河南中医药大学信息技术学院（智能医疗行业学院）智能医学工程专业《互联网医疗服务开发》课程

第06章：JavaScript开发

冯顺磊

河南中医药大学信息技术学院（智能医疗行业学院）
河南中医药大学信息技术学院智能医疗教研室
https://aitcm.hactcm.edu.cn
2025/12/3

本章概要

- 异步编程
- 绘图
- 本地存储

1. 异步编程

- JavaScript代码采用单线程执行模式，这意味着在同一时刻，它仅能执行一个任务。JavaScript不支持多线程执行，主要因为其诞生于浏览器环境。作为浏览器脚本，它需与用户交互并操作UI的DOM结构，若支持多线程并发操作，会引发一系列复杂问题。例如，当线程1正在DOM上添加一个节点时，线程2却尝试删除该节点，这种操作必然会产生冲突。为避免此类冲突，JavaScript被设计为单线程执行模式。
- 单线程执行的特性是，前一个任务结束后，后一个任务才会开始执行。通常，在单线程执行模式下，代码多以同步编程的形式编写，处理器会严格依照既定顺序执行代码，每句代码执行后会立即产生效果或返回执行结果，示例代码如下。

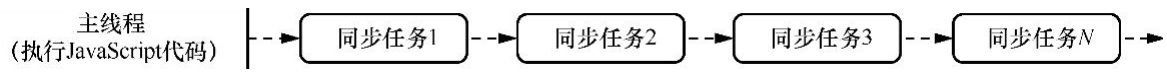
```
1. let x=1;
2. console.log(x);
```


1. 异步编程

- 但同步编程存在一定的局限性，具体体现为当某句或某段代码的执行耗时较长，如网络通信、文件存取、数据库连接等操作时，程序会持续等待，直至代码执行完毕。在此期间，整个程序将处于阻塞状态。例如，在浏览器中打开开发者工具，并在控制台执行以下代码，会发现此代码执行过程中无法点击浏览器页面。这就是因为同步代码尚未执行完毕，程序陷入阻塞，无法继续执行后续任务。

```
1. for (let i = 0; i < 100000000; i++) {
2.     let date = new Date();
3.     myDate = date
4. }
```

- 由此，可知当执行同步任务时，其执行顺序如下图所示。

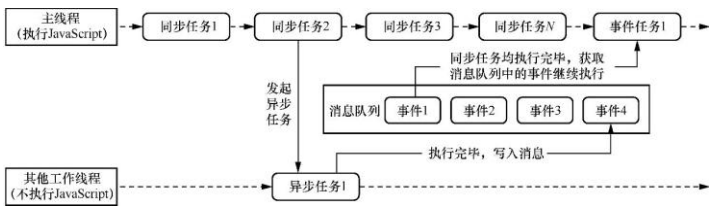


1. 异步编程

- 若因计算量过大而无法及时处理是正常情况，但在多数情形下，CPU处于闲置状态，诸如网络通信、文件读取以及数据库连接等操作，通常不会占用CPU资源，仅需长时间等待结果。
- 因此，采用同步编程模式不仅效率欠佳，且会导致用户体验不佳。
- 因此，JavaScript支持采用异步编程模式，将需要等待的网络通信、文件读取、数据库连接等任务进行挂起操作，优先执行后续无需等待的任务。待挂起任务产生返回结果后，再执行相应的回调任务。
- 基于JavaScript的异步运行机制，先后诞生了不同的异步编程模式，主要的异步编程模式如下。
  - 回调函数
  - Promise对象(ECMAScript 6)
  - async/await语法(ECMAScript 7)

1. 异步编程

- 异步执行的机制如下
  - JavaScript 主线程按顺序依次执行各个同步任务。
  - 当执行到特定代码时，JavaScript 会通知其他非 JavaScript 线程执行任务，这些任务不运行 JavaScript，也不会进入 JavaScript 主线程。
  - 此时，JavaScript 主线程不会等待，而是继续执行下一个同步任务。
  - 当非 JavaScript 线程完成任务后，会将执行结果以事件通知的形式存入 JavaScript 任务队列。
  - 待 JavaScript 主线程的所有同步任务执行完毕，主线程会检查 JavaScript 任务队列中存在哪些事件，定位到具体的回调函数代码，随后以同步任务的方式依次执行这些回调函数。



1. 异步编程

- 回调函数是JavaScript 最早且最基础的异步解决方案，其核心思路在于将一个函数（即回调函数）作为参数传递给另一个执行异步操作的函数。当异步操作完成后，执行异步操作的函数会调用此回调函数，并将结果或错误信息作为参数传递给它，其代码示例如下。

```
1. // 基础的回调案例
2. function greet(name, callback) {
3.     console.log('hello ' + name)
4.     callback()
5.  }
6.
7. function callbackFunction() {
8.   console.log('Callback function is executed!');
9. }

10. greet('Licy', callbackFunction);

11. // 输出
12. // hello Licy
13. // Callback function is executed!
```


1. 异步编程

- 若以 setTimeout 函数为例，模拟异步获取数据的回调函数，它将接收一个回调函数和延迟时间作为参数，在指定时间后执行该回调函数，示例代码如下。

```
1. // 模拟异步获取数据
2. function fetchData(url, callback) {
3.   console.log(`Fetching data from ${url}...`);
4.   setTimeout(() => {
5.     if (url === '/users') {
6.       callback(null, [{ id: 1, name: 'Alice' }]); // 成功，传递数据
7.     } else if (url === '/posts/1') {
8.       callback(null, { userId: 1, title: 'My First Post' }); // 成功
9.     } else {
10.       callback(new Error('Not Found'), null); // 失败，传递错误
11.     }
12.   }, 1000);
13. }
```

1. 异步编程

1.2 回调函数

- 尽管回调函数具有简单直观的优点，但在多个异步操作存在依赖关系时，极易形成“回调地狱”（Callback Hell），即代码出现层层嵌套的情况，导致代码难以阅读、理解和维护，同时错误处理也变得复杂且分散，示例代码如下。

```
// 回调地狱示例
fetchData('/users', (err1, users) => {
  if (err1) {
    console.error('Error fetching users:', err1.message);
  } else {
    console.log('Users fetched:', users);
    const userId = users[0].id;
    fetchData(`/posts/${userId}`, (err2, post) => {
      if (err2) {
        console.error('Error fetching post:', err2.message);
      } else {
        console.log('Post fetched:', post);
        // 如果还有后续依赖操作，会继续嵌套...
      }
    });
  }
});
```


1. 异步编程

1.3 Promise对象

- 为解决回调地狱的问题，ECMAScript 2015（ES6）标准化了 Promise 对象，其代表一个异步操作的最终完成（或失败）及其结果值。一个 Promise 对象有三种状态：Pending（进行中）、Fulfilled（已成功）和 Rejected（已失败）。Promise 使得异步代码的编写更符合线性的思维习惯。开发者可以创建一个 Promise 来包装一个异步操作，并在操作完成时调用 resolve（成功）或 reject（失败），其示例代码如下。

```
// 使用 Promise 封装异步操作
function fetchDataPromise(url) {
  return new Promise((resolve, reject) => {
    console.log(`Fetching data from ${url}...`);
    setTimeout(() => {
      if (url === '/users') {
        resolve([{ id: 1, name: 'Alice' }]); // 成功
      } else if (url === '/posts/1') {
        resolve({ userId: 1, title: 'My First Post' }); // 成功
      } else {
        reject(new Error('Not Found')); // 失败
      }
    }, 1000);
  });
}
```

1. 异步编程

1.3 Promise对象

- Promise 提供 .then() 方法用于注册操作成功时的回调，以及 .catch() 方法（或 .then() 的第二个参数）用于注册操作失败时的回调。通过链式调用 .then() 方法，可以将嵌套的回调结构扁平化，使代码逻辑更清晰，错误处理也更集中和方便，其示例代码如下。

```
1. // 使用 Promise 链式调用
2. fetchDataPromise('/users')
3.   .then(users => {
4.     console.log('Users fetched:', users);
5.     const userId = users[0].id;
6.     return fetchDataPromise(`/posts/${userId}`); // 返回新的 Promise 实现链式调用
7.   })
8.   .then(post => {
9.     console.log('Post fetched:', post);
10.    // 可以继续 .then() 处理后续操作
11.  })
12.  .catch(error => {
13.    console.error('An error occurred:', error.message); // 统一处理链中的任何错误
14.  });
```


