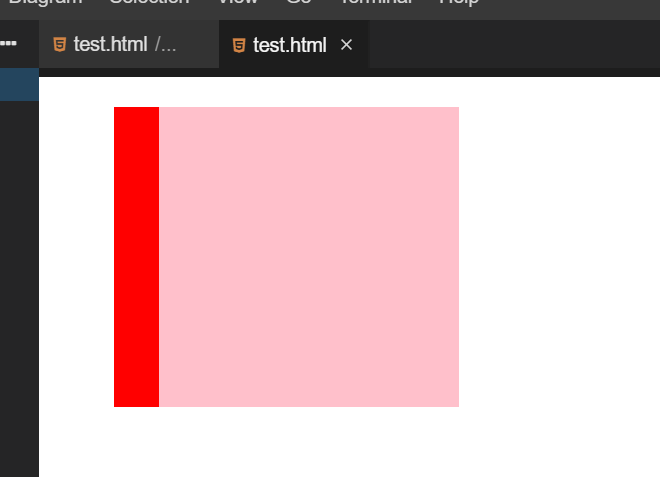
<div class="box"></div>

<div class="box1"></div>

</body>

</html>

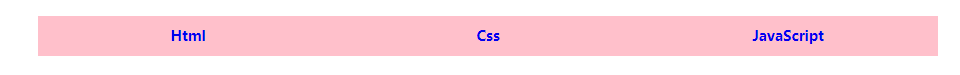
来看看效果：



### 挑战：导航

通过前面的学习，我们可以试着来做一下导航。

我们需要完成的最终效果：



思路：我们首先给一个外层的 div 盒子，让它在浏览器中水平居中，让它与浏览器视口上方有一段距离。然后在 div 盒子里面嵌套一个 ul 标签，ul 标签下嵌套三个 li 标签，每一个 li 标签下再写一个 a 标签，li 标签的宽度为 div 盒子的三分之一。文字内容相对于 li 标签是水平居中的，链接点了之后自己设置一个与默认颜色不同的颜色，字体样式为： 14px/40px 微软雅黑 加粗。尝试动手写一下吧。具体的宽度、颜色等可以自己设置成自己喜欢的样式。

源码：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style type="text/css">

body,

ul,

li {

margin: 0;

padding: 0;

}

ul,

li {

list-style: none;

}

.nav {

width: 900px;

height: 40px;

background: pink;

margin: 20px auto;

}

.nav ul li {

float: left;

width: 300px;

text-align: center;

}

.nav ul li a {

display: inline-block;

height: 40px;

font-family: 'yahei';

font-weight: bold;

padding: 0 20px;

line-height: 40px;

text-decoration: none;

}

.nav ul li a:hover {

background: #aaa;

}

</style>

</head>

<body>

<div class="nav">

<ul>

<li><a href="#">Html</a></li>

<li><a href="#">Css</a></li>

<li><a href="#">JavaScript</a></li>

</ul>

</div>

</body>

</html>

# CSS3

## 背景

在 CSS3 中，背景的样式作了一些升级调整，让我们可以更好的控制背景元素的样式。这里主要介绍以下几个调整：

* background-size
* background-origin
* background-clip

#### background-size

background-size 指定背景图像的大小，取代了以前背景图像的大小由实际大小决定。背景图像的大小可以是像素或是百分比大小。

background-size 的语法为：

background-size: length|percentage|cover|contain;



| **值** | **描述** |
| --- | --- |
| length | 设置背景图片高度和宽度。第一个值设置宽度，第二个值设置的高度。如果只给出一个值，第二个是设置为 **auto**(自动) |
| percentage | 以父元素的百分比来设置背景图像的宽度和高度，如 background-size:50% 50%;。第一个值设置宽度，第二个值设置高度。同 length 的用法，如果只给出一个值，那么第二个值会被设置为 auto。 |
| cover | 把背景图像扩展到可以完全覆盖背景区域为止，图片的某些部分可能会无法显示。 |
| contain | 使背景图片的高度和宽度完全适应内容区域。 |

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

border: 2px solid;

margin: 50px;

width: 200px;

height: 200px;

padding: 50px;

background-image: url('image.png');

background-size: 100% 100%;

/\* 如需要查看其它效果，可将注释中的内容替换上面的 background-size \*/

/\* background-size:100px 200px; \*/

/\* background-size:50% 50%; \*/

/\* background-size:cover; \*/

/\* background-size:contain; \*/

background-repeat: no-repeat;

}

</style>

</head>

<body>

<div></div>

</body>

</html>



在浏览器中运行结果图为：



请自行调整 background-size 的值，观察背景图片的变化。

#### background-origin

background-origin 的语法为：

padding-box|border-box|content-box;



background-origin 字面理解是背景图片起点的意思，它规定 background-position 属性相对于什么位置来定位。它有三个值：

| **值** | **描述** |
| --- | --- |
| padding-box | 默认）背景图片从盒模型的 padding-box 左上角开始绘制，即背景图片相对于内边距框来定位。 |
| border-box | 背景图片从盒模型的 border-box 左上角开始绘制，即背景图片相对于边框和来定位。 |
| content-box | 背景图片从盒模型的 content-box 左上角开始绘制，即背景图片相对于内容框来定位。 |

我们用上一个例子中的代码来演示 background-origin 不同值的效果，为了对比背景图片的不同的起始位置，我们使用图片的原尺寸，即删除 background-size ：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

/\* background-origin: border-box \*/

div {

border: 2px solid;

margin: 50px;

width: 200px;

height: 200px;

padding: 50px;

background-image: url('image.png');

background-repeat: no-repeat;

background-origin: border-box;

}

</style>

</head>

<body>

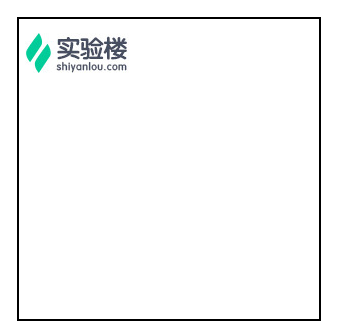
<div></div>

</body>

</html>



在浏览器中运行效果图为：



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

/\* background-origin: content-box \*/

div {

border: 2px solid;

margin: 50px;

width: 200px;

height: 200px;

padding: 50px;

background-image: url('image.png');

background-repeat: no-repeat;

background-origin: content-box;

}

</style>

</head>

<body>

<div></div>

</body>

</html>



在浏览器中运行效果图为：



我们很容易可以发现，background-origin 设置为 content-box 和 border-box 的区别，一个是从盒模型的 content-box 的左上角开始绘制，一个是从盒模型 border-box 左上角开始绘制。

**注意**：如果背景不是 no-repeat，这个属性无效，他会从边框开始显示。

#### background-clip

background-clip 的语法为：

background-clip: border-box|padding-box|content-box;



background-clip 与 background-origin 可以取到的值相同，它们的功能大致相同。但是两者存在一些细微的差别。当它们的值都为 border-box 时，虽然都是从盒模型的边框左上角开始绘制背景图片，但是 background-clip 绘制出来的背景图片不会覆盖左方和上方的边框，而 background-origin 绘制出来的背景图片则会覆盖左方和上方的边框。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

/\* background-origin: border-box \*/

div {

border: 2px solid;

margin: 50px;

width: 200px;

height: 200px;

padding: 50px;

background-image: url('image.png');

background-repeat: no-repeat;

background-clip: border-box;

}

</style>

</head>

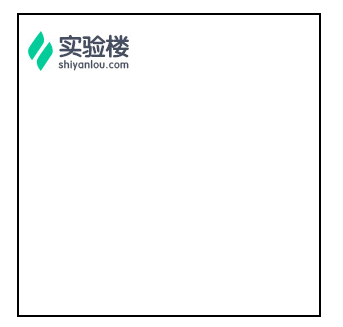
<body>

<div></div>

</body>

</html>

在浏览器中运行效果图为：



## 边框

### border-radius

在 CSS2 中添加边框圆角是一件很困难的事情，我们需要在每个角落使用不同的图像。但是，在 CSS3 中，我们只需要使用属性 border-radius 就可以创建圆角。

border-radius: 20px; /\*所有的角都使用半径为20px的圆角\*/



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

margin: 20px;

border: 2px solid;

border-radius: 20px;

width: 200px;

height: 200px;

}

</style>

</head>

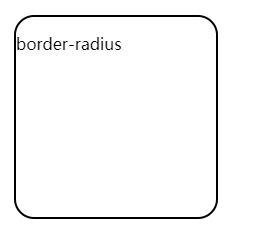
<body>

<div><p>border-radius</p></div>

</body>

</html>

在浏览器中运行效果图为：



border-radius 属性其实是 border-top-left-radius、border-top-right-radius、border-bottom-right-radius、border-bottom-left-radius 四个属性的简写模式，四个数值的书写顺序是：左上角、右上角、右下角、左下角。

* 当 border-radius 只设置一个值，即将该值应用到元素的四个角上。
* 当 border-radius 设置两个或三个值时，按照对角线相同原则，如：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

margin: 20px;

border: 2px solid;

border-radius: 20px 50px; /\*省略了左下和右下，按照对角线相同的原则，应为 20px 50px 20px 50px\*/

width: 200px;

height: 200px;

}

</style>

</head>

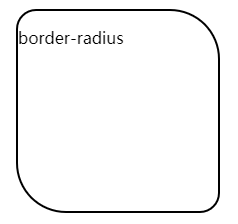
<body>

<div><p>border-radius</p></div>

</body>

</html>

在浏览器中运行效果图为：



* border-radius 按照语法的完整写法其实是 border-radius:10px 10px 10px 10px / 10px 10px 10px 10px; ，前面四个表示圆角的水平半径，后面四个表示圆角的垂直半径。

除了为边框添加圆角，你还可以将 border-radius属性运用到图片上来制作你的圆角图片。

**注意：**不要以为 border-radius 只可以用 px 单位，也可以用百分比或者 em，但目前的兼容性不太好。

### border-image

使用属性 border-image ，我们可以使用图片来创建一个边框。语法为：

border-image: source(url) slice width outset repeat;



* source:所使用图片的路径。
* slice:图片边框向内偏移。
* width:图片边框宽度。
* outset:边框图像区域超出边框的量。
* repeat:图像边框是否应该平铺（repeat）、铺满（round）或拉伸（stretch）。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

/\* background-origin: border-box \*/

div {

border: 40px solid;

margin: 50px;

width: 200px;

height: 200px;

padding: 50px;

border-image: url('image.png') 30 30 stretch;

}

</style>

</head>

<body>

<div></div>

</body>

</html>

在浏览器中运行效果图为：



### box-shadow

属性 box-shadow 用于给盒子添加阴影，语法与 text-shadow 有些相似：

box-shadow: 水平阴影 垂直阴影 模糊阴影 阴影尺寸 阴影颜色 inset;

与 text-shadow 相比，水平阴影、垂直阴影、模糊阴影的作用和 text-shadow 的对应属性相同，另外 box-shadow 多了阴影尺寸和 inset。

* 水平阴影：规定水平阴影的位置（阴影与盒子的水平偏移，正值向右偏移，负值向左偏移），必选项。
* 垂直阴影：规定垂直阴影的位置（阴影与盒子的垂直偏移，正值向下偏移，负值向上偏移），必选项。
* 模糊阴影：规定模糊的距离，可选项；值越大，模糊半径越大，阴影也就越大越淡。
* 阴影尺寸：阴影尺寸就是阴影外延出去的总长度。
* inset：阴影默认是向外发散的，inset 参数的作用就是使外阴影变为内阴影（可以写在参数的第一个和最后一个，其他位置是无效的）。
* 阴影颜色：规定阴影颜色，可选项；如不规定该项的值，则使用 UA（用户代理）自行选择的颜色。如果想保持跨浏览器的一致性，阴影的颜色就需要明确的被指定为一种。

注：参数的顺序不要乱写。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.shadow1 {

float: left;

margin: 50px;

width: 200px;

height: 200px;

background-color: #eee8cd;

box-shadow: 0 0 0 10px #000;

/\*将除阴影尺寸之外的数值都设置为0,观察阴影尺寸的效果\*/

}

.shadow2 {

float: left;

margin: 50px;

width: 200px;

height: 200px;

background-color: #eee8cd;

box-shadow: 20px 20px 10px #000;

/\*水平阴影和垂直阴影都为正值，阴影在原盒子右下方\*/

}

.shadow3 {

float: left;

margin: 50px;

width: 200px;

height: 200px;

background-color: #eee8cd;

box-shadow: -20px -20px 10px #000;

/\*水平阴影和垂直阴影都为负值，阴影在原盒子左上方\*/

}

.shadow4 {

float: left;

margin: 50px;

width: 200px;

height: 200px;

background-color: #eee8cd;

box-shadow: 0 0 100px #000 inset;

