HTML 是构成 Web 世界的一砖一瓦，它定义了网页内容的含义和结构。其基本结构是如下的标签以及内容：

```
<opening_tag attribute_name="value">
    content
</closing_tag>
```

`<html>`

包含所有网页内容的标签

- `lang="zh-CN"`

`<head>`

包含了网页的头部数据，例如标题，引入 CSS 等

标题

```
<title>Title of page<title/>
```

元数据

文字编码

```
<meta charset="utf-8" />
```

作者和描述

- `author`：作者
- `description`：描述，用于搜索引擎等

```
<meta name="author" content="Van" />
<meta name="description" content="This is a site." />
```

图标

```
<link rel="icon" href="favicon.ico" type="image/x-icon" />
```

CSS 和 JavaScript

```
<link rel="stylesheet" href="a.css" />
<script src="a.js" defer></script>
```

`<body>`

文本

语义元素

- `<p>`：段落
- `<h1>`~`<h6>`：标题
- `<ul>`：无序列表
- `<ol>`：有序列表
  - 列表元素用 `<li>` 包裹
  - 可嵌套列表
- `<strong>`：粗体强调
- `<em>`：斜体强调

表象元素

无语义

- `<b>`：粗体
- `<i>`：斜体
- `<u>`：下划线

链接

```
<a href="test.html#article" title="jump">
    <p>jump<p/>
<a/>
<p id="article">
    here.
<p/>
```

可以包含块级内容，如图片等

- `href`：指向的链接
  - `mailto:xxx@xxx.com`：邮件
  - `xxx#id`：文档片段链接（在相应的板块加入 `id="id_name"`）
- `title`：鼠标悬浮时显示的信息
- `download`：（下载文件）默认保存文件名

其他文本

- `<dl>`：描述列表
  - `<dt>`：术语
  - `<dd>`：描述
  - 一个术语可以有多个描述
- `<blockquote>`：块级引用。浏览器会缩进
- `<q>`：行内引用
  - `cite=：用URL指向引用的资源
- `<cite>`：引用元素，应使用 `<a>` 链接到引用源。默认为*斜体*
- `<abbr>`：缩略
  - `title`：完整解释（鼠标悬浮显示）
- `<sup>`，`<sub>`：上标与下标
- 代码：
  - `<code>`：用于标记计算机通用代码。
  - `<pre>`：用于保留空白字符（通常用于代码块）——如果文本中使用了缩进或多余的空白，浏览器将忽略它，你将不会在呈现的页面上看到它。但是，如果你将文本包含在 `<pre></pre>` 标签中，那么空白将会以与你在文本编辑器中看到的相同的方式渲染出来。
  - `<var>`：用于标记具体变量名。
  - `<kbd>`：用于标记输入电脑的键盘（或其他类型）输入。
  - `<samp>`：用于标记计算机程序的输出。
- `time`：日期
  - `datetime`

文档架构

结构

- `<header>`：页眉
- `<nav>`：导航
- `<form>`：搜索栏
- `<main>`：主内容
  - `<article>`
  - `<section>`
  - `<aside>`：侧边栏
- `<footer>`：页脚

无语义元素

- `<span>`：内联
- `<div>`：块级

分隔

- `<br>`：换行
- `<hr>`：水平线

多媒体

图片

```
<img src="test.png" alt="A picture" />
```

- `src`：图片地址
- `alt`：替换文字
- `width`，`height`：图片未加载时也可以生效

视频

```
<video src="test.mp4" controls>
    <p>Can't play.</p> //无法播放的后备内容
</video>
```

- `controls`：使用浏览器自带的播放器
- `width`，`height`：不拉伸
- `autoplay`
- `loop`
- `muted`
- `poster="xx.jpg"`
- `preload=`：缓冲
    - `none`
    - `auto`：页面加载后才缓冲
    - `metadata`：仅缓冲元数据
- 提供多种格式：

```
<video controls>
 <source src="van.mp4" type="video/mp4" />
 <source src="van.webm" type="video/webm" />
</video>
```