1. 异步编程

1.4 async与await

- 尽管 Promise 显著提升了异步编程体验，但在应对复杂异步流程时，链式调用有时仍显得冗长烦琐。ECMAScript 2017 （ES8） 引入了 async 和 await 关键字，它们作为基于 Promise 构建的语法糖，旨在使异步代码的表现形式更趋近于同步代码，具体说明如下。
  - async 关键字用于声明一个函数为异步函数，此函数会隐式返回一个 Promise。
  - await 操作符仅能在 async 函数内部使用，它会暂停 async 函数的执行，直至其后的 Promise 对象达到 Fulfilled 状态，随后恢复执行并将 Promise 的解决值作为结果返回。
  - 若 Promise 被 Rejected，await 会抛出错误，这使得开发者能够运用标准的 try...catch 语句捕获异步操作中的错误。

1. 异步编程

1.4 async与await

- async/await 显著提升了异步代码的可读性与可维护性，让代码结构更为清晰、逻辑更加直观，同时使错误处理更具统一性和简洁性，具体代码示例如下。

```
// 使用 Async/Await
async function fetchUserData() {
  try {
    console.log('Starting data fetch...');
    const users = await fetchDataPromise('/users'); // 等待 Promise 解决
    console.log('Users fetched:', users);
    const userId = users[0].id;
    const post = await fetchDataPromise(`/posts/${userId}`); // 等待下一个 Promise
    console.log('Post fetched:', post);
    console.log('Data fetch complete.');
    return post; // async 函数返回的值会被包装成 Promise
  } catch (error) {
    console.error('Failed to fetch data:', error.message);
    // 可以在这里处理错误或向上抛出
    throw error; // 重新抛出错误，让调用者知道失败了
  }
}

// 调用 async 函数
fetchUserData()
  .then(result => console.log('Async function finished successfully with result:', result))
  .catch(err => console.log('Async function failed:', err.message));
```


1. 异步编程

1.5 AJAX

- AJAX，全称为 Asynchronous JavaScript and XML（异步的 JavaScript 与 XML），其并非一种全新的编程语言，而是运用现有标准的创新方法。
- 它是一项技术组合，主要涵盖用于样式设计的HTML/XHTML、CSS，用于显示与交互的JavaScript，用于动态呈现和交互的DOM（文档对象模型），以及用于与服务器进行异步通信的XMLHttpRequest对象或Fetch API。
- 其核心特性为“异步性”。在同步操作过程中，浏览器会等待服务器响应，此期间用户界面通常会被锁定；而在异步操作时，浏览器发送请求后可继续执行其他脚本并处理用户交互，待服务器响应返回，再通过回调函数或Promise处理响应数据。
- 尽管AJAX名称中包含XML，但当前数据交换格式更常采用JSON，也可使用纯文本或其他格式。

1. 异步编程

- XMLHttpRequest（XHR）作为实现 AJAX 的传统核心对象，提供了客户端与服务器之间的数据传输接口。运用 XHR 发送请求，一般包含以下步骤：
  - 创建 XMLHttpRequest 实例；
  - 调用 open() 方法初始化请求，明确 HTTP 方法（例如 GET、POST）、请求的 URL 以及请求是否为异步；
  - 可通过设置请求头（如 setRequestHeader()）添加额外信息；
  - 使用 send() 方法发送请求，对于 POST 请求，请求体数据将作为 send() 方法的参数。
- 为处理服务器响应，需监听 XHR 对象的状态变化。
  - onreadystatechange 事件处理器会在 readyState 属性每次变更时被触发。
  - readyState 属性体现请求状态，范围从 0（未初始化）至 4（请求完成且响应就绪）。
  - 当 readyState 为 4 时，还需检查 status 属性（HTTP 状态码），通常 status 为 200 意味着请求成功。
  - 请求成功后，可通过 responseText（获取字符串形式的响应数据）或 responseXML（获取 XML 格式的响应数据）访问服务器返回的内容。

1. 异步编程

- 现代浏览器还支持更为简洁的 onload 和 onerror 事件来处理成功与失败的情况，以下为使用 XMLHttpRequest 发送 GET 请求的简易示例：

```
const xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  // 检查请求是否完成且成功
  if (xhr.readyState === 4 && xhr.status === 200) {
    // 处理响应数据
    console.log('Received data:', xhr.responseText);
    // 在这里可以更新页面 DOM
  } else if (xhr.readyState === 4) {
    // 处理错误情况
    console.error('Request failed with status:', xhr.status);
  }
};
// 初始化一个 GET 请求，目标 URL 为 '/api/data'，异步执行
xhr.open('GET', '/api/data', true);
// 发送请求
xhr.send();
```

1. 异步编程

- 由此可知AJAX的基本工作流程如下。
  - 由用户在浏览器中的某个操作触发，例如点击按钮或输入内容。
  - 操作会调用 JavaScript 函数，该函数创建并配置 XMLHttpRequest 对象或使用 Fetch API。
  - JavaScript 向服务器发送一个 HTTP 请求。
  - 服务器接收到请求后进行处理，可能涉及数据库查询、业务逻辑计算等，然后将响应数据（如 JSON 或 HTML 片段）返回给浏览器。
  - 浏览器中的 JavaScript 接收到响应后，会执行预设的回调函数或 Promise 的处理程序。在这个处理程序中，JavaScript 解析响应数据，并使用 DOM 操作来更新网页的特定部分，而无需刷新整个页面。

1. 异步编程

- Fetch API 作为 XMLHttpRequest 的现代替代方案，具备更强大、更灵活的特性。
- 它提供了一个简洁而高效的接口，用于异步获取网络资源。
- 基于 Promise 实现的 Fetch API，使异步操作的链式调用和错误处理更为直观便捷，有效避免了传统 XHR 中“回调地狱”的问题。
- fetch() 函数是 Fetch API 的核心，它至少需要一个参数，即要请求资源的 URL。
- 该函数返回一个 Promise，此 Promise 会在接收到服务器响应头后立即解析为一个 Response 对象。
- 这个 Response 对象包含了响应的所有信息，如状态码、头部信息等。
- 不过，响应体需要通过调用 Response 对象的方法（如 response.json()、response.text()、response.blob() 等）来异步获取，这些方法同样会返回 Promise。

1. 异步编程


```
fetch('/api/data') // 发送 GET 请求到 '/api/data'
  .then(response => {
    // 检查响应状态是否表示成功 (status code 200-299)
    if (!response.ok) {
      // 如果不成功，抛出错误，会被 catch 捕获
      throw new Error('Network response was not ok: ' + response.statusText);
    }
    // 解析响应体为 JSON 格式
    return response.json();
  })
  .then(data => {
    // 处理成功获取并解析的 JSON 数据
    console.log('Received data:', data);
    // 在这里可以更新页面 DOM
  })
  .catch(error => {
    // 处理请求过程中的任何错误（网络错误、解析错误等）
    console.error('Fetch error:', error);
  });
```

- 相较于 XMLHttpRequest（XHR），Fetch API 具备更为清晰的语义、更强大的功能，例如对 Service Workers 的集成以及请求和响应对象的标准化，并采用基于 Promise 的更出色的异步处理模型。然而，Fetch API 默认情况下不会发送 cookies，需要进行显式配置，并且其错误处理机制与 XHR 存在差异，例如HTTP 状态码错误不会直接触发 catch 操作。

1. 异步编程

- AJAX 技术在现代 Web 开发中得到广泛应用，极大地提升了用户体验。其常见应用场景如下：
  - 在不刷新页面的前提下提交表单数据并展示结果；
  - 动态加载页面内容，如通过无限滚动加载新闻或商品列表；
  - 实现搜索框的自动完成功能或提供实时搜索建议；
  - 构建实时聊天应用，动态更新消息；
  - 在地图应用中，当进行拖动或缩放操作时异步加载地图

- AJAX适用于各种需要与服务器进行频繁、小规模数据交互的场景，通过减少整页刷新的频率，AJAX 让 Web 应用运行更加流畅、响应更为迅速，更趋近于桌面应用程序的使用体验。

2. 绘图

- <canvas> 元素实现了一个无色透明的图形容器，可被用来通过 JavaScript（ Canvas API 或 WebGL API ）绘制图形及图形动画。canvas 可以用于动画、游戏画面、数据可视化、图片编辑以及实时视频处理等方面。其中，Canvas API 主要聚焦于 2D 图形，而 WebGL API 则用于绘制硬件加速的 2D 和 3D 图形。
- <canvas> 元素只有两个属性：width 和 height，当没有设置宽度和高度的时候，<canvas> 会初始化宽度为 300px 和高度为150px。

```
1. <canvas id="tutorial" width="150" height="150"></canvas>
```


