ESTR 2106 Web 开发

> Building Web Applications

①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳㉑㉒㉓㉔㉕㉖㉗㉘㉙㉚㉛㉜㉝㉞㉟㊱㊲㊳㊴㊵㊶㊷㊸㊹㊺㊻㊼㊽㊾㊿

Lecture 0

不计 Attendance.

3 Assignments：30%

Midterm：40%

Project：30%

（没有 Final，但是 Midterm 在期末进行，包含所有内容）

允许使用 AI 工具. 注意：AI 不一定正确.

Lecture 1 Web 简介

> Introduction

本课特点：轻理论，重实践.

内容：网络开发的各个基本方面.

建议浏览器：最新的 Google Chrome.

编辑器：应使用基于文本的编辑器. 如 VS Code.

推荐一款功能强大的富文本编辑器：TinyMCE

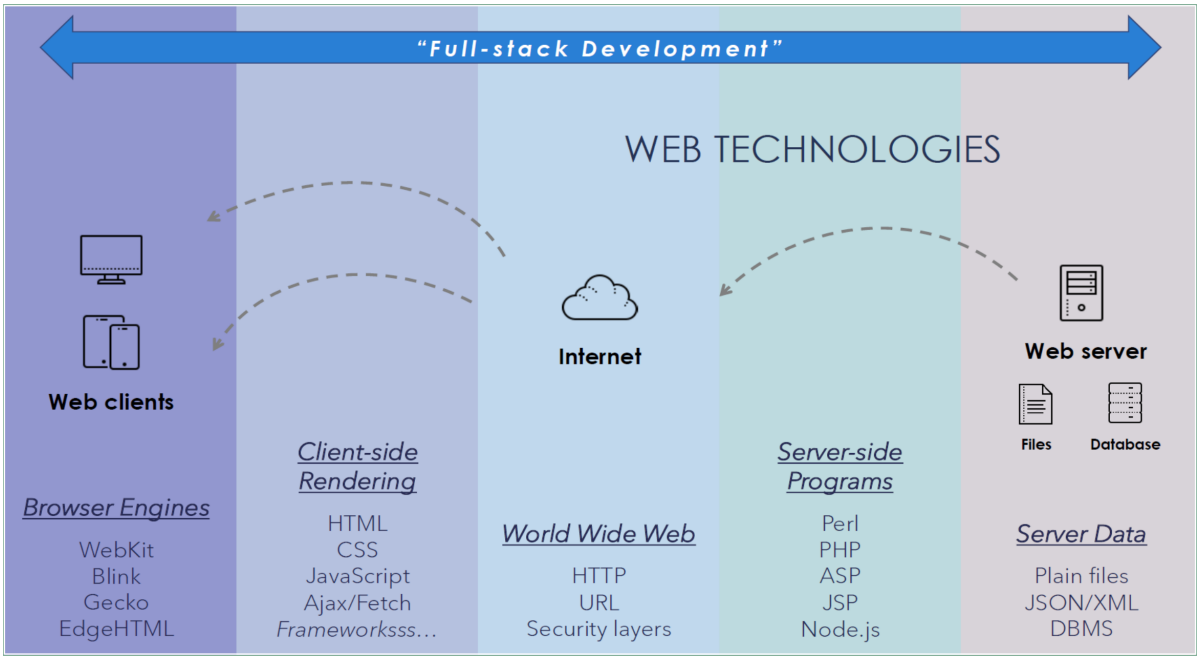
本地服务器：用 Cursor / VS Code 的 Live Server 插件即可. 作业若有具体要求，则按要求.

1. 万维网

> World Wide Web, 简写为 Web

网络（Web）是一种允许计算机共享和交互数据的服务.

网络应用程序（Web Application）通常是由浏览器处理的软件应用，通过局域网或互联网访问，并基于客户端-服务器模式.



全栈开发分为客户端（≈ 前端）和服务器端（≈ 后端）.

工作方式：客户端向服务器发送 HTTP 请求，服务器处理后返回包含 JavaScript、HTML 和 CSS 的 HTTP 相应.

1.1 客户端

- HTML
- CSS
- JavaScript

1.2 服务器端

- Web 服务器：接受请求并交付内容.
- 脚本（Scripts）：用于准备内容的程序，如 `Node.js`.
- HTTP：服务器和客户端之间的通信协议.
- 其他服务器端程序包括 Perl、PHP、ASP、JSP.

- 服务器数据：包括纯文件、JSON/XML、DBMS.

Lecture 2 HTML

1. HTML 基本知识

1.1 发展历史

待补充.

1.2 简单例子：Hello World

A simple HTML example:

```html
<!DOCTYPE html>
<html>
<head>
    <title>CSCI2720 is fun</title>
</head>

<body>
    <h1>hello world</h1>
    <p>hello world again</p>
    <!-- Every CS course has to use HELLO WORLD as the first example-->
</body>

</html>
```

- `<!DOCTYPE html>` 声明文档类型，表示 HTML5 文件.

  > 任何现代 HTML（即现在新写的）必须包含这个，否则浏览器可能进入怪异模式（兼容老 HTML 的渲染模式，可能会出现不想要的效果）

- `<head>` 部分包含有用的数据但**不用于显示**.
- `<title>` 元素通常放在 `<head>` 里，内容会显示在**浏览器窗口**（例如 Chrome 的顶部）. 此外，当用户将网页添加到浏览器的书签 / **收藏夹**时，浏览器通常会使用 `<title>` 中的内容作为默认的名称. **搜索引擎**在搜索结果中会显示 `<title>` 的主要文本. 浏览器的**历史记录**会使用 `<title>` 的内容标识网页.
- `<body>` 部分包含所有要在屏幕上显示的内容.

- `<h1>` 定义一级标题，`<p>` 定义段落（普通文本）.
- `<!-- something -->` 表示注释，渲染时会被忽略.

`test1.html`：

```
<!DOCTYPE html>
<html>

<head>
    <title>CSCI2720 is fun</title>
    123
</head>


<body>
    <p>hello world</p>
    <h1>hello world h1</h1>
    <h2>hello world h2</h2>
    <p>hello world again</p>
    <!-- something -->
</body>

<html>
```

显示为：

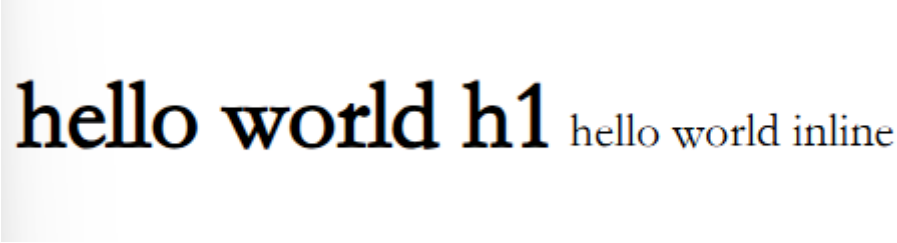
注意：这里 `<head></head>` 元素中有一个裸露的文本 "123". 这里 “裸露” 指内容没有被开始标签和结束标签包裹. 这里有两处不规范：在 `<head>` 中，通常不包含想要显示的内容，因此这个文本应该放到 `<body>` 中；即使放到 `<body>` 中，仍有一处不规范，即任何内容都应该用正确的标签来指示内容的角色. 对于普通文本，应该使用段落标签 `<p></p>` .

不过，即使有这么多规范问题，但技术上仍然能够正常渲染. 因为现代浏览器非常宽容，会尝试猜测你的意图并修复错误. 浏览器能识别到 `<head>` 标签内的 `123` 是一个错误放置的普通文本，为了最大程度兼容、显示所有内容，浏览器通常会将这些内容移动到 `body` 的开头（`<body>` 内部的第一个元素之前）进行渲染.

注意：`<h1>` 是块级元素，会占据父容器的全部可用宽度，无论其实际内容有多长，块级元素渲染时都会从**新的一行开始**，并在其**内容结束后换行**. 因此它的右侧不能添加其他内容. 如果希望一个元素/文本紧接着块级元素显示在同一行，需要使用内联元素，涉及 CSS 样式. 例如：

```
    <h1 style="display: inline-block;">hello world h1</h1>
    <p style="display: inline-block;"> hello world inline</p>
```

显示为：



1.3 基本结构

最简单的 HTML 结构形如：

```
<!DOCTYPE html>
<html>

<head>
</head>

<body>
</body>

<html>
```

其中，`<head></head>`、`<body></body>` 等称为元素，其内部可以进一步包含其他子元素.

2. 元素

元素以块（block）的形式呈现，可以在 `<head>` 里或 `<body>` 里.

2.1 Head

显示和 web 相关，但不会显示在页面上的内容. 这些内容统称为元数据（Metadata）

例如：

- `<title>`：窗口标签.
- `<link rel="icon" type="image/png" href="/images/favicon-32x32.png" sizes="32x32">`："favicon"，显示在窗口左侧的小图标，通常是你网站的 logo，能够增加辨识度和独特性.

  其中，`rel` 属性声明链接的目标文件与当前文档的关系（不能随便写，要按照规范）；

  `type` 声明图标文件的类型；`href` 指定图标文件的 URL 地址，可以是相对路径或绝对路径；

  `size` 指定图标大小.

  注意：属性的顺序不重要. 但可以出于可读性遵循一致的风格.
- `<<meta charset="UTF-8">`：声明字符编码.
- `<meta name="description" content="...">`：搜索引擎摘要
- `<meta name="keywords" content="...">` 搜索引擎关键词列表
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">` 控制网页在不同设备上的显示宽度和缩放.
- `<meta name="author" content="Dr Colin Tsang">`：声明文档作者.
- `<link rel="stylesheet" href="style.css">`：链接外部文件.

2.2 Body

包含所有希望显示在页面中的内容，例如 Headings, paragraphs, images, videos, tables...

结尾可以放 js 文件.

2.2.1 语义化结构

语义化结构：You can create a structure to present the contents in a semantic manner, using a header, paragraphs, sections, and a footer.

目的：便于维护，结构清晰；便于机器（搜索引擎）理解网页.

例：`test2.html`：

```
<!DOCTYPE html>
<html lang="zh-Hant">
<head>
    <meta charset="UTF-8">
    <title>语义化结构示例</title>
</head>
<body>

    <header>
        <h1>我的个人博客</h1>
        <p>欢迎来到我的技术分享空间。</p>
    </header>

    <nav>
        <ul>
            <li><a href="#about">关于我</a></li>
            <li><a href="#articles">文章列表</a></li>
            <li><a href="#contact">联系我</a></li>
        </ul>
    </nav>

    <section id="articles">
        <h2>最新技术文章</h2>

        <article>
            <h3>HTML 语义化入门</h3>
            <p>使用语义元素可以帮助指示文本的角色  56]。这篇文章介绍了 HTML5 的语义化标签。
</p>
            <p>作者：AI 助手</p>
        </article>

        <article>
            <h3>CSS 布局基础</h3>
            <p>我们将学习如何使用 CSS 来美化和布局页面，这是下一个主题。</p>
        </article>
    </section>

    <footer>
        <p>&copy; 2025 版权所有。</p>
        <p>联系方式：example@mail.com</p>
    </footer>

</body>
</html>
```

显示为：

我的个人博客

欢迎来到我的技术分享空间。

- 关于我
- 文章列表
- 联系我

最新技术文章

HTML 语义化入门

使用语义元素可以帮助指示文本的角色 [cite: 56]。这篇文章介绍了 HTML5 的语义化标签。

作者：AI 助手

CSS 布局基础

我们将学习如何使用 CSS 来美化和布局页面，这是下一个主题。

© 2025 版权所有。

联系方式：example@mail.com

其中，

- `<header>`：不同于 `<head>`，这是 `<body>` 的头部，通常包含显示出来的标题和介绍.
- `<nav>`：明确这是一个导航链接区块.
- `<section>`：定义了文档的一个通用且独立的部分，例如某个主题区域.
- `<article>`：在 `<section>` 内，用 `<article>` 标记每一个独立的文章，表明这是一块可以独立于网站其他内容存在的完整内容单元.

2.2.2 文本标记元素

用于丰富文本形式，通常在 `<body>` 中使用.

① **六级标题** `<h1>`~`<h6>`

`<h1>` 最大，`<h6>` 最小.

② **段落 `<p>` 与换行 `<br>`**

③ **格式化 `<b/i/u/sub/sup>`**

> 格式化（Formatting），包括上下标/粗体斜体/下划线

`<b>`：bold，加粗

`<i>`：italic，斜体

`<u>`：underline，下划线

`<sub>`：下标

`<sup>`：上标

④ **预格式化 `<pre>`**

可以让内容按照代码中的格式呈现，如果不用预格式化，连续多个空格会被渲染为单个空格.

⑤ **列表 `<ol/ul/li>`**

`<ol>`：Ordered List，有序列表，浏览器会用数字自动标记.

`<ul>`: Unordered List，无序列表，浏览器会用点来标记.

`<li>`：List Item，定义列表中的项目. 无论是有序列表还是无序列表，列表中的每一个具体内容都要用 `<li>` 标签来定义. `<li>` 必须是 `<ol>` 或 `ul` 的子元素.

⑥ **表格 `<table/tr/td/th>`**

`<tr>`：Table row，定义行.

`<td>`：table data，定义单元格.

`<th>`：Table header，定义表头单元格（默认粗体居中）

例子

例：`test5.html`

```
    <h2>数据表格</h2>
    <table border="1">
        <tr>
            <th>项目编号 (&lt;th&gt;)</th>
            <th>名称 (&lt;th&gt;)</th>
```

```
            <th>状态 (&lt;th&gt;)</th>
        </tr>
        <tr>
            <td>001 (&lt;td&gt;)</td>
            <td>HTML</td>
            <td>已完成</td>
        </tr>
        <tr>
            <td>002</td>
            <td>CSS</td>
            <td>进行中</td>
        </tr>
    </table>
```

`<table border="1">`：可以用 `border` 属性来控制表格边框的粗细. 但现代 Web 开发不常用，现代开发通常使用 CSS 来统一管理表现和样式，HTML 只负责内容的组织结构.

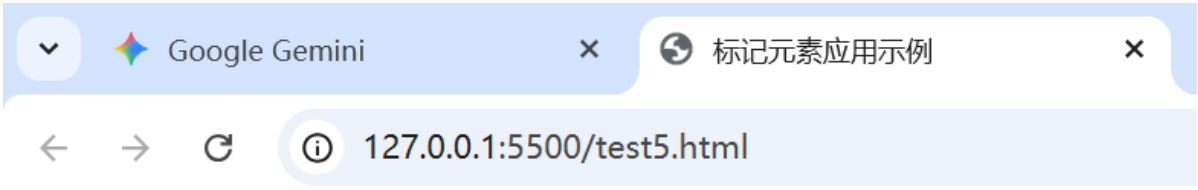
例：`test5.html`

```
<!DOCTYPE html>
<html lang="zh-Hant">
<head>
    <meta charset="UTF-8">
    <title>标记元素应用示例</title>
</head>
<body>

    <h1>一、这是最大标题 (h1)</h1>
    <p>这是在 h1 后的一个普通段落。</p>

    <h2>二、标题等级 h2</h2>
    <h3>三、标题等级 h3</h3>
    <h4>四、标题等级 h4</h4>
    <h5>五、标题等级 h5</h5>
    <h6>六、这是最小标题 (h6)</h6>

    <hr> <h2>段落与换行</h2>
    <p>
        这是一个完整的段落。段落是块级元素，
        无论你在代码中写多少空格或换行，它们最终都会被压缩成一个空格。
    </p>

    <p>
        如果我需要强制换行，<br>
        我可以使用 &lt;br&gt; 标签来插入一个
        <br>
        单行换行。
    </p>

    <hr>
```

```
    <h2>文本格式化</h2>
    <p>
        这是一段 <b>粗体文本</b>（使用 &lt;b&gt; 标签）。<br>
        这是一段 <i>斜体文本</i>（使用 &lt;i&gt; 标签）。<br>
        这是一段 <u>带下划线的文本</u>（使用 &lt;u&gt; 标签）。
    </p>
    <p>
        在科学公式中：水是 H<sub>2</sub>O（使用 &lt;sub&gt; 下标）。<br>
        在数学公式中：X 平方是 X<sup>2</sup>（使用 &lt;sup&gt; 上标）。
    </p>

    <hr>

    <h2>预格式化文本</h2>
    <p>以下代码使用 &lt;pre&gt; 标签，将保留空格和换行：</p>
    <pre>
def hello_world():
    print("Hello,")
    print("World!")
    </pre>

    <hr>

    <h2>列表应用</h2>

    <h3>无序列表 (&lt;ul&gt;)</h3>
    <ul>
        <li>第一个无序列表项</li>
        <li>第二个无序列表项</li>
        <li>第三个无序列表项</li>
    </ul>

    <h3>有序列表 (&lt;ol&gt;)</h3>
    <ol>
        <li>第一步：启动程序</li>
        <li>第二步：输入数据</li>
        <li>第三步：查看结果</li>
    </ol>

    <hr>

    <h2>数据表格</h2>
    <table border="1">
        <tr>
            <th>项目编号 (&lt;th&gt;)</th>
            <th>名称 (&lt;th&gt;)</th>
            <th>状态 (&lt;th&gt;)</th>
        </tr>
        <tr>
            <td>001 (&lt;td&gt;)</td>
            <td>HTML</td>
            <td>已完成</td>
        </tr>
        <tr>
            <td>002</td>
```

```
            <td>CSS</td>
            <td>进行中</td>
        </tr>
    </table>

</body>
</html>
```

显示为：



一、这是最大标题 (h1)

这是在 h1 后的一个普通段落。

二、标题等级 h2

三、标题等级 h3

四、标题等级 h4

五、标题等级 h5

六、这是最小标题 (h6)

段落与换行

这是一个完整的段落。段落是块级元素，无论你在代码中写多少空格或换行，它们最终都会被压缩成一个空格。

如果我需要强制换行，
我可以使用
 标签来插入一个
单行换行。

文本格式化

这是一段 **粗体文本**（使用 <b> 标签）。
这是一段 *斜体文本*（使用 <i> 标签）。
这是一段 带下划线的文本（使用 <u> 标签）。

在科学公式中：水是 H_2O（使用 <sub> 下标）。

在数学公式中：X 平方是 X^2（使用 <sup> 上标）。

预格式化文本

以下代码使用 <pre> 标签，将保留空格和换行：

```
def hello_world():
    print("Hello,")
    print("World!")
```

列表应用

无序列表 (<ul>)

- 第一个无序列表项
- 第二个无序列表项
- 第三个无序列表项

有序列表 (<ol>)

1. 第一步：启动程序
2. 第二步：输入数据
3. 第三步：查看结果

数据表格

项目编号 (<th>)	名称 (<th>)	状态 (<th>)
001 (<td>)	HTML	已完成
002	CSS	进行中

2.2.3 图像元素

> Including images

① `<img>`

`<img>` 标签：用于插入图片，现代浏览器支持多种图片类型.

```
<img src="cuhk.jpg" width="100" height="150">
```

② `<svg>`

`<svg>` 标签：用于可缩放矢量图形（Scalar Vector Graphics）

可以指定图形内容（自己画），例如 `<circle>` 或 `<line>`

例子

`test6.html`：

```
<!DOCTYPE html>
<html>
    <head>
        <title>Image test</title>
    </head>
    <body>
        <img
src="https://www.google.com/images/branding/googlelogo/1x/googlelogo_color_272x92
dp.png" alt="Google logo">
        <svg width="100" height="100">
            <circle cx="50" cy="50" r="40" stroke="black" stroke-width="2"
fill="red" />
        </svg>
    </body>
</html>
```

显示为：

关于 `<svg>` 中的 `<circle>`：

`cx` and `cy` referred to the position of the circle center.

`r` defines the radius.

The `fill` attribute define what color to fill the circle.

注意：不同于 `<h1>` 和 `<p>` 这些块级元素，`<img>` 和 `<svg>` 在 HTML 中默认是内联元素（Inline Element）. 内联元素只占据它内容所需的宽度，并允许其他内容（文本或其他内联元素）紧随其后显示在同一行.

替换特性：替换元素（图片、视频）的内容和尺寸由外部资源决定

People also use `<picture>` for detailed control on responsiveness

> 后面会讲.

3. 超链接

3.1 标签与属性

元素在代码中以一组 `<tag><\tag>` 表示，里面可以填写内容.

`<tag attribute="...">`：标签内可以添加若干属性（也可以不定义）

> HTML 大小写不敏感，但建议统一小写.

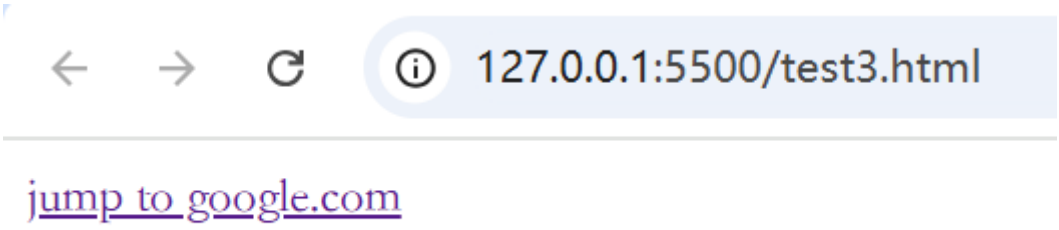
3.2 href 属性：`<a>` 标签的超链接

超链接（Hyperlink）就是一种属性. 添加超链接时，通常使用 `<a>` 元素. 例如 `test3.html`：

```
<!DOCTYPE html>
<html>

<head>
    <title>Hyperlink test</title>
</head>


<body>
    <a href="https://www.google.com">jump to google.com</a>
    <!-- something -->
```

```
</body>

</html>
```

显示为：



点击 jump to google.com 后会跳转到 https://www.google.com

> 颜色与普通文本不同，如果未访问为蓝色，已访问为紫色.

3.3 target 属性：href 打开方式

`<a>` 元素除了使用 `href` 属性指定跳转到的链接之外，还可以指定一个 `target` 属性. 例如：

```
<a href="https://www.cuhk.edu.hk" target="_blank"> Click me to open a new tab
</a>
```

常用的有：

`target="_blank"`：新开一个窗口来加载链接.

`target="_self"`：在当前浏览器窗口/标签页中加载链接.

> 点击前的页面会被覆盖，但是点击浏览器的返回键可以返回.

如果不填写，默认 `target="_self"`

3.4 URL 链接与锚点定位

超链接允许用户脱离传统的 “一页看完才能看下一页” 的线性阅读方式，而是根据兴趣在不同文档和文档内部跳转.

`href` 有两种指定方式：URL 链接与锚点定位.

3.4.1 href="https...": URL 链接

Inline links: pointing to another file.

可以是本地链接（相对位置）或外部链接（如网址）. 指向另一个文件（可以任意形式文件，不一定是 html，不一定安全）.

> 见 `3.5 绝对位置 vs 相对位置`

3.4.2 href="#...": 锚点定位

> Anchor

锚点定位允许 `<a>` 标签跳转到**同一文件的另一个位置**. 这需要另一个位置添加对应的配合属性: `id`.
例:

```
    <a href="#section1">jump to section 1</a>
    ...
    <h2 id="section1">section 1</h2>
    ...
```

完整 demo: `test4.html`

```
<!DOCTYPE html>
<html>

<head>
    <title>Anchor test</title>
</head>


<body>
    <a href="#section1">jump to section 1</a>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <h2 id="section1">section 1</h2>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
```

```
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
    <p>some content</p>
</body>

</html>
```

显示为：

jump to section 1

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

section 1

some content

some content

some content

点击超链接后：

section 1

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

some content

如果跳转到的元素上下有足够的内容，跳转后它会在页面最顶部，如果下方内容不够，会尽量把它往顶部移动，直到页面无法下滑.

3.5 绝对位置 vs 相对位置

> 待补充.

4. 特殊字符处理

> Encoding special characters

因为浏览器使用 `<` 和 `>` 这两个符号表示标签，当它们在文本中时可能会造成混淆. 因此，规定了它们在文本中的编码方式.

4.1 `<`

用 `<` 表示

4.2 `>`

用 `>` 表示

4.3 空格

空格不会造成混淆，但是默认情况下（无预格式化），HTML 的连续空白（空格/换行/制表符）会被视为一个空格.

使用 ` `（Non-breaking space）来创建固定空格，这些空格不会在浏览器窗口调整大小时折叠或引起换行.

5*. 补充材料

元素速查表

https://www.wpkube.com/html5-cheat-sheet/

HTML living standard

https://html.spec.whatwg.org

w3schoools.com HTML5 tutorial

https://www.w3schools.com/html/

MDN HTML guides and tutorials

https://developer.mozilla.org/en-US/docs/Learn/HTML

Lecture 3 Bootstrap

> https://getbootstrap.com/docs/4.0/getting-started/introduction/

1. 简介

Bootstrap 是一个用于快速开发 Web 应用程序和网站的**前端框架**，基于 HTML, CSS 和 JavaScript.