/\*除了模糊阴影外的值都设置为 0 ，另外使用参数 inset ，阴影在盒子内部\*/

}

.shadow5 {

float: left;

margin: 50px;

width: 200px;

height: 200px;

background-color: #eee8cd;

box-shadow: -40px 0 100px red, /\*左边阴影\*/ 0 -40px 100px #000,

/\*上边阴影\*/ 40px 0 100px blue, /\*右边阴影\*/ 0 40px 100px yellow; /\*下边阴影\*/

/\*盒子的多阴影，设置了上下左右四个阴影\*/

}

</style>

</head>

<body>

<div class="shadow1">

将除阴影尺寸之外的数值都设置为0，可以看到阴影尺寸就是阴影外延出去的长度，相当于增加了

border 的宽度。需要注意的是，box-shadow 不占据空间，所以阴影尺寸虽然增加了

border 的宽度，但它并不是 border。

</div>

<div class="shadow2">水平阴影和垂直阴影都为正值，阴影在原盒子右下方</div>

<div class="shadow3">水平阴影和垂直阴影都为负值，阴影在原盒子左上方</div>

<div class="shadow4">

除了模糊阴影外的值都设置为 0 ，另外使用参数 inset ，阴影在盒子内部

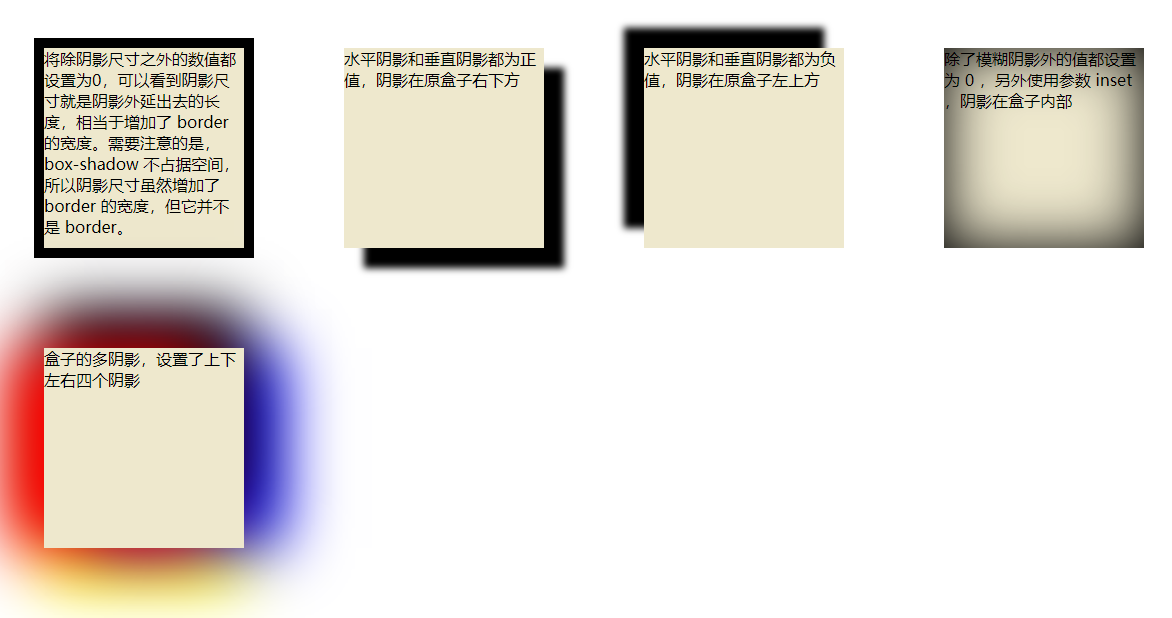
</div>

<div class="shadow5">盒子的多阴影，设置了上下左右四个阴影</div>

</body>

</html>

在浏览器中运行效果图为：



#### 模糊阴影和阴影尺寸的区别

模糊阴影：其值只能是为正值，如果其值为 0 时，表示阴影不具有模糊效果，其值越大阴影的边缘就越模糊。

阴影尺寸：其值可以是正负值，如果值为正，则整个阴影都延展扩大，反之值为负值时，则缩小；

## 颜色之 RGBA

RGB 是一种色彩标准，是由红(R)、绿(G)、蓝(B)的变化以及相互叠加来得到各式各样的颜色。RGBA 是在 RGB 的基础上增加了控制 alpha 透明度的参数。

语法：

color：rgba(R,G,B,A)



取值：

* R：红色值，正整数 | 百分数。
* G：绿色值，正整数 | 百分数。
* B：蓝色值，正整数 | 百分数。
* A：Alpha 透明度，取值 0~1 之间。

以上 R、G、B 三个参数，正整数值的取值范围为：0 - 255。百分数值的取值范围为：0.0% - 100.0%。超出范围的数值将被截至其最接近的取值极限。并非所有浏览器都支持使用百分数值。A 为透明度参数，取值在 0~1 之间，不可为负值。

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Document</title>

<style>

div {

width: 50px;

height: 50px;

background-color: rgb(255, 0, 0, 0.3);

}

</style>

</head>

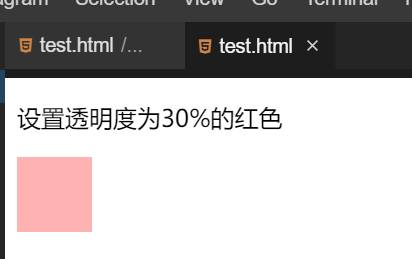
<body>

<p>设置透明度为30%的红色</p>

<div></div>

</body>

</html>



## text-shadow

属性 text-shadow 用于给文本添加阴影，语法为：

text-shadow: 水平阴影 垂直阴影 模糊阴影 阴影颜色;



* 水平阴影：规定水平阴影的位置（阴影与文本的水平偏移，正值向右偏移，负值向左偏移），必选项
* 垂直阴影：规定垂直阴影的位置（阴影与文本的垂直偏移，正值向下偏移，负值向上偏移），必选项
* 模糊阴影：规定模糊的距离，可选项;值越大，模糊半径越大，阴影也就越大越淡
* 阴影颜色：规定阴影颜色，可选项；如不规定该项的值，则使用 UA（用户代理）自行选择的颜色。如果想保持跨浏览器的一致性，阴影的颜色就需要明确的被指定为一种。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.shadow1 {

margin: 50px;

color: green;

text-shadow: 20px 20px 3px #000;

/\* 水平和垂直阴影都为正值，阴影呈现在原文本右下方 \*/

}

.shadow2 {

margin: 50px;

color: green;

text-shadow: -20px -20px 7px #000;

/\* 水平和垂直阴影都为负值，阴影呈现在原文本左上方 \*/

}

.shadow3 {

margin: 50px;

color: green;

text-shadow: 0 0 50px #000;

}

.shadow4 {

margin: 50px;

color: green;

/\*多重阴影，每个边的阴影用逗号隔开,阴影的应用顺序为从前到后，第一个指定的阴影在顶部\*/

text-shadow: -40px 0 10px red, /\*左边阴影\*/ 0 -40px 10px #000,

/\*上边阴影\*/ 40px 0 10px blue, /\*右边阴影\*/ 0 40px 10px yellow; /\*下边阴影\*/

}

</style>

</head>

<body>

<h1 class="shadow1">水平和垂直阴影都为正值，阴影呈现在原文本右下方</h1>

<h1 class="shadow2">水平和垂直阴影都为负值，阴影呈现在原文本左上方</h1>

<h1 class="shadow3">

没有设置水平与垂直阴影，阴影扩散的范围就是模糊半径的范围

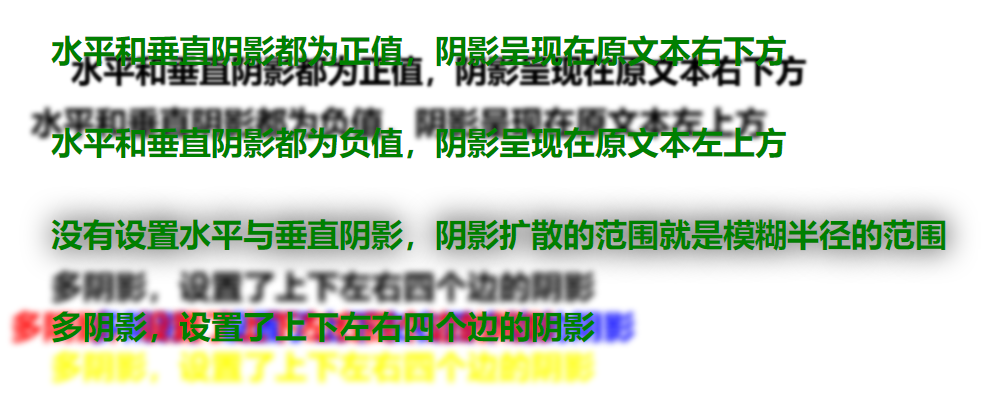
</h1>

<h1 class="shadow4">多阴影，设置了上下左右四个边的阴影</h1>

</body>

</html>

在浏览器中运行效果图为：



可以尝试修改参数值，体会数值正负对水平阴影和垂直阴影的影响,以及模糊阴影对模糊半径范围的影响。

## text-overflow

文本框经常会出现文本溢出的情况，在只有单行文本框，文本又出现溢出的情况，这时候就需要 text-overflow 来处理这些溢出的文本了。text-overflow 有三个取值：

* clip ：修剪文本。
* ellipsis ：显示省略符号来代替被修剪的文本。
* string ：使用给定的字符串来代替被修剪的文本。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.item1 {

white-space: nowrap;

border: 1px solid;

width: 150px;

overflow: hidden;

text-overflow: clip;

}

.item2 {

white-space: nowrap;

border: 1px solid;

width: 150px;

overflow: hidden;

text-overflow: ellipsis;

}

/\*

.item3{

white-space: nowrap;

border: 1px solid;

width: 150px;

overflow: hidden;

text-overflow: "i'm string";

}

字符串替代溢出字符的方式目前只有 Firefox 9.0+ 支持，大部分浏览器都不支持该溢出替换方式。

\*/

</style>

</head>

<body>

<div class="item1">hhhhhhhhhhhhhhhhhhhhhhhh</div>

<div class="item2">hhhhhhhhhhhhhhhhhhhhhhhh</div>

<!-- <div class="item3">hhhhhhhhhhhhhhhhhhhhhhhh</div> -->

</body>

</html>

在浏览器中运行效果图为：

此处输入图片的描述

可以发现，item1 中溢出的文本直接被截断，文本框边缘还出现了截断一半的 h 字母，因为 text-overflow 设置为 clip ；同样的，item2 中溢出的文本采用省略号代替，因为 text-overflow 设置为了 ellipsis。由于大部分浏览器都不支持溢出使用字符串替代的方式，且实验楼 Web IDE 提供的浏览器内核不是 Gecko ，所以我们将这种方式注释起来，感兴趣的同学可以在自己主机的 Firefox 浏览器中运行调试。

## overflow-wrap & word-break

overflow-wrap 和 word-break 都可以起到对单词的换行作用，前者用来表明是否允许浏览器在单词内进行短句，例如出现了一个很长的 url 或者单词，由于找不到自然段据点空格或标点符号而产生了溢出，这时候就可以使用 overflow-wrap 来解决这个问题；word-break 则是用于表明怎样在单词内进行短句。

#### overflow-wrap

overflow-wrap 常用的有两个取值：

* normal:浏览器保持默认处理
* break-word:允许在长单词内进行换行

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

margin: 50px;

border: 1px solid;

width: 100px;

overflow-wrap: break-word;

}

.div2 {

margin: 50px;

border: 1px solid;

width: 100px;

overflow-wrap: normal;

}

</style>

</head>

<body>

<div class="div1">Hello world!veryveryveryveryverylongword isn't it?</div>

<div class="div2">Hello world!veryveryveryveryverylongword isn't it?</div>

</body>

</html>



在浏览器中运行效果图为：



可以看到，句子中出现了一个很长的单词，属性 overflow-wrap 设置为 break-word，即允许从中间断开，没有出现溢出的情况，在句子开始的 Hello World！中，可以看到这两个单词是处于两行，也就是说如果把它们放在一行内会出现溢出的情况，break-word 不会拆散短单词，所以这里出现了换行；属性 overflow-wrap 设置为 normal 时，采用浏览器默认的换行方式，出现了溢出。

#### word-break

word-break 有三个取值：

* normal:保持浏览器默认换行方式
* break-all:表明对于 non-CJK（非中文/日文/韩文）文本，可在任意字符间断行
* keep-all:CJK 文本不断行，non-CJK 文本表现同 normal 。展开一点来说就是只能在半角空格或连字符处换行,也就是说，keep-all 只能在语句的正常语句起始处和单词起始处换行，如果 non-CJK 单词过长，使用值 keep-all 就有可能造成文本框溢出

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.item1 {

margin: 20px;

border: 1px solid;

width: 100px;

word-break: normal;

}

.item2 {

margin: 20px;

border: 1px solid;

width: 100px;

word-break: break-all;

}

.item3 {

margin: 20px;

border: 1px solid;

width: 100px;

word-break: keep-all;

}

</style>

</head>

<body>

<div class="item1">

我爱学习，我爱实验楼楼楼楼楼！Hello world!veryveryveryveryverylongword

isn't it?

</div>

<div class="item2">

我爱学习，我爱实验楼楼楼楼楼！Hello world!veryveryveryveryverylongword

isn't it?

</div>

<div class="item3">

我爱学习，我爱实验楼楼楼楼楼！Hello world!veryveryveryveryverylongword

isn't it?