2. 绘图

- getContext()
  - <canvas> 元素创造了一个固定大小的画布，然而，要在这个画布上进行绘制操作，必须先获取其渲染上下文 ，getContext()就是用来获得渲染上下文的方法，该方法接收两个参数，分别是上下文类型和上下文属性，其语法说明如下。

```
1. let ctx = canvas.getContext(contextType, contextAttributes?);
```

  - 其中的上下文类型（contextType）通常有以下几种常见取值，这些取值决定了不同的渲染上下文类型，具体取值如下。
    - 2d：创建一个二维渲染上下文
    - webgl：创建一个三维渲染上下文（WebGL 版本 1）
    - webgl2：创建一个三维渲染上下文（WebGL 版本 2）
    - bitmaprenderer：创建一个只提供将 canvas 内容替换为指定 ImageBitmap 功能的 ImageBitmapRenderingContext。

2. 绘图

- getContext()
  - 其中的上下文属性（contextAttributes）则根据不同的上下文类型有所不同，具体取值如下。
  - 2d
    - alpha：布尔值，表明 canvas 包含一个 alpha 通道。如果设置为 false, 浏览器将认为 canvas 背景总是不透明的, 这样可以加速绘制透明的内容和图片.
  - webgl
    - alpha: 布尔值，表明 canvas 包含一个 alpha 通道。
    - antialias: 布尔值，表明是否开启抗锯齿。
    - depth: 布尔值，表明绘制缓冲区包含一个深度至少为 16 位的缓冲区。
    - failIfMajorPerformanceCaveat: 布尔值，表明在一个系统性能低的环境是否创建该上下文。
    - powerPreference: 指示浏览器在运行 WebGL 上下文时使用相应的 GPU 电源配置。 可能值如下:
      - default: 自动选择，默认值。
      - high-performance: 高性能模式。
      - low-power: 节能模式。
    - premultipliedAlpha: 布尔值，表明排版引擎将假设绘制缓冲区包含预混合 alpha 通道。
    - preserveDrawingBuffer: 布尔值，是否保存缓冲区，直到被清除或被使用者覆盖。
    - stencil: 布尔值，表明绘制缓冲区包含一个深度至少为 8 位的模板缓冲区。

2. 绘图

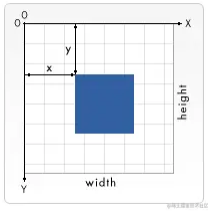
- getContext()
  - 使用getContext()方法获取渲染上下文后，绘制了两个长方形，且其中的一个有着透明度，其代码示例如下。

```
<body>
  <canvas id="canvas" width="150" height="150"></canvas>
  <script>
    var canvas = document.getElementById("canvas"); // 得到DOM对象
    if (canvas.getContext) {
      var ctx = canvas.getContext("2d"); // 得到渲染上下文

      // 绘制第一个长方形
      ctx.fillStyle = "rgb(200,0,0)";
      ctx.fillRect(10, 10, 55, 50);

      // 绘制第二个长方形
      ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
      ctx.fillRect(30, 30, 55, 50);
    }
  </script>
</body>
```

2. 绘图

2.1 Canvas基础知识

- 坐标
  - 在canvas绘图之前需要先了解一下 canvas 的坐标体系（x,y）。canvas 是一个二维网格，原点 (0,0) 在左上角，所有元素的位置都相对于原点定位取正数。所以图中蓝色方形左上角的坐标为距离左边（X 轴）x 像素，距离上边（Y 轴）y 像素，它的坐标就是 (x,y)。



2. 绘图

2.2 基础图形绘制

- 绘制矩形
  - 首先是绘制矩形，<canvas> 提供了三种方法绘制矩形，用以满足不同的绘制需求，矩形的左上角为 (x,y)，具体方法如下。
    - fillRect(x, y, width, height)：绘制一个填充的矩形
    - strokeRect(x, y, width, height)：绘制一个矩形的边框
    - clearRect(x, y, width, height)：清除指定矩形区域，让清除部分完全透明
  - 使用fillRect()函数绘制了一个边长为 100px 的黑色正方形，使用clearRect()函数从正方形的中心开始擦除了一个 60*60px 的正方形，使用strokeRect()在清除区域内生成一个 50*50 的正方形边框，其代码示例如下。

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

    ctx.fillRect(25, 25, 100, 100);
    ctx.clearRect(45, 45, 60, 60);
    ctx.strokeRect(50, 50, 50, 50);
  }
}
```

2. 绘图

- 绘制路径
  - 图形的基本元素为路径，路径是由不同颜色和宽度的线段或曲线相连所构成的、呈现出不同形状的点的集合。无论是一个完整的路径，还是其中的子路径，均为闭合状态。运用路径绘制图形时，需遵循以下步骤：首先，创建路径起始点；接着，运用绘图命令绘制路径；随后，封闭路径。路径生成后，即可通过描边或填充路径区域的方式对图形进行渲染，其中使用得到方法如下。
    - beginPath()：新建一条路径，生成之后，图形绘制命令被指向路径上生成路径。
    - closePath()：闭合路径之后图形绘制命令又重新指向到上下文中。
    - stroke()：通过线条来绘制图形轮廓。
    - fill()：通过填充路径的内容区域生成实心的图形。

2. 绘图

- 绘制路径
  - 生成路径的第一步叫作 beginPath()。每次这个方法调用之后，列表清空重置，然后就可以重新绘制新的图形。
  - 第二是调用函数指定绘制路径，本章中会详细介绍各种绘制路径的函数及示例 。
  - 第三项为闭合路径 closePath()，此方法并非必需。该方法会绘制一条从当前点到起始点的直线，从而闭合图形。若图形已经处于闭合状态，也就是当前点即为起始点时，该函数不会执行任何操作。
  - 需注意，调用 fill() 函数时，所有未闭合的形状将自动闭合，因此无需调用 closePath() 函数；而调用 stroke() 函数时，形状不会自动闭合。
  - 使用路径绘制一个三角形，其代码示例如下。

```
1. function draw() {
2.   var canvas = document.getElementById("canvas");
3.   if (canvas.getContext) {
4.     var ctx = canvas.getContext("2d");

5.     ctx.beginPath();
6.     ctx.moveTo(75, 50);
7.     ctx.lineTo(100, 75);
8.     ctx.lineTo(100, 25);
9.     ctx.fill();
10.  }
11.}
```

2. 绘图

- 绘制路径
  - moveTo()函数不能画出任何东西，只是从一个点到另一个点的移动过程，是路径绘制中的重要辅助操作，能灵活调整笔触起始位置。
    - moveTo(x, y)：将笔触移动到指定的坐标 x 以及 y 上。
  - 在 canvas 完成初始化或调用 beginPath() 方法后，可通过 moveTo() 函数来设置路径起点。此外，还能利用 moveTo() 函数绘制不连续的路径。
  - 使用moveTo()绘制一个笑脸，其代码示例如下。

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

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


2. 绘图

- 绘制路径
  - 在进行直线绘制时，需要运用 lineTo() 方法。此方法能够精确地在画布上确定直线的走向与终点，为各类图形的绘制奠定基础。
    - lineTo(x, y)：绘制一条从当前位置到指定 x 以及 y 位置的直线。
  - 此方法包含两个参数：x 和 y，它们表示坐标系中直线终点的坐标。直线的起点与先前的绘制路径相关，前一路径的终点即为后续路径的起点，以此类推。此外，还可通过 moveTo() 函数更改起点位置。
  - 使用lineTo()绘制两个三角形，一个是填充的，另一个是描边的，其示例代码如下。

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

    // 填充三角形
    ctx.beginPath();
    ctx.moveTo(25, 25);
    ctx.lineTo(105, 25);
    ctx.lineTo(25, 105);
    ctx.fill();

    // 描边三角形
    ctx.beginPath();
    ctx.moveTo(125, 125);
    ctx.lineTo(125, 45);
    ctx.lineTo(45, 125);
    ctx.closePath();
    ctx.stroke();
  }
}
```

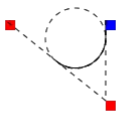
2. 绘图

- 绘制路径
  - 绘制圆弧或者圆，使用arc()或arcTo()，这两个方法各自具有独特的参数和使用方式，能够满足不同的绘图需求。
    - arc(x, y, radius, startAngle, endAngle, anticlockwise)：画一个以（x,y）为圆心的以 radius 为半径的圆弧（圆），从 startAngle 开始到 endAngle 结束，按照 anticlockwise 给定的方向（默认为顺时针）来生成。
  - 使用arc()绘制不同角度的圆弧以展示其灵活应用效果，其代码示例如下。

```
for (i = 0; i < 12; i++) {
  ctx.beginPath();
  let x = 25 + i * 50;
  let y = 25;
  let radius = 20;
  let startAngle = 0;
  let endAngle = Math.PI + (Math.PI * i) / 12; // 从半圆到全圆

  ctx.arc(x, y, radius, startAngle, endAngle);
  ctx.fill();
}
```