Bootstrap 提供了一系列预先设计的 CSS 样式和 JavaScript 组件.

2. 优势

易于上手：具备良好的浏览器兼容性.

一致性：组件外观一致且美观.

响应式设计：提供响应式网格系统用于布局.

与原生 CSS 相比：

- 使用 Bootstrap，可以通过设置简单的 `class` 属性来美化 HTML 元素，无需自己编写 CSS 代码.
- 对于初学者很方便，但对于样式精细控制的灵活度较低.

3. 使用

3.1 添加到 HTML

在 HTML 的 `<head>` 中添加：

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
```

> 类似调包的操作. 通过 CDN（Content Delivery Network）引入最新版本的 CSS 文件.

`<head>` 中同时添加 `<meta>` 标签确保响应式视口：

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

> Latest browser usually have it by default.

例：`bootstrap_demo.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Bootstrap Example</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <h1>Bootstrap Example</h1> <button class="btn btn-danger">Click me</button>
    </div>
  </body>
</html>
```

显示为：

去掉这行：

```
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
```

显示为：

Bootstrap 在 `<body>` 中通过各个元素的 `class` 属性来实现：

```
    <div class="container">
      <h1>Bootstrap Example</h1> <button class="btn btn-danger">Click me</button>
    </div>
```

注意，`class` 可以应用多个类，之间用空格隔开，如这里的 `btn btn-danger`. 因为不同的类用空格隔开，所以单个类名内不能添加空格.

如果引入 Bootstrap，只需要给元素添加需要的 `class` 属性既可，对应的 CSS style 会自动应用. 不需要自己再编写 CSS.

3.2 响应式布局

> Responsive layout

① 断点

> breakpoints

Bootstrap 定义了 6 个断点，用于根据设备或屏幕尺寸实现响应式布局. 通过在类名中添加断点中缀（Class infix）来实现响应式. 例如：

```
<nav class="navbar navbar-expand-lg"></nav>
```

表示导航栏只在屏幕宽度达到 “large” 断点时展开.

Breakpoint	Class infix	Dimensions
X-Small	*None*	<576px
Small	`sm`	≥576px
Medium	`md`	≥768px
Large	`lg`	≥992px
Extra large	`xl`	≥1200px
Extra extra large	`xxl`	≥1400px

Font sizes are by default responsive after enabling Bootstrap.

② 容器

> Containers

建议将内容放入容器（`div` with class `container`）以获得良好的内边距.

> `div` 是 HTML 自带的元素（division，分区），`container` 是 Bootstrap 里的 CSS 类.

容器是响应式的.

	Extra small <576px	**Small** ≥576px	**Medium** ≥768px	**Large** ≥992px	**X-Large** ≥1200px	**XX-Large** ≥1400px
`.container`	100%	540px	720px	960px	1140px	1320px
`.container-sm`	100%	540px	720px	960px	1140px	1320px
`.container-md`	100%	100%	720px	960px	1140px	1320px
`.container-lg`	100%	100%	100%	960px	1140px	1320px
`.container-xl`	100%	100%	100%	100%	1140px	1320px
`.container-xxl`	100%	100%	100%	100%	100%	1320px
`.container-fluid`	100%	100%	100%	100%	100%	100%

③ 网格系统

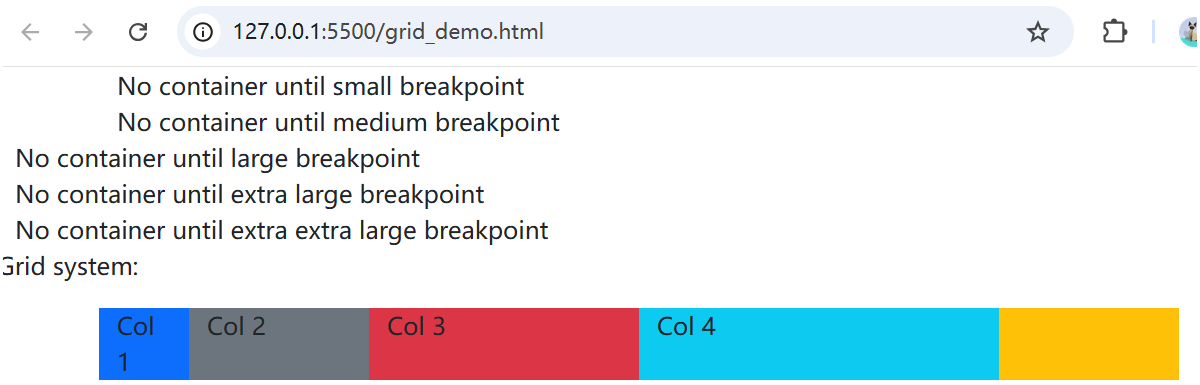
> Grid System

在 Bootstrap 中，一个容器总是被等分为 12 列. 你可以使用特定的类表示在给定屏幕尺寸下，容器应该占据多少

例：`grid_demo.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Bootstrap Example</title> <!-- Include Bootstrap CSS -->
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container-sm">
       No container until small breakpoint
    </div>
    <div class="container-md">
       No container until medium breakpoint
    </div>
    <div class="container-lg">
       No container until large breakpoint
    </div>
    <div class="container-xl">
       No container until extra large breakpoint
    </div>
    <div class="container-xxl">
       No container until extra extra large breakpoint
    </div>
    <p>Grid system:</p>
    <div class="container bg-warning">
      <div class="row">
        <div class="col-sm-1 col-xl-4 bg-primary">
           Col 1
        </div>
        <div class="col-sm-2 col-xl-3 bg-secondary">
           Col 2
        </div>
        <div class="col-sm-3 col-xl-2 bg-danger">
           Col 3
        </div>
        <div class="col-sm-4 col-xl-1 bg-info">
           Col 4
        </div>
      </div>
    </div>
  </body>
</html>
```

显示为：



注意，如果持续缩小，到屏幕比 576 pixels 还小，则没有达到任何断点，此时 Bootstrap 会应用自己的自动响应，把所有都铺满全屏宽度，然后从上到下排列.

④ Mobile-First 原则

Bootstrap 设计遵循移动设备优先的原则，即优先对最小的屏幕（移动设备）构建样式，然后随着屏幕尺寸增大逐步添加更复杂的布局.

因此，对于最小屏幕，在没有指定列宽类属性（如 `col-*`）时，列元素在 Extra Small 的屏幕上会默认占据完整的 12 列宽度.

No container until small breakpoint
No container until medium breakpoint
No container until large breakpoint
No container until extra large breakpoint
No container until extra extra large breakpoint

Grid system:

而上面的代码：

```
<div class="container bg-warning">
  <div class="row">
    <div class="col-sm-1 col-xl-4 bg-primary">
       Col 1
    </div>
    <div class="col-sm-2 col-xl-3 bg-secondary">
       Col 2
    </div>
    <div class="col-sm-3 col-xl-2 bg-danger">
       Col 3
    </div>
    <div class="col-sm-4 col-xl-1 bg-info">
       Col 4
    </div>
  </div>
</div>
```

因为这里只指定了 sm 和 xl，因此， 拉宽屏幕到 sm 时（≥ 576 px）才会生效第一批指定的列宽类.

	xs <576px	sm ≥576px	md ≥768px	lg ≥992px	xl ≥1200px	xxl ≥1400px
Container `max-width`	None (auto)	540px	720px	960px	1140px	1320px
Class prefix	`.col-`	`.col-sm-`	`.col-md-`	`.col-lg-`	`.col-xl-`	`.col-xxl-`
# of columns	12					
Gutter width	1.5rem (.75rem on left and right)					
Custom gutters	Yes					
Nestable	Yes					
Column ordering	Yes					

3.3 组件与工具

> Components & utilities
>
> https://getbootstrap.com/docs/5.1/components/accordion/

组件：Bootstrap 提供了大量即用组件来美化页面内容. 如：手风琴、警报、按钮、卡片、导航栏等.

大多数是纯 CSS，但有些需要 JS 或 Popper 库.

实用工具：提供了用于设置边框、颜色、间距、阴影等的实用工具类.

例如：`.border-start`（左边框）、`.text-warning`（黄色文本）、`.bg-success`（绿色背景）

例：`com.html`

```
<!DOCTYPE html>
<html lang="zh-Hans">
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 组件示例</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
</head>
<body>

    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <div class="container">
            <a class="navbar-brand" href="#">组件展示</a>
        </div>
    </nav>

    <div class="container mt-4">

        <h1 class="mb-4">Bootstrap 组件示例</h1>

        <div class="alert alert-success" role="alert">
```

```
            这是一个成功(Success)警告组件。
        </div>

        <h2 class="mt-5">按钮示例</h2>
        <p>
            <button type="button" class="btn btn-primary me-2">主要按钮 (.btn-
primary)</button>
            <button type="button" class="btn btn-warning">警告按钮 (.btn-warning)
</button>
        </p>

        <h2 class="mt-5">卡片示例 (Card)</h2>
        <div class="card" style="width: 18rem;">
            <div class="card-body">
                <h5 class="card-title">卡片标题</h5>
                <h6 class="card-subtitle mb-2 text-muted">卡片副标题</h6>
                <p class="card-text">
                    卡片是一个很好的内容组织工具。
                </p>
                <a href="#" class="card-link text-success">链接 (.text-success)
</a>
            </div>
        </div>

    </div>

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js
"></script>
</body>
</html>
```

显示为：

上例中，`class="container mt-4"` `class="mb-4"` `class="mt-5"` 是 Bootstrap 的 Utilities，主要用于控制布局和间距. 其中，

- `container` 是一个布局类. 它创建了一个响应式的容器，用于放置页面内容，并提供合适的左右内边距（padding）
- `mb-4` 和 `mt-4` 是间距工具类（Spacing Utilities），它们用于快速设置元素的外边距（margin）或内边距（padding）. 格式是 `{property}{sides}-{size}`. 例如，`mb-4` 表示 {margin}{bottom}-{4}，即给元素设置底部外边距，大小为 4

`navbar-brand` 是 Bootstrap 导航栏组件中的一个类. 它通常应用于导航栏中的 Logo 或网站名称.

`role` 属性是 HTML5 中 ARIA（Accessible Rich Internet Applications）标准的一部分. 它不影响视觉样式，但会向屏幕阅读器等辅助技术表明该元素的用途.

`type=“button“` 属性用于定义 `<button>` 元素的行为. 在 HTML 中，`<button` 元素有三种可能的 `type`：

- `type="submit"`：用于提交表单数据（默认）.
- `type=”button“`：一个普通的、没有默认行为的按钮. 通常与 JavaScript 配合，用于执行客户端脚本.
- `type="reset"`：用于重置表单.

`class="btn btn-primary me-2"` 中，`btn` 和 `btn-primary` 必须一起写，因为 `btn` 是一个基类，提供了所有按钮共享的基本样式（内边距、圆角、字体、默认鼠标悬停效果等）. 而 `btn-primary` 是一个修饰类，只负责设置按钮的颜色（通常是蓝色）.

`style="width: 18rem;"`，这里 `rem` 是 CSS 中的一个相对长度单位 Root EM（元素字体大小），1 rem 等于 HTML 根元素的字体大小. 在响应式设计中，通过设置 18 rem 可以按比例缩放页面上所有使用 rem 作为单位的元素（字体、宽度、边距等），从而更容易实现全局的缩放和可访问性.

3.4 其他功能

- 表格样式：可以轻松设置带状颜色或悬停效果的表格.
- 表单：用于定制表单控件的外观，并支持响应式断点设置.
- CSS reboot：修复浏览器之间的兼容性问题.
- 比例（Ratios）：使用伪元素来维持 HTML 元素的宽高比.
- Bootstrap 图标：可以额外引入图标字体样式表，使用 `<i class="bi bi-airplane"></i>` 等标签加载图标.

① 表单控件

> Form Controls
>
> 和组件的区别：组件是复杂的 Bootstrap 预先构建的 UI 结构，控件是相对简单的有收集用户输入和选择的功能的原生 HTML 元素.

例：`form.html`

```
<!DOCTYPE html>
<html lang="zh-Hans">
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 表单示例</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
</head>
<body>

    <div class="container mt-5">
        <h1>Bootstrap 表单控件定制</h1>

        <h2 class="mt-4">浮动标签 (.form-floating)</h2>
        <div class="form-floating mb-3">
            <input type="email" class="form-control" id="floatingInput"
placeholder="name@example.com">
            <label for="floatingInput">电子邮箱地址</label>
```

```
        </div>

        <h2 class="mt-4">表单组和文本提示 (.form-text)</h2>
        <div class="mb-3">
            <label for="inputPassword" class="form-label">密码</label>
            <input type="password" class="form-control" id="inputPassword">
            <div id="passwordHelpBlock" class="form-text">
                您的密码长度必须为 8-20 个字符，包含字母和数字。
            </div>
        </div>

        <h2 class="mt-4">范围滑块 (.form-range)</h2>
        <label for="customRange" class="form-label">选择等级 (1-10)</label>
        <input type="range" class="form-range" min="1" max="10" step="1"
id="customRange">

        <button type="submit" class="btn btn-primary mt-4">提交</button>
    </div>

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js
"></script>
</body>
</html>
```

显示为：

② 图标

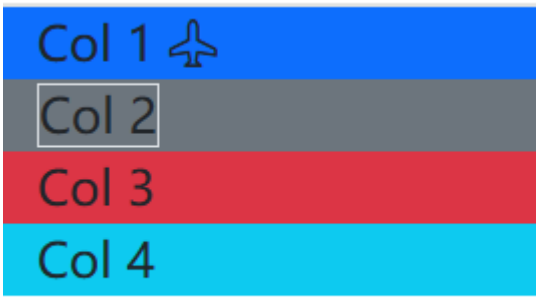
注意：Bootstrap 图标是独立于核心框架的，所以要使用图标需要额外加载一个 CSS（共 2 个）

```
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.9.1/font/bootstrap-icons.css">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
```

例：`icon_demo.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Bootstrap Example</title> <!-- Include Bootstrap CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.9.1/font/bootstrap-icons.css">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container bg-warning">
      <div class="row">
        <div class="col-sm-1 col-xl-4 bg-primary">
           Col 1 <i class="bi bi-airplane"></i>
        </div>
        <div class="col-sm-2 col-xl-3 bg-secondary"> <span class="border">Col
2</span>
        </div>
        <div class="col-sm-3 col-xl-2 bg-danger">
           Col 3
        </div>
        <div class="col-sm-4 col-xl-1 bg-info">
           Col 4
        </div>
      </div>
    </div>
  </body>
</html>
```

显示为：



4. 定制

> 待补充.

Lecture 4 CSS

> Cascading Style Sheets

1. 基础

1.1 基本知识

CSS 是层叠样式表（Cascading Style Sheets），它不是一种编程语言，而是用于美化和布局 HTML 内容的.

核心理念：旨在将内容（HTML）与表现/样式（CSS）分离.

优点：使得开发中内容与设计可由不同团队处理；易于改变网页外观；可在多个页面间共享样式表统一风格.

長度單位： 如 `px` (像素)、`%` (相對父元素)、`em` (相對當前字體大小)、`rem` (相對根元素字體大小)、`vh` (視口高度的 1%)、`vw` (視口寬度的 1%) .

顏色： 可用顏色名稱、十六進制 (#rrggbb)、或函數（如 `rgb()`, `hsl()`）表示.

1.2 使用

有三种在 HTML 中使用 CSS 的方式：

外部样式表：连接一个外部的 `.css` 文件，可在多个 HTML 文件间共享.

内部样式表：在 HTML 文件的 `<head>` 标签内使用 `<style>` 标签定义样式.

行内样式：直接在特定的 HTML 标签内使用 `style` 属性设定样式. 通常会覆盖外部和内部样式表.

语法：

- CSS 通常不区分大小写，但 HTML 属性值（如 `id="SomeName"`）除外
- 必须使用美式拼写 color
- 基本结构为 `Selector { property: value; property: value; }`. 其中 `Selector` 是开发者希望应用统一样式的 HTML 元素. 如 `h1`，`p`，`.highlight` 等. `property` 是属性，即开发者希望改变的样式属性，如 `color`，`font-size`，`line-height` 等. `value` 是值，即开发者为属性设定的具体值，例如 `blue`，`24px`，`2em` 等.
- 分号 `;` 用来分隔不同的样式宣告.

1.3 继承与层叠

继承（Inheritance）：子元素若未指定属性，会继承父元素的属性.

> 子元素即包含在父元素内部的元素，代码上有缩进.

层叠（Cascading）优先级：

- 行内样式 > 内部样式 > 外部样式
- 具体的规则覆盖通用的规则
- 后面的规则覆盖前面的规则
- 标记 `!important` 的属性会覆盖其他所有规则.

1.4 选择器

选择器用于选中要应用样式的 HTML 元素. 常见的选择器包括：

選擇器類型	範例	描述
元素選擇器	`p`	選中所有 `<p>` 元素
通配符選擇器	`*`	選中所有元素
ID 選擇器	`#example`	選中具備 `id="example"` 屬性的唯一 HTML 元素
類別選擇器	`.new`	選中所有具備 `class="new"` 屬性的 HTML 元素
偽類選擇器	`a:hover`	選中滑鼠游標懸停在 `<a>` 元素上時的狀態
偽元素選擇器	`p::first-letter`	選中所有 `<p>` 元素的第一個字母

1.5 布局与显示

元素类型：块级元素、行内元素、特殊元素

Display 属性：可更改元素的预设显示类型，例如将块级元素变为行内元素.

- `display: none;` 元素不显示且不占用任何空间.
- `visibility: hidden;` 元素不显示但仍占用其空间.

定位（Positioning）

> 待补充.

- `static`（预设）：所有 HTML 元素的预设位置
- `absolute`：使用 `top` 和 `left` 属性定义绝对位置
- `fixed`：相对于浏览器视窗定位.
- `relative`：相对于其原始静态位置定位.

1.6 盒模型

所有 HTML 元素都被视为盒子，包含以下属性：

- Height/width：内容区域的高度/宽度
- Padding（内边距）：内部空间，取内容的背景颜色.
- Border（边框）：包围盒子的线条.
- Margin（外边距）：外部空间，取父元素的背景颜色.
- 简写顺序：外边距或内边距的简写通常按**上、右、下、左**的顺序指定.

1.7 响应式设计

目标：确保网页能根据屏幕尺寸良好呈现.

视口设定：設置 `<meta name="viewport" content="width=device-width, initial-scale=1">` 以將視口寬度設為設備寬度，並將縮放設為 100%

图片处理：使用 `max-width: 100%; height: auto;` 使图片能随父元素缩放并保持长宽比.

1.8 预处理器

CSS 預處理器 (Preprocessors): 如 Sass 和 LESS 。它們讓網頁設計更有效率，提供變數、特殊選擇器等，最終被編譯成常規 CSS.

2. 进阶

2.1 变换

> CSS Transforms

CSS 变换可以在 X、Y 或 Z 轴上进行 2D 或 3D 转换.

变换包括平移（Translate）、旋转（Rotate）、缩放（Scale）、扭曲（Skew）

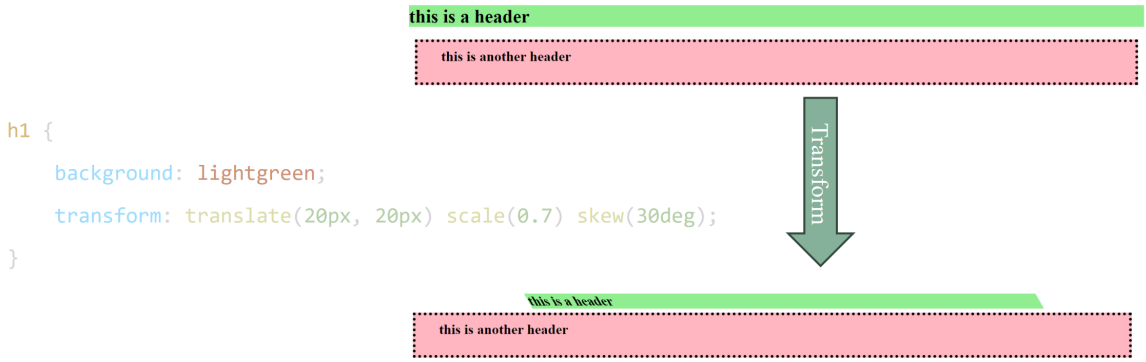
2.1.1 平移

以一定像素位移.

例:

```
h1 {
    background: lightgreen;
    transform: translate(20px, 20px) scale(0.7) skew(30deg);
}
```

> 这个 h1 是在 `<style>` 里的 Selector. 而 `<style>` 又是 `<head>` 的子元素. `<head>` 是 `<html>` 的子元素. 注意层级.



2.1.2 旋转

待补充

2.1.3 缩放

待补充

2.1.4 扭曲

待补充

2.2 过渡

> transition

语法:

```
#<some ID>:hover{
    property 1: result 1;
    property 2: result 2:
    ...
    property n: result n;
    transition: <property 1> <duration 1> <timing-function 1> <delay 1>,
<property 2> <duration 2> <timing-function 2> <delay 2>, ..., <property n>
<duration n> <timing-function n> <delay n>;
    }
```

> 整个放在 `<head>` 的 `<style>` 里.

`transition` 属性用于指定元素状态变化时的动画效果. 它指定以下内容:

- 属性（Property）：可以是元素的任何属性.

  > 只需要定义想要进行过渡的属性，保持不变的属性不写即可.

- 持续时间（Duration）：完成过渡所需的时间
- 计时函数（Timing-fucntion）

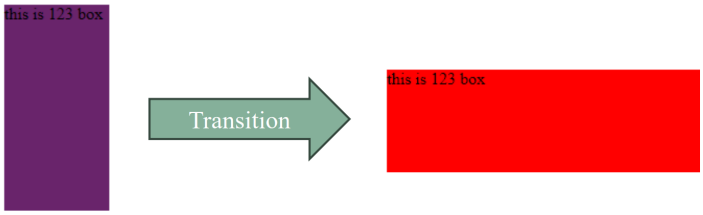
  `ease`：慢-快-慢

  `linear`：全程速度相同
- 延迟（Delay）：开始过渡前的等待时间

> 持续时间和延迟如果不定义，默认 0s. 延迟默认为 0s 常见，但是持续时间一般不为 0s，不然就等于没写 transition.

例：同时定义了某个元素的选择器和它的伪类选择器（悬停）

```
#box123 {
  background: #69246b;
  width: 100px;
  height: 200px;
}
#box123:hover{
  background: red;
  width: 300px;
  height: 100px;
  transition: background 1s ease 5s, width 2s linear 10s, height 3s ease-in 15s;
  /* property, duration, timing-function, delay */
}
```



注意：需要在 `<body>` 中应用该效果的元素里添加一个 `id=box123` 的属性. 例如：

```
<div id="box123">this is 123 box</div>
```

例：`transition.html`

```
<!DOCTYPE html>
<html>
<head>
    <title>Transition</title>
    <style>
        #box123 {
            background: #69246b;
            width: 100px;
            height: 200px;
        }
        #box123:hover{
            background: red;
            width: 300px;
            height: 100px;
            transition: background 1s ease 0s, width 2s linear 0s, height 3s
ease-in 0s; /* property, duration, timing-function, delay */
        }
    </style>
</head>
```

```
<body>
    <div id="box123">this is 123 box</div>
</body>
</html>
```

这里延迟手动设置为 0s，三个属性同时开始过渡，但是它们过渡时间不同，所以 background 最先结束，height 最后结束.

2.3 动画

> CSS Animations

除了 transition，更精细的动画可以使用 `@keyframes` 创建，它允许指定元素从 0% 到 100% 的行为变化

例：`animation.html`

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            #box123{
                background: blue;
                width: 50vw;
                height: 50vh;
                animation-name: my-animation;
                animation-duration: 10s;
            }
            @keyframes my-animation {
                0% {font-size: 10px; background: blue;}
                50% {font-size: 50px; background: green}
                100% {font-size: 100px; background: red}
            }
        </style>
    </head>
    <body>
        <div id="box123">test</div>
    </body>
</html>
```

`@keyframes my-animation`：定义动画关键帧.

`animation-name: my-animation;`：`animation-name` 是元素的一个属性，可以给它指定要绑定的 `@keyframes` 规则.

> 每个关键帧最后一个分号可写可不写，但最佳实践是写.

效果：`#box456` 会从打开页面开始的 10s 中，背景色由蓝变绿再变红，字体逐渐增大. 动画播放结束后回到初始状态. 但是如果题目说 "replicates the visual result of the CSS Animation"，则是要我们复制动画最后一刻的状态，不用考虑动画结束后.

2.4 无限滚动

> CSS trick: Infinite scrolling

目标：使用一张单一背景图像，水平重复滚动，看起来有无限连续效果.

`<body>` 中内容：

```
<div class="sliding-container">
    <div class="sliding-background"></div>
</div>
```

`<head>` 中定义两个类和一个关键帧动画：

```
            @keyframes slide{
                0% {
                    transform: translate(0);
                }
                100% {
                    transform: translate(-2139px);
                }
            }

            .sliding-container {
                width: 100%;
                height: 400px;
                overflow: hidden;
                position: relative;
            }

            .sliding-background {
                background: url("sliding.png") repeat-x;
                height: 400px;
                width: 300vw;
                font-size: large;
                color: pink;
                animation-name: slide;
                animation-duration: 10s;
                animation-timing-function: linear;
                animation-iteration-count: infinite;
            }
```

这里

```
animation-iteration-count: infinite;
```

是动画播放结束后重复播放，无限循环的意思.

首先，背景元素的宽度设置为大于屏幕尺寸.