</div>

</body>

</html>

在浏览器中运行效果图为：



第一个文本框的 word-break 属性设置为 normal，可以看到,浏览器默认换行是 CJK 换行，non-CJK 换行方式是在半角空格连字符的地方换行，长单词不换行。 第二个文本框的 word-break 属性设置为 break-all，可以看到 CJK 和 non-CJK 文本被随意拆分换行，不会出现文本框溢出，但是 non-CJK 文本也失去原有的样子； 第三个文本框的 word-break 属性设置为 keep-all，由于只能在半角空格或连字符处换行，所以过长的 non-CJK 文本产生了溢出现象，但是每个单词都能在一行内呈现，另外，设置为 keep-all 时，CJK 文本是不断行的，在第一个逗号的地方产生了换行，第二行的我爱实验楼楼楼楼楼中间没有半角空格或连字符，所以没有换行产生了溢出，直到感叹号的地方才产生换行。

## CSS3 动画和过渡

### 渐变

在 CSS3 之前，渐变效果需要图像来完成，这增大了下载时间和带宽的使用。 CSS3 的渐变是由浏览器生成的，所以其更具优势。CSS3 定义了两种类型的渐变：

* 线性渐变
* 径向渐变

渐变属于 image 类型，所以渐变只能在可以使用 image 的地方使用。

#### 线性渐变

渐变至少是一种颜色朝另一种颜色变化，可以让你在两个或多个指定的颜色之间显示平稳的过渡。所以必须至少定义两种颜色节点。同时，可以给出颜色变化的方向，线性渐变的语法如下：

background: linear-gradient(to direction | angle, color-stop1, color-stop2...);



**线性渐变 - 从上到下（默认情况下）**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>实验楼</title>

<style>

#box1 {

width: 300px;

height: 200px;

background-color: red; /\* 浏览器不支持时显示 \*/

background-image: linear-gradient(#e66465, #9198e5);

}

</style>

</head>

<body>

<p>从顶部开始的线性渐变。起点是红色，慢慢过渡到蓝色：</p>

<div id="box1"></div>

</body>

</html>





**线性渐变 - 从左到右**

下面的实例演示了从左边开始的线性渐变。起点是红色，慢慢过渡到蓝色：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>实验楼</title>

<style>

#box1 {

width: 300px;

height: 200px;

background-color: red; /\* 浏览器不支持时显示 \*/

background-image: linear-gradient(to right, #e66465, #9198e5);

}

</style>

</head>

<body>

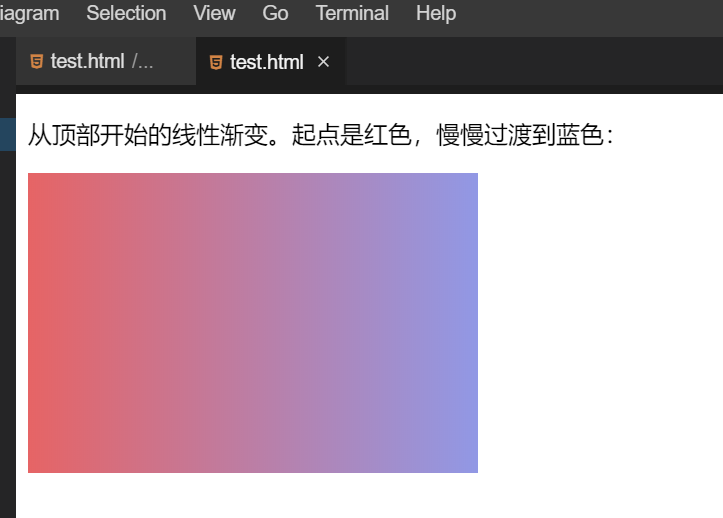
<p>从顶部开始的线性渐变。起点是红色，慢慢过渡到蓝色：</p>

<div id="box1"></div>

</body>

</html>





其实就是加上了 to right 这个值。

**线性渐变-对角**

你可以通过指定水平和垂直的起始位置来制作一个对角渐变。下面的实例演示了从左上角开始（到右下角）的线性渐变。起点是红色，慢慢过渡到黄色：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>实验楼</title>

<style>

#box1 {

width: 300px;

height: 200px;

background-color: red; /\* 浏览器不支持时显示 \*/

background-image: linear-gradient(to bottom right, red, yellow);

}

</style>

</head>

<body>

<p>从顶部开始的线性渐变。起点是红色，慢慢过渡到蓝色：</p>

<div id="box1"></div>

</body>

</html>





#### 径向渐变

径向渐变由它的中心定义。为了创建一个径向渐变，你也必须至少定义两种颜色结点。颜色结点即你想要呈现平稳过渡的颜色。同时，你也可以指定渐变的中心、形状（圆形或椭圆形）、大小。默认情况下，渐变的中心是 center（表示在中心点），渐变的形状是 ellipse（表示椭圆形），渐变的大小是 farthest-corner（表示到最远的角落）。

background-image: radial-gradient(

shape size at position,

start-color,

...,

last-color

);



**径向渐变 - 颜色结点均匀分布（默认情况下）**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>实验楼</title>

<style>

#box1 {

width: 300px;

height: 200px;

background-color: red; /\* 浏览器不支持的时候显示 \*/

background-image: radial-gradient(

red,

yellow,

blue

); /\* 标准的语法（必须放在最后）\*/

}

</style>

</head>

<body>

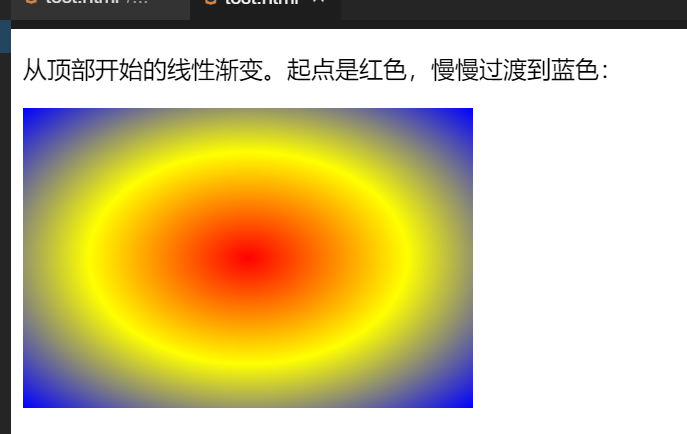
<p>从顶部开始的线性渐变。起点是红色，慢慢过渡到蓝色：</p>

<div id="box1"></div>

</body>

</html>





**径向渐变 - 颜色结点不均匀分布**

<style>

#box1 {

width: 300px;

height: 200px;

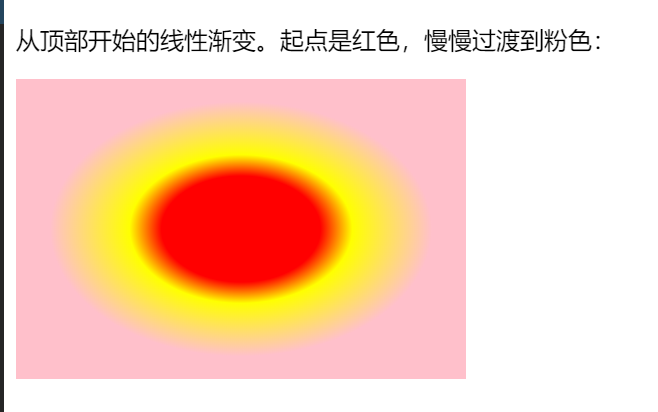
background-color: red; /\* 浏览器不支持的时候显示 \*/

background-image: radial-gradient(red 25%, yellow 35%, pink 60%);

}

</style>





**设置形状**

<style>

#box1 {

width: 300px;

height: 200px;

background-color: red; /\* 浏览器不支持的时候显示 \*/

background-image: radial-gradient(circle, red 25%, yellow 35%, pink 60%);

}

</style>



注意：由于不同的渲染引擎实现渐变的语法不同，这里只针对线性渐变的 W3C 标准语法来分析其用法，其余大家可以查阅相关资料。

### 过渡

在 CSS3 中，我们如果想要实现一种样式到另一种样式的转变，无需再使用 Flash 动画或者是 JavaScript ，CSS3 为我们提供了过渡的效果。

#### transition

属性 transition 应用于元素的过渡效果，使用时需要指定希望过渡的属性以及过渡持续的时间。

transition属性对应了四个值：

transition: transition | transition-duration | transition-timing-function |

transition-delay;



* transition-property 规定了应用于过渡效果的 CSS 属性的名称，默认值为 all 。
* transition-duration 规定了过渡持续的时间，默认值为 0s。也就是说，如果不指定过渡时间，那么将不会产生过渡效果。
* transition-timing-function 规定了过渡效果的时间曲线，如 ease （默认）：慢速开始，中间变快，慢速结束。liner ：匀速运动。
* transition-delay 规定了过渡效果的延迟时间，默认为 0s 。

transition-timing-function 有一些常用的值：

* linear:匀速，相当于（cubic-bezier(0,0,1,1)）。
* ease:慢速开始，然后变快，然后慢速结束，相当于（cubic-bezier(0.25,0.1,0.25,1)）。
* ease-in:慢速开始，相当于（cubic-bezier(0.42,0,1,1)）。
* ease-out:慢速结束，相当于（cubic-bezier(0,0,0.58,1)）。
* ease-in-out:慢速开始，慢速结束，相当于（cubic-bezier(0.42,0,0.58,1)）。
* cubic-bezier(n,n,n,n):在 cubic-bezier 函数中定义自己的值，取值范围是 0 ~ 1。

当指定多个属性的过渡效果时，用逗号隔开即可：

transition: width 2s ease-in-out, height 2s ease-in-out;



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

margin: 50px;

width: 200px;

height: 200px;

background-color: lightgreen;

opacity: 0.5;

transition: 2s ease-in-out;

/\*transition-property 默认为 all，所以将应用到 width 和 height 上\*/

}

.div1:hover {

width: 400px;

height: 400px;

opacity: 1;

}

.div2 {

margin: 50px;

width: 200px;

height: 200px;

background-color: cyan;

opacity: 0.5;

transition: height 2s cubic-bezier(0.25, 0.1, 0.25, 1);

/\*transition-property 只指定了 height，所以 width 的改变不会使用过渡效果;cubic-bezier(0.25,0.1,0.25,1) 相当于 ease，慢速开始，然后变快，慢速结束。\*/

}

.div2:hover {

width: 400px;

height: 400px;

opacity: 1;

}

.div3 {

margin: 50px;

width: 200px;

height: 200px;

background-color: orange;

opacity: 0.5;

transition: 2s ease-in-out 2s;

/\*transition-delay 设置为 2s，当鼠标移到该盒子上，过渡效果将会延迟 2s 开始、结束\*/

}

.div3:hover {

width: 400px;

height: 400px;

opacity: 1;

}

.div4 {

margin: 50px;

width: 200px;

height: 200px;

background-color: grey;

opacity: 0.5;

transition: ;

/\*所有属性使用默认值，transition-timing-function 的默认值为 0s，所以 div4 无过渡效果\*/

}

.div4:hover {

width: 400px;

height: 400px;

opacity: 1;

}

</style>

</head>

<body>

<div class="div1">div1 transition: 2s ease-in-out;</div>

<div class="div2">

div2 transition: height 2s cubic-bezier(0.25,0.1,0.25,1);

</div>

<div class="div3">div3 transition: 2s ease-in-out 2s;</div>

<div class="div4">div4 transition: ;</div>

</body>

</html>

将鼠标挪动到矩形上观察效果。

可以发现，div1 的持续时间设置为 2s，效果采用 ease-in-out，其余为默认值，所以 opacity、width、height 的改变都会采用过渡效果；div2 中指定了过渡的属性为 height，所以其它两个属性不会使用过渡效果；div3 设置了 transition-delay，过渡效果的开始和结束都会延迟相应的时间；div4 所有的值都使用默认值，包括持续时间 0s，所以 div4 无过渡效果，变化会非常生硬。

自行改变 transition 的值查看效果。

### 2D 、3D 转换

CSS3 转换可以对元素进行移动、缩放、转动、拉长或拉伸。在前面的章节中我们已经学习了 CSS3 的过渡效果，在本节中，我们将结合过渡功能对 CSS3 的 2D 、3D 转换进行学习。

* 2D 转换
* 3D 转换

转换有下列属性：

* transform: 将 2D 或 3D 转换应用到元素上去
* transform-origin：可以改变被转换元素的位置

transform-origin: x-axis y-axis z-axis;

/\*

x-axis:定义视图在 X 轴的位置，left | center | right | length | %

y-axis:定义视图在 Y 轴的位置，top | center | bottom | length | %

z-axis:定义视图在 Z 轴的位置，length

\*/



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

width: 300px;

height: 300px;

margin: 200px;

border: 2px solid;

text-align: center;

}

.div1 div {

width: 300px;

height: 300px;

background: lightgreen;

transform: rotate(45deg);

}

.transform-origin div {

transform-origin: 10% 10%;

}

</style>

</head>

<body>

<div class="div1">

<div>变换基点在默认位置</div>

</div>

<div class="div1 transform-origin">

<div>变换基点在 10% 10% 位置</div>

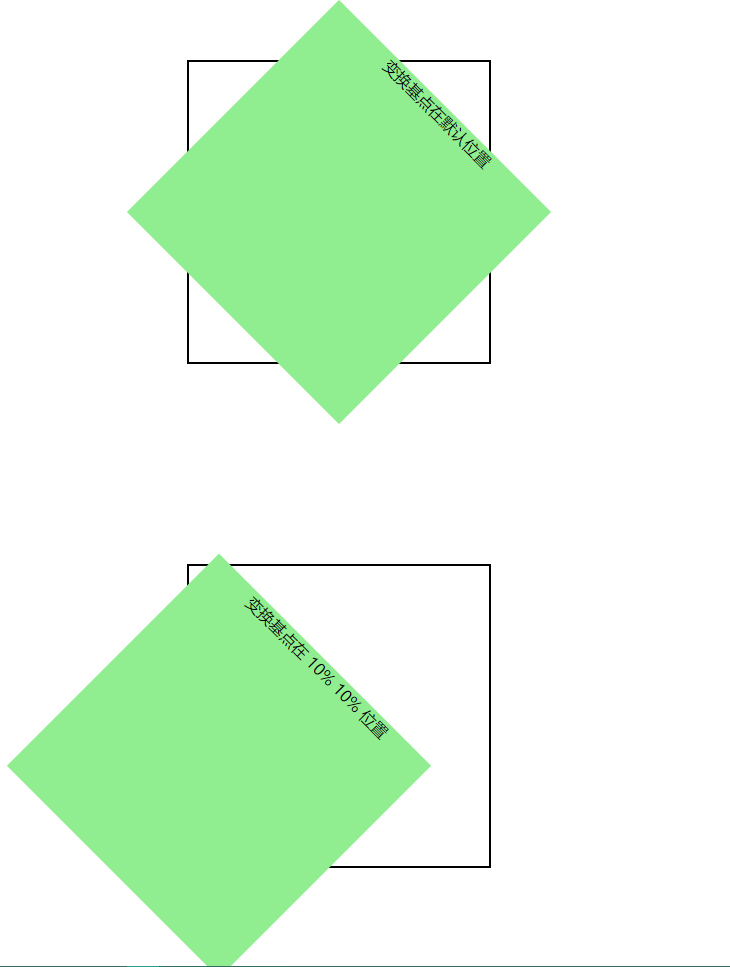
</div>

</body>

</html>



在浏览器中运行效果图为：



可以看到两个变换的基点有了明显的不同，第二个将变换基点设置到 10% 10% 的位置，变换也出现了不同。

* transform-style：规定被嵌套元素如何在 3D 空间中显示

transform-style: flat | preserved-3d;