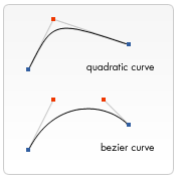

2. 绘图

- 绘制路径
  - arcTo(x1, y1, x2, y2, radius)：根据给定的控制点和半径画一段圆弧，再以直线连接两个控制点，其图片示例如下。



2. 绘图

- 绘制路径
  - 贝塞尔曲线，一般用来绘制复杂有规律的图形。它通过控制点来精确地控制曲线的形状和走向，在图形设计和动画制作中应用广泛，具体使用方式如下。
  - • quadraticCurveTo(cp1x, cp1y, x, y)：绘制二次贝塞尔曲线，cp1x,cp1y 为一个控制点，x,y 为结束点。
  - • bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)：绘制三次贝塞尔曲线，cp1x,cp1y为控制点一，cp2x,cp2y为控制点二，x,y为结束点。



2. 绘图

- 绘制路径
  - 使用quadraticCurveTo()绘制对话气泡，其代码示例如左。

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

    // 二次贝塞尔曲线
    ctx.beginPath();
    ctx.moveTo(75, 25);
    ctx.quadraticCurveTo(25, 25, 25, 62.5);
    ctx.quadraticCurveTo(25, 100, 50, 100);
    ctx.quadraticCurveTo(50, 120, 30, 125);
    ctx.quadraticCurveTo(60, 120, 65, 100);
    ctx.quadraticCurveTo(125, 100, 125, 62.5);
    ctx.quadraticCurveTo(125, 25, 75, 25);
    ctx.stroke();
  }
}
```

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

    //三次贝塞尔曲线
    ctx.beginPath();
    ctx.moveTo(75, 40);
    ctx.bezierCurveTo(75, 37, 70, 25, 50, 25);
    ctx.bezierCurveTo(20, 25, 20, 62.5, 20, 62.5);
    ctx.bezierCurveTo(20, 80, 40, 102, 75, 120);
    ctx.bezierCurveTo(110, 102, 130, 80, 130, 62.5);
    ctx.bezierCurveTo(130, 62.5, 130, 25, 100, 25);
    ctx.bezierCurveTo(85, 25, 75, 37, 75, 40);
    ctx.fill();
  }
}
```

  - 使用bezierCurveTo()绘制心形，其代码示例如右。

2. 绘图

- 绘制路径
  - 除了初始所见的绘制矩形方式外，还有三种可以直接在画布上绘制矩形的额外方法。同样地，也存在 rect() 方法，可将矩形路径添加到当前路径中。
    - rect(x, y, width, height)：绘制一个左上角坐标为（x,y），宽高为 width 以及 height 的矩形。
  - 使用rect()方法绘制矩形，其代码示例如下。

```
// 获取canvas元素和绘图上下文
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
// 开始新路径
ctx.beginPath();
// 创建矩形路径
ctx.rect(20, 20, 150, 100);
// 描边路径，绘制矩形
ctx.stroke();
```


2. 绘图

- 绘制路径
  - Path2D 能够缓存或记录绘图命令，类似于将某一段路径封装成一个可复用的函数。这种方式不仅简化了代码，还提升了性能。在实际绘图时，使用 Path2D 对象可以快速回顾路径，避免重复编写相同的绘图命令，尤其适用于需要多次绘制相同图形的场景。
    - Path2D()：Path2D()会返回一个新初始化的 Path2D 对象，可能将某一个路径作为变量，或者将一个包含 SVG path 数据的字符串作为变量。
  - 其语法说明如下。

```
new Path2D(); // 空的 Path 对象
new Path2D(path); // 克隆 Path 对象
new Path2D(d); // 从 SVG 建立 Path 对象
```

  - 其代码示例如下。

```
var path1 = new Path2D();
path1.rect(10, 10, 100, 100);

var path2 = new Path2D(path1);
path2.moveTo(220, 60);
path2.arc(170, 60, 50, 0, 2 * Math.PI);

ctx.stroke(path2);
```

2. 绘图

- 绘制路径
  - 若要为图形上色，可借助两个关键属性实现，即 fillStyle 和 strokeStyle。通过设置这两个属性的颜色值，能为图形赋予丰富多样的色彩和样式，使绘制的图形更加生动和美观。
    - fillStyle = color：设置图形的填充颜色。
    - strokeStyle = color：设置图形轮廓的颜色。
  - 其语法格式如下。

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


2. 绘图

- 绘制路径
  - 在以下示例中，将运用两层 for 循环来绘制方格阵列，且每个方格具有不同颜色。具体而言，我们使用两个变量 i 和 j 为每个方格生成唯一的 RGB 色彩值。在此过程中，仅对红色和绿色通道的值进行修改，而保持蓝色通道的值恒定。通过调整这些颜色通道的值，能够生成各式各样的色板。此外，通过提高渐变频率，还可绘制出类似于 Photoshop 中的调色板，其代码示例如下。

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 6; i++) {
    for (let j = 0; j < 6; j++) {
      ctx.fillStyle = `rgb(${Math.floor(255 - 42.5 * i)} ${Math.floor(
        255 - 42.5 * j,
      )} 0)`;
      ctx.fillRect(j * 25, i * 25, 25, 25);
    }
  }
}
```

2. 绘图

- 绘制路径
  - 以下示例与上述示例颇为相似，不过此次运用的是 strokeStyle 属性，并非绘制方格，而是借助 arc 方法来绘制圆形。其代码示例如下。

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 6; i++) {
    for (let j = 0; j < 6; j++) {
      ctx.strokeStyle = `rgb(0 ${Math.floor(255 - 42.5 * i)} ${Math.floor(
        255 - 42.5 * j,
      )})`;
      ctx.beginPath();
      ctx.arc(12.5 + j * 25, 12.5 + i * 25, 10, 0, 2 * Math.PI, true);
      ctx.stroke();
    }
  }
}
```


2. 绘图

- 绘制路径
  - Canvas提供了两种渲染文本的方法，即填充文本和绘制文本边框。
    - fillText(text, x, y [, maxWidth])：在指定的 (x,y) 位置填充指定的文本。绘制的最大宽度是可选的。
    - strokeText(text, x, y [, maxWidth])：在指定的 (x,y) 位置绘制文本边框。绘制的最大宽度是可选的。
  - 使用fillText()方法在画布上填充指定文本，代码示例如下。

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.font = "48px serif";
  ctx.fillText("好世界", 10, 50);
}
```

  - 使用strokeStyle()方法在画布上为绘制的文本添加边框样式，代码示例如下。

```
function draw() {
  const ctx = document.getElementById("canvas").getContext("2d");
  ctx.font = "48px serif";
  ctx.strokeText("好世界", 10, 50);
}
```

2. 绘图

- 绘制路径
  - 以上示例展示了如何使用 strokeStyle 为文本添加边框，除了字体和边框样式，还可通过其他属性进一步调整文本显示效果，如文本对齐、基线对齐等，能让文本呈现更加多样化，具体如下所示。
    - font = value：当前用来绘制文本的文本样式。这个字符串使用和 CSS font 属性相同的语法。默认的字体是 10px sans-serif。
    - textAlign = value：文本对齐选项。可选的值包括：start、end、left、right 或 center。默认值是 start。
    - textBaseline = value：基线对齐选项。可选的值包括：top、hanging、middle、alphabetic、ideographic、bottom。默认值是 alphabetic。
    - direction = value：文本方向。可能的值包括：ltr、rtl、inherit。默认值是 inherit。

2. 绘图

- canvas 具备强大的图像操作能力，可用于动态图像合成，或作为图形背景，也适用于构建游戏界面等场景。浏览器所支持的任意格式外部图片均可使用，例如 PNG、GIF 或 JPEG 格式。甚至，同一页面中其他 canvas 元素生成的图片也能作为图片源，引入图像到 canvas 里需要两步操作。
  - 获取一个指向 HTMLImageElement 对象或另一个 canvas 元素的引用作为源，也可通过提供 URL 来使用图片
  - 利用 drawImage() 函数将图片绘制到画布上
- canvas 的 API 可使用以下类型之一作为图片的源，这些源统一由 CanvasImageSource 类型来引用，具体如下。
  - HTMLImageElement：此类图片由 Image() 函数构造而成，或为任意 <img> 元素所呈现。
  - HTMLVideoElement：以 HTML 的 <video> 元素作为图像源，可从视频中抓取当前帧作为图像。
  - HTMLCanvasElement：可将另一个 <canvas> 元素用作图像源。
  - ImageBitmap：这是一种具备高性能的位图，能够实现低延迟绘制，可从上述所有源以及其他若干种源中生成。

2. 绘图

- 使用canvas绘制图像时可使用 Image() 构造函数创建一个新的 HTMLImageElement 对象，其代码示例如下。