```
width: 300vw;
```

三倍屏幕宽度.

> 至少为屏幕宽度 + 单张图片宽度.

然后，把背景图片重复平铺：

```
background: url("sliding.png") repeat-x;
```

接下来，设计一个平移变换：

```
@keyframes slide{
    0% {
        transform: translate(0);
    }
    100% {
        transform: translate(-2139px);
    }
}
```

为了实现完美的**无限滚动**，平移的距离（在 `@keyframes slide` 的 `100%` 处设置的 `translate` 值）必须**精确等于**你设置的背景图片（或平铺图案）的宽度.

原理：在一次平移后，右侧平铺的图片正好回到初始位置. 然后动画从头播放，无限循环，看起来就像没有停止地滚动.

例：`infinite_scrolling.html`

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            /* 关键动画定义：告诉浏览器如何移动元素 */
            @keyframes slide{
                0% {
                    transform: translate(0);
                }
                100% {
                    transform: translate(-2139px); /* 移动距离需根据图片和宽度调整 */
                }
            }

            .sliding-container {
                width: 100%;
                height: 400px;
```

```
                overflow: hidden;
                position: relative;
            }

            .sliding-background {
                background: url("sliding.png") repeat-x; /* 假设路径正确 */
                height: 400px;
                width: 300vw; /* 关键：设置宽度大于视口 */
                /* font-size 和 color 对背景滚动没有影响，但保留样式 */
                font-size: large;
                color: pink;
                animation-name: slide;
                animation-duration: 10s;
                animation-timing-function: linear;
                animation-iteration-count: infinite; /* 关键：无限循环 */
            }
        </style>
    </head>

    <body>
        <div class="sliding-container">
            <div class="sliding-background"></div>
        </div>
    </body>
</html>
```

2.5 逻辑函数

`min(...)`

- Pick the smallest among parameters

`max(...)`

- Pick the largest among parameters

`clamp(...)` 取中间值

- To ensure value is between min and max, based on ideal
- 例如 `width: clamp(30px, 50vw, 100px)`，默认 50 vw. 若 50vw 小于 30px，则使用 30px，若 50vw 大于 100px，则使用 100px.

2.6 显示进阶：Flexbox

We discussed the display options of inline and block in CSCI2720.

还有更多的 display options：

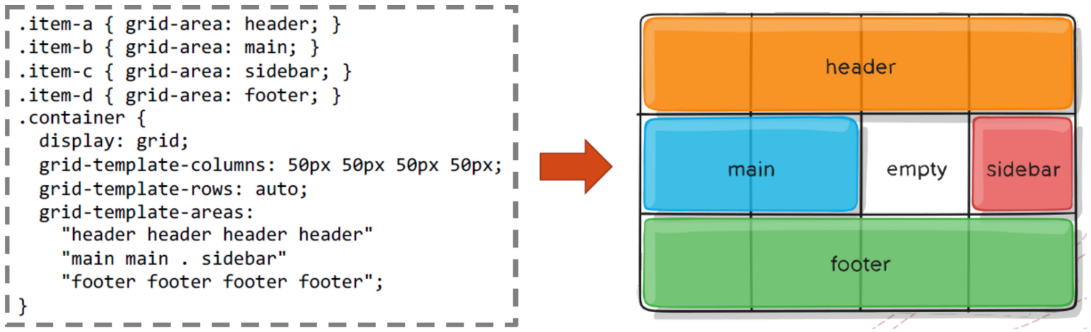
`display: flex`

- 内容根据 `flex-direction` 流动
- 常用于垂直居中

> 待补充

2.7 布局进阶：Grid

Similar to the grid system in Bootstrap, but customizable.



所有 `class="header"` 的元素都会被分配到第一行的四列空间，如果有两个元素被分配到同一个空间，它们都会从左往右排，也就是会重合.

> 解决方案：待补充.

Lecture 5 JavaScript & TypeScript

1. JavaScript

特点：

Interpreted, or just-in-time compiled language

Single-threaded

Multi-paradigm: object-oriented, imperative, functional

和 Java 是不同语言，语法有些类似.

但也有语法区别：

- 变量声明：JS 使用 `var`，`let`，`const`，而 Java 使用类型关键字（`int`，`String`）
- 变量类型：JS 是动态类型，变量可以存储不同类型的数据，Java 是静态类型，需要明确的类型声明.
- 面向对象：Java 是严格的面向对象，JS 是多范式，支持对象导向、命令式和函数式编程.

浏览器中的每一个元素都可以用 JS 创建和操作. 除了浏览器，也可以用于设置 Web 服务器等.

功能：表单验证、网页交互、服务器通信等.

限制：

- JS 在浏览器运行，受浏览器运行环境约束；
- 除非用户明确选择，否则不能直接访问文件系统或内存.
- 通信限制：只能通过浏览器端口或协议（HTTP/HTTPS）进行通信.

JavaScript “lives” in the browser.

JavaScript is bounded by the browser runtime environment.

No direct file system or memory access

Except when user explicitly selects a file to open

No access to hardware devices unless explicitly granted

Can only communicate over browser ports or protocols, e.g., HTTP/HTTPS

We only discuss client-side JavaScript in this lecture

本课仅介绍客户端 JavaScript.

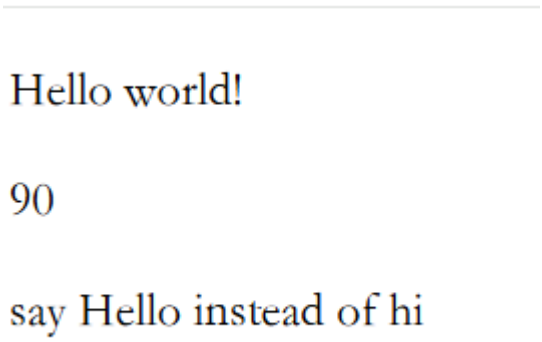
例：`js_demo.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>JS Example</title>
</head>
<body>


<!-- Embedding code in HTML -->
<p>
  <script>
    document.write("Hello world!");
  </script>
</p>
```

```
<p>
  <script>
    let x = 10;
    const y = 20;
    var z = 30;
    x = x + 30;
    document.write(x+y+z);
  </script>
</p>


<p>
  <script>
    var greeter = "hi";
    var times = 4;

    if (times > 3) {
        var greeter = "say Hello instead of hi"; // var can be redeclared
    }
    document.write(greeter) // "say Hello instead"
  </script>
</p>


<script>
  // Declaring and initializing a boolean variable
  let isLogged = false;

  // Simulating a login process
  function login() {
    // Perform login validation and set the boolean variable accordingly
    isLogged = true;
  }

  // Checking the login status
  function checkLoginStatus() {
    if (isLogged) {
      console.log("User is logged in.");
    } else {
      console.log("User is not logged in.");
    }
  }


  // Calling the functions
  checkLoginStatus(); // Output: User is not logged in.
  login();
  checkLoginStatus(); // Output: User is logged in.
</script>

<script>
  // using regular expression to check valid email
  function validateEmail(email) {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/ ;
```

```
    return regex.test(email); // test()  method is used to check if th egiven
email matches the regex
  }

  let email1 = "example@example.com";
  let email2 = "invalid_email";
  let email3 = "colintsang@cuhk.edu.hk";
  let email4 = "abc def@example.com";

  console.log(validateEmail(email1));
  console.log(validateEmail(email2));
  console.log(validateEmail(email3));
  console.log(validateEmail(email4));
</script>


</body>
</html>
```

显示为：



1.1 使用

有两种方式执行 JS 代码：

连接到 `.js`

在 HTML 的 `<head>` 中使用

```
<script src="myscriptfile.js"></script>
```

优点：易于分离维护.

嵌入 HTML：通常放在 `<body>` 的底部，即 `</body>` 之前.

```
<script>
    document.write("Hello world!");
</script>
```

1.2 调试和输出

可以在主流浏览器的 JS Console 中调试.

> Ctrl + Shift + J
>
> 或 F12 - Console

向控制台输出：`console.log(...)`

向警告框输出：`window.alert(...)`

1.3 变量与数据类型

标识符（Identifiers）：区分大小写，不能以数字开头.

变量（Variable）：

- `var`：旧式，可更新和重新声明
- `let` 首选，可更新但不能重新声明
- `const`：不能更新或重新声明

数据类型:

- string: 文本数据，由单引号或双引号括起
- number: 双精度 64 位，包括 infinity 和 NaN
- bigint: 任意大的数字
- boolean：true or false
- undefined: 变量刚声明时自动赋值

除了 `0`、`null`、`false`、`NaN`、`undefined` 和 `""` 之外的任何值都为 `true`

JS 是 **动态类型**（dynamically typed）语言，同一个变量可以存储不同类型的数据 。可以使用 `typeof` 来检查数据类型

```
> let x
< undefined
> typeof x
< "undefined"
> typeof 123
< "number"
> typeof "hello world"
< "string"
```

类型转换

When adding a number and a string, the number is treated as a string

```
> let x = 1 + 2 + 3
< undefined
> x
< 6
> let y = 1 + '2' + 3
< undefined
> y
< "123"
```

> 执行变量声明或赋值语句时，没有显示的返回值，控制台默认输出 undefined.

强制类型转化

`Number()`

`String()`

`Boolean()`

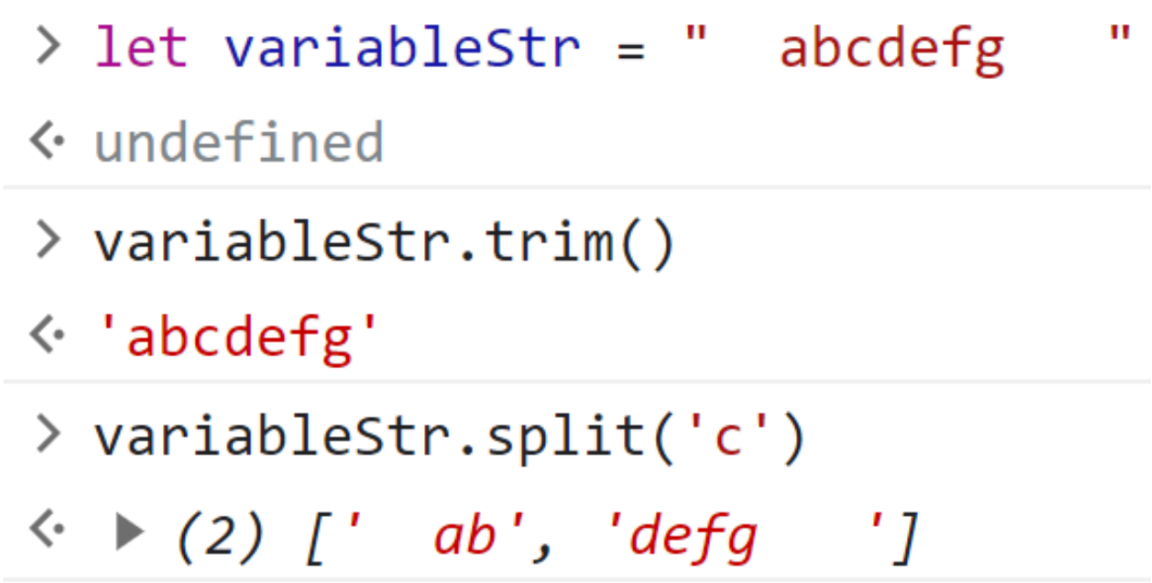
1.4 字符串

单引号或双引号括住. 可以直接用 < > == 比较（按字典序）.

String characters can be accessed like array contents.

```
let stringVar = "abcde";
// stringVar[0] is "a"
```

字符串方法



正则表达式

验证一个字符串是否符合电子邮件地址的基本格式：

```
const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

`/ /`：用于创建正则表达式

`^`：断言字符串的起始位置

`[^\s@]+`：匹配一个或多个字符.

- `[^abc]` matches any single characters that is not a, b, or c.

- `[^abc]+` matches one or more characters that is not a, b, or c.
- 这里的意思是，匹配一个或多个不是空白符（/s）或 @ 符号的字符.

`@`：匹配 @

`\.`：匹配点

> 注意，匹配点要额外的反斜杠，因为点本身是一个特殊字符，含义是匹配任何单个字符. 反斜杠是用来转义的.

`$`：断言字符串的结束位置

字符串插值

> String Interpolation

字符串插值是一种将变量或表达式的值嵌入（或"插入"）到字符串字面量中的技术

传统方法：使用 `+` 运算符（类似其他语言）

```
const a = 5;
const b = 10;
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
```

结果：

```
"Fifteen is 15 and
not 20."
```

现代方法：使用模板字面量（Template Literals）

```
const a = 5;
const b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

- **要点：**
  - 字符串必须使用 **反引号** (```) 括起来，而不是单引号或双引号
  - 变量或表达式必须放在 `${}` 结构中，JavaScript 会自动计算并插入它们的值
  - 模板字面量支持**多行**. 你可以在反引号内直接输入换行符，而不需要使用 `\n`

1.5 运算符

`===` 相等比较

`!==` 值或类型不相等

Operators

- Arithmetic operators
  - +, -, *, **, /, %, ++, --
- Assignment operators
  - =, +=, -=, *=, /=, %=, **=
- Comparison operators
  - ==, !=, >, <, >-, <=
  - === : equal value and type
  - !== : unequal value or type
- Logical operators
  - &&, ||, !
- Bitwise operators
  - &, |
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators

1.6 数组

> array

```
> let CSE = ['CSCI', 'CENG'];
< undefined
> CSE[2] = 'AIST';
< "AIST"
> CSE
< ["CSCI", "CENG", "AIST"] (3)
> CSE.length;
< 3
```

An array is a list of items, which can be of mixed types

Index starts at 0

An array item can be another array

The array has a type of **object**

```
> typeof NaN
< "number"
> 9999999999999999
< 10000000000000000
> 0.5+0.1==0.6
< true
> 0.1+0.2==0.3
< false
> Math.max()
< -Infinity
> Math.min()
< Infinity
> []+[]
< ""
> []+{}
< "[object Object]"
> {}+[]
< 0
> true+true+true===3
< true
> true-true
< 0
```

```
> true==1
< true
> true===1
< false
> (!+[]+[]+![]).length
< 9
> 9+"1"
< "91"
> 91-"1"
< 90
> []==0
< true
```

条件语句 & 循环语句

同 C 和 Java

The *for...of* loops iterating in arrays, strings, maps, NodeLists, etc.

```
> let cars = ["honda", "toyota",  "nissan"];
< undefined
> let x;
< undefined
> for (x of cars) {
      console.log(x);
  }
  honda
  toyota
  nissan
< undefined
```

The *for...in* loop iterating in object properties with key-value pairs

```
> let person = {fname:"John", lname:"Doe", age:20};
< undefined
> let text = "";
< undefined
> let x;
< undefined
> for (x in person){
      text += person[x];
  }
< 'JohnDoe20'
```

1.7 函数

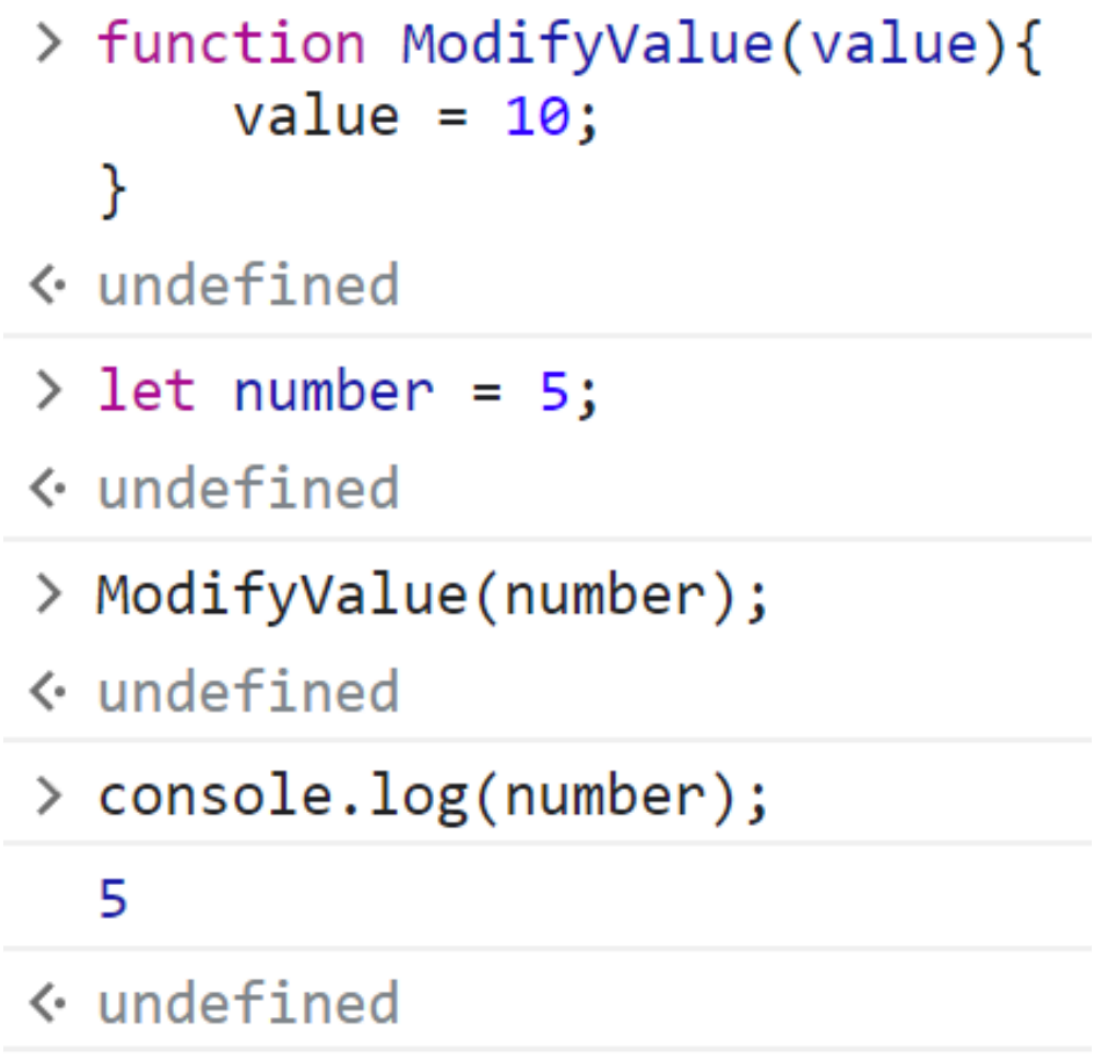
关键字 `function` 声明

```
function myFunction(a,b) {
    return a*b;
}
```

匿名函数：直接存在变量中. 变量相当于昵称

```
let x = function (a,b) {return a*b};
let z = x(4,3);
```

按值传参，函数内部修改不影响原始变量



箭头函数

```
hello = function() {
    return "hello world";
}

hello = () => {
    return "hello world";
}

hello = () => "hello world";
```

如果执行任务只有一个 return statement，可以进一步简化.

1.8 浏览器窗口

In every browser, there is a window **object** representing the browser's window

All global JS variables, objects, and functions are member of window

Variables: window properties

Functions: window methods

Some window objects:

- window.screen: widthm, heitgh, pixelDepth, etc.
- window.location: hostname, protocol, etc.

1.9 Popup boxes

Messages to user can be delivered with JS popup boxes

Alert box: `window.alert(message)`

Confirm box: `window.confirm(message)`

return true for OK, and false for cancel or anything else

Prompt box: `window.prompt(message)`

return the string of user input

These boxes are browser dependent, and not CSS skinnable.

2. TypeScript

JavaScript 的超集.

执行方式：Web 浏览器只能执行 JavaScript. 因此，TypeScript 代码在浏览器或服务器上执行之前，必须先编译（转换）成纯 JavaScript.

优势：TypeScript 编译器（tsc）能帮助开发人员检查类型错误和其他问题. 在早期开发阶段捕获错误非常有用.

工具：使用 `npm` 安装 `tsc` 编译器.

下面介绍 TypeScript 的核心特性.

2.1 可选类型注解

- JavaScript 采用动态类型，数据类型在代码执行时确定，不能显式声明变量类型.
- TypeScript 允许使用类型注解来声明变量类型，例如 `let x: number`

类型检查：如果在 TS 中将一个 `string` 类型的值赋给一个被注解为 `number` 的变量，编译器会报错.

可选性：类型注解是可选的，省略注解仍然是有效的 TS.

`any` 类型：如果需要数据类型的灵活性，可以使用特殊的 `any` 类型. 如果未指定类型且 TS 无法从上下文中推断出类型，则类型默认为 `any`.

2.2 数组

TS 的数组可以要求所有元素都是相同类型的.

例如：

```
let a1: number[] = [5, 4, 3, 2, 1];
```

如果需要混合数据类型，可以使用联合类型（Union Type）

例如：