Iframe 嵌入

```
<iframe
    src="//player.bilibili.com/player.html?
isOutside=true&aid=19390801&bvid=BV1bW411n7fY&cid=31621681&p=1"
    scrolling="no"
    border="0"
```

```
    frameborder="no"
    framespacing="0"
    allowfullscreen>
</iframe>
```

一些常用属性：

- `src`
- `sandbox`：尚不清楚，总之安全性
- `allowfullscreen`
- `border: none`

Embed 与 object 嵌入

属性	`<embed>`	`<object>`
嵌入内容的 URL	`src`	`data`
嵌入内容的*准确*媒体类型	`type`	`type`
由插件控制的盒子高度和宽度（以 CSS 像素为单位）	`height`，`width`	`height`，`width`
名称和值，作为参数提供给插件	具有这些名称和值的 ad hoc 属性	单标签 `<param>` 元素，包含在 `<object>` 元素里面
用作后备资源的独立的 HTML 内容，以防资源不可用	不受支持（`<noembed>` 已过时）	包含在 `<object>` 中，在 `<param>` 元素之后

使用例：

```
<object data="mypdf.pdf" type="application/pdf" width="800" height="1200">
  <p>
    You don't have a PDF plugin, but you can
    <a href="mypdf.pdf">download the PDF file. </a>
  </p>
</object>
```

表格

- `<tr>`：存放一行的元素
- `<td>`：表格的一个 cell，`<th>`：同性质，只是作为表头，应用不一样的样式
  - `colspan`，`rowspan`：拓展 cell 大小
- `<colgroup>`：统一制定列样式，写在 `<table>` 的开头（其他的地方也可，毕竟是 HTML

```
<colgroup>
  <col /> // 无样式的列需要这样跳过
```

```
  <col style="background-color: yellow" span="2" />
</colgroup>
```

- `<thead>`, `<tbody>`, `<tfoot>`：结构化表格，方便应用 CSS

CSS

CSS 是一门基于规则的语言，可以通过指定应用于网页上特定元素或元素组的样式组，来定义规则。他它的基本结构如下：

```
selector {
    attribute : value;
}
```

Selector 选择器

类型

```
h1 {
}
```

类

```
.highlight.class {
}
```

- 匹配含所有指定类的元素，不加空格地写入类名

ID

```
#unique {
}
```

属性

选择器	描述	注
`a[attr]`	含有某属性	
`a[attr=value]`	含有某属性，其值特定	
`a[attr~=value]`	含有某属性，且属性中有此值	`class="A B"` 即有两个值
`a[attr|=value]`	含有某属性，其值以 `value-` 为开头，或正为 `value`	注意连字符
`li[attr^=value]`	开头	

选择器	描述	注
`li[attr$=value]`	结尾	
`li[attr*=value]`	属性出现字段	`class="AB"` 可以这样匹配到 `A` 或 `B`
`a[attr=value i]`	忽略大小写	

伪类，伪元素

伪类以冒号开头：

```
a:first-child{
  font-size: 150%;
}
```

一些常用伪类：

- `:first-child`
- `:last-child`
- `:only-child`
- `:hover`：指针悬浮
- `:focus`：键盘选定
  伪元素以双冒号开头：

```
article p::first-line {
  font-size: 120%;
  font-weight: bold;
}
```

一些伪元素：

- `::after`：原有元素的实际内容之后的第一个可样式化元素
- `::before`
- `::first-line`
- `::first-letter`
- `::selection`：被选中部分

关系

- 所有后代：空格

```
article p
```

- 直接子代：`>`

```
body > h1
```

- 邻接：`+`，同级的元素旁边

```
h1 + p
```

匹配紧邻标题的段落

- 通用兄弟：`~`，此元素之后的所有同级元素

```
h1 ~ p
```

此标题下的所有段落

层叠、优先级与继承

层叠

同级规则应用于同一元素，总是后者生效

继承

子代继承父亲的一部分样式，其中 `width`、`margin`、`padding` 和 `border` 不继承
设置样式时，可用以下属性值：

- `inherit`：继承父类
- `initial`：初始值
- `revert`：重置为默认值
- `revert-layer`
- `unset`：自然值，属性是自然继承则取 `inherit`，否则取 `initial`