```
1. var img = new Image(); // 创建一个<img>元素
2. img.src = "myImage.png"; // 设置图片源地址
```

- 脚本执行后，图片开始加载。若在调用 drawImage 时，图片尚未加载完成，则不会有任何效果。因此，应利用 load 事件确保在图片加载完毕后再使用它，其代码示例如下。

```
1. var img = new Image(); // 创建 img 元素
2. img.onload = function () {
3.   // 执行 drawImage 语句
4. };
5. img.src = "myImage.png"; // 设置图片源地址
```


2. 绘图

- 一旦获取到源图对象，便能够运用 drawImage 方法将其渲染至 canvas 中。drawImage 方法具备三种形式，以下为基础形式。
  - drawImage(image, x, y)：其中 image 是 image 或者 canvas 对象，x 和 y 是其在目标 canvas 里的起始坐标。
- 使用背景图可避免绘制复杂背景，从而节省大量代码。仅需使用一个 image 对象，随后在其 onload 事件的响应函数中触发绘制操作。通过 drawImage 方法，可将背景图精准放置于 canvas 的左上角 (0,0) 处，其代码示例如下。

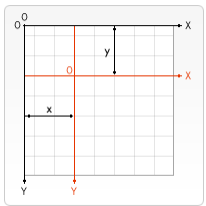
```
1. function draw() {
2.   var ctx = document.getElementById("canvas").getContext("2d");
3.   var img = new Image();
4.   img.onload = function () {
5.     ctx.drawImage(img, 0, 0);
6.     ctx.beginPath();
7.     ctx.moveTo(30, 96);
8.     ctx.lineTo(70, 66);
9.     ctx.lineTo(103, 76);
10.    ctx.lineTo(170, 15);
11.    ctx.stroke();
12.  };
13.  img.src = "backdrop.png";
14.}
```

2. 绘图

- 在探讨图形变形之前，先介绍两种在绘制复杂图形时必不可少的方法。理解图形的变形与组合对于绘图工作至关重要，它们能够使我们更加灵活地掌控图形的位置、大小和方向。运用这两种方法，在绘制复杂图形时能够更好地维持画面的原有状态，避免不同图形在绘制过程中相互干扰，这对于精确绘图而言十分关键。
  - save()：保存画布 (canvas) 的所有状态
  - restore()：save 和 restore 方法是用来保存和恢复 canvas 状态的，都没有参数。canvas 的状态就是当前画面应用的所有样式和变形的一个快照。

2. 绘图

- translate 方法，它用来移动 canvas 和它的原点到一个不同的位置，通过改变原点位置实现图形的移动效果。
  - translate(x, y)：translate方法接受两个参数。x 是左右偏移量，y 是上下偏移量，如下图所示。




```
1. function draw() {
2.   var ctx = document.getElementById("canvas").getContext("2d");
3.   for (var i = 0; i < 3; i++) {
4.     for (var j = 0; j < 3; j++) {
5.       ctx.save();
6.       ctx.fillStyle = "rgb(" + 51 * i + ", " + (255 - 51 * i) + ", 255)";
7.       ctx.translate(10 + j * 50, 10 + i * 50);
8.       ctx.fillRect(0, 0, 25, 25);
9.       ctx.restore();
10.    }
11.  }
12.}
```

2. 绘图

- rotate方法，它用于以原点为中心旋转 canvas。使用该方法能灵活改变图形的方向，为绘制具有不同角度的图形提供便利。通过设置不同的旋转角度，可绘制出各种倾斜的图形，满足多样化的绘图需求。在使用时，需注意旋转角度的设置要符合预期，以确保绘制出准确的图形。
  - rotate(angle) 方法：该方法仅接受一个参数，即旋转角度 (angle)，此角度为顺时针方向，且以弧度为单位。

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

  // left rectangles, rotate from canvas origin
  ctx.save();
  // blue rect
  ctx.fillStyle = "#0095DD";
  ctx.fillRect(30, 30, 100, 100);
  ctx.rotate((Math.PI / 180) * 25);
  // grey rect
  ctx.fillStyle = "#4D4E53";
  ctx.fillRect(30, 30, 100, 100);
  ctx.restore();

  // right rectangles, rotate from rectangle center
  // draw blue rect
  ctx.fillStyle = "#0095DD";
  ctx.fillRect(150, 30, 100, 100);

  ctx.translate(200, 80); // translate to rectangle center
  // x = x + 0.5 * width
  // y = y + 0.5 * height
  ctx.rotate((Math.PI / 180) * 25); // rotate
  ctx.translate(-200, -80); // translate back

  // draw grey rect
  ctx.fillStyle = "#4D4E53";
  ctx.fillRect(150, 30, 100, 100);
}
```


2. 绘图

- 使用缩放能够对 canvas 中的图形像素数目进行增减，进而实现形状与位图的缩小或放大操作。这种缩放功能可使图形在不同场景下呈现出恰当的大小，以满足多样化的绘图需求。通过灵活调整缩放因子，能够精准控制图形的尺寸变化，为绘图工作赋予更多创意与可能性。
  - scale(x, y)：scale方法可以缩放画布的水平和垂直的单位。两个参数都是实数，可以为负数，x 为水平缩放因子，y 为垂直缩放因子，如果比 1 小，会缩小图形，如果比 1 大会放大图形。默认值为 1，为实际大小。

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

  // draw a simple rectangle, but scale it.
  ctx.save();
  ctx.scale(10, 3);
  ctx.fillRect(1, 10, 10, 10);
  ctx.restore();

  // mirror horizontally
  ctx.scale(-1, 1);
  ctx.font = "48px serif";
  ctx.fillText("MDN", -135, 120);
}
```

2. 绘图

- 变形矩阵能够达成复杂多样的图形变换效果，为图形绘制赋予了更多的可能性。通过精心设置参数，可实现图形的倾斜、缩放与移动等操作，进而满足多样化的绘图需求。然而，在使用过程中，务必留意参数的取值范围，以防出现异常状况。
  - transform(a, b, c, d, e, f)：将当前的变形矩阵乘上一个基于自身参数的矩阵，如下面的矩阵所示：

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

  - 如果任意一个参数是 Infinity，变形矩阵也必须被标记为无限大，否则会抛出异常，函数的参数各自代表如下：
    - a：水平方向的缩放
    - b：竖直方向的倾斜偏移
    - c：水平方向的倾斜偏移
    - d：竖直方向的缩放
    - e：水平方向的移动
    - f：竖直方向的移动

2. 绘图

- 其代码示例如下：

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

3.   var sin = Math.sin(Math.PI / 6);
4.   var cos = Math.cos(Math.PI / 6);
5.   ctx.translate(100, 100);
6.   var c = 0;
7.   for (var i = 0; i <= 12; i++) {
8.     c = Math.floor((255 / 12) * i);
9.     ctx.fillStyle = "rgb(" + c + "," + c + "," + c + ")";
10.    ctx.fillRect(0, 0, 100, 10);
11.    ctx.transform(cos, sin, -sin, cos, 0, 0);
12.  }

13.  ctx.setTransform(-1, 0, 0, 1, 100, 100);
14.  ctx.fillStyle = "rgba(255, 128, 255, 0.5)";
15.  ctx.fillRect(0, 50, 100, 100);
16.}
```

2. 绘图

- canvas 上的所有绘制操作均通过 JavaScript 进行操控，因此可以实现简单的交互动画。但使用 canvas 制作动画的最大限制或许在于，图像一旦绘制完成，若要移动它，就必须对所有元素（包括之前已绘制的）进行重绘。重绘操作极为耗时，且性能高度依赖计算机的处理速度。
- 使用canvas绘制动画的基本步骤如下。
  - 清空画布：除非后续绘制的内容将完全覆盖画布（如背景图），否则需清空画布。最简单的方式是使用 clearRect 方法。
  - 保存画布状态：若要更改会影响画布状态的设置（如样式、变形等），且希望每一帧绘制时都恢复到初始状态，则需先保存当前画布状态。
  - 绘制动画图形：此步骤为重绘动画帧。
  - 恢复画布状态：若已保存画布状态，可先恢复该状态，再重绘下一帧。