```
let a3: (string|number)[] = [1, "two", 3, "four"];
```

2.3 函数

> 待补充.

2.4 对象和接口

> 待补充.

2.5 自定义类型

待补充

2.6 安装和设置

待补充

Lecture 6 Form

> 表单

本讲座主要介绍了 HTML 表单的使用、控制元素、提交方式、CSS 样式以及验证

1. 表单的用途和结构

- HTML 表单允许用户提供输入
- 用户输入可用于**提交回服务器**或与**脚本交互**
- 整个表单必须包含在 `<form>` 元素内

2. 表单控制元素

- **文本输入字段**:
  - `<input type="text">`：单行输入
  - `<input type="password">`：密码输入
  - `<textarea>`：多行文本输入，可以包含初始值
- **新输入控件**（提供验证或特殊效果）：
  - `<input type="email">`：确保输入是电子邮件地址
  - `<input type="tel">`：在移动设备上调用数字键盘
  - `<input type="url">`：确保输入是正确的 URL
  - `<input type="color">`：显示颜色选择器
  - `<input type="date">`：显示日期选择器
- **选项列表**:
  - `<input type="chatbox">`：复选框
  - `<input type="radio">`：单选按钮，通过 `name` 属性分组，只允许选择一个选项
  - `<select>` 和 `<option>`：创建用户可选择的列表
- **表单分组**：使用 `<fieldset>` 组合项目，并使用 `<legend>` 显示组标题

3. 输入属性

- `value`：初始值
- `readonly`：字段只读 85]。
- `disabled`：字段不可用 86]。
- `required`：字段必须填写 87]。
- `autofocus`：页面加载时字段获得焦点 88]。

4. 表单标签 (`<label>`)

- 用于定义表单元素的标题 92]。
- 通过 `for` 属性与控制元素的 `id` 属性关联 93, 94]。
- 这有助于浏览器更轻松地选择控件，并使屏幕阅读器正确理解关系 95, 96]。

5. 表单按钮和操作

- **简单按钮**：`<button type="button">` 113]。
- **提交按钮**：`<button type="submit">`，默认运行表单操作，将数据发送到服务器脚本 113]。
- **重置按钮**：`<button type="reset">`，清除并恢复表单中的所有输入控件 113]。

6. 表单提交

- **传统方式**：将数据提交给服务器端脚本 (e.g., PHP/Node.js) 114]。
  - 数据以名称-值对的形式发送 114]。
  - **GET** 方法：数据编码到 URL 作为查询字符串；数据量有限 (~2KB)；URL 可被书签保存和在历史记录中可见，存在**安全问题** 114, 115]。
  - **POST** 方法：数据嵌入到 HTTP 请求主体；数据大小受主体大小限制 (~1MB 到 2GB) 114, 115]。
- **现代方式**：在客户端使用 JavaScript 进行预处理，并可进行异步提交（无刷新）和表单验证 116]。

7. 表单样式与验证

- **CSS**：可以使用属性选择器 `input[type="..."]` 和伪类 (e.g., `:hover`, `:required`, `:valid`) 对表单控件进行样式设置 117]。
- **验证**：HTML 表单应在服务器端处理前进行验证 118]。
  - 确保用户输入正确数据、防止脚本崩溃、减轻处理脚本的工作量 118]。
  - 通常通过 **JavaScript** 处理 118]。
  - CSS 伪类 `:valid` 和 `:invalid` 可用于验证后的样式设置 118]。

实验 4: 表单与 Fetch 总结

本实验旨在通过实际操作，让学生使用 **Bootstrap** 样式改进 HTML 表单，并通过 **JavaScript Fetch API** 实现客户端数据处理、加载和保存到服务器

1. 任务 1: 改进表单

- 为评论表单添加更多颜色选项的**单选按钮** (`<input type="radio">`)
- 要求重复使用不同的 `id` 但相同的 `name` 属性来实现单选分组

2. 任务 2: 设置 JavaScript 事件处理程序

- 为 "Add comment" 按钮设置 `onclick` 事件，链接到 `processform()` 函数
- `processform()` 函数的任务是：
  - 从表单输入中获取新评论的**电子邮件**、**评论内容**和**颜色选择**
  - 动态创建新的 HTML 元素（包括带有 SVG 圆圈的 `div`）来显示评论
  - 将新评论的**内容**、**类名**和 SVG **填充颜色**设置好
  - 将新评论添加到 `#comments` 容器的顶部
  - 最后，使用 `form.reset()` 清空表单内容

3. 任务 3: 使用 JS Fetch 加载数据

- 创建 "Load File" 按钮，并链接到 `loadfile()` 函数 230, 231]。
- **本地开发问题**：Google Chrome 的安全设置不允许 `fetch()` 本地文件，需要使用**本地 Web 服务器** 235, 236, 237]。
  - 推荐使用 **Simple Web Server**，并配置允许文件上传和设置 CORS 标头 239, 241, 242]。
  - 必须通过服务器地址 (e.g., `http://localhost:8080/`) 访问 HTML 文件 247, 248]。
- `loadfile()` **函数**：使用 `fetch("lab_04_file.txt")` 读取文件内容，然后将获取到的文本内容显示到评论输入框 (`#new-comment`) 中 252]。

4. 任务 4: 使用 JS Fetch 写入数据

- 创建一个 "Save File" 按钮，当点击时将评论上传到服务器 261]。
- **PUT 方法**：使用 `fetch()` 配合 `method: 'PUT'` 可以直接上传数据而不需要服务器脚本 258, 263]。
- 提示需要使用 `.value` 属性从 HTML 元素中提取数据，并将其放入 `fetch()` 请求的 `body` 中 262, 263]。

Lecture 7 异步

> Callback函数/Promise对象和 .then 方法/Fetch API/async 和 await 关键字

1. 异步 JS

> Asynchrony in JS

JavaScript 是单线程的（single-threaded），一次只能执行一个操作. 执行序列由调用栈（Call Stack）决定，遵循后进先出（Last-In-First-Out）的原则.

阻塞问题：如果某个操作耗时较长，其他所有操作都会被阻塞.

解决方案：**异步编程**允许启动任务后无需等待其完成. 当任务完成时，会触发某些后续操作（Events、callbacks、Promises...）

异步机制：

回调队列（Callback queue）：也称为事件队列或消息队列，用于存放已调度好、在特定条件满足时执行的函数.

事件循环（Event loop）：JS 引擎中的机制，持续检查调用栈是否为空，并将队列中的事件移入调用栈.

调度方法：可以使用 `setTimeout()`（延迟执行一次）和 `setInterval()`（重复间隔执行）来安排函数在稍后执行.

例：`timeout.html`

```
<!DOCTYPE html>

<html>
    <head>
        <title>延时执行</title>
    </head>
    <body>
        <p>1+1=</p>
        <script>
            setTimeout( () => console.log("2"), 5000);
        </script>
    </body>
```

网页打开后 5000ms（五秒）后在 concole 输出 2.

这里箭头函数：

```
setTimeout( () => console.log("2"), 5000);
```

是传统函数的简洁写法. 传统函数：

```
setTimeout(function(){console.log("2")}, 5000)
```

例：`clock.html`

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>setTimeout 和 setInterval 示例</title>
    <style>
        body { font-family: Arial, sans-serif; text-align: center; margin-top:
50px; }
        .container { border: 1px solid #ccc; padding: 20px; border-radius: 8px;
max-width: 500px; margin: 0 auto; }
        button { padding: 10px 20px; margin: 5px; cursor: pointer; }
        #timer-output { font-size: 2em; margin-bottom: 20px; }
        #message-output { color: green; font-weight: bold; }
    </style>
</head>
<body>

    <div class="container">
        <h3>间隔计时器 (setInterval)</h3>
        <div id="timer-output">0</div>
        <button onclick="startInterval()">开始间隔</button>
        <button onclick="stopInterval()">停止间隔 (clearInterval)</button>

        <hr>

        <h3>延迟执行 (setTimeout)</h3>
        <button onclick="scheduleTimeout()">设置 3 秒后执行</button>
        <button onclick="clearScheduledTimeout()">取消延迟 (clearTimeout)</button>
        <p id="message-output">等待消息...</p>
    </div>

    <script>
        // --- 1. setInterval 和 clearInterval ---
        let intervalId = null;
        let count = 0;

        /**
         * 开始间隔执行
         */
        function startInterval() {
            if (intervalId === null) {
                // setInterval(): 每 1000 毫秒 (1 秒) 运行一次函数
                intervalId = setInterval(function() {
                    count++;
                    document.getElementById('timer-output').textContent = count;
                    console.log(`Interval running: ${count}`);
                }, 1000);
                document.getElementById('timer-output').style.color = 'blue';
            }
        }

        /**
         * 停止间隔执行
         */
```

```
        function stopInterval() {
            if (intervalId !== null) {
                // clearInterval(): 停止 setInterval() 设置的重复执行
                clearInterval(intervalId);
                intervalId = null;
                console.log('Interval stopped.');
                document.getElementById('timer-output').style.color = 'red';
            }
        }

        // --- 2. setTimeout 和 clearTimeout ---
        let timeoutId = null;

        /**
         * 安排一次性延迟执行
         */
        function scheduleTimeout() {
            // 清理之前的延迟，以防重复点击
            clearScheduledTimeout();

            document.getElementById('message-output').textContent = '已设置延迟...
3秒后出现消息。';

            // setTimeout(): 在 3000 毫秒 (3 秒) 后运行一次函数
            timeoutId = setTimeout(function() {
                document.getElementById('message-output').textContent =
'“Hello”消息成功出现！';
                console.log('Timeout executed.');
                timeoutId = null; // 执行后清空ID
            }, 3000);
        }

        /**
         * 取消延迟执行
         */
        function clearScheduledTimeout() {
            if (timeoutId !== null) {
                // clearTimeout(): 取消 setTimeout() 设置的延迟执行
                clearTimeout(timeoutId);
                timeoutId = null;
                document.getElementById('message-output').textContent = '延迟执行已
取消。';
                console.log('Timeout cleared.');
            }
        }
    </script>

</body>
</html>
```

原理：当你调用 `setInteval()` 时，它会启动一个持续运行的循环计时器，为了以后能停止这个循环，`setInteval()` 会返回一个特殊的 ID 号. 我们用 `intervalId` 作为储存这个 ID 号的变量.

在需要停止时，把这个 ID 传递给 `clearInterval()` 即可.

同理，`setTimeout()` 也会返回一个特殊 ID 号，我们用 `timeooutId` 来保存.

注意，`setInterval()` 和 `setTimeout()` 都在被调用的瞬间返回值. 但是执行会延后，一个以设定时间重复执行，一个等到设定时间后执行一次.

在执行前，

1.1 回调函数

> Callback functions

定义：当一个函数作为另一个函数的参数传递，并在稍后被调用时，它被称为回调函数. 它会在调用函数完成时被执行.

回调地狱（Callback hell）：通过链式调用多个回调函数可以实现多次等待，但会导致代码结构混乱，难以阅读和维护.

上面的例子中，`setTimeout(function, delay)` 和 `setInteval(function, inteval)` 的第一个参数，就是回调函数. 这个函数会被存储起来，等到外部函数完成主要工作后，才会被调用执行. 在等待过程中，不仅回调函数不占用主调用栈，`setTimeout()` 函数本身也不占用（只在被调用的瞬间占用了调用栈，返回一个 ID 并将回调函数和延迟时间交给浏览器或 Node.js，时间成本可以忽略不计）.

当等待时间结束，如果此时主线程恰好在执行其他任务，回调函数的任务会进入回调队列中排队，当主线程结束当前任务后，队列中等待最久（队首）的任务会被推入调用栈（主线程）执行.

> 所以设定的延迟时间只是一个最小等待时间，而不是一个保证执行时间. 所以计时器运行久了很可能不准.

2. Promise

目的：Promise 对象代表异步执行的结果，旨在解决回调地狱问题.

三种状态：

- pending（待定）：初始状态
- fulfilled（已实现）：任务成功完成，可以获取结果值.
- rejected（已拒绝）：任务失败，可以找到错误对象.

使用方式：通过 Promise 的 `then()` 方法来使用它，该方法接受成功（success）和失败（failure）两个回调函数（可选）.

2.1 对象声明

```
const myPromise = new Promise((resolve, reject) => {
    // 启动异步任务...

    setTimeout(() => {
        const success = Math.random() > 0.5;

        if (success) {
            resolve("操作完成，这是数据");
        } else {
            reject(new Error("操作失败"));
        }
    }, 1000);
});
```

解释:

- `new Promise(...)` 是构造函数，创建并返回一个新的 Promise 对象. 默认状态为 Pending.
- `function(resolve, reject) {...}` （上面采用的是箭头写法，等价）是执行器函数，它在 Promise 对象创建时立即执行，接受两个函数参数（我们只需要定义名称，不需要定义函数. 这两个函数是 JavaScript 引擎自动定义和提供的）：
    - `resolve()` 和 `reject()` 可以接受任意类型的值，但是通常 `resolve(value)` 和 `reject(reason)`，其中 `value` 是异步操作成功是返回的东西，可以是字符串或者数字. `reason` 通常是一个 `Error` 对象，说明错误原因. 也可以什么都不传，直接调用 `resolve()` 或 `reject()`
    - 如果异步任务结束还没有调用任何一个，那这个 Promise 对象的状态就永远处于 pending 了.

resolve 和reject通常是在 Promise 处理的异步任务中，根据任务的结果被调用. 注意 Promise 对象的状态不可逆转，也就是只有第一次调用会改变它的状态，之后再次调用也无效了.

例:

```
let p = new Promise ( (resolve, reject ) => {
    let a = 1+1
    if (a==2){
        resolve('sucess')
    } else {
        reject('failed')
    }
})

p.then((message)=>{
    console.log('this is the then '+message)
}).catch((message)=>{
    console.log('this is the catch '+message)
})
```

2.2 `.then()` 方法

Promise 对象可以使用 .then 方法. 语法:

```
myPromise.then(
    (value) => {...},
    (error) => {...}
)
```

.then 方法接受两个函数参数，其中第二个可选. 第一个函数接受的参数 value 为 myPromise 变为 fulfilled 时调用 resolve 传回的参数；第二个函数接受的参数 `error` 为 myPromise 变为 rejected 时调用 reject 传回的参数. 因为 myPromise 在调用 .then 方法时只可能有一种状态，若两个函数都定义了，则 fulfilled 时调用第一个，rejected 时调用第二个；若只定义了一个函数，则 fulfilled 时调用第一个，rejected 时不进行任何操作.

如果 .then 方法只接受第一个参数，那么假设 Promise 是rejected，则 .then 方法不会进行任何操作.

.then 方法内部细分为多种情况:

- 没有返回东西：then 方法返回一个以 undefined 为值的 fulfilled 的新 Promise 对象.
- 任务返回了一个非 Promise 对象（如普通值），then 方法会返回一个以该值为 resolve 值的 fulfilled 新 Promise.
- 任务返回了一个 Promise 对象：then 方法会返回一个新的 Promise 对象，其状态和值同任务返回的 Promise 对象（但不是同一个）

例：`promise.html`

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Promise 状态与 .then() 示例</title>
    <style>
        body { font-family: Arial, sans-serif; text-align: center; margin-top:
50px; }
        .container { border: 1px solid #ccc; padding: 20px; border-radius: 8px;
max-width: 600px; margin: 0 auto; }
        button { padding: 10px 20px; margin: 10px; cursor: pointer; }
        .status-box {
            margin-top: 20px;
            padding: 15px;
            border: 2px solid #333;
            min-height: 50px;
            text-align: left;
        }
        .pending { background-color: #fffacd; border-color: orange; }
        .fulfilled { background-color: #e6ffe6; border-color: green; }
```

```
        .rejected { background-color: #ffe6e6; border-color: red; }
    </style>
</head>
<body>

    <div class="container">
        <h2>Promise 状态演示</h2>
        <p>点击按钮模拟异步操作（随机成功或失败）</p>

        <button onclick="runAsyncOperation(true)">模拟成功 (Resolve)</button>
        <button onclick="runAsyncOperation(false)">模拟失败 (Reject)</button>

        <div id="promise-output" class="status-box pending">
            <strong>当前状态:</strong> <span id="status-text">初始状态 (Pending)
</span>
            <hr>
            <strong>结果/错误:</strong> <span id="result-text">等待执行...</span>
        </div>
    </div>

    <script>
        // 获取输出元素
        const outputDiv = document.getElementById('promise-output');
        const statusText = document.getElementById('status-text');
        const resultText = document.getElementById('result-text');

        /**
         * 重置显示状态为 Pending
         */
        function resetDisplay() {
            outputDiv.className = 'status-box pending';
            statusText.textContent = '初始状态 (Pending)';
            resultText.textContent = '操作进行中...';
            console.log('Promise: Pending...');
        }

        /**
         * 模拟异步操作并返回一个 Promise
         * @param {boolean} shouldSucceed - 决定 Promise 应该成功 (true) 还是失败
(false)
         */
        function createAndRunPromise(shouldSucceed) {
            // Promise 对象代表异步执行的结果  63]
            return new Promise((resolve, reject) => {
                // 模拟耗时操作，例如 2 秒延迟
                setTimeout(() => {
                    if (shouldSucceed) {
                        // 任务成功完成 (fulfilled) -> 返回结果值
                        resolve("数据成功获取！用户 ID: 12345");
                    } else {
                        // 任务失败 (rejected) -> 返回错误对象
                        reject(new Error("网络连接失败，请检查您的设置。"));
                    }
                }, 2000);
            });
        }
```

```
        /**
         * 执行 Promise 并使用 .then() 处理结果
         * @param {boolean} succeed - 是否应该成功
         */
        function runAsyncOperation(succeed) {
            resetDisplay();

            const myPromise = createAndRunPromise(succeed);

            // 使用 then() 方法来使用 Promise  69]
            myPromise.then(
                // 1. 成功回调 (Success Callback)
                (value) => {
                    // Promise 状态：Fulfilled
                    outputDiv.className = 'status-box fulfilled';
                    statusText.textContent = '已实现 (Fulfilled)';
                    resultText.textContent = value;
                    console.log('Promise: Fulfilled. Value:', value);
                },
                // 2. 失败回调 (Failure Callback)
                (error) => {
                    // Promise 状态：Rejected
                    outputDiv.className = 'status-box rejected';
                    statusText.textContent = '已拒绝 (Rejected)';
                    resultText.textContent = '错误: ' + error.message;
                    console.error('Promise: Rejected. Error:', error);
                }
            );
        }
    </script>

</body>
</html>
```

现代写法：

```
myPromise
    .then((value) => {
        // 成功时执行的代码
        console.log("Success:", value);
        // 这里的成功回调是必须的
    })
    .catch((error) => {
        // 失败时执行的代码
        console.log("Error:", error);
        // 这里的失败处理是可选的，但强烈推荐
    });
```

只使用 .then 方法的第一个参数，如果是错误留给 .catch 方法处理.

链式调用（Promise chain）：Promise 支持链式调用，可以使复杂的异步操作序列结构更加清晰.

2.3 强制类型转换

Promise 对象在和数字相加时，会自动调用自己的 toString 方法强制转换为字符串，然后数值和字符串相加数值也会强制转化为字符串，最后的结果就是字符串拼接.

例:

```
<!DOCTYPE html>

<html>

<body>
    <script>
        function waitfornum(x, delay){
            return new Promise((resolve, reject) => {
                setTimeout(()=>{
                    console.log("Waiting for 2 seconds, with x="+x);
                    resolve(x);
                }, delay);
            })
        }

        let add = async x => {
            let a = waitfornum(10, 2000);
            let b = waitfornum(20, 2000);
            return x + await a + b;
        }

        add(5).then(sum => console.log(sum))
    </script>
</body>

</html>
```

若 add return 的 b 前有 await，则控制台输出:

```
Waiting for 2 seconds, with x = 10
Waiting for 2 seconds, with x = 20
35
```

若如例中所示，b 前没有 await，则控制台输出:

```
Waiting for 2 seconds, with x = 10
15[object Promise]
Waiting for 2 seconds, with x = 20
```

这里有一个微任务优先的机制：

Promise 回调属于宏任务，.then、add 函数继续 return 的加法执行都是微任务. 微任务优先于宏任务执行. 所以先 add 函数 return 15[object Promise]，然后 add.then 执行，把这个输出到 console，然后才执行 b Promise 对象的回调，即输出 Waiting for 2 seconds, with x=20.

2.4 `.all` 方法

当同时需要多个 Promise 解决值，且它们解决的先后顺序不重要时，可以使用 `Promise.all()` 方法. 它接受一个 Promise 可迭代对象（通常是一个 Promise 数组），返回**单个**新的 Promise 对象.

- 只要输入数组中有任何一个 Promise 被拒绝，.all 返回的单个 Promise 就会立即以第一个拒绝的 error 被拒绝.
- 只有当输入数组中所有 Promise 都成功解决，.all 返回的单个 Promise 才会 fulfilled，它的解决值是一个新数组，包含了所有输入 Promise 的解决值，顺序与输入数组的顺序一致.

例：

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
}); // [3, 42, "foo"]
```

3. Fetch API

在不重新加载网页的情况下，从服务器异步检索新数据. 它也可以用于向服务器提交数据.

返回对象：`fetch()` 返回一个 Promise 对象，便于处理.

`async/await`

- `async` 关键字将函数声明为异步函数，允许在函数内部使用 `await` .
- `await` 关键字用于暂停函数执行，知道 `fetch()` 返回的 Promise 被解决（resolved）.

数据提取

- 对响应对象调用 `json()` 方法，可以从 HTTP 响应中提取 JSON 数据.
- 还可以使用 `text()`、`formData()`、`arrayBuffer()` 等方法提取其他类型的数据.

同源策略：处于安全原因，`fetch()` 默认要求异步 JS 数据加载必须遵守同源（即在同一服务器/端口）策略.

3.1 原理

fetch 实际上也是执行一个延时任务（假定请求response需要一定时间），它会马上返回一个promise对象，但是是pending，只有成功返回response后才会调用这个promise对象的resolve让它变为fulfilled. await的作用在于，在这个异步函数内部，在执行到fetch时暂停这个异步函数（异步函数外的任务可以正常执行），直到fetch调用了promise的resolve或reject改变了promise的状态，才能继续执行异步函数后面的内容.

注意，fetch 不一定总能返回 fulfilled 的 Promise. 有可能由于服务器问题一直卡着，这时候通常在客户端设置一个超时计时器，然后使用 Promise.race() 让计时器所在的 Promise 和 fetch 返回的 Promise 进行竞速，如果超时的时候 fetch 还没有调用 resolve，则客户端使用 AbortController 取消正在进行的 fetch 请求.

例：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>Promise.race() 示例 - 超时控制</title>
</head>
<body>

    <h2>Promise.race() 演示</h2>
    <p>点击按钮，模拟一个需要 3 秒才能完成的任务，但设置了 2 秒的超时限制。</p>
    <button onclick="runRace(3000)">开始任务 (3秒)</button>
    <div id="output">结果将显示在这里...</div>

    <script>
        const outputDiv = document.getElementById('output');

        /**
         * 1. 创建一个模拟耗时任务的 Promise
         * @param {number} delay 任务完成所需的时间 (毫秒)
         */
        function slowTaskPromise(delay) {
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    // 假设任务成功完成
                    resolve(`任务在 ${delay / 1000} 秒后成功完成！`);
                }, delay);
```

```
            });
        }

        /**
         * 2. 创建一个超时报警的 Promise
         * @param {number} timeout 超时时间 (毫秒)
         */
        function timeoutPromise(timeout) {
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    // 如果计时器先到，就调用 reject 报告超时错误
                    reject(new Error(`请求超时！已超过 ${timeout / 1000} 秒限制。`));
                }, timeout);
            });
        }

        /**
         * 3. 运行竞速
         * @param {number} taskTime 任务时间
         */
        async function runRace(taskTime) {
            const timeout = 2000; // 设置 2 秒的超时限制

            outputDiv.innerHTML = `<span style="color:orange;">状态：任务已开始 (预计 ${taskTime}ms) ... 超时限制：${timeout}ms</span>`;

            try {
                // Promise.race() 接收一个 Promise 数组，并返回最快完成的结果
                const result = await Promise.race([
                    slowTaskPromise(taskTime), // 任务 Promise
                    timeoutPromise(timeout)    // 超时 Promise
                ]);

                // 如果执行到这里，说明 slowTaskPromise 先完成了
                outputDiv.innerHTML = `<span style="color:green;">成功: ${result}</span>`;

            } catch (error) {
                // 如果执行到这里，说明 timeoutPromise 先 rejected 了
                outputDiv.innerHTML = `<span style="color:red;">失败: ${error.message}</span>`;
            }
        }
    </script>

</body>
</html>
```

3.2 .json 方法

Fetch API 最常见的场景就是从 URL 请求返回一个 Promise 对象，该对象包含了所有响应信息，其中一部分我们需要，一部分不需要. 因此把这个对象赋给变量 Response，再进行：

```
data = Response.json();
data.then(text => console.log(text));
```

此处 data 同样为 Promise 对象，但其值已格式化为 json 形式.

> 实际上就是把 Promise 的解决值进行 json 格式化，然后传给 data. json 格式化后的内容就是我们需要的.

这里 `data.then(text => console.log(text));` 只是一个示例，这个数据也可以用于其他操作.

例：`fetch.html`

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Fetch API 和 Async/Await 演示</title>
    <style>
        body { font-family: Arial, sans-serif; padding: 20px; }
        .container { max-width: 800px; margin: 0 auto; border: 1px solid #ddd;
padding: 20px; border-radius: 8px; }
        button { padding: 10px 15px; background-color: #007bff; color: white;
border: none; border-radius: 5px; cursor: pointer; margin-bottom: 20px; }
        button:hover { background-color: #0056b3; }
        #output-area {
            margin-top: 20px;
            padding: 15px;
            border: 1px solid #f0f0f0;
            background-color: #f9f9f9;
            white-space: pre-wrap; /* 保持 JSON 格式 */
            text-align: left;
            min-height: 100px;
        }
        .status { font-weight: bold; margin-bottom: 10px; }
        .loading { color: orange; }
        .success { color: green; }
        .error { color: red; }
    </style>
</head>
<body>

    <div class="container">
        <h2>使用 Fetch 和 async/await 获取数据</h2>
        <button onclick="fetchData()">点击获取用户数据 (ID: 1)</button>

        <div id="status-message" class="status">点击按钮开始</div>
```

```
        <div id="output-area">
            数据将显示在这里...
        </div>
    </div>

    <script>
        const outputArea = document.getElementById('output-area');
        const statusMessage = document.getElementById('status-message');

        /**
         * 异步函数：使用 fetch() 和 async/await 获取数据
         */
        // 1. 使用 async 关键字将函数声明为异步函数
        async function fetchData() {
            const userId = 1;
            const url = `https://jsonplaceholder.typicode.com/users/${userId}`;

            // 重置状态显示
            statusMessage.textContent = '状态: 正在加载中...';
            statusMessage.className = 'status loading';
            outputArea.textContent = '请求已发出，等待服务器响应...';

            try {
                // 2. await: 暂停函数执行，直到 fetch() 返回的 Promise 被解决
(Resolved)
                // fetch() 返回一个 Promise，resolved 后得到 Response 对象
                const response = await fetch(url);

                // 检查 HTTP 状态码，例如 404
                if (!response.ok) {
                    throw new Error(`HTTP 错误！状态码: ${response.status}`);
                }

                // 3. json(): 对 Response 对象调用 json() 方法，它本身也返回一个
Promise
                // 我们再次使用 await 来等待 JSON 数据提取完成
                const data = await response.json();

                // 更新成功状态
                statusMessage.textContent = '状态: 数据获取成功！';
                statusMessage.className = 'status success';

                // 格式化并显示数据
                outputArea.textContent = JSON.stringify(data, null, 2);

            } catch (error) {
                // 捕获请求或数据提取过程中的任何错误
                statusMessage.textContent = '状态: 获取失败！';
                statusMessage.className = 'status error';
                outputArea.textContent = `错误信息: ${error.message}`;
                console.error('Fetch 错误:', error);
            }
        }
    </script>
```

```
</body>
</html>
```

3.3 .then 处理 vs await 处理

fetch() 函数返回一个 Promise 对象，因此它可以使用连续的 .then 来处理成 json 并输出：

```
fetch('http://example.com/movies.json')
.then(res => res.json())
.then(data => console.log(data))
```

当然，它也可以使用 async/await 来处理：

```
let getjson = async () => {
    let response = await fetch('http://example.com/movies.json');
    let data = await response.json();
    return data;
}
getjson().then(text => console.log(text));
```

3.4 request 和 response

待补充.