优先级

样式冲突时参考：`!important` > 内联 > ID > 类 > 元素

- 此优先级具有累加性，累加对应匹配次数
- 具有 `!important` 声明的规则会覆盖其他规则
- 内联，即在元素标签内的 `style`

层叠层

层叠层的优先级 #待补充

创建层叠层，以及添加样式

```
@layer site;
@layer site {
    font-size: 120%;
}
@layer {
    background-color: yellow;
}
@layer page {
    color: green;
}
body {
    color: green
}
```

- 创建了具名层、匿名层与未分层的样式

导入层叠层

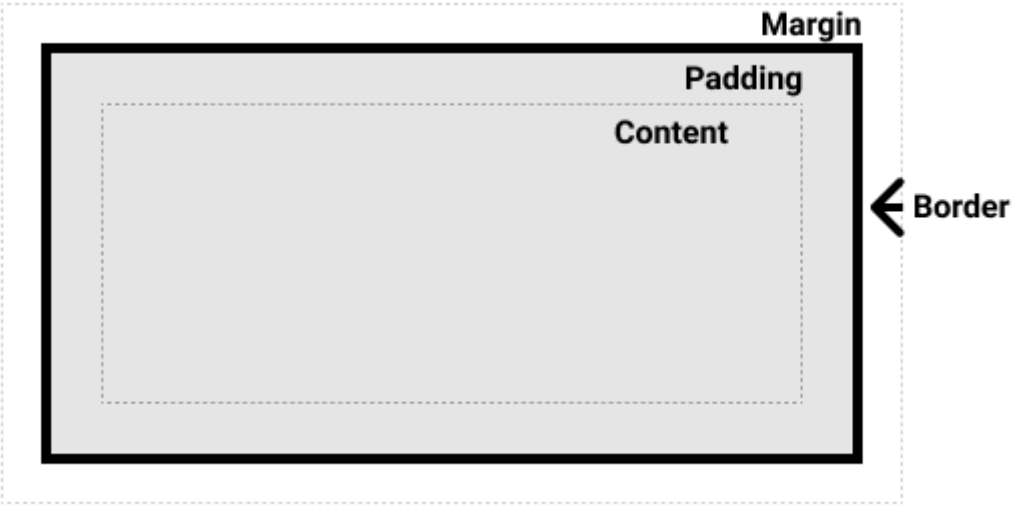
```
@import url("test.css") layer(page);
@import url("yum.css") layer(page);
```

- 须在任何样式之前

嵌套层

```
@layer site.wide {
    color: black;
}
```

Box Model



把 HTML 元素看作盒子
盒子有**区块盒子**与**行内盒子**，**内部显示**与**外部显示**

```
.block {
    display: block;
}
```

JavaScript

JavaScript 是一门跨平台、面向对象的脚本语言，它能使网页可交互（例如，拥有复杂的动画、可点击的按钮、弹出菜单等）。

数据结构

JS 的数据结构具有以下特征：

- 弱类型
- 动态类型

函数

```
function mine(argu1, argu2 = 10) {
}
```

- 在函数后加括号即立刻调用，在添加事件监听器等时注意
- 可选参数加 `=`

匿名函数

先看一个原型：`event` 见事件对象

```
function logKey(event) {
  console.log(`You pressed "${event.key}".`);
}

textBox.addEventListener("keydown", logKey);
```

匿名函数用于传递函数给另一函数，而不需先声明

```
textBox.addEventListener("keydown", function (event) {
  console.log(`You pressed "${event.key}".`);
});
```