/\*

flat：子元素将不保留 3D 位置

preserved-3d：子元素将保留 3D 位置

在实现一些 3D 效果的时候，一般都要声明为 preserved-3d

\*/



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

width: 200px;

height: 200px;

border: 1px solid #bbb;

margin: 100px;

/\* 在父元素中设置transform-style: preserve-3d; \*/

transform-style: preserve-3d;

transform: perspective(600px);

}

.div1 div {

width: 200px;

height: 200px;

background: lightgreen;

transform: rotateX(60deg);

}

.div2 {

width: 200px;

height: 200px;

border: 1px solid #bbb;

margin: 100px;

/\* 在父元素中设置transform-style: flat; \*/

transform-style: flat;

transform: perspective(600px);

}

.div2 div {

width: 200px;

height: 200px;

background: lightgreen;

transform: rotateX(60deg);

}

</style>

</head>

<body>

<div class="div1">

<div></div>

</div>

<div class="div2">

<div></div>

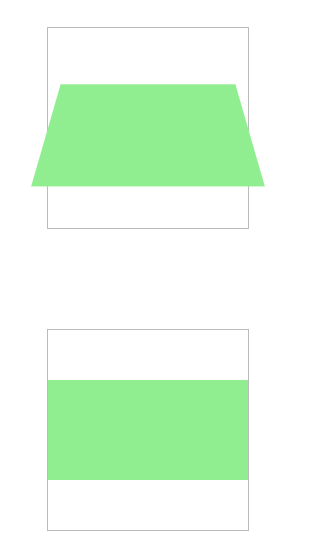
</div>

</body>

</html>



在浏览器中运行效果图为：



可以看到，div1 和 div2 除了 transform-style 属性之外，其它完全相同，一个保留了 3d 效果，一个未保留 3d 效果。

* perspective：规定 3D 元素的透视效果，与 perspective-origin 结合使用，可以改变 3D 元素的底部位置,在实现一些 3D 效果，这一项是必不可少的。

perspective: number | none;

/\*

number：单位 px，元素距离视图的距离

none：默认值，0，无透视

\*/



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

width: 300px;

height: 300px;

margin: 50px;

border: 1px solid #bbb;

}

.div1 div {

width: 100%;

height: 100%;

background: lightgreen;

transform: perspective(600px) rotateY(45deg);

/\*设置了透视和变换方式\*/

}

</style>

</head>

<body>

<div class="div1">

<div></div>

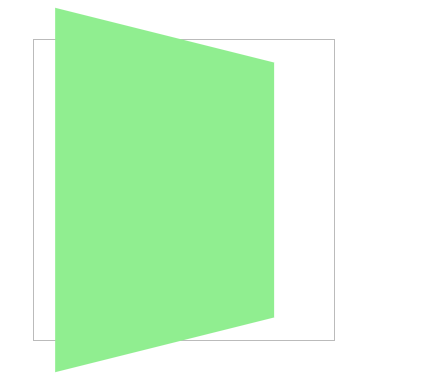
</div>

</body>

</html>



在浏览器中运行效果图为：



* perspective-origin：规定 3D 元素的底部位置

perspective-origin: x-axis y-axis;

/\*

x-axis:定义该视图在 X 轴上的位置。默认为 50%。left|center|right|length|%

y-axis:定义该视图在 Y 轴上的位置。默认为 50%。top|center|bottom|length|%

将透视聚集点改变到其他位置。

\*/



* backface-visibility：定义元素在不面对屏幕时是否可见

backface-visibility: visible|hidden;

/\*

visible：背面可见

hidden：背面不可见

\*/



<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

width: 200px;

height: 200px;

background: lightgreen;

transform: rotateY(180deg);

/\* 将 div 元素都翻转 180deg，翻转到背面 \*/

}

.div1 {

backface-visibility: hidden;

}

.div2 {

backface-visibility: visible;

}

</style>

</head>

<body>

<div class="div1">div1 backface-visibility: hidden;</div>

<div class="div2">div2 backface-visibility: visible;</div>

</body>

</html>



在浏览器中运行效果图为：



可以看到，两个 div 块在翻转后都是背面朝向我们，当我们设置 div1 的 backface-visibility: hidden 时，div1 就不可见了，只有 div2 是可见的。

#### 2D 转换

2D 转换有以下几种常用方法：

* translate(x,y) ：根据原始位置的 x 、y 轴位置，分别偏离 x 、y 的距离。
* rotate(angle) ：根据给定的 angle 大小，顺时针旋转，如果 angle 为负，则逆时针旋转。
* scale(x,y) ：根据 x 、y 的值，对应的放大宽度和高度。
* skew(angle,angle) ：根据给定的两个角度，分别在水平方向和垂直方向倾斜，第二个角度默认为 0 。
* matrix() ：matrix 变换涉及到了线性代数的知识，在此不作讲述。感兴趣的同学可以自行了解。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

margin: 50px;

width: 200px;

height: 200px;

background-color: lightgreen;

opacity: 0.5;

transition: 2s;

}

.div1:hover {

transform: translate(30px, 40px);

/\*translate(x,y)\*/

opacity: 1;

}

.div2 {

margin: 50px;

width: 200px;

height: 200px;

background-color: cyan;

opacity: 0.5;

transition: 2s;

}

.div2:hover {

transform: rotate(180deg);

/\*rotate(angle)\*/

opacity: 1;

}

.div3 {

margin: 50px;

width: 200px;

height: 200px;

background-color: orange;

opacity: 0.5;

transition: 2s;

}

.div3:hover {

transform: skew(20deg, 30deg);

/\*skew(angle,angle)\*/

opacity: 1;

}

</style>

</head>

<body>

<div class="div1">div1</div>

<div class="div2">div2</div>

<div class="div3">div3</div>

</body>

</html>



把鼠标移动到矩形上，观察效果。

#### 3D 转换

3D 转换语法与 2D 转换语法较为类似

* translate3d(x,y,z):定义 3D 转化，给出三个值，分别应用于三个坐标轴上
* translateX(x):定义 3D 转化，仅用于 X 轴
* translateY(y):定义 3D 转化，仅用于 Y 轴
* translateZ(z):定义 3D 转化，仅用于 Z 轴
* scale3d(x,y,z):定义 3D 缩放转换
* scaleX(x):定义 3D 缩放转换，仅用于 X 轴
* scaleY(y):定义 3D 缩放转换，仅用于 Y 轴
* scaleZ(z):定义 3D 缩放转换，仅用于 Z 轴
* rotate3d(x,y,z,angle):定义 3D 旋转
* rotateX(angle):定义沿 X 轴的 3D 旋转
* rotateY(angle):定义沿 Y 轴的 3D 旋转
* rotateZ(angle):定义沿 Z 轴的 3D 旋转
* perspective:定义 3D 转换元素的透视视图

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

margin: 50px;

width: 200px;

height: 200px;

background-color: lightgreen;

opacity: 0.5;

transition: 2s;

}

.div1:hover {

/\*X 轴旋转\*/

transform: rotateX(180deg);

opacity: 1;

}

.div2 {

margin: 50px;

width: 200px;

height: 200px;

background-color: cyan;

opacity: 0.5;

transition: 2s;

}

.div2:hover {

/\*Y 轴伸缩\*/

transform: scaleY(2);

opacity: 1;

}

.div3 {

margin: 50px;

width: 200px;

height: 200px;

background-color: orange;

opacity: 0.5;

transition: 2s;

}

.div3:hover {

/\*X 轴移动\*/

transform: translateX(50%);

opacity: 1;

}

</style>

</head>

<body>

<div class="div1">div1</div>

<div class="div2">div2</div>

<div class="div3">div3</div>

</body>

</html>

把鼠标移动到盒子上，观察效果。

### 动画

动画是使元素从一种样式逐渐变化为另一种样式的效果。在 CSS3 中 ，我们可以创建动画，取代以往的网页动画图象。

#### @keyframes

在 CSS3 标准下，我们使用 @keyframes 规则来创建动画。我们无需将动画的每一帧都描绘出来，只需要给出变化的关键帧，然后给出变化时间，其余的等待它自动计算完成就可以了。 @keyframes 里描述了动画的关键帧的相关属性，语法如下：

@keyframes 动画名称 {

0% {

/\*css code 描述关键帧属性\*/

}

25% {

/\*css code 描述关键帧属性\*/

}

50% {

/\*css code 描述关键帧属性\*/

}

75% {

/\*css code 描述关键帧属性\*/

}

100% {

/\*css code 描述关键帧属性\*/

}

}

动画创建完成后，我们使用 animation 属性将动画绑定到对应的元素上去，并给出运行时间，就完成了一个简单的动画。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

@keyframes myfirstcartoon {

0% {

background-color: lightgreen;

}

25% {

background-color: blue;

}

50% {

background-color: orange;

}

75% {

background-color: red;

}

100% {

background-color: yellow;

}

}

body {

animation: myfirstcartoon 5s;

}

</style>

</head>

<body></body>

</html>

在 Preview 或 Mini Browser 中打开，观察网页的变化。

CSS3 动画还有很多有趣的属性：

* animation:动画属性的简写属性，除了 animation-play-state 属性。
* animaiton-name：规定 @keyframes 动画的名称。
* animation-duration：规定动画一个周期的时间，默认为 0。
* animaiton-timing-function：规定动画的速度曲线。默认为 ease。
* animaiton-delay：规定动画的播放延迟时间。
* animaiton-iteration-count：规定动画播放次数，默认为 1。取值为 infinite 时，动画被无限次播放。
* animation-direction：规定动画是否在下一周期逆向播放，默认为 normal。取值为 normal 时，动画正常播放，取值为 alternate 时，动画轮流反向播放。
* animation-play-state：规定动画是否正在运行或暂停，默认为 running；暂停时为 paused。
* animation-fill-mode：规定动画在播放之前或之后，其动画效果是否可见。取值为 none 时，不改变默认行为；取值为 forwards 时，在动画完成后，保持最后一个属性；取值为 backwards 时，在动画开始之前应用开始属性值；取值为 both 时，向前向后的模式都将被启用。

我们可以使用 animaiton 来简写动画的这些属性：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

@keyframes myfirstcartoon {

0% {

background-color: lightgreen;

}

25% {

background-color: blue;

}

50% {

background-color: orange;

}

75% {

background-color: red;

}

100% {

background-color: yellow;

}

}

body {

animation: myfirstcartoon 5s ease-in-out 1s infinite alternate;

/\*上述值分别对应了：name duration timing-function delay iteration-count direction\*/

}

</style>

</head>

<body></body>

</html>

在 Preview 或 Mini Browser 中打开，观察网页的变化。

注：创建好的动画需要使用 animation 绑定到一个选择器上，否则动画不会运行；并且，需要给出运行时间，因为时间默认值是 0s ，如果不给出时间，动画将同样不会运行。

## CSS3布局

### Flex 布局

Flex 是 Flexible Box（弹性盒子）的缩写，意为“弹性布局”，用来为盒状模型提供最大的灵活性。任何一个容器都可以指定为 Flex 布局。

div {

display: flex;

}

行内元素也可以使用 Flex 布局。

span {

display: inline-flex;

}

Webkit 内核的浏览器，必须加上 -webkit 前缀。

div {

display: -webkit-flex; /\* Safari \*/

display: flex;

}

弹性布局是一种当页面需要适应不同的屏幕大小以及设备类型时确保元素拥有恰当的行为的布局方式。

引入弹性盒布局模型的目的是提供一种更加有效的方式来对一个容器中的子元素进行排列、对齐和分配空白空间。

#### 基本概念

采用 Flex 布局的元素，称为 Flex 容器（flex container），简称“容器”。它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称“项目”。



容器默认存在两根轴：水平的主轴和垂直的交叉轴。

#### 容器属性

* flex-direction
* flex-wrap
* flex-flow
* justify-content
* align-items
* align-content

#### 子元素属性

* flex
* align-self

#### flex-direction

flex-direction 属性指定了主轴线的方向，子元素默认是按照主轴线排列的，所以 flex-direction 也指定了弹性子元素在弹性容器中的排列方式， 有以下四个值：

* row ：（默认）横向从左到右排列。
* row-reverse ：顾名思义，从右向左横向排列，反向。
* column ： 纵向从上往下排列。
* column-reverse ： 纵向从下往上排列。