4. AJAX

待补充.

5. `async` 和 `await` 关键字

在 `3.1` 中，我们提到了这两个关键字.

`async` 函数返回一个 Promise 对象，如果函数内部返回一个非 Promise 值，JavaScript 会自动将其包装成一个 resolved 的 Promise. 不过不一定要使用这个对象.

`await` 关键字只能在 `async` 函数内部使用，用于等待一个 Promise 解决或拒绝. 注意，`await` 后面可以接任何返回 Promise 对象的表达式（函数、方法、Promise 变量）.

利用这两个关键字可以把复杂的 Promise 嵌套改为等效的简单的一系列异步任务：

```
function waitnprint(str){
    return new Promise((resolve, reject)=>{
        setTimeout(function(){
            console.log(str);
            resolve();
        }, 1000);
    })
}
```

定义这个函数后，执行：

```
waitnprint("hello")
.then(()=>waitnprint("world"))
.then(()=>waitnprint("!")
.then(()=>waitnprint("END")
.catch((err)=>console.log(err));
```

如果需要构建 Promise 链，则需要一个接一个的 .then. 不美观，不清晰.

因为我们的核心目标是实现一个输出完才能输出下一个，因此可以等价为执行一个 `async` 函数，这种一个接一个的过程可以用 `await` 来实现：

```
async function longwait() {
    await waitnprint("Hello");
    await waitnprint("World");
    await waitnprint("!");
    await waitnprint("END");
}
longwait();
```

6. 错误处理

6.1 try/catch/finally

在 `async` 函数中，Promise 链中的错误处理（`.catch()` 和 `.finally`）可以通过标准的 `try...catch...finally` 块实现

`await` 表达式可以放在 `try` 块内.

```
async function f(){
    try{
        await something;
    } catch(error){
        ...
    } finally{
        ...
    }
}
```

Lecture 8 Events and Programming

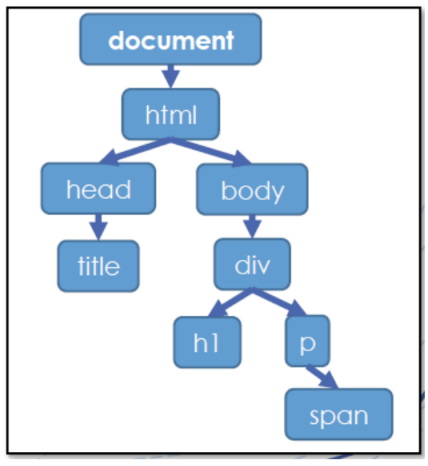
8.1 DOM 树

定义： 浏览器渲染 HTML 文档时，会构建一个树形数据结构，即 **window.document** 对象 。这个树被称为 **文档对象模型 (Document Object Model, DOM)**

结构： DOM 树是一种层次结构，由通过边连接的节点组成；每个节点可以有多个子节点，但只有一个父节点

构建： DOM 树仅在页面加载时构建一次 。它包含所有元素的实际对象、元素的属性以及访问和处理事件的方法

```
<html>
<head>
  <title>Hello!</title>
</head>
<body>
  <div>
  <h1>Nice try</h1>
  <p>DOM is <span>fun</span>!</p>
  </div>
</body>
</html>
```



8.2 访问 HTML 元素

本节指的都是JS 访问、修改 HTML 元素.

访问 HTML 元素有以下几种主要方法：

- **Selectors API：** 使用与 CSS 相同的选择器
  - `querySelector()`：返回第一个匹配的元素
  - `querySelectorAll()`：返回所有匹配的元素，作为一个列表
- **传统技术：**
  - `getElementById()`：根据唯一的 ID 在文档中查找元素
  - `getElementsByClassName()` / `getElementsByTagName()`：在元素内部搜索子元素，返回一个列表

- **对象集合：** 例如 `document.images`、`document.links`、`document.scripts`

方法	特点	返回值类型
`getElementById()`	更快，但 ID 必须唯一 14；最多返回一个元素 15。	单个元素 16
`querySelector()`	较慢，但更灵活 17；可匹配类/属性/ID (#) 1818。	单个元素 1919
`querySelectorAll()`	使用 CSS 选择器匹配 20202020。	**静态** `NodeList` 21
`getElementsByClassName()` / `getElementsByTagName()`	传统技术 22。	**实时** `HTMLCollection` 23

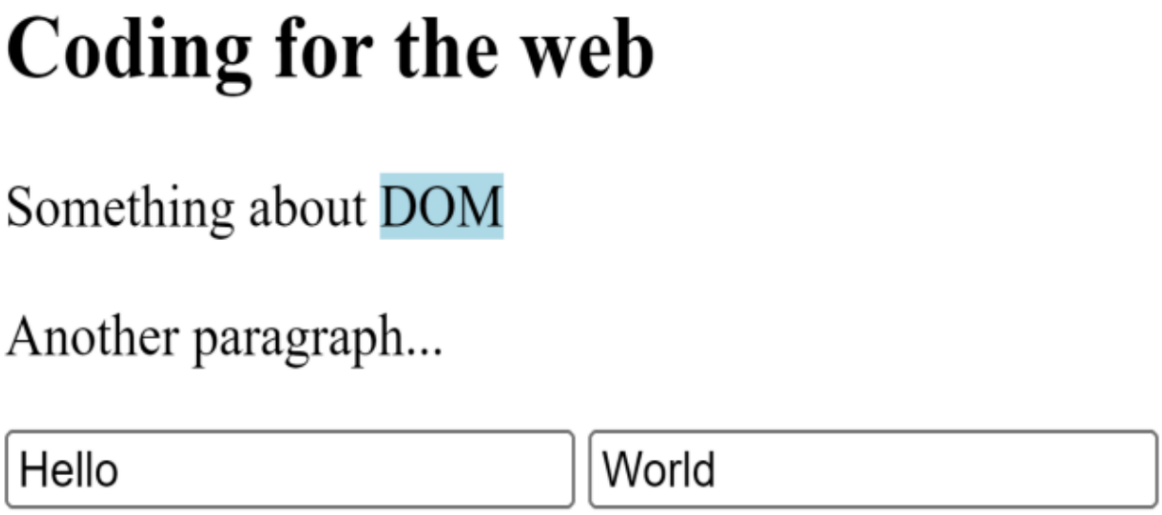
修改内容和属性

可以获取或修改元素的内容和属性：

- `element.innerHTML`：包含标签在内的全部内容
- `element.innerText`：纯文本内容
- `element.value`：仅用于表单元素的值
- `element.attribute`：直接设置 HTML 属性（如 `class`）
- `element.style.property`：直接设置 CSS 属性

```
<!DOCTYPE html>
<html>
<head>
<title>Accessing elements</title>
</head>
```

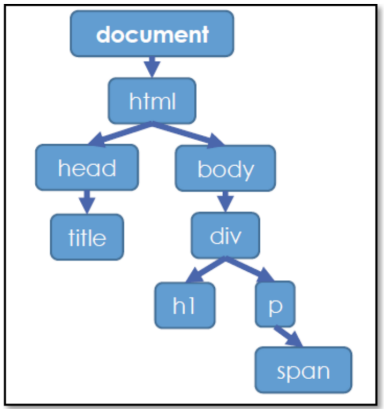
```
<body>
<h2 id="head">Coding for the web</h2>
<p id="para1">Something about <span>DOM</span></p>
<p id="para2">Another paragraph...</p>
<input type="text">
<input type="text">

<script>
// get the <span> inside id=para1, change CSS bg color
document.getElementById("para1").getElementsByTagName("span")
[0].style.backgroundColor = "lightblue";
// get the id attribute of h2
console.log(document.querySelector("h2").id);
// change the value entered in the text input box
document.querySelectorAll("input")[0].value = "Hello";
document.querySelectorAll("input")[1].value = "World";
</script>
</body>
</html>
```



8.3 节点间定位

Navigation

- Navigation between nodes
  - **parentNode**
  - **children[node#]**
  - **firstElementChild**
  - **lastElementChild**
  - **nextElementSibling**
  - **previousElementSibling**
- Node editing
  - **createElement()**
  - **createTextNode()**
  - **appendChild()**
  - **insertBefore()**
  - **remove()**
  - **removeChild()**
  - **replaceChild()**



8.4 Events

Events have a vital role for web interaction, e.g.

• onclick

• onload

• onunload

• onchange

• onmouseover

• onfocus

Either specify event to object or use the `addEventListener()` method.

8.5 对象

> Objects

JS 不是严格意义上的面向对象编程语言

JS 对象本质上是一个 **属性的集合** (A collection of properties)

每个属性都是一个 **键/值对** (key/value pair)，例如 `{name: “john”, age: 18}`

键 (Key) 的规则

- **键** (key) **必须是字符串 (string) 或 Symbol**

- **数字键的特殊性**：`obj[1]` 和 `obj['1']` 是等效的 。JS 会将数字键自动转换为字符串

- **点表示法 (`.`) 的限制**：
  - `obj.1` 是不可能的，因为 `1` 不是一个有效的标识符 (identifier)

    > 在 JavaScript 中，标识符必须以字母、下划线 (`_`) 或美元符号 (`$`) 开头，不能以数字开头

  - 但是，`obj.x` 是可以的，它等同于 `obj['x']`

值 (Value) 的规则

- **值** (value) **可以是任何东西** (anything)
- 值甚至可以是**另一个对象** (another object) 或 `null`

创建对象

```
// 1. Use literal notation (使用字面量表示法)
let stu1 = {name:"john", age:18} // the most commonly seen method

// 2. Define properties directly (直接定义属性)
let stu1 = new Object();
stu1.name = "john";
stu1.age = 18;

// 3. Use a constructor (function/class) (使用构造函数)
function Student(name, age){
    this.name = name;
    this.age = age;
}
let stu1 = new Student("john", 18)
let stu2 = new Student("tom", 17)
```

In most cases, the keyword `this` is in a function refers to the object through which the function is being called

No `this` for arrow functions

8.6 JSON

> JavaScript Object Notation

核心特点

- **用途**：JSON 正在成为一种轻量级的数据交换格式，非常流行
- **内容类型**：其内容类型为 `application/json`
- **语法**：它与 JavaScript 语法的一个子集非常相似，尽管它本身并不是一个严格的子集
- **字符串引号**：JSON 字符串中的字符串字面量**必须**用**双引号** (`"`) 括起来
- **结构**：JSON 支持嵌套结构，例如，对象内的对象、对象的数组等
- **字符支持**：JSON 支持 **UTF-8** 编码，可用于非 ASCII 字符

两种表示

JSON 的数据结构可以表示为以下两种方式：

1. 键值对

> 名称/值对的集合 (Collection of name/value pairs) – 对象字面量 (Object Literal)

- **概念**：这表示一个键值对的集合
- **在其他语言中的对应**：在其他编程语言中，这可以被实现为对象 (object)、记录 (record)、结构体 (struct)、字典 (dictionary)、哈希表 (hash table) 或关联数组 (associative array)
- **示例**：一个包含三个属性 `a`、`b` 和 `c` 的对象：

  JSON

  ```
  { "a":1, "b":2, "c":3 }
  ```

2. 有序列表

> 有序的值列表 (An ordered list of values) – 数组字面量 (Array Literal)

- **概念**：这表示一个有序的值的列表
- **在其他语言中的对应**：在其他编程语言中，这可以被实现为数组 (array)、向量 (vector)、列表 (list) 或序列 (sequence)
- **示例**：一个包含三个整数和一个字符串值的数组：

JSON

```
[1, 2, 3, "value #4"]
```

在 JS 中使用 JSON

JSON 本身是一段字符串 (a piece of string)，但它可以很容易地被解析成 JavaScript (JS) 对象.

```
// 1. JSON 字符串示例
let myJSONtext1 = '{"name":"john", "age":18}'; // pay attention to quotes!

// 2. 解码 (Decode) JSON 编码的数据
let myData = JSON.parse(myJSONtext);

// 3. 编码 (Encode) 数据
let myJSONtext2 = JSON.stringify(myData); // return a string
```

`JSON` 对象是 JS 内置的全局对象，这里使用的方法也是内置的.

8.7 jQuery 遗产

- **定义**：jQuery 是一个用于操作 DOM 树（例如，访问元素）的 JavaScript 库
- **地位变化**：jQuery 在 Web 领域已经存在了 10 多年 。然而，由于 **JavaScript 自身的改进**，它现在正在逐渐淡出
- **基础**：jQuery 是一个构建在 DOM 之上的 JS 库

jQuery 的优势

- **易用性 (Ease-of-use)**：jQuery 非常方便
  - 代码量更少
  - 界面统一 (uniform interface) 等
- **跨浏览器兼容性 (Cross-browser compatibility)**：在兼容性方面，jQuery 以前可能做得更好

Lecture 9 Functions and Arrays

本次讲座主要涵盖 **JavaScript 函数** 和 **数组** 两个主要部分

JavaScript 函数

- 一个 JS 函数包括 `function` 关键字、可选的函数名、可选的参数以及可选的返回值
- **参数 (Parameters)**：函数定义中的输入名称列表
- **实参 (Arguments)**：函数调用时传递的实际值
- **默认值**：可以为参数提供默认值，缺少实参时参数的值为 `undefined`
- 在非箭头函数中，实参也可以在 `arguments` 对象中找到

函数参数

> Function Parameters

除非定义时给了默认值，否则没传参记为传入 undefined

```
function f1(x, y=2, z) {
    console.log("x = " + x);
    console.log("y = " + y);
    console.log("z = " + z);
}
```

```
f1(5);
"x = 5"
"y = 2"
"z = undefined"
```

非 arrow functions 可以把接到的未定义参数放入 augments 对象. 例：

```
function f2() {
    for (i of arguments) {
    console.log(i);
    }
}
```

```
f2(1,2,3)
"1"
"2"
"3"
```

Rest Operator (...)：一种获取未知数量实参的新方式

- Rest 参数必须是参数列表中的**最后一项**
- 它将多余的实参收集到一个数组中
- 可以在箭头函数中使用

```
function f3(x, y, ...more) {
    console.log("x is " + x);
    console.log("y is " + y);
    console.log(more);
    console.log(typeof more);
    }
```

```
f3(2,4,6,8,10)
"x is 2"
"y is 4"
[6,8,10]
"object"
```

- Rest 参数必须是参数列表中的**最后一项** 9。

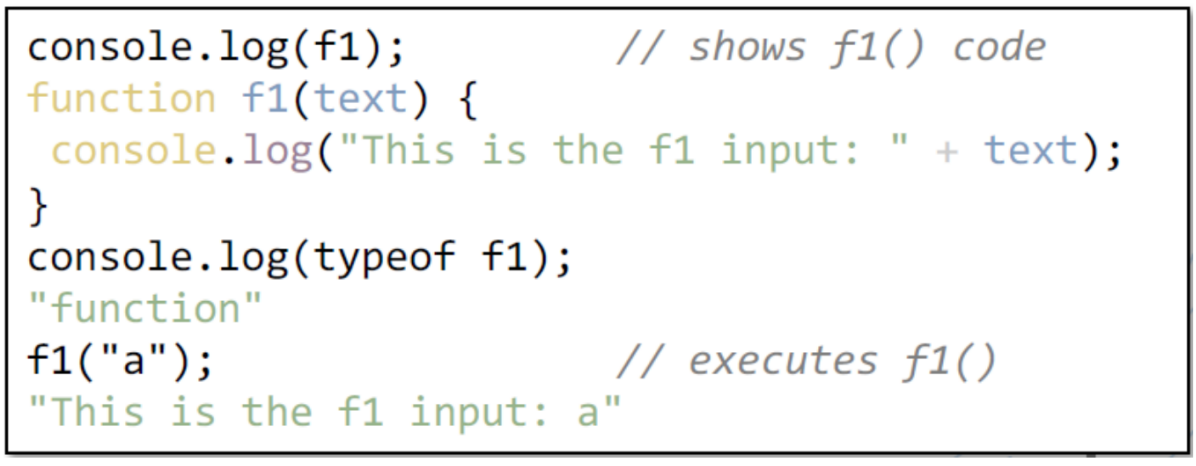
- 它将多余的实参收集到一个数组中 10。

- 可以在箭头函数中使用 11。

函数声明与表达式

> Function Declaration vs Expression

```
// function declaration
function f1(text) {
    console.log("This is the f1 input: " + text);
}

// function expression with anonymous function
let f2 = function (text) {
    console.log("This is the f2 input: " + text);
};

// arrow (anonymous) function in expression
let f3 = text => console.log("This is the f3 input: " + text);

// 带有输出的代码块
console.log(f1);
// function f1(text) {
//     console.log("This is the f1 input: " + text);
// }

console.log(typeof f1);

f1("a");
```



- 函数代码在 JavaScript 中是以普通值的形式存储的
- **函数声明 (Function Declaration)** 会被提升 (hoisted) 到其作用域的顶部，即可以在声明之前使用

- **函数表达式 (Function Expression)** 通常将一个函数赋值给一个变量

箭头函数

> Arrow Functions

- 语法：`(para1, para2, ...) => { statements; }`
- **单个参数**时可以省略圆括号 `()`
- **单行表达式**时可以省略大括号 `{}` 和 `return` 关键字
- **多行语句**时必须使用大括号 `{}`
- **与常规函数的区别**：箭头函数没有自己的 `this` 和 `arguments`，不适合作为构造函数

调用函数

> Invoking Functions

- 函数通过函数/变量名后的圆括号 `()` 来调用（执行）
- 不带圆括号将返回函数代码本身

定义后立即调用

```
(function() {
    console.log("Hello there");
})();
```

箭头形式

```
(() => console.log("Hello again"))();
```

- **立即执行函数表达式 (IIFE)** 的作用：变量无法从外部作用域访问，防止潜在的命名冲突，用于一次性的初始化或设置任务

```
// it simply invokes the alert() function.
document.querySelector("#btn1").onclick = alert("hi");
// 常见错误，错误原因：alert("hi") 语句在代码执行到这一行时会立即运行 。它会弹出一个包含 "hi" 的警告框，并且它的返回值（alert() 函数返回 undefined）随后被赋给了 #btn1 元素的 onclick 属性 结果：当页面加载并执行到这行代码时，警告框会立即弹出 。此后，点击 #btn1 按钮时将不会发生任何事情，因为它的 onclick 属性被赋值为 undefined

// either arrow function or regular function is okay
document.querySelector("#btn2").onclick = () => alert("Correct onclick alert");

//正确原因：这里赋值给 onclick 属性的是一个函数（一个箭头函数表达式 () => alert(...)）。这个函数是一个 回调函数，它被存储起来，不会立即执行 只有当用户点击 #btn2 按钮时，这个函数才会被 调用（执行），从而弹出警告框
```

嵌套函数

> Nested Functions

- JS 中的函数可以嵌套
- 内部函数可以访问外部函数的变量，但**反之则不行**

例:

```
function f1(a) {
    function f2(b) {
        return a + b;
    }
    return f2; //返回代码，而不是执行结果
}

console.log(f1(10)); // code of f2 is returned，即function f2(b) { return a + b; }
console.log(f1(10)(5)); // results of f2(5) is returned.这种结构是函数柯里化（Currying）的一个简单例子，其中一个函数返回另一个函数
//与上一步相同，f1(10) 返回了函数 f2 的代码引用
//紧接着，返回的函数 f2 立即被调用，参数 b 的值为 5
//f2 执行 return a + b。由于 a 被 f1 的作用域锁定为 10（这是一个闭包），所以它返回 10 + 5（输出15）
```

回调函数

> Callback Functions

```
function f0(callback1, callback2) {
    let x = prompt("A number?");
    if (x % 2)
        callback1();
    else
        callback2();
}

f0(
    () => alert("Odd"),
    () => alert("Even")
); //传入两个箭头函数作为参数并执行
```

- 由于函数在 JavaScript 中是简单的值，因此它们可以作为函数的**参数**传递，这些被传递的函数就是回调函数（callbacks）
- 常用于异步 JS，在等待时间或事件发生后才被调用

Generator 函数

> Generator Functions

- Generator 函数可以返回一个值并在稍后重新进入
- 使用特殊关键字 `function*` 和 `yield`
  - `yield` 关键字用于**暂停**函数执行并**返回**一个值
  - `next()` 方法用于**恢复**执行

```
<button onclick="alert(count.next().value)">
    Click
</button>

<script>
    function* g(inc) {
        let x = 0;
        while (1) // 一个无限循环
            yield x += inc;
    }
    let count = g(2); //创建 Generator 对象 count
</script>
```

调用 `g(2)` **不会立即执行** `g` 函数体内的代码。

它返回一个 **Generator 对象**（在这里命名为 `count`），这个对象有一个 `next()` 方法。增量 `inc` 被设置为 2

按钮点击和执行

- **按钮代码**：`<button onclick="alert(count.next().value)">Click</button>`

每次点击按钮时，都会执行以下操作：

1. **调用** `count.next()`：恢复 `g` 函数的执行，直到遇到下一个 `yield` 语句
2. **执行** `yield x += inc;`：
   - `x` 增加 `inc` (即 2)
   - 函数暂停，并返回一个包含 `value` 和 `done` 属性的对象
   - `value` 属性包含 `x` 的新值
3. **显示** `value`：`alert(...)` 显示返回对象的 `value` 属性

Generator 函数的关键在于它可以在多次调用中**保持**（记住）其局部变量 `x` 的状态

对象方法

> Object Methods

- 对象可以包含函数，这些函数称为对象方法
- 对象方法可以使用简写语法 `shout() { ... }`
- 对象方法常用于操作该对象内部的数据

```
let human = {
    keyword: "Hello!",
    shout: function() { alert(this.keyword) }
};//传统对象方法定义

let human2 = {
    keyword: "Hello again!",
    shout() { alert(this.keyword) } // alternative shorter syntax for methods
};//简洁定义

human.shout();
human2.shout();
```

JavaScript 数组

> JS Arrays

JS 数组

- JS 数组是有序的集合
- 数据类型不限，可以包含函数、对象和/或数组
- 数组是一种特殊的对象
- 使用 `Array.isArray()` 来验证一个变量/表达式是否为数组
- 数组像对象一样是**按引用复制**的
- 如果需要**按值复制**，可以使用 `...` (Spread Operator)、`[].concat(x)` 或 `Array.from(x)`

```
let x = [1, 2, 3];
console.log(Array.isArray(x)); // true

let y = x; //复制引用（指针）
console.log(y); // [1,2,3]
y[1] = 0;
console.log(x); // [1,0,3]
console.log(y); // [1,0,3]

// if you want to copy by value
s = [...x]; //将 x 中的所有元素“展开”到一个新的数组 s 中
u = [].concat(x); //使用空数组调用 concat() 方法并传入 x，concat 会返回一个包含 x 元素的新数组 u
v = Array.from(x); //Array.from() 方法从一个类数组对象或可迭代对象（如数组 x）创建一个新的、浅拷贝的 Array 实例 v

//这些方法执行的都是浅拷贝。对于包含原始值（数字、字符串）的一维数组，它们等同于深拷贝。但如果数组中包含对象或其他数组，这些对象/数组仍然是按引用复制的
```

创建数组

> Creating Arrays

- 可以通过 `Array.from()` 从一个**类数组对象**（具有 `length` 属性和索引元素的对象）创建数组
- 可以使用 **Spread Operator (`...`)** 组合其他数组

```
let s = "Hello World";
let array = Array.from(s);
console.log(array);
// ["H","e","l","l","o"," ","W","o","r","l","d"]
console.log(s.length); // 11
console.log(array.length); // 11
let a = [1,3,5];
let b = [2,4,6];
let c = [...a, 0, ...b];
console.log(c); // [1,3,5,0,2,4,6]
```

解构数组

> Destructuring Arrays

- 可以将数组解构成单独的变量
- 这使得函数可以返回多个值
- 也支持 Rest Operator (`...`)

```
let a, b, restVar;
[a, b] = [10, 20];
console.log(a);// 10
console.log(b);// 20
[a, b, ...restVar] = [10, 20, 30, 40, 50];
console.log(restVar);// [30,40,50]
```