箭头函数

传递**匿名函数**的一种方法，用 `(event) =>` 代替了 `function(event)`，若有多个参数，只需添加在括号内

```
textBox.addEventListener("keydown", (event) =>
  console.log(`You pressed "${event.key}".`),
);
```

对只有一行 `return` 语句的函数，可以：

```
const originals = [1, 2, 3];
const doubled = originals.map(item => item * 2);
```

闭包

闭包（closure）是一个函数以及其捆绑的周边环境状态（**lexical environment**，**词法环境**）的引用的组合，随着函数的创建而创建，而不随函数作用域结束而消灭

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 1
```

- 相当于将 5 和 10 保存在了 `add5` 和 `add10` 中

事件

一些常用事件：

- `click`
- `focus`：聚焦
- `blur`：失焦
- `dblclick`
- `mouseover`
- `mouseout`
  最好不要使用内联事件处理器

添加与移除事件监听器

```
btn.addEventListener("click", function, { signal: controller.signal });
btn.removeEventListener("click", function);
// or
controller.abort();
```

- 可添加多个

事件处理器

```
btn.onclick = funciton;
```

- 会覆盖之前的

事件对象

事件对象是自动传递给函数的 `event` 对象

```
function bgChange(event) {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  e.target.style.backgroundColor = rndCol;
  console.log(event);
}

btn.addEventListener("click", bgChange);
```

属性与额外属性

```
textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
```

```
});
```

事件 `keydown` 会创造 `KeyboardEvent` 对象，其有一个 `key` 属性，是触发事件时按下的键
类似的， `video` 有 `play()` 方法

阻止默认行为

```
  e.preventDefault();
```

如表单验证

冒泡

最先触发点击的元素的事件，然后依次向外触发其相应父元素事件
例如，点击了 `div` 中的 `button`，会触发 `button` 的 `click` 事件后触发 `div` 的 `click`

- 阻止冒泡：

```
event.stopPropagation();
```

捕获

```
document.body.addEventListener("click", handleClick, { capture: true });
container.addEventListener("click", handleClick, { capture: true });
button.addEventListener("click", handleClick);
```

这样则先在最小嵌套元素（最外层）触发

委托

```
container.addEventListener("click", (event) => {
  event.target.style.backgroundColor = bgChange();
});
```

这样，子元素的事件先冒泡到父元素，再由父元素通过 `event.target` 获取目标元素

- `event.currentTarget` 则访问处理该事件的元素，上面的例子即父元素

对象

```
const objectName = {
  member1Name: member1Value,
  member2Name: member2Value,
  speak() {
```

```
    console.log("114514");
  },
};
```

包含**属性**与**方法**

- 对象的属性也可以是对象
- 点表示法：`object.member.smallerMember`，不能接受变量，只能直接写出
- 括号表示法：`object["member"]["smallerMember"]`，可以接受变量作为属性名
- 可以动态地添加成员
- 用 `this` 指定当前代码运行时的对象

构造函数

```
function Person(name) {
    const obj = {};
    obj.name = name;
    obj.introduceSelf = function () {
        console.log(`你好！我是 ${this.name}。`);
    };
    return obj;
}
```

- 一般，构造函数以其创建的对象名称命名，且首字母大写
- 要使用 `new` 调用构造函数：

```
const van = new Person("van");
```

对象原型

所有对象拥有一个内置属性，称为 **prototype**。这也是一个对象，故其拥有原型，构成**原型链**，终于以 `null` 为原型的对象。
访问对象属性时，如果找不到，则向其原型查找，直到末端返回 `undefined`

- `Object.getPrototypeOf()` 返回了原型
- `Object.prototype` 是所有原型的基础
  属性遮蔽：同名属性存在，则不会向原型查找，造成了遮蔽

设置原型

1. `Object.create()` 方法