下面使用值 row 操作属性 flex-direction：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

flex-direction: row;

width: 400px;

height: 450px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 20px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

<div class="item">item3</div>

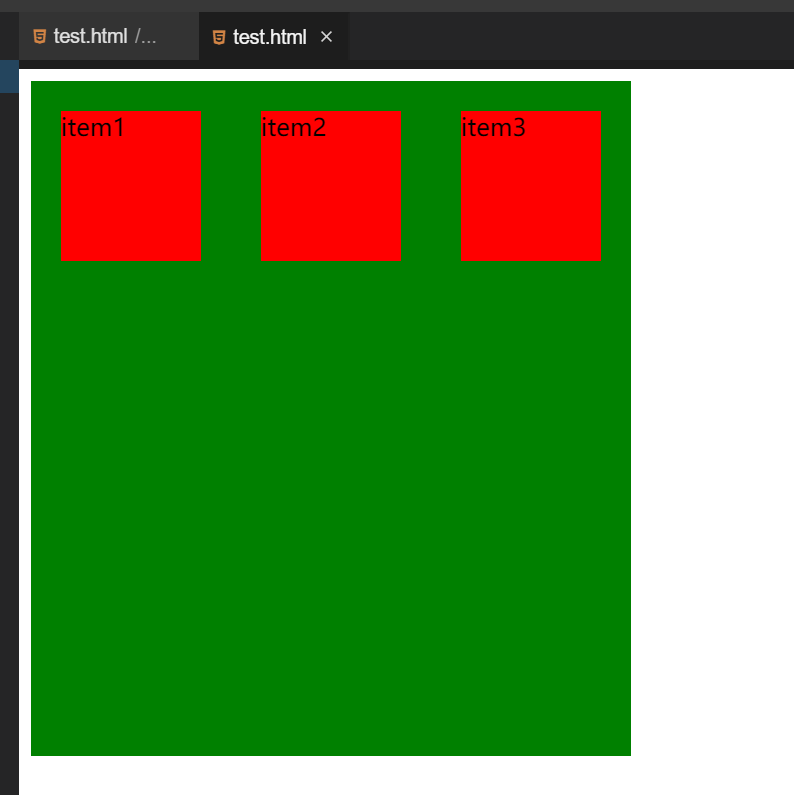
</div>

</body>

</html>

****

在浏览器中运行的效果图为：



如图所示，三个子元素按照值 row 的排列方式，在容器的横向从左到右排列。

当我们修改 flex-direction 为 column ， 在容器的纵向从上往下排列。

#### flex-wrap

弹性盒子默认情况下所有子元素都是排在一行上的，但容器大小有限，很有可能出现子元素被截断的情况，这时候就需要换行，flex-wrap 就是用来操作子元素换行的属性。flex-wrap 有以下三个值：

* nowrap ：（默认） 不换行。
* wrap ：换行，第一行排满后自动换到第二行。
* wrap-reverse ：换行，与 wrap 不同的是如果出现换行，换行后的元素在第一行上方。

下面使用值 wrap 来操作属性 flex-wrap ：

<style>

.container {

display: flex;

flex-wrap: wrap;

width: 350px;

height: 450px;

background-color: green;

}

.item {

background-color: red;

width: 120px;

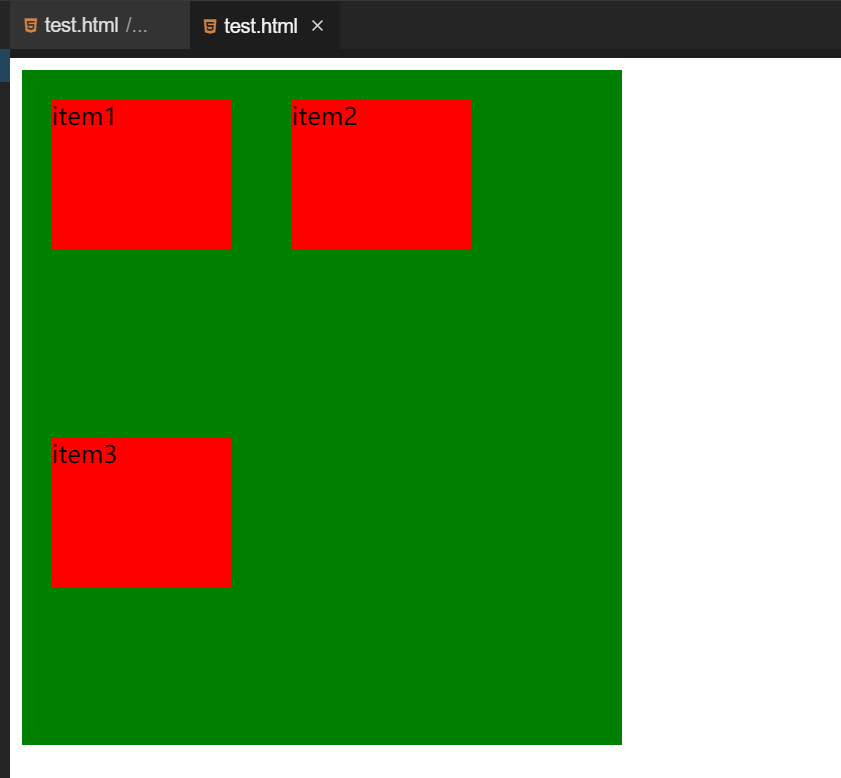
height: 100px;

margin: 20px;

}

</style>

在浏览器中运行效果图为：



当我们的父元素（flex-container）宽度为 35px 时，子元素（flex item）为 120px，父元素的一排排不满三个盒子，那么设置 wrap 属性就能够换行，第一行排满后自动换到第二行。

而如果我们强制不换行，设置 nowrap 属性值：



子元素的宽度就会被自动减少，挤在一行。

#### flex-flow

flex-flow 是 flex-direction 和 flex-wrap 的简写形式，默认值为 row nowrap。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

flex-flow: row wrap;

/\*从左往右 换行\*/

width: 350px;

height: 350px;

background-color: green;

}

.item {

background-color: red;

width: 120px;

height: 100px;

margin: 20px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

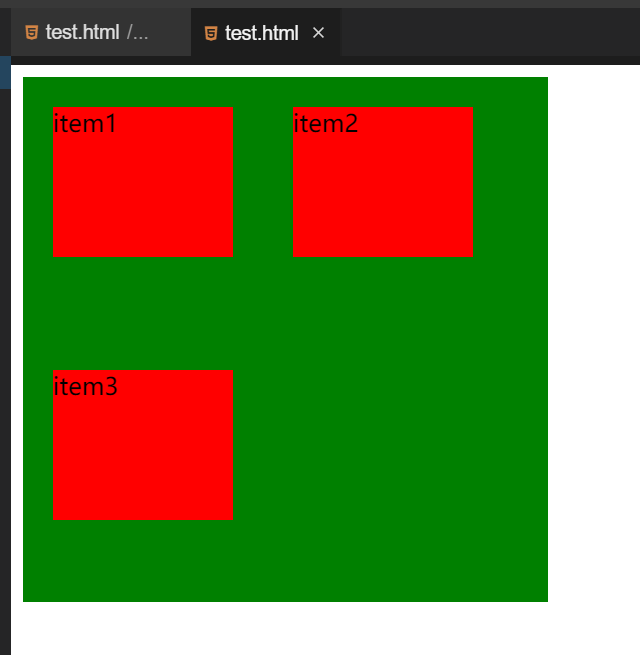
<div class="item">item3</div>

</div>

</body>

</html>

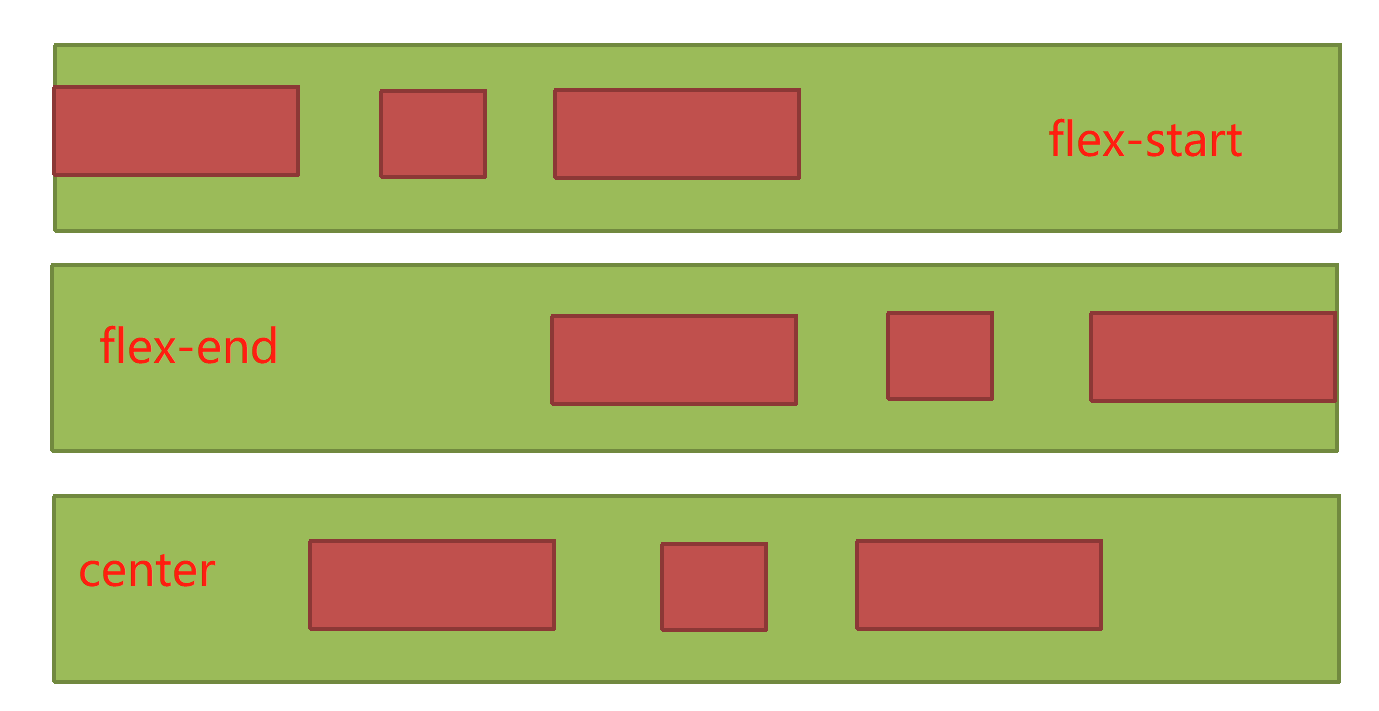
在浏览器中运行效果图为：

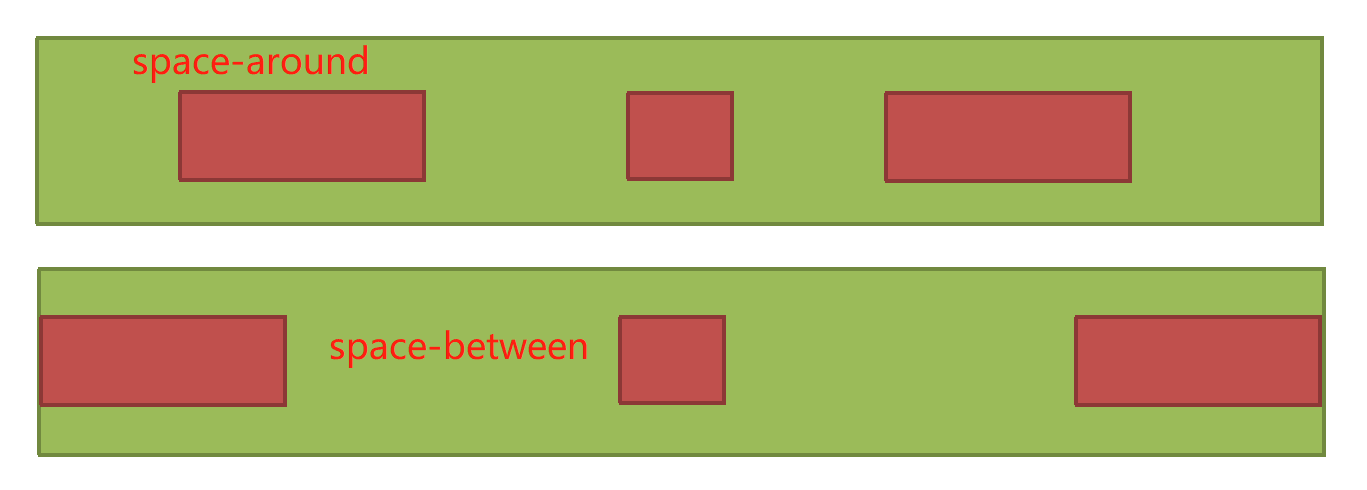


#### justify-content

子元素在容器中默认是以左对齐的布局排列在主轴线上的，属性 justify-content 用来调整子元素在主轴线上的对齐方式，它有以下五个值：

* flex-start ：左对齐
* flex-end ：右对齐
* center ：居中
* space-between ：两端对齐，项目之间的间隔都相等。
* space-around ：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。