2. 绘图

- 在 canvas 上绘制内容需借助 canvas 提供的方法或自定义方法，且通常只有在脚本执行结束后才能看到绘制结果。例如，在 for 循环中完成动画几乎是不可行的。因此，若要实现动画效果，就需要采用能够定时执行重绘的方法。实现这种动画操控有两种途径，第一种是使用 setInterval 和 setTimeout 方法，它们能在设定的时间点执行重绘，这两种方法能满足不同的定时需求，为动画操控提供了灵活的选择。
  - setInterval()：当设定好间隔时间后，function 会定期执行
  - setTimeout()：在设定好的时间之后执行函数
- 第二种是使用 requestAnimationFrame 方法，此方法能告知浏览器期望执行一个动画，并在重绘前请求浏览器执行特定函数来更新动画，其优势在于能与浏览器的重绘周期同步，节省系统资源，提升动画性能。
  - requestAnimationFrame()：告诉浏览器希望执行一个动画，并在重绘之前，请求浏览器执行一个特定的函数来更新动画

2. 绘图

2.6 实现动画

```
1. <canvas id="canvas" width="300" height="300"></canvas>
```

```
1. const sun = new Image();
2. const moon = new Image();
3. const earth = new Image();
4. function init() {
5.   sun.src = "canvas_sun.png";
6.   moon.src = "canvas_moon.png";
7.   earth.src = "canvas_earth.png";
8.   window.requestAnimationFrame(draw);
9. }

10.function draw() {
11.   const ctx = document.getElementById("canvas").getContext("2d");

12.   ctx.globalCompositeOperation = "destination-over";
13.   ctx.clearRect(0, 0, 300, 300); // 清除画布

14.   ctx.fillStyle = "rgb(0 0 0 / 40%)";
15.   ctx.strokeStyle = "rgb(0 153 255 / 40%)";
16.   ctx.save();
17.   ctx.translate(150, 150);

18.   // 地球
19.   const time = new Date();
20.   ctx.rotate(
21.     ((2 * Math.PI) / 60) * time.getSeconds() +
22.       ((2 * Math.PI) / 60000) * time.getMilliseconds(),
23.   );
24.   ctx.translate(105, 0);
25.   ctx.fillRect(0, -12, 40, 24); // 阴影
26.   ctx.drawImage(earth, -12, -12);
```

```
27.   // 月亮
28.   ctx.save();
29.   ctx.rotate(
30.     ((2 * Math.PI) / 6) * time.getSeconds() +
31.       ((2 * Math.PI) / 6000) * time.getMilliseconds(),
32.   );
33.   ctx.translate(0, 28.5);
34.   ctx.drawImage(moon, -3.5, -3.5);
35.   ctx.restore();

36.   ctx.restore();

37.   ctx.beginPath();
38.   ctx.arc(150, 150, 105, 0, Math.PI * 2, false); // 地球轨道
39.   ctx.stroke();

40.   ctx.drawImage(sun, 0, 0, 300, 300);

41.   window.requestAnimationFrame(draw);
42.}

43.init();
```


2. 绘图

2.6 实现动画

```
1. <canvas id="canvas" width="150" height="150">当前时间</canvas>
```

```
1. function clock() {
2.   const now = new Date();
3.   const canvas = document.getElementById("canvas");
4.   const ctx = canvas.getContext("2d");
5.   ctx.save();
6.   ctx.clearRect(0, 0, 150, 150);
7.   ctx.translate(75, 75);
8.   ctx.scale(0.4, 0.4);
9.   ctx.rotate(-Math.PI / 2);
10.  ctx.strokeStyle = "black";
11.  ctx.fillStyle = "white";
12.  ctx.lineWidth = 8;
13.  ctx.lineCap = "round";

14.  // 小时刻度
15.  ctx.save();
16.  for (let i = 0; i < 12; i++) {
17.    ctx.beginPath();
18.    ctx.rotate(Math.PI / 6);
19.    ctx.moveTo(100, 0);
20.    ctx.lineTo(120, 0);
21.    ctx.stroke();
22.  }
23.  ctx.restore();

24.  // 分钟刻度
25.  ctx.save();
26.  ctx.lineWidth = 5;
27.  for (let i = 0; i < 60; i++) {
28.    if (i % 5 !== 0) {
29.      ctx.beginPath();
30.      ctx.moveTo(117, 0);
31.      ctx.lineTo(120, 0);
32.      ctx.stroke();
33.    }
34.    ctx.rotate(Math.PI / 30);
35.  }
36.  ctx.restore();
```

```
37.   const sec = now.getSeconds();
38.   // 要显示扫秒式的时钟，请使用：
39.   // const sec = now.getSeconds() +
      now.getMilliseconds() / 1000;
40.   const min = now.getMinutes();
41.   const hr = now.getHours() % 12;

42.   ctx.fillStyle = "black";

43.   // 显示图像描述
44.   canvas.innerText = `当前时间：${hr}:${min}`;

45.   // 时针
46.   ctx.save();
47.   ctx.rotate(
48.     (Math.PI / 6) * hr + (Math.PI / 360) * min +
      (Math.PI / 21600) * sec,
49.   );
50.   ctx.lineWidth = 14;
51.   ctx.beginPath();
52.   ctx.moveTo(-20, 0);
53.   ctx.lineTo(80, 0);
54.   ctx.stroke();
55.   ctx.restore();

56.   // 分针
57.   ctx.save();
58.   ctx.rotate((Math.PI / 30) * min + (Math.PI / 1800)
      * sec);
59.   ctx.lineWidth = 10;
60.   ctx.beginPath();
61.   ctx.moveTo(-28, 0);
62.   ctx.lineTo(112, 0);
63.   ctx.stroke();
64.   ctx.restore();

65.
```

```
66.   // 秒针
67.   ctx.save();
68.   ctx.rotate((sec * Math.PI) / 30);
69.   ctx.strokeStyle = "#D40000";
70.   ctx.fillStyle = "#D40000";
71.   ctx.lineWidth = 6;
72.   ctx.beginPath();
73.   ctx.moveTo(-30, 0);
74.   ctx.lineTo(83, 0);
75.   ctx.stroke();
76.   ctx.beginPath();
77.   ctx.arc(0, 0, 10, 0, Math.PI * 2, true);
78.   ctx.fill();
79.   ctx.beginPath();
80.   ctx.arc(95, 0, 10, 0, Math.PI * 2, true);
81.   ctx.stroke();
82.   ctx.fillStyle = "rgb(0 0 0 / 0%)";
83.   ctx.arc(0, 0, 3, 0, Math.PI * 2, true);
84.   ctx.fill();
85.   ctx.restore();

86.   ctx.beginPath();
87.   ctx.lineWidth = 14;
88.   ctx.strokeStyle = "#325FA2";
89.   ctx.arc(0, 0, 142, 0, Math.PI * 2, true);
90.   ctx.stroke();

91.   ctx.restore();

92.   window.requestAnimationFrame(clock);
93.}

94.window.requestAnimationFrame(clock);
```

3. 本地存储

3.1 Cookie

- Cookie 的主要功能并非本地存储，而是“维持状态”。
- 由于 HTTP 协议是无状态的，它自身不会保存请求与响应之间的通信状态。
- 这极大地限制了交互式 Web 应用程序的实现，以网上购物为例，若不借助额外手段，服务器将无法得知用户的具体购买内容，而 Cookie 就是绕过 HTTP 无状态性的“额外手段”之一。
- 它能让服务器在用户每次请求时识别其身份，记录用户操作和状态，从而为用户提供连贯的服务体验。
- 不过，虽然 Cookie 发挥着重要作用，但其存储容量有限，通常大小限制在 4KB 左右，并且在 HTTP 请求中以明文传递，存在一定的安全风险，使用时需谨慎考虑其适用场景。
- 对于涉及用户敏感信息或大量数据存储的情况，可考虑使用 Web Storage 或 IndexedDB 等其他存储方式来弥补 Cookie 的不足，以提升数据存储的安全性和效率。

3. 本地存储

3.1 Cookie

- 借助 Cookie，服务器能够识别请求的客户端来源，进而实现客户端状态的维护。例如，在用户登录后刷新页面，请求头会携带登录时响应头（Response Header）中的 Set - Cookie 信息。当 Web 服务器接收到请求时，便可读取 Cookie 的值，并依据其内容判断和恢复用户的部分信息状态，cookie 以键值对的形式存在，如下图所示。