```
const personPrototype = {
  greet() {
```

```
    console.log("hello!");
  },
};

const carl = Object.create(personPrototype);
```

2. 构造函数

```
const personPrototype = {
  greet() {
    console.log(`你好，我的名字是 ${this.name}! `);
  },
};

function Person(name) {
  this.name = name;
}

Object.assign(Person.prototype, personPrototype);
const reuben = new Person("Reuben");
reuben.greet(); // 你好，我的名字是 Reuben!
```

- 其中，`name` 为**自有属性**，即不从原型中继承

类

使用 `class` 关键字声明：

```
class Person {
    name;

    constructor(name) {
        this.name = name;
    }

    introduceSelf() {
        console.log(`Hi! I'm ${this.name}`);
    }
}
```

- 使用 `constructor` 关键字声明构造函数，可以省略
- 调用构造函数时仍使用类名，如

```
const van = new Person("van");
```

继承

```
class Student extends Person {
    age;

    consturctor(name, age) {
        super(name);
        this.age = age;
    }

    introduceSelf() {
        console.log(`114514! I'm ${this.name} of ${this.age}`);
    }
}
```

- 使用 `super()` 调用父类的构造函数，并传递 `name`
- 覆盖 `introduceSelf()` 函数

封装

```
class Student extends Person {
    #age;

    consturctor(name, age) {
        super(name);
        this.#age = age;
    }

    introduceSelf() {
        console.log(`114514! I'm ${this.name} of ${this.#age}`);
    }

    #speak() {
        console.log("1919810!");
    }
}
```

- 完成了 `age` 和 `speak()` 的封装

异步

Promise

Promise 是一个由异步函数返回的对象，指示操作当前的状态

- 不能像操作一般对象一样操作 Promise 对象：

  × Failure

```
Const promise = fetchProducts ();
Console.Log (promise[0]. Name);
```

- 而应使用 `then()` 方法：

✓ Success

```
Const promise = fetchProducts ();
Promise.Then ((data) => console.Log (data[0]. Name));
```

Promise 具有三种状态：

- `pending`：待定
- `fulfilled`：兑现，此时调用 `then()`
- `rejected`：拒绝，此时调用 `catch()`
  后两种状态统称为 `settled`（敲定）
  若 Promise 被敲定，或被锁定以跟随另一个 Promise，称 `resolved`（解决）

Promise 的链式使用

当有多个异步函数，且开始下个函数前要完成前一个函数，使用 Promise 链
注意 `then()` 方法仍然返回 Promise 对象

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
);

fetchPromise
  .then((response) => response.json())
  .then((data) => {
    console.log(data[0].name);
  });
```

合并 Promise

当所有 Promise 都兑现，且不相互依赖，使用 `promise.all()`，其接受 Promise 数组，返回单一 Promise：

- 当且仅当数组中*所有*的 Promise 都被兑现时，才会通知 `then()` 处理函数并提供一个包含所有响应的数组，数组中响应的顺序与被传入 `all()` 的 Promise 的顺序相同。
- 如果数组中有*任何一个* Promise 被拒绝。此时，`catch()` 处理函数被调用，并提供被拒绝的 Promise 所抛出的错误。
  例子：

```
const fetchPromise1 = fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
    "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
    .then((responses) => {
        for (const response of responses) {
            console.log(`${response.url}: ${response.status}`);
        }
    })
    .catch((error) => {
        console.error(`获取失败: ${error}`);
    });
```

异步函数

使用 `async` 关键字:

```
async function test() {
 // do some thing
}
```

在一个返回 Promise 的函数之前使用 `await` 就可以使代码在此等待，直到 Promise 完成:

```
const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
```

- 只能对 `aysnc` 函数使用 `await`

`fetch()`

```
const fetchPromise = fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
      );

console.log(fetchPromise);

fetchPromise.then((response) => {
    console.log(`已收到响应：${response.status}`);
});

console.log("已发送请求……");
```

`fetch` 函数返回一个 `response` 对象，其有一些属性：

- `status`：状态码
- `statusText`：状态信息
- `ok`：请求是否成功
- `headers`：包含头信息的对象
  还有一些方法：
- `json()`：转化为 JSON
- `text()`
- `blob()`