下面使用值 space-between 来操作属性 justify-content ：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

justify-content: space-around;

width: 400px;

height: 250px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

<div class="item">item3</div>

</div>

</body>

</html>

在浏览器中运行效果图为：

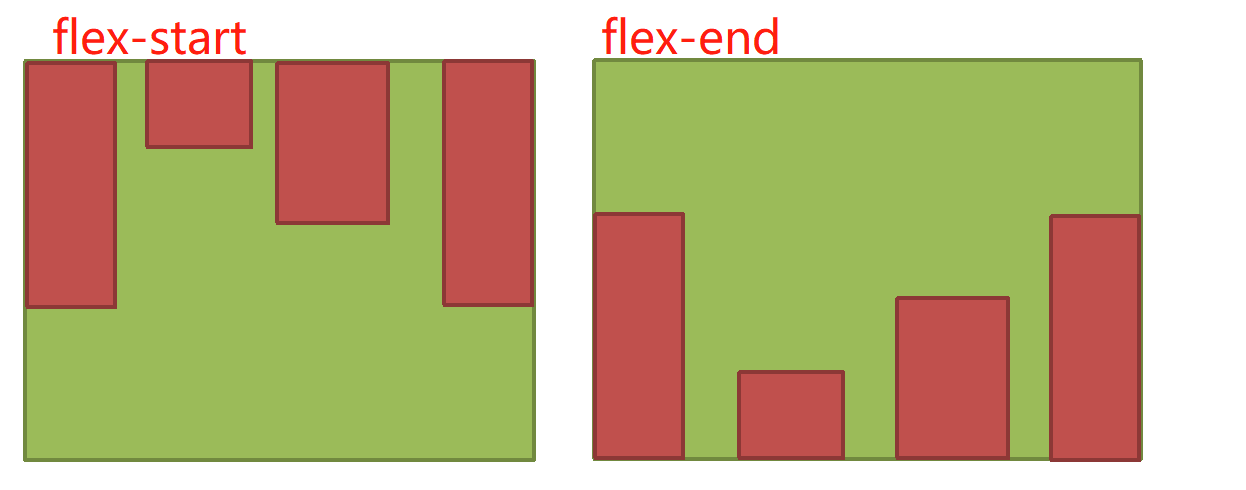


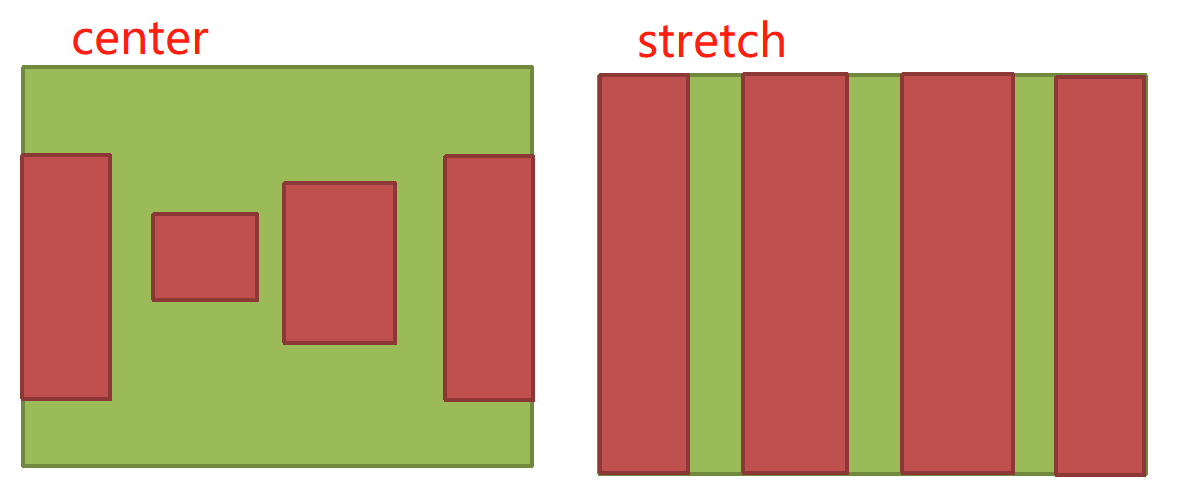
容器的宽度为 400px，三个子元素的宽度和加上它们的边框小于 400px，所以不会出现换行。

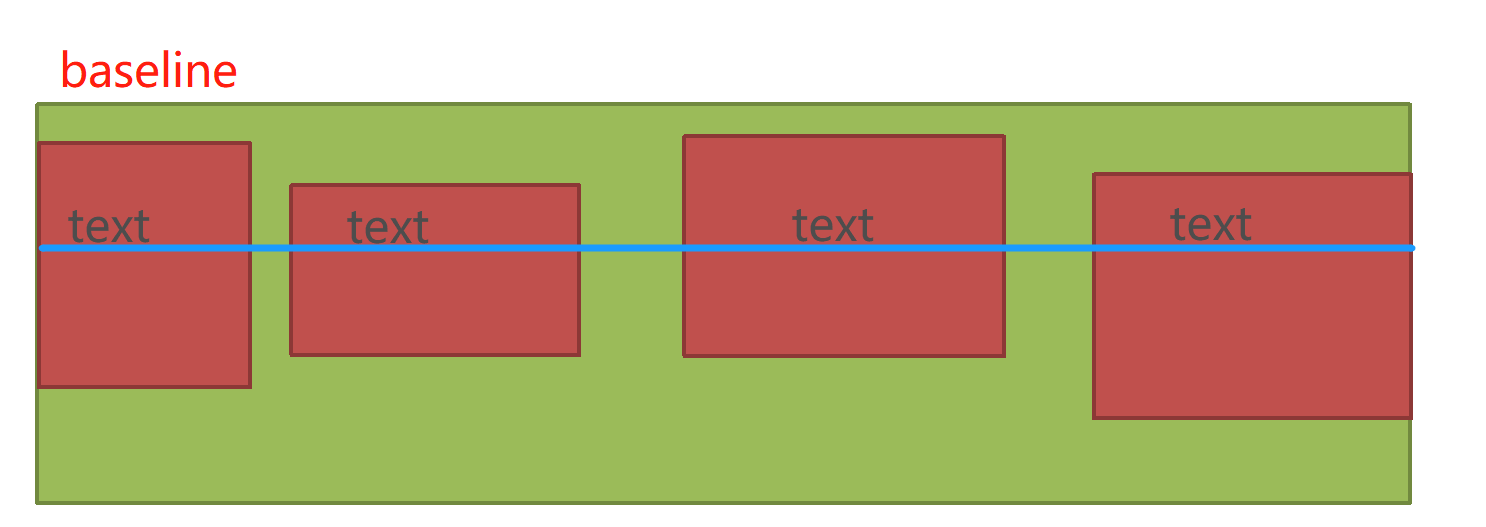
#### align-items

与主轴垂直的轴我们称它为交叉轴，属性 align-items 用于定义子元素在交叉轴上的排列方式，它有下列五个值：

* flex-start ：交叉轴的起点对齐。
* flex-end ：交叉轴的终点对齐。
* center ：交叉轴的中点对齐。
* baseline ：项目的第一行文字的基线对齐。
* stretch ：如果子元素未设置高度或设置为 auto ，子元素将占满整个容器高度。







下面使用值 flex-end 来操作属性 align-items ：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

flex-direction: column;

align-items: flex-end;

width: 400px;

height: 250px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

<div class="item">item3</div>

</div>

</body>

</html>

在浏览器中运行效果图为：



注：这里的 flex-direction 设置为 column ，即主轴线的方向是垂直的，即交叉轴的方向是水平的；align-items 设置为 flex-end，即子元素在交叉轴结束位置上的边界对齐，交叉轴的结束位置在右侧，所以三个子元素在右边界上对齐。

#### align-content

align-content属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。，align-content 有以下六个值：

* flex-start ：与交叉轴的起点对齐。
* flex-end ：与交叉轴的终点对齐。
* center ：与交叉轴的中点对齐。
* space-between ：与交叉轴两端对齐，轴线之间的间隔平均分布。
* space-around ：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。
* stretch：轴线占满整个交叉轴。

下面使用值 center 来操作属性 align-content ：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

flex-wrap: wrap;

align-content: center;

width: 200px;

height: 600px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

<div class="item">item3</div>

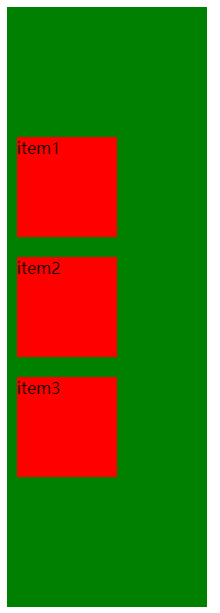
</div>

</body>

</html>

****

在浏览器中运行效果图为：



注：该处设置 flex-wrap 换行属性为 wrap，允许换行。align-content 属性在容器内不存在换行时不产生效果，所以我们将容器的 width 设置为 200px，一行只能放下一个子元素。align-content：center 表示每一行在交叉轴上居中排列，三个子元素只是在交叉轴上居中排列，因为我们没有设置主轴线的对齐方式，如果我们将 CSS 代码修改为如下所示，其余代码不变，那么子元素的排列效果就会发生变化。

.container {

display: flex;

flex-wrap: wrap;

align-content: center;

justify-content: center;

width: 200px;

height: 600px;

background-color: green;

}

.item {

background-color: red;

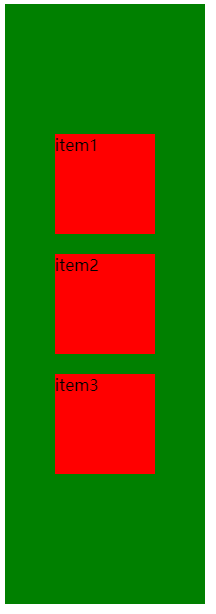
width: 100px;

height: 100px;

margin: 10px;

}

在浏览器中运行效果图为：



可以看到，三个子元素的排列已经完全居中了

#### Flex

flex 属性实际上是三个属性的简写形式：

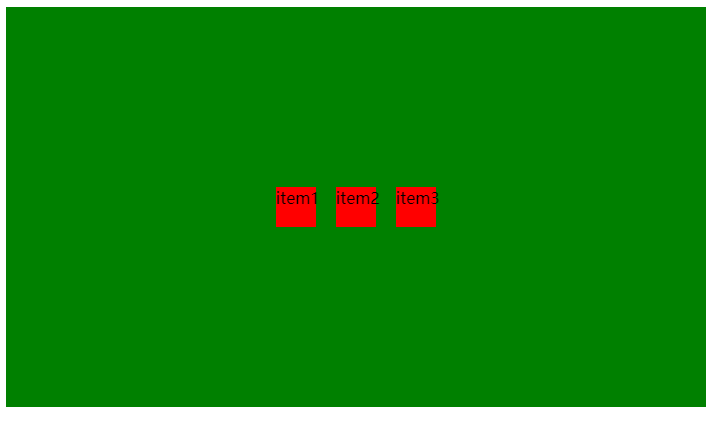
flex: flex-grow || flex-shrink || flex-basis;

****

所以在介绍 flex 之前，我们先来介绍这三个属性各自的特性。

#### flex-grow

当容器比子元素大很多，子元素的宽高又设置的很小，这个时候即使水平垂直居中，整个布局看起来仍然很奇怪：



又或者根据实际的需要，子元素需要将容器占满。这个时候，我们就可以使用属性 flex-grow 来操作。 flex-grow 是设置子元素如何分配容器的剩余空间的属性。

假设容器的宽度为 400px，A 元素的宽度为 100px，B 元素的宽度也为 100px。

* 当 flex-grow 的值为 0 （默认），子元素不会分配剩余空间，所以 A 和 B 的宽度仍然是 100px 。
* 当 flex-grow 的值为 1 ，子元素平均分配剩余空间，即：剩余宽度为 200px ，A 和 B 各分配得到 100px ，A 和 B 的宽度都变为了 200px 。
* 当 flex-grow 的值大于 1，则按照各自所占的比例分配剩余空间。例如 A 元素的 flex-grow = 1 ，B 元素的 flex-grow = 2 ，则 A 分配到 1/3 的剩余空间，即 1/3 \\* 200px ，B 元素同理。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

justify-content: center;

width: 600px;

height: 200px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

.item1 {

flex-grow: 3;

}

.item2 {

flex-grow: 6;

}

.item3 {

flex-grow: 1;

}

</style>

</head>

<body>

<div class="container">

<div class="item item1">item1</div>

<div class="item item2">item2</div>

<div class="item item3">item3</div>

</div>

</body>

</html>

****

在浏览器中运行效果图为：



三个子元素按照比例分配了剩余的空间，形成了上图的布局。

#### flex-shrink

与 flex-grow 相反，当容器过小，子元素过大时，属性 flex-shrink 用来缩小子元素。 假设容器的宽度为 300px ，A 元素的宽度为 200px ，B 元素的宽度也为 200px 。

* 当 flex-shrink 的值为 0 ，子元素不会缩小，可能会导致元素溢出容器。
* 当 flex-shrink 的值为 1 （默认），子元素会按照同样的比例缩小自身的大小。A 的宽度 = （200-100\*（200\*1/（200\*1+200\*1）））= 150 。B 元素也同理。
* 当 flex-shrink 的大于 1 ，子元素会按照给定的比例缩小自身的大小。例如 A 的 flex-shrink 设置为 2 ，B 的 flex-shrink 设置为 3 ，则 A 的宽度 = （200-100\*（200\*2/（200\*2+200\*3）））= 160 。B 元素也同理。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

justify-content: center;

width: 300px;

height: 200px;

background-color: green;

}

.item {

background-color: red;

width: 200px;

height: 100px;

margin: 10px;

}

.item1 {

flex-shrink: 3;

}

.item2 {

flex-shrink: 6;

}

.item3 {

flex-shrink: 1;