3. 本地存储

3.1 Cookie

- Cookie具备丰富的应用场景，可有效提升用户体验与网站服务质量。常见应用场景如下：
  - 用户选择“记住密码”后，下次访问网站时即可自动登录；购物车功能便于用户管理商品；记录用户浏览数据能够实现精准的商品或广告推荐。
  - Cookie可记录用户的地域信息，从而提供本地化服务；还能记录用户的登录状态，避免用户在浏览过程中频繁登录。
  - Cookie可统计用户的访问频率和时长，助力网站进行数据分析与优化；也能实现多语言切换功能，依据用户选择展示相应语言的页面。
  - 通过跟踪用户在不同页面间的跳转路径，Cookie有助于网站优化页面布局与导航设计。
  - 对于需要用户持续互动的网站，Cookie可记录用户的操作习惯，如设置页面的字体大小、主题颜色等，为用户打造个性化的浏览体验。
- 然而，尽管Cookie功能多样，但使用时仍需谨慎，需充分考量其存储容量、性能开销以及安全性等方面的问题。

3. 本地存储

3.1 Cookie

- 当首次访问网站时，浏览器会发出请求，服务器响应该请求后，会在响应头中添加Set-Cookie选项，将Cookie置入响应请求。当浏览器第二次发起请求时，会通过Cookie请求头部将Cookie信息发送给服务器，服务器借此辨别用户身份。此外，Cookie的过期时间、域、路径、有效期以及适用站点均可按需指定。

- 当首次访问网站时，浏览器会发出请求，服务器响应该请求后，会在响应头中添加Set-Cookie选项，将Cookie置入响应请求。当浏览器第二次发起请求时，会通过Cookie请求头部将Cookie信息发送给服务器，服务器借此辨别用户身份。此外，Cookie的过期时间、域、路径、有效期以及适用站点均可按需指定。

3. 本地存储

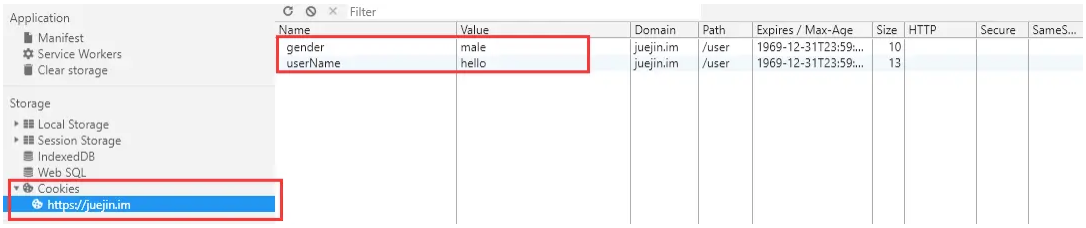
- Cookie的生成方式主要有两种：
  - 生成方式一：http response header中的set-cookie
  - 可通过响应头中的 Set-Cookie 字段指定需存储的 Cookie 值。默认情况下，domain 属性会被设置为设置 Cookie 页面的主机名，不过也能够手动对 domain 的值进行设置，当Cookie的过期时间被设定时，设定的日期和时间只与客户端相关，而不是服务端。

```
1. Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2018 07:28:00 GMT;//可以指定一个特定的过期时间（Expires）或有效期（Max-Age）
```


3. 本地存储

- Cookie的生成方式主要有两种：
  - 生成方式二：在 JavaScript中，可以借助 document.cookie 来读写 Cookie，其以键值对的形式呈现。
  - 例如在控制台输入以下三句代码，便可在Edge的 Application 面板查看生成的cookie，具体代码如下，具体图片如下。

```
1. document.cookie="userName=hello"
2. document.cookie="gender=male"
3. document.cookie='age=20;domain=.baidu.com'
```



  - 其中，domain 标识用于指定可接受 cookie 的域名。若未设置 domain，它将自动绑定到执行语句的当前域。若将其设置为 “.baidu.com”，则所有以 “baidu.com” 结尾的域名均可访问该 cookie。

3. 本地存储

- Cookie的缺陷
  - （1）Cookie不够大
  - Cookie 的大小通常限制在 4KB 左右，对于复杂的存储需求而言，这一容量明显不足。一旦 Cookie 大小超过 4KB，就会面临被裁切的情况。由此可见，Cookie 仅适用于存储少量信息。此外，许多浏览器还对单个站点的 Cookie 数量进行了限制。
  - （2）过多的 Cookie 会带来巨大的性能浪费
  - Cookie 与域名紧密关联。在同一域名下的所有请求，都会携带 Cookie。不妨设想一下，当我们仅仅是请求一张图片或一个 CSS 文件时，却仍需携带 Cookie（而 Cookie 中存储的信息在此场景下并无必要），这无疑是一种资源的浪费。尽管单个 Cookie 体积较小，但请求数量众多，随着请求的不断累积，这些不必要的 Cookie 所带来的开销将巨大。
  - Cookie 主要用于维护用户信息，且域名（domain）下的所有请求都会携带 Cookie。然而，对于静态文件的请求而言，携带 Cookie 信息并无实际意义。此时，可以通过将存储静态文件的 CDN 域名与主站域名分离的方式来解决这一问题。

3. 本地存储

- Cookie的缺陷
  - （3）由于在HTTP请求中的Cookie是明文传递的，所以安全性成问题，除非用HTTPS。
  - 因此，为提升 Cookie 的安全性，可设置 HttpOnly 和 Secure 标记。标记为 Secure 的 Cookie 仅会在通过 HTTPS 协议加密的请求中传输至服务端。即便有此保障，也切勿通过 Cookie 传输敏感信息。此外，带有 HttpOnly 标记的 Cookie 无法被客户端 JavaScript 脚本访问，仅能发送给服务端，从而有效防范脚本操作引发的安全风险。
  - 为防范跨域脚本 (XSS) 攻击，通过 JavaScript 的 Document.cookie API 无法访问带有 HttpOnly 标记的 Cookie，此类 Cookie 仅应发送给服务端。若包含服务端 Session 信息的 Cookie 不希望被客户端 JavaScript 脚本调用，那么就应当为其设置 HttpOnly 标记，其代码示例如下。

```
1. Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

3. 本地存储

- 为弥补 Cookie 的局限性，HTML5 引入了全新的本地存储解决方案——Web Storage，其分为 sessionStorage 和 localStorage 两类。借助 Web Storage，Cookie 得以专注于本职工作，即作为客户端与服务器交互的通道，维持客户端状态。
- LocalStorage具有以下特性：
  - 所保存的数据持久存在，下次访问该网站时，网页可直接读取先前保存的数据。
  - 存储空间约为5M。
  - 仅在客户端使用，不与服务端进行通信。
  - 接口封装较为完善。
- 考虑到这些特性，localStorage在众多场景中都能发挥重要作用。例如，对于图片丰富的电商网站，可将 Base64 格式的图片字符串存储于其中，以减少重复请求，提升用户体验。
- localStorage保存的数据以“键值对”形式存在，即每一项数据都有对应的键名和值，且所有数据均以文本格式保存。

3. 本地存储

- 存入数据可使用setItem方法，该方法接受两个参数，第一个为键名，第二个为要保存的数据。

```
localStorage.setItem("key","value");
```

- 读取数据可使用getItem方法，该方法仅需一个参数，即键名。

```
var valueLocal = localStorage.getItem("key");
```

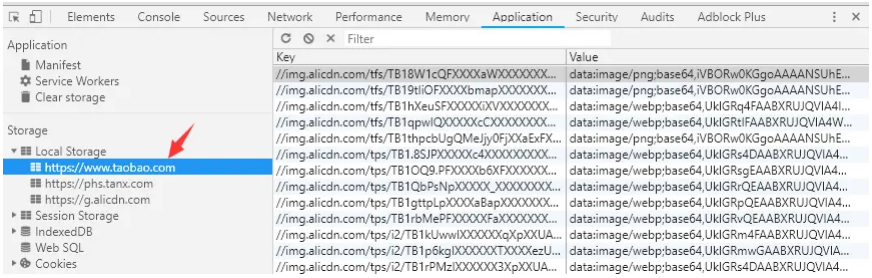
- 在使用时，首先判断浏览器是否支持LocalStorage，然后使用setItem方法存储姓名和性别信息，若要读取这些信息，可使用getItem方法获取对应键名的值，其代码示例如下。

```
<script>
if(window.localStorage){
  localStorage.setItem ('name','world')
  localStorage.setItem ( “gender','famale')
}
</script>
```