修改数组

> Modifying Arrays

如果您只想**提取**数组的一部分并**不改变**原数组，请使用 `slice`。

如果您需要**删除、添加或替换**数组中的元素，并**改变**原数组，请使用 `splice`。

方法	描述	是否修改原数组
`array.slice(start, [end])`	返回从 `start`（包含）到 `end`（不包含）的新切片数组	**否**
`array.splice(start, [deleteCount, [itemsToAdd...]])`	更改原数组，并返回一个包含被删除元素的数组	**是**
`array.pop()`	移除并返回最后一个元素（栈操作 LIFO） 49。	**是**
`array.push(items)`	向数组末尾添加元素（栈操作 LIFO） 51。	**是**
`array.shift()`	移除并返回第一个元素（队列操作 FIFO） 53。	**是**
`array.unshift(items)`	向数组开头添加元素（队列操作 FIFO） 55。	**是**

```
let c = ["cyan","magenta","yellow","black"];
let c1 = c.slice(1,2);
console.log(c);
// ["cyan","magenta","yellow","black"]
console.log(c1);
// ["magenta"]
let c2 = c.splice(2,1,"red","green","blue");
console.log(c);
// ["cyan","magenta","red","green","blue","black"]
console.log(c2);// ["yellow"]
```

迭代数组

> Iterating Arrays

方法	描述
传统 `for` 循环	灵活性高，速度最快

方法	描述
`for...of` 循环	便于获取数组元素的副本（例如，用于展示），不能修改原数组元素
`forEach(item, index, array)`	接受一个函数作为输入，可以访问元素、索引和整个数组。用于修改原数组需要使用第三个参数 `array`（例如 `c[i]+=1`）

```
// 示例 1: 传统 for 循环 (修改原数组)
let a = [1, 3, 5];
for (let i=0; i<a.length; i++)
{
    console.log(a[i]); // 输出当前值
    a[i] = a[i]+1;    // 修改数组元素
}
console.log(a); // [2,4,6]

// 示例 2: for...of 循环 (不修改原数组)
let b = [1, 3, 5];
for (let item of b) {
    console.log(item); // 输出当前值
    item = item + 1;   // 仅修改局部变量 item，不修改 b 数组
}
console.log(b); // [1,3,5] not modified

// 示例 3: forEach() - 打印
let c = [1, 3, 5];
c.forEach(item=>console.log(item));
// 1
// 3
// 5

// 示例 4: forEach() - 尝试修改 (失败)
let c = [1, 3, 5];
c.forEach(item=>item=item+1);
console.log(c); // [1,3,5] not modified

// 示例 5: forEach() - 通过索引修改 (成功)
// forEach() 方法接受一个回调函数，该函数最多可以接收三个参数：(item, index, array)
let c = [1, 3, 5];
c.forEach((item, i, c) => c[i] = c[i]+1);
console.log(c); // [2,4,6]
```

搜索数组

> Searching in Arrays

方法	描述
`array.indexOf(item, start)`	从 `start` 查找 `item` 的索引，未找到返回 `-1` 61。
`array.lastIndexOf(item, start)`	从后向前查找 `item` 的索引 62。
`array.includes(value)`	如果数组包含该值则返回 `true` 63。
`array.find(function(item, index, array))`	返回**第一个**使函数返回 `true` 的元素 64。
`array.filter(function(item, index, array))`	返回一个包含**所有**匹配元素的数组 65。

```
const a = ['a', 'b', 'c', 'b'];

// 1. a.indexOf('b');
// 结果: 1
// 解释: 从数组开头搜索 'b'，第一次出现在索引 1 的位置。

// 2. a.indexOf('b', 2);
// 结果: 3
// 解释: 从索引 2 ('c') 开始搜索 'b'。'b' 第一次出现在索引 3 的位置。

// 3. a.includes('a');
// 结果: true
// 解释: 数组 a 中包含元素 'a'。

// 4. a.includes(1);
// 结果: false
// 解释: 数组 a 中不包含数字 1。
```

```
let num = [10, 12, 13, 15, 20];

// 1. console.log( num.find(n => n % 5) )
// 结果: 12
// 解释:
// 回调函数是 `n => n % 5`。在 JavaScript 中，任何非零的数字在布尔环境中都被视为 `true`，零被视为 `false`。
// - 10 % 5 = 0 (False)
// - 12 % 5 = 2 (True) -> **找到第一个匹配项**The first item returning true in function will be returned
// `find()` 立即停止搜索并返回 2 的元素值，即 **12**。
```

```
// 2. console.log( num.filter(n => n % 5) )
// 结果: [12, 13]
// 解释:
// `filter()` 会遍历整个数组，并收集所有返回 `true` 的元素。
// - 10 % 5 = 0 (False)
// - 12 % 5 = 2 (True) -> 12 被加入结果数组
// - 13 % 5 = 3 (True) -> 13 被加入结果数组
// - 15 % 5 = 0 (False)
// - 20 % 5 = 0 (False)
// 最终返回包含所有匹配元素的数组：**[12, 13]**。
```

转换数组

> Transform Arrays

- `array.reverse()`：反转数组中元素的顺序
- `array.split()` / `array.join()`：用于字符串与字符数组之间的转换

  > Converting a string to character array, or vice versa

- `array.map(function(item, index, array))`：返回一个应用了转换函数的新数组

```
let num = [10, 12, 13, 15, 20];
console.log( num.map(n => n * 2) )
// [20, 24, 26, 30, 40]
```

返回一个新数组，原始数据本身未被修改

HTMLCollection vs NodeList

共同点

- 两者都是**“类数组”对象**（Array-like Objects）
  - 它们具有数字索引和 `length` 属性
  - 它们缺少数组提供的完整方法集合
  - 它们在内存使用和性能方面更轻量lighter（因此在 Web 开发中很重要）
  - 可以使用 `Array.from()` 将它们转换为功能完整的数组

区别

NodeList is **static**, meaning that it won't update the property (e.g., length) even after we update the elements.

HTMLCollection is **live**, in some sense same as an array.

功能上：

- `HTMLCollection`：没有内置的 `forEach()`、`find()` 等函数 它通常是**实时**的 (live)，与数组相似
- `NodeList`：保留了 `forEach()` 等函数，但仍然有限制（例如，不能使用 `filter()`） 它通常是**静态**的 (static)，更新元素后其属性（如 `length`）不会更新

Lecture 10 ReactJS

前面讨论的是 Web 应用的基础组成，即 HTML + CSS + JS. 为了使用增强功能和语法糖（JSX、TypeScript、Sass/Less），需要额外的转译过程（源到源编译），生成浏览器可读的文件.

前端库和框架：React、Vue.js 和 Angular 是主要的前端技术.

React：快速、模块化、可扩展、灵活.

特性：

- 虚拟 DOM
- JSX
- 组件
- 属性
- 状态
- 事件
- 单向数据流
- 条件渲染
- 表单
- 生命周期方法

1. 使用

本课：Embedding React using `<script>`

```
<head>
...
<script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script> // @18 specifies the version to use
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
...
</head>
```

2. 虚拟 DOM

> Document Object Model

You can pass the DOM control of your HTML to reactDOM by specifying an element with ID.

```
const root = ReactDOM.createRoot(document.querySelector('#app'));
root.render(element);
```

The element with id=app will be updated by React automatically.

加载完之后，所有 id 为 app 的原始元素都会被 React 元素 element 替换.

2.1 React 元素

React 元素是在 JS 中定义的 JavaScript 对象，这是它和 HTML 元素的区别. 但是声明的语法很相似，因为开发者发明了 JSX（JavaScript XML），它遵循 XML/HTML 的标签结构，目的是让前端开发者能够以熟悉且直观的方式来描述 UI 结构.

JSX 不能直接运行，要引入 Babel 转移器.

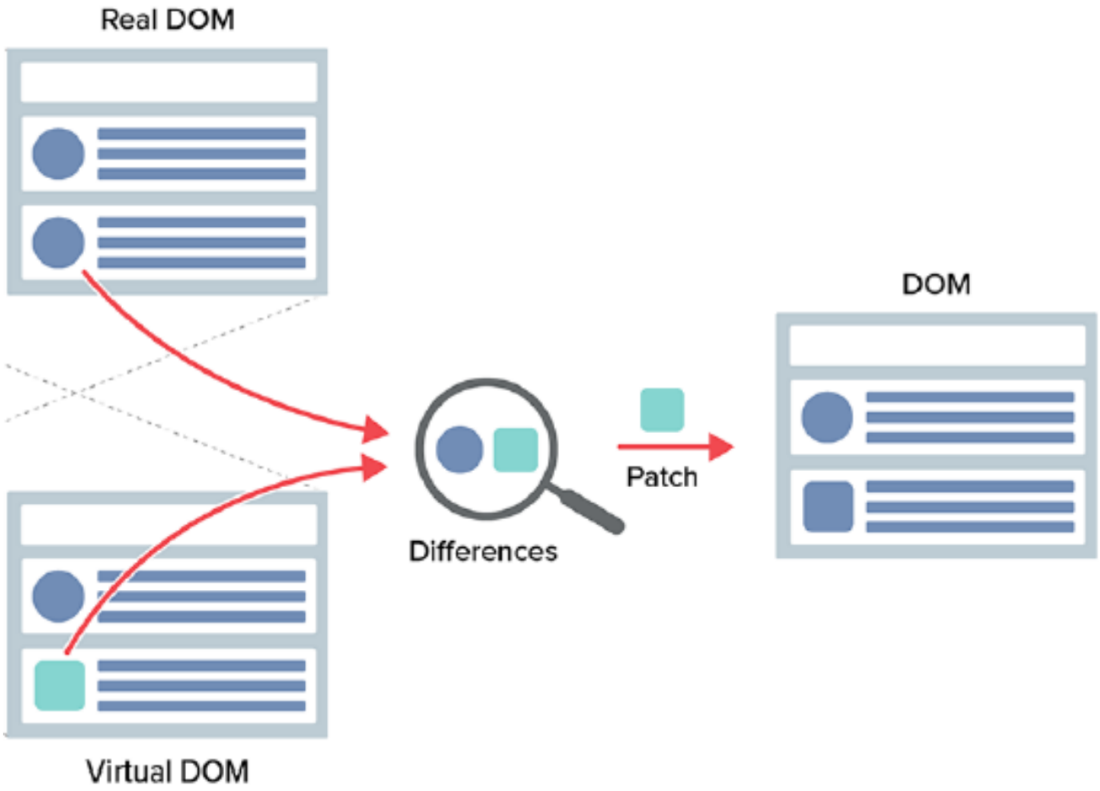
例：`vdom.html`

```
<!DOCTYPE html>
<html>
<head>
    <title>我的第一个 React 应用</title>
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
```

```
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>

    <div id="app">
        <h1>React 正在加载...</h1>
    </div>

    <script type="text/babel">
        // 1. 定义一个简单的 React 元素
        const element = <h1>Hello, React 已经接管了 DOM 控制权!</h1>;

        // 2. 找到要由 React 控制的 HTML 元素
        const container = document.querySelector('#app');

        // 3. 创建一个 React 根 (Root)
        // 这是将 DOM 控制权交给 React 的关键一步
        const root = ReactDOM.createRoot(container);

        // 4. 将 React 元素渲染到根容器中
        // 具有 id=app 的元素将被 React 自动更新
        root.render(element);
    </script>

</body>
</html>
```

第三个引入的脚本是 Babel 转译器.

2.2 DOM



Real DOM：浏览器为了渲染和显示 HTML 元素，会维护一个 DOM 树. 直接操作这个 Real DOM 成本很高，效率很低.

Virtual DOM：React 维护了一个额外的、在内存中的数据结构，作为 DOM 的一个表示，通常被称为 ReactDOM.

更新流程：当应用程序中的数据（State 或 Props）发生变化时，更新流程如下：

- 整个 UI 重新渲染，当某些内容发生变化时，整个 UI 会在 ReactDOM 中被重新渲染.
- 计算差异：React 会找出这个新版本的虚拟 DOM 与原始版本（旧的虚拟 DOM）之间的差异.
- 应用补丁（Patching）：计算出差异后，React 只将这些计算出的差异（补丁）应用到实际的浏览器 DOM 上.

优势：性能提升. 操作内存中的虚拟 DOM 比操作浏览器的真实 DOM 要快得多.

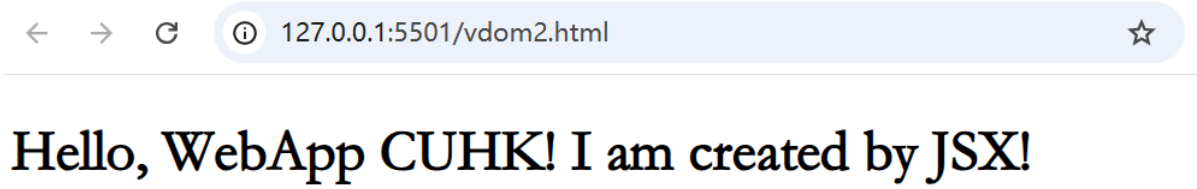
高效更新：即使整个 UI 在虚拟 DOM 中被重新渲染，最终也只有需要改变的部分才会被更新到屏幕上，实现了快速响应的用户体验.

3. JSX

JavaScript 的语法拓展，可选. 被广泛使用用于生成 React 元素，需要 Babel 进行转译.

例：

```
<script type="text/babel">
    function formatName(u) {
    return u.firstName + ' ' + u.lastName;
}
    const user = { firstName: 'WebApp', lastName: 'CUHK' };
    const element = <h1>Hello, {formatName(user)}! I am created by JSX!</h1>;
    const root = ReactDOM.createRoot(document.querySelector('#app'));
    root.render(element);
</script>
```



注意（Lab 6）：若一个父组件中有一些子组件渲染失败（例如未定义），父组件就无法成功渲染，整个都不会被添加到实际的 DOM 中.

4. React 元素 + CSS

React 元素可以使用 CSS：

```
const myStyle = {
    color: 'blue',
    fontSize: '24px',
};
const element = <h1 style={myStyle}>Hello, I am created by JSX!</h1>;
```

注意，这是 JSX 下的代码，如果没有 JSX：

```
const element = React.createElement("h1", {style: myStyle}, "Hello, I am from
React with
CSS.");
```

5. 组件

> Components

是 UI 中的任何元素（如段落、按钮等），作为可重用的构建模块

- 组件名称通常以**大写字母**开头
- 有**函数式组件**和**类组件**

props is a special keyword that represents properties passed to a component

5.1 函数式组件

> Functional Components

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>;
}
function App() {
    return (
    <div>
    <Welcome name="Colin" />
    <Welcome name="all students" />
    </div>
    );
}
const root = ReactDOM.createRoot(document.querySelector('#app'));
root.render(<App />);
```

5.2 类组件

> People usually use class component instead of functional component

```
class App extends React.Component {
    render() {
        return (
        <div className="container">
        <Item />
        <Item />
        <Item />
        </div>
        );
    }
}
class Item extends React.Component {
    render() { return <div className="box">CSCI</div>; }
}
const root = ReactDOM.createRoot(document.querySelector('#app'));
root.render(<App />);
```

其中，所有类组件都要有一个 `render()` 方法. 该方法负责返回（渲染）要显示在屏幕上的 JSX 结构.

`App` 是父组件，组合了多个 Item 组件（可重复使用性）. `App` 在其 render() 方法中返回一个带有 `className="container"` 的 `div`

渲染到 DOM：

```
const root = ReactDOM.createRoot(document.querySelector('#app'));
root.render(<App />);
```

入口：找到 HTML 中 `id="app"` 的元素，创建 React 根.

渲染：根调用 .render 方法指示 React 开始渲染最顶层的 App 组件及其所有子组件到 #app 容器中.

类组件的特点：

- State：类组件可以拥有和管理状态（this.state），并通过 this.setState() 来更新状态.
- 生命周期方法（Lifecycle methods）：类组件提供了一系列方法，允许你在组件的挂载、更新和卸载等不同阶段插入自定义逻辑.

5.3 属性

用于从父组件向子组件传递信息：

```
class App extends React.Component {
render() {
return (
<div className="container">
<Item subject="CSCI" />
<Item subject="CENG" />
<Item subject="AIST" />
</div>
);
}
}
class Item extends React.Component {
render() { return <div className=“box”> My major is {this.props.subject}</div>; }
}
const root = ReactDOM.createRoot(document.querySelector('#app'));
root.render(<App />);
```

在父组件中引入子组件时，在子组件标签里可以直接定义并传递属性. 在子组件中，用 `this.props.{属性名}` 即可调用.

5.4 状态

待补充.

定义了组件在给定时刻的行为，状态值只能通过 `this.setState()` 更新.

当状态改变时，受影响的组件可能会被重新渲染.

状态设置语法：

```
this.setState({state1: stateValue})
```

属性 vs 状态

- 属性不可变，由父组件传递给子组件. 只读
- 状态可变，组件内部自身管理，只能在组件的 constructor 中初始化. 可以修改，通过 `this.setState()` 可以修改自己的状态

6. Events

待补充.

7. 条件渲染

待补充.

通常基于布尔值决定是否显示某项内容.

8. 表单

待补充.

React 倾向于使用受控组件，将用户输入存储到state中，并提供 handleChange() 和 `handleSubmit()` 等方法来管理输入和提交.

9. 生命周期方法

待补充.

组件经历挂载 - 更新 - 卸载的生命周期，这些方法允许在特定阶段插入自定义功能.

10. Hook

特征	类组件	函数组件
基础	ES2015 之后支持的类	易于使用的函数（包括箭头函数）
Props 读取	通过 `this.props`	直接通过 `props`
事件处理	作为类方法，通过 `this` 调用	组件内部的简单函数
状态管理	使用 `this.state` 读取，`this.setState()` 更改	使用新的 `useState()` Hook

10.1 `useState()`

语法：

```
const [状态变量, 更新函数] = useState(初始值);
```

`useState()` 是 React 新版本（自 v16.8, 2019 年）中用于设置 Hook 的函数

它返回一个解构数组 `[var, func]`

- `var`：状态变量，用于读取当前值.
- `func`：更新状态的函数，是更新该变量的唯一方式.

`useState()` 参数用于状态的初始化，类似于类组件构造函数中的 `this.State`

调用更新函数类似于类组件中的 `this.setState()`

例：

```
function Count() {
    // 1. 初始化状态：count = 0
    const [count, myFunc] = useState(0);

    // 2. 定义事件处理函数，通过更新函数 myFunc 改变状态
    const userClick = () => myFunc(count + 1);

    return (
        <h1 onClick={userClick}> 当前计数是 {count}</h1>
    );
}
```

10.2 `useEffect()`

生命周期方法允许在组件的挂载、更新和卸载等阶段插入自定义功能

挂载/更新时：`componentDidMount()`、`componentDidUpdate()` 等

卸载时：`componentWillUnmount()`

`useEffect()` 是函数组件中用于处理**副作用**（如数据获取、订阅或手动更改 DOM）的 Hook

它取代了类组件中的一些生命周期方法：

调用时机：它在组件**渲染时**和组件**更新时**都会被调用

清理：`useEffect()` 内部返回一个函数可以用来模拟 `componentWillUnmount()` 的行为，执行清理工作

通过 `useEffect()`，可以轻松实现类似于 `componentDidMount()` 和 `componentDidUpdate()` 的逻辑，例如示例中用于实现倒计时功能

```
useEffect(() => {
    // 1. 副作用代码 (例如：设置定时器、发起网络请求)