}

</style>

</head>

<body>

<div class="container">

<div class="item item1">item1</div>

<div class="item item2">item2</div>

<div class="item item3">item3</div>

</div>

</body>

</html>

****

在浏览器中运行效果图为：



#### flex-basis

flex-basis 用于设置子元素的宽度，规定子元素的基准值，如果子元素另外设置了 width ，会被该属性的值覆盖。 flex-basis 有几个特殊的取值，我们介绍其中的 auto 取值（默认）。当 flex-basis 取值为 auto 时，如果子元素的宽度不为 auto ，那么子元素的宽度就设置为自身的宽度；如果子元素的宽度设置为 auto ，那么子元素的宽度就根据子元素的内容来自动布局。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

justify-content: center;

width: 400px;

height: 200px;

background-color: green;

}

.item {

background-color: red;

flex-basis: auto;

width: auto;

height: 100px;

margin: 10px;

}

</style>

</head>

<body>

<div class="container">

<div class="item">item1</div>

<div class="item">item2</div>

<div class="item">item3</div>

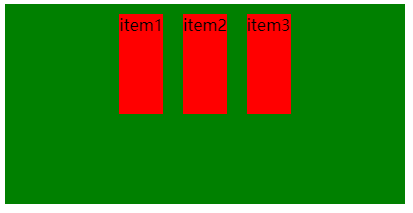
</div>

</body>

</html>

****

在浏览器中运行效果图为：



我们将子元素的宽度设置为 auto ，flex-basis 也设置为 auto ，布局就会根据子元素的内容进行调整适配。

flex 是以上三个属性值的组合，假设以上三个属性都取默认值，则 flex 的值为 0 1 auto。flex 有以下几个常用值：

* auto ：1 1 auto
* none ：0 0 auto
* inherit ：从父元素继承

当 flex 取一个非负数字，则将其视为 flex-grow 的值，flex-shrink 取默认值，flex-basis 取 0%。 当 flex 取一个长度或百分比，则将其视为 flex-basis 的值，flex-grow 和 flex-shrink 都取 1 。 当 flex 取两个非负数字，则分别将其视为 flex-grow和 flex-shrink 的值，flex-basis 取 0%。

下面我们来操作 flex 属性：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

flex-wrap: wrap;

align-content: center;

width: 400px;

height: 250px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

.item1 {

flex: 3;

}

.item2 {

flex: 1;

}

.item3 {

flex: 1;

}

</style>

</head>

<body>

<div class="container">

<div class="item item1">item1</div>

<div class="item item2">item2</div>

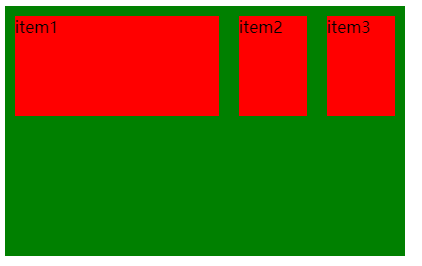
<div class="item item3">item3</div>

</div>

</body>

</html>

在浏览器中运行效果图为：



在上面代码中，我们将 item1 的 flex 设置为 3，其余两个子元素的 flex 设置为 1，运行后我们发现，item1 分到了剩余空间的 3/5 ，其余两个子元素各自分到了 1/5 。因为 flex 取一个非负数字的时候，flex-basis 为 0%，即子元素的宽度被覆盖为 0 ，此时分配剩余空间也可以理解为将容器按比例分配给各个子元素， item1 占容器 3/5 的大小，其余两个子元素各占 1/5 。

#### align-self

属性 align-self 可以使某个子元素的对齐方式与其它的子元素不同，可以覆盖 align-items 的值。align-self 的取值与 align-items 的取值相比只多了一个 auto ，其余均与 align-items 相同。 下面我们操作一下属性 align-self：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style>

.container {

display: flex;

align-items: flex-start;

width: 400px;

height: 250px;

background-color: green;

}

.item {

background-color: red;

width: 100px;

height: 100px;

margin: 10px;

}

.item1 {

align-self: inherit;

}

.item2 {

align-self: flex-end;

}

.item3 {

align-self: inherit;

}

</style>

</head>

<body>

<div class="container">

<div class="item item1">item1</div>

<div class="item item2">item2</div>

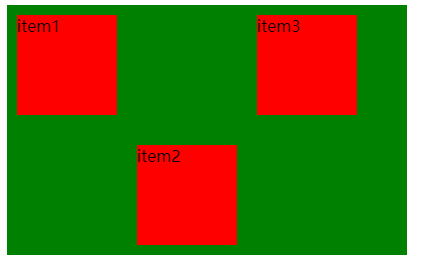
<div class="item item3">item3</div>

</div>

</body>

</html>

在浏览器中运行效果图为：



### 多媒体查询

#### 媒体类型

CSS3 的多媒体查询继承了 CSS2 的多媒体类型的所有思想，取代了查找设备的类型，CSS3 根据设置自适应显示，CSS3 有以下几种多媒体类型：

* all ：用于所有的多媒体类型设备。
* print ：用于打印机。
* screen ：用于电脑屏幕、智能手机、平板等。
* speech ：用于屏幕阅读器。

@media 的语法为：

@media not|only mediatype and (expressions) {

/\*css code here\*/

}



下面操作一下 @media：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

@media screen and (min-width: 480px) {

body {

background-color: red;

}

} /\*窗口尺寸大于480px，背景颜色为红色\*/

@media screen and (min-width: 720px) {

body {

background-color: lightgreen;

}

} /\*窗口尺寸大于720px，背景颜色为绿色\*/

</style>

</head>

<body></body>

</html>



调整浏览器的窗口大小，观察背景颜色的变化。

#### 媒体查询使用方法

媒体查询能在不同的条件下使用不同的样式，使页面在不同在终端设备下达到不同的渲染效果。媒体查询有其自己的使用规则。具体来说，媒体查询的使用方法如下。

@media 媒体类型and （媒体特性）{你的样式}

使用媒体查询必须要使用 @media 开头，然后指定媒体类型（也可以称为设备类型），随后是指定媒体特性（也可以称之为设备特性）。媒体特性的书写方式和样式的书写方式非常相似，主要分为两个部分，第一个部分指的是媒体特性，第二部分为媒体特性所指定的值，而且这两个部分之间使用冒号分隔。

媒体特性是通过 min/max 来表示大于等于或小于做为逻辑判断，接下来一起来看看媒体查询在实际项目中常用的方式。

**1.最大宽度 max-width**

max-width 是媒体特性中最常用的一个特性，其意思是指媒体类型小于或等于指定的宽度时，样式生效

@media screen and (max-width: 400px) {

.box {

display: none;

}

}

上面表示的是：当屏幕小于或等于 400px 时,页面中的 .box 都将被隐藏。

**2.最小宽度 min-width**

min-width 与 max-width 相反，指的是媒体类型大于或等于指定宽度时，样式生效。

@media screen and (min-width: 1000px) {

.box {

width: 400px;

}

}

上面表示的是：当屏幕大于或等于 1000px 时，容器 .box 的宽度为 980px。

**3.多个媒体特性使用**

媒体查询可以使用关键词 and 将多个媒体特性结合在一起。也就是说，一个媒体查询中可以包含 0 到多个表达式，表达式又可以包含 0 到多个关键字，以及一种媒体类型。

@media screen and (min-width: 700px) and (max-width: 1000px) {

.box {

width: 500px;

}

}

当屏幕在 700px~1000px 之间时，容器 .box 的宽度为 500px。

**4.not 关键词**

使用关键词 not 是用来排除某种制定的媒体类型，也就是用来排除符合表达式的设备。换句话说，not 关键词表示对后面的表达式执行取反操作。

@media not print and (max-width: 1200px) {

.box {

width: 500px;

}

}

在除打印设备和设备宽度小于 1200px 下所有设备中，容器 .box 的宽度为 500px。

### 用户界面

相比 CSS2 ，CSS3 增加了一些新的特性来调整元素尺寸、框尺寸和外边框。

* resize
* box-sizing
* outline-offset

#### resize

属性 resize 允许用户调整元素的尺寸，但是需要将元素的 overflow 属性设置为 auto 、hidden 或 scroll，resize 属性才能生效。resize 有以下四个值：

* none ：用户无法调整元素尺寸。
* both ：用户可以调整元素的高度和宽度。
* horizontal ：用户可以调整元素的宽度。
* vertical ：用户可以调整元素的高度。

下面使用值 both 操作属性 resize：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

overflow: auto; /\* auto 、hidden or scroll\*/

border: 2px solid;

width: 300px;

height: 250px;

resize: both;

}

</style>

</head>

<body>

<div>

<p>用户可以调整该元素的大小</p>

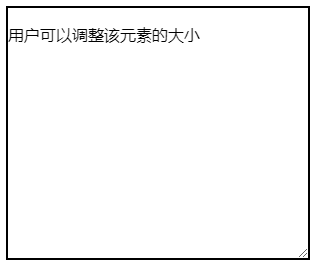
</div>

</body>

</html>



在浏览器中运行效果图为：



你可以尝试拖动边框右下角，看看会有什么事情发生。

#### box-sizing

属性 box-sizing 和盒子模型有关，它允许用户以特定的方式定义匹配某个区域的特定元素。box-sizing 有以下三个值：

* content-box ：（默认）设置属性为该值时，设定的高度和宽度值只应用到 content 内容上，不包括 padding 和 border。
* border-box ：设置属性为该值时，设定的高度和宽度应用到 content + padding + border 上，也就是说如果要得到 content 的宽高需要减去 padding 和 border。
* inherit ：继承父元素的 box-sizing 取值。

下面我们对比一下 content-box 和 border-box：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.container {

width: 660px; /\* width = item1.width + 2 \* item1.border + item2.width \*/

height: 310px; /\* height = item1.height + 2 \* item1.border \*/

display: flex;

flex-flow: row wrap;

border: 2px solid;

}

.content-box {

box-sizing: content-box;

border: 30px solid yellow;

width: 300px;

height: 250px;

}

.border-box {

box-sizing: border-box;

border: 30px solid blue;

margin-top: 30px;

width: 300px;

height: 250px;

}

</style>

</head>

<body>

<div class="container">

<div class="content-box">item1</div>

<div class="border-box">item2</div>

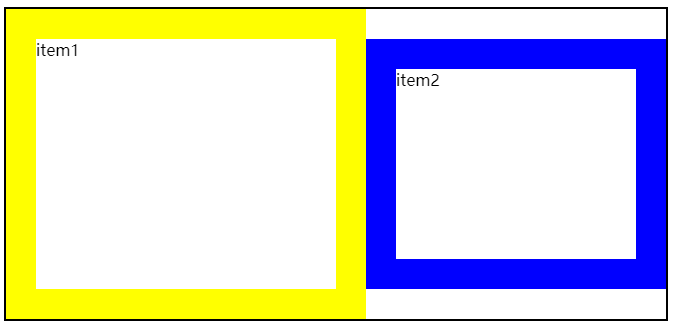
</div>

</body>

</html>



在浏览器中运行效果图为：



item1 和 item2 的边框都设置为 30px，宽高都设置成相同数值。按照 box-sizing 的属性规定，item1 的 box-sizing 为 content-box，item1 最终的宽高应该为设置的宽高加上边框和填充的大小，填充默认为 0，则 item1 最终的宽高应该为 360px 310px。item2 的 box-sizing 为 border-box，item2 最终的宽高就等于设置的宽高 300px 250px。运行后可以发现，item2 content 高加 padding 高加 border 高等于 item1 content 高，符合我们的预期。如果还有些疑惑，可以利用 container 的宽度计算一下 item1 和 item2 的宽度，会发现结果如上述对 box-sizing 属性描述的一样。

#### outline-offset

属性 outline-offset 对轮廓进行偏移，并在边框边缘进行绘制，轮廓不会占用空间，并且不一定是矩形。 下面操作 outline-offset 属性：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

div {

margin: 100px;

width: 200px;

height: 200px;

border: 2px solid;

outline: 2px solid green;

outline-offset: 20px;

}

</style>

</head>

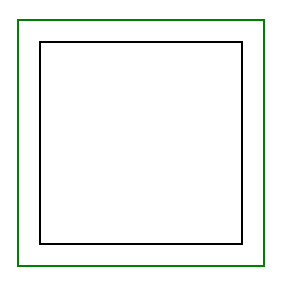
<body>

<div></div>

</body>

</html>

在浏览器中运行效果图为：



## CSS3 选择器

### 常用选择器

CSS 的常用选择器包括了通配符选择器、元素选择器、类选择器等。CSS3 中新增了一个常用选择器：

#### 通用兄弟元素选择器（E~F）

E 和 F 属于同一父元素之内，并且 F 在 E 之后，通用兄弟选择器会选择符合这样排列的所有的 F 元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.test ~ p {

background: red;

}

</style>

</head>

<body>

<div>

<p>1</p>

<p class="test">2</p>

<p>3</p>

<p>4</p>

<p>5</p>

<p>6</p>

<p>7</p>

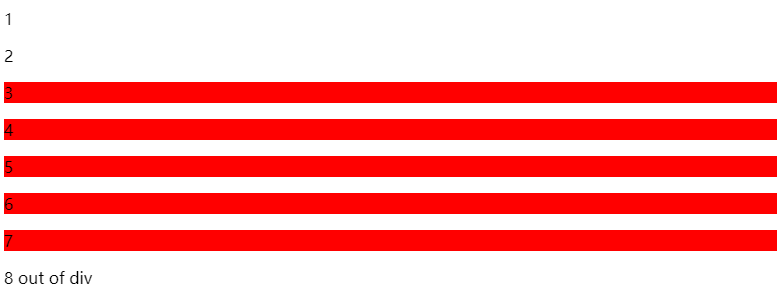
</div>

<p>8 out of div</p>

</body>

</html>

在浏览器中运行效果图为：



可以看到，选择器

.test ~ p {

background: red;