```
<body>
<div id="name"></div>
<div id="gender"></div>
<script>
var name=localStorage.getItem('name')
var gender=localStorage.getItem('gender')
document.getElementById('name').innerHTML=name
document.getElementById('gender').innerHTML=gender
</script>
</body>
```

3. 本地存储

- LocalStorage 在存储方面并无特殊限制，理论上，凡是 Cookie 无法处理、可通过简单键值对进行存取的数据存储任务，均可交由 LocalStorage 完成。鉴于 LocalStorage 具有持久性这一特点，有时更适合用其存储内容稳定的资源。例如，图片内容丰富的电商网站会利用它来存储 Base64 格式的图片字符串：



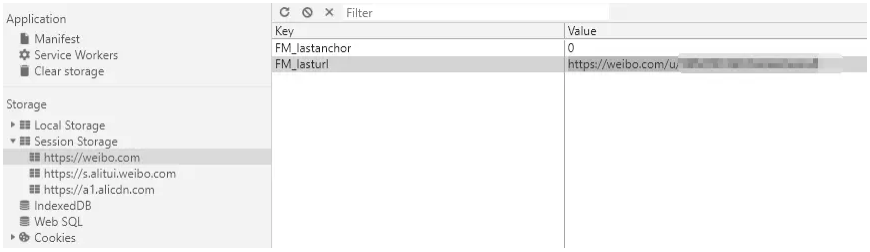
3. 本地存储

- sessionStorage
  - sessionStorage 所保存的数据仅适用于浏览器的单次会话，当会话结束（通常为窗口关闭时），数据会被清空。sessionStorage 的独特之处在于，即便两个页面处于相同域名下，只要它们并非在同一浏览器窗口中打开，其 sessionStorage 内容便无法实现共享。与之不同的是，localStorage 和 cookie 在所有同源窗口中均可共享。除保存期限存在差异外，SessionStorage 的属性和方法与 LocalStorage 完全一致，具体表现为以下特性：
    - 会话级别的浏览器存储
    - 大小为5M左右
    - 仅在客户端使用，不和服务端进行通信
    - 接口封装较好

3. 本地存储

3.2 Web Storage

- sessionStorage
  - 鉴于上述特性，sessionStorage 能够有效地维护表单信息，例如在页面刷新时，确保表单信息不会丢失。sessionStorage 更适宜存储与自身生命周期同步的会话级信息。此类信息仅适用于当前会话，当开启新会话时，这些信息需要相应地更新或释放。例如，微博的 sessionStorage 主要用于存储本次会话的浏览记录，“lasturl” 对应上一次访问的 URL 地址，该地址具有即时性。在切换 URL 时，它会随之更新；而当页面关闭时，保留此地址已无实际意义，可将其释放。使用 “sessionStorage” 处理这类数据比较合适。



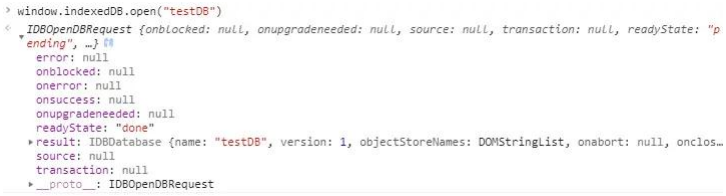
3. 本地存储

3.4 IndexedDB

- IndexedDB 作为一种低级应用程序编程接口（API），主要用于在客户端存储大量结构化数据，其中涵盖文件和二进制大对象（blobs）。此 API 借助索引功能，实现对存储数据的高性能搜索。IndexedDB 是运行于浏览器端的非关系型数据库，既然属于数据库范畴，其存储容量便绝非 5M、10M 这般量级。从理论层面而言，IndexedDB 不存在存储上限（通常情况下，其存储容量不小于 250M）。此外，它不仅能够存储字符串，还支持二进制数据的存储。
  - 键值对储存：IndexedDB 内部运用对象仓库（object store）来存储数据。它支持直接存入所有类型的数据，其中包括 JavaScript 对象。在对象仓库里，数据以“键值对”的形式保存，每条数据记录都配有唯一的主键。若主键出现重复，系统将抛出错误。
  - 异步：在进行 IndexedDB 操作时，浏览器不会被锁定，用户仍可自如地进行其他操作，这与采用同步操作方式的 LocalStorage 形成鲜明对比。IndexedDB 的异步设计旨在避免因大量数据的读写操作而导致网页性能下降。
  - 支持事务：IndexedDB 支持事务（transaction），这表明在一系列操作步骤中，一旦有一步失败，整个事务将被取消，数据库会回滚至事务发生前的状态，不会出现仅部分数据被改写的情形。
  - 同源限制：IndexedDB 遵循同源限制原则，每个数据库与创建它的域名一一对应。网页仅能访问自身域名下的数据库，无法访问跨域数据库。
  - 储存空间大：IndexedDB 的储存空间相较于 LocalStorage 更为可观，通常不少于 250MB，甚至不存在上限。
  - 支持二进制储存：IndexedDB 不仅能够储存字符串，还支持储存二进制数据，如 ArrayBuffer 对象和 Blob 对象。

3. 本地存储

- 在 IndexedDB 中，大部分操作并非采用常见的调用方法并返回结果的模式，而是采用请求-响应模式，其具体执行步骤如下。
  - （1）创建并打开 IndexedDB ---- window.indexedDB.open("testDB")
  - 该指令并不会直接返回一个数据库（DB）对象的句柄，而是返回一个 IDBOpenDBRequest 对象。所需的数据库（DB）对象存储在该对象的 result 属性中，具体如下图所示。



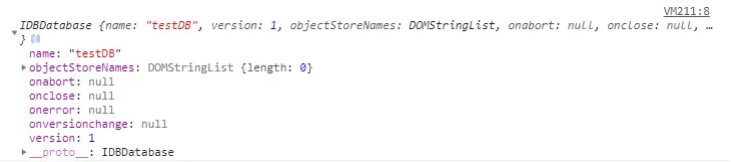
3. 本地存储

- 除了“result”，IDBOpenDBRequest 接口定义了以下几个重要属性：
  - onerror：请求失败时的回调函数句柄
  - onsuccess：请求成功时的回调函数句柄
  - onupgradeneeded：处理数据库版本变化请求的句柄
- 其代码示例如下。

```
<script>
    function openDB(name) {
        var request = window.indexedDB.open(name)//建立打开IndexedDB
        request.onerror = function (e) {
            console.log('open indexdb error')
        }
        request.onsuccess = function (e) {
            myDB.db = e.target.result//这是一个 IDBDatabase对象，这就是IndexedDB对象
            console.log(myDB.db)//此处就可以获取到db实例
        }
    }
    var myDB = {
        name: 'testDB',
        version: '1',
        db: null
    }
    openDB(myDB.name)
</script>
```

3. 本地存储

- （2）控制台将得到一个 IDBDatabase对象，这就是IndexedDB对象。成功获取该对象后，就可以对其进行操作，如正常使用关闭、删除等功能，以管理数据库资源和数据存储 。此外，还能进一步利用IndexedDB进行数据的增删改查操作，以满足实际业务对于大量结构化数据存储和管理的需求。



3. 本地存储

- （3）关闭IndexedDB：indexdb.close()，关闭操作能够释放数据库占用的资源，避免资源浪费，确保系统性能稳定。在实际应用中，当不再需要使用数据库时，应及时调用此方法关闭数据库。

```
1. function closeDB(db){
2.     db.close();
3. }
```

- （4）删除IndexedDB：window.indexedDB.deleteDatabase(indexdb)，删除操作可彻底移除数据库，释放更多存储空间。当确定不再需要某个数据库时，可使用此方法进行删除。在删除前需谨慎考虑，确保数据已无保留价值。

```
1. function deleteDB(name) {
2.   indexedDB.deleteDatabase(name)
3. }
```

3. 本地存储

3.5 本地存储方案对比

- Cookie、Session、SessionStorage和LocalStorage均为常见的Web存储解决方案，每种方案都具备其特定的适用场景与特点。
  - Cookie可用于在客户端存储数据，适用于存储会话标识符、用户偏好设置以及追踪用户行为等场景。
  - Session用于在服务器端存储并管理用户的会话状态，适用于身份验证、购物车管理以及个性化设置等场景。
  - SessionStorage用于在浏览器会话期间存储临时数据，适用于数据传递、表单数据保存以及单页应用状态管理等场景。
  - LocalStorage用于在客户端存储持久化数据，适用于本地数据存储、离线应用以及单页应用状态管理等场景。
  - 依据具体需求和场景，选择恰当的存储方案能够更有效地管理和运用数据。

	属性	存储位置	生命周期	安全性	大小限制	跨域限制
Cookie	键值对	客户端	可配置	受同源策略限制	约4KB	是
Session	会话ID和服务器端存储	服务器端	可配置	较高（会话ID保护）	无	否
SessionStorage	键值对	客户端	浏览器会话期间	同源	约5MB	否
LocalStorage	键值对	客户端	永久（需显式删除）	同源	约5MB	否

信创智能医疗系统研发课程体系
河南中医药大学信息技术学院（智能医疗行业学院）

河南中医药大学信息技术学院（智能医疗行业学院）智能医疗教研室
河南中医药大学医疗健康信息工程技术研究所