    return () => {
        // 2. 清理函数 (可选)
        // 在组件卸载或下次副作用运行前执行清理 (模拟 componentWillUnmount)
    };
}, [依赖项数组]); // 3. 依赖项数组 (可选)
```

例：`hook.html`

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>React 概念演示：类组件与函数组件</title>
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
    <style>
        body { font-family: Arial, sans-serif; display: flex; justify-content:
space-around; padding-top: 50px; }
        .component-box { border: 2px solid #ccc; padding: 20px; border-radius:
8px; width: 45%; text-align: center; }
        h2 { border-bottom: 2px solid #eee; padding-bottom: 10px; }
        button { padding: 10px 20px; font-size: 16px; cursor: pointer;
background-color: #007bff; color: white; border: none; border-radius: 5px;
margin-top: 10px; }
    </style>
</head>
<body>

    <div id="app"></div>

    <script type="text/babel">
        // 从 React 中解构出 useState 和 useEffect Hook  3, 8]
        const { useState, useEffect } = React;

        // -------------------------------------
        // 1. 函数组件 (Functional Component) 演示：useState 和 useEffect
        // -------------------------------------

        function Counter() {
            // 使用 useState() 初始化状态变量 count 和更新函数 setCount  3, 5]
            const [count, setCount] = useState(0);

            // 定义点击事件处理函数
            const userClick = () => setCount(count + 1); // 通过 setCount() 更新状
态  5]

            // 使用 useEffect() 模拟生命周期副作用  8]
            useEffect(() => {
                console.log(`计数器已更新到: ${count}`);

                // 返回一个清理函数，模拟 componentWillUnmount
                return () => {
                    console.log('计数器组件卸载或更新前清理...');
                };
            // 依赖数组为空 [] 表示只在挂载和卸载时运行（类似
componentDidMount/componentWillUnmount）
            }, []);
```

```
        return (
            <div className="component-box">
                <h2>函数组件 (Hook)</h2>
                <p>当前计数: <strong>{count}</strong></p>
                <button onClick={userClick}>点击增加计数</button>
            </div>
        );
    }

    // ------------------------------------
    // 2. 类组件 (Class Component) 演示：State 和生命周期
    // ------------------------------------

    class Countdown extends React.Component {
        // 构造函数用于状态初始化  16]
        constructor(props) {
            super(props);
            // 使用 this.state 初始化状态  10, 16]
            this.state = { count: 10 };
        }

        // 挂载成功后执行（类似 componentDidMount） 1]
        componentDidMount() {
            this.timerID = setInterval(
                () => this.tick(),
                1000
            );
        }

        // 组件更新后执行（类似 componentDidUpdate） 1]
        componentDidUpdate() {
            if (this.state.count === 0) {
                clearInterval(this.timerID);
            }
        }

        // 组件卸载前执行（类似 componentWillUnmount） 1]
        componentWillUnmount() {
            clearInterval(this.timerID);
        }

        tick() {
            if (this.state.count > 0) {
                // 必须通过 this.setState() 来更新状态  10]
                this.setState({
                    count: this.state.count - 1
                });
            }
        }

        render() {
            // 读取状态使用 this.state.count  10]
            return (
                <div className="component-box">
                    <h2>类组件 (Class)</h2>
                    <p>倒计时：<strong>{this.state.count}</strong></p>
```

```
                    </div>
                );
            }
        }

        // ------------------------------------
        // 3. 根组件 (App Component)
        // ------------------------------------

        function App() {
            return (
                <div style={{ display: 'flex', gap: '20px' }}>
                    <Counter />
                    <Countdown />
                </div>
            );
        }

        // ------------------------------------
        // 4. 渲染到 DOM
        // ------------------------------------

        // 获取 ID 为 'app' 的 DOM 元素  13]
        const root = ReactDOM.createRoot(document.querySelector('#app'));
        // 渲染根组件  13]
        root.render(<App />);

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

Lecture 11 NPM

> 包管理器

11.1 npm

npm 是 **Node.js** 的一部分，最初是“Node Package Manager”的缩写

要获取 npm，需要下载安装 Node.js

本地包存放在项目中的 `node_modules` 文件夹中

全局包存放在系统文件夹中（需要管理员权限）

大多数包在 Node.js 环境中用于后端开发

11.2 npx

npx 是 **npm** 的补充工具

它允许您**无需安装即可执行 Node 包**

示例：`npx create-react-app my-app`

11.3 CREATE-REACT-APP

create-react-app 是一种准备 **React** 应用骨架的常见方法

使用命令 `npx create-react-app theAppName`

然后，使用 `npm start` 命令可以在开发模式下转译代码、构建应用并启动本地 Web 服务器

使用 `npm run build` 命令可以构建用于生产环境的应用，优化后的 HTML/CSS/JS 会存放在 `build` 文件夹中

Lecture 12 SPA and Routing

懒加载

Use `loading = "lazy"` in `<img>` or `iframe`

待补充.

Lecture 13 HTTP

万维网采用客户端-服务器架构. 客户端从中心化的服务器请求获取服务（request），服务器等待客户端请求并作出响应（Response）

OSI 网络模型：网络通信可视为多层. OSI 模型有 7 层，第一层是 Physics Layer，第七层是 Application Layer.

第四层：Transport Layer，传输层

使用 TCP (Transmission Control Protocol) 传输协议传输数据.

网络协议（Communication Protocols）：为了让两台计算机通信，需要对步骤进行清晰定义，包括规则、语法、语义、同步和错误恢复方法

地址：

MAC 地址（Layer 2）：用于在本地网络上定位通信设备

IP 地址（Layer 3）：用于在网络中标识网络接口（IPv4 为 32 位，IPv6 为 128 位）. 私有地址限制数据仅在本地网络内发送

端口（Ports）：网络连接是通过网络设备上的端口建立的.

- **知名端口 (Well known ports)**：0 - 1023 (HTTP 为 80，HTTPS 为 443)

  客户端通常为每个新连接使用一个随机的**私有端口** (49152-65535)

  Web 服务器 (如 Apache) 默认监听 80 和 443 端口

Socket (套接字)：在网络编程中，网络套接字是通信的端点，等于**传输协议 + IP 地址 + 端口号**

Localhost (本地主机)：允许通信只在一台计算机内进行，通常用 `localhost` 或 `127.0.0.1` 标识

HTTP：请求与响应

- **HTTP (HyperText Transfer Protocol)**：是一种在 Web 客户端和 Web 服务器之间传输数据的协议
  - 通信由**客户端**发起，发送 HTTP **请求**给服务器
  - 服务器返回 HTTP **响应**给客户端
  - HTTP 是**无状态**的，每个请求都被视为独立的请求

- **HTTP 客户端 (Clients)**：也称为"用户代理" (`user-agent`)，是通过 HTTP/HTTPS 与 Web 服务器通信的软件
  - **Web 浏览器** (如 Chrome, Safari)
  - **Web 爬虫/搜索引擎**
  - **程序化接口** (如 Postman)

- **HTTP 服务器 (Servers)**：处理 HTTP/HTTPS 的 Web 文档请求，并支持服务器端脚本
  - 常见的 Web 服务器有 **Nginx**、**Apache**、**Microsoft IIS**、**Node.js**

- **URL (Uniform Resource Locator)**：用于识别和定位 Web 资源 。它由**协议**、**主机名**、**路径**和**文件名**等部分组成

请求

HTTP 请求 (Request)

HTTP 请求由**请求行**、**头部 (Headers)** 和可选的**主体 (Body)** 组成

- **HTTP 方法 (Methods)**:
  - **GET**：最常用，用于用户点击链接、浏览器获取文件或提交表单
  - 数据在请求 URL 的文本形式**内部**传递
  - 数据量有限 (~2KB)

  - **POST**：用于向服务器发送数据，数据封装在请求**主体**内部 。数据大小受主体大小限制 (~1MB 到 2GB)

  - **GET vs. POST 安全性**：GET 请求的 URL 会被保存和可见，可能存在安全隐患；POST 请求的 URL 不记录主体数据

- **请求 URL**：通常是 `/path?query` 形式的文件路径字符串，查询字符串包含 URL 编码的名称-值对

- **头部 (Headers)**：包含多个名称-值对的头部字段
  - `User-Agent`：客户端及其操作系统的信息
  - `Referer`：将客户端带到请求页面的网页 URL
  - `Content-Type`：指示主体中资源的媒体类型或数据编码方案

- **主体 (Body)**：当请求方法为 **POST** 或 **PUT** 时携带数据

响应

HTTP 响应 (Response)

HTTP 响应由**状态行 (Status Line)**、**头部 (Headers)** 和**主体 (Body)** 组成

- **状态行**：包含 HTTP 版本、**状态码**和**原因短语** (Reason-Phrase)，如 `404 Not Found`
  - `200–299`：成功响应

- `400-499`：客户端错误
- `500-599`：服务器错误

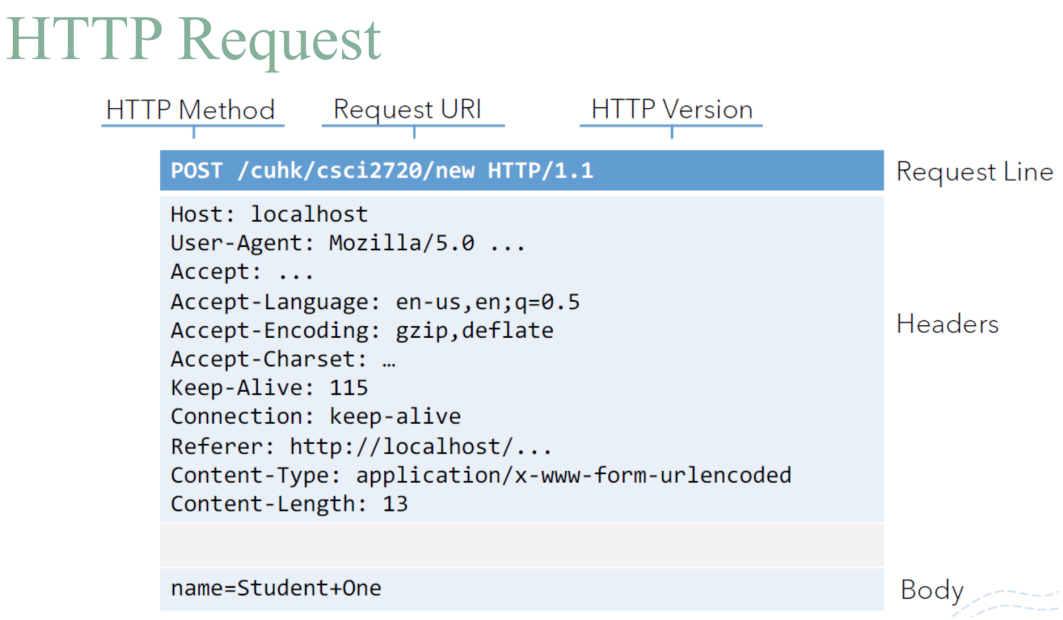
- **主体 (Body)**：包含请求的资源，可以是静态或动态生成的文件或编码数据

- **头部 (Headers)**：
  - `Location`：指示重定向页面的 URL
  - `Cache-Control`，`Expires`，`ETag`：与缓存相关的头部

  - `Set-Cookie`：向客户端发送 Cookie

  - `Content-Type`：指示主体中资源的媒体类型

  - `Content-Disposition`：请求 Web 客户端保存 (而不是显示) 内容



Lecture 14 NodeJS & Express

Node.js 是一个JavaScript运行时环境.

- 在服务器端运行JavaScript.
- 非阻塞 I/O

Express 是 Node.js 的一个模块，是一个最小且最灵活的 Web 应用程序框架.

14.1 请求-响应

Web 服务器的请求-响应循环：

- Routing（路由）：根据 URL 和 HTTP 方法决定采取的操作.
- 从 HTTP 请求中检索数据.
- 处理数据.
- 生成 HTTP 响应.

14.2 Express 框架

Express 允许您执行以下操作：

- 定义路由表：将请求 URL 和 HTTP 方法映射到操作.
- 设置中间件来响应 HTTP 请求.
- 使用模板引擎来生成 HTML 输出.

14.3 用 npm 管理模块

Node.js 允许通过 npm 管理模块（类似 python 的库）

安装的模块作为 `node_modules` 文件夹存在于应用程序文件夹下.

例：安装 Express

```
npm init
npm install express
```

```
E:\材料\大学\大三\大三 term 1\ESTR 2106 Web 开发\Lectures\Lecture_14_NodeJS\hello_world>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (hello_world)
version: (1.0.0)
description:
entry point: (index.js)
```

注意，`npm init` 时会让你手动配置，一路按 Enter 就会使用默认规范，但是这里的入口点默认是 `index.js`，我们按照课件，手动设置为 `app.js`

运行 `npm init` 成功后，会创建一个 `package.json`

然后运行 `npm install express`，会下载一个 `node_modules`（大约 3MB，不大）和一个 `package-lock.json`

14.4 入口：app.js

在设置好 Node.js 和 Express 后，可以创建一个 app.js 作为入口点.

> `app.js` 不在浏览器中运行，而是在 Node/Express 的 Web 服务器上运行. 其中 Node.js 是 JavaScript 的运行时环境，Express 是 Node.js 的一个模块/框架.
>
> 如果是在电脑上用 `npm app.js` 启动服务器，则 `app.js` 在电脑上通过 Node.js 运行时环境执行，电脑扮演 Web 服务器的角色.
>
> 交互方式：
>
> 1. 在浏览器地址栏输入 `https://localhost:3000`
> 2. 浏览器向电脑上的 Node.js 服务器发送一个 HTTP 请求
> 3. Node.js/Express 服务器接收到请求，根据路由和业务逻辑（`app.js` 中的代码）进行处理.
> 4. 服务器生成一个 HTTP 响应（通常是 HTML、JSON 或其他数据）
> 5. 浏览器接收到这个响应，并将其内容显示给用户.
>
> 当响应数据包从服务器到达浏览器后，浏览器会开始解析和显示内容.
>
> 一旦服务器完成发送响应，并关闭连接（大多数传统的 HTTP 请求），这个特定的请求-响应事务就结束了.
>
> 当你接收到新的响应，浏览器会清空当前窗口的内容，并显示新的响应内容，上一次的内容就消失了. 浏览器可能会根据 HTTP 响应头将部分响应数据缓存在本地磁盘上，如果下次请求相同的资源，浏览器可能会直接使用缓存的版本，而不需要再次向服务器发送请求.
>
> Node.js 服务器通常不会在内存中保留上一次发送给特定客户端的响应内容，服务器只处理当前的请求.

> 关闭浏览器会销毁当前进程，因此内存中显示的所有响应内容（页面状态、DOM 结构）都会被清除. 如果关闭浏览器，服务器端的 Node.js 进程本身不会受到影响，它会继续运行，监听新的连接，并等待其他客户端/其他浏览器窗口发送请求.
>
> 当你使用 `node app.js` 命令启动 Express 应用程序时，Node.js 进程会一直运行，监听指定的端口（如 `3000`），等待接收来自客户端的 HTTP 请求.
>
> 在运行 Node.js 进程的终端或命令行窗口中，需要执行 `Ctrl + C` 来停止服务器，或者直接关闭命令行窗口. 把 `Node.js` 的运行当作正常程序即可.

注意，app.js 最好和 `node_modules`、`package.json` 放在同一个应用程序根目录下.

原因：当你在 `app.js` 中使用 `require('express')` 时，Node.js 默认从 `app.js` 所在的目录向上查找 `node_modules` 文件夹. 将它们放在一起可以确保模块能被正确找到和加载.

> 注意，引入包是 require，不是 request

启动服务器：`node app.js`

> 这里 app.js 填实际设置的入口.

最简单的代码来实现服务器启动：

```
const express = require('express')
const app = express();
app.listen(3000, ()=>{
    console.log("启动服务器端口3000");
});
```

但是它处理不了任何请求. 屏幕上显示：

```
Cannot GET /
```

窗口标签显示 `Error`

这是 Express 服务器返回的一个标准错误信息. 因为运行代码成功启动服务器，但是它没有告诉 express（`app` 示例）收到请求后应该做什么. 当你尝试在浏览器中访问 `http://localhost:3000` 时，浏览器默认使用 `GET` 方法，请求根路径 `/`，Express 服务器收到了这个 `GET /` 请求，然后检查内部路由表，发现没有任何规则与 `GET /` 匹配. 在 Express 中，如果一个请求没有匹配任何定义的路由，它会默认返回一个 `404 Not Found` 错误，并附带 `Cannot GET /` 的提示信息作为响应体.

这里屏幕不显示404 Not Found是不是因为这个甚至都要设置一个res.send才能实现. 否则只能显示 Cannot GET /. 如果您想显示更友好的 `"404 Not Found"` 页面，您需要**手动定义**一个路由来捕获所有未被处理的请求，并设置自定义的状态码和响应内容，例如在所有精确路由的最后设置一条通配的路由，来捕获所有未被处理的请求.

14.5 Express 定义路由表

例：route_test/index.js

```
const express = require('express')
const app = express();

app.get('/path1', function (req, res) {
    res.send('You made a GET path1 request');
})

app.get('/path2', function (req, res) {
    res.send('You made a GET path2 request');
});

app.all('/*', function (req, res) {
    res.send('You made a request');
});

app.get('/path3', function (req, res) {
    res.send('You will not see this');
});

app.listen(3000, ()=>{
    console.log("启动服务器端口3000");
});
```

`node index.js` 启动服务器后，浏览器访问 `http://localhost:3000`，页面显示：

```
You made a request
```

访问 `http://localhost:3000/path1`，页面显示

```
You made a GET path1 request
```

访问 `http://localhost:3000/path3`，页面显示：

```
You made a request
```

因为 express 是按顺序匹配，path3会先被通配符 * 匹配掉，没有机会使用 /path3 的路由

注意，过新的 express 版本可能功能有区别. 建议安装 4.x 版本：

```
npm uninstall express
npm install express@4
```

14.5.1 通配符路由

上例中，

```
app.all('/*', function (req, res) {
    res.send('You made a request');
});
```

是通配符路由（Catch-all Routes），如果它不放在最后，会吞噬掉后续所有请求.

14.5.2 高级路由匹配

```
// Regular expression matching: e.g., any path that ends with .jpg
// Note: The expression is not enclosed by any quotes
app.all(/.+\.jpg$/, (req, res) => res.send("You requested a JPG file"));

// Route parameters matching
// e.g., http://hostname/course/2720/lecture/6
app.all('/course/:cID/lecture/:lID', (req, res) => res.send(req.params));
// Output: {"cID":"2720", "lID":"6"}

// hyphen and dot (~ and .) are interpreted literally
// e.g., http://hostname/csci2720-t2
app.all('/:course-:tutorial', (req, res) => res.send(req.params));
// Output: {"course":"csci2720", "tutorial":"t2"}
```

解释:

```
app.all(/.+\.jpg$/, (req, res) => res.send("You requested a JPG file"));
```

正则表达式匹配，不需要引号

```
app.all('/course/:cID/lecture/:lID', (req, res) => res.send(req.params));
```

路由参数匹配. 访问 `http://localhost:3000/course/1/lecture/1` 时，屏幕输出:

```
{"cID":"1","lID":"1"}
```

```
app.all('/:course-:tutorial', (req, res) => res.send(req.params));
```

路由参数与特殊字符：Express 的路由路径解析器会**字面解释**连字符（`-`）和点（`.`）等字符

用途： 这允许开发者在 URL 中使用这些分隔符来创建更具可读性的 URL 结构，同时仍然可以通过 `req.params` 捕获到参数值

输出： 如果访问 `http://hostname/csci2720-t2`，`req.params` 将返回 `{ "course":"csci2720", "tutorial":"t2" }`

这种方法常用于处理具有一致 URL 结构内的值

14.5.3 res.send()

当你在 Express 中调用 `res.send(string)` 时，Express 会做以下两件关键的事情：

- 设置 Content-Type 响应标头. 例如，当你发送一个不含任何 HTML 标签的普通字符串时，Express 通常会将 `Content-Type` 设置为 `text/html`.
- 发送响应体：`res.send(string)` 将 `string` 这个字符串作为响应的 body 发送回去.

浏览器接收到响应后，先检查 `Content-Type` 标头. 当它看到 `text/html` 时，它就知道应该把响应体中的内容视为 HTML 文档进行解析和渲染. 例如，它会把 `string` 作为 HTML 文档的内容，将其渲染在页面左上角.

除了纯文本，也可以发送 HTML 标签：

`res.send('<h1>Hello World!</h1>')`

浏览器会解析并显示一个一级标题.

以在屏幕上显示 Hello World 为例：

```
const express = require('express');
const app = express();

app.use((req, res) =>{
    res.send('Hello World!');
});

app.listen(3000, () => {
    console.log(`Express server listening at http://localhost:3000`);
});
```

解释：

```
const express = require('express');
```

引入 Express 模块：这行代码加载了 Express 模块（Express 是 Node.js 的一个模块/“add-in”）。`require()` 是 Node.js 用来管理和加载模块的方式

```
const app = express();
```

初始化 Express 应用程序：这行代码创建了一个 Express 应用程序实例，通常命名为 `app`。这个实例将用于定义路由、中间件和配置服务器

```
app.use((req, res) =>{
    res.send('Hello World!');
});
```

定义中间件（路由处理）：`app.use()` 用于加载和执行中间件函数

由于没有指定路径，这个中间件会匹配**所有**进入服务器的请求（无论路径是什么）

`req` 代表 HTTP 请求对象，`res` 代表 HTTP 响应对象

其中，`res.send('Hello World!');` 表示：

发送响应：一旦接收到任何请求，中间件立即执行 `res.send()`，生成并发送一个包含 `"Hello World!"` 的 HTTP 响应给客户端（浏览器）

因为一旦发送了响应，请求处理链就会停止，所以这个中间件**捕获了所有请求**，使它成为唯一有效的“路由"

> 一次请求只能出发一次 res.send()，一旦触发就会立即停止后续处理，然后把 response 发回客户端

注意：

```
app.listen(3000, () => {
    console.log(`Express server listening at http://localhost:3000`);
});
```

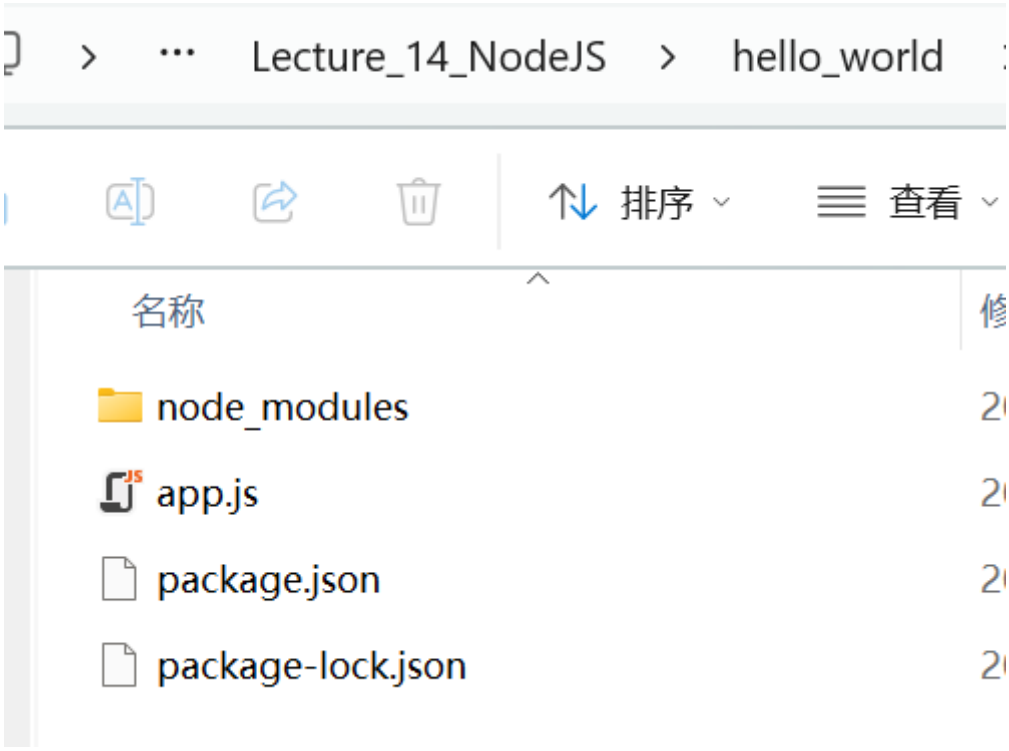
启动服务器：这行代码启动了 Express 服务器，并让它开始监听计算机上的 **3000 端口**

- 只有当服务器启动后，你才能通过浏览器访问 `http://localhost:3000`

- 箭头函数中的 `console.log` 只是在服务器启动成功后在命令行中打印一条确认信息

如果代码中缺少 `app.listen()`，那么在命令行运行 `node app.js` 基本上没有用，因为应用程序无法作为 Web 服务器工作（没有监听功能，收不到请求，且没有 `app.listen()` 的话，在执行完最后一行后立即结束）

在项目根目录下运行 `npm init`、`npm install express` 后，文件结构如下：

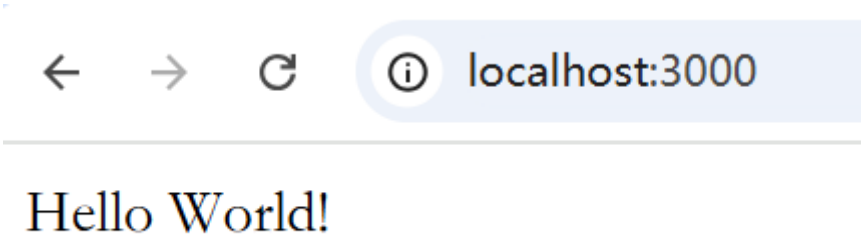


根目录下命令行运行 `node app.js`，命令行输出：

```
Express server listening at http://localhost:3000
```

然后持续保持运行状态，现在浏览器可以访问 http://localhost:3000

访问后，屏幕显示：

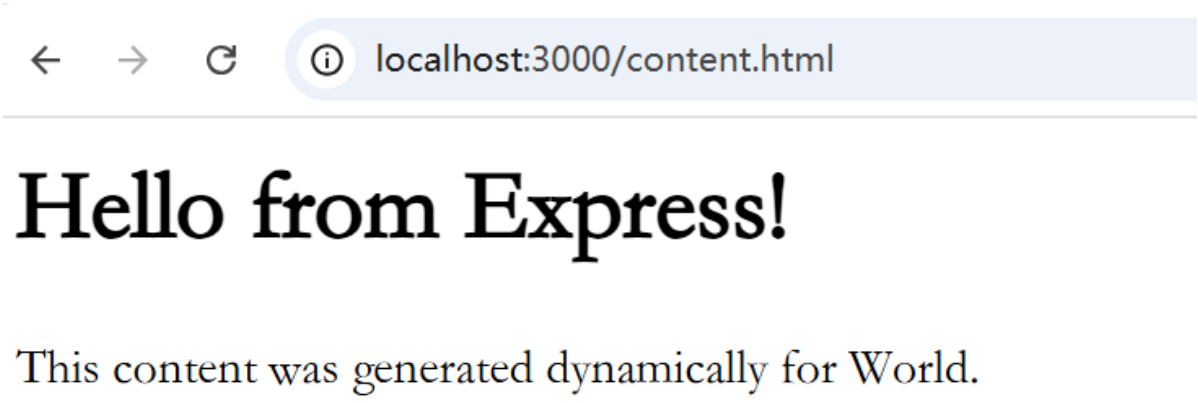


14.5.4 动态构建响应 HTML

除了响应一个简单的字符串，也可以构造一个复杂的 HTML 然后作为响应发送回去：

```
app.get('/content.html', (req, res) => {
    // 1. 创建一个字符串变量来存储要发送的内容
    var buf = '<h1>Hello from Express!</h1>';

    // 2. 可以在这里添加更多的动态内容
    // 假设我们有一个变量 name
    const name = "World";
    buf += `<p>This content was generated dynamically for ${name}.</p>`;

    // 3. 将字符串作为 HTTP 响应发送
    // 默认情况下，res.send() 会将内容视为 HTML 文件
    res.send(buf); // Note: send() can only be called once!
});
```

访问 `http://localhost:3000/content.html`，屏幕显示：



注意：根目录下并不需要存在一个 `content.html` 静态文件.

14.5.5 响应静态文件

除了动态构建文件（如 HTML）并返回，也可以响应一个根目录下已有的静态文件：

```
app.get('/', (req, res) => {
    // Send the file 'index.html' in the folder of the current script
    res.sendFile(__dirname + '/index.html');
    // __dirname holds absolute path of the folder of the current script
});
```

这样，匹配到 `/` 时（即 `http://localhost:3000`），会响应根目录下的 `index.html` 文件.
`__dirname` 是 Node.js 的全局变量，持有当前脚本文件所在文件夹的绝对路径.

注意 `/index.html` 的斜杠要加，不然会和根目录的名字黏在一起

这是一个 html 文件，`res.sendFile()` 会根据该扩展名自动设置 `Content-Type`，浏览器接收到该响应会直接渲染这个 html 文件. 等价于用浏览器直接访问这个文件.

14.6 req.query() 查询字符串

> Query String

路由表中的各规则匹配的是路径本身. 浏览器在路径之后，可以添加额外的查询字符串，它不会被算进路径里. 查询字符串通过一个问号开头来表示，问号后附加多个键值对给服务器提供信息：

```
http://hostname/search?key1=value1&key2=value2...
```

其中，浏览器请求的路径是 `http://hostname/search`. 这样的设计通常应用于搜索、过滤、分页功能，路径本身就像通往搜索/过滤/翻页功能的门，而查询字符串就像是告诉这个功能具体要做什么（搜索什么、过滤什么、翻到第几页）的纸条.

服务器处理 Query String 的代码：

```
// Handle GET request to /search?mykey=some_value
app.get('/search', (req, res) => {
    var keyword = req.query['mykey'];

    if (keyword === undefined || keyword === '')
        res.send('No keyword specified');
    else
        res.send('The keyword is ' + keyword);
});
```

核心：`req.query['mykey'];` 返回 key 对应的 value，是字符串.

浏览器访问

```
http://localhost:3000/search?mykey=1
```

显示

```
The keyword is 1
```

浏览器访问

```
http://localhost:3000/search
```

显示

```
No keyword specified
```

浏览器访问

```
http://localhost:3000/
```

显示

```
Cannot GET /
```

> 因为这里没有对根目录 request 进行路由处理

多个键值对可以用 `&` 符号连接：

```
http://localhost:3000/search?mykey=1&mykey=2
```

显示：

```
The keyword is 1,2
```

> 同一个键传入多个值，req.query() 返回字符串数组.