}

选择了 p.test 元素后的所有兄弟元素 <p> ,由于 <p>8 out of div</p> 和前七个 <p> 不是兄弟元素，所以未被选中。

### 属性选择器

#### E[attr^="value"]

如果你熟悉 CSS 的属性选择器，这个选择器就会很容易理解，因为它与 E[attr="value"]很类似，只不过多了一个 ^ ，该选择器选择属性名为 attr ，属性值以 value 开头的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

a[title^='test'] {

background: red;

}

</style>

</head>

<body>

<a href="https://www.lanqiao.cn" title="test">title="test"</a><br />

<a href="https://www.lanqiao.cn" title="testaaa">title="testaaa"</a><br />

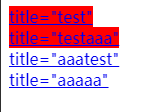
<a href="https://www.lanqiao.cn" title="aaatest">title="aaatest"</a><br />

<a href="https://www.lanqiao.cn" title="aaaaa">title="aaaaa"</a><br />

</body>

</html>

在浏览器中运行效果图为：



可以看到前两个 <a> 标签的 title 都是以 test 开头，它们都被选择器选中了。

#### E[attr$="value"]

该选择器选择属性名为 attr，属性值以 value 结尾的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

a[title$='test'] {

background: red;

}

</style>

</head>

<body>

<a href="https://www.lanqiao.cn" title="test">title="test"</a><br />

<a href="https://www.lanqiao.cn" title="testaaa">title="testaaa"</a><br />

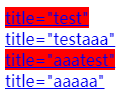
<a href="https://www.lanqiao.cn" title="aaatest">title="aaatest"</a><br />

<a href="https://www.lanqiao.cn" title="aaaaa">title="aaaaa"</a><br />

</body>

</html>

在浏览器中运行效果图为：



可以看到第一个和第三个 <a> 标签的 title 都是以 test 结尾，它们都被选择器选中了。

#### E[attr\*="value"]

该选择器选择属性名为 attr，属性值包含 value 的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

a[title\*='test'] {

background: red;

}

</style>

</head>

<body>

<a href="https://www.lanqiao.cn" title="test">title="test"</a><br />

<a href="https://www.lanqiao.cn" title="testaaa">title="testaaa"</a><br />

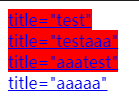
<a href="https://www.lanqiao.cn" title="aaatest">title="aaatest"</a><br />

<a href="https://www.lanqiao.cn" title="aaaaa">title="aaaaa"</a><br />

</body>

</html>

在浏览器中运行效果图为：



可以看到前三个 <a> 标签的 title 都包含 test ，它们都被选择器选中了。

### 伪类选择器

#### E:root

该选择器选择文档的根节点，效果和设置 html 相同

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

:root {

background: lightgreen;

}

/\*

与下列样式效果相同

html{

background: lightgreen;

}

\*/

</style>

</head>

<body></body>

</html>

在浏览器中运行效果图为：



#### E:nth-child(n)

该选择器选择 E 元素父元素的第 n 个子元素。n 有两种取值：

* n 取值为整数

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:nth-child(3) {

background-color: red;

}

</style>

</head>

<body>

<p>1</p>

<a href="">2</a>

<p>3</p>

<p>4</p>

</body>

</html>

****

在浏览器中运行效果图为：



选择器选取了 <p> 的父元素的第三个子元素。

注：这些子元素不一定是同种类型，如上述例子中，父元素的第二个子元素是 <a> ,它与其它 <p> 是兄弟元素。另外，当该选择器取整数时，只能取正整数。

* n 取值为表达式

当取值为表达式时，变量只能使用 n，并且 n 从 0 开始依次加一，即取值为表达式时，选择器选择多个值，[n=0,1,2,3...]。当计算结果小于等于 0 或 大于子元素总数时，不选取元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:nth-child(2n + 1) {

background-color: red;

}

</style>

</head>

<body>

<p>1</p>

<a href="">2</a>

<p>3</p>

<p>4</p>

</body>

</html>

在浏览器中运行效果图为：



表达式为 2n-1，[n=0,1,2,3...]，计算结果为 nth-child(1)，nth-child(3)...，所以该选择器选择了第一个和第三个元素。

注：使用表达式时，CSS3 提供了两个特殊值：odd 和 even。odd 相当于 2n+1，even 相当于 2n。

#### E:nth-last-child(n)

与 E:nth-child(n) 相比多了一个 last，其实该选择器与前者十分相似，只不过前者是从前往后计数第 n 个，该选择器是从最后一个往前计数。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:nth-last-child(2n + 1) {

background-color: red;

}

</style>

</head>

<body>

<p>1</p>

<p>2</p>

<a href="">3</a>

<p>4</p>

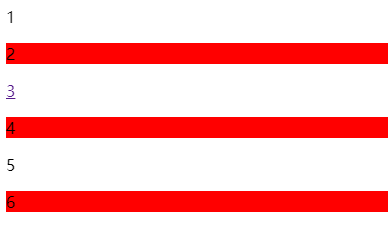
<p>5</p>

<p>6</p>

</body>

</html>

在浏览器中运行效果图为：



可以看到，从最后一个元素往前计数，第一个、第三个、第五个元素都被选择器选中。

#### E:nth-of-type()

该选择器与 E:nth-child(n) 语法相同，在选择的元素上有些区别。该元素选择的是与 E 相同类型的兄弟元素，我们来看一个例子：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:nth-of-type(2n + 1) {

background-color: red;

}

</style>

</head>

<body>

<p>1th of ele,1th of p</p>

<p>2th of ele,2th of p</p>

<a href="https://www.lanqiao.cn">3th of ele,1th of a</a>

<p>4th of ele,3th of p</p>

<p>5th of ele,4th of p</p>

<p>6th of ele,5th of p</p>

</body>

</html>

在浏览器中运行效果图为：



该选择器选择的是与 p 类型相同的兄元素，表达式为 2n+1，即选择奇数项元素，可以看到第一个第三个第五个 p 元素都被选中了。

#### E:nth-last-of-type(n)

该选择器与之前我们学习的 E:nth-last-child(n) 相类似，它与上一个选择器 E:nth-of-type(n) 选择元素的方式相同，只不过计数是从最后一个符合条件的兄弟元素开始往前计数。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:nth-last-of-type(2n + 1) {

background-color: red;

}

</style>

</head>

<body>

<p>1th of ele,1th of p</p>

<p>2th of ele,2th of p</p>

<a href="https://www.lanqiao.cn">3th of ele,1th of a</a>

<p>4th of ele,3th of p</p>

<p>5th of ele,4th of p</p>

<p>6th of ele,5th of p</p>

<p>7th of ele,6th of p</p>

</body>

</html>

在浏览器中运行效果图为：



#### E:last-child

该选择器选择 E 元素的父元素的最后一个子元素：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:last-child {

background: red;

}

</style>

</head>

<body>

<p>wo</p>

<p>ai</p>

<p>shi</p>

<p>yan</p>

<p>lou</p>

</body>

</html>

****

在浏览器中运行效果图为：



注：该选择器需要满足父元素的最后一个元素和 E 才会选取，否则不会选取，例如下面这段代码

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:last-child {

background: red;

}

</style>

</head>

<body>

<p>wo</p>

<p>ai</p>

<p>shi</p>

<p>yan</p>

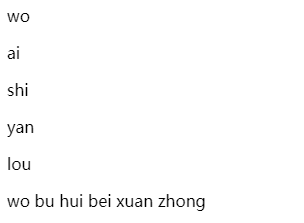
<p>lou</p>

<span>wo bu hui bei xuan zhong</span>

</body>

</html>

在浏览器中运行效果图为：



#### E:first-of-type

该选择器选择 E 元素父元素的子元素的第一个与 E 元素类型一致的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:first-of-type {

background: red;

}

</style>

</head>

<body>

<h1>shiyanlou</h1>

<p>wo</p>

<p>ai</p>

<p>shi</p>

<p>yan</p>

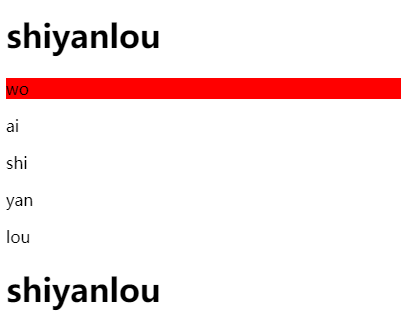
<p>lou</p>

<h1>shiyanlou</h1>

</body>

</html>

在浏览器中运行效果图为：



可以看到，第一个子元素是 <h1>，第一个符合类型一致的子元素是 <p>wo</p>，选择器选择了该元素。

#### E:last-of-type

该选择器选择 E 元素父元素的子元素的最后一个与 E 元素类型一致的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:last-of-type {

background: red;

}

</style>

</head>

<body>

<h1>shiyanlou</h1>

<p>wo</p>

<p>ai</p>

<p>shi</p>

<p>yan</p>

<p>lou</p>

<h1>shiyanlou</h1>

</body>

</html>

在浏览器中运行效果图为：



可以看到，最后一个子元素是 <h1>，最后一个符合类型一致的子元素是 <p>lou</p>，选择器选择了该元素。

注：first-of-type、 last-of-type 与之前的 last-child 和在 CSS2 中我们这里没有讲解的 first-child 的区别就是，first-of-type、last-of-type 选择的是第一个或最后一个类型一致的元素，这个元素一定存在，而 first-child 和 last-child 则第一个或最后一个元素与 E 类型一致时才会选择，否则选择器不生效，即符合的元素不一定存在。

#### E:only-child

该元素选择的是 E 是其父元素唯一的子元素：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

border: 2px solid;

}

.div2 {

margin-top: 10px;

border: 2px solid;

}

p:only-child {

background: red;

}

</style>

</head>

<body>

<div class="div1">

<p>hello world</p>

</div>

<div class="div2">

<p>hello shiyanlou</p>

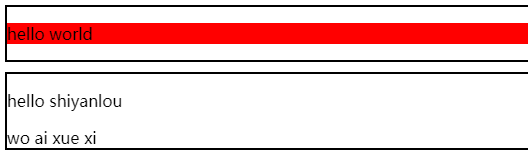
<span>wo ai xue xi</span>

</div>

</body>

</html>

在浏览器中运行效果图为：



可以看到，父元素中唯一有一个 <p> 元素的是 div1，div2 中有两个元素，所以选择器选择了 <p>hello world</p>。

#### E:only-of-type

该选择器选择的是 E 是其父元素唯一的子元素类型：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.div1 {

border: 2px solid;

}

.div2 {

margin-top: 10px;

border: 2px solid;

}

p:only-of-type {

background: red;

}

</style>

</head>

<body>

<div class="div1">

<p>hello world</p>

</div>

<div class="div2">

<p>hello shiyanlou</p>

<span>wo ai xue xi</span>

</div>

</body>

</html>

在浏览器中运行效果图为：



可以看到，选择器选择了两个元素，在两个 div 块中，都只有一个 <p> 类型，所以都被选取。

#### E:empty

该选择器选择的是空元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

p:empty {

width: 1000px;

height: 20px;

background: red;

}

</style>

</head>

<body>

<p>hello world</p>

<p></p>

<p>hello shi yan lou</p>

</body>

</html>

在浏览器中运行效果图为：



可以看到，第二个空的 <p> 元素被选取了。

#### E:target

该选择器选择的是一个 id 与 当前 url 匹配的元素

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

:target {

border: 2px solid;

background: lightgreen;

}

</style>

</head>

<body>

<a href="#p1">jump to p1</a><br />

<a href="#p2">jump to p2</a>

<p id="p1">p1</p>

<p id="p2">p2</p>

</body>

</html>

在浏览器中运行效果图为：



点击两个 link 链接，可以发现，对应跳转的元素样式相应的改变了。

#### E:not(s)

该选择器匹配不符合参数选择器 s 描述的元素。

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

.demo {

color: lightgreen;

}

p:not(.demo) {

color: red;

}

:not(.demo) {

color: blue;

}

</style>

</head>

<body>

<p class="demo">hello world</p>

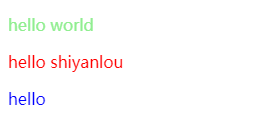
<p>hello shiyanlou</p>

<span>hello</span>

</body>

</html>

在浏览器中运行效果图为：



可以看到，第一个否定伪类选择器选择了 class 不为 demo 的 <p> 元素，样式为红色；第二个否定伪类选择器选择了 class 不为 demo 的 <span> 元素，样式为蓝色。

#### E:enabled & E:disabled

该选择器匹配的是元素的可用和禁用状态：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

input:enabled {

background: red;

}

input:disabled {

background: lightgreen;

}

</style>

</head>

<body>

<input

type="text"

name="test1"

placeholder="disabled"

disabled="true"

/><br />

<input type="text" name="test2" placeholder="enabled" />

</body>

</html>

在浏览器中运行效果图为:

此处输入图片的描述

可以看到，输入框的可用和不可用两种状态都被选择器获取到了。

#### E:checked

该选择器一般用于 radio-button 或 checkbox，选择它们的 checked 状态：

<!DOCTYPE html>

<html>

<head>

<title>shiyanlou</title>

<style type="text/css">

input[type='checkbox']:checked {

margin: 100px;

}

</style>

</head>

<body>

<input type="checkbox" name="checkbox" />

</body>

</html>

点击复选框，查看选择器选中元素的效果。可以看到，当复选框属性为 checked 时，复选框被选中，样式改变。