```
http://localhost:3000/search?mykey=1&yourkey=2
```

> 多个键值对，可以用多个 req.query() 分别查询不同 key 对应的值.

14.7 req.params() 和 req.query()

`req.params()` 是 `14.5.2 高级路由匹配` 里的路由参数匹配. recall:

```
app.all('/course/:cID/lecture/:lID', (req, res) => res.send(req.params));
```

路由参数匹配. 访问 `http://localhost:3000/course/1/lecture/1` 时，屏幕输出：

```
{"cID":"1","lID":"1"}
```

`req.query()` 是查询字符串.

对比：

	req.params	req.query
Example URL	`/test/123`	`/test?mykey=123&yourkey=456`
Example data access	`app.get('/test/:keyid', (req, res) => res.send(req.params.keyid))`	`app.get('/test', (req, res) => res.send(req.query.mykey))`
Usage	Handle values within a consistent URL structure	Searching items within URL parameters

最佳实践:

通常建议使用 `req.params` 来处理资源的**唯一标识符 (ID)** 或 URL 结构中不可或缺的部分，因为这提供了更简洁、更具语义的 URL

使用 `req.query` 来处理**可选的、影响结果集**的参数，例如搜索关键词、排序方式或分页

14.8 POST 请求的参数解析

前面所讲的参数提取，都是针对 GET 请求（浏览器地址栏输入网址并回车，默认使用 GET）.

浏览器发出 POST 请求主要用两种常见方式:

- 开发者创建一个 HTML 表单（`<form>`），并设置其 `method` 属性为 `POST`，`action` 属性设置为请求的目标 URL.
- 使用 JavaScript API（如 fetch）

以表单为例：`post`

这是根目录下的 `index.html`

```
<!DOCTYPE html>

<html>
    <head>
        <title>form post test</title>
    </head>

    <body>
        <form method="POST" action="/login">
            <input type="text" name="loginid">
            <input type="password" name="passwd">
```

```
            <button type="submit">登录</button>
        </form>
    </body>

</html>
```

注意，表单的 `name` 属性是用来提供给 Post 的 Body 的.

这是根目录下的 `index.js`:

```
const express = require('express')
const app = express()

app.use(express.urlencoded({ extended: true }));
// 全局的中间件，可以解析POST Body

app.get("/", (req, res)=>{
    res.sendFile(__dirname + "/index.html");
});

app.post('/login', (req,res)=>{
    let id = req.body['loginid'], pwd = req.body['passwd'];
    res.send('Login success. Your id is ' + id + ' and password is ' + pwd);
});

app.listen(3000, ()=>{
    console.log("监听3000")
});
```

解释：用户首先浏览根目录 `localhost:3000`，然后 `index.js` 路由并响应一个 `index.html` 给用户.

里面有一个表格，用户填写账号密码并点击登录按钮. 表单会被提交给 `action` 属性定义的 URL，即当前路径下的 `/login` . 这个提交动作就是一个 Post Request. `index.js` 先使用一个全局中间件:

```
app.use(express.urlencoded({ extended: true }));
```

把 Post Body 解析，然后就可以在路由

```
app.post('/login', (req,res)=>{
    let id = req.body['loginid'], pwd = req.body['passwd'];
    res.send('Login success. Your id is ' + id + ' and password is ' + pwd);
});
```

中进一步解析 body 的内容并根据这些内容来自定义响应. 注意，必须先对 body 进行解析，否则 body 为未定义. 最终，用户点击之后会立即出现新界面，上面显示:

```
Login success. Your id is ... and password is ...
```

省略号是用户填写的内容.

14.9 解析请求头 & 设置响应头

> Retrieving request headers

14.9.1 解析请求头

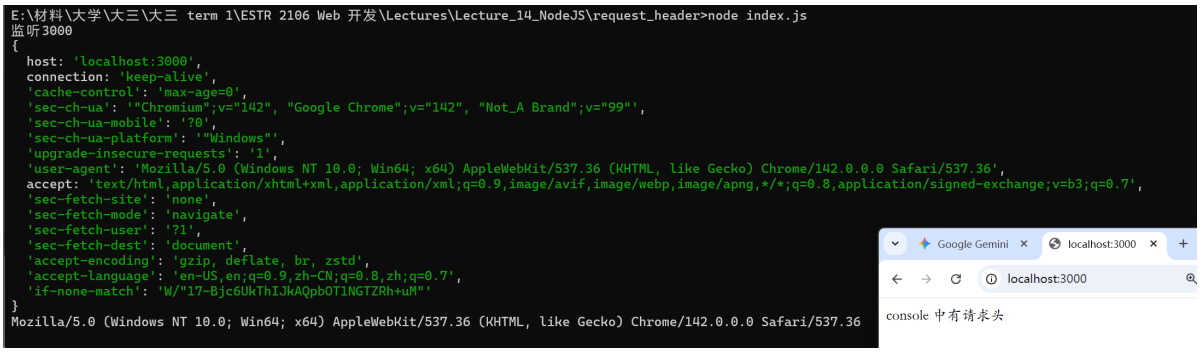
底层解析由 Node.js 完成，我们考虑的是 HTTP 请求被转化为结构化数据后，如何解析其中的头部数据（body 数据的解析前面都讲了）.

```
// HTTP Request Header contains info about a client,
// info about the content embedded in the body, cookies, and more...
app.get('/*', (req, res) => {
    // Header fields in the request found as properties in req.headers
    console.log(req.headers);

    // Helper function to get the value of a specific header with header
    // name case-insensitive; returns undefined if it does not exist
    console.log(req.get('user-agent'));
});
```

例：`index.js` 中代码如下

```
const express = require('express')
const app = express()

app.get('/*', (req, res) => {
    console.log(req.headers);
    console.log(req.get('user-agent'));
    res.send("console 中有请求头")
});

app.listen(3000,()=>{
    console.log("监听3000");
});
```

访问 `localhost:3000` 得到



注意，这里 `console.log(req.headers);` 是在服务器端运行的，因此会在运行 Node.js 的命令行输出，而不是在客户端（浏览器）的 console 输出. 浏览器会显示 `console 中有请求头`，因为这个是发送到客户端的响应. 浏览器的 console 里什么都不会出现.

14.9.2 设置响应头

重点在 `Content-Type` 的设置：

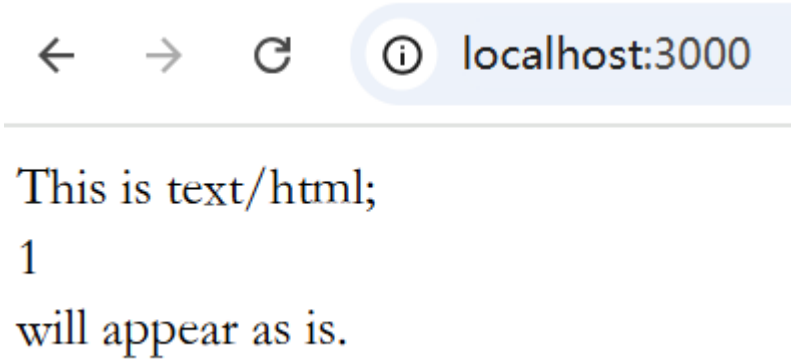
```
// HTTP Response Header contains info about a server,
// info about the content embedded in the cookies, and more...
app.get('/*', (req, res) => {
    var buf = 'This is plain text; "<br>" will appear as is.\n';

    res.set('Content-Type', 'text/plain');

    // Note: Headers can only be set before any output is sent
    res.send(buf);
});
```

例：

```
const express = require('express')
const app = express()


app.get('/*', (req, res) => {
    var buf = 'This is text/html; <br>1</br> will appear as is.\n';

    res.set('Content-Type', 'text/html');

    // Note: Headers can only be set before any output is sent
    res.send(buf);
});

app.listen(3000, ()=>{
    console.log("监听3000")
```

```
})
```

显示为：

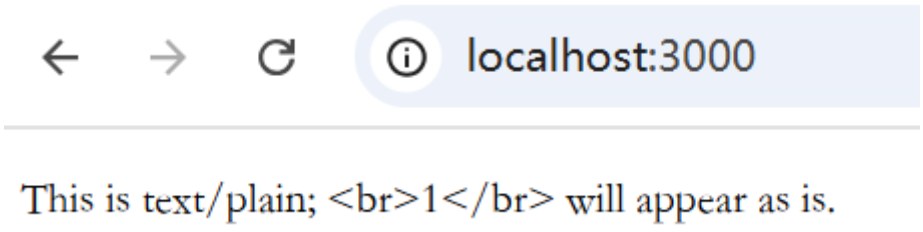


解释：如果设置响应头的 `Content-Type` 为 `text/html`（这个是默认值，不设置就是这个值），则按照 html 的规范来渲染.

例：

```
const express = require('express')
const app = express()


app.get('/*', (req, res) => {
    var buf = 'This is text/plain; <br>1</br> will appear as is.\n';

    res.set('Content-Type', 'text/plain');

    // Note: Headers can only be set before any output is sent
    res.send(buf);
});

app.listen(3000, ()=>{
    console.log("监听3000")
})
```

显示为



解释：如果 `Content-Type` 设置为 `text/plain`，则会把它当作纯文本显示给用户（浏览器可以渲染纯文本文件，如 .txt）.

14.10* 中间件

> Middleware and routingEx

Express 应用程序本质上是一系列中间件调用，它们按照定义的顺序依次执行.

中间件是一个函数，形如：

```
function (req, res, next){...}
```

- `req` 表示当前 HTTP 请求对象
- `res` 表示当前 HTTP 响应对象
- `next` 一个回调函数，如果当前中间件完成了自己的任务，并且希望将控制权传递给流水线中的下一个中间件或路由处理器，它就会调用 `next()`

中间件的作用是响应 HTTP 请求，在请求到达最终路由处理器之前，对请求进行处理或修改. 常见的功能包括：

- 数据解析：读取请求主体（Body）中的内容，并将其解析到 `req.body` 对象上（如解析 POST 表单数据）
- 服务文件：为客户端提供静态文件.
- 请求日志：记录请求信息（IP、时间、路径）以便调试
- 权限检查：检查用户是否已登录或具有执行操作的权限.
- 数据预处理：验证或清理请求中携带的数据.

中间件：

```
// 引入 Express 模块
const express = require('express');
const app = express();

// 定义一个自定义中间件：requestTime
const requestTime = (req, res, next) => {
  // 修改 req 对象，添加一个属性 req.myTime  28]
  req.myTime = new Date().toLocaleDateString('en-GB');  39]
  // 调用 next()，将控制权交给栈中的下一个中间件或路由  30, 40]
  next();  40]
};

// 应用 requestTime 中间件到所有路由  42]
```

```
app.use('/*', requestTime);  42]

// 定义一个路由，访问 req.myTime 属性
app.get('/', (req, res) => {
  console.log('请求时间记录: ' + req.myTime);  45]
  res.send('Hello World! 请求时间已记录。');  46]
});

// app.listen(3000);
```

Lecture 15 Cookie & Session

HTTP 协议是无状态的，这意味着服务器无法自动记住不同请求之间的用户身份或上下文. 为了解决这个问题，Web 应用程序需要一种机制来记住状态，例如：

- 用户是否登录
- 当前用户是谁

“无状态” 的意思是，服务器处理每一个 HTTP 请求视为一个全新的、独立的事务. HTTP 服务器不知道请求的身份，只是一个接一个地处理请求. 服务器不会自动记住用户是谁、用户是否已登录、用户在上一个请求中做了什么.

因此要引入一些新的协议和 HTTP 配合.

15.1 Cookies

> 超文本传输协议 Cookie

Cookie 是服务器端脚本发送给 Web 客户端的一小段数据，用于在一段时间内保持持久性.

工作原理：Cookies 嵌入在 HTTP 头部中. 浏览器将它们存储在客户端设备上，并在随后的每个 HTTP 请求中自动发送回服务器.

客户端发送给服务器的 Cookie 包含在 HTTP 请求头中，服务器发给客户端的 Cookie 包含在响应头中.

典型用途：

- 个性化：记住用户偏好
- 会话管理：保存一个会话 ID 来识别用户
- 跟踪：记住用户在一个网站上的活动，或通过第三方 Cookie 跨网站跟踪用户.

缺点：

- 客户端可以篡改 Cookies，因为它们是 plain text file.
- 浏览器窗口和标签页共享同一组 Cookies.
- 每个服务器的 Cookie 数量和大小都有限制.
- 每个请求都会增加 HTTP 请求头部的大小.
- 隐私泄露（切勿在 Cookies 中存储敏感数据！）

15.1.1 Cookie 属性

Expires：Cookie 过期的时间. 未指定则为会话结束时（浏览器关闭时）

Max-Age：Cookie 过期的毫秒（milliesecond）数. 比 Expires 优先.

Domain：指定 Cookie 应发送到的域. 如果指定，子域也包括在内.

Path：指定 URL 路径，只有当请求资源包含此路径时，Cookie 头才会被发送.

HttpOnly：Cookie 只能在服务器端访问.

Signed：Cookie 附加签名，服务器端脚本可以检测是否被客户端修改.

15.2 Session

Session 是一种将多个 HTTP 请求关联起来的典型方法，例如，确定请求是否来自同一客户端的同一用户.

工作原理：

- 服务器为每个用户生成一个唯一的会话 ID（Session ID）
- **会话 ID 保存在客户端**（通常存储在 Cookie 或查询字符串中）
- 服务器为每个创建的会话 ID 创建一个**服务器端存储空间**.
- 后续请求中的会话 ID 允许服务器端脚本共享这个存储空间中的数据，例如登录状态.

15.3 客户端存储

`window.localStorage` 和 `window.sessionStorage` 允许 Web 应用程序通过 JavaScript 在浏览器内存储数据 。

- **特点:**
  - 允许至少 **5MB** 的存储空间
  - 存储的数据**不会**传输到服务器
  - 来自同一源的所有页面都可以访问相同存储中的数据
  - 数据以名称-值对的形式存储，值必须是字符串或 JSON 编码的字符串

- **类型区分:**
  - `sessionStorage`: 在浏览器关闭时过期
  - `localStorage`: 永久保留

备考

Q1 HTML (8 marks)

(1) Which of the following tags is NOT a valid HTML5 element?

A `<fieldset>`

B `<footer>`

C `<select>`

D `<grid>`

答案：D.

解析：不要和 Bootstrap 的网格系统弄混.

A `<fieldset>` 在 Lecture 6 表单中介绍；

B `<footer>` 是 HTML 页面底部的脚注；

C `<select>` 在 Lecture 6 表单中介绍.

(2) Which of the following statements about HTML is INCORRECT?

A: Without using other tags, you cannot create consecutive whitespace inside a `<p>` tag.

B: Semantic elements can help search engines understand your web page.

C: HTML can be generated by JavaScript.

D: The company's logo image usually uses the SVG format

答案：A

解析：A不正确，你可以使用 HTML 实体 ` ` (non-breaking space) 来创建连续的空格.

语义元素有助于搜索引擎理解网页;

HTML 可以由 JavaScript 生成；

网页的 favicon 通常在 `<head>` 部分定义.

(3) Which of the following statements about SVG is INCORRECT?

A: SVG can be enlarged without losing quality or sharpness.

B: For the same images, the SVG format is always smaller in size compared to the JPEG format.

C: Search engines can easily understand the text in an SVG.

D: The company's logo image usually uses the SVG format.

答案：B

SVG 是一种基于 XML 的矢量图形格式，它**放大时不会失真**，**搜索引擎容易理解其中的文本**，**常用于公司 Logo**. 然而，SVG 只适用于结构简单的图像，对于复杂的图像，其尺寸**不一定**比 JPEG 格式小.

(4) Consider this code snippet.

```
<ol type="square">
  <li>Red</li>
  <li>Green</li>
  <li>Blue</li>
</ol>
```

Which of the following statements best describes the rendered page?

A: Nothing will be displayed as `<ol>` does not accept attribute `type="square"`

B: An ordered list of colors with numbers from 1 to 3.

C: An ordered list of colors with letters from “a” to “c”.

D: An unordered list of colors with square markers.

答案：B

解析：`<ol>` 标签是**有序列表** (Ordered List). 虽然 `type="square"` 是一个无效或不推荐的属性值（`square` 是 `<ul/>` 无序列表的标记类型），但浏览器会忽略无效的 `type` 属性，并使用默认的设置，即数字编号.

Q2 CSS and Bootstrap (20 marks)

(1) 解释内联元素 (inline elements) 和块级元素 (block-level elements) 的区别.

答案：

Block-level: occupy full width

Inline: only occupy width of their content

Block-level: always start at a new line

Inline: can start anywhere

(2) 给出两个内联元素的例子

答案：`<i>`（斜体）、`<a>`（链接）、`<img>`（图像）等.

(3) 给出两个块级元素的例子

答案：`<p>`（段落）、`<h>` 标题、`<table>`（表格）等.

(4) CSS 选择器和样式覆盖 (Override)

Suppose we have the following header in our HTML body.

```
<h1 id="abc">This is a heading</h1>
```

4.1) Using the id selector, write down the CSS rules to change the above element into an inline element with a yellow color in an internal style sheet.

```
<style>
#abc {
    display: inline;
    color: yellow;
}
</style>
```

4.2) Without modifying the above internal style sheet, describe TWO different methods to override the yellow color with a green color.

法一：使用内联样式（Inline Style）

```
<h1 id="abc" style="color: green">This is a
heading</h1>
```

内联样式优先级最高.

法二：使用 `!important` 关键字

In an **external style sheet**, use `!important` to override the internal style sheet

```
#abc { color: green !important; }
```

(5) Consider the grid system in Bootstrap. The following code snippet is demonstrated in one of our lectures.

```
<div class="container bg-warning">
  <div class="row">
    <div class="col-sm-1 col-xl-4 bg-primary"> Col 1</div>
    <div class="col-sm-2 col-xl-3 bg-secondary"> Col 2</div>
    <div class="col-sm-3 col-xl-2 bg-danger"> Col 3</div>
    <div class="col-sm-4 col-xl-1 bg-info"> Col 4</div>
  </div>
</div>
```

Suppose the following is currently being displayed in the browser.

(1) A student claims that the yellow area (i.e., the area on the righthand side of Col 4) has the exact same width compared to the grey area (i.e., Col 2). Do you agree? Explain your answer briefly.

答案：

The student is correct.

Bootstrap grid system total columns = 12

Current display = 1+2+3+4 = 10

So yellow background color = 12-10 = 2 = width of Col 2

(2) What will happen if I zoom out to a level where the "xl" breakpoint is reached?

答案：The current ratio for “Col 1:Col 2:Col 3:Col 4:background” is “1:2:3:4:2” according to the above display screen. When “xl” is reached, the ratio will turn into 4:3:2:1:2.

(3) What will happen if I continue to zoom in and eventually reach a very small screen size (e.g., a screen smaller than 576 pixels in width)? Hint: Bootstrap provides an automatic responsive behaviour for such a small screen size

答案：All col 1,2,3,4 will stack up. Each (i.e., each col) will occupy the entire row. The background color will disappear.

Q3. JavaScript (17 marks)

(1) 正则表达式

In our lecture, we discussed the following regular expression to check a standard email address:

```
const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

Suppose we want to check if the address is our university staff email account. How would you modify the regular expression? Write down your answer below. i.e., it must start with one or more characters that are not whitespace or the @ symbol, followed by the @ symbol, and then it must be cuhk.edu.hk

答案：

```
const regex = /^[^\s@]+@cuhk\.edu\.hk$/;
```

(2) `typeof`

Suppose I run the following code snippet in the console. What will be the output?

```
let midterm = ['100 pts', 'very easy']
typeof midterm
```

答案：

```
'object'
```

解析：在 JavaScript 中，数组 (Array) 的类型是 `object`

(3) In our lecture, we mentioned that JavaScript often has a floating-point error. For example, the following code snippet will output "false", which doesn't make sense.

```
if (0.1 + 0.2 == 0.3){
    console.log("true")
}
else{
    console.log("false")
}
```

Suggest one way you can bypass this floating-point error in JavaScript to compare 0.1+0.2 and 0.3 correctly. Write down your solution code

答案：

```
let x = 0.0000001 // any small number is fine
if (0.1+0.2-0.3 < x){
    console.log('true')
}
else{
    console.log('false')
}
```

使用一个很小的容忍度进行比较.

(4) Consider the following function that returns the n-th Fibonacci number:

```
function f(n){
    if (n<=1) return n;
    return f(n-1) + f(n-2);
}
```

Is it possible to write it in a single-line arrow function? If yes, write down the code. If no, write down the multiple-line arrow function.

答案：yes

```
f = (n) => n <= 1 ? n : f(n-1) + f(n-2);
```

> 三元运算符

(5) Write down the output of this code snippet:

```
function change(arr, num) {
    arr.push(4);
    num = 2;
}
let a = [1, 2, 3];
let b = 1;
change(a, b);
console.log(b);
console.log(a);
```

1

[1, 2 ,3, 4]

(6) Suppose I want to attach an on-click event to a button to invoke an `alert()` function. In the lecture, a common mistake was mentioned:

```
document.querySelector('#btn1').onclick = alert("hi");
```

Write down the correct code (use the same ID and alert message):

```
document.querySelector("#btn1").onclick = ()
=> alert("hi");
```

(7) Write down the output of this code snippet:

```
let c = [1,2,3];
c.forEach(item=>item+1);
console.log(c);

let d = [1,2,3];
d.forEach((item, i, d) => d[i]+=1);
console.log(d);
```

答案:

```
[1, 2 ,3]
[2, 3, 4]
```

Q4. Form and Promise (12 marks)

Q5. ReactJS (13 marks)

Q6. Node.JS (14 marks)

Consider this Node.JS file. Assume the server runs on http://localhost:3000.

```
const express = require('express');
const app = express();

app.all('/loc/:lid-:eid', (req, res) => {
    let str = 'Location ID: ' + req.params.lid;
    str += '<br>';
    str += 'Event ID: ' + req.params.eid;
    res.send(str);
});

app.get('/midterm', (req, res) => {
    res.send('your score is ' + req.query.myscore);
});

app.all('/*', (req, res) => res.send('Welcome!'));

app.get('/', (req, res) => res.send('Location?'));

const server = app.listen(3000);
```

(1) What will be displayed on the browser screen when accessing this Express site via the following URL?

URL: `http://localhost:3000/loc/csci-2720`

答案:

```
Location ID: csci
Event ID: 2720
```

(2) Suppose the URL is now:

URL: `http://localhost:3000/loc`

What will be displayed on the browser screen?

答案:

```
Welcome!
```

(3) Suppose the browser screen displayed this:

> your score is 100

What should be the corresponding URL?

答案：

```
http://localhost:3000/midterm?myscore=100
```

(4) HTTP Cookies 问题

Consider this code snippet.

```
res.cookie('cookie', '123', {
    maxAge: '100'
});
```

Answer the following questions:

(A) Write down the piece of data being stored as a cookie.

答案：A key-value pair is stored. The **Key** is `'cookie'` and the **value** is `'123'`.

(B) Where will it be stored? Client or server?

答案：response，存在 client.

(C) What is the meaning of the properties maxAge: '100'

答案：100 milliseconds 后 be deleted.

(D) Suppose I added an expires property to this cookie. Is it necessary to delete the maxAge property if I want the cookie to expire on a specific date?

答案：Yes, it must be deleted as maxAge has precedence over expires

(E)Assume that the cookie with an expiration date is implemented successfully and there is no further server-side action after the cookie is stored. Suggest ONE reason why the cookie is gone before the expiration date.

答案：被 User manually delete 了

Q7. T/F 判断题 (16 marks)

(1) The original purpose of HTML is to share and format scientific information among researchers.

T

(2) Suppose we have two CSS rules. Both of them apply to the same element (say, the tag) and are written in the same external sheet. The earlier one will override the later one

F. 后写的覆盖先写的

(3) JavaScript is multi-threaded

F. 单线程

(4) A Promise object always require one successful callback and a failure callback.

F. 可以没有 failure callback

(5) To start a web server using npm, we use the command `npm begin`

F. `npm start`

(6) Session keeps temporary data on the server side

T

(7) Suppose we have an element with id='123'. The return of getElementbyID('123') and querySelector('#123') are NOT exactly the same

T

(8) The client can find cookies in the HTTP response body

F. It should be **head** instead of body.

Project

ddl: 12.18 